UNIVERSIDAD AUTÓNOMA DE BAJA CALIFORNIA
FACULTAD DE CIENCIAS QUÍMICAS E INGENIERÍA

APUNTES DE ALGORITMOS Y ESTRUCTURA DE DATOS

ELABORADOS POR

M.I ALMA LETICIA PALACIOS GUERRERO

UNIDAD 1 Introducción a los Algoritmos y Estructuras de Datos

1.1 Definición de Algoritmo

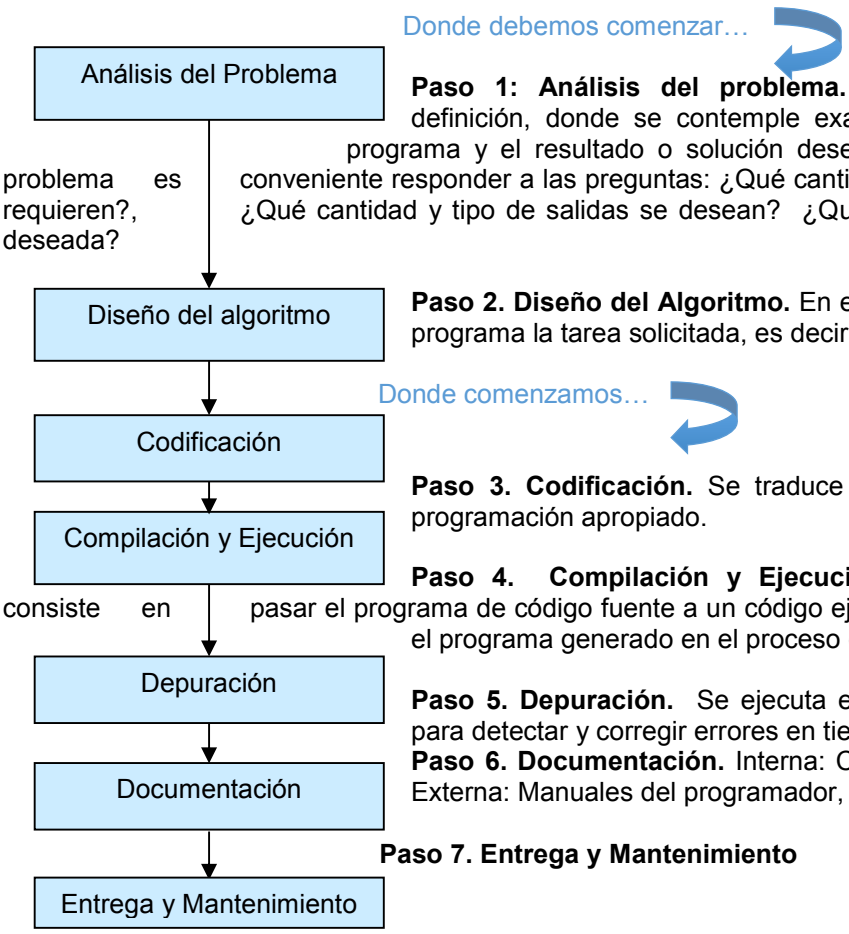
Algoritmo se define como un conjunto de instrucciones que la computadora debe seguir para resolver un problema. La palabra **algoritmo** se deriva de la traducción al latín del nombre Muhammad Musa **Al-khawarizmi,** un matemático y astrónomo árabe que en el siglo IX escribió un tratado sobre manipulación de números y ecuaciones.

1.1.1 Características de los Algoritmos

- Son independientes del lenguaje de programación a utilizar.
- Sencillo, los pasos deben ser claros y bien definidos.
- Precisos, cada vez que se ejecutan con las mismas entradas se obtiene el mismo resultado.
- Definidos, indican claramente el orden de realización paso a paso. Finitos, tienen un número de pasos contables.

1.2 Modelo de Cascada para el desarrollo de Sistemas

La resolución de un problema con una computadora conduce a la escritura de un programa y a su ejecución en la misma. Aunque el proceso de diseñar programas es un proceso creativo, los pasos a seguir se establecen en varios modelos para el desarrollo de sistemas uno de ellos es el modelo de cascada. Este se define como el proceso que se sigue para diseñar, construir, entregar y hacer evolucionar el software, desde la concepción de una idea hasta la entrega y el retiro del sistema[1].



Paso 1: Análisis del problema. Esta fase requiere una clara definición, donde se contemple exactamente lo que debe hacer el programa y el resultado o solución deseada. Para poder definir bien un conveniente responder a las preguntas: ¿Qué cantidad y tipo de datos de entrada se ¿Qué cantidad y tipo de salidas se desean? ¿Qué método(s) produce(n) la salida

Paso 2. Diseño del Algoritmo. En esta etapa se define como hace el programa la tarea solicitada, es decir, se define el algoritmo.

Paso 3. Codificación. Se traduce el algoritmo en un lenguaje de programación apropiado.

Paso 4. Compilación y Ejecución. El proceso de compilación pasar el programa de código fuente a un código ejecutable. La ejecución es "correr" el programa generado en el proceso de compilación.

Paso 5. Depuración. Se ejecuta el programa con datos de prueba para detectar y corregir errores en tiempo de ejecución.

Paso 6. Documentación. Interna: Comentarios dentro del programa. Externa: Manuales del programador, del usuario, de instalación, etc.

Paso 7. Entrega y Mantenimiento

1.3 Definición de Estructura de Datos.

En el desarrollo de programas, existe una fase previa a la escritura del programa, esta es el diseño del algoritmo que conducirá a la solución del problema, en esta fase también deberá considerarse la estructura de datos que se va a utilizar. **Estructura de datos:** Conjunto de variables agrupadas y organizadas de cierta forma para representar un comportamiento. El término **estructura de datos** se refiere a la forma en que los datos están organizados dentro de un programa. La correcta organización de datos puede conducir a algoritmos más simples y más eficientes.

1.3.1 Clasificación de las estructuras de datos según su tamaño.

Las estructuras de datos según su tamaño en memoria, se clasifican en:

Estructuras de datos estáticas: Son aquellas estructuras cuyo tamaño en memoria está definido al iniciar y no cambia durante la ejecución.

Estructuras de datos dinámicas: Son las estructuras que permiten variar su tamaño en memoria de acuerdo a las necesidades del ambiente o del usuario.

Clasificación	Características	Implementación	Ejemplos
Estructuras Estáticas	Tienen un tamaño predefinido y no varían durante la ejecución	Se implementa con arreglos	Juegos, Tableros
Estructuras Dinámicas	El tamaño se adapta a las necesidades del usuario	Se implementa con listas enlazadas	Navegadores, Editores de texto, Playlist

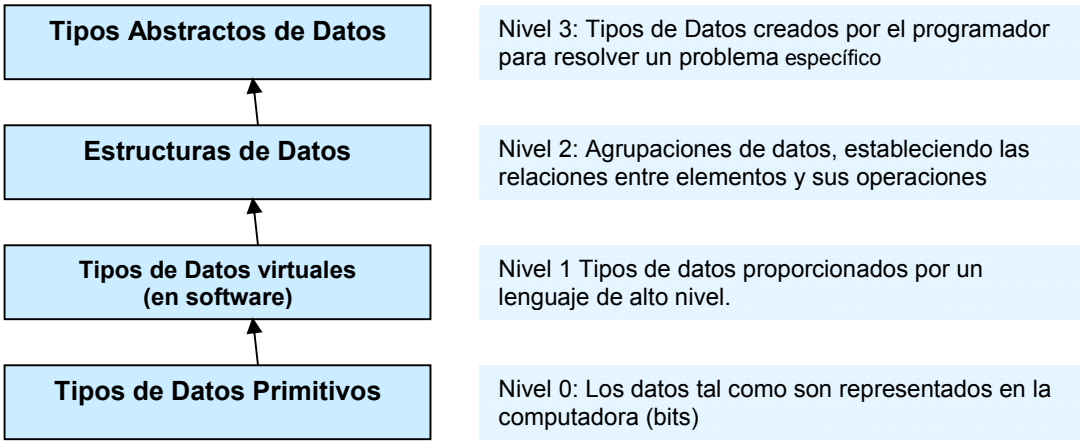
1.4 Abstracción

Desde el desarrollo de la informática los programadores han luchado con el problema de la complejidad inherente al software, una de las herramientas utilizadas para resolverlo es la **abstracción**. Esta se puede definir como la consideración aislada de las cualidades esenciales de un objeto. En otro término la abstracción es la capacidad para encapsular y aislar la información del diseño y ejecución.

1.5.1 Tipo Abstracto de Datos (TAD)

Hacia 1974, **John Guttag** propone el concepto de tipo abstracto de datos, el cual constituye el máximo nivel de abstracción. Un tipo abstracto de datos es un modelo (estructura) con un número de operaciones que afectan a ese modelo. Un TAD es un tipo de datos definido por el programador que se puede manipular de un modo similar a los tipos de datos definidos por el sistema.

Niveles de Abstracción



1.5.3 Diferencia entre Estructura de Datos y Tipo Abstracto de Datos.

En los lenguajes como Pascal y C, es posible crear una estructura y funciones para realizar operaciones sobre dicha estructura, pero no es posible asociar la estructura y las funciones como una unidad, o sea no es posible asociarlos como un TAD. Dependiendo del lenguaje en que el TAD se esté implementando recibe un nombre diferente, por ejemplo, en Java y C++ a un TAD se le conoce como clase mientras que en Pascal es una unidad y en ADA es un paquete.

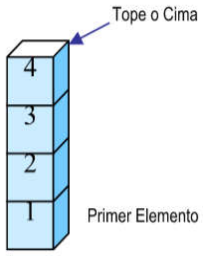
1.5.3.1 Ventajas de los TAD

- Mejora la representación de la realidad.
- Mejora la robustez del sistema ya que permite comprobar tipos para evitar errores de tipo en tiempo de ejecución.
- El conocimiento de los objetos permite optimización de tiempo de compilación.

UNIDAD 2 Pilas, Colas y Listas

2.1 Pilas

Una pila es un contenedor de datos cuyo comportamiento está regido por el principio LIFO (Last Input First Output), es decir, el último elemento en entrar a la pila es el primer en salir. En este tipo de estructura los datos o elementos pueden insertarse y suprimirse elementos solamente por un extremo, llamado tope o cima.



2.1.2 Aplicaciones

Los navegadores de Internet almacenan las direcciones visitadas recientemente. Cada vez que el usuario visita una página, su dirección es almacenada en una pila, de forma que cada vez que el usuario hace click en back se retira el último elemento insertado en la pila, esto es, se muestra en pantalla la última página visitada.

2.1.3 Operaciones con pilas

Las operaciones que se manejan para el uso de las pilas son las siguientes:

Básicas

- **pop():** Regresa el último elemento insertado en la pila
- **push():** inserta un elemento en el tope de la pila.

Auxiliares

- **llena():** Regresa verdadero si la pila está llena.
- **vacia():** Regresa verdadero si la pila está vacía.
- **size():** Regresa el tope de la pila.
- **vaciar():** Elimina todos los elementos de la pila

2.1.4 Implementación de Pila Estática en Java

```
import java.io.*;
import java.lang.*;

public class Pila
{           public static final int CAPACIDAD=1000;
            int capacidad;
            private int elementos[];
            int tope;

           public Pila() {this(CAPACIDAD);}  /*Constructores*/

            public Pila(int cap) {capacidad=cap;
                                  elementos=new int[cap];
                                  tope=0;
           }

            public int tamano(){
            return (tope);
            }

            public boolean vacia()       {
                        return (tope==0) ? true : false;
           }

            public boolean  llena() {
                        return (tope==capacidad)? true : false;
            }

            public void push(int objeto){

                                  Teclado x=new Teclado();
                                  try {
           if (!llena())    {
                                                elementos[tope]=objeto;
                                                tope++;
                                          }
                                          else{
                                                System.out.println("Pila Llena");
                                                   x.espera();
                                          }
                                  }

                                 catch (java.lang.ArrayIndexOutOfBoundsException Error) {
                                          System.out.println("Desborde de Pila");
                                  }
                     }
            public int pop() {
                        return(elementos[--tope]);
           }
```

```
public class pruebaPila{

        public static void main(String arg[]) throws IOException{
                                Pila pila=new Pila(3);
                            int obj1,opcion;
                                 Integer cadena=new Integer(0);
                                 Teclado keyboard = new Teclado();

                                do{
                                  Teclado.clrscr();
                                  System.out.println("1) Ver tamaño de Pila");
                                  System.out.println("2) Meter dato a la Pila");
                                  System.out.println("3) Sacar dato de la Pila");
                                  System.out.println("4) Fin de Programa");
                                  System.out.println("Opcion:  ");
                                  opcion=cadena.parseInt(keyboard.leer() );
                                  switch(opcion){

                                   case 1: System.out.println("La pila tiene "+ pila.tamano() + "
                                                    elementos");
                                                  keyboard.espera();
                                                  break;
                                          case 2:  System.out.println("Escribe un numero: ");
                                                  pila.push(cadena.parseInt(keyboard.leer()) );
                                                  break;
                                          case 3: if (!pila.vacia())
                                                          System.out.println("Dato sacado " + pila.pop() );
                                                   else
                                                          System.out.println("Pila Vacia");
                                                  keyboard.espera();
                                                  break;
                                          case 4:   break;
                                    }

                                } while(opcion!=4);

                            }
        }// fin de PruebaPila
//Fin de Clase Pila
}
```

2.1.5 Código pila estática en C

```
#include<conio.h>
#include<stdio.h>
#define max 3

struct Pila
{ int tope;
  int dato[max];
};

typedef struct Pila tipoPila;

void push (tipoPila * pila, int x)
{ pila->dato[pila->tope++]=x;}
int pop (struct Pila * pila)
{return(pila->dato[--pila->tope]);}

int vacia(struct Pila * pila)
{if (pila->tope==0) return (1); else return(0);}

int llena(struct Pila * pila)
{ if (pila->tope==max) return(1); else return(0);}

void main()
{  struct Pila pila={0};
  char op;
  int num;
  do
  { clrscr();
    puts("1) Meter un elemento");
    puts("2) Sacar un elemento");
    puts("3) Tamaºo de la Pila");
    puts("4) Salida");
    printf("Opcion:  "); op=getche();

    switch(op)
    { case '1':  if(!llena(&pila) )
                 { printf("Que dato desea ingresar");
                   scanf("%d",&num);
                   push(&pila,num);
                 }
                 else
                 {  printf("Pila LLena");
                    getch();
                 }
                 break;
      case '2':  if(!vacia(&pila) )
                   printf(" %d",pop(&pila));
                 else
                   printf("La pila esta vacia");
                 getch();
                 break;
      case '3':  printf("\nHay %d datos ",pila.tope);getch();
      case '4': break;
  }
}while(op!='4');
}
```

Estructuras de datos Dinámicas

La implementación de pilas y colas usando arreglos es simple, pero tiene la desventaja de ser poco útil en algunos casos debido a que el tamaño de la estructura debe decidirse antes de empezar a utilizar la estructura. Sin embargo, hay otras formas de implementar tales estructuras sin las limitaciones de los arreglos. La solución es implementar las estructuras de datos en forma dinámica, usando apuntadores; de esta manera no existe un límite de elementos ya que el uso de apuntadores se supone una memoria infinita. Antes de implementar estructuras de datos dinámicas en C, se establecerán los conceptos básicos para la manipulación de apuntadores en C.

