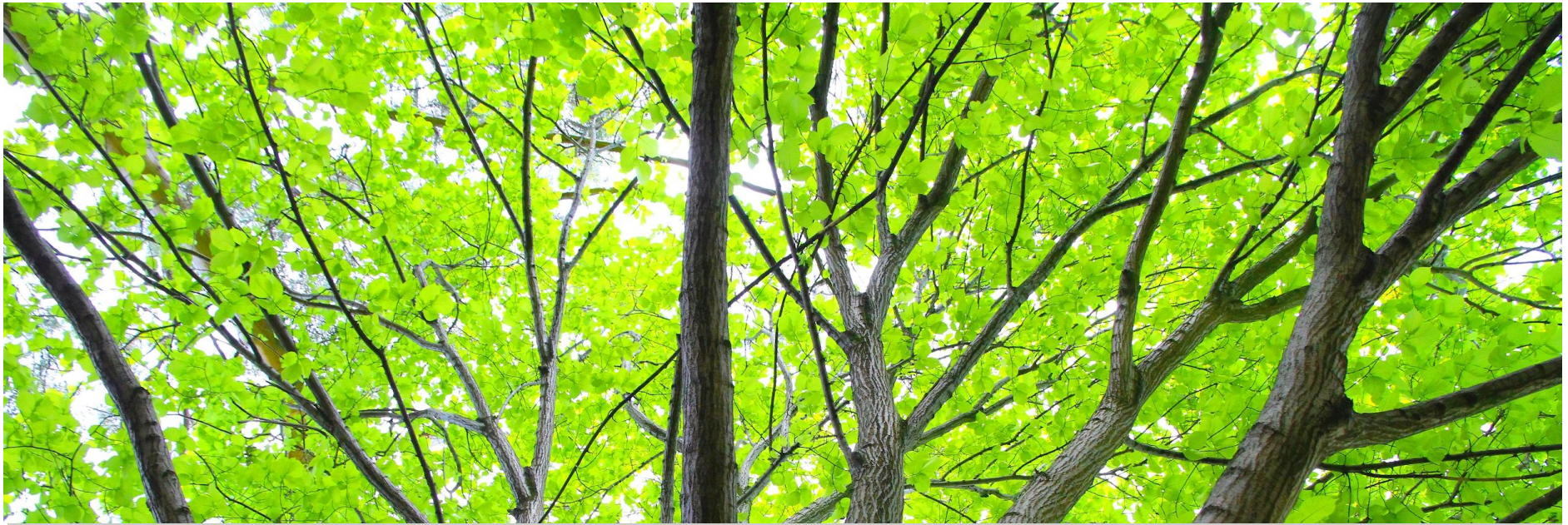


Búsqueda

AED I

2020

UBP UNIVERSIDAD BLAS PASCAL

Búsqueda

búsqueda

Supongamos que tenemos que buscar un nombre de usuario en un arreglo de 1000 usuarios

búsqueda secuencial

Una forma posible es **recorrer el array secuencialmente** y **comparar** cada elemento con el elemento que buscamos.

mejor caso: 1 comparación
peor caso: n comparaciones
caso promedio: (n+1) / 2

Si tenemos 1000000 de registros, esto es bastante.

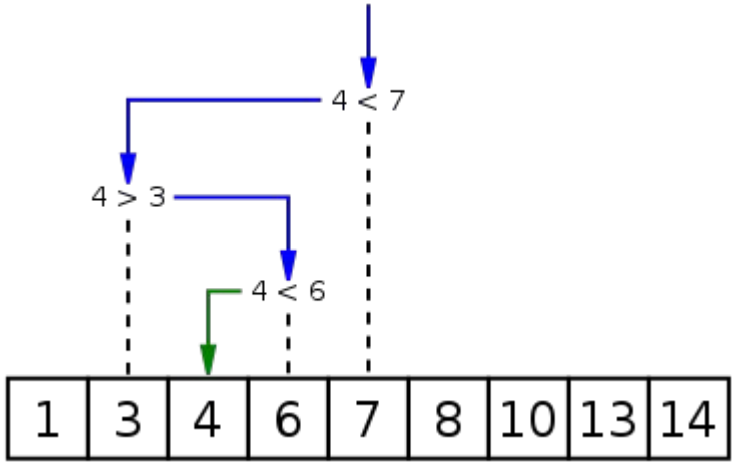
Si no sabemos nada acerca de la organización de los elementos en el arreglo, no podemos mejorarlo mucho

búsqueda binaria

Pero si los elementos en el arreglo están ordenados.

- Si comparamos el elemento que buscamos con el elemento en la mitad del arreglo.
- Dependiendo de si es mayor o menor que este elemento podemos descartar la mitad del arreglo en la que no está el elemento que buscamos.
- Realizamos la misma comparación en la mitad resultante.
- Seguimos hasta que resulta ser el elemento seleccionado igual al elemento que buscamos.

búsqueda binaria



búsqueda binaria

Si tenemos un arreglo con 1000 elementos: ¿Cuál es el máximo número de comparaciones que haremos?

Empezamos con 1000, después reducimos a 500, 250, 125, 62, 31, 15, 7, 3, 1
En total, necesitamos como máximo **10** comparaciones.

$$\log_2 1000 = 10$$

búsqueda binaria

¿Qué tanto más eficiente es la búsqueda binaria comparada con la búsqueda lineal?

Mucho

Supongamos que tenemos 1.000.000.000 de elementos, necesitamos solo 27 ($\log_2$ 1.000.000.000) comparaciones para encontrar un valor.

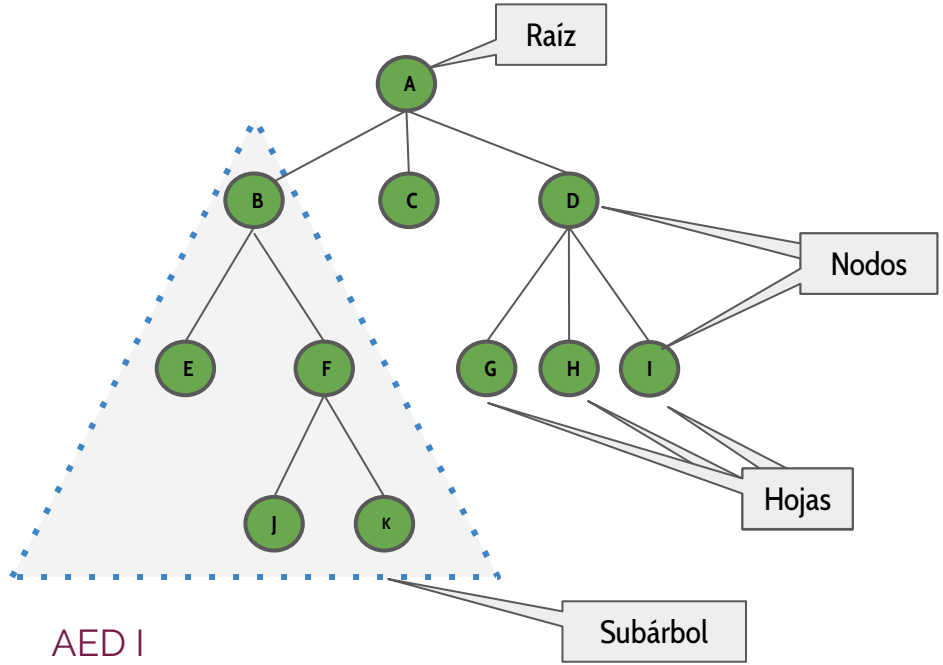
Ejercicio 5.01.

Crear un programa que implemente búsqueda binaria en un arreglo

Árboles

árbol: concepto y elementos

A diferencia de los arrays, las listas enlazadas, las pilas y las colas, que son estructuras lineales, en las que los datos están organizados secuencialmente. Los **árboles** son **estructuras de datos jerárquicas**



G, H e I son **hijos** de D

F es **padre** de J y K

Cada árbol tiene **subárboles**

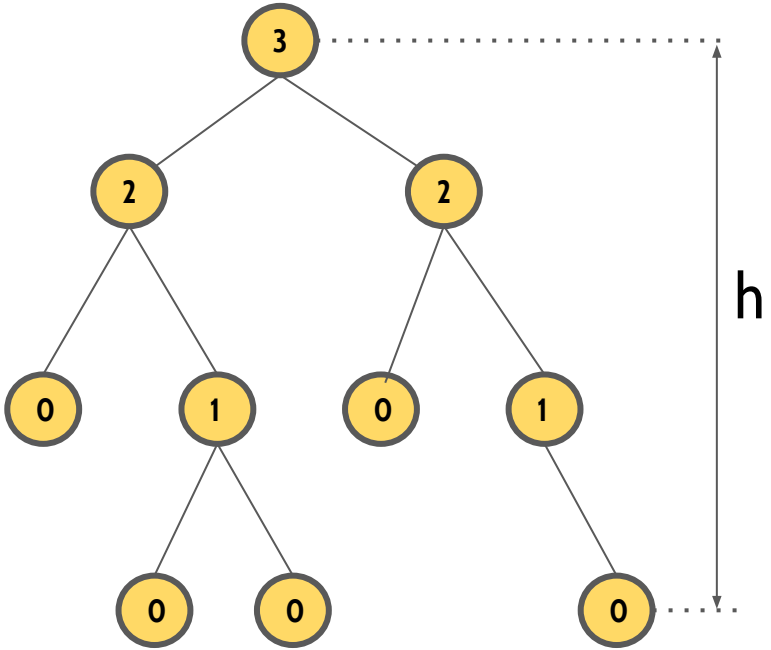
Un nodo n1 es un **ancestro** de un nodo n2 (y n2 es un **descendiente** de n1) si n1 es el padre de n2 o el padre de algún **ancestro** de n2.

árbol: uso

Entre algunos de los usos de los árboles se incluyen:

- Manipular información jerárquica
- Facilitar la búsqueda
- Administrar listas de datos ordenados
- Algoritmos de enrutamiento
- Representación de estrategias de decisión en múltiples pasos (ajedrez)

altura (h)

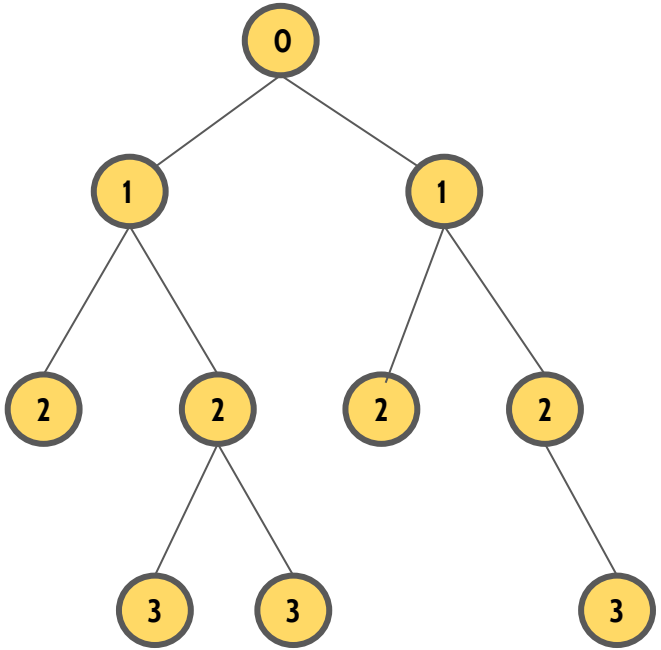


La **altura de un nodo** en un árbol se define como la **longitud del camino más largo** que comienza en el nodo y termina en una hoja

1. La altura de un nodo hoja es 0.
2. La altura de cualquier nodo es igual a la mayor altura de sus hijos más 1.

La **altura** de un árbol es la altura de la raíz.

profundidad o nivel

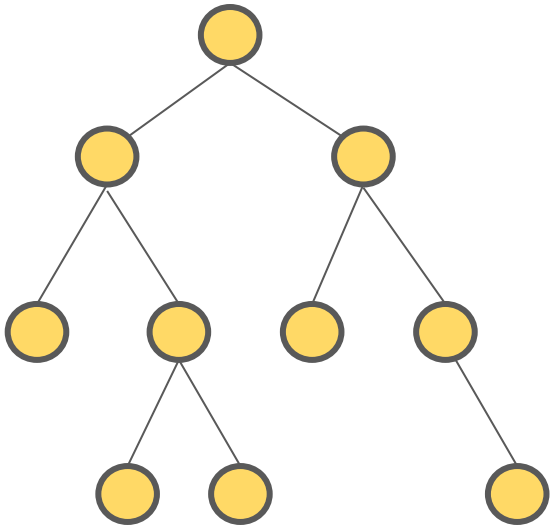


El **nivel** de un nodo en un árbol se define como:

1. La raíz del árbol tiene el nivel 0.
2. El nivel de cualquier otro nodo en el árbol es uno más que el nivel de su padre.

La **profundidad o nivel** de un árbol es el máximo nivel de cualquier hoja en el árbol.

Árbol binario



árbol binario

Cada nodo puede tener **2 hijos** como máximo.

Se conoce el nodo de la izquierda como **hijo izquierdo** y el nodo de la derecha como **hijo derecho**.

árbol binario lleno, completo, balanceado

Árbol lleno: todos los nodos del árbol tienen o ningún hijo o todos sus hijos.

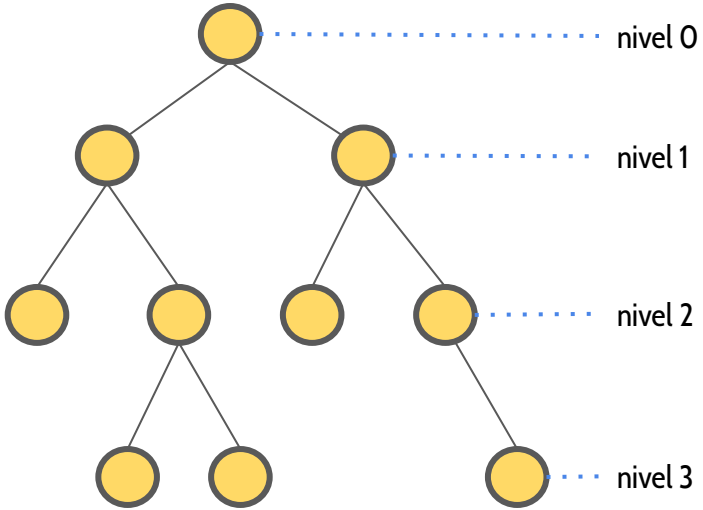
Árbol completo: es un árbol con todos sus niveles llenos salvo quizás el último, que deberá estar completo, (sin "huecos") de izquierda a derecha.

Árbol perfectamente balanceado: es aquel en el que el número de nodos de cada subárbol de cada nodo interno, no varía en más de uno.

Árbol balanceado en altura: es el árbol cuyos subárboles tienen alturas que difieren a lo más en una unidad y también son equilibrados en altura.

Árbol degenerado: aquél en el que cada nodo sólo tiene un subárbol. Equivale a una lista

costo de las operaciones



La cantidad máxima de nodos en un árbol de nivel i será n = $\mathbf{2^{(i+1)} - 1}$

Un árbol binario puede tener $\mathbf{n^i}$ nodos como máximo en el nivel **i**

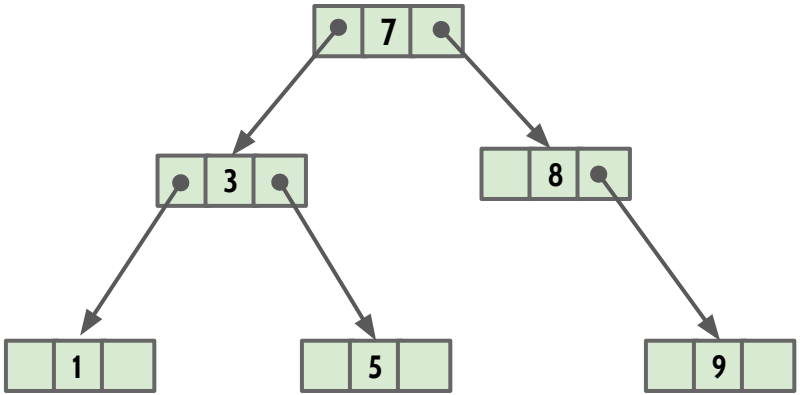
En general, todo árbol binario con **n** nodos tiene una altura máxima de **n** y una mínima de $\mathbf{\log_2(n+1)}$.

El **costo** de la mayoría de las operaciones está en función de la **altura**. Por ejemplo: en un árbol binario de búsqueda, el costo de buscar, insertar o remover un nodo es **O(h)**

Ese costo es mínimo si el árbol está **balanceado**, es decir si sus subárboles tienen alturas que difieren como máximo en 1

UBP UNIVERSIDAD BLAS PASCAL

implementación



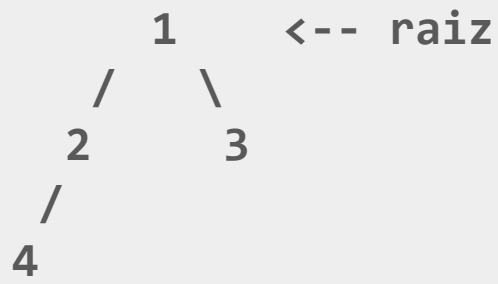

```
struct nodoAbb {
    int dato;
    nodoAbb* izquierdo;
    nodoAbb* derecho;
}
```

implementación

Pensá cómo implementarías un árbol en el que un nodo puede tener cualquier cantidad de hijos.

Ejercicio 5.02.

Escribir un programa que defina la estructura necesaria para representar un árbol binario y cree un árbol como el siguiente.



Ejercicio 5.02.

```
nodo* crearNodo(int dato){
    nodo* n = new nodo;
    n->dato = dato;
    n->izquierdo = NULL;
    n->derecho = NULL;
    return n;
}
```

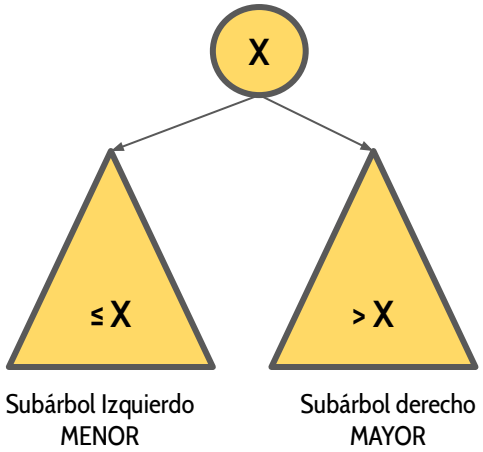
```
int main(){
    nodo* arbol = NULL;

    arbol = crearNodo(1);
    arbol->izquierdo = crearNodo(2);
    arbol->derecho = crearNodo(3);
    arbol->izquierdo->izquierdo = crearNodo(4);

    return 0;
}
```

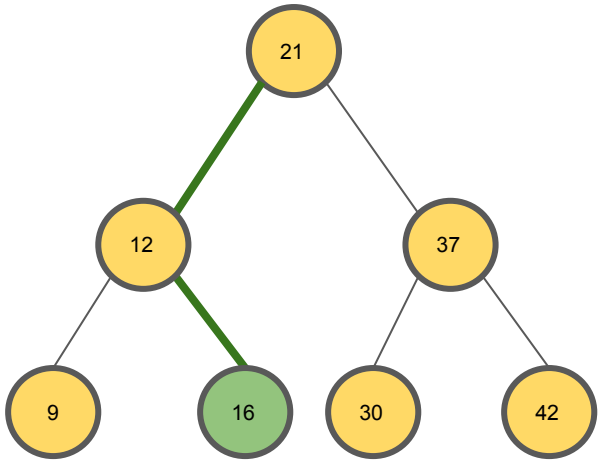
Árbol binario de búsqueda ABB

árbol binario de búsqueda

Un árbol binario de búsqueda (ABB) (BST en inglés) **no tiene valores duplicados en los nodos** y además:

1. Los valores en cualquier **subárbol izquierdo** son **menores** que el valor en su nodo padre.
2. Los valores en cualquier **subárbol derecho** son **mayores** que el valor en su nodo padre.



