ESTRUCTURA DE DATOS

“APUNTES”

INTRODUCCION

Como ya sabemos, las computadoras fueron diseñadas o ideadas como una herramienta mediante la cual podemos realizar operaciones de cálculo complicadas en un lapso de mínimo tiempo. Pero la mayoría de las aplicaciones de este fantástico invento del hombre, son las de almacenamiento y acceso de grandes cantidades de información.

La información que se procesa en la computadora es un conjunto de datos, que pueden ser simples o estructurados. Los datos simples son aquellos que ocupan sólo un localidad de memoria, mientras que los estructurados son un conjunto de casillas de memoria a las cuales hacemos referencia mediante un identificador único.

Debido a que por lo general tenemos que tratar con conjuntos de datos y no con datos simples (enteros, reales, booleanos, etc.) que por sí solos no nos dicen nada, ni nos sirven de mucho, es necesario tratar con estructuras de datos adecuadas a cada necesidad.

Las estructuras de datos son una colección de datos cuya organización se caracteriza por las funciones de acceso que se usan para almacenar y acceder a elementos individuales de datos.

Una estructura de datos se caracteriza por lo siguiente:

-Pueden descomponerse en los elementos que la forman.

-La manera en que se colocan los elementos dentro de la estructura afectará la forma en que se realicen los accesos a cada elemento.

-La colocación de los elementos y la manera en que se accede a ellos puede ser encapsulada

UNIDAD I. TIPOS DE DATOS

TIPOS DE DATOS

Definición: El tipo de un dato es el conjunto de valores que puede tomar durante el programa. Si se le intenta dar un valor fuera del conjunto se producirá un **error**.

La asignación de tipos a los datos tiene **dos objetivos** principales:

- Por un lado, detectar errores en las operaciones
- Por el otro, determinar cómo ejecutar estas operaciones

Un lenguaje fuertemente tipeado es aquel en el que todos los datos deben de tener un tipo declarado explícitamente, y además que existen ciertas restricciones en las expresiones en cuanto a los tipos de datos que en ellas intervienen. Una ventaja de los lenguajes *fuertemente tipeados* es que se gasta mucho menos esfuerzo en depurar (corregir) los programas gracias a la gran cantidad de errores que detecta el compilador.

Clasificaciones en los tipos de datos

Existen muchas clasificaciones para los tipos de datos. Una de estas es la siguiente:

- Dinámicos
- Estáticos
    - El tipo cadena
    - Estructurados
    - Simples
        - Ordinales
        - No-ordinales

Tipos estáticos

Casi todos los tipos de datos son estáticos, la excepción son los punteros. Que un tipo de datos sea estático quiere decir que el tamaño que ocupa en memoria no puede variar durante la ejecución del programa. Es decir, una vez declarada una variable de un tipo determinado, a ésta se le asigna un trozo de memoria fijo, y este trozo no se podrá aumentar ni disminuír.

Tipos dinámicos.

Dentro de esta categoría entra sólamente el tipo puntero. Este tipo te permite tener un mayor control sobre la gestión de memoria en tus programas. Con

ellos puedes manejar el tamaño de tus variables en tiempo de ejecución, o sea, cuando el programa se está ejecutando. Los punteros quizás sean el concepto más complejo a la hora de aprender un lenguaje de programación.

Tipos simples

Como su nombre indica son los tipos básicos. Son los más sencillos y los más fáciles de aprender. Los tipos simples más básicos son: entero, lógico, carácter y real. Y la mayoría de los lenguajes de programación los soportan, no como ocurre con los estructurados que pueden variar de un lenguaje a otro.

Tipos estructurados

Mientras que una variable de un tipo simple sólo referencia a un elemento, los estructurados se refieren a colecciones de elementos.
Las colecciones de elementos que aparecen al hablar de tipos estructurados son muy variadas: tenemos colecciones **ordenadas** que se representan mediante el tipo array, colecciones **sin orden** mediante el tipo conjunto, e incluso colecciones que **contienen otros tipos**, son los llamados registros.

Tipos ordinales

Dentro de los tipos simples, los ordinales son los más abundantes. De un tipo se dice que es ordinal porque el conjunto de valores que representa se puede *contar*, es decir, podemos establecer una relación uno a uno entre sus elementos y el conjunto de los números naturales.
Dentro de los tipos simples ordinales, los más importantes son:

- El tipo entero.
- El tipo lógico.
- El tipo carácter.

Tipos no-ordinales

Simplificando, podríamos reducir los tipos simples no-ordinales al tipo real. Este tipo nos sirve para declarar variables que pueden tomar valores dentro del conjunto de los números reales. A diferencia de los tipos ordinales, los no-ordinales no se pueden contar. No se puede establecer una relación uno a uno entre ellos y los número naturales. Dicho de otra forma, para que un conjunto se considere ordinal se tiene que poder calcular la posición, el anterior elemento y el siguiente de un elemento cualquiera del conjunto.**¿Cuál es el sucesor de 5.12?** Será 5.13, o 5.120, o 5.121, ...

UNIDAD II. ESTRUCTURAS SECUENCIALES

INTRODUCCION

Supongamos que nos enfrentamos a un problema como este: Una empresa que cuenta con 150 empleados, desea establecer una estadística sobre los salarios de sus empleados, y quiere saber cual es el salario promedio, y también cuantos de sus empleados gana entre $1250.00 y $2500.00.
Si tomamos la decisión de tratar este tipo de problemas con datos simples, pronto nos percataríamos del enorme desperdicio de tiempo, almacenamiento y velocidad. Es por eso que para situaciones de este tipo la mejor solución son los datos estructurados.
Un arreglo puede definirse como un grupo o una colección finita, homogénea y ordenada de elementos. Los arreglos pueden ser de los siguientes tipos:

- De una dimensión.
- De dos dimensiones.
- De tres o más dimensiones.

Arreglos Unidimensionales

Un arreglo unidimensional es un tipo de datos estructurado que está formado de una colección finita y ordenada de datos del mismo tipo. Es la estructura natural para modelar listas de elementos iguales.
El tipo de acceso a los arreglos unidimensionales es el acceso directo, es decir, podemos acceder a cualquier elemento del arreglo sin tener que consultar a elementos anteriores o posteriores, esto mediante el uso de un índice para cada elemento del arreglo que nos da su posición relativa.
Para implementar arreglos unidimensionales se debe reservar espacio en memoria, y se debe proporcionar la dirección base del arreglo, la cota superior y la inferior.

REPRESENTACION EN MEMORIA

Los arreglos se representan en memoria de la forma siguiente:

```
x : array[1..5] of integer
```

Para establecer el rango del arreglo (número total de elementos) que componen el arreglo se utiliza la siguiente formula:

RANGO = Ls - (Li+1)

donde:

ls = Límite superior del arreglo

li = Límite inferior del arreglo

Para calcular la dirección de memoria de un elemento dentro de un arreglo se usa la siguiente formula:

A[i] = base(A) + [(i-li) * w]

donde :

A = Identificador único del arreglo

i = Indice del elemento

li = Límite inferior

w = Número de bytes tipo componente

Si el arreglo en el cual estamos trabajando tiene un índice numerativo utilizaremos las siguientes fórmulas:

RANGO = ord (ls) - (ord (li)+1)

A[i] = base (A) + [ord (i) - ord (li) * w]

Arreglos Bidimensionales

Este tipo de arreglos al igual que los anteriores es un tipo de dato estructurado, finito ordenado y homogéneo. El acceso a ellos también es en forma directa por medio de un par de índices.

Los arreglos bidimensionales se usan para representar datos que pueden verse como una tabla con filas y columnas. La primera dimensión del arreglo representa las columnas, cada elemento contiene un valor y cada dimensión representa una relación

La representación en memoria se realiza de dos formas : almacenamiento por columnas o por renglones.

