ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS ARBORESCENTES

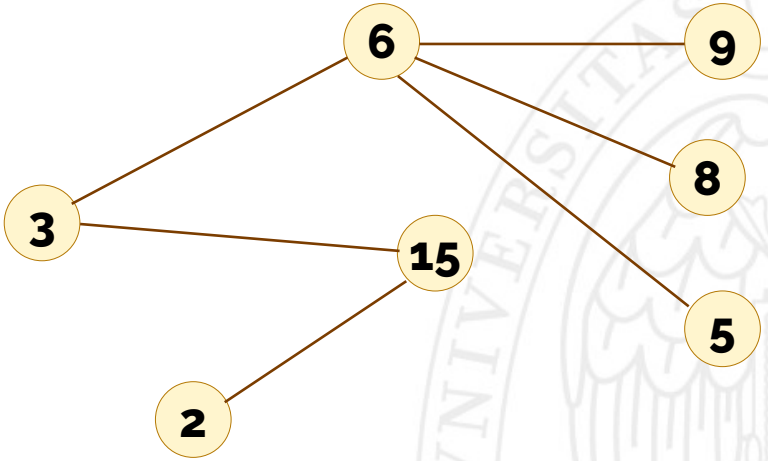
Introducción a los árboles

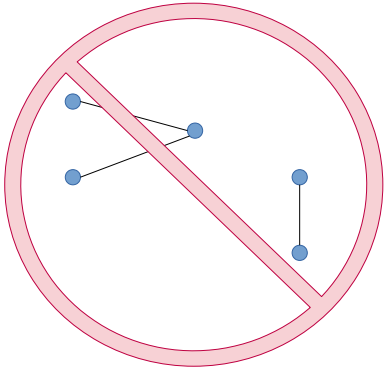
Manuel Montenegro Montes
Departamento de Sistemas Informáticos y Computación
Facultad de Informática – Universidad Complutense de Madrid

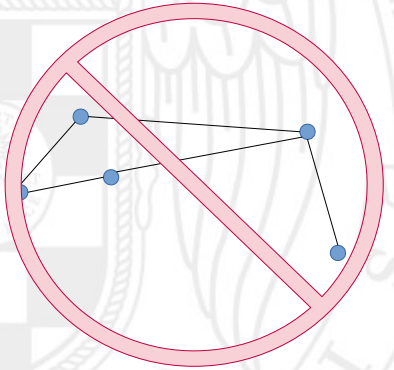
¿Qué es un árbol?

El TAD Árbol

- Un árbol es un grafo **conexo** y **sin ciclos**.
- Llamamos a sus vértices **nodos**.

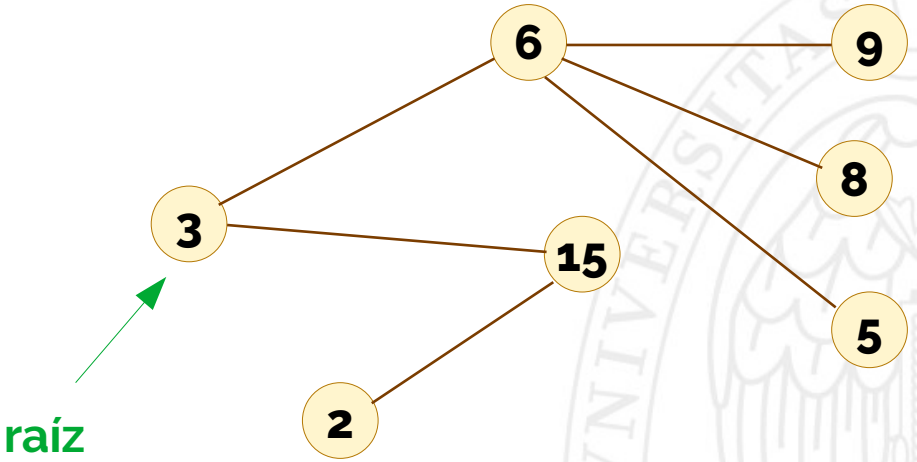






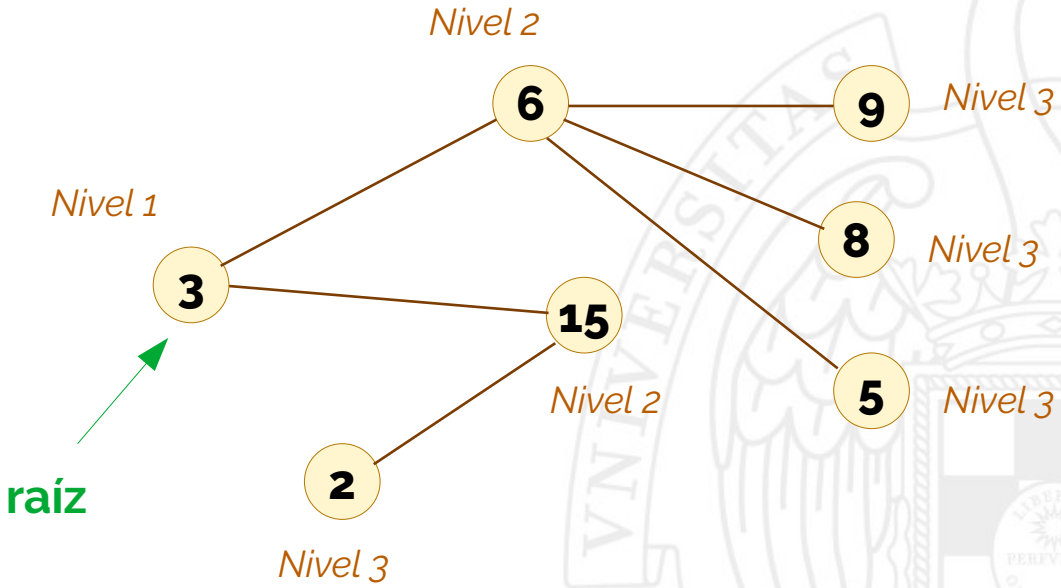
Árboles con raíz

- Distinguimos un nodo en particular, que es la **raíz** del árbol.

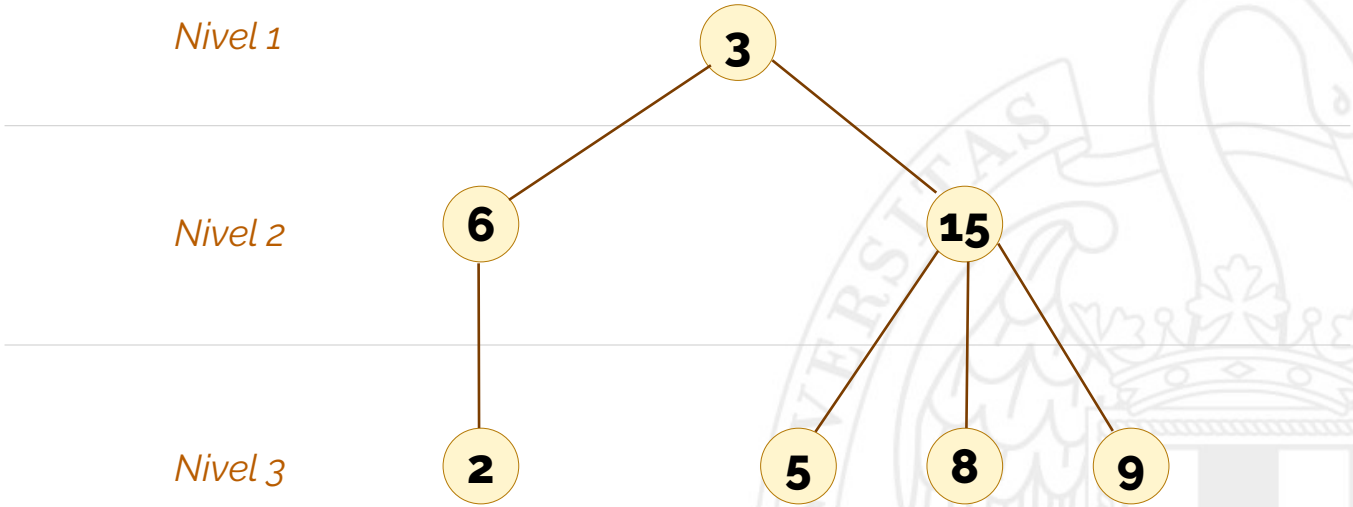


Nivel de un nodo

- El **nivel** de un nodo se define como el número de aristas que lo separan de la raíz incrementado en 1.
- La raíz está en el nivel 1.

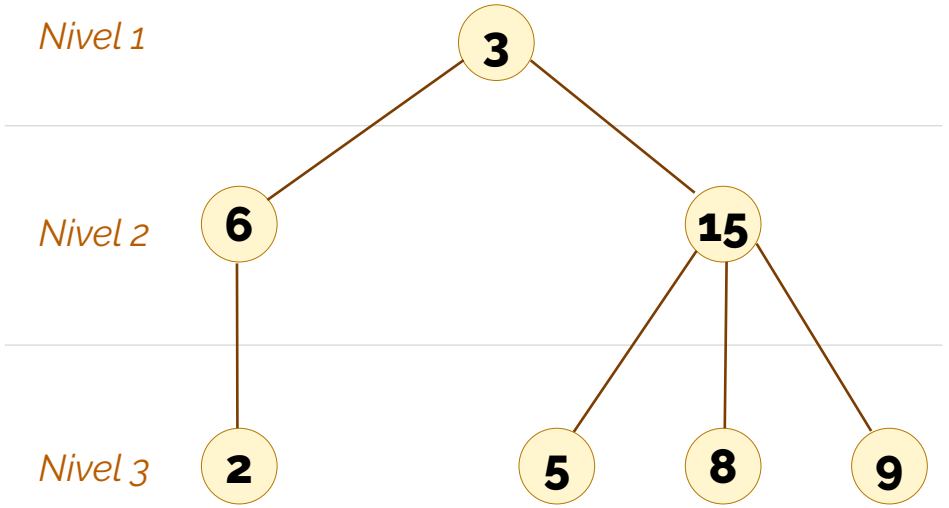


Nivel de un nodo



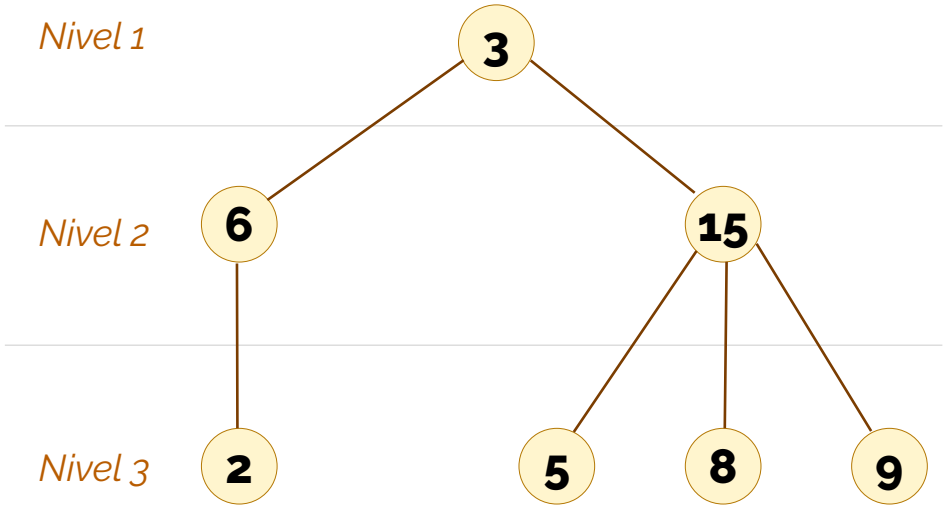
Definiciones

Padres e hijos



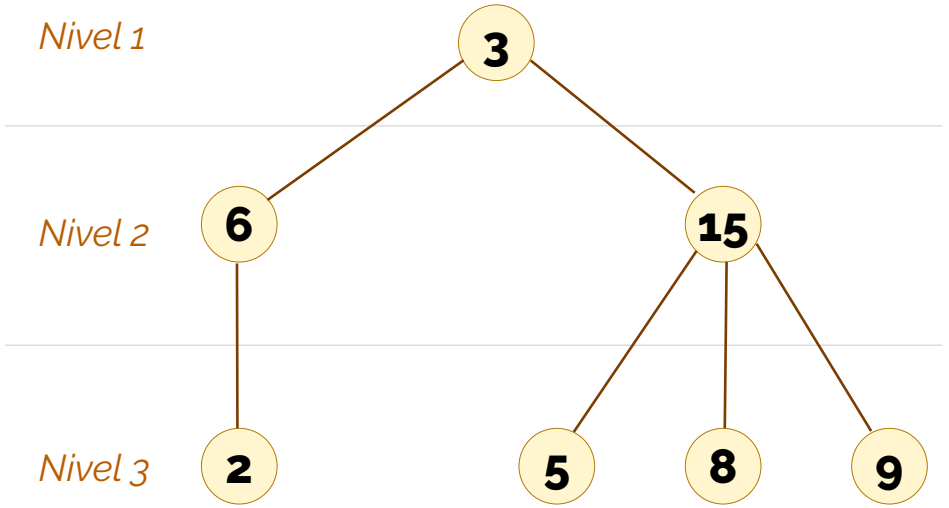
- Si **X** es un nodo que está a nivel *n*, su **padre** es el que está conectado con él en el nivel *n-1*.

Padres e hijos



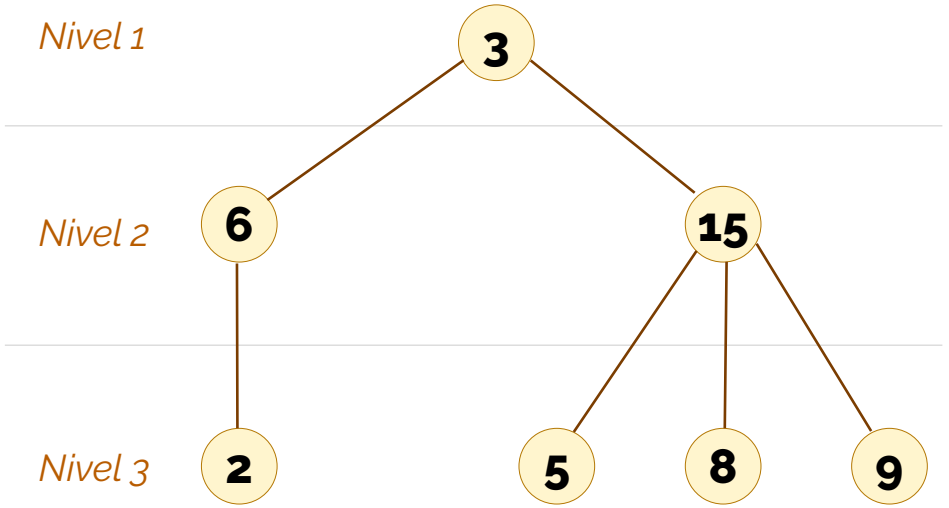
- Si **X** es un nodo que está a nivel *n*, sus **hijos** son aquellos conectados con él en el nivel *n+1*.

Hermanos



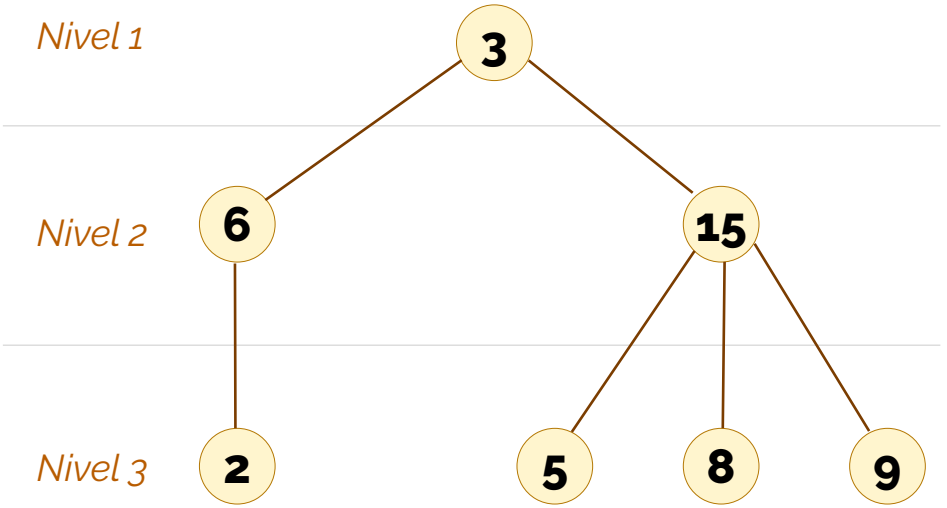
- Dos nodos son **hermanos** si tienen el mismo padre.

Hojas vs nodos internos



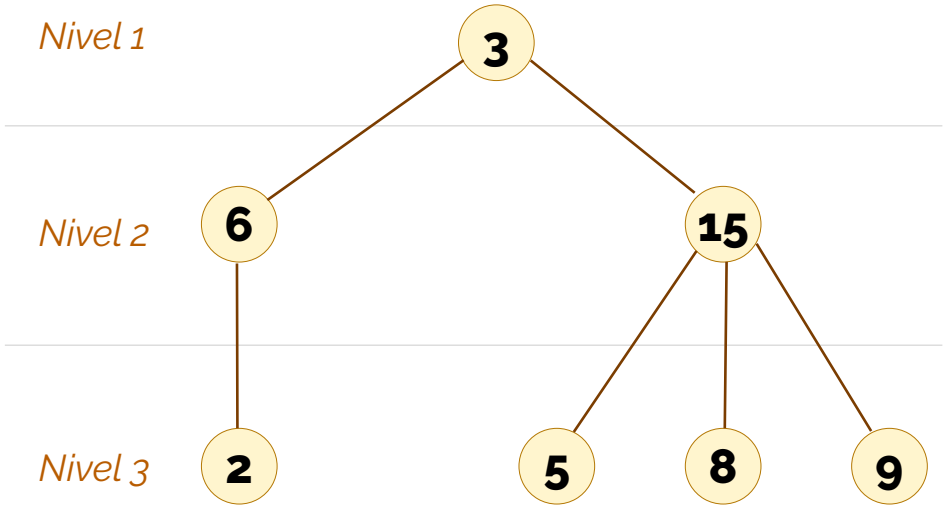
- Una **hoja** es un nodo que no tiene hijos.
- El resto de nodos son **nodos internos**.

Caminos y longitud



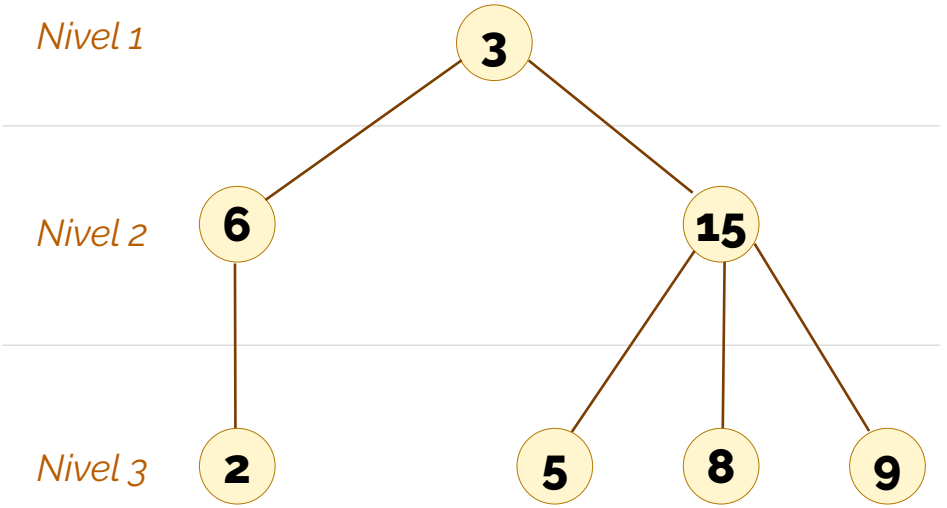
- Un **camino** es una sucesión de nodos en la que cada nodo es padre del siguiente.
- La **longitud de un camino** es el número de nodos que hay en él.

Ramas



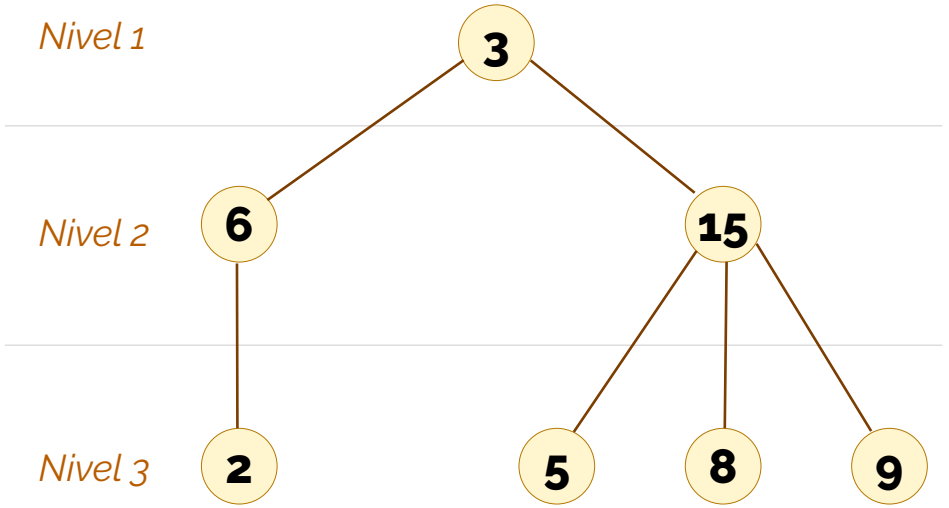
- Si un camino empieza en la raíz y termina en una hoja, decimos que es una **rama.**

Antepasados y descendientes



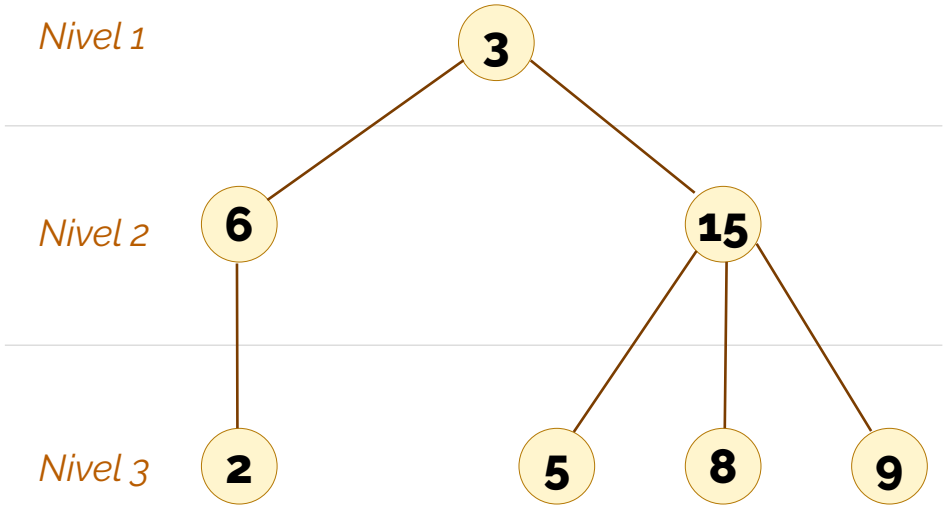
- Decimos que X es **antepasado** de Y si existe un camino de X a Y.
- Decimos que Y es **descendiente** de X si existe un camino de X a Y.

Altura



- La **altura** de un árbol es el máximo de los niveles de los nodos.
- Equivalentemente, es la longitud de la rama más larga.

Grado o aridad

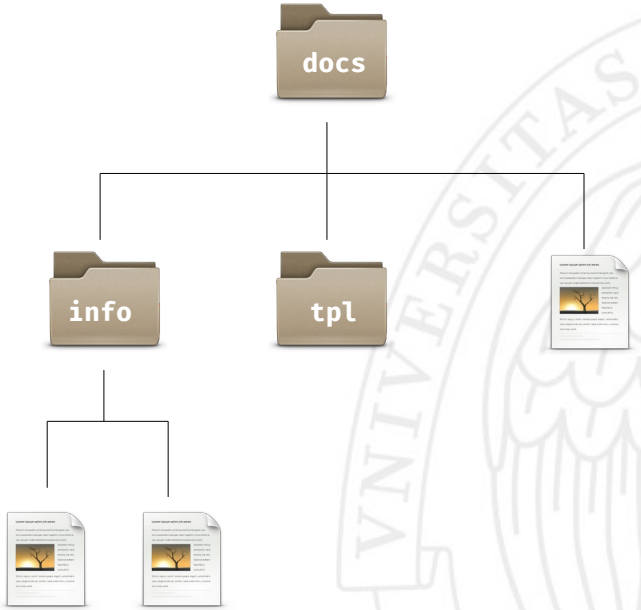


- El **grado** (o aridad) de un **nodo** es el número de hijos que tiene.
- La **aridad** de un **árbol** es el máximo de los grados de los nodos.

Aplicaciones en un árbol

Aplicaciones de los árboles

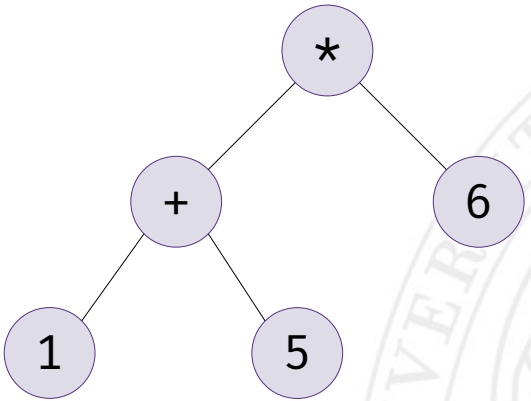
- Los árboles se utilizan para representar datos que están jerarquizados de alguna manera, o se contienen unos a otros.



Aplicaciones de los árboles

- Los árboles se utilizan para representar datos que están jerarquizados de alguna manera, o se contienen unos a otros.

`“(1 + 5) * 6”`



Aplicaciones de los árboles

- Los árboles se utilizan para representar datos que están jerarquizados de alguna manera, o se contienen unos a otros.

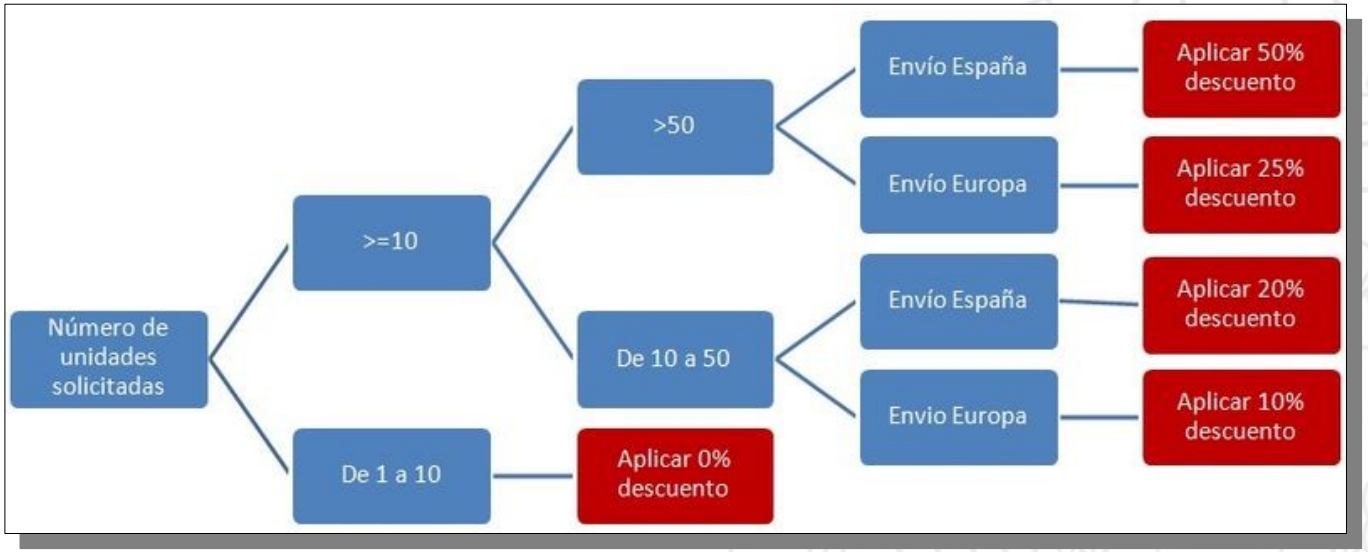


Imagen: Sargantano (CC BY-SA 3.0)

ESTRUCTURAS DE DATOS

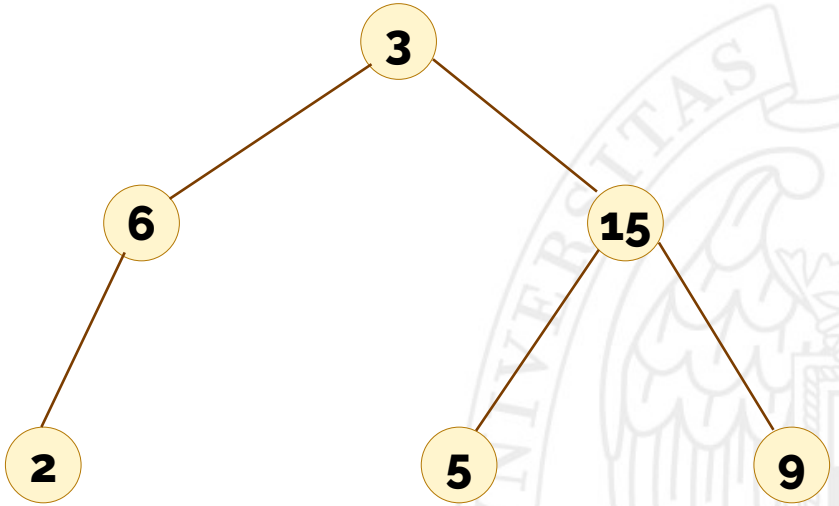
TIPOS ABSTRACTOS DE DATOS ARBORESCENTES

El TAD Árbol Binario

Manuel Montenegro Montes
Departamento de Sistemas Informáticos y Computación
Facultad de Informática – Universidad Complutense de Madrid

Árboles binarios

- Un árbol binario es un árbol de aridad 2.
- Cada nodo tiene **2** hijos, algunos de los cuales pueden ser vacíos.

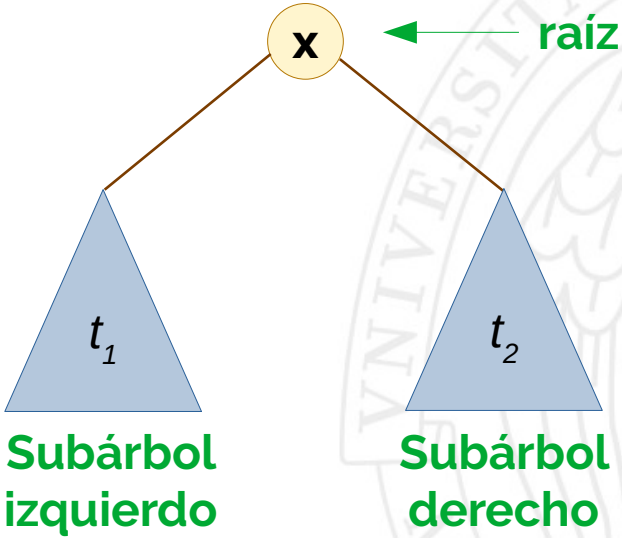


Definición inductiva de un árbol binario

- *Caso base:* Un grafo sin nodos es un **árbol vacío**.

—

- *Caso recursivo:* Si t_1 y t_2 son árboles binarios, y x es un elemento, entonces lo siguiente es un árbol binario:



Si t_1 o t_2 son vacíos, no existen aristas desde x hacia ellos.

Definición inductiva de un árbol binario

- Un árbol binario T es un conjunto finito tal que:
  - $T = \varnothing$, o bien
  - $T = \{x\} \uplus T_1 \uplus T_2$, donde T_1 y T_2 son árboles.

    x es la raíz,

    T_1 es el subárbol izquierdo, y

    T_2 es el subárbol derecho.

Operaciones en el TAD Árbol Binario

- Constructoras:
  - Crear un árbol vacío: ***create_empty***.
  - Crear una hoja: ***create_leaf***.
  - Crear un árbol a partir de una raíz y dos hijos: ***create_tree***.
- Observadoras:
  - Determinar si el árbol es vacío: ***empty***.
  - Obtener la raíz si el árbol no es vacío: ***root***.
  - Obtener el hijo izquierdo, si existe: ***left***.
  - Obtener el hijo derecho, si existe: ***right***.

