

# RESUMEN DE PROGRAMACIÓN

## Índice:

1.Introducción a C++.	2	9.TAD Cola de prioridad.	18
2.Tipos abstracto de datos.	10	10.Heaps.	19
3.Recursión.	11	11.Árboles.	21
4.TAD lista.	12	11.HASH.	23
5.TAD Pila.	14	12. TAD Multisets.	23
6.TAD Cola.	15	13.TAD Tabla.	24
7.TAD Set.	16	14.Tiempo de ejecución.	25
8.TAD Diccionario.	17	15.Multiestructuras.	27

## Acerca de C++:

- {} son el begin y end en pascal.
- Printf, imprime algo en la salida.
- Return, sirve para devolver un valor al terminar la función o procedimiento.
- Main, se escribe para hacer referencia a que es el programa principal.
- Scanf, leer valores de la entrada.

## Tipo de datos elementales:

- Char: carácter.
- Int: entero.
- Bool: booleano.
- float/double: real.

Las variables se pueden declarar en cualquier lugar.

## Operadores de comparación:

- == (igual).
- !=(distinto).
- <= (menor o igual).
- >= (mayor o igual).
- >(mayor)
- <(menor).

Se recomienda  
declarar las  
variables  
cuando se  
van a usar



## Operadores lógicos:

- `&&`(AND).
- `||`(OR).
- `!`(NOT).

## Operadores aritméticos:

- `+`(suma).
- `-`(resta).
- `*`(multiplicación).
- `/`(división).
- `%`(módulo).

➔ `++a`, incrementa el valor de `a` y retorna su valor **luego** del incremento.

➔ `a++`, incrementa el valor de `a` y retorna su valor **antes** del incremento.

➔ `Variable1 += Variable2` (`variable 1 = variable1 + variable2`)

## Estructuras de control:

- IF:

if(condición){

-----

}

(lo mismo que pascal, y en if-else).

- SWITCH(example):

switch (valor)

case 6: case 7: case 8: case 9: case 10: case 11: case 12:

printf(\*Aprobado\*);

cantidad\_aprobados++;

break;

case 3: case 4: case 5:

printf(\*Examen\*);

break;

case 0: case 1: case 2:

printf(\*Reprobado\*);

break;

default

printf(\*Valor incorrecto\*);

- WHILE.

While (condición){

-----

}

Definir un struct:

- struct variable{

tip01 nombre1;

tip02 nombre2;

tip03 nombre3;

}

variable.nombre1  maneras de acceder al campo nombre1 de variable.

Se puede definir también que nombre1,2,3 sea un puntero a algo de tipo tip01,2,3.


- struct variable{

tip01\* nombre1;

tip02\* nombre2;

tip03\* nombre3;

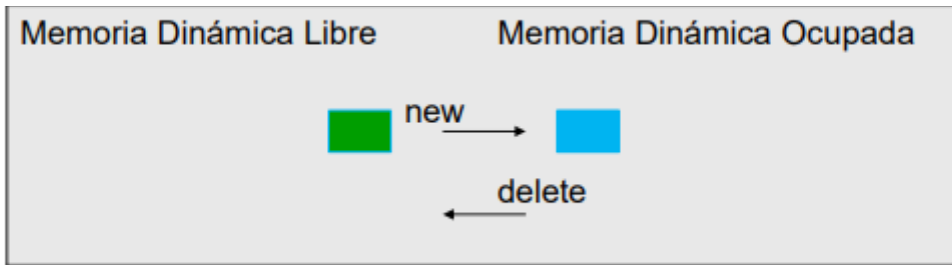
}

typedef variable\* Tvariable;  TVariable  $\rightarrow$  nombre1

## Puntero:

- `tipo1*nombre; //(nombre se declaró como un puntero a algo de tipo tipo1)`
- `new` pide memoria.

`nombre= new tipo1;`



Se hace delete  
a lo que una vez  
se le hizo new

- `delete` borra la memoria.

`delete(nombre);`

- `variable1= variable2` (igualdad de punteros, van a apuntar a lo mismo)

**CUIDADO:** `int*p4,p5` esto es equivalente a `-int*p4` (`p4` es puntero)  
`-int p5` (`p5` **NO** es puntero)

**estático:** `int arr[n]; // las celdas posibles arr[0].....arr[n-1]arr[n]`

**dinámico:**

crearlo → `int* arr= new int[n];`

borrarlo → `delete[] arr;`



## FUNCIONES:

```
tipo nombre(parámetros){
```

```
-----
```

```
}
```

tipo  $\longrightarrow$  el tipo del elemento que devuelve la función.

Nombre  $\longrightarrow$  nombre de la función.

Parámetros  $\longrightarrow$  se encuentran todos los parámetros pasados por referencia o valor.

## PROCEDIMIENTOS:

```
void nombre(parámetros){
```

```
-----
```

```
}
```

La diferencia con las funciones es que el procedimiento no devuelve ningún valor(void).

