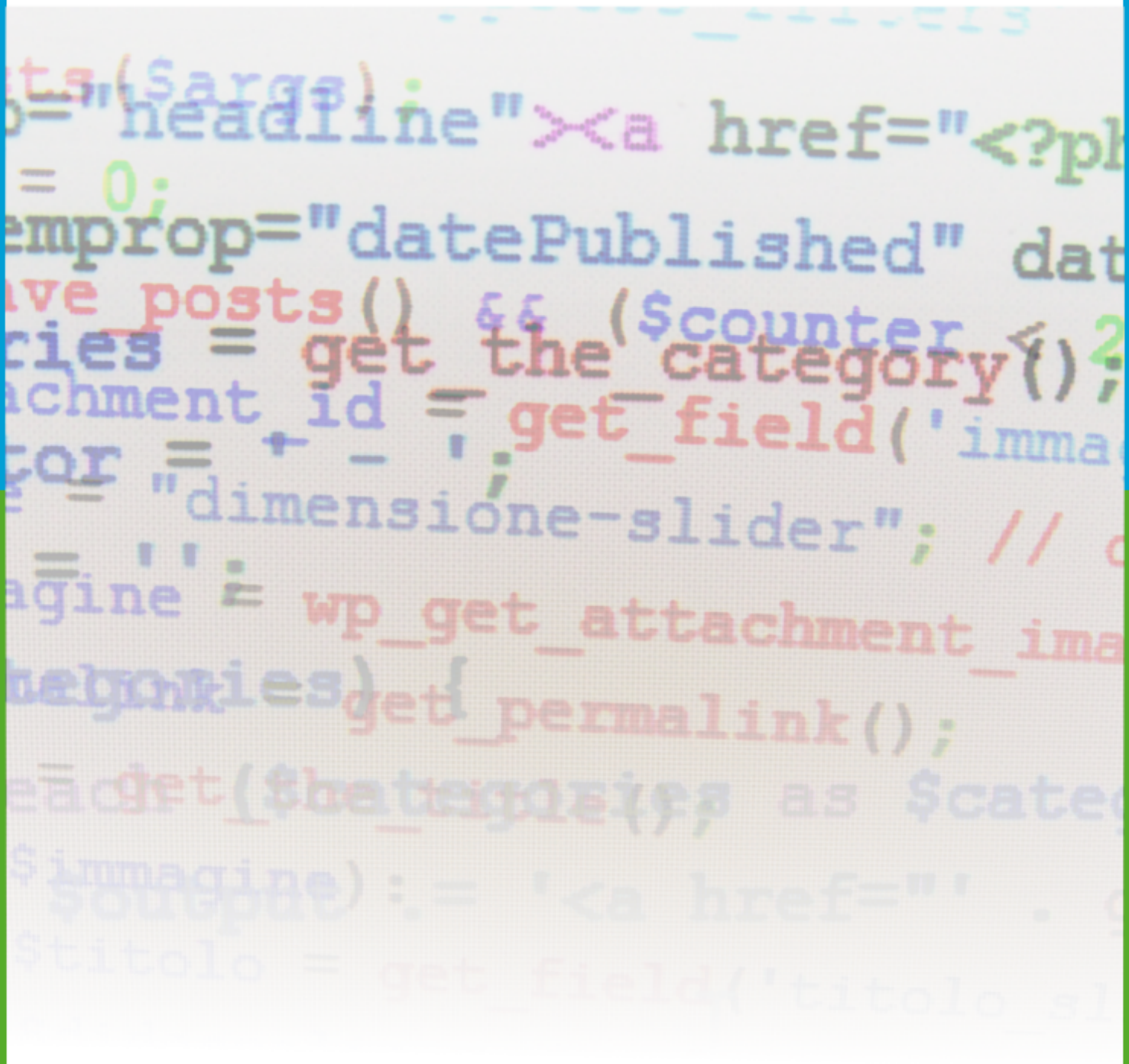
Proyecto LATIn
INICIATIVA LATINOAMERICANA DE LIBROS DE TEXTO ABIERTOS

ALFA
FINANCED BY THE EUROPEAN UNION

Estructuras de datos



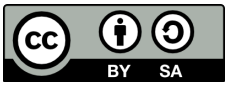
AUTORES

José Fager
W. Libardo Pantoja Yépez
Marisol Villacrés
Luz Andrea Páez Martínez
Daniel Ochoa
Ernesto Cuadros-Vargas

Estructuras de Datos

1a ed. - Iniciativa Latinoamericana de Libros de Texto Abiertos (LATIn), 2014. 222 pag.

Primera Edición: Marzo 2014
Iniciativa Latinoamericana de Libros de Texto Abiertos (LATIn)
http://www.proyectolatin.org/



Este texto forma parte de la Iniciativa Latinoamericana de Libros de Texto abiertos (LATIn), proyecto financiado por la Unión Europea en el marco de su Programa ALFA III EuropeAid.

El Proyecto LATIn está conformado por: Escuela Superior Politécnica del Litoral, Ecuador (ESPOL); Universidad Autónoma de Aguascalientes, México (UAA), Universidad Católica de San Pablo, Perú (UCSP); Universidade Presbiteriana Mackenzie, Brasil(UPM); Universidad de la República, Uruguay (UdelaR); Universidad Nacional de Rosario, Argentina(UR); Universidad Central de Venezuela, Venezuela (UCV), Universidad Austral de Chile, Chile (UACH), Universidad del Cauca, Colombia (UNICAUCA), Katholieke Universiteit Leuven, Bélgica (KUL), Universidad de Alcalá, España (UAH), Université Paul Sabatier, Francia (UPS).

Índice general

1 — Introducción a las Estructuras de Datos

1.1 Conceptos básicos sobre estructuras de datos

1.1.1 Definición

En un lenguaje de programación, un **tipo de dato** esta definido por el conjunto de valores que representa y por el conjunto de operaciones que se pueden realizar con dicho tipo de dato. Por ejemplo, el tipo de dato entero en Java puede representar números en el rango de -2^31 a 2^31-1 y cuenta con operaciones como suma, resta, multiplicación, división, etc.

Por otro lado, podemos decir que en la solución de un problema a ser procesado por un computador podemos encontrar dos grandes tipos de datos: datos simples y datos estructurados. Los **datos simples** son aquellos que, al ser representados por el computador, ocupan solo una casilla de memoria. Debido a esto, una variable de un tipo de dato simple hace referencia a un único valor a la vez. Ejemplo de estos tipos de datos son los enteros, reales, caracteres y booleanos.

Así mismo, los **datos estructurados** se caracterizan por que su definición está compuesta de otros tipos de datos simples, así como de otros datos estructurados. En este caso, un nombre (identificador de la variable estructurada) hace referencia no solo a una casilla de memoria, sino a un grupo de casillas.

En programación, el término **estructura de datos** se utiliza para referirse a una forma de organizar un conjunto de datos que se relacionan entre si, sean estos simples o estructurados, con el objetivo de facilitar su manipulación y de operarlo como un todo.

Por otro lado, tenemos el término **Tipo de Dato Abstracto**, o TDA, que es muy comunmente utilizado como equivalente al término estructura de datos para referirse justamente a un tipo de dato estructurado que representa un concepto a través de la definición de sus características(datos que lo conforman) y de sus operaciones(algoritmos que manipulan los datos que lo conforman)

1.1.2 Operaciones

Sobre una estructura de datos se puede efectuar diferentes tipos de operaciones, entre las más importantes están:

- *Inserción*. Es aquella mediante la cual se incluye un nuevo elemento en la estructura.
- *Modificación*. Permite variar parcial o totalmente el contenido de la información de los elementos de la estructura.
- *Eliminación*. Como su nombre lo indica, es la que permite suprimir elementos de la estructura.
- *Navegar por la estructura*: Esta es una operación básica que garantiza que se puede recuperar información almacenada.
- *Búsqueda*. Permite determinar si un elemento se encuentra o no en la estructura.
- *Consulta de la información*. Permite obtener información de uno o más elementos de la estructura.

- *Copia parcial o total*: Mediante esta operación se puede obtener total o parcialmente una estructura con características similares a la original.
- *Prueba*. Permite determinar si uno o varios elementos cumplen determinadas condiciones.
- *Verificar si es vacía*. Permite determinar si existen o no elementos sobre la estructura.

1.2 Clasificación

Una clasificación de estructuras de datos es según dónde residan: Internas y externas. Si una estructura de datos reside en la*memoria central* del computador se denomina *estructura de datos interna*. Recíprocamente, si reside en un soporte *externo*, se denomina *estructura de datos externa*.

Las estructuras de datos internas pueden ser de dos tipos:

- Estructuras de Datos Estáticas.
- Estructuras de Datos Dinámicas.

1.2.1 Estructuras de Datos Estáticas

Tienen un número fijo de elementos que queda determinado desde la declaración de la estructura en el comienzo del programa. Ejemplo los arreglos. Las estructuras de datos estáticas, presentan dos inconvenientes:

1. La reorganización de sus elementos, si ésta implica mucho movimiento puede ser muy costosa. Ejemplo: insertar un dato en un arreglo ordenado.
2. Son estructuras de datos estáticas, es decir, el tamaño ocupado en memoria es fijo, el arreglo podría llenarse y si se crea un arreglo de tamaño grande se estaría desperdiciando memoria.

1.2.2 Estructuras de Datos Dinámicas

Las estructuras de datos dinámicas nos permiten lograr un importante objetivo de la programación orientada a objetos: la reutilización de objetos. Al contrario de un arreglo, que contiene espacio para almacenar un número fijo de elementos, una estructura dinámica de datos se amplía y contrae durante la ejecución del programa.

A su vez, este tipo de estructuras se pueden dividir en dos grandes grupos según la forma en la cual se ordenan sus elementos.

- Lineales
- No lineales

Estructuras de Datos Lineales

En este tipo de estructuras los elementos se encuentran ubicados secuencialmente. Al ser dinámica, su composición varía a lo largo de la ejecución del programa que lo utiliza a través de operaciones de inserción y eliminación. Dependiendo del tipo de acceso a la secuencia, haremos la siguiente distinción:

- Listas: podemos acceder (insertar y eliminar) por cualquier lado.
- Pilas: sólo tienen un único punto de acceso fijo a través del cual se añaden, se eliminan o se consultan elementos.
- Colas: tienen dos puntos de acceso, uno para añadir y el otro para consultar o eliminar elementos.

Estructuras de Datos No Lineales

Dentro de las estructuras de datos no lineales tenemos los árboles y grafos. Este tipo de estructuras los datos no se encuentran ubicados secuencialmente. Permiten resolver problemas computacionales complejos.

2 — Introducción al Diseño y Análisis de Algoritmos

El diseño de la solución a un problema que nos lleve a la generación de un algoritmo tiene dos principales componentes: la identificación de los datos que intervienen en la solución con su respectiva especificación de diseño y la especificación de las instrucciones a ejecutar para resolver el problema.

Los datos son los valores que manejamos en la resolución de un problema, tanto los valores de entrada, como los de proceso y los de salida. Es decir, los datos son información y por lo tanto, para manejarlos se requieren varios tipos de datos.

Un tipo de dato se puede definir como un conjunto de valores y un conjunto de operaciones definidas por esos valores. Clasificar los datos en distintos tipos aporta muchas ventajas, como por ejemplo indicarle al compilador la cantidad de memoria que debe reservar para cada instancia dependiendo del tipo de dato al que pertenezca. Los tipos de datos abstractos van todavía más allá; extienden la función de un tipo de dato ocultando la implementación de las operaciones definidas por el usuario. Esta capacidad de ocultamiento permite desarrollar software reutilizable y extensible.

Por otro lado, un algoritmo es sinónimo de procedimiento computacional y es fundamental para las ciencias de la computación. Un algoritmo es una secuencia finita de instrucciones, cada cual con un significado concreto y cuya ejecución genera un tiempo finito. Un algoritmo debe terminar en un tiempo finito.

Algoritmo es toda receta, proceso, rutina, método, etc. que además de ser un conjunto de instrucciones que resuelven un determinado problema, cumple las siguientes condiciones:

- Ser finito. La ejecución de un algoritmo acaba en un tiempo finito; un procedimiento que falle en la propiedad de la finitud es simplemente un procedimiento de cálculo.
- Ser preciso. Cada instrucción de un algoritmo debe ser precisa; deberá tenerse en cuenta un rigor y no la ambigüedad, esta condición es la definibilidad: cada frase tiene un significado concreto.
- Posee entradas. Las entradas se toman como un conjunto específico de valores que inicializan el algoritmo.
- Posee salida. Todo algoritmo posee una o más salidas; la salida es la transformación de la entrada.
- Ser eficaz. Un algoritmo es eficaz cuando resuelve el problema.
- Ser eficiente. Un algoritmo es eficiente cuando resuelve el problema de la mejor manera posible, o sea utilizando la mínima cantidad de recursos.

Este capítulo pretende cubrir estas dos grandes areas: la relevancia de los datos y sus tipos en la solución de un problema, y el análisis de eficiencia de las instrucciones que se eligen para resolver un problema.

2.1 Tipos de Datos

En el diseño de un algoritmo, y su paso de algoritmo a programa, se manejan constantemente conceptos como tipo de datos, estructura de datos y tipo de datos abstracto. A pesar de que suenan similares, es extremadamente relevante entender la diferencia entre estos términos.

En un lenguaje de programación, el **tipo de dato de una variable** esta definido por el conjunto de valores que una variable puede tomar y por el conjunto de operaciones que se pueden realizar con y sobre dicha variable. Por ejemplo, el tipo de dato entero en Java puede representar números en el rango de -2^31 a 2^31-1 y cuenta con operaciones como suma, resta, multiplicación, división, etc.

Por otro lado, podemos decir que en la solución de un problema a ser procesado por un computador usualmente se encuentran dos grandes tipos de datos: datos simples y datos estructurados. Los **datos simples** son aquellos que, al ser representados por el computador, ocupan solo una casilla de memoria. Debido a esto, una variable de un tipo de dato simple hace referencia a un único valor a la vez. Ejemplo de estos tipos de datos son los enteros, reales, caracteres y booleanos.

Así mismo, los **datos estructurados** se caracterizan por que su definición está compuesta de otros tipos de datos simples, así como de otros datos estructurados. En este caso, un nombre (identificador de la variable estructurada) hace referencia no solo a una casilla de memoria, sino a un grupo de casillas. Un ejemplo de esto es cuando definimos un arreglo de 10 enteros, el arreglo sería un dato estructurado. Otro ejemplo sería la definición del tipo de dato estructurado Fecha, el cual estaría compuesto por un entero para el definir el día, otro para el mes y finalmente uno más para definir el año de la fecha.

Por otro lado, tenemos el término **Tipo de Dato Abstracto**, o TDA, que muy comunmente se relaciona con los tipos de datos estructurados. Un TDA permite representa un concepto a través de la definición de sus características(datos que lo conforman) y de sus operaciones(algoritmos que manipulan los datos que lo conforman). La implementación de un TDA en un lenguaje de programación usualmente se sirve de un tipo de dato estructurado para la definición de los datos que lo conforman. Por ejemplo, si, quisieramos crear un modelo matemático de las fechas que nos rodean, tendríamos que comenzar definiendo las características o datos que conforman una Fecha: un mes, un día y un año, todos números enteros. Luego, deberíamos definir las operaciones que se pueden realizar con las Fechas: dos fechas se pueden restar, dando como resultado el número de días entre ellas, a una fecha se le puede sumar un número de días, dando como resultado otra fecha, etc. Para representar la definición de los datos de la Fecha usuaríamos el tipo de dato estructurado Fecha, que definimos en el ejemplo anterior.

Ahora, usualmente en programación se utiliza el término **estructura de datos** para referirse a la forma concreta en que la descripción lógica de un TDA será implementada. Un ejemplo de esto sería el TDA Lista que se refiere a figura lógica de un conjunto de datos que, en teoría, puede crecer indefinidamente, al cual se le pueden agregar y eliminar elementos sin mayores restricciones. El TDA Lista puede ser implementado por diferentes estructuras de de datos, como por ejemplo, por una lista enlazada o por un arreglo.

2.2 Tipos de Datos Abstractos

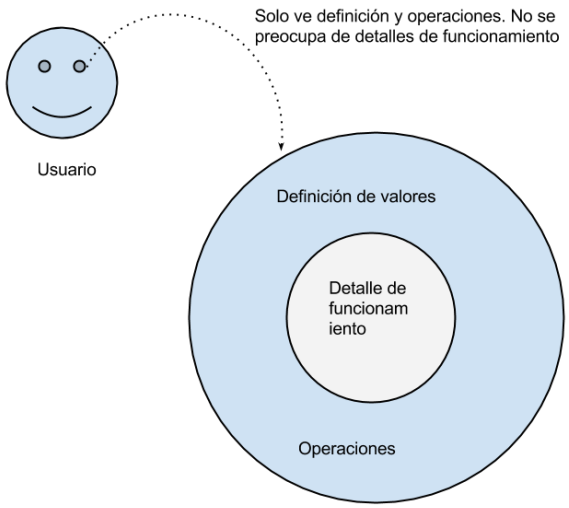
Como ya se revisó, un TDA se refiere al concepto matemático básico que define a un tipo de dato. Están formados por los datos (estructuras de datos) y las operaciones (procedimientos o funciones) que se realizan sobre esos datos.

TDA = Valores +operaciones

La definición de un TDA habilita al diseñador de una solución a imaginar variables de este tipo de dato. Estas variables, sin embargo, ya no serían consideradas variables de tipos de datos simples, como un entero. Serían consideradas entidades abstractas llamadas **objetos**. El crear variables de un TDA se conoce como **instanciar** un TDA. Cada objeto de un TDA sería una instancia del mismo, con su propia identidad única. Ejemplo:

Si existiese el TDA Fecha, podemos crear dos instancias del mismo: Fecha a y Fecha b. a y b son objetos de tipo Fecha: ambos con día, mes y año y con la capacidad de realizar todas las operaciones definidas para el TDA Fecha. Sin embargo, cada uno tendrá sus propios valores de día, mes y año y la ejecución de sus operaciones ser realizará con dichos valores únicos: Si se ejecuta un aumento de los días de la Fecha a, esta operación no afectará los días de la Fecha b.

Los TDAs son diseñados bajo los principios de **abstracción**, **encapsulamiento** y **modularidad**. Por el principio de abstracción, el TDA es diseñado para resaltar su definición y operaciones, y esconder los detalles de su funcionamiento. Por ejemplo, si se definiese el TDA Polinomio, su diseño estaría enfocado en resaltar que un Polinonio es un conjunto de Términos, que un Polinomio se puede sumar con otro, que se puede derivar, etc. Pero no estaría enfocado en detallar como se representará cada término, como se representará el conjunto de Términos ni como o que variables se verán afectadas cuando se ejecute la operación que deriva un Polinomio dando un nuevo Polinomio como resultado. Por el principio de **encapsulamiento**, estos detalles se encontrarán escondidos, encapsulados por lo realmente importante para el diseño: la forma en que se interactuará con el TDA.



Esto implica que, en la implementación del TDA usando un lenguaje de programación, ni las estructuras de datos que utilizamos para almacenar la representación del TDA, ni los detalles de implementación de las operaciones que lo definen, serán visibles para los usuarios programadores que usarán eventualmente el TDA. Para lograrlo, la implementación del TDA está basada en el principio de **modularidad:** En la programación modular se descompone un programa creado en pequeñas abstracciones independientes las unas de las otras, que se pueden relacionar fácilmente unas con las otras. La implementación de un TDA en un lenguaje de programación se puede descomponer de una sección de definición llamada interfaz y de una sección de implementación; mientras que en la interfaz se declaran las operaciones y los datos, la implementación contiene el

código fuente de las operaciones, las cuales permanecen ocultos al usuario.

Es por esto que, cuando vamos a diseñar un TDA es necesario tener una representación abstracta del objeto sobre el cual se va a trabajar, sin tener que recurrir a un lenguaje de programación. Esto nos va permitir crear las condiciones ,expresiones ,relaciones y operaciones de los objetos sobre los cuales vamos a trabajar.

Por ejemplo, si se va a desarrollar software para la administración de notas de una escuela, los TDAs Curso, Estudiante, Nota, Lista, etc., van a permitir expresar la solución de cualquier problema planteado, independientemente del lenguaje de programación con el cual se implementarán.

2.2.1 Especificación de un TDA

El diseño de un TDA puede ser específicado utilizando ya sea un enfoque informal, como un enfoque formal.

Especificación informal: Describe en lenguaje natural todos los datos y sus operaciones, sin aplicar conceptos matemáticos complicados para las persona que no están familiarizados con los TADs, de manera, que todas las personas que no conocen a fondo las estructura de los TADs, lo puedan entender de manera sencilla y que esas mismas personas puedan explicarlo de la misma manera natural , a todos con la misma facilidad con la que ellos lo entendieron.

Podemos representar de manera informal un TAD de la siguiente manera :

Nombre del TDA
Valores: Descripción de los posibles valores que definirán al TDA .
Operaciones: descripcion de cada operacion.

Comenzamos escribiendo el nombre del TAD, luego describimos los posibles valores de este tipo de dato de manera abstracta, sin tener que pensar en la realizacion concreta y por ultimos describiremos cada una de las operaciones creadas para el TAD

A continuación, definiremos de manera informal un TDA sencillo.

Ejemplo: Especificar de manera informal un TDA Vector.

//nombre del TDA
Vector
//Valores
Conjunto de elementos todos del mismo tipo.
//Operaciones
Crear Vector
Asignar un elemento al Vector .
Obtener número de elementos del Vector .

Especificación Formal: Una de las ventajas de especificar formalmente el diseño de un TDA es que permite la posibilidad de simular especificaciones a través de la definición de precondiciones y postcondiciones para las operaciones de los TDAs

Tipos: nombre de los tipos de datos.
Sintaxis: Forma de las operaciones.
Semántica: Significado de las operaciones.

La sintaxis proporciona el tipo de dato de entrada como los tipos de datos de salida de las operaciones creadas , mientras que la semántica nos dice el comportamiento de las operaciones creadas en el TDA.

TAD <nombre>
<objeto abstracto>
<invariante>
<operaciones >
<operacion 1>
<operacion k-1>
...
<operacion k>: < dominio > <codominio>

<prototipo operación >
/*Explicación de la operación*/
{ precondicion : .... } /* condición logica */
{ post condicion: ... } /*condición logica */

La precondición y las postcondición mencionadas anteriormente se refieren a los elementos que componen los objetos abstractos y a los argumentos que recibe . En la especificación se debe considerar implícito en la precondición y la postcondición ,que el objeto abstracto sobre el cual se va a operar deba cumplir con el invariante .

Es importante elaborar una breve descripción de cada operación que se crea,de manera que el usuario del TDA pueda darse una rápida idea de las cosas que puede realizar el TDA, sin necesidad de entrar a un análisis detallado, por lo cual esto va dirigido en especial a los programadores.

Ejemplo: Tomaremos el ejemplo anterior para la creación del TAD de manera formal:

/*Objeto abstracto de vector */
TAD Vector[T] /* nombre del vector , donde T puede ser cualquier tipo de dato*/
{ invariante: n>0 }

/*Crea y retorna un vector de dimensión [0...fil-1, 0] inicializada en 0 */
Vector crearVector(int fil , int col , int valor)
{ pre : 0 fil = 0 col< N}
{ post : crearVector es un vector de dimensión [0...fil-1], xik = 0 }

/* Asigna a la casilla de coordenadas [fil, col] el valor val */
void asignarValorACasilaVector(Vector v,int)
{ pre: 0 fil =, 0 col < N }
{ post: X fil, col = val }

/* Retorna el contenido de la casilla de coordenadas [fil, col] */
int obtenerInfoVector(Vector v, int fil, int col)
{ pre: 0 fil =0, col < N }

{ post: infoMat = X fil, col }

/* Retorna el número de columnas del vector */
int obtenerColumVect(Vector v)
{ post: colum_Vect = N }

2.2.2 Tipos de Operaciones de los TDAs

Las operaciones de un TDA se clasifican en 3 grupos, según su función sobre el objeto abstracto:

Constructora: es la operación encargada de crear elementos del TDA. En el caso típico, es la encargada de crear el objeto abstracto más simple. Tiene la siguiente estructura:

Clase <constructora> (<argumentos>)
{ pre: }
{ post: }

En el ejemplo anterior crearVector la operación constructora del TDA Vector, pero un vector puede tener varias operaciones constructoras.

Modificadora: esta operación que puede alterar el estado de un elemento del TDA. Su misión es simular una reacción del objeto.

void <modificadora> (<objeto Abstracto>, <argumentos>)
{ pre: }
{ post: }

En el ejemplo anterior del TDA Vector creado anteriormente, la operación modificadora es asignarValorACasilaVector,que altera el contenido de una casilla del vector.

Analizadora: es una operación que no altera el estado del objeto, sino que tiene como misión consultar su estado y retornar algún tipo deinformación.

<tipo> <analizadora> (<objeto Abstracto>, <argumentos>)
{ pre: }
{ post: = función ( ) }

Existen otro tipos de operaciones que podemos agregar a un TDA como lo son :

Comparación: Es una analizadora que permite hacer calculable la noción de igualdad entre dos objetos del TDA.
Copia: Es una modificadora que permite alterar el estado de un objeto del TDA copiandolo a partir de otro.
Destrucción: Es una modificadora que se encarga de retornar el espacio de memoria dinámica ocupado por un objeto abstracto. Después de su ejecución el objeto abstracto deja de existir, y cualquier operación que se aplique sobre él va a generar un error. Sólo se debe llamar esta

operación, cuando un objeto temporal del programa ha dejado de utilizarse.
Salida a pantalla: Es una analizadora que le permite al cliente visualizar el estado de un elemento del TDA. Esta operación, que parece más asociada con la interfaz que con el modelo del mundo, puede resultar una excelente herramienta de depuración en la etapa de pruebas del TDA.
Persistencia: Son operaciones que permiten salvar/leer el estado de un objeto abstracto de algún medio de almacenamiento en memoria secundaria. Esto permite a los elementos de un TDA sobrevivir a la ejecución del programa que los utiliza.

2.2.3 Ejemplo de un TAD

Crear el TDA RACIONAL, que corresponde al concepto matemático de un número racional. Un número racional es aquel que puede expresarse como el cociente de dos números enteros. Se definen que las operaciones disponibles para este TDA serán: la creación de un número racional a partir de dos enteros, la adición, la multiplicación, una prueba de igualdad y la impresión de un número racional en pantalla. A continuación el TDA especificado de manera semi formal:

```
abstract typedef <integer, integer> RACIONAL;
condition RACIONAL[1]!=0;

/* definición del operador */
abstract RACIONAL crearRacional(a,b)
int a,b;
precondition b!=0;
postcondition crearRacional[0]==a;
                              crearRacional[1]==b;

/* a+ b */
abstract RACIONAL sumar(a,b)
RACIONAL a,b;
postcondition sumar[1]==a[1]*b[1];
                        sumar[0]==a[0]*b[1]+b[0]*a[1];

/* a*b */
abstract RACIONAL multiplicar(a,b)
RACIONAL a,b;
postcondition multiplicar[0]==a[0]*b[0];
multiplicar[1]==a[1]*b[1];

/* a==b */
abstract sonIguales(a,b)
RACIONAL a,b;
postcondition sonIguales ==(a[0]*b[1]==b[0]*a[1]);

/* imprimir(a) */
abstract imprimirRacional(a)
RACIONAL a,b;
```

2.2.3.1 Uso del TDA

Una vez que se cuenta con una definición clara del TDA, se pueden diseñar soluciones que lo utilicen. En el caso del TDA Racional, por ejemplo, podría querer resolverse el siguiente problema:

Escriba un programa que reciba 10 números racionales por teclado y que muestre en pantalla el resultado de su suma.

Un ejemplo del diseño de una solución a este problema, utilizando la definición dada y pseudocodigo sería:

```
Programa: Suma de 10 números racionales
Entorno: Enteros n,d,i
Racional A, rSuma
Algoritmo:
i = 0
rSuma = crearRacional(0,1)
mientras (i < 10)
                escribir "Introduzca el numerador y el denominador del Racional #" + i
                leer n, d
                A = crearRacional(n,d)
                rSuma = sumar(rSuma,A)
                i = i+1

finmientras
    escribir "El resultado de la suma es "
    imprimirRacional(rSuma)
Finprograma
```

Nótese que, por los principios de abstracción y encapsulamiento, en toda la solución, el usuario jamás se deberá preocupar por lo que encierra el TDA, por como se manipulan el numerador y denominador para realizar la suma de dos racionales, o para imprimir un racional.

2.2.3.2 Implementación

La implementación de un TDA en un lenguaje de programación se realiza en base a la definición del mismo. Durante la implementación, el programador sí se enfoca en los detalles de como se manipulan los datos para realizar las operaciones necesarias. En esta fase, el programador tendrá acceso a todos los datos del TDA de forma irrestricta.

En este libro, se procuraran dar ejemplos de implementación tanto en lenguaje C como en Java:

```
/***************** MODULO DE INTERFAZ ***********************************/
#ifndef _racional_H
#define _racional_H
    typedef struct Racional{
        int numerador, denominador;
    }Racional;
    Racional * racionalCrear(int n, int d);
    Racional *racionalSumar(Racional *a, Racional *b);
    Racional *racionalMultiplicar(Racional *a, Racional *b);
    int sonIguales(Racional *a, Racional *b);
```

```
    void racionalImprimir(Racional *a, Racional *b);
#endif /* _racional_H */

/***************** MODULO DE IMPLEMENTACION ****************************/
#include <stdlib.h>
#include "racional.h"

Racional * racionalCrear(int n, int d){
    Racional *nuevo = NULL;
    if(d != 0){
        nuevo = (Racional *)malloc(sizeof(Racional));
        nuevo->numerador = n;
        nuevo->denominador = d;
    }
    return nuevo;
}
Racional *racionalSumar(Racional *a, Racional *b){
    int nr, dr;
    Racional *r;
    nr = a->numerador*b->denominador + b->numerador*a->denominador;
    dr = a->denominador * b->denominador;
    r = racionalCrear(nr, dr);
    return r;
}
Racional *racionalMultiplicar(Racional *a, Racional *b){
    int nr, dr;
    Racional *r;
    nr = a->numerador*b->numerador;
    dr = a->denominador * b->denominador;
    r = racionalCrear(nr, dr);
    return r;
}
int sonIguales(Racional *a, Racional *b){
    return (a->numerador*b->denominador==b->numerador*a->denominador);
}
void racionalImprimir(Racional *a){
    printf("%d/%d\n", a->numerador, a->denominador);
}

/***************** MODULO DE USO ***********************************/
#include <stdlib.h>
#include "racional.h"

void main(){
    Racional *A, *rsuma;
    int i=0, d, n;
    rsuma = racionalCrear(0,1);
    while(i<0){
        ("Ingrese el numerador y el denominador del Racional #%d:", i);
```

```
        scanf("%d%d", &n, &d);
        A = racionalCrear(n,d);
        rsuma = racionalSumar(rsuma, A);
        i++;
    }
    printf("El resultado de la suma es:");
    racionalImprimir(rsuma);
}
```

2.2.4 Ejercicios Propuestos

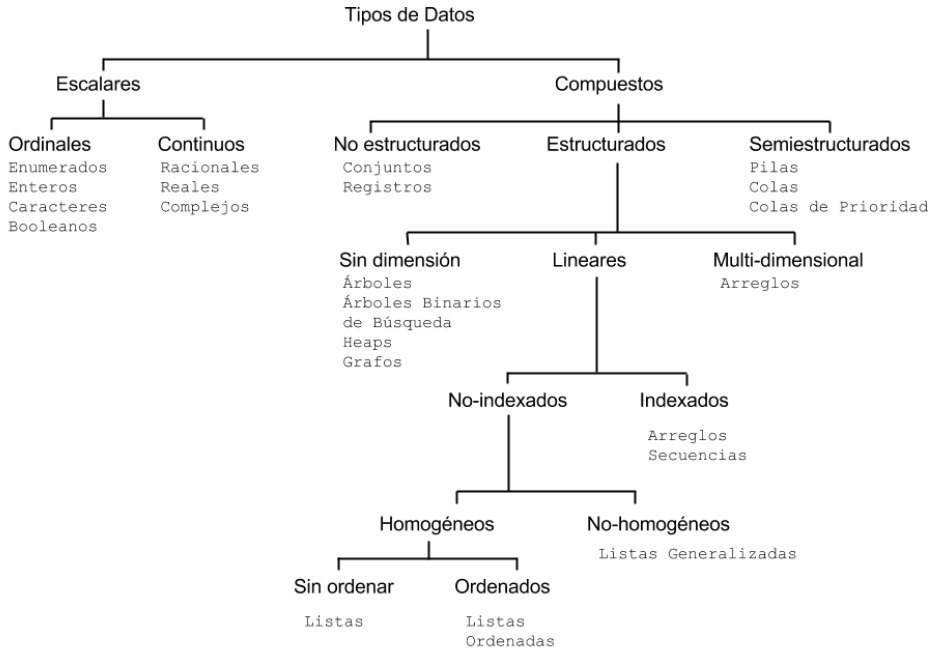
1. Crear un TDA Raiz en el que se creen la operaciones necesarias para ver si el elemento ingresado es válido o no.
2. Crear un TAD Logaritmo en el cual se le hagan todas las operaciones necesarias para su desarrollo.
3. Crear un TAD Clínica, en el se informe informe de una pequeña lista de pacientes ,en que veamos su información ,su prioridad de atención y su orden de salida.
4. Crea un TAD alcancía ,en el que se informe cuanto billetes y monedas hay en la alcancía, ingresar una nueva moneda y un nuevo billete ,decir el mayor valor de las monedas y e los billetes ingresados.
5. Crear un TAD garaje en el cual se digan el numero de auto ingresados y el de salida,saber la hora de ingreso y cuanto es la tarifa más alta pagar.
6. En una heladería se ofrecen conos con bolas de 25 gramos de 3 sabores distintos a escoger entre chocolate, vainilla, fresa y limón. El congelador de helados almacena una cubeta por cada sabor, cada cubeta con su cantidad de helado disponible. Al congelador se le puede: Aumentar una cantidad de helado de un determinado sabor, consultar es alcanza helado para crear un cono dados tres sabores disponibles, consultar las combinaciones de bolas de helado disponibles para crear conos. Cree un TDA para representar un congelador de helados e implemente los metodos indicados
7. Crear un TDA Binario que represente un numero binario. El TDA debe disponer de las siguientes operaciones:
   a) Sumar dos numeros binarios
   b) Convertir un numero binario a un numero decimal
   c) Convertir un numero entero a un numero binario
8. Crear un TDA Fecha compuesta de dia mes y año, y defina las siguientes operaciones
   a) Sumar un dia a una fecha
   b) Restar dos fechas
9. Crear un TDA EnteroGrande que represente un numero entero de hasta 100 digitos. Provea al mismo de las siguientes operaciones:
   a) Suma de dos enteros grandes
   b) Conversion de un entero normal en un entero grande
   c) Conversion de un entero grande si es posible a un entero normal
10. Crear un TDA CajaRegistradora que tiene 6 contenedores, uno para cada denominacion: 5, 10, 20, 50, 100, 200. Una caja permite
    a) Cargar, lo cual incrementa un contenedor con la cantidad de billetes indicada
    b) DarVuelto que dada una cantidad multiplo de 5 e inferior de 500, devuelve el numero de billetes de cada tipo que la satisface utilizando lso billetes de mayor valor siempre que haya disponibles en el cajero
    c) EstadoySaldo que devuelve el saldo total del cajero y la disponibilidad de cada tipo de billete

2.3 Clasificación de los Tipos de Datos

Podemos clasificar los tipos de datos tanto desde el punto de vista funcional del usuario como desde el punto de vista de las estructuras de datos que se usan para su implementación. Ahora, la primera clasificación estara basada en tipos de datos y no en tipos de datos abstractos, debido a que realmente la percepción de abstracción depende del punto de vista de quien use el tipo de dato. Por ejemplo, incluso el tipo de dato entero puede ser representado de forma abstracta: sus valores están definidos por la definición matemática de los números enteros y entre sus operaciones se encuentran la suma, resta, multiplicación, etc.

2.3.1 Clasificación desde el punto de vista funcional

Dale y Walker (1990) explican que desde el punto de vista del usuario, lo único relevante es como interactuar con el tipo de dato para obtener el resultado esperado y proponen la siguiente clasificación desde este punto de vista:



Tipos de Datos: Escalares y Compuestos

Los escalares se refieren a tipos de datos básicos que describen un único valor con operaciones que típicamente permiten combinar dos valores y generar un tercero, o modificar el valor que almacenan con otro nuevo valor. Los tipos de datos compuestos se refieren a aquellos que describen varios valores como una unidad, y cuyas operaciones usualmente permiten almacenar un valor en el grupo de valores que componen variables de estos tipos, recuperar valores del conjunto o eliminar valores del conjunto.

Escalares: Ordinales y Continuos

Los tipos de datos ordinales se refieren a una representación de los valores como entidades discretas, separadas, como por ejemplo, los números enteros. Usualmente los valores en estos tipos de datos guardan un orden, y las operaciones suelen aprovechar esto permitiendo consultar el valor anterior o el sucesor del actual, así como operaciones relacioanles como $>$, $>=$, etc.

Así mismo, como la mayoría de los tipos de datos ordinales describen valores numéricos, las operaciones más comunes son la suma, resta, multiplicación y división.

Por otro lado, los tipos de datos continuos representan valores que son parte de una región continua de una linea, plano o espacio. Una vez más, usualmente estos tipos de datos describen números, pero como son valores continuos, no permiten operaciones para conocer antecesor o sucedor de un valor dado. Más bien, permiten operaciones aritméticas como suma, multiplicación y división. En el caso de que los valores que se representan guarden un orden entre sí, se permiten también operaciones relacionales.

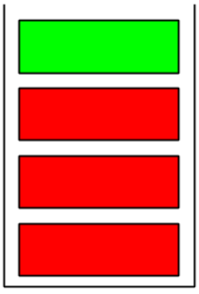
Compuestos: No Estructurados, Estructurados y Semi-estructurados

Dado que los tipos de datos compuestos representan varios valores como un solo todo, estos siempren dependen de operaciones que permitan el almacenamiento de nuevos valores, recuperación de valores existentes y eliminación de valores del conjunto.

En ocasiones, será requerido que los valores que se almacenan como uno solo, manejen una cierta estructura: que se almacenen en un cierto orden, que se encuentren indexados, o que guarden cierta relación entre ellos. En ese caso, a estos tipos de datos compuestos se les conoce como estructurados.

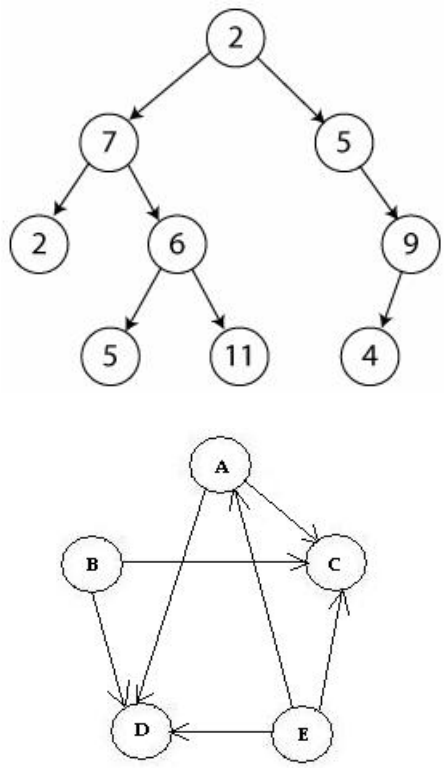
Cuando no se requiere que se guarde ninguna relación entre los valores que conforman el todo, se los conoce como tipos de datos copuestos sin estructura. Un ejemplo son los conjuntos, donde no importa en que orden sean almacenados los datos cuando se agregan al conjunto y las operaciones son orientadas al almacenamiento, recuperación, eliminación o determinación de presencia de los valores del conjunto.

En un invel intermedio, algunos tipos de datos compuestos restrigen el orden en que los datos son almacenados o pueden ser consultados. Estos tipos de datos se conocen como semi-estructurados. Un ejemplo es la Pila, donde, al agregar un elemento, este debe ir al final del conjunto, y al consultar un elemento, el único en capacidad de ser consultado es el último elemento que se agregó.



Compuestos/Estructuados: Sin Dimensiones, Lineares y Multi-Dimensionales

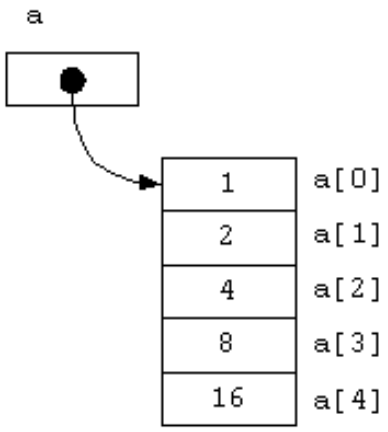
La clasificación de los tipos de datos estructurados se basa en el tipo de conexión o estructura que relacion a los valores del conjunto. En los tipos de datos sin dimension, se requiere que conexión entre varios pares de elementos del conjunto se indique ya sea de forma explicita o de forma implicita. Por ejemplo, para los árboles, se indica que la relación entre un valor y otro es de tipo jerarquica y esto implicitamente indica como será la conexió entre un par de elementos; en un grafo todas las relaciones entre los pares de elementos del conjunto son indicadas explicitamente a través de los arcos que los unen.

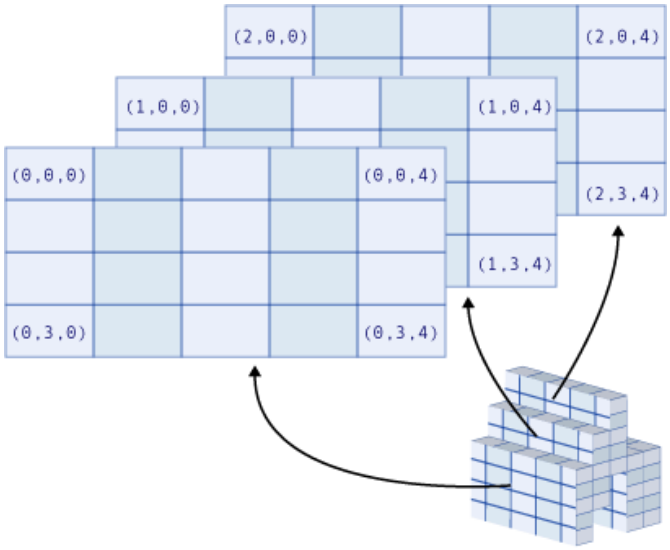


Fuente: http://decsai.ugr.es/~jfv/ed1/c++/cdrom5/ejercicios/ej_grafos.htm

Las operaciones para los tipos de datos sin dimension incluyen almacenamiento, recuperación, eliminación de elementos, así como especificación de la conexión entre elementos y formas de recorrer el conjunto (pasar de un elemento a otro).

Cuando la estructura se impone en base a uno o más valores que permiten indexar un elemento, o en base a un orden natural (puede ser de llegada o orden numérico, etc) de los elementos, el tipo de dato estructurado puede ser lineal o multilineal. Una estructura basada en un valor para indexar un elemento o en el ordenamiento de los elementos, se conoce como lineal. Por ejemplo, un arreglo de una sola dimensión es una estructura lineal. Una estructura multi-dimensional se basa en dos o mas valores para indexar un elemento del conjunto. Un ejemplo de esto son los arreglos multidimensionales. Las operaciones para los tipos de datos multi-dimensionales usualmente son de almacenamiento y recuperación de elementos.





Compuestos/Estructurados/Lineales: No indexados e indexados

En un tipo de dato lineal indexado, el orden de los elementos es específicado por una variable que sirve para referirse a un elemento dado. Un ejemplo de esto es el arreglo multidimensional de dos dimensiones, donde se usan 2 indices para referirse al elemento: uno para la fila y otro para la columna.

En un tipo de dato lineal no indexado no se necesita una variable que se permita referise a un elemento dado, el ordenamiento de los datos puede darse ya sea por el orden en que fueron almacenados o por un orden numérico, etc.). Un ejemplo de esto es el tipo de dato Lista, donde los datos son almacenados uno detrás de otro. Operaciones típicas en estos tipos de datos son de almacenamiento, eliminación, recuperación del primer elemento, recuperación del último elemento, recorrido de elementos, etc.

Compuestos/Estructurados/Lineales/No indexados: Homogénos y No Homogéneos

Un tipo de dato no indexado contiene datos que se encuentran en orden, ya sea impuesto por el programador o implicitamente debido a la estructura de conexión de los elementos (Por ejemplo, orden de llegada). Tomando esto en cuenta, los tipos de datos no indexados pueden ser utilizados para almacenar elementos del diferentes tipos de datos, cuyo caso se los conoce como tipos de datos no homogéneos. Las operaciones de estos tipos de datos no podrían permitir comparación entre elementos, búsqueda ni recorrido. Estas acciones las debería llevar a cabo el usuario programador que conozca los tipos de datos almacenados en la estructura. Operaciones que usualmente se dan en estos tipos de datos son de almacenamiento, eliminación, recuperación del primero, recuperación del siguiente elemento a un elemento dado, y la recuperación del tipo de dato de un elemento dado.

Por otro lado, un tipo de dato que almacene elementos todos del mismo tipo se conoce como Homogéneo.

Compuestos/Estructurados/Lineales/No Indexados/Homogéneos: Sin Ordenar y Ordenados

Los tipos de datos homogéneos almacenan varios elementos y estos se encuentran en un orden lineal. El ordenamiento puede ser inferido de los datos almacenados, en cuyo caso se habla de tipos de datos ordenados. Si, por otro lado, el ordenamiento se da por otros casos, como el orden de llegada, se dice habla de tipos de datos sin ordenar.

En los tipos de datos sin ordenar, el usuario programador del TDA debe especificar donde en el orden lineal de la estructura, desea que se almacene un nuevo elemento. Por ejemplo, en una

Lista, el usuario programador debe indicar si quiere que un nuevo elemento se almacene antes del primer elemento, luego de un elemento ya existente o al final. Otras operaciones comunes en estos tipos de datos son la obtención de la longitud de la lista, recuperación del primero o del último de la lista, etc.

2.3.2 Clasificación desde el punto de vista de Estructuras de Datos

Una clasificación de los tipos de datos desde el punto de vista de las estructuras de datos que se usan para implementarlos, inica basándose en el lugar donde residen: Internas y externas. Si una estructura de datos reside en la *memoria central* del computador se denomina *estructura de datos interna*. Recíprocamente, si reside en un soporte *externo*, se denomina *estructura de datos externa*.

Las estructuras de datos internas pueden ser de dos tipos:

- Estructuras de Datos Estáticas.
- Estructuras de Datos Dinámicas.

Estructuras de Datos Estáticas

Tienen un número fijo de elementos que queda determinado desde la declaración de la estructura en el comienzo del programa. Ejemplo los arreglos. Las estructuras de datos estáticas, presentan dos inconvenientes:

1. La reorganización de sus elementos, si ésta implica mucho movimiento puede ser muy costosa. Ejemplo: insertar un dato en un arreglo ordenado.
2. Son estructuras de datos estáticas, es decir, el tamaño ocupado en memoria es fijo, el arreglo podría llenarse y si se crea un arreglo de tamaño grande se estaría desperdiciando memoria.

Estructuras de Datos Dinámicas

Las estructuras de datos dinámicas nos permiten lograr un importante objetivo de la programación orientada a objetos: la reutilización de objetos. Al contrario de un arreglo, que contiene espacio para almacenar un número fijo de elementos, una estructura dinámica de datos se amplía y contrae durante la ejecución del programa.

A su vez, este tipo de estructuras se pueden dividir en dos grandes grupos según la forma en la cual se ordenan sus elementos.

- Lineales
- No lineales

Estructuras de Datos Lineales

En este tipo de estructuras los elementos se encuentran ubicados secuencialmente. Al ser dinámica, su composición varía a lo largo de la ejecución del programa que lo utiliza a través de operaciones de inserción y eliminación. Dependiendo del tipo de acceso a la secuencia, haremos la siguiente distinción:

- Listas: podemos acceder (insertar y eliminar) por cualquier lado.
- Pilas: sólo tienen un único punto de acceso fijo a través del cual se añaden, se eliminan o se consultan elementos.
- Colas: tienen dos puntos de acceso, uno para añadir y el otro para consultar o eliminar elementos.

Estructuras de Datos No Lineales

Dentro de las estructuras de datos no lineales tenemos los árboles y grafos. Este tipo de estructuras los datos no se encuentran ubicados secuencialmente. Permiten resolver problemas computacionales complejos.

2.4 Análisis de Algoritmos

2.4.1 Los Algoritmos

Una vez que tenemos un algoritmo que resuelve un problema y podemos decir que es de alguna manera correcto, un paso importante es tener idea de la cantidad de recursos, como tiempo de procesador o espacio en la memoria principal que requerirá.

Los objetivos del análisis de algoritmos son:

- Conocer los factores que influyen en la eficiencia de un algoritmo.
- Aprender a calcular el tiempo que emplea un algoritmo.
- Aprender a reducir el tiempo de ejecución de un programa (por ejemplo, de días o años a
- fracciones de segundo).

Factores que influyen en la eficiencia de un algoritmo

Podemos tomar en cuenta muchos factores que sean externos al algoritmo como la computadora donde se ejecuta (hardware y software) o factores internos como la longitud de entrada del algoritmo. Veamos algunos de estos factores.

- **El Hardware**. Por ejemplo: procesador, frecuencia de trabajo, memoria, discos, etc.
- **El Software**. Por ejemplo: sistema operativo, lenguaje de programación, compilador, etc.
- **La longitud de entrada**. El enfoque matemático considera el tiempo del algoritmo como una función del tamaño de entrada. Normalmente, se identifica la longitud de entrada (tamaño de entrada), con el número de elementos lógicos contenidos en un ejemplar de entrada, por ejemplo: en un algoritmo que calcula el factorial de un número, la longitud de entrada sería el mismo número, porque no es lo mismo calcular el factorial de 4 que calcular el factorial de 1000, las iteraciones que tenga que hacer el algoritmo dependerá de la entrada. De igual manera se puede considerar como longitud de entrada: al tamaño de un arreglo, el número de nodos de una lista enlazada, el número de registros de un archivo o el número de elementos de una lista ordenada). A medida que crece el tamaño de un ejemplar del programa, generalmente, crece el tiempo de ejecución. Observando cómo varía el tiempo de ejecución con el tamaño de la entrada, se puede determinar la tasa de crecimiento del algoritmo, expresado normalmente en términos de n, donde n es una medida del tamaño de la entrada. La tasa de crecimiento de un problema es una medida importante de la eficiencia ya que predice cuánto tiempo se requerirá para entradas muy grandes de un determinado problema. Para que un algoritmo sea eficiente, se debe optimizar el tiempo de ejecución y el espacio en la memoria, aunque se producirá la optimización de uno a costa del otro.

2.4.2 Análisis de Algoritmos

El análisis de algoritmo que hacemos toca únicamente el punto de vista temporal (tiempo de ejecución de un algoritmo) y utilizamos como herramienta el lenguaje de programación Java.

Es difícil realizar un análisis simple de un algoritmo que determine la cantidad exacta de tiempo requerida para ejecutarlo. La primera complicación es que la cantidad exacta de tiempo dependerá de la implementación del algoritmo y de la máquina real en que se ejecuta. El análisis normalmente debe ser independiente de la computadora (hardware y software) y del lenguaje o máquina que se utilice para implementar el algoritmo. La tarea de calcular el tiempo exacto requerido suele ser bastante pesado.

Un algoritmo es un conjunto de instrucciones ordenados de manera lógica que resuelven un problema. Estas instrucciones a su vez pueden ser: enunciados simples (sentencias) o enunciados compuestos (estructuras de control). El tiempo de ejecución dependerá de como esté organizado ese conjunto de instrucciones, pero nunca será constante.

Es conveniente utilizar una función T(n) para representar el número de unidades de tiempo (o tiempo de ejecución del algoritmo) tomadas por un algoritmo de cualquier entrada de tamaño n. La evaluación se podrá hacer desde diferentes puntos de vista:

- **Peor caso.** Se puede hablar de T(n) como el tiempo para el peor caso. Se trata de aquellos ejemplares del problema en los que el algoritmo es menos eficiente (no siempre existe el caso peor). Ejemplos: insertar al final de una lista, ordenar un vector que está ordenado en orden inverso, etc. Nos interesa mucho.
- **Mejor caso.** Se habla de T(n) como el tiempo para el mejor caso. Se trata de aquellos ejemplares del problema en los que el algoritmo es más eficiente; por ejemplo: insertar en una lista vacía, ordenar un vector que ya está ordenado, etc. Generalmente no nos interesa.
- **Caso medio.** Se puede computar T(n) como el tiempo medio de ejecución del programa sobre todas las posibles ejecuciones de entradas de tamaño n. El tiempo de ejecución medio es aveces una medida más realista del rendimiento en la práctica, pero es, normalmente, mucho más difícil de calcular que el tiempo de ejecución en el pero caso, ya que requiere definir una distribución de probabilidades de todo el conjunto de datos de entrada, el cuál típicamente es una tarea difícil.

2.4.3 Función de Complejidad

Definición

La función de complejidad de un algoritmo es el número de operaciones elementales que utiliza un algoritmo cualquiera para resolver un problema de tamaño n. Matemáticamente se define la Función de complejidad asi:

Sea A un algoritmo, la función de complejidad del algoritmo A T(n) se define como el número máximo de operaciones elementales que utiliza el algoritmo para resolver un problema de tamaño n.

T(n) = Max {nx: nx es el número de operaciones que utiliza A para resolver una instancia x de tamaño n}

Nota: Una operación elemental es cualquier operación cuyo tiempo de ejecución es acotado por una constante (que tenga tiempo constante). Por ejemplo: una operación lógica, una operación aritmética, una asignación, la invocación a un método.

2.4.4 Operaciones elementales

¿Que es una operación elemental? Una operación elemental, también llamado operador básico es cualquier operación cuyo tiempo de ejecución es constante, es decir, es una operación cuyo tiempo de ejecución siempre va a ser el mismo.

Tipos de Operaciones Elementales

- Operación Lógica: Son operaciones del tipo a $>$ b, o por ejemplo los indicadores que se suelen utilizar en los condicionales que si se cumpla esta condición o esta otra haga esto. Ese o es una operación lógica.
- Operacion Aritmetica: Son operaciones del tipo a + b, o a / b, etc.
- Asignación: Es cuando asignamos a una variable un valor, ejemplo: int a = 20+30, el igual (=) en este caso es la operación de asignación.
- Invocación a un Método: Como su nombre lo dice es cuando llamamos, cuando invocamos

a un método.

Cálculo del T(n)

Para hallar la función de complejidad (t(n)) de un algoritmo se puede evaluar el algoritmos desde tres puntos de vista:

- **Peor Caso:** Se puede hablar de T(n) como el tiempo de ejecución para el peor de los casos, en aquellos ejemplares del problema en el que el algoritmo es Menos Eficiente.
- **Caso Medio:** Se puede comportar el T(n) como el tiempo medio de ejecución del programa sobre todas las posibles ejecuciones de entrada de tamaño n. Es una medida más realista del rendimiento del algoritmo en la práctica, pero es mucho más difícil del cálculo, ya que requiere una distribucion de probabilidades de todo el conjunto de entrada lo cual es una tarea difícil.
- **Mejor Caso:** Se puede hablar de T(n) como el tiempo de ejecución para el mejor de los casos, en aquellos ejemplares del problema en el que el algoritmo es Más Eficiente.

Lo ideal sería poder evaluar el algoritmo en el caso promedio, pero por el nivel de operaciones y dificultad que conlleva este, el estudio de la complejidad de los algoritmos se evalúa en el peor de los casos.

Para calcular el T(n), se deben calcular el número de operaciones elementales de un algoritmo, las cuales estan dadas por: Sentencias consecutivas, condicionales y ciclos.

Calcular T(n) para Sentencias Consecutivas

Las sentencias consecutivas como su nombre lo dicen son aquellas que tienen una secuencia, que van una detrás de otra, y se derivan en dos casos:

1. Si se tiene una serie de instrucciones consecutivas:

Sentencia 1 $\rightarrow$ n1 operaciones elementales
Sentencia 2 $\rightarrow$ n2 operaciones elementales
.
.
.
Sentencia k $\rightarrow$ nk operaciones elementales

$$\sum_{i=1}^{k} ni = n1 + n2 \pm \cdots + nk$$

2. Procedimientos:

Procedimiento 1 $\rightarrow$ f1(n)
Procedimiento 2 $\rightarrow$ f2(n)
.
.
.
Procedimiento k $\rightarrow$ fk(n)

$$\sum_{i=1}^{n} f(i)$$

2.4.5 Calcular T(n) para Ciclos

Para hallar la complejidad computacional de un ciclo existen, los siguientes casos:

Caso A: Cuando los incrementos o decrementos son de uno en uno y son constantes.

El ciclo While o Mientras es el básico, a partir de él se analizan los demás ciclos.

Codigo base:

i ← 1
mientras i <= n Haga
Proceso

 # De Operaciones (Formula) = 1 + (n+1) * nc + n * np

Siendo:

nc: Número de operaciones de la condición del ciclo.
np: Número de operaciones del ciclo.

Ejemplo: Hallar la función de complejidad del siguiente algoritmo:

lineas	Código	# De Operaciones
1	A = 0 ——————————- >	1
2	i = 1 ——————————- >	1
3	Mientras (i <= n) ————— >	nc = 1
4	A = A + i ————— >	2
5	i = i + 1 ————— >	2 np = (2+2) = 4
6	imprima A ——————————- >	1

La complejidad se analiza por lineas, de la siguiente forma:

Lineas	Complejidad	
de 2 a 5	1 + (n + 1) * nc + (n * np) 1 + n + 1 + 4n 5n + 2	1 + (n + 1) * 1 + (n * 4)
de 1 a 6	5n + 2 + 2	

Finalmente:

T(n) = 5n + 4

Ejemplo: Hallar la función de complejidad del siguiente algoritmo.

lineas	Código	# De Operaciones
1	x = 1 ——————————>	1
2	i = 1 ——————————->	1
3	Mientras (i <= n) ——————>	nc = 1
4	x = x * i ——————->	2
5	i = i + 1 ——————>	2 np = (2+2) = 4
6	Si (x > 100) entonces ————>	nc = 1
7	x = 100 ——————>	1
8	imprima x ——————>	1

Lineas	Complejidad
de 2 a 5	1 + (n+1)nc + n * np 1 + (n+1)1 + n * 4 1 + n + 1 + 4n 5n + 2
de 6 a 7	nc + max 1 + 1 2
de 1 a 8	(5n + 2) + (2) + 1 + 1 5n + 6

Finalmente:

$T(n) = 5n + 6$

Ciclo For o Para:

Codigo base:

Para (i←1 ; i <=n ; i++)
Proceso

 # De Operaciones (Formula) = 1 + (n+1) * nc + n * (np+1)

Siendo:

nc: Número de operaciones de la condición del ciclo.
np: Número de operaciones del ciclo.

Ejemplo: Hallar la función de complejidad del siguiente algoritmo.

lineas	Código	# De Operaciones
1	f = 0 ——————————>	1
2	Para (i = 1 ; i <=n ; i++) –>	nc = 1
3	f = f * i ——————>	2 operaciones —> np = 2
4	imprima f ——————->	1

Lineas	Complejidad
de 2 a 3	1 + (n+1)nc + n(np + 1) 1 + (n+1)1 + n(2 + 1) 1 + n + 1 + n(3) 1 + n + 1 + 3n 4n + 2
de 1 a 4	(4n + 2) + 2 4n + 4

Finalmente:

T(n) = 4n + 4

Ejemplo:

NOTA PARA TENER EN CUENTA:

Para ciclos ya sean MIENTRAS(while) O PARA(for), observamos que cada ciclo tiene un límite superior que define hasta donde va a iterar(repetir) ya sea definido estrictamente menor (<) que n, o en dado caso que n sea una constante ; por consecuencia debemos usar la siguiente fórmula, el cual nos permite aclarar el valor real de n; es decir, cuantas veces itera el ciclo (# de veces que ingresa al ciclo)

FORMULA: n = Ls - Li + 1
Siendo:

Ls: Limite superior del ciclo. (Hasta donde itera)
Li: Limite inferior del ciclo. (Desde donde inicializa)

En este ejemplo tenemos el ciclo → Para (i = 1 ; i < n ; i++).