Operaciones constructoras

{ *true* }

create_empty() → (*T*: ArBin)

{ *T* = — }

{ *true* }

create_leaf(*x*: Elem) → (*T*: ArBin)

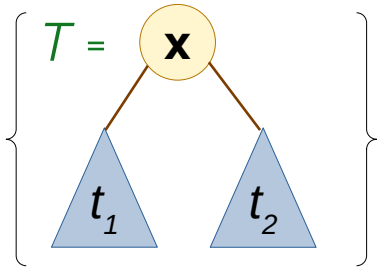
{ *T* = **x** }

{ T_1 = t_1 T_2 = t_2 }

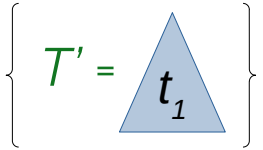
create_tree(T_1: ArBin, *x*: Elem, T_2: ArBin) → (*T*: ArBin)

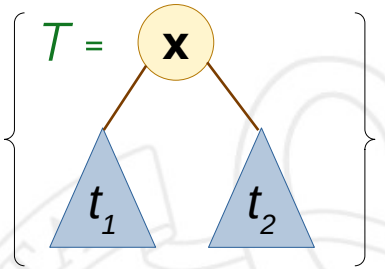
{ *T* = **x** (t_1, t_2) }

Operaciones observadoras

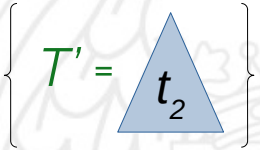


left(*T: ArBin*) → (*T' : ArBin*)

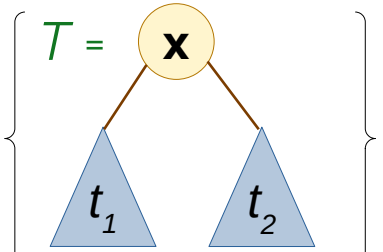




right(*T: ArBin*) → (*T' : ArBin*)



Operaciones observadoras

{ *T* = 

}

root(*T: ArBin*) → (*e: elem*)

{ *e* = **x** }

{ *true* }

empty(*T: ArBin*) → (*b: bool*)

{ *b* ⇔ *T* = — }

ESTRUCTURAS DE DATOS

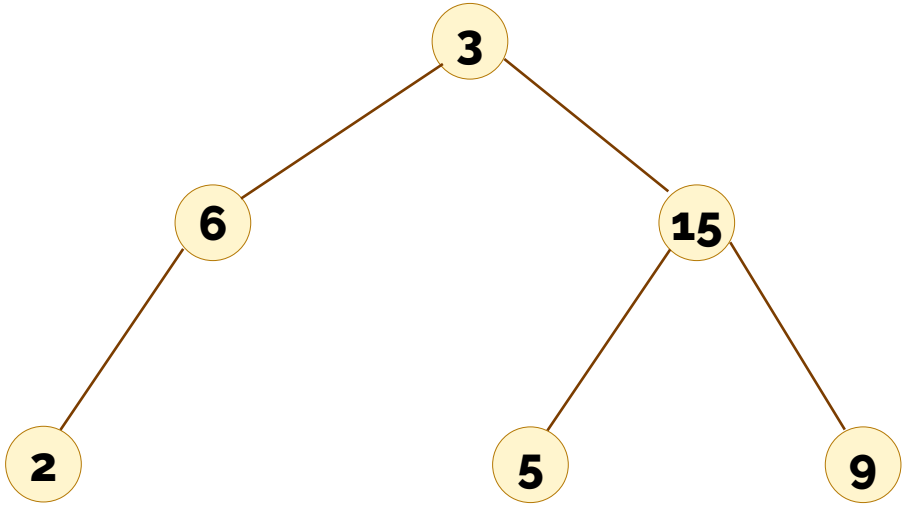
TIPOS ABSTRACTOS DE DATOS ARBORESCENTES

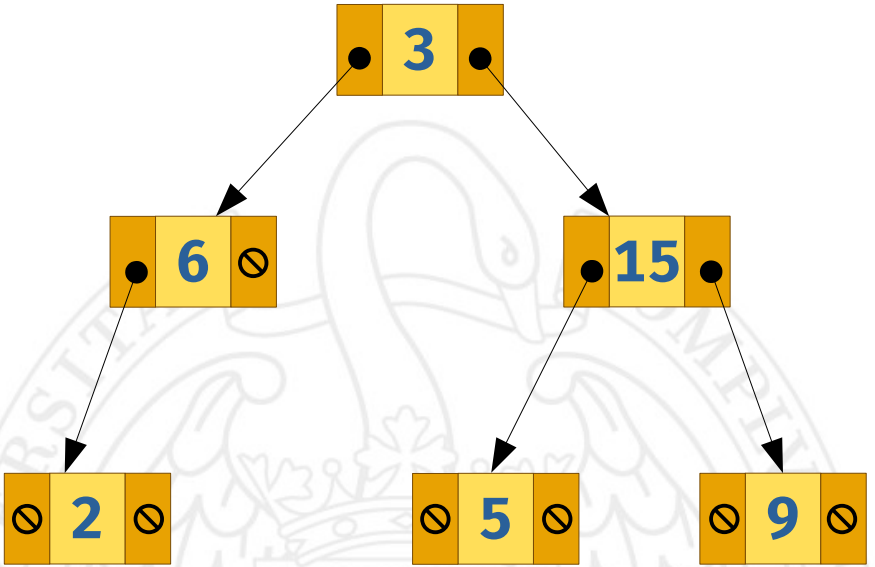
Implementación de árboles binarios

Manuel Montenegro Montes
Departamento de Sistemas Informáticos y Computación
Facultad de Informática – Universidad Complutense de Madrid

Representación mediante nodos

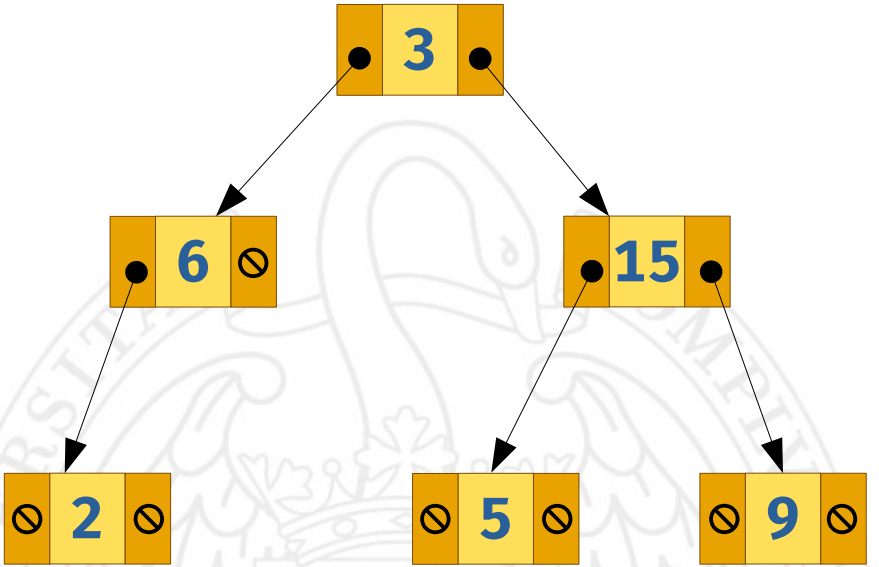
Representando árboles binarios





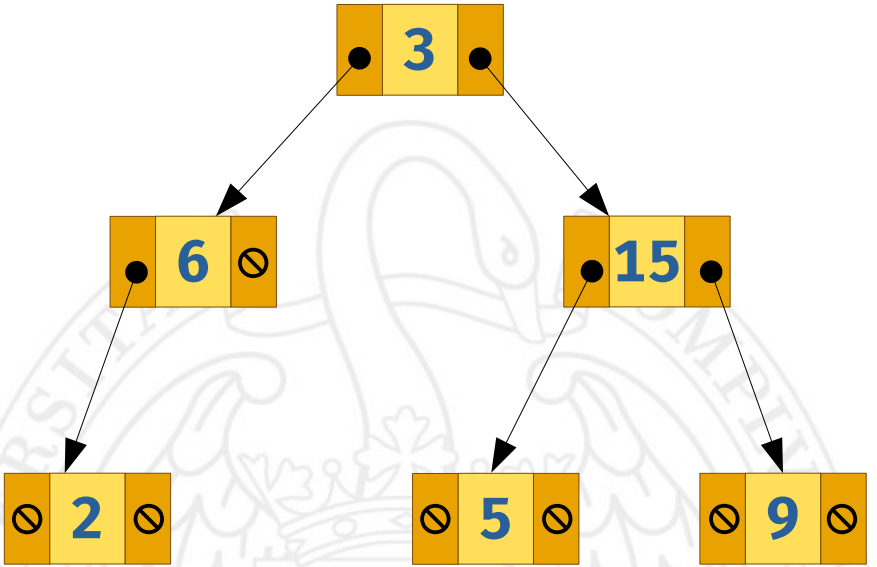
Representando árboles binarios

```
struct TreeNode {
  T elem;
  TreeNode *left, *right;
};
```



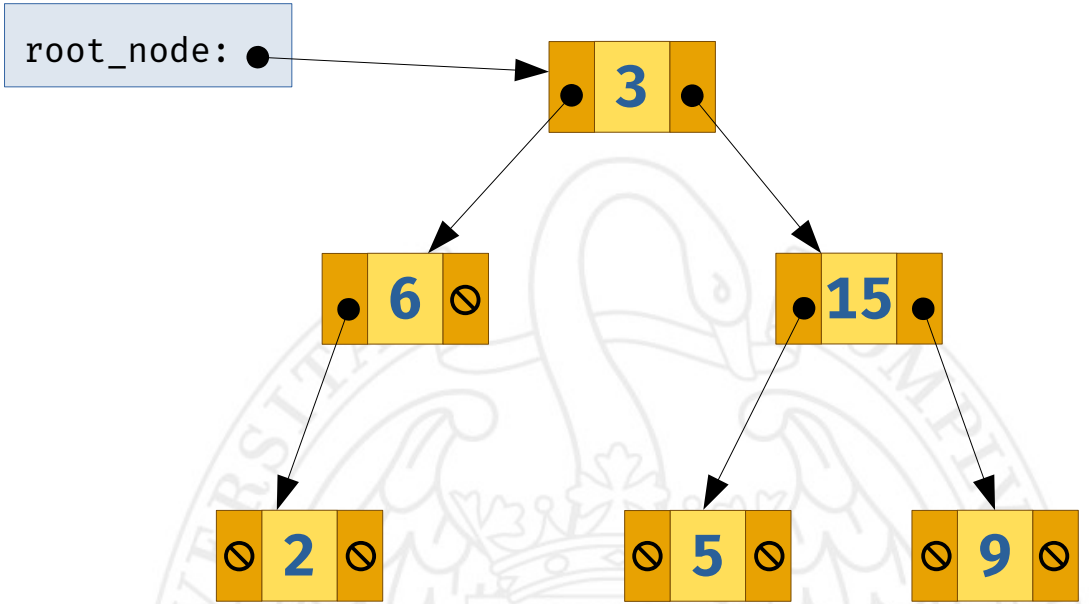
Representando árboles binarios

```
struct TreeNode {
  T elem;
  TreeNode *left, *right;

  TreeNode(const TreeNode *left,
           const T &elem,
           const TreeNode *right)
    : elem(elem), left(left),
      right(right) { }
};
```



La clase BinTree

```
template<class T>
class BinTree {
public:
    …
private:
  struct TreeNode { … }
  TreeNode *root_node;
};
```



En el caso en que el árbol es vacío:

`root_node == nullptr`

Operaciones básicas

Operaciones en el TAD Árbol Binario

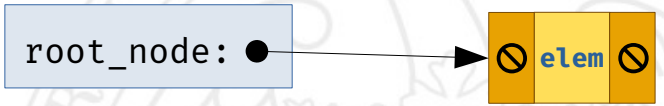
- Constructoras:
  - Crear un árbol vacío: ***create_empty***.
  - Crear una hoja: ***create_leaf***.
  - Crear un árbol a partir de una raíz y dos hijos: ***create_tree***.
- Observadoras:
  - Determinar si el árbol es vacío: ***empty***.
  - Obtener la raíz si el árbol no es vacío: ***root***.
  - Obtener el hijo izquierdo, si existe: ***left***.
  - Obtener el hijo derecho, si existe: ***right***.

Interfaz de la clase `BinTree`

```
template<class T>
class BinTree {
public:
  BinTree();
  BinTree(const T &elem);
  BinTree(const BinTree &left, const T &elem, const BinTree &right);

  const T & root() const;
  BinTree left() const;
  BinTree right() const;
  bool empty() const;

private:
  struct TreeNode { … }
  TreeNode *root_node;
};
```

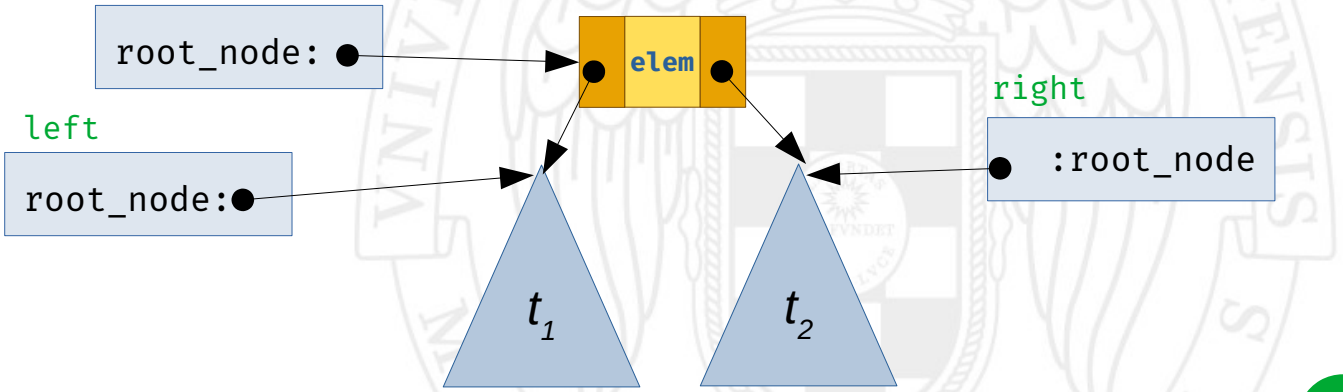
Creación de árboles

```
template<class T>
class BinTree {
public:
  BinTree(): root_node(nullptr) { }

  BinTree(const T &elem)
    : root_node(new TreeNode(nullptr, elem, nullptr)) { }



private:
  struct TreeNode { … }
  TreeNode *root_node;
};
```



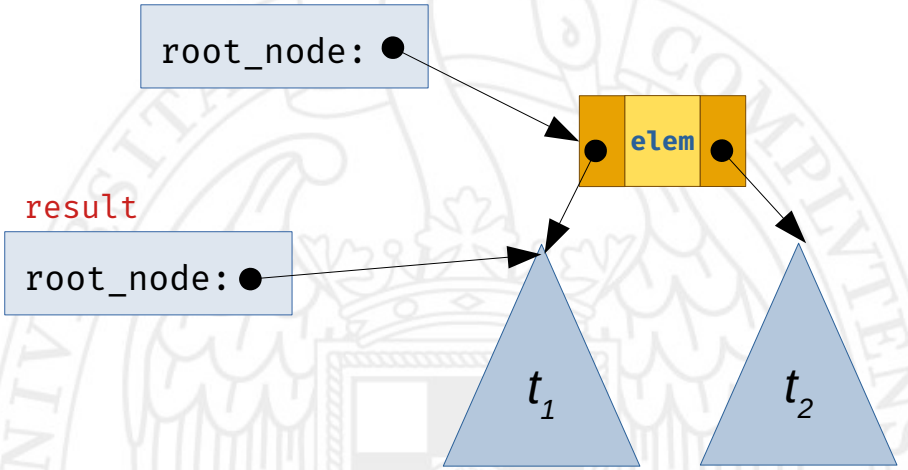
Creación de árboles

```
template<class T>
class BinTree {
public:
  BinTree(): root_node(nullptr) { }

  BinTree(const T &elem)
    : root_node(new TreeNode(nullptr, elem, nullptr)) { }

  BinTree(const BinTree &left, const T &elem, const BinTree &right)
    : root_node(new TreeNode(left.root_node, elem, right.root_node)) { }

private:
  struct TreeNode { … }
  TreeNode *root_node;
};
```



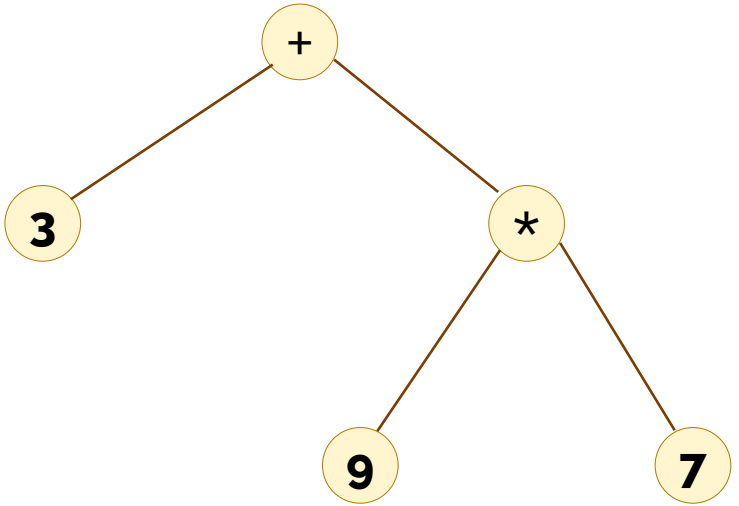
Operaciones observadoras

```
template<class T>
class BinTree {
public:
   ...
  const T & root() const {
    assert(root_node ≠ nullptr);
    return root_node→elem;
  }

  BinTree left() const {
    assert (root_node ≠ nullptr);
    BinTree result;
    result.root_node = root_node→left;
    return result;
  }

  bool empty() const {
    return root_node == nullptr;
  }

};
```



E/S de árboles

Representación textual de un árbol

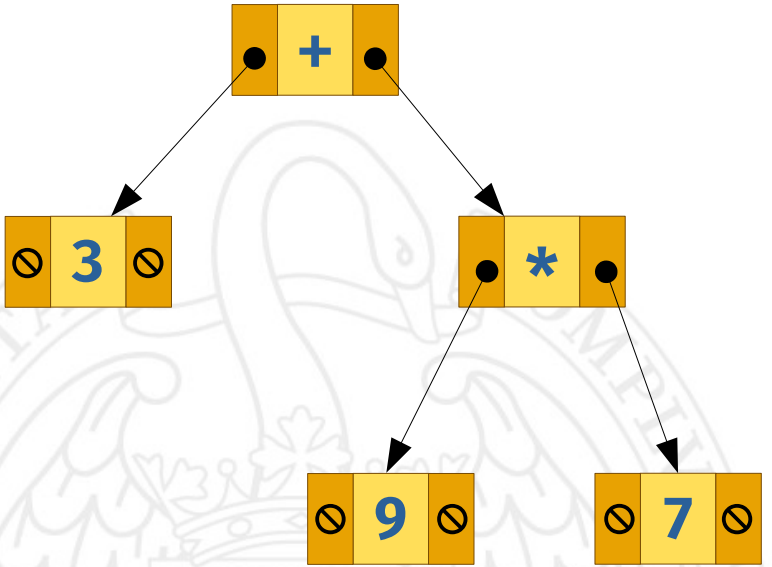



```
((. 3 .) + ((. 9 .) * (. 7 .)))
```

- Árbol vacío: .
- Árbol no vacío: (*hijo-iz raiz hijo-dr*)

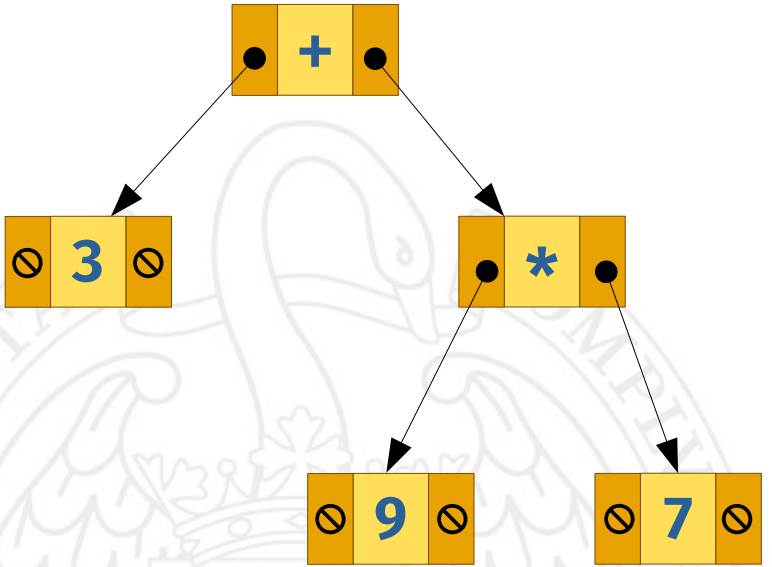
Mostrar un árbol por pantalla

```
template<class T>
class BinTree {
   ...

private:
  struct TreeNode { … }
  TreeNode *root_node;

  static void display_node(const TreeNode *root,
                           std::ostream &out) {
    if (root == nullptr) {
      out << ".";
    } else {
      out << "(";
      display_node(root→left, out);
      out << " " << root→elem << " ";
      display_node(root→right, out);
      out << ")";
    }
  }
};
```



Mostrar un árbol por pantalla

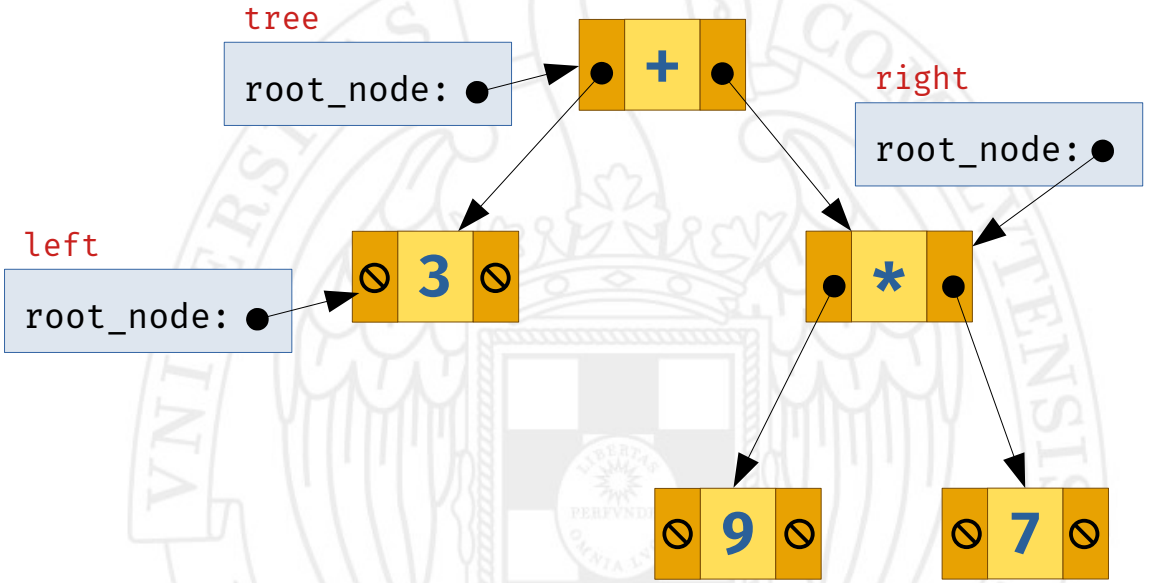
```
template<class T>
class BinTree {
public:
   ...

  void display(std::ostream &out) const {
    display_node(root_node, out);
  }

private:
  TreeNode *root_node;

};

template<typename T>
std::ostream & operator<<(std::ostream &out, const BinTree<T> &tree) {
  tree.display(out);
  return out;
}
```



Ejemplo

```
int main() {
  BinTree<std::string> left("3");
  BinTree<std::string> right(BinTree<std::string>("9"), "*", BinTree<std::string>("7"));
  BinTree<std::string> tree(left, "+", right);

  std::cout << tree << std::endl;

  return 0;
}
```

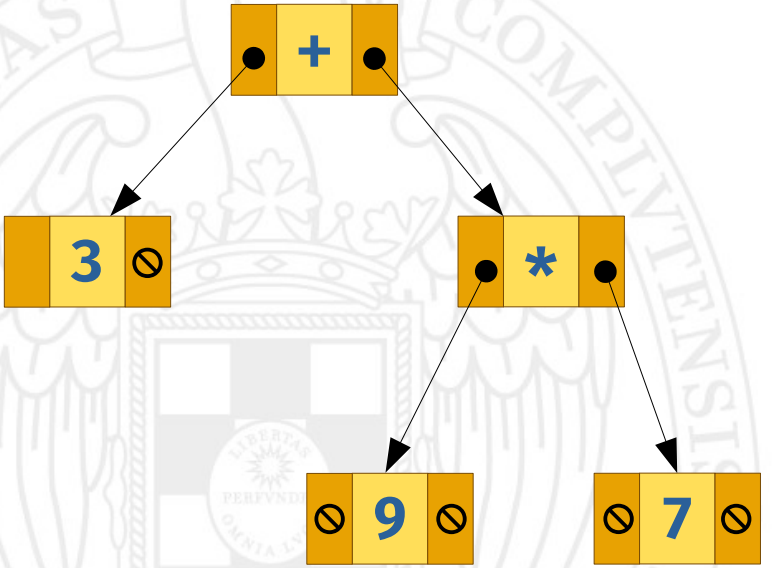
```
((. 3 .) + ((. 9 .) * (. 7 .)))
```



Ejemplo

```
int main() {
  BinTree<std::string> tree = {{"3"} , "+", {{"9"} , "*", {"7"}}};

  std::cout << tree << std::endl;

  return 0;
}
```

```
((. 3 .) + ((. 9 .) * (. 7 .)))
```



Leer un árbol por entrada

```
template<typename T>
BinTree<T> read_tree(std::istream &in) {
  char c;
  in >> c;
  if (c == '.') {
    return BinTree<T>();
  } else {
    assert (c == '(');
    BinTree<T> left = read_tree<T>(in);
    T elem;
    in >> elem;
    BinTree<T> right = read_tree<T>(in);
    in >> c;
    assert (c == ')');
    BinTree<T> result(left, elem, right);
    return result;
  }
}
```

```
((. 3 .) + ((. 9 .) * (. 7 .)))
```

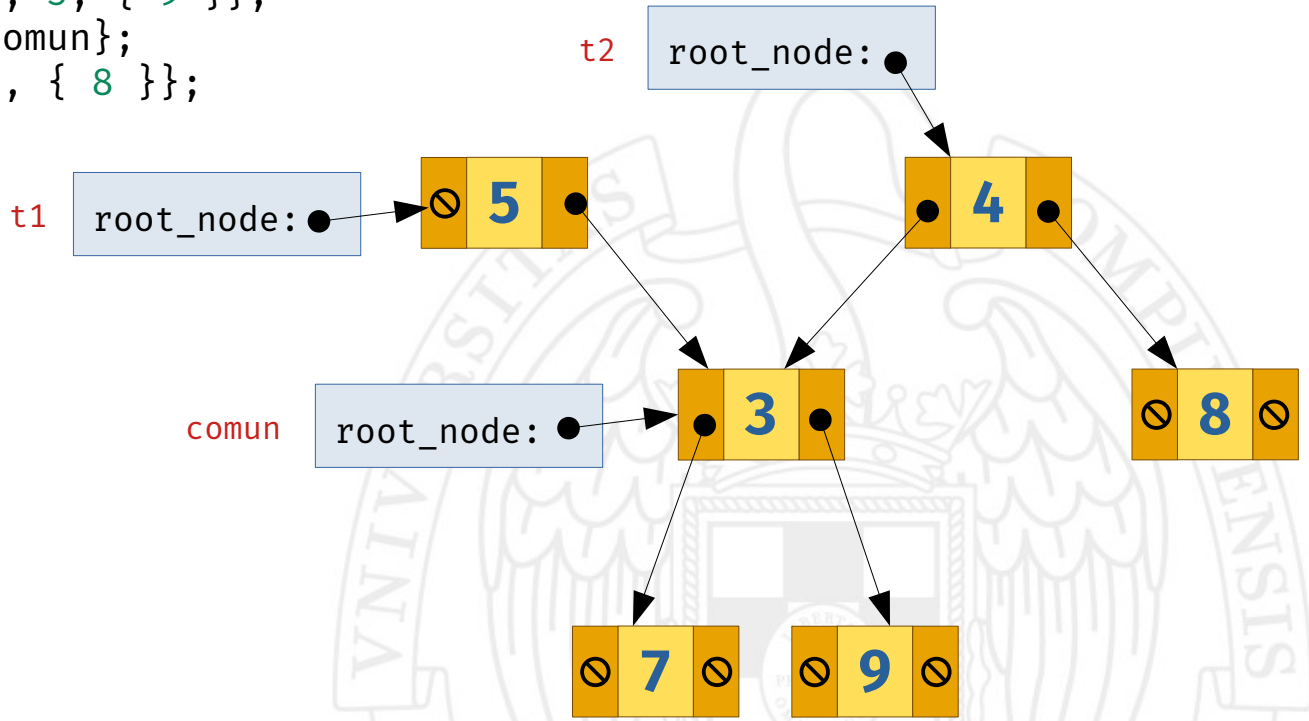
Destrucción de memoria

Problema importante

- ¡No estamos liberando la memoria ocupada por los nodos!
- Hay que hacerlo con cuidado...

El problema de la compartición en árboles

```
BinTree<int> comun = {{ 7 }, 3, { 9 }};
BinTree<int> t1 = {{}, 5, comun};
BinTree<int> t2 = {comun, 4, { 8 }};
```



¿Cómo liberamos la memoria ocupada por los nodos?

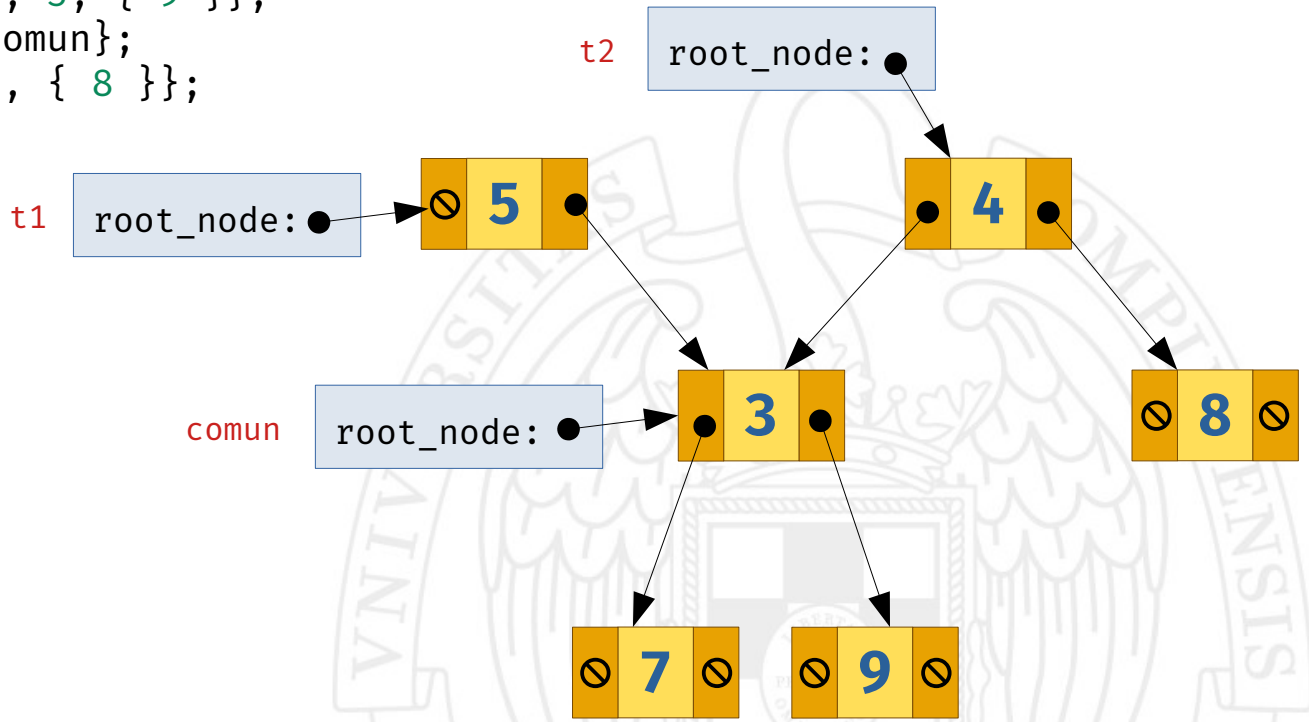
Intento fallido de destructor

```
template<class T>
class BinTree {
public:
  ...
   ~BinTree() {
     delete_with_children(root_node);
   }

private:

   static void delete_with_children(const TreeNode *node) {
     if (node ≠ nullptr) {
       delete_with_children(node→left);
       delete_with_children(node→right);
       delete node;
     }
   }
};
```

En nuestro ejemplo...

```
BinTree<int> comun = {{ 7 }, 3, { 9 }};
BinTree<int> t1 = {{}, 5, comun};
BinTree<int> t2 = {comun, 4, { 8 }};
```

¿Qué pasa cuando `t1`, `t2` y `comun` salen de ámbito?



Soluciones

Para evitar liberar nodos más de una vez, podemos optar por alguna de las siguientes alternativas:

1) *Evitar la compartición de nodos entre árboles.* ← **Ejercicio**

   Cada vez que construyamos un árbol a partir de otros, debemos hacer una copia de los nodos de estos últimos.

2) *Aceptar la compartición de nodos entre árboles.* ← **Otro vídeo**

   Utilizamos mecanismos de conteo de referencias para saber cuándo liberar la memoria.

ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS ARBORESCENTES

Compartición en árboles binarios

Manuel Montenegro Montes
Departamento de Sistemas Informáticos y Computación
Facultad de Informática – Universidad Complutense de Madrid

Definición actual de `TreeNode`

```
struct TreeNode {
  T elem;
  TreeNode *left, *right;

  TreeNode(const TreeNode *left,
           const T &elem,
           const TreeNode *right)
    : elem(elem), left(left),
      right(right) { }
};
```

- Vamos a sustituir los punteros estándar de C++ por *smart pointers*.
- En lugar de

  `TreeNode *`

  utilizamos

  `std::shared_ptr<TreeNode>`

Cambios en TreeNode

```
struct TreeNode {
  T elem;
  std::shared_ptr<TreeNode> left, right;

  TreeNode(const std::shared_ptr<TreeNode> &left,
           const T &elem,
           const std::shared_ptr<TreeNode> &right)
    : elem(elem), left(left),
      right(right) { }
};
```

Cambios en TreeNode

```
using NodePointer = std::shared_ptr<TreeNode>;

struct TreeNode {
  T elem;
  NodePointer left, right;

  TreeNode(const NodePointer &left,
           const T &elem,
           const NodePointer &right)
    : elem(elem), left(left),
      right(right) { }
};
```

Cambios en BinTree

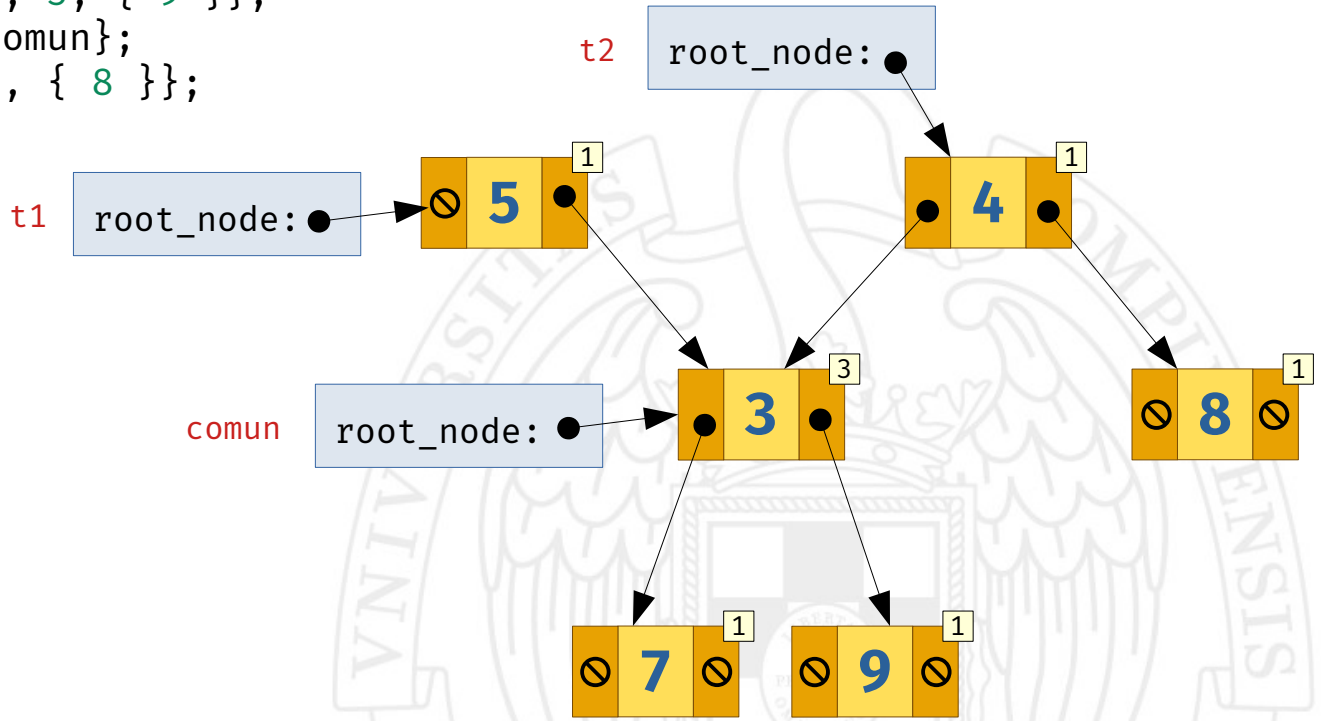
```
template<class T>
class BinTree {
public:

  BinTree(): root_node(nullptr) { }
  BinTree(const T &elem)
    : root_node(std::make_shared<TreeNode>(nullptr, elem, nullptr)) { }
  BinTree(const BinTree &left, const T &elem, const BinTree &right)
    : root_node(std::make_shared<TreeNode>(left.root_node, elem, right.root_node)) { }

  …

private:
  using NodePointer = std::shared_ptr<TreeNode>;
  struct TreeNode { … }

  NodePointer root_node;

  static void display_node(const NodePointer &root, std::ostream &out) { … }
};
```

No necesitamos...

- Destructor
  - Cuando se elimina un objeto `BinTree` se llama automáticamente al destructor de `root_node`.
  - El destructor de `root_node` decrementa el contador de referencias del nodo raíz, y lo libera, en caso de llegar a 0.
- Constructor de copia
  - El constructor de copia por defecto `BinTree` nos sirve, ya que llama al constructor de copia de `root_node`.
  - El constructor de copia de `root_node` incrementa el contador de referencias del nodo raíz.