Conceptos Básicos de Apuntadores en C

Apuntador es una variable que contiene una dirección de memoria. Generalmente esa dirección es la dirección de otra variable de memoria. Si una variable contiene la dirección de otra variable, entonces se dice que la primera apunta a la segunda

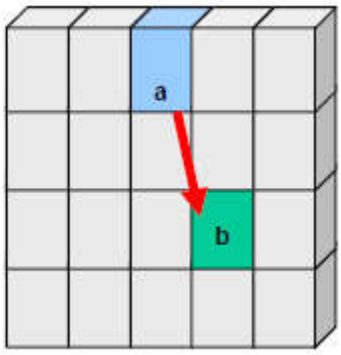
Declaración: **tipo * nombre;**

El tipo del apuntador determina el tipo de dirección variables a las que puede apuntar el apuntador. Por ejemplo: Supongamos que existen la variable a y el apuntador b.

```
int * a;
int b=5;
```

Ahora digamos que las variables a y b están ubicadas en las direcciones 100A y ABC0
Si se ejecuta la línea: **a=&b;**

Entonces la celda 100A contendrá la dirección de b, es decir, el apuntador a “apuntará” a b .



Los apuntadores se utilizan en situaciones en las que pasar valores es difícil o indeseable, otras razones son las siguientes:

- Para regresar más de un valor de una función
- Pasar arreglos fácilmente de una función a otra.
- Manipular arreglos más fácilmente, manipulando punteros a ellos, o a partes de ellos, en vez de mover los arreglos en sí.
- Crear estructuras de datos complejas, como son las listas encadenadas y árboles binarios, donde una estructura de datos debe tener referencias a otras estructuras de datos.
- Para comunicar información acerca de la memoria.

Operadores para apuntadores

& Devuelve la dirección de memoria de su operando. Por ejemplo, en la línea m=&cuenta; m guarda la dirección de la variable cuenta.

***** Devuelve el valor de la variable localizada en la dirección que tiene la variable que sigue al operador.

Por ejemplo: Si q* m y a m le asignamos un valor de 100 entonces en q tendremos un valor de 100.

Ejemplo No. 1

```
# include <conio.h>
#include <stdio.h>

void main()
{           int * direccion;   // Declaración del apuntador
            int edad;
            clrscr();
            printf("cuantos años tienes");  scanf("%d",&edad);
            direccion=&edad;
            printf ("Tu edad esta guardada en la posición de memoria %p.", direccion);
            printf("\nTu edad es : %d ", *direccion);
            getch();
}
```

Ejemplo No. 2 Manejo de un vector con apuntadores

```
#include<conio.h>
#include<stdio.h>

void imprimir(int * v)
{ int i;
  i=0;
  do
  { printf("%d\n",*(v+i) );
    i++;
  }while(i<5);
  getch();
}

void main()
{ int vector[5]={3,0,2,2,4};
  clrscr();
  imprimir(vector);
}
```

Ejemplo No. 3 Implementación de un vector usando memoria dinámica

```
#include<conio.h>
#include<stdio.h>
#include<alloc.h>

void main()
{ int *inicio, *aux,i,j;
  char usuario;

  inicio=(int *) malloc(sizeof(int) );
  i=0;
  aux=inicio;
  clrscr();
  do
  { printf("\nEscriba un valor para la posicion %p: ",aux);
    scanf("%d",aux);
    (aux)++;
    i++;
    printf("\nOtro numero");
    usuario=getch();
  }while(usuario!='n');

  j=0;
  aux=inicio;
  clrscr();
  while(j<i)
  {
     printf("%d esta guardado en la posicion %p\n",*aux,aux);
     j++;
     (aux)++;
  }
  getch();
}
```

2.1.6 Implementación de una pila usando una lista enlazada simple y apuntadores simples

// Pila01.c

```
#include <alloc.h>

struct nodo
{int dato;
 struct nodo * ant;
};

int vacia(struct nodo *tope) { return (!tope)? 1:0; }

struct nodo * crearnodo(int dato)
{  struct nodo* p;
   p=(struct nodo*)malloc (sizeof (struct nodo));
   p->dato=dato;
   p->ant=NULL;
   return(p);
}


struct nodo* push(struct nodo * tope,int dato)
{ struct nodo *aux;
   aux=crearnodo(dato);
   aux->ant=tope;
   tope=aux;
   return(tope);
}

void mostrar(struct nodo *tope)
{ struct nodo *aux;
  clrscr();
 if(!tope)  printf("Esta vacia");
 else
 { aux=tope;
   do{
     printf("\n%d",aux->dato);
     aux=aux->ant;
   }while(aux!= NULL);
  }
getch();
}




struct nodo* pop(struct nodo *tope,int* valor)
{ struct nodo *aux;
  int dato;
  aux=tope;
  *valor=tope->dato;
  tope=tope->ant;
  free(aux);
  return(tope);
}









void main()
{struct nodo *tope=NULL;
 char opc;
 int dato,ban;
  do{

                gotoxy(27,8);  printf(" 1. Push");
                gotoxy(27,10); printf(" 2. Pop");
                gotoxy(27,12); printf(" 3. Mostrar Pila");
                gotoxy(27,14); printf(" 4. Salir");
                gotoxy(27,16); printf("Opcion: [  ]\b\b");
                opc=getchar();
                switch(opc)
                { case '1':clrscr();
                                printf("Escribe dato:  ");
                                scanf("%d",&dato);
                                if(tope==NULL)
                                  tope=crearnodo(dato);
                                else
                                  tope=push(tope,dato);
                                break;
                  case '2': clrscr();
                                  if(!vacia(tope)) {
                               tope=pop(tope,&dato);
                               printf("El dato en la cima es: %d",dato);
                                   }
                                  else printf("Pila Vacia");
                                  getch();              break;
                  case '3': if (!vacia(tope) )
                                          mostrar(tope);
                                 else printf("Pila Vacia");
                                 break;
                  case '4': break;
                }
      }while(opc!='4');
}
```

2.1.7 Implementación de una pila usando una lista enlazada simple y apuntadores dobles

```
/* Pila2.c Pila Implementada sobre una lista con apuntadores dobles*/

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

struct nodo{
  int num;
  struct nodo *anterior;
};

int vacia(struct nodo* cima){ return (cima==NULL)? 1:0; }

void push(int dato, struct nodo **cima ){
    struct nodo * tempo;

    tempo= (struct nodo * ) malloc( sizeof(struct nodo) );
    tempo->num=dato;

    if(*cima==NULL)
     tempo->anterior=NULL;
    else
     tempo->anterior=*cima;
    *cima=tempo;
}


int pop(struct nodo ** cima){
    struct nodo *tempo;
    int regresa;
      tempo=*cima;
      regresa=(*cima)->num;
      *cima=(*cima)->anterior;
      free(tempo);
      return (regresa);
}

void main(){
     struct nodo * pila=NULL;
     char op;
     int  numero;
     do{clrscr();
puts("1) Push");
puts("2) Pop");
puts("3) Salida");
printf("Opcion: ");op=getch();
switch(op){
   case '1': printf("Numero: "); scanf("%d",&numero);
                    push(numero, &pila);
                    break;
   case '2': if ( vacia(pila) )
                              printf("No hay datos");
                    else{
                              numero=pop(&pila);
                              printf("El numero de la cima es %d ", numero);
                    }
                    getch();
                    break;
   case '3': break;
}
     }
     while (op!='3');
   }
```

2.1.8 Recursión

El concepto de recursión es fundamental en Matemáticas y Ciencias de la Computación. La definición más simple es que un programa recursivo es aquel que se invoca a sí mismo.3. La recursión se utiliza porque permite expresar algoritmos complejos en forma compacta y elegante sin reducir la eficiencia. Un algoritmo recursivo es aquel que resuelve un problema resolviendo una o más instancias menores que el mismo problema. Los algoritmos recursivos se implementan con métodos recursivos.

Reglas de la recursión

1) Caso base: Siempre debe existir casos base que se resuelven sin hacer uso de la recursión.

2) Progreso: Cualquier llamada recursiva debe progresar hacia un caso base.

3) Diseño: Asumir que toda llamada recursiva interna funciona correctamente.

4) Regla de Interés compuesto: Evitar duplicar el trabajo resolviendo la misma instancia de un problema en llamadas recursivas compuestas.

Ejemplo 1: Cálculo del n número de la serie fibonacci

```
int fibonacci (int n)
{
    if (n<=1)
      return 1; // Caso Base
   else

   return fibonacci(n-1)+ fibonacci(n-2);
}
```

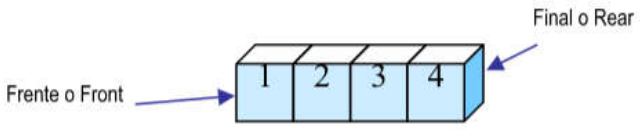
Ejemplo 2: Evaluación del factorial de n

```
public static long factorial(int n)
{
   if (n<=1)
     return 1; // Caso Base
  else
     return n* factorial(n-1);
}
```

2.2 Colas

2.2.1 Definición

Otra estructura de datos ampliamente utilizada es la cola. A la cola se le considera un primo cercano de la pila. La cola puede definirse como un contenedor de datos que funciona de acuerdo al principio FIFO (First Input First Output) porque el primer elemento que entra a la cola es el primero que sale. En una cola los datos entran por un extremo llamado final (rear) y se insertan por el otro extremo llamado frente (front). Una buena analogía de esta estructura de datos es un grupo de personas esperando en línea para entrar al cine.



2.2.2 Representación

Esta estructura de datos se usa en muchos sistemas operativos, por ejemplo Unix, para llevar el control de la ejecución de procesos, cada proceso en el sistema es almacenado en una lista y esta se va recorriendo, dándole un pequeño tiempo del microprocesador a cada proceso, durante la fracción de segundo de cada proceso este asume que tiene el control total del procesador.

2.2.3 Manejo

Las operaciones para el manejo de colas son las siguientes:

Básicas

insertar(): Agregar un elemento al final de la cola.
remover(): Remover el primer elemento de la cola.

Auxiliares

llena(): Regresa verdadero cuando la cola esta llena.
vacía(): Regresa verdadero cuando la cola esta llena.

2.2.4 Código de colas estáticas en C

```
#include<conio.h>
#include<stdio.h>
#define max 3
struct cola{  int datos[max];
              int inicio,final;};

void crearCola(struct cola * q)
{ q->inicio=-1;
  q->final=0;
}

int llena(struct cola q)
{  if (q.final==max) return (1);
   return(0);
}
int vacia(struct cola q)
{  if (q.final==0) return (1);
   return(0);
}

int remover(struct cola *q)
{ int x,i;
  x=q->datos[0];
  for(i=0;i<q->final-1;i++)
     q->datos[i]=q->datos[i+1];
  q->final--;
  return(x);
}
```

```
void insertar(struct cola *q, int dato)
{   q->datos[q->final]=dato;
    q->final++;
}

void main()
{  struct cola C;
   char op;
   int num;
   crearCola(&C);
   do
   { clrscr();  puts("1) Agregar un elemento");
    puts("2) Sacar un elemento");

    puts("3) Salida");   printf("Opcion: "); op=getche();
    clrscr();
    switch(op){
     case '1':  if(!llena(C) ){  printf("Dato:"); scanf("%d",&num);
                                 insertar(&C,num);
                              }
                              else{   printf("Cola LLena"); getch();
                              }
                              break;
     case '2':  if(!vacia(C) )   printf("Dato extraido
%d",remover(&C));
                              else
                                printf("No hay datos");
                             getch();
                             break;
     case '3': break;
    }
  }while(op!='3');
}
```