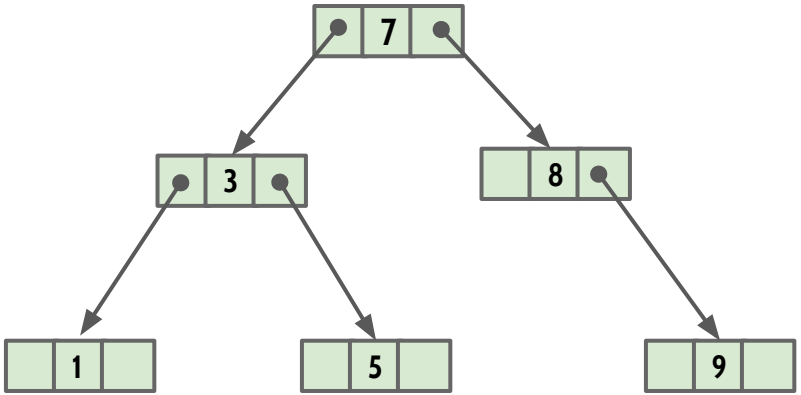


comparación con otras estructuras

Eficiencia en las operaciones

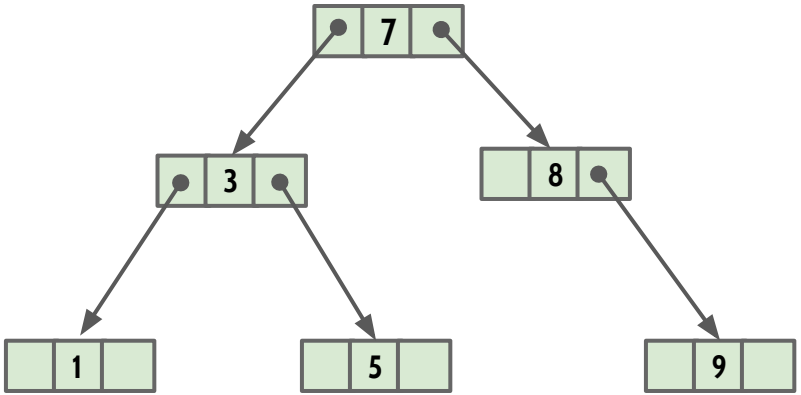
	Arreglo (desordenado)	**Lista enlazada**	**Arreglo (ordenado)**	**Árbol binario de búsqueda**
buscar(x)	O(n)	O(n)	O(log n)	O(log n)
insertar(x)	O(1) (al final)	O(1) (al principio)	O(n)	O(log n)
remover(x)	O(n)	O(n)	O(n)	O(log n)

abb: nodo




```
struct nodoAbb {
    int dato;
    nodoAbb* izquierdo;
    nodoAbb* derecho;
}
```

abb: clase Abb




```
class Abb {
    private:
     nodo* raiz;
     nodo* crearNodo(int);
     nodo* insertarAbb(int, nodo*);
    public:
     Abb();
     ~Abb();
     void insertar(int);
};
```

Ejercicio 5.03.

Escribir un programa que permita crear una clase que represente un árbol binario de búsqueda ABB e insertar nodos en el árbol.

Escriba el código para realizar una prueba, creando una o más instancias de la clase e insertando nodos.

Escriba un destructor para la clase ABB

Ejercicio 5.03.

```
class Abb {
    private:
      nodo* raiz;
      nodo* crearNodo(int);
      nodo* insertarAbb(int, nodo*);
    public:
      Abb();
      ~Abb();
      void insertar(int);
};

Abb::Abb(){
    raiz = NULL;
}
```

Ejercicio 5.03.

```
nodo* Abb::insertarAbb(int dato,  nodo* hoja){
    if (hoja == NULL)
      hoja = crearNodo(dato);
    else if (dato < hoja->dato)
      hoja->izquierdo = insertarAbb(dato, hoja->izquierdo);
    else if (dato > hoja->dato)
     hoja->derecho = insertarAbb(dato, hoja->derecho);
    return hoja;
}

void Abb::insertar(int dato){
    raiz = insertarAbb(dato, raiz);
}
```

Ejercicio 5.03.

```
Abb::~Abb(){
    removerSubarbol(raiz);
}

void Abb::removerSubarbol(nodo* nodo){
    if(nodo){
        if(nodo->izquierdo)
            removerSubarbol(nodo->izquierdo);
        if(nodo->derecho)
            removerSubarbol(nodo->derecho);
        delete nodo;
    }
}
```

Ejercicio 5.03.

```
int main()
{
    Abb a1;
    a1.insertar(8);
    a1.insertar(7);
    a1.insertar(2);
    a1.insertar(21);
    a1.insertar(9);
    a1.insertar(30);

    return 0;
}
```

Ejercicio 5.04.

Implementar el método buscar(x) que busque un nodo en un árbol binario por su valor.

Ejercicio 5.04.

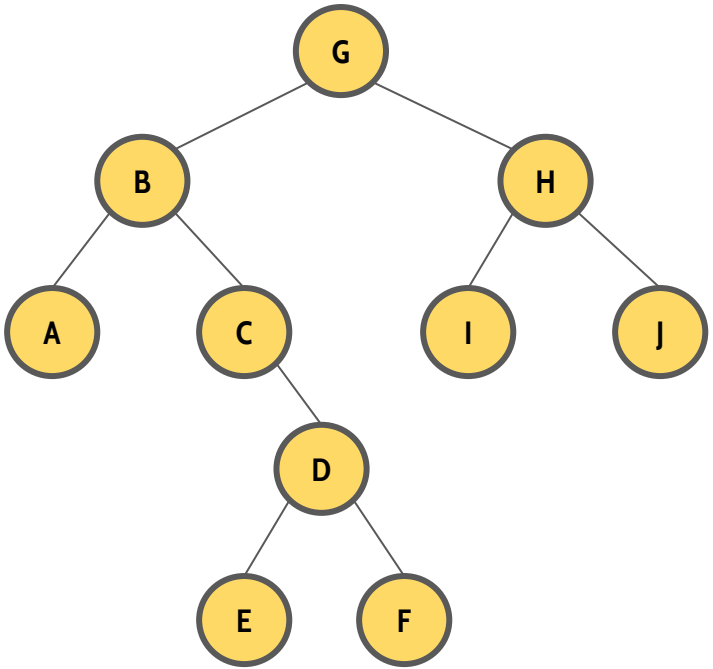
```
nodo* Abb::buscarAbb(int dato, nodo* hoja){
    if (hoja == NULL)
      return NULL;
    else if (dato < hoja->dato)
      return buscarAbb(dato, hoja->izquierdo);
    else if (dato > hoja->dato)
      return buscarAbb(dato, hoja->derecho);
    else
      return hoja;
}

nodo* Abb::buscar(int dato){
    return buscarAbb(dato, raiz);
}
```

recorrer un árbol binario

Es el proceso de visitar cada nodo en un árbol una sola vez y en un orden determinado

Se puede recorrer:

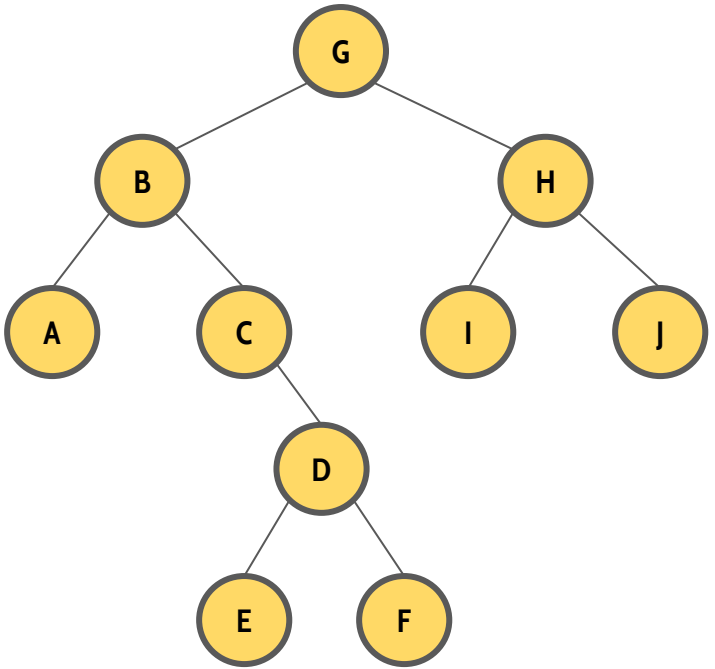
- En profundidad
- En amplitud



preorden

Secuencia de recorrido **preorden**:

1. Se visita la raíz
2. Se recorre el subárbol izquierdo
3. Se recorre el subárbol derecho

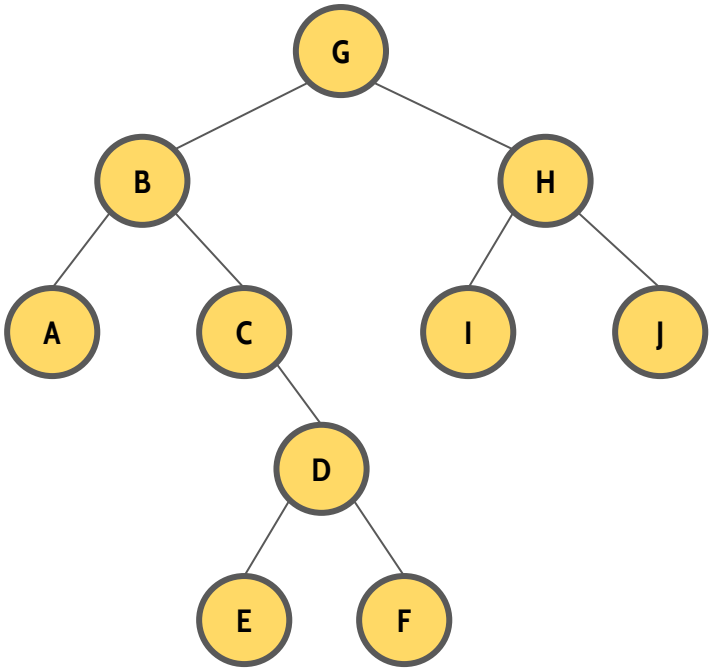


preorden

Secuencia de recorrido **preorden**:

1. Se visita la raíz
2. Se recorre el subárbol izquierdo
3. Se recorre el subárbol derecho

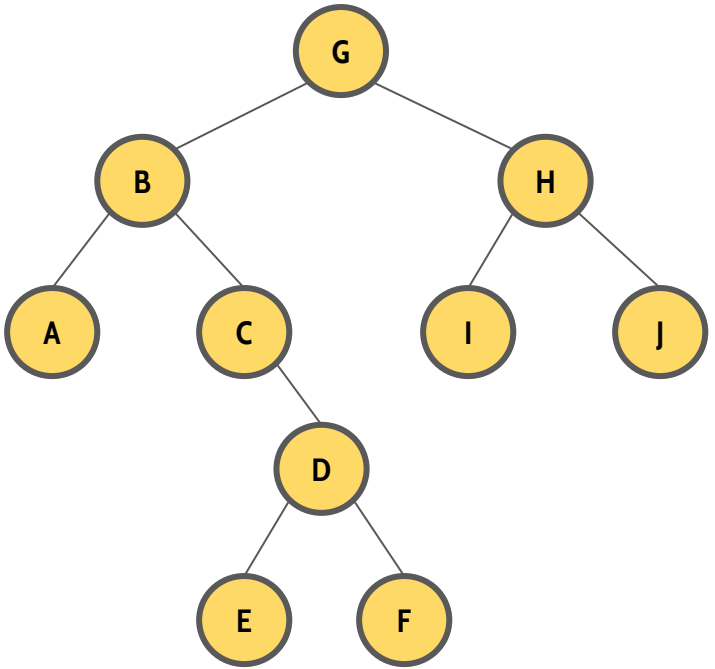
Recorrido: **G, B, A, C, D, E, F, H, I, J**



inorden

Secuencia de recorrido **inorden**:

1. Se recorre el subárbol izquierdo
2. Se visita la raíz
3. Se recorre el subárbol derecho

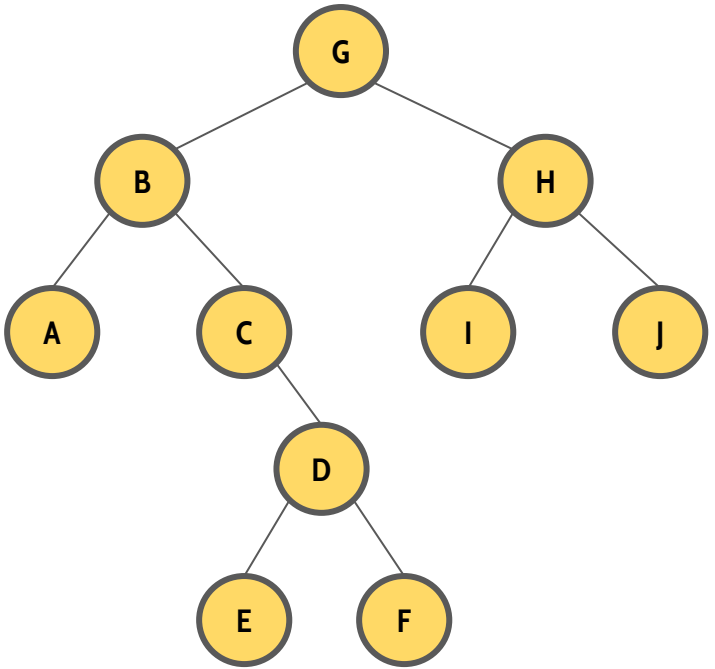


inorden

Secuencia de recorrido **inorden**:

1. Se recorre el subárbol izquierdo
2. Se visita la raíz
3. Se recorre el subárbol derecho

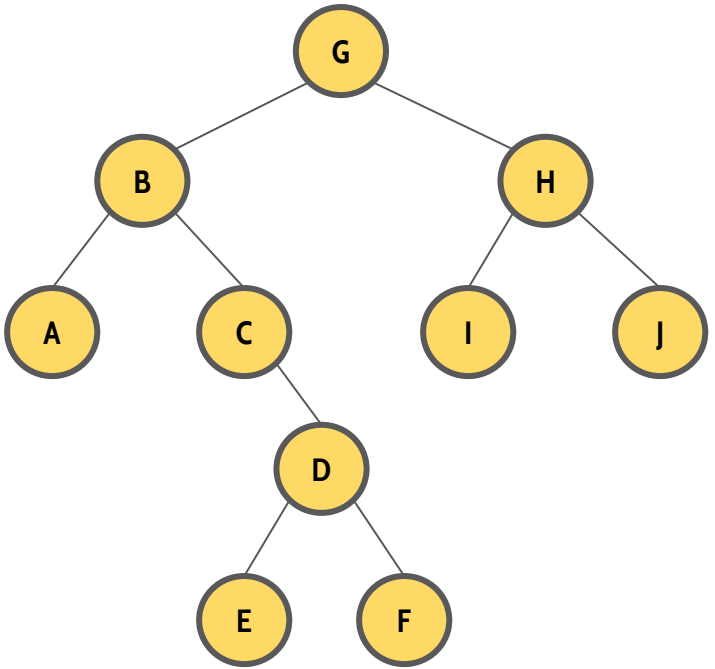
Recorrido: **A, B, C, E, D, F, G, I, H, J**



postorden

Secuencia de recorrido **postorden**:

1. Se recorre el subárbol izquierdo
2. Se recorre el subárbol derecho
3. Se visita la raíz

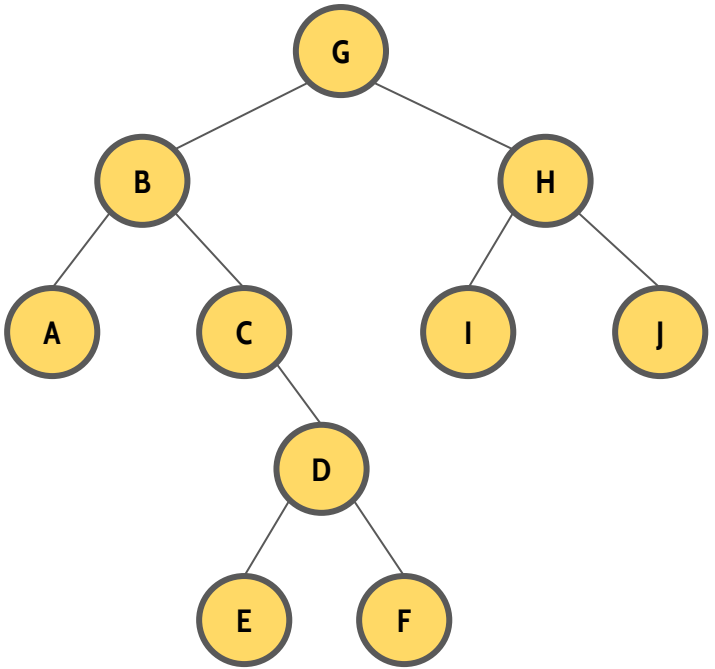


postorden

Secuencia de recorrido **postorden**:

1. Se recorre el subárbol izquierdo
2. Se recorre el subárbol derecho
3. Se visita la raíz

Recorrido: **A, E, F, D, C, B, I, J, H, G**



Ejercicio 5.05.

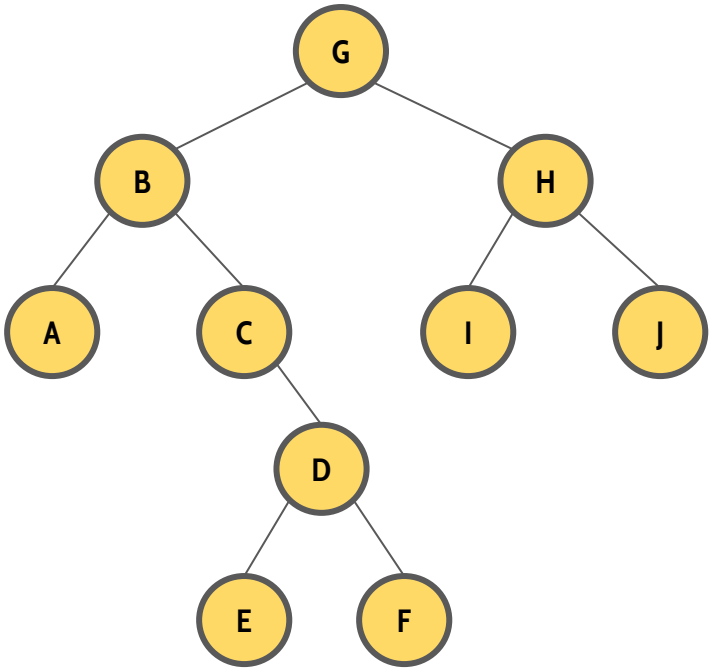
Implementar los métodos para imprimir el árbol recorriendolo en preorden, inorden y postorden

Ejercicio 5.05.

```
void Abb::mostrarInOrdenAbb(nodo* hoja){
    if (hoja){
      mostrarInOrdenAbb(hoja->izquierdo);
      cout << hoja->dato << " ";
      mostrarInOrdenAbb(hoja->derecho);
    }
}

void Abb::mostrarInOrden(){
    mostrarInOrdenAbb(raiz);
    cout << endl;
}
```

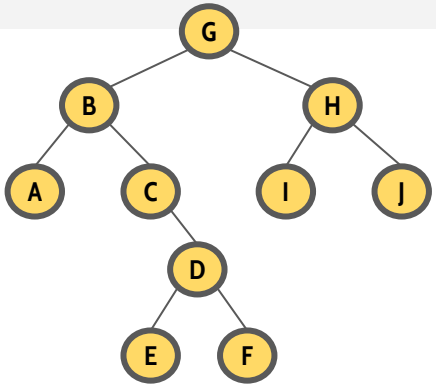
recorrer en amplitud

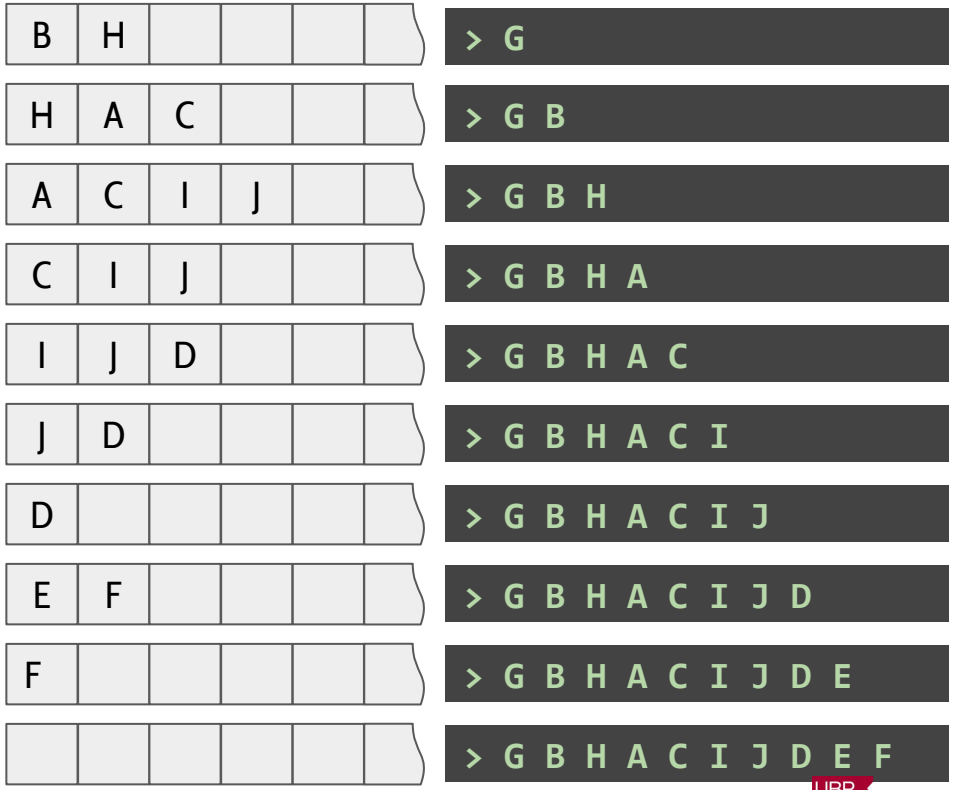
Recorrido: G, B, H, A, C, I, J, D, E, F



recorrer en amplitud: pseudocodigo

```
mostrarAmplitud(raiz)
{
      crear cola c
      nodoActual ← raiz
      mientras nodoActual no sea nulo
      {
            imprimir nodoActual.dato
            c.ingresar(nodoActual.izquierdo)
            c.ingresar(nodoActual.derecho)
            nodoActual ← c.extraer()
      }
}
```





Ejercicio 5.06.

