INTITUTO TECNOLOGICO SUPERIOR DE CAJEME

Manual de Referencia

Estructura de Datos

Orientado a Objetos

MANUAL DE REFERENCIA

Estructura de Datos

Orientado a Objetos

Tabla de contenido

7.1.2 Mezcla natural.

CAPÍTULO 8

MÉTODOS DE BÚSQUEDA.

8.1 Algoritmos de ordenación externa.

- 8.1.1 Secuencial.
- 8.1.2 Binaria.
- 8.1.3 Hash.

8.2 Búsqueda externa.

- 8.2.1 Secuencial.
- 8.2.2 Binaria.
- 8.2.3 Hash.

Capítulo

1

Análisis de Algoritmos

1. 1 Concepto de Complejidad de algoritmos.

1.1.1. CONCEPTOS BÁSICOS

Una posible definición de algoritmo es un conjunto de reglas que permiten obtener un resultado determinado apartir de ciertas reglas definidas.
Otra definición sería, algoritmo es una secuencia finita de instrucciones, cada una de las cuales tiene un significado preciso y puede ejecutarse con una cantidad finita de esfuerzo en un tiempo finito. Ha de tener las siguientes características: Legible, correcto, modular, eficiente, estructurado, no ambiguo y a ser posible se ha de desarrollar en el menor tiempo posible.
Características de un algoritmo de computador:

Ser algoritmo: Tiene que consistir en una secuencia de instrucciones claras y finitas.

Ser correcto: El algoritmo ha de resolver el problema planteado en todas sus facetas.

Ser legible: El Algoritmo debe de se entendible por cualquier persona que conozca el lenguaje.

Ser eficiente: Es relativa porque depende de la maquinas en la que lo ejecutemos. Existen ejemplos de algoritmos eficientes que ocupan demasiado espacio para ser aplicados sin almacenamiento secundario lento, lo cual puede anular la eficiencia.

Un algoritmo eficiente pero complicado puede ser inapropiado porque posteriormente puede tener que darle mantenimiento otra persona distinta del escritor.

1.1.2. DISEÑO DE ALGORITMOS.

Fases de diseño de algoritmos.
Diseño: se dan las especificaciones en lenguaje natural y se crea un primer modelo matemático apropiado. La solución en esta etapa es un algoritmo expresado de manera muy informal.
Implementación: El programador convierte el algoritmo en código, siguiendo alguna de estas 3 metodologías.

A. TOP-DOWN se alcanza el programa sustituyendo las palabras del palabras del pseudocódigo por secuencias de proposiciones cada vez mas detalladas, en un llamado refinamiento progresivo.
B. BOTTON-UP parte de las herramientas mas primitivas hasta que se llega al programa.
C. TAD'S modularización dependiendo de los recursos.Tenemos unas estructuras abstractas implementadas, y una serie de conocimientos asociados a esos recursos.

Pruebas: Es un material que se pasa al programa para detectar posibles errores. Esto no quiere decir que el diseño no tenga errores, puede tenerlos para otros datos.

1.1.3. COMPLEJIDAD DE ALGORITMOS.

La eficiencia de un determinado algoritmo depende de la maquina, y de otros factores externos al propio diseño. Para comparar dos algoritmos sin tener en cuenta estos factores externos se usa la complejidad. Esta es una media informativa del tiempo de ejecución de un algoritmo, y depende de varios factores:

- Los datos de entrada del programa. Dentro de ellos, lo más importante es la cantidad, su disposición, etc.
- La calidad del código generado por el compilador utilizado para crear el programa.
- La naturaleza y rapidez de las instrucciones empleados por la máquina y la propia máquina.
- La propia complejidad del algoritmo base del programa.

El hecho de que el tiempo de ejecución dependa de la entrada, indica que el tiempo de ejecución del programa debe definirse como una función de la entrada. En general la longitud de la entrada es una medida apropiada de tamaño, y se supondrá que tal es la medida utilizada a menos que se especifique lo contrario.
Se acostumbra, pues, a denominar T(n) al tiempo de ejecución de un algoritmo en función de n datos de entrada. Por ejemplo algunos programas pueden tener un tiempo de ejecución. T(n)=Cn2, donde C es una constante que engloba las características de la máquina y otros factores.
Las unidades de T(n) se dejan sin especificar, pero se puede considerar a T(n) como el número de instrucciones ejecutadas en un computador idealizado, y es lo que entendemos por complejidad.
Para muchos programas, el tiempo de ejecución es en realidad una función de la entrada específica, y no sólo del tamaño de ella. En cualquier caso se define T(n) como el tiempo de ejecución del peor caso, es decir, el máximo valor del tiempo de ejecución para las entradas de tamaño n.
Un algoritmo es de orden polinomial si T(n) crece mas despacio, a medida que aumenta n, que un polinomio de grado n. Se pueden ejecutar en un computador.

En el caso contrario se llama exponencial, y estos no son ejecutables en un computador.

3.1 Notación O grande.

Para hacer referencia a la velocidad de crecimiento de los valores de una función se usara la notación conocida como "O grande".Decimos que un algoritmo tiene un orden O(n) si existen y un n0 y un c, siendo c>0, tal que para todo n>=n0, T(n)<=c f(n).
C estará determinado por: -Calidad del código obtenido por el compilador.
-Características de la propia máquina.

Ejemplo:
T(n) = (n+1)2
n2+2n+1 <= c-n2
n >= n0
n0 = 1
Orden n2 es O(n2)

3.2 Propiedades de la notación O grande.

Si multiplicamos el orden de una función por una constante, el orden del algoritmo sigue siendo el mismo.
O(c-f(n)) = O(f(n))
La suma del orden de dos funciones es igual al orden de la mayor.
O(f(n)+g(n)) = O(máx(f(n)*g(n)).
Si multiplicamos el orden de dos funciones el resultado esta multiplicación de los ordenes.

3.4 Orden de crecimiento de funciones conocidas.

O(1) < O(log(n)) < O(n) < O(log(n)n) < O(n^k) < O(k^n)

3.5 Reglas de calculo de complejidad de T(n).

El tiempo de ejecución de cada sentencia simple, por lo común puede tomarse como O(1).
El tiempo de ejecución de una secuencia de proposiciones se determina por la regla de la suma. Es el máximo tiempo de ejecución de una proposición de la sentencia.
Para las sentencias de bifurcación (IF,CASE) el orden resultante será el de la bifurcación con mayor orden.
Para los bucles es el orden del cuerpo del bucle sumado tantas veces como se ejecute el bucle.
El orden de una llamada a un subprograma no recursivo es el orden del subprograma.

Capítulo

2

Manejo de Memoria

Java ofrece toda la funcionalidad de un lenguaje potente, pero sin las características menos usadas y más confusas de éstos. C++ es un lenguaje que adolece de falta de seguridad, pero C y C++ son lenguajes más difundidos, por ello Java se diseñó para ser parecido a C++ y así facilitar un rápido y fácil aprendizaje.

Java elimina muchas de las características de otros lenguajes como C++, para mantener reducidas las especificaciones del lenguaje y añadir características muy útiles como el garbage collector (reciclador de memoria dinámica). No es necesario preocuparse de liberar memoria, el reciclador se encarga de ello y como es un thread de baja prioridad, cuando entra en acción, permite liberar bloques de memoria muy grandes, lo que reduce la fragmentación de la memoria.

2.1 Manejo de Memoria Dinámica

El recolector de basura

Un argumento en contra de lenguajes como C++ es que los programadores se encuentran con la carga añadida de tener que administrar la memoria de forma manual. En C++, el desarrollador debe asignar memoria en una zona conocida como ‘’’heap’’ (montículo) para crear cualquier objeto, y posteriormente desalojar el espacio asignado cuando desea borrarlo. Un olvido a la hora de desalojar memoria previamente solicitada, o si no lo hace en el instante oportuno, puede llevar a una ‘’’fuga de memoria’’’, ya que el sistema operativo piensa que esa zona de memoria está siendo usada por una aplicación cuando en realidad no es así. Así, un programa mal diseñado podría consumir una cantidad desproporcionada de memoria. Además, si una misma región de memoria es desalojada dos veces el programa puede volverse inestable y llevar a un eventual ‘’’cuelgue’’’.

En Java, este problema potencial es evitado en gran medida por el recolector automático de basura (o automatic garbage collector). El programador determina cuando se crean los objetos y el entorno en tiempo de ejecución de Java (Java runtime) es el responsable de gestionar el ciclo de vida de los objetos. El programa, u otros objetos pueden tener localizado un objeto mediante una referencia a éste (que, desde un punto de vista de bajo nivel es una dirección de memoria). Cuando no quedan referencias a un objeto, el recolector de basura de Java borra el objeto, liberando así la memoria que ocupaba previniendo posibles fugas (ejemplo: un objeto creado y únicamente usado dentro de un método sólo tiene entidad dentro de éste; al salir del método el objeto es eliminado). Aún así, es posible que se produzcan fugas de memoria si el código almacena referencias a objetos que ya no son necesarios—es decir, pueden aún ocurrir, pero en un nivel conceptual superior. En definitiva, el recolector de basura de Java permite una fácil creación y eliminación de objetos, mayor seguridad y frecuentemente más rápida que en C++.

La recolección de basura de Java es un proceso prácticamente invisible al desarrollador. Es decir, el programador no tiene conciencia de cuándo la recolección de basura tendrá lugar, ya que ésta no tiene necesariamente que guardar relación con las acciones que realiza el código fuente.

Debe tenerse en cuenta que la memoria es sólo uno de los muchos recursos que deben ser gestionados.

Capítulo

3

Estructuras Lineales Estáticas y Dinámicas

3.1 Pilas

Son un tipo de estructura de datos lineales, las cuales están compuestas por un conjunto de elementos. Las inserciones y eliminaciones de los elementos se realizan solo por un extremo, al cual se le denomina ***cima*** o ***tope (top).*** En consecuencia a esta restricción en las pilas los elementos serán eliminados en orden inverso en el que fueron insertados, es decir, el último elemento que se mete es el primero que se saca. Por este concepto las pilas corresponden a las estructuras de tipo **LIFO** (Last In, First Out: último en entrar, primero en salir).

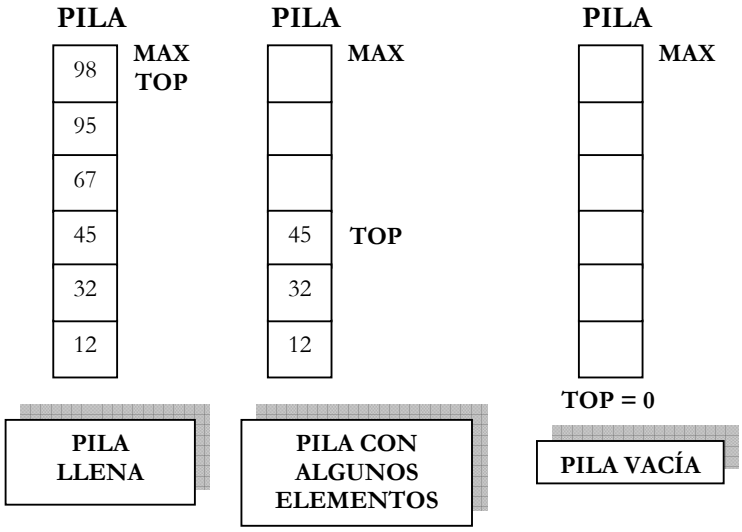
Existen numerosos ejemplos de las pilas en la vida cotidiana.

3.1 REPRESENTACIÓN EN MEMORIA.

Las pilas no son estructuras de datos fundamentales, es decir, no están definidas como tales en los lenguajes de programación (como el caso de los arreglos). Las pilas pueden representarse mediante el uso de:

- Arreglos
- Listas enlazadas

En nuestro caso haremos uso de los arreglos, por los tanto debemos definir el tamaño máximo de la pila, y además una variable auxiliar a la que denominaremos TOP, el cual será un apuntador al último elemento insertado en la pila.



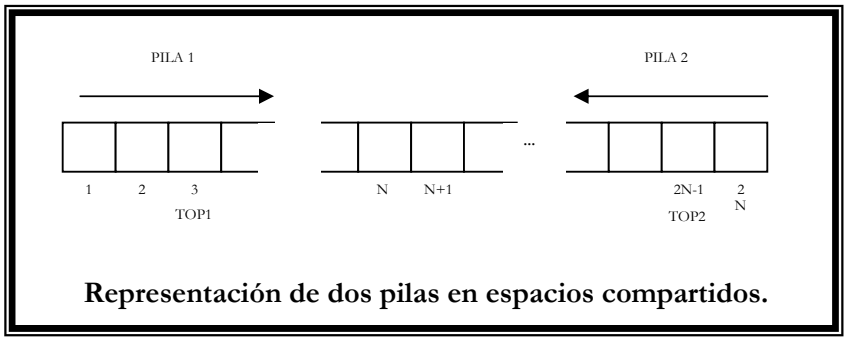
Por el hecho de implementar arreglos para la representación de las pilas, existe una limitante llamada "espacio de memoria reservada". Esto significa que al momento de establecer el máximo de la capacidad de la pila, ya no es posible insertar más elementos. Si la pila estuviera llena y se insertara un nuevo elemento, se arrojaría un error conocido como **desbordamiento** (Overflow).

En la tarea de la solución de este problema se puede definir una pila demasiada extensa, sin embargo, resultaría ineficiente y costoso si solo se utiliza para unos cuantos elementos. No siempre será posible conocer el numero de elementos a utilizar, por lo tanto siempre existe la posibilidad de cometer un error de

desbordamiento (si se reserva menos espacio del que efectivamente se usara) o bien, de hacer uso ineficiente de la memoria (si se reserva más espacio del que se empleará).

Existe otra alternativa de solución a este problema. Consistente en el uso de **espacios compartidos** de memoria para la implementación de pilas. Suponga que necesita dos pilas, cada una con un tamaño máximo de N elementos. En este caso se definirá un solo arreglo de 2*N elementos en lugar de dos arreglos de N elementos.

En este caso implementaremos dos apuntadores: TOP1 para apuntar al último elemento insertado en la pila1 y TOP2 para apuntar al último elemento de la pila2. cada una de las pilas insertará sus elementos por los extremos opuestos, es decir, la pila1 iniciará a partir de la posición 1 del arreglo y la pila2 iniciará en la localidad 2N. De este modo si la pila1 necesita más de N espacios y la pila2 no tiene ocupados sus N espacios, entonces se podrá seguir insertando elementos en la pila1 sin caer en el error de "**Overflow".**



Representación de dos pilas en espacios compartidos.

Operaciones de una pila

La definición de una estructura de datos queda completa al incluir las operaciones que se puedan realizar en ella. En el caso de la pila las operaciones básicas que se pueden llevar a cabo son:

- Poner un Elemento (Push).
- Quita un Elemento (Pop).

De las cuales se determinan de los siguientes codigos:

```
public void Push(int d)
{
        if(T==max)
            System.out.print("PIla llena");
        else
        {
            T++;
            Pila[T]=d;
        }
}
```

```
public void Pop()
      {
            if(T==-1)
              System.out.println("Pila Vacia");
            else
            {
                System.out.print("Dato de Salida" + Pila[T]
                Ti--;
            }
      }
```

3.2 NOTACIONES CON EXPRESIONES.

Las pilas son estructuras de datos implementadas para la solución de diversos tipos de problemas. Pero tal vez la aplicación más importante de estas es el tratamiento de expresiones matemáticas. El convertir expresiones en notación infija en su equivalente en notación postfija (o prefija). A continuación es necesario analizar algunos conceptos para introducirnos a este tema.

- La expresión **A+B** se dice que esta en notación **infija,** y su nombre se debe a que el operador + está entre los operandos A y B.
- Dada la expresión **AB+** se dice que esta en notación **postfija** y su nombre se debe a que el operador + esta después de los operandos A y B.
- Dada la expresión **+AB** se dice que esta en notación **prefija,** y su nombre se debe a que el operador + está antes que los operandos A y B.

La ventaja de usar estas expresiones en notación postfija o prefija radica en que no son necesarios los paréntesis para indicar el orden de la operación, ya que este queda establecido por la ubicación de los operadores con respecto a los operandos.

Para convertir expresiones a las diferentes notaciones ya descritas, es necesario establecer ciertas condiciones:

- Solamente se manejaran los siguientes operadores en orden de prioridad:

  ^ potencia
  * / Multiplicación y división
  + - suma y resta

- Los operandos de más alta prioridad se ejecutan primero
- Si hubiese una expresión de dos o más operadores de igual prioridad, se ejecutan de izquierda a derecha.

- Las subexpresiones parentizadas tendrán más prioridad que cualquier operador.

EJEMPLOS:

A) EXPRESIÓN INFIJA: **X + Z * W**

EXPRESIÓN POSTFIJA: **X Z W * +**

PASO	EXPRESION
0	**X + Z * W**
1	**X + Z W ***
2	**X Z W * +**

B) EXPRESIÓN INFIJA: **(X + Z) * W / T ^Y - V**

EXPRESIÓN POSTFIJA: **X Z + W * T Y ^ / V –**

PASO	EXPRESION
0	**(X + Z) * W / T ^Y – V**
1	**X Z + * W / T ^Y – V**
2	**X Z + * W / T Y ^ – V**
3	**X Z + W * / T Y ^ – V**
4	**X Z + W * T Y ^ / – V**
5	**X Z + W * T Y ^ / V -**

FUNCIONAMIENTO DEL ALGORITMO:

A) EXPRESIÓN INFIJA: X + Z * W

PASO	EXPRESIÓN INFIJA	SIMBOLO ANALIZADO	PILA	EXPRESIÓN POSTFIJA
0	X + Z * W			

1	+ Z * W	X		X
2	Z * W	+	+	X
3	* W	Z	+	XZ
4	W	*	+ *	XZ
5		W	+ *	XZW
6			+	XZW *
7				XZW * +

B) EXPRESIÓN INFIJA: (X + Z) * W / T ^Y - V

PASO	EXPRESIÓN INFIJA	SIMBOLO ANALIZADO	PILA	EXPRESIÓN POSTFIJA
0	(X + Z) * W / T ^Y - V			
1	X + Z) * W / T ^Y - V	(	(	
2	+ Z) * W / T ^Y - V	X	(	X
3	Z) * W / T ^Y - V	+	(+	X
4	) * W / T ^Y - V	Z	(+	XZ
5	* W / T ^Y - V	))	(	X Z + X Z +
6	W / T ^Y – V	*	*	X Z +
7	/ T ^Y – V	W	*	X Z + W
8	T ^Y – V	/ /	 /	X Z + W * X Z + W *
9	^Y – V	T	/	X Z + W * T
10	Y – V	^	/ ^	X Z + W * T
11	- V	Y	/ ^	X Z + W * T Y
12	V	- - -	/ -	X Z + W * T Y ^ X Z + W * T Y ^ / X Z + W * T Y ^ /
13		V	-	X Z + W * T Y ^ / V -
14				X Z + W * T Y ^ / V -

Los pasos que se consideran mas relevantes son: en el paso 5, al analizar el paréntesis derecho se extraen repetidamente todos los elementos de PILA (solo el operador +), agregándolos a EXPRESIÓN POSTFIJA hasta encontrar un paréntesis izquierdo. El paréntesis izquierdo se quita de PILA pero no se incluye en EXPRESIÓN POSTFIJA (recuerde que las expresiones de notación polaca no necesitan paréntesis para indicar prioridades). Cuando se trata de operador división, paso 8, se quita de PILA el operador * y se agrega a EXPRESIÓN POSTFIJA, ya que la multiplicación tiene igual prioridad que la división. Cuando se analiza el operador resta, paso 12, se extraen de PILA y se incorporan a EXPRESIÓN POSTFIJA todos los operadores de mayor o igual prioridad, en este caso son todos los que están en ella (la potencia y la división), agregándolo finalmente en PILA. El siguiente es el pseudocódigo de Postfija:

```
Tope=0
Repetir mientras EI<>fin de línea
  Símbolo=Símbolo mas a la  izq. de EI
  Si símbolos =paréntesis izq.
     Tope=Tope+1
     Pila[Tope]=símbolo
  Sino   Si   símbolo =paréntesis der.
              Repetir mientras Pila[Tope]<>paréntesis izq.
                   EPOS=Pila[Tope]
                   Tope=Tope-1
               Fin repetir
               Tope=Tope-1
          Sino
              Si símbolo = Operando
                agregar símbolo a EPOS
            Sino
               Repetir mientras Tope>0 y la prioridad del operador sea menor o igual
                                que  la prioridad que el operador que esta en la cima de Pila
                        Epos=Epos+Pila[Tope]
                        Tope= Tope-1
               Fin Repetir
               Tope=Tope+1
                Pila[Tope]=símbolo
              Fin Si
           Fin Si
   Fin Si
Fin Repetir
```

Ahora se analizarán los pasos para la conversión de expresiones infijas a notaciones prefijas.

A) EXPRESIÓN INFIJA: X + Z * W

EXPRESIÓN PREFIJA: + X * Z W

PASO	EXPRESIÓN
0	X + Z * W
1	X + * Z W
2	+ X * Z W

B) EXPRESIÓN INFIJA: (X + Z) * W / T ^Y - V

EXPRESIÓN PREFIJA: - / * + X Z W ^ T Y V

PASO	EXPRESIÓN
0	(X + Z) * W / T ^Y - V
1	+ X Z * W / T ^Y – V
2	+ X Z * W / ^ T Y – V
3	* + X Z W / ^ T Y - V
4	/ * + X Z W ^ T Y - V
5	- / * + X Z W ^ T Y V

Lo primero que se procesa es la subexpresión parentizada, paso 1. El orden en que se procesan los operadores es el mismo que se siguió para la conversión de infijas a posfijas. Por lo tanto, seria reiterativo volver a explicar paso por paso el contenido de la tabla 3.6. sin embargo cabe hacer notar la posición que ocupan los operadores con respecto a los operando. Los primeros proceden a los segundos.

A continuación se incluyen el algoritmo de conversión de notación fija a notación polaca prefija. Este algoritmo se diferencia del anterior básicamente en el hecho de que los elementos de la expresión en notación infija se recorrerán de derecha a izquierda.

A) EXPRESIÓN INFIJA: **X + Z * W**

EXPRESIÓN PREFIJA: **+ X * Z W**

PASO	EXPRESIÓN INFIJA	SIMBOLO ANALIZADO	PILA	EXPRESIÓN POSTFIJA
0	X + Z * W			
1	X + Z *	W		W
2	X + Z	*	*	W
3	X +	Z	*	W Z
4	X	+		W Z *
		+	+	W Z *
5		X	+	W Z * X
6				W Z * X +
7	INVERTIR EPRE	+ X * Z W		

Paso 2, el operador de multiplicación se pone en PILA al ser examinado. Al estar vacía PILA, no hay otros operadores que pudieran quitarse (según su prioridad) antes de poner el operador *. En cambio al analizar el operador de suma, paso 4, se compara su prioridad con la del operador del tope de PILA. En este caso es

menor, y por lo tanto se extrae el elemento del tope de PILA y se agrega a la expresión prefija, poniendo finalmente el operador + en PILA. Cuando la expresión de entrada queda vacía, es decir, que ya han analizado todos sus símbolos, se extrae repetidamente cada elemento de la PILA y se agrega a la expresión prefija. Así hasta que quede vacía.

B) EXPRESIÓN INFIJA: (X + Z) * W / T ^Y - V

EXPRESIÓN PREFIJA: - / * + X Z W ^ T Y V

PASO	EXPRESIÓN INFIJA	SIMBOLO ANALIZADO	PILA	EXPRESIÓN POSTFIJA
0	(X + Z) * W / T ^ Y - V			
1	(X + Z) * W / T ^ Y -	V		V
2	(X + Z) * W / T ^ Y	-	-	V
3	(X + Z) * W / T ^	Y	-	V Y
4	(X + Z) * W / T	^	- ^	V Y
5	(X + Z) * W /	T	- ^	V Y T
6	(X + Z) * W	/	-	V Y T ^
		/	- /	V Y T ^
7	(X + Z) *	W	- /	V Y T ^ W
8	(X + Z)	*	- / *	V Y T ^ W
9	(X + Z	)	- / *)	V Y T ^ W
10	(X +	Z	- / *)	V Y T ^ W Z
11	(X	+	- / *) +	V Y T ^ W Z
12	(	X	- / *) +	V Y T ^ W Z X
13		(	- / *)	V Y T ^ W Z X +
		(	- / *	
				V Y T ^ W Z X +
14			- /	
			-	V Y T ^ W Z X + *
				V Y T ^ W Z X + * /
				V Y T ^ W Z X + * / -
15	Invertir EPRE	- / * + X Z W ^ T Y V		

Como la expresión se recorre de derecha a izquierda, el primer operador que se procesa es la resta, paso 2. pero, éste es el operador de la expresión de más baja prioridad, por lo tanto permanecerá en PILA hasta el final del proceso de conversión, paso 14. cuando se encuentra un paréntesis derecho, paso 9, se agrega directamente a PILA. Mientras que cuando el símbolo analizado es un paréntesis izquierdo, paso 13, se extrae repetidamente cada elemento de PILA agregándolo a EPRE, hasta que se encuentra un paréntesis derecho. Este se retira de PILA y no se agrega a EPRE. Analizados todos los símbolos de la expresión se procede a retirar repetidamente de PILA sus elementos, añadiéndolos a EPRE. Finalmente se invierte EPRE para obtener la expresión en notación prefija, paso 15.

```
Tope=0
Repetir mientras EI<>fin de línea
  Símbolo=Símbolo mas a la  der. de EI
  Si símbolos =paréntesis der.
     Tope=Tope+1
     Pila[Tope]=símbolo
  Sino   Si   símbolo =paréntesis izq.
              Repetir mientras Pila[Tope]<>paréntesis der.
                   EPRE= EPRE +Pila[Tope]
                   Tope=Tope-1
               Fin repetir
               Tope=Tope-1
           Sino
               Si símbolo = Operando
                  agregar símbolo a EPRE
              Sino
                  Repetir mientras Tope>0 y la prioridad del operador sea menor que
                            la prioridad del operador que esta en la cima de Pila
                            EPRE = EPRE +Pila[Tope]
                            Tope= Tope-1
                  Fin Repetir
                  Tope=Tope+1
                   Pila[Tope]=símbolo
                 Fin Si
            Fin Si
   Fin Si
Fin Repetir
Repetir mientras Tope>0
       EPRE=EPRE+Pila[Tope]
       Tope=Tope-1
Fin Repetir
```

3.2 Colas

Es una lista de elementos en la que los elementos se introducen por un extremo y se eliminan por otro. Los elementos se eliminan en el mismo orden en el que se insertaron. Por lo tanto el primer elemento en entrar a la cola será el primer elemento en salir. Debido a esta característica las colas reciben también el nombre de QUEUE o FIFO (First-In, First-Out: Primero en Entrar Primero en Salir).
En la vida real existen numerosos casos de colas: personas esperando para usar un teléfono público, las personas que esperan para ser atendidas en la caja de un banco, etc.

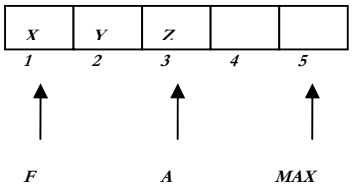
Otras aplicaciones ligadas a la computación son: en las colas de Impresión, en el caso de que existiese una sola computadora para atender a varios usuarios, puede suceder que algunos de ellos soliciten los servicios de impresión al mismo tiempo o mientras el dispositivo está ocupado. En estos casos se forma una cola con los trabajos que esperan a ser impresos. Los mismos se irán imprimiendo en el orden en que fueron llegando a la cola.

Otro caso es cuando se manejan los sistemas en tiempo compartido. Varios usuarios comparten ciertos recursos, como CPU o memoria RAM. Los recursos se asignan a los procesos que están en cola de espera, suponiendo que todos tienen la misma prioridad, en el orden en el cual fueron introducidos en la cola.

2.1. Representación En Memoria Estática.

Al igual que las estructuras de pilas, las colas no existen como estructuras de datos estándares en los lenguajes de programación.
En este caso se tratarán las colas como arreglos de elementos, en las cuales es necesario definir el tamaño de la cola y dos apuntadores. El primer apuntador se usará para acceder al primer elemento al cual le llamaremos F y el otro apuntador

que guarde el último elemento y le llamaremos A. Además uno llamado MÁXIMO para definir el número máximo de elementos en la cola.
Ejemplo:
En la siguiente figura se muestra una cola que ha guardado tres elementos: X,Y,Z.



2.2. COLA LINEAL.

Es un tipo de almacenamiento creado por el usuario que trabaja bajo la técnica **FIFO.** las operaciones que podemos realizar son: iniciación, inserción y extracción.