Como observamos el límite inferior(Li) del ciclo es 1 porque inicializa desde 1; el límite superior(Ls) es (n-1) porque va hasta i estrictamente menor (<) que n; es decir el ciclo ingresaría n veces menos 1 por que es estrictamente menor (<).

En los ejemplos anteriores, los ciclos tienen como limite superior n porque llegaba hasta i<=n, es decir ingresaba las n veces al ciclo.

Para comprobar lo dicho en este caso, entonces usamos la fórmula:

n = Ls - Li + 1
n = (n-1) - (1) + 1
n = n-1

En conclusión, como n = n-1 ; el número total de veces que se ingresa al ciclo es (n-1) veces; en consecuencia reemplazamos el valor de n que es (n-1) en la complejidad.

lineas	Código	# De Operaciones
1	f = 1 ——————————>	1
2	Para (i = 1 ; i < n ; i++) –>	nc = 1
3	f = f * i ——————>	2 operaciones —> np = 2
4	imprima f ——————->	1

Lineas	Complejidad
de 2 a 3	1 + (n+1)nc + n(np + 1) “como n=(n-1) entonces”
	1 + ((n-1)+1)(1) + (n-1)(2 + 1)
	1 + n + (n-1)(3)
	1 + n + (3n - 3)
	1 + n + 3n - 3
	4n - 2
de 1 a 4	(4n - 2) + 2
	4n

FInalmente:

$T(n) = 4n$

Caso B: Cuando los incrementos o decrementos dependen del valor que tome i.

While o Mientras:
Codigo base:

i ← 1
mientras i <= n Haga
Proceso (i)

$$\#\ de\ Operaciones\ (Formula) = 1 + (n+1) * nc + \sum_{i=1}^{n} f(i)$$

Siendo:

nc: Número de operaciones de la condición del ciclo.
f(i): Función de complejidad del proceso.

For o Para
Codigo base:

Para (i←1 ; i <=n ; i++)
Proceso (i)

Siendo:

$$\#\ de\ Operaciones\ (Formula) = 1 + (n+1) * nc + n + \sum_{i=1}^{n} f(i)$$

nc: Número de operaciones de la condición del ciclo.
f(i): Función de complejidad del proceso.

Ejemplo:

lineas	Código	# De Operaciones
1	s = 0 ——————————————->	1
2	Para (i = 1 ; i <=n ; i++) ——————>	nc = 1
3	Para (j = 1 ; j<= i ; j++) ——>	nc = 1
4	s = s + 1 ——————->	2 operaciones —> np = 2
5	imprima f ——————————————>	1

Lineas **Complejidad**

de 3 a 4

$1 + (n+1)nc + n(np+1)$ "$CasoA \rightarrow n = i''$
$1 + (i+1)1 + i(2+1)$
$1 + i + 1 + i(3)$
$1 + i + 1 + 3i$
$4i + 2 - - - - - > f(i)$

de 2 a 4

$1 + (n+1)nc + n + \sum_{i=1}^{n} f(i)$
$1 + (n+1)nc + n + \sum_{i=1}^{n} 4i + 2$
$2n + 2 + \sum_{i=1}^{n} 4i + \sum_{i=1}^{n} 2$
$2n + 2 + 4\sum_{i=1}^{n} i + 2\sum_{i=1}^{n} 1$
$2n + 2 + 4\left(n\frac{(n+1)}{2}\right) + 2n$
$4n + 2 + (2n^2) + 2n$
$(2n^2) + 6n + 2$

de 1 a 5

$(2n^2) + 6n + 2 + 2$
$(2n^2) + 6n + 4$

En conclusión:

$$T(n) = 2n^2 + 6n + 4$$

CASO C: Cuando no sabemos cuánto va a ser el incremento o decremento del proceso.

Código base:

mientras<Condición> Haga
Proceso

 # De Operaciones (Formula) = (nr + 1) * nc + (nr * np)

siendo:

nc: Número de operaciones de la condición del ciclo.
np: Número de operaciones del ciclo.
nr: Número de veces que se repite el proceso, en sí es una fórmula que cumple las veces que se ejecuta el proceso para todo número n.

Función Piso / Techo

Sabemos claramente que (5/4) es igual a 1.25

Función piso es aproximar el fraccionario a su más cercano valor entero menor o igual a él, un ejemplo sería:

$$\text{Función piso: } \left\lfloor \tfrac{5}{4} \right\rfloor = \text{sería 1}$$

ahora bien si tomamos el mismo fraccionario y hallamos función techo sería aproximarlo a su más cercano valor mayor o igual a él, un ejemplo sería:

$$\text{Función Techo: } \left\lceil \tfrac{5}{4} \right\rceil = \text{sería 2}$$

NOTA: Si tenemos un entero la función piso y techo es el mismo entero

¿Como hallar el nr?

Para permitirnos calcular t(n) o función de complejidad con un incremento no lineal será incierto saber cual es el número de veces que va a iterar el ciclo ya sea while o un for

Cuando un incremento por lo general no en todas las ocasiones sean de suma de 2 (contador=contador+2) dentro del ciclo, el contador irá sumando de 2 en 2; el nr podría ser un fraccionario con numerador siempre n y denominador la constante que se está sumando en este caso 2.

Ejemplo: Hallar la función de complejidad del siguiente algoritmo

lineas	Código	# De Operaciones
1	t = 0 ——————————————->	1
2	i = 1 ——————————————->	1
3	Mientras (i <= n)——————————>	nc = 1
4	t = t + 1——————————>	2
5	i = i + 2 ——————————>	2
		np = (2 + 2) = 4
6	imprima t ——————————>	1

Estudio para hallar nr se debe analizar la cantidad de veces que itera el bucle para cada valor de n:

6	III	$nr = \lceil \frac{n}{2} \rceil = \lceil \frac{6}{2} \rceil = 3$	3
7	IIII	$nr = \lceil \frac{n}{2} \rceil = \lceil \frac{7}{2} \rceil = 4$	4
8	IIII	$nr = \lceil \frac{n}{2} \rceil = \lceil \frac{8}{2} \rceil = 4$	4
9	IIIII	$nr = \lceil \frac{n}{2} \rceil = \lceil \frac{9}{2} \rceil = 5$	5
10	IIIII	$nr = \lceil \frac{n}{2} \rceil = \lceil \frac{10}{2} \rceil = 5$	5

Para para hallar nr se debe analizar la cantidad de veces que itera el bucle para cada valor de n. En este caso es clave observar que la variable controladora del ciclo es i, y tiene incrementos de 2 en 2, por lo tanto,

$$nr = \frac{\lceil n \rceil}{2}$$

Por tanto, la complejidad del algoritmo se calcula así:

Lineas	Complejidad
de 3 a 5	$(nr+1)nc + nr*(np)$ $\left(\frac{\lceil n \rceil}{2}+1\right)*(1)+\left(\frac{\lceil n \rceil}{2}\right)*(4)$ $\frac{\lceil 5n \rceil}{2}+1$
de 1 a 6	$\frac{\lceil 5n \rceil}{2}+1+3$

Finalmente,

$$\boxed{T(n) = \frac{\lceil 5n \rceil}{2} + 4}$$

Ejemplo:

lineas	Código	# De Operaciones
1	t = 0 ——————————->	1
2	i = 1 ——————————->	1
3	Mientras (i <= n)——————>	nc = 1
4	t = t + 1——————>	2
5	i = i * 2 ——————>	2
		np = (2 + 2) = 4
6	imprima t ——————>	1

En este caso el ciclo mientras tiene incrementos en base 2 (i = i*2), por lo tanto nr es una expresión logarítmica dada por la formula:

$$nr = \lfloor \log_2(n) \rfloor + 1$$

Por tanto, la complejidad del algoritmo se calcula así:

Lineas	Complejidad
de 3 a 5	$(nr+1)nc + nr*(np)$
	$((\lfloor \log_2(n) \rfloor + 1) + 1) * (1) + (\lfloor \log_2(n) \rfloor + 1)(4)$
	Simplificando la expresión,
	$5 \lfloor \log_2(n) \rfloor + 6$
de 1 a 6	$(5 \lfloor \log_2(n) \rfloor + 6) + 3$

Finalmente,

$$5 \lfloor \log_2(n) \rfloor + 9$$

2.4.6 Orden de Magnitud (Notación O Grande)

Cuando se trata de algoritmos, el cálculo detallado del tiempo de ejecución de todas las operaciones primitivas llevaría mucho tiempo. Además, ¿qué importancia tendría el número de instrucciones primitivas ejecutadas por un algoritmo? Es más útil en el análisis de algoritmos, ver la velocidad de crecimiento del tiempo de ejecución como una función del tamaño de la entrada n, en lugar de realizar cálculos detallados. Es más significativo saber que un algoritmo crece, por ejemplo, proporcionalmente a n, a razón de un factor constante pequeño que depende del hardware o software y puede variar en un cierto rango, dependiendo de una entrada n específica. Esto lo que se conoce como orden de magnitud “O(g(n))” o notación asintótica o notación “O grande”. El orden de magnitud se utiliza para comparar la eficiencia de los algoritmos.

La notación O grande es una técnica utilizada para el análisis de la complejidad computacional de un algoritmo, este análisis se centra en el término dominante (El término que más aumenta), permitiendo así ignorar constantes y términos de orden menor.

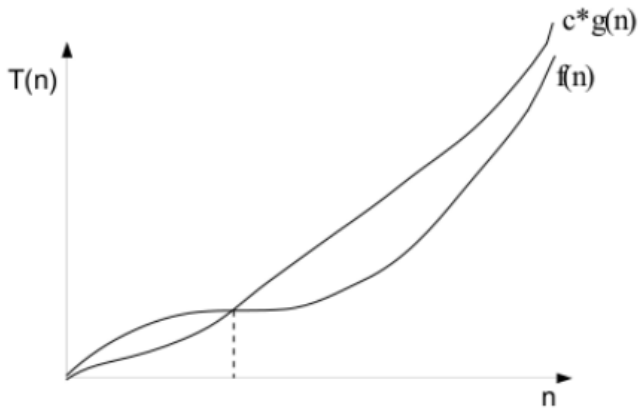
Por ejemplo:

T(n) = 1	**----** > O (1) ----> Orden Constante
T(n) = $log_2 n$	**----** > O $(log\, n)$ ----> Orden Logaritmico
T(n) = $an + b$	**----** > O (n) ----> Orden Lineal
T(n) = $nlog_2 n$	**----** > O (n) ----> Orden $nlog_2 n$
T(n) = $an^2 + bn + c$	**----** > O (n^2) ---->Orden Cuadratico
T(n) = $an^3 + bn^2 + c$	**----** > O (n^3) ---->Orden Cubico
T(n) = $n^m,\ m = 0, 1, 2, 3\ldots$	**----** > O (n^2) ---->Orden Polinomial
T(n) = $c^n,\ c > 1$	**----** > O (n^3) ---->Orden Exponencial
T(n) = $n!$	**----** > O (n^2) ---->Orden Factorial

Definición conceptual del Orden de Magnitud

Sean f(n) y g(n) funciones definidas sobre enteros no negativos. Diremos que f(n) es O(g(n)) si existe un número real constante c>0 y un entero constante n0>=1, tal que f(n)≤c*g(n) para todo entero n>=n0.

Por consiguiente, g(n) es un límite superior para f(n), como se ilustra en la figura 2.4.6.



El orden de magnitud es ampliamente utilizado para caracterizar los tiempos de ejecución en términos de la longitud de entrada n, el cual varía de problema en problema, pero es usualmente una noción intuitiva del "tamaño" del problema.

Asimismo, el orden de magnitud nos permite ignorar factores constantes y términos de orden menor y apuntar a los componentes principales de una función que influyen en su crecimiento.

Cuando se dice que f(n) es del O(g(n)), se está garantizando que la función f(n) crece a una velocidad no mayor que g(n); así g(n) es una cota superior de f(n).

Propiedades del Orden de Magnitud

1. O(f(x)) + k = O(f(x))
2. O(k f(x)) = O(f(x))

3. O(f(x)) * O(g(x)) = O(f(x) * g(x))
   O(f(x)) + O(g(x)) = max (O(f(x)), O(g(x)))

2.4.7 Recursividad

Def: Es una técnica de programación en la cual un método puede llamarse a sí mismo, en la mayoría de casos un algoritmo iterativo es más eficiente que uno recursivo si de recursos de la computadora se trata, pero un algoritmo recursivo en muchos casos permite realizar problemas muy complejos de una manera más sencilla.

Reglas de la recursividad:

Para que un problema pueda resolverse de forma recursiva debe cumplir las siguientes 3 reglas:

Regla 1:Por lo menos debe tener un caso base y una parte recursiva.
Regla 2:Toda parte recursiva debe tender a un caso base.
Regla 3:El trabajo nunca se debe duplicar resolviendo el mismo ejemplar de un problema en llamadas recursivas separadas.

Ejemplo: Calcular el factorial de un número.

FACTORIAL DE UN NÚMERO N

8! = 8 * 7!
7! = 7 * 6!
6! = 6 * 5!

.

.

En general,
n! = n * (n-1)!

Veamos un caso particular, calculemos el factorial de 5 (5!):

factorial de 5 = 5 * 4! ——-> “factorial de 5 es igual 5 multiplicado por factorial de 4”
factorial de 4 = 4 * 3! ——->“factorial de 4 es igual 4 multiplicado por factorial de 3”
factorial de 3 = 3 * 2!——-> “factorial de 3 es igual 3 multiplicado por factorial de 2”
factorial de 2 = 2 * 1!——-> “factorial de 2 es igual 2 multiplicado por factorial de 1”
factorial de 1 = 1 ——-> “factorial de 1 es 1” ——> “caso base”

Una implementación en java seria:

```
public long factorial (int n){
    if (n == 0 || n==1) //Caso Base
            return 1;
    else
            return n * factorial (n - 1); //Parte Recursiva
}
```

2.4.8 Complejidad de un Algoritmo Recursivo

Método del árbol de recursión

Existen varios métodos para calcular la complejidad computacional de algoritmos recursivos. Uno de me los métodos más simples es el árbol de recursión, el cual esadecuado para visualizar que pasa cuando una recurrencia es desarrollada. Un árbol de recursión se tienen en cuenta los siguientes elementos:

Nodo: Costo de un solo subproblema en alguna parte de la invocación recursiva.

Costo por Nivel: La suma de los costos de los nodos de cada nivel.

*Costo Total: Es la suma de todos los costos del árbol.

Ejemplo. Utilizando el método del árbol de recursión calcular la complejidad computacional del algoritmo recursivo del factorial. Lo primero es calcular las operaciones elementales de cada linea:

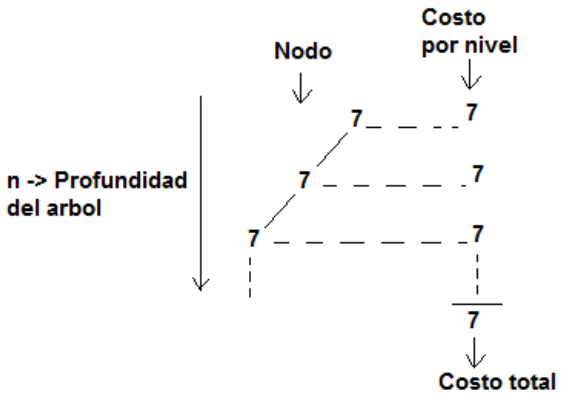
```
public long factorial (int n){ #operaciones elementales
    if (n == 0 || n==1) //Caso Base 3
            return 1; 1
    else
            return n * factorial (n - 1); //Parte Recursiva 4
}
```

$$T(n) = \begin{cases} 4 \quad si \quad (n = 0) \; o \; (n = 1) \\ 7 + T(n-1), \quad Si \quad n>1 \end{cases}$$

Para hallar la complejidad se debe resolver esta recurrencia:

$$T(n) = 7 + T(n - 1)$$

El árbol de recursión es el siguiente.



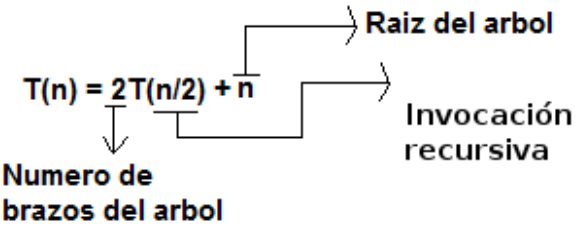
El costo total es la sumatoria de los costos de cada nivel:

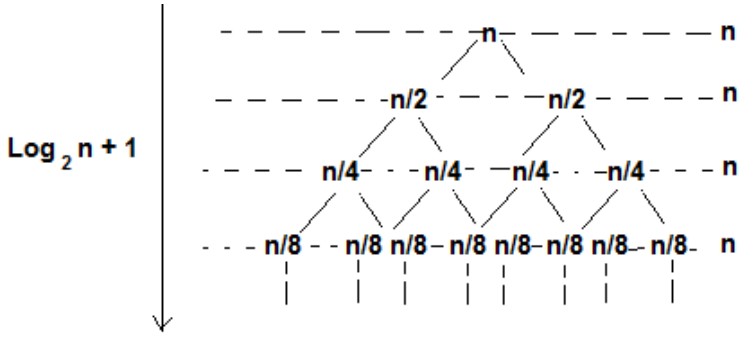
$$\sum_{i=1}^{n} 7 = 7n \; O(n) \rightarrow \text{Orden lineal}$$

Ejemplo

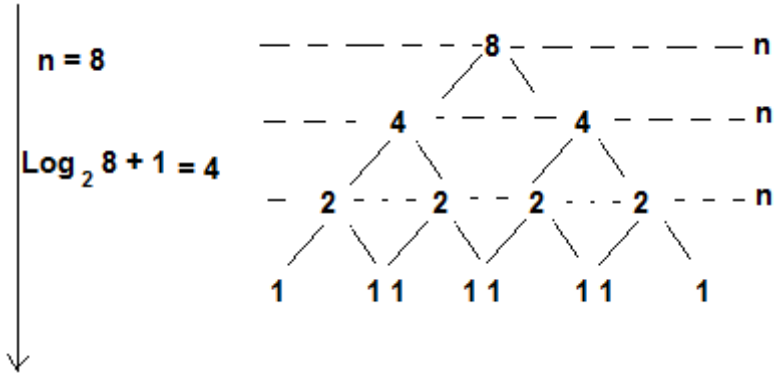
Utilizando el método del árbol de recursión, calcular la complejidad computacional de la siguiente recurrencia:

T(n) = 2 T(n/2) + n





Para entender mejor el árbol de recursión anterior, ilustramos cómo sería cuando n = 8:



Finalmente, la complejidad de la recurrencia está dada por la suma de los costos de cada nivel del árbol:

$$\sum_{i=1}^{Log_2 n+1} n = n+n+n+n = n\log_2(n+1) \qquad O(nLog_2(n))$$

Ejercicios Propuestos

*Ejercicios Caso A.
Calcular la función de complejidad T(n) de los siguientes algoritmos:
1)
c = 0
i = 1
Mientras (i <= n)

```
    c = c + 1
    i = i + 1
imprima c
2)
a = 0
i = 1
Mientras (i <n)
    a = a + i
    i = i + 1
imprima a
3)
a=0
i = 8
Mientras (i <n)
    a = a + i
    i = i + 1
imprima a
4)
a=0
i = 3
Mientras (i <n)
    a = a + i
    i = i + 1
imprima a
5)
f = 1
Para( i = 1; i <= n; i++)
    f = f * i
imprima f
6)
f = 6
Para(i = 5; i <= n/3 ; i++)
    f = f * 6 + i
imprima f
```

Respuestas:
1) T(n) = 5n + 4
2) T(n) = 5n 1
3) T(n) = 5n 36
4) T(n) = 5n 11
5) T(n) = 4n
6) T(n) = 2n 13

Ejercicios Caso B
Calcular la función de complejidad T(n) de los siguientes algoritmos
1)

```
s = 0
Para( i = 1; i <n; i++)
    Para( j = 1; j <i; j++)
```

```
            Para( k = 5; k <= j; k++)
               s = s + 4 * s
imprima s
2)
t = 0
Para( i = 1; i <= n; i++)
    Para( j = 1; j <= n; j++)
          t = t + j + i
imprima t
3)
t = 10
Para( i = 1; i <= n; i++)
    Para( j = 3; j <= i; j++)
          t = t + 1
imprima t
4)
t = 0
Para( i = 1; i <= n; i++)
    Para( j = 1; j <= i; j++)
          Para( k = 1; k <= j; k++)
             t = t + 1
imprima t
5)
t = 20
Para( i = 1; i <n; i++)
    Para( j = 1; j <2 * n; j++)
          t = t + 1
imprima t
```

Respuestas:
1) T(n) = 6/5n3 – 21/2n2 + 89/3 – 16
2) T(n) = 5n2 + 4n + 4
3) T(n) = 2n2 – 2n + 4
4) T(n) = 2/3n3 + 4n2 + 22/3n + 4
5) T(n) = 10n2 – 10n + 4

Ejercicios Caso C.
Calcular la función de complejidad T(n) de los siguientes algoritmos. “recuerde que los incrementos ya no son lineales(hallar nr)”

```
1)
i = 0
s = 0
Mientras (i <= n)
    s = s + 1 + i
    i = i + 3
imprima s
2)
i = 0
t = 3
```

```
Mientras (i <= n)
    t = t + 1
    i = i * 3
imprima t
3)
i = n
t = 0
Mientras (i <= n)
    t = t + 1 + i
    i = i/2
imprima t
```

Ejercicios Recurrencias
Calcular la complejidad de las siguientes recurrencias:

1) $T(n) = 3T(n/4) + n^2$
2) $T(n) = T(n/2) + 1$
3) $T(n) = 2T(n/2) + 1$
4) $T(n) = 2T(n - 1) + k$

2.5 Algoritmos de Búsqueda y Ordenamiento

A lo largo del tiempo, varias técnicas identificadas han llevado a proponer algoritmos eficientes para la resolución de problemas típicos en el area de computación. Entre esos problemas se encuentran el de búsqueda en un conjunto de datos y el de ordenamiento. Es importante recalcar que cada algoritmo tiene un nivel de eficiencia diferente, y es importante poder determinar, dadas las características de un problema en si, y de las entradas que van a ser procesadas, cual es el algoritmo más óptimo a ser utilizado.

2.5.1 Algoritmos de Búsqueda

El ser humano desarrolla un sinnúmero de actividades, muchas de las cuales requieren que la recopilación de elementos que en ellas se emplean esten ordenados de una determinada manera. Una empresa, por ejemplo, constantemente necesita realizar búsquedas relacionadas con los datos de sus empleados o clientes; buscar información de un elemento en una lista.

El problema de la búsqueda radica en la **recuperación de la información lo más rápidamente posible**. Consiste en localizar un elemento en una lista o secuencia de elementos.

La operación de búsqueda puede llevarse a cabo sobre elementos ordenados o sobre elementos desordenados. En el primer caso, la búsqueda se facilita, y por lo tanto se ocupará menos tiempo que si se trabaja con elementos desordenados.

Los métodos de búsqueda pueden clasificarse en internos y externos, dependiendo el lugar donde estén almacenados los datos sobre los cuales se hará la búsqueda. Se denomina:

- **Búsqueda interna** si todos los elementos se encuentran en memoria principal (por ejemplo, almacenados en arreglos, vectores o listas enlazadas).
- **Búsqueda externa** si los elementos se encuentran en memoria secundaria. (Ya sea disco duro, disquete, cintas magnéticas, CD's, memorias flash).

A continuación nos centraremos en la búsqueda interna. Así, la organización de los datos sobre la que resolveremos el problema de la búsqueda consiste en un arreglo de **n** elementos de tipo elemento (valores puntuales, registros, etc.); Es necesario que los **n** elementos sean

distintos. Si existen elementos iguales, al realizar la búsqueda se localiza únicamente uno de ellos, pudiendo dar lugar a errores al no contener la información correcta que se quería recuperar.

Cada algoritmo de búsqueda procura localizar en un arreglo un elemento X. Una vez finalizada la búsqueda puede suceder:

- Que la búsqueda haya tenido éxito, habiendo localizado la posición donde estaba almacenado el elemento X.
- Que la búsqueda no haya tenido éxito, concluyendo que no existía ningún elemento X.

Después de la búsqueda sin éxito, a veces es interesante introducir el elemento X. Un algoritmo con tal objetivo se denomina de **búsqueda e inserción**

2.5.1.1. Búsqueda Secuencial

La búsqueda secuencial o lineal, consiste en recorrer y examinar cada uno de los elementos del arreglo, mediante un bucle voraz de izquierda a derecha, hasta encontrar el o los elementos buscados, o hasta que se han evaluado todos los elementos del arreglo.

El algoritmo implementado en java es el siguiente:

```
public class Busqueda {
/*** Busca secuencialmente un valor en el arreglo
* ORDEN(N) EN EL PEOR CASO
* @param valor
* valor a buscar
* @param arreglo
* arreglo de datos en cualquier orden
* @return true si lo encuentra, false si no encuentra el valor
*/
public boolean buscarSecuencial(int valor, int[] arreglo) {
    boolean encontrado = false;
    int i = 0;
    int n = arreglo.length;
    while (i < n && !encontrado) {
        if (arreglo[i] == valor)
            encontrado = true;
        else {
            i++;
    }
    return encontrado;
}
```

El programa principal que invoca este método de búsqueda sería:

```
public class ClienteMain { public static void main(String[] args) {
        Busqueda b = new Busqueda();
        int[] arreglo = { 1, 2, 3, 4, 5, 6, 8, 9 };
        System.out.println(b.secuencial(5, arreglo));
    }
}
```

Complejidad computacional de la búsqueda secuencial

El **mejor caso**, se produce cuando el elemento a buscado sea el primero que se examina, de modo ques sólo se necesita una comparación. En el **peor caso,** el elemento deseado es el último que se examina, de modo ques se necesitan ***n***comparaciones. En el **caso medio,** se encontrará el elemento deseado aproximadamente en el centro de la colección, haciendo ***n/2*** comparaciones
Su complejidad es:

$$T(n) = \sum_{i=1}^{n} 1 = \varepsilon\ O(n)$$

Concluimos entonces que:

- Para el mejor caso **T(n) = O(1)**
- Para el caso Promedio y Peor caso **T(n) = O(n)**

2.5.1.2. Búsqueda Binaria

Si los elementos sobre los que se realiza la búsqueda están ordenados, entonces podemos utilizar un algoritmo de búsqueda mucho más rápido que el secuencial: la búsqueda binaria. Consiste en reducir en cada paso el ámbito de búsqueda a la mitad de los elementos, basándose en comparar el elemento a buscar con el elemento que se encuentra en la mitad del intervalo y con base en esta comparación:

Si el elemento buscado es menor que el elemento medio, entonces sabemos que el elemento está en la mitad inferior de la tabla.
Si es mayor es porque el elemento está en la mitad superior.
Si es igual se finaliza con éxito la búsqueda ya que se ha encontrado el elemento.

Si se vuelve a repetir el mismo proceso anterior con la parte del arreglo que no hemos descartado, iremos avanzando rápidamente hacia el valor que queremos localizar.

Puede darse el caso en el que el sub-arreglo a dividir está vacío y aún no se a encontrado el elemento. Sobre entendemos que el valor buscado no existe en el arreglo.

El algoritmo implementado en java es el siguiente:

```
public class Busqueda {

/*ORDEN (LOG N) EN EL PEOR CASO Y CASO MEDIO. ORDEN(1) EN EL MEJOR
*CASO
* @param valor
* valor que estamos buscando
* @param arreglo
* arreglo ordenado de datos
* @return true cuando lo encuentra, false cuando no encuentra el dato a
* buscar
*/

public boolean binaria(int valor, int[] arreglo) {
boolean encontrado = false;
int inicio = 0;
int fin = arreglo.length - 1;
while (inicio <= fin && !encontrado) {
```

```
int medio = (inicio + fin) / 2;
if (arreglo[medio] == valor) {
encontrado = true;
}
else {
if (arreglo[medio] > valor)
fin = medio - 1;
else
inicio = medio + 1;
}
}
return encontrado;
}
```

El programa principal que invoca este método de búsqueda sería:

```
public class ClienteMain {
public static void main(String[] args) {
Busqueda b = new Busqueda();
int[] arreglo = { 1, 2, 3, 4, 5, 6, 8, 9 };
System.out.println(b.binaria(5, arreglo));
}
}
```

Complejidad de la búsqueda binaria

El algoritmo determina en qué mitad está el elemento y descarta la otra mitad. En cada división, el algoritmo hace una comparación. El número de comparaciones es igual al número de veces que el algoritmo divide el array por la mitad. Si se supone que n es aproximadamente igual a 2k entonces k ó k + 1 es el número de veces que n se puede dividir hasta tener un elemento encuadrado en (k = log2 n). Su función de complejidad es:

$$T(n) = \sum_{i=1}^{\log(n)} 1 = \log_2(n)\ \varepsilon\ O(\log_2(n))$$

Por consiguiente el algoritmo es $O(log_2 n)$ en el peor de los casos. En el caso medio $O(\log_2(n))$ en el caso medio y O(1) en el mejor de los casos.

En general, este método realiza $log_2(n)$ comparaciones antes de encontrar el elemento, o antes de descubrir que no está. Este número es muy inferior que el necesario para la búsqueda lineal para casos grandes.

Versión recursiva de la búsqueda binaria

Su única novedad es que después de comparar el elemento de búsqueda con el elemento de la mitad de la tabla, se invoca recursivamente a realizar la búsqueda en uno de los dos posibles intervalos, el inferior o el superior, finalizando en el momento en el que se encuentre el elemento o ya se tenga un único elemento y no coincida con el buscado.

```
public int BinariaRecursiva(int [] A, int X, int fin, int inicio)
{ int medio;
    if (inicio > fin) return -1;
    else{
```

```
        medio = (inicio + fin) / 2;
        if (A[medio] > X)
           return BinariaRecursiva(A, X, medio+1, fin);
        else
           if (A[medio] > X)
              return BinariaRecursiva(A, X, inicio, medio -1);
           else
              return medio;
    }
}
```

Complejidad de la búsqueda binaria recursiva

Para medir la velocidad de cálculo del algoritmo de búsqueda binaria, se deberán obtener el número de comparaciones que realiza el algoritmo, es decir, el número de vueltas del ciclo o el número de recursiones. Aunque en principio puede parecer que ambas versiones invierten el mismo tiempo, la recursiva es más lenta a medida que se incrementa el número de elementos, ya que existirán más llamadas a la función por resolver, con el consiguiente gasto de tiempo de guardar y restaurar parámetros.

En el mejor caso, la búsqueda binaria podría toparse con el elemento buscado en el primer punto medio, requiriéndose sólo una comparación de elementos. Esto equivale al mejor caso durante una búsqueda secuencial, pero en el peor de los casos la búsqueda binaria es mucho más rápida cuando N es grande.

El algoritmo de búsqueda binaria progresivamente va disminuyendo el número de elementos sobre el que realizar la búsqueda a la mitad: n, (n/2), (n/4), ... Así, tras $log_2(N)$ divisiones se habrá localizado el elemento o se tendrá la seguridad de que no estaba.

Mejor Caso: En sus casos óptimos, tanto la búsqueda secuencial como la binaria requieren sólo una comparación; esto significa que sus tiempos de ejecución óptimos no dependen de la cantidad de datos: son constantes y por tanto proporcionales a 1, es decir, son de O(1).

Peor Caso: En el peor caso, la búsqueda secuencial y la binaria sí dependen de N. La primera recorre todo el arreglo, requiriendo un tiempo de O(n); la binaria divide el arreglo, requiriendo sólo un tiempo O(log n).

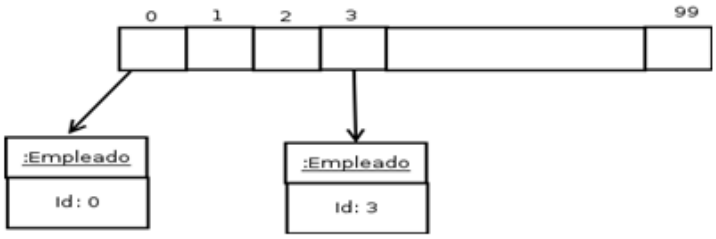
2.5.1.3. Búsqueda Hash

La búsqueda binaria proporciona un medio para reducir el tiempo requerido para buscar en una lista. Este método, sin embargo, exige que los datos estén ordenados y siempre depende del número n de elementos que conforman el conjunto de datos.

Surge entonces una pregunta,***¿es posible lograr una búsqueda de O(1)?,*** es decir, una búsqueda que tome el mismo tiempo para buscar cualquier elemento de una lista. La respuesta es sí. Para ello se utiliza la técnica hashing.Este método se conoce como transformación de claves (clave-dirección) y consiste en convertir el elemento almacenado (numérico o alfanumérico) en una dirección (índice) dentro de un arreglo, de manera que se puede acceder al elemento directamente.

Ejemplo.

Vamos a partir de un sencillo ejemplo. Supongamos que tenemos una lista de empleados de una pequeña empresa. Cada empleado tiene asignado un número de identificación de 0 hasta 99. Entonces se necesitaría un vector de tamaño fijo de 100 posiciones para almacenar los empleados. Ahora, podríamos tener *una relación directa* entre el valor clave de cada empleado con el índice del arreglo. Así:



De esta forma es posible acceder directamente a la información de cada empleado conociendo el número de identificación de cada empleado. Por ejemplo, si se quiere acceder al empleado con identificación 3, simplemente se utilizaría la instrucción: arreglo[3].

Sin embargo, hacer esta correspondencia en la práctica no es posible, ya que generalmente los números de identificación son números largos como es el caso de los números de cédula. Entonces, *no sería posible crear un arreglo tan grande* para contener tal cantidad de elementos.

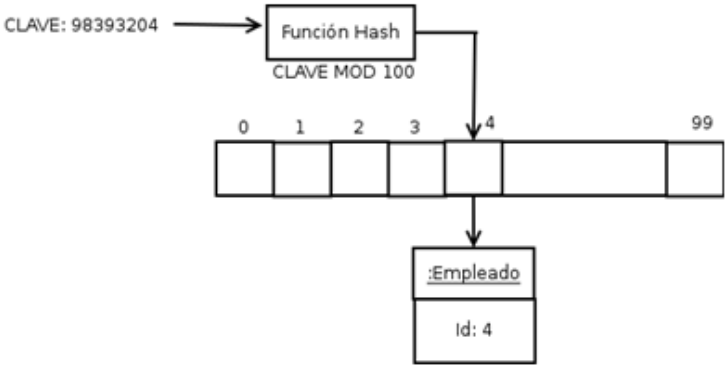
¿Qué alternativa entonces existe?

Una alternativa sencilla para no crear arreglos enormes, sería usar los dos últimos campos del *campo clave*. Por ejemplo, en el caso de la cédula utilizar los dos últimos campos. Así pues, si la cédula es 98393274, esta cédula se almacenaría en el arreglo en la posición 74.

Para ello, se necesita tener una función hash, que determine, a partir del campo clave, en qué posición del arreglo estará cada objeto. La función hash, determinará el método de acceso al arreglo.

En este caso, la función hash aplicará la operación: *CLAVE MOD 100*. Así por ejemplo, si la clave es 98393204, la función hash devolverá que se debe acceder a la posición 4 (98393204 MOD 100 = 4).

De la siguiente manera:



A continuación se da el código fuente en java de cómo sería la función hash:

```
public int funcionHash(int clave){
    return clave%100;
}
```

El problema de esta técnica son las colisiones, lo cual se explica a continuación.

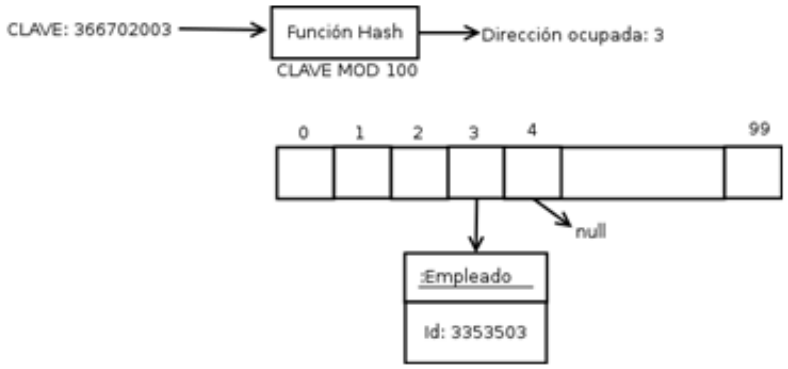
Colisiones

El esquema explicado anteriormente de la función hash, no garantiza *direcciones únicas (colisiones)*. Por ejemplo, para las claves 98393204 y 76236304, la función hash arrojaría la misma posición: 4.

Las colisiones son imposibles de evitar, lo que se debe tener en cuenta es que una buena función hash, debe minimizar las colisiones extendiendo los registros uniformemente a través de la tabla.

Existen varias alternativas para manejar las colisiones las cuales se explican a continuación.

Método 1: Hashing y búsqueda

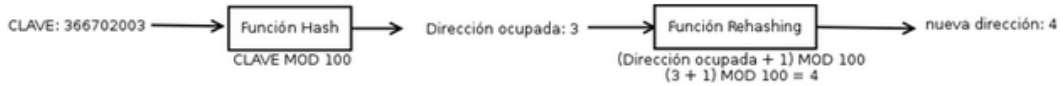


Esta técnica es sencilla, consiste en almacenar cada elemento colisionado en el siguiente espacio disponible. Por ejemplo, si queremos *almacenar nuevo registro* con la clave 366702003, al aplicar la función hash daría la posición 3. Suponiendo que esta ya esté ocupada, el algoritmo propone buscar el siguiente espacio disponible (donde haya *null*) para poder almacenar el dato. Si la búsqueda llega al final y están todas las posiciones ocupadas, se debe buscar desde el principio. La Figura 3, ilustra este ejemplo, donde se puede apreciar que la posición 3 devuelta por la función hash ya está ocupada, por lo tanto, el nuevo dato debería almacenarse en la siguiente posición libre, en este caso, la posición 4 (null).

Para *buscar un elemento* con esta técnica, se aplica la función hash sobre la clave, luego se compara la clave devuelta con la clave real. Si las claves no coinciden, se hace una búsqueda secuencial comenzando por la siguiente posición del array. Nuevamente, si se llega al final del vector, se sigue buscando desde el principio.

Método 2: Rehashing

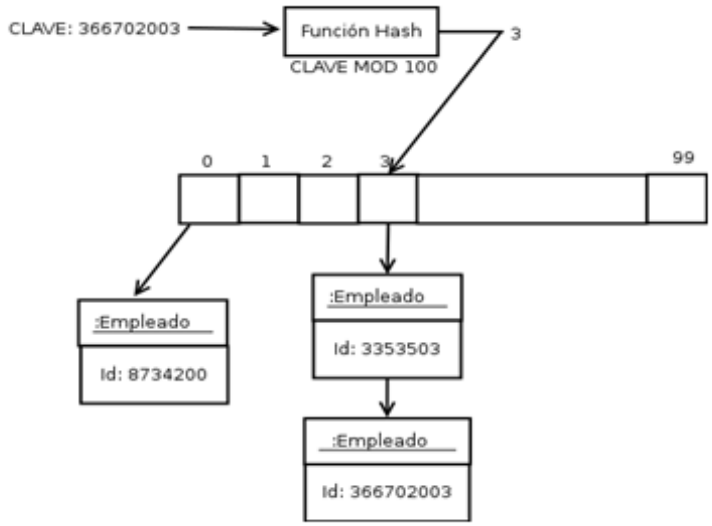
Si el primer cálculo de la función hash produce una colisión, se usa la dirección transformada como entrada para una función rehash y se calcula una nueva dirección. Un ejemplo de función rehash sencilla sería: *(Dirección ocupada + 1) mod 100*. Si el resultado sigue siendo un espacio ocupado, se puede aplicar la función rehashing hasta obtener un espacio disponible.



La clave 36670003 es pasada como entrada a la función hash, ésta arroja la dirección 3, la cual en el caso está ocupada; por lo tanto, tendría que pasar por la función rehashing. En este caso la función rehashing devolvería la nueva dirección 4. Si el valor 4, está ocupado, se aplica la función rehashing hasta obtener un espacio disponible.

Método 3: múltiples espacios

Esta tercer técnica, consiste en modificar la tabla de tal forma que en lugar de almacenar un objeto en cada posición, se pueda almacenar en cada espacio varios objetos. De esta forma, varias direcciones colisionadas arrojadas por la función hash se pueden almacenar en el mismo espacio.



Se puede apreciar que la función hash devuelve la dirección 3. Como en la posición 3 ya está almacenado con un objeto empleado con cédula: 3353503, entonces, simplemente el nuevo objeto se encadena a éste.

La idea general de usar la clave para determinar la dirección del registro es una excelente idea, pero se debe modificar de forma que no se desperdicie tanto espacio. Esta modificación se lleva a cabo mediante una función que transforma una clave en un índice de una tabla y que se denomina *función de Randomización o Hash*.

Si H es una función hash y X es un elemento a almacenar, entonces H(X) es la función hash del elemento y se corresponde con el índice donde se debe colocar X. En nuestro ejemplo, la función hash sería H(X)=X % 53 (función resto).

Los valores generados por H deben cubrir todo el conjunto de índices de la tabla. Además, el tamaño de la tabla debe ser un poco más grande que el número de elementos que han de ser insertados, aunque queden posiciones de la tabla sin uso.

El método anterior tiene una deficiencia: suponer que dos elementos X e Y son tales que H(X) = H(Y). Entonces, cuando un el elemento X entra en la tabla, éste se inserta en la posición dada por su función Hash, H(X). Pero cuando al elemento Y le es asignado su posición donde será insertado mediante la función hash, resulta que la posición que se obtiene es la misma que la del elemento X. Esta situación se denomina colisión o choque.

Una buena función Hash será aquella que *minimiza las colisiones, y que distribuya los elementos uniformemente a través del arreglo*. Esta es la razón por la que *el tamaño del arreglo debe ser un poco mayor que el número real de elementos a insertar*, pues cuanto más grande sea el rango de la función de randomización, es menos probable que dos claves generen el mismo valor de asignación o hash, es decir, que se asigne una misma posición a más de un elemento.

Habrá que llegar a un compromiso entre Eficiencia en Espacio-Tiempo: el dejar espacios vacíos en la tabla es una deficiencia en cuanto a espacio, mientras que reduce la necesidad de resolver los casos de choque en la asignación, y por lo tanto es más eficiente en términos de tiempo.

Una solución al problema de las colisiones: Zona de desbordamiento.

Se trata de mantener una zona reservada para aquellos elementos que llegan a colisionar, de manera que cuando se produzca una colisión el elemento se va a localizar en esta zona de desbordamiento.

Al realizar la búsqueda y comprobar si el elemento buscado está en la posición dada por su tabla hash, si esa posición ya está ocupada por otro elemento con el mismo valor de hashing, se seguirá buscando a partir del inicio de la zona de desbordamiento de manera secuencial, hasta encontrar el elemento o llegar al final de dicha zona de desbordamiento.

Siguiendo con el ejemplo, para cada código se obtenía el residuo (resto) de su división por un número primo (en este caso el 53). Es recomendable usar números primos ya que que reduce la probabilidad de colisión al establecer funciones del tipo: ***H(X) = (X% primo)***

Ahora bien, ¿cuál valor primo escoger? En este ejemplo se manejan 30 códigos, de modo que ese valor debe ser superior a 30, y que su magnitud se acerque al doble de elementos, por ello una buena opción era el 53 (o también el 61). Valores primos muy cercanos a la cantidad original de datos da lugar a demasiadas colisiones y un valor primo grande (por ejemplo, 97) da lugar a un considerable desperdicio de espacio (casillas vacías).

¿Cómo nos quedarían ubicados los elementos en el vector? Hay que tener en cuenta algo más: como el valor primo utilizado en la función de Hashing fue el 53, inicialmente el vector destino tendría 53 casillas, pero a eso hay que agregarle las casillas de la zona de desbordamiento, de modo que habría que aumentar esas casillas alrededor de un 25%, y en ese caso nuestro vector quedaría con 53 * 1.25 = 66.25, o sea 66 casillas en total. Las 53 primeras almacenarán los códigos según la posición que resulte al aplicar la función de Hashing, las 13 restantes representan la zona de desbordamiento a donde irán los códigos que colisionen.

Veamos un gráfico de cómo quedarían los códigos en ese vector:

0		33	104611010059
1		34	
2	104611010505	35	
3	46101046	36	
4		37	46082053
5	46101048	38	
6	104611010138	39	
7		40	104611024641
8	104611010776	41	46101031
9	46102006	42	
10	46092043	43	46102040
11		44	46102041
12	104611010038	45	
13		46	104611010178
14		47	
15	104611010253	48	
16	46091095	49	
17	46102014	50	46081165
18		51	46082014
19		52	
20		**53**	**46102030**
21	46092107	**54**	**46102043**
22	104611010843	**55**	**46092103**
23		**56**	
24	46102021	**57**	
25	1046110103	**58**	
26		**59**	
27		**60**	
28	46102025	**61**	
29		**62**	
30	46102080	**63**	
31	1046110109	**64**	
32		**65**	

2.5.1.4 Ejercicios Propuestos

1. Dado el siguiente arreglo:

2	3	18	20	22	25	30	100	200
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

Mostrar que elementos recorrería la búsqueda binaria hasta encontrar los valores:
a) 18
b) 3
c) 100
d) 25

2. Se tienen los siguientes valores:

40 50 62 103 204 304 328

almacenar los valores en una tabla hash con 50 posiciones, usando como función hash: CLAVE MOD 50, y como método de resolución de colisiones la función rehashing: (CLAVE + 1) MOD 50

2.5.2 Algoritmos de Ordenamiento

En computación y matemáticas un algoritmo de ordenamiento es un algoritmo que pone elementos de una lista o un vector en una secuencia dada por una relación de orden, es decir, el resultado de salida ha de ser una permutación —o reordenamiento— de la entrada que satisfaga la relación de orden dada. Las relaciones de orden más usadas son el orden numérico y el orden lexicográfico. Ordenamientos eficientes son importantes para optimizar el uso de otros algoritmos (como los de búsqueda y fusión) que requieren listas ordenadas para una ejecución rápida. También es útil para poner datos en forma canónica y para generar resultados legibles por humanos.

Desde los comienzos de la computación, el problema del ordenamiento ha atraído gran cantidad de investigación, tal vez debido a la complejidad de resolverlo eficientemente a pesar de su planteamiento simple y familiar. Por ejemplo, BubbleSort fue analizado desde 1956.1 Aunque muchos puedan considerarlo un problema resuelto, nuevos y útiles algoritmos de ordenamiento se siguen inventado hasta el día de hoy (por ejemplo, el ordenamiento de biblioteca se publicó por primera vez en el 2004). Los algoritmos de ordenamiento son comunes en las clases introductorias a la computación, donde la abundancia de algoritmos para el problema proporciona una gentil introducción a la variedad de conceptos núcleo de los algoritmos, como notación de O mayúscula, algoritmos divide y vencerás, estructuras de datos, análisis de los casos peor, mejor, y promedio, y límites inferiores. Fuente: Wikipedia

2.5.2.1. Ordenamiento Por Selección

El algoritmo de ordenamiento por selección se basa en la idea de tener dividido el arreglo que se está ordenando en dos partes: una, con un grupo de elementos ya ordenados, que ya encontraron su posición final. La otra, con los elementos que no han sido todavía ordenados. En cada iteración, localizamos el menor elemento de la parte no ordenada, lo intercambiamos con el primer elemento de esta misma región e indicamos que la parte ordenada ha aumentada en un elemento.

El algoritmo implementado en java es el siguiente:

```
public classOrdenamiento {
```

```
/**
 * Despues de la primera iteracion el elemento menor queda en la primera
 * posicion del arreglo.Para ello busca el menor valor de todo el arreglo y
 * lo intercambia con el que queda en la primera posicion. Luego repite el
 * mismo proceso pero comenzando desde el segundo elemento del arreglo,
 * busca el menor y lo intercambia con el primer elemento que se encuentra
 * en la parte sin ordenar. ORDEN (N*N)
*/

    public int[] Seleccion(int[] arregloSinOrdenar) {
        int[] arreglo = darCopiaValores(arregloSinOrdenar);
        int n = arreglo.length;
        // lleva la parte sin ordenar del arreglo
        for(inti = 0; i < n - 1; i++) {
            intmenor = arreglo[i];
            int posMenor = i;
            // busca el menor de la parte sin ordenar
            for(int j = i + 1; j < n; j++) {
                if(arreglo[j] < menor) {
                    menor = arreglo[j];
                    posMenor = j;
                }
            }
            // después de haber encontrado el dato lo intercambia con
            // el que se encuentra en la casilla i, para eso utiliza temp.
            int temp = arreglo[i];
            arreglo[i] = menor;
            arreglo[posMenor] = temp;
        }
        return arreglo;
    }

    /**
    *
    * Saca una copia del vector
    */
    public int[] darCopiaValores(int[] arreglo) {
        int[] arregloNuevo = new int[arreglo.length];
        for(int i = 0; i < arreglo.length; i++) {
          arregloNuevo[i] = arreglo[i];
        }
        return arregloNuevo;
    }
}
```

El programa principal que invoca este método de ordenamiento sería:

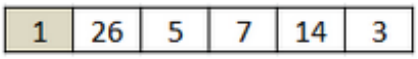
```
public class ClienteMain {
    public static voidmain(String[] args) {
        Ordenamiento o = newOrdenamiento();
        int[] arreglo = { 5,26,1,7,14,3};
        mostrar(arreglo);
        // Cambiar aqui por el algoritmo de ordenamiento a probar
        mostrar(o.seleccion(arreglo));
    }


    /**
    * Muestra por consola todos los datos del arreglo, separados por espacio
    *
    * @param arreglo
```

```
     * arreglo a imprimir
     */
     public static voidmostrar(int[] arreglo) {
        for(int i = 0; i < arreglo.length; i++) {
           System.out.print(arreglo[i] +"␣");
        }
        System.out.println("");
     }
}
```

El algoritmo anterior funciona de la siguiente manera:

Tenemos un arreglo de enteros no ordenados de 6 posiciones:

Después de la primera iteración del algoritmo, queremos que el menor elemento (1) quede en la primera posicion. para esto buscamos el menor valor de todo el arreglo y lo intercambiamos con el que esta en la primera posicion (5).



Luego, repetimos el mismo proceso pero comenzando desde el segundo elemento del arreglo. Buscamos el menor (3) y lo reemplazamos con el primer elemento que se encuentra en la parte sin ordenar (26).

El proceso continúa hasta que todo el arreglo quede ordenado.

Estudio de la complejidad:
Para el estudio de la complejidad se debe contar el número de comparaciones e intercambios que requiere la ordenación de n elementos. El procedimiento siempre realiza n-1 intercambios. Existen n-1 llamadas a intercambio. El bucle interno hace i-1 comparaciones cada vez; el bucle externo itera n-1 veces, de modo que el número total de comparaciones es $O(n^2)$, por lo que el número de comparaciones claves es cuadrático. Ha de observarse que el algoritmo no depende de la disposición inicial de los datos. Además, el número de intercambios del arreglo es lineal O(n).

2.5.2.2. Ordenamiento Burbuja

Burbuja u ordenamiento por intercambio, intercambia todo par de elementos consecutivos que no se encuentran en orden. Al final de cada pasada haciendo este intercambio, un nuevo elemento queda ordenado y todos los demás elementos se acercaron a su posición final. Se llama burbuja porque al final de cada iteración el mayor va surgiendo al final.

El algoritmo implementado en java es el siguiente:

```
public class Ordenamiento {

    /**
    * Burbuja u ordenamiento por intercambio
    * @param arreglo a ser ordenado
    * @return arreglo ordenado
    */
    public int[] burbuja(int[] arregloSinOrdenar) {
       int[] arreglo = darCopiaValores(arregloSinOrdenar);
       int n = arreglo.length;
       // controla el punto hasta el cual lleva el proceso de intercambio.
       // Termina cuando queda un solo elemento
       for (int i = n; i > 1; i--) {
          // lleva el proceso de intercambio empezando en 0 hasta llegar
          // al punto anterior al límite marcado por la variable i
          for(int j = 0; j < i - 1; j++) {
             if (arreglo[j] > arreglo[j + 1]) {
                int temp = arreglo[j];
                arreglo[j] = arreglo[j + 1];
                arreglo[j + 1] = temp;
             }
          }
       }
       return arreglo;
    }


    /**
    *
    * Saca una copia del vector
    */
    public int[] darCopiaValores(int[] arreglo) {
       int[] arregloNuevo = new int[arreglo.length];
       for(int i = 0; i < arreglo.length; i++) {
          arregloNuevo[i] = arreglo[i];
       }
       return arregloNuevo;
    }
}
```

El programa principal que invoca este método de ordenamiento sería:

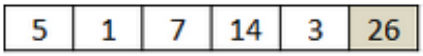
```
public classClienteMain {
    public static voidmain(String[] args) {
      Ordenamiento o = new Ordenamiento();
      int[] arreglo = {5,26,1,7,14,3};
      mostrar(arreglo);
      // Cambiar aquí por el algoritmo de ordenamiento a probar
      mostrar(o.burbuja(arreglo));
    }


    /**
    * Muestra por consola todos los datos del arreglo, separados por espacio
    *
    * @param arreglo
    * arreglo a imprimir
    */
    public static voidmostrar(int[] arreglo) {
```

```
        for(int i = 0; i < arreglo.length; i++) {
          System.out.print(arreglo[i] +"␣");
        }
        System.out.println("");
    }
}
```

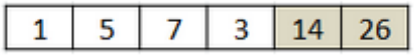
El algoritmo anterior funciona de la siguiente manera:

Tenemos un arreglo de enteros no ordenados de 6 posiciones:

En la primera iteración recorremos el arreglo de izquierda a derecha intercambiando todo par de valores consecutivos que no se encuentren ordenados. Al final de la iteración, el mayor de los elementos (26), debe quedar en la última posicion del arreglo, mientras todos los otros valores han avanzado un poco hacia su posicion final.



Este mismo proceso se repite para la parte del arreglo que todavia no esta ordenada.



En esta segunda iteración podemos ver que varios elementos del arreglo han cambiado de posicion mientras que el elemento mayor de la parte no ordenada a alcanzado su ubicación.

El proceso termina cuando solo quede un elemento en la zona no ordenada del arreglo, lo cual requiere 5 iteraciones en nuestro ejemplo.

Estudio de la complejidad

Para el estudio de la complejidad se debe contar el número de comparaciones e intercambios que requiere la ordenación de n elementos. El bucle externo itera n-1 veces (desde n hasta 1). El bucle interno hace i-1 comparaciones cada vez; de modo que el número total de comparaciones es $O(n^2)$.

2.5.2.3. Ordenamiento por inserción

Este método separa la secuencia en dos grupos: una parte con los valores ordenados (inicialmente con un solo elemento) y otra con los valores por ordenar (inicialmente todo el resto). Luego vamos pasando uno a uno los valores a la parte ordenada, asegurándose que se vayan colocando ascendentemente.

El algoritmo implementado en java es el siguiente:

```
public class Ordenamiento {
/**
 * Este metodo separa la secuencia en dos grupos: una parte con los valores
 * ordenados (inicialmente con un solo elemento) y otra con los valores por
```

```
   * ordenar (inicialmente todo el resto). Luego vamos pasando uno a uno los
   * valores a la parte ordenada, asegurandonos que se vayan colocando
   * ascendentemente.
   *
   * @param arregloSinOrdenar
   * arreglo sin ordenar. ORDEN (N*N)
   * @return arreglo ordenado
   */

     public int[] insercion(int[] arregloSinOrdenar) {
       int[] arreglo = darCopiaValores(arregloSinOrdenar);
       int n = arreglo.length;
       // i señala la posicion del elemento que va a insertar,
       // va desplazando hacia la izquierda, casilla a casilla,
       // el elemento que se encontraba inicialmente en i,
       // hasta que encuentra la posicion adecuada
       for (int i = 1; i < n; i++) {
         for(int j = i; j > 0 && arreglo[j - 1] > arreglo[j]; j--) {
           int temp = arreglo[j];
           arreglo[j] = arreglo[j - 1];
           arreglo[j - 1] = temp;
         }
       }
       return arreglo;
     }

}
```

El programa principal que invoca este método de ordenamiento sería:

```
public class ClienteMain {
    public static void main(String[] args) {
      Ordenamiento o = newOrdenamiento();
      int[] arreglo = { 5,26,1,7,14,3 };
      mostrar(arreglo);
      // Cambiar aquí por el algoritmo de ordenamiento a probar
      mostrar(o.insercion(arreglo));
    }
    /**
    * Muestra por consola todos los datos del arreglo, separados por espacio
    *
    * @param arreglo
    * arreglo a imprimir
    */
    public static void mostrar(int[] arreglo) {
      for (inti = 0; i < arreglo.length; i++) {
        System.out.print(arreglo[i] +"␣");
      }
      System.out.println("");
    }

}
```

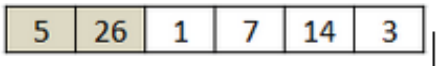
El algoritmo anterior funciona de la siguiente manera:
Tenemos un arreglo de enteros no ordenados de 6 posiciones:

5	26	1	7	14	3

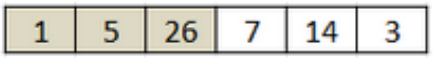
Al inicio, podemos suponer que en la parte ordenada hay un elemento (5), no está todavía en

su posición final, pero siempre es cierto que una secuencia de un solo elemento está ordenada.

luego tomamos uno a uno los elementos de la parte no ordenada y los vamos insertando ascendentemente en la parte ordenada, comenzamos con el (26) y vamos avanzando hasta insertar el (3).
primera iteración:



segunda iteración:



quinta y última iteración:

en esta iteración no hay movimientos ya que el último elemento ya esta ordenado.

Estudio de la complejidad:

El bucle externo se ejecuta n-1 veces. Dentro de este bucle existe otro bucle que se ejecuta a lo más el valor de i veces para valores de i que están en el rango de n-1 a 1. Por consiguiente, en el peor de los casos, las comparaciones del algoritmo vienen dadas por:

$$1 + 2 + \ldots + (n\text{-}1) = n(n\text{-}1)/2$$

Por otra parte, el algoritmo mueve los datos como máximo el mismo número de veces. Existe por lo tanto, en el peor de los casos, los siguiente movimientos:

$$n(n\text{-}1)/2$$

Por consiguiente, el algoritmo de ordenación por inserción es $O(n^2)$ en el caso peor.

2.5.2.4. Ordenamiento Shell

El método se denomina Shell en honor de su inventor Donald Shell. Realiza comparaciones entre elementos NO consecutivos, separados por una distancia salto. El valor salto al principio es n/2 y va decreciendo en cada iteración hasta llegar a valer 1. Cuando salto vale 1 se comparan elementos consecutivos. El valor se encontrará ordenado cuando salto valga 1 y no se puedan intercambiar elementos consecutivos porque están en orden.