Implementar los métodos para recorrer el árbol en amplitud

Ejercicio 5.06.

```
void abb::recorrerAmplitud(){
    queue<nodoAbb*> c;
    nodoAbb* nodoActual = raiz;
    while (nodoActual) {
       cout << nodoActual->dato << " ";
       if (nodoActual->izq)
            c.push(nodoActual->izq);
        if (nodoActual->der)
            c.push(nodoActual->der);
       nodoActual = c.front();
       c.pop();
    }
    cout << endl;
}
```

Ejercicio 5.07.

El recorrido en postorden de un ABB que contiene caracteres es:

DMLCTAISRUNOKB

Y en inorden es:

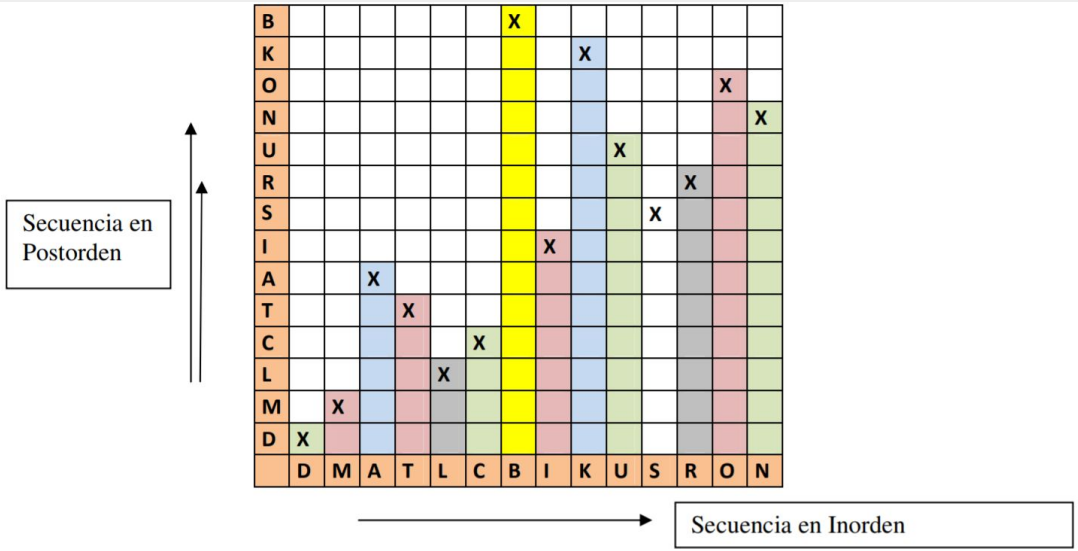
DMATLCBIKUSRON

Dibujar el árbol binario

Escribir una función que reconstruya el árbol original a partir de sus recorridos postorden e inorden

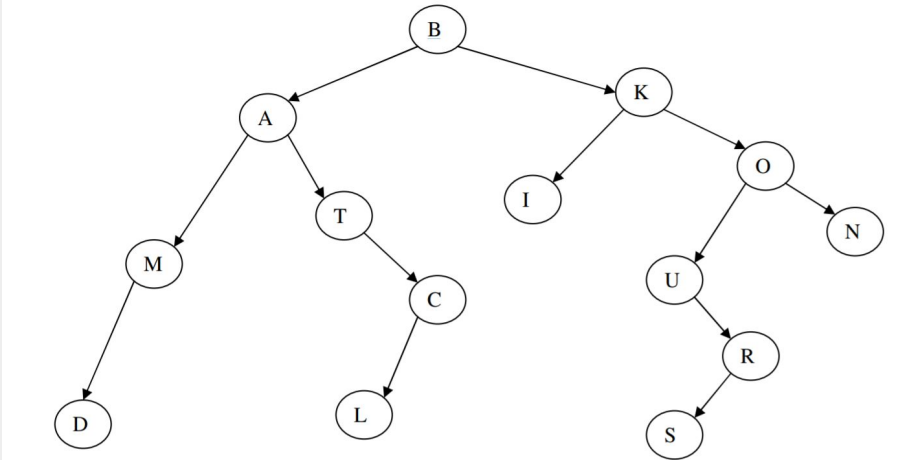
Ejercicio 5.07.

Para poder deducir el árbol binario que tiene esos recorridos, se debe recordar que dado un inorden y un pre o postorden, existe un único árbol binario que cumple ambos recorridos. Por lo tanto dibujaremos una tabla donde en la vertical tendremos el postorden, de abajo hacia arriba, y en la horizontal tendremos el inorden:



Ejercicio 5.07.

Nos damos cuenta de que existe solo una casilla por cada letra donde fila y columna coinciden. Pero qué hacemos con esto? Construimos el árbol. La marca de más arriba será la raíz del árbol y con eso tenemos el primer nodo. Los hijos de ese nodo serán, para las sub-tablas izquierda y derecha, el nodo que esté más alto, y los hijos de esos nodos serán a su vez los nodos más altos de las sub-tablas acotadas por las raíces ya obtenidas. En el diagrama pintamos de un color los nodos de un mismo nivel. El orden de los colores es: amarillo, azul, rojo, verde, plomo, blanco.
Dibujando el árbol binario construido:



Ejercicio 5.08.

Implementar los métodos

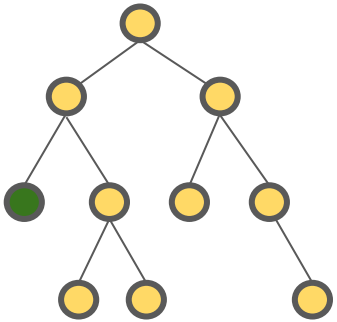
```
nodoAbb *obtenerMenor(nodoAbb *raiz)

nodoAbb *obtenerMayor(nodoAbb *raiz)
```

que obtengan el menor y el mayor elemento del árbol respectivamente

buscar el menor elemento

Equivale a encontrar el elemento más a la izquierda

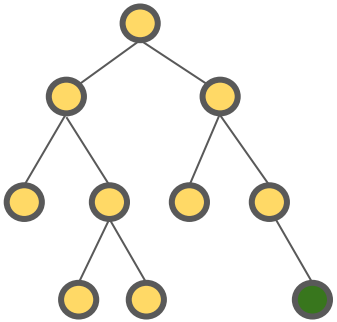
```
nodoAbb *obtenerMenor(nodoAbb *hoja)
{
     if (hoja == NULL)
     {
          return NULL;
     }
     else if (hoja->izquierdo == NULL)
     {
          return (hoja);
     }
     else
     {
          obtenerMenor(hoja>izquierdo);
     }
}
```



buscar el mayor elemento

Equivale a encontrar el elemento más a la derecha

```
nodoAbb *obtenerMayor(nodoAbb *nodo)
{
     if (nodo == NULL)
     {
          return NULL;
     }
     else if (nodo->derecho == NULL)
     {
          return (nodo);
     }
     else
     {
          obtenerMayor(nodo->derecho);
     }
}
```

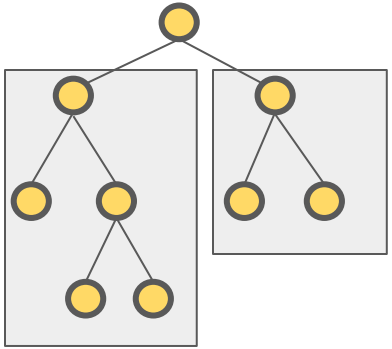


calcular la altura

La altura de un nodo es igual a la mayor de las alturas de sus subárboles + 1

La altura de un nodo vacío es -1

```
int obtenerAltura(nodo)
{
    si nodo es vacío
    {
        retornar -1
    }
    sino
    {
        alturaIzquierdo ← obtenerAltura(nodo.izquierdo);
        alturaDerecho ← obtenerAltura(nodo.derecho);
        retornar max(alturaIzquierdo, alturaDerecho) + 1;
    }
}
```



Ejercicio 5.08.

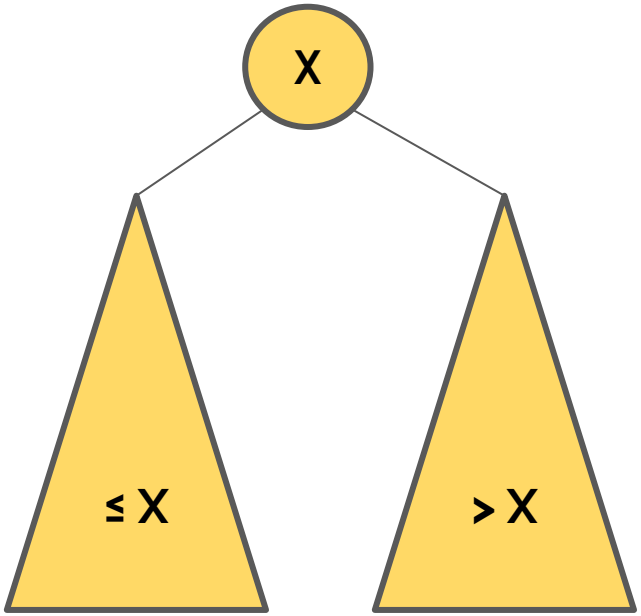
Implementar una función que permita obtener la altura de un nodo.

Ejercicio 5.08.

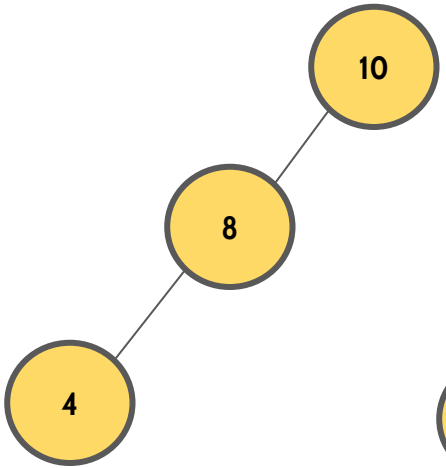
```
int Abb::altura(nodo* hoja){
    if (hoja){
        return max(altura(hoja->izquierdo), altura(hoja->derecho)) + 1;
    }
    else {
        return -1;
    }
}

int Abb::altura(){
    return altura(raiz);
}
```

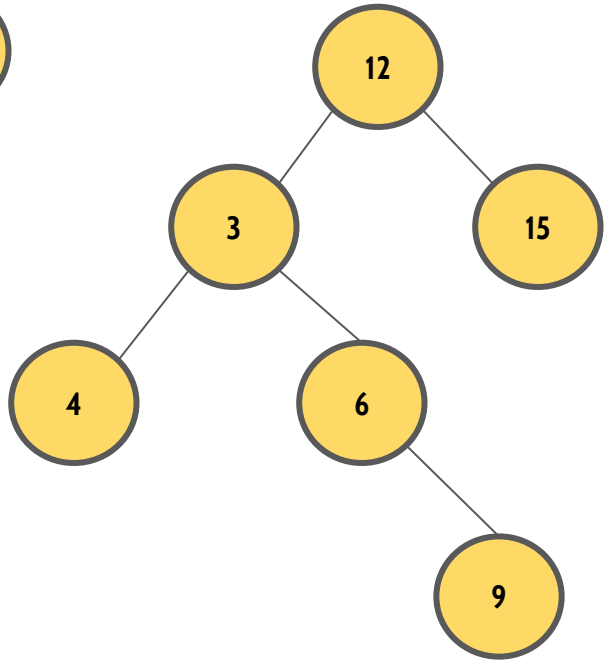
determinar si es un árbol binario de búsqueda

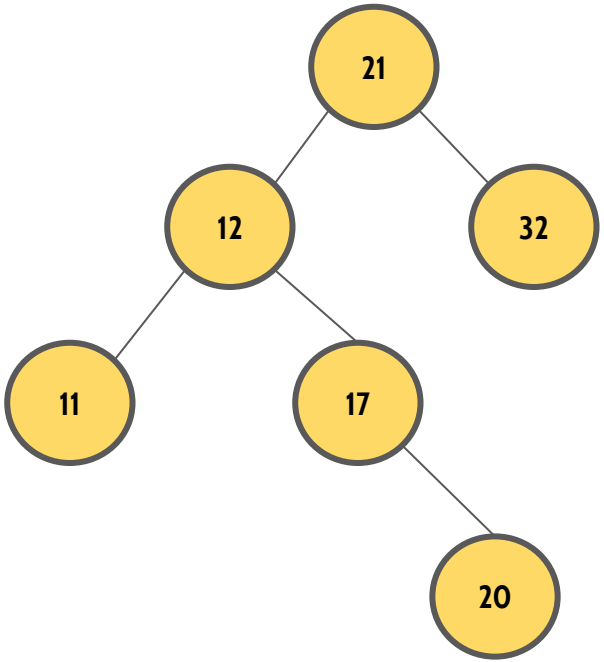
Para cada nodo



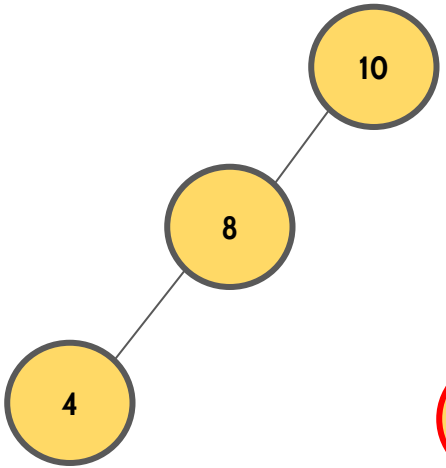
determinar si es un árbol binario de búsqueda

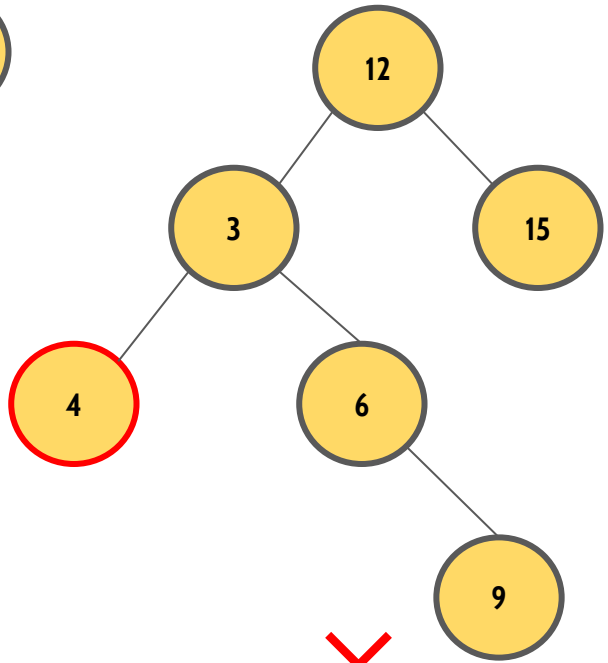


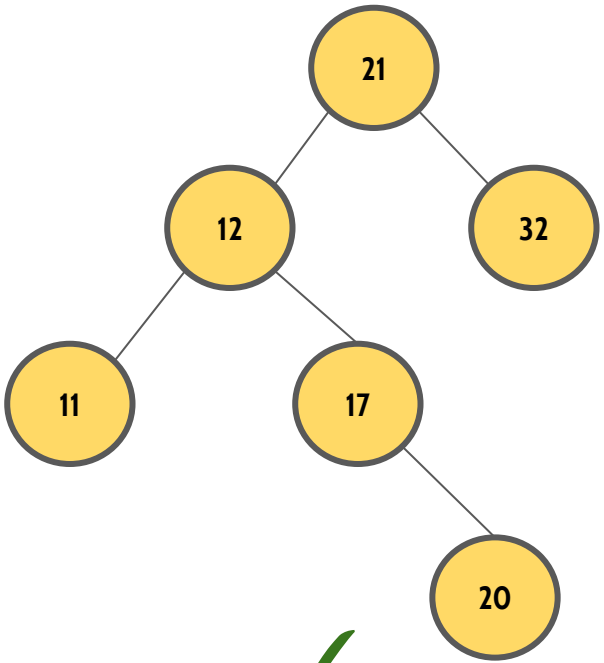




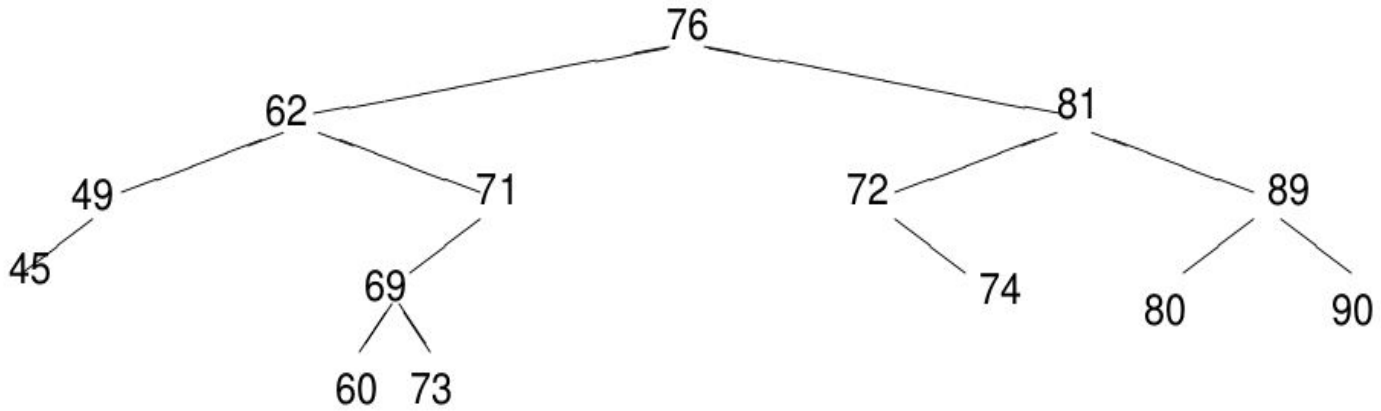
determinar si es un árbol binario de búsqueda





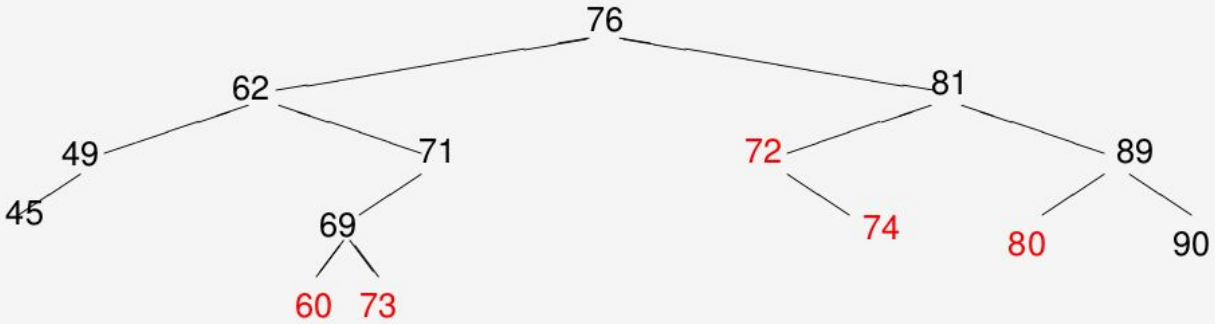


determinar si es un árbol binario de búsqueda



¿Es un árbol binario de búsqueda?

determinar si es un árbol binario de búsqueda



No es un árbol binario de búsqueda.