2.2.5 Código de colas estáticas en Java

```
import java.io.*;
import java.lang.*;

class ColaEstatica
{       int frente, fin, capacidad;
        private int elementos[];
        public ColaEstatica(int cap) {capacidad=cap;   elementos=new int[capacidad];     frente=-1; fin=0;}

        public boolean vacia()          {return (fin==0) ? true : false;}
        public boolean  llena() {return (fin==capacidad)? true : false;}

        public void insertar(int objeto){
                        try {       if (!llena())   {elementos[fin]=objeto;    fin++; }
                                    else{  System.out.println("Pila Llena");    new Teclado().espera(); }
                            }
                        catch (java.lang.ArrayIndexOutOfBoundsException Error) { System.out.println("Desborde de Cola"); }
        }

        public int remover() {  int x=elementos[0];
                        for(int i=0; i<fin-1;i++)
                                elementos[i]=elementos[i+1];
                        fin--;
                        return(x);
                        }
        }
        public class pruebaColaEstatica{
        public static void main(String arg[]) throws IOException{
                                ColaEstatica cola=new ColaEstatica(5);
                          int obj1,opcion;
                                Integer cadena=new Integer(0);
                                Teclado keyboard = new Teclado();
                                do{
                                  Teclado.clrscr();
                                  System.out.println("1) Insertar dato");
                                  System.out.println("2) Remover dato");
                                  System.out.println("3) Fin de Programa");
                                  System.out.println("Opcion:  "); opcion=cadena.parseInt(keyboard.leer() );
                                  switch(opcion)
                                  { case 1:  if (!cola.llena() ){System.out.println("Escribe un numero: ");
                                                cola.insertar(cadena.parseInt(keyboard.leer()) );
                                                                }
                                                                else{ System.out.println("Cola Llena");
                                                                    keyboard.espera();
                                                                }
                                                                break;
                                        case 2:if (!cola.vacia())
                                                                    System.out.println("Dato sacado " + cola.remover() );
                                                                 else
                                                                      System.out.println("Cola Vacia");
                                                                keyboard.espera();
                                                               break;
                                        case 3: break;
                                  }
                                } while(opcion!=3);
                        }
        }
```

2.4. Implementación de una Cola utilizando una lista enlazada simple en Lenguaje C

```
//Lista-00.C
#include<alloc.h>


struct nodo{
  int dato;
   struct nodo * sig;
};

struct nodo* crearNodo(int dato){
  struct nodo * p;
  p=(struct nodo*) malloc(sizeof (struct nodo) );
  p->dato=dato;
  p->sig=NULL;
  return(p);
}


struct nodo* insertar(struct nodo *inicio, int dato){
  struct nodo * p,* nuevo, *q;
   nuevo=crearNodo(dato);
   p=inicio;
   while(p!=NULL){
     q=p;
     p=p->sig;
   }
   q->sig=nuevo;
   return(inicio);
}

void mostrar(struct nodo* inicio){
   struct nodo *aux;
   clrscr();
   if (!inicio)       printf("Vacia");
   else
   { aux=inicio;
     do{ printf("\n%d", aux->dato);
           aux=aux->sig;
     }while(aux);
   }
   getch();
}
```

```
struct nodo* borrar(struct nodo *inicio,int dato){
  struct nodo *p, *q=NULL;
    p=inicio;
    while(p->dato!=dato && p!=NULL){
                q=p;
                p=p->sig;
    }
    if(q==NULL)
    { inicio=p->sig;
      free(p);
    }
    else
      if(p!=NULL)
      { q->sig=p->sig;
              free(p);
      }
      else
      { printf("No encontrado");  getch();
      }
  return inicio;
}

  void main()
  { struct nodo * inicio=NULL;
    char op;
    int x;
    do{   clrscr();
          puts("1) Insertar\n 2) Mostrar\n 3)Borrar\n 4) Salida");
          printf("Opcion:[  ]\b\b");       op=getch();
          switch(op){
             case '1':  clrscr(); printf("Dato: ");  scanf("%d",&x);
                        if(!inicio)
                          inicio=crearNodo(x);
                        else
                          inicio=insertar(inicio,x);
                        break;
              case '2': clrscr(); mostrar(inicio); break;
              case '3': clrscr(); printf("Dato: ");scanf("%d",&x);
                        inicio=borrar(inicio,x);
                        break;
      }
    }while(op!='4');
  }
```

2.5 Implementación de una Cola utilizando una lista enlazada simple en Lenguaje Java

```
import java.io.*;
import java.lang.*;

class ColaDinamica{
    private class Nodo{
                Object info;
                Nodo sig;

                Nodo(Object obj){ this.info=obj; sig=null;}
            }
    private Nodo inicio, fin;

    ColaDinamica(){inicio=null; fin=null;}

    boolean vacia(){ return(inicio==null);}

    void insertar(Object objeto){  Nodo nuevo=new Nodo(objeto);
                                    if(vacia() ) inicio=fin=nuevo;
                                    else{       fin.sig=nuevo;
                                                fin=nuevo;
                                   }
    }

   Object remover(){Object regresa=inicio.info;
                        inicio=inicio.sig;
                        return(regresa);
    }

   void mostrar(){ Teclado x=new Teclado();
                    Nodo aux=inicio;
                    while(aux!=null){ System.out.println(aux.info.toString());
                                      aux.sig;
                    }
                   x.espera();
   }
}

public class pruebaColaDinamica{
            public static void main(String arg[]) throws IOException{
                        ColaDinamica cola=new ColaDinamica();
                        int opcion=1;
                        Integer cadena=new Integer(0);
                        Teclado keyboard = new Teclado();int ban;
                        do{ System.out.println("1) Insertar"); System.out.println("2) Remover");
                            System.out.println("3) Mostrar"); System.out.println("4) Fin de Programa");
                            System.out.println("Opcion:  ");
                            do{   ban=0;
                                  try{ opcion=cadena.parseInt(keyboard.leer() );}
                                  catch(NumberFormatException exception){
                                            System.out.println("Tienes que escribir un numero");
                                             ban=1;
                                  }
                             }while(ban==1);
                             switch(opcion){ case 1:  System.out.println("Palabra?: "); cola.insertar(keyboard.leer()); break;
                                             case 2:if (!cola.vacia()){Object objeto=cola.remover();
                                                                       System.out.println("Dato sacado " + objeto.toString());
                                                                      }
                                                    else
                                                       System.out.println("Cola Vacia");
                                                    keyboard.espera();  break;
                                            case 3: cola.mostrar(); break;
                                            case 4: break;
                                            }
                        } while(opcion!=4);
                }
            }// fin de PruebaColaDinamica
```

2.6 Listas

Una lista es un contenedor secuencial en el que se pueden insertar y borrar elementos independientemente del tamaño del contenedor. La lista enlazada básica consta de una colección de nodos conectados entre sí, dichos nodos están situados en la memoria dinámica en direcciones no consecutivas. Cada nodo se compone de una sección de datos y una referencia al siguiente nodo de la lista. Los nodos típicos la estructura Lista y del ADT Lista son como los de la figura:

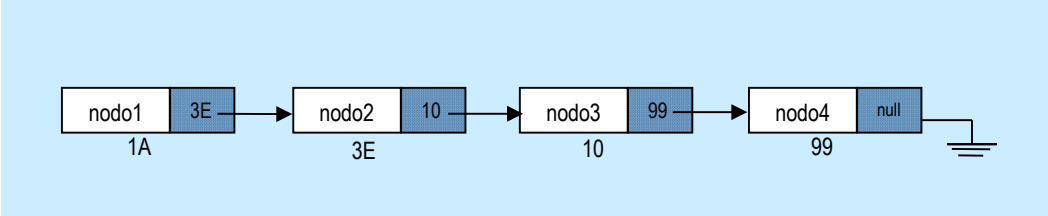
ADT Lista

```
clase Lista{

Object información;
 Nodo referencia;

 ____________

 Método 1
 Metodo 2
 Metodo N
}
```

Estructura de Datos Lista

```
struct Nodo{
int información;
struct nodo* referencia;
};
```

La siguiente figura muestra una lista simple.

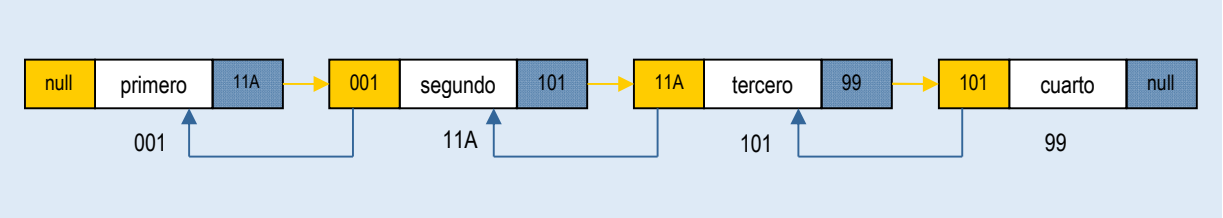


Para insertar un elemento cualquiera debemos ir recorriendo la lista, lo que pudiera hacer creer que las listas son menos prácticas que los vectores, pero estas tienen sus ventajas: una inserción en medio de la lista no requiere mover todos los elementos que se encuentran después del punto de inserción mientras que en un vector es necesario recorrer todos los elementos para abrir espacio al nuevo elemento.

Cabe hacer notar que si se permite el acceso sólo al primer elemento, entonces la lista se comporta como una pila mientras que si las inserciones se realizan sólo por el último y los accesos sólo por el inicio, entonces la lista se ha implementado para comportarse con el esquema FIFO.

2.7 Listas Doblemente Enlazadas

En una lista doblemente enlazada cada nodo tiene una referencia al nodo que le precede y al que le sigue. Al estar doblemente ligada se facilita la implementación de la búsqueda y recorridos en ambos sentidos. Al igual que en las implementaciones de otras estructuras, en la lista doblemente enlazada los nodos cabecera y final tienen referencias a null. Una buena analogía de una lista sería un tren, donde cada vagón esta conectado con el que le precede y el que le sigue.



2.7 Listas Dobles Implementación en C

```
#include <alloc.h>

struct nodo
{int dato;
 struct nodo * ant;
 struct nodo * sig;
};

struct nodo *  crearnodo(int dato)
{  struct nodo *p;
   p=(struct nodo*)malloc (sizeof (struct nodo));
   p->dato=dato;
   p->sig=NULL;
   p->ant=NULL;
  return(p);
}

struct nodo* insertar(struct nodo *inicio,int dato)
{  struct nodo *p,  *q=NULL, *nuevo;
    p=inicio;
    nuevo=crearnodo (dato);
    while(p!=NULL && dato>= p->dato  )
    {  q=p;
       p=p->sig;
    }
   printf("p=%p y q=%p",p,q); getch();
   if (q==NULL)
    {  nuevo->sig=p;
       p->ant=nuevo;
       inicio=nuevo;
    }
    else
    {  q->sig=nuevo;
       nuevo->ant=q;
       nuevo->sig=p;
       p->ant=nuevo;
    }
    return(inicio);
 }

void mostrar(struct nodo *inicio)
{ struct nodo *aux;
 clrscr();
if(!inicio)
 printf("Esta vacia");
else
{ aux=inicio;
  do
  { printf("\n%d",aux->dato);
    aux=aux->sig;
  }while(aux!= NULL);
}
getch();
}


struct nodo* eliminar(struct nodo *inicio,int dato)
{  struct nodo *p, *q=NULL;
   p=inicio;
   while(p->dato!=dato && p!=NULL)
   {    q=p;
                p=p->sig;
   }
   if(q==NULL)
   { inicio=p->sig;
     inicio->ant=NULL;
     free(p);
   }
   else
     if(p!=NULL)
     { q->sig=p->sig;
               p->sig->ant=q;
               free(p);
     }
     else
     { printf("No encontrado");
               getch();
     }
 return inicio;
}

void main()
{struct nodo *inicio=NULL;
 char opc;
```

```
int dato,ban,pos;
 do{  clrscr ();
              gotoxy(27,9);  printf(" 1. Insertar en la lista");
              gotoxy(27,10); printf(" 2. Eliminar un elemento ");
              gotoxy(27,12); printf(" 3. Imprimir la lista");
              gotoxy(27,14); printf(" 4. Salir");
              gotoxy(27,16); printf("Opcion: [  ]\b\b"); opc=getche();
              switch(opc)
              { case '1': printf("Dato:");  scanf("%d",&dato);


                          if(inicio==NULL)
                                inicio=crearnodo(dato);
                               else
                                    inicio=insertar(inicio,dato);
                                   break;
                case '3': mostrar(inicio);
                                   break;
                case '2': clrscr(); printf("Escribe dato a borrar:  ");
                                   scanf("%d",&dato);
                                   inicio=eliminar(inicio,dato);
                                  break;
                case '4': break;
                }
      }while(opc!='4');
}
```

