UNIVERSIDAD NACIONAL ABIERTA
VICERRECTORADO ACADÉMICO
AREA: INGENIERÍA / CARRERA: INGENIERÍA DE SISTEMAS

GUIA INSTRUCCIONAL DE APOYO

NOMBRE: **COMPUTACION II**
Código: 324
U.C.**:** 4

CARRERA: Ingeniería de Sistemas /
TSU en Mantenimiento de Sistemas Informáticos

Código: 236 / 237

SEMESTRE: III

AUTOR: Lic.Maria Eugenia Mazzei,Ing. Nelly Escorcha (Especialista de Contenido)

COMITÉ TÉCNICO: MSc. Carmen Maldonado (Coordinadora de la Carrera)
Lic. Carmen Velásquez de Rojas (Evaluadora)
Dr. Antonio Alfonzo (Diseñador de la Instrucción)

Caracas, Marzo 2015

II. Introducción

El presente material tiene como propósito orientar al estudiante en la utilización del libro, específicamente en el manejo de los conceptos claves para la asimilación de los contenidos y en su aplicación práctica, que es fundamental para el logro del objetivo del curso. A través de él se hará énfasis en aquellos aspectos que se consideran significativos para la comprensión y la buena utilización de las estructuras de datos, bajo el enfoque de *tipos abstractos de datos*, así como de los algoritmos que operan sobre ellos, de acuerdo a las pautas establecidas para este curso y tomando como base el uso del libro texto seleccionado. Como se especifica en el Plan de Curso el libro recomendado es: *Estructuras de Datos en C++***,** de Joyanes L. y Zahonero I., Editorial Mc. Graw Hill y Programación en C++, de Joyanes L. y Sanchez L. Como texto alternativo se recomienda el siguiente libro: *Estructuras de Datos y Algoritmos*, de Aho A. V., Hopcroft J. E. y de Ullman J. D., Addison Wesley Longman.

El contenido está organizado siguiendo la estructura del curso: **Módulo I** (Estructuras Lineales de Datos), **Módulo II** (Estructuras No Lineales de Datos) y **Módulo III** (Métodos de Ordenación y de Búsqueda). Cada módulo comprende dos unidades y en cada unidad se intercalan secciones referentes a los conceptos estudiados, ejemplos que describen situaciones específicas, ejercicios en donde se incluyen interrogantes, llamadas a reflexión, observaciones o aclaraciones que tengan lugar y al final de cada unidad según se requiera se incluyen algunas referencias a páginas W*eb*.
Para el buen uso de este material se recomienda tener a la mano el Plan de Curso de la asignatura y el texto seleccionado; estos dos instrumentos conjuntamente con la Guía Instruccional de Apoyo integran esencialmente el paquete instruccional concebido para facilitar el logro de los objetivos.

Iconos empleados en el material instruccional

A lo largo de la lectura de este material encontrará diversos iconos, cuyo significado se explica a continuación.

	**AMPLIACIÓN DE CONOCIMIENTOS:** está dirigido al estudiante que desea profundizar más en sus conocimientos en determinado tema.
	**ATENCIÓN:** se presenta cuando se quiere hacer una aclaratoria, una advertencia o una reflexión sobre algún aspecto del contenido.
	**CASO DE ESTUDIO:** es la exposición de una situación muy similar a la realidad a la cual se le dará solución.
	**CONSULTA EN LA WEB:** indica referencias a páginas Web.
	**CONSULTA EN OTROS LIBROS:** se refiere a un llamado a consulta en libros que no figuran como textos de carácter obligatorio para el curso.
	**EJERCICIOS Y ACTIVIDADES PROPUESTAS:** son ejercicios o actividades sugeridas a manera de práctica sobre algún tema de la unidad.
	**EJERCICIOS DE AUTOEVALUACIÓN:** ejercicios que debe realizar el estudiante y posteriormente verificar contra los resultados aquí presentados.
	**EJEMPLO:** es la exposición de un caso alusivo al tema en cuestión y su resolución.
	**RECORDATORIO:** indica algún aspecto a ser enfatizado, relacionado con los conocimientos adquiridos previamente por el estudiante.
	**LECTURAS:** indica un texto de carácter obligatorio para la consecución de los objetivos del curso.
	**OBJETIVO:** indica la finalidad de la unidad, que es lo que se quiere lograr con el estudio de la misma.
	**NOTAS MATEMÁTICAS:** son informaciones complementarias sobre el tema que se está tratando.
	**RESPUESTA A LOS EJERCICIOS DE AUTOEVALUACIÓN:** presenta la clave de respuestas a los ejercicios de auto evaluación, de manera que puedas reforzar tus conocimientos o corregir cualquier error.
	**TIEMPO ESTIMADO DE ESTUDIO:** lapso que se considera suficiente para el aprendizaje del objetivo propuesto.

Los términos resaltados en letra negrilla, internamente en los párrafos, indican características, tipos, operaciones, procedimientos o funciones que son de gran importancia en el ámbito tratado. Las porciones de texto que contienen codificaciones en lenguaje de programación se presentan con sombreado, en este caso las palabras en negrilla son palabras reservadas del lenguaje. En las codificaciones se incluyen llamadas cuando se intenta destacar el significado del parámetro, variable o instrucción.

III. Objetivo del Curso

	Codificar algoritmos con sentido lógico y coherente, utilizando las estructuras de datos apropiadas, así como los métodos de clasificación y búsqueda, aplicados a la resolución de problemas específicos

MÓDULO I

Tipos abstractos de datos y estructuras lineales de datos

Este módulo comprende el estudio de las abstracciones de datos y las estructuras lineales de datos, estas últimas integradas bajo el concepto de TAD. Los tipos abstractos de datos (TAD), se asemejan a los objetos con sus métodos y su uso representa un avance dentro del ámbito del desarrollo de programas orientados hacia la aplicación de la Metodología de Orientación a Objetos. Entre las ventajas que conlleva el uso de esta metodología están la reusabilidad, la eficiencia, la confiabilidad y la adaptabilidad, las cuales conducen al mejoramiento de la productividad del desarrollador.

	**Objetivo Modulo I** Codificar con sentido lógico y coherente, algoritmos en lenguaje de programación, empleando Tipos Abstractos de Datos y/o estructuras lineales de datos apropiadas, para resolver problemas específicos

Estructura del Módulo I

Unidad 1: Tipos Abstractos de Datos Asociados a Estructuras Lineales de Datos.

Unidad 2: Estructuras lineales de datos - Listas

Unidad 3: Estructuras lineales de datos – Colas y Pilas

UNIDAD 1

Tipos Abstractos de Datos Asociados a Estructuras Lineales de Datos.

Abstracción. Consiste en ignorar los detalles de la manera particular en que esta hecha una cosa, quedándonos solamente con su visión general.

Para afianzar lo estudiado en el libro texto sobre estructuras de datos lineales, se presentan algunas recomendaciones incluyendo un conjunto de actividades antes de proseguir a analizar los ejemplos aquí presentados.

Se define como un conjunto de valores que pueden tomar los datos de ese tipo, junto a las operaciones que los manipulan.

Objetivo de la Unidad I

Elaborar un TAD (Tipo Abstracto de Datos) para representar un dato particular

Contenido de la Unidad 1: La abstracción. Modularidad. Tipos abstractos de datos. Uso de tipos de datos y estructuras de datos fundamentales: cadenas, arreglos simples y multidimensionales, registros y conjuntos, punteros o apuntadores. Orientación a objetos: conceptos.

Recomendaciones para el estudio de la Unidad

- **Leer** *Tipos abstracto de datos*. Los mismos están dirigidos fundamentalmente hacia el conocimiento del software algorítmico, pseudocódigo, definición de los tipos abstracto de datos y el concepto de objeto, estos conceptos le permitirán la comprensión del lenguaje y la técnica de programación adecuada para la solución de problemas a través de la computadora.

- **Estudiar** el concepto de modularidad en el ámbito del diseño de programas. Una vez leído el contenido citado, responda lo siguiente: ¿Cuáles son las ventajas que proporciona la modularidad? ¿Qué facilidades provee el LENGUAJE C++ en este sentido?

- **Repasar** los tipos de datos del Lenguaje de Programación LENGUAJE C++

Dato	Tipo	Significado
Int	entero	Números enteros
char	carácter	Caracteres y símbolos especiales
float	coma flotante	Son números que tienen una parte fraccional.
double	coma flotante de doble precisión	Rango superior normalmente de 64 bits, 8 bytes ó 4 palabras, con un rango de 1, 7E-308 a 1, 7E+308).
long double	doble precisión largos	tamaño de 80 bits ó 5 palabras, con un rango de+/-1,18E-4932 a 1,18E-4932
void		ocupan cero Bits o creación de punteros genéricos
dato bool	booleano	Se le puede asignar las constantes true (verdadero) y false (falso).

- **Repasar** las definiciones de tipo de dato enumerado y la función template en C++.

- **Estudiar** los capítulos del libro Luis Joyanes “Abstracción de datos y objetos” y “Abstracción de Control”.

- **Estudiar (**para una mejor comprensión sobre estructuras de datos) las abstracciones, las abstracciones procedimentales y las abstracciones de los datos. Responda a las siguientes preguntas: ¿Qué son tipos abstractos de datos? ¿Para qué sirven?

- **Estudiar** (para el correcto manejo del lenguaje C++) los conceptos relativos a la Metodología Orientada a Objetos.

A continuación se presentan algunos ejemplos que permiten ilustrar los conceptos aprendidos.

Ejemplo 1.1 (Tipo de Datos)

Se tienen cadenas de caracteres que se refieren a los nombres de asignaturas cuya longitud no excede los 35 caracteres. ¿Qué tipo de dato podemos asociarle para representarlas?

Podemos declarar estas cadenas en C++, empleando el tipo de dato char, de la siguiente manera:

```
Char Nom_ASIG[35]
```

Con las cadenas de caracteres se pueden realizar muchas operaciones como por ejemplo:

- Concatenar cadenas (pegar cadenas).
- Extraer alguna subcadena de una cadena dada
- Eliminar alguna subcadena de una cadena dada
- Determinar la longitud de una cadena

Dado el caso que la variable M contenga el valor: 'Introduccion a la Ingeniería de Sistemas", se quiere extraer una subcadena de diez caracteres a partir de la posición 19. En C++ se hace de la siguiente manera:

```
string M="Ingeniería de Sistemas";
string subcadena(M,0,10);
```

La cadena de texto string subcadena, permite extraer la subcadena "Ingeniería" de la variable tipo string, M.

Ejemplo 1.1 (Tipo de Datos)

Suponga que se tiene una matriz 3x3, de números reales. Elabore un TAD para este tipo de dato. Consideremos el tipo de dato matriz bidimensional (arreglo bidimensional), sobre la cual podemos definir varias operaciones, como se representa en el siguiente gráfico:

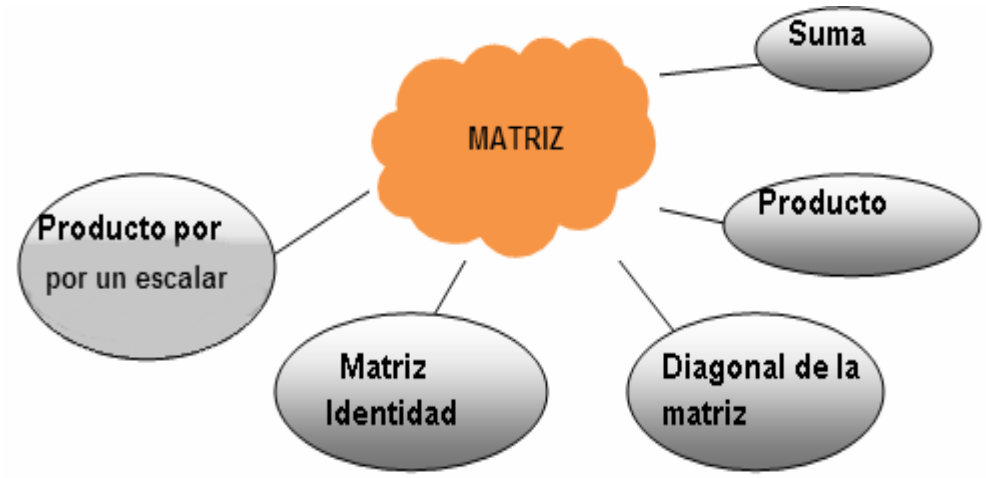


Figura 1.1. Matriz Bidimensional

A continuación describimos las operaciones señaladas:

La operación **Suma**, actúa sobre dos matrices cuadradas y permite obtener la matriz suma de los elementos de ambas matrices.

La operación **Producto** actúa sobre dos matrices cuadradas y permite obtener la matriz producto de ambas matrices. Al respecto es importante considerar también el caso del producto de matrices que no sean cuadradas, respetando el orden que deben tener las matrices para que sea posible obtener el producto de ellas.

La operación **Diagonal de una Matriz**, extrae la diagonal de una matriz cuadrada y la coloca en un arreglo.

La operación **Matriz Identidad**, genera una matriz identidad cuadrada (posee ceros en todas las entradas excepto en la diagonal en donde hay unos).

La operación **Producto por un Escalar**λ, obtiene la matriz resultante de multiplicar cada elemento de una matriz cuadrada por el valorλ.

	Para un TAD con un comportamiento determinado pueden existir diferentes implementaciones

Definiremos el tipo de datos en C++ de la siguiente manera:

```
Int i,j,k, a[d][d],b[d][d],c[d][d];
```

Ahora definiremos la operación **producto de dos matrices de 3x3**

```
#include <iostream>
#define d 3
using std::cout;
using std::cin;
using std::endl;

int main(){
   int i, j, k, a[d][d], b[d][d], c[d][d];
   cout << "MATRIZ A." << endl; // Introduce los elementos de la matriz A
   for(i = 0 ; i < d ; i++){
      for(j = 0 ; j < d ; j++){
    cout << "Introduzca el valor del elemento [" << i << "]["
     << j << "]: ";
    cin >> a[i][j];
 }
   }
   cout << endl;
   for(i = 0 ; i < d ; i++){ // Imprime los elementos de la matriz A
      for(j = 0 ; j < d ; j++){
         cout << a[i][j] << " ";
         if(j == 2)
            cout << endl;
   }
   }
   cout << endl;
   cout << "MATRIZ B." << endl; // Introduce los elementos de la
   matriz B
   for(i = 0 ; i < d ; i++){
      for(j = 0 ; j < d ; j++){
         cout << "Introduzca el valor del elemento [" << i << "]["
               << j << "]: ";
         cin >> b[i][j];
      }
   }
   cout << endl;
   for(i = 0 ; i < d ; i++){ // Imprime los elementos de la matriz B
      for(j = 0 ; j < d ; j++){
         cout << b[i][j] << " ";
         if(j == 2)
            cout << endl;
      }
   }
 for(i=0;i<d;i++){ /* Realiza el producto de matrices y guarda
                  el resultado en una tercera matriz*/
    for(j=0;j<d;j++){
```

```
        c[i][j]=0;
        for(k=0;k<d;k++){
           c[i][j]=c[i][j]+(a[i][k]*b[k][j]);
        }
    }
}
cout << endl << "MATRIZ C (Matriz A*B)." << endl;
cout << endl;
for(i=0;i<d;i++){ // Imprime la matriz resultado
    for(j=0;j<d;j++){
        cout << c[i][j] << " ";
        if(j==2)
          cout << endl;
    }
}
system("PAUSE");
return 0;
}
```

Igualmente podemos codificar las demás operaciones, algunas de las cuales actúan sobre dos matrices y otras sobre una matriz.

Recuerde que la clave en la abstracción de los datos es incluir en único paquete los datos y el código.

