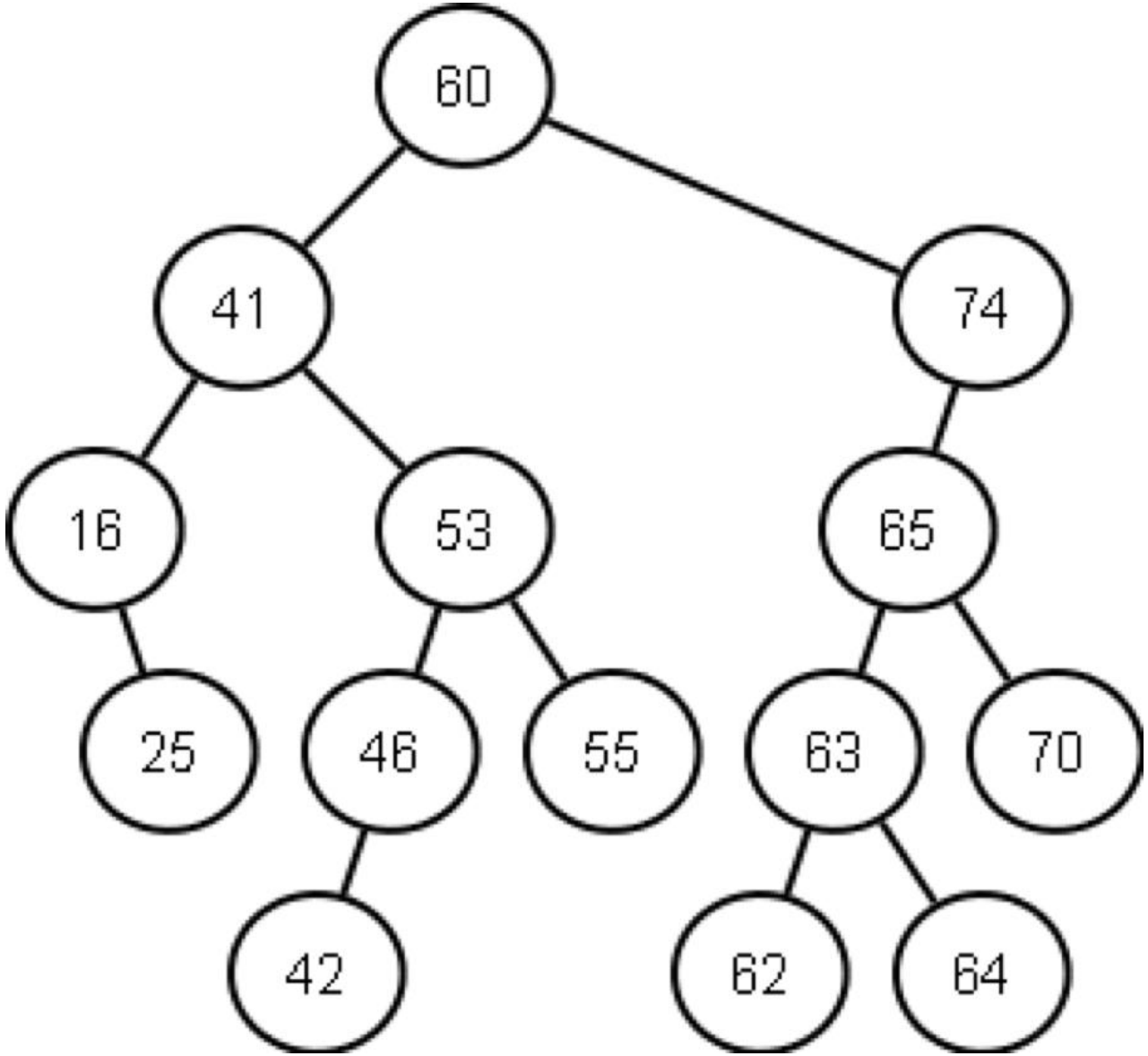
ESTRUCTURAS DE DATOS

LIC. REDES Y SERVICIOS DE CÓMPUTO

Clase

- Unidad VII. Árboles
  - Fundamentos teóricos
  - Tipos de árboles
  - Implantación de árboles binarios de búsqueda
  - Operaciones básicas de árboles binarios de búsqueda
  - Operaciones complementarias de árboles binarios de búsqueda
  - Aplicaciones de árboles binarios de búsqueda.



Un árbol se puede definir como una estructura jerárquica y en forma no lineal, aplicada sobre una colección de elementos u objetos llamados nodos.

(Cairó & Guardati, 2006).

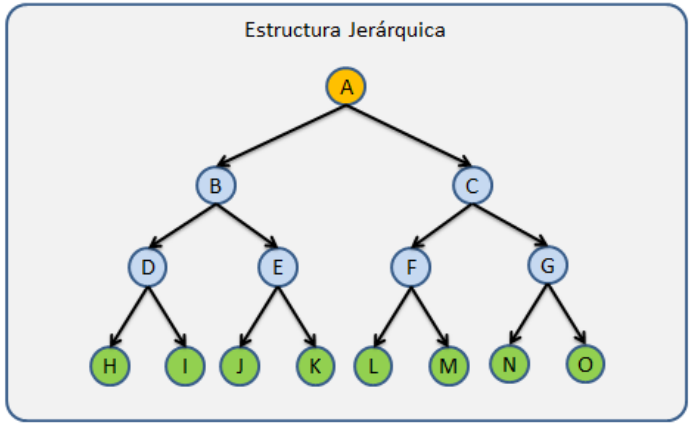
Árboles

- Los árboles son considerados las estructuras de datos **no lineales** y **dinámicas** de datos muy importantes del área de computación.
- Los árboles son muy utilizados en informática como un método eficiente para búsquedas grandes y complejas.
- Casi todos los sistemas operativos almacenan sus archivos en árboles o estructuras similares a árboles.

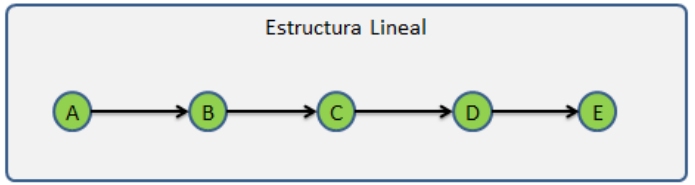
Árboles

Se les llama estructuras dinámicas, porque las mismas pueden cambiar tanto de forma como de tamaño durante la ejecución del programa.

Y estructuras no lineales porque cada elemento del árbol puede tener más de un sucesor



V.S.



Estructuras estáticas	Estructuras dinámicas
Arreglos	Listas
Registros	Árboles
	Gráficas

Estructuras lineales	Estructuras no lineales
Arreglos	Árboles
Registros	Gráficas
Pilas	
Colas	
Listas	

Características de los árboles

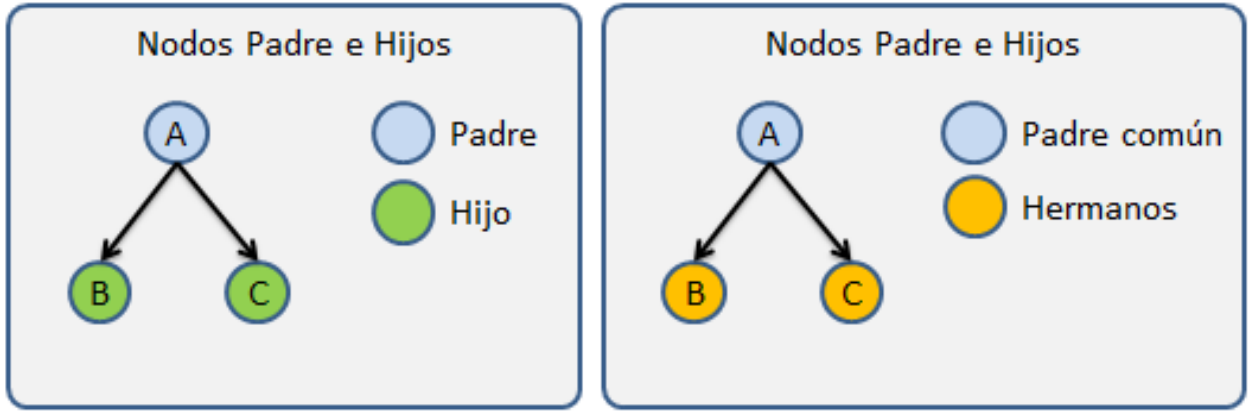
En relación con otros nodos

Nodos. Se le llama nodo a cada elemento que contiene el árbol

Nodo padre. Se utiliza este término para llamar a todos aquellos nodos que tienen al menos un hijo.

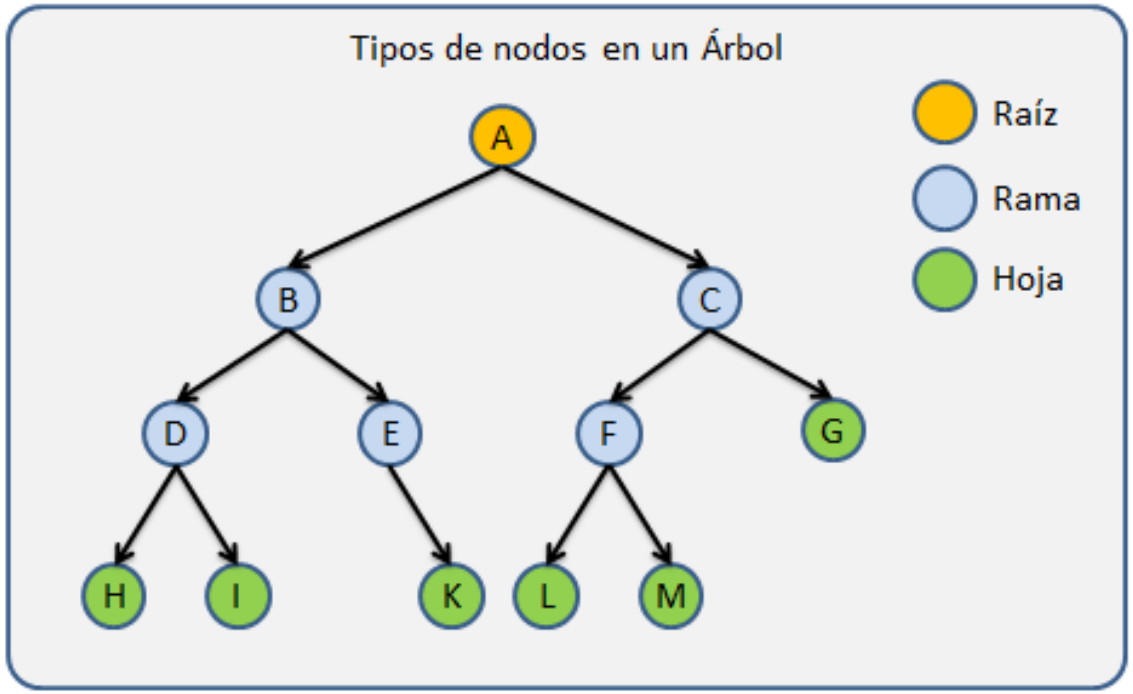
Nodo hijo. Los hijos son todos aquellos nodos que tienen un padre.

Nodo hermano. Los nodos hermanos son aquellos nodos que comparten un mismo padre en común dentro de la estructura.



En relación a la posición dentro del árbol

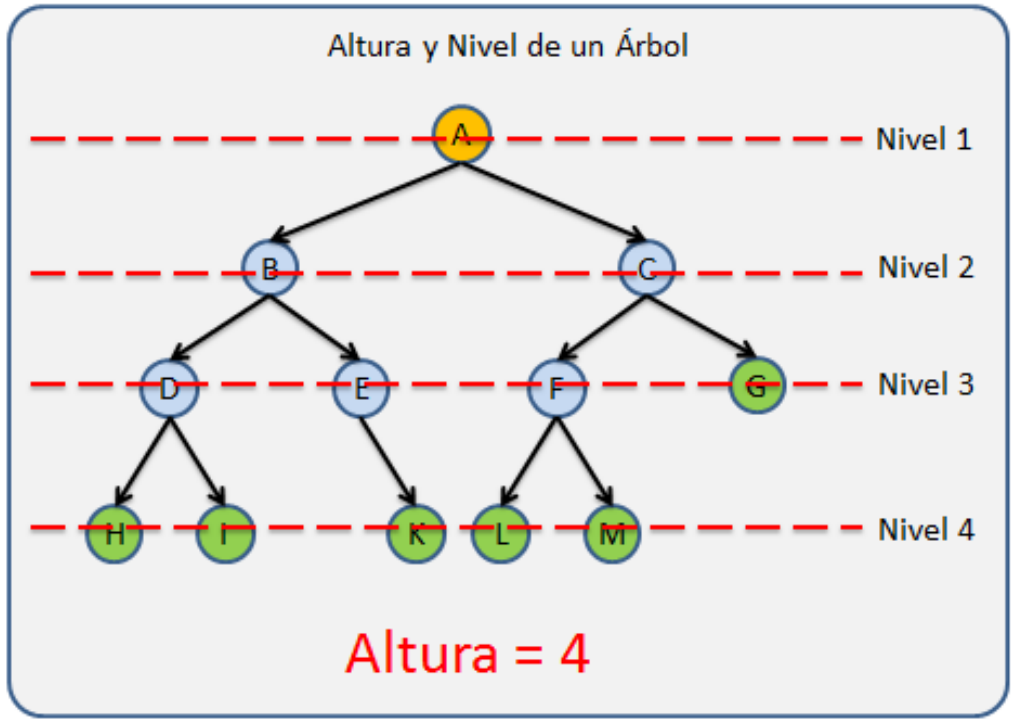
- **Nodo Raíz**. Se refiere al primer nodo de un árbol, Solo un nodo del árbol puede ser la raíz.
- **Nodo Hoja**. Son todos aquellos nodos que no tienen hijos, los cuales siempre se encuentran en los extremos de la estructura.
- **Nodo Interior o Rama**. Estos son todos aquellos nodos que no son la raíz y que además tiene al menos un hijo.



En relación al tamaño del árbol

Nivel. El nivel de un nodo es su distancia a la raíz. Por lo tanto:

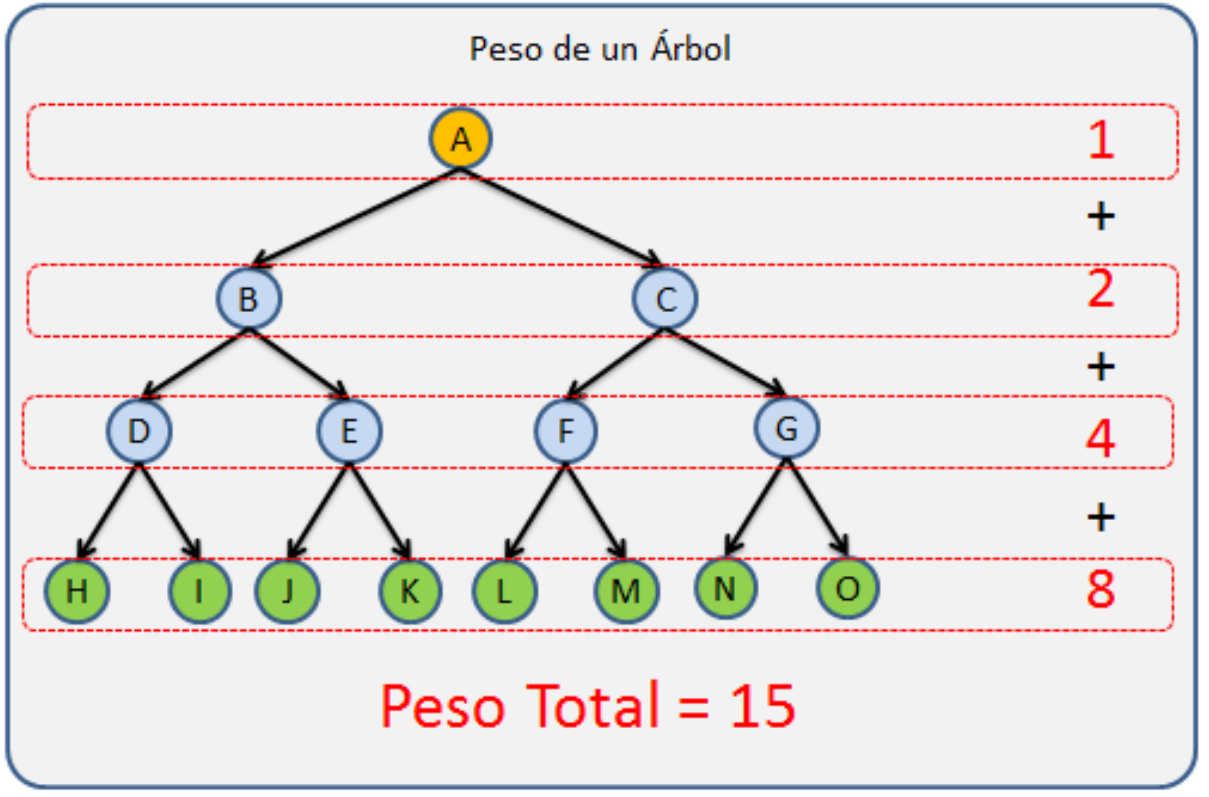
- Un *árbol* vacío tiene 0 niveles
- El nivel de la *raíz* es 1
- El nivel de cada nodo se calculado contando cuantos nodos existen sobre él, hasta llegar a la raíz + 1, y de forma inversa también se podría, contar cuantos nodos existen desde la *raíz* hasta el nodo buscado + 1.



Altura. Se le llama altura al número máximo de niveles de un *árbol*.

En relación al tamaño del árbol

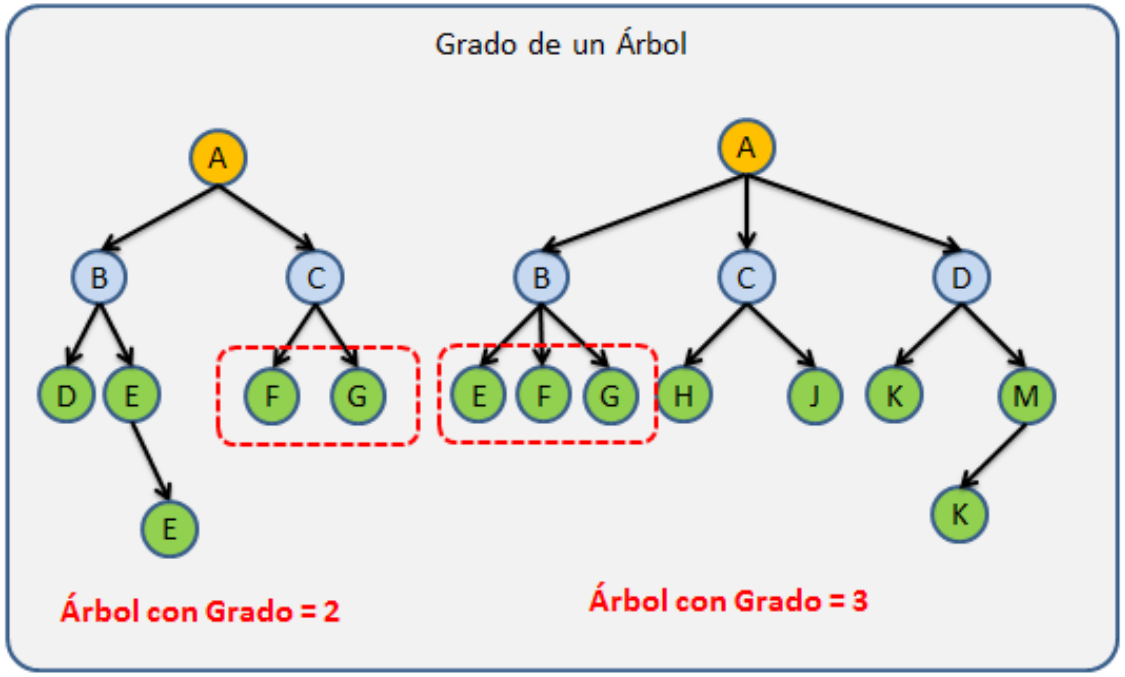
Peso. Es el número de nodos que tiene un árbol.



En relación al tamaño del árbol

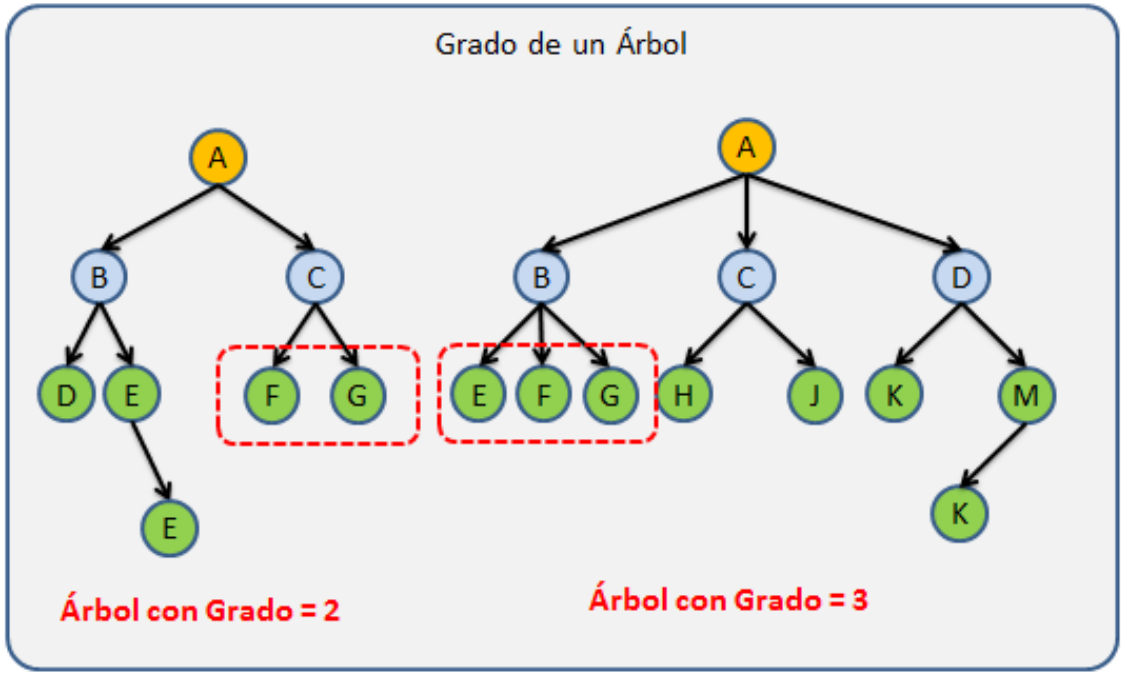
Orden. El Orden de un árbol es el número máximo de hijos que puede tener un Nodo. Es una constante que se define antes de crear el árbol.

Este valor no se calcula, si no que ya se conoce cuando se diseña la estructura.



En relación al tamaño del árbol

Grado. Número de hijos de un nodo y está limitado por el Orden, ya que este indica el número máximo de hijos que puede tener un nodo. El **grado de un árbol** se define como el máximo grado de todos sus nodos.

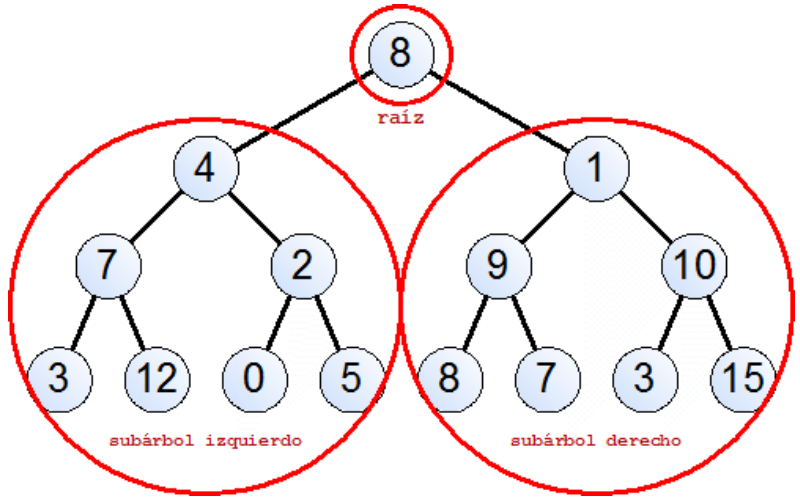


En relación al tamaño del árbol

Camino: Secuencia de nodos conectados dentro de un árbol.

Longitud del camino: Cantidad de nodos que se deben recorrer para llegar desde la raíz a un nodo determinado.

Sub-Árbol: Conocemos como Sub-Árbol a todo Árbol generado a partir de una sección determinada del Árbol, Por lo que podemos decir que un Árbol es un nodo Raíz con N Sub-Árboles.



Tipos de árboles

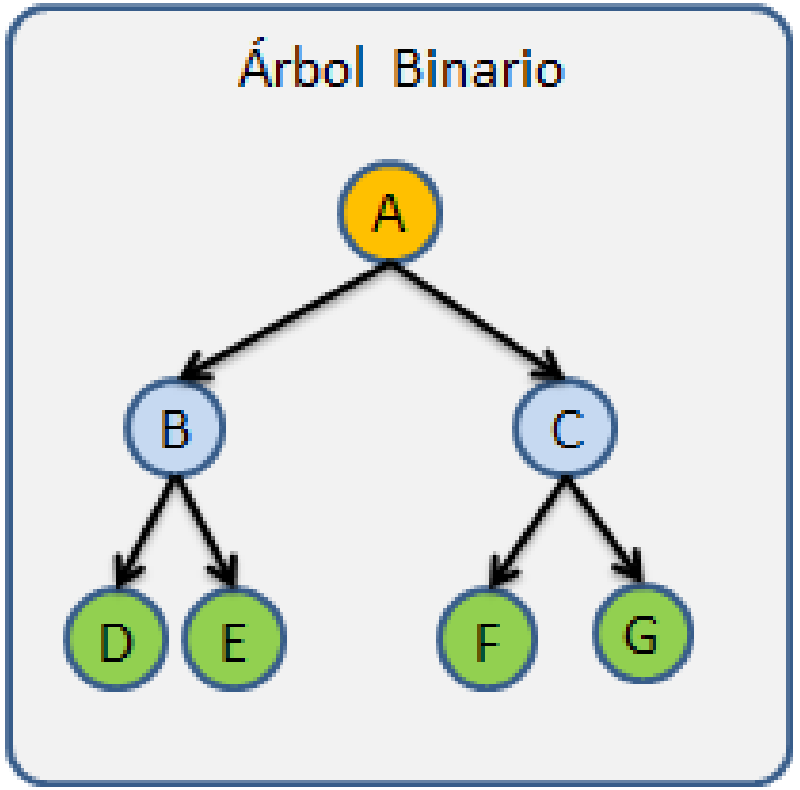
Tipos de árboles

Los arboles pueden clasificarse tomando en cuenta su estructura y funcionamiento. A continuación, se presentan los tipos de árboles más utilizados (Cairó & Guardati, 2006):

- Arboles binarios.
  - Arboles binarios distintos
  - Arboles binarios similares
  - Arboles binarios equivalentes
  - Arboles binarios completos
  - Arboles binarios llenos
  - Arboles binarios degenerados
  - Arboles binarios de búsqueda
  - Arboles equilibrados
- Arboles multicaminos
  - Arboles-B
  - Arboles B+
  - Arboles 2-4

Arboles binarios

Esta estructura se caracteriza por que cada nodo solo puede tener máximo 2 hijos, dicho de otra manera, es un árbol grado dos.

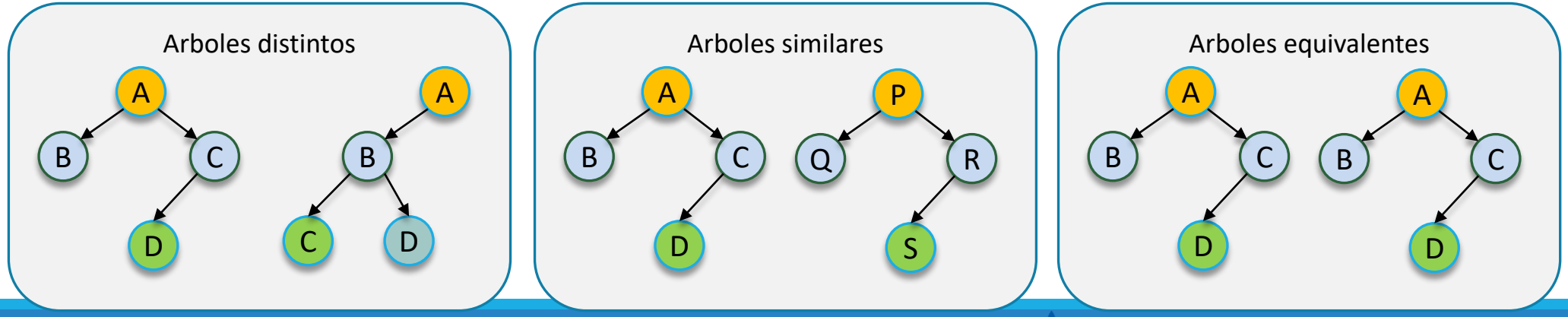


Árboles binarios distintos, similares y equivalentes

Árboles binarios distintos. Dos árboles binarios son distintos cuando sus estructuras son diferentes.

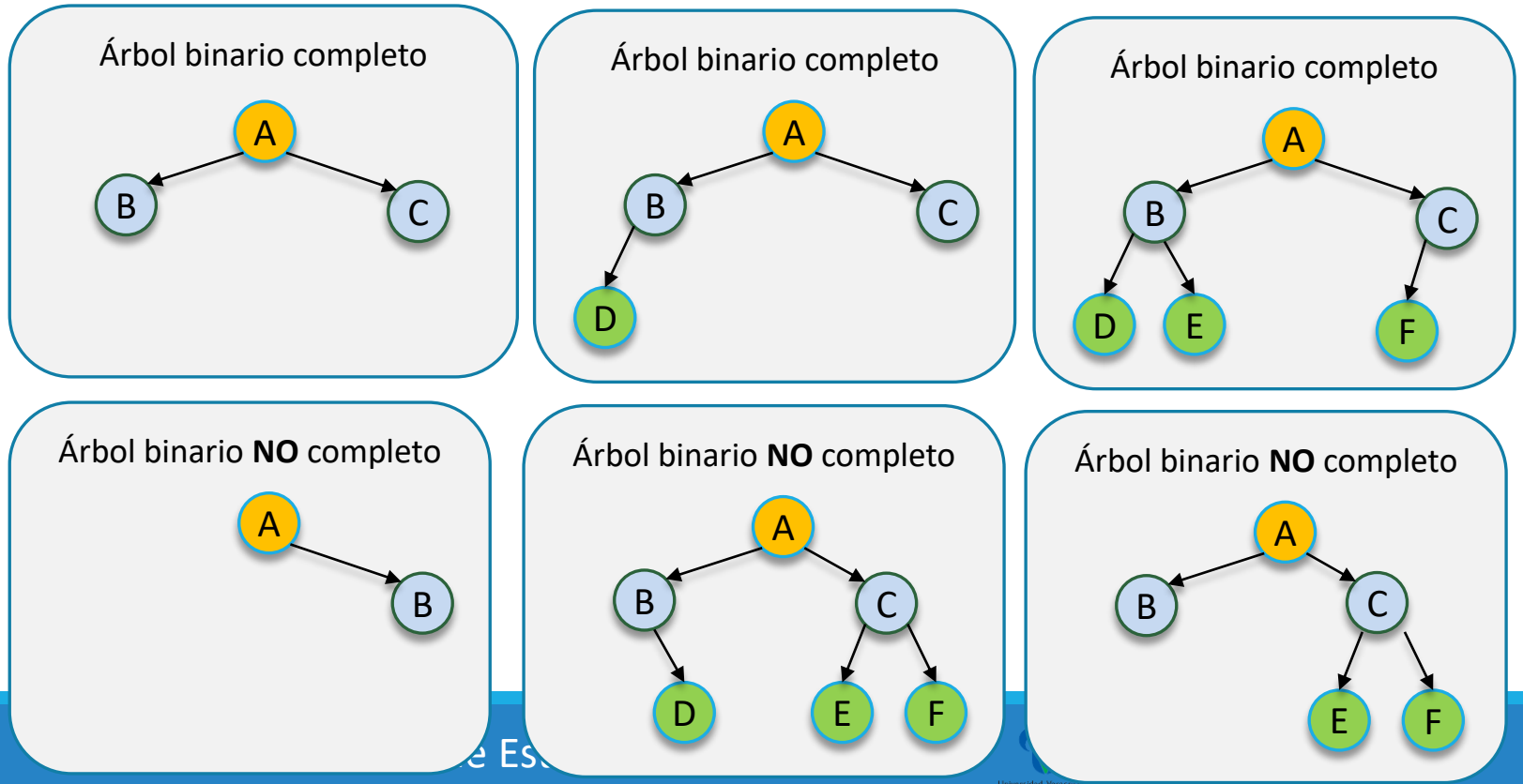
Árboles binarios similares. Dos árboles binarios son similares cuando sus estructuras son idénticas, pero la información que contienen sus nodos difiere entre sí.

Árboles binarios equivalentes. Los árboles binarios equivalentes se definen como aquellos que son similares y además los nodos contienen la misma información.



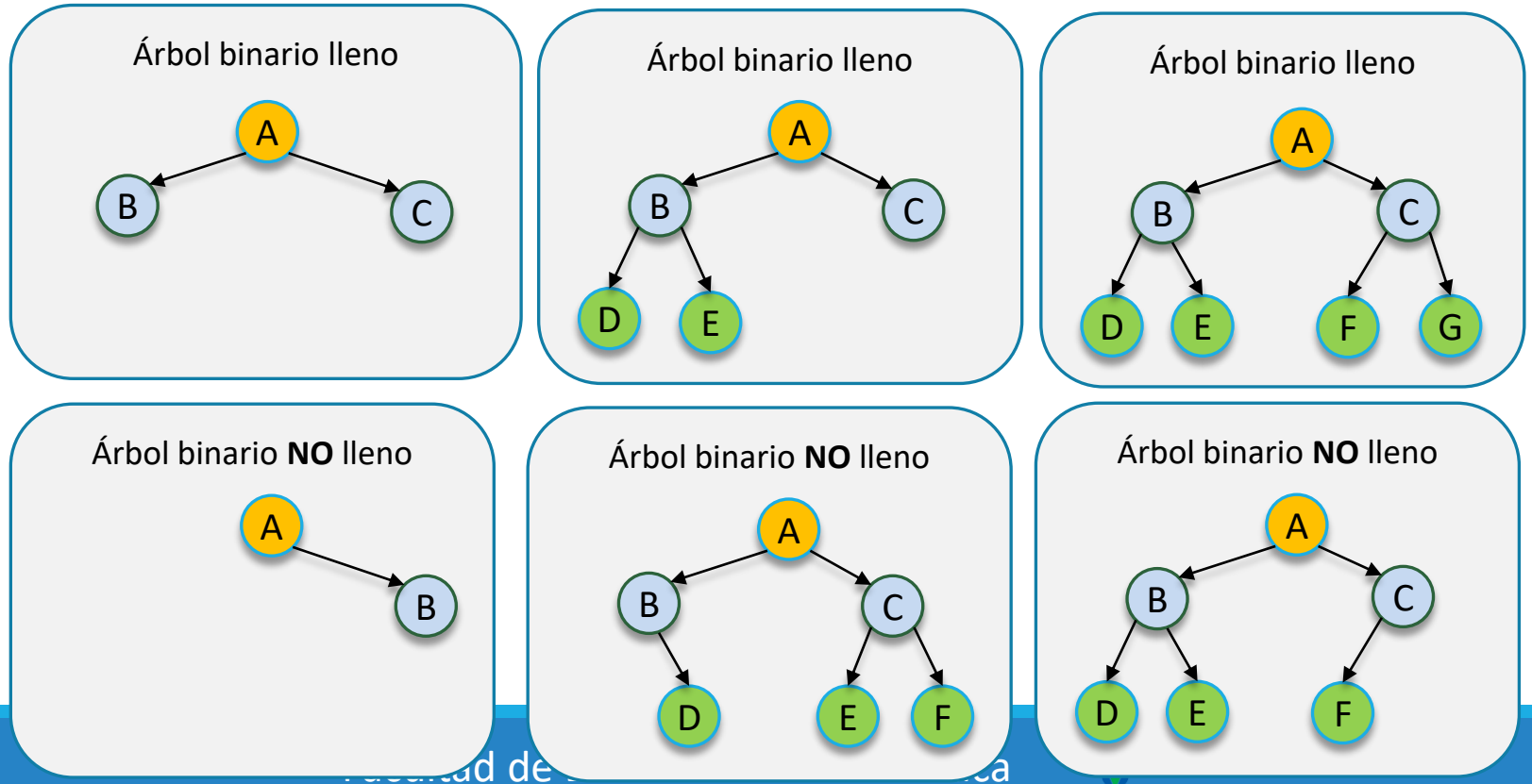
Árboles binarios completos

Un árbol binario completo de profundidad n es un árbol en el que, para cada nivel, del 0 al nivel n-1 tiene un conjunto lleno de nodos y todos los nodos hoja a nivel n ocupan las posiciones más a la izquierda del árbol



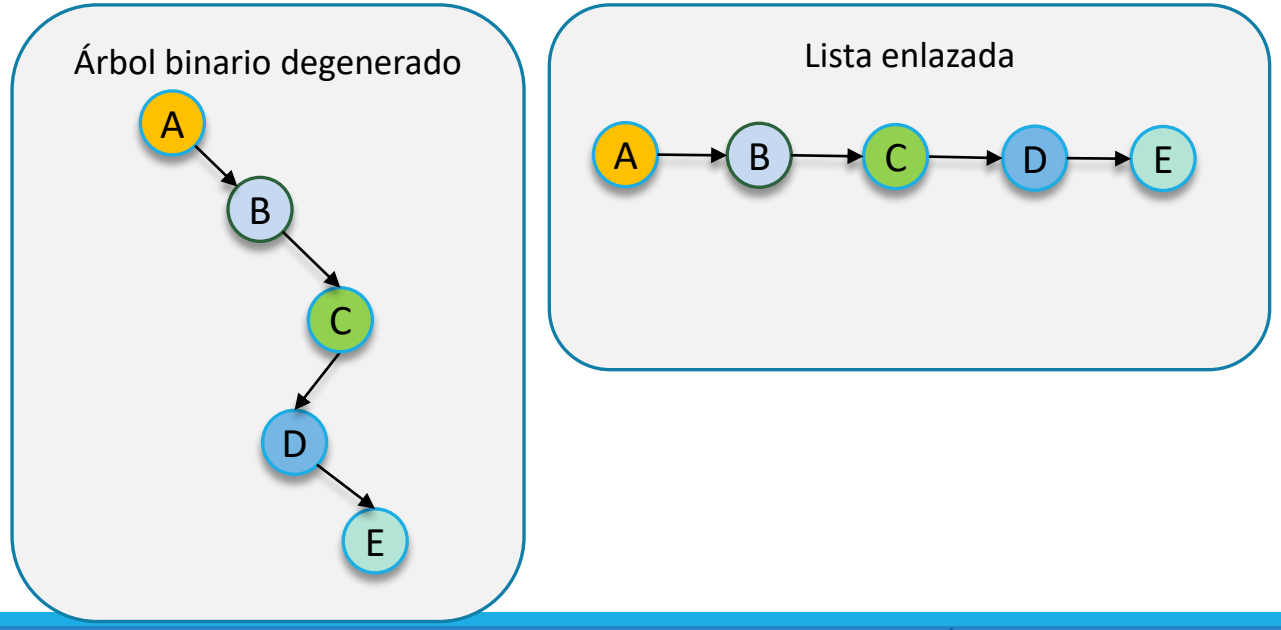
Árboles binarios llenos

Es un árbol lleno donde todos los nodos tienen cero o dos hijos. Es decir, no existe un nodo que tenga un solo hijo.



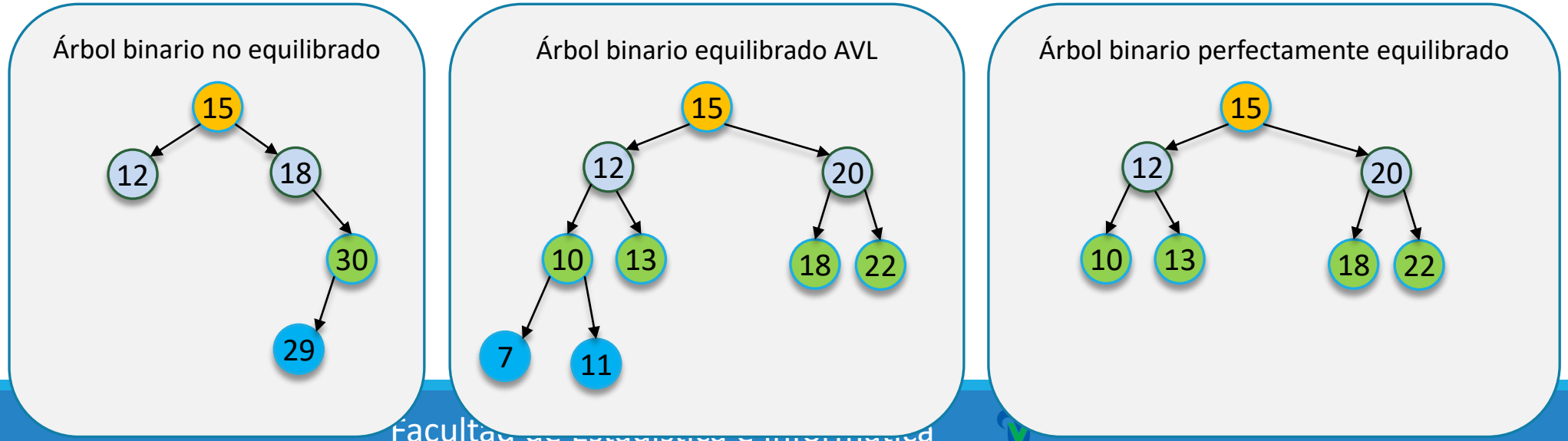
Árboles binarios degenerados

Es un tipo especial denominado árbol degenerado en el que hay un solo nodo hoja y cada nodo no hoja sólo tiene un hijo. Un árbol degenerado es equivalente a una lista enlazada.



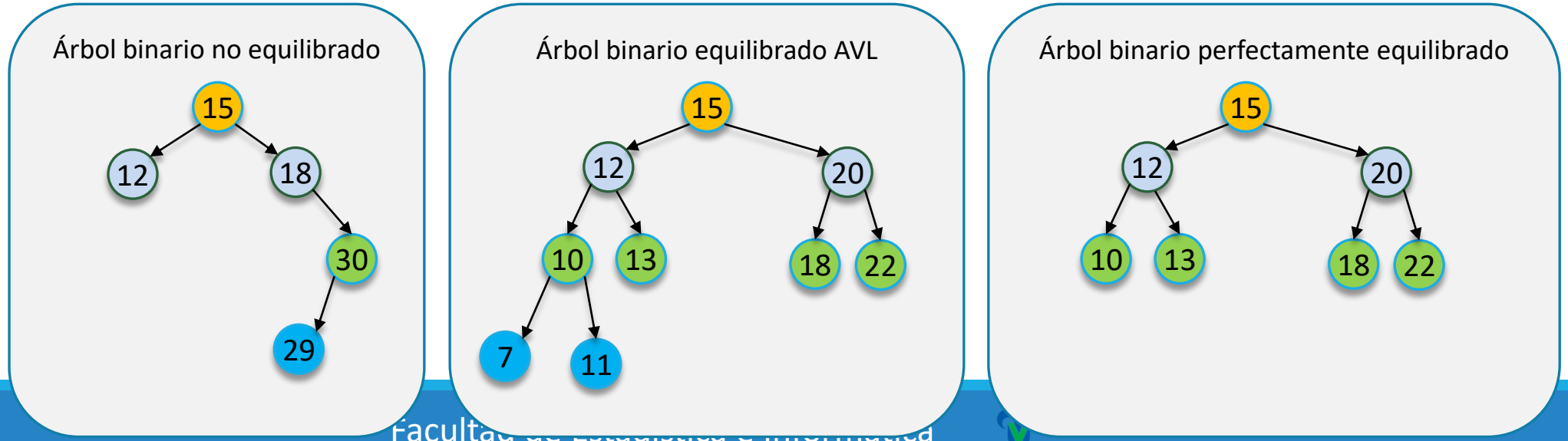
Árboles binarios equilibrados

Cuando un árbol binario de búsqueda crece descontroladamente hacia un extremo su rendimiento puede disminuir considerablemente. Para mantener la eficiencia de operación surgen los árboles equilibrados o balanceados. Estos pueden realizar acomodos o balanceos después de inserciones o eliminaciones de elementos.



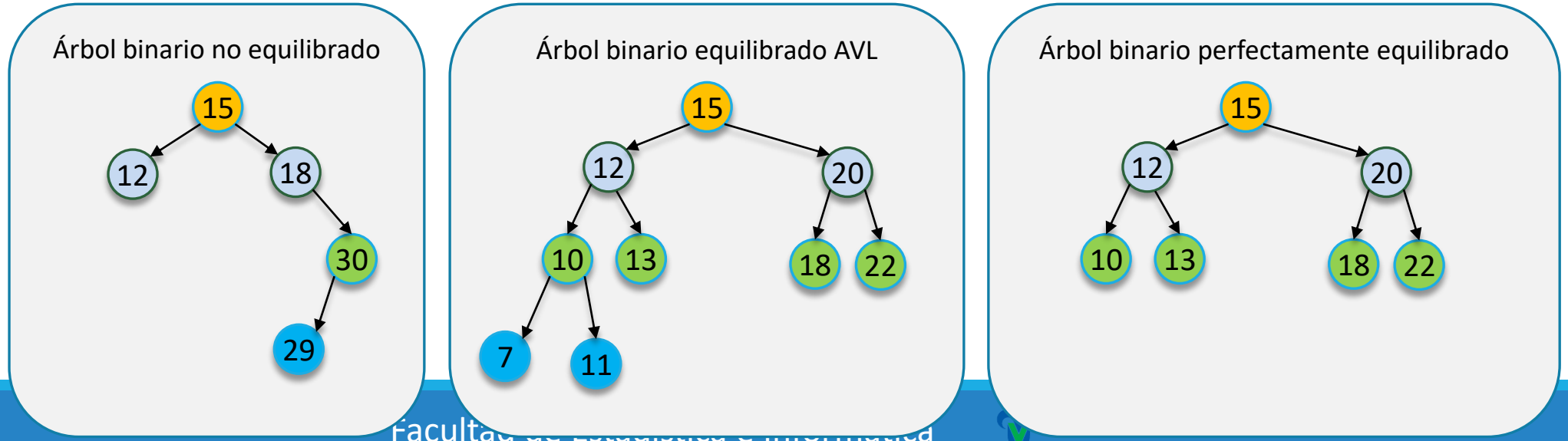
Arboles binarios perfectamente equilibrados

Un árbol perfectamente equilibrado es un árbol binario en el que, para todo nodo, el número de nodos en el subárbol izquierdo y el número de nodos en el subárbol derecho difieren como mucho en una unidad.