- 60 debe cambiar por uno entre 63 y 68
- 72 debe cambiar por uno entre 77 y 80
- 73 debe cambiar por 70
- 74 debe cambiar por uno entre 77 y 80.
- 80 debe cambiar por uno entre 82 y 88.

determinar si es un árbol binario de búsqueda

```
bool esAbb(nodo)
{
      si nodo es NULL retornar verdadero

      si (esSubarbolMenor(nodo.izquierdo, nodo.dato)
            && esSubarbolMayor(nodo.derecho, nodo.dato)
            && esAbb(nodo.izquierdo)
            && esAbb(nodo.derecho))
            retornar verdadero
      sino
            retornar falso
}
```

Ejercicio 5.09.

Implementar el método que permita determinar si un árbol es abb de acuerdo al procedimiento recursivo analizado.

Algoritmo 1

determinar si es abb

```
private:
     bool esSubarbolMenor(nodo*, int);
     bool esSubarbolMayor(nodo*, int);
     bool esAbb(nodo*);
public:
      bool esAbb();
```

Algoritmo 1

determinar si es abb

```
bool Abb::esSubarbolMenor(nodo* hoja, int dato){
    if (!hoja) return true;
    if (hoja->dato <= dato
      && esSubarbolMenor(hoja->izquierdo, dato)
      && esSubarbolMenor(hoja->derecho, dato))
      return true;
    else
      return false;
}
```

Algoritmo 1

determinar si es abb

```
bool Abb::esSubarbolMayor(nodo* hoja, int dato){
    if (!hoja) return true;
    else if (hoja->dato >= dato
      && esSubarbolMayor(hoja->izquierdo, dato)
      && esSubarbolMayor(hoja->derecho, dato))
      return true;
    else
      return false;
}
```

Algoritmo 1

determinar si es abb

```
bool Abb::esAbb(nodo* hoja){
    if (!hoja) return true;
    else if (esAbb(hoja->izquierdo) && esAbb(hoja->derecho)
      && esSubarbolMenor(hoja->izquierdo, hoja->dato)
      && esSubarbolMayor(hoja->derecho, hoja->dato))
      return true;
    else
      return false;
}

bool Abb::esAbb(){
    return esAbb(raiz);
}
```

determinar si es abb

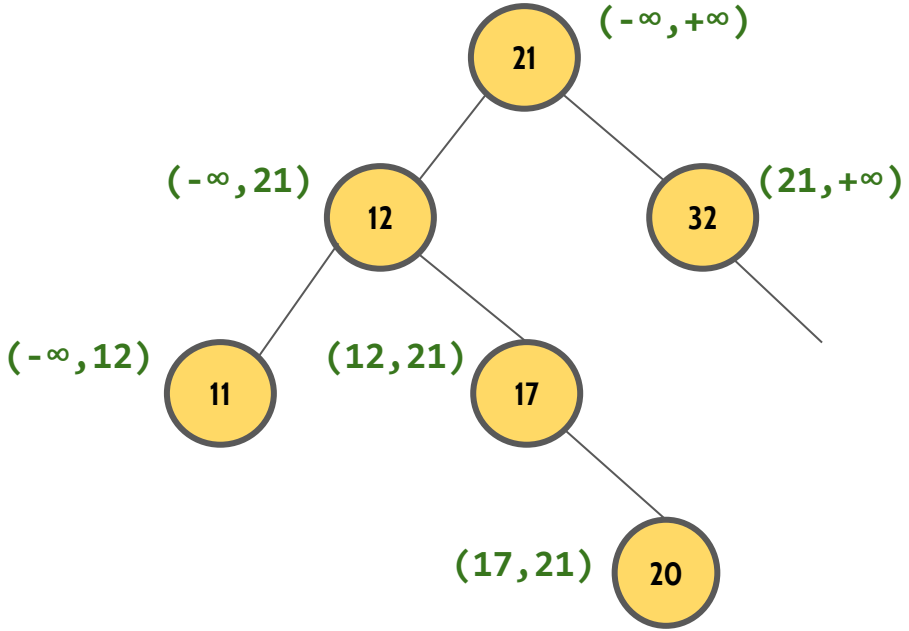
El algoritmo que acabamos de implementar es muy ineficiente.

Se recorren los mismos subárboles en varias oportunidades, para comprobar si los valores son menores o mayores que uno dado, dependiendo el caso.

Existe una variante que nos permite recorrer el árbol una sola vez.

determinar si es abb: mejora

Consiste en determinar los rangos en los que los valores de un subárbol deben estar
Comenzando con (-∞, +∞) y restringiendo los rangos en la medida en que descendemos en el árbol



Ejercicio 5.10.

Implementar el método que permita determinar si un árbol es abb de acuerdo al procedimiento mejorado que acabamos de exponer.

Algoritmo 2

determinar si es abb

```
bool Abb::esAbbMejorado(nodo* hoja, int min, int max){
    if (!hoja) return true;
    else if((hoja->dato > min) && (hoja->dato < max)
      && esAbbMejorado(hoja->izquierdo, min, hoja->dato)
      && esAbbMejorado(hoja->derecho, raiz->dato, max))
      return true;
    else
      return false;

}

bool Abb::esAbbMejorado(){
    return esAbbMejorado(raiz, INT_MIN, INT_MAX);
}
```

determinar si es abb

Una tercera posibilidad es utilizar el recorrido inorder: cuando el ABB se recore inorder, los valores deben estar ordenados de menor a mayor, es decir: debe ser $\mathbf{n_{i+1} > n_i}$ para dos nodos cualesquiera

Ejercicio 5.11.

Implementar el método que permita determinar si un árbol es abb usando el recorrido inorden.

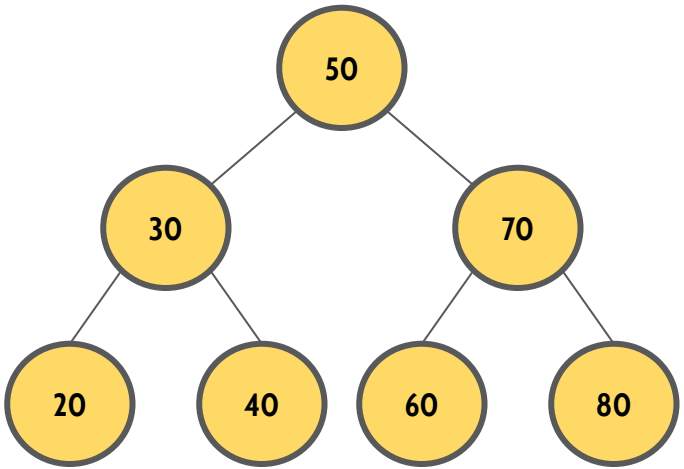
Ejercicio 5.11.

```
bool Abb::esAbbInOrden(nodo* hoja){
   static nodo* prev = NULL;
   if (hoja)
   {
     if (!esAbbInOrden(hoja->izquierdo)) return false;
     if (prev != NULL && hoja->dato <= prev->dato) return false;
     prev = hoja;
     return esAbbInOrden(hoja->derecho);
     }
     return true;
}

bool Abb::esAbbInOrden(){
    return esAbbInOrden(raiz);
}
```

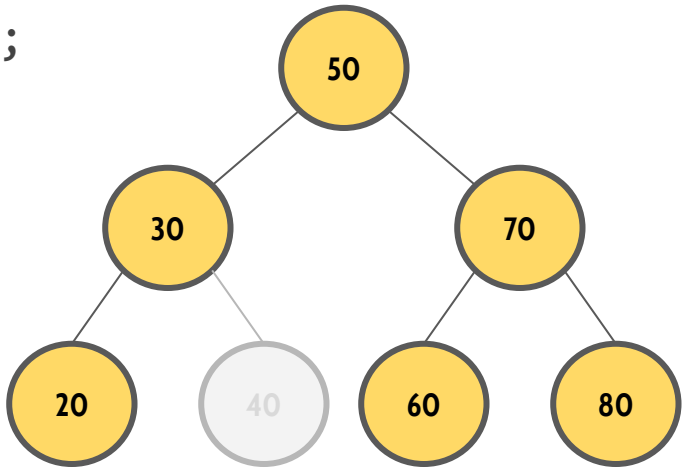
eliminar nodos

Surgen tres escenarios distintos:

1. El nodo es una hoja (no tiene descendientes)



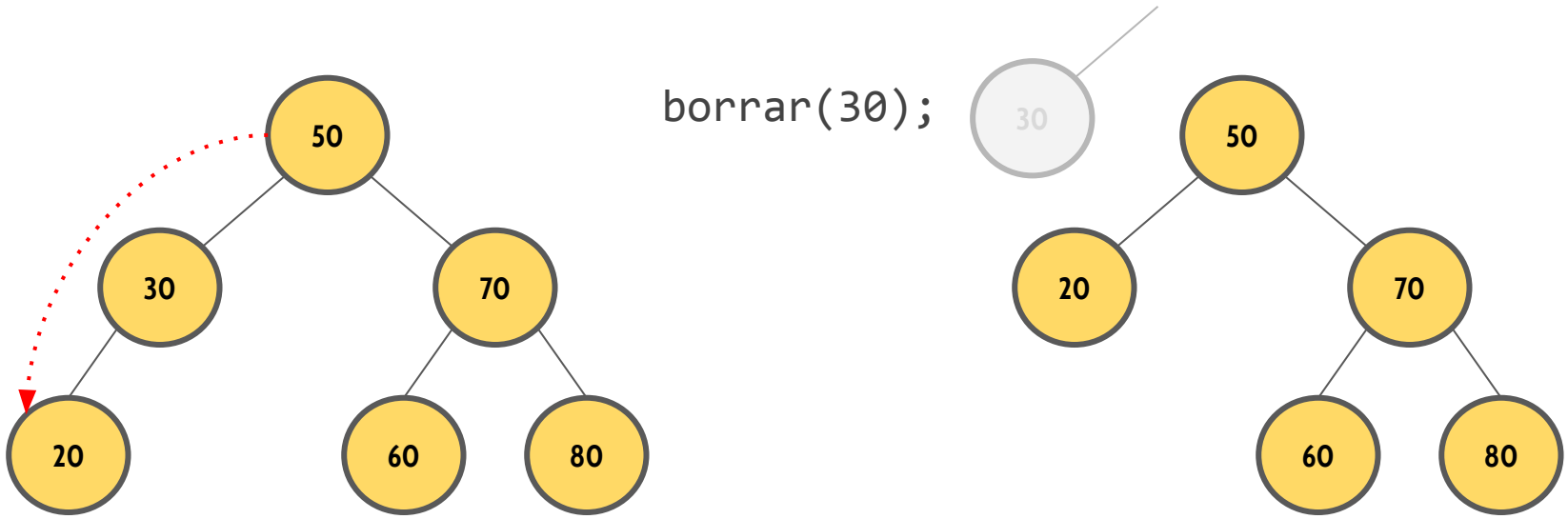

```
borrar(40);
```



Simplemente eliminamos el nodo

eliminar nodos

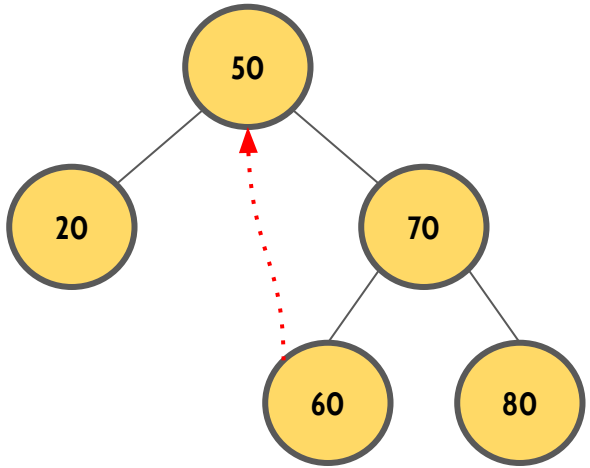
2. El nodo a ser borrado tiene solo un hijo



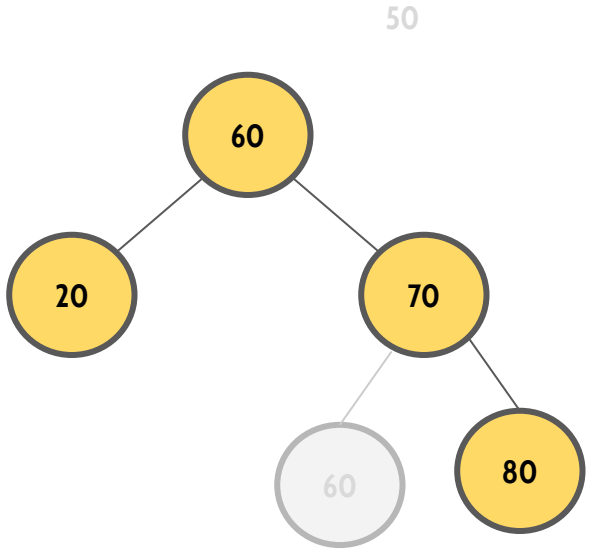
Conectamos el padre del nodo a eliminar con el hijo del nodo a eliminar y removemos el nodo en cuestión

eliminar nodos

3. El nodo a ser borrado tiene dos hijos




```
borrar(50);
```



Buscamos el menor nodo del subárbol derecho

Reemplazamos el valor del nodo a borrar por ese valor

Eliminamos el duplicado del subárbol derecho usando los casos anteriores

Ejercicio 5.12.

Implementar el método eliminar que permite eliminar un nodo determinado a partir del valor provisto

Ejercicio 5.12.

```
nodo* Abb::eliminar(nodo* hoja, int x){
    if (!hoja) return hoja;
    else if (x < hoja->dato) hoja->izquierdo = eliminar(hoja->izquierdo, x);
    else if (x > hoja->dato) hoja->derecho = eliminar(hoja->derecho, x);
    else{
        // Caso 1: es una hoja
        if (hoja->izquierdo == NULL && hoja->derecho == NULL){
            delete hoja;
            hoja = NULL;
        }
        // Caso 2: un solo hijo
        else if (hoja->izquierdo == NULL){
            nodo* tmp = hoja;
            hoja = hoja->derecho;
            delete tmp;
        }
        else if (hoja->derecho == NULL){
            nodo* tmp = hoja;
            hoja = hoja->izquierdo;
            delete tmp;
        }
```

Ejercicio 5.12.

```
        // Caso 3: tiene dos hijos
        else{
            nodo* tmp = buscarMenor(hoja->derecho);
            hoja->dato = tmp->dato;
            eliminar(hoja->derecho, tmp->dato);
        }
    }
    return hoja;
}



void Abb::eliminar(int dato){
    raiz = eliminar(raiz, dato);
}
```

Ejercicio 5.13.

Implementar el método eliminar que permite mediante un algoritmo iterativo permite eliminar un nodo determinado a partir del valor provisto

Ejercicio 5.14.

Escriba una función que encuentre el segundo mayor valor de un abb

Ejercicio 5.15.

Dados dos árboles ABB: listar todos los elementos comunes a ambos árboles.

(nota: hay una solución trivial a este problema que es recorrer uno de los árboles e ir buscando cada elemento en el otro árbol. Esta solución es O(n1 x h2) donde n1 es la cantidad de elementos del primer árbol y h2 es la altura del segundo árbol. Debe encontrar una solución más eficiente)

Ejercicio 5.16.

Imprimir un árbol binario en la consola.

Ejercicio 5.16.

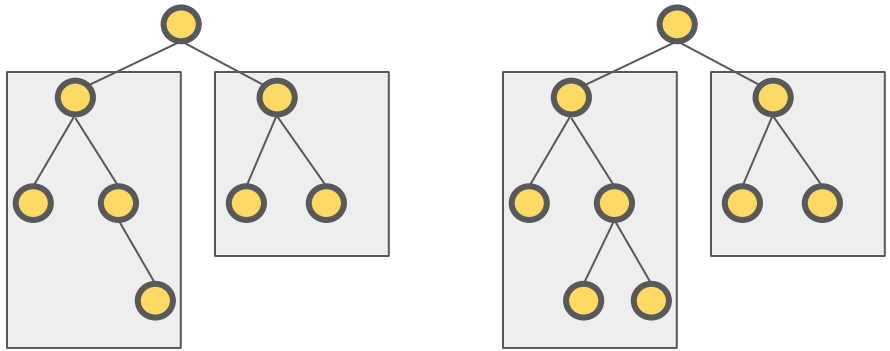
```
void Abb::mostrar(nodo* nodo, int espacio){
    if(nodo == NULL)
        return;
    espacio += 10;
    mostrar(nodo->derecho, espacio);
    cout << endl;
    for (int i = 10; i < espacio; i++) {
        cout << " ";
    }
    cout << "[" << nodo->dato << "]\n";
    mostrar(nodo->izquierdo, espacio);
}
void Abb::mostrar(){
    mostrar(raiz, 0);
}
```

Árbol AVL

árbol balanceado

Un árbol está balanceado cuando:

- El **número de nodos** del subárbol izquierdo difiere como máximo en uno del **número de nodos** del subárbol derecho
- La **altura** del subárbol izquierdo difiere como máximo en uno de la **altura** del subárbol derecho (menos restrictiva)

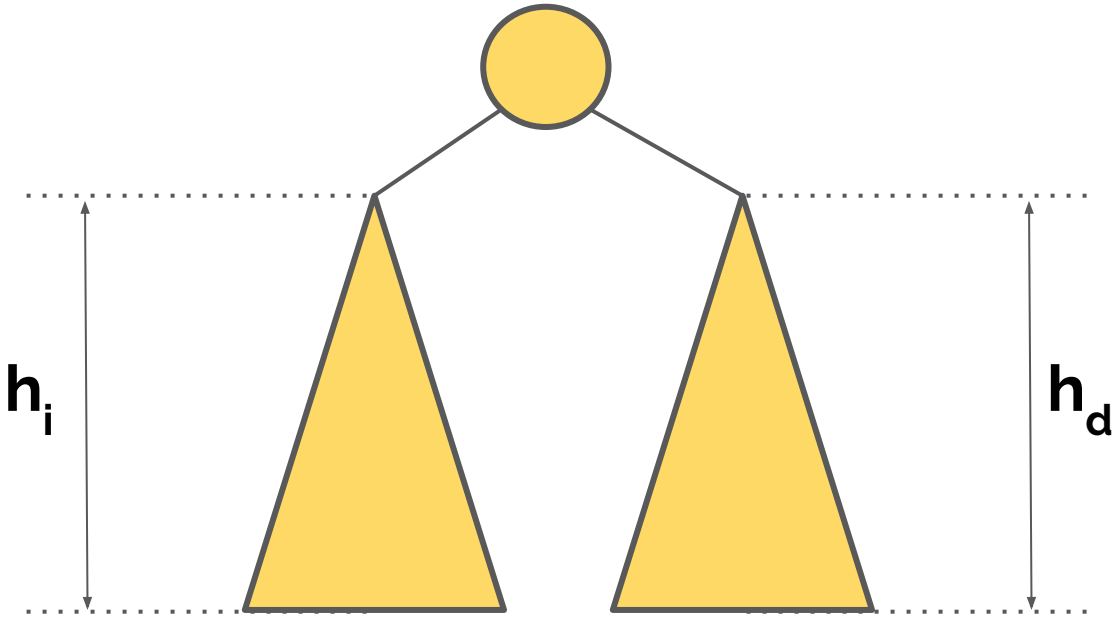


árbol AVL

Ideado por los matemáticos rusos: Georgy **A**delson-**V**elsky y Evgenii **L**andis

- Propusieron un árbol **autobalanceado**, de modo que para todos los nodos, la altura de la rama izquierda no difiere en más de una unidad de la altura de la rama derecha
- Al insertar o extraer nodos, si se pierde el equilibrio, se ejecutan operaciones que restauran el balance del árbol.
- Esto nos permite mantener, para las operaciones de búsqueda, un costo de O(log n) en el caso promedio y en el peor caso.

