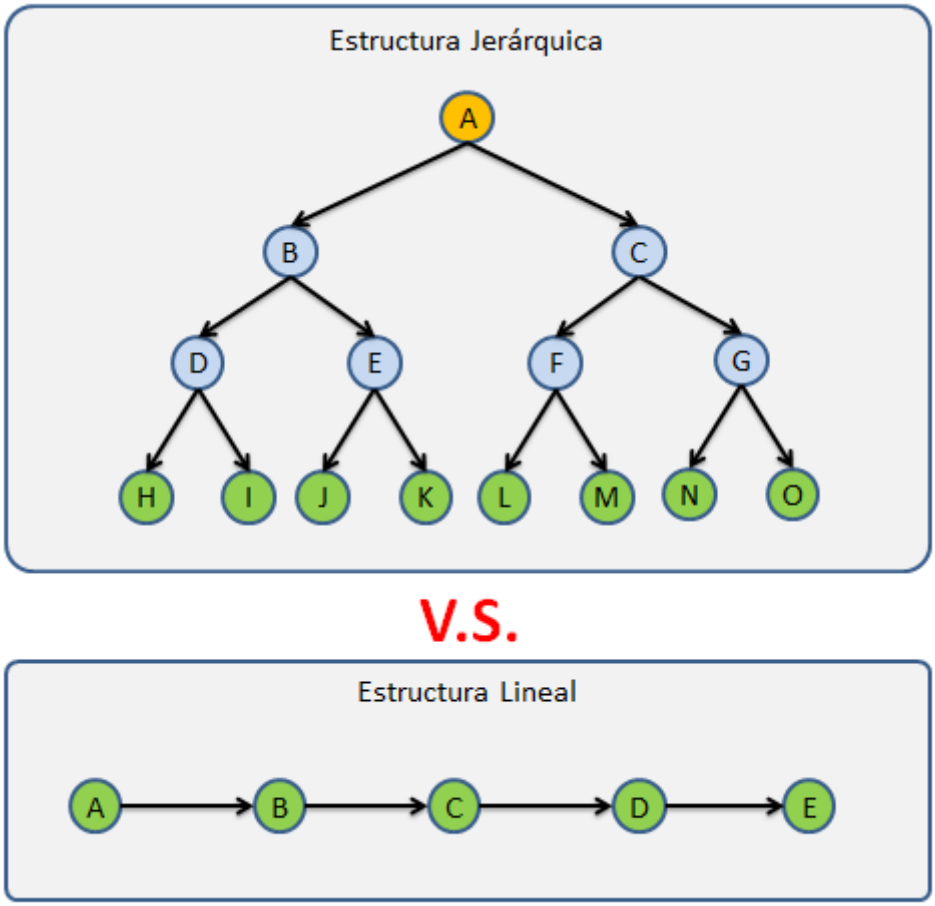
Índice

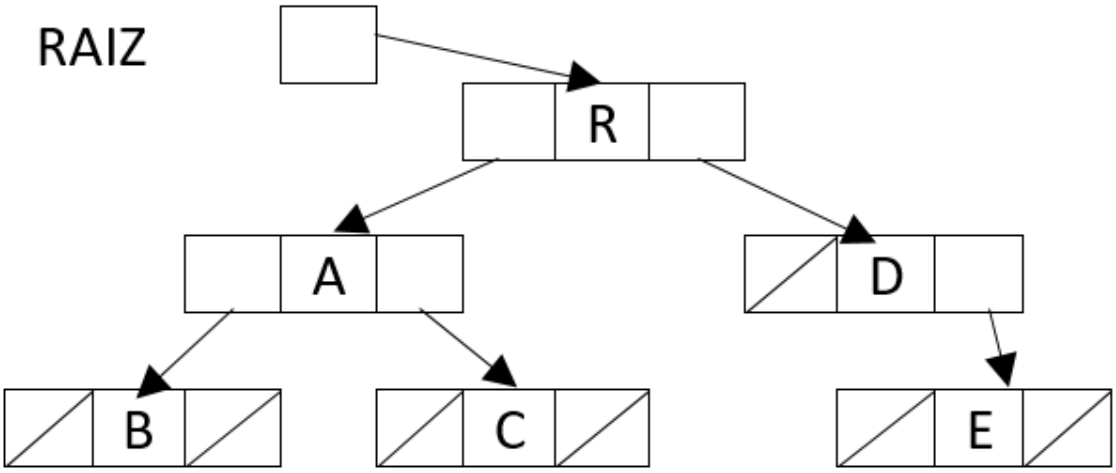
Definicion y conceptos

Definición 1: Un árbol es una estructura no lineal en la que cada nodo puede apuntar a uno o varios nodos. Cada nodo tiene un padre y sólo uno (excepto la *raíz*), pero puede tener cero o más hijos.

Definición 2: Un árbol es una estructura en compuesta por un la *raíz* y una lista de árboles. El nodo *raíz* es el padre de las raíces de los árboles que componen la lista, a partir del cual se establece la relación de paternidad entre ellos.



Un gráfico de un árbol con nodos sería algo así:

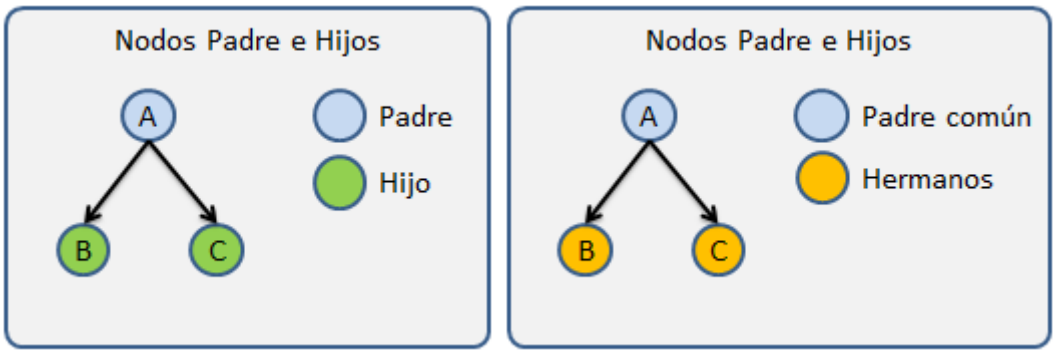


A veces por una cuestión de comodidad, se puede agregar un puntero más al nodo, que apunte al padre.

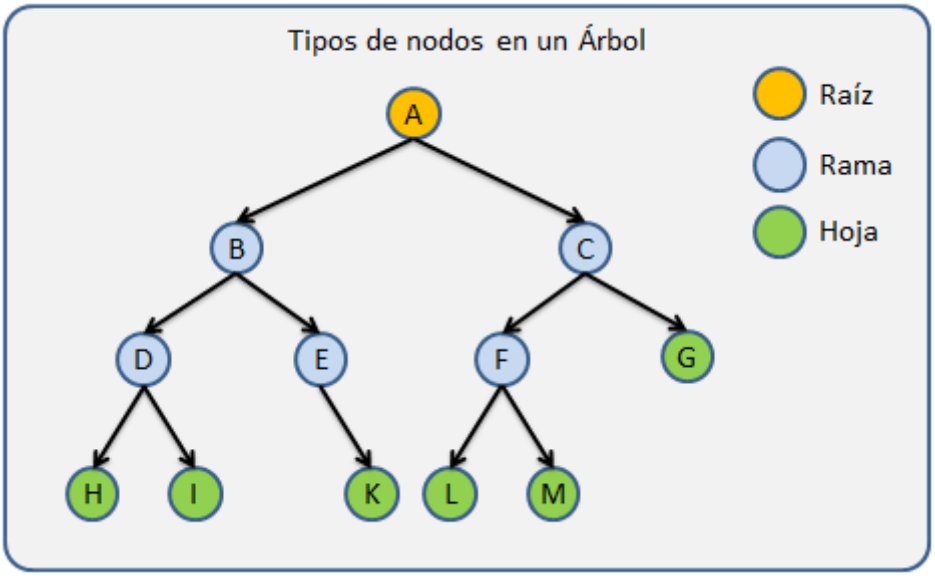
Algunos conceptos que es necesario entender antes de seguir:

- *Camino* de un nodo n a m: es la secuencia de nodos n, n1, n2, ... , m
- *Longitud* de un camino: es el número de nodos en el camino menos uno
- Nodo ancestro o padre: aquellos nodos que tienen al menos un descendiente o hijo.
- Nodo descendiente o hijo: aquellos nodos que tienen un padre

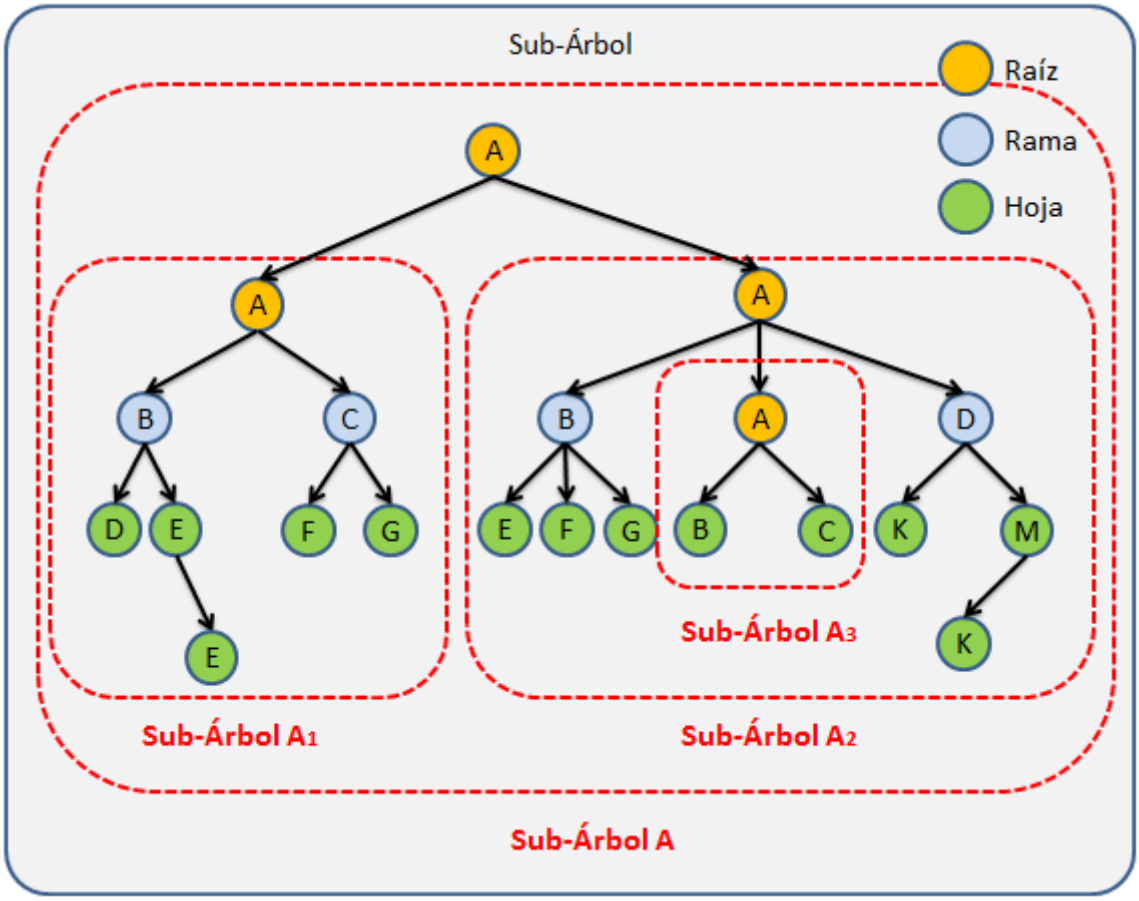
- Nodo hermano: aquellos que comparten a un mismo padre en común dentro de la estructura



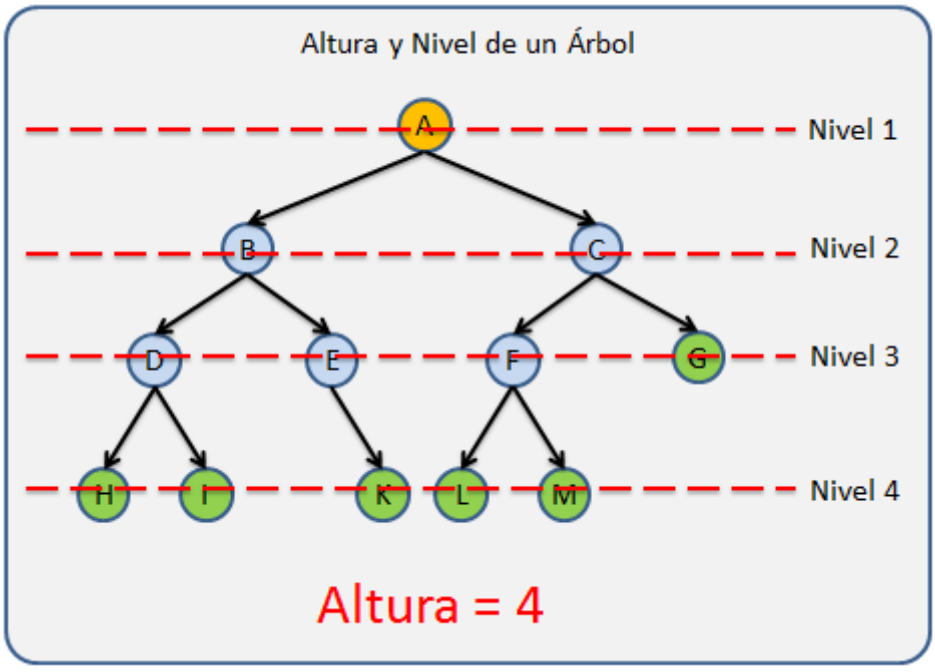
- Nodo *hoja:* es un nodo sin descendientes



