5 Tipos Abstractos de Datos

Contenido

1. Tipos abstractos de datos

- *Abstracción* o *encapsulamiento*:

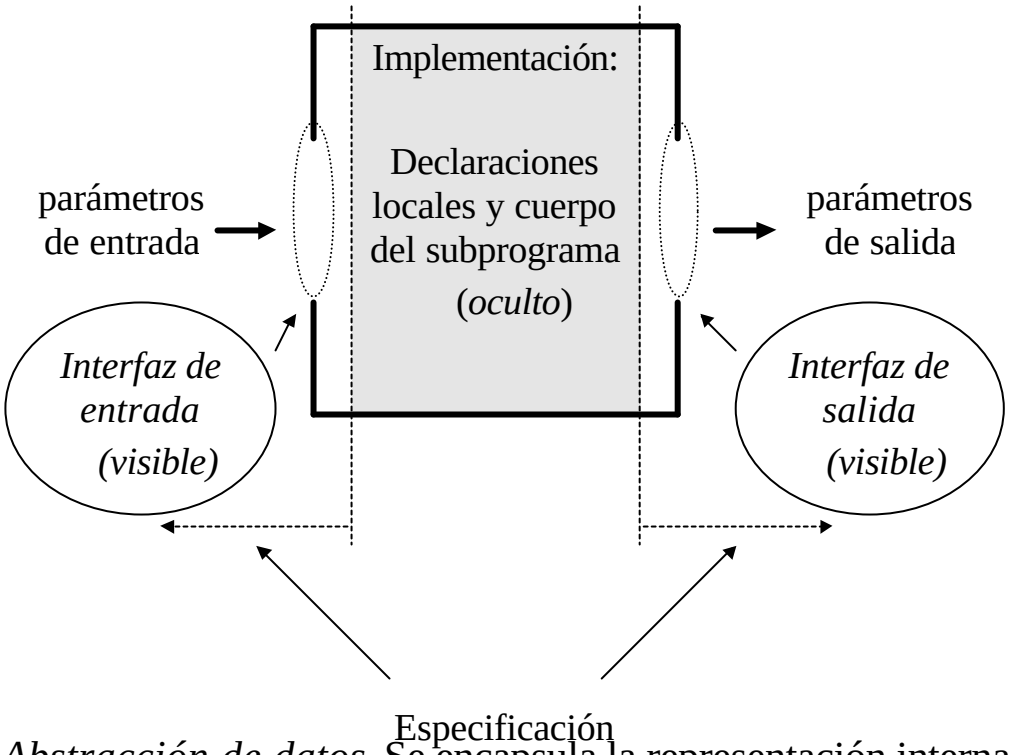
  Separación de la *especificación* de un objeto o algoritmo de su *implementación*, en base a que su *utilización* dentro de un programa sólo debe depender de un interfaz explícitamente definido (la especificación) y no de los detalles de su representación física (la implementación), los cuales están *ocultos.*

- Ventajas de la abstracción:

  Establece la *independencia* de QUÉ es el objeto (o QUÉ hace el algoritmo) y de CÓMO está implementado el objeto (o algoritmo), permitiendo la modificación del CÓMO sin afectar al QUÉ, y por lo tanto, sin afectar a los programas que utilizan este objeto o algoritmo.

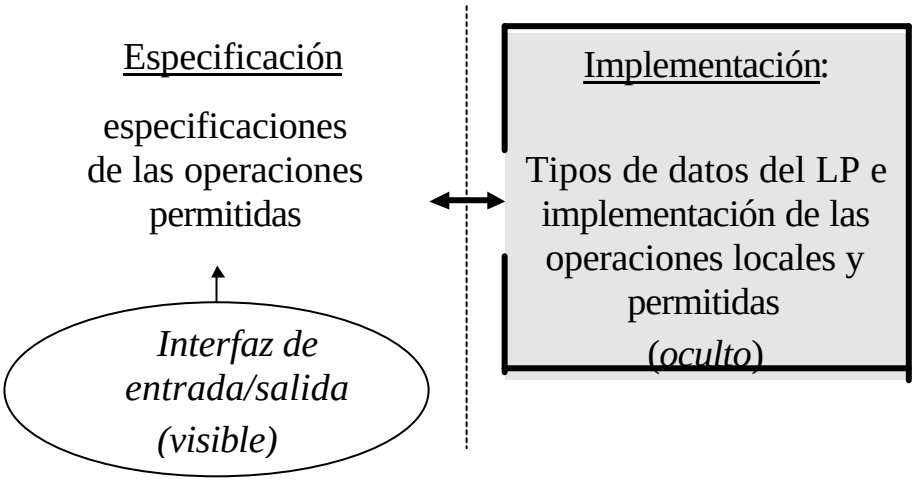
- Tipos de abstracciones:

  Abstracción de operaciones. Una serie de operaciones básicas se encapsulan para realizar una operación más compleja. En los lenguajes de programación este tipo de abstracción se logra mediante los *subprogramas*:

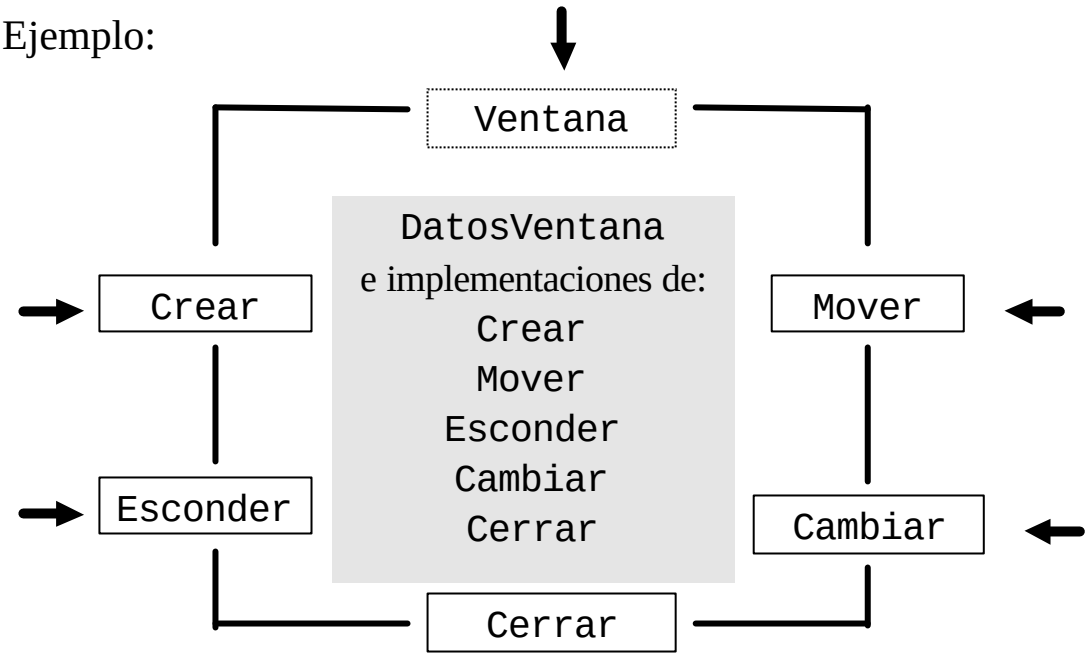


  Abstracción de datos. Se encapsula la representación interna de un dato junto con las implementaciones de todas las operaciones que se pueden realizar con ese dato. Esta encapsulación implica que estos datos

ocultos sólo pueden modificarse a través de la especificación (o interfaz de e/s). En los lenguajes de programación, la abstracción de datos se logra mediante los tipos de datos que suministra el lenguaje (enteros, reales, arrays, registros, …) y los subprogramas que van a implementar las operaciones permitidas, algunas de cuyas cabeceras formarán su especificación*.



- Ejemplo:



- En el siguiente punto del tema veremos cuatro tipos abstractos de datos (o TADs) muy utilizados en la programación:

  a) *Listas*

  b) *Colas*

* En algunos lenguajes como Modula-2, además, las reglas de ámbito y visibilidad ayudan a mantener ocultos datos y subprogramas que no deben ser accedidos directamente desde el exterior.

c) *Pilas*

d) *Arboles*

- Cada TAD consta de su *especificación o definición*, que será independiente de su *implementación* (un TAD puede implementarse de diversas formas siempre que cumplan su definición). Además, la definición nos proporcionará las *operaciones*, las cuales, a su vez, nos determinarán la *utilización* del TAD.

2. Listas

Especificación del TAD `Lista`

`TipoLista`

Colección de elementos homogéneos (del mismo tipo: `TipoElemento`*) con una relación LINEAL establecida entre ellos. Pueden estar ordenadas o no con respecto a algún valor de los elementos y se puede acceder a cualquier elemento de la lista.*

operaciones

```
TipoLista Crear()
void Imprimir(TipoLista lista)
int ListaVacía(TipoLista lista)
void Insertar(TipoLista lista, TipoElemento elem)
void Eliminar(TipoLista lista, TipoElemento elem)
```

Opcionalmente, si la implementación lo requiere, puede definirse:

```
int ListaLlena(TipoLista lista)
```

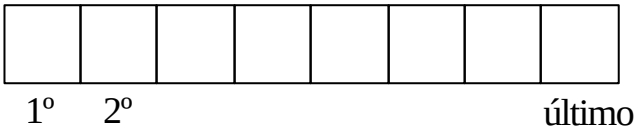
- Hay que tener en cuenta que la posición de la inserción no se especifica, por lo que dependerá de la implementación. No obstante, cuando la posición de inserción es la misma que la de eliminación, tenemos una subclase de lista denominada ***pila***, y cuando insertamos siempre por un extremo de la lista y eliminamos por el otro tenemos otra subclase de lista denominada ***cola***.
- En el procedimiento `Eliminar` el argumento `elem` podría ser una clave, un campo de un registro `TipoElemento`.

Implementación

La *implementación* o *representación física* puede variar:

- Representación ***secuencial***: el orden físico coincide con el orden lógico de la lista.

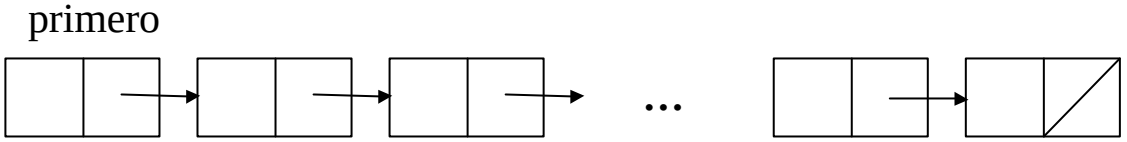
  ▸ *Ejemplo:* arrays. Para insertar o eliminar elementos podría exigirse que se desplazaran los elementos de modo que no existieran huecos. Por otra parte, tiene la desventaja de tener que dimensionar la estructura global de antemano, problema propio de las estructuras estáticas.



- Representación ***enlazada***: da lugar a la lista enlazada ya estudiada, en la que el orden físico no es necesariamente equivalente al orden lógico, el cual viene determinado por un campo de enlace explícito en cada elemento de la lista. ***Ventaja*:** se evitan movimientos de datos al insertar y eliminar elementos de la lista.

  Podemos implementar una lista enlazada de elementos mediante variables dinámicas o estáticas (el campo enlace será un puntero o un índice de array):

1. Variables de tipo puntero: Esta representación permite dimensionar en tiempo de ejecución (cuando se puede conocer las necesidades de memoria). La implementación de las operaciones son similares a las ya vista sobre la lista enlazada de punteros.



2. Variables estáticas: se utilizan arrays de registros con dos campos: elemento de la lista y campo enlace (de tipo índice del array) que determina el orden lógico de los elementos, indicando la posición del siguiente elemento de la lista (simulan un puntero). Esta representación es útil cuando el lenguaje de programación no dispone de punteros o cuando tenemos una estimación buena del tamaño total del array.

Ejemplo: Una lista formada por A,B,C,D,E (en este orden) puede representarse mediante el siguiente array de registros:

datos	A		D		B		E	C		
enlace	4		6		7		-1	4		
	0	1	2	3	4	5	6	7	8	9

Con esta implementación debemos escribir nuestros propios algoritmos de manejo de memoria (Asignar y free). Nuestra memoria para almacenar la lista de elementos será el array de registros. De él tomaremos memoria para almacenar un nuevo elemento, y en él liberaremos el espacio cuando un elemento se elimina de la lista.