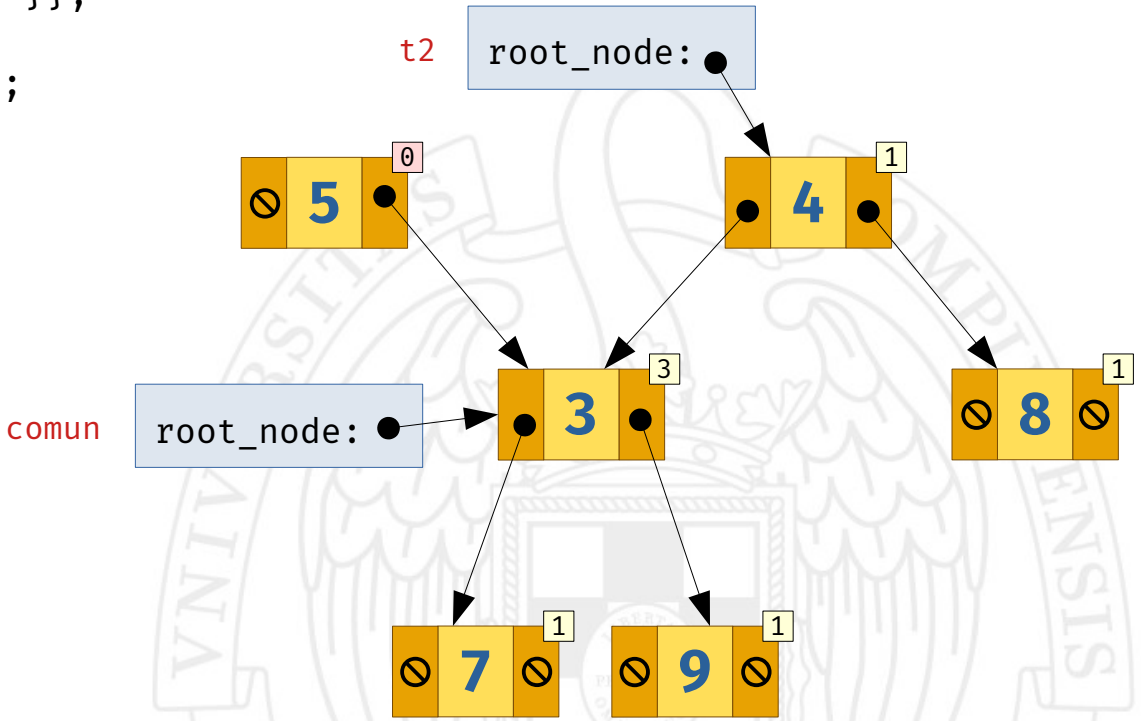
Ejemplo

```
BinTree<int> comun = {{ 7 }, 3, { 9 }};
BinTree<int> t1 = {{}, 5, comun};
BinTree<int> t2 = {comun, 4, { 8 }};
```



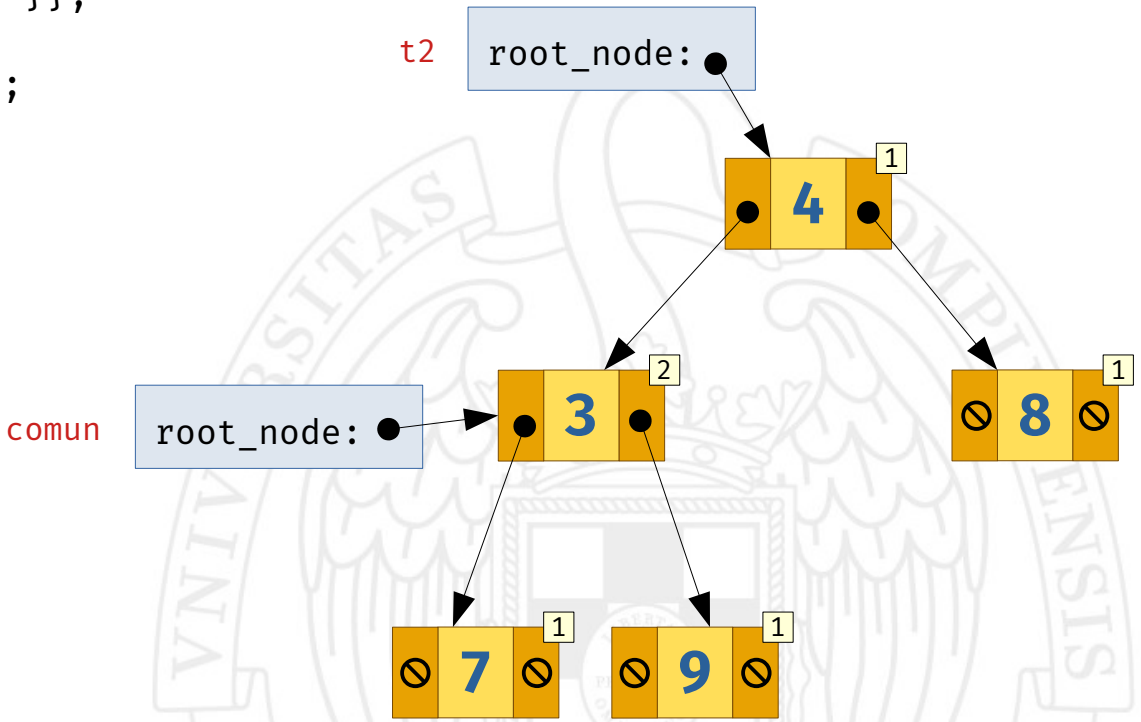
Ejemplo

```
BinTree<int> comun = {{ 7 }, 3, { 9 }};
BinTree<int> t1 = {{}, 5, comun};
BinTree<int> t2 = {comun, 4, { 8 }};
```



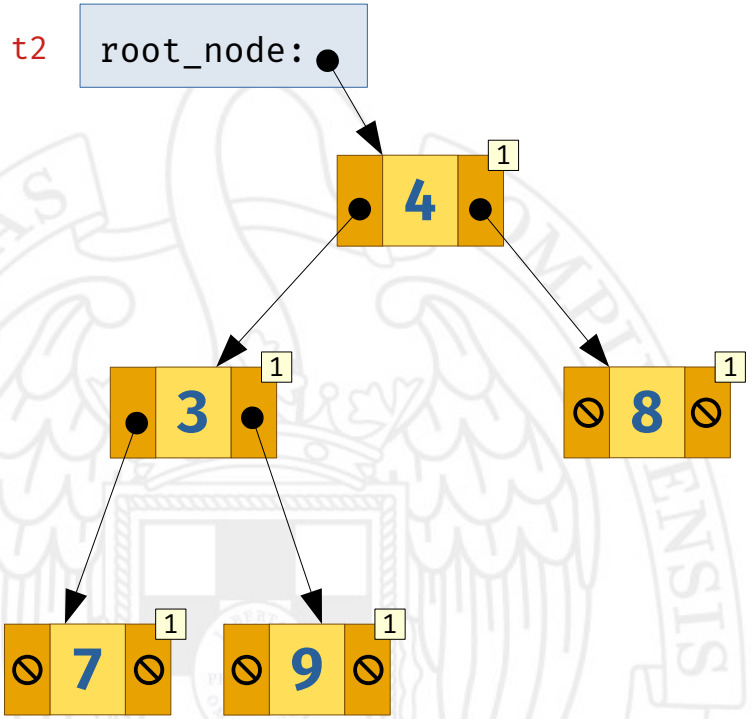
Ejemplo

```
BinTree<int> comun = {{ 7 }, 3, { 9 }};
BinTree<int> t1 = {{}, 5, comun};
BinTree<int> t2 = {comun, 4, { 8 }};
```



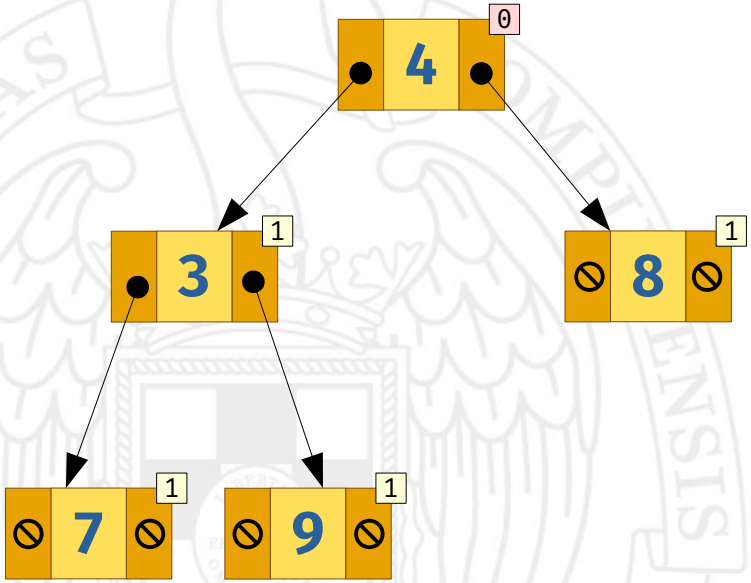
Ejemplo

```
BinTree<int> comun = {{ 7 }, 3, { 9 }};
BinTree<int> t1 = {{}, 5, comun};
BinTree<int> t2 = {comun, 4, { 8 }};
```



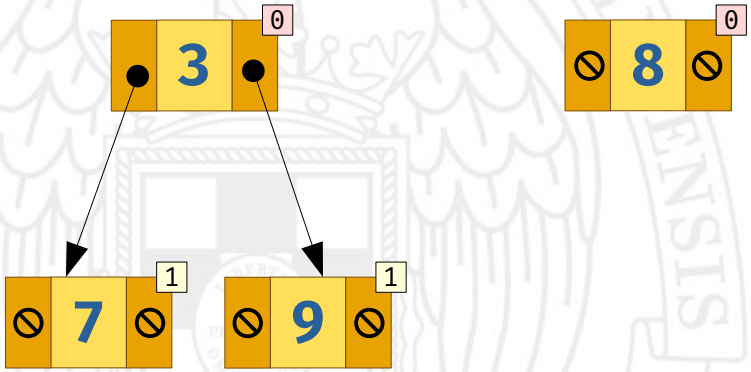
Ejemplo

```
BinTree<int> comun = {{ 7 }, 3, { 9 }};
BinTree<int> t1 = {{}, 5, comun};
BinTree<int> t2 = {comun, 4, { 8 }};
```



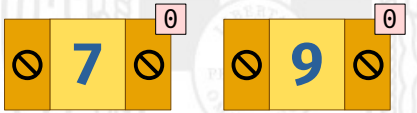
Ejemplo

```
BinTree<int> comun = {{ 7 }, 3, { 9 }};
BinTree<int> t1 = {{}, 5, comun};
BinTree<int> t2 = {comun, 4, { 8 }};
```



Ejemplo

```
BinTree<int> comun = {{ 7 }, 3, { 9 }};
BinTree<int> t1 = {{}, 5, comun};
BinTree<int> t2 = {comun, 4, { 8 }};
```



Ejemplo

```
BinTree<int> comun = {{ 7 }, 3, { 9 }};
BinTree<int> t1 = {{}, 5, comun};
BinTree<int> t2 = {comun, 4, { 8 }};
```

ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS ARBORESCENTES

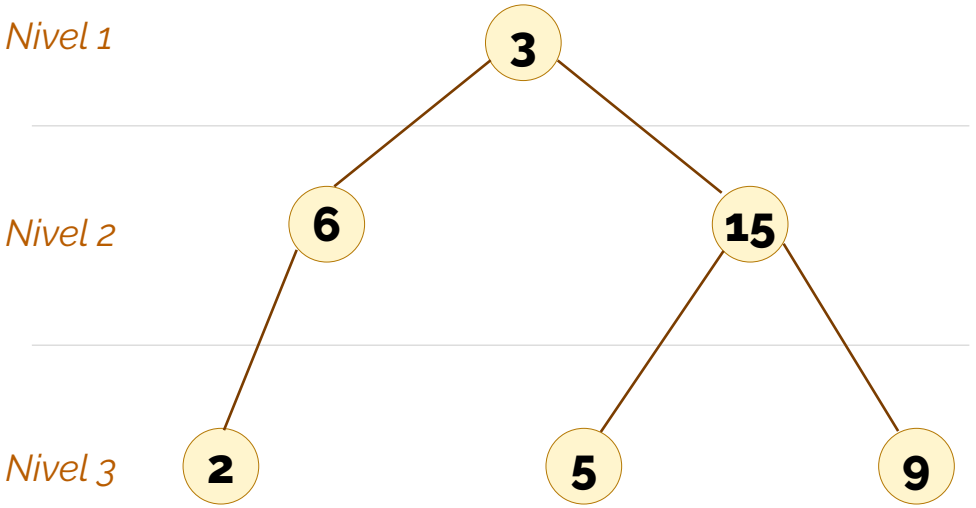
Funciones sobre árboles binarios

Manuel Montenegro Montes
Departamento de Sistemas Informáticos y Computación
Facultad de Informática – Universidad Complutense de Madrid

Recordatorio: interfaz de BinTree<T>

```
template<class T>
class BinTree {
public:
  BinTree();
  BinTree(const T &elem);
  BinTree(const BinTree &left, const T &elem, const BinTree &right);

  const T & root() const;
  BinTree left() const;
  BinTree right() const;
  bool empty() const;

private:
  ...
};
```

Recordatorio: altura de un árbol binario

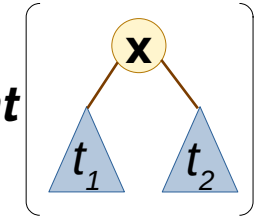


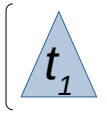
- La **altura** de un árbol es el máximo de los niveles de los nodos.

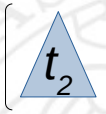
Definición recursiva de altura

- Es posible definir recursivamente la altura de un árbol binario:

height(—) = 0

height 

= 1 + max(***height*** 

, ***height*** 

)

Función height

$$height(—) = 0$$

$$height\left(\begin{array}{c} x \\ t_1 \quad t_2 \end{array}\right) = 1 + \max(\, height(t_1), height(t_2)\,)$$

```
template<typename T>
int height(const BinTree<T> &tree) {
  if (tree.empty()) {
    return 0;
  } else {
    return 1 + std::max(height(tree.left()), height(tree.right()));
  }
}
```

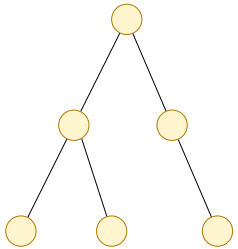
Coste en tiempo

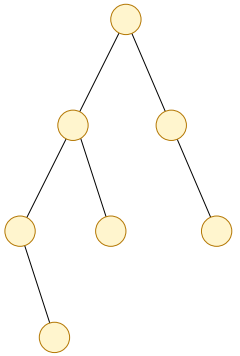
```
template<typename T>
int height(const BinTree<T> &tree) {
  if (tree.empty()) {
    return 0;
  } else {
    return 1 + std::max(height(tree.left()), height(tree.right()));
  }
}
```

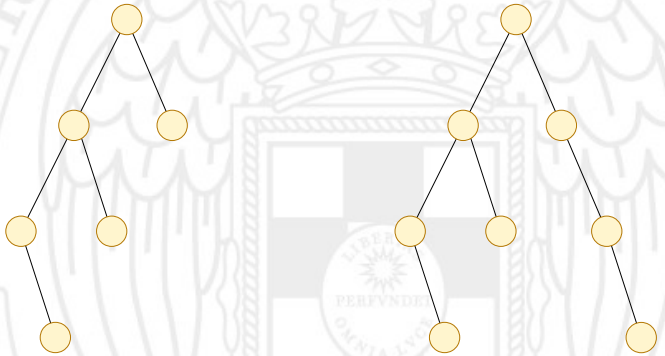
Árboles equilibrados en altura

Un árbol está **equilibrado en altura** si:

- Es el árbol vacío, o bien
- La diferencia entre las alturas de sus hijos es, como mucho, 1, y ambos están equilibrados en altura.







Definición recursiva

$$\textbf{balanced}(\text{—}) = \textit{true}$$

$$\textbf{balanced}\left(\begin{array}{c} x \\ t_1 \quad t_2 \end{array}\right) = \textbf{balanced}(t_1) \wedge \textbf{balanced}(t_2) \wedge \left|\textbf{height}(t_1) - \textbf{height}(t_2)\right| \leq 1$$

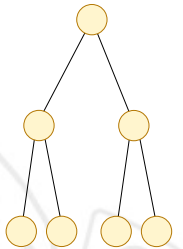
Función balanced

```
template<typename T>
bool balanced(const BinTree<T> &tree) {
  if (tree.empty()) {
    return true;
  } else {
    bool bal_left = balanced(tree.left());
    bool bal_right = balanced(tree.right());
    int height_left = height(tree.left());
    int height_right = height(tree.right());
    return bal_left && bal_right && abs(height_left - height_right) ⩽ 1;
  }
}
```

¿Cuál es el coste en tiempo?

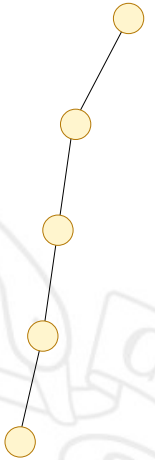
Función balanced: caso mejor

```
template<typename T>
bool balanced(const BinTree<T> &tree) {
  if (tree.empty()) {
    return true;
  } else {
    bool bal_left = balanced(tree.left());
    bool bal_right = balanced(tree.right());
    int height_left = height(tree.left());
    int height_right = height(tree.right());
    return bal_left && bal_right && abs(height_left - height_right) ⩽ 1;
  }
}
```

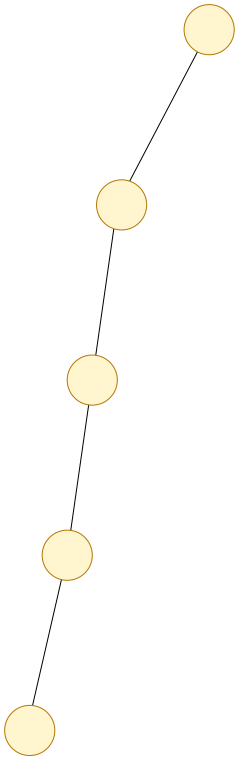


Función balanced: caso peor

```
template<typename T>
bool balanced(const BinTree<T> &tree) {
  if (tree.empty()) {
    return true;
  } else {
    bool bal_left = balanced(tree.left());
    bool bal_right = balanced(tree.right());
    int height_left = height(tree.left());
    int height_right = height(tree.right());
    return bal_left && bal_right && abs(height_left - height_right) ⩽ 1;
  }
}
```



Problema de llamar a `height`



¿Cómo solucionarlo?

- Implementando una función auxiliar recursiva que **simultáneamente** calcule la altura y determine si un árbol está equilibrado.
- Esta función devuelve dos valores:
  - `balanced` (`bool`) – si el árbol está equilibrado o no.
  - `height` (`int`) – altura del árbol.
- La función `balanced_height` devolverá ambos valores como parámetros de salida.

Función balanced_height

```
template<typename T>
void balanced_height(const BinTree<T> &tree, bool &balanced, int &height) {
  if (tree.empty()) {
    balanced = true;
    height = 0;
  } else {
    bool bal_left, bal_right;
    int height_left, height_right;
    balanced_height(tree.left(), bal_left, height_left);
    balanced_height(tree.right(), bal_right, height_right);
    balanced = bal_left && bal_right && abs(height_left - height_right) ⩽ 1;
    height = 1 + std::max(height_left, height_right);
  }
}
```

Función balanced

```
template<typename T>
bool balanced(const BinTree<T> &tree) {
  bool balanced;
  int height;
  balanced_height(tree, balanced, height);
  return balanced;
}
```

Moraleja

- La mayoría de las funciones que operan sobre árboles son recursivas.
- En muchos casos estas funciones deben devolver valores auxiliares adicionales para evitar costes en tiempo elevados.

ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS ARBORESCENTES

Recorridos de árboles binarios

Manuel Montenegro Montes
Departamento de Sistemas Informáticos y Computación
Facultad de Informática – Universidad Complutense de Madrid

¿Qué es un recorrido?

- **Recorrer** un árbol significa visitar los nodos de un árbol, de modo que cada nodo es visitado exactamente una vez.
- **Visitar** un nodo significa realizar una acción específica, que puede depender del valor contenido dentro de ese nodo.
  - Imprimir por pantalla el valor del nodo. ← **Comenzaremos aquí**
  - Sumar el valor del nodo a una variable externa.
  - Escribir el valor del nodo en un fichero.
  - Incrementar un contador externo.

Tipos de recorridos

- Recorrido en profundidad

  Depth First Search (DFS)

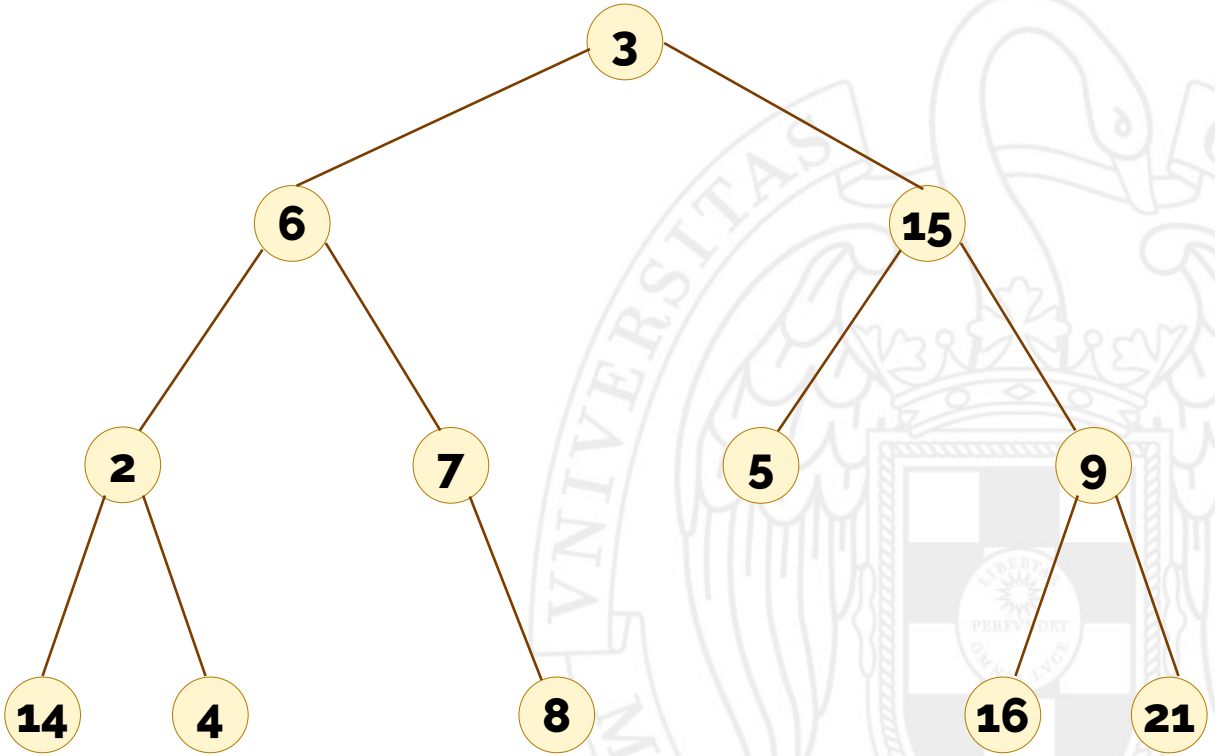
  - Preorden
  - Inorden
  - Postorden

- Recorrido en anchura

  Breadth First Search (BFS)

Recorridos en profundidad

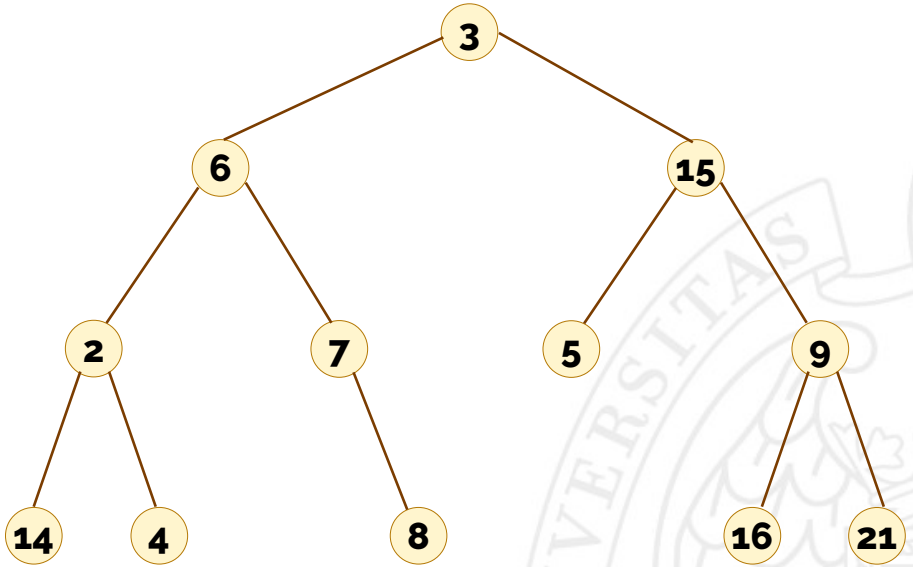
- Se explora completamente un hijo antes de pasar al siguiente.



Recorridos en profundidad

- Se explora completamente un hijo antes de pasar al siguiente.
  - **Preorden:** Visitar raíz, luego recorrer hijo izquierdo, luego recorrer hijo derecho.

Recorrido en preorden

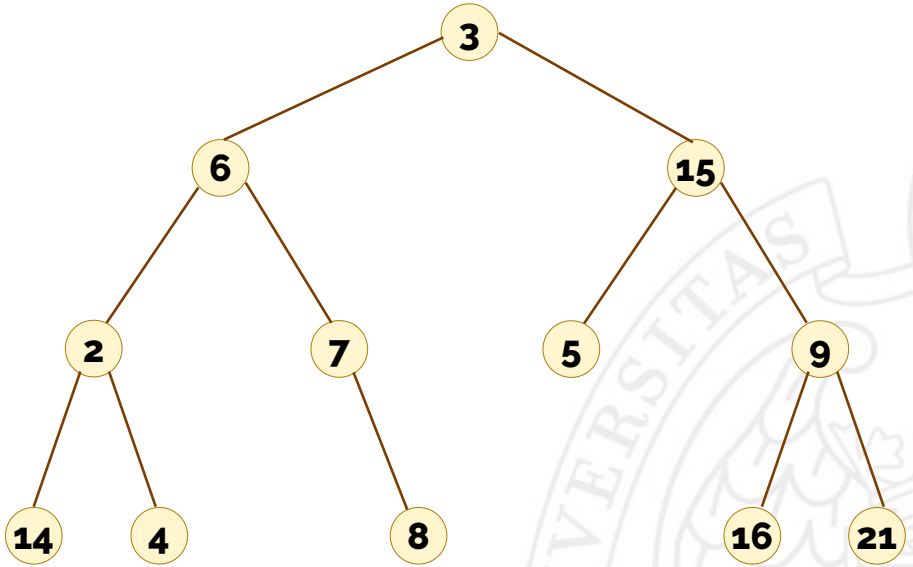


3 6 2 14 4 7 8 15 5 9 16 21

Recorridos en profundidad

- Se explora completamente un hijo antes de pasar al siguiente.
  - **Preorden:** Visitar raíz, luego recorrer hijo izquierdo, luego recorrer hijo derecho.
  - **Inorden:** Recorrer hijo izquierdo, visitar raíz, luego recorrer hijo derecho.

Recorrido en inorden

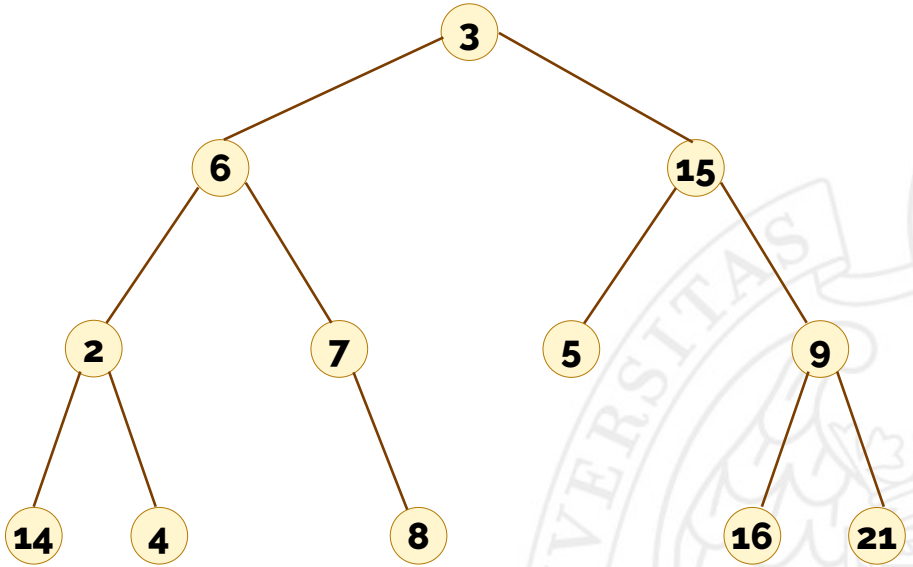


14 2 4 6 7 8 3 5 15 16 9 21

Recorridos en profundidad

- Se explora completamente un hijo antes de pasar al siguiente.
  - **Preorden:** Visitar raíz, luego recorrer hijo izquierdo, luego recorrer hijo derecho.
  - **Inorden:** Recorrer hijo izquierdo, visitar raíz, luego recorrer hijo derecho.
  - **Postorden:** Recorrer hijo izquierdo, luego recorrer hijo derecho, luego visitar raíz.

Recorrido en postorden



14 4 2 8 7 6 5 16 21 9 15 3

Recorridos en profundidad

- Se explora completamente un hijo antes de pasar al siguiente.
  - **Preorden:** Visitar raíz, luego recorrer hijo izquierdo, luego recorrer hijo derecho.
  - **Inorden:** Recorrer hijo izquierdo, visitar raíz, luego recorrer hijo derecho.
  - **Postorden:** Recorrer hijo izquierdo, luego recorrer hijo derecho, luego visitar raíz.

Tipos de recorridos

- Recorrido en profundidad

  Depth First Search (DFS)

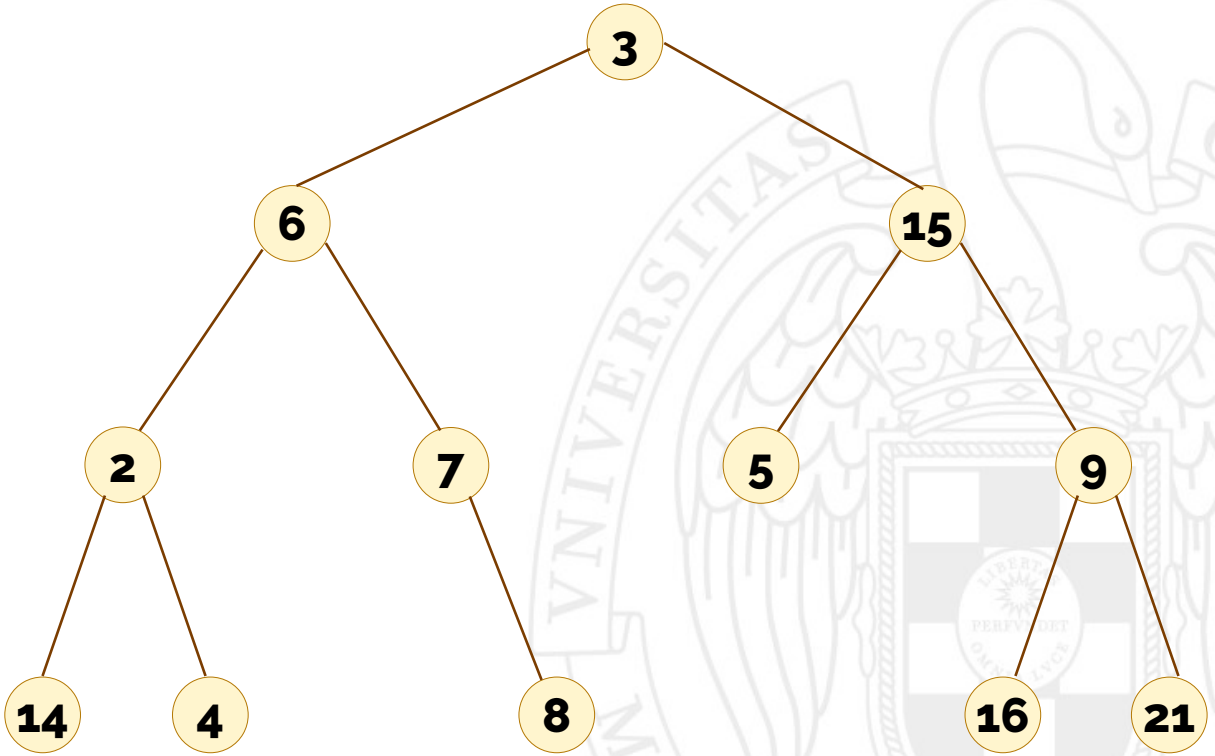
  - Preorden
  - Inorden
  - Postorden

- Recorrido en anchura

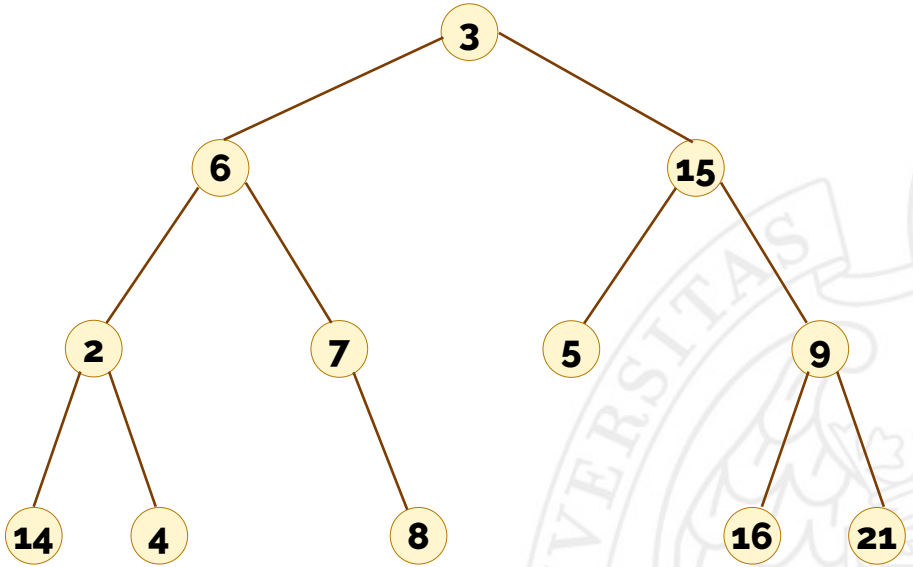
  Breadth First Search (BFS)

Recorridos en anchura (*por niveles*)

- Se explora completamente un nivel antes de pasar al siguiente.



Recorrido en anchura (*por niveles*)



3 6 15 2 7 5 9 14 4 8 16 21

ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS ARBORESCENTES

Implementando recorridos en profundidad (*DFS*)

Manuel Montenegro Montes
Departamento de Sistemas Informáticos y Computación
Facultad de Informática – Universidad Complutense de Madrid

Tipos de recorridos

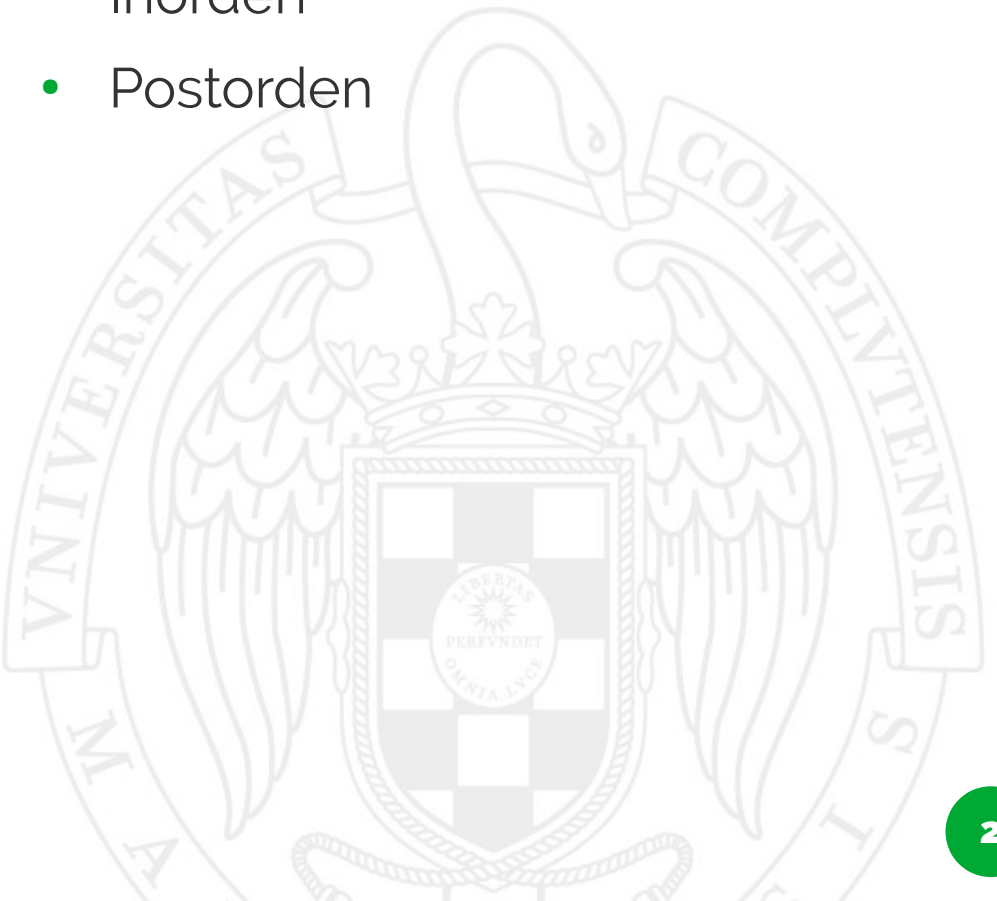
- Recorrido en profundidad

  Depth First Search (DFS)

  - Preorden
  - Inorden
  - Postorden

- Recorrido en anchura

  Breadth First Search (BFS)



Recordatorio: interfaz de BinTree<T>

```
template<class T>
class BinTree {
public:
  BinTree();
  BinTree(const T &elem);
  BinTree(const BinTree &left, const T &elem, const BinTree &right);

  const T & root() const;
  BinTree left() const;
  BinTree right() const;
  bool empty() const;

private:
  ...
};
```

Recordatorio: interfaz de `BinTree<T>`

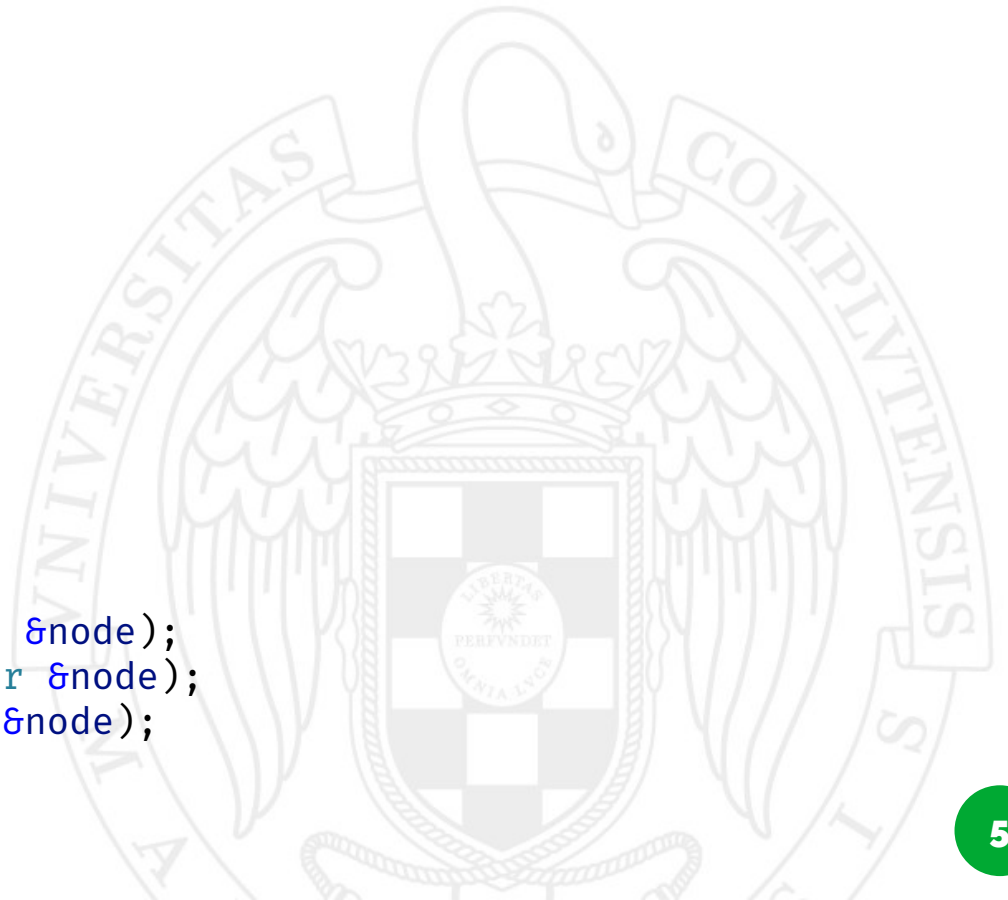
```
template<class T>
class BinTree {
public:
  // ...
  void preorder() const;
  void inorder() const;
  void postorder() const;

private:
  ...
};
```

- Añadimos tres nuevos métodos a `BinTree<T>`.

