Estructuras de datos jerárquicas

# Árboles

Es una estructura de datos (almacenamiento) combina las ventajas de los arreglos ordenados y las listas enlazadas.

Listas En 🡪 cualquier tamaño, búsqueda lenta, estructura dinamina

Array 🡪 búsqueda rápida por acceso indexado

Nodos (círculos) conectados por aristas (líneas).

Son entidades matemáticas

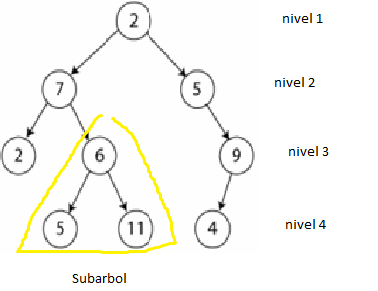
Es una especialización de los grafos.

Pequeños en el tope, grandes al final

(Binarios, heap, AVL, B B+ B\*, Expression tres, N-ary trees)

Máximo 2 hijos

Solo existe una ruta entre dos pares de nodos (cada nodo tiene una ruta única)



# Funciones

Funcionan mucho para hacer sistemas de archivos. Permiten organizar los archivos. Para facilitar la búsqueda de la información

En bases de datos la información se almacena en arboles B.

Algoritmos de compresión

Arboles de expresión para ciertos compiladores

# Terminología

Recorrer, es el procedo de ir de un X a un nodo Y

Ruta, es la lista de nodos en un recorrido

Raíz, esta en la cabeza del árbol, solo puede haber una. (Dentro del árbol en general existen subárboles, y por lo tanto raíces de subárboles).

Cualquier nodo tiene UNA UNICA arista hacia arriba (un nodo solo tiene un padre). Solo no aplica para la raíz.

Cualquier nodo puede tener una o mas líneas hacia abajo (incluso ninguna), estas líneas conectan con los nodos hijos (esto para árboles en general)

Nodo hoja, es un nodo que no tiene hijos.

Cualquier nodo puede ser considerado una raíz de un subárbol, e incluye absolutamente todos los nodos jerárquicamente debajo de este.

Dos nodos con el mismo padre se llaman Nodos hermanos.

Largo de una ruta (desde cualquier parte), es la cantidad de aristas presentes en esa ruta.

Siempre hay una ruta del nodo a el mismo de largo 0.

Profundidad de un nodo (desde raíz), el largo de la ruta única desde el nodo a la raíz. La profundidad de la raíz es cero.

La altura de un nodo es el largo de la ruta mas larga hasta un nodo hoja

Un nodo es visitado cuando se realiza alguna clase de acción sobre la información (si solo paso por él no se puede considerar como visitado).

Se recorre un árbol se VISITAN todos los nodos de un árbol.

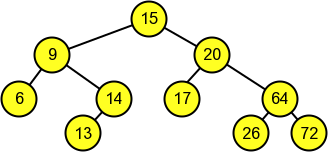
# Arboles binarios

Un árbol donde cada nodo tiene a lo sumo 2 hijos. Es decir (0,1,2). Si esto no se cumple no es binario.

Los hijos se referencian como hijo izquierdo e hijo derecho.

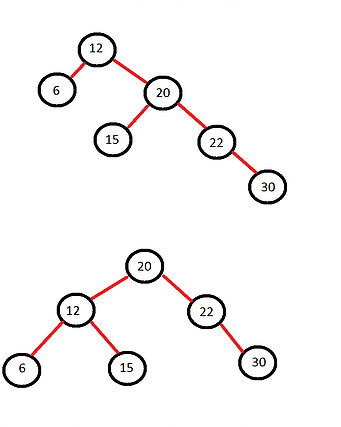
## Árbol binario de búsqueda

Para todo nodo se cumple que: “mi hijo izquierdo es menor y mi hijo derecho es mayor que yo”.



Dependiendo del orden de como entren los datos puede tenerse un árbol desbalanceado

Uno de los dos lados tiene más niveles que otro.



Se puede implementar con arreglos y con estructuras de datos dinámicas.

Pero no es tan conveniente utilizar arreglos.

Clase Nodo <T extends Comparable<? Super T>>

Nodo<T> left;

Nodo<T> right;

Clase arbolBinario <T extends Comparable<? Super T>>

Private Node<T> root;

## Operaciones

Boolean Contiene ()

True si esta el elemento

False si esta vacio o no lo tine

Public booleab contains (element)

Return this.contains(element, this.root);

Private Boolean contains (element, nodo){

If (node == null){

Return false;

Else{

Int compareResult = element.compareTo(node.element);

If (compareResult < 0)

Return contains(element, node.left);

Else

If (compareResult > 0)

Return contains(element, node.right);

Else

Return true;

FindMin()

findMin(nodo)

If this.isEmp

Return findMin(node.left)

Mínimo (Full izq, hasta encontrar uno sin izquierda)

Máximo (Full der, hasta encontrar uno sin derecha)

## Insert

Recorra el árbol con contiene y asegúrese q no existe el dato a insertar. Decida que hacer en caso de que si esté. Si no está inserte el nodo en la posición a donde seguiría buscando.