Por tanto, en realidad coexisten dos listas enlazadas dentro del array de registros, una de memoria ocupada y otra de memoria libre, ambas terminadas con el enlace *nulo* `-1`.

datos	A		D		B		E	C		
enlace	4	5	6	8	7	3	-1	4	9	-1
	0	1	2	3	4	5	6	7	8	9

De esta forma, tendremos dos variables: *lista*, que contiene el índice del array donde comienza la lista enlazada de elementos (lista = 0) y *libre*, que contendrá el índice del array donde comienza la lista de posiciones de memoria libres (libre = 1).

Cuando queramos **añadir** un nuevo elemento a la lista, se tomará un hueco de la lista libre, disminuyendo ésta y aumentando la primera. Si queremos **eliminar** un elemento, realizaremos la operación contraria.

Cuando no existe ningún elemento en la lista, sólo existirá la lista libre, que enlaza todas las posiciones del array, mientras que si la lista se llena, no tendremos lista libre.

Las siguientes declaraciones nos servirían para construir esta representación, la cual se basa en un array y simula una lista enlazada:

```
/* CONSTANTES */
const  NULO = -1;  /* simula NULL de los punteros */
```

```
   const  Max = 100; /* simula el tope de la mem. dinám.
*/
   /* TIPOS */
      typedef int TipoPuntero; /* enlace genérico */
      typedef ¿???? TipoElemento;  /* cualquiera */
      struct {
         TipoElemento elem;
         TipoPuntero enlace;
      }TipoNodo;
      typedef TipoNodo Memoria[1][Max];
      typedef TipoPuntero TipoLista /* enlace cabecera */
   /* VARIABLES */
      Memoria monton;  /* memoria dinámica */
      TipoLista lista; /* lista nodos usados */
      TipoLista libre; /* lista nodos libres */
```

El valor NULO hace las veces de NULL como terminador de lista, tanto de la de nodos ocupados como de la de nodos libres. (Ver anexo).

Utilizamos el array y la variable Libre con ámbito global para simular el manejo de memoria real mediante la asignación dinámica de memoria.

TAD **Lista** (* representación enlazada con variables estáticas *)

```
#include <stdio.h>

/* CONSTANTES */
const NULO = -1;  /* Simula NULL de los punteros */
const Max = 100; /* Simula el tope de la memoria dinámica */

/* TIPOS */
typedef int TipoPuntero; /* Enlace genérico */
typedef int TipoElemento;  /* Entero */
typedef struct {
  TipoElemento elem;
  TipoPuntero enlace;
}TipoNodo;

typedef TipoNodo Memoria[100];
typedef TipoPuntero TipoLista; /* enlace cabecera */

/* VARIABLES */
Memoria monton;  /* memoria dinamica */
```

```
TipoLista libre; /* lista de nodos libres */
TipoLista lista; /* lista de nodos ocupados */


void Crear(void);
void Imprimir(void);
int ListaVacia(void);
void Insertar (TipoElemento elem);
void Eliminar(TipoElemento elem);
int ListaLlena(void);
void ObtenerNodo(TipoPuntero *ptr);

void main(){
  int i;         /* para recorrer bucles */

  Crear();
  if  (ListaVacia())  printf("\nInicialmente  la  lista  esta
vacia");

  for (i=0; i<10; i++)
      Insertar(i);

  printf("\nLa lista es: ");
  Imprimir();

  for (i=4; i<10; i++)
      if (i%2) /* Elimina los impares a partir del numero 5 */
          Eliminar(i);

  printf("\nDespues  de  elminar  los  impares,  la  lista  queda:
");
  Imprimir();

  if (!ListaLlena()) printf("\nLa lista no esta llena");

  Insertar(20);

  printf("\nPor ultimo, la lista es: ");
  Imprimir();

  printf("\Las 10 primeras posiciones del vector son: ");
  printf("\nElementos: ");
  for (i=0; i<10; i++)
      printf("%2d ", monton[i].elem);
  printf("\nEnlaces:   ");
  for (i=0; i<10; i++)
      printf("%2d ", monton[i].enlace);
  printf("\nlista: %d", lista);
```

```
  printf("\nlibre: %d", libre);

}

void Crear(){
  /* Crea la lista con todos los nodos libres inicialmente */
  TipoPuntero ptr;

  lista = NULO;
  libre = 0;
  for (ptr = 0; ptr < Max-1; ptr++)
      monton[ptr].enlace = ptr+1;
  monton[ptr].enlace = NULO;
}


int ListaVacia (){
  return(lista == NULO);
}


int ListaLlena(){
  return(libre == NULO);
}

void Imprimir(){
/* Saca por pantalla el contenido de toda la lista */
  TipoPuntero ptr;

  ptr = lista;
  while (ptr != NULO){
      printf("%d ", monton[ptr].elem);
      ptr = monton[ptr].enlace;
  }
}


void ObtenerNodo(TipoPuntero *p){ /* Simula p=malloc() */
  *p = libre;
  if (libre != NULO)
      libre = monton[libre].enlace;
}


void freeNodo(TipoPuntero *p){ /* Simula free(p) */
  monton[*p].enlace = libre;
  libre = *p;
```

```
  *p = NULO;
}


void Insertar(TipoElemento elem){
/* Se inserta al principio. Suponemos lista no ordenada */
  TipoPuntero ptr;

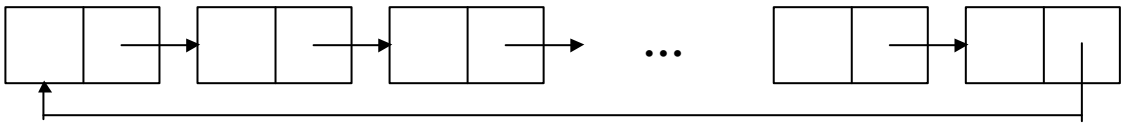
  ObtenerNodo(&ptr);
  monton[ptr].elem = elem;
  monton[ptr].enlace = lista;
  lista= ptr;
}


void Eliminar(TipoElemento elem){
  TipoPuntero actual, anterior;

  /* suponemos que el elemento a borrar está en la lista */
  actual = lista;
  anterior = NULO;
  while (monton[actual].elem != elem){
      anterior = actual;
      actual = monton[actual].enlace;
  }
  if (anterior == NULO)
      /* el nodo a eliminar está al principio de la lista */
      lista = monton[lista].enlace;
  else
      monton[anterior].enlace = monton[actual].enlace;
  freeNodo(&actual);
}
```

Listas circulares

- Son listas en las que el último elemento está enlazado con el primero, en lugar de contener el valor `NULL` o `NULO`. Evidentemente se implementarán con una representación enlazada (con punteros o variables estáticas). Con punteros sería:



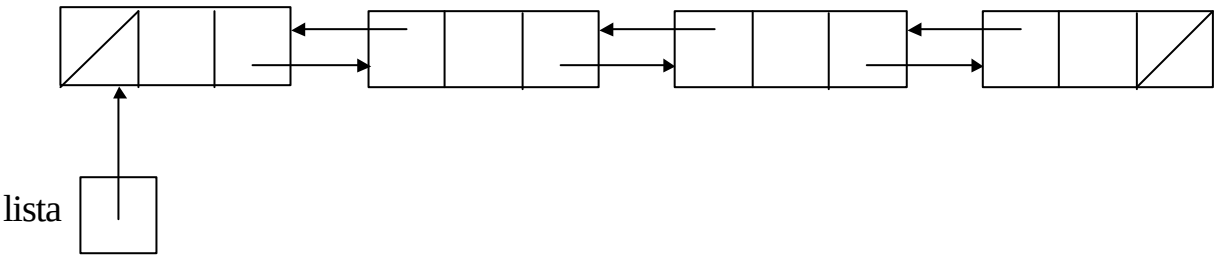
- Una lista vacía lo representaremos por el valor `NULL` o `NULO`. Es conveniente guardar en algún sitio el primer o último elemento de la lista circular para detectar el principio o el final de la lista, ya que `NULL` o `NULO` ahora sólo significa que la lista está vacía. Así, las comparaciones para detectar final de lista se harán con el propio puntero de la misma.
- Ejemplo: algoritmo que imprime en pantalla una lista circular con representación enlazada mixta (con variables estáticas):

```
void Imprimir (ListaCircular lista){
  TipoPuntero ptr; /* al registro */

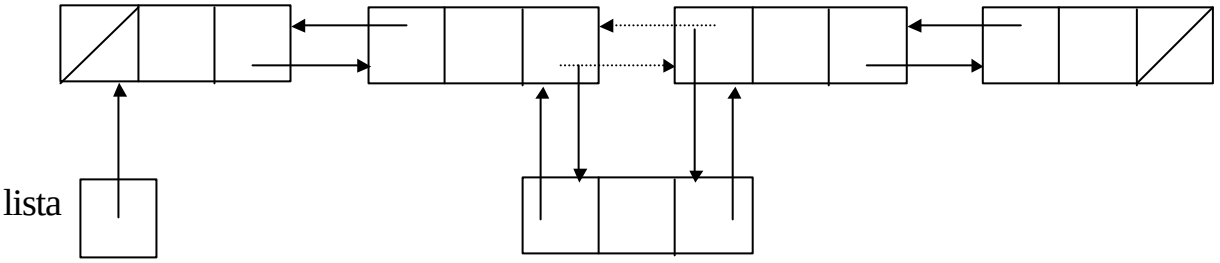
  /* suponemos que tenemos un puntero que
     apunta al primer nodo */
  ptr = lista;
  if (ptr!=NULO){
     do{
        printf("%d ", monton[ptr].elemento);
        ptr = monton[ptr].enlace;
     }while (ptr ¡= lista);
  }
}
```

Listas doblemente enlazadas

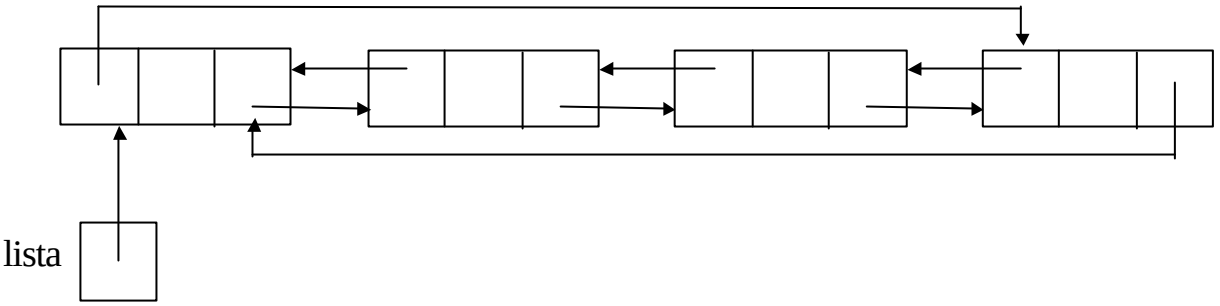
- Son listas en las que cada nodo, además de contener los datos información propios del nodo, contiene un enlace al nodo anterior y otro al nodo siguiente:



- Aparte de que ocupan más memoria (el tamaño de un puntero por cada nodo más), los algoritmos para implementar operaciones para listas dobles son más complicados que para las listas simples porque requieren manejar más punteros. Para insertar un nodo, por ejemplo, habría que cambiar cuatro punteros (o índices si se simulan con un array):