## Parámetros:

```
void nombre(tipo1 parametro1, tipo2 &parametro2){
```

```
-----
```

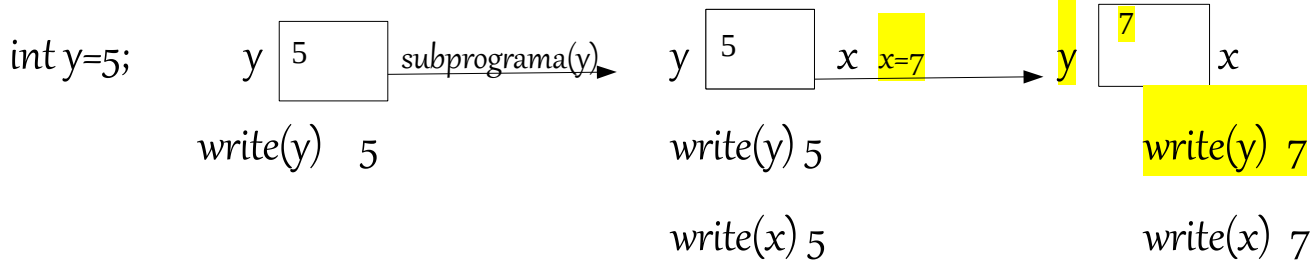
```
}
```

- parametro2 parámetro pasado por referencia.
- Parametro1 parámetro pasado por valor.

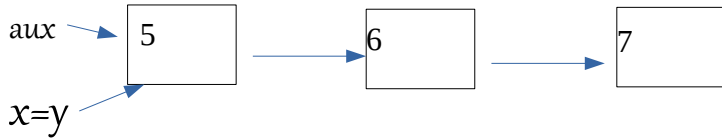
NO SE PUEDE  
DEFINIR  
FUNCIONES  
DENTRO DE  
FUNCIONES



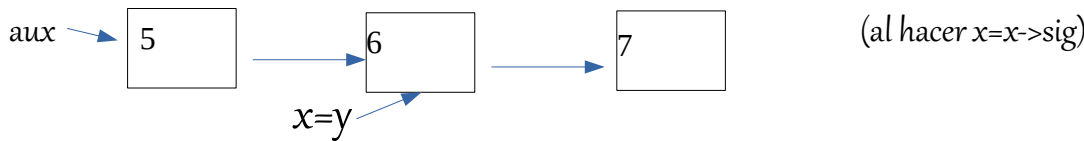
Cuando se habla de **memoria estática**, en pasaje de referencia: `subprograma(int &x)`



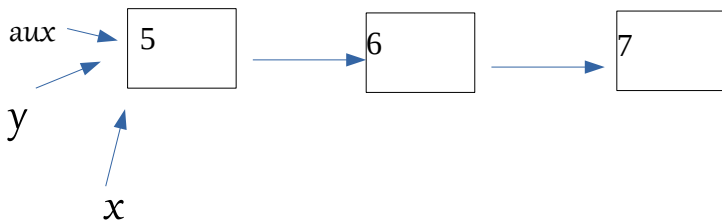
**Memoria dinámica**, en pasaje por referencia: `subprograma(int* &x)`



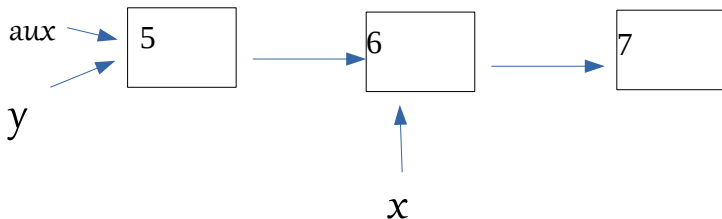
`subprograma(y)` → si imprimimos el valor de `y`, imprime 5 pero si hacemos `x=x → sig` nos va a imprimir 6(valor de `y`).



En cambio si lo **pasamos por valor** (`subprograma(int* x)`), al principio nos imprimir 5.



Pero si hacemos `x=x → sig` sig imprime 5 y.



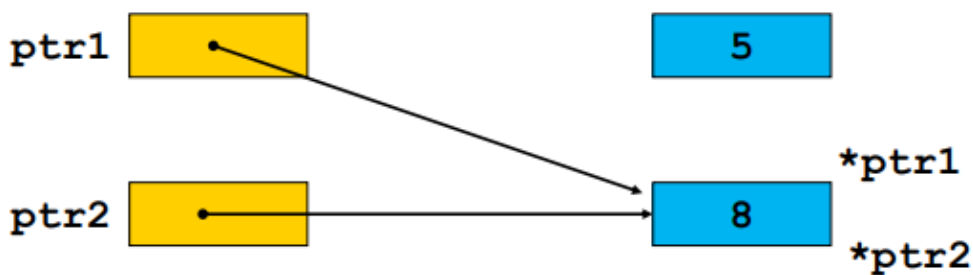
- Es decir que por referencia son un mismo puntero y por valor solo se copia la dirección de memoria (es decir dos punteros que apuntan a lo mismo).