## Delete

Es la operación mas compleja. Cuando se encuentra el nodo hay que analizar si:

* El nodo es hoja:

Solo se elimina

* El nodo tiene un hijo

El abuelo tiene que tomar la referencia del abuelo

* El nodo tiene dos hijos:

Remplazar el dato del nodo actual con el dato más pequeño del subárbol derecho, o bien el mayor del subárbol izquierdo. Y después llamar recursivamente el método para eliminar el nodo que acabamos de mover.2

# Árbol ALV

Nombre por sus creadores  matemáticos rusos **[A](https://es.wikipedia.org/wiki/Georgii_Adelson-Velskii" \o "Georgii Adelson-Velskii)**[delson-](https://es.wikipedia.org/wiki/Georgii_Adelson-Velskii" \o "Georgii Adelson-Velskii)**[V](https://es.wikipedia.org/wiki/Georgii_Adelson-Velskii" \o "Georgii Adelson-Velskii)**[elskii](https://es.wikipedia.org/wiki/Georgii_Adelson-Velskii" \o "Georgii Adelson-Velskii) y **[L](https://es.wikipedia.org/wiki/Yevgeniy_Landis" \o "Yevgeniy Landis)**[andis](https://es.wikipedia.org/wiki/Yevgeniy_Landis" \o "Yevgeniy Landis)

Es un árbol que siempre se mantienen como máximo a 1 nivel de la diferencia.

La altura es el máximo nivel +1.

Son arboles binarios siempre balanceados

Tomo el nodo hoja más bajo de cada lado (empezando de un root) y los comparo

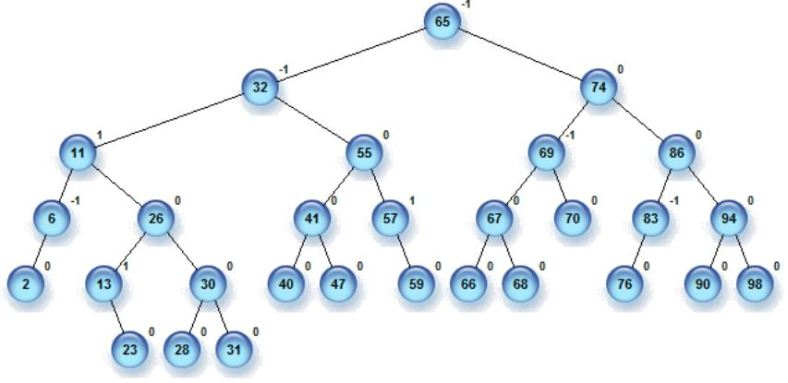
Un árbol AVL no contiene subárboles ALV.

Un árbol balanceado perfecto tiene todos sus nodos hoja al mismo nivel.

El factor de balanceo es la altura del árbol derecho – la altura del subárbol izquierdo

* Se calcula para cada nodo

Para un árbol AVL el factor de balanceo es siempre 1,0,-1, nada más.



Al insertar un nodo puede romperse la condición del ALV. Si esto ocurre aquí hay que aplicar Rotación. Solo los nodos que estén en la ruta del nuevo nodo pueden haber sufrido camios es su factor de balanceo. Desde la hoja a la raíz.

El nuevo nodo se agrega como una hoja con FB = 0. Y se realiza un backtrack para asegurarse si hay cambios en el FB de estos.

Cuando hay un desbalance hay cuatro casos:

Izquierdo a Izquierdo

Derecho a Derecho

* Se aplica una rotación sencilla (single rotation)

Izquierdo a derecho

Derecho a izquierdo

* Se aplica una rotación doble (double rotation)

## Izquierdo a Izquierdo (LL)

n.left = n1.right;

n1.right = n;

n = n1

(n está colocado donde está el nodo con el nodo desbalanceado)

(n1 es el nodo izq de n)

## Derecho a Derecho (DD)

n.right =n1.left,

n1.left = n;

n = n1

(n1 es el nodo derecho de n)

## Derecho a izquierdo

LL por el subárbol derecho + DD por el root

n1.left = n2.right;

n2.right = n1;

n.right = n2.left;

n2.left = n;

n = n2

## Izquierdo a Derecho

DD por el subárbol derecho + LL por el root

n1.right = n2.left;

n2.left = n1;

n.left = n2.right;

n2.right = n;

n = n2

# Splay tree (Arbol biselar)

Es una mejora en eficiencia del árbol binario, se enfoca en reducir el tiempo de acceso.