Ejercicios y Actividades Propuestas

- Investigue qué funciones provee C++ para facilitar la implementación de las operaciones mencionadas.
- Indique cómo se expresan en C++ las variables tipo apuntador para su debida utilización en direcciones de memoria.
- Determine las ventajas de emplear estructuras dinámicas en un programa.
- Precise cuáles operaciones pueden hacerse con este tipo de variables y cómo se implementan en C++.
- Implemente las operaciones de Suma, Diagonal de una Matriz, Matriz Identidad y Producto de una Matriz por un escalar.
- Investigue acerca de otras operaciones de matrices que se podrían implementar.
- Investigue qué son las **librerías** en C++ ¿Cuáles son las ventajas que proporcionan? ¿Cómo se implementan?

Investigue sobre los lenguajes de programación Orientados a Objetos. Igualmente investigue sobre los lenguajes imperativos y los mixtos.

Esta actividad le permitirá una mejor comprensión sobre el Lenguaje C++, al tener la información de cómo funciona la programación orientada a objetos, y distinguirlo como un lenguaje imperativo y/o mixto.

UNIDAD 2

Estructuras lineales de datos

Para la asimilación del contenido de esta unidad se sugiere seguir paso a paso las recomendaciones generales que se exhiben en esta sección. Es importante recordar que la realización de ejercicios y su implementación en el lenguaje de programación son de suma importancia para la asimilación de los contenidos.

Los temas a estudiar en esta unidad resultan muy intuitivos. Es fácil observar en la vida real, series de objetos enlazados, la formación de colas: de personas, de trabajos y de objetos en general. Igualmente es frecuente acumular pilas de papeles o documentos, en donde el elemento que está en la cima de las mismas es el de más reciente colocación. Piense en agrupaciones de este tipo y cómo operarlas.

Objetivo de la Unidad 2
Resolver problemas de tipo algorítmico aplicando el TAD Lista Enlazada

Contenido: Especificación formal del TAD Lista. Implementación del TAD Lista con estructuras dinámicas. Operaciones. Aplicaciones. Listas doblemente enlazadas. Listas circulares.Especificación formal del TAD Pila. Implementación de Pilas con Arreglos. Implementación de Pilas con variables dinámicas. Aplicaciones. Especificación formal del TAD Cola y Dipolo o Bicola. Implementación del TAD Cola o Dipolo con estructuras estáticas y/o dinámicas. Operaciones. Aplicaciones.

Recomendaciones para el estudio del contenido de la Unidad

- **Estudiar** en el capítulo del libro “Programación en C++: punteros”. Realizar los problemas que están en el libro. ¿Cómo se definen estos tipos de datos en C++?

- **Estudiar** en el libro I, los capítulos: “Listas” y “Listas doblemente enlazadas”, la estructura LISTA o en el libro II, las secciones sobre “Listas enlazadas”. Determine las operaciones que pueden definirse sobre ellos.

- **Estudiar** el capítulo del libro I titulado: “Pilas”, la estructura PILA o en el libro II, las secciones relacionadas con PILAS. Determine las operaciones que pueden definirse sobre ella.

- **Estudiar** en el capítulo del libro titulado: “Colas: el TAD COLA”, o el libro II, las secciones relacionadas con concepto de COLAS.

- **Realizar** los ejercicios propuestos en el libro.

- **Implementar** las soluciones de los problemas resueltos en C++ y emplear el computador para realizar las pruebas. Es muy importante emplear el estilo de programación modular en el desarrollo de programas. Elabore sus procedimientos y funciones con parámetros apropiados, esto facilitará la reusabilidad de los mismos.

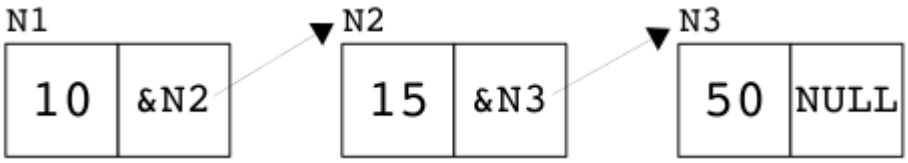
  A continuación se presentan algunos ejemplos que permiten ilustrar los conceptos aprendidos.

Listas

Las listas enlazadas son estructuras dinámicas de datos. Son ejemplos de listas lineales enlazadas, conjuntos de elementos que tienen alguna relación por ejemplo de distancia, de posición o una clave con respecto a la cual se mantiene un orden determinado.


Ejemplo 2.1 (listas enlazadas)

Implementación de una lista enlazada simple y sus operaciones básicas tales como recorrer, insertar, eliminar y buscar.



Las listas simples están formadas por nodos con un solo enlace. Este enlace hace referencia al siguiente nodo de la lista o a NULL si es el último nodo. NULL es un valor que representa que la referencia esta vacía. Debido a que poseen una sola referencia el recorrido de las listas simples es en una sola dirección: del primero hacia el último. No es posible ir de un nodo al nodo anterior.

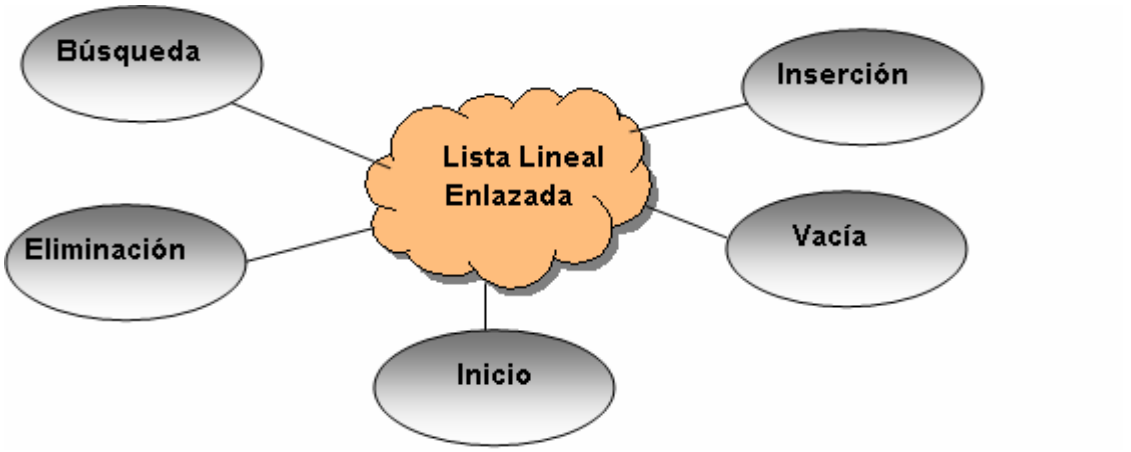


Figura 2.2 Lista lineal Enlazada

Los procedimientos indicados en la Figura 2.2 son los siguientes:

Inserción: Consiste en agregar un nuevo nodo a la lista. La inserción normal se hace al final de la lista. Cuando se trata de una **inserción ordenada** hay que recorrerla y comparar contra la información de la clave de cada elemento para determinar dónde insertar.

Vacía: Es una función que determina si la lista está vacía o no.

Inicio: Inicia una lista, por ejemplo asignándole valores nulos a los apuntadores.

Eliminación: Consiste en eliminar un nodo específico de la lista.

Búsqueda: Busca información en una lista.

Tanto la **inserción** como la **búsqueda** en una lista lineal enlazada implican que debe recorrerse dicha lista Existen otros procedimientos o funciones que se podrían implementar como **Avanza**, el cual permitiría avanzar al nodo siguiente de la lista. Otra función que es importante implementar es aquella que determina si un elemento está a la cabeza de la lista o es interno y en algunos casos si se trata del último elemento.

Para implementar una lista primero debemos definir su nodo.

```
struc nodo{
int data;//almacena la data (un entero en este caso)
nodo *sgte; //almacena la dirección de otro nodo
};
```

Las operaciones sobre una lista las implementamos mediante funciones que reciben como parámetro la cabecera de la lista sobre la que queremos trabajar entre otros parámetros dependiendo de la operación. Ahora veamos las operaciones básicas que se pueden realizar sobre una lista enlazada simple.

Recorrido de la lista.
Consiste en ir desde el primer elemento de la lista hacia el último. El propósito de esta operación es variado, se puede usar por ejemplo para contar cuantos elementos tiene la lista o para mostrar la data de todos los elementos.

```
//Función para recorrer una lista y mostrar la data.
void recorrer(nodo *cab){
 //verifica si la lista esta vacía
 if(cab == NULL){
  //si esta vacía muestra este mensaje
    cout<<"La lista esta vacia."<<endl;
 }
 else{
  //si no esta vacía declara un puntero p al primer elemento de la lista
  nodo *p = cab;
```

```
    do{
   //muestra la data del nodo referido por p
    cout<<"Elemento: "<<p->data<<endl;
   //asigna a p la dirección de nodo siguiente o NULL si no lo hay
   p = p->sgte;
   //repite el proceso mientras p sea diferente de NULL
  }while(p != NULL);
    cout<<"Fin del recorrido."<<endl;
 }
}
```

Insertar un elemento a la lista

La inserción de un elemento puede hacerse de tres maneras:

- Inserción por el final,
- inserción por el inicio
- inserción en la posición enésima.

En todas la idea es crear un nuevo nodo con la data que se recibe por parámetro y enlazarlo con la lista. Veamos por ejemplo la inserción por el final.

Inserción por el final

```
//Función que inserta un nodo nuevo al final de la lista
void insertarFinal(nodo **pcab, int data){
 //crea un nuevo nodo dinámicamente e inicia sus campos
 nodo *nuevo = new nodo;
 nuevo->data = data;
 nuevo->sgte = NULL;

 //valida si la lista esta vacia (pcab apunta a la cabecera osea pcab==&cab)
 if(*pcab == NULL){
  //modifica la cabecera para que apunte al nuevo nodo
  *pcab = nuevo;
 }
 else{
  //declara un puntero p que apunta al primer nodo (p==cab)
  nodo *p = *pcab;
  //avanza p hasta que llega al ultimo nodo (el que tiene NULL en sgte)
  while(p->sgte != NULL){
   p = p->sgte;
  }
  //asigna la dirección del nuevo nodo al campo sgte del ultimo nodo
  p->sgte = nuevo;
 }
}
```

Ejercicios y actividades propuestas

- Implemente los procedimientos mencionados para el tipo lista.
- Compare la operación en listas lineales enlazadas con respecto a las implementadas en estructuras contiguas.
- Realice la inserción por el inicio e inserción en la posición enésima en la lista.

UNIDAD 3

Estructuras lineales de datos- Pilas y/o Colas

Las pilas son estructuras de datos que tienes dos operaciones básicas: Push (para insertar un elemento) y pop (para extraer un elemento).Su característica fundamental es que al extraer se obtiene siempre el último elemento que acaba de insertarse. Por esta razón también se conocen como estructuras de datos LIFO (del inglés Last In First Out).

Objetivo de la Unidad 3
Resolver problemas de tipo algorítmico aplicando el TAD Pila y/o el TAD Cola

Contenido: Especificación formal del TAD Pila. Implementación de Pilas con Arreglos. Implementación de Pilas con variables dinámicas. Operaciones Aplicaciones. Especificación formal del TAD Cola y Dipolo o Bicola. Implementación del TAD Cola o Dipolo con estructuras estáticas y/o dinámicas. Operaciones. Aplicaciones.

Recomendaciones para el estudio del contenido de la Unidad

- **Estudiar** el capítulo del libro I titulado: “Pilas”, la estructura PILA o en el libro II, las secciones relacionadas con PILAS. Determine las operaciones que pueden definirse sobre ella.

- **Estudiar** en el capítulo del libro titulado: “Colas: el TAD COLA”, o el libro II, las secciones relacionadas con concepto de COLAS.

- **Realizar** los ejercicios propuestos en el libro.

- **Implementar** las soluciones de los problemas resueltos en C++ y emplear el computador para realizar las pruebas. Es muy importante emplear el estilo de programación modular en el desarrollo de programas. Elabore sus procedimientos y funciones con parámetros apropiados, esto facilitará la reusabilidad de los mismos.

A continuación se presentan algunos ejemplos que permiten ilustrar los conceptos aprendidos.

Ejemplo 3.1

Se quiere determinar si una cadena de caracteres es un PALINDROMA
Utilizar para ello el TAD pila de caracteres.
Una palabra es palindroma si se escribe igual al derecho y al revés. Ejemplo, la Palabra: ORO es palindroma.
Antes de resolver el problema planteado, analizaremos el TAD pila.
La pila es una estructura de datos que se mantiene y opera bajo la disciplina LIFO (el primero que entra es el último que sale). Son ejemplos de ellas: una pila de documentos y una pila de platos, en los cuales se agregan y se quitan elementos en la cima de la misma. Las expresiones aritméticas se evalúan empleando una pila.
En la siguiente figura se representa gráficamente una pila:

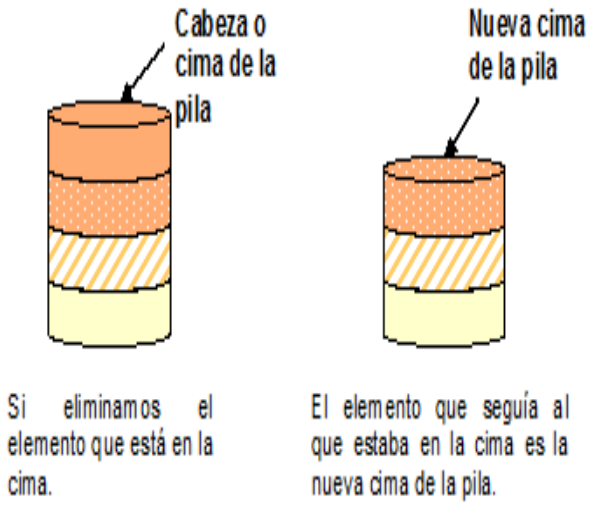


Figura 3.1 Representación de una Pila

:

Básicamente se puede asociar a un TAD pila las siguientes operaciones:

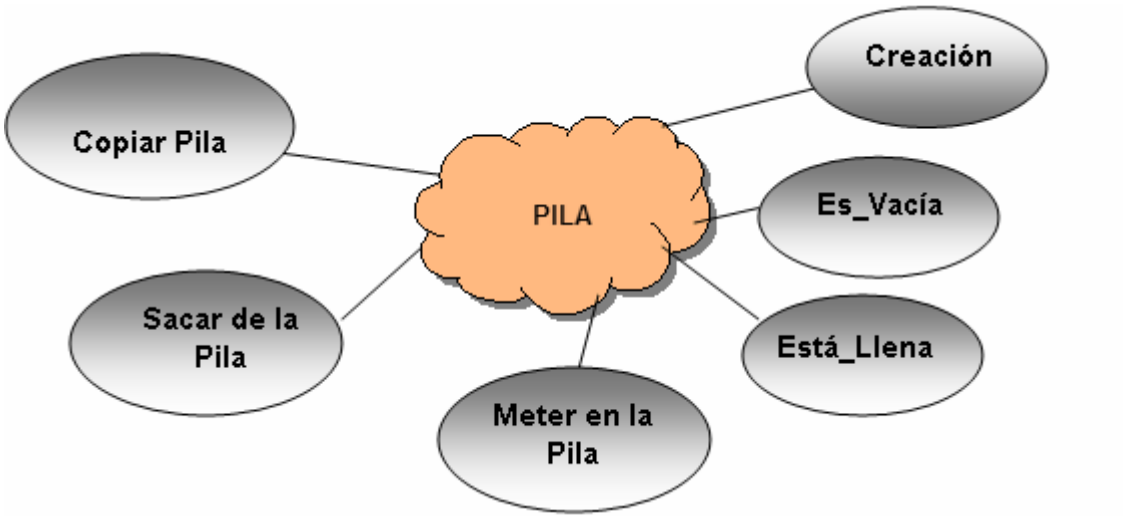


Figura 3.2 Pilas

El procedimiento de **Creación** generalmente crea una pila vacía, ó sea que el apuntador a la cima de ella es nulo. La operación **Es Vacía** permite determinar si la pila está vacía o no. La operación **Está Llena** se emplea para determinar si una pila está o no llena, esta función es útil cuando se tienen pilas limitadas. La operación **Apilar** (*Push* en inglés), inserta un elemento en la pila. La operación **Desapilar** (*Pop* en inglés), extrae el último elemento apilado. Finalmente la operación **Copiar Pila**, reproduce o copia una pila en otra pila.
Para su ilustración veamos un ejemplo en C++:

```
   struct Pila   //creando nodo tipo  puntero (tipo de dato)

{
   char dato;
   struct Pila *Sig;
};

typedef struct Pila *Ap;
```

A continuación presentamos el procedimiento que permite meter un elemento en la pila (apilar):

```
void push(Ap *Tope, char pLetra)
{
   struct Pila *x;
   x = new struct Pila;     // apuntamos al nuevo nodo creado
   x->dato = pLetra;
   x->Sig = *Tope;
   *Tope = x;
}
```

El siguiente procedimiento permite sacar el elemento de la cima de la pila (desapilar):

```
char pop(Ap *Tope)
{
   char a;

   if(*Tope == NULL)
   {
      printf("PILA VACIA...");
      return 0;
   }
   else
   {
      a = (*Tope)->dato;
      *Tope = (*Tope)->Sig;       //Desapilar elemento(devuelve elemento)
      return (a);
   }
}
```

Una manera de determinar si una palabra es palíndroma, es descomponerla en caracteres, meter cada carácter en la pila y luego vaciar la pila y formar una nueva palabra, si ambas son iguales la palabra es palíndroma.