Se deben considerar algunas condiciones en el manejo de este tipo de colas:

1. OverFlow (Cola Llena)→ al realizar una inserción
2. UnderFlow (Cola Vacía)→ al requerir extraer un elemento
3. Vacío

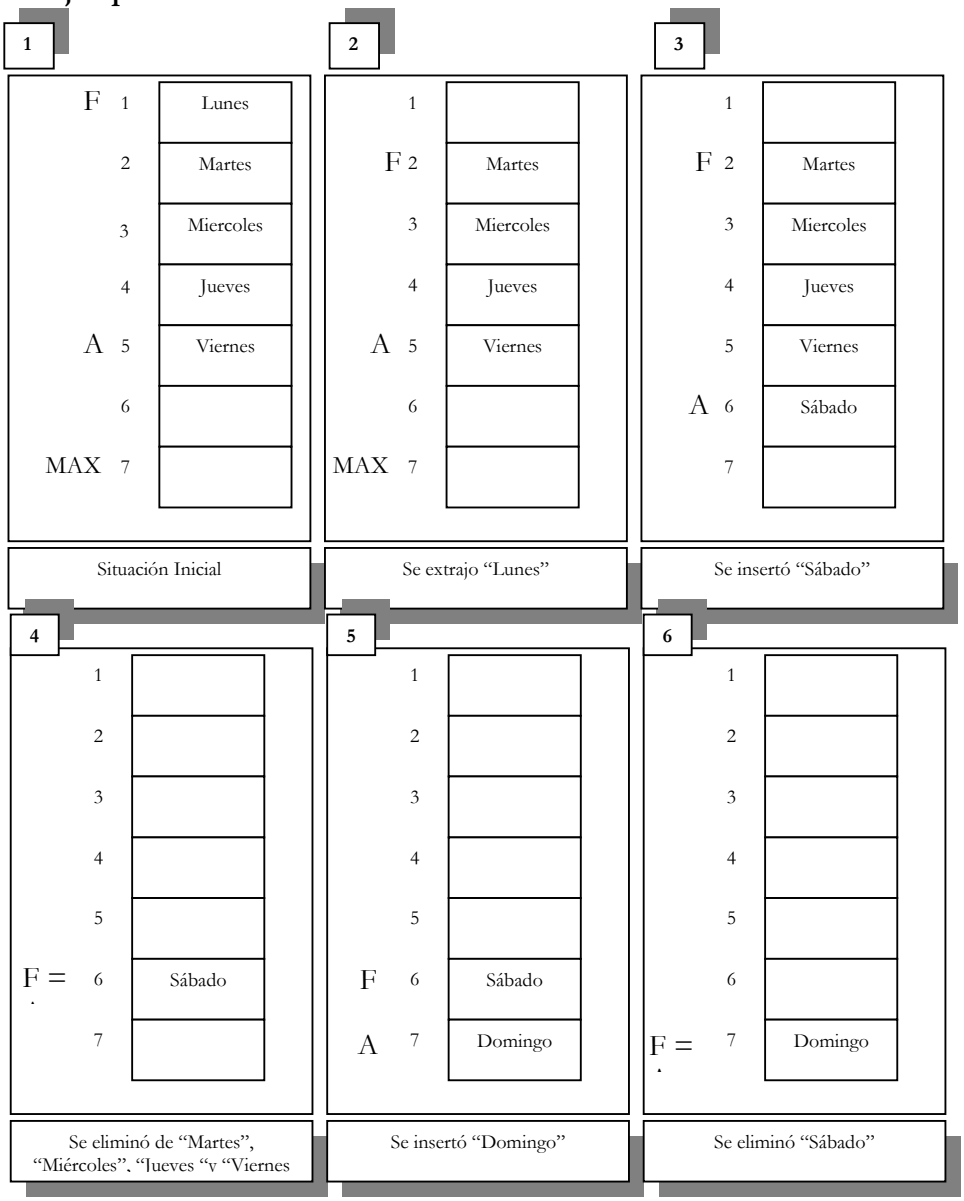
```
class Cola{
        static int c[],tf,ti,max;
        Cola(){
                c=new int[6];
                max=6;
        }
        Cola(int max){
                c=new int [max];
                this.max=max;
                tf=ti=-1;
        }
        public static void insert(int d){
                if(tf==max-1)
                System.out.print("Cola llena");
                else
                {
                        if(ti==-1)
                        ti=tf=0;
                        else tf++; //el else termina aqui.
                        c[tf]=d;
                }
        }
        public static void eliminar(){
                if(ti==-1)
                System.out.println("No se puede eliminar");
                else {
                        if(tf==ti)
                        ti=tf=-1;
                        else ti++;
                }
        }
        public static int mostrar(){
                return c[ti];
        }
        public static boolean estaVacia(){
                if(tf==-1)
                return true;
                return false;
        }
}
```

2.3. COLA CIRCULAR

Las colas lineales muestran una gran debilidad: las extracciones solo se pueden realizar por un extremo, puede llegar el momento en que el apuntador **A** sea igual

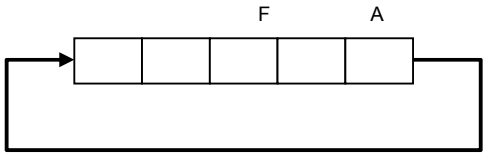
al máximo de elementos en la cola, siendo que al frente de la misma existan lugares vacíos, y al insertar un nuevo elemento nos mandará un error de OverFlow (Cola Llena).

Ejemplo:

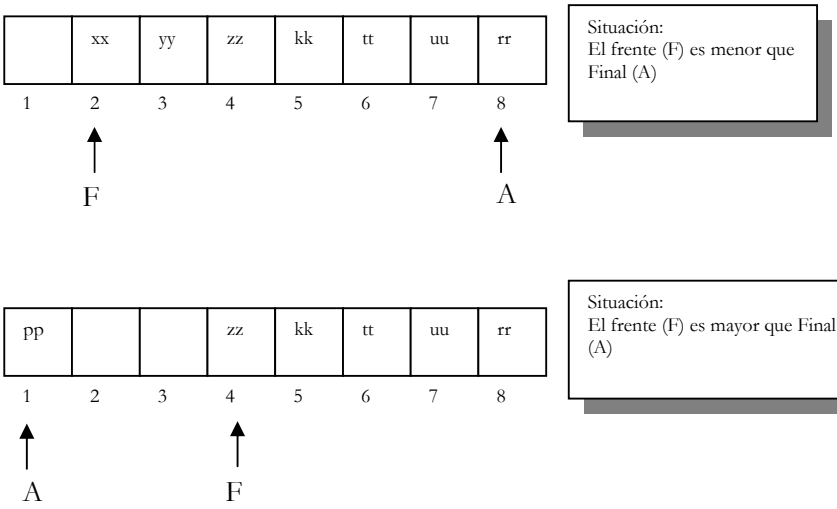


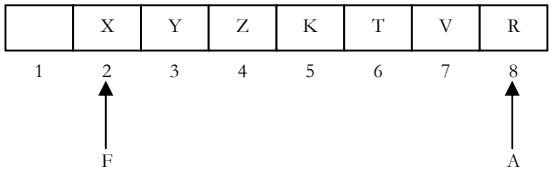
En esta situación del caso 6, si se quisiera insertar un nuevo elemento ya no se podría, puesto que **A** no es menor que **MAX** (A = MAX = 7).
Así mismo se observa que existe espacio disponible, sin embargo no se pueden insertar elementos nuevos.

Para hacer un uso más eficiente de la memoria disponible se trata a la cola como una estructura circular, es decir, el elemento anterior al primero es el último, en otras palabras existe un apuntador desde el último elemento al primero de la cola. Vea la siguiente figura:

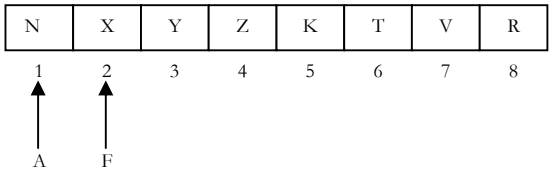


Ejemplos:

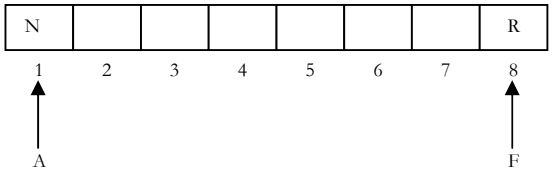




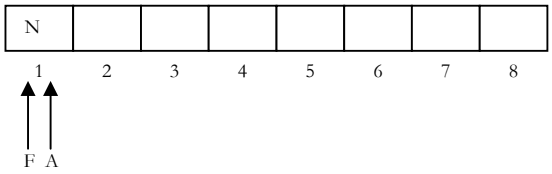
Estado Inicial



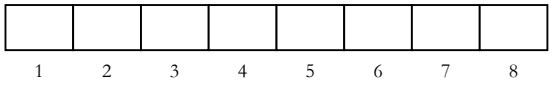
Al agregar "N", el apuntador A se dirige a la posición 1. si se quisiera insertar otro elemento seria imposible debido a que A+1=F "OverFlow"



Se eliminaron "x","y","Z","K","T","V". El apuntador F se dirige a Max



Al eliminar "R" y al ser F=Max, se le da el valor de 1.



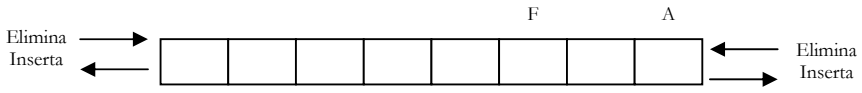
F = A = 0

Al eliminar a "N" y como F = A actualizamos los punteros F y A a 0 y la cola queda vacía.

```
class ColaC extends Cola {

        void ins(int d) {
                if(tf==max-1 && ti==0 || tf==ti-1) {
                        System.out.println ("La cola esta llena");
                }       else

                        if (tf==max-1) {
                                tf=0;
                                c[tf]=d;
                                }

                                else
                                super.insert(d);
                        }

        void elim() {
                if (ti==max-1 && ti!=tf) {
                        ti=0;
                }       else
                        super.elimina();
                }

        }
```

24. DOBLE COLA

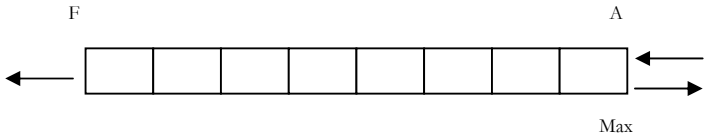
La **doble cola** o **Bicola** es una generalización de una cola simple. En una doble cola los elementos pueden ser insertados o eliminados por cualquiera de los extremos de la bicola, es decir, por el FRENTE o por el FINAL de la cola. Este tipo de cola se representa de la siguiente manera:



Existen dos variantes de las dobles colas:

- Doble cola con entrada restringida
- Doble cola con salida restringida

La variante Doble cola con entrada restringida permite que las eliminaciones puedan hacerse por cualquiera de los dos extremos, mientras que las inserciones solo por el final de la cola



ALGORITMOS DE ENTRADA RESTRINGIDA

a) INICIALIZACION

F ← 1

A ← 0

```
class ColaD extends Cola {

        public void ins(int d) throws Exception{
                Teclado t=new Teclado();
        if (ti>=0 && tf==max) {
                System.out.println ("La cola esta llena");
                }
                else
                System.out.println ("Por donde desea insertar: ");
                System.out.println ("1.- Derecha \n2.- Izquierda");
                int op=t.leeEntero();
                if (op==1) {
                        ti=tf=0;
                        c[tf]=d;
                        super.insert(d);
                }
                if (op==2) {
                        tf=tf+1;
                        c[tf]=d;
                        super.insert(d);
                }
                }

                public void elim() {
                        if (ti<0 && tf==-1) {
                                System.out.println ("No hay datos que
eliminar");
                                }
                                else
                                super.eliminar();
                }


        }
```

En esta estructura a cada uno de los elementos se le son asignados una prioridad, basándose en las siguientes reglas:

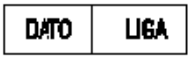
1. Un elemento de mayor prioridad procesado al principio.
2. Dos elementos con la misma prioridad son procesados de acuerdo al orden en el que fueron insertados en la cola.

3.3 Listas Enlazadas

Una lista enlazada o encadenada es una colección de elementos ó nodos, en donde cada uno contiene datos y un enlace o liga.

4.1 REPRESENTACIÓN EN MEMORIA

Un **nodo** es una secuencia de caracteres en memoria dividida en campos (de cualquier tipo). Un nodo siempre contiene la dirección de memoria del siguiente nodo de información si este existe.
Un **apuntador** es la dirección de memoria de un nodo
La figura siguiente muestra la estructura de un nodo:



El campo liga, que es de tipo puntero, es el que se usa para establecer la liga con el siguiente nodo de la lista. Si el nodo fuera el último, este campo recibe como valor NIL (vacío).
Donde el código en java es el siguiente para el nodo:

```
public class Nodo{
        private int dato;
        private Nodo liga;

        public void setDato(int x){
                this.dato=x;
        }
        public int getDato(){
                return dato;
        }
        public Nodo getLiga(){
                return liga;
        }
        public void setLiga(Nodo liga){
                this.liga=liga;
        }
```

A continuación se muestra el esquema de una lista :

4.2 OPERACIONES EN LISTAS ENLAZADAS

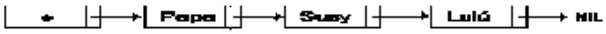
Las operaciones que podemos realizar sobre una lista enlazada son las siguientes:

- **Recorrido.** Esta operación consiste en visitar cada uno de los nodos que forman la lista . Para recorrer todos los nodos de la lista, se comienza con el primero, se toma el valor del campo liga para avanzar al segundo nodo, el campo liga de este nodo nos dará la dirección del tercer nodo, y así sucesivamente.
- **Inserción.** Esta operación consiste en agregar un nuevo nodo a la lista. Para esta operación se pueden considerar tres casos:
    - Insertar un nodo al inicio.
    - Insertar un nodo antes o después de cierto nodo.
    - Insertar un nodo al final.
- **Borrado.** La operación de borrado consiste en quitar un nodo de la lista, redefiniendo las ligas que correspondan. Se pueden presentar cuatro casos:
    - Eliminar el primer nodo.
    - Eliminar el último nodo.
    - Eliminar un nodo con cierta información.
    - Eliminar el nodo anterior o posterior al nodo cierta con información.
- **Búsqueda.** Esta operación consiste en visitar cada uno de los nodos, tomando al campo liga como puntero al siguiente nodo a visitar.

4.3 LISTAS LINEALES

En esta sección se mostrarán algunos algoritmos sobre listas lineales sin nodo de cabecera y con nodo de cabecera.

Una lista con nodo de cabecera es aquella en la que el primer nodo de la lista contendrá en su campo dato algún valor que lo diferencie de los demás nodos (como: *, -, +, etc.). Un ejemplo de lista con nodo de cabecera es el siguiente:



En el caso de utilizar listas con nodo de cabecera, usaremos el apuntador CAB para hacer referencia a la cabeza de la lista. Para el caso de las listas sin nodo de cabecera, se usará la expresión TOP para referenciar al primer nodo de la lista, y TOP(dato), TOP(liga) para hacer referencia al dato almacenado y a la liga al siguiente nodo respectivamente.

```
class Listas{
	Nodo p;
	Teclado t=new Teclado();
	public void insert(int d) throws Exception{
		int opcion;
		System.out.print("\nIngrese el metodo de insercion:\n\n1. rincipio\n2.
			Final\n3.	Antes\n4.  Despues\nOPCION: ");
		opcion=t.leeEntero();
		switch(opcion){
			case 1: principio(d); break;
			case 2: fin(d); break;
			case 3: antes(d); break;
			case 4: despues(d); break;
		}
	}
	private void principio(int d){
		Nodo n=new Nodo();
		n.setDato(d);
		n.setLiga(p);
		p=n;
	}
private void fin(int d){
		if(p==null){
			principio(d);
		}
		else{
				Nodo q=p;
			Nodo n=new Nodo();
			while(q.getLiga()!=null){
				q=q.getLiga();
				}
				q.setLiga(n);
				n.setDato(d);
		}
	}
	private void antes(int d) throws Exception{
		int b;
		if(p==null)
			System.out.println("\nLista Vacia");
		else{
			Nodo r=null,q,n;
			q=p;
			System.out.print("\nIntrodusca el dato de referencia para insertar
antes: ");
```

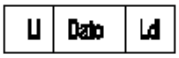
```
                                        q=q.getLiga();
                                }
                                if(q!=null)
                                        if(q==p)
                                                principio(d);
                                        else{
                                                n=new Nodo();
                                                r.setLiga(n);
                                                n.setLiga(q);
                                                n.setDato(d);
                                        }
                                else{
                                        System.out.println("Dato de Referencia no encontrado");
                                }
                        }
                }
                private void despues(int d) throws Exception{
                        int b;
                        if(p==null)
                                System.out.println("\nLista vacia");
                        else{
                                Nodo r,q,n;
                                q=p;
                                System.out.print("\nIntrodusca el dato de referencia para insertar
despues: ");
                                b=t.leeEntero();
                                while(q!=null&&q.getDato()!=b){
                                        q=q.getLiga();
                                }
                                if(q!=null){
                                        n=new Nodo();
                                        if(q.getLiga()!=null){
                                                r=q.getLiga();
                                                n.setLiga(r);
                                        }
                                        n.setDato(d);
                                        q.setLiga(n);
                                }
                                else{
                                        System.out.println("\nDato de referencia no encontrado");
                                }
                        }
                }
                public void visual(){
                        Nodo m;
                        for(m=p;m!=null;m=m.getLiga())
                          System.out.print(m.getDato()+" ");
                        System.out.println();
                }
                public void elim() throws Exception{
                        int opcion;
                        System.out.print("\nIngrese el metodo de Eliminacion:\n\n1.          Principio\n2.
 Final\n3.   Antes\n4.  Despues\n5.          ElimX\nOPCION: ");
                        opcion=t.leeEntero();
                        switch(opcion){
                                case 1: elimPrincipio(); break;
                                case 2: elimFin(); break;
                                case 3: elimAntes(); break;
                                case 4: elimDespues(); break;
                                case 5: elimX(); break;
                        }
```

```
	}
	public void elimPrincipio(){
		if(p!=null)
			p=p.getLiga();
			System.out.println("No hay datos en la lista");
	}
	public void elimFin(){
		Nodo q=p,r=null;
		while(q.getLiga()!=null){
				r=q;
				q=q.getLiga();
		}
		if(p==q)
		p=null;
		else{
			r.setLiga(null);
		}
	}
	public void elimAntes() throws Exception{
		//Nodo q=p,r=null;
		int b;
		if(p==null)
			System.out.println("\nLista Vacia");
		else{
			Nodo r=null,q=p;
			System.out.print("\nIntrodusca el dato de referencia para eliminar antes:
");
			b=t.leeEntero();
			while(q!=null&&q.getDato()!=b){
				r=q;
				q=q.getLiga();
			}
			if(q!=null)
				if(q==p)
					elimPrincipio();
				else{
					q=q.getLiga();
					r.setLiga(q);
				}
			else{
				System.out.println("Dato de Referencia no encontrado");
			}
		}

	}	public void elimDespues()throws Exception{
		int b;
		if(p==null)
			System.out.println("\nPila vacia");
		else{
			Nodo r=null,q=p;
			System.out.print("\nIntrodusca el dato de referencia para eliminar
despues: ");
			b=t.leeEntero();
			while(q.getLiga()!=null&&q.getDato()!=b){
				q=q.getLiga();
			}
			if(q!=null){
				if(q.getLiga()!=null){
					r=q.getLiga();
					q.setLiga(r.getLiga());
				}
				else{
					System.out.println("ERROR: No se puede
eliminar por que ha llegado al final");
```

```
                                        }
                                }
                                else{
                                        System.out.println("Dato no encontrado");
                                }
                        }

        }
        public void elimX()throws Exception{
                int b;
                if(p==null)
                        System.out.println("\nPila vacia");
                else{
                        Nodo r=null,q=p;
                        System.out.print("\nIntrodusca el dato a eliminar: ");
                        b=t.leeEntero();
                        q=p;
                        while(q.getLiga()!=null&&q.getDato()!=b){
                                r=q;
                                q=q.getLiga();
                        }
                        if(q.getLiga()==null)
                                System.out.println("ERROR: No se puede eliminar porque ha
llegado al final");
                        else{
                                r.setLiga(q.getLiga());
                        }
                }

        }
}
```

4.4 LISTAS DOBLES

Una lista doble , ó doblemente ligada es una colección de nodos en la cual cada nodo tiene dos punteros, uno de ellos apuntando a su predecesor (li) y otro a su sucesor(ld). Por medio de estos punteros se podrá avanzar o retroceder a través de la lista, según se tomen las direcciones de uno u otro puntero.
La estructura de un nodo en una lista doble es la siguiente:



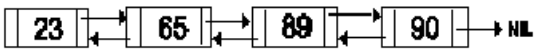
Existen dos tipos de listas doblemente ligadas:

- **Listas dobles lineales.** En este tipo de lista doble, tanto el puntero izquierdo del primer nodo como el derecho del último nodo apuntan a NIL.
- **Listas dobles circulares.** En este tipo de lista doble, el puntero izquierdo del primer nodo apunta al último nodo de la lista, y el puntero derecho del último nodo apunta al primer nodo de la lista.

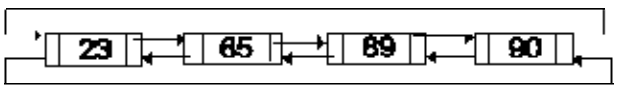
Donde el codigo del nuevo nodo doble en java es:

```
class NodoDoble extends Nodo{
        NodoDoble ll;
        public void setll(NodoDoble l){
                ll=l;
        }
        public NodoDoble getll(){
                return ll;
        }
}
```

Debido a que las listas dobles circulares son más eficientes, los algoritmos que en esta sección se traten serán sobre listas dobles circulares.
En la figura siguiente se muestra un ejemplo de una lista doblemente ligada lineal que almacena números:



En la figura siguiente se muestra un ejemplo de una lista doblemente ligada circular que almacena números:



A continuación mostraremos algunos algoritmos sobre listas enlazadas. Como ya se mencionó, llamaremos li al puntero izquierdo y ld al puntero derecho, también usaremos el apuntador top para hacer referencia al primer nodo en la lista, y p para referenciar al nodo presente.

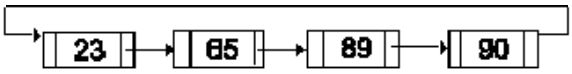
```
class ListasDobles{
	Teclado t=new Teclado();
	NodoDoble p;

	//Metodos de insercion.
	public void insert(int d) throws Exception{
		int opcion;
		System.out.print("\nIngrese el metodo de insercion:\n\n1.		Principio\n2.
	 Final\n3.	Antes\n4.	Despues\nOPCION: ");
		opcion=t.leeEntero();
		switch(opcion){
			case 1: insPrincListDob(d); break;
			case 2: insFinListDob(d); break;
			case 3: insAntesListDob(d); break;
			case 4: insDespListDob(d); break;
		}
	}
	private void insPrincListDob(int d){
		NodoDoble n=new NodoDoble();
		n.setDato(d);
		n.setLiga(p);
		if(p!=null)
			p.setll(n);
		p=n;
	}
	private void insAntesListDob(int d)throws Exception{
		//*****ERROR*****//
		int b;
		NodoDoble q,n;
		if(p!=null){
			System.out.print("Ingrese el valor de referencia antes de: ");
			b=t.leeEntero();
			q=p;
			while(q.getLiga()!=null&&q.getDato()!=b){
				q=(NodoDoble)q.getLiga();
			}
			if(q!=p){
				if(q!=null){
					n=new NodoDoble();
					n.setLiga(q);
					n.setll(q.getll());
					q.setLiga(n);
					n.getll().setLiga(n);
				}
				else
					insPrincListDob(d);
			}
			else
				System.out.println("No se encontro el dato de referencia");
		}
		else
			System.out.println("No existen datos");
	}
```

```
	private void insFinListDob(int d){
		NodoDoble q;
		q=p;
		if(p!=null){
			while(q.getLiga()!=null){
				q=(NodoDoble)q.getLiga();
			}
		NodoDoble n=new NodoDoble();
		q.setLiga(n);
		n.setll(q);
		n.setDato(d);
		}
		else
			insPrincListDob(d);
	}
	private void insDespListDob(int d)throws Exception{
		//*****ERROR*****//
		int b;
		NodoDoble q,n;
		if(p!=null){
			System.out.println("Ingrese el dato de referencia despues de: ");
			b=t.leeEntero();
			q=p;
			while(q!=null&&q.getDato()!=b){
				q=(NodoDoble)q.getLiga();
			}
			if(q!=p){
				if(q!=null){
					n=new NodoDoble();
					n.setLiga(q.getLiga());
					n.setll(q);
					n.setDato(d);
				}
				else
					insFinListDob(d);
			}
			else
				System.out.println("No se encontro el dato de referencia");
		}
		else
			System.out.println("No hay Datos");
	}


//Metodos de eliminacion.
	public void elim(int d) throws Exception{
			int opcion;
		System.out.print("\nIngrese el metodo de eliminacion:\n\n1.		Principio\n2.
	 Final\n3.	Antes\n4.	Despues\n5.		Dato\nOPCION: ");
		opcion=t.leeEntero();
		switch(opcion){
			case 1: elimPrincListDob(); break;
			case 2: elimFinListDob(); break;
			case 3: elimAntesListDob(); break;
			case 4: elimDespListDob(); break;
			case 5: elimXListDob(d);
		}
	}
	private void elimPrincListDob(){
		if(p==null)
			System.out.println("No hay datos en la lista");
			else{
				p=(NodoDoble)p.getLiga();
				if(p!=null)
					p.setll(null);
			}
	}
	private void elimFinListDob(){
		NodoDoble q;
		if(p==null)
			System.out.println("No hay datos");
		else{
```

```
                        q=p;
                        while(q.getLiga()!=null)
                                q=(NodoDoble)q.getLiga();
                        if(q==p)
                                p=null;
                        else
                                q.getll().setLiga(null);
                }
        }
        private void elimAntesListDob()throws Exception{
                int b;
                NodoDoble q;
                if(p==null)
                        System.out.println("No hay datos");
                else{
                        System.out.print("Introdusca el dato de referencia antes de: ");
                        b=Teclado.leeEntero();
                        q=p;
                        while(q.getLiga()!=null&&q.getDato()!=b)
                                q=(NodoDoble)q.getLiga();
                        if(q!=null)
                                elimXListDob(q.getll().getDato());
                        else
                                System.out.println("No se encontro el dato de referencia");
                }
        }
        private void elimDespListDob()throws Exception{
                int b;
                NodoDoble q;
                if(p==null)
                        System.out.println("No hay datos");
                else{
                        System.out.print("Introdusca el dato de referencia despues de: ");
                        b=Teclado.leeEntero();
                        q=p;
                        while(q.getLiga()!=null&&q.getDato()!=b)
                                q=(NodoDoble)q.getLiga();
                        if(q==p)
                                elimPrincListDob();
                        else{
                                if(q.getLiga()!=null)
                                        elimXListDob(q.getLiga().getDato());
                                else
                                        System.out.println("No se encontro el dato de
referencia");
                        }
                }
        }

private void elimXListDob(int b){
                NodoDoble q;
                if(p==null)
                        System.out.println("No hay datos");
                else{
                        q=p;
                        while(q!=null&&q.getDato()!=b)
                                q=(NodoDoble)q.getLiga();
                if(q.getDato()==b)
                        if(q==p)
                                elimPrincListDob();
                        else
                                if(q.getLiga()==null)
                                        elimFinListDob();
                                else{
                                        q.getll().setLiga(q.getLiga());
                                        ((NodoDoble)q.getLiga()).setll(q.getll());
                                }
                else
                        System.out.println("No se encontro el dato de referencia");
                }
        }

        //Visualizacion de listas.
        public void visual(){
                NodoDoble m;
                for(m=p;m!=null;m=(NodoDoble)m.getLiga())
                  System.out.print(m.getDato()+" ");
                System.out.println();
        }
```

4.5 LISTAS CIRCULARES

Las listas circulares tienen la característica de que el último elemento de la misma apunta al primero
La siguiente figura es una representación gráfica de una lista circular.



Capítulo

4

Recursividad

La recursividad es una técnica de programación importante. Se utiliza para realizar una llamada a una función desde la misma función. Como ejemplo útil se puede presentar el cálculo de números factoriales. Él factorial de 0 es, por definición, 1. Los factoriales de números mayores se calculan mediante la multiplicación de 1 * 2 * ..., incrementando el número de 1 en 1 hasta llegar al número para el que se está calculando el factorial.

El siguiente párrafo muestra una función, expresada con palabras, que calcula un factorial.

"Si el número es menor que cero, se rechaza. Si no es un entero, se redondea al siguiente entero. Si el número es cero, su factorial es uno. Si el número es mayor que cero, se multiplica por él factorial del número menor inmediato."

Para calcular el factorial de cualquier número mayor que cero hay que calcular como mínimo el factorial de otro número. La función que se utiliza es la función en la que se encuentra en estos momentos, esta función debe llamarse a sí misma para el número menor inmediato, para poder ejecutarse en el número actual. Esto es un ejemplo de recursividad.