- Es importante cuando se maneja los punteros alias el **NO** desperdiciar memoria.

La siguiente figura representa lo que sucede al hacer las siguientes instrucciones:

- `int *ptr1=5;`
- `int* ptr2=8`
- `ptr1=ptr2` (es diferente de hacer `*ptr1=*ptr2`)



#### ESPECIFICADORES:

- `%d` int
- `%c` char
- `%f` float
- `%s` `char*` → arreglo de caracteres

#### Operaciones:

- Constructoras(crear,insertar).
- Selectoras(eliminar elemento, pedir elemento).
- Predicados(Ver si es vacío,ver si es lleno).
- Destructoras(destruir).

## Tipo abstracto de datos(TAD):

¿Qué es un tipo abstracto de datos?

Es un tipo de datos, es decir, puede ser especificado como tal con precisión, pero no está dado en términos de los constructores de tipo del lenguaje.

La especificación de TAD, es cuando se introduce el mismo. Es decir se le da un nombre y se le asocia una cantidad de operaciones(subprogramas) aplicable a los elementos del tipo.

Por otro lado tenemos a la implementación de TAD, que consiste en un tipo de estructura de datos concreto que se elige como representación del TAD y las correspondientes implementaciones de los subprogramas.

La especificación es mas para el programador o quien lo vaya a usar, ya que tiene las operaciones y que hacen cada una de ellas. Solamente con la especificación, sin saber la estructura del dato, le debe alcanzar al programador para poder utilizar las operaciones.

En cambio la implementación, ademas de tener el nombre y que hacen las operaciones tiene el código de las mismas, esto es para las computadoras.

## TAD's acotados y no acotados

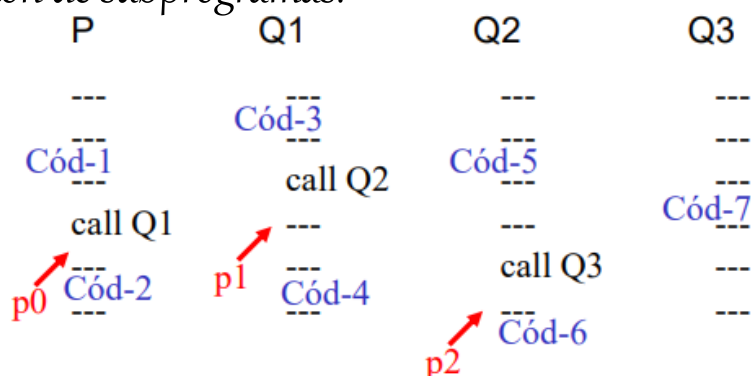
Los TAD's pueden ser acotados o no en la cantidad de elementos (hay cambios en la especificación). En general, las implementaciones estáticas refieren a una versión del TAD que es acotada (incluso en la especificación del TAD), mientras que las implementaciones dinámicas admiten una versión del TAD no acotada (que permite expresar la noción de estructuras arbitrariamente grandes).

La diferencia entre acotado y no acotado en la especificación es ver si el TAD esta lleno y al crear se pasa un parámetro para acotar la cantidad de elementos.

## RECURSIÓN:

Subprogramas hechos con recursión, son subprogramas que se invocan a si mismos.

Invocación de subprogramas:



Ejecución: Cód-1, Cód-3, Cód-5, Cód-7, Cód-6, Cód-4, Cód-2

**STACK**

p2  
p1  
p0

**PIQUE:** Para escribir el código de un subprograma con recursión es importante ANTES de escribir el código, identificar cuales son los casos BASES, el caso en el que el subprograma NO TENDRÍA EFECTO(a veces no hay) y cuál es el caso donde se llama a la RECURSIÓN.

En el caso de la recursión, es también importante identificar que se hace ANTES y DESPUÉS de llamar a la recursión. Acordarse que lo que este DESPUÉS se va a ejecutar una vez que llegue al final de la recursión.

## Punteros:

ptr1 = new tipo1; (se obtiene una dirección de memoria apropiado para almacenar algo de tipo tipo1.)

ejemplo: ptr1 = new int;

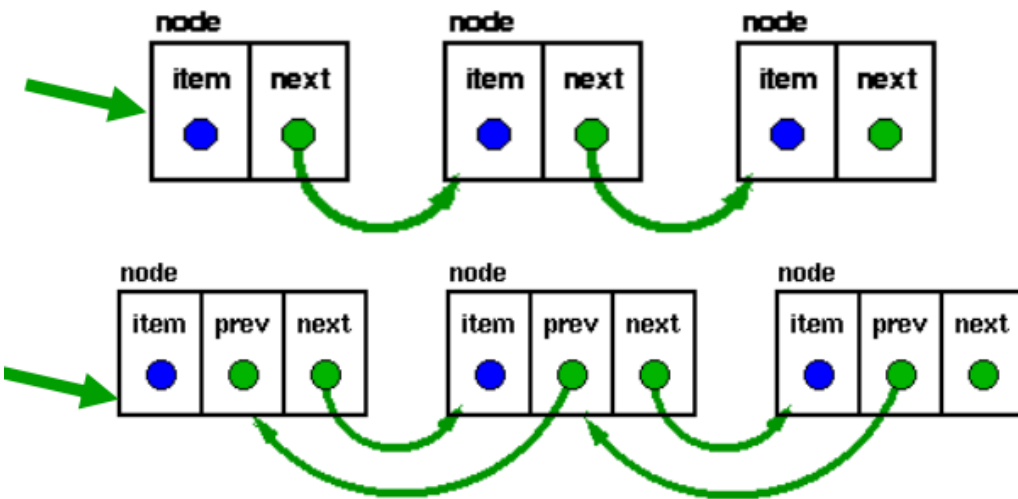
\*ptr = 5;