El procedimiento para determinar si la palabra es palíndromo o no, es el siguiente:

```
void Palindromo()
{
   char frase[30], invert[30], letra[20];
   int j = 0;
   Ap Tope;
   Tope = NULL;

   fflush(stdin);
   printf("Escriba una palabra: \n");

   gets(frase);
```

```
   for(int i=0; i<strlen(frase); i++)
   {
      if(frase[i] != 32)
      {
         letra[j] = frase[i];
         push(&Tope, frase[i]);

         j++;
      }
   }

   display(Tope);

   for(int k = 0; k<j; k++)
   {
      invert[k] = pop(&Tope);
   }

   if(strncmp (letra,invert,strlen(invert)) == 0) //strncmp (letra,invert,strlen(invert))
== 0 || strcmp(letra, invert) == 0
      printf("Si es un palindromo\n");
   else
      printf("No es un palindromo\n");

   getch();
}
```

 Está claro que en el ejemplo presentado se sugiere emplear una pila para que el estudiante se familiarice con las operaciones de una pila. Pudo hacerse sólo con operaciones de datos tipo *string*.

 Ejercicios y actividades propuestas

Halle otro tipo de solución al problema planteado, por ejemplo utilice una pila para copiar la pila original y otra para vaciar la pila original. Al comparar elemento a elemento, realizando operaciones de extracción en ambas pilas, se puede determinar si una palabra es palíndromo.

Colas

Las colas son estructuras que se forman y operan bajo la disciplina FIFO (el primer elemento que entra es el primero que sale). Son ejemplos de colas las líneas de espera en los bancos, supermercados, taquillas de pagos, de boletería, los autolavados, etc.
Una abstracción de la cola es la siguiente:

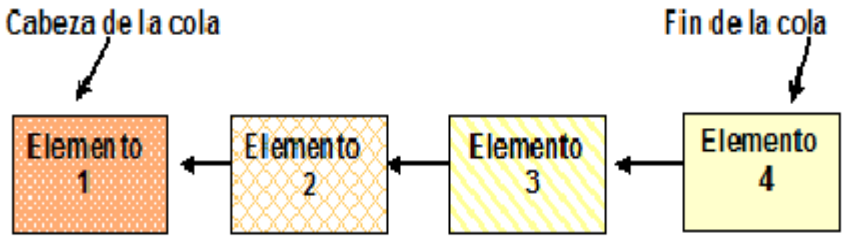


Figura 3.3

Las inserciones se realizan al fin de la cola y las eliminaciones a la cabeza de la cola, por lo tanto se debe mantener dos apuntadores, uno a la cabeza y otro al fin de la cola. El siguiente ejemplo ilustra una situación común de estructura de colas.

Ejemplo 3.2 (colas)

Se requiere manejar una estructura de colas de documentos, con la siguiente característica: se pueden insertar documentos por la izquierda o por la derecha. Pero sólo se puede retirar documentos por la izquierda, como se muestra en la Figura 2.6.
Sobre la base de esta situación, elabore un procedimiento en C++, denominado **Insercion_Especial** que permita manejar el proceso de inserción en este tipo de cola, empleando manejo dinámico de memoria. Describa las estructuras de datos en PASCAL, considerando que de cada documento se registran los siguientes datos:

- Nombre del Documento
- Fecha (*dd/mm/aa*)

El procedimiento de inserción recibe como parámetros los datos del documento, además si es de carácter normal o urgente. Según sea el carácter del documento se realizará la inserción apropiada.

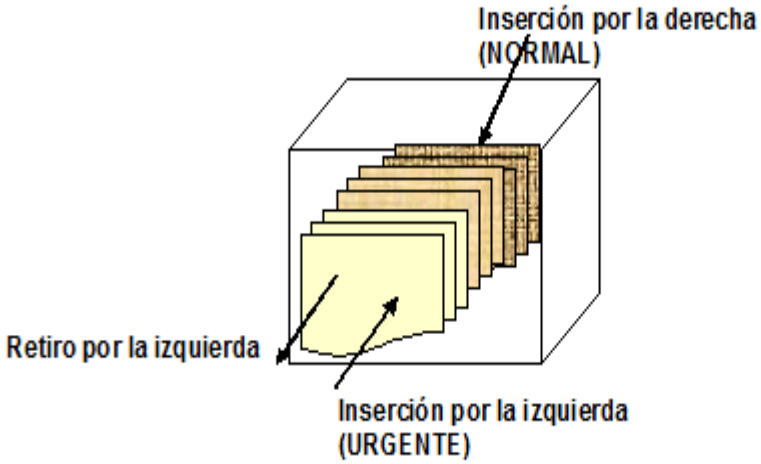


Figura 2.6 Insercion_Especial

Fig 3.4

Al analizar este problema podemos observar que se tratar de una bicola o dipolo restringido a que sólo pueden hacerse eliminaciones por un extremo o cabeza de la cola. Hay dos tipos de inserciones: normales al final de la cola y urgentes en la cabeza de la cola.
En general podemos definir un conjunto básico de operaciones que se realizan sobre una estructura de datos de este tipo, como se muestra en la Figura 2.7

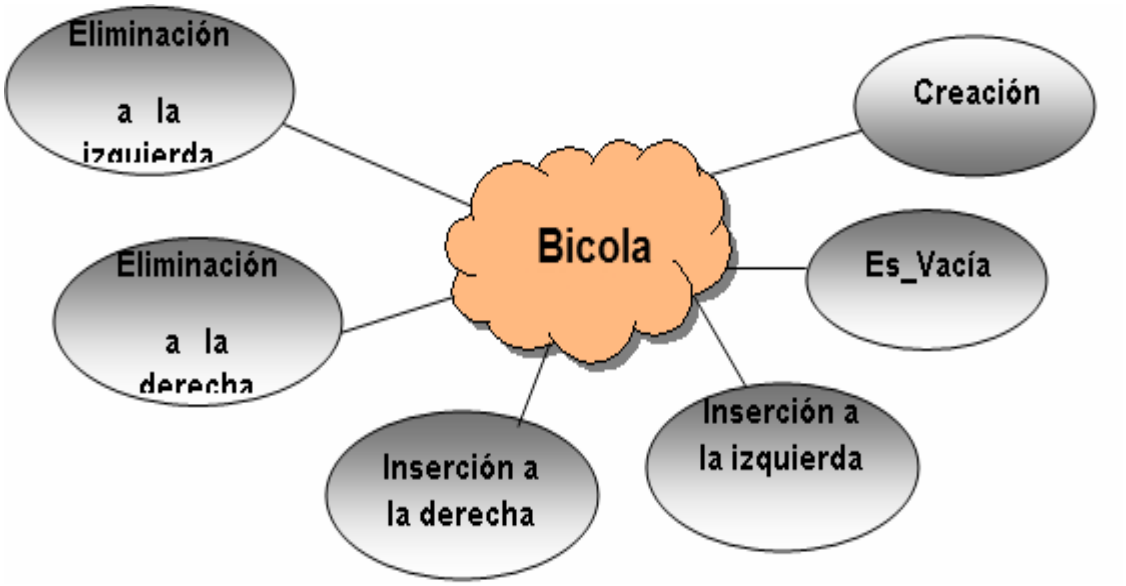


Figura 3.5 Bicola

Para la situación dada no se necesita realizar un procedimiento de eliminación por la derecha, ya que el retiro de documentos se hace siempre por la izquierda. El procedimiento **Inserción_Especial** solicitado en las especificaciones del problema, permitirá Insertar a la derecha o a la izquierda según el tipo de documento. A continuación presentamos un ejemplo del procedimiento de

Inserción Especial, requerido para resolver el problema de la inserción en un dipolo con las características dadas; previamente describimos el nodo de la cola en C++:

```
/*              Estructura de los nodos de la cola
--------------------------------------------------------------------*/
struct nodo

{
   int dato;
   struct nodo *sgte;
};

/*                   Estructura de la cola
--------------------------------------------------------------------*/
struct cola
{
   nodo *delante;
   nodo *atras  ;
};
```

Inserción_Especial

```
int main()
{
   struct cola q;

   q.delante = NULL;
   q.atras   = NULL;

   int c  ;  // caracter a encolar
   char x ;    // caracter que devuelve la funcion pop (desencolar)
   int op;     // opcion del menu
   int pos;    // posicion de insertar  (inicio o fin)

   do
   {
      menu();  cin>> op;

      switch(op)
      {
         case 1:
            cout<< "\n Ingrese documento: ";
            cin>> c;

            cout<<"\n\t[1] Inserta al inicio Urgente" <<endl;
            cout<<"\t[2] Inserta al final  Normal" <<endl;
            cout<<"\n\t    Opcion : ";
            cin>> pos;
```

```
            encolar( q, c, pos );

            cout<<"\n\n\t\tNumero '" << c << "' encolado...\n\n";

          break;

          case 2:
            cout << "\n\n MOSTRANDO COLA\n\n";

            if(q.delante!=NULL)
              muestraCola( q );
            else
              cout<<"\n\n\tCola vacia...!" << endl;

          break;
        }

       cout<<endl<<endl;
       system("pause");  system("cls");

    }while(op!=3);

    return 0;
}
```

```
/*                    Inserción a la izquierda
-------------------------------------------------------------------------*/
void encolar( struct cola &q, char x, int pos )
{
    struct nodo *aux = crearNodo(x);

    if( pos==1 )
    {
        if( q.delante==NULL)
        {
            aux->sgte = q.delante;
            q.delante = aux;
            q.atras   = aux;
        }
        else
        {
            aux->sgte = q.delante;
            q.delante = aux;
        }
    }
    else
    {
        if( q.atras==NULL )
```

```
        {
            aux->sgte = q.atras;
            q.delante = aux;
            q.atras   = aux;
        }
        else
        {
            aux->sgte = (q.atras)->sgte;
            (q.atras)->sgte = aux;
        }
    }
}
```

Ejercicios y actividades propuestas

- Implemente los procedimientos mencionados para el tipo bicola o dipolo, considerando el ejemplo presentado.

- Resuelva el problema 2.2 sobre la determinación de palabras palíndromos empleando una pila y una cola para almacenar los componentes de la palabra. Como la operación de eliminación en la cola tradicional es inversa con respecto a la eliminación en la pila, al extraer los caracteres en ambas estructuras se puede comprobar que una palabra es o no palíndromo.

Ejercicios de Repaso

1) Que tipo de dato se debe asignar a la siguiente operación: 2*2

2) Escriba una función scanf() que lea dos números enteros y número decimal:

3) Función que permite la impresión de datos

4) En el siguiente código seleccione en que fila hay errores:

```
 1) #include "stdafx.h"
 2) #include "stdlib.h"
 3) #include "conio.h"
 4) int _tmain(int argc, _TCHAR* argv[])
 5) int Li,Ls,rm,i,j;
 6) printf("ingrese el límite inferior\n");
 7) scanf("%d",&Li);
 8) printf("ingrese el limite superior\n");
 9) scanf("%d",&Ls);
10) i=Li;
11) while (i<=Ls)
12) system("cls");
13) j=1;
```

```
14) while(j<=12)
15) printf("%d * %d = %d\n",j,i,j*i);
16) j++;
20) system("pause");
21) i++;
22) return 0;
```

5) Escoja las sentencias de escape correctas
   a) \n
   b) \t
   c) \v
   d) \q
   e) \z
   f) \a
6) Que permite la instrucción include
7) Cuál es la Función que permite la impresión de datos
8) Dado int a=6;
   int b=4;
   int c=2;
   Sustituir los valores en las siguientes expresiones y dar su resultado:
   a) b%
   b) a+b+c
   c) (2*(a+b))-c*c
9) Marque las variables válidas(v) y las no válidas (nv):
   a) suma
   b) totalpagar
   c) m*|
   d) #volumen
   e) a_e
   f) 1ap
   g) peso base
10) Que tipo de sentencia es IF
11) Indicar las opciones correctasen las siguientes expresiones:
   a) scanf("%d",casa)
   b) printf("%i,%i",n,m)
   c) int x;
      x=4
   d) printf("%i,%i,%f,%c,n,m)
   e) scanf("%f",&p)
   f) char e
   g) e=2*9
   h) float a;
   i) a=34

Para reforzar lo aprendido en la unidad en relación con los tipos

abstractos de datos y estructuras lineales de datos puede consultar las siguientes direcciones electrónicas, tomando en cuenta que éstas podrían ser dadas de baja por sus autores en cualquier momento.

Estructuras de Datos: http://c.conclase.net/edd/index.php

Para repasar las instrucciones del lenguaje C++, consulte las direcciones:

http://www.cplusplus.com/doc/tutorial/

Objetivo Modulo II
Implementar algoritmos en lenguaje de programación, empleando estructura de grafo y árbol para la resolución de problemas específicos.

UNIDAD 4

La herramienta de la recursividad y estructuras de datos tipo árbol

En relación con el estudio de esta unidad, se sugiere igualmente seguir las recomendaciones indicadas. Paralelamente se recomienda intensificar las actividades prácticas, utilizando el computador como herramienta para realizar las pruebas de las abstracciones que Ud. diseñe.

Objetivo de la Unidad 4

Resolver problemas de tipo algorítmico aplicando el TAD Árbol

Contenido: Recursividad, Concepto. Utilidad. Algoritmos típicos. Implementación de procedimientos recursivos.
Concepto de árbol binario. Árboles de Expresión. Construcción. Recorrido. Aplicaciones. Árbol binario de búsqueda. Operaciones.

Recomendaciones para el estudio del contenido de la unidad

- **Estudiar** el capítulo del libro *Estructuras de Datos en C++***,**: "Algoritmos recursivos". Establezca la noción de recursividad, su importancia y los problemas recursivos. Determine cómo se implementan los procedimientos y las funciones recursivas en C++. Examine los ejemplos presentados en el libro.
- **Realizar** los ejercicios propuestos en el libro.
- **Estudiar** el capítulo del libro Estructuras de Datos en C++: "Árboles. Árboles binarios y Árboles ordenados. "Examine los ejemplos de árboles binarios de expresión, evaluación de expresiones, notación infijo a postfijo, recorridos en un árbol y árboles de búsqueda.
- **Realizar** los ejercicios presentados en el libro.
- **Implementar** en el computador problemas de recursividad
- **Implementar** en el computador problemas relacionados con árboles binarios.
- **Emplear** algoritmos recursivos de recorrido de árbol.