La recursividad y la iteración (ejecución en bucle) están muy relacionadas, cualquier acción que pueda realizarse con la recursividad puede realizarse con iteración y viceversa. Normalmente, un cálculo determinado se prestará a una técnica u otra, sólo necesita elegir el enfoque más natural o con el que se sienta más cómodo.

Claramente, esta técnica puede constituir un modo de meterse en problemas. Es fácil crear una función recursiva que no llegue a devolver nunca un resultado definitivo y no pueda llegar a un punto de finalización. Este tipo de recursividad hace que el sistema ejecute lo que se conoce como bucle "infinito".

Para entender mejor lo que en realidad es el concepto de recursión veamos un poco lo referente a la **secuencia de Fibonacci.**

Principalmente habría que aclarar que es un ejemplo menos familiar que el del factorial, que consiste en la secuencia de enteros.

0,1,1,2,3,5,8,13,21,34,...,

Cada elemento en esta secuencia es la suma de los precedentes (por ejemplo 0 + 1 = 0, 1 + 1 = 2, 1 + 2 = 3, 2 + 3 = 5, ...) sean *fib*(0) = 0, *fib* (1) = 1 y así sucesivamente,

entonces puede definirse la secuencia de Fibonacci mediante la **definición recursiva** (define un objeto en términos de un caso mas simple de si mismo):

Fib (n) = n if n = = 0 or n = = 1

Fib (n) = fib (n - 2) + fib (n - 1) if n >= 2

Por ejemplo, para calcular *fib* (6), puede aplicarse la definición de manera recursiva para obtener:

Fib (6) = fib (4) + fib (5) = fib (2) + fib (3) + fib (5) = fib (0) + fib (1) + fib (3) + fib (5) = 0 + 1 fib (3) + fib (5)

1. + fib (1) + fib (2) + fib(5) =

1. + 1 + fib(0) + fib (1) + fib (5) =
2. + 0 + 1 + fib(5) = 3 + fib (3) + fib (4) =
3. + fib (1) + fib (2) + fib (4) =

   3 + 1 + fib (0) + fib (1) + fib (4) =

4. + 0 + 1 + fib (2) + fib (3) = 5 + fib (0) + fib (1) + fib (3) =
5. + 0 + 1 + fib (1) + fib (2) = 6 + 1 + fib (0) + fib (1) =
6. + 0 + 1 = 8

Obsérvese que la definición recursiva de los números de Fibonacci difiere de las definiciones recursivas de la función factorial y de la multiplicación . La definición recursiva de *fib* se refiere dos veces a sí misma . Por ejemplo, *fib* (6) = *fib* (4) + *fib* (5), de tal manera que al calcular *fib* (6), *fib* tiene que aplicarse de manera recursiva dos veces. Sin embargo calcular *fib* (5) también implica calcular *fib* (4), así que al aplicar la definición hay mucha redundancia de cálculo. En ejemplo anterior, *fib*(3) se calcula tres veces por separado. Sería mucho mas eficiente "recordar" el valor de *fib*(3) la primera vez que se calcula y volver a usarlo cada vez que se necesite. Es mucho mas eficiente un método iterativo como el que sigue parar calcular *fib* (n).

```
class Fibonacci{
        public static int fib(int x){
                if (x==1)
                        return 1;
                else
                        return fib(x-1)+fib(x-2);
        }
}
```

Compárese el numero de adiciones (sin incluir los incrementos de la variable índice, i) que se ejecutan para calcular *fib* (6) mediante este algoritmo al usar la definición recursiva. En el caso de la función factorial, tienen que ejecutarse el mismo numero de multiplicaciones para calcular *n!* Mediante ambos métodos: recursivo e iterativo.

Lo mismo ocurre con el numero de sumas en los dos métodos al calcular la multiplicación. Sin embargo, en el caso de los números de Fibonacci, el método recursivo es mucho mas costoso que el iterativo.

4.2. Propiedades de las definiciones o algoritmos recursivos:

Un requisito importante para que sea correcto un algoritmo recursivo es que no genere una secuencia infinita de llamadas así mismo. Claro que cualquier algoritmo que genere tal secuencia no termina nunca. Una función recursiva *f* debe definirse en términos que no impliquen a *f* al menos en un argumento o grupo de argumentos. Debe existir una "salida" de la secuencia de llamadas recursivas.

Si en esta salida no puede calcularse ninguna función recursiva. Cualquier caso de definición recursiva o invocación de un algoritmo recursivo tiene que reducirse a la larga a alguna manipulación de uno o casos mas simples no recursivos.

4.3. Cadenas recursivas

Una función recursiva no necesita llamarse a sí misma de manera directa. En su lugar, puede hacerlo de manera indirecta como en el siguiente ejemplo:

```
A (formal parameters) b (formal parameters)

{ {

. .

B (argumenta); a (argumenta);

. .

} /*fin de a*/ } /*fin de b*/
```

En este ejemplo la función *a* llama a *b,* la cual puede a su vez llamar a *a*, que puede llamar de nuevo a *b.* Así, ambas funciones *a* y *b*, son recursivas, dado que se llamas a sí mismo de manera indirecta. Sin embargo, el que lo sea no es obvio a partir del examen del cuerpo de una de las rutinas en forma individual. La rutina *a*, parece llamar a otra rutina *b* y es imposible determinar que se puede llamar así misma de manera indirecta al examinar sólo a *a*.

Pueden incluirse mas de dos rutinas en una cadena recursiva. Así, una rutina *a* puede llamar a *b*, que llama a *c*, ..., que llama a *z*, que llama a *a*. Cada rutina de la cadena puede potencialmente llamarse a sí misma y, por lo tanto es recursiva. Por supuesto, el programador debe asegurarse de que un programa de este tipo no genere una secuencia infinita de llamadas recursivas.

4.4. Definición recursiva de expresiones algebraicas

Como ejemplo de cadena recursiva consideremos el siguiente grupo de definiciones:

a. una **expresión** es un término seguido por un signo mas seguido por un término, o un término solo

b. un **término** es un factor seguido por un asterisco seguido por un factor, o un factor solo.
c. Un **factor** es una letra o una expresión encerrada entre paréntesis.

Antes de ver algunos ejemplos, obsérvese que ninguno de los tres elementos anteriores está definido en forma directa en sus propios términos. Sin embargo, cada uno de ellos se define de manera indirecta. Una expresión se define por medio de un término, un término por medio de un factor y un factor por medio de una expresión. De manera similar, se define un factor por medio de una expresión, que se define por medio de un término que a su vez se define por medio de un factor. Así, el conjunto completo de definiciones forma una cadena recursiva.

La forma mas simple de un factor es una letra. Así A, B, C, Q, Z y M son factores. También son términos, dado que un término puede ser un factor solo. También son expresiones dado que una expresión puede ser un término solo. Como A es una expresión, (A) es un factor y, por lo tanto, un término y una expresión. A + B es un ejemplo de una expresión que no es ni un término ni un factor, sin embargo (A + B) es las tres cosas. A * B es un término y, en consecuencia, una expresión, pero no es un factor. A * B + C es una expresión, pero no es un factor.

Cada uno de los ejemplos anteriores es una expresión valida. Esto puede mostrarse al aplicar la definición de una expresión de cada uno. Considérese, sin embargo la cadena A + * B. No es ni una expresión, ni un término, ni un factor. Sería instructivo para el lector intentar aplicar la definición de expresión, término y factor para ver que ninguna de ellas describe a la cadena A + * B. De manera similar, (A + B*) C y A + B + C son expresiones nulas de acuerdo con las definiciones precedentes.

La función *factor* reconoce factores y debería ser ahora bastante sencilla. Usa el programa común de biblioteca *isalpha* (esta función se encuentra en la biblioteca *ctype.h*), que regresa al destino de cero si su carácter de parámetro es una letra y cero (o FALSO) en caso contrario.

```
class Factorial{
        public static int factor(int m){
                if(m==1)
                        return 1;
                else
                        return m*factor(m-1);
        }
}
```

Las tres rutinas son recursivas, dado que cada una puede llamar a sí misma da manera indirecta. Pos ejemplo, si se sigue la acción del programa para la cadena de entrada " (a * b + c * d) + (e * (f) + g) " se encontrará que cada una de las tres rutinas *expr*, *term* y *factor* se llama a sí misma.

4.5. Programación Recursiva

Es mucho mas difícil desarrollar una solución recursiva en C para resolver un problema especifico cuando no se tiene un algoritmo. No es solo el programa sino las definiciones originales y los algoritmos los que deben desarrollarse. En general, cuando encaramos la tarea de escribir un programa para resolver un problema no hay razón para buscar una solución recursiva. La mayoría de los problemas pueden resolverse de una manera directa usando métodos no recursivos. Sin embargo, otros pueden resolverse de una manera mas lógica y elegante mediante la recursión.

Volviendo a examinar la función factorial. El factor es, probablemente, un ejemplo fundamental de un problema que no debe resolverse de manera recursiva, dado que su solución iterativa es directa y simple. Sin embargo, examinaremos los elementos que permiten dar una solución recursiva. Antes que nada, puede reconocerse un gran número de casos distintos que se deben resolver. Es decir, quiere escribirse un programa para calcular 0!, 1!, 2! Y así sucesivamente. Puede identificarse un caso "trivial" para el cual la solución no recursiva pueda obtenerse en forma directa. Es el caso de 0!, que se define como 1. El siguiente paso es encontrar un método para resolver un caso "complejo" en términos de uno mas "simple", lo cual permite la reducción de un problema complejo a uno mas simple. La transformación del caso complejo al simple resultaría al final en el caso trivial. Esto significaría que el caso complejo se define, en lo fundamental, en términos del mas simple.

Examinaremos que significa lo anterior cuando se aplica la función factorial. 4! Es un caso mas complejo que 3!. La transformación que se aplica al numero a para obtener 3 es sencillamente restar 1. Si restamos 1 de 4 de manera sucesiva llegamos a 0, que es el caso trivial. Así, si se puede definir 4! en términos de 3! y, en general, n! en términos de (n – 1)!, se podrá calcular 4! mediante la definición de n! en términos de (n – 1)! al trabajar, primero hasta llegar a 0! y luego al regresar a 4!. En el caso de la función factorial se tiene una definición de ese tipo, dado que:

n! = n * (n – 1)!

Axial, 4! = 4 * 3! = 4 * 3 * 2! = 4 * 3 * 2 * 1! = 4 * 3 * 2 * 1 * 0! = 4 * 3 * 2] *] = 24

Estos son los ingredientes esenciales de una rutina recursiva: poder definir un caso "complejo" en términos de uno más "simple" y tener un caso "trivial" (no recursivo) que pueda resolverse de manera directa. Al hacerlo, puede desarrollarse una solución si se supone que se ha resuelto el caso más simple. La versión C de la función factorial supone que esta definido (n –1)! y usa esa cantidad al calcular n!.

Otra forma de aplicar estas ideas a otros ejemplos antes explicados. En la definición de *a * b*, es trivial el caso de *b* = 1, pues a * b es igual a *a*. En general, *a + b* puede definirse en términos de *a * (b – 1)* mediante la definición *a * b = a * (b – 1) + a*. De nuevo, el caso complejo se transforma en un caso mas simple al restar 1, lo que lleva, al final, al caso trivial de *b = 1*. Aquí la recursión se basa únicamente en el segundo parámetro, *b*.

Con respecto al ejemplo de la función de Fibonacci, se definieron dos casos triviales: fib(0) = 0 y fib(1) = 1. Un caso complejo fib(n) se reduce entonces a dos más simples: fib(n –1) y fib(n –2). Esto se debe a la definición de fib(n) como fib(n –1) + fib(n – 2), donde se requiere de dos casos triviales definidos de manera directa. Fib(1) no puede definirse como fib(0) + fib(-1) porque la función de Fibonacci no está definida para números negativos.

4.6. Asignación estática y dinámica de memoria

Hasta este momento solamente hemos realizado asignaciones estáticas del programa, y más concretamente estas asignaciones estáticas no eran otras que las declaraciones de variables en nuestro programa. Cuando declaramos una variable se reserva la memoria suficiente para contener la información que debe almacenar. Esta memoria permanece asignada a la variable hasta que termine la ejecución del programa (función main). Realmente las variables locales de las funciones se crean cuando éstas son llamadas pero nosotros no tenemos control sobre esa memoria, el compilador genera el código para esta operación automáticamente.

En este sentido las variables locales están asociadas a asignaciones de memoria dinámicas, puesto que se crean y destruyen durante la ejecución del programa.

Así entendemos por asignaciones de memoria dinámica, aquellas que son creadas por nuestro programa mientras se están ejecutando y que por tanto, cuya gestión debe ser realizada por el programador.

El lenguaje C dispone, como ya indicamos con anterioridad, de una serie de librerías de funciones estándar. El fichero de cabeceras stdlib.h contiene las declaraciones de dos funciones que nos permiten reservar memoria, así como otra función que nos permite liberarla.

Las dos funciones que nos permiten reservar memoria son:

- malloc (cantidad_de_memoria);
- calloc (número_de_elementos, tamaño_de_cada_elemento);

Estas dos funciones reservan la memoria especificada y nos devuelven un puntero a la zona en cuestión. Si no se ha podido reservar el tamaño de la memoria especificado devuelve un puntero con el valor 0 o NULL. El tipo del puntero es, en principio void, es decir, un puntero a cualquier cosa. Por tanto, a la hora de ejecutar estás funciones es aconsejable realizar una operación cast (de conversión de tipo) de cara a la utilización de la aritmética de punteros a la que aludíamos anteriormente. Los compiladores modernos suelen realizar esta conversión automáticamente.

Antes de indicar como deben utilizarse las susodichas funciones tenemos que comentar el operador sizeof. Este operador imprescindible a la hora de realizar programas portables, es decir, programas que puedan ejecutarse en cualquier máquina que disponga de un compilador de C.

El operador sizeof (tipo_de_dato), nos devuelve el tamaño que ocupa en memoria un cierto tipo de dato, de esta manera, podemos escribir programas independientes del tamaño de los datos y de la longitud de palabra de la máquina. En resumen si no utilizamos este operador en conjunción con las conversiones de tipo cast probablemente nuestro programa sólo funciones en el ordenador sobre el que lo hemos programado.

Por ejemplo, el los sistemas PC, la memoria está orientada a bytes y un entero ocupa 2 posiciones de memoria, sin embargo puede que en otro sistema la máquina esté orientada a palabras (conjuntos de 2 bytes, aunque en general una máquina orientada a palabras también puede acceder a bytes) y por tanto el tamaño de un

entero sería de 1 posición de memoria, suponiendo que ambas máquinas definan la misma precisión para este tipo.

Con todo lo mencionado anteriormente veamos un ejemplo de un programa que reserva dinámicamente memoria para algún dato.

```
Class prueba
{
      public static void main(String args[])
        {
            Integer  p_int;
            Float mat;
            p_int = new Integer();
            mat = new Float();
            if ((p_int==NULL)||(mat==NULL))
            {
                  printf ("\nNo hay memoria");
                  exit(1);
            }
            /* Aquí irían las operaciones sobre los datos */
            /* Aquí iría el código que libera la memoria */
      }
}
```

Este programa declara dos variables que son punteros a un entero y a un float. A estos punteros se le asigna una zona de memoria, para el primero se reserva memoria para almacenar una variable entera y en el segundo se crea una matriz de veinte elementos cada uno de ellos un float. Obsérvese el uso de los operadores cast para modificar el tipo del puntero devuelto por malloc y calloc, así como la utilización del operador sizeof.

Como se puede observar no resulta rentable la declaración de una variable simple (un entero, por ejemplo, como en el programa anterior) dinámicamente, en primer lugar por que aunque la variable sólo se utilice en una pequeña parte del programa, compensa tener menos memoria (2 bytes para un entero) que incluir todo el código de llamada a malloc y comprobación de que la asignación fue correcta (esto seguro que ocupa más de dos bytes).

En segundo lugar tenemos que trabajar con un puntero con lo cual el programa ya aparece un poco más engorroso puesto que para las lecturas y asignaciones de las variables tenemos que utilizar el operador *.

Para termina un breve comentario sobre las funciones anteriormente descritas. Básicamente da lo mismo utilizar malloc y calloc para reservar memoria es equivalente:

mat =new Float();

mat =new Float();

La diferencia fundamental es que, a la hora de definir matrices dinámicas calloc es mucho más claro y además inicializa todos los elementos de la matriz a cero. Nótese también que puesto que las matrices se referencian como un puntero la asignación

dinámica de una matriz nos permite acceder a sus elementos con instrucciones de la forma:

NOTA: En realidad existen algunas diferencias al trabajar sobre máquinas con alineamiento de palabras.

mat[0] = 5;

mat[2] = mat[1]*mat[6]/67;

Con lo cual el comentario sobre lo engorroso que resultaba trabajar con un puntero a una variable simple, en el caso de las matrices dinámicas no existe diferencia alguna con una declaración normal de matrices.

La función que nos permite liberar la memoria asignada con malloc y calloc es free(puntero), donde puntero es el puntero devuelto por malloc o calloc.

En nuestro ejemplo anterior, podemos ahora escribir el código etiquetado como: /* Ahora iría el código que libera la memoria */

free (p_int);

free(mat);

Hay que tener cuidado a la hora de liberar la memoria. Tenemos que liberar todos los bloque que hemos asignado, con lo cual siempre debemos tener almacenados los punteros al principio de la zona que reservamos. Si mientras actuamos sobre los datos modificamos el valor del puntero al inicio de la zona reservada, la función free probablemente no podrá liberar el bloque de memoria.

4.7. Ejemplos:

4.7.1. Las Torres de Hanoi:

A continuación se verá cómo pueden usarse técnicas recursivas para lograr una solución lógica y elegante de un problema que no se especifica en términos recursivos. EL problema es el de "las torres de Hanoi", cuyo planteamiento inicial se muestra en la figura a continuación...

Hay tres postes: A, B y C. En el poste A se ponen cinco discos de diámetro diferente de tal manera que un disco de diámetro mayor siempre queda debajo de uno de diámetro menor. El objetivo es mover los discos al poste C usando B como auxiliar.

Sólo puede moverse el disco superior de cualquier poste a otro poste, y un disco mayor jamás puede quedar sobre uno menor. Considérese la posibilidad de encontrar una solución. En efecto, ni siquiera es claro que exista una.

Ahora se verá si se puede desarrollar una solución. En lugar de concentrar la atención en una solución para cinco discos, considérese el caso general de *n* discos. Supóngase que se tiene una solución para *n* – 1 discos y que en términos de ésta, se pueda plantear la solución para *n* – 1 discos. El problema se resolvería entonces. Esto sucede porque en el caso trivial de un disco (al restar 1 de *n* de manera sucesiva se producirá, al final, 1) la solución es simple: sólo hay que el único disco del poste A a C. Así se habrá desarrollado una solución recursiva si se plantea una solución para *n* discos en términos de *n* – 1. Considérese la posibilidad de encontrar tal relación. Para el caso de cinco discos en particular, supóngase que se conoce la forma de mover cuatro de ellos del poste A al otro, de acuerdo con las reglas. ¿Cómo puede completarse entonces el trabajo de mover el quinto disco? Cabe recordar que hay 3 postes disponibles.

Supóngase que se supo cómo mover cuatro discos del poste A al C. Entonces, se pondrá mover éstos exactamente igual hacia B usando el C como auxiliar. Esto da como resultado la situación los cuatro primeros discos en el poste B, el mayor en A y en C ninguno. Entonces podrá moverse el disco mayor de A a C y por último aplicarse de nuevo la solución recursiva para cuatro discos para moverlo de B a C, usando el poste A como auxilia. Por lo tanto, se puede establecer una solución recursiva de las torres de Hanoi como sigue:

Para mover *n* discos de A a C usando B como auxiliar:

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

Con toda seguridad este algoritmo producirá una solución completa por cualquier valor de *n*. Si *n* = = , el paso 1 será la solución correcta. Si *n* = = 2, se sabe entonces que hay una solución para *n* – 1 = = 1, de manera tal que los pasos 2 y 4 se ejecutaran en forma correcta. De manera análoga, cuando *n* = = 3 ya se habrá producido una solución para *n* – 1 = = 2, por lo que los pasos 2 y 4 pueden ser ejecutados. De esta forma se puede mostrar que la solución funciona para *n* = = 1, 2, 3, 4, 5,... hasta el valor para el que se desee encontrar una solución. Adviértase que la solución se desarrollo mediante la identificación de un caso trivial (*n* = = 1) y una solución para el caso general y complejo (*n*) en términos de un caso mas simple (*n* – 1).

Ya se demostró que las transformaciones sucesivas de una simulación no recursivas de una rutina recursiva pueden conducir a un programa mas simple para resolver un problema. Ahora se simulara la recursión del problema y se intentara simplificar la simulación no recursiva.

```
class hanoi

{

  public static void main (String args[])

  {
```

```
    if (args.length != 1) {
      System.err.println("error: a single integer argument
needed");
      System.exit(1);
    }
    Integer N = new Integer(args[0]);
    H_dohanoi(N.intValue(), 3, 1, 2);
    System.exit(0);
  }

  static void H_dohanoi(int n, int t, int f, int u)
  {
    if (n > 0) {
      H_dohanoi(n-1, u, f, t);
      H_moveit(f, t);
      H_dohanoi(n-1, t, u, f);
    }
  }

  static void H_moveit(int from, int to)
  {
    System.out.print("move ");
    System.out.print(from);
    System.out.print(" --> ");
    System.out.println(to);
  }
}
```

En esta función, hay cuatro parámetros, cada uno de los cuales esta sujeto a cambios en cada llamada recursiva. En consecuencia, el área de datos debe contener elementos que representen a los cuatro. No hay variables locales. Hay un solo valor temporal que se necesita para guardar el valor de *n* – 1, pero esta se puede representar por un valor temporal similar en el programa de simulación y no tiene que estar apilada. Hay tres puntos posibles a los que regresa la función en varias llamadas: el programa de llamada y los dos puntos que siguen a las llamadas recursivas. Por lo tanto, se necesitan cuatro etiquetas.

La dirección de regreso se codifica como un entero (1, 2 o 3) dentro de cada área de datos.

Considérese la siguiente simulación no recursiva de *towers*:

Ahora se simplificará el programa. En primer lugar, debe observarse que se usan tres etiquetas para indicar direcciones de regreso; una para cada una de las dos llamadas recursivas y otra para el regreso al programa principal. Sin embargo, el regreso al programa principal puede señalarse por un subdesborde en la pila, de la misma for que en la segunda versión *simfact*. Esto deja dos etiquetas de regreso. Si pudiera eliminarse otra mas, sería innecesario guardar en la pila la dirección de regreso, ya que solo restaría un punto al que se podría transferir el control si se eliminan los elementos de la pila con éxito. Ahora dirijamos nuestra atención a la segunda llamada recursiva y a la instrucción:

towers (n – 1, auxpeg, topeg, frompeg);

Las acciones que ocurren en la simulación de esta llamada son las siguientes:

1. Se coloca el área de datos vigente *a* 1 dentro de la pila.
2. En la nueva área de datos vigente a 2, se asignan los valores respectivos n – 1, auxpeg, topeg, y frompeg a los parámetros.
3. En el área de datos vigente a 2, se fija la etiqueta de retorno a la dirección de la instrucción que sigue de inmediato a la llamada.
4. Se salta hacia el principio de la rutina simulada.
5. Después de completar la rutina simulada, ésta queda lista parar regresar. Las siguientes acciones se llevan a efecto:
6. Se salva la etiqueta de regreso, /, de área de datos vigentes *a* 2.
7. Se eliminan de la pila y se fija el área de datos vigente como el área de datos eliminada de la pila, *a* 1.
8. Se transfiere el control a /.

Sin embrago, / es la etiqueta del final del bloque del programa ya que la segunda llamada a *towers* aparece en la última instrucción de la función. Por lo tanto, el siguiente paso es volver a eliminar elementos de la pila y regresar. No se volverá a hacer uso de la información del área de datos vigente *a* 1, ya que ésta es destruida en la eliminación de los elementos en la pila tan pronto como se vuelve a almacenar. Puesto que no hay razón para volver a usar esta área de datos, tampoco hay razón para salvarla en la pila durante la simulación de la llamada. Los datos se deben salvar en la pila sólo si se van a usar otra vez. En consecuencia, la seguida llamada a *towers* puede simularse en forma simple mediante:

1. El cambio de los parámetros en el área de datos vigente a sus valores respectivos.
2. El "salto" al principio de la rutina simulada.

Cuando la rutina simulada regresa puede hacerlo en forma directa a la rutina que llamó a la versión vigente. No hay razón para ejecutar un regreso a la versión vigente, sólo para regresar de inmediato a la versión previa. Por lo tanto, se elimina la necesidad de guardar en la pila la dirección de regreso al simular la llamada externa (ya que se puede señalar mediante subdesborde y simular la segunda llamada recursiva, ya que no hay necesidad de salvar y volver a almacenar al área de datos de la rutina que llamada en este momento). La única dirección de regreso que resta es la que sigue a la primera llamada recursiva.

Ya que sólo queda una dirección de regreso posible, no tiene caso guardarla en la pila para que se vuelva a insertar y eliminar con el resto de los datos. Siempre se eliminan elementos de la pila con éxito, hay una solo dirección hacia la que se puede ejecutar un "salto" (la instrucción que sigue a la primera llamada). Como los nuevos valores de las variables del área de datos vigente se obtendrán a partir de los datos antiguos de área de datos vigente, es necesario declarar una variable adicional, *temp*, de manera que los valores sean intercambiables.

4.7.2. El Problema de las Ocho Reinas:

El problema de las ocho reinas y en general de las *N* reinas, se basa en colocar *8* reinas en un tablero de *8´ 8* (o *N* en un tablero de *NxN*, si el problema se generaliza), de forma que en no puede haber dos piezas en la misma línea horizontal, vertical o diagonal, ver Figura 1.

Para ver el gráfico seleccione la opción ¨Bajar trabajo¨ del menú superior

Este programa has sido muy estudiado por los matemáticos. Se trata de un problema NP-Completo que no tiene solución para N=2 y N=3. Para N=4 tiene una única solución. Para N=8 hay más de 80 soluciones dependiendo de las restricciones que se impongan.

Una forma de abordar el problema se lleva a cabo mediante la construcción de un predicado en Prolog del tipo siguiente: reinas (N, Solución), donde N representa las dimensiones del tablero y el número de reinas y Solución es una lista que contiene la permutación de la lista de números que resuelven el problema. Los índices de dicha lista representan la columna en la que se encuentra una reina y el número que almacena la posición o índice representa la fila donde la reina está colocada. Así, para el ejemplo mostrado en la Figura 1, tenemos que R=[2,4,1,3].

Este problema es resuelto, de modo clásico, por el método de prueba y error, luego se adapta perfectamente a un algoritmo de backtracking. Básicamente, el problema se reduce a colocar una reina, e intentar repetir el proceso teniendo en cuenta la reina colocada. Si logramos poner todas las reinas el problema se ha resuelto, en caso contrario, deshacemos lo que llevamos hasta ahora y probamos con otra combinación. Por tanto, hemos de generar un conjunto de permutaciones y seleccionar aquella que cumpla los requisitos impuestos por el juego.

Veamos el código que resuelve el problema:

```
rango(N, N, [N]).
```

```
rango(N, M, [N|Cola]):-N<M, Aux is N+1, rango(Aux, M, Cola).

dame(X,[X|Xs],Xs).

dame(X,[Y|Ys],[Y|Zs]):-dame(X,Ys,Zs).

permuta([],[]).

permuta(L,[Z|Zs]):-dame(Z,L,Ys), permuta(Ys,Zs).

atacada(X,Y):-atacada(X,1,Y).

atacada(X,N,[Y|Ys]):-X is Y+N; X is Y-N.

atacada(X,N,[Y|Ys]):-N1 is N+1, atacada(X,N1,Ys).

correcta([]).

correcta([SOY]):-correcta(Y), not atacada(X,Y).

reina(N, Solucion):-rango(1,N,L1), permuta(L1,Solucion), correcta(Solucion).
```

Es muy importante comprender como funciona cada uno de los predicados para entender el funcionamiento del algoritmo general.

Prolog permite implementar los programas casi directamente a partir de las especificaciones realizadas a partir de un análisis y diseño de la solución desde un alto nivel de abstracción. Además el procedimiento de backtracking está implícito en el propio motor de inferencia, luego este paradigma se adapta perfectamente a nuestro problema.

Si analizamos y diseñamos nuestro problema tenemos que la forma de resolverlo se resume en los pasos siguientes:

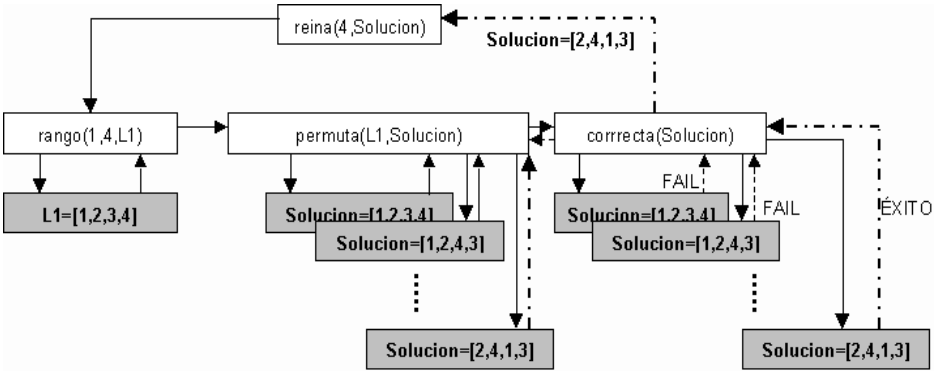
Para N, obtenemos una lista de números comprendidos entre 1 y N: [1,2,3,4,...,N].