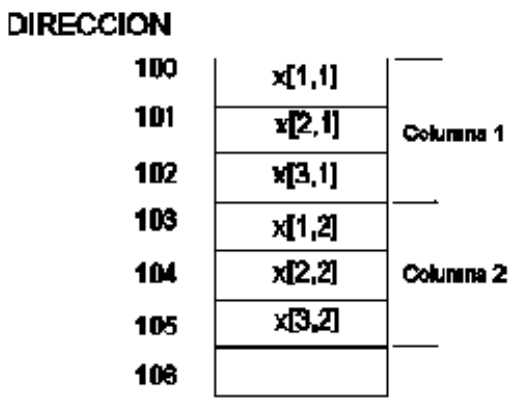
Para determinar el número total de elementos en un arreglo bidimensional usaremos las siguientes fórmulas:

RANGO DE RENGLONES| (R1) = Ls1 - (Li1+1)

RANGO DE COLUMNAS (R2) = Ls2 - (Li2+1)

No. TOTAL DE COMPONENTES = R1 * R2

REPRESENTACION EN MEMORIA POR COLUMNAS

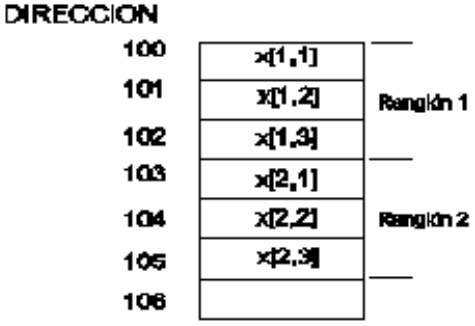


x : array [1..5,1..7] of integer

Para calcular la dirección de memoria de un elemento se usan la siguiente formula:

A[i,j] = base (A) + [((j - li2) R1 + (i + li1))*w]

REPRESENTACION EN MEMORIA POR RENGLONES



x : array [1..5,1..7] of integer

Para calcular la dirección de memoria de un elemento se usan la siguiente formula:

A[i,j] = base (A) + [((i - li1) R2 + (j + li2))*w]

donde:

i = Indice del renglón a calcular

j = Indice de la columna a calcular

li1 = Límite inferior de renglones

li2 = Límite inferior de columnas

w = Número de bytes tipo componente

Arreglos Multidimensionales

Este también es un tipo de dato estructurado, que está compuesto por n dimensiones. Para hacer referencia a cada componente del arreglo es necesario utilizar n índices, uno para cada dimensión

Para determinar el número de elementos en este tipo de arreglos se usan las siguientes fórmulas:

RANGO (Ri) = lsi - (lii + 1)

No. TOTAL DE ELEMENTOS = R1 * R2* R3 * ...* Rn

donde:

i = 1 ... n

n = No. total de dimensiones

Para determinar la dirección de memoria se usa la siguiente formula:

LOC A[i1,i2,i3,...,in] = base(A) + [(i1-li1)*R3*R4*Rn + (i2-li2)*R3*R2*... (in - lin)*Rn]*w

Operaciones Con Arreglos

Las operaciones en arreglos pueden clasificarse de la siguiente forma:

- Lectura
- Escritura
- Asignación
- Actualización
- Ordenación
- Búsqueda

a) LECTURA

Este proceso consiste en leer un dato de un arreglo y asignar un valor a cada uno de sus componentes.
La lectura se realiza de la siguiente manera:

```
para i desde 1 hasta N haz
    x<--arreglo[i]
```

b) ESCRITURA

Consiste en asignarle un valor a cada elemento del arreglo.
La escritura se realiza de la siguiente manera:

```
para i desde 1 hasta N haz
        arreglo[i]<--x
```

c) ASIGNACION

No es posible asignar directamente un valor a todo el arreglo, por lo que se realiza de la manera siguiente:

```
para i desde 1 hasta N haz
        arreglo[i]<--algún_valor
```

d) ACTUALIZACION

Dentro de esta operación se encuentran las operaciones de eliminar, insertar y modificar datos. Para realizar este tipo de operaciones se debe tomar en cuenta si el arreglo está o no ordenado.

Para arreglos ordenados los algoritmos de inserción, borrado y modificación son los siguientes:

1.- Insertar.

Si i< mensaje(arreglo contrario caso En arreglo[i]<--valor i<--i+1 entonces>

2.- Borrar.

```
Si N>=1 entonces
inicio
    i<--1
    encontrado<--falso
    mientras i<=n y encontrado=falso
            inicio
                si arreglo[i]=valor_a_borrar entonces
                        inicio
                            encontrado<--verdadero
                            N<--N-1
                             para k desde i hasta N haz
                                    arreglo[k]<--arreglo[k-1]
                        fin
                en caso contrario
                        i<--i+1
            fin
fin
Si encontrado=falso entonces
            mensaje (valor no encontrado)
```

3.- Modificar.

```
Si N>=1 entonces
    inicio
        i<--1
        encontrado<--falso
        mientras i<=N y encontrado=false haz
            inicio
                Si arreglo[i]=valor entonces
                    arreglo[i]<--valor_nuevo
                    encontrado<--verdadero
                En caso contrario
                    i<--i+1
            fin
    fin
```

Matríz Poco Densa Regular

Una matríz poco densa es aquella que está formada por elementos que en su mayoría son ceros. Este tipo de matrices son matrices cuadradas que se dividen en los siguientes tipos:

- Matríz triangular superior
- Matríz triangular inferior
- Matríz tridiagonal

MATRIZ TRIANGULAR SUPERIOR

En este tipo de matríz los elementos iguales a cero se encuentran debajo de la diagonal principal. Ejemplo:

$$M = \begin{bmatrix} 5 & 8 & 2 & 7 \\ 4 & -3 & -6 & 0 \\ 9 & -5 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

Para evitar el desperdicio de memoria que se ocasionaría al almacenar una matríz en donde la mayoría de los elementos son ceros, es conveniente traspasar a un arreglo unidimensional todos los elementos diferentes de cero. El arreglo con los elementos distintos de cero de la matríz anterior es el siguiente:

5	8	2	7	4	-3	-6	9	-5	1

Una vez que hallamos vaciado la matríz, es indispensable conocer el lugar dentro del arreglo unidimensional en el cual quedaron situados los elementos, y esto se logra con la siguiente formula:

LOC(A[i,j])=base(A) + (n*(i-1)) - ((i-2)*(i-1))/2 + (j-1)

donde:
A=Matríz triangular superior
n=No. total de elementos
j= renglones
i=columnas

MATRIZ TRIANGULAR INFERIOR

En este tipo de matrices los elementos iguales a cero se encuentran por encima de la diagonal principal. Ejemplo:

$$M = \begin{bmatrix} 5 & 0 & 0 & 0 \\ 8 & 2 & 0 & 0 \\ 6 & 1 & -4 & 0 \\ 3 & 7 & 6 & 9 \end{bmatrix}$$

Una vez que vaciamos la matríz en un arreglo unidimensional, la formula para obtener las posiciones de los elementos es la siguiente:

5	8	2	6	1	-4	3	7	6	9

LOC(A[i,j])=base(A) + ((i-1)*i)/2 + (j-1)

MATRIZ TRIDIAGONAL

En ésta, los elementos diferentes de cero se encuentran en la diagonal principal ó en las diagonales por debajo ó encima de ésta. Ejemplo:

$$M = \begin{bmatrix} 4 & 5 & 0 & 0 \\ 8 & 3 & 2 & 0 \\ 0 & 7 & 6 & 1 \\ 0 & 0 & 9 & -5 \end{bmatrix}$$

Y el arreglo con los elementos diferentes de cero correspondiente a esta matríz es el siguiente:

4	5	8	3	2	7	6	1	9	5

La localización de los elementos distintos de cero en el arreglo unidimensional se realiza aplicando la siguiente formula:

LOC(A[i,j])=base(A) + 2*i + (j-3)

Ordenaciones en Arreglos

La importancia de mantener nuestros arreglos ordenados radica en que es mucho más rápido tener acceso a un dato en un arreglo ordenado que en uno desordenado.
Existen muchos algoritmos para la ordenación de elementos en arreglos, enseguida veremos algunos de ellos.

a)Selección Directa

Este método consiste en seleccionar el elemento más pequeño de nuestra lista para colocarlo al inicio y así excluirlo de la lista.

Para ahorrar espacio, siempre que vayamos a colocar un elemento en su posición correcta lo intercambiaremos por aquel que la esté ocupando en ese momento.

El algoritmo de selección directa es el siguiente:

```
i <- 1
mientras i<= N haz
    min <-i
    j <- i + 1
    mientras j <= N haz
        si arreglo[j] < [min] entonces
            min <-j
        j <- j + 1
    intercambia(arreglo[min],arreglo[i])
    i <- i +1
```

b)Ordenación por Burbuja

Es el método de ordenación más utilizado por su fácil comprensión y programación, pero es importante señalar que es el más ineficiente de todos los métodos .