RECURSIVIDAD

La recursividad es una herramienta muy potente para resolver cierto tipo de problemas. No todos los problemas pueden ser resueltos de manera recursiva. Uno de los ejemplos típicos de recursión es la determinación de los términos de la serie de Fibonacci y el cálculo del factorial de un número natural. Un procedimiento recursivo o una función recursiva es aquella que se llama a si misma, pero ¿cuántas veces se auto invoca? Depende del problema, todo procedimiento o función recursiva debe tener una **condición de borde** o **condición de parada**, que se puede decir que es cuando el procedimiento "toca fondo" y una llamada a sí mismo o llamada recursiva. Cada invocación a un procedimiento ocasiona que se guarden los valores de las variables en una pila, esto es debido a que estas llamadas no son atendidas, sino hasta que se produce la condición de parada, que es cuando se asigna el valor establecido y se regresa a resolver las otras llamadas de acuerdo a la disciplina LIFO.
Casi todos los algoritmos basados en los esquemas de vuelta atrás y divide y vencerás son recursivos.

Ejemplo 4.1 (recursividad)

Dados dos números *a* (número entero) y *b* (número natural mayor o igual que cero) se desea determinar $\mathbf{a}^{\mathbf{b}}$, empleando una función recursiva.
Primero que todo elaboraremos la siguiente función en C++:

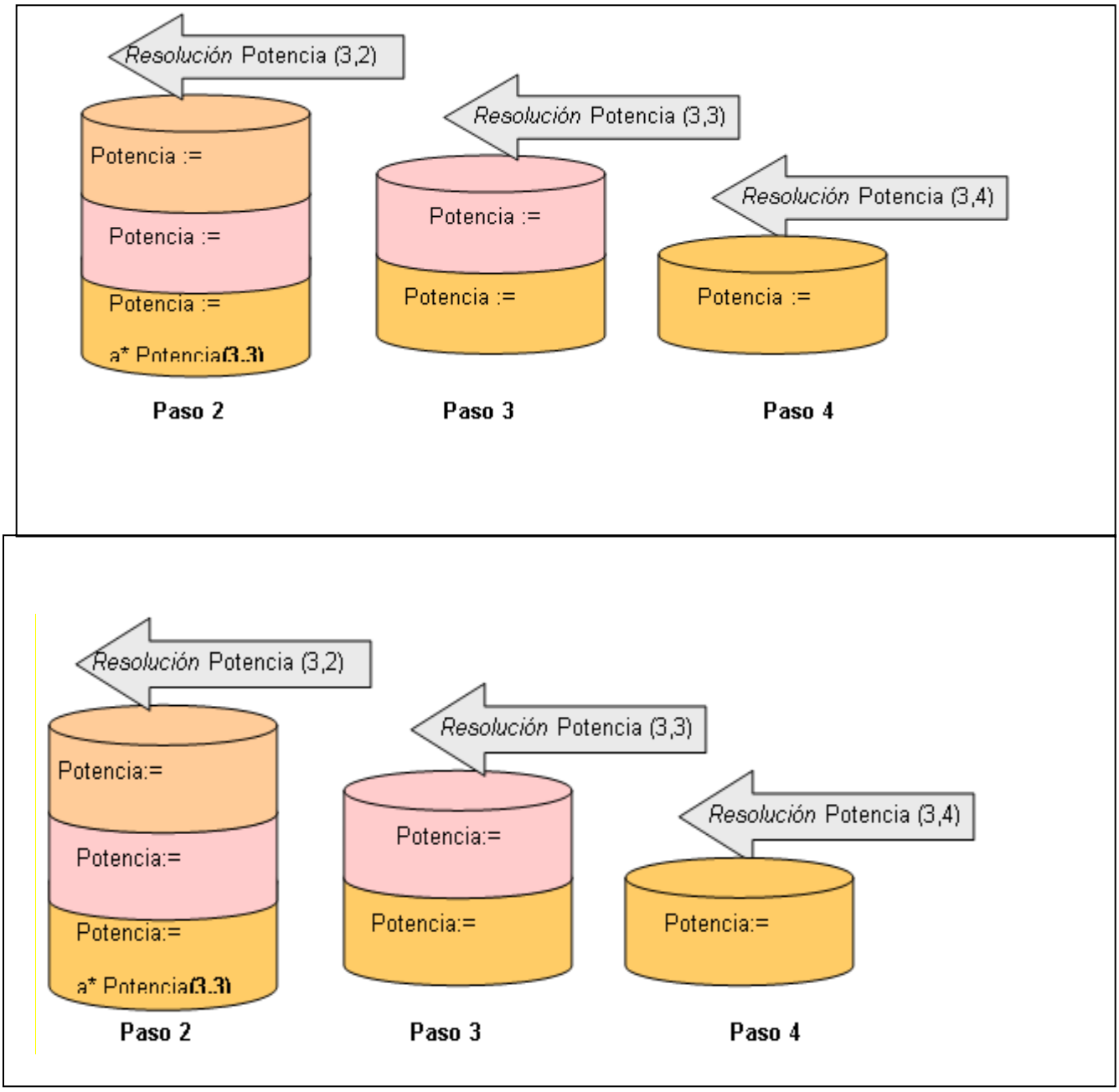
```
int potencia( int n, int e)
   {
     int r = 1;
     for( int i = 1; i <= e; i++){

        r *= n;
     }
     return r;
   }
```

Condición de parada

Llamada recursiva

En la Figura 4.1, aparece en forma gráfica la ejecución de esta función cuando se invoca. Observe el desencadenamiento de llamadas que provoca la ejecución de la función para **a** = 3 y **b** = 4



Secuencia de llamadas y resoluciones para el ejemplo dado
Figura 4.1

El valor obtenido es $a^b = 81$

Es de suma importancia determinar el tipo de parámetros (por referencia o por valor) a pasar en un procedimiento o función recursiva. ¿Qué efecto tienen ellos?

Ejercicios y actividades propuestas

- Investigue sobre las ventajas y las desventajas de implementar un procedimiento recursivo.
- Compare los procedimientos iterativos con los recursivos.
- Implemente en C++ y pruebe algoritmos recursivos.
- Elabore un procedimiento iterativo, para resolver el problema planteado en el ejemplo 3.1, y compare con el aquí presentado.
- En los cursos de Matemática se definen las sucesiones recurrentes o recursivas empleando una fórmula explícita como la siguiente: $c_1 = 4,\ c_n = 2c_{n-1},\ 2 \leq n \leq 8.$ Elabore una función recursiva que halle todos los términos de esta sucesión finita.

Ejercicios de Auto evaluación

4**.1** La siguiente expresión se denomina **fracción continua**:

$$q_0 + \cfrac{1}{q_1 + \cfrac{1}{q_2 + \cfrac{1}{q_3 + \ddots + \cfrac{1}{q_{n-1}}}}} \qquad (1)$$

en donde $q_1, q_2, \ldots, q_n$ son enteros positivos, a los cuales se les denominan cocientes incompletos y q_0 es un entero no negativo.

Las **fracciones continuas** son de gran aplicación en las matemáticas, especialmente en la aproximación de números irracionales por números racionales. El proceso de conversión de un número en una fracción continua se denomina desarrollo de este número en fracción continua. Para determinar los cocientes incompletos en este desarrollo de una fracción a / b, emplearemos el Algoritmo de Euclides como sigue:

$$a = bq_0 + r_1$$

$$
\begin{aligned}
b &= r_1 q_1 + r_2 \\
r_1 &= r_2 q_2 + r_3 \\
&\cdots\cdots\cdots\cdots \\
r_{n-2} &= r_{n-1} q_{n-1} + r_n \\
r_{n-1} &= r_n q_n
\end{aligned}
$$

De la primera igualdad se obtiene:

$$\frac{a}{b} = q_0 \quad + \quad \frac{r_1}{b} = q_0 + \frac{1}{\frac{b}{r_1}}$$

De la segunda igualdad se deduce:

$$\frac{b}{r_1} = q_1 \quad + \quad \frac{r_2}{r_1} = q_1 + \frac{1}{\frac{r_1}{r_2}}$$

Resultando:

$$\frac{a}{b} = q_0 \quad + \quad \frac{1}{q_1 + \frac{1}{\frac{r_1}{r_2}}}$$

El proceso continúa, si operamos con las siguientes igualdades hasta obtener la expresión (1).
Sobre la base de esta información elabore un programa en C++, que emplee una **función recursiva** denominada **FRACCION.** La función calculará el cociente entre un par de números enteros a, b empleando **fracciones continuas** como se expresa en (1) y formará parte de un programa de lectura de pares de números.

Ampliación de conocimientos

Existen lenguajes de programación que no soportan la **recursividad**, investigue cuáles la soportan y cuáles no.
La recursividad es muy útil en el campo de la Investigación de Operaciones, específicamente en el cálculo de las funciones de costo o de ganancia recursivas, cuando se resuelven problemas de Programación Dinámica. La Programación Dinámica consiste en resolver un problema de optimización, dividiéndolo en sub-problemas más sencillos de resolver, con miras a reducir su complejidad del problema original. Un ejemplo típico de la Programación Dinámica es la resolución del problema de la mochila. Existen también problemas que constan de múltiples períodos, para cuya resolución se aplica la técnica de la Programación Dinámica

Investigue en qué consiste el problema y su solución cuando se aplica la estrategia de **vuelta atrás**.

Respuestas a ejercicios de autoevaluación

4.1

```
int main()
{
//variables
int a,b,j,i,r,m;
//inicializar
a,b,j,m=0;
int a1,b1;
int q[20];
double y;
```

Procedimiento para calcular los cocientes q_i

```
    a1= a;
    b1 = b;
    r = 100;
    i = 0;
    while (r!=0) {
      r = a1 % b1;
      q[i] = (a1 - r)/b1;
      a1 = b1;
      b1 = r;
      i = i + 1;
     }
      printf( " %s %d %d \n"," cocientes : ", q[0], q[1]);
      m = i - 1;
      i = 0;
      y = Fraccion(q,i,m);
      printf(" %s %f ",  "Fraccion", y);
      system("pause");
```

Función recursiva Fracción

```
function Fraccion(q : tipo_cociente; var i: integer; N : integer): real;
var
      j : integer;
begin
  if (i < N)  then


  begin
     j := i + 1;
     Fraccion:= q[i] + 1/Fraccion(q,j,N)
```

Llamada recursiva

```
    end
  else Fraccion:= 1/q[N]
end;
```

La llamada a la **function** recursiva dentro del programa principal es la siguiente:

```
y = Fraccion(q,i,m);
```

Atención

La solución aquí presentada consiste en un procedimiento iterativo para el cálculo de los cocientes y una función recursiva, que utiliza los cocientes previamente calculados y envía el índice del arreglo de los cocientes como parámetro variable. ¿Es posible obtener el valor de la fracción continua empleando sólo recursión? Intente otro tipo de solución.

ÁRBOLES

En la vida real es frecuente observar representaciones en las que se emplean los árboles, entre ellas están los sistemas jerárquicos, los organigramas, los árboles genealógicos, etc. En computación los árboles son estructuras no lineales dinámicas de gran utilización en la representación y operación de datos determinados. El estudio de los árboles como estructuras de datos contempla básicamente los árboles binarios, los árboles binarios balanceados o AVL y árboles multicaminos, entre los cuales están los árboles B, árboles B^+ y finalmente los árboles generalizados.

Los **árboles binarios** son aquellos en los que cada nodo puede tener a lo sumo dos hijos. Los **árboles AVL** son árboles binarios en los que los procesos de inserción, búsqueda e información son más eficientes, en ellos se mantiene un balance o reacomodo de los nodos. Las estructuras mencionadas se almacenan y se operan en memoria, mientras que los **árboles multicaminos** son aquellos en los que se requiere emplear almacenamiento auxiliar para su operación. Los **árboles generalizados** son árboles cuyos nodos tienen múltiples hijos (tres o más); existen procedimientos para convertirlos en árboles binarios. En esta unidad sólo trataremos los árboles binarios.

A continuación, se presenta un **TAD ÁRBOL**, con las operaciones básicas que comúnmente se emplean para operar con árboles:

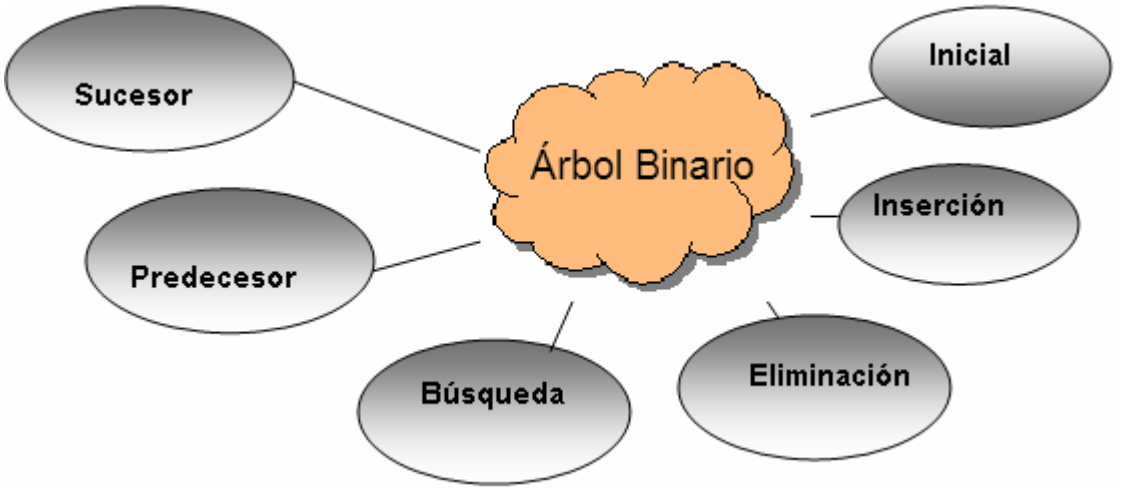


Fig. 3.2 Árbol Binario

Los procedimientos o funciones básicamente consisten en lo siguiente:

OPERACIÓN	DESCRIPCIÓN
Inicial:	Crea un árbol cuya raíz es nula
Inserción:	Agrega un nuevo nodo al árbol
Eliminación:	Elimina un nodo del árbol.
Búsqueda:	Busca información en un árbol, bajo un criterio
Predecesor:	Halla el nodo que precede a un nodo dado
Sucesor:	Halla el nodo que sigue a un nodo dado

Ejemplo 4.2 (operación con árboles)

Una de las operaciones que se requiere hacer eventualmente con un árbol binario, es determinar si el árbol contiene un valor y si el mismo está en un nodo interior al árbol o si está en un nodo terminal u hoja. Suponga que se tiene el siguiente árbol:

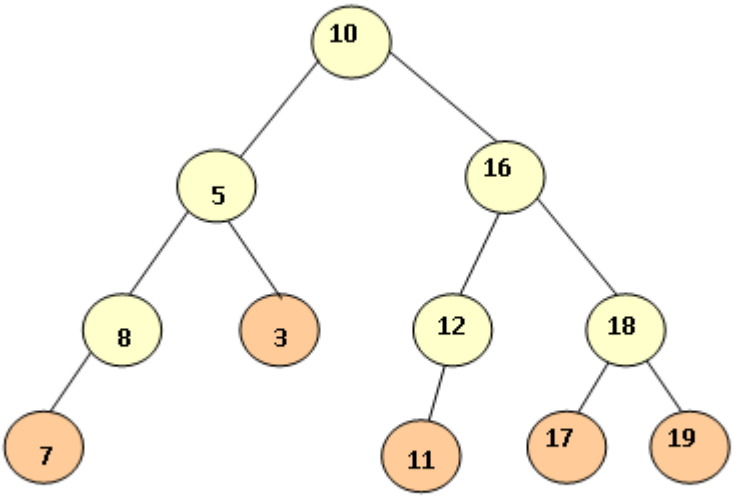


Figura 4.3

Se observa que:

El valor 3 está en el árbol y está en un nodo hoja
El valor 5 está en el árbol y está en un nodo interior.
El valor 20 no está en el árbol.