“Extender el árbol hacia arriba”.

Cada vez que se accede (visita) a la información de un nodo pase a ser la raíz del nodo.

No se balancea, no tiene altura.

## Rotaciones

Buscamos x

### Zig (LL) / Zag(RR)

Rotación simple del avl.

Visité x, ahora x va a ser raíz.

X no es la raíz, pero su padre es la raíz.

Movemos x a la raíz mediante una rotación simple

### Zig-zag (left-right) / zag-zig (right-left)

X es hijo derecho.

P nodo padre, es hijo izquierdo.

G nodo abuelo.

Se va a aplicar una doble rotación estilo AVL (dependiendo del tipo)

### Zig-zig (zag-zag)

P y X son ambos nodos hijos izquierdos o derechos.

Buscar como funciona:

## Inserción

Lo inserta (hasta abajo) y después rota hasta que sea raíz este la raíz.

## Find

Si la búsqueda es satisfactoria el nodo pasa a ser la raíz.

Si no fue satisfactoria, la raíz pasa a ser el nodo ultimo contra el que comparé.

FindMin and FindMax, ese mínimo o máximo rotan hasta ser la raíz

### DeleteMin:

El mínimo pasa a ser la raíz, no hay hijo izquierdo, y la nueva raíz va a ser el hijo derecho

### DeleteMax:

El máximo pasa a ser la raíz, no hay hijo derecho, y la nueva raíz va a ser el hijo izq.

## Remove

Convierta el nodo a borrar en raíz

Deje dos subárboles

Haga un findMax en el subárbol L para que ya no tenga hijo izq. Y péguele como hijo derecho el subárbol R a L. (se puede hacer con un findMin y el hijo izq).

# Heap

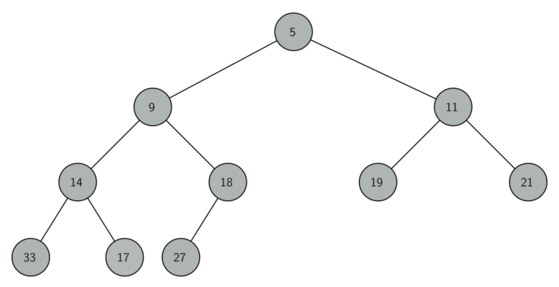
Es un árbol que implementa colas de prioridad con de árbol.

Cuando hay un programa en memoria existe un espacio para:

Stack crece hacia abajo

Heap crece hacia arriba

La inserción en el heap es muchos más rápida que en array o lista enlazada (nlogn)

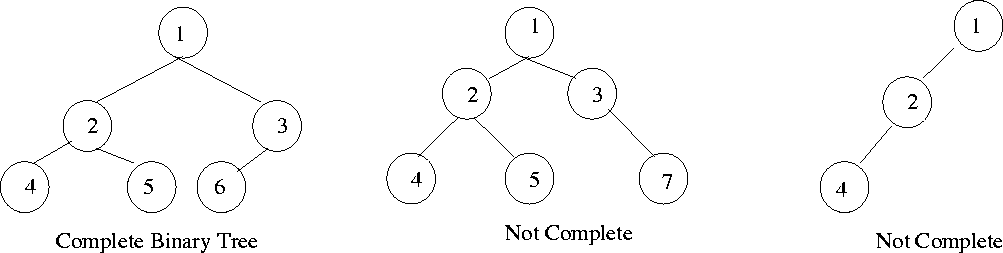


## Heap es un árbol binario que:

* **Completo**

De izquierda a derecha de arriaba abajo si hay un nodo vacío no hay más nodos.

“la fila de nodos termina cuando hay uno vacío, y no hay más árbol abajo”



* **Esta implementado con arreglos**

Tiene que usar arreglos (el árbol en si es un arreglo)

### Encontrar hijos y padre:

k es la posición del nodo, empezando en 0

El padre de un nodo 🡪(k-1)/2

Hijo derecho 🡪 k\*2+2

Hijo izquierdo 🡪 k\*2+1

Gracias al que el árbol es binario y es completo.

* **Satisface la condición del heap**

Los nodos llave hacia debajo de un nodo son menores (independientemente de izquierda o derecha)

Y por lo tanto la llave mayor es la raíz

Mas valor de llave más prioridad tiene el elemento, por eso es una cola de prioridad

(igual podría ser por menor u otra cosa)

No tiene función find por ser una cola de prioridad, es decir solo va a interactuar con la raíz.