Obtenemos una permutación del conjunto de números de la lista.

Comprobamos que la permutación es correcta.

Si la permutación no es correcta, lo que debemos hacer es volver al paso 2 para generar una permutación nueva.

Comencemos a analizar la solución implementada. El problema se resuelve con el predicado reina(N, Solución): rango(1,N,L1), permuta(L1,Solucion), correcta Solución).Como vemos es, sencillamente, una copia de las especificaciones realizadas más arriba. Se genera el rango entre 1 y N, se obtiene una permutación y se comprueba si la permutación es, o no, correcta. En el caso de que cualquier predicado del consecuente falle, la propia máquina Prolog se encarga de realizar el proceso de backtracking. Con lo cual ya tenemos cubiertos los cuatro pasos fundamentales del algoritmo. Para tener más claras las ideas, observemos el árbol de ejecución general del objetivo reina(4,Solucion)



Árbol de ejecución para el objetivo reina(4,Solucion)

He aquí otro ejemplo sobre el problema de las ocho reinas. Primero se mostrara un pseudocodigo sobre el mismo y luego su respectiva codificación en el lenguaje C.

<PRINCIPIO ensayar> (i: entero)

inicializar el conjunto de posiciones de la reina i-ésima

+-REPETIR hacer la selección siguiente

| +-SI segura ENTONCES

| | poner reina

| | +-SI i < 8 ENTONCES

| | | LLAMAR ensayar (i + 1)

| | | +-SI no acertado ENTONCES

| | | | quitar reina

| | | +-FINSI

| | +-FINSI

| +-FINSI

+-HASTA acertada O no hay más posiciones

<FIN>

Observaciones sobre el código:

1) Estudiar la función ensayar() a partir de este pseudocódigo.

2) Vectores utilizados:

int posiciones_en_columna[8]; RANGO: 1..8

BOOLEAN reina_en_fila[8]; RANGO: 1..8

BOOLEAN reina_en_diagonal_normal[15]; RANGO: -7..7

BOOLEAN reina_en_diagonal_inversa[15]; RANGO: 2..16

En C, el primer elemento de cada vector tiene índice 0, esto

es fácil solucionarlo con las siguientes macros:

#define c(i) posiciones_en_columna[(i)-1]

#define f(i) reina_en_fila[(i)-1]

#define dn(i) reina_en_diagonal_normal[(i)+7]

#define di(i) reina_en_diagonal_inversa[(i)-2]

Significado de los vectores:

c(i) : la posición de la reina en la columna i

f(j) : indicativo de que no hay reina en la fila j-ésima

dn(k): indicativo de que no hay reina en la diagonal normal

(\) k-ésima

di(k): indicativo de que no hay reina en la diagonal

invertida (/) k-ésima

Dado que se sabe, por las reglas del ajedrez, que una reina actúa sobre todas las piezas situadas en la misma columna, fila o diagonal del tablero se deduce que cada columna puede contener una y sólo una reina, y que la elección de la situación de la reina i-ésima puede restringirse a los cuadros de la columna i. Por tanto, el parámetro i se convierte en el índice de columna, y por ello el proceso de selección de posiciones queda limitado a los ocho

posibles valores del índice de fila j.

A partir de estos datos, la línea poner reina del pseudocódigo es:

c (i) = j; f (j) = di (i + j) = dn (i - j) = FALSE;

y la línea quitar reina del pseudocódigo:

f (j) = di (i + j) = dn (i - j) = TRUE;

y la condición segura del pseudocódigo:

```
f (i) && di (i + j) && dn (i - j)
```

```
/* Ficheros a incluir: */
#include <stdio.h> /* printf () */
#include <conio.h> /* getch () */
/* Macros: */
#define BOOLEAN int
#define TRUE 1
#define FALSE 0
/* Variables globales: */
BOOLEAN acertado;
int posiciones_en_columna[8];
BOOLEAN reina_en_fila[8];
BOOLEAN reina_en_diagonal_normal[15];
BOOLEAN reina_en_diagonal_inversa[15];
#define c(i) posiciones_en_columna[(i)-1]
/* rango de índice: 1..8 */
#define f(i) reina_en_fila[(i)-1]
/* rango de índice: 1..8 */
#define dn(i) reina_en_diagonal_normal[(i)+7]
/* rango de índice: -7..7 */
#define di(i) reina_en_diagonal_inversa[(i)-2]
/* rango de índice: 2..16 */
/* Prototipos de las funciones: */
void proceso (void);
void ensayar (int i);
```

```
/* Definiciones de las funciones: */

void main (void)

{

printf ("\n\nPROBLEMA DE LAS OCHO REINAS:\n ");

proceso ();

printf ("\n\nPulsa cualquier tecla para finalizar. ");

getch ();

}

void proceso (void)

{

register int i,j;

for (i = 1; i <= 8; i++)

f (i) = TRUE;

for (i = 2; i <= 16; i++)

di (i) = TRUE;

for (i = -7; i <= 7; i++)

dn (i) = TRUE;

ensayar (1);

if (acertado)

for (printf ("\n\nLA SOLUCION ES:\n\n"), i = 1; i <= 8; i++)

{

for (j = 1; j <= 8; j++)

printf ("%2d", c (j) == i ? 1 : 0);

printf ("\n");

}

else
```

```
printf ("\n\nNO HAY SOLUCION.\n");
}
void ensayar (int i)
{
int j = 0;
do
{
j++;
acertado = FALSE;
if (f (j) && di (i + j) && dn (i - j))
{
c (i) = j;
f (j) = di (i + j) = dn (i - j) = FALSE;
if (i < 8)
{
ensayar (i + 1);
if (! acertado)
f (j) = di (i + j) = dn (i - j) = TRUE;
}
else
acertado = TRUE;
}
} while (! acertado && j != 8);
}
```

Capítulo

5

Árboles Binarios

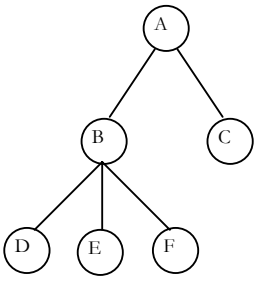
Esta unidad se centra la atención sobre una estructura de datos, *el árbol*, cuyo uso esta muy extendido es muy útil en numerosas aplicaciones. Se define formas de esta estructura de datos (árboles generales, árboles binarios y árboles binarios de búsqueda) y como se pueden representar en cualquier lenguaje como C ó Pascal, así como el método para su aplicación en la resolución de una amplia variedad de problemas. Al igual que a sucedido anteriormente con las listas, los árboles se tratan principalmente como estructuras de datos en lugar de cómo tipos de datos. Es decir, nos centramos principalmente en los algoritmos e implementaciones en lugar de en definiciones matemáticas.

Los árboles junto a los grafos constituyen estructuras no lineales. Las listas enlazadas tienen grandes ventajas o flexibilidad sobre la representación contigua de estructuras de datos(Arreglos), pero tiene gran debilidad: son listas secuénciales; es decir, están dispuestas de modo que es necesario moverse a través de ellas, una posición cada vez.

Los árboles Superan esa desventaja utilizando los métodos de puntero y listas enlazadas para su implementación. Las estructuras de datos organizadas como árboles serán muy valiosas en una gran gama grande de aplicaciones, sobre todo problema de recuperación de información.

Terminología

Un árbol es una estructura que organiza sus elementos, denominados nodos, formando jerarquías. Los científicos los utilizan los árboles generales para representar relaciones. Fundamentalmente, la relación clave es la de «padre-hijo» entre los nodos del árbol. Si existen una arista (rama) dirigida del nodo *n* al nodo *m*, entonces *n* es **padre** de *m* y *m* es **hijo** de *n*.



A Padre

B,C Hijos del nodo A y hermano entre ellos.

D, E, F Hermanos

Figura 1. Árbol General.

Cada nodo de un árbol tiene al menos un padre y existe un único nodo, denominado **raíz** del árbol, que no tiene padre. El nodo A es la raíz del árbol. Un nodo que no tiene hijos se llama **hoja** (terminal) del árbol. Las hojas del árbol de la figura 1 son C,D,E,F. Todo nodo que no es raíz, ni hoja es conocido como **interior** como es B**.**

La relación Padre-Hijo entre los nodos se generaliza en las relaciones **ascendente**(antecesor) y **descendiente**. En la Fig.1 A es un antecesor de D y por consiguiente D es un descendiente de A. Un **subárbol** de un árbol es cualquier nodo del árbol junto con todos sus descendientes como en la figura 1, B y sus descendientes son un subárbol. El **grado** (aridad) es el numero de hijos del nodo. La aridad de un árbol se define como el máximo de la aridad de sus nodos.

Se le denomina **camino** a la secuencia de nodos conectados dentro de un árbol, donde la **rama** es un camino que termina en una hoja. Para determinar el **nivel** de un nodo, se calcula por medio de los nodos que se encuentra entre el y la raíz, por consiguiente el nivel de un árbol es el numero de nodos que se encuentran entre la raiz y la hoja mas profunda; al máximo nivel de un árbol es también conocido como **Altura**.

Ejemplo 1.1

Dado el árbol general de la figura 1.2, se hace sobre el las siguientes consideraciones.

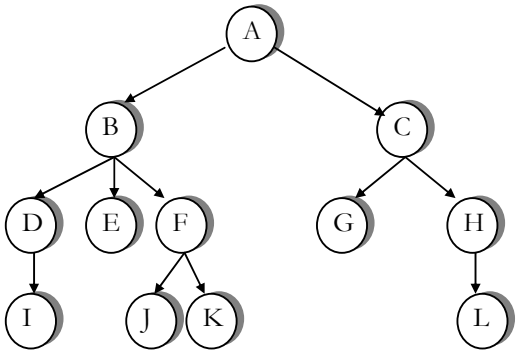


Figura 1.2 Árbol General

1. A es la raíz del árbol

2. B es hijo de A
   C es hijo de A
   D es hijo de B
   E es hijo de B
   H es hijo de L

3. A es padre de B
   B es padre de D
   D es padre de I
   C es padre de G
   L es padre de H

4. B y C son hermanos
   D, E y F son hermanos
   G y H son hermanos
   J y K son hermanos

5. I,E,J,K,G y L son nodos terminales u hojas.

6. B, D, F, C y H son nodo interiores.

7. El grados del nodo A es 2
   El grados del nodo B es 3
   El grados del nodo C es 2
   El grados del nodo D es 1
   El grados del nodo E es 0

   El grados del arbol es 3

8. El nivel de D es 1
   El nivel de B es 2
   El nivel de D es 3
   El nivel de C es 2
   El nivel de L es 4

9. La altura del del árbol es 4

Debido a su naturaleza jerárquica de los árboles se puede utilizar para representar en formaciones que sean jerárquicas por naturaleza, por ejemplo diagramas de organizaciones, árboles genealógicos, árbol de la especie humana, etc

Longitud de Camino

Se define la longitud de camino X como el numero de arcos que debe ser recorridos para llegar desde la raíz al nodo X. Por definición la raiz tiene longitud de 1, sus descendientes directos longitud de camino 2 y así sucesivamente. Considérese la figura 1.22 el nodo B tiene la longitud de camino 2, el nodo I longitud de camino 4 y el nodo Longitud de camino 3.

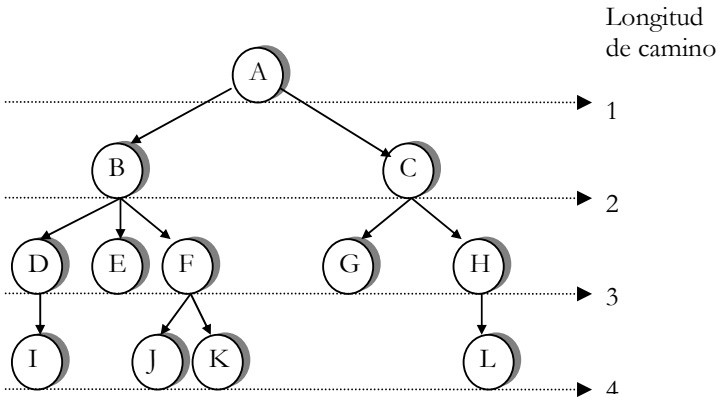


Figura 1.22 longitud de Caminos en un árbol

Longitud de Camino Interno

La longitud de camino interno es la suma de las longitudes de camino de todos los nodos del árbol. Puede calcularse por medio de las siguientes formula:

$$LCI = \sum_{i=1}^{h} n_i * i$$

donde i representa el nivel del árbol, h su altura y n_i el numero de nodos en el nivel i. La LCI del árbol de la figura 1.22 se calcula así:

$$LCI = 1 * 1 + 2 * 2 + 5 * 3 + 4 * 4 = 36$$

Ahora bien, la media de la longitud de camino interno (LCIM) se calcula dividiendo la LCI entre el numero de nodos del árbol (n). Se expresa:

$$LCIM = \frac{LCI}{n}$$

y significa el numero de arcos que deben ser recorridos en un promedio para llegar, partiendo desde la raiz, a un nodo cualquiera del árbol.

La LCIM del árbol de la figura 1.22 se calcula mediante:

$$LCIM = 36/12 = 3$$

Longitud de Camino Externo

Para definir la longitud de camino externo es necesario primero definir los conceptos **árbol extendido y nodo especial.** Un árbol extendido es aquel en el que el numero de hijos de cada nodo es igual al grado del árbol. Si alguno de los nodos del árbol no cumple con esta condición entonces debe incorporársele al mismo nodo especial; tanto como sea necesario satisfacer la condición. Los nodos especiales tienen como objetivo remplazar las ramas vacías o nulas, no pueden tener descendientes y normalmente se representan con la forma de un cuadrado. En la figura 1.23 se presenta el árbol extendido de la figura 1.22.

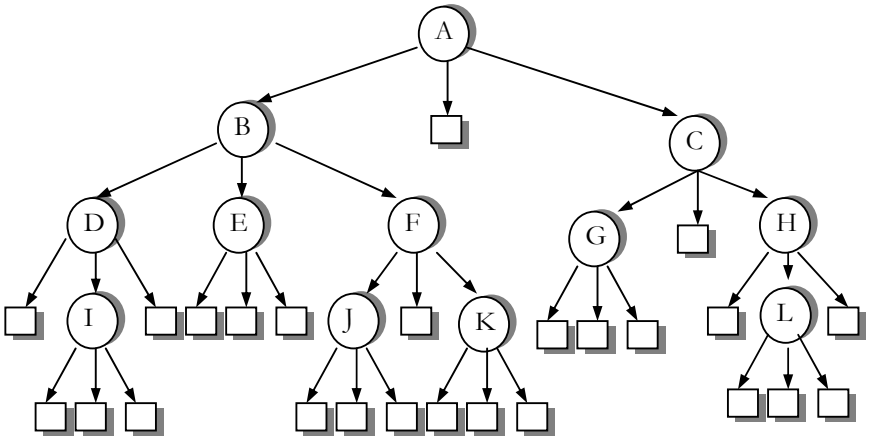


Figura 1.23 Árbol Extendido

El numero de nodos especiales de este nodo es de 25. se puede definie ahora la longitud de camino externo como la suma de las longitudes de caminos externo como la suma de las longitudes de caminos de todos los nodos especiales del árbol. Se calcula por medio de la siguiente formula:

$$LCE = \sum_{i=2}^{h+1} ne_i * i$$

donde *i* representa el nivel del árbol ,h su altura y ne_i el numero de nodos especiales en el nivel *i*. Obsérvese que *i* comienza desde 2, puesto que la raiz se encuentra en el nivel 1 y no puede ser nodo especial.

La LCE del árbol de la figura 1.23 se calcula de la siguiente manera:

$$LCE = 1 * 2 + 1 * 3 + 11 * 4 + 12 * 5 = 109$$

Ahora bien la media de la longitud d camino externo (LCEM)se calcula dividiendo LCE entre el numero de nodos especiales del árbol (ne). Se expresa:

$$LCEM = \frac{LCE}{ne}$$

y significa el numero de arcos que debe ser recorridos en promedio para llegar partiendo de la raiz, a un nodo especial cualquiera del árbol.

$$LCEM=109/25=4.36$$

Ejemplo:

Dado el árbol general de la figura 1.24 y el árbol extendido de la figura 1.25 se calcula:

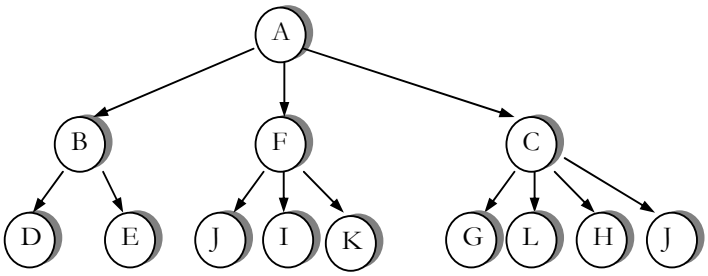


Figura 1.24 Árbol General

La longitud de camino interno:

$$LCI = 1 * 1 + 3 * 2 + 9 * 3 = 34$$

La media de la longitud de camino interno:

$$LCIM = 34 / 13 = 2.61$$

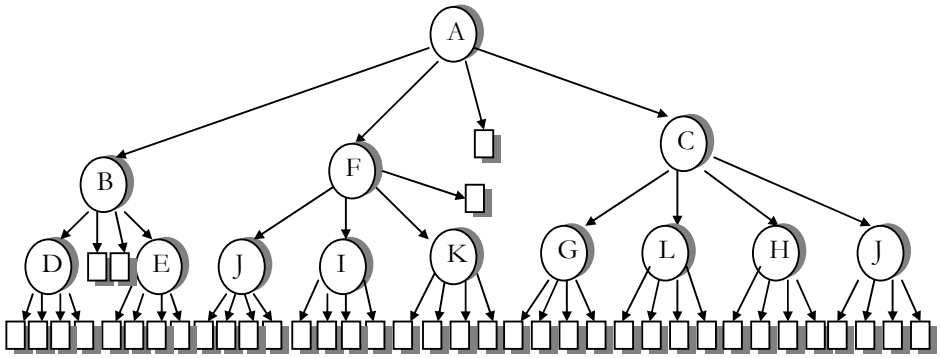


Figura 1.25 Árbol Extendido

La longitud de camino externo:

$$LCE = 1 * 2 + 3 * 3 + 36 * 4 = 155$$

Y la media de la longitud de camino externo:

LCEM = 155 / 40 = 3.87

Árboles Binarios

Un árbol ordenado es aquel en el que las ramas de los nodos del árbol están ordenadas. Los árboles ordenados de grado 2 son de especial interés puesto que representan una de las estructuras de datos más importantes en computación, conocidas como **árboles binarios.**

En un árbol binario cada nodo puede tener como máximo dos subárboles y siempre es necesario distinguir entre el subárbol izquierdo y el subárbol derecho.

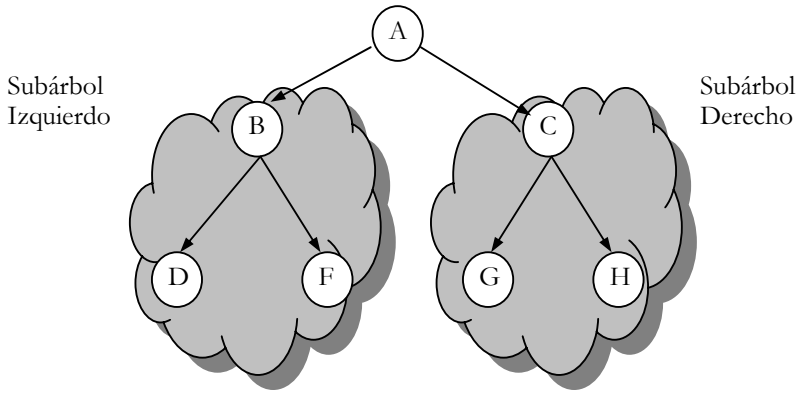


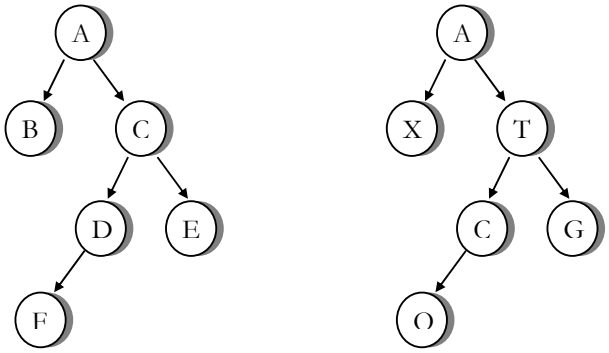
Figura 1.3 Subárboles de un árbol binario

Formalmente podemos definir un árbol binario de tipo T como una estructura homogénea que es la concatenación de un elemento de tipo T, llamada raíz, con dos árboles binarios disjuntos. Una forma particular de árbol binario puede ser la estructura vacía.

Los árboles binarios se clasifican en cuatro tipo que son : distintos, similares, equivalentes y completos. Cuando dos árboles binarios se dice que son **similares** si tiene la misma estructura y son **equivalentes** si son similares y contienen la misma información. En caso contrario se dice que estos árboles son **distintos.** Un arbol binario esta equilibrado si la altura de los dos subárboles de cada nodo del arbol se diferencia en una unidad como máximo.

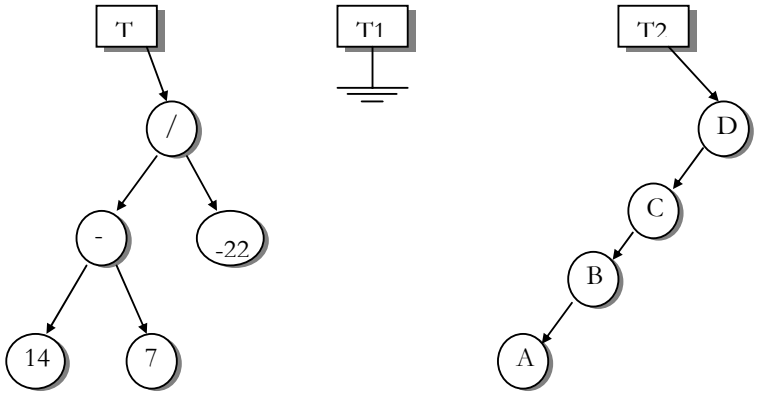
Altura(Subárbol izquierdo) - Altura(Subárbol derecho) ≤ 1

El procedimiento de arbol binarios equilibrados es mas sencillo que los árboles no equilibrados.



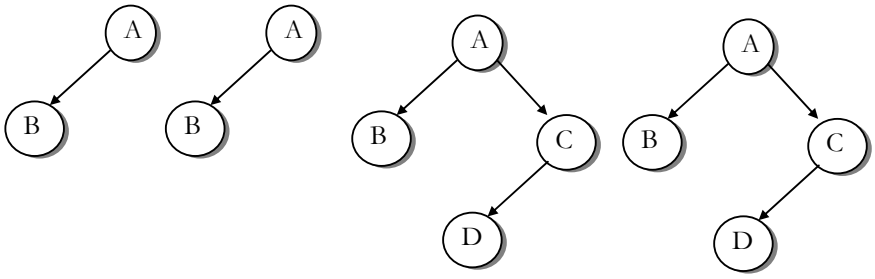
a) b)

Figura 1.4 Árboles binarios Similares



a) b) c)

figura 1.5 Arboles binarios de diferentes alturas: (a) altura 3, (b) arbol vacio altura 0, (c) altura 4



a) b)

Figura 1.6 Árboles binarios equivalentes

Se define un árbol **completo** (lleno) como un árbol en el que todos sus nodos, excepto los del ultimo nivel, tienen dos hijos ; el subárbol izquierdo y el subárbol derecho.

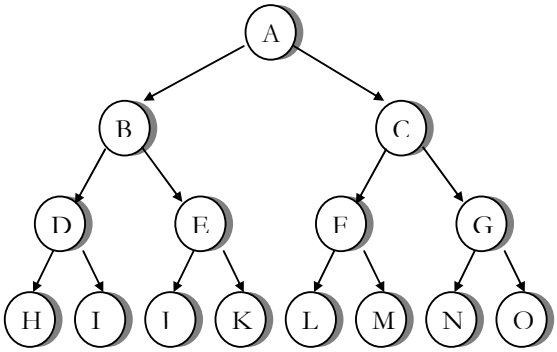


Figura 1.7 Árbol binario Completo de altura 4

Se puede calcular el numero de nodos de un arbol binario completo de altura h, aplicando la siguiente formula

$$\text{Numero de nodos}_{AB} = 2^h - 1$$

Representación grafica de un árbol general a binario.

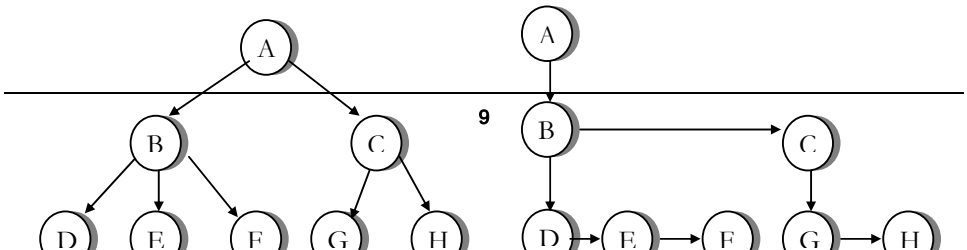
Los árboles binarios representa una de las estructuras de datos mas importantes en la computación. Esto por su dinamismo, la no-linealidad entre sus elementos y por su sencilla programación. En lugar que en los árboles generales ya es imprescindible deducir cuantas ramas o caminos se desprenden de un nodo en un momento dado. Por ello y dado que los árboles binarios siempre se cuelgan como máximo de dos subárboles su programación es mas sencilla.

Afortunadamente existe una técnica sencilla para la conversión de un árbol general a formato de árbol binario. El algoritmo de conversión tiene tres pasos fáciles:

1. Debe enlazarse los hijos de cada nodo en forma horizontal(los hermanos)

2. Debe enlazarse en forma vertical el nodo padre con el hijo que se encuentra mas a la izquierda. Además debe eliminarse el vinculo de ese padre con el resto de sus hijos.

3. Debe rotarse el diagrama resultante, aproximadamente 45 grados a la izquierda y así se obtendrá el árbol binario correspondiente.



Representación en memoria

Los árboles binarios pueden ser representados de modos diferentes :

- Mediante punteros (memoria dinámica)
- Mediante arreglos (memoria estática)

Sin embargo se utilizará y se explicará la primera forma, puesto que es la mas natural para tratar este tipo de estructuras. los nodos del arbol binario seran representados como registros, que contendran como minimo tres campos. En un campo almacenara la información del nodo. Los dos restantes se utilizaran para apuntar a los subárboles. Dado el siguiente nodo T:

IZQ	INFO	DER

Este tiene tres campos:

IZQ: campo donde se almacena la dirección del subárbol izquierdo del nodo T.

INFO: campo donde se almacena la información de interés del nodo.

DER: campo donde se almacena la dirección del subárbol derecho del nodo T.

La definición del árbol binario en lenguaje algorítmico es como sigue:

```
public class NodoArb{
        private int dato;
        private int FE;
        private NodoArb li,ld;

        public NodoArb(int d){
                this.dato=d;
        }

        public void setDato(int x){
                this.dato=x;
        }
        public int getDato(){
                return dato;
        }
        public NodoArb getLI(){
                return li;
        }
        public NodoArb getLD(){
                return ld;
        }
        public void setLI(NodoArb liga){
                this.li=liga;
        }
        public void setLD(NodoArb liga){
                this.ld=liga;
        }

}
```

Nota:

Se utiliza el símbolo ^para representar el concepto de dato tipo puntero

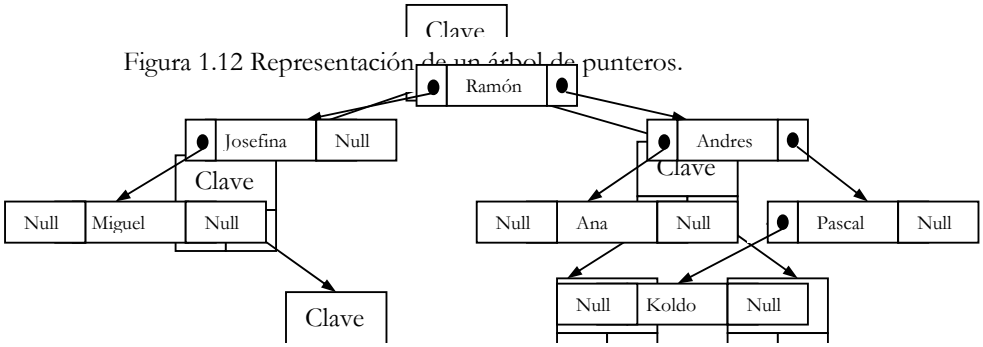


Figura 1.12 Representación de un árbol de punteros.