Este método consiste en llevar los elementos menores a la izquierda del arreglo ó los mayores a la derecha del mismo. La idea básica del algoritmo es comparar pares de elementos adyacentes e intercambiarlos entre sí hasta que todos se encuentren ordenados.

```
i <- 1
mientras i < N haz
    j <- N
    mientras j > i haz
        si arreglo[j] < arreglo[j-1] entonces
            intercambia(arreglo[j],arreglo[j-1])
        j < j - 1
    i <- i +1
```

c)Ordenación por Mezcla

Este algoritmo consiste en partir el arreglo por la mitad, ordenar la mitad izquierda, ordenar la mitad derecha y mezclar las dos mitades ordenadas en un array ordenado. Este último paso consiste en ir comparando pares sucesivos de elementos (uno de cada mitad) y poniendo el valor más pequeño en el siguiente hueco.

```
procedimiento mezclar(dat,izqp,izqu,derp,deru)
inicio
    izqa <- izqp
    dera <- derp
    ind <- izqp
    mientras (izqa <= izqu) y (dera <= deru) haz
        si arreglo[izqa] < dat[dera] entonces
            temporal[ind] <- arreglo[izqa]
            izqa <- izqa + 1
        en caso contrario
            temporal[ind] <- arreglo[dera]
            dera <- dera + 1
        ind <- ind +1
    mientras izqa <= izqu haz
        temporal[ind] <- arreglo[izqa]
        izqa <- izqa + 1
        ind <- ind +1
    mientras dera <= deru haz
        temporal[ind] <=dat[dera]
        dera <- dera + 1
        ind <- ind + 1
    para ind <- izqp hasta deru haz
        arreglo[ind] <- temporal[ind]
fin
```

Búsquedas en Arreglos

Una búsqueda es el proceso mediante el cual podemos localizar un elemento con un valor especifico dentro de un conjunto de datos. Terminamos con éxito la búsqueda cuando el elemento es encontrado.

A continuación veremos algunos de los algoritmos de búsqueda que existen.

a)Búsqueda Secuencial

A este método tambien se le conoce como búsqueda lineal y consiste en empezar al inicio del conjunto de elementos , e ir atravez de ellos hasta encontrar el elemento indicado ó hasta llegar al final de arreglo.
Este es el método de búsqueda más lento, pero si nuestro arreglo se encuentra completamente desordenado es el único que nos podrá ayudar a encontrar el dato que buscamos.

```
ind <- 1
encontrado <- falso
mientras no encontrado y ind < N haz
        si arreglo[ind] = valor_buscado entonces
                encontrado <- verdadero
        en caso contrario
                ind <- ind +1
```

b)Búsqueda Binaria

Las condiciones que debe cumplir el arreglo para poder usar búsqueda binaria son que el arreglo este ordenado y que se conozca el numero de elementos.
Este método consiste en lo siguiente: comparar el elemento buscado con el elemento situado en la mitad del arreglo, si tenemos suerte y los dos valores coinciden, en ese momento la búsqueda termina. Pero como existe un alto porcentaje de que esto no ocurra, repetiremos los pasos anteriores en la mitad inferior del arreglo si el elemento que buscamos resulto menor que el de la mitad del arreglo, o en la mitad superior si el elemento buscado fue mayor.
La búsqueda termina cuando encontramos el elemento o cuando el tamaño del arreglo a examinar sea cero.

```
encontrado <- falso
primero <- 1
ultimo <- N
mientras primero <= ultimo y no encontrado haz
```

```
        mitad <- (primero + ultimo)/2
        si arreglo[mitad] = valor_buscado entonces
                encncntrado <- verdadero
        en caso contrario
                si arreglo[mitad] > valor_buscado entonces
                        ultimo <- mitad - 1
                en caso contrario
                        primero <- mitad + 1
```

c)Búsqueda por Hash

La idea principal de este método consiste en aplicar una función que traduce el valor del elemento buscado en un rango de direcciones relativas. Una desventaja importante de este método es que puede ocasionar colisiones.

```
funcion hash (valor_buscado)
        inicio
                hash <- valor_buscado mod numero_primo
        fin

inicio <- hash (valor)
il <- inicio
encontrado <- falso
repite
        si arreglo[il] = valor entonces
                encontrado <- verdadero
        en caso contrario
                il <- (il +1) mod N
hasta encontrado o il = inicio
```

COLAS

Una cola es una estructura de almacenamiento, donde la podemos considerar como una lista de elementos, en la que éstos van a ser insertados por un extremo y serán extraídos por otro.
Las colas son estructuras de tipo FIFO (first-in, first-out), ya que el primer elemento en entrar a la cola será el primero en salir de ella.
Existen muchísimos ejemplos de colas en la vida real, como por ejemplo: personas esperando en un teléfono público, niños esperando para subir a un juego mecánico, estudiantes esperando para subir a un camión escolar, etc.

Representación en Memoria

Podemos representar a las colas de dos formas :

- Como arreglos
- Como listas ordenadas

En esta unidad trataremos a las colas como arreglos de elementos, en donde debemos definir el tamaño de la cola y dos apuntadores, uno para accesar el primer elemento de la lista y otro que guarde el último. En lo sucesivo, al apuntador del primer elemento lo llamaremos F, al de el último elemento A y MAXIMO para definir el número máximo de elementos en la cola.

Cola Lineal

La cola lineal es un tipo de almacenamiento creado por el usuario que trabaja bajo la técnica FIFO (primero en entrar primero en salir). Las colas lineales se representan gráficamente de la siguiente manera:

Las operaciones que podemos realizar en una cola son las de inicialización, inserción y extracción. Los algoritmos para llevar a cabo dichas operaciones se especifican más adelante.
Las condiciones a considerar en el tratamiento de colas lineales son las siguientes:

- Overflow (cola llena), cuando se realice una inserción.
- Underflow(cola vacía), cuando se requiera de una extracción en la cola.
- Vacío

ALGORITMO DE INICIALIZACIÓN

```
F < -- 1
A <-- 0
```

ALGORITMO PARA INSERTAR

```
Si A=máximo entonces
    mensaje (overflow)
en caso contrario
     A<-- A+1
     cola[A]<-- valor
```

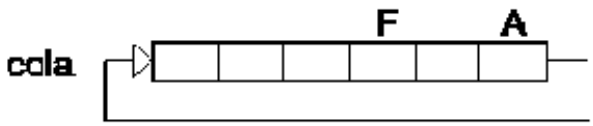
ALGORITMO PARA EXTRAER

```
Si A&ltF entonces

      mensaje (underflow)
en caso contrario
      F <-- F+1
       x <-- cola[F]
```

Cola Circular

Las colas lineales tienen un grave problema, como las extracciones sólo pueden realizarse por un extremo, puede llegar un momento en que el apuntador A sea igual al máximo número 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).
Para solucionar el problema de desperdicio de memoria se implementaron las colas circulares, en las cuales existe un apuntador desde el último elemento al primero de la cola.
La representación gráfica de esta estructura es la siguiente:



La condición de vacío en este tipo de cola es que el apuntador F sea igual a cero.

Las condiciones que debemos tener presentes al trabajar con este tipo de estructura son las siguientes:

- Over flow, cuando se realice una inserción.
- Under flow, cuando se requiera de una extracción en la cola.
- Vacio

ALGORITMO DE INICIALIZACIÓN

F < -- 0
A<-- 0

ALGORITMO PARA INSERTAR

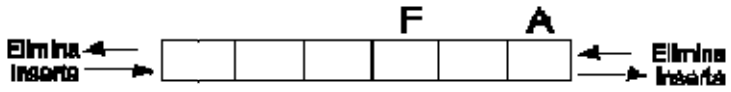
```
Si (F+1=A) ó (F=1 y A=máximo) entonces
        mensaje (overflow)
en caso contrario
        inicio
                si A=máximo entonces
                        A<--1
                        cola[A]<-- valor
                en caso contrario
                        A <--A+1
                         cola[A]<-- valor
                si F=0 entonces
                         F <-- 1
         fin
```

ALGORITMO PARA EXTRAER

```
Si F=0 entonces
        mensaje (underflow)
en caso contrario
        x <-- cola[F]
         si F=A entonces
                F <-- 0
                A<-- 0
         en caso contrario
                si F=máximo entonces
                        F <--1 en caso contrario F <-- F+1
```

Doble Cola

Esta estructura es una cola bidimensional en que las inserciones y eliminaciones se pueden realizar en cualquiera de los dos extremos de la bicola. Gráficamente representamos una bicola de la siguiente manera:



Existen dos variantes de la doble cola:

- Doble cola de entrada restringida.
- Doble cola de salida restringida.