La llave máxima siempre es la raíz, y es la que voy a sacar. Pero incumple la condición del heap cuando quito la raíz, por lo que cuando se ejecuta:

* Quito la raíz
* Mi nueva raíz es ahora el ultimo nodo
* Acomode con swaps la nueva raíz: Comparo los hijos de la nueva raíz, y elijo al mayor
* Después comparo el hijo seccionado con la raíz, si es menor hago swap
* Repita (con el nodo que solía ser la raíz) hasta que sus hijos sean menores

Cuando uno inserta un nuevo nodo se inserta en el primer espacio disponible. Es decir, en el siguiente nodo tomando en cuanta la completitud del árbol.

Pregunta si hace swap contra el padre (porque solo hay un padre por nodo). Y así hasta quedar en la posición indicada. Para cumplir con la condición del heap (debajo todos son menores)

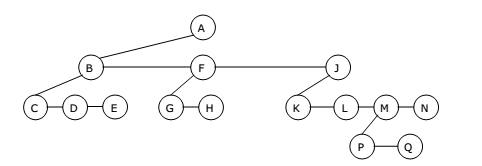
# Árbol n-ario

Árbol no binario, árbol k-ario, árbol general

Cada nodo tiene un método

getMiHermanoDerecho()

getHijoMasALaIzquierda()



Cada nodo tiene una referencia al head de una listaHijos

Sirven para hacer búsquedas por texto.

# Arboles B

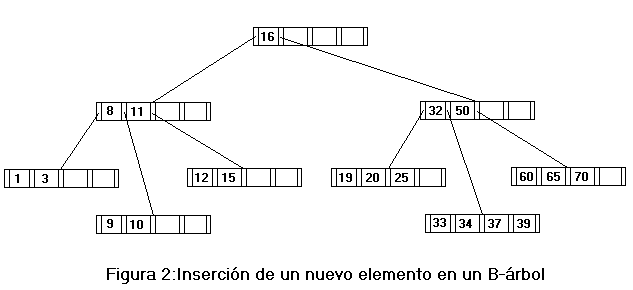
Árboles que se implementan en disco duro y no en memoria principal. A diferencia a todos los antes vistos. Cuando hay millones de nodos los uso en disco duro, porque crece mucho.

Por lo tanto, tengo un árbol que almacena muchos datos.

La velocidad de lectura sobre un disco duro es mucho menor que en la memoria principal.

Gano mucho en espacio, pero pierdo mucho en latencia (costo en latencia).

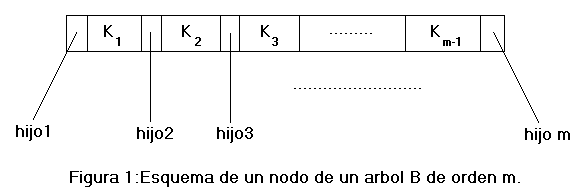
El árbol B viene a intentar mitigar el efecto en la latencia, no lo elimina, el precio ha de pagarse.



Son arboles de orden m, es un árbol m-iario

Los árboles B no tiene nodos, sino paginas

Y donde n es la cantidad máxima de hijos que puede tener una página.

Todas las hojas deben estar en el mismo nivel. 

Todas las paginas tienen un máximo de m ramificaciones y un mínimo de m/2 ramificaciones

La cantidad de llaves que hay al interno de cada pagina debe coincidir con la cantidad de ramificaciones -1 (7 ramificaciones, 6 llaves, por ejemplo).

Las llaves dividen a las ramificaciones(ramas) de manera similar al árbol binario de búsqueda. Donde todo a la derecha es mayor que la llave y lo menos a la izquierda.

El hijo derecho de una llave puede ser el hijo izquierdo de su hermana.

La raíz igual tiene un máximo de m y un mínimo de menos de dos.

Un árbol B siempre está perfectamente ordenado/balanceado

Por lo tanto, el objetivo del árbol B es reducir la profundidad del árbol. (menos accesos al disco en un caso)

Para determinar m, se utiliza q la cantidad de llaves a ramificaciones -1

Los nodos tienen que estar half-full para evitar que se convierta a un árbol binario de búsqueda

El orden coincide con el tamaño de los bloques del disco.

El árbol B no crece hacia abajo, si no hacia arriba.