Figura 1.13 Representación de un árbol con una estructura de punteros

Otro  método de

representación, y uno de los más fáciles es mediante tres arreglos lineal es paralelos que contemplan el campo de la información y los dos punteros de ambos subárboles. Utilizando el mismo método anterior

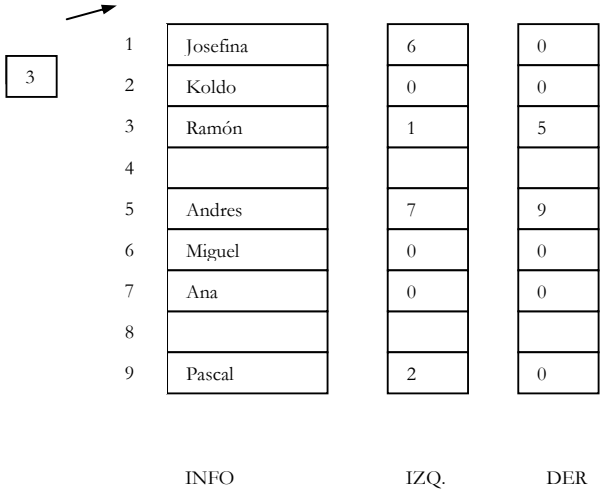


Figura 1.14 árbol binario como arreglo

Otra manera representar por medio de arreglos es utilizando el algoritmo siguiente:

La raíz del árbol se guarda en ARBOL[1]
Si un nodo n esta en ARBOL[i] entonces
*Su hijo izquierdo se pone en ARBOL[2*i]*
*Y su hijo derecho en ARBOL[2*i+1]*
Si un subárbol esta vació, se le da el valor de NULL

La raíz del árbol se guarda en ARBOL[1]Este sistema requiere mas posición de memoria que nodos tiene el árbol. Así la transformación necesita un arreglo con 2^{h+2} elementos, si el árbol tiene una profundidad h. Utilizando el ejemplo anterior que tiene una altura de 3requerirá 32 posiciones.

Arreglo Árbol

1	Ramón
2	Josefina
3	Andrés
4	Miguel
5	NULL
6	Ana
7	Pascal
8	NULL
9	NULL
10	
11	
12	NULL
13	NULL
14	Koldo
15	NULL

16	
17	
18	
....	
28	NULL
29	NULL
30	
31	
32	

Figura 1.15 árbol binario con un solo arreglo

Recorrido de un Árbol Binario

Una de las operaciones más importantes a realizar en un árbol binario es el recorrido de los mismos. Recorrer significa visitar los nodos del árbol en forma sistemática; de tal manera que todos los nodos del mismo sean visitados una sola vez. Existen tres formas diferentes de efectuar el recorrido de un árbol binario:

1. visitar el nodo (N)

2. Recorrer el subárbol izquierdo(I)

3. Recorrer el subárbol derecho(D)

Según sea la estrategia a seguir los recorridos se conocen como **enorden** (Inorder), **preorden** (preorder) y **postorden** (postorder). Cada uno de estos son de naturaleza recursiva.

preorden (nodo-izdo-dcho) (**NID**)

enorden (izdo-nodo-dcho) (**IND**)

postorden (izdo-dcho-nodo) (**IDN**)

Recorrido enorden

Si el árbol no esta vacío, el método implica los siguientes pasos:

1. Recorre el subárbol izquierdo (I)

2. visita el nodo raiz (N)

3. Recorre el subárbol derecho (D)

El algoritmo correpondientes es .

```
public void preorden(NodoArb p){
        if(p!=null){
                System.out.print(p.getDato()+" ");
                preorden(p.getLI());
                preorden(p.getLD());
        }
}
```

El árbol de la figura 1.16los nodos se han numerado en el orden en el que son visitados durante el recorrido enorden. El primer subárbol recorrido es el izquierdo del nodo raiz. Este subárbol consta de los nodos B, D y E, que es a su vez otro arbol con el nodo B como raiz, por lo que siguiendo el orden IND ese visita primero D, a continuación B y por ultimo E (derecha).después de visitar el subárbol izquierdo se visita el nodo raiz A y por ultimo se visita el subárbol derecho que consta de los nodos C, F y G. Y continua con el orden IND para el subárbol derecho.

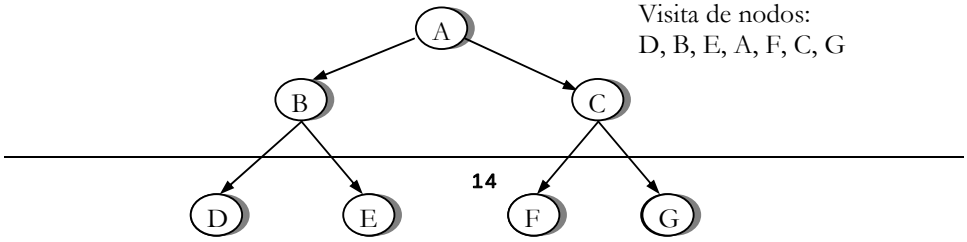


Figura 1.16. Recorrido de ENORDEN árbol binario

Recorrido preorden

El recorrido preorden (NID) conlleva los siguientes pasos:

1. visitar el nodo (N)
2. Recorrer el subárbol izquierdo(I)
3. Recorrer el subárbol derecho(D)

El algoritmo correspondiente es:

```
PREORDEN(NODO)
Inicio
    Si NODO ≠ NULL entonces
        Visualizar NODO^.INFO
        PREORDEN( NODO^.IZQ)
        PREORDEN( NODO^.DER)
    FIN SI
FIN
```

Si utilizamos el recorrido preorden del árbol de la figura 1.17 se visita primero la raíz (nodo A). A continuación se visita el subárbol, que consta de los nodos B, D y E. Dado que el subárbol es a su vez un árbol, se visitan los nodos utilizando el orden NID. Por consiguiente, se visitan primero el nodo B, después D (izquierdo) y por ultimo E (derecho).

A continuación se visita subárbol derecho A, que es un árbol que contiene los nodos C, F y G. De nuevo siguiendo el orden NID, se visita primero el nodo C, a continuación F y por ultimo G.

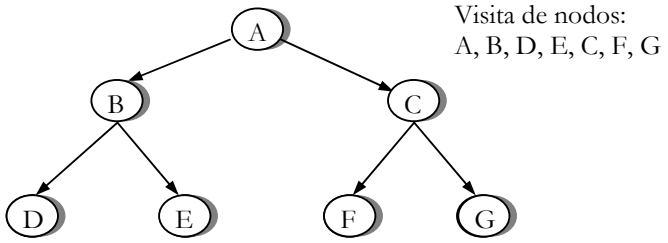


Figura 1.17. Recorrido de PREORDEN árbol binario

Recorrido postorden

El recorrido del postorden (IDN) realiza los pasos siguientes:

1. Recorrer el subárbol izquierdo(I)
2. Recorrer el subárbol derecho(D)
3. visitar el nodo (N)

El algoritmo correspondiente es:

```
POSTORDEN(NODO)
Inicio
    Si NODO ≠ NULL entonces
        POSTORDEN (NODO^.IZQ)
        POSTORDEN (NODO^.DER)
        Visualizar NODO^.INFO
    FIN SI
FIN
```

Si se utiliza el recorrido postorden del árbol de la figura 1.18, se visitara primero el subárbol izquierdo A. Este subárbol consta de los nodos B, D y E y siguiendo el orden IDN, se visitara primero D(izquierdo), luego E (derecho) y por ultimo B(nodo). A continuación se visitara el subárbol derecho que consta con los nodos C, F y G. Siguiendo el orden IDN para este árbol, se visita primero F, después a G y por ultimo C. Finalmente se visita el raíz A.

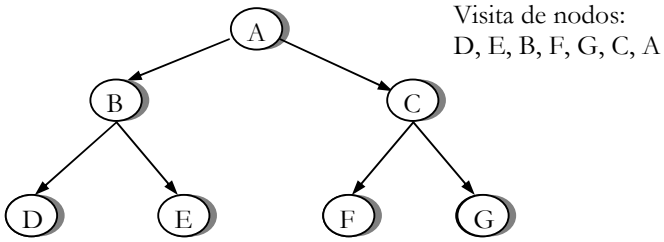
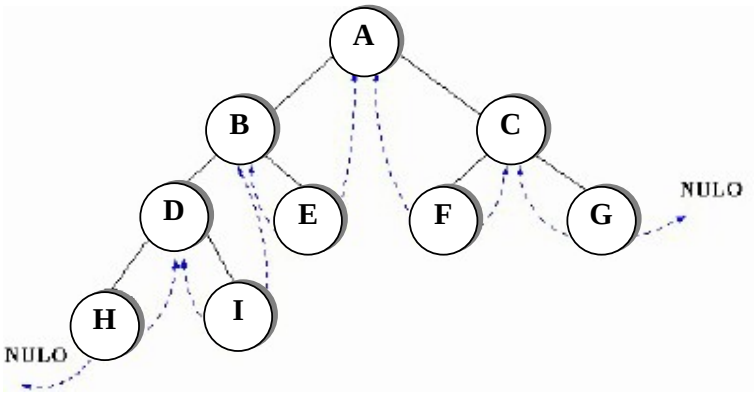


Figura 1.18. Recorrido de POSTORDEN árbol binario

Árboles Enhebrados

Al estudiar la representación enlazada de un árbol binario es fácil observar que existen muchos enlaces nulos. De hecho, existen mas enlaces nulos que punteros con valores reales. En concreto, para un árbol con n nodos, existen n+1 enlaces nulos de

los 2n enlaces existentes en la representación (mas de la mitad). Como el espacio de memoria ocupada por los enlaces nulos es el mismo que el ocupado por los no nulos, podría resultar conveniente utilizar estos enlaces nulos para almacenar alguna información de interés para la manipulación del árbol binario. Una forma de utilizar estos enlaces es sustituir por punteros a otros nodos del árbol. En particular, los enlaces nulos situados en el subárbol derecho de un nodo se suelen reutilizar para apuntar al sucesor de ese nodo en un determinado recorrido del árbol, por ejemplo enorden, mientras que los enlaces nulos en el mismo tipo de recorrido. Si para algunos nodos no existe predecesor (porque es el primero del recorrido) o su sucesor (porque es el ultimo), se mantiene con valor nulo el enlace correspondiente.



La ventaja de este tipo de representación no es sólo el mejor aprovechamiento de la memoria disponible, sino por la posibilidad de un acceso rápido al sucesor (o al predecesor) de un nodo, que como hemos visto anteriormente, puede ser una operación frecuentemente necesaria. En general, los algoritmos que impliquen recorrer el árbol se podrán diseñar de una manera más eficiente.

Para poder manejar correctamente toda la información de la que se dispone en la representación **enhebrada** o **hilvanada** (con hilos) del árbol binario, es necesario poder distinguir entre lo que son punteros normales, que representan las relaciones reales entre los nodos, y lo que son **hilos**. Esto se puede hacer añadiendo dos campos boléanos (en el lenguaje C por valores que similares) a la representación de los nodos del árbol. Estos nuevos campos indicarán si los enlaces izquierdo y derecho son hilos o no.

La estructura de un nodo, siguiendo la representación utilizada en lenguaje C, vendría dada por la siguiente declaración:

```
Type
   Struct Nodo
   {
           Int Info;
           Struct Nodo *Izq, *Der;
           Int HiloIzq, HiloDer;
   }
   typedef nodo arbol;
Var
```

```
Árbol raiz;
```

Con el objeto de no mantener absolutamente ningún enlace nulo y para facilitar el recorrido del árbol, se suele añadir a la estructura un nodo raíz que no contiene información real (como se ha visto en las listas). El nodo raíz que representa el árbol vacío tendrá la estructura que se muestra en la figura:

Hilo izq	Izq	Info	Der	Hilo Der
Cierto/falso	-	X	-	Cierto/falso

En general, la utilización de enlaces hilos simplifica los algoritmos de recorrido del árbol, por lo que resultan recomendables cuando el recorrido del árbol (o los movimientos parciales dentro del mismo) es una operación frecuente. Sin embargo, desde el punto de vista de la manipulación general del árbol, hay que tener en cuenta que la inserción de nuevos nodos debe mantener en todo momento esta estructura de enlaces con los nodos sucesor y predecesor, y que cada vez que se inserte un nodo se deberá comprobar si existen enlaces de este tipo que deban ser modificados o creados, además de los punteros normales que relacionan los nodos del árbol.

Árboles en Montón

Un bosque representa un conjunto normalmente ordenado de uno o más árboles generales. Es posible utilizar el algoritmo de conversión analizado en el punto anterior con algunas modificaciones, para representar un bosque en un árbol binario.

Considérese el bosque, constituido por tres árboles generales. Los pasos que deben aplicarse para lograr la conversión del bosque a un árbol binario son los siguientes:

1. Debe enlazarse en forma horizontal las raíces de los distintos árboles generales.

2. Debe enlazarse los hijos de cada nodo en forma horizontal (los hermanos)

3. Debe enlazarse en forma vertical el nodo padre con el hijo que se encuentra más a la izquierda. Además debe eliminarse el vínculo de ese padre con el resto de sus hijos.

4. Debe rotarse el diagrama resultante, aproximadamente 45 grados a la izquierda y así se obtendrá el árbol binario correspondiente.

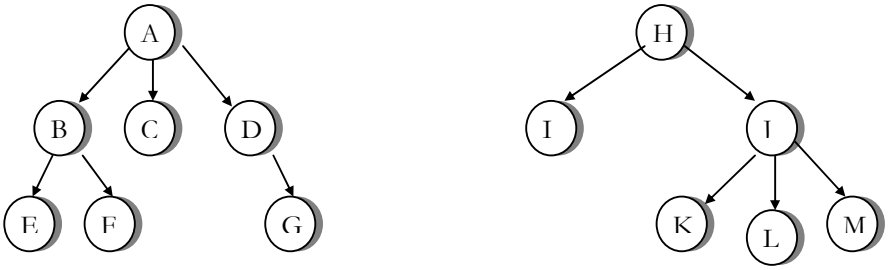


Figura 1.9 Bosque de árboles generales

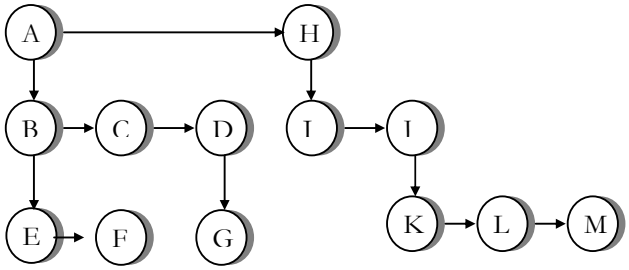


Figura 1.10 Conversión de un bosque en un árbol binario luego de aplicar los pasos 1,2 y 3

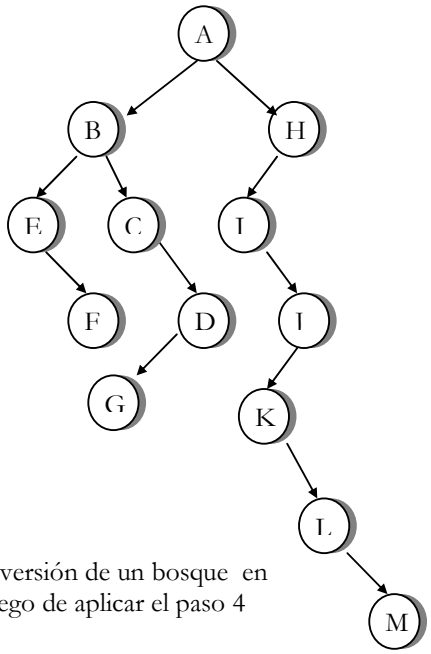


Figura 1.11 conversión de un bosque en árbol binario luego de aplicar el paso 4

Creación de un Árboles Binarios

Hasta el momento se analizado los distintos tipos de recorrido de un árbol binario. Sin embargo, debe recordarse que antes de recorrer un árbol, debe de crearse el mismo en memoria. Un algoritmo muy simple que crea los nodos de un árbol binario en memoria es el siguiente. El siguiente algoritmo como los mostrados anteriormente utiliza recursividad para la inserción de un nuevo nodo, donde la variable Nodo es de tipo puntero. La primera vez Nodo es creado en el programa principal:

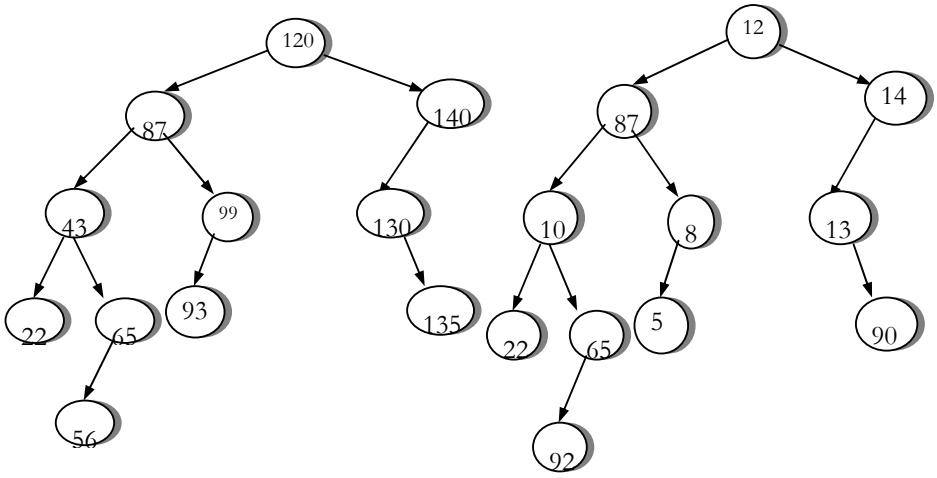
```
public NodoArb inserta(NodoArb p,int d){
        if(p==null){
                NodoArb n=new NodoArb(d);
                return n;
        }
        else
                if(p.getDato()>d)
                        p.setLI(inserta(p.getLI(),d));
                else
                        if(p.getDato()<d)
                                p.setLD(inserta(p.getLD(),d));
                        else
                                System.out.println("Ya existe el dato");
        return p;
}
```

Árboles Binarios de Búsqueda

Los árboles vistos hasta el momento no tienen orden definido; sin embargo los árboles binarios ordenados tiene sentido. Estos árboles se denominan árboles binarios de búsqueda, debido a que se pueden buscar en ellos un termino utilizando un algoritmo de búsqueda binaria similar al empleado en los arreglos estructuras de datos 1.

Un árbol binario de búsqueda es un árbol binario, que puede estar vacío, y que si es no vacío cumple con las siguientes propiedades:

- Todos los nodos están identificados por una clave y no existen dos elementos con la misma clave.
- Las claves de los nodos del subárbol izquierdo son menores que la clave del nodo raíz.
- Las claves de los nodos del subárbol derecho son mayores que la clave del nodo raíz.
- Los subárboles izquierdo y derecho son también árboles binarios de búsqueda.



a) b)

Figura 1.19 a) Árbol binario de búsqueda b) no es árbol binario de búsqueda.

En las figura 1.19 se muestran dos árboles, en donde el primero de ellos es un árbol binario de búsqueda ya que cada nodo esta ordenado dependiendo de su valor. Observe que en el nodo 120 al lado izquierdo se encuentra los valores menores que el, y al lado derecho los mayores. Estas mismas reglas se cumplen para cada uno de sus nodos.

Operaciones en Árboles Binarios de Búsqueda

De lo expuesto hasta ahora se deduce que los árboles binarios tienen naturaleza recursiva y en consecuencia las operaciones sobre los árboles son recursivas, si bien siempre tenemos la opción de realizarlas de forma iterativa. Estas operaciones son:

- Búsqueda de un nodo.
- Inserción de un nodo.
- Supresión de un nodo.

Búsqueda

La definición de árbol binario de búsqueda especifica un criterio en la estructuración del árbol en función de las claves de los nodos. En este caso, existe un criterio de ordenación de los nodos. Por lo tanto, será bastante simple describir un método eficiente de búsqueda que explote esta ordenación.

Suponer que se busca un elemento que posea la clave x. La búsqueda comenzará por el nodo raíz del árbol. La clave de ese nodo informará por dónde debe continuar la búsqueda, ya no es necesario recorrer exhaustivamente todos los nodos del árbol. Si la clave del nodo es igual a x, la búsqueda finaliza con éxito. Si la clave es menor que x, se sabe que si existe un nodo en el árbol que posea como clave el valor x deberá estar en el subárbol derecho, por lo tanto la búsqueda deberá continuar por esa parte del árbol. Si, por el contrario, la clave del nodo es mayor que x, entonces la búsqueda deberá continuar por el subárbol izquierdo. El proceso continuará hasta que se encuentre un nodo con clave igual a x o un subárbol vacío, en cuyo caso se puede asegurar que no existe ningún nodo con clave x en el árbol. Este método de búsqueda sugiere seguir un esquema recursivo.

```
Busca(Nodo, infor)
Inicio
  SI Nodo = NULO entonces
     Regresa(NULO)
  Sino
     Si  infor =  Nodo^.info entonces
          Regresa(Nodo)
     Sino
        Si infor < Nodos^.info entonces
            Regresa(Busca(Nodo^.Izq,infor))
        Sino
            Regresa(Busca(Nodo^.Der, infor))
        Fin Si
     Fin Si
  Fin Si
Fin
```

En otro caso de búsqueda, la recursividad puede ser fácilmente sustituida por un esquema iterativo mediante la utilización de un bucle de repetición "mientras". En ese caso, el algoritmo de búsqueda iterativo sería:

```
Búsqueda(Nodo,infor)
Inicio
  Aux = Nodo
  Enc = FALSO
  Mientras (aux <> NULL) Y (Enc = FALSO) hacer
    Si Aux^.info = x entonces
       Enc = CIERTO
    Sino
       Si x < Aux^.info entonces
          Aux = Aux^.Izq
       Sino
          Aux = Aux^.Der
    Fin Si
  Fin Mientras
  Regresa(Enc)
Fin
```

El método de búsqueda se asemeja mucho al método de búsqueda binaria sobre arreglos ordenados, tras la comparación con un elemento se puede decidir en qué región de la estructura se puede encontrar la información buscada, descartándose el resto de los elementos de la estructura. De esta forma, se reduce considerablemente el número de comparaciones necesarias para localizar el elemento.

Inserción

La inserción de un nuevo nodo en un árbol binario de búsqueda debe realizarse de tal forma que se mantengan las propiedades del árbol. De modo que lo primero que hay que hacer es comprobar que en el árbol no existe ningún nodo con clave igual a la del elemento que se desea insertar. Si el elemento no existe la inserción se realiza en un nodo en el que al menos uno de los dos punteros Izq o Der tengan valor NULL.

Para realizar la condición anterior se desciende en el árbol a partir del nodo raíz , dirigiéndose de izquierda a derecha de un nodo, según el valor a insertar sea inferior o superior al valor del campo Info de este nodo. Cuando se alcanza un nodo del arbolen que no se pueda continuar, el nuevo elemento se engancha a la izquierda o derecha de este nodo en función de su valor sea inferior o superior al del nodo alcanzado.

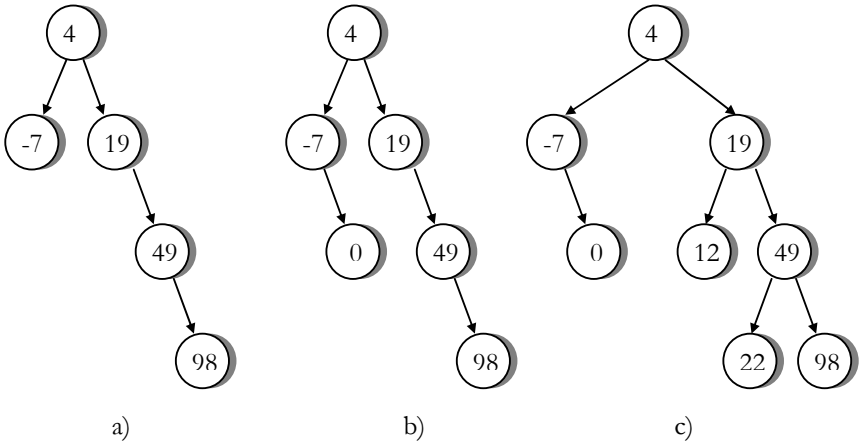


Figura 1.20 Inserción de un árbol de búsqueda binaria: (a) inserta 100, (b) inserta 0, (c) inserta 22 y 12

En este manual se manejan dos algotimos de inserción, dando la opcion al usuario de escoger entre uno u otro.

```
public NodoArb inserta(NodoArb p,int d){
        if(p==null){
                NodoArb n=new NodoArb(d);
                return n;
        }
        else
                if(p.getDato()>d)
                        p.setLI(inserta(p.getLI(),d));
                else
                        if(p.getDato()<d)
                                p.setLD(inserta(p.getLD(),d));
                        else
                                System.out.println("Ya existe el dato");
        return p;
}
```

Eliminación

La operación de eliminación de un nodo es también una extensión de la operación de búsqueda, si buen es mas compleja que la inserción debido a que el nodo a suprimir puede ser cualquiera y la operación de su supresión deba mantener la estructura de árbol binario de búsqueda después de la eliminación de datos. Los pasos a seguir son:

1. buscar en el árbol para encontrar la posición de nodo a eliminar.

2. reajustar los punteros de sus antecesores si el nodo a suprimir tiene menos de dos hijos, o subir a la posición que este ocupa el nodo descendiente con la clave inmediatamente superior o inferior con objeto de mantener la estructura del árbol binario.

La eliminación de una clave y su correspondiente nodo, presenta dos casos claramente diferenciados. En primer lugar en un nodo hoja o tiene un único descendiente, resulta una tarea fácil, ya que lo único que hay que hacer es asignar el enlace desde el nodo padre (según el camino de búsqueda) el descendiente del nodo a eliminar (o bien NULL). En segundo lugar, que el nodo tenga los dos descendientes. Para mantener la estructura del árbol de búsqueda tenemos dos alternativas, remplazar la clave a eliminar por la mayor de las claves menores, o bien remplazar por la menor de las mayores. Se elige la primera opción, lo que se supone bajar por la derecha de la rama izquierda del nodo a eliminar hasta llegar al nodo hoja, que será el que este mas a la derecha dentro del subárbol izquierdo de la clave a borrar.