La primer variante sólo acepta inserciones al final de la cola, y la segunda acepta eliminaciones sólo al frente de la cola

ALGORITMOS DE ENTRADA RESTRINGIDA

Algoritmo de Inicialización

```
F < -- 1
A <-- 0
```

Algoritmo para Insertar

```
Si A=máximo entonces
        mensaje (overflow)
en caso contrario
        A <--A+1
        cola[A]<-- valor
```

Algoritmo para Extraer

```
Si F&gtA entonces
        mensaje (underflow)
en caso contrario
        mensaje (frente/atrás)
            si frente entonces
                x <-- cola[F]
                F <-- F+1
            en caso contrario
                x <-- cola[A]
                A <-- A-1
```

ALGORITMOS DE SALIDA RESTRINGIDA

Algoritmo de Inicialización

```
F <--1
A <-- 0
```

Algoritmo para Insertar

```
Si F&gtA entonces
        mensaje (overflow)
en caso contrario
        mensaje (Frente/Atrás)
         si Frente entonces
              cola[F] <--valor
         en caso contrario
              A <-- A+1
              cola[A] <--valor
```

Algoritmo para Extraer

```
Si F=0 entonces
        mensaje (underflow)
en caso contrario
        x <--cola[F]
        F <-- F+1
```

Cola de Prioridades

Esta estructura es un conjunto de elementos donde a cada uno de ellos se les asigna una prioridad, y la forma en que son procesados es la siguiente:

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

Algoritmo para Insertar

```
x <--1
final<--verdadero
para i desde 1 hasta n haz
    Si cola[i]&gtprioridad entonces
        x <--i
        final <--falso
        salir
        si final entonces
            x <--n+1
        para i desde n+1 hasta x+1
            cola[i] <--prioridad
            n <-- n+1
```

Algoritmo para Extraer

```
Si cola[1]=0 entonces
    mensaje(overflow)
en caso contrario
    procesar <--cola[1]
    para i desde 2 hasta n haz
        cola[i-1] <--cola[1]
        n <-- n-1
```

Operaciones en Colas

Las operaciones que nosotros podemos realizar sobre una cola son las siguientes:

- Inserción.
- Extracción.

Las inserciones en la cola se llevarán a cabo por atrás de la cola, mientras que las eliminaciones se realizarán por el frente de la cola (hay que recordar que el primero en entrar es el primero en salir).

PILAS

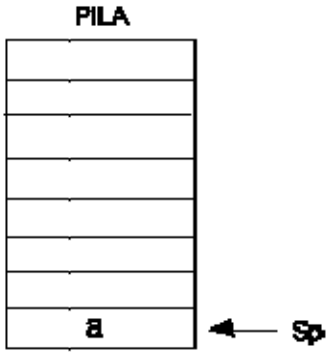
Las pilas son otro tipo de estructura de datos lineales, las cuales presentan restricciones en cuanto a la posición en la cual pueden realizarse las inserciones y las extracciones de elementos.
Una pila es una lista de elementos en la que se pueden insertar y eliminar elementos sólo por uno de los extremos. Como consecuencia, los elementos de una pila serán eliminados en orden inverso al que se insertaron. Es decir, el último elemento que se metió a la pila será el primero en salir de ella.
En la vida cotidiana existen muchos ejemplos de pilas, una pila de platos en una alacena, una pila de latas en un supermercado, una pila de papeles sobre un escritorio, etc.
Debido al orden en que se insertan y eliminan los elementos en una pila, también se le conoce como estructura LIFO (Last In, First Out: último en entrar, primero en salir).

Representación en Memoria

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

- Arreglos.
- Listas enlazadas.

Nosotros ahora usaremos los arreglos. Por lo tanto debemos definir el tamaño máximo de la pila, además de un apuntador al último elemento insertado en la pila el cual denominaremos SP. La representación gráfica de una pila es la siguiente:



Como utilizamos arreglos para implementar pilas, tenemos la limitante de espacio de memoria reservada. Una vez establecido un máximo de capacidad para la pila, ya no es posible insertar más elementos.
Una posible solución a este problema es el uso de espacios compartidos de memoria. Supongase que se necesitan 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 que dos arreglos de n elementos.
En este caso utilizaremos dos apuntadores: SP1 para apuntar al último elemento insertado en la pila 1 y SP2 para apuntar al último elemento insertado en la pila 2. Cada una de las pilas insertará sus elementos por los extremos opuestos, es decir, la pila 1 iniciará a partir de la localidad 1 del arreglo y la pila 2 iniciará en la localidad 2n. De este modo si la pila 1 necesita más de n espacios (hay que recordar que a cada pila se le asignaron n localidades) y la pila 2 no tiene ocupados sus n lugares, entonces se podrán seguir insertando elementos en la pila 1 sin caer en un error de desbordamiento.

Notación Infija, Postfija y Prefija

Las pilas son estructuras de datos muy usadas para la solución de diversos tipos de problemas. Pero tal vez el principal uso de estas estructuras es el tratamiento de expresiones matemáticas.

ALGORITMO PARA CONVERTIR EXPRESIONES INFIJAS EN POSTFIJAS (RPN)

1. Incrementar la pila
2. Inicializar el conjunto de operaciones
3. Mientras no ocurra error y no sea fin de la expresión infija haz
    - Si el carácter es:
        1. PARENTESIS IZQUIERDO. Colocarlo en la pila
        2. PARENTESIS DERECHO. Extraer y desplegar los valores hasta encontrar paréntesis izquierdo. Pero NO desplegarlo.
        3. UN OPERADOR.
            - Si la pila esta vacía o el carácter tiene más alta prioridad que el elemento del tope de la pila insertar el carácter en la pila.
            - En caso contrario extraer y desplegar el elemento del tope de la pila y repetir la comparación con el nuevo tope.
        4. OPERANDO. Desplegarlo.
4. Al final de la expresión extraer y desplegar los elementos de la pila hasta que se vacíe.

ALGORITMO PARA EVALUAR UNA EXPRESION RPN

1. Incrementar la pila
2. Repetir
    - Tomar un caracter.
    - Si el caracter es un operando colocarlo en la pila.
    - Si el caracter es un operador entonces tomar los dos valores del tope de la pila, aplicar el operador y colocar el resultado en el nuevo tope de la pila. (Se produce un error en caso de no tener los 2 valores)
3. Hasta encontrar el fin de la expresión RPN.

Recursión

Podemos definir la recursividad como un proceso que se define en términos de sí mismo.

El concepto de recursión es difícil de precisar, pero existen ejemplos de la vida cotidiana que nos pueden servir para darnos una mejor idea acerca de lo que es recursividad. Un ejemplo de esto es cuando se toma una fotografía de una fotografía, o cuando en un programa de televisión un periodista transfiere el control a otro periodista que se encuentra en otra ciudad, y este a su vez le transfiere el control a otro.

Casos típicos de estructuras de datos definidas de manera recursiva son los árboles binarios y las listas enlazadas.

La recursión se puede dar de dos formas:

- *DIRECTA.* Este tipo de recursión se da cuando un subprograma se llama directamente a sí mismo.
- *INDIRECTA* Sucede cuando un subprograma llama a un segundo subprograma, y este a su vez llama al primero, es decir el subproceso A llama al B, y el B invoca al subproceso A.

