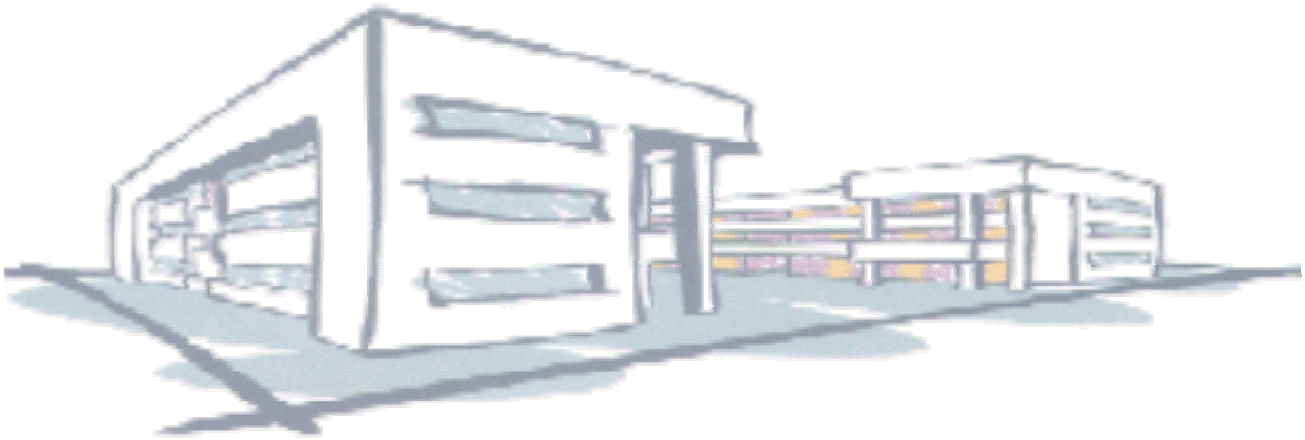
Instituto Politécnico Nacional
Escuela Superior de Cómputo
Departamento de Ciencias e Ingeniería de la Computación
Academia de Algoritmia y Programación

Apuntes de

ESTRUCTURAS DE DATOS



Araujo Díaz David.

México, D.F. Abril del año 2010.

Contenido

Página

Unidad 1: Tipos Abstractos de Datos

1.1 Abstracción en Lenguajes de Programación

La abstracción consiste en aislar un elemento de su contexto o del resto de los elementos que lo acompañan **[1**, **5**, **7**, **8**, **9**, **10** y **18]**. En programación, el término se refiere al énfasis en el *¿qué hace?* más que en el *¿cómo lo hace?*). El común denominador en la evolución de los lenguajes de programación, desde los clásicos o imperativos hasta los orientados a objetos, ha sido el nivel de abstracción del que cada uno de ellos hace uso.

Los lenguajes de programación son las herramientas mediante las cuales los diseñadores de lenguajes pueden implementar los **modelos abstractos**. La abstracción ofrecida por los lenguajes de programación se puede dividir en dos categorías: **abstracción de datos** (pertenecientes a los datos) y **abstracción de control** (perteneciente a las estructuras de control).

Los diferentes paradigmas de programación han aumentado su nivel de abstracción, comenzando desde los lenguajes de máquina, los más próximos al microprocesador y más lejano a la comprensión humana; pasando por los lenguajes de comandos, los imperativos, la orientación a objetos, la Programación Orientada a Aspectos; u otros paradigmas como la programación declarativa, lógica, funcional, etc.

Un lenguaje para computadora esta formado por un código, las reglas de sintaxis y las instrucciones. Los lenguajes se clasifican de acuerdo con la cercanía que exista entre el usuario y la máquina; los niveles son (**Figura 1.1**):

- **Lenguajes de alto nivel**.- son fáciles de programar, son transportables e independientes del tipo de máquina. Son lentos además de ser extensos. Por ejemplo tenemos FORTRAN, BASIC, PASCAL, etc.
- **Lenguajes de medio nivel**.- Combina las características de los lenguajes de alto y bajo nivel, como velocidad, transportabilidad, independencia, compactos y facilidad de programación. Por ejemplo se tienen Turbo C, FORTH, etc.
- **Lenguajes de bajo nivel**.- son rápidos, compactos, son muy completos y dependen del microprocesador. Son no transportables y difíciles de programar. Ejemplo: ENSAMBLADOR.

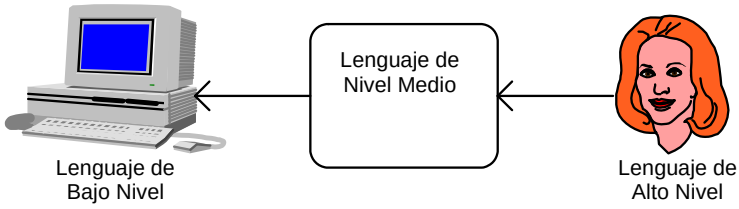


Figura 1.1: Cercanía de los lenguajes con el usuario.

1.2 Tipos Abstractos de Datos

En el mundo de la programación existen diversos lenguajes que se han ido creando con el paso del tiempo y que se han perfeccionado debido a las necesidades de los programadores de la época a la que pertenecen. Los primeros lenguajes de programación eran de tipo lineal, ya que un programa se recorría desde un punto marcado como *inicio* hasta llegar a un punto *fin*. Con el tiempo se fueron creando nuevos lenguajes y en nuestros días los más utilizados son los llamados Lenguajes Orientados a Objetos.

Los Lenguajes Orientados a Objetos tienen la característica de que no son lenguajes lineales, sino que se forman de diversas funciones, las cuales son llamadas en el orden en que el programa mismo las pide o el usuario determina. La evolución hacia los Lenguajes Orientados a Objetos, se basa en un concepto fundamental en las Estructuras de Datos denominado Abstracción de Datos y que es parte importante de estos Lenguajes y de la manera en que funciona la mayoría del software comercial de nuestros días.

El concepto de **Tipo Abstracto de Datos** (**TAD**), fue propuesto por primera vez hacia **1974** por **John Guttag** y otros investigadores, pero no fue hasta **1975** que por primera vez **Liskov** lo propuso para un lenguaje de programación.

El lenguaje **Turbo Pascal** fue determinante para la común aceptación de los **TAD** con la introducción de las **`Units`**. Si bien estas no cumplen con las características básicas de un Tipo Abstracto de Datos, como por ejemplo: la encapsulación de los datos. El lenguaje **Ada** pudo implementar exitosamente los **TAD** con sus **`Packages`**.

La **abstracción de datos** consiste en ocultar las características de un objeto y obviarlas, de manera que solamente utilizamos el nombre del objeto en nuestro programa. El concepto de abstracción es muy útil en la programación, ya que un usuario no necesita mencionar todas las características y funciones de un objeto cada vez que éste se utiliza, sino que son declarados por separado en el programa y simplemente se utiliza el término abstracto.

Supongamos que en algún Lenguaje de Programación un pequeño programa saca el área de un rectángulo de las dimensiones que un usuario decida. Pensemos también que el usuario probablemente quiera saber el área de varios rectángulos. Sería muy tedioso para el programador definir la multiplicación de *base* por *altura* varias veces en el programa, además que limitaría al usuario a sacar un número determinado de áreas. Por ello, el programador puede crear una función denominada **`Área`**, la cual va a ser llamada el número de veces que sean necesitadas por el usuario y así el programador se evita mucho trabajo, el programa resulta más rápido, más eficiente y de menor longitud.

Al hecho de guardar todas las características y habilidades de un objeto por separado se le llama **Encapsulamiento** y es también un concepto importante para entender la estructuración de datos.

Cuando se usa en un programa de computación, un **TAD** es representado por su interfaz, la cual sirve como cubierta a la correspondiente implementación. Los usuarios de un **TAD** tienen que preocuparse por la interfaz, pero no con la implementación, ya que esta puede cambiar en el tiempo y afectar a los programas que usan el **TAD**. Esto se basa en el concepto de **Ocultación** de información, una protección para el programa de decisiones de diseño que son objeto de cambio.

La solidez de un **TAD** reposa en la idea de que la implementación está escondida al usuario. Solo la interfaz es pública. Esto significa que el **TAD** puede ser implementado de diferentes formas, pero mientras se mantenga consistente con la interfaz, los programas que lo usan no se ven afectados.

Por ejemplo, un **TAD** de una lista puede ser implementado mediante un **Arreglo** o una **Lista Enlazada** o hasta un **Árbol**. Una **lista** es un **TAD** con operaciones bien definidas (agregar elemento, agregar al final, agregar al principio, recuperar, eliminar, etc.) mientras una **lista enlazada** es una estructura de datos basada en punteros o referencias (dependiendo del lenguaje) que puede ser usada para crear una representación de una **Lista**. La **Lista Enlazada** es comúnmente usada para representar una **TAD Lista**.

De forma similar, un **TAD Árbol** puede ser representado de muchas maneras: Árbol binario, Árbol AVL, Árbol rojo-negro, Arreglo, etc. A pesar de la implementación un Árbol binario siempre tiene las mismas operaciones (insertar, eliminar, encontrar, etc.)

Los **TAD** que tienen informaciones simples pero dependientes de un comportamiento estructural serán llamados **TAD polilíticos** y aquellos **TAD** simples, como son los tipos predefinidos donde la

información no es relacionada mediante ninguna estructura y no admiten más que un valor en cada momento serán denominados **TAD monolíticos**.

Cuando se hable de un **TAD** no haremos ninguna alusión al tipo de los elementos sino tan sólo a la forma en que están dispuestos estos elementos. Sólo nos interesa la estructura que soporta la información y sus operaciones. Para determinar el comportamiento estructural basta con observar la conducta que seguirán los datos.

Un **TAD** tendrá una parte que será invisible al usuario la cual hay que proteger y que se puede decir que es irrelevante para el uso del usuario y está constituida tanto por la maquinaria algorítmica que implemente la semántica de las operaciones como por los datos que sirvan de enlace entre los elementos del **TAD**, es decir, información interna necesaria para la implementación que se esté haciendo para ese comportamiento del **TAD**.

Un **TAD** representa una abstracción, en donde:

- Se destacan los detalles (normalmente pocos) de la especificación (el qué).
- Se ocultan los detalles (casi siempre numerosos) de la implementación (el cómo).

La abstracción, es una de las herramientas que más nos ayuda a la hora de solucionar un problema, es un mecanismo fundamental para la comprensión de problemas y fenómenos que poseen una gran cantidad de detalles, su idea principal consiste en manejar un problema, fenómeno, objeto, tema o idea como un concepto general, sin considerar la gran cantidad de detalles que estos puedan tener. El proceso de abstracción presenta dos aspectos complementarios.

1. Destacar los aspectos relevantes del objeto.
2. Ignorar los aspectos irrelevantes del mismo (la irrelevancia depende del nivel de abstracción, ya que si se pasa a niveles más concretos, es posible que ciertos aspectos pasen a ser relevantes).

De modo general podemos decir que la abstracción permite establecer un nivel jerárquico en el estudio de los fenómenos, el cual se establece por niveles sucesivos de detalles. Generalmente, se sigue un sentido descendente de detalles, desde los niveles más generales a los niveles más concretos.

Los diferentes tipos de abstracción que podemos encontrar en un programa son:

1. **Abstracción funcional:** crear procedimientos y funciones e invocarlos mediante un nombre donde se destaca qué hace la función y se ignora cómo lo hace. El usuario sólo necesita conocer la especificación de la abstracción (el qué) y puede ignorar el resto de los detalles (el cómo). Aparece al pensar de manera abstracta las operaciones que necesitamos para resolver un problema. Este tipo de abstracción nos permite definir operaciones nuevas en una aplicación que anteriormente carecía de ellas. La abstracción funcional fue la primera en aparecer ya que es fácil de llevar a la práctica debido a que su implementación es posible en la gran mayoría de los lenguajes de programación. Suele corresponderse con el uso de procedimientos o funciones.

2. **Abstracción de datos:** surge cuando se abstrae el significado de los diferentes tipos de datos que aparecen en nuestro problema. Este tipo de abstracción nos permite crear nuevos tipos de datos pensando en los posibles valores que pueden tomar y en las operaciones que los manipulan. Como cabe esperar, estas operaciones serán a su vez abstracciones funcionales. Se presentan las siguientes definiciones.

- **Tipo de datos:** proporcionado por los leguajes de alto nivel. La representación usada es invisible al programador, al cual solo se le permite ver las operaciones predefinidas para cada tipo.

- **Tipos definidos:** por el programador que posibilitan la definición de valores de datos más cercanos al problema que se pretende resolver.
- **TAD:** para la definición y representación de tipos de datos (valores + operaciones), junto con sus propiedades.
- **Objetos:** Son **TAD** a los que se añade propiedades de reutilización y de compartición de código.

Algunos ejemplos de utilización de **TAD** en programación son:

- **Conjuntos:** Implementación de conjuntos con sus operaciones básicas (unión, intersección y diferencia), operaciones de inserción, borrado, búsqueda, etc.
- **Árboles Binarios de Búsqueda:** Implementación de árboles de elementos, utilizados para la representación interna de datos complejos. Aunque siempre se los toma como un **TAD** separado son parte de la familia de los grafos.
- **Pilas y Colas:** Implementación de los algoritmos **FIFO** y **LIFO**.
- **Grafos:** Implementación de grafos; una serie de vértices unidos mediante una serie de arcos o aristas.

Las principales ventajas que nos aportan los **TAD** son las siguientes:

1. Mejoran la conceptualización y hacen más claro y comprensible el código.
2. Hacen que el sistema sea más robusto.
3. Reducen el tiempo de compilación.
4. Permiten modificar la implementación sin que afecte al interfaz público.
5. Facilitan la extensibilidad.

1.3 Especificación de los TAD

La construcción de un **TAD** consta de dos fases bien diferenciadas entre ellas: la especificación (formal e informal) y la implementación.

Las características de un **TAD** no deben depender de su realización concreta, sino solamente de cómo queremos que sea su comportamiento, lo cual llamamos **especificación**. Para la especificación de un tipo abstracto de datos en lenguaje natural (**especificación informal**) se emplea el siguiente esquema:

- **TIPO DE DATOS:** Nombre del tipo (Lista de operaciones).
- **VALORES:** Descripción de los posibles valores.
- **OPERACIONES:** Descripción de cada operación.

Primero indicaremos el nombre del **TAD** y citaremos todas las operaciones definidas. En el apartado **valores** describiremos los posibles valores de los datos de este tipo, pero lo haremos desde un punto de vista abstracto, sin pensar en la posible realización concreta. Finalmente en el apartado **operaciones** haremos una descripción de cada una de las operaciones definidas sobre el **TAD**. En la **especificación informal**, habitualmente hacemos referencia a conceptos (como por ejemplo conceptos matemáticos). El problema surge cuando estos datos auxiliares no están definidos tan precisamente.

La **especificación formal** nos permite definir conceptos de manera mucho más precisa. Para ello utilizaremos este esquema:

- **TIPO:** Nombre del tipo de datos.
- **SINTAXIS:** Forma de las operaciones.
- **SEMÁNTICA:** Significado de las operaciones.

En el apartado **sintaxis** proporcionamos el tipo de datos de entrada y de salida de cada una de las funciones definidas sobre el **TAD**, mientras que en **semántica** describiremos sus comportamientos. Sin embargo, ésta vez lo haremos siguiendo unas normas algebraicas básicas.

En la **implementación** del **TAD** lo que hacemos es elegir la forma en que se van a representar los distintos valores que tomarán los datos. También seleccionaremos la manera en que se realizarán las operaciones. Para esta elección debemos tener en cuenta que todas las operaciones se realicen de la forma más eficiente posible, aunque con la práctica veremos que la mayoría de las veces una implementación facilita mucho algunas operaciones mientras que complica otras.

Los tipos abstractos de datos básicos se clasifican habitualmente, atendiendo a su **estructura**, en **lineales** y **no lineales**.

a. TAD lineales

- **Listas.** Esta forma de almacenar elementos consiste en colocarlos en una lista lineal que tiene un enlace por cada elemento para determinar cual es el elemento siguiente. Las listas se utilizan habitualmente para guardar elementos del mismo tipo y se caracterizan porque pueden contener un número indeterminado de elementos y porque siguen un orden explícito. La lista de cero elementos se denomina lista vacía.
- **Colas.** En el contexto de la programación, una cola es una lista en la que los elementos se insertan por un extremo (llamado fondo) y se suprimen por el otro (llamado frente). En esta estructura de datos el primer elemento que entra es el primero en salir. Es un tipo de dato muy común tanto dentro de la informática como en la vida real.
- **Pilas.** Una pila es una estructura de datos en la cual todas las inserciones y las eliminaciones se realizan por el mismo extremo, denominado cima de la pila. En una pila, el último elemento en entrar es el primero en salir, al contrario de lo que pasa en las colas.

b. TAD no lineales

- **Árboles.** El árbol es un **TAD** que organiza sus elementos (nodos) de forma jerárquica. Si una rama va del nodo **a** al nodo **b**, entonces **a** es el padre de **b**. Todos los nodos tienen un padre, excepto el nodo principal, denominado **raíz del árbol**.
- **Árboles binarios de búsqueda.** El árbol binario de búsqueda es una variación del **TAD árbol**. Se trata de aquellos árboles binarios (cada nodo tiene dos hijos como máximo) en los cuales todos los datos del subárbol izquierdo son menores que los datos del nodo, y los datos del subárbol derecho son mayores que los datos del nodo.
- **Grafos.** Hemos visto que los árboles binarios representan datos entre los cuales existe una jerarquía. Los grafos sin embargo se utilizan para representar relaciones arbitrarias entre datos. En su representación, cada elemento forma un nodo. La relación entre nodos forma una arista que puede ser dirigida o bidireccional (no dirigida).

1.4 Tipos abstractos de datos en Lenguaje C

Un tipo abstracto de datos llamado es un conjunto de valores y operaciones que se definen por especificación, siendo independientes de su implementación.

Ejemplo: definir un número fraccionario.

1. Se crea un archivo con extensión **`*.h`**, en el cual pondremos la definición del tipo y las declaraciones de las variables y funciones a utilizar, llamémoslo **`fraccion1.h`** y contendrá:

```
#include <stdio.h>
#ifndef _fraccion_ /*sentencia del preprocesador de c, sirve para
comprobar que no hay otro tipo fracción en cuyo caso no podrá ser
definido nuestro TAD*/
#define _fraccion_

typedef struct estructura{
      int numerador;
      int denominador; /*numero fraccionario tipo-> 1/2,2/4,5/6.....*/
    }tipofraccion;

typedef tipofraccion *fraccion;
fraccion crear_fraccion(int x,int y);     /*crea la fracción          */
void destruir_fraccion(fraccion f);       /*libera la memoria         */
int numerador(fraccion f);                /*devuelve denominador de f */
int denominador(fraccion f);              /*devuelve denominador de f */
void producto(fraccion f,fraccion g);     /*suma f=f*g                */
#endif
```

2. Una vez que se tiene definido el **TAD**, deberemos de implementar las funciones en un archivo de extensión **`*.c`**, llamémoslo **`archivo.c`**:

```
#include “fraccion1.h”
#include “stdio.h”
#include “stdlib.h”

fraccion crear_fraccion(int x,int y){
      fraccion f;
      f = (fraccion)malloc(sizeof(tipofraccion));
      if(f == NULL){
            perror("error: no hay memoria\n");
            exit(1);
            }
      f -> numerador = x;
      f -> denominador = y;
      return f;
      }

void destruir_fraccion(fraccion f){
      free(f);
      }

int numerador(fraccion f){
      return f -> numerador;
      }

int denominador(fraccion f){
      return f -> denominador;
      }

void producto(fraccion f, fraccion g){
      f -> numerador *= g -> numerador;
      f -> denominador *= g -> denominador;
      }
```

3. Utilización dentro de un programa:

```
#include “stdio.h”
#include “stdlib.h”
#include “archivo.h”

int main (void){
   fraccion f,g;
   f = crear_fraccion(2,4);
   g = crear_fraccion(3,5);
   producto(f,g);
   printf(  "Multiplicar f con g es: %d,%d\n",
            numerador(f),denominador(f));
   destruir_fraccion(f);
   destruir_fraccion(g);
   return 0;
}
```

Unidad 2: Estructuras de Datos Lineales, Estáticas y Dinámicas

2.1 Pilas

Una **pila** es una colección ordenada de elementos en la que pueden insertarse y suprimirse, en un extremo llamado **tope** nuevos elementos (**Figura 2.1**) **[1, 5, 7, 8, 9, 10** y **18]**. También se define como una **estructura de datos lineal**, en la cual las operaciones se realizan por uno de los extremos de una lista.

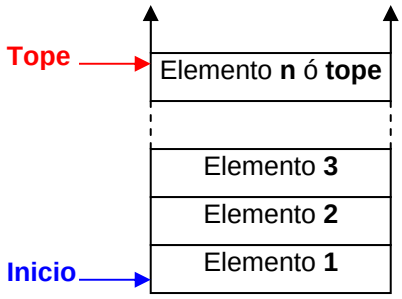


Figura 2.1: Ejemplo de una pila con sus elementos.

La definición de **pila** a diferencia de los **arreglos**, integra los elementos de **inserción** y **supresión**, por lo que se puede considerar como un objeto dinámico, es decir, constantemente variable. Cuando se agregan o se suprimen elementos de la pila, se agregan o suprimen a través de un elemento llamado **tope**, el cual es un extremo de la pila, previamente definido.

Un ejemplo, es una **pila de platos**; en ella, al añadir o quitar platos, se hace siempre por la parte superior de la pila. Este tipo de pilas reciben también el nombre de listas **LIFO** (**L**ast **I**n - **F**irst **O**ut; **Ú**ltimo en **E**ntrar - **P**rimero en **S**alir -**UEPS**).

Otros ejemplos son: una pila de tortillas, un estacionamiento en forma de calle cerrada, las filas de cines y teatros que se encuentran entre la pared y el pasillo.

En computación, las pilas son ampliamente empleadas en las operaciones de salto a subrutina, a nivel de lenguaje ensamblador, en los microcontroladores y microprocesadores, aunque en muchos casos las operaciones que se realizan con la pila, son transparentes al usuario. Las **pilas** son la segunda estructura de datos después de los **arreglos**, por que se mapean directamente en memoria.

2.1.1 Operaciones con pilas

Las operaciones básicas en una pila son las de **insertar** y **suprimir**, las cuales son conocidas en el lenguaje ensamblador como **push** y **pop** respectivamente. Cuando se recupera un elemento de la pila, también se elimina de la misma. Estas operaciones toman un tiempo **Θ(1).**

- **Push** (`Insertar`).- inserta un elemento en la pila.
- **Pop** (`Suprimir`).- elimina el elemento que fue insertado más recientemente en la pila (**tope**).

En ocasiones a una pila se le conoce como **lista de apilamiento**, debido a la operación **push**, que inserta elementos a la pila.

No existe un límite superior en el numero de elementos de la pila (se vería limitado solo por la cantidad de memoria disponible en la computadora), pero si la pila no contiene elementos, se le llama **pila vacía**.

La representación de una pila como un tipo abstracto de datos (**TDA**), es directa y se puede denotar de la forma en que se muestra en la **Figura 2.2**, en esa figura, también es posible observar la implementación de forma dinámica de las operaciones **push** y **pop**, que se emplearán más adelante para la manipulación de pilas en la resolución de problemas.

TDA de una pila

Elementos:	Depende de la aplicación.
Estructura:	Lineal.
Dominio:	La pila tendrá capacidad de almacenar cualquier cantidad de elementos, según su representación lo permita.
Operaciones:	• **Insertar** (**Push**).- Agrega un elemento a la pila. **Entradas**: la pila a la que se va agregar el elemento y el elemento que se generará. **Salidas**: La pila con un elemento más. **Precondición**: la pila esta creada y no está llena. **Poscondición**: La pila queda con un elemento adicional, agregado por el extremo del tope. • **Suprimir** (**Pop**).- Elimina el último elemento que se agrego a la pila. **Entradas**: La pila a la que se va a quitar un elemento. **Salidas**: La pila con un elemento menos y el elemento que se elimino. **Precondición**: La pila esta creada y no esta vacía. **Poscondición**: La pila queda con un elemento menos, eliminado por el extremo del tope de la pila. • **Pila Vacía**.- Verifica si una determinada pila esta vacía o no. **Entradas**: la pila que se va a verificar. **Salidas**: valor booleno que indique si la pila está vacía o no. **Precondición**: La pila por verificar existe. **Poscondición**: La estructura no se modifica • **Llena**: Verifica si la pila determinada esta llena o no. **Entradas**: la pila que se va a verificar. **Salidas**: Valor booleano que indica si la pila está llena o no. **Precondición**: La pila por verificar existe. **Poscondición**: la estructura de datos no se modifica.

Figura 2.2: Definición de pila como un **tipo abstracto de datos** (**TAD**).

2.1.2 Implementación estática

La implementación de una pila como una colección ordenada de elementos en lenguaje **C** se puede realizar mediante un **arreglo** (un arreglo es una colección de objetos del mismo tipo). Está representación, es solamente válida si la solución del problema requiere del uso de una sola pila. Un arreglo y una pila son dos cosas diferentes, en primer lugar el arreglo tiene un número predefinido de elementos, mientras que una pila varía constantemente de tamaño, cuando se agregan o suprimen elementos (**Tabla 2.1**).

Así un arreglo, a pesar de que no es una pila, puede usarse como tal. Es decir, puede declararse un arreglo lo suficientemente grande para admitir el tamaño máximo de la pila.

El fondo fijo de la pila es un extremo del arreglo, mientras que su tope cambia en forma constante. Por lo que debemos de asegurar saber en donde se encuentra el tope de la pila.

Diferencias

Arreglo	Pila
• Conjunto predefinido de elementos. • Estructura estática. • Se puede asignar un valor a cada elemento refiriéndose al índice.	• Conjunto no definido de elementos. • Estructura dinámica. • Presenta operaciones de inserción y supresión.

Coincidencias

Arreglo	Pila
• Los elementos son del mismo tipo. • Tiene un elemento inicial.	• Los elementos comparten la misma estructura. • Tiene un inicio.

Tabla 2.1: Comparación entre **arreglos** y **pilas**.

Ejemplo 2.1: Realizar un programa para **evaluar expresiones**, con los operadores **+**, **-**, ***** y **/**, y en la cual se utilice la notación **postfija** (Implementar con **pilas estáticas**).

En la notación **postfija** ó notación **polaca inversa**, los operadores matemáticos van a continuación de sus operandos. Por ejemplo, la expresión **ab + c ***, equivale a la expresión convencional (**infija**): **(a + b) * c**.

En la notación **postfija**, no es necesario el uso de paréntesis, a pesar de que en la notación convencional son necesarios para asegurar el orden correcto de la evaluación. Su uso no es necesario, debido a que se encuentran perfectamente definida la precedencia de operadores.

La expresión **postfija** es **recorrida de izquierda a derecha**, así, cada vez que se encuentra un operando, este es apilado en la pila. Cuando se encuentra un operador, se aplica a los dos operandos más recientes apilados en la pila; el resultado se almacena en el tope de la pila.

La evaluación de expresiones en forma postfija, es un buen ejemplo del uso de pilas, pues como se observa en la **Figura 2.3** es posible emplearla de forma simple para hallar el resultado de la operación **ab + c ***.

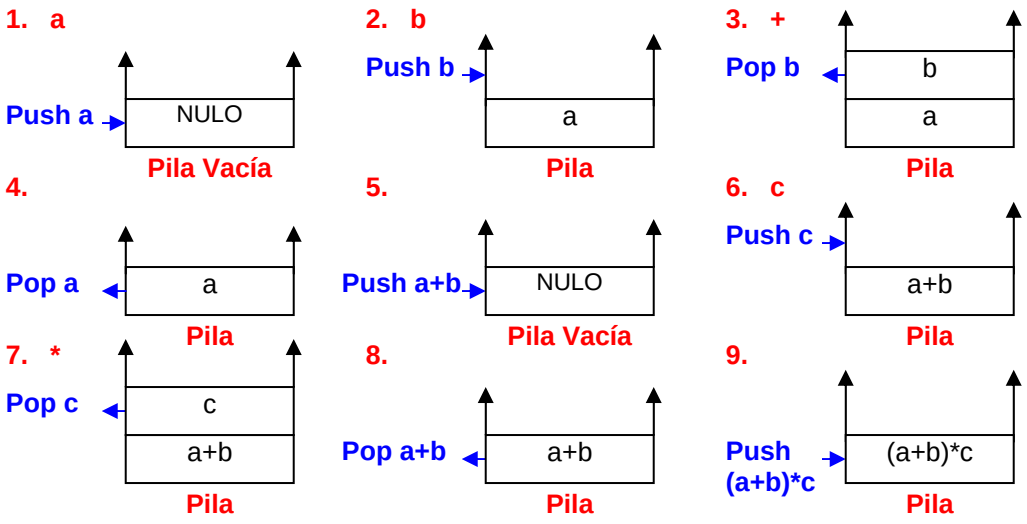


Figura 2.3: Uso de una pila para la evaluación de una expresión postfija (**ab + c ***).

En el **Programa 2.1**, se observa la implementación de una pila estática, mediante la declaración de un arreglo `pila`, y una variable global llamada `tope`, la cual es usada en todo momento para determinar la posición del tope de la pila.

```
/* Librerías                                                    */
#include "stdio.h"
#include "stdlib.h"
#include "conio.h"
#define     TAM         100         /* Tamaño de la pila       */

/* Arreglo de datos estático                                    */
float pila[TAM];                    /* Dato de tipo real       */
int tope;

/* Añade un elemento a la pila                                  */
void push(float x){
tope++;
pila[tope] = x;
}

/* Recupera un elemento de la pila                              */
float pop(){
float x;
if(tope < 0){
      tope = -1;
      printf("\n\a\tError: POP = Pila Vacía...\n");
      return(0);
      }
x = pila[tope];
tope--;
return(x);
}

/* Carga la expresión en un arreglo                             */
void lee(char ent[]){
int pos = 0;
printf("\n\n\tEntra la Expresión Postfija : ");
while((ent[pos++] = getchar()) != '\n');
ent[--pos] = '\0';
}

/* Evalúa la expresión                                          */
float evalua(char ent[]){
float a,b;
char op[1];
int pos = 0;
tope = -1;
while(ent[pos] != '\0'){
      *op = ent[pos++];
      switch(*op){
            case '+':
                  b = pop();
                  a = pop();
                  push(a + b);
                  break;
            case '-':
                  b = pop();
                  a = pop();
                  push(a - b);
                  break;
```

Programa 2.1: Definición de **pila estática** para **evaluar expresiones postfijas** (Parte **1**/2).

```
            case '*':
                  b = pop();
                  a = pop();
                  push(a * b);
            break;
            case '/':
                  b = pop();
                  a = pop();
                  if(b == 0.){
                        printf("\n\t\aDivisión por CERO\n");
                        return(0.);
                        break;
                        }
                  push(a / b);
                  break;
            default:
                  push(atof(op)); /* Almacena dato en la pila          */
            }
      }
return(pop());
}

/* Programa principal                                                  */
void main(void){
char ent[100];
clrscr();
printf("Calculadora Básica.");
printf("Emplea Notación postfija: 2 + 5  -> 25+");
printf("Operaciones: Suma +, Resta -, Producto * y División /");
lee(ent);
printf("\n\n\t%s = %g",ent,evalua(ent));
}
```

Programa 2.1: Definición de **pila estática** para **evaluar expresiones postfijas** (Parte **2**/2).

Al ejecutar el programa anterior produce el resultado que se muestra en la **Figura 2.4**.

```
                     Calculadora Básica.
Emplea Notación postfija: 2 + 5  -> 25+
Operaciones: Suma +, Resta -, Producto * y División /
Entra la Expresión Postfija: 4837+*2/+
4837+*2/+ = 44
```

Figura 2.4: Resultados del programa para evaluar expresiones postfijas con pila estática.

La expresión evaluada en su forma **Infija** es: **((7 + 3) * 8) / 2 + 4 = 44**

2.1.3 Implementación Dinámica

Una **estructura dinámica** es aquella que tiene la capacidad de variar su tamaño. Por lo tanto no es posible asignar una cantidad fija de memoria, se emplea entonces la **asignación dinámica de memoria**. La implementación de una **pila** como una **estructura dinámica** se realiza en lenguaje **C** mediante operaciones de asignación dinámica de memoria. Para la asignación dinámica de memoria es necesario emplear la función **`malloc`** (regresa un apuntador al espacio asignado) para crear un espacio de memoria para el tipo de dato y la función **`free`** para liberar la memoria, por lo que las operaciones **push** y **pop**, deberán emplear estas funciones.

Ejemplo 2.2: Realizar un programa para evaluar expresiones, con los operadores **+,-,*** y **/**, y en la cual se utilice la notación **postfija** (Implementar con **pilas dinámicas**).

En el **Programa 2.1.2**, se observa la implementación de una **pila dinámica**, mediante la declaración de un **Tipo Abstracto de Datos** (**TAD**), llamado **`elemento`**, y cuya estructura **`datos`** representa cada elemento en la pila. En la estructura **`datos`** se encuentra definido el tipo de dato (**`dato`**) a emplear y un apuntador al elemento **`anterior`**.

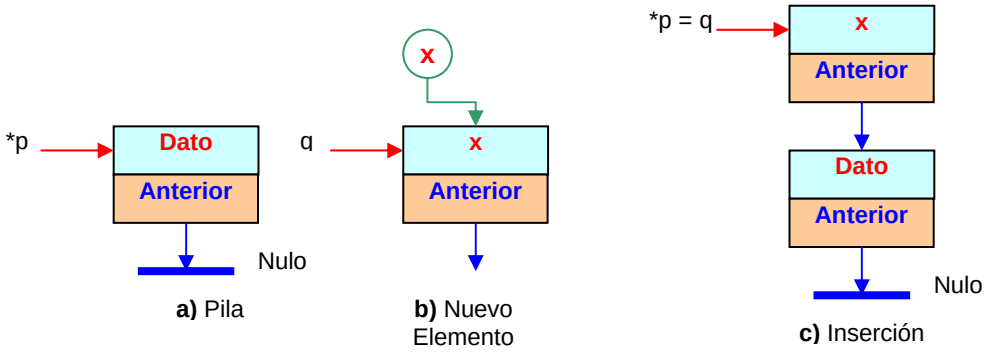
Cada elemento de la pila se encuentra constituido por una estructura, la cual contiene el tipo del elemento y un apuntador al elemento anterior (**Figura 2.5**).

a) Pila Vacía **b)** Un elemento en la Pila

Figura 2.5: Una nueva pila y un elemento en la pila.

Para la implementación de la **operación push** (**Figura 2.6**) se siguen las siguientes acciones:

- Crear un elemento para el elemento **x**, que se agregará a la pila.
- Insertar el elemento.
- Mover el tope y el apuntador al elemento siguiente.



a) Pila **b)** Nuevo Elemento **c)** Inserción

Figura 2.6: Inserción de un elemento en la pila.

Para la implementación de la **operación pop** (**Figura 2.7**) se siguen las siguientes acciones:

- Si la pila esta vacía, enviar un mensaje.
- Copiar el elemento a regresar, borrar el elemento de la pila.
- Regresar el elemento que se encontraba en la pila.

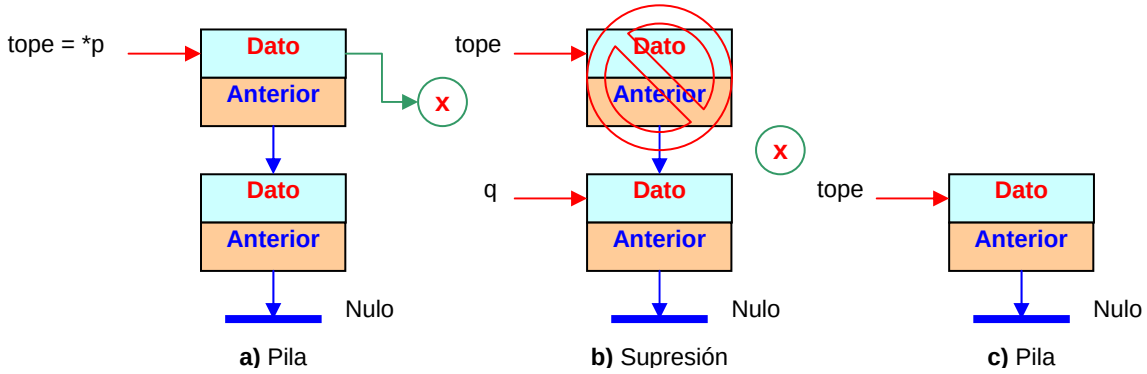


Figura 2.7: Supresión de un elemento en la pila.

Para evitar que se pierda el control del programa, al emplear una dirección de memoria, la cual no ha sido asignada, es necesario verificar que el nuevo elemento, tenga espacio, esto se hace en la función **Nuevo**. El **Programa 2.2** es igual que en el **Ejemplo 2.1**, solo que se ha implementado con una pila en la cual los elementos se asignan de forma dinámica.

```
/* Librerías                                                          */
#include "stdio.h"
#include "stdlib.h"
#include "conio.h"
#define     Pila_Vacia  (tope == NULL)    /* Pila Vacía              */

typedef struct datos elemento;      /* Define tipo de elemento       */
struct datos{
float dato;                   /* Dato de tipo real                   */
elemento *anterior;           /* Apuntador al elemento anterior      */
};

/* Función de error                                                   */
void error(void){
perror("\n\aERROR: Memoria insuficiente...");
exit(1);
}

/* Crea un nuevo elemento                                             */
elemento *Nuevo(){
elemento *q = (elemento *)malloc(sizeof(elemento));
if(!q) error();
return(q);
}
```

Programa 2.2: Definición de **pila dinámica** para **evaluar expresiones postfijas** (Parte **1**/3).

```
/* Añade un elemento a la pila                                          */
void push(elemento **p,float x){

elemento *q,*tope;
tope = *p;                    /* Tope de la pila                        */
q = Nuevo();
q -> dato = x;
q -> anterior = tope;
tope = q;
*p = tope;
}

/* Recupera un elemento de la pila                                      */
float pop(elemento **p){
elemento *tope;
float x;
tope = *p;                    /* Tope de la pila                        */
if(Pila_Vacia){
      printf("\n\a\tError: POP = Pila Vacía...\n");
      return(0);
      }
x = tope -> dato;
*p = tope -> anterior;
free(tope);
return(x);
}

/* Carga la expresión en un arreglo                                     */
void lee(char ent[]){
int pos = 0;
printf("\n\n\tEntra la Expresión Postfija : ");
while((ent[pos++] = getchar()) != '\n');
ent[--pos] = '\0';
}

/* Evalúa la expresión                                                  */
float evalua(char ent[]){
float a,b;
char op[1];
elemento *pila = NULL;        /* Pila Vacía                             */
int pos = 0;
while(ent[pos] != '\0'){
      *op = ent[pos++];
      switch(*op){
            case '+':
                  b = pop(&pila);
                  a = pop(&pila);
                  push(&pila,a + b);
                  break;
            case '-':
                  b = pop(&pila);
                  a = pop(&pila);
                  push(&pila,a - b);
                  break;
```

Programa 2.2: Definición de **pila dinámica** para **evaluar expresiones postfijas** (Parte **2**/3).

```
            case '*':
                  b = pop(&pila);
                  a = pop(&pila);
                  push(&pila,a * b);
            break;
            case '/':
                  b = pop(&pila);
                  a = pop(&pila);
                  if(b == 0.){
                        printf("\n\t\aDivisión por CERO\n");
                        return(0.);
                        break;
                        }
                  push(&pila,a / b);
                  break;
            default:
                  push(&pila,atof(op)); /* Almacena dato en la pila    */
            }
      }
return(pop(&pila));
}

/* Programa principal */
void main(void){
char ent[100];
clrscr();
printf("Calculadora Básica.");
printf("Emplea Notación postfija: 2 + 5  -> 25+");
cprintf("Operaciones: Suma +, Resta -, Producto * y División /");
lee(ent);
printf("\n\n\t%s = %g",ent,evalua(ent));
}
```

Programa 2.2: Definición de **pila dinámica** para **evaluar expresiones postfijas** (Parte **3**/3).

Al ejecutar el programa anterior produce el resultado que se muestra en la **Figura 2.8**.

```
                        Calculadora Básica.
Emplea Notación postfija: 2 + 5  -> 25+
Operaciones: Suma +, Resta -, Producto * y División
Entra la Expresión Postfija : 4837+*2/+
4837+*2/+ = 44
```

Figura 2.8: Resultados del programa para evaluar expresiones postfijas con pila dinámica.

La expresión evaluada en su forma **Infija** es: **((7 + 3) * 8) / 2 + 4 = 44**

2.2 Colas

Una **cola** es un conjunto ordenado de elementos del que pueden **suprimirse** por un extremo llamado **frente** y en el que pueden **insertarse** elementos en el otro extremo llamado **final** **[1, 7, 8, 10, 11, 17** y **18]**.

Por ejemplo tenemos cualquier fila, por ejemplo la fila en un banco. Este tipo de listas recibe el nombre de listas **FIFO** (First In - First Out ó **PEPS** Primero en Entrar - Primero en Salir). Este orden, es la única forma de insertar y recuperar un elemento de la cola (**Figura 2.9**).

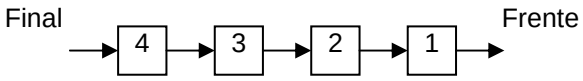


Figura 2.9: Una cola con sus elementos frente y final.

2.2.1 Implementación estática de colas

Para la **estructura de datos cola**, se tiene un arreglo `cola[]`, y las posiciones `frente` y `final`, las cuales representan los **extremos de la cola**. También se conserva el número de elementos que se encuentran en la cola `tamaño`. Toda esta información puede ser parte de una estructura. Una cola con algunos valores se muestra en la **Figura 2.10**.

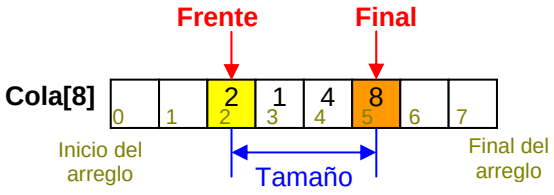


Figura 2.10: Una cola como arreglo.

Se presenta un problema cuando el arreglo se llena, por ejemplo para la cola de **8** elementos de la **Figura 2.10** se observa que una vez que se llegue al final del arreglo, ya no será posible seguir insertando elementos. A este problema se le conoce como **problema de desbordamiento**.

Una solución al problema de desbordamiento es dejar un extremo fijo (frente o final), sin embargo esta solución es costosa, debido a que cuando se realiza una inserción o supresión, hay que mover el resto de los elementos para conservar el extremo fijo.

Por otro lado se conoce que la cantidad de elementos en la cola, por lo regular se mantiene en un numero pequeño. La solución simple es que, siempre que `frente` y `final` lleguen al final del arreglo, regresen al inicio; por eso se refiere a este tipo de arreglos como **estructura circular**.

Un ejemplo de este tipo de cola se muestra en el **Programa 2.3**.

Para **insertar** un elemento `x`, incrementamos `tamaño` y `final`, luego hacemos `cola[final] = x`. Para **suprimir** un elemento, se decrementa `tamaño` y se incrementa `frente` y se pone el valor a devolver igual a `cola[frente]`.

```
#include "stdio.h"
#define MAX 100               /* Tamaño del arreglo cola                */

/* Definiciones de cola                                                 */
int cola[MAX];
int frente,final,tam;

/* Inserta un Elemento de la Cola                                       */
void inserta(int x){
tam++;
final++;
if(final >= MAX) final = 0;
cola[final] = x;
}

/* Suprime un elemento de la cola                                       */
int suprime(){
int x;
if(tam == 0) return(-1);
tam--;
frente++;
if(frente >= MAX) frente = 0;
x = cola[frente];
return(x);
}

/* Programa principal                                                   */
void main(void){
int x;
char c;
tam = 0;
frente = 0;
final = 0;
clrscr();
do    {
      printf("\n\t\tOperaciones con Colas");
      printf("\n\t>");
      c = tolower(getch());
      switch(c){
            case 'i':
                  printf("\n\tIntroduce elemento: ");
                  scanf("%d",&x);
                  inserta(x);
                  break;
            case 's':
                  x=suprime();
                  if(x<0) printf("\n\tCola Vacía"); /* Desbordamiento  */
                  else printf("\n\tElemento: %d",x);
                  break;
            }
      }while(c != 'q');
}
```

Programa 2.3: Implementación estática de una cola (Parte **2**/2).

En la **Figura 2.11** se muestra el resultado, al insertar y suprimir elementos.

```
        Operaciones con Colas
I = Insertar elemento
S = Suprimir elemento
Q = Salir
> I
Introduce elemento: 11
> I
Introduce elemento: 22
> S
Elemento: 11
> S
Elemento: 22
> S
Cola Vacía
```

Figura 2.11: Ejemplo del uso de colas.

2.2.2 Implementación dinámica de colas

La implementación de la **operación insertar** en una cola se ilustra en la **Figura 2.12**.

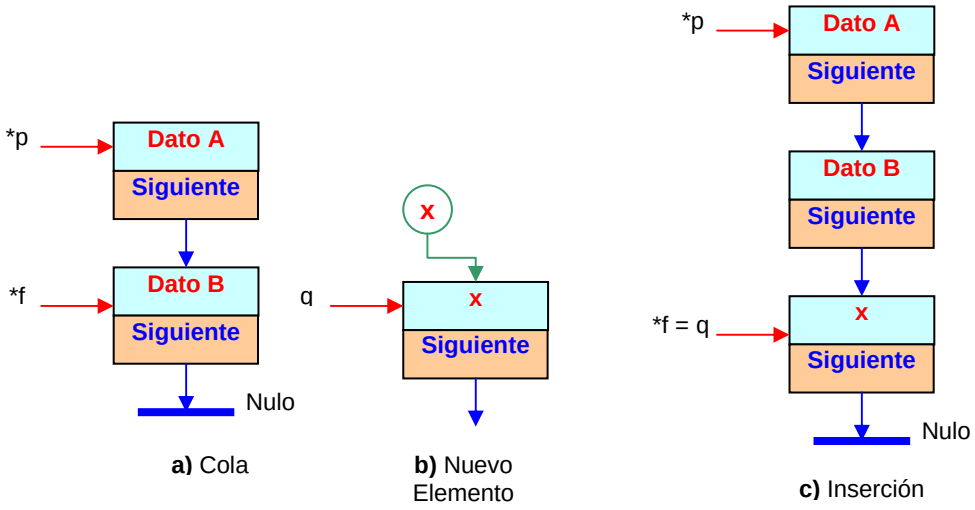


Figura 2.12: Inserción de un elemento en una cola.