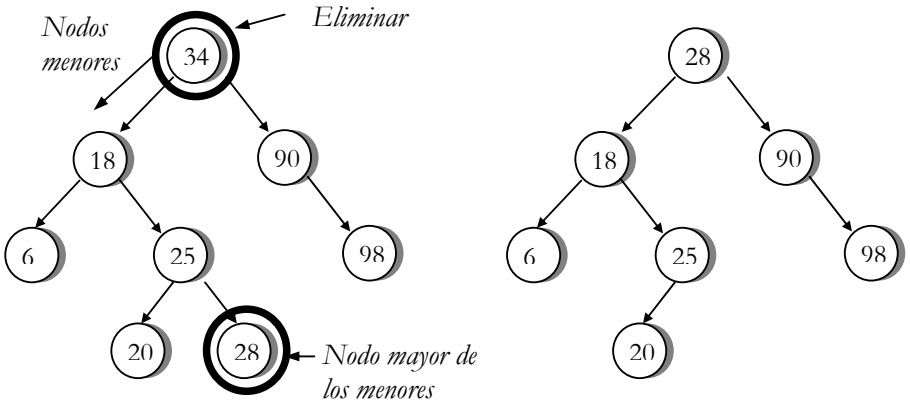


Figura 1.21 Eliminación de un árbol de búsqueda binaria

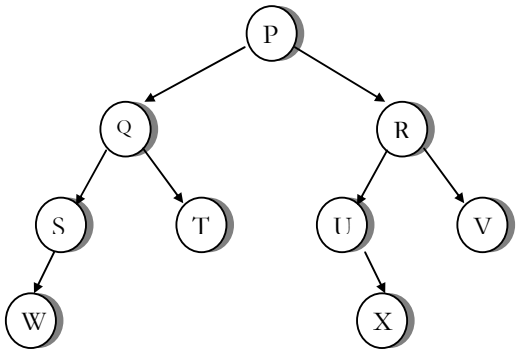
La figura 1.20 nos muestra la eliminación del nodo 34, lo primero que se debe de hacer es localizar el nodo que se desea eliminar, posteriormente se toma como raíz y para elimínalo se puede hacer de dos formas. El primer método se va por el lado izquierdo donde se encuentran los nodos menores a la raiz, encontrándose el nodo 18 como el primer nodos, a continuación se toma el nodo mas mayor de los menor que vienen siendo el nodo mas a la derecha, o sea que se van pasando por el nodo 25, y después al mas a la derecha el 28. Se puede cambiar ese nodo por el 34 y no cambia nada el árbol. El siguiente algoritmo muestra la

```
public NodoArb elimina(NodoArb p, int d){
        NodoArb q=null,r=null;
        if(p==null)
                System.out.print("Elemento no existe");
        else
                if(p.getDato()<d)
                        p.setLD(elimina(p.getLD(),d));
                else
                        if(p.getDato()>d)
                                p.setLI(elimina(p.getLI(),d));
                        else
                                if(p.getLD()==null&&p.getLI()==null)
                                p=null;
                                else
                                        if(p.getLI()!=null){
                                                q=p.getLI();
                                                while(q.getLD()!=null){
                                                        r=q;
                                                        q=q.getLD();
                                                }
                                                if(r==null)
                                                        p=q;
                                                else{

p.setDato(q.getDato());
                                                        r.setLD(p.getLI());
                                                }
                                        }
                                        else
                                                p=p.getLD();
        return p;
}
```

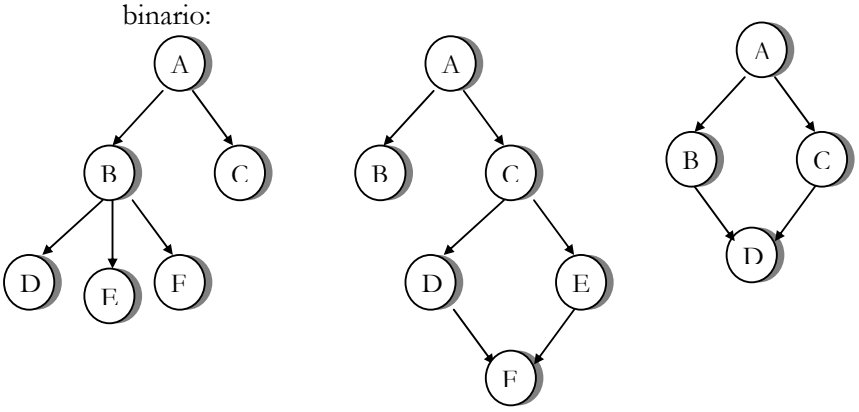
Ejercicios

1.1. Considerando el árbol siguiente:

a) ¿Cuál es su altura?
b) ¿Esta el árbol equilibrado?¿Porque?
c) Lista todos los nodos hoja
d) ¿Cuál es el predecesor inmediato (padre)del nodo U?
e) Lista los hijos del nodo R
f) Lista los sucesores del nodo R



1.2. Explica porque cada una de las siguientes estructuras no es un árbol binario:



1.3. Para cada una de las siguientes listas de letras:
1. M, Y, T, E, R.
2. R, E, M, Y, T.
3. T, Y, M, E, R.
4. C, O, R, N, F, L, A, K, E, S.

a) Dibuja el árbol de búsqueda que se construye cuando las letras se insertan en el orden dado.
b) Realiza recorridos enorden, preorden y postorden del árbol y mostrar la secuencia de letras que resultan en cada caso.

1.4. Para el árbol del ejercicio 1.1, recorrer cada árbol utilizando los ordenes siguientes: NDI, DNI, DIN.

1.5. Dibujar los árboles binarios que representan las siguientes operaciones:
a) (A + B) / (C – D)
b) A + B + C / D
c) A - (B - (C – D) / (E + F))
d) (A + B) * ((C + D) / (E + F))
e) (A - B) / ((C * D) - (E / F))

1.6. El recorrido preorden de un cierto árbol binario produce
ADFGHKLPQRWZ
Y en recorrido enorden produce
GFHKDLAWRQPZ
Dibujar el árbol binario.

1.7. Escribir una función recursiva que cuete las hojas de un árbol binario.

1.8. escribir un programa que procese un árbol binario cuyos nodos contengan caracteres y a partir del siguiente menú de opciones:

I	(seguido de un carácter)	: Insertar un carácter
B	(seguido de un carácter)	: Buscar un carácter
RE		: Recorrido en orden
RP		: Recorrido en preorden
RT		: Recorrido en postorden
SA		: Salir

1.9. Escribir una función que tome un árbol como entrada y devuelva el numero de hijos del árbol.
1.10. Escribir una función booleana a la que se le pase un puntero a un árbol binario y devuelva verdadero si el árbol es completo y falso en caso contrario.
1.11. Diseñar una función recursiva que devuelva un puntero a un elemento en un árbol binario de búsqueda.
1.12. Diseñar una función iterativa que encuentre un elemento en un árbol binario de búsqueda.

Problemas

1.13. Escribir un programa que lea un texto de longitud indeterminada y que produzca como resultado la lista de todas las palabras diferentes contenidas en el texto, así como su frecuencia de aparición.
1.14. Se dispone de un árbol binario de elementos de tipo entero. Escribir funciones que calculen:
   a) La suma de sus elementos
   b) La suma de sus elementos que son múltiplos de 3.
1.15. Escribir una función booleana IDÉNTICOS que permita decir si dos árboles binarios son iguales.
1.16. Diseñar un programa interactivo que permita dar de alta, baja, listar, etc., en un árbol binario de búsqueda.
1.17. Construir un procedimiento recursivo para encontrar una determinada clave en un árbol binario.
1.18. Dados dos árboles binarios de búsqueda indicar mediante un programa si los árboles tiene o no elementos comunes.
1.19. Dado un árbol binario de búsqueda construir su árbol espejo (árbol espejo es aquel que se construye a partir de uno dado, convirtiendo el subárbol derecho en izquierdo y viceversa).

Árboles Balanceados

Se utilizan los árboles binarios de búsqueda para almacenar datos organizados jerárquicamente. Sin embargo en muchas ocasiones la inserción y eliminación de los elementos en el árbol no ocurren en el orden predecible; es decir, los datos no esta organizados jerárquicamente.

Aquí se estudiara el tipo de árboles adicionales : árboles balanceados (equilibrados) o árboles AVL, como también se conocen, que ayudan eficientemente a resolver estas situaciones.

Árboles Balanceados

Al manejarse los árboles binarios de búsqueda se menciono que es una estructura sobre la cual se puede realizar eficiente mente las operaciones de búsqueda, inserción y eliminación. Sin embargo, si el árbol crece o decrece descontroladamente, el rendimiento puede disminuir considerablemente. El caso mas desfavorable se produce cuando se inserta un conjunto de claves ordenadas de forma ascendente o descendente, como se muestra en la figura 2.1

Es de notar que el numero promedio de comparaciones que debe realizarse para localizar una determinada clave, en un árbol binario de búsqueda con crecimiento descontrolado es N/2, cifra que muestra el rendimiento muy pobre de la estructura.

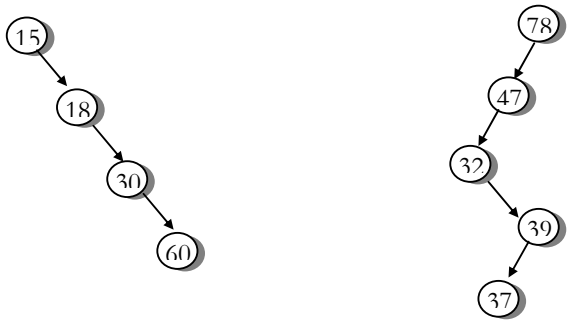
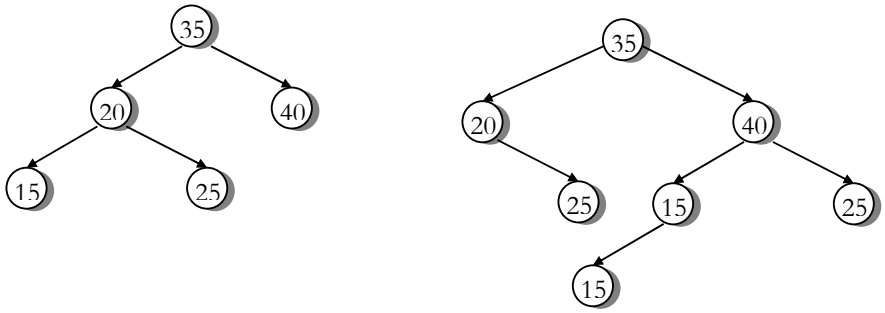


Figura 2.1 árboles binaros de búsqueda con crecimiento descontrolado

Con el objeto de mejorar el rendimiento de la búsqueda surgen los árboles balanceados (equilibrados). La idea central de estos es la de realizar reacomodos o balanceos, después de la inserción o eliminación de los elementos. Estos árboles también reciben el nombre de árboles AVL en honor a sus inventores, dos matemáticos rusos, G.M. Adelson-Velskii y E.M Landis.

Formalmente se define un árbol balanceado como un árbol binario de búsqueda, en el cual se debe cumplir las siguientes condiciones : "Para todo nodo T del árbol , la altura de los subárboles izquierdo y derecho no debe diferir en mas de una unidad ". En la figura 2..2 que muestran dos diagramas con árboles balanceados.



a) b)

Figura 2.2 Dos árboles balanceados a) con altura 3. b)Con altura 4

Creación de un Árbol Equilibrado de N Claves

Considerando la formación de los árboles balanceados, se parece mucho a la secuencia de los números Fibonacci:

$$a(n) = a(n-2) + a(n-1)$$

de igual forma, un árbol de fibonacci (árbol equilibrado) puede definise:

1. Un árbol vació es un árbol de fibonacci de altura 0.

2. Un nodo único es un árbol de fibonacci de altura 1.

3. A_{h-1} y A_{h-2} son árboles de fibonacci de altura h-1 y h-2 entonces $A_h = <A_{h-1}, X, A_{h-2}>$ es un árbol de fibonacci de altura h.

El numero de nodos de A_h viene dado por la sencilla relación recurrente.

$N_0=0$

$N_1=1$

$N_h=N_{h-1}+1+N_{h-2}$

Para conseguir un árbol AVL, con un numero dado N, de un nodo hay que distribuir equitativamente los nodos a la izquierda y a la derecha de un nodo dado. En definitiva, es seguir la relación de recurrencia anterior que podemos expresar recursivamente:

1. Crea nodo raiz.

2. Genera un subárbol izquierdo con $n_i=n/2$ nodos del nodo raiz utilizando la misma estrategia.

3. Genera un subárbol derecho con $n_d=n/2$ nodos del nodo raiz utilizando la misma estrategia.

El Pseudocódigo del progama GeneraBalanceado implementa la creación de un árbol equilibrado de n claves.

```
/* Pseudocódigo Genera Balanceado */

Ptrae = ^Nodo  /* Puntero  de tipo  registro Nodo*/
Nodo  = Record
  Clave : integer
  Izq, Der :Nodo
Fin Record

R : Ptrae
N : Integer
```

```
Modulo ÁrbolEq ( N: integer ) : Ptrae
Nuevo: Ptrae
Niz, Ndr : Integer
Inicio
  Si N=0 Entoces
    Regresa (NULL)
  Sino
    Niz=n div 2
    Ndr=N-Niz-1
    Crear(Nuevo)
    Escribe"Clave"
    Leer Nuevo^.Clave
    Nuevo^.Izdo=Arboleq(Niz)
    Nuevo^.Drch=Arboleq(Ndr)
   Regresa Nuevo
 Fin  Si
Fin

Modulo Dibujaarbol ( R : Ptrae, H: integer )
I : Integer
Inicio
  Si  R< > NULL Entonces
    Dibuja árbol(R.^.Izdo, H+1)
    Para I=1 to H Hacer
       Escribe " "
    Fin Para
     Escribe R.^.Clave
     Dibuja árbol(R.^.Drch, H+1)
  Fin  Si
Fin

Inicio
  Escribir "¿Numeros de nodos  del árbol ?"
  Leer N
  R = Arboleq(N)
  Dibujaarbol(R,0)
Fin
```

Factor de Equilibrio

Para poder comenzar en el proceso de equilibrio de un árbol binario que fue afectado por una inserción se debe conocer que es el factor de equilibrio (FE). Cada nodo, además de la información que se pretende almacenar, debe tener los dos punteros a los árboles derecho e izquierdo, igual que los ABB, y además un miembro nuevo: el factor de equilibrio.
El factor de equilibrio es la diferencia entre las alturas del árbol derecho y el izquierdo:

$$FE = H_{RD} - H_{RI}$$

Por definición, para un árbol AVL, este valor debe ser -1, 0 ó 1. Utilizando las figuras 2.1 y 2.2 se mostrara el FE de cada uno de los nodos.

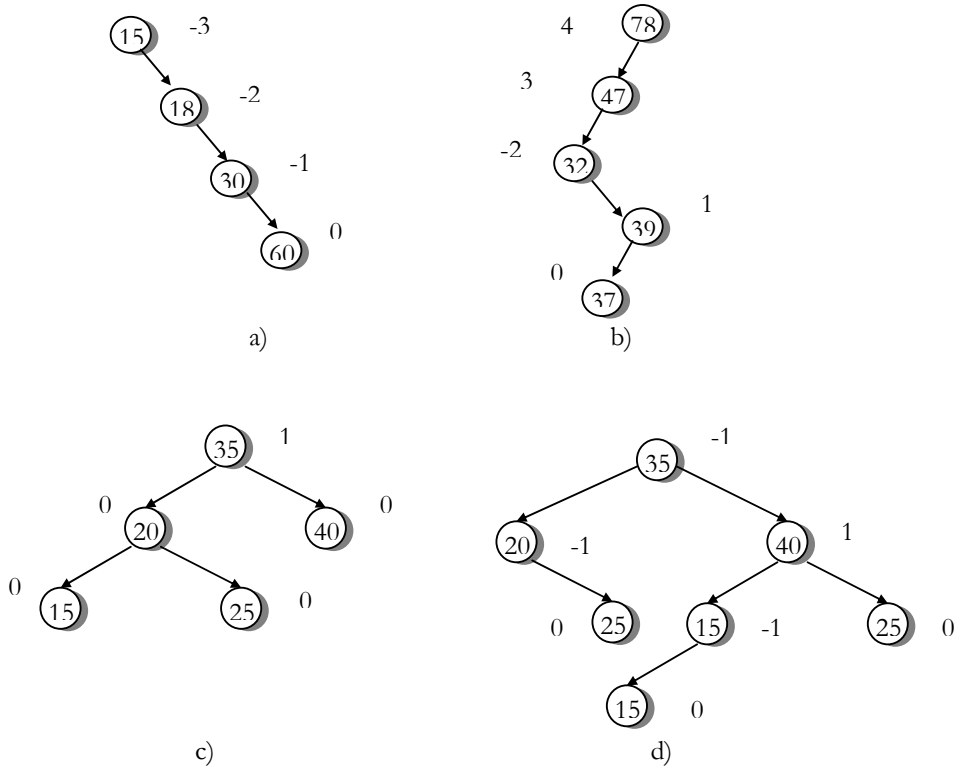


Figura 2.3 Los árboles a) y b) no son árboles equilibrados. Los arboles c) y d) son árboles equilibrados

Obsérvese que en la figura 2.3 d) el FE del nodo 40 es de –1, puesto que la altura del subárbol derecho es igual a 1 y la altura del subárbol izquierdo 2.

$$FE_{40} = 2 - 1 = 1$$

El FE de 20 se calcula como:

$$FE_{20} = 0 - 1 = -1$$

Cada uno de los nodos de los árboles de la figura 2.3 muestra el FE de cada nodo. Y como se menciono anteriormente el FE debe de cumplir que para ser un árbol equilibrado todos los árboles del nodo deben de tener un FE entre –1 a 1. Por tal motivo los árboles a)y b) no son árboles equivalentes porque contiene nodos con mayor FE de 1 o menor que-1. En cambio los árboles c) y d) se encuentran en el rango permitido. Cada uno de los nodos debe de guarda su FE, o sea que la estructura del nodo cambiaria de la siguiente manera:

```
Enlace = ^Nodo
Nodo = registro
          Info : tipo de dato
          FE : integer
          Izq, Der : enlace
Fin registro
```

Y en lenguaje C se mostraría :

```
Struct Nodo
{
          int Info;
          int FE;
          Struct  Nodo *Izq, Der;
}
```

Casos Para la Inserción de un Nuevo Nodo

Ahora se va a insertar un nuevo nodo , y como ocurre con las inserciones , como nodo hoja. Al insertar un elemento (nodo) en un árbol balanceado debe distinguirse los siguientes casos:

1. La rama izquierda (RI) y derecha (RD) del árbol tiene la misma altura (H_{RI}= H_{RD}), por lo tanto:

   1.1. Si se inserta un elemento en RI entonces H_{RI} será mayor a H_{RD}.

   1.2. Si se inserta un elemento en RD entonces H_{RD} será mayor a H_{RI}.

   Observe que en cualquiera de los dos casos mencionados no se viola el criterio de equilibrio.

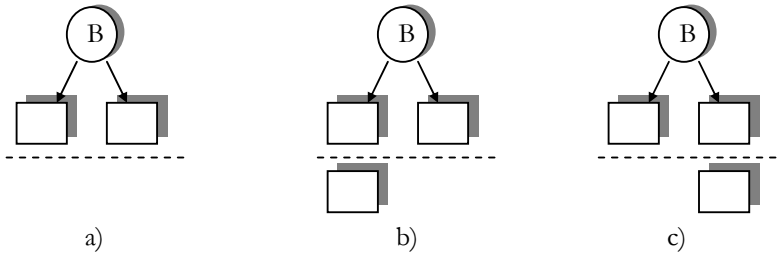


Figura 2.4 Los diferentes situaciones de inserción de la primera caso a) Caso 1 , b) Caso 1.1 y c) Caso 1.2

2. La rama izquierda (RI) y la derecha (RD) del árbol tienen altura diferente($H_{RI} = H_{RD}$):

   2.1. Supóngase que $H_{RI} < H_{RD}$:

      2.1.1. Si se inserta un elemento en RI entonces H_{RI} será igual a H_{RD}. (las ramas tiene la misma altura, por lo que será mejor el equilibrio).

      2.1.2. Si se inserta un elemento en RD entonces se rompe el criterio de equilibrio del árbol y es necesario reestructurarlo.

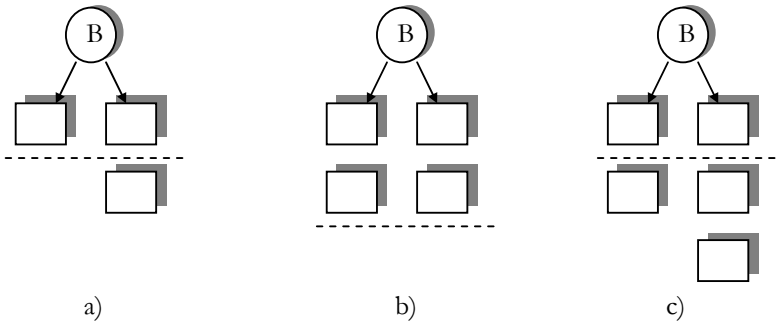


Figura 2.5 Los diferentes situaciones de inserción de la primera caso a) Caso 2.1, b) Caso 2.1.1 y c) Caso 2.1.2

   2.2. Supóngase que $H_{RI} > H_{RD}$.

      2.2.1. Si se inserta un elemento en RI, entonces se rompe el criterio de equilibrio del árbol y es necesario estructurarlo.

      2.2.2. Si se inserta un elemento en RD entonces H_{RD} será igual a H_{RI}. (las ramas tiene la misma altura, por lo que será mejor el equilibrio).

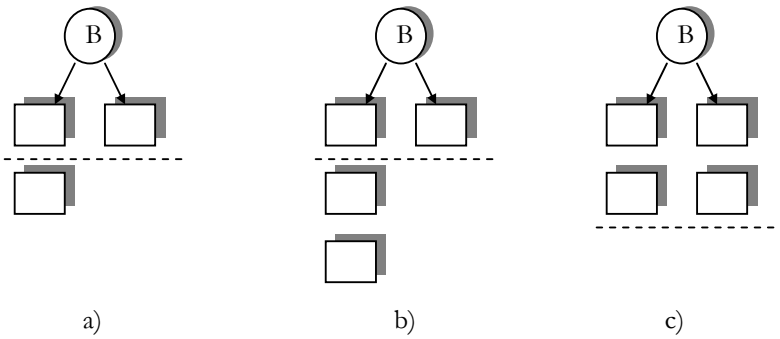


Figura 2.6 Los diferentes situaciones de inserción de la primera caso a) Caso 2.2. , b) Caso 2.2.1 y c) Caso 2.2.2

Reestructuración del Árbol Balanceado (Rotación)

En todo este proceso hay que recordar que estamos trabajando con árboles binarios de búsqueda con un campo adicional que es el factor de equilibrio (FE), por lo que el árbol resultante debe seguir siendo un árbol de búsqueda .

Para añadir un nuevo nodo al árbol, primero se baja por el árbol siguiendo el camino de búsqueda determinado por el valor de la clave, hasta llegar al lugar donde hay que insertar el nodo. Este nodo se inserta como hoja del árbol, por lo que su FE será 0.

Una vez hecho esto, se regresa por el camino de búsqueda marcado, calculando el factor de equilibrio de los distintos nodos que forman el camino. Este calculo hay que hacerlo porque al añadir el nuevo nodo al menos una rama de un subárbol a aumentado en altura. Puede ocurrir que el nuevo FE de un nodo viole el criterio de balanceo. Si es así, debe de reestructurar el árbol (subárbol) de raíz dicho nodo.

El proceso de regreso termina cuando se llega a la raíz del árbol o cuando se realiza la reestructuración en un nodo del mismo; en cuyo caso no es necesario determinar el FE de los resultantes nodos, debido a que dicho factor queda como el que tenia antes de la inserción. Pues el efecto de reestructuración hace que no aumente la altura.

Las violaciones del FE de un nodo pueden darse de cuatro maneras distintas. El equilibrio o reestructuración se realizan con el desplazamiento particular de los nodos implicados, rotando los nodos. La rotación puede ser simple o compuesta. El primer caso involucra dos nodos y el segundo caso afecta a tres. Si la rotación es simple puede realizarse por las ramas derechas (DD) o por las ramas izquierdas (II). Si la rotación es compuesta puede realizarse por las ramas derecha e izquierda (DI) o por las ramas izquierda y derecha (ID).

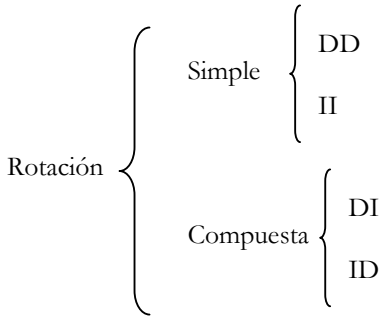
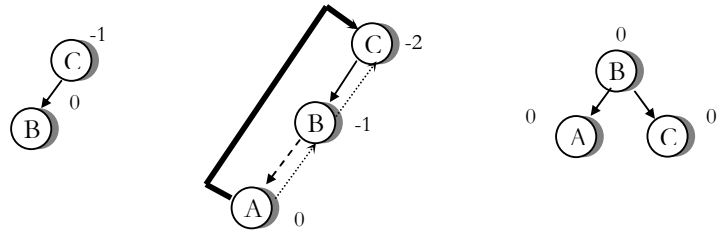


Figura 2.7 Diagrama de los tipos de Rotaciones

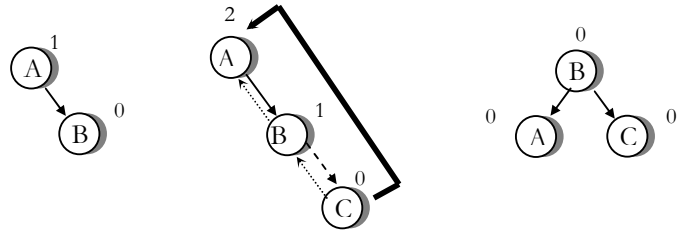
Los cuatro casos de violación del balanceo de un nodo se muestran en las siguientes figuras, la línea continua (____) marca el estado de los nodos del árbol antes de realizar la inserción. La línea discontinua (- - - - -) indica el nuevo elemento insertado. La línea de puntos (.....) marca el camino de regreso hasta que se detecta el desequilibrio del árbol. La línea gruesa (━━━━) indica el movimiento de los nodos en la rotación, así como la reestructuración que equilibra el árbol:



a) b) c)

Figura 2.8. Rotación Simple II de Árbol Balanceado

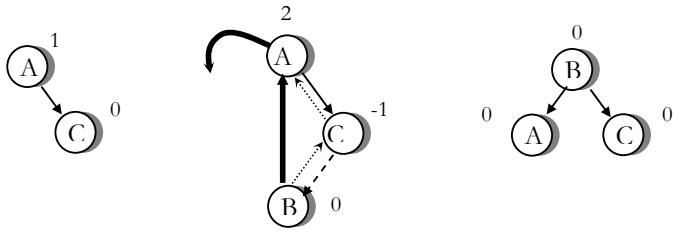
En la figura 2.8 a) es el árbol original, en el b) se inserta el nodo A (Siguiendo el camino de búsqueda) y cambia el factor de equilibrio. En el c) el nodo C se ha rotado el criterio de equilibrio, la reestructuración consiste en rotación a la izquierda, izquierda (rotación simple II).



a) b) c)

Figura 2.9. Rotación Simple DD de Árbol Balanceado

En la figura 2.9 a) es el árbol original, en el b) ahora se inserta el nodo C (Siguiendo el camino de búsqueda) y cambia el factor de equilibrio. En el c) el nodo A se ha rotado el criterio de equilibrio, la reestructuración consiste en rotación a la derecha, derecha (rotación simple DD).



a) b) c)

Figura 2.10. Rotación Compuesta DI de Árbol Balanceado

En la figura 2.10 a) es el árbol original, en el b) ahora se inserta el nodo B (Siguiendo el camino de búsqueda) y cambia el factor de equilibrio. En el c) el nodo A se ha rotado el criterio de equilibrio, la reestructuración consiste en rotación a la derecha, izquierda (rotación simple DI).

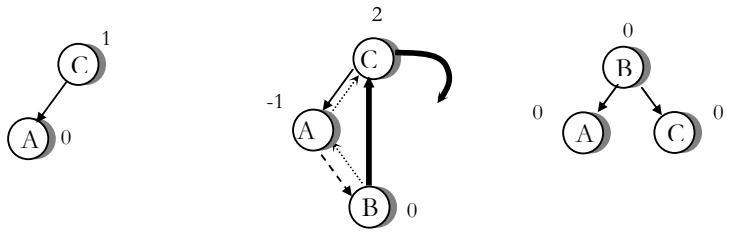


Figura 2.11. Rotación Compuesta ID de Árbol Balanceado

En la figura 2.11 a) es el árbol original, en el b) ahora se inserta el nodo B (Siguiendo el camino de búsqueda) y cambia el factor de equilibrio. En el c) el nodo C se ha rotado el criterio de equilibrio, la reestructuración consiste en rotación a la izquierda, derecha (rotación simple ID).

Formación de un Árbol Balanceado

Se simulara la inserción de los nodos en un árbol de búsqueda balanceado, partiendo del árbol vació. Por comodidad que el campo clave es entero. El nodo se representara como una estructura, donde contendrá cada uno de los campos representados anteriormente. Los punteros N, N1 y N2 referencia al nodo que viola las condiciones de equilibrio y a los descendientes en el camino de búsqueda.

Supóngase que se va a insertar las siguientes claves en un árbol balanceado:

Claves : 65, 50, 23, 70, 82, 68, 39.