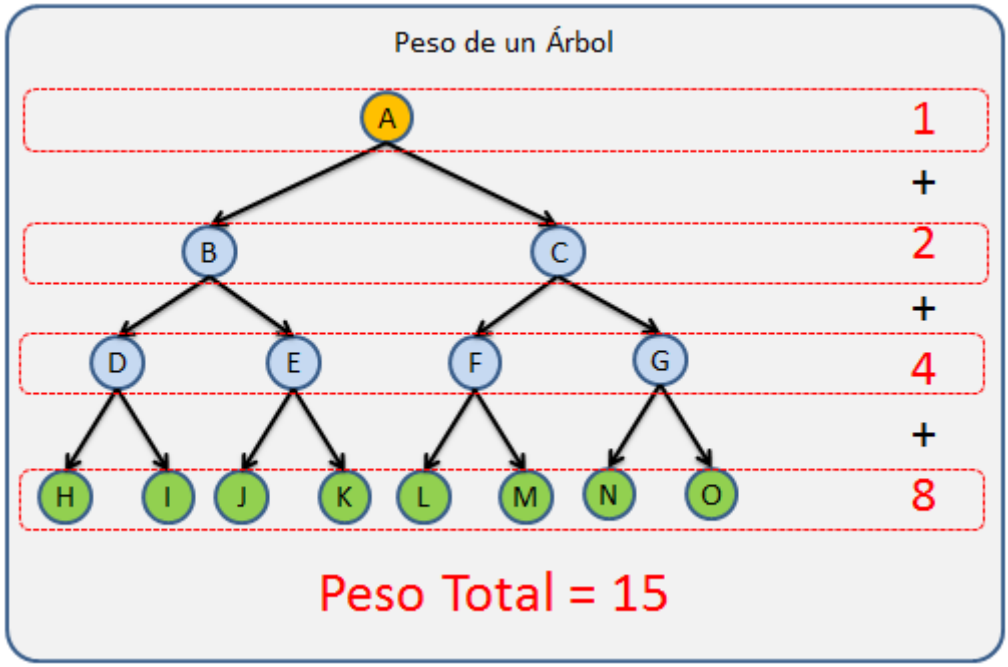
- *Subárbol:* es un nodo del árbol junto con todos sus descendientes



- *Profundidad* es la longitud del camino único de la raíz al nodo
- *Nivel*: un nivel es una generación dentro del árbol. Cada generación tiene un número de nivel distinto que las demas generaciones.
    - Un árbol vacío tiene 0 niveles
    - El nivel de la raíz es 1
    - El nivel de cada nodo se calcula contando cuantos nodos existen sobre el, hasta llegar a la raíz + 1
- *Altura*: es el número máximo de niveles de un árbol

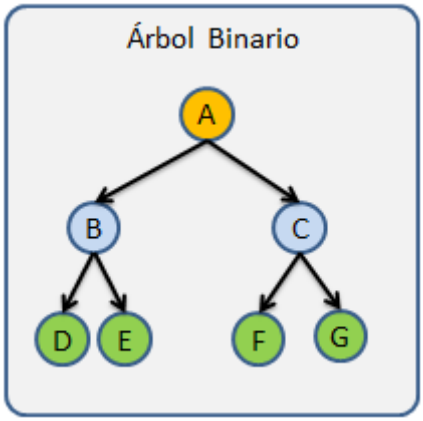


- *Peso:* es el número de nodos que tiene un árbol

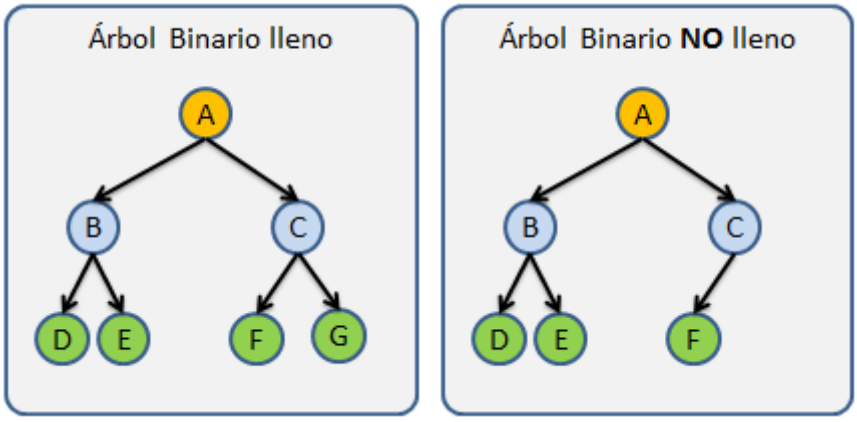


Arbol binario

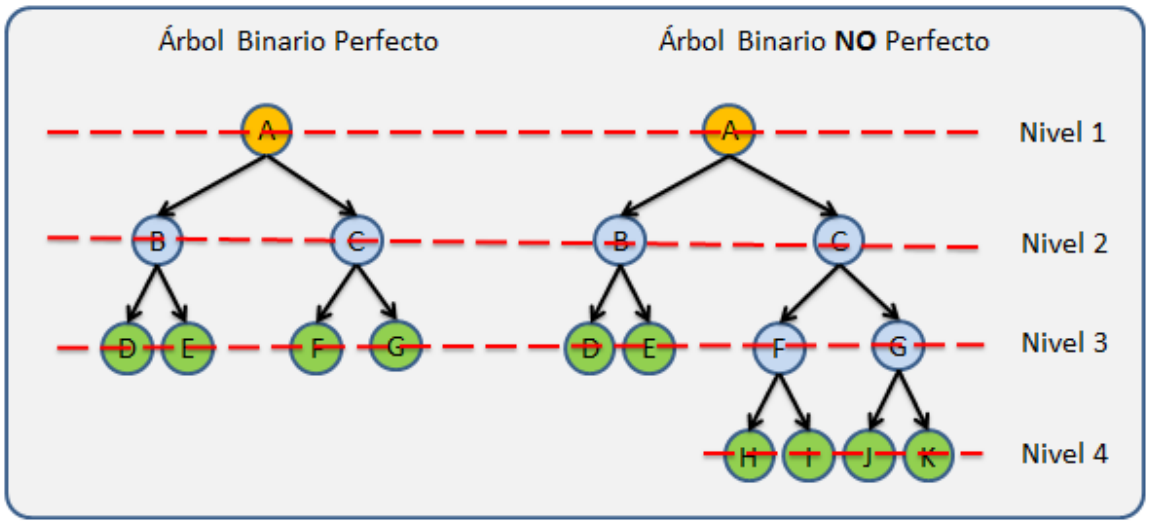
Esta estructura se caracteriza por que cada nodo solo puede tener como máximo 2 hijos.



Se dice que el árbol binario está lleno cuando todos los nodos tienen 0 o 2 hijos (con excepción de la raíz)



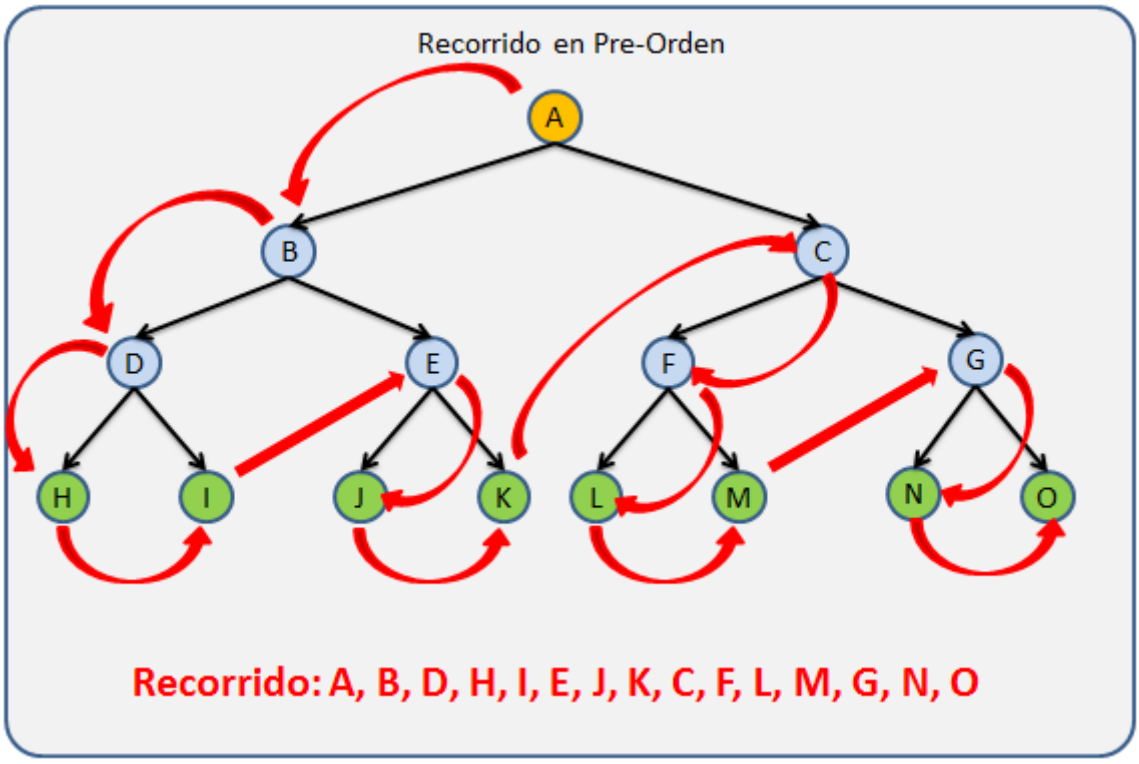
Se dice que un arból binario es perfecto, cuando está lleno y balanceado



Busqueda en profundidad

Recorrido preorden

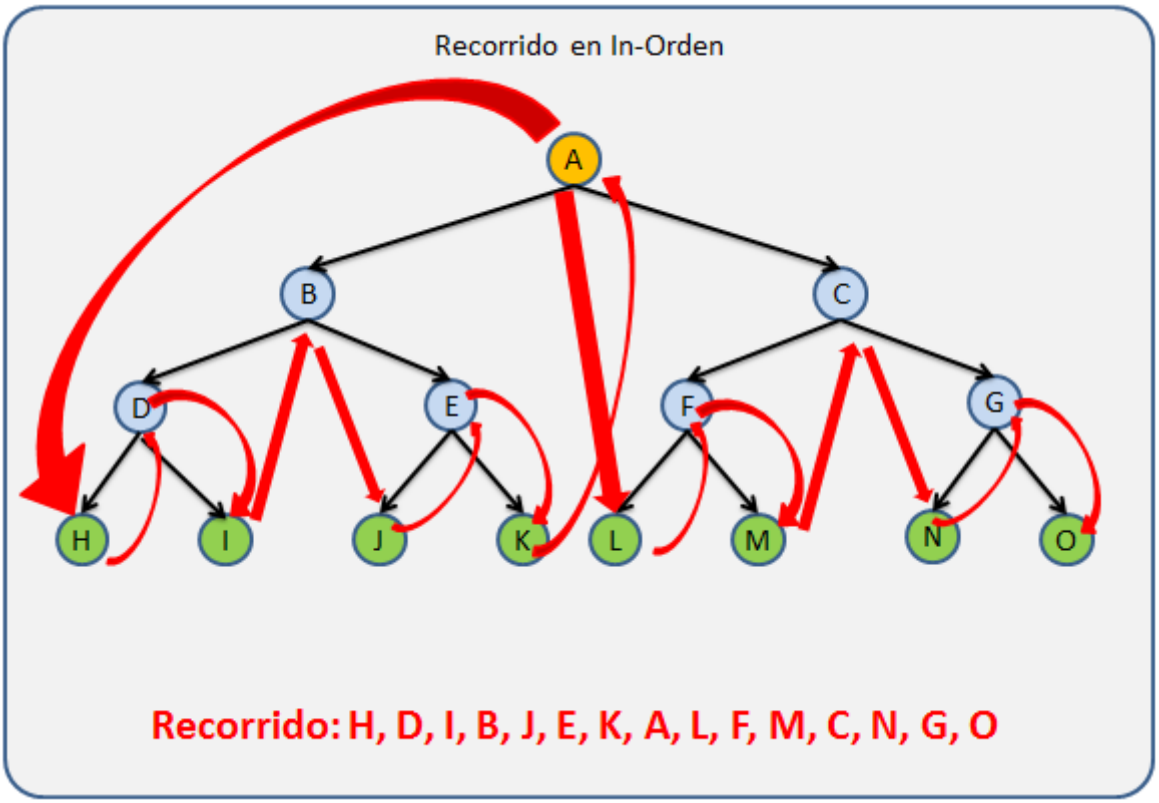
El recorrido inicia en la Raíz y luego se recorre en pre-orden cada unos de los sub-árboles de izquierda a derecha.




```
// Implementación recursiva
void preorden(NodoArbol* raiz) {
    if (raiz != NULL) {
        tratar(raiz);
        preorden(raiz->obtenerHijoIzquierda());
        preorden(raiz->obtenerHijoDerecha());
    }
}
```