Para la implementación de la **operación suprimir** se observa en la **Figura 2.13**.

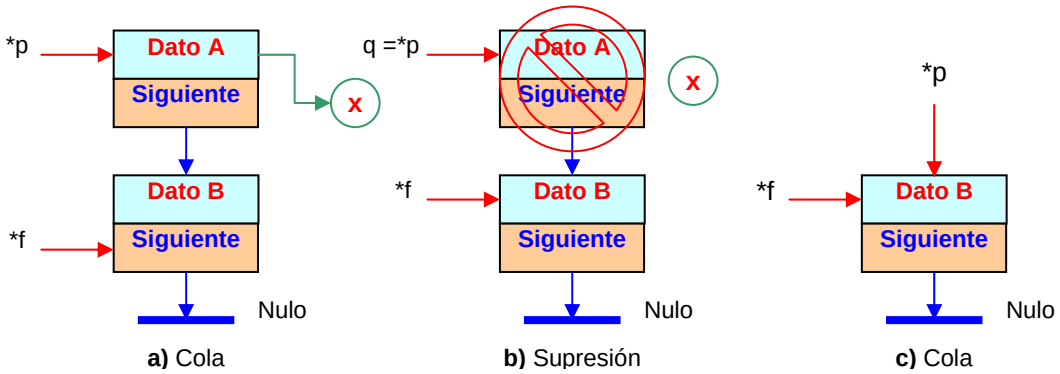


Figura 2.13: Supresión de un elemento en una cola.

La representación de la cola como un **tipo abstracto de datos**, es directa, al igual que para el caso de una pila. La implementación de las operaciones **insertar** y **suprimir** de forma dinámica, se observan en el **Programa 2.4**, en donde se maneja una cola de cadenas de caracteres.

```
#include "stdio.h"
#include "stdlib.h"
#include "string.h"
#include "conio.h"

/* Tipo de estructura y definición de estructura                    */
typedef struct datos elemento;
struct datos{
char dato[80];
elemento *siguiente;
};

/* Mensaje de ERROR                                                 */
void error(void){
perror("\n\t\aERROR: Memoria insuficiente...");
exit(1);
}

/* Crea un nuevo elemento del tipo de la estructura                 */
elemento *Nuevo(){
elemento *q = (elemento *)malloc(sizeof(elemento));
if(!q) error();
return(q);
}

/* Pone un Menú                                                     */
void menu(void){
printf("Uso de  Operaciones con Colas");
printf("I = Insertar un Dato");
printf("S = Suprimir un Dato");
printf("Q = Salir");
printf("Elija una Opción : ");
}

/* Inserta elementos al final de la cola                            */
void insertar(elemento **p,elemento **f,char dato[]){
elemento *pc,*fc,*q;    /* pc = Frente y fc = Final de la cola      */
pc = *p;
fc = *f;
q = Nuevo();
strcpy(q -> dato,dato);
q -> siguiente = NULL;
if(fc == NULL) pc = fc = q;
else fc = fc -> siguiente = q;
*p = pc;
*f = fc;
}
```

Programa 2.4: Implementación dinámica de una cola (Parte **1**/2).

```
/* Saca y elimina un dato del frente de la cola                          */
char *suprimir(elemento **p,elemento **f){
elemento *pc,*fc,*q;    /* pc = Frente y fc = Final de la cola           */
char *dato;
pc = *p;
fc = *f;
if(pc != NULL){
      q = pc;
      dato = (char *)malloc(strlen(q -> dato) + 1);
      strcpy(dato,q -> dato);
      pc = pc -> siguiente;
      if(pc == NULL) fc = NULL;
      free(q);
      *p = pc;
      *f = fc;
      return(dato);
      }
return("¡No hay más Datos!");
}

/* Programa Principal                                                    */
void main(void){
elemento *frente,*final;
char opcion,dato[80];
frente = NULL;
final = NULL;
while(1){
      do    {
            clrscr();
            menu();
            opcion = tolower(getch());
            }while(opcion != 'i' && opcion != 's' && opcion != 'q');
      clrscr();
      switch(opcion){
            case 'i':
                  printf("\n\t Introducir un Dato : ");
                  gets(dato);
                  insertar(&frente,&final,dato);
                  break;
            case 's':
                  strcpy(dato,suprimir(&frente,&final));
                  if(*dato) printf("Dato : %s",dato);
                  printf("\n\t Presione alguna tecla para continuar...");
                  getch();
                  break;
            case 'q':exit(0);
            }
      }
}
```

Programa 2.4: Implementación dinámica de una cola (Parte **2**/2).

En la **Figura 2.14** se muestra el resultado, de insertar los elementos `Elemento 1` y `Elemento 2` y suprimir los elementos insertados al ejecutar el **Programa 2.4**.

```
        Uso de  Operaciones con Colas
I = Insertar un Dato
S = Suprimir un Dato
Q = Salir
Elija una Opción:
I> Introducir un Dato: Elemento 1
I> Introducir un Dato: Elemento 2
S> Dato: Elemento 1
Presione alguna tecla para continuar...
S> Dato: Elemento 2
Presione alguna tecla para continuar...
S> Dato: ¡No hay más Datos!
Presione alguna tecla para continuar...
```

Figura 2.14: Ejemplo del uso de colas con implementación dinámica.

2.2.3 Colas de prioridad

En las estructuras **pila** y **cola**, los elementos están ordenados en base al **orden en que se insertaron**, las operaciones de **insertar** y **suprimir** no toman en cuenta el orden intrínseco de los elementos; es decir, si se encuentran el orden alfabético o numérico.

En una **cola**, los elementos pueden ser eliminados de forma distinta a la forma descrita, creándose en este caso **prioridades**. En la vida real, la **prioridad** se logra por finalidad, por parentesco, por propina, por soborno, etc.

La **cola de prioridad**, es una estructura de datos en la que el ordenamiento intrínseco de los elementos determina el resultado de sus operaciones básicas. Existen dos tipos de colas de prioridad. La **cola de prioridad ascendente**, que es una colección de elementos en la que pueden **insertarse** elementos de manera **arbitraria** y de la que puede **eliminarse** solo el **elemento menor**.

La **cola de prioridad descendente** es similar, pero solo se permite la **eliminación** del **elemento mayor**.

Los elementos de una **cola de prioridad** no necesariamente son números o caracteres, los cuáles pueden compararse de forma directa. Pueden ser estructuras complejas ordenadas en uno o varios campos. Las colas de prioridad se emplean en modelación. El objetivo es modelizar algún sistema físico (reacción química, sistema meteorológico, línea de montaje, etc.), de tal forma que podamos analizar y predecir su comportamiento.

Hay varias formas obvias de **implementar** de forma simple una **cola de prioridad.** Podríamos realizar inserciones en un extremo (en un tiempo **1**), y recorrer los elementos (en un tiempo **n**), para eliminar el mínimo. Alternativamente podríamos ordenar siempre los elementos de la cola, lo que haría a las inserciones costosas (en un tiempo **n**) y eliminar el mínimo (en un tiempo **1**). La primera opción es la menos costosa, por que nunca hay más eliminaciones que inserciones (**Tabla 2.2**).

Implementación	Inserción	Supresión
Inserciones en un extremo	Toma un tiempo corto.	Es muy costosa, por que hay que buscar el elemento mínimo en toda la cola.
Ordenar los elementos al insertarlos	Es muy costosa, por que hay que buscar el punto de inserción en toda la cola.	Toma un tiempo corto.

Tabla 2.2: Características de las formas de implementación de una cola de prioridad.

A manera de ejemplo, simularemos una cola ascendente de números enteros, en la cual los datos son insertados mediante asignación dinámica de memoria, buscando el punto de inserción y cambiando solo los apuntadores que indican el elemento precedente y sucesor.

El programa incluye las siguientes funciones (**Programa 2.5**):

- **Insertar elemento**.- inserta un elemento, al principio de la cola, o después de un elemento, de acuerdo a un orden ascendente, en donde el frente de la cola es el elemento menor y por lo tanto el que será eliminado.
- **Suprimir un elemento**.- elimina y muestra al elemento menor de la cola.
- **Ver elementos de la cola**.- realiza un recorrido por los elementos de la cola, y muestra sus valores actuales.

```
#include "conio.h"

/* Tipo de estructura y definición de estructura                    */
typedef struct datos elemento;
struct datos{
int dato;
elemento *siguiente;
};

/* Mensaje de ERROR                                                 */
void error(void){
perror("\n\t\aERROR: Memoria insuficiente...");
exit(1);
}

/* Crea un nuevo elemento del tipo de la estructura                 */
elemento *Nuevo(){
elemento *q = (elemento *)malloc(sizeof(elemento));
if(!q) error();
return(q);
}

/* Presenta el Menú                                                 */
void menu(void){
printf("Colas de Prioridad");
printf("I = Insertar un Dato");
printf("S = Suprimir un Dato");
printf("V = Ver todos los elementos");
printf("Q = Salir");
cprintf("Elija una Opción : ");
}

/* Inserta un elemento de forma ordenada en la cola                 */
void insertar_asc(elemento **p,int dato){
elemento *pf = *p;
elemento *actual = pf, *anterior = pf,*q;
if(pf == NULL){         /* Si está vacía crear un nuevo elemento    */
      pf = Nuevo();
      pf -> dato = dato;
      pf -> siguiente = NULL;
      *p = pf;
      return;
      }
```

Programa 2.5: Operaciones con colas de prioridad (Parte **1**/3).

```
/* Busca en la cola un punto de inserción                                  */
while(actual != NULL && dato > actual -> dato){
      anterior = actual;
      actual = actual -> siguiente;
      }
/* Se inserta al principio o al final                                      */
q = Nuevo();
q -> dato = dato;
if(anterior == actual){
      q -> siguiente = pf;
      pf = q;
      }
else  {
      q -> siguiente = actual;
      anterior -> siguiente = q;
      }
*p = pf;
}

/* Saca y elimina un dato de la cola                                       */
int suprimir_asc(elemento **p){
elemento *frente,*q;
int dato;
frente = *p;
if(frente != NULL){
      q = frente;
      dato = q -> dato;
      frente = frente -> siguiente;
      free(q);
      *p = frente;
      return(dato);
      }
printf("\n\tCola Vacía");
return(-1);
}

/* Recorre la cola y muestra los datos                                     */
void ver(elemento *cola){
elemento *actual = cola;
int i;
printf("\n\tDatos de la Cola");
if(actual == NULL) printf("\n\tCola vacía...");
else  {
      i = 0;
      while(actual != NULL){
            printf("\n\tElemento %d = %d",i++,actual -> dato);
            actual = actual -> siguiente;
            }
      }
getch();
}
```

Programa 2.5: Operaciones con colas de prioridad (Parte **2**/3).

```
/* Programa Principal                                                    */
void main(void){
elemento *cola;
int opcion,dato;
cola = NULL;            /* Inicia una cola vacía                         */
while(1){
      do    {
            clrscr();
            menu();
            opcion=tolower(getch());
            }while(opcion != 'i' && opcion != 's' &&
                   opcion != 'v' && opcion != 'q');
      clrscr();
      switch(opcion){
            case 'i':
                  printf("\n\tIntroducir un Dato : ");
                  scanf("%d",&dato);
                  insertar_asc(&cola,dato);
                  break;
            case 's':
                  dato = suprimir_asc(&cola);
                  if(dato > 0) printf("Dato : %d",dato);
                  printf("\n\tPresione alguna tecla para continuar...");
                  getch();
                  break;
            case 'v':
                  ver(cola);
                  break;
            case 'q':exit(0);
            }
      }
}
```

Programa 2.5: Operaciones con colas de prioridad (Parte **3**/3).

En la **Figura 2.15** se muestra el resultado, al insertar los datos `15, 5, 8` y `50`, ver y suprimir elementos después de ejecutar el **Programa 2.5**.

```
        Colas de Prioridad
I = Insertar un Dato      S = Suprimir un Dato      V = Ver todos los elementos
Q = Salir
Elija una Opción:
> i
Introducir un Dato: 50
> v
Datos de la Cola
Elemento 0 = 5
Elemento 1 = 8
Elemento 2 = 15
Elemento 3 = 50
> s
Dato: 5
Presione alguna tecla para continuar...
```

Figura 2.15: Ejemplo del uso de colas de prioridad.

Otro ejemplo del uso de colas de prioridad, es la de **agregar cadenas** de forma ordenada en un archivo.

Agregación de cadenas.- agregar cadenas a un archivo, empleando una cola de prioridad. El programa debe de recibir parámetros de la línea de comandos y de no ser así, usar parámetros por omisión.

Modificamos el programa anterior, para que acepte un tipo de datos cadena, además de realizar las rutinas para `guardar` y `cargar` los datos desde un archivo. El **Programa 2.6** muestra la forma de implementar estas rutinas.

```
#include "stdio.h"
#include "stdlib.h"
#include "string.h"
#include "conio.h"

#define MAX 100         /* Tamaño máximo del arreglo de entrada        */

/* Tipo de estructura y definición de estructura                       */
typedef struct datos elemento;
struct datos{
char dato[MAX];
elemento *siguiente;
};

/* Mensaje de ERROR                                                    */
void error(void){
perror("\n\t\aERROR: Memoria insuficiente...");
exit(1);
}

/* Crea un nuevo elemento del tipo de la estructura                    */
elemento *Nuevo(){
elemento *q = (elemento *)malloc(sizeof(elemento));
if(!q) error();
return(q);
}

/* Pone un Menú                                                        */
void menu(void){
printf("Colas de Prioridad");
printf("I = Insertar un Dato");
printf("S = Suprimir un Dato");
printf("V = Ver todos los elementos");
printf("Q = Salir");
printf("Elija una Opción : ");
}

/* Inserta un elemento de forma ordenada en la cola                    */
void insertar_asc(elemento **p,char dato[MAX]){
elemento *pf = *p;
elemento *actual = pf, *anterior = pf,*q;
if(pf == NULL){
      pf = Nuevo();
      strcpy(pf -> dato,dato);
      pf -> siguiente = NULL;
      *p = pf;
      return;
      }
```

Programa 2.6: Agregación de cadenas con colas de prioridad (Parte **1**/4).

```
/* Busca en la lista un punto de inserción                              */
while(actual != NULL && strcmp(dato,actual -> dato) > 0){
      anterior = actual;
      actual = actual -> siguiente;
      }
/* Se inserta al principio o al final                                   */
q = Nuevo();
strcpy(q -> dato,dato);
if(anterior == actual){
      q -> siguiente = pf;
      pf = q;
      }
else  {
      q -> siguiente = actual;
      anterior -> siguiente = q;
      }
*p = pf;
}

/* Saca y elimina un dato de la cola                                    */
char *suprimir_asc(elemento **p){
elemento *frente,*q;
char dato[MAX];
frente = *p;
if(frente != NULL){
      q = frente;
      strcpy(dato,q -> dato);
      frente = frente -> siguiente;
      free(q);
      *p = frente;
      return(dato);
      }
return("Cola Vacía");
}

/* Recorre la cola y muestra datos                                      */
void ver(elemento *cola){
elemento *actual = cola;
int i;
printf("\n\tDatos de la Cola");
if(actual == NULL) printf("\n\tCola vacía...");
else  {
      i = 0;
      while(actual != NULL){
            printf("\n\tElemento %d = %s",i++,actual -> dato);
            actual = actual -> siguiente;
            }
      }
getch();
}
```

Programa 2.6: Agregación de cadenas con colas de prioridad (Parte **2**/4).

```
/* Guarda datos en el archivo de salida                                  */
void guarda(elemento *cola,char sal[20]){
elemento *actual = cola;
FILE *fp;
if((fp = fopen(sal,"w")) == NULL){
      printf("\n\tNo se pudo abrir el archivo...");
      return;
      }
if(actual == NULL){
      fclose(fp);
      return;
      }
while(actual != NULL){
      fprintf(fp,"%s\n",actual -> dato);
      actual = actual -> siguiente;
      }
fclose(fp);
}
/* Carga datos desde el archivo de entrada                               */
void carga(elemento **p,char ent[20]){
char dato[MAX],arc;
int pos;
FILE *fp;
if((fp = fopen(ent,"r")) == NULL){
      printf("\n\tNo se pudo abrir el archivo...");
      return;
      }
arc = getc(fp);              /* Carga primer carácter                    */
while(arc != EOF){
      pos = 0;
      do dato[pos++] = arc;
      while((arc = getc(fp)) != '\n' && arc != EOF);
      dato[pos] = '\0';
      arc = getc(fp);        /* Salta salto de línea                     */
      insertar_asc(p,dato);
      }
fclose(fp);
}

/* Programa Principal                                                    */
void main(int argc, char *argv[]){
elemento *cola;
int opcion,pos;
char dato[MAX];
char ent[20],sal[20];
cola = NULL;
if(argc == 1){
      strcpy(ent,"agda1.txt");
      strcpy(sal,"agda1.txt");
      }
if(argc == 2){
      strcpy(ent,argv[1]);
      strcpy(sal,"agda1.txt");
      }
```

Programa 2.6: Agregación de cadenas con colas de prioridad (Parte **3**/4).

```
if(argc == 3){
      strcpy(ent,argv[1]);
      strcpy(sal,argv[2]);
      }
carga(&cola,ent);
while(1){
      do    {
            clrscr();
            menu();
            opcion = tolower(getch());
            }while(opcion != 'i' && opcion != 's' &&
                   opcion != 'v' && opcion != 'q');
      clrscr();
      switch(opcion){
            case 'i':
                  pos = 0;
                  printf("\n\tIntroducir un Dato : ");
                  while((dato[pos++] = getchar()) != '\n');
                  dato[--pos] = '\0';
                  insertar_asc(&cola,dato);
                  break;
            case 's':
                  strcpy(dato,suprimir_asc(&cola));
                  if(*dato) printf("Dato : %s",dato);
                  printf("\n\tPresione alguna tecla para continuar...");
                  getch();
                  break;
            case 'v':
                  ver(cola);
                  break;
            case 'q':
                  guarda(cola,sal);
                  exit(0);
            }
      }
}
```

Programa 2.6: Agregación de cadenas con colas de prioridad (Parte **4**/4).

En las **Figuras 2.16** y **2.17** se muestran los resultados, después de ejecutar el **Programa 2.6**.

```
        Colas de Prioridad
I = Insertar un Dato
S = Suprimir un Dato
V = Ver todos los elementos
Q = Salir
Elija una Opción:
> I
Introducir un Dato: Último elemento a insertar
> I
Introducir un Dato: Alguna cita por ejemplo
> I
Introducir un Dato: 2 Primer elemento
> V
Datos de la Cola
Elemento 0 = 2 Primer elemento
Elemento 1 = Alguna cita por ejemplo
Elemento 2 = Último elemento a insertar
```

Figura 2.16: Ejemplo del uso de colas de prioridad para agregar cadenas.

```
2 Primer elemento
Alguna cita por ejemplo
Último elemento a insertar
```

Figura 2.17: Contenido del archivo `agda1.txt`.

2.2.4 Aplicaciones de colas

Algunas aplicaciones de colas son las siguientes:

- Cuando se envían trabajos a una impresora, se quedan en orden de llegada.
- En teoría, toda fila real es una cola. Por ejemplo las colas de las taquillas, son colas, pues se atiende primero a quién llega primero.
- En redes de computadoras, cuando todos los usuarios se encuentran conectados a un disco servidor, la primera petición en llegar es atendida primero.
- Las llamadas telefónicas en una compañía se colocan en una cola de espera.
- Una rama completa de las matemáticas, trata del estudio de la llamada **teoría de colas**, se ocupa de realizar cálculos probabilísticos del tiempo que por ejemplo deben de esperar los usuarios al llegar a una fila.
- En una línea de producción, la línea puede ser modelada como cola, y cada estación de trabajo como un elemento dentro de la línea, con sus tiempos respectivos.
- Simulación de tráfico en un cruce con semáforos.
- Para establecer la circulación en una estación de gasolina.

2.3 Listas Enlazadas

2.3.1 Lista lineal

Cuando se desea una lista de elementos u objetos de cualquier tipo, originalmente vacía que durante la ejecución del programa vaya creciendo o decreciendo elemento a elemento, según las necesidades del programa, entonces se tiene que construir una **lista lineal**, en la que cada elemento apunte o direccione al siguiente **[1, 2, 7, 8, 10, 17** y **19]**. A la **lista lineal** también se le conoce como **lista enlazada** ó **lista vinculada lineal**.

Las listas, son una de las estructuras de datos fundamentales; son usadas para almacenar elementos. Una lista es una **estructura dinámica**, es decir, que cambia con el tiempo. Al primer elemento de una lista se le conoce como **cabeza de la lista**; mientras que al último elemento se le llama **cola de la lista**.

En una lista, a cada elemento se le conoce como **nodo**. Cada nodo puede contener un campo o más de información (llave, clave, dato, etc.), además de un elemento que apunta al **siguiente** nodo de la lista.

Una lista se dice que es **homogénea** cuando todos los elementos almacenados en la lista son del mismo tipo. Una **lista heterogénea** es aquella lista que tiene distintos tipos de elementos.

Para construir una lista lineal, primero se debe de definir la clase de objetos que van a formar parte de la misma, por ejemplo para declarar un tipo denominado **nodo**, se debe de declarar uno de sus miembros como un apuntador a un objeto del mismo tipo (**Figura 2.18**).

```
typedef struct datos nodo;
struct datos{
int dato;         /* Información, dato, clave, llave, key, etc.        */
nodo *siguiente;  /* Apuntador al siguiente nodo                       */
};

main(){
...
nodo *p;          /* Apuntador p a un objeto de tipo nodo              */
p = Nuevo();      /* Asigna espacio al objeto de tipo nodo             */
p -> siguiente = NULL; /* Indica que después de este ya no hay otro    */
...
}
```

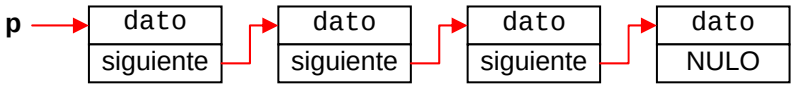


Figura 2.18: Definición de una lista **p** con 4 nodos enteros.

Para tener acceso a una lista, se debe de hacer desde un extremo, por lo que debe de existir un apuntador al **primer nodo** de la lista (**p**). También el último elemento de la lista debe de tener un apuntador **NULO**, para indicar el **fin de la lista**.

Una **lista vacía**, es aquella que no tiene ningún nodo.

Un **nodo de cabecera**, es un nodo que puede formar parte de la lista, pero que se mantiene en la parte delantera de la lista. Puede ser empleado por ejemplo, para guardar información del numero de nodos; en este caso, cada vez que se realice una inserción, este debe de ser actualizado, sin embargo, se puede conocer el numero de nodos de la lista leyendo solo este nodo.

2.3.2 Operaciones primitivas de una lista

Las operaciones fundamentales que podemos realizar con una lista son las siguientes:

- Inserción de un nodo en la lista.
- Supresión de un nodo en la lista.
- Recorrido de una lista.
- Búsqueda de un nodo en la lista.

Inserción de un nodo al inicio de la lista

Primero, se crea un nodo y después se le asignan los apuntadores, como se indica a continuación:

```
q = Nuevo();              /* Crea un nuevo nodo                          */
q -> dato = n;            /* Se le asigna el valor n                     */
q -> siguiente = p;       /* Se reasigna el apuntador                    */
p = q;                    /* Apuntador actual                            */
```

De esta forma, el orden de los nodos en una lista, es el inverso del orden en que han llegado.

Inserción de un nodo en cualquier parte de la lista

La inserción de un nodo en la lista, a continuación de otro apuntado por **p**, se realiza de la forma:

```
q = Nuevo();              /* Crea un nuevo nodo                          */
q -> dato = n;            /* Se le asigna el valor insertado n           */
q -> siguiente = p -> siguiente;    /* Se reasigna el apuntador          */
p -> siguiente = q;       /* Apuntador actual                            */
```

Este proceso se observa en **Figura 2.19**.

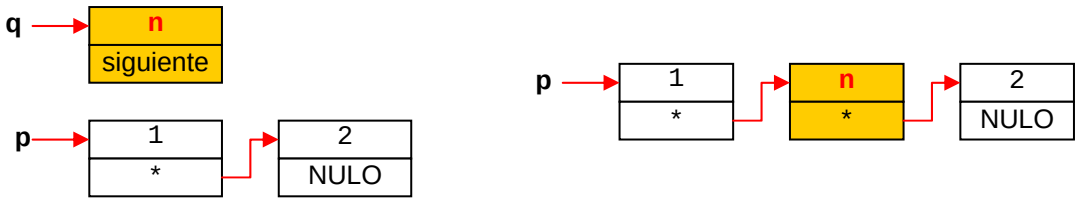


Figura 2.19: Inserción de un nodo en una lista.

Supresión de un nodo de la lista

Para suprimir el nodo siguiente apuntado por **p**, se tiene que:

```
q = p -> siguiente;       /* q apunta al siguiente de p                  */
p -> siguiente = q -> siguiente;    /* p apunta al siguiente de p        */
free(q);                  /* Libera memoria asignada al nodo             */
```

Para suprimir el nodo apuntado por **p**, se tiene que realizar:

```
q = p -> siguiente;       /* q apunta al siguiente de p                  */
*p = *q;                  /* q se asigna a p                             */
free(p);                  /* Libera memoria asignada al nodo             */
```

Este proceso se observa en **Figura 2.20**.

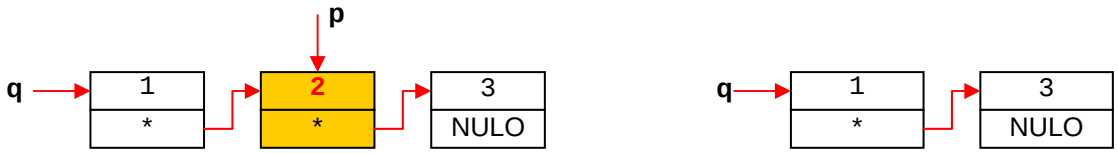


Figura 2.20: Supresión de un nodo en una lista.

Recorrido de una lista

Cuando se requiere realizar alguna operación con todos los nodos de una lista, cuyo primer nodo se encuentra apuntado por **p**, se realizan las siguientes acciones, si por ejemplo, queremos imprimir el valor contenido de cada nodo.

```
q = p;            /* Almacena apuntador al primer nodo de la lista     */
while(q != NULL){       /* Mientras no se encuentre el último nodo     */
printf("%d",q -> dato); /* Imprime el valor del dato                   */
q = q -> siguiente;     /* Apunta al siguiente nodo                    */
}
```

Búsqueda de un nodo de la lista

Por ejemplo, para una búsqueda secuencial, se busca nodo a nodo, termina cuando se encuentra el nodo **x** buscado ó cuando se llega el final de la lista.

```
q = p;            /* Almacena apuntador al primer nodo de la lista     */
while(q != NULL && q -> dato != x) /* Si no se encuentra, continua     */
q = q -> siguiente;     /* Apunta al siguiente nodo                    */
```

La implementación en **lenguaje C** de las operaciones básicas de una lista, se muestra en el **Programa 2.7**. En el programa, los datos se insertan en orden ascendente.

```
#include "stdio.h"
#include "stdlib.h"

/* Tipo de estructura y definición de estructura                       */
typedef struct datos nodo;
struct datos{
int dato;
nodo *siguiente;
};

/* Mensaje de ERROR                                                    */
void error(void){
perror("\n\t\aERROR: Memoria insuficiente...");
exit(1);
}

/* Crea un nuevo nodo del tipo de la estructura                        */
nodo *Nuevo(){
nodo *q = (nodo *)malloc(sizeof(nodo));
if(!q) error();
return(q);
}
```

Programa 2.7: Operaciones Básicas con listas enlazadas (Parte **1**/4).

```
/* Pone un Menú                                                    */
void menu(void){
printf("Operaciones Básicas con Listas");
printf("I = Insertar un Nodo");
printf("S = Suprimir un Nodo");
printf("T = Invierte los elementos de la lista");
printf("B = Buscar un Nodo");
printf("V = Ver Todos los Nodos");
printf("Q = Salir");
printf("Elija una Opción : ");
}

/* Inserta un nodo de forma ordenada                               */
void insertar(nodo **p,int dato){
nodo *pf = *p;
nodo *actual = pf, *anterior = pf,*q;
if(pf == NULL){
      pf = Nuevo();
      pf -> dato = dato;
      pf -> siguiente = NULL;
      *p = pf;
      return;
      }
/* Busca en la lista un punto de inserción                         */
while(actual != NULL && dato > actual -> dato){
      anterior = actual;
      actual = actual -> siguiente;
      }
/* Se inserta al principio o al final                              */
q = Nuevo();
if(anterior == actual){
      q -> dato = dato;
      q -> siguiente = pf;
      pf = q;
      }
else  {
      q -> dato = dato;
      q -> siguiente = actual;
      anterior -> siguiente = q;
      }
*p = pf;
}

/* Busca y elimina un dato de la lista                             */
void suprimir(nodo **p,int dato){
nodo *cabecera = *p;
nodo *actual = cabecera, *anterior = cabecera;
if(cabecera == NULL){
      printf("\n\tLista Vacía...");
      return;
      }
while(actual != NULL && dato != actual -> dato){
      anterior = actual;
      actual = actual -> siguiente;
      }
if(actual == NULL) {
```

Programa 2.7: Operaciones Básicas con listas enlazadas (Parte **2**/4).

```
      printf("\n\tEl dato %d no esta en la lista...",dato);
      return;     /* No está  el dato                                    */
      }
/* Borrar dato de la lista                                               */
if(anterior == actual) cabecera = cabecera -> siguiente;
else anterior -> siguiente = actual -> siguiente;
free(actual);
*p = cabecera;
}

/* Recorre la lista y muestra sus datos                                  */
void ver(nodo *lista){
nodo *actual = lista;
int i;
printf("\n\tDatos de la Lista");
if(actual == NULL) printf("\n\tLista vacía...");
else  {
      i = 0;
      while(actual != NULL){
            printf("\n\tNodo %d = %d",i++,actual -> dato);
            actual = actual -> siguiente;
            }
      }
getch();
}

/* Busca un dato en la lista                                             */
nodo *busca(nodo *lista,int dato){
nodo *actual = lista;
while(actual != NULL && dato != actual->dato) actual = actual->siguiente;
return(actual);
}

/* Recorre la lista y e invierte sus datos                               */
void invierte(nodo **lista){
nodo *actual = *lista, *ant0 = *lista,*ant1 = *lista;
int i = 0;
printf("\n\tInvirtiendo...");
if(actual == NULL) printf("\n\tLista vacía...");
else  {
      while(actual != NULL){
            ant1 = actual;
            actual = actual -> siguiente;
            if(i == 0){
                  ant1 -> siguiente = NULL;
                  i = 1;
                  }
            else ant1 -> siguiente = ant0;
            ant0 = ant1;
            }
      *lista = ant1;
      }
}
```

Programa 2.7: Operaciones Básicas con listas enlazadas (Parte **3**/4).

```
/* Programa principal                                                    */
void main(void){
nodo *lista;
nodo *q;
int op,dato;
lista = NULL;
while(1){
      do    {
            clrscr();
            menu();
            op = tolower(getch());
            }while(op != 'i' && op!='b' && op != 's' && op != 't'
                  && op != 'v' && op != 'q');
      clrscr();
      switch(op){
            case 'i':
                  printf("\n\tIntroducir un Dato : ");
                  scanf("%d",&dato);
                  insertar(&lista,dato);
                  break;
            case 'b':
                  printf("\n\tDato a Buscar : ");
                  scanf("%d",&dato);
                  q = busca(lista,dato);
                  if(q) printf("\n\tDato %d si se encuentra",dato);
                  else printf("\n\tLista vacía...");
                  printf("\n\tPresione alguna tecla para continuar.");
                  getch();
                  break;
            case 's':
                  printf("\n\tDato a suprimir : ");
                  scanf("%d",&dato);
                  suprimir(&lista,dato);
                  printf("\n\tPresione alguna tecla para continuar.");
                  getch();
                  break;
            case 'v':
                  ver(lista);
                  break;
            case 't':
                  invierte(&lista);
                  break;
            case 'q':exit(0);
            }
      }
}
```

Programa 2.7: Operaciones Básicas con listas enlazadas (Parte **4**/4).

El resultado de la ejecución del **Programa 2.7** se muestra en la **Figura 2.21**.

```
          Operaciones Básicas con Listas
I = Insertar un Nodo
S = Suprimir un Nodo
B = Buscar un Nodo
V = Ver Todos los Nodos
Q = Salir
Elija una Opción:
> I
Introducir un Dato: 44
> V
Datos de la Lista
Nodo 0 = 0
Nodo 1 = 11
Nodo 2 = 44
Nodo 3 = 66
Nodo 4 = 88
>B
Dato a Buscar: 44
Dato 44 si se encuentra en la lista...
Presione alguna tecla para continuar.
>S
Dato a suprimir: 44
Presione alguna tecla para continuar.
```

Figura 2.21: Operaciones Básicas con listas enlazadas.

En las siguientes secciones se observa el uso de las operaciones básicas con listas enlazadas, para obtener la derivada de un polinomio y la suma y resta de dos polinomios.

2.3.3 Listas circulares

Una **lista circular**, es una lista lineal, en la que el último elemento se enlaza con el primero. Así, es posible acceder a cualquier elemento de la lista desde cualquier punto dado. Las operaciones en las listas circulares, son más simples, debido a que se eliminan los casos especiales.

Para encontrar el **final** de una lista circular, se debe de encontrar de nuevo el punto de inicio, por lo que debemos guardar el punto de partida, esto se hace, por ejemplo almacenando un **apuntador fijo** (**Figura 2.22**).

Otra forma de encontrar el **final**, es poner un elemento especial, conocido como **nodo de cabecera de la lista**, esto tiene la ventaja de que la lista nunca estará vacía.

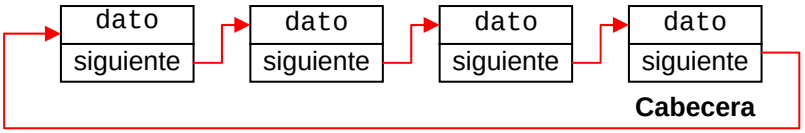


Figura 2.22: Lista circular, con un nodo de cabecera.

La implementación en **lenguaje C** de las operaciones básicas con listas circulares se observa en el **Programa 2.8** y en la **Figura 2.23**.

```
#include "stdlib.h"
#include "string.h"
#include "conio.h"

/* Tipo de estructura y definición de estructura                    */
typedef struct datos nodo;
struct datos{
int dato;
nodo *siguiente;
};

/* Mensaje de ERROR                                                 */
void error(void){
perror("\n\t\aERROR: Memoria insuficiente...");
exit(1);
}

/* Crea un nuevo nodo del tipo de la estructura                     */
nodo *Nuevo(){
nodo *q = (nodo *)malloc(sizeof(nodo));
if(!q) error();
return(q);
}

/* Pone un Menú                                                     */
void menu(void){
printf("Operaciones Básicas con Listas Circulares");
printf(" I = Insertar un Nodo");
printf(" S = Suprimir un Nodo");
printf(" B = Buscar un Nodo");
printf(" V = Ver Todos los Nodos");
printf(" Q = Salir");
cprintf("Elija una Opción : ");
}

/* Inserta un nodo en la lista                                      */
void insertar(nodo **p,int dato){
nodo *q,*lista;
lista = *p;
q = Nuevo();
q -> dato = dato;
if(lista == NULL) lista = q;
else q -> siguiente = lista -> siguiente;
lista -> siguiente = q;
*p = q;
}

/* Busca y elimina un dato de la lista                              */
void suprimir(nodo **p,int dato){
nodo *cabecera = *p,*final = *p;
nodo *actual = cabecera,*anterior = cabecera;
if(cabecera == NULL){
      printf("\n\tLista Vacía...");
      return;
      }
```

Programa 2.8: Operaciones básicas con listas circulares (Parte **1**/3).

```
do    {
      anterior = actual;
      actual = actual -> siguiente;
      }while(dato != actual -> dato && actual != final);
if(dato != actual -> dato) {
      printf("\n\tEl dato %d no está en la lista...",dato);
      return;     /* No está el dato                                    */
      }
/* Borrar dato de la lista                                             */
if(anterior == actual) cabecera = cabecera -> siguiente;
else anterior -> siguiente = actual -> siguiente;
if(actual == final) cabecera = anterior;
free(actual);
*p = cabecera;
}

/* Recorre la lista y muestra sus datos                                */
void ver(nodo *lista){
nodo *actual = lista,*final = lista;
int i;
printf("\n\tDatos de la Lista");
if(actual == NULL) printf("\n\tLista vacía...");
else  {
      i = 0;
      do    {
            actual = actual -> siguiente;
            printf("\n\tNodo %d = %d",i++,actual -> dato);
            }while(actual != final);
      }
getch();
}

/* Busca un dato en la lista                                           */
nodo *busca(nodo *lista,int dato){
nodo *actual = lista,*final = lista;
if(actual == NULL) return(NULL);
do    {
      actual = actual -> siguiente;
      if(dato == actual -> dato) return(actual);
      }while(actual != final);
return(NULL);
}

/* Programa principal                                                  */
void main(void){
nodo *lista;
nodo *q;
int op,dato,k;
lista = NULL;
while(1){
      do    {
            clrscr();
            menu();
            op = tolower(getch());
            }while(op != 'i' && op != 'b' && op != 's' &&
                   op != 'v' && op != 'q');
```

Programa 2.8: Operaciones básicas con listas circulares (Parte **2**/3).

```
      clrscr();
      switch(op){
            case 'i':
                  printf("\n\tIntroducir un Dato : ");
                  scanf("%d",&dato);
                  insertar(&lista,dato);
                  break;
            case 'b':
                  printf("\n\tDato a Buscar : ");
                  scanf("%d",&dato);
                  q = busca(lista,dato);
                  if(q) printf("\n\tDato %d si se encuentra
                        en la lista...",dato);
                  else  printf("\n\tLista vacía...");
                  printf("\n\tPresione alguna tecla para continuar.");
                  getch();
                  break;
            case 's':
                  printf("\n\tDato a suprimir : ");
                  scanf("%d",&dato);
                  suprimir(&lista,dato);
                  printf("\n\tPresione alguna tecla para continuar.");
                  getch();
                  break;
            case 'v':
                  ver(lista);
                  break;
            case 'q':exit(0);
            }
      }
}
```

Programa 2.8: Operaciones básicas con listas circulares (Parte **3**/3).

```
        Operaciones Básicas con Listas Circulares
I = Insertar un Nodo
S = Suprimir un Nodo
B = Buscar un Nodo
V = Ver Todos los Nodos
Q = Salir
Elija una Opción: I
Introducir un Dato: 5
Elija una Opción: V
Datos de la Lista
Nodo 0 = 8
Nodo 1 = 9
Nodo 2 = 3
Nodo 3 = 5
Elija una Opción: B
Dato a Buscar: 3
Dato 3 si se encuentra en la lista...
Presione alguna tecla para continuar.
```

Figura 2.23: Operaciones básicas con listas circulares.

2.3.4 Listas doblemente enlazadas

Una **lista doblemente enlazada**, es una lista lineal en la que cada elemento tiene dos enlaces, uno al nodo `siguiente` y otro al nodo `anterior`. Esto permite leer la lista en cualquier dirección (**Figura 2.24**).

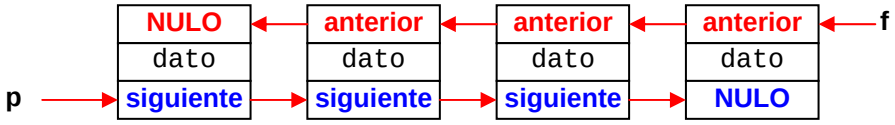


Figura 2.24: Lista doblemente enlazada.

Son tres las operaciones básicas que podemos realizar con una lista doblemente enlazada:

- **Insertar**.- comprende varios casos: **a)** insertar un elemento al **principio** de la lista, **b)** insertar un elemento **entre otros dos** y **c)** insertar un elemento al **final**.
- **Borrar**.- comprende los siguientes casos: **a)** borrar el **primer** elemento y **b)** borrar un elemento **cualquiera** que no sea el primero.
- **Ver**.- recorre e imprime los elementos de la lista.

La implementación se observa en el **Programa 2.9**.

```
#define MAX 80

/* Tipo de estructura y definición de estructura                    */
typedef struct datos nodo;
typedef nodo *pnodo;
struct datos{
pnodo siguiente;
char llave[MAX];
pnodo anterior;
};

/* Estructura de la lista                                           */
typedef struct lista Lista_Doble;
struct lista{
pnodo principio;
pnodo final;
};

/* Pone un Menú                                                     */
void menu(void){
printf("Listas Doblemente Enlazadas");
printf("I = Insertar un nodo");
printf("S = Suprimir un nodo");
printf("V = Ver Todos los nodos");
printf("Q = Salir");
printf("Elija una Opción : ");
}

/* Mensaje de ERROR                                                 */
void error(void){
perror("\n\t\aERROR: Memoria insuficiente...");
exit(1);
}
```

Programa 2.9: Operaciones con lista doblemente enlazadas (Parte **1**/3).

```
/* Crea un nuevo nodo del tipo de la estructura                          */
nodo *Nuevo(){
nodo *q = (nodo *)malloc(sizeof(nodo));
if(!q) error();
return(q);
}

/* Inserta un nodo                                                       */
void insertar(Lista_Doble *listaD,char llave[]){
pnodo q,pactual,panterior;
q = Nuevo();
strcpy(q -> llave,llave);
q -> anterior = q -> siguiente = NULL;
if(listaD -> principio == NULL){
      listaD -> principio = listaD -> final = q;
      return;
      }
pactual = panterior = listaD -> principio;
while(pactual != NULL && strcmp(llave,pactual -> llave) > 0){
      panterior = pactual;
      pactual = pactual -> siguiente;
      }
if(panterior == pactual){
      q -> siguiente = listaD -> principio;
      listaD -> principio = pactual -> anterior = q;
      }
else  {
      q -> anterior = panterior;
      q -> siguiente = pactual;
      panterior -> siguiente = q;
      if(pactual) pactual -> anterior = q;
      }
}
/* Saca y elimina un nodo                                                */
char *suprimir(Lista_Doble *listaD,char llave[]){
char dato[MAX];
pnodo panterior,pactual;
if(listaD -> principio == NULL) return("Lista Vacía");
panterior = pactual = listaD -> principio;
while(pactual != NULL && strcmp(llave,pactual -> llave) != 0){
      panterior = pactual;
      pactual = pactual -> siguiente;
      }
if(pactual == NULL){
      sprintf(dato,"El nodo %s No está en la lista...",llave);
      return(dato);
      }
if(panterior == pactual){
      listaD -> principio = listaD -> principio -> siguiente;
      if(listaD -> principio) listaD -> principio -> anterior = NULL;
      }
else  {
      panterior -> siguiente = pactual -> siguiente;
      if(pactual -> siguiente)     panterior -> siguiente -> anterior =
                                   pactual -> anterior;
      }
```

Programa 2.9: Operaciones con lista doblemente enlazadas (Parte **2**/3).

```
strcpy(dato,pactual -> llave);
free(pactual);
return(dato);
}

/* Imprime lista                                                    */
void ver(Lista_Doble listaD){
int i = 0;
pnodo pactual = listaD.principio;
if(pactual == NULL) printf("\n\tLista Vacía...");
while(pactual != NULL){
      printf("nodo %d = ",++i);
      printf("%s",pactual -> llave);
      pactual = pactual -> siguiente;
      }
}

/* Programa principal                                               */
void main(void){
Lista_Doble listaD;
char opcion,llave[MAX];
listaD.principio = listaD.final = NULL;
while(1){
      do    {
            clrscr();
            menu();
            opcion = tolower(getch());
            }while(     opcion != 'i' && opcion != 's' &&
                        opcion != 'v' && opcion != 'q');
      clrscr();
      switch(opcion){
            case 'i':
                  printf("Introduzca la llave a insertar : ");
                  gets(llave);
                  insertar(&listaD,llave);
                  break;
            case 's':
                  printf("Introduzca la llave a suprimir : ");
                  gets(llave);
                  strcpy(llave,suprimir(&listaD,llave));
                  printf("Suprimir : %s",llave);
                  printf("Presione alguna tecla para continuar.");
                  getch();
                  break;
            case 'v':
                  ver(listaD);
                  printf("Presione alguna tecla para continuar.");
                  getch();
                  break;
            case 'q':
                  exit(0);
                  break;
            }
      }
}
```

Programa 2.9: Operaciones con lista doblemente enlazadas (Parte **3**/3).

En la **Figura 2.25** se observa una ejecución del programa, después de insertar `AA, A Primer, B segundo` y `C tercero`.

```
        Listas Doblemente Enlazadas
I = Insertar un nodo
S = Suprimir un nodo
V = Ver Todos los nodos
Q = Salir
Elija una Opción: I
Introduzca la llave a insertar: C tercero
>V
nodo 1 = A primer
nodo 2 = AA
nodo 3 = B segundo
nodo 4 = C Tercero
Presione alguna tecla para continuar.
>S
Introduzca la llave a suprimir: AA
Suprimir: AA
Presione alguna tecla para continuar.
>V
nodo 1 = A primer
nodo 2 = B segundo
nodo 3 = C Tercero
```

Figura 2.25: Operaciones con lista doblemente enlazadas.

2.4 Tablas Hash ó de dispersión

Los **algoritmos hash** son métodos de búsqueda, que proporcionan una **longitud de búsqueda** pequeña y una flexibilidad superior a la de los otros métodos, como puede ser el método de búsqueda binaria, el cuál requiere que los elementos del arreglo se encuentren ordenados. La propiedad más importante de una buena función de dispersión, es que pueda ser calculada muy rápidamente, y que al mismo tiempo se minimicen las colisiones.

Por **longitud de búsqueda** se entiende el número de accesos que es necesario efectuar sobre un arreglo para encontrar el elemento deseado.

Las **tablas hash**, permiten como operaciones básicas: `búsqueda, inserción` y `supresión`.

Un **arreglo hash** es un arreglo producto de la aplicación de un **algoritmo hash**. Estos se emplean ampliamente en los sistemas, para acceso de datos. Gráficamente se puede representar de la forma observada en la **Figura 2.26**.

Clave	Contenido
253	Elemento 1
124	Elemento 2
***	***
021	Elemento n

Figura 2.26: Arreglo hash.

El arreglo se organiza con elementos formados por dos miembros: **clave** y **contenido**.

La **clave** constituye el medio de acceso al **contenido**. Aplicando a la **clave** una función de acceso **f**, previamente definida, obtenemos un número entero **i**, que nos da la posición del

elemento correspondiente del arreglo: **i = f(clave)**. Por ejemplo, tenemos una lista de asistencia, en donde el número de lista puede ser la **clave**, que nos da acceso al **contenido**, es decir, al **nombre**.