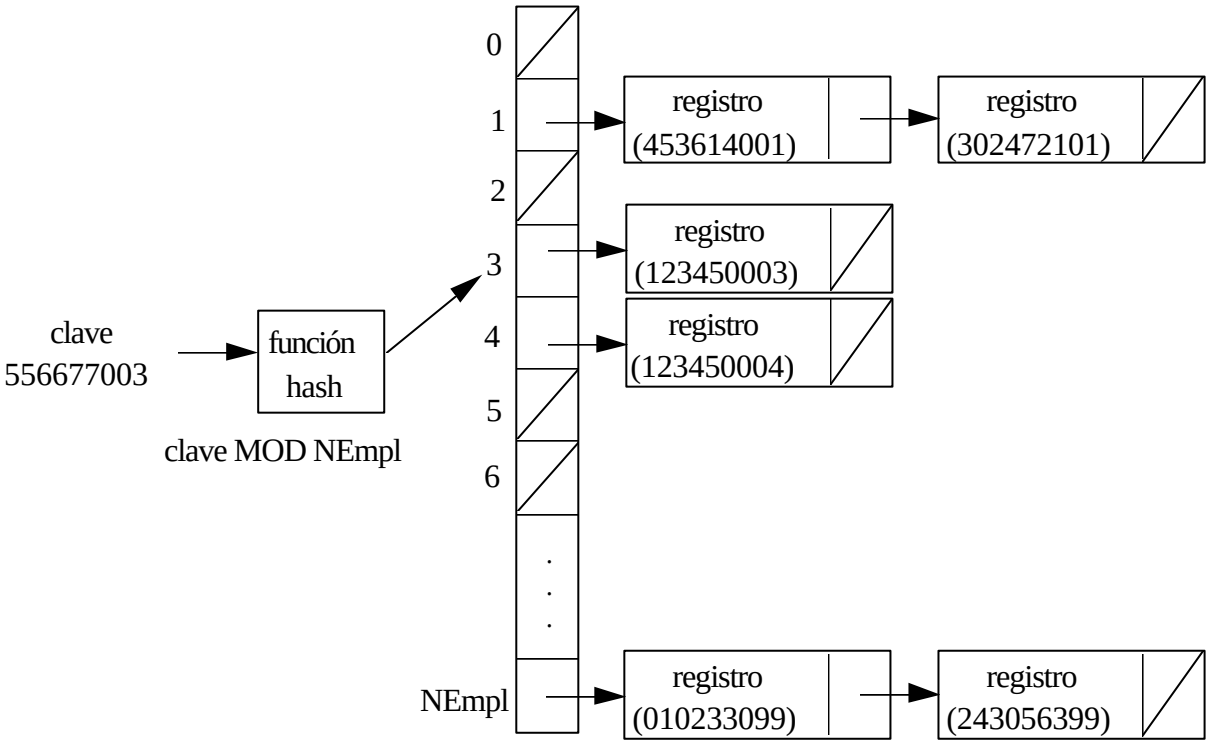
- La ventaja es que pueden ser recorridas en ambos sentidos, con la consiguiente eficiencia para determinados procesamientos.
- Una lista doble también puede ser circular:



Utilización de listas

- Las listas se utilizan en casi todo tipo de software, especialmente su representación enlazada con punteros. Un ejemplo de aplicación del TAD lista son las tablas hash con encadenamiento de sinónimos. La tabla sería un array de listas de longitud variable: un nodo por cada colisión, lo que evitaría sobredimensionar la tabla y además su posible desborde:

```
/* TIPOS */
  typedef ¿? TipoElemento /* un registro p. ej. */
  TipoLista TablaHash[NmEmpl];
```



- Un elemento de la tabla, `TablaHash[índice]`, sería un TAD lista, por lo que podríamos utilizar sus operaciones para almacenarlo:

```
Insertar(TablaHash[fhash(clave)], elemento);
```

- para buscarlo:

```
Buscar(TablaHash[fhash(clave)],clave,elemento,está)
```

- o para eliminarlo:

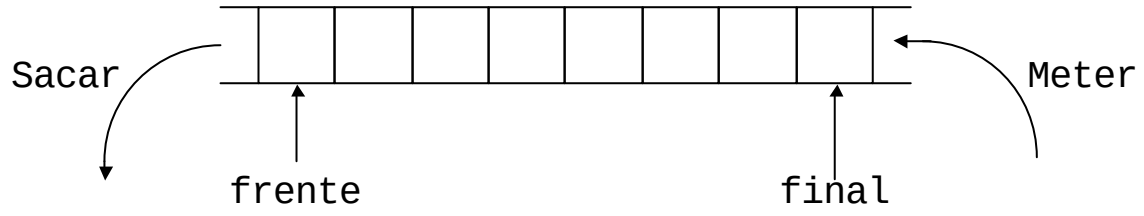
```
Eliminar(TablaHash[fhash(clave)], elemento)
```

3. Colas

Especificación del TAD `Cola`

`TipoCola`

Colección de elementos homogéneos (del mismo tipo: `TipoElemento`*) ordenados cronológicamente (por orden de inserción) y en el que sólo se pueden añadir elementos por un extremo (final) y sacarlos sólo por el otro (frente). Es una estructura FIFO (First In First Out):*



operaciones

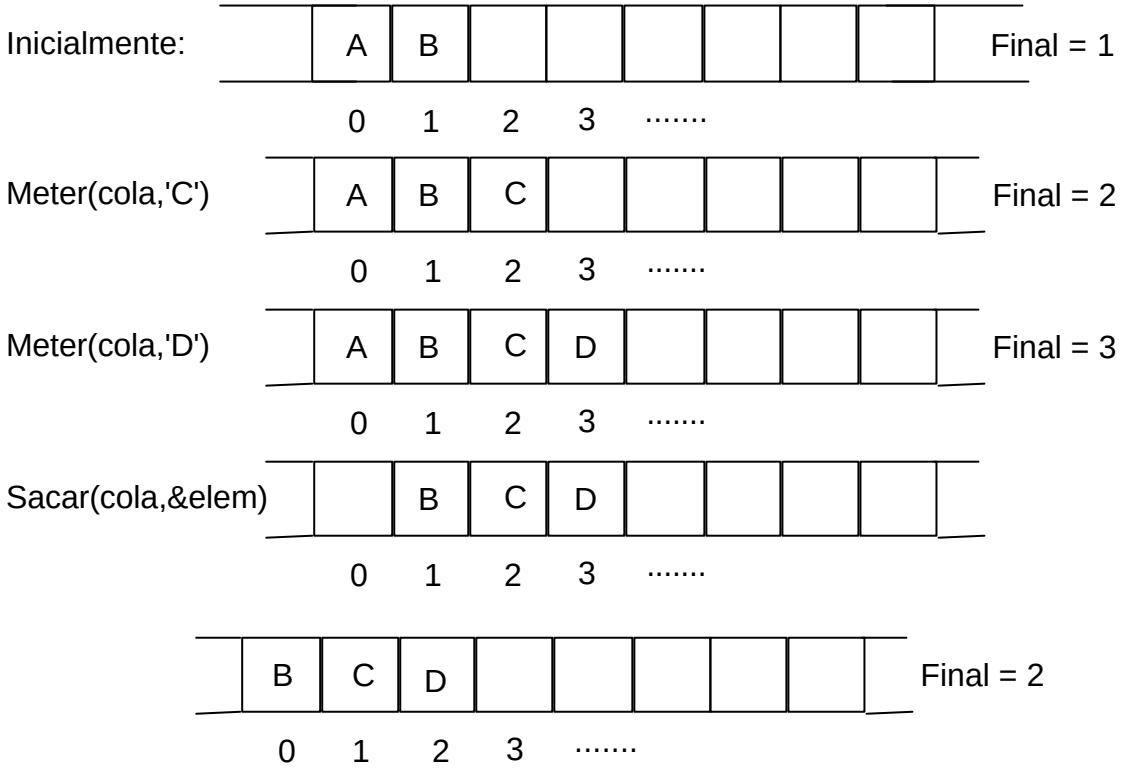
```
TipoCola Crear();
/* Crea y devuelve una cola vacía */
int ColaVacía(Tipocola Cola);
/* Devuelve 1 sólo si "cola" está vacía */
int ColaLlena(TipoCola Cola);
/* Devuelve 1 sólo si "cola" está llena */
int Sacar(Tipocola Cola,  TipoElemento *elem);
/* Saca el siguiente elemento del frente y lo pone en "elem" */
int Meter(TipoCola Cola, TipoElemento elem);
/* Mete "elem" al final de la cola */
```

- ¡Ojo con las precondiciones de las operaciones `Meter` y `Sacar`!:
  a) No se puede meter más elementos en una cola llena.
  b) No se puede sacar elementos de un cola vacía.
  Pueden tenerse en cuenta ANTES de cada llamada a `Meter` y `Sacar`, o en los propios procedimientos `Meter` y `Sacar`.

Implementación

Con un array de frente fijo

- La cola es un array de tamaño `MaxCola` cuyo frente coincide con la primera posición de éste y final inicialmente vale -1. Para añadir elementos simplemente variamos el final. Pero cada vez que sacamos un elemento del frente (de la 1ª posición), tenemos que desplazar el resto una posición en el array para hacer coincidir de nuevo el frente de la cola con la primera posición del array. Así frente siempre vale la primera posición y final indica la última posición ocupada del array. Ejemplo:



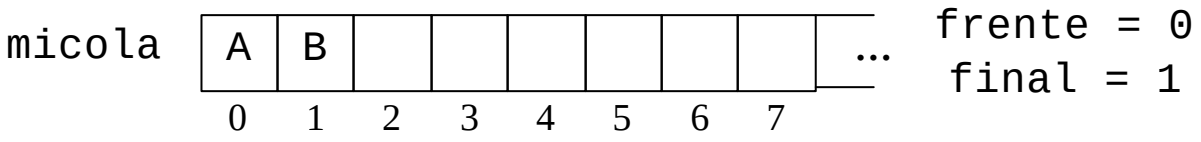
- Ventaja: simplicidad en las operaciones `Crear`, `ColaVacía`, `ColaLlena` y `Meter`.
- Desventajas:

  a) Las propias de las estructuras estáticas: desaprovechamiento de la memoria.

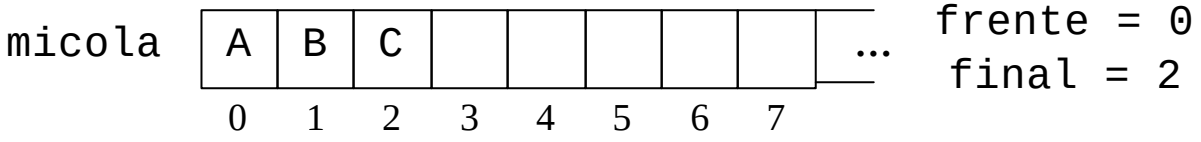
  b) Si la cola va a almacenar muchos elementos o elementos grandes, la operación `Sacar` resulta muy costosa.

Con un array de frente móvil

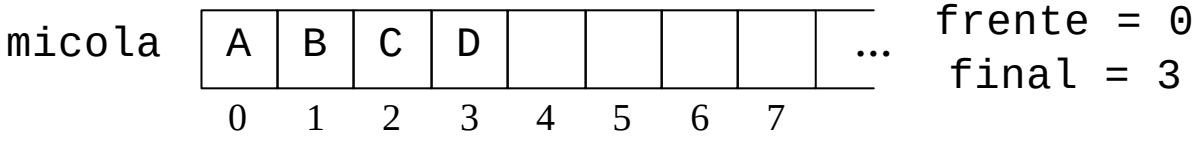
- Llevamos un índice para el frente igual que para el final y permitimos que ambos fluctúen circularmente por un array de rango [1..MaxCola]:



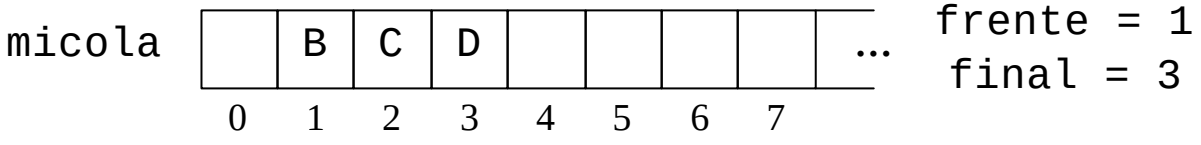

```
Meter(micola, 'C')
```



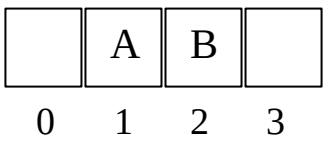

```
Meter(micola, 'D')
```



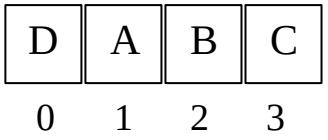

```
Sacar(micola, elem)
```



- Ventaja: la operación Sacar es más sencilla y menos costosa.
- Desventaja: hay que tratar el array como una estructura circular, lo que origina el problema de que no podemos diferenciar la situación de cola vacía y cola llena, ya que en ambos casos coincidirían los valores de final y frente. Para comprobar que la lista está vacía se podría utilizar la condición *if (final==frente-1)*, pero esta condición se cumpliría igualmente en el caso en que la lista estuviese llena si se ha dado la vuelta por el otro extremo al ser la lista una lista circular.