Arboles binarios equilibrados AVL

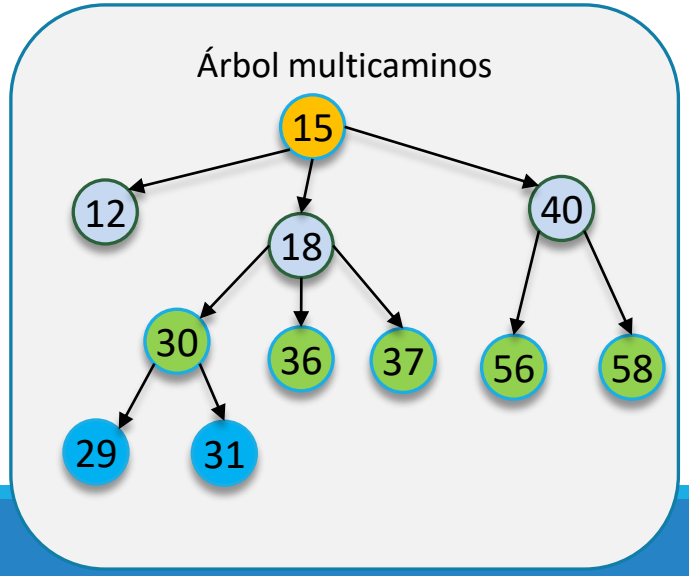
Un árbol equilibrado en sentido AVL (Adelson-Velskii y Landis, 1962) es un árbol binario en el que la diferencia de alturas de los subárboles izquierdo y derecho correspondientes a cualquier nodo del árbol no es superior a uno.



Arboles multicaminos

Un árbol multicaminos es una estructura de datos homogénea, dinámica y no lineal, en donde a cada nodo le pueden seguir una cantidad n de nodos hijos...

Mientras que los árboles binarios fueron pensados para trabajar en memoria principal, los arboles multicaminos fueron diseñados para trabajar con sistemas de archivos (Serrano Montero, 2006).



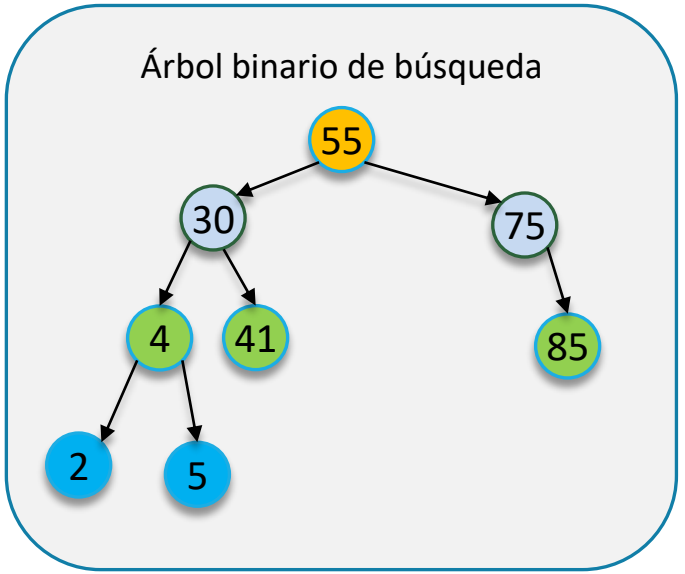
Árboles binarios de búsqueda

Árboles binarios de búsqueda

Un árbol binario que tiene los nodos ordenados de alguna manera se conoce como árbol binario de búsqueda.

Se puede buscar aplicando el criterio de búsqueda binaria similar al utilizado con arreglos.

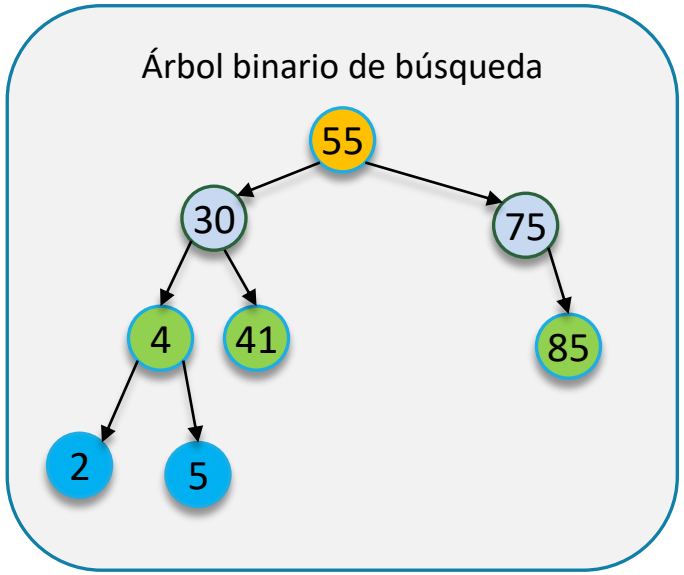
En concreto, un árbol binario de búsqueda es aquel en que, dado un nodo, todos los datos del subárbol izquierdo son menores que los datos de ese nodo, mientras que todos los datos del subárbol derecho son mayores que el nodo.



Árboles binarios de búsqueda

Formalmente se define un **árbol binario de búsqueda** de la siguiente manera (Cairó & Guardati, 2006):

> Para todo nodo *T* del árbol se debe cumplir que todos los valores almacenados en el subárbol izquierdo de *T* sean menores a la información guardada en el nodo *T*. De forma similar, todos los valores almacenados en el subárbol derecho de *T* deben ser mayores a la información guardada en el nodo *T*.



Ventajas de los árboles binarios de búsqueda

Es una estructura de datos sobre la cual se pueden realizar eficientemente las operaciones de búsqueda, inserción y eliminación.

Comparando con otras estructuras de datos, se pueden observar ciertas ventajas:

Ventajas de los árboles binarios de búsqueda

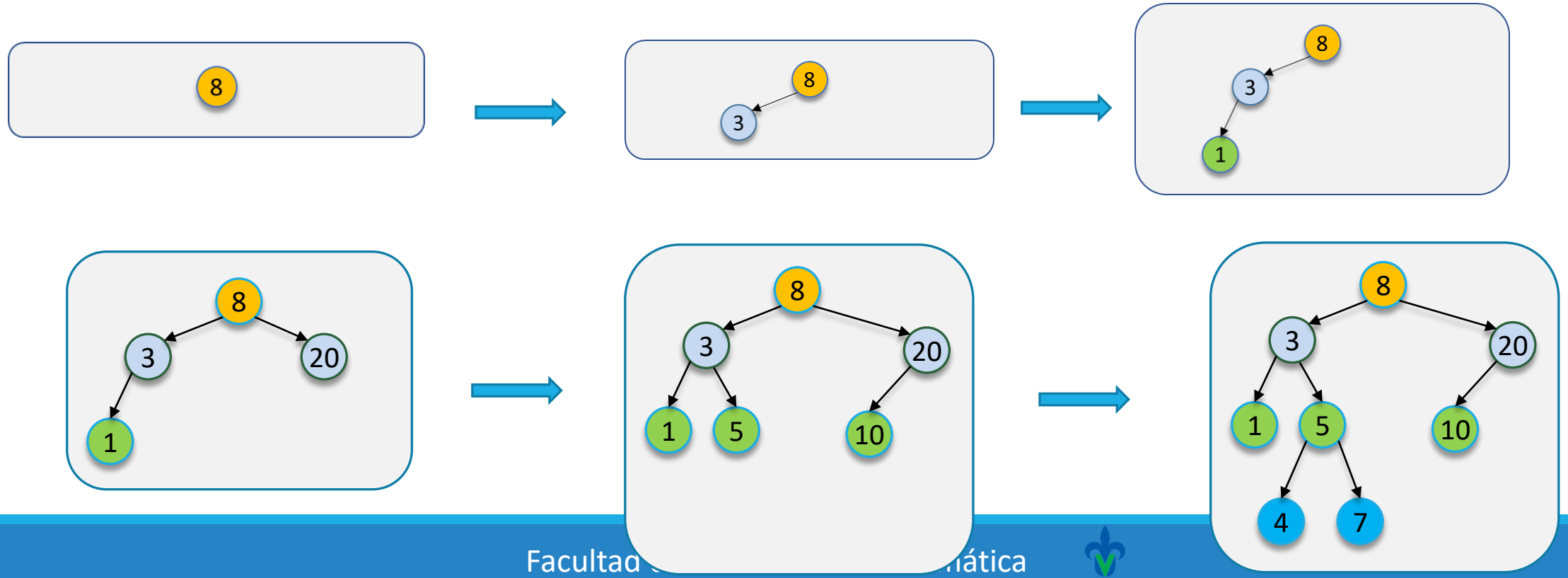
En un arreglo es posible localizar datos eficientemente si estos se encuentran ordenados, pero las operaciones de inserción y eliminación resultan costosas, porque involucran movimiento de los elementos dentro del arreglo.

En las listas, por otra parte, dichas operaciones se pueden llevar a cabo con facilidad, pero la operación de búsqueda, en este caso, es una operación que demanda recursos, pudiendo inclusive requerir recorrer todos los elementos de ella para llegar a uno en particular.

Creación de un árbol binario de búsqueda

- Graficar un árbol binario con los siguientes datos:

8, 3, 1, 20, 5, 10, 7, 4



Creación de un árbol binario de búsqueda

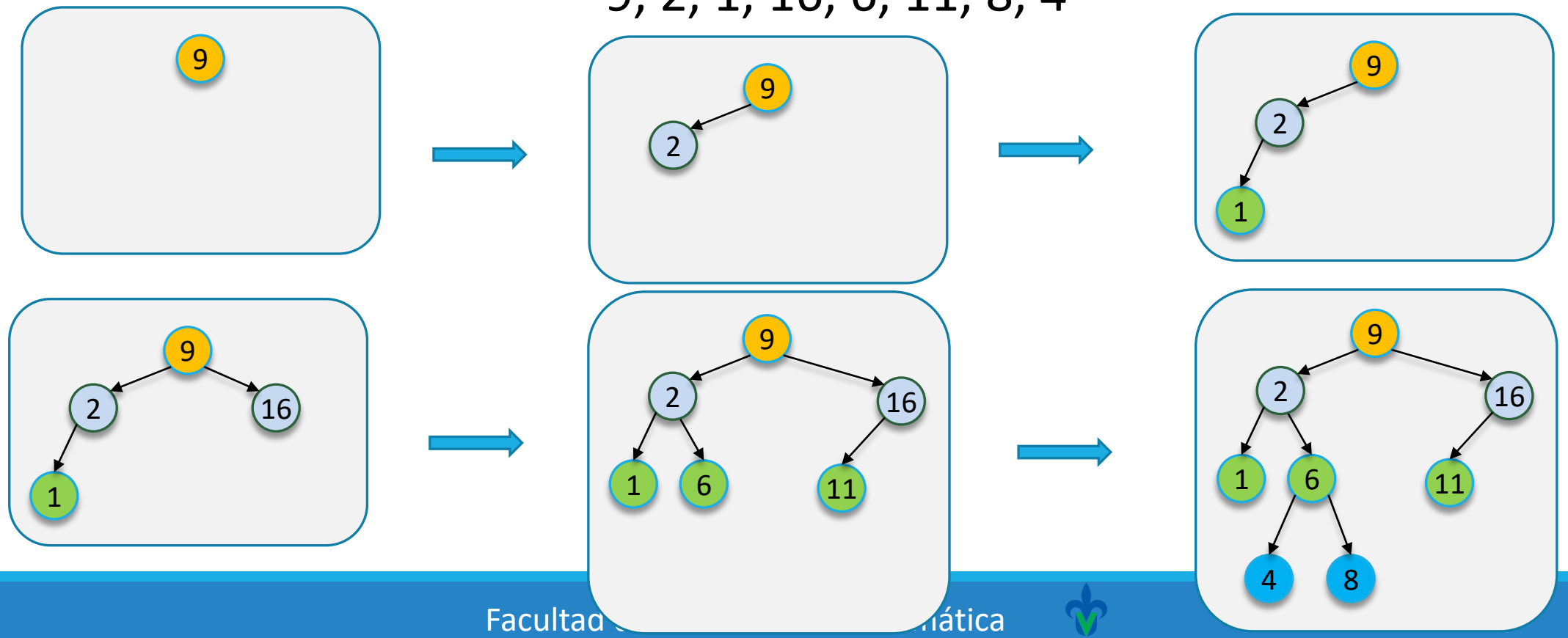
- Graficar un árbol binario con los siguientes datos:

9, 2, 1, 16, 6, 11, 8, 4

Creación de un árbol binario de búsqueda

- Graficar un árbol binario con los siguientes datos:

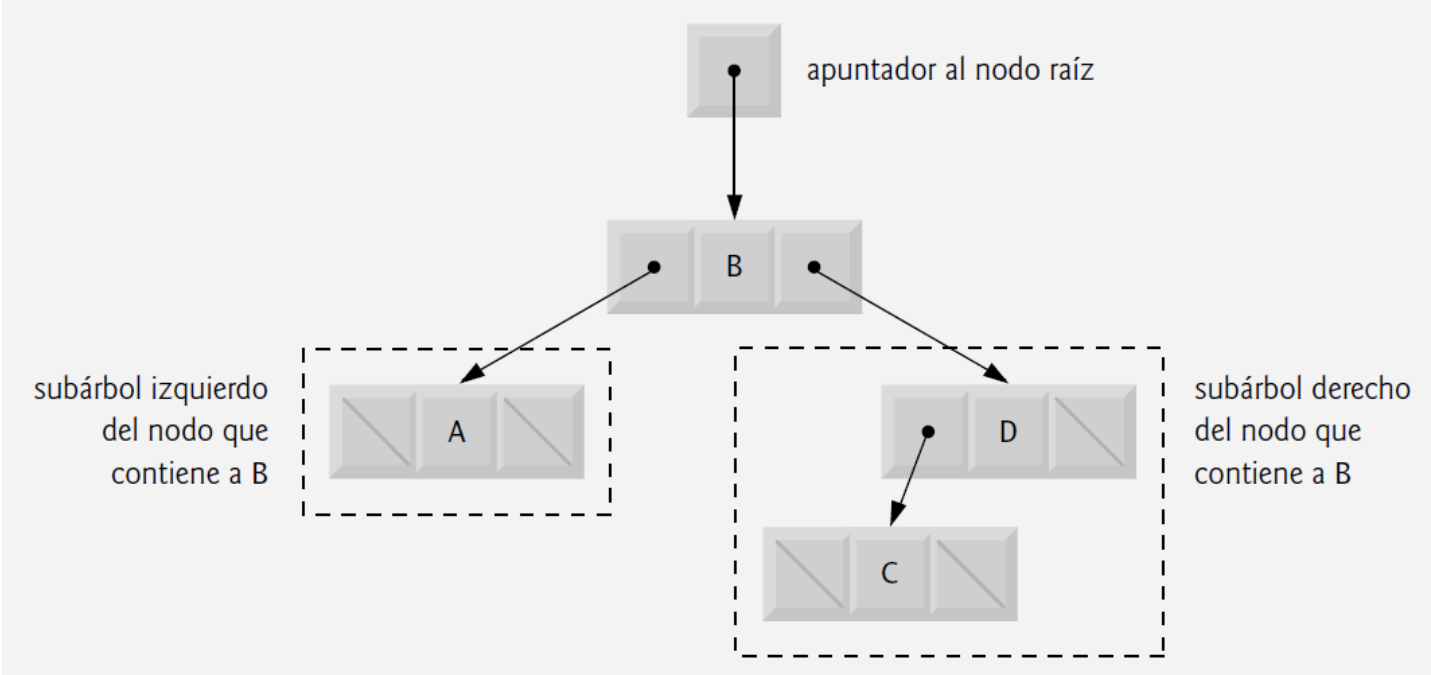
9, 2, 1, 16, 6, 11, 8, 4



Árboles binarios de búsqueda

Implementación

El **nodo raíz** (nodo B) es el primer nodo en un árbol. Cada enlace en el nodo raíz hace referencia a un **hijo** (nodos A y D). El **hijo izquierdo** (nodo A) es el nodo raíz del **subárbol izquierdo** (que sólo contiene el nodo A), y el **hijo derecho** (nodo D) es el nodo raíz del **subárbol derecho** (que contiene los nodos D y C).



Árboles binarios de búsqueda

Implementación

esVacio()

Constructor

Destructor

[1] Insertar elemento

[2] Mostrar árbol completo acostado con la raíz a la izquierda

[3] Graficar árbol completo

[4] Buscar un elemento en el árbol

[5] Recorrer el árbol en PreOrden

[6] Recorrer el árbol en InOrden

[7] Recorrer el árbol en PostOrden

[8] Eliminar un nodo del árbol PREDECESOR

[9] Eliminar un nodo del árbol SUCESOR

Árboles binarios de búsqueda

[10] Recorrer el árbol por niveles (Amplitud)

[11] Altura del árbol

[12] Cantidad de hojas del árbol

[13] Cantidad de nodos del árbol

[15] Revisa si es un árbol binario completo

[16] Revisa si es un árbol binario lleno

[17] Eliminar el árbol

Árboles binarios de búsqueda. **Implementación**

- La clase Nodo

- Crea un Nodo formado por los campos:
  - **dato** (de tipo entero) que corresponde al valor que contiene el nodo.
  - **Nodo** *izquierdoPtr
  - **Nodo** *derechoPtr

```
class Nodo
{

public:
    int dato;    // INFO, va
    Nodo *izquierdoPtr; //
    Nodo *derechoPtr;   //
};
```

Árboles binarios de búsqueda. **Implementación**

- La clase Árbol
- Crea un Nodo raíz del árbol.

```
class Arbol
{

private:
    Nodo *raizPtr;
```

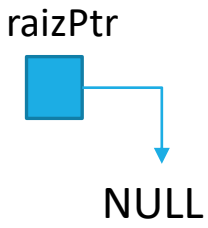
raizPtr

```
// definicion del arbol de nodos
class Arbol
{

private:
    Nodo *raizPtr;  // P, apuntador al inicio del arbol
    bool esVacio();
    void gotoxy(int x,int y);

public:
    Arbol();                                    // constructor
    ~Arbol();                                   // destructor
    Nodo *regresaRaiz();
    void insertarNodo(int valor, Nodo *&nodoPtr);
    void muestraAcostado(int nivel, Nodo *nodoPtr);
    void graficarArbol(Nodo *nodoPtr, int x, int y, int n);
    bool busqueda(int x, Nodo *nodoPtr);
    Nodo *buscaMenor(Nodo *nodoPtr);
    Nodo *buscaMayor(Nodo* nodoPtr);
    void eliminarPredecesor(int x, Nodo *&nodoPtr);
    void eliminarSucesor(int x, Nodo *&nodoPtr);
    void preOrden(Nodo *nodoPtr);
    void inOrden(Nodo *nodoPtr);
    void postOrden(Nodo *nodoPtr);

    void recorridoNiveles(Nodo *nodoPtr);
    int altura(Nodo *nodoPtr);
    int contarHojas(Nodo *nodoPtr);
    int contarNodos(Nodo *nodoPtr);
    void arbolEspejo(Nodo *nodoPtr, Nodo *&nodoEspejoPtr);
    bool esCompleto(Nodo *nodoPtr);
    bool esLleno(Nodo *nodoPtr);
    void podarArbol(Nodo *&nodoPtr);
};
```

Árboles binarios de búsqueda. **Implementación**

- Método Constructor Arbol::Arbol()
- Inicializa la raíz del Árbol con valor Nulo.

```
// constructor predeterminado
Arbol::Arbol()
{
    raizPtr = NULL;
}
```



Árboles binarios de búsqueda. **Implementación**

- Método Destructor Arbol::~Arbol()

- Libera la memoria ocupada por los nodos del árbol.
- Emplea el método Arbol::podarArbol(Nodo *&nodoPtr)

```
// destructor predeterminado
Arbol::~Arbol()
{
    podarArbol(raizPtr);
}
```

Árboles binarios de búsqueda. **Implementación**

- Método Arbol::esVacio()

- Verifica si la raíz del árbol apunta a vacío.

```
bool Arbol::esVacio()
{
    return raizPtr == NULL;
}
```

Árboles binarios de búsqueda. **Implementación**

- Método Nodo *Arbol::regresaRaiz()
- Regresa el nodo de la raíz del árbol.

```
// regresa el apuntador a la raíz
Nodo *Arbol::regresaRaiz()
{
    return raizPtr;
}
```

Árboles binarios de búsqueda. **Implementación**

- Método void Arbol::insertarNodo(int valor, Nodo *&nodoPtr)
- Crea un nuevo nodo y lo inserta en el lugar correspondiente dentro del árbol.

```
void Arbol::insertarNodo(int valor, Nodo *&nodoPtr)
{
    if( nodoPtr == NULL )   // buscamos una hoja
    {
        nodoPtr = new Nodo();
        nodoPtr->dato = valor;
        nodoPtr->izquierdoPtr = NULL;
        nodoPtr->derechoPtr = NULL;
    }
    else if( valor < nodoPtr->dato )
        insertarNodo( valor, nodoPtr->izquierdoPtr );
    else if( valor > nodoPtr->dato )
        insertarNodo( valor, nodoPtr->derechoPtr );
}
```

Árboles binarios de búsqueda. **Implementación**

- Método void Arbol::insertarNodo(int valor, Nodo *&nodoPtr)
- **Valor**=4

```
void Arbol::insertarNodo(int valor, Nodo *&nodoPtr)
{
    if( nodoPtr == NULL )    // buscamos una hoja
    {
        nodoPtr = new Nodo();
        nodoPtr->dato = valor;
        nodoPtr->izquierdoPtr = NULL;
        nodoPtr->derechoPtr = NULL;
    }
    else if( valor < nodoPtr->dato )
        insertarNodo( valor, nodoPtr->izquierdoPtr );
    else if( valor > nodoPtr->dato )
        insertarNodo( valor, nodoPtr->derechoPtr );
}
```

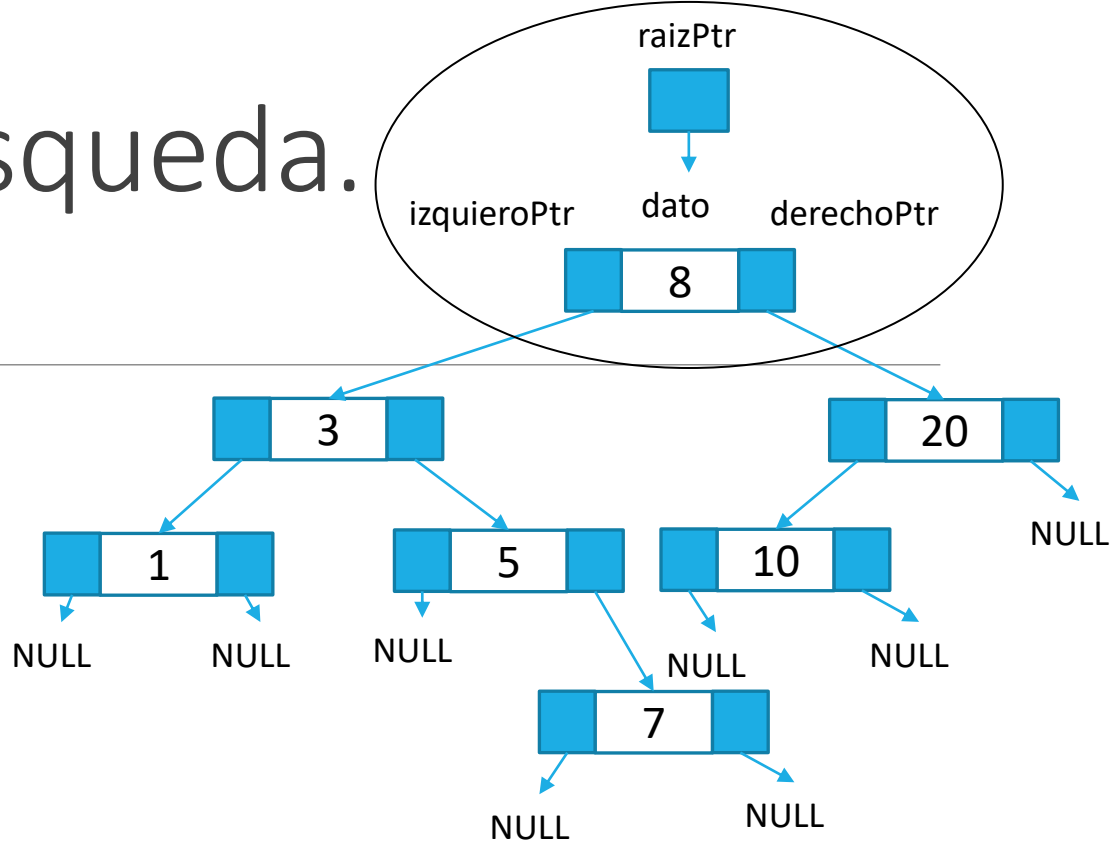
Llamada a Arbol::insertarNodo en el menú principal, donde podemos observar los parámetros que se le envían:

```
Arbol arbolEnteros; // crea el objeto Arbol
Nodo *raizArbolPtr = arbolEnteros.regresaRaiz();

case 1:
    cout << "\nIngrese valor entero: ";
    cin >> valor;

    arbolEnteros.insertarNodo( valor, raizArbolPtr );
```

Árboles binarios de búsqueda. **Implementación**

- Método void Arbol::insertarNodo(int valor, Nodo *&nodoPtr)
- **Valor**=4

```
void Arbol::insertarNodo(int valor, Nodo *&nodoPtr)
{
    if( nodoPtr == NULL )    // buscamos una hoja
    {
        nodoPtr = new Nodo();
        nodoPtr->dato = valor;
        nodoPtr->izquierdoPtr = NULL;
        nodoPtr->derechoPtr = NULL;
    }
    else if( valor < nodoPtr->dato )
        insertarNodo( valor, nodoPtr->izquierdoPtr );
    else if( valor > nodoPtr->dato )
        insertarNodo( valor, nodoPtr->derechoPtr );
}
```

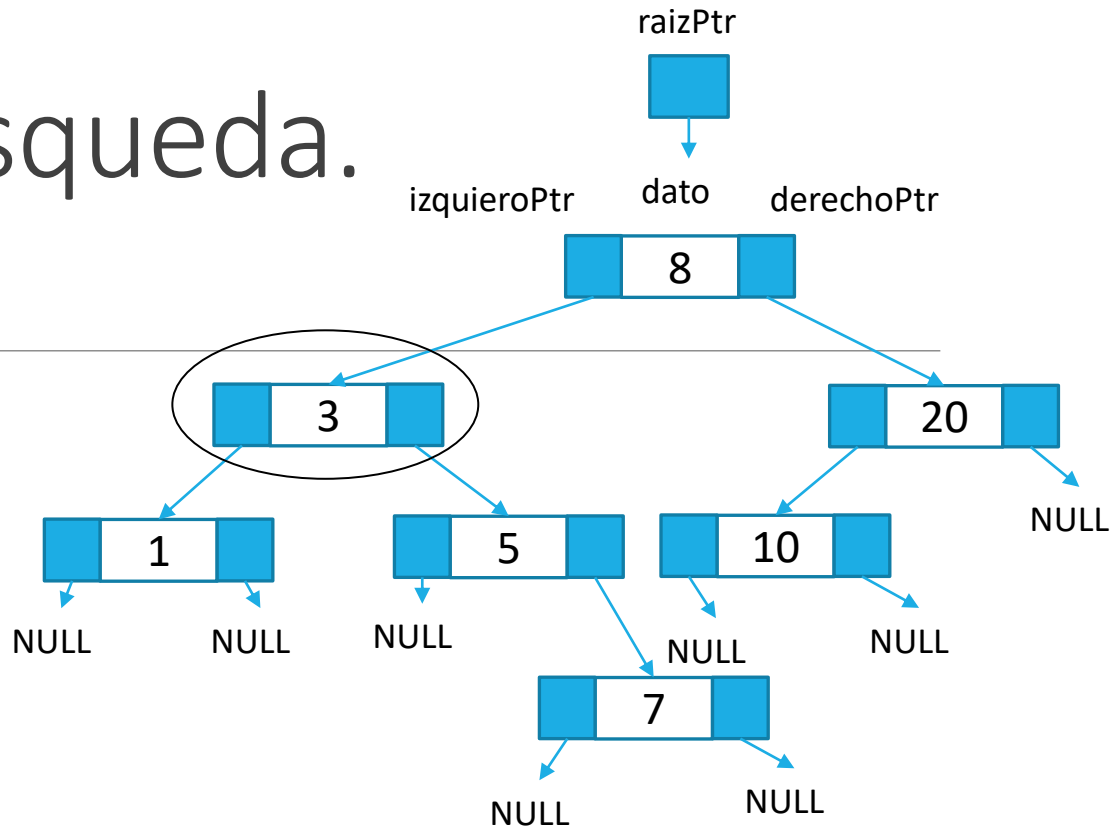


Primera iteración de la recursión:
nodoPtr = raizPtr (nodoPtr no es nulo)
4 < 8
insertarNodo (4, nodoPtr→izquierdoPtr)
Se llama nuevamente el método insertar Nodo con el valor 4 a insertar y la dirección del nodo que contiene el número 3 (a la izquierda del 8).

Árboles binarios de búsqueda. **Implementación**

- Método void Arbol::insertarNodo(int valor, Nodo *&nodoPtr)
- **Valor**=4

```
void Arbol::insertarNodo(int valor, Nodo *&nodoPtr)
{
    if( nodoPtr == NULL )    // buscamos una hoja
    {
        nodoPtr = new Nodo();
        nodoPtr->dato = valor;
        nodoPtr->izquierdoPtr = NULL;
        nodoPtr->derechoPtr = NULL;
    }
    else if( valor < nodoPtr->dato )
        insertarNodo( valor, nodoPtr->izquierdoPtr );
    else if( valor > nodoPtr->dato )
        insertarNodo( valor, nodoPtr->derechoPtr );
}
```

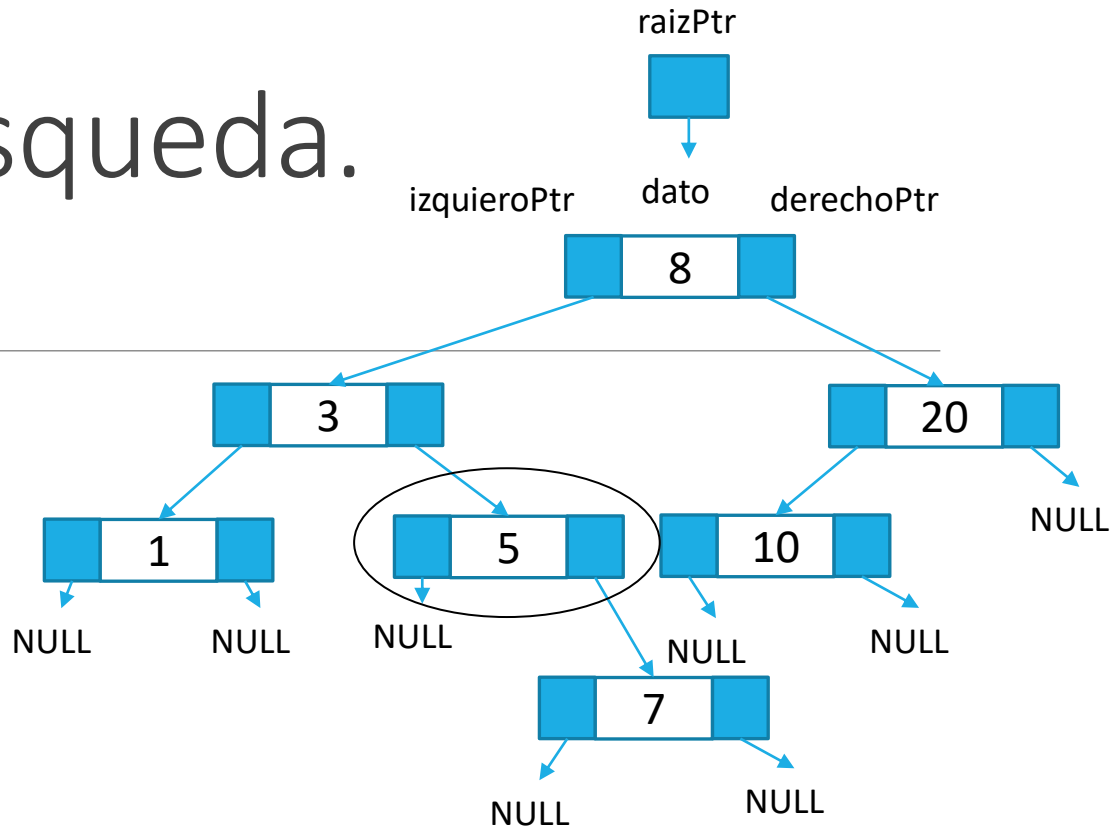


Segunda iteración de la recursión:
nodoPtr = nodo con el dato 3 (nodoPtr no es nulo)
4 > 3
insertarNodo (4, nodoPtr→derechoPtr)
Se llama nuevamente el método insertar Nodo con el valor 4 a insertar y la dirección del nodo que contiene el número 5 (a la derecha del 3).

Árboles binarios de búsqueda. **Implementación**

- Método void Arbol::insertarNodo(int valor, Nodo *&nodoPtr)
- **Valor**=4

```
void Arbol::insertarNodo(int valor, Nodo *&nodoPtr)
{
    if( nodoPtr == NULL )    // buscamos una hoja
    {
        nodoPtr = new Nodo();
        nodoPtr->dato = valor;
        nodoPtr->izquierdoPtr = NULL;
        nodoPtr->derechoPtr = NULL;
    }
    else if( valor < nodoPtr->dato )
        insertarNodo( valor, nodoPtr->izquierdoPtr );
    else if( valor > nodoPtr->dato )
        insertarNodo( valor, nodoPtr->derechoPtr );
}
```

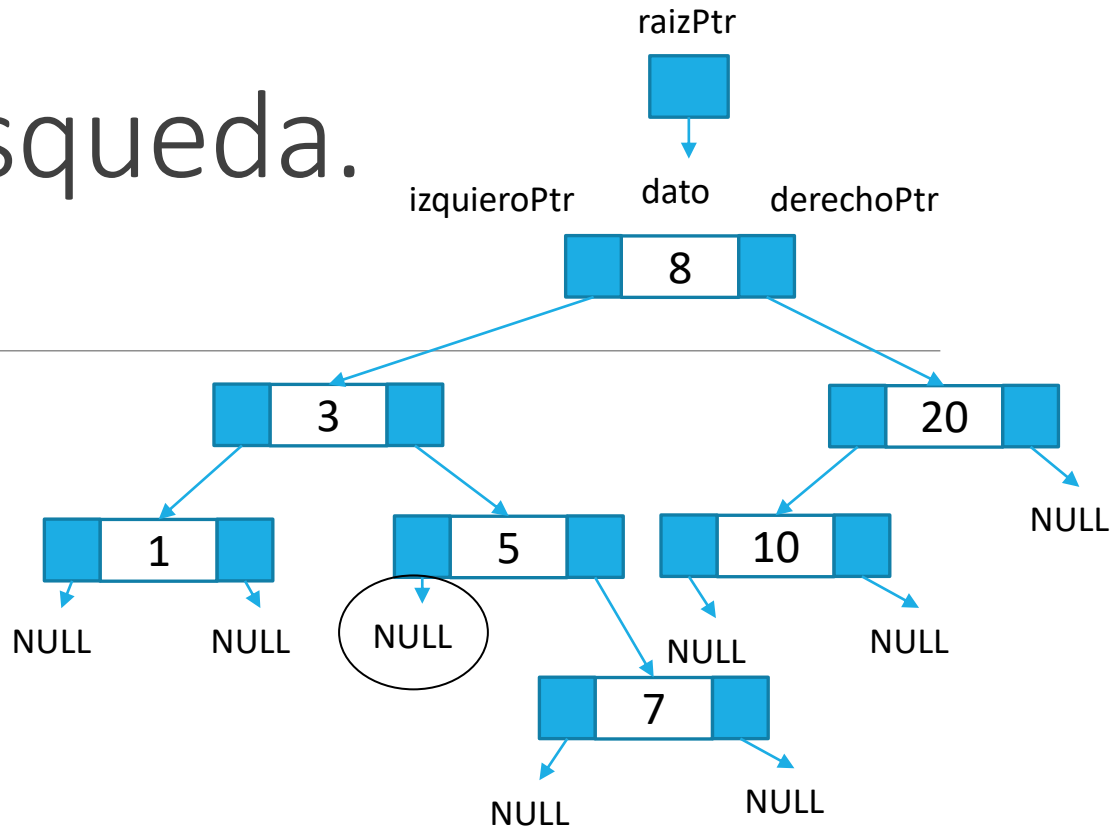


Tercera iteración de la recursión:
nodoPtr = nodo con el dato 5 (nodoPtr no es nulo)
4 < 5
insertarNodo (4, nodoPtr→izquierdoPtr)
Se llama nuevamente el método insertar Nodo con el valor 4 a insertar y la dirección del nodo a la izquierda de 5, es decir NULL.

Árboles binarios de búsqueda. **Implementación**

- Método void Arbol::insertarNodo(int valor, Nodo *&nodoPtr)
- **Valor**=4

```
void Arbol::insertarNodo(int valor, Nodo *&nodoPtr)
{
    if( nodoPtr == NULL )    // buscamos una hoja
    {
        nodoPtr = new Nodo();
        nodoPtr->dato = valor;
        nodoPtr->izquierdoPtr = NULL;
        nodoPtr->derechoPtr = NULL;
    }
    else if( valor < nodoPtr->dato )
        insertarNodo( valor, nodoPtr->izquierdoPtr );
    else if( valor > nodoPtr->dato )
        insertarNodo( valor, nodoPtr->derechoPtr );
}
```

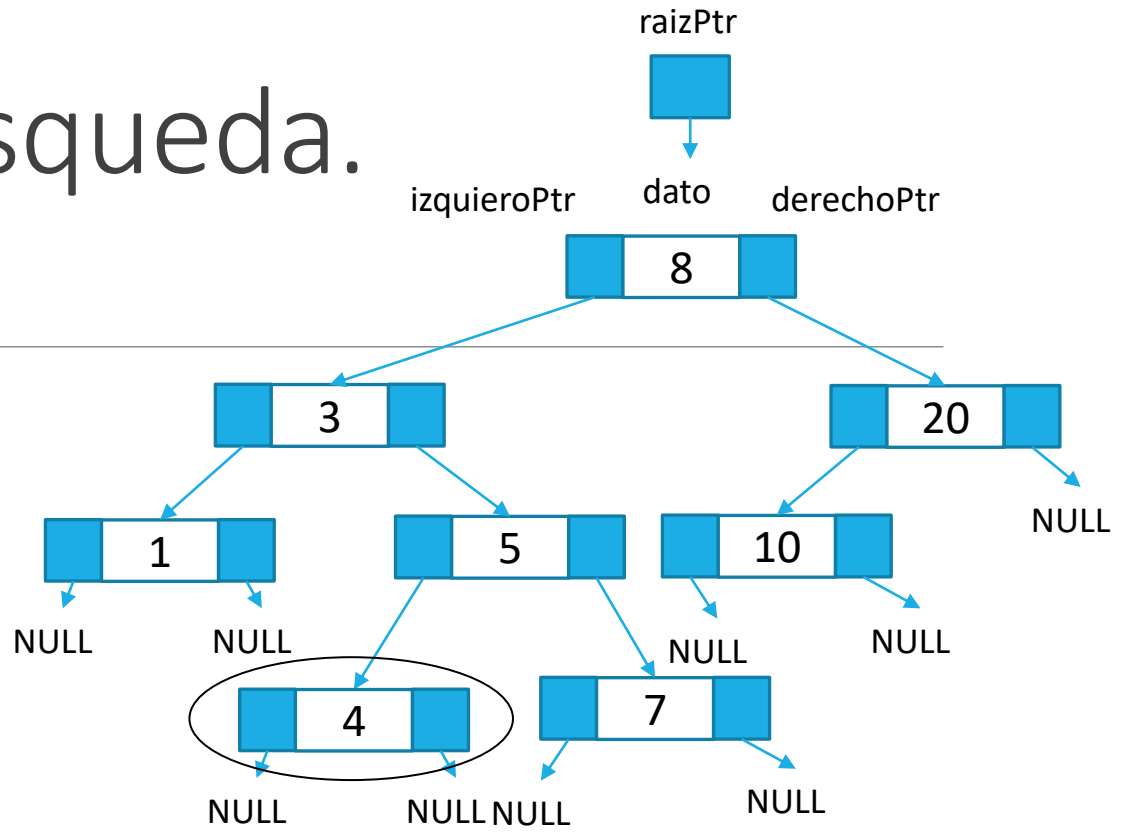


Cuarta iteración de la recursión:
nodoPtr = NULL
Por lo tanto ingresa al if (nodoPtr==NULL) y crea un nuevo nodoPtr:

Árboles binarios de búsqueda. **Implementación**

- Método void Arbol::insertarNodo(int valor, Nodo *&nodoPtr)
- **Valor**=4

```
void Arbol::insertarNodo(int valor, Nodo *&nodoPtr)
{
    if( nodoPtr == NULL )    // buscamos una hoja
    {
        nodoPtr = new Nodo();
        nodoPtr->dato = valor;
        nodoPtr->izquierdoPtr = NULL;
        nodoPtr->derechoPtr = NULL;
    }
    else if( valor < nodoPtr->dato )
        insertarNodo( valor, nodoPtr->izquierdoPtr );
    else if( valor > nodoPtr->dato )
        insertarNodo( valor, nodoPtr->derechoPtr );
}
```

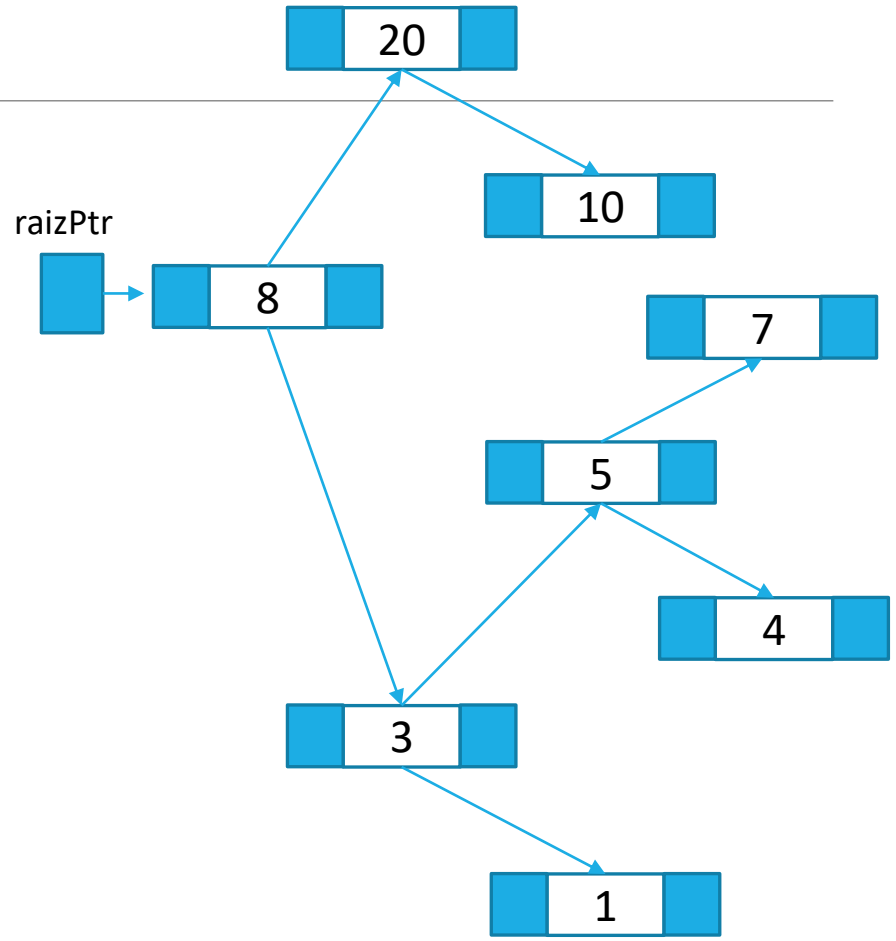


Cuarta iteración de la recursión:
nodoPtr contiene el nuevo nodo creado a la izquierda del nodo con el dato 5.

Árboles binarios de búsqueda. **Implementación**

- Método Arbol::muestraAcostado(int nivel, Nodo *nodoPtr)

- Escribe en pantalla el árbol en forma horizontal, con la raíz a la izquierda.