Conociendo la posición, tenemos acceso al contenido. El contenido puede ser la información, ó un apuntador a la dirección si la cantidad de datos es muy grande. Este acceso se conoce como **acceso directo**.

Sin embargo, el cálculo de la **clave** es un factor crítico en este tipo de búsquedas, debido a que varios contenidos pueden producir valores idénticos para la **clave**, este proceso se conoce como **colisión**. Existen varias técnicas que permiten reducir el número de colisiones que se presentan en cada conjunto de datos. La manera en que cada una de esas técnicas resuelva las colisiones afecta directamente la eficiencia de la búsqueda. A continuación se describen algunas funciones de dispersión.

Funciones de dispersión

Método de la división.- son aquellas funciones de dispersión que se generan calculando una división, por ejemplo **k mod m**.

Si los contenidos de entrada son enteros, se acepta como una buena estrategia la función **`clave = contenido % tamaño`**, es decir, el **modulo** (%) entre el **contenido** y el **tamaño** de la **tabla hash**. Sin embargo se puede presentar una mala elección del tamaño, por ejemplo si todos los contenidos acaban con cero y se elige un **tamaño de 10**, entonces la dispersión estándar es una mala elección. Se ha encontrado que una buena elección para el tamaño, es regularmente un **número primo**, para un conjunto de contenidos aleatorios, las llaves generadas con un número primo son muy uniformes.

Método de la suma.- si los contenidos son cadenas de caracteres, se pueden sumar los valores de los caracteres **ASCII**, y declarar un **índice** para cada uno de esos valores, de la forma, **`clave = tamaño_suma_cadena`**. Aunque está función es fácil de implementar, las claves no siempre se encuentran bien distribuidas, para un tamaño grande de claves.

Método de la multiplicación.- se generan valores para la dispersión en dos pasos. Primero se calcula la parte decimal del producto de **k** y cierta constante real **A**, donde **0 < A < 1**. Este resultado es entonces multiplicado por **m** antes de aplicarle la función de truncado, para obtener el valor de la dispersión. Es decir: **`clave = m (contenido * A – ⌊contenido * A⌋)`**, Donde **`(contenido * A – ⌊contenido * A⌋)`**, obtiene la parte decimal del número real. Como la parte decimal es mayor que cero, los valores de la dispersión son enteros positivos en el rango entre **0**, **1**, **2**, .., **m - 1**. Una elección para elegir **A**, con la cual se obtiene una buena dispersión es la **razón Áurea**: **$A = \phi = 1.61803399$**.

2.4.1 Tablas Hash abiertas ó de encadenamiento separado

Una de las estrategias más simples de resolución de colisiones es la **dispersión abierta ó encadenamiento separado**, consiste en tener una **lista** de todos los elementos que se dispersan en el mismo valor. Así, para buscar un valor, se encuentra su valor de dispersión y luego se recorre la lista. Para insertar, se hace lo mismo, y se inserta en la lista adecuada. En al **Figura 2.27**, se observa un **tabla hash** de este tipo, en donde se tienen **5** valores para la tabla, en donde se ha empleado **clave % tamaño,** (es decir: **clave % 5**) para asignar los valores a las listas.

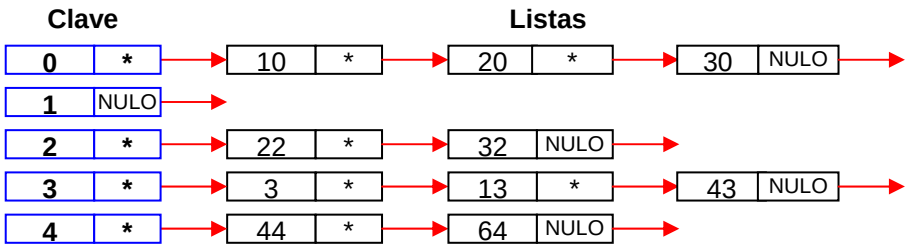


Figura 2.27: Tabla Hash abierta.

La desventaja de este tipo de **tablas hash**, es que se emplean apuntadores y estructuras, por lo que puede hacer un poco lento al algoritmo. Un ejemplo de implementación para una **tabla hash abierta**, en donde se emplea el cálculo de la función de dispersión a través del **método de la división**, se muestra en el **Programa 2.10**.

```
#include "stdio.h"
#include "stdlib.h"
#include "math.h"

/* Estructura de los nodos                                          */
typedef struct nodos nodo;
struct nodos{
    int dato;           /* Contenido                                */
    nodo *siguiente;
};

/* Mensaje de ERROR                                                 */
void error(void){
perror("\n\t\aERROR: Memoria insuficiente...");
exit(1);
}

/* Crea un nuevo nodo del tipo de la estructura                     */
nodo *Nuevo(){
nodo *q = (nodo *)malloc(sizeof(nodo));
if(!q) error();
return(q);
}

nodo **Tabla_Hash;      /* Tabla hash                               */
int t_tam;              /* Tamaño  de la tabla                      */

/* Tabla hash: Método de la División                                */
int hash1(int dato) {
return (dato % t_tam);
}

/* Inserta un nodo en la tabla                                      */
nodo *insertar(int dato) {
nodo *p, *q;
int clave;
clave = hash1(dato);
p = Nuevo();
q = Tabla_Hash[clave];
```

Programa 2.10: Tabla hash abierta con el método de la división (Parte **1**/4).

```
Tabla_Hash[clave] = p;
p -> siguiente = q;
p -> dato = dato;
return p;
}

/* Suprime un nodo de la tabla                                          */
void suprimir(int dato){
nodo *q, *p;
int clave;
q = 0;
clave = hash1(dato);
p = Tabla_Hash[clave];
/* Busca el nodo a suprimir                                             */
while (p && !(p -> dato == dato)) {
      q = p;
      p = p->siguiente;
      }
if (!p) return;
if (q) q -> siguiente = p -> siguiente;
else Tabla_Hash[clave] = p -> siguiente;
free (p);
}

/* Busca un dato en la tabla                                            */
nodo *buscar (int dato) {
nodo *p;
p = Tabla_Hash[hash1(dato)];
while (p && !(p -> dato == dato)) p = p -> siguiente;
return p;
}

/* Pone menú                                                            */
void menu(void){
printf("A = Llenar un tabla con datos aleatorios");
printf("M = Llenar un tabla de forma manual");
printf("B = Buscar un dato");
printf("S = Suprimir un Dato");
printf("V = Ver Datos de la Tabla");
printf("Q = Salir");
printf("Elija una Opción: ");
}

/* Programa principal                                                   */
void main(void) {
int i,op,*a,max,dato;
nodo *p;
t_tam = 20;
max = 0;
while(1){
      menu();
      op = tolower(getch());
      switch(op){
      case 'a':
            for (i = max - 1; i >= 0; i--) suprimir(a[i]);
            printf("\n\tIntroduce el número de datos : ");
```

Programa 2.10: **Tabla hash** abierta con el método de la división (Parte **2**/4).

```
            scanf("%d",&max);
            if((a = malloc(max * sizeof(*a))) == 0) {
                  perror ("Memoria Insuficiente...\n");
                  exit(1);
                  }
            if ((Tabla_Hash = malloc(t_tam * sizeof(nodo *))) == 0) {
                  perror("Memoria Insuficiente...\n");
                  exit(1);
                  }
            for(i = 1;i <= max;i++){
                  a[i] = random(100);
                  printf("\n\tDato %d : %d",i,a[i]);
                  insertar(a[i]);
                  }
            getch();
            break;
      case 'm':
            printf("\n\tIntroduce el número de datos : ");
            scanf("%d",&max);
            if((a = malloc(max * sizeof(*a))) == 0) {
                  perror ("Memoria Insuficiente...\n");
                  exit(1);
                  }
            if ((Tabla_Hash = malloc(t_tam * sizeof(nodo *))) == 0) {
                  perror("Memoria Insuficiente...\n");
                  exit(1);
                  }
            for(i = 1;i <= max;i++){
                  printf("\n\tDato %d : ",i);
                  scanf("%d",&a[i]);
                  insertar(a[i]);
                  }
            getch();
            break;
      case 'b':
            printf("\n\tIntroduce el Dato a Buscar : ");
            scanf("%d",&dato);
            p = buscar(dato);
            if(p) printf("\n\tDato %d si se encontró.",p -> dato);
            else printf("\a\a\n\tDato %d NO se encontró...",dato);
            getch();
            break;
      case 's':
            printf("\n\tIntroduce el Dato a Suprimir : ");
            scanf("%d",&dato);
            p = buscar(dato);
            if(p) {
                  suprimir(dato);
                  printf("\n\tDato %d se suprimir.",dato);
                  }
            else printf("\a\a\n\tDato %d NO se encontró...",dato);
            getch();
            break;
      case 'v':
            printf("\n\tDato: Hash : Dato\n\t",
                              hash1(p -> dato),p -> dato);
```

Programa 2.10: **Tabla hash abierta** con el método de la división (Parte **3**/4).

```
            for (i = max; i >= 0; i--) {
                  p = buscar(a[i]);
                  if(p) printf("\n\tDato: %02d : %d",
                              hash1(p -> dato),p -> dato);
                  }
            getch();
            break;
      case 'q':
            for (i = max-1; i >= 0; i--) suprimir(a[i]);
            exit(1);
            break;
            }
      }
}
```

Programa 2.10: **Tabla hash abierta** con el método de la división (Parte **4**/4).

Al ejecutar el **Programa 2.10** y cargar datos de forma aleatoria se obtuvieron los resultados de la **Figura 2.28**.

```
        Tabla Hash
A = Llenar un tabla con datos aleatorios
M = Llenar un tabla de forma manual
B = Buscar un dato
S = Suprimir un Dato
V = Ver Datos de la Tabla
Q = Salir
Elija una Opción: A
Introduce el número de datos: 10
Dato 1: 1
Dato 2: 0
Dato 3: 33
Dato 4: 3
Dato 5: 35
Dato 6: 21
Dato 7: 53
Dato 8: 19
Dato 9: 70
Dato 10: 94
> B
Introduce el Dato a Buscar: 52
Dato 52 NO se encontró...
> B
Introduce el Dato a Buscar: 33
Dato 33 si se encontró.
> V
Dato: Hash: Dato
Dato: 14: 94
Dato: 10: 70
Dato: 19: 19
Dato: 13: 53
Dato: 01: 21
Dato: 15: 35
Dato: 03: 3
Dato: 13: 33
```

Figura 2.28: **Tabla hash abierta** con el método de la división (Parte **1**/2).

```
Dato: 00: 0
Dato: 01: 1
> S
Introduce el Dato a Suprimir: 53
Dato 53 se suprimió.
> V
Dato: Hash: Dato
Dato: 14: 94
Dato: 10: 70
Dato: 19: 19
Dato: 01: 21
Dato: 15: 35
Dato: 03: 3
Dato: 13: 33
Dato: 00: 0
Dato: 01: 1
```

Figura 2.28: Tabla hash abierta con el método de la división (Parte **2**/2).

Si las listas se utilizan de forma no ordenada, el tiempo para realizar la `inserción` es **Θ(1)**. Los tiempos de ejecución en el peor caso son **Θ(n)**, con **n** como el número de elementos que se dispersan en la misma lista, sin embargo, si se supone una dispersión uniforme, los tiempos se reducen a **Θ(n/m)**, para las operaciones `buscar y borrar`, donde **m** es el tamaño de la tabla.

2.4.2 Tablas Hash cerradas ó de direccionamiento abierto

La **dispersión cerrada** o de **direccionamiento abierto** es una alternativa de resolución a las colisiones en las listas enlazadas. En un sistema de dispersión cerrada, si ocurre una colisión, se buscan celdas alternativas hasta encontrar una vacía; por lo que se necesita una tabla más grande para poder cargar todos los datos. Es decir, todos los elementos se almacenan en su propia tabla dispersa. Las colisiones se resuelven calculando una secuencia de lugares vacíos ó huecos de dispersión. Algunas estrategias comunes se describen a continuación.

Exploración lineal

Este es el método más sencillo de implementar, sin embargo su rendimiento tiende a decaer rápidamente con forme aumenta el factor de carga. Consiste en ensayar la siguiente posición (suponiendo una tabla circular), hasta encontrar el elemento con la llave especificada ó una localización vacía.

Para calcular los índices $\mathbf{h_i}$, podemos usar:

$$\mathbf{h_0 = H(k)}$$
$$\mathbf{h_i = (h_0 + i) \% n} \quad \textbf{con} \quad \mathbf{i = 1, 2, 3, \ldots, n}$$

Donde: **H(k)** es la **función hash**, por ejemplo: **H(k) = dato % n**
n es Tamaño de la **tabla hash**

El empleo de la **exploración lineal** conduce a un problema conocido como **agrupamiento**, en donde los elementos tienden a juntarse o agruparse en la tabla dispersa de tal forma que los elementos solo pueden ser accedidos por medio de una larga secuencia de exploración, por ejemplo, después de un gran número de colisiones.

Exploración cuadrática

La **exploración cuadrática**, es una extensión simple de la exploración lineal en la que uno de los elementos suministrados a la operación de modulo, es una función cuadrática. Se elimina el problema de agrupamiento, por que se eliminan las secuencias de exploración por simples desplazamientos cíclicos. Sin embargo, también es posible encontrar agrupamientos, llamados **secundarios**, debido a una mala elección del valor inicial. Se calcula de la forma:

$h_0 = H(k)$
$h_i = (h_0 + i^2) \% n$ con $i > 0$

Dispersión doble

Consiste en el cálculo de la función de dispersión a partir de dos funciones existentes. De esta forma se pueden eliminar los agrupamientos primarios y secundarios. Una elección común es la siguiente:

$H(i) = i * H_1(i)$

Donde: **H(i)** es la **función hash** que se va a calcular
$H_1(i)$ es la primera **función hash** que puede ser: $H_1(i) = i \% n$

Dispersión coalescente (Coalesced Hashing)

La **dispersión coalescente**, es similar a la dispersión abierta, solo que todos los elementos se almacenan en la propia tabla dispersa. Esto se consigue permitiendo que cada hueco de la tabla dispersa almacene, no solo un elemento, sino, también un apuntador. Estos apuntadores pueden almacenar un valor nulo ó bien, la dirección de alguna otra componente dentro de la tabla dispersa.

2.4.3 Tablas Hash de redispersión

Si la tabla de dispersión se llena demasiado, el tiempo de ejecución de las operaciones empezará a prolongarse, por lo que una solución es construir otra tabla de casi el doble de tamaño y recorrer toda la tabla original calculando el nuevo valor de la dispersión de cada elemento e insertándolo en la tabla nueva.

En algún momento la tabla se llena, y se presenta un **desborde de la tabla**. Esto no es un problema en las tablas de dispersión abiertas, debido a que tienen una asignación dinámica de memoria, por lo que se puede asignar toda la memoria disponible. Así, los desbordamientos de una **tabla hash** se presentan en las tablas de dispersión cerradas.

Existen básicamente dos técnicas que evitan el problema de desbordamiento de la tabla asignando memoria adicional; en cualquier caso, no se puede esperar a asignar más memoria una vez que se ha agotado, se debe de asignar cuando se llegue a un **factor de carga** α, que supere a cierto umbral α_1.

Expansión de la Tabla

Consiste en asignar una tabla más grande cuando una inserción produzca que el factor de carga exceda α_1, y trasladar el contenido de la tabla antigua a la nueva. La memoria empleada por la tabla antigua puede ser entonces liberada. Debido a que los datos en la tabla tienen que ser redispersados, los costos adicionales hacen que este método sea demasiado lento.

Dispersión extensible

Divide a la tabla en bloques, por lo que se eliminan los costos de la redispersión. Se realizan dos pasos: primero se comprueban los bits de menor orden de una clave para determinar en que bloque será almacenado un elemento, entonces el elemento se dispersa en un hueco particular de ese bloque utilizando alguno de los métodos descritos anteriormente. Las direcciones de estos bloques se almacenan en una tabla de directorio, en donde se almacenan los números de bit de menor orden utilizados durante el primer paso de la dispersión.

Unidad 3: Recursividad y Estructuras de Datos No Lineales

3.1 Recursión

La **recursión**, es la propiedad de una función de llamarse así misma **[4, 9, 10** y **18]**. En matemáticas, la recursión se presenta cuando una función está definida en términos de sí misma. La idea de hacer una función recursiva, es que la función pueda ser expresada en unas pocas líneas. La **recursión** es un método general de resolver los problemas reduciéndolos a problemas más simples de un tipo similar (estrategia **Divide y Vencerás**). Por ejemplo tenemos la función:

$$f(x) = \begin{cases} 0 & x = 0 \\ 2 * f(x-1) + x^2 & x > 0 \end{cases}$$

La función anterior puede ser evaluada para algunos valores de **x**, por ejemplo: `f(0) = 0`, `f(1) = 1`, `f(2) = 6`, `f(3) = 21`, etc. Este tipo de funciones manejan, lo que se conoce como **caso base**, es decir, un valor para el cual el valor de la función es conocido, sin necesidad de recursión.

Cuando una función en **lenguaje C** llama a otra función, tiene que comunicar el valor actual ó la posición de todas las variables y parámetros a la nueva función. Para realizar esto, hay que colocar estos valores en una **pila**. Si otra función necesita resultados de una tercera, es necesario guardar otra vez las variables en la **pila**. Sin embargo, este proceso no requiere que una función conozca lo que se hace en la otra, es decir, este sistema es el que hace posible la recursión. Al escribir una función recursiva, se deben de tomar en cuenta las **cuatro reglas** siguientes:

- **Caso base**.- siempre se debe de tener uno o más casos base, los cuales se puedan resolver sin recursión.
- **Progreso**.- para los casos que se resuelvan recursivamente, la llamada recursiva siempre debe de tener un **caso base**.
- **Regla de diseño**.- se supone que todas las llamadas recursivas funcionan.
- **Regla de interés compuesto**.- el trabajo nunca se debe de duplicar resolviendo el mismo problema en llamadas recursivas separadas.

Los **problemas** con la recursión residen en los costos de su funcionamiento. Aunque estos costos casi siempre son justificables por que los programas, no solo por que son más pequeños, sino también por que son más claros, la *recursión nunca debe de usarse para sustituir un solo ciclo*. Muchas estructuras de datos usan recursión, como ejemplo tenemos: ordenación por fusión, ordenación rápida ó QuickSort, árboles binarios, fractales, etc.

3.1.1 Ejemplos de funciones recursivas

Suma de enteros no negativos.- este ejemplo solo es ilustrativo, pues todos los lenguajes de programación tienen esta operación, implementada de forma más eficiente.

$$suma(a,b) = \begin{cases} a & b = 0 \\ suma(a,b-1) + 1 & b > 0 \end{cases}$$

```
int suma(int a, int b){
if(b == 0) return(a);
else return(suma(a,b - 1) + 1);
}
```

Para **a = 2** y **b = 3**: La primera llamada **`suma(2,3)`** no es recursiva:

Recursión	a,b	Llamada	Retorno
1	2,3	suma(2,2) + 1	**2 + 1 +1** +1
2	2,2	suma(2,1) + 1	**2 + 1** + 1
3	2,1	suma(2,0) + 1	**2** + 1

Se observa que la cantidad de llamadas recursivas es **b**.

Multiplicación.- este ejemplo solo es ilustrativo, pues todos los lenguajes de programación tienen esta operación, implementada de forma más eficiente.

$$mult(a,b) = \begin{cases} 0 & b = 0 \\ mult(a,b-1) + a & b > 0 \end{cases}$$

```
/* Recursiva */
int mult(int a,int b){
if(b == 0) return(0);
else return(mult(a,b - 1) + a);
}
```

```
/* Iterativa */
int mult(int a,int b){
int c = 0,i;
for(i = b; i > 0;i--) c = c + a;
return(c);
}
```

Para **a = 2** y **b = 3**: La primera llamada **`mult(2,3)`** no es recursiva:

Recursión	a,b	Llamada	Retorno
1	2,3	mult(2,2) + 2	**0 + 2 + 2** + 2
2	2,2	mult(2,1) + 2	**0 + 2** + 2
3	2,1	mult(2,0) + 2	**0** + 2

Iteración	c
1	2
2	4
3	6

Se observa que la cantidad de **llamadas recursivas** es **b**. El número de **iteraciones** es **b**. Sin embargo la multiplicación es factible en un solo paso.

Factorial.- la función factorial es un ejemplo claro del uso de la recursión, sin embargo se considera que es más eficiente el empleo de técnicas iterativas. Para calcular el factorial de un número **n** es necesario realizar el producto sucesivo de **1*2*3*…*n**, por definición para **0! = 1**.

$$fact(n) = \begin{cases} 1 & n = 0 \\ fact(n-1) * n & n > 0 \end{cases}$$

```
/* Factorial Recursiva */
double factorial(double n){
if(n == 0) return(1);
else return(n * factorial(n - 1));
}
```

```
/* Factorial Iterativa */
double factorial(double n){
int i;
double fact = 1;
for(i = 1;i <= n;i++)
      fact = fact * i;
return(fact);
}
```

Para **n = 4**: La primera llamada **`factorial(4)`** no es recursiva:

Recursión	n	Llamada	Retorno
1	4	4* factorial(3)	**1*1*2*3***4
2	3	3* factorial(2)	**1*1*2***3
3	2	2* factorial(1)	**1*1***2
4	1	1* factorial(0)	**1***1

Iteración	factorial
1	1
2	1*2
3	1*2*3
4	1*2*3*4

Se observa que la cantidad de **llamadas recursivas** es **n**. El número de **iteraciones** es **n**.

Números de Fibonacci.- este un problema que puede ser resuelto fácilmente usando la recursión. La secuencia de números **Fibonacci** es de la forma: **0, 1, 1, 2, 3, 5, 8, 13, 21,…** La fórmula general es:

$$fib(n) = \begin{cases} 0 & n \leq 1 \\ 1 & n = 2 \\ fib(n-1) + fib(n-2) & n > 2 \end{cases}$$

En este caso se tienen dos casos base, uno para `fib(1) = 0` y otra para `fib(2) = 1`, de otra forma, la llamada recursiva se encontraría indefinida a partir de `n = 3`.

```
/* Fibonacci Recursiva */
int fib(int n){
if(n <= 1) return(0);
if(n == 2) return(1);
return(fib(n-1) + fib(n-2));
}
```

```
/* Fibonacci Iterativa */
int Fib(int n){
int i,a = 0,b = 1,c;
if(n <= 1) return(a);
if(n == 2) return(b);
for(i = 2;i < n;i++) {
    c = a + b; a = b; b = c;
    }
return(c);
}
```

Para **n = 5**:

Recursión	Llamada
1	**fib(4) + fib(3)**
2	**fib(3) + fib(2) + fib(2) + fib(1)**
3	**fib(2) + fib(1)** + 1 + 1 + 0

Iteración	c
1	1
2	2
3	3

El numero de funciones recursivas crece **exponencialmente**, por lo que el uso de la recursión en este caso es inadecuado. El número de llamadas recursivas para algunos valores de **n**, se muestra en la **Tabla 3.1** y en la **Figura 3.1**.

n	Fib(n)	Llamadas
5	3	8
6	5	14
7	8	24
8	13	40
9	21	66
10	34	108
11	55	176
12	89	286
13	144	464
14	233	752
15	377	1,218

n	Fib(n)	Llamadas
16	610	1,972
17	987	3,192
18	1,597	5,166
19	2,584	8,360
20	4,181	13,528
21	6,765	21,890
22	10,946	35,420
23	17,711	57,312
24	28,657	92,734
25	46,368	150,048

Tabla 3.1: Secuencia **Fibonacci** y numero de llamadas recursivas.

El número de llamadas recursivas para `Fibonacci(n)` es:

Llamadas = (2 / √5) * (∅ n + (1 / ∅) n) – 2

Donde: **∅ = (1 + √5) / 2**

El número de llamadas iterativas es **n - 2**.

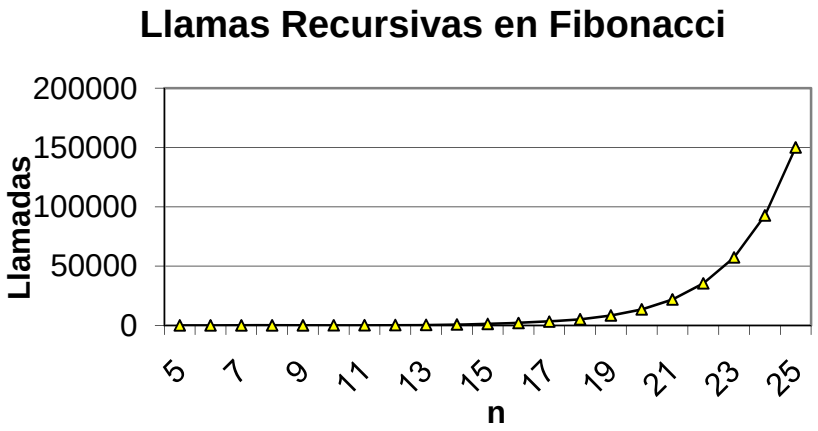


Figura 3.1: Número de llamadas para la secuencia **Fibonacci**.

Existe una función más eficiente, incluso que la iteración (Formula analítica). Está es una fórmula que permite obtener la secuencia de **Fibonacci** para n $\geq$ 5:

$$fib(n) = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^{n-1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n-1}\right)$$

3.1.2 Torres de Hanói

Hasta ahora, los ejemplos vistos, es posible resolverlos de manera más eficiente de forma iterativa y hemos usado la recursión para ejemplificar su uso. Sin embargo; hay problemas, los cuales sería muy difícil resolverlos por métodos iterativos, el ejemplo más famoso de estos problemas, es el de las **Torres de Hanói**, el cual es descrito a continuación:

En el gran templo de Benares, bajo la cúpula que marca el centro del mundo, descansa una placa de bronce en la que se encuentran fijadas tres agujas de diamante, altas y tan delgadas como el cuerpo de una abeja. En una de estas agujas Dios colocó, en la creación sesenta y cuatro discos de oro puro, el mayor descansando sobre la placa de bronce y los demás en tamaño decreciente hacia arriba. Esta es la torre de Brama. Día y noche sin parar, los sacerdotes transfieren los discos de una aguja de diamante a otra de acuerdo a las leyes fijas e inmutables de Brama, que indican que el sacerdote de servicio no debe mover más de un disco cada vez y que puede colocarlo en una aguja en la que no haya un disco de tamaño menor. Cuando se hayan transferido los sesenta y cuatro discos de la aguja en la que Dios los colocó en la Creación a otra de las agujas, torre, templo y Brahamanes se desmoronarán y se convertirán en polvo y, con un trueno, el mundo se desvanecerá.

Es decir, para **n** como el **número de discos** y tres columnas **A, B** y **C**, el programa debe de seguir estas reglas:

1. Si **n = 1**, mover el disco único de **A** a **B** y parar.
2. Mover el disco superior de **A** a **C**, **n-1** veces usando **B** como auxiliar.
3. Mover el disco restante de **A** a **C**.
4. Mover los discos **n-1** de **C** a **B** usando **A** como auxiliar.

El programa que realiza estas operaciones es el siguiente:

```
/* Torres de Hanoi                                                    */
void hanoi(int n,char p1,char p2,char p3){
if(n == 1) {
      printf("\tMover disco %d del poste %c al poste %c\n",1,p1,p2);
      return;
      }
hanoi(n - 1,p1,p3,p2);
printf("\tMover disco %d del poste %c al poste %c\n",n,p1,p2);
hanoi(n - 1,p3,p2,p1);
}

/* Programa principal                                                 */
void main(){
int n;
printf("\n\t\tProblema de las Torres de Hanoi\n");
n = 3;
hanoi(n,'A','B','C');
}
```

Programa 3.1: Problema de las **Torres de Hanói**.

```
Problema de las Torres de Hanoi para 3 discos
Mover disco 1 del poste A al poste B
Mover disco 2 del poste A al poste C
Mover disco 1 del poste B al poste C
Mover disco 3 del poste A al poste B
Mover disco 1 del poste C al poste A
Mover disco 2 del poste C al poste B
Mover disco 1 del poste A al poste B
```

Figura 3.2: Resultado del programa para las **Torres de Hanói** para **3** discos.

En la **Figura 3.3** se muestran los movimientos que se tienen que realizar para mover los tres discos, de acuerdo a las reglas anteriores.

Como se observa, el número de movimientos para **n** discos, es $\mathbf{2^n - 1}$. La **Tabla 3.2**, muestra el tiempo de necesario para mover los discos, para algunos valores de **n.** (**mps** = **m**ovimientos **p**or **s**egundo).

n	1,000 mps	1,000,000 mps
8	255 ms	255 μs
16	65.5 s	66 ms
32	49.71 días	71.58 min
64	584,942 milenios	585 milenios

Tabla 32: Tiempo para resolver el problema de las **Torres de Hanoi**.

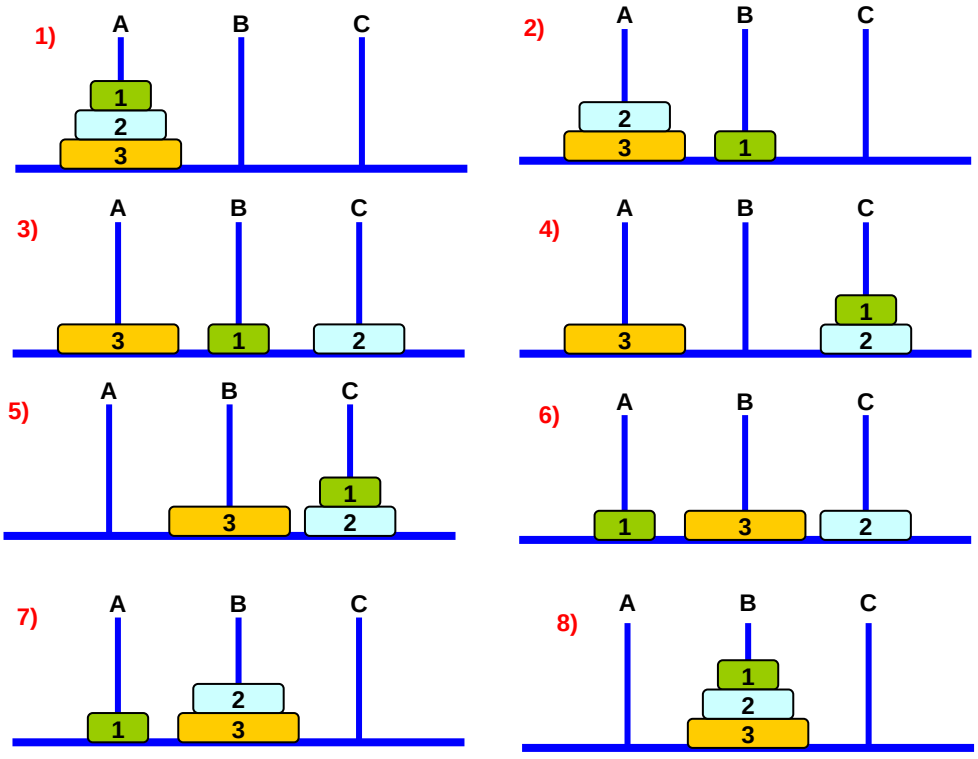


Figura 3.3: Torres de Hanoi para **3** discos.

3.1.3 Función de Ackerman

La función de **Ackerman**, es una función recursiva, la cual está definida para valores enteros no negativos, **n** y **m**, de la forma:

A(0,n) = n + 1	**para m = 0 y n ≥ 0**
A(m,0) = A(m - 1,1)	**para m ≥ 0 y n = 0**
A(m,n) = A(m - 1, A(m,n - 1))	**para m > 0 y n ≥ 0**

El código para resolver la función de **Ackerman** es el siguiente:

```
float ack(int m,int n){
if(m == 0) return(n + 1);
else  if (n == 0) return(ack(m - 1,1));
else return(ack(m - 1, ack(m,n - 1)));
}
```

En la **Tabla 3.3** se muestran algunos valores y el número de llamadas recursivas para algunas evaluaciones de la función de **Ackerman**.

Función	Valor	Llamadas recursivas
A(1,1)	3	4
A(1,2)	4	6
A(1,3)	5	8
A(1,4)	6	10
A(2,1)	5	14
A(3,1)	13	106
A(4,1)	65533	*
A(2,2)	7	27
A(2,3)	9	44
A(3,3)	61	2,432
A(3,4)	125	10,307

Tabla 3.3: Valores y numero de llamadas para la función de **Ackerman**.

Esta función también crece de forma extremadamente rápida, por lo que para algunos valores la cantidad de recursos disponibles en una computadora, no es suficiente para su cálculo.

3.2 Árboles Binarios

Un **Árbol Binario** es un árbol ordenado en el cual cada nodo del árbol tiene uno, dos ó ningún hijo (Por ejemplo, un árbol genealógico) **[1**, **3**, **5**, **7**, **8**, **9**, **10**, **11**, **17**, **18** y **19]**. Es decir, tiene tres conjuntos desarticulados, los cuales se describen como:

1) Un solo elemento llamado **raíz** (**Figura 3.4: A**).
2) Un **subárbol izquierdo** (**Figura 3.4: B**, **D** y **E**).
3) Un **subárbol derecho** (**Figura 3.4: C** y **F**).

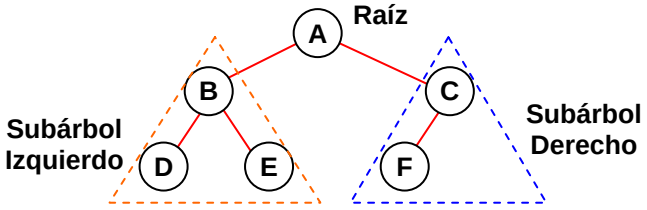


Figura 3.4: Árbol Binario.

- **Nodo**.- es cada elemento del árbol (**A**, **B**, **C**, **D**, **E** y **F**). Al contenido se le llama dato, elemento, clave, llave, key, registro, etc.
- **Subárboles vacíos**.- son los nodos sin hijos, de la **Figura 3.4**, se consideran a **D**, **E** y **F**, como árboles vacíos.
- **Hoja**.- es un nodo sin hijos (**D**, **E** y **F**). Cuando se hace distinción de nodos hojas y nodos no hojas, se dice que son **nodos internos** a los que no son hojas y **nodos externos** a los nodos hojas.
- **Padre ó ancestro**.- un nodo con descendientes (**A**, **B** y **C**).
- **Hijo ó descendiente**.- un nodo que tiene un ancestro, un hijo puede ser **izquierdo** o **derecho** (**B**, **C**, **D**, **E** y **F**).
- **Hermano**.- si dos nodos son descendientes, izquierdo y derecho del mismo padre (**B** y **C**) y (**D** y **E**).
- **Nivel**.- la raíz tiene un **nivel 0**, para otros nodos, el nivel es el del padre más uno. En la **Figura 3.5** se observan los niveles de cada árbol.
- **Profundidad ó altura**.- es el máximo nivel de cualquier hoja del árbol (**Figura 3.5**, profundidad = **3**).
- Un **árbol binario heterogéneo** es aquél en el que en la estructura del nodo se tienen diferentes tipos de datos.

- Un **Árbol estrictamente binario** es un árbol binario, que para **h** hojas, tiene **n = 2h - 1** nodos. Es decir, un nodo que no es una hoja tiene subárboles izquierdo y derecho no vacíos (**Figura 3.5**).

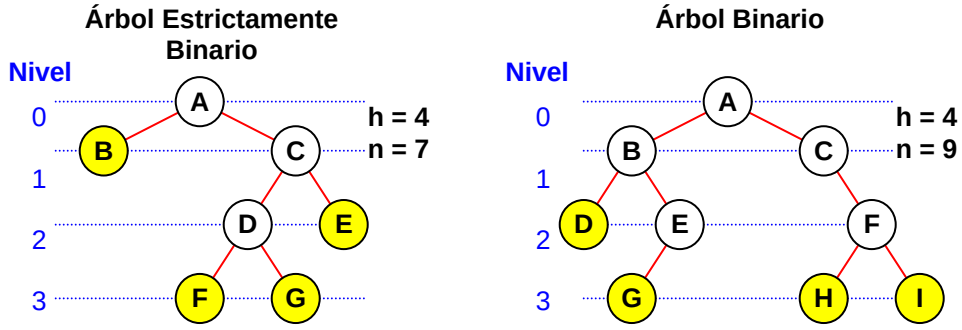


Figura 3.5: Árbol Estrictamente Binario y Árbol Binario.

- **Árbol Binario Completo**.- es un árbol de profundidad **d**, que es **estrictamente binario** y cuyas hojas están en el nivel **d** (**Figura 3.6**).

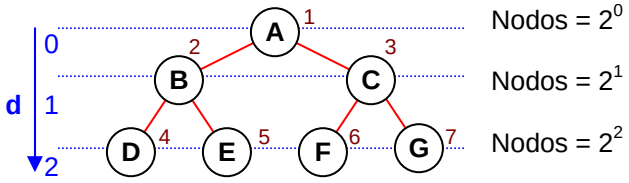


Figura 3.6: Árbol Binario Completo.

Es posible calcular el número de nodos del árbol binario completo, de acuerdo a la **Ec. 1**, la cual puede obtenerse por inducción.

$$\text{Número de nodos} = n = \sum_{j=0}^{d} 2^j = 2^{d+1} - 1 \qquad \text{... (Ec. 1)}$$

Por lo tanto podemos conocer el número de hojas y la profundidad (**Ecs. 2** y **3**).

$$\text{Numero de hojas} = h = 2^d \qquad \text{... (Ec. 2)}$$

$$\text{Profundidad} = d = \log_2 (n + 1) - 1 \qquad \text{... (Ec. 3)}$$

Enumeración.- a la raíz se le asigna el número **1**, al hijo izquierdo el numero **2**, al hijo derecho el numero **3**, etc. En la **Figura 3.6** se observa la enumeración de los nodos.

Árbol Binario de Búsqueda.- es un árbol ordenado. Las ramas de cada nodo están ordenadas de acuerdo a las siguientes reglas:

- Para todo nodo $\mathbf{a_i}$, todos los datos del subárbol izquierdo de $\mathbf{a_i}$ son menores que el dato de $\mathbf{a_i}$.
- Todos los datos del subárbol derecho de $\mathbf{a_i}$ son mayores ó iguales que el dato de $\mathbf{a_i}$.

3.2.1 Recorridos en árboles binarios

Los árboles binarios son empleados para imponer una jerarquía ó un orden a un conjunto de datos. Por lo que la forma en que se recorre un árbol produce diferentes resultados. Existen algunas formas comunes de recorrer un árbol: en **preorden** ó **depth first order**, en **orden**, en **postorden** y por **nivel** ó **breadth first order** (se recorre de acuerdo a su enumeración). A manera de ejemplo consideremos el **árbol binario de expresiones** de la **Figura 3.7**.

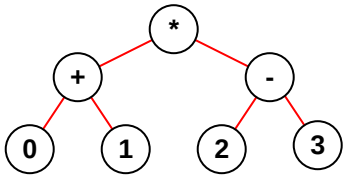


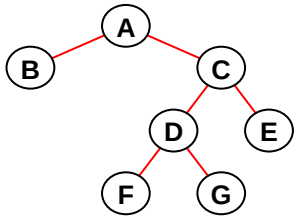
Figura 3.7: Árbol Binario de expresiones.

- **Recorrido en preorden (Orden de primero en profundidad ó depth first order)**.- cuando los nodos se visitan en el orden: **raíz, hijo izquierdo e hijo derecho**. Para el árbol de la **Figura 3.7** se obtiene el resultado: *** + 0 1 - 2 3**. Está expresión en matemáticas se conoce como **prefija** y representa a la operación **(0 + 1) * (2 - 3)**.
- **Recorrido en orden (orden simétrico)**.- cuando los nodos se visitan de la forma **hijo izquierdo, raíz e hijo derecho**. Para el árbol de la **Figura 3.7** se obtiene el resultado: **0 + 1 * 2 - 3**. Está expresión en matemáticas se conoce como **infija** (convencional), y es la expresión algebraica sin paréntesis de **(0 + 1) * (2 - 3)**.
- **Recorrido en postorden**.- cuando los nodos se visitan en el orden: **hijo izquierdo, hijo derecho y raíz**. Para el árbol de la **Figura 3.7** se obtiene el resultado: **0 1 + 2 3 - ***. Está expresión en matemáticas se conoce como **postfija** y representa a la misma operación.
- **Recorrido por nivel ó breadth first order**.- cuando los nodos se recorren por nivel, es decir, de la forma en que son enumerados. Para el árbol de la **Figura 3.7** se obtiene el resultado: *** + - 0 1 2 3**.

Recorrer un árbol binario de cualquier forma toma un tiempo **Θ(n)**, con **n** como el número de nodos. Una vez que se ha llamado al proceso recursivo cada nodo es llamado dos veces en el árbol, por lo que en realidad se llama a la función **2n** veces.

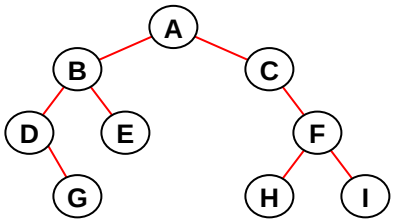
Ejemplos: Algunos ejemplos de estos recorridos se muestran a continuación:

a)



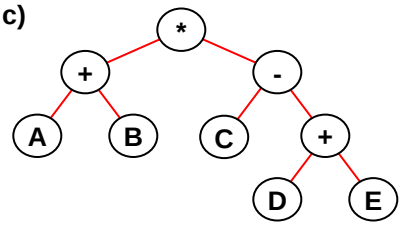
Preorden:	**ABCDFGE**
En orden:	**BAFDGCE**
Postorden:	**BFGDECA**
Depth first order:	**ABCDFGE**
Breadth first order:	**ABCDEFG**

b)



Preorden:	**ABDGECFHI**
En orden:	**DGBEACHFI**
Postorden:	**GDEBHIFCA**
Depth first order:	**ABDGECFHI**
Breadth first order:	**ABCDEFGHI**

c)



Preorden:	***+AB-C+DE**
En orden:	**A+B*C-D+E**
Postorden:	**AB+CDE+-***
Depth first order:	***+AB-C+DE**
Breadth first order:	***+-ABC+DE**

3.2.2 Operaciones con árboles Binarios de Búsqueda

Las **operaciones** comunes una vez que un árbol binario de búsqueda se encuentra construido son: `Búsqueda`, `Mínimo` y `Máximo`, cada una de estás puede tomar un tiempo **O(d),** donde **d** es la **profundidad del árbol**. También las operaciones para la construcción del árbol `Inserción` y `Supresión` toman un tiempo **O(d)**. Otras funciones como `contar nodos` y `recorrer los nodos` toman un tiempo **O(n)**, con **n** como el **número de nodos** en el árbol. A continuación se describe cada una de estás operaciones básicas.

- **Búsqueda**.- busca un **dato** dentro de un nodo, en todos los nodos del árbol.
- **Mínimo**.- busca el elemento **mínimo** de un árbol. El elemento **mínimo** se encuentra en el nodo que está **más a la izquierda** del árbol.
- **Máximo**.- busca el elemento **máximo** de un árbol. El elemento **máximo** se encuentra en el nodo que está **más a la derecha** del árbol.
- **Numero de elementos**.- cuenta el número de nodos del árbol.
- **Inserción**.- en la **Figura 3.8** se inserta un nuevo elemento a un árbol. Se crea un nodo, luego se determina el punto de inserción (se busca el dato en el árbol, hasta que se encuentra un nodo nulo ó una hoja) y luego se le asigna una dirección.

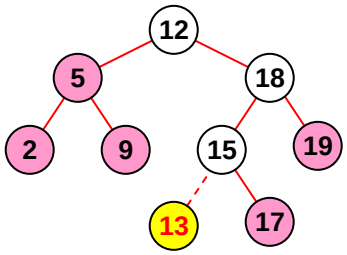


Figura 3.8: Inserción del nodo con clave **13** en un árbol binario.

- **Supresión**.- en la **Figura 3.9** se muestran las formas de borrar el nodo **z** de un árbol **T**. Cuando se borra un nodo con un solo nodo terminal o un único descendiente, es una tarea fácil. Pero cuando un nodo tiene dos hijos, el nodo a borrar debe de ser reemplazado, bien por el nodo más a la derecha en el subárbol izquierdo de dicho nodo ó bien por el nodo más a la izquierda en el subárbol derecho.