## Insertar

* Busque la llave en el árbol, igual que en arboles binarios de búsqueda (donde esta o donde debería estar). Si la llave no esta vas a llegar a un nodo hoja. Las llaves también están ordenadas.
* Si la hoja no esta llena se inserta (de manera ordenada) en la pagina
* Si la hoja está llena se va a ejecutar un Split. –> Separe la pagina en dos mitades. El que está en la mitad sube a la pagina del padre. (esta operación se puede propagar hasta llegar a la raíz), y se inserta donde le corresponda.

## Eliminación

## Caso 1: eliminar una hoja con de sobra

### Caso 1: Nodo hoja con más del mínimo

(al eliminarlo todavía la pagina tiene >= el mínimo de elementos mínimo [m/2, la mitad del grado])

Tan solo se busca y elimina

## Caso 2: Eliminar de un nodo interior

### Caso 2A: Nodo interior con mínimo y el predecesor le sobra llave

Elimina el valor

Se toma el predecesor (máx. de la lista/hijo izquierdo) y se sube a donde estaba el valor eliminado

### Caso 2B: Nodo interior con mínimo y el predecesor no le sobra llave

(pero al sucesor si le sobra) Se hace lo mismo

Toma al sucesor (el min de la lista/hijo derecho)

### Caso 2C: No le sobra a ninguno

Se baja el valor a eliminar de la página y se hace un merge entre su hijo izquierdo y su hijo derecho, y luego se elimina

## Caso 3: eliminar una hoja con el mínimo

### Caso 3A: Nodo hoja con mínimo, hermano le sobra

Hace una rotación

El hermano sube uno, el padre baja uno (al otro lado) y la llave se elimina.

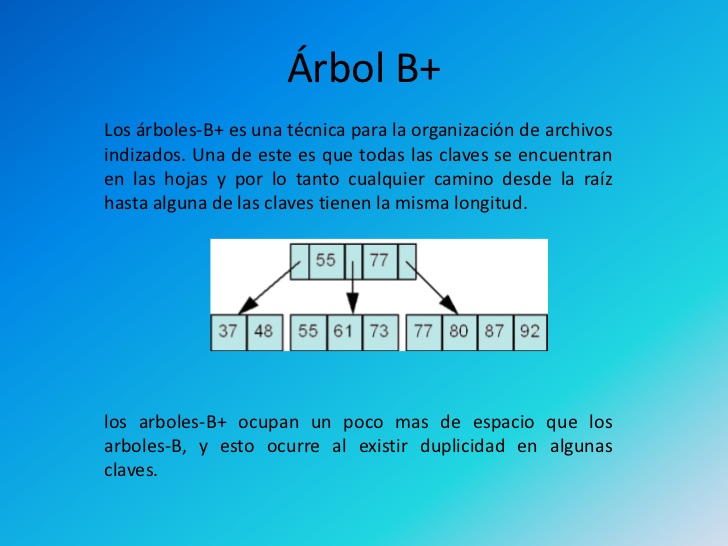
### Caso 3B: Nodo hoja con mínimo, hermano no le sobra

Se elimina el valor de la página y se hace un merge entre él y su hermano.

# Árbol B+

Los nodos con información se almacenan únicamente en los nodos hoja. Los nodos que no son hoja tienen únicamente una referencia al dato que se está buscando.

La hoja tiene referencia al hermano derecho.



# Árbol B\*

Consiste en que se aumenta el número mínimo de valores de clave que puede contener un nodo que no sea la raíz, de forma que en lugar de garantizar un 50% de utilización de espacio como en el árbol B, se aproveche como mínimo las dos terceras partes del mismo.

Se busca un árbol aun menos profundo.

Cada nodo excepto hoja y raíz tiene al menos 2m-1/3 hijos.

La raíz tiene al menos 2 hijos y máximo 2((2m-2) /3) +1 hijos.

El proceso de inserción en más lento

Cualquier nodo que los sea hoja de k hijos tiene k -1 llaves.

Al insertar la idea es no hacer Split. Entonces:

1.Insertar la llave como si realmente hubiera espacio.

2.Crear una nueva página que pasara a ser la raíz del árbol.

3.Dividir la pagina en dos partes equitativas y subir la mediana al nuevo nodo creado.

4.Las dos páginas creadas pasaran a ser respectivamente el lado izquierdo y derecho de la nueva página.

# Árboles de expresión

Son arboles binarios.

Una expresión son una secuencia de tokens (operadores y keywords/componentes léxicos con una serie de regles).