árbol AVL: factor de equilibrio

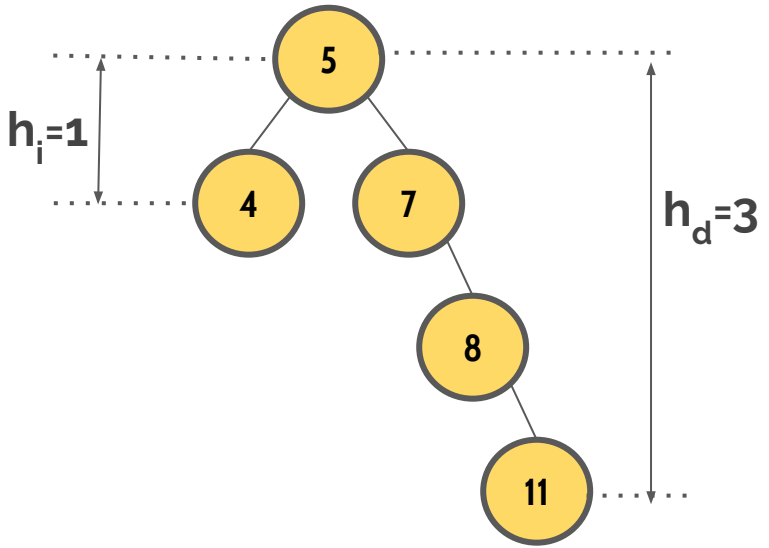


Tiene que ser $|h_i - h_d| \leq 1$ para cada nodo

árbol AVL: factor de equilibrio

Árbol 1

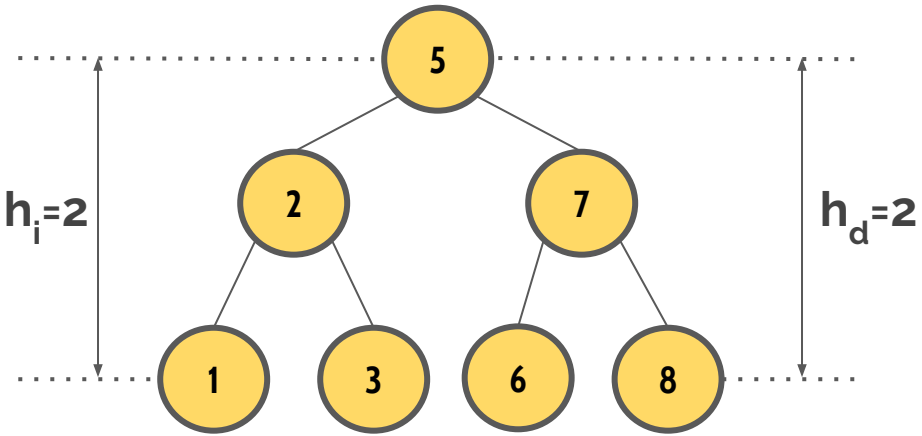
5, 4, 7, 8, 11



FB = h_i-h_d = 1 - 3 = -2

Árbol 2

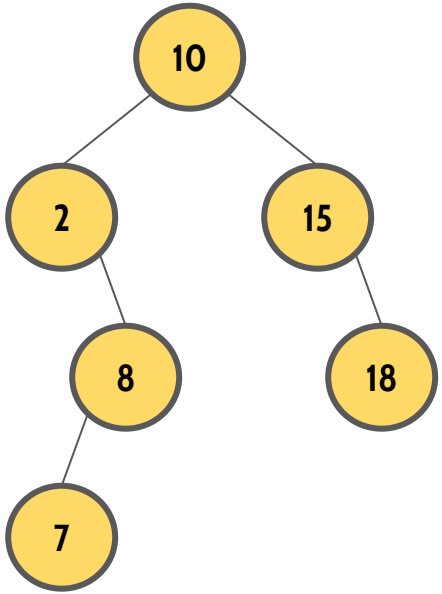
5, 2, 7, 1, 3, 8, 6



FB = h_i-h_d = 2 - 2 = 0

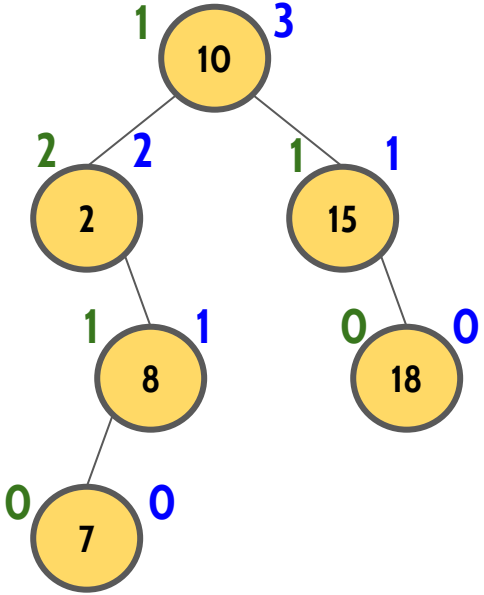
factor de equilibrio

¿Es un árbol AVL?



factor de equilibrio

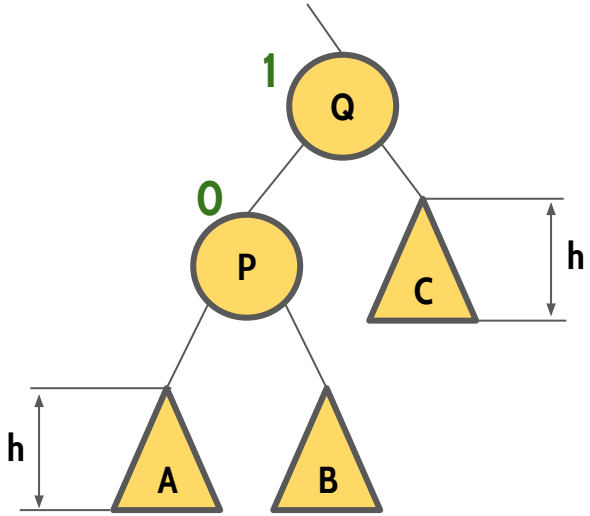
¿Es un árbol AVL?