Recordatorio: interfaz de `BinTree<T>`

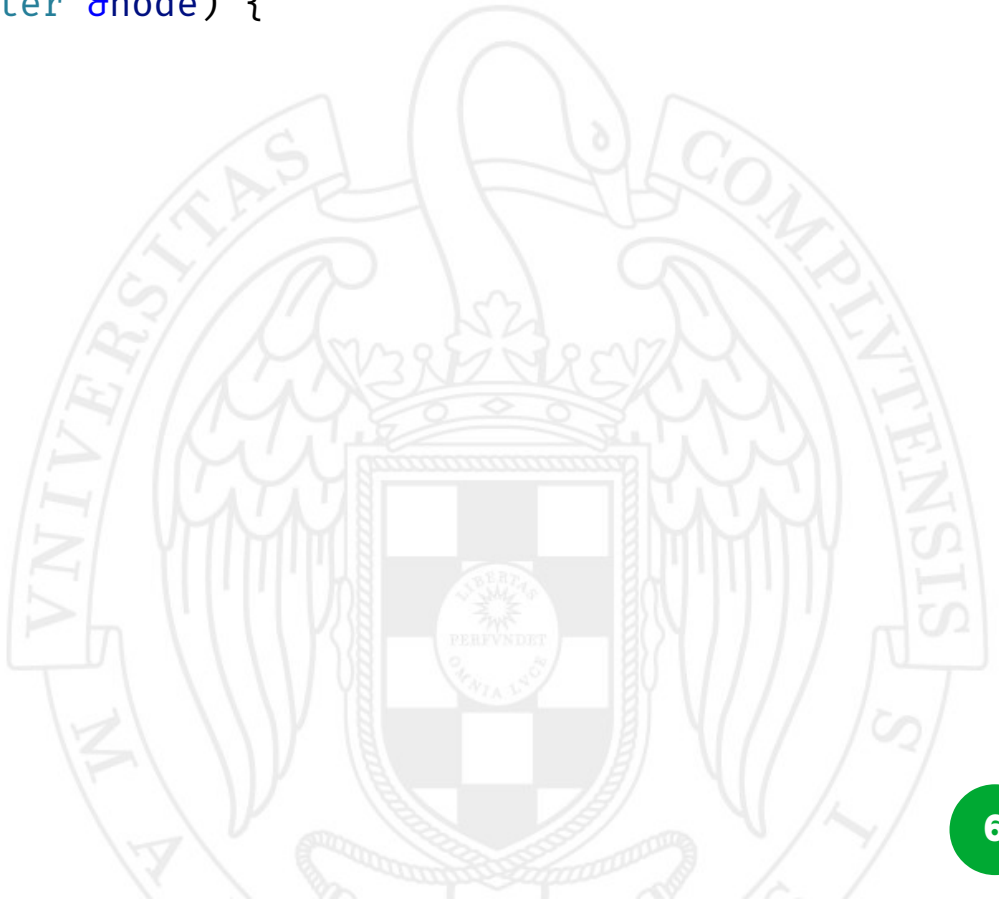
```
template<class T>
class BinTree {
public:
  // ...
  void preorder() const {
    preorder(root_node);
  }

  void inorder() const {
    inorder(root_node);
  }

  void postorder() const {
    postorder(root_node);
  }

private:
  static void preorder(const NodePointer &node);
  static void postorder(const NodePointer &node);
  static void inorder(const NodePointer &node);
...
};
```

- Estos métodos harán uso de otros tres métodos privados auxiliares.



Método auxiliar preorder

```
template<typename T>
void BinTree<T>::preorder(const NodePointer &node) {
  if (node ≠ nullptr) {
    std::cout << node→elem << " ";
    preorder(node→left);
    preorder(node→right);
  }
}
```

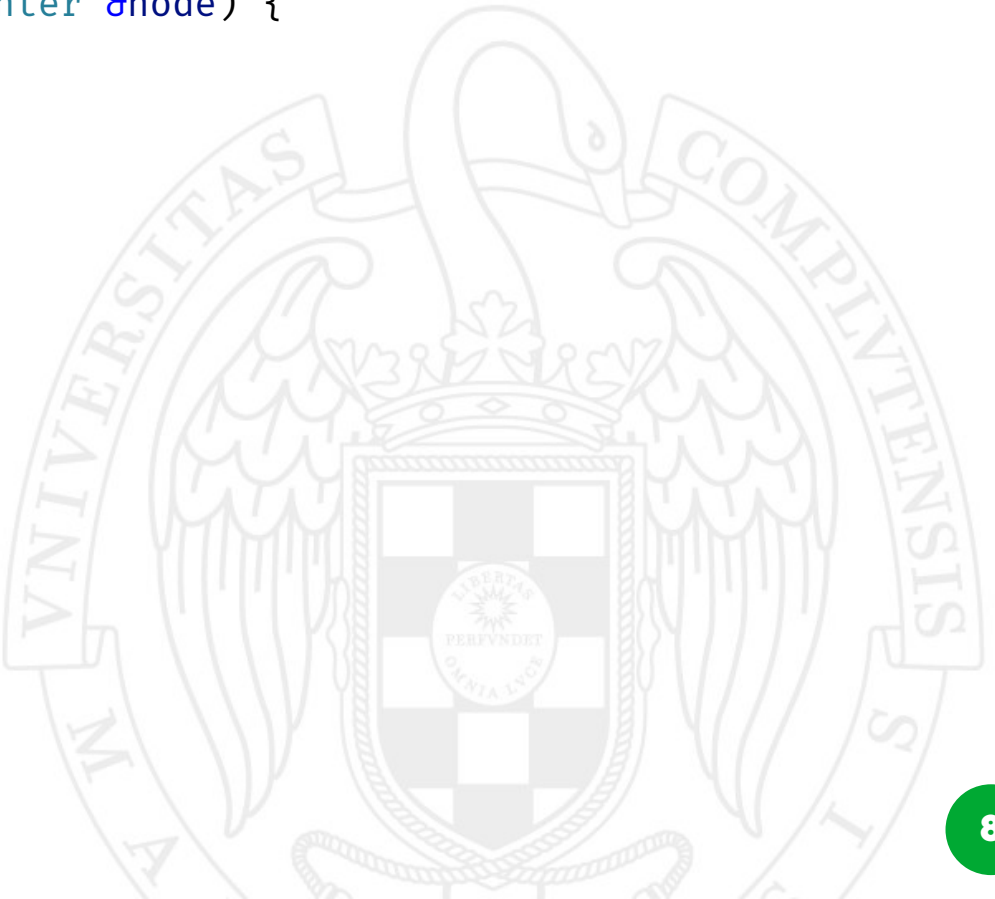


Método auxiliar `inorder`

```
template<typename T>
void BinTree<T>::inorder(const NodePointer &node) {
  if (node ≠ nullptr) {
    inorder(node→left);
    std::cout << node→elem << " ";
    inorder(node→right);
  }
}
```

Método auxiliar postorder

```
template<typename T>
void BinTree<T>::postorder(const NodePointer &node) {
  if (node ≠ nullptr) {
    postorder(node→left);
    postorder(node→right);
    std::cout << node→elem << " ";
  }
}
```



Ejemplo

```
int main() {
  BinTree<int> tree = {{{ 9 }, 4, { 5 }}, 7, {{ 10 }, 4, { 6 }}};

  std::cout << "Recorrido en preorden: " << std::endl;
  tree.preorder();
  std::cout << std::endl;

  std::cout << "Recorrido en inorden: " << std::endl;
  tree.inorder();
  std::cout << std::endl;

  std::cout << "Recorrido en postorden: " << std::endl;
  tree.postorder();
  std::cout << std::endl;

  return 0;
}
```

```
Recorrido en preorden:
7 4 9 5 4 10 6
Recorrido en inorden:
9 4 5 7 10 4 6
Recorrido en postorden:
9 5 4 10 6 4 7
```

ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS ARBORESCENTES

Implementando recorridos en anchura (*BFS*)

Manuel Montenegro Montes
Departamento de Sistemas Informáticos y Computación
Facultad de Informática – Universidad Complutense de Madrid

Tipos de recorridos

- Recorrido en profundidad

  Depth First Search (DFS)

  - Preorden
  - Inorden
  - Postorden

- Recorrido en anchura

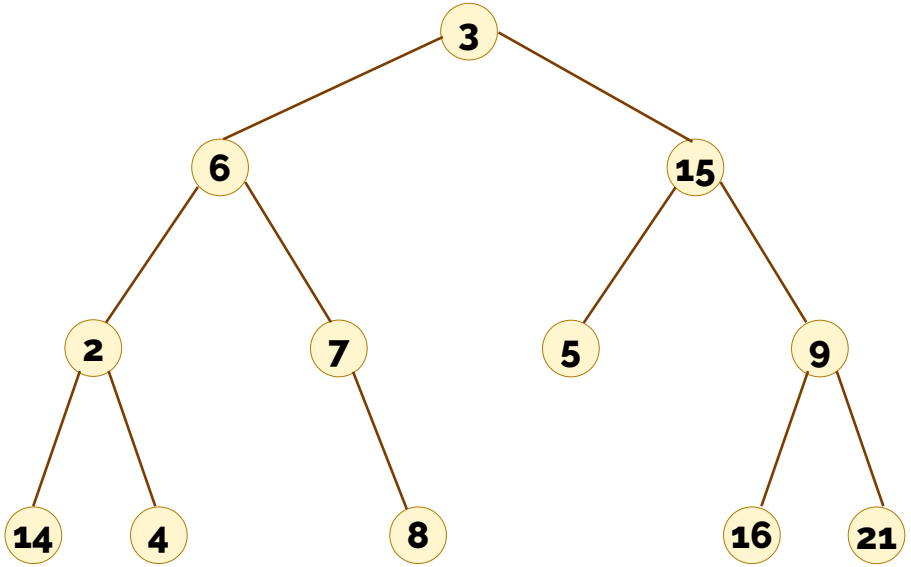
  Breadth First Search (BFS)

Nuevo método

```
template<class T>
class BinTree {
public:
  // ...
  void preorder() const;
  void inorder() const;
  void postorder() const;

  void levelorder() const;

private:
  ...
};
```

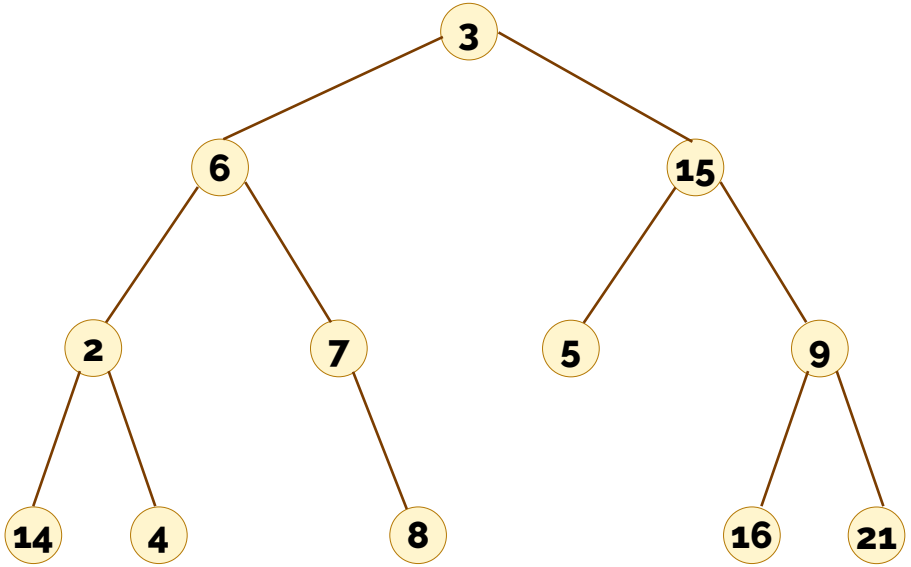
- Implementamos el recorrido en profundidad mediante un nuevo método.

Algoritmo de recorrido en anchura



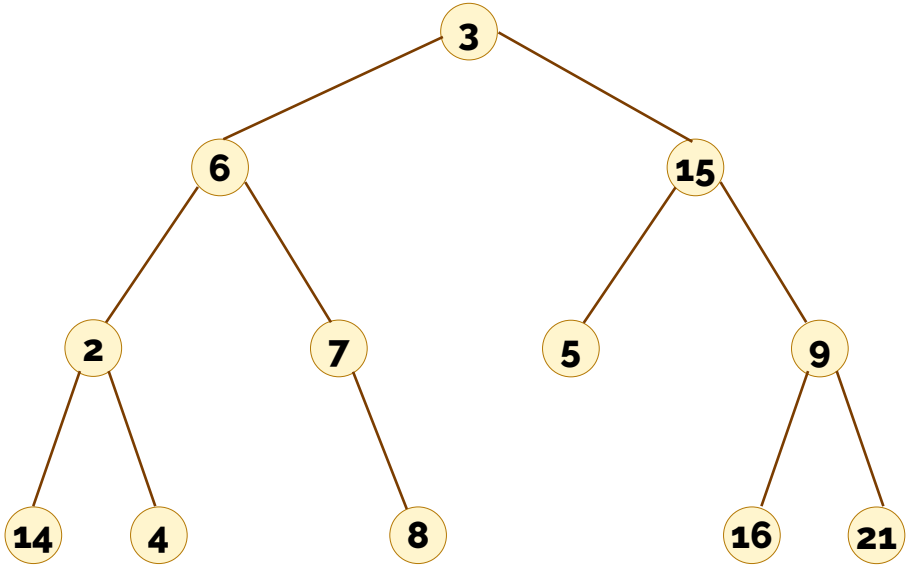
- Utilizaremos una **cola** que contiene punteros a nodos.
- Esta cola representa los nodos pendientes que nos quedan por visitar.

Algoritmo de recorrido en anchura

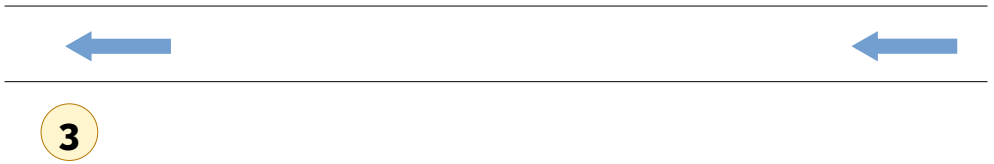


- Insertamos la raíz en la cola.
- Repetimos:

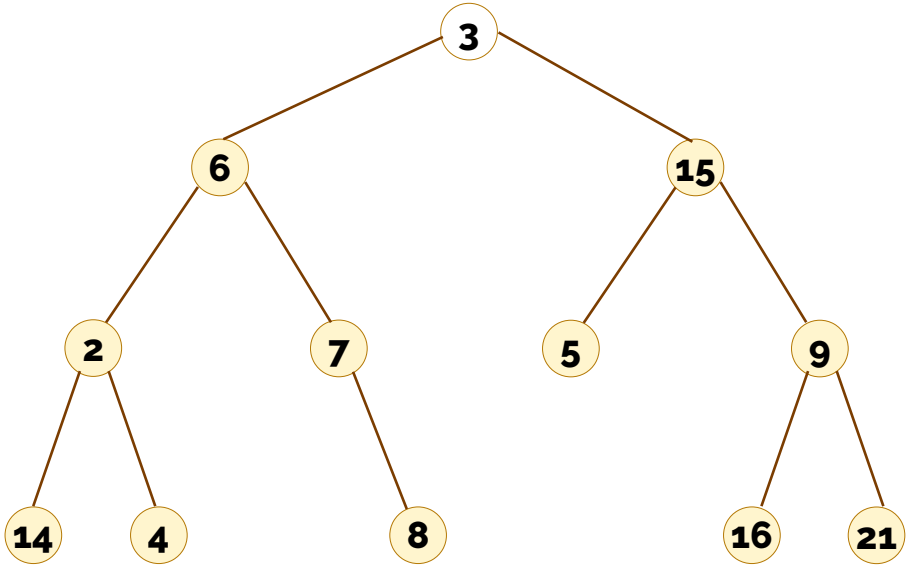
Algoritmo de recorrido en anchura



- Insertamos la raíz en la cola.
- Repetimos:
  - Sacamos nodo de la cola.

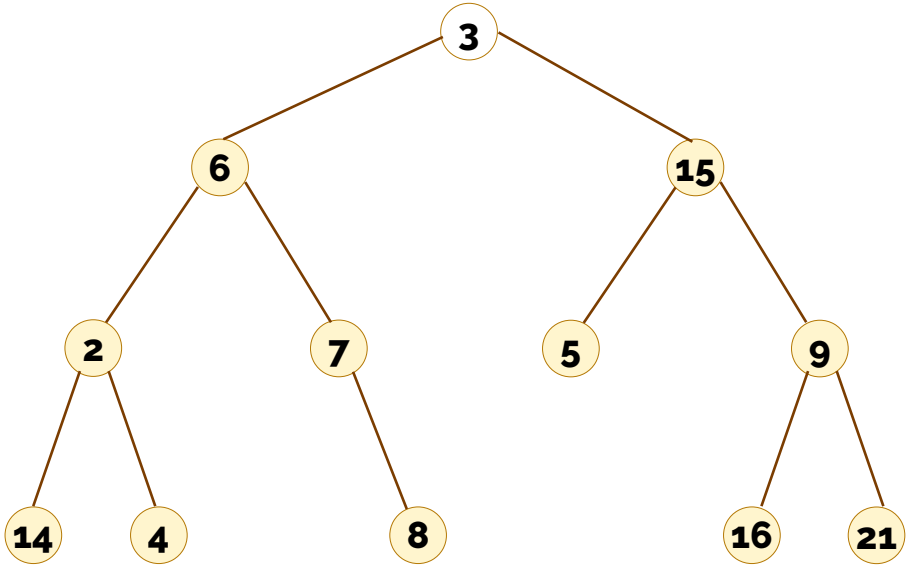


Algoritmo de recorrido en anchura



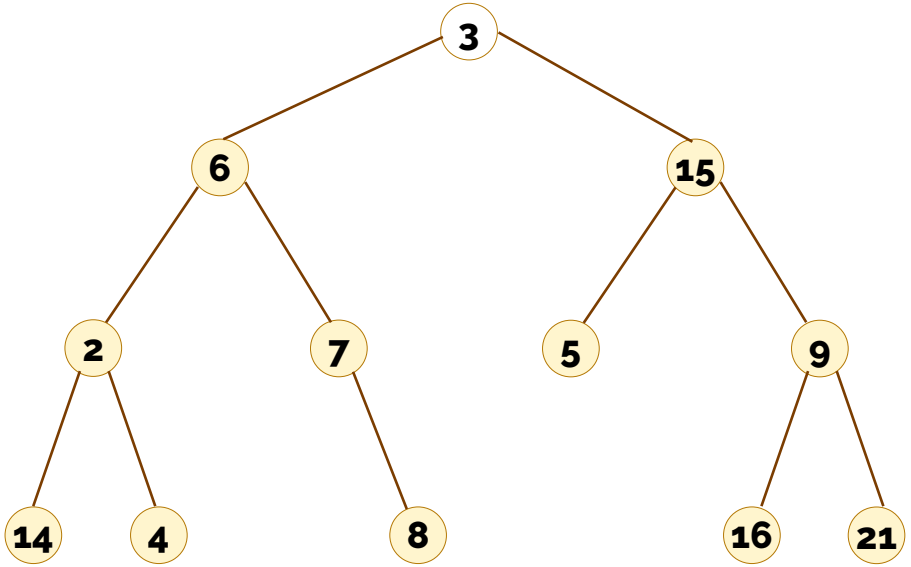
- Insertamos la raíz en la cola.
- Repetimos:
  - Sacamos nodo de la cola.
  - Visitamos ese nodo.

Algoritmo de recorrido en anchura

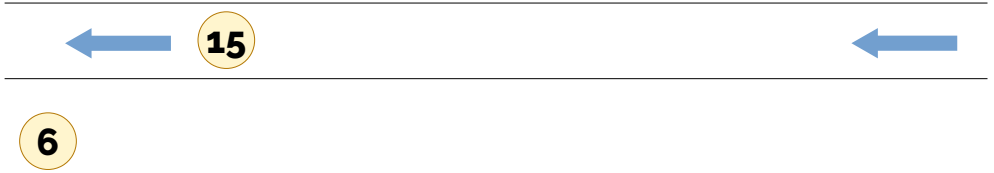


- Insertamos la raíz en la cola.
- Repetimos:
  - Sacamos nodo de la cola.
  - Visitamos ese nodo.
  - Insertamos sus hijos en la cola.

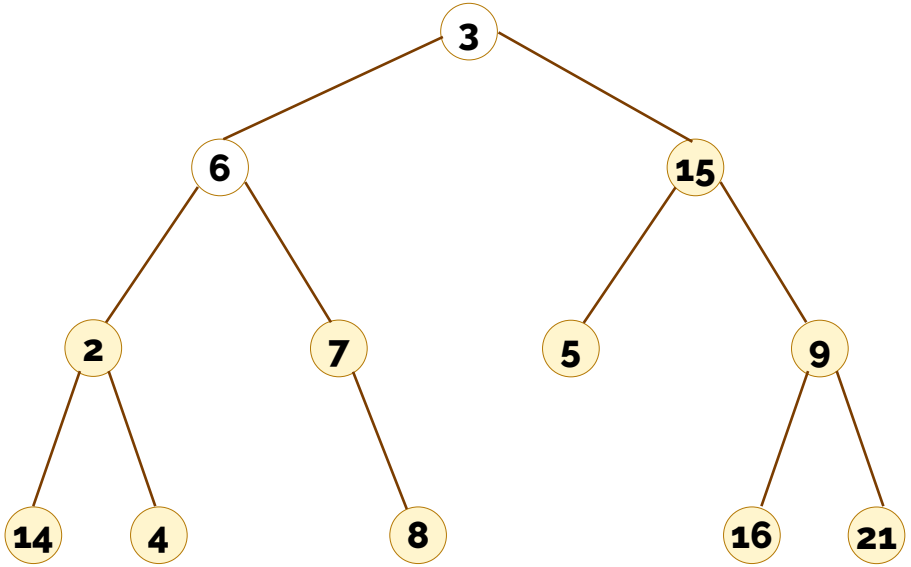
Algoritmo de recorrido en anchura



- Insertamos la raíz en la cola.
- Repetimos:
  - Sacamos nodo de la cola.
  - Visitamos ese nodo.
  - Insertamos sus hijos en la cola.

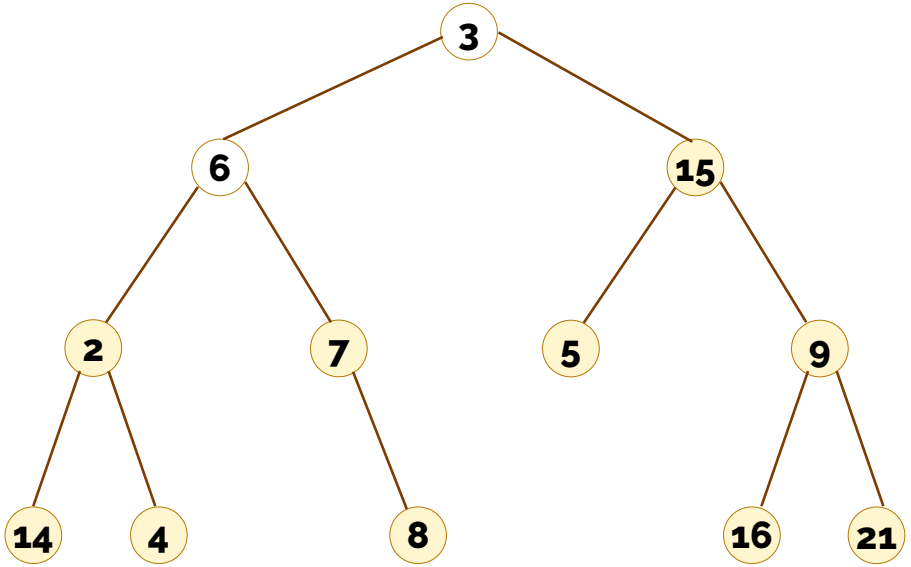


Algoritmo de recorrido en anchura



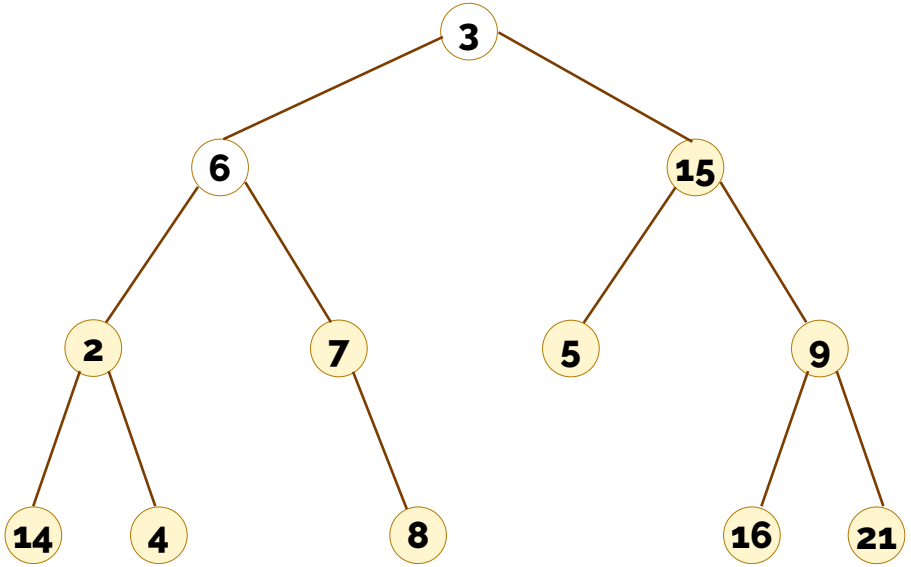
- Insertamos la raíz en la cola.
- Repetimos:
  - Sacamos nodo de la cola.
  - Visitamos ese nodo.
  - Insertamos sus hijos en la cola.

Algoritmo de recorrido en anchura

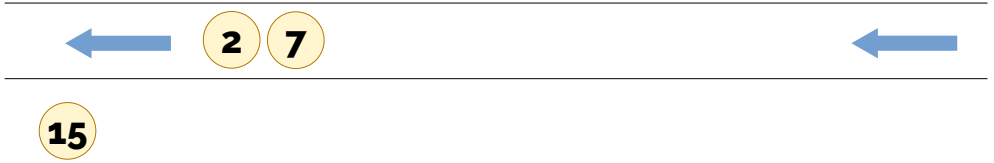


- Insertamos la raíz en la cola.
- Repetimos:
  - Sacamos nodo de la cola.
  - Visitamos ese nodo.
  - Insertamos sus hijos en la cola.

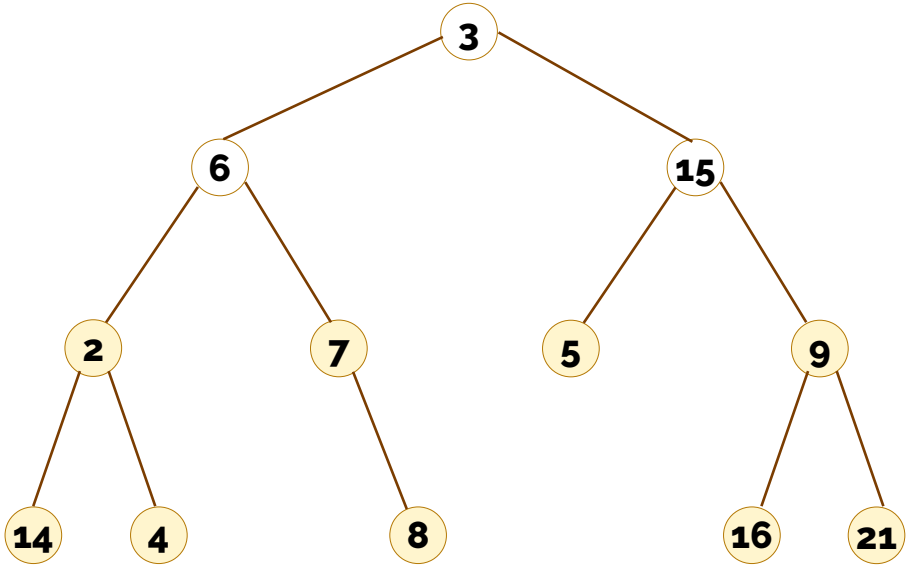
Algoritmo de recorrido en anchura



- Insertamos la raíz en la cola.
- Repetimos:
  - Sacamos nodo de la cola.
  - Visitamos ese nodo.
  - Insertamos sus hijos en la cola.

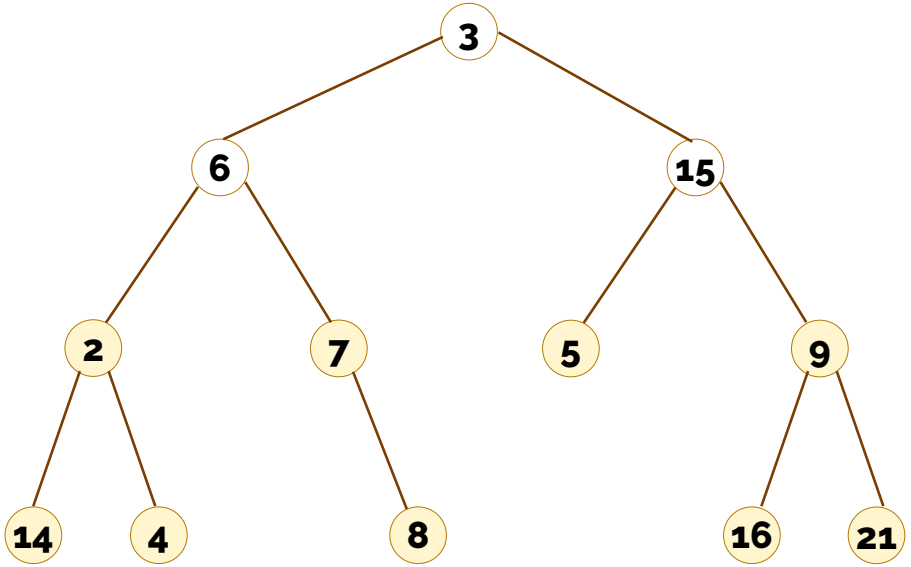


Algoritmo de recorrido en anchura



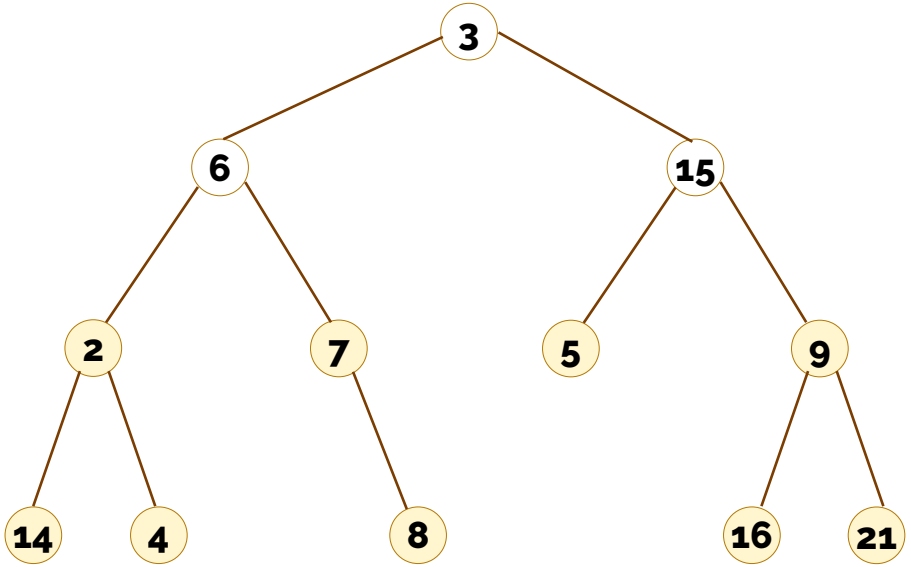
- Insertamos la raíz en la cola.
- Repetimos:
  - Sacamos nodo de la cola.
  - Visitamos ese nodo.
  - Insertamos sus hijos en la cola.

Algoritmo de recorrido en anchura

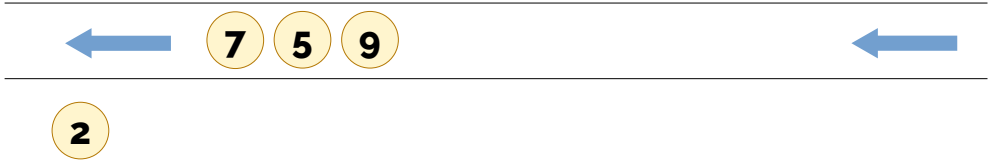


- Insertamos la raíz en la cola.
- Repetimos:
  - Sacamos nodo de la cola.
  - Visitamos ese nodo.
  - Insertamos sus hijos en la cola.

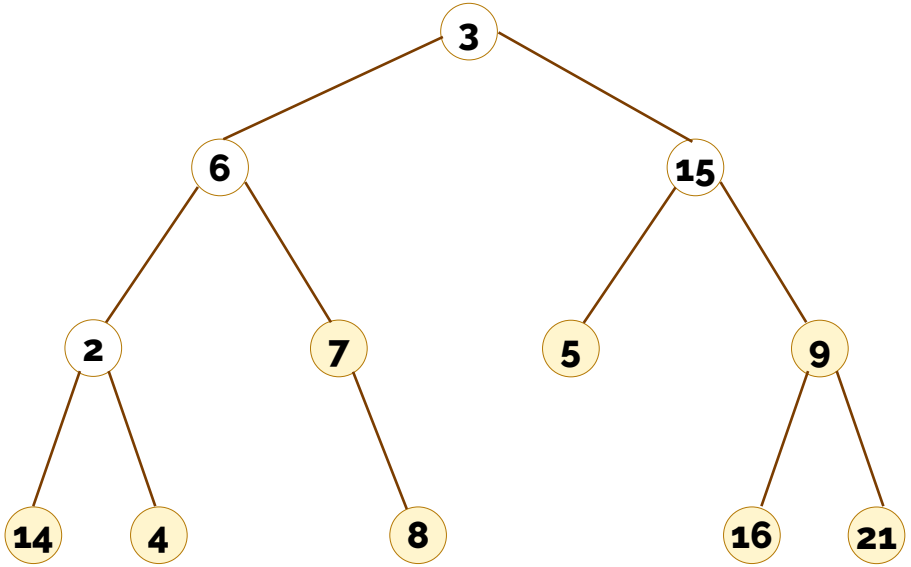
Algoritmo de recorrido en anchura



- Insertamos la raíz en la cola.
- Repetimos:
  - Sacamos nodo de la cola.
  - Visitamos ese nodo.
  - Insertamos sus hijos en la cola.

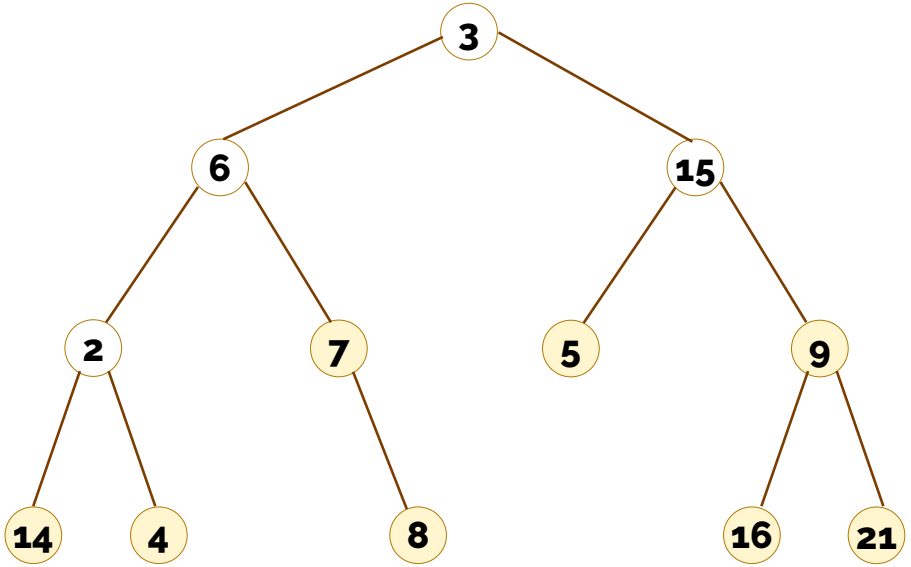


Algoritmo de recorrido en anchura

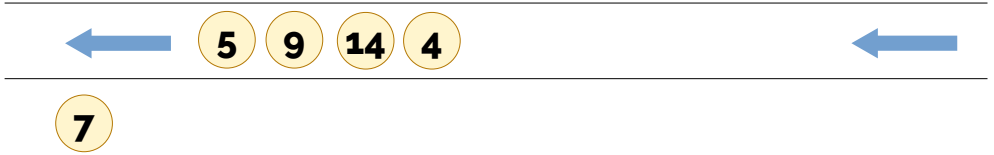


- Insertamos la raíz en la cola.
- Repetimos:
  - Sacamos nodo de la cola.
  - Visitamos ese nodo.
  - Insertamos sus hijos en la cola.

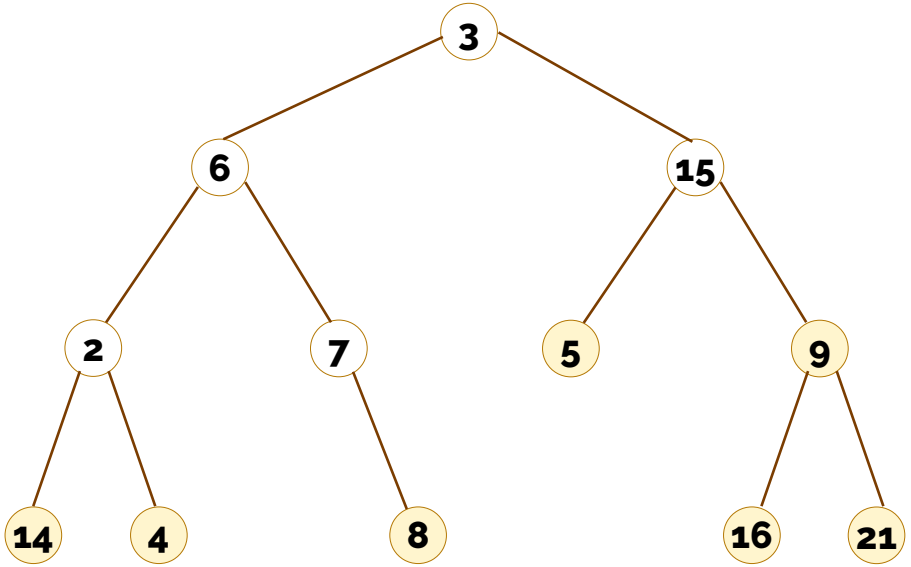
Algoritmo de recorrido en anchura



- Insertamos la raíz en la cola.
- Repetimos:
  - Sacamos nodo de la cola.
  - Visitamos ese nodo.
  - Insertamos sus hijos en la cola.