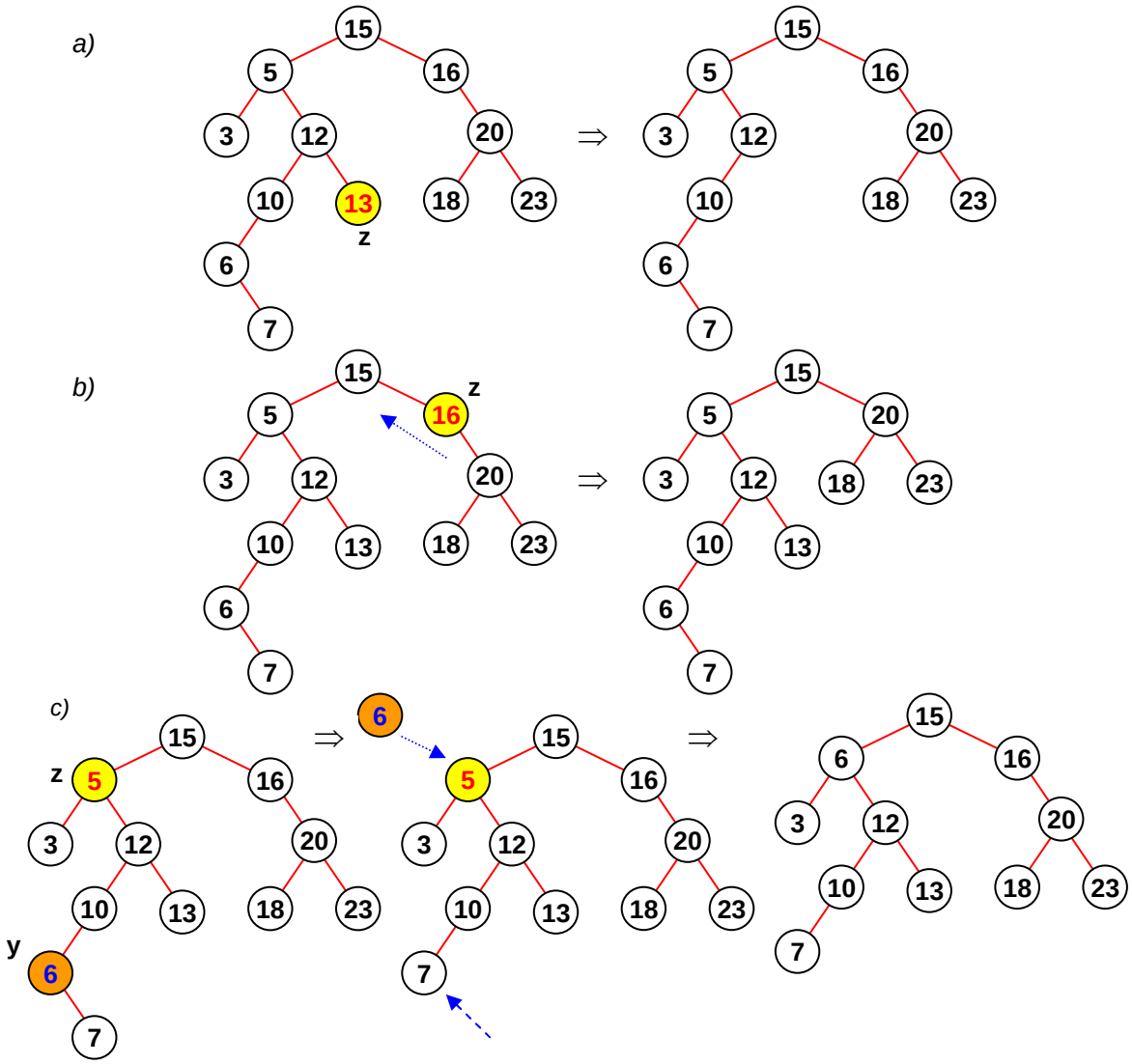


Figura 3.9: Borrado del **nodo z**: *a)* si **z** no tiene hijos, *b)* si **z** tiene un solo hijo y *c)* si **z** tiene 2 hijos. La implementación de las **operaciones básicas**, se muestra en el **Programa 3.2**.

```
#include "stdio.h"
#include "stdlib.h"
#include "conio.h"
#include "malloc.h"

/* Estructura de un nodo del árbol                                    */
typedef struct nodo arbol;
struct nodo{
int dato;
arbol *izquierdo;
arbol *derecho;
};
```

Programa 3.2: Operaciones básicas con árboles binarios (Parte **1**/5).

```
/* Mensaje de ERROR                                                    */
void error(void){
perror("\n\t\aERROR: Memoria insuficiente...");
exit(1);
}

/* Crea un nuevo nodo del tipo de la estructura                        */
arbol *Nuevo(){
arbol *q = (arbol *)malloc(sizeof(arbol));
if(!q) error();
return(q);
}

/* Cambia el dato de p por el dato de s                                */
arbol *cambia(arbol *p,arbol *s){
arbol *temp;
if(s -> derecho != NULL) s -> derecho = cambia(p,s -> derecho);
else  {
      temp = s;
      p -> dato = s -> dato;
      s = s -> izquierdo;
      free(temp);
      }
return(s);
}

/* Cuenta el número de nodos en el árbol                               */
int num_nodos(arbol *p){
if(p == NULL) return(0);
return(num_nodos(p -> izquierdo) + 1 + num_nodos(p -> derecho));
}

/* Busca y suprime un dato del árbol                                   */
arbol *suprimir(arbol *p,int dato){
arbol *temp;
if(p == NULL) printf("\n\tÁrbol Vacío...");
else if(dato < p -> dato)
      p -> izquierdo = suprimir(p -> izquierdo,dato);
else if(dato > p -> dato)
      p -> derecho = suprimir(p -> derecho,dato);
else  {
      temp = p;
      if(temp -> derecho == NULL) {
            p = temp -> izquierdo;
            free(temp);
            }
      else if(temp -> izquierdo == NULL){
            p = temp -> derecho;
            free(temp);
            }
      else  temp -> izquierdo = cambia(temp,temp -> izquierdo);
      }
return(p);
}
```

Programa 3.2: Operaciones básicas con árboles binarios (Parte **2**/5).

```
/* Busca un dato en el árbol                                          */
int buscar(arbol *p,int dato){
if(p == NULL) return(NULL);
if(dato == p -> dato) return(1);
if(dato < p -> dato) return(buscar(p -> izquierdo,dato));
else return(buscar(p -> derecho,dato));
}

/* Busca el dato mínimo en el árbol                                   */
int buscar_min(arbol *p){
if(p == NULL) return(NULL);
else if(p -> izquierdo == NULL) return(p -> dato);
else return(buscar_min(p -> izquierdo));
}

/* Busca el dato máximo en el árbol                                   */
int buscar_max(arbol *p){
if(p == NULL) return(NULL);
else if(p -> derecho == NULL) return(p -> dato);
else return(buscar_max(p -> derecho));
}

/* Recorre y muestra todos los nodos; Breadth first Order             */
void ver(int nivel,arbol *p){
int i;
if(p != NULL) {
      ver(nivel + 1,p -> derecho);
      printf("\n");
      for(i = 0;i < nivel;i++) printf("   ");
      printf("%d",p -> dato);
      ver(nivel + 1,p -> izquierdo);
      }
}

/* Recorre el árbol en Pre-Orden; Depth first Order                   */
void pre_orden(arbol *p){
if(p != NULL){
      printf("%d ",p -> dato);
      pre_orden(p -> izquierdo);
      pre_orden(p -> derecho);
      }
}

/* Recorre el árbol en En-Orden                                       */
void en_orden(arbol *p){
if(p != NULL) {
      en_orden(p -> izquierdo);
      printf("%d ",p -> dato);
      en_orden(p -> derecho);
      }
}
```

Programa 3.2: Operaciones básicas con árboles binarios (Parte **3**/5).

```
/* Recorre el árbol en Post-Orden                                    */
void post_orden(arbol *p){
if(p != NULL){
      post_orden(p -> izquierdo);
      post_orden(p -> derecho);
      printf("%d ",p -> dato);
      }
}

/* Insertar un nuevo nodo en el árbol                                */
arbol *insertar(int dato,arbol *p){
if(p == NULL) {
      p = Nuevo();
      p -> dato = dato;
      p -> izquierdo = NULL;
      p -> derecho = NULL;
      return(p);
      }
if(dato < p -> dato) p -> izquierdo = insertar(dato,p -> izquierdo);
else p -> derecho = insertar(dato,p -> derecho);
return(p);
}

/* Pone menú                                                         */
void menu(void){
printf("Operaciones Básicas con un árbol Binario");
printf("A = Llenar un árbol de forma aleatoria");
printf("C = Llenar un árbol de forma manual");
printf("B = Buscar un dato");
printf("S = Suprimir un Dato");
printf("M = Buscar Mínimo y Máximo");
printf("N = Contar Nodos");
printf("R = Ver Recorridos");
printf("V = Ver árbol");
printf("Q = Salir");
printf("Elija una Opción: ");
}

/* Programa principal                                                */
void main(void){
int n,i,dato;
char op;
arbol *p = NULL;        /* Árbol Vacío                               */
while(1){
      menu();
      op = tolower(getch());
      switch(op){
            case 'a': printf("Número de nodos del árbol: ");
                  scanf("%d",&n);
                  n = abs(n);       /* Solo n positivo               */
                  randomize();
                  for(i = 0;i < n;i++) {
                        printf("\n\tElemento No. %d : ",i+1);
                        dato = random(100);
                        printf("%d",dato);
                        p = insertar(dato,p);
                        }
```

Programa 3.2: Operaciones básicas con árboles binarios (Parte **4**/5).

```
                  break;
            case 'c': printf("Número de nodos del árbol: ");
                  scanf("%d",&n);
                  n = abs(n);
                  for(i = 0;i < n;i++) {
                        printf("\n\tElemento No. %d: ",i+1);
                        scanf("%d",&dato);
                        p = insertar(dato,p);
                        }
                  break;
            case 'v':printf("Árbol Binario");
                  ver(0,p);
                  break;
            case 'm':
                  printf("\n\n\tMínimo = %d",buscar_min(p));
                  printf("\n\n\tMáximo = %d",buscar_max(p));
                  break;
            case 'n': printf("\n\n\tNodos: %d",num_nodos(p));
                  break;
            case 'b':
                  printf("\n\tDato a Buscar : ");
                  scanf("%d",&dato);
                  printf("\n\t");
                  if(buscar(p,dato) != NULL)
                        printf("\n\tSe encontró el dato %d",dato);
                  else  printf("\n\t\aNo se encontró el dato %d",dato);
                  break;
            case 'r': printf("Recorrido en Pre-Orden:");
                  pre_orden(p);
                  printf("Recorrido En Orden:");
                  en_orden(p);
                  printf("Recorrido en Post-Orden:");
                  post_orden(p);
                  break;
            case 's':printf("Dato a Suprimir: ");
                  scanf("%d",&dato);
                  suprimir(p,dato);
                  break;
            case 'q':
                  exit(1);
                  break;
            }
      getch();
      }
}
```

Programa 3.2: Operaciones básicas con árboles binarios (Parte **5**/5).

En la **Figura 3.10**, se observa un ejemplo del **Programa 3.2** para el árbol de la **Figura 3.11**. Se observa el árbol binario generado con números aleatorios y como queda después de suprimir el nodo con el dato **38**.

```
          Operaciones Básicas con un Árbol Binario
A = Llenar un árbol de forma aleatoria
C = Llenar un árbol de forma manual
B = Buscar un dato
S = Suprimir un Dato
M = Buscar Mínimo y Máximo
N = Contar Nodos
R = Ver Recorridos
V = Ver Árbol
Q = Salir
Elija una Opción: A
Número de nodos del árbol: 10
Elemento No. 1: 39
Elemento No. 2: 88
Elemento No. 3: 19
Elemento No. 4: 38
Elemento No. 5: 17
Elemento No. 6: 3
Elemento No. 7: 40
Elemento No. 8: 67
Elemento No. 9: 37
Elemento No. 10: 51
>V
        Árbol Binario
   88
          67
             51
      40
39
      38
         37
   19
      17
         3
>R
Recorrido en Pre-Orden:
39 19 17 3 38 37 88 40 67 51
Recorrido En Orden:
3 17 19 37 38 39 40 51 67 88
Recorrido en Post-Orden:
3 17 37 38 19 51 67 40 88 39
>B
Dato a Buscar: 19
Se encontró el dato 19
>S
Dato a Suprimir: 38
>R
Recorrido en Pre-Orden:
39 19 17 3 37 88 40 67 51
Recorrido En Orden:
3 17 19 37 39 40 51 67 88
Recorrido en Post-Orden:
3 17 37 19 51 67 40 88 39
>V
        Árbol Binario
   88
          67
             51
      40
39
      37
   19
      17
         3
```

Figura 3.10: Operaciones Básicas de un Árbol Binario.

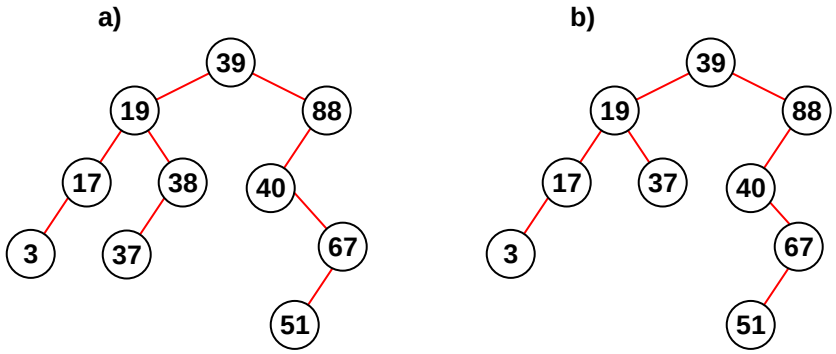


Figura 3.11: a) Árbol Binario Generado y **b)** Árbol Binario después de suprimir el dato 38.

En el **Programa 3.2** se presenta también, como los árboles binarios pueden ser empleados como **árboles binarios de ordenamiento**. Se observa que al realizar un **recorrido en orden** del árbol binario, los datos quedan ordenados de forma **ascendente**.

Un árbol binario de este tipo tiene la propiedad de que todos los elementos del subárbol izquierdo de un nodo **n** son menores que el contenido en **n** y todos los elementos del subárbol derecho son mayores o iguales que **n**. Un **árbol binario** con está propiedad se conoce como **árbol de búsqueda binario.**

3.3 Árboles Balanceados, Árboles AVL ó Árboles Adelson-Velskii-Landis

La estructura de datos más vieja y mejor conocida para **árboles balanceados** es el árbol **AVL**. Un **árbol AVL** es un árbol binario de búsqueda en el que las alturas de los subárboles izquierdo y derecho de cualquier nodo difieren a lo sumo en **1**. Para mantenerlo balanceado es necesario saber la altura o la diferencia en alturas de todos los subárboles y eso provoca que se tenga que guardar información adicional en cada nodo, un **contador de la diferencia** entre las alturas de sus dos subárboles. Probablemente la principal característica de los **árboles AVL** es su excelente tiempo de ejecución para las diferentes operaciones **$O(\log_2 n)$** (inserción, supresión y búsqueda).

El teorema demostrado por **Adelson-Velskii-Landis** garantiza que el árbol nunca tendrá una altura mayor que el **45%** respecto a un **árbol perfectamente balanceado**, por muchos nodos que haya. Un ejemplo de un **árbol AVL**, se muestra en la **Figura 3.12**. Un **árbol totalmente equilibrado** se caracteriza por que la altura de la rama izquierda es igual que la altura de la rama derecha para cada uno de los nodos del árbol. En un árbol real, no siempre se puede conseguir que esté totalmente balanceado.

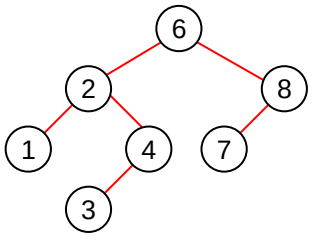


Figura 3.12: Árbol **AVL**.

El nombre es en honor de **Adelson-Velskii-Landis**, que fueron los primeros científicos en estudiar las propiedades de esta estructura de datos. Son árboles ordenados o de búsqueda que además cumplen la condición de balanceo para cada uno de los nodos.

Un **árbol equilibrado** ó **árbol AVL** es un árbol de búsqueda en el que las alturas de los subárboles izquierdo y derecho de cualquier nodo difieren a lo máximo en **1**.

La **Figura 3.13** muestra dos árboles de búsqueda, el de la izquierda está equilibrado. El de la derecha es el resultado de **insertar 2** en el anterior, según el algoritmo de inserción de árboles binarios de búsqueda. La inserción provoca que se viole la condición de equilibrio en la raíz.

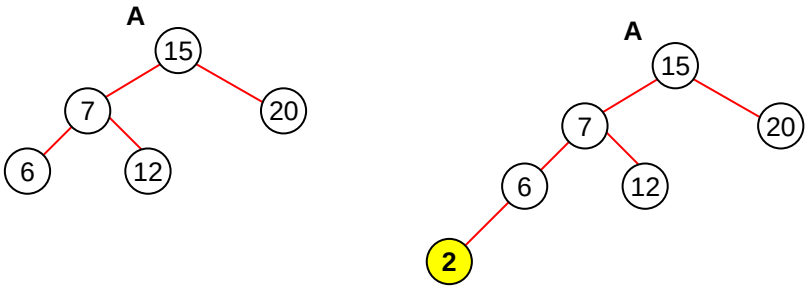


Figura 3.13: Árbol equilibrado y árbol binario.

La **condición de equilibrio** de cada nodo implica una restricción en las alturas de los subárboles de un árbol **AVL**. Si **A** es la raíz de cualquier subárbol de un árbol equilibrado, y **h** la altura de la rama izquierda, entonces la altura de la rama derecha puede tomar los valores: **h - 1**, **h** ó **h + 1**. Esto aconseja asociar a cada nodo el parámetro denominado **factor de equilibrio** (**fe**) o **balance de un nodo**, el cual se define como la **altura del subárbol derecho menos la altura del subárbol izquierdo** correspondiente. El **factor de equilibrio** de cada nodo en un árbol equilibrado puede tomar los valores: **1**, **-1** ó **0**. La **Figura 3.14** muestra un árbol balanceado con el factor de equilibrio de cada nodo.

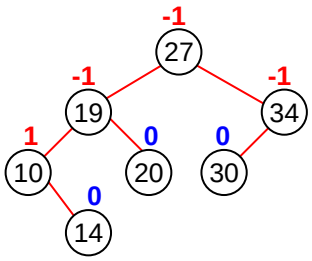


Figura 3.14: Árbol equilibrado con el **factor de equilibrio** de cada nodo.

La **altura** o **profundidad** de un árbol binario es el nivel máximo de sus hojas más **1**. La altura de un árbol nulo es cero.

Altura de un árbol equilibrado ó **árbol AVL**. No resulta fácil determinar la altura promedio de un árbol **AVL**, por lo que se determina la altura en el peor de los casos, es decir, la altura máxima que puede tener un árbol equilibrado con un número de nodos **n**. La altura es un parámetro importante ya que coincide con el número de iteraciones que se realizan para bajar desde el nodo raíz al nivel más profundo de las hojas. La eficiencia de los algoritmos de búsqueda, inserción y borrado depende de la **altura del árbol AVL**.

Inserción en árboles de búsqueda equilibrados. Los **árboles equilibrados** o **árboles AVL**, son árboles de búsqueda y por consiguiente la inserción se ha de hacer siguiendo el camino de búsqueda. El proceso de inserción se hace aplicando el algoritmo de inserción de un nuevo nodo en un árbol binario de búsqueda, comparando el contenido; con el contenido del nodo raíz, y siguiendo por la rama izquierda o derecha según sea menor o mayor; termina insertándose como nodo hoja. Esta operación, como ha quedado demostrado al determinar la altura en el peor de los casos, tiene una complejidad logarítmica. Sin embargo, la nueva inserción puede hacer que aumente la altura de una rama, de manera que cambie el factor de equilibrio del nodo de dicha rama. Este hecho hace necesario que el algoritmo de inserción, regrese por el camino de búsqueda actualizando el factor de equilibrio de los nodos. La **Figura 3.15** muestra un árbol equilibrado y el mismo árbol justo después de la inserción de un nuevo nodo que provoca se rompa la condición de balanceo.

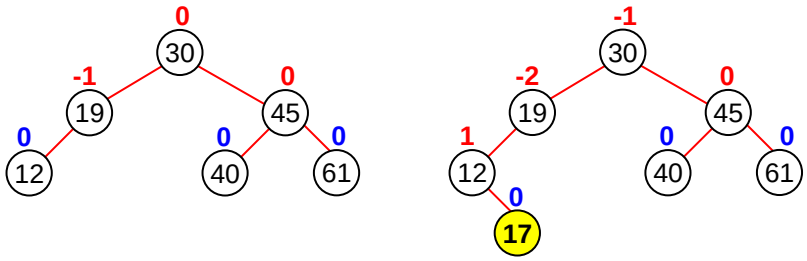


Figura 3.15: Árbol equilibrado; el mismo después de insertar el nodo **17**.

Una inserción de un nuevo nodo o su eliminación, puede destruir el criterio de equilibrio de varios nodos del árbol. Se debe de recuperar la condición de equilibrio del árbol, antes de dar por finalizada la operación, para que siga equilibrado.

La estructura del nodo en un árbol equilibrado es en lo fundamental, la misma que en un árbol binario de búsqueda. Para determinar si el árbol está equilibrado debe manejarse información relativa al **balanceo** o **factor de equilibrio** de cada nodo del árbol. Por esta razón, se añade, al tipo de datos que representa a cada nodo, un campo más; el **factor de equilibrio** (**fe**). Este campo puede tomar los valores: **-1**, **0** ó **+1**.

```
typedef struct nodo Nodo;
struct nodo {
      TipoElemento dato;
      int fe;
      Nodo *izquierdo, *derecho;
      };
```

Proceso de inserción de un nuevo nodo. Sí consideramos un árbol binario de búsqueda equilibrado, se pueden tener los siguientes casos:

i) El árbol esta **vacío** ó va a ser formado. Se crea el primer nodo como en un árbol de búsqueda binario.

ii) Sí el árbol ya esta formado. Se inserta un nuevo nodo, para lo que se aplica el algoritmo de inserción en un árbol de búsqueda, este sigue el camino de búsqueda hasta llegar al fondo del árbol y se inserta como nodo hoja y con factor de equilibrio **0**. Pero el proceso no puede terminar, hay que recorrer el camino de búsqueda en sentido contrario, hacia la raíz, para actualizar el criterio de equilibrio, en el campo adicional **factor de equilibrio**. Después de una inserción sólo los nodos que se encuentran en el camino de búsqueda pueden haber cambiado el factor de equilibrio. Pueden darse los siguientes casos.

ii.1) La actualización del **factor de equilibrio** (**fe**) puede hacer que este mejore. Esto ocurre cuando un nodo está descompensado a la izquierda y se inserta el nuevo nodo en la rama izquierda, al crecer en altura dicha rama el **fe** se hace **0**, se ha mejorado el equilibrio. La **Figura 3.16** muestra el árbol en el que el **nodo 90** tiene **fe = 1**; y el árbol después de insertar el **nodo 60**, el **nodo 90** tiene ahora un **fe = 0**.

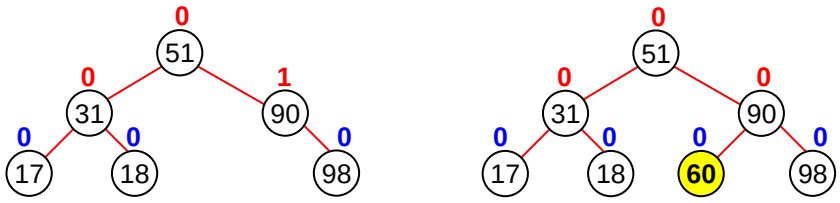


Figura 3.16: Mejora en la condición de equilibrio al insertar un nuevo **nodo 60**.

ii.2) Si al actualizar el nodo cuyas ramas izquierda y derecha del árbol tienen altura diferente, $|h_{Ri} - h_{Rd}| = 1$, si se inserta el nodo en la rama más alta, entonces se rompe el criterio de equilibrio del árbol, la diferencia de altura pasa a ser **2** y es necesario reestructurarlo.

Hay cuatro casos que se deben tener en cuenta al reestructurar un **nodo A**, según el lugar donde se haya hecho la inserción (**Figura 3.17**):

1. Inserción en el subárbol izquierdo de la rama izquierda de **A**.
2. Inserción en el subárbol derecho de la rama izquierda de **A**.
3. Inserción en el subárbol derecho de la rama derecha de **A**.
4. Inserción en el subárbol izquierdo de la rama derecha de **A**.

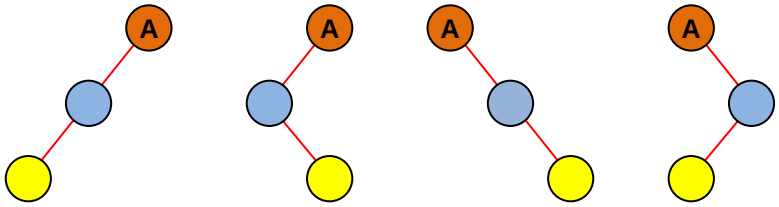


Figura 3.17: Cuatro tipos de reestructuraciones del equilibrio de un nodo.

El caso **1.** y **3.** (izquierda-izquierda y derecha-derecha), se resuelven con una **rotación simple**. El caso **2.** y **4.** (izquierda-derecha y derecha-izquierda), se resuelven con una **rotación doble**.

La **rotación simple** implica a dos nodos, el **nodo A** (nodo con **|fe| = 2**) y el descendiente izquierdo o derecho según el caso.

En la **rotación doble** están implicados los tres nodos, el **nodo A**, **nodo descendiente izquierdo** y el **descendiente derecho** de este; o bien el caso simétrico, **nodo A**, **descendiente derecho** y el **descendiente izquierdo** de este.

La **rotación simple** resuelve la violación del equilibrio de un nodo **izquierda-izquierda** ó la **derecha-derecha**. El árbol de la **Figura 3.18** tiene el **nodo A** con factor de equilibrio **-1**; luego se muestra el árbol después de insertar el **nodo C**. Resulta que ha crecido la altura de la rama izquierda, es un desequilibrio izquierda-izquierda que se resuelve con una rotación simple, que en este caso se puede denominar **rotación Izquierda-Izquierda**, **rotación II** ó **rotación simple derecha**. La **Figura 3.19** es el árbol resultante de la rotación, en el que el **nodo B** se ha convertido en la raíz, en el **nodo A** su rama derecha y el **nodo C** continúa como rama izquierda. Con estos movimientos el árbol sigue siendo de búsqueda y se equilibra.

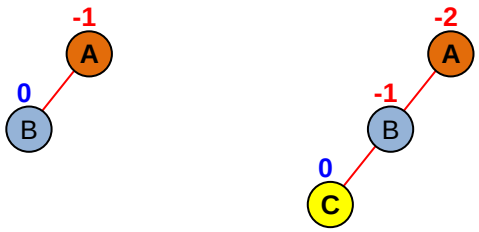


Figura 3.18: Árbol binario **AVL** y árbol después de insertar un nuevo nodo por la izquierda.

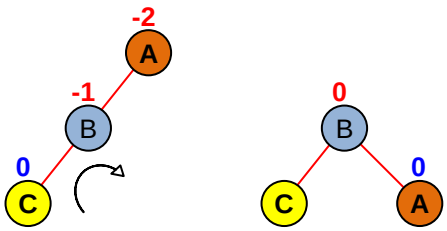


Figura 3.19: Árbol binario después de **rotación II**.

En la **Figura 3.20** se muestra el otro caso de violación de la condición de equilibrio que se resuelve con una **rotación simple**. Inicialmente, el **nodo A** tiene como **factor de equilibrio +1**, al insertar el **nodo C** y después de actualizar, el factor de equilibrio de **A** pasa a ser **+2**, se ha insertado por la derecha y por consiguiente ha crecido la altura de la rama derecha. La **Figura 3.21** muestra la resolución de este desequilibrio, una rotación simple que se puede denominar **rotación Derecha-Derecha**, **rotación DD** ó **rotación simple izquierda**. En el árbol resultante de la rotación el **nodo B** se ha convertido en la raíz, el **nodo A** es su rama izquierda y el **nodo C** continúa como rama derecha. Con estos movimientos el árbol sigue siendo de búsqueda y se equilibra.

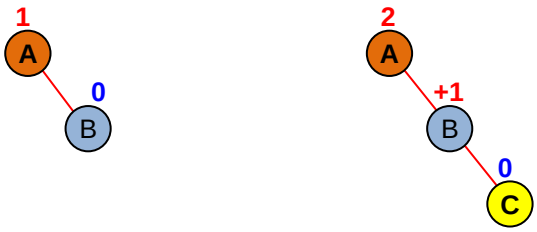


Figura 3.20: Árbol binario **AVL** y árbol después de insertar un nuevo nodo por la derecha.

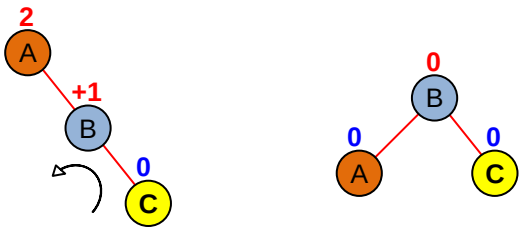


Figura 3.21: Árbol binario después de **rotación DD**.

Rotación doble. En la **Figura 3.22a** se observa un árbol de búsqueda desequilibrado, con factores de equilibrio **+2**, **-1** y **0**. La única solución consiste en subir el **nodo C** como raíz del subárbol, como rama izquierda situar al **nodo A** y como rama derecha al **nodo B**, la altura del subárbol resultante es la misma que antes de insertar. Se ha realizado una **rotación doble** ó **rotación derecha-izquierda**, en la que intervienen los tres nodos.

En la **Figura 3.22c** se observa un árbol de búsqueda desequilibrado, con factores de equilibrio **-2**, **+1** y **0**. La única solución consiste en subir el **nodo C** como raíz del subárbol, como rama izquierda situar al **nodo B** y como rama derecha al **nodo C**, la altura del subárbol resultante es la misma que antes de insertar. Se ha realizado una **rotación doble** ó **rotación izquierda-derecha**, en la que intervienen los tres nodos.

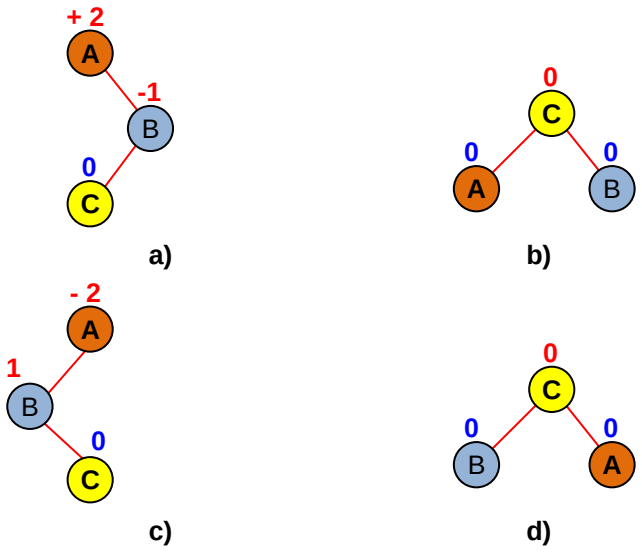


Figura 3.22 a) Árbol después de insertar el **nodo C**.
b) Rotación doble que resuelve el problema.

La **Figura 3.23c** muestra un árbol binario de búsqueda después de insertar el **nodo 60**. Al volver por el camino de búsqueda para actualizar los factores de equilibrio, el **nodo 75** pasa a tener **fe = -1** (se ha insertado por su izquierda), el **nodo 50** pasa a tener **fe = +1** y el **nodo 80** tendrá **fe = +2**. Es el caso simétrico al descrito en la **Figura 3.22**, se restablece el equilibrio con una rotación doble, simétrica con respecto a la anterior como se muestra en la **Figura 3.23d**.

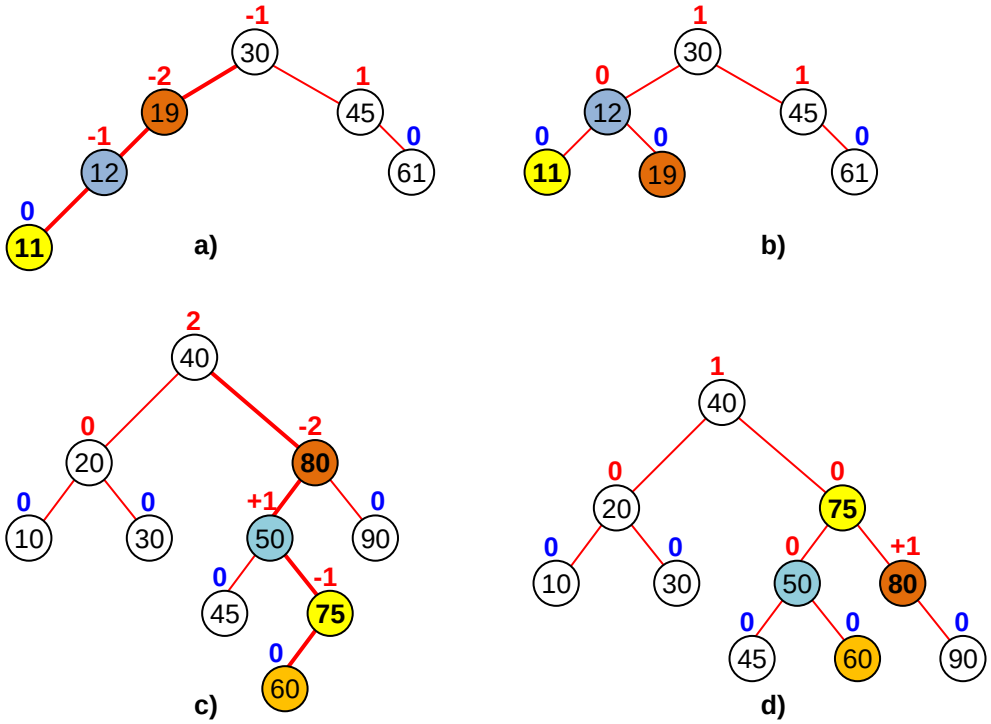


Figura 3.23: a) Árbol equilibrado de búsqueda después de insertar el **nodo 11**.
b) Rotación simple, izquierda-izquierda.
c) Árbol equilibrado de búsqueda después de insertar el **nodo 60**.
d) Rotación doble, izquierda derecha.

Eliminación de un nodo en un árbol equilibrado. La operación de eliminación consiste en eliminar un nodo con un dato de un árbol de búsqueda equilibrado. Evidentemente, el árbol resultante debe seguir siendo un árbol de búsqueda balanceado ($|h_{Ri} - h_{Rd} <= 1$).

El **algoritmo de eliminación** puede descomponerse en dos partes diferenciadas. La **primera** sigue la estrategia de la eliminación en árboles de búsqueda. La **segunda** es la actualización del **factor de equilibrio**, para lo que recorre el camino de búsqueda hacia la raíz, actualizando el factor de equilibrio de los nodos. Ahora, al eliminarse un nodo, la altura de la rama en que se encuentra disminuye en **1**.

Si después de la actualización de un nodo ocurre que se viola la condición equilibrio, el **fe = ± 2**, hay que restaurar el equilibrio con una rotación simple o doble.

En el algoritmo para eliminar un nodo que contiene un cierto dato, lo primero que se hace es buscar el nodo, para lo que se sigue el camino de búsqueda. A continuación se procede a eliminar el nodo, se distinguen los siguientes casos:

1. El nodo a borrar es un **nodo hoja** o con **un único descendiente**. Entonces, simplemente se suprime, o bien se sustituye por su descendiente.

2. El nodo a eliminar tiene **dos subárboles**. En este caso se busca el nodo más a la derecha del subárbol izquierdo, es decir, el de mayor valor en el subárbol de los menores, se copia este en el nodo a eliminar y por último se elimina el nodo copiado (que tiene las características del primer caso).

Una vez eliminado el nodo, el algoritmo tiene que prever actualizar los factores de equilibrio de los nodos que han formado el camino de búsqueda ya que la altura de alguna de las dos ha decrecido. Por consiguiente, regresa por el camino de búsqueda, hacia la raíz, calculando los nuevos factores de equilibrio de los nodos visitados. No siempre es necesario recorrer todo el camino de regreso, si se actualiza un nodo con **factor de equilibrio 0**, el nuevo será **± 1**, pero la altura neta del subárbol con raíz en el nodo actualizado no ha cambiado, entonces el algoritmo termina ya que los factores de equilibrio de los nodos restantes no cambian.

En la **Figura 3.24a**, se observa un árbol del cual se elimina el **nodo 30**, y en la **Figura 3.24b** se observa el árbol al actualizar los factores de equilibrio para todos los nodos.

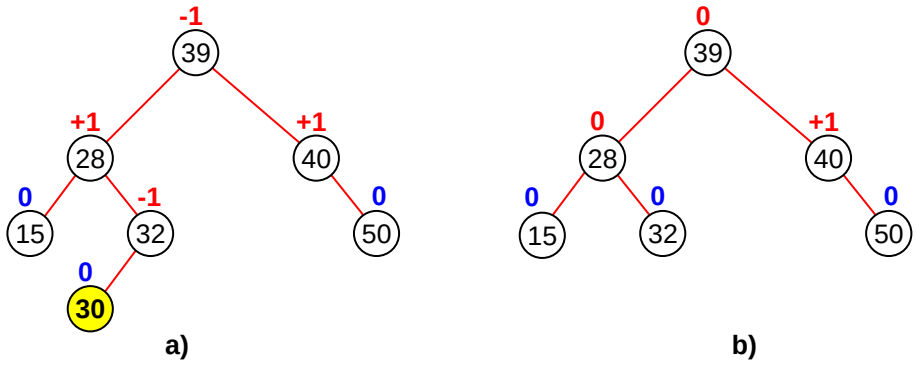


Figura 3.24: a) Árbol de búsqueda equilibrado.
b) Después de eliminar el **nodo 30** y actualizar factores de equilibrio

La **Figura 3.25a** muestra un árbol equilibrado, en la **Figura 3.25b** el árbol resultante después de eliminar el **nodo 90**, al volver por el camino de búsqueda del **nodo 70** su factor de equilibrio pasa a ser **0**, el **nodo 60** tiene como factor **0** se actualiza y pasa a ser **-1** (ha disminuido su rama derecha), su altura neta no ha cambiado y sigue siendo **3**, por lo que el algoritmo termina.

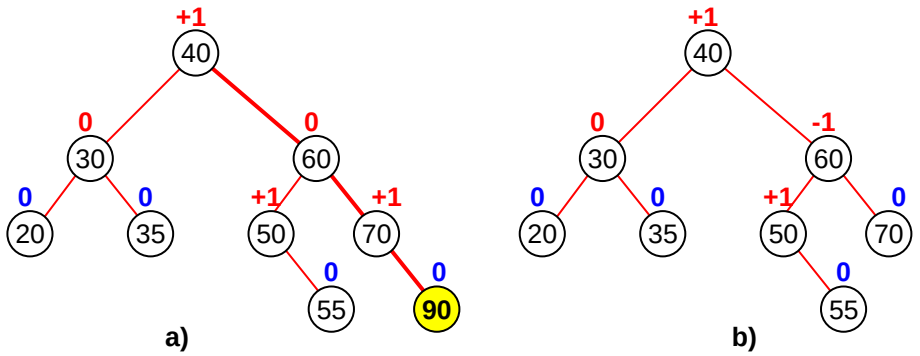


Figura 3.25: a) Árbol de búsqueda equilibrado. **b)** El árbol después de eliminar el **nodo 90**.

En la **Figura 3.26b** se tiene el árbol de búsqueda después de eliminar el **nodo 25** del árbol de la **Figura 3.26a**. Al actualizar los nodos del camino de búsqueda, el **nodo 20** tiene como **fe = +1**, éste pasa a ser **0** ya que se ha eliminado por la rama derecha. El proceso continúa ya que la altura del subárbol de **raíz 20** ha disminuido, el nodo antecesor, **29**, tiene como **fe = +1**, éste pasa a ser **+2** ya que se ha eliminado por su rama izquierda, viola la condición de equilibrio. Para restaurar el equilibrio es necesario, en este caso, una rotación simple, derecha-derecha, ya que el nodo de la rama derecha de **29** tiene como **fe = 0**. La **Figura 3.26c** es el árbol equilibrado después de la rotación.

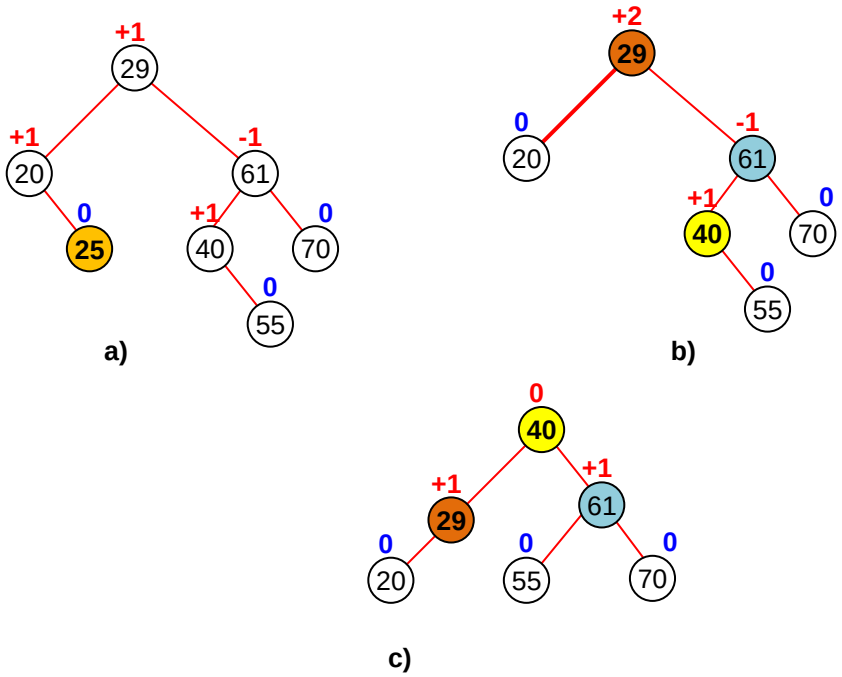


Figura 3.26: a) Árbol de búsqueda equilibrado. **b)** El árbol después de eliminar el **nodo 25**. **c)** Árbol una vez realizada la rotación doble derecha-izquierda.

La **Figura 3.27b** es un árbol de búsqueda después de haber eliminado el **nodo 21**, y con la actualización de los factores de equilibrio, el **nodo 43** está desequilibrado, hay que aplicar una rotación simple derecha-derecha. La **Figura 3.27c** muestra el árbol después de la rotación y la posterior actualización del factor de equilibrio del **nodo 70**, que también exige aplicar otra rotación, en este caso una rotación doble derecha-izquierda. La **Figura 3.27d** es el árbol equilibrado después de la última rotación y del fin del algoritmo.

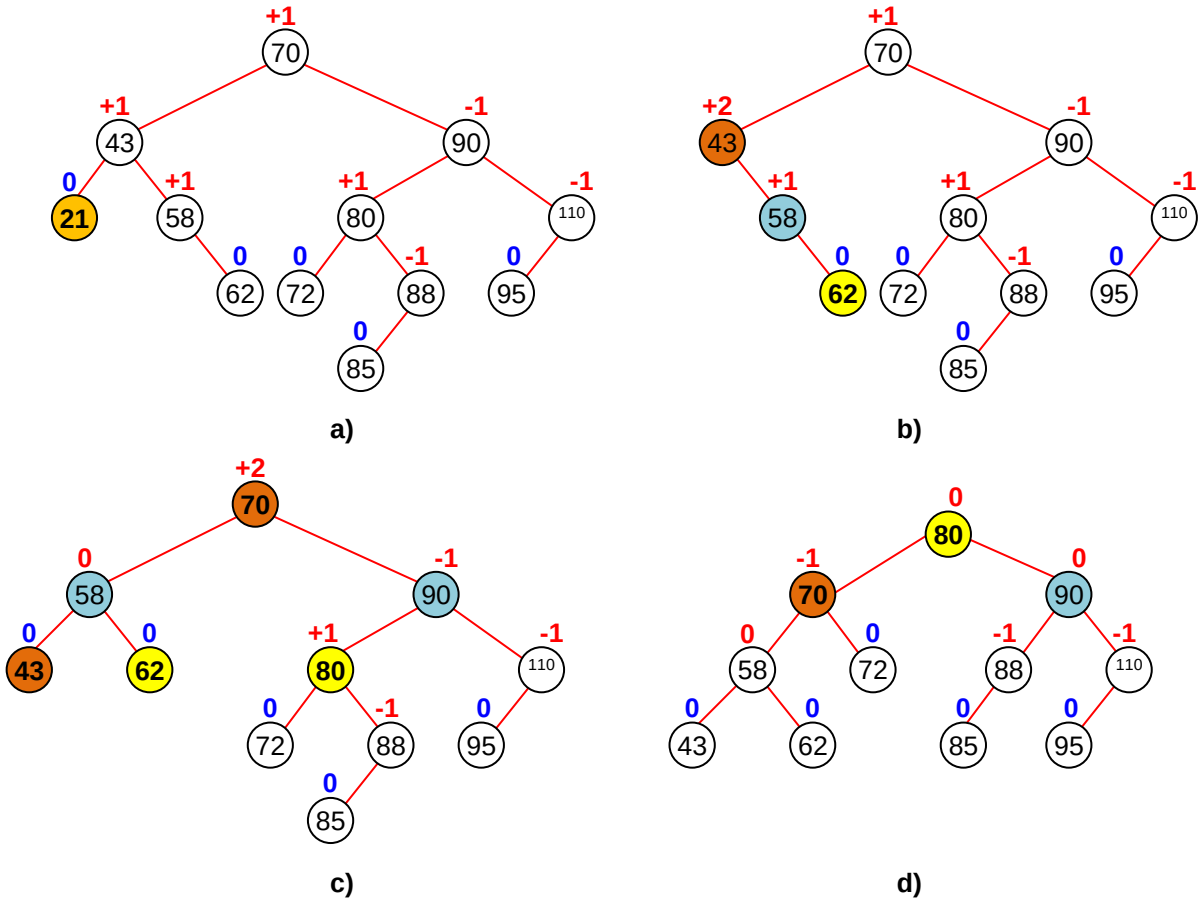


Figura 3.27: a) Árbol de búsqueda equilibrado. **b)** El árbol después de eliminar el **nodo 21**.
c) Árbol una vez realizada la rotación derecha-derecha.
d) Árbol después de la nueva rotación derecha -izquierda.