- Altura
- Factor de Equilibrio

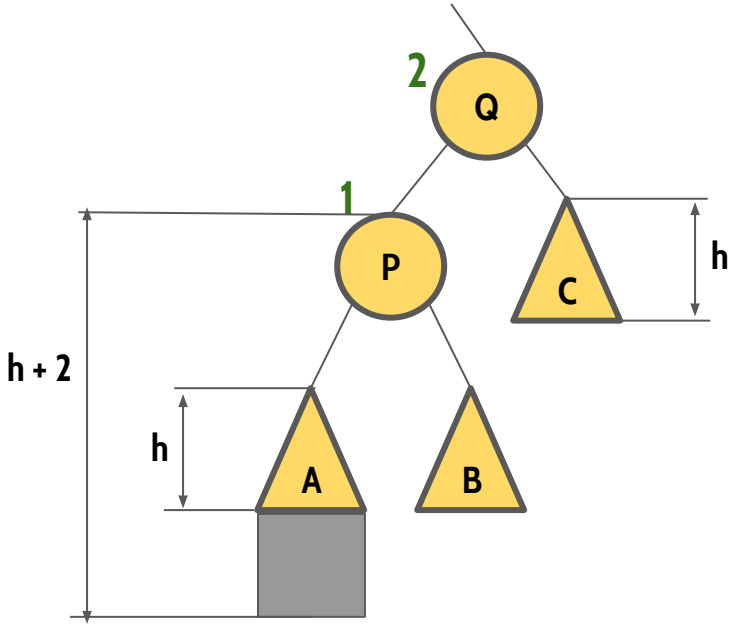
rotación

Rotación simple: izquierda



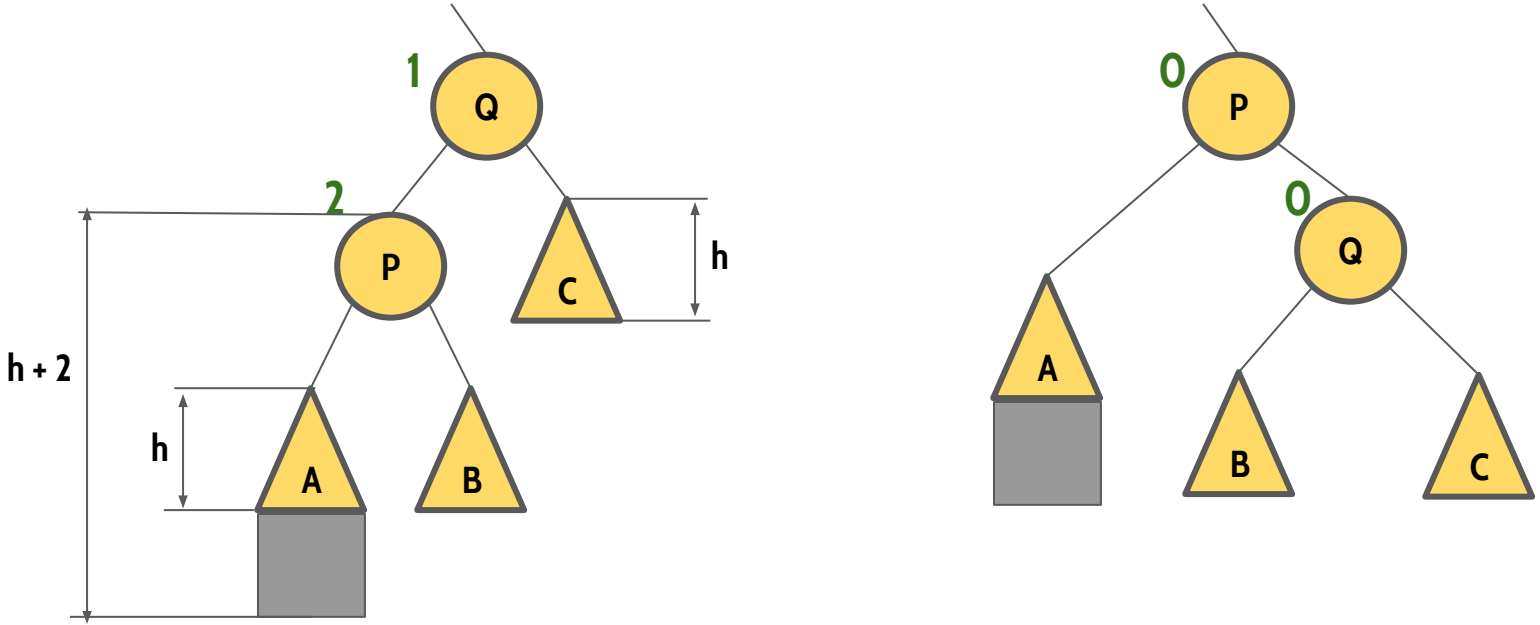
rotación

Rotación simple: izquierda



rotación

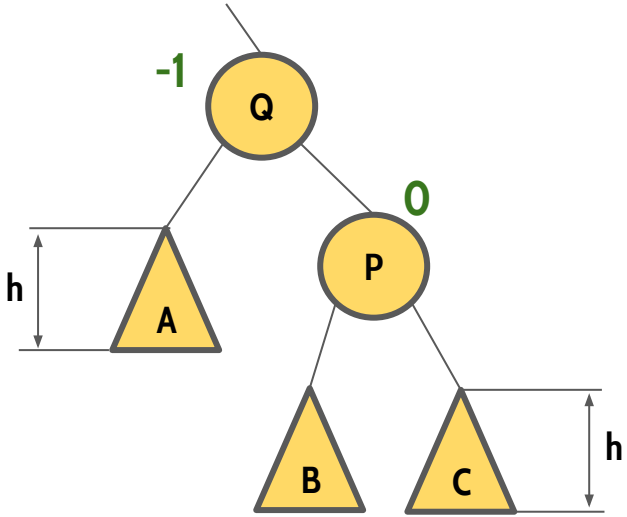
Rotación simple: izquierda



Problema izquierda - izquierda se resuelve con rotación simple izquierda

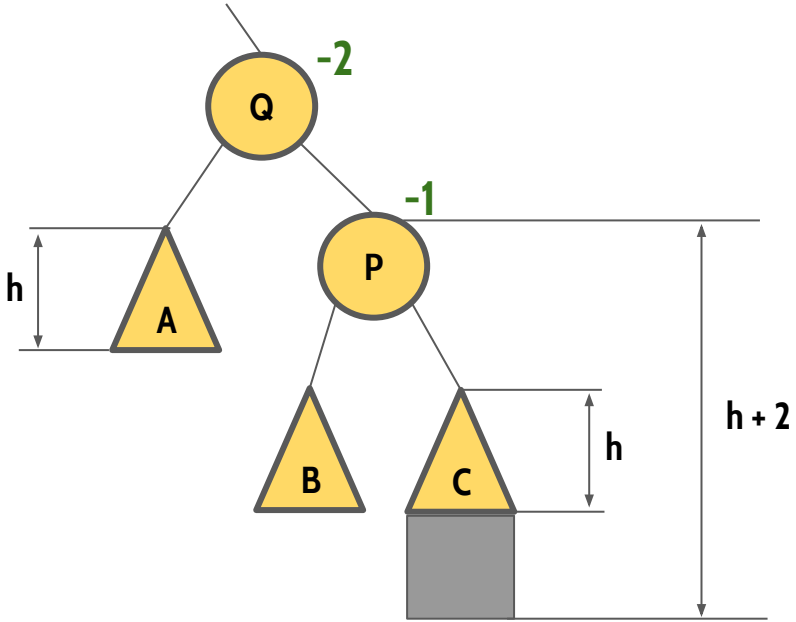
rotación

Rotación simple: derecha



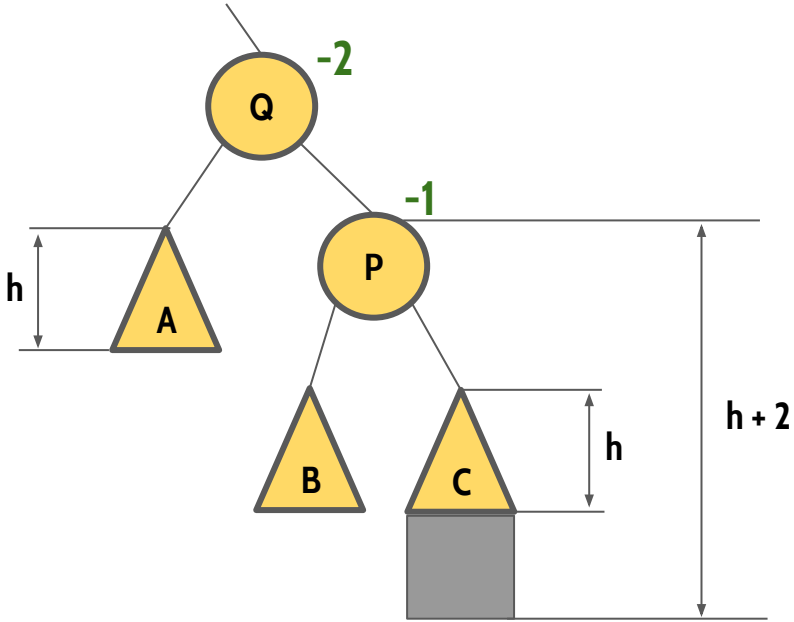
rotación

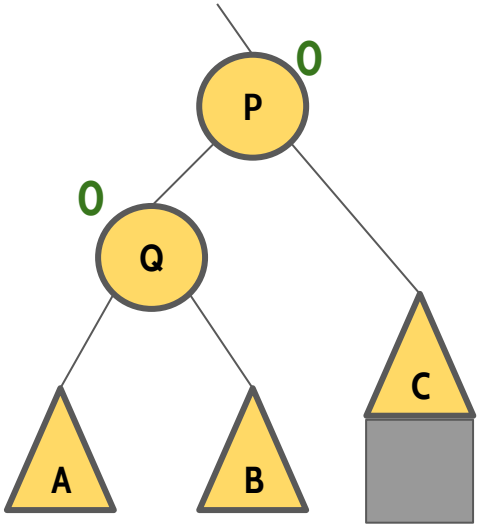
Rotación simple: derecha



rotación

Rotación simple: derecha

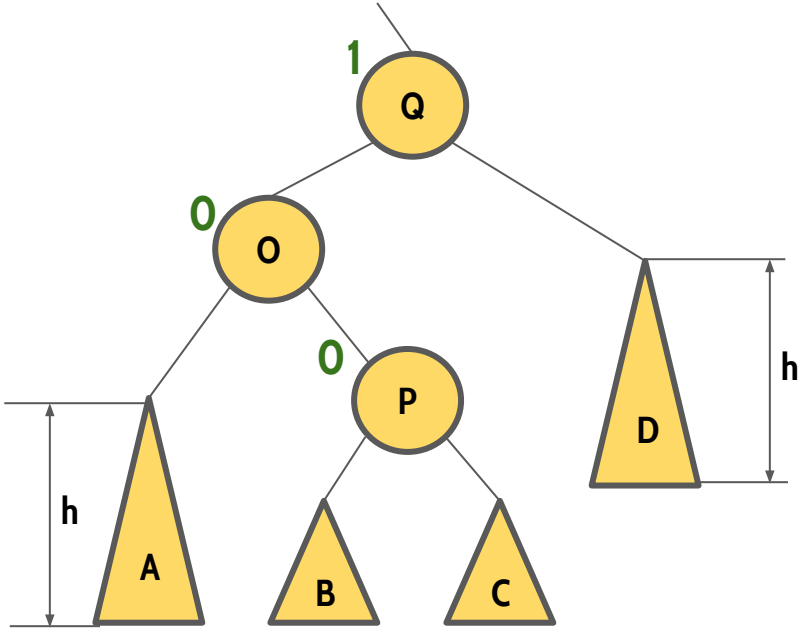




Problema derecha - derecha se resuelve con rotación simple derecha

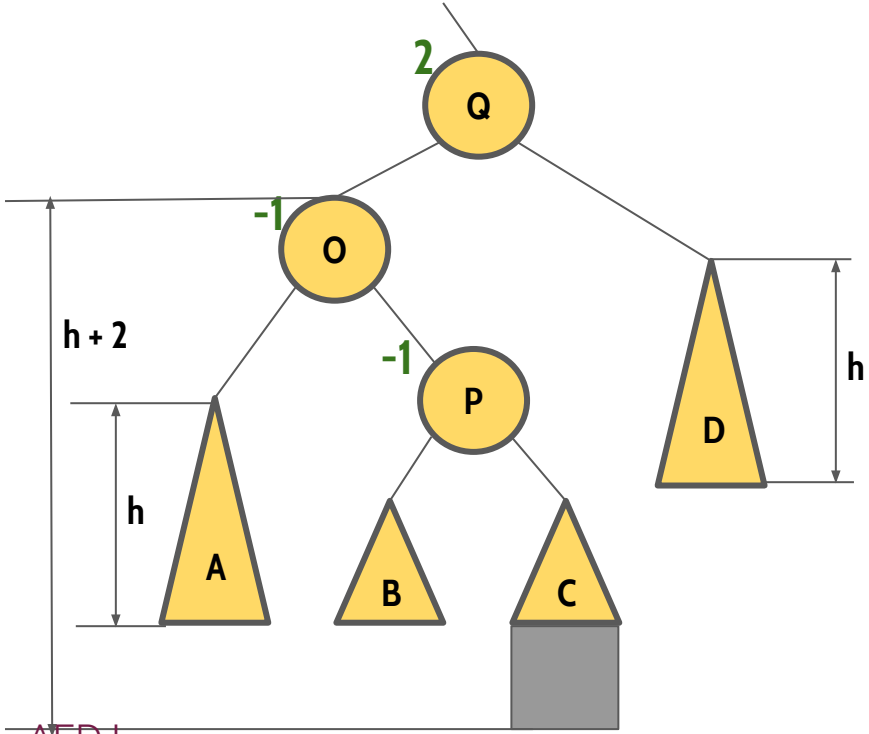
rotación

Rotación doble: izquierda derecha



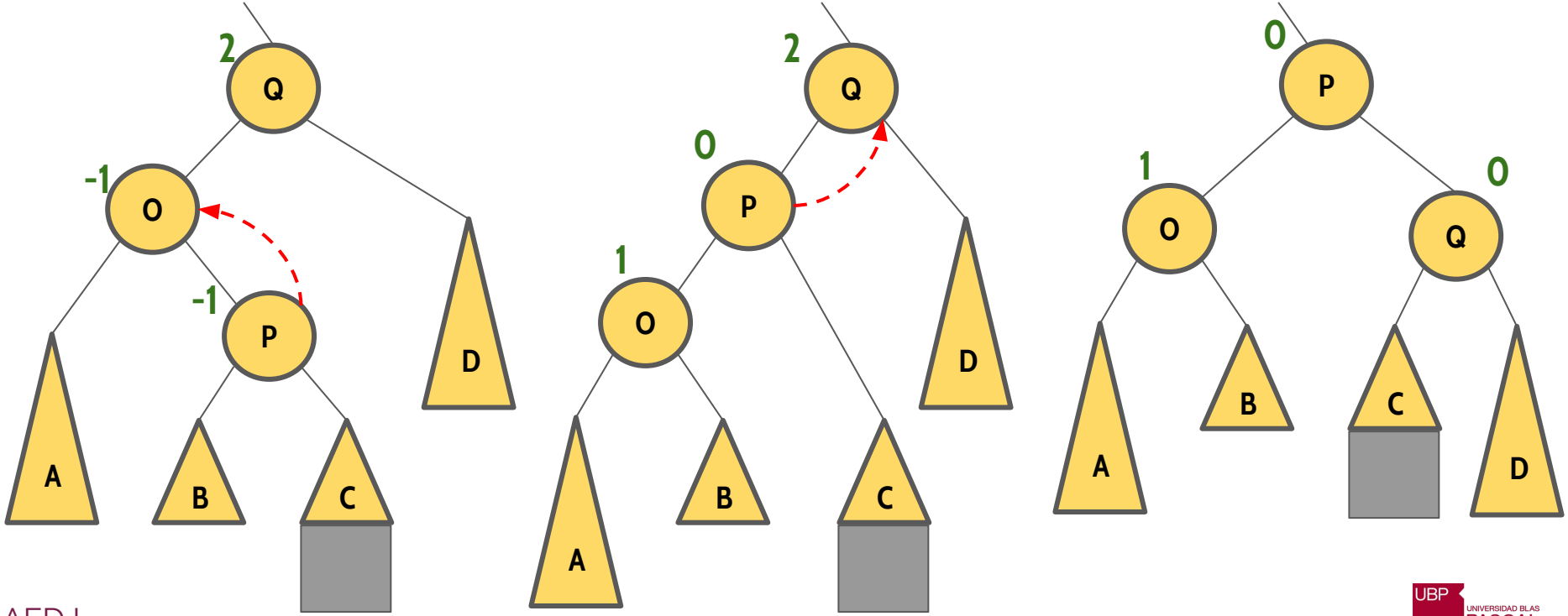
rotación

Rotación doble: izquierda derecha



rotación

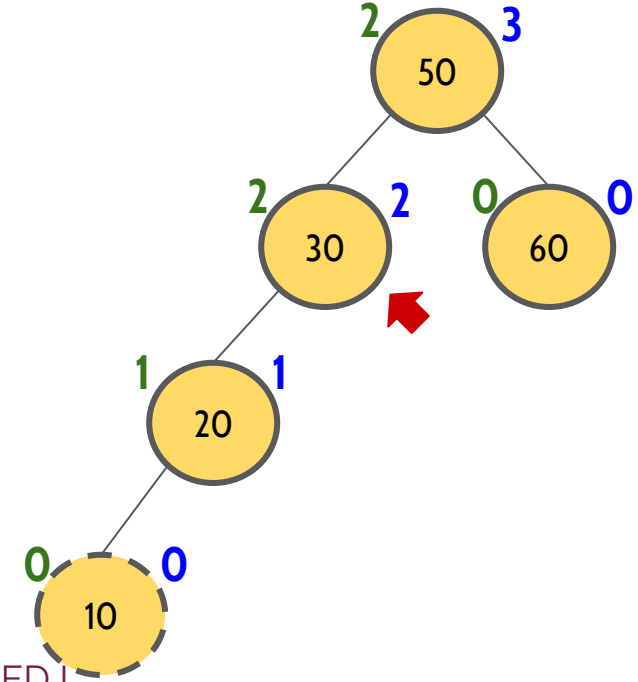
Rotación doble: izquierda derecha



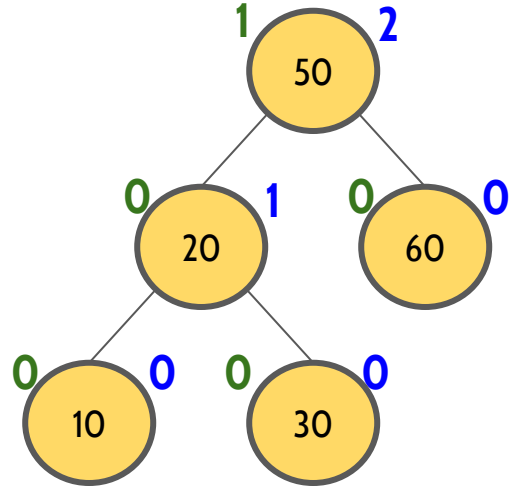
Problema izquierda - derecha se resuelve con rotación doble izquierda - derecha

inserción en un árbol AVL

```
insertar(10);
```



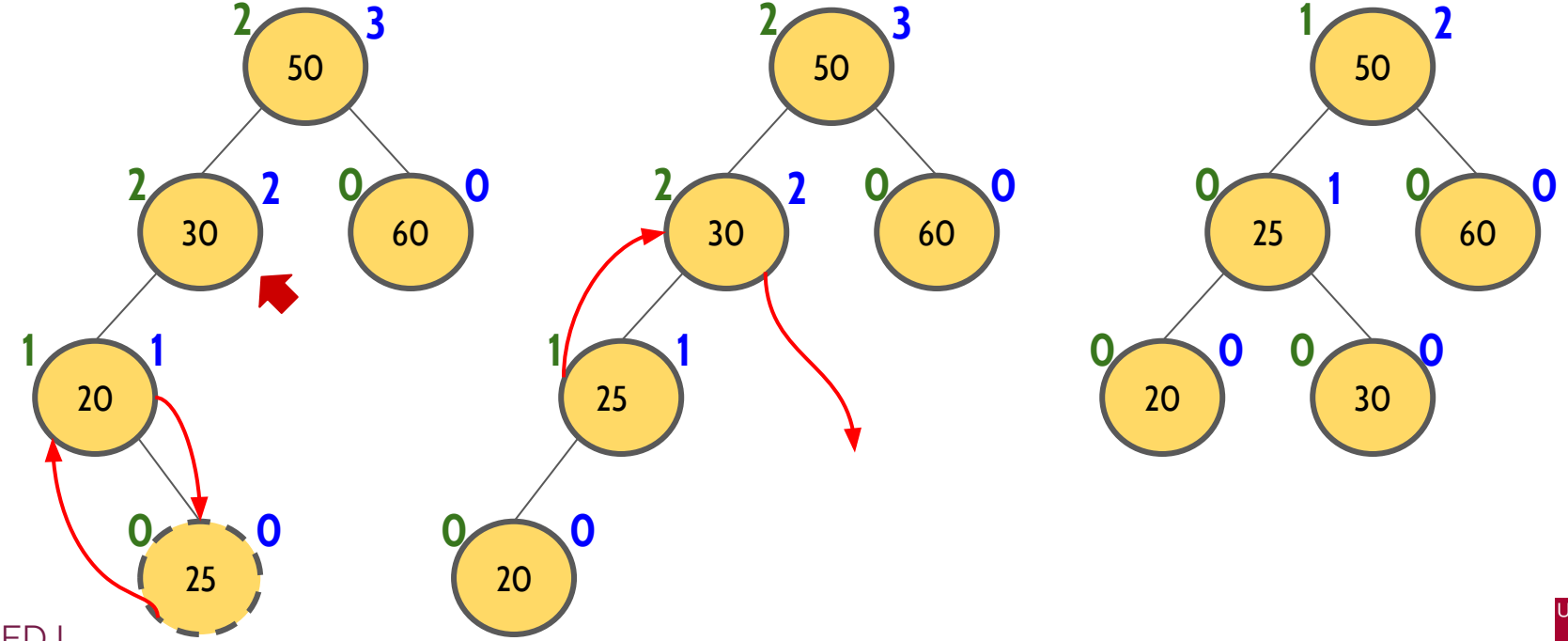
rotación simple derecha



inserción en un árbol AVL

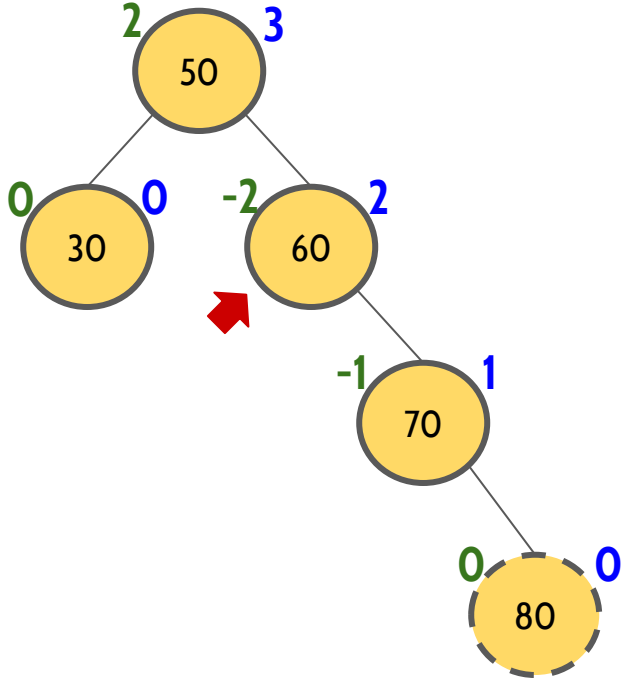
```
insertar(25);
```

doble rotación izquierda derecha

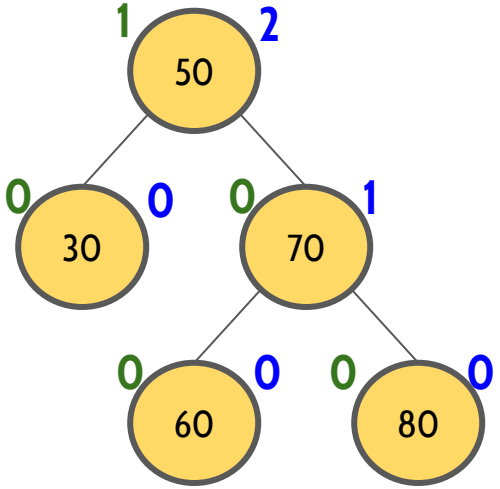


inserción en un árbol AVL

```
insertar(80);
```



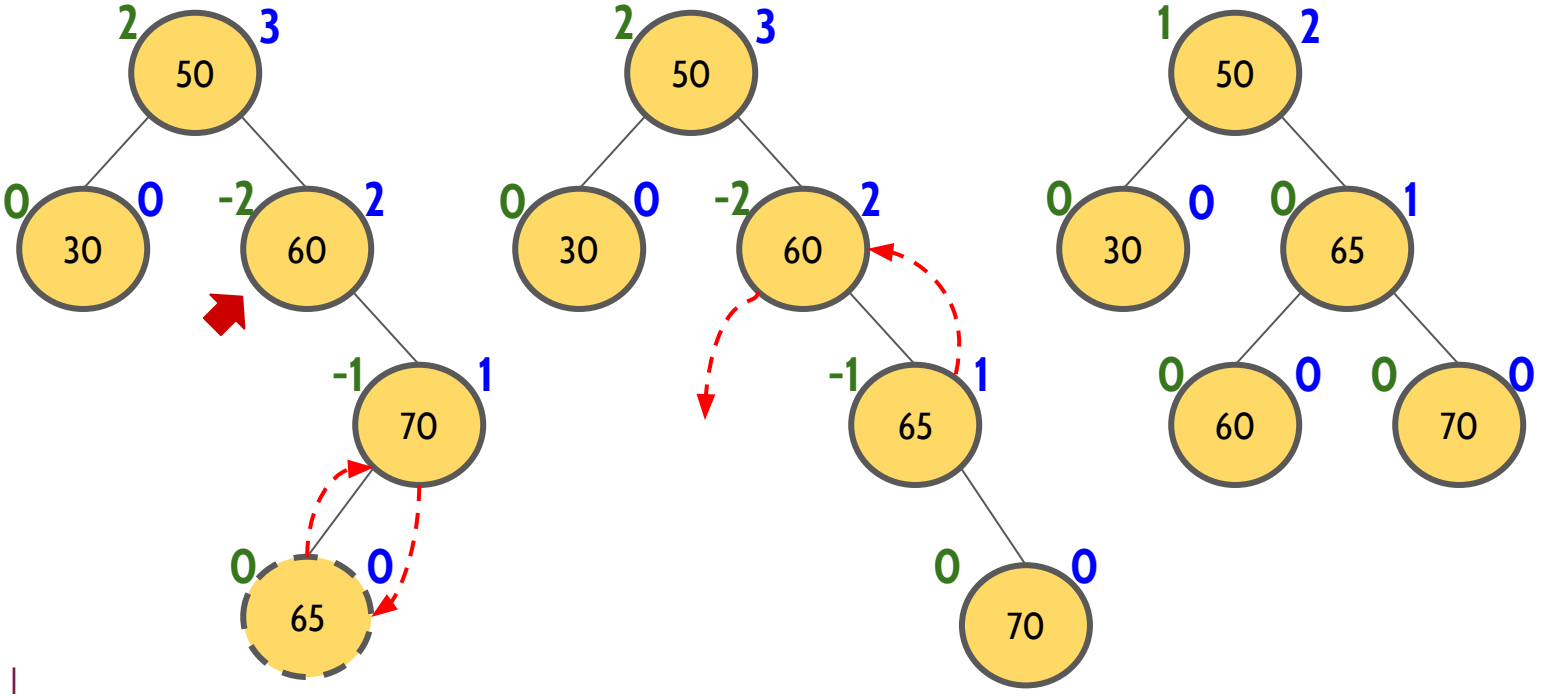
rotación simple izquierda



inserción en un árbol AVL

`insertar(65);`

doble rotación derecha izquierda



Ejercicio 5.17.

Determinar los estados intermedios y el estado final del árbol AVL en el que se inserta la siguiente secuencia de valores:

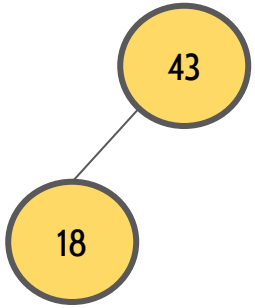
43, 18, 22, 9, 21, 6, 8, 20, 63, 50, 62, 51

Ejercicio: paso 1


```
insertar(43);
```

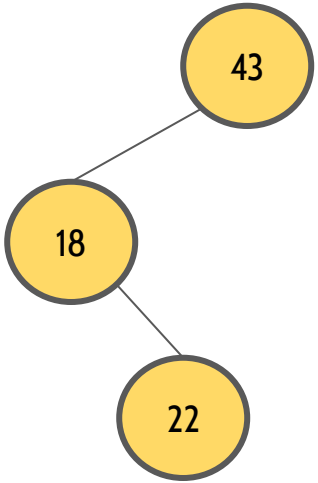
Ejercicio: paso 2

```
insertar(18);
```



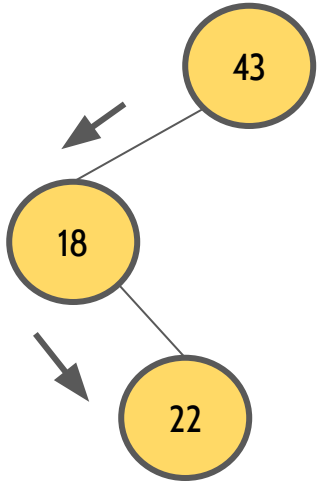
Ejercicio: paso 3

```
insertar(22);
```



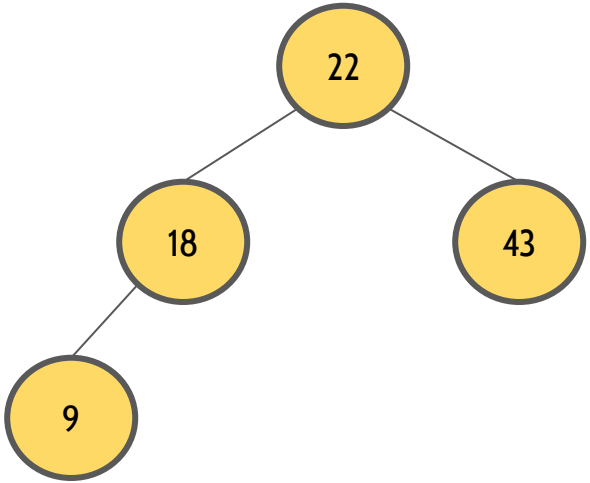
Ejercicio: paso 3

```
insertar(22);
rotación izq-der;
```



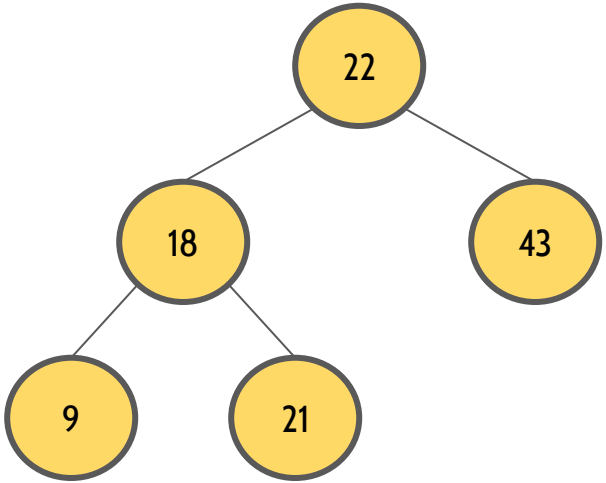
Ejercicio: paso 4

```
insertar(9);
```



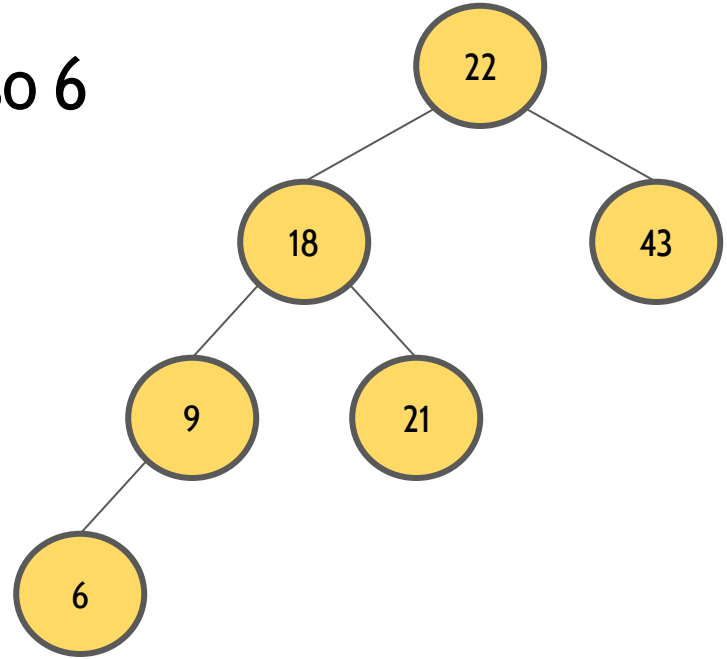
Ejercicio: paso 5

```
insertar(21);
```