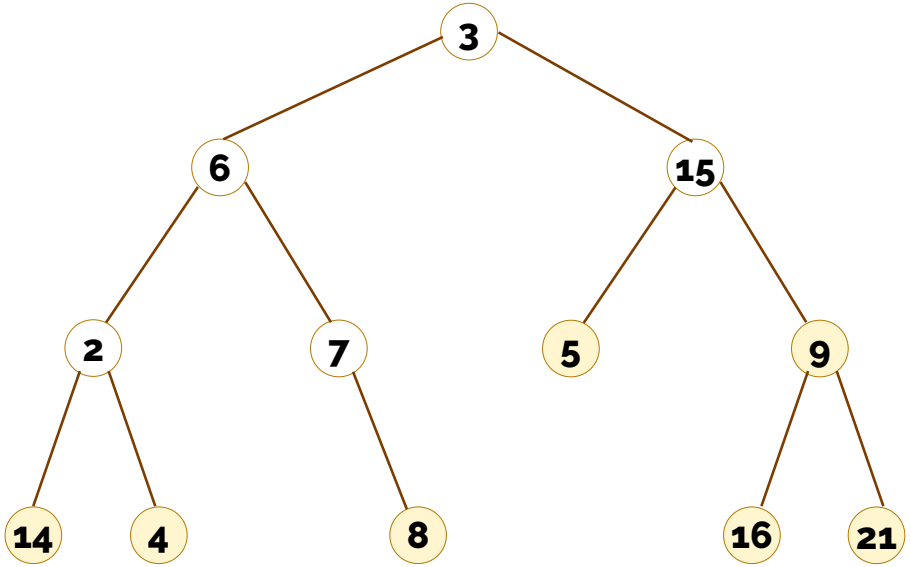


Algoritmo de recorrido en anchura

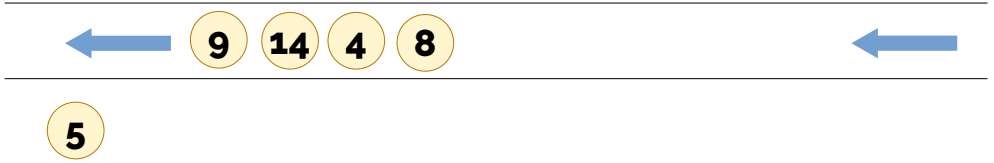


- Insertamos la raíz en la cola.
- Repetimos:
  - Sacamos nodo de la cola.
  - Visitamos ese nodo.
  - Insertamos sus hijos en la cola.

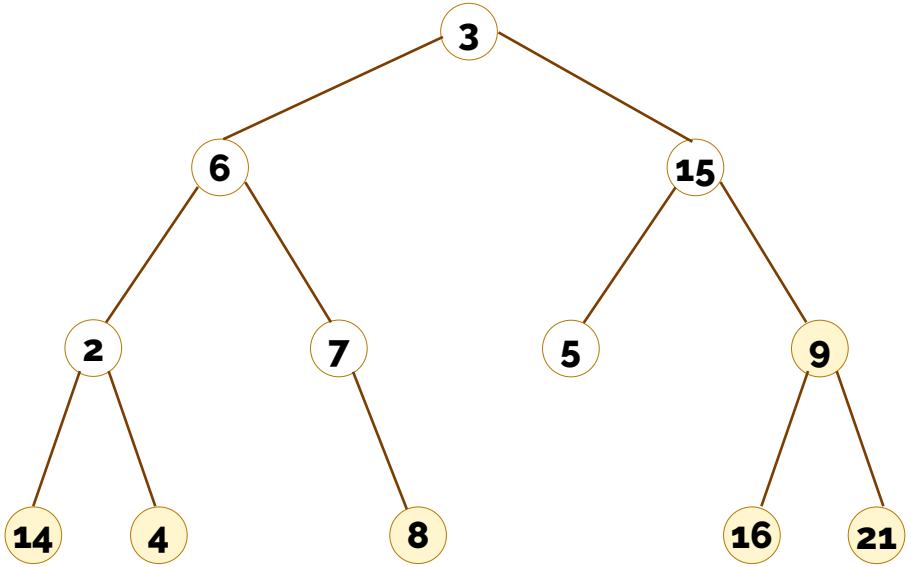
Algoritmo de recorrido en anchura



- Insertamos la raíz en la cola.
- Repetimos:
  - Sacamos nodo de la cola.
  - Visitamos ese nodo.
  - Insertamos sus hijos en la cola.

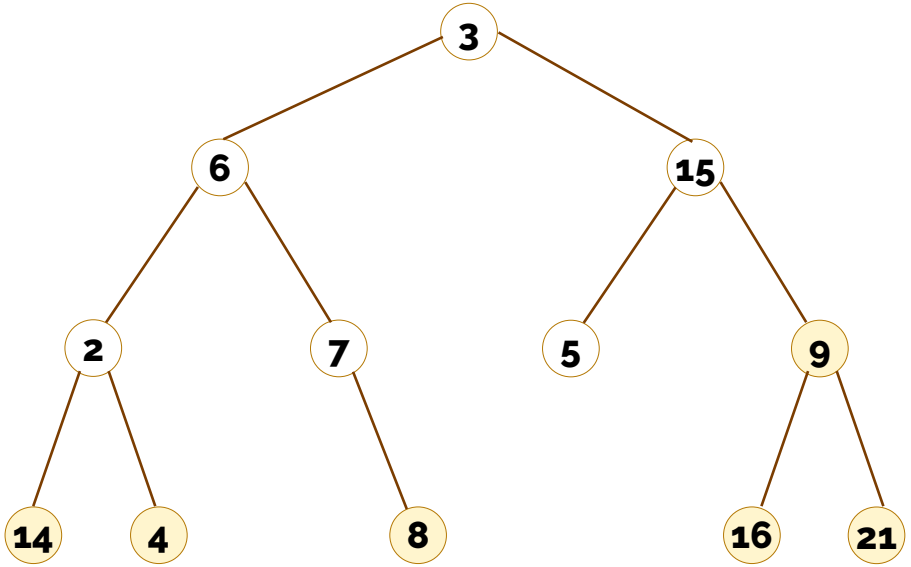


Algoritmo de recorrido en anchura

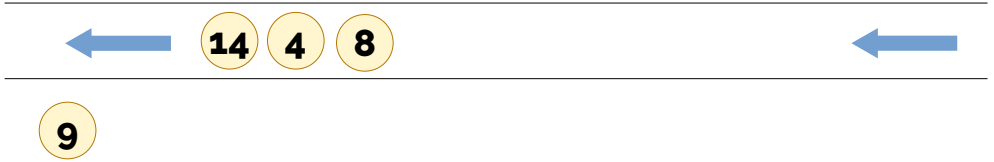


- Insertamos la raíz en la cola.
- Repetimos:
  - Sacamos nodo de la cola.
  - Visitamos ese nodo.
  - Insertamos sus hijos en la cola.

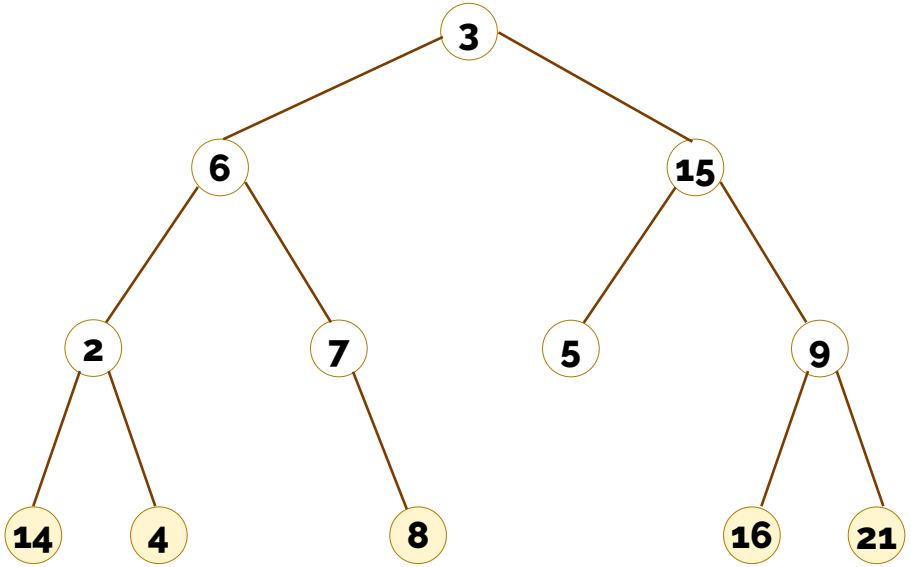
Algoritmo de recorrido en anchura



- Insertamos la raíz en la cola.
- Repetimos:
  - Sacamos nodo de la cola.
  - Visitamos ese nodo.
  - Insertamos sus hijos en la cola.

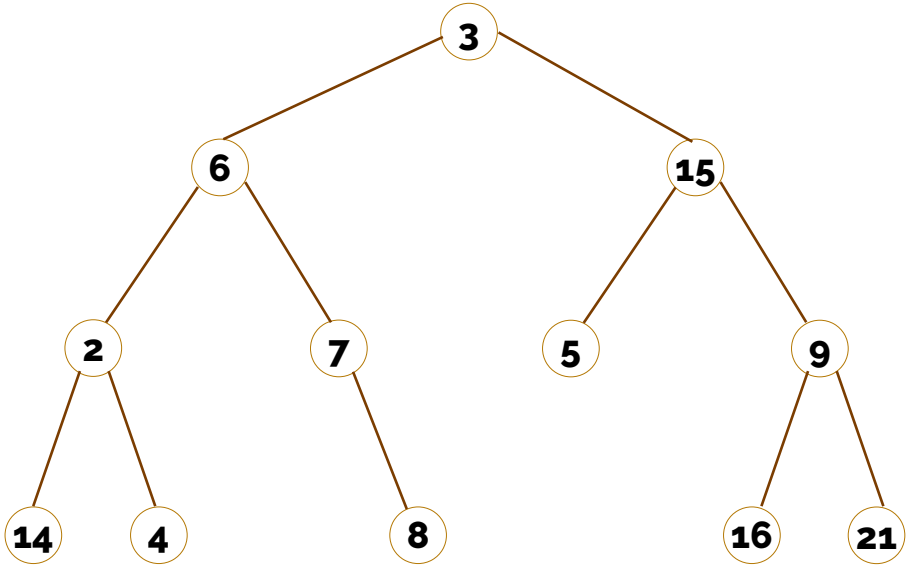


Algoritmo de recorrido en anchura



- Insertamos la raíz en la cola.
- Repetimos:
  - Sacamos nodo de la cola.
  - Visitamos ese nodo.
  - Insertamos sus hijos en la cola.

Algoritmo de recorrido en anchura



- Insertamos la raíz en la cola.
- Repetimos:
  - Sacamos nodo de la cola.
  - Visitamos ese nodo.
  - Insertamos sus hijos en la cola.

hasta que la cola esté vacía.

Código de `levelorder`

```
template<typename T>
void BinTree<T>::levelorder() const {
  std::queue<NodePointer> pending;
  pending.push(root_node);

  while (!pending.empty()) {
    NodePointer current = pending.front();
    pending.pop();
    std::cout << current→elem << " ";
    if (current→left ≠ nullptr) {
      pending.push(current→left);
    }
    if (current→right ≠ nullptr) {
      pending.push(current→right);
    }
  }
}
```

- Insertamos la raíz en la cola.
- Repetimos:
  - Sacamos nodo de la cola.
  - Visitamos ese nodo.
  - Insertamos sus hijos en la cola.

hasta que la cola esté vacía.

Código de `levelorder`

```
template<typename T>
void BinTree<T>::levelorder() const {
  std::queue<NodePointer> pending;
  if (root_node ≠ nullptr) {
    pending.push(root_node);
  }
  while (!pending.empty()) {
    NodePointer current = pending.front();
    pending.pop();
    std::cout << current→elem << " ";
    if (current→left ≠ nullptr) {
      pending.push(current→left);
    }
    if (current→right ≠ nullptr) {
      pending.push(current→right);
    }
  }
}
```

- Añadimos una guarda en el caso en el que el árbol esté vacío.

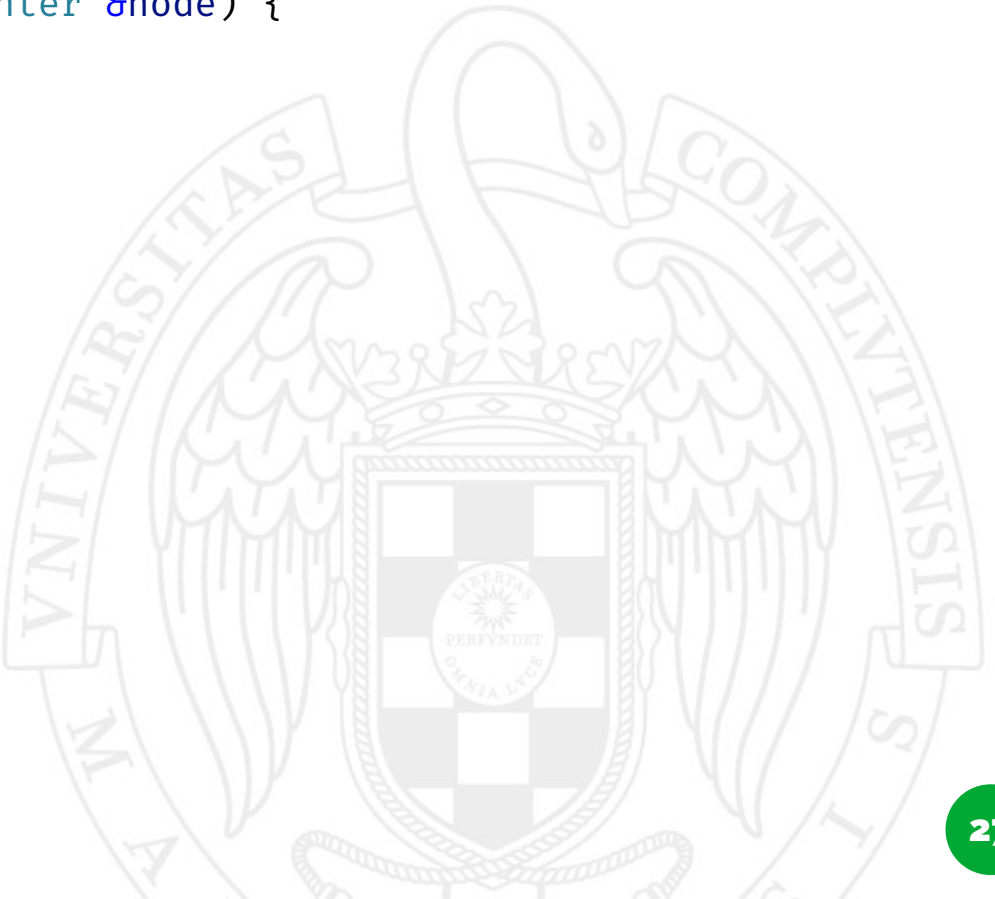
Ejemplo

```
int main() {
  BinTree<int> tree {{{ 9 }, 4, { 5 }}, 7, {{ }, 4, { 6 }}};


  std::cout << "Recorrido por niveles: " << std::endl;
  tree.levelorder();
  std::cout << std::endl;

  return 0;
}
```

```
Recorrido por niveles:
7 4 4 9 5 6
```

Método auxiliar postorder

```
template<typename T>
void BinTree<T>::postorder(const NodePointer &node) {
  if (node ≠ nullptr) {
    postorder(node→left);
    postorder(node→right);
    std::cout << node→elem << " ";
  }
}
```



Ejemplo

```
int main() {
  BinTree<int> tree = {{{ 9 }, 4, { 5 }}, 7, {{ 10 }, 4, { 6 }}};

  std::vector<int> vec;
  std::cout << "Recorrido en preorden: " << std::endl;
  tree.preorder();
  std::cout << std::endl;

  std::cout << "Recorrido en inorden: " << std::endl;
  tree.inorder();
  std::cout << std::endl;

  std::cout << "Recorrido en postorden: " << std::endl;
  tree.postorder();
  std::cout << std::endl;

  return 0;
}
```

```
Recorrido en preorden:
7 4 9 5 4 10 6
Recorrido en inorden:
9 4 5 7 10 4 6
Recorrido en postorden:
9 5 4 10 6 4 7
```

ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS ARBORESCENTES

Parametrizando el recorrido de un árbol

Manuel Montenegro Montes
Departamento de Sistemas Informáticos y Computación
Facultad de Informática – Universidad Complutense de Madrid

Recorrer un árbol

- Recorrer un árbol significa *visitar* todos sus nodos.
- **Visitar** un nodo significa realizar una acción que dependa del valor contenido en dicho nodo.

  Hasta ahora:

```
template<typename T>
void BinTree<T>::preorder(const NodePointer &node) {
  if (node ≠ nullptr) {
    std::cout << node→elem << " ";
    preorder(node→left);
    preorder(node→right);
  }
}
```

Parametrizar el recorrido

- Podemos parametrizar el recorrido con respecto a la acción a realizar en cada nodo.

```
template<typename T>
void BinTree<T>::preorder(const NodePointer &node) {
  if (node ≠ nullptr) {
    std::cout << node→elem << " ";
    preorder(node→left);
    preorder(node→right);
  }
}
```

Parametrizar el recorrido

- Podemos parametrizar el recorrido con respecto a la acción a realizar en cada nodo.

```
template<typename T>
template<typename U>
void BinTree<T>::preorder(const NodePointer &node, U func) {
  if (node ≠ nullptr) {
    func(node→elem);
    preorder(node→left, func);
    preorder(node→right, func);
  }
}
```

Parametrizar el recorrido

- Modificamos también el método `preorden( )` de la clase, que realiza la llamada inicial a la función recursiva:

```
template<class T>
class BinTree {
public:

   ...

  template <typename U>
  void preorder(U func) const {
    preorder(root_node, func);
  }

   ...

};
```

Ejemplos

- Supongamos que tenemos el siguiente árbol:

```
BinTree<int> tree {{{ 9 }, 4, { 5 }}, 7, {{ 10 }, 4, { 6 }}};
```

- Imprimir el recorrido en preorden:

```
tree.preorder([] (int x) { std::cout << x << " "; });
```

- Imprimir solamente los elementos pares:

```
tree.preorder([] (int x) {
  if (x % 2 == 0) {
    std::cout << x << " ";
  }
});
```

Ejemplos

- Supongamos que tenemos el siguiente árbol:

```
BinTree<int> tree {{{ 9 }, 4, { 5 }}, 7, {{ 10 }, 4, { 6 }}};
```

- Sumar los elementos del árbol:

```
int acum = 0;
tree.preorder([&acum](int x) { acum += x; });
std::cout << acum << std::endl;
```

- Contar el número de elementos de un árbol:

```
int num_elems = 0;
tree.preorder([&num_elems](int x) { num_elems++; });
std::cout << num_elems << std::endl;
```

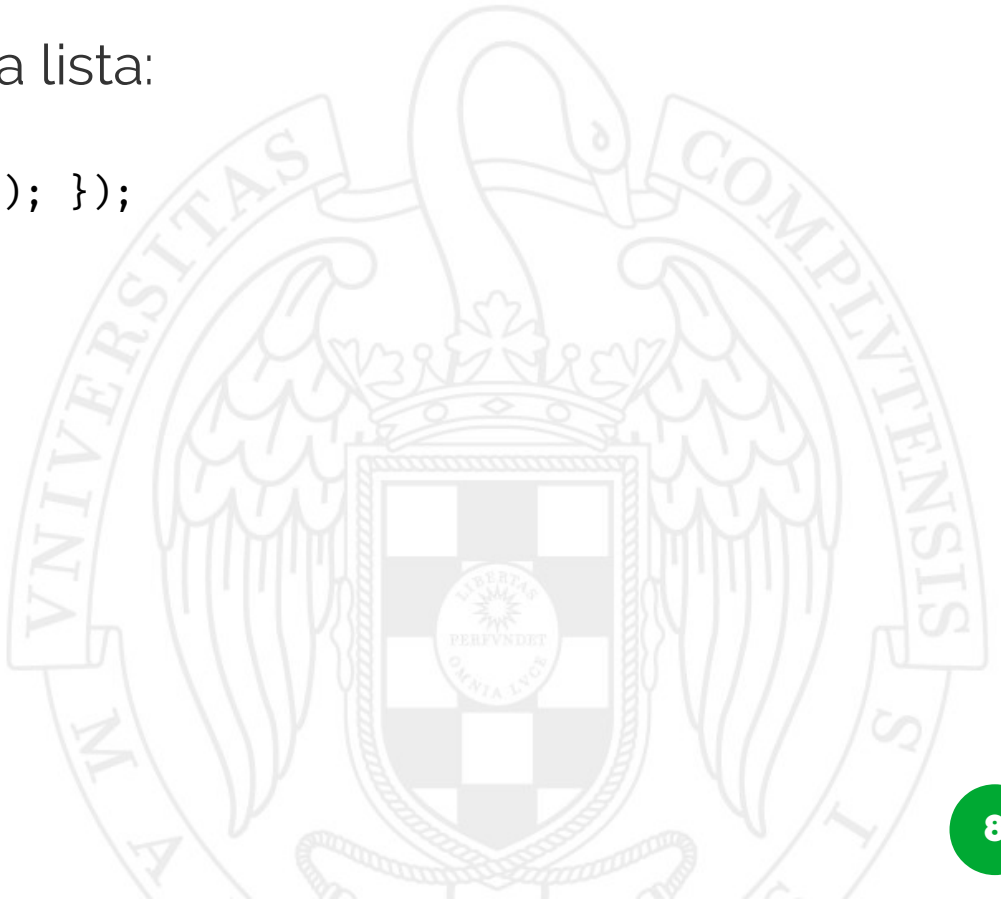
Ejemplos

- Supongamos que tenemos el siguiente árbol:

```
BinTree<int> tree {{{ 9 }, 4, { 5 }}, 7, {{ 10 }, 4, { 6 }}};
```

- Añadir los elementos del árbol a una lista:

```
std::vector<int> v;
tree.preorder([&v](int x) { v.push_back(x); });
```



ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS ARBORESCENTES

El TAD Conjunto

Manuel Montenegro Montes
Departamento de Sistemas Informáticos y Computación
Facultad de Informática – Universidad Complutense de Madrid

Conjuntos

- Un **conjunto** es una colección de elementos del mismo tipo.
- ¿Cuál es la diferencia entre un conjunto y una lista?

Lista

- El orden de los elementos es relevante:

$$[1, 4, 5] \neq [4, 5, 1]$$

- Pueden contener elementos duplicados:

$$[1, 4, 4, 5] \neq [1, 4, 5]$$

Conjunto

- No existe el concepto de orden entre elementos:

$$\{1, 4, 5\} = \{4, 5, 1\}$$

- La existencia de duplicados es irrelevante:

$$\{1, 4, 5\} \cup \{4\} = \{1, 4, 5\}$$

Modelo de conjuntos

- Varias formas de implementar un conjunto.
- Cuando el conjunto está implementado y tan solo tenemos que utilizarlo, pensamos en términos del **modelo**.
- Cada instancia del TAD Conjunto representa un **conjunto finito**.

$$\{x_1, x_2, x_3, \ldots, x_n\}$$

Operaciones en el TAD conjunto

- Constructoras:
  - Crear un conjunto vacío: ***create_empty***
- Mutadoras:
  - Añadir un elemento al conjunto: ***insert***
  - Eliminar un elemento del conjunto: ***erase***
- Observadoras:
  - Averiguar si un elemento está en el conjunto: ***contains***
  - Saber si el conjunto está vacío: ***empty***
  - Saber el tamaño del conjunto: ***size***

Operaciones constructoras y mutadoras

{ *true* }

create_empty() → (*S: Set*)

{ *S = ∅* }

{ *true* }

insert(*x: Elem, S: Set*)

{ *S = old(S) ∪ {x}* }

{ *true* }

erase(*x: Elem, S: Set*)

{ *S = old(S) - {x}* }

Operaciones observadoras

$\{ \; true \; \}$

contains$(x: Elem, S: Set) \rightarrow (b: bool)$

$\{ \; b \Leftrightarrow x \in S \; \}$

$\{ \; true \; \}$

empty$(S: Set) \rightarrow (b: bool)$

$\{ \; b \Leftrightarrow S = \emptyset \; \}$

$\{ \; true \; \}$

size$(S: Set) \rightarrow (n: int)$

$\{ \; n = |S| \; \}$

Interfaz en C++

```
class set {
public:
  set();
  set(const set &other);
  ~set();

  void insert(const T &elem);
  void erase(const T &elem);

  bool contains(const T &elem) const;
  int size() const;
  bool empty() const;

private:
  // ???
};
```

Dos implementaciones

- Mediante **listas**.
- Mediante **árboles binarios de búsqueda**.

ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS ARBORESCENTES

Implementación del TAD Conjunto mediante listas ordenadas

Manuel Montenegro Montes
Departamento de Sistemas Informáticos y Computación
Facultad de Informática – Universidad Complutense de Madrid

Operaciones en el TAD Conjunto

- Constructoras:
  - Crear un conjunto vacío: ***create_empty***
- Mutadoras:
  - Añadir un elemento al conjunto: ***insert***
  - Eliminar un elemento del conjunto: ***erase***
- Observadoras:
  - Averiguar si un elemento está en el conjunto: ***contains***
  - Saber si el conjunto está vacío: ***empty***
  - Saber el tamaño del conjunto: ***size***

Representación mediante listas ordenadas

- Clase que contiene un único atributo: `list_elems`, de tipo Lista.
- El atributo `list_elems` contiene los elementos del conjunto que se quiere representar de modo que:
  - `list_elems` almacena los elementos en orden ascendente.
  - `list_elems` no almacena duplicados.

$\{4, 5, 7, 3, 1\}$

`list_elems:` $[1, 3, 4, 5, 7]$

Representación mediante listas ordenadas

- Clase que contiene un único atributo: `list_elems`, de tipo Lista.
- El atributo `list_elems` contiene los elementos del conjunto que se quiere representar de modo que:
  - `list_elems` almacena los elementos en orden ascendente.
  - `list_elems` no almacena duplicados.

```
template <typename T>
class SetList {
public:
  ...
private:
  using List = ???
  List list_elems;
};
```

```
std::vector<T>
std::list<T>
```

Representación mediante listas ordenadas

- **Función de abstracción:**

  Si s es una instancia de la clase `SetList`:

  $f($`s`$)$ = `{ s.list_elems[i] | 0 ⩽ i < s.list_elems.size() }`

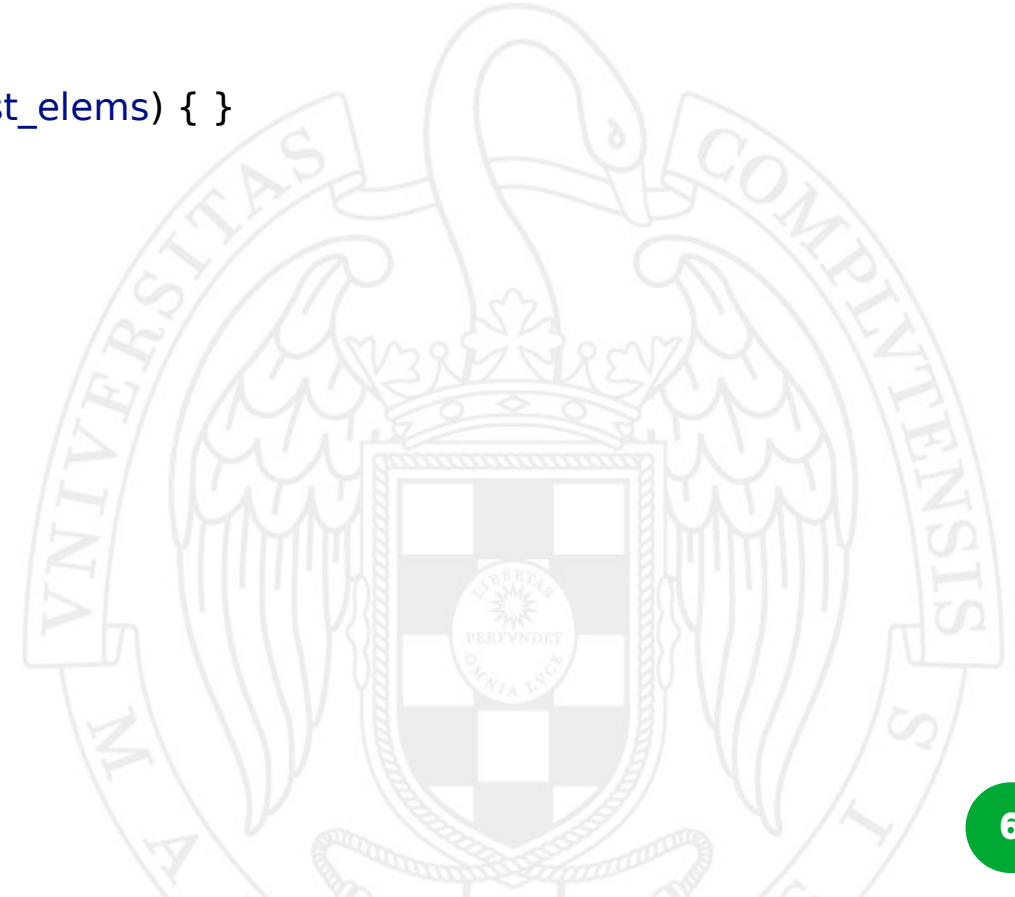
- **Invariante de representación**

  $I($`s`$) \equiv$ `∀i, j: 0 ⩽ i < j < s.list_elems.size()`

  $\Longrightarrow$ `s.list_elems[i] < s.list_elems[j]`

Operaciones constructoras

```
template <typename T>
class SetList {
public:
  SetList() { }
  SetList(const SetList &other): list_elems(other.list_elems) { }
  ~SetList() { }

private:
  ...
  List list_elems;
};
```



Operaciones observadoras

```
template <typename T>
class SetList {
public:
  ...
  bool contains(const T &elem) const { ... }

  int size() const {
    return list_elems.size();
  }

  bool empty() const {;
    return list_elems.empty();
  }

private:
  ...
  List list_elems;
};
```

Operación *contains*

- Utilizamos una función de búsqueda binaria

  `bool binary_search(iterator first, iterator last, const T& val)`

  definida en `<algorithm>`

```
template <typename T>
class SetList {
public:

  bool contains(const T &elem) const {
    return std::binary_search(list_elems.begin(), list_elems.end(), elem);
  }

  ...
};
```

Operaciones mutadoras

```
template <typename T>
class SetList {
public:
  ...
  void insert(const T &elem) { ... }
  void erase(const T &elem) { ... }

private:
  ...
  List list_elems;
};
```

Operación *insert*

- Necesitamos insertar el elemento en `list_elems` de modo que la lista permanezca ordenada.
- Podemos utilizar búsqueda binaria para saber dónde insertar el elemento.
- Problema: `binary_search` solamente indica si un elemento está en la lista o no.
- Pero tenemos la función `lower_bound`:

```
iterator lower_bound(iterator begin, iterator end, const T &elem)
```

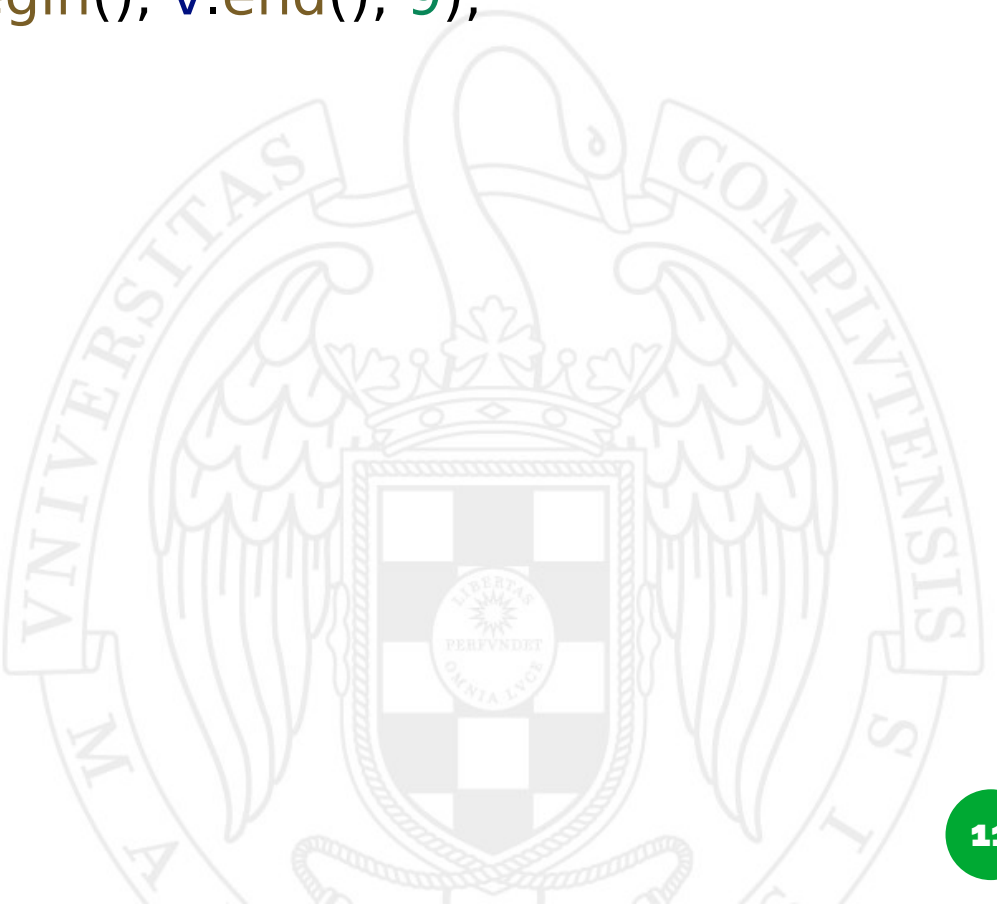
Devuelve un iterador al primer elemento contenido entre `begin` y `end` que no es estrictamente menor que `elem`.

Si todos son menores que `elem`, devuelve `end`.

Los elementos que hay entre `begin` y `end` han de estar ordenados.