Implementar Recursión Usando Pilas

Otra de las aplicaciones en las que podemos utilizar las pilas es en la implementación de la recursividad. A continuación se mostrarán algunos ejemplos.

```
          |
          |    1 ,              N=0
Factorial   <
          |    N*(n-1)!,     N&gt0
          |
```

```
sp <--0
mientras n <> 1 haz
      push(pila,n)
      n<--n-1
mientras sp <> 0 haz
      factorial<--factorial*pop(pila)
```

```
          |
          |    0  ,             si a < b
Q         <
          |  Q(a-b,b)+1,   si a<=b
          |
```

```
sp<--0
Q<--0
lee(a), lee(b)
mientras a>=b haz
      push(pila,1)
      a<--a-b
mientras sp< > 0 haz
      Q<-- Q + pop(pila)
```

Operaciones en Pilas

Las principales operaciones que podemos realizar en una pila son:

- Insertar un elemento (push).
- Eliminar un elemento (pop).

Los algoritmos para realizar cada una de estas operaciones se muestran a continuación. La variable máximo para hacer referencia al máximo número de elementos en la pila.

Inserción (Push)

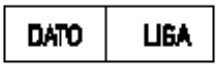
```
si sp=máximo entonces
        mensaje (overflow)
en caso contrario
        sp<-- sp+1
        pila[sp]<-- valor
```

Eliminación (Pop)

```
si sp=0 entonces
       mensaje (underflow)
en caso contrario
        x<--pila[sp]
        sp<--sp-1
```

UNIDAD III. 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.
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).
A continuación se muestra el esquema de una lista :

Operaciones en Listas Enlazadas

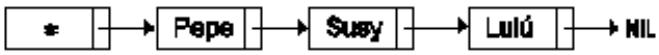
Las operaciones que podemos realizar sobre una lista enlazada son las siguientes:

- **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.

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 diferencíe 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.

Algoritmo de Creación

```
top<--NIL
repite
        new(p)
        leer(p(dato))
         si top=NIL entonces
       top<--p
         en caso contrario
       q(liga)<--p
          p(liga)<--NIL
          q<--p
          mensaje('otro nodo?')
           leer(respuesta)
hasta respuesta=no
```

Algoritmo para Recorrido

```
p<--top
mientras p<>NIL haz
       escribe(p(dato))
       p<--p(liga:)
```

Algoritmo para insertar al final

```
p<--top
mientras p(liga)<>NIL haz
       p<--p(liga)
new(q)
p(liga)<--q
q(liga)<--NIL
```

Algoritmo para insertar antes/después de 'X' información

```
p<--top
mensaje(antes/despues)
lee(respuesta)
si antes entonces
        mientras p<>NIL haz
                si p(dato)='x' entonces
                        new(q)
                        leer(q(dato))
                        q(liga)<--p
                        si p=top entonces
                                top<--q
                        en caso contrario
                                r(liga)<--q
                        p<--nil
                en caso contrario
                        r<--p
                        p<--p(link)
si despues entonces
        p<--top
        mientras p<>NIL haz
                si p(dato)='x' entonces
                        new(q)
                        leer(q(dato))
                        q(liga)<--p(liga)
                        p(liga)<--q
                        p<--NIL
                en caso contrario
                        p<--p(liga)
        p<--top
        mientras p(liga)<>NIL haz
                p<--p(liga)
        new(q)
        p(liga)<--q
        q(liga)<--NIL
```

Algoritmo para borrar un nodo

```
p<--top
leer(valor_a_borrar)
mientras p<>NIL haz
    si p(dato)=valor_a_borrar entonces
        si p=top entonces
            si p(liga)=NIL entonces
                top<--NIL
            en caso contrario
                top(liga)<--top(liga)
        en caso contrario
            q(liga)<--p(liga)
        dispose(p)
        p<--NIL
    en caso contrario
        q<--p
        p<--p(liga)
```

Algoritmo de creación de una lista con nodo de cabecera

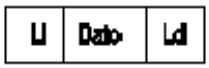
```
new(cab)
cab(dato)<--'*'
cab(liga)<--NIL
q<--cab
repite
     new(p)
     leer(p(dato))
     p(liga)<--NIL
     q<--p
     mensaje(otro nodo?)
      leer(respuesta)
hasta respuesta=no
```

Algoritmo de extracción en una lista con nodo de cabecera

```
leer(valor_a_borrar)
p<--cab
q<--cab(liga)
mientras q<>NIL haz
    si q(dato)=valor_a_borrar entonces
        p<--q(liga)
        dispose(q)
        q<--NIL
    en caso contrario
        p<--q
        q<--q(liga)
```

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:

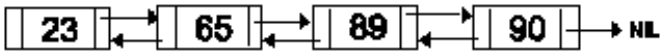


Existen dos tipos de listas doblemente ligadas:

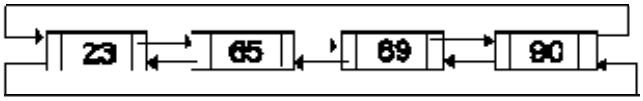
- **Listas dobles lineales.** En este tipo de lista doble, tanto el puntero izquierdo del primer nodo como el derecho del último nodo apuntan a NIL.
- **Listas dobles circulares.** En este tipo de lista doble, el puntero izquierdo del primer nodo apunta al último nodo de la lista, y el puntero derecho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

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.

Algoritmo de creación

```
top<--NIL
repite
     si top=NIL entonces
               new(p)
               lee(p(dato))
               p(ld)<--p
               p(li)<--p
               top<--p
    en caso contrario
               new(p)
               lee(p(dato))
               p(ld)<--top
               p(li)<--p
               p(ld(li))<--p
     mensaje(otro nodo?)
      lee (respuesta)
hasta respuesta=no
```

Algoritmo para recorrer la lista

```
--RECORRIDO A LA DERECHA.

p<--top
repite
    escribe(p(dato))
    p<--p(ld)
hasta p=top

--RECORRIDO A LA IZQUIERDA.

p<--top
repite
     escribe(p(dato))
      p<--p(li)
hasta p=top(li)
```

Algoritmo para insertar antes de 'X' información

```
p<--top
mensaje (antes de ?)
lee(x)
repite
     si p(dato)=x entonces
               new(q)
               leer(q(dato))
               si p=top entonces
                         top<--q
               q(ld)<--p
               q(li)<--p(li)
               p(ld(li))<--q
               p(li)<--q
               p<--top
     en caso contrario
               p<--p(ld)
hasta p=top
```

Algoritmo para insertar despues de 'X' información

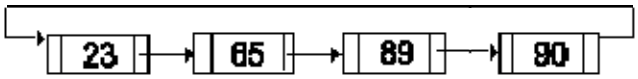
```
p<--top
mensaje(despues de ?)
lee(x)
repite
      si p(dato)=x entonces
            new(q)
            lee(q(dato))
            q(ld)<--p(ld)
            q(li)<--p
            p(li(ld))<--q
            p(ld)<--q
            p<--top
     en caso contrario
            p<--p(ld)
hasta p=top
```

Algoritmo para borrar un nodo

```
p<--top
mensaje(Valor a borrar)
lee(valor_a_borrar)
repite
     si p(dato)=valor_a_borrar entonces
            p(ld(li))<--p(ld)
            p(li(ld))<--p(li)
            si p=top entonces
                        si p(ld)=p(li) entonces
                                    top<--nil
                        en caso contrario
                                    top<--top(ld)
                        dispose(p)
                        p<--top
     en caso contrario
            p<--p(ld)
hasta p=top
```

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.



Enseguida se mostrarán los algoritmos más comunes en listas circulares. Al igual que en las secciones anteriores, utilizaremos el apuntador top para hacer referencia al primer nodo en la lista.