Árboles binarios de búsqueda. **Implementación**

- Método Arbol::muestraAcostado(int nivel, Nodo *nodoPtr)

- Escribe en pantalla el árbol en forma horizontal, con la raíz a la izquierda.

```
void Arbol::muestraAcostado(int nivel, Nodo *nodoPtr)
{
    if( nodoPtr == NULL )
        return;

    muestraAcostado( nivel+1, nodoPtr->derechoPtr );

    for(int i=0; i<nivel; i++)
        cout<<"   ";

    cout<< nodoPtr->dato <<endl;

    muestraAcostado( nivel+1, nodoPtr->izquierdoPtr );
}
```

Árboles binarios de búsqueda. **Implementación**

- Método Arbol::muestraAcostado(int nivel, Nodo *nodoPtr)

```
void Arbol::muestraAcostado(int nivel, Nodo *nodoPtr)
{
    if( nodoPtr == NULL )
        return;

    muestraAcostado( nivel+1, nodoPtr->derechoPtr );

    for(int i=0; i<nivel; i++)
        cout<<"    ";

    cout<< nodoPtr->dato <<endl;

    muestraAcostado( nivel+1, nodoPtr->izquierdoPtr );
}
```

Llamada a Arbol::muestraAcostado en el menú principal, donde podemos observar los parámetros que se le envían:

```
case 3:
    if( raizArbolPtr == NULL )
    {
        cout << "\nEl arbol esta vacio.\n\n";
        system("pause");
        break;
    }
    cout << "\nLos elementos del arbol acostado con I
    arbolEnteros.muestraAcostado( 0, raizArbolPtr );
```

Recorrido en InOrden

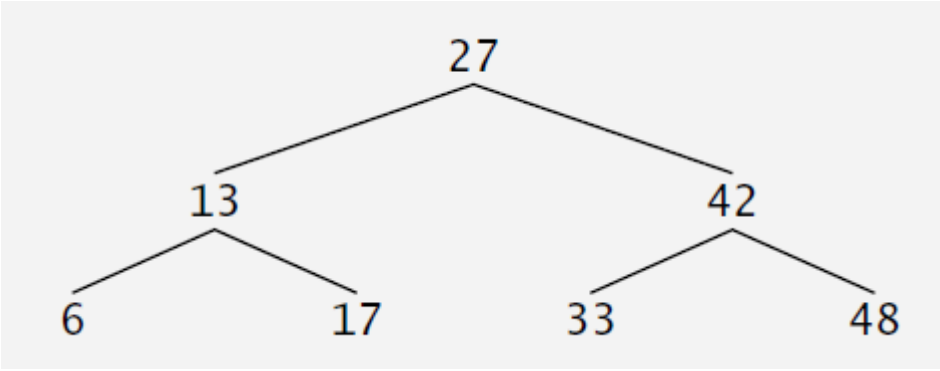
(izquierdo, **raíz**, derecho). El valor en un nodo no se procesa hasta que se procesen los valores en su subárbol izquierdo.

Para recorrer un árbol binario no vacío en InOrden, hay que realizar las siguientes operaciones recursivamente en cada nodo, comenzando con el nodo raíz:

Recorrer el subárbol izquierdo **InOrden**

Procesa la raíz

Recorrer el subárbol derecho **InOrden**



imprime los valores en orden ascendente.

Secuencia: 6 – 13 – 17 – 27 – 33 – 42 - 48

Recorrido en InOrden

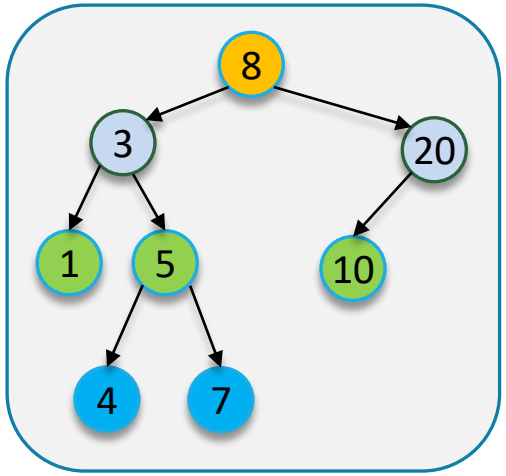
(izquierdo, **raíz**, derecho). El valor en un nodo no se procesa hasta que se procesen los valores en su subárbol izquierdo.

Para recorrer un árbol binario no vacío en InOrden, hay que realizar las siguientes operaciones recursivamente en cada nodo, comenzando con el nodo raíz:

Recorrer el subárbol izquierdo **InOrden**

Procesa la raíz

Recorrer el subárbol derecho **InOrden**



imprime los valores del nodo en orden ascendente.

Secuencia: 1 – 3 – 4 – 5 – 7 – 8 – 10 – 20

Recorrido en InOrden

(izquierdo, **raíz**, derecho)

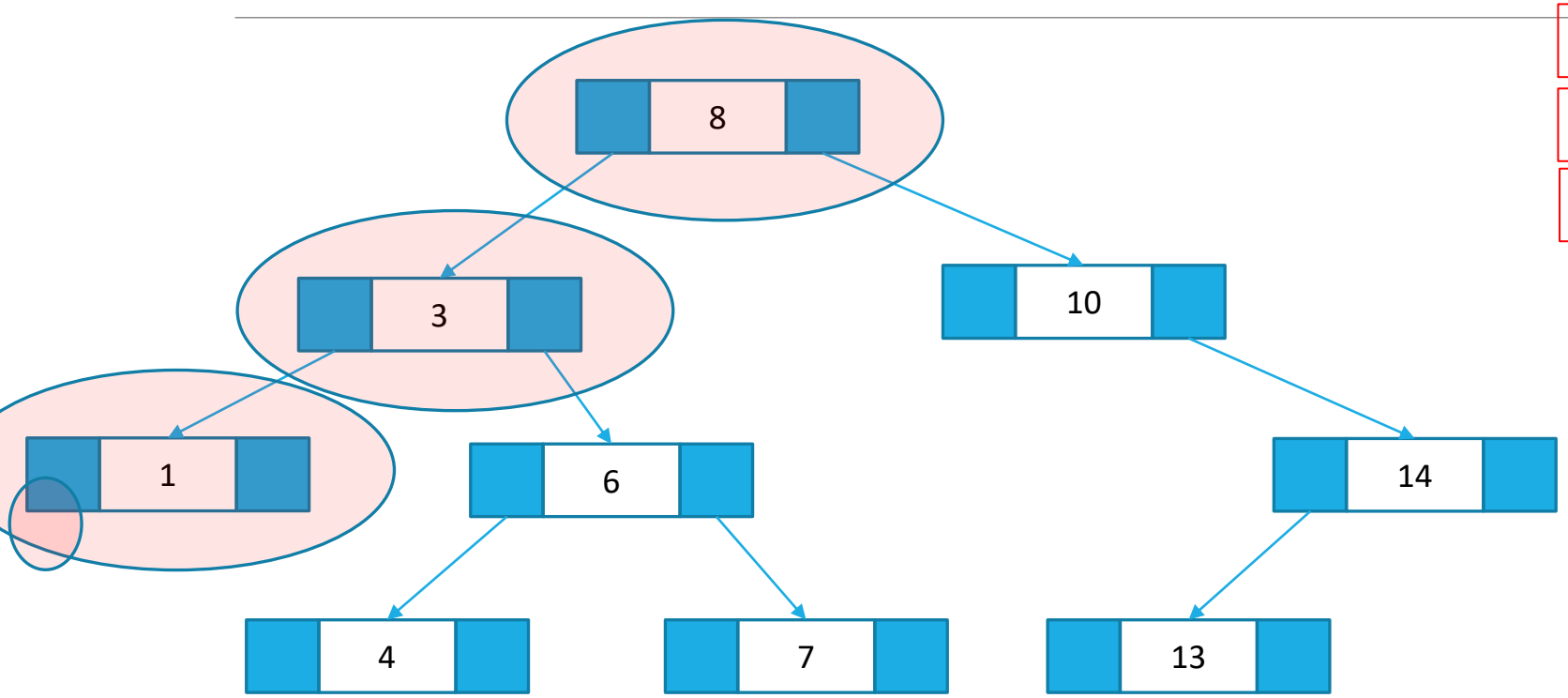
```
void Arbol::inOrden(Nodo *nodoPtr)
{
    if(nodoPtr == NULL)
        return;

    inOrden(nodoPtr->izquierdoPtr);
    cout << nodoPtr->dato << " - ";
    inOrden(nodoPtr->derechoPtr);
}
```

Llamada en main:

```
case 7:
    if( raizArbolPtr == NULL )
    {
        cout << "\nEl arbol esta vacio.\n\n";
        system("pause");
        break;
    }

    cout << "\nRecorrido en InOrden: ";
    arbolEnteros.inOrden(raizArbolPtr);
    cout << "\n\n";
    system("pause");
    break;
```

Secuencia:

Recorrido InOrden



Pendiente: 8

Pendiente: 3

Pendiente: 1

```
void Arbol::inOrden(Nodo *nodoPtr)
{
    if(nodoPtr == NULL)
        return;

    inOrden(nodoPtr->izquierdoPtr);
    cout << nodoPtr->dato << " - ";
    inOrden(nodoPtr->derechoPtr);
}
```

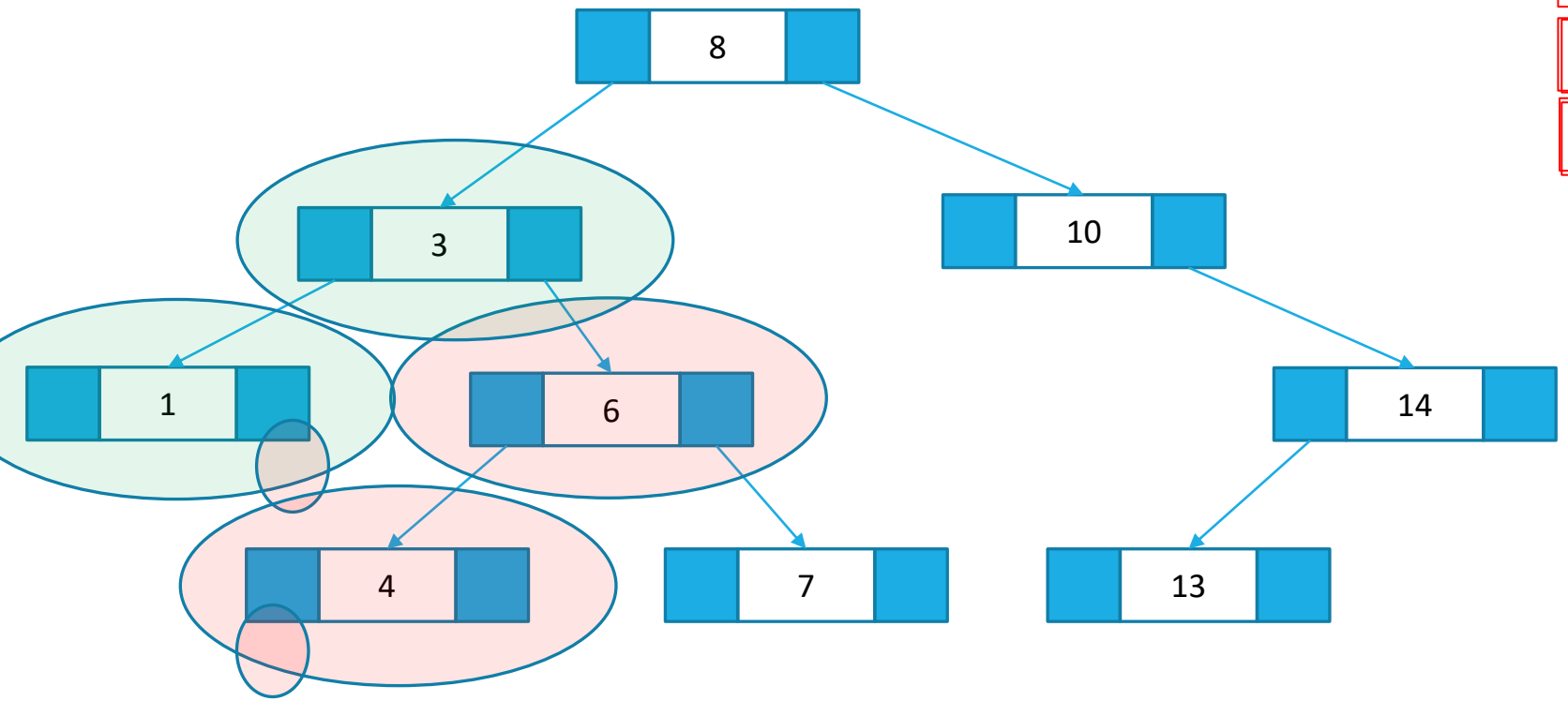
Pendiente

Cada vez que (nodoPtr == NULL)
se re regresa a los nodos pendientes

Recorrido InOrden

Secuencia:

1 3



Pendiente: 8

Pendiente: 3

Pendiente: 1

```
void Arbol::inOrden(Nodo *nodoPtr)
{
    if(nodoPtr == NULL)
        return;

    inOrden(nodoPtr->izquierdoPtr);
    cout << nodoPtr->dato << " - ";
    inOrden(nodoPtr->derechoPtr);
}
```

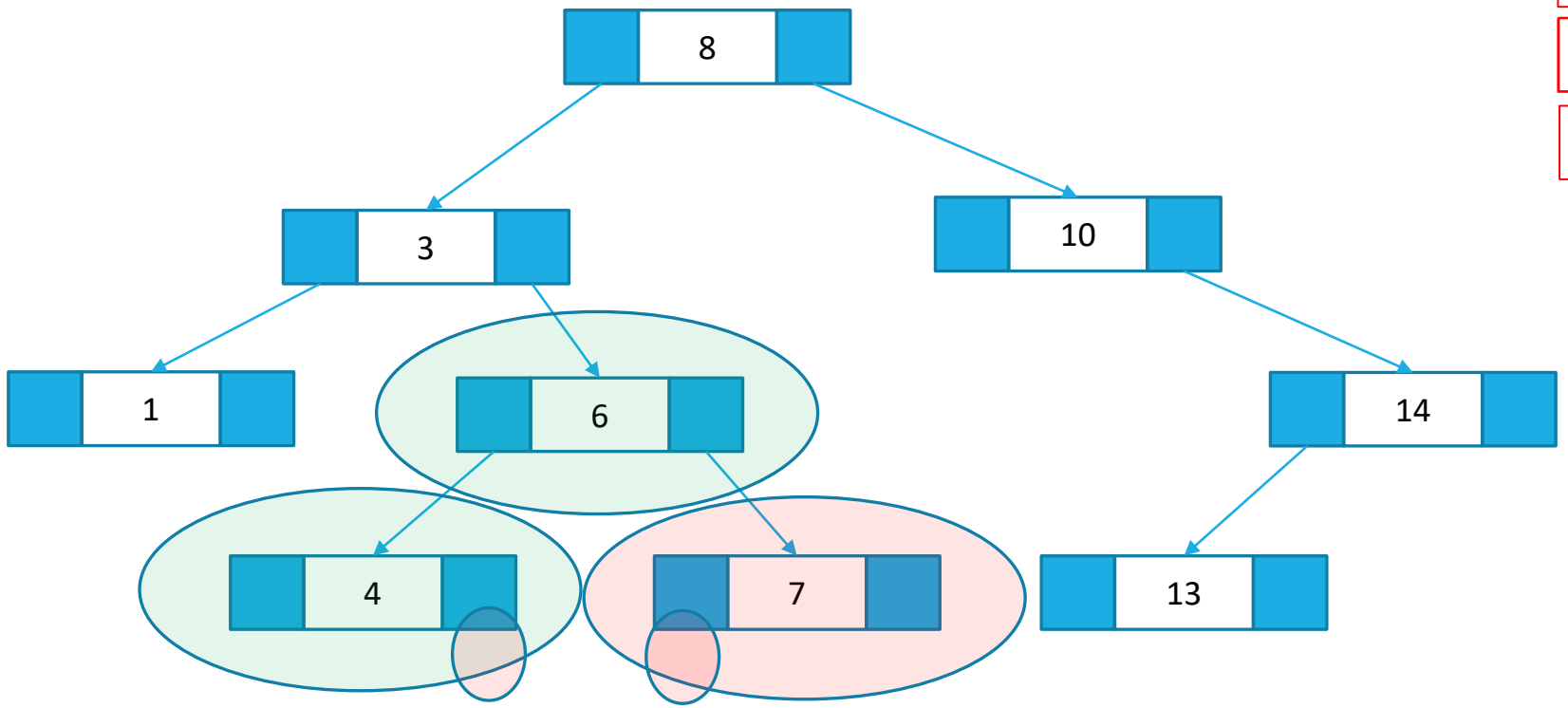
Pendiente

Cada vez que (nodoPtr == NULL)
se re regresa a los nodos pendientes

Recorrido InOrden

Secuencia:

1 3 4 6



Pendiente: 8

Pendiente: 3

Pendiente: 4

```
void Arbol::inOrden(Nodo *nodoPtr)
{
    if(nodoPtr == NULL)
        return;

    inOrden(nodoPtr->izquierdoPtr);
    cout << nodoPtr->dato << " - ";
    inOrden(nodoPtr->derechoPtr);
}
```

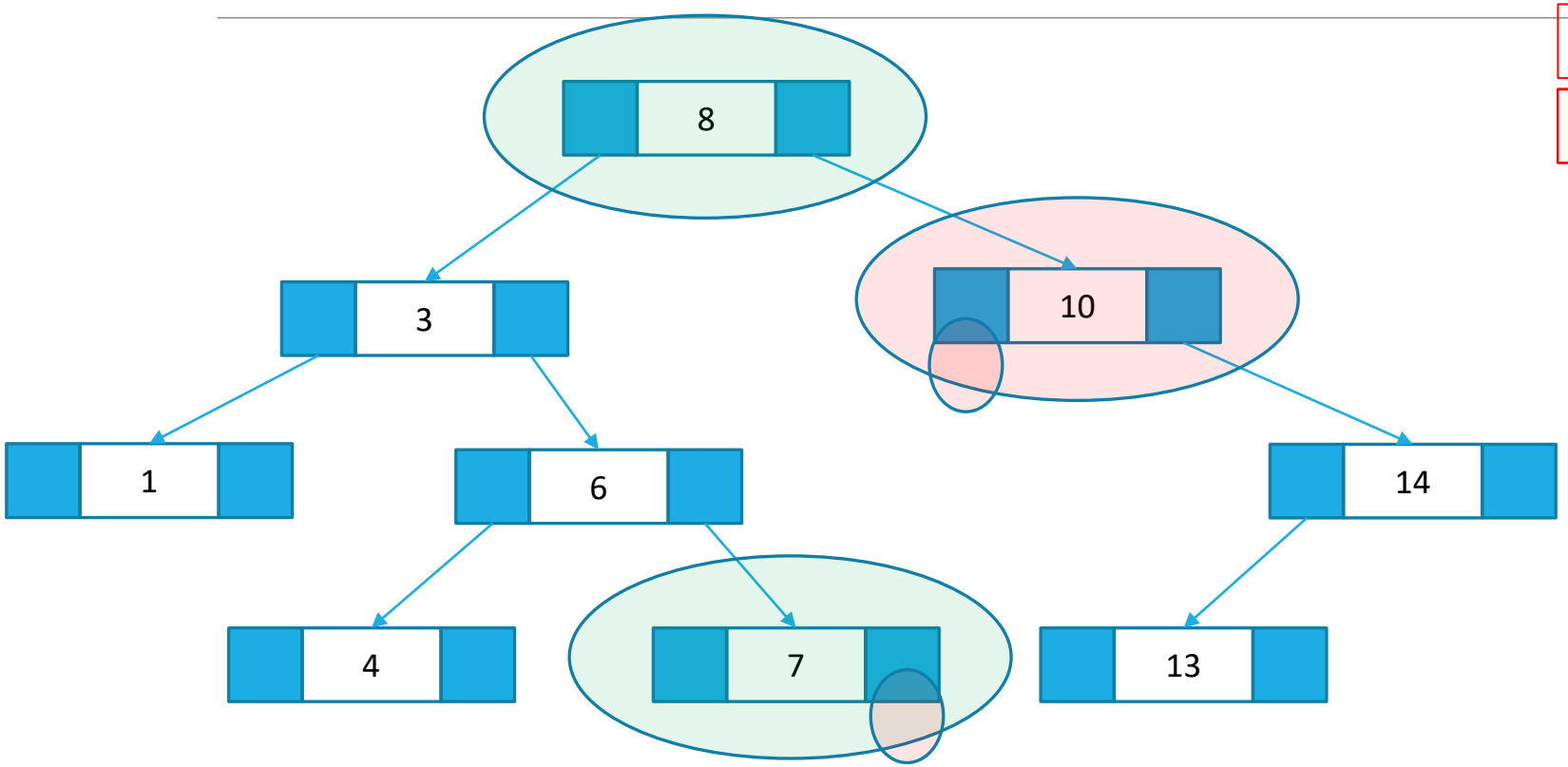
Pendiente

Cada vez que (nodoPtr == NULL)
se re regresa a los nodos pendientes

Recorrido InOrden

Secuencia:

1 3 4 6 7 **8**



Pendiente: 8

Pendiente: 10

```
void Arbol::inOrden(Nodo *nodoPtr)
{
    if(nodoPtr == NULL)
        return;

    inOrden(nodoPtr->izquierdoPtr);
    cout << nodoPtr->dato << " - ";
    inOrden(nodoPtr->derechoPtr);
}
```

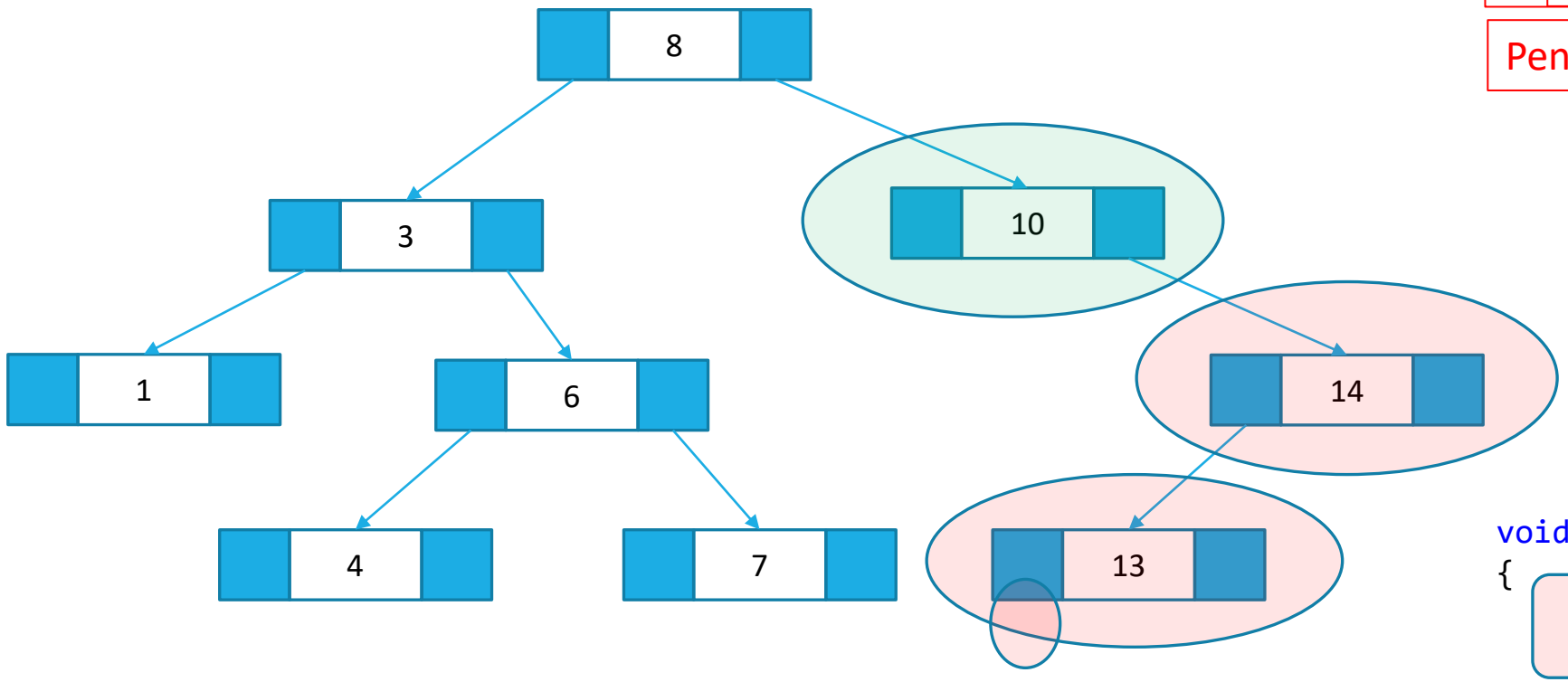
Pendiente

Cada vez que (`nodoPtr == NULL`)
se re regresa a los nodos pendientes

Recorrido InOrden

Secuencia:

1 3 4 6 7 **8** 10



Pendiente: 14

Pendiente: 10

Pendiente: 13

```
void Arbol::inOrden(Nodo *nodoPtr)
{
   if(nodoPtr == NULL)
      return;

   inOrden(nodoPtr->izquierdoPtr);
   cout << nodoPtr->dato << " - ";
   inOrden(nodoPtr->derechoPtr);
}
```

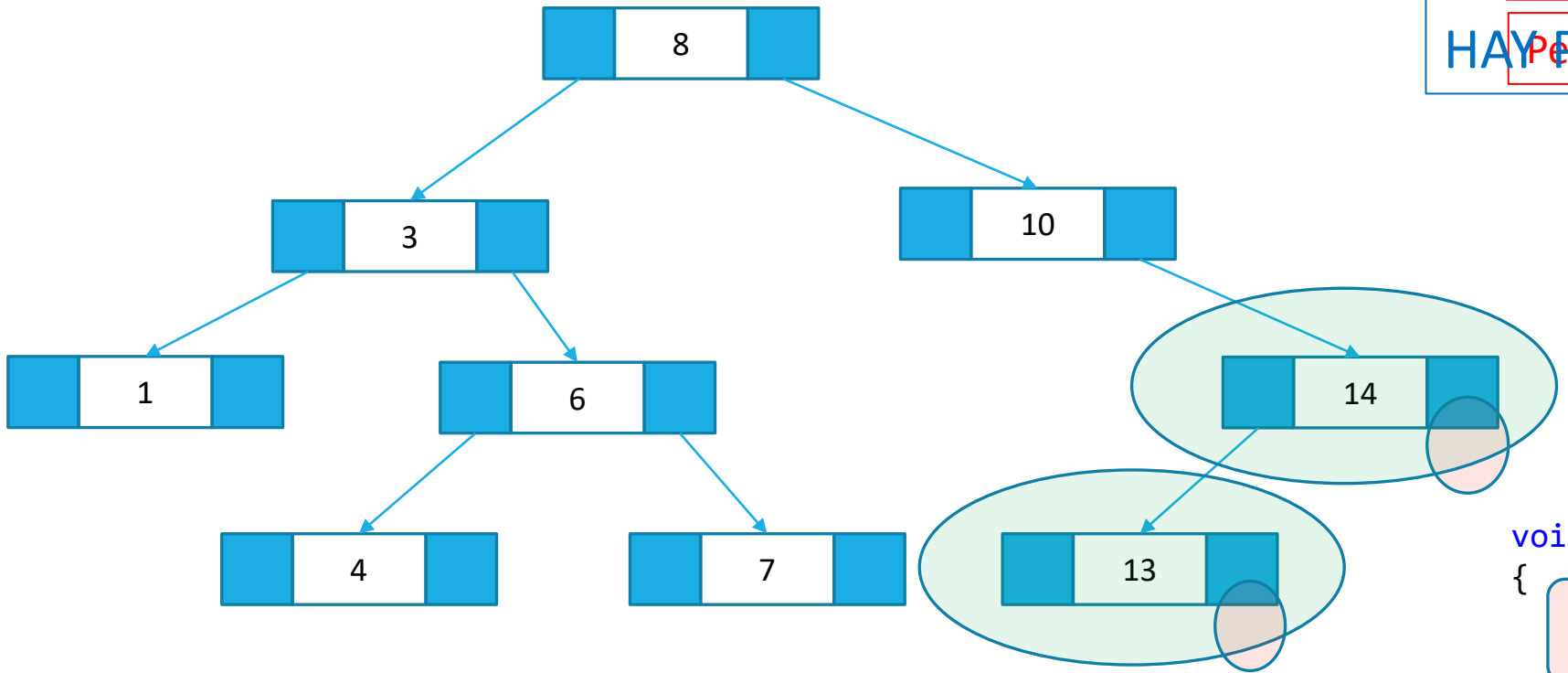
Pendiente

Cada vez que (`nodoPtr == NULL`)
se re regresa a los nodos pendientes

Recorrido InOrden

Secuencia:

1 3 4 6 7 **8** 10 13 14



Pendiente: 14

Pendiente: 13

TERMINA PUES YA NO HAY PENDIENTES

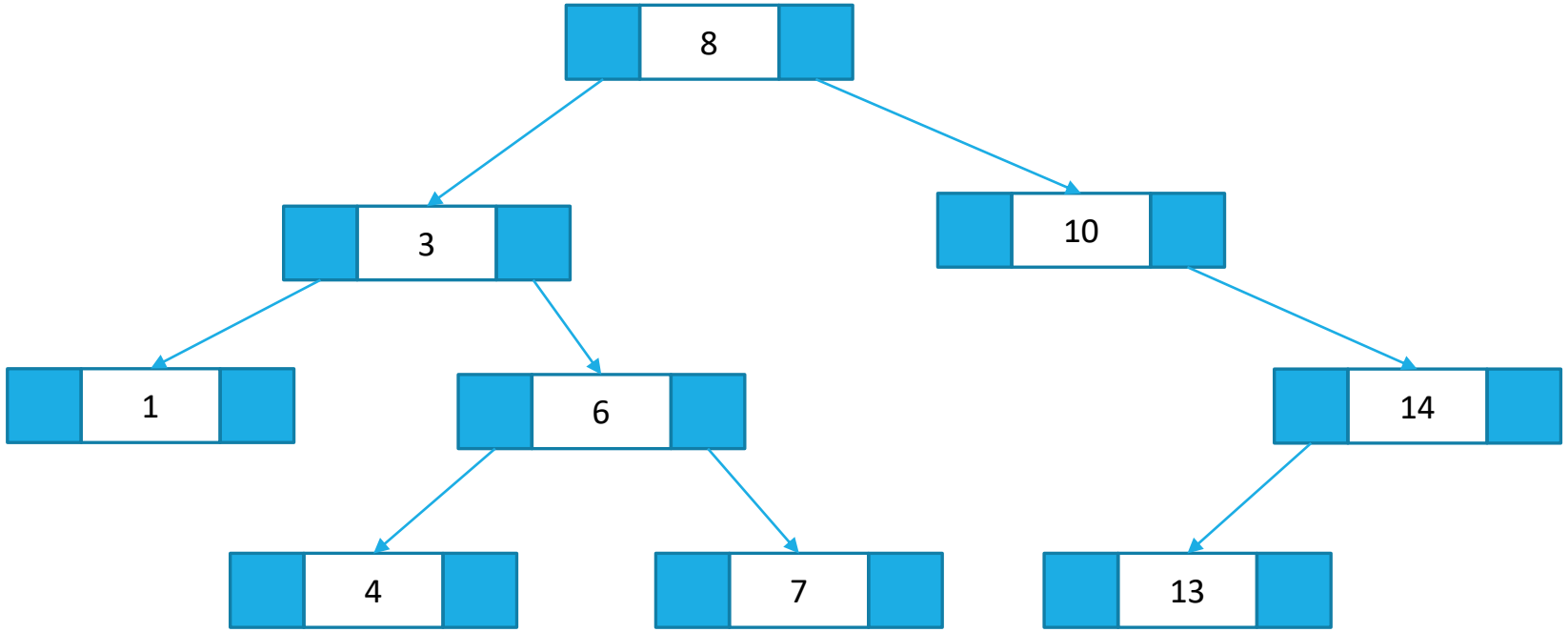
```
void Arbol::inOrden(Nodo *nodoPtr)
{
    if(nodoPtr == NULL)
        return;

    inOrden(nodoPtr->izquierdoPtr);
    cout << nodoPtr->dato << " - ";
    inOrden(nodoPtr->derechoPtr);
}
```

Cada vez que (`nodoPtr == NULL`)
se re regresa a los nodos pendientes

Recorrido InOrden

Secuencia:

1 3 4 6 7 **8** 10 13 14



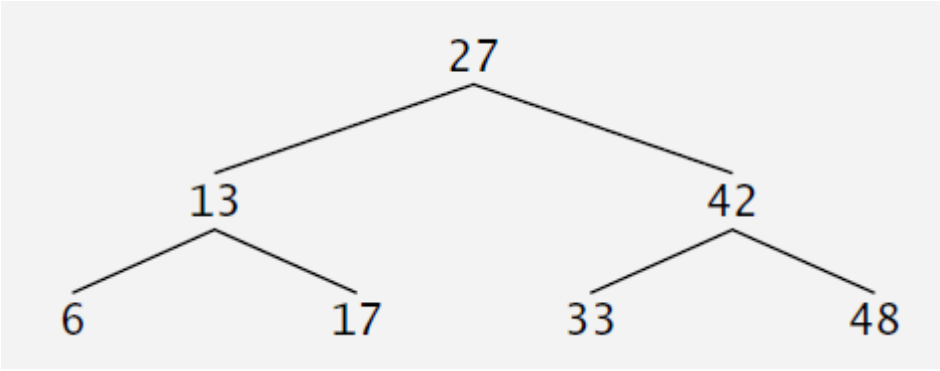
Recorrido en PreOrden

(**raíz**, izquierdo, derecho). Para recorrer un árbol binario no vacío en PreOrden, hay que realizar las siguientes operaciones recursivamente en cada nodo, comenzando con el nodo raíz:

Procesa la raíz

Recorrer el subárbol izquierdo en **PreOrden**

Recorrer el subárbol derecho en **PreOrden**



Secuencia: 27 – 13 – 6 – 17 – 42 – 33 - 48

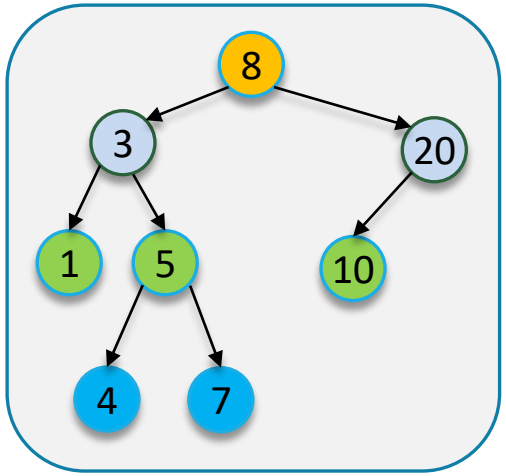
Recorrido en PreOrden

(**raíz**, izquierdo, derecho). Para recorrer un árbol binario no vacío en PreOrden, hay que realizar las siguientes operaciones recursivamente en cada nodo, comenzando con el nodo raíz:

Procesa la raíz

Recorrer el subárbol izquierdo en **PreOrden**

Recorrer el subárbol derecho en **PreOrden**



Secuencia: 8 – 3 – 1 – 5 – 4 – 7 – 20 – 10

Recorrido en PreOrden

(**raíz**, izquierdo, derecho)

```
void Arbol::preOrden(Nodo *nodoPtr)
{
    if(nodoPtr == NULL)
        return;

    cout << nodoPtr->dato << " - ";
    preOrden(nodoPtr->izquierdoPtr);
    preOrden(nodoPtr->derechoPtr);
}
```

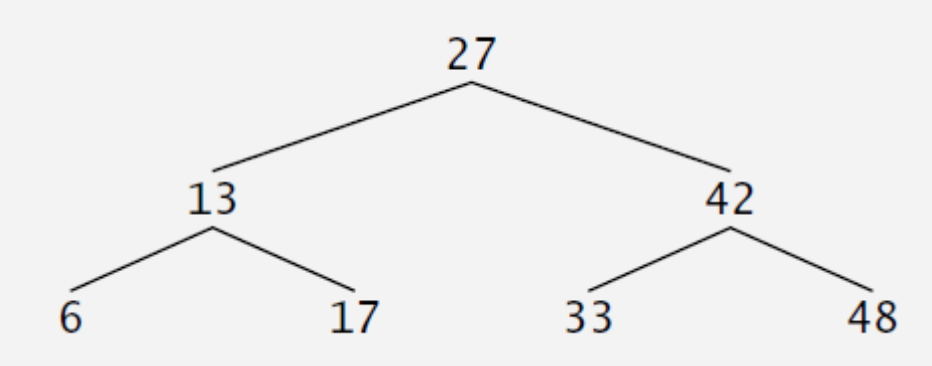
Recorrido en PostOrden

(izquierdo, derecho, **raíz**). Para recorrer un árbol binario no vacío en PostOrden, hay que realizar las siguientes operaciones recursivamente en cada nodo, comenzando con el nodo raíz:

Recorrer el subárbol izquierdo en **PostOrden**

Recorrer el subárbol derecho en **PostOrden**

Procesa la raíz



Secuencia: 6 – 17 – 13 – 33 – 48 – 42 - 27

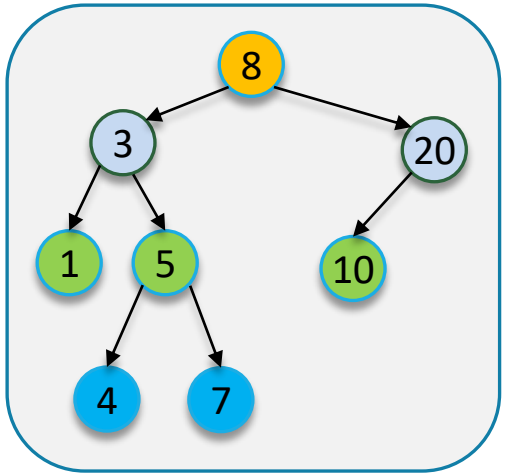
Recorrido en PostOrden

(izquierdo, derecho, **raíz**). Para recorrer un árbol binario no vacío en PostOrden, hay que realizar las siguientes operaciones recursivamente en cada nodo, comenzando con el nodo raíz:

Recorrer el subárbol izquierdo en **PostOrden**

Recorrer el subárbol derecho en **PostOrden**

Procesa la raíz



Secuencia: 1 – 4 – 7 – 5 – 3 – 10 – 20 – 8

Recorrido en PostOrden

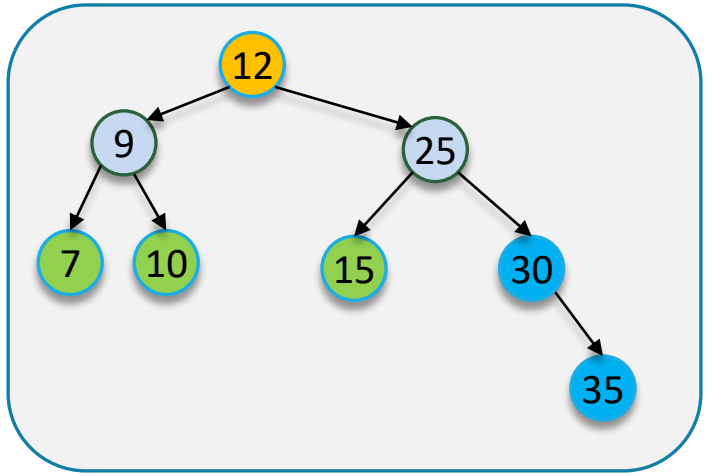
(izquierdo, derecho, **raíz**)

```
void Arbol::postOrden(Nodo *nodoPtr)
{
    if(nodoPtr == NULL)
        return;

    postOrden(nodoPtr->izquierdoPtr);
    postOrden(nodoPtr->derechoPtr);
    cout << nodoPtr->dato << " - ";
}
```

Árboles binarios de búsqueda. **Implementación**

- Actividad:

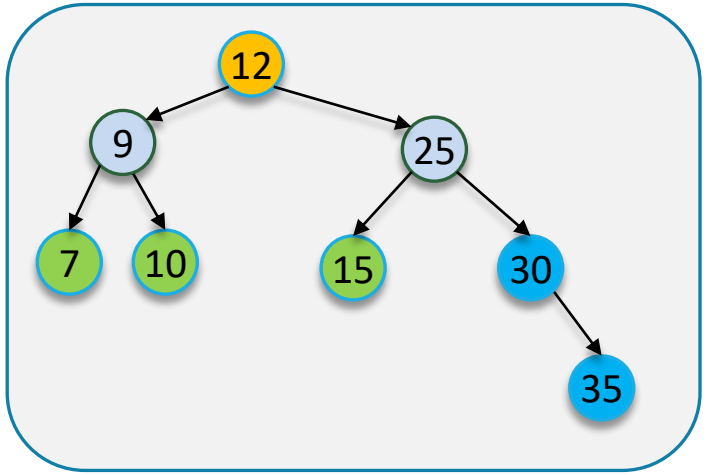
- Escribe la secuencia de elementos del siguiente árbol recorriéndolo en: inOrden, preOrden y postOrden.



Árboles binarios de búsqueda. **Implementación**

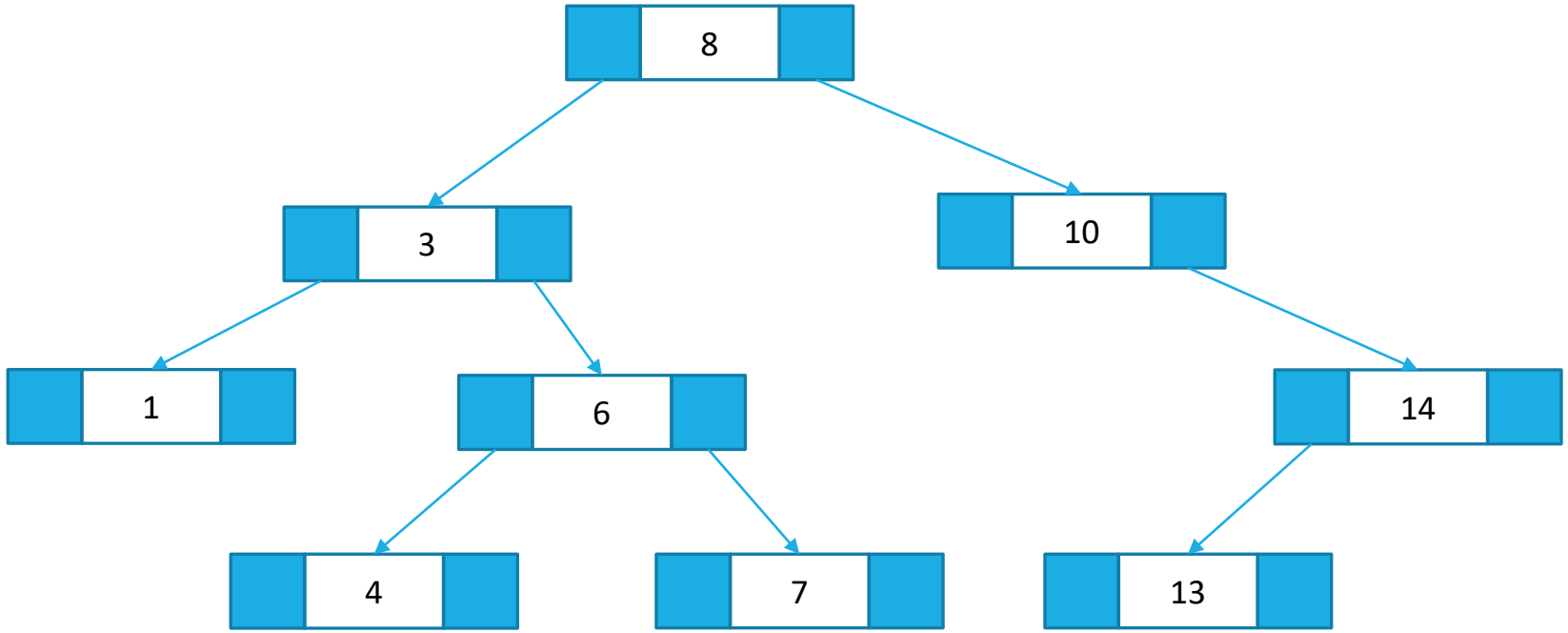
- Actividad:

- Solución:
- inOrden: 7, 9, 10, 12, 15, 25, 30, 35.
- preOrden: 12, 9, 7, 10, 25, 15, 30, 35.
- postOrden: 7, 10, 9, 15, 35, 30, 25, 12.



Árboles binarios de búsqueda. **Implementación**

- Actividad:
- Elabora la prueba de escritorio del recorrido en PostOrden del siguiente árbol:




```
void Arbol::postOrden(Nodo *nodoPtr)
{
    if(nodoPtr == NULL)
        return;

    postOrden(nodoPtr->izquierdoPtr);
    postOrden(nodoPtr->derechoPtr);
    cout << nodoPtr->dato << " - ";
}
```

Árboles binarios de búsqueda. **Implementación**

- Actividad:

- Elabora el método Arbol::busqueda(int x, Nodo *nodoPtr).