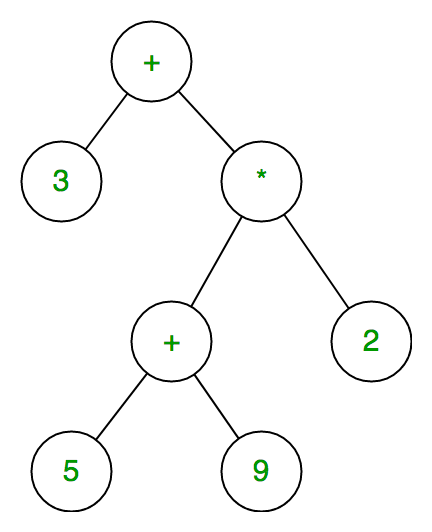
Propiedades:

Cada hoja es un operando

La raíz y los nodos internos son operadores.

Y cada subárbol es una sobrexpresión con la raíz como operador.

Son utilizados en lenguajes compiladores.



3+((5+9)\*2)

Si hay un nodo con un operador, abre paréntesis

Recorrer arboles por orden:

## Infijo / Inorden

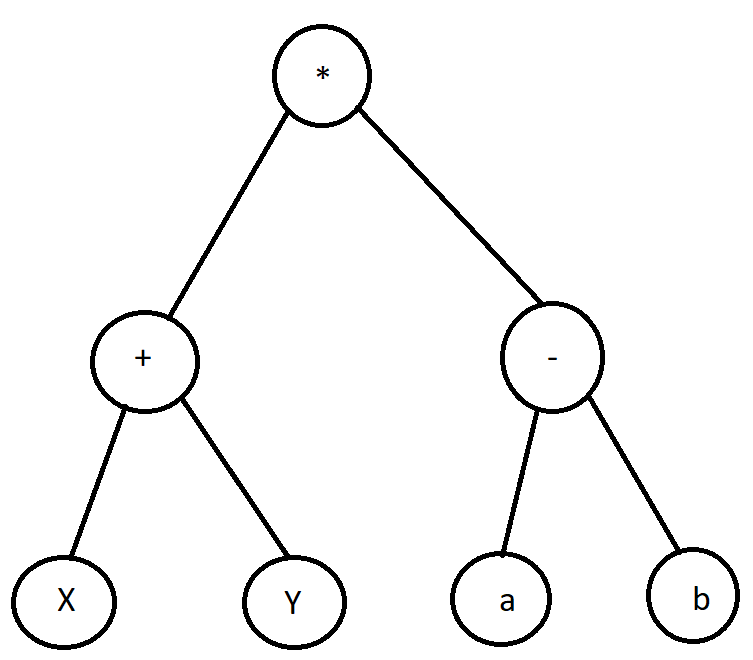
Lo recorre de izq. a derecha en triangulitos empezando del más pequeño de ese subárbol

Posfijo / Posorden

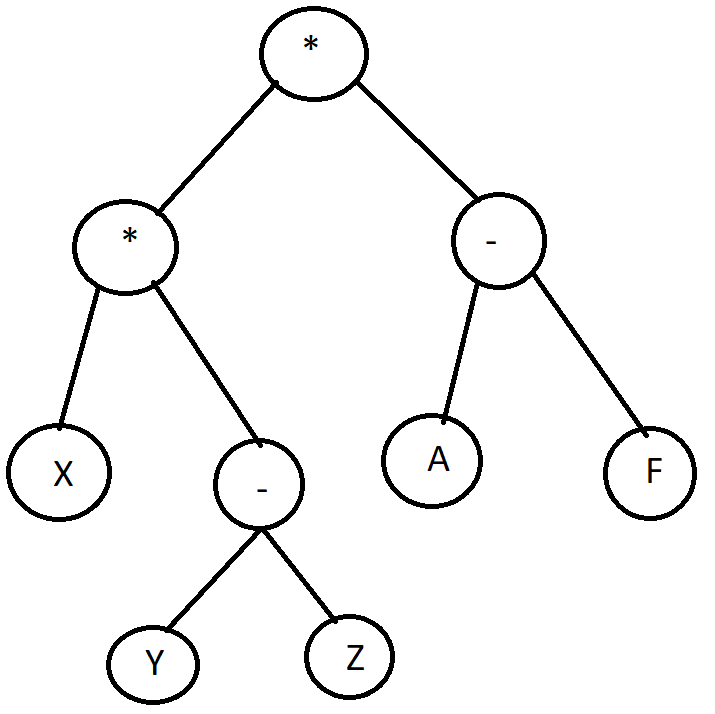
Empieza desde abajo sin llegar hasta la raíz, primero izq. después derecha (de abajo a arriba) Da énfasis a los menores