Ejemplo: `lower_bound`

```
std::vector<int> v = {1, 5, 8, 10, 20};
auto it_pos = std::lower_bound(v.begin(), v.end(), 9);
std::cout << *it_pos << std::endl;
```

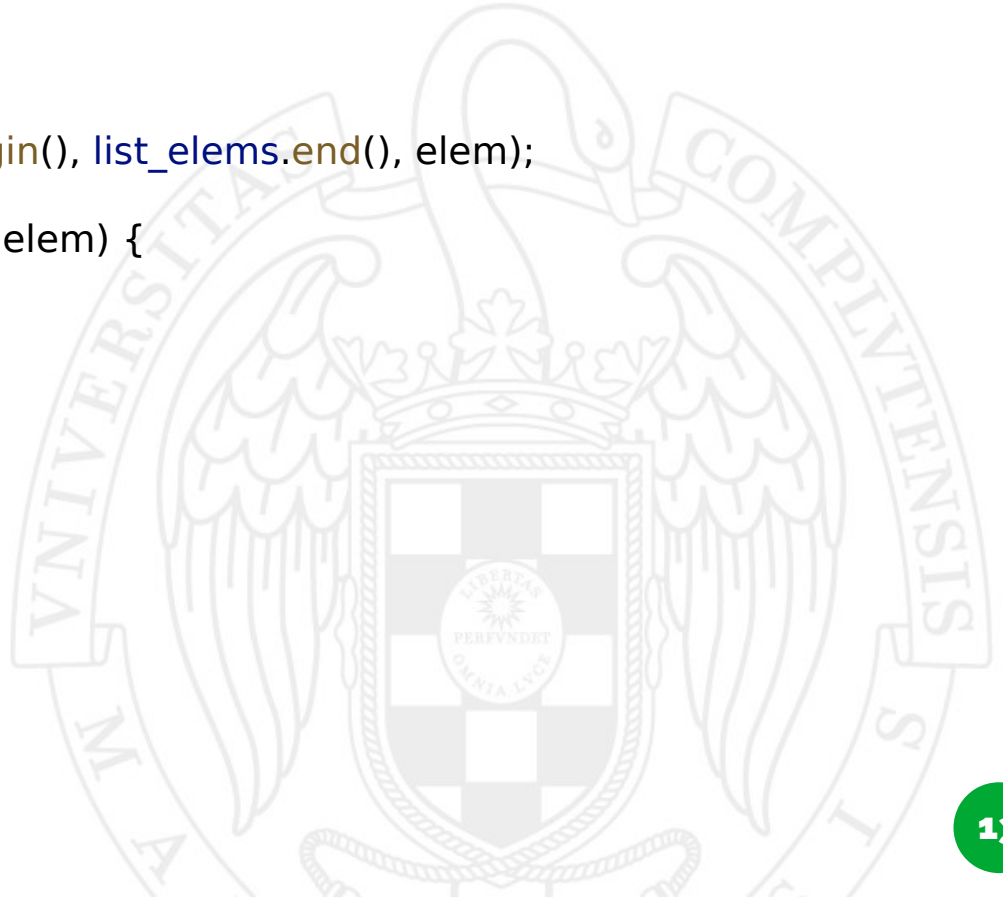


Operación *insert*

```
template <typename T>
class SetList {
public:
  ...
  void insert(const T &elem) {
    auto position = std::lower_bound(list_elems.begin(), list_elems.end(), elem);

    if (position == list_elems.end() || *position != elem) {
      list_elems.insert(position, elem);
    }
  }


private:
  ...
  List list_elems;
};
```

Operación *erase*

```
template <typename T>
class SetList {
public:
  ...
  void erase(const T &elem) {
    auto position = std::lower_bound(list_elems.begin(), list_elems.end(), elem);

    if (position != list_elems.end() && *position == elem) {
      list_elems.erase(position);
    }
  }

private:
  ...
  List list_elems;
};
```



¿Qué utilizo?

```
template <typename T>
class SetList {
public:
  ...
private:
  using List = ???
  List list_elems;
};
```

```
std::vector<T>
std::list<T>
```

Coste de las operaciones auxiliares

Operación	std::vector	std::list
`binary_search`	*O(log n)*	*O(n) (no es búsq. binaria)*
`lower_bound`	*O(log n)*	*O(n) (no es búsq. binaria)*
`insert` *(en listas)*	*O(n)*	*O(1)*
`erase` *(en listas)*	*O(n)*	*O(1)*

n = longitud de list_elems

Coste de las operaciones

Operación	std::vector	std::list
constructor	$O(1)$	$O(1)$
`empty`	$O(1)$	$O(1)$
`size`	$O(1)$	$O(1)$
`contains`	$O(\log n)$	$O(n)$
`insert`	$O(\log n) + O(n)$	$O(n) + O(1)$
`erase`	$O(\log n) + O(n)$	$O(n) + O(1)$

n = número de elementos del conjunto

Coste de las operaciones

Operación	std::vector	std::list
constructor	$O(1)$	$O(1)$
`empty`	$O(1)$	$O(1)$
`size`	$O(1)$	$O(1)$
`contains`	$O(\log n)$	$O(n)$
`insert`	$O(n)$	$O(n)$
`erase`	$O(n)$	$O(n)$

n = número de elementos del conjunto

¿Qué utilizo?

```
template <typename T>
class SetList {
public:
  ...
private:
  using List = std::vector<T>;
  List list_elems;
};
```

ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS ARBORESCENTES

Árboles binarios de búsqueda

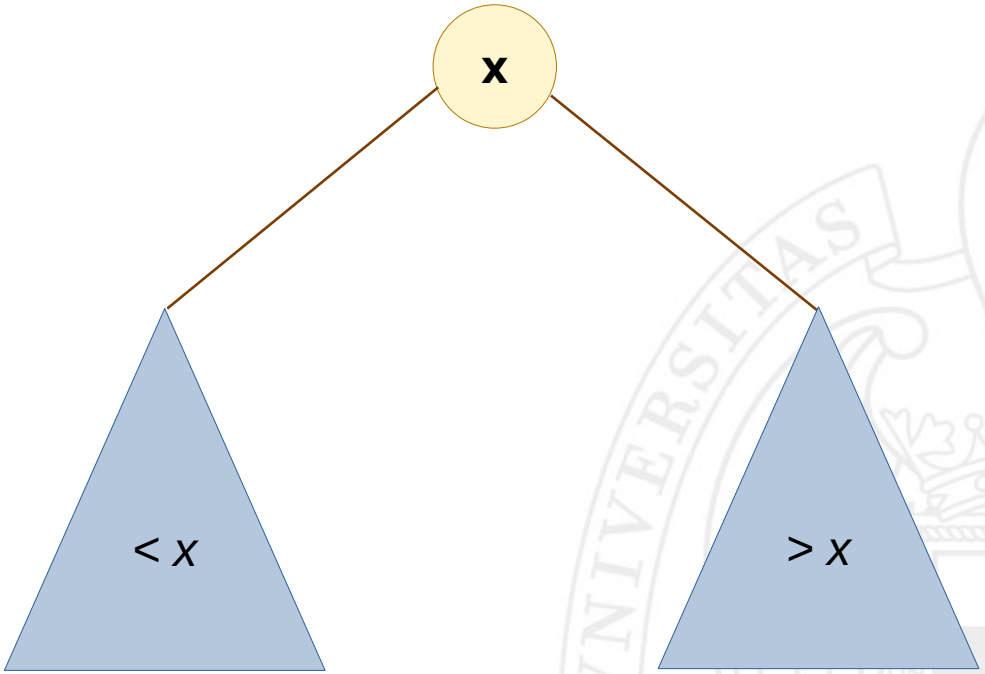
Manuel Montenegro Montes
Departamento de Sistemas Informáticos y Computación
Facultad de Informática – Universidad Complutense de Madrid

Árboles binarios de búsqueda (ABBs)

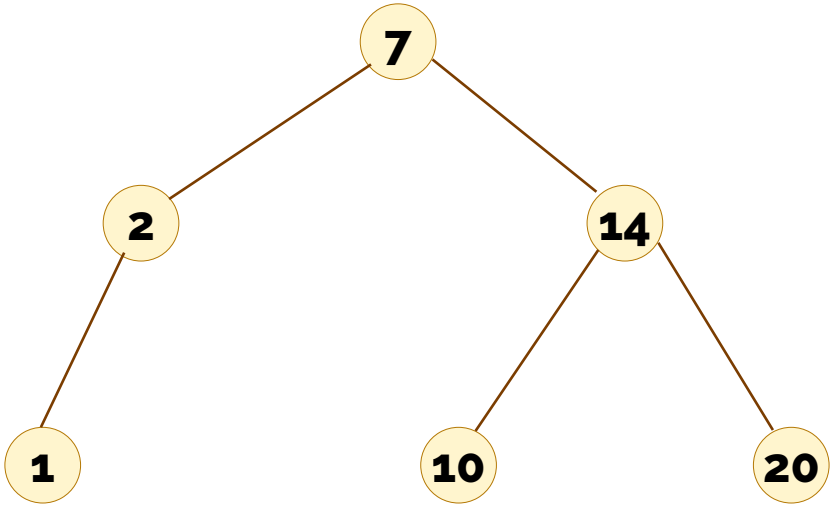
Un árbol binario es de búsqueda si:

- Es un árbol vacío, o bien,
- Es una hoja, o bien,
- Su raíz es un nodo interno, y además:
  - Todos los elementos de su **hijo izquierdo** son estrictamente **menores** que la raíz.
  - Todos los elementos de su **hijo derecho** son estrictamente **mayores** que la raíz.
  - Los hijos izquierdo y derecho son árboles de búsqueda.

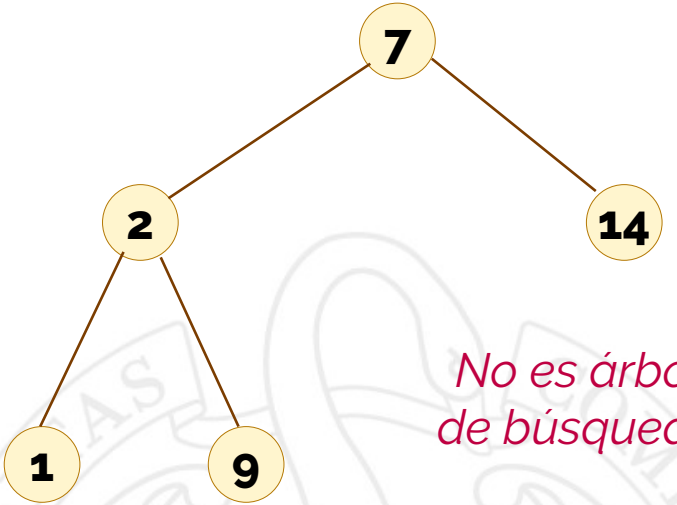
Árboles binarios de búsqueda



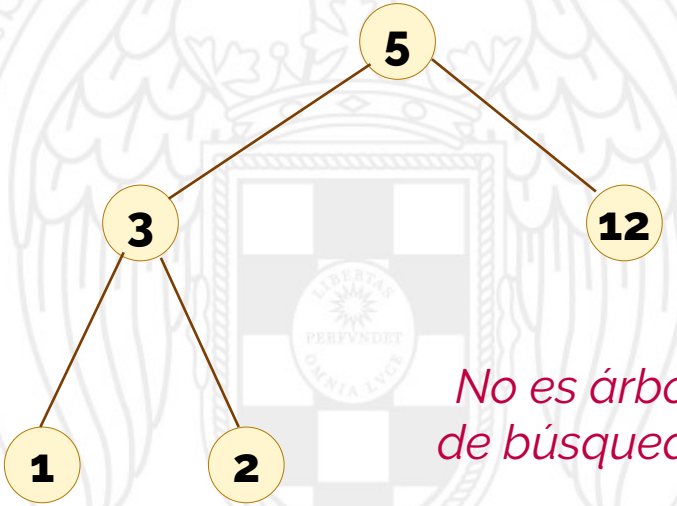
Ejemplos



Árbol de búsqueda



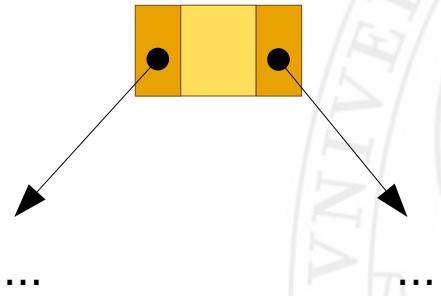
No es árbol de búsqueda



No es árbol de búsqueda

Representación mediante nodos

```
template <typename T>
struct Node {
  T elem;
  Node *left, *right;

  Node(Node *left, const T &elem, Node *right): left(left), elem(elem), right(right) { }
};
```

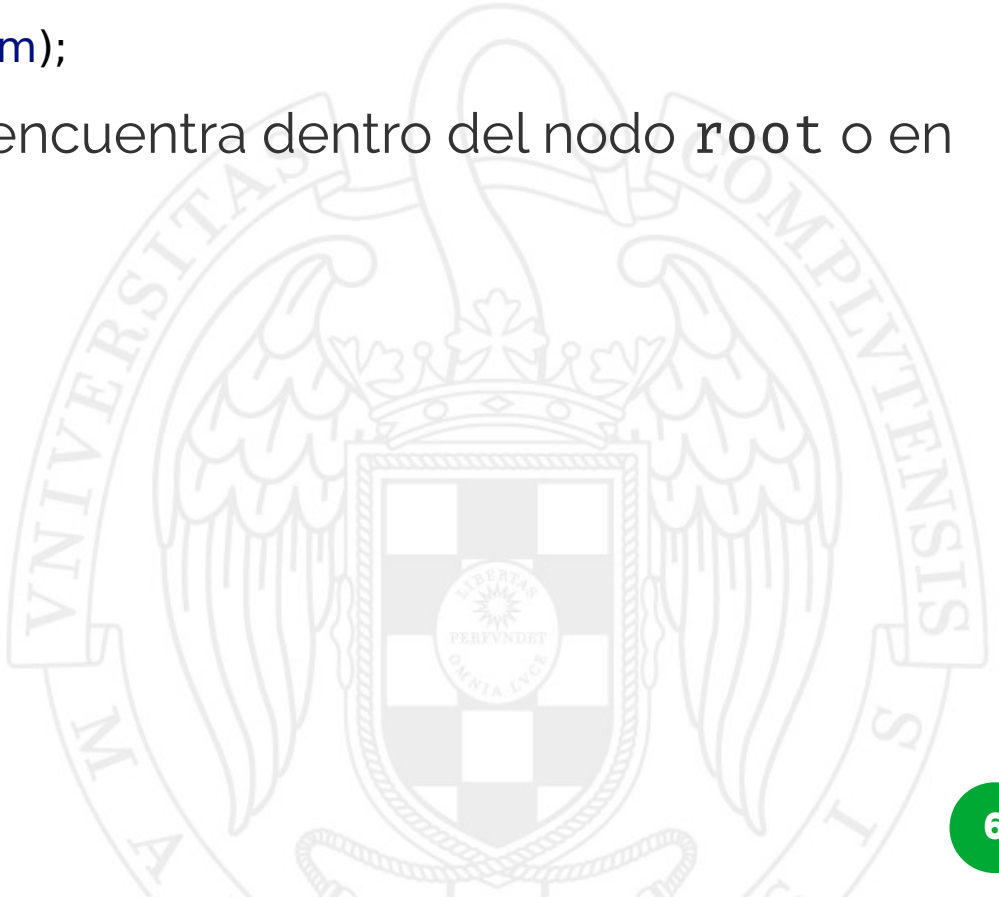


Búsqueda en un ABB

- Queremos implementar una función que determine si un elemento se encuentra en un árbol de búsqueda

```
bool search(const Node *root, const T &elem);
```

- La función determina si el `elem` se encuentra dentro del nodo `root` o en alguno de sus descendientes.
- Distinguimos cuatro casos.



Caso 1: Árbol vacío

```
bool search(const Node *root, const T &elem);
```

- Si `root == nullptr`, el árbol es vacío.
- En ese caso, `elem` no pertenece al árbol.
- Devolvemos `false`.

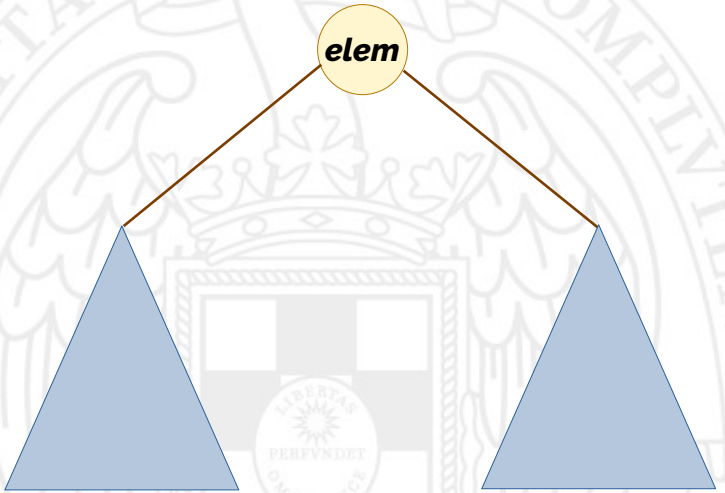
```
bool search(const Node *root, const T &elem) {
  if (root == nullptr) {
    return false;
  } else { ... }
}
```

Caso 2: `elem` == raíz del árbol

```
bool search(const Node *root, const T &elem);
```

- En este caso, hemos encontrado `elem` en el árbol. Devolvemos `true`.

```
bool search(const Node *root, const T &elem) {
  if (root == nullptr) {
    return false;
  } else if (elem == root->elem) {
    return true;
  } else { ... }
}
```

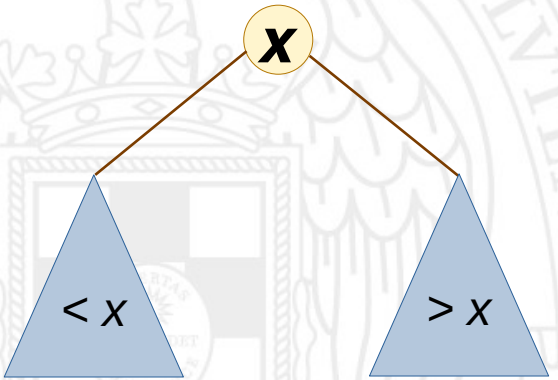


Caso 3: `elem` < raíz del árbol

```
bool search(const Node *root, const T &elem);
```

- Si el elemento a buscar es estrictamente menor que la raíz del árbol, lo buscamos recursivamente en el hijo izquierdo.

```
bool search(const Node *root, const T &elem) {
  if (root == nullptr) {
    return false;
  } else if (elem == root->elem) {
    return true;
  } else if (elem < root->elem) {
    return search(root->left, elem);
  } else { ... }
}
```

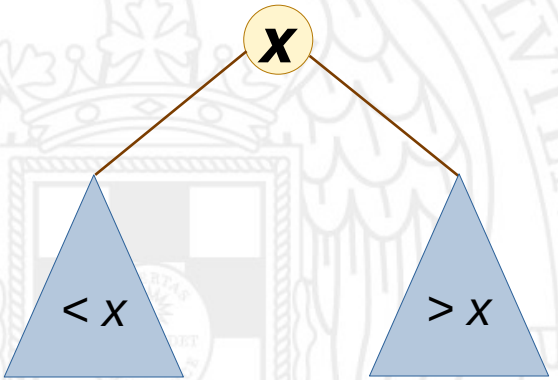


Caso 4: `elem` > raíz del árbol

```
bool search(const Node *root, const T &elem);
```

- Si el elemento a buscar es estrictamente mayor que la raíz del árbol, lo buscamos recursivamente en el hijo derecho.

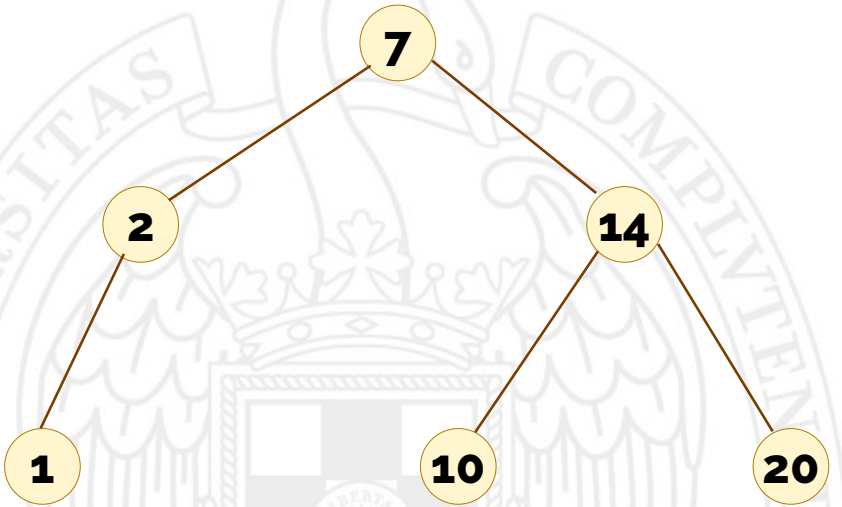
```
bool search(const Node *root, const T &elem) {
  if (root == nullptr) {
    return false;
  } else if (elem == root->elem) {
    return true;
  } else if (elem < root->elem) {
    return search(root->left, elem);
  } else {
    return search(root->right, elem);
  }
}
```



Ejemplo

- Buscamos el 10

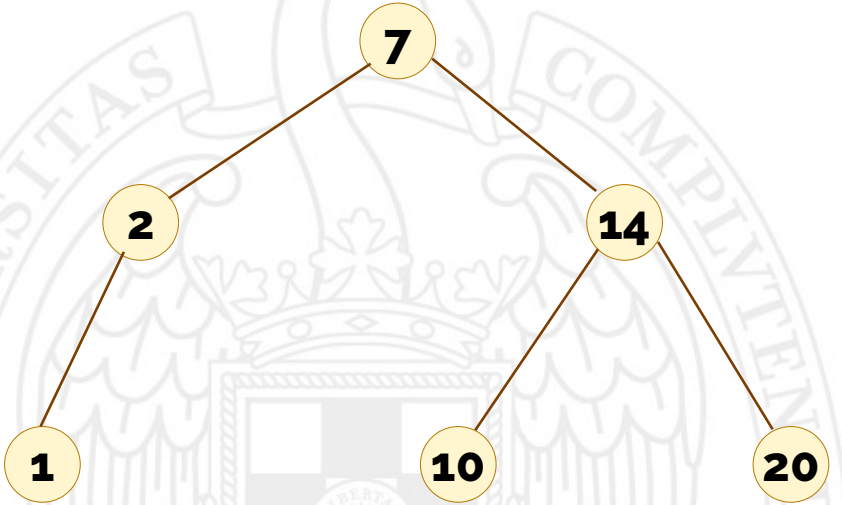
```
bool search(const Node *root, const T &elem) {
  if (root == nullptr) {
    return false;
  } else if (elem == root->elem) {
    return true;
  } else if (elem < root->elem) {
    return search(root->left, elem);
  } else {
    return search(root->right, elem);
  }
}
```



Ejemplo

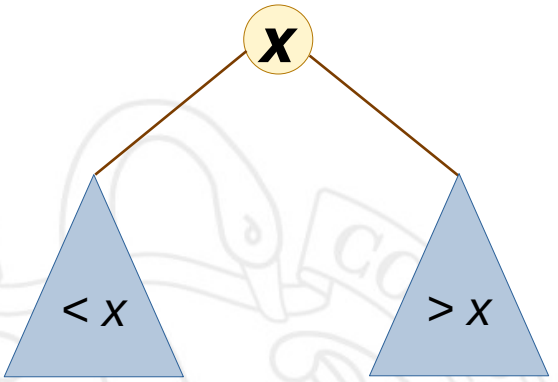
- Buscamos el 3

```
bool search(const Node *root, const T &elem) {
  if (root == nullptr) {
    return false;
  } else if (elem == root->elem) {
    return true;
  } else if (elem < root->elem) {
    return search(root->left, elem);
  } else {
    return search(root->right, elem);
  }
}
```



Coste de la función search

```
bool search(const Node *root, const T &elem) {
  if (root == nullptr) {
    return false;
  } else if (elem == root->elem) {
    return true;
  } else if (elem < root->elem) {
    return search(root->left, elem);
  } else {
    return search(root->right, elem);
  }
}
```



Coste de la función `search`

- En el caso peor, la función `search` desciende desde la raíz hasta las hojas.
- El coste en tiempo de la función `search` es lineal con respecto a la altura del árbol.

¿Y con respecto al número de nodos?

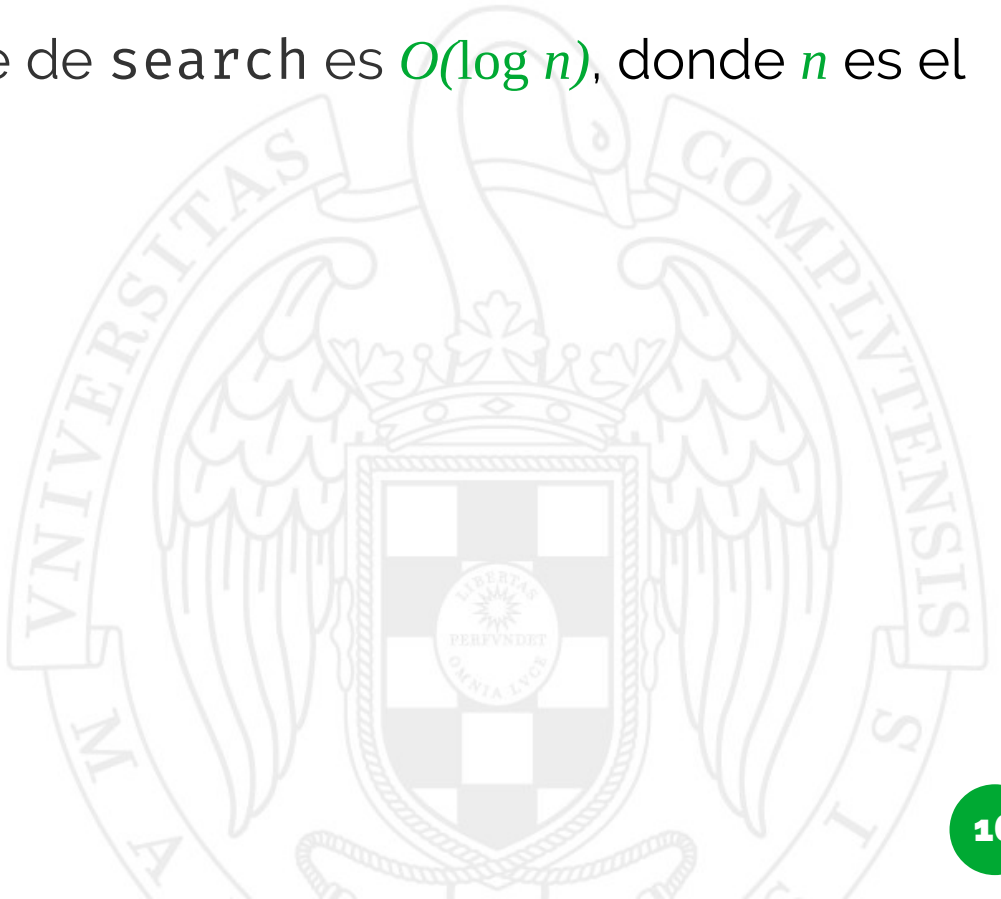
Recordatorio

Sea h la altura de un árbol y n su número de nodos.

- Si el árbol es **degenerado**, $h \in O(n)$
- Si el árbol es **equilibrado**, $h \in O(\log n)$
- Si no sabemos nada acerca de si el árbol está equilibrado o no, el caso peor es el árbol degenerado.

Coste de la función `search`

- Si el árbol es **degenerado**, el coste de `search` es $O(n)$, donde n es el número de nodos del árbol.
- Si el árbol está **equilibrado**, el coste de `search` es $O(\log n)$, donde n es el número de nodos del árbol.



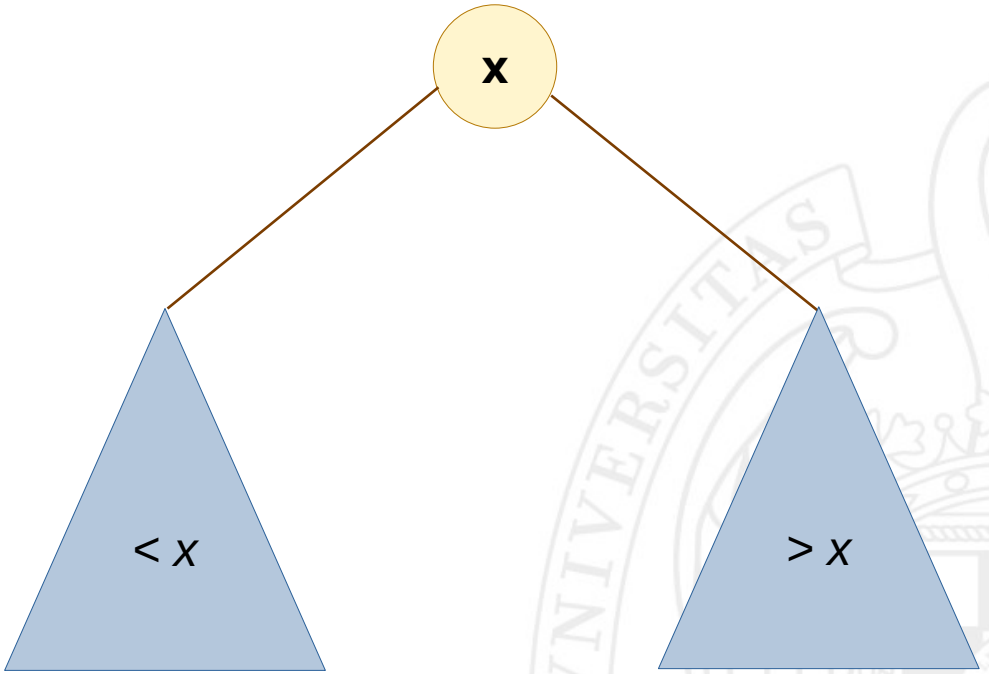
ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS ARBORESCENTES

Inserción en ABBs

Manuel Montenegro Montes
Departamento de Sistemas Informáticos y Computación
Facultad de Informática – Universidad Complutense de Madrid

Recordatorio: árboles binarios de búsqueda



Objetivo

- Implementar una función `insert(root, elem)`, que añada un nodo con el valor `elem` al ABB cuya raíz es `root`.

```
void insert(Node *root, const T &elem);
```

- El árbol resultante también ha de ser un ABB.
- Si `elem` ya se encuentra en el ABB, no hace nada.

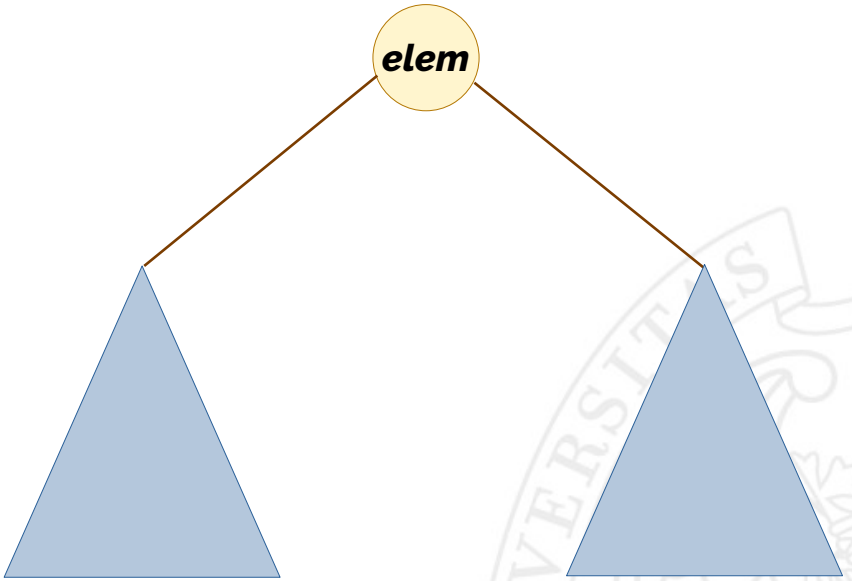
Caso 1: Árbol vacío (`root == nullptr`)

Antes de la inserción

—

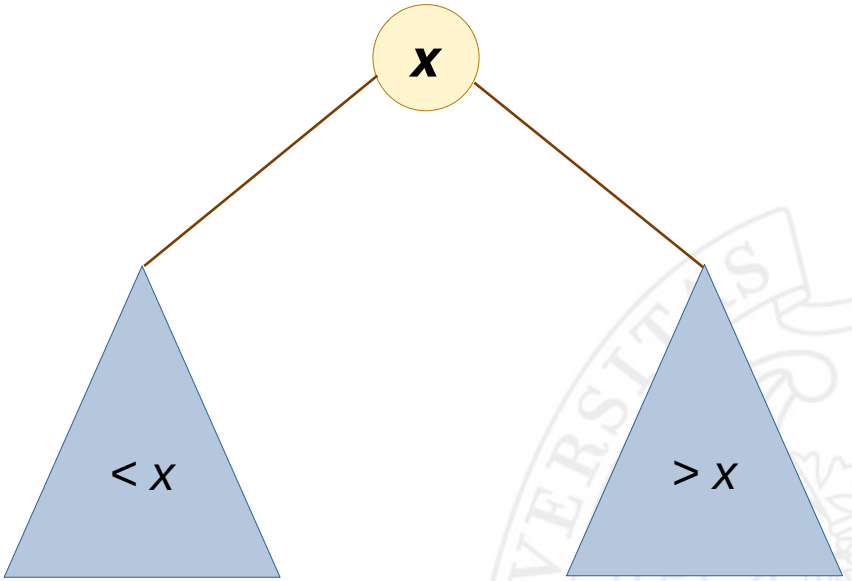
Después de la inserción

Caso 2: `elem` coincide con la raíz



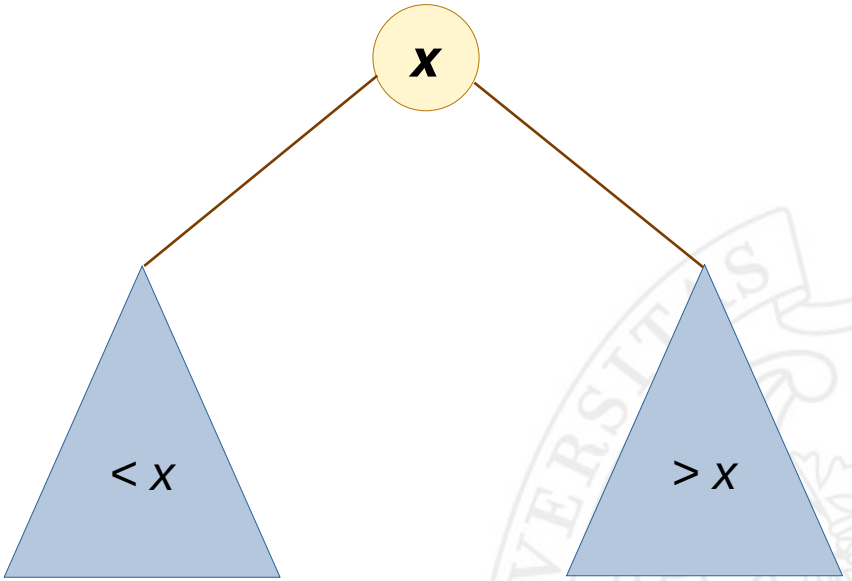
- El elemento que quiero insertar ya está en el árbol. No hacemos nada.

Caso 3: `elem` < raíz



- Insertamos recursivamente `elem` en el hijo izquierdo de la raíz.

Caso 4: `elem` > raíz



- Insertamos recursivamente `elem` en el hijo derecho de la raíz.

Antes de implementar

- En uno de los casos **la raíz del árbol cambia**.

  Caso 1: si el árbol es vacío, la raíz acaba siendo el nodo recién creado.

- Por tanto, la función `insert` debe devolver también **la nueva raíz del árbol**.
- En lugar de:

```
void insert(Node *root, const T &elem);
```

tendremos:

```
Node * insert(Node *root, const T &elem);
```

Nueva raíz

Implementación

```
Node * insert(Node *root, const T &elem) {
  if (root == nullptr) {
     return new Node(nullptr, elem, nullptr);
  } else if (elem < root→elem) {



  } else if (root→elem < elem) {



  } else {

  }
}
```

- **Caso 1:** árbol vacío.

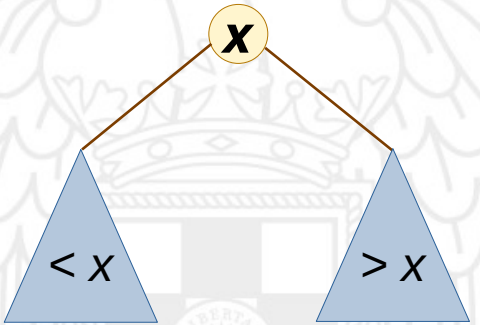
  Creamos un nodo con el valor que se quiere insertar, y ese nodo es la nueva raíz del árbol.

Implementación

```
Node * insert(Node *root, const T &elem) {
  if (root == nullptr) {

  } else if (elem < root→elem) {
     Node *new_root_left = insert(root→left, elem);
     root→left = new_root_left;
     return root;
  } else if (root→elem < elem) {



  } else {

  }
}
```

- **Caso 3:** `elem` < raiz

  Insertamos en el hijo izquierdo. Conectamos la raíz con la nueva raíz del hijo izquierdo.

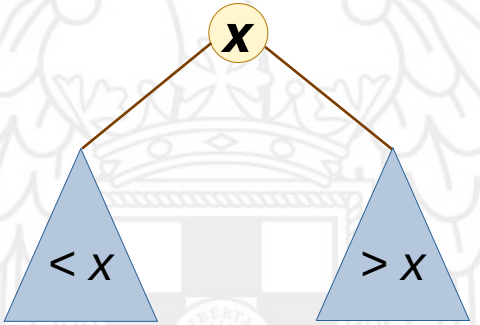


Implementación

```
Node * insert(Node *root, const T &elem) {
  if (root == nullptr) {

  } else if (elem < root→elem) {



  } else if (root→elem < elem) {
     Node *new_root_right = insert(root→right, elem);
     root→right = new_root_right;
     return root;
  } else {

  }
}
```

- **Caso 4:** `elem` > raiz

  Dual al anterior

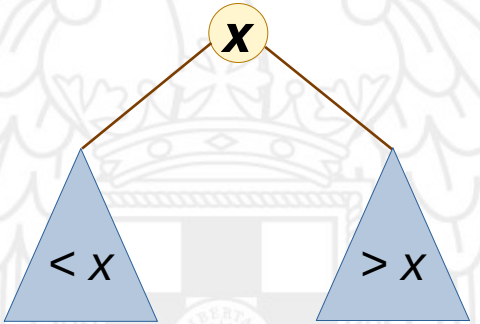


Implementación

```
Node * insert(Node *root, const T &elem) {
  if (root == nullptr) {

  } else if (elem < root→elem) {



  } else if (root→elem < elem) {



  } else {
     return root;
  }
}
```

- **Caso 2:** `elem` == raiz

  No se hace nada. La raíz no varía.



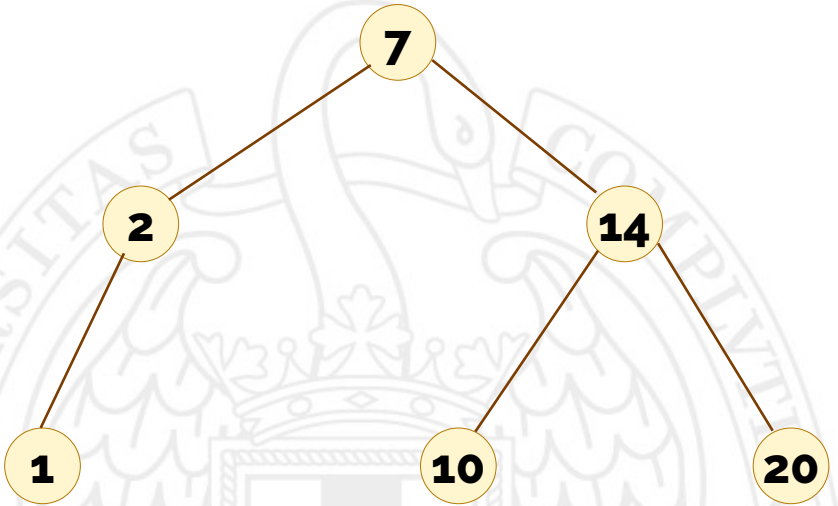
Implementación

```
Node * insert(Node *root, const T &elem) {
  if (root == nullptr) {
     return new Node(nullptr, elem, nullptr);
  } else if (elem < root→elem) {
     Node *new_root_left = insert(root→left, elem);
     root→left = new_root_left;
     return root;
  } else if (root→elem < elem) {
     Node *new_root_right = insert(root→right, elem);
     root→right = new_root_right;
     return root;
  } else {
     return root;
  }
}
```

Ejemplo

- Insertar el valor **9**

```
Node * insert(Node *root, const T &elem) {
  if (root == nullptr) {
     return new Node(nullptr, elem, nullptr);
  } else if (elem < root→elem) {
     Node *new_root_left = insert(root→left, elem);
     root→left = new_root_left;
     return root;
  } else if (root→elem < elem) {
     Node *new_root_right = insert(root→right, elem);
     root→right = new_root_right;
     return root;
  } else {
     return root;
  }
}
```



Coste en tiempo

- El el caso peor, el nodo se inserta en la rama más larga del árbol.
- Por tanto, si h es la altura del árbol, el coste es $O(h)$.
- Y si n es el número de nodos del árbol:
  - Si el árbol está equilibrado, el coste es $O(\log n)$.
  - Si no, el coste es $O(n)$ en el caso peor.