Algoritmo de creación

```
repite
      new(p)
      lee(p(dato))
       si top=nil entonces
        top<--p
        q<--p
      en caso contrario
        q(liga)<--p
        q<--p
     p(liga)<--top
     mensaje (otro nodo ?)
     lee(respuesta)
hasta respuesta=no
```

Algoritmo para recorrer la lista

```
p<--top
repite
      escribe(p(dato))
       p<--p(liga)
hasta p=top
```

Algoritmo para insertar antes de 'X' información

```
new(p)
lee(p(dato))
si top=nil entonces
      top<--p
      p(liga)<--top
en caso contrario
      mensaje(antes de ?)
      lee(x)
      q<--top
       r<--top(liga)
       repite
       si q(dato)=x entonces
            p(liga)<--q
            r(liga)<--p
            si p(liga)=top entonces
                 top<--p
       q<--q(liga)
       r<--r(liga)
     hasta q=top
```

Algoritmo para insertar después de 'X' información

```
new(p)
lee(p(dato))
mensaje(después de ?)
lee(x)
q<--top
r<--top(liga)
repite
     si q(dato)=x entonces
       q(liga)<--p
       p(liga)<--r
     q<--q(liga)
     r<--r(liga)
hasta q=top
```

Algoritmo para borrar

```
mensaje(valor a borrar )
lee(valor_a_borrar)
q<--top
r<--top
p<--top
mientras q(liga)<>top haz
      q<--q(liga)
repite
    si p(dato)=valor_a_borrar entonces
      si p=top entonces
            si top(liga)=top entonces
                  top<--NIL
            en caso contrario
                  top<--top(liga)
                  q(liga)<--top
      en caso contrario
            r(liga)<--p(liga)
      dispose(p)
      p<--top
    en caso contrario
      r<--p
      p<--p(liga)
hasta p=top
```

Listas Ortogonales

En este tipo de lista se utiliza para representar matrices. Los nodos contienen cuatro apuntadores. Uno para apuntar al nodo izquierdo (li),otro para apuntar al derecho(ld), otro al nodo inferior(lb) y por último un apuntador al nodo superior(la).

Creación de una lista ortogonal

```
top<--nil
mensaje(número de renglones)
lee(númreo_renglones)
mensaje(número de columnas)
lee(número_columnas)
desde y=1 hasta número_renglones haz
        new(p)
        lee(p(dato))
        p(li)<--p
        p(ld)<--p
        si top=NIL entonces
                top<--p
                top(lb)<--p
        en caso contrario
                p(la)<--top(la)
                p(lb(la))<--p
                p(lb)<--top
        top(la)<--p
q<--top
desde y=1 hasta número_columnas-1 haz
        s<--NIL
        desde j=1 hasta número_renglones haz
                new(q)
                p(ld)<--q
                p(li)<--q(li)
                p(ld(li))<--p
                q(li)<--p
                si s=NIL entonces
                        s<--p
                        p(la)<--p
                        p(lb)<--p
                en caso contrario
                        p(la)<--s(la)
                        p(lb(la))<--p
                        p(lb)<--s
                        s(la)<--p
                q<--q(lb)
```

Algoritmo para recorrer una lista ortogonal

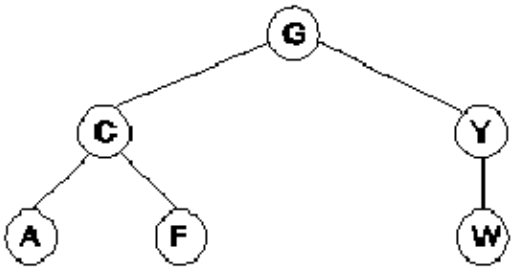
```
q<--top
repite
    p<--q
    repite
        escribe(p(dato))
        p<--p(ld)
    hasta p=q
    q<--q(lb)
hasta q=top
```

Borrado en listas ortogonales

```
q<--top
mensaje(valor a borrar)
lee(valor_a_borrar)
repite
    p<--q
    repite
        si p(dato)=valor_a_borrar entonces
                si p=top entonces
                        p(dato)<-- -999
                en caso contrario
                        aux<--p
                        p(ld(li))<--p(ld)
                        p(li(ld))<--p(li)
                        p(la(lb))<--p(la)
                        p(lb(la))<--p(lb)
                        dispose(aux)
                        p<--q
        en caso contrario
                p<--p(ld)
    hasta p=q
    q<--q(lb)
hasta q=top
```

UNIDAD IV. ESTRUCTURAS NO LINEALES

A los arboles ordenados de grado dos se les conoce como arboles binarios ya que cada nodo del árbol no tendrá más de dos descendientes directos. Las aplicaciones de los arboles binarios son muy variadas ya que se les puede utilizar para representar una estructura en la cual es posible tomar decisiones con dos opciones en distintos puntos.
La representación gráfica de un árbol binario es la siguiente:

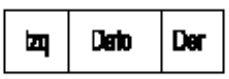


Representación en Memoria

Hay dos formas tradicionales de representar un árbol binario en memoria:

- Por medio de datos tipo punteros también conocidos como variables dinámicas o listas.
- Por medio de arreglos.

Sin embargo la más utilizada es la primera, puesto que es la más natural para tratar este tipo de estructuras.
Los nodos del árbol binario serán representados como registros que contendrán como mínimo tres campos. En un campo se almacenará la información del nodo. Los dos restantes se utilizarán para apuntar al subarbol izquierdo y derecho del subarbol en cuestión.
Cada nodo se representa gráficamente de la siguiente manera:



El algoritmo de creación de un árbol binario es el siguiente:

```
Procedimiento crear(q:nodo)
  inicio
            mensaje("Rama izquierda?")
            lee(respuesta)
            si respuesta = "si" entonces
                        new(p)
                        q(li) <-- nil
                        crear(p)
            en caso contrario
                        q(li) <-- nil
            mensaje("Rama derecha?")
            lee(respuesta)
            si respuesta="si" entonces
                        new(p)
                        q(ld)<--p
                        crear(p)
            en caso contrario
                        q(ld) <--nil
  fin
INICIO
            new(p)
            raiz<--p
            crear(p)
FIN
```

Clasificación de Arboles Binarios

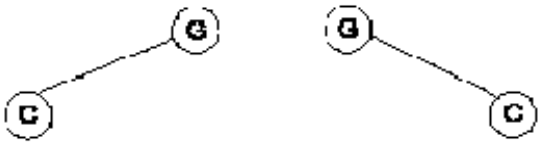
Existen cuatro tipos de árbol binario:.

- A. B. Distinto.
- A. B. Similares.
- A. B. Equivalentes.
- A. B. Completos.

A continuación se hará una breve descripción de los diferentes tipos de árbol binario así como un ejemplo de cada uno de ellos.

A. B. DISTINTO

Se dice que dos árboles binarios son distintos cuando sus estructuras son diferentes. Ejemplo:



A. B. SIMILARES

Dos arboles binarios son similares cuando sus estructuras son idénticas, pero la información que contienen sus nodos es diferente. Ejemplo:

A. B. EQUIVALENTES

Son aquellos arboles que son similares y que además los nodos contienen la misma información. Ejemplo:

A. B. COMPLETOS

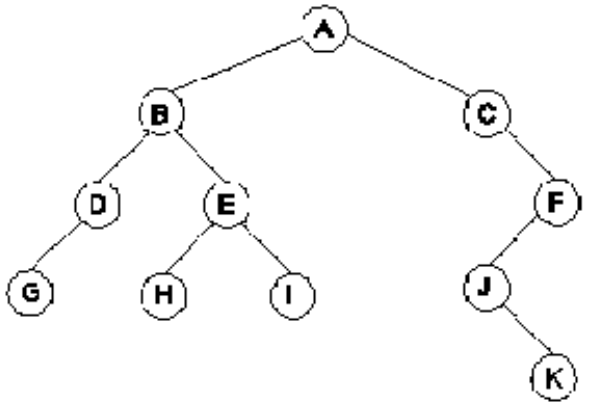
Son aquellos arboles en los que todos sus nodos excepto los del ultimo nivel, tiene dos hijos; el subarbol izquierdo y el subarbol derecho.

Recorrido de un Arbol Binario

Hay tres manera de recorrer un árbol : en inorden, preorden y postorden. Cada una de ellas tiene una secuencia distinta para analizar el árbol como se puede ver a continuación:

1. ***INORDEN***
    - Recorrer el subarbol izquierdo en inorden.
    - Examinar la raíz.
    - Recorrer el subarbol derecho en inorden.
2. ***PREORDEN***
    - Examinar la raíz.
    - Recorrer el subarbol izquierdo en preorden.
    - recorrer el subarbol derecho en preorden.
3. ***POSTORDEN***
    - Recorrer el subarbol izquierdo en postorden.
    - Recorrer el subarbol derecho en postorden.
    - Examinar la raíz.

A continuación se muestra un ejemplo de los diferentes recorridos en un árbol binario.