Recorrido inorden

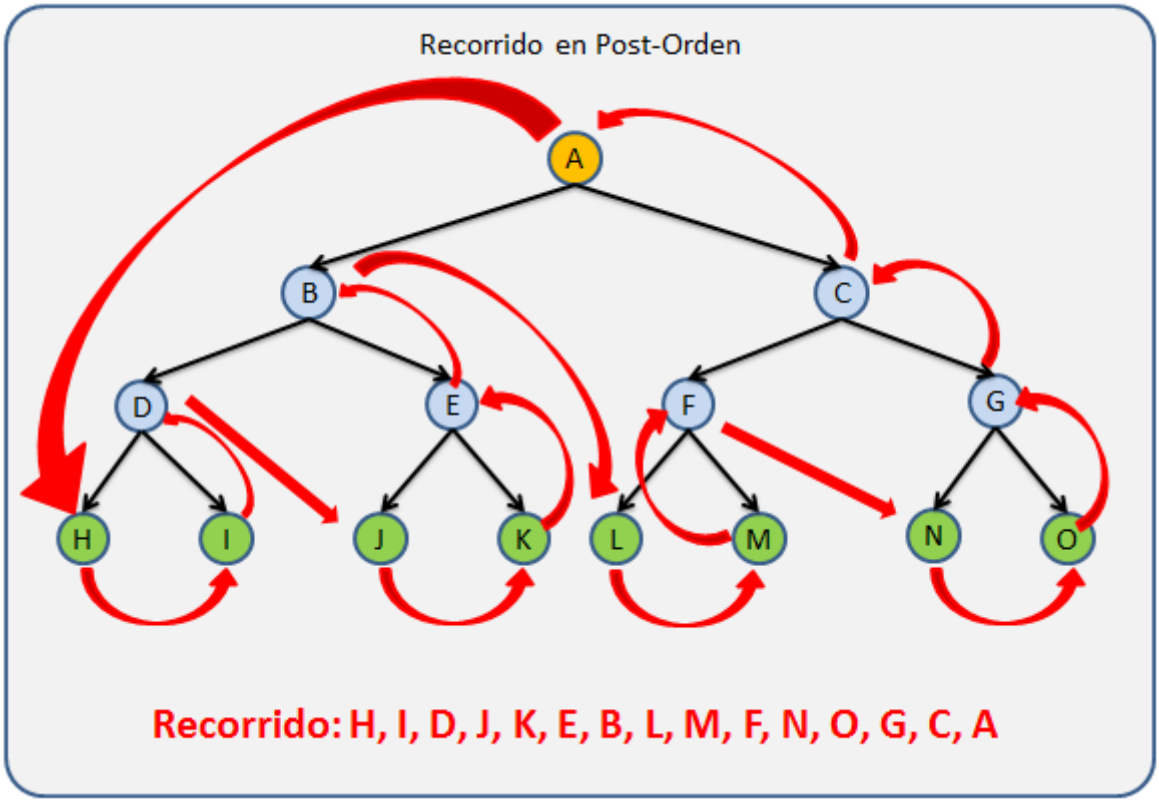
Se recorre en in-orden el primer sub-árbol, luego se recorre la raíz y al final se recorre en in-orden los demas sub-árboles.




```
// Implementación recursiva
void inorden(NodoArbol* raiz) {
    if (raiz != NULL) {
        preorden(raiz->obtenerHijoIzquierda());
        tratar(raiz);
        preorden(raiz->obtenerHijoDerecha());
    }
}
```

Recorrido posorden

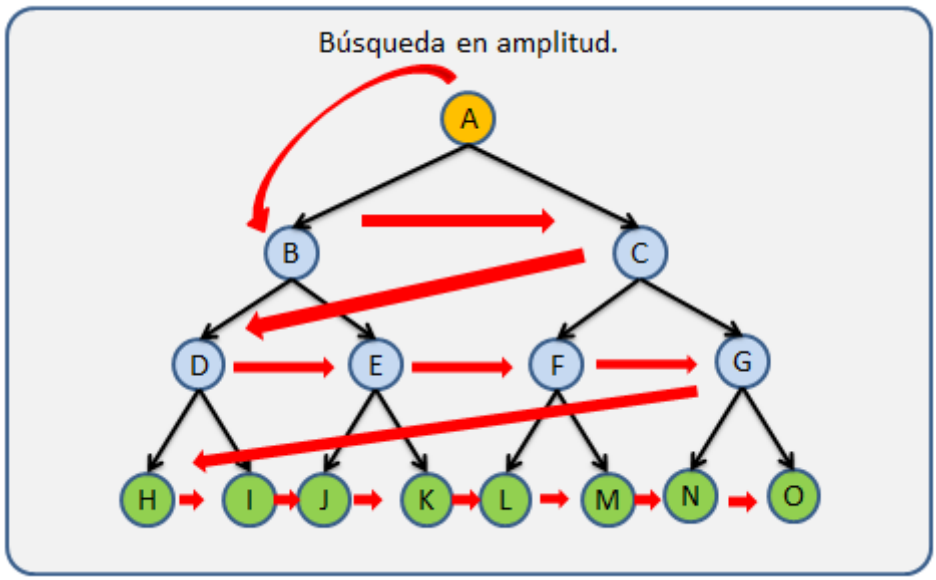
Se recorre el pos-orden cada uno de los sub-árboles y al final se recorre la raíz.




```
// Implementación recursiva
void postorden(NodoArbol* raiz) {
    if (raiz != NULL) {
        preorden(raiz->obtenerHijoIzquierda());
        preorden(raiz->obtenerHijoDerecha());
        tratar(raiz);
    }
}
```

Búsqueda en amplitud

Este tipo de recorrido no es recursivo, porque se recorre primero la raíz, luego se recorren los demas nodos ordenados por el nivel al que pertenecen en orden de Izquierda a derecha.



Para implementarlo se necesita una estructura auxiliar de tipo cola. Primero se encola la raíz, y mientras la cola no esté vacía voy desencolando y tratando los nodos. Si el nodo que estoy procesando tiene un hijo a izquierda, lo encolo, me fijo si tiene un hijo a derecha y lo encolo.

Arbol Binario de Búsqueda (ABB)

La característica que diferencia a los ABB de otro tipo de árboles es que siempre los elementos del subárbol izquierdo son menores que la raíz, y los del subárbol derecho mayores.

Nodo

```
template <typename Type>
class BSTNode {

    private:
        BSTNode<Type>* left;
        BSTNode<Type>* right;
        BSTNode<Type>* parent;
        Type key;

    public:
        BSTNode(Type key);
        BSTNode(Type key, BSTNode* left, BSTNode* right);
        Type getKey();
        BSTNode<Type>* getLeft();
        BSTNode<Type>* getRight();
        BSTNode<Type>* getParent();
        void setKey(Type key);
        void setLeft(BSTNode<Type>* left);
        void setRight(BSTNode<Type>* right);
        void setParent(BSTNode<Type>* parent);
        bool isLeaf();
        bool onlyRightChildren();
        bool onlyLeftChildren();
};

/* ------------------------------ Public Methods ------------------------------
- */
template <typename Type>
BSTNode<Type>:: BSTNode(Type key) {
    this->key = key;
    left = NULL;
    right = NULL;
    parent = NULL;
}

////////////////////////////////////////////
template <typename Type>
BSTNode<Type>:: BSTNode(Type key, BSTNode* left, BSTNode* right) {
    this->key = key;
    this->left = left;
    this->right = right;
}

////////////////////////////////////////////
```

```
template <typename Type>
Type BSTNode<Type>:: getKey() {
    return key;
}

////////////////////////////////////////////
template <typename Type>
BSTNode<Type>* BSTNode<Type>:: getLeft() {
    return left;
}

////////////////////////////////////////////
template <typename Type>
BSTNode<Type>* BSTNode<Type>:: getRight() {
    return right;
}

////////////////////////////////////////////
template <typename Type>
BSTNode<Type>* BSTNode<Type>:: getParent() {
    return parent;
}

////////////////////////////////////////////
template <typename Type>
void BSTNode<Type>:: setKey(Type key) {
    this->key = key;
}

////////////////////////////////////////////
template <typename Type>
void BSTNode<Type>:: setLeft(BSTNode<Type>* left) {
    this->left = left;
}


////////////////////////////////////////////
template <typename Type>
void BSTNode<Type>:: setRight(BSTNode<Type>* right) {
    this->right = right;
}

////////////////////////////////////////////
template <typename Type>
void BSTNode<Type>:: setParent(BSTNode<Type>* parent) {
    this->parent = parent;
}

////////////////////////////////////////////
template <typename Type>
bool BSTNode<Type>:: isLeaf() {
    return (right == NULL && left == NULL);
}

////////////////////////////////////////////
template <typename Type>
bool BSTNode<Type>:: onlyLeftChildren() {
    return (right == NULL && left != NULL);
```

```
}

//////////////////////////////////////////
template <typename Type>
bool BSTNode<Type>:: onlyRightChildren() {
    return (right != NULL && left == NULL);
}
/* ------------------------------------------------------------------------
- */
```