El algoritmo implementado en java es el siguiente:

```
 public class Ordenamiento {
/**
* Realiza comparaciones entre elementos NO consecutivos, separados por una
* distancia salto El valor salto al principio es n/2 y va decreciendo en
* cada iteracion hasta llegar a valer 1. Cuando salto vale 1 se comparan
* elementos consecutivos. El valor se encontrara ordenado cuando salto
* valga 1 y no se puedan intercambiar elementos consecutivos porque estan
* en orden
*
* @param arregloSinOrdenar
* arreglo sin ordenar
* @return arreglo ordenado
*/
public int[] shell(int[] arregloSinOrdenar) {
```

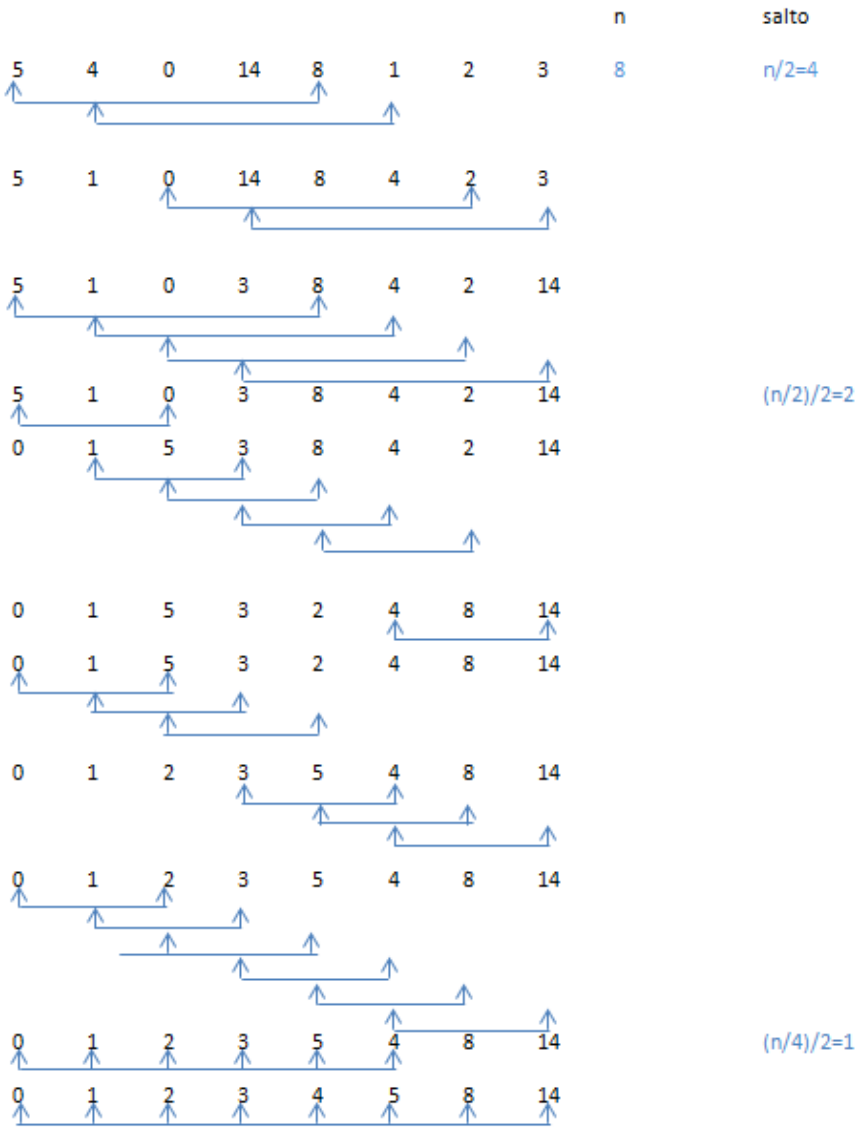
```
    int[] arreglo = darCopiaValores(arregloSinOrdenar);
    int n = arreglo.length;
    int salto = n;
    booleanordenado;
    while(salto > 1) {
      salto = salto / 2;
      do{
        ordenado = true;
        for(intj = 0; j <= n - 1 - salto; j++) {
          int k = j + salto;
          if(arreglo[j] > arreglo[k]) {
            intaux = arreglo[j];
            arreglo[j] = arreglo[k];
            arreglo[k] = aux;
            ordenado = false;
          }
        }
      } while (!ordenado);
    }
    return arreglo;

}
```

El programa principal que invoca este método de ordenamiento sería:

```
 public class ClienteMain {
    public static void main(String[] args) {
      Ordenamiento o = newOrdenamiento();
      int[] arreglo = {5, 4, 0, 14, 8, 1, 2, 3};
      mostrar(arreglo);
      // Cambiar aquí por el algoritmo de ordenamiento a probar
      mostrar(o.shell(arreglo));
    }

    /**
    * Muestra por consola todos los datos del arreglo, separados por espacio
    * @param arreglo
    * arreglo a imprimir
    */
    public static voidmostrar(int[] arreglo) {
      for(int i = 0; i < arreglo.length; i++) {
        System.out.print(arreglo[i] +"␣");
      }
      System.out.println("");
    }

}
```

La siguiente gráfica muestra un seguimiento del método de ordenación Shell para los datos de entrada:

5, 4, 0, 14, 8, 1, 2, 3.



El proceso termina cuando salto=1 y no hay ningún cambio en la iteración.

Estudio de la complejidad

Su implementación original, requiere O(n2) comparaciones e intercambios en el peor caso. Un cambio menor presentado en el libro de V. Pratt produce una implementación con un rendimiento de O(n log2 n) en el peor caso. Aunque es fácil desarrollar un sentido intuitivo de cómo funciona este algoritmo, es muy difícil analizar su tiempo de ejecución.

El Shell sort es una generalización del ordenamiento por inserción, teniendo en cuenta dos observaciones:

1. El ordenamiento por inserción es eficiente si la entrada está çasi ordenada".
2. El ordenamiento por inserción es ineficiente, en general, porque mueve los valores sólo una posición cada vez.

El algoritmo Shell sort mejora el ordenamiento por inserción comparando elementos separados por un espacio de varias posiciones. Esto permite que un elemento haga “pasos más grandes” hacia su posición esperada. Los pasos múltiples sobre los datos se hacen con tamaños de espacio cada vez más pequeños.

El último paso del Shell sort es un simple ordenamiento por inserción, pero para entonces, ya está garantizado que los datos del vector están casi ordenados.

2.5.2.5. Merge Sort

Este método de ordenación divide el vector por la posición central, ordena cada una de las mitades y después realiza la mezcla ordenada de las dos mitades. El caso base es aquel que recibe un vector con ningún elemento, o con 1 solo elemento, ya que obviamente está ordenado.

El algoritmo implementado en java es el siguiente:

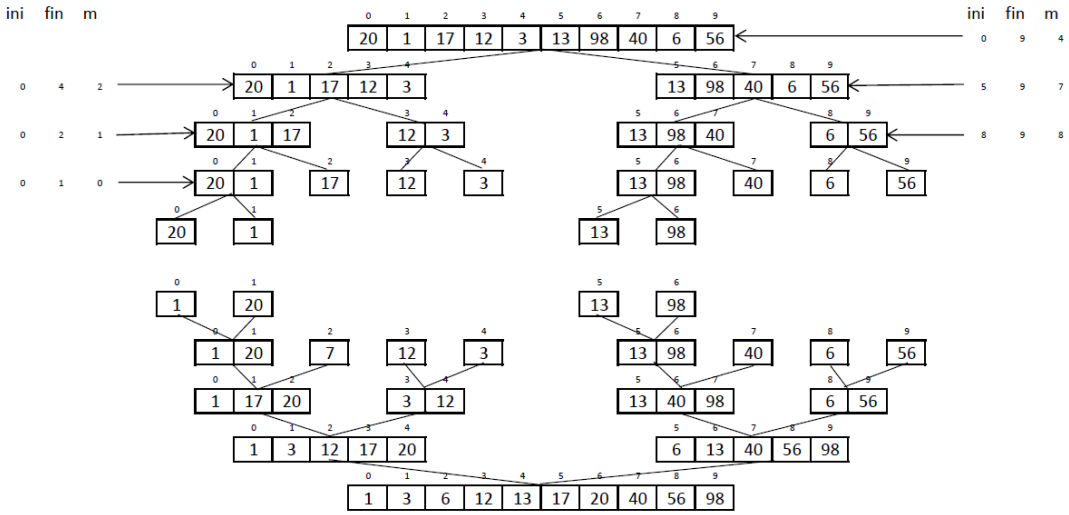
```
public class Ordenamiento {
  /**
  * Este m todo de ordenaci n divide el vector por la posici n central,
  * ordena cada una de las mitades y despu s realiza la mezcla ordenada
      de
  * las dos mitades. El caso base es aquel que recibe un vector con
      ning n
  * elemento, o con 1 solo elemento, ya que obviamente est  ordenado.
      ORDEN
  * (N LOG N)
  *
  * @param arreglo
  * , arreglo de elementos
  * @param ini
  * , posicion inicial
  * @param fin
  * , posici n final
  */
  public void mergeSort(int arreglo[], int ini, int fin) {
    int m = 0;
    if(ini < fin) {
      m = (ini + fin) / 2;
      mergeSort(arreglo, ini, m);
      mergeSort(arreglo, m + 1, fin);
      merge(arreglo, ini, m, fin);
    }
  }
  /**
  * Une el arreglo en uno solo. O MezclaLista
  *
  * @param arreglo
  * arreglo de elementos
  * @param ini
  * posici n inicial
  * @param m
  * posici n intermedia
  * @param fin
  * posici n final
  */
  private void merge(int arreglo[], int ini, int m, int fin) {
    int k = 0;
    int i = ini;
    int j = m + 1;
    int n = fin - ini + 1;
    intb[] = new int[n];
    while(i <= m && j <= fin) {
      if (arreglo[i] < arreglo[j]) {
        b[k] = arreglo[i];
        i++;
        k++;
      }else {
```

```
            b[k] = arreglo[j];
            j++;
            k++;
          }
        }
        while(i<= m) {
          b[k] = arreglo[i];
          i++;
          k++;
        }
        while(j<= fin) {
          b[k] = arreglo[j];
          j++;
          k++;
        }

        for(k=0; k<n; k++) {
          arreglo[ini + k] = b[k];
        }
      }

      /**
      * Toma los datos para invocar al algoritmo mergeSort recursivo
      * @param arregloSinOrdenar
      * @return
      */

      public int[] mergeSortTomaDatos(intarregloSinOrdenar[]) {
        int[] arreglo = darCopiaValores(arregloSinOrdenar);
        intn = arreglo.length;
        mergeSort(arreglo, 0, arreglo.length - 1);
        return arreglo;
      }

}
```

El programa principal que invoca este método de ordenamiento sería:

```
public classClienteMain {
    public static voidmain(String[] args) {
      Ordenamiento o =newOrdenamiento();
      int[] arreglo = {20,1,17,12,3,13,98,40,6,56};
      mostrar(arreglo);
      // Cambiar aquí por el algoritmo de ordenamiento a probar
      mostrar(o.mergeSortTomaDatos(arreglo));
    }
    /**
    * Muestra por consola todos los datos del arreglo, separados por espacio
    *
    * @param arreglo
    * arreglo a imprimir
    */
    public static voidmostrar(int[] arreglo) {
      for(inti = 0; i < arreglo.length; i++) {
        System.out.print(arreglo[i] +"␣");
      }
      System.out.println("");

    }

}
```

El algoritmo anterior funciona de la siguiente manera:
Tenemos un arreglo de enteros no ordenados de 10 posiciones:



Estudio de la complejidad

El algoritmo es recursivo esa razón que se quiere determinar el tiempo empleado por cada una de las 3 fases del algoritmo divide y vencerás. Cuando se llama a la función mezclista se deben mezclar las dos listas mas pequeñas en una nueva lista con n elementos. la función hace una pasada a cada una de las sublistas. Por consiguiente, el número de operaciones realizadas será como máximo el producto de una constante multiplicada por n. si se consideran las llamadas recursivas se tendrá entonces el número de operaciones: constante *n* (profundidad de llamadas recursivas). El tamaño de la lista a ordenar se divide por dos en cada llamada recursiva, de modo que el número de llamadas es aproximadamente igual al número de veces que nse puede dividir por 2, parándose cuando el resultado es menor o igual a 1.

Por consiguiente, la ordenación por mezcla se ejecutará, aproximadamente, el número de operaciones: alguna constante multiplicada por n y después por (logn), Resumiendo hay dos llamadas recursivas, y una iteración que tarda tiempo n por lo tanto la recurrencia es:

T(n) = n + 2T(n/2) si n>1 y 1 en otro caso. De esta forma, aplicando expansión de recurrencia se tiene: T(n) = n + 2T(n/2) = n +2 n/2+4T(n/4) =..... = n+n+n+ +... ..n(k= logn veces)=n*logn O(n) por lo tanto la ordenación por mezcla tiene un tiempo de ejecución de O(n*logn)

2.5.2.6. Quicksort

Este algoritmo divide en array en dos subarray, que se pueden ordenar de modo independiente. Se selecciona un elemento específico del array arreglo[centro] llamado pivote. Luego, se debe re situar los demás elementos de la lista a cada lado del pivote, de manera que a un lado queden todos los menores que él, y al otro los mayores. Los elementos iguales al pivote pueden ser colocados tanto a su derecha como a su izquierda, dependiendo de la implementación deseada. En este momento, el pivote ocupa exactamente el lugar que le corresponderá en la lista ordenada. Finalmente, se divide el array original en dos subarrays que se ordenarán de modo independiente mediante llamadas recursivas del algoritmo.

El algoritmo implementado en java es el siguiente:

```
 public classOrdenamiento {
/**
*
* Este algoritmo divide en array en dos subarray, que se pueden ordenar de
* modo independiente. Se selecciona un elemento especifico del array
```

```
* arreglo[centro] llamado pivote y se divide el array original en dos
* subarrays que se ordenaran de modo independiente mediante llamadas
* recursivas del algoritmo.
*
* @param arreglo
* arreglo que se esta ordenando
* @param inicio
*posici n de inicio - ORDEN(N LOG N) EN EL CASO MEDIO.
* ORDEN(N*N) EN EL PEOR CASO
* @param fin
* posicion final
*/
public voidquickSort(int[] arreglo, intinicio,intfin) {
    inti = inicio;// i siempre avanza en el arreglo hacia la derecha
    intj = fin; // j siempre avanza hacia la izquierda
    intpivote = arreglo[(inicio + fin) / 2];
    do{
      while (arreglo[i] < pivote)
        // si ya esta ordenado incrementa i
        i++;
      while (pivote < arreglo[j])
        // si ya esta ordenado decrementa j
        j--;
      if(i <= j) {// Hace el intercambio
        int aux = arreglo[i];
        arreglo[i] = arreglo[j];
        arreglo[j] = aux;
        i++;
        j--;
      }
    }
    while (i <= j);

    if (inicio < j)
      quickSort(arreglo, inicio, j);// invocación recursiva
    if(i < fin)
      quickSort(arreglo, i, fin);// invocacion recursiva
}
/**
*M todo intermediario que invoca al algoritmo quickSort recursivo
*
* @param arregloSinOrdenar
* arreglo sin ordenar
* @return arreglo ordenado
*/
public int[] quickSortTomaDatos(int[] arregloSinOrdenar) {
int[] arreglo = darCopiaValores(arregloSinOrdenar);
quickSort(arreglo, 0, arreglo.length- 1);
return arreglo;
}
/**
*
* Saca una copia del vector
*/
public int[] darCopiaValores(int[] arreglo) {
int[] arregloNuevo = new int[arreglo.length];
for(inti = 0; i < arreglo.length; i++) {
arregloNuevo[i] = arreglo[i];
}return arregloNuevo;
}
}
```

El programa principal que invoca este método de ordenamiento sería:

```
public classClienteMain {
    public static voidmain(String[] args) {
      Ordenamiento o = newOrdenamiento();
      int[] arreglo = { 1, 5, 7, 9, 6, 10, 3, 2, 4, 8 };
      mostrar(arreglo);
      // Cambiar aqui por el algoritmo de ordenamiento a probar
      mostrar(o.quickSortTomaDatos(arreglo));
    }
    /**
    * Muestra por consola todos los datos del arreglo, separados por espacio
    *
    * @param arreglo
    * arreglo a imprimir
    */
    public static voidmostrar(int[] arreglo) {
      for(inti = 0; i < arreglo.length; i++) {
        System.out.print(arreglo[i] + "␣");
      }
      System.out.println("");
    }

}
```

Estudio de la Complejidad

Estabilidad: NO es estable.
Requerimientos de Memoria: No requiere memoria adicional en su forma recursiva.
Tiempo de Ejecución:
***Caso promedio**. La complejidad para dividir una lista de n es O(n). Cada sublista genera en promedio dos sublistas más de largo n/2. Por lo tanto la complejidad se define en forma recurrente como:
f(1) = 1
f(n) = n + 2 f(n/2)
La forma cerrada de esta expresión es: f(n) = n log2n Es decir, la complejidad es O(n log2n).
***El peor caso** ocurre cuando la lista ya está ordenada, porque cada llamada genera sólo una sublista (todos los elementos son menores que el elemento de división). En este caso el rendimiento se degrada a O(n2). Con las optimizaciones mencionadas arriba puede evitarse este comportamiento.
Ventajas:
El más rápido
No requiere memoria adicional
Desventajas:
Implementación un poco más complicada.
Mucha diferencia entre el peor caso (n2) y el mejor caso (log n).
Intercambia registros iguales.

Diversos estudios realizados sobre el comportamiento del algoritmo Quicksort, demuestran que si se escoge en cada pasada ele elemento que ocupa la posición central del conjunto de datos a analizar, el número de pasadas necesarias para ordenar el arreglo es del orden de log n.

Respecto al número de comparaciones, si el tamaño del arreglo es una potencia de 2, en la primera pasada realizará (n-1) comparaciones, en la segunda pasada realizará (n-1)/2 compara-

ciones pero en dos conjuntos diferentes, en la tercera pasada realizará (n-1)/4 comparaciones diferentes pero en cuatro conjuntos diferentes y así sucesivamente.

Por lo tanto: c=(n-1)+(n-1)+(n-1)+.....+(n-1).

Si se considera a cada uno de los componentes de la sumatoria como un término y el número de términos de la sumatoria es igual a m, entonces se tiene que: c=(n-1)*m.

Considerando que el número de términos de la sumatoria(m) es igual al número de pasadas, y que éste es igual a (log n), la expresión anterior queda: c=(n-1)*log n.

2.5.2.7. Ejercicios Propuestos

1. Dada la siguiente secuencia: (20, 1, 17, 12, 3, 13, 98, 40, 6, 56), mostrar el proceso de ordenamiento de los siguientes métodos (pruebas de escritorio):
a) Ordenamiento por selección
b) Burbuja
c) Ordenación por inserción
d) Shell
e) QuickSort
f) MergeSort
2. Cuando todos los elementos son iguales, ¿cuál es el tiempo de ejecución de los siguientes métodos?
a) Inserción
b) Shell
c) MergeSort
d) QuickSort
3. Cuando la entrada ya está ordenada, ¿cuál es el tiempo de ejecución de los siguientes métodos?
a) Inserción
b) Shell
c) MergeSort
d) QuickSort
4. Cuando la entrada está originalmente ordenada, pero en orden inverso, ¿cuál es el tiempo de ejecución de los siguientes métodos:
a) Inserción
b) Shell
c) MergeSort
d) QuickSort
5. A continuación se describen los pasos de otro algoritmo de ordenamiento:
a) Se recorre el vector

Primero se compara V[i] con V[i+1] para todos los valores de i pares,

Luego se compara V[i] con V[i+1] para todos los valores de i impares

Cada vez que V[i] es mayor que V[i+1] se intercambian los valores.

b) Se continúa en forma alterna hasta que el conjunto de datos esté ordenado. El algoritmo se apoya en una variable de control, de modo que si no hubo intercambios en todo el recorrido, significa que ya está ordenado y terminará.

Implementar en un lenguaje de programación el anterior algoritmo.

3 — Algoritmos de Búsqueda

3.1 Introducción a los algoritmos de búsqueda

El ser humano desarrolla un sinnúmero de actividades, muchas de las cuales requieren que la recopilación de elementos que en ellas se emplean esten ordenados de una determinada manera. Una empresa, por ejemplo, constantemente necesita realizar búsquedas relacionadas con los datos de sus empleados o clientes; buscar información de un elemento en una lista.

El problema de la búsqueda radica en la **recuperación de la información lo más rápidamente posible**. Consiste en localizar un elemento en una lista o secuencia de elementos.

La operación de búsqueda puede llevarse a cabo sobre elementos ordenados o sobre elementos desordenados. En el primer caso, la búsqueda se facilita, y por lo tanto se ocupará menos tiempo que si se trabaja con elementos desordenados.

Los métodos de búsqueda pueden clasificarse en internos y externos, dependiendo el lugar donde estén almacenados los datos sobre los cuales se hará la búsqueda. Se denomina:

- **Búsqueda interna** si todos los elementos se encuentran en memoria principal (por ejemplo, almacenados en arreglos, vectores o listas enlazadas).
- **Búsqueda externa** si los elementos se encuentran en memoria secundaria. (Ya sea disco duro, disquete, cintas magnéticas, CD's, memorias flash).

A continuación nos centraremos en la búsqueda interna. Así, la organización de los datos sobre la que resolveremos el problema de la búsqueda consiste en un arreglo de **n** elementos de tipo elemento (valores puntuales, registros, etc.); Es necesario que los **n** elementos sean distintos. Si existen elementos iguales, al realizar la búsqueda se localiza únicamente uno de ellos, pudiendo dar lugar a errores al no contener la información correcta que se quería recuperar.

Cada algoritmo de búsqueda procura localizar en un arreglo un elemento X. Una vez finalizada la búsqueda puede suceder:

- Que la búsqueda haya tenido éxito, habiendo localizado la posición donde estaba almacenado el elemento X.
- Que la búsqueda no haya tenido éxito, concluyendo que no existía ningún elemento X.

Después de la búsqueda sin éxito, a veces es interesante introducir el elemento X. Un algoritmo con tal objetivo se denomina de **búsqueda e inserción**.

3.2 Búsqueda Secuencial

3.2.1 Conceptos

La búsqueda secuencial o lineal, consiste en recorrer y examinar cada uno de los elementos del arreglo, mediante un bucle voraz de izquierda a derecha, hasta encontrar el o los elementos buscados, o hasta que se han evaluado todos los elementos del arreglo.

El algoritmo implementado en java es el siguiente:

```
public class Busqueda {
```

```
/*** Busca secuencialmente un valor en el arreglo
* ORDEN(N) EN EL PEOR CASO
* @param valor
* valor a buscar
* @param arreglo
* arreglo de datos en cualquier orden
* @return true si lo encuentra, false si no encuentra el valor
*/
public boolean buscarSecuencial(int valor, int[] arreglo) {
    boolean encontrado = false;
    int i = 0;
    int n = arreglo.length;
    while (i < n && !encontrado) {
      if (arreglo[i] == valor)
        encontrado = true;
      else {
        i++;
      }
      return encontrado;
    }

}
```

El programa principal que invoca este método de búsqueda sería:

```
public class ClienteMain {
    public static void main(String[] args) {
      Busqueda b = new Busqueda();
      int[] arreglo = { 1, 2, 3, 4, 5, 6, 8, 9 };
      System.out.println(b.secuencial(5, arreglo));
    }

}
```

3.2.2 Complejidad computacional de la búsqueda secuencial

El **mejor caso**, se produce cuando el elemento a buscado sea el primero que se examina, de modo ques sólo se necesita una comparación. En el **peor caso,** el elemento deseado es el último que se examina, de modo ques se necesitan ***n***comparaciones. En el **caso medio,** se encontrará el elemento deseado aproximadamente en el centro de la colección, haciendo ***n/2*** comparaciones

Su complejidad es:

$$T(n) = \sum_{i=1}^{n} 1 = \varepsilon\ O(n)$$

Concluimos entonces que:

- Para el mejor caso **T(n) = O(1)**
- Para el caso Promedio y Peor caso **T(n) = O(n)**

3.3 Búsqueda Binaria

3.3.1 Introducción

Si los elementos sobre los que se realiza la búsqueda están ordenados, entonces podemos utilizar un algoritmo de búsqueda mucho más rápido que el secuencial: la **búsqueda binaria**. Consiste en reducir en cada paso el ámbito de búsqueda a la mitad de los elementos, basándose

en comparar el elemento a buscar con el elemento que se encuentra en la mitad del intervalo y con base en esta comparación:

- Si el elemento buscado es **menor** que el elemento medio, entonces sabemos que el elemento está en la mitad inferior de la tabla.
- Si es **mayor** es porque el elemento está en la mitad superior.
- Si es **igual** se finaliza con éxito la búsqueda ya que se ha encontrado el elemento.

Si se vuelve a repetir el mismo proceso anterior con la parte del arreglo que no hemos descartado, iremos avanzando rápidamente hacia el valor que queremos localizar. Puede darse el caso en el que el sub-arreglo a dividir está vacío y aún no se a encontrado el elemento. Sobre entendemos que el valor buscado no existe en el arreglo.

El algoritmo implementado en java es el siguiente:

```
 public class Busqueda {
/*ORDEN (LOG N) EN EL PEOR CASO Y CASO MEDIO. ORDEN(1) EN EL MEJOR *CASO
* @param valor
* valor que estamos buscando
* @param arreglo
* arreglo ordenado de datos
* @return true cuando lo encuentra, false cuando no encuentra el dato a
* buscar
*/
public boolean binaria(int valor, int[] arreglo) {
    boolean encontrado = false;
    int inicio = 0;
    int fin = arreglo.length - 1;
    while (inicio <= fin && !encontrado) {
      int medio = (inicio + fin) / 2;
      if (arreglo[medio] == valor) {
        encontrado = true;
      }
      else {
        if (arreglo[medio] > valor)
          fin = medio - 1;
        else
          inicio = medio + 1;
      }
    }
    return encontrado;

}
```

El programa principal que invoca este método de búsqueda sería:

```
 public class ClienteMain {
    public static void main(String[] args) {
      Busqueda b = new Busqueda();
      int[] arreglo = { 1, 2, 3, 4, 5, 6, 8, 9 };
      System.out.println(b.binaria(5, arreglo));
    }

}
```

3.3.2 Complejidad de la búsqueda binaria

El algoritmo determina en qué mitad está el elemento y descarta la otra mitad. En cada división, el algoritmo hace una comparación. El número de comparaciones es igual al número de veces que el algoritmo divide el *array* por la mitad. Si se supone que ***n*** es aproximadamente igual

a 2k entonces *k ó k + 1* es el número de veces que ***n*** se puede dividir hasta tener un elemento encuadrado en (k = log2 n). Su función de complejidad es:

$$T(n) = \sum_{i=1}^{\log(n)} 1 = \log_2(n)\ \varepsilon\ O(\log_2(n))$$

Por consiguiente el algoritmo es O(log2n) en el peor de los casos. En el caso medio O(log2n) en el caso medio y *O(1)* en el mejor de los casos.

En general, este método realiza **log2 (n)** comparaciones antes de encontrar el elemento, o antes de descubrir que no está. Este número es muy inferior que el necesario para la búsqueda lineal para casos grandes.

3.3.3 Versión recursiva de la búsqueda binaria

Su única novedad es que después de comparar el elemento de búsqueda con el elemento de la mitad de la tabla, se invoca recursivamente a realizar la búsqueda en uno de los dos posibles intervalos, el inferior o el superior, finalizando en el momento en el que se encuentre el elemento o ya se tenga un único elemento y no coincida con el buscado.

```
 public int BinariaRecursiva(int [] A,  int X, int fin, int inicio)
{
        int medio;
        if (inicio > fin) return -1;
        else{
          medio = (inicio + fin) / 2;
          if (A[medio] > X)
            return BinariaRecursiva(A, X, medio+1, fin);
          else
            if (A[medio] > X)
              return BinariaRecursiva(A, X, inicio, medio -1);
            else
              return medio;
    }
}
```

3.3.4 Complejidad de la búsqueda binaria recursiva

Para medir la velocidad de cálculo del algoritmo de búsqueda binaria, se deberán obtener el número de comparaciones que realiza el algoritmo, es decir, el número de vueltas del ciclo o el número de recursiones. Aunque en principio puede parecer que ambas versiones invierten el mismo tiempo, la recursiva es más lenta a medida que se incrementa el número de elementos, ya que existirán más llamadas a la función por resolver, con el consiguiente gasto de tiempo de guardar y restaurar parámetros.

En el mejor caso, la búsqueda binaria podría toparse con el elemento buscado en el primer punto medio, requiriéndose sólo una comparación de elementos. Esto equivale al mejor caso durante una búsqueda secuencial, pero en el peor de los casos la búsqueda binaria es mucho más rápida cuando N es grande.

El algoritmo de búsqueda binaria progresivamente va disminuyendo el número de elementos sobre el que realizar la búsqueda a la mitad: **n, (n/2), (n/4), ...** Así, tras **log2(N)** divisiones se habrá localizado el elemento o se tendrá la seguridad de que no estaba.

Mejor Caso: En sus casos óptimos, tanto la búsqueda secuencial como la binaria requieren sólo una comparación; esto significa que sus tiempos de ejecución óptimos no dependen de la cantidad de datos: son constantes y por tanto proporcionales a **1**, es decir, son de ***O(1).***

Peor Caso: En el peor caso, la búsqueda secuencial y la binaria sí dependen de **N**. La primera recorre todo el arreglo, requiriendo un tiempo de ***O(n);*** la binaria divide el arreglo, requiriendo sólo un tiempo ***O(log n).***

3.4 Búsqueda Hash

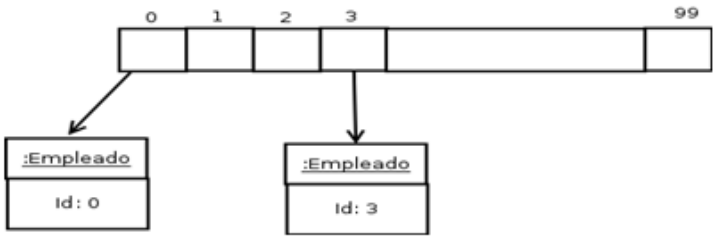
3.4.1 Introducción

La búsqueda binaria proporciona un medio para reducir el tiempo requerido para buscar en una lista. Este método, sin embargo, exige que los datos estén ordenados y siempre depende del número n de elementos que conforman el conjunto de datos.

Surge entonces una pregunta,***¿es posible lograr una búsqueda de O(1)?,*** es decir, una búsqueda que tome el mismo tiempo para buscar cualquier elemento de una lista. La respuesta es sí. Para ello se utiliza la técnica hashing.Este método se conoce como transformación de claves (clave-dirección) y consiste en convertir el elemento almacenado (numérico o alfanumérico) en una dirección (índice) dentro de un arreglo, de manera que se puede acceder al elemento directamente.

Ejemplo.

Vamos a partir de un sencillo ejemplo. Supongamos que tenemos una lista de empleados de una pequeña empresa. Cada empleado tiene asignado un número de identificación de 0 hasta 99. Entonces se necesitaría un vector de tamaño fijo de 100 posiciones para almacenar los empleados. Ahora, podríamos tener *una relación directa* entre el valor clave de cada empleado con el índice del arreglo. Así:



De esta forma es posible acceder directamente a la información de cada empleado conociendo el número de identificación de cada empleado. Por ejemplo, si se quiere acceder al empleado con identificación 3, simplemente se utilizaría la instrucción: arreglo[3].

Sin embargo, hacer esta correspondencia en la práctica no es posible, ya que generalmente los números de identificación son números largos como es el caso de los números de cédula. Entonces, *no sería posible crear un arreglo tan grande* para contener tal cantidad de elementos.

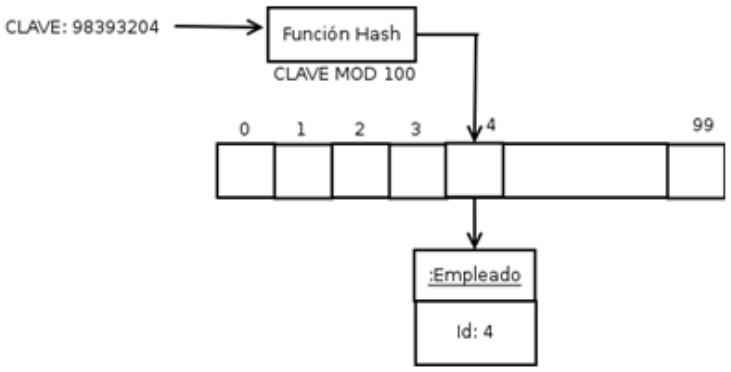
¿Qué alternativa entonces existe?

Una alternativa sencilla para no crear arreglos enormes, sería usar los dos últimos campos del *campo clave*. Por ejemplo, en el caso de la cédula utilizar los dos últimos campos. Así pues, si la cédula es 98393274, esta cédula se almacenaría en el arreglo en la posición 74.

Para ello, se necesita tener una función hash, que determine, a partir del campo clave, en qué posición del arreglo estará cada objeto. La función hash, determinará el método de acceso al arreglo.

En este caso, la función hash aplicará la operación: *CLAVE MOD 100*. Así por ejemplo, si la clave es 98393204, la función hash devolverá que se debe acceder a la posición 4 (98393204 MOD 100 = 4).

De la siguiente manera:



A continuación se da el código fuente en java de cómo sería la función hash:

```
public int funcionHash(int clave){
    return clave%100;
}
```

El problema de esta técnica son las colisiones, lo cual se explica a continuación.

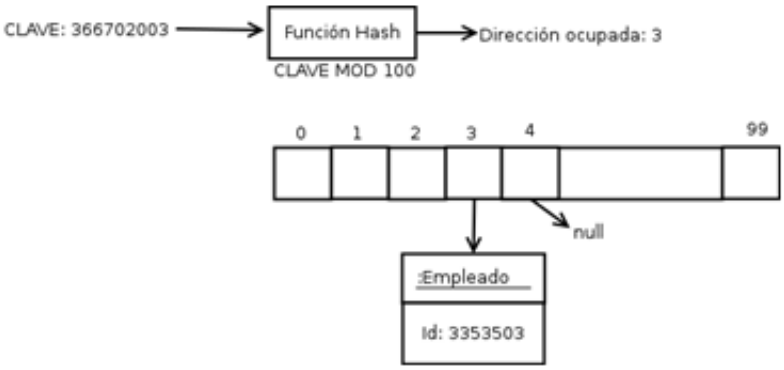
3.4.2 Colisiones

El esquema explicado anteriormente de la función hash, no garantiza *direcciones únicas (colisiones)*. Por ejemplo, para las claves 98393204 y 76236304, la función hash arrojaría la misma posición: 4.

Las colisiones son imposibles de evitar, lo que se debe tener en cuenta es que una buena función hash, debe minimizar las colisiones extendiendo los registros uniformemente a través de la tabla.

Existen varias alternativas para manejar las colisiones. Las cuales se explican a continuación.

Método 1: Hashing y búsqueda



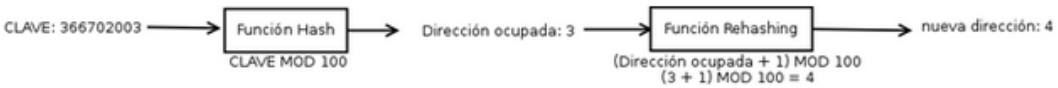
Esta técnica es sencilla, consiste en almacenar cada elemento colisionado en el siguiente espacio disponible. Por ejemplo, si queremos *almacenar nuevo registro* con la clave 366702003, al aplicar la función hash daría la posición 3. Suponiendo que esta ya esté ocupada, el algoritmo propone buscar el siguiente espacio disponible (donde haya *null*) para poder almacenar el dato. Si la búsqueda llega al final y están todas las posiciones ocupadas, se debe buscar desde el principio.

La Figura 3, ilustra este ejemplo, donde se puede apreciar que la posición 3 devuelta por la función hash ya está ocupada, por lo tanto, el nuevo dato debería almacenarse en la siguiente posición libre, en este caso, la posición 4 (null).

Para *buscar un elemento* con esta técnica, se aplica la función hash sobre la clave, luego se compara la clave devuelta con la clave real. Si las claves no coinciden, se hace una búsqueda secuencial comenzando por la siguiente posición del array. Nuevamente, si se llega al final del vector, se sigue buscando desde el principio.

Método 2: Rehashing

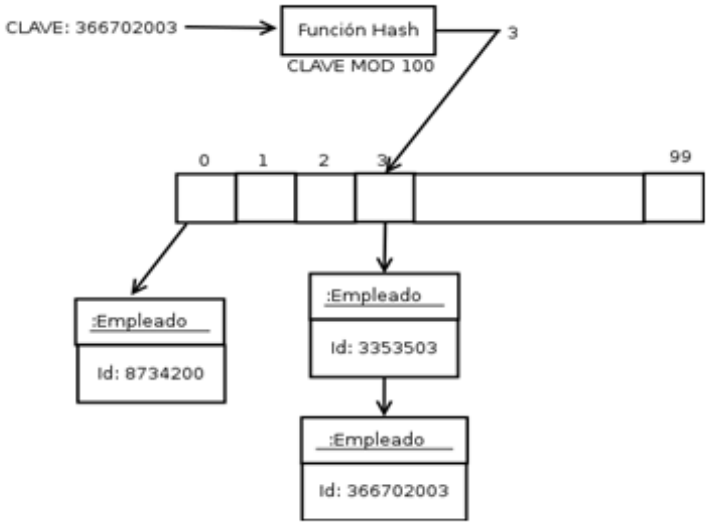
Si el primer cálculo de la función hash produce una colisión, se usa la dirección transformada como entrada para una función rehash y se calcula una nueva dirección. Un ejemplo de función rehash sencilla sería: *(Dirección ocupada + 1) mod 100*. Si el resultado sigue siendo un espacio ocupado, se puede aplicar la función rehashing hasta obtener un espacio disponible.



La clave 36670003 es pasada como entrada a la función hash, ésta arroja la dirección 3, la cual en el caso está ocupada; por lo tanto, tendría que pasar por la función rehashing. En este caso la función rehashing devolvería la nueva dirección 4. Si el valor 4, está ocupado, se aplica la función rehashing hasta obtener un espacio disponible.

Método 3: múltiples espacios

Esta tercer técnica, consiste en modificar la tabla de tal forma que en lugar de almacenar un objeto en cada posición, se pueda almacenar en cada espacio varios objetos. De esta forma, varias direcciones colisionadas arrojadas por la función hash se pueden almacenar en el mismo espacio.



Se puede apreciar que la función hash devuelve la dirección 3. Como en la posición 3 ya está almacenado con un objeto empleado con cédula: 3353503, entonces, simplemente el nuevo objeto se encadena a éste.

La idea general de usar la clave para determinar la dirección del registro es una excelente idea, pero se debe modificar de forma que no se desperdicie tanto espacio. Esta modificación se lleva a cabo mediante una función que transforma una clave en un índice de una tabla y que se denomina *función de Randomización o Hash*.

Si H es una función hash y X es un elemento a almacenar, entonces H(X) es la función hash del elemento y se corresponde con el índice donde se debe colocar X. En nuestro ejemplo, la función hash sería H(X)=X % 53 (función resto).

Los valores generados por H deben cubrir todo el conjunto de índices de la tabla. Además, el tamaño de la tabla debe ser un poco más grande que el número de elementos que han de ser insertados, aunque queden posiciones de la tabla sin uso.

El método anterior tiene una deficiencia: suponer que dos elementos X e Y son tales que H(X) = H(Y). Entonces, cuando un el elemento X entra en la tabla, éste se inserta en la posición dada por su función Hash, H(X). Pero cuando al elemento Y le es asignado su posición donde será insertado mediante la función hash, resulta que la posición que se obtiene es la misma que la del elemento X. Esta situación se denomina colisión o choque.

Una buena función Hash será aquella que *minimiza las colisiones, y que distribuya los elementos uniformemente a través del arreglo*. Esta es la razón por la que *el tamaño del arreglo debe ser un poco mayor que el número real de elementos a insertar*, pues cuanto más grande sea el rango de la función de randomización, es menos probable que dos claves generen el mismo valor de asignación o hash, es decir, que se asigne una misma posición a más de un elemento.

Habrá que llegar a un compromiso entre Eficiencia en Espacio-Tiempo: el dejar espacios vacíos en la tabla es una deficiencia en cuanto a espacio, mientras que reduce la necesidad de resolver los casos de choque en la asignación, y por lo tanto es más eficiente en términos de tiempo.

Una solución al problema de las colisiones: Zona de desbordamiento.

Se trata de mantener una zona reservada para aquellos elementos que llegan a colisionar, de manera que cuando se produzca una colisión el elemento se va a localizar en esta zona de desbordamiento.

Al realizar la búsqueda y comprobar si el elemento buscado está en la posición dada por su tabla hash, si esa posición ya está ocupada por otro elemento con el mismo valor de hashing, se seguirá buscando a partir del inicio de la zona de desbordamiento de manera secuencial, hasta encontrar el elemento o llegar al final de dicha zona de desbordamiento.

Siguiendo con el ejemplo, para cada código se obtenía el residuo (resto) de su división por un número primo (en este caso el 53). Es recomendable usar números primos ya que que reduce la probabilidad de colisión al establecer funciones del tipo: ***H(X) = (X % primo)***

Ahora bien, ¿cuál valor primo escoger? En este ejemplo se manejan 30 códigos, de modo que ese valor debe ser superior a 30, y que su magnitud se acerque al doble de elementos, por ello una buena opción era el 53 (o también el 61). Valores primos muy cercanos a la cantidad original de datos da lugar a demasiadas colisiones y un valor primo grande (por ejemplo, 97) da lugar a un considerable desperdicio de espacio (casillas vacías).

¿Cómo nos quedarían ubicados los elementos en el vector? Hay que tener en cuenta algo más: como el valor primo utilizado en la función de Hashing fue el 53, inicialmente el vector destino tendría 53 casillas, pero a eso hay que agregarle las casillas de la zona de desbordamiento, de modo que habría que aumentar esas casillas alrededor de un 25 %, y en ese caso nuestro vector quedaría con 53 * 1.25 = 66.25, o sea 66 casillas en total. Las 53 primeras almacenarán los códigos según la posición que resulte al aplicar la función de Hashing, las 13 restantes representan la zona de desbordamiento a donde irán los códigos que colisionen.

Veamos un gráfico de cómo quedarían los códigos en ese vector:

0		33	104611010059
1		34	
2	104611010505	35	
3	46101046	36	
4		37	46082053
5	46101048	38	
6	104611010138	39	
7		40	104611024641
8	104611010776	41	46101031
9	46102006	42	
10	46092043	43	46102040
11		44	46102041
12	104611010038	45	
13		46	104611010178
14		47	
15	104611010253	48	
16	46091095	49	
17	46102014	50	46081165
18		51	46082014
19		52	
20		**53**	**46102030**
21	46092107	**54**	**46102043**
22	104611010843	**55**	**46092103**
23		**56**	
24	46102021	**57**	
25	1046110103	**58**	
26		**59**	
27		**60**	
28	46102025	**61**	
29		**62**	
30	46102080	**63**	
31	1046110109	**64**	
32		**65**	

3.5 Ejercicios sobre búsquedas

1. **Dado el siguiente arreglo:**

2	3	18	20	22	25	30	100	200
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

Mostrar que elementos recorrería la búsqueda binaria hasta encontrar los valores:
a) 18
b) 3
c) 100
d) 200
e) 25

2. Se tienen los siguientes valores:

40 50 62 103 204 304 328

almacenar los valores en una tabla hash con 50 posiciones, usando como función hash: CLAVE MOD 50, y como método de resolución de colisiones la función rehashing: (CLAVE + 1) MOD 50

4 — Algoritmos de Ordenamiento

4.1 Introducción a los algoritmos de Ordenamiento

En computación y matemáticas un **algoritmo de ordenamiento** es un algoritmo que pone elementos de una lista o un vector en una secuencia dada por una relación de orden, es decir, el resultado de salida ha de ser una permutación —o reordenamiento— de la entrada que satisfaga la relación de orden dada. Las relaciones de orden más usadas son el orden numérico y el orden lexicográfico. Ordenamientos eficientes son importantes para optimizar el uso de otros algoritmos (como los de búsqueda y fusión) que requieren listas ordenadas para una ejecución rápida. También es útil para poner datos en forma canónica y para generar resultados legibles por humanos.

Desde los comienzos de la computación, el problema del ordenamiento ha atraído gran cantidad de investigación, tal vez debido a la complejidad de resolverlo eficientemente a pesar de su planteamiento simple y familiar. Por ejemplo, BubbleSort fue analizado desde 1956.1 Aunque muchos puedan considerarlo un problema resuelto, nuevos y útiles algoritmos de ordenamiento se siguen inventado hasta el día de hoy (por ejemplo, el ordenamiento de biblioteca se publicó por primera vez en el 2004). Los algoritmos de ordenamiento son comunes en las clases introductorias a la computación, donde la abundancia de algoritmos para el problema proporciona una gentil introducción a la variedad de conceptos núcleo de los algoritmos, como notación de O mayúscula, algoritmos divide y vencerás, estructuras de datos, análisis de los casos peor, mejor, y promedio, y límites inferiores.

Fuente: Wikipedia

4.2 Ordenamiento por selección

El algoritmo de ordenamiento por selección se basa en la idea de tener dividido el arreglo que se está ordenando en dos partes: una, con un grupo de elementos ya ordenados, que ya encontraron su posición final. La otra, con los elementos que no han sido todavía ordenados. En cada iteración, localizamos el menor elemento de la parte no ordenada, lo intercambiamos con el primer elemento de esta misma región e indicamos que la parte ordenada ha aumentada en un elemento.

El algoritmo implementado en java es el siguiente:

```
public classOrdenamiento {
/**
 * Despues de la primera iteracion el elemento menor queda en la primera
 * posicion del arreglo.Para ello busca el menor valor de todo el arreglo y
 * lo intercambia con el que queda en la primera posicion. Luego repite el
 * mismo proceso pero comenzando desde el segundo elemento del arreglo,
 * busca el menor y lo intercambia con el primer elemento que se encuentra
 * en la parte sin ordenar. ORDEN (N*N)
*/
```

```
    public int[] Seleccion(int[] arregloSinOrdenar) {
        int[] arreglo = darCopiaValores(arregloSinOrdenar);
        int n = arreglo.length;
        // lleva la parte sin ordenar del arreglo
        for(inti = 0; i < n - 1; i++) {
            intmenor = arreglo[i];
            int posMenor = i;
            // busca el menor de la parte sin ordenar
            for(int j = i + 1; j < n; j++) {
                if(arreglo[j] < menor) {
                    menor = arreglo[j];
                    posMenor = j;
                }
            }
            // después de haber encontrado el dato lo intercambia con
            // el que se encuentra en la casilla i, para eso utiliza temp.
            int temp = arreglo[i];
            arreglo[i] = menor;
            arreglo[posMenor] = temp;
        }
        return arreglo;
    }

    /**
    *
    * Saca una copia del vector
    */
    public int[] darCopiaValores(int[] arreglo) {
        int[] arregloNuevo = new int[arreglo.length];
        for(int i = 0; i < arreglo.length; i++) {
          arregloNuevo[i] = arreglo[i];
        }
        return arregloNuevo;
    }
}
```

El programa principal que invoca este método de ordenamiento sería:

```
public class ClienteMain {
    public static voidmain(String[] args) {
        Ordenamiento o = newOrdenamiento();
        int[] arreglo = { 5,26,1,7,14,3};
        mostrar(arreglo);
        // Cambiar aqui por el algoritmo de ordenamiento a probar
        mostrar(o.seleccion(arreglo));
    }


    /**
    * Muestra por consola todos los datos del arreglo, separados por espacio
    *
    * @param arreglo
    * arreglo a imprimir
    */
    public static voidmostrar(int[] arreglo) {
        for(int i = 0; i < arreglo.length; i++) {
            System.out.print(arreglo[i] +"␣");
        }
        System.out.println("");
    }
}
```

El algoritmo anterior funciona de la siguiente manera:

Tenemos un arreglo de enteros no ordenados de 6 posiciones:

Después de la primera iteración del algoritmo, queremos que el menor elemento (1) quede en la primera posicion. para esto buscamos el menor valor de todo el arreglo y lo intercambiamos con el que esta en la primera posicion (5).

Luego, repetimos el mismo proceso pero comenzando desde el segundo elemento del arreglo. Buscamos el menor (3) y lo reemplazamos con el primer elemento que se encuentra en la parte sin ordenar (26).

El proceso continúa hasta que todo el arreglo quede ordenado.

Estudio de la complejidad:
Para el estudio de la complejidad se debe contar el número de comparaciones e intercambios que requiere la ordenación de n elementos. El procedimiento siempre realiza n-1 intercambios. Existen n-1 llamadas a intercambio. El bucle interno hace i-1 comparaciones cada vez; el bucle externo itera n-1 veces, de modo que el número total de comparaciones es $O(n^2)$, por lo que el número de comparaciones claves es cuadrático. Ha de observarse que el algoritmo no depende de la disposición inicial de los datos. Además, el número de intercambios del arreglo es lineal O(n).

4.3 Ordenamiento burbuja

Burbuja u ordenamiento por intercambio, intercambia todo par de elementos consecutivos que no se encuentran en orden. Al final de cada pasada haciendo este intercambio, un nuevo elemento queda ordenado y todos los demás elementos se acercaron a su posición final. Se llama burbuja porque al final de cada iteración el mayor va surgiendo al final.

El algoritmo implementado en java es el siguiente:

```
public class Ordenamiento {

    /**
    * Burbuja u ordenamiento por intercambio
    * @param arreglo a ser ordenado
    * @return arreglo ordenado
    */
    public int[] burbuja(int[] arregloSinOrdenar) {
       int[] arreglo = darCopiaValores(arregloSinOrdenar);
```

```
        int n = arreglo.length;
        // controla el punto hasta el cual lleva el proceso de intercambio.
        // Termina cuando queda un solo elemento
        for (int i = n; i > 1; i--) {
            // lleva el proceso de intercambio empezando en 0 hasta llegar
            // al punto anterior al límite marcado por la variable i
            for(int j = 0; j < i - 1; j++) {
                if (arreglo[j] > arreglo[j + 1]) {
                    int temp = arreglo[j];
                    arreglo[j] = arreglo[j + 1];
                    arreglo[j + 1] = temp;
                }
            }
         }
         return arreglo;
    }


    /**
    *
    * Saca una copia del vector
    */
    public int[] darCopiaValores(int[] arreglo) {
        int[] arregloNuevo = new int[arreglo.length];
        for(int i = 0; i < arreglo.length; i++) {
            arregloNuevo[i] = arreglo[i];
        }
        return arregloNuevo;
    }
}
```

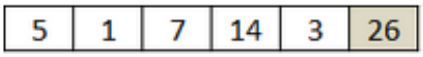
El programa principal que invoca este método de ordenamiento sería:

```
public classClienteMain {
    public static voidmain(String[] args) {
      Ordenamiento o = new Ordenamiento();
      int[] arreglo = {5,26,1,7,14,3};
      mostrar(arreglo);
      // Cambiar aquí por el algoritmo de ordenamiento a probar
      mostrar(o.burbuja(arreglo));
    }


    /**
    * Muestra por consola todos los datos del arreglo, separados por espacio
    *
    * @param arreglo
    * arreglo a imprimir
    */
    public static voidmostrar(int[] arreglo) {
      for(int i = 0; i < arreglo.length; i++) {
        System.out.print(arreglo[i] +"␣");
      }
      System.out.println("");
    }
}
```

El algoritmo anterior funciona de la siguiente manera:

Tenemos un arreglo de enteros no ordenados de 6 posiciones:

En la primera iteración recorremos el arreglo de izquierda a derecha intercambiando todo par de valores consecutivos que no se encuentren ordenados. Al final de la iteración, el mayor de los elementos (26), debe quedar en la última posicion del arreglo, mientras todos los otros valores han avanzado un poco hacia su posicion final.



Este mismo proceso se repite para la parte del arreglo que todavia no esta ordenada.

En esta segunda iteración podemos ver que varios elementos del arreglo han cambiado de posicion mientras que el elemento mayor de la parte no ordenada a alcanzado su ubicación.

El proceso termina cuando solo quede un elemento en la zona no ordenada del arreglo, lo cual requiere 5 iteraciones en nuestro ejemplo.

Estudio de la complejidad

Para el estudio de la complejidad se debe contar el número de comparaciones e intercambios que requiere la ordenación de n elementos. El bucle externo itera n-1 veces (desde n hasta 1). El bucle interno hace i-1 comparaciones cada vez; de modo que el número total de comparaciones es $O(n^2)$.

4.4 Ordenamiento por inserción

Este método separa la secuencia en dos grupos: una parte con los valores ordenados (inicialmente con un solo elemento) y otra con los valores por ordenar (inicialmente todo el resto). Luego vamos pasando uno a uno los valores a la parte ordenada, asegurándose que se vayan colocando ascendentemente.

El algoritmo implementado en java es el siguiente:

```
public class Ordenamiento {
/**
 * Este metodo separa la secuencia en dos grupos: una parte con los valores
 * ordenados (inicialmente con un solo elemento) y otra con los valores por
 * ordenar (inicialmente todo el resto). Luego vamos pasando uno a uno los
 * valores a la parte ordenada, asegurandonos que se vayan colocando
 * ascendentemente.
 *
 * @param arregloSinOrdenar
 * arreglo sin ordenar. ORDEN (N*N)
 * @return arreglo ordenado
 */
```

```
    public int[] insercion(int[] arregloSinOrdenar) {
      int[] arreglo = darCopiaValores(arregloSinOrdenar);
      int n = arreglo.length;
      // i señala la posicion del elemento que va a insertar,
      // va desplazando hacia la izquierda, casilla a casilla,
      // el elemento que se encontraba inicialmente en i,
      // hasta que encuentra la posicion adecuada
      for (int i = 1; i < n; i++) {
        for(int j = i; j > 0 && arreglo[j - 1] > arreglo[j]; j--) {
          int temp = arreglo[j];
          arreglo[j] = arreglo[j - 1];
          arreglo[j - 1] = temp;
        }
      }
      return arreglo;
    }

}
```

El programa principal que invoca este método de ordenamiento sería:

```
public class ClienteMain {
    public static void main(String[] args) {
      Ordenamiento o = newOrdenamiento();
      int[] arreglo = { 5,26,1,7,14,3 };
      mostrar(arreglo);
      // Cambiar aquí por el algoritmo de ordenamiento a probar
      mostrar(o.insercion(arreglo));
    }
    /**
    * Muestra por consola todos los datos del arreglo, separados por espacio
    *
    * @param arreglo
    * arreglo a imprimir
    */
    public static void mostrar(int[] arreglo) {
      for (inti = 0; i < arreglo.length; i++) {
        System.out.print(arreglo[i] +"␣");
      }
      System.out.println("");
    }

}
```

El algoritmo anterior funciona de la siguiente manera:
Tenemos un arreglo de enteros no ordenados de 6 posiciones:

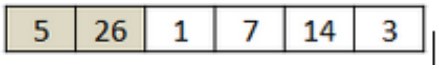
5	26	1	7	14	3

Al inicio, podemos suponer que en la parte ordenada hay un elemento (5), no esta todavia en su posicion final, pero siempre es cierto que una secuencia de un solo elemento está ordenada.

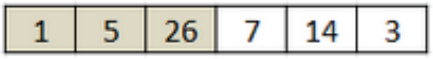
5	26	1	7	14	3

luego tomamos uno a uno los elementos de la parte no ordenada y los vamos insertando ascendentemente en la parte ordenada, comenzamos con el (26) y vamos avanzando hasta insertar el (3).

primera iteración:



segunda iteración:



quinta y última iteración:

en esta iteración no hay movimientos ya que el último elemento ya esta ordenado.

Estudio de la complejidad:

El bucle externo se ejecuta n-1 veces. Dentro de este bucle existe otro bucle que se ejecuta a lo más el valor de i veces para valores de i que están en el rango de n-1 a 1. Por consiguiente, en el peor de los casos, las comparaciones del algoritmo vienen dadas por:

$$1 + 2 + \ldots + (n\text{-}1) = n(n\text{-}1)/2$$

Por otra parte, el algoritmo mueve los datos como máximo el mismo número de veces. Existe por lo tanto, en el peor de los casos, los siguiente movimientos:

$$n(n\text{-}1)/2$$

Por consiguiente, el algoritmo de ordenación por inserción es $O(n^2)$ en el caso peor.

4.5 Ordenamiento Shell

El método se denomina Shell en honor de su inventor Donald Shell. Realiza comparaciones entre elementos NO consecutivos, separados por una distancia salto. El valor salto al principio es n/2 y va decreciendo en cada iteración hasta llegar a valer 1. Cuando salto vale 1 se comparan elementos consecutivos. El valor se encontrará ordenado cuando salto valga 1 y no se puedan intercambiar elementos consecutivos porque están en orden.

El algoritmo implementado en java es el siguiente:

```
 public class Ordenamiento {
/**
* Realiza comparaciones entre elementos NO consecutivos, separados por una
* distancia salto El valor salto al principio es n/2 y va decreciendo en
* cada iteracion hasta llegar a valer 1. Cuando salto vale 1 se comparan
* elementos consecutivos. El valor se encontrara ordenado cuando salto
* valga 1 y no se puedan intercambiar elementos consecutivos porque estan
* en orden
*
* @param arregloSinOrdenar
* arreglo sin ordenar
* @return arreglo ordenado
*/
public int[] shell(int[] arregloSinOrdenar) {
    int[] arreglo = darCopiaValores(arregloSinOrdenar);
    int n = arreglo.length;
    int salto = n;
    booleanordenado;
    while(salto > 1) {
      salto = salto / 2;
```

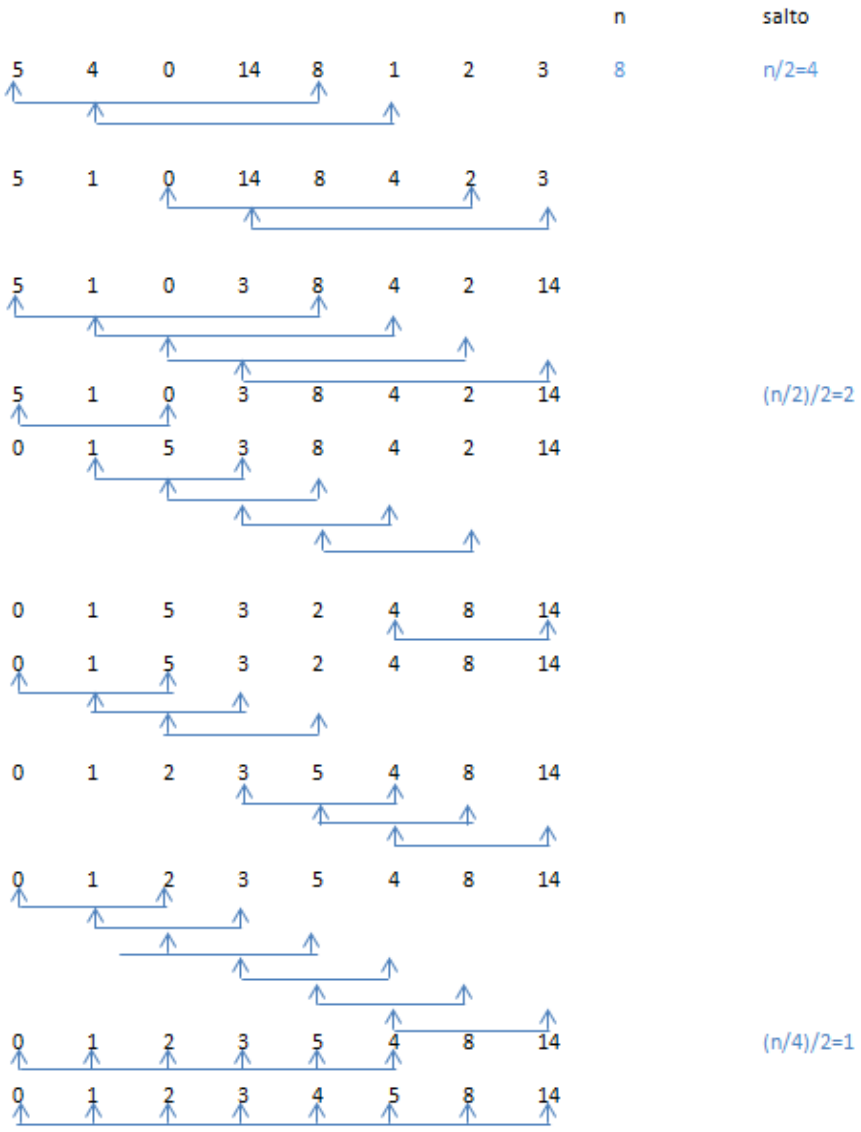
```
      do{
        ordenado = true;
        for(intj = 0; j <= n - 1 - salto; j++) {
          int k = j + salto;
          if(arreglo[j] > arreglo[k]) {
            intaux = arreglo[j];
            arreglo[j] = arreglo[k];
            arreglo[k] = aux;
            ordenado = false;
          }
        }
      } while (!ordenado);
    }
    return arreglo;

}
```

El programa principal que invoca este método de ordenamiento sería:

```
public class ClienteMain {
    public static void main(String[] args) {
      Ordenamiento o = newOrdenamiento();
      int[] arreglo = {5, 4, 0, 14, 8, 1, 2, 3};
      mostrar(arreglo);
      // Cambiar aquí por el algoritmo de ordenamiento a probar
      mostrar(o.shell(arreglo));
    }

    /**
    * Muestra por consola todos los datos del arreglo, separados por espacio
    * @param arreglo
    * arreglo a imprimir
    */
    public static voidmostrar(int[] arreglo) {
      for(int i = 0; i < arreglo.length; i++) {
        System.out.print(arreglo[i] +"␣");
      }
      System.out.println("");
    }

}
```

La siguiente gráfica muestra un seguimiento del método de ordenación Shell para los datos de entrada:

5, 4, 0, 14, 8, 1, 2, 3.