- Es el método para buscar un valor x de manera recursiva, en el campo dato de los nodos de un árbol.
- La llamada al método en el menú principal es realizada de la siguiente manera:

Árboles binarios de búsqueda. **Implementación**

- Actividad:
- Es decir, como valores iniciales se tienen el valor a buscar y la raíz del árbol.

```
case 5:
    if( raizArbolPtr == NULL )
    {
        cout << "\nEl arbol esta vacio.\n\n";
        system("pause");
        break;
    }

    cout << "\nIngrese valor entero: ";
    cin >> x;

    if( arbolEnteros.busqueda( x, raizArbolPtr ) == true )
        cout << "\nElemento " << x << " ha sido encontrado en el arbol\n";
    else
        cout << "\nElemento no encontrado\n";
```

Árboles binarios de búsqueda. **Implementación**

- Actividad:

- Por lo tanto:

1. nodoPtr toma como valor inicial, al ingresar al método, la raíz del árbol. Si el valor de x es mayor a la raíz, se realiza el paso recursivo con el lado derecho. Si el valor de x es menor se realiza el paso recursivo con el lado izquierdo.
2. Si x no es mayor ni menor, significa que se ha encontrado el nodo con el valor de x y regresará True.
3. Si se llegó a un nodo Nulo, se retorna False.

```
bool Arbol::busqueda(int x, Nodo *nodoPtr)
```

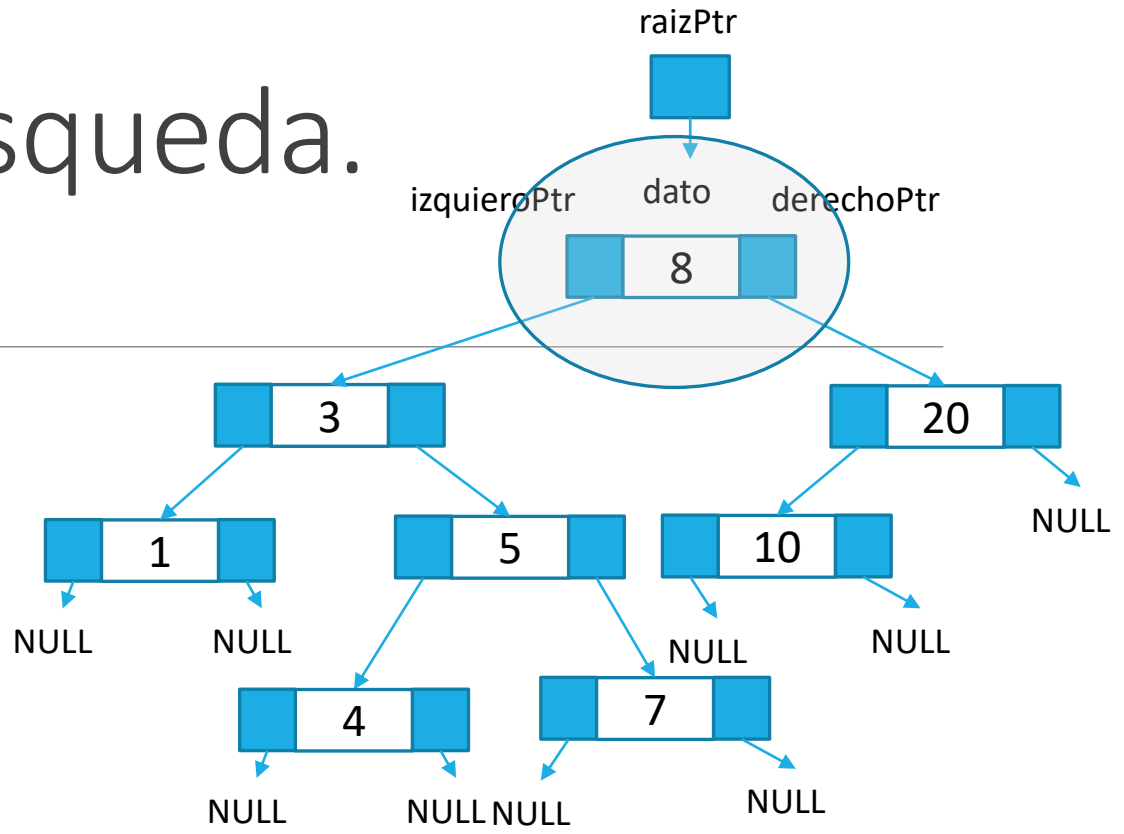
Árboles binarios de búsqueda. **Implementación**

- Método Arbol::busqueda(int x, Nodo *nodoPtr)
- Busca el valor de “x” en los nodos del árbol.

```
bool Arbol::busqueda(int x, Nodo *nodoPtr)
{
    if( nodoPtr == NULL )
        return false;
    else if( x < nodoPtr->dato )
        return busqueda( x, nodoPtr->izquierdoPtr );
    else if(x > nodoPtr->dato)
        return busqueda( x, nodoPtr->derechoPtr );
    else
        return true;
}
```

Árboles binarios de búsqueda. **Implementación**

```
bool Arbol::busqueda(int x, Nodo *nodoPtr)
{
    if( nodoPtr == NULL )
        return false;
    else if( x < nodoPtr->dato )
        return busqueda( x, nodoPtr->izquierdoPtr );
    else if(x > nodoPtr->dato)
        return busqueda( x, nodoPtr->derechoPtr );
    else
        return true;
}
```



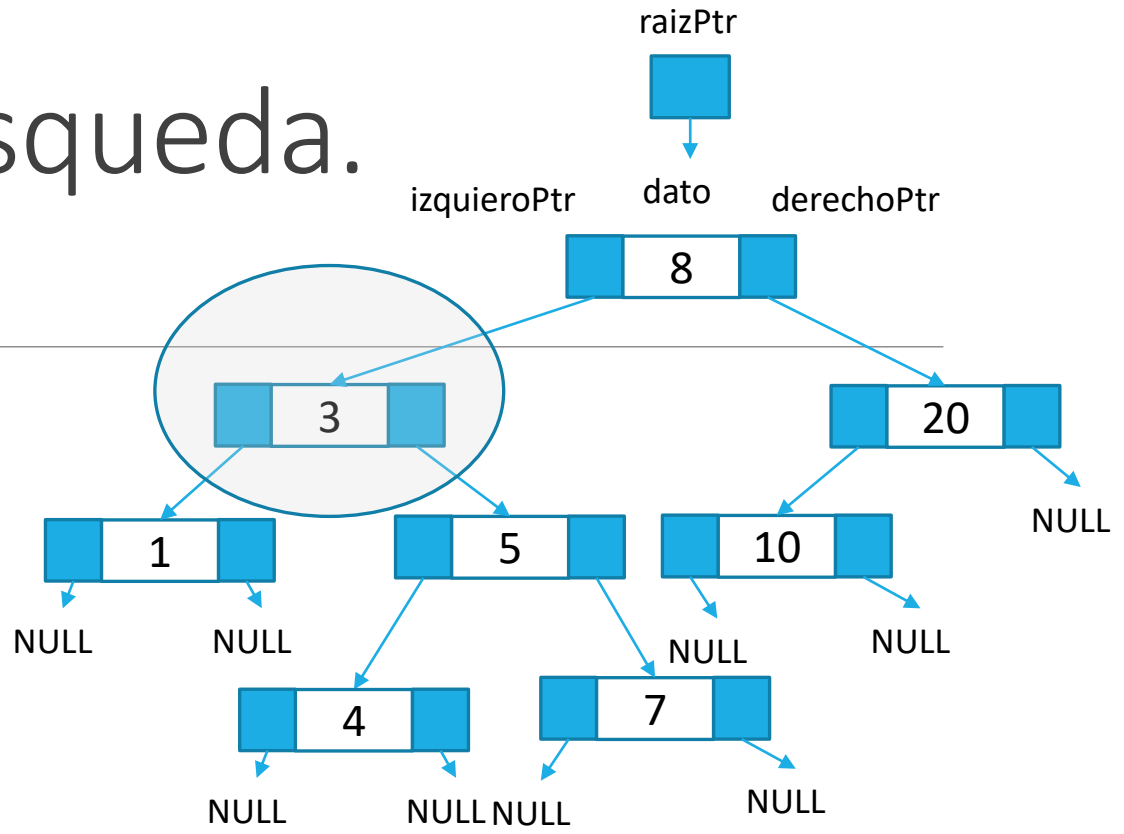
x=7

nodoPtr==Null?

x (7) < nodoPtr→dato (8)? ✓ **return búsqueda(x, nodoPtr→izquierdoPtr)**

Árboles binarios de búsqueda. **Implementación**

```
bool Arbol::busqueda(int x, Nodo *nodoPtr)
{
    if( nodoPtr == NULL )
        return false;
    else if( x < nodoPtr->dato )
        return busqueda( x, nodoPtr->izquierdoPtr );
    else if(x > nodoPtr->dato)
        return busqueda( x, nodoPtr->derechoPtr );
    else
        return true;
}
```



x=7

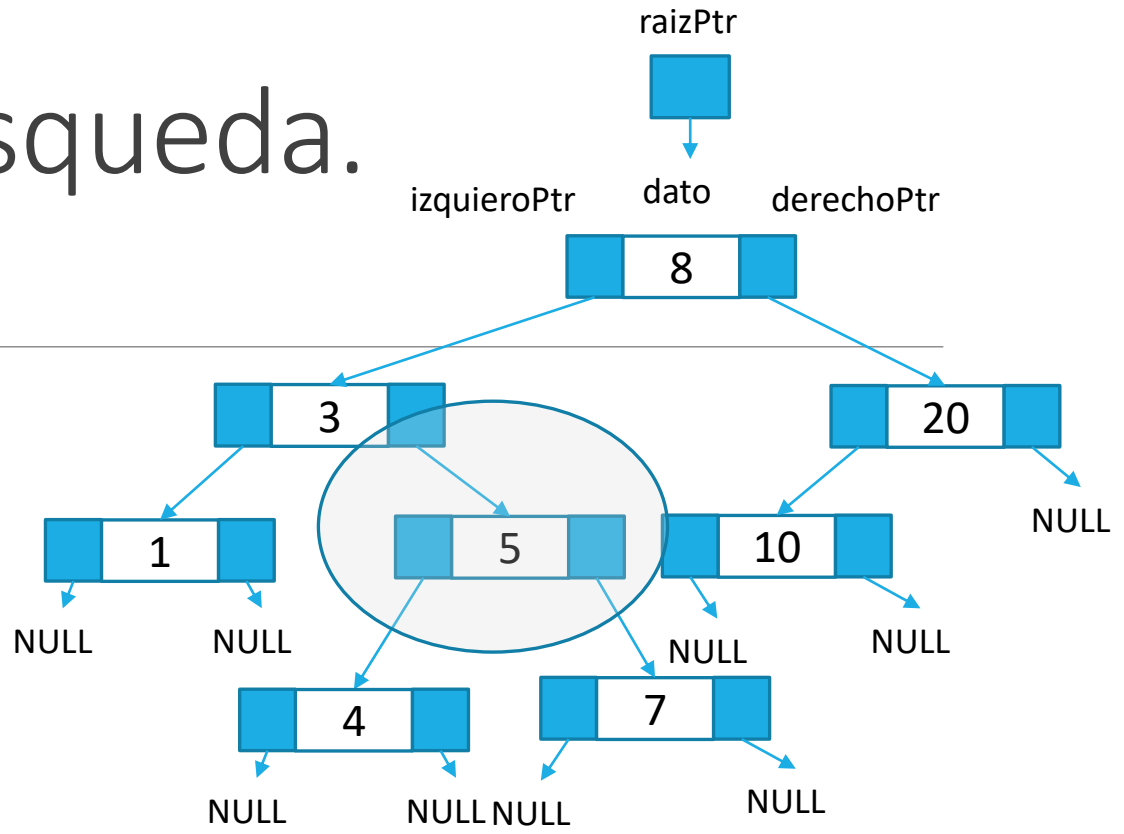
nodoPtr==Null?

x (7) < nodoPtr→dato (3)?

x (7) > nodoPtr→dato (3)? ✓ **return búsqueda(x, nodoPtr→derechoPtr)**

Árboles binarios de búsqueda. **Implementación**

```
bool Arbol::busqueda(int x, Nodo *nodoPtr)
{
    if( nodoPtr == NULL )
        return false;
    else if( x < nodoPtr->dato )
        return busqueda( x, nodoPtr->izquierdoPtr );
    else if(x > nodoPtr->dato)
        return busqueda( x, nodoPtr->derechoPtr );
    else
        return true;
}
```



x=7

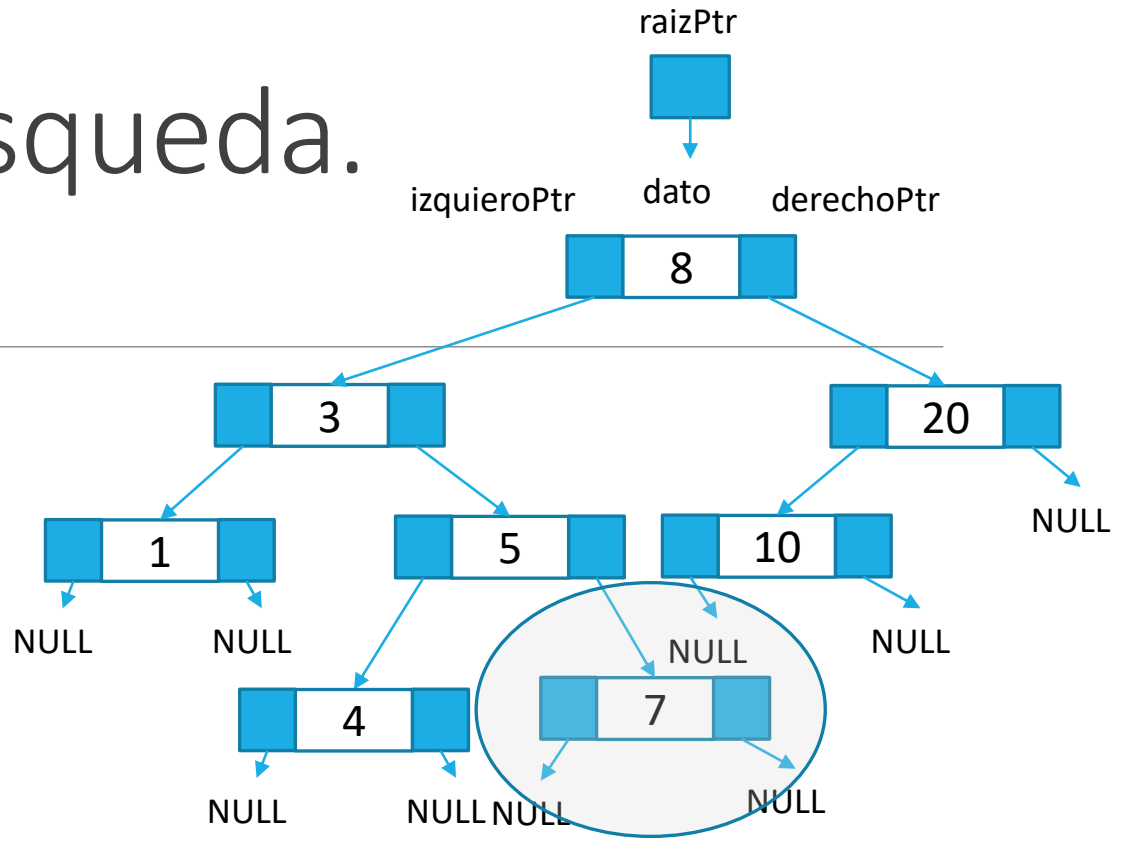
nodoPtr==Null?

x (7) < nodoPtr→dato (5)?

x (7) > nodoPtr→dato (5)? ✓ **return búsqueda(x, nodoPtr→derecho)**

Árboles binarios de búsqueda. **Implementación**

```
bool Arbol::busqueda(int x, Nodo *nodoPtr)
{
    if( nodoPtr == NULL )
        return false;
    else if( x < nodoPtr->dato )
        return busqueda( x, nodoPtr->izquierdoPtr );
    else if(x > nodoPtr->dato)
        return busqueda( x, nodoPtr->derechoPtr );
    else
        return true;
}
```



x=7

nodoPtr==Null?

x (7) < nodoPtr→dato (7)?

x (7) > nodoPtr→dato (7)?

return true ✓

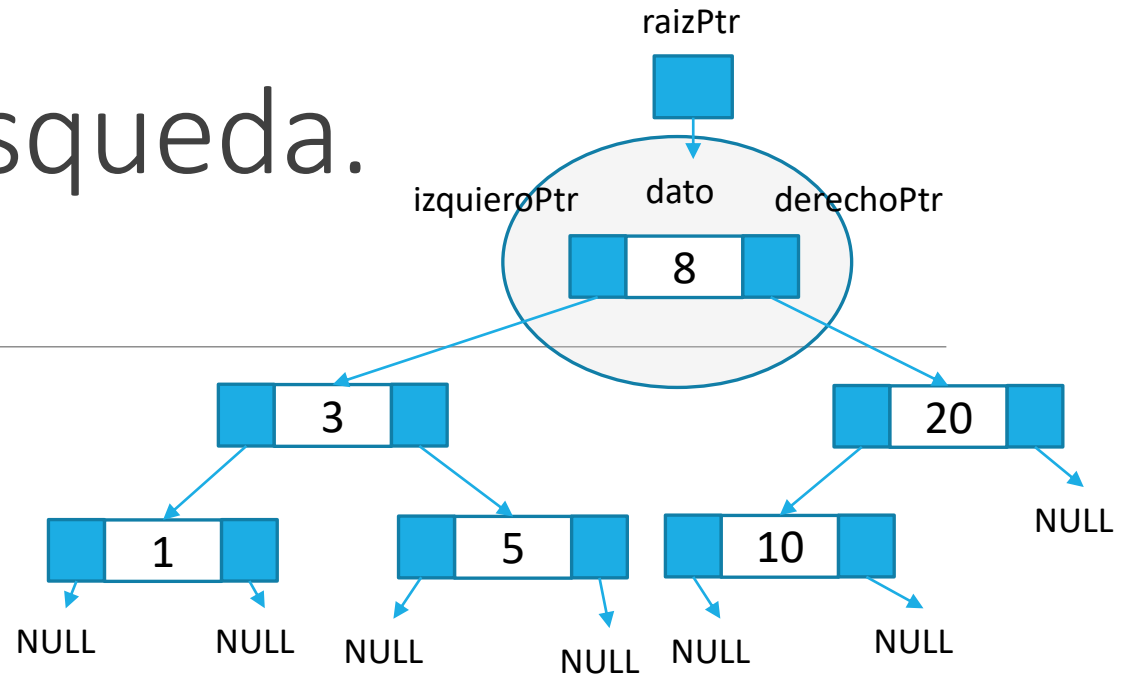
Árboles binarios de búsqueda. **Implementación**

- void Arbol::podarArbol(Nodo *&nodoPtr)

- Libera memoria de los nodos del árbol.

```
void Arbol::podarArbol(Nodo *&nodoPtr)
{
    if(nodoPtr == NULL)
        return; // Termina

    // Se elimina el subnodoPtr izquierdo
    podarArbol(nodoPtr->izquierdoPtr);
    // Se elimina el subnodoPtr derecho
    podarArbol(nodoPtr->derechoPtr);
    // Se elimina el nodo actual
    delete nodoPtr;
    nodoPtr = NULL;
}
```

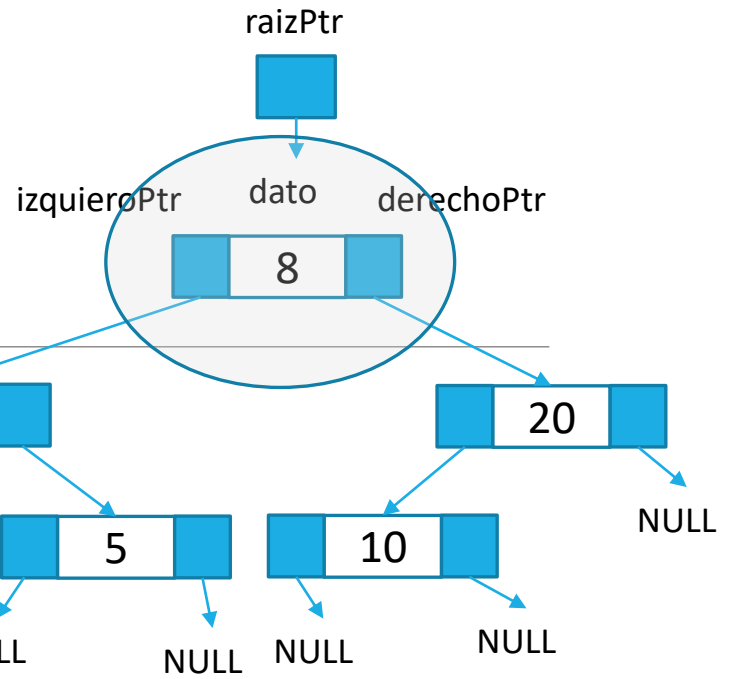
Árboles binarios de búsqueda. **Implementación**



- void Arbol::podarArbol(Nodo *&nodoPtr)

```
void Arbol::podarArbol(Nodo *&nodoPtr)
{
    if(nodoPtr == NULL)
        return; // Termina

    // Se elimina el subnodoPtr izquierdo
    podarArbol(nodoPtr->izquierdoPtr);
    // Se elimina el subnodoPtr derecho
    podarArbol(nodoPtr->derechoPtr);
    // Se elimina el nodo actual
    delete nodoPtr;
    nodoPtr = NULL;
}
```

Árboles binarios de búsqueda. **Implementación**

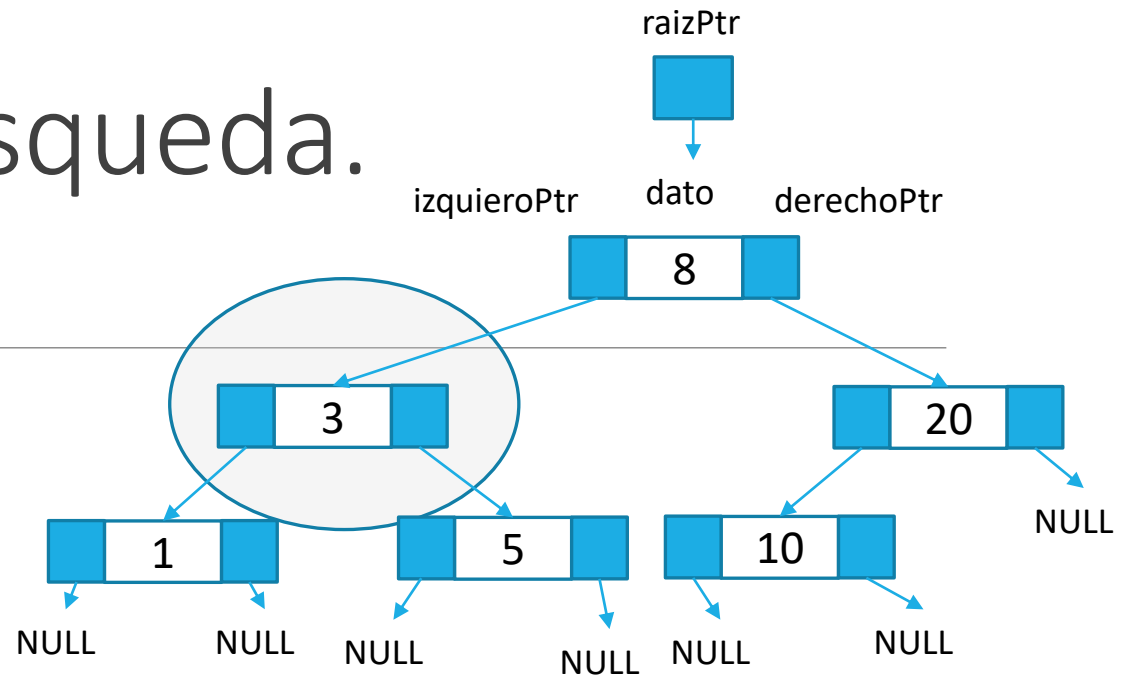


- void Arbol::podarArbol(Nodo *&nodoPtr)

```
void Arbol::podarArbol(Nodo *&nodoPtr)
{
    if(nodoPtr == NULL)
        return; // Termina

    // Se elimina el subnodoPtr izquierdo
    podarArbol(nodoPtr->izquierdoPtr);
    // Se elimina el subnodoPtr derecho
    podarArbol(nodoPtr->derechoPtr);
    // Se elimina el nodo actual
    delete nodoPtr;
    nodoPtr = NULL;
}
```

Pendiente 8

Árboles binarios de búsqueda. **Implementación**

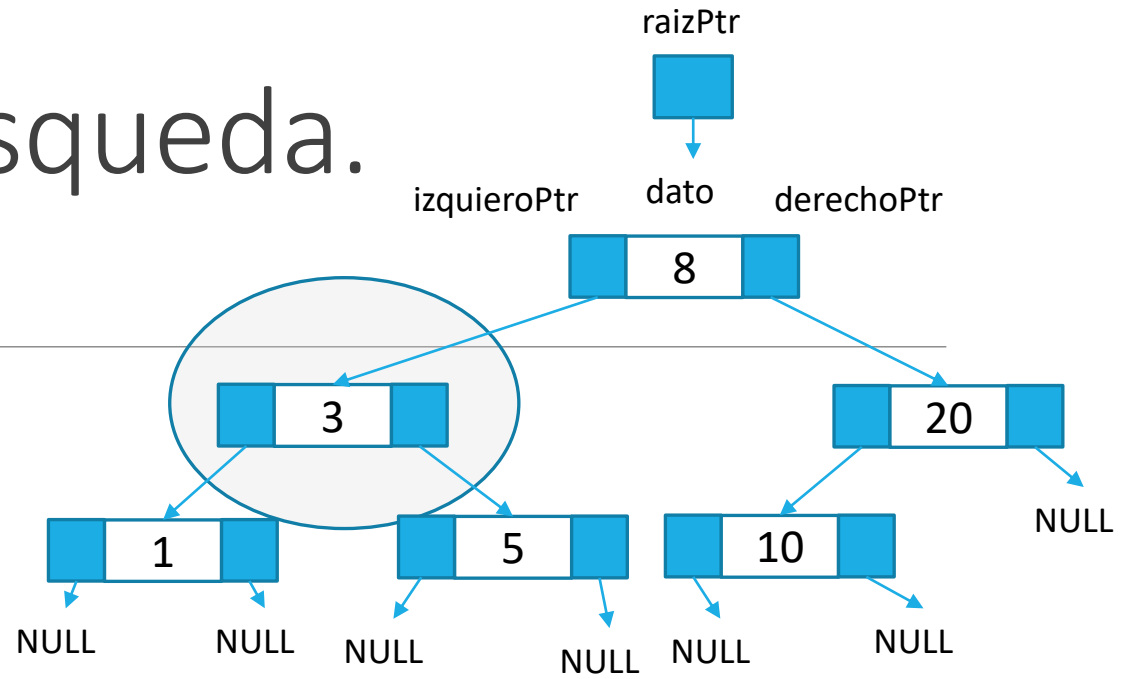


- void Arbol::podarArbol(Nodo *&nodoPtr)

```
void Arbol::podarArbol(Nodo *&nodoPtr)
{
    if(nodoPtr == NULL)
        return; // Termina

    // Se elimina el subnodoPtr izquierdo
    podarArbol(nodoPtr->izquierdoPtr);
    // Se elimina el subnodoPtr derecho
    podarArbol(nodoPtr->derechoPtr);
    // Se elimina el nodo actual
    delete nodoPtr;
    nodoPtr = NULL;
}
```

Pendiente 8

Árboles binarios de búsqueda. **Implementación**



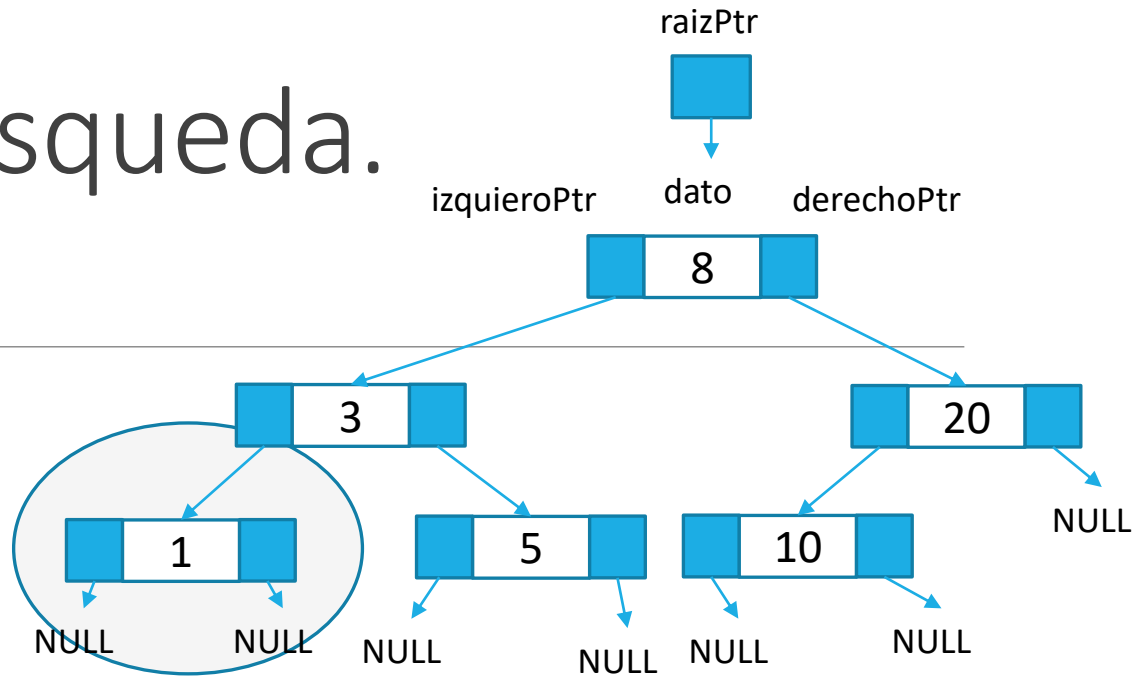
- void Arbol::podarArbol(Nodo *&nodoPtr)

```
void Arbol::podarArbol(Nodo *&nodoPtr)
{
    if(nodoPtr == NULL)
        return; // Termina

    // Se elimina el subnodoPtr izquierdo
    podarArbol(nodoPtr->izquierdoPtr);
    // Se elimina el subnodoPtr derecho
    podarArbol(nodoPtr->derechoPtr);
    // Se elimina el nodo actual
    delete nodoPtr;
    nodoPtr = NULL;
}
```

Pendiente 8

Pendiente 3

Árboles binarios de búsqueda. **Implementación**



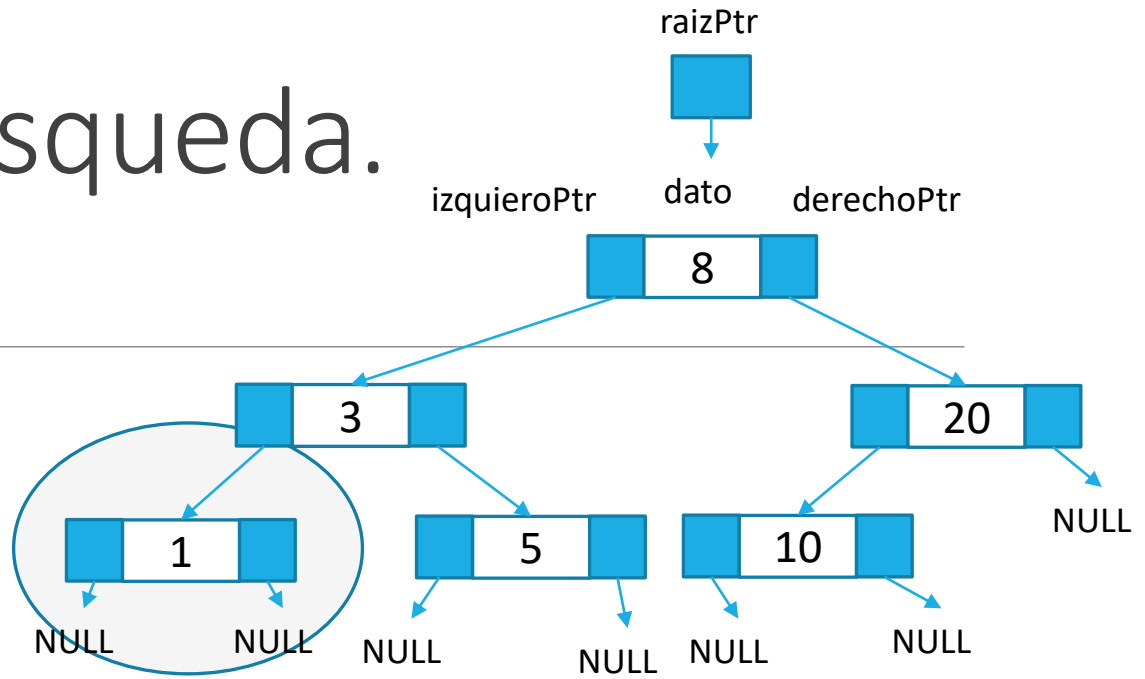
- void Arbol::podarArbol(Nodo *&nodoPtr)

```
void Arbol::podarArbol(Nodo *&nodoPtr)
{
    if(nodoPtr == NULL)
        return; // Termina

    // Se elimina el subnodoPtr izquierdo
    podarArbol(nodoPtr->izquierdoPtr);
    // Se elimina el subnodoPtr derecho
    podarArbol(nodoPtr->derechoPtr);
    // Se elimina el nodo actual
    delete nodoPtr;
    nodoPtr = NULL;
}
```

Pendiente 8

Pendiente 3

Árboles binarios de búsqueda. **Implementación**



- void Arbol::podarArbol(Nodo *&nodoPtr)

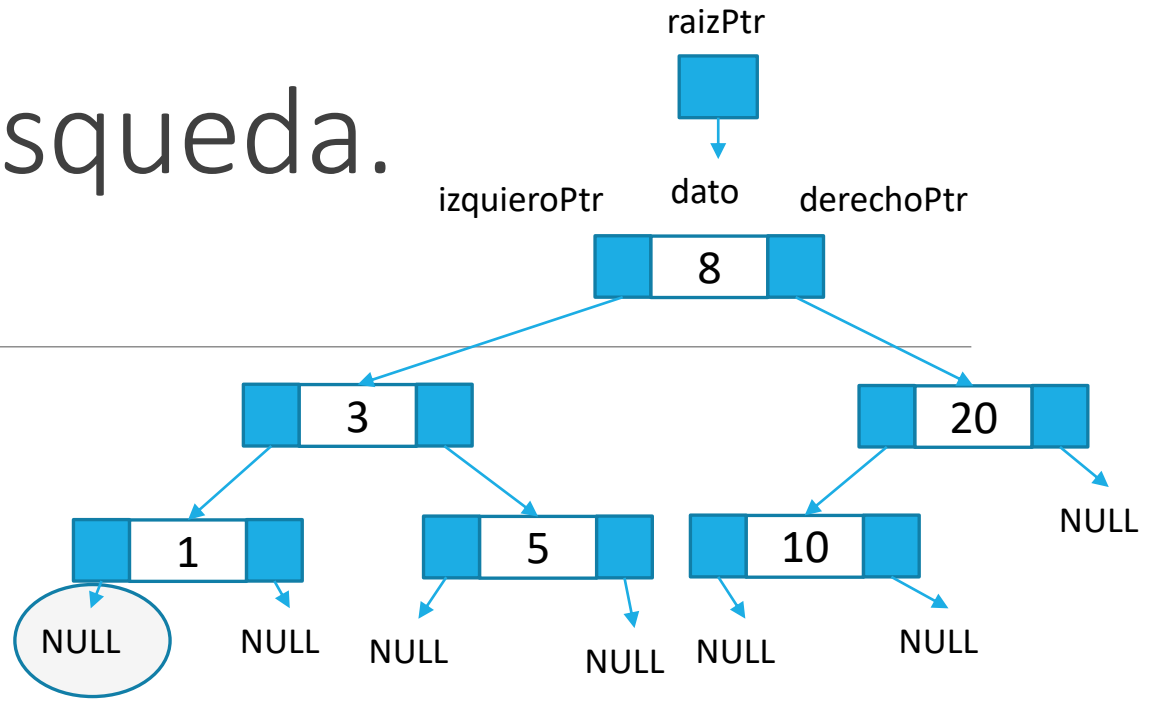
```
void Arbol::podarArbol(Nodo *&nodoPtr)
{
    if(nodoPtr == NULL)
        return; // Termina

    // Se elimina el subnodoPtr izquiendo
    podarArbol(nodoPtr->izquierdoPtr);
    // Se elimina el subnodoPtr derecho
    podarArbol(nodoPtr->derechoPtr);
    // Se elimina el nodo actual
    delete nodoPtr;
    nodoPtr = NULL;
}
```

Pendiente 8

Pendiente 3

Pendiente 1

Árboles binarios de búsqueda. **Implementación**



- void Arbol::podarArbol(Nodo *&nodoPtr)

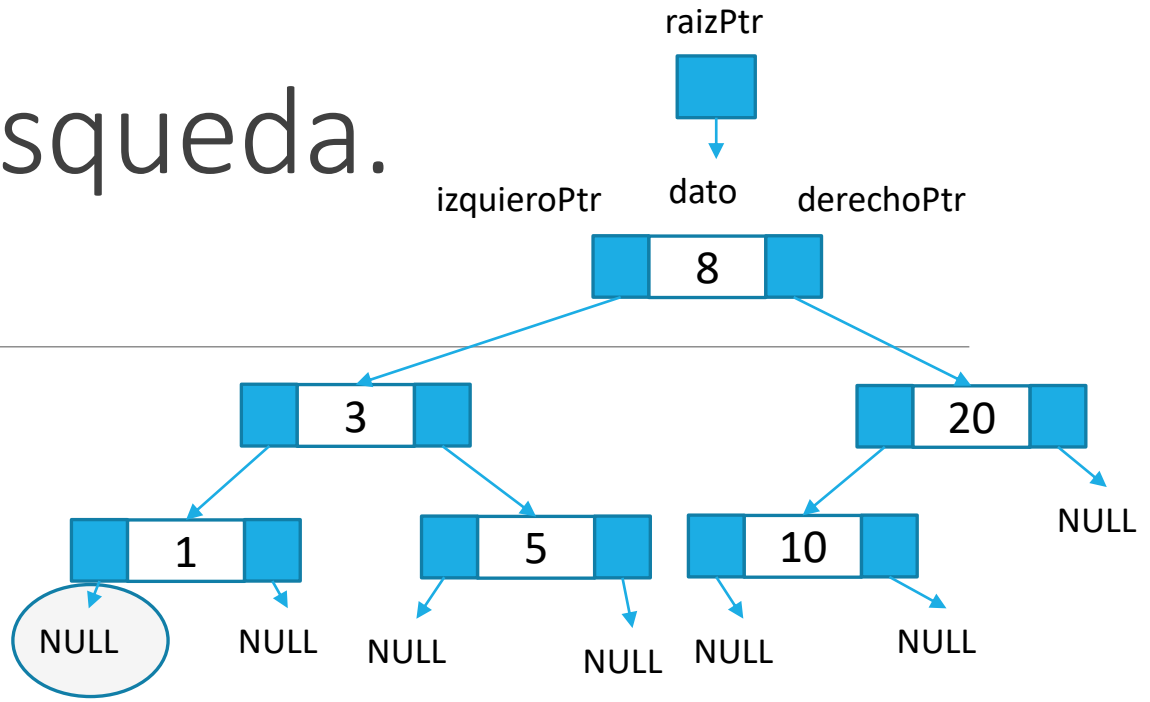
```
void Arbol::podarArbol(Nodo *&nodoPtr)
{
    if(nodoPtr == NULL)
        return; // Termina

    // Se elimina el subnodoPtr izquierdo
    podarArbol(nodoPtr->izquierdoPtr);
    // Se elimina el subnodoPtr derecho
    podarArbol(nodoPtr->derechoPtr);
    // Se elimina el nodo actual
    delete nodoPtr;
    nodoPtr = NULL;
}
```

Pendiente 8

Pendiente 3

Pendiente 1

Árboles binarios de búsqueda. **Implementación**



- void Arbol::podarArbol(Nodo *&nodoPtr)

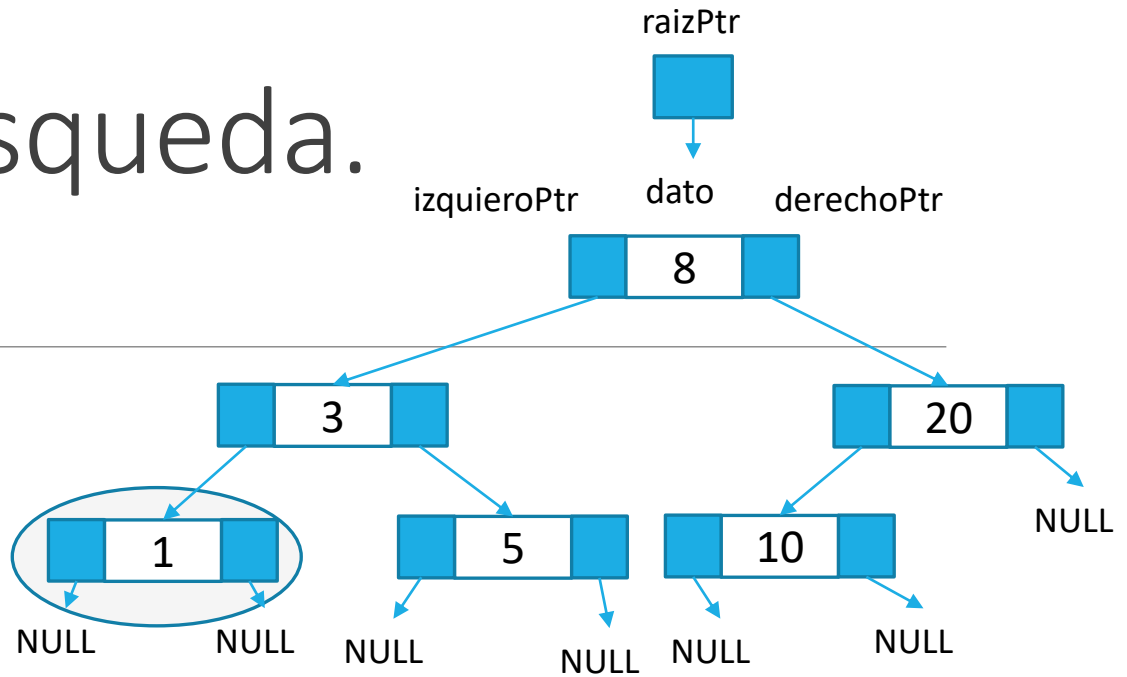
```
void Arbol::podarArbol(Nodo *&nodoPtr)
{
    if(nodoPtr == NULL)
        return; // Termina

    // Se elimina el subnodoPtr izquierdo
    podarArbol(nodoPtr->izquierdoPtr);
    // Se elimina el subnodoPtr derecho
    podarArbol(nodoPtr->derechoPtr);
    // Se elimina el nodo actual
    delete nodoPtr;
    nodoPtr = NULL;
}
```

Pendiente 8

Pendiente 3

Pendiente 1

Árboles binarios de búsqueda. **Implementación**



- void Arbol::podarArbol(Nodo *&nodoPtr)

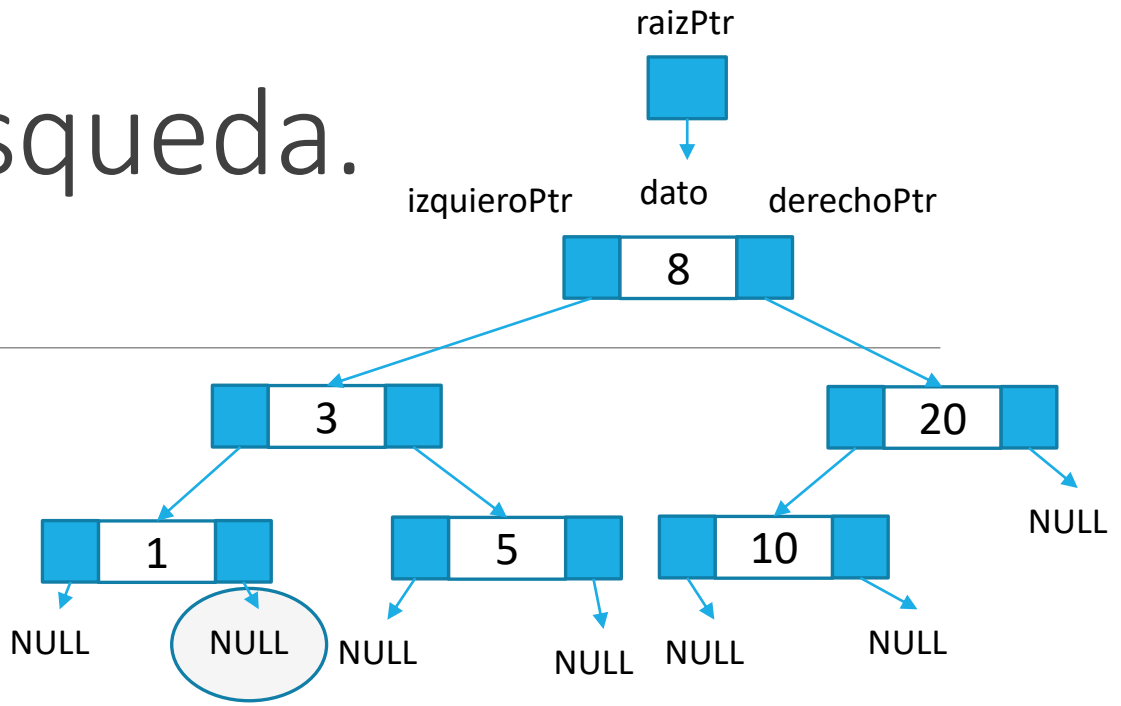
```
void Arbol::podarArbol(Nodo *&nodoPtr)
{
    if(nodoPtr == NULL)
        return; // Termina

    // Se elimina el subnodoPtr izquierdo
    podarArbol(nodoPtr->izquierdoPtr);
    // Se elimina el subnodoPtr derecho
    podarArbol(nodoPtr->derechoPtr);
    // Se elimina el nodo actual
    delete nodoPtr;
    nodoPtr = NULL;
}
```

Pendiente 8

Pendiente 3

Pendiente 1

Árboles binarios de búsqueda. **Implementación**



- void Arbol::podarArbol(Nodo *&nodoPtr)

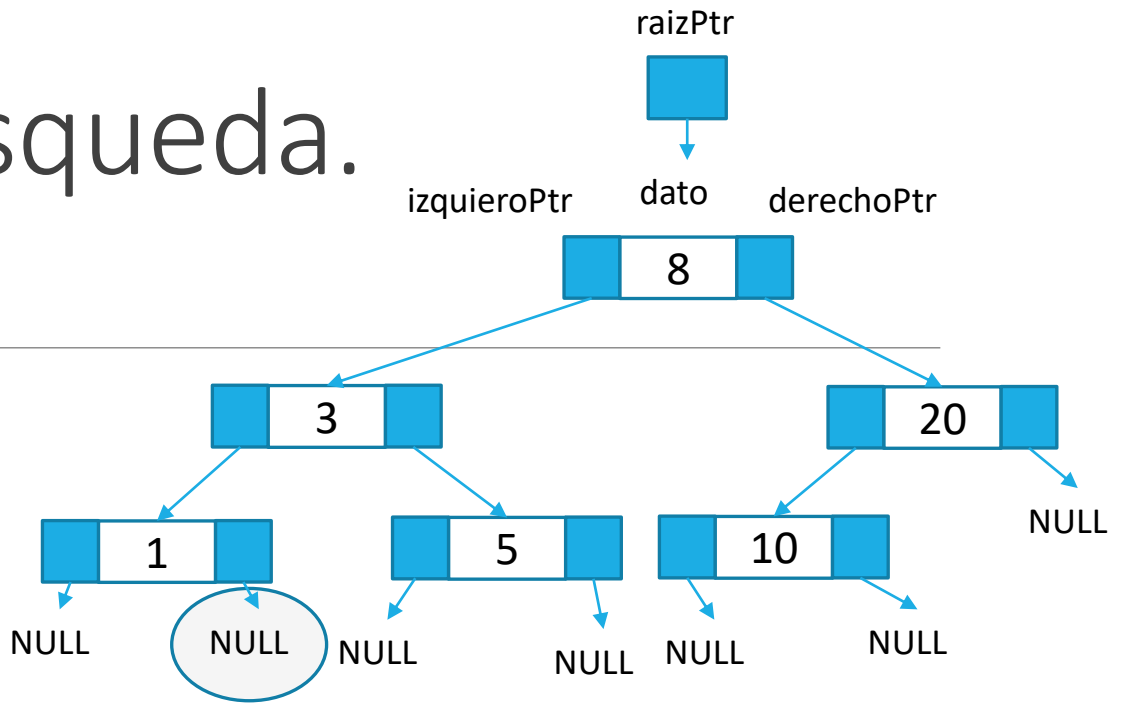
```
void Arbol::podarArbol(Nodo *&nodoPtr)
{
    if(nodoPtr == NULL)
        return; // Termina

    // Se elimina el subnodoPtr izquierdo
    podarArbol(nodoPtr->izquierdoPtr);
    // Se elimina el subnodoPtr derecho
    podarArbol(nodoPtr->derechoPtr);
    // Se elimina el nodo actual
    delete nodoPtr;
    nodoPtr = NULL;
}
```