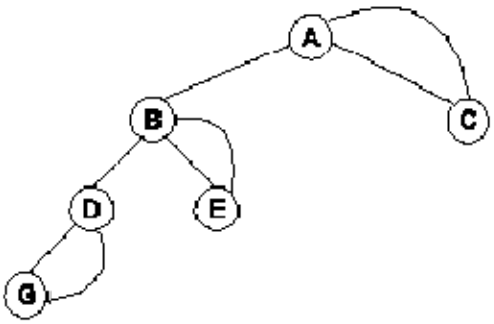
Inorden: GDBHEIACJKF

Preorden: ABDGEHICFJK

Postorden: GDHIEBKJFCA

Arboles Enhebrados

Existe un tipo especial de árbol binario llamado enhebrado, el cual contiene hebras que pueden estar a la derecha o a la izquierda. El siguiente ejemplo es un árbol binario enhebrado a la derecha.



- **ARBOL ENHEBRADO A LA DERECHA.** Este tipo de árbol tiene un apuntador a la derecha que apunta a un nodo antecesor.
- **ARBOL ENHEBRADO A LA IZQUIERDA.** Estos arboles tienen un apuntador a la izquierda que apunta al nodo antecesor en orden.

Control de Acceso a los Elementos de un Arreglo

En C++, el acceso a los elementos de un arreglo tiene que ser controlado por el programador, ya que el lenguaje no restringe la posibilidad de accesar a posiciones de memoria que están más abajo de la última posición reservada para el arreglo. Lo mismo sucede cuando se manejan cadenas, donde el programador tiene la responsabilidad de controlar el acceso a los caracteres de la cadena, tomando como límite el terminador nulo. En el listado 5.6 se presenta un ejemplo de acceso a los 5 bytes colocados abajo del terminador nulo de una cadena dada por el usuario.

```
#include     // Para gets() y puts()
#include     // Para clrscr() y gotoxy()
#include    // Para strlen()

void main()
{
  char nombre[31];
  clrscr();
  gotoxy(10,1);
  puts("¿ Cuál es tu nombre ? ");
  gotoxy(35,1);
  gets(nombre);
  clrscr();
  gotoxy (10,1);
  puts("ELEMENTO     CARACTER  DIRECCION");
  for( int x=0 ; (x < x,nombre[x],nombre[x],&nombre[x]); u?, c="%4d" printf(?nombre[%2d]
gotoxy(10,x+2); x++) x
```

Colocación de los elementos de una cadena en memoria RAM.

Arboles en Montón

Esta sección consiste en transformar un bosque en un árbol binario.

Entenderemos como bosque a un conjunto normalmente ordenado de dos o más árboles generales.

La serie de pasos que debemos seguir para lograr la conversión de un bosque en un árbol binario es la siguiente:

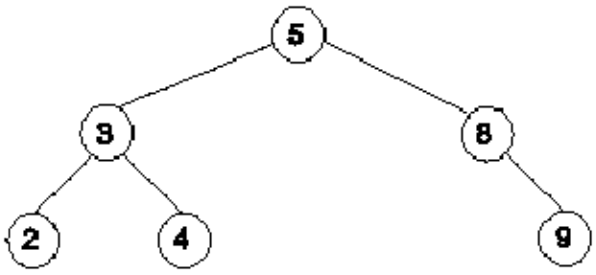
1. Enlazar horizontalmente las raíces de los distintos árboles generales.
2. Enlazar los hijos de cada nodo en forma horizontal (los hermanos).
3. Enlazar verticalmente el nodo padre con el hijo que se encuentra más a la izquierda. Además debe eliminarse el vínculo de ese padre con el resto de sus hijos.
4. Debe rotarse el diagrama resultante aproximadamente 45 grados hacia la izquierda y así se obtendrá el árbol binario correspondiente.

Arboles binarios de busqueda

Un árbol de búsqueda binaria es una estructura apropiada para muchas de las aplicaciones que se han discutido anteriormente con listas. La ventaja especial de utilizar un arbol es que se facilita la búsqueda.

Un árbol binario de búsqueda es aquel en el que el hijo de la izquierda (si existe) de cualquier nodo contiene un valor más pequeño que el nodo padre, y el hijo de la derecha (si existe) contiene un valor más grande que el nodo padre.

Un ejemplo de arbol binario de búsqueda es el siguiente:

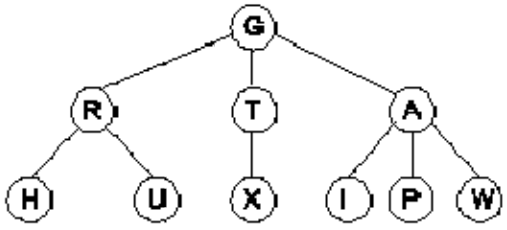


Los arboles representan las estructuras no lineales y dinámicas de datos más importantes en computación . Dinámicas porque las estructuras de árbol pueden cambiar durante la ejecución de un programa. No lineales, puesto que a cada elemento del árbol pueden seguirle varios elementos.

Los arboles pueden ser construidos con estructuras estáticas y dinámicas. Las estáticas son arreglos, registros y conjuntos, mientras que las dinámicas están representadas por listas.

La definición de árbol es la siguiente: es una estructura jerárquica aplicada sobre una colección de elementos u objetos llamados nodos; uno de los cuales es conocido como raíz. además se crea una relación o parentesco entre los nodos dando lugar a términos como padre, hijo, hermano, antecesor, sucesor, ansestro, etc.. Formalmente se define un árbol de tipo T como una estructura homogénea que es la concatenación de un elemento de tipo T junto con un número finito de arboles disjuntos, llamados subarboles. Una forma particular de árbol puede ser la estructura vacía.

La figura siguiente representa a un árbol general.



Se utiliza la recursión para definir un árbol porque representa la forma más apropiada y porque además es una característica inherente de los mismos.
Los arboles tienen una gran variedad de aplicaciones. Por ejemplo, se pueden utilizar para representar fórmulas matemáticas, para organizar adecuadamente la información, para construir un árbol genealógico, para el análisis de circuitos eléctricos y para numerar los capítulos y secciones de un libro.
La terminología que por lo regular se utiliza para el manejo de arboles es la siguiente:

- **HIJO.** X es hijo de Y, sí y solo sí el nodo X es apuntado por Y. También se dice que X es descendiente directo de Y.
- **PADRE.** X es padre de Y sí y solo sí el nodo X apunta a Y. También se dice que X es antecesor de Y.
- **HERMANO.** Dos nodos serán hermanos si son descendientes directos de un mismo nodo.
- **HOJA.** Se le llama hoja o terminal a aquellos nodos que no tienen ramificaciones (hijos).
- **NODO INTERIOR.** Es un nodo que no es raíz ni terminal.
- **GRADO.** Es el número de descendientes directos de un determinado nodo.
- **GRADO DEL ARBOL** Es el máximo grado de todos los nodos del árbol.
- **NIVEL.** Es el número de arcos que deben ser recorridos para llegar a un determinado nodo. Por definición la raíz tiene nivel 1.
- **ALTURA.** Es el máximo número de niveles de todos los nodos del árbol.
- **PESO.** Es el número de nodos del árbol sin contar la raíz.
- **LONGITUD DE CAMINO.** Es el número de arcos que deben ser recorridos para llegar desde la raíz al nodo X. Por definición la raíz tiene longitud de camino 1, y sus descendientes directos longitud de camino 2 y así sucesivamente.

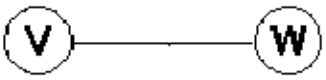
Transformación de un Arbol General en un Arbol Binario.

En esta sección estableceremos los mecanismos necesarios para convertir un árbol general en un árbol binario. Para esto, debemos seguir los pasos que se describen a continuación:

1. Enlazar los hijos de cada nodo en forma horizontal (los hermanos).
2. Enlazar en forma vertical el nodo padre con el nodo hijo que se encuentra más a la izquierda. Además, debe eliminarse el vínculo de ese padre con el resto de sus hijos.
3. Rotar el diagrama resultante aproximadamente 45 grados hacia la izquierda, y así se obtendrá el árbol binario correspondiente.

GRAFOS

Un grafo dirigido G consiste en un conjunto de vértices V y un conjunto de arcos o aristas A. Los vertice se denominan también nodos o puntos.
Un arco, es un par ordenado de vértices(V,W) donde V es el vértice inicial y W es el vértice terminal del arco. Un arco se expresa como: V-->W y se representa de la siguiente manera:



Los vértice de un grafo pueden usarse para representar objetos. Los arcos se utilizan para representar relaciones entre estos objetos.
Las aplicaciones más importantes de los grafos son las siguientes:

- Rutas entre ciudades.
- Determinar tiempos máximos y mínimos en un proceso.
- Flujo y control en un programa.