Sobre la base del ejemplo mostrado, elabore un procedimiento en C++ denominado **Búsqueda,** tal que dado un árbol binario, cuyos nodos, contienen lo siguiente:

- Información: un valor entero
- Apuntador Izquierdo a nodo
- Apuntador derecho a nodo,

y la información *x*, se efectuará un recorrido en dicho árbol y en caso que *x* esté en un nodo, imprimirá el mensaje de que está y además si se trata de un nodo interior o de un nodo hoja; en caso contrario proporcionará la información apropiada sobre la no existencia de *x* en el árbol.

Solución: Primero definiremos en C++ la clase Número:

Clase Número

```
class Numero {
public:
   string numero1;
   string numero2;
   string numero3;
   string numero4;
   string numero5;
   string numero6;
   string numero7;
   string numero8;
   string numero9;
   string numero10;
   string numero11;

   Numero() {}
   ~Numero() {}
   bool operator==(const Numero &numero) const
   {
      return this==&numero || this->numero1==numero.numero1;
   }

   bool operator<(const Numero &numero) const
   {
      return this->numero1<numero.numero1;
   }

   Numero& operator=(const Numero &numero)
   {
      if (this!=&numero)
      {
         this->numero1 = empleado.numero1;
         this->numero2 = empleado.numero2;
         this->numero3 = empleado.numero3;
         this->numero4 = empleado.numero4;
         this->numero5 = empleado.numero5;
         this->numero6 = empleado.numero6;
         this->numero7 = empleado.numero7;
         this->numero8 = empleado.numero8;
         this->numero9 = empleado.numero9;
         this->numero10 = empleado.numero10;
         this->numero11 = empleado.numero11;

      }
      return *this;
   }
```

El procedimiento **Búsqueda**, elaborado en C++, se presenta a continuación:

```
static T* buscar (Arbol<T>* arbol, const T &dato)
   {
      return arbol!=NULL ? (dato==arbol->dato ? &arbol->dato :
         (dato<arbol->dato ? buscar (arbol->izq, dato) :
         buscar (arbol->der, dato))) : NULL;
   }
```

Ejercicios y actividades propuestas

- Dado el árbol presentado en la Figura 3.2, determine para cada nodo, cuáles son los nodos hijos y hermanos. Igualmente determine cuáles son los niveles de dicho árbol y su altura.
- En el ejemplo anterior elabore un procedimiento de inserción. Implemente el algoritmo desarrollado en el computador y realice pruebas con múltiples datos.
- Halle otras operaciones que podrían asociarse al TAD árbol.

Ejercicios de Autoevaluación

4.1 Dado un árbol binario, no necesariamente completo, que contiene en cada nodo esencialmente información relativa a sumas de dinero, se desea elaborar un procedimiento, tal que dada la dirección del nodo raíz, obtenga la suma de las cantidades existentes en las hojas del árbol.
Suponga el siguiente árbol binario, en donde por razones de simplicidad sólo mostramos la data completa de las hojas de dicho árbol:

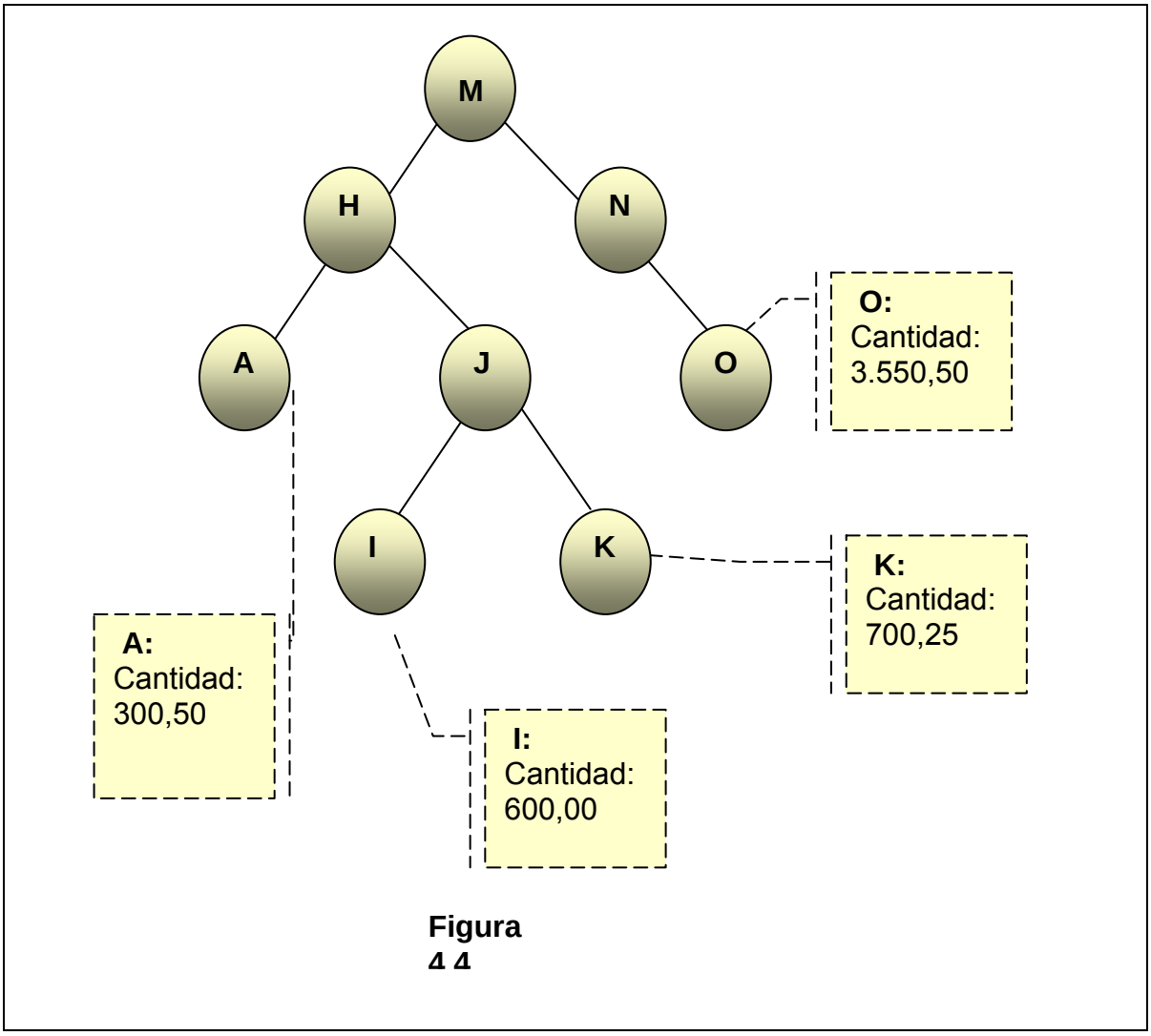


Figura 4.4

Al sumar las cantidades contenidas en las hojas, obtenemos el total de: **300,5 + 600 + 700,25 +3.550,5 = 5.151,25**

Sobre la base de esta información realice lo siguiente:
-Construya un procedimiento recursivo o una función recursiva en C++, tal que dada la dirección de la raíz de un árbol binario, no necesariamente completo, recorra el árbol y calcule la suma de las cantidades existentes en las hojas. El procedimiento o función imprimirá el resultado.

-Describa las estructuras de datos empleadas en C++, considerando que los nodos del árbol poseen la siguiente data:

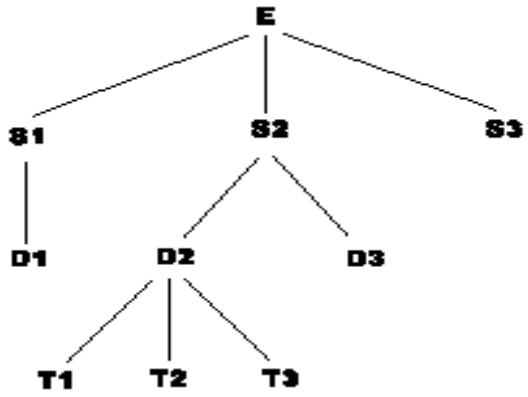
- Info: cadena de caracteres(longitud: 5)
- Cantidad
- Apuntador Izquierdo a nodo
- Apuntador derecho a nodo

Se presume que la información contenida en cada nodo del árbol ha sido previamente validada.

Ampliación de conocimientos

- Intente representar un árbol binario empleando estructuras de listas doblemente enlazadas. Desarrolle los algoritmos para operar en ellas.
- Investigue sobre el significado y utilidad de los *tries.* Diseñe un TAD para un *trie.*
- Investigue sobre la importancia del uso de los árboles en diferentes campos. En el área de Análisis de decisiones se utilizan para representar los estados y las decisiones de un problema. En el área de Sistemas Expertos son utilizados como estructuras de autoaprendizaje. Asimismo en algunos Sistemas Expertos se emplean técnicas de recorrido de árboles, como la de encadenamiento hacia atrás, para hallar soluciones de problemas.

4.2 Consideremos la siguiente figura:



Podemos observar que cada uno de los identificadores representa un nodo y la relación padre-hijo se señala con una línea. Los árboles normalmente se presentan en forma descendente y se interpretan de la siguiente forma:

- *E* es la raíz del árbol.
- *S1, S2, S3* son los hijos de *E*.
- *S1, D1* componen un subárbol de la raíz.
- *D1, T1, T2, T3, D3, S3* son las hojas del árbol.

Consideremos la siguiente terminología:

1. **Grado de salida** o simplemente **grado** .Se denomina grado de un nodo al número de hijos que tiene. Así el grado de un nodo hoja es cero. En la figura anterior el nodo con etiqueta *E* tiene grado 3.
2. **Caminos**. Si n_1, n2,...,n_k es una sucesión de nodos en un árbol tal que n_i es el padre de n_{i+1} para 1<=i<=k-1 ,entonces esta sucesión se llama un camino del nodo n_i al nodo n_k. La longitud de un camino es el número de nodos menos uno, que haya en el mismo. Existe un camino de longitud cero de cada nodo a sí mismo. Ejemplos sobre la figura anterior:
   i. *E, S2, D2, T3* es un camino de *E* a *T3* ya que *E* es padre de *S2*, éste es padre de *D2*, etc.
   ii. *S1, E, S2* no es un camino de *S1* a *S2* ya que *S1* no es padre de *E*.
3. **Ancestros y descendientes.** Si existe un camino, del nodo *a* al nodo *b*, entonces *a* es un ancestro de *b* y *b* es un descendiente de *a*. En el ejemplo anterior los ancestros de *D2* son *D2,S2* y *E* y sus descendientes *D2,T1,T2* y *T3*(cualquier nodo es a la vez ancestro y descendiente de sí mismo). Un ancestro o descendiente de un nodo, distinto de sí mismo se llama un ancestro propio o descendiente propio respectivamente. Podemos definir en términos de ancestros y descendientes los conceptos de raíz, hoja y subárbol:

   i. En un árbol, la raíz es el único nodo que no tiene ancestros propios.
   ii. Una hoja es un nodo sin descendientes propios.
   iii. Un subárbol de un árbol es un nodo, junto con todos sus descendientes.
4. Algunos autores prescinden de las definiciones de ancestro propio y descendiente propio asumiendo que un nodo no es ancestro ni descendiente de sí mismo.
5. **Altura.** La altura de un nodo en un árbol es la longitud del mayor de los caminos del nodo a cada hoja. La altura de un árbol es la altura de la raíz. Ejemplo: en la figura anterior la altura de *S2* es 2 y la del árbol es 3.
6. **Profundidad.** La profundidad de un nodo es la longitud del único camino de la raíz a ese nodo. Ejemplo: en la figura anterior la profundidad de *S2* es 1.
7. **Niveles.** Dado un árbol de altura *h* se definen los niveles *0...h* de manera que el nivel *i* está compuesto por todos los nodos de profundidad *i*.
8. **Orden de los nodos.** Los hijos de un nodo usualmente están ordenados de izquierda a derecha. Si deseamos explícitamente ignorar el orden de los dos hijos, nos referiremos a un árbol como un árbol no-ordenado.

   La ordenación izquierda-derecha de hermanos puede ser extendida para comparar cualesquiera dos nodos que no están relacionados por la relación ancestro-descendiente. La regla a usar es que si n_1 y n_2 son hermanos y n_1 está a la izquierda de n_2, entonces todos los descendientes de n_1 están a la izquierda de todos los descendientes de n_2.

A partir de los conceptos presentados se desea calcular:

1) Nivel del árbol
2) Numero de Nodos
3) Nivel Perfecto de Nodos
4) Factor de Equilibrio

Respuestas a ejercicios de autoevaluación

4.2 Función total de hojas

```
float totall(float a)
{
  float totall;
  totall = totall+a;
  //printf(" Total: %f ",totall);
  //printf("\t\n ");

  return totall;
```

Procedimiento: el siguiente modelo de procedimiento, recorre el árbol en preorden y verifica si el nodo visitado es una hoja, en cuyo caso guarda los totales.

```
void preorden(T_ARBOL *nodo)
{

 if (nodo != NULL) {
   visualizar(nodo);
   preorden(nodo->izq);
   preorden(nodo->der);
   if(nodo->izq==NULL)
      totall(nodo->cantidad);
   if(nodo->der==NULL)
      totalD(nodo->cantidad);
   }

}
```

Función Nivel de árbol:

```
int Nivel_arbol(NodoArbol *subArbol)
{
   int a=0,b=0;
 if(subArbol==NULL)
 {
        return 1;
 }

 else
 {
        a=Nivel_arbol(subArbol->HijoIzquierdo)+1;
        b=Nivel_arbol(subArbol->HijoDerecho)+1;
 }
 if(a>=b)
        return a++; //Le sumo la raiz
 else
        return b++; //Le sumo la raiz

}
```

Función Número de nodos: Donde FE es factor de equilibrio

```
int NumeroNodo (NodoArbol *subArbol)
{
      if(subArbol==NULL)
            return 0;

      else
      return          1+NumeroNodo(subArbol->HijoIzquierdo)+NumeroNodo(subArbol-
>HijoDerecho);
}

int FE(NodoArbol *subArbol)
{
      if(subArbol==NULL)
            return 0;

      else
            return          Nivel_arbol(subArbol->HijoDerecho)-Nivel_arbol(subArbol-
>HijoIzquierdo);
}
```

Función Factor de Equilibrio

```
int FE(NodoArbol *subArbol)
{
      if(subArbol==NULL)
        return 0;

      else
        return          Nivel_arbol(subArbol->HijoDerecho)-Nivel_arbol(subArbol-
>HijoIzquierdo);
}
int Total_hojas(NodoArbol *subArbol)
```

UNIDAD 5

La estructura de datos tipo grafo y algoritmos fundamentales.

Para el estudio de esta unidad, se sugiere, como en las anteriores, seguir las recomendaciones indicadas. Igualmente sugerimos continuar con las pruebas de los algoritmos en el computador y su adaptación a cada situación.

Objetivo de la Unidad 5

Implementar algoritmos en lenguaje de programación, a través del empleo de la estructura de grafo para la resolución de problemas específicos.

Contenido: Concepto de grafo. Representación del TAD grafo. Matriz de caminos. Algoritmos fundamentales con grafos. Aplicaciones.