El proceso termina cuando salto=1 y no hay ningún cambio en la iteración.

Estudio de la complejidad

Su implementación original, requiere O(n2) comparaciones e intercambios en el peor caso. Un cambio menor presentado en el libro de V. Pratt produce una implementación con un rendimiento de O(n log2 n) en el peor caso. Aunque es fácil desarrollar un sentido intuitivo de cómo funciona este algoritmo, es muy difícil analizar su tiempo de ejecución.

El Shell sort es una generalización del ordenamiento por inserción, teniendo en cuenta dos observaciones:

1. El ordenamiento por inserción es eficiente si la entrada está çasi ordenada".
2. El ordenamiento por inserción es ineficiente, en general, porque mueve los valores sólo una posición cada vez.

El algoritmo Shell sort mejora el ordenamiento por inserción comparando elementos separados por un espacio de varias posiciones. Esto permite que un elemento haga “pasos más grandes” hacia su posición esperada. Los pasos múltiples sobre los datos se hacen con tamaños de espacio cada vez más pequeños.

El último paso del Shell sort es un simple ordenamiento por inserción, pero para entonces, ya está garantizado que los datos del vector están casi ordenados.

4.6 Merge Sort

Este método de ordenación divide el vector por la posición central, ordena cada una de las mitades y después realiza la mezcla ordenada de las dos mitades. El caso base es aquel que recibe un vector con ningún elemento, o con 1 solo elemento, ya que obviamente está ordenado.

El algoritmo implementado en java es el siguiente:

```
public class Ordenamiento {
    /**
    * Este m todo de ordenaci n divide el vector por la posici n central,
    * ordena cada una de las mitades y despu s realiza la mezcla ordenada
        de
    * las dos mitades. El caso base es aquel que recibe un vector con
        ning n
    * elemento, o con 1 solo elemento, ya que obviamente est  ordenado.
        ORDEN
    * (N LOG N)
    *
    * @param arreglo
    * , arreglo de elementos
    * @param ini
    * , posicion inicial
    * @param fin
    * , posici n final
    */
    public void mergeSort(int arreglo[], int ini, int fin) {
      int m = 0;
      if(ini < fin) {
        m = (ini + fin) / 2;
        mergeSort(arreglo, ini, m);
        mergeSort(arreglo, m + 1, fin);
        merge(arreglo, ini, m, fin);
      }
    }
    /**
    * Une el arreglo en uno solo. O MezclaLista
    *
    * @param arreglo
    * arreglo de elementos
    * @param ini
    * posici n inicial
    * @param m
    * posici n intermedia
    * @param fin
    * posici n final
    */
    private void merge(int arreglo[], int ini, int m, int fin) {
      int k = 0;
      int i = ini;
      int j = m + 1;
      int n = fin - ini + 1;
      intb[] = new int[n];
      while(i <= m && j <= fin) {
        if (arreglo[i] < arreglo[j]) {
```

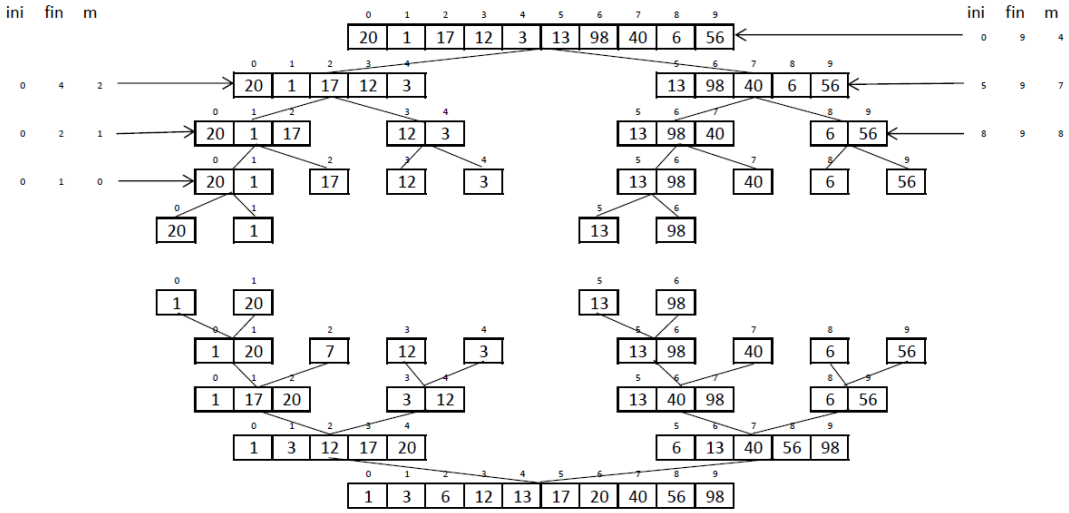
```
            b[k] = arreglo[i];
            i++;
            k++;
          }else {
            b[k] = arreglo[j];
            j++;
            k++;
          }
        }
        while(i<= m) {
          b[k] = arreglo[i];
          i++;
          k++;
        }
        while(j<= fin) {
          b[k] = arreglo[j];
          j++;
          k++;
        }

        for(k=0; k<n; k++) {
          arreglo[ini + k] = b[k];
        }
      }

      /**
      * Toma los datos para invocar al algoritmo mergeSort recursivo
      * @param arregloSinOrdenar
      * @return
      */

      public int[] mergeSortTomaDatos(intarregloSinOrdenar[]) {
        int[] arreglo = darCopiaValores(arregloSinOrdenar);
        intn = arreglo.length;
        mergeSort(arreglo, 0, arreglo.length - 1);
        return arreglo;
      }

}
```

El programa principal que invoca este método de ordenamiento sería:

```
public classClienteMain {
    public static voidmain(String[] args) {
      Ordenamiento o =newOrdenamiento();
      int[] arreglo = {20,1,17,12,3,13,98,40,6,56};
      mostrar(arreglo);
      // Cambiar aquí por el algoritmo de ordenamiento a probar
      mostrar(o.mergeSortTomaDatos(arreglo));
    }
    /**
    * Muestra por consola todos los datos del arreglo, separados por espacio
    *
    * @param arreglo
    * arreglo a imprimir
    */
    public static voidmostrar(int[] arreglo) {
      for(inti = 0; i < arreglo.length; i++) {
        System.out.print(arreglo[i] +"␣");
      }
      System.out.println("");
```

```
    }
}
```

El algoritmo anterior funciona de la siguiente manera:
Tenemos un arreglo de enteros no ordenados de 10 posiciones:



Estudio de la complejidad

El algoritmo es recursivo esa razón que se quiere determinar el tiempo empleado por cada una de las 3 fases del algoritmo divide y vencerás. Cuando se llama a la función mezclista se deben mezclar las dos listas mas pequeñas en una nueva lista con n elementos. la función hace una pasada a cada una de las sublistas. Por consiguiente, el número de operaciones realizadas será como máximo el producto de una constante multiplicada por n. si se consideran las llamadas recursivas se tendrá entonces el número de operaciones: constante *n* (profundidad de llamadas recursivas). El tamaño de la lista a ordenar se divide por dos en cada llamada recursiva, de modo que el número de llamadas es aproximadamente igual al número de veces que nse puede dividir por 2, parándose cuando el resultado es menor o igual a 1.

Por consiguiente, la ordenación por mezcla se ejecutará, aproximadamente, el número de operaciones: alguna constante multiplicada por n y después por (logn), Resumiendo hay dos llamadas recursivas, y una iteración que tarda tiempo n por lo tanto la recurrencia es:

T(n) = n + 2T(n/2) si n>1 y 1 en otro caso. De esta forma, aplicando expansión de recurrencia se tiene: T(n) = n + 2T(n/2) = n +2 n/2+4T(n/4) =..... = n+n+n+ +... ..n(k= logn veces)=n*logn O(n) por lo tanto la ordenación por mezcla tiene un tiempo de ejecución de O(n*logn)

4.7 Quick Sort

Este algoritmo divide en array en dos subarray, que se pueden ordenar de modo independiente. Se selecciona un elemento específico del array arreglo[centro] llamado pivote. Luego, se debe re situar los demás elementos de la lista a cada lado del pivote, de manera que a un lado queden todos los menores que él, y al otro los mayores. Los elementos iguales al pivote pueden ser colocados tanto a su derecha como a su izquierda, dependiendo de la implementación deseada. En este momento, el pivote ocupa exactamente el lugar que le corresponderá en la lista ordenada. Finalmente, se divide el array original en dos subarrays que se ordenarán de modo independiente mediante llamadas recursivas del algoritmo.

El algoritmo implementado en java es el siguiente:

```
 public classOrdenamiento {
/**
*
* Este algoritmo divide en array en dos subarray, que se pueden ordenar de
* modo independiente. Se selecciona un elemento especifico del array
* arreglo[centro] llamado pivote y se divide el array original en dos
* subarrays que se ordenaran de modo independiente mediante llamadas
* recursivas del algoritmo.
*
* @param arreglo
* arreglo que se esta ordenando
* @param inicio
*posici n de inicio - ORDEN(N LOG N) EN EL CASO MEDIO.
* ORDEN(N*N) EN EL PEOR CASO
* @param fin
* posicion final
*/
public voidquickSort(int[] arreglo, intinicio,intfin) {
     inti = inicio;// i siempre avanza en el arreglo hacia la derecha
     intj = fin; // j siempre avanza hacia la izquierda
     intpivote = arreglo[(inicio + fin) / 2];
     do{
       while (arreglo[i] < pivote)
         // si ya esta ordenado incrementa i
         i++;
       while (pivote < arreglo[j])
         // si ya esta ordenado decrementa j
         j--;
       if(i <= j) {// Hace el intercambio
         int aux = arreglo[i];
         arreglo[i] = arreglo[j];
         arreglo[j] = aux;
         i++;
         j--;
       }
     }
     while (i <= j);

     if (inicio < j)
       quickSort(arreglo, inicio, j);// invocación recursiva
     if(i < fin)
       quickSort(arreglo, i, fin);// invocacion recursiva
}
/**
*M todo intermediario que invoca al algoritmo quickSort recursivo
*
* @param arregloSinOrdenar
* arreglo sin ordenar
* @return arreglo ordenado
*/
public int[] quickSortTomaDatos(int[] arregloSinOrdenar) {
int[] arreglo = darCopiaValores(arregloSinOrdenar);
quickSort(arreglo, 0, arreglo.length- 1);
return arreglo;
}
/**
*
```

```
* Saca una copia del vector
*/
public int[] darCopiaValores(int[] arreglo) {
int[] arregloNuevo = new int[arreglo.length];
for(inti = 0; i < arreglo.length; i++) {
arregloNuevo[i] = arreglo[i];
}return arregloNuevo;
}
}
```

El programa principal que invoca este método de ordenamiento sería:

```
 public classClienteMain {
    public static voidmain(String[] args) {
      Ordenamiento o = newOrdenamiento();
      int[] arreglo = { 1, 5, 7, 9, 6, 10, 3, 2, 4, 8 };
      mostrar(arreglo);
      // Cambiar aqui por el algoritmo de ordenamiento a probar
      mostrar(o.quickSortTomaDatos(arreglo));
    }
    /**
    * Muestra por consola todos los datos del arreglo, separados por espacio
    *
    * @param arreglo
    * arreglo a imprimir
    */
    public static voidmostrar(int[] arreglo) {
      for(inti = 0; i < arreglo.length; i++) {
        System.out.print(arreglo[i] + "␣");
      }
      System.out.println("");
    }

}
```

Estudio de la Complejidad

Estabilidad: NO es estable.
Requerimientos de Memoria: No requiere memoria adicional en su forma recursiva.
Tiempo de Ejecución:
***Caso promedio**. La complejidad para dividir una lista de n es O(n). Cada sublista genera en promedio dos sublistas más de largo n/2. Por lo tanto la complejidad se define en forma recurrente como:
f(1) = 1
f(n) = n + 2 f(n/2)
La forma cerrada de esta expresión es: f(n) = n log2n Es decir, la complejidad es O(n log2n).
***El peor caso** ocurre cuando la lista ya está ordenada, porque cada llamada genera sólo una sublista (todos los elementos son menores que el elemento de división). En este caso el rendimiento se degrada a O(n2). Con las optimizaciones mencionadas arriba puede evitarse este comportamiento.
Ventajas:
El más rápido
No requiere memoria adicional
Desventajas:
Implementación un poco más complicada.

Mucha diferencia entre el peor caso (n2) y el mejor caso (log n).
Intercambia registros iguales.

Diversos estudios realizados sobre el comportamiento del algoritmo Quicksort, demuestran que si se escoge en cada pasada ele elemento que ocupa la posición central del conjunto de datos a analizar, el número de pasadas necesarias para ordenar el arreglo es del orden de log n.

Respecto al número de comparaciones, si el tamaño del arreglo es una potencia de 2, en la primera pasada realizará (n-1) comparaciones, en la segunda pasada realizará (n-1)/2 comparaciones pero en dos conjuntos diferentes, en la tercera pasada realizará (n-1)/4 comparaciones diferentes pero en cuatro conjuntos diferentes y así sucesivamente.

Por lo tanto: c=(n-1)+(n-1)+(n-1)+.....+(n-1).

Si se considera a cada uno de los componentes de la sumatoria como un término y el número de términos de la sumatoria es igual a m, entonces se tiene que: c=(n-1)*m.

Considerando que el número de términos de la sumatoria(m) es igual al número de pasadas, y que éste es igual a (log n), la expresión anterior queda: c=(n-1)*log n.

4.8 Ejercicios sobre algoritmos de ordenamiento

1. Dada la siguiente secuencia: (20, 1, 17, 12, 3, 13, 98, 40, 6, 56), mostrar el proceso de ordenamiento de los siguientes métodos (pruebas de escritorio):
a) Ordenamiento por selección
b) Burbuja
c) Ordenación por inserción
d) Shell
e) QuickSort
f) MergeSort

2. Cuando todos los elementos son iguales, ¿cuál es el tiempo de ejecución de los siguientes métodos?
a) Inserción
b) Shell
c) MergeSort
d) QuickSort

3. Cuando la entrada ya está ordenada, ¿cuál es el tiempo de ejecución de los siguientes métodos?
a) Inserción
b) Shell
c) MergeSort
d) QuickSort

4. Cuando la entrada está originalmente ordenada, pero en orden inverso, ¿cuál es el tiempo de ejecución de los siguientes métodos:
a) Inserción
b) Shell
c) MergeSort
d) QuickSort

5. A continuación se describen los pasos de otro algoritmo de ordenamiento:

a) Se recorre el vector

- Primero se compara V[i] con V[i+1] para todos los valores de i pares,
- Luego se compara V[i] con V[i+1] para todos los valores de i impares
- Cada vez que V[i] es mayor que V[i+1] se intercambian los valores.

b) Se continúa en forma alterna hasta que el conjunto de datos esté ordenado. El algoritmo se apoya en una variable de control, de modo que si no hubo intercambios en todo el recorrido, significa que ya está ordenado y terminará.

Implementar en un lenguaje de programación el anterior algoritmo.

5 — Complejidad computacional

5.1 Conceptos

5.1.1 Definición de algoritmo

Algoritmo es sinónimo de procedimiento computacional y es fundamental para las ciencias de la computación.

Un algoritmo es una secuencia finita de instrucciones, cada cual con un significado concreto y cuya ejecución genera un tiempo finito. Un algoritmo debe terminar en un tiempo finito.

Algoritmo es toda receta, proceso, rutina, método, etc. que además de ser un conjunto de instrucciones que resuelven un determinado problema, cumple las siguientes condiciones:

1. **Ser finito**. La ejecución de un algoritmo acaba en un tiempo finito; un procedimiento que falle en la propiedad de la finitud es simplemente un procedimiento de cálculo.
2. **Ser preciso**. Cada instrucción de un algoritmo debe ser precisa; deberá tenerse en cuenta un rigor y no la ambigüedad, esta condición es la definibilidad: cada frase tiene un significado concreto.
3. **Posee entradas**. Las entradas se toman como un conjunto específico de valores que inicializan el algoritmo.
4. **Posee salida**. Todo algoritmo posee una o más salidas; la salida es la transformación de la entrada.
5. **Ser eficaz**. Un algoritmo es eficaz cuando resuelve el problema.
6. **Ser eficiente**. Un algoritmo es eficiente cuando resuelve el problema de la mejor manera posible, o sea utilizando la mínima cantidad de recursos.

Una vez que tenemos un algoritmo que resuelve un problema y podemos decir que es de alguna manera correcto, un paso importante es tener idea de la cantidad de recursos, como tiempo de procesador o espacio en la memoria principal que requerirá.

Los objetivos del análisis de algoritmos son:

- Conocer los factores que influyen en la eficiencia de un algoritmo.
- Aprender a calcular el tiempo que emplea un algoritmo.
- Aprender a reducir el tiempo de ejecución de un programa (por ejemplo, de días o años a fracciones de segundo).

5.1.2 Factores que influyen en la eficiencia de un algoritmo

Podemos tomar en cuenta muchos factores que sean externos al algoritmo como la computadora donde se ejecuta (hardware y software) o factores internos como la longitud de entrada del algoritmo. Veamos algunos de estos factores.

- **El Hardware**. Por ejemplo: procesador, frecuencia de trabajo, memoria, discos, etc.
- **El Software**. Por ejemplo: sistema operativo, lenguaje de programación, compilador, etc.
- **La longitud de entrada**. El enfoque matemático considera el tiempo del algoritmo como una función del tamaño de entrada. Normalmente, se identifica la longitud de entrada (tamaño de entrada), con el número de elementos lógicos contenidos en un ejemplar de

entrada, por ejemplo: en un algoritmo que calcula el factorial de un número, la longitud de entrada sería el mismo número, porque no es lo mismo calcular el factorial de 4 que calcular el factorial de 1000, las iteraciones que tenga que hacer el algoritmo dependerá de la entrada. De igual manera se puede considerar como longitud de entrada: al tamaño de un arreglo, el número de nodos de una lista enlazada, el número de registros de un archivo o el número de elementos de una lista ordenada). A medida que crece el tamaño de un ejemplar del programa, generalmente, crece el tiempo de ejecución. Observando cómo varía el tiempo de ejecución con el tamaño de la entrada, se puede determinar la tasa de crecimiento del algoritmo, expresado normalmente en términos de n, donde n es una medida del tamaño de la entrada. La tasa de crecimiento de un problema es una medida importante de la eficiencia ya que predice cuánto tiempo se requerirá para entradas muy grandes de un determinado problema. Para que un algoritmo sea eficiente, se debe optimizar el tiempo de ejecución y el espacio en la memoria, aunque se producirá la optimización de uno a costa del otro.

5.1.3 Análisis de Algoritmos

El análisis de algoritmo que hacemos toca únicamente el punto de vista temporal (tiempo de ejecución de un algoritmo) y utilizamos como herramienta el lenguaje de programación Java.

Es difícil realizar un análisis simple de un algoritmo que determine la cantidad exacta de tiempo requerida para ejecutarlo. La primera complicación es que la cantidad exacta de tiempo dependerá de la implementación del algoritmo y de la máquina real en que se ejecuta. El análisis normalmente debe ser independiente de la computadora (hardware y software) y del lenguaje o máquina que se utilice para implementar el algoritmo. La tarea de calcular el tiempo exacto requerido suele ser bastante pesado.

Un algoritmo es un conjunto de instrucciones ordenados de manera lógica que resuelven un problema. Estas instrucciones a su vez pueden ser: enunciados simples (sentencias) o enunciados compuestos (estructuras de control). El tiempo de ejecución dependerá de como esté organizado ese conjunto de instrucciones, pero nunca será constante.

Es conveniente utilizar una función T(n) para representar el número de unidades de tiempo (o tiempo de ejecución del algoritmo) tomadas por un algoritmo de cualquier entrada de tamaño n. La evaluación se podrá hacer desde diferentes puntos de vista:

- **Peor caso.** Se puede hablar de T(n) como el tiempo para el peor caso. Se trata de aquellos ejemplares del problema en los que el algoritmo es menos eficiente (no siempre existe el caso peor). Ejemplos: insertar al final de una lista, ordenar un vector que está ordenado en orden inverso, etc. Nos interesa mucho.
- **Mejor caso.** Se habla de T(n) como el tiempo para el mejor caso. Se trata de aquellos ejemplares del problema en los que el algoritmo es más eficiente; por ejemplo: insertar en una lista vacía, ordenar un vector que ya está ordenado, etc. Generalmente no nos interesa.
- **Caso medio.** Se puede computar T(n) como el tiempo medio de ejecución del programa sobre todas las posibles ejecuciones de entradas de tamaño n. El tiempo de ejecución medio es aveces una medida más realista del rendimiento en la práctica, pero es, normalmente, mucho más difícil de calcular que el tiempo de ejecución en el pero caso, ya que requiere definir una distribución de probabilidades de todo el conjunto de datos de entrada, el cuál típicamente es una tarea difícil.

5.2 Función de Complejidad

5.2.1 Definición

La función de complejidad de un algoritmo es **el número de operaciones elementales** que utiliza un algoritmo cualquiera para resolver un problema de tamaño **n.** Matemáticamente se define la Función de complejidad asi:

Sea A un algoritmo, la función de complejidad del algoritmo A T(n) se define como el número máximo de operaciones elementales que utiliza el algoritmo para resolver un problema de tamaño n.

T(n) = Max {nx: nx es el número de operaciones que utiliza A para resolver una instancia x de tamaño n}

Nota: Una operación elemental es cualquier operación cuyo tiempo de ejecución es acotado por una constante (que tenga tiempo constante). Por ejemplo: una operación lógica, una operación aritmética, una asignación, la invocación a un método.

5.2.2 Tipos de operaciones Elementales

¿Que es una operación elemental? Una operación elemental, también llamado operador básico es cualquier operación cuyo tiempo de ejecución es constante, es decir, es una operación cuyo tiempo de ejecución siempre va a ser el mismo.

Tipos de Operaciones elementales:

- **Operación Lógica:** Son operaciones del tipo a $>$ b, o por ejemplo los indicadores que se suelen utilizar en los condicionales que si se cumpla esta condición o esta otra haga esto. Ese o es una operación lógica.
- **Operacion Aritmetica:** Son operaciones del tipo a + b, o a / b, etc.
- **Asignación:** Es cuando asignamos a una variable un valor, ejemplo: int a = 20+30, el igual (=) en este caso es la operación de asignación.
- **Invocación a un Método:** Como su nombre lo dice es cuando llamamos, cuando invocamos a un método.

5.2.3 Cálculo del T(n)

Para hallar la función de complejidad (t(n)) de un algoritmo se puede evaluar el algoritmos desde tres puntos de vista:

- **Peor Caso:** Se puede hablar de T(n) como el tiempo de ejecución para el peor de los casos, en aquellos ejemplares del problema en el que el algoritmo es Menos Eficiente.
- **Caso Medio:** Se puede comportar el T(n) como el tiempo medio de ejecución del programa sobre todas las posibles ejecuciones de entrada de tamaño n. Es una medida más realista del rendimiento del algoritmo en la práctica, pero es mucho más difícil del cálculo, ya que requiere una distribucion de probabilidades de todo el conjunto de entrada lo cual es una tarea difícil.
- **Mejor Caso:** Se puede hablar de T(n) como el tiempo de ejecución para el mejor de los casos, en aquellos ejemplares del problema en el que el algoritmo es Más Eficiente.

Lo ideal sería poder evaluar el algoritmo en el caso promedio, pero por el nivel de operaciones y dificultad que conlleva este, el estudio de la complejidad de los algoritmos se evalúa en el **peor** de los casos.

Para calcular el T(n), se deben calcular el número de operaciones elementales de un algoritmo, las cuales estan dadas por: Sentencias consecutivas, condicionales y ciclos.

Sentencias Consecutivas

Las sentencias consecutivas como su nombre lo dicen son aquellas que tienen una secuencia, que van una detrás de otra, y se derivan en dos casos:

1. Si se tiene una serie de instrucciones consecutivas:

Sentencia 1 $\rightarrow$ n1 operaciones elementales
Sentencia 2 $\rightarrow$ n2 operaciones elementales
.
.
.
Sentencia k $\rightarrow$ nk operaciones elementales

$$\sum_{i=1}^{k} ni = n1 + n2 \pm \cdots + nk$$

2. Procedimientos:

Procedimiento 1 $\rightarrow$ f1(n)
Procedimiento 2 $\rightarrow$ f2(n)
.
.
.
Procedimiento k $\rightarrow$ fk(n)

$$\sum_{i=1}^{n} f(i)$$

3. Ciclos

Para hallar la complejidad computacional de un ciclo existen, los siguientes casos:

Caso A: Cuando los incrementos o decrementos son de uno en uno y son constantes.

El ciclo While o Mientras es el básico, a partir de él se analizan los demás ciclos.

Codigo base:

i $\leftarrow$ *1*
    mientras i <= n Haga
Proceso

 # De Operaciones (Formula) = 1 + (n+1) * nc + n * np

Siendo:

nc: Número de operaciones de la condición del ciclo.
np: Número de operaciones del ciclo.

Ejemplo: Hallar la función de complejidad del siguiente algoritmo:

lineas	Código	# De Operaciones
1	A = 0 ——————————- >	1
2	i = 1 ——————————- >	1
3	Mientras (i <= n) ————— >	nc = 1
4	A = A + i ————— >	2
5	i = i + 1 ————— >	2 np = (2+2) = 4
6	imprima A ——————————- >	1

Lineas	Complejidad	
de 2 a 5	1 + (n + 1) * nc + (n * np) 1 + n + 1 + 4n 5n + 2	1 + (n + 1) * 1 + (n * 4)
de 1 a 6	5n + 2 + 2	

En conclusion:

T(n) = 5n + 4

Ejemplo: Hallar la función de complejidad del siguiente algoritmo.

lineas	Código	# De Operaciones
1	x = 1 ——————————>	1
2	i = 1 ——————————->	1
3	Mientras (i <= n) ——————>	nc = 1
4	x = x * i ——————->	2
5	i = i + 1 ——————>	2 np = (2+2) = 4
6	Si (x > 100) entonces ————>	nc = 1
7	x = 100 ——————>	1
8	imprima x ——————>	1

Lineas	Complejidad
de 2 a 5	1 + (n+1)nc + n * np 1 + (n+1)1 + n * 4 1 + n + 1 + 4n 5n + 2
de 6 a 7	nc + max 1 + 1 2
de 1 a 8	(5n + 2) + (2) + 1 + 1 5n + 6

En conclusion:

T(n) = 5n + 6

- **For o Para:**

Codigo base:

Para (i←1 ; i <=n ; i++)
Proceso

 # De Operaciones (Formula) = 1 + (n+1) * nc + n * (np+1)

Siendo:

- **nc:** Número de operaciones de la condición del ciclo.
- **np:** Número de operaciones del ciclo.

Ejemplo: Hallar la función de complejidad del siguiente algoritmo.

lineas	Código	# De Operaciones
1	f = 0 ——————————>	1
2	Para (i = 1 ; i <=n ; i++) –>	nc = 1
3	f = f * i ——————>	2 operaciones —> np = 2
4	imprima f ——————->	1

Lineas	Complejidad
de 2 a 3	1 + (n+1)nc + n(np + 1) 1 + (n+1)1 + n(2 + 1) 1 + n + 1 + n(3) 1 + n + 1 + 3n 4n + 2
de 1 a 4	(4n + 2) + 2 4n + 4

En conclusion:

T(n) = 4n + 4

Ejemplo:

NOTA PARA TENER EN CUENTA:

"Para ciclos ya sean **MIENTRAS(while) O PARA(for),** observamos que cada ciclo tiene un límite superior que define hasta donde va a iterar(repetir) **ya sea definido estrictamente menor (<) que n, o en dado caso que n sea una constante** ; por consecuencia debemos usar la siguiente fórmula, el cual nos permite aclarar el valor real de n; es decir, cuantas veces opera y/o itera el ciclo(# de veces que ingresa al ciclo) "

FORMULA: n = Ls - Li + 1
Siendo:

- **Ls:** Limite superior del ciclo. (Hasta donde itera)
- **Li:** Limite inferior del ciclo. (Desde donde inicializa)

En este ejemplo tenemos el ciclo → **Para** (i = 1 ; **i** < **n** ; i++).

"Como observamos el **límite inferior(Li) del ciclo es 1** porque inicializa desde 1; el **límite superior(Ls) es (n-1)** porque va hasta i estrictamente menor (<) que n; es decir el ciclo ingresaría n veces menos 1 por que es estrictamente menor (<).

En los ejemplos anteriores, los ciclos tienen como limite superior **n** porque llegaba hasta i<=n, es decir ingresaba las n veces al ciclo.

Para comprobar lo dicho en este caso, entonces usamos la fórmula:

n = Ls - Li + 1
n = (n-1) - (1) + 1
n = n-1

en conclusión, como n = n-1 ; el número total de veces que se ingresa al ciclo es (n-1) veces; en consecuencia reemplazamos el valor de n que es (n-1) en la complejidad.

lineas	Código	# De Operaciones
1	f = 1 ————————————>	1
2	Para (i = 1 ; i < n ; i++) –>	nc = 1
3	f = f * i ————————>	2 operaciones —> np = 2
4	imprima f ————————->	1

Lineas	Complejidad
de 2 a 3	1 + (n+1)nc + n(np + 1) “como n=(n-1) entonces”
	1 + ((n-1)+1)(1) + (n-1)(2 + 1)
	1 + n + (n-1)(3)
	1 + n + (3n - 3)
	1 + n + 3n - 3
	4n - 2
de 1 a 4	(4n - 2) + 2
	4n

En conclusion:

T(n) = 4n

Caso B: Cuando los incrementos o decrementos dependen del valor que tome **i.**

- **While o Mientras:**

Codigo base:

i ← 1
mientras i <= n Haga
Proceso **(i)**

$$\#\ de\ Operaciones\ (Formula) = 1 + (n+1) * nc + \sum_{i=1}^{n} f(i)$$

Siendo:

nc: Número de operaciones de la condición del ciclo.
f(i): Función de complejidad del proceso.

- **For o Para**

Codigo base:

Para (i←1 ; i <=n ; i++)
Proceso **(i)**

Siendo:

$$\#\ de\ Operaciones\ (Formula) = 1 + (n+1) * nc + n + \sum_{i=1}^{n} f(i)$$

- **nc:** Número de operaciones de la condición
- **f(i):** Función de complejidad del proceso.

Ejemplo:

lineas	Código	# De Operaciones
1	s = 0 ——————————————>	1
2	Para (i = 1 ; i <=n ; i++) ——————>	nc = 1
3	Para (j = 1 ; j<= i ; j++) ——>	nc = 1
4	s = s + 1 ——————->	2 operaciones —> np = 2
5	imprima f ——————————————>	1

Lineas **Complejidad**

de 3 a 4

$1 + (n+1)nc + n(np+1)$ "$CasoA \rightarrow n = i''$

$1 + (i+1)1 + i(2+1)$

$1 + i + 1 + i(3)$

$1 + i + 1 + 3i$

$4i + 2 - - - - - > f(i)$

de 2 a 4

$1 + (n+1)nc + n + \sum_{i=1}^{n} f(i)$

$1 + (n+1)nc + n + \sum_{i=1}^{n} 4i + 2$

$2n + 2 + \sum_{i=1}^{n} 4i + \sum_{i=1}^{n} 2$

$2n + 2 + 4\sum_{i=1}^{n} i + 2\sum_{i=1}^{n} 1$

$2n + 2 + 4\left(n\frac{(n+1)}{2}\right) + 2n$

$4n + 2 + (2n^2) + 2n$

$(2n^2) + 6n + 2$

de 1 a 5

$(2n^2) + 6n + 2 + 2$

$(2n^2) + 6n + 4$

En conclusión:

$$T(n) = 2n^2 + 6n + 4$$

CASO C: Cuando no sabemos cuánto va a ser el incremento o decremento del proceso.

Código base:

mientras<Condición> Haga
Proceso

 # De Operaciones (Formula) = (nr + 1) * nc + (nr * np)

siendo:

nc: Número de operaciones de la condición del ciclo.
np: Número de operaciones del ciclo.
nr: Número de veces que se repite el proceso, en sí es una fórmula que cumple las veces que se ejecuta el proceso para todo número n.

Función Piso / Techo

Sabemos claramente que (5/4) es igual a 1.25

Función piso es aproximar el fraccionario a su más cercano valor entero menor o igual a él, un ejemplo sería:

$$\text{Función piso: } \left\lfloor \tfrac{5}{4} \right\rfloor = \text{sería 1}$$

ahora bien si tomamos el mismo fraccionario y hallamos función techo sería aproximarlo a su más cercano valor mayor o igual a él, un ejemplo sería:

$$\text{Función Techo: } \left\lceil \tfrac{5}{4} \right\rceil = \text{sería 2}$$

NOTA: Si tenemos un entero la función piso y techo es el mismo entero

¿Como hallar el nr?

Para permitirnos calcular t(n) o función de complejidad con un incremento no lineal será incierto saber cual es el número de veces que va a iterar el ciclo ya sea while o un for

Cuando un incremento por lo general no en todas las ocasiones sean de suma de 2 (contador=contador+2) dentro del ciclo, el contador irá sumando de 2 en 2; el nr podría ser un fraccionario con numerador siempre n y denominador la constante que se está sumando en este caso 2.

Ejemplo: Hallar la función de complejidad del siguiente algoritmo

lineas	Código	# De Operaciones
1	t = 0 ——————————————->	1
2	i = 1 ——————————————->	1
3	Mientras (i <= n)——————————>	nc = 1
4	t = t + 1——————————>	2
5	i = i + 2 ——————————>	2
		np = (2 + 2) = 4
6	imprima t ——————————>	1

Estudio para hallar nr se debe analizar la cantidad de veces que itera el bucle para cada valor de n:

6	III	$nr = \lceil \frac{n}{2} \rceil = \lceil \frac{6}{2} \rceil = 3$	3
7	IIII	$nr = \lceil \frac{n}{2} \rceil = \lceil \frac{7}{2} \rceil = 4$	4
8	IIII	$nr = \lceil \frac{n}{2} \rceil = \lceil \frac{8}{2} \rceil = 4$	4
9	IIIII	$nr = \lceil \frac{n}{2} \rceil = \lceil \frac{9}{2} \rceil = 5$	5
10	IIIII	$nr = \lceil \frac{n}{2} \rceil = \lceil \frac{10}{2} \rceil = 5$	5

5.3 Algoritmos recursivos

Def: La recursividad es una técnica de programación en la cual un método puede llamarse a sí mismo, en la mayoría de casos un algoritmo iterativo es más eficiente que uno recursivo si de recursos de la computadora se trata, pero un algoritmo recursivo en muchos casos permite realizar problemas muy complejos de una manera más sencilla.

Reglas de la recursividad:

Para que un problema pueda resolverse de forma recursiva debe cumplir las siguientes 3 reglas:

Regla 1:Por lo menos debe tener un caso base y una parte recursiva.
Regla 2:Toda parte recursiva debe tender a un caso base.
Regla 3:El trabajo nunca se debe duplicar resolviendo el mismo ejemplar de un problema en llamadas recursivas separadas.

Ejemplo: Calcular el factorial de un número.

FACTORIAL DE UN NÚMERO N

8! = 8 * 7!

7! = 7 * 6!
6! = 6 * 5!
.
.
En general,
n! = n * (n-1)!

Veamos un caso particular, calculemos el factorial de 5 (5!):

factorial de 5 = 5 * 4! ——-> “factorial de 5 es igual 5 multiplicado por factorial de 4”
factorial de 4 = 4 * 3! ——->“factorial de 4 es igual 4 multiplicado por factorial de 3”
factorial de 3 = 3 * 2!——-> “factorial de 3 es igual 3 multiplicado por factorial de 2”
factorial de 2 = 2 * 1!——-> “factorial de 2 es igual 2 multiplicado por factorial de 1”
factorial de 1 = 1 ——-> “factorial de 1 es 1” ——> “caso base”

Una implementación en java seria:

```
public long factorial (int n){
    if (n == 0 || n==1) //Caso Base
            return 1;
    else
            return n * factorial (n - 1); //Parte Recursiva
}
```

5.4 Complejidad computacional de los algoritmos recursivos

5.4.1 Método del árbol de recursión

Existen varios métodos para calcular la complejidad computacional de algoritmos recursivos. Uno de me los métodos más simples es el árbol de recursión, el cual esadecuado para visualizar que pasa cuando una recurrencia es desarrollada. Un árbol de recursión se tienen en cuenta los siguientes elementos:

Nodo: Costo de un solo subproblema en alguna parte de la invocación recursiva.

Costo por Nivel: La suma de los costos de los nodos de cada nivel.

*Costo Total: Es la suma de todos los costos del árbol.

Ejemplo. Utilizando el método del árbol de recursión calcular la complejidad computacional del algoritmo recursivo del factorial. Lo primero es calcular las operaciones elementales de cada linea:

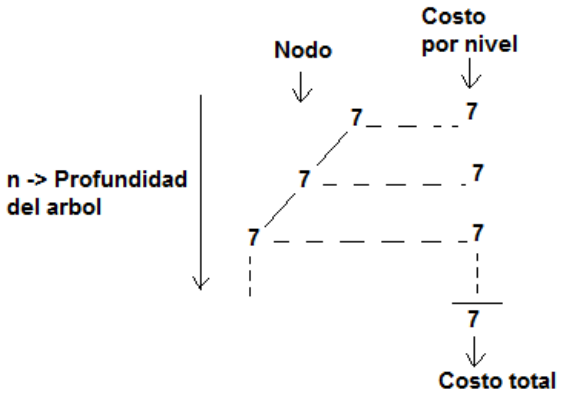
```
public long factorial (int n){ #operaciones elementales
    if (n == 0 || n==1) //Caso Base 3
            return 1; 1
    else
            return n * factorial (n - 1); //Parte Recursiva 4
}
```

$$T(n) = \begin{cases} 4 \quad si \quad (n = 0) \; o \; (n = 1) \\ 7 + T(n-1) \;, \quad Si \quad n>1 \end{cases}$$

Para hallar la complejidad se debe resolver esta recurrencia:

$$T(n) = 7 + T(n - 1)$$

El árbol de recursión es el siguiente.



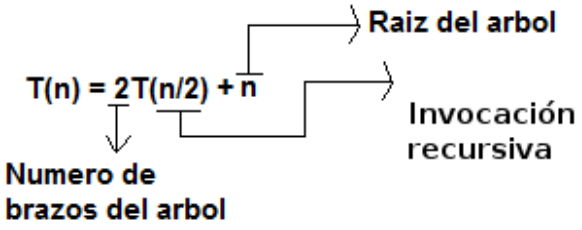
El costo total es la sumatoria de los costos de cada nivel:

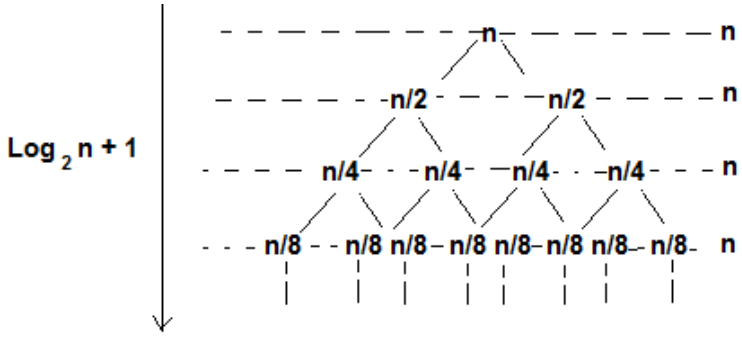
$$\sum_{i=1}^{n} 7 = 7n \; O(n) \rightarrow \text{Orden lineal}$$

Ejemplo

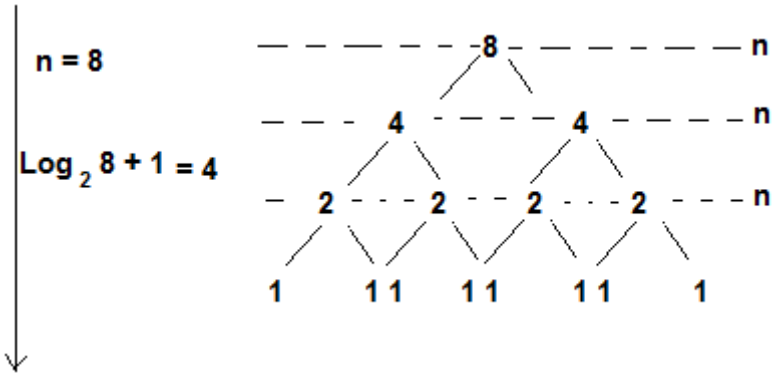
Utilizando el método del árbol de recursión, calcular la complejidad computacional de la siguiente recurrencia:

$T(n) = 2\,T(n/2) + n$





Para entender mejor el árbol de recursión anterior, ilustramos cómo sería cuando n = 8:



Finalmente, la complejidad de la recurrencia está dada por la suma de los costos de cada nivel del árbol:

$$\sum_{i=1}^{Log_2 n+1} n = n+n+n+n = n\log_2(n+1) \qquad O(nLog_2(n))$$

5.5 Ejercicios sobre análisis de algoritmos

*Ejercicios Caso A.
Calcular la función de complejidad T(n) de los siguientes algoritmos:
1)

```
c = 0
i = 1
Mientras (i <= n)
    c = c + 1
    i = i + 1
imprima c
```

2)

```
a = 0
i = 1
Mientras (i <n)
    a = a + i
    i = i + 1
imprima a
```

3)

```
a=0
i = 8
Mientras (i <n)
    a = a + i
    i = i + 1
imprima a
```

4)

```
a=0
i = 3
Mientras (i <n)
    a = a + i
```

```
    i = i + 1
imprima a
5)
f = 1
Para( i = 1; i <= n; i++)
    f = f * i
imprima f
6)
f = 6
Para(i = 5; i <= n/3 ; i++)
    f = f * 6 + i
imprima f
```

Respuestas:
1) T(n) = 5n + 4
2) T(n) = 5n 1
3) T(n) = 5n 36
4) T(n) = 5n 11
5) T(n) = 4n
6) T(n) = 2n 13

Ejercicios Caso B
Calcular la función de complejidad T(n) de los siguientes algoritmos

```
1)
s = 0
Para( i = 1; i <n; i++)
    Para( j = 1; j <i; j++)
        Para( k = 5; k <= j; k++)
            s = s + 4 * s
imprima s
2)
t = 0
Para( i = 1; i <= n; i++)
    Para( j = 1; j <= n; j++)
        t = t + j + i
imprima t
3)
t = 10
Para( i = 1; i <= n; i++)
    Para( j = 3; j <= i; j++)
        t = t + 1
imprima t
4)
t = 0
Para( i = 1; i <= n; i++)
    Para( j = 1; j <= i; j++)
        Para( k = 1; k <= j; k++)
            t = t + 1
imprima t
```

5)
t = 20
Para(i = 1; i <n; i++)
    Para(j = 1; j <2 * n; j++)
        t = t + 1
imprima t

Respuestas:
1) T(n) = 6/5n3 – 21/2n2 + 89/3 – 16
2) T(n) = 5n2 + 4n + 4
3) T(n) = 2n2 – 2n + 4
4) T(n) = 2/3n3 + 4n2 + 22/3n + 4
5) T(n) = 10n2 – 10n + 4

Ejercicios Caso C.
Calcular la función de complejidad T(n) de los siguientes algoritmos. “recuerde que los incrementos ya no son lineales(hallar nr)”
1)
i = 0
s = 0
Mientras (i <= n)
    s = s + 1 + i
    i = i + 3
imprima s
2)
i = 0
t = 3
Mientras (i <= n)
    t = t + 1
    i = i * 3
imprima t
3)
i = n
t = 0
Mientras (i <= n)
    t = t + 1 + i
    i = i/2
imprima t

Ejercicios Recurrencias
Calcular la complejidad de las siguientes recurrencias:

1) $T(n) = 3T(n/4) + n^2$
2) T(n) = T(n/2) + 1
3) T(n) = 2T(n/2) + 1
4) T(n) = 2T(n - 1) + k

6 — Tipos Abstractos de Datos

6.1 Conceptos TADs

6.1.1 Tipos Abstractos de Datos (TAD's)

Para hablar de la abstracción es necesario tener clara la diferencia que existe entre los datos, los tipos de datos y los tipos abstractos de datos.

Los datos son los valores que manejamos en la resolución de un problema, tanto los valores de entrada, como los de proceso y los de salida. Es decir, los datos son información y por lo tanto, para manejarlos se requieren varios tipos de datos.

Un tipo de dato se puede definir como un conjunto de valores y un conjunto de operaciones definidas por esos valores. Clasificar los datos en distintos tipos aporta muchas ventajas, como por ejemplo indicarle al compilador la cantidad de memoria que debe reservar para cada instancia dependiendo del tipo de dato al que pertenezca.

Un tipo de dato se puede definir como un conjunto de valores y un conjunto de operaciones definidas por esos valores. Clasificar los datos en distintos tipos aporta muchas ventajas, como por ejemplo, indicarle al compilador la cantidad de memoria que debe reservar para cada instancia dependiendo del tipo de dato al que pertenezca. Entre ellos estan:enteros, flotantes, dobles, cadenas de caracteres, . . . ,etc.

Los tipos de datos abstractos van todavía más allá; extienden la función de un tipo de dato ocultando la implementación de las operaciones definidas por el usuario. Esta capacidad de ocultamiento permite desarrollar software reutilizable y extensible.

6.1.2 Aspectos Teóricos

Un TAD es un tipo de dato definido por el programador. Los tipos abstractos de datos están formados por los datos (estructuras de datos) y las operaciones (procedimientos o funciones) que se realizan sobre esos datos. El término "tipo abstracto de datos" se refiere al concepto matemático básico que define el tipo de datos. Una guía útil para los programadores que que quieran usar los tipos de datos de forma correcta.

Las estructuras de los TAD'S se componen de un interfaz y la implementación .Las estructuras de datos que utilizamos para almacenar la representación de un TAD no son visibles para los usuarios. Mientras que en la interfaz declaramos las operaciones y los datos, la implementación contiene el código fuente de las operaciones y las operaciones , las cuales permanecen ocultos al usuario.

TAD = Valores +operaciones

Un TAD es independiente del lenguaje de programación aunque facilita métodos para su desarrollo ,ya que a la hora de pasar el TAD a un lenguaje de programación este da una idea al programador de qué tan sencillo o complejo puede ser el programa. La persona que desarrolla el TAD es libre de elegir o buscar otras alternativas posibles, pero lo que es importante es que

el resultado que se obtenga al desarrollar el TAD sea el correcto de los elementos que allí se encuentran.

Cuando vamos a crear un TAD es necesario tener una representación abstracta del objeto sobre el cual se va a trabajar, sin tener que recurrir a un lenguaje de programación .Esto nos va permitir crear las condiciones ,expresiones ,relaciones y operaciones de los objetos sobre los cuales vamos a trabajar.

Por ejemplo, si se va a desarrollar software para la administración de notas de una escuela, los TAD Curso, Estudiante, Nota, Lista, etc., van a permitir expresar la solución de cualquier problema planteado, en términos más sencillos, más

6.1.3 La modularidad

En la programación modular se descompone un programa creado en pequeñas abstracciones independientes las unas de las otras, que se pueden relacionar fácilmente unas con las otras. El módulo se caracteriza por su interfaz y su implementación . Al utilizar la modularidad para programar se debe seguir los principios de ocultación de la información.

La modularidad es un aspecto muy importante en la creacion de los TAD‘s, ya que es el reflejo de la independencia de la especificación y la implementación. Es la demostración de que un TAD puede funcionar con diferentes implementaciones.

Invariante

La invariante establece una validez para cada uno de sus objetos abstractos, en términos de condiciones sobre su estructura interna y sus componentes.Esto indica en qué casos un objeto abstracto modela un elemento del mundo del problema.

6.1.4 Métodos para Especificar un TAD

El objetivo es describir el comportamiento del TAD, existen 2 formas de especificar un TAD ,los cuales, pueden tener un enfoque formal y otro informal.

Especificación informal: Describe en lenguaje natural todos los datos y sus operaciones, sin aplicar conceptos matemáticos complicados para las persona que no están familiarizados con los TAD‘s, de manera, que todas las personas que no conocen a fondo las estructura de los TAD‘s, lo puedan entender de manera sencilla y que esas mismas personas puedan explicarlo de la misma manera natural , a tros con la misma facilidad con la que ellos lo entendieron.

Podemos representar de manera informal un TAD de la siguiente manera :
Tipos de datos : nombre del tipo de dato.
Valores: Descripción de los posibles valores.
Operaciones: descripcion de cada operacion.

Comenzamos escribiendo el nombre del TAD, luego describimos los posibles valores de este tipo de dato de manera abstracta ,sin tener que pensar en la realizacion concreta y por ultimos describiremos cada una de las operaciones creadas para el TAD

A continuación, crearemos de manera informal , en el lenguaje natural, un TAD sencillo.

Ejemplo: Especificar de manera informal un TAD Vector.

//nombre del TAD
TAD Vector
crear Vector //crea un vector

//operaciones
Asignar un elemento al vector .
Información del vector.
columnas del Vector .

Especificación Formal : Una de las ventajas de especificar formalmente con respecto a las informales , es que hay la posibilidad de simular especificaciones y las cuales se establecen mediantes precondiciones y postcondiciones. Aquí damos un conjunto de axiomas que van a describir el comportamiento de todas las operaciones.

Tipos: nombre de los tipos de datos.
Sintaxis: Forma de las operaciones.
Semántica: significado de las operaciones.

La sintaxis proporciona el tipo de dato de entrada como los tipos de datos de salida de las operaciones creadas , mientras que la semántica nos dice el comportamiento de las operaciones creadas en el TAD.

TAD <nombre>
<objeto abstracto>
<invariante>
<operaciones >
<operacion 1>
<operacion k-1>
...
<operacion k>: < dominio > <codominio>

<prototipo operación >
*/*Explicación de la operación*/*
{ precondicion : . . . . } / condición logica */*
*{ post condicion: . . . } /*condición logica */*

La precondición y las postcondición del mencionadas anteriormente ,se refiere a los elementos que componen los objetos abstractos y a los argumentos que recibe . En la especificación se debe considerar implícito en la precondición y la postcondición ,que el objeto abstracto sobre el cual se va a operar deba cumplir con el invariante .

Es importante elaborar una breve descripción de cada operación que se crea,de manera que el cliente pueda darse una rápida idea de las cosas que puede realizar el TAD, sin necesidad de entrar a un análisis detallado ,por lo cual esto va dirigido en especial al programador.

Ejemplo: Tomaremos el ejemplo anterior para la creación del TAD de manera formal:

TAD Vector[T]/* nombre del vector , donde T puede ser cualquier tipo de dato*/
/*Objeto abstracto de vector */
{ invariante: n>0 }

Vector crearVect(int fil , int col , int valor)
/*Crea y retorna un vector de dimensión [0...fil-1, 0] inicializada en 0 */

{ pre : 0 fil = 0 col< N}
{ post : crearVect es un vector de dimensión [0...fil-1], xik = 0 }

---

void asig_Vect(Vector v,int)
/* Asigna a la casilla de coordenadas [fil, col] el valor val */
{ pre: 0 fil =, 0 col < N }
{ post: X fil, col = val }

---

int infoVect(Vector v, int fil, int col)
/* Retorna el contenido de la casilla de coordenadas [fil, col] */
{ pre: 0 fil =0, col < N }
{ post: infoMat = X fil, col }

---

int columVect(Vector v)
/* Retorna el número de columnas del vector */
{ post: colum_Vect = N }

6.1.5 Clasificación de las Operaciones

Las operaciones de un TAD se clasifican en 3 grupos, según su función sobre el objeto abstracto:

- **Constructora**: es la operación encargada de crear elementos del TAD. En el caso típico, es la encargada de crear el objeto abstracto más simple. Tiene la siguiente estructura:

  Clase <constructora> (<argumentos>)
  { pre: }
  { post: }

  En el ejemplo anterior crearVectes la operación constructora del TAD Vector, pero un vector puede tener operaciones constructoras.

- **Modificadora**: esta operación que puede alterar el estado de un elemento del TAD. Su misión es simular una reacción del objeto.

  void <modificadora> (<objeto Abstracto>, <argumentos>)
  { pre: }
  { post: }

  En el ejemplo anterior del TAD Vector creado anteriormente, la operación modificadora es asig_Vect,que altera el contenido de una casilla del vector.

- **Analizadora**: es una operación que no altera el estado del objeto, sino que tiene como misión consultar su estado y retornar algún tipo deinformación.

<tipo> <analizadora> (<objeto Abstracto>, <argumentos>)
{ pre: }
{ post: = función ( ) }

Existen otro tipos de operaciones que podemos agregar a un TAD como lo son :

- **Comparación**: Es una analizadora que permite hacer calculable la noción de igualdad entre dos objetos del TAD.
- **Copia**: Es una modificadora que permite alterar el estado de un objeto del TAD copiandolo a partir de otro.
- **Destrucción**: Es una modificadora que se encarga de retornar el espacio de memoria dinámica ocupado por un objeto abstracto. Después de su ejecución el objeto abstracto deja de existir, y cualquier operación que se aplique sobre él va a generar un error. Sólo se debe llamar esta operación, cuando un objeto temporal del programa ha dejado de utilizarse.
- **Salida a pantalla**: Es una analizadora que le permite al cliente visualizar el estado de un elemento del TAD. Esta operación, que parece más asociada con la interfaz que con el modelo del mundo, puede resultar una excelente herramienta de depuración en la etapa de pruebas del TAD.
- **Persistencia**: Son operaciones que permiten salvar/leer el estado de un objeto abstracto de algún medio de almacenamiento en memoria secundaria. Esto permite a los elementos de un TAD sobrevivir a la ejecución del programa que los utiliza.

6.1.6 Ejemplo de un TAD

Crear el TAD RATIONAL, que corresponde al concepto matemático de un número racional. Un número racional es aquel que puede expresarse como el cociente de dos números enteros. Se definen las operaciones son la creación de un número racional a partir de dos enteros, la adición, la multiplicación y una prueba de igualdad. A continuación el TAD especificado de manera semi formal:

```
abstract typedef <integer, integer> RATIONAL;
condition RATIONAL[1]!=0;

/* definición del operador */
abstract RATIONAL makerational(a,b)
int a,b;
precondition b!=0;
postcondition makerational[0]==a;
makerational[1]==b;

/* written a+ b */
abstract RATIONAL add(a,b)
RATIONAL a,b;
postcondition add[1]==a[1]*b[1];
add[0]==a[0]*b[1]+b[0]*a[1];
/* written a*b */
abstract RATIONAL mult(a,b)
RATIONAL a,b;
postcondition mult[0]==a[0]*b[0];
```

*mult[1]==a[1]*b[1];*

/ written a==b */*
abstract equal(a,b)
RATIONAL a,b;
*postcondition equal==(a[0]*b[1]==b[0]*a[1]);*

6.2 Ejercicios sobre TADs

1. Crear un tad raíz en el que se creen la operaciones necesarias par ver si el elemento ingresado es válido o no.
2. Crear un TAD logaritmo en el cual se le hagan todas las operaciones necesarias para su desarrollo.
3. Crear un TAD Clínica ,en el se informe informe de una pequeña lista de pacientes ,en que veamos su información ,su prioridad de atención y su orden de salida.
4. crea un TAD alcancía ,en el que se informe cuanto billetes y monedas hay en la alcancía, ingresar una nueva moneda y un nuevo billete ,decir el mayor valor de las monedas y de los billetes ingresados.
5. Crear un TAD garaje en el cual se digan el numero de carro ingresados y el de salida,saber la hora de ingreso y cuanto es la tarifa más alta pagar.

7 — TDAs Lineales: Listas

Las Listas son tipos de datos abstractos lineales que representan secuencias de elementos y que presentan una particular flexibilidad en su manejo: pueden crecer y acortarse según se necesite, todos los elementos del conjunto se pueden acceder, se puede añadir nuevos elementos en cualquier lugar de la secuencia donde se especifique, así como se puede eliminar cualquier elemento del conjunto.

Una lista es equivalente a un contenedor de elementos, donde los valores pueden repetirse. Los valores almacenados en ella se conocen como item o elementos de la lista. Con frecuencia se representan las listas como una sucesión de elementos separados por comas:

a1, a2, ..., an

Donde n representa la longitud de la lista y es mayor a 0 y ai representa a cada elemento. Si n = 0 tendremos una lista vacia.

Las listas difieren de los arreglos porque sus elementos no se encuentran indexados, es decir, el acceso a cada elemento debe ser hecho de forma secuencial, mientras que en los arreglos se hace de forma aleatoria.

El TDA Lista puede ser implementado utilizando diferentes estructuras de datos: la implementación estática de las Listas implica la utilización de un arreglo para representar al conjunto de datos, mientras que la implementación dinámica se basa en el uso de referencias a objetos o punteros. Muchos lenguajes de programación proveen una implementación del tipo de dato Lista.

En este capítulo se ofrecerán definciones informales y formales del TDA (sus características y operaciones), se discutirán sus posibles implementaciones, se pondrá a disposición código fuente de algunas implementaciones en varios lenguajes y se propondrán problemas de aplicación de las Listas.

7.1 Definición y Formas de Uso

7.1.1 Definición

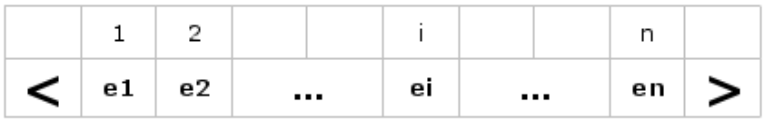
Se define una lista como una secuencia de cero o más elementos de un mismo tipo. El formalismo escogido para representar este tipo de objeto abstracto es:

$< e1, e2, \ldots, en >$

Cada ei modela un elemento del agrupamiento. Así, e1 es el primero de la lista, en es el último y la lista formada por los elementos $< e2, e3, \ldots, en >$ corresponde al resto de la lista inicial. La longitud de una lista se define como el número de elementos que la componen. Si la lista no tiene ningún elemento la lista se encuentra vacía y su longitud es 0. Esta estructura sin

elementos se representa mediante la notación $< >$, y se considera, simplemente, como un caso particular de una lista con cero elementos.

La posición de un elemento dentro de una lista es el lugar ocupado por dicho elemento dentro de la secuencia de valores que componen la estructura. Es posible referirse sin riesgo de ambigüedad al elemento que ocupa la i-ésima posición dentro de la lista y hacerlo explícito en la representación mediante la notación:



Esta indica que **ei** es el elemento que se encuentra en la posición i de la lista y que dicha lista consta de n elementos. Esta extensión del formalismo sólo se utiliza cuando se quiere hacer referencia a la relación entre un elemento y su posición.

El sucesor de un elemento en una lista es aquél que ocupa la siguiente posición. Por esta razón, el único elemento de una lista que no tiene sucesor es el último. De la misma manera, cada elemento de una lista, con excepción del primero, tiene un antecesor, correspondiente al elemento que ocupa la posición anterior. Es conveniente indicar que existe una posición que sucede a la del último elemento de la lista. Esta será un posición inválidad para cualquier elemento, pero marca el fin de la lista.