2.8 Implementación de una Lista Doblemente Enlazada en Java

```
Public class ListaDoble{

        private class Nodo{
        Object objeto;
        Nodo sig;
        Nodo ant;

        Nodo(Object obj){
             objeto=obj;
             sig=null;
             ant=null;
        }
    }

   private Nodo inicio, fin;

   ListaDoble(){
                inicio=null;
                 fin=null;
   }

     boolean vacia(){ return(inicio==null); }

     void insertar(Object obj){
        Nodo n=fin;
        fin=new Nodo(obj);
        if (vacia() )
          inicio=fin;
        else{
                n.sig=fin;
               fin.ant=n;
         }
     }

void insertarOrdenado(Object obj){
  Nodo n, p,q=null;
```

```
  p=inicio;
  n=new Nodo(obj);
  if (inicio==null) {
     inicio=fin=n;
  }
  else{
    while(p!=null&&obj.toString().compareTo(p.objeto.toString())>0){
           q=p;
           p=p.sig;
    }
    if (q==null ) {
                n.sig=inicio;
                inicio.ant=n;
                inicio=n;
   }
   else{q.sig=n;
         n.ant=q;
         n.sig=p;
          if (p==null)
            fin=n;
          else
            p.ant=n;
   }
  }
}

void  remover(Object obj){
 Nodo p,q=null;
 p=inicio;
 while( p!=null && obj.toString().compareTo(p.objeto.toString())!=0){
        q=p;
        p=p.sig;
 }

 if (q==null ) {
               inicio=inicio.sig;
               inicio.ant=null;
  }
  else{
          if (p!=null)
            if (p.sig!=null){
                     q.sig=p.sig;
                     p.sig.ant=q;
            }
            else{
                    q.sig=null;
                    fin=q;
            }
          else {
               System.out.println("No se encontro");
                new Teclado().espera();}
          }
 }
```

```
 Object remover(){
        Object obj=inicio.objeto;
        inicio=inicio.sig;
        return obj;
 }

  void mostrarDer(){
    Nodo n=inicio;
            do{ System.out.println(n.objeto);
                        n=n.sig;
            }while(n!=null);
 }

void mostrarIzq(){
            Nodo n=fin;
            do
            {  System.out.println(n.objeto);
              n=n.ant;
            }while(n!=null);
            }
}


public class pruebaListaDoble{
public static void main(String args[]){
  Integer x=new Integer(0);
  Teclado keyb=new Teclado();
   ListaDoble listaJava=new ListaDoble();
    int opc=1;
    boolean ban;
     do{ keyb.clrscr();
         System.out.println("1) Agregar ");
         System.out.println("2) Remover un objeto de la lista");
         System.out.println("3) Remover el primero objeto");
         System.out.println("4) Mostrar Izquierda-Derecha");
         System.out.println("5) Mostrar Derecha-Izquierda");
         System.out.println("6) Salida");
         do{
             ban=false;
             try{   System.out.println("Opcion ");
                    opc=x.parseInt(keyb.leer() );
              }
             catch(NumberFormatException exception){
                    System.out.println("Tienes que escribir un numero");
                     ban=true;
              }
           }while(ban);

           switch( opc){
             case 1: System.out.println("Dato que desea agregar");
                         listaJava.insertarOrdenado( keyb.leer());
```

```
                    break;
          case 2: if  (listaJava.vacia() )
                      System.out.println("No hay datos");
                    else{
                         System.out.println("Dato que desea borrar");
                        listaJava.remover( keyb.leer());
                    }
                    break;
             case 3: if(listaJava.vacia() )
                          System.out.println("No hay datos");
                      else{
                           System.out.println(listaJava.remover() );
                            keyb.espera();
                      }
                      break;
            case 4: keyb.clrscr();
                      if (!listaJava.vacia() )
                         listaJava.mostrarDer();
                      keyb.espera();
                      break;
            case 5: keyb.clrscr();
                      if (!listaJava.vacia() )
                        listaJava.mostrarIzq();
                      keyb.espera();
                      break;
          }
        } while(opc!=6);
   }
}
```

3. Ordenamientos y Búsquedas

Unidad III Ordenamientos y Búsquedas

3.1 Complejidad de los Algoritmos

Objetivos en la elección de un algoritmo

a) Qué el algoritmo sea fácil de entender, codificar y depurar.
b) Qué el algoritmo use eficientemente los recursos de la computadora y en especial que se ejecute con la mayor rapidez.

3.1.1 Análisis de Algoritmos vs. Análisis Empírico

Una vez que se ha definido un algoritmo para resolver un problema y se ha probado que es correcto, el siguiente paso es determinar la cantidad de recursos tales como tiempo y espacio que el algoritmo requerirá para su aplicación. La determinación del tiempo de ejecución puede hacerse mediante análisis empírico o con análisis de algoritmos.

Análisis empírico.

- La comparación de dos o más algoritmos se lleva a cabo ejecutándolos, por lo que requiere que ambos algoritmos estén correctamente implementados.
- Determina el uso de recursos y tiempo requerido en la misma máquina, con los mismos datos y el mismo ambiente. Por lo tanto, es muy importante la selección de datos de prueba, estos pueden ser aleatorios, datos reales o datos riesgosos.
- El código puede ejecutarse a diferentes velocidades dependiendo de la carga del sistema.

Análisis de Algoritmos

- El análisis matemático es más informativo y barato, pero puede dificultarse si no se conocen todas las fórmulas matemáticas.
- El código de programación de alto nivel podría no reflejar correctamente la ejecución en términos de lenguaje máquina. En ocasiones la optimización con la que se haya configurado el compilador puede afectar el código ejecutable.
- Identifica las operaciones abstractas en las cuales el algoritmo está basado y separa el análisis de la implementación.
- Identifica los datos para el mejor caso, el caso promedio y el peor de los casos.

3.1.2 Tiempo de Ejecución

El tiempo de ejecución de un algoritmo es una función que mide el número de operaciones (elementales o significativas) que realiza el algoritmo para un tamaño de entrada dado. El tiempo de ejecución de un algoritmo es, por tanto, función del tamaño de entrada. El valor exacto de esta función depende de muchos factores, tales como velocidad de la máquina, la calidad del compilador, y en algunos casos, la calidad del programa. Por ejemplo: para un mismo procesador efectuar el producto de dos números es mucho más lento si los números son de tipo real que si son enteros.

Con mucha frecuencia, el costo de un algoritmo depende no solo del tamaño de la entrada sino de los datos en particular.

Se presentan tres posibles situaciones:

a) $T_{sup}(n)$: costo del peor caso, el costo máximo del algoritmo.
b) $T_{inf(n)}$: costo del mejor caso, el costo mínimo del algoritmo.
c) $T_{med}(n)$: costo del caso promedio.

3.2 Conceptos Matemáticos Básicos

3.2.1 Series

$$1)\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$$

$$2)\sum_{i=0}^{n} a^i = \frac{a^{n+1} - 1}{a - 1}$$

$$3)\sum_{i=0}^{n} a^i = \frac{1}{1 - a}$$ **para 0>a>1**

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

$$5)\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

$$6)\sum_{i=1}^{n} i^k = \frac{n^{k+1}}{|k+1|}, k \neq 1$$

$$7)\sum_{i=1}^{N} \frac{1}{i} \approx \log_e N$$

$$8)\sum_{i=1}^{n} f(n) = nf(n)$$

$$9)\sum_{i=n_0}^{N} f(i) = \sum_{i=1}^{N} f(i) - \sum_{i=1}^{n_0 - 1} f(i)$$

3.3 Fundamentos Matemáticos del Análisis de Algoritmos

El análisis experimental, permite conocer el tiempo de ejecución exacto de un algoritmo, pero es muy complicado, porque requiere el manejo de lenguaje de bajo nivel del programa generado al compilar un código y del ambiente en que se ejecuta dicho programa.

Así que, en vez de lo anterior, se hace el análisis en forma directa en el programa de alto nivel o en el pseudocódigo. Se definen una serie de operaciones primitivas de alto nivel. Entre ellas se cuentan:

- Asignación de un valor a una variable.
- Llamada a un método.
- Ejecución de una operación aritmética.
- Comparaciones.

- Poner índices a un arreglo.
- Seguir referencia de objeto.
- Regresar de un método.

[3] Algorithms in Java. Mark Allen Weiss

En vez de determinar el tiempo exacto de ejecución de cada operación primitiva, sólo se contarán cuántas operaciones primitivas se ejecutan y esta cantidad se usará como estimado, en alto nivel, del tiempo de ejecución del algoritmo.

En un análisis de algoritmos es útil enfocarse en la tasa de crecimiento del tiempo de ejecución, en función del tamaño *n* de la entrada, adoptando un método de panorámica y no detenerse en los pequeños detalles.

3.2 Funciones de Crecimiento Típicas

C	Constante	La mayoría de las instrucciones se ejecutan una o cuando mucho unas cuantas veces. Si todas las instrucciones de un programa tienen esta propiedad, decimos que el tiempo de ejecución del programa es constante. Esta función de crecimiento, es deseable, pero difícil de conseguir
log N	Logarítmicas	Cuando el tiempo de ejecución de un programa es logarítmico el programa es ligeramente más rápido que el crecimiento de N. Esta función de crecimiento se presenta en programas que solucionan un problema transformándolo en una serie de pequeños problemas, esto es, dividen el tamaño del problema en una fracción constante en cada iteración.
N	Lineal	El tiempo de ejecución lineal, generalmente se encuentra cuando cada elemento de entrada requiere una pequeña parte de procesamiento. Cuando N es 10000 el tiempo de ejecución también. Cuando N se duplica el tiempo también se duplica. Esta situación es óptima para un algoritmo que debe procesar N entradas.
NlogN	NlogN	El tiempo de ejecución NlogN se presenta cuando los algoritmos solucionan un problema dividiéndolo en pequeños problemas, los resuelve en forma independiente y luego combina las soluciones. Cuando N es 1,000,000 NlogN es 20,000,000. Esta función de crecimiento es típica de los algoritmos divide y vencerás.
N^2	cuadrática	Cuando el tiempo de ejecución de un algoritmo es cuadrático, el algoritmo sólo es práctico para problemas relativamente pequeños. Esta función de crecimiento surge en algoritmos que procesan todos los pares de datos, por ejemplo, en ciclos anidados. Un ejemplo donde se observa esta función de crecimiento es el peor caso de Quicksort.
N^3	Cúbica	Un algoritmo que procesa ternas de datos tiene tiempos cúbicos, por ejemplo la multiplicación de matrices implementada en ciclos triples. El uso de estos algoritmos es práctico sólo en problemas pequeños.
2^N	Exponencial	Pocos algoritmos con tiempo de ejecución exponencial son apropiados para su uso, aunque surgen naturalmente como soluciones forzadas. Por ejemplo, cuando N es 20, el tiempo de ejecución es 1’000’000. Cuando N se duplica el tiempo de ejecución se eleva al cuadrado.

3.3.1 Notación Asintótica O

Asíntota: Las **asíntotas** son rectas a las cuales una función se aproxima indefinidamente, cuando x o f(x) tienden al infinito.

Para expresar los límites de complejidad de un algoritmo se usa una notación asintótica. La más común es la O (big Oh). La Notación O es un recurso matemático que permite suprimir los detalles en el análisis de algoritmos. Sea f(n) y t(n) funciones que representan como enteros no negativos a números reales. Se dice que:

$T(n) = O(f(n))$ si existen constantes C y n_0 tal que $T(n) \leq a + cf(n)$ cuando $n > n_0$

La variable a representa el costo de cualquier tarea o labor cotidiana.
c es una constante de multiplicación que representa el costo en unidades de ejecución de una operación fundamental.
En la práctica ignoramos los efectos de a, c y cualquier operación de poca importancia cuando comparamos complejidades.

Esta definición se lee T(n) es O de f(n). De otra forma, la complejidad de una función t(n) está limitada por la función f(n); es decir, el número máximo de operaciones fundamentales ejecutadas por T(n) no será mayor que f(n). La notación O es similar al menor o igual, cuando estamos refiriéndonos a tasas de crecimiento.

3.3.2 Notación Omega

Existen otras notaciones que nos permiten hacer comparaciones entre funciones. Una de ellas es la notación Omega $T(n) = \Omega(F(n))$, esta indica que la tasa de crecimiento de T(n) es mayor o igual que la de F(n).

3.3.3 Notación Theta

La notación Theta $T(n) = \Theta(F(n))$, esta indica que la tasa de crecimiento de T(n) es igual que la de F(n). Cuando utilizamos la notación theta estamos proporcionando no solamente un límite superior del tiempo de ejecución, sino que garantizamos que el análisis que nos lleva a este límite superior es lo más ajustado posible.

3.4 Modelo para el análisis de algoritmos

El modelo para el análisis de algoritmos es una computadora normal, en la cual las instrucciones se ejecutan secuencialmente. Esta tiene el típico conjunto de instrucciones: Asignación, comparación, suma, etc. El tiempo de ejecución de cada instrucción será una unidad de tiempo. También se asume una memoria infinita.

3.5 Conteo de Operaciones primitivas para el Análisis de Algoritmos

1) El acceso a un arreglo a través de un índice contará como una operación.
2) La asignación de un valor a una variable tiene un tiempo de ejecución de 1.
3) La inicialización de un contador tiene valor de 1.
4) La comparación de la condición de salida de un ciclo cuenta como una unidad.
5) El regreso de valores de una función o método tiene un peso de 1.
6) El tiempo de ejecución de un ciclo es el tiempo de ejecución de las instrucciones dentro del ciclo, multiplicado por el número de iteraciones.
7) El tiempo de ejecución de las instrucciones dentro de un grupo de ciclos anidados es el tiempo de ejecución de las instrucciones multiplicado por el tamaño de todos los ciclos.
8) Los enunciados consecutivos solo se suman.
9) El tiempo de ejecución de una instrucción if-else es el tiempo de la condición más el tiempo de la parte verdadera o falsa que requiera mayor tiempo de ejecución.

Ejemplo 1:

```
int encontrarMayor(int arreglo [], int n)
{int mayor, i;
```

```
   mayor=arreglo[0];
   for (i=0;i<n;i++)
      if (mayor<arreglo[i])
         mayor = arreglo[i];
   return(mayor);
}
```

Análisis para el Peor caso O

$2+1+n+\sum_{i=1}^{n} 6+1$ = 2+1+n +1 +6n +1 = **7n+5** por lo tanto se dice que es O(n)

Análisis para el Mejor caso Θ

$2+1+n+\sum_{i=1}^{n} 4+1$ = 2+1+ n+1 +4n+1 = **5 n+ 5** por lo tanto se dice que es Θ (n)

Ejemplo 2:

```
 int encontrarMayor(int arreglo [], int n)
{int mayor, i;
   mayor=arreglo[0];
   for (i=1;i<n;i++)
      if (mayor<arreglo[i])
         mayor = arreglo[i];
   return(mayor);
}
```

Análisis para el Peor caso O

$2+1+n+\sum_{i=1}^{n-1} 6+1$ = 2+1+n+6(n-1)+1 = 2+1+n+6n-6+1 = **7n-2** por lo tanto se dice que es O(n)

Análisis Mejor caso Θ

$2+1+n+\sum_{i=1}^{n-1} 4+1$ = 2+1+n+4(n-1)+1 = 2 +1+n+4n-4+1 = **5 n** por lo tanto se dice que es Θ (n)

3.6 Medidas Significativas

El conteo de las operaciones se puede hacer usando diferentes medidas significativas, estas son:

a) Número de Asignaciones Totales
b) Número de Operaciones Aritméticas
c) Número de Accesos al arreglo.
d) Número de Operaciones Elementales.