## Preorden

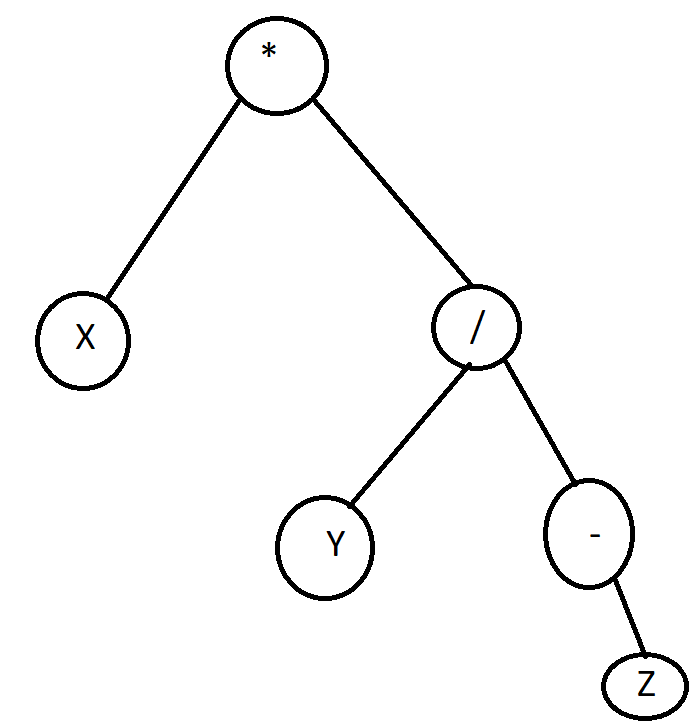
raíz, izquierda, derecha (de arriba abajo dando prioridad izq.)



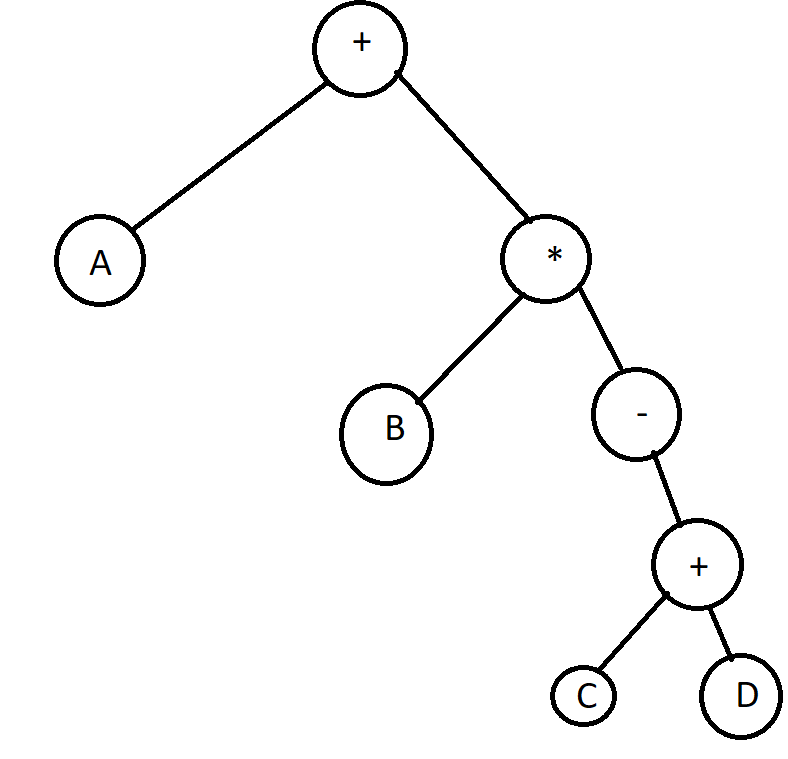
((x+y)\*(a-b))



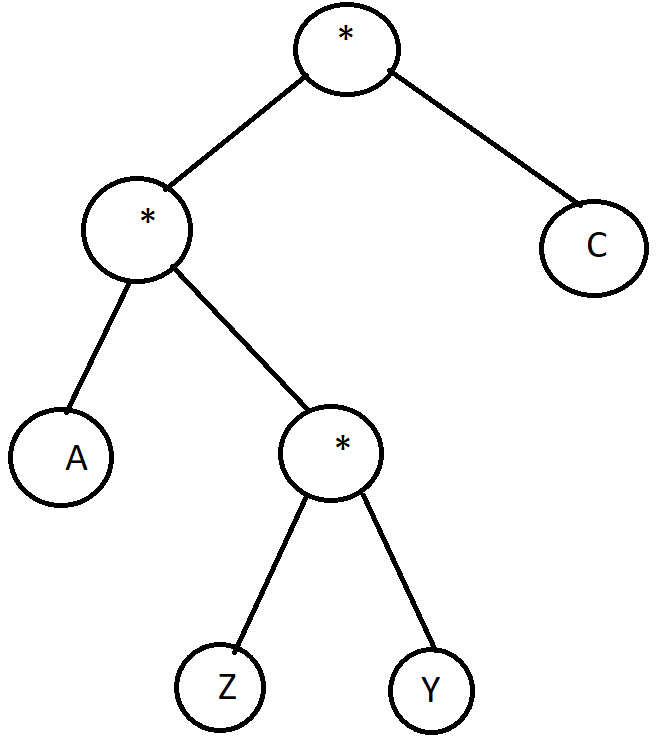
((X\*(Y-Z))\*(A-F))