Para completar la definición del TDA, se debe definir un conjunto de operaciones para las Listas. Es importante reconocer que ningún conjunto de operaciones logrará satisfacer las necesidades de todas sus posibles aplicaciones. A continuación se propone un grupo de operaciones representativo en la cual *valor,* es un objeto del tipo del cual son los elementos de la lista y *pos,* es de tipo Posición. Posición en este caso es otro tipo de dato cuya especificación variará dependiendo de la implementación que se haga de la Lista. Aunque usualmente se piense en las posiciones como enteros, en la práctica las posiciones pueden ser de otros tipos de datos:

crearLista(): permite instanciar una Lista, creando una nueva lista vacía.
agregar(valor): adiciona el valor al final de la lista.
insertar(valor,pos): adiciona el valor en la posición pos en la lista.
remover(pos): elimina el nodo que se encuentra en la posición indicada.
esVacia(): retorna true si la lista esta vacía, false en caso contrario.
buscar(valor): retorna la posición del elemento cuyo valor coincida con el especificado.
getTamaño(): retorna el tamaño de la lista.
getValor(pos): retorna el valor almacenado en la posición pos.
getPrimero(): retorna la primera posición de la Lista.
getUltimo(): retorna la última posición de la Lista.
getSiguiente(pos): retorna la posicion del siguiente elemento del elemento de posición pos.
getAnterior(pos): retorna la posición del anterior elemento del elemento de posición pos.
getFin(): retorna la posición que sigue a la posición n en una lista L de n elementos.
limpiar(): remueve todos los elementos de la lista, dejándola vacía
destruirLista(): elimina la lista

Especificación

TAD Lista [T]
{ invariante: TRUE }
Constructoras:

crearLista:()
Modificadoras:
agregar(valor)
insertar(valor,pos)
remover(pos)
Analizadoras:
getTamaño()
getValor(pos)
esVacia()
buscar(valor)
getPrimero()
getUltimo(valor)
getSiguiente(pos)
getAnterior(pos)
getFin()
limpiar()
Destructora:
destruirLista()

---

crearLista()
/ Crea una lista vacía */*
{ post: crearLista = }

---

agregar(T valor)
/ Agrega valor al final de la lista*/*
{ post: lista = e1, e2, .. valor}

---

insertar(T valor, Posicion i)
/ Agrega valor en la posición i de la lista*/*
{ pre: 1 <= i <= n}
{ post: list = e1, ..., ei, valor, ei+1, ..., en }

---

remover(Posicion i)
/ Elimina el elemento de la lista de la posición i*/*
{ pre: 1 <= i <= n }
{ post: lista = e1, ..., ei-1, ei+1, ..., en}

---

getTamano()
/ Retorna la longitud de la Lista */*
{ pre: }
{ post: getTamano = n }

---

getValor(Posicion i)
/ Retorna el valor almacenado en la posición i */*
{ pre: 1 <= i <= n }
{ post: getValor = ei }

---

esVacia()

/ Informa si la lista está vacía */*
{ post: esVacia = (lista =) }

---

buscar(T valor)
/ Retorna la posición del valor */*
{ pre: valor = ei}
{ post: buscar = i }

---

getPrimero()
/ Retorna la primera posición de la lista*/*
{ pre: n>0}
{ post: getPrimero = p, p es la posicion de e1 }

---

getUltimo(T valor)
/ Retorna la última posición de la lista */*
{ pre: n>0}
{ post: getUltimo = p, p es la posicion de en }

---

getSiguiente(Posicion i)
/ Retorna la posición del elemento siguiente a aquel de posicion i */*
{ pre: 1 <= i <= n && i<>n}
{ post: getSiguiente = p, p es la posición de ei+1 }

---

getAnterior(Posicion i)
/ Retornala posición del elemento anterior a aquel de posicion i */*
{ pre: 1 <= i <= n && i<>1}
{ post: getSiguiente = p, p es la posición de ei-1 }

---

getFin()
/ Retorna la posición que sigue a la posición en en una lista L de n elementos */*
{ pre: }
{ post: getFin = n+1}

---

void destruirLista()
/ Destruye la lista retornando toda la memoria ocupada */*
{post: lista ha sido destruida }

7.1.2 Formas de Uso

Establecida la definición de la lista, se propondrá su utilización para resolver el problema de, dada una lista L cuyos elementos son números enteros, remover de esta lista aquellos elementos con valor par.

Para esto se necesitara moverse a través de la lista, desde el primer elemento hasta el último revisando si el valor almacenado cumple con la condición de ser par. Si se da el caso, el elemento será removido de la lista. Se propone una solución en pseudocódigo con enfoque orientado a objetos.

procedimiento removerPares(Lista L)

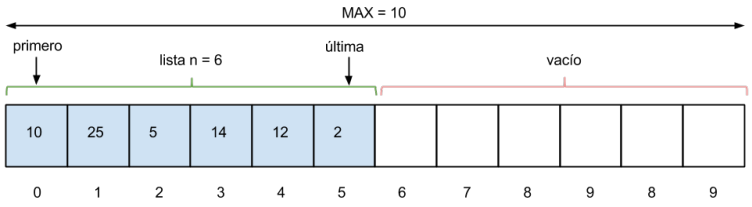
```
var
        Posicion p
Inicio
        p = L.getPrimero()
        Mientras p<> L.getUltimo()
            Si L.getValor(p) mod 2 == 0
                L.remover(p)
            FinSi
                p = L.getSiguiente(p)
        FinMientras
Fin
```

7.2 Implementación

Como se mencionó anteirormente, las listas se pueden implementar con diferentes estructuras de datos. En esta sección propodremos la implementación mediante arreglos y mediante punteros.

7.2.1 Mediante Arreglos

Al usar arreglos para almacenar los elementos de la lista, una opción es almacenar a los elementos en celdas contiguas. Para esto se mantendrá el tamaño de la lista en una variable n, la cual crecerá con cada operación de ingreso de nuevos elementos a la lista. El agregar un nuevo elemento al final de la lista es relativamente sencillo, simplemente se agrega luego del último elemento del arreglo y se aumenta el valor del tamaño n. Sin embargo, insertar un elemento en la mitad de la lista es un tanto más complicado, pues implica obligar al resto de elementos a desplazarse una posición dentro del arreglo. Así mismo, el remover un elemento de la lista implicaría desplacar todos los elementos para llenar de nuevo el vacío formado.



Para la implementación con arreglos se deberá definir un tipo de dato Lista como una clase con tres propiedades: el arreglo de elementos de una longitud adecuada para contener una lista de mayor tamaño de lo que se pueda necesitar, la maxima cantidad de elementos que el arreglo podrá almacenar y un entero que indique el tamaño del arreglo. En este caso, las posiciones de la lista se represtnarán mediante enteros y la operación getFin devolverá la posición siguiente del último: último + 1, es decir, n.

A continuación una posible implementación con arreglos usando Java:

```
public class ListaArreglo {
  private static final int MAX = 50;
 private Object elementos[];
 private int n;
 public ListaArreglo() {
```

```
  elementos = new Object[MAX];
  n = 0;
}
public void agregar(Object valor)
throws  ListException{
  if (n >= MAX) {
     throw new ListException("ListException␣en␣insertar:␣out␣of␣memory");
  }
  // agregar nuevo elemento
  elementos[n] = valor;
  n++;
}
public void insertar(Object valor, int pos)
throws  ListException,
        ListIndexOutOfBoundsException {
  if (n >= MAX) {
     throw new ListException("ListException␣en␣insertar:␣out␣of␣memory");
  }
  if (pos < 0 || pos > n+1) {
     throw new ListIndexOutOfBoundsException("ListIndexOutOfBoundsException
         ␣en␣insertar");
  }
  // hacer espacio para el nuevo elemento moviendo los otros una posición más adelante
  for (int i = n; i >= pos; i--) {
     elementos[i] = elementos[i-1];
  }
  // insertar nuevo elemento
  elementos[pos-1] = valor;
  n++;
}

public void remover(int pos)
throws ListIndexOutOfBoundsException {
  if (pos >= 0 && pos < n) { // desplazar los valores para reemplazar al valor a eliminar
     for (int i = pos+1; i <= getTamanio(); i++) {
       elementos[i-2] = elementos[i-1];
     }
     n--;
  }
  else {
     throw new ListIndexOutOfBoundsException("ListIndexOutOfBoundsException
         ␣al␣remover");
  }
}

public boolean esVacia() {
  return (n == 0);
}
public int buscar(Object valor) {
  for(int i = 0; i < n; i++){
      if(valor.equals(elementos[i])

          return i;

  }
  return -1;
}
public int getTamanio() {
  return n;
}
public Object getValor(int pos)
```

```
 throws ListIndexOutOfBoundsException {
   if (pos >= 0 && pos < n) {
      return elementos[pos-1];
   }
   else {
      throw new ListIndexOutOfBoundsException(
        "ListIndexOutOfBoundsException en getValor");
   }
 }
 public int getPrimero() {
   if (esVacia()) {
      return getFin();
   }
   else {
      return 0;
   }
 }
 public int getUltimo()
   if (esVacia()) {
      return getFin();
   }
   else {
      return 0;
   }
 }
 public int getSiguiente(int pos)
 throws ListIndexOutOfBoundsException {
   if (pos >= 0 && pos < n) {
      return pos+1;
   }
   else {
      throw new ListIndexOutOfBoundsException(
        "ListIndexOutOfBoundsException en getSiguiente");
   }
 }
 public int getAnterior(int pos)
 throws ListIndexOutOfBoundsException {
   if (pos >= 0 && pos < n) {
      return pos-1;
   }
   else {
      throw new ListIndexOutOfBoundsException(
        "ListIndexOutOfBoundsException en getAnterior");
   }
 }
 public int getFin()
    return n+1;
 }
 public void limpiar(){
    int oldn = n;
    for(int i = getUltimo();i>=0;i++)
        remover(i);
 }
}
```

Ventajas

Existe un rápido acceso aleatorio a los elementos

Presenta un eficiente manejo de memoria: se necesita muy poca memoria.

Desventajas

La eliminación e inserción de elementos toma mucho tiempo

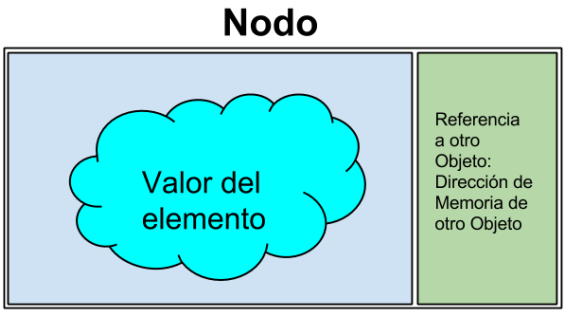
Se debe conocer el tamaño máximo del arreglo y este no se puede modificar luego de indicado.

7.2.2 Mediante Referencias a Objetos

La segunda forma de implementación de listas es a través del uso de referencias a objetos que permitan enlazar los elementos consecutivos. Esta implementación permite evitar el problema que se daba con los arreglos de tener que conocer previamente el tamaño máximo del conjunto para usarlo. Otro problema que se evita es el de los desplazamientos que se debían dar tanto en la eliminación como el ainserción de elementos. Sin embargo, el precio adicional por estas ventajas es el de la memoria que cada elemento ocupará.

Sin embargo, con arreglos, una lista de enteros ocupa en memoria lo que ocupa un entero por cada elemento. Con referencias a objetos, por cada elemento nuevo es necesario crear un contenedor que este listo para almacenarlo y para enlazarse con el siguiente elemento. Este contenedor se conoce como **Nodo**.

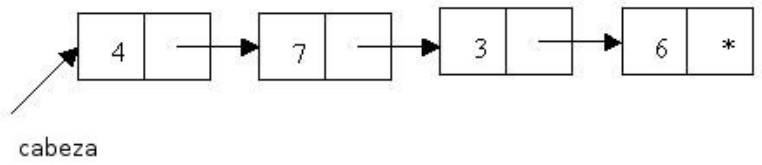
En una lista simplemente enlazada, un Nodo tiene la siguiente estructura:



Listas Simplemente Enlazadas

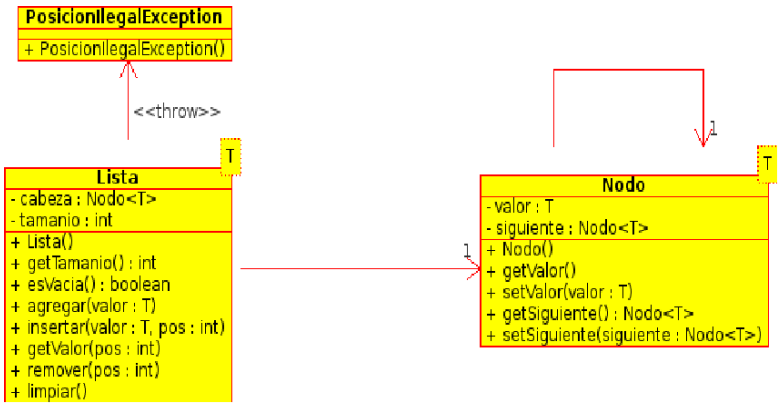
Una lista enlazada consiste en una secuencia de nodos, en los que se guardan campos de datos arbitrarios y una referencia al nodo posterior. La referencia que cada Nodo contiene no es más que una propiedad del nodo que almacena la dirección de memoria del siguiente nodo, o un valor Nulo si el nodo no se encuentra enlazado con un siguiente nodo.

Es decir, si la lista es a1, a2, ... an, el nodo que contiene ai contiene una referencia al nodo que contiene ai+1, para i = 1, 2, .. n-1. El nodo que contiene an contiene una referencia a un valor nulo. De esa forma, una lista enlazada puede ser representada gráficamente de la siguiente forma:



Como se puede apreciar en la gráfica, para tener acceso a todos los elementos de la lista solamente se requiere conocer la referencia al primer nodo de la lista, también conocido como Header o Cabeza. En este caso, la posición que se obtiene con la operación getFin() será siempre el valor NULL.

Entonces, el tipo de dato Lista estaría compuesto de la referencia al primer nodo de la lista y de un entero que llevaría el control del tamaño de la lista. El tipo de dato Nodo, por otro lado, tendrá dos atributos: el **valor** que almacena (que puede ser de cualquier tipo T) y la referencia al **siguiente** Nodo. La Clase Lista representa la lista como tal con todas las operaciones definidas en el TDA. A continuación un diagrama de clases que representa lo descrito:



En base al diseño indicado, se ofrece una implementación en java del TDA Lista usando referencias a objetos.

```
 package capitulo2.listas;
public class Nodo<T> {
        private T valor;
        private Nodo<T> siguiente;
        public Nodo() {
                valor = null;
                siguiente = null;
        }
        public T getValor() {
                return valor;
        }
        public void setValor(T valor) {
                this.valor = valor;
        }
        public Nodo<T> getSiguiente() {
                return siguiente;
        }
        public void setSiguiente(Nodo<T> siguiente) {
                this.siguiente = siguiente;
        }
}
package capitulo2.listas;
public class Lista<T> {
        /**
         * Primer elemento de la lista
         */
        private Nodo<T> cabeza;
        /**
         * Total de elementos de la lista
         */
        private int tamanio;
        /**
         * Constructor por defecto
         */
        public Lista() {
                cabeza = null;
```

```
        tamanio = 0;
}
/**
 * Devuelve el tama o de la lista
 * @return tama o de la lista
 */
public int getTamanio() {
        return tamanio;
}
/**
 * Consulta si la lista est  vacia o o
 * @return true cuando esta vacia, false cuando no
 */
public boolean esVacia() {
        return (cabeza == null);
}
/**
 * Agrega un nuevo nodo al final de la lista
 * @param valor valor a agregar
 */
public void agregar(T valor) {
        Nodo<T> nuevo = new Nodo<T>();
        nuevo.setValor(valor);
        if (esVacia()) {
                cabeza = nuevo;
        } else {
                // agregar al final de la lista
                Nodo<T> aux = cabeza;
                while (aux.getSiguiente() != null) {
                        aux = aux.getSiguiente();
                }
                aux.setSiguiente(nuevo);
        }
        tamanio++;
}

/**
 * inserta un nuevo nodo en la lista
 * @param valor valor a agregar
 * @param pos posicion donde se insertara el nodo
 * @throws PosicionIlegalException excepcion en caso que la posicion
     no exista
 */
public void insertar(T valor, int pos) throws Exception {
        if (pos >= 0 && pos <= tamanio) {
                Nodo<T> nuevo = new Nodo<T>();
                nuevo.setValor(valor);
                // el nodo a insertar esta al comienzo de la lista
                if (pos == 0) {
                        nuevo.setSiguiente(cabeza);
                        cabeza = nuevo;
                } else {
                        // El nodo a insertar va al final de la lista
                        if (pos == tamanio) {
                                Nodo<T> aux = cabeza;
                                while (aux.getSiguiente() != null) {
                                        aux = aux.getSiguiente();
                                }
                                aux.setSiguiente(nuevo);
                        }
                        // el nodo a insertar esta en medio
```

```
                        else {
                                Nodo<T> aux = cabeza;
                                for (int i = 0; i < pos - 1; i++) {
                                        aux = aux.getSiguiente();
                                }
                                Nodo<T> siguiente = aux.getSiguiente
                                    ();
                                aux.setSiguiente(nuevo);
                                nuevo.setSiguiente(siguiente);
                        }
                }
                tamanio++;
        } else {
                throw new Exception("posici n ilegal en la lista");
        }
}
/**
 * Devuelve el valor de una determinada posicion
 * @param pos posicion
 * @return el valor de tipo T
 * @throws PosicionIlegalException
 */
public T getValor(int pos) throws Exception {
        if (pos >= 0 && pos < tamanio)
        {
                T valor;
                if (pos == 0) {
                        valor = cabeza.getValor();
                        return valor;
                }
                else {
                        Nodo<T> aux = cabeza;
                        for (int i = 0; i < pos; i++) {
                                aux = aux.getSiguiente();
                        }
                        valor = aux.getValor();
                }
                return valor;
        }
        else
        {
                throw new Exception("posici n ilegal en la lista");
        }
}
/**
 * Elimina un nodo en una determinada posicion
 * @param pos posicion
 * @throws PosicionIlegalException
 */
public void remover(int pos) throws Exception {
        if (pos >= 0 && pos < tamanio)
        {
                if (pos == 0) {
                        //El nodo a eliminar esta en la primera posicion
                        cabeza = cabeza.getSiguiente();
                }
                else {
                        Nodo<T> aux = cabeza;
                        for (int i = 0; i < pos - 1; i++) {
                                aux = aux.getSiguiente();
                        }
```

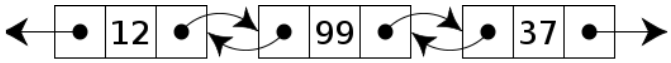
```
                                Nodo<T> prox = aux.getSiguiente();
                                aux.setSiguiente(prox.getSiguiente());

                        }
                        tamanio--;
                }
                else
                {
                        throw new Exception("posici n ilegal en la lista");
                }
        }
        /**
         * Clear, elimina todos los nodos de la lista
         */
        public void limpiar(){
                cabeza = null;
                tamanio=0;
        }
}
public class ClienteMain {
        /**
         * @param args
         */
        public static void main(String[] args) {
                try {
                        Lista<Integer> lista = new Lista<Integer>();
                        lista.agregar(12);
                        lista.agregar(15);
                        lista.agregar(20);
                        System.out.println("Dato en la posicion 0: " + lista
                            .getValor(0)); //Debe imprimir
                            12
                        System.out.println("Dato en la posicion 1: " + lista
                            .getValor(1)); //Debe imprimir
                            15
                        System.out.println("Dato en la posicion 2: " + lista
                            .getValor(2)); //Debe imprimir
                            20
                        lista.insertar(13, 1);
                        lista.insertar(16, 3);
                        //lista.insertar(16, 10);//Esta linea arroja excepcion porque la posicion 10 no
                            existe
                        System.out.println("Datos despues de la insercion");
                        System.out.println("Dato en la posicion 0: " + lista
                            .getValor(0)); //Debe imprimir
                            12
                        System.out.println("Dato en la posicion 1: " + lista
                            .getValor(1)); //Debe imprimir
                            13
                        System.out.println("Dato en la posicion 2: " + lista
                            .geTtValor(2)); //Debe imprimir
                            15
                        System.out.println("Dato en la posicion 3: " + lista
                            .getValor(3)); //Debe imprimir
                            16
                        System.out.println("Dato en la posicion 4: " + lista
                            .getValor(4)); //Debe imprimir
                            20

                        lista.remover(0);
                        lista.remover(3);
```

```
                        System.out.println("Datos despues de la eliminacion");
                        System.out.println("Dato en la posicion 0: " + lista.getValor(0)); //Debe imprimir 13
                        System.out.println("Dato en la posicion 1: " + lista.getValor(1)); //Debe imprimir 15
                        System.out.println("Dato en la posicion 2: " + lista.getValor(2)); //Debe imprimir 16

                } catch (Exception e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }
        }
}
```

El ClienteMain sirve para probar la lista, dentro de su main, invoca operaciones definidas en el TAD Lista.

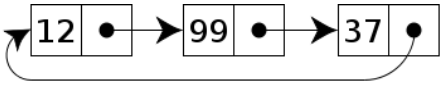
Listas doblemente enlazadas

En las listas doblemente enlazadas, cada nodo tiene dos enlaces: uno apunta al nodo anterior, o apunta al valor Nulo si es el primer nodo; y otro que apunta al siguiente nodo, o apunta al valor Nulo si es el último nodo.



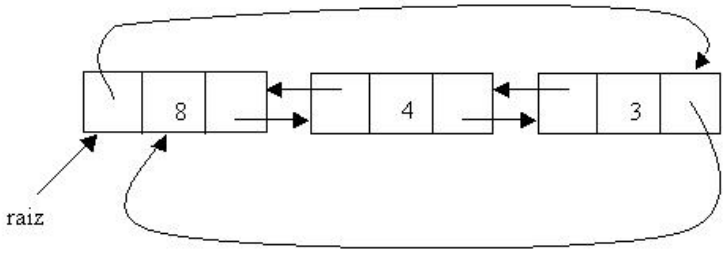
Listas circulares

En una lista enlazada circular, el primer y el último nodo están unidos.



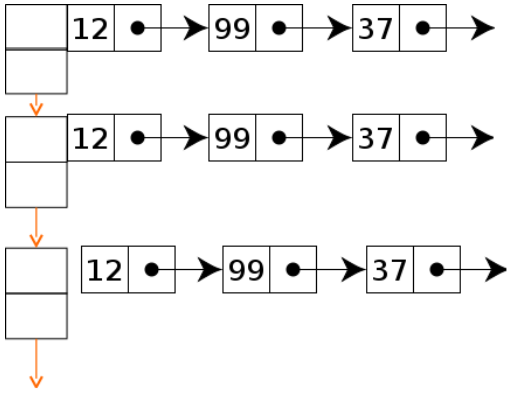
Listas doblemente enlazadas circulares

Una lista circular puede también ser doblemente encadenada (o enlazada):



Listas de listas

En las listas de listas, el campo de datos de un nodo puede ser otra lista enlazada.



Listas de prioridades

Esta estructura de datos es similar a la anterior pero las sublistas están ordenadas de acuerdo a una prioridad específica.

7.3 Casos de estudio

7.3.1 Tienda de Libros

Se debe desarrollar una aplicación usando el lenguaje de programación Java, empleando la implementación del TAD de una lista (puede ser una lista doblemente enlazada, circular o doblemente enlazada circular). La aplicación deberá manejar información relacionada con una tienda de libros, de acuerdo al siguiente ejemplo:

- ISBN: 130369977
- Titulo: Estructura de datos con C y C++
- Edicion: Segunda
- Idioma: Español
- Año: 2001
- Editorial: Prentice Hall
- Autores:

  - Nombres Apellidos
  - Yedidyah Langsam
  - Moshe J. Augenstein
  - Aaron M. Tenembaum
  - Nota: El número de autores que se pueden almacenar no se encuentra definido (esto le obliga a utilizar una lista de autores).

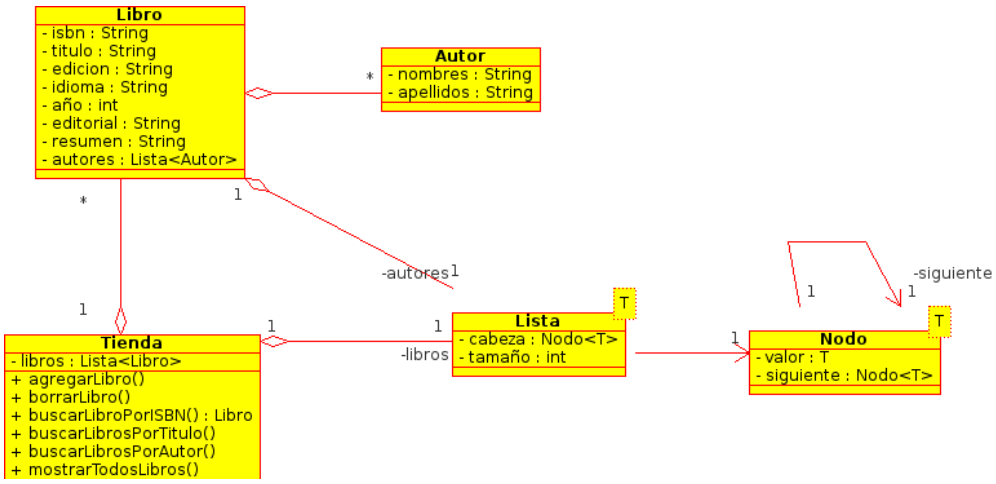
- Resumen: Este texto está diseñado para un curso de dos semestres de estructuras de datos y programación. El libro presenta los conceptos abstractos, muestra la utilidad de tales conceptos en la resolución de problemas, y después enseña cómo concretar las abstracciones usando un lenguaje de programación

El programa debe permitir realizar las siguientes operaciones:

- Agregar toda la información sobre un libro.
- Borrar un libro de la lista.
- Buscar un libro por ISBN y presentar de forma clara toda su información.

- Buscar llibros por Titulo y presentar de forma clara toda su información.
- Buscar los libros por autor.
- Mostrar el listado de todos los libros.

A continuación se da la respuesta en un diagrama de clases.



Se puede apreciar en el diagrama lo siguiente:

- La clase Autor representa un autor que tiene como atributos nombres y apellidos.
- La clase Libro representa un libro con todos los atributos solicitados. Como un libro tiene varios autores, éstos se almacenan en una Lista de Autores.
- La clase Tienda representa la tienda de libros, donde están las operaciones de los requisitos solicitados. Conceptualmente una tienda posee un conjunto de libros. En este caso el atributo libros es una Lista de Libros.
- Las Clases Lista y Nodo representan la Lista como tal.

La clase Tienda debería tener la siguiente estructura en java:

```
public class Libro{
        //Atributos
        private String isbn;
        private String titulo;
          private String edicion;;
          private String idioma
          private int a o;
          private String editorial;
          private String resumen;
        private Lista<Autor> autores;
       //Métodos
        //...
}
```

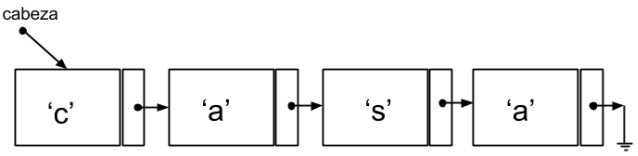
7.3.2 TDA String con Listas

Cree el TDA String implementándolo a través de listas, donde cada carácter se almacenará en un nodo. Implemente las siguientes operaciones del TDA.

1. Crear, que cree un String con los caracteres indicados en un arreglo de caracteres
2. Length, que calcule la longitud del String
3. Compare, que dos strings
4. Copy, que copia una cadena en un String, ejemplo: String S; S = String_Crear(); String_Copy(S, “mama”);

5. RemoveSubString, que dadas dos Strings, retorne una copia de la primera, que no incluya a la segunda cadena. Ejemplo: Si el primer String es “FUNDAMENTOS” y el segundo es “DAM”, el resultado es “FUNMENTOS”
6. LTrim, que dado un String remueva los espacios en blanco a la izquierda del String.
7. ReplaceString que dados tres Strings, donde la segunda es substring de la primera, reemplace en la primera dicha subcadena con la tercera. Ejemplo: Primera cadena “VIVIR”, Segunda cadena “VI”, Tercera cadena “ASA”, Resultado: “ASAASAR”

Para resolver este problema es necesario definir al tipo de dato String como una Lista de caracteres, e implementar cada una de las operaciones requeridas trabajando sobre el string como se trabajaría sobre la Lista. Ejemplo:



Así, si trabajásemos este problema usando lenguaje C, la definición del TDA String seria:

```
typedef Lista String;

String *newString(char cad[]);
int stringLength(String *s);
boolean stringCompare(String *s1, String *s2);
String *stringCopy(String *s, char cad[]);
String *stringRemoveSubString(String *s1, String *s2)
String *stringLTrim(String *s);
String *stringReplace(String *s1, String *s2);
```

A continuación la implementación sugerida para algunas de las operaciones arriba definidas:

```
String *newString(char cad[]){
    List *LString = crearLista();
    for(int i = 0; i < strlen(cad); i++)
    {
        agregar(LString, crearNodo(charAObjeto(cad[i]));
    }
    return LString;
}
int stringLength(String *s){
    Nodo *p;
    int l = 0;
    for(p = getPrimero(s); p!= getFin(s); p = getSiguiente(s,p)){
        l = l+1;
    }
    return l;
}
String stringTrim(String *s){
    Nodo *p = getPrimero(s);
    if(esVacia(s)) return s;
    do{
```

```
        char c = ObjectToChar(nodoGetValor(p));
        if(c == '␣') remover(s,p);
        p = getSiguiente(s,p);
    }
    while(c=='␣');
    return s;
}
```

7.3.3 Materias y Estudiantes

Se dispone de un archivo que contiene las materias dictadas en una facultad(Materias.dat)
ICM00604|ALGEBRA LINEAL(B)
ICM00216|CALCULO I(B)
ICQ00018|QUIMICA GENERAL I (B)
ICM00646|CALCULO II (B)
ICHE00885|ECOL.EDUC.AMB.(B)

Ademas, tenemos un archivo con los estudiantes registrados(Estudiantes.dat).
20020101|Raul Vaca
20012121|Juan Garcia
19929212|Luis Granda
19958823|Carlos Echeverria
29939323|Jorge Mendieta
29939121|Alberto Muñoz
29933232|Luis Alvarado
29934434|Jose Peña
29935544|Carlos Moreno
29912123|Jorge Ruiz
29933223|Maria Vera
29933443|Gabriel Vale

La facultad registró a sus estudiantes al inicio del semestre. Los registros fueron almacenados en un archivo Registros.dat. Cada linea de este archivo contiene un codigo de materia y la matricula de un estudiante que se registro en dicha materia.
Ejemplo:
ICM00604|20020101
ICHE00885|29934434
ICM00604|29939121
ICHE00885|20020101
ICM00216|29934434
ICM00216|29939323
ICM00604|19958823
ICHE00885|29934434
ICHE00885|29939323
ICM00604|29935544
ICM00604|29933443
ICM00216|29934434
ICHE00885|29933223

¿Como podemos darnos cuenta, un estudiante toma muchas materias y una materia es tomada

por muchos estudiantes. ¿La facultad necesita que un estudiante de Estructura de Datos cree un programa que permita, dados estos tres archivos, efectuar las siguientes consultas:

- Con la matricula de un estudiante, conocer que materias esta tomando
- Con el código de una materia, conocer que estudiantes la estan tomando

7.3.4 Análisis del problema

¿Qué entidades reconoce en este problema? Pueden reconocerse, fácilmente: Estudiante y Materia.

Ademas, existe una entidad especial que ENLAZA a un Estudiante con una Materia o visceversa, llamemosla Registro. Se puede decir entonces que

- Una materia recepta muchos registros y en ese caso el registro enlazara a materia con estudiante
- Un estudiante efectua muchos registros y en este caso el registro enlazara a estudiante con materia

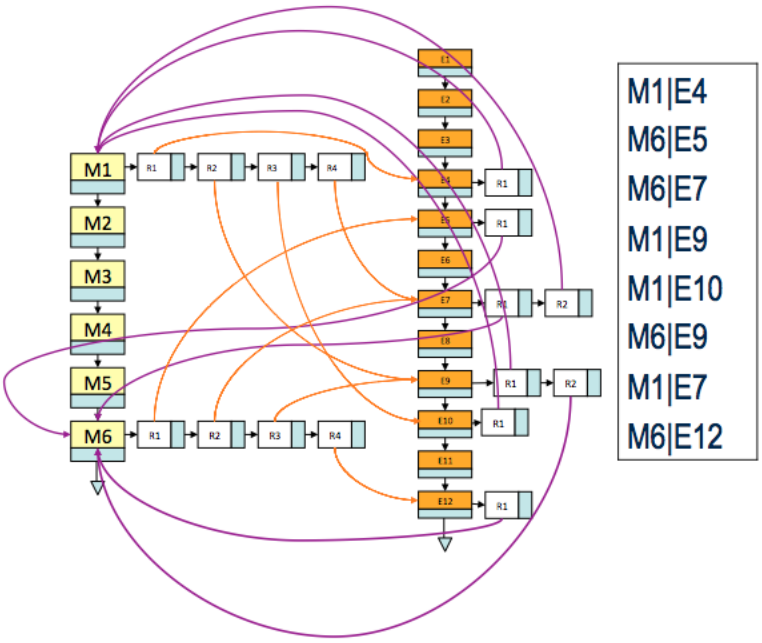
Ahora podemos definir los atributos de cada uno de los TDAs que representarán a las entidades:
Materia
- ‘Codigo
- ‘Descripción
- ‘Conjunto de Registros(Lista)

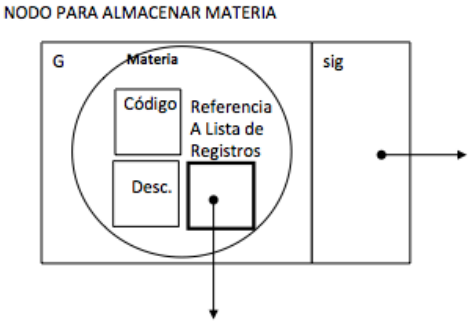
Estudiante
- ‘Matricula
- ‘Nombre
- ‘Apellido
- ‘Conjunto de Registros(Lista)

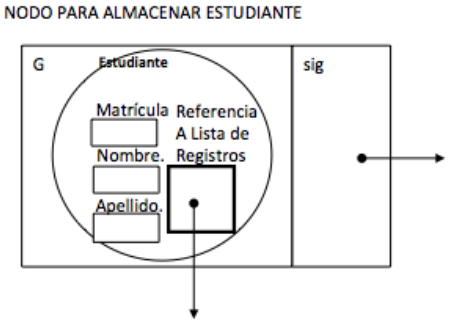
Registro
- ‘Codigo de materia
- ‘Matricula de estudiante
- ‘Referencia a la materia
Referencia al estudiante

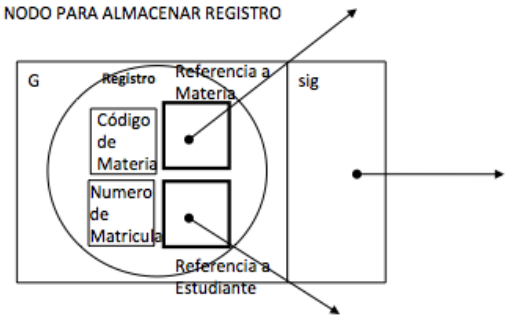
Propuesta de Diseño de Solución

Entonces, como parte del diseño de la solución manejaremos una Lista de Materias y una Lista de Estudiantes. Por otro lado, cada Materia y cada Estudiante tendrán como atributo su Lista de Registros. Cada Registro de esta lista tiene como objetivo proveer referencias a la Materia y al Estudiante que lo componen. La siguiente gráfica busca ilustrar el diseño planteado.









Una posible definición en lenguaje C de los atributos de los TDAs identificados podría ser:

```
typedef struct{
  char  codigo[100];
  char  nombre[100];
  Lista *LEstudiantes;
 } Materia;
typedef struct{
  char  matricula[100];
  char  nombre[100];
  char  apellido[100];
```

```
  Lista *LMaterias;
} Estudiante;
typedef struct{
  char  cod_mat[100];
  char  cod_est[100];
  Estudiante *refEstudiante;
  Materia *refMateria;
 } Registro;
```

Una vez que los datos de los archivos sean cargados en las listas de Materias, de Estudiantes y de Registros, y los enlaces de las referencias definidas sean efectuados, ya se pueden llevar a cabo las consultas solicitadas:

- Dada una materia, deseamos conocer los estudiantes que en ella se registraron
- Dado un estudiante, deseamos conocer en que materias se ha registrado

A continuación se propone una forma de realizar la primera consulta, implementada en lenguaje C. Esta retorna una Lista con los resultados de la consulta:

```
Lista* consultarEstudiantesxMateria(Lista *LMaterias, char *codMateria)
{
 Nodo *preg;
  Materia *mencontrada;
  List *LRegistros, *LEstudiantesResultadoConsulta = crearLista();
  mencontrada = buscarMateria(LMaterias,codMateria);
  if(!mencontrada) return LEstudiantesResultadoConsulta;
  LRegistros = materiaObtenerListaRegistros(mencontrada);
  for(preg = getPrimero(LRegistros); preg!=getFin(LRegistros); preg=
      getSiguiente(LRegistros,preg)){
      Registro *r = nodoGetValor(preg);
      Estudiante *e = registroGetEstudiante(r);
      agregar(LEstudiantesResultadoConsulta,crearNodo(e));
  }
  return LEStudiantesResultadoConsulta;
}
```

7.4 Ejercicios propuestos

1. Concatenar dos listas lst1 y lst2, dejando el resultado en la primera de ellas. Asumir m como la longitud de la segunda lista:

/* pre: lst1 = < x1, . . . , xn >, lst2 = LST2 = < y1, . . . , ym > */
/* post: lst1 = < x1, . . . , xn, y1, . . . , ym >, lst2 = LST2 */

public void concatLista(Lista<T> lst1, Lista<T> lst2)

---

2. Crear un método que determine si dos listas son iguales:

/* post: igualesListas = (st1 es igual a st2) */
public boolean igualesListas(Lista<T> lst1, Lista<T> lst2);

---

3. Crear un método que localice un elemento en la lista. Si hay varias ocurrencias del elemento, buscar la primera. Si el elemento no se encuentra en la lista, debe devolver null:
public T buscar(T elemento)

---

4. Crear un método que permita copiar una lista:

```
/* post: copiarLista es una copia de lst */
public Lista copiarLista( Lista<T> lst )
```

---

5. Invertir una lista, destruyendo la lista original:

```
/* pre: lst = < x1,...,xn > */
/* post: invLista = < xn, ..., x1 >, lst = < > */

public Lista<T> invLista( Lista<T> lst )
```

---

6. Eliminar todas las ocurrencias de un elemento en una lista:

```
/* post: se han eliminado todas las ocurrencias de elem en lst */
public void elimTodosLista(T elem )
```

---

7. Indicar si una lista de números enteros se encuentra ordenada ascendentemente:

```
/* pre: lst = < x1,..., xn > */
/* post: ordenadaLista = ( x i <= xi+1 ) */
```

---

8. Se quiere representar el tipo abstracto de datos conjunto de tal forma que los elementos este almacenados en una lista enlazada. Escribir una unidad para implementar el TAD conjunto mediante listas. En la unidad debera contener los tipos de datos necesarios y las operaciones:
a. Conjunto vacio,
b. Añadir un elemento al conjunto
c. Union e intersección de conjuntos, diferencia de conjuntos
d. Los elementos seran detipo cadena

---

9. Escriba un programa que forme lista ordenada de registros de empleados. La ordenación ha de ser respecto al campo entero Sueldo. Con esta lista ordenada realice la siguientes acciones: Mostrar los registros cuyo sueldo S es tal que p1 <= s <= p2, aumentar en un 7 % el sueldo de los empleados que ganan menos de P pesetas. Aumentar en un 3 % el sueldo de los empleados que ganan mas de P pesetas. dar de baja a los empleados con mas de 35 años de antigüedad

---

10. Se quiere listar en orden alfabrtica las palabras de que consta un archivo de texto junto con los numeros de linea en que aparecen. Para ello hay que utilizar una estructura multienlazda en la que la lista directorio es la lista ordenada de palabras. De cada nodo con la plabra emerge otra lista con los numeros de linea en que aparece la palabra

---

11. Escriba un procedimiento que reciba como parámetro tres listas enlazadas y ordenadas y cree

una lista doblemente enlazada circular con los nodos que aparecen en dos de las tres listgas

---

12. Escriba un procedimiento que reciba una lista enlazda y la convierta en una lista circular

---

13. Sea L una lista doblemente enlazada. Escriba un programa que transforme la lista en una doblememente enlazda sin caracteres repetidos

---

14. Escribir una funcion que reciba la entrada la lista L de numeros enteros, que tiene numeros repetidos. El procedimiento creara otra lista cuyos nodos contendran las direcciones de los nodos repetidos en la lista L. Definir los tipos de datos para represenar ambas listas

---

15. Dada una lista ordenada con palabras repetidas, escribir un procedimiento que inserte una palabra en la lista, si la calve ya esta en la lista, se inserte al final de todas las que tienen la misma clave

---

16. El polinomio P(x) = a0 + a1x + a2x2 + ... + anxn deseamos representarlo en una lista enlazada de tal forma que cada nodo contenga el coeficiente y el grado de un monomio. Escribir un programa que tenga como entrada los coeficientes y exponenetes de cada termino de un polinomio y forme una lista enlazada, siempre en orden decreciente.
a. Evalue el polinomio con un valor para x
b. Derive el polinomio ->otro polinomio
c. Sume dos polinomios -> otro polinomio

---

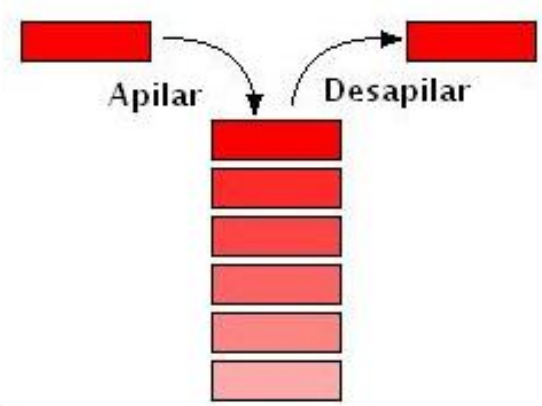
17. Cree el TDA empleados(nombre, apellido, cedula, años en la empresa y sueldo). Una lista enlazada ordenada almacenara registros de empleados. Escriba las siguientes rutinas
a. ListarEntre que recibe dos sueldos, y lista aquellos empleados con sueldos entre lo recibido
b. Aumentar que aumenta n porciento a los empleados que reciben mas de P sueldo.
c. DarBaja que saca del listado a los empleados con mas de a años de antiguedad.

---

18. Un organizador de fotos permite clasificarlas por eventos. Un evento esta definido por un codigo numerico, nombre, una fecha de inicio, así como ciudad donde se llevo a cabo y esta asociada a un grupo de fotos.
Una foto esta definida por un nombre, fecha en la que fue tomada, sitio donde fue tomada y codigo de evento a la que pertenece.
Si posee un archivo eventos.txt y otro foto.txt, cree un programa que los cargue en dos listas. Luego asocie dichas listas entre ellas para representar la relación entre un evento y una foto. Permita las siguientes consultas:
1. Dado una ciudad, mostrar todos los eventos que se llevaron a cabo en esa ciudad ordenados por fecha de inicio
2. Mostrar todos los eventos y permitir elegir uno de ellos, luego mostrar todas las fotos de dicho evento
3. Dado un sitio, mostrar todos los eventos en los que se tomaron fotos de dicho sitio

8 — TDAs Lineales: Pilas

Una **pila** (stack en inglés) es una lista ordinal o estructura de datos en la que el modo de acceso a sus elementos es de tipo LIFO (del inglés Last In First Out, último en entrar, primero en salir) que permite almacenar y recuperar datos. Se aplica en multitud de ocasiones en informática debido a su simplicidad y ordenación implícita en la propia estructura.



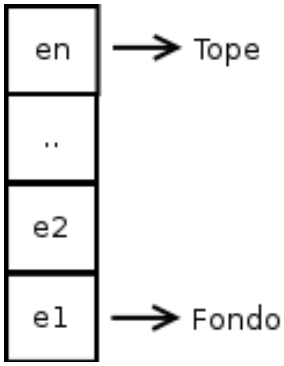
Las pilas son muy utilizadas en programación, para evaluar expresiones, reconocer lenguajes, recorrer árboles y simular procesos recursivos. En todo momento, el único elemento visible de la estructura es el último que se colocó. Se define el tope de la pila como el punto donde se encuentra dicho elemento, y el fondo, como el punto donde se encuentra el primer elemento incluído en la estructura

8.1 Definición y Formas de Uso

8.1.1 Definición

La administración de una pila se puede hacer con muy pocas operaciones: una constructora (permite crear pilas vacías), dos modificadoras (para agregar y eliminar elementos) y dos analizadoras (retornar el elemento del tope, e informar si la pila está vacía). Se incluye, además, una destructora para retornar el espacio ocupado por la pila. Por simplicidad, no se contemplan operaciones de persistencia.

El formalismo escogido para referirse al objeto abstracto pila se muestra a continuación. Allí se da un nombre a cada uno de los elementos que hacen parte de la estructura y se marca claramente el tope y el fondo.



Si la pila no tiene ningún elemento se dice que se encuentra vacía y no tiene sentido referirse a su tope ni a su fondo. Una pila vacía se representa con el símbolo Ø. Por último, se define la longitud de una pila como el número de elementos que la conforman.

Una pila tiene las siguientes operaciones básicas:
Crear: se crea la pila vacía
Apilar: (push), que coloca un elemento en la pila
Retirar (o desapilar, pop), es la operación inversa que retira el último elemento apilado
Cima: devuelve el elemento que esta en la cima de la pila (top o peek).
Vacía: devuelve true si la pila está vacía o falso en caso contrario.

Especificación

TAD Pila [T]
{ invariante: TRUE }
Constructoras:
crearPila()
Modificadoras:
apilar()
desapilar()
Analizadoras:
cima()
esVacia()
Destructora:
destruirPila()

---

crearPila(void)
/* Crea una pila vacía */
{ post: crearPila = }

---

void apilar(T elem)
/* Coloca sobre el tope de la pila el elemento elem */
{ post: pil = e1, e2, .. elem}

---

void desapilar()
/* Elimina el elemento que se encuentra en el tope de la pila */
{ pre: pil =e1, e2, ..en, n > 0 }
{ post: pil =e1, e2, .., en-1 }

T cima()
/* Retorna el elemento que se encuentra en el tope de la pila */
{ pre: n > 0 }
{ post: cima = en }

---

boolean esVacia()
/* Informa si la pila está vacía */
{ post: esVacia = (pil =) }

---

void destruirPila()
/* Destruye la pila retornando toda la memoria ocupada */
{post: pil ha sido destruida }

8.1.2 Formas de Uso

Una vez definida la Pila podemos usarla en la solución del siguiente problema: una vez que un conjunto de números fue recibido por teclado, se necesita mostrarlos en pantalla pero en el orden inverso al cual fueron ingresados. Este es un ejercicio donde típicamente se usa una pila, puesto que los elementos que fueron almacenados en la misma fueron acumulándose, de tal forma que el último elemento en ser ingresado se encuentra al tope. Cuando se requiera obtener uno a uno los datos almaceados, la Pila los irá dando desde el último hacia el primero: en orden inverso.

Se propone una solución en pseudocódigo con enfoque orientado a objetos.

```
procedimiento imprimirInverso()
    var
        Pila P = crearPila()
        int valor
    Inicio
        Hacer
            valor = leer()
            P.apilar(valor);
        Mientras valor!=99;

        Mientras (!P.esVacia())
            valor = P.desapilar();
            escribir(valor);
        FinMientras
    Fin
```

8.2 Implementaciones y Algoritmos fundamentales.

8.2.1 Implementación en Java

A continuación se muestra una implementación en java de una estructura pila.

```
package capitulo2.pilas;
public class Nodo<T> {
        private T valor;
        private Nodo<T> siguiente;

        public Nodo() {
                valor = null;
                siguiente = null;
```

```
        }

        public T getValor() {
                return valor;
        }

        public void setValor(T valor) {
                this.valor = valor;
        }

        public Nodo<T> getSiguiente() {
                return siguiente;
        }

        public void setSiguiente(Nodo<T> siguiente) {
                this.siguiente = siguiente;
        }
}

package capitulo2.pilas;
public class Pila<T> {
        private Nodo<T> cabeza;
        private int tamanio;

        public Pila() {
                cabeza = null;
                tamanio = 0;
        }

        public int getTamanio() {
                return tamanio;
        }

        public boolean esVacia() {
                return (cabeza == null);
        }

        public void apilar(T valor) {
                Nodo<T> nuevo = new Nodo<T>();
                nuevo.setValor(valor);
                if (esVacia()) {
                        cabeza = nuevo;
                } else {
                        nuevo.setSiguiente(cabeza);
                        cabeza = nuevo;
                }
                tamanio++;
        }

        public void retirar() {
                if (!esVacia()) {
                        cabeza = cabeza.getSiguiente();
                        tamanio--;
                }
        }

        public T cima() {
                if (!esVacia())
                        return cabeza.getValor();
                else
                        return null;
```

```
        }
}

package capitulo2.pilas;
public class ClienteMain {
        public static void main(String[] args) {
                Pila<Integer> pila2 = new Pila<Integer>();
                pila2.apilar(2);
                pila2.apilar(5);
                pila2.apilar(7);
                System.out.println(pila2.cima());
                pila2.retirar();
                System.out.println(pila2.cima());
                pila2.retirar();
                System.out.println(pila2.cima());
                pila2.retirar();
                System.out.println(pila2.cima());
                //Probar con otra pila, donde se almacenen objetos tipo Persona o Contacto o Libro, etc.
//Algo así;
                //Pila <Contacto>pila2 = new Pila<Contacto>();
                //pila2.apilar(new Contacto(2,"Juan Perez", "31245434","juanito@hotmail.com");
                //pila2.desapilar();
                //...
        }
}
```

El ClienteMain sirve para probar la pila, dentro de su main, invoca operaciones de apilar y retirar y en cada una, imprime por consola lo que haya en la pila.

8.2.2 Implementación en C++

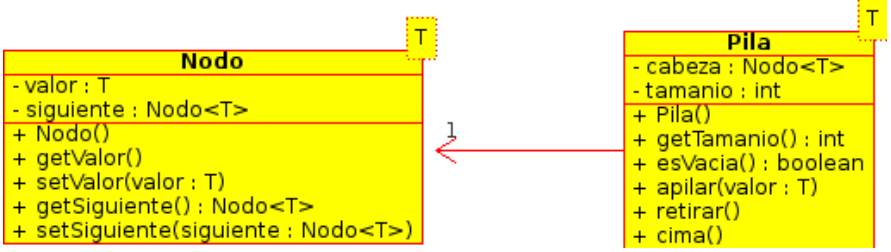
La implementación en lenguaje C++ sería de la siguiente forma:

```
 // pila.h
template <typename T>
class Pila
{
    class Nodo
    {
        private: T valor;
        Nodo *pSiguiente;
        public:
          Nodo(T _valor, Nodo *pSiguiente = NULL){
          {
            setValor(_valor);
            setSiguiente(_pSiguiente);
          }
          T getValor() {        return valor;       }
          void setValor(T &_valor) { valor = _valor;              }
          Nodo *getSiguiente() {        return pSiguiente;       }
          void setSiguiente(Nodo *_pSiguiente) { pSiguiente = _pSiguiente;
                    }
        };

        private:
          Nodo *pCabeza;
          int tamanio;
        public:
          Pila(): pCabeza(NULL), tamanio(0)        {        }
          int getTamanio() {        return tamanio;              }
          boolean esVacia(){        return (pCabeza == NULL);       }
```

```
        void apilar(T valor)
        {
          new Nodo(valor, pCabeza);
          tamanio++;
        }
        void retirar()
        {
          if (esVacia())
              throw Pila vac a ;
          Nodo *pNodo = pCabeza;
          pCabeza = pCabeza->getSiguiente();
          delete pNodo;
          tamanio--;
        }

        T cima()
        {
           if (esVacia())
               return null;
            return pCabeza->getValor();
        }
    };

// main.cpp
#include <iostream.h>
#include pila . h
void main()
{
    Pila<int> pila;
    pila2.apilar(2);
    pila2.apilar(5);
    pila2.apilar(7);
    cout << pila2.cima();
    pila2.retirar();
    cout << pila2.cima();
    pila2.retirar();
    cout << pila2.cima();
    pila2.retirar();
    cout << pila2.cima();

    //Probar con otra pila, donde se almacenen objetos tipo Persona o Contacto o Libro, etc.
    //Algo así;
    //Pila <Contacto *>pila3;
    //pila3.apilar(new Contacto(2,"Juan Perez", "31245434","juanito@hotmail.com");
    //pila3.retirar();
    //...
}
```

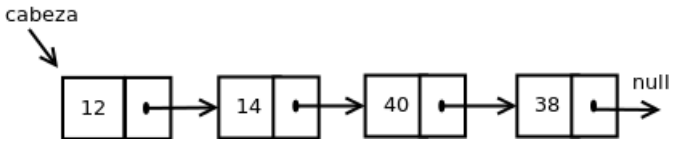
8.3 Formas de representación.

8.3.1 Representación como Objetos



La figura anterior muestra un diagrama de clases que permite implementar la estructura de datos Pila. La clase Nodo tiene dos atributos: valor que almacena un elemento de cualquier tipo (T) y siguiente, que es una referencia a otro nodo. La Clase Pila representa la pila como tal con todas las operaciones definidas en el TAD. La clase Pila tiene una referencia al primer nodo llamado cabeza.

El siguiente diagrama muestra la pila después de haberle aplicado las operaciones: apilar(38), apilar(40), apilar(14), apilar(12). Se puede apreciar que la cabeza siempre apunta al último elemento apilado.



8.4 Casos de Estudio

8.4.1 Correspondencia de delimitadores

La pila es muy útil en situaciones cuando los datos deben almacenarse y luego recuperarse en orden inverso. Una aplicación de la pila es la correspondencia de delimitadores en un programa. Esto es un ejemplo importante debido a que la correspondencia de delimitadores es parte de cualquier compilador. Ningún programa se considera correcto si los delimitadores no tienen su pareja. Ejemplo:

```
while (m<(n[8]+o)) { p = 7; /* comentarios */ 6=6;}
```

El algoritmo que permite evaluar la correspondencia de delimitadores de una expresión de izquierda a derecha y se siguen los siguientes pasos:

1. Obtener caracter de la expresión y repetir pasos 2 al 3 para cada caracter.
2. Si es un operador de apertura: (, {, [, /* se lo apila.
3. Si es un operador de cierre:), },], */:
   3.1. Comparar que corresponda con el operador del tope de la pila.
   3.2. Si no corresponde, termina el programa y la expresión es incorrecta.
   3.3. Si hay correspondencia, se elimina elemento del tope de la pila y volver al paso 1.
4. Si al terminar de evaluar la expresión quedan elementos en la pila, la expresión es incorrecta.
5. Si al terminar de evaluar la expresión la pila queda vacía, la expresión es correcta.
6. Fin de algoritmo.

El anterior algoritmo tiene como base fundamental, la utilización de una pila. Sin ella, sería muy complicado evaluar la validez de la expresión.

8.4.2 Evaluación de expresiones aritméticas

Las pilas se utilizan para evaluar expresiones aritméticas. Por ejemplo:

```
(100 + 23) * 231 / 33 ^2 – 34
```

Para evaluar estas expresiones se deben pasar a la notación postfija y luego aplicar un algoritmo para evaluar la expresión en notación postfija.

Conceptos

- La expresión A+B se dice que esta en notación infija y su nombre se debe a que el operador + está entre los operandos A y B.
- Dada la expresión AB+ se dice que esta en notación postfija y su nombre se debe a que el operador + está después de los operandos A y B.
- Dada la expresión +AB se dice que esta en notación prefija y su nombre se debe a que el operador + está antes que los operandos A y B.

La ventaja de usar expresiones en notación postfija es que no son necesarios los paréntesis para indicar orden de operación, ya que éste queda establecido por la ubicación de los operadores con respecto a los operandos.

Expresión infija: (X + Z) * W / T ^Y – V
Expresión postfija: X Z + W * T Y ^/ V –

8.4.3 Convertir una expresión dada en notación INFIJA a una notación POSTFIJA

Para convertir una expresión dada en notación INFIJA a una notación POSTFIJA, deberá establecerse previamente ciertas condiciones:

- Solamente se manejarán los siguientes operadores (Están dados ordenadamente de mayor a menor según su prioridad de ejecución):

Operador	Prioridad dentro de la pila	Prioridad fuera de la pila
^(potencia)	3	3
*, /	2	2
+, -	1	1
(	0	4

- Los operadores de más alta prioridad se ejecutan primero. Si hubiera en una expresión dos o más operadores de igual prioridad, entonces se procesan de izquierda a derecha.
- Las subexpresiones que se encuentren dentro de paréntesis tendrán más prioridad que cualquier operador.
- Obsérvese que no se trata el paréntesis derecho ya que éste provoca sacar operadores de la pila hasta alcanzar el paréntesis izquierdo.
- Se parte de una expresión en notación infija que tiene operandos, operadores y puede tener paréntesis. Los operandos vienen representados por letras y los operadores son: ^, *, /, +, -
- La transformación se realiza utilizando una pila en la cual se almacenan los operadores y los paréntesis izquierdos.
- La expresión aritmética se va leyendo desde el teclado de izquierda a derecha, caracter a caracter, los operandos pasan directamente a formar parte de la expresión en postfija la cual se guarda en un arreglo.
- Los operadores se meten en la pila siempre que ésta esté vacía, o bien siempre que tengan mayor prioridad que el operador de la cima de la pila (o bien si es la máxima prioridad).
- Si la prioridad es menor o igual se saca el elemento cima de la pila y se vuelve a hacer comparación con el nuevo elemento cima.
- Los paréntesis izquierdos siempre se meten en la pila; dentro de la pila se les considera de mínima prioridad para que todo operador que se encuentra dentro del paréntesis entre en la pila.
- Cuando se lee un paréntesis derecho hay que sacar todos los operadores de la pila pasando a formar parte de la expresión postfija, hasta llegar a un paréntesis izquierdo, el cual se elimina ya que los paréntesis no forman parte de la expresión postfija.
- El proceso termina cuando no hay más elementos de la expresión y la pila esté vacía.

El algoritmo que permite convertir una expresión de infija a postfija es la siguiente:

1. Obtener caracteres de la expresión y repetir pasos 2 al 5 para cada caracter.
2. Si es un operando pasarlo a la expresión postfija (es decir, concatenarlo)
3. Si es un operador:
   3.1. Si la pila está vacía, meterlo en la pila. Repetir a partir del paso 1.
   3.2. Si la pila no está vacía:
   - Si la prioridad del operador leído es mayor que la prioridad del operador cima de la pila, meterlo en la pila y repetir a partir del paso 1.
   - Si la prioridad del operador es menor o igual que la prioridad del operador de la cima, sacar operador cima de la pila y pasarlo a la expresión postfija.
   - Volver al paso 3.
4. Si es paréntesis derecho:
   4.1. Sacar operador cima de la pila y pasarlo a la expresión postfija.
   4.2. Si nueva cima es paréntesis izquierdo, suprimir elemento cima.
   4.3. Si cima no es paréntesis izquierdo volver a paso 4.1
   4.1. Volver a partir del paso 1.
5. Si es paréntesis izquierdo pasarlo a la pila.
6. Si quedan elementos en la pila, pasarlos a la expresión postfija
7. Fin de algoritmo

A continuación un ejemplo donde se aplica el algoritmo paso a paso. Pasar la expresión infija (X+Z)*W/T^Y-V a su equivalente postfija:

Caracter leído	Pila	Expresión Postfija
(	(	
X	(	X
+	(+	X
Z	(+	XZ
)		XZ+
*	*	XZ+
W	*	XZ+W
/	/	XZ+W*
T	/	XZ+W*T
^	/^	XZ+W*T
Y	/^	XZ+W*TY
-	-	XZ+W*TY^/
V	-	XZ+W*TY^/V
		XZ+W*TY^/V-

8.4.4 Evaluación de la Expresión en notación postfija

Se almacena la expresión aritmética transformada a notación postfija en un vector, en la que los operandos están representados por variables de una sola letra. Antes de evaluar la expresión requiere dar valores numéricos a los operandos.

Una vez que se tiene los valores de los operandos, la expresión es evaluada.

El algoritmo de evaluación utiliza una pila de operandos, en definitiva de números reales. El algoritmo es el siguiente:

1. Examinar el vector desde el elemento 1 hasta el N, repetir los pasos 2 y 3 para cada elemento del vector.
2. Si el elemento es un operando meterlo en la pila.

3. Si el elemento es un operador, lo designamos por ejemplo con +:
   * Sacar los dos elementos superiores de la pila, los denominamos con los identificadores x,y respectivamente.
   * Evaluar x+y; el resultado es z = x + y.
   * El resultado z, meterlo en la pila.
   * Repetir a partir del paso 1.
4. El resultado de la evaluación de la expresión está en el elemento cima de la pila.
5. Fin de algoritmo.