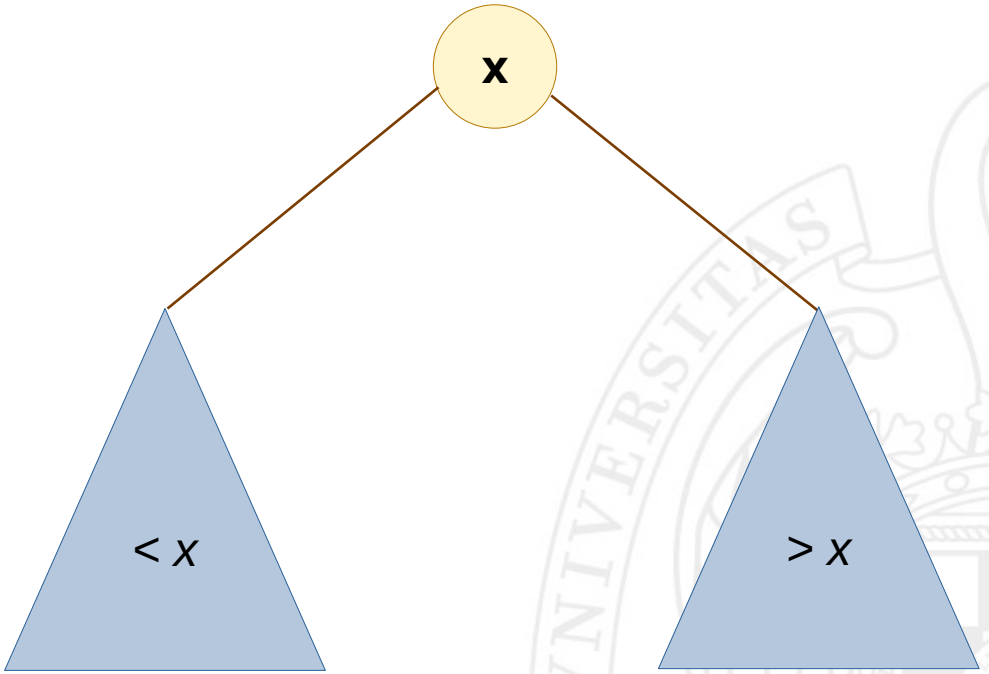
ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS ARBORESCENTES

Eliminación en ABBs

Manuel Montenegro Montes
Departamento de Sistemas Informáticos y Computación
Facultad de Informática – Universidad Complutense de Madrid

Recordatorio: árboles binarios de búsqueda



Objetivo

- Implementar una función `erase(root, elem)`, que elimine el nodo que contenga `elem` del ABB cuya raíz es `root`.
- El árbol resultante también ha de ser un ABB.
- Si `elem` no se encuentra en el ABB, no hace nada.

```
void erase(Node *root, const T &elem);
```

- En algunos casos, la raíz del árbol va a cambiar. Por tanto:

```
Node * erase(Node *root, const T &elem);
```

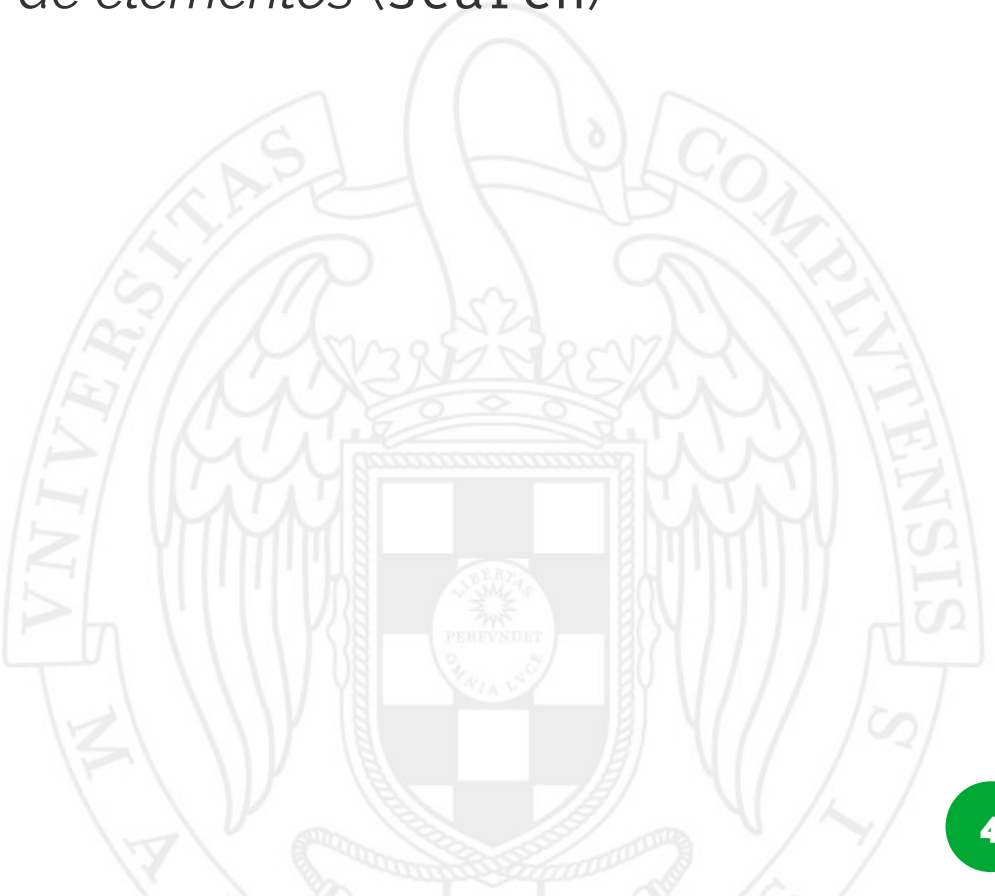
Dos fases

1) Buscar el nodo a eliminar.

   Similar al algoritmo de búsqueda de elementos (`search`)

2) Si se encuentra, eliminarlo.

   ...y poner otra cosa en su lugar.



Fase 1: búsqueda del nodo

```
Node * erase(Node *root, const T &elem) {
  if (root == nullptr) {
     return root;
  } else if (elem < root→elem) {
     Node *new_root_left = erase(root→left, elem);
     root→left = new_root_left;
     return root;
  } else if (root→elem < elem) {
     Node *new_root_right = erase(root→right, elem);
     root→right = new_root_right;
     return root;
  } else {
     return remove_root(root);
  }
}
```

Fase 1: búsqueda del nodo

```
Node * erase(Node *root, const T &elem) {
  if (root == nullptr) {
     return root;
  } else if (elem < root→elem) {



  } else if (root→elem < elem) {



  } else {

  }
}
```

Si llegamos al árbol vacío, no hemos encontrado el nodo a borrar.

Fase 1: búsqueda del nodo

```
Node * erase(Node *root, const T &elem) {
  if (root == nullptr) {

  } else if (elem < root→elem) {
     Node *new_root_left = erase(root→left, elem);
     root→left = new_root_left;
     return root;
  } else if (root→elem < elem) {



  } else {

  }
}
```

Borramos en el hijo izquierdo

Fase 1: búsqueda del nodo

```
Node * erase(Node *root, const T &elem) {
  if (root == nullptr) {

  } else if (elem < root→elem) {



  } else if (root→elem < elem) {
     Node *new_root_right = erase(root→right, elem);
     root→right = new_root_right;
     return root;
  } else {

  }
}
```

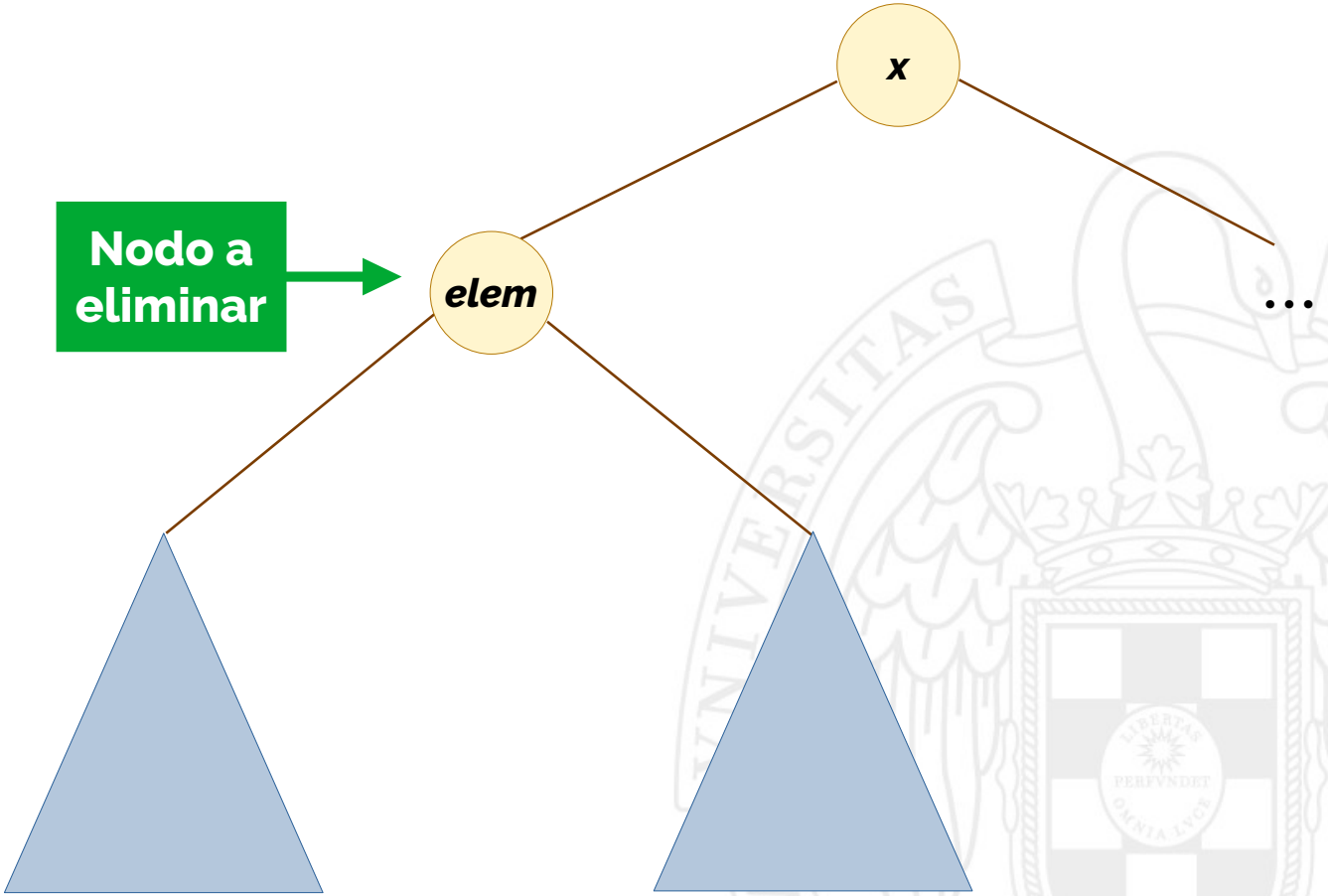
Borramos en el hijo derecho

Fase 1: búsqueda del nodo

```
Node * erase(Node *root, const T &elem) {
  if (root == nullptr) {

  } else if (elem < root→elem) {



  } else if (root→elem < elem) {



  } else {
     return remove_root(root);
  }
}
```

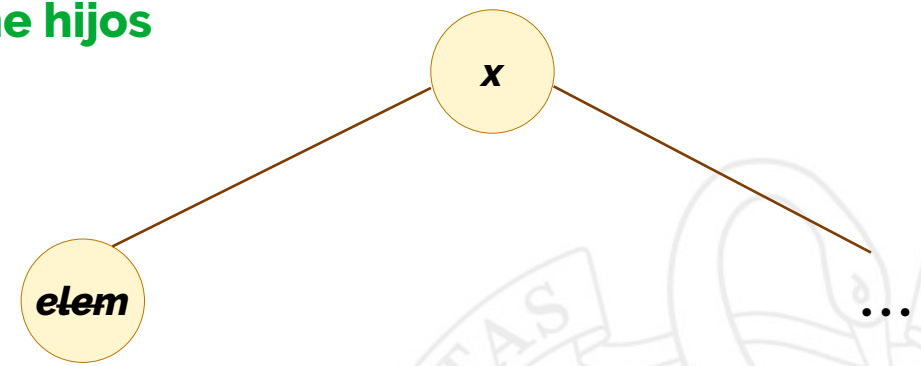
Caso `root→elem == elem`
Pasamos a fase 2

Fase 2: eliminación del nodo



Fase 2: eliminación del nodo

Caso 1: El nodo a eliminar no tiene hijos



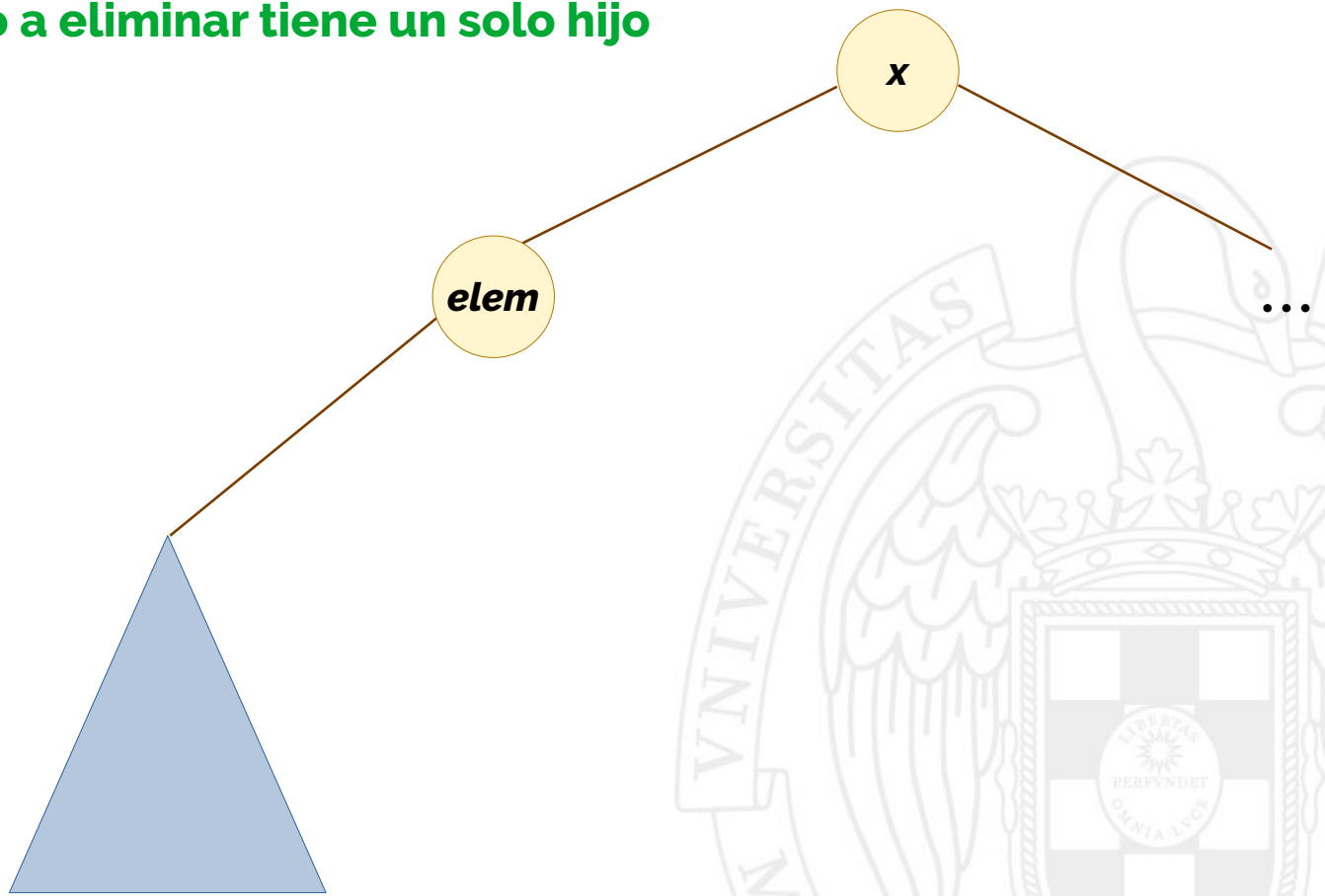
Fase 2: eliminación del nodo

Caso 1: El nodo a eliminar no tiene hijos

```
Node * remove_root(Node *root) {
  Node *left_child = root→left, *right_child = root→right;
  delete root;
  if (left_child == nullptr && right_child == nullptr) {
    return nullptr;
  } else if (left_child == nullptr) {

  } else if (right_child == nullptr) {

  } else {




  }
}
```

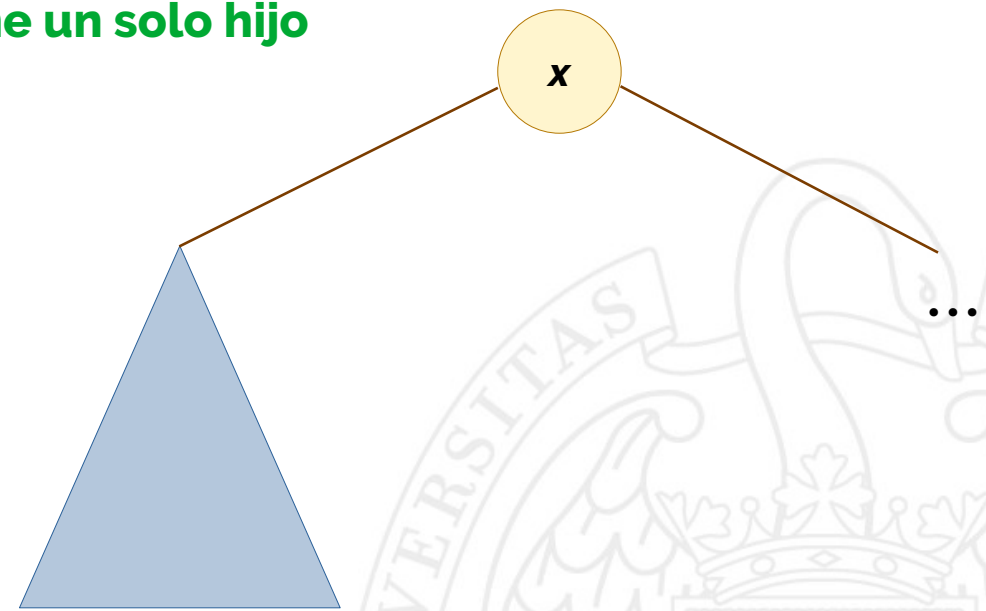
Fase 2: eliminación del nodo

Caso 2: El nodo a eliminar tiene un solo hijo



Fase 2: eliminación del nodo

Caso 2: El nodo a eliminar tiene un solo hijo



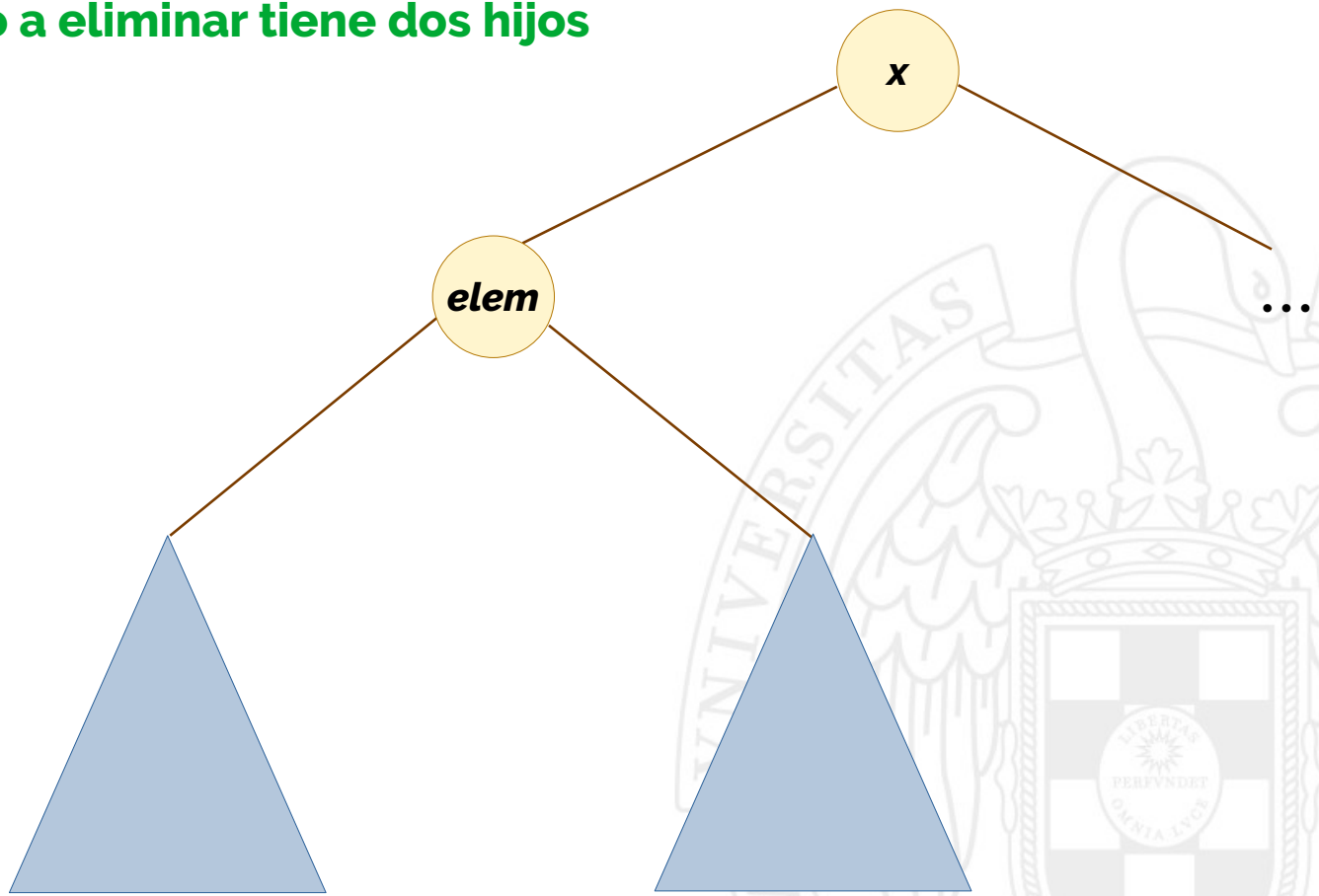
Fase 2: eliminación del nodo

Caso 2: El nodo a eliminar tiene un solo hijo

```
Node * remove_root(Node *root, Node * &new_root) {
  Node *left_child = root→left, *right_child = root→right;
  delete root;
  if (left_child == nullptr && right_child == nullptr) {

  } else if (left_child == nullptr) {
    return right_child;
  } else if (right_child == nullptr) {
    return left_child;
  } else {




  }
}
```

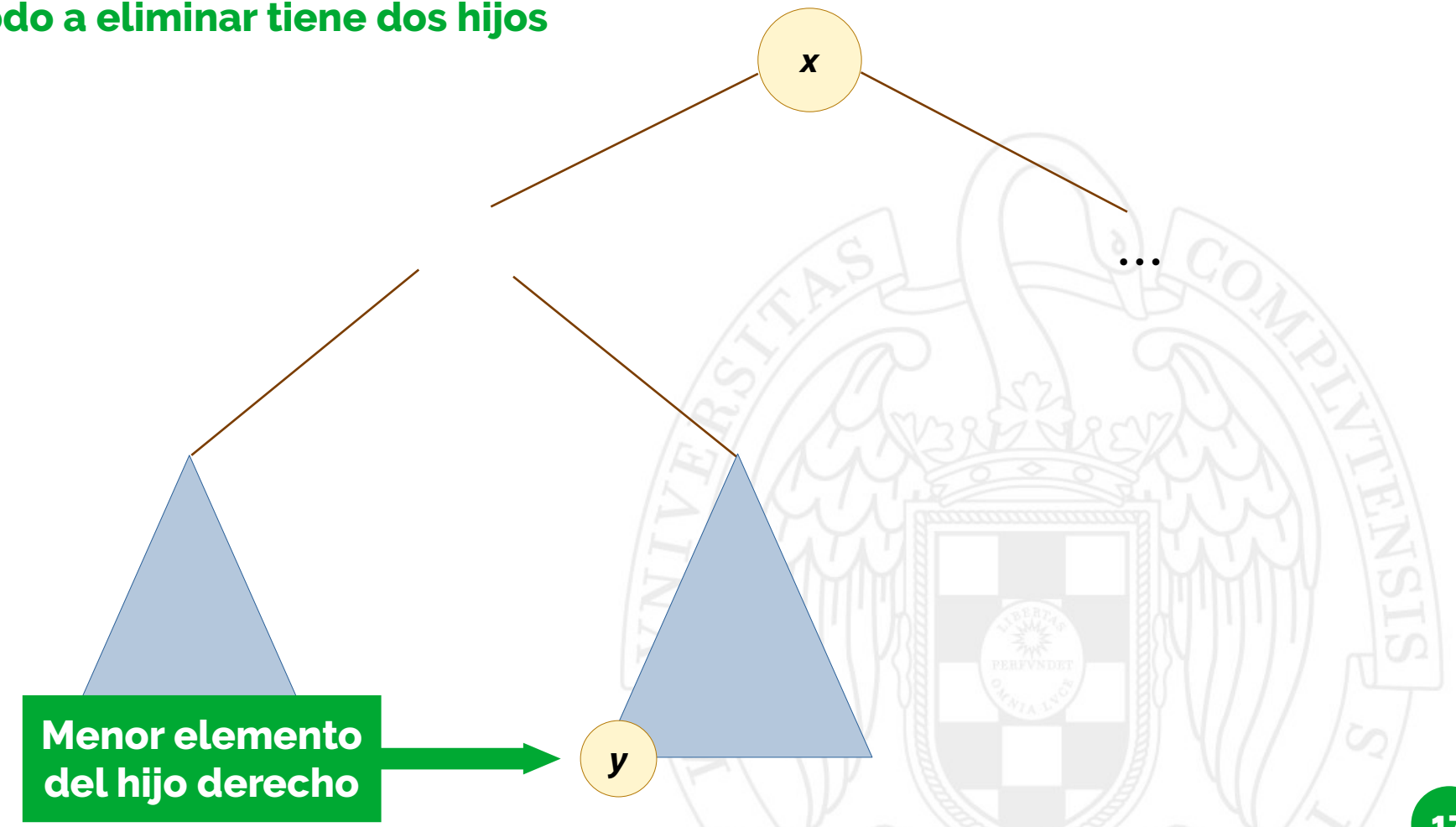
Fase 2: eliminación del nodo

Caso 3: El nodo a eliminar tiene dos hijos



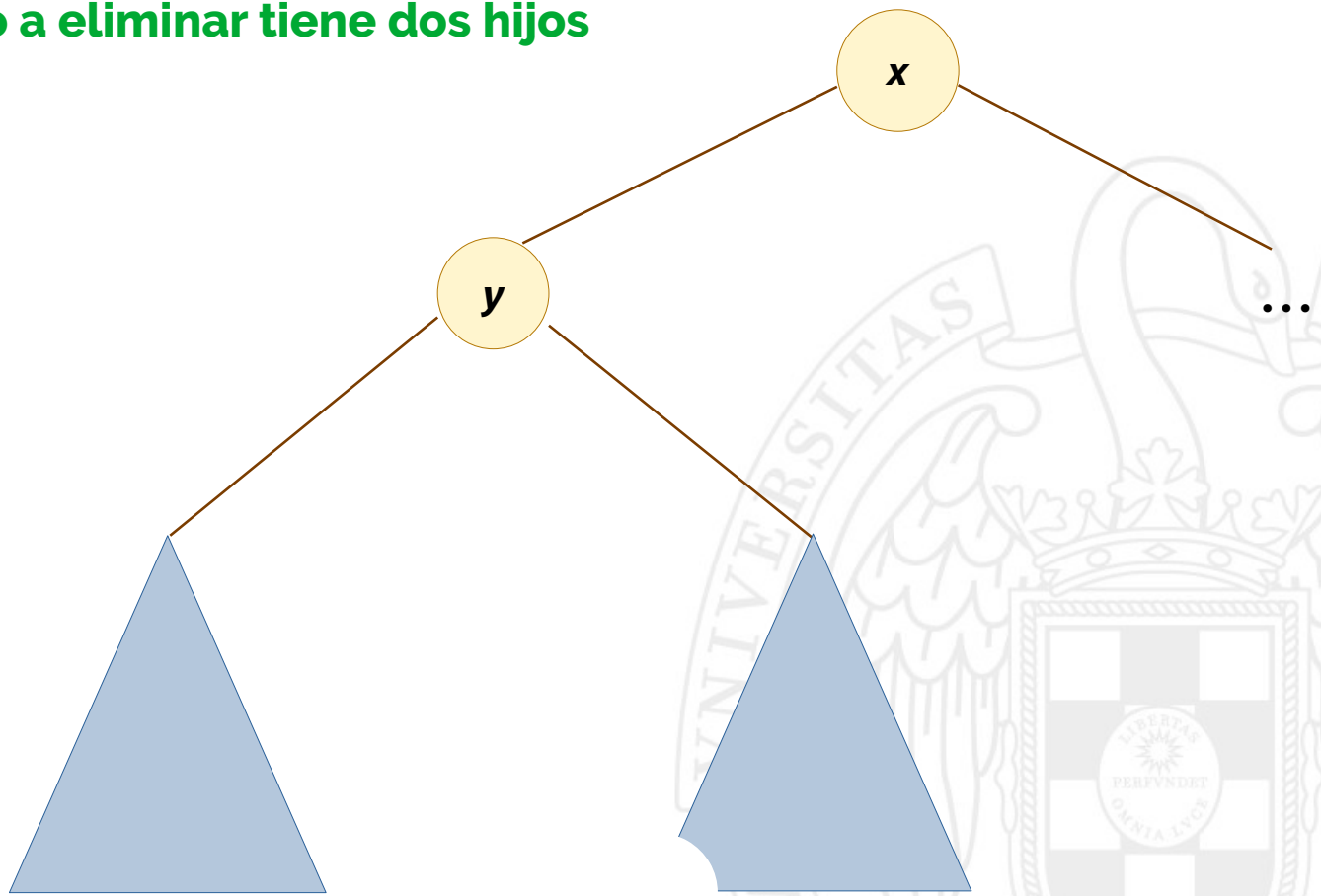
Fase 2: eliminación del nodo

Caso 3: El nodo a eliminar tiene dos hijos



Fase 2: eliminación del nodo

Caso 3: El nodo a eliminar tiene dos hijos



Fase 2: eliminación del nodo

Caso 3: El nodo a eliminar tiene dos hijos

```
Node * remove_root(Node *root, Node * &new_root) {
  Node *left_child = root→left, *right_child = root→right;
  delete root;
  if (left_child == nullptr && right_child == nullptr) {

  } else if (left_child == nullptr) {

  } else if (right_child == nullptr) {

  } else {
    auto [lowest, new_right_root] = remove_lowest(right_child);
    lowest→left = left_child;
    lowest→right = new_right_root;
    return lowest;
  }
}
```

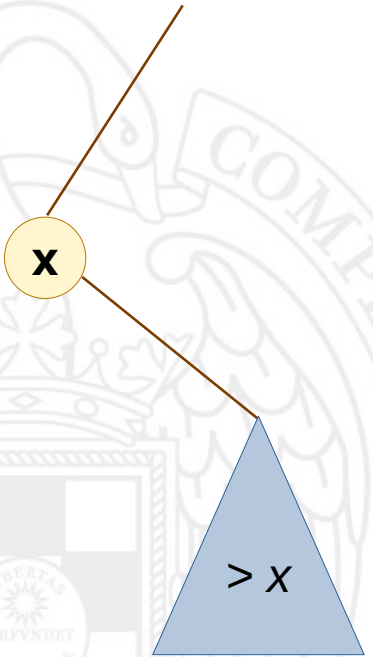
Eliminar el nodo más pequeño de un árbol

```
std::pair<Node *, Node *> remove_lowest(Node *root)
```

- Dado un árbol cuya raíz es `root`, devuelve el nodo con el valor más pequeño y lo «desengancha» del árbol.
- Devuelve dos punteros:
  - Puntero al nodo desenganchado del árbol.
  - Puntero a la nueva raíz del árbol tras desenganchar el nodo.

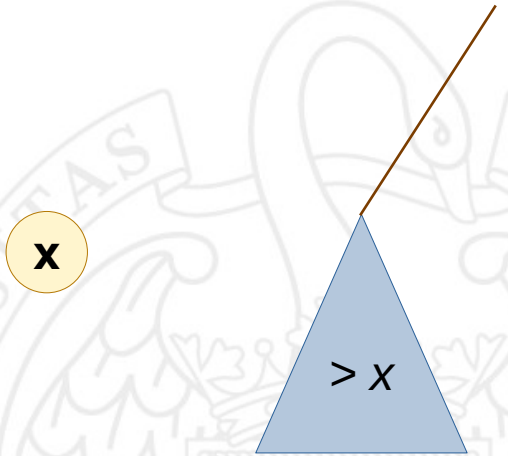
Eliminar el nodo más pequeño de un árbol

```
std::pair<Node *, Node *> remove_lowest(Node *root) {
  assert (root ≠ nullptr);
  if (root→left == nullptr) {
    return {root, root→right};
  } else {




  }
}
```



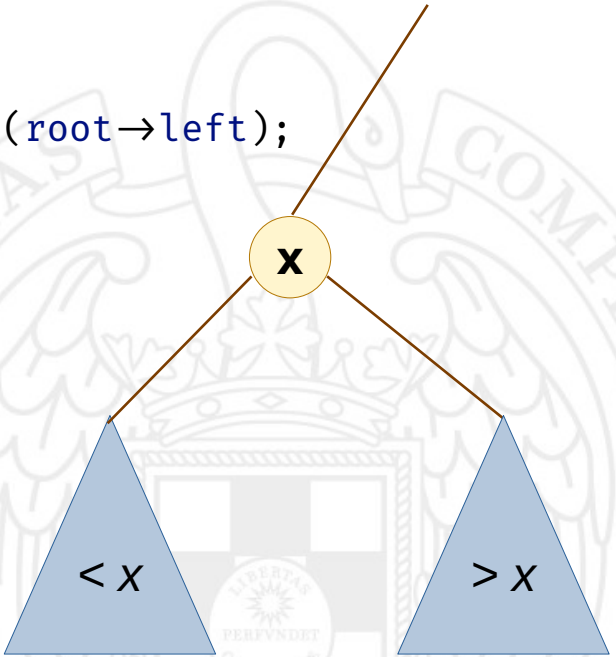
Eliminar el nodo más pequeño de un árbol

```
std::pair<Node *, Node *> remove_lowest(Node *root) {
  assert (root ≠ nullptr);
  if (root→left == nullptr) {
    return {root, root→right};
  } else {




  }
}
```



Eliminar el nodo más pequeño de un árbol

```
std::pair<Node *, Node *> remove_lowest(Node *root) {
  assert (root ≠ nullptr);
  if (root→left == nullptr) {
    return {root, root→right};
  } else {
    auto [removed_node, new_root_left] = remove_lowest(root→left);
    root→left = new_root_left;
    return {removed_node, root};
  }
}
```



Recapitulando

- `erase(root, elem)`

  Busca el elemento que se quiere eliminar. Cuando se encuentra, llama a `remove_root` sobre el nodo que contiene el elemento.

- `remove_root(root)`

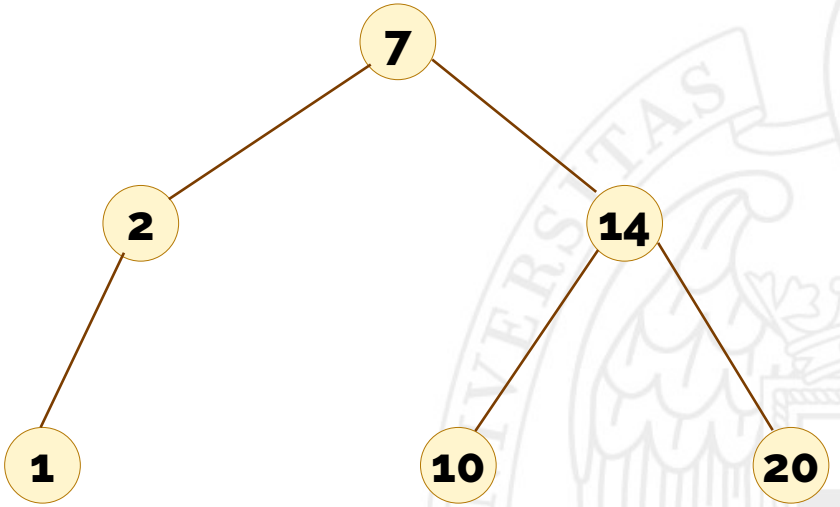
  Elimina la raíz del árbol, devolviendo la nueva raíz. Si la raíz tiene dos hijos, la nueva raíz es el nodo con el menor valor del hijo derecho. Se llama a `remove_lowest` para obtener este último nodo.

- `remove_lowest(root)`

  Devuelve el nodo que contiene el valor más pequeño del árbol cuya raíz es `root` y lo desengancha del árbol.