Pendiente 8

Pendiente 3

Pendiente 1

Árboles binarios de búsqueda. **Implementación**



- void Arbol::podarArbol(Nodo *&nodoPtr)

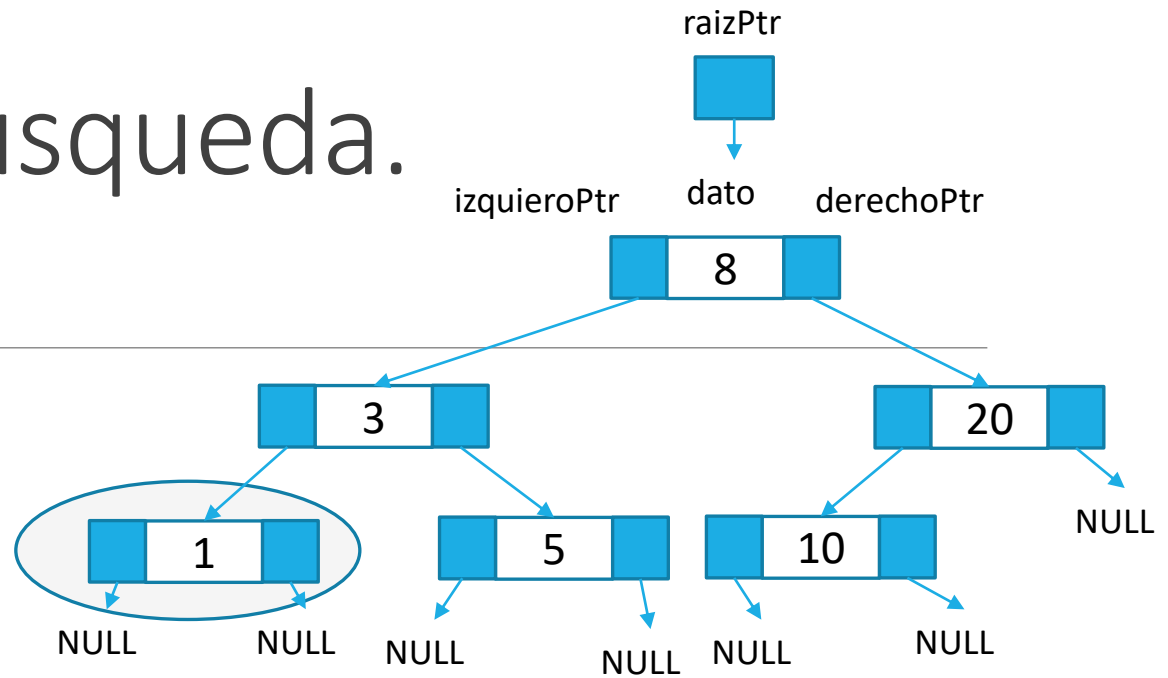
```
void Arbol::podarArbol(Nodo *&nodoPtr)
{
    if(nodoPtr == NULL)
        return; // Termina

    // Se elimina el subnodoPtr izquierdo
    podarArbol(nodoPtr->izquierdoPtr);
    // Se elimina el subnodoPtr derecho
    podarArbol(nodoPtr->derechoPtr);
    // Se elimina el nodo actual
    delete nodoPtr;
    nodoPtr = NULL;
}
```

Pendiente 8

Pendiente 3

Pendiente 1

Árboles binarios de búsqueda. **Implementación**



- void Arbol::podarArbol(Nodo *&nodoPtr)

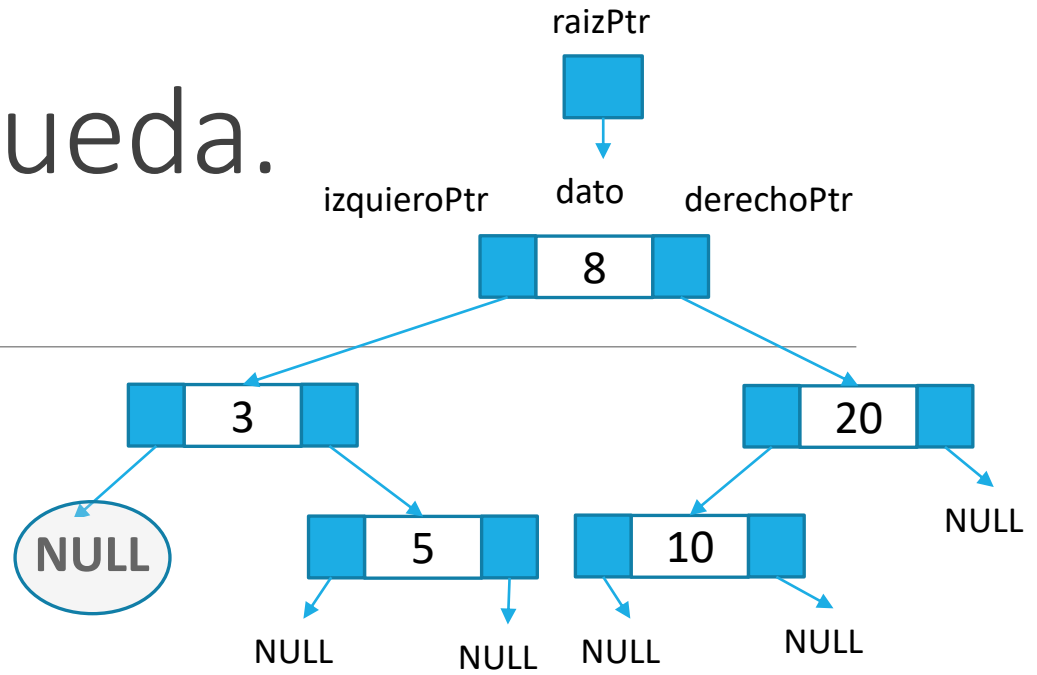
```
void Arbol::podarArbol(Nodo *&nodoPtr)
{
    if(nodoPtr == NULL)
        return; // Termina

    // Se elimina el subnodoPtr izquierdo
    podarArbol(nodoPtr->izquierdoPtr);
    // Se elimina el subnodoPtr derecho
    podarArbol(nodoPtr->derechoPtr);
    // Se elimina el nodo actual
    delete nodoPtr;
    nodoPtr = NULL;
}
```

Pendiente 8

Pendiente 3

Pendiente 1. Elimina nodo

Árboles binarios de búsqueda. **Implementación**



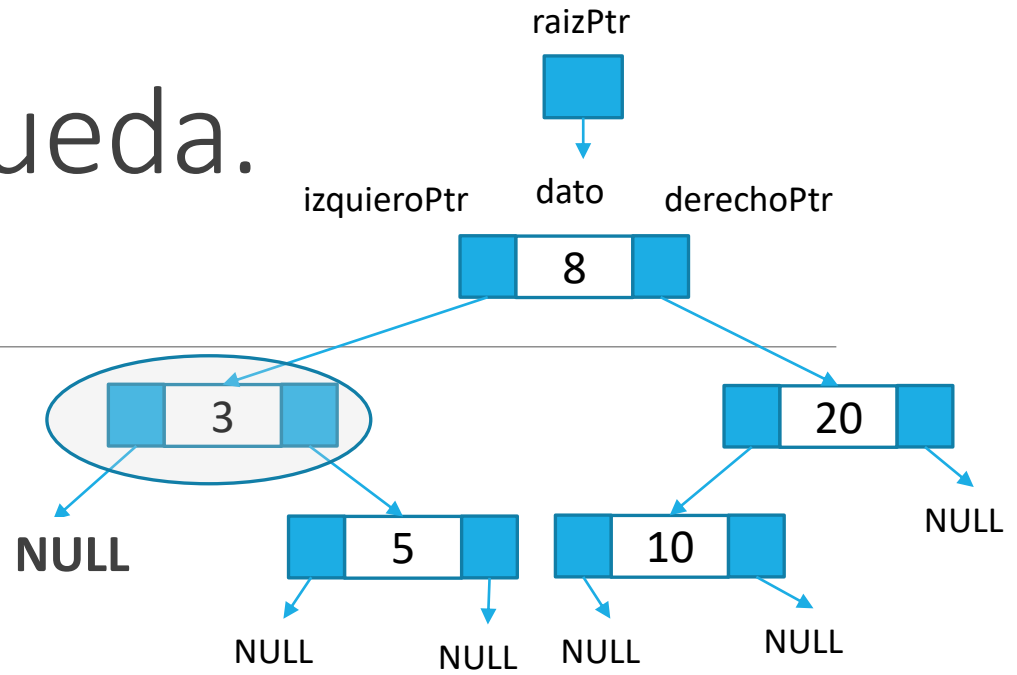
- void Arbol::podarArbol(Nodo *&nodoPtr)

```
void Arbol::podarArbol(Nodo *&nodoPtr)
{
    if(nodoPtr == NULL)
        return; // Termina

    // Se elimina el subnodoPtr izquierdo
    podarArbol(nodoPtr->izquierdoPtr);
    // Se elimina el subnodoPtr derecho
    podarArbol(nodoPtr->derechoPtr);
    // Se elimina el nodo actual
    delete nodoPtr;
    nodoPtr = NULL;
}
```

Pendiente 8

Pendiente 3

Árboles binarios de búsqueda. **Implementación**



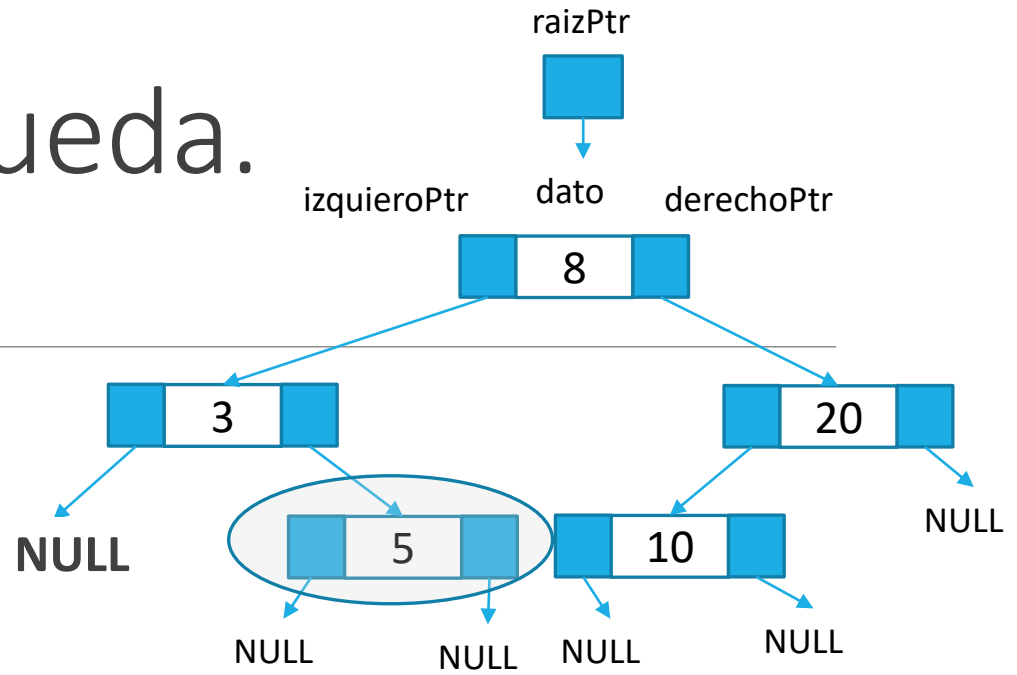
- void Arbol::podarArbol(Nodo *&nodoPtr)

```
void Arbol::podarArbol(Nodo *&nodoPtr)
{
    if(nodoPtr == NULL)
        return; // Termina

    // Se elimina el subnodoPtr izquierdo
    podarArbol(nodoPtr->izquierdoPtr);
    // Se elimina el subnodoPtr derecho
    podarArbol(nodoPtr->derechoPtr);
    // Se elimina el nodo actual
    delete nodoPtr;
    nodoPtr = NULL;
}
```

Pendiente 8

Pendiente 3

Árboles binarios de búsqueda. **Implementación**



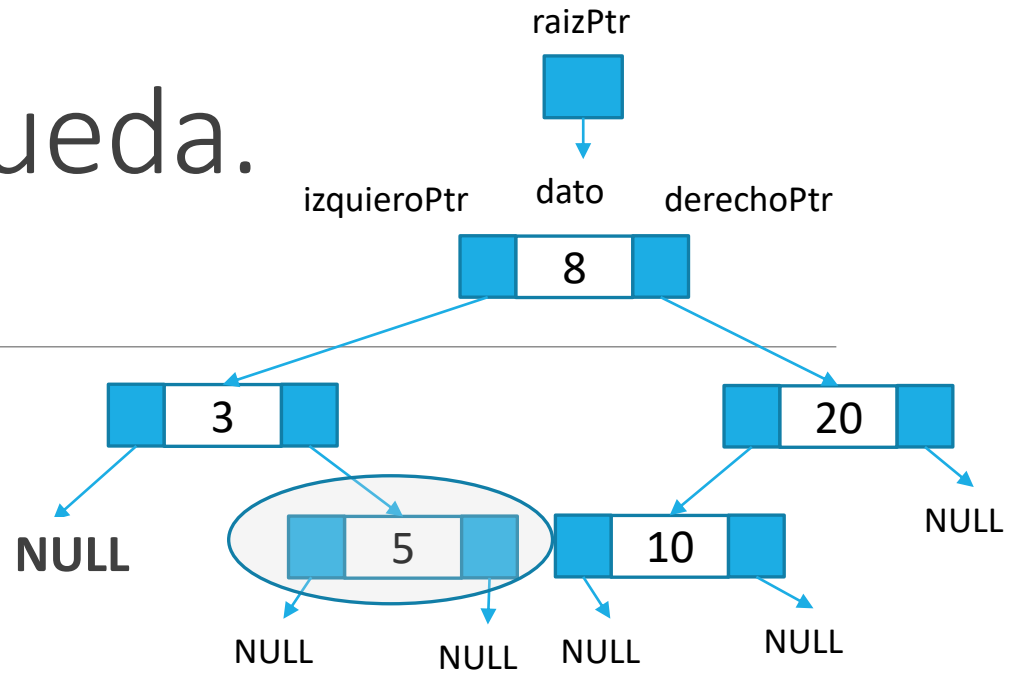
- void Arbol::podarArbol(Nodo *&nodoPtr)

```
void Arbol::podarArbol(Nodo *&nodoPtr)
{
    if(nodoPtr == NULL)
        return; // Termina

    // Se elimina el subnodoPtr izquierdo
    podarArbol(nodoPtr->izquierdoPtr);
    // Se elimina el subnodoPtr derecho
    podarArbol(nodoPtr->derechoPtr);
    // Se elimina el nodo actual
    delete nodoPtr;
    nodoPtr = NULL;
}
```

Pendiente 8

Pendiente 3

Árboles binarios de búsqueda. **Implementación**



- void Arbol::podarArbol(Nodo *&nodoPtr)

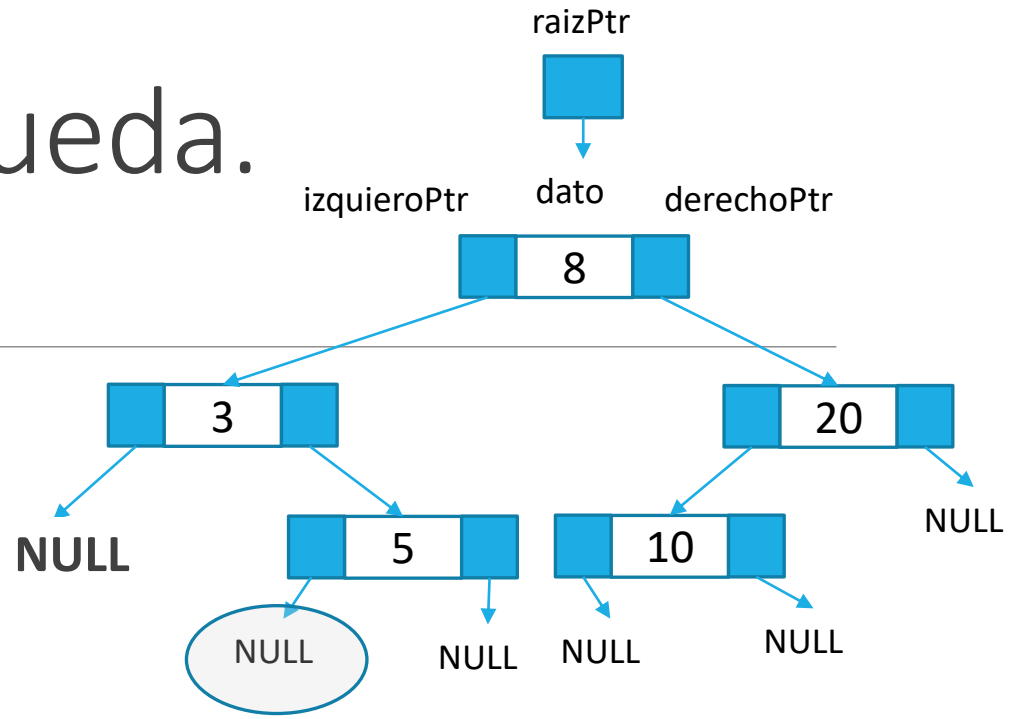
```
void Arbol::podarArbol(Nodo *&nodoPtr)
{
    if(nodoPtr == NULL)
        return; // Termina

    // Se elimina el subnodoPtr izquierdo
    podarArbol(nodoPtr->izquierdoPtr);
    // Se elimina el subnodoPtr derecho
    podarArbol(nodoPtr->derechoPtr);
    // Se elimina el nodo actual
    delete nodoPtr;
    nodoPtr = NULL;
}
```

Pendiente 8

Pendiente 3

Pendiente 5

Árboles binarios de búsqueda. **Implementación**



- void Arbol::podarArbol(Nodo *&nodoPtr)

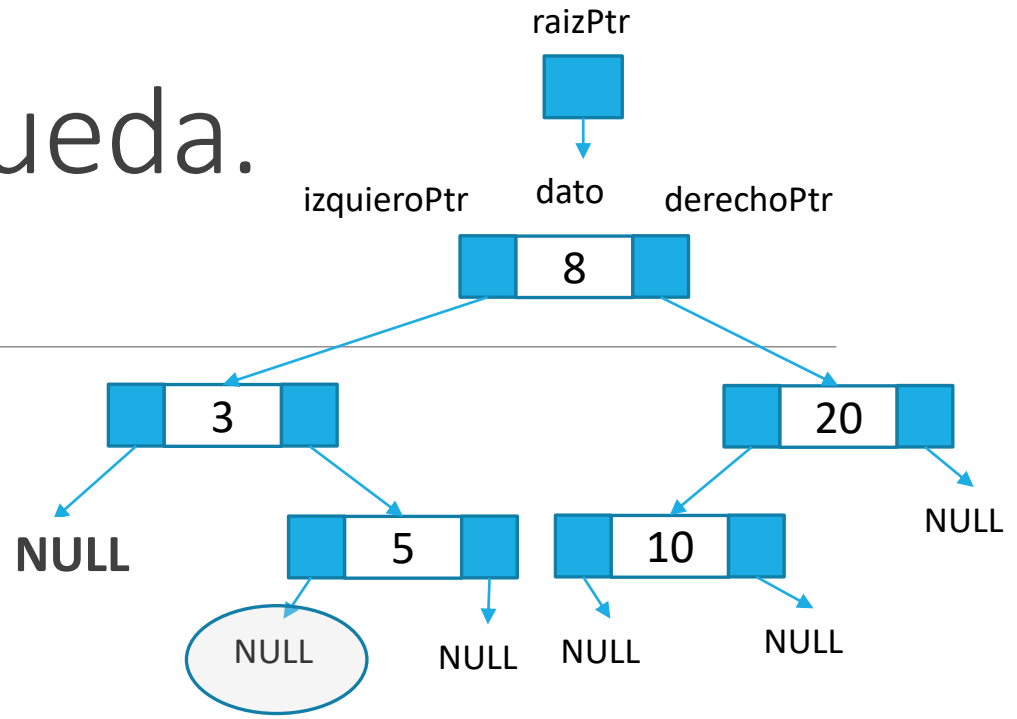
```
void Arbol::podarArbol(Nodo *&nodoPtr)
{
    if(nodoPtr == NULL)
        return; // Termina

    // Se elimina el subnodoPtr izquierdo
    podarArbol(nodoPtr->izquierdoPtr);
    // Se elimina el subnodoPtr derecho
    podarArbol(nodoPtr->derechoPtr);
    // Se elimina el nodo actual
    delete nodoPtr;
    nodoPtr = NULL;
}
```

Pendiente 8

Pendiente 3

Pendiente 5

Árboles binarios de búsqueda. **Implementación**



- void Arbol::podarArbol(Nodo *&nodoPtr)

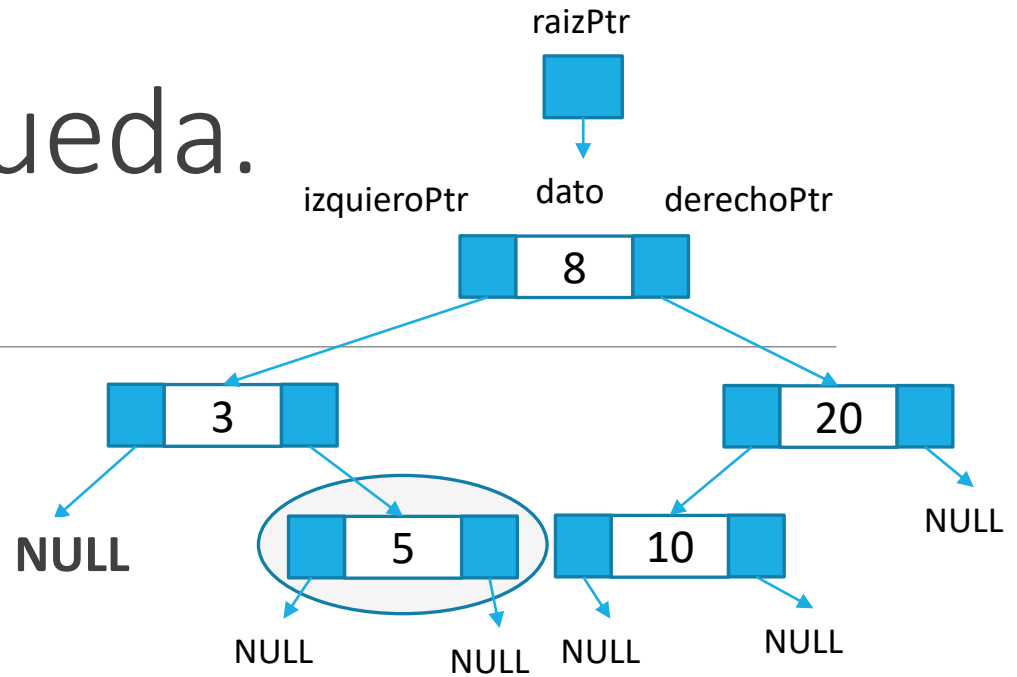
```
void Arbol::podarArbol(Nodo *&nodoPtr)
{
    if(nodoPtr == NULL)
        return; // Termina

    // Se elimina el subnodoPtr izquierdo
    podarArbol(nodoPtr->izquierdoPtr);
    // Se elimina el subnodoPtr derecho
    podarArbol(nodoPtr->derechoPtr);
    // Se elimina el nodo actual
    delete nodoPtr;
    nodoPtr = NULL;
}
```

Pendiente 8

Pendiente 3

Pendiente 5

Árboles binarios de búsqueda. **Implementación**



- void Arbol::podarArbol(Nodo *&nodoPtr)

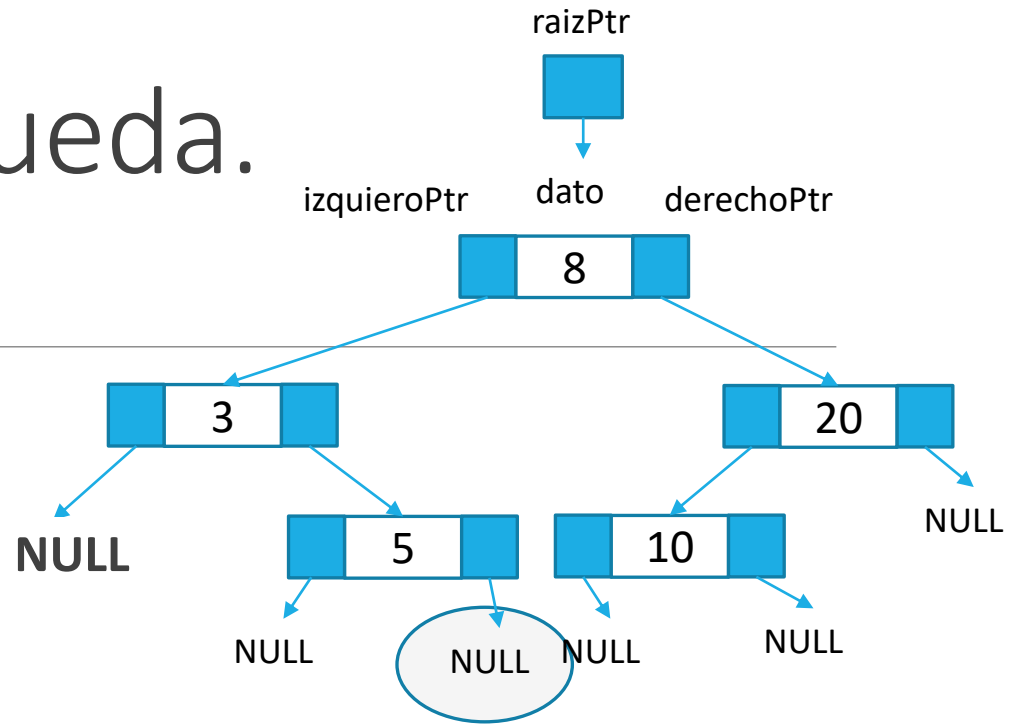
```
void Arbol::podarArbol(Nodo *&nodoPtr)
{
    if(nodoPtr == NULL)
        return; // Termina

    // Se elimina el subnodoPtr izquierdo
    podarArbol(nodoPtr->izquierdoPtr);
    // Se elimina el subnodoPtr derecho
    podarArbol(nodoPtr->derechoPtr);
    // Se elimina el nodo actual
    delete nodoPtr;
    nodoPtr = NULL;
}
```

Pendiente 8

Pendiente 3

Pendiente 5

Árboles binarios de búsqueda. **Implementación**



- void Arbol::podarArbol(Nodo *&nodoPtr)

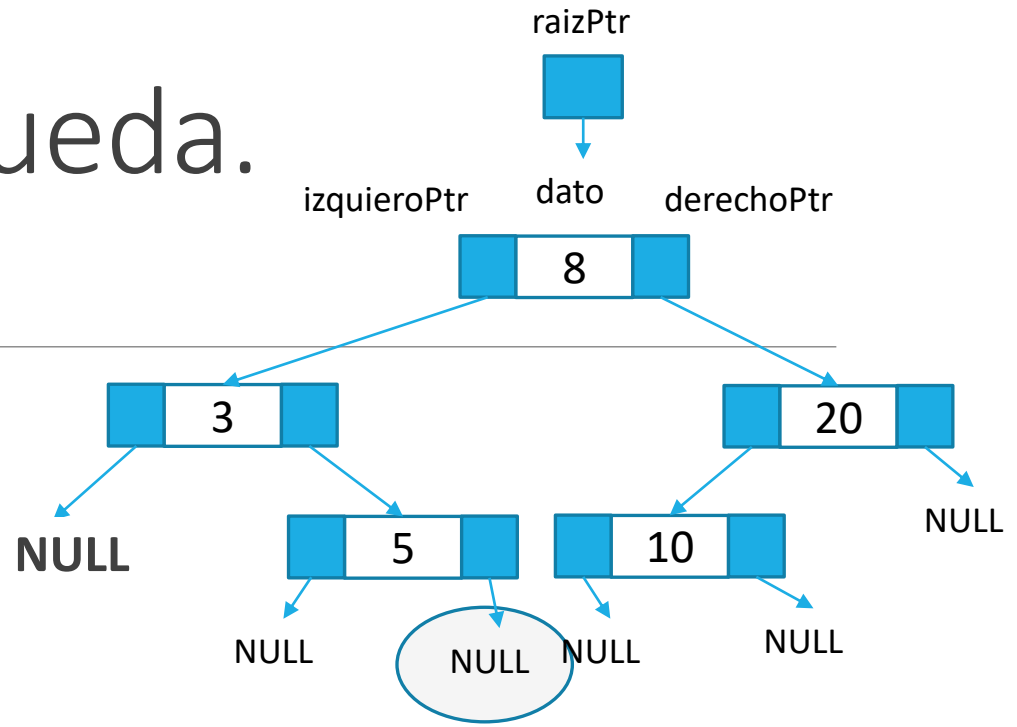
```
void Arbol::podarArbol(Nodo *&nodoPtr)
{
    if(nodoPtr == NULL)
        return; // Termina

    // Se elimina el subnodoPtr izquierdo
    podarArbol(nodoPtr->izquierdoPtr);
    // Se elimina el subnodoPtr derecho
    podarArbol(nodoPtr->derechoPtr);
    // Se elimina el nodo actual
    delete nodoPtr;
    nodoPtr = NULL;
}
```

Pendiente 8

Pendiente 3

Pendiente 5

Árboles binarios de búsqueda. **Implementación**



- void Arbol::podarArbol(Nodo *&nodoPtr)

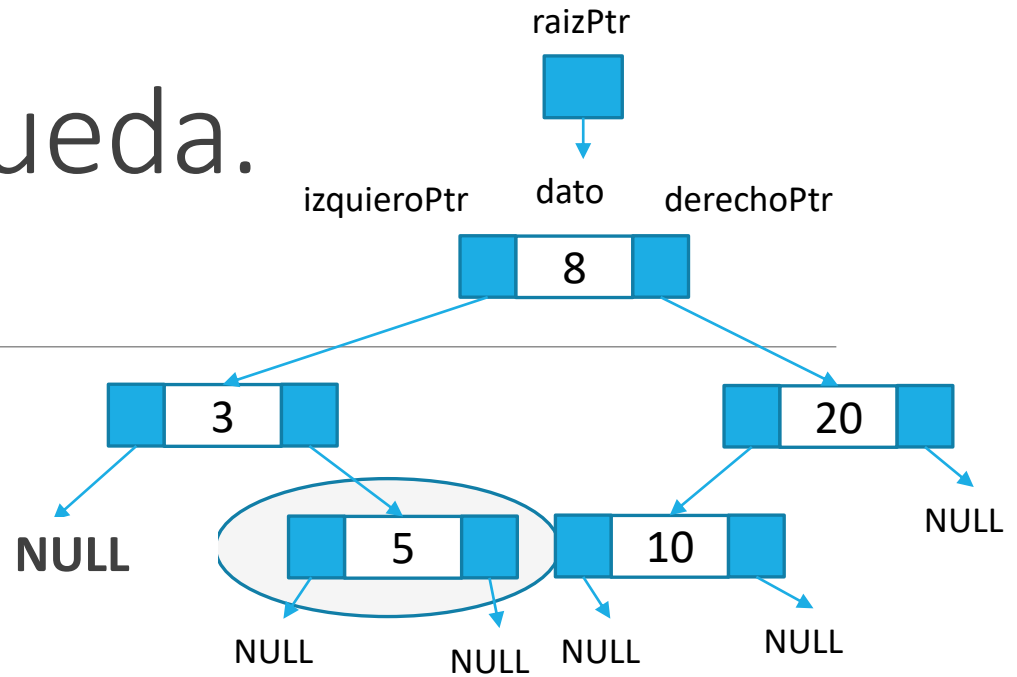
```
void Arbol::podarArbol(Nodo *&nodoPtr)
{
    if(nodoPtr == NULL)
        return; // Termina

    // Se elimina el subnodoPtr izquierdo
    podarArbol(nodoPtr->izquierdoPtr);
    // Se elimina el subnodoPtr derecho
    podarArbol(nodoPtr->derechoPtr);
    // Se elimina el nodo actual
    delete nodoPtr;
    nodoPtr = NULL;
}
```

Pendiente 8

Pendiente 3

Pendiente 5

Árboles binarios de búsqueda. **Implementación**



- void Arbol::podarArbol(Nodo *&nodoPtr)

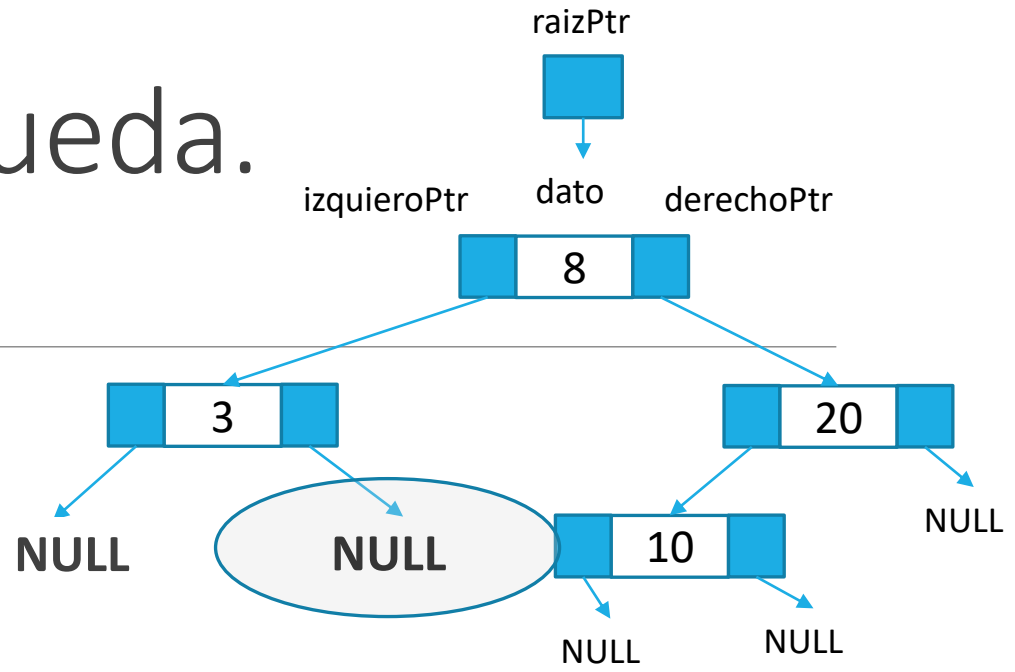
```
void Arbol::podarArbol(Nodo *&nodoPtr)
{
    if(nodoPtr == NULL)
        return; // Termina

    // Se elimina el subnodoPtr izquierdo
    podarArbol(nodoPtr->izquierdoPtr);
    // Se elimina el subnodoPtr derecho
    podarArbol(nodoPtr->derechoPtr);
    // Se elimina el nodo actual
    delete nodoPtr;
    nodoPtr = NULL;
}
```

Pendiente 8

Pendiente 3

Pendiente 5. Elimina nodo

Árboles binarios de búsqueda. **Implementación**



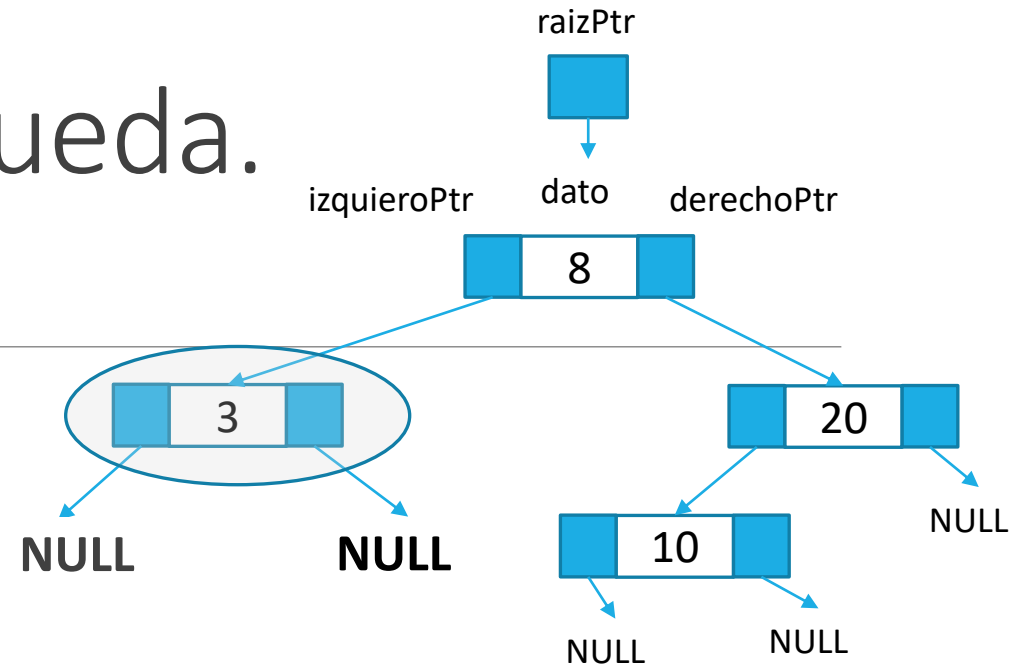
- void Arbol::podarArbol(Nodo *&nodoPtr)

```
void Arbol::podarArbol(Nodo *&nodoPtr)
{
    if(nodoPtr == NULL)
        return; // Termina

    // Se elimina el subnodoPtr izquierdo
    podarArbol(nodoPtr->izquierdoPtr);
    // Se elimina el subnodoPtr derecho
    podarArbol(nodoPtr->derechoPtr);
    // Se elimina el nodo actual
    delete nodoPtr;
    nodoPtr = NULL;
}
```

Pendiente 8

Pendiente 3

Árboles binarios de búsqueda. **Implementación**



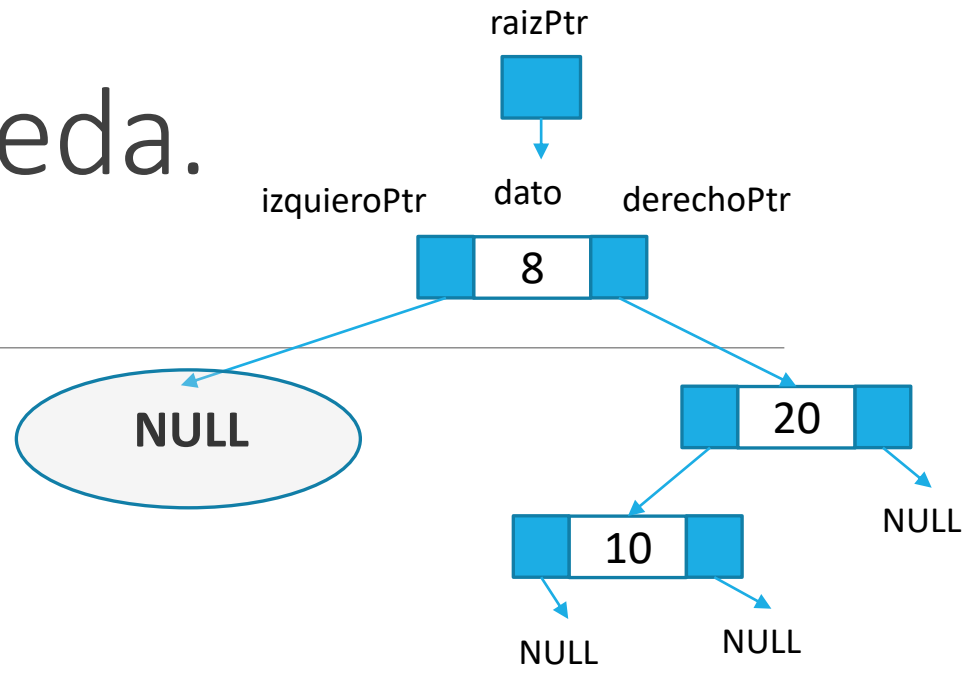
- void Arbol::podarArbol(Nodo *&nodoPtr)

```
void Arbol::podarArbol(Nodo *&nodoPtr)
{
    if(nodoPtr == NULL)
        return; // Termina

    // Se elimina el subnodoPtr izquierdo
    podarArbol(nodoPtr->izquierdoPtr);
    // Se elimina el subnodoPtr derecho
    podarArbol(nodoPtr->derechoPtr);
    // Se elimina el nodo actual
    delete nodoPtr;
    nodoPtr = NULL;
}
```

Pendiente 8

Pendiente 3. Elimina nodo.