A continuación un ejemplo donde se aplica el algoritmo paso a paso. Se toma como expresión postfija **XZ+W*TY^/V-**, y se reemplazan los siguientes valores: X=5, Z=3, W=4, T=2, Y=8, V=100

Es decir, la expresión postfija quedaría así:
5 3 + 4 * 2 3 ^/ 100 -

Elemento leído	Pila
5	5
3	5 3
+	8
4	8 4
*	32
2	32 2
3	32 2 3
^	32 8
/	4
100	4 100
-	**-96**

8.5 Ejercicios Propuestos

- Utilizando el algoritmo estudiado, pasar a notación POSTFIJA las siguientes expresiones:
(X+Y*Q)/R^T *RTA: XYQ*+RT^/*
X^Y+Z-A*D-R *RTA: XY^Z+AD*-R-*
A+(B-D)/Y–(A-C)/(Y-X)+Z *RTA: ABD-Y/+AC-YX-/-Z+*

- De las anteriores expresiones POSTFIJAS, dar valores a las variables y aplicar el algoritmo que permite evaluar una expresión POSTFIJA.

- Buscar en Internet un algoritmo que permita pasar expresiones de notación INFIJA a notación PREFIJA. Entender el algoritmo mediante ejemplos.

- Proponer una implementación en un lenguaje de programación que permita pasar expresiones en notación infija a postfija. Luego que permita sustituir las variables por valores numéricos reales, y finalmente, que permita evaluar la expresión

- Crear un algoritmo que permita Invertir una lista utilizando una pila
/* pre: lst = < x1, , xN > */
/* post: lst = < xN, , x1 > */
public void invLista(Lista<T> lst)

- Crear un algoritmo que copie una pila.

/* pre: pil = PIL = */

/* post: copiarPila = , pil = PIL */

- Crear un algoritmo que decidir si dos pilas son iguales sin destruir su contenido

/* pre: pil1 = , pil2 = */
/* post: igualesPilas = (N = M xk = yk k N */

- Crear un algoritmo que sume todos los elementos de la pila y retorne el resultado

9 — Estructuras de Datos Lineales. Colas.

9.1 Aspectos Teóricos de las Colas

Una **cola** es una estructura de datos, caracterizada por ser una secuencia de elementos en la que la operación de **inserción** se realiza por un extremo y la operación de **extracción** por el otro. También se le llama estructura **FIFO** (del inglés *First In First Out*), debido a que el primer elemento en entrar será también el primero en salir.

Las colas se utilizan en sistemas informáticos, transportes y operaciones de investigación (entre otros), donde los objetos, personas o eventos son tomados como datos que se almacenan y se guardan mediante colas para su posterior procesamiento.

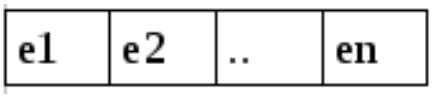
Las colas se utilizan mucho en los procesos de simulación, en los cuales se quiere determinar el comportamiento de un sistema que presta servicio a un conjunto de usuarios, quienes esperan mientras les toca el turno de ser atendidos. Como ejemplos de estos sistemas se pueden nombrar los bancos, los aeropuertos (los aviones hacen cola para despegar y aterrizar) y los procesos dentro de un computador. Las colas también se utilizan en muchos algoritmos de recorrido de árboles y grafos.

Representación de una Cola

9.2 Especificación de las colas

9.2.1 Especificación semi formal del TAD Cola

El formalismo escogido para expresar el estado de un objeto abstracto Cola es:



Se define la longitud de una cola como el número de elementos que la conforman. Si la longitud es cero (no tiene ningún elemento), se dice que la cola está vacía.

TAD Cola [T]
{ invariante: TRUE }
Constructoras:
crearCola()
Modificadoras:
encolar()
desencolar()
Analizadoras:
frente()
esVacia()
Destructora
destruirCola()

---

crearCola()
/* Crea una cola vacía */
{ post: crearCola = }

---

void encolar(T elem)
/* Agrega elem al final de la cola*/
{ post: col = e1, e2, .. elem}

---

void desencolar()
/* Elimina el primer elemento de la cola */
{ pre: n > 0 }
{ post: col = e2, .., en }

T frente()
/* Retorna el primer elemento de la cola */
{ pre: n > 0 }
{ post: frente = e1 }

---

boolean esVacia()
/* Informa si la cola está vacía */
{ post: esVacia = (col =) }

---

void destruirCola()
/* Destruye la cola retornando toda la memoria ocupada */
{post: La cola ha sido destruida }

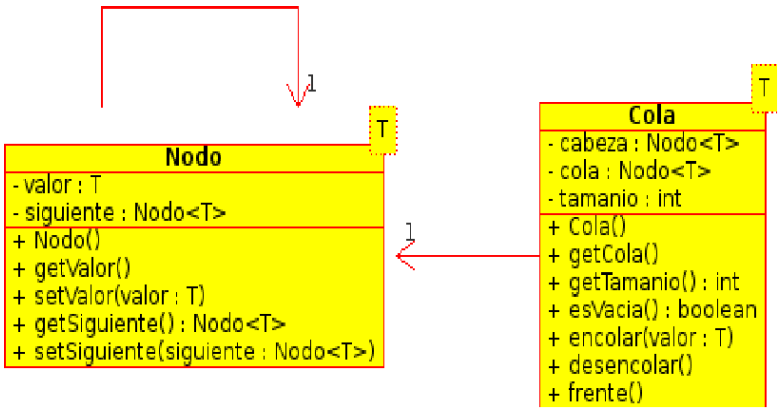
9.2.2 Especificación no formal

Las operaciones básicas de una cola son:

- **Crear**: se crea la cola vacía.
- **Encolar** (añadir, entrar, push): se añade un elemento a la cola. Se añade al final de ésta.
- **Desencolar** (sacar, salir, pop): se elimina el elemento frontal de la cola, es decir, el primer elemento que entró.
- **Frente** (consultar, front): se devuelve el elemento frontal de la cola, es decir, el primero elemento que entró (cabeza).

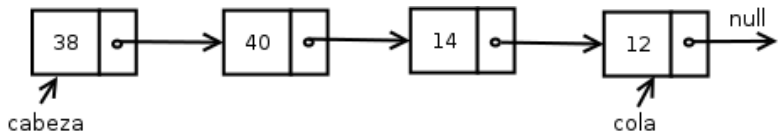
9.3 Formas de Representación

Este tipo de estructura de datos abstracta se implementa en lenguajes orientados a objetos mediante clases, en forma de listas enlazadas.



La figura anterior muestra un diagrama de clases que permite implementar la estructura de datos Cola. La clase Nodo tiene dos atributos: valor que almacena un elemento de cualquier tipo (T) y siguiente, que es una referencia a otro nodo. La Clase Cola representa la cola como tal con todas las operaciones definidas en el TAD. La clase Cola tiene una referencia al primer nodo llamado cabeza, y además, una referencia al último nodo llamado cola.

El siguiente diagrama muestra la cola después de haberle aplicado las operaciones: encolar(38), encolar(40), encolar(14), encolar(12). Se puede apreciar que la cabeza siempre apunta al primer elemento y cola apunta al último elemento.



9.4 Implementaciones y Algoritmos Fundamentales

9.4.1 Implementación en Java

A continuación se muestra una implementación en java de una estructura cola.

```
package capitulo3.colas;
public class Nodo<T> {
        private T valor;
        private Nodo<T> siguiente;

        public Nodo() {
                valor = null;
                siguiente = null;
        }

        public T getValor() {
                return valor;
        }

        public void setValor(T valor) {
                this.valor = valor;
        }

        public Nodo<T> getSiguiente() {
```

```
                return siguiente;
        }

        public void setSiguiente(Nodo<T> siguiente) {
                this.siguiente = siguiente;
        }
}

package capitulo3.colas;
public class Cola<T> {
        private Nodo<T> cabeza; //Primer elemento de la cola
        private Nodo<T> cola; //Ultimo elemento de la cola
        private int tamanio;

        public Cola() {
                cabeza = null;
                cola = null;
                tamanio = 0;
        }

        public T getCola() {
                return cola.getValor();
        }

        public int getTamanio() {
                return tamanio;
        }

        public boolean esVacia() {
                if (cabeza == null)
                        return true;
                else
                        return false;
        }

        public void encolar(T valor) {
                Nodo<T> nuevo = new Nodo<T>();
                nuevo.setValor(valor);
                if (esVacia()) {
                        cabeza = nuevo;
                        cola = nuevo;
                } else {
                        cola.setSiguiente(nuevo);
                        cola = nuevo;
                }
                tamanio++;
        }

        public void desencolar() {
                if (!esVacia()) {
                        if (cabeza == cola) {
                                cabeza = null;
                                cola = null;
                        } else {
                                cabeza = cabeza.getSiguiente();
                        }
                        tamanio--;
                }
        }

        public T frente() {
```

```
                if (!esVacia())
                        return cabeza.getValor();
                else
                        return null;
        }

}

package capitulo3.colas;
public class ClienteMain {

        /**
         * @param args
         */
        public static void main(String[] args) {
                Cola<Integer> cola = new Cola<Integer>();
                cola.encolar(12);
                cola.encolar(13);
                cola.encolar(14);
                cola.encolar(15);


                System.out.println(cola.frente());

                cola.desencolar();
                System.out.println(cola.frente().toString());

                cola.desencolar();
                System.out.println(cola.frente());

                cola.desencolar();
                System.out.println(cola.frente());

                cola.desencolar();
                System.out.println(cola.frente());

                //Probar con otra cola, donde se almacenen objetos tipo Persona
        }
}
```

El ClienteMain sirve para probar la cola, dentro de su main, invoca operaciones de encolar, desencolar y frente.

9.5 Casos de Estudio del uso de Colas

Simulador de procesos de un sistema operativo

Se debe utilizar una cola para simular el algoritmo de planificación de procesos round-robin (turno circular) de un sistema operativo. A continuación se describe su funcionamiento.

Los sistemas operativos modernos permiten ejecutar simultáneamente dos o más procesos. Cuando hay más de un proceso ejecutable, el sistema operativo debe decidir cuál proceso se ejecuta primero. La parte del sistema operativo que toma esta decisión se llama planificador; el algoritmo que usa se denomina algoritmo de planificación.

El algoritmo de planificación de procesos Round Robin es uno de los más sencillos y antiguos. A cada proceso se le asigna un intervalo de tiempo en la CPU, llamado cuanto, durante el cual se le permite ejecutarse. Si el proceso todavía se está ejecutando al expirar su cuanto, el sistema operativo se apropia de la CPU y se la da a otro proceso. Si el proceso se bloquea o termina antes

de expirar el cuanto , se hace la conmutación de proceso. Para ello, el planificador mantiene una lista de procesos ejecutables. Cuando un proceso gasta su cuanto, se le coloca al final de la lista

Los requisitos del simulador son:

Utilizando una cola de procesos, simular la planificación de procesos con el algoritmo round-robin.
Construir un archivo (procesos.txt) con la información de los procesos según su orden de llegada. El formato debe ser así (identificador de proceso, tiempo de cpu requerido en ms), por ejemplo:
P1, 100
P2,15
P3,40
Construir un programa que simule la operación de la ejecución de procesos a partir de la información del archivo procesos.txt y un valor de cuanto determinado. Un ejemplo de la salida de la simulación para un cuanto de 20 ms, con los tres procesos descritos anteriormente sería:
Tiempo 0: P1 entra a ejecución.
Tiempo 20: P1 se conmuta. Pendiente por ejecutar 80 ms
Tiempo 20: P2 entra a ejecución
Tiempo 35: P2 se conmuta. Pendiente por ejecutar 0 ms
Tiempo 35: P3 entra a ejecución
Tiempo 55: P3 se conmuta. Pendiente por ejecutar 20 ms
Tiempo 55: P1 entra a ejecución
Tiempo 75: P1 se conmuta. Pendiente por ejecutar 60 ms
Tiempo 75: P3 entra a ejecución
Tiempo 95: P3 se conmuta. Pendiente por ejecutar 0 ms.
Tiempo 95: P1 entre en ejecución
Tiempo 155: P1 termina su ejecución

ESTADÍSTICAS GENERADAS:
Total tiempo de ejecución de todos los procesos:155 ms
Total de conmutaciones: 3

La respuesta a este simulador se basa en el uso de la cola. A continuación se da el algoritmo una posible solución.

1. Encolar uno a uno los procesos extraídos a partir de la información del archivo plano.
2. Mientras la cola no esté vacía (No esVacia)
2.1. Leer el siguiente proceso (frente).
2.1.1. Si el proceso termina su ejecución antes de finalizar el cuanto se desencola (desencolar).
2.1.2. Si finaliza el cuanto y el proceso no termina su ejecucion, se lo manda la final de la cola (encolar)
2.2. Actualizar estadísticas
3. Se imprimen las estadísticas
4. FIN

9.6 Ejercicios Colas

- Se debe utilizar una cola para simular el despegue de aviones del aeropuerto el Dorado de Bogotá (proponer un algoritmo). Llas políticas que rigen las operaciones aéreas son las siguientes:

Se tiene estimado que del aeropuerto pueden salir vuelos cada CINCO minutos, siempre y cuando no hayan solicitudes de aterrizaje.
Un vuelo no puede salir del aeropuerto, antes de la hora programada.
Un vuelo no puede despegar del aeropuerto si se encuentran vuelos pendientes por despegar (se debe respetar la cola de vuelos).
El aeropuerto conoce con anterioridad la hora de salida de todos los vuelos programados para ese día. Las aerolíneas tienen programados vuelos con 10 minutos de diferencia.
Los aterrizajes ocurren de manera aleatoria. El aterrizaje tarda DIEZ minutos (es prioritario el aterrizaje de un avión).
Existe un archivo (vuelos.txt) con la programación ordenada por hora de salida de los vuelos de un día cualquiera. El formato debe ser así (numero de vuelo, aerolinea, horas, minutos)
Se asume que la simulación inicia con mínimo 10 solicitudes de despegue.

- Una cola de prioridades es una estructura de datos en la que los elementos se atienden en el orden indicado por una prioridad asociada a cada uno de dichos elementos. Si varios elementos tienen la misma prioridad, se atenderán de modo convencional según la posición que ocupen. Este tipo especial de colas tienen las mismas operaciones que las colas, pero con la condición de que los elementos se atienden en orden de prioridad. Ejemplos de la vida diaria serían la sala de urgencias de un hospital, ya que los enfermos se van atendiendo en función de la gravedad de su enfermedad. La prioridad es un valor numérico, donde 10 es la mayor prioridad y 1 la menor prioridad. De esta forma, cada que se agrega un elemento (encolar), la cola se mantiene ordenada por orden de prioridad. a) Proponga una especificación semiformal del TAD ColaPrioridad, b) Elabore el diagrama de clases que implemente el TAD ColaPrioridad mediante nodos enlazados c) Proponga una implementación en un lenguaje de programación del TAD ColaPrioridad.

- Una bicola es una estructura lineal en la cual los elementos pueden ser adicionados, consultados y eliminados por cualquiera de sus dos extremos. También se les llama DEQUE (Double Ended QUEue). A continuación se muestra el TAD de esta estructura de datos.

TAD BiCola [T]
{ invariante: TRUE }
Constructoras:
    inicBiCola()
Modificadoras:
    adicBiColaFrente()
    adicBiColaFinal()
    elimBiColaFrente()
    elimBiColaFinal()
Analizadoras:
    infoBiColaFrente()
    infoBiColaFinal()
    vaciaBiCola()
Destructora:
destruirBiCola()

---

inicBiCola()
/* Crea una Bicola vacía */
{ post: inicBiCola = }

---

void adicBiColaFrente(T elem)
/* Agrega el elemento elem al frente de la cola */
{ post: bicola = elem,x1,x2,. . . ,xn }

---

void adicBiColaFinal(T elem)
/* Agrega el elemento elem al final de la cola */
{ post: bicola = x1,x2,. . . ,xn, elem }

---

void elimBiColaFrente()
/* Elimina el elemento que está al frente de la bicola */
{ pre: n $>$ 0 }
{ post: bicola = x2,. . . ,xn }

---

void elimBiColaFinal()
/* Elimina el elemento que está al final de la bicola */
{ pre: n $>$ 0 }
{ post: bicol = x1,x2,. . . ,xn-1 }

T infoBiColaFrente()
/* Retorna el elemento que está al frente de la bicola */
{ pre: n $>$ 0 }
{ post: infoBiColaFrente = x1 }

T infoBiColaFinal()
/* Retorna el elemento que está al final de la bicola */
{ pre: n $>$ 0 }
{ post: infoBiColaFinal = xn }

---

bool vaciaBiCola()
/* Informa si la bicola está vacía */
{ post: vaciaBiCola = (bicol =) }

---

void destruirBiCola()
/* Destruye la bicola retornando toda la memoria ocupada */
{ post: la bicola ha sido destruida }

a) Elabore el diagrama de clases que implemente el TAD BiCola mediante nodos enlazados b) Proponga una implementación en un lenguaje de programación del TAD BiCola.

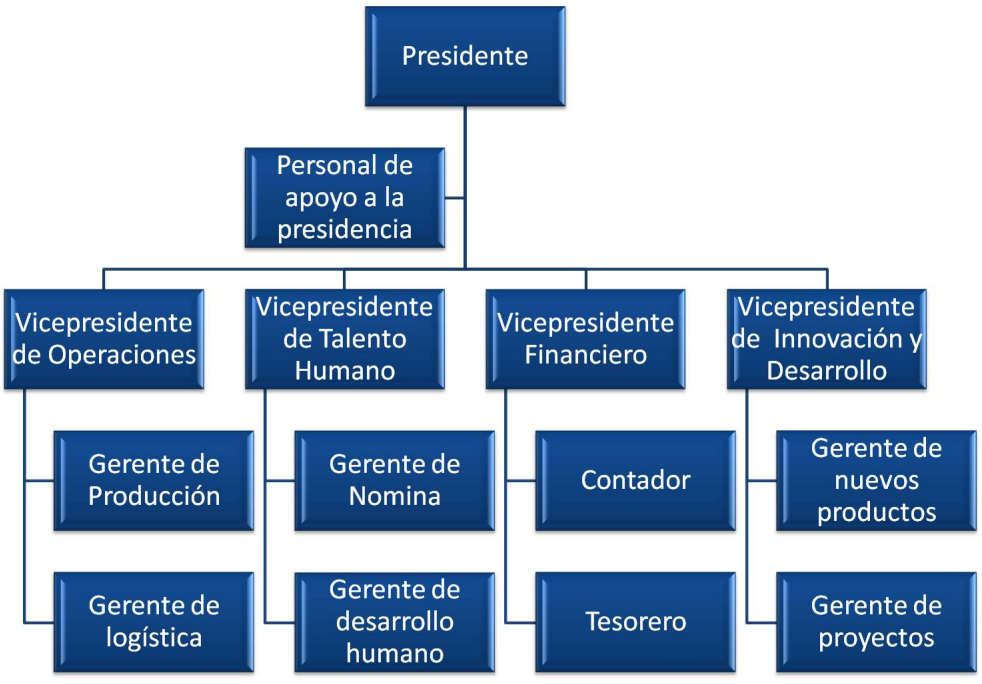
10 — Estructuras de Datos No Lineales. Arboles Binarios

Arboles.

Muchas veces los seres humanos nos inspiramos en la naturaleza para resolver nuestros problemas. En el caso de las estructuras de datos, cuando debemos representar jerarquías o taxonomías y otro tipo de problemas un poco más complejos, nos es fácil pensar en ellas como estructuras arbóreas invertidas en las cuales el elemento de mayor jerarquía se considera como la raíz, los elementos de menor jerarquía ocupan los niveles inferiores dentro de un orden estipulado permitiendo un manejo de información adecuado del problema que deseamos modelar.

Así por ejemplo los organigramas típicos de una organización se representan de forma natural cómo árboles en los cuales el presidente ocupa la raíz, los vicepresidentes el siguiente nivel del árbol y así sucesivamente hasta completar la planta de personal de una organización.

Figura 10.1: Ejemplo de Información representada con árboles.



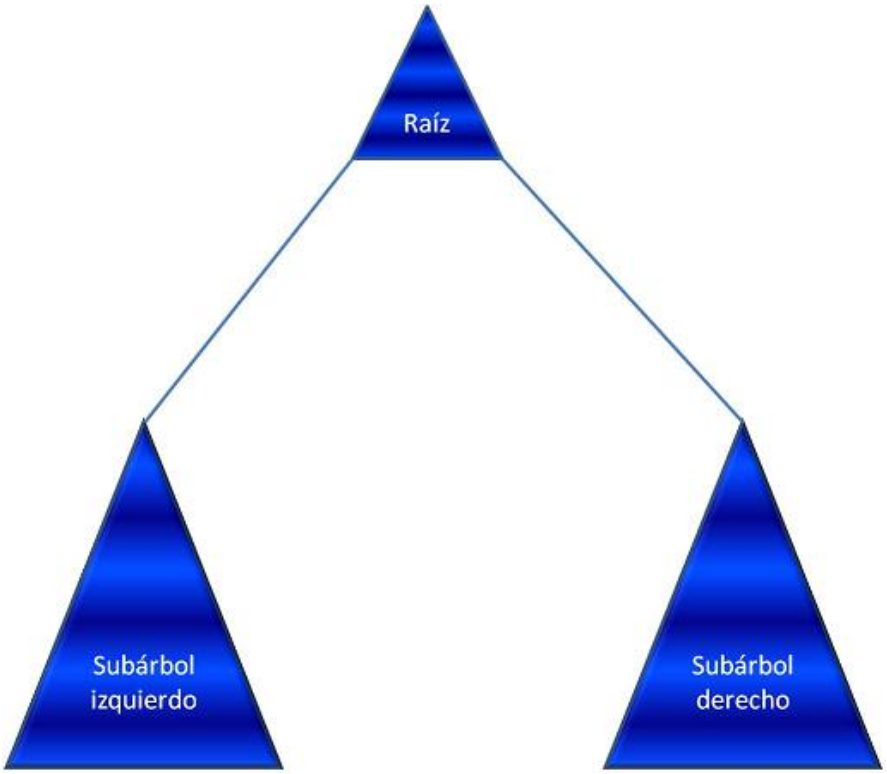
La gran ventaja del almacenamiento de información en árboles es que muchos de los problemas que se resuelven mediante esta representación reducen su complejidad en razón de la misma representación y no de los algoritmos que se utilicen. Tal es el caso de la búsqueda de elementos en un árbol ordenado, como se detallará más adelante.

Arboles binarios

Los árboles binarios son estructuras de datos no lineales, también denominadas recursivas en razón de que su construcción puede hacerse en la aplicación de estructuras básicas aplicadas sobre sí mismas, tal y como ocurre con el caso de los fractales. En este caso un árbol binario es una estructura recursiva que consta de un elemento que denominaremos raíz y dos elementos adicionales que pueden ser vacios a los que denominaremos el subárbol izquierdo y el subárbol derecho, los cuales a su vez son árboles con su respectiva raíz, subárbol izquierdo y subárbol derecho.

Al igual que en otras estructuras la información que se almacena en todos y cada uno de los árboles, debe ser del mismo tipo para garantizar la integridad y coherencia de la información almacenada. En este sentido decimos que los árboles almacenan elementos de un mismo conjunto y que dichos elementos pueden mantener o no una relación de orden entre ellos dependiendo de la necesidad del problema a resolver.

Para representar gráficamente un árbol podemos hacer uso de representaciones como las siguientes en las cuales el nivel de abstracción marca la diferencia. Todas estas representaciones son válidas y depende de la necesidad específica, su uso.



10.1 Casos de estudio del uso de Árboles Binarios

Como caso de estudio de estudiarán los Arboles de Búsqueda Equilibrados

10.1.1 Eficiencia en la búsqueda de un árbol equilibrado

La eficiencia de una búsqueda en un árbol binario ordenado varía entre O(n) y O(Log(n)), dependiendo de la estructura que presente el árbol. La estructura del árbol depende del orden en que sean añadidos los datos. Así, si todos los elementos se insertan en orden creciente o decreciente, el árbol tendrá todas las ramas izquierdas o derechas, respectivamente, vacías. En este caso, la búsqueda en dicho árbol será totalmente secuencial.

Sin embargo, si el árbol está equilibrado, las comparaciones para buscar un elemento serán máximo Log2(n).

Para optimizar los tiempos de búsqueda en los árboles ordenados surgen los árboles casi equilibrados, en la que la complejidad de la búsqueda es logarítmica, O(Log(n)).

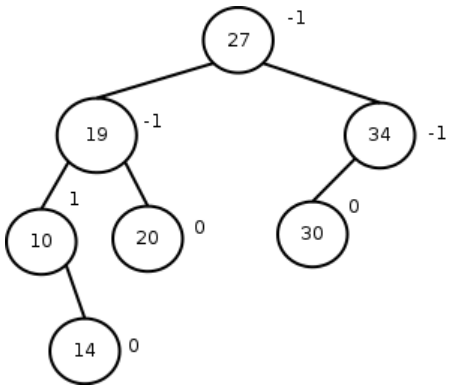
10.1.2 Árbol binario equilibrado, árboles AVL

Un árbol totalmente equilibrado se caracteriza porque la altura de la rama izquierda es igual que la altura de la rama derecha para cada uno de los nodos del árbol. Es un árbol ideal, pero no siempre es posible conseguir que el árbol esté totalmente balanceado.

La estructura de datos de árbol que se utiliza es la del árbol AVL. El nombre es en honor a Adelson- Velskii-Landis, que fueron los primeros científicos en estudiar las propiedades de esta estructura de datos.

Un árbol equilibrado o AVL es un árbol binario de búsqueda en las que las alturas de los árboles izquierdo y derecho de cualquier nodo difieren como máximo en 1.

A cada nodo se asocia el parámetro denominado **factor de equilibrio** o balance de nodo, el cual se define como la altura del subárbol derecho menos la altura del subárbol izquierdo. El factor de equilibrio de cada nodo de un árbol equilibrado puede tomar valores: 1, -1 ó 0. La siguiente figura muestra un árbol balanceado con el factor de equilibrio de cada nodo.



Nota: La altura o profundidad de un árbol binario es el nivel máximo de sus hojas más uno. La altura de un árbol nulo se considera cero.

Inserción en árboles AVL

Para añadir un elemento se sigue el mismo algoritmo de los árboles de búsqueda. Sin embargo, una vez agregado o borrado un elemento puede destruir la condición de equilibrio de varios nodos del árbol. Por ello, se requiere que el algoritmo recupere la condición de equilibrio. Este hecho hace necesario que el algoritmo regrese por el camino de búsqueda actualizando el factor de equilibrio de los nodos.

La estructura del nodo de un árbol AVL exige un nuevo campo: el factor de equilibrio (fe). Se puede tener algo como:

```
public class NodoAvl extends Nodo {

        private int fe; //factor equilibrio

        public NodoAvl(Object valor) {
                super(valor);
                fe=0;
        }
        //..
}
```

Las operaciones básicas de un árbol AVL son insertar y borrar un elemento; además, se necesitan operaciones auxiliares para mantener los criterios de equilibrio:

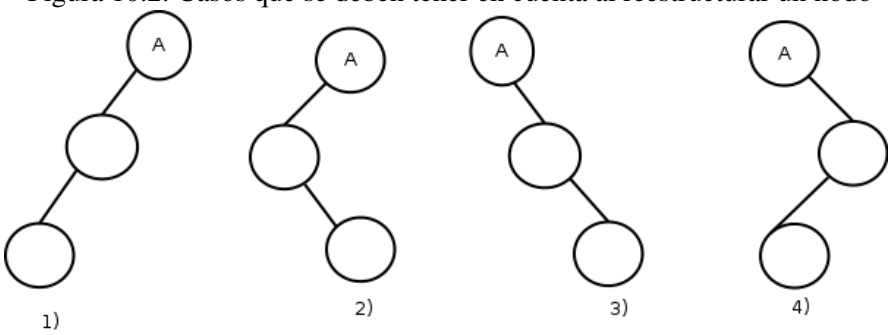
```
public class ArbolAVL {
        private Nodo raiz;

        public ArbolAVL() {
                raiz=null;
        }
        //..
}
```

Proceso de inserción de un nuevo nodo

Inicialmente se aplica el algoritmo de inserción en un árbol de búsqueda; este algoritmo sigue el camino de búsqueda hasta llegar al fondo del árbol y se enlaza como nodo hoja y con el factor de equilibrio cero. Luego, es necesario recorrer el camino de búsqueda en sentido contrario, hasta la raíz, para actualizar el campo adicional factor de equilibrio. Después de la inserción solo los nodos que se encuentran en el camino de búsqueda pueden haber cambiado el factor de equilibrio. Al actualizar el nodo cuyas ramas izquierda y derecha del árbol tienen altura diferente, si se inserta el nodo en la rama más alta rompe el criterio de equilibrio del árbol, la diferencia de altura pasa a ser 2 y es necesario reestructurarlo. Hay 4 casos que se deben tener en cuenta al reestructurar un nodo A, según donde se haya hecho la inserción (ver Figura 10.2):

1. Inserción en el subárbol izquierdo de la rama izquierda de A (rotación izquierda-izquierda).
2. Inserción en el subárbol derecho de la rama izquierda de A (rotación izquierda-derecha)..
3. Inserción en el subárbol derecho de la rama derecha de A (rotación derecha-derecha).
4. Inserción en el subárbol izquierdo de la rama derecha de A (rotación derecha-izquierda).

Figura 10.2: Casos que se deben tener en cuenta al reestructurar un nodo

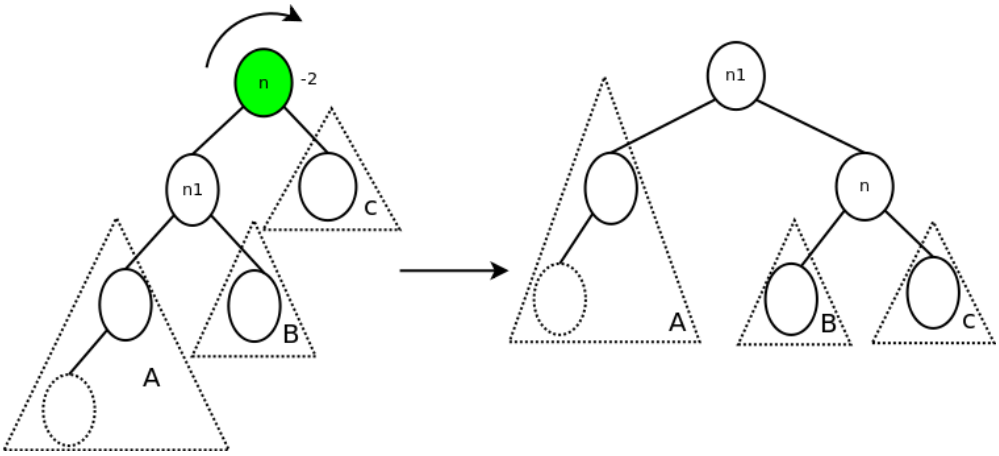


El caso 1) y caso 3) se resuelven con una rotación simple. El caso 2) y 4) se resuelven con una rotación doble.

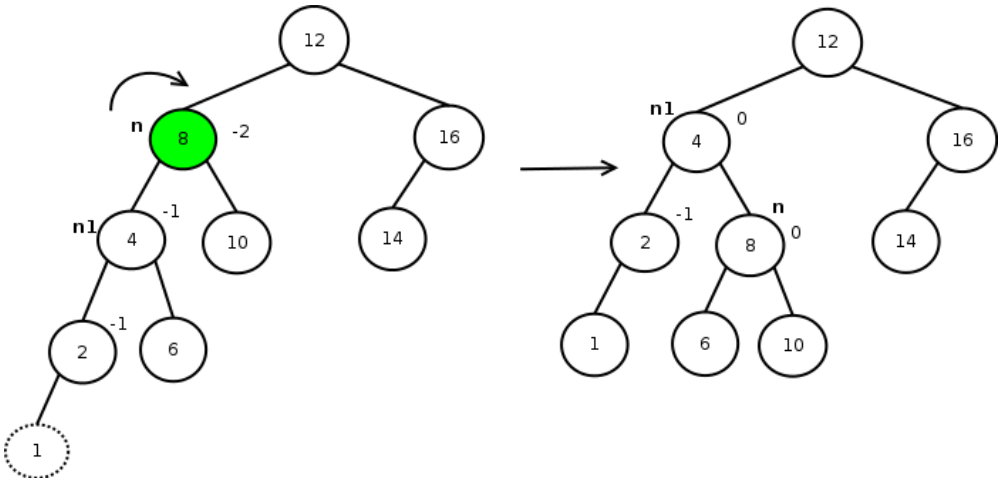
Rotación simple

Caso 1: rotación simple izquierda-izquierda

En la siguiente Figura se muestra la rotación simple que resuelve el caso 1 (rotación izquierda-izquierda). El gráfico izquierdo muestra el árbol antes del ajuste; a la derecha se muestra el árbol después de dicho ajuste. El nodo dibujando en lineas punteadas, corresponde al nodo que se inserta y provoca que el nodo n (pintado de verde) rompa el factor de equilibrio.



La siguiente Figura , muestra un ejemplo donde se aplica la inserción simple izquierda-izquierda, la cual corresponde a la rotación hecha en la Figura anterior.



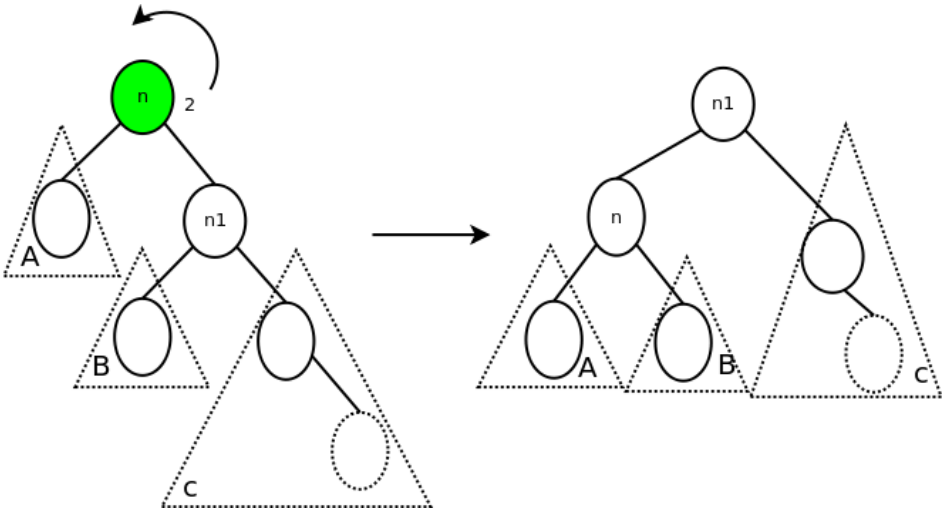
Los cambios descritos en la rotación simple afectan a dos nodos. Los ajustes necesarios de los enlaces, suponiendo que n sea la referencia al nodo problema, y n1 la referencia al nodo de su rama izquierda son:

n.izquierdo = n1.derecho
n1.derecho = n
n = n1

Una vez realizada la rotación, los factores de equilibrio de los nodos que intervienen siempre son 0, los subárboles izquierdo y derecho tienen la misma altura. Incluso la altura del subárbol implicado es la misma después de la inserción que antes.

Caso 3: rotación simple derecha-derecha

La siguiente Figura ilustra la rotación simple derecha-derecha (caso 3).



En la rotación simple derecha-derecha, los cambios en los enlaces del nodo n (con factor de equilibrio +2) y del nodo de su rama derecha, n1, son:

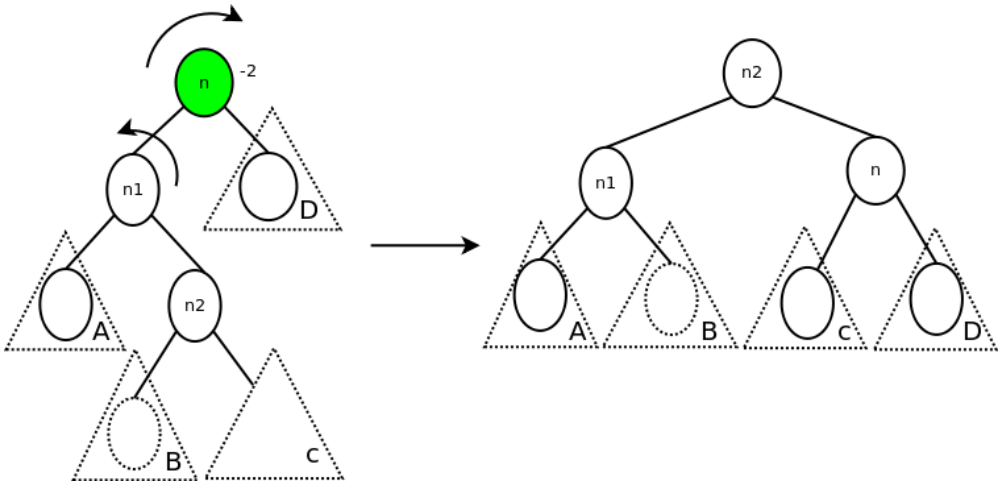
n.derecho = n1.izquierdo
n1.izquierdo = n
n = n1

Rotación doble

En los casos 2) y 4), una rotación simple no reduce lo suficiente su profundidad, se necesita una rotación doble.

Rotación doble izquierda-derecha

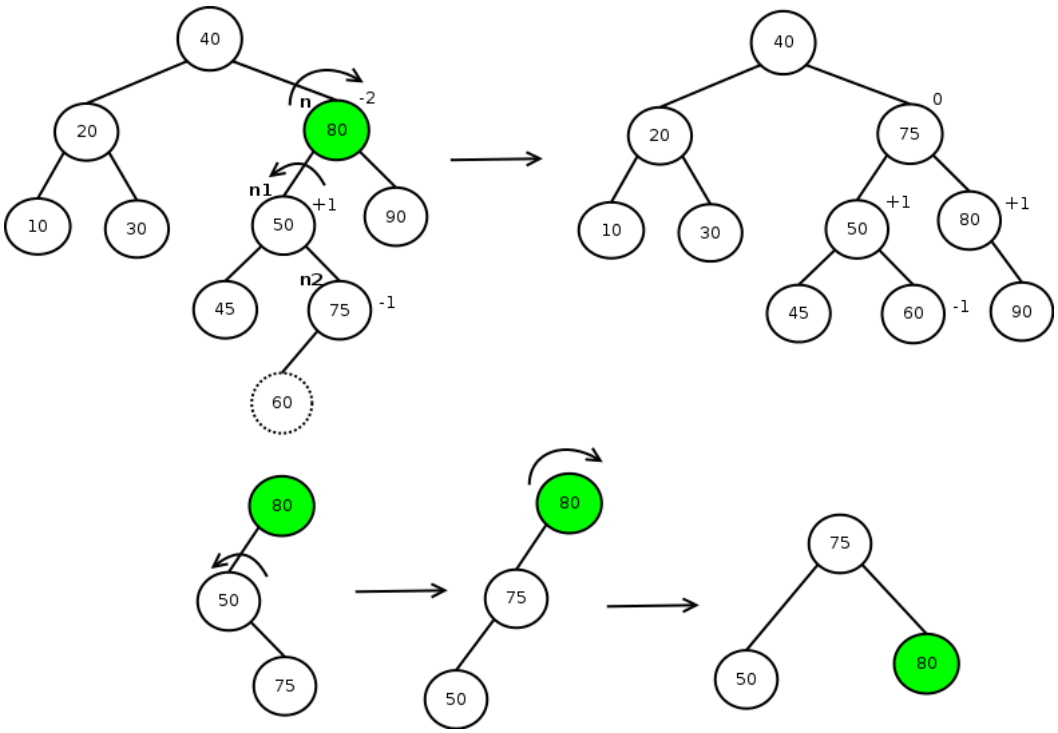
La sigiente Figura ilustra la rotación doble izquierda-derecha (caso 2). Los cambios de la rotación doble afectan a tres nodos: el nodo problema n, el descendiente por la rama desequilibrada n1, y el descendiente de n1 (por la izquierda o derecha, según el tipo de rotación doble) apuntado por n2. En los dos casos simétricos de rotación doble, rotación izquierda-derecha y rotación derecha-izquierda, el nodo n2 pasa a ser la raíz del nuevo subárbol.



Ejemplo:

La siguiente Figura muestra un árbol binario de búsqueda después de insertar la clave 60 (caso 4). Al volver por el camino de búsqueda para actualizar los factores de equilibrio, el nodo

75 pasa a tener factor de equilibrio -1, el nodo 50 +1 y el nodo 80 tendrá -2. En este caso se restablece el equilibrio con una rotación doble. La gráfica inferior toma una sección del árbol para mostrar con claridad las dos rotaciones.



En el caso de la rotación izquierda-derecha, los movimientos de enlace para realizar la rotación son:

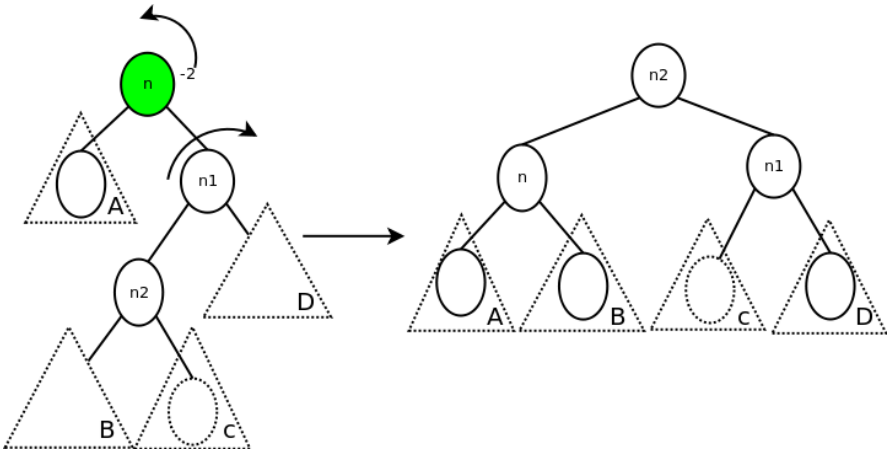
n1.derecho = n2.izquierdo
n2.izquierdo = n1
n.izquierdo = n2.derecho
n2.derecho = n
n = n2

Los factores de equilibrio de los nodos implicados en la rotación izquierda-derecha dependen del factor de equilibrio, antes de la inserción, del nodo apuntado por n2, según esta tabla:

Si	n2.fe = -1	n2.fe = 0	n2.fe = 1
	n.fe = 1	0	0
	n1.fe = 0	0	-1
	n2.fe = 0	0	0

Rotación doble derecha-izquierda

La siguiente Figura, muestra la rotación doble derecha-izquierda (caso 4).



Los movimientos de los enlaces para realizar la rotación derecha-izquierda (observar la simetría de los movimientos) son:

n1.izquierdo = n2.derecho
n2.derecho = n1
n.derecho = n2.izquierdo
n2.izquierdo = n
n = n2

Los factores de equilibrio de los nodos implicados en la rotación derecha-izquierda también dependen del factor de equilibrio del nodo n2, según la tabla:

Si	n2.fe = -1	n2.fe = 0	n2.fe = 1
	n.fe = 0	0	-1
	n1.fe = 1	0	0
	n2.fe = 0	0	0

La complejidad de inserción de una clave en un árbol de búsqueda AVL es O(Log n).

Borrado de un nodo en un árbol equilibrado

Esta operación elimina un nodo, con cierta clave, de un árbol de búsqueda equilibrado; el árbol resultante debe seguir siendo equilibrado.

En el algoritmo de borrado lo primero que se hace es buscar el nodo con la clave a eliminar, para ellos se sigue el camino de búsqueda. A continuación, se procede a eliminar el nodo. Se distinguen dos casos:

1. El nodo a eliminar es un nodo **hoja**, ó con un un **único descendiente**. Entonces, simplemente se suprime, o bien se sustituye por su descendiente.
2. El nodo a eliminar tiene **dos subárboles**. Se procede a buscar el nodo más a la derecha del subárbol izquierdo, es decir, el de mayor clave en el subárbol de claves menores; éste se copia en el nodo a eliminar y, por último se retira el nodo copiado.

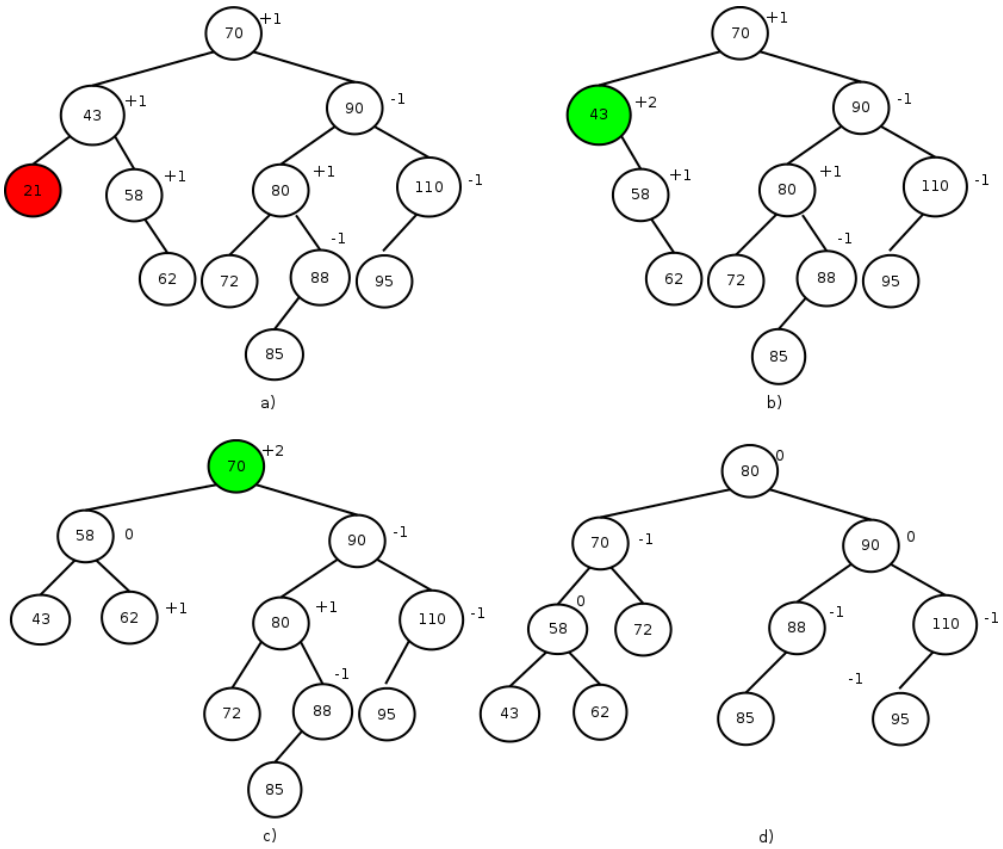
Una vez eliminado el nodo, el algoritmo tiene que prever la actualización de los factores de equilibrio de los nodos que han formado el **camino de búsqueda**, ya que la altura de alguna de las dos ramas ha disminuido. Por consiguiente se regresa por los nodos del camino, hacia la raíz, calculando el factor de equilibrio. No siempre es necesario recorrer todo el camino de regreso.

Cuando el factor de equilibrio de algún nodo es +2 o -2, se viola la condición de equilibrio y se debe restaurar la condición de equilibrio mediante las rotaciones simple o doble. El tipo específico de rotación depende del fe del nodo problema, apuntado por n, y del nodo descendiente n1:

- Si n.fe == +2, entonces n1 es su hijo derecho, de tal forma que si n1.fe >= 0 la rotación a aplicar es derecha-derecha. Y si n1.fe == -1, la rotación a aplicar es derecha-izquierda.
- De forma simétrica, si n.fe == -2, entonces n1 es su hijo izquierdo, de tal forma que si n1.fe <= 0, la rotación a aplicar es izquierda-izquierda. Y si n1.fe == +1, la rotación a aplicar es izquierda-derecha.

En el proceso de eliminar una clave una vez que se aplica una rotación, la altura del subárbol puede disminuir, por ello el proceso de actualización del fe debe continuar, ya que otro nodo del camino hacia la raíz puede que viole la condición de equilibrio y sea necesario aplicar otra rotación.

La siguiente figura, en su parte b) muestra el árbol de búsqueda después de haber eliminado la clave 21; con la actualización del factor de equilibrio, el nodo 43 está desequilibrado, hay que aplicar una rotación **derecha-derecha**. La figura c) muestra el árbol después de la rotación y la posterior actualización del factor de equilibrio del nodo 70, que también exige aplicar otra rotación, en este caso, derecha-izquierda. La parte d) de la figura muestra el árbol equilibrado después de la última rotación y finalización del algoritmo.



El algoritmo de borrado de una clave en un árbol de búsqueda AVL puede necesitar más de una rotación para que el árbol resultante siga siendo equilibrado. La complejidad de toda la operación es logarítmica.

10.1.3 Tarea

Implementar el TAD del árbol AVL, el cual tiene las siguientes operaciones:

- Inserción de un nodo. Crea un nodo con su dato asociado y lo añade, en orden al árbol. Si el árbol se des-balancea se debe recuperar la condición de equilibrio.
- Búsqueda de un nodo. Devuelve la referencia al nodo del árbol o null.
- Borrado de un nodo. Busca el nodo del árbol que contiene el dato y lo quita. El árbol debe seguir siendo de búsqueda y debe recuperar su condición de equilibrio.
- Recorrido de un árbol. Los mismos recorridos de un árbol binario: anchura, preorden, inorden y postorden.
- Imprimir árbol: imprime por consola la estructura del árbol

El programa debe tener un sencillo menú de opciones que permita probar ágilmente la correcta funcionalidad de todas las operaciones.

10.2 Conceptos de árboles

10.2.1 Introducción

El árbol es una estructura de datos muy importante en informática y en ciencias de la computación. Los árboles son estructuras no lineales, al contrario de las listas y los arreglos que son estructuras lineales.

Los árboles se utilizan para representar fórmulas algebraicas, para organizar objetos en orden de tal forma que las búsquedas sean muy eficientes y en aplicaciones diversas tales como la inteligencia artificial o algoritmos de cifrado. Casi todos los sistemas operativos almacenan sus archivos en árboles o estructuras similares a árboles. También los árboles se utilizan en el diseño de compiladores, procesado de texto y algoritmos de búsqueda

10.2.2 Definición

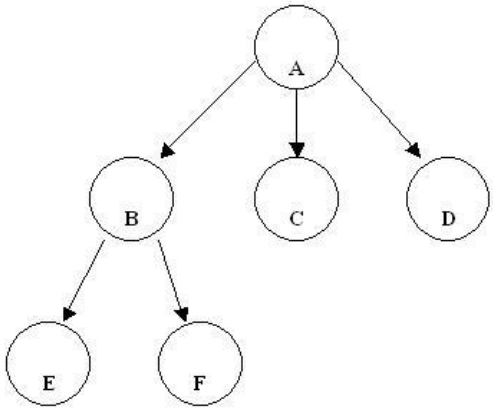
En la vida real, hay muchos ejemplo de árboles: árbol genealógico, la estructura de una empresa también es un árbol.

Definición 1: Un árbol consta de un conjunto finito de elementos llamados nodos y un conjunto finito de lineas dirigidas, llamadas ramas, que conectan los nodos.

Definición 2: Un árbol es un conjunto de uno o más nodos tales que:

1. Hay un nodo diseñado especialmente llamado raíz.
2. Los nodos restantes se dividen en n>=0 conjuntos disjuntos, T1 ... Tn, tales que cada uno de esos conjuntos es un árbol. A T1 ... Tn se les denomina subárboles del raíz.

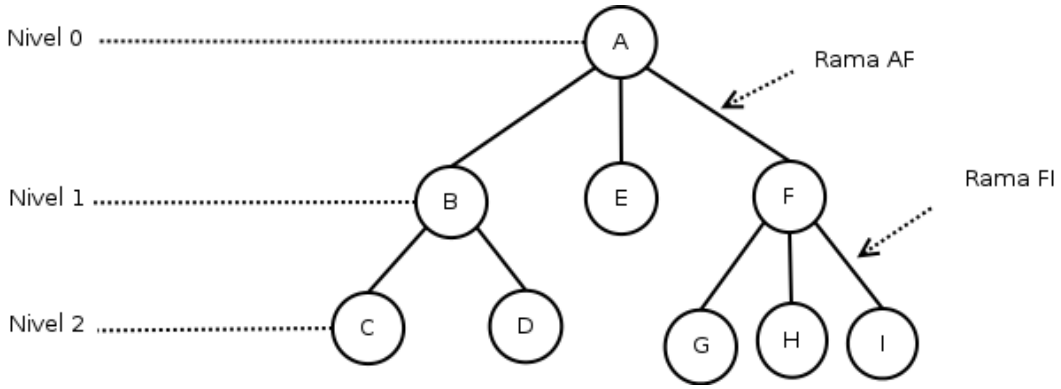
Si un árbol no está vacío, entonces el primer nodo se llama raíz. Obsérvese en la definición 2, que el árbol ha sido definido de modo recursivo, ya que los árboles se definen como árboles. La siguiente gráfica, muestra un ejemplo gráfico de una estructura tipo árbol.



10.2.3 Terminología

Además del nodo raíz, existen muchos términos utilizados en la descripción de los atributos de un árbol, los cuales se explican a continuación.

Nodo padre: Un nodo es padre si tiene sucesores.
Nodo hijo: Son los sucesores de un nodo padre.
Nodos descendientes: Los hijos de un nodo y los hijos de éste se llaman descendientes.
Nodos ascendientes: El padre y los y los abuelos de un nodos son los ascendientes.
Nodos hermanos: Dos o más nodos con el mismo padre se llaman hermanos
Nodo hoja: Es un nodo sin descendientes (Nodo terminal). Ej. Nodos E – F – C y D.
Nodo interior: Es un nodo que no es hoja. Ej. Nodos A y B.
Nivel de un nodo: Es la distancia al nodo raíz. La raíz tiene una distancia de cero de sí misma, por eso se dice que está en el nivel cero. Los hijos del nodo raíz están en el nivel 1, sus hijos están en el nivel 2, y así sucesivamente. Los hermanos están siempre en el mismo nivel, pero no todos los nodos de un mismo nivel son necesariamente hermanos. La siguiente gráfica ilustra estos conceptos.



- **Padres**: A, B, F
- **Hijos**: B, E, F, C, D, G, H, I
- **Hermanos**: {B, E, F}, {C, D}, {G, H, I}
- **Hojas**: C, D, E, G, H, I

Camino: es una secuencia de nodos en los que cada nodo es adyacente al siguiente. Cada nodo del árbol puede ser alcanzado (se llega a él) siguiendo un único camino que comienza en el nodo raíz. En la figura 2, el camino desde la raíz a la hoja I, se representa por AFI, incluye dos ramas

distintas AF y FI.

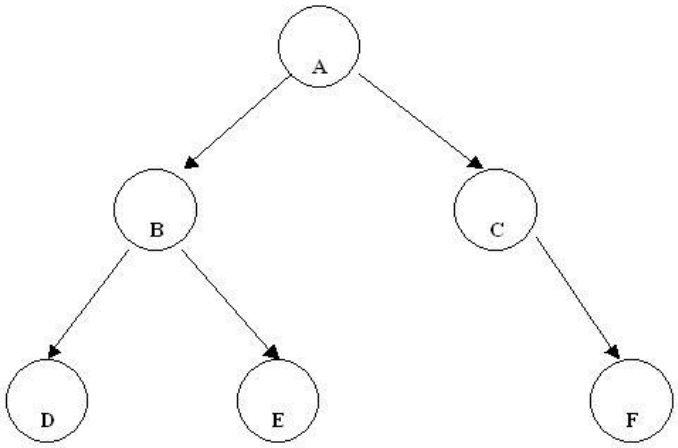
Altura o profundidad: La altura o profundidad de un árbol es el nivel de la hoja del camino más largo desde la raíz más uno. Por definición la altura de un árbol vacío es cero. La figura 2, contiene nodos en tres niveles: 0, 1 y 2. su altura es 3.

Subárbol: un subárbol de un árbol es un conjunto de nodos del árbol, conectados por ramas del propio árbol, esto es a su vez un árbol.

Definición Recursiva

Un árbol se divide en subárboles. Un subárbol es cualquier estructura conectada por debajo del nodo raíz. Cada nodo de un árbol es la raíz de un subárbol que se define por el nodo y todos sus descendientes. El primer nodo de un subárbol se conoce como el nodo raíz del subárbol y se utiliza para nombrar el subárbol. Además los subárboles se pueden subdividir en subárboles. En la figura 2, BCD es un subárbol al igual que E y FGHI. Un nodo simple también es un subárbol. Por consiguiente, el subárbol B, se pueden dividir en subárboles C y D, mientras que el subárbol F contiene los subárboles G, H e I. Se dice que G, H, I, C y D son subárboles sin descendientes. *El concepto de subárbol conduce a una definición recursiva de un árbol. Un subárbol es un conjunto de nodos que:*
1. Es vacío
2. O tiene un nodo determinado, llamado raíz, del que jerárquicamente descienden cero o más subárboles, que son también árboles.

Árbol binario: Un árbol es binario si cada nodo tiene como máximo 2 descendientes.(Ver la siguiente Gráfica).



Para cada nodo está definido el subárbol izquierdo y el derecho. Para el nodo A el subárbol izquierdo está constituido por los nodos B, D y E. Y el subárbol derecho está formado por los nodos C y F. Lo mismo para el nodo B tiene el subárbol izquierdo con un nodo (D) y un nodo en el subárbol derecho (E). El nodo D tiene ambos subárboles vacíos. El nodo C tiene el subárbol izquierdo vacío y el subárbol derecho con un nodo (F).

10.2.4 Ejercicio.

Dibujar su propio árbol (cualquiera), e identificar cada uno de los términos estudiados (Nodo padre, hijos, hermanos...)

10.3 Aspectos teóricos. Especificación formal.

10.3.1 TAD ARBOL BINARIO

La estructura de un árbol binario constituye un tipo abstracto de datos; las operaciones básicas que definen el TAD árbol binario son las siguientes:

Tipo de dato:Dato que se almacena en los nodos del árbol

Operaciones

- CrearArbol:Inicia el árbol como vacío
- Construir: Crea un árbol con un elemento raíz dos ramas, izquierda y derecha que son a su vez árboles.
- EsVacio:Comprueba si el árbol no contiene nodos
- Raiz: Devuelve el nodo raíz
- Izquierdo: Obtiene la rama o subárbol izquierdo de un árbol dado
- Derecho: Obtiene la rama o subárbol derecho de un árbol dado
- Borrar: Elimina del árbol el nodo con un elemento determinado
- Pertenece:Determina si un elemento se encuentra en el árbol

Operaciones en árboles binarios

Algunas de las operaciones típicas que se realizan en los árboles binarios son las siguientes:

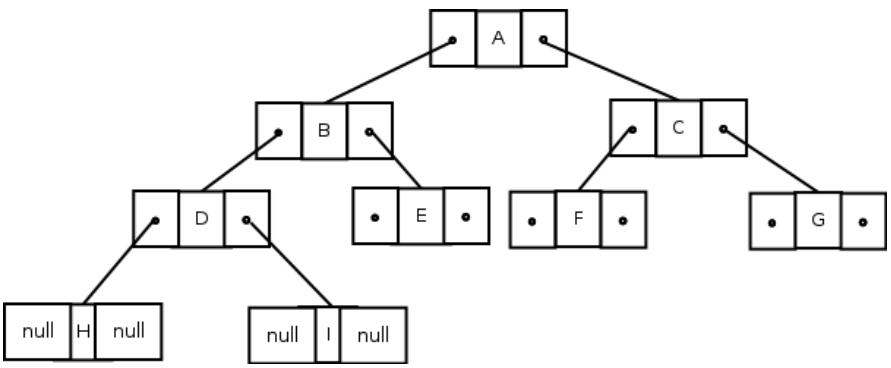
- Determinar su altura.
- Determinar su número de elementos
- Hacer una copia
- Visualizar el árbol binario en pantalla o en impresora
- Determinar si dos árboles binarios son idénticos
- Borrar (eliminar el árbol)
- Si es un árbol de expresión, evaluar la expresión

Todas estas operaciones se realizan recorriendo el árbol de manera sistemática. El recorrido implica que la visita a cada nodo se debe hacer una sola vez.