## TAD LISTA :

Existen Lista indizada no acotada, listas doblemente encadenadas, puede ser con punteros al inicio y al final o solo al inicio.

### Variantes de Listas



## Operaciones:

### Constructoras:

- **CrearLista:** construye una lista vacía.
- **Insertar:** dada una lista, un entero  $n$  y un elemento  $e$ , inserta  $e$  en la posición  $n$  de la lista.

### Predicados:

- **EstaVacía:** retorna true si y solo si la lista es vacía.
- **EstaDefinido:** dada una lista y un entero  $n$ , retorna true si y solo si la lista esta definida en la posición  $n$ .



Selectoras:

- Elemento: dados una lista y un entero  $n$ , retorna el elemento de la posición  $n$ . Si la lista tiene longitud  $< n$  la función no esta definida.
- Borrar: dada una lista y un  $n$ , elimina de la lista el elemento que este en la posición  $n$ . Si la posición no esta definida la operación no hace nada.

Destructora:

- Destruir: destruye la lista, liberando la memoria que esta ocupada.

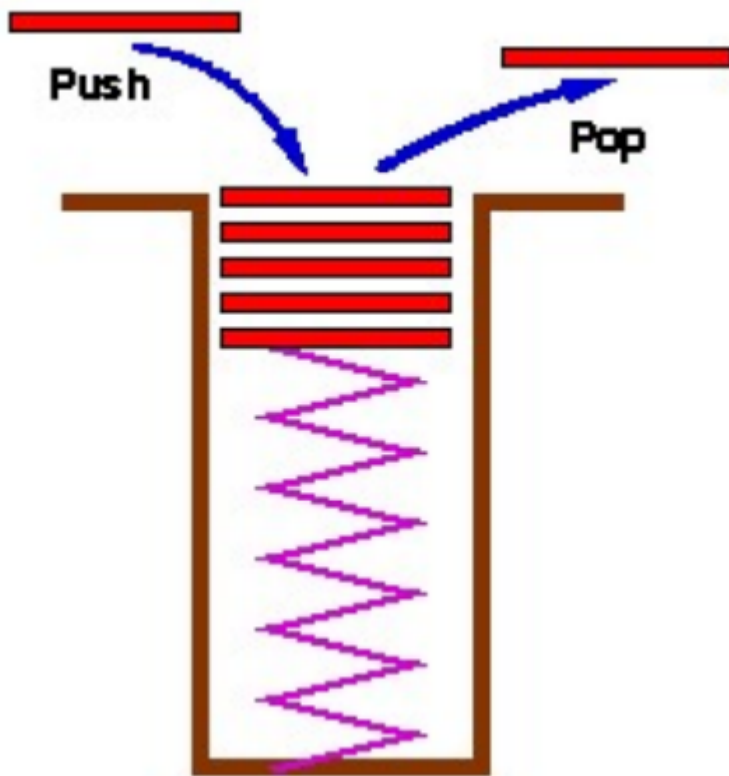
Otra funciones para el manejo de listas son las de concatenar dos listas, hacer una copia de la lista, insertar solo al final, invertir una lista, imprimir una lista, saber la cantidad de elementos en la lista.

Hay muchas especificaciones del TAD lista. Por ejemplo en insertar puede hacerlo al inicio, final, en una posición  $n$ , en la posición  $n+1$  o de manera ordenada.

Borrar puede ser el último, el primero, el de la posición  $n$  o  $n+1$ , y como en insertar también de forma ordenada.

## TAD PILA(STACK):

Una pila es una lista LIFO(Last int First out).



### Operaciones:

#### Constructoras:

- CrarPila: construye una pila vacía.
- Apilar: inserta un elemento en el tope de la pila, sino esta llena.

#### Predicados:

- EsVaciaPila: retorna true si y solo si es vacia la pila.
- EsLLenaPila: devuelve true si y solo si la pila esta llena.(Si es TAD Pila acotada)

#### Selectoras:

- Desapilar: remueve el elemento que se encuentra en el tope de la pila.
- Cima: que retorna el elemento que se encuentra en el tope de la Pila.

#### Destructora:

- DestruirPila: libera la memoria ocupada.

## TAD COLA

Una cola es una lista FIFO (First in first out).