Árboles binarios de búsqueda. **Implementación**

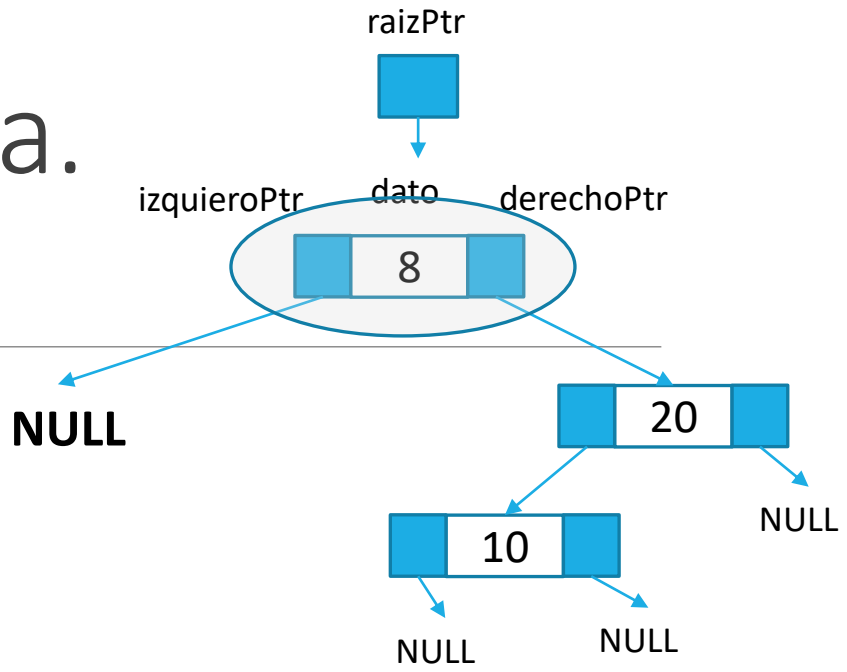


- void Arbol::podarArbol(Nodo *&nodoPtr)

```
void Arbol::podarArbol(Nodo *&nodoPtr)
{
    if(nodoPtr == NULL)
        return; // Termina

    // Se elimina el subnodoPtr izquierdo
    podarArbol(nodoPtr->izquierdoPtr);
    // Se elimina el subnodoPtr derecho
    podarArbol(nodoPtr->derechoPtr);
    // Se elimina el nodo actual
    delete nodoPtr;
    nodoPtr = NULL;
}
```

Pendiente 8

Árboles binarios de búsqueda. **Implementación**

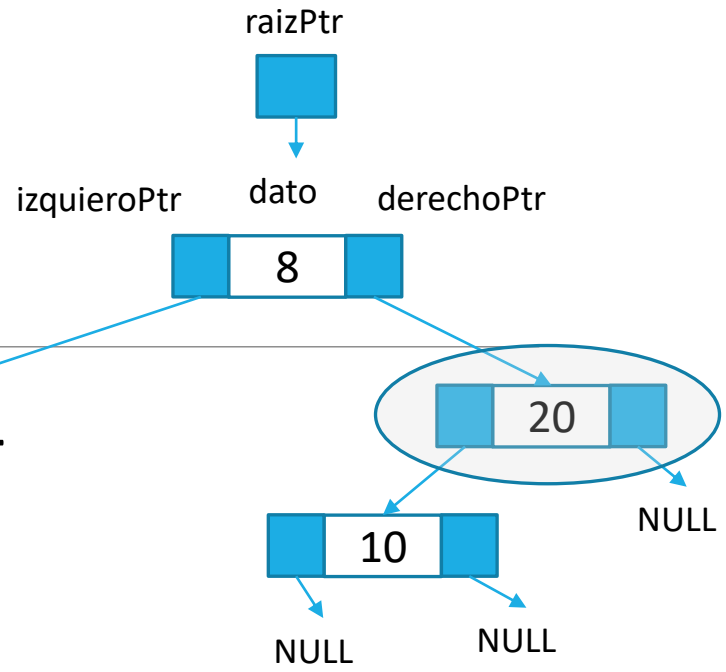


- void Arbol::podarArbol(Nodo *&nodoPtr)

```
void Arbol::podarArbol(Nodo *&nodoPtr)
{
    if(nodoPtr == NULL)
        return; // Termina

    // Se elimina el subnodoPtr izquierdo
    podarArbol(nodoPtr->izquierdoPtr);
    // Se elimina el subnodoPtr derecho
    podarArbol(nodoPtr->derechoPtr);
    // Se elimina el nodo actual
    delete nodoPtr;
    nodoPtr = NULL;
}
```

Pendiente 8

Árboles binarios de búsqueda. **Implementación**

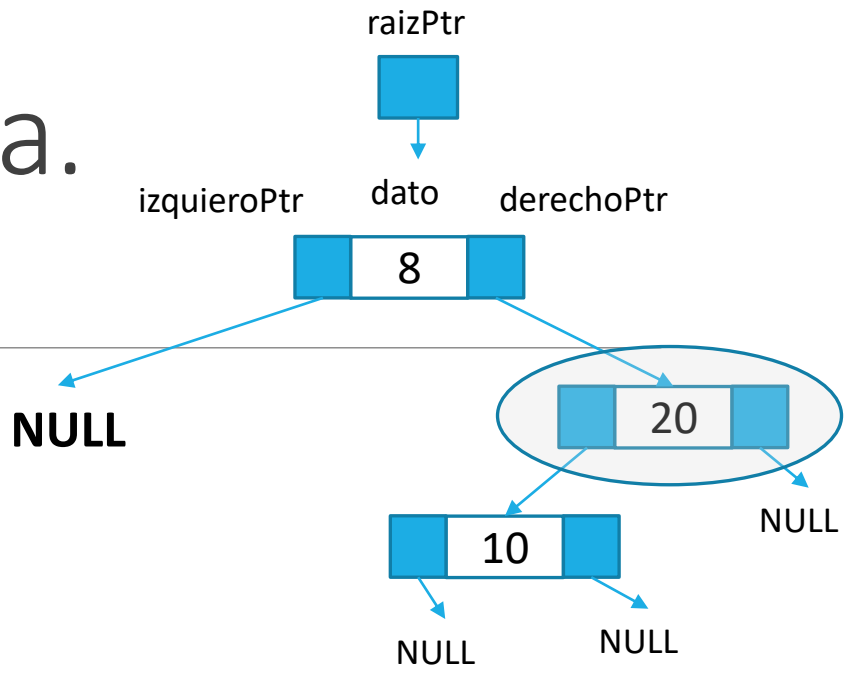


- void Arbol::podarArbol(Nodo *&nodoPtr)

```
void Arbol::podarArbol(Nodo *&nodoPtr)
{
    if(nodoPtr == NULL)
        return; // Termina

    // Se elimina el subnodoPtr izquierdo
    podarArbol(nodoPtr->izquierdoPtr);
    // Se elimina el subnodoPtr derecho
    podarArbol(nodoPtr->derechoPtr);
    // Se elimina el nodo actual
    delete nodoPtr;
    nodoPtr = NULL;
}
```

Pendiente 8

Árboles binarios de búsqueda. **Implementación**



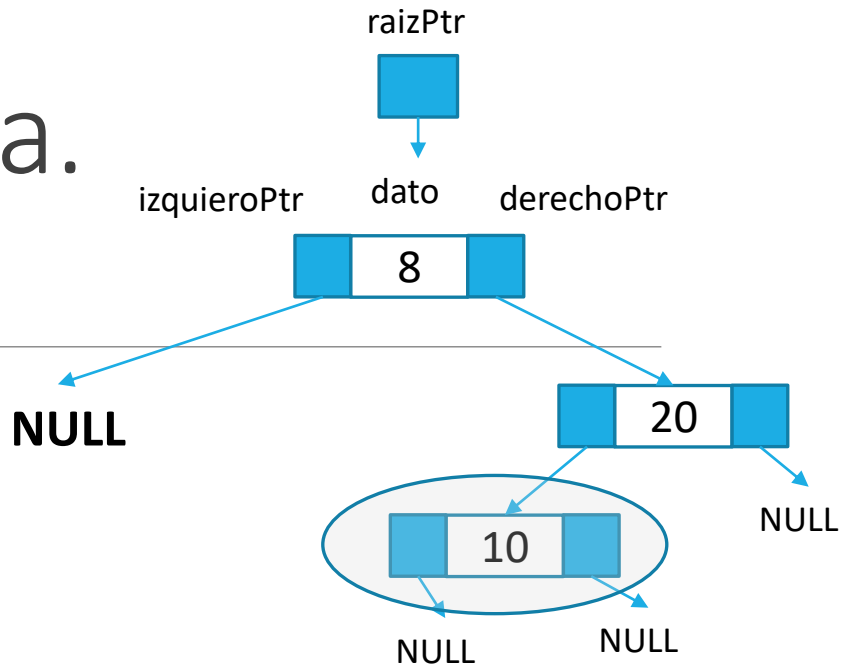
- void Arbol::podarArbol(Nodo *&nodoPtr)

```
void Arbol::podarArbol(Nodo *&nodoPtr)
{
    if(nodoPtr == NULL)
        return; // Termina

    // Se elimina el subnodoPtr izquiendo
    podarArbol(nodoPtr->izquierdoPtr);
    // Se elimina el subnodoPtr derecho
    podarArbol(nodoPtr->derechoPtr);
    // Se elimina el nodo actual
    delete nodoPtr;
    nodoPtr = NULL;
}
```

Pendiente 8

Pendiente 20

Árboles binarios de búsqueda. **Implementación**



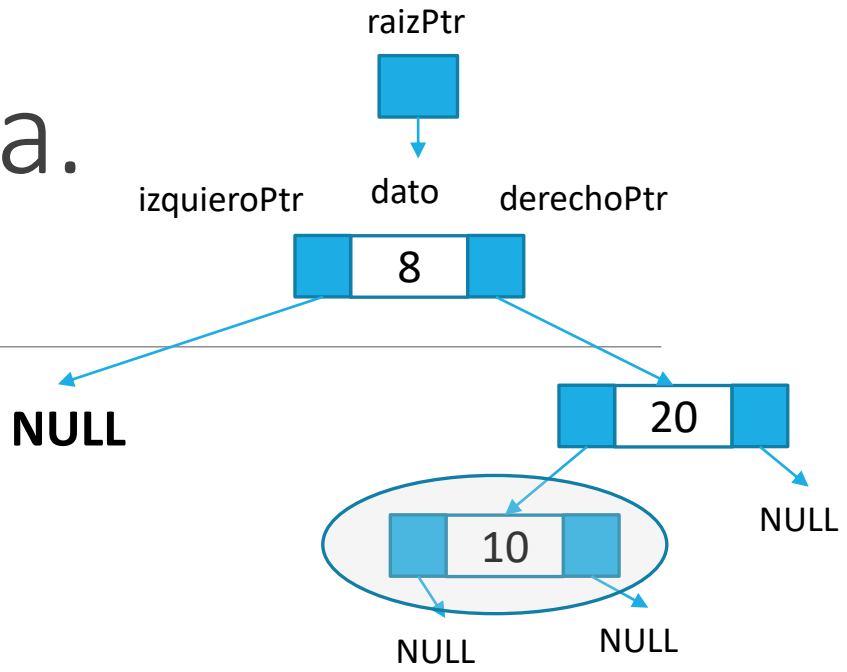
- void Arbol::podarArbol(Nodo *&nodoPtr)

```
void Arbol::podarArbol(Nodo *&nodoPtr)
{
    if(nodoPtr == NULL)
        return; // Termina

    // Se elimina el subnodoPtr izquierdo
    podarArbol(nodoPtr->izquierdoPtr);
    // Se elimina el subnodoPtr derecho
    podarArbol(nodoPtr->derechoPtr);
    // Se elimina el nodo actual
    delete nodoPtr;
    nodoPtr = NULL;
}
```

Pendiente 8

Pendiente 20

Árboles binarios de búsqueda. **Implementación**



- void Arbol::podarArbol(Nodo *&nodoPtr)

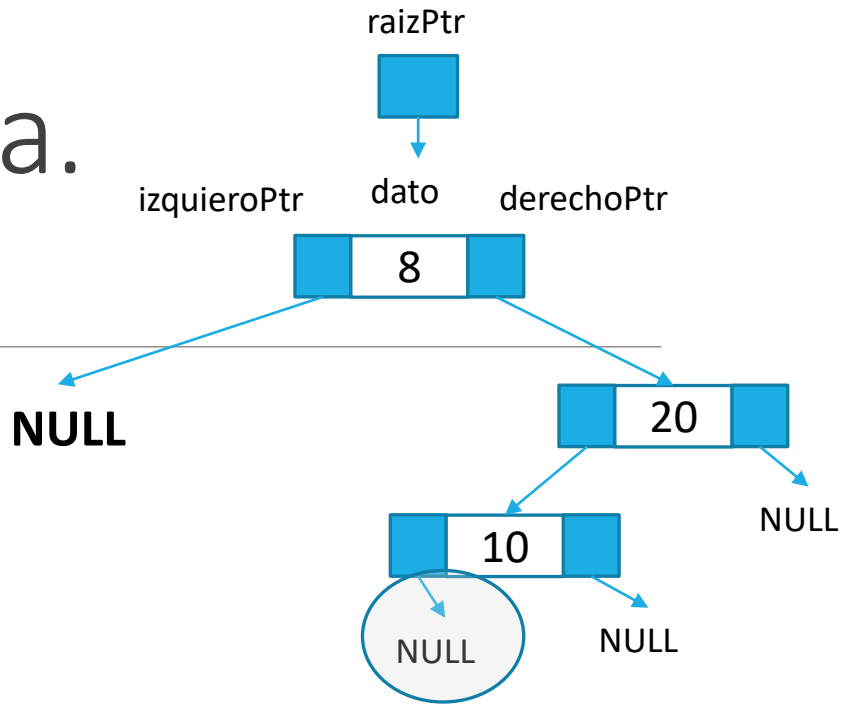
```
void Arbol::podarArbol(Nodo *&nodoPtr)
{
    if(nodoPtr == NULL)
        return; // Termina

    // Se elimina el subnodoPtr izquiendo
    podarArbol(nodoPtr->izquierdoPtr);
    // Se elimina el subnodoPtr derecho
    podarArbol(nodoPtr->derechoPtr);
    // Se elimina el nodo actual
    delete nodoPtr;
    nodoPtr = NULL;
}
```

Pendiente 8

Pendiente 20

Pendiente 10

Árboles binarios de búsqueda. **Implementación**



- void Arbol::podarArbol(Nodo *&nodoPtr)

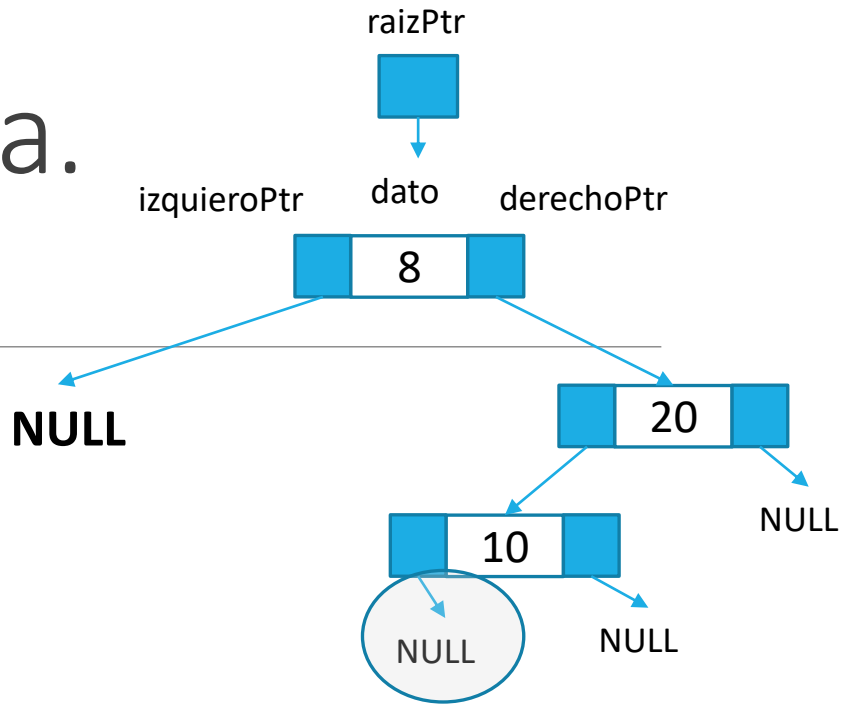
```
void Arbol::podarArbol(Nodo *&nodoPtr)
{
    if(nodoPtr == NULL)
        return; // Termina

    // Se elimina el subnodoPtr izquierdo
    podarArbol(nodoPtr->izquierdoPtr);
    // Se elimina el subnodoPtr derecho
    podarArbol(nodoPtr->derechoPtr);
    // Se elimina el nodo actual
    delete nodoPtr;
    nodoPtr = NULL;
}
```

Pendiente 8

Pendiente 20

Pendiente 10

Árboles binarios de búsqueda. **Implementación**



- void Arbol::podarArbol(Nodo *&nodoPtr)

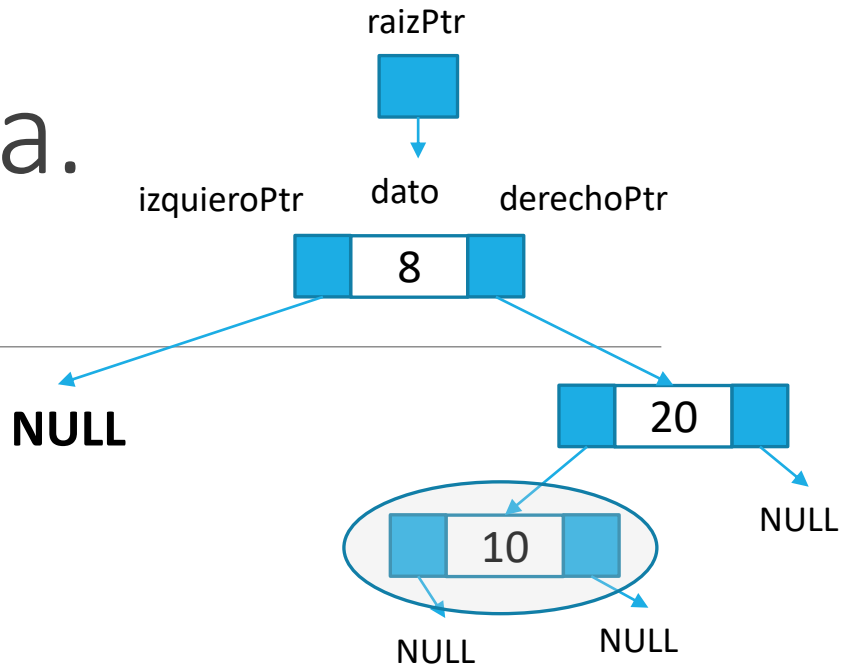
```
void Arbol::podarArbol(Nodo *&nodoPtr)
{
    if(nodoPtr == NULL)
        return; // Termina

    // Se elimina el subnodoPtr izquierdo
    podarArbol(nodoPtr->izquierdoPtr);
    // Se elimina el subnodoPtr derecho
    podarArbol(nodoPtr->derechoPtr);
    // Se elimina el nodo actual
    delete nodoPtr;
    nodoPtr = NULL;
}
```

Pendiente 8

Pendiente 20

Pendiente 10

Árboles binarios de búsqueda. **Implementación**



- void Arbol::podarArbol(Nodo *&nodoPtr)

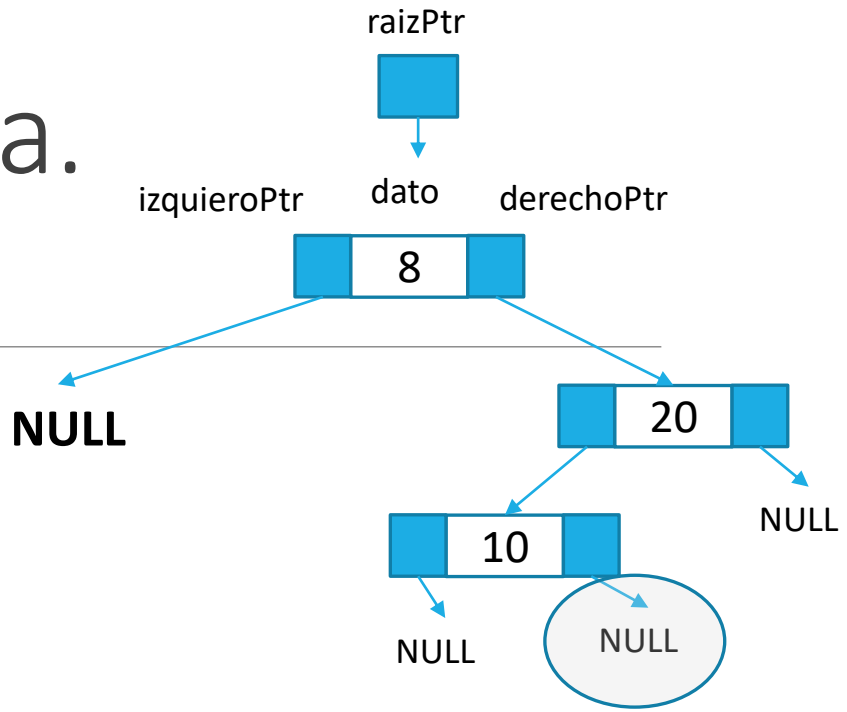
```
void Arbol::podarArbol(Nodo *&nodoPtr)
{
    if(nodoPtr == NULL)
        return; // Termina

    // Se elimina el subnodoPtr izquierdo
    podarArbol(nodoPtr->izquierdoPtr);
    // Se elimina el subnodoPtr derecho
    podarArbol(nodoPtr->derechoPtr);
    // Se elimina el nodo actual
    delete nodoPtr;
    nodoPtr = NULL;
}
```

Pendiente 8

Pendiente 20

Pendiente 10

Árboles binarios de búsqueda. **Implementación**



- void Arbol::podarArbol(Nodo *&nodoPtr)

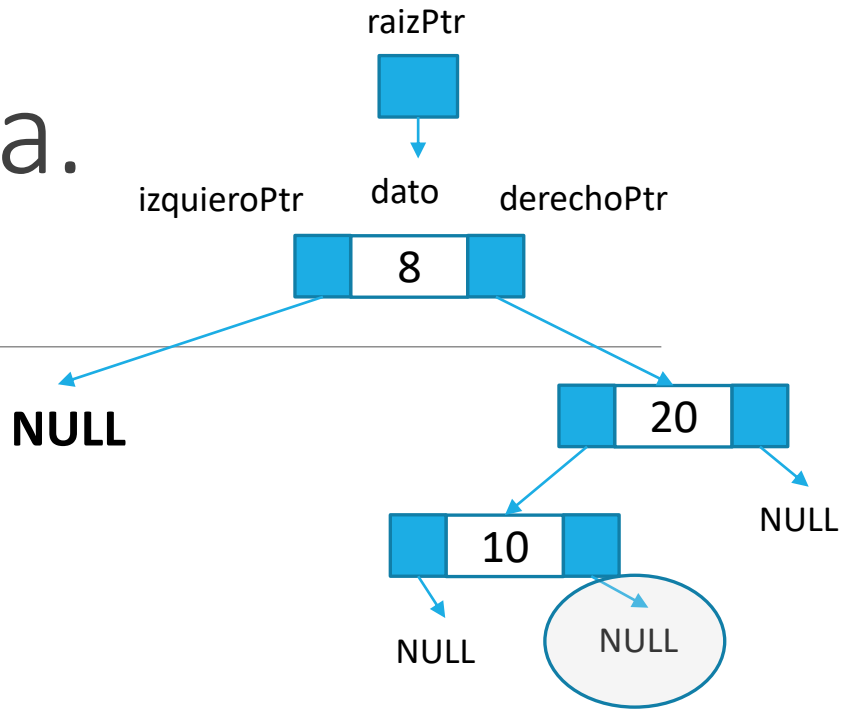
```
void Arbol::podarArbol(Nodo *&nodoPtr)
{
    if(nodoPtr == NULL)
        return; // Termina

    // Se elimina el subnodoPtr izquierdo
    podarArbol(nodoPtr->izquierdoPtr);
    // Se elimina el subnodoPtr derecho
    podarArbol(nodoPtr->derechoPtr);
    // Se elimina el nodo actual
    delete nodoPtr;
    nodoPtr = NULL;
}
```

Pendiente 8

Pendiente 20

Pendiente 10

Árboles binarios de búsqueda. **Implementación**



- void Arbol::podarArbol(Nodo *&nodoPtr)

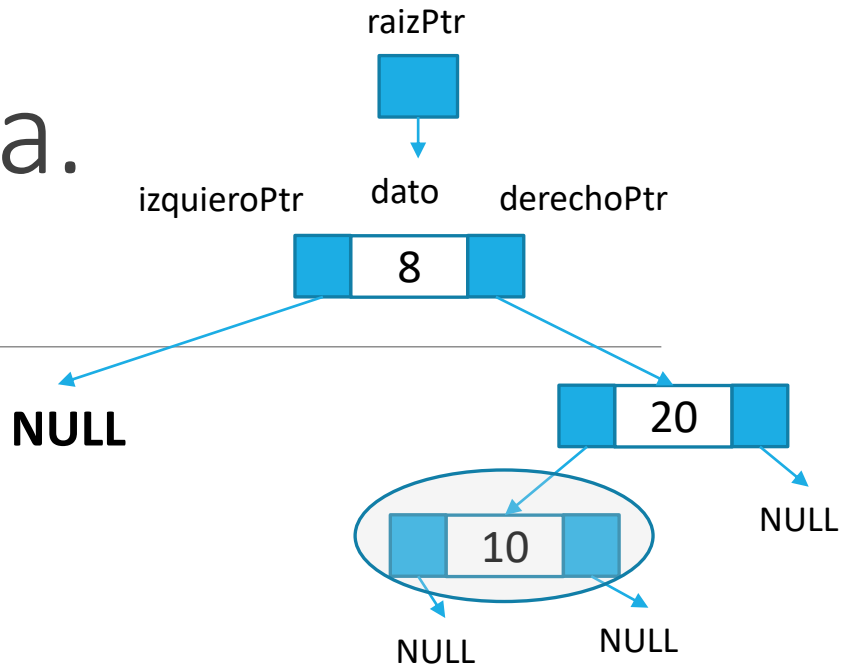
```
void Arbol::podarArbol(Nodo *&nodoPtr)
{
    if(nodoPtr == NULL)
        return; // Termina

    // Se elimina el subnodoPtr izquierdo
    podarArbol(nodoPtr->izquierdoPtr);
    // Se elimina el subnodoPtr derecho
    podarArbol(nodoPtr->derechoPtr);
    // Se elimina el nodo actual
    delete nodoPtr;
    nodoPtr = NULL;
}
```

Pendiente 8

Pendiente 20

Pendiente 10

Árboles binarios de búsqueda. **Implementación**



- void Arbol::podarArbol(Nodo *&nodoPtr)

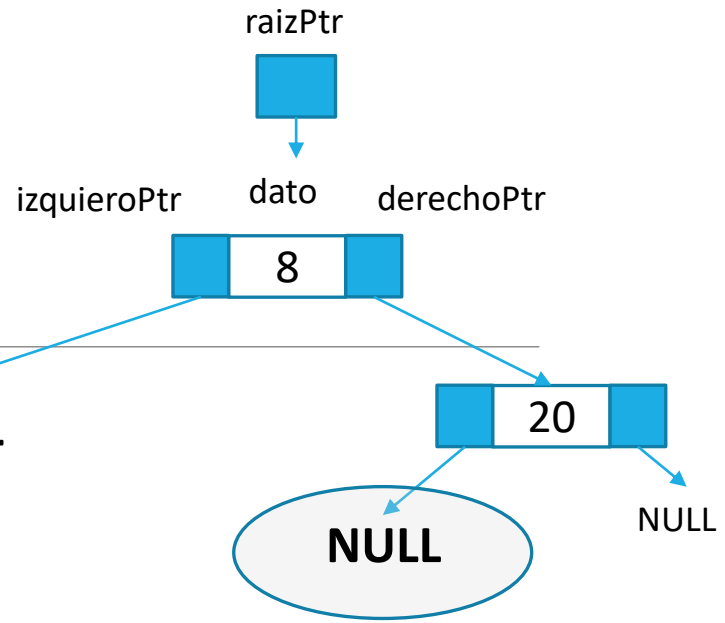
```
void Arbol::podarArbol(Nodo *&nodoPtr)
{
    if(nodoPtr == NULL)
        return; // Termina

    // Se elimina el subnodoPtr izquierdo
    podarArbol(nodoPtr->izquierdoPtr);
    // Se elimina el subnodoPtr derecho
    podarArbol(nodoPtr->derechoPtr);
    // Se elimina el nodo actual
    delete nodoPtr;
    nodoPtr = NULL;
}
```

Pendiente 8

Pendiente 20

Pendiente 10. Elimina nodo

Árboles binarios de búsqueda. **Implementación**



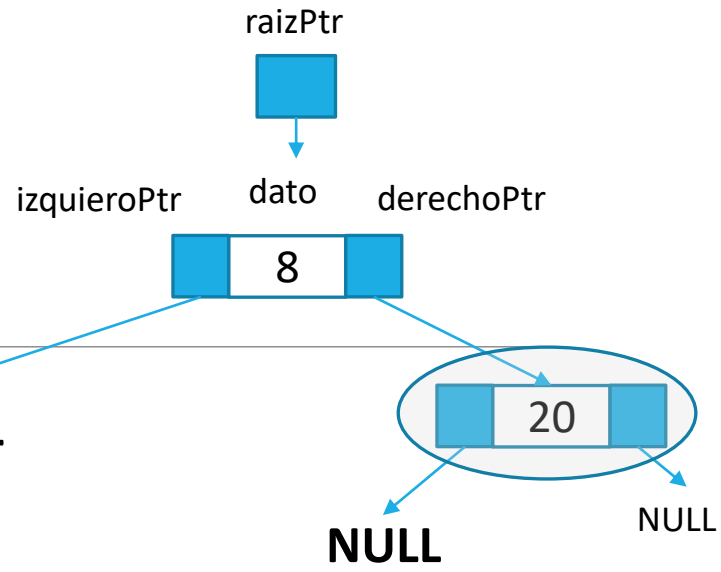
- void Arbol::podarArbol(Nodo *&nodoPtr)

```
void Arbol::podarArbol(Nodo *&nodoPtr)
{
    if(nodoPtr == NULL)
        return; // Termina

    // Se elimina el subnodoPtr izquierdo
    podarArbol(nodoPtr->izquierdoPtr);
    // Se elimina el subnodoPtr derecho
    podarArbol(nodoPtr->derechoPtr);
    // Se elimina el nodo actual
    delete nodoPtr;
    nodoPtr = NULL;
}
```

Pendiente 8

Pendiente 20

Árboles binarios de búsqueda. **Implementación**



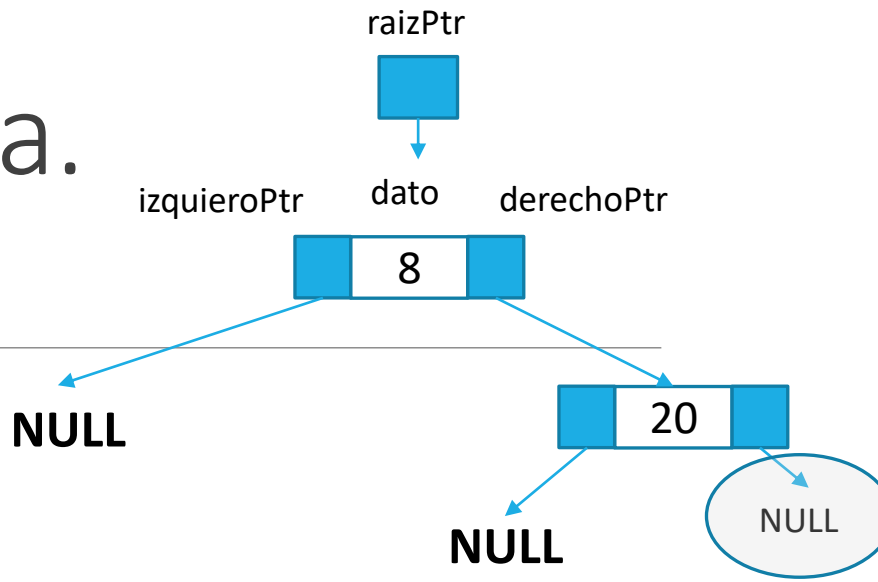
- void Arbol::podarArbol(Nodo *&nodoPtr)

```
void Arbol::podarArbol(Nodo *&nodoPtr)
{
    if(nodoPtr == NULL)
        return; // Termina

    // Se elimina el subnodoPtr izquierdo
    podarArbol(nodoPtr->izquierdoPtr);
    // Se elimina el subnodoPtr derecho
    podarArbol(nodoPtr->derechoPtr);
    // Se elimina el nodo actual
    delete nodoPtr;
    nodoPtr = NULL;
}
```

Pendiente 8

Pendiente 20

Árboles binarios de búsqueda. **Implementación**



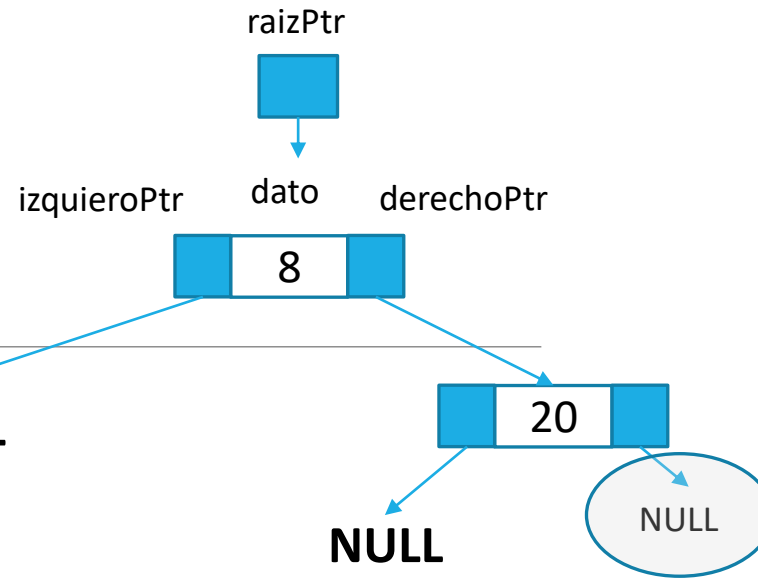
- void Arbol::podarArbol(Nodo *&nodoPtr)

```
void Arbol::podarArbol(Nodo *&nodoPtr)
{
    if(nodoPtr == NULL)
        return; // Termina

    // Se elimina el subnodoPtr izquierdo
    podarArbol(nodoPtr->izquierdoPtr);
    // Se elimina el subnodoPtr derecho
    podarArbol(nodoPtr->derechoPtr);
    // Se elimina el nodo actual
    delete nodoPtr;
    nodoPtr = NULL;
}
```

Pendiente 8

Pendiente 20

Árboles binarios de búsqueda. **Implementación**



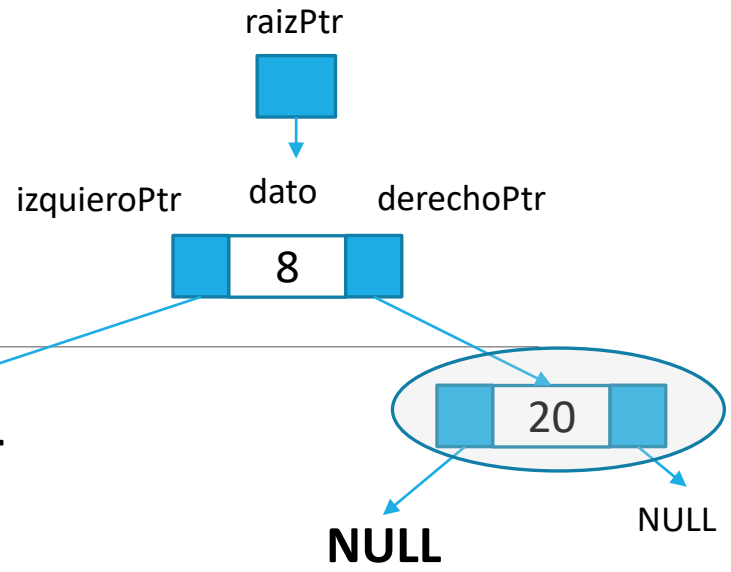
- void Arbol::podarArbol(Nodo *&nodoPtr)

```
void Arbol::podarArbol(Nodo *&nodoPtr)
{
    if(nodoPtr == NULL)
        return; // Termina

    // Se elimina el subnodoPtr izquierdo
    podarArbol(nodoPtr->izquierdoPtr);
    // Se elimina el subnodoPtr derecho
    podarArbol(nodoPtr->derechoPtr);
    // Se elimina el nodo actual
    delete nodoPtr;
    nodoPtr = NULL;
}
```

Pendiente 8

Pendiente 20

Árboles binarios de búsqueda. **Implementación**



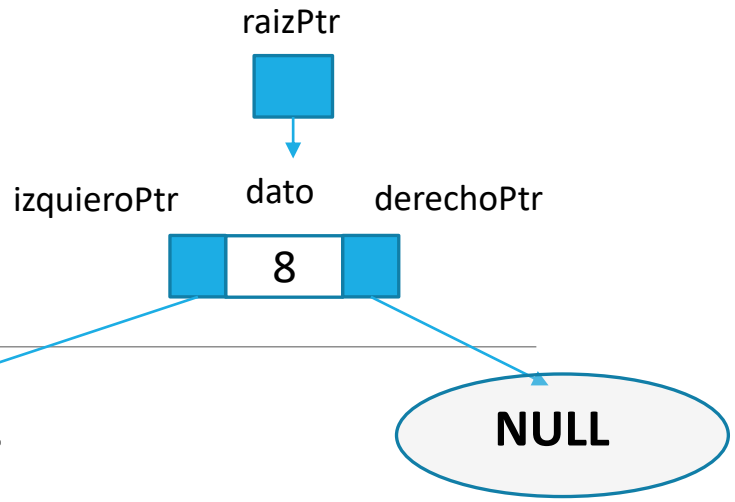
- void Arbol::podarArbol(Nodo *&nodoPtr)

```
void Arbol::podarArbol(Nodo *&nodoPtr)
{
    if(nodoPtr == NULL)
        return; // Termina

    // Se elimina el subnodoPtr izquierdo
    podarArbol(nodoPtr->izquierdoPtr);
    // Se elimina el subnodoPtr derecho
    podarArbol(nodoPtr->derechoPtr);
    // Se elimina el nodo actual
    delete nodoPtr;
    nodoPtr = NULL;
}
```

Pendiente 8

Pendiente 20. Elimina nodo.

Árboles binarios de búsqueda. **Implementación**

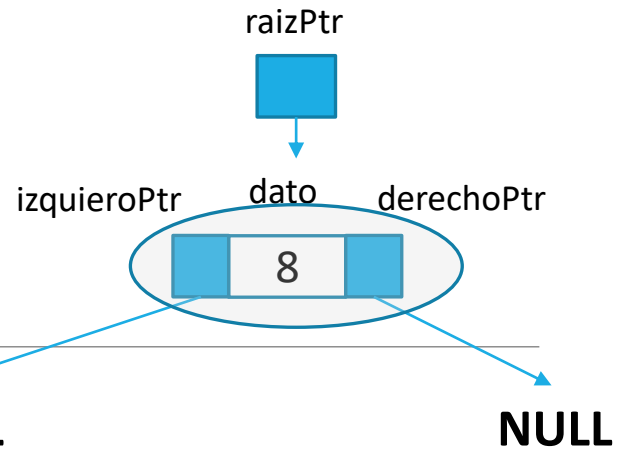


- void Arbol::podarArbol(Nodo *&nodoPtr)

```
void Arbol::podarArbol(Nodo *&nodoPtr)
{
    if(nodoPtr == NULL)
        return; // Termina

    // Se elimina el subnodoPtr izquierdo
    podarArbol(nodoPtr->izquierdoPtr);
    // Se elimina el subnodoPtr derecho
    podarArbol(nodoPtr->derechoPtr);
    // Se elimina el nodo actual
    delete nodoPtr;
    nodoPtr = NULL;
}
```

Pendiente 8

Árboles binarios de búsqueda. **Implementación**

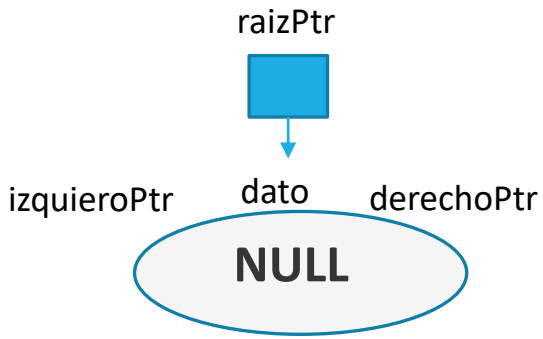


- void Arbol::podarArbol(Nodo *&nodoPtr)

```
void Arbol::podarArbol(Nodo *&nodoPtr)
{
    if(nodoPtr == NULL)
        return; // Termina

    // Se elimina el subnodoPtr izquierdo
    podarArbol(nodoPtr->izquierdoPtr);
    // Se elimina el subnodoPtr derecho
    podarArbol(nodoPtr->derechoPtr);
    // Se elimina el nodo actual
    delete nodoPtr;
    nodoPtr = NULL;
}
```

Pendiente 8. Elimina nodo.

Árboles binarios de búsqueda. **Implementación**



- void Arbol::podarArbol(Nodo *&nodoPtr)

```
void Arbol::podarArbol(Nodo *&nodoPtr)
{
    if(nodoPtr == NULL)
        return; // Termina

    // Se elimina el subnodoPtr izquierdo
    podarArbol(nodoPtr->izquierdoPtr);
    // Se elimina el subnodoPtr derecho
    podarArbol(nodoPtr->derechoPtr);
    // Se elimina el nodo actual
    delete nodoPtr;
    nodoPtr = NULL;
}
```

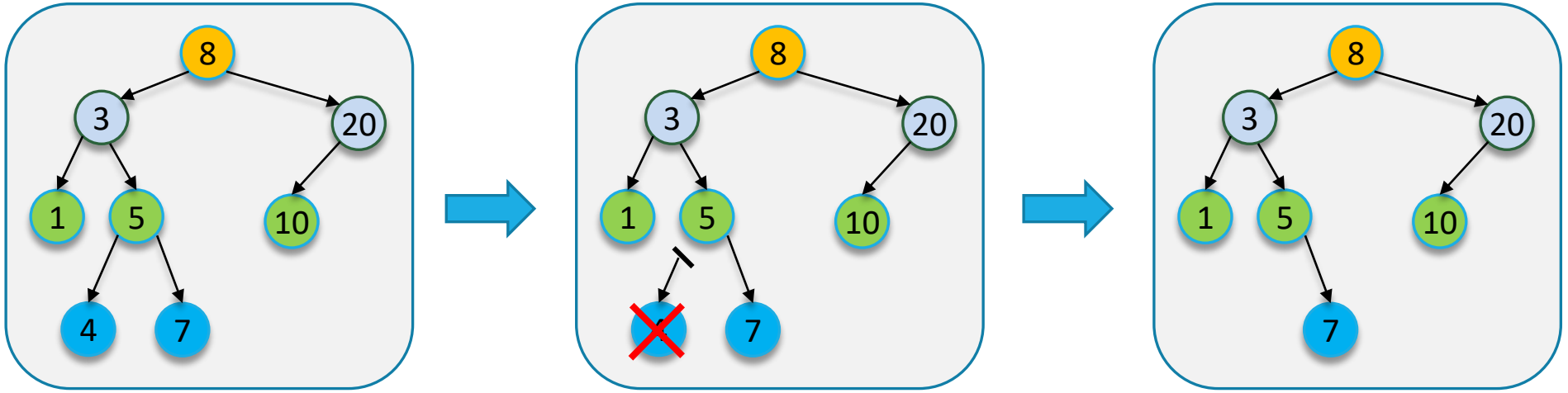
Árboles binarios de búsqueda.
Implementación- Eliminación de un nodo

La operación de eliminado se vuelve más compleja, dado que existen varios casos a tomar en cuenta:

- Eliminar un nodo sin hijos (nodo hoja)
- Eliminar un nodo con un subárbol hijo
- Eliminar un nodo con dos subárboles hijo

Árboles binarios de búsqueda. **Implementación.** Eliminar un nodo sin hijos

Se elimina el nodo y se establece a nulo el apuntador de su padre.

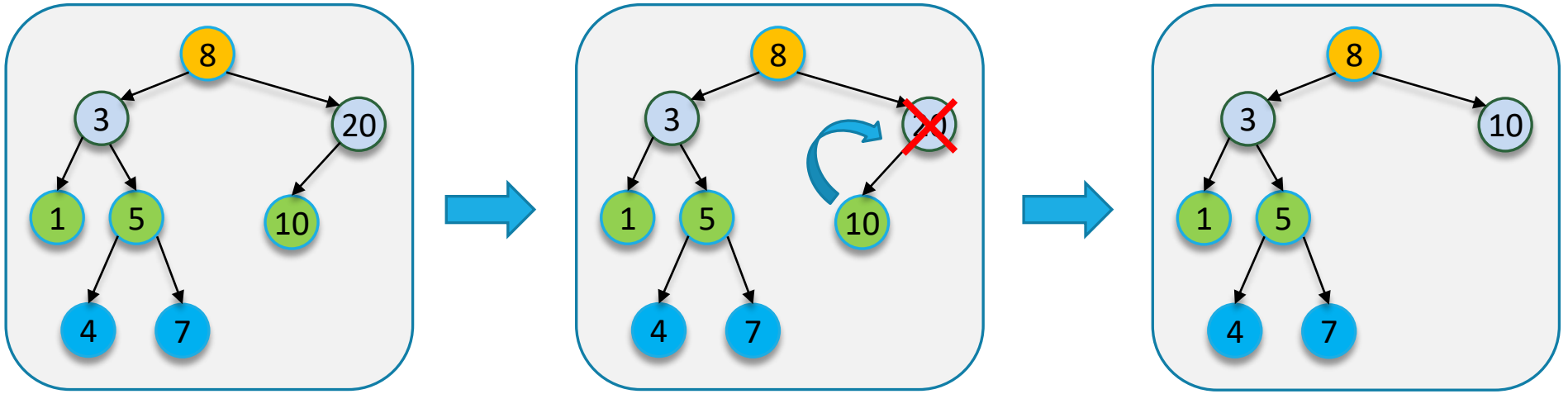


Eliminar el valor 4

El apuntador de su padre a NULL y eliminarlo

Árboles binarios de búsqueda. **Implementación.** Eliminar un nodo con un subárbol hijo

Se elimina el nodo y se asigna su subárbol hijo como subárbol de su padre.



Eliminar el valor 20

Subárbol hijo como subárbol padre y eliminar el nodo

Árboles binarios de búsqueda. **Implementación.** Eliminar un nodo con dos subárboles hijo

Se reemplaza el valor del nodo, ya sea por el de su predecesor o por el de su sucesor en **inOrden** (izquierdo, **raíz**, derecho) y posteriormente se borra el nodo.

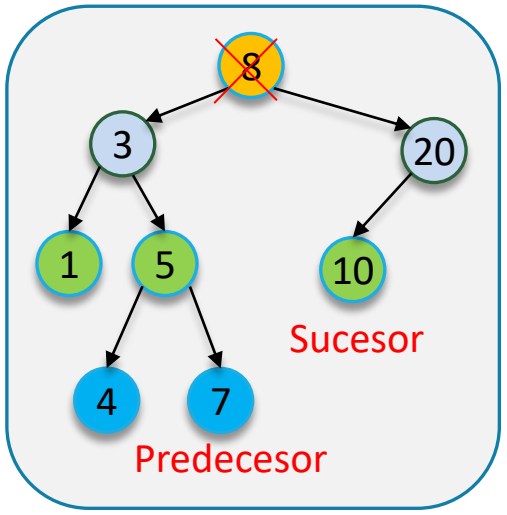
Su predecesor en **inOrden** será el nodo más a la derecha de su subárbol izquierdo (**mayor nodo del subárbol izquierdo**), y su sucesor el nodo más a la izquierda de su subárbol derecho (**menor nodo del subárbol derecho**).