Árbol

```
#include "BSTNode.h"
using namespace std;

template <typename Type>
class BST {

    private:
        BSTNode<Type>* root;

    public:
        BST();
        ~BST();
        void insert(Type data);
        int getHeight();
        BSTNode<Type>* getRoot();
        BSTNode<Type>* getMaxNode(BSTNode<Type>* treeNode);
        BSTNode<Type>* getMinNode(BSTNode<Type>* treeNode);
        void deleteKey(Type key);
        void deleteAll();
        void balance();
        bool isBalanced();
        bool search(Type key);
        Type successor(Type key);
        Type predecessor(Type key);
        void inOrder();
        void preOrder();
        void postOrder();
        void displayData();

    private:
        BSTNode<Type>* insert(BSTNode<Type>* treeNode, Type key);
        void balance(BSTNode<Type>* treeNode);
        bool isBalanced(BSTNode<Type>* treeNode);
        int getHeight(BSTNode<Type>*treeNode);
        BSTNode<Type>* deleteKey(BSTNode<Type>* treeNode, Type key);
        BSTNode<Type>* deleteCaseLeaf(BSTNode<Type>* treeNode);
        BSTNode<Type>* deleteCaseOneChild(BSTNode<Type>* treeNode);
        BSTNode<Type>* deleteCaseTwoChilds(BSTNode<Type>* treeNode);
        void deleteAll(BSTNode<Type>* treeNode);
        BSTNode<Type>* search(BSTNode<Type>* treeNode, Type key);
        Type successor(BSTNode<Type>* treeNode);
        Type predecessor(BSTNode<Type>* treeNode);
        void inOrder(BSTNode<Type>* treeNode);
```

```
        void preOrder(BSTNode<Type>* treeNode);
        void postOrder(BSTNode<Type>* treeNode);
};

/* ------------------------------ Public Methods ----------------------------
- */

template <typename Type>
BST<Type>:: BST() {
    root = 0;
}

////////////////////////////////////////////////
template <typename Type>
BST<Type>:: ~BST() {
    deleteAll();
}

////////////////////////////////////////////////
template <typename Type>
void BST<Type>:: insert(Type data) {
    root = insert(root, data);
}

////////////////////////////////////////////////
template <typename Type>
int BST<Type>:: getHeight() {
    return getHeight(root);
}

////////////////////////////////////////////////
template <typename Type>
BSTNode<Type>* BST<Type>:: getRoot() {
    return root;
}

////////////////////////////////////////////////
template <typename Type>
BSTNode<Type>* BST<Type>:: getMaxNode(BSTNode<Type>* treeNode) {
    if (!root) {
        cout <<  "The BST is empty!" << endl;
        return NULL;
    }
    while(treeNode->getRight())
        treeNode = treeNode->getRight();
    return treeNode;
}

////////////////////////////////////////////////
template <typename Type>
BSTNode<Type>* BST<Type>:: getMinNode(BSTNode<Type>* treeNode) {
    if (!root) {
        cout <<  "The BST is empty!" << endl;
        return NULL;
    }
    while(treeNode->getLeft())
        treeNode = treeNode->getLeft();
```

```
    return treeNode;
}

////////////////////////////////////////////////
template <typename Type>
void BST<Type>:: deleteKey(Type key) {
    root = deleteKey(root, key);
}

////////////////////////////////////////////////
template <typename Type>
void BST<Type>:: deleteAll() {
    deleteAll(root);
}

////////////////////////////////////////////////
template <typename Type>
void BST<Type>:: balance() {
    if (!isBalanced())
        root = balance(root);
}

////////////////////////////////////////////////
template <typename Type>
bool BST<Type>:: isBalanced() {
    return isBalanced(root);
}

////////////////////////////////////////////////
template <typename Type>
bool BST<Type>:: search(Type key) {
    BSTNode<Type>* result = search(root, key);
    return result != NULL;
}

////////////////////////////////////////////////
template <typename Type>
Type BST<Type>::successor(Type key) {

    BSTNode<Type>* keyNode = search(root, key);

    if(keyNode == NULL)
        return -1;
    else
        return successor(keyNode);
}

////////////////////////////////////////////////
template <typename Type>
Type BST<Type>::predecessor(Type key) {

    BSTNode<Type>* keyNode = search(root, key);

    if(keyNode == NULL)
        return -1;
    else
        return predecessor(keyNode);
```

```
}

////////////////////////////////////////////////
template <typename Type>
void BST<Type>:: inOrder() {
    inOrder(root);
    cout << "\n";
}

////////////////////////////////////////////////
template <typename Type>
void BST<Type>:: preOrder() {
    preOrder(root);
    cout << "\n";
}

////////////////////////////////////////////////
template <typename Type>
void BST<Type>:: postOrder() {
    postOrder(root);
    cout << "\n";
}

////////////////////////////////////////////////
template <typename Type>
void BST<Type>:: displayData() {
    cout << "\tIn order: ";
    inOrder();
    cout << "\tPre order: ";
    preOrder();
    cout << "\tPost order: ";
    postOrder();

    cout << "\n\tThe root of this BST is: " << root->getKey() << "\n";
    cout << "\tThe height of this BST is: " << getHeight() << "\n";
    cout << "\tMax value: " << getMaxNode(root)->getKey() << "\n";
    cout << "\tMin value: " << getMinNode(root)->getKey() << "\n";

    if (isBalanced())
        cout << "\tBST is balanced! " << "\n\n";
    else
        cout << "\tBST is not balanced! " << "\n\n";

    cout << "_____________________________________\n";
}

/* ------------------------------------------------------------------------
- */


/* ----------------------------- Private Methods ---------------------------
- */

////////////////////////////////////////////////
template <typename Type>
BSTNode<Type>* BST<Type>:: insert(BSTNode<Type>* treeNode, Type key) {
    if (treeNode == NULL)
        treeNode = new BSTNode<Type>(key);
```

```
    else if (key > treeNode->getKey())
        treeNode->setRight(insert(treeNode->getRight(), key));

    else
        treeNode->setLeft(insert(treeNode->getLeft(), key));

    return treeNode;
}

////////////////////////////////////////////////
template<typename Type>
int BST<Type>:: getHeight(BSTNode<Type>* treeNode) {
    if (treeNode)
        return 1 + max(getHeight(treeNode->getLeft()), getHeight(treeNode-
>getRight()));
    else
        return 0;
}

////////////////////////////////////////////////
template<typename Type>
bool BST<Type>:: isBalanced(BSTNode<Type>* treeNode) {
    if (!treeNode)
        return false;

    int leftHeight = getHeight(treeNode->getLeft());
    int rightHeight = getHeight(treeNode->getRight());

    return abs(leftHeight - rightHeight) <= 1;
}

////////////////////////////////////////////////
template<typename Type>
BSTNode<Type>* BST<Type>:: deleteCaseLeaf(BSTNode<Type>* treeNode) {
    delete treeNode;
    return NULL;
}

////////////////////////////////////////////////
template<typename Type>
BSTNode<Type>* BST<Type>:: deleteCaseOneChild(BSTNode<Type>* treeNode) {

    if (treeNode->onlyRightChildren()) {
        BSTNode<Type>* rightNode = treeNode->getRight();
        rightNode->setParent(treeNode->getParent());
        BSTNode<Type>* aux = treeNode;
        delete aux;
        return rightNode;
    }

    else if (treeNode->onlyLeftChildren()) {
        BSTNode<Type>* leftNode = treeNode->getLeft();
        leftNode->setParent(treeNode->getParent());
        BSTNode<Type>* aux = treeNode;
        delete aux;
        return leftNode;
    }
```

```
}

////////////////////////////////////////////////
template<typename Type>
BSTNode<Type>* BST<Type>:: deleteCaseTwoChilds(BSTNode<Type>* treeNode) {
    Type successorKey = successor(treeNode->getKey());
    treeNode->setKey(successorKey);
    treeNode->setRight(deleteKey(treeNode->getRight(), successorKey));
    return treeNode;
}

////////////////////////////////////////////////
template<typename Type>
BSTNode<Type>* BST<Type>:: deleteKey(BSTNode<Type>* treeNode, Type key) {

    if (treeNode == NULL)
        return NULL;

    if(treeNode->getKey() == key) {

        if (treeNode->isLeaf())
            treeNode = deleteCaseLeaf(treeNode);

        else if (treeNode->onlyRightChildren() || treeNode->onlyLeftChildren())
            treeNode = deleteCaseOneChild(treeNode);

        else
            treeNode = deleteCaseTwoChilds(treeNode);
    }

    else if (treeNode->getKey() < key)
        treeNode->setRight(deleteKey(treeNode->getRight(), key));

    else
        treeNode->setLeft(deleteKey(treeNode->getLeft(), key));

    return treeNode;
}

////////////////////////////////////////////////
template <typename Type>
void BST<Type>:: deleteAll(BSTNode<Type>* treeNode) {
    if(treeNode != NULL) {
        deleteAll(treeNode->getLeft());
        deleteAll(treeNode->getRight());
        delete treeNode;
    }
}

////////////////////////////////////////////////
template <typename Type>
BSTNode<Type>* BST<Type>:: search(BSTNode<Type> *treeNode, Type key) {
    if (treeNode == NULL || treeNode->getKey() == key)
        return treeNode;
    if (key > treeNode->getKey())
        return search(treeNode->getRight(), key);
    return search(treeNode->getLeft(), key);
}
```

```
////////////////////////////////////////////////
template<typename Type>
Type BST<Type>:: successor(BSTNode<Type>* treeNode) {

    if (!treeNode->onlyRightChildren()) {
        BSTNode<Type>* minNode = getMinNode(treeNode->getRight());
        return minNode->getKey();
    }

    BSTNode<Type>* successor = NULL;
    BSTNode<Type>* predecessor = root;

    while (predecessor != treeNode) {
        if (treeNode->getKey() < predecessor->getKey()) {
            successor = predecessor;
            predecessor = predecessor->getLeft();
        }
        else
            predecessor = predecessor->getRight();
    }
    return successor->getKey();
}

////////////////////////////////////////////////
template<typename Type>
Type BST<Type>:: predecessor(BSTNode<Type>* treeNode) {
    if (!treeNode->onlyLeftChildren())
        return getMinNode(treeNode->onlyLeftChildren());

    BSTNode<Type>* successor = NULL;
    BSTNode<Type>* predecessor = root;

    while (predecessor != treeNode) {
        if (treeNode->getKey() < predecessor->getKey()) {
            successor = predecessor;
            predecessor = predecessor->getRight();
        }
        else
            predecessor = predecessor->getLeft();
    }
    return successor->getKey();
}

////////////////////////////////////////////////
template<typename Type>
void BST<Type>:: inOrder(BSTNode<Type>* treeNode) {
    if (treeNode) {
        inOrder(treeNode->getLeft());
        cout << treeNode->getKey() << " " ;
        inOrder(treeNode->getRight());
    }
}

////////////////////////////////////////////////////////////////////////////////
/////////
template<typename Type>
void BST<Type>:: preOrder(BSTNode<Type>* treeNode) {
```

```
    if (treeNode) {
        cout << treeNode->getKey() << " ";
        preOrder(treeNode->getLeft());
        preOrder(treeNode->getRight());
    }
}

////////////////////////////////////////////////////////////////////////////////
/////////
template<typename Type>
void BST<Type>:: postOrder(BSTNode<Type>* treeNode) {
    if (treeNode) {
        postOrder(treeNode->getLeft());
        postOrder(treeNode->getRight());
        cout << treeNode->getKey() << " ";
    }

}

/* ----------------------------------------------------------------------------
- */
```

AVL - ABB balanceado por altura

Los árboles AVL son un tipo ABB que están balanceados por su altura. Los AVL nos garantizan que para cada nodo, la diferencia entre la altura de sus hijos es **igual o menor a 1.**

Para implementar esto, hay que verificar si el árbol esta desbalanceado al momento de insertar o eliminar un nodo, y si esta desbalanceado balancearlo con rotaciones.

Cuando hablamos de un árbol balanceado o desbalanceado, estamos hablando del *factor de equilibrio/balance*, que es la diferencia entre las alturas del árbol izquierdo y el derecho.

FE = altura subarbol derecho - altura subarbol izquierdo

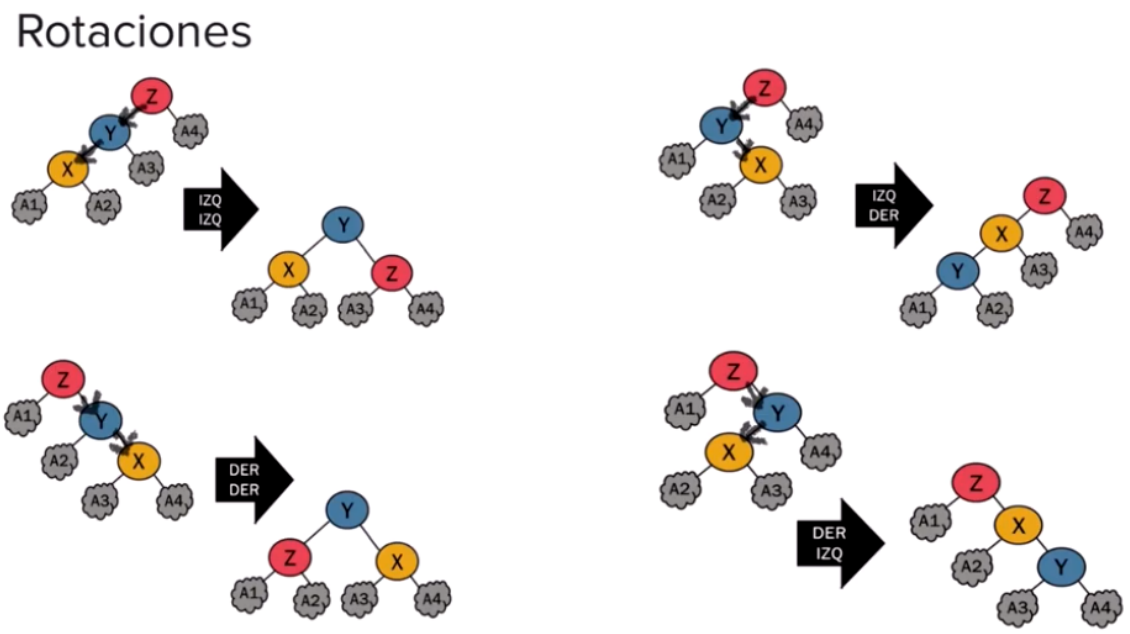
- Si FE < -1 => está cargado a la izquierda
- Si FE = 0 => está equilibrado
- Si FE > 1 => está cargado a la derecha

Recordemos que para que sea un AVL, FE debe ser **-1, 0 o 1**. Donde -1 es cargado a la izquierda, 0 es equilibrado y 1 es cargado a la derecha.

Las *rotaciones* pueden ser cuatro:

- Rotacion simple a la derecha
- Rotacion simple a la izquierda
- Rotacion doble a la derecha (consta de una rotación simple a la izquierda y luego una simple a la derecha)
- Rotacion doble a la izquierda (consta de una rotación simple a la derecha y luego una simple a la izquierda)

Gráficamente:

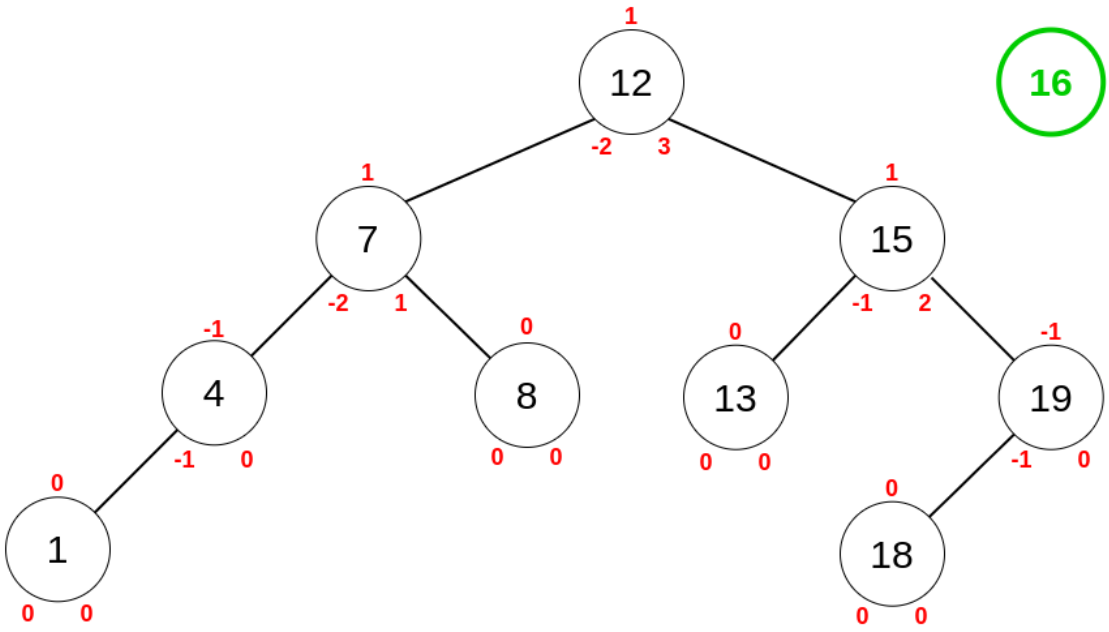


Insertar

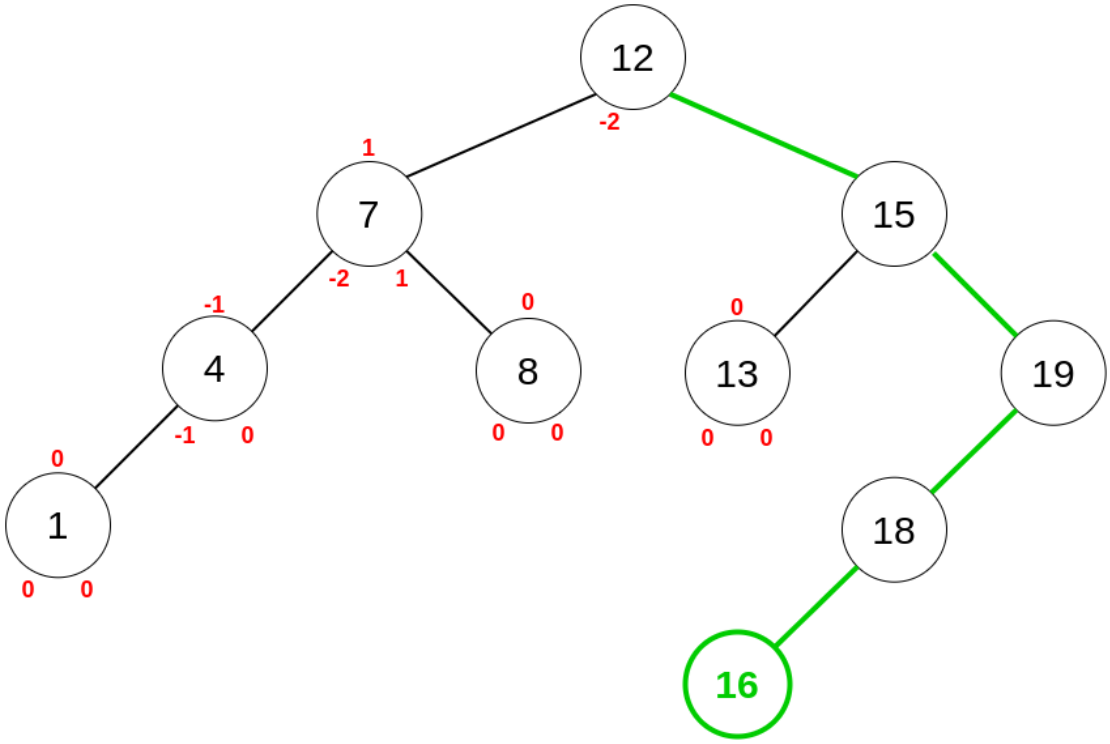
Al insertar un nuevo dato, hay que verificar que el árbol esté balanceado pero no sobre todo el árbol, sólo sobre el camino de inserción. Por ejemplo si tengo el siguiente árbol y quiero agregar el 16

Nota: hay dos errores en las imágenes:

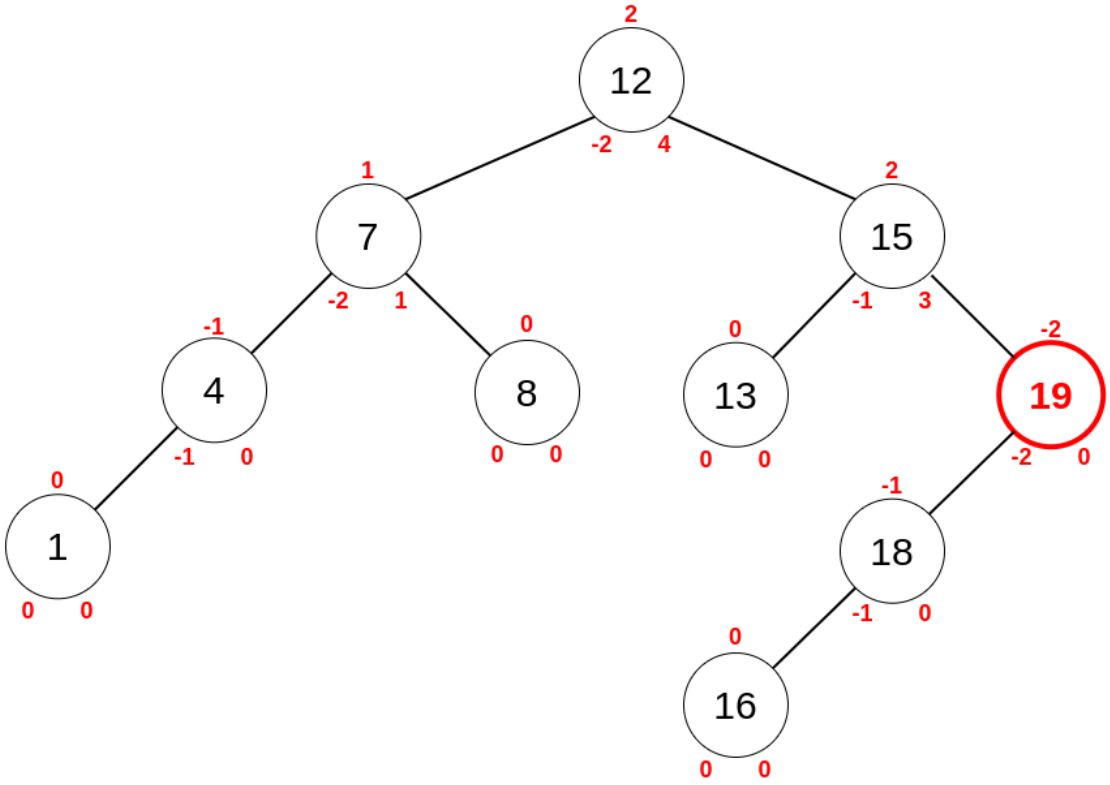
1. ***El FE del 7 debería decir -1 arriba, no 1***
2. **El FE de 12 debería decir -3 a la izquierda y 0 arriba**



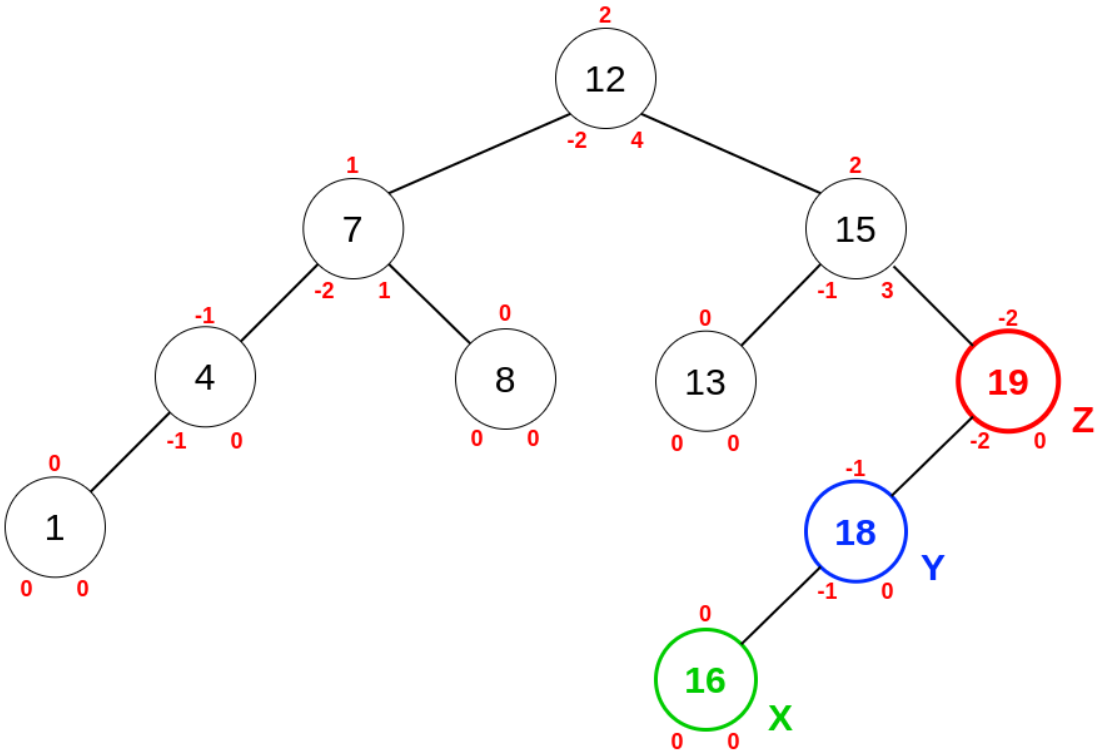
Quedaría en la siguiente posición, y lo que hay que verificar si quedó o no balanceado es el camino verde.



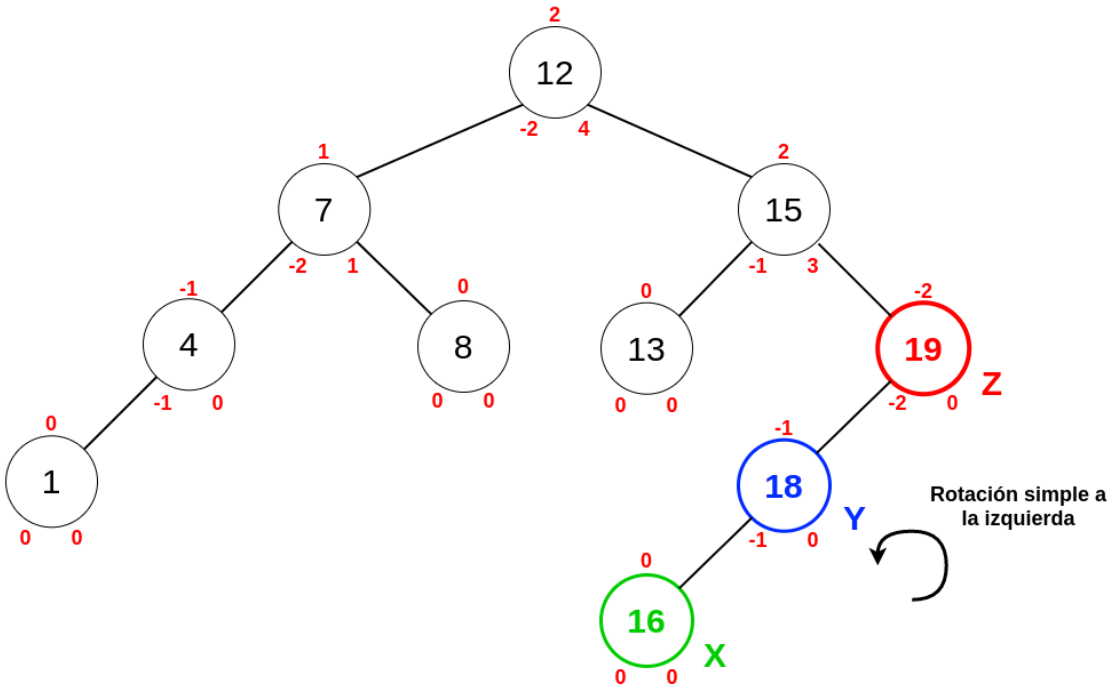
Chequeando nuevamente vemos que 16 está balanceado, 18 está balanceado, pero 19 no.



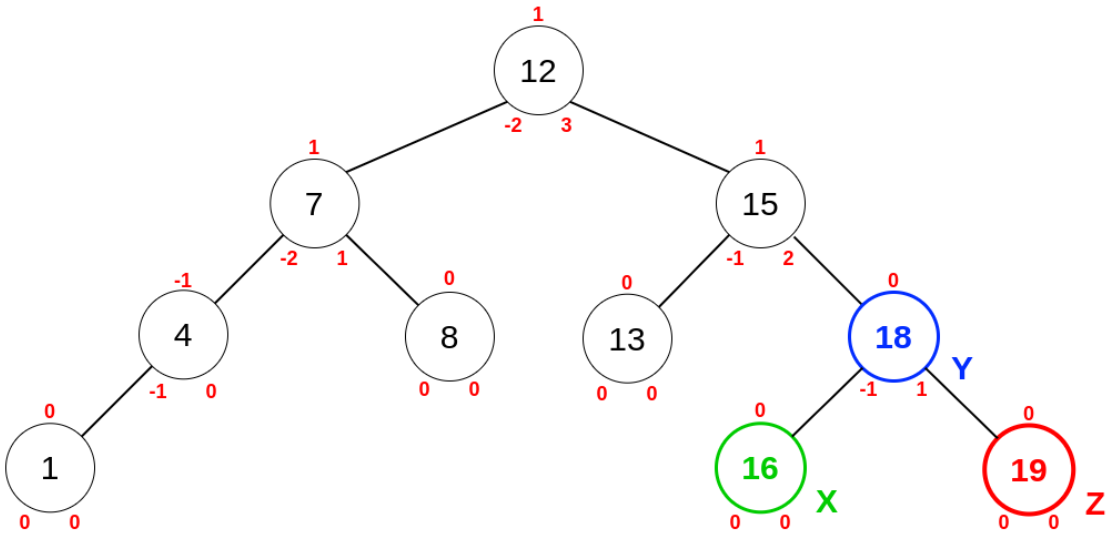
El 19 fue el nodo que rompio con la invariante del AVL, le pondremos el nombre Z, y luego siguiendo el camino de insercón le pondremos Y y X a los que siguen.