## Operaciones:

Constructoras:

- crearCola: la operación que construye una cola vacía.
- Encolar: que inserta un elemento al final de la cola.

Selectoras:

- Front: que retorna el elemento que se encuentra en el comienzo de la cola.
- Desencolar: que borra el primer elemento de la cola.

Predicados:

- esVacía: que testea si la cola es vacía.
- EstaLLena: que testea si la cola está llena (si es una cola acotada).

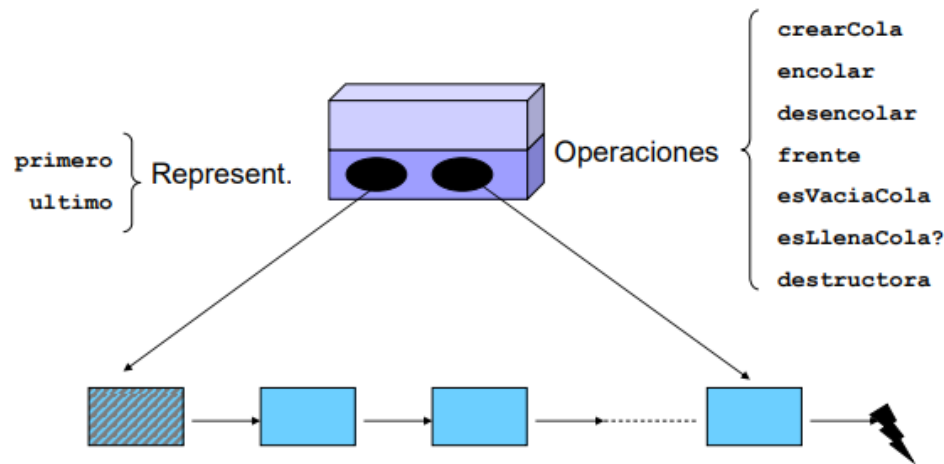
Destructoras:

- Destruir: elimina una cola y libera la memoria que ésta ocupa.

El TAD COLA PODRÍA SER ACOTADO, en ese caso se puede usar un arreglo circular con tope (esto está en el pdf de la módulo 6, pág 60 ).



## Implementación del TAD Queue (cont.)



El TAD Cola puede ayudar en árboles, al hacer un recorrido por niveles del menor al mayor, por ejemplo imprimirlos. Acuérdense que esto era al crear una cola vacía, se encolaba el árbol y antes de desencolarlo se lo guardaba en una variable. Después se imprimía la raíz y encolaba los subárboles izquierdo y derecho de la raíz del árbol, luego se repetía.

### TAD SET(conjuntos)

Un conjunto es una colección de elementos, no puede tener elementos repetidos.

Operaciones:

- Vacio c: construye el conjunto c vacío;
- Insertar x c: agrega x a c, si no estaba en el conjunto.
- EsVacio c: retorna true si y sólo si el conjunto c está vacío.
- Pertenece x c: retorna true si y sólo si x está en c.
- Borrar x c: elimina a x del conjunto c, si estaba.
- Destruir c: destruye el conjunto c, liberando su memoria.
- Operaciones para la unión, intersección y diferencia de conjuntos, entre otras.
- Otras operaciones pueden ser:

Borrar, Borrar\_max o Borrar\_min, Min o Max.

Hay dos TAD especialmente utilizados, basados en el modelo de conjuntos.

## COLAS DE PRIORIDAD

-Vacio  
-Insertar  
-esVacio  
-Borrar\_Min  
-Min

## DICCIONARIOS

-Vacio  
-Insertar  
-esVacio  
-Borrar  
-Pertenece

Para el TAD SET son válidas las siguientes implementaciones:

- LISTAS ENCADENADAS(ORDENADAS O NO).(se adecuá bien a una versión NO acotada de diccionarios y conjuntos)
- ARREGLO DE BOOLEANOS(O DE BITS).(versión acotada)
- ARREGLO CON TOPE.(Versión acotada de dicc y conj)
- ABB O AVLs.(Versión No acotada)
- TABLA DE HASH.

## TAD Diccionarios:

Es un TAD Set con las operaciones: Crear, insertar, esVacio, borrar y pertenece.

	A	B	C	D	E	F	G	H	I
1	Diccionario	Vacio	esVacio	Insertar	Borrar	Pertenece	Intersección	Unión	Diferencia
2	Arreglo de <u>bool</u>	O(n)	O(n)	O(1)	O(1)	O(1)			
3	Listas encadenas ordenadas	O(1)	O(1)	O(n)	O(n)	O(n)	O(n+m)	O(n+m)	O(n+m)
4	Listas encadenas no ordenada	O(1)	O(1)	O(n)	O(n)	O(n)	O(n*m)	O(n*m)	O(n*m)
5	ABB	O(1)	O(1)	O(n)	O(n)	O(n)			
6	AVLs	O(1)	O(1)	O(logn)	O(logn)	O(logn)			
7	HASH	O(1) peor caso	O(1) peor caso	O(1) promedio	O(1) promedio	O(1) promedio			

## TAD Cola de Prioridad:

El término “cola de prioridad” se relaciona con:

- La palabra “cola” sugiere la espera de cierto servicio por ciertas personas u objetos.
- La palabra “prioridad” sugiere que este servicio no se proporciona por medio de una disciplina “primero en llegar, primero en ser atendido” que es la base del TAD Cola, sino que cada elemento tiene una prioridad basada en “la urgencia de su necesidad” (de ser atendido).