3.6.1 Ejemplos

Dado el siguiente algoritmo que suma los elementos de una matriz cuadrada de tamaño n.

```
suma=0;
```

```
i=0;
while (i<n)
{  j=0;
   while(j<n)
   { suma=suma+matriz[i][j];
     j++;
   }
   i++;
}
```

a) El análisis para el peor de los casos considerando número de **asignaciones totales** es: $2+\sum_{i=1}^{n}2+\sum_{i=1}^{n}\sum_{j=1}^{n}2=2+2n+2n^2$ por lo tanto el algoritmo es O(n²)

b) Tomando como medida significativa el número de **operaciones aritméticas** $\sum_{i=1}^{n}1+\sum_{i=1}^{n}\sum_{j=1}^{n}2=n+2n^2$ el algoritmo tiene un tiempo de ejecución O(n²).

c) Si se toma como medida significativa el número de accesos al arreglo: $\sum_{i=1}^{n}\sum_{j=1}^{n}1=n^2$ entonces el tiempo de ejecución del algoritmo es O(n²)

d) Tomando como medida significativa el **número de operaciones elementales**

$$2+n+1+\sum_{i=1}^{n}(1+n+1+\sum_{j=1}^{n}5+2)=$$

$$3+n+\sum_{i=1}^{n}(2+n+5+2)=$$

$$3+n+2n+n^2+5n^2+2n$$

$$6n^2+5n+3$$

entonces el algoritmo tiene un tiempo de ejecución O(n²).

3.7 Ejemplo

```
void main()
{int N=20,i,j,k,suma=0;
 clrscr();
 for(i=1;i<=N;i++)
  for(j=i;j<=N;j++)
    for(k=j;k<=N;k++)
          suma++;
```

```
}
```

$$\sum_{i=1}^{N}\sum_{j=i}^{N}\sum_{k=i}^{j}1 \qquad\qquad \sum_{i=1}^{N}\sum_{j=i}^{N} j-i+1$$

considerando que 1+2+3+… +(N-i+1) entonces lo anterior se puede escribir

$$\sum_{i=1}^{N}\sum_{j=i}^{N} N-i+1 \quad \text{aplicando} \quad \boxed{\sum_{i=1}^{n} i = \frac{n(n+1)}{2}}$$

$$\sum_{i=1}^{N}\frac{(N-i+1)(N-i+1+1)}{2} = \sum_{i=1}^{N}\frac{(N-i+1)(N-i+2)}{2}$$

$$\frac{1}{2}\sum_{i=1}^{N}(N-i+1)(N-i+2) \quad = \frac{1}{2}\sum_{i=1}^{N} N^2 - iN + 2N - iN + i^2 - 2i + N - i + 2$$

$$\frac{1}{2}\sum_{i=1}^{N} N^2 - 2iN + 3N + i^2 - 3i + 2 \quad = \frac{1}{2}\sum_{i=1}^{N}(N^2+3N+2) + \frac{1}{2}\sum_{i=1}^{N}(i^2 - 2iN - 3i)$$

$$\frac{1}{2}\sum_{i=1}^{N}(N^2+3N+2) + \frac{1}{2}\sum_{i=1}^{N} i^2 - \frac{1}{2}(2N+3)\sum_{i=1}^{N} i \quad \text{Aplicando} \quad \boxed{\sum_{i=1}^{n} i = \frac{n(n+1)}{2}} \text{ y } \boxed{\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}}$$

$$\frac{1}{2}N(N^2+3N+2) + \frac{1}{2}\frac{N(N+1)(2N+1)}{6} - \frac{1}{2}(2N+3)\frac{(N)(N+1)}{2}$$

$$\frac{1}{2}\left(N^3+3N^2+2N\right) + \frac{1}{12}\left(N^2+N\right)(2N+1) - \frac{1}{4}(2N+3)\left(N^2+N\right)$$

$$\frac{1}{2}\left(N^3+3N^2+2N\right) + \frac{1}{12}\left(2N^3+N^2+2N^2+N\right) - \frac{1}{4}\left(2N^3+2N^2+3N^2+3N\right)$$

$$\frac{1}{2}\left(N^3+3N^2+2N\right) + \frac{1}{12}\left(2N^3+3N^2+N\right) - \frac{1}{4}\left(2N^3+5N^2+3N\right)$$

$$\frac{6N^3+18N^2+12N+2N^3+3N^2+N-6N^3-15N^2-9N}{12}$$

$$= \frac{2N^3+6N^2+4N}{12} = \boxed{\frac{N^3+3N^2+2N}{6}}$$

3.6.2 Ejercicios. En cada ejercicio encuentre el tiempo de ejecución para el peor de los casos considerando todas las operaciones significativas.

1)
```
for (i=1;i<=n;i++)
      x++;
```

2)
```
 for (i=1;i<=n;i++)
      for (j=1;j<=n;j++)
           x++;
```

3)
```
for (i=1;i<=n;i++)
      for (j=1;j<=n;j++)
           for (k=1;k<=n;k++)
                x++;
```

4)
```
i=1;
do
{ x++;
  i=i+2;
} while(i<=n);
```

5)
```
      x=1;
      while(x<N)
          x=2*x;
```

3.2 Métodos de Ordenación

Tipos de Ordenamiento

Interno: Cuando los datos están almacenados en la memoria principal.
Externo: Cuando los datos están guardados en un medio de almacenamiento secundario.

3.2.1 Método de Inserción directa

Este método consiste en dividir la lista de datos a ordenar en dos sublistas una ordenada y la otra desordenada. El algoritmo es el siguiente:

1. Recorrer la lista para seleccionar el elemento a ordenar i=1.
2. Recorrer la sublista desordenada desde el inicio hasta i-1 para encontrar la posición que debe ocupar el dato.
3. Recorrer hacia la derecha todos los elementos
4. Insertar el dato en la posición correspondiente.

```
void insercion(int lista[] )
{
  int i,k,j,pos;
  for(i=1;i<tam;i++)
  {  k=lista[i];
     j=0;
```

```
do
{  if(lista[i]<lista[j])
       {  for(pos=i;pos>j;pos--)
                lista[pos]=lista[pos-1];
          lista[j]=k;
       }
                                        else
                                         j++;
                                    }while(j<i);
                                  }
                                 }
```

3.2.2 Método de Inserción Binaria

Este método consiste en dividir la lista de datos a ordenar en dos sublistas una ordenada y la otra desordenada. El algoritmo es el siguiente:

1. Recorrer la lista para seleccionar el elemento a ordenar i=1.
2. Buscar la posición del elemento en la mitad izquierda o derecha de la lista, según la tendencia.
3. Recorrer hacia la derecha todos los elementos
4. Insertar el dato en la posición correspondiente

```
void ordinsbin(int L[])
{       int i, j, k, izq, der, mitad;
        for(i=1; i<10; i++){
        k=L[i];
    izq=0;
    der=i;
    while(izq<der){
                mitad=(izq+der)/2;
                        if(L[mitad]<=k)
                izq=mitad+1;
                        else der=mitad;
          }//while
          for(j=i; j>der; j--)
        L[j]=L[j-1];
        L[der]=k;
  }//for
}
```

3.3 Ejercicios

1. Tenemos un vector que contiene N números. Queremos saber si contiene dos números cuya suma sea igual a un número dado K. Por ejemplo, si la entrada es (8, 4, 1, 6) y K es 10, la respuesta es sí, (4, 6). Un mismo número puede ser usado dos veces. Hacer lo siguiente:

   a) De un algoritmo O(n^2) para resolver el problema.
   b) De un algoritmo O(N log N) para resolver el problema. Ordene primero los elementos. Después de hacer esto puede resolver el problema en tiempo lineal.
   c) Escriba un programa para los incisos a y b y compare sus tiempos de ejecución.

2. ¿Cuándo todos los elementos son iguales, cuál es el tiempo de ejecución de los siguientes métodos?

   a) Ordenación por inserción
   b) Shellsort
   c) Mergesort
   d) Quicksort

3. ¿Cuando la entrada está ordenada, pero en orden inverso, cuál es el tiempo de ejecución de los siguientes métodos?

   a) Ordenación por inserción
   b) Shellsort
   c) Mergesort
   d) Quicksort

4. Un array de registros se quiere ordenar según el campo clave fecha de nacimiento. Dicho campo consta de tres subcampos: día, mes y año, de 2,2, y 4 dígitos respectivamente. Adaptar el método de ordenación Radixsort a esta ordenación.

5. Un array contiene los elementos indicados más abajo. Utilizando el algoritmo de búsqueda binaria, trazar las etapas necesarias para encontrar el número 88.

   8 13 17 26 44 56 88 97

   Hacer la misma búsqueda pero para el número 20.

6. Supongamos que se tiene una secuencia de n números que deben ser clasificados, utilizando el método de Shell, Cuántas comparaciones y cuántos intercambios se requieren para clasificar la secuencia si

   a) Ya está ordenado
   b) Ya está orden en inverso

4.

4.5 Ejercicios

1. Sea C una cola circular de 6 elementos. Inicialmente la cola está vacía. Dibuje el estado de C luego de realizar cada una de las siguientes operaciones.
   Insertar los elementos: AA, BB y CC
   Eliminar el elemento: AA
   Insertar los elementos: DD, EE y FF
   Insertar el elemento: GG
   Insertar el elemento: HH
   Eliminar los elementos: BB y CC

   ¿Con cuántos elementos quedó C?
   ¿Hubo algún caso de error de overflow o underflow?

2. Sea C una doble cola circular de 6 elementos. Inicialmente la cola doble está vacía. Dibuje el estado de C luego de realizar cada una de las siguientes operaciones.
   Insertar por el extremo derechos los 3 elementos: A, B y C
   Eliminar por el extremo izquierdo un elemento
   Insertar por el extremo derechos los 2 elementos: D y E
   Eliminar por la derecha un elemento

   ¿Con cuántos elementos quedó C?
   ¿Hubo algún caso de error de overflow o underflow?

3. Se tiene una lista enlazada a la que se accede por el primer nodo. Escribir un método que imprima los nodos de la lista en orden inverso, desde el último nodo al primero; como estructura auxiliar utilizar una pila y sus operaciones.

4. Se quieren determinar las frases que son palíndromo, para lo cual se ha de seguir la siguiente estrategia: considerar cada línea de una frase; añadir cada carácter de la frase a la pila.

5. Escriba un subprograma recursivo que, dadas dos listas ordenadas ascendentemente, las mezcle generando una nueva lista ordenada descendentemente.

3.1 Búsquedas

3.1.1 Búsqueda Lineal

El método de búsqueda más sencillo es la búsqueda lineal, este consiste en recorrer todo el vector comparando con la llave de búsqueda, después de encontrar el elemento que coincida regresa la posición donde lo encontró. Si no encuentra coincidencia alguna entonces regresa -1.

3.1.2 Búsqueda Binaria

Este método requiere que el arreglo este previamente ordenado. La búsqueda binaria funciona de la siguiente manera:

1. Encontrar el elemento central del arreglo, la llave se compara con el centro si es igual aquí termina la búsqueda.
2. Si no es igual determinar si la llave se encuentra en el lado izquierdo o derecho de la lista.
3. Redefinir el inicio o el final según donde ese haya ubicado la llave
4. Calcular el centro de la lista.

5. Comparar con el elemento del nuevo centro.
6. Repetir hasta encontrar el dato o hasta que ya no sea posible dividir más.

Implementación

```
int binaria(int lista[25], int num)
{
  int izq=0, der=tam-1, centro,i=1;
   while(izq<=der)
   {
            centro=(izq+der)/2;
            if(num==lista[centro])
              return(centro);

            else
              if (num>lista[centro])
                izq=centro+1;
              else
                der=centro-1;
            i++;
   }//while
   return(-1);
}
```

La búsqueda binaria es logarítmica debido a que en cada iteración el tamaño de la lista de búsqueda se reduce a la mitad.

3.1.2 Búsqueda Interpolada

Si los datos están distribuidos de forma uniforme, este método puede ser más eficiente que la búsqueda binaria. Básicamente el algorimo es el mismo que el de la búsqueda binaria. La diferencia radica en que en la búsqueda interpolada no se busca el dato en el centro del arreglo sino que se calcula su posición aproximada.

```
int interpolada(int lista[25], int num)
{
  int izq=0, der=tam-1, centro,i=1;
   while(izq<=der)
   {
            centro=izq+(der-izq)*( (num-
lista[izq])/(lista[der]-lista[izq] ));
            if(num==lista[centro])
            {    printf("Iteraciones: %d",i);
                 getch();
                return(centro);
            }
            else
              if (num>lista[centro])
                izq=centro+1;
              else
                der=centro-1;
            i++;
   }//while
     printf("Iteraciones: %d",i);
   getch();

   return(-1);
}
```

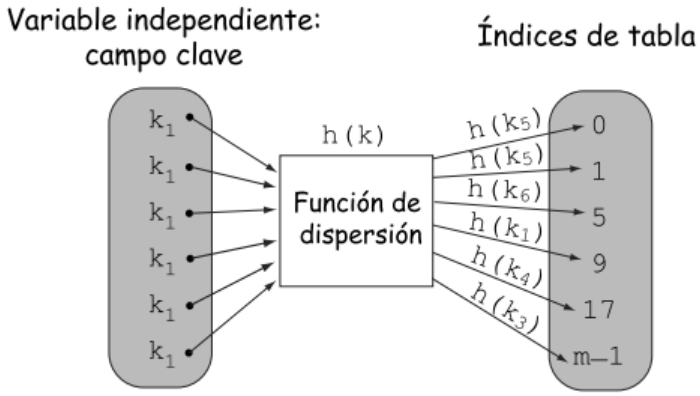
3.1.3 Tablas Hash

Suponga que una compañía tiene 300 empleados identificados con un número de empleado de 0 a 999, suponiendo además, que los datos de los empleados se almacenan en un arreglo y considerando que los número de empleado no se repiten, se puede utilizar el número como índice para acceder de manera directa a un empleado con una eficiencia O(1). Sin embargo, muchas posiciones de la tabla, 700 para este caso, estarán vacías y por lo tanto habría memoria desaprovechada. ¿Cómo evitar que el arreglo utilizado esté en proporción adecuada al número real de registros? Las funciones de transformación de claves, funciones hash, permiten que el rango de índices sea proporcional al número real de registros.