Lo que hay que analizar ahora, es qué rotación tendremos que hacer:



Y finalmente quedaría:

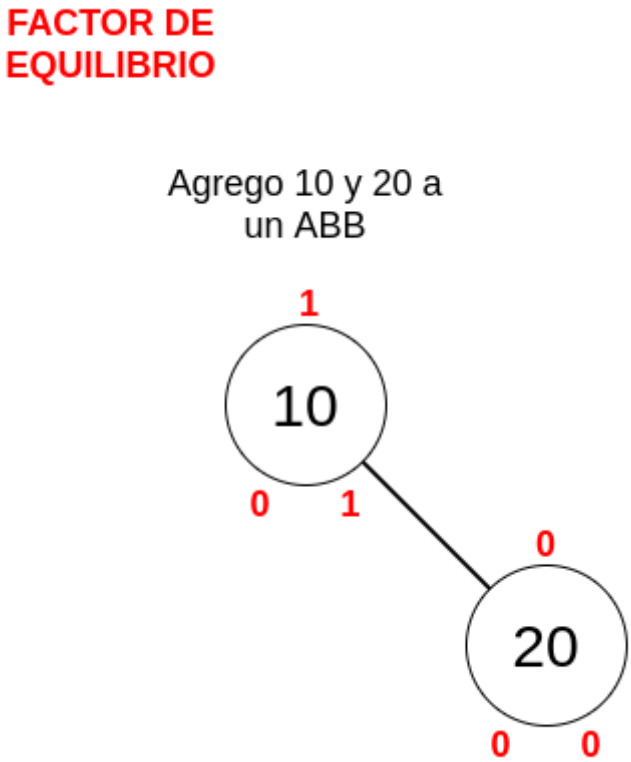


Seguimos analizando el camino de inserción, y vemos que el resto del árbol está balanceado así que no es necesario hacer ninguna otra operación.

Eliminar

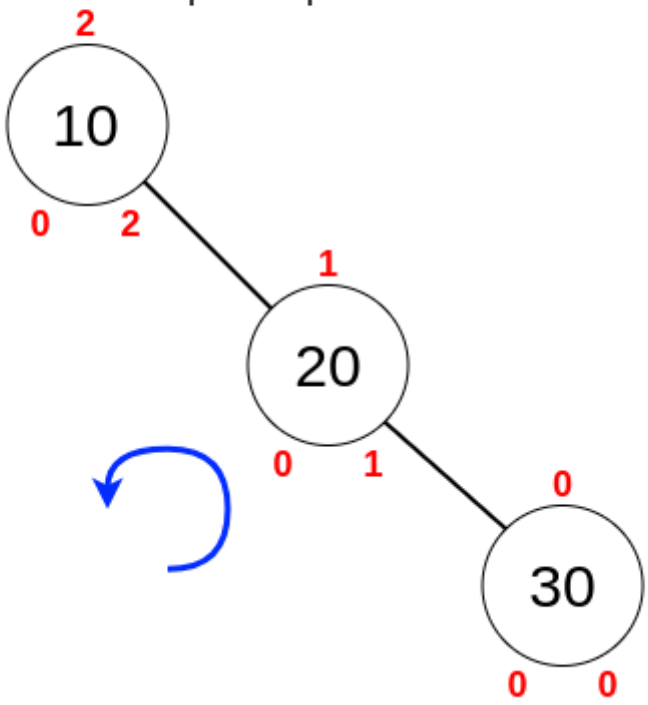
Para eliminar hay que hacer lo mismo que para insertar. Una vez que llegamos al nodo a eliminar, nos fijamos cual es el camino, y luego de eliminarlo verificamos recorriendo ese camino si quedó o no balanceado, realizando las operaciones necesarias en el caso de tener que balancear.

Ejemplo: quiero cargar en un ABB los siguientes números: 10, 20, 30, 25, 40, 50 y 60.



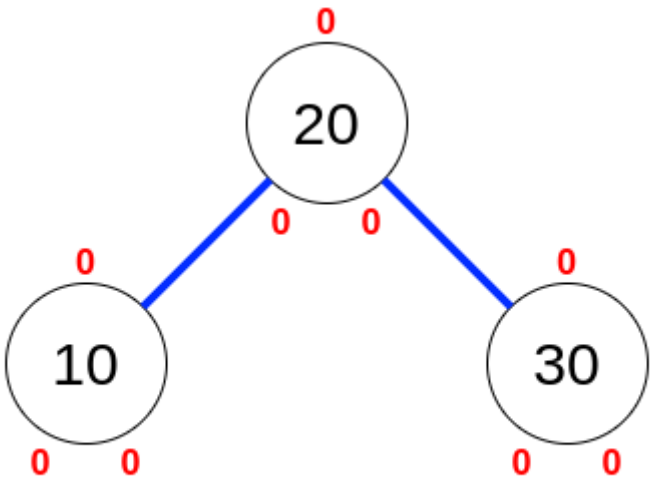
FACTOR DE EQUILIBRIO

Agrego 30, el árbol se desbalanceo. Hay que aplicar una rotación simple a la izquierda para balancear

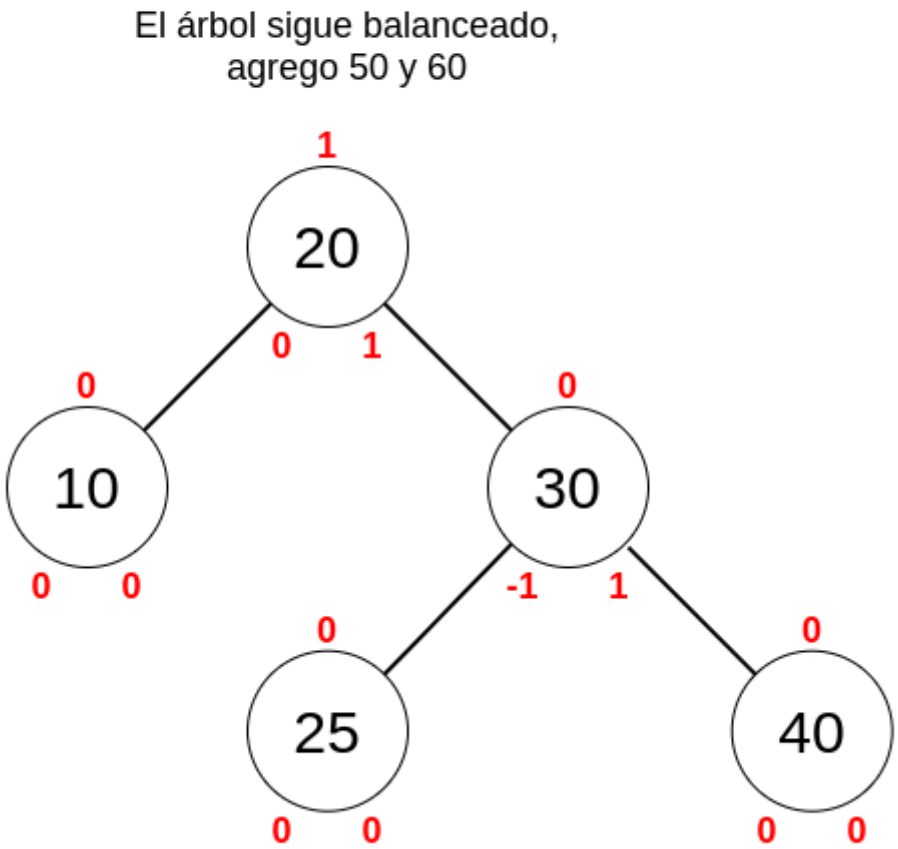


FACTOR DE EQUILIBRIO

Una vez balanceado, agrego 25 y 40



FACTOR DE EQUILIBRIO

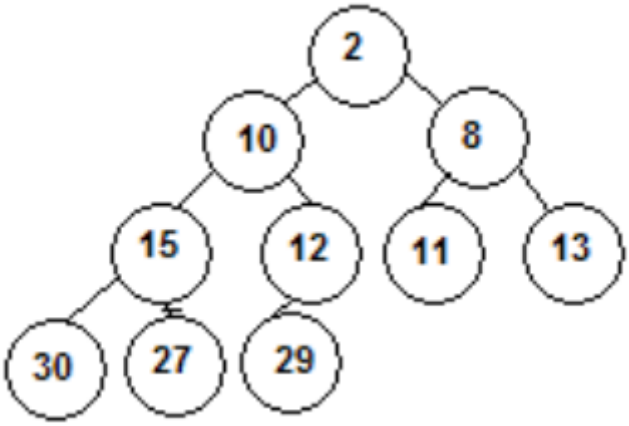


Heap

Un heap es un caso específico de los árboles binarios que cumple las siguientes propiedades:

1. El valor de cada nodo es mayor (en el caso de un heap máximo, si es un heap mínimo sería menor) al de sus hijos
2. El árbol está balanceado
3. El árbol está completo, y si no lo está los hijos están a la izquierda

Podemos decir que estos árboles están *parcialmente ordenados* por la propiedad 1, donde todos los nodos hijos son mayores (o menores) que el padre pero no hay ninguna restricción u orden entre los hijos.



En general los heaps se implementan con un array, donde se guardan los valores por niveles (como si fuera un recorrido en ancho).

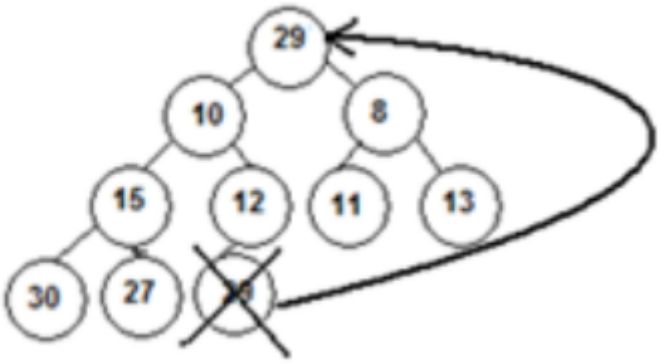
2	10	8	15	12	11	13	30	27	29

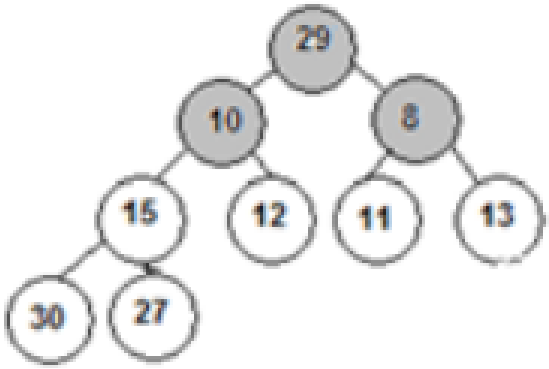
Podemos ver que la posicion 0 es la raiz, la 1 el hijo izquierdo y la 2 el hijo derecho. Generalizando podemos decir que si el padre está en la posicion *p* => el hijo izq. está en *2 p + 1* y el der. en *2p + 1*

Sacar raíz

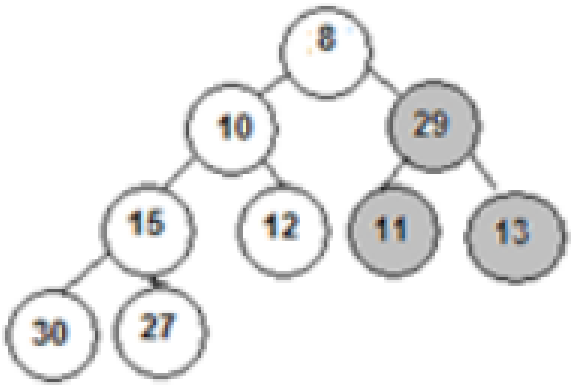
1. Se saca la raiz y reemplazandola por la última hoja
2. Se restaura el heap: se compara el valor de la nueva raíz con el de su hijo menor, y de ser necesario se realiza el intercambio. Luego se sigue comparando hacia abajo el valor trasladado desde la raíz hasta las hojas o hasta ubicar el dato en la posicion definitiva.

Gráficamente:

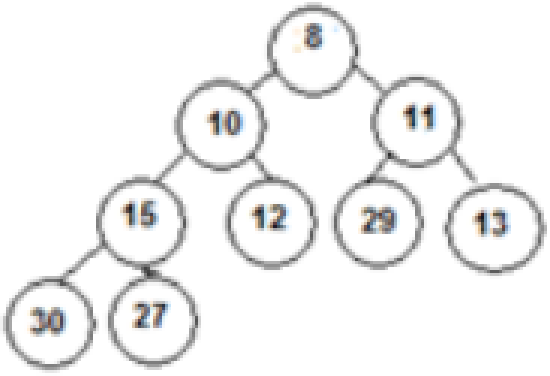




Ahora tenemos que ver el menor hijo del subárbol gris, que es el 8, por lo que se intercambia el 29 con él:



Nuevamente nos fijamos el menor hijo del subárbol gris, en este caso es el 11, así que se intercambia el 29 con él:



Ahora sí el heap está restaurado.