## Operaciones:

Constructoras:

- Crear: crea una cola de prioridad vacía.
- Insertar: inserta el elemento en la cola de prioridad.

Predicados:

- EsVacio: devuelve true si y solo si la cola de prioridad es vacía.

Selectores:

- Borrar\_Min: elimina el elemento con más prioridad de la cola de prioridad.
- Min: devuelve el elemento mas prioritario.

Binary heaps es una implementación para el TAD de colas de prioridad acotadas.

## Heaps:

Los Heaps tienen 2 propiedades (al igual que los AVL):

- una propiedad de la estructura.
- una propiedad de orden del heap.

**Prop de estructura:** Es un árbol binario completo lleno, es decir todos los niveles están completos, con la posible excepción el nivel mas bajo, el cual se llena de izquierda a derecha.

**Prop de orden:** La clave de cada nodo es prioritario con respecto a las de sus descendientes.

Debido a que un árbol binario completo es tan regular, se puede almacenar en un arreglo, sin recurrir a apuntadores.

## Operaciones:

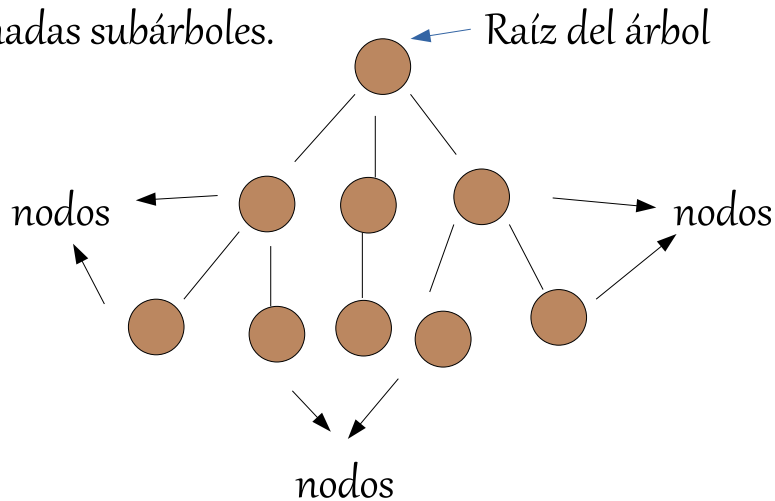
- Min:  $O(1)$ .
- crear:  $O(1)$ .
- EsVacio:  $O(1)$
- Insertar:  $O(\log n)$ .
- Borrar\_Min:  $O(\log n)$ .

Cola de prioridad extendidas: También se pueden incluir para decrementar o incrementar la prioridad o actualizarla, eliminar el nodo en determinada posición.

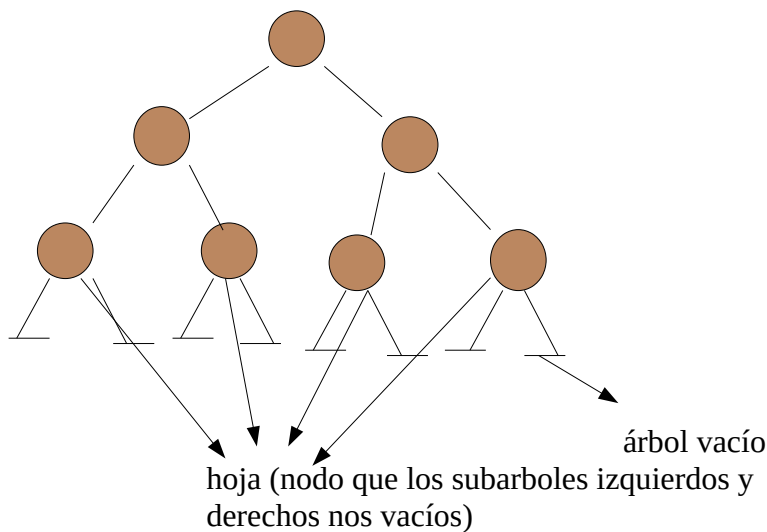


## Estructuras arborescentes:

Una estructura árbol (árbol general o finitario) con tipo base  $T$  es, o bien la estructura vacía o bien un elemento de tipo  $T$  junto con un número finito de estructuras de árbol, de tipo base  $T$ , disjuntas, llamadas subárboles.



Arboles binarios: Es un árbol general pero cada nodo tiene dos subárboles.