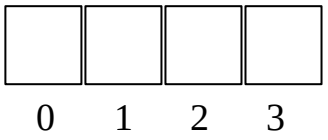


A continuación inserto C y D, con lo que la lista está llena:




```
frente = 1
final = 0
```

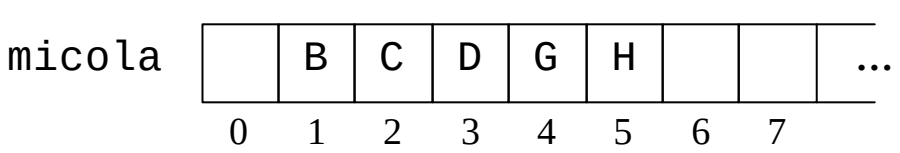
Esto es indistinguible de tener, inicialmente, la lista anterior, y borrar todos los elementos, con lo que estaría la lista vacía y, nuevamente:



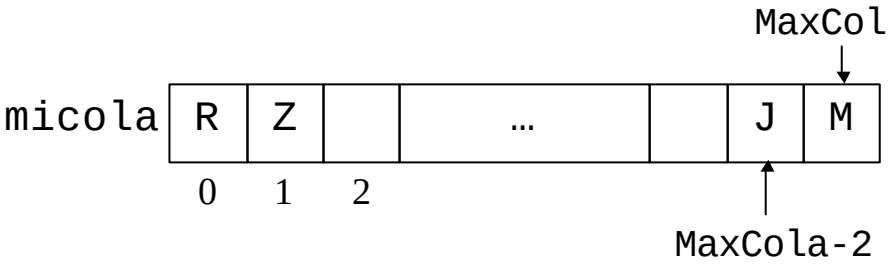

```
frente = 1
final = 0
```

Para solucionar este problema tenemos dos posibles implementaciones:

a) Disponer de otra variable, además de `frente` y de `final`, que lleve la cuenta de los elementos de la cola. Si esta variable es cero, la cola está vacía, y si es `Maxcola`, la cola está llena. Ejemplos:

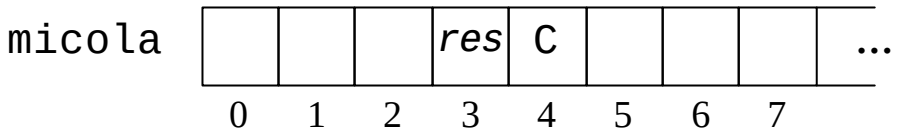



```
frente = 1
final = 5
cuenta = 5
```



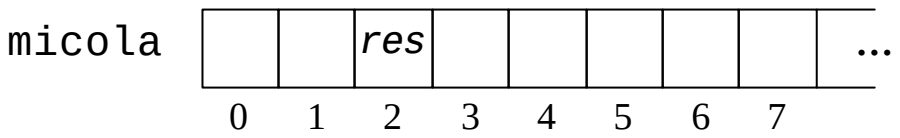

```
frente=MaxCola-2
final = 1
cuenta = 4
```

b) Hacer que `frente` apunte a la casilla del array que precede a la del elemento frente de la cola, en vez de a la del propio elemento frente, la cual tendrá un valor ***reservado***[*] que también fluctúa por el array:




```
frente = 3
final = 4
```

Así, si `frente` es igual a `final`, ello significa que la cola está vacía[†]:

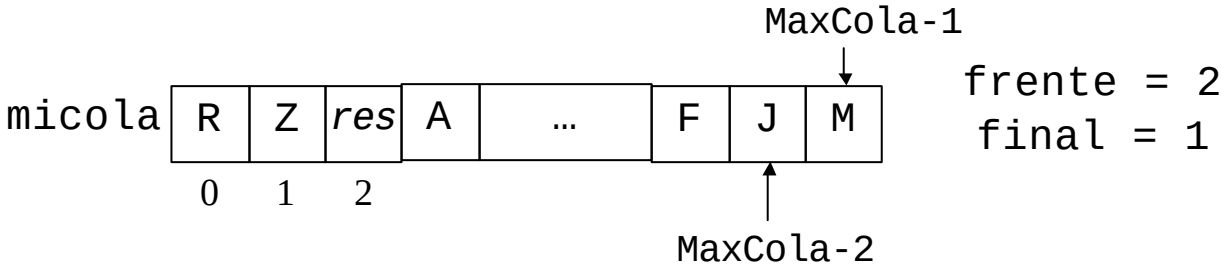



```
frente = 2
final = 2
```

[*] ¿Cuál será su posición inicial? -> en 0. Inicialmente, *frente* y *final* valen 0 para indicar que la lista está vacía

[†] ¿Cuánto valdrá `final` inicialmente? -> 0, al igual que *frente*

La cola está llena si el siguiente espacio disponible (`final+1`) está ocupado por el valor especial ***reservado*** e indicado por `frente`:



- La implementación con la primera opción usará los tipos:

```
/* TIPOS */

typedef int TipoElemento /* U otro cualquiera */
typedef struct{
      TipoElemento elementos[MaxCola];
      unsigned frente; /* Frente: [0..MaxCola-1] */
      int final; /* Final: [0..MaxCola-1] */
      unsigned cuenta; /* Cuenta: [0..MaxCola] */
}TipoCola;
```

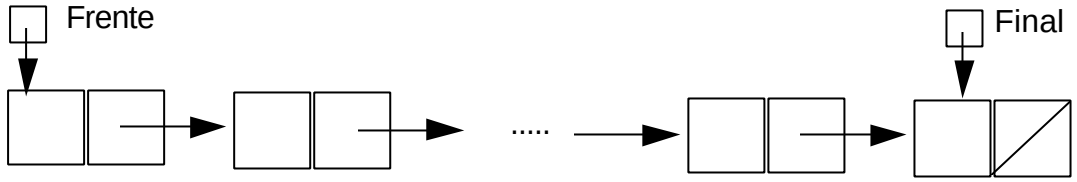
- Mientras que la segunda opción utilizará:

```
/* TIPOS */

typedef int TipoElemento /* U otro cualquiera */
typedef struct{
      TipoElemento elementos[MaxCola];
      unsigned frente; /* Frente: [0..MaxCola-1] */
      int final; /* Final: [0..MaxCola-1] */
}TipoCola;
```

Utilizando listas enlazadas con punteros

- Dos variables de tipo puntero, `frente` y `final`, van a apuntar a los nodos que contienen los elementos frente y final respectivamente.



- Para sacar elementos de la cola podemos utilizar un algoritmo similar al de borrar el primer nodo de una lista, considerando que `frente` apunta al mismo.
- Para añadir un elemento a la cola realizamos tres pasos:
  1) Creamos un nuevo nodo para el elemento a añadir.
  2) Insertamos el nuevo nodo al final de la cola.
  3) Actualizamos `final`.
- Los tipos necesarios serán:

```
/* TIPOS */

typedef int TipoElemento; /* O cualquiera */

typedef                struct                 nodo{
    TipoElemento                             valor;
    struct                nodo                *sig;
}TipoNodo;

typedef TipoNodo *TipoPuntero;

typedef                                      struct{
    TipoPuntero                              frente;
    TipoPuntero                               final;
}TipoCola;
```

- Ventaja: aprovechamiento de la memoria por el uso de punteros.
- Desventajas: las propias de punteros.

TAD **Cola** (* representación con array circular sin cuenta *)

```
#include <stdio.h>
const int MaxCola = 100; /* tamanyo maximo de la cola */
/* TIPOS */
typedef int TipoElemento; /* U otro cualquiera */
typedef struct{
      TipoElemento elementos[100];
      unsigned int frente; /* Frente: [0..MaxCola-1] */
      unsigned int final; /*  Final:  [0..MaxCola-1] */
}TipoCola;

TipoCola Crear(void);
int ColaVacia(TipoCola cola);
int ColaLlena(TipoCola cola);
int Sacar(TipoCola *cola, TipoElemento *elem);
int Meter(TipoCola *cola, TipoElemento elem);

TipoCola Crear(){
  /* Crea una cola vacía */
  TipoCola cola;

  cola.frente = MaxCola-1;   /* precede al primero*/
  cola.final  = MaxCola-1;   /* vacia */
  return(cola);
}

int ColaVacia(TipoCola cola){
  return(cola.final == cola.frente);
}

int ColaLlena(TipoCola cola){
  if (cola.final==MaxCola-1)
      return(!cola.frente);
  else
      return(cola.final+1 == cola.frente);
}

int Sacar(TipoCola *cola,TipoElemento *elem){
/* Saca un elemento de la cola si no está vacía. Sólo si está
   vacía ANTES de la operación, se devuelve 1, si no, 0" */

  int esta_vacia;
```

```
  esta_vacia = ColaVacia(*cola);
  if (!esta_vacia){
      if (cola->frente==MaxCola-1) /* Al final */
          cola->frente=0;
      else
          cola->frente++;
      *elem = cola->elementos[cola->frente];
  }
  return(esta_vacia);
}

int Meter (TipoCola *cola,TipoElemento elem){
/* Mete un elemento en la cola si no está llena. Sólo si está
   llena ANTES de la operación, se devuelve 1, si no, 0*/

  int esta_llena;
  esta_llena=ColaLlena(*cola);

  if (!esta_llena){
      if (cola->final==MaxCola-1) /* Al final */
          cola->final=0;
      else
          cola->final++;
      cola->elementos[cola->final] = elem;
  }
  return(esta_llena);
}

void main(){
  TipoCola micola;
  int estado=0;
  int i;
  TipoElemento mielem;

  micola=Crear();

  for (i=1; i<=10 && !estado; i++)
      estado=Meter(&micola,i);

  for (i=1; i<=5 && !estado; i++){
      estado=Sacar(&micola,&mielem);
      printf("%d ", mielem);
  }
```

```
}
```

Utilización

- Ejemplo: reconocedor del lenguaje:

{`w.w` / `w` es una cadena sin el carácter '`.`'}

```
void Lenguaje_Mitades_Iguales(){
/* reconocedor del lenguaje de cadenas de caracteres
   con dos mitades iguales (no simétricas) separadas
   por un punto. Cambiaremos TipoElemento a char */
  TipoElemento c1, c2;
  TipoCola cola;
  int bien, llena, vacia;

  cola = Crear();
  llena=LeerCola(&cola);
  fflush(stdin);
  if (llena)
     printf("\nError, la cola se lleno");
  else{
     c1=getchar();
     fflush(stdin);
     bien = 1;
     while (bien && c1!='f'){
        vacia = Sacar(&cola, &c2);
        if (vacia || c1 != c2)
           bien = 0;
        else{
           c1=getchar();
           fflush(stdin);
        }
     }
     if (bien && ColaVacia(cola))
        printf("\nPertenece al lenguaje.");
     else
        printf("\nNo pertenece al lenguaje.");
  }
```

```
}

int LeerCola(TipoCola *cola){
  char c;
  int llena;

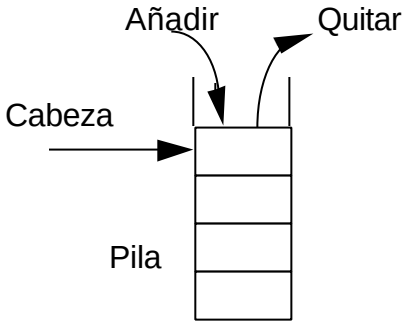
  c=getchar();
  while (!ColaLlena(*cola) && c!='.'){
      llena=Meter(cola, c);
      c=getchar();
  }
  return(c != '.');
}
```

4. Pilas

Especificación del TAD `Pila`

`TipoPila`

Colección de elementos homogéneos (del mismo tipo: `TipoElemento`*) ordenados cronológicamente (por orden de inserción) y en el que sólo se pueden añadir y extraer elementos por el mismo extremo, la cabeza. Es una estructura LIFO (Last In First Out):*



operaciones

```
TipoPila Crear(void);  /* vacía */
/* Crea una pila vacía */
int PilaVacia(TipoPila pila);
/* Comprueba si la pila está vacía */
int PilaLlena(TipoPila pila);
/* Comprueba si la pila está llena */
void Sacar(TipoPila *pila,TipoElemento *elem);
/* Saca un elemento. No comprueba antes si la pila está vacía
*/
void Meter(TipoPila pila, TipoElemento elem);
/* Mete un elemento en la pila. No comprueba si está llena. */
```

Implementación

Con un array

- Podemos poner los elementos de forma secuencial en el array, colocando el primer elemento en la primera posición, el segundo en la segunda posición, y así sucesivamente. La última posición utilizada será la cabeza, que será la única posición a través de la cual se podrá acceder al array (aunque el array permita el acceso a cualquiera de sus elementos). Para almacenar el valor de la cabeza utilizamos una variable del tipo índice del array (normalmente entera):

```
const int MaxPila = 100; /* Dimension
estimada */
/* TIPOS */
   typedef int TipoElemento; /* cualquiera */
   typedef struct{
      int Cabeza; /* Indice */
      TipoElemento elementos[100];
   }TipoPila;
```

- Si el tipo base de los elementos de la pila coincide con el tipo índice o es compatible, podemos utilizar una posición del propio array (`pila[0]` por ejemplo), para guardar el valor de la cabeza:

```
const MaxPila = 100;
const Cabeza = 0; /* posición con el índice
                     de la cabeza */

/* TIPOS */
   TipoElemento TipoPila[100];
```

- ¡Ojo con las precondiciones de las operaciones `Meter` y `Sacar`!:

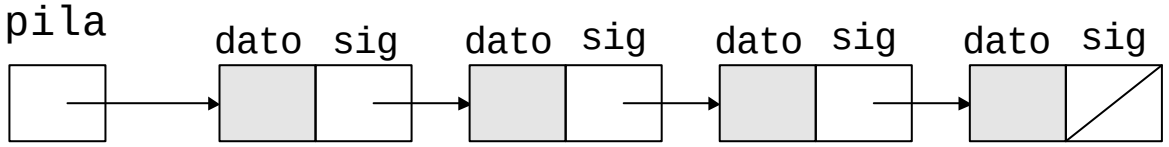
  a) No se puede meter más elementos en una pila llena.

  b) No se puede sacar elementos de un pila vacía.