10.4 Formas de Representación de los Arboles Binarios

10.4.1 Estructura de un árbol binario

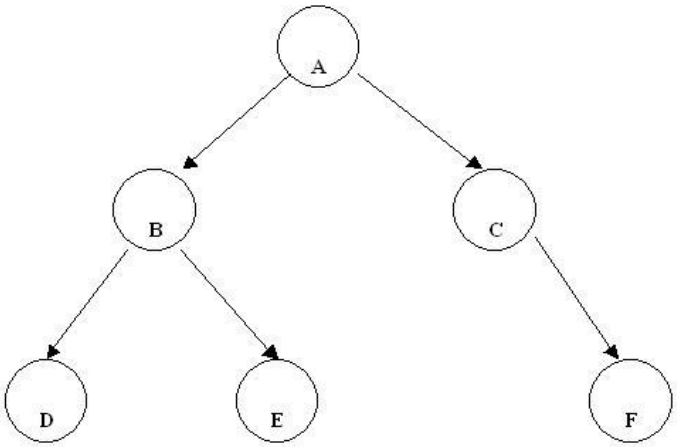
Un árbol binario se construye con nodos. Cada nodo debe contener el campo dato (datos a almacenar) y dos campos de enlace (apuntador o referencia), uno al subárbol izquierdo y el otro al subárbol derecho. La siguiente gráfica ilustra esta estructura.



10.5 Conceptos de árboles binarios

10.5.1 Definición

Un árbol binario es un árbol cuyos nodos no pueden tener más de dos subárboles. En un árbol binario, cada nodo puede tener, cero, uno o dos hijos (subárboles). Se conoce el nodo de la izquierda como hijo izquierdo y el nodo de la derecha como hijo derecho. La siguiente gráfica , muestra un árbol binario.



Un árbol binario es una *estructura recursiva*. Cada nodo es la raíz de su propio subárbol y tiene hijos, que son raíces de árboles, llamados subárbol derecho e izquierdo del nodo, respectivamente.

En cualquier nivel n, un árbol binario puede contener de 1 a 2n nodos. El número de nodos por nivel contribuye a la densidad del árbol.

10.5.2 Equilibrio

La distancia de un nodo a la raíz determina la eficiencia con la que puede ser localizado. Dado un nodo, se lo puede localizar mediante un solo camino de bifurcación de ramas. Esta característica conduce al concepto de **balance** o **equilibrio**. Para determinar si un árbol está equilibrado se calcula su factor de equilibrio.

El **factor de equilibrio** de un árbol binario es la diferencia en altura entre los subárboles derecho e izquierdo. Si la altura del subárbol izquierdo es hI, y la altura del subárbol derecho es hD, entonces el factor de equilibrio del árbol B se determina por la siguiente fórmula:

$B = hD - hI$ (LA D y la I deben ir SUBINDICES)

Utilizando esta fórmula, el equilibrio del nodo raíz de la figura 1 es 0.

Un árbol está **perfectamente equilibrado** si su equilibrio o balance es cero y sus subárboles son también perfectamente equilibrados. Dado que esta condición ocurre raramente se dice que un árbol está **equilibrado** si la altura de sus subárboles difiere en no más de uno y sus subárboles son también equilibrado; por lo tanto, el factor de equilibrio de cada nodo puede tomar los valores -1, 0, +1.

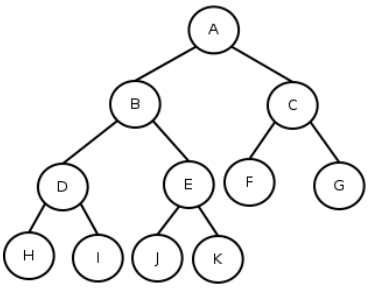
10.5.3 Arboles binario completos

Un árbol binario completo de profundidad n es un árbol en el que cada nivel, del 0 al nivel n-1, tiene un conjunto lleno de nodos, y todos los nodos hoja a nivel n ocupan las posiciones más a la izquierda del árbol.

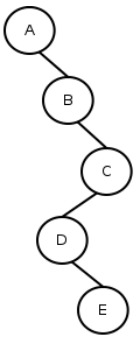
Un árbol binario completo que contiene 2^n nodos a nivel *n* es un **árbol lleno**. Un árbol lleno es un árbol binario que tiene el máximo número de entradas para su altura.

Un **árbol degenerado** es aquel que cada nodo contiene solo un hijo. Un árbol degenerado es equivalente a una lista enlazada.

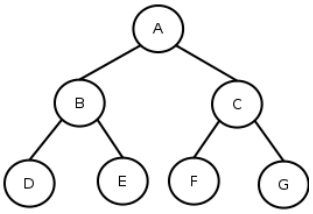
La siguiente gráfica muestra la clasificación de árboles binarios.



a) Árbol completo de profundidad 4



b) Árbol degenerado de profundidad 5



c) Árbol lleno de profunidad 3

La altura o profundidad de un árbol binario completo de n nodos se calcula mediante la siguiente fórmula:

h = |Log2n| + 1 (parte entera de Log2n +1)

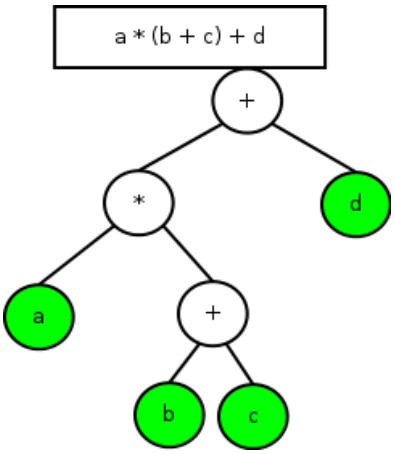
Ejemplo. Suponiendo que se tiene n = 10.000 elementos que van a ser los nodos de un árbol binario completo. Determinar la profundidad del árbol.

h = int(Log2n) + 1 = int (Log210000) + 1 = int(13.28) + 1 = 14

10.5.4 Árbol de expresión

Una aplicación importante de los árboles binarios son los árboles de expresiones. Una expresión es una secuencia de tokens (componentes léxicos que siguen unas reglas establecidas). Un token puede ser un operando o u operador.

La siguientes gráfica muestra la expresión: a * (b + c) + d



10.5.5 Recorrido de un árbol

Para recorrer o consultar los datos de un árbol se necesita recorrer el árbol o visitar los nodos del mismo. Al contrario de las listas enlazadas los árboles no tienen un primer valor, un segundo valor, un tercer valor, etc. Por ello existen diferentes recorridos.

El recorrido de un árbol binario exigen que cada nodo sea visitado una sola vez. Existen dos enfoques:

- En el recorrido en **profundidad**, el proceso exige un camino desde la raíz a través de un hijo, al descendiente más lejano del primer hijo antes de proseguir a un segundo hijo. En otras palabras en el recorrido en profundidad, todos los descendientes de un hijo se procesan antes del siguiente hijo.
- En el recorrido en **anchura**, el proceso se realiza horizontalmente desde la raíz a todos sus hijos; a continuación, a los hijos de sus hijosy así sucesivamente hasta que todos los nodos hayan sido procesados. En el recorrido en anchura, cada nivel se procesa totalmente antes de que comience el siguiente nivel.

Recorrido en preorden

El recorrido preorden (RID) conlleva los siguientes pasos, en los que el nodo raíz va antes que los subárboles:

1. Visitar el nodo raíz (R)
2. Recorrer el subárbol izquierdo (I) en preorden
3. Recorrer el subárbol izquierdo (D) en preorden

Preorden, viene del prefijo latino pre que significa “ir antes” (la raíz se procesa antes que los subárboles izquierdo y derecho).

Dadas las características recursivas de los árboles, el algoritmo de recorrido tiene naturaleza recursiva. Primero se procesa la raíz; a continuación, el subárbol izquierdo y, posteriormente el subárbol derecho. Para procesar el subárbol derecho. Para procesar el subárbol izquierdo, se siguen los mismos pasos: raíz, subárbol izquierdo y subárbol derecho (proceso recursivo). Luego se hace lo mismo con el subárbol derecho.

Recorrido en orden

El recorrido en orden (inorden IRD) procesa primero el subárbol izquierdo, después el raíz, y a continuación el subárbol derecho. El significado de in es que la raíz se procesa entre los subárboles. El método conlleva los siguientes pasos:

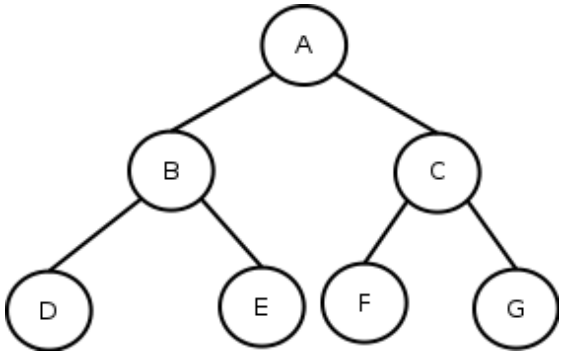
1. Recorrer el subárbol izquierdo (I) en orden
2. Visitar el nodo raíz (R)
3. Recorrer el subárbol izquierdo (D) en orden

Recorrido postorden

El recorrido postorden (IDR) procesa el nodo raiz (post) después de que los subárboles izquierdo y derecho se hayan procesado. Comienza situándose en la hoja más a la izquierda, a continuación se procesa su subárbol derecho. Por último se procesa su nodo raíz. Las etapas del algoritmo son:

1. Recorrer el subárbol izquierdo (I) en postorden
2. Recorrer el subárbol izquierdo (D) en postorden
3. Visitar el nodo raíz (R)

La siguiente gráfica muestra un árbol binario y cómo serían el orden de los elementos en cada uno de los tres recorridos.



Recorrido en preorden: A, B, D, E, C, F, G
Recorrido enorden: D, B,E, A, F, C, G
Recorrido en postorden: D, E, B, F, G, C, A
Recorrido en anchura: A, B, C, D, E, F, G

10.5.6 Implementación de los recorridos

A continuación una implementación recursiva de los tres recorridos en profundidad.

```
void preorden(Nodo aux) {
        if (aux != null) {
                visitar(aux);
                preorden(aux.getIzquierdo());
                preorden(aux.getDerecho());
        }
}
void inorden(Nodo aux) {
        if (aux != null) {
                inorden(aux.getIzquierdo());
                visitar(aux);
                inorden(aux.getDerecho());
        }
}
void postorden(Nodo aux) {
        if (aux != null) {
                postorden(aux.getIzquierdo());
                postorden(aux.getDerecho());
                visitar(aux);
        }
}
```

Nota: el método visitar, depende de la aplicación que se esté realizando. Si simplemente se quieren listar los nodos por consola, podría tener la siguiente implementación:

```
void visitar(Nodo aux) {
        System.out.print(aux.getValor() + "␣");
```

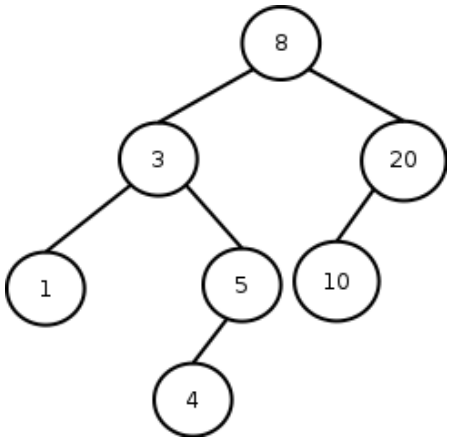
10.6 Implementaciones de los Arboles Binarios y Algoritmos fundamentales

Definición

Un **árbol binario de búsqueda** (o un árbol binario ordenado) es aquel en que, dado un nodo, todos los datos del subárbol izquierdo son menores que los datos de ese nodo, mientras que todos los datos del subárbol derecho son mayores que ese nodo. Se denomina árbol binario de

búsqueda porque se puede buscar en ellos un término utilizando un algoritmo de búsqueda binaria.

Ejemplo: Dibujar un árbol binario de búsqueda con los valores: 8, 3, 1, 20, 5, 4.La respuesta se muestra en la siguiente gráfica.



Implementación orientada a objetos en JAVA

Cada nuevo elemento se debe ir insertando como hoja del árbol, bajando por la rama izquierda o por la rama derecha.

10.6.1 Nodo de un árbol binario de búsqueda

Un nodo de un árbol binario de búsqueda es muy similar a un árbol binario, tiene un campo de datos y dos enlaces a los subárboles izquierdo y derecho respectivamente. Al ser un árbol ordenado, los datos deben implementar la interfaz Comparable:

```
public interface Comparable {
        public boolean esMenor(Object q);
        public boolean esMayor(Object q);
        public boolean esIgual(Object q);

}
```

Hay que tener en cuenta que el campo datos o valor de un nodo, podría ser un objeto de cualquier tipo. Por ejemplo, se podría tener un árbol binario de búsqueda relativa a estudiantes. Cada nodo contiene el nombre del estudiante y su código (dato entero), que se puede utilizar para ordenar. En este caso la clase Estudiante debe implementar la interfaz Comparable de la siguiente manera:

```
public class Estudiante implements Comparable {
        private int codigo;
        private String nombre;


        public boolean esMenor(Object q) {
                Estudiante obj = (Estudiante)q;
                return (codigo < obj.getCodigo());

        }
        //..
}
```

Ejercicio. Suponiendo que el árbol binario de búsqueda almacenar valores enteros, ¿como sería en java la clase Entero?

10.6.2 Operaciones en árboles binarios de búsqueda

Un árbol binario de búsqueda también tiene naturaleza recursiva (aunque sus operaciones también pueden ser implementadas de manera iterativa). Estas operaciones son:

- **Búsqueda de un nodo**. Devuelve la referencia al nodo del árbol o null.
- **Inserción de un nodo**. Crea un nodo con su dato asociado y lo añade, en orden al árbol.
- **Borrado de un nodo**. Busca el nodo del árbol que contiene el dato y lo quita. El árbol debe seguir siendo de búsqueda.
- **Recorrido de un árbol**. Los mismos recorridos de un árbol: preorden, inorden y postorden.

La clase ArbolBinarioBusqueda implementa estas operaciones. Podría se planteado como una extensión (herencia) de ArbolBinario:

```
public class ArbolBinarioBusqueda extends ArbolBinario{

        public ArbolBinarioBusqueda() {
                raiz = null;
        }
        //...
}
```

10.6.3 Insertar un nodo

Para añadir un nodo al árbol, se sigue el camino de búsqueda y, al final del camino, se enlaza el nuevo nodo; por consiguiente, siempre se inserta como hoja del árbol. El árbol que resulta después de insertar el nodo, sigue siendo de búsqueda.

El **algoritmo** se inserción se apoya en la búsqueda de un elemento, de modo que si se encuentra el elemento buscado, no es necesario hacer nada; en caso contrario, se inserta el nuevo elemento justo en el lugar donde se ha acabado la búsqueda (es decir, en el lugar donde habría estado en caso de existir).

Implementación iterativa de la inserción

A continuación la implementación iterativa en java de la operación de inserción.

```
/**
* Inserta un nodo en el  rbol  binario
*
* @param valor
*             valor a insertar
*/
public void insertar(Object valor) throws Exception {
        Comparable dato = (Comparable) valor;
        Nodo nuevo = new Nodo();
        nuevo.setValor(dato);

        if (raiz == null)
                raiz = nuevo;
        else {
                // anterior: referencia al padre de aux
                Nodo anterior = null;
                // aux: auxiliar que va recorriendo los nodos, desde la raiz
                Nodo aux = raiz;
                while (aux != null) {
```

```
                        anterior = aux;
                        if (dato.esMenor(aux.getValor()))
                                aux = aux.getIzquierdo();
                        else if (dato.esMayor(aux.getValor()))
                                aux = aux.getDerecho();
                        else
                                throw new Exception("Dato duplicado");
                }
                if (dato.esMenor(anterior.getValor()))
                        anterior.setIzquierdo(nuevo);
                else
                        anterior.setDerecho(nuevo);
        }
}
```

Implementación recursiva de la inserción

A continuación la implementación recursiva en java de la operación de inserción.

```
 /**
 * insertar2 es la interfaz de la operaci n, llama al m todo
 * insertarRecursivo que realiza la operaci n y devuelve la raiz del nuevo
 *  rbol. A este m todo interno se le pasa la raiz actual, a partir de la
 * cual se describe el camino de b squeda, y al final, se enlaza. En un
 * arbol binario de b squeda no hay nodos duplicados; por ello, si se
 * encuentra un nodo igual que el que se desea insertar, se lanza una
 * excepcion
 *
 * @param valor
 *            valor del nodo a insertar
 */
public void insertar2(Object valor) throws Exception {
        Comparable dato = (Comparable) valor;
        raiz = insertarRec(raiz, dato);
}

private Nodo insertarRec(Nodo raizSub, Comparable dato) throws Exception {
        if (raizSub == null) {
                // caso base, termina la recursividad
                raizSub = new Nodo(dato);
        } else {
                if (dato.esMenor(raizSub.getValor())) {
                        Nodo iz = insertarRec(raizSub.getIzquierdo(), dato);
                        raizSub.setIzquierdo(iz);
                } else {
                        if (dato.esMayor(raizSub.getValor())) {
                                Nodo dr = insertarRec(raizSub.getDerecho(),
                                    dato);
                                raizSub.setDerecho(dr);
                        } else {
                                // Dato duplicado
                                throw new Exception("Nodo duplicado");
                        }
                }
        }
        return raizSub;
}
```

10.6.4 Búsqueda

La búsqueda de un nodo comienza en el nodo raíz y sigue estos pasos:

1. La clave buscada se compara con el nodo raíz.
2. Si las claves son iguales, la búsqueda se detiene.
3. Si la clave buscada es mayor que la clave raíz, la búsqueda se reanuda en el subárbol derecho. Si la clave buscada es menor que la clave raíz, la búsqueda se reanuda en el subárbol izquierdo.

Implementación iterativa de la búsqueda

```
 /**
 * Busca un dato en el  rbol  comenzando desde la raiz
 *
 * @param valor
 *            valor a buscar
 * @return
 */
public Nodo buscar(Object valor) {
        Comparable dato = (Comparable) valor;

        if (raiz == null)
                return raiz;
        else {
                // aux: auxiliar que va recorriendo los nodos, desde la raiz
                Nodo aux = raiz;
                while (aux != null) {
                        if (dato.esIgual(aux.getValor()))
                                return aux;
                        if (dato.esMenor(aux.getValor()))
                                aux = aux.getIzquierdo();
                        else
                                aux = aux.getDerecho();
                }
                return null;
        }
}
```

Implementación recursiva de la búsqueda

El método buscar2(), es la interfaz de la operación de búsqueda. La búsqueda del nodo la realiza el método localizar(), que recibe la referencia a la raíz del subárbol y el dato; el algoritmo de búsqueda es el siguiente:

1. Si el nodo raíz contiene el dato buscado, la tarea es fácil: el resultado es simplemente, su referencia y termina el algoritmo.
2. Si el árbol no está vacío, el subárbol específico por donde proseguir depende de que el dato requerido sea menor o mayor que el dato raíz.
3. El algoritmo termina si el árbol está vacío, en cuyo caso devuelve null.

```
 /**
 * Interfaz de buscar que invoca al m todo recursivo localizar
 *
 * @param buscado
 * @return
 */
public Nodo buscar2(Object buscado) {
        Comparable dato = (Comparable) buscado;
        if (raiz == null)
                return null;
        else
                return localizar(raiz, dato);
}
```

```
public Nodo localizar(Nodo raizSub, Comparable buscado) {
        if (raizSub == null)
                return null;
        else if (buscado.esIgual(raizSub.getValor()))
                return raizSub;
        else if (buscado.esMenor(raizSub.getValor()))
                return localizar(raizSub.getIzquierdo(), buscado);
        else
                return localizar(raizSub.getDerecho(), buscado);
}
```

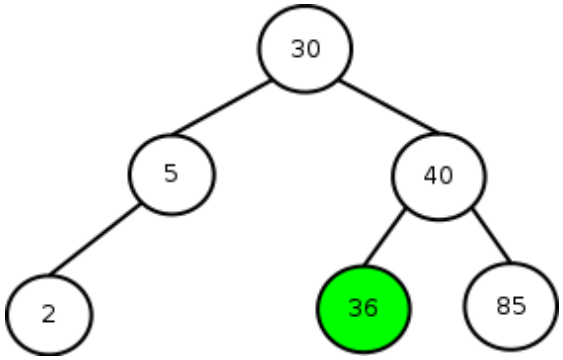
10.6.5 Eliminar un nodo

La operación de eliminación de un nodo es también una extensión de la operación de búsqueda, si bien es más compleja que la inserción, debido a que el nodo a suprimir puede ser cualquiera y la operación debe mantener la estructura del árbol binario de búsqueda después de quitar el nodo.

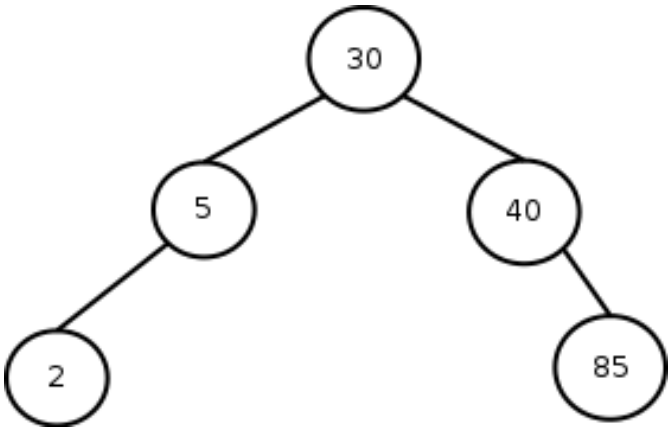
Los pasos a seguir son:

1. Buscar en el árbol para encontrar la posición del nodo a eliminar.
2. Si el nodo a suprimir tiene menos de dos hijos, reajustar los enlaces de su antecesor.
3. Si el nodo tiene dos hijos (rama izquierda y derecha), es necesario subir a la posición que éste ocupa el dato más próximo de sus subárboles (el inmediatamente superior o el inmediatamente inferior) con el fin de mantener la estructura del árbol binario de búsqueda.

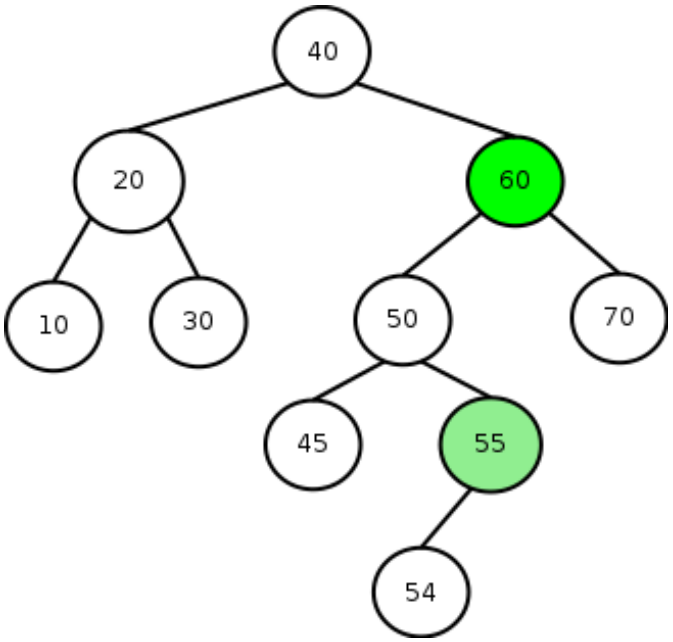
Ejemplo 1 (cuando el nodo a eliminar tiene menos de dos hijos)
Suprimir el elemento clave 36 del siguiente árbol binario de búsqueda:



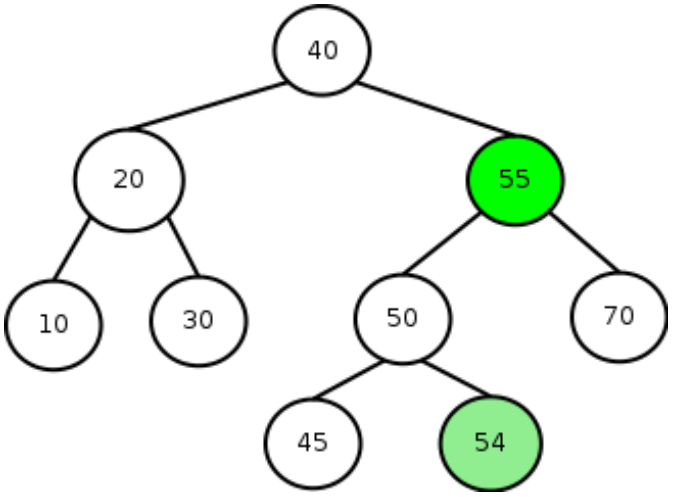
Cómo el nodo a eliminar es una hoja, simplemente se reajustan los enlaces del nodo procedente en el camino de búsqueda. El árbol resultante es:



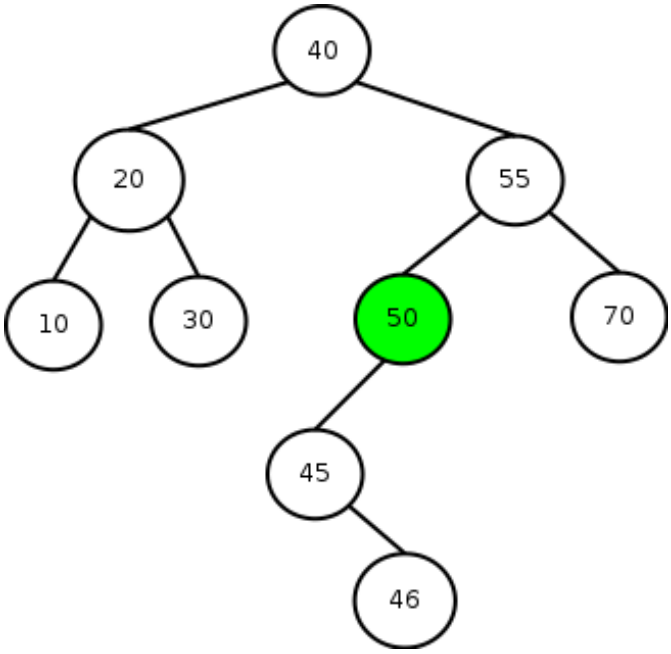
Ejemplo 2 (cuando el nodo a eliminar tiene dos hijos)
Borrar el elemento clave 60 del siguiente árbol



En este caso se reemplaza por el elemento mayor (55) en su subárbol izquierdo. También existe la opción de reemplazarlo por el elemento menor (70) de su subárbol derecho. El árbol después de eliminar el 60 quedaría.



Ejercicio. Borrar el elemento clave 50 del siguiente árbol. ¿Qué caso es?



10.6.6 Implementación iterativa de la eliminación

```
public boolean eliminar(Object valor){
        Comparable dato = (Comparable)valor;
        //Buscar el nodo a eliminar y su antecesor

        Nodo antecesor=null; //antecesor del nodo a eliminar
        // aux: auxiliar que va recorriendo los nodos, desde la raiz
        Nodo aux = raiz;
        while (aux != null) {
                if (dato.esIgual(aux.getValor())){
                        break;
                }
                antecesor = aux;
                if (dato.esMenor(aux.getValor()))
                        aux = aux.getIzquierdo();
                else
                        aux = aux.getDerecho();
        }
        if (aux==null)
                return false; //dato no encontrado

        //Si llega a este punto, el nodo a eliminar existe y es aux, y su antecesor es antecesor

        //Examinar cada caso
        //1. Si tiene menos de dos hijos, incluso una hoja, reajustar los enlaces de su antecesor
        if (aux.getIzquierdo()==null)
                if (aux.getValor().esMenor(antecesor.getValor()))
                        antecesor.setIzquierdo(aux.getDerecho());
                else
                        antecesor.setDerecho(aux.getDerecho());
        else if (aux.getDerecho()==null)
                if (aux.getValor().esMenor(antecesor.getValor()))
                        antecesor.setIzquierdo(aux.getIzquierdo());
                else
                        antecesor.setDerecho(aux.getIzquierdo());

        //El nodo a eliminar tiene rama izquierda y derecha
```

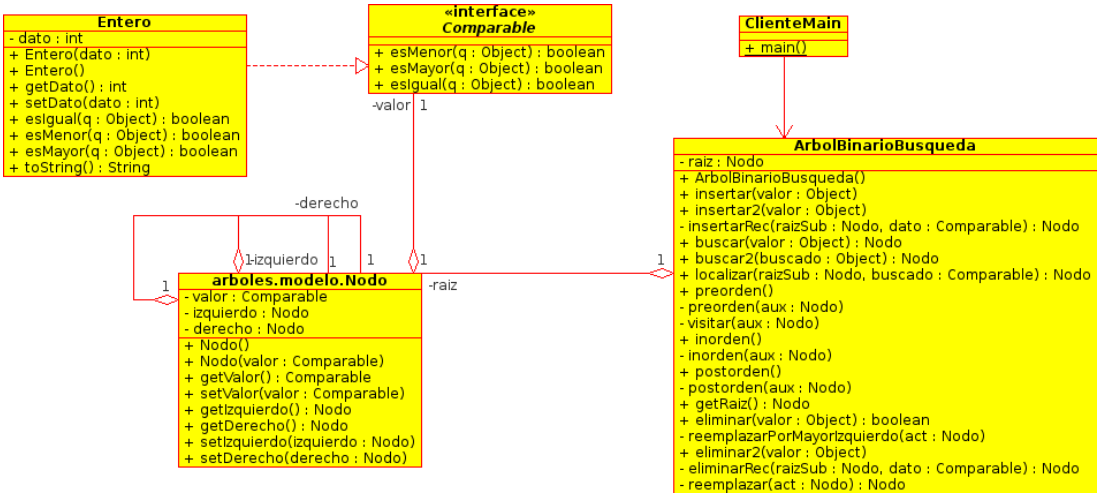
```
            reemplazarPorMayorIzquierdo(aux);

            aux=null;
            return true;
    }
    /**
     * Reemplaza el nodo actual, por el mayor de la rama izquierda
     * @param act nodo actual o nodo a eliminar que tiene rama izquierda y
         derecha
     */
    private void reemplazarPorMayorIzquierdo(Nodo act){
            Nodo mayor=act;
            Nodo ant = act;
            mayor = act.getIzquierdo();
            //Buscar el mayor de la rama izquierda
            //ant es el antecesor de mayor
            while(mayor.getDerecho()!=null){
                    ant = mayor;
                    mayor=mayor.getDerecho();
            }
            act.setValor(mayor.getValor()); //reemplaza
            //reajuste
            if(ant==act)
                    ant.setIzquierdo(mayor.getIzquierdo());
            else
                    ant.setDerecho(mayor.getIzquierdo());
    }
```

Ejercicio. Crear la implementación recursiva de eliminar.

10.6.7 Tarea

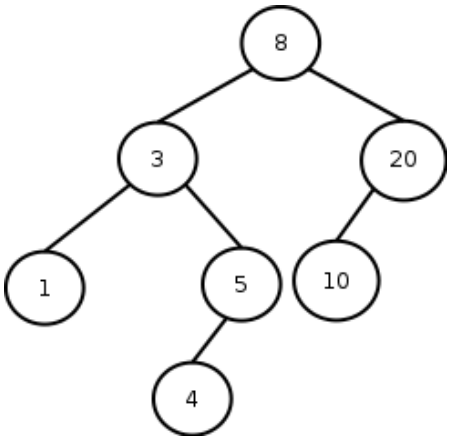
Hacer un programa completo en java que permita implementar todas las operaciones de un árbol binario de búsqueda: insertar, buscar, eliminar, recorridos (preorden, inorden, postorden). Preferiblemente de las operaciones básicas elaborar la versión recursiva y la versión iterativa. Además crear un ClienteMain que permita probar la correcta funcionalidad de las operaciones. La mayoría del código fuente ya está dado en en la teoría estudiada en este capítulo-

A continuación el diagrama de clases.



10.7 Ejercicios propuestos para Árboles Binarios

1. Un árbol binario de búsqueda (o un árbol binario ordenado) es aquel en que, dado un nodo, todos los datos del subárbol izquierdo son menores que los datos de ese nodo, mientras que todos los datos del subárbol derecho son mayores que ese nodo. Se denomina árbol binario de búsqueda porque se puede buscar en ellos un término utilizando un algoritmo de búsqueda binaria. Dado el siguiente árbol binario de búsqueda realizar los recorridos enorden, preorden y postorden del árbol:



2. Para cada una de las siguientes listas de letras,
   a) dibujar el árbol binario de búsqueda que se construye cuando las letras se insertan en el orden dado,
   b) realizar recorridos enorden, preorden y postorden del árbol y mostrar las secuencias de las letras que resultan en cada caso.
   - M, Y, T, E, R
   - T, Y, M, E, R
   - R,E,M, Y, T
   - C, Q, R, N F, L, A, K, E S
3. Dibuja los árboles binarios que representan las siguientes expresiones:
   a) (A+B)/(C-D)
   b) A+B+C/D
   c) A-(B-(C-D)/(E+F))
4. El recorrido preorden de un cierto árbol binario produce: ADFGHKLPQRWZ. Y el recorrido enorden produce: GFHKDLAWRQPZ. Dibujar el árbol binario.
5. Escribir un método recursivo que cuente las hojas de un árbol binario.
6. Escribir un método que determine el número de nodos que se encuentran en el nivel n de un árbol binario.
7. Escribir un método que tome un árbol como entrada y devuelva el número de descendientes del árbol.
8. Escribir un método booleano al que se le pase una referencia a un árbol binario y devuelva verdadero (true) si el árbol es completo y falso (false) en caso contrario.
9. Se dispone de un árbol binario de elementos tipo entero. Escribir métodos que calculen:
   a) La suma de sus elementos
   b) La suma de sus elementos que son múltiplos de 3
10. Diseñar un método iterativo que encuentre el número de nodos hoja en un árbol binario.
11. Escribir un método booleano identicos() que permita decir si dos árboles binarios son iguales.
12. Construir un método que encuentre el nodo máximo de un árbol binario.

13. Construir un método recursivo para escribir todos los nodos de un árbol binario de búsqueda cuyo campo clave sea mayor que un valor dado (el campo clave es de tipo entero).
14. Escribir un método que determine la altura de un nodo. Escribir un programa que cree un árbol binario con números generados aleatoriamente y muestre por pantalla:
    a) La altura de cada nodo del árbol
    b) La diferencia de alturas entre la ramas y derecha de cada nodo.
15. Diseñar métodos no recursivos que listen los nodos de un árbol inorden, preorden y postorden.
16. Diseñar un método que muestre los nodos de un árbol por niveles.
17. Un árbol binario de búsqueda puede implementarse con un array. La representación no enlazada correspondiente consiste en que para cualquier nodo del árbol almacenado en la posición i del array, si hijo izquierdo se encuentra en la posición 2*i y su hijo derecho en la posición 2*i+1. A partir de esta representación, diseñar los métodos con las operaciones correspondientes para gestionar interactivamente un árbol de números enteros.
18. Dado un árbol binario de búsqueda diseñar un método que liste los nodos del árbol ordenados descendentemente.
19. Escriba un programa que muestre o visualice por consola todos los nodos del árbol binario.
20. Escriba un applet que visualice gráficamente el árbol binario.
21. Dibujar el árbol de búsqueda equilibrado que se produce con las claves: 14, 6, 24, 35, 59, 17, 21, 32, 4, 7, 15 y 22.
22. Dada la secuencia de claves enteras: 100, 29, 71, 82, 48, 39, 101, 22, 46, 17, 3, 20, 25 y10, dibujar el árbol AVL correspondiente. Eliminar claves consecutivamente hasta encontrar un nodo que viole la condición de equilibrio y cuya restauración sea con una rotación doble.
23. Encontrar una secuencia de n claves que al ser insertadas en un árbol binario de búsqueda vacío permiten aplicar los cuatro casos de rotaciones: II, ID, DD, DI.
24. Dado un archivo de texto, construya el árbol AVL con todas sus palabras y frecuencias. El archivo se debe llamar carta.txt. El programa debe tener un método que, dada una palabra, devuelva el número de veces que aparece en el texto. Proponer el diagrama de clases que solucione este problema.

11 — Estructura de Datos No Lineales. Introducción a Grafos

11.1 Definiciones

11.1.1 Grafo, vértice y arista

Grafo: Los grafos son conjuntos de elementos denominados “vértices” (o nodos) unidos entre si por “aristas”.

Ejemplo

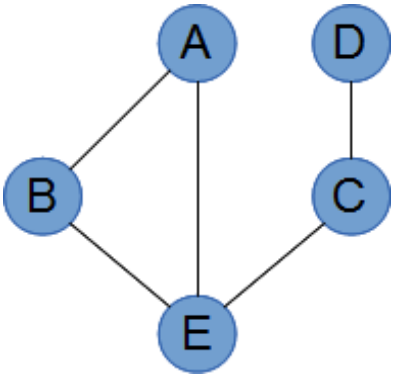


ILUSTRACIÓN 1. Se puede ver que A, B, C, D y E son vértices y que las líneas que unen los vértices son aristas.

11.1.2 Grafos dirigidos y no dirigidos

Las aristas también pueden estar “dirigidas” de manera de indicar cual puede ser el sentido del flujo en un grafo. En estos casos se dice que se está en un “grafo dirigido”. Si el sentido de las aristas no está especificado el grafo es “no dirigido”.

Ejemplo

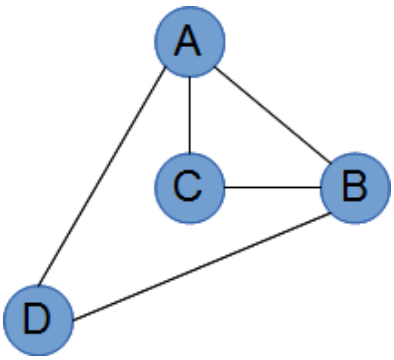


ILUSTRACIÓN 2. Ejemplo de un grafo no dirigido.

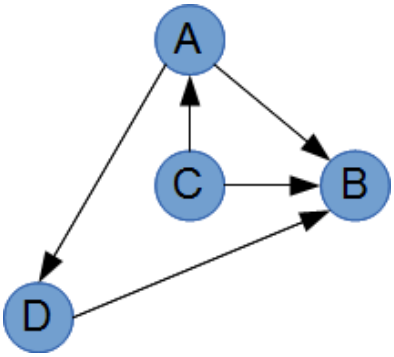


ILUSTRACIÓN 3. Ejemplo de un grafo dirigido.

11.1.3 Grado de entrada de un vértice

En un "grafo dirigido" la cantidad de aristas que se dirigen a un vértice se denomina "grado de entrada" del vértice.

En un "grafo no dirigido" la cantidad de aristas que confluyen en un vértice se denomina "grado de entrada" del vértice.

Ejemplo

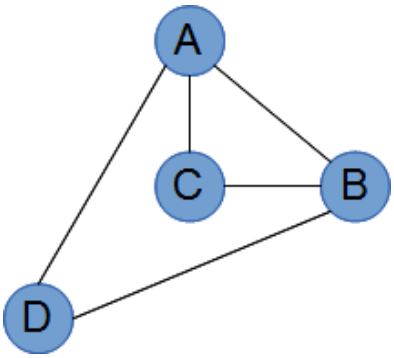


ILUSTRACIÓN 4 - En este grafo no dirigido los grados de entrada de los vértices A, B, C y D son respectivamente 3, 3, 2 y 2.

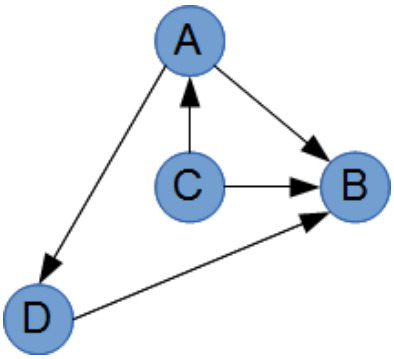


ILUSTRACIÓN 5 - En este grafo dirigido los grados de entrada de los vértices A, B, C y D son respectivamente 1, 3, 0 y 1.

11.1.4 Grafos ponderados y no ponderados

Un grafo dirigido o no, es "ponderado" cuando sus aristas tienen un peso asociado, de lo contrario es "no ponderado".

Ejemplo

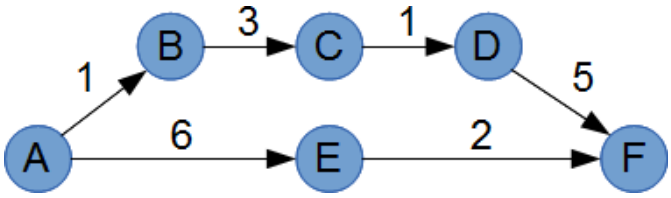


ILUSTRACIÓN 6 - Se puede ver un grafo ponderado y en la “TABLA 1” el correspondiente peso de cada arista.

Arista	Peso
AB	1
BC	3
CD	1
DF	5
AE	6
EF	2

11.2 Ejercicios - Definiciones

11.2.1 Ejercicios

Ejercicio 1

Dado el grafo de la ilustración 7 indicar cuales son sus vértices, cuales son sus aristas, si es o no dirigido, si es o no ponderado y en caso de ser ponderado indicar el peso de sus aristas.

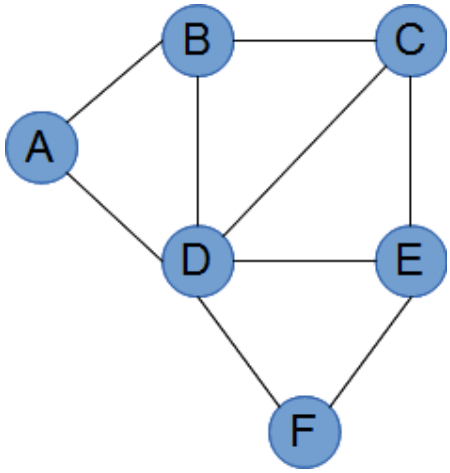


ILUSTRACIÓN 7 - Grafo del ejercicio 1.

Ejercicio 2

Dado el grafo de la ilustración 8 indicar cuales son sus vértices, cuales son sus aristas, si es o no dirigido, si es o no ponderado y en caso de ser ponderado indicar el peso de sus aristas.

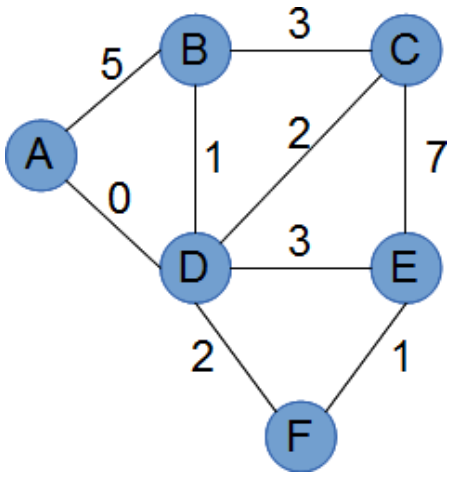


ILUSTRACIÓN 8 - Grafo del ejercicio 2.

Ejercicio 3

Dado el grafo de la ilustración 9 indicar cuales son sus vértices, cuales son sus aristas, si es o no dirigido, si es o no ponderado y en caso de ser ponderado indicar el peso de sus aristas.

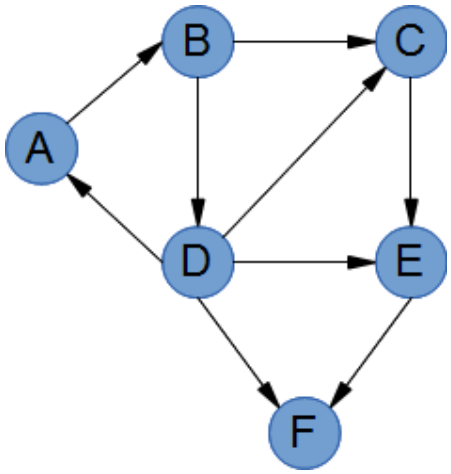


ILUSTRACIÓN 9 - Grafo del ejercicio 3.

Ejercicio 4

Dado el grafo de la ilustración 10 indicar cuales son sus vértices, cuales son sus aristas, si es o no dirigido, si es o no ponderado y en caso de ser ponderado indicar el peso de sus aristas.

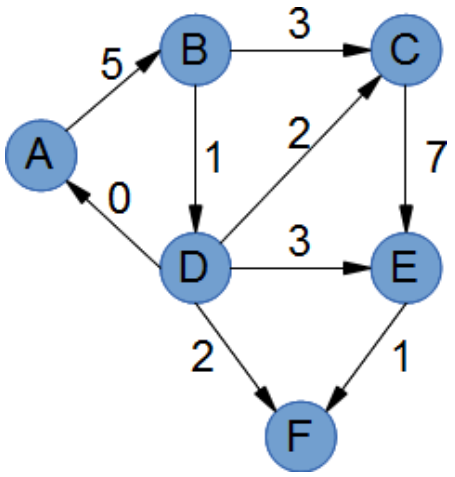


ILUSTRACIÓN 10 - Grafo del ejercicio 4.

11.2.2 Soluciones

Solución del ejercicio 1

El grafo de la ILUSTRACIÓN 7 es no dirigido y no ponderado, sus vértices son: A, B, C, D, E, y F. Sus aristas son: (A,B) o (B,A), (A,D) o (D,A), (B,C) o (C,B), (B,D) o (D,B), (C,D) o (D,C), (C,E) o (E,C), (D,E) o (E,D), (D,F) o (F,D) y (E,F) o (F,E). Los grados de entrada de los vértices A, B, C, D, E, y F son respectivamente: 2, 3, 3, 5, 3, y 2. Por no ser ponderado sus aristas no tienen pesos asociados.

Solución del ejercicio 2

El grafo de la ILUSTRACIÓN 8 es no dirigido y ponderado, sus vértices son: A, B, C, D, E, y F. Sus aristas son: (A,B) o (B,A), (A,D) o (D,A), (B,C) o (C,B), (B,D) o (D,B), (C,D) o (D,C), (C,E) o (E,C), (D,E) o (E,D), (D,F) o (F,D) y (E,F) o (F,E). Los grados de entrada de los vértices A, B, C, D, E, y F son respectivamente: 2, 3, 3, 5, 3, y 2. Los pesos de sus aristas son: (A,B) o (B,A) peso 5, (A,D) o (D,A) peso 0, (B,C) o (C,B) peso 3, (B,D) o (D,B) peso 1, (C,D) o (D,C) peso 2, (C,E) o (E,C) peso 7, (D,E) o (E,D) peso 3, (D,F) o (F,D) peso 2 y (E,F) o (F,E) peso 1.

Solución del ejercicio 3

El grafo de la ILUSTRACIÓN 9 es dirigido y no ponderado, sus vértices son: A, B, C, D, E, y F. Sus aristas son: (A,B), (D,A), (B,C), (B,D), (D,C), (C,E), (D,E), (D,F) y (E,F). Los grados de entrada de los vértices A, B, C, D, E, y F son respectivamente: 1, 1, 2, 1, 2, y 2. Por no ser ponderado sus aristas no tienen pesos asociados.

Solución del ejercicio 4

El grafo de la ILUSTRACIÓN 10 es dirigido y ponderado, sus vértices son: A, B, C, D, E, y F. Sus aristas son: (A,B), (D,A), (B,C), (B,D), (D,C), (C,E), (D,E), (D,F) y (E,F). Los grados de entrada de los vértices A, B, C, D, E, y F son respectivamente: 1, 1, 2, 1, 2, y 2. Los pesos de sus aristas son: (A,B) peso 5, (D,A) peso 0, (B,C) peso 3, (B,D) peso 1, (D,C) peso 2, (C,E) peso 7, (D,E) peso 3, (D,F) peso 2 y (E,F) peso 1.

11.3 Caminos y Ciclos

11.3.1 Definición de camino

Un “camino” es la secuencias de aristas que unen a dos vértices.

En un grafo dirigido se debe tener en cuenta el sentido de las aristas.

Ejemplo

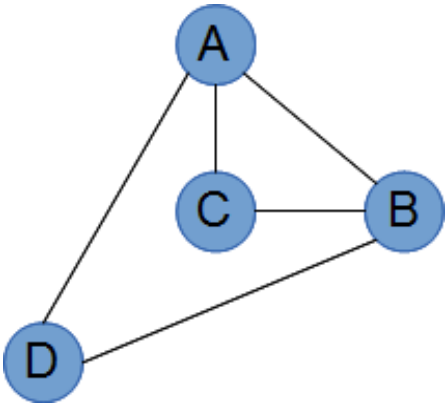


ILUSTRACIÓN 11 - Un caminos para ir del vértice C al vértice D puede ser la secuencia de vértices: “C, B, D”; otras secuencias válidas también son: “C, A, D”, “C, A, B, D” o “C, A, C, B, D”.

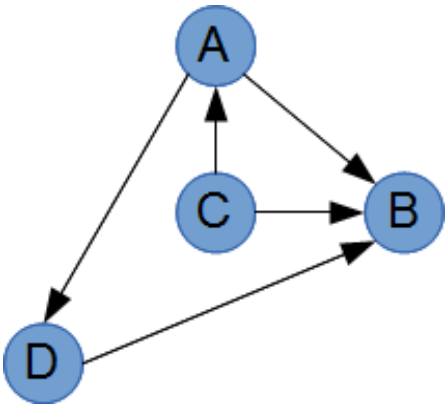


ILUSTRACIÓN 12 - Un caminos para ir del vértice C al vértice B puede ser la secuencia de vértices: “C, A, D, B”; otra secuencias válidas también es: “C, B”. En este caso por ser un grafo dirigido se debe tener en cuenta el sentido de las aristas para construir la secuencia.

11.3.2 Peso de un camino

En un grafo dirigido o no, pero ponderado, el “peso de un camino” es la suma de los pesos de todas las aristas que lo integran.

Ejemplo

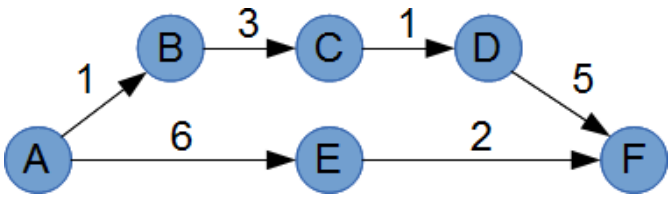


ILUSTRACIÓN 13 - Para el camino A, B, C, D; el peso es 1+3+1 = 5.

11.3.3 Ciclo

Un “ciclo” es un camino que empieza y termina en el mismo vértice.

Ejemplo

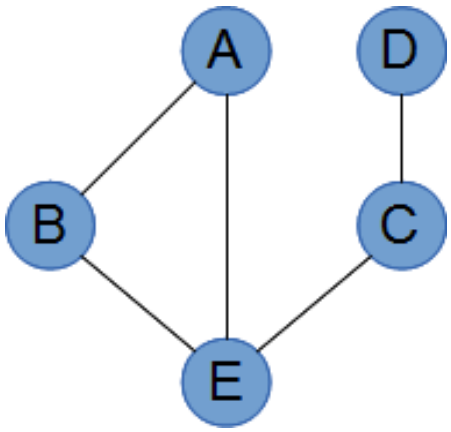


ILUSTRACIÓN 14 - A, B, E, A es un camino y un ciclo a la vez pues empieza y termina en el mismo vértices.

11.3.4 Grafos conexos

Grafo conexo

Un grafo dirigido o no, es “conexo” si a para todo par de vértices existe por lo menos un camino que los une sin importar el sentido de sus aristas.

Grafo fuertemente conexo

Un grafo dirigido o no, es “fuertemente conexo” si a para todo par de vértices existe por lo menos un camino que los une tomando en cuenta el sentido de sus aristas.

Enunciados

Enunciado I

Si en un grafo existe un ciclo (sin importar el sentido de las aristas) que contiene a todos los vértices del grafo, entonces el grafo es conexo.

Enunciado II

Si en un grafo dirigido existe un ciclo (importar el sentido de las aristas) que contiene a todos los vértices del grafo, entonces es fuertemente conexo.

11.4 Ejercicios - Caminos y Ciclos

11.4.1 Ejercicios

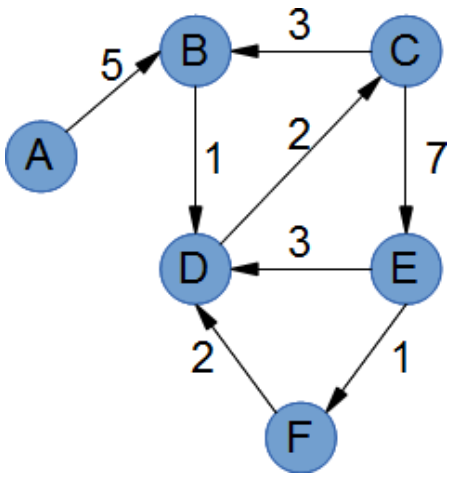


ILUSTRACIÓN 15 - Grafo dirigido y ponderado.

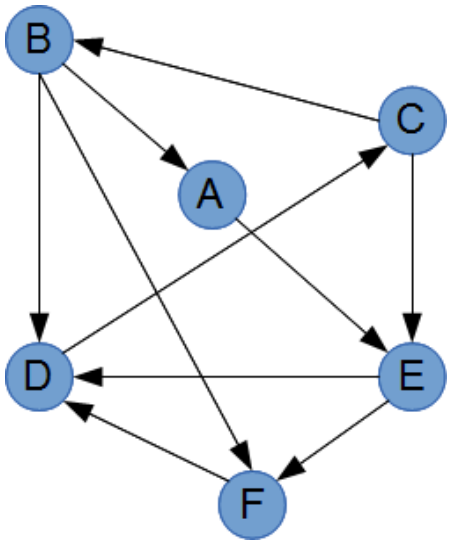


ILUSTRACIÓN 16 - Grafo dirigido y no ponderado.

Ejercicio 1

Dado el grafo de la "ILUSTRACIÓN 15":

a) Indique un camino que une el vértice C con el vértice D.
b) Indique el peso del camino encontrado.
c) Indique si el camino es único, en caso negativo de otros caminos posibles e indique sus pesos.
d) Indique si es posible encontrar algún ciclo en el grafo.

Ejercicio 2

Dado el grafo de la "ILUSTRACIÓN 15" demuestre si es o no conexo y/o fuertemente conexo.

Ejercicio 3

Dado el grafo de la "ILUSTRACIÓN 16" demuestre si es o no conexo y/o fuertemente conexo. Sugerencia: Buscar un camino que cumpla el "Enunciado I" y/o el "Enunciado II".

11.4.2 Soluciones

Solución del ejercicio 1

Para ir de C a D se tienen varios posibles caminos, por ejemplo una opción es realizar la siguiente secuencia de vértices: “C, E, D”, lo cual implica las aristas (C,E) y (E,D) las cuales tienen pesos 7 y 3 respectivamente, con lo cual el peso de este camino sería: 7 + 3 = 10.

Otro posible camino para ir de C a D es: “C, E, F, D”, unidos por las aristas (C,E), (E,F) y (F,D) con pesos 7, 1 y 2 respectivamente, dando un peso total para el camino de: 7 + 1 +2 = 10.

Para ejemplificar ciclos podemos observar la secuencia de vértices: “C, E, D, C”, otros ejemplos de ciclos son: “B, D, C, B” o “B, D, C, E, D, C, B”.

Solución del ejercicio 2

¿El grafo es conexo?. Si, pues para todo par de vértices existe un camino que los une si no se toma en cuenta el sentido de las aristas.

¿El grafo es fuertemente conexo?. No, pues por ejemplo no hay ningún camino que une al vértice A con el vértice B, ya que en este caso importa el sentido de las aristas.

Solución del ejercicio 3

En este caso se busca un camino que permita aplicar el “Enunciado I” y/o el “Enunciado II”.

El camino: “A,E,F,D,C,B,A” cumple el “Enunciado II”, por lo tanto el grafo es fuertemente conexo y por ende también conexo.

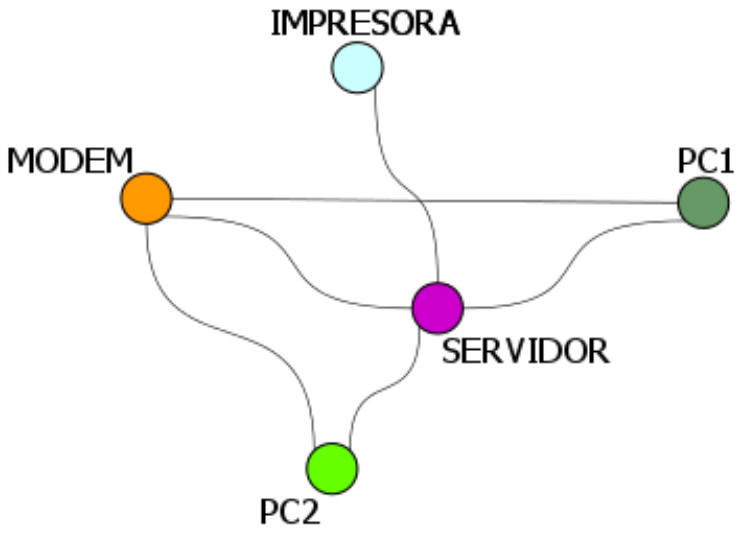
12 — Estructuras de Datos No Lineales. Grafos.

Introducción

Los grafos son estructuras de datos

- Representan relaciones entre objetos
- Relaciones arbitrarias, es decir
  - No jerárquicas
- Son aplicables en
  - Química
  - Geografía
  - Ing. Eléctrica e Industrial, etc.
  - Modelado de Redes
  - *De alcantarillado*
  - *Eléctricas*
  - *Etc.*

EJEMPLO:
Dado un escenario donde ciertos objetos se relacionan, se puede “modelar el grafo” y luego aplicar algoritmos para resolver diversos problemas

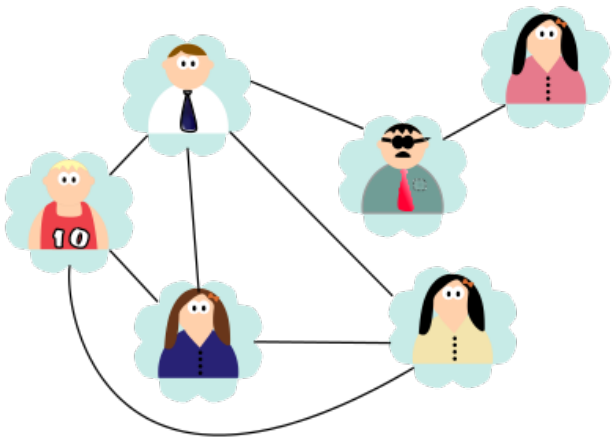


12.1 Introducción

¿Qué es un grafo?

En capítulos anteriores se discutieron tipos abstractos de datos que, además de almacenar información de las entidades en un problema, establecen relaciones de secuencialidad, orden o jerarquía entre multiples intancias de estas entidades. Por ejemplo, una lista de estudiantes, la cola de prioridad para ingresar a un cine, etc. El grafo es un tipo abstrato de datos donde no existen restricciones en la forma como las entidades se relacionan entre sí. Esta característica hace al grafo particularmente útil en problemas donde las relaciones entre entidades pueden representarse usando una red. La construcción del un grafo es relativamente simple, supongamos que queremos representar los intereses y relaciones entre personas. Una persona puede ser amigo de otra, extender su círculo social a los amigos de sus amigos, o eventualmente terminar una amistad. La figura 1.1, representa nuestra red social, una línea entre dos persona indica que son amigos entre sí.

Figura 12.1:



Cuando dos personas establecen una amistad usualmente comparten información. Así que además de amistad, las lineas de nuestra red tambien representan este flujo de información. El flujo de información puede ir en una o ambas direcciones, las saetas en los extremos de las lineas indican la dirección del flujo de información.