## Altura:

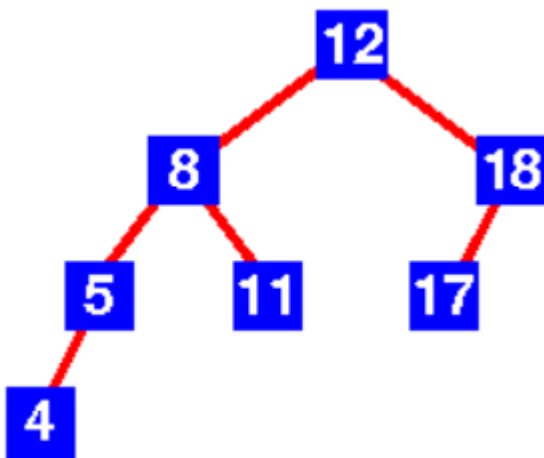
Es la cantidad de niveles que tiene, o la cantidad de nodos en el camino más largo de la raíz a una hoja. (La altura del árbol binario vacío es 0).

Para recorrer un árbol no vacío hay tres órdenes naturales, según la raíz sea visitada:

- antes que los subárboles (PreOrden - preorder)
- entre las recorridas de los subárboles (EnOrden - inorder)
- después de recorrer los subárboles (PostOrden - postorder)

### Árbol Binario de Búsqueda(ABB):

Los árboles binarios son usualmente usados para representar conjuntos de datos los cuales pueden ser representados por medio de una clave única. Los ABB cumplen la propiedad de que teniendo un nodo  $x$ , el nodo a la izquierda de  $x$  tiene una clave menor a la del nodo  $x$  y el nodo a la derecha es mayor al nodo  $x$ .



El mínimo está lo más a la izquierda y el máximo lo más a la derecha.

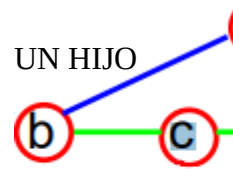
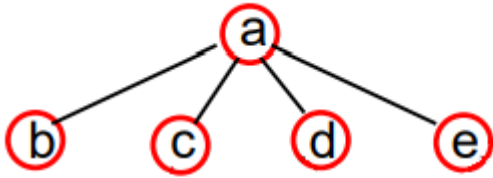
### Operaciones:

- crearABB: crea un árbol vacío y lo devuelve.
- BuscarEnABB: devuelve true si y solo si el elemento pasado por parámetro está en el ABB.
- InsertarEnABB: inserta un elemento al ABB.
- BorrarEnABB: elimina un elemento de ABB.
- esVacioABB: devuelve true si y solo si el árbol es vacío.

- LiberarABB: libera la memoria del ABB y la de todos sus elementos.

Árbol General

Árbol Binario



C,D Y E SON HERMANOS ENTRE  
S

Se puede representar un árbol general en un árbol binario. El árbol binario se representa de la forma **primer hijo**, **siguiente hermano**. En el árbol general se puede ver que a tiene como “hijos” a b, c, d y e, en el árbol binario eso se representa poniendo como hijo uno de ellos(en el dibujo se usa a b), y al resto como hermanos de el mismo(c,d,e).

Tener en cuenta: en altura, bajar al primero hijo aumenta la altura pero estar entre los hermanos no.

En borrar elementos, se borra el elementos y los hijos.

Árboles AVL:

Un árbol AVL es una ABB con una condición de equilibrio. La condición AVL: para cada nodo del árbol, la altura de los subárboles izquierdo y derecho puede diferir a lo más en 1 (hay que llevar y mantener la información de la altura de cada nodo, en el registro del nodo). Hay que tener cuidado al insertar un elemento porque puede dejar de estar equilibrado, para esto están las rotaciones, hacia la izquierda y derecha(simple) y la rotación doble(izq-der o der-izq).



## HASH

Consiste en :

- Un array de M elementos conocidos como “buckets”.
- Una función hash, que determina la celda donde se agrega el elemento.

Los elementos por lo general son números enteros o cadenas de string.

PARA NÚMERO ENTEROS  $\longrightarrow$   $\text{valor} \% M$ .

PARA STRINGS  $\longrightarrow$  sumar valores ASCII de los caracteres y el resultado  $\% M$  es el índice de la celda donde se agrega.

Puede pasar que se agregue el mismo valor, eso se llaman colisiones. Las estrategias mas comunes son:

- Hashing abierto.
- Hashing cerrado.
- Doble hashing.

## Tad Multisets:

Constructoras:

- crear: construye un multiset vacío.
- Insertar: inserta el elemento en el multiset.

Predicadora:

- EsVacio: devuelve true si y solo si el multiset es vacío.

Selectora:

- Ocurrencias: retorna la cantidad de veces que el elemento esta en el multiset.

- **Borrar:** elimina una ocurrencia del elemento en el multiset, si es que hay alguna.

### Destructora:

- **Destruir:** libera la memoria del multiset.

### Tad Mapping o Tablas:

Una tabla es una función parcial de elementos de un tipo, llamado el tipo dominio, a elementos de otro (posiblemente el mismo) tipo, llamado el tipo recorrido o rango.