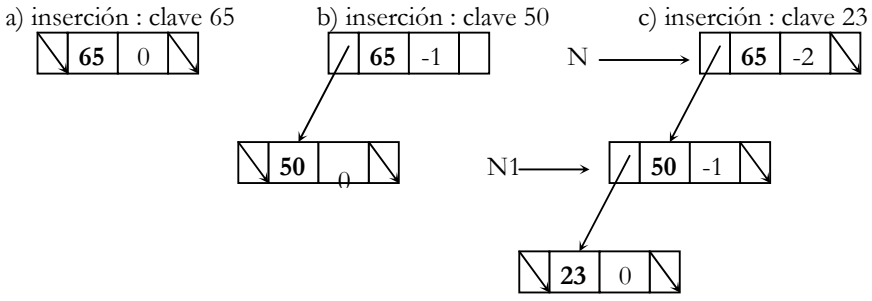


Figura 2.12. Seguimiento de la inserción en el árbol

Una vez insertado el nodo con la clave 23, al regresar por el camino de búsqueda cambia los factores de equilibrio, así el del nodo 50 pasa a –1 y en el nodo 65 se hace pasa a –2. Se ha roto el criterio de equilibrio y debe de reestructurarse. Se apunta con N a la clave 65 y con N1 la rama izquierda de dicha clave. Luego de verificar los

factores de equilibrio –1 y –2 deben de realizarse una rotación de los nodos II para rehacer el equilibrio. Los movimientos de los punteros para realiza esta rotación II es el siguiente:

```
1) N^.Izq = N1^.Der
2) N1^.Der = N
3) N = N1
```

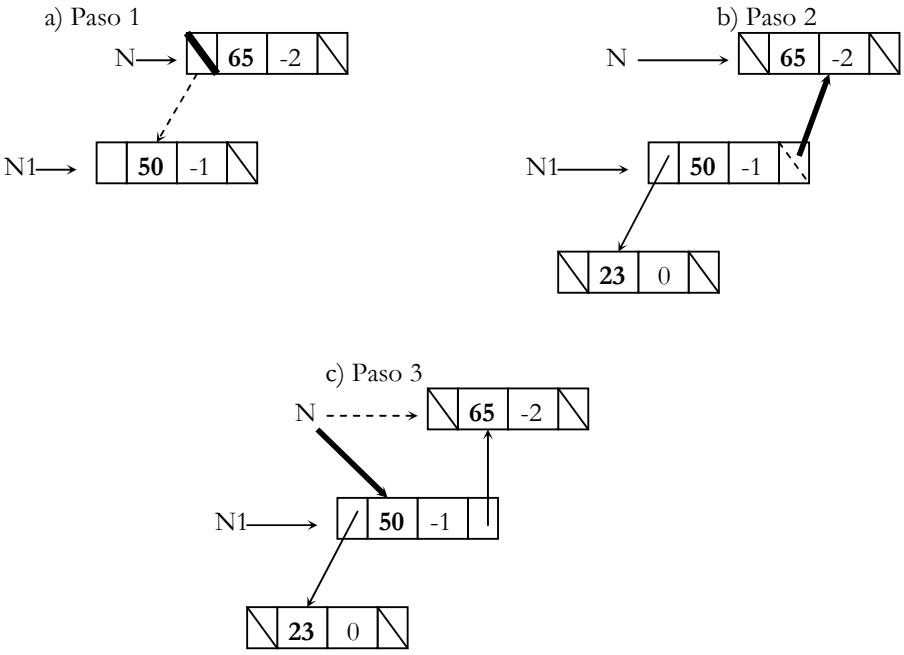


Figura 2.13. Seguimiento del algoritmo de rotación II.

El orden que llevan cada uno de los pasos es el mismo que lleve el algoritmo, con la línea discontinua (- - - -) se puede apreciar el valor que se tenia antes de cada el paso. Con la línea normal (_____) son los valores que no se afectan en ese paso y con la línea gruesa (━━━) se muestra el paso después de elaborarse.

Respecto al FE de los nodos afectados el reacomodo, este será siempre 0 (cero) en el caso de las rotaciones simples

```
N^.FE = 0
N1^.FE = 0
```

Luego de efectuar el reacomodo, el árbol queda así:

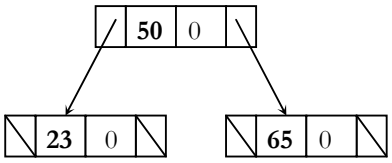


Figura 2.14. Árbol reestructurado por II

Al regresar siguiendo el camino de búsqueda se modifica el FE de los nodos 65 y 50 pero el equilibrio del árbol se mantiene.

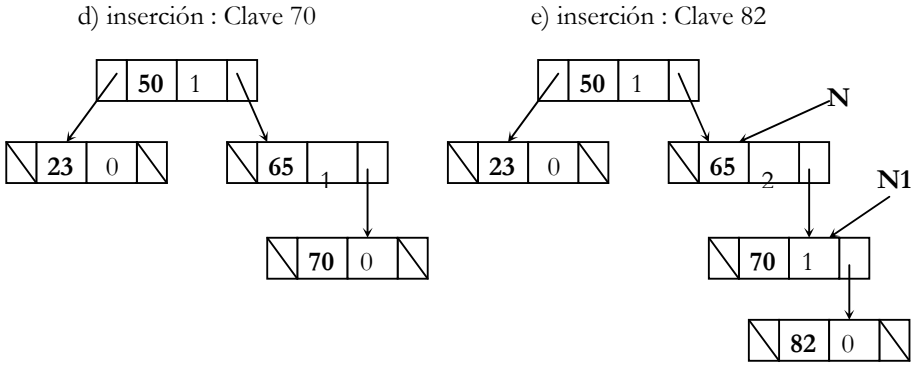
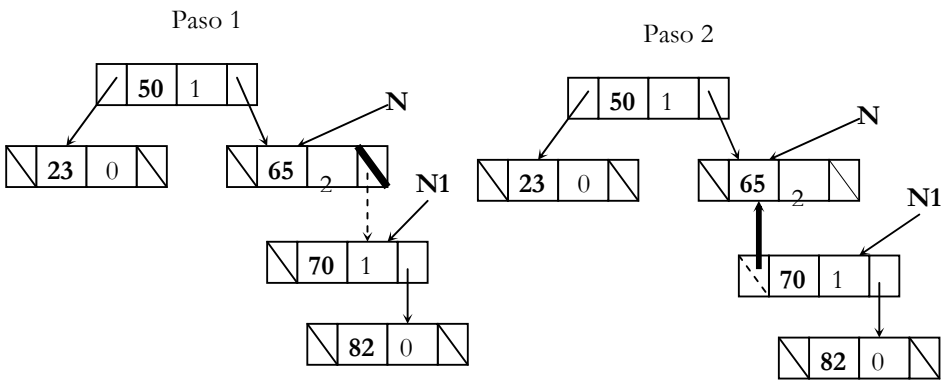


Figura 2.14. continuación del seguimiento de la inserción de Árbol

Una vez insertado el nodo con la clave 82 a la derecha del nodo 70 , y regresar por el camino de búsqueda para así calcular los nuevos factores de equilibrio, se observa que dichos factor queda incrementado en 1 pues la inserción ha sido por la derecha. En el nodo con la clave 65 queda roto el equilibrio. Para reestructurar se realizo una rotación DD. Los movimientos de los puntos para realizar esta rotación DD son:

```
1) N^.Der = N1^.Izq
2) N1^.Izq = N
3) N = N1
```



Paso 3

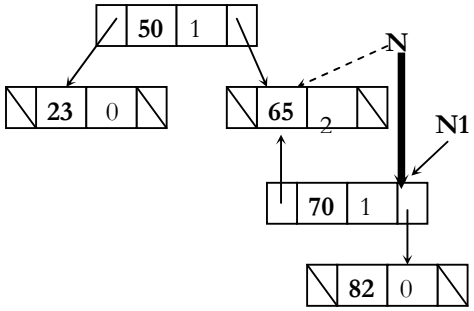


Figura 2.15. Seguimiento del algoritmo de rotación DD.

Como el anterior, una vez realizada la rotación, los factores de equilibrio de los nodos implicados serán 0 como ocurre en las rotaciones simples, y el árbol que da de la siguiente forma:

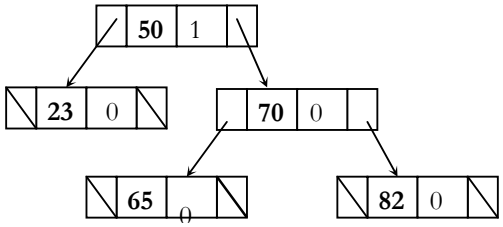


Figura 2.16. árbol reestructurado

f) inserción : clave 68

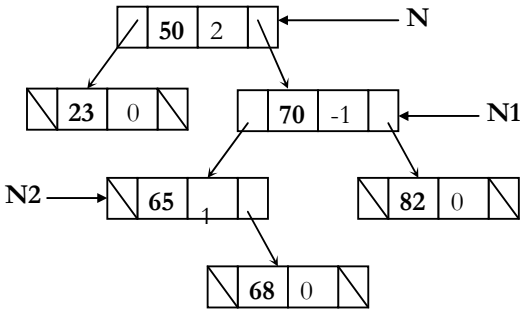


Figura 2.17. Continuación del seguimiento de la inserción de Árbol

Para insertar el nodo 68 se sigue el camino : derecha de 50, izquierda 70 y se inserta ala derecha del nodos 65, al regresar por el camino de búsqueda los factores de equilibrio en 1 si se fue por la rama derecha, se decremento en 1 si se fue por la rama izquierda.

En el nodo 50, el balanceo se ha roto. La rotación de los nodos para restablecer el equilibrio de DI. Los movimientos de los punteros para realizar esta rotación DI son :

1) N1^.Izq = N2^.Der
2) N2^.Der = N1
3) N^.Der =N2^.Izq
4) N2^.Izq = N
5) N = N2

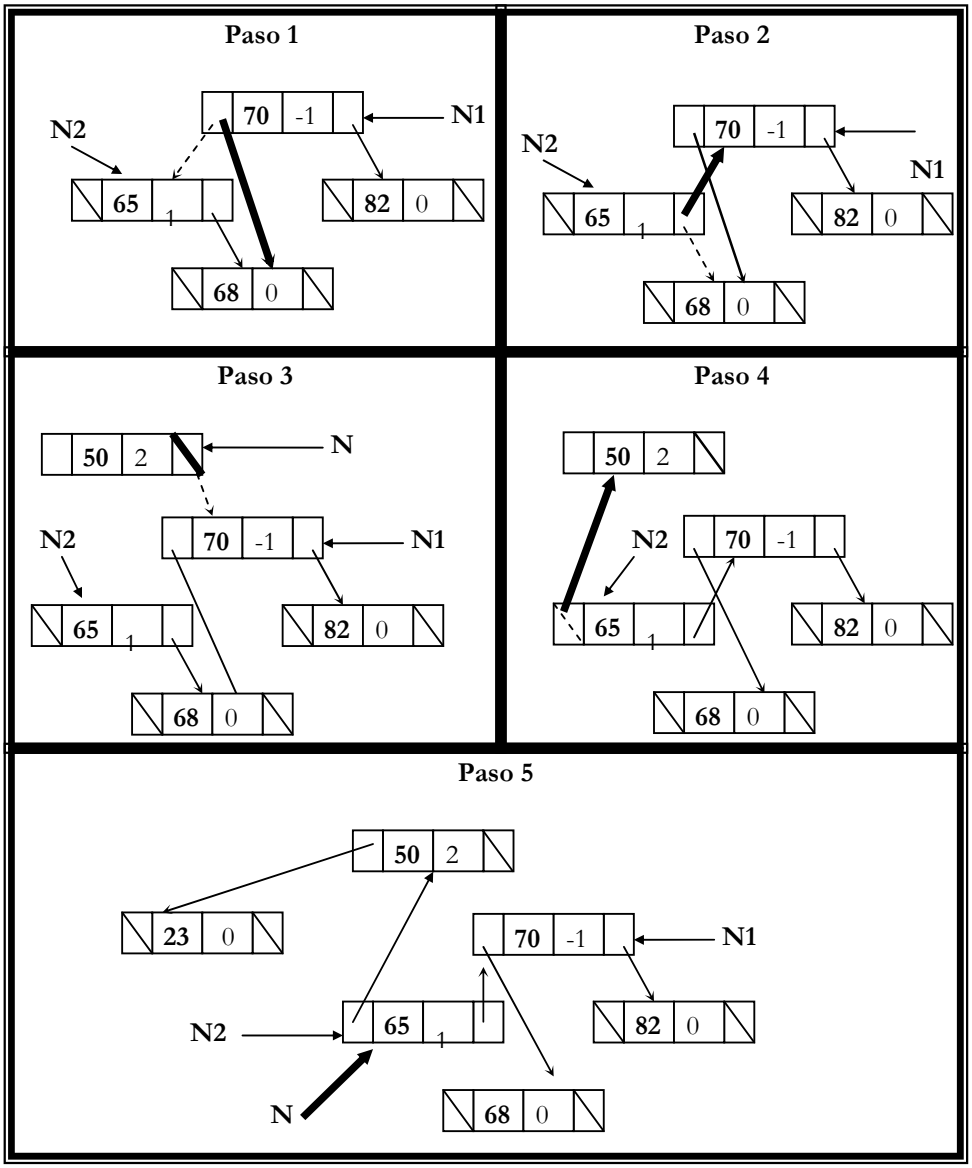


Figura 2.18. Seguimiento del algoritmo de rotación DI

El FE de los nodos involucrados se decide en base a la siguiente tabla:

N2^.FE=-1	N2^.FE=0	N2^.FE=1
N^.FE=0	N^.FE=0	N^.FE=-1
N1^.FE=1	N1^.FE=0	N1^.FE=0
N2^.FE=0	N2^.FE=0	N2^.FE=0

Como el ejemplo presentado el FE de N2 es igual a 1, se realiza las siguientes asignaciones:

```
N^.FE = -1
N1^.FE = 0
N2^.FE = 0
```

Luego de realizar la reestructuración, el árbol queda de la siguiente manera:

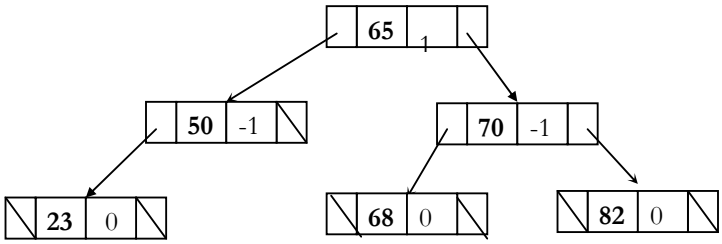


Figura 2.19. árbol reestructurado por DI

g) Inserción : Clave 39

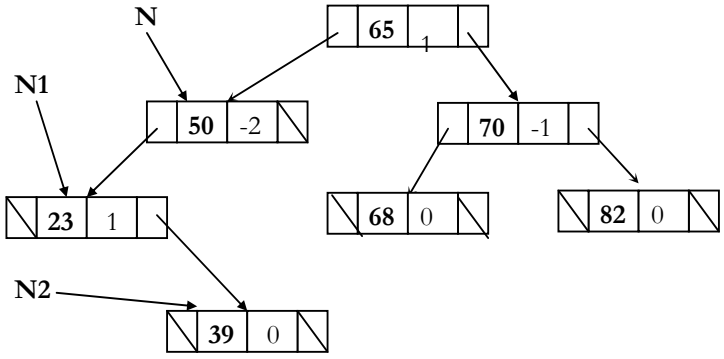
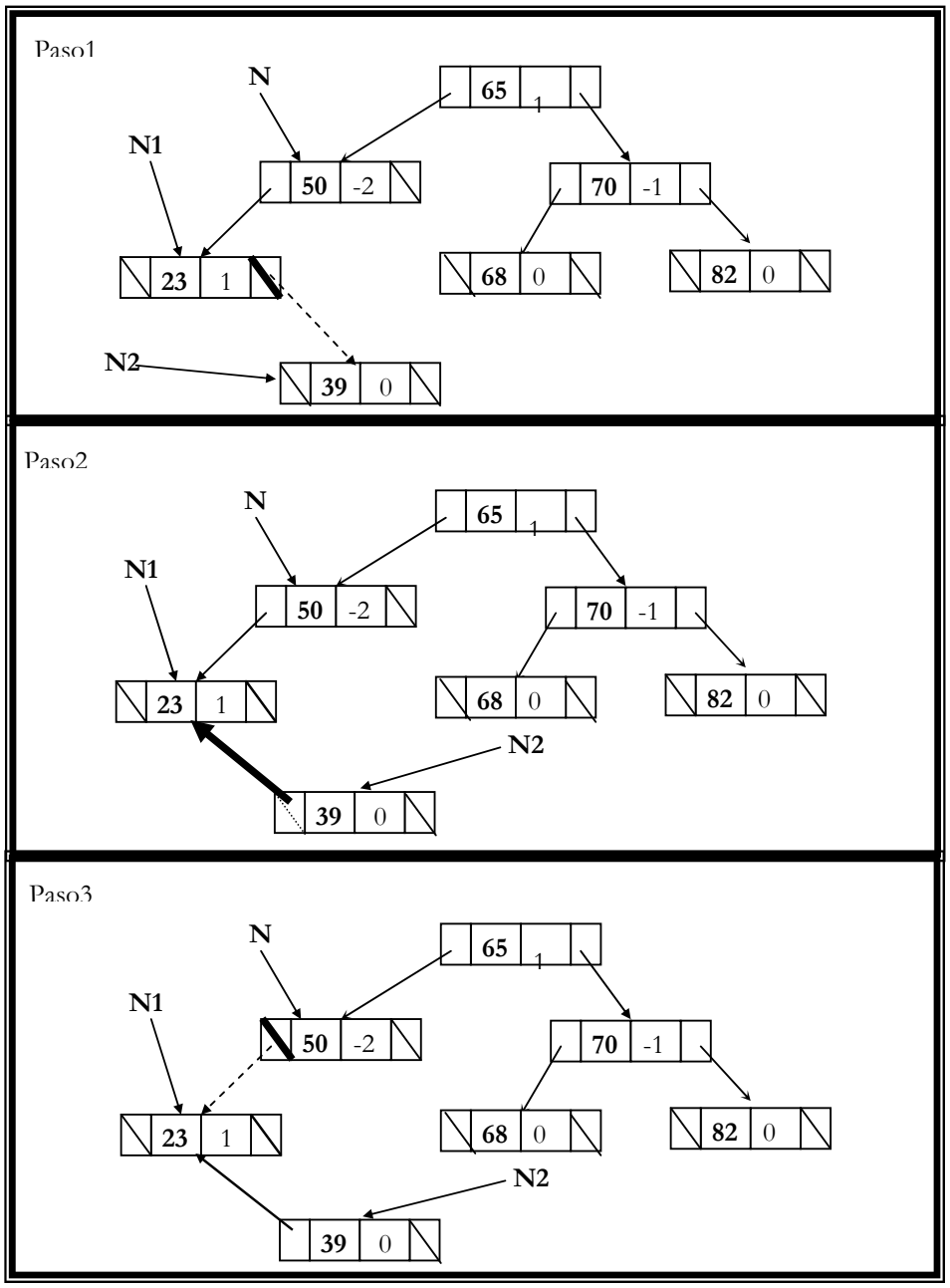


Figura 2.20. Continuación del seguimiento de la inserción de Árbol

El camino seguido para insertar el nodo con la clave 39 ha seguido el camino de izquierda 65, izquierda de 50, derecha de 23. al regresar por el camino de búsqueda, el factor de equilibrio del nodo 29 se incrementa en 1 por seguir el camino de la rama derecha, el del nodo 45 se decremento en 1 por seguir la rama izquierda y pasa a ser –2 se ha roto el criterio de equilibrio. La rotación de los nodos para reestablecer el equilibrio es ID.

```
1) N1^.Der = N2^.Izq
2) N2^.Izq = N1
3) N^.Izq =N2^.Der
4) N2^.Der = N
5) N = N2
```



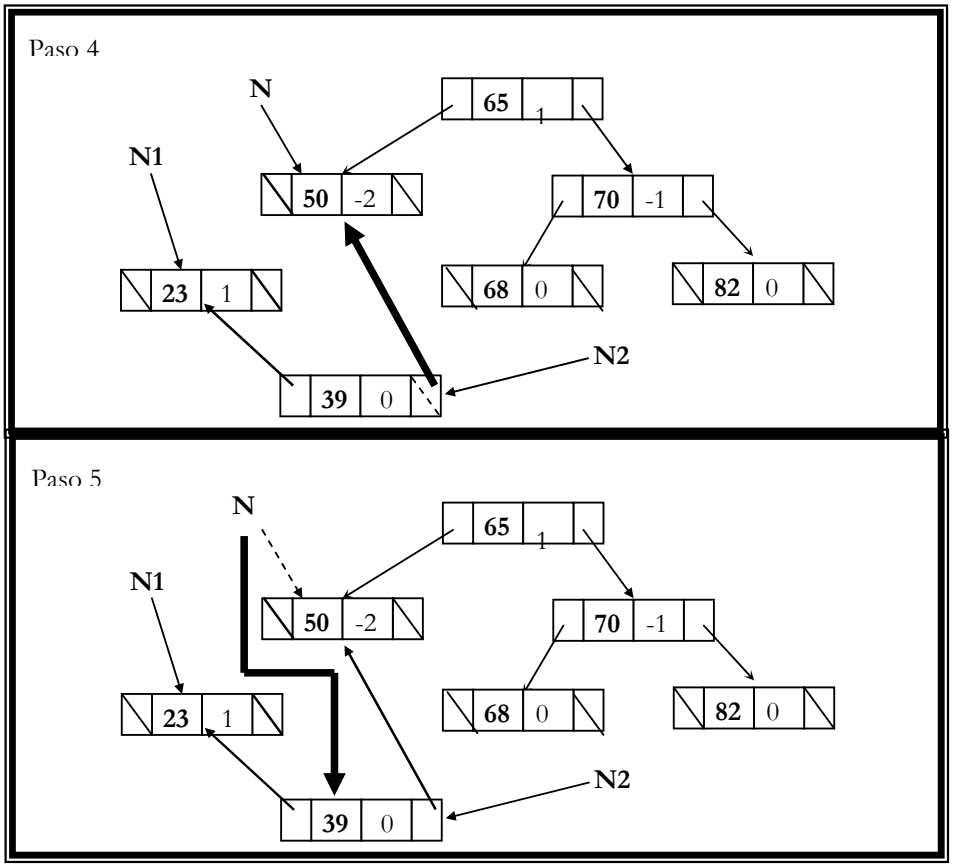


Figura 2.18. Seguimiento del algoritmo de rotación ID

El FE de los nodos involucrados se decide en base a la siguiente tabla:

N2^.FE=-1	N2^.FE=0	N^.FE=1
N^.FE=0	N^.FE=0	N^.FE=-1
N1^.FE=1	N1^.FE=0	N1^.FE=0
N2^.FE=0	N2^.FE=0	N2^.FE=0

Como en el ejemplo presentado el FE de N2 es igual a 0, se realizan las siguientes asignaciones:

```
N^.FE =  0
N1^.FE = 0
N2^.FE = 0
```

Luego volver a equilibrarlo, el árbol queda de la siguiente forma:

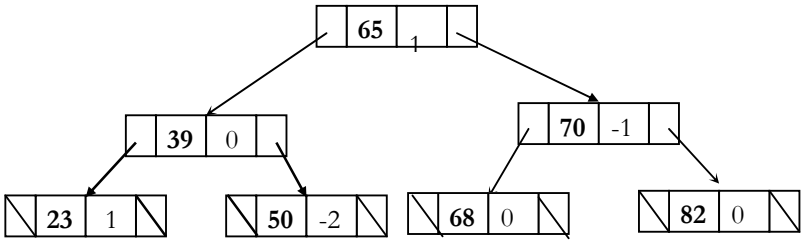


Figura 2.22. Árbol Reestructurado por ID

Procedimiento de Inserción por Balanceo

El algoritmo inserta un elemento en un árbol balanceado. N es una variable de tipo puntero (por referencia). BO es una variable de tipo booleano (por referencia). BO se utiliza para indicar que la altura del árbol ha crecido, su valor inicial es falso. Infor es una variable de tipo entero que contiene la información del elemento que queremos insertar.

```
public NodoArb insBal(NodoArb p, int d){
                        if(p!=null)
                                if(p.getDato()==d)
                                        System.out.println("ERROR: El dato que esta intentando
insertar ya existe");
                                else
                                        if(p.getDato()<d){
                                                p.setLI(insBal(p.getLD(),d));
                                                if(B){
                                                        switch(p.FE){
                                                                case -1:
p.FE=0;B=false;break;
                                                                case 0: p.FE=1;break;
                                                                case 1: n=p.getLD();

        if(n.FE==1){

        //Derecha-Derecha

        p.setLD(n.getLI());

        n.setLI(p);

        n.fe=0;

        p.fe=0;

        p=n;                                                                    }

        else{
```

```
//Derecha-Izquierda
                n1=n.getLI();
                n.setLI(n1.getLD());
                n1.setLD(n);
                p.setLD(n1.getLI());
                n1.setLI(p);
                switch(n1.FE){
                        case -1: p.FE=0;
                                        n.FE=1; break;
                        case 1: p.FE=-1;
                                        n.FE=0; break;
                        case 0: p.FE=0;
                                        n.FE=0;
                }
                n1.FE=0;
                p=n1;
                                                                                                }
                                                        }
                                                }
                                        }
                                        else
                                        //Hacer lo mismo, cambia los ld por li y los switch(cambiar los
casos del switch), derecha-derecha=>izquierda-izquierda; derecha-izquierda=>izquierda-derecha
                                                //Solo cambiar los cases los FE quedan igual y los
demas igual.
                                                if(p.getDato()>d){
                                                        p.setLD(insBal(p.getLI(),d));
                                                        *if(B){
                                                        *               switch(p.FE){
                                                        *                       case -1:
n=p.getLI();
                                                        *
                if(n.FE==1){
                                                        *
                        //Derecha-Derecha
                                                        *
                        p.setLD(n.getLI());
                                                        *
                        n.setLI(p);
                                                        *
                        n.fe=0;
                                                        *
                        p.fe=0;
                                                        *
                        p=n;
                                                        *
                }
                                                        *
                else{
                                                        *
                        //Derecha-Izquierda
                                                        *
                        n1=n.getLI();
                                                        *
                        n.setLI(n1.getLD());
                                                        *
                        n1.setLD(n);
                                                        *
                        p.setLD(n1.getLI());
                                                        *
                        n1.setLI(p);
```

Eliminación de un Nodo en un Árbol Balanceado

La operación de borrado en un árboles balanceados es un poco mas compleja que la operación de inserción. Consiste en quitar un nodo del árbol sin violar los principios que definen justamente un árbol balanceado. Recuérdese que se definió como una estructura en la cual, para todo nodo del árbol, se debe cumplir que "la altura del subárbol izquierdo y la altura del subárbol derecho no deben diferir en mas de una unidad".

La complejidad en la operación de borrado resulta ser cierta a pesar de que se utiliza el mismo algoritmo de borrado (idéntico en lógica pero diferente en implementación)que en los árboles binarios de búsqueda y la misma operación de reacomodo que se utiliza en el algoritmo de inserción en árboles balanceados.

Recuerde que la operación de borrado en árbol balanceado deben distinguirse los siguientes casos:

1. Si el elemento a borrar es terminal u hoja, simplemente se suprime.

2. Si el elemento a borrar tiene un solo descendiente entonces, tiene que sustituirse por ese descendiente.

3. Si el elemento a borrar tiene los dos descendientes, entonces tiene que sustituirse por el nodo que se encuentra mas a la izquierda en el subárbol derecho o por el nodo que se encuentra mas a la derecha del subárbol izquierdo.

Para eliminar un nodo en un árbol balanceado lo primero que debe hacerse es localizar su posición en el árbol. Se elimina siguiendo los criterios establecidos anteriormente y se regresa por el camino de búsqueda calculando FE de lso nodos visitados. Si en alguno de los nodos se viola el criterio de equilibrio, entonces debe de reestructurase el árbol. El proceso termina cuando se llega hasta la raíz del árbol. Cabe aclarar que mientras en el algoritmo de inserción una vez que era efectuada una rotación podía detenerse el proceso, en este algoritmo debe continuarse puesto que se puede producir mas de una rotación en el camino hacia atrás.

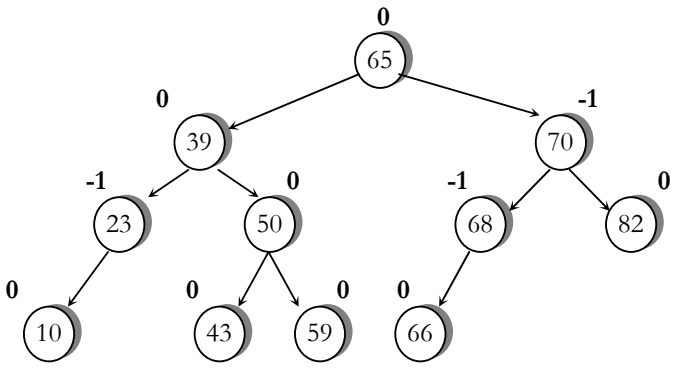
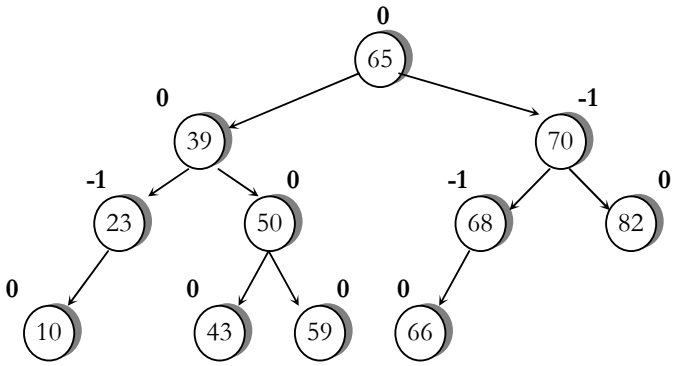
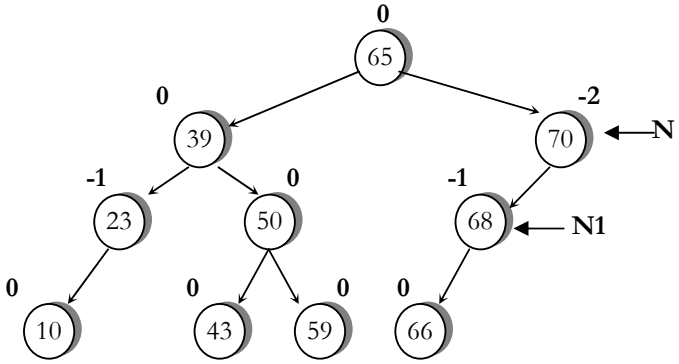


Figura 2.23. Árbol Balanceado.