Las funciones que transforman números grandes en números más pequeños se conocen como funciones de dispersión o funciones hash

Una tabla hash o tabla de dispersión es una estructura que ofrece una rápida inserción y búsqueda, sin importar la cantidad de elementos que haya, estas operaciones y en ocasiones, la eliminación*, tienen un tiempo de ejecución que se aproxima a una constante O(1).

Las tablas hash permiten el acceso a información almacenada por medio de claves en vez de utilizar índices. Una tabla hash está conformada por dos valores, la clave y un valor asociado a dicha clave única.



Una tabla hash está formada por un arreglo de entradas, que será la estructura que almacene la información, y por una función de dispersión. La función de dispersión permite asociar el elemento almacenado en una entrada con la clave de dicha entrada. Por lo tanto, es un algoritmo clave para el buen funcionamiento de la estructura.

Las tablas Hash tienen desventajas, están basadas en arreglos y los arreglos son difíciles de expandir una vez que han sido creados, además para algunas tablas hash el desempeño puede degradarse cuando las tablas están a punto de llenarse, por lo que el programador requiere tener una idea cercada a la cantidad de datos que serán almacenados o preparase para transferir datos a una tabla más grande, lo que es un proceso que consume tiempo.

Operaciones en tablas de dispersión

- Insertar(Tabla T, elemento k) añade el elemento k, T[h(clave(k)] ← k
- Buscar(Tabla T, clave x) devuelve el elemento de la tabla T[h(x)]
- Eliminar(Tabla T, clave x) retira el elemento con clave x, T[h(x)] ← free

En las tablas dispersas no se utiliza directamente la clave para acceder a un elemento, el índice se calcula con una función matemática, función hash h(x), que tiene como argumento la clave del elemento y devuelve una dirección o índice en el rango de la tabla.

El parámetro que mide la proporción entre el número de elementos almacenados, n, en una tabla dispersa y el tamaño de la tabla, m, se denomina factor de carga, $\lambda = n/m$. Se recomienda elegir m de tal forma que el factor de carga sea $\lambda \leq 0.8$.

Función de dispersión Aritmética modular

Utiliza la aritmética modular genera valores dispersos calculando el residuo de la división entre la clave x y el tamaño de la tabla m.

$$h(x) = x \text{ modulo } m$$

Normalmente, la clave asociada con un elemento es de tipo entero, si no es así antes se deberá transformar la clave a un valor entero. Por ejemplo, si la clave es una cadena de caracteres, se puede transformar considerando el valor ASCII de cada carácter como si fuera un dígito entero de base 128. La operación módulo genera un numero entre 0 y m-1 cuando el segundo operando es m. Por tanto, esta función de dispersión proporciona valores enteros dentro del rango 0 ... m-1. Con el fin de que esta función disperse lo más uniformemente posible, es necesario tener
ciertas precauciones con la elección del tamaño de la tabla, m. Así, no es recomendable escoger el valor de m múltiplo de 2 ni tampoco de 10.

Método de la multiplicación

La dispersión de una clave utilizando el método de la multiplicación genera direcciones en tres pasos.

1. Multiplica la clave x por una constante real, R, 0 y 1. Una elección de la constante R es la inversa de la razón áurea, R = 0.6180334.

2. Determina la parte decimal, d, del número obtenido en la multiplicación, R*x.

3. Multiplica el tamaño de la tabla, m, por d y al truncarse el resultado se obtiene un número entero en el rango 0 ... m-1 que será la dirección dispersa.

1. R * x
2. d = R * x – *ParteEntera*(R * x)
3. h(x) = *ParteEntera*(m * d)

Implementación en Java

```
package tablasHash;

import java.io.*;
public class DispersionHash{
   static final int M = 1024;
   static final double R = 0.618034;

   static long transformaClave(String clave) {
      long d;
      d = 0;
      for (int j = 0; j < Math.min(clave.length(),10); j++){
         d = d * 27 + (int)clave.charAt(j);
      }

      if (d < 0)        /* Para un valor mayor que el máximo entero genera un numero negativo. */
         d = -d;

      return d;
   }

      static int dispersion(long x) {
         double t;
         int v;
         t = R * x - Math.floor(R * x); // parte decimal
         v = (int) (M * t);
         return v;
      }
      public static void main(String[]a) throws IOException {
      String clave;
      long valor;
      BufferedReader entrada = new BufferedReader( new InputStreamReader(System.in));

      for (int k = 1; k <= 10; k++)  {
         System.out.print("\nClave a dispersar: ");
         clave = entrada.readLine();
         valor = transformaClave(clave);
         valor = dispersion(valor);
         System.out.println("Dispersion de la clave " +
         clave + " \t " + valor);
      }
   }
}
```

4. Árboles.

El árbol es una abstracción matemática de una estructura no lineal que modela una estructura jerárquica. El árbol juega un papel central en el diseño y análisis de algoritmos ya que se utilizan para describir propiedades dinámicas de los algoritmos y porque se construyen. Los árboles se encuentran frecuentemente en la vida diaria: en árboles genealógicos y representación de torneos. En computación los encontramos en los compiladores, en la organización de sistemas de archivos la estructura de herencia de las clases de Java es un árbol, la invocación de los métodos en tiempo de ejecución en Java es un árbol; procesamiento de textos y algoritmos de búsqueda.

4.1 Árboles binarios

Un árbol binario es un conjunto finito de elementos que puede estar vacío o puede estar dividido en tres subconjuntos desarticulados. El primer subconjunto contiene un solo elemento llamado raíz del árbol. Los otros dos son en sí mismos árboles binarios, llamados subárboles izquierdo y derecho del árbol original. Un subárbol izquierdo o derecho puede estar vacío. Cada elemento del árbol binario se llama nodo. [A. Tenenbaum].

- Conjunto finito de nodos el cual puede ser vacío o tener un par de árboles llamados izquierdo y derecho. Cuando un nodo no tiene hijos se le llama hoja o nodo terminal.
- La Altura de un árbol es el número de niveles que tiene.
- Un árbol es completo cuando contiene el número máximo de nodos para su altura.

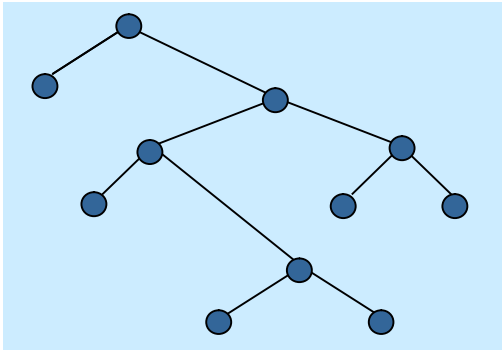


Fig. 5-1

4.2 Árboles Ternarios

Un árbol ternario es una estructura similar a un árbol, tiene una raíz y cada nodo tiene máximo tres hijos.

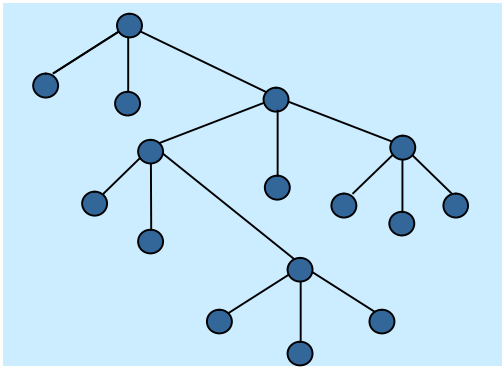


Fig. 5-2

4.3 Árboles libres

Es una colección de vértices y lados que satisfacen ciertos requerimientos. Un vértice es un objeto que tienen un nombre y puede contener otra información asociada. Un lado es una conexión entre dos vértices.

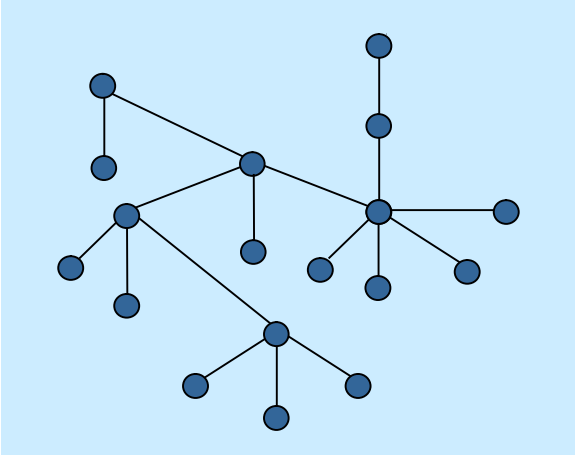


Fig. 5-3

4.4 Árboles con raíz

En este árbol un nodo es designado como la raíz del árbol, en computación se usa a este concepto se le conoce simplemente como árbol.

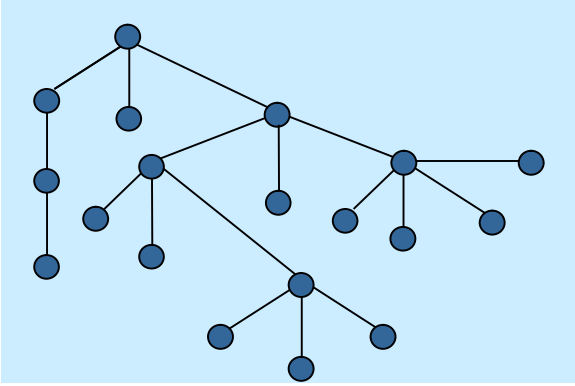


Fig. 5-4

4.5 Recorrido de Árboles

Notación PreOrden

1) Visita Raiz
2) Visitar Izquierdo
3) Visitar Derecho

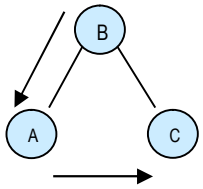


Fig. 5-5

El resultado de visitar un árbol como el de la figura 5.5 es B-A-C

Notación InOrden

1) Visitar Izquierdo
2) Visita Raiz
3) Visitar Derecho

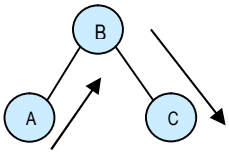
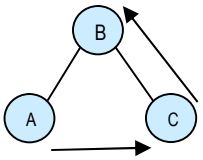


Fig. 5-6

El resultado de visitar en orden el árbol de la figura 5.6 es A-B -C

Notación PostOrden

1) Visitar Izquierdo
2) Visitar Derecho
3) Visita Raiz



El resultado de visitar en orden el árbol de la figura 5.6 es A-C-B

5.5 Implementación de un árbol binario en C.

```
#include <alloc.h>
#include "c:\lety\algori~1\estruc~1\tree.h"

void main() {
 struct nodo *raiz=NULL;
 char opc;
 int dato,x,s;
    do{      clrscr ();
             gotoxy(27,8);  printf(" 1. Insertar  dato ");
             gotoxy(27,10); printf(" 2. Imprime preorden");
             gotoxy(27,12); printf(" 3. Imprime inorden");
             gotoxy(27,14); printf(" 4. Imprime postorden");
             gotoxy(27,16); printf(" 5. Buscar dato");
             gotoxy(27,18); printf(" 6. Borrar dato");
             gotoxy(27,20); printf(" 7. Salir");             gotoxy(27,22); printf("Opcion: [
]\b\b");
             opc=getche();
               switch(opc)  {
             case '1':  printf("Escribe dato: ");
                        scanf("%d",&x);
                        if (raiz==NULL)
                         raiz=crear(x);
                        else
                         insertar(raiz,x);
                       break;
             case '2': preorden(raiz);
                        getch();
                       break;
             case '3':  orden(raiz);
                        getch();
                       break;
             case '4':  postorden(raiz);
                        getch();
                       break;
             case '5':  printf("Escribe dato a buscar: ");
                        scanf("%d",&s);
                        busqueda(raiz,s);
                       break;
             case '6':  printf("Escribe dato a borrar: ");
                        scanf("%d",&x);
                        raiz=borrar(raiz,x);
                        getch();
                       break;
             case '7': break;
             default: clrscr();
                       gotoxy(31,10); printf("La opcion no es valida ");
                       getch();
                 break;
              }

       }while(opc!='7');
}
```

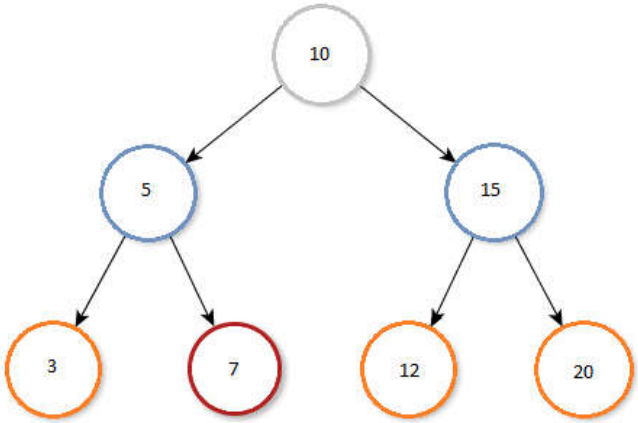
```
// Archivo Tree.h
struct nodo{ int dato;
                struct nodo *izq;
                 struct nodo *der;
 };

struct nodo * crear(int s)
{struct nodo *p;
 p=(struct nodo *) malloc(sizeof(struct nodo));
 p->izq=NULL;
 p->der=NULL;
 p->dato=s;
 return(p);
}

void insertar(struct nodo *raiz,int s)
{ struct nodo *nuevo,*q,*p;
  nuevo=crear(s);
  q=NULL;
  p=raiz;
  while(p!=NULL && p->dato!=nuevo->dato)
  { q=p;
    if (p->dato > nuevo->dato)
       p=p->izq;
    else
//       if (p->dato< nuevo->dato)
           p=p->der;
  }
  if (p->dato!=nuevo->dato)
     if (q->dato>nuevo->dato)
           q->izq=nuevo;
     else
           q->der=nuevo;
  }









void busqueda(struct nodo *raiz, int s){
if (s < raiz->dato){
```