Figura 12.2:

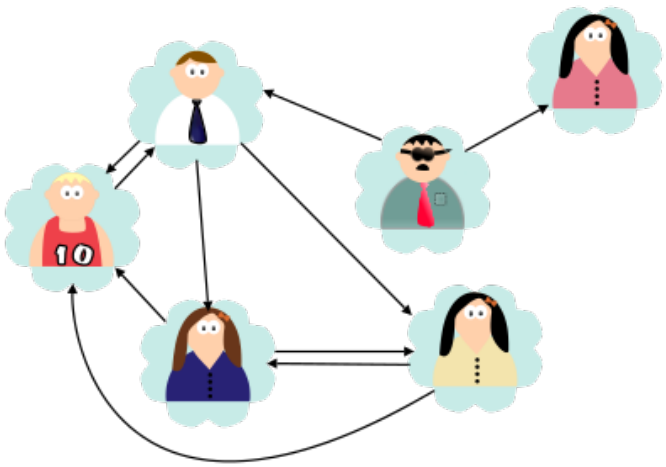
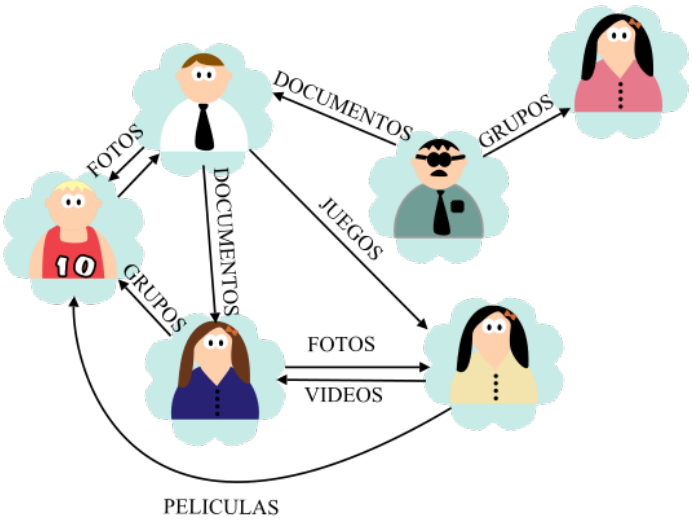
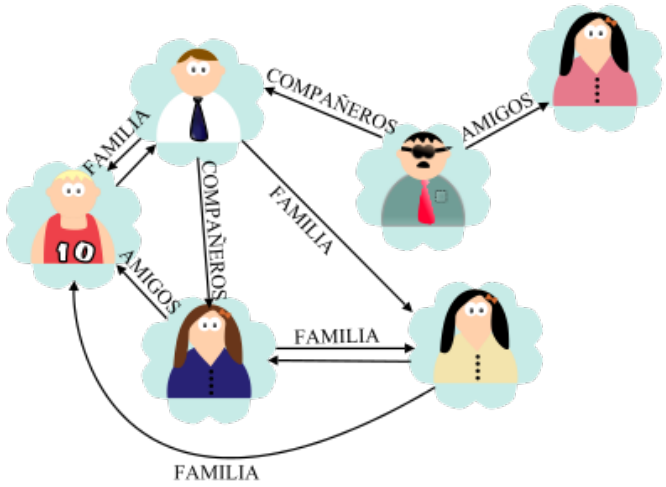


Figura 12.3:



Si consideramos además que un amigo pueden ser un "buen amigo", ün colega de trabajo.º simplemente un çonocidoçada amistad de nuestra red puede ser etiquetada según su tipo. En consequencia, nuestra red ahora no solo contiene entidades (personas) y relaciones (líneas) sino también la importancia relativa de estas relaciones (etiquetas), ver la figura 1.4. Como vemos en nuestro red hay una entidad más importante que otra, no existe ningún orden especifico para leer o modificar la información de una entidad o límites para el número de relaciones entre entidades.

Figura 12.4:



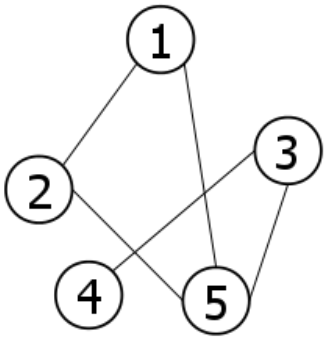
En la teoría de grafos, cada uno de los elementos de la figura 1.4, tiene un nombre: a las instancias de lasentidades se las conoce como vértices, a las lineas como arcos y a las etiquetas (si son numéricas) como factores de peso. Entonces, un grafo es un conjunto de nodos unidos por arcos. Muchos problemas computacionales pueden explorarse usando grafos. Por ejemplo: las redes electricas, las redes de transporte o las relativamente recientes redes sociales. En las siguientes secciones se provee la descripción formal de un grafo y algoritmos útiles para resolver varios problemas prácticos.

12.2 Definición

Definición formal

Un grafo G es un conjunto V de vértices y un conjunto A de arcos. Donde, los elementos de V se las entidades de un problema y los elementos de A indican la existencia de alguna relación entre entidades del conjunto V. Sea un grafo G={V,A} donde: V={1,2,3,4,5} , yA={(1,2),(1,5),(2,5),(4,3),(3,5)} En este ejemplo, las entidades representadas en el grafo son caracteres, instancias de un tipo de dato simple, pero en general los vertices pueden referirse a cualquier TDA.Cuando existe una relación entre un par de vértices se dice que son adyacentes. Nuestra definición de grafo no incluye ningun mecanismo para inferir la existencia o no alguna relación entre pares de vértices. Usualmente, estás reglas se extraen del problema a resolver y por ende no se pueden generalizar. A pesar de esto, como veremos a continuación, los diferentes tipos de relación entre vértices determinan ciertas propiedades de los grafos. Si colocamos cada elemento de V dentro de un circulo y una línea entre cada par de elementos referenciados en A se genera el siguiente gráfico:

Figura 12.5: Ilustración de un grafo.

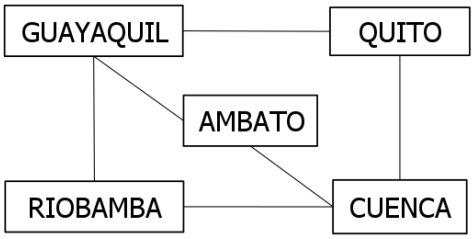


12.3 Tipos de Grafos y Conceptos

12.3.1 Conceptos asociados a los grafos

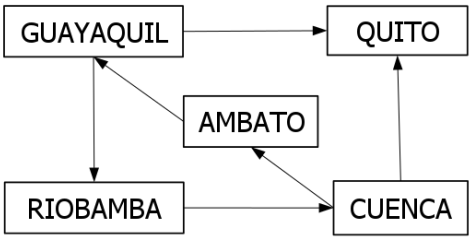
Cuando se emplean grafos para resolver problemas computacionales es interesante conocer que tipo de relación existen entre los vértices. La relación entre un vértice y otro puede ser recíproca, por ejemplo una carretera de dos sentidos que conecta un par de cuidades implica que se puede ir de una cuidad a otra y viceversa. Cuando entre cada par de vértices se verifica este tipo de relación se tiene un grafo es no dirigido. Esto implica que el orden de cada par de vértices del conjunto de arcos R no es relevante. En un grafo no dirigido a los arcos se los suelen llamar aristas.

Figura 12.6: Representación de carreteras entre ciudades.



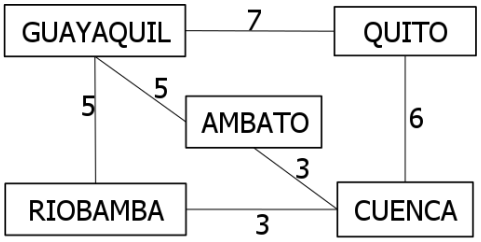
Si la relación entre vértices es simple, en nuestro ejemplo de ciudades una carretera de un solo sentido, el orden de los pares de vértices en R determinan la "dirección" de la relación. Así, un par ordenado de ciudades en R indica que hay una ruta que permite ir de la primera a la segunda ciudad y no viceversa. Si para cada par de vértices se verifica esta condición, se tiene un grafo dirigido. En un grafo dirigido los arcos tiene saetas que indican el vértice destino de la relación.

Figura 12.7: Dirección del recorrido entre las ciudades.



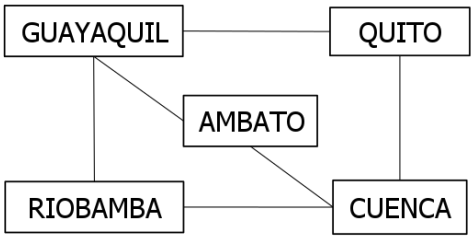
Cuando se asocia un número a un arco, se obtiene un grafo valorado. A este número se lo llama factor de peso. El cálculo del factor de peso y su interpretación es dependiente del problema. Por ejemplo, la longitud de la carretera que conecta dos cuidades o el grado de amistad entre dos personas. En general, se espera que los valores de los factores de peso sean siempre números positivos pues muchos algoritmos asociados a los grafos requieren esta condición.

Figura 12.8: Explicación del factor de peso, distancia entre ciudades.



En muchos problemas relacionados con grafos es importante conocer cuantos vértices son adyancentes a un vértice dado. A éste número se le conoce como el grado del vértice. La manera de calcular el grado de un vértice depende de si el grafo es dirigido o no dirigido. En el caso de un grafo no dirigido, el grado de un vértice es igual al número de arcos que conectan a ese vértice con sus vértices adyacentes.

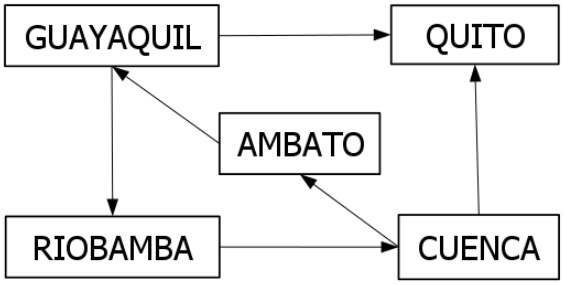
Figura 12.9: Se visualiza que la ciudad de Guayaquil contiene 3 aristas.



Grado(Guayaquil) = 3

Para los grafos dirigidos, el grado de un vértice puede calcularse en función del número de arcos que llegan o salen del vértice. Respectivamente, el grado de entrada y salida de un vértice.

Figura 12.10: La ciudad de Ambato cuenta con dos arcos, uno de salida y otro de entrada.



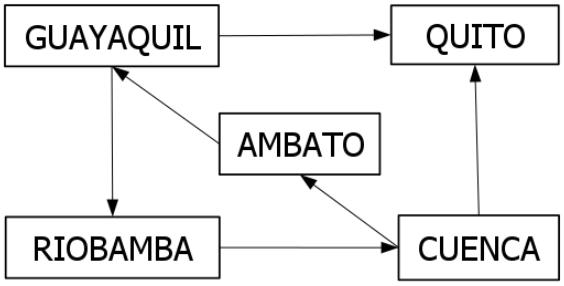
GradoEntrada(Ambato) = 1 y GradSalida(Ambato) = 1

12.3.2 Caminos en grafos

Si dos vértices son adyancentes, es decir si hay un arco entre ellos, es fácil deducir que hay un camino directo entre ambos. Ahora, si un par de vértices no son adyancentes el camino que permite ir de un vértice V0 a otro Vn va a requerir pasar por uno o más vértices intermedios. En general, un camino P en un grafo G, desde V0 a Vn es la secuencia de n+1 vértices {V0,V1,..Vn} tal que (Vi,Vi+1) ε R para $0 \leq i \leq n$ Vi. Nuestra definición de camino se aplica a grafos dirigidos y no dirigidos. En caso de los grafos dirigidos es necesario considerar la dirección de los arcos.

Existen varios algortimos para extraer caminos de un grafo, por ahora nos concentraremos en las propiedades de los caminos. La primera y más simple de calcular es la longitud de un camino es decir el número de arcos que lo forman. En nuestro ejemplo, esto se traduce en la ciudades por las que hay que pasar para llegar a nuestro destino.

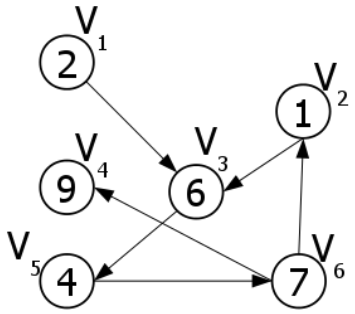
Figura 12.11: Recorrido realizado entre ciudades, con su respectivo sentido.



Camino entre Ambato y Cuenca
P = {Ambato, Guayaquil, Riobamba, Cuenca},Longitud: 3

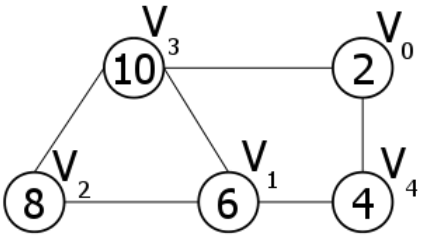
Cuando todos los vértices de un camino son distintos se dice que se tiene un camino simple. Si algún vértice se repite en un camino P, se dice que existe un bucle. Como veremos más adelantes para algunos algoritmos relacionados con el recorrido de los vértices de grafos bucles representan un potencial problema.

Figura 12.12: Se puede obtener un camino entre V2 y V2, determinamos que existe la presencia de un bucle.



Otro concepto importante en grafos, es la conectividad. En el ejemplo de la figura 3.1, podriamos preguntarnos si hay manera de llegar a cualquier ciudad sin importar donde estemos. Si es así, nuestro grafo es conexo ya que existe un camino entre cualquier par de vértices.

Figura 12.13: Planteamiento de un grafo conexo.



Si hay más de un camino entre cualquier par de nodos nuestro grafo es fuertemente conexo. Este concepto se aplica usualemente a los grafos dirigidos.

Figura 12.14: Grafo fuertemente conexo.

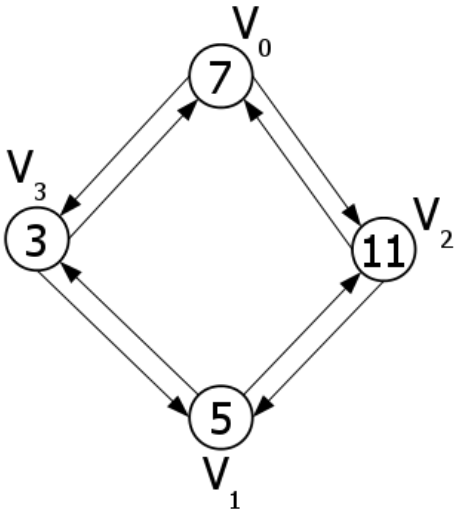
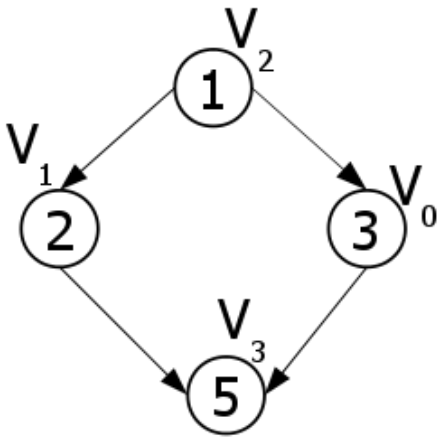


Figura 12.15: Ilustración de un grafo no conexo



12.4 TDA Grafo

Definicion formal

El TDA grafo, independientemente de como se represente, contiene dos elementos: un conjunto de V de vértices y un conjunto A de arcos. Los TDAs definidos para los elementos en V y A deben contener las referencias necesarias para establecer las relaciones entre los vértices del grafo.

El conjunto mínimo de operaciones asociadas el TDA grafo debe permitir colocar y remover los elementos de los conjuntos V y A.

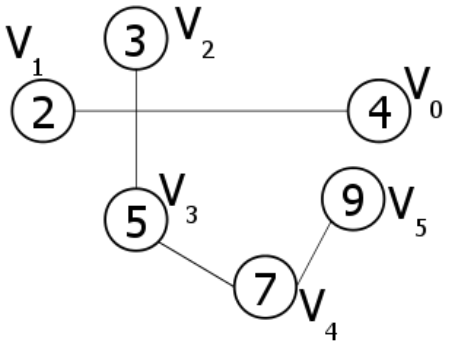
Grafo CrearGrafo(); // crear un grafo vacio
void AñadirVertice(Grafo G, Vértice V) *Añadir un nuevo vértice*
void BorrarVertice(Grafo G, Vertice V) *Eliminar un vertice existente*
void AñadirArco (Grafo G, Vértice V1, Vértice V2, int p) *Crear un arco de V1 a V2 con factor de peso p*
Void BorrarArco(Grafo G, Vértice V1, Vértice V2) *Eliminar un Arco*
bool EsAdyacente(Grafo G, Vértice V1, Vértice V2) *Conocer si dos vértices son o no adyacentes*

Estas operaciones se traducen en funciones que operan sobre la representación escogida para almacenar los datos del grafo. Las operaciones para manipular del contenido de los vertices y arcos dependen del TDA usado, por ende no forma parte de la definición del grafo. Usualmente, los vértices y arcos suelen contener información adicional relevante para el problema a resolver.

12.5 Representación de grafos

Existen dos maneras comunes de representar a un grafo en un computador. La primera es la matriz de adyacencia que asume que los vértices están enumerados. Así, dado un grafo G = (V, A), se definen dos matrices: la matriz V que contiene de forma ordenada la información de los vértices V0, V1, ... Vn y la matriz A donde el valor de cada Aij indica si hay o no un arco, que tiene como origen a Vi y destino a Vj. Tipicamente, Aij=1 implica la existencia de un arco y Aij=0 lo contrario.

Figura 12.16: Esquema de un grafo. Descripción de la enumeración, relación y contenido de los vértices.



Cuadro 12.1: Matriz de adyacencia

	V0	V1	V2	V3	V4	V5
V0	0	1	0	0	0	0
V1	1	0	0	0	0	0
V2	0	0	0	1	0	0
V3	0	0	1	0	1	0

Si el grafo fuese valorado, en vez de 1 se coloca el factor de peso. Llevar esta representación a un lenguaje de programación no es complicado, basta con declarar un TDA que almacene ambas matrices y el tipo del grafo.

```
Grafo{
    bool dirigido;
    int MatAdj [Max_vertices][Max_vertices];
    Vertices V [Max_vertices];
}
```

El uso de una matriz de adyacencia facilita la manipulación de arcos. Crear un arco entre los vértices V1 y V2 se traduce en un simple asignación (G.MatAdj[V1][V2] = 1). Si el grafo es no dirigido, es necesario hacer la misma asignación invirtiendo los índices de los vértices (G.MatAdj[V2][V1] = 1). Es importante anotar que los subíndices en cada vértice corresponden a sus posiciones en la matriz V. Una desventaja del uso de la matriz de adyacencia es que para matrices con muchos vértices, pero pocos arcos se deperdicia memoria almacenado 0‘s.

Otra forma de representar un grafo es por medio de listas enlazadas. Usualmente conocido como lista de adyacencia el TDA grafo consiste en una lista simple donde cada nodo contiene a un vértice Vi y una referencia a otra lista cuyos nodos representan los arcos conectan Vi son sus vértices adyacentes. En comparación con la matriz de adyacencia esta nueva representación es más compleja pues requiere definir un TDA para cada tipo de nodo y la búsqueda de arcos toma más tiempo. Sin embargo, la lista de adyacencia tiene la ventaja de permitir usar grafos con un número indeterminado de vértices o arcos.

```
Grafo {
    Lista ListaAdj;
}
NodoListaAdj {
    Vertice V;
```

```
    Lista* ListaArcos;
}
NodoListaArcos{
    Vertice * Vorigen;
    Vertice * Vfuente;
    int factor_peso;
}
```

Figura 12.17:

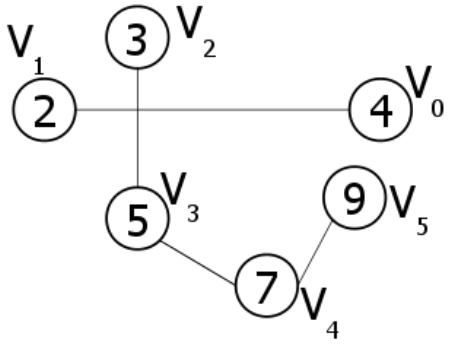
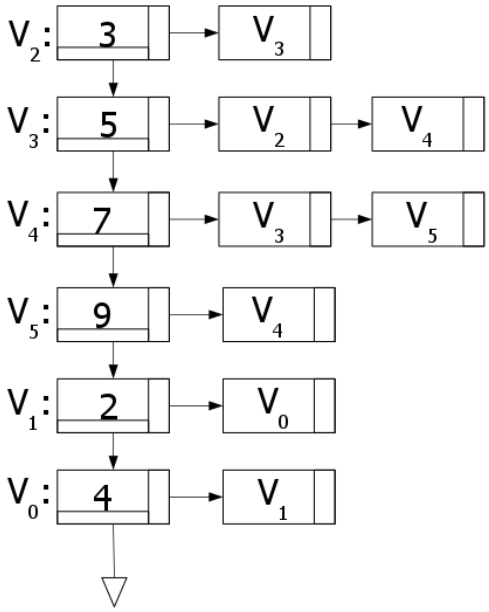


Figura 12.18: Representación de la lista de adyacencia consecuente al grafo de la figura 12.16



12.6 Recorrido de un Grafo

Recorrido de un grafo

Recorrer un grafo consiste en tratar de visitar todos los vértices posibles desde un vértice de partida D. Saber a que vértices se puede llegar es una operación necesaria en muchos problemas prácticos. **(Ejemplo)**. Dos algoritmos comunmente utilizados son el recorrido en anchura y el

recorrido en profundidad. El recorrido por anchura da prioridad a visitar primero los vértices más cercanos a D. El recorrido en profundida de prioridad a visitar primero los nodos más alejados a D.

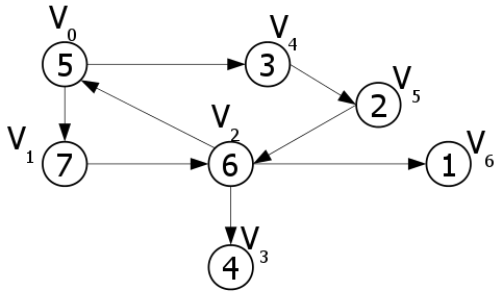
Recorrido en anchura

Este algoritmo utiliza un cola fifo C que contiene temporalmente los vértices a visitar, en el grafo a ser procesado; el frente de la cola W se refiere al vértice visitado en cada iteración.

1. Encolar vértice de partida en la cola C.
2. Marcarlo como "visitado".
3. Mientras la cola no este vacía.
4. Desencolar por el frente de la cola, vértice W.
5. Mostrar contenido de W.
6. Marcar como visitados los vértices adyacentes de W que no hayan sido ya visitados y encolarlos

Los pasos 3 a 6 se repiten hasta que la cola esté vacía.

Figura 12.19: Ejemplo práctico de un grafo dirigido.



Iteración	C	W	Adyacentes a W no visitados
1	5	5	7,3
2	7,3	7	6
3	3,6	3	2
4	6,2	6	4,1
5	2,4,1	2	——
6	4,1	4	——
7	1	1	——

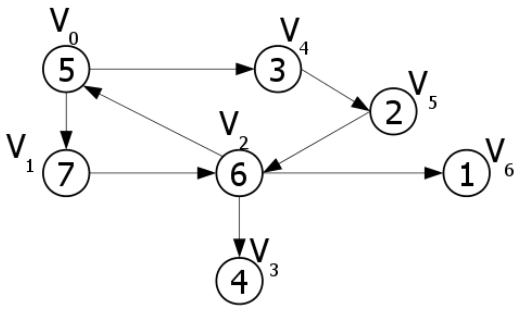
Recorrido en produndidad

Este algoritmo utiliza un cola pila S que contiene referencias a los vértices a visitar del grafo, el tope de la pila W se refiere al vértice visitado en cada iteración.

1. Hacer push al vértice partida en la pila S.
2. Marcarlo como visitado
3. Mientras que la pila no este vacía
4. Hacer pop de la pila, vertice W.
5. Mostrar el contenido de W
6. Marcar W como visitados los vértices adyacentes de W que no haya sido visitado y hacer push.

Los pasos 3 a 7 se repiten hasta que la cola esté vacía.

Figura 12.20:



Iteración	S	W	Adyacentes de W no visitados
1	5	5	7,3
2	7,3	3	2
3	7,2	2	6
4	7,6	6	4,1
5	7,4,1	1	—
6	7,4	4	—
7	7	7	—

12.7 Algoritmos útiles en Grafos

En esta sección estudiamos algunas técnicas frecuentemente usadas para extraer información acerca los caminos existentes en un grafo. La definición de grafos solo permite conocer si dos vértices son adyacentes, es decir si existe un camino de longitud 1 entre ellos. Si queremos saber si existe un camino, de cualquier longitud, entre dos vértices tenemos que buscar todos los posibles caminos en el grafo. A fin de simplificar la explicación de los algoritmos asumiremos que los grafos están implementados usando la matriz de adyacencia. El resultado de nuestra búsqueda de caminos se almacenará en una matriz P, conocidad como matriz de caminos. Esta matriz tiene la misma naturaleza que la matriz de adyacencia salvo que la entrada Pij indica si existe un camino entre los vertices Vi y Vj.

12.7.1 El algoritmo de Warshall

Este algoritmo fue propuesto por varios científcos de las ciencias computacionales entre 1959 y 1962. La versión más eficiente se le atribuye a Warshall. En algoritmo parte de una matriz P0 que indica si entre un par de vertices Vi y Vj existen o nó un camino directo, es decir es una copia de la matriz de adyacencia. A partir de P0 se deriva una serie de matrices P1,P2,...,Pn donde las entradas de Pk[i][j] indican si hay un camino de longitud k entre los vertices Vi y Vj. A fin de derivar Pk (k>0) se añade el vertice Vk-1 a los caminos existentes en Pk-1 y se verifica si se forman nuevos caminos entre Vi y Vj que pasen por Vk-1.

La idea es relativamente sencilla, a partir P0, se agrega el vertice V0 a todos los posibles caminos y se verifica si se puede forma un nuevo camino entre Vi y Vj que incluya a V0. El resultado se almacena en P1 usando la siguiente expresion:

P1[i][j]=P0[i][j] | P0[i][0] && P0[0][j]

en general, para k>0

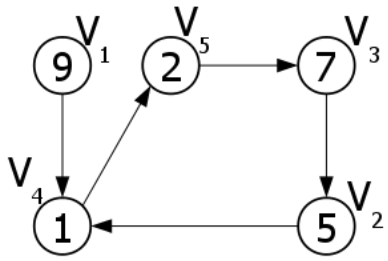
Pk[i][j]=Pk-1[i][j] | Pk-1[i][k] && Pk-1[k][j]

Pn es la matriz de caminos que estamos buscando. El algoritmo se describe a continuacion, parte un grafo G y la matriz de camino P.

```
MatrizAdy P;
CopiarMatrices(P, G.A);
for(k = 0; k < G.nvertices; k++){
      for(i = 0; i < G.nvertices; i++){
            for(j = 0; j < G.nvertices; i++){
                  P[i][j]= P[i][j] || (P[i][k] && P[k][j]);
            }
      }
}
```

Este algoritmo usa tres lazos de repetición anidados, el más externo sirve para añadir el siguiente Vk-1 al análisis de caminos. Los más internos para probar con cada par de vértices si se genera un nuevo camino. El siguiente ejemplo demuestra como se genera Pk para el grafo de la figura 12.21 .

Figura 12.21: Grafo dirigido para representar el análisis de caminos.



			P0		
	V1	V2	V3	V4	V5
V1	0	0	0	1	0
V2	0	0	0	1	0
V3	0	1	0	0	0
V4	0	0	0	0	1
V5	0	0	1	0	0

Dado que las entradas de P son producto de una operación lógica, esta matriz solo nos indica si existe un camino, no cual és. Si quieren extraer los caminos se debe de modificar el algoritmo para que almacene cada uno de los caminos generados.

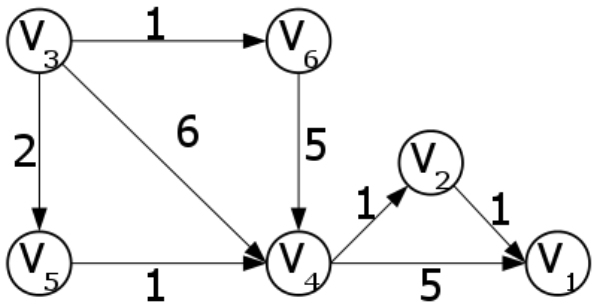
12.7.2 Algoritmo de Dijkstra

Usualmente, no basta con conocer si hay un camino entre dos vértices sino que se requiere saber cual de todos los posibles camino es el más corto. El significado del camino más corto depende de lo que el grafo represente. Por ejemplo, si los factores de peso un grafo son distancias entre ciudades, el camino mas corto es aquel que literalmente permite llegar de una ciudad en el menor tiempo. El algoritmo que explicaremos a continuación fue propuesto por Edsger Dijkstra en 1956. Este algoritmo evalua en cada paso, potenciales caminos y conserva aquel que reporta de menor distancia a los ya existentes. Así, dado un vertice inicial V0, el algoritmo llena un arreglo D con las distancias más cortas al resto de vértices siguiendo estos pasos:

1. Inicializar D con los factores de peso de los vértices adyacentes a V0 e infinito para los no adyacentes.

2. En cada iteración, escoger un vértice Vk que no haya sido escogido antes y que tenga el camino más corto a V0.
3. Revisar si alguna distancia D puede ser mejorada pasando por Vk.
4. Repetir 2 y 3 , hasta que se hayan visitado todos los vértices.

Figura 12.22: Grafo dirigido con factor de peso, ejemplo empleado en el desarrollo del Algortimo de Dijkstra



El algoritmo de Dijkstra es un algoritmo ávido, pues la selección de Vk considera primero aquel vértice que reporta la menor distancia. Esto implica el uso de una cola de prioridad para ordenar los vértices no evaluados según su distancia actual a V0.

Visitados	Vk	D[0] V1	D[1] V2	D[2] V3	D[3] V4	D[4] V5	D[5] V6
V3	V3	∞	∞	0	6	2	1
V3,V6	V6	∞	∞	0	6	2	1
V3,V6,V5	V5	∞	∞	0	3	2	1
V3,V6,V5,V4	V4	8	4	0	3	2	1
V3,V6,V5,V4,V2	V2	5	4	0	3	2	1

12.7.3 Algoritmo de Floyd

Extraer los caminos más cortos entre cualquier par de vértices se puede hacer aplicando el algoritmo de Dijkstra a cada vértice y utilizar cada una de las matrices D resultantes para construir una matriz de caminos mínimos. Una mejor alternativa, en términos de rapidez de ejecución, es el algoritmo de Floyd. Este se basa en el algoritmo de Warshall. Genera en cada iteración nuevos caminos entre Vi y Vj que pasen por un Vk y actualiza las entradas de la matriz de caminos con las distancias mínimas. En la práctica, esto implica reemplazar la condicion:

Pk[i][j]=Pk-1[i][j] | Pk-1[i][k] && Pk-1[k][j]

por

Pk[i][j]=Min(Pk-1[i][j] , Pk-1[i][k] + Pk - 1[k][j])

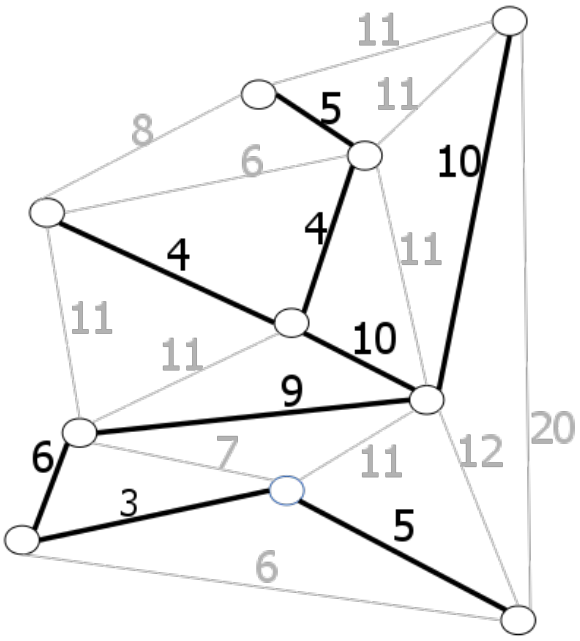
Donde la función Min retorna el menor entre dos valores.

12.7.4 Árbol de expasion mínima

Dado un grafo G, no dirigido, valorado, con pesos no negativos el árbol de expansión mínima es un grafo conexo con la propiedad de que la suma de los factores de pesos de todas las aristas es mínima. El árbol de expansión mínima tiene múltiples aplicaciones en problemas de

optimización. Por ejemplo, si se quiere reducir el costo de producción de un circuito electrónico cada chip del circuito puede ser representado por un vértice y cada posible conexión entre ellos con un arco cuyo factor de peso es la cantidad de cobre necesaria para la conexión. Reducir la cantidad total de cobre necesario para conectar elementos electrónicos se puede hacer extrayendo el árbol de expansión mínima dado que la suma de sus factores de peso ser traduce en la cantidad total de cobre a colocar por cada circuito. Dos algoritmos comúnmente empleados para extraer el árbol de expansión mínima de un grafo se explican a continuación:

Figura 12.23: Árbol de expansión mínima



Dos algoritmos comunmente empleados para extraer el árbol de expansión mímina se explican a continuación.

12.7.5 Algoritmo de Prim

Este algoritmo construye el árbol buscando vértices adyacentes con arcos de menor peso posible, de forma similar al algoritmo de Dikjstra. Se parte un vértice inicial de un grafo y se lo copia a un árbol vacío En el grafo se busca el vértice adyacente al vértice inicial con el arco de menor peso. Se agrega al árbol una copia del vértice y del arco seleccionado. Se sigue buscando el arco de menor peso que una a los vértices del árbol con alguno de los vértices del grafo no seleccionados hasta que el árbol tenga el mismo número de vértice que el grafo. El árbol resultante es el árbol de expansión mínima. Los pasos del algoritmo de Prim se muestran a continuación:

1. Se crea un grafo M con los mismos vértices del grafo original G y sin arcos.
2. En G se selecciona un vértice de partida Vo que se marca como visitado.
3. Los arcos de Vo, cuyos vértices destino no han sido visitados, se encolan en C.
4. Se desencola de C el arco menor en peso y se copia en M.
5. El vértice destino del arco de menor peso Vd se marca como visitado en G
6. Vo es ahora igual a Vd.
7. Se repiten los pasos del 3 al 6 hasta que el número de vértices marcados como visitados

sea igual al número de vértices en M.

Figura 12.24: Grafo conexo, no dirigido. Ejemplo empleado para extraer el árbol de expansión minima.

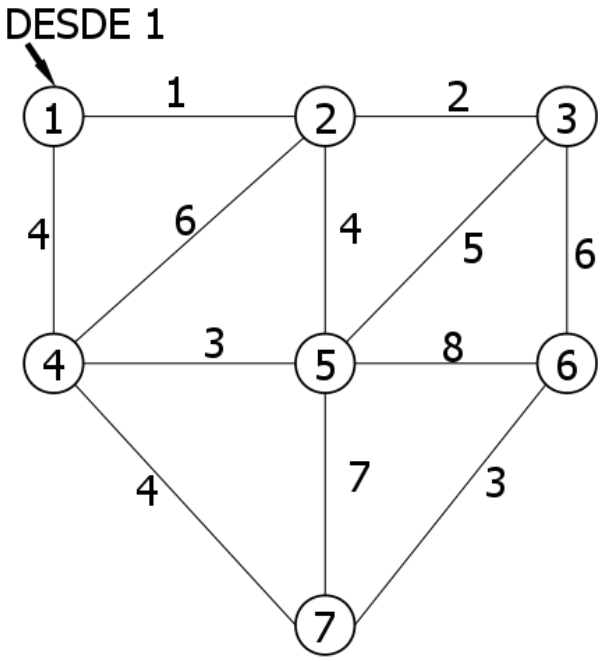
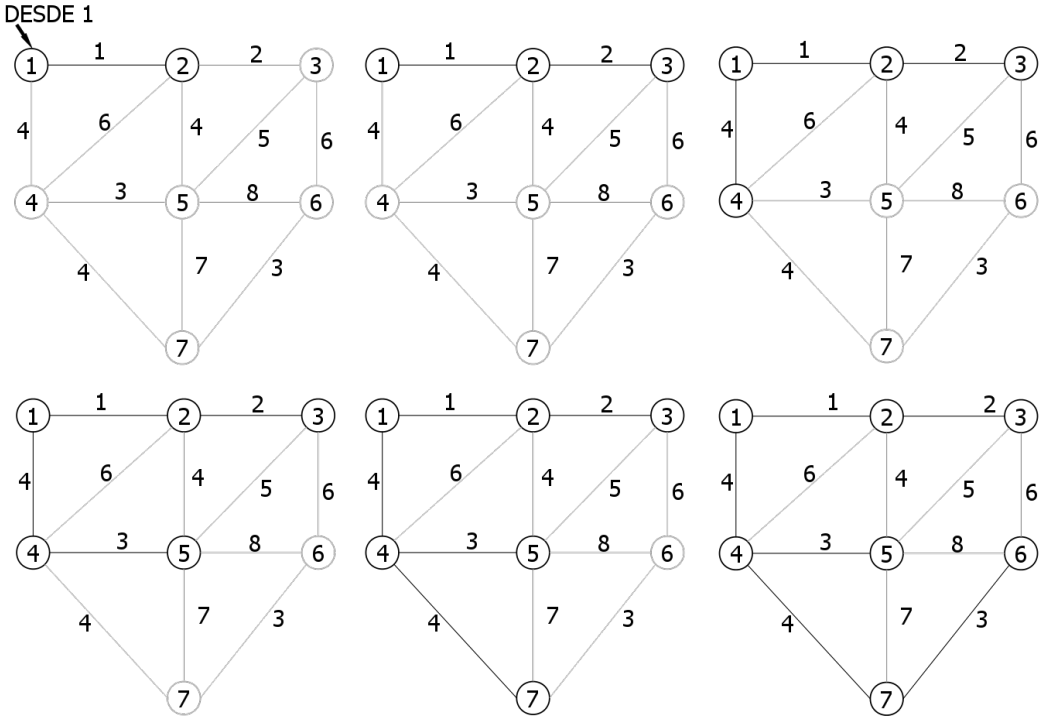


Figura 12.25: Pasos del algoritmo de Prim.

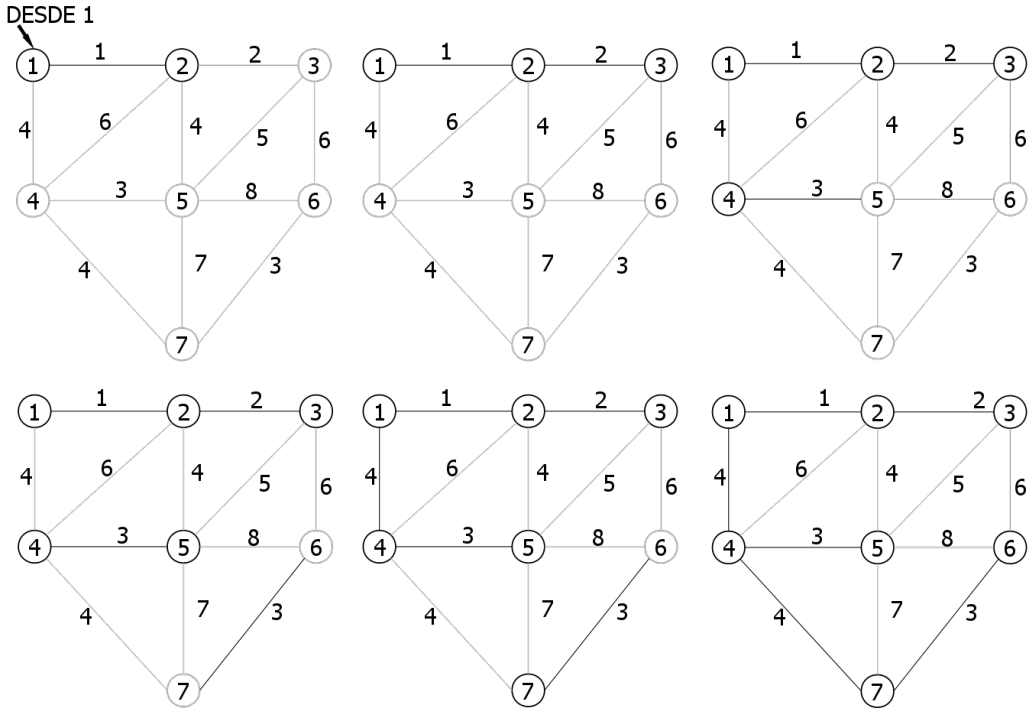


12.7.6 Algoritmo de Kruskal

Este algoritmo se basa en el concepto de componente conexas. Una componente conexa es cualquier subconjunto de vértices de un grafo G que presenta la propiedad de ser conexo. A diferencia de Prim, este algoritmo permite extraer el árbol de expansión mínima si G no es conexo. La idea es sencilla, parte del concepto que cada vértice es una componente conexa y se buscan entre los arcos aquellos permitan unir estas componentes con un costo mínimo y sin formar ciclos. Eventualmente, no se podrá reducir el número de componentes conexas y el grafo resultante será el árbol de expansión mínimo. El algoritmo sigue los siguientes pasos:

1. Copias los vertices de G a un arbol A, donde forman n componentes conexas.
2. En G elegir el arco de mínimo costo que:
   - *Que no hayan sido elegido anteriormente*
   - *Q*ue *no una dos vértices de una misma componente conexa (no forme un ciclo)*
3. Repetir paso 2 hasta que se haya unido todos los vértices

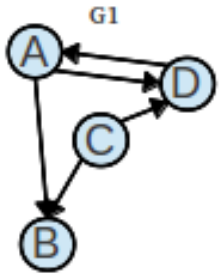
Figura 12.26: Pasos del algoritmo de Kruskal



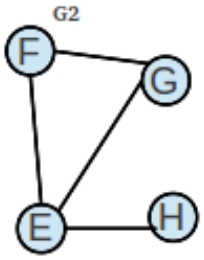
Algoritmo de Dijkstra, Edsger Wybe Dijkstra; http://users.dcc.uchile.cl/~rbaeza/inf/dijkstra.html
Algoritmo de Kruskal, Joseph Kruskal: http://www-history.mcs.st-andrews.ac.uk/Biographies/Kruskal_Joseph.html
Algoritmo de Prim, Vojtěch Jarník y Robert C. Prim: http://algoritmoprim.blogspot.com/2013/04/biografia-del-autor.html
Algoritmo de Warshall, Robert Floy, Stephen Warshall y Bernard Roy:http://algoritmofloywarshall.blogspot.com/2013/04/biografia-de-sus-creadores.html

12.8 Preguntas

1. **Deducir el conjunto de vértices (V) y arcos (A) del grafo G1.**
   V={ }
   A={ }



2. **Deducir el conjunto de vertices (V) y arcos (A) del grafo G2.**
   V={ }
   A={ }



3. **Según el tipo de arcos G1 es un grafo:**
   - Dirigido
   - No dirigido
   - Conexo
   - Dirigido valorado

4. **Según el tipo de arcos G2 es un grafo:**
   - Dirigido
   - No dirigido
   - Conexo
   - Dirigido valorado

5. **El vertice con mayor grado de salida en G1 es: _____**

6. **El vertice con menor grado en G2 es: _____**

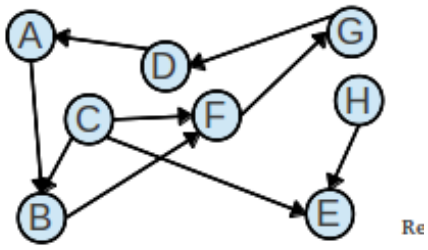
7. **Escriba la matriz de adyacencia de G1:**

8. **Escriba la matriz de adyacencia de G2:**

9. **Qué se puede decir con respecto a la conectividad de G1 y G2 , seleccione la opción que ud. considera verdadera:**
   - G1 y G2 son conexos.

- G1 es fuertemente conexo y G2 conexo.
- G1 es conexo y G2 no es conexo.
- El concepto de conectividad no se aplica.

10. **Del siguiente grafo calcule su recorrido en anchura partiendo del nodo A.**



Recorrido={ }

11. **Un general a cargo de una flota de barcos de guerra planea invadir un archipielago, si entre cada par de islas el general es capaz de estimar el número de bajas debido a los combates, que algoritmo deberia utilizar para llevar crear un plan de invasión que evite el major número de bajas posibles en su ejercito.**
    - Camino mínimo.
    - Árbol de expansion mínima.
    - Recorrido por anchura.
    - Recorrido por profundidad.
    - Deducir la matriz de caminos.

12. **Un corrector ortográfico contiene una grafo con palabras válidas, dos palabras son adyacentes si difieren en un caracter. Cada vez que un usuario del procesador de texto escribe una palabra el corrector sugiere 5 posibles correcciones si no encuentra la palabra en su lista. La primera sugerencia es aquella palabra que comparte el mayor numero de caracteres iguales y en la misma posición que la palabra escrita por el usuario. Que algoritmo debería usar el corrector obtener para hacer, de ser posible, las 4 sugerencias adicionales:**
    - Camino mínimo.
    - Matriz de caminos.
    - Recorrido en anchura.
    - Recorrido en profundidad.
    - Árbol de expansion mínimo.

13. **En base a informacion topográfica que envía una sonda espacial, un robot móvil que navega por la superficie de marte debe de planear sus próximos movimientos para llegar de un crater a otros cráteres ubicados a diferentes distancias y con diferentes tipos de caracteristicas topograficas. Que algoritmo deberia usar el robot para deducir que viajes son posibles realizar y cuales nó, sabiendo que sus ruedas no permiten subir en planos inclinados de más de 50 grados.**
    - Dijkstra.
    - Floyd.
    - Kruskal.
    - Prim.

- Warshall.

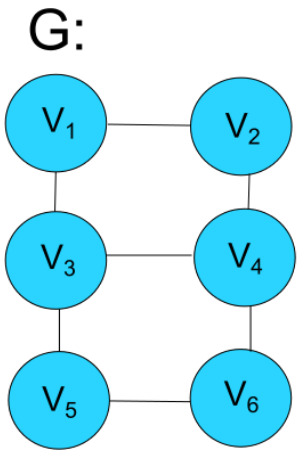
14. **Un corredor de bolsa, tiene que invertir 500 USD en la compra de N tipos de acciones durante 5 dias a razón de 100 USD por dia. El riesgo de perder el dinero invertido varía a diario. Usando datos de los 5 últimos dias él construye un grafo que representa el dinero que le queda despues de cada inversion y el riesgo de invertir los proximos 100 USD por cada tipo de acción al siguiente día. Cual sería el algoritmo que deberia usar el corredor para decubir en que acciones debería haber invertido su dinero cada día minimizando el riesgo de perder todo el dinero.**
    - Dijkstra.
    - Floyd.
    - Kruskal.
    - Prim.
    - Warshall.

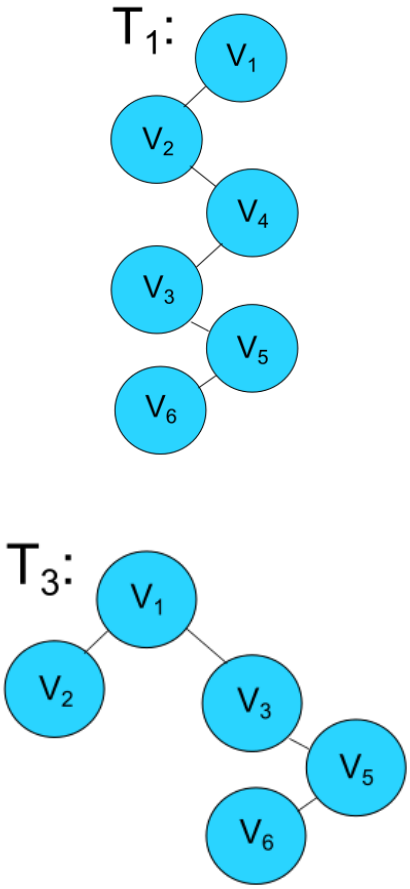
15. **Suponga que usted es el dueño de Facebook y representa cada usuario como un nodo en un grafo y cada amigo de un usuatio con un arco entre nodos. Si un nuevo usuario se crea en facebook, en base a la invitacion de un usuario existente, que algoritmo utilizaria para sugerir potenciales amigos para el nuevo usuario, en base a las amistades del usuario que hizo al invitacion.**
    - Matriz de caminos.
    - Recorrido en anchura.
    - Recorrido en profundidad.
    - Árbol de expansion mínimo.
    - Camino mínimo.

12.9 Ejercicios

EJERCICIO 1

Un árbol T es un árbol de expansión de un Grafo G si T es un subgrafo de G que contiene todos los vértices de G.





T1 y T2 son arboles de expansión del grafo G.T3 no es un árbol de expansión del grafo G. Realizar una función cuyo prototipo es int EsArbolDeExpansion(Grafo T,Grafo G) que devuelve 1 si un grafo T pertenece al conjunto de árbol de expansión o abarcador de un grafo G, devuelve 0 en caso contrario. Considere que solo puede ser generador arboles de expansión de tipo binario a partir del grafo G.

Solución:
Antes de pensar en la solución del problema, es necesario reforzar algunos conocimientos teóricos. Considere lo siguiente:
Sea G=(V,E) , es decir un grafo G que esta definido por un conjunto de vértices V y un conjunto de arcos E.
Sea G1=(V1,E1). G1 es un subgrafo de G si:
En otras palabras, G1 sera un subgrafo de G si existe al menos un par de vértices en V1 y el conjunto de arcos E1 es un subconjunto del conjunto de arcos que definen a G. Note que esta implícito el hecho de que V1 también debe ser un subconjunto de V.

Si V1=V entonces podemos afirmar que G1 es un árbol expansión de G.
Analizando los ejemplos que nos plantea el problema podemos ver que el grafo T es un subgrafo de G, y todos los vértices definidos en G, también forman parte de T. En conclusión T es un árbol de expansión del grafo G.

Ahora al analizar el grafo T3, vemos que también es un subgrafo de G pero no todos

los vértices de G forman parte de T3, falta el vértice V4. Entonces T3 no es un árbol expansión de G.

Entonces para implementar la función que se nos solicita, debemos asegurarnos de dos cosas: Primero que todos los vértices del grafo G también están en el grafo T. Y también que los arcos que conforman al grafo T están contenidos en G.
Lo primero que haremos es encolar todos los vértices del grafo G en la cola N1 (queue_grafo) . Desencolamos el primer vértice de la cola N1, lo referenciamos con el puntero v_graph, buscamos si existe una ocurrencia de este vértice en el grafo T. Si es que no existe quiere decir que ese vértice definido en G no se encuentra en T, entonces no sirve de nada continuar ya que T no sera un árbol recubridor.
Si es que si existe, guardamos esa referencia en la variable v_tree. Usamos esta variable para encolar todos los arcos incidentes a v_tree en la cola N2(queue_tree). En resumen, tenemos una cola N2 que almacena todos los arcos incidentes a v_tree.
Como ya hemos dicho para que T sea un árbol recubridor de G, todos los arcos de v_tree deben estar contenidos en el conjunto de arcos de v_graph. Lo que haremos a continuación es ir removiendo cada arco de la cola N2 y buscarlo en el conjunto de arcos de v_tree. Recuerde que si es que no se encuentra algún arco T no sera un árbol recubridor de G. Si es que si los encuentra a todos procedemos a analizar el siguiente elemento de la cola N1.
Hacemos esto hasta que la cola N1 este vacía. Al final la función retornara uno o cero según el caso.
El código a continuación muestra la implementación del algoritmo que hemos descrito. Observe que nunca se detalla que tipo de dato contienen los vértices, así que como ya hemos visto en ejemplos anteriores haremos que nuestra función sea lo mas genérica posible, por eso se ha agregado al conjunto de parámetros una función CallBack que compara el contenido de los vértices.

Código:

```
int EsArbolDeExpansion(Graph *arbol, Graph *grafo,cmpfn compararVertices) {
    int flag = 1;
    Queue*queue_grafo = queueNew(); //cola N1
    Queue*queue_tree = queueNew(); //cola N2
    NodeList*p_grafo,*p,*q;
    NodeList*it = listGetHeader(grafo);
    Gvertex*v_graph,*v;
    GVertex*v_tree;
    GEdge*arco;
    while(it!=NULL){ //encolar todos los vertices del grafo
        v= nodeListGetCont(it);
        queueEnqueue(queue_grafo, nodeListNew(v));
        it=nodeListGetNext(it);
    }
    while (!queueIsEmpty(queue_grafo)) {
        p_grafo = queueDequeue(queue_grafo);
        v_graph = nodeListGetCont(p_grafo);
        v_tree=graphSearchVertex(arbol,v_graph->content,compararVertices);
            //busca el contenido del vertice v_graph en el grafo
            arbol
        if(v_tree!=NULL){
            for(p=listGetHeader(v_tree->LAdjacents);p!=NULL;p=
                nodeListGetNext(p)){  //encola los arcos de
                v_tree
                arco=nodeListGetCont(p);
```

```
                queueEnqueue(queue_tree,nodeListNew(arco));
            }
            while(!queueIsEmpty(queue_tree)){
                q=queueDequeue(queue_tree);
                arco=nodeListGetCont(q);
                GVertex*destino_arbol=gEdgeGetDestination(arco);
                GVertex*tmp=NULL;
                //busqueda del destino arbol en el grafo g
                for(q=listGetHeader(v_graph->LAdjacents);q!=NULL;q=
                    nodeListGetNext(q)){
                    GEdge*e=nodeListGetCont(q);
                    GVertex*destino_grafo=gEdgeGetDestination(e->destination
                        );
                    if(compararVertices(destino_grafo->content,destino_arbol
                        ->content)==0){
                        tmp=destino_grafo;
                    }
                }
                if(tmp==NULL){ //si nunca lo encuentra se cambia el valor de la bandera y se rompe el
                    lazo ya que no es necesario seguir iterando
                    flag=0;
                    break;
                }
            }
        }else{ //si no lo encuentra se cambia el valor de la bandera y se rompe el lazo ya que no es necesario
            seguir iterando
            flag=0;
            break;
        }
    }
    return flag;
}
```

EJERCICIO 2

La red telefónica del país enlaza las provincias, asignando un costo a las conexiones directas entre ellas. Una conexión directa establece una relación bidireccional entre las dos provincias. Por ejemplo: Llamar de la provincia X a la provincia Y cuesta $0.5 el minuto, y de la provincia Y a la provincia X, cuesta $0,9.
Cada provincia esta identificada por un código único(numero entre 01 y 09). Por cada provincia, se maneja un conjunto de los números existentes en la misma.
- Defina los TDAs que participan en este problema.
- Escriba una función que dada la Red telefónica del país y dos números telefónicos retorne el costo mas bajo de la llamada por minuto, o -1 si no existe forma de comunicarse entre esos números. Recuerde explotar las funciones definidas en la librería.
Ejemplo:
042387690 y 08909987
Debería ubicar el costo mas bajo para comunicar a la provincia de código 04 con la de código 08. Ademas, es necesario saber si el numero 2387690 pertenece a la provincia 04 y 909987 pertenece a la provincia 08. Suponga que el costo de llamar de la provincia del código 04 a la del código 08 sea $0.4 el minuto y llamar de la provincia del código 08 a la del código 04 cuesta $0.9 el minuto. Su función debe retornar la tarifa con el costo mas bajo, este caso $0.4.
Asuma que dispone de las siguientes funciones : int obtenerCodigo(int numero_telefono); que dado un numero separa los dos primeros dígitos que representan el código de

una provincia y int obtenerNumero(int numero_telefono) que separa los dígitos que conforman el numero telefónico.

Solución:
La red telefónica del país es un conjunto de provincias que están conectadas entre si. Si dos provincias están conectadas esto supone que se podrán realizar llamadas desde la una a la otra. Esta red puede ser modelada como un grafo no dirigido, con pesos definidos entre cada arco incidente en un par de vértices. Los pesos serán la tarifa que representa por llamar de una provincia a otra. Observe que definimos el grafo como no dirigido ya que el texto menciona que una conexión entre dos provincias siempre sera en ambos sentidos.
Recordemos que en la librería del curso, se ha implementado los pesos de los arcos como un valor de tipo int. En este ejemplo se especifica que los pesos deben ser de tipo float. No nos fijaremos en este detalle, asumiremos que los arcos de la Red serán de tipo entero, así que nuestra función debe retornar un entero.
Cada vértice del grafo es una provincia en la Red. Una provincia almacena como información un código que la identifica , esto lo podemos representar como una variable de tipo int. Ademas la provincia tiene un conjunto de números telefónicos existentes en ella, usaremos una lista enlazada que almacena enteros para modelar este campo.
Para implementar la función que se nos pide, vamos a elaborar otras pequeñas funciones que nos hará resolver el problema de forma mas fácil. El prototipo sera el siguiente: int costoMasBajo(Graph*RedTelefonica,int num_uno,int num_dos);
En ninguna parte del texto se nos pide construir la Red Telefónica, mucho menos generar la información de las provincias.
Lo primero que debemos hacer es que a partir de los números recibidos por la función separar el código de la provincia y el numero de teléfono. Esto lo haremos con la ayuda de las funciones que nos han proporcionado.
Con el código obtenido debemos buscar a que provincia pertenece. Este trabajo lo realizara la función: GVertex*buscarProvinciaPorCodigo(Graph*RedTelefonica,int code)
Cuando ya obtengamos la referencia a la provincia correcta, ahora debemos buscar que el numero de teléfono ingresado este en la lista de números almacenados en esa provincia. Si no se encuentra, retornaremos un mensaje notificando dicho evento. La funcion: NodeList *buscarNumeroDeProvincia(GVertex*vertice,int numero); resuelve el problema descrito.
Observe que la implementación de las funciones anteriores son muy parecidos a problemas que ya nos hemos encontrado en otros ejercicios.
Teniendo ya las referencias a los vértices que almacenan las provincias implicadas, ahora buscaremos los dos arcos (ida y vuelta) que las conectan. Esto es resuelto fácilmente por la función que disponemos en la librería: GEdge *graphGetLink(GVertex *source,GVertex *destination); que nos retorna el arco desde un vértice origen a un destino.
Ahora debemos verificar cual de los dos arcos tiene el menor peso, este valor retornaremos.

Código:

```
GVertex*buscarProvinciaPorCodigo(Graph*RedTelefonica,int code){
    NodeList*it=NULL;
    GVertex*vertex=NULL,*v=NULL;
    Provincia*p;
    for(it=listGetHeader(RedTelefonica);it!=NULL;it=nodeListGetNext(it)){
        vertex=nodeListGetCont(it);
        p=(Provincia*)gVertexGetContent(vertex);//cast
        if(p->codigo==code){//si la provincia p tiene el mismo código que el que recibimos por
            parámetro entonces hemos encontrado el vertice objetivo
            v=vertex;
```

```
            break; //rompemos el lazo por que no hace falta seguir buscando
        }
    }
    return v;
}
```

Hemos decidido implementar esta función para que usted tenga muy claro como se realiza este proceso de búsqueda de un elemento dentro de un conjunto de datos, el cual se denomina búsqueda secuencial, ya existe una función en la librería con el prototipo GVertex *graphSearchVertex(Graph *G, Generic cont, cmpfn cmp) que brinda la misma funcionalidad de buscar un vértice dentro de un grafo.

```
NodeList *buscarNumeroDeProvincia(GVertex*vertice,int numero){
    Provincia*provincia=(Provincia*)gVertexGetContent(vertice);//cast
    List*lista_numeros=provincia->num_existentes;
    NodeList*n=listSearch(lista_numeros,integerNew(numero),integerCmp);//busca
        numero en la lista de la provincia
    return n;
}

int costoMasBajo(Graph*RedTelefonica,int num_uno,int num_dos){
    int codigo=obtenerCodigo(num_uno);
    int numero=obtenerNumero(num_uno);
    int codigo_dos=obtenerCodigo(num_dos);
    int numero_dos=obtenerNumero(num_dos); int min;
    GVertex*vertice_uno=buscarProvinciaPorCodigo(RedTelefonica,codigo);
    GVertex*vertice_dos=buscarProvinciaPorCodigo(RedTelefonica,codigo_dos);

    NodeList*n=buscarNumeroDeProvincia(vertice_uno,numero);
    NodeList*n_dos=buscarNumeroDeProvincia(vertice_dos,numero_dos);

    if(n==NULL || n_dos==NULL){
        printf("Numero␣Telef nico␣no␣encontrado");
        return 0;
    }else{
        GEdge*arco_uno=graphGetLink(vertice_uno,vertice_dos);
        GEdge*arco_dos=graphGetLink(vertice_dos,vertice_uno);
        if(arco_uno==NULL || arco_dos==NULL ){//si no existe conexión entre las dos
            provincias implicadas se retornara-1
            return -1;
        }else{
        int peso_uno=arco_uno->weight;
        int peso_dos=arco_dos->weight;
        min=peso_uno;
            if(peso_dos<peso_uno){
                min=peso_dos;
            }
        }
        return min;
    }
}
```

Edición: Marzo de 2014.
Este texto forma parte de la Iniciativa Latinoamericana de Libros de Texto abiertos (LATIn), proyecto financiado por la Unión Europea en el marco de su Programa ALFA III EuropeAid.