(X\*(Y/(-Z)))



(A+(B\*(-(C+D))))

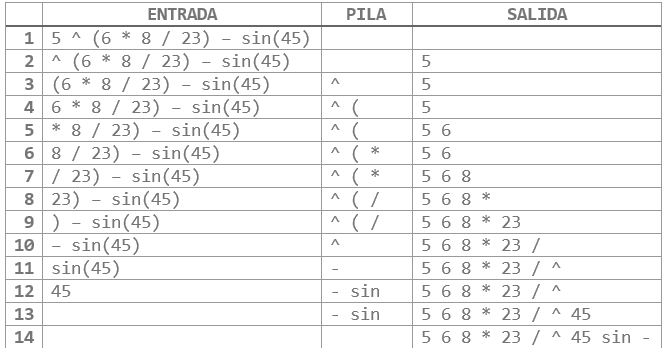


((A\*(X+Y))\*C)

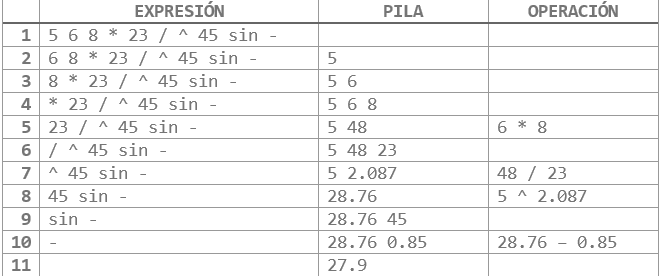
Output q (operador)

Stack(operando)

Metolos operandos. Cuando hay un paréntesis que cierra método todo lo que me aqueda en el paréntesis que se cerró. Y los quito.

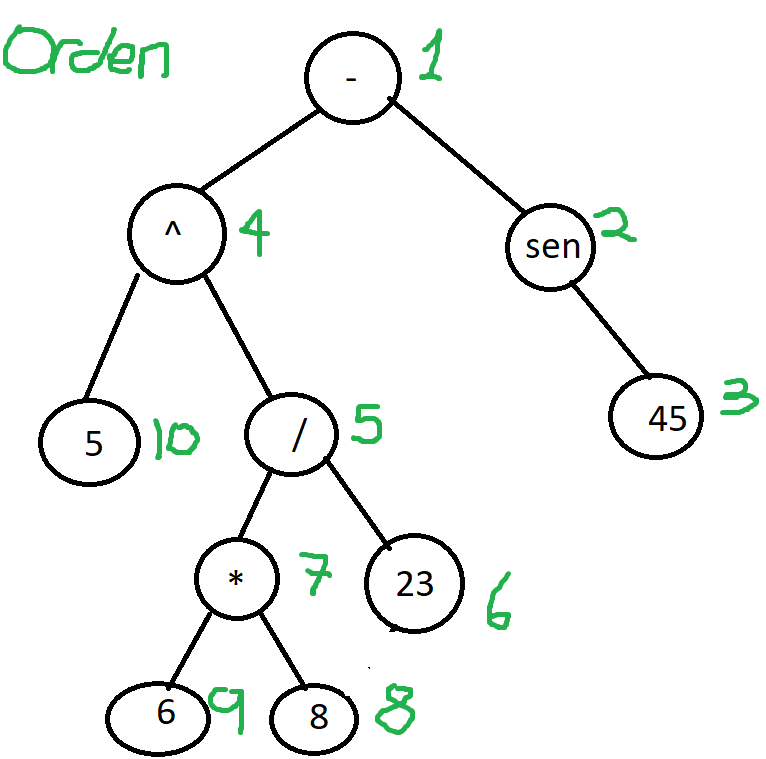


Después trabajo en stack para hacer la operación o en cola si voy a montar un árbol.



Para el árbol se agrega en preorden pero al inverso

El árbol quedaría asi:



Se pasa del 3 al 4 así pues es imposible que sen pueda tener más de un hijo.