En el **Programa 3.3** se observa la implementación de algunas de las operaciones básicas que es posible realizar con árboles **AVL**, en la **Figura 3.28** se muestra el resultado al ejecutar el programa, obsérvese, que cuando se repite un número, por ejemplo el **35**, este no crea un nuevo nodo, y simplemente se ignora.

```
#include "stdlib.h"

/* Estructura del nodo en un árbol AVL                                    */
typedef struct nodo avl;
struct nodo{
int dato;
int altura;
avl *izquierdo;
avl *derecho;
};
```

Programa 3.3: Operaciones básicas en un **árbol AVL** (Parte **1**/6).

```
/* Mensaje de ERROR                                                    */
void error(void){
perror("\n\t\aERROR: Memoria insuficiente...");
exit(1);
}

/* Crea un nuevo nodo del tipo de la estructura                        */
avl *Nuevo(){
avl *q = (avl *)malloc(sizeof(avl));
if(!q) error();
return(q);
}

/* Cuenta el número de nodos en el árbol                               */
int num_nodos(avl *p){
if(p == NULL) return(0);
return(num_nodos(p -> izquierdo) + 1 + num_nodos(p -> derecho));
}

/* Borra todo el árbol AVL                                             */
avl *borrar(avl *p){
if(p != NULL){
      borrar(p -> izquierdo);
      borrar(p -> derecho);
      free(p);
      }
return(NULL);
}

/* Busca un dato en el árbol                                           */
avl *buscar(int dato, avl *p) {
if(p == NULL) return(NULL);
if(dato < p -> dato) return(buscar(dato, p -> izquierdo));
else if(dato > p -> dato) return(buscar(dato, p -> derecho));
else return(p);
}

/* Busca el mínimo en el árbol                                         */
avl *buscar_min(avl *p) {
if(p == NULL) return(NULL);
else if(p -> izquierdo == NULL) return(p);
else return(buscar_min( p -> izquierdo ));
}

/* Busca el máximo en el árbol                                         */
avl *buscar_max(avl *p) {
if(p != NULL) while(p -> derecho != NULL) p = p -> derecho;
return(p);
}

/* Calcula la altura del árbol                                         */
int altura(avl *p){
if(p == NULL ) return(-1);
else return(p -> altura);
}
```

Programa 3.3: Operaciones básicas en un **árbol AVL** (Parte **2**/6).

```
/* Regresa el máximo entre A y B                                    */
int Max(int a, int b) {
return(a > b ? a : b);
}

/* Rotación simple a la izquierda                                   */
avl *rot_izq(avl *p) {
avl *q;
q = p -> izquierdo;
p -> izquierdo = q -> derecho;
q -> derecho = p;
p -> altura = Max(altura(p -> izquierdo ), altura( p -> derecho)) + 1;
q -> altura = Max(altura(q -> izquierdo ), p -> altura) + 1;
return(q);  /* Nueva Raíz                                           */
}

/* Rotación simple a la derecha                                     */
avl *rot_der(avl *q){
avl *p;
p = q -> derecho;
q -> derecho = p -> izquierdo;
p -> izquierdo = q;
q -> altura = Max(altura(q -> izquierdo), altura(q -> derecho)) + 1;
p -> altura = Max(altura(p -> derecho), q -> altura) + 1;
return(p);  /* Nueva raíz                                           */
}

/* Rotación doble a la izquierda                                    */
avl *rot_dob_izq(avl *k){
k -> izquierdo = rot_der(k -> izquierdo);
return(rot_izq(k));
}

/* Rotación doble a la derecha                                      */
avl *rot_dob_der(avl *q) {
q -> derecho = rot_izq(q -> derecho);
return(rot_der(q));
}

/* Recorre el árbol en Pre-Orden                                    */
void pre_orden(avl *p){
if(p != NULL){
      printf("%d ",p -> dato);
      pre_orden(p -> izquierdo);
      pre_orden(p -> derecho);
      }
}

/* Recorre el árbol En-Orden                                        */
void en_orden(avl *p){
if(p != NULL) {
      en_orden(p -> izquierdo);
      printf("%d ",p -> dato);
      en_orden(p -> derecho);
      }
}
```

Programa 3.3: Operaciones básicas en un **árbol AVL** (Parte **3**/6).

```
/* Recorre el árbol en Post-Orden                                          */
void post_orden(avl *p){
if(p != NULL){
      post_orden(p -> izquierdo);
      post_orden(p -> derecho);
      printf("%d ",p -> dato);
      }
}

/* Inserta un dato en el árbol p                                           */
avl *insertar(int dato, avl *p){
if(p == NULL) {
      p = Nuevo();
      p -> dato = dato;
      p -> altura = 0;
      p -> izquierdo = NULL;
      p -> derecho = NULL;
      }
else if(dato < p -> dato) {
      p -> izquierdo = insertar(dato, p -> izquierdo);
      if(altura(p -> izquierdo) - altura(p -> derecho) == 2)
      if(dato < p -> izquierdo -> dato) p = rot_izq(p);
      else p = rot_dob_izq(p);
      }
else if(dato > p -> dato) {
      p -> derecho = insertar(dato, p -> derecho);
      if(altura(p -> derecho) - altura(p -> izquierdo) == 2)
      if(dato > p -> derecho -> dato) p = rot_der(p);
      else p = rot_dob_der(p);
      }
p -> altura = Max(altura(p -> izquierdo), altura(p -> derecho)) + 1;
return(p);
}

/* Pone menú                                                               */
void menu(void){
printf("\n\tOperaciones con Árboles AVL");
printf("A = Llenar un árbol de forma aleatoria");
printf("C = Llenar un árbol de forma manual");
printf("B = Buscar un dato");
printf("M = Buscar Mínimo y Máximo");
printf("N = Contar Nodos y Altura");
printf("R = Ver Recorridos");
printf("V = Ver árbol");
printf("Q = Salir");
printf("Elija una Opción : ");
}

/* Recorre y muestra todos los nodos                                       */
void ver(int nivel,avl *p){
int i;
if(p != NULL) {
      ver(nivel + 1,p -> derecho);
      printf("\n");
      for(i = 0;i < nivel;i++) printf("   ");
```

Programa 3.3: Operaciones básicas en un **árbol AVL** (Parte **4**/6).

```
      textcolor(nivel + 7);
      cprintf("%d",p -> dato);
      ver(nivel + 1,p -> izquierdo);
      }
}

/* Programa principal                                                    */
void main(void){
int n,i,dato;
char op;
avl *p = NULL,*q;       /* Árbol Vacío                                   */
while(1){
      menu();
      op = tolower(getch());
      switch(op){
            case 'a':
                  printf("Numero de nodos del árbol: ");
                  scanf("%d",&n);
                  n = abs(n);
                  randomize();
                  for(i = 0;i < n;i++) {
                        printf("\n\tElemento No. %d: ",i + 1);
                        dato = random(100);
                        printf("%d",dato);
                        p = insertar(dato,p);
                        }
                  break;
            case 'c':
                  printf("Numero de nodos del árbol: ");
                  scanf("%d",&n);
                  n = abs(n);
                  randomize();
                  for(i = 0;i < n;i++) {
                        printf("\n\tElemento No. %d: ",i + 1);
                        scanf("%d",&dato);
                        p = insertar(dato,p);
                        }
                  break;
            case 'v':
                  printf("Árbol AVL");
                  ver(0,p);
                  break;
            case 'm':
                  q = buscar_min(p);
                  printf("\n\n\tMínimo = %d",q -> dato);
                  q = buscar_max(p);
                  printf("\n\n\tMáximo = %d",q -> dato);
                  break;
            case 'n':
                  printf("\n\n\tNodos : %d\n\n\tAltura =
                        %d",num_nodos(p),altura(p));
                  break;
            case 'b':
                  printf("\n\tDato a Buscar: ");
                  scanf("%d",&dato);
                  q = buscar(dato,p);
```

Programa 3.3: Operaciones básicas en un **árbol AVL** (Parte **5**/6).

```
                if(q != NULL) printf("Se encontró el dato %d",dato);
                else printf("No se encontró el dato %d",dato);
                break;
          case 'r':
                printf("Recorrido en Pre-Orden:");
                pre_orden(p);
                printf("Recorrido En Orden:");
                en_orden(p);
                printf("Recorrido en Post-Orden:");
                post_orden(p);
                break;
          case 'q':
                borrar(p);  /* Borra el árbol                          */
                exit(1);
                break;
          }
     getch();
     }
}
```

Programa 3.3: Operaciones básicas en un **árbol AVL** (Parte **6**/6).

```
        Operaciones Básicas con un árbol AVL
A = Llenar un árbol de forma aleatoria
C = Llenar un árbol de forma manual
B = Buscar un dato
M = Buscar Mínimo y Máximo
N = Contar Nodos y Altura
R = Ver Recorridos
V = Ver árbol
Q = Salir
Elija una Opción: A
Numero de nodos del árbol: 10
Elemento No. 1: 35
Elemento No. 2: 12
Elemento No. 3: 21
Elemento No. 4: 46
Elemento No. 5: 66
Elemento No. 6: 85
Elemento No. 7: 36
Elemento No. 8: 26
Elemento No. 9: 35
Elemento No. 10: 43
> B
Dato a Buscar: 43
Se encontró el dato 43
> M
Mínimo = 12
Máximo = 85
> N
Nodos: 9
Altura = 3
> R
Recorrido en Pre-Orden:
46 35 21 12 26 36 43 66 85
```

Figura 3.28: Operaciones básicas con un **árbol AVL** (Parte **1**/2).

```
Recorrido En Orden:
12 21 26 35 36 43 46 66 85
Recorrido en Post-Orden:
12 26 21 43 36 35 85 66 46
> V
        Árbol AVL
      85
   66
46
         43
      36
   35
         26
      21
         12
```

Figura 3.28: Operaciones básicas con un **árbol AVL** (Parte **2**/2).

3.4 Árboles B

Un **árbol B** es un tipo especial de árbol con determinadas propiedades que lo hacen útil para guardar y acceder de forma eficiente a grandes cantidades de información en memoria secundaria. **Bayer** y **McCreight** desarrollaron en **1970** la idea de los **árboles B**, motivados por encontrar una nueva técnica para administrar grandes volúmenes de información.

En un **árbol B** la **búsqueda** de un elemento requiere del recorrido de un camino, desde la raíz del árbol hacia alguna de las hojas; estos árboles están completamente balanceados, por lo que se garantiza eficiencia en los algoritmos de **búsqueda**, **inserción** y **supresión**. Sin embargo, el proceso de **inserción** y **supresión** de elementos varía ligeramente respecto a un árbol binario tradicional. A diferencia del **árbol binario balanceado** (**AVL**), los **árboles B** pueden guardar en sus nodos más de un elemento y tener más de dos hijos, por lo que son **árboles n-arios**. Esta propiedad permite que se almacenen grandes cantidades de información sin que la altura del árbol sea muy grande, lo que optimiza los algoritmos de acceso en memoria secundaria.

Una definición general del **árbol B** es la siguiente:

Un **árbol B de orden n** es aquel en el que:

1. Todas las hojas en el árbol están en el mismo nivel.
2. Cada nodo contiene entre **n** y **2n** elementos (excepto la raíz que tiene entre **1** y **2n**).
3. Si un nodo tiene **m** elementos, el nodo contendrá **0** ó **m + 1** hijos.
4. Los elementos de un nodo del árbol están ordenados linealmente en el nodo.
5. Los elementos del **árbol B** están organizados siguiendo las propiedades de un árbol de búsqueda binaria, es decir, los elementos menores a la izquierda y los mayores a la derecha del nodo original (**Figura 3.29**).

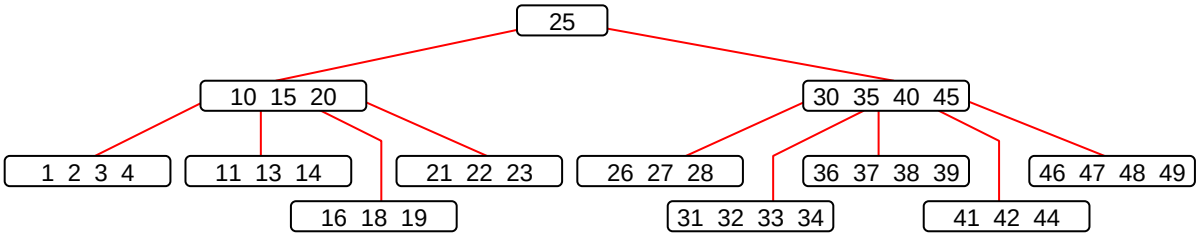


Figura 3.29: Ejemplo de un **árbol B**. Este árbol puede ser de **orden 2** o de **orden 3**, sin romper ninguna de las reglas.

Puesto que el **árbol B** es una estructura para guardarse en memoria secundaria, los nodos de estos árboles pueden almacenarse como una página en disco. Las aplicaciones reales de **árboles B** manejan órdenes de **100** o más elementos; sin embargo, para ilustrar su funcionamiento se manejarán árboles **B** con un orden pequeño.

Proceso de inserción en un árbol B

El algoritmo general para dar de alta un nuevo elemento en un **árbol B** de **orden n** se muestra enseguida:

Buscar el **nodo hoja** donde se debe insertar el nuevo elemento. Sobre este nodo pueden ocurrir los siguientes casos:

1. El nodo **no está lleno**, es decir, tiene menos de **2n** elementos; y tiene capacidad para guardar más. Debe insertarse el nuevo elemento en el nodo y terminar con el proceso de inserción.
2. El **nodo está lleno**, es decir, tiene **2n** elementos; por lo que no tiene capacidad de guardar más. Debe hacerse una división del nodo de la siguiente forma:

   Dado que se tienen **2n + 1** elementos:

- Se crea un nuevo nodo y recibe a los **n** elementos más grandes.
- Los **n** elementos más pequeños quedan en el nodo donde ocurrió el desborde.
- El valor medio pasa a ser el padre de los dos nodos mencionados anteriormente.

El proceso de inserción continúa. Se inserta el valor medio (padre) en el nodo correspondiente, verificando de nuevo en qué caso se está ubicado.

3. El **nodo está lleno** y es la **raíz del árbol**. Debe hacerse una división del nodo de la siguiente forma:

   Dado que se tienen **2n + 1** elementos:

- Se crea un nuevo nodo y recibe a los **n** elementos más grandes.
- Los **n** elementos más pequeños quedan en el nodo donde ocurrió el desborde.
- Se crea un nuevo nodo, que toma como único elemento el valor medio y este nodo pasa a ser la nueva raíz del árbol.

- Termina el proceso de inserción.

A través del siguiente ejemplo, se ilustra el funcionamiento general del algoritmo de **inserción**.

Ejemplo: Suponga que se tiene el **árbol B** de **orden 2** de la **Figura 3.30**. La cantidad máxima de elementos que puede tener un nodo es cuatro, el número mínimo es dos (excepto la raíz) y la cantidad de hijos que tendrá un nodo depende de la cantidad de elementos del nodo, pero podrá variar entre **0**, **2**, **3**, **4** ó **5** hijos.

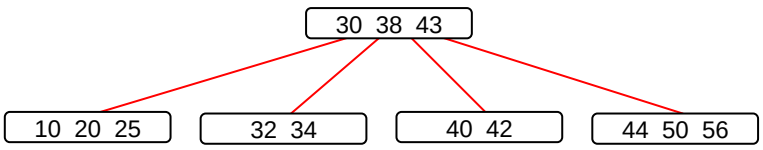


Figura 3.30: Ejemplo de un **árbol B** (Original).

Al insertar el **elemento 58** sobre el árbol se cae en el **caso 1** (**Figura 3.31**), puesto que el nodo tiene espacio para recibir un elemento más. El árbol queda así:

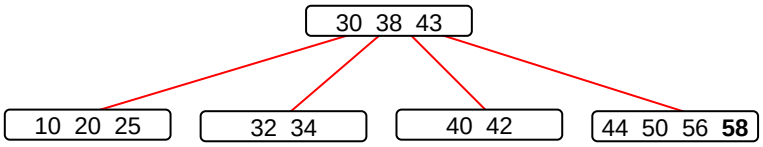


Figura 3.31: Ejemplo de un **árbol B** (después de insertar **58**).

Al insertar el **elemento 60** sobre el árbol se cae en el **caso 2** (**Figura 3.32**), por lo que ocurre un desborde en el nodo y existe la necesidad de una división. El árbol queda así:

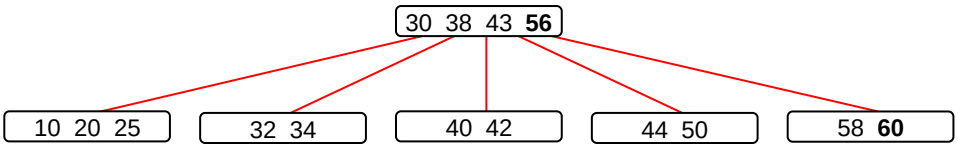


Figura 3.32: Ejemplo de un **árbol B** (después de insertar **60**).

Después de insertar los **elementos 52**, **54** y **46** (**Figura 3.33**), el árbol tiene que crecer en un nivel (**caso 3**) y queda de la siguiente forma:

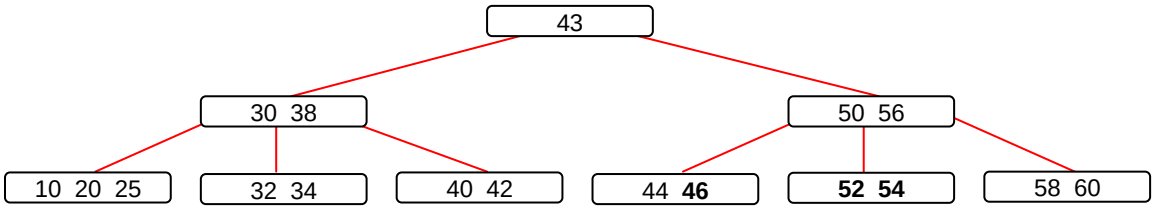


Figura 3.33: Ejemplo de un **árbol B** (después de insertar **52**, **54** y **46**).

Observaciones importantes sobre la inserción:

- El árbol siempre se resiste a crecer. Se trata de distribuir los elementos en los nodos ya existentes.
- Cuando el árbol aumenta en altura, sólo se agrega una nueva raíz. El árbol siempre permanece balanceado y crece de abajo hacia arriba.

Proceso para eliminar un elemento en el árbol B

El algoritmo general para **suprimir** un elemento del **árbol B** de **orden n** considera casos parecidos a los de la inserción.

Al igual que en los otros tipos de árboles de búsqueda, la supresión siempre se dará en un **nodo hoja**; cuando el elemento no está directamente allí, se busca quién lo sustituya aplicando alguna de las metodologías de sustitución ya conocidas: el "elemento mayor de los menores" (del subárbol izquierdo, el nodo de más a la derecha, el último elemento en la lista) o el elemento "menor de los mayores" (del subárbol derecho el nodo más a la izquierda, el primer elemento en la lista).

Una vez que se ha encontrado el nodo hoja donde se borrará un elemento (sea el elemento real o el sustituto), se procede de acuerdo con lo siguiente:

1. El nodo que contiene el elemento por eliminar posee más del mínimo de elementos. En este caso, se elimina el elemento del nodo y el proceso de eliminación termina (**Figura 3.34**).

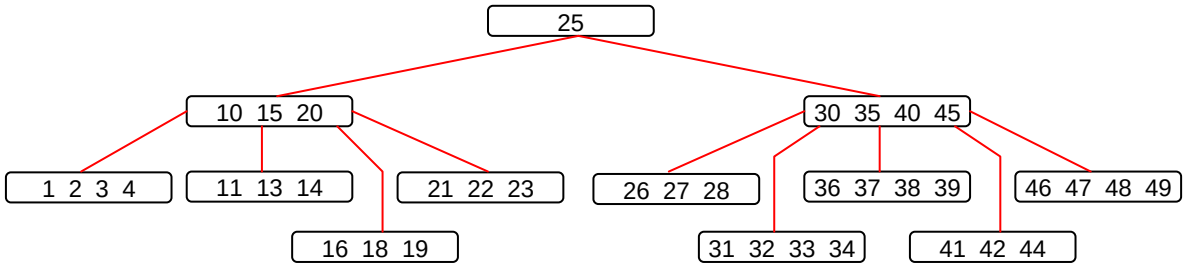


Figura 3.34: La eliminación de cualquier elemento del árbol, si es de **orden 2**, no afecta la cantidad mínima de elementos que puede tener un nodo.

2. El nodo que contiene el elemento por eliminar posee exactamente el mínimo de elementos. En este caso se buscará otro elemento que pueda sustituir al que se dará de baja y, si no existe un sustituto, se deberá unir dos nodos.

Cuando alguno de los hermanos adyacentes al nodo que se dará de baja tiene más del mínimo de elementos, el elemento padre del nodo con el elemento por borrar lo sustituirá cuando sea borrado. Un elemento (el mayor o el menor, según el caso) del nodo hermano adyacente con más del mínimo de elementos pasa a ser el nuevo elemento padre y el proceso de eliminación termina (**Figura 3.35**).

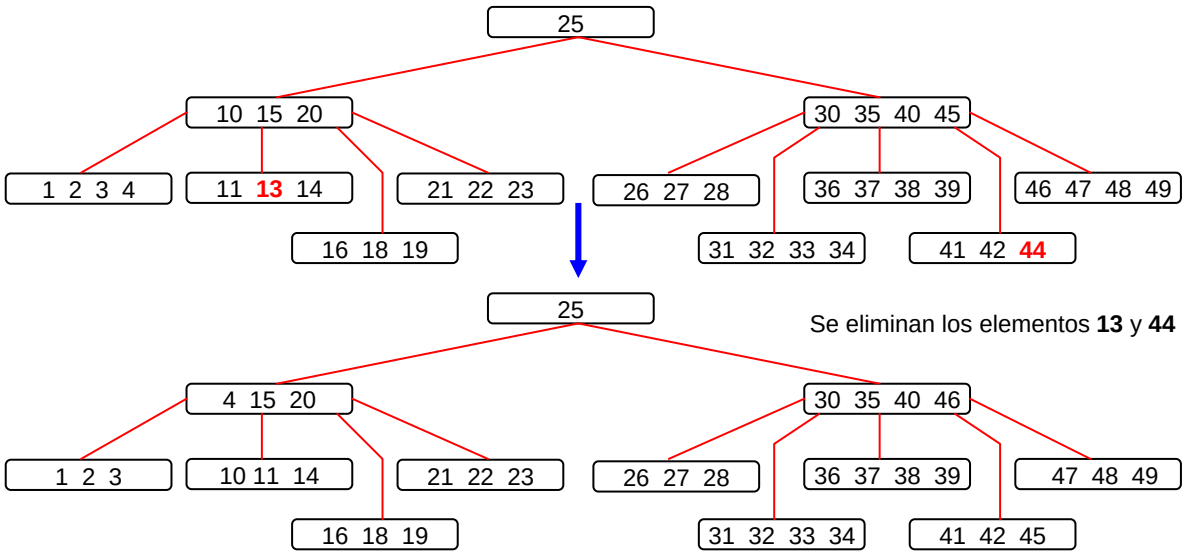


Figura 3.35: Si el árbol es de **orden 3**, la eliminación de los **elementos 13** y **44** provoca un ajuste en el árbol, de tal forma que algún nodo hermano adyacente con más del mínimo de elementos transfiere un elemento para no desbalancear el árbol.

Si no existiera algún nodo hermano adyacente con más del mínimo de elementos se procederá de la siguiente manera: el nodo donde ocurre del mínimo de elementos termina el proceso (**Figura 3.36**). Si no, se analiza la eliminación en el nodo padre aplicando los casos que ya se han explicado (**Figura 3.37**).

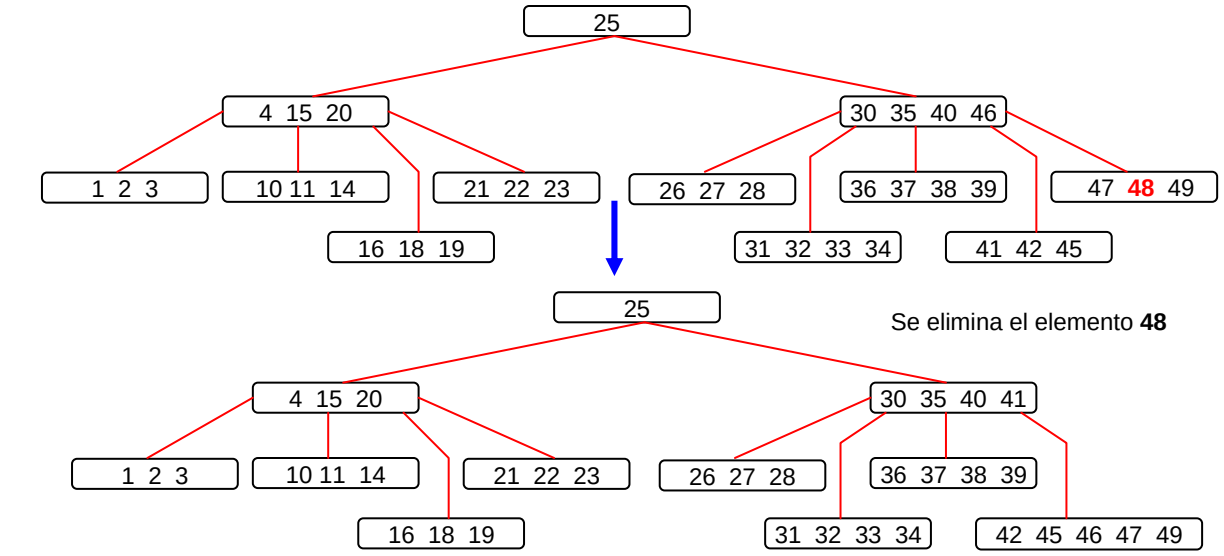


Figura 3.36: Si el árbol es de **orden 3**, la eliminación del **elemento 48** provoca la unión del nodo donde se encontraba con su nodo hermano adyacente, eliminando a su elemento padre del nodo padre, sin afectar el resto del árbol.

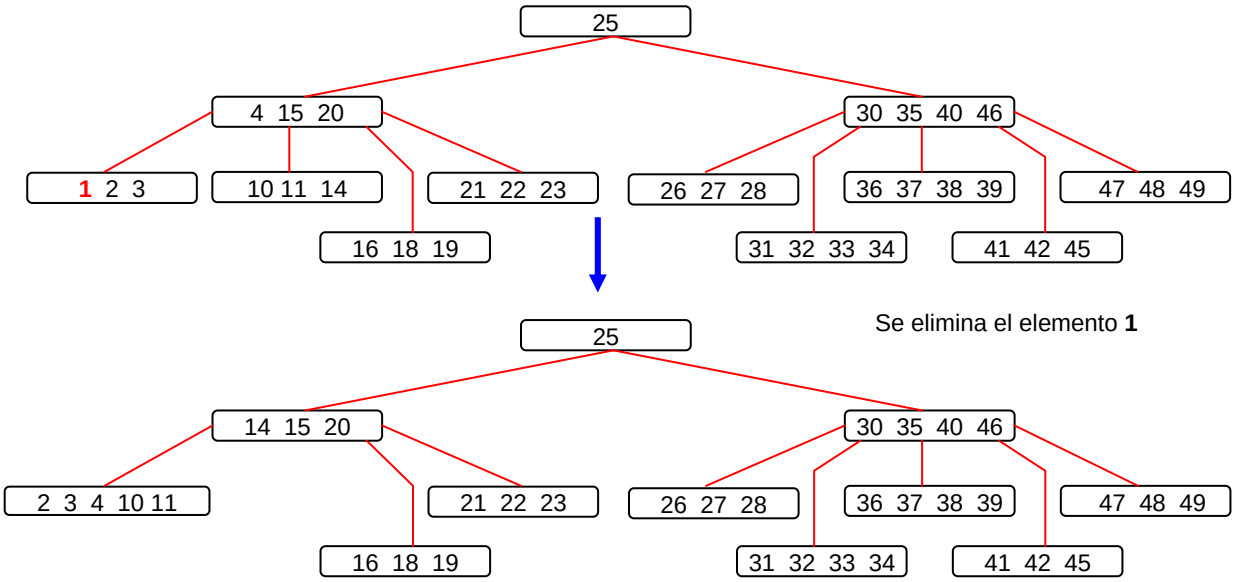


Figura 3.37: Si el árbol es de **orden 3**, la eliminación del **elemento 1** provoca la unión del nodo donde se encontraba con su hermano adyacente y, además, la transferencia de un elemento al nodo padre, para completar su cantidad mínima de elementos. Observe que se movió un nodo de un hermano a otro.

3. El nodo que contiene el elemento por eliminar es la raíz del árbol y posee sólo un elemento. En este caso, el nodo desaparece y la nueva raíz del árbol será el nodo resultado de la unión de los dos hijos, que dependían de la raíz anterior (**Figura 3.38**).

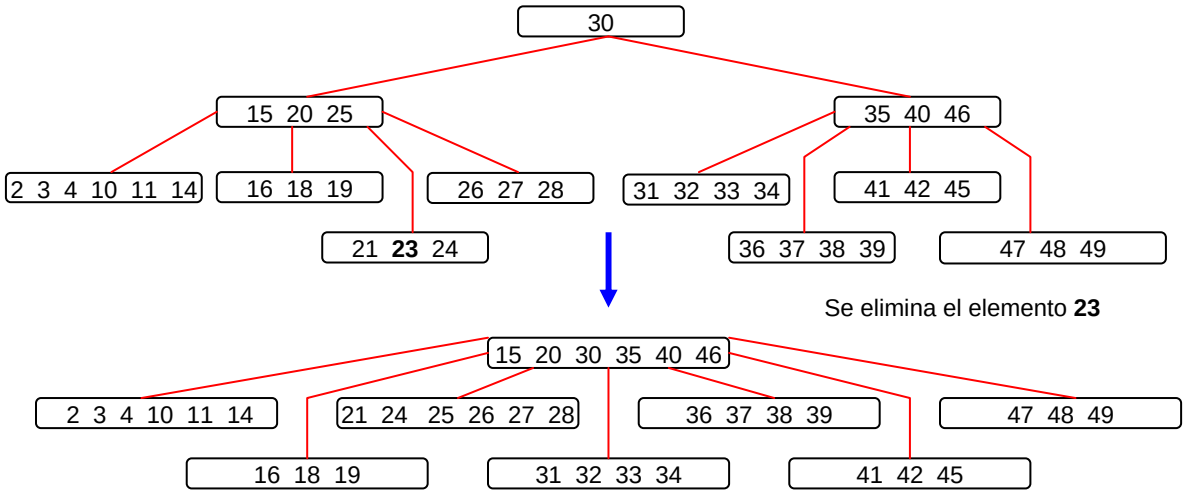


Figura 3.38: Si el árbol es de **orden 3**, la eliminación del **elemento 23** provoca la disminución de la altura del árbol, ya que se unen los nodos del nivel inferior y los nodos padres, generando una nueva raíz.

Análisis general de comportamiento de los árboles B

El análisis de la eficiencia de un **árbol B** siempre ha sido un importante tema de estudio. La capacidad de almacenamiento de los nodos, en conjunto con la cantidad probable de elementos por insertar, determina la eficiencia de la búsqueda (altura). Por lo tanto, es importante conocer cómo se afecta la eficiencia al hacer variaciones en estos parámetros.

Primero se analiza cómo se comporta el **árbol B** cuando se utilizan los nodos a su mínima capacidad:

¿Cuántos elementos mínimo se guardan en el **nivel 0**? **1**; por lo tanto, hay **2** hijos.

¿Cuántos elementos mínimos se guardan en el **nivel 1**? **n** en cada nodo, **2n** en total; por lo tanto, hay **2(n + 1)** hijos.

¿Cuántos elementos, mínimo se guardan en el **nivel 2**? **n** en cada nodo, **2n(n + 1)** en total; por lo tanto, hay **2(n + 1)(n + 1)** hijos.

¿Cuántos elementos mínimos se guardan en el **nivel 3**? **n** en cada nodo, $\mathbf{2n(n + 1)^2}$ en total; por lo tanto, hay $\mathbf{2(n + 1)^2 (n + 1)}$ hijos.

En general, ¿cuántos elementos mínimos se guardan en el **nivel k**? **n** en cada nodo, $\mathbf{2n(n + 1)^{k-1}}$ en total; por lo tanto, hay $\mathbf{2(n + 1)^k (n + 1)}$ hijos, si no es el último nivel.

Con estas fórmulas, se pueden resolver preguntas como:

- ¿Cuál es el **número mínimo de elementos** o de **nodos** que tiene un **árbol B** de cierta altura **h**?
- ¿Cuál es la **altura máxima** que tiene un árbol con cualquier cantidad de elementos?

Ahora, se analiza cómo se comporta el **árbol B** cuando se utilizan los nodos a su máxima capacidad:

¿Cuántos elementos máximo se guardan en el **nivel 0**? **2n**, por lo tanto, hay **2n + 1** hijos.

¿Cuántos elementos máximos se guardan en el **nivel 1**? **2n** en cada nodo, **2n(2n + 1)** en total; por lo tanto, hay **(2n + 1)(2n + 1)** hijos.

¿Cuántos elementos máximos se guardan en el **nivel 2**? **2n** en cada nodo, $\mathbf{2n(2n + 1)^2}$ en total; por lo tanto, hay $\mathbf{(2n + 1)^2}$ **(2n + 1)** hijos.

¿Cuántos elementos máximos se guardan en el **nivel 3**? **2n** en cada nodo, $\mathbf{2n(2n + 1)^3}$ en total; por lo tanto, hay $\mathbf{(2n + 1)^3}$ **(2n + 1)** hijos.

En general, ¿cuántos elementos máximos se guardan en el **nivel k**? **2n** en cada nodo, $\mathbf{2n(2n + 1)^k}$ en total; por lo tanto, hay $\mathbf{(2n + 1)^{k+1}}$ hijos, si no es el último nivel.

3.4.1 Ventajas de un árbol B

Puesto que los nodos de un **árbol B** están en memoria secundaria, se busca minimizar los accesos que se hagan a disco porque los tiempos de ejecución son muy costosos. La altura que tenga un **árbol B** determinará cuántos accesos al disco (en el peor de los casos), se tendrían que realizar en alguna operación sobre el árbol.

La tabla de la **Tabla 3.4** muestra la altura correspondiente al **árbol B** para diversos órdenes de un árbol y diversas cantidades de elementos:

ORDEN	NÚMERO DE ELEMENTOS					
	10^4	10^5	10^6	10^7	10^8	10^9
16	4	5	5	6	7	8
32	3	4	4	5	6	6
64	3	3	4	4	5	5

Tabla 3.4: Altura correspondiente al **árbol B** para diversos órdenes de un árbol y diversas cantidades de elementos.

Estos resultados demuestran el beneficio de almacenar datos en un **árbol B**. Por ejemplo, para almacenar un billón de elementos en un **árbol B** de **orden 16**, la altura del árbol cuando mucho será de **8** (ocho accesos a disco); sin embargo, un árbol de búsqueda binaria requiere una altura de **29 niveles** para guardar **999,999,999** elementos en el caso ideal y con las implicaciones correspondientes a su implantación en memoria secundaria.

Si los nodos se ven como páginas de memoria secundaria (disco), esta estructura es útil en la administración del nivel físico de una base de datos. Existen algunas variantes sobre estos árboles, como los **árboles B ***, **B+** y **B+ link**, que actualmente se emplean en la implementación de **manejadores de bases de datos**.

Implementación de un árbol B en lenguaje C

El **Programa 3.4** muestra como operan los **árboles B** sobre operaciones de un archivo en disco. Se pueden observar las operaciones básicas con números enteros, como son `insertar`, `suprimir` y `buscar`.

```
/* Librerías                                                          */
#include "stdio.h"
#include "stdlib.h"
#include "ctype.h"
#include "io.h"
#include "fcntl.h"
#include "sys/stat.h"

/* Macros                                                             */
#define M         2
#define MAX       4
#define NULO      (-1L)

/* Enumeración de Eventos                                             */
typedef enum {
      Insercion_no_completa, Suceso, Dato_duplicado,
      Sobreflujo, Indefinido
} estado;

/* Estructura de los nodos                                            */
typedef struct{
int contador;
int dato[MAX];
long siguiente[MAX + 1];
} nodo;

/* Variables globales                                                 */
nodo nodo_raiz;
long inicio[2], raiz = NULO, Limpiar_lista = NULO;
FILE *arch_arbol;

/* Prototipos de Función                                              */
void error(char *cadena);
void leer_nodo(long t, nodo *pnodo);
void escribir_nodo(long t, nodo *pnodo);
long cargar_nodo(void);
void encontrado(long t,  int i);
void no_encontrado(int x);
int busqueda_binaria(int x, int *a, int n);
estado buscar(int x);
estado ins(int x, long t, int *y, long *u);
estado insertar(int x);
void liberar_nodo(long t);
void lee_inicio(void);
void esc_inicio(void);
estado sup(int x, long t);
estado suprimir_nodo(int x);
void ver_arbol(long t);

/* Programa Principal                                                 */
main(){
int x, codigo = 0;
char op, arbol_nomb[51], nomb[51];
FILE *fp;
clrscr();
```

Programa 3.4: Operaciones sobre un archivo en disco con **árboles B** (Parte **1**/9).

```
puts("\n\t\t\tÁrboles B");
printf("\n\tNombre del archivo para el árbol B:  ");
scanf("%50s",arbol_nomb);
arch_arbol = fopen(arbol_nomb, "r + b"); /* Archivo Binario           */
if (arch_arbol == NULL) {
      arch_arbol = fopen(arbol_nomb, "w + b");
      esc_inicio();
      }
else  {
      lee_inicio();
      ver_arbol(raiz);
      }
puts("\n\tSecuencia de enteros, termina con un / :");
while(scanf("%d", &x) == 1) {
      insertar(x);
      codigo = 1;
      }
if(codigo)  ver_arbol(raiz);
printf("\n\tLeer un entero de un archivo (S/N)?: ");
scanf("%c",&op);
while (getchar() != '\n');
if(tolower(op) == 's'){
      printf("\n\tEntra el Nombre para el archivo : ");
      scanf("%50s", nomb);
      if ((fp = fopen(nomb, "r")) == NULL)
            error("Archivo no disponible");
      while (fscanf(fp, "%d", &x) == 1) insertar(x);
      fclose(fp);
      ver_arbol(raiz);
      }
for ( ; ; ){
      printf("\n\tEntra un Entero : ");
      codigo = scanf("%d", &x);
      printf("\n\tPresione: I=Insertar, S=Suprimir, B=Buscar ó
                  Q=Salir : ");
      scanf(" %c", &op);
      op = toupper(op);
      if (codigo)
      switch (op) {
            case 'I': if (insertar(x) == Suceso)
                        ver_arbol(raiz);
                  break;
            case 'S': if (suprimir_nodo(x) == Suceso)
                        ver_arbol(raiz);
                  else  puts("No encontrado");
                  break;
            case 'B': if (buscar(x) == Indefinido)
                        puts("No encontrado");
                  break;
            }
      else    if (op == 'Q') break;
      }
esc_inicio();
fclose(arch_arbol);
return(0);
}
```

Programa 3.4: Operaciones sobre un archivo en disco con **árboles B** (Parte **2**/9).

```
/* Imprime Error                                                    */
void error(char *cadena){
printf("\nError: %s\n", cadena);
exit(1);
}

/* Lee un nodo                                                      */
void leer_nodo(long t, nodo *pnodo){
if (t == raiz){
      *pnodo = nodo_raiz;
      return;
      }
if (fseek(arch_arbol, t, SEEK_SET))
      error("ERROR en Leer");
if (fread(pnodo, sizeof(nodo), 1, arch_arbol) == 0)
      error("ERROR en Leer");
}

/* Escribe un nodo                                                  */
void escribir_nodo(long t, nodo *pnodo){
if (t == raiz) nodo_raiz = *pnodo;
if (fseek(arch_arbol, t, SEEK_SET))
      error("ERROR en Escribir");
if (fwrite(pnodo, sizeof(nodo), 1, arch_arbol) == 0)
      error("ERROR en Escribir");
}

/* Carga un nodo                                                    */
long cargar_nodo(void){
long t;
nodo nod;
if (Limpiar_lista == NULO) {
      if (fseek(arch_arbol, 0L, SEEK_END))
            error("ERROR en Cargar");
      t = ftell(arch_arbol);
      escribir_nodo(t, &nod);
      }
else  {
      t = Limpiar_lista;
      leer_nodo(t, &nod);
      Limpiar_lista = nod.siguiente[0];
      }
return(t);
}

/* Busca el contenido de un nodo                                    */
void encontrado(long t,  int i){
nodo nod;
printf("\tEncontrado en el nodo con la posición %d, contenido :  ", i);
leer_nodo(t, &nod);
for (i=0; i < nod.contador; i++)
      printf("  %d", nod.dato[i]);
}
```

Programa 3.4: Operaciones sobre un archivo en disco con **árboles B** (Parte **3**/9).

```
/* Sino se encuentra                                                    */
void no_encontrado(int x){
printf("\tDato %d no encontrado\n", x);
}

/* Búsqueda Binaria                                                     */
int busqueda_binaria(int x, int *a, int n){
int i, izquierda, derecha;
if (x <= a[0]) return 0;
if (x > a[n - 1]) return n;
izquierda = 0;
derecha = n - 1;
while (derecha -  izquierda > 1){
      i = (derecha + izquierda)/2;
      if (x <= a[i]) derecha = i;
      else izquierda = i;
      }
return(derecha);
}

/* Busca un dato en el árbol B                                          */
estado buscar(int x){
int i, j,*k, n;
nodo nod;
long t = raiz;
puts("\tRuta de la búsqueda:");
while (t != NULO){
      leer_nodo(t, &nod);
      k = nod.dato;
      n = nod.contador;
      for (j=0; j < n; j++) printf("  %d", k[j]);
      puts("");
      i = busqueda_binaria(x, k, n);
      if (i < n && x == k[i]){
            encontrado(t,i);
            return(Suceso);
            }
      t = nod.siguiente[i];
      }
return(Indefinido);
}

/* Inserta x en la raíz del árbol B                                     */
estado ins(int x, long t, int *y, long *u){
long p_nuevo, p_final, *p;
int i, j, *n, k_final, *k, x_nuevo;
estado codigo;
nodo nod, nodo_nuevo;
if (t == NULO){
      *u = NULO;
      *y = x;
      return(Insercion_no_completa);
      }
leer_nodo(t, &nod);
```

Programa 3.4: Operaciones sobre un archivo en disco con **árboles B** (Parte **4**/9).

```
n = & nod.contador;
k = nod.dato;
p = nod.siguiente;
i = busqueda_binaria(x, k, *n);
if (i < *n && x == k[i])
      return(Dato_duplicado);
codigo = ins(x, p[i], &x_nuevo, &p_nuevo);
if (codigo != Insercion_no_completa)
      return codigo;
if (*n < MAX){
      i = busqueda_binaria(x_nuevo, k, *n);
      for (j = *n; j > i; j--){
            k[j] = k[j-1];
            p[j+1] = p[j];
            }
      k[i] = x_nuevo;
      p[i+1] = p_nuevo;
      ++*n;
      escribir_nodo(t, &nod);
      return(Suceso);
      }
if (i == MAX){
      k_final = x_nuevo;
      p_final = p_nuevo;
      }
else  {
      k_final = k[MAX - 1];
      p_final = p[MAX];
      for (j = MAX - 1; j > i; j--){
            k[j] = k[j - 1];
            p[j+1] = p[j];
            }
      k[i] = x_nuevo;
      p[i+1] = p_nuevo;
      }
*y = k[M];
*n = M;
*u = cargar_nodo();
nodo_nuevo.contador = M;
for (j = 0; j < M - 1; j++){
      nodo_nuevo.dato[j] = k[j + M + 1];
      nodo_nuevo.siguiente[j] = p[j + M + 1];
      }
nodo_nuevo.siguiente[M - 1] = p[MAX];
nodo_nuevo.dato[M - 1] = k_final;
nodo_nuevo.siguiente[M] = p_final;
escribir_nodo(t, &nod);
escribir_nodo(*u, &nodo_nuevo);
return(Insercion_no_completa);
}

/* Maneja las inserción con ins                                          */
estado insertar(int x){
long p_nuevo, u;
int x_nuevo;
estado codigo = ins(x, raiz, &x_nuevo, &p_nuevo);
```

Programa 3.4: Operaciones sobre un archivo en disco con **árboles B** (Parte **5**/9).

```
if (codigo == Dato_duplicado)
      printf("\tDato Duplicado: %d (se ignora) \n", x);
else if (codigo == Insercion_no_completa){
      u = cargar_nodo();
      nodo_raiz.contador = 1;
      nodo_raiz.dato[0] = x_nuevo;
      nodo_raiz.siguiente[0] = raiz;
      nodo_raiz.siguiente[1] = p_nuevo;
      raiz = u;
      escribir_nodo(u, &nodo_raiz);
      codigo = Suceso;
      }
return(codigo);
}

/* Libera un nodo                                                    */
void liberar_nodo(long t){
nodo nod;
leer_nodo(t, &nod);
nod.siguiente[0] = Limpiar_lista;
Limpiar_lista = t;
escribir_nodo(t, &nod);
}

/* Lee el inicio                                                     */
void lee_inicio(void){
if (fseek(arch_arbol, 0L, SEEK_SET))
      error("ERROR en Inicio 1");
if (fread(inicio, sizeof(long), 2, arch_arbol) == 0)
      error("ERROR en Inicio 2");
leer_nodo(inicio[0], &nodo_raiz);
raiz = inicio[0];
Limpiar_lista = inicio[1];
}

/* Escribe Inicio                                                    */
void esc_inicio(void){
inicio[0] = raiz;
inicio[1] = Limpiar_lista;
if (fseek(arch_arbol, 0L, SEEK_SET))
      error("ERROR en Inicio 3");
if (fwrite(inicio, sizeof(long), 2, arch_arbol) == 0)
      error("ERROR en Inicio 4");
if (raiz != NULO)
      escribir_nodo(raiz, &nodo_raiz);
}

/* Suprime x del árbol B                                             */
estado sup(int x, long t){
int i, j, *k, *n,*dato, *nizq, *nder,*ldat,*rdat,b=0,nq,*agr;
estado codigo;
long *p,izquierda,derecha,*lptr,*rptr,q,q1;
nodo nod,nod1,nod2,nodL,nodR;
if (t == NULO) return(Indefinido);
leer_nodo(t, &nod);
n = & nod.contador;
```

Programa 3.4: Operaciones sobre un archivo en disco con **árboles B** (Parte **6**/9).

```
k = nod.dato;
p = nod.siguiente;
i = busqueda_binaria(x, k, *n);
if (p[0] == NULO){
      if (i == *n || x < k[i])
            return Indefinido;
      for (j = i + 1; j < *n; j++){
            k[j-1] = k[j];
            p[j] = p[j + 1];
            }
      --*n;
      escribir_nodo(t, &nod);
      return(*n >= (t == raiz ? 1 : M) ? Suceso: Sobreflujo);
      }
dato = k + i;
izquierda = p[i];
leer_nodo(izquierda, &nod1);
nizq = & nod1.contador;
if (i < *n && x == *dato){
      q = p[i];
      leer_nodo(q, &nod1);
      nq = nod1.contador;
      while (q1 = nod1.siguiente[nq], q1 != NULO){
            q = q1;
            leer_nodo(q, &nod1);
            nq = nod1.contador;
            }
      agr = nod1.dato + nq - 1;
      *dato = *agr;
      *agr = x;
      escribir_nodo(t, &nod);
      escribir_nodo(q, &nod1);
      }
codigo = sup(x, izquierda);
if (codigo != Sobreflujo) return codigo;
if (i < *n) leer_nodo(p[i+1], &nodR);
if (i == *n || nodR.contador == M){
      if (i > 0){
            leer_nodo(p[i - 1], &nodL);
            if (i == *n || nodL.contador > M) b = 1;
            }
      }
if (b)      {
      dato = k + i - 1;
      izquierda = p[i - 1];
      derecha = p[i];
      nod1 = nodL;
      leer_nodo(derecha, &nod2);
      nizq = & nod1.contador;
      }
else  {
      derecha = p[i + 1];
      leer_nodo(izquierda, &nod1);
      nod2 = nodR;
      }
```

Programa 3.4: Operaciones sobre un archivo en disco con **árboles B** (Parte **7**/9).

```
nder = & nod2.contador;
ldat = nod1.dato;
rdat = nod2.dato;
lptr = nod1.siguiente;
rptr = nod2.siguiente;
if (b)      {
      rptr[*nder + 1] = rptr[*nder];
      for (j = *nder; j>0; j--){
            rdat[j] = rdat[j - 1];
            rptr[j] = rptr[j - 1];
            }
      ++*nder;
      rdat[0] = *dato;
      rptr[0] = lptr[*nizq];
      *dato = ldat[*nizq - 1];
      if (--*nizq >= M){
            escribir_nodo(t, &nod);
            escribir_nodo(izquierda, &nod1);
            escribir_nodo(derecha, &nod2);
            return Suceso;
            }
      }
else if (*nder > M){
       ldat[M-1] = *dato;
       lptr[M] = rptr[0];
       *dato = rdat[0];
       ++*nizq;
       --*nder;
       for (j = 0; j < *nder; j++){
            rptr[j] = rptr[j + 1];
            rdat[j] = rdat[j + 1];
            }
       rptr[*nder] = rptr[*nder + 1];
       escribir_nodo(t, &nod);
       escribir_nodo(izquierda, &nod1);
       escribir_nodo(derecha, &nod2);
       return(Suceso);
       }
ldat[M-1] = *dato;
lptr[M] = rptr[0];
for (j = 0; j < M; j++){
      ldat[M + j] = rdat[j];
      lptr[M + j + 1] = rptr[j + 1];
      }
*nizq = MAX;
liberar_nodo(derecha);
for (j = i + 1; j < *n; j++){
      k[j - 1] = k[j];
      p[j] = p[j + 1];
      }
--*n;
escribir_nodo(t, &nod);
escribir_nodo(izquierda, &nod1);
return( *n >= (t == raiz ? 1 : M) ? Suceso : Sobreflujo);
}
```

Programa 3.4: Operaciones sobre un archivo en disco con **árboles B** (Parte **8**/9).

```
/* Suprime a través de sup                                              */
estado suprimir_nodo(int x){
long raiz1;
estado codigo = sup(x, raiz);
if (codigo == Sobreflujo){
      raiz1 = nodo_raiz.siguiente[0];
      liberar_nodo(raiz);
      if (raiz1 != NULO)
            leer_nodo(raiz1, &nodo_raiz);
      raiz = raiz1;
      codigo = Suceso;
      }
return(codigo);
}

/* Imprime el árbol B                                                   */
void ver_arbol(long t){
static int pos = 0;
int i, *k, n;
nodo nod;
if (t != NULO){
      pos += 6;
      leer_nodo(t, &nod);
      k = nod.dato;
      n = nod.contador;
      printf("%*s", pos, "");
      for (i = 0; i < n; i++)printf(" %d", k[i]);
      puts("");   /* Limpia Buffer */
      for (i = 0; i <= n; i++) ver_arbol(nod.siguiente[i]);
      pos -= 6;
      }
}
```

Programa 3.4: Operaciones sobre un archivo en disco con **árboles B** (Parte **9**/9).