  Pueden tenerse en cuenta ANTES de cada llamada a `Meter` y `Sacar`, o preferiblemente en los propios procedimientos `Meter` y `Sacar`.

Como una lista enlazada de punteros

- Los nodos de la pila están enlazados con punteros, de forma que se va reservando/liberando memoria según se necesita (por lo que no es necesario implementar la operación `PilaLlena`). El comienzo de la lista enlazada se trata como si fuera la cabeza de la pila: todas las operaciones se realizan al principio de la lista.



- Los tipos necesarios serían:

```
/* TIPOS */
   typedef int TipoElemento; /*o cualquiera */
   typedef struct nodo{
      TipoElemento dato;
      struct nodo *sig;
   }TipoNodo;
   typedef TipoNodo *TipoPuntero;
   typedef TipoPuntero TipoPila;
```

- La operación `Meter` se corresponderá con la de insertar un nodo al principio de la lista enlazada y la de `Sacar` con la de recuperar el primer nodo de la lista enlazada y eliminarlo después.

Utilización

- A este nivel se dispone de la idea abstracta de la estructura de datos `TipoPila` y de las operaciones que se pueden utilizar para manejarla, pero sin necesidad de saber cómo están implementadas ni cuál es la estructura con que se ha implementado la pila.
- Ejemplo: evaluación de una expresión en notación postfija. Precondiciones:
  1) La expresión postfija es correcta y está en un array de caracteres.
  2) Los operadores son `+`, `-`, `*` y `/`, todos binarios.
  3) Los operandos son letras mayúsculas.

```
(* Importar el TAD Pila *)
TIPOS
  TipoArray = ARRAY [1..20] DE $
  TipoElemento = Z

  SUBvoid Operando(c: $): Z
  (* cada letra se sustituye por un valor entero *)
  VARIABLES
     res:Z

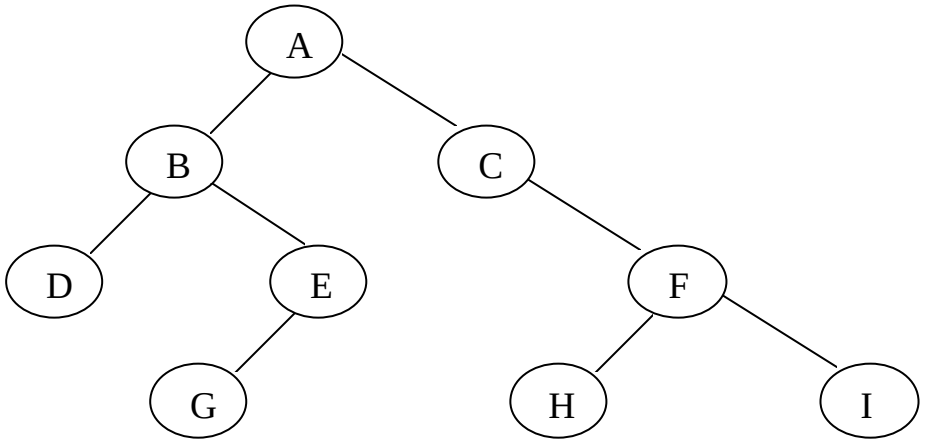
     CASO c SEA
        'A' : res = 5 (* por ejemplo *)
        'B' : res = 7 (* por ejemplo *)
        'C' : res = -1   (* por ejemplo *)
        'D' : res = 11   (* por ejemplo *)
        ...
     }CASO
     return res
  }
```

```
void Evaluar_Postfija(exp:TipoArray; ultimo:Z): Z
VARIABLES
  pila: TipoPila          (* TAD Pila *)
  i, op1, op2, result: Z  (* variables temporales *)
  c: $                    (* carácter actual *)
  llena: B  (* indicador de pila llena prematuramente *)

  pila = Crear()
  llena = FALSO
  i = 1
   MIENTRAS ((i <= ultimo) Y (NO llena)) HACER
      c = exp[i]
      SI (c='*') O (c='/') O (c='+') O (c='-') ENTONCES
         Sacar(pila, op2)
         Sacar(pila, op1)
         CASO  c SEA
            '+' : Meter(pila, op1+op2)
          | '-' : Meter(pila, op1-op2)
          | '*' : Meter(pila, op1*op2)
          | '/' : Meter(pila, op1/op2)
         }CASO
      SINO
         SI PilaLlena(pila) ENTONCES
            llena = CIERTO
         SINO
            Meter(pila, Operando(c))
         }SI
      }SI
      INC(i)
   }MIENTRAS
   SI llena ENTONCES
      Escribir("Error, la pila se llenó")
   SINO
      Sacar(pila,result)
      return result
   }SI
}
```

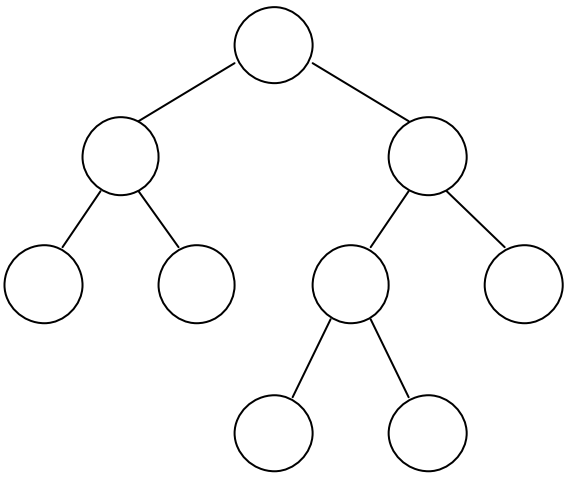
5. Arboles

- Un *árbol binario* es un conjunto finito de elementos que está vacío o partido en tres subconjuntos disjuntos. El primer subconjunto contiene un único elemento llamado *raíz* del árbol binario. Los otros dos subconjuntos son a su vez árboles binarios, llamados *subárboles izquierdo* y *derecho* del árbol binario original. Nótese que un subárbol izquierdo o derecho puede estar vacío. Cada elemento de un árbol binario se denomina *nodo*. Si un subárbol o la raíz contiene a su vez dos subárboles vacíos, es un nodo *hoja*.
- Método convencional para representar gráficamente un árbol binario:



- Relaciones de *parentesco*. Ejemplos:
  - `C` es *padre* de `F`
  - `C` es *antecesor* de `I`, de `F`, y de `H`
  - `E` es el *hijo (derecho)* de `B`
  - `G` es *descendiente (izquierdo)* de `E`
  - `G` es *descendiente (derecho)* de `B`
  - `D` y `E` son *hermanos*
  - `E` y `F` NO son hermanos
- En general, podemos decir que un *árbol* es conjunto finito de elementos que está vacío o está partido en varios subconjuntos disjuntos. El primer subconjunto contiene un único elemento llamado *raíz* del árbol. Los otros subconjuntos son a su vez árboles.

- *Arbol estrictamente binario* es aquél cuyos nodos contienen subárboles simultáneamente vacíos (nodos hoja) o simultáneamente llenos (nodos no hoja):



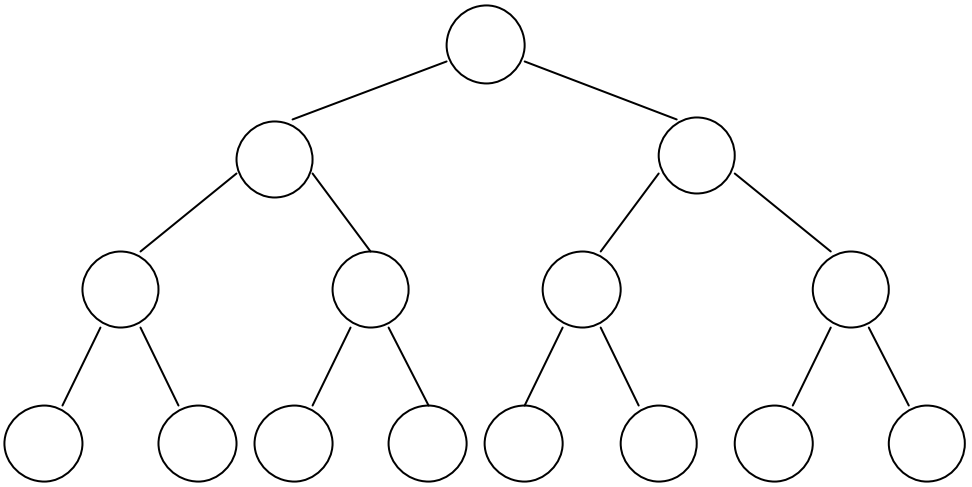
El *nivel* de la raíz es el nivel 0, y el del resto es `1+nivel(padre)`.

La *profundidad* es el nivel máximo.

Si hay `m` nodos en el nivel `L`, habrá `(2*m)` nodos como máximo en el nivel `L+1`.

En un nivel `L` hay 2^L nodos como máximo.

- Un *árbol binario completo de profundidad* `d` es aquél que tiene 2^L nodos en el nivel `L`, o lo que es lo mismo, todas las hojas tienen nivel `d`. El número total de nodos es $2^0+2^1+2^2+...+2^d = 2^{d+1}-1$:



Especificación del TAD *Arbol binario*

`TipoABin`

Estructura de elementos homogéneos (del mismo tipo: `TipoElemento`*) con una relación JERARQUICA establecida entre ellos a modo de árbol binario.*

operaciones

```
CrearVacío(): TipoABin
(* Crea un árbol vacío *)
CrearRaíz(elem: TipoElemento): TipoABin
(* Crea un árbol de un único elemento *)
ArbolVacío(árbol: TipoABin): B
(* Comprueba si "árbol" está vacío *)
AsignarIzq(VAR árbol: TipoABin; izq: TipoABin)
(* Establece "izq" como subárbol izq. de "árbol" *)
AsignarDer(VAR árbol: TipoABin; der: TipoABin)
(* Establece "izq" como subárbol izq. de "árbol" *)
Info(árbol: TipoABin): TipoElemento
(* Devuelve el nodo raíz de "árbol" *)
Izq(árbol: TipoABin): TipoABin
(* Devuelve el subárbol izquierdo de "árbol" *)
Der(árbol: TipoABin): TipoABin
(* Devuelve el subárbol derecho de "árbol" *)
BorrarRaíz(VAR árbol, izq, der: TipoABin)
(* Devuelve en "izq" y "der" los subárboles de "árbol" *)
Imprimir(árbol: TipoABin)
(* Imprime en pantalla el contenido completo de "árbol" *)
```

- Obsérvese que no se define una operación de inserción de elementos ya que el punto de inserción puede variar de una aplicación a otra, razón por la cual se deja al usuario del TAD que defina su propio `Insertar()`.
- En `Imprimir()` habría que definir de alguna manera el orden de impresión.