El costo es de O(log 2 n)

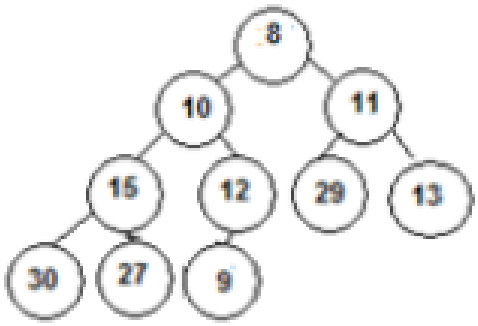
Agregar

El nuevo elemento siempre se inserta como la última hoja, y luego se restaura el heap analizando hacia arriba hasta ubicar el nuevo elemento en donde corresponda. Supongamos que al heap anterior quiero agregarle un 9

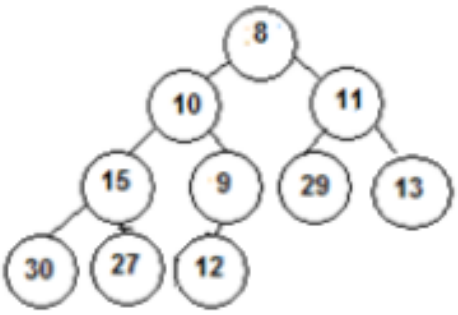
1. Queda como el hijo a la izquierda del 12 (en el mismo nivel que 30 y 27)
2. Analizamos el 9 con respecto a su padre, 12. En este caso 9 es menor así que se intercambian.
3. Analizamos el 9 con respecto a su nuevo padre, 10. Vuelve a ser menor 9 asi que volvemos a intercambiar los valores.

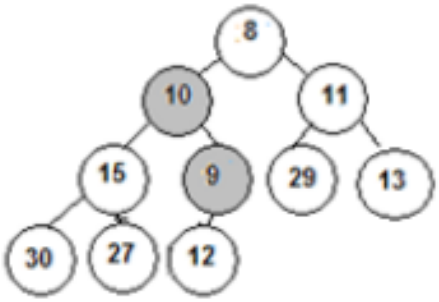
4. Analizamos el 9 con respecto a su nuevo padre, 8. Como se verifica la condicion del heap (los hijos son mayores al padre) no se realiza ningún intercambio y podemos decir que se restauró el heap

Situación inicial:

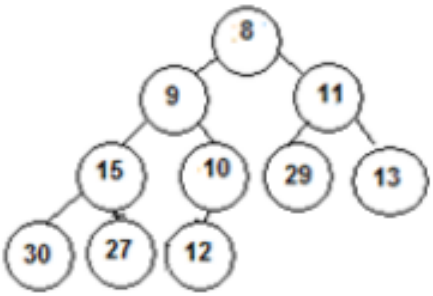


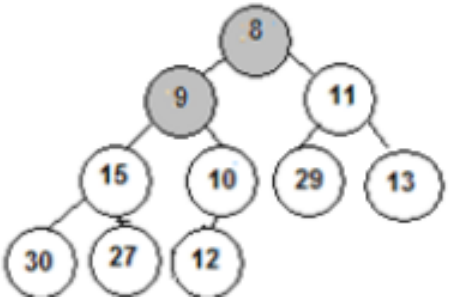
Se intercambia el 9 con el 12 y se analiza el 9 con el 10





Se intercambia el 9 con el 10 y se analiza el 9 con el 8



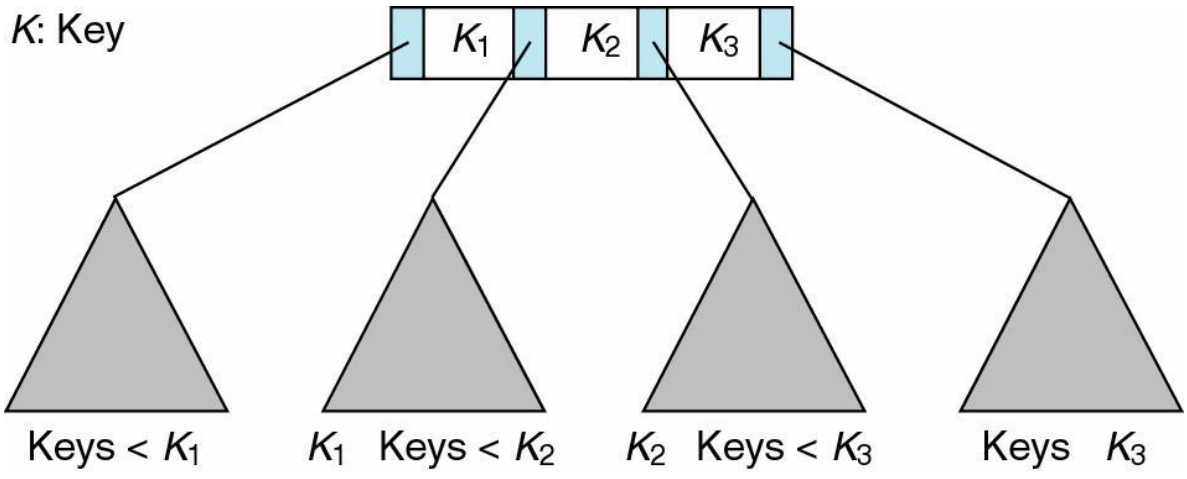


Ahora sí el heap está restaurado.

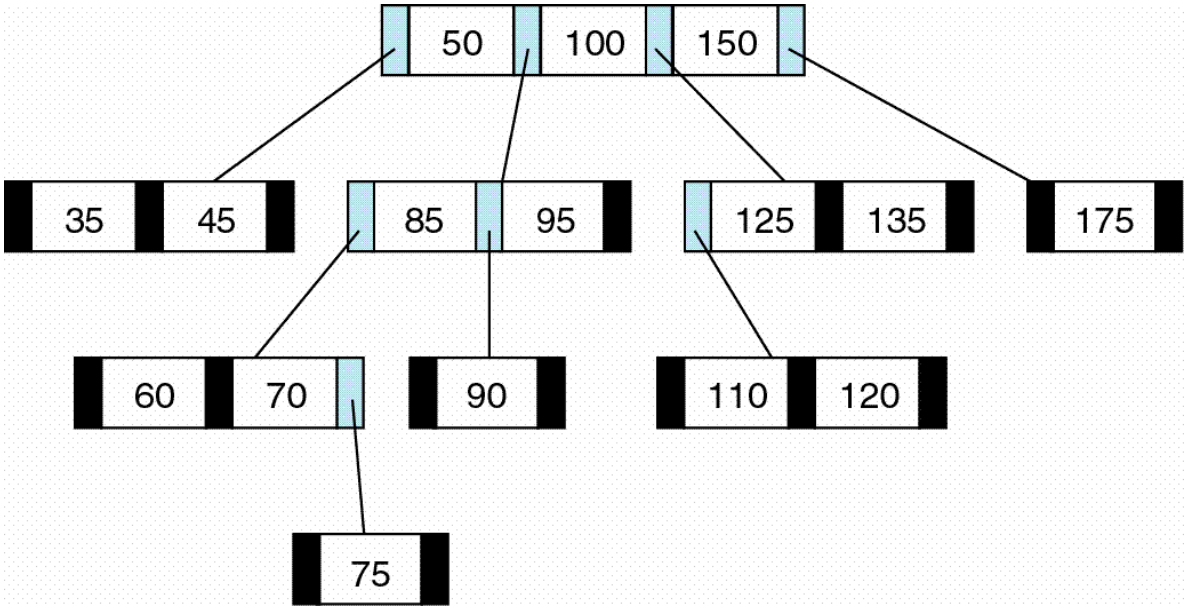
El costo es de O(log 2 n)

Multivia

Los ABB evolucionan a árboles de búsqueda de múltiples vías. Los nodos tienen a lo sumo m hijos y m-1 claves. Las claves dentro de cada nodo están ordenadas de menor a mayor en espacio contiguos, la forma gráfica de verlos sería algo así:



Y un ejemplo más puntual podría ser así:



B

Los árboles multivías B son un tipo particular de árboles multivía de búsqueda que están balanceados. Para hacer esto, se arman de abajo hacia arriba, y cumplen las siguientes condicines:

- Todos los nodos excepto la raíz están completos con claves al menos hasta la mitad
- La raiz es hoja o tiene al menos dos hijos
- Si el nodo tiene n claves, tiene n+1 hijos
- Todas las hojas están en el mismo nivel

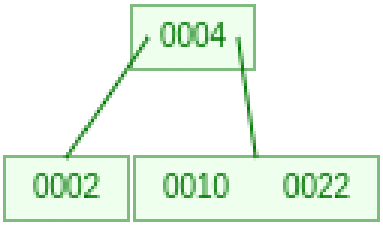
El grado del árbol lo determina el programador, lo recomendable es 5 por ser el más eficiente.

Ejemplo

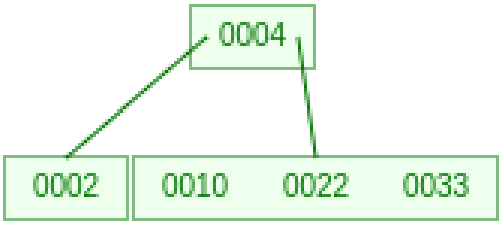
Quiero crear un arbol de grado 4 con las siguientes claves: 10 - 22 - 4 - 2 - 33 - 45 - 5 - 12 - 29 - 11

Al insertar los primeros valores, quedaría:

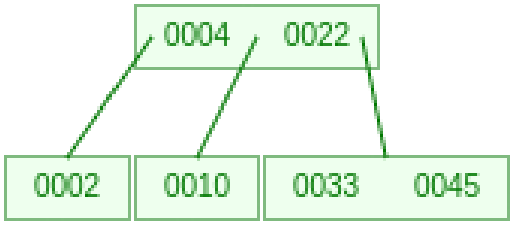
Al agregar el 2, la raiz se divide y bajan los valores a la derecha:



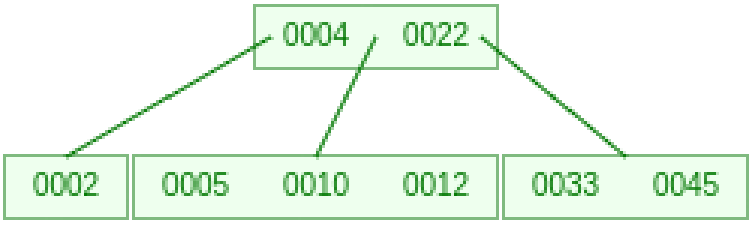
Agrego el 33, no pasa nada simplemente se agrega a la derecha del 22:



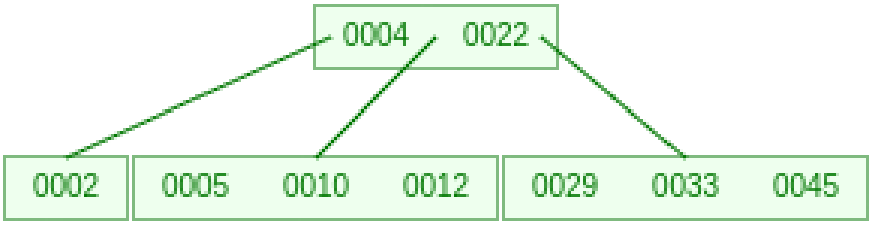
Agrego el 45, sube el 22 y se divide el 10 del 33 y el 45:



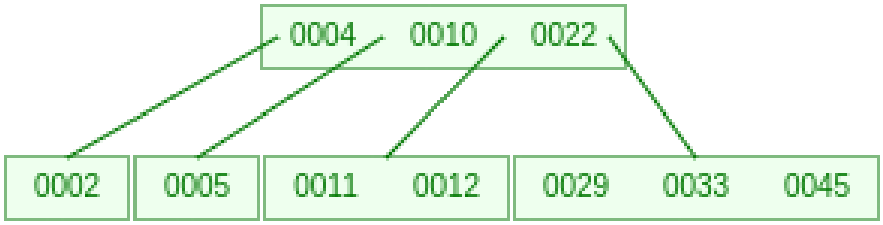
Agrego el 5 y el 12, no pasa nada simplemente se agrega a la izquierda del 10 el 5 y a la derecha el 12:



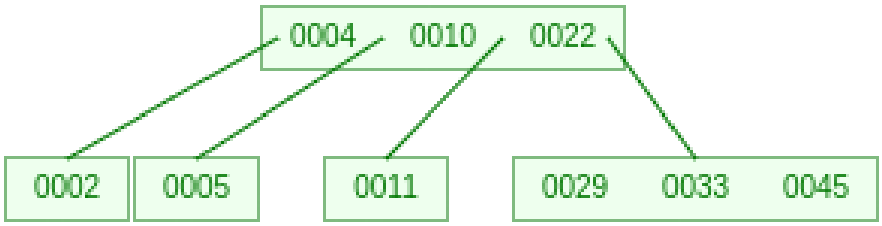
Agrego el 29, no pasa nada, simplemente se agrega a la izquierda del 33:



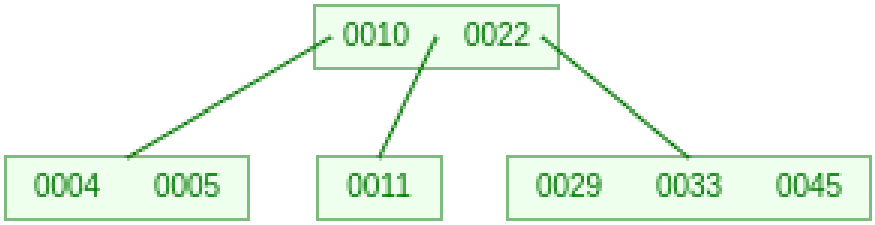
Agrego el 11, sube el 10, queda el 5 separado a la izquierda, y el 11 con el 12:



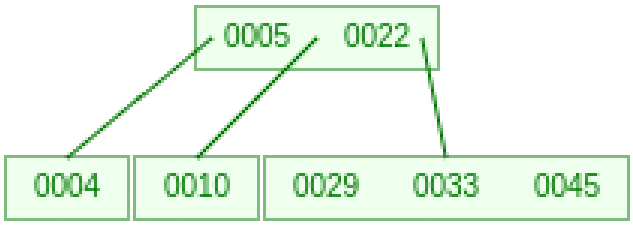
Ahora supongamos que empiezo a sacar valores. Primero saco el 12 y no pasa nada, queda:



Saco el 2, y baja el 4 al lado del 5:



Si ahora saco el 11, baja el 10 y sube el 5:



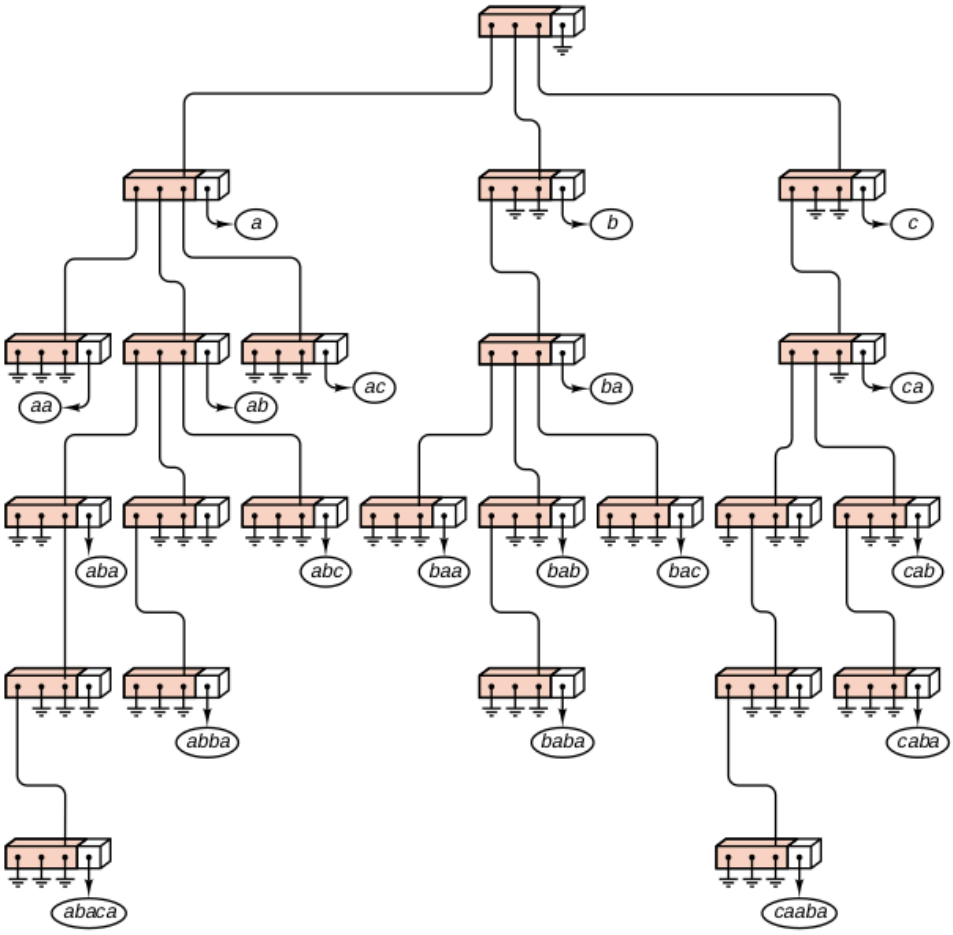
Trie

Es una estructura de datos que almacena un conjunto de claves de tipo string. Si bien hay muchas formas distintas de plasmarlo, la que vamos a ver es con un árbol donde cada nodo representa un carácter de una palabra insertada en el trie y los nodos hijos son caracteres que aparecen después de él en la palabra.

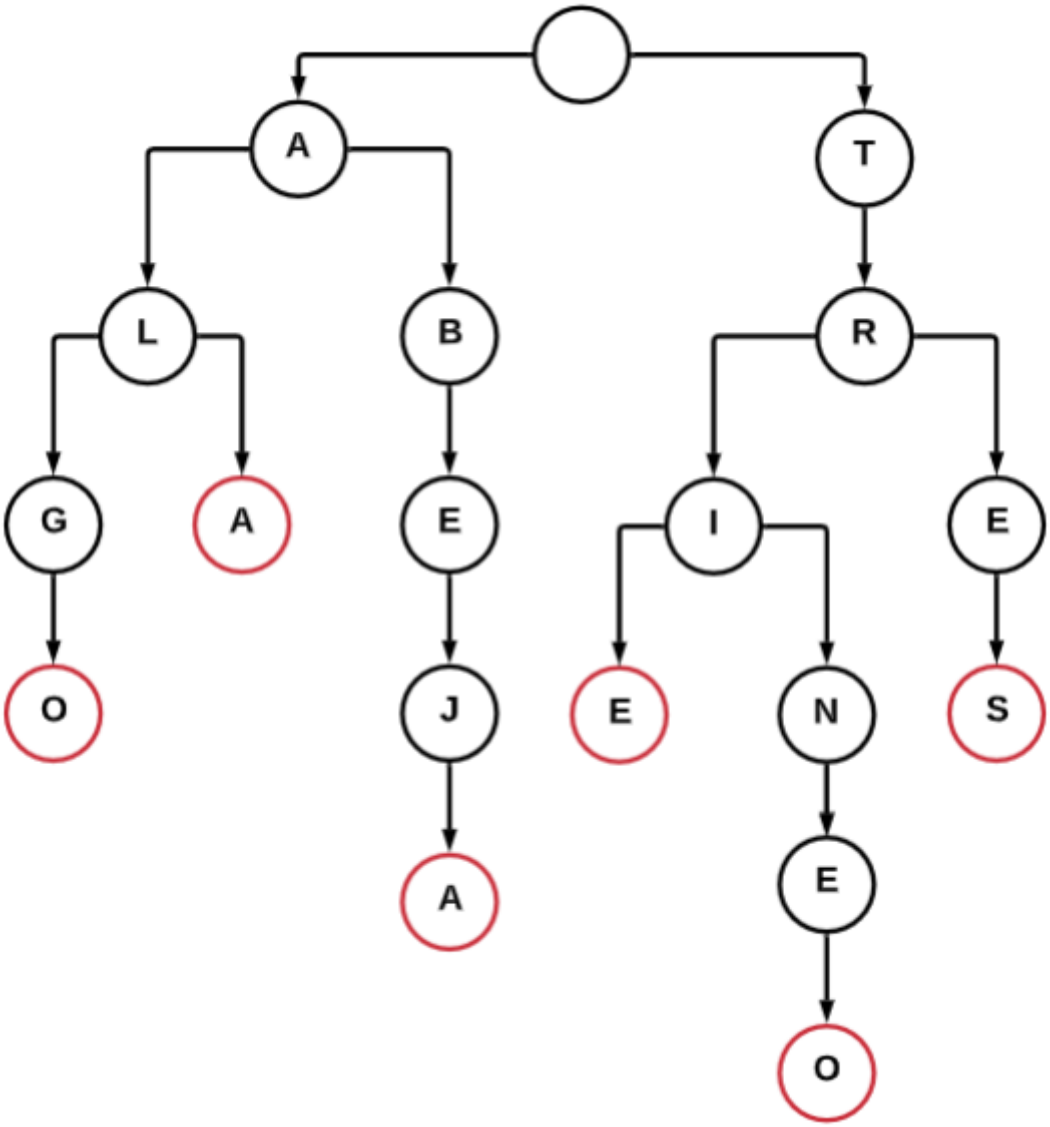
Cosas importantes a tener en cuenta:

- La raíz es en general el caractér vacío, podemos considerar que éste aparece al inicio de cualquier palabra
- Si el alfabeto tiene k símbolos cada nodo del trie tiene k+1 punteros (el extra es por si guarda una palabra)

En este caso el alfabeto tiene 3 letras: a, b y c. Las claves formadas son: aa, ab, aba, abc, abba, abaca, ac, b, ba, baa, bab, baba, bac, c, ca, caaba, cab y caba.



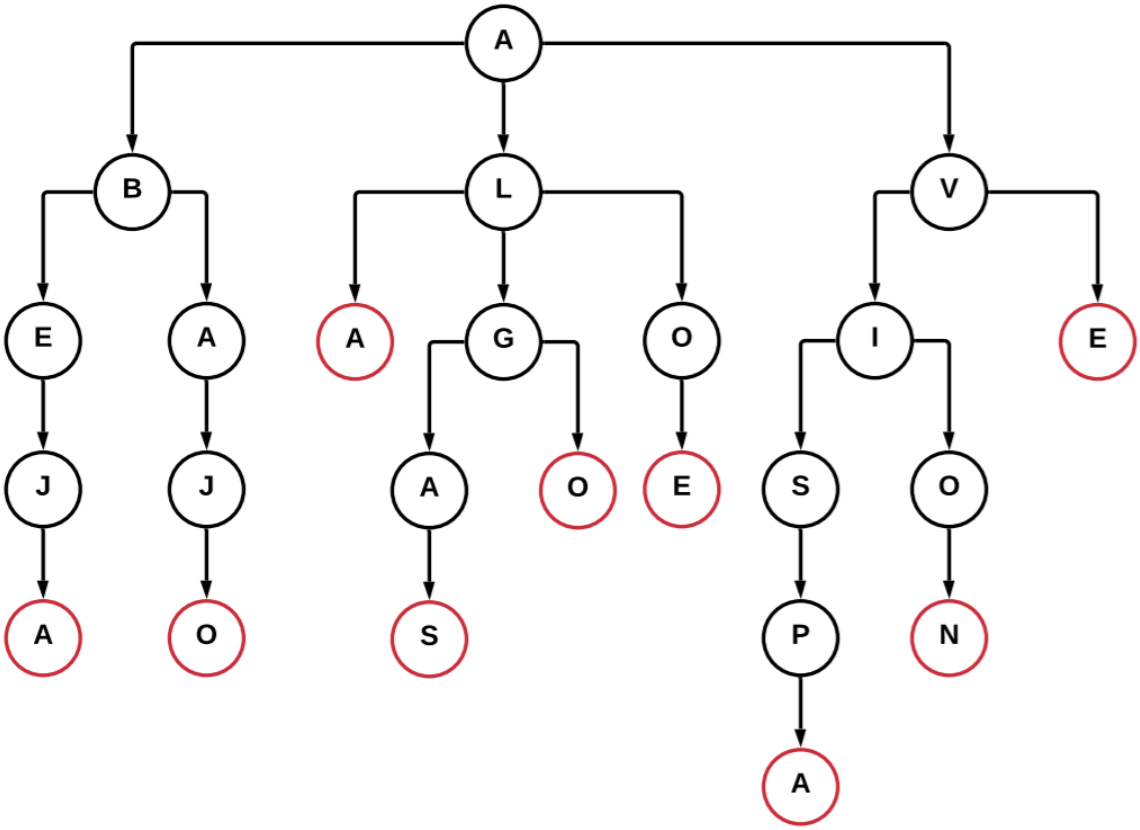
En este caso se simplificó el diagrama porque hay un total de 11 símbolos, pero la estructura de los nodos es similar. Las claves son: algo, ala, abeja trie, trineo y tres.



La desventaja que tienen estas estructuras es el costo espacial, porque hay muchos nodos con punteros a vacío, como se ve en la primera imagen. Sin embargo algunas de las ventajas son: nos permite imprimir todas las palabras en orden alfabético fácilmente y optimiza el tiempo de búsqueda de una clave, que en el peor caso si la clave es de longitud n el costo será de O(n).

Ternary Search Trie

El objetivo es reducir la cantidad de memoria que utiliza un trie, para eso cada nodo tiene hasta 3 punteros y un símbolo. El símbolo o la cadena que representa la raíz es común para todas las siguientes cadenas. El costo en el peor caso si la clave es de longitud n será de O(n)



Este TST tiene las palabras: abeja, abajo, ala, alga, algas, algo, aloe, avispa, avion y ave