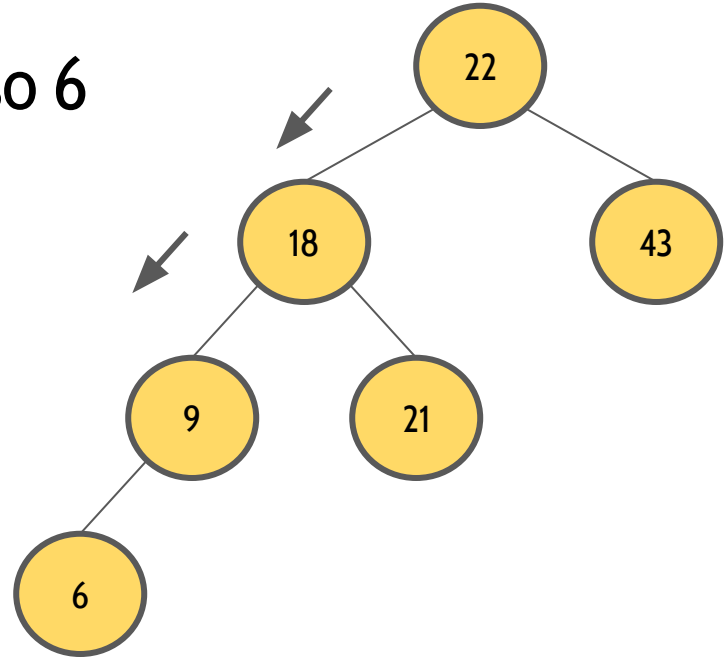
Ejercicio: paso 6

```
insertar(6);
```



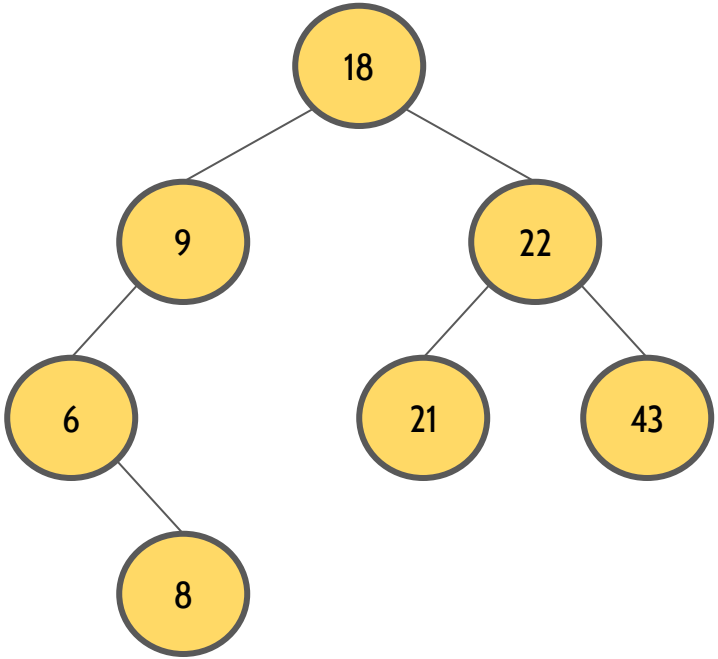
Ejercicio: paso 6

```
insertar(6);
rotación izq;
```



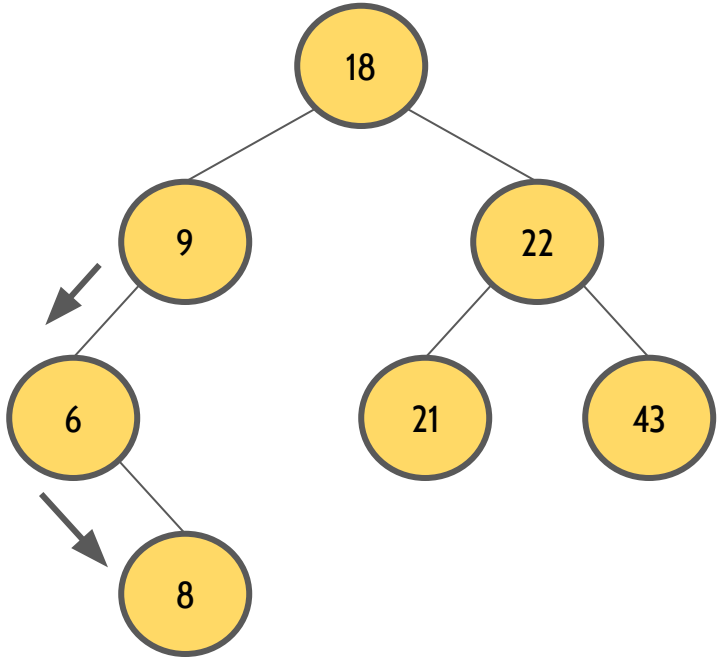
Ejercicio: paso 7

```
insertar(8);
```



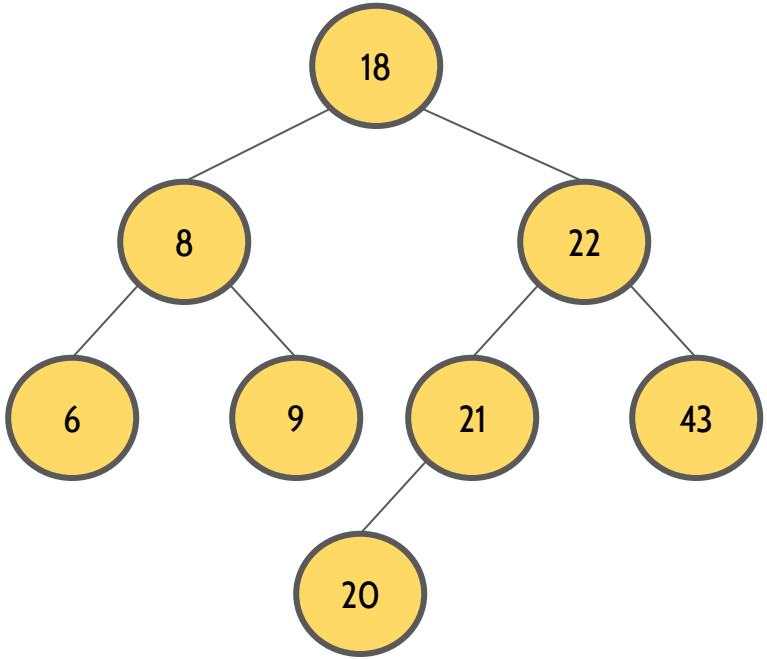
Ejercicio: paso 7

```
insertar(8);
rotación izq-der;
```



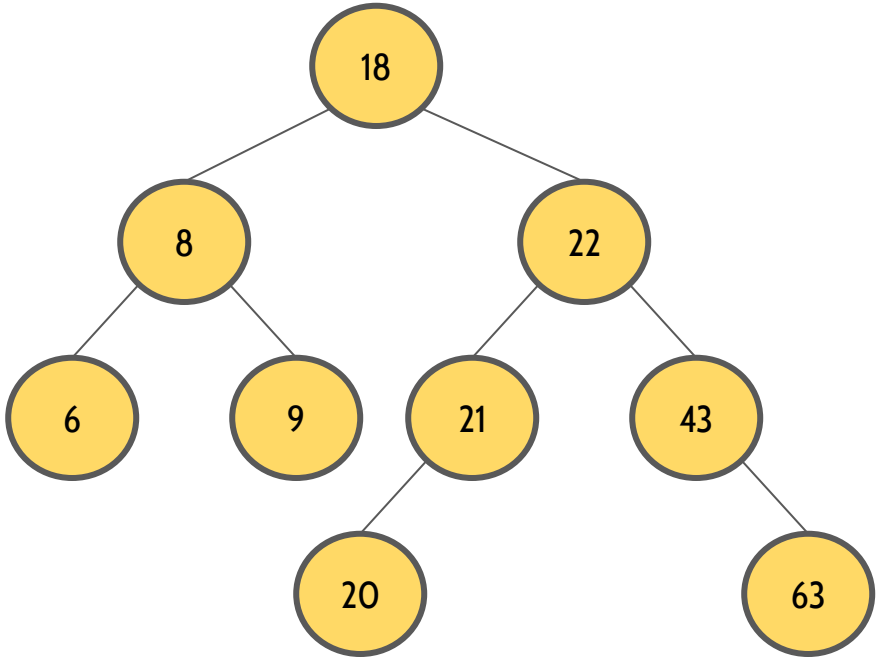
Ejercicio: paso 8

```
insertar(20);
```



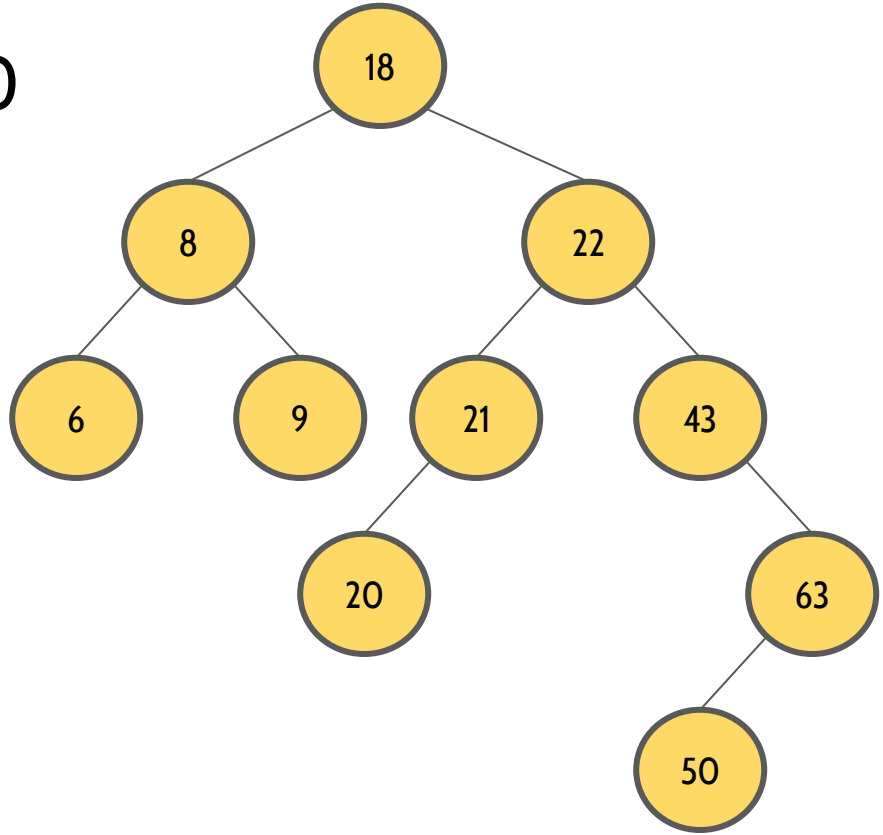
Ejercicio: paso 9

```
insertar(63);
```



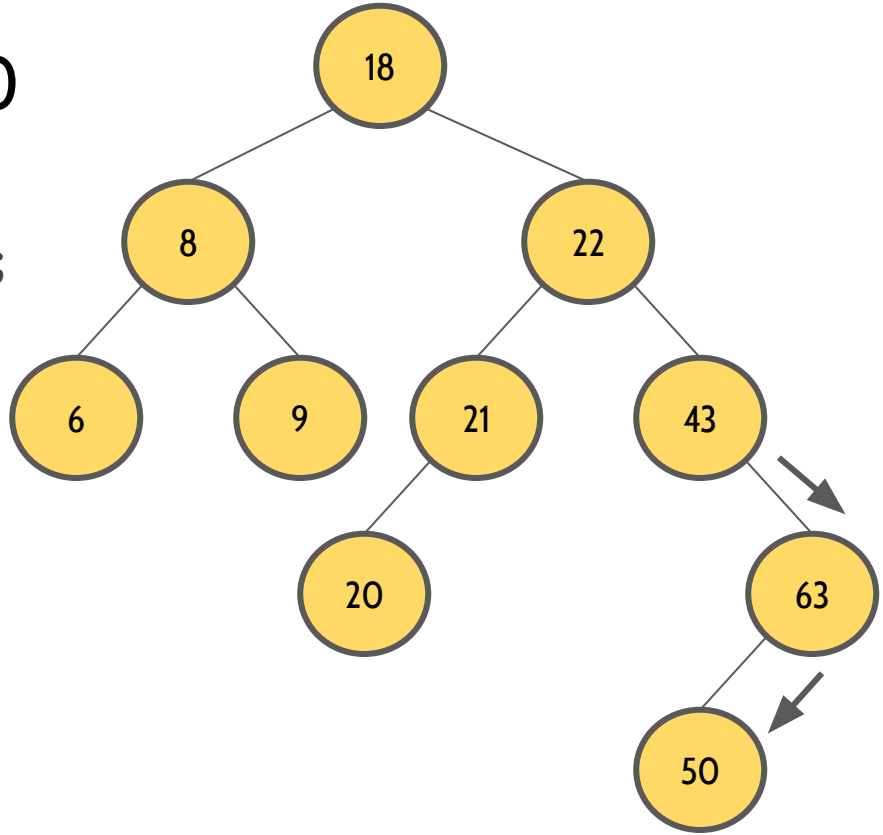
Ejercicio: paso 10

```
insertar(50);
```



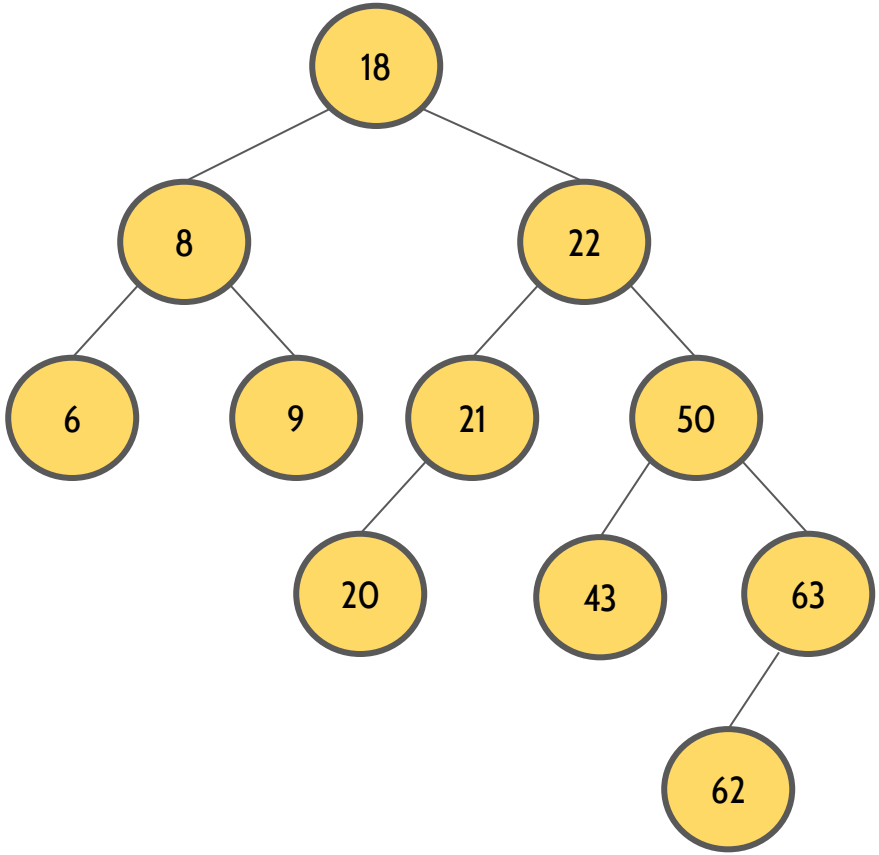
Ejercicio: paso 10

```
insertar(50);
rotación der-izq;
```



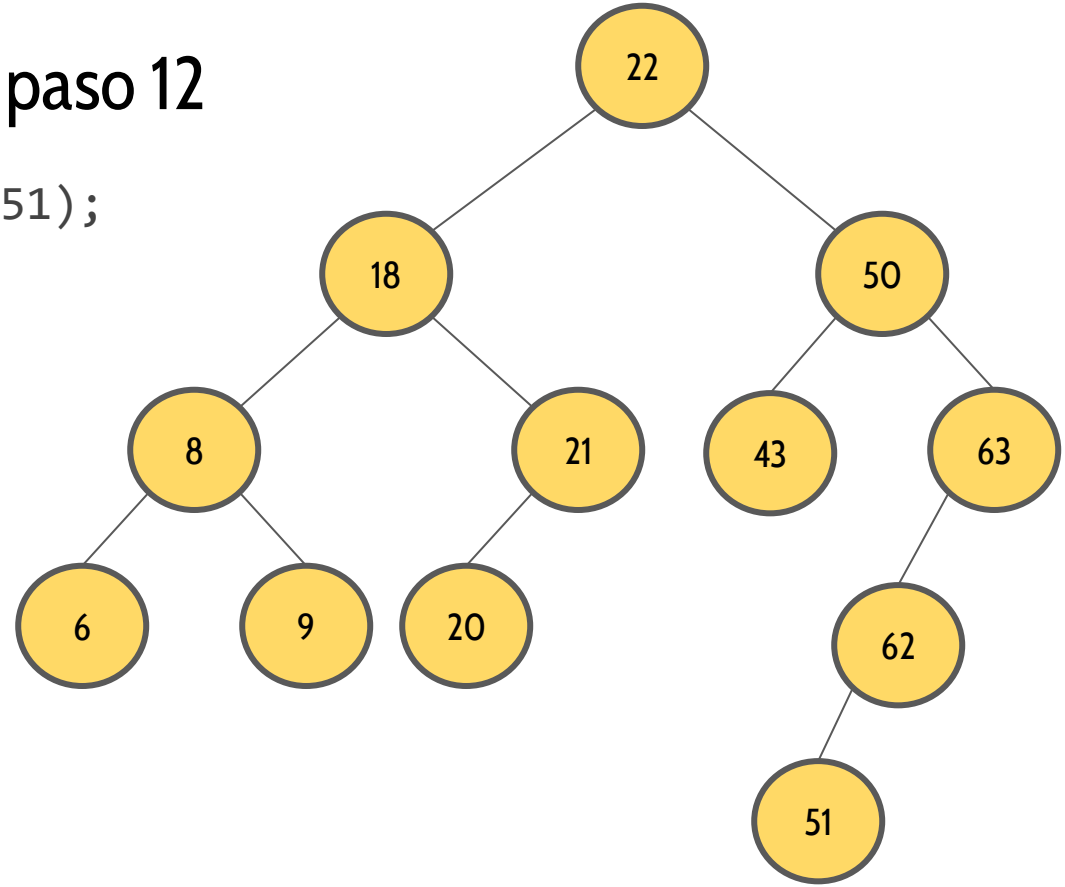
Ejercicio: paso 11

```
insertar(62);
```



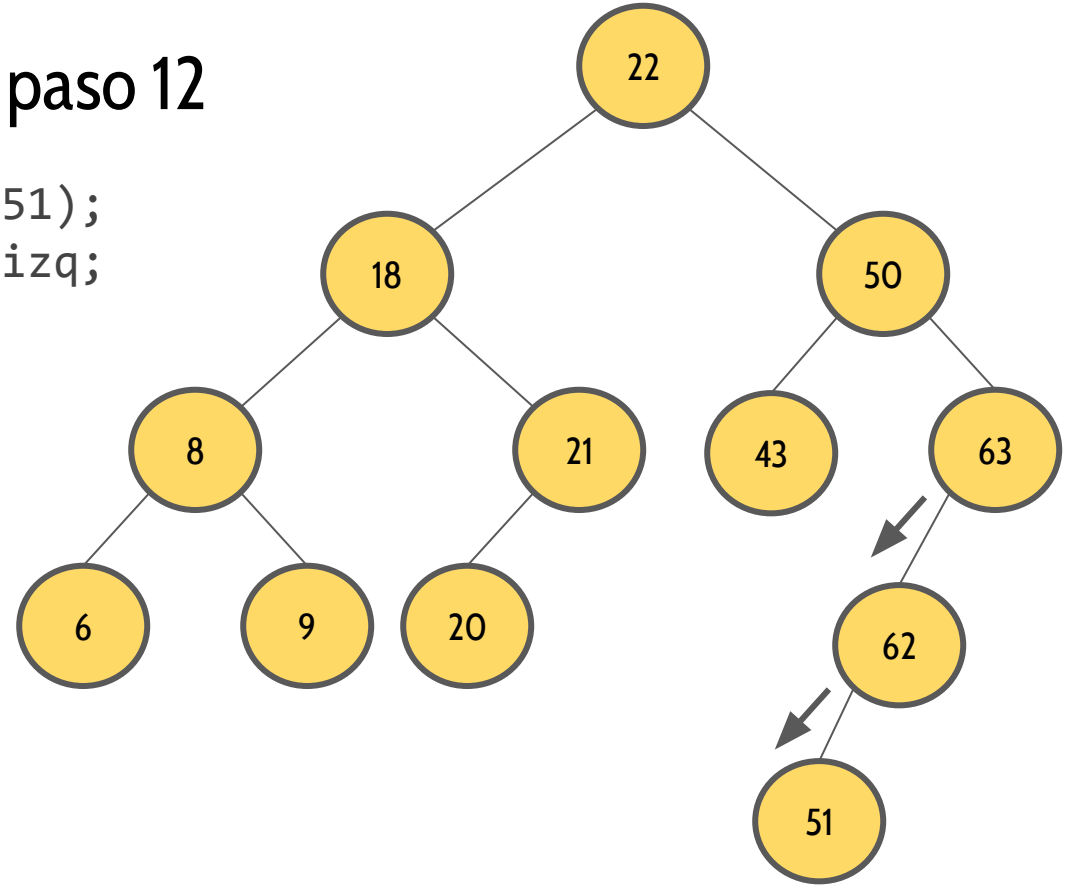
Ejercicio: paso 12

```
insertar(51);
```

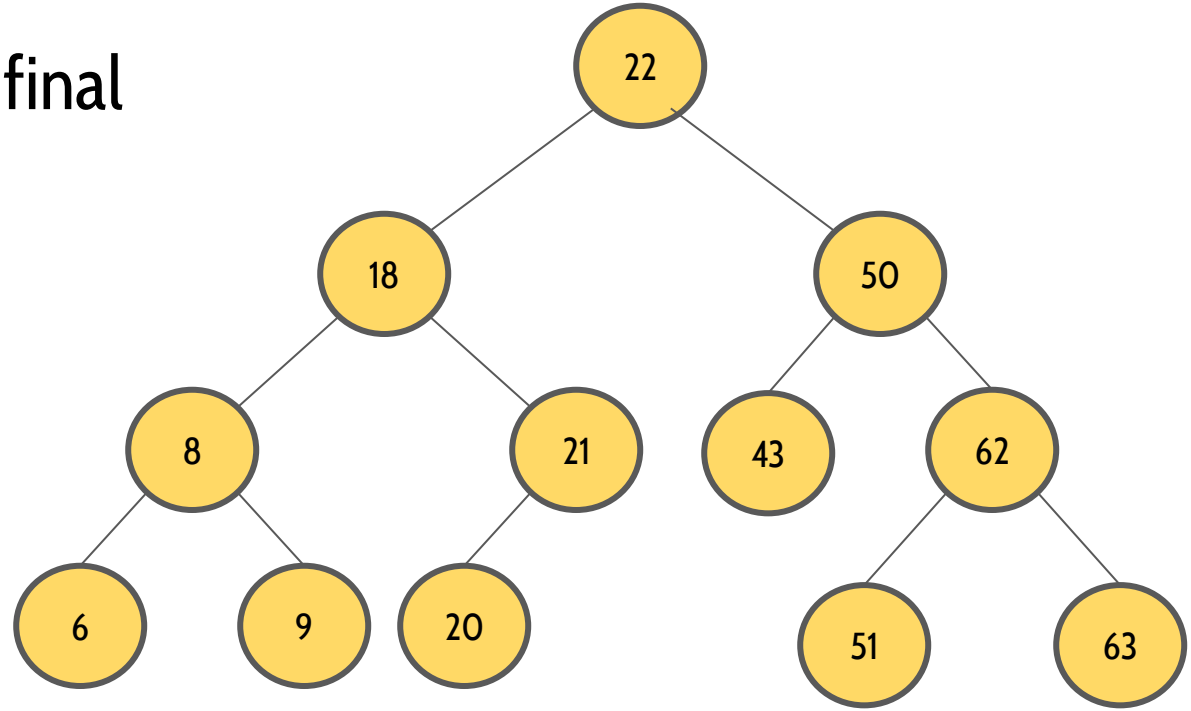


Ejercicio: paso 12

```
insertar(51);
rotacion izq;
```



Ejercicio: final



Ejercicio 5.18.