Recomendaciones para el estudio del contenido de la unidad

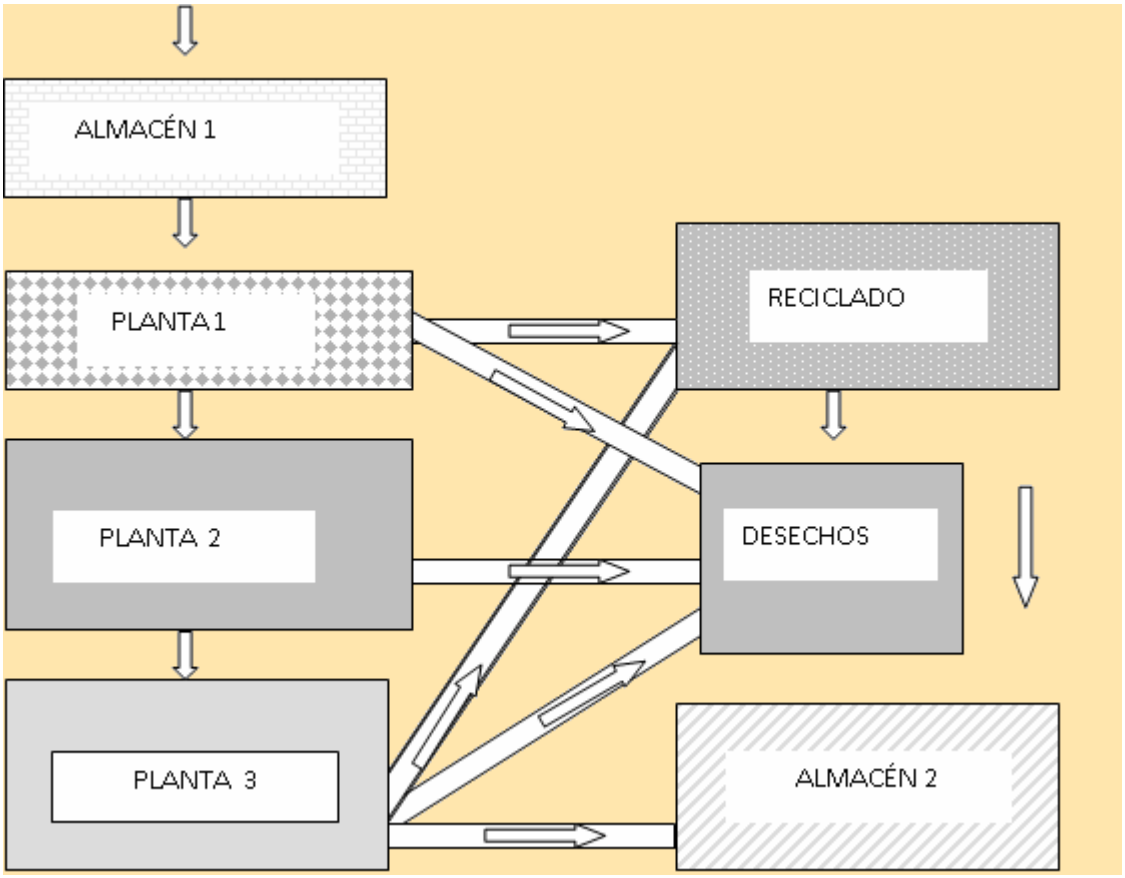
- **Estudiar** en el libro I el capítulo del libro "Grafos. Representación y operaciones", la estructura GRAFO. Si utiliza el texto II, estudie los GRAFOS dirigidos y no dirigidos, representación.
- **Profundizar** en su estudio de la representación Matriz de Adyacencia, Listas de Adyacencia, Matriz de Caminos y Cierre Transitivo.
- **Examinar** el recorrido de los grafos.
- **Realizar** los ejercicios propuestos en este capítulo.
- **Implementar** los algoritmos relacionados con la estructura GRAFO, en lenguaje de programación. Emplee el paquete de *Software* para probar estos algoritmos.
- **Investigar** sobre ejemplos de la vida real que puedan representarse como grafos.

GRAFOS

Es frecuente representar objetos que tienen múltiples relaciones con otros. Los grafos son abstracciones de estas relaciones y tienen gran aplicabilidad en diversos campos. Los modelos de rutas aéreas o terrestres, las redes de comunicaciones, los circuitos eléctricos, las relaciones entre individuos, son ejemplos, en donde los elementos se representan por puntos y sus relaciones por arcos. Los grafos o gráficas son estructuras cuyos nodos pueden poseer múltiples enlaces. La teoría de grafos se inicia con ideas geométricas muy simples pero de gran utilidad para resolver muchos problemas en diferentes campos.

Ejemplo 5.1 (representación en forma de grafo)

Sea la simplificación de un plano de una planta industrial (Figura 5. 1). A efectos de estudiar, bien sea el tráfico de los productos elaborados y semielaborados o del personal que labora, se podría representar este plano empleando un grafo, el cual simboliza una abstracción de la planta industrial, como se muestra en la Figura 5.2



DISEÑO DE UNA PLANTA INDUSTRIAL

Figura 5.1

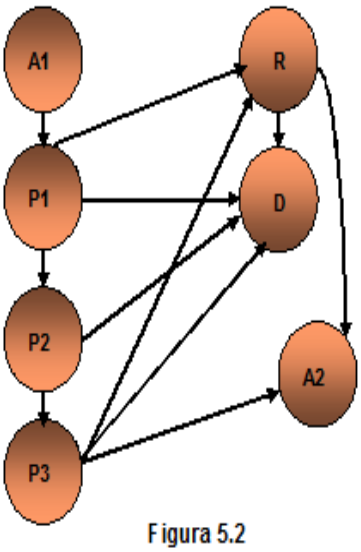


Figura 5.2

ABSTRACCION DISEÑO DE UNA PLANTA INDUSTRIAL

Seguidamente se representa un **TAD GRAFO**, con las siguientes operaciones:

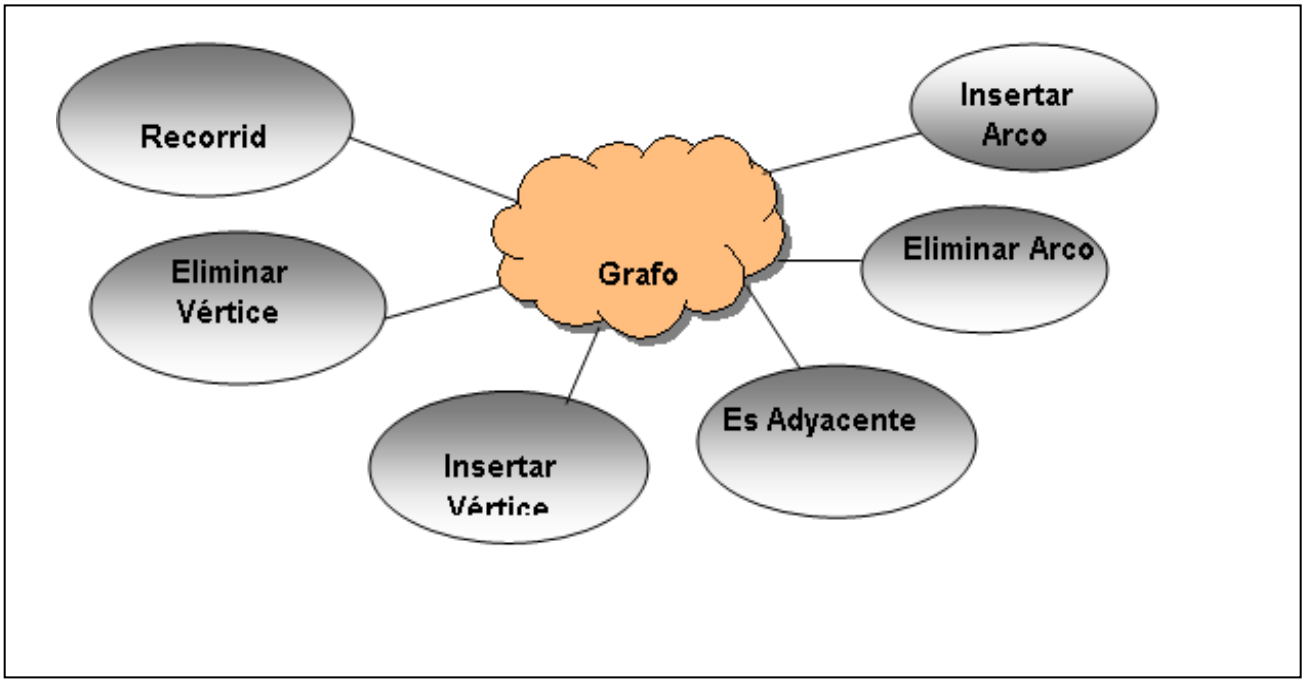


Figura 5.3 TAD Grafo

Los procedimientos o funciones básicamente consisten en lo siguiente:

OPERACIÓN	DESCRIPCIÓN
Insertar Arco	Agrega un vértice al grafo
Eliminar Arco	Elimina un arco del grafo
Es Adyacente	Determina si dos arcos son o no adyacentes
Insertar Vértice	Agrega un vértice
Eliminar Vértice	Elimina un vértice del grafo
Recorrido	Recorre el grafo.

Como se indica en el libro, el recorrido de un grafo podrá ser en **anchura** y **en profundidad**.

Ejemplo 5.2 (solución de un problema empleando grafos)

El departamento de una empresa consta de N empleados que denotaremos: E_1, E_2,...,E_N, algunos de los cuales pueden influir sobre los otros en la toma de decisiones relacionadas con sus funciones en la empresa. Es posible que ocurra que E_i influye sobre E_j y viceversa, por supuesto nunca ocurrirá que cualquier E_i influya sobre si mismo. También se ha analizado la denominada influencia de r etapas, la cual consiste en lo siguiente: E_i tiene una influencia de r etapas sobre E_j, si existe una trayectoria de longitud r de E_i hasta E_j. Estas relaciones también nos permiten determinar cuáles son los líderes del equipo de los N empleados.

Para resolver este problema primero debemos definir las estructuras de datos a emplear. En este caso podemos representar las relaciones entre los N empleados, utilizando una **matriz de adyacencia A**, cuyos elementos son unos y ceros, de la siguiente manera:

Si *Ei* ***R*** *Ej entonces* $a_{ij} = 1$
Si *Ei* ***R*** *Ej entonces* $a_{ij} = 0$

en donde R representa la relación "***influye sobre"***

Sobre la base de esta situación, elabore un programa en PASCAL, que en forma modular y estructurada permita resolver lo siguiente:
Dada una matriz de Adyacencia que representa la relación descrita, para un equipo de N empleados, determine lo siguiente:

- La influencia transitiva, la cual consiste en lo siguiente:
  Una influencia es transitiva si: dados tres empleados E_i, E_j y E_k, se cumple lo siguiente: si E_i influye sobre E_j y E_j influye sobre E_k, entonces E_i influye sobre E_k.

- La influencia en r etapas de E_i sobre E_j, dados r y un par de empleados E_i y E_j.

Solución:

Descripción de la matriz en C++:

```
float aleat=0.0;
 bool arreglo[5][5];
```

Los elementos de la matriz, igualmente pudieron definirse del tipo booleano, es decir con valores VERDADERO y FALSO.

```
//Crea una matriz NxN de ceros y unos
 void pedir(bool arreglo[5][5]=0)

for(int i=1; i<=N; i++)
     for(int j=1; j<=N; j++)
   {
       srand(time(NULL));
       aleat=1+rand()%(100-1);
       //cout << aleat << " ";
       //system ("pause");
       if (aleat <= 0.5){
          arreglo[j][i]=0;
       }
       if (aleat > 0.5){
          arreglo[i][j]=1;
       }
    }
```

Generación de un número aleatorio entre 0 y 100

Con este procedimiento se crea una matriz de Adyacencia, cuyos elementos fueron creados de manera aleatoria.
Una vez creada la matriz debe eliminarse los unos (o valores verdaderos) en la diagonal ya que esta relación no es reflexiva. Para ello diseñamos el siguiente procedimiento:

```
//Consiste en poner ceros en la diagonal de la matriz.
//Esto debe hacerse ya que la relación no debe ser reflexiva
for (int i =1; i<=N; i++){

       if (arreglo[i][i]= 1)
         arreglo[i][i]= 0;

      }
```

Matriz de Adyacencia

La matriz de Adyacencia la representación de los arcos que forman el grafo, suponiendo que los nodos o vértices pueden representarse con números ordinales. Para el caso de un grafo de N vértices, la matriz de Adyacencia tiene NxN elementos, a_{ij} ,

Una matriz de adyacencia, para cuatro empleados, creada con estos procedimientos podría ser como la que se muestra a continuación:

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

Procedimiento para determinar la influencia transitiva**.**

```
void DeterminarTransitividad(bool arreglo[5][5])

{
  ch='S';
    while (ch=='S'){ //Lectura de datos
       cout<<"Ingrese 3 números enteros menores que N y diferentes, cada uno en
una linea: ";
       cin >> i;
       cin >> j;
       cin >> k;
       getchar();

       if (arreglo[i][j]== 1 and arreglo[j][k]==1) {
        cout<<"La relacion es transitiva: ";
       }


       else if (arreglo[i][j]!= 1 and arreglo[j][k] != 1){
        cout<<"La relación no es transitiva: ";
       }
       cout<<"Continua(S/N)";
       cin >> ch;
       getchar();
    }
}
```

La matriz obtenida es la que representa la influencia en r etapas. Si a_{ij} = 1, entonces existe una influencia de r etapas del empleado i sobre el empleado j.

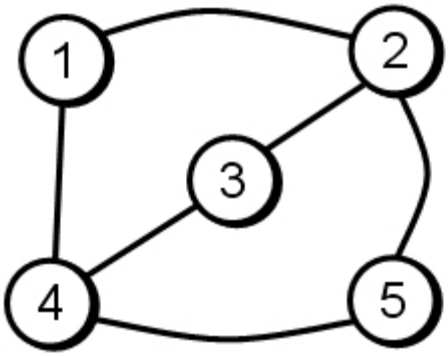
Ejemplo 5.3 (representación de grafos dirigidos)

Los grafos se pueden clasificar en dos grupos:
Dirigidos y no Dirigidos.

En un grafo dirigido cada arco está representado por un par ordenado de vértices, de forma que representan dos arcos diferentes.
En un grafo no dirigido el par de vértices que representa un arco no está ordenado.

Formas de Representación

Existen diferentes implementaciones del tipo grafo: con una matriz de adyacencias (forma acotada) y con listas y multilistas de adyacencia (no acotadas).



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

Fig. 5.4 Matriz de Adyacencia

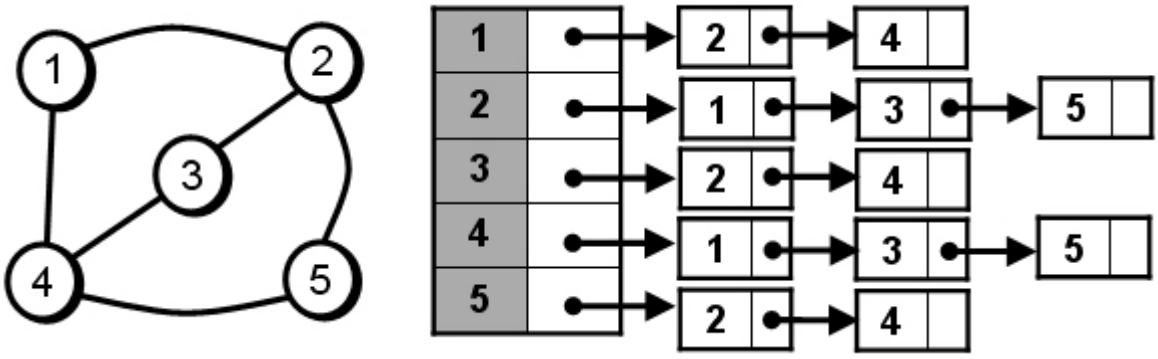


Fig. 5.5 Lista de Adyacencias

Aplicando la forma de representacion: de Lista de Adyacencia, realizar las siguientes operaciones de un grafo dirigido.