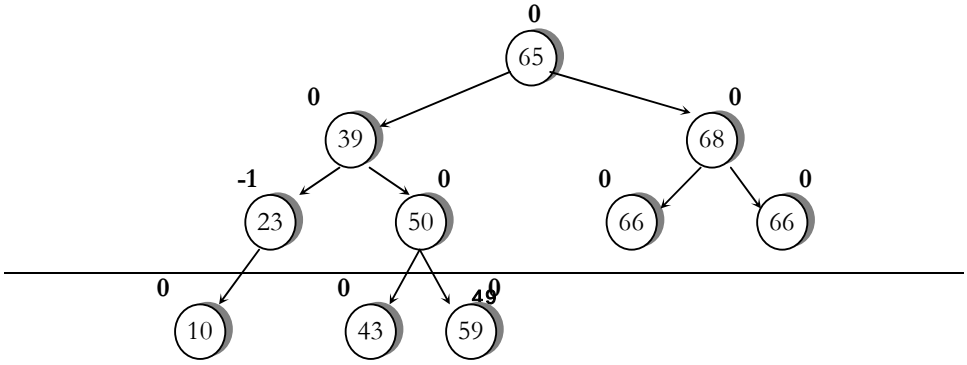
En el árbol de la figura 2.23 se va eliminando el nodo con clave 82: al ser un nodo hoja es borrado es simple, se suprime el nodo (Figura 2.24 a). Al volver por el camino de búsqueda para determinar FE, resulta que el FE del nodo 70 pasaría a ser -2 ya que ha decrementazo la altura de la rama derecha, es violado el criterio de equilibrio (Figura 2.24 b). Sea punta con N la clave 70 y con N1 la rama izquierda de N. Se verifica el FE de N1 y como es igual a –1, entonces se realiza la rotación II. Luego de la reestructuración de el árbol en la raíz 68(Figura 2.24 c).



a)



b)



c)

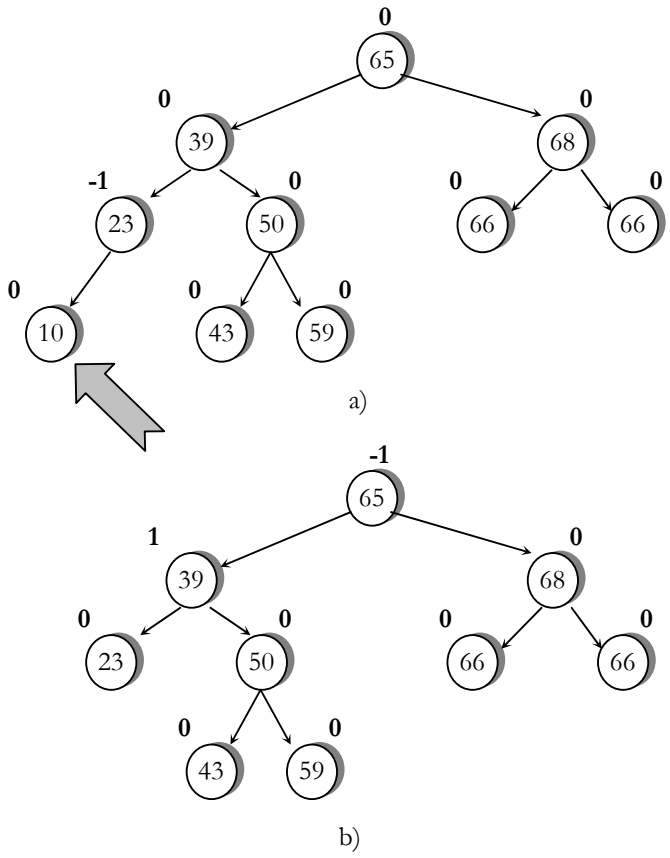
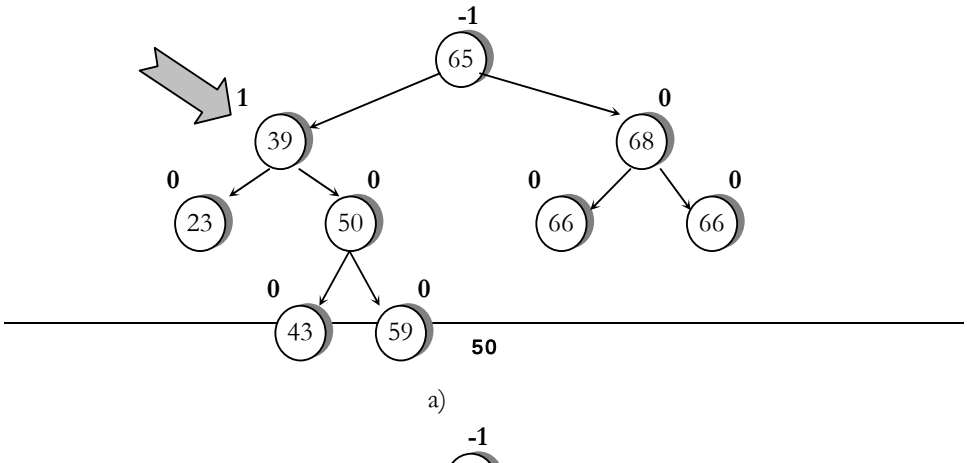
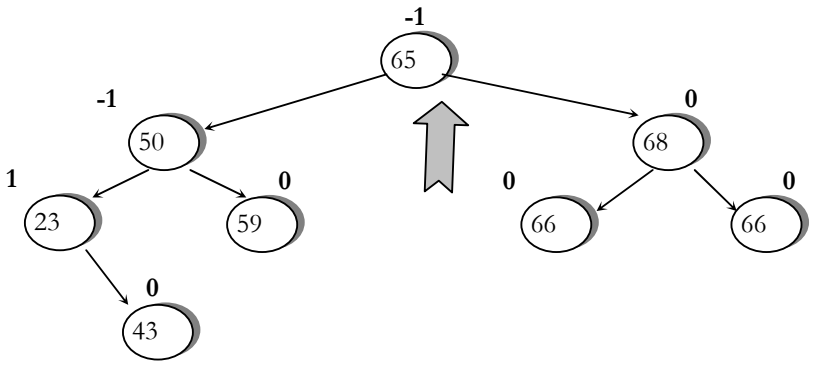


Figura 2.25. Eliminación de un nodo hoja

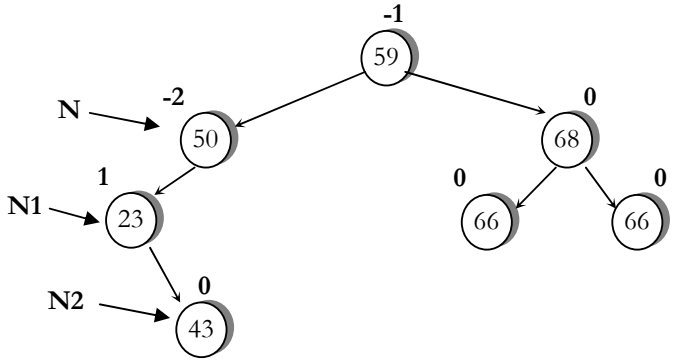
La eliminación de la clave 10 es un proceso sencillo (figura 2.25 a) no debe reestructurar el árbol y solo es necesario cambiar el FE de los nodos 23 y 39 (figura 2.25 b).



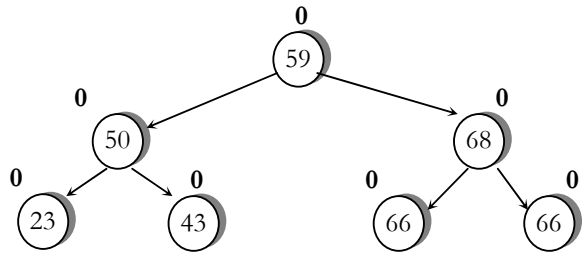
Al eliminar la clave 39 se origina el caso mas difícil de borrado en árboles: la eliminación de una clave con dos descendientes (fig. 2.26 a) se opta por sustituir dicha clave por el nodo que se encuentra mas a la derecha en el subárbol izquierda (23). Luego de la sustitución se observa que en dicha clave se viola el criterio de equilibrio y debe reestructurarse el árbol. Se apunta el con N a la clave 23 y con N1 la rama derecha de N. Se verifica el FE de N1 y como este es igual a 0, se realiza la rotación DD. Luego del reacomodo el árbol queda como en la figura 2.26 c.



a)



b)



c)

Figura 2.27. Eliminación de un nodo con el método ID

Al eliminar la clave 65 surge nuevamente el tercer caso de borrado , que corresponde a una clave con dos descendientes (figura 2.27a). Se sustituye dicha clave por el nodo

que se encuentra mas a la derecha en el subárbol izquierdo (59). Es evidente que depuse de la sustitución, en la clave 50 se viola el criterio de equilibrio y que debe reestructurarse el árbol. Se apunta con N a la clave 50 y N1 la rama izquierda N y se verifica el FE. Como en este caso es igual a 1, se apunta N2 la rama derecha de N1 y se realiza la rotación ID, posteriormente se reestructura (Fig. 2.27 c).

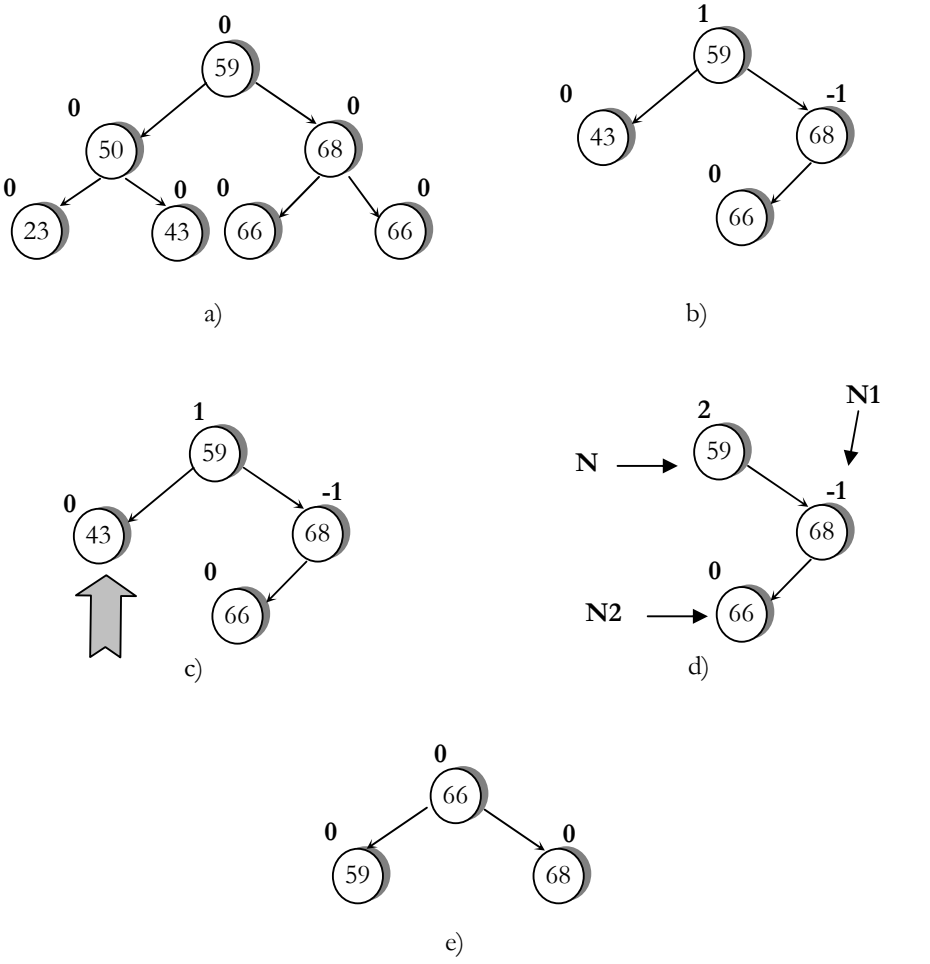


Figura 2.28. Eliminación de un nodo con el método DI

Luego eliminación de las claves 70, 23 y 50, el árbol queda como en la figura 2.28 b. La eliminación de la clave 43 corresponde al primer caso de borrado en árboles, es el caso mas simple (figura 2.28 c). Sin embargo, al verificar el FE de la clave 59 se advierte que se rompe el equilibrio del árbol y debe reestructurarse (figura 2.28 d). Se apunta con N la clave 59 y con N1 la rama derecha de N, y se verifica el FE de N1. Como es igual a –1, se apunta con N2 la rama izquierda de N1 y se realiza la rotación DI. Luego de la reestructuración, el árbol queda como en la figura 2.28 e.

Con el fin de darle mayor modularidad al algoritmo de eliminación en árboles balanceados, se estudiaran tres algoritmos auxiliares. El primero **reestructura1,** se utiliza cuando la altura de la rama izquierda ha disminuido. El segundo, **reestructura2** se emplea cuando la altura de la rama derecha ha disminuido. El ultimo **borrado** se utiliza en el caso mas difícil de borrado en árboles , con el objeto de sustituir el elemento que desea eliminarse por el que se encuentra mas a la derecha en el subárbol izquierdo.

A continuación se muestra el algoritmo de eliminación en árboles balanceados.

Ejercicios

2.1 Dada la secuencia de claves enteras:100, 29, 71, 82, 48, 39, 101, 22, 46, 17, 3, 20, 25, 10.Representar gráficamente el árbol AVL correspondiente. Elimine claves consecutivamente hasta encontrar un desequilibrio y dibuje la estructura del árbol tras efectuarse la oportuna restauración.

2.2 Obtener la secuencia de rotaciones resultante de la inserción del conjunto de elementos {1,2,3,4,5,6,7,15,14,13,12,11,10,9,8} en un árbol AVL.

2.3 Inserte las claves en el orden indicado a fin de incorporarlas a un árbol AVL.

a. 10,100,20,80,40,70.
b. 5,10,20,30,40,50,60.

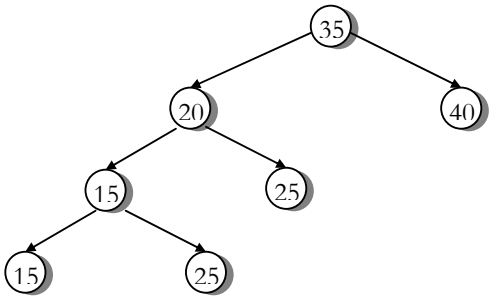
Problemas

2.4 Escribir un algoritmo de inserción eficiente para un árbol de búsqueda binaria con el fin de incluir un registro nuevo cuya llave se sabe que no existe en el árbol.
2.5 Muestre que es posible un arbol de búsqueda binaria en el cual solo existe una hoja unica incluso si los elementos del árbol no están insertados en orden estrictamente ascendente o descendente.
2.6 verificar por simulación que si están no presentes registros para el algo timo de búsqueda e inserción en el arbol binario por orden aleatorio, la cantidad de comparaciones de llaves es O(Log n).
2.7 Muestre que cada arbol de búsqueda binaria de n nodos no tiene la misma probabilidad (suponiendo que los elementos se insertan en orden aleatorio) y que los árboles AVL son más probables que los árboles en línea recta.
2.8 Escriba un algoritmo para suprimir un nodo de un arbol binario que sustituye el nodo con su antecesor de orden inmediato y no con su sucesor de orden inmediato.
2.9 Escribir un algoritmo que busque y suprima un registro con la llave a partir de un árbol de búsqueda binaria.

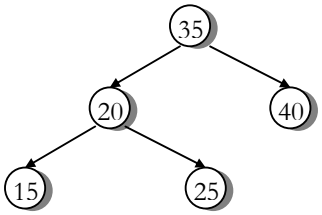
K	R	LEFT	RIGHT

2.10 Escribir un algoritmo que elimine todos los registros entre un rango de llaves de un arbol de búsqueda binaria cuyo no es similar al registro anterior

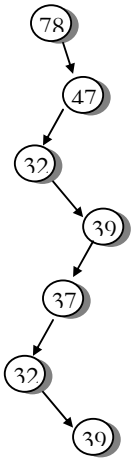
2.11 considere los siguientes árboles de búsqueda



a)



b)



c)

a) Cuantas permutaciones de los enteros del 1 al 7 producirán los de búsqueda binaria.
b) Cuantas permutaciones de los enteros del 1 al 7 producirán los de búsqueda binaria similares
c) Cuantas permutaciones de los enteros del 1 al 7 producirán los de búsqueda binaria con la misma cantidad de nodos en cada nivel iguales a los arboles de las figuras.
d) Encontrar una asignación de probabilidades para los primeros 7 enteros positivos como algoritmos de búsqueda que hacen optimo cada uno de los árboles.

2.12 Muestra que el árbol Fibonacci de orden h+1, es un árbol de altura balanceada h y que tiene menos nodos que cualquier otro árbol de altura balanceada h.

Capítulo

6

Métodos de Ordenación y Busqueda

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

La colocación en orden de una lista de valores se llama Ordenación. Por ejemplo, se podría disponer una lista de valores numéricos en orden ascendente o descendente, o bien una lista de nombres en orden alfabético. La localización de un elemento de una lista se llama búsqueda.

Tal operación se puede hacer de manera más eficiente después de que la lista ha sido ordenada.

Existen varios métodos para ordenamiento, clasificados en tres formas:

- Intercambio
- Selección
- Inserción.

En cada familia se distinguen dos versiones: un método simple y directo, fácil de comprender pero de escasa eficiencia respecto al tiempo de ejecución, y un método rápido, más sofisticado en su ejecución por la complejidad de las operaciones a realizar, pero mucho más eficiente en cuanto a tiempo de ejecución. En general, para arreglos con pocos elementos, los métodos directos son más eficientes (menor tiempo de ejecución) mientras que para grandes cantidades de datos se deben emplear los llamados métodos rápidos.

Intercambio

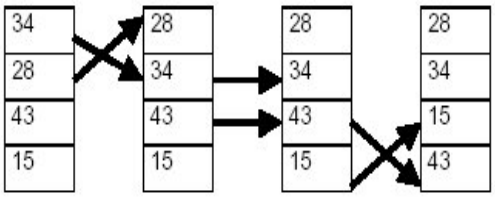
El método de intercambio se basa en comparar los elementos del arreglo e intercambiarlos si su posición actual o inicial es contraria inversa a la deseada. Pertenece a este método el de la burbuja clasificado como intercambio directo. Aunque no es muy eficiente para ordenar listas grandes, es fácil de entender y muy adecuado para ordenar una pequeña lista de unos 100 elementos o menos. Una pasada por la ordenación de burbujeo consiste en un recorrido completo a través del arreglo, en el que se comparan los contenidos de las casillas adyacentes, y se cambian si no están en orden. La ordenación por burbujeo completa consiste en una serie de pasadas ("burbujeo") que termina con una en la que ya no se hacen cambios porque todo está en orden.

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

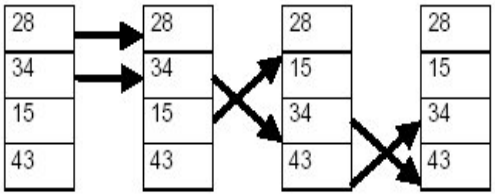
Comparar el contenido de x[2] con el de x[3]; e intercambiarlos si fuera necesario.

Comparar el contenido de x[3] con el de x[4]; e intercambiarlos si fuera necesario.

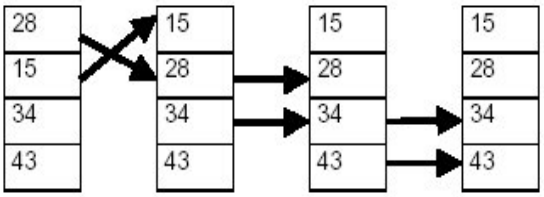
Al final de la primera pasada, el mayor de los números estará en x[4].



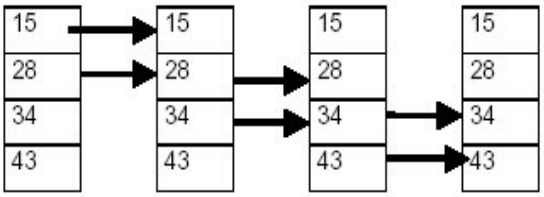
Al final de la segunda pasada, los últimos dos elementos estarán ordenados como se muestra:



Al final de la tercera pasada, los últimos tres elementos estarán ordenados como se muestra:



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



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

Quicksort.

Si bien el método de la burbuja era considerado como el peor método de ordenación simple o menos eficiente, el método Quicksort basa su estrategia en la idea intuitiva de que es más fácil ordenar una gran estructura de datos subdividiéndolas en otras más pequeñas introduciendo un orden relativo entre ellas. En otras palabras, si dividimos el array a ordenar en dos subarrays de forma que los elementos del subarray inferior sean más pequeños que los del subarray superior, y aplicamos el método reiteradamente, al final tendremos el array inicial totalmente ordenado. Existen además otros métodos conocidos, el de ordenación por montículo y el de shell.

Shell

Fue nombrado Ordenamiento de disminución incremental debido a su inventor Donald Shell.

Ordena subgrupos de elementos separados K unidades (respecto de su posición en el arreglo) del arreglo original. El valor K es llamado incremento.

Después de que los primeros K subgrupos han sido ordenados (generalmente utilizando INSERCION DIRECTA), se escoge un nuevo valor de K más pequeño, y el arreglo es de nuevo partido entre el nuevo conjunto de subgrupos. Cada uno de los

subgrupos mayores es ordenado y el proceso se repite de nuevo con un valor más pequeño de K.

Eventualmente el valor de K llega a ser 1, de tal manera que el subgrupo consiste de todo el arreglo ya casi ordenado.

Al principio del proceso se escoge la secuencia de decrecimiento de incrementos; el último valor debe ser 1.

"Es como hacer un ordenamiento de burbuja pero comparando e intercambiando elementos."

Cuando el incremento toma un valor de 1, todos los elementos pasan a formar parte del subgrupo y se aplica inserción directa.

El método se basa en tomar como salto N/2 (siendo N el número de elementos) y luego se va reduciendo a la mitad en cada repetición hasta que el salto o distancia vale 1.

Ejemplo:

Para el arreglo a = [6, 1, 5, 2, 3, 4, 0]

Tenemos el siguiente recorrido:

Recorrido	Salto	Lista Ordenada	Intercambio
1	3	2,1,4,0,3,5,6	(6,2),(5,4),(6,0)
2	3	0,1,4,2,3,5,6	(2,0)
3	3	0,1,4,2,3,5,6	Ninguno
4	1	0,1,2,3,4,5,6	(4,2),(4,3)
5	1	0,1,2,3,4,5,6	Ninguno

```
Public  void Ordenar(int[] Arreglo,int n)
{
        int salto=0,sw=0,i=0,aux=0;
        System.out.println("\nEl vector ingresado es: \n");
        for(i=0;i<n;i++){
            System.out.println (" ["+Arreglo[i]+"] ");
                        }
        salto=n/2;
        while(salto>0){
                sw=1;
                while(sw!=0){
                        sw=0 ;
                        i=1;
                        while(i<=(n-salto)){
                                if (Arreglo[i-1]>Arreglo[(i-1)+salto]){
                                        aux=Arreglo[(i-1)+salto];
                                        Arreglo[(i-1)+salto]=Arreglo[i-1];
                                        Arreglo[(i-1)]=aux;
                                        sw=1;
                                }
                        i++;
                }
        }
        salto=salto/2;
        }
        System.out.println ("\nEl vector ordenado es: \n");
        for(i=0;i<n;i++){
                System.out.println (" ["+Arreglo[i]+"] ");
        }
        System.out.println ();
        return;
        }
}
```

Selección.

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

Burbuja

Este es el algoritmo más sencillo probablemente. Ideal para empezar. Consiste en ciclar repetidamente a través de la lista, comparando elementos adyacentes de dos en dos. Si un elemento es mayor que el que está en la siguiente posición se intercambian. ¿Sencillo no?

Vamos a ver un ejemplo. Esta es nuestra lista:

1. 4 - 3 - 5 - 2 - 1

Tenemos 5 elementos. Es decir, TAM toma el valor 5. Comenzamos comparando el primero con el segundo elemento. 4 es mayor que 3, así que intercambiamos. Ahora tenemos:

2. **3** - **4** - 5 - 2 - 1

Ahora comparamos el segundo con el tercero: 4 es menor que 5, así que no hacemos nada. Continuamos con el tercero y el cuarto: 5 es mayor que 2. Intercambiamos y obtenemos:

3. 3 - 4 - **2** - **5** - 1

Comparamos el cuarto y el quinto: 5 es mayor que 1. Intercambiamos nuevamente:

4. 3 - 4 - 2 - **1** - **5**

Repitiendo este proceso vamos obteniendo los siguientes resultados:

5. 3 - 2 - 1 - 4 - 5
6. 2 - 1 - 3 - 4 - 5
7. 1 - 2 - 3 - 4 - 5

```
public class Burbuja
{
        public static void main (String[] args)
        {
                int hasta = 0 ;
                int [] arreglo ;
                int aux = 0;
                System.out.print("Introduzca el numero de
datos a cargar =>");
                hasta = Leer.datoInt();
                arreglo = new int[hasta];

                for (int i=0;i<=hasta-1;i++)
                {
                        arreglo[i] = Leer.datoInt();

                }


                System.out.println("Los datos ordenados
son:");


                for (int i2=0;i2<=hasta-2;i2++)

                {
```

```
                for (int i=0 ;i<=hasta-2;i++)
                        {

                        if (arreglo[i]>arreglo[i+1])
                        {
                                aux=arreglo[i+1];
                                arreglo[i+1]=arreglo[i];
                                arreglo[i]=aux;
                        }

                        }
                }

                for (int i=0 ;i<=hasta-1;i++)
                {
                        System.out.println(arreglo[i]);
                }



        }
}
```

Inserción.

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

Búsqueda

La búsqueda es una operación que tiene por objeto la localización de un elemento dentro de la estructura de datos. A menudo un programador estará trabajando con grandes cantidades de datos almacenados en arreglos y pudiera resultar necesario determinar si un arreglo contiene un valor que coincide con algún valor clave o buscado.

Siendo el array de una dimensión o lista una estructura de acceso directo y a su vez de acceso secuencial, encontramos dos técnicas que utilizan estos dos métodos de acceso, para encontrar elementos dentro de un array: búsqueda lineal y búsqueda binaria.

Búsqueda Secuencial:

La búsqueda secuencial es la técnica más simple para buscar un elemento en un arreglo. Consiste en recorrer el arreglo elemento a elemento e ir comparando con el valor buscado (clave). Se empieza con la primera casilla del arreglo y se observa una casilla tras otra hasta que se encuentra el elemento buscado o se han visto todas las casillas. El resultado de la búsqueda es un solo valor, y será la posición del elemento buscado o cero. Dado que el arreglo no está en ningún orden en particular, existe la misma probabilidad de que el valor se encuentra ya sea en el primer elemento, como

en el último. Por lo tanto, en promedio, el programa tendrá que comparar el valor buscado con la mitad de los elementos del arreglo.
El método de búsqueda lineal funciona bien con arreglos pequeños o para arreglos no ordenados. Si el arreglo está ordenado, se puede utilizar la técnica de alta velocidad de búsqueda binaria, donde se reduce sucesivamente la operación eliminando repetidas veces la mitad de la lista restante.

```
Public boolean Busqueda(int Elem)
{
      if(N!=0)
      {
            for(int i=0;i<N;i++)
            if(Lista[i]==Elem)
            {
                  System.out.println("El "+Elem+" esta
                  en la Lista");
                  Return true;
            }
            System.out.println ("El "+Elem+" no esta en
            la Lista");
            Return false;
      }
      System.out.println ("lista Vacia...");
      Return false;
}
```

Búsqueda Binaria.

La búsqueda binaria es el método más eficiente para encontrar elementos en un arreglo ordenado. El proceso comienza comparando el elemento central del arreglo con el valor buscado. Si ambos coinciden finaliza la búsqueda. Si no ocurre así, el elemento buscado será mayor o menor en sentido estricto que el central del arreglo. Si el elemento buscado es mayor se procede a hacer búsqueda binaria en el subarray superior, si el elemento buscado es menor que el contenido de la casilla central, se debe cambiar el segmento a considerar al segmento que está a la izquierda de tal sitio central

```
class Busqbin {

   public static void main (String [] args) {
      int [] x = { -5, 12, 15, 20, 30, 72, 456 };
      int loIndex = 0;
      int hiIndex = x.length - 1;
      int midIndex, srch = 72;

      while (loIndex <= hiIndex)  {
         midIndex = (loIndex + hiIndex) / 2;
         if (srch > x [midIndex])
             loIndex = midIndex + 1;
         else if (srch < x [midIndex])
             hiIndex = midIndex - 1;
         else
             break;
      }
      if (loIndex > hiIndex)
          System.out.println (srch + " not found");
      else
          System.out.println (srch + " found");
   }
}
```