Eliminar el valor 8

Predecesor

InOrder: 1 – 3 – 4 – 5 – 7 – **8** – 10 – 20

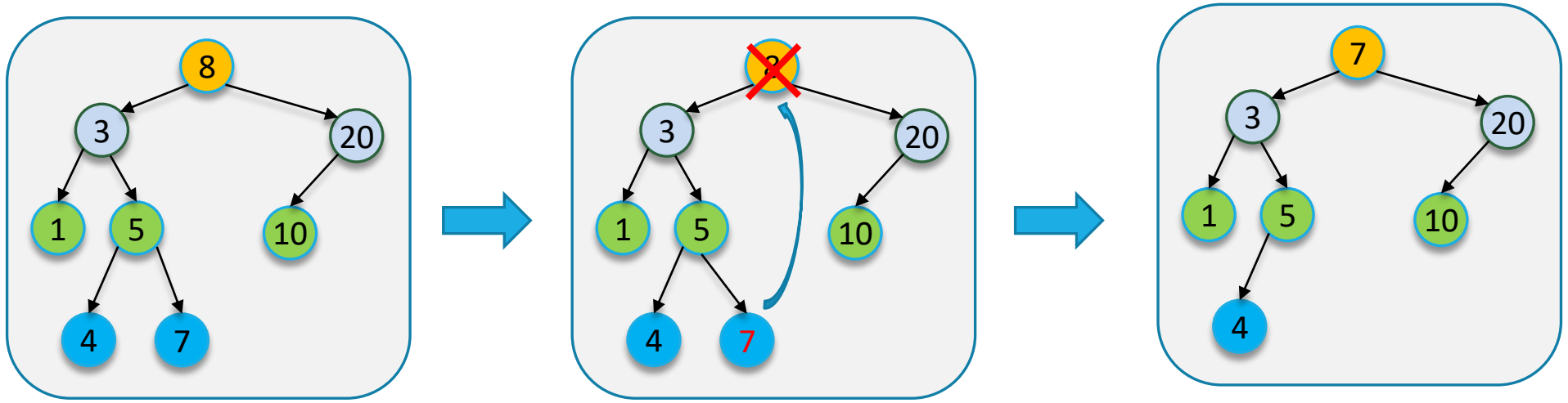
Sucesor



Eliminar un nodo con dos subárboles hijo. Sustituir por el **Predecesor**

Predecesor

InOrder: 1 – 3 – 4 – 5 – 7 – **8** – 10 – 20



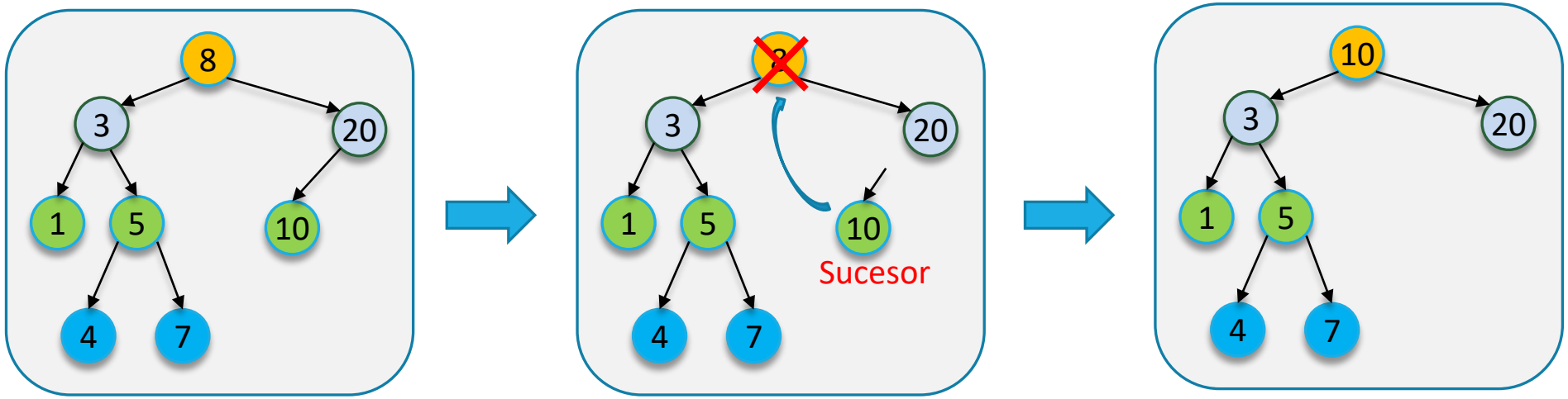
Eliminar el valor 8

Se reemplaza el predecesor y eliminarlo

Eliminar un nodo con dos subárboles hijo. Sustituir por el **Sucesor**

InOrder: 1 – 3 – 4 – 5 – 7 – **8** – 10 – 20

Sucesor



Eliminar el valor 8

Se reemplaza el sucesor y eliminarlo

Árboles binarios de búsqueda. **Implementación**

- Método Arbol::buscaMenor(Nodo *nodoPtr)

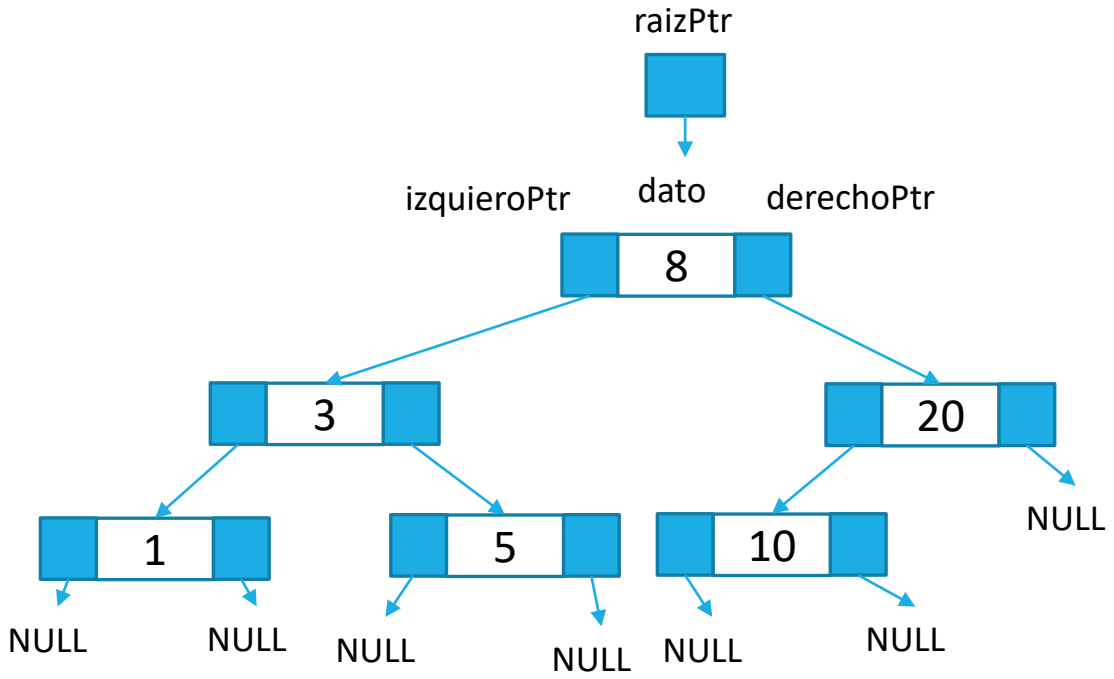
- Regresa el nodo con el dato menor del árbol.

```
Nodo *Arbol::buscaMenor(Nodo *nodoPtr)
{
    if(nodoPtr == NULL)
        return NULL;
    else if(nodoPtr->izquierdoPtr == NULL)
        return nodoPtr;
    else
        return buscaMenor(nodoPtr->izquierdoPtr);
}
```

Árboles binarios de búsqueda. **Implementación**

- Método Arbol::buscaMenor(Nodo *nodoPtr)

```
Nodo *Arbol::buscaMenor(Nodo *nodoPtr)
{
    if(nodoPtr == NULL)
        return NULL;
    else if(nodoPtr->izquierdoPtr == NULL)
        return nodoPtr;
    else
        return buscaMenor(nodoPtr->izquierdoPtr);
}
```



Árboles binarios de búsqueda. **Implementación**

- Método *Arbol::buscaMayor(Nodo* nodoPtr)

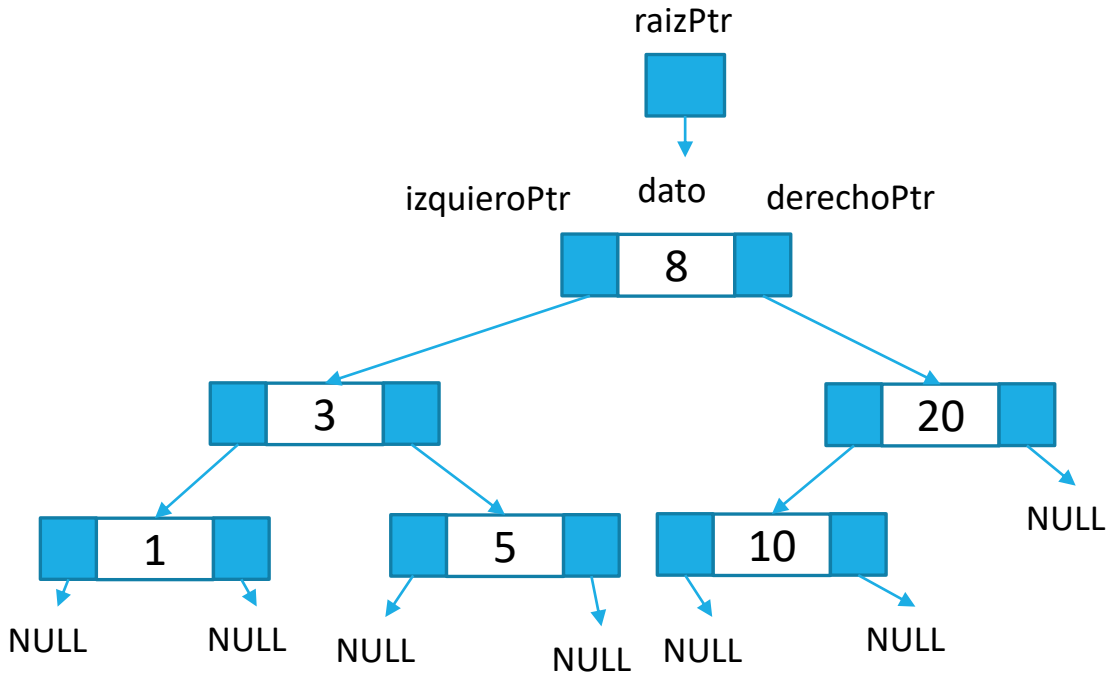
- Regresa el nodo con el dato mayor del árbol.

```
Nodo *Arbol::buscaMayor(Nodo* nodoPtr)
{
    if(nodoPtr == NULL)
        return NULL;
    else if(nodoPtr->derechoPtr == NULL)
        return nodoPtr;
    else
        return buscaMayor(nodoPtr->derechoPtr);
}
```

Árboles binarios de búsqueda. **Implementación**

- Método *Arbol::buscaMayor(Nodo* nodoPtr)

```
Nodo *Arbol::buscaMayor(Nodo* nodoPtr)
{
    if(nodoPtr == NULL)
        return NULL;
    else if(nodoPtr->derechoPtr == NULL)
        return nodoPtr;
    else
        return buscaMayor(nodoPtr->derechoPtr);
}
```



Árboles binarios de búsqueda.
Implementación

- Método Arbol::eliminarPredecesor(int x, Nodo *&nodoPtr)

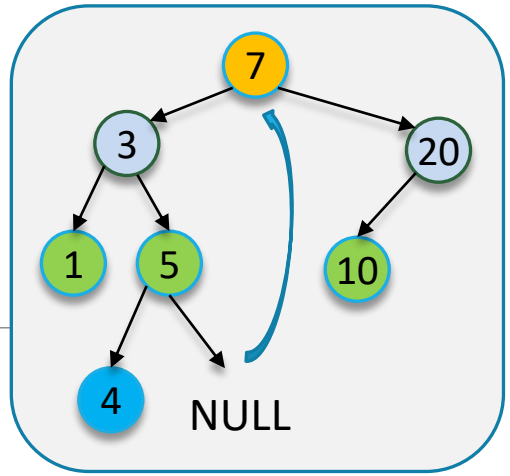
- Elimina el nodo del árbol que contiene el dato “x”; si dicho nodo tiene dos hijos, lo sustituye por el Predecesor en inOrden.

```
void Arbol::eliminarPredecesor(int x, Nodo *&nodoPtr)
{
    if(nodoPtr == NULL)
        return;
    else if(x < nodoPtr->dato)
        eliminarPredecesor(x, nodoPtr->izquierdoPtr);
    else if(x > nodoPtr->dato)
        eliminarPredecesor(x, nodoPtr->derechoPtr);
    else if(nodoPtr->izquierdoPtr && nodoPtr->derechoPtr)
    {
        // Tiene dos hijos
        Nodo* mayor = buscaMayor(nodoPtr->izquierdoPtr);
        nodoPtr->dato = mayor->dato;
        eliminarPredecesor(mayor->dato, nodoPtr->izquierdoPtr);
    }
    else
    {
        // Tiene un solo hijo o ninguno
        Nodo *temp = nodoPtr;
        if(nodoPtr->izquierdoPtr == NULL)
            nodoPtr = nodoPtr->derechoPtr;
        else if(nodoPtr->derechoPtr == NULL)
            nodoPtr = nodoPtr->izquierdoPtr;

        // Elimina el nodo
        delete temp;
    }
}
```

Árboles binarios de búsqueda. **Implementación.** Método Arbol::eliminarPredecesor(int x, Nodo *&nodoPtr)

```
void Arbol::eliminarPredecesor(int x, Nodo *&nodoPtr)
{
    if(nodoPtr == NULL)
        return;
    else if(x < nodoPtr->dato)
        eliminarPredecesor(x, nodoPtr->izquierdoPtr);
    else if(x > nodoPtr->dato)
        eliminarPredecesor(x, nodoPtr->derechoPtr);
    else if(nodoPtr->izquierdoPtr && nodoPtr->derechoPtr)
    {
        // Tiene dos hijos
        Nodo* mayor = buscaMayor(nodoPtr->izquierdoPtr);
        nodoPtr->dato = mayor->dato;
        eliminarPredecesor(mayor->dato, nodoPtr->izquierdoPtr);
    }
    else
    {
        // Tiene un solo hijo o ninguno
        Nodo *temp = nodoPtr;
        if(nodoPtr->izquierdoPtr == NULL)
            nodoPtr = nodoPtr->derechoPtr;
        else if(nodoPtr->derechoPtr == NULL)
            nodoPtr = nodoPtr->izquierdoPtr;

        // Elimina el nodo
        delete temp;
    }
}
```

Busca el nodo que contiene x



Si tiene nodo izquierdo y nodo derecho (tiene dos hijos)
eliminarPRedecesor(Mayor→7, nodoPtr→izquierdoPtr (Es el nodo a la izquierda de la raíz, que es el que tiene 3))

Tiene un nodo izquierdo o tiene un nodo derecho o ninguno.
Temp=7

Árboles binarios de búsqueda. **Implementación-** Eliminación de un nodo

Actividad:

Elabora el método void Arbol::eliminarSucesor(int x, Nodo *&nodoPtr). Toma en cuenta que es posible tener los tres casos analizados en el método anterior:

- Eliminar un nodo sin hijos (nodo hoja)
- Eliminar un nodo con un subárbol hijo
- Eliminar un nodo con dos subárboles hijo, sustituyendo dicho nodo con el sucesor (en inOrden), es decir, el nodo con el menor valor del subárbol derecho.

Árboles binarios de búsqueda. **Implementación**

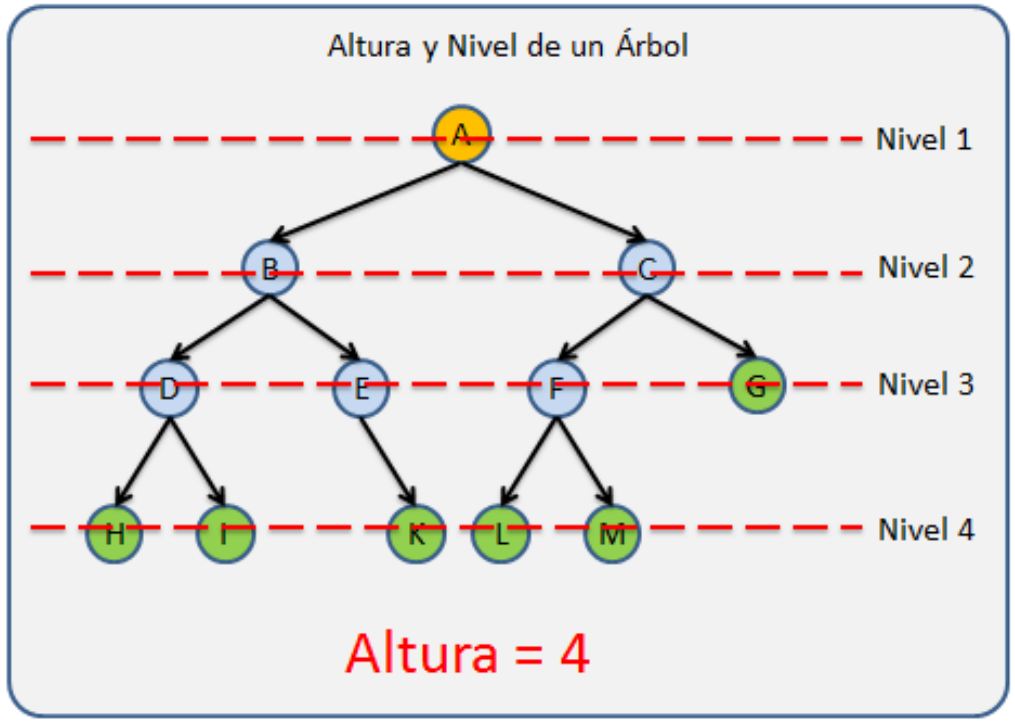
- Método Arbol::eliminarSucesor(int x, Nodo *&nodoPtr)

```
void Arbol::eliminarSucesor(int x, Nodo *&nodoPtr)
{
    if(nodoPtr == NULL)
        return;
    else if(x < nodoPtr->dato)
        eliminarSucesor(x, nodoPtr->izquierdoPtr);
    else if(x > nodoPtr->dato)
        eliminarSucesor(x, nodoPtr->derechoPtr);
    else if(nodoPtr->izquierdoPtr && nodoPtr->derechoPtr)
    {
        // Tiene dos hijos
        Nodo* menor = buscaMenor(nodoPtr->derechoPtr);
        nodoPtr->dato = menor->dato;
        eliminarSucesor(menor->dato, nodoPtr->derechoPtr);
    }
    else
    {
        // Tiene un solo hijo
        Nodo *temp = nodoPtr;
        if(nodoPtr->izquierdoPtr == NULL)
            nodoPtr = nodoPtr->derechoPtr;
        else if(nodoPtr->derechoPtr == NULL)
            nodoPtr = nodoPtr->izquierdoPtr;

        // Elimina el nodo hoja
        delete temp;
    }
}
```

En relación al tamaño del árbol

Nivel. El nivel de un nodo es su distancia a la raíz. Por lo tanto:

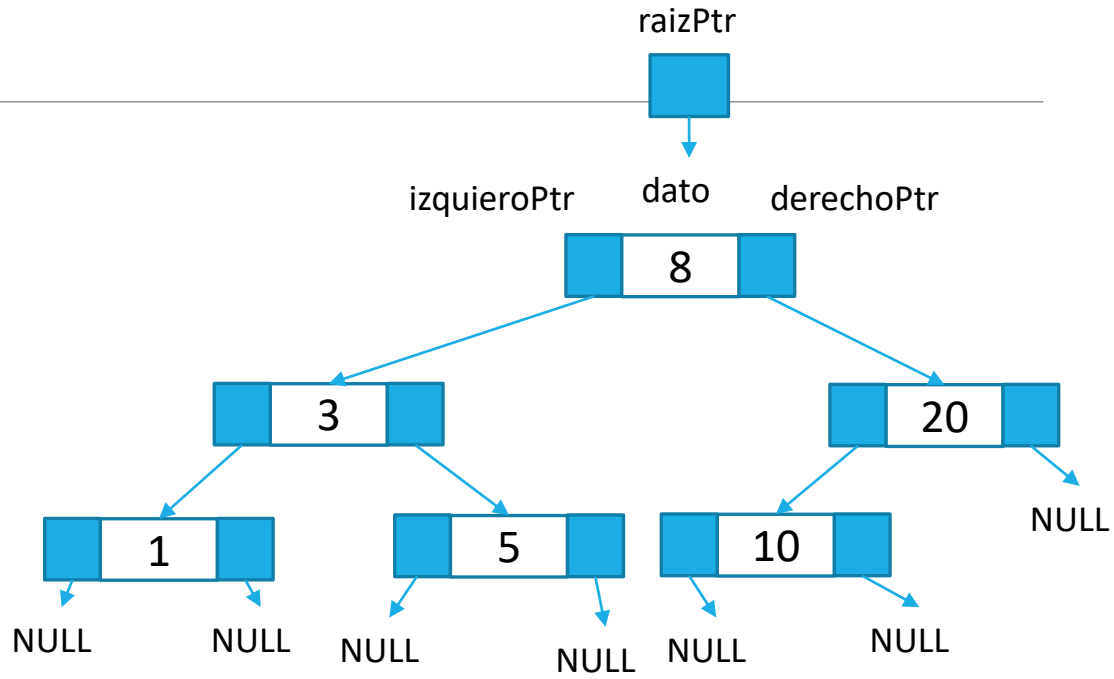
- Un *árbol* vacío tiene 0 niveles
- El nivel de la *raíz* es 1
- El nivel de cada nodo se calculado contando cuantos nodos existen sobre él, hasta llegar a la raíz + 1, y de forma inversa también se podría, contar cuantos nodos existen desde la *raíz* hasta el nodo buscado + 1.



Altura. Se le llama altura al número máximo de niveles de un *árbol*.

Árboles binarios de búsqueda. **Implementación**

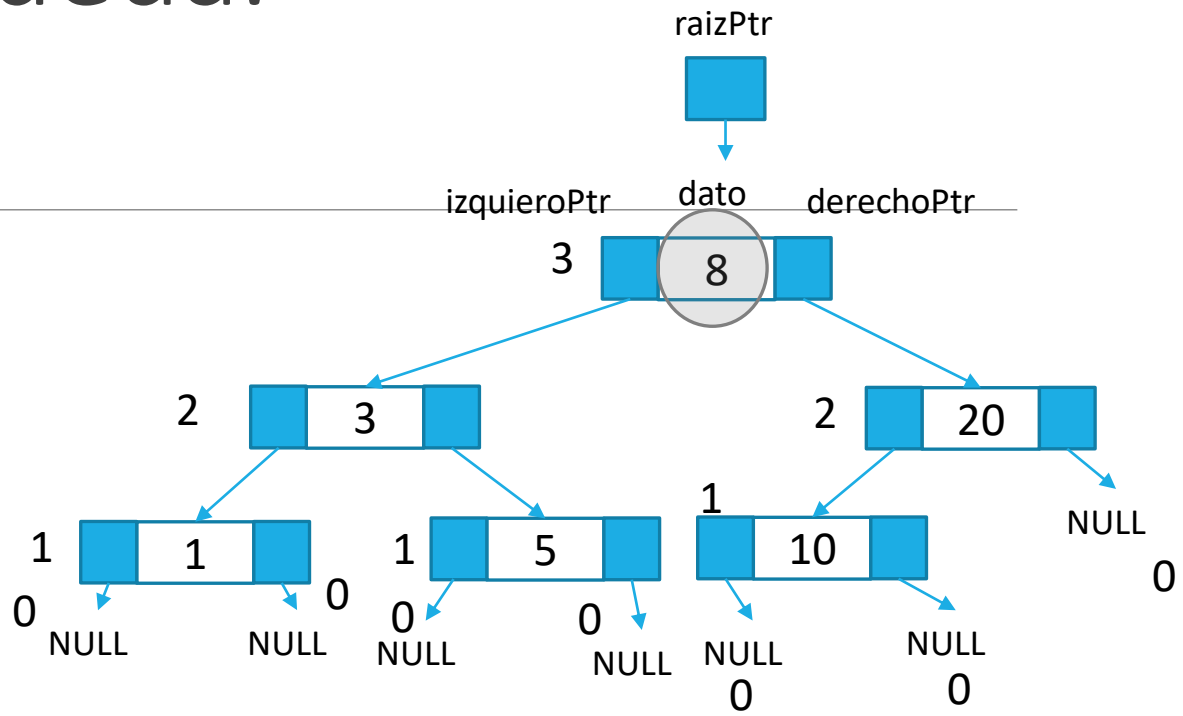
- Método Arbol::altura(Nodo *nodoPtr)
- Obtiene la altura (número máximo de niveles de un árbol.)




```
int Arbol::altura(Nodo *nodoPtr)
{
    if(nodoPtr == NULL)
        return 0;

    return (1 + max(altura(nodoPtr->izquierdoPtr),altura(nodoPtr->derechoPtr)));
}
```

Árboles binarios de búsqueda. **Implementación**

- Método Arbol::altura(Nodo *nodoPtr)




```
int Arbol::altura(Nodo *nodoPtr)
{
    if(nodoPtr == NULL)
        return 0;

    return (1 + max(altura(nodoPtr->izquierdoPtr),altura(nodoPtr->derechoPtr)));
}
```

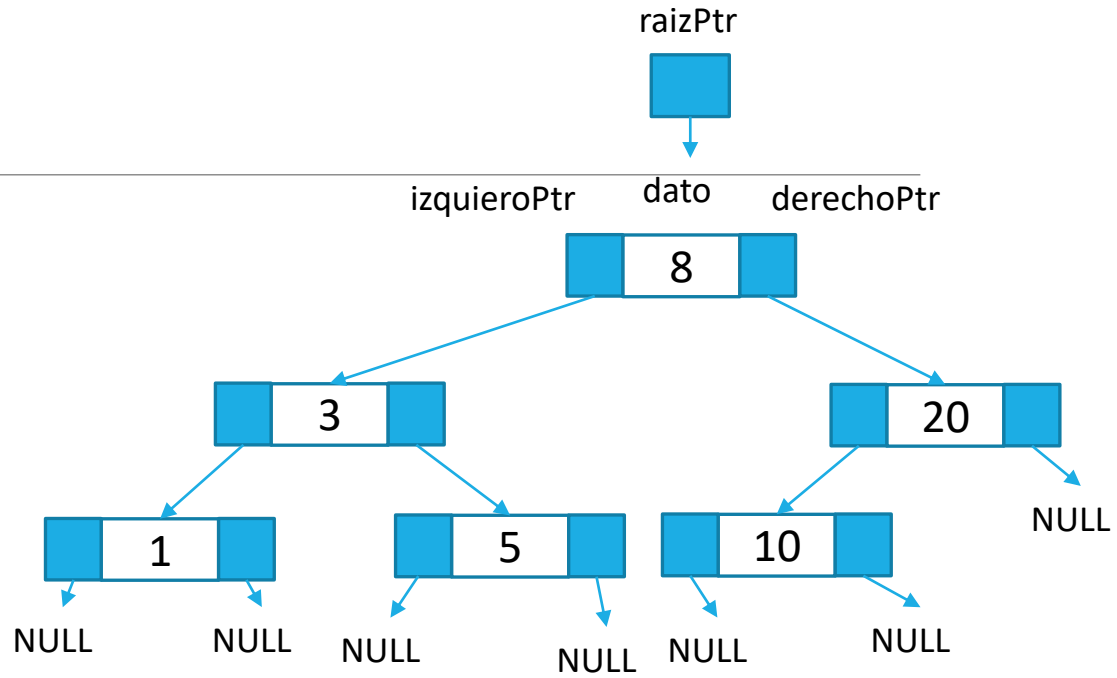
Árboles binarios de búsqueda. **Implementación**

- Método Arbol::contarHojas(Nodo *nodoPtr)
- Obtiene el número de nodos hoja de un árbol

```
int Arbol::contarHojas(Nodo *nodoPtr)
{
    if(nodoPtr == NULL)
        return 0;

    if(nodoPtr->derechoPtr==NULL && nodoPtr->izquierdoPtr==NULL)
        return 1;
    else
        return contarHojas(nodoPtr->izquierdoPtr) + contarHojas(nodoPtr->derechoPtr);
}
```

Árboles binarios de búsqueda. **Implementación**

- Actividad:
- Realiza la prueba de escritorio del método Arbol::contarHojas(Nodo *nodoPtr) que se muestra a continuación




```
int Arbol::contarHojas(Nodo *nodoPtr)
{
    if(nodoPtr == NULL)
        return 0;

    if(nodoPtr->derechoPtr==NULL && nodoPtr->izquierdoPtr==NULL)
        return 1;
    else
        return contarHojas(nodoPtr->izquierdoPtr) + contarHojas(nodoPtr->derechoPtr);
}
```

Árboles binarios de búsqueda. **Implementación**

- Actividad:
- Elabora el Método int Arbol::contarNodos(Nodo *nodoPtr)
- Obtiene el número de nodos de un árbol.

- Consiste en contar de manera recursiva los nodos del árbol izquierdo, contar los nodos del árbol derecho y sumar el resultado. Y a este resultado sumar 1 para contar la raíz.

Árboles binarios de búsqueda. **Implementación**

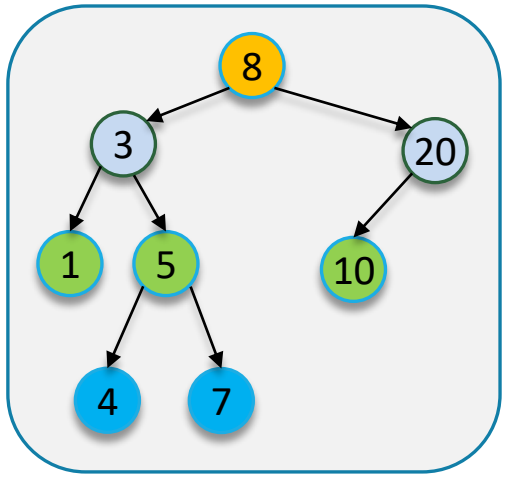
- Actividad:
- Elabora el Método int Arbol::contarNodos(Nodo *nodoPtr)
- Obtiene el número de nodos de un árbol.

```
int Arbol::contarNodos(Nodo *nodoPtr)
{
    if(nodoPtr == NULL)
        return 0;

    return 1 + contarNodos(nodoPtr->izquierdoPtr) + contarNodos(nodoPtr->derechoPtr);
}
```

Árboles binarios de búsqueda. **Implementación**

- Método void Arbol::recorridoNiveles(Nodo *nodoPtr)
- Escribe en pantalla los nodos del árbol, recorriéndolo por niveles. El recorrido por orden de nivel de un árbol binario visita los nodos del árbol fila por fila, empezando con el nivel del nodo raíz. En cada nivel del árbol, los nodos se visitan de izquierda a derecha.
- Es decir, para el siguiente árbol el recorrido por niveles es:

8, 3, 20, 1, 5, 10, 4, 7



Árboles binarios de búsqueda. **Implementación**

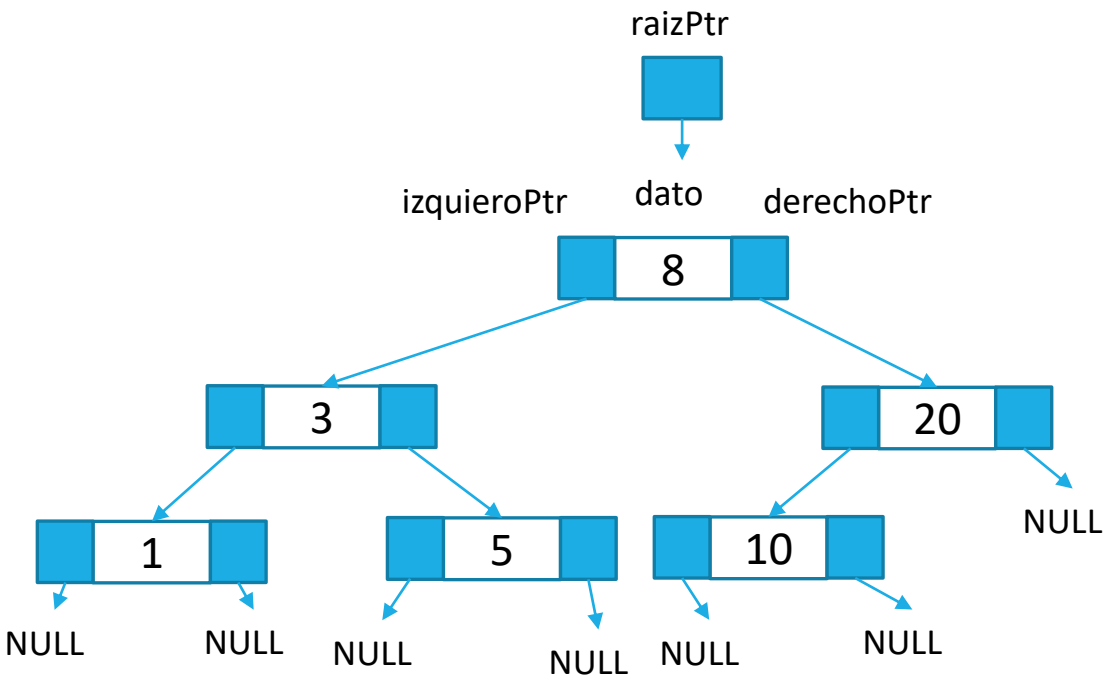
- Método void Arbol::recorridoNiveles(Nodo *nodoPtr)

```
void Arbol::recorridoNiveles(Nodo *nodoPtr)
{
    Nodo *nodoAuxiliar;
    queue<Nodo*> colaAuxiliar;
    colaAuxiliar.push(nodoPtr);

    while (! colaAuxiliar.empty() )
    {
        nodoAuxiliar = colaAuxiliar.front();
        cout << nodoAuxiliar->dato << " - ";
        colaAuxiliar.pop();

        if(nodoAuxiliar->izquierdoPtr!=NULL)
            colaAuxiliar.push(nodoAuxiliar->izquierdoPtr);
        if(nodoAuxiliar->derechoPtr!=NULL)
            colaAuxiliar.push(nodoAuxiliar->derechoPtr);
    }
}
```

Árboles binarios de búsqueda. **Implementación**

- Método void Arbol::recorridoNiveles(Nodo *nodoPtr)

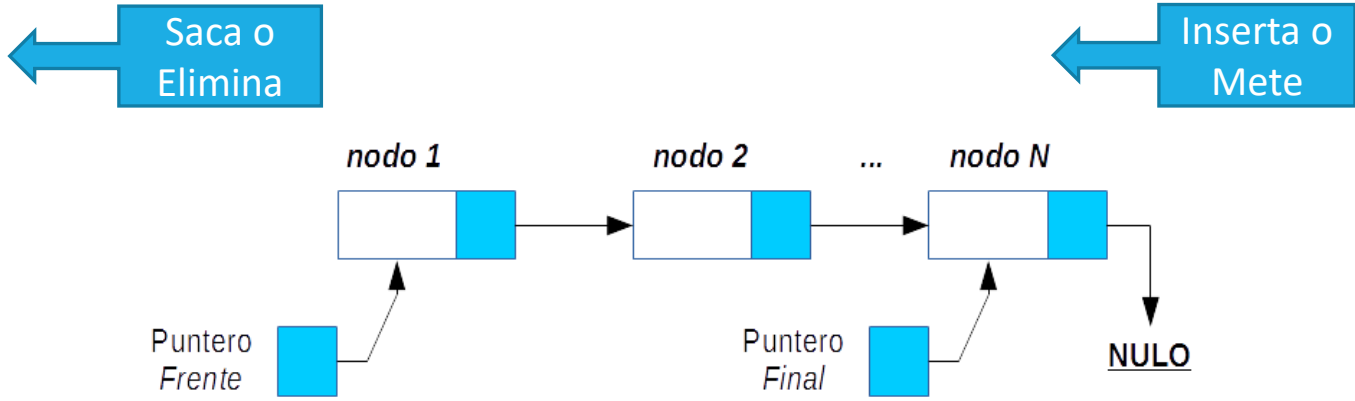
```
void Arbol::recorridoNiveles(Nodo *nodoPtr)
{
    Nodo *nodoAuxiliar;
    queue<Nodo*> colaAuxiliar;
    colaAuxiliar.push(nodoPtr);

    while (! colaAuxiliar.empty() )
    {
        nodoAuxiliar = colaAuxiliar.front();
        cout << nodoAuxiliar->dato << " - ";
        colaAuxiliar.pop();

        if(nodoAuxiliar->izquierdoPtr!=NULL)
            colaAuxiliar.push(nodoAuxiliar->izquierdoPtr);
        if(nodoAuxiliar->derechoPtr!=NULL)
            colaAuxiliar.push(nodoAuxiliar->derechoPtr);
    }
}
```



Árboles binarios de búsqueda. **Implementación**

- Método void Arbol::recorridoNiveles(Nodo *nodoPtr)

```
void Arbol::recorridoNiveles(Nodo *nodoPtr)
{
    Nodo *nodoAuxiliar;
    queue<Nodo*> colaAuxiliar;
    colaAuxiliar.push(nodoPtr);

    while (! colaAuxiliar.empty() )
    {
        nodoAuxiliar = colaAuxiliar.front();
        cout << nodoAuxiliar->dato << " - ";
        colaAuxiliar.pop();

        if(nodoAuxiliar->izquierdoPtr!=NULL)
            colaAuxiliar.push(nodoAuxiliar->izquierdoPtr);
        if(nodoAuxiliar->derechoPtr!=NULL)
            colaAuxiliar.push(nodoAuxiliar->derechoPtr);
    }
}
```



Utiliza una cola para almacenar los nodos por nivel y escribirlos.

Agregar la librería:

#include <queue>

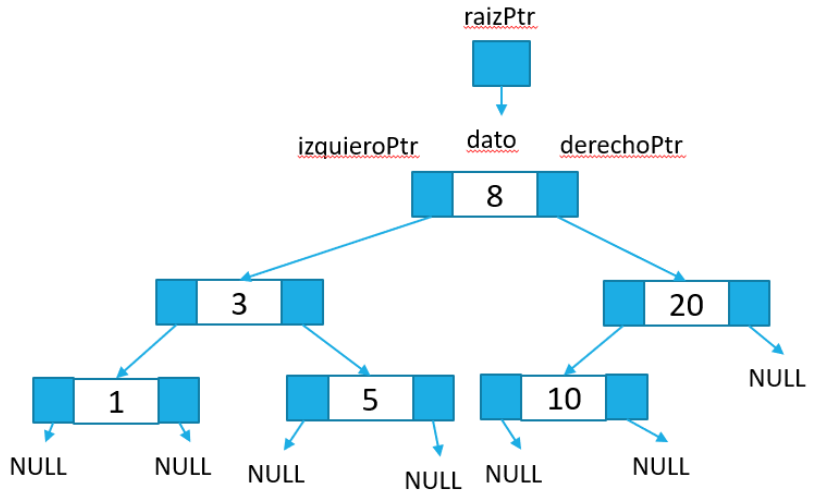
Push – Mete

Pop – Saca

Front- Obtiene el nodo al frente

Árboles binarios de búsqueda. **Implementación**

- Método void Arbol::recorridoNiveles(Nodo *nodoPtr)

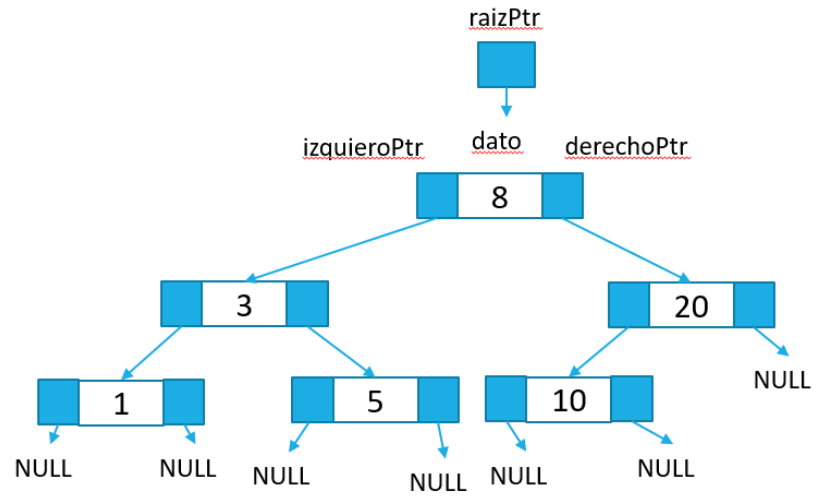



```
void Arbol::recorridoNiveles(Nodo *nodoPtr)
{
    Nodo *nodoAuxiliar;
    queue<Nodo*> colaAuxiliar;
    colaAuxiliar.push(nodoPtr);

    while (! colaAuxiliar.empty() )
    {
        nodoAuxiliar = colaAuxiliar.front();
        cout << nodoAuxiliar->dato << " - ";
        colaAuxiliar.pop();

        if(nodoAuxiliar->izquierdoPtr!=NULL)
            colaAuxiliar.push(nodoAuxiliar->izquierdoPtr);
        if(nodoAuxiliar->derechoPtr!=NULL)
            colaAuxiliar.push(nodoAuxiliar->derechoPtr);
    }
}
```

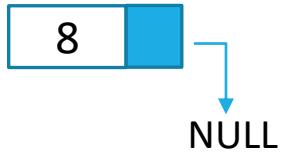
i. Crea una variable de tipo nodo llamada nodoAuxiliar.
ii. Crea una cola que contendrá elementos de tipo nodo, llamada colaAuxiliar.
iii. Mete el primer nodo, 8, (la raíz) a la cola.

Árboles binarios de búsqueda.
Implementación

- Método void Arbol::recorridoNiveles(Nodo *nodoPtr)



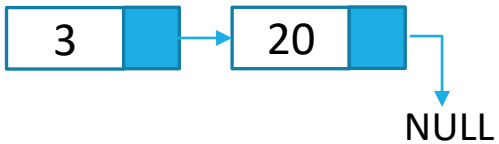

```
void Arbol::recorridoNiveles(Nodo *nodoPtr)
{
    Nodo *nodoAuxiliar;
    queue<Nodo*> colaAuxiliar;
    colaAuxiliar.push(nodoPtr);

    while (! colaAuxiliar.empty() )
    {
        nodoAuxiliar = colaAuxiliar.front();
        cout << nodoAuxiliar->dato << " - ";
        colaAuxiliar.pop();

        if(nodoAuxiliar->izquierdoPtr!=NULL)
            colaAuxiliar.push(nodoAuxiliar->izquierdoPtr);
        if(nodoAuxiliar->derechoPtr!=NULL)
            colaAuxiliar.push(nodoAuxiliar->derechoPtr);
    }
}
```

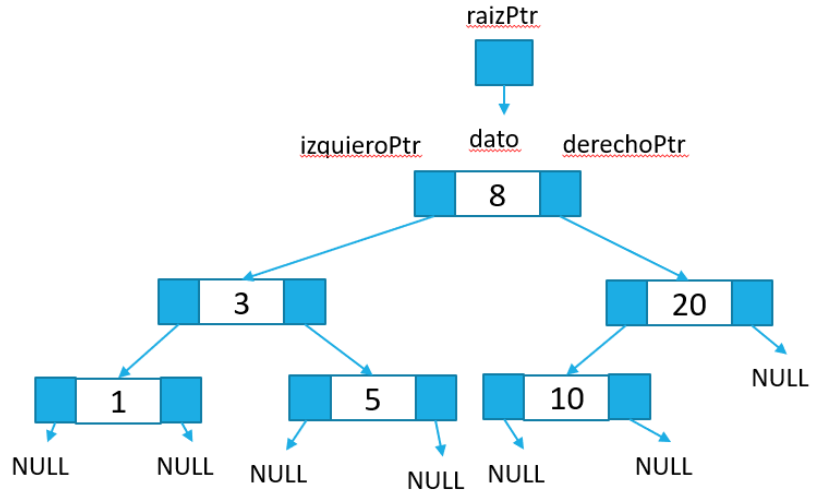


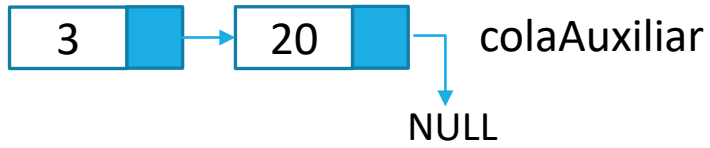
colaAuxiliar

i. Mientras la colaAuxiliar no sea vacía:
   i. nodoAuxiliar = 8
   ii. Escribe 8 –
   iii. Saca el 8
   iv. ¿izquierdoPtr de 8 es diferente de nulo?
      i. Mete el 3 a la cola

   v. ¿derechoPtr de 8 es diferente de nulo?
      i. Mete el 20 a la cola

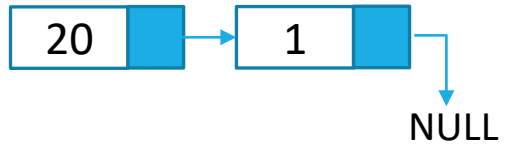




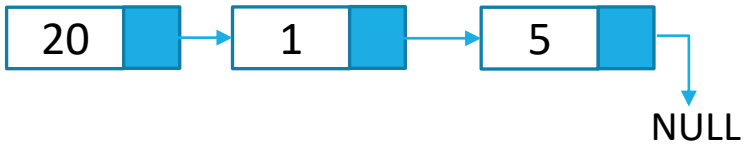


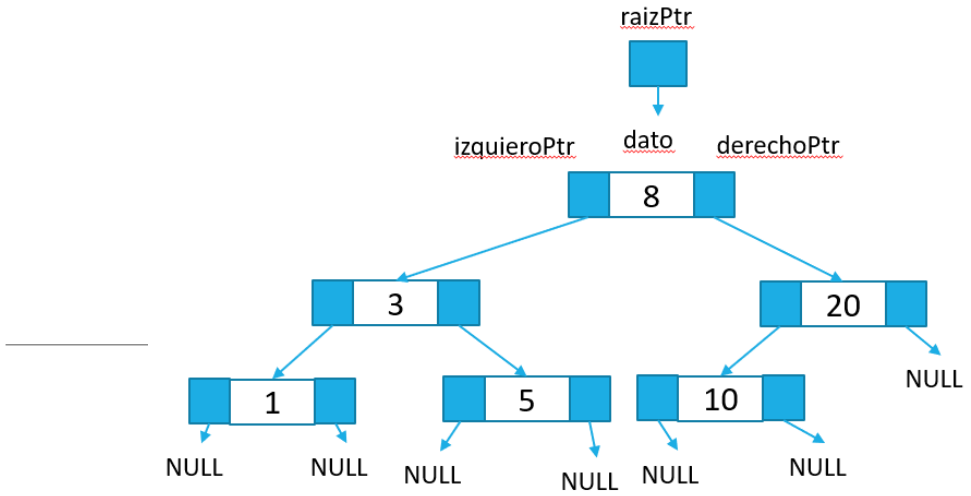

```
void Arbol::recorridoNiveles(Nodo *nodoPtr)
{
    Nodo *nodoAuxiliar;
    queue<Nodo*> colaAuxiliar;
    colaAuxiliar.push(nodoPtr);

    while (! colaAuxiliar.empty() )
    {
        nodoAuxiliar = colaAuxiliar.front();
        cout << nodoAuxiliar->dato << " - ";
        colaAuxiliar.pop();

        if(nodoAuxiliar->izquierdoPtr!=NULL)
            colaAuxiliar.push(nodoAuxiliar->izquierdoPtr);
        if(nodoAuxiliar->derechoPtr!=NULL)
            colaAuxiliar.push(nodoAuxiliar->derechoPtr);
    }
}
```

i. Mientras la colaAuxiliar no sea vacía:
   i. nodoAuxiliar = 3
   ii. Escribe 8 – 3 –
   iii. Saca el 3
   iv. ¿izquierdoPtr de 3 es diferente de nulo?
      i. Mete el 1 a la cola

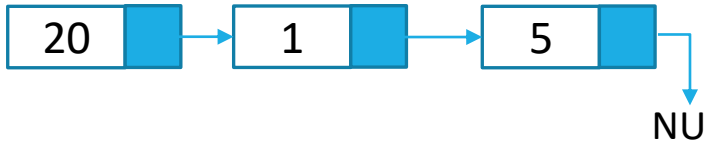


   v. ¿derechoPtr de 3 es diferente de nulo?
      i. Mete el 5 a la cola



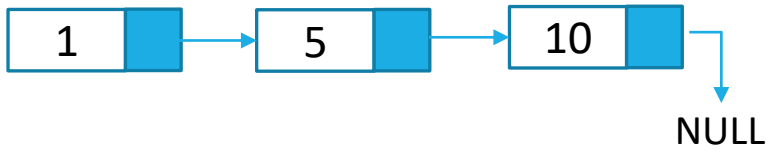


colaAuxiliar

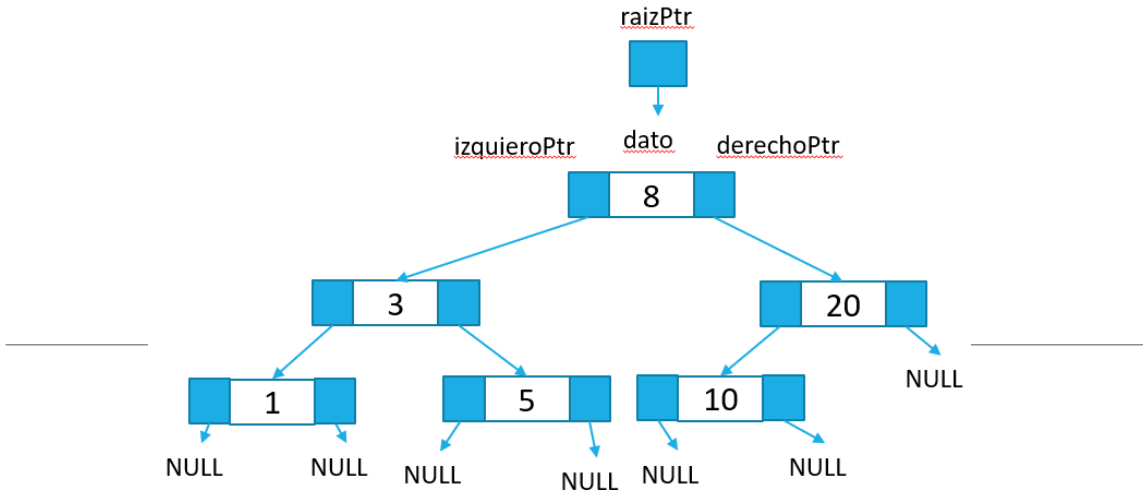



```
void Arbol::recorridoNiveles(Nodo *nodoPtr)
{
    Nodo *nodoAuxiliar;
    queue<Nodo*> colaAuxiliar;
    colaAuxiliar.push(nodoPtr);

    while (! colaAuxiliar.empty() )
    {
        nodoAuxiliar = colaAuxiliar.front();
        cout << nodoAuxiliar->dato << " - ";
        colaAuxiliar.pop();

        if(nodoAuxiliar->izquierdoPtr!=NULL)
            colaAuxiliar.push(nodoAuxiliar->izquierdoPtr);
        if(nodoAuxiliar->derechoPtr!=NULL)
            colaAuxiliar.push(nodoAuxiliar->derechoPtr);
    }
}
```

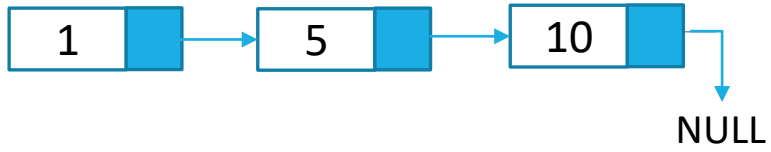
i. Mientras la colaAuxiliar no sea vacía:
   i. nodoAuxiliar = 20
   ii. Escribe 8 – 3 – 20 –
   iii. Saca el 20
   iv. ¿izquierdoPtr de 20 es diferente de nulo?
      i. Mete el 10 a la cola



   v. ¿derechoPtr de 20 es diferente de nulo?
      i. No. Es igual a nulo.