(básicamente son asociaciones)

### Operaciones:

#### Constructoras:

- **crear:** devuelve una tabla vacía.
- **Insertar:** inserta la asociación en la tabla.

#### Predicadora:

- **EsVacio:** devuelve true si y solo si la tabla es vacía.

#### Selectora:

- **estaAsociado:** retorna true si y solo si el valor d esta asociado a otro valor en la tabla.
- **Asociado:** devuelve el valor que tiene asociado el elemento en la tabla.
- **Borrar:** elimina una correspondencia del elemento en la tabla, si es que hay alguna.

#### Destructora:

- **Destruir:** libera la memoria de la tabla.

Existen muchas implementaciones de las tablas con dominios finitos, una de las mas usadas es la de tablas de hashing., también se puede usar un AVL.(esto en caso de que los valores del dominio sean de tipo elemental de C/C++).

En una implementación con arreglos s e debe tener en cuenta un valor minimo y otro máximo para el dominio.

### Tiempo de ejecución:

- $T(n)$  = tiempo de ejecución de un programa con una entrada de tamaño  $n$ = número de instrucciones ejecutadas en un computador idealizado con una entrada de tamaño.
- $T_{\text{peor}}(n)$  = tiempo de ejecución para el peor caso (Nos centraremos en este caso y lo llamaremos  $T(n)$ )
- $T_{\text{prom}}(n)$  = tiempo de ejecución del caso promedio.
- Velocidad de crecimiento-  $O(n)$

$T(n)$  es  $O(f(n))$  “orden  $f(n)$ ” si existen constantes positivas  $c$  y  $n_0$  tales que  $T(n) \leq c \cdot f(n)$  cuando  $n \geq n_0$ .  $f(n)$  es una cota superior para la velocidad (taza) de crecimiento de un programa con tiempo de ejecución  $T(n)$ .

- Velocidad de crecimiento -  $\Omega(n)$

$T(n)$  es  $\Omega(g(n))$  si existen constantes positivas  $c$  y  $n_0$  tales que  $T(n) \geq c \cdot g(n)$  cuando  $n \geq n_0$ .  $g(n)$  es una cota inferior para la velocidad (taza) de crecimiento de un programa con tiempo de ejecución  $T(n)$

## Algunas velocidades de crecimiento típicas

Para  $n$   
grande

Función	Nombre
c	constante
$\log(n)$	logarítmica
$\log^2(n)$	log-cuadrado
n	lineal
$n \cdot \log(n)$	
$n^2$	cuadrática
$n^3$	cúbica
$2^n$	exponencial

- **Regla de la Suma:** Si  $T_1(n)$  es  $O(f_1(n))$  y  $T_2(n)$  es  $O(f_2(n))$  entonces  $T_1(n) + T_2(n)$  es  $O(\max(f_1(n), f_2(n)))$  —————> Puede usarse para calcular el tiempo de ejecución de una secuencia de pasos de programa.
- **Regla del Producto:** Si  $T_1(n)$  es  $O(f_1(n))$  y  $T_2(n)$  es  $O(f_2(n))$  entonces  $T_1(n) \cdot T_2(n)$  es  $O(f_1(n) \cdot f_2(n))$ .  $O(c \cdot f(n))$  es lo mismo que  $O(f(n))$  (c es una cte positiva)

### REGLAS PARA CALCULO DE $T(n)$

- Para una asignación (lectura/escritura e instrucciones básicas) es en general  $O(1)$  (tiempo constante).
- Para una secuencia de pasos se determina por la regla de la suma (dentro de un factor cte, el “máximo”).
- Para un “if (Cond) Sent” es el tiempo para Sent más el tiempo para evaluar Cond (este último en general  $O(1)$  para condiciones simples).
- Para un “if (Cond) Sent<sub>1</sub> else Sent<sub>2</sub>” es el tiempo para evaluar Cond más el máximo entre los tiempos para Sent<sub>1</sub> y Sent<sub>2</sub>.

- Para un ciclo es la suma, sobre todas las iteraciones del ciclo ( $\sum$ ), del tiempo de ejecución del cuerpo y del empleado para evaluar la condición de terminación (este último suele ser  $O(1)$ ).  $\longrightarrow$  A menudo este tiempo es, despreciando factores constantes, el producto del número de iteraciones del ciclo y el mayor tiempo posible para una ejecución del cuerpo.

**PIQUE:** Para reducir el tiempo es mejor secuencializar las instrucciones, si estas anidadas se multiplica el tiempo y secuencializadas se suman ( $10 \times 10 = 100$ , en cambio  $10 + 10 = 20$ ).

**Multiestructuras y Diseño de TADs:**

**Multiestructuras:** uso simultáneo de dos o más estructuras diferentes para el mismo conjunto o correspondencia, buscando acceso rápido pero sin redundancia de información.

**PIQUE:** Piensen en las estructuras y sus tiempos en las operaciones básicas, y piensen cuáles conviene más para tener un mejor tiempo.