La terminología que manejaremos regularmente para el uso de grafos es la siguiente:

- **CAMINO.**Es una secuencia de vértices V1, V2, V3, ... , Vn, tal que cada uno de estos V1-&gtV2, V2-&gtV3, V1-&gtV3.
- **LONGITUD DE CAMINO.** Es el número de arcos en ese camino.
- **CAMINO SIMPLE.** Es cuando todos sus vértices, excepto tal vez el primero y el último son distintos.
- **CICLO SIMPLE.** Es un camino simple de longitud por lo menos de uno que empieza y termina en el mismo vértice.
- **ARISTAS PARALELAS.** Es cuando hay más de una arista con un vértice inicial y uno terminal dados.
- **GRAFO CICLICO.** Se dice que un grafo es cíclico cuando contiene por lo menos un ciclo.
- **GRAFO ACICLICO.** Se dice que un grafo es aciclíco cuando no contiene ciclos.
- **GRAFO CONEXO.** Un grafo G es conexo, si y solo si existe un camino simple en cualesquiera dos nodos de G.
- **GRAFO COMPLETO ó FUERTEMENTE CONEXO.**Un grafo dirigido G es completo si para cada par de nodos (V,W) existe un camino de V a W y de W a V (forzosamente tendrán que cumplirse ambas condiciones), es decir que cada nodo G es adyacente a todos los demás nodos de G.
- **GRAFO UNILATERALMENTE CONEXO.**Un grafo G es unilateralmente conexo si para cada par de nodos (V,W) de G hay un camino de V a W o un camino de W a V.
- **GRAFO PESADO ó ETIQUETADO.** Un grafo es pesado cuando sus aristas contienen datos (etiquetas). Una etiqueta puede ser un nombre, costo ó un valor de cualquier tipo de dato. También a este grafo se le denomina *red de actividades*, y el número asociado al arco se le denomina *factor de peso.*
- **VERTICE ADYACENTE.** Un nodo o vértice V es adyacente al nodo W si existe un arco de m a n.
- **GRADO DE SALIDA.**El grado de salida de un nodo V de un grafo G, es el número de arcos o aristas que empiezan en V.

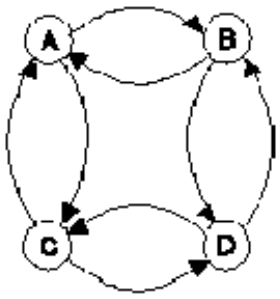
- **GRADO DE ENTRADA.**El grado de entrada de un nodo V de un grafo G, es el número de aristas que terminan en V.
- **NODO FUENTE.**Se le llama así a los nodos que tienen grado de salida positivo y un grado de entrada nulo.
- **NODO SUMIDERO.**Se le llama sumidero al nodo que tiene grado de salida nulo y un grado de entrada positivo.

Representación En Memoria Secuencial

Los grafos se representan en memoria secuencial mediante matrices de adyacencia.

Una matríz de adyacencia, es una matríz de dimensión n*n, en donde n es el número de vértices que almacena valores booleanos, donde matríz M[i,j] es verdadero si y solo si existe un arco que vaya del vértice y al vértice j.

Veamos el siguiente grafo dirigido:

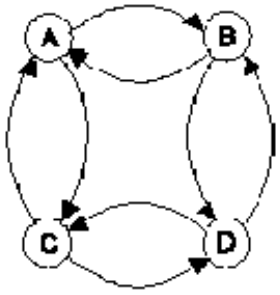


La matríz de adyacencia, que se obtuvo a partir del grafo anterior es la siguiente:

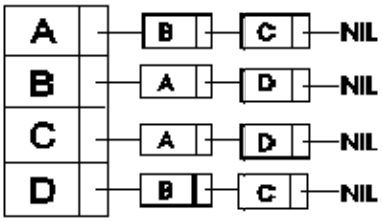
$$M = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

Representación en Memoria Enlazada

Los grafos se representan en memoria enlazada mediante listas de adyacencia.
Una lista de adyacencia, se define de la siguiente manera: Para un vértice i es una lista en cierto orden formada por todos los vértices adyacentes [a,i]. Se puede representar un grafo por medio de un arreglo donde cabeza de i es un apuntador a la lista de adyacencia al vértice i.
Veamos el siguiente grafo dirigido:



La lista de adyacencia, que se obtuvo a partir del grafo anterior es la siguiente:



Operaciones Sobre Grafos

En esta sección analizaremos algunas de las operaciones sobre grafos, como :

- Creación.
- Inserción.
- Búsqueda.
- Eliminación.

En esta sección, continuaremos utilizando los apuntadores que se usaron en las secciones anteriores. TOP para hacer referencia al primer nodo, LD para indicar liga derecha y LA para indicar liga abajo, por último usaremos los apuntadores P y Q para hacer referencia a los nuevos nodos que vayan a ser usados.

ALGORITMO DE CREACION.

```
repite
    si top=NIL entonces
      new(top)
      top(la)<--NIL
      top(ld)<--NIL
      lee(top(dato))
      q<--top
    en caso contrario
      new(p)
      p(ld)<--NIL
      p(la)<--NIL
      q(la)<--p
      lee(p(dato))
      q<--p
    mensaje(otro vertice ?)
     lee(respuesta)
hasta repuesta=no
p<--top
mientras p<>NIL haz
      mensaje(tiene vértices adyacentes p(dato) ?)
      lee(respuesta)
      si respueta=si entonces
            repite
                new(q)
                lee(q(dato))
                q(ld)<--p(ld)
                p(ld)<--q
                mensaje(otro vértice ?)
                lee(respuesta2)
            hasta respuesta2=no
      p<--p(la)
```

ALGORITMO DE INSERCION

```
mensaje(valor a insertar ?)
lee(valor_a_insertar)
si top<>NIL entonces
        p<--top
        mientras p(la)<>NIL haz
                p<--p(la)
        new(q)
        lee(q(dato))
        p(la)<--q
        q(la)<--NIL
        mensaje(Hay vértices adyacentes?)
        lee(respuesta)
        si respuesta=si entonces
                mensaje(Cuantos vértices?)
                lee(número_vértices)
                desde i=1 hasta número_vértices haz
                        new(p)
                        lee(p(dato))
                        q(ld)<--p
                        q<--q(ld)
en caso contrario
        mensaje(no existe lista)
```

ALGORITMO DE BUSQUEDA

```
mensaje(vértice a buscar)
lee(vértice_a_buscar)
p<--top
repite
     si p(dato)=vértice_a_buscar entonces
       repite
            p<--p(ld)
            escribe(p(dato))
       hasta p(ld)=NIL
     en caso contrario
       p<--(la)
hasta p=NIL
```

ALGORITMO DE BORRADO

```
mensaje(vértice a borrar ?)
lee(vértice_a_borrar)
p&Lt--top
r<--p
q<--p
sw<--falso
repite
    si p(dato)=vértice_a_borrar entonces
      si p=top entonces
            top<--top(la)
            r<--top
            sw<--verdadero
      en caso contrario
            r(la)<--p(la)
      repite
          p<--p(ld)
          dispose(q)
          q<--p
      hasta p=NIL
      si sw=verdadero entonces
            p<--r
            q<--p
      en caso contrario
            p<--r(la)
            q<--p
    en caso contrario
      r<--p
      repite
           q<--p(ld)
           si q(dato)=vértice_a_borrar entonces
             p(ld)<--q(ld)
             dispose(q)
             q<--p
           en caso contrario
             p<--p(ld)
hasta p=NIL
```

Camino Mínimo

Se denomina camino mínimo entre dos vértices V y W, al camino óptimo entre ambos vértices.Para determinar el camino mínimo entre dos vértices se utiliza el siguiente algoritmo:

```
desde i=1 hasta número_vértices haz
        desde j=1 hasta número_vértices haz
                si w[i,j]=0 entonces
                        q[[i,j]<--infinito
                en caso contrario
                        q[i,j]<--w[i,j]
desde k=1 hasta número_vértices haz
        desde i=1 hasta número_vértices haz
                desde j=1 hasta número_vértices haz
                        q[i,j]<--min(q[i,j], q[i,k] + q[k,j])
```