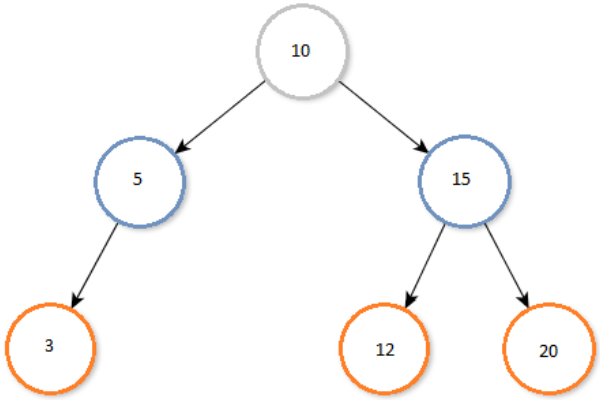
```
  if(raiz->izq==NULL){
     printf("\n\nNo existe el dato");
     getch();
    }
  else
    busqueda (raiz->izq,s);
  }

else
{ if(s>raiz->dato)
  {  if(raiz->der==NULL)
     {      printf("\no se encuentra");
            getch();
     }
     else
            busqueda (raiz->der,s);
  }
  else
  {  printf("\n\nEl dato si esta en el arbol");
     getch();
  }
 }

void preorden (struct nodo *raiz)
{
   if(raiz != NULL){
      printf("\n\t%d  ",raiz->dato);
      preorden(raiz->izq);
      preorden(raiz->der);
  }
}

void orden(struct nodo *raiz){
   if(raiz != NULL){
      orden(raiz->izq);
      printf("\n%d",raiz->dato);
      orden(raiz->der);
  }
}

void postorden (struct nodo *raiz){
   if(raiz != NULL){
      postorden(raiz->izq);
      postorden(raiz->der);
      printf("\n%d",raiz->dato);
   }
}
```

Eliminar un nodo de un árbol binario

Caso 1 Eliminar un nodo sin hijos

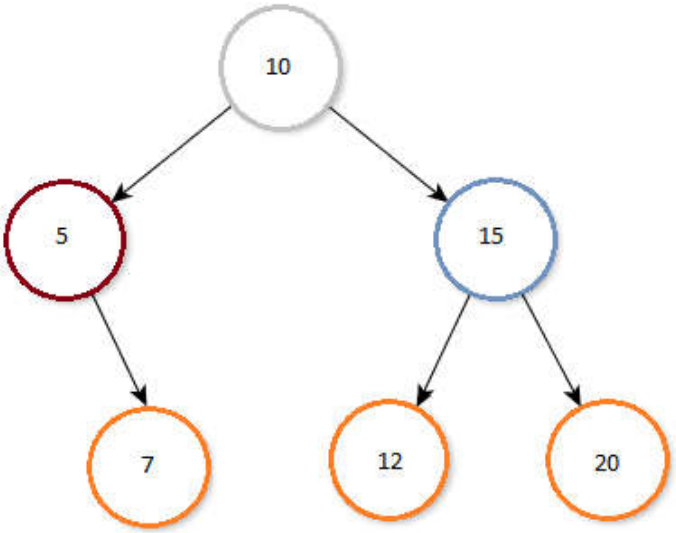
Es el caso más simple, lo único que hay que hacer es poner el apuntador de su padre a nulo y eliminar el nodo, con free si la implementación es en C o dejar de utilizarlo y esperar a que la máquina virtual lo recicle.

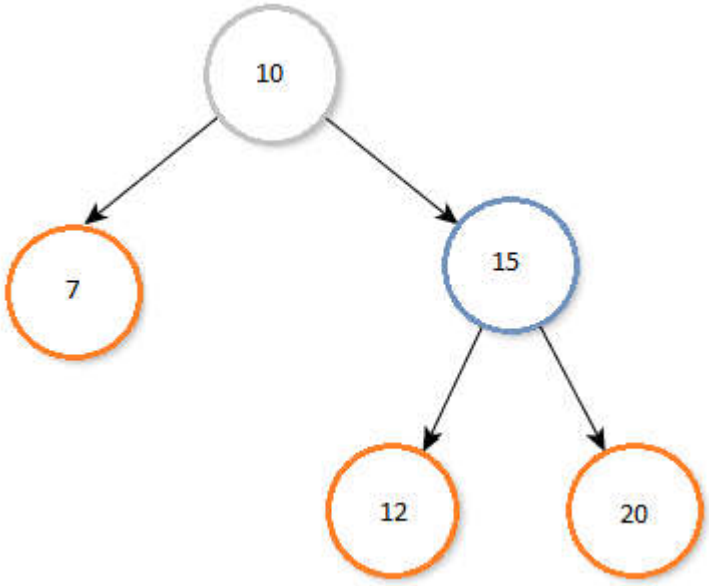




Caso 2 Borrar un nodo con subárbol

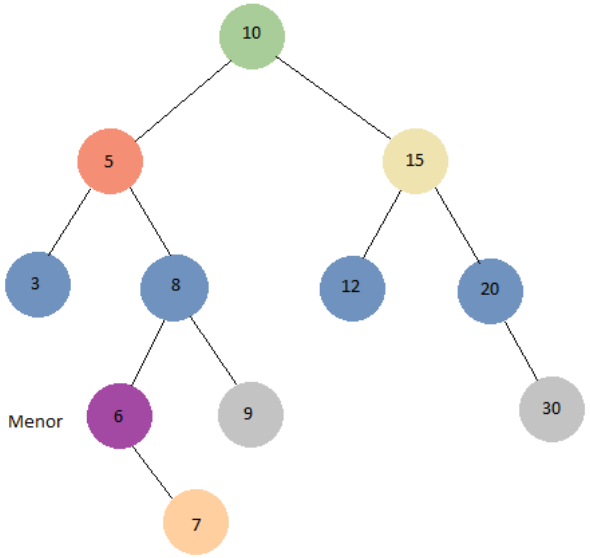
Si el nodo que vamos a eliminar tiene un solo hijo, se cambia la liga de manera que el hijo ocupa el lugar del padre y el nodo se libera.

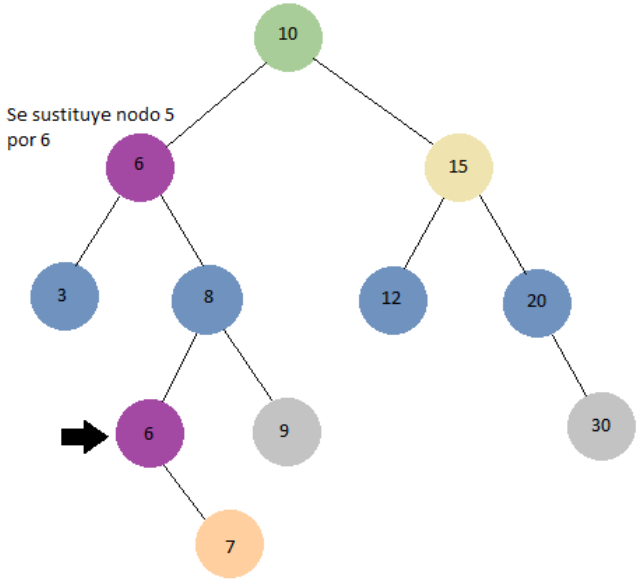


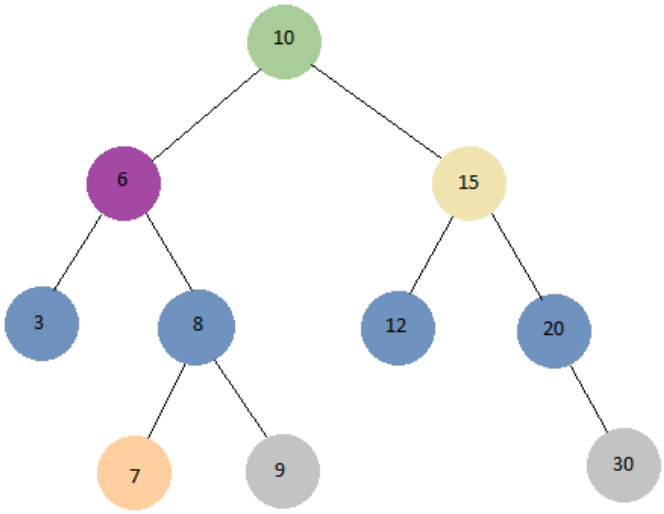


Caso 3. El nodo que se va a eliminar tiene dos subárboles

- Se toma el hijo derecho del nodo que se va a eliminar
- Se recorre por ese camino hasta el hijo llegar de más a la izquierda
- Se sustituye el nodo que se va a eliminar por el hijo que se encontró en el paso anterior.
- Si el subárbol tiene hijos se procede como en los casos 1 y 2







Implementación de forma Recursiva

```
void borrar(tarbol **a, int elem)
{
 void sustituir(tarbol **a, tarbol **aux);
 tarbol *aux;

 if (*a == NULL)
  return;

 if ((*a)->clave < elem) borrar(&(*a)->der, elem);
 else if ((*a)->clave > elem) borrar(&(*a)->izq, elem);
 else if ((*a)->clave == elem) {
  aux = *a;
  if ((*a)->izq == NULL) *a = (*a)->der;
  else if ((*a)->der == NULL) *a = (*a)->izq;
  else sustituir(&(*a)->izq, &aux);

  free(aux);
 }
}
void sustituir(tarbol **a, tarbol **aux)
{
 if ((*a)->der != NULL) sustituir(&(*a)->der, aux);
 else {
  (*aux)->clave = (*a)->clave;
  *aux = *a;
  *a = (*a)->izq;
 }
}
```

Implementación Iterativa

```
struct nodo * borrar(struct nodo *raiz,int num)
{ struct nodo *q,*p;
 struct nodo *rp, *s, *f;
 q=NULL;
 p=raiz;
 while(p!=NULL && p->dato!=num)
 { q=p;
  if (p->dato > num)
            p=p->izq;
    else
     if (p->dato< num)
               p=p->der;
 }

 if(p==NULL)
   printf("No existe");
 else
   if (p->izq == NULL)
            rp=p->der;
   else
    if(p->der==NULL)
              rp=p->izq;
    else
    { f=p;
             rp=p->der;
             s=rp->izq;
             while(s!=NULL)
             { f=rp;
               rp=s;
               s=rp->izq;
             }
             if(f!=p)
             {  f->izq=rp->der;
                rp->der=p->der;
             }
             rp->izq=p->izq;
    }

    if (q==NULL)
              raiz=rp;
           else
            if(p==q->izq)
              q->izq=rp;
            else
              q->der=rp;
           free(p);
 return(raiz);
}
```

6 Grafos

6.1 Definiciones

Un grafo se define como un "conjunto finito de puntos (vértices y nodos), algunos de los cuales están conectados por líneas o flechas (aristas)." [1]
Es un objeto matemático que permite expresar de forma sencilla y efectiva las relaciones entre elementos de diferente tipo. [2]
Un grafo G=<V,E,Φ,> consta de un conjunto no vacío V llamado conjunto de nodos (puntos o vértices) del grafo. Se dice que E es el conjunto de aristas del grafo y Φ es una transformación del conjunto de aristas E a un conjunto de pares, ordenados o no ordenados de elementos de V.

El documento más antiguo sobre Teoría de Grafos es, al parecer, "Solutio problematis ad geometriam situs pertinentis", escrito por Leonhard Euler. En este documento Euler plantea si es o no posible pasear por Konigsberg cruzando cada uno de sus puentes solamente una vez.

6.2 Elementos de un Grafo

Un grafo está formado por:

- Un conjunto de nodos, también llamados por algunos autores vértices o puntos.
- Un conjunto de arcos, también llamados bordes o aristas.

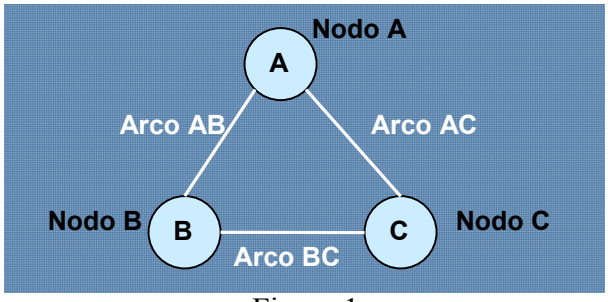


Figura 1

Vértices o Nodos Adyacentes: Son aquellos nodos que forma parte de la misma arista, en la figura 1 Ay B son adyacentes.

Arista o arco incidente: Se dice que una arista o arco es incidente en un nodo cuando el nodo es uno de los extremos de la arista. En el ejemplo el arco AC es incidente en A y en C.

6.3 Tipos de Grafos

6.3.1. Grafo dirigido. También llamado dígrafo. Se dice que un grafo es dirigido cuando todos los pares de nodos están ordenados; es decir, el grafo se recorre en una dirección pero no en la dirección contraria.

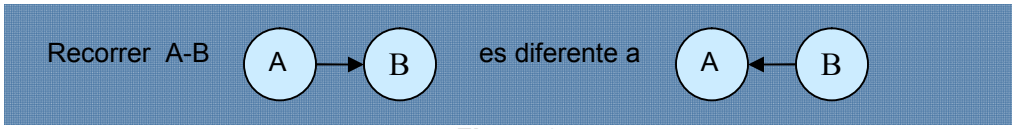


Figura 2

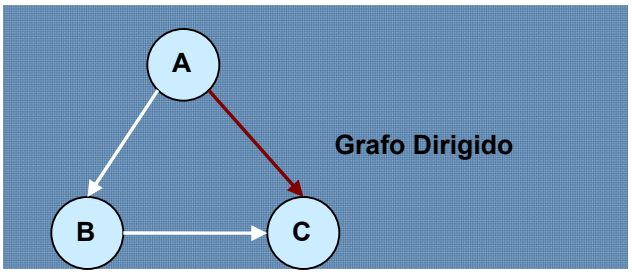


Figura 3

Arista o Arco Saliente: Se dice que un arco dirigido es saliente de un nodo A cuando el nodo A es el origen del arco. En la figura 3 se dice que "el arco AC es saliente de A".
Arista o Arco Entrante: Se dice que un arco dirigido es entrante a un nodo C cuando el nodo C es el destino del arco. En la figura 3 se dice que "el arco AC es entrante a C".
Grado de entrada. El grado de entrada de un vértice es la cantidad de arcos entrantes a él. Se denota por $deg_E(x)$
Grado de Salida. El grado de entrada de un vértice es la cantidad de arcos saliente de él. Se denota por $deg_S(x)$
Grado de un nodo. El grado de un nodo o vértice es la cantidad de arcos incidentes que tiene. Se denota por deg(x).