Implementación

Con variables estáticas

- La implementación de árboles mediante variables estáticas no suele estar justificada si el lenguaje dispone de punteros. En este caso, habría que usar un array y anotar para cada nodo las posiciones (índices del array) de los subárboles izquierdo y derecho, usando una posición inexistente para indicar la ausencia de hijos. El tratamiento sería más complejo.

Con punteros

- Los punteros son los tipos de datos más apropiados para implementar estructuras de árboles. Un `TipoABin` fácilmente se puede definir como:

```
TIPOS
   TipoElemento = (* Cualquiera *)
   TipoABin = PUNTERO A Nodo
   Nodo = REGISTRO DE
      dato: TipoElemento
      izq, der: TipoABin
   }REGISTRO
```

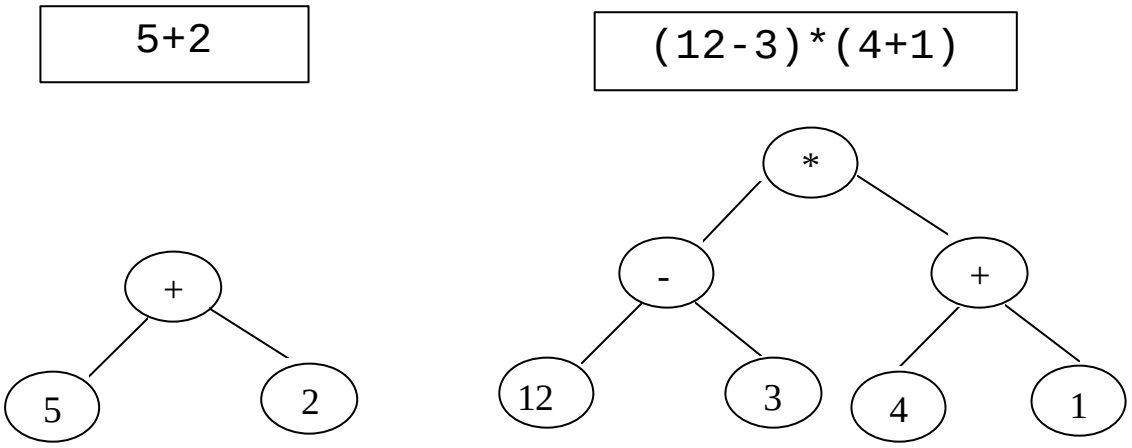
- Algunos algoritmos pueden ser iterativos, otros sólo pueden implementarse con la recursividad. Ejemplo: eliminar un árbol completo:

```
void Eliminar( árbol: TipoABin)

   SI NO ArbolVacío(árbol) ENTONCES
     SI NO ArbolVacío(árbol->izq) ENTONCES
       Eliminar(árbol->izq)
     }SI
     SI NO ArbolVacío(árbol->der) ENTONCES
       Eliminar(árbol->der)
     }SI
     free(árbol)
   }SI
}
```

Utilización

- Un árbol binario es una estructura de datos muy útil cuando se deben tomar decisiones de "dos caminos" en cada punto de un determinado proceso.
- También es muy utilizado en aplicaciones relacionadas con expresiones aritméticas binarias. Estas se pueden almacenar en un árbol binario de forma que los nodos no-hoja contengan operadores y los nodos hoja los operandos sobre los que actúan. Por ejemplo:



- Un algoritmo recursivo para evaluar las expresiones aritméticas así almacenadas, suponiendo el árbol ya creado con la expresión almacenada y correcta, daría como valor del árbol:

```
SI consta de un solo nodo ENTONCES
   return (valor que contiene el nodo)
SINO
   return (Operando1 Operador Operando2)
   (* Operando1 y Operando2 implicarían sendas
      llamadas recursivas *)
}SI
```

donde

`Operador` es `+`, `-`, `*`, `/`

`Operando1` es el valor del subárbol izquierdo

`Operando2` es el valor del subárbol derecho

- Refinando y haciendo uso de registro variantes:

```
(* Importar TAD ArbolBinario *)
TIPOS
    TipoInfo = (OPERADOR, OPERANDO)
    TipoElemento = REGISTRO DE
        CASO contenido: TipoInfo SEA
            OPERADOR: oper: $
            OPERANDO: val: R
        }CASO
    }REGISTRO

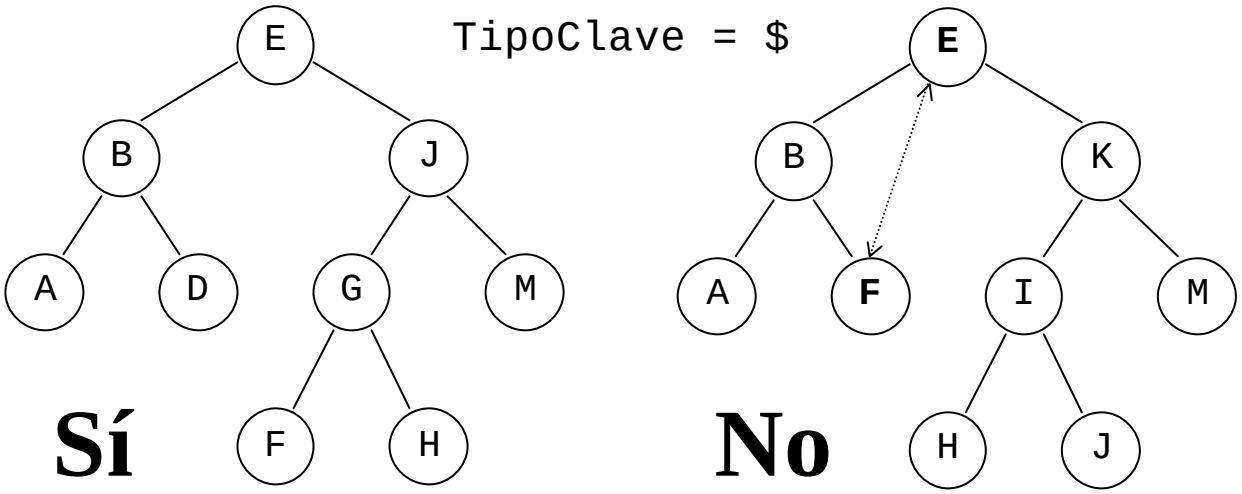
void Eval(árbol: TipoABin): R
(* Evalúa y da un valor de tipo real a “árbol”. Si es
 vacío o el operador es inválido, se devuelve cero *)
VARIABLES
    dato: TipoElemento
    res: R

    dato = Info(arbol)
    SI (dato.contenido = OPERANDO) ENTONCES
        res = dato.val
    SINO (* OPERADOR *)
        CASO dato.oper SEA
        '+':res = Eval(Izq(árbol))+Eval(Der(árbol))
        '-':res = Eval(Izq(árbol))-Eval(Der(árbol))
        '*':res = Eval(Izq(árbol))*Eval(Der(árbol))
        '/':res = Eval(Izq(árbol))/Eval(Der(árbol))
        EN OTRO CASO
            res = 0.0
        }CASO
    }SI
    return res
}
```

Especificación del TAD *Arbol binario de búsqueda*

TipoABB

Estructura de elementos homogéneos (del mismo tipo: `TipoElemento`*) con una relación JERARQUICA establecida entre ellos a modo de árbol binario en el que el subárbol izquierdo de cualquier nodo, si no está vacío, contiene valores menores, con respecto a algún campo clave de tipo* `TipoClave`*, que el valor que contiene dicho nodo, y el subárbol derecho, si no está vacío, contiene valores mayores.*:



operaciones

Las ya definidas para árboles binarios (cambiando `TipoABin` por `TipoABB`) y las siguientes propias:

```
Buscar(abb: TipoABB; clave: TipoClave;
      VAR elem: TipoElemento; VAR está: B)
(* Pone en "elem" el nodo "clave". "está" indica si está "clave"*)
Insertar(VAR abb: TipoABB; elem: TipoElemento)
(* Insertar en "abb" el nodo "elem" *)
Eliminar(VAR abb: TipoABB; clave: TipoClave)
(* Elimina el nodo "clave" de "abb" *)
Imprimir(abb: TipoABB; rec: TipoRecorrido)
(* Saca en pantalla "abb" completo, recorriéndolo según "rec"
```

- Normalmente, aunque depende de cómo se hayan realizado las inserciones, el acceso a los nodos es más eficiente que en las listas, ya que sólo se recorre desde la raíz la rama que lo contiene en base a la ordenación del árbol.
- Supondremos que no existen nodos con claves duplicadas.
- `rec` indica la forma de recorrer el árbol según se procese la raíz antes (`PREORDEN`), después (`POSTORDEN`) o entre (`ENORDEN`) el procesamiento de sus dos subárboles.

Implementación

Con variables estáticas

- La implementación de árboles ordenados mediante variables estáticas conlleva la misma problemática que la implementación de árboles sin ordenar, lo que hace que tampoco sea viable si se dispone de punteros en el lenguaje.

Con punteros

- Se realiza fácilmente definiendo:

```
TIPOS
    TipoRecorrido = (ENORDEN, PREORDEN, POSTORDEN)
    TipoClave = (* cualquier ordinal *)
    TipoElemento = REGISTRO DE
        clave: TipoClave
        ... (* resto de la información *)
    }REGISTRO
    TipoABB = PUNTERO A Nodo
    Nodo = REGISTRO DE
        izq, der: TipoABB
        elem: TipoElemento
    }REGISTRO
```

- La operación `Buscar` puede implementarse mediante un algoritmo recursivo o iterativo.
- Dado que la profundidad del árbol determina el máximo número de comparaciones en una búsqueda, la forma del árbol influye en la eficiencia, siendo deseable que la profundidad sea mínima.

- Ejemplo: algoritmo de búsqueda:

```
void Buscar_Iter(abb: TipoABB; c: TipoClave;
          { elem: TipoElemento; { está:B)

   está = FALSO
   MIENTRAS (abb <> NIL) Y (NO está) HACER
      SI (abb->elem.clave = c) ENTONCES
         está = CIERTO
         elem = abb->elem
      SINO
         SI (abb->elem.clave > c) ENTONCES
            abb = abb->izq
         SINO
            abb = abb->der
         }SI
      }SI
   }MIENTRAS
}
```

- Podemos optar por un algoritmo recursivo:

```
void Buscar_Recur(abb: TipoABB; c: TipoClave;
                  { elem: TipoElemento; { está:B)

   SI abb = NIL ENTONCESS
      está = FALSO
   SINO
      SI (abb->elem.clave = c) ENTONCES
         está = CIERTO
         elem = abb->elem
      SINO
         SI c < abb->elem.clave ENTONCES
            Buscar_Recur(abb->izq, c, elem, está)
         SINO
            Buscar_Recur(abb->der, c, elem, está)
         }SI
      }SI
   }SI
```

```
}
```

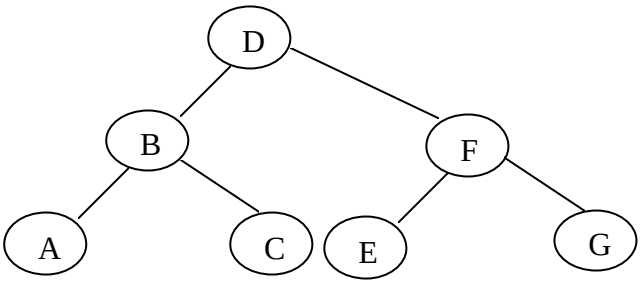
- Un algoritmo iterativo para la inserción es:

```
void Insertar( abb: TipoABB;
               elem: TipoElemento)
VARIABLES
   nuevonodo, actual, anterior: TipoABB
   clave: TipoClave

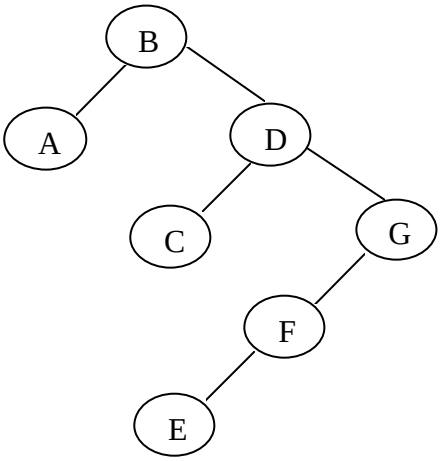
   (* crear nuevo nodo *)
   Asignar(nuevonodo)
   nuevonodo->izq = NIL
   nuevonodo->der = NIL
   nuevonodo->elem = elem
   clave = elem.clave
   actual = abb
   anterior = NIL
   (* buscar su posición de inserción *)
   MIENTRAS (actual <> NIL) HACER
      anterior = actual
      SI (actual->elem.clave > clave) ENTONCES
         actual = actual->izq
      SINO
         actual = actual->der
      }SI
   }MIENTRAS
   (* insertar el nuevo nodo *)
   SI (anterior = NIL) ENTONCES
      (* no ha entrado en el bucle: árbol vacío *)
      abb = nuevonodo
   SINO (* hemos llegado a un nodo hoja *)
      SI (clave < anterior->elem.clave) ENTONCES
         anterior->izq = nuevonodo
      SINO
         anterior->der = nuevonodo
      }SI
   }SI
}
```