- ✓ Insertar un nodo(vertice)
- ✓ Insertar una arista
- ✓ Eliminar un nodo

- ✓ Eliminar una arista
- ✓ Mostrar el grafo
- ✓ Mostrar aristas de un nodo(vertice)

Solucion: Estructura de los nodos

```
struct nodo{
        char nombre;//nombre del vertice o nodo
        struct nodo *sgte;
        struct arista *ady;//puntero hacia la primera arista del nodo
        };
struct arista{
          struct nodo *destino;//puntero al nodo de llegada
          struct arista *sgte;
          };
typedef struct nodo *Tnodo;//  Tipo Nodo
typedef struct arista *Tarista; //Tipo Arista

Tnodo p;//puntero cabeza
```

Procedimientos a implementar

```
void menu();
void insertar_nodo();
void agrega_arista(Tnodo &, Tnodo &, Tarista &);
void insertar_arista();
void vaciar_aristas(Tnodo &);
void eliminar_nodo();
void eliminar_arista();
void mostrar_grafo();
void mostrar_aristas();
```

Funcion Principal

```
int main(void)
{   p=NULL;
    int op;     // opcion del menu

    system("color 0b");

    do
    {
       menu();
       cin>>op;

       switch(op)
       {
            case 1:
                  insertar_nodo();
                break;
          case 2: insertar_arista();
                break;
          case 3: eliminar_nodo();
```

```
                break;
          case 4: eliminar_arista();
                break;
          case 5: mostrar_grafo();
                break;
          case 6: mostrar_aristas();
                break;

          default: cout<<"OPCION NO VALIDA...!!!";
                 break;

       }
       cout<<endl<<endl;
       system("pause");  system("cls");

   }while(op!=7);
   getch();
   return 0;
}
```

Procedimiento Insertar Nodo

```
void insertar_nodo()
{
   Tnodo t,nuevo=new struct nodo;
   cout<<"INGRESE VARIABLE:";
   cin>>nuevo->nombre;
   nuevo->sgte = NULL;
   nuevo->ady=NULL;

   if(p==NULL)
    {
      p = nuevo;
      cout<<"PRIMER NODO...!!!";
    }
   else
    {
      t = p;
      while(t->sgte!=NULL)
       {
         t = t->sgte;
       }
      t->sgte = nuevo;
      cout<<"NODO INGRESADO...!!!";
    }

}
```

Procedimiento Agregar Arista

```
void agrega_arista(Tnodo &aux, Tnodo &aux2, Tarista &nuevo)
{
   Tarista q;
   if(aux->ady==NULL)
   {  aux->ady=nuevo;
      nuevo->destino=aux2;
      cout<<"PRIMERA ARISTA....!";
   }
   else
   {  q=aux->ady;
      while(q->sgte!=NULL)
         q=q->sgte;
      nuevo->destino=aux2;
      q->sgte=nuevo;
      cout<<"ARISTA AGREGADA...!!!!";
   }
}
```

Procedimiento Insertar Arista

```
void insertar_arista()
{  char ini, fin;
   Tarista nuevo=new struct arista;
   Tnodo aux, aux2;

   if(p==NULL)
    {
       cout<<"GRAFO VACIO...!!!!";
       return;
    }
   nuevo->sgte=NULL;
   cout<<"INGRESE NODO DE INICIO:";
   cin>>ini;
   cout<<"INGRESE NODO FINAL:";
   cin>>fin;
   aux=p;
   aux2=p;
   while(aux2!=NULL)
   {
      if(aux2->nombre==fin)
      {
         break;
      }
      aux2=aux2->sgte;
   }
   while(aux!=NULL)
   {
      if(aux->nombre==ini)
      {
         agrega_arista(aux,aux2, nuevo);
         return;
      }
     aux = aux->sgte;
   }
}
```

Vaciar Arista

```
void vaciar_aristas(Tnodo &aux)
{
   Tarista q,r;
   q=aux->ady;
   while(q->sgte!=NULL)
   {
      r=q;
      q=q->sgte;
      delete(r);
   }
}
```

Eliminar Nodo

```
void eliminar_nodo()
{   char var;
    Tnodo aux,ant;
    aux=p;
    cout<<"ELIMINAR UN NODO\n";
    if(p==NULL)
     {
        cout<<"GRAFO VACIO...!!!!";
        return;
     }
    cout<<"INGRESE NOMBRE DE VARIABLE:";
    cin>>var;
    while(aux!=NULL)
    {
       if(aux->nombre==var)
       {
          if(aux->ady!=NULL)
             vaciar_aristas(aux);

          if(aux==p)
          {

                p=p->sgte;
                delete(aux);
                cout<<"NODO ELIMINADO...!!!!";
                return;
          }
          else
          {
             ant->sgte = aux->sgte;
             delete(aux);
             cout<<"NODO ELIMINADO...!!!!";
             return;
          }
       }
       else
       {
          ant=aux;
          aux=aux->sgte;
        }
    }

}
```

Eliminar Arista

```
void eliminar_arista()
{
char ini, fin;
    Tnodo aux, aux2;
    Tarista q,r;
    cout<<"\n ELIMINAR ARISTA\n";
    cout<<"INGRESE NODO DE INICIO:";
    cin>>ini;
    cout<<"INGRESE NODO FINAL:";
    cin>>fin;
    aux=p;
    aux2=p;
    while(aux2!=NULL)
    {
       if(aux2->nombre==fin)
       {
          break;
       }
       else
       aux2=aux2->sgte;
    }
     while(aux!=NULL)
    {
       if(aux->nombre==ini)
       {
          q=aux->ady;
          while(q!=NULL)
          {
             if(q->destino==aux2)
             {
                if(q==aux->ady)
                   aux->ady=aux->ady->sgte;
                else
                   r->sgte=q->sgte;
                delete(q);
                cout<<"ARISTA  "<<aux->nombre<<"----->"<<aux2->nombre<<" ELIMINADA.....!!!!";
                return;
             }
          }
          r=q;
          q=q->sgte;
       }
       aux = aux->sgte;
    }
}
```

Mostrar Grafo

```
void mostrar_grafo()
{   Tnodo ptr;
    Tarista ar;
    ptr=p;
    cout<<"NODO|LISTA DE ADYACENCIA\n";
    while(ptr!=NULL)
    {   cout<<"   "<<ptr->nombre<<"|";
        if(ptr->ady!=NULL)
        {
            ar=ptr->ady;
            while(ar!=NULL)
            {   cout<<" "<<ar->destino->nombre;
                ar=ar->sgte;
             }
        }
        ptr=ptr->sgte;
        cout<<endl;
    }
}
```

Mostrar Aristas

```
void mostrar_aristas()
{
    Tnodo aux;
    Tarista ar;
    char var;
    cout<<"MOSTRAR ARISTAS DE NODO\n";
    cout<<"INGRESE NODO:";
    cin>>var;
    aux=p;
    while(aux!=NULL)
    {
        if(aux->nombre==var)
        {
            if(aux->ady==NULL)
            {   cout<<"EL NODO NO TIENE ARISTAS...!!!!";
                return;
             }
            else
            {
                cout<<"NODO|LISTA DE ADYACENCIA\n";
                cout<<"   "<<aux->nombre<<"|";
                ar=aux->ady;
                while(ar!=NULL)
                {
                    cout<<ar->destino->nombre<<" ";
                    ar=ar->sgte;
                }
                cout<<endl;
                return;
            }
        }
        else
        aux=aux->sgte;
    }
}
```

Ejercicios y actividades propuestas

- En el ejemplo determine cuál o cuáles son los líderes del equipo. Elabore un procedimiento o función que determine el líder del grupo.
- Determine la ventaja de emplear Listas de Adyacencias en lugar de matrices.

Ampliación de conocimientos

- En el campo de la Investigación de Operaciones existen muchos problemas cuya resolución se facilita empleando el enfoque de grafos. Son muy comunes el problema de la ruta más corta, el problema del flujo máximo y el problema del costo mínimo. Si desea ampliar más sus conocimientos al respecto, investigue sobre los algoritmos de Dijkstra, Floyd y Prim & Kruskal.

- Un concepto que tiene gran aplicación en este campo es el de **Ordenación Topológica**. Existen algoritmos para resolver problemas en donde se requiera aplicar este concepto, por ejemplo cuando se tiene una matriz curricular de una carrera y se necesita hallar los prerequisitos de un curso (o prelaciones). Investigue sobre los algoritmos apropiados para hallar soluciones empleando este concepto.

- Existen clases de problemas que no tienen solución a través de algoritmos eficientes. La teoría de la **NP - completitud** ha sido desarrollada para ofrecer respuestas y estudiar estas situaciones; a través de ella se ha podido demostrar que aquellos problemas difíciles, para los cuales no existen algoritmos eficientes para resolverlos, son equivalentes entre sí. Por lo tanto si se pudiera desarrollar un algoritmo para resolver un problema de esta clase, se estaría en capacidad de desarrollar un algoritmo eficiente para algún problema de la misma clase. Investigue sobre la teoría de la NP- Completitud.

- El problema de la **Coloración de Mapas** ha sido objeto de estudio por parte de los matemáticos por varias décadas. El mismo consiste en asignar colores a un mapa, disponiendo del menor número de colores, de manera tal que a dos países o regiones contiguas no se les asigne el mismo color. El medio de representación empleado para simbolizar el mapa es un grafo. En general el problema de colorear un grafo arbitrario con el menor número de colores posibles es un problema NP - completo. Investigue sobre las aplicaciones que tiene la coloración de mapas.

Módulo III

Métodos de ordenación y búsqueda

Implementar algoritmos en lenguaje de programación, empleando la estructura de grafo para la resolución de problemas específicos.

Objetivo del Módulo III

Implementar algoritmos de Ordenación y de búsqueda en lenguaje de programación, para la resolución de problemas específicos.

Estructura del modulo III

Unidad 6: Métodos de Ordenación

Unidad 7: Métodos de Búsqueda

UNIDAD 6

Métodos de Ordenación

La ordenación o clasificación de datos (sort, en inglés) es una operación consistente en disponer un conjunto —estructura— de datos en algún determinado orden con respecto a uno de los campos
de elementos del conjunto. Por ejemplo, cada elemento del conjunto de datos de una guía telefónica tiene un campo nombre, un campo dirección y un campo número de teléfono; la guía telefónica está dispuesta en orden alfabético de nombres; los elementos numéricos se pueden ordenar en orden creciente o decreciente de acuerdo al valor numérico del elemento. En terminología de ordenación, el elemento por el cual está ordenado un conjunto de datos (o se está buscando) se denomina clave
.
Una colección de datos (estructura) puede ser almacenada en un archivo, un array (vector o tabla), un array de registros, una lista enlazada o un árbol. Cuando los datos están almacenados en un array, una lista enlazada o un árbol, se denomina ordenación interna. Si los datos están almacenados en un archivo, el proceso de ordenación se llama ordenación externa.

Objetivo de la Unidad 6
Implementar algoritmos de Ordenación en lenguaje de programación, para la resolución de problemas específicos

Contenido: Ordenación. Algoritmo de Burbuja. Algoritmo de Inserción. Algoritmo *Shell*. Algoritmo *Quicksort*. Algoritmo *Heapsor*t. Otros.

Recomendaciones para el estudio de la Unidad

- **Estudiar** en el libro I el capítulo del libro "Algoritmo de Ordenación y búsqueda.
- **Profundizar** en el estudio de ordenación por intercambio, por selección, por inserción, por burbuja, Shell, ordenación rápida.
- **Examinar** Búsqueda en listas: secuencial y binario.
- **Realizar** los ejercicios propuestos en este capítulo.
- **Implementar** los algoritmos de ordenación básicos.
- **Investigar** sobre ejemplos en le procesamiento de datos donde se apliquen los métodos de ordenación.

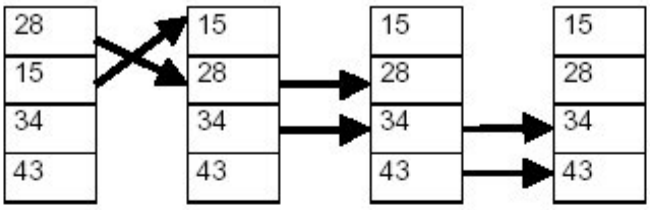
ORDENAMIENTO.

Uno de los procedimientos más comunes y útiles en el procesamiento de datos, es la clasificación u ordenación de los mismos. Se considera ordenar al proceso de reorganizar un conjunto dado de objetos en una secuencia determinada. Cuando se analiza un método de ordenación, hay que determinar cuántas comparaciones e intercambios se realizan para el caso más favorable, para el caso medio y para el caso más desfavorable.

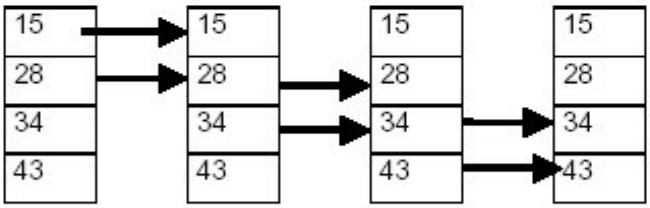
.

Ejemplo 6.1 (algoritmo de Burbuja)

Para ilustrar el método: Supóngase que están almacenados cuatro números en un arreglo con casillas de memoria de x[1] a x[4]. Se desea disponer esos números en orden creciente. La primera pasada de la ordenación por burbujeo haría lo siguiente:
Comparar el contenido de x[1] con el de x[2]; si x[1] contiene el mayor de los números, se intercambian sus contenidos.
Comparar el contenido de x[2] con el de x[3]; e intercambiarlos si fuera necesario.
Comparar el contenido de x[3] con el de x[4]; e intercambiarlos si fuera necesario.
Al final de la primera pasada, el mayor de los números estará en x[4].



Nótese que el arreglo ya quedó ordenado, sin embargo, se hace una pasada más porque la computadora no advierte que el arreglo está en orden hasta que ocurre una pasada sin intercambios.



La ordenación por burbujeo se llama así porque los números más pequeños ascienden como burbujas hasta la parte superior, mientras que los mayores se hunden y caen hasta el fondo. Está garantizado que cada pasada pone el siguiente número más grande en su lugar, aunque pueden colocarse más de ellos en su lugar por casualidad.

Se tiene como ejemplo un programa que lee un arreglo de 10 números y los ordena con el método de la burbuja de manera ascendente o descendente según se elija. Para esto utiliza una función que recibe el arreglo y la variable 'ord' (1=asc, 2=desc), luego utiliza otra función para imprimirlo.

Proceso de ordenación

```
int ordenar(int lista[],int ord)
{
  int c1,c2,aux;
  for(c1=0;c1<=9;c1++)
  {
    for(c2=0;c2<9;c2++)
    {
      if(ord==1)
      {
        if(lista[c2]>lista[c2+1])
        {
          aux=lista[c2];
          lista[c2]=lista[c2+1];
          lista[c2+1]=aux;
        }
      }
      else
      {
        if(lista[c2]<lista[c2+1])
        {
          aux=lista[c2];
          lista[c2]=lista[c2+1];
          lista[c2+1]=aux;
        }
      }
    }
  }
  return 0;
}
```

Ejemplo 6.2 (Inserción)

El fundamento de este método consiste en insertar los elementos no ordenados del arreglo en subarreglos del mismo que ya estén ordenados. Dependiendo del método elegido para encontrar la posición de inserción tendremos distintas versiones del método de inserción.

Supongamos que tenemos un mazo de cartas. ¿Cómo se ordenan cuando se reciben? Se toma la primera y se coloca en la mano. Luego se toma la segunda y se compara con la que se tiene: si es mayor, se pone en la mano derecha, y si es menor a la izquierda. Después se toma la tercera y se compara con las que se tienen en la mano, desplazándola hasta que quede en su posición final. Se continúa haciendo

esto, *insertando* cada carta en la posición que le corresponde, hasta que se tienen todas en orden. El algoritmo de inserción tiene el mismo concepto.
Para simular esto en un programa necesitamos tener en cuenta algo: no podemos desplazar los elementos así como así o se perderá un elemento. Lo que hacemos es guardar una copia del elemento actual (que sería como la carta que tomamos) y desplazar todos los elementos mayores hacia la derecha. Luego copiamos el elemento guardado en la posición del último elemento que se desplazó.