En la **Figura 3.39**, se observan los resultados después de ejecutar el **Programa 3.4**.

```
                        Árboles B
Nombre del archivo para el Árbol B: prueba.dat
Secuencia de enteros, termina con un /:
2 3 7 9 11 14 15 18 20 24 25 27 32 34 35 40 43 45 50 55 56 58 61 63 67 70
72 75 85 /
       20 45
             7 14
                   2 3
                   9 11
                   15 18
             27 35
                   24 25
                   32 34
                   40 43
             56 63 72
                   50 55
                   58 61
                   67 70
                   75 85
Entra un Entero: 67
Presione: I=Insertar, S=Suprimir, B=Buscar ó Q=Salir: B
Ruta de la búsqueda:
  20  45
  56  63  72
  67  70
Encontrado en el nodo con la posición 0, contenido:    67  70
Entra un Entero: 85
Presione: I=Insertar, S=Suprimir, B=Buscar ó Q=Salir: B
Ruta de la búsqueda:
  20  45
  56  63  72
  75  85
Encontrado en el nodo con la posición 1, contenido:    75  85
Entra un Entero: 150
Presione: I=Insertar, S=Suprimir, B=Buscar ó Q=Salir: Q
```

Figura 4.17: Operaciones sobre un archivo en disco con **árboles B**.

3.5 Árboles N-arios

Los mismos conceptos empleados en árboles binarios, pueden ser extendidos a árboles **N-arios**, en donde **N** representa el número máximo de hijos en cada nodo. Así un **árbol binario** sería un **árbol 2-ario**.

La **cantidad de nodos** (**n**) de un **árbol N-ario** completo de **altura d** se obtiene sumando todos los hijos de cada nivel, es decir en cada nivel hay $\mathbf{N^d}$ hijos, por lo cuál:

$$n = \sum_{i=0}^{d} N^i = \frac{N^{d+1} - 1}{N - 1}$$

En los **árboles N-arios**, cada nodo tiene un número finito de estructuras de árboles disjuntas asociadas al tipo de nodo, cada uno llamado subárbol. Cada elemento en un árbol se denomina **nodo** del árbol (**Figura 3.40**).

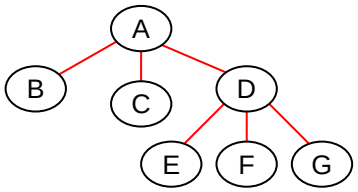


Figura 3.40: Árbol **N-ario**.

Un nodo sin subárboles se llama **hoja** (**Figura 3.40: B, C, E, F** y **G**). Los demás términos son empleados en el mismo sentido que en los árboles binarios como padre, hijo, hermano, ancestro, descendiente, etc. El **grado de un nodo** es el número de hijos. Por ejemplo de la **Figura 3.40 B** y **C** tienen un **grado 0** y son hojas mientras que **A** y **D** tienen un **grado 3**.

Un **árbol ordenado** se define como un árbol en el que los subárboles de cada nodo forman un conjunto ordenado.

El **primer hijo** de un nodo se denomina el **hijo más viejo** y el **último el hijo**, **hijo más joven**.

Un **bosque** es un conjunto ordenado de árboles ordenados.

Los recorridos en este tipo de árboles, se realizan de forma similar a los que se realizan en los árboles binarios, y son iguales a los recorridos que es posible realizar en un bosque y se pueden definir de la siguiente forma:

Recorrido en Preorden

- Visitar la raíz del primer árbol del bosque.
- Recorrer en preorden el bosque formado por los subárboles del primer árbol, en caso de que lo haya.
- Recorrer en preorden el bosque formado por los árboles restantes del bosque, en caso de que los haya.

Recorrido en orden

- Recorrer en orden el bosque formado por los subárboles del primer árbol del bosque, en caso de que lo haya.
- Visitar la raíz del primer árbol.
- Recorrer en orden el bosque formado por los árboles restantes del bosque, en caso de que los haya.

Recorrido en Posorden

- Recorrer en posorden el bosque formado por los subárboles del primer árbol del bosque, en caso de que lo haya.
- Recorrer en posorden el bosque formado por los árboles restantes del bosque, en caso de que los haya.
- Visitar la raíz del primer árbol del bosque.

Unidad 4: Desarrollo de Aplicaciones

4.1 Planificación de un proyecto de sistemas

Es el proceso de gestión para la creación de un **sistema** o **software**, la cual encierra un conjunto de actividades, una de las cuales es la estimación, estimar es echar un vistazo al futuro y aceptamos resignados cierto grado de incertidumbre. Aunque la estimación, es mas un arte que una ciencia, es una actividad importante que no debe llevarse a cabo de forma descuidada. Existen técnicas útiles para la estimación de costos de tiempo. Y dado que la estimación es la base de todas las demás actividades de planificación del proyecto y sirve como guía para una buena Ingeniería sistemas y software.

Al estimar tomamos en cuenta no solo del procedimiento técnico a utilizar en el proyecto, sino que se toma en cuenta los recursos, costos y planificación. El tamaño del proyecto es otro factor importante que puede afectar la precisión de las estimaciones. A medida que el tamaño aumenta, crece rápidamente la interdependencia entre varios elementos del software.

La disponibilidad de información Histórica es otro elemento que determina el riesgo de la estimación.

4.1.1 Objetivos de la Planificación del Proyecto

El objetivo de la planificación del proyecto de software es proporcionar un marco de trabajo que permita al gestor hacer estimaciones razonables de recursos costos y planificación temporal. Estas estimaciones se hacen dentro de un marco de tiempo limitado al comienzo de un proyecto de software, y deberían actualizarse regularmente medida que progresa el proyecto. Además las estimaciones deberían definir los escenarios del mejor caso, y peor caso, de modo que los resultados del proyecto pueden limitarse.

El objetivo de la planificación se logra mediante un proceso de descubrimiento de la información que lleve a estimaciones razonables.

4.1.2 Actividades asociadas al proyecto de software

4.1.2.1 Ámbito del Software

Es la primera actividad de llevada a cabo durante la planificación del proyecto de software.

En esta etapa se deben evaluar la función y el rendimiento que se asignaron al software durante la ingeniería del sistema de computadora para establecer un ámbito de proyecto que no sea ambiguo, e incomprensible para directivos y técnicos

Describe la función, el rendimiento, las restricciones, las interfaces y la fiabilidad, se evalúan las funciones del ámbito y en algunos casos se refinan para dar mas detalles antes del comienzo de la estimación. Las restricciones de rendimiento abarcan los requisitos de tiempo de respuesta y procesamiento, identifican los límites del software originados por el hardware externo, por la memoria disponible y por otros sistemas existentes.

El ámbito se define como un pre-requisito para la estimación y existen algunos elementos que se debe tomar en cuenta como es:

La Obtención de la Información necesaria para el software. Para esto el analista y el cliente se reúnen sobre las expectativas del proyecto y se ponen de acuerdo en los puntos de interés para su desarrollo.

4.1.2.2 Recursos

La segunda tarea de la planificación del desarrollo de software es la estimación de los recursos requeridos para acometer el esfuerzo de desarrollo de software, esto simula a una pirámide donde las herramientas (hardware y software), son la base proporciona la infraestructura de soporte al esfuerzo de desarrollo, en segundo nivel de la pirámide se encuentran los componentes reutilizables.
Y en la parte mas alta de la pirámide se encuentra el recurso primario, las personas (el recurso humano).

Cada recurso queda especificado mediante cuatro características:

- Descripción del Recurso.
- Informes de disponibilidad.
- Fecha cronológica en la que se requiere el recurso.
- Tiempo durante el que será aplicado el recurso

Recursos Humanos

La cantidad de personas requeridas para el desarrollo de un proyecto de software solo puede ser determinado después de hacer una estimación del esfuerzo de desarrollo (por ejemplo personas mes o personas años), y seleccionar la posición dentro de la organización y la especialidad que desempeñara cada profesional.

Recursos o componentes de software reutilizables

Cualquier estudio sobre recursos de software estaría incompleto sin estudiar la reutilización, esto es la creación y la reutilización de bloques de construcción de software.

Tales bloques se deben establecer en catálogos para una consulta más fácil, estandarizarse para una fácil aplicación y validarse para la también fácil integración.

Se sugieren cuatro categorías de recursos de software que se deberían tener en cuenta a medida que se avanza con la planificación:

- Componentes ya desarrollados.
- Componentes ya experimentados.
- Componentes con experiencia Parcial.
- Componentes nuevos.

4.1.3 Recursos de entorno

El entorno es donde se apoya el proyecto de software, llamado a menudo entorno de Ingeniería de software, incorpora hardware y software.

El hardware proporciona una plataforma con las herramientas (software) requeridas para producir los productos que son el resultado de la buena practica de la Ingeniería del Software, un planificador de proyectos debe determinar la ventana temporal requerida para el Hardware y el Software, y verificar que estos recursos estén disponibles. Muchas veces el desarrollo de las pruebas de validación de un proyecto de software para la composición automatizada puede necesitar un compositor de fotografías en algún punto durante el desarrollo. Cada elemento de hardware debe ser especificado por el planificador del Proyecto de Software.

4.1.4 Estimación del Proyecto de Software

En el principio el costo del Software constituía un pequeño porcentaje del costo total de los sistemas basados en computadoras. Hoy en día el Software es el elemento más caro de la mayoría de los sistemas informáticos.

Un gran error en la estimación del costo puede ser lo que marque la diferencia entre beneficios y perdidas, la estimación del costo y del esfuerzo del software nunca será una ciencia exacta, son demasiadas las variables: humanas, técnicas, de entorno, políticas, que pueden afectar el costo final del software y el esfuerzo aplicado para desarrollarlo.

Para realizar estimaciones seguras de costos y esfuerzos tienen varias opciones posibles:

- Deje la estimación para más adelante (obviamente podemos realizar una estimación al cien por cien fiable después de haber terminado el proyecto.
- Base las estimaciones en proyectos similares ya terminados.
- Utilice técnicas de descomposición relativamente sencillas para generar las estimaciones de costos y esfuerzo del proyecto.
- Desarrolle un modelo empírico para él calculo de costos y esfuerzos del Software.

Desdichadamente la primera opción, aunque atractiva no es práctica.

La Segunda opción puede funcionar razonablemente bien si el proyecto actual es bastante similar a los esfuerzos pasados y si otras influencias del proyecto son similares. Las opciones restantes son métodos viables para la estimación del proyecto de software. Desde el punto de vista ideal, se deben aplicar conjuntamente las técnicas indicadas usando cada una de ellas como comprobación de las otras.

Antes de hacer una estimación, el planificador del proyecto debe comprender el ámbito del software a construir y generar una estimación de su tamaño.

4.1.4.1 Estimación basada en el Proceso

Es la técnica más común para estimar un proyecto es basar la estimación en el proceso que se va a utilizar, es decir, el proceso se descompone en un conjunto relativamente pequeño de actividades o tareas, y en el esfuerzo requerido para llevar a cabo la estimación de cada tarea.

Al igual que las técnicas basadas en problemas, la estimación basada en el proceso comienza en una delineación de las funciones del software obtenidas a partir del ámbito del proyecto. Se mezclan las funciones del problema y las actividades del proceso. Como ultimo paso se calculan los costos y el esfuerzo de cada función y la actividad del proceso de software.

4.1.5 Diferentes Modelos de Estimación

Existen diferentes modelos de estimación como son:

Los Modelos Empíricos:

Donde los datos que soportan la mayoría de los modelos de estimación obtienen una muestra limitada de proyectos. Por está razón, el modelo de estimación no es adecuado para todas las clases de software y en todos los entornos de desarrollo. Por lo tanto los resultados obtenidos de dichos modelos se deben utilizar con prudencia.

El Modelo COCOMO.

Barry Boehm, en su libro clásico sobre economía de la Ingeniería del Software, introduce una jerarquía de modelos de estimación de Software con el nombre de **COCOMO**, por su nombre en Ingles (**Co**nstructive, **Co**st, **Mo**del) modelo constructivo de costos. La jerarquía de modelos de Boehm esta constituida por los siguientes:

- Modelo I. El Modelo COCOMO básico calcula el esfuerzo y el costo del desarrollo de Software en función del tamaño del programa, expresado en las líneas estimadas.
- Modelo II. El Modelo COCOMO intermedio calcula el esfuerzo del desarrollo de software en función del tamaño del programa y de un conjunto de conductores de costos que incluyen la evaluación subjetiva del producto, del hardware, del personal y de los atributos del proyecto.

- Modelo III. El modelo COCOMO avanzado incorpora todas las características de la versión intermedia y lleva a cabo una evaluación del impacto de los conductores de costos en cada caso (análisis, diseño, etc.) del proceso de ingeniería de Software.

Herramientas Automáticas de Estimación

Las herramientas automáticas de estimación permiten al planificador estimar costos y esfuerzos, así como llevar a cabo análisis del tipo, que pasa si, con importantes variables del proyecto, tales como la fecha de entrega o la selección del personal. Aunque existen muchas herramientas automáticas de estimación, todas exhiben las mismas características generales y todas requieren de una o más clases de datos.

A partir de estos datos, el modelo implementado por la herramienta automática de estimación proporciona estimaciones del esfuerzo requerido para llevar a cabo el proyecto, los costos, la carga de personal, la duración, y en algunos casos la planificación temporal de desarrollo y riesgos asociados.

En resumen el planificador del Proyecto de Software tiene que estimar tres cosas antes de que comience el proyecto: cuanto durara, cuanto esfuerzo requerirá y cuanta gente estará implicada. Además el planificador debe predecir los recursos de hardware y software que va a requerir y el riesgo implicado.

Para obtener estimaciones exactas para un proyecto, generalmente se utilizan al menos dos de las tres técnicas referidas anteriormente. Mediante la comparación y la conciliación de las estimaciones obtenidas con las diferentes técnicas, el planificador puede obtener una estimación más exacta. La estimación del proyecto de software nunca será una ciencia exacta, pero la combinación de buenos datos históricos y técnicas puede mejorar la precisión de la estimación.

4.2 Análisis de Sistemas de Computación

El **análisis** es un conjunto o disposición de procedimientos o programas relacionados de manera que juntos forman una sola unidad. Un conjunto de hechos, principios y reglas clasificadas y dispuestas de manera ordenada mostrando un plan lógico en la unión de las partes. Un método, plan o procedimiento de clasificación para hacer algo. También es un conjunto o arreglo de elementos para realizar un objetivo predefinido en el procesamiento de la Información. Esto se lleva a cabo teniendo en cuenta ciertos principios:

- Debe presentarse y entenderse el dominio de la información de un problema.
- Defina las funciones que debe realizar el Software.
- Represente el comportamiento del software a consecuencias de acontecimientos externos.
- Divida en forma jerárquica los modelos que representan la información, funciones y comportamiento.

El proceso debe partir desde la información esencial hasta el detalle de la Implementación.

La función del Análisis puede ser dar soporte a las actividades de un negocio, o desarrollar un producto que pueda venderse para generar beneficios. Para conseguir este objetivo, un Sistema basado en computadoras hace uso de seis (6) elementos fundamentales:

- Software, que son Programas de computadora, con estructuras de datos y su documentación que hacen efectiva la logística metodología o controles de requerimientos del Programa.
- Hardware, dispositivos electrónicos y electromecánicos, que proporcionan capacidad de cálculos y funciones rápidas, exactas y efectivas (Computadoras, Censores, maquinarias, bombas, lectores, etc.), que proporcionan una función externa dentro de los Sistemas.
- Personal, son los operadores o usuarios directos de las herramientas del Sistema.
- Base de Datos, una gran colección de informaciones organizadas y enlazadas al Sistema a las que se accede por medio del Software.
- Documentación, Manuales, formularios, y otra información descriptiva que detalla o da instrucciones sobre el empleo y operación del Programa.

- Procedimientos, o pasos que definen el uso especifico de cada uno de los elementos o componentes del Sistema y las reglas de su manejo y mantenimiento.

Un Análisis de Sistema se lleva a cabo teniendo en cuenta los siguientes objetivos en mente:

- Identifique las necesidades del Cliente.
- Evalúe que conceptos tiene el cliente del sistema para establecer su viabilidad.
- Realice un Análisis Técnico y económico.
- Asigne funciones al Hardware, Software, personal, base de datos, y otros elementos del Sistema.
- Establezca las restricciones de presupuestos y planificación temporal.
- Cree una definición del sistema que forme el fundamento de todo el trabajo de Ingeniería.

Para lograr estos objetivos se requiere tener un gran conocimiento y dominio del Hardware y el Software, así como de la Ingeniería humana (Manejo y Administración de personal), y administración de base de datos.

4.2.1 Objetivos del Análisis

Es el primer paso del análisis del sistema, en este proceso en Analista se reúne con el cliente y/o usuario (un representante institucional, departamental o cliente particular), e identifican las metas globales, se analizan las perspectivas del cliente, sus necesidades y requerimientos, sobre la planificación temporal y presupuestal, líneas de mercadeo y otros puntos que puedan ayudar a la identificación y desarrollo del proyecto.

Algunos autores suelen llamar a esta parte “Análisis de Requisitos” y lo dividen en cinco partes:

- Reconocimiento del problema.
- Evaluación y Síntesis.
- Modelado.
- Especificación.
- Revisión

Antes de su reunión con el analista, el cliente prepara un documento conceptual del proyecto, aunque es recomendable que este se elabore durante la comunicación Cliente – analista, ya que de hacerlo el cliente solo de todas maneras tendría que ser modificado, durante la identificación de las necesidades.

4.2.2 Estudio de Viabilidad

Muchas veces cuando se emprende el desarrollo de un proyecto de Sistemas los recursos y el tiempo no son realistas para su materialización sin tener pérdidas económicas y frustración profesional. La viabilidad y el análisis de riesgos están relacionados de muchas maneras, si el riesgo del proyecto es alto, la viabilidad de producir software de calidad se reduce, sin embargo se deben tomar en cuenta cuatro áreas principales de interés:

Una evaluación de los costos de desarrollo, comparados con los ingresos netos o beneficios obtenidos del producto o Sistema desarrollado.

Viabilidad económica.- Un estudio de funciones, rendimiento y restricciones que puedan afectar la realización de un sistema aceptable.

Viabilidad Técnica.

Viabilidad Legal.- Es determinar cualquier posibilidad de infracción, violación o responsabilidad legal en que se podría incurrir al desarrollar el Sistema.

Alternativas. Una evaluación de los enfoques alternativos del desarrollo del producto o Sistema.

El estudio de la viabilidad puede documentarse como un informe aparte para la alta gerencia.

El análisis económico incluye lo que llamamos, el análisis de costos – beneficios, significa una valoración de la inversión económica comparado con los beneficios que se obtendrán en la comercialización y utilidad del producto o sistema.

Muchas veces en el desarrollo de Sistemas de Computación estos son intangibles y resulta un poco dificultoso evaluarlo, esto varia de acuerdo a la características del Sistema. El análisis de costos – beneficios es una fase muy importante de ella depende la posibilidad de desarrollo del Proyecto.

En el Análisis Técnico, el Analista evalúa los principios técnicos del Sistema y al mismo tiempo recoge información adicional sobre el rendimiento, fiabilidad, características de mantenimiento y productividad.

Los resultados obtenidos del análisis técnico son la base para determinar sobre si continuar o abandonar el proyecto, si hay riesgos de que no funcione, no tenga el rendimiento deseado, o si las piezas no encajan perfectamente unas con otras.

Cuando queremos dar a entender mejor lo que vamos a construir en el caso de edificios, Herramientas, Aviones, Maquinas, se crea un modelo idéntico, pero en menor escala (mas pequeño).

Sin embargo cuando aquello que construiremos es un Software, nuestro modelo debe tomar una forma diferente, deben representar todas las funciones y subfunciones de un Sistema. Los modelos se concentran en lo que debe hacer el sistema no en como lo hace, estos modelos pueden incluir notación gráfica, información y comportamiento del Sistema.

Todos los Sistemas basados en computadoras pueden modelarse como transformación de la información empleando una arquitectura del tipo entrada y salida.

4.3 Diseño de Sistemas de Computación

El Diseño de Sistemas se define el proceso de aplicar ciertas técnicas y principios con el propósito de definir un dispositivo, un proceso o un Sistema, con suficientes detalles como para permitir su interpretación y realización física.

La etapa del Diseño del Sistema encierra cuatro etapas:

1. Trasforma el modelo de dominio de la información, creado durante el análisis, en las estructuras de datos necesarios para implementar el Software.

2. El diseño de los datos. Define la relación entre cada uno de los elementos estructurales del programa.

3. El Diseño Arquitectónico.Describe como se comunica el Software consigo mismo, con los sistemas que operan junto con el y con los operadores y usuarios que lo emplean.

4. El Diseño de la Interfaz.

4.4 Ejemplos de Aplicaciones con Estructuras de Datos

4.4.1 Balanceo de Delimitadores

Realizar un programa, en el cual se verifique el **balanceo de limitadores** (Paréntesis, Corchetes y Llaves) en una expresión algebraica.

El programa contempla una entrada del tipo: **`([{(a + b) * c} - d] / e)`**

El programa debe de asegurarse que:

- El símbolo que abre un ámbito, es del mismo tipo del que lo cierra
- Hay el mismo número de símbolos izquierdos y derechos correspondientes
- Todo símbolo derecho es precedido por su símbolo izquierdo correspondiente

La expresión es válida si se cumple que:

- El conteo de símbolos es cero
- El conteo de símbolos no es negativo
- Para cada símbolo que abre un ámbito el que lo cierra es del mismo tipo

Este ejemplo se resuelve mediante pilas, por que el último ámbito abierto, debe de ser el primero en cerrarse. Así, poner un elemento dentro de la pila, corresponde a abrir un ámbito y sacar un elemento de la pila, corresponde a cerrar un ámbito.

Para la expresión **`{[a / (b + c)] * c}`** tenemos que una pila se comporta como:

Pila	Entrada	Comentarios
Vacía	{	Se abre ámbito 1
{	[	Se abre ámbito 2
{ [	a /	No se agrega nada a la pila
{ [	(	Se abre ámbito 3
{ [(	b + c	No se agrega nada a la pila
{ [	)	Se cierra ámbito 3
{ [	* c	No se agrega nada a la pila
{ [	]	Se cierra ámbito 2
Vacía	}	Se cierra ámbito 1

El seudo código para resolver este problema se presenta a continuación:

```
Válida = 1              /* La expresión es válida                          */
Pila = Vacía            /* Pila vacía                                      */
Do      {               /* Mientras no se halla leído la expresión completa */
        If(símbolo == '(' || símbolo == '{' || símbolo == '[') push(pila, símbolo)
        If(símbolo == ')' || símbolo == '}' || símbolo == ']') {
                If(Pila == Vacía) Válida = 0
                Else    {
                        i = pop (pila)
                        if(Si i es diferente al símbolo de inicio) Válida = 0
                        }
        } While (Condición para terminar)
if(Válida == 1 && !pila_vacia)   printf("Expresión Válida")
else                             printf("Expresión NO Válida")
```

La implementación en lenguaje **C** se muestra en el **Programa 4.1**.

```
/* Librerías                                                          */
#include "stdio.h"
#include "stdlib.h"
#include "conio.h"
#define     Pila_Vacia  (pila == NULL)    /* Pila Vacía             */

typedef struct datos elemento;      /* Define tipo de elemento       */
struct datos{
char dato;                    /* Dato de tipo real                   */
elemento *anterior;           /* Apuntador al elemento anterior      */
};

/* Función de error */
void error(void){
perror("\n\aERROR: Memoria insuficiente...");
exit(1);
}

/* Crea un nuevo elemento                                             */
elemento *Nuevo(){
elemento *q = (elemento *)malloc(sizeof(elemento));
if(!q) error();
return(q);
}

/* Añade un elemento a la pila                                        */
void push(elemento **p,char x){
elemento *q,*tope;
tope = *p;                    /* Tope de la pila                     */
q = Nuevo();
q -> dato = x;
q -> anterior = tope;
tope = q;
*p = tope;
}

/* Recupera un elemento de la pila                                    */
char pop(elemento **p){
elemento *tope;
char x;
tope = *p;                    /* Tope de la pila                     */
if(tope == NULL){
      printf("\n\a\tError: POP = Pila Vacía...\n");
      return('0');
      }
x = tope -> dato;
*p = tope -> anterior;
free(tope);
return(x);
}
```

Programa 4.1: Verifica el Balanceo de Delimitadores de una expresión (Parte **1**/2).

```
/* Carga la expresión en un arreglo                                    */
void lee(char ent[]){
int pos = 0;
printf("\n\n\tEntra la Expresión Infija : ");
while((ent[pos++] = getchar()) != '\n');
ent[--pos] = '\0';
}

/* Verifica delimitadores                                              */
int verifica(char ent[]){
elemento *pila = NULL;                /* Pila Vacía                    */
int valido,pos = 0;
char op,a;
valido = 1;
while(ent[pos] != '\0'){
      op = ent[pos++];
      if(op == '(' || op == '{' ||  op == '[') push(&pila,op);
      if(op == ')' || op == '}' ||  op == ']') {
            if(Pila_Vacia)     valido = 0;
            else   {
                  a = pop(&pila);
                  if(op == ')' && a != '(') valido = 0;
                  if(op == '}' && a != '{') valido = 0;
                  if(op == ']' && a != '[') valido = 0;
                  }
            }
      }
if(!Pila_Vacia)     valido = 0;
return(valido);
}

/* Programa principal                                                  */
void main(void){
char ent[100];
clrscr();
printf("Verificador de Delimitadores.");
printf("Verifica los símbolos: (), [], y {}");
printf("en una expresión infija.");
lee(ent);
if(verifica(ent)) printf("Cadena Válida");
else  printf("Cadena NO Válida");
}
```

Programa 4.1: Verifica el Balanceo de Delimitadores de una expresión (Parte **2**/2).

Al ejecutar el programa anterior produce el resultado que se muestra en la **Figura 4.1**.

```
                Verificador de Delimitadores.
Verifica los símbolos: (), [], y {} en una expresión infija.
Entra la Expresión Infija: {[a / (b + c)] * c}
Cadena Válida

                Verificador de Delimitadores.
Verifica los símbolos: (), [], y {} en una expresión infija.
Entra la Expresión Infija: {[a / (b + c)] * c
Cadena NO Válida
```

Figura 4.1: Resultados del programa verificador de delimitadores.

4.4.2 Conversión de expresiones infijas a postfijas

Realizar un programa para realizar la **conversión** de una expresión algebraica de su forma **infija a postfija**. Este programa realiza la conversión de una expresión infija a postfija, para una entrada como **(a+b)*c** el programa realiza las siguientes operaciones sobre una pila.

Pila	Entrada	Postfija
(	(	Vacía
(	a	a
(+	+	a
(+	b	ab
Vacía	)	ab+
*	*	ab+
*	c	ab+c
Vacía	Vacío	ab+c*

Tabla de Precedencia de Operadores		
Símbolo	**Pila**	**Entrada**
(	0	3
*, /	2	2
+, -	1	1
Otro	-1	-1

La implementación en lenguaje **C** se muestra en el **Programa 4.2**.

```
#include "conio.h"
#include "ctype.h"
#define Pila_Vacia      (ps == NULL)/* Pila Vacía                        */
#define MAX             80          /* Numero máximo de elementos       */
#define TRUE            1           /* Verdadero                        */
#define FALSE           0           /* Falso                            */

/* Estructura de los elementos de la pila                                */
typedef struct datos elemento;      /* Define tipo de elemento          */
struct datos{
char dato;                          /* Dato de tipo real                */
elemento *anterior;                 /* Apuntador al elemento anterior   */
};

/* Función de error */
void error(void){
perror("\n\aERROR: Memoria insuficiente...");
exit(1);
}

/* Crea un nuevo elemento                                                */
elemento *Nuevo(){
elemento *q = (elemento *)malloc(sizeof(elemento));
if(!q) error();
return(q);
}

/* Añade un elemento a la pila                                           */
void push(elemento **ps,char x){
elemento *q,*tope;
tope = *ps;                         /* Tope de la pila                  */
q = Nuevo();
q -> dato = x;
q -> anterior = tope;
tope = q;
*ps = tope;
}
```

Programa 4.2: Conversión **Infija-Postfija** (Parte **1**/3).

```
/* Recupera un elemento de la pila */
char pop(elemento **ps){
elemento *tope;
char x;
tope = *ps;                          /* Tope de la pila                  */
if(Pila_Vacia){
      printf("\n\a\tError: POP = Pila Vacía...\n");
      return(NULL);
      }
x = tope -> dato;
*ps = tope -> anterior;
free(tope);
return(x);
}

/* Carga la expresión en un arreglo                                       */
void lee(char ent[]){
int pos = 0;
printf("\n\n\tEntra la Expresión Infija : ");
while((ent[pos++] = getchar()) != '\n');
ent[--pos] = '\0';
}

/* Reglas de precedencia                                                  */
/* op1 = Símbolo de la Pila                                               */
/* op2 = Símbolo de Entrada                                               */
int precedencia(char op1,char op2){
int ent,cima;
if(op1 == '\0' || op2 == '\0') return(FALSE);
cima = -1;
if(op1 == '(') cima = 0;
if(op1 == '*' || op1 == '/') cima = 2;
if(op1 == '+' || op1 == '-') cima = 1;
ent = -1;
if(op2 == '(') ent = 3;
if(op2 == '*' || op2 == '/') ent = 2;
if(op2 == '+' || op2 == '-') ent = 1;
if(cima >= ent) return(TRUE);
else return(FALSE);
}

/* Realiza la conversión infija a postfija                                */
void postfija(char infija[],char postr[]){
int posicion,actpos,aux;
char sim_tope,simbolo;
elemento *ps = NULL;                 /* Pila Vacía                       */
actpos = 0;
for(posicion = 0;infija[posicion] != '\0'; posicion++){
      simbolo = infija[posicion];
      if (isdigit(simbolo)) postr[actpos++] = simbolo;
      else  {
            aux = 0;
            sim_tope = '\0';
            if(!Pila_Vacia) sim_tope = pop(&ps);
```

Programa 4.2: Conversión **Infija-Postfija** (Parte **2**/3).

```
            while(!Pila_Vacia && precedencia(sim_tope,simbolo)){
                  if(sim_tope != '(' && sim_tope != ')')
                        postr[actpos++] = sim_tope;
                  sim_tope = pop(&ps);
                  aux = 1;
                  }
            if(aux == 0 || !Pila_Vacia) push(&ps,sim_tope);
            if(Pila_Vacia || simbolo != ')') push(&ps,simbolo);
            else sim_tope = pop(&ps);
            }
      }
while(!Pila_Vacia){
      sim_tope = pop(&ps);
      if(sim_tope != '(' && sim_tope != ')') postr[actpos++] = sim_tope;
      }
postr[actpos]='\0';
}

/* Programa principal                                                    */
void main(){
char infija[MAX],postr[MAX];
clrscr();
printf("Conversión Infija-Postfija.");
printf("Convierte expresiones de la forma 1*(2+3)+4/5 en 123+*45/+");
printf("Al terminar la expresión, presione : ENTER");
lee(infija);
printf("\n\n\tExpresión Infija   =   %s\n",infija);
postfija(infija,postr);
printf("\n\n\tExpresión Postfija =   %s\n",postr);
}
```

Programa 4.2: Conversión **Infija-Postfija** (Parte **3**/3).

Al ejecutar el programa anterior produce el resultado que se muestra en la **Figura 4.2**.

```
               Conversión Infija-Postfija.
Convierte expresiones de la forma 1*(2+3)+4/5 en 123+*45/+
Al terminar la expresión, presione: ENTER
Entra la Expresión Infija: ((7+3)*9)/2+4
Expresión Infija   =   ((7+3)*9)/2+4
Expresión Postfija =   73+9*2/4+
```

Figura 4.2: Resultados del programa para conversión Infija-Postfija.

4.4.3 Evaluación de expresiones infijas

Realizar un programa para evaluar expresiones **infijas**, para operadores ***, /, +** y **-**, y que considere los paréntesis como delimitadores. Los operandos deben ser de un sólo dígito.

La realización de este programa, consiste en juntar los dos programas anteriormente descritos. Se requieren dos versiones diferentes de las rutinas de manipulación de **pila**: `estructura, push, pop` y `elemento nuevo`. El **Programa 4.3** empleado para resolver este problema, se muestra a continuación.

```
#include "conio.h"
#include "ctype.h"
#include "math.h"

#define Pila_Vacia      (ps2 == NULL)     /* Pila Vacía                 */
#define Pila_Vacia1     (ps1 == NULL)     /* Pila Vacía                 */
#define MAX             80                /* Numero máximo de elementos */
#define TRUE            1                 /* Verdadero                  */
#define FALSE           0                 /* Falso                      */

/* Estructura de los elementos de la pila                               */
typedef struct datos elemento;      /* Define tipo de elemento          */
struct datos{
char dato;                          /* Dato de tipo real                */
elemento *anterior;                 /* Apuntador al elemento anterior   */
};

typedef struct datos1 elemento1;    /* Define tipo de elemento          */
struct datos1{
float dato;                         /* Dato de tipo real                */
elemento1 *anterior;                /* Apuntador al elemento anterior   */
};

/* Función de error                                                     */
void error(void){
perror("\n\aERROR: Memoria insuficiente...");
exit(1);
}

/* Crea un nuevo elemento                                               */
elemento1 *Nuevo1(){
elemento1 *q = (elemento1 *)malloc(sizeof(elemento1));
if(!q) error();
return(q);
}

/* Añade un elemento a la pila                                          */
void push1(elemento1 **p,float x){
elemento1 *q,*ps1;
ps1 = *p;                           /* Tope de la pila                  */
q = Nuevo1();
q -> dato = x;
q -> anterior = ps1;
ps1 = q;
*p = ps1;
}

/* Recupera un elemento de la pila                                      */
float pop1(elemento1 **p){
elemento1 *ps1;
float x;
ps1 = *p;                           /* Tope de la pila                  */
```

Programa 4.3: Evaluación de expresiones **infijas** (Parte **1**/4).

```
if(Pila_Vacia1){
      printf("\n\a\tError: POP = Pila Vacía...\n");
      return(0);
      }
x = ps1 -> dato;
*p = ps1 -> anterior;
free(ps1);
return(x);
}

/* Crea un nuevo elemento                                          */
elemento *Nuevo(){
elemento *q = (elemento *)malloc(sizeof(elemento));
if(!q) error();
return(q);
}

/* Añade un elemento a la pila                                     */
void push(elemento **ps2,char x){
elemento *q,*tope;
tope = *ps2;                        /* Tope de la pila             */
q = Nuevo();
q -> dato = x;
q -> anterior = tope;
tope = q;
*ps2 = tope;
}

/* Recupera un elemento de la pila                                 */
char pop(elemento **ps2){
elemento *tope;
char x;
tope = *ps2;                        /* Tope de la pila             */
if(Pila_Vacia){
      printf("\n\a\tError: POP = Pila Vacía...\n");
      return(NULL);
      }
x = tope -> dato;
*ps2 = tope -> anterior;
free(tope);
return(x);
}

/* Carga la expresión en un arreglo                                */
void lee(char ent[]){
int pos = 0;
printf("\n\n\tEntra la Expresión Infija : ");
while((ent[pos++] = getchar()) != '\n');
ent[--pos] = '\0';
}
```

Programa 4.3: Evaluación de expresiones **infijas** (Parte **2**/4).

```
/* Reglas de precedencia                                                */
/* op1 = Símbolo de la Pila                                             */
/* op2 = Símbolo de Entrada                                             */
int precedencia(char op1,char op2){
int ent,cima;
if(op1 == '\0' || op2 == '\0') return(FALSE);
cima = -1;
if(op1 == '(') cima = 0;
if(op1 == '*' || op1 == '/') cima = 2;
if(op1 == '+' || op1 == '-') cima = 1;
ent = -1;
if(op2 == '(') ent = 3;
if(op2 == '*' || op2 == '/') ent = 2;
if(op2 == '+' || op2 == '-') ent = 1;
if(cima >= ent) return(TRUE);
else return(FALSE);
}

/* Realiza la conversión infija a postfija                              */
void postfija(char infija[],char postr[]){
int posicion,actpos,aux;
char sim_tope,simbolo;
elemento *ps2 = NULL;              /* Pila Vacía                        */
actpos = 0;
for(posicion = 0;infija[posicion] != '\0'; posicion++){
      simbolo = infija[posicion];
      if (isdigit(simbolo)) postr[actpos++] = simbolo;
      else  {
            aux = 0;
            sim_tope = '\0';
            if(!Pila_Vacia) sim_tope = pop(&ps2);
            while(!Pila_Vacia && precedencia(sim_tope,simbolo)){
                  if(sim_tope != '(' && sim_tope != ')')
                        postr[actpos++] = sim_tope;
                  sim_tope = pop(&ps2);
                  aux = 1;
                  }
            if(aux == 0 || !Pila_Vacia) push(&ps2,sim_tope);
            if(Pila_Vacia || simbolo != ')') push(&ps2,simbolo);
            else sim_tope = pop(&ps2);
            }
      }
while(!Pila_Vacia){
      sim_tope = pop(&ps2);
      if(sim_tope != '(' && sim_tope != ')') postr[actpos++] = sim_tope;
      }
postr[actpos] = '\0';
}
```

Programa 4.3: Evaluación de expresiones **infijas** (Parte **3**/4).

```
/* Evalúa la expresión                                                    */
float evalua(char ent[]){
float a,b;
char op[1];
elemento1 *pila = NULL;            /* Pila Vacía                          */
int pos = 0;
while(ent[pos] != '\0'){
      *op = ent[pos++];

      switch(*op){
            case '+':
                  b = pop1(&pila);
                  a = pop1(&pila);
                  push1(&pila,a + b);
                  break;
            case '-':
                  b = pop1(&pila);
                  a = pop1(&pila);
                  push1(&pila,a - b);
                  break;
            case '*':
                  b = pop1(&pila);
                  a = pop1(&pila);
                  push1(&pila,a * b);
            break;
            case '/':
                  b = pop1(&pila);
                  a = pop1(&pila);
                  if(b == 0.){
                        printf("\n\t\aDivisión por CERO\n");
                        return(0.);
                        break;
                        }
                  push1(&pila,a / b);
                  break;
            default:
                  push1(&pila,atof(op)); /* Almacena dato en la pila   */
            }
      }
return(pop1(&pila));
}

/* Programa principal */
void main(){
char infija[MAX],postr[MAX];
clrscr();
printf("Evalúa expresiones Infijas.");
printf("\n\tTransforma una expresión Infija en PostFija y la Evalúa.");
printf("Al terminar la expresión, presione : ENTER");
lee(infija);
printf("\n\n\tExpresión Infija   =   %s\n",infija);
postfija(infija,postr);
printf("\n\n\tExpresión Postfija =   %s\n",postr);
printf("\n\n\tEvaluación de %s =   %g\n",infija,evalua(postr));
}
```

Programa 4.3: Evaluación de expresiones **infijas** (Parte **4**/4).

Al ejecutar el programa anterior produce el resultado que se muestra en la **Figura 4.3**.

```
                Evalúa expresiones Infijas.
Transforma una expresión Infija en PostFija y la Evalúa.
Al terminar la expresión, presione: ENTER
Entra la Expresión Infija: ((7+3)*8)/2+4
Expresión Infija   =   ((7+3)*8)/2+4
Expresión Postfija =   73+8*2/4+
Evaluación de ((7+3)*8)/2+4 =   44
```

Figura 4.3: Resultados del programa para evaluación de expresiones **infijas**.

4.4.4 Derivada de un polinomio

La derivada de un polinomio con elementos de la forma $\mathbf{c}\,x^{n}$, es la suma de las derivadas de cada elemento: $\mathbf{c}\,x^{n} \Rightarrow \mathbf{n*c}\,x^{n-1}$

En una lista cada **término** del polinomio puede ser representado por un **nodo** que contenga su coeficiente y su exponente de la forma en que se observa en **Figura 4.4**

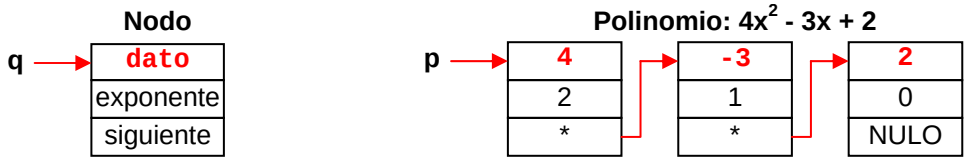


Figura 4.4: Un término de un polinomio en una lista y un polinomio como lista.

En el **Programa 4.4** se observa la implementación de este tipo de datos. Para derivar el polinomio, el programa recorre el polinomio y aplica la ecuación de la derivada.

```
#include "stdio.h"
#include "stdlib.h"
#include "string.h"
#include "conio.h"

/* Tipo de estructura y definición de estructura                    */
typedef struct datos nodo;
struct datos{
float dato;
int exp;
nodo *siguiente;
};

/* Mensaje de ERROR                                                 */
void error(void){
perror("\n\t\aERROR: Memoria insuficiente...");
exit(1);
}

/* Crea un nuevo nodo del tipo de la estructura                     */
nodo *Nuevo(){
nodo *q = (nodo *)malloc(sizeof(nodo));
if(!q) error();
return(q);
}
```

Programa 4.4: Calculo de la derivada de un polinomio (Parte **1**/4).

```
/* Pone un Menú                                                    */
void menu(void){
printf("Deriva Polinomio");
printf("P = Entra el Polinomio");
printf("D = Deriva el polinomio");
printf("B = Borra los Polinomios");
printf("V = Ver Polinomios");
printf("Q = Salir");
printf("Elija una Opción : ");
}

/* Inserta un nodo de forma ordenada                               */
void insertar(nodo **p,float dato,int exp){
nodo *pf = *p;
nodo *actual = pf, *anterior = pf,*q;
if(pf == NULL){
      pf = Nuevo();
      pf -> dato = dato;
      pf -> exp = exp;
      pf -> siguiente = NULL;
      *p = pf;
      return;
      }
/* Busca en la lista un punto de inserción                         */
while(actual != NULL && exp > actual -> exp){
      anterior = actual;
      actual = actual -> siguiente;
      }
/* Se inserta al principio o al final                              */
q = Nuevo();
if(anterior == actual){
      q -> dato = dato;
      q -> exp = exp;
      q -> siguiente = pf;
      pf = q;
      }
else  {
      q -> dato = dato;
      q -> exp = exp;
      q -> siguiente = actual;
      anterior -> siguiente = q;
      }
*p = pf;
}

/* Recorre la lista e imprime sus datos                            */
void ver(nodo *lista){
nodo *actual = lista;
if(actual == NULL) printf("\n\tLista vacía...");
else  {
      while(actual != NULL){
            printf("%+gx^%d ",actual -> dato,actual -> exp);
            actual = actual -> siguiente;
            }
      }
}
```

Programa 4.4: Calculo de la derivada de un polinomio (Parte **2**/4).

```
/* Programa principal                                                    */
void main(void){
nodo *lista1,*lista2;
nodo *q1;
int op,exp;
float dato;
lista1 = NULL;
lista2 = NULL;
while(1){
      do    {
            clrscr();
            menu();
            op = tolower(getch());
            }while(op != 'b' && op != 'd' && op != 'v' &&
                  op != 'p' && op != 'q');
      clrscr();
      switch(op){
            case 'p':
                  printf("\n\tIntroducir Polinomio
                        (Coeficiente=0 para terminar): ");
                  do{
                        printf("\n\tIntroducir Coeficiente : ");
                        scanf("%f",&dato);
                        if(dato == 0) break;
                        printf("\n\tIntroducir Exponente : ");
                        scanf("%d",&exp);
                        insertar(&lista1,dato,exp);
                  }while(dato != 0);
                  break;
            case 'd':
                  lista2 = NULL;
                  /* Deriva el polinomio */
                  q1 = lista1;
                  while(q1 != NULL){
                        if(q1 -> dato != 0 && q1 -> exp > 0)
                        insertar(&lista2,q1->exp * q1->dato,q1->exp - 1);
                        q1 = q1 -> siguiente;
                        }
                  /* Imprime resultados                                */
                  printf("* Polinomio :");
                  ver(lista1);
                  printf("* Derivada :");
                  ver(lista2);
                  getch();
                  break;
            case 'b':
                  lista1 = NULL;
                  lista2 = NULL;
                  break;
            case 'v':
                  /* Imprime resultados                                */
                  printf("* Polinomio :");
                  ver(lista1);
                  printf("* Derivada :");
                  ver(lista2);
```

Programa 4.4: Calculo de la derivada de un polinomio (Parte **3**/4).

```
                getch();
                break;
            case 'q':exit(0);
            }
      }
}
```

Programa 4.4: Calculo de la derivada de un polinomio (Parte **4**/4).