- Los mismos datos, insertados en orden diferente, pueden producir árboles con formas o distribuciones de elementos muy distintas:

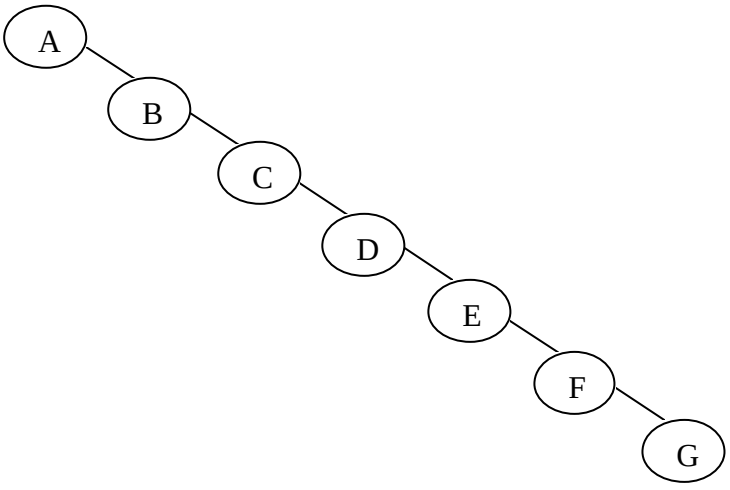
  a) Entrada: DBFACEG



  b) Entrada: BADCGFE

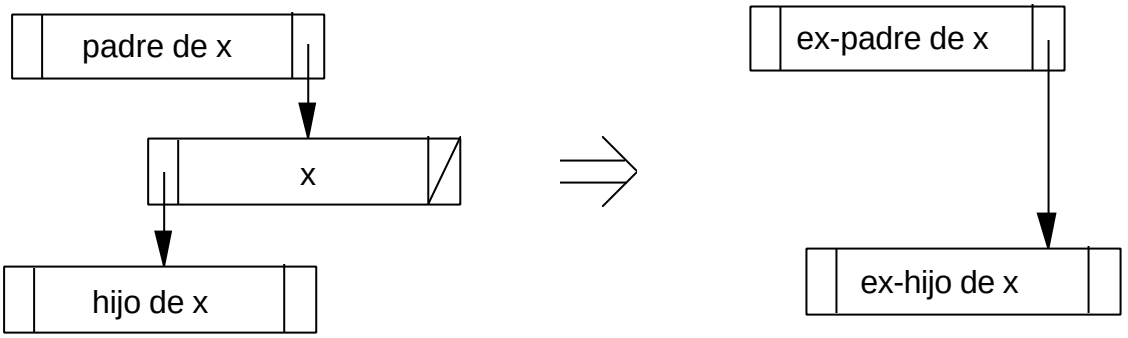


  c) Entrada: ABCDEFG

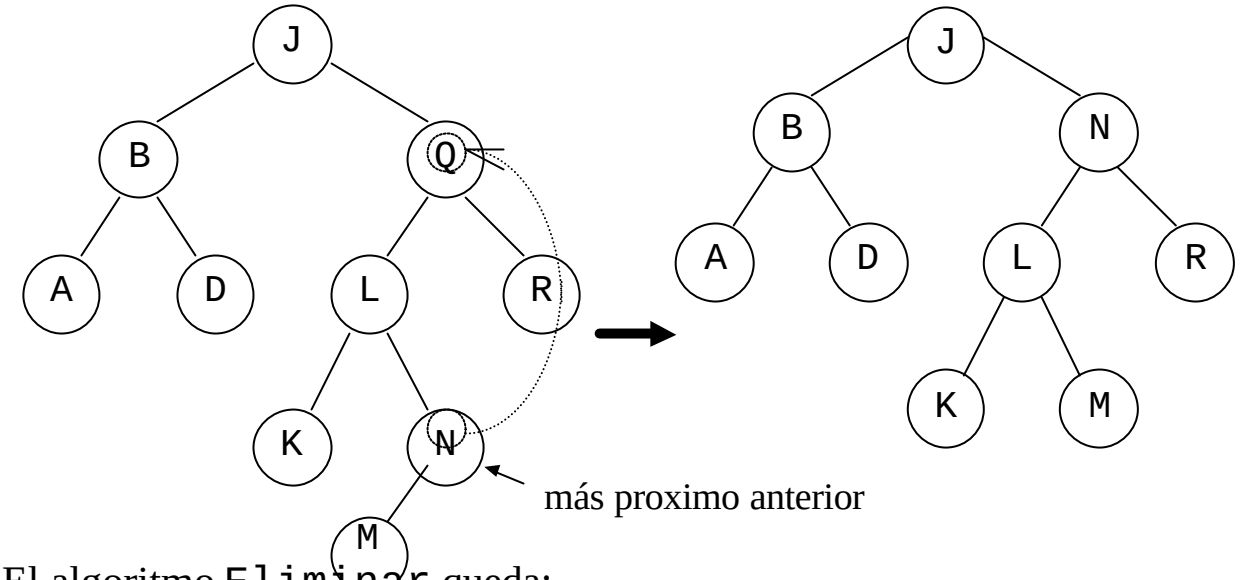


- Para `Eliminar` elementos del árbol hay que seguir los pasos siguientes:
  1) Encontrar el nodo a suprimir.
  2) Eliminarlo del árbol.

  El primer paso es sencillo, es equivalente a la operación `Buscar`, pero el segundo es complejo si el nodo no es hoja. En este caso, si el nodo no tiene hermanos, se enlaza su nodo padre con los hijos del nodo a eliminar:



  Pero si el nodo tiene hermanos, el asunto se complica porque el padre no puede apuntar a más de dos hijos. Entonces, primero hay que sustituir la información del nodo a eliminar (campo `elem` del tipo `Nodo`) por la del nodo más *próximo* (*anterior o posterior* en cuanto a la clave) a él en el árbol y después se elimina éste (antes, si fuera necesario, se enlazaría su hijo con su padre –tendría un hijo como máximo). Por ejemplo, para suprimir `Q` en el árbol siguiente (utilizando proximidad anterior):



- El algoritmo `Eliminar` queda:

```
void Eliminar( abb: TipoABB; c: TipoClave)
VARIABLES
  actual, anterior: TipoABB

  (* Paso 1: buscar nodo -suponemos que está *)
  actual = abb
  anterior = NIL
  MIENTRAS (actual->elem.clave <> c) HACER
      anterior = actual
      SI (actual->elem.clave > c) ENTONCES
          actual = actual->izq
      SINO
          actual = actual->der
      }SI
  }MIENTRAS

  (* paso 2: suprimir *)
  SI (actual = abb) ENTONCES
      (* la raíz es el elemento a eliminar *)
      SuprimirNodo(abb)
  SINO
      SI (anterior->izq = actual) ENTONCES
          SuprimirNodo(anterior->izq)
      SINO
          SuprimirNodo(anterior->der)
      }SI
  }SI
}
```

- ```
  SUBvoid SuprimirNodo( abb: TipoABB)
  VARIABLES
    temp, anterior: TipoABB

    temp = abb
    SI (abb->der = NIL) ENTONCES (* 0 ó 1 hijo *)
       abb = abb->izq
    SINO
       SI (abb->izq = NIL) ENTONCES (* 1 hijo *)
          abb = abb->der
       SINO (* 2 hijos *)
          temp = abb->izq
          anterior = abb
          (* buscar más próximo anterior *)
          MIENTRAS (temp->der <> NIL) HACER
             anterior = temp
             temp = temp->der
          }MIENTRAS
          (* antes de eliminar “temp”, su contenido
             sustituye al del nodo a eliminar *)
          abb->elem = temp->elem
          (* enlazar padre de “temp” (=“anterior”) con
            hijo izquierdo de “temp” aunque sea NIL *)
          SI (anterior = abb) ENTONCES
             anterior->izq = temp->izq
          SINO
             anterior->der = temp->izq
          }SI
       }SI
    }SI
    free(temp)
  }
  ```

- ¿Qué modificaciones habría que hacer para utilizar *proximidad posterior*?
  ```

- En la operación `Imprimir` tenemos que recorrer el árbol exhaustivamente. Para recorrer un árbol binario en general (de búsqueda o no), podemos hacerlo recursivamente de tres formas distintas:

  a) `ENORDEN`.

  1) Recorrer el subárbol izquierdo en `ENORDEN`.
  2) Procesar el valor del nodo raiz.
  3) Recorrer el subárbol derecho en `ENORDEN`.

  b) `PREORDEN`.

  1) Procesar el valor del nodo raiz.
  2) Recorrer el subárbol izquierdo en `PREORDEN`.
  3) Recorrer el subárbol derecho en `PREORDEN`.

  c) `POSTORDEN`.

  1) Recorrer el subárbol izquierdo en `POSTORDEN`.
  2) Recorrer el subárbol derecho en `POSTORDEN`.
  3) Procesar el valor del nodo raiz y procesarlo.

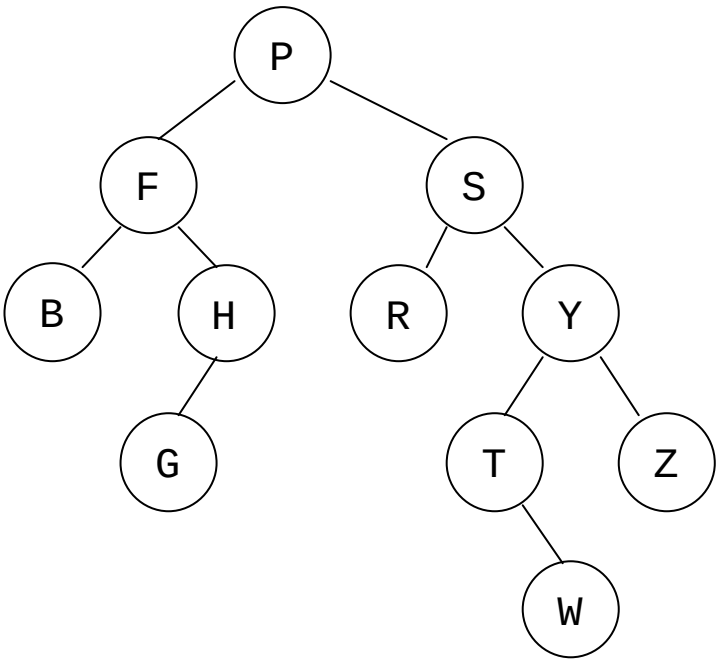
- En nuestro caso "Procesar" un nodo es `Imprimir` el elemento, a través de algún algoritmo (`ImpNodo`) que no implementamos. Además, el árbol binario es de búsqueda. Así, el código para `Imprimir` en `PostOrden` sería (los otros dos serían similares pero cambiando el orden de operaciones):

```
void Imp_PostOrden(abb: TipoABB)