Proceso de ordenación

```
for (i=1; i<TAM; i++)
temp = lista[i];
j = i - 1;
while ( (lista[j] > temp) && (j >= 0) )
  lista[j+1] = lista[j];
  j--;
  lista[j+1] = temp;
```

Ejemplo 6.3 (selección)

Los métodos de ordenación por selección se basan en dos principios básicos:
Seleccionar el elemento más pequeño (o más grande) del arreglo.
Colocarlo en la posición más baja (o más alta) del arreglo.
A diferencia del método de la burbuja, en este método el elemento más pequeño (o más grande) es el que se coloca en la posición final que le corresponde.
Ejemplo: dado n cantidad de números ordenarlo en forma ascendente:

Proceso de ordenación

```
void seleccionsort (int  A[], int n)
{
      int min,i,j,aux;
      for (i=0; i<n-1; i++)
      {
            min=i;
            for(j=i+1; j<n; j++)
                  if(A[min] > A[j])
                     min=j;
            aux=A[min];
            A[min]=A[i];
            A[i]=aux ;
      }

}
```

Biblioteca Leearreglo.h de Ingreso y Salida de números

```
#include<iostream>
using namespace std;
void leeCadena(int cant,int n[])
{
   int i;
   for(i=0;i<cant;i++)
   {
      cout<<"Ingresa numero "<<i+1<<": ";
      cin>>n[i];
   }

}

void muestraCadena(int cant,int n[])
{
   int i;
   for(i=0;i<cant;i++)
   {
      cout<<n[i]<<endl;
   }
   printf("\n ");
   system("pause");
   system("cls");
}
```

Ejercicios y actividades propuestas

- Realizar un programa donde se introduzcan una lista con las notas de los N estudiantes de la asignatura de Programación Estructurada e imprima las notas en formato ascendente (menor a mayor). Con el método de la burbuja.

- Realizar un programa que inicialice una lista con valores e imprima los datos ordenados de menor a mayor utilizando el método de quicksort. Nota: Versión Array.
- Realizar la Implementación de lo métodos de ordenamiento burbuja, burbuja mejorada, insercion, seleccion, shell y mezcla.

Ampliación de conocimientos

Quicksort.

- Si bien el método de la burbuja era considerado como el peor método de ordenación simple o menos eficiente, el método Quicksort basa su estrategia en la idea intuitiva de que es más fácil ordenar una gran estructura de datos subdividiéndolas en otras más pequeñas introduciendo un orden relativo entre ellas. En otras palabras, si dividimos el array a ordenar en dos subarrays de forma que los elementos del subarray inferior sean más pequeños que los del subarray superior, y aplicamos el método reiteradamente, al final tendremos el array inicial totalmente ordenado. Existen además otros métodos conocidos, el de ordenación por montículo y el de shell.
  El algoritmo es éste: Recorres la lista simultáneamente con i y j: por la Izquierda con i (desde el primer elemento) y por la derecha con j (desde el Último elemento). Cuando lista[i] sea mayor que el elemento de división y lista[j] sea menor los intercambias.
  Se repite esto hasta que se crucen los índices.
  El punto en que se cruzan los índices es la posición adecuada para colocar El elemento de división, porque sabemos que a un lado los elementos son todos menores y al otro son todos mayores (o habrían sido intercambiados). Al finalizar este procedimiento el elemento de división queda en una posición En que todos los elementos a su izquierda son menores que él y los que están a su derecha son mayores.
  Realizar el código del algoritmo en C++.

***Heapsor*t.**

- La ordenación por montículos toma su nombre de las estructuras conocidas como montículos, que son una clase especial de árboles. Este tipo de árboles poseen la característica de que son binarios completos, lo que significa que cada vértice tiene a los más dos hijos, y que todos sus niveles están completos, excepto (posiblemente) el último, el cuál se llena de izquierda a derecha. Otra particularidad que deben cumplir es que todo vértice debe ser igual o mayor a cualquiera de sus hijos. En consecuencia, el elemento mayor siempre los podemos encontrar en la raíz. La ordenación por montículos utiliza la propiedad de la raíz para ordenar el arreglo. Una vez que el arreglo cumpla las propiedades del montículo, quitamos la raíz y la colocamos al final del arreglo. Con los datos que sobran, creamos otro montículo y repetimos hasta que todos los datos estén ordenados.

Mediana de la Serie.

Se da una serie de *n* enteros no negativos. Definamos a la mediana de dicha serie. Si *n* es un número impar, la mediana es el elemento que se encuentra a la mitad de la serie una vez que ésta es ordenada. Puedes darte cuenta que en

este caso la mediana es elemento de posición en la serie ordenada, si empezamos a contar desde 1. Si *n* es par, la mediana se obtiene mediante la semi-suma de los dos elementos que se encuentran en medio de la serie ordenada. Esto es, es la semi-suma de los elementos con posiciones $\frac{n}{2}$ y $\frac{n}{2}+1$ de la serie ordenada. Tenga en cuenta que la serie no necesariamente está ordenado. Realizar la mediana de la serie en C++.

Shell sort

- El ordenamiento Shell (Shell sort en inglés) es un algoritmo de ordenamiento. El método se denomina Shell en honor de su inventor Donald Shell que lo publicó en 1959.

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

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

  El algoritmo Shell sort mejora el ordenamiento por inserción comparando elementos separados por un espacio de varias posiciones.

  Esto permite que un elemento haga "pasos más grandes" hacia su posición esperada.Los pasos múltiples sobre los datos se hacen con tamaños de espacio cada vez más pequeños. El último paso del Shell sort es un simple ordenamiento por inserción, pero para entonces, ya está garantizado que los datos del vector están casi ordenados.

  Actividad: Realizar el algoritmo Shell sort en C++.

Radix Sort

- Llamado también ordenamiento Radix, es un algoritmo de ordenamiento estable que puede ser usado para ordenar ítems identificados por llaves (o claves) únicas.Cada llave debe ser una cadena o un número capaz de ser ordenada Alfanuméricamente.

  Reglas para ordenar:

  Empezar en el dígito más significativo y avanzar por los dígitos menos significativos mientras coinciden los dígitos correspondientes en los dos números. El número con el dígito más grande en la primera posición en la cual los dígitos de los dos números no coinciden es el mayor de los dos (por supuesto sí coinciden todos los dígitos de ambos números, son iguales). Este mismo principio se toma para Radix Sort.

  Actividad: Realizar el algoritmo Radix sort en C++.

UNIDAD 7

Métodos de Búsqueda

Los algoritmos de búsqueda permiten recuperar uno o varios elementos particulares de un gran volumen de información previamente almacenada (e.j.) buscar un elemento contenido en una lista). La información típicamente está almacenada en registros que contienen una clave y la data. El objetivo de la operación de búsqueda es encontrar todos los registros cuya claves coincidan con una cierta clave especificada con el propósito de acceder a la información (no sólo a la clave) para su procesamiento.
Algunos términos comunes para describir las estructuras de datos relacionadas con las operaciones de búsqueda son: i) diccionarios y ii) tablas de símbolos. En esta unidad estudiaremos métodos de búsqueda elementales y avanzados. Como ocurre regularmente, es preferible considerar que los algoritmos de búsqueda pertenecen a conjunto de rutinas empaquetadas que realizan una serie de rutinas genéricas, que se pueden disociar de las implementaciones particulares, de forma tal de facilitar el pase de una implementación a otra. En consecuencia, se define un tipo de dato abstracto cuyas operaciones principales son:

- buscar
- inicializar
- insertar un nuevo registro
- eliminar un registro
- ordenar (dar como salida todos los registros ordenados)
- unir 2 diccionarios en uno solo (de mayor tamaño)

Objetivo de la Unidad 7
Implementar algoritmos de Búsqueda en lenguaje de programación, para la resolución de problemas específicos.

Contenido: Búsqueda. Búsqueda Lineal. Búsqueda Binaria. *Hashing*. Función de *Hashing*

Recomendaciones para el estudio de la Unidad

- **Estudiar** en el libro I el capítulo del libro "Búsquedas en Listas Búsqueda secuencial y binaria.
- **Profundizar** Búsqueda secuencial.
- **Examinar** Búsqueda Binaria.
- **Realizar** los ejercicios propuestos en este capítulo.
- **Implementar** los algoritmos de búsqueda básicos.
- **Investigar** sobre ejemplos en le procesamiento de datos donde se apliquen los métodos de ordenación.

Ejemplo 7.1 (Búsqueda Secuencial).

También conocida como búsqueda lineal. Supongamos una colección de registros organizados como una lista lineal. El algoritmo básico de búsqueda secuencial consiste en empezar al inicio de la lista es ir a través de cada registro hasta encontrar la llave indicada (k), o hasta al final de la lista (Fig. 7.1)

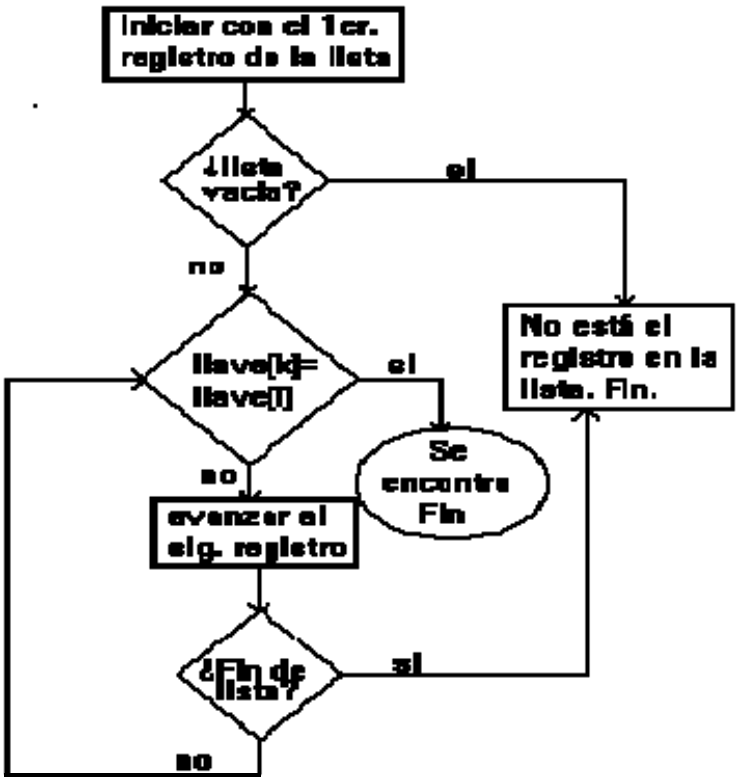


Fig. 7.1 Búsqueda Secuencial

La situación óptima es que el registro buscado sea el primero en ser examinado. El peor es cuando las llaves de todos los n registros son comparados con k (lo que se busca). El promedio en n/2 comparaciones.
Este método de búsqueda es muy lento, pero si los datos no están en orden es el único método que puede emplearse para realizar las búsquedas. Si los valores de la llave no son únicos para encontrar todos los registros con una llave particular, se requiere buscar en toda la lista.

Programa de Búsqueda Lineal

```
//Busqueda lineal //en un arreglo.
#include <iostream>
using std::cout;
using std::cin;
using std::endl;
void mostrarArreglo(const int[], int); //prototipo de funcion que recibe un arreglo constante
int busquedaLineal(const int[], int, int); //arreglo, tamano, clave
int main()
{
  int clave =0;
  const int tamano = 15;
  int arreglo[tamano] = {25,17,13,16,41,32,12,115,95,84,54,63,78,21,10};
  cout << "Elementos del arreglo: " << endl;
  mostrarArreglo(arreglo,tamano);
  cout << "Indique un valor a buscar y se le devolvera el indice: " << endl;
  cin >> clave;
  cout<< "Su valor se encuentra en arreglo["<<busquedaLineal(arreglo,tamano,clave)<<"]"<< endl;
  cout << "Fin del programa :)" << endl;
  return 0;
}//fin de main
void mostrarArreglo(const int arreglo[], int tamano)
{
  for (int i = 0 ; i < tamano ; i++)
    cout << "arreglo[" << i <<"]=" << arreglo[i] << endl;
}
int busquedaLineal(const int arreglo[], int tamano, int clave)
{
  for (int i = 0; i< tamano ; i++)
    if (arreglo[i] == clave)
      return i;
  return -1;
}
```

Ejemplo 7.2 (Búsqueda Binaria).

La búsqueda binaria sólo se puede implementar si el arreglo está ordenado. La idea consiste en ir dividiendo el arreglo en mitades. Por ejemplo supongamos que tenemos este vector:
int vector[10] = {2,4,6,8,10,12,14,16,18,20};

La clave que queremos buscar es 6. El algoritmo funciona de la siguiente manera:

1. Se determinan un índice arriba y un índice abajo, Iarriba=0 e Iabajo=9 respectivamente.
2. Se determina un índice central, Icentro = (Iarriba + Iabajo)/2, en este caso quedaría Icentro = 4.
3. Evaluamos si vector [Icentro] es igual a la clave de búsqueda, si es igual ya encontramos la clave y devolvemos Icentro.
4. Si son distintos, evaluamos si vector [Icentro] es mayor o menos que la clave, como el arreglo está ordenado al hacer esto ya podemos descartar una mitad del arreglo asegurándonos que en esa mitad no está la clave que buscamos. En

nuestro caso vector [Icentro] = 4 < 6, entonces la parte del arreglo vector [0...4] ya puede descartarse.

5. Reasignamos larriba o labajo para obtener la nueva parte del arreglo en donde queremos buscar. larriba, queda igual ya que sigue siendo el tope. labajo lo tenemos subir hasta 5, entonces quedaría larriba = 9, labajo = 5. Y volvemos al paso 2.

Programa de Búsqueda Binaria

```
//Busqueda binaria
//en un arreglo.
#include <iostream>
using std::cout;
using std::cin;
using std::endl;
void mostrarArreglo(const int[], int); //prototipo de funcion que
recibe un arreglo constante
int busquedaBinaria(const int[], int, int); //arreglo, tamano, clave
void ordenarArreglo(int[], int); //prototipo que modifica y ordena el
arreglo
void intercambiar(int&, int&); //prototipo, intercambia
los valores de dos elementos
int main()
{
  int clave =0;
  const int tamano = 15;
  int arreglo[tamano] = {25,17,13,16,41,32,12,115,95,84,54,63,78,21,10};
  //ordenamos el arreglo para que funcione la busquedaBinaria
  ordenarArreglo(arreglo,tamano);
  cout << "Elementos del arreglo: " << endl;
  mostrarArreglo(arreglo,tamano);
  cout << "Indique un valor a buscar y se le devolvera el indice: " << endl;
  cin >> clave;
  cout<< "Su valor se encuentra en arreglo["<<busquedaBinaria(arreglo,tamano,clave)<<"]" << endl;
  cout << "Fin del programa :)" << endl;
  return 0;
}//fin de main
void mostrarArreglo(const int arreglo[], int tamano)
{
  for (int i = 0 ; i < tamano ; i++)
    cout << "arreglo[" << i << "]=" << arreglo[i] << endl;
}
int busquedaBinaria(const int arreglo[], int tamano, int clave)
{
  int larriba = tamano-1;
  int labajo = 0;
  int lcentro;
  while (labajo <= larriba)
    {
      lcentro = (larriba + labajo)/2;
      if (arreglo[lcentro] == clave)
 return lcentro;
      else
 if (clave < arreglo[lcentro])
   larriba=lcentro-1;
 else
   labajo=lcentro+1;
    }
```

```
  return -1;
}
void ordenarArreglo(int arreglo[], int tamano)
{
  for (int i = 0; i< tamano -1 ; i++)
    for (int j = 0; j< tamano -1 ; j++)
      if (arreglo[j] > arreglo[j+1])
 intercambiar(arreglo[j],arreglo[j+1]);
}
void intercambiar(int &a, int &b)
{
  int tmp = b;
  b = a;
  a = tmp;
}
```