En la **Figura 4.5** se observa como opera el programa cuando se aplica al polinomio **$4x^2$ - 3x + 2**, cuya derivada es **8x – 3.**

```
        Deriva Polinomio
P = Entra el Polinomio
D = Deriva el polinomio
B = Borra los Polinomios
V = Ver Polinomios
Q = Salir
Elija una Opción: P
Introducir Polinomio (Coeficiente = 0 para terminar):
Introducir Coeficiente: 4
Introducir Exponente: 2
Introducir Coeficiente: -3
Introducir Exponente: 1
Introducir Coeficiente: 2
Introducir Exponente: 0
Introducir Coeficiente: 0
> D
* Polinomio:
+2 -3x +4x^2
* Derivada:
-3 +8x
```

Figura 4.5: Programa para derivar polinomios.

4.4.5 Suma y resta de polinomios

Los polinomios pueden ser representados, al igual que el caso anterior mediante una lista, en donde un nodo de la lista almacena el **coeficiente** y otro nodo al **exponente**, de cada término de los dos polinomios a sumar y restar.

En la solución, solo se almacenan los términos no nulos, por ejemplo tenemos los siguientes polinomios y su suma y resta:

Polinomio 1: $2x^4 - x + 10$
Polinomio 2: $x^3 - x + 5$
Suma: $\mathbf{2x^4 + x^3 - 2x + 15}$

Polinomio 2: $x^3 - x + 5$
Polinomio 1: $2x^4 - x + 10$
Resta: $\mathbf{-2x^4 + x^3 - 5}$

El **Programa 4.5** muestra la forma de implementar la suma y resta de dos polinomios.

```
#include "string.h"
#include "conio.h"

/* Tipo de estructura y definición de estructura                    */
typedef struct datos nodo;
struct datos{
float dato;
int exp;
nodo *siguiente;
};

/* Mensaje de ERROR                                                 */
void error(void){
perror("\n\t\aERROR: Memoria insuficiente...");
exit(1);
}

/* Crea un nuevo nodo del tipo de la estructura                     */
nodo *Nuevo(){
nodo *q = (nodo *)malloc(sizeof(nodo));
if(!q) error();
return(q);
}

/* Pone un Menú                                                     */
void menu(void){
printf("Suma y Resta de Polinomios");
printf("I = Entra Polinomio 1");
printf("P = Entra Polinomio 2");
printf("S = Suma y resta de los polinomios");
printf("B = Borra los Polinomios");
printf("V = Ver Polinomios");
printf("Q = Salir");
printf("Elija una Opción : ");
}

/* Inserta un nodo de forma ordenada                                */
void insertar(nodo **p,float dato,int exp){
nodo *pf = *p;
nodo *actual = pf, *anterior = pf,*q;
if(pf == NULL){
      pf = Nuevo();
      pf -> dato = dato;
      pf -> exp = exp;
      pf -> siguiente = NULL;
      *p = pf;
      return;
      }
/* Busca en la lista un punto de inserción                          */
while(actual != NULL && exp > actual -> exp){
      anterior = actual;
      actual = actual -> siguiente;
      }
/* Se inserta al principio o al final                               */
q = Nuevo();
```

Programa 4.5: Suma y Resta de polinomios (Parte **1**/4).

```
if(anterior == actual){
      q -> dato = dato;
      q -> exp = exp;
      q -> siguiente = pf;
      pf = q;
      }
else  {
      q -> dato = dato;
      q -> exp = exp;
      q -> siguiente = actual;
      anterior -> siguiente = q;
      }
*p = pf;
}

/* Recorre la lista e imprime sus datos                                    */
void ver(nodo *lista){
nodo *actual = lista;
if(actual == NULL) printf("\n\tLista vacía...");
else  {
      while(actual != NULL){
            printf("%+gx^%d ",actual -> dato,actual -> exp);
            actual = actual -> siguiente;
            }
      }
}

/* Busca un dato en la lista                                               */
nodo *busca(nodo *lista,int exp){
nodo *actual = lista;
while(actual != NULL && exp != actual -> exp) actual = actual->siguiente;
return(actual);
}

/* Programa principal                                                      */
void main(void){
nodo *lista1,*lista2,*lista3,*lista4;
nodo *q,*q1;
int op,exp;
float dato;
lista1 = NULL;
lista2 = NULL;
lista3 = NULL;
lista4 = NULL;
while(1){
      do    {
            clrscr();
            menu();
            op = tolower(getch());
            }while(op != 'i' && op != 'b' && op != 's' &&
                   op != 'v' && op != 'p' && op != 'q');
      clrscr();
      switch(op){
            case 'i':
                  printf("\n\tIntroducir Polinomio 1
                        (Coeficiente = 0 para terminar): ");
```

Programa 4.5: Suma y Resta de polinomios (Parte **2**/4).

```
                do    {
                      printf("\n\tIntroducir Coeficiente: ");
                      scanf("%f",&dato);
                      if(dato == 0) break;
                      printf("\n\tIntroducir Exponente: ");
                      scanf("%d",&exp);
                      insertar(&lista1,dato,exp);
                      }while(dato != 0);
                break;
          case 'p':
                printf("\n\tIntroducir Polinomio 1
                      (Coeficiente = 0 para terminar): ");
                do    {
                      printf("\n\tIntroducir Coeficiente: ");
                      scanf("%f",&dato);
                      if(dato == 0) break;
                      printf("\n\tIntroducir Exponente: ");
                      scanf("%d",&exp);
                      insertar(&lista2,dato,exp);
                      }while(dato != 0);
                break;
          case 's':
                lista3 = NULL;
                /* Busca exponentes iguales y los suma                */
                q1 = lista1;
                while(q1 != NULL){
                      q = busca(lista2, q1 -> exp);
                      if(!q)      {
                            /* El exponente no esta en la lista 2     */
                            insertar(&lista3, q1 -> dato,q1 -> exp);
                            q1 = q1 -> siguiente;
                            continue;
                            }
                      if((q->dato + q1->dato) != 0)
                      insertar(&lista3, q->dato + q1->dato, q->exp);
                      q1 = q1 -> siguiente;
                      }
                /* Exponentes de la lista 2 que no están en lista 1   */
                q1 = lista2;
                while(q1 != NULL) {
                      q = busca(lista1,q1 -> exp);
                      if(!q)      {
                            insertar(&lista3,q1->dato, q1->exp);
                            q1 = q1 -> siguiente;
                            continue;
                            }
                      q1 = q1 -> siguiente;
                      }
                lista4 = NULL;
                /* Busca exponentes iguales y los resta               */
                q1 = lista1;
                while(q1 != NULL){
                      q = busca(lista2,q1 -> exp);
                      if(!q)      {
                            /* El exponente no esta en la lista 2     */
                            insertar(&lista4, -(q1->dato), q1->exp);
```

Programa 4.5: Suma y Resta de polinomios (Parte **3**/4).

```
                              q1 = q1 -> siguiente;
                              continue;
                              }
                        if((q->dato - q1->dato) != 0)
                        insertar(&lista4, q->dato - q1->dato, q->exp);
                        q1 = q1 -> siguiente;
                        }
                  /* Exponentes de la lista 2 que no están en lista 1  */
                  q1 = lista2;
                  while(q1 != NULL) {
                        q = busca(lista1,q1 -> exp);
                        if(!q)      {
                              insertar(&lista4, q1->dato, q1->exp);
                              q1 = q1 -> siguiente;
                              continue;
                              }
                        q1 = q1 -> siguiente;
                        }
                  /* Imprime resultados                                  */
                  printf("* Polinomio 1:");
                  ver(lista1);
                  printf("* Polinomio 2:");
                  ver(lista2);
                  printf("* Polinomio SUMA  : P1 + P2");
                  ver(lista3);
                  printf("* Polinomio RESTA : P2 - P1");
                  ver(lista4);
                  getch();
                  break;
             case 'b':
                  lista1 = NULL;
                  lista2 = NULL;
                  lista3 = NULL;
                  lista4 = NULL;
                  break;
            case 'v':
                  /* Imprime resultados                                  */
                  printf("* Polinomio 1:");
                  ver(lista1);
                  printf("* Polinomio 2:");
                  ver(lista2);
                  printf("* Polinomio SUMA  : P1 + P2");
                  ver(lista3);
                  printf("* Polinomio RESTA : P2 - P1");
                  ver(lista4);
                  getch();
                  break;
            case 'q':exit(0);
            }
      }
}
```

Programa 4.5: Suma y Resta de polinomios (Parte **4**/4).

En la **Figura 4.6** se observa como opera el programa cuando se aplica a los siguientes polinomios:

Polinomio 1: $2x^4 - x + 10$
Polinomio 2: $x^3 - x + 5$
Suma: $2x^4 + x^3 - 2x + 15$

Polinomio 2: $x^3 - x + 5$
Polinomio 1: $2x^4 - x + 10$
Resta: $-2x^4 - x^3 - 5$

```
        Suma y Resta de Polinomios
I = Entra Polinomio 1
P = Entra Polinomio 2
S = Suma y resta de los polinomios
B = Borra los Polinomios
V = Ver Polinomios
Q = Salir
Elija una Opción: I
Introducir Polinomio 1 (Coeficiente = 0 para terminar):
Introducir Coeficiente: 2
Introducir Exponente: 4
Introducir Coeficiente: -1
Introducir Exponente: 1
Introducir Coeficiente: 10
Introducir Exponente: 0
Introducir Coeficiente: 0
>P
Introducir Polinomio 1 (Coeficiente = 0 para terminar):
Introducir Coeficiente: 1
Introducir Exponente: 3
Introducir Coeficiente: -1
Introducir Exponente: 1
Introducir Coeficiente: 5
Introducir Exponente: 0
Introducir Coeficiente: 0
>S
* Polinomio 1:
+10 -1x +2x^4
* Polinomio 2:
+5 -1x +1x^3
* Polinomio SUMA: P1 + P2
+15 -2x +1x^3 +2x^4
* Polinomio RESTA: P2 - P1
-5 +1x^3 -2x^4
```

Figura 4.6: Suma y Resta de polinomios.

4.4.6 Problema de Josephus

El **problema de Josephus**, establece que un grupo de soldados se encuentra rodeado por la fuerza enemiga. No hay esperanza de victoria sin refuerzos, pero sólo hay un caballo para escapar. Los soldados hacen un pacto para determinar cuál de ellos va escapar para pedir ayuda. Forman un círculo y se elige un número **n** y un **nombre**. Iniciando en **nombre**, y en sentido de las manecillas del reloj, se cuenta hasta **n** y el soldado se retira del círculo, y la cuenta continúa en el soldado siguiente. El proceso sigue, hasta que solo quede el soldado que escapa.

Esté problema puede solucionarse de forma directa mediante una lista circular, en donde cada **nodo** tiene el **nombre** del soldado y se elimina aquel nodo que se encuentre en la posición **n**.

Por ejemplo, para **n = 3**, y los siguientes nombres: `aa, bb, cc, dd` y `ee`. Tenemos:

Pasada	Nombres	Se retira
1	aa, bb, cc, dd, ee	cc
2	bb, dd, ee	aa
3	bb, dd	ee
4	Dd	bb

El soldado que escapa es: **dd**
El **Programa 4.6** muestra la solución al **problema de Josephus**.

```
#include "string.h"
#include "conio.h"
#define MAX 80

/* Tipo de estructura y definición de estructura                    */
typedef struct datos nodo;
struct datos{
char dato[MAX];
nodo *siguiente;
};

/* Mensaje de ERROR                                                 */
void error(void){
perror("\n\t\aERROR: Memoria insuficiente...");
exit(1);
}

/* Crea un nuevo nodo del tipo de la estructura                     */
nodo *Nuevo(){
nodo *q = (nodo *)malloc(sizeof(nodo));
if(!q) error();
return(q);
}

/* Pone un Menú                                                     */
void menu(void){
printf("Problema de Josephus");
printf("I = Introducir un Nombre");
printf("N = Introducir N");
printf("R = Resolver el problema");
printf("V = Ver Todos los Nombres");
printf("Q = Salir");
printf("Elija una Opción : ");
}

/* Inserta un nodo en la lista                                      */
void insertar(nodo **p,char dato[]){
nodo *q,*lista;
lista = *p;
q = Nuevo();
strcpy(q -> dato,dato);
if(lista == NULL) lista = q;
```

Programa 4.6: Problema de Josephus (Parte **1**/3).

```
else q -> siguiente = lista -> siguiente;
lista -> siguiente = q;
*p = q;
}
/* Busca y elimina un dato de la lista                                    */
void suprimir(nodo **p,char dato[]){
nodo *cabecera = *p,*final = *p;
nodo *actual = cabecera,*anterior = cabecera;
if(cabecera == NULL){
      printf("\n\tLista Vacía...");
      return;
      }
do    {
      anterior = actual;
      actual = actual -> siguiente;
      }while(strcmp(dato,actual -> dato) && actual != final);
if(strcmp(dato,actual -> dato)) return;  /* No está el dato           */
/* Borrar dato de la lista */
if(anterior == actual) cabecera = cabecera -> siguiente;
else anterior -> siguiente = actual -> siguiente;
if(actual == final) cabecera = anterior;
free(actual);
*p = cabecera;
}

/* Recorre la lista y muestra sus datos                                   */
int ver(nodo *lista){
nodo *actual = lista,*final = lista;
int i = 0;
if(actual == NULL) printf("\n\tLista vacía...");
else  {
      do    {
            actual = actual -> siguiente;
            printf("\n\tSoldado %d = %s",++i,actual -> dato);
            }while(actual != final);
      }
return(i);
}

/* Programa principal                                                     */
void main(void){
nodo *lista,*final;
nodo *q;
char op,dato[MAX];
int n,i,pos,k;
lista = NULL;
n = 3;
while(1){
      do    {
            clrscr();
            menu();
            op = tolower(getch());
            }while(op != 'i' && op != 'r' && op != 'n' &&
                   op != 'v' && op != 'q');
      clrscr();
```

Programa 4.6: Problema de Josephus (Parte **2**/3).

```
	switch(op){
		case 'i':
			printf("\n\tNombre : ");
			pos = 0;
			while((dato[pos++] = getchar()) != '\n');
			dato[--pos] = '\0';
			insertar(&lista,dato);
			break;
		case 'n':
			printf("\n\tN (%d) : ",n);
			scanf("%d",&n);
			break;
		case 'r':
			final = lista;
			if(lista == NULL){
				printf("\n\tLISTA VACÍA...");
				getch();
				break;
				}
			i = 3;
			while(lista != final -> siguiente && i > 1) {
				for(k = 1;k <= n;k++) lista = lista->siguiente;
				suprimir(&lista,lista -> dato);
				i = ver(lista);
				}
			strcpy(dato,lista -> dato);
			lista = lista -> siguiente;
			suprimir(&lista,lista -> dato);
			lista = NULL;
			printf("El soldado que escapa es: %s",dato);
			getch();
			break;
		case 'v':
			i = ver(lista);
			getch();
			break;
		case 'q':exit(0);
		}
	}
}
```

Programa 4.6: Problema de Josephus (Parte **3**/3).

En la **Figura 4.7** se observa como opera el programa cuando se introduce los nombres `aa, bb, cc, dd` y `ee` y **n = 3**.

```
         Problema de Josephus
I = Introducir un Nombre
N = Introducir N
R = Resolver el problema
V = Ver Todos los Nombres
Q = Salir
Elija una Opción: R
Datos de la Lista
Soldado 1 = dd
Soldado 2 = ee
Soldado 3 = aa
Soldado 4 = bb
Datos de la Lista
Soldado 1 = bb
Soldado 2 = dd
Soldado 3 = ee
Datos de la Lista
Soldado 1 = bb
Soldado 2 = dd
Datos de la Lista
Soldado 1 = dd
El soldado que escapa es: dd
Presione alguna tecla para continuar.
```

Figura 4.7: Problema de Josephus.

4.4.7 Suma de enteros grandes

Las computadoras, solo permiten la adición de números de cierta longitud. Si se requiere sumar dos números de muchos dígitos, las listas circulares, con **encabezado**, son una buena opción. Lo único que se requiere es representar números mediante una lista, el encabezado (por ejemplo **-1**), es necesario para conocer el inicio y/o el final del número. La representación del número `901,256,781,234` con listas circulares se observa en **Figura 2.2.10**.

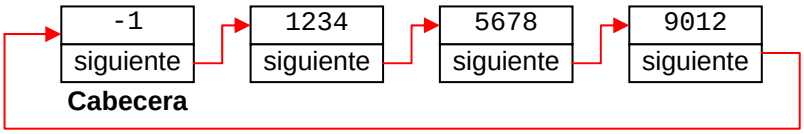


Figura 4.8: Lista circular, para representar números enteros grandes.

Para sumar dos números representados de esta forma, basta seguir las reglas de la suma, sumando nodo con nodo y tomar en cuenta el acarreo.

El **Programa 4.7** suma dos enteros largos **A** y **B**, ingresados nodo por nodo, en donde cada nodo tiene un máximo de cuatro dígitos.

```
/* Tipo de estructura y definición de estructura                    */
typedef struct datos nodo;
struct datos{
int dato;
nodo *siguiente;
};

/* Mensaje de ERROR                                                 */
void error(void){
perror("\n\t\aERROR: Memoria insuficiente...");
exit(1);
}

/* Crea un nuevo nodo del tipo de la estructura                     */
nodo *Nuevo(){
nodo *q = (nodo *)malloc(sizeof(nodo));
if(!q) error();
return(q);
}

/* Pone un Menú                                                     */
void menu(void){
printf("Suma dos enteros grandes con Listas Circulares");
printf("A = Introducir Entero A");
printf("B = Introducir Entero B");
printf("S = Sumar A + B");
printf("V = Ver Listas");
printf("I = Iniciar Listas");
printf("Q = Salir");
printf("Elija una Opción : ");
}

/* Inserta un nodo en la lista                                      */
void insertar(nodo **p,int dato){
nodo *q,*lista;
lista = *p;
q = Nuevo();
q -> dato = dato;
if(lista == NULL) lista = q;
else q -> siguiente = lista -> siguiente;
lista -> siguiente = q;
*p = q;
}

/* Suma dos listas con apuntadores a p y a q                        */
nodo *suma(nodo *a,nodo *b){
long int div = 10000L;
long int acarreo, numero, total;
nodo *s,*p = a,*q = b;
p = p -> siguiente;     /* p y q apuntan al último nodo             */
q = q -> siguiente;
p = p -> siguiente;     /* p y q apuntan al nodo -1                 */
q = q -> siguiente;
s = Nuevo();            /* p y q apuntan al primer nodo             */
s -> dato = -1;
s -> siguiente = s;
```

Programa 4.7: Suma de enteros grandes (Parte **1**/3).

```
acarreo = 0;      /* Suma dos términos de los números                    */
while(p -> dato != -1 && q -> dato != -1){
      total = p -> dato + q -> dato + acarreo;
      numero = total % div;
      insertar(&s,numero);
      q = q -> siguiente;
      p = p -> siguiente;
      acarreo = total / div;
      }
while(p -> dato != -1){ /* Inserta los restantes del primer número     */
      total = p -> dato + acarreo;
      numero = total%div;
      insertar(&s,numero);
      p = p -> siguiente;
      acarreo = total / div;
      }
while(q -> dato != -1){ /* Inserta los restantes del segundo número    */
      total = q -> dato + acarreo;
      numero = total % div;
      insertar(&s,numero);
      q = q -> siguiente;
      acarreo = total / div;
      }
if(acarreo == 1) insertar(&s,acarreo);
return(s);
}

/* Recorre la lista y muestra sus datos                                */
void ver(nodo *lista){
nodo *actual = lista;
if(actual -> dato == -1) printf("\n\tLista vacía...");
else  {
      actual = actual -> siguiente;
      printf("\n");
      gotoxy(70,wherey());
      do    {
            actual = actual -> siguiente;
            if(actual -> dato != -1)
                  printf("%04d\b\b\b\b\b\b\b\b\b",actual -> dato);
            }while(actual -> dato != -1);
      }
}

/* Programa principal                                                  */
void main(void){
nodo *a,*b,*r;
long int ent;
char c;
a = Nuevo();
a -> dato = -1;
a -> siguiente = a;
b = Nuevo();
b -> dato = -1;
b -> siguiente = b;
r = Nuevo();
r -> dato = -1;
```

Programa 4.7: Suma de enteros grandes (Parte **2**/3).

```
r -> siguiente = r;
do    {
      clrscr();
      menu();
      c = tolower(getch());
      clrscr();
      switch(c){
            case 'a':
                  printf("\n\tIntroducir dato A #### : ");
                  scanf("%4d",&ent);
                  insertar(&a,ent);
                  flushall();
                  break;
            case 'b':
                  printf("\n\tIntroducir dato B #### : ");
                  scanf("%4d",&ent);
                  insertar(&b,ent);
                  flushall();
                  break;
            case 's':
                  r = suma(a,b);
                  printf("Entero A:");
                  ver(a);
                  printf("Entero B:");
                  ver(b);
                  printf("SUMA:");
                  ver(r);
                  getch();
                  break;
            case 'v':
                  printf("Entero A:");
                  ver(a);
                  printf("Entero B:");
                  ver(b);
                  printf("SUMA:");
                  ver(r);
                  getch();
                  break;
            case 'i':
                  a = Nuevo();
                  a -> dato = -1;
                  a -> siguiente = a;
                  b = Nuevo();
                  b -> dato = -1;
                  b -> siguiente = b;
                  r = Nuevo();
                  r -> dato = -1;
                  r -> siguiente = r;
                  break;
            }
      }while(c != 'q');

}
```

Programa 4.7: Suma de enteros grandes (Parte **3**/3).

En la **Figura 4.9** se muestran los resultados al sumar dos enteros largos.

```
        Suma dos enteros grandes con Listas Circulares
Los números A y B se introducen nodo a nodo de 4 dígitos.
A = Introducir Entero A
B = Introducir Entero B
S = Sumar A + B
V = Ver Listas
I = Iniciar Listas
Q = Salir
Elija una Opción: S
Entero A:
       9012 5678 1234
Entero B:
            5412 2536
SUMA:
       9013 1090 3770
```

Figura 4.9: Suma de enteros grandes.

4.4.8 Suma de Polinomios

A manera de ejemplo del uso de listas circulares, realizaremos la suma de dos ecuaciones algebraicas o polinómicas con variables x, y y z. Por ejemplo:

Polinomio 1: $2z + 2y^2 + 4x^4$
Polinomio 2: $3y^2 + 5x^4$
Suma: $\mathbf{2z + 5y^2 + 9x^4}$

Cada término del polinomio es representado de la forma en que se observa en la **Figura 4.10**, en donde también se representa a un polinomio.

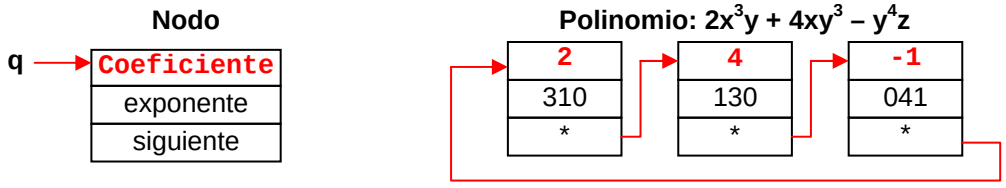


Figura 4.10: Un elemento de un polinomio en una lista circular y un polinomio como lista circular.

El **Programa 4.8** incluye las siguientes funciones:

- **Lee**.- lee un polinomio en orden creciente de los exponentes; si se crea una lista, esta solo consta de un elemento de cabecera.
- **Iniciar**.- sitúa el apuntador `actual` sobre el primer término de un polinomio.
- **Compara**.- compara termino a termino dos polinomios, y determina si ambos términos se suman o se insertan para formar la solución.
- **Suprimir**.- elimina el término nulo de un polinomio.
- **Escribir**.- permite ver el polinomio solución, recorriendo la lista.

```
/* Tipo de estructura y definición de estructura                    */
typedef struct datos nodo;
typedef nodo *pnodo;
struct datos{
float coeficiente;
int exponente;
pnodo siguiente;
};

/* Estructura de la lista circular                                  */
typedef struct lista lista_C;
struct lista{
pnodo cabecera;
pnodo anterior;
pnodo actual;
};

/* Definición de funciones                                          */
void lee(lista_C *,char);
void iniciar(lista_C *);
void compara(lista_C *,lista_C *);
void suma(lista_C *,lista_C *);
void suprimir(lista_C *);
void insertar(lista_C *,lista_C *);
void escribir(lista_C);

/* Mensaje de ERROR                                                 */
void error(void){
perror("\n\t\aERROR: Memoria insuficiente...");
exit(1);
}

/* Crea un nuevo nodo del tipo de la estructura                     */
nodo *Nuevo(){
nodo *q = (nodo *)malloc(sizeof(nodo));
if(!q) error();
return(q);
}

/* Lee un polinomio                                                 */
void lee(lista_C *polX,char n){
int exp;
float coef;
pnodo q;
polX -> cabecera = Nuevo();
polX -> cabecera -> coeficiente = 0;
polX -> cabecera -> exponente = -001;
polX -> cabecera -> siguiente = polX -> cabecera;
polX -> anterior = polX -> actual = NULL;
printf("\n\tIntroducir Polinomio: Coeficiente(x^a.y^b.z^c)",n);
printf("\n\tPara finalizar presione Coeficiente = 0");
printf("%c Coeficiente: ",n);
scanf("%f",&coef);
while(coef){
      printf("Exponente abc (sin espacios): ");
      scanf("%d",&exp);
      q = Nuevo();
```

Programa 4.8: Suma de polinomios con listas circulares (Parte **1**/3).

```
      q -> coeficiente = coef;
      q -> exponente = exp;
      q -> siguiente = polX -> cabecera -> siguiente;
      polX -> cabecera -> siguiente = q;
      printf("%c Coeficiente: ",n);
      scanf("%f",&coef);
      }
}

/* Iniciar proceso                                                    */
void iniciar(lista_C *polX){
polX -> anterior = polX -> cabecera;
polX -> actual = polX -> cabecera -> siguiente;
}

/* Comparar términos de los polinomios                                */
void compara(lista_C *polP,lista_C *polQ){
while(!(polP -> actual -> exponente < 0)){
      while(polP->actual->exponente < polQ->actual->exponente){
            polQ -> anterior = polQ -> actual;
            polQ -> actual = polQ -> actual -> siguiente;
            }
      if(polP->actual->exponente == polQ->actual->exponente) {
            suma(polP,polQ);
            }
      else  {
            insertar(polP,polQ);
            polP -> actual = polP -> actual -> siguiente;
            }
      }
}

/* Suma ambos polinomios término a término                            */
void suma(lista_C *polP,lista_C *polQ){
if(polP -> actual -> exponente < 0) return;
else  {
      polQ->actual->coeficiente += polP->actual->coeficiente;
      if(polQ -> actual -> coeficiente == 0){
            suprimir(polQ);
            polP -> actual = polP -> actual -> siguiente;
            }
      else  {
            polP -> actual = polP -> actual -> siguiente;
            polQ -> anterior = polQ -> actual;
            polQ -> actual = polQ -> actual -> siguiente;
            }
      }
}

/* Inserta un nodo                                                    */
void insertar(lista_C *polP,lista_C *polQ){
pnodo q;
q = Nuevo();
q -> coeficiente = polP -> actual -> coeficiente;
q -> exponente = polP -> actual -> exponente;
q -> siguiente = polQ -> actual;
```

Programa 4.8: Suma de polinomios con listas circulares (Parte **2**/3).

```
polQ -> anterior = polQ -> anterior -> siguiente = q;
return;
}

/* Saca y elimina un nodo                                                */
void suprimir(lista_C *polQ){
pnodo q;
q = polQ -> actual;
polQ -> actual = polQ -> actual -> siguiente;
polQ -> anterior -> siguiente = polQ -> actual;
free(q);
return;
}

/* Escribe la suma, recorriendo la lista                                 */
void esribir(lista_C polQ){
printf("Suma: ");
polQ.cabecera = polQ.cabecera -> siguiente;
while(polQ.cabecera -> exponente != -1){
      if(polQ.cabecera -> coeficiente != 0)
            printf(" %+g xyz%03d",polQ.cabecera -> coeficiente,
                                  polQ.cabecera -> exponente);
      polQ.cabecera = polQ.cabecera -> siguiente;
      }
}

/* Programa principal                                                    */
void main(void){
lista_C polP,polQ;
clrscr();
printf("Suma de Polinomios, con listas circulares.");
lee(&polP,'A');
lee(&polQ,'B');
iniciar(&polP);
iniciar(&polQ);
compara(&polP,&polQ);
clrscr();
printf("Suma de Polinomios, con listas circulares.");
esribir(polQ);
}
```

Programa 4.8: Suma de polinomios con listas circulares (Parte **3**/3).

En la **Figura 4.11** se observa como opera el programa cuando se aplica a los siguientes polinomios:

Polinomio 1: $2z + 2y^2 + 4x^4$
Polinomio 2: $3y^2 + 5x^4$
Suma: $\mathbf{2z + 5y^2 + 9x^4}$

```
Suma de Polinomios, con listas circulares.
A Coeficiente: 2
Exponente abc (sin espacios): 001
A Coeficiente: 2
Exponente abc (sin espacios): 020
A Coeficiente: 4
Exponente abc (sin espacios): 400
Introducir Polinomio B. Término a término: Coeficiente(x^a.y^b.z^c)
B Coeficiente: 3
Exponente abc (sin espacios): 020
B Coeficiente: 5
Exponente abc (sin espacios): 400
Suma de Polinomios, con listas circulares.
Suma:
 +9x^4 +5y^2 +2z
```

Figura 4.11: Suma de polinomios con listas circulares.

4.4.9 Búsqueda Binaria

En la **búsqueda binaria**, los datos están ordenados, con lo que se aumenta la eficiencia de la búsqueda. Por ejemplo tenemos un directorio telefónico ó un diccionario, en donde los datos se encuentran ordenados de forma alfabética. Si no se procediera a darles un orden alfabético, el diccionario o el directorio telefónico serían inoperables.

La idea es entonces que dado un elemento **x** elegido para su búsqueda, compararlo con un elemento del arreglo elegido al azar, este puede ser el elemento que se encuentra a la mitad del arreglo, si ambos coinciden se termina la búsqueda, si es menor que **x**, se busca hacia la parte superior y si es mayor que **x** se busca hacia la parte inferior, siempre tratando de empezar cada nueva búsqueda por la parte central del subarreglo restante.

Por ejemplo tenemos el siguiente arreglo de datos:

Datos	2	3	5	7	15	18	23	31	36	48	56
Índice	0	1	2	3	4	5	6	7	8	9	10

Para encontrar el elemento x = 48 se sigue el siguiente procedimiento:

1) Seleccionamos la mitad del arreglo: **mitad = (0 + 10) / 2 = 5**, y comparamos: **x** con el dato del índice **5**.

Datos	2	3	5	7	15	**18**	23	31	36	48	56
Índice	0	1	2	3	4	**5**	6	7	8	9	10

Como **48** es mayor a **18** seguimos con el proceso, tomando la parte superior del arreglo.

2) Seleccionamos la mitad del arreglo: **mitad = (6 + 10) = 8**, y comparamos: **x** con el dato del índice **8**.

Datos	2	3	5	7	15	18	23	31	**36**	48	56
Índice	0	1	2	3	4	5	6	7	**8**	9	10

Como **48** es mayor a **36** seguimos con el proceso, tomando la parte superior del arreglo.

3) Seleccionamos la mitad del arreglo: **mitad = (9 + 10) = 9**, y comparamos: **x** con el dato del índice **9**.

Datos	2	3	5	7	15	18	23	31	36	**48**	56
Índice	0	1	2	3	4	5	6	7	8	**9**	10

Como **48** es igual a **48**, por lo que terminamos el proceso, pues encontramos el dato buscado en **índice = 9**.

En solo tres comparaciones encontramos el elemento buscado, por lo que se observa un enorme incremento en la eficiencia, respecto a la búsqueda lineal, la cual hubiese requerido de nueve comparaciones.El número promedio de comparaciones para este tipo de búsqueda es $\log_2 n$. El algoritmo que realiza este tipo de búsqueda se muestra en el **Programa 4.9**.

```
#include "conio.h"
int a[] = {0,2,3,4,5,6,7,8,9,10};
int x;

/* Búsqueda Binaria                                                    */
int busqueda_binaria(int min,int max){
int med;
if(min > max) return(-1);
med =(min + max) / 2;
if(x == a[med]) return(med);
if(x > a[med]) return(busqueda_binaria(med + 1,max));
else return(busqueda_binaria(min,med - 1));
}

/* Programa principal */
void main(){
int y;
x = 10;
y = busqueda_binaria(0,10);
if(y >= 0)printf("\n\ta[%d] = %d",y,a[y]);
else printf("\n\n\t%d No esta en el arreglo.",x);
}
```

Programa 4.9: Búsqueda Binaria.

4.4.10 Definición recursiva de expresiones algebraicas

Una cadena algebraica es una cadena recursiva, en donde cada elemento se define de forma recursiva dadas las siguientes definiciones:

- **Expresión**.- es un término seguido por un signo más seguido por un término, o un término solo.
- **Término**.- es un factor seguido por un asterisco seguido por un factor o un factor solo.
- **Factor**.- es una letra o una expresión encerrada entre paréntesis.

Un programa recursivo, puede determinar si una cadena es válida, de acuerdo a las definiciones anteriores. El **Programa 4.10** muestra un programa que acepta expresiones del tipo **a*(b+c)** y determina si es válida o no.

```
#include "stdio.h"
#include "ctype.h"
#define TRUE      1
#define FALSE     0
#define MAX 100
char lee_simb(char cad[],int largo,int *p);
char factor(char cad[],int largo,int *p);
char term(char cad[],int largo,int *p);
char expr(char cad[],int largo,int *p);

/* Regresa un carácter o un espacio                                    */
/* cad = entrada de la cadena de caracteres                            */
/* largo = longitud de cad                                             */
/* *p = es la ultima posición usada de cad                             */
char lee_simb(char cad[],int largo,int *p){
char c;
if(*p < largo) c = cad[*p];
else c = ' ';
(*p)++;
return(c);
}

/* Regresa verdadero si es un factor                                   */
/* cad = entrada de la cadena de caracteres                            */
/* largo = longitud de cad                                             */
/* *p = es la ultima posición usada de cad                             */
char factor(char cad[],int largo,int *p){
int c;
if((c = lee_simb(cad,largo,p)) != '(') return(isalpha(c));
return(expr(cad,largo,p) && lee_simb(cad,largo,p) == ')');
}

/* Regresa verdadero si es un Término                                  */
/* cad = entrada de la cadena de caracteres                            */
/* largo = longitud de cad                                             */
/* *p = es la ultima posición usada de cad                             */
char term(char cad[],int largo,int *p){
if(factor(cad,largo,p) == FALSE) return(FALSE);
if(lee_simb(cad,largo,p) != '*') {
      (*p)--;
      return(TRUE);
      }
return(factor(cad,largo,p));
}
```

Programa 4.10: Verificador de cadenas algebraicas (Parte **1**/2).

```
/* Regresa verdadero si es una expresión                                */
/* cad = entrada de la cadena de caracteres                             */
/* largo = longitud de cad                                              */
/* *p = es la ultima posición usada de cad                              */
char expr(char cad[],int largo,int *p){
if(term(cad,largo,p) == FALSE) return(FALSE);
if(lee_simb(cad,largo,p) != '+') {
      (*p)--;
      return(TRUE);
      }
return(term(cad,largo,p));
}

/* Programa principal                                                   */
void main(){
char cad[MAX];
int largo,pos;
printf("Verifica que una expresión algebraica sea válida.");
printf("Al terminar la expresión presione ENTER.");
printf("\n\n\t> ");
largo = 0;
while((cad[largo++] = getchar()) != '\n');
cad[--largo]='\0';
pos = 0;
if(expr(cad,largo,&pos) == TRUE && pos >= largo) printf("Cadena Válida");
else printf("Cadena no válida");
}
```

Programa 4.10: Verificador de cadenas algebraicas (Parte **2**/2).

Al ejecutar el **Programa 4.10** tenemos la salida de la **Figura 4.12**.

```
Verifica que una expresión algebraica sea válida.
Al terminar la expresión presione ENTER.
> (a*b+c*d)+(e*(f)+g)
Cadena Válida
> (a*b+c*d)+(e*(f)+g
Cadena no válida
```

Figura 4.12: Verificador de cadenas algebraicas.

4.4.11 Conversión prefija a postfija

Se trata de convertir una cadena prefija a su forma postfija (**Programa 4.11**). El algoritmo que realiza dicha conversión es el siguiente:

- Si la cadena prefija es de una sola variable, esta es su propia equivalencia postfija.
- Se busca al primer operador de la cadena prefija.
- Se encuentra el primer operando de la cadena y se convierte a postfija.
- Se encuentra el segundo operando de la cadena y se convierte a postfija.
- Se juntan las cadenas postfijas encontradas.

```
#include "stdio.h"
#include "ctype.h"
#include "string.h"
#define MAX 100
void pp(char prefija[],char postfija[]);

/* Recorta cad entre n y m y regresa op1                                */
char substr(char cad[],int n,int m,char op1[]){
int i,j = 0;
for(i = n;i <= m;i++) op1[j++] = cad[i];
op1[j] = '\0';
return(*op1);
}

/* Realiza la conversión prefija a postfija                             */
void pp(char prefija[],char postfija[]){
char f[2],t1[MAX],t2[MAX];
f[0] = prefija[0];
f[1] = '\0';
substr(prefija,1,strlen(prefija)-1,prefija);
if(f[0] == '+' || f[0] == '-' ||  f[0] == '*' ||  f[0] == '/' ){
      pp(prefija,t1);
      pp(prefija,t2);
      strcat(t1,t2);
      strcat(t1,f);
      substr(t1,0,strlen(t1),postfija);
      return;
      }
postfija[0] = f[0];
postfija[1] = '\0';
}

/* Programa principal                                                   */
void main(){
char prefija[MAX],postfija[MAX];
int pos = 0;
printf("Convierte una Cadena Prefija a Postfija.");
printf("\n\n\t> ");
while((prefija[pos++] = getchar()) != '\n');
prefija[--pos] = '\0';
printf("Cadena prefija: %s",prefija);
pp(prefija,postfija);
printf("Cadena Postfija: %s",postfija);
}
```

Programa 4.11: Conversión prefija a postfija.

Al ejecutar el **Programa 4.11** tenemos la salida de la **Figura 4.13**.

```
Convierte una Cadena Prefija a Postfija.
Al terminar la expresión presione ENTER.
> +a*bc
Cadena Prefija:  +a*bc
Cadena Postfija: abc*+
```

Figura 4.13: Conversión prefija a postfija.

4.4.12 Fractales

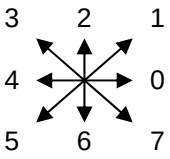
Una aplicación de los procesos recursivos, se encuentra en la elaboración de graficas fractales. Regularmente se forman por la superposición repetida de un patrón grafico simple.

Curvas de Hilbert

El procedimiento de elaboración, consiste en dibujar líneas de una longitud definida, según una serie de reglas. Estas curvas reciben su nombre en honor al matemático que las invento en 1891. Las reglas son las que se muestran a continuación, en donde, cada flecha indica la dirección de la línea y cada letra una función:

Inicio A

A:	D ←	A ↓	A →	B
B:	C ↑	B →	B ↓	A
C:	B →	C ↑	C ←	D
D:	A ↓	D ←	D ↑	C



Por ejemplo para la **curva de Hilbert** de orden 1 y 2, aplicamos las definiciones y se generan las curvas que se muestran en la **Figura 4.14**.

Figura 4.14: Curvas de Hilbert de orden 1 y 2.

El **Programa 4.12** muestra, como se codifica en **C**, este procedimiento.

```
#include "graphics.h"
int u,x0,y0;
void a(int);
void b(int);
void c(int);
void d(int);

/* Imprime línea con dirección y tamaño */
void linea(int dir,int tam){
int x1,y1;
if(dir == 0){
      x1 = x0 + tam;
      y1 = y0;
      }
if(dir == 2){
      x1 = x0;
      y1 = y0 - tam;
      }
if(dir == 4){
      x1 = x0 - tam;
      y1 = y0;
      }
if(dir == 6){
      x1 = x0;
      y1 = y0 + tam;
      }
```

Programa 4.12: Curvas de Hilbert (Parte **1**/3).

```
line(x0,y0,x1,y1);
x0 = x1;
y0 = y1;
}

void a(int i){
if(i > 0){
      d(i - 1);
      linea(4,u);
      a(i - 1);
      linea(6,u);
      a(i - 1);
      linea(0,u);
      b(i - 1);
      }
}

void b(int i){
if(i > 0){
      c(i - 1);
      linea(2,u);
      b(i - 1);
      linea(0,u);
      b(i - 1);
      linea(6,u);
      a(i - 1);
      }
}

void c(int i){
if(i > 0){
      b(i - 1);
      linea(0,u);
      c(i - 1);
      linea(2,u);
      c(i - 1);
      linea(4,u);
      d(i - 1);
      }
}

void d(int i){
if(i > 0){
      a(i - 1);
      linea(6,u);
      d(i - 1);
      linea(4,u);
      d(i - 1);
      linea(2,u);
      c(i - 1);
      }
}
```

Programa 4.12: Curvas de Hilbert (Parte **2**/3).

```
void main(){
int graphdriver = DETECT,graphmode;
initgraph(&graphdriver,&graphmode,"");
cleardevice();
u = 30;
x0 = 400;
y0 = 80;
a(5);
getch();
closegraph();
}
```

Programa 4.12: Curvas de Hilbert (Parte **3**/3).

En la **Figura 4.15** se observa la **curva de Hilbert** de orden **5**.

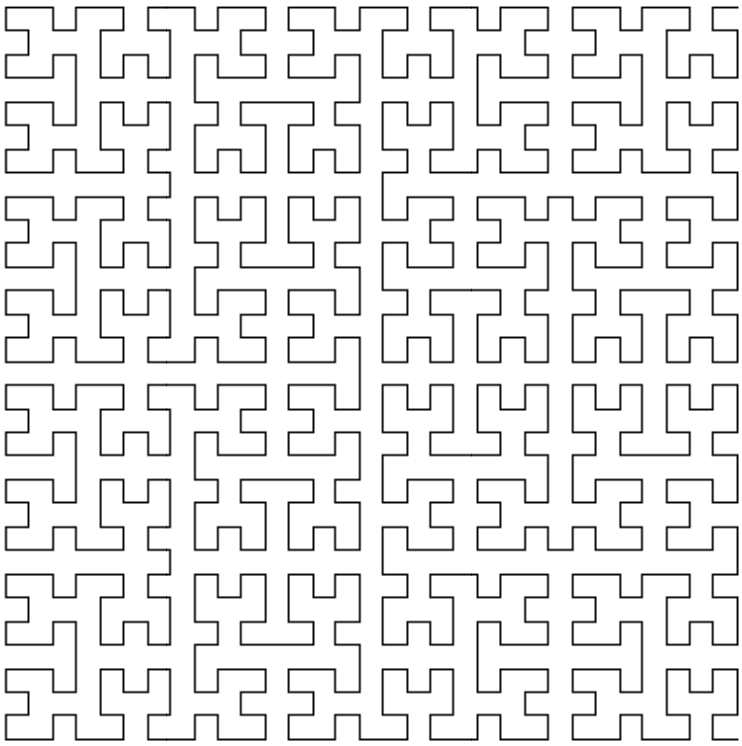
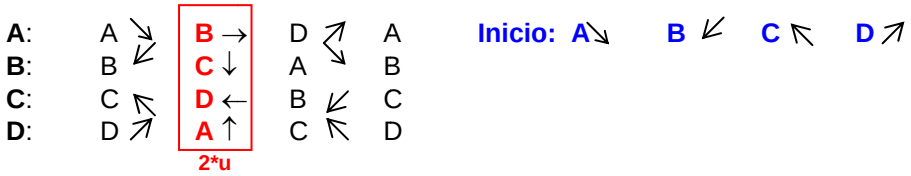


Figura 4.15: Curva de Hilbert de orden 5.

El número de llamadas recursivas realizadas, coincide con el número de líneas dibujadas y es: $4^n - 1$.

Curvas Sierpinski

Es otro procedimiento de superposición de curvas, la diferencia es que estas curvas son cerradas, el patrón es el siguiente:



En donde las líneas verticales y horizontales son del doble de tamaño. Por ejemplo, para las **Curvas Sierpinski** de orden 1 y 2 se muestran en la **Figura 4.16**.

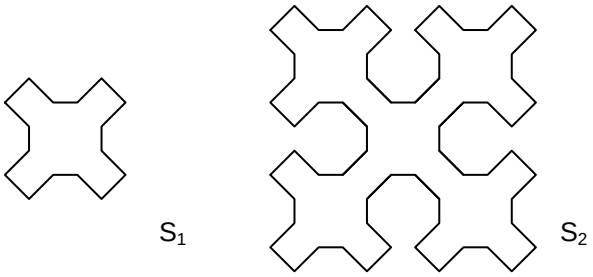


Figura 4.16: **Curvas Sierpinski** de orden 1 y 2.