    SI (abb <> NIL) ENTONCES
        Imp_PostOrden(abb->izq)
        Imp_PostOrden(abb->der)
        ImpNodo(abb->elem)
    }SI
}
```

  Y el principal sería simplemente una sentencia `CASO` que seleccionaría el algoritmo recursivo correspondiente en función del argumento `rec` de `TipoRecorrido`.

- Por ejemplo, considerando caracteres como elementos, para el árbol:



El recorrido ENORDEN mostraría: BFGHPRSTWYZ

El recorrido PREORDEN: PFBHGSRYTWZ

El recorrido POSTORDEN: BGHFRWTZYSP

Utilización

- El árbol binario de búsqueda va a ser útil en aquellas aplicaciones con operaciones de búsqueda de elementos. Como ejemplo podemos citar el procesamiento de índices de palabras de libros. Se puede usar un árbol binario de búsqueda para construir y manipular un índice de este tipo de una manera fácil y eficiente.
- Supongamos que tenemos construido un árbol binario de búsqueda que contiene un índice de palabras de un determinado libro. Cada nodo contiene la palabra, que será la clave, y un array con las páginas donde aparece esa palabra. Queremos realizar actualizaciones sobre él a partir de datos introducidos por teclado. Estos datos serán:

  a) Código operación.

  b) Palabra.

  c) Páginas (según la operación).

  donde la operación puede ser:

  a) `I`. Insertar una nueva palabra con sus páginas.

  b) `E`. Eliminar una palabra del índice.

  c) `A`. Añadir más páginas a una palabra.

  d) `F`. Finalizar la actualización.

- Los tipos necesarios serían:

```
TIPOS
    TipoClave = ARRAY [1..30] DE $
    TipoPaginas = REGISTRO DE
        total:N
        num: ARRAY [1..10] DE N
    }REGISTRO
    TipoElemento = REGISTRO DE
        clave: TipoClave
        pag: TipoPaginas
    }REGISTRO
```

- El algoritmo principal sería:

```
void ActualizarIndice( indice: TipoABB)
VARIABLES (* importar TAD Arbol Binario de Búsqueda *)
   palabra: TipoClave; elem: TipoElemento
   pag: N;   codigo: $;   encontrada: B

   Leer(codigo)
   MIENTRAS (CAP(codigo) <> 'F') HACER
      CASO CAP(codigo) SEA
      'I' :   Leer(elem.clave)
           elem.pag.total = 0
           Leer(pag)
           MIENTRAS (pag <> 0) HACER
              INC(elem.pag.total)
              elem.pag.num[elem.pag.total] = pag
              Leer(pag)
           }MIENTRAS
           Insertar(indice, elem)
      'E' :   Leer(palabra)
           Eliminar(indice, palabra)
      'A' :   Leer(palabra)
           Buscar(indice, palabra, elem, encontrada)
           SI (NO encontrada) ENTONCES
              Escribir("La palabra no está en el indice")
           SINO
              Leer(pag)
              MIENTRAS (pag <> 0) HACER
                 INC(elem.pag.total)
                 elem.pag.num[elem.pag.total] = pag
                 Leer(pag)
              }MIENTRAS
              Eliminar(indice, palabra)
              Insertar(indice, elem)
           }SI
      }CASO
      Leer(codigo)
   }MIENTRAS
```

```
}
```

Anexo

Notaciones infija, prefija y postfija

- Consideramos sólo los operadores binarios `+`, `-`, `*`, `/` y suponemos que los operandos son letras mayúsculas. Una expresión algebraica con estos operadores puede seguir tres notaciones según la posición de los operadores con respecto a los dos operandos: *infija*, *prefija* y *postfija*.
- Notación infija

  El operador binario está situado entre sus dos operandos: `A+B`

  Esta notación necesita reglas de asociatividad, precedencia y uso de paréntesis para evitar ambigüedades. Por ejemplo: `A+B*C` es ambigua. Si establecemos que `*` tiene más precedencia que `+`, deja de serlo.
- Notación prefija

  El operador binario aparece justo antes de sus dos operandos: `+AB`

  Una gramática que define una notación prefija puede ser:

```
<expr_pref> :=   <letra> |
                 <operador><expr_pref><expr_pref>
<letra> :=A | B | C | ... | Z
<operador> := + | - | * | /
```

  Ejemplos en infija y su correspondiente prefija:

  `A+(B*C)` ⟶ `+A*BC`

  `(A+B)*C` ⟶ `*+ABC`
- Notación postfija

  El operador binario aparece justo después de sus dos operandos: `AB+`

  Una gramática que genera expresiones postfijas puede ser:

```
<exp_post> := <letra> |
              <expr_post><exp_post><operador>
<letra> :=A| B ... |Z
<operador> := + | - | * | /
```

  Ejemplos en infija y su correspondiente postfija:

```
A+(B*C)        ABC*+
(A+B)*C        AB+C*
```

- Ventaja de las notaciones prefijas y postfijas: no se necesitan reglas de precedencia ni paréntesis, por lo que su procesamiento es muy simple.

Algunas implementaciones de TADs

```
    TAD Lista (* representación enlazada mixta o simulada *)
(* VISIBLE *)
TIPOS
    TipoLista

(* OCULTO *)
CONSTANTES
    NULO = 0  (* simula NIL de los punteros *)
    Max = 100 (* simula el tope de la memoria dinámica *)
TIPOS
    TipoPuntero = [0..Max] (* puntero simulado genérico *)
    TipoElemento = (* cualquiera *)
    TipoNodo = REGISTRO DE
        elem: TipoElemento
        enlace: TipoPuntero
    }REGISTRO
    Memoria = ARRAY [1..Max] DE TipoNodo
    TipoLista = TipoPuntero (* puntero simulado a la cabeza *)
    (* TipoPuntero es OCULTO pero TipoLista es VISIBLE *)
{
    (* globales pero ocultas al exterior de la implementación *)
    montón: Memoria    (* memoria dinámica *)
    lista: TipoLista   (* lista nodos usados *)
    libre: TipoLista   (* lista nodos libres *)


void Crear(): TipoLista
(* Crea la lista con todos los nodos libres inicialmente *)
VARIABLES
    ptr: TipoPuntero

    lista = NULO
    libre = 1
    PARA ptr = 1 HASTA (Max-1) HACER
        montón[ptr].enlace = ptr+1
    }PARA
    montón[Max].enlace = NULO
    return lista
}




void ListaVacía(lista: TipoLista): B
```

```
    return lista = NULO
}


void ListaLlena(lista: TipoLista): B

    return libre = NULO
}


void Imprimir(lista: TipoLista)
(* Saca por pantalla el contenido de toda la lista *)
VARIABLES
    ptr: TipoPuntero

    ptr = lista
    MIENTRAS ptr <> NULO HACER
        Escribir(montón[ptr].elem)  (* depende de TipoElemento *)
        ptr = montón[ptr].enlace
    }MIENTRAS
}


void ObtenerNodo( p: TipoPuntero) (* Simula Asignar(p) *)

    p = libre
    SI (libre <> NULO) ENTONCES
        libre = montón[libre].enlace
    }SI
}


void freeNodo( p: TipoPuntero) (* Simula free(p) *)

    montón[p].enlace = libre
    libre = p
    p = NULO
}
```

```
void Insertar( lista: TipoLista; elem: TipoElemento)
(* Se inserta al principio. Suponemos lista no ordenada *)
VARIABLES
    ptr: TipoPuntero

    ObtenerNodo(ptr)
    montón[ptr].elem = elem
    montón[ptr].enlace = lista
    lista= ptr
}


void Eliminar( lista: TipoLista; elem: TipoElemento)
{
    actual, anterior: TipoPuntero

    (* suponemos que el elemento a borrar está en la lista *)
    actual = lista
    anterio = NULO
    MIENTRAS montón[actual].elem <> elem HACER
        anterior = actual
        actual = montón[actual].enlace
    }MIENTRAS
    SI (anterior = NULO) ENTONCES
        (* el nodo a eliminar está al principio de la lista *)
        lista = montón[lista].enlace
    SINO
        montón[anterior].enlace = montón[actual].enlace
    }SI
    freeNodo(actual)
}
```

TAD **Cola** (* representación con array circular sin cuenta *)

```
(* VISIBLE *)
TIPOS
    TipoCola

(* OCULTO *)
CONSTANTES
    MaxCola = 100 (* tamaño máximo de la cola *)
```

```
TIPOS
    TipoElemento = (* cualquiera *)
    TipoCola = REGISTRO DE
        elementos: ARRAY[1..MaxCola] DE TipoElemento
        frente: [1..MaxCola]
        final: [1..MaxCola]
    }REGISTRO


void Crear(): TipoCola
(* Crea una cola vacía *)
{
    cola: TipoCola

    cola.frente = MaxCola  (* precede al 1º *)
    cola.final = MaxCola   (* vacía *)
    return cola
}


void ColaVacía(cola: TipoCola): B

    return cola.final = cola.frente
}


void ColaLlena(cola: TipoCola): B

    return (cola.final MOD MaxCola + 1) = cola.frente
    (* MOD permite tratar fácilmente el array circular *)
}


void Sacar( cola: TipoCola; { elem: TipoElemento;
               { está_vacía: B)
(* Saca un elemento de la cola si no está vacía. Sólo si está
   vacía ANTES de la operación, se pone a CIERTO "está_vacía" *)

    está_vacía = ColaVacía(cola)
    SI NO está_vacía ENTONCES
        cola.frente = cola.frente MOD MaxCola + 1
        elem = cola.elementos[cola.frente]
    }SI
```

```
}
void Meter( cola: TipoCola; { elem: TipoElemento;
                { está_llena: B)
(* Mete un elemento en la cola si no está llena. Sólo si está
   llena ANTES de la operación, se pone a CIERTO "está_llena" *)

    está_llena = ColaLlena(cola)
    SI NO está_llena ENTONCES
        cola.final = cola.final MOD MaxCola + 1
        cola.elementos[cola.final] = elem
    }SI
}
```

TAD **Pila** (* representación enlazada con punteros *)

```
(* VISIBLE *)
TIPOS
    TipoPila

(* OCULTO *)
TIPOS
    TipoElemento = (* cualquiera *)
    TipoPuntero = PUNTERO A Nodo
    Nodo = REGISTRO DE
        elem: TipoElemento
        sig: TipoPuntero
    }REGISTRO
    TipoPila = TipoPuntero
    (* TipoPuntero es OCULTO pero TipoPila es VISIBLE *)


void Crear(): TipoPila
(* Crea una pila vacía *)
{
    pila: TipoPila

    pila = NIL
    return pila
}


void PilaVacía(pila: TipoPila): B

    return pila = NIL
```

```
}

void PilaLlena(pila: TipoPila): B

    return FALSO
}



void Sacar( Pila: TipoPila; { elem: TipoElemento)
(* Saca un elemento. No comprueba antes si la pila está vacía *)
VARIABLES
    p: TipoPuntero

    elem = pila->elem
    p = pila
    pila = pila->sig
    liberar(p)
}


void Meter( Pila: TipoPila; elem: TipoElemento)
(* Mete un elemento en la pila. No comprueba si está llena. *)
{
    nodo = TipoPuntero

    Asignar(nodo)
    nodo->elem = elem
    nodo->sig = pila
    pila = nodo
}
```

TAD **Arbol Binario** (* representación enlazada con punteros *)

```
(* VISIBLE *)
TIPOS
    TipoABin
(* OCULTO *)
TIPOS
    TipoElemento = (* cualquiera *)
    TipoPuntero = PUNTERO A Nodo
    Nodo = REGISTRO DE
        elem: TipoElemento
        izq, der: TipoPuntero
    }REGISTRO
```

```
    TipoABin = TipoPuntero
    (* TipoPuntero es OCULTO pero TipoABin es VISIBLE *)
void CrearVacío(): TipoABin

    return NIL
}


void CrearRaíz(elem: TipoElemento): TipoABin
{
    árbol: TipoABin

    Asignar(árbol)
    árbol->elem = elem
    árbol->izq = NIL   (* o bien, árbol->izq = CrearVacío() *)
    árbol->der = NIL   (* o bien, árbol->der = CrearVacío() *)
    return árbol
}


void ArbolVacío(árbol: TipoABin): B

    return árbol = NIL
}


void AsignarIzq( árbol: TipoABin; izq: TipoABin)

    árbol->izq = izq
}


void AsignarDer( árbol:TipoABin; der:TipoABin)

    árbol->der = der
}


void Info(árbol:TipoABin): TipoElemento

    return árbol->elem
}
```

```
void Izq(árbol:TipoABin): TipoABin

    return árbol->izq
}


void Der(árbol:TipoABin): TipoABin

    return árbol->der
}


void BorrarRaíz( árbol,izq,der:TipoABin)

    izq = árbol->izq
    der = árbol->der
    free(árbol)
}
```