Determinar los estados intermedios y el estado final del árbol AVL en el que se inserta la siguiente secuencia de valores:

21, 26, 30, 9, 4, 14, 28, 18, 15, 10, 2, 3, 7

Ejercicio 5.19.

Implementar una clase que modele el comportamiento del árbol AVL

```
struct nodoAvl{
    int dato;
    nodoAvl* izquierdo;
    nodoAvl* derecho;
    int altura;
};
class ArbolAvl {
    private:
        nodoAvl* raiz;
        nodoAvl* crearNodo(int);
        nodoAvl* vaciarArbol(nodoAvl*);
        nodoAvl* insertar(int, nodoAvl*);
        void mostrarInOrden(nodoAvl*);
        nodoAvl* buscarMenor(nodoAvl*);
        nodoAvl* eliminar(nodoAvl*, int);
        int altura(nodoAvl*);
        nodoAvl* rotacionSimpleIzquierda(nodoAvl*&);
        nodoAvl* rotacionSimpleDerecha(nodoAvl*&);
        nodoAvl* rotacionDobleIzquierda(nodoAvl*&);
        nodoAvl* rotacionDobleDerecha(nodoAvl*&);
    public:
        ArbolAvl();
        ~ArbolAvl();
        void insertar(int);
        void mostrarInOrden();
        void eliminar(int);
};
```

```
nodoAvl* ArbolAvl::insertar(int dato,  nodoAvl* hoja){
    if(hoja == NULL)
    {
        hoja = crearNodo(dato);
    }
    else if(dato < hoja->dato)
    {
        hoja->izquierdo = insertar(dato, hoja->izquierdo);
        if(altura(hoja->izquierdo) - altura(hoja->derecho) == 2)
        {
              if(dato < hoja->izquierdo->dato)
                    hoja = rotacionSimpleDerecha(hoja);
              else
                    hoja = rotacionDobleDerecha(hoja);
        }
    }
    else if(dato > hoja->dato)
    {
        hoja->derecho = insertar(dato, hoja->derecho);
        if(altura(hoja->derecho) - altura(hoja->izquierdo) == 2)
        {
              if(dato > hoja->derecho->dato)
                    hoja = rotacionSimpleIzquierda(hoja);
              else
                    hoja = rotacionDobleIzquierda(hoja);
        }
    }

    hoja->altura = max(altura(hoja->izquierdo), altura(hoja->derecho))+1;
    return hoja;
}
```

```
nodoAvl* ArbolAvl::rotacionSimpleDerecha(nodoAvl* &hoja)
{
    nodoAvl* aux = hoja->izquierdo;
    hoja->izquierdo = aux->derecho;
    aux->derecho = hoja;
    hoja->altura = max(altura(hoja->izquierdo), altura(hoja->derecho))+1;
    aux->altura = max(altura(aux->izquierdo), hoja->altura)+1;
    return aux;
}
nodoAvl* ArbolAvl::rotacionSimpleIzquierda(nodoAvl* &hoja)
{
    nodoAvl* aux = hoja->derecho;
    hoja->derecho = aux->izquierdo;
    aux->izquierdo = hoja;
    hoja->altura = max(altura(hoja->izquierdo), altura(hoja->derecho))+1;
    aux->altura = max(altura(hoja->derecho), hoja->altura)+1 ;
    return aux;
}
nodoAvl* ArbolAvl::rotacionDobleIzquierda(nodoAvl* &hoja)
{
    hoja->derecho = rotacionSimpleDerecha(hoja->derecho);
    return rotacionSimpleIzquierda(hoja);
}
nodoAvl* ArbolAvl::rotacionDobleDerecha(nodoAvl* &hoja)
{
    hoja->izquierdo = rotacionSimpleIzquierda(hoja->izquierdo);
    return rotacionSimpleDerecha(hoja);
}
int ArbolAvl::altura(nodoAvl* hoja)
{
    return (hoja == NULL ? -1 : hoja->altura);
}
```

Tablas hash (tablas de dispersión)

hashing

Supongamos que queremos **almacenar clientes** de una campaña telefónica en un contact center. Estos clientes están identificados por el **número de teléfono.**

Sobre este directorio de clientes necesitamos realizar **eficientemente** las siguientes **operaciones**:

- **Insertar** un número telefónico y la información relacionada
- **Buscar** un número telefónico y la información relacionada
- **Eliminar** un número telefónico y la información relacionada

hashing

Consideremos las opciones que hemos estudiado hasta ahora:

	insertar	**buscar**	**eliminar**
arreglo	O(1)	O(n)	O(n)
lista enlazada	O(1)	O(n)	O(n)
arreglo ordenado	O(n)	O(log n)	O(n)
árbol binario balanceado	O(log n)	O(log n)	O(log n)

hashing

Consideremos otra alternativa: una tabla de acceso directo que consistiría en un gran arreglo donde yo pueda usar el número de teléfono como índice de ese arreglo.

Entonces las tres operaciones se pueden realizar en **O(1)**

En muchos casos, como en el que estamos estudiando, esta alternativa no es viable debido a limitaciones prácticas:

- Espacio requerido $O(m * 10^n)$
- El int no me permite direccionar todos los números de 10 dígitos (INT_MAX: +2147483647)

hashing

Consideremos otra alternativa: una tabla de acceso directo que consistiría en un gran arreglo donde yo pueda usar el número de teléfono como índice de ese arreglo.

Entonces las tres operaciones se pueden realizar en **O(1)**

En muchos casos, como en el que estamos estudiando, esta alternativa no es viable debido a limitaciones prácticas:

- Espacio requerido $O(m * 10^n)$
- El int no me permite direccionar todos los números de 10 dígitos (INT_MAX: +2147483647)

hashing

La solución para estas limitaciones es el **hashing**

El **hashing** nos permite obtener un tiempo de búsqueda promedio de **O(1)** (bajo algunas consideraciones) que representa una mejora considerable con respecto a las otras estructuras estudiadas

> La idea es usar una función (**función hash**) que convierta una clave (en nuestro caso el número de teléfono) en un entero de tamaño razonable y usar ese entero como índice de una tabla que llamamos **tabla hash**

función hash

función hash (o de dispersión): es una función que convierte una clave dada (string o entero grande) en un entero pequeño que puede ser utilizado como índice en una tabla hash

La función hash se debe resolver en tiempo constante (no puede involucrar operaciones por ejemplo para comprobar qué celdas están libres u ocupadas)

Una buena función hash debe:

- Ser de cálculo eficiente
- Distribuir las claves uniformemente

tabla hash

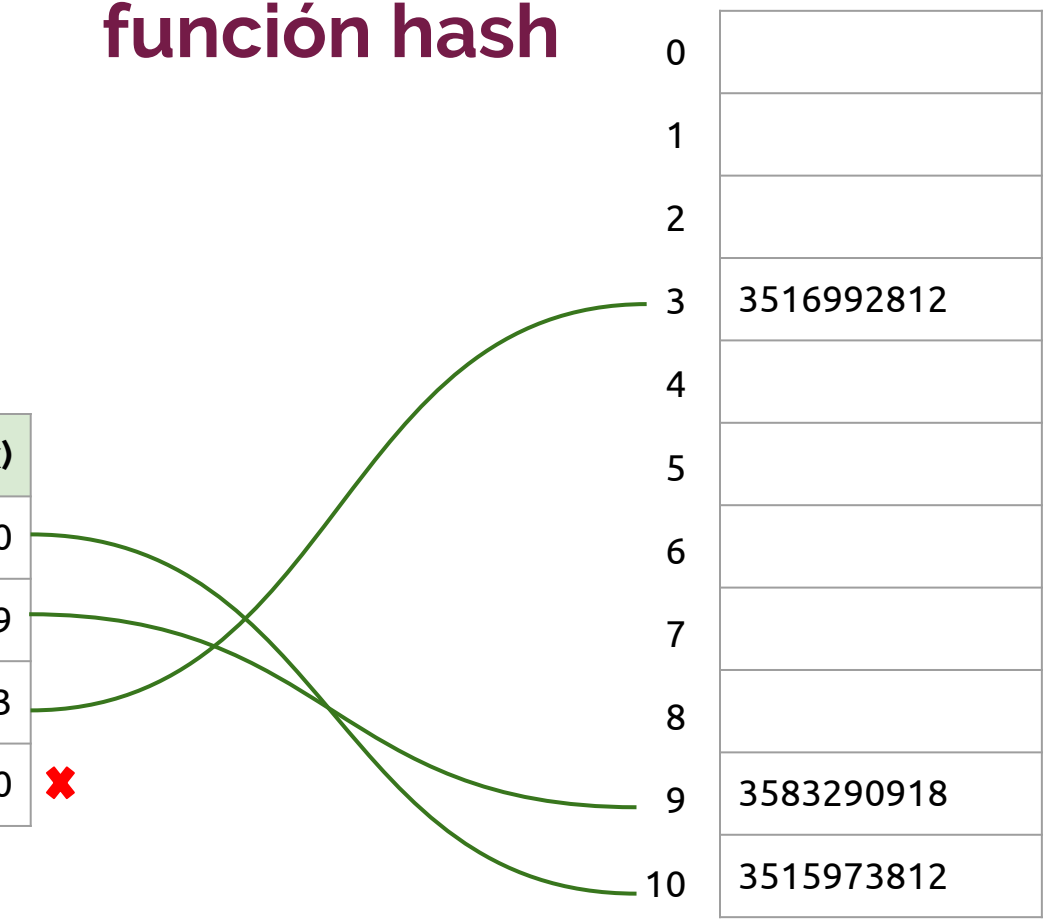
tabla hash (o matriz asociativa): una tabla que almacena los punteros a los registros correspondientes a una clave dada. Si la clave no ha sido asignada el valor de la entrada es NULL

función hash

función hash

f(x) = x mod 11

# teléfono	f(x)
3515973812	10
3583290918	9
3516992812	3
3515973845	10 ✖



colisiones

Como la función de hash asigna un número relativamente pequeño a un valor grande o un string, existe la posibilidad de que para **dos claves distintas** la función arroje el **mismo resultado**.

A esta situación se la conoce como **colisión** y debe ser administrada de alguna manera para que la técnica sea efectiva.

colisiones

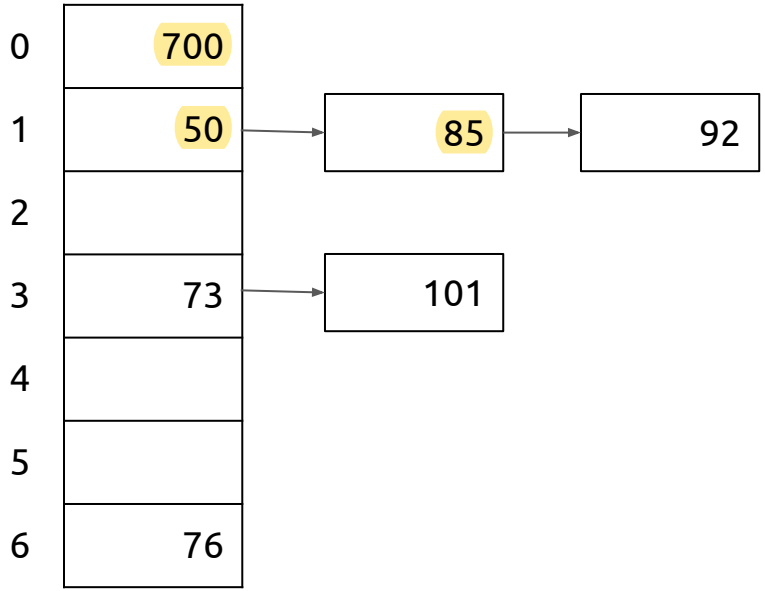
Existen dos tipos de mecanismos de resolución de colisiones:

Encadenamiento:
La celda de la tabla hash apunta a una lista enlazada que contiene todos los registros con el mismo valor de la función hash

Direccionamiento abierto:
Todos los elementos se almacenan en la misma tabla hash. Cuando se busca un elemento, se examinan los casilleros uno por uno hasta que se encuentra el elemento o se determina que el elemento no está en la tabla

encadenamiento enlazado (chaining)

Consideremos la función hash f(x) = x mod 7 y la secuencia de claves:
50, 700, 76, 85, 92, 73, 101



encadenamiento enlazado

ventajas:

- fácil de implementar
- nunca se llena
- menos sensible a la función de hash
- se utiliza cuando se conoce poco de cuántos y qué tan frecuentemente se insertan y eliminan las claves

desventajas:

- mala performance de cache, ya que las claves se guardan en la lista enlazada
- desperdicio de espacio (celdas no usadas en la tabla)
- si las listas se vuelven largas, el tiempo de búsqueda se acerca a O(n)
- se utiliza espacio extra para mantener las listas enlazadas

encadenamiento enlazado

Eficiencia

llamamos:
m: número de celdas en la tabla
n: número de claves a ser insertadas
α = factor de carga

α = n/m

Tiempo de búsqueda, inserción y eliminación: **O(1+α)**

Si **α** es **O(1)** el tiempo de las operaciones es **O(1)**

direccionamiento abierto

Es otro mecanismo de resolución de colisiones en el cual los elementos se almacenan **siempre en la misma tabla.** De modo que siempre el tamaño de la tabla debe ser mayor que la cantidad de claves a ser almacenadas.

Insertar: se continúa explorando hasta que una celda esté libre, si se encuentra una celda libre, se inserta la clave.
Buscar: se explora hasta que se encuentra la clave o se encuentra una celda vacía
Eliminar: para eliminar, no se pone como vacía una celda simplemente, porque fallaría la búsqueda. Cuando se elimina una clave, se marca la celda como borrada.

direccionamiento abierto

Es otro mecanismo de resolución de colisiones en el cual los elementos se almacenan **siempre en la misma tabla.** De modo que siempre el tamaño de la tabla debe ser mayor que la cantidad de claves a ser almacenadas.

Insertar: se continúa explorando hasta que una celda esté libre, si se encuentra una celda libre, se inserta la clave.
Buscar: se explora hasta que se encuentra la clave o se encuentra una celda vacía
Eliminar: para eliminar, no se pone como vacía una celda simplemente, porque fallaría la búsqueda. Cuando se elimina una clave, se marca la celda como borrada.

direccionamiento abierto

Existen distintas variantes para implementar el direccionamiento abierto:

1. Exploración lineal: se explora la próxima celda mediante saltos lineales (típicamente el salto es 1, pero se puede elegir otro factor

Supongamos que m es el tamaño de la tabla hash

```
si la celda hash(x) % m está llena, probamos (hash(x) + 1) % m
si la celda (hash(x) + 1) % m está también llena, probamos (hash(x) + 2) % m
si la celda (hash(x) + 2) % m está también llena, probamos (hash(x) + 3) % m
```

direccionamiento abierto

insertar (76)
76 % 7 = 6

0	
1	
2	
3	
4	
5	
6	76

insertar (93)
93% 7 = 2

0	
1	
2	93
3	
4	
5	
6	76

insertar (40)
40 % 7 = 5

0	
1	
2	93
3	
4	
5	40
6	76

insertar (47)
47% 7 = 5

0	47
1	
2	93
3	
4	
5	40
6	76

insertar (10)
10% 7 = 3

0	47
1	
2	93
3	10
4	
5	40
6	76

insertar (55)
55% 7 = 6

0	47
1	55
2	93
3	10
4	
5	40
6	76

direccionamiento abierto

2. Exploración cuadrática: se explora la próxima celda mediante saltos cuadráticos i^2

Supongamos que m es el tamaño de la tabla hash

```
si la celda hash(x) % m está llena, probamos (hash(x)+ 1*1) % m
si la celda (hash(x)+ 1*1) % m está también llena, probamos (hash(x)+ 2*2) % m
si la celda (hash(x)+ 2*2) % m está también llena, probamos (hash(x)+ 3*3) % m
```

direccionamiento abierto

3. Doble hashing: usamos una segunda función hash (hash2) para los siguientes intentos

Supongamos que m es el tamaño de la tabla hash

```
si la celda hash(x)% m está llena, probamos (hash(x)+ 1*hash2(x))% m
si la celda (hash(x)+ 1*hash2(x))% m está tambien llena, probamos (hash(x)+ 2*hash2(x))% m
si la celda (hash(x)+ 2*hash2(x))% m está tambien llena, probamos (hash(x)+ 3*hash2(x))% m
```

direccionamiento abierto

Eficiencia

llamamos:
m: número de celdas en la tabla
n: número de claves a ser insertadas
α = factor de carga

α = n/m (<1)

Tiempo de búsqueda, inserción y eliminación: **O(1/(1-α))**

aplicaciones de las tablas hash

Correctores sintácticos
Diccionarios
Tablas de símbolos en compiladores

desventajas de las tablas hash

Para almacenar n elementos, necesitamos una tabla de m > n para que el factor de carga sea reducido, y las operaciones sobre la tabla sean eficientes

No es posible calcular el mínimo y máximo con un costo inferior a O(m)

No es posible ordenar con un costo inferior a O(m)

Ejercicio 5.20.

Implementar una tabla hash que utilice exploración lineal como mecanismo de resolución de colisiones

Ejercicio 5.21.

Hacer las modificaciones necesarias a la implementación de la tabla hash para que almacene nombres de personas. Comprobar su correcto funcionamiento.

Ejercicio 5.22.

Implementar método que cuente la cantidad de hojas de un árbol de búsqueda binaria

Ejercicio 5.23.

Implementar método que muestre en que nivel (profundidad) está cada nodo de un árbol binario de búsqueda

Ejercicio 5.24.

Implementar método que verifique si un árbol de búsqueda binaria está o no balanceado en altura

Ejercicio 5.25.

Implementar método que retorne la distancia entre dos nodos cualesquiera de un árbol de búsqueda binaria

artículos

How Radix trees made blocking IPs 5000 times faster
https://blog.sqreen.io/demystifying-radix-trees/