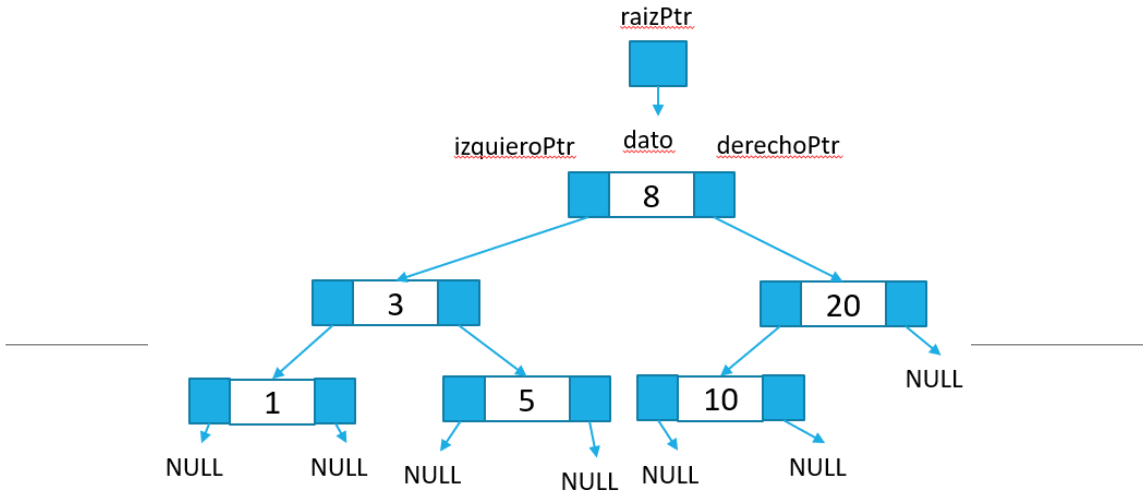
colaAuxiliar



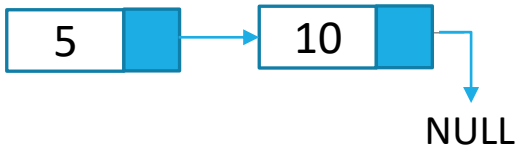

```
void Arbol::recorridoNiveles(Nodo *nodoPtr)
{
    Nodo *nodoAuxiliar;
    queue<Nodo*> colaAuxiliar;
    colaAuxiliar.push(nodoPtr);

    while (! colaAuxiliar.empty() )
    {
        nodoAuxiliar = colaAuxiliar.front();
        cout << nodoAuxiliar->dato << " - ";
        colaAuxiliar.pop();

        if(nodoAuxiliar->izquierdoPtr!=NULL)
            colaAuxiliar.push(nodoAuxiliar->izquierdoPtr);
        if(nodoAuxiliar->derechoPtr!=NULL)
            colaAuxiliar.push(nodoAuxiliar->derechoPtr);
    }
}
```

i. Mientras la colaAuxiliar no sea vacía:
   i. nodoAuxiliar = 1
   ii. Escribe 8 – 3 – 20 – 1 –
   iii. Saca el 1
   iv. ¿izquierdoPtr de 1 es diferente de nulo?
      i. No. Es igual a nulo.

   v. ¿derechoPtr de 1 es diferente de nulo?
      i. No. Es igual a nulo.



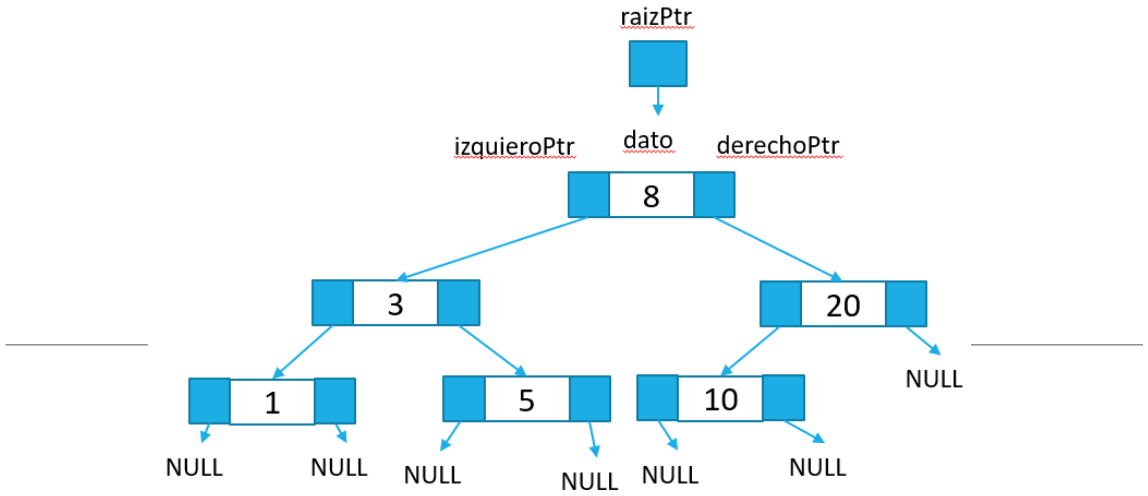
colaAuxiliar



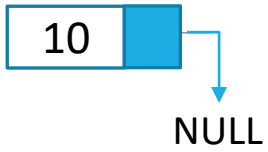

```
void Arbol::recorridoNiveles(Nodo *nodoPtr)
{
    Nodo *nodoAuxiliar;
    queue<Nodo*> colaAuxiliar;
    colaAuxiliar.push(nodoPtr);

    while (! colaAuxiliar.empty() )
    {
        nodoAuxiliar = colaAuxiliar.front();
        cout << nodoAuxiliar->dato << " - ";
        colaAuxiliar.pop();

        if(nodoAuxiliar->izquierdoPtr!=NULL)
            colaAuxiliar.push(nodoAuxiliar->izquierdoPtr);
        if(nodoAuxiliar->derechoPtr!=NULL)
            colaAuxiliar.push(nodoAuxiliar->derechoPtr);
    }
}
```

i. Mientras la colaAuxiliar no sea vacía:
   i. nodoAuxiliar = 5
   ii. Escribe 8 – 3 – 20 – 1 – 5 –
   iii. Saca el 5
   iv. ¿izquierdoPtr de 5 es diferente de nulo?
      i. No. Es igual a nulo.

   v. ¿derechoPtr de 5 es diferente de nulo?
      i. No. Es igual a nulo.



colaAuxiliar




```
void Arbol::recorridoNiveles(Nodo *nodoPtr)
{
    Nodo *nodoAuxiliar;
    queue<Nodo*> colaAuxiliar;
    colaAuxiliar.push(nodoPtr);

    while (! colaAuxiliar.empty() )
    {
        nodoAuxiliar = colaAuxiliar.front();
        cout << nodoAuxiliar->dato << " - ";
        colaAuxiliar.pop();

        if(nodoAuxiliar->izquierdoPtr!=NULL)
            colaAuxiliar.push(nodoAuxiliar->izquierdoPtr);
        if(nodoAuxiliar->derechoPtr!=NULL)
            colaAuxiliar.push(nodoAuxiliar->derechoPtr);
    }
}
```

i. Mientras la colaAuxiliar no sea vacía:
   i. nodoAuxiliar = 10
   ii. Escribe 8 – 3 – 20 – 1 – 5 – 10 –
   iii. Saca el 10
   iv. ¿izquierdoPtr de 10 es diferente de nulo?
      i. No. Es igual a nulo.

   v. ¿derechoPtr de 10 es diferente de nulo?
      i. No. Es igual a nulo.

Árboles binarios de búsqueda. **Implementación**

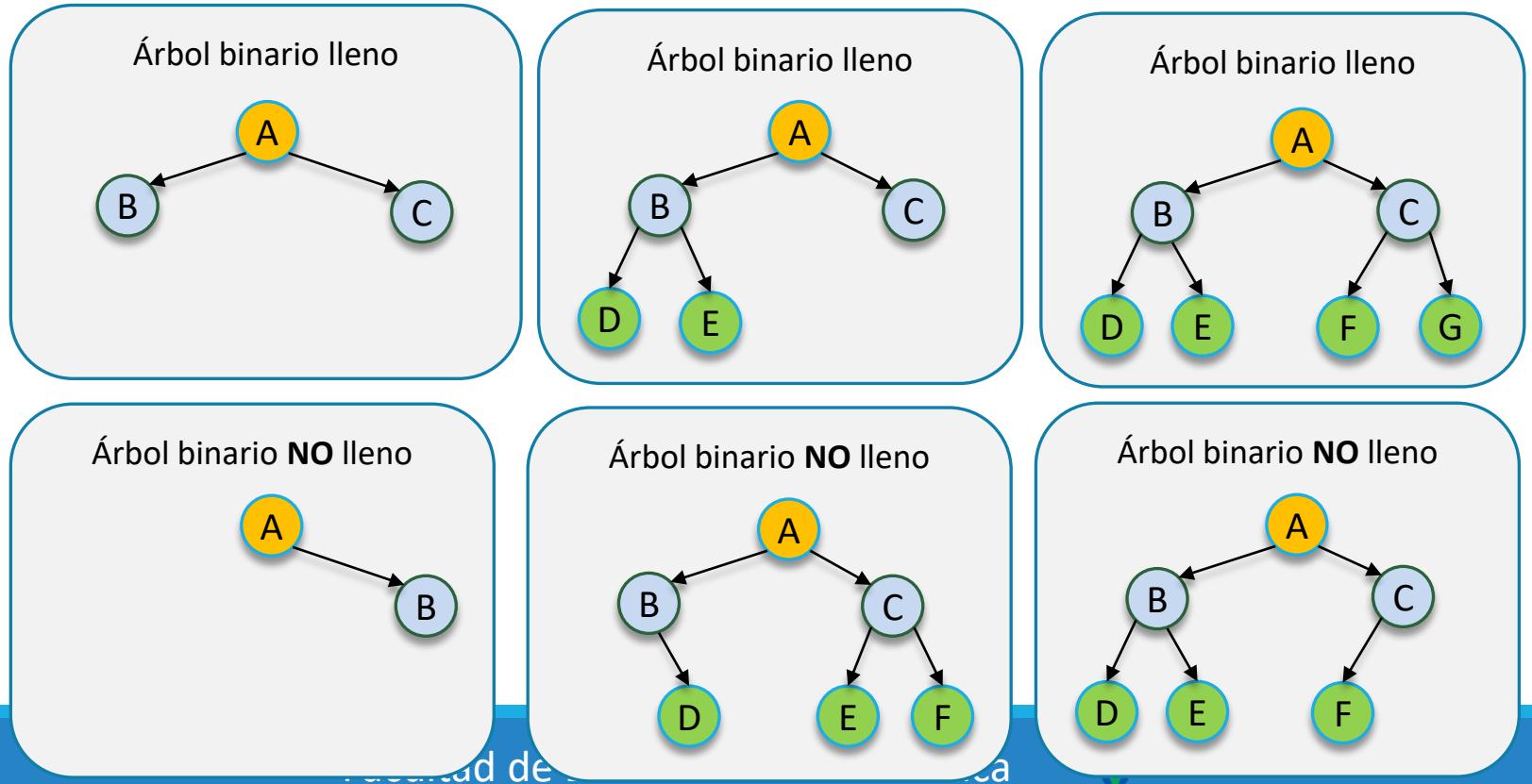
- Actividad:
- Elabora el Método bool Arbol::esLleno(Nodo *nodoPtr)

- Determina si un árbol binario de búsqueda es lleno.

- A continuación recordaremos lo que es un ABB lleno:

Árboles binarios llenos

Es un árbol lleno donde todos los nodos tienen cero o dos hijos. Es decir, no existe un nodo que tenga un solo hijo.



Árboles binarios de búsqueda. **Implementación**

- Actividad:
- Elabora el Método bool Arbol::esLleno(Nodo *nodoPtr)

- Por lo tanto, es un método recursivo que visita cada nodo del árbol, de tal manera que:
  - Retorna True si el nodo es nulo (un nodo nulo es lleno).
  - Retorna True si su nodo izquierdo y su nodo derecho son nulos (tiene cero hijos)
  - Si tiene nodo izquierdo y nodo derecho (tiene dos hijos), debe retornar y verificar de manera recursiva, que tanto su nodo derecho como su nodo izquierdo sean llenos.
  - Si ninguna de las condiciones se cumple, retorna Falso.

Árboles binarios de búsqueda. **Implementación**

- Actividad:
- Elabora el Método bool Arbol::esLleno(Nodo *nodoPtr)

```
bool Arbol::esLleno(Nodo *nodoPtr)
{
    if(nodoPtr == NULL)
        return true;    // Un nodoPtr vacio es lleno

    // Es una hoja
    if(nodoPtr->derechoPtr==NULL && nodoPtr->izquierdoPtr==NULL)
        return true;

    // Si tiene ambos hijos, revisamos que ambos
    // subnodos sean llenos
    if(nodoPtr->derechoPtr && nodoPtr->izquierdoPtr)
        return ( esLleno(nodoPtr->izquierdoPtr) && esLleno(nodoPtr->derechoPtr) );

    // En cualquier otro caso, no es lleno
    return false;
}
```

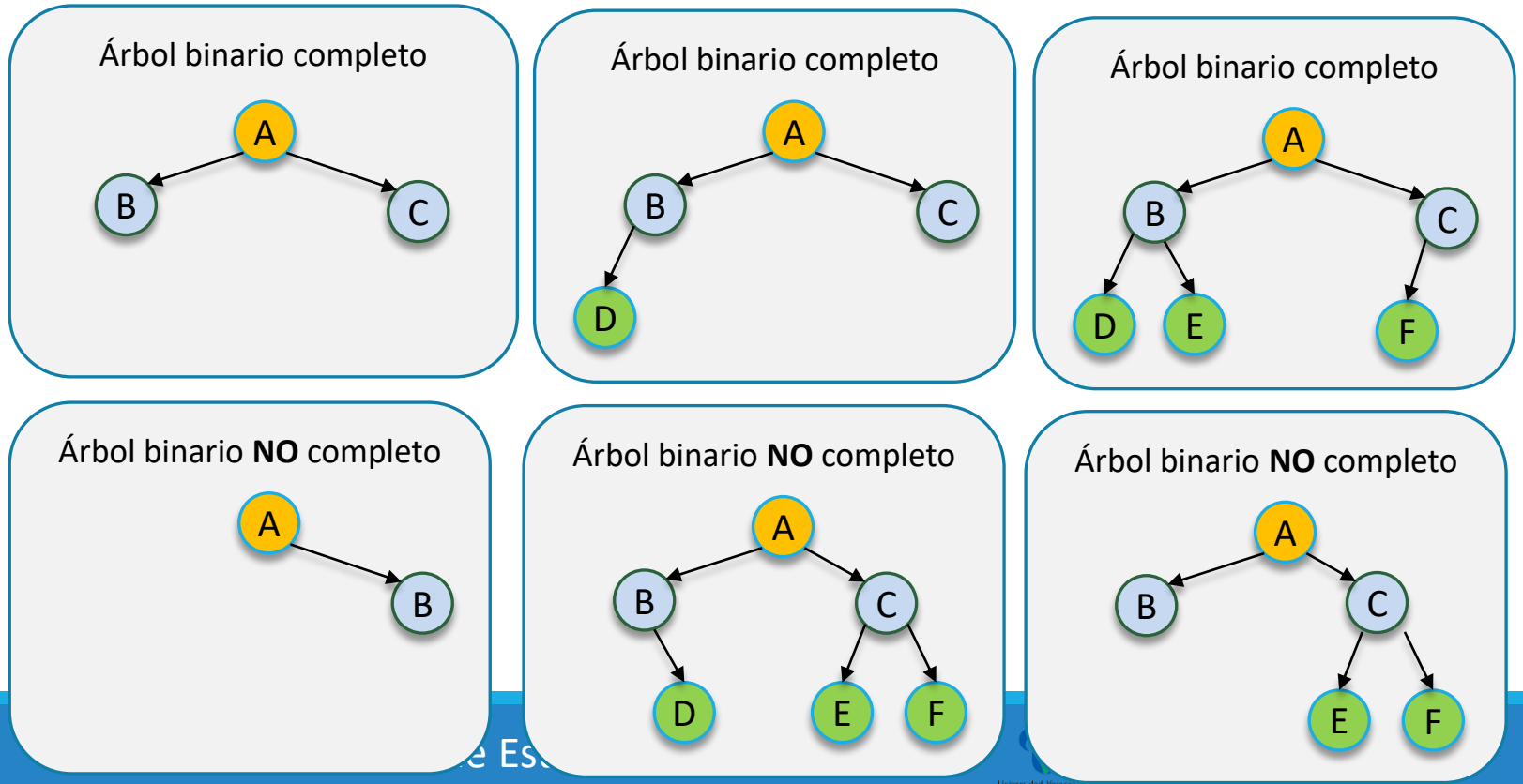
Árboles binarios de búsqueda. **Implementación**

- Actividad:
- Elabora el Método bool Arbol::esCompleto(Nodo *nodoPtr)
- Determina si un árbol binario de búsqueda es completo.

- Recordemos lo que es un ABB completo:

Árboles binarios completos

Un árbol binario completo de profundidad n es un árbol en el que, para cada nivel, del 0 al nivel n-1 tiene un conjunto lleno de nodos y todos los nodos hoja a nivel n ocupan las posiciones más a la izquierda del árbol



Árboles binarios de búsqueda.
Implementación

- Actividad:
- Elabora el Método bool Arbol::esCompleto(Nodo *nodoPtr)
- La verificación es por niveles, hasta llegar a las hojas; por lo tanto, no es recursivo.

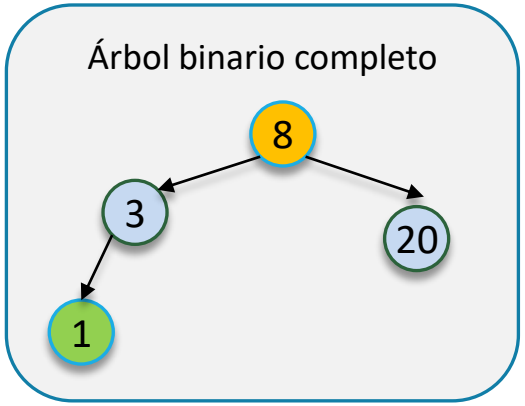
Árboles binarios de búsqueda. **Implementación**

Cola auxiliar:
nodoAuxiliar: 1
nodoNoLleno: True

- Actividad:
- Elabora el Método bool Arbol::esCompleto(Nodo *nodoPtr)

Considera una bandera nodoNoLleno=False, para "prender" cuando un nodo tenga 0 o 1 hijo en un nivel n-1 o cuando el nodo izquierdo no tenga hijos.

Se mete la raíz a la cola auxiliar
Mientras la cola auxiliar no esté vacía:
    Sacamos el nodo del frente de la cola, guardándolo en un nodo auxiliar
    Si tiene nodo izquierdo
        Si nodoNoLleno es verdadero, no es un árbol completo **Retorna falso**
        Se mete en la cola
    Si no tiene nodo izquierdo no es un nodo lleno. (nodoNoLleno verdadero)
    Si tiene nodo derecho
        Si nodoNoLleno es verdadero, no es un árbol completo **Retorna falso**
        Se mete en la cola
    Si no tiene nodo derecho, no es un nodo lleno. (nodoNoLleno verdadero)
Retorna verdadero

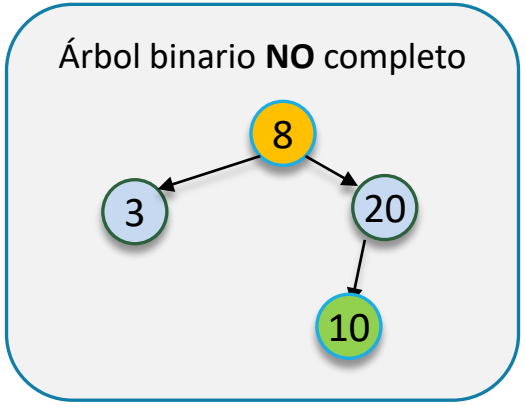


Árboles binarios de búsqueda. **Implementación**

Cola auxiliar:
nodoAuxiliar:
nodoNoLleno:

- Actividad:
- Elabora el Método bool Arbol::esCompleto(Nodo *nodoPtr)

Considera una bandera nodoNoLleno, para "prender" cuando un nodo tenga 0 o 1 hijo en un nivel n-1 o cuando el nodo izquierdo no tenga hijos.
Se mete la raíz a la cola auxiliar
Mientras la cola auxiliar no esté vacía:
    Sacamos el nodo del frente de la cola, guardándolo en un nodo auxiliar
    Si tiene nodo izquierdo
        Si nodoNoLleno es verdadero, no es un árbol completo **Retorna falso**
        Se mete en la cola
    Si no tiene nodo izquierdo no es un nodo lleno. (nodoNoLleno verdadero)
    Si tiene nodo derecho
        Si nodoNoLleno es verdadero, no es un árbol completo **Retorna falso**
        Se mete en la cola
    Si no tiene nodo derecho, no es un nodo lleno. (nodoNoLleno verdadero)
Retorna verdadero



Árboles binarios de búsque
Implementación

- Actividad:
- Elabora el Método bool Arbol::esCompl *nodoPtr)
- Determina si un árbol binario de búsqueda es

- Recordemos lo que es un ABB completo:

```
bool Arbol::esCompleto(Nodo *nodoPtr)
{
    if(nodoPtr == NULL)
        return true;    // Un arbol vacio es completo

    Nodo *nodoAuxiliar;
    queue<Nodo*> colaAuxiliar;
    colaAuxiliar.push(nodoPtr);
    bool nodoNoLleno = false;

    while (! colaAuxiliar.empty() )
    {
        nodoAuxiliar = colaAuxiliar.front();    // obtiene el valor del frente
        colaAuxiliar.pop(); // saca el valor de la cola

        // Revisa si el nodo izq esta presente
        if(nodoAuxiliar->izquierdoPtr){
            // Si hemos visto un nodo NO lleno, y vemos un nodo
            // con un nodo izq ocupado, no es un arbol completo
            if (nodoNoLleno == true)
                return false;
            colaAuxiliar.push(nodoAuxiliar->izquierdoPtr);
        }else{
            // Si el nodo izq no esta presente,
            // entonces no es un arbol completo
            nodoNoLleno = true;
        }

        // Revisa si el nodo der esta presente
        if(nodoAuxiliar->derechoPtr){
            // Si hemos visto un nodo NO lleno, y vemos un nodo
            // con un nodo der ocupado, no es un nodoPtr completo
            if (nodoNoLleno == true)
                return false;
            colaAuxiliar.push(nodoAuxiliar->derechoPtr);
        }else{
            // Si el nodo der no esta presente,
            // entonces no es un nodoPtr completo
            nodoNoLleno = true;
        }
    }
    // Es un arbol completo
    return true;
}
```

Árboles binarios de búsqueda. **Implementación**

- Arbol::graficarArbol(Nodo *nodoPtr, int x, int y, int n)

```
void Arbol::graficarArbol(Nodo *nodoPtr, int x, int y, int n)
{
    if(nodoPtr == NULL)
        return;

    gotoxy(x,y); cout<< nodoPtr->dato;

    graficarArbol(nodoPtr->izquierdoPtr, x-15+n*6, y+2, n+1);

    graficarArbol(nodoPtr->derechoPtr, x+15-n*6, y+2, n+1);

}
```

Árboles binarios de búsqueda. **Implementación**

Método void Arbol::gotoxy(int x,int y)

```
void Arbol::gotoxy(int x,int y)
{
    COORD coord;
    coord.X = x;
    coord.Y = y;
    SetConsoleCursorPosition(GetStdHandle(STD_OUTPUT_HANDLE), coord);
}
```

Árboles binarios de búsqueda. **Implementación**

- void menú()

```
void menu()
{
    system("cls");
    cout << "\n    ..[ ÁRBOL BINARIO DE BÚSQUEDA ]..";
    cout << "\n  ..[ Erika Meneses Rico ].. \n\n";
    cout << "[1] Insertar elemento \n";
    cout << "[2] Imprime el valor de la raiz \n";
    cout << "[3] Mostrar arbol completo acostado con la raiz a la izquierda \n";
    cout << "[4] Graficar arbol completo \n";
    cout << "[5] Buscar un elemento en el arbol \n";
    cout << "[6] Recorrer el arbol en PreOrden \n";
    cout << "[7] Recorrer el arbol en InOrden \n";
    cout << "[8] Recorrer el arbol en PostOrden \n";
    cout << "[9] Eliminar un nodo del arbol PREDECESOR \n";
    cout << "[10] Eliminar un nodo del arbol SUCESOR \n";
    cout << "------------------------------------------------------------ \n";
    cout << "[11] Recorrer el arbol por niveles (Amplitud) \n";
    cout << "[12] Altura del arbol \n";
    cout << "[13] Cantidad de hojas del arbol \n";
    cout << "[14] Cantidad de nodos del arbol \n";
    cout << "[15] Mostrar arbol espejo \n";
    cout << "[16] Revisa si es un arbol binario completo \n";
    cout << "[17] Revisa si es un arbol binario lleno \n";
    cout << "[18] Construir el arbol 8, 3, 1, 20, 1, 5, 10, 7, 4 \n";
    cout << "[19] Eliminar el arbol \n";
    cout << "[20] SALIR \n";
    cout << "\nIngrese opcion : ";
}
```

Árboles binarios de búsqueda. **Implementación**

- Int main()

```
419  int main()
420  {
421      int opcion; // almacena la opcion del usuario
422      int valor;  // almacena el valor del nodo
423      int x;  // almacena un valor temporal
424
425      Arbol arbolEnteros; // crea el objeto Arbol
426      Nodo *raizArbolPtr = arbolEnteros.regresaRaiz();    // obtengo su nodo raiz
427
428      Arbol arbolCopiaEnteros;    // crea el objeto Arbol
429      Nodo *raizArbolCopiaPtr = arbolCopiaEnteros.regresaRaiz();  // obtengo su nodo raiz
430
431      system("color 1F");   // poner color a la consola
432
433      do
434      {
435          menu();
436          cin >> opcion;
437
438          switch( opcion )
439          {
440              case 1:
441                  cout << "\nIngrese valor entero: ";
442                  cin >> valor;
443
444                  arbolEnteros.insertarNodo( valor, raizArbolPtr );
445
446                  cout << "\nLos elementos del arbol acostado con la raiz a la izquierda son:\n\n";
447                  arbolEnteros.muestraAcostado( 0, raizArbolPtr );
448                  cout << "\n\n";
449                  system("pause");
```

Árboles binarios de búsqueda. **Implementación**

- Int main()

```
449             system("pause");
450             break;
451         case 2:
452             //nodoPtr = arbolEnteros.regresaRaiz();
453             if( raizArbolPtr == NULL )
454             {
455                 cout << "\nEl arbol esta vacio.\n\n";
456                 system("pause");
457                 break;
458             }
459
460             cout << "\nEl valor del nodo de raiz es: " << raizArbolPtr->dato << "\n\n";
461             system("pause");
462             break;
463         case 3:
464             if( raizArbolPtr == NULL )
465             {
466                 cout << "\nEl arbol esta vacio.\n\n";
467                 system("pause");
468                 break;
469             }
470             cout << "\nLos elementos del arbol acostado con la raiz a la izquierda son:\n\n";
471             arbolEnteros.muestraAcostado( 0, raizArbolPtr );
472             cout << "\n\n";
473             system("pause");
474             break;
475         case 4:
476             if( raizArbolPtr == NULL )
477             {
478                 cout << "\nEl arbol esta vacio.\n\n";
479                 system("pause");
```

Árboles binarios de búsqueda. **Implementación**

- Int main()

```
478                 cout << "\nEl arbol esta vacio.\n\n";
479                 system("pause");
480                 break;
481             }
482
483             system("cls");
484             arbolEnteros.graficarArbol(raizArbolPtr, 45, 2, 0);
485             cout << "\n\n\n\n\n\n\n\n\n";
486             system("pause");
487             break;
488         case 5:
489             if( raizArbolPtr == NULL )
490             {
491                 cout << "\nEl arbol esta vacio.\n\n";
492                 system("pause");
493                 break;
494             }
495
496             cout << "\nIngrese valor entero: ";
497             cin >> x;
498
499             if( arbolEnteros.busqueda( x, raizArbolPtr ) == true )
500                 cout << "\nElemento " << x << " ha sido encontrado en el arbol\n";
501             else
502                 cout << "\nElemento no encontrado\n";
503
504             cout << "\nLos elementos del arbol acostado con la raiz a la izquierda son:\n\n";
505             arbolEnteros.muestraAcostado( 0, raizArbolPtr );
506             cout << "\n\n";
507             system("pause");
508             break;
509         case 6:
```

Árboles binarios de b
Implementación

- Int main()

```
508             break;
509         case 6:
510             if( raizArbolPtr == NULL )
511             {
512                 cout << "\nEl arbol esta vacio.\n\n";
513                 system("pause");
514                 break;
515             }
516
517             cout << "\nRecorrido en PreOrden: ";
518             arbolEnteros.preOrden(raizArbolPtr);
519             cout << "\n\n";
520             system("pause");
521             break;
522         case 7:
523             if( raizArbolPtr == NULL )
524             {
525                 cout << "\nEl arbol esta vacio.\n\n";
526                 system("pause");
527                 break;
528             }
529
530             cout << "\nRecorrido en InOrden: ";
531             arbolEnteros.inOrden(raizArbolPtr);
532             cout << "\n\n";
533             system("pause");
534             break;
535         case 8:
536             if( raizArbolPtr == NULL )
537             {
538                 cout << "\nEl arbol esta vacio.\n\n";
539                 system("pause");
540                 break;
541             }
542
543             cout << "\nRecorrido en PostOrden: ";
544             arbolEnteros.postOrden(raizArbolPtr);
545             cout << "\n\n";
546             system("pause");
547             break;
```

Árboles binario
Implementació

- Int main()

```
case 9:
    if( raizArbolPtr == NULL )
    {
        cout << "\nEl arbol esta vacio.\n\n";
        system("pause");
        break;
    }

    cout << "\nIngrese el valor entero a eliminar PREDECESOR: ";
    cin >> x;

    arbolEnteros.eliminarPredecesor(x, raizArbolPtr);

    cout << "\nLos elementos del arbol acostado con la raiz a la izquierda son:\n\n";
    arbolEnteros.muestraAcostado( 0, raizArbolPtr );
    cout << "\n\n";
    system("pause");
    break;
case 10:
    if( raizArbolPtr == NULL )
    {
        cout << "\nEl arbol esta vacio.\n\n";
        system("pause");
        break;
    }

    cout << "\nIngrese el valor entero a eliminar SUCESOR: ";
    cin >> x;

    arbolEnteros.eliminarSucesor(x, raizArbolPtr);

    cout << "\nLos elementos del arbol acostado con la raiz a la izquierda son:\n\n";
    arbolEnteros.muestraAcostado( 0, raizArbolPtr );
    cout << "\n\n";
    system("pause");
    break;
```

Árboles binarios de búsqueda. **Implementación**

- Int main()

```
case 11:
    cout << "\nRecorrido por niveles: ";
    arbolEnteros.recorridoNiveles(raizArbolPtr);

    cout << "\n\n";
    system("pause");
    break;
case 12:
    cout << "\nLa altura del arbol es: " << arbolEnteros.altura(raizArbolPtr);

    cout << "\n\n";
    system("pause");
    break;
case 13:
    cout << "\nEl numero de hojas del arbol es: " << arbolEnteros.contarHojas(raizArbolPtr);

    cout << "\n\n";
    system("pause");
    break;
case 14:
    cout << "\nEl numero de nodos del arbol es: " << arbolEnteros.contarNodos(raizArbolPtr);

    cout << "\n\n";
    system("pause");
    break;
```

Árboles binarios de búsqueda. **Implementación**

- Int main()

```
case 16:
    cout << "\nLos elementos del arbol acostado con la raiz a la izquierda son:\n\n";
    arbolEnteros.muestraAcostado( 0, raizArbolPtr );
    cout << "\n\n";

    if(arbolEnteros.esCompleto(raizArbolPtr))
        cout << "\nSI es un arbol binario completo.";
    else
        cout << "\nNO es un arbol binario completo.";

    cout << "\n\n";
    system("pause");
    break;
case 17:
    cout << "\nLos elementos del arbol acostado con la raiz a la izquierda son:\n\n";
    arbolEnteros.muestraAcostado( 0, raizArbolPtr );
    cout << "\n\n";

    if(arbolEnteros.esLleno(raizArbolPtr))
        cout << "\nSI es un arbol binario lleno.";
    else
        cout << "\nNO es un arbol binario lleno.";

    cout << "\n\n";
    system("pause");
    break;
```

Árboles binarios de búsqueda.
Implementación

- Int main()

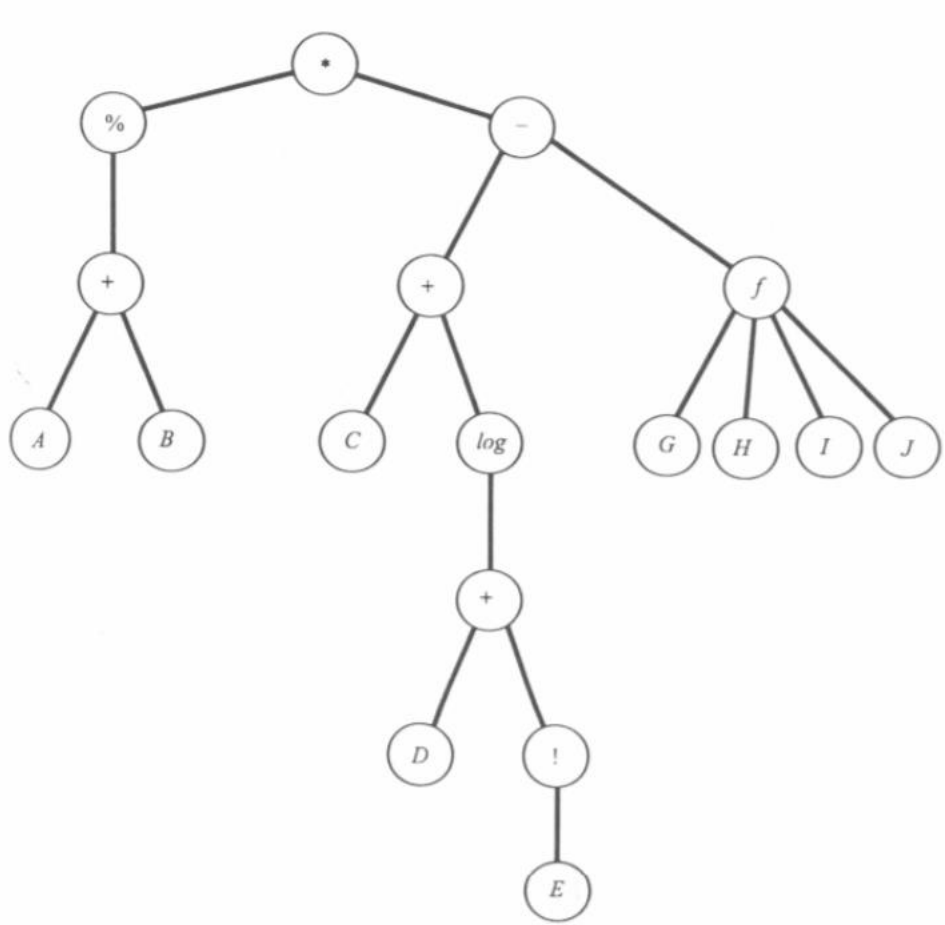
```
            break;
        case 18:
            arbolEnteros.insertarNodo(8, raizArbolPtr);
            arbolEnteros.insertarNodo(3, raizArbolPtr);
            arbolEnteros.insertarNodo(1, raizArbolPtr);
            arbolEnteros.insertarNodo(20, raizArbolPtr);
            arbolEnteros.insertarNodo(1, raizArbolPtr);
            arbolEnteros.insertarNodo(5, raizArbolPtr);
            arbolEnteros.insertarNodo(10, raizArbolPtr);
            arbolEnteros.insertarNodo(7, raizArbolPtr);
            arbolEnteros.insertarNodo(4, raizArbolPtr);

            cout << "\nLos elementos del arbol acostado con la raiz a la izquierda son:\n\n";
            arbolEnteros.muestraAcostado( 0, raizArbolPtr );
            cout << "\n\n";
            system("pause");
            break;
        case 19:
            arbolEnteros.podarArbol(raizArbolPtr);
            cout << "\nEl arbol ha sido podado.";
            cout << "\n\n";

            cout << "\n\n";
            system("pause");
            break;
    }
}while( opcion != 20 );

return 0;
}
```

Árboles binarios de búsqueda. **Aplicaciones**

- Aplicaciones de Árboles

- Una aplicación particular de los Árboles es la representación de expresiones aritméticas.
- El árbol mostrado representa la siguiente expresión:

$$(a)\ -(A+B)*(C+log(D+E!)-f(G,H,I,J))$$

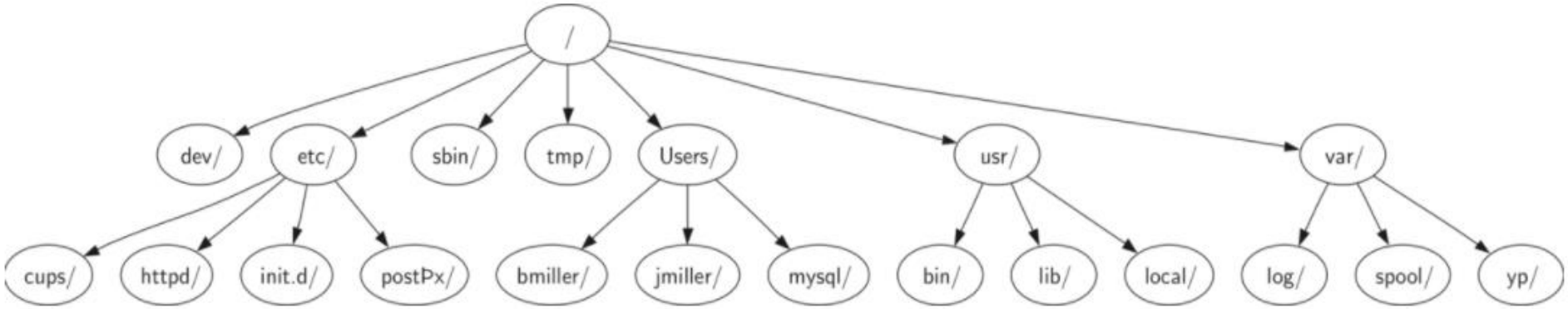


(a) $-(A+B)*(C+log(D+E!)-f(G,H,I,J))$

Árboles binarios de búsqueda. **Aplicaciones**

- Aplicaciones de Árboles

- Otra aplicación de los árboles son los sistemas de archivos.



Árboles binarios de búsqueda. **Aplicaciones**

- Aplicaciones de Árboles

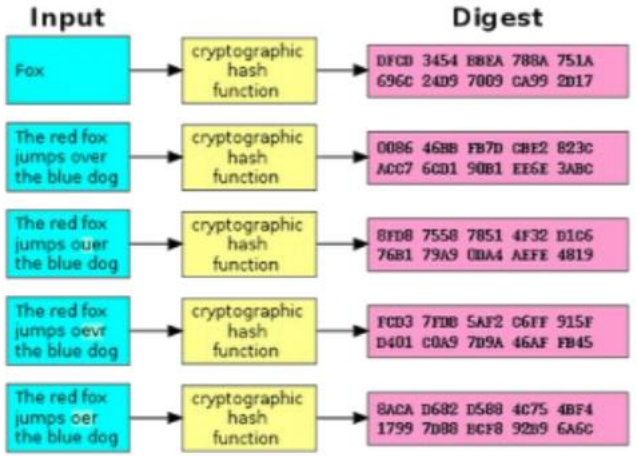
- En general, como un método eficiente para búsquedas grandes y complejas, en bases de datos por ejemplo, para evitar redundancia por ejemplo.
- Diseño de compiladores, proceso de texto y algoritmos de búsqueda.

Árboles binarios de búsqueda. **Aplicaciones**

- Aplicaciones de Árboles

- Árbol Hash de Merkle.

Es un árbol de decisión estadístico compuesto por hashes. Se utiliza en seguridad, verificación y como forma de comprensión de datos



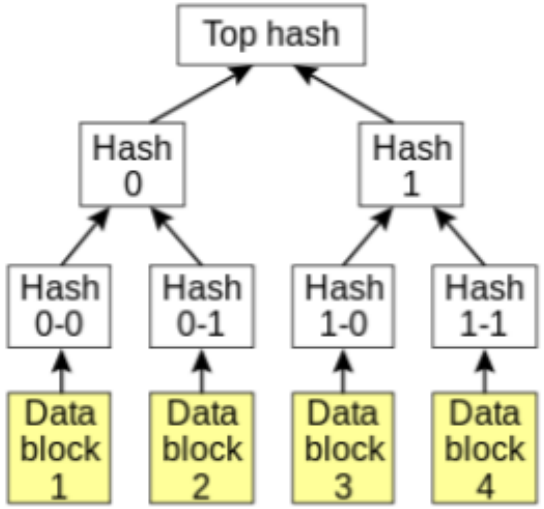
Árboles binarios de búsqueda. **Aplicaciones**

- Aplicaciones de Árboles

- Árbol Hash de Merkle.

En la parte superior del árbol se encuentra el “Hash Raíz” que es el que sirve como forma de verificación de datos. Por ejemplo, en una operación p2p.

Cuando se obtienen los diferentes bloques informativos del mensaje o archivo, se podría obtener el hash raíz de una fuente confiable para poder interactuar con el resto de la red. De esta forma, se recibirá el hash raíz de cualquier usuario con el que se entable relación directa y se comparará con el confiable y, así, las fuentes falsas o con el contenido inadecuado serán rechazadas en post de otras que provean el hash correcto.



Árboles binarios de búsqueda. **Aplicaciones**

- Aplicaciones de Árboles

- Árbol Hash de Merkle.

En Bitcoin , las transacciones una vez registradas pasan a la red. Los mineros se encargan de recoger todas estas transacciones y validar la autenticidad de las mismas.

Si las transacciones son válidas el minero las añade a su registro y comienza la creación del bloque.

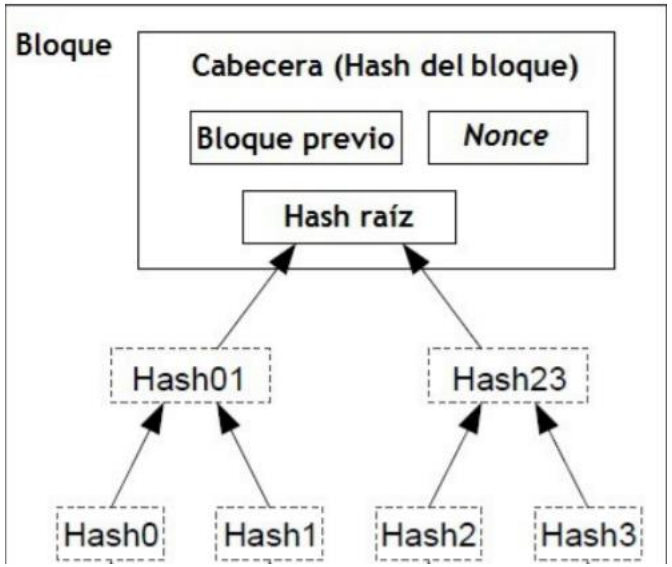
Árboles binarios de búsqueda. **Aplicaciones**

- Aplicaciones de Árboles

- Árbol Hash de Merkle.

Para crear un bloque, se hashean las transacciones utilizando el algoritmo SHA-256 y se organizan en el árbol de Merkle.

La raíz pasara a formar parte de la cabecera del bloque y, junto al hash del bloque anterior y el nonce, se obtendrá el hash del bloque que sirve como identificador.



Bibliografía

Cairó, O. y Guardati, S. (2002). Estructuras de Datos, 2da. Edición. McGraw-Hill.

Deitel P.J. y Deitel H.M. (2008) Cómo programar en C++. 6ª edición.Prentice Hall.

Joyanes, L. (2006). Programación en C++: Algoritmos, Estructuras de datos y objetos. McGraw-Hill.