El **Programa 4.13** muestra, como se codifica en C, este procedimiento.

```
#include "graphics.h"

int u,x0,y0;
void a(int);
void b(int);
void c(int);
void d(int);

void linea(int dir,int tam){
int x1,y1;
if(dir == 0){
      x1 = x0 + tam;
      y1 = y0;
      }
if(dir == 1){
      x1 = x0 + tam;
      y1 = y0 - tam;
      }
if(dir == 2){
      x1 = x0;
      y1 = y0 - tam;
      }
if(dir == 3){
      x1 = x0 - tam;
      y1 = y0 - tam;
      }
if(dir == 4){
      x1 = x0 - tam;
      y1 = y0;
      }
if(dir == 5){
      x1 = x0 - tam;
      y1 = y0 + tam;
      }
if(dir == 6){
      x1 = x0;
      y1 = y0 + tam;
      }
```

Programa 4.13: Curvas de Sierpinski (Parte **1**/3).

```
if(dir == 7){
      x1 = x0 + tam;
      y1 = y0 + tam;
      }
line(x0,y0,x1,y1);
x0 = x1;
y0 = y1;
}

void a(int k){
if(k > 0){
      a(k - 1);
      linea(7,u);
      b(k - 1);
      linea(0,2 * u);
      d(k - 1);
      linea(1,u);
      a(k - 1);
      }
}

void b(int k){
if(k > 0){
      b(k - 1);
      linea(5,u);
      c(k - 1);
      linea(6,2 * u);
      a(k-  1);
      linea(7,u);
      b(k - 1);
      }
}

void c(int k){
if(k > 0){
      c(k - 1);
      linea(3,u);
      d(k - 1);
      linea(4,2 * u);
      b(k - 1);
      linea(5,u);
      c(k - 1);
      }
}
void d(int k){
if(k > 0){
      d(k - 1);
      linea(1,u);
      a(k - 1);
      linea(2,2 * u);
      c(k - 1);
      linea(3,u);
      d(k - 1);
      }
}
```

Programa 4.13: Curvas de Sierpinski (Parte **2**/3).

```
void main(){
int graphdriver = DETECT,graphmode;
initgraph(&graphdriver,&graphmode,"");
cleardevice();
u = 15;
x0 = 50;
y0 = 70;
a(5);
linea(7,u);
b(5);
linea(5,u);
c(5);
linea(3,u);
d(5);
linea(1,u);
getch();
cleardevice();
closegraph();
}
```

Programa 4.13: Curvas de Sierpinski (Parte **3**/3).

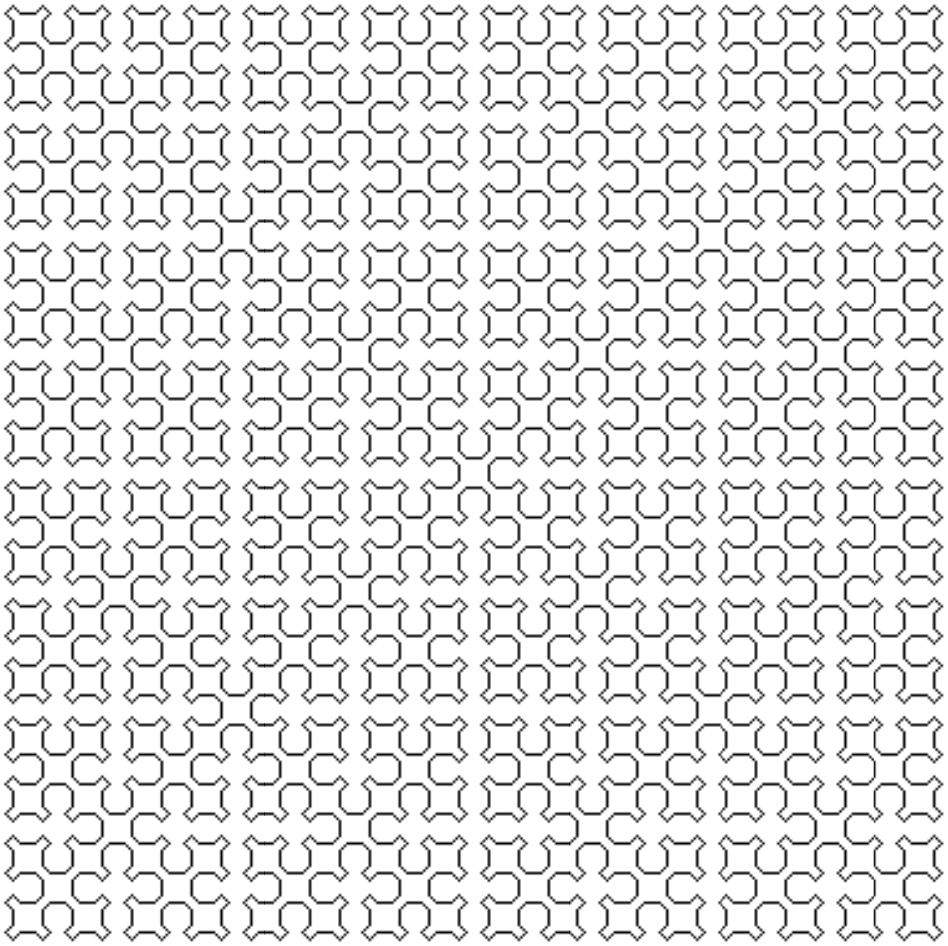


Figura 4.17: Curva Sierpinski de orden 5.

El número de llamadas recursivas realizadas, coincide con el número de líneas dibujadas y es: 4^{n+1}.

4.4.13 El problema de las ocho Reinas

Este es un **algoritmo de tanteo** o **rastreo inverso**. El problema es que hay que colocar ocho reinas en un tablero de ajedrez en una forma en que ninguna reina se coma a otra. El **Programa 4.14** muestra como es posible obtener todas las soluciones a este problema (**92** en total). Y en el **Programa 4.15** se observa una segunda solución a este problema; ambas soluciones realizan el mismo número de llamadas recursivas.

```
#include "stdio.h"
#include "math.h"

int a[10],b[20],c[20],x[10];

/* Busca las soluciones de forma recursiva */
void reina(int i){
int i,j;
for(j = 1;j <= 8;j++){
      if(a[j] && b[i+j] && c[i – j + 8]){
            x[i] = j;
            a[j] = 0;
            b[i + j] = 0;
            c[i – j + 8] = 0;
            if(i < 8) reina(i + 1);
            else for(i = 1;i <= 8;i++) printf(" %d",x[i]);
            a[j] = 1;
            b[i+j] = 1;
            c[i – j + 8] = 1;
            }
      }
}

/* Programa Principal */
void main(){
int i;
for(i = 0;i <= 9;i++) a[i] = 1;
for(i = 0;i <= 18;i++) {
      b[i] = 1;
      c[i] = 1;
      }
reina(1);
}
```

Programa 4.14: Problema de las ocho Reinas.

El programa entrega las posiciones en la fila correspondiente del tablero, las primeras seis soluciones se tienen en la **Figura 4.18**. También se observa la distribución de las reinas en el tablero.

```
1:    1 5 8 6 3 7 2 4
2:    1 6 8 3 7 4 2 5
3:    1 7 4 6 8 2 5 3
4:    1 7 5 8 2 4 6 3
5:    2 4 6 8 3 1 7 5
6:    2 5 7 1 3 8 6 4
```

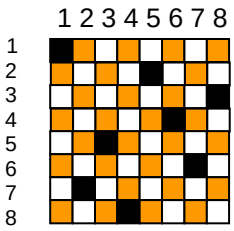


Figura 4.18: Soluciones al problema de las ocho Reinas.

```
#include "stdio.h"
#include "conio.h"
#define     FALSE 0
#define     TRUE  1
#define     TT    8     /* Celdas del Tablero                              */

/* Variables Globales                                                      */
int tablero[TT][TT];

/* Válida una posición                                                     */
int valido(){
int i,j,k,l,m;
for(i = 0;i < TT;i++){
      for(j = 0;j < TT;j++){
            if(tablero[i][j]){
                  for(k = 0;k < TT;k++){
                        if(tablero[i][k] && k != j)
                              return(FALSE);
                        if(tablero[k][j] && k != i)
                              return(FALSE);
                              }
                        l = i + 1;
                        m = j + 1;
                        while(l < TT && m < TT){
                              if(tablero[l][m])
                                    return(FALSE);
                              l++;
                              m++;
                              }
                        l = i + 1;
                        m = j - 1;
                        while(l < TT && m > 0){
                        if(tablero[l][m])
                              return(FALSE);
                        l++;
                        m--;
                        }
                  }
            }
      }
return(TRUE);
}
```

Programa 4.15: Problema de las ocho Reinas (Parte **1**/2).

```
/* Imprime el tablero de salida                                          */
void salida(){
int i,j;
for(i = 0;i < TT;i++){
      for(j = 0;j < TT;j++){
            if(tablero [i][j])printf("R ");
            else  cprintf("* ");
            }
      printf("\n");
      }
}

/* Busca la posición de las reinas en el tablero                         */
int reinas(int n){
int i;
for(i = 0;i < TT;i++) {
      tablero[n][i] = TRUE;
      if(n == TT - 1 && valido()) return TRUE;
      if(valido() && reinas(n + 1)) return TRUE;
      tablero[n][i] = FALSE;
      if(i == TT - 1) return(FALSE);
      }
return(TRUE);
}

/* Programa Principal                                                    */
void main(){
int i,j;
textbackground(1);
textcolor(14);
clrscr();
for(i = 0;i < TT;i++)
      for (j = 0;j < TT;j++)
            tablero[i][j] = FALSE;
if(reinas(0)) salida();
}
```

Programa 4.15: Problema de las ocho Reinas (Parte **2**/2).

El programa entrega las posiciones en el tablero (**Figura 4.19**).

```
R * * * * * * *
* * * * R * * *
* * * * * * * R
* * * * * R * *
* * R * * * * *
* * * * * * R *
* R * * * * * *
* * * R * * * *
```

Figura 4.19: Solución al **problema de las ocho Reinas**.

4.4.14 Contador de palabras

Un árbol binario puede ser empleado como **contador de palabras** de un archivo; al mismo tiempo es posible encontrar la frecuencia de cada una de esas palabras, con la finalidad, por ejemplo de hacer un índice. El **Programa 4.16** determina cuantas veces aparece una palabra en un archivo y cuantas palabras diferentes tiene.

```
#define MAX 80

/* Estructura de un nodo del árbol                                    */
typedef struct nodo arbol;
struct nodo{
int conteo;
char dato[MAX];
arbol *izquierdo;
arbol *derecho;
};

/* Mensaje de ERROR                                                   */
void error(void){
perror("\n\t\aERROR: Memoria insuficiente...");
exit(1);
}

/* Crea un nuevo nodo del tipo de la estructura                       */
arbol *Nuevo(){
arbol *q = (arbol *)malloc(sizeof(arbol));
if(!q) error();
return(q);
}

/* Cuenta el número de nodos en el árbol                              */
int num_nodos(arbol *p){
if(p == NULL) return(0);
return(num_nodos(p -> izquierdo) + 1 + num_nodos(p -> derecho));
}

/* Busca un dato en el árbol                                          */
char *buscar(arbol *p,char dato[]){
if(p == NULL) return(NULL);
if(strcmp(dato,p -> dato) == 0) return("NO SE ENCONTRÓ LA PALABRA");
if(strcmp(dato,p -> dato) < 0) return(buscar(p -> izquierdo,dato));
else return(buscar(p -> derecho,dato));
}

/* Busca el dato mínimo en el árbol                                   */
char *buscar_min(arbol *p){
if(p == NULL) return(NULL);
else if(p -> izquierdo == NULL) return(p -> dato);
else return(buscar_min(p -> izquierdo));
}
```

Programa 4.16: Contador de Palabras (Parte **1**/4).

```
/* Busca el dato máximo en el árbol                                    */
char *buscar_max(arbol *p){
if(p == NULL) return(NULL);
else if(p -> derecho == NULL) return(p -> dato);
else return(buscar_max(p -> derecho));
}

/* Recorre y muestra todos los nodos                                   */
void ver(int nivel,arbol *p){
int i;
if(p != NULL) {
      ver(nivel + 1,p -> derecho);
      printf("\n");
      for(i = 0;i < nivel;i++) printf("       ");
      printf("%s(%d)",p -> dato,p -> conteo);
      ver(nivel + 1,p -> izquierdo);
      }
}

/* Recorre el árbol en Pre-Orden                                       */
void pre_orden(arbol *p){
if(p != NULL){
      printf("%s(%d) ",p -> dato,p -> conteo);
      pre_orden(p -> izquierdo);
      pre_orden(p -> derecho);
      }
}

/* Recorre el árbol e incrementa el contador                           */
void contador(arbol *p,char dato[]){
if(p != NULL){
      if(strcmp(p -> dato,dato) == 0) p -> conteo++;
      contador(p -> izquierdo,dato);
      contador(p -> derecho,dato);
      }
}

/* Insertar un nuevo nodo en el árbol                                  */
arbol *insertar(char dato[],arbol *p){
if(buscar(p,dato) != NULL) {
      contador(p,dato);
      return(p);
      }
if(p == NULL){
      p = Nuevo();
      strcpy(p -> dato,dato);
      p -> conteo = 1;
      p -> izquierdo = NULL;
      p -> derecho = NULL;
      return(p);
      }
if(strcmp(dato,p->dato) < 0) p->izquierdo = insertar(dato,p->izquierdo);
else p -> derecho = insertar(dato,p -> derecho);
return(p);
}
```

Programa 4.16: Contador de Palabras (Parte **2**/4).

```
/* Pone menú                                                          */
void menu(void){
printf("A = Leer Archivo de texto");
printf("B = Buscar una Palabra");
printf("M = Buscar Mínimo y Máximo");
printf("N = Número de Palabras Diferentes");
printf("R = Ver Frecuencia de Palabras");
printf("V = Ver árbol");
printf("Q = Salir");
printf("Elija una Opción : ");
}

/* Carga datos desde el archivo de entrada                            */
void carga(arbol **p,char ent[20]){
char dato[MAX],arc;
int pos;
FILE *fp;
if((fp = fopen(ent,"r")) == NULL){
      printf("\n\tNo se pudo abrir el archivo...");
      return;
      }
arc = getc(fp);              /* Carga primer carácter                 */
while(arc != EOF){
      pos = 0;
      do dato[pos++] = arc;
      while((arc = getc(fp)) != '\n' && arc != EOF && arc != ' '
            && arc != '.' && arc != ',');
      dato[pos] = '\0';
      arc = getc(fp);        /* Salta salto de línea */
      *p = insertar(dato,*p);
      if(arc == '\n' || arc == '.' || arc == ','
                     || arc==' ' && arc != EOF) arc=getc(fp);
      }
fclose(fp);
}

/* Programa principal                                                 */
void main(void){
int i,pos;
char op,dato[MAX],n[20];
arbol *p = NULL;       /* Árbol Vacío                                 */
while(1){
      menu();
      op = tolower(getch());
      switch(op){
            case 'a':
                  printf("Nombre del archivo a leer: ");
                  scanf("%s",&n);
                  carga(&p,n);
                  break;
            case 'v':
                  printf("Árbol Binario ");
                  ver(0,p);
                  break;
```

Programa 4.16: Contador de Palabras (Parte **3**/4).

```
            case 'm':
                  printf("\n\n\tMínimo = %s",buscar_min(p));
                  printf("\n\n\tMáximo = %s",buscar_max(p));
                  break;
            case 'n':
                  printf("\n\n\tNúmero de Palabras: %d",num_nodos(p));
                  break;
            case 'b':
                  printf("\n\tPalabra a Buscar : ");
                  pos = 0;
                  while((dato[pos++] = getchar()) !='\n');
                  dato[--pos] = '\0';
                  if(buscar(p,dato) != NULL)
                  printf("\n\tSe encontró la palabra %s",dato);
                  else printf("\n\t\aNo se encontró la palabra %s",dato);
                  break;
            case 'r':
                  printf("Palabra(Frecuencia)");
                  printf("Recorrido en Pre-Orden:");
                  printf("\n\t");
                  pre_orden(p);
                  break;
            case 'q':
                  exit(1);
                  break;
            }
      getch();
      }
}
```

Programa 4.16: Contador de Palabras (Parte **4**/4).

En la **Figura 4.20**, se observa un ejemplo del **Programa 4.16** para el archivo de entrada **`ent.txt`** cuyo contenido es:

```
Érase de un marinero
que hizo un jardín junto al mar,
y se metió a jardinero.
Estaba el jardín en flor,
y el jardinero se fue
por esos mares de Dios.
```

```
Contador de Palabras con un Árbol Binario
A = Leer Archivo de texto
B = Buscar una Palabra
M = Buscar Mínimo y Máximo
N = Número de Palabras Diferentes
R = Ver Frecuencia de Palabras
V = Ver Árbol
Q = Salir
Elija una Opción: A
Nombre del archivo a leer: ent.txt
>B
Palabra a Buscar: se
Se encontró la palabra se
>M
Mínimo = Dios
Máximo = Érase
>N
Número de Palabras Diferentes: 24
>R
Palabra(Frecuencia)
Recorrido en Pre-Orden:
Érase(1) de(2) al(1) a(1) Estaba(1) Dios(1) un(2) marinero(1) hizo(1) el(2)
en(1) flor(1) esos(1) fue(1) jardín(2) jardinero(2) junto(1) mar(1) mares(1)
que(1) metió(1) por(1) se(2) y(2)
>V
Érase(1)
                    y(2)
              un(2)
                                  se(2)
                            que(1)
                                        por(1)
                                  metió(1)
                    marinero(1)
                                                    mares(1)
                                              mar(1)
                                        junto(1)
                                  jardín(2)
                                        jardinero(2)
                            hizo(1)
                                                    fue(1)
                                              flor(1)
                                                    esos(1)
                                        en(1)
                                  el(2)
      de(2)
              al(1)
                    a(1)
                            Estaba(1)
                                  Dios(1)
```

Figura 4.20: Ejemplo del programa contador de palabras.

4.4.15 Códigos Huffman

La **compresión de datos** consiste en la transformación de una cadena de caracteres de cierto alfabeto en una nueva cadena que contiene la misma información, pero cuya longitud es más pequeña que la original. El **código de Huffman** es un **código** que representa de forma única todo carácter de un texto de entrada.

Un **código**, transforma **mensajes fuente** a **palabras código**, en un proceso conocido como **codificación**; la **decodificación** es la transformación de **palabras código** a **mensajes fuente**.

Un **código de longitud fija**, consiste de **palabras código** que contienen una longitud fija, por ejemplo el código **ASCII**, en donde sus **256** caracteres tienen una longitud de **8** bits. Se puede reducir significativamente la cantidad de bits empleados si se emplea un **código de longitud variable**; es decir, que el número de bits varié de carácter a carácter. Si se emplea un código de longitud variable, es necesario colocar un bit de **inicio** y **final** para cada carácter, sin embargo, está solución puede aumentar la cantidad de bits empleados, ó una palabra puede aparecer como prefijo de otra. Otra solución es tener una única secuencia de caracteres que asegure que ninguna palabra es prefijo de otra, un código así, se conoce como **instantáneamente decodificable**.

Un **algoritmo voraz ó ávido** que emplea una cola de prioridad para crear prefijos óptimos para un mensaje es el **árbol de Huffman** y el correspondiente código de prefijos se conoce como **código de Huffman**. La idea básica consiste en crear de abajo hacia arriba un árbol binario, obteniendo un árbol más grande a través de dos árboles pequeños, del bosque con prioridades mayores. La prioridad de un árbol viene dada por la suma de las frecuencias de los caracteres almacenados en sus hojas.

Por ejemplo, para el conjunto de caracteres **a**, **b**, **c**, **d**, **e** y **f**, se tienen las frecuencias **30**, **10**, **7**, **8**, **40** y **14**, para generar el **código de Huffman** de este conjunto se sigue el procedimiento de la **Figura 4.21** y la implementación se muestra en el **Programa 4.17** y la **Figura 4.22**.

En la **Figura 4.21** se observa en **a)** el conjunto inicial con **6** subárboles de un único nodo. La prioridad de cada uno se muestra dentro del nodo como la frecuencia de cada carácter. De **b)** a **e)** se muestran los subárboles obtenidos durante el proceso y en **f)** se tiene el árbol final. Los valores binarios que se encuentran en cada camino desde la raíz hasta una hoja son la **palabra código** del carácter almacenado en la hoja, y se muestran a continuación.

Carácter	**e**	Código: `0`
Carácter	**a**	Código: `10`
Carácter	**c**	Código: `1100`
Carácter	**d**	Código: `1101`
Carácter	**b**	Código: `1110`
Carácter	**f**	Código: `1111`

Por ejemplo para la palabra **aba** tenemos la codificación:

ASCII:	`01100001 01100010 01100001`	**24 bits**
Huffman:	`10 1110 10`	**8 bits**

Para **decodificar el código Huffman** es necesario conocer el árbol generado, luego seguir desde la raíz la palabra codificada hasta encontrar una hoja, decodificar el carácter y continuar con la palabra hasta completarla.

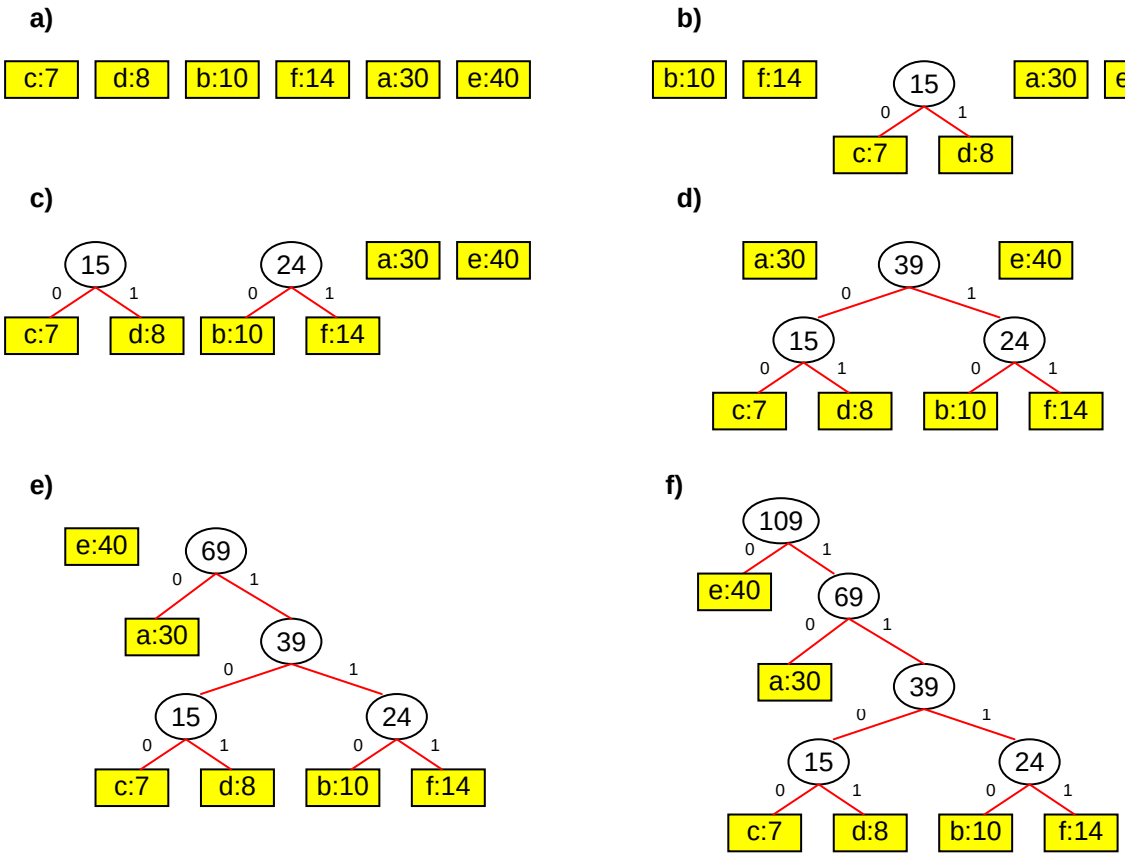


Figura 3.21: Generación del árbol para obtener el **código Huffman**.

```
#include "stdio.h"
#include "string.h"

/* Estructura de un nodo del árbol                                    */
typedef struct nodo arbol;
struct nodo{
int frecuencia;
char dato;
arbol *derecho;
arbol *izquierdo;
};

/* Mensaje de ERROR                                                   */
void error(void){
perror("\n\t\aERROR: Memoria insuficiente...");
exit(1);
}

/* Crea un nuevo nodo del tipo de la estructura                       */
arbol *Nuevo(){
arbol *q = (arbol *)malloc(sizeof(arbol));
if(!q) error();
return(q);
}
```

Programa 4.17: **Código Huffman** (Parte **1**/3).

```
/* Inserta un nuevo nodo en el árbol                                    */
arbol *insertar(char dato, int x){
arbol *q;
q = Nuevo();
q -> frecuencia = x;
q -> dato = dato;
q -> derecho = q -> izquierdo = NULL;
return(q);
}

/* Ordena los nodos del árbol de acuerdo a su frecuencia                */
void ordenar(arbol *a[], int n){ /* Ripple Sort                         */
int i, j;
arbol *temp;
for (i = 0; i < n - 1; i++)
      for (j = i; j < n; j++)
            if (a[i] -> frecuencia > a[j] -> frecuencia){
                  temp = a[i];
                  a[i] = a[j];
                  a[j] = temp;
                  }
}

/* Asigna un código a cada carácter                                     */
void asigna_codigo(arbol *p, int c[], int n){
int i;
if ((p -> izquierdo == NULL) && (p -> derecho == NULL)){
      printf("\n\t%c Código : ", p -> dato);
      for (i = 0; i < n; i++) printf("%d", c[i]);
      }
else  {
      c[n] = 0;
      n++;
      asigna_codigo(p -> izquierdo,c,n);
      c[n - 1] = 1;
      asigna_codigo(p -> derecho,c,n);
      }
}

/* Borra el árbol creado                                                */
void borra_arbol(arbol *p){
if(p != NULL) {
      borra_arbol(p -> izquierdo);
      borra_arbol(p -> derecho);
      free(p);
      }
}

/* Programa Principal                                                   */
void main(void){
arbol *p1;
int i,j,n,u,c[20];
char dato;
arbol *a[10];     /* Hasta 10 subárboles de inicio                      */
int frecuencia;
clrscr();
```

Programa 4.17: Código Huffman (Parte **2**/3).

```
printf("Código de Huffman");
printf("\n\n\tEntra el número de caracteres a codificar: ");
scanf("%d",&n);
for (i = 0; i < n; i++) {
      printf("\n\tEntra Carácter y Frecuencia: ");
      dato = getche();
      scanf("%d",&frecuencia);
      a[i] = insertar(dato,frecuencia);
      }
while (n > 1){
      ordenar(a, n);
      u = a[0] -> frecuencia + a[1] -> frecuencia;
      dato = a[1] -> dato;
      p1 = insertar(dato,u);
      p1 -> derecho = a[1];
      p1 -> izquierdo = a[0];
      a[0] = p1;
      for (j = 1;j < n - 1; j++) a[j] = a[j+1];
      n--;
      }
asigna_codigo(a[0],c,0);
getch();
borra_arbol(a[0]);
}
```

Programa 4.17: Código Huffman (Parte **3**/3).

```
        Código de Huffman
Entra el número de caracteres a codificar: 6
Entra Carácter y Frecuencia: a 30
Entra Carácter y Frecuencia: b 10
Entra Carácter y Frecuencia: c 7
Entra Carácter y Frecuencia: d 8
Entra Carácter y Frecuencia: e 40
Entra Carácter y Frecuencia: f 14
e Código: 0
a Código: 10
c Código: 1100
d Código: 1101
b Código: 1110
f Código: 1111
```

Figura 3.22: Código Huffman obtenido para los caracteres `a, b, c, d, e` y `f`.

4.4.16 Evaluación de expresiones

En está sección se muestra como es posible evaluar una **expresión postfija**. Para no complicar demasiado el problema, se considera como entrada una **expresión postfija**, con **A, B** y **C** como **variables** y con **valores** de cada una comprendidos entre **0** a **9**. Este método, almacena la expresión en un árbol binario, y emplea una pila para realizar el recorrido de izquierda a derecha para evaluar la expresión.

Por ejemplo, para evaluar la expresión **`5*(7+3)-2`**, podemos representarla mediante el árbol de la **Figura 4.23**. La forma postfija es: **`5 7 3 + * 2 -.`**

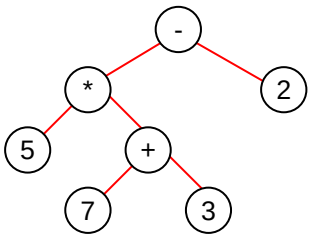


Figura 4.23: Árbol de expresiones para evaluación de una expresión.

El **Programa 4.18**, realiza las siguientes acciones para evaluar la expresión:

- Lee la expresión postfija y crea el árbol binario.
- Imprime el árbol.
- Lee los valores de **A, B** y **C**.
- Evalúa la expresión postfija.

```
#include "stdio.h"
#include "conio.h"
#include "malloc.h"
#include "dos.h"
#include "ctype.h"
#include "stdlib.h"

/* Pila                                                              */
#define MAX             100
#define TAM(p_S)        ((p_S) -> tope - (p_S) -> base)
/* Árbol                                                             */
#define DATO(T)         ((T) -> p_dato)
#define IZQ(T)          ((T) -> izq)
#define DER(T)          ((T) -> der)
#define TIPO(n)         ((n) -> nodo_t)
/* Evaluación                                                        */
#define VAL_OP(n)       ((n) -> nodo_val.tipo_op)
#define VAL_SIM(n)      ((n) -> nodo_val.nomb_sim)
#define VAL_CON(n)      ((n) -> nodo_val.val_con)
#define PERROR(var)     if(var == ERROR) return ERROR;

/* Definiciones de tipo                                              */
typedef enum{PRE_ORDEN,EN_ORDEN,POST_ORDEN} ORDEN;
typedef enum{OPERADOR,CONSTANTE,VARIABLE} Nodo_T;
typedef enum{SUMA,RESTA,MULT,DIVISION} Operador;
typedef enum{OPERADOR_T,VARIABLE_T,CONSTANTE_T,EOL_T} tipo_ent;
typedef enum{OK,ERROR} estado;
typedef char *tipo;
typedef enum{FALSE = 0,TRUE = 1} bool;

/* Estructura de la expresión                                        */
typedef struct dato_act{
tipo_ent tipo_act;
int valor_act;
}dato_act;

/* Tipo de datos actuales                                            */
dato_act *lee_act();
```

Programa 4.18: Evaluación de expresiones (Parte **1**/9).

```
/* Estructura de la Pila                                                */
typedef struct{
tipo base[MAX];
tipo *tope;
}pila;

/* Pila Vacía                                                           */
bool pila_vacia(pila *p_S){
return (p_S -> tope == p_S -> base) ? TRUE : FALSE;
}

/* Inicia Pila                                                          */
estado inicia_pila(pila *p_S){
p_S -> tope = p_S -> base;
return OK;
}

/* Inserta un dato en la Pila                                           */
estado push(pila *p_S,tipo dato){
if(TAM(p_S) == MAX) return ERROR;
*p_S -> tope = dato;
p_S -> tope++;
return OK;
}

/* Suprime un dato de la Pila                                           */
estado pop(pila *p_S,tipo *p_data){
if(pila_vacia(p_S) == TRUE) return ERROR;
p_S -> tope--;
*p_data = *p_S -> tope;
return OK;
}

/* Prototipo de la tabla de símbolos                                    */
typedef struct Tabla_Sim{
int A,B,C;
}Tabla_Sim;

/* Nodo de la expresión                                                 */
typedef struct nodo_expr{
Nodo_T nodo_t;
union {
      Operador tipo_op;
      char nomb_sim;
      int val_con;
      }nodo_val;
}nodo_expr;

/* Estructura del árbol                                                 */
typedef struct nodo_arbol nodo_arbol, *arbol;
struct nodo_arbol{
tipo p_dato;
arbol izq;
arbol der;
};
```

Programa 4.18: Evaluación de expresiones (Parte **2**/9).

```
/* Inserta un nodo en el árbol */
estado insertar(arbol *p_T,tipo dato){
arbol T = (arbol)malloc(sizeof(nodo_arbol));
if(T == NULL) return ERROR;
*p_T = T;
DATO(T) = dato;
IZQ(T) = NULL;
DER(T) = NULL;
return OK;
}

/* Inicia árbol                                                    */
estado inicia_arbol(arbol *p_T){
*p_T = NULL;
return OK;
}

/* Árbol Vacío                                                     */
bool arbol_vacio(arbol T){
return (T == NULL) ? TRUE : FALSE;
}

/* Coloca un nodo como raíz                                        */
estado raiz_arbol(arbol *p_T,tipo dato,arbol izq,arbol der){
if(arbol_vacio(*p_T) == FALSE) return ERROR;
if(insertar(p_T,dato) == ERROR) return ERROR;
IZQ(*p_T) = izq;
DER(*p_T) = der;
return OK;
}

/* Inserta un nodo en el árbol                                     */
estado inserta_arbol(pila *p_S,arbol T){
arbol *p_T = (arbol *)malloc(sizeof(arbol));
if(p_T == NULL) return ERROR;
*p_T = T;
if(push(p_S,(tipo) p_T) == ERROR) {
      free(p_T);
      return ERROR;
      }
return OK;
}

/* Borra un nodo en el árbol                                       */
estado suprimir_arbol(pila *p_S,arbol *p_T){
arbol *ptr;
if(pop(p_S,(tipo *) &ptr) == ERROR) return ERROR;
*p_T = *ptr;
free(ptr);
return OK;
}

/* Almacena una Constante                                          */
nodo_expr *almacena_constante(int valor){
nodo_expr *p_expr = (nodo_expr *)malloc(sizeof(nodo_expr));
```

Programa 4.18: Evaluación de expresiones (Parte **3**/9).

```
if(p_expr != NULL){
      TIPO(p_expr) = CONSTANTE;
      VAL_CON(p_expr) = valor;
      }
return p_expr;
}

/* Almacena una Variable                                                */
nodo_expr *almacena_simbolo(char valor){
nodo_expr *p_expr = (nodo_expr *)malloc(sizeof(nodo_expr));
if(p_expr != NULL){
      TIPO(p_expr) = VARIABLE;
      VAL_SIM(p_expr) = valor;
      }
return p_expr;
}

/* Almacena un Operador                                                 */
nodo_expr *almacena_operador(Operador valor){
nodo_expr *p_expr = (nodo_expr *)malloc(sizeof(nodo_expr));
if(p_expr != NULL){
      TIPO(p_expr) = OPERADOR;
      VAL_OP(p_expr) = valor;
      }
return p_expr;
}

/* Lee expresión                                                        */
estado lee_expr(arbol *p_T){
pila S;
arbol temp_a,h_der,h_izq;
nodo_expr *p_expr;
estado rc;
dato_act *p_act;
inicia_pila(&S);
while(TRUE){
      p_act = lee_act();
      switch(p_act -> tipo_act){
            case EOL_T:
                  rc = suprimir_arbol(&S,p_T);
                  PERROR(rc);
                  return OK;
            case CONSTANTE_T:
            case VARIABLE_T:
                  if(p_act -> tipo_act == CONSTANTE_T){
                        char aux1[2];
                        aux1[0] = (char) p_act -> valor_act;
                        aux1[1] = '\0';
                        p_expr = almacena_constante(atoi(aux1));
                        }
                  else  p_expr = almacena_simbolo(p_act -> valor_act);
                  if(p_expr == NULL) return ERROR;
                  rc = inicia_arbol(&temp_a);
                  PERROR(rc);
                  rc = raiz_arbol(&temp_a,(tipo)p_expr,NULL,NULL);
                  PERROR(rc);
```

Programa 4.18: Evaluación de expresiones (Parte **4**/9).

```
                rc = inserta_arbol(&S,temp_a);
                PERROR(rc);
                break;
          case OPERADOR_T:
          switch(p_act -> valor_act){
                case '+':
                      p_expr = almacena_operador(SUMA);
                      break;
                case '-':
                      p_expr = almacena_operador(RESTA);
                      break;
                case '*':
                      p_expr = almacena_operador(MULT);
                      break;
                case '/':
                      p_expr = almacena_operador(DIVISION);
                      break;
                default:
                      return ERROR;
                      }
                if(p_expr == NULL) return ERROR;
                rc = suprimir_arbol(&S,&h_der);
                PERROR(rc);
                rc = suprimir_arbol(&S,&h_izq);
                PERROR(rc);
                rc = inicia_arbol(&temp_a);
                PERROR(rc);
                rc = raiz_arbol(&temp_a,(tipo)p_expr,h_izq,h_der);
                PERROR(rc);
                rc = inserta_arbol(&S,temp_a);
                PERROR(rc);
                break;
          default:
                return ERROR;
          }
     }
}

/* Recorrido en preorden                                                    */
static estado r_preorden(arbol T,estado (*p_func_f)()){
estado rc;
if(arbol_vacio(T) == TRUE) return OK;
rc = (*p_func_f)(DATO(T));
if(rc == OK) rc = r_preorden(IZQ(T),p_func_f);
if(rc == OK) rc = r_preorden(DER(T),p_func_f);
return rc;
}

/* Recorrido en orden                                                       */
static estado r_enorden(arbol T,estado (*p_func_f)()){
estado rc;
if(arbol_vacio(T) == TRUE) return OK;
rc = r_enorden(IZQ(T),p_func_f);
if(rc == OK) rc = (*p_func_f)(DATO(T),p_func_f);
if(rc == OK) rc = r_enorden(DER(T),p_func_f);
return rc;
}
```

Programa 4.18: Evaluación de expresiones (Parte **5**/9).

```
/* Recorrido en postorden                                                */
static estado r_postorden(arbol T,estado (*p_func_f)()){
estado rc;
if(arbol_vacio(T) == TRUE) return OK;
rc = r_postorden(IZQ(T),p_func_f);
if(rc == OK) rc = r_postorden(DER(T),p_func_f);
if(rc == OK) rc = (*p_func_f)(DATO(T));
return rc;
}

/* Selecciona tipo de recorrido en el árbol                              */
estado recorrido(arbol T,estado (*p_func_f)(),ORDEN orden){
switch(orden){
      case PRE_ORDEN:
            return r_preorden(T,p_func_f);
      case EN_ORDEN:
            return r_enorden(T,p_func_f);
      case POST_ORDEN:
            return r_postorden(T,p_func_f);
      }
return ERROR;
}

/* Imprime un nodo de la expresión                                       */
estado imprime_nodo(nodo_expr *p_expr){
char c;
switch(TIPO(p_expr)){
      case OPERADOR:
            switch(VAL_OP(p_expr)){
                  case SUMA:
                        c = '+';
                        break;
                  case RESTA:
                        c = '-';
                        break;
                  case MULT:
                        c = '*';
                        break;
                  case DIVISION:
                        c = '/';
                        break;
                  }
            printf("%c ",c);
            break;
      case CONSTANTE:
            printf("%d ",VAL_CON(p_expr));
            break;
      case VARIABLE:
            printf("%c ",VAL_SIM(p_expr));
            break;
      }
return OK;
}
```

Programa 4.18: Evaluación de expresiones (Parte **6**/9).

```
/* Imprime expresión                                                    */
estado print_expr(arbol T){
printf("\tLa expresión (Pre_Orden)  es : ");
recorrido(T,imprime_nodo,PRE_ORDEN);
putchar('\n');
printf("\tLa expresión (En_Orden)   es : ");
recorrido(T,imprime_nodo,EN_ORDEN);
putchar('\n');
printf("\tLa expresión (Post_Orden) es : ");
recorrido(T,imprime_nodo,POST_ORDEN);
putchar('\n');
return OK;
}

/* Asigna el valor a cada símbolo                                       */
int carga_simbolos(Tabla_Sim Tab_simbolos,char sim){
if(sim == 'A' )   return Tab_simbolos.A;
if(sim == 'B' ) return Tab_simbolos.B;
if(sim == 'C' ) return Tab_simbolos.C;
return 0;
}

/* Lee símbolos desde el teclado                                        */
int lee_simbolos(Tabla_Sim *sim){
printf("\n\tEntra A B y C : ");
scanf("%d %d %d",&sim -> A,&sim -> B,&sim -> C);
return;
}

/* Evaluación de la expresión                                           */
estado evalua(arbol T,Tabla_Sim Tab_simbolos,int *p_valor){
estado rc;
nodo_expr *p_expr;
int valor_der,valor_izq;
if(arbol_vacio(T) == TRUE) *p_valor = 0;
else  {
      p_expr = (nodo_expr *)DATO(T);
      switch(TIPO(p_expr)){
            case CONSTANTE:
                  *p_valor = VAL_CON(p_expr);
                  break;
            case VARIABLE:
                  *p_valor =
                  carga_simbolos(Tab_simbolos,VAL_SIM(p_expr));
                  break;
            case OPERADOR:
                  rc = evalua(IZQ(T),Tab_simbolos,&valor_der);
                  PERROR(rc);
                  rc = evalua(DER(T),Tab_simbolos,&valor_izq);
                  PERROR(rc);
                  switch(VAL_OP(p_expr)){
                        case SUMA:
                              *p_valor = valor_der + valor_izq;
                              break;
                        case RESTA:
                              *p_valor = valor_der - valor_izq;
                              break;
```

Programa 4.18: Evaluación de expresiones (Parte **7**/9).

```
                        case MULT:
                              *p_valor = valor_der * valor_izq;
                              break;
                        case DIVISION:
                              if(valor_izq == 0) return ERROR;
                              *p_valor = valor_der / valor_izq;
                              break;
                        }
                  break;
            }
      }
return OK;
}

/* Analizador léxico                                                    */
dato_act *lee_act(){
static dato_act actual;
static char aux1[100],*p_aux1;
if(p_aux1 == 0){
      if((p_aux1 = gets(aux1)) == NULL){
            aux1[0]='\0';
            p_aux1 = aux1;
            }
      }
while(*p_aux1 != '\0' && isspace(*p_aux1)) p_aux1++;
if(*p_aux1 == '\0') actual.tipo_act = EOL_T;
else if(isdigit(*p_aux1)) actual.tipo_act = CONSTANTE_T;
else if(es_operador(*p_aux1)) actual.tipo_act = OPERADOR_T;
else actual.tipo_act = VARIABLE_T;
actual.valor_act = toupper(*p_aux1);
p_aux1++;
return &actual;
}

/* Verifica si es un operador                                           */
es_operador(char c){
return(c == '*' || c == '/' || c == '+' || c == '-');
}

/* Limpia pantalla                                                      */
void limpia(void){
clrscr();
gotoxy(10,22);
cprintf("Presione S para salir u otra tecla para continuar.");
gotoxy(15,1);
cprintf("Evaluador de Expresiones Postfijas con árboles binarios\n\t");
}

/* Programa principal                                                   */
void main(void){
int resultado;
arbol T;
Tabla_Sim Tab_simbolos;
char op;
limpia();
```

Programa 4.18: Evaluación de expresiones (Parte **8**/9).

```
printf("Entra la expresión postfija  : ");
if(lee_expr(&T) == ERROR){
      perror("Error de lectura");
      exit(-1);
      }
print_expr(T);
do    {
      lee_simbolos(&Tab_simbolos);
      evalua(T,Tab_simbolos,&resultado);
      printf("\tLa evaluaci¢n es : %d\n",resultado);
      printf("\tOpci¢n : ");
      op = tolower(getch());
      }while(op != 's');
printf("FIN DEL PROGRAMA");
}
```

Programa 4.18: Evaluación de expresiones (Parte **9**/9).

En la **Figura 3.30** se observan los resultados de ejecutar el **Programa 4.18:**.

```
      Evaluador de Expresiones Postfijas con árboles binarios
Entra la expresión postfija : 5 7 3 + * 2 -
La expresión (Pre_Orden)  es: - * 5 + 7 3 2
La expresión (En_Orden)   es: 5 * 7 + 3 - 2
La expresión (Post_Orden) es: 5 7 3 + * 2 -
Entra A B y C: 1 1 1
La evaluación es: 48
Opción: S

      Evaluador de Expresiones Postfijas con árboles binarios
Entra la expresión postfija : a b * c a * b + +
La expresión (Pre_Orden)  es: + * A B + * C A B
La expresión (En_Orden)   es: A * B + C * A + B
La expresión (Post_Orden) es: A B * C A * B + +
Entra A B y C: 5 4 7
La evaluación es: 59
Opción: A
Entra A B y C: 1 2 3
La evaluación es: 7
Opción: S FIN DEL PROGRAMA
```

Figura 3.30: Evaluación de expresiones.

Referencias

[1] Ceballos, Francisco J.; **"Curso de programación con C"**, Ed. Addison Wesley, U.S.A., 1991, 507pp.

[2] Chapra, Steven C. y Canale, Raymond P.; **"Métodos Numéricos para Ingenieros, con aplicaciones en computadoras personales"**, Ed. McGraw-Hill, México, 1996, 641pp.

[3] Cormen, T., Leiserson, C. y Rivest, R.; **"Introduction to Algorithms",** Ed. McGraw-Hill, U.S.A., 1990, 1027pp.

[4] Cornell, Gary; **"Manual de Visual Basic 3 para Windows"**, Ed. McGraw-Hill, México, 1995, 792pp.

[5] Deitel, H. M. y Deitel, P. J.; **"Como Programar en C++"**, Ed. Prentice Hall, México, 1998, 1130pp.

[6] Deitel, H.M.; **"Sistemas Operativos"**, Ed. Addison Wesley, 2ª ed., México, 1999, 938pp.

[7] Esakov, J. y Weiss, T.; **"Data Structures. An Advanced Approach Using C**", Ed. Prentice Hall, U.S.A., 1989, 372pp.

[8] Euán, Jorge I. y Cordero, B. Luis G.; **"Estructuras de datos**", Ed. Noriega, México, 1989, 219pp.

[9] Heileman, Gregory L.; **"Estructuras de Datos, Algoritmos y Programación Orientada a Objetos"**, Ed. McGraw-Hill, 1998, 305pp.

[10] Langsam, Y., Augenstein, M. y Tenenbaum, A.; **"Estructuras de Datos con C y C++"**, 2ª ed., Ed. Prentice Hall, México, 1997, 672pp.

[11] Schilt, Herbert; **"Turbo C, Manual de Bolsillo"**, Ed. McGraw-Hill, España, 1990, 240pp.

[12] Schilt, Herbert; **"Utilización de C en inteligencia artificial"**, Ed. McGraw-Hill, México, 1989, 340pp.

[13] Schilt, Herbert; **"Turbo C/C++. Manual de Referencia"**, Ed. McGraw-Hill, México, 1992, 874pp.

[14] Senn, James A.; **"Análisis y Diseño de Sistemas de Información"**, Ed. Mc-Graw Hill, Colombia, 1990, 643pp.

[15] Stern, Robert A. y Stern, Nancy B.; **"Principios de Procesamiento de Datos"**, Ed. Limusa, México, 1985, 726pp.

[16] Tanenbaum, Andrew S. y Woodhull, Albert S.; **"Sistemas Operativos. Diseño e Implementación"**, Ed. Prentice-Hall, 2ª ed., México, 1997, 939pp.

[17] Tenenbaum A., Langsam, Y. y Augenstein, M.; **"Estructuras de Datos en C"**, Ed. Prentice Hall, México, 1993, 696pp.

[18] Weiss, Mark Allen; **"Estructuras de datos y algoritmos"**, Ed. Addison-Wesley, U.S.A., 1995, 489pp.

[19] Wirth, Niklaus; **"Algoritmos y estructuras de datos"**, Ed. Prentice Hall, México, 1998, 305pp.