6.3.2 Grafo no dirigido.

También llamado dígrafo. Se dice que un grafo es dirigido cuando ninguno de los pares de nodos están ordenados.

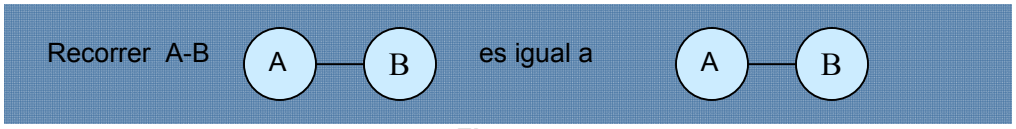


Figura 4

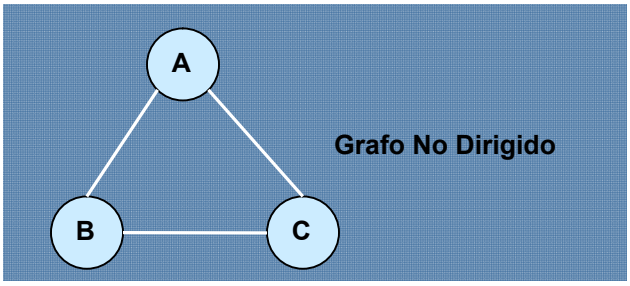
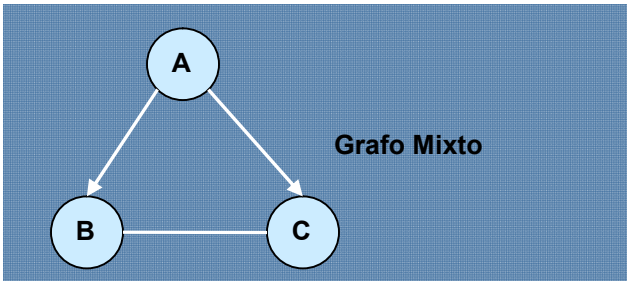


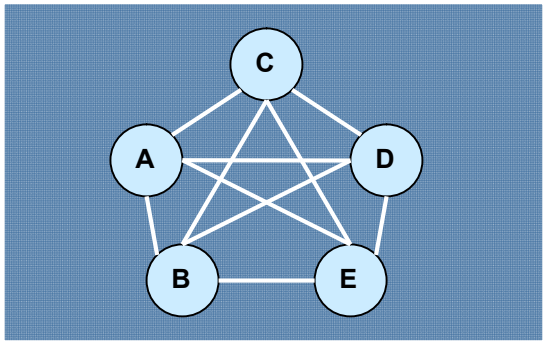
Figura 5

6.3.3 Grafo mixto.

Se dice que un grafo es mixto si algunas de las aristas están ordenadas y otras no



6.4 Clasificación por número de aristas
Por su cantidad de aristas los grafos también se clasifican en grafos completos, dispersos y densos.
6.4.1 Grafo Completo. Se dice que un grafo es completo si todos sus nodos estan conectados entre sí.



6.4.2 Grafo Disperso Un grafo disperso es aquel que tiene pocas aristas conectadas entre sí.

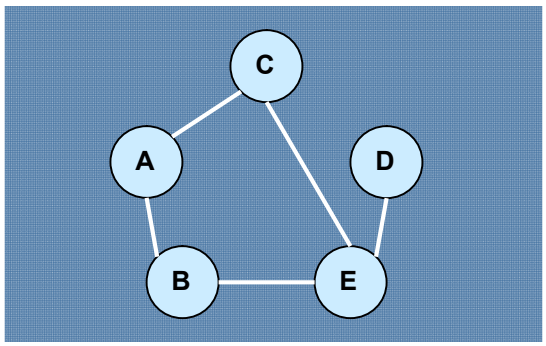


Figura 10

6.4.3 Grafo Denso Los grafos densos se caracterizan porque la mayoría de sus aristas están interconectadas.

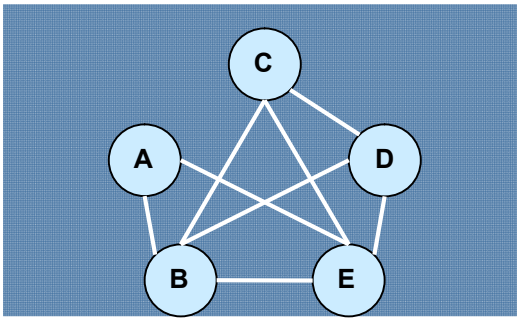
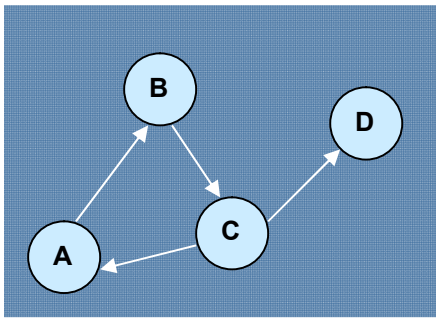


Figura 11

6.5 Aplicaciones de los grafos
La estructura de ligas de un website se puede representar como un grafo dirigido, los vértices son las páginas disponibles en el website, una arista dirigida AB existe si y solo si la página A contiene una liga hacia B. Un enfoque similar puede darse a problemas en viajes, biología, diseño de chips para computadora y muchos otros campos.

6.6 Representación de un grafo dirigido con una matriz de adyacencia

La matriz de adyacencia consiste en un arreglo de dos dimensiones en el que se guardan las conexiones entre pares de vértices. En la matriz se marca con **true** cuando la conexión existe y con **false** si la conexión no existe.



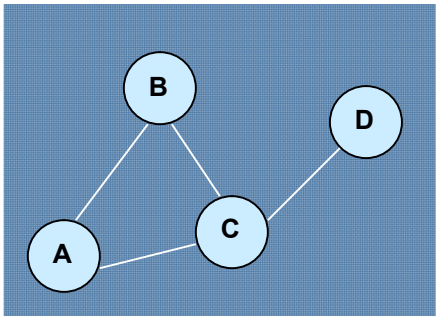
Grafo Dirigido

	A	B	C	D
A	F	V	V	F
B	V	F	V	F
C	V	V	F	V
D	F	F	V	F

Matriz de Adyacencia de un Grafo dirigido

6.7 Representación de un grafo no dirigido con una matriz de adyacencia

Cuando un grafo es no dirigido, la matriz de adyacencia es simétrica

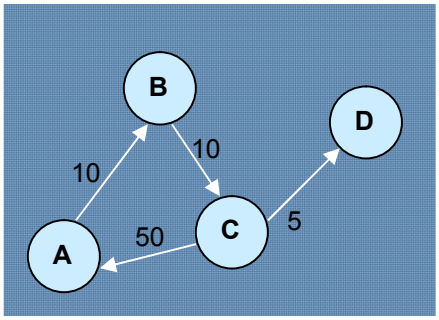


Grafo No Dirigido

	A	B	C	D
A	F	V	F	F
B	F	F	V	F
C	V	F	F	V
D	F	F	F	F

Matriz de Adyacencia de un Grafo Dirigido

6.8 Representación de un grafo valorado con una matriz de adyacencia



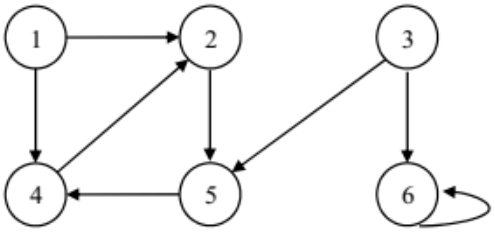
Grafo Valorado

	A	B	C	D
A	0	10	50	0
B	10	0	10	0
C	50	10	0	5
D	0	0	5	0

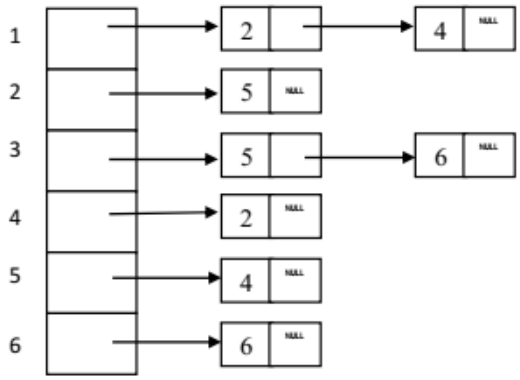
Matriz de Adyacencia de un grafo valorado

6.9 Representación de un grafo con una lista de adyacencia.

Este método se emplea principalmente para representar grafos que tienen muchos vértices y pocas aristas. Se utiliza una lista enlazada por cada vértice v del grafo que tenga otros adyacentes desde él. El grafo completo incluye dos partes: un directorio y un conjunto de listas enlazadas. Hay una entrada en el directorio por cada nodo del grafo. La entrada en el directorio del nodo i apunta a una lista enlazada que representa los nodos que son conectados al nodo i.



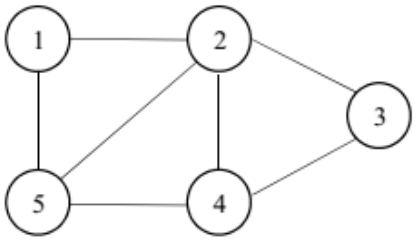
Grafo Dirigido



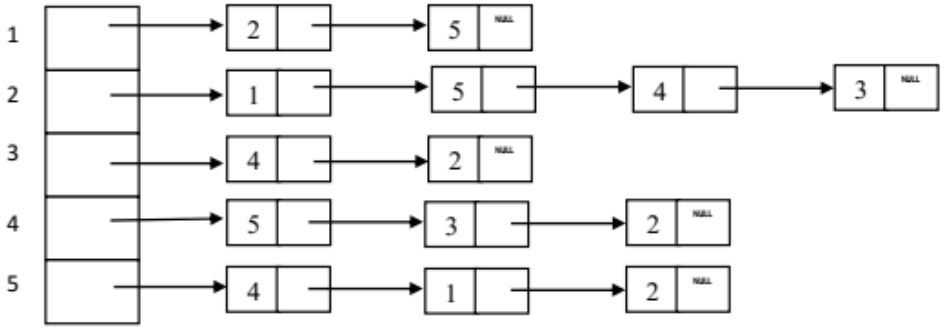
Representación como lista de adyacencias

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Representación como matriz de adyacencias



Grafo No Dirigido



Representación como lista de adyacencias

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Representación como matriz de adyacencias

Implementación de un grafo no dirigido en C

```
#include <stdio.h>
#include <stdlib.h>

struct nodo
{
   int dest;
   struct nodo* sig;
};


struct listaAdy
{
   struct nodo *inicio;
};


struct grafo
{
   int cantV;
   struct listaAdy* array;
};


struct nodo* creaNodo(int dest)
{
   struct nodo* nuevo = (struct nodo*) malloc(sizeof(struct nodo));
   nuevo->dest = dest;
   nuevo->sig = NULL;
   return nuevo;
}

struct grafo* creaGrafo(int cantidad)
{ int i;
   struct grafo* g = (struct grafo*) malloc(sizeof(struct grafo));
   g->cantV = cantidad;
   g->array = (struct listaAdy*) malloc(cantidad * sizeof(struct listaAdy));
   for (i = 0; i < cantidad; i++)
      g->array[i].inicio = NULL;

   return g;
}


void agregaArco(struct grafo* graph, int origen, int dest)
{

   struct nodo* nuevo;

   nuevo = creaNodo(dest);
   nuevo->sig = graph->array[origen].inicio;
   graph->array[origen].inicio = nuevo;


   nuevo = creaNodo(origen);
   nuevo->sig = graph->array[dest].inicio;
   graph->array[dest].inicio = nuevo;
```

```
}

void muestraGrafo(struct grafo* g)
{
   int v;
   struct nodo* lista;
   for (v = 0; v < g->cantV; v++)
   {
      lista= g->array[v].inicio;
      printf(" Lista de adyacencia del vertice %d\n inicio ", v);
      while (lista)
      {
         printf("-> %d", lista->dest);
         lista = lista->sig;
      }
      printf("\n");
   }
}

int main()
{

   int vertices= 5;
   struct grafo* g;

   g=creaGrafo(vertices);

   agregaArco(g, 0, 1);
   agregaArco(g, 0, 4);
   agregaArco(g, 1, 2);
   agregaArco(g, 1, 3);
   agregaArco(g, 1, 4);
   agregaArco(g, 2, 3);
   agregaArco(g, 3, 4);


   muestraGrafo(g);

   return 0;
}
```

Bibliografía

1. **Algoritmos Computacionales Introducción al análisis y diseño.**
   Sara Baase, Allen Van Gelder,
   Addisson Wesley, 2002

2. **Estructuras de Datos en C**
   Yedidyah Langsam, Moshe J. Augenstein, Aaron Tenenbaum.
   Prentice Hall, 1993

3. **Algorithmics. The spirit of computing**
   David Harel
   Addison-Wesley, 1987

4. **Introduction to Algorithms**
   Thomas H. Cormen, Charles E. Leiserson
   Ed. McGraw Hill, 1986

5. **Data Structures & Algorithm Analysis in Java**
   Mark Allen Weiss
   Addison-Wesley, 1998