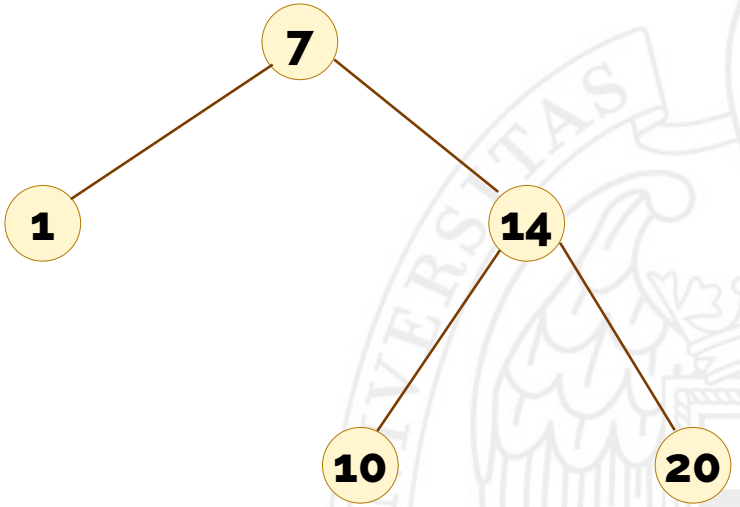
Ejemplo

- Eliminar el valor **2**



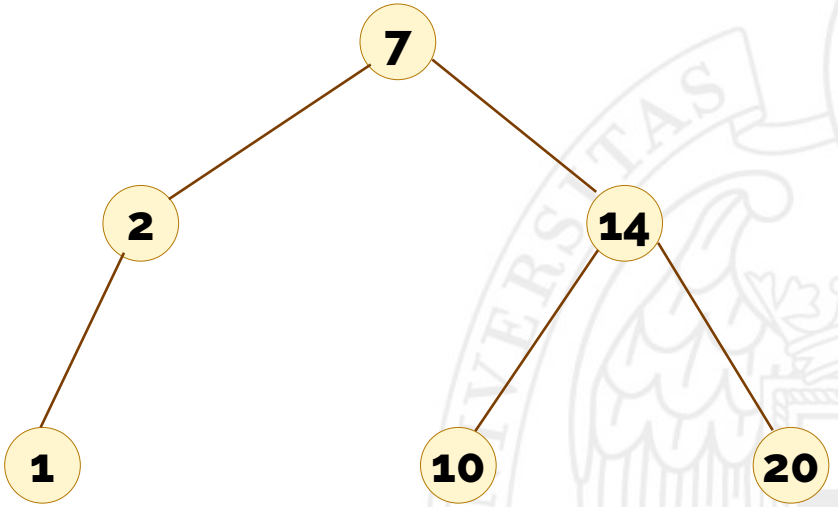
Ejemplo

- Eliminar el valor **2**



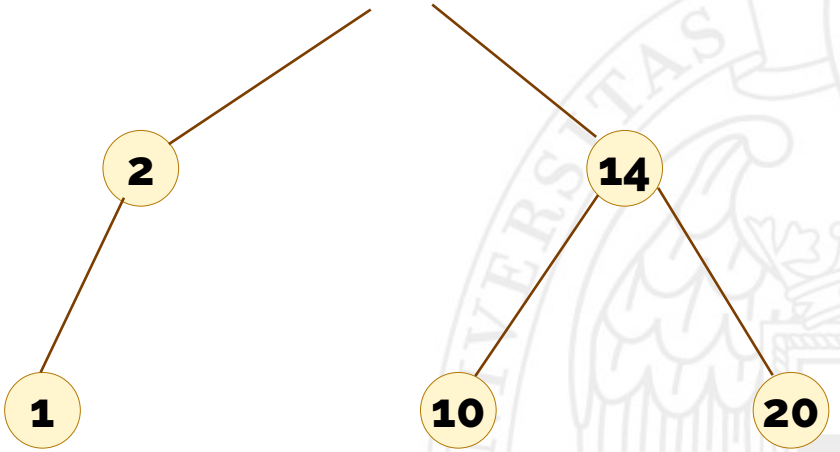
Ejemplo

- Eliminar el valor **7**



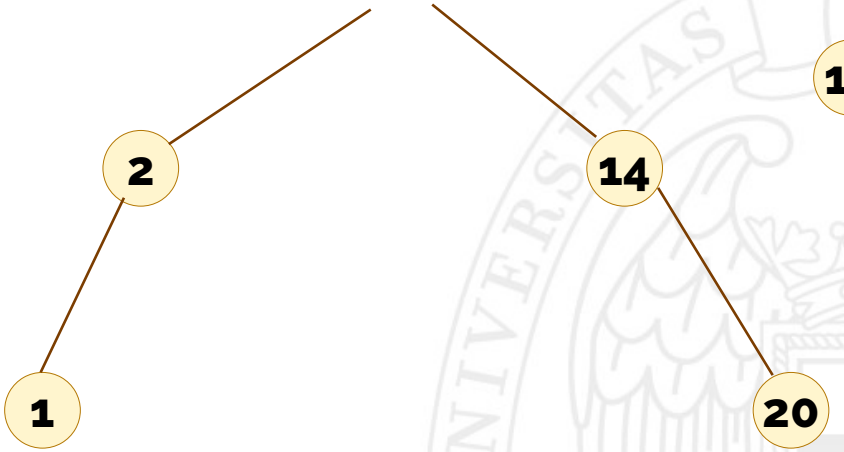
Ejemplo

- Eliminar el valor **7**



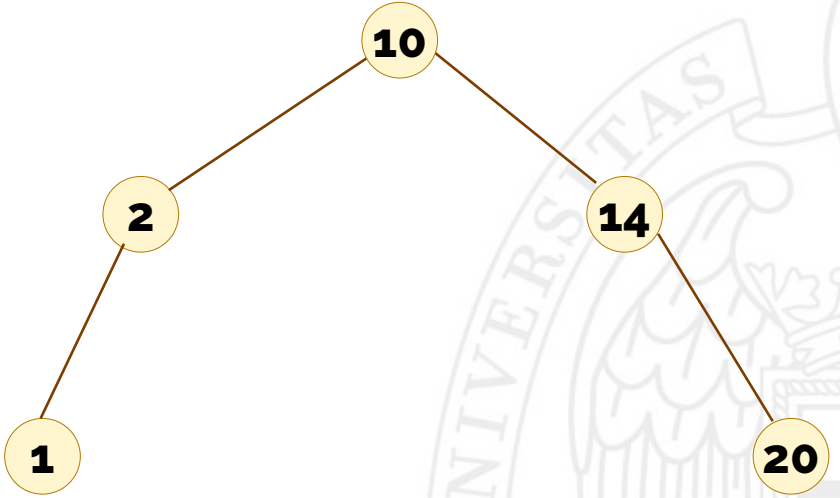
Ejemplo

- Eliminar el valor **7**



Ejemplo

- Eliminar el valor **7**



Coste en tiempo

- Si h es la altura del árbol, el coste es $O(h)$.
- Y si n es el número de nodos del árbol:
  - Si el árbol está equilibrado, el coste es $O(\log n)$.
  - Si no, el coste es $O(n)$ en el caso peor.

ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS ARBORESCENTES

Implementación del TAD Conjunto mediante ABBs

Manuel Montenegro Montes
Departamento de Sistemas Informáticos y Computación
Facultad de Informática – Universidad Complutense de Madrid

Operaciones en el TAD Conjunto

- Constructoras:
  - Crear un conjunto vacío: ***create_empty***
- Mutadoras:
  - Añadir un elemento al conjunto: ***insert***
  - Eliminar un elemento del conjunto: ***erase***
- Observadoras:
  - Averiguar si un elemento está en el conjunto: ***contains***
  - Saber si el conjunto está vacío: ***empty***
  - Saber el tamaño del conjunto: ***size***

Dos implementaciones

- Mediante **listas**.
- Mediante **árboles binarios de búsqueda**.

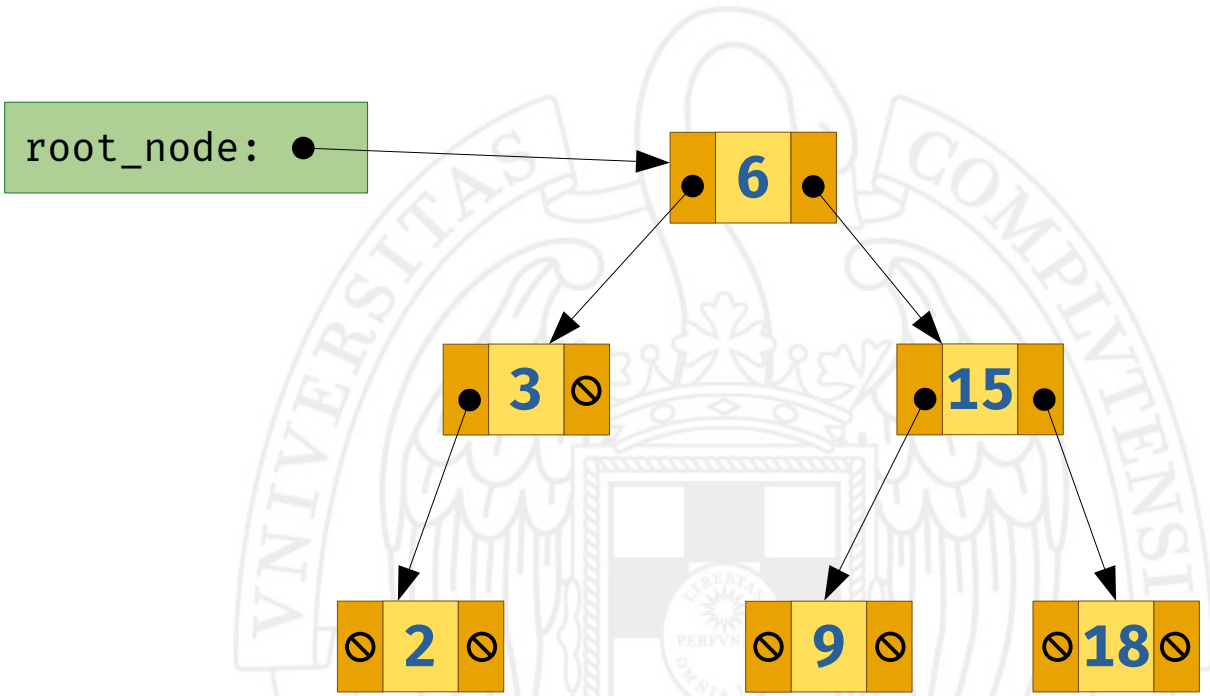
Interfaz de SetTree

```
template <typename T>
class SetTree {
public:
  SetTree();
  SetTree(const SetTree &other);
  ~SetTree();

  void insert(const T &elem);
  void erase(const T &elem);

  bool contains(const T &elem) const;
  int size() const;
  bool empty() const;

private:
   ...
};
```

Implementación de SetTree

```
template <typename T>
class SetTree {
public:
   ...
private:

  struct Node {
    T elem;
    Node *left, *right;
     ...
  };

  Node *root_node;
};
```

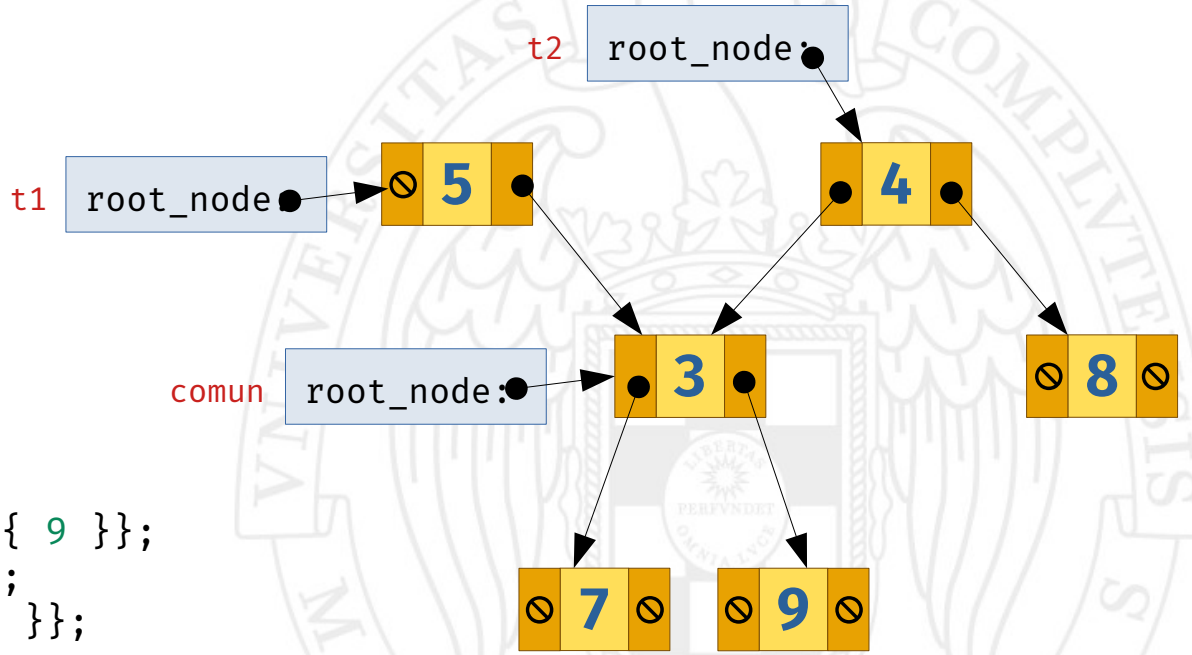
{2, 3, 6, 9, 15, 18}



Sobre la compartición

Anteriormente...

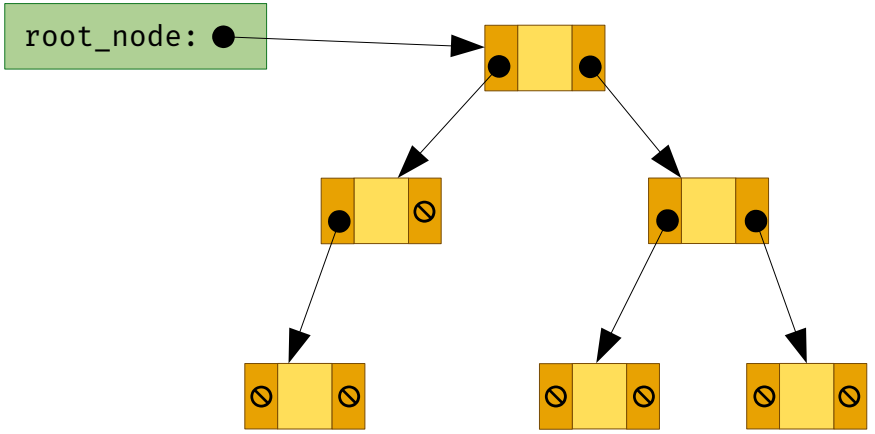
- Implementamos un TAD para árboles binarios.
- Utilizábamos *smart pointers* para enlazar los nodos, porque árboles binarios distintos podían compartir nodos:

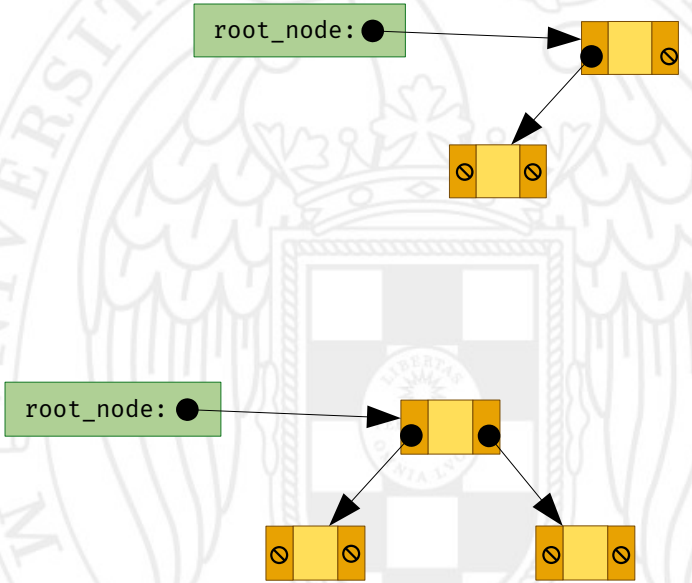



```
BinTree<int> comun = {{ 7 }, 3, { 9 }};
BinTree<int> t1 = {{}, 5, comun};
BinTree<int> t2 = {comun, 4, { 8 }};
```

Pero aquí...

- Implementamos un TAD para conjuntos.
- Cada objeto de la clase `SetTree` apunta a la raíz de su propio árbol de nodos.
- **No hay compartición** entre los nodos de dos `SetTree` distintos.





Pero aquí...

- Implementamos un TAD para conjuntos.
- Cada objeto de la clase `SetTree` apunta a la raíz de su propio árbol de nodos.
- **No hay compartición** entre los nodos de dos `SetTree` distintos.
- Consecuencias:
  - No necesitamos punteros inteligentes.
  - Cada `SetTree` es responsable de liberar sus nodos.
  - El constructor de copia de `SetTree` debe copiar los nodos del conjunto origen al conjunto destino.

Constructores y destructor de SetTree

```
template <typename T>
class SetTree {
public:
  SetTree(): root_node(nullptr) { }

  SetTree(const SetTree &other): root_node(copy_nodes(other.root_node)) { }

  ~SetTree() {
    delete_nodes(root_node);
  }

private:
  ...
  Node *root_node;
};
```

Operaciones consultoras y mutadoras

Métodos auxiliares

```
template <typename T>
class SetTree {
public:




private:
  Node *root_node;
   ...
  static Node * insert(Node *root, const T &elem);
  static bool search(const Node *root, const T &elem);
  static Node * erase(Node *root, const T &elem);
   ...
};
```

Métodos de la interfaz

```
template <typename T>
class SetTree {
public:

  void insert(const T &elem) { root_node = insert(root_node, elem); }
  void erase(const T &elem) { root_node = erase(root_node, elem); }
  bool contains(const T &elem) const { return search(root_node, elem); }
  bool empty() const { return root_node == nullptr; }
  int size() const { return num_nodes(root_node); }



private:
  Node *root_node;
  ...
  static Node * insert(Node *root, const T &elem);
  static bool search(const Node *root, const T &elem);
  static Node * erase(Node *root, const T &elem);
  ...
};
```

Coste de las operaciones

Operación	Árbol equilibrado	Árbol no equilibrado
constructor	$O(1)$	$O(1)$
`empty`	$O(1)$	$O(1)$
`size`	$O(n)$	$O(n)$
`contains`	$O(\log n)$	$O(n)$
`insert`	$O(\log n)$	$O(n)$
`erase`	$O(\log n)$	$O(n)$

n = número de elementos del conjunto

Mejorando la operación `size()`

Mejorando el coste de `size()`

- Contar los nodos de un árbol binario de búsqueda tiene coste lineal con respecto al número de nodos.
- Es posible mejorar ese coste incluyendo un atributo `num_elems` en la clase `SetTree` y actualizándolo cada vez que haya una inserción o eliminación.

Mejorando el coste de `size()`

```
template <typename T>
class SetTree {
public:

  int size() const { return num_elems; }

  void insert(const T &elem) {
    root_node = insert(root_node, elem);
    num_elems++;
  }
  void erase(const T &elem) {
    root_node = erase(root_node, elem);
    num_elems--;
  }

   ...
private:
  Node *root_node;
  int num_elems;
   ...
};
```

¡Incorrecto!

¡Incorrecto!

¿Por qué no es correcto `insert`?

- Porque si el elemento a insertar ya se encuentra en el conjunto, no aumenta el número de elementos del conjunto.
- En este caso, no tenemos que incrementar `num_elems`.
- Cambiamos la función auxiliar `insert`:

```
Node * insert(Node *node, const T &elem)
```

por:

```
pair<Node *, bool> insert(Node *node, const T &elem)
```

- La función devuelve `true` si el `elem` se ha insertado realmente, o `false` si no se ha insertado porque ya existía en el conjunto.

¿Por qué no es correcto `erase`?

- Porque si el elemento a eliminar no se encuentra en el conjunto, la función `erase` no elimina nada.
- En este caso, no tenemos que decrementar `num_elems`.
- Cambiamos la función auxiliar `erase`:

```
Node * erase(Node *node, const T &elem)
```

por:

```
pair<Node *, bool> erase(Node *node, const T &elem)
```

Cambios en `insert`

```
static std::pair<Node *, bool> insert(Node *root, const T &elem) {
  if (root == nullptr) {
    return {new Node(nullptr, elem, nullptr), true};
  } else if (elem < root->elem) {
    auto [new_root_left, inserted] = insert(root->left, elem);
    root->left = new_root_left;
    return {root, inserted};
  } else if (root->elem < elem) {
    auto [new_root_right, inserted] = insert(root->right, elem);
    root->right = new_root_right;
    return {root, inserted};
  } else {
    return {root, false};
  }
}
```

Cambios en la clase SetTree

```
template <typename T>
class SetTree {
public:
  ...
  void insert(const T &elem) {
    auto [new_root, inserted] = insert(root_node, elem);
    root_node = new_root;
    if (inserted) { num_elems++; }
  }

  void erase(const T &elem) {
    auto [new_root, removed] = erase(root_node, elem);
    root_node = new_root;
    if (removed) { num_elems--; }
  }
  ...
private:
  Node *root_node;
  int num_elems;
  ...
};
```

Coste de las operaciones

Operación	Árbol equilibrado	Árbol no equilibrado
constructor	$O(1)$	$O(1)$
`empty`	$O(1)$	$O(1)$
`size`	$O(1)$	$O(1)$
`contains`	$O(\log n)$	$O(n)$
`insert`	$O(\log n)$	$O(n)$
`erase`	$O(\log n)$	$O(n)$

n = número de elementos del conjunto

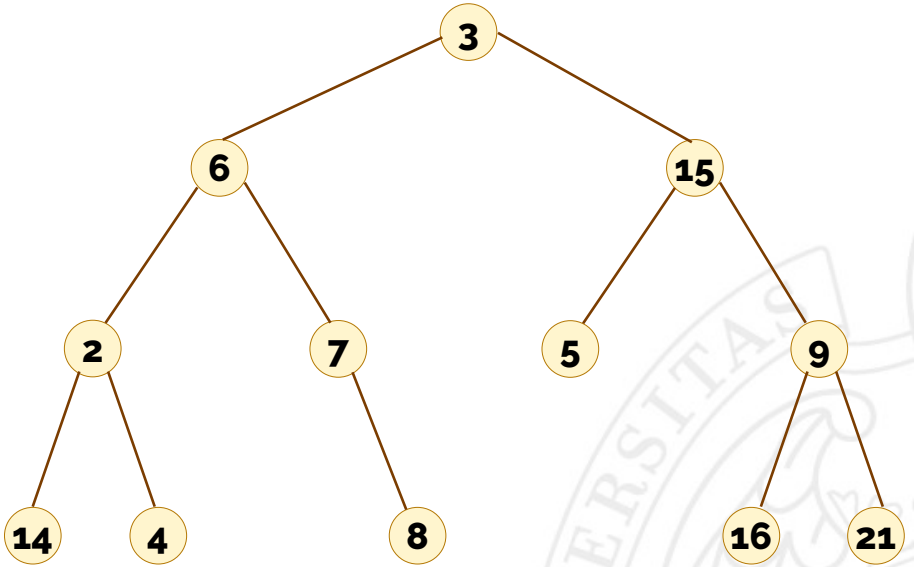
ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS ARBORESCENTES

Recorrido en inorden iterativo (1)

Manuel Montenegro Montes
Departamento de Sistemas Informáticos y Computación
Facultad de Informática – Universidad Complutense de Madrid

Recordatorio: recorrido en inorden



14 2 4 6 7 8 3 5 15 16 9 21

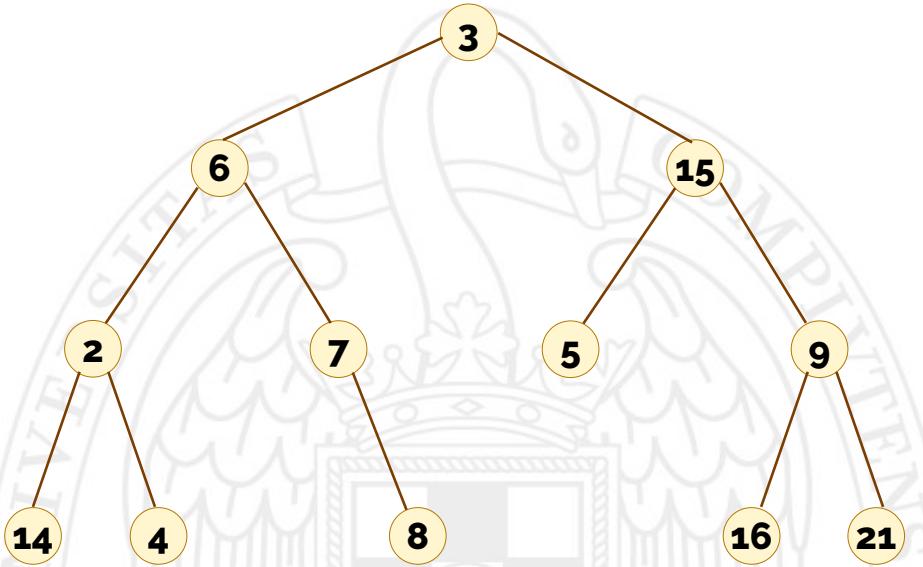
inorden hijo izquierdo — *raíz* — *inorden hijo derecho*

Observaciones previas

Consideraciones previas

- *¿Cuál es el primer nodo que se visita?*

  El que se alcanza tras descender por los hijos izquierdos hasta que no se pueda más.

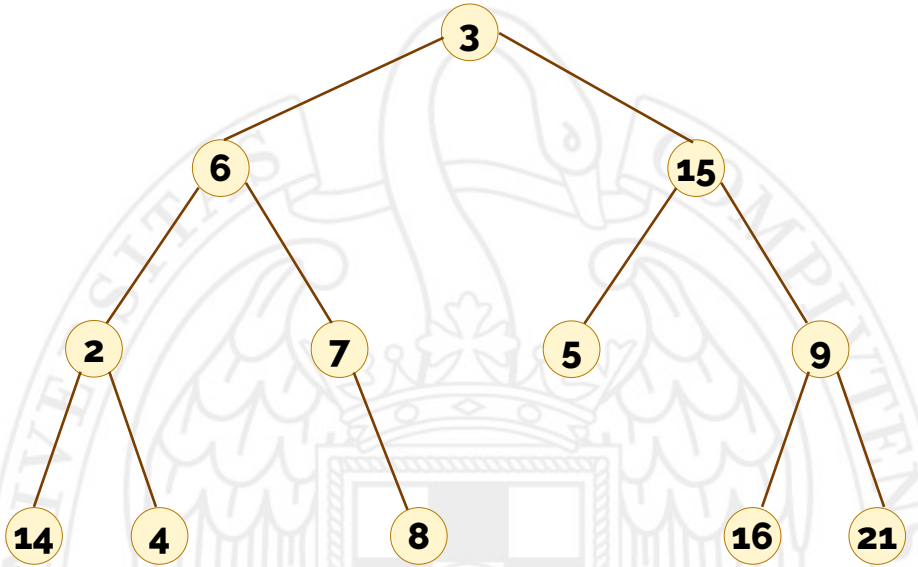


Consideraciones previas

- *Acabo de visitar un nodo. ¿Cuál es el siguiente?*

  Baja al hijo derecho, y busca allí el primer nodo que habría que visitar:

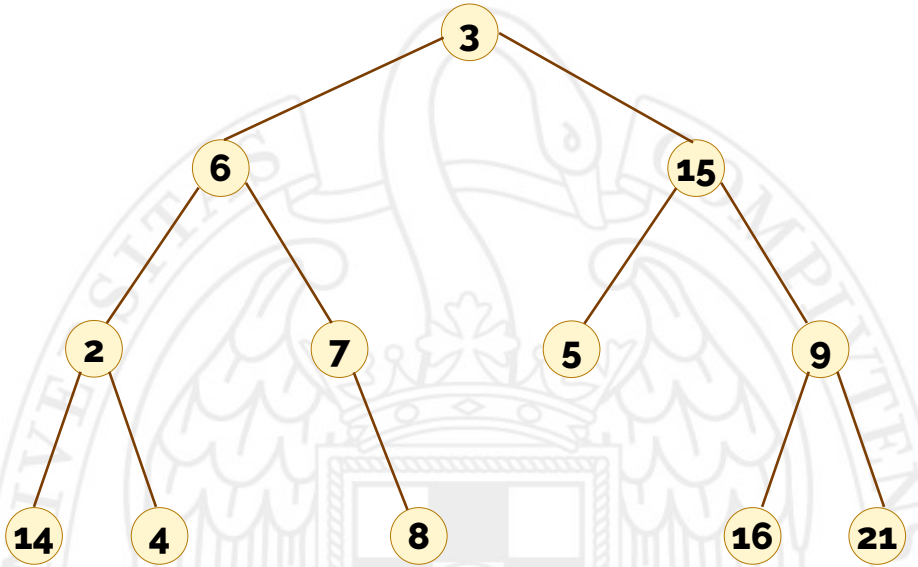
  El que se alcanza tras descender por los hijos izquierdos hasta que no se pueda más.



Consideraciones previas

- *¿Y si no hay hijo derecho?*

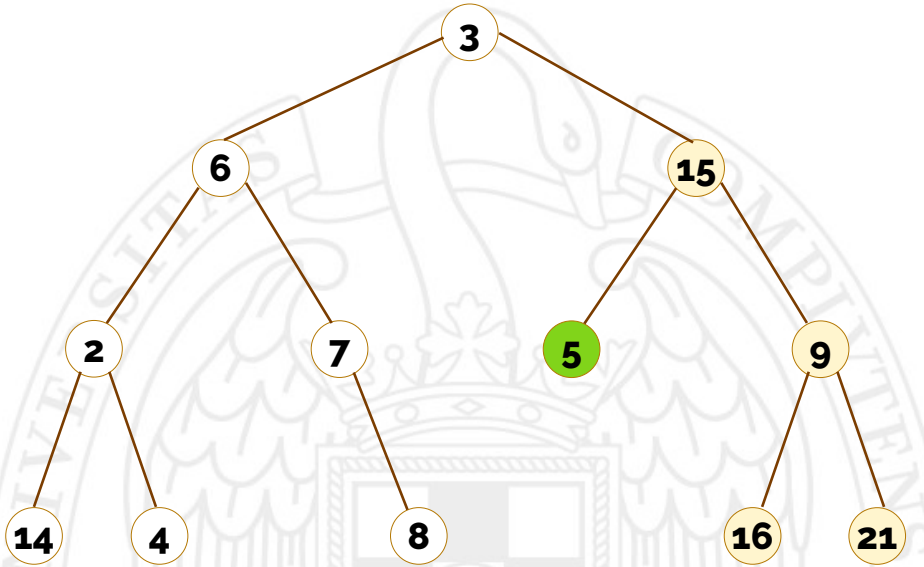
  Hay que subir por el árbol hasta el primer antecesor no visitado.



Consideraciones previas

- *¿Y si no hay hijo derecho?*

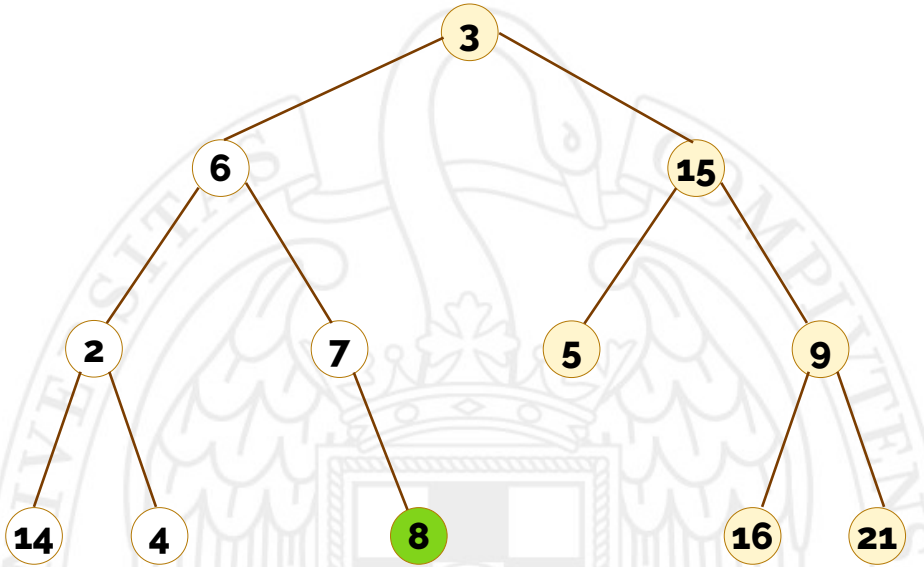
  Hay que subir por el árbol hasta el primer antecesor no visitado.



Consideraciones previas

- *¿Y si no hay hijo derecho?*

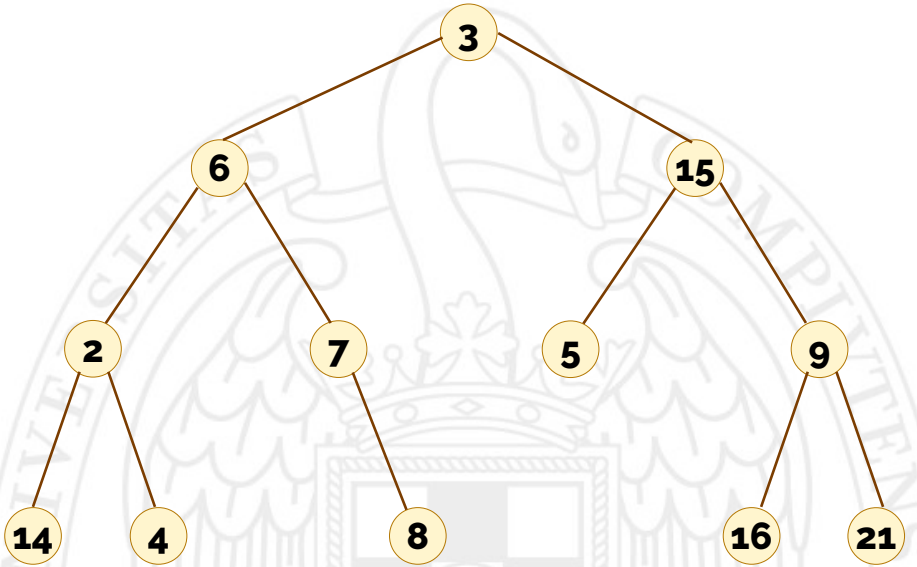
  Hay que subir por el árbol hasta el primer antecesor no visitado.



Consideraciones previas

- *¿Cómo subo hasta el antecesor no visitado? Solamente tengo punteros a los hijos, pero no al nodo padre.*

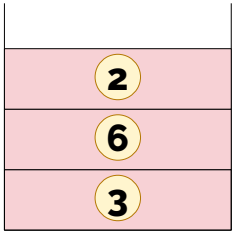
  Es necesaria una **pila** que almacene los antecesores no visitados hasta uno dado.

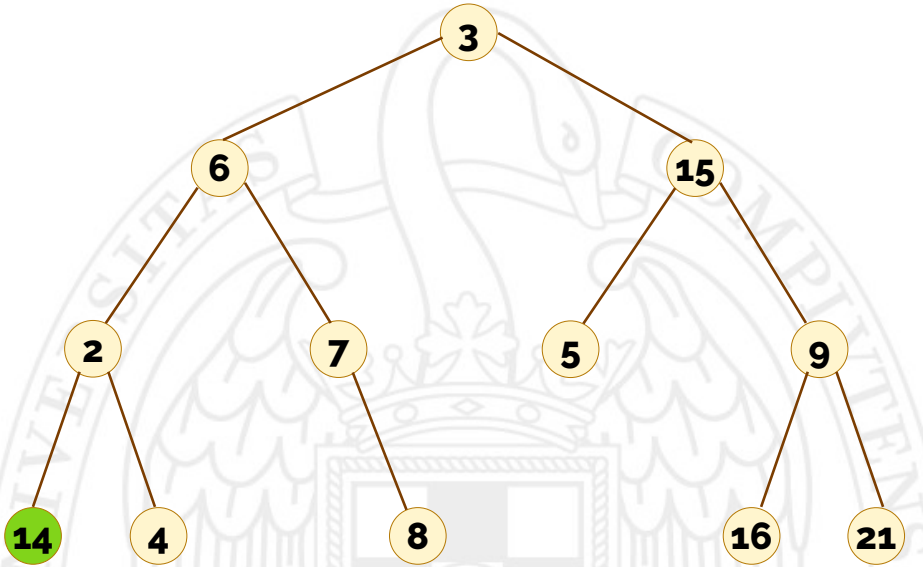


Consideraciones previas

- *¿Cómo subo hasta el antecesor no visitado? Solamente tengo punteros a los hijos, pero no al nodo padre.*

  Es necesaria una **pila** que almacene los antecesores no visitados hasta uno dado.

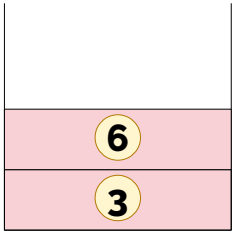


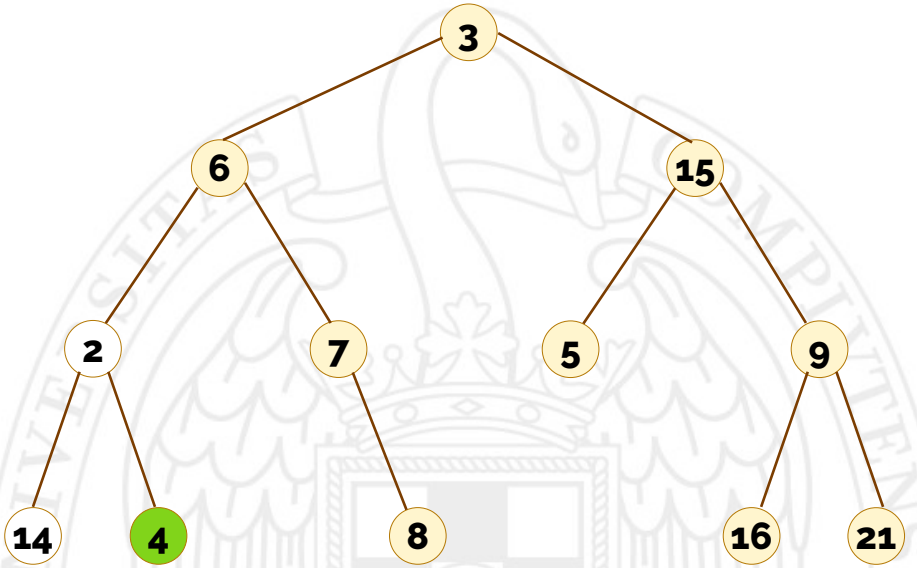


Consideraciones previas

- *¿Cómo subo hasta el antecesor no visitado? Solamente tengo punteros a los hijos, pero no al nodo padre.*

  Es necesaria una **pila** que almacene los antecesores no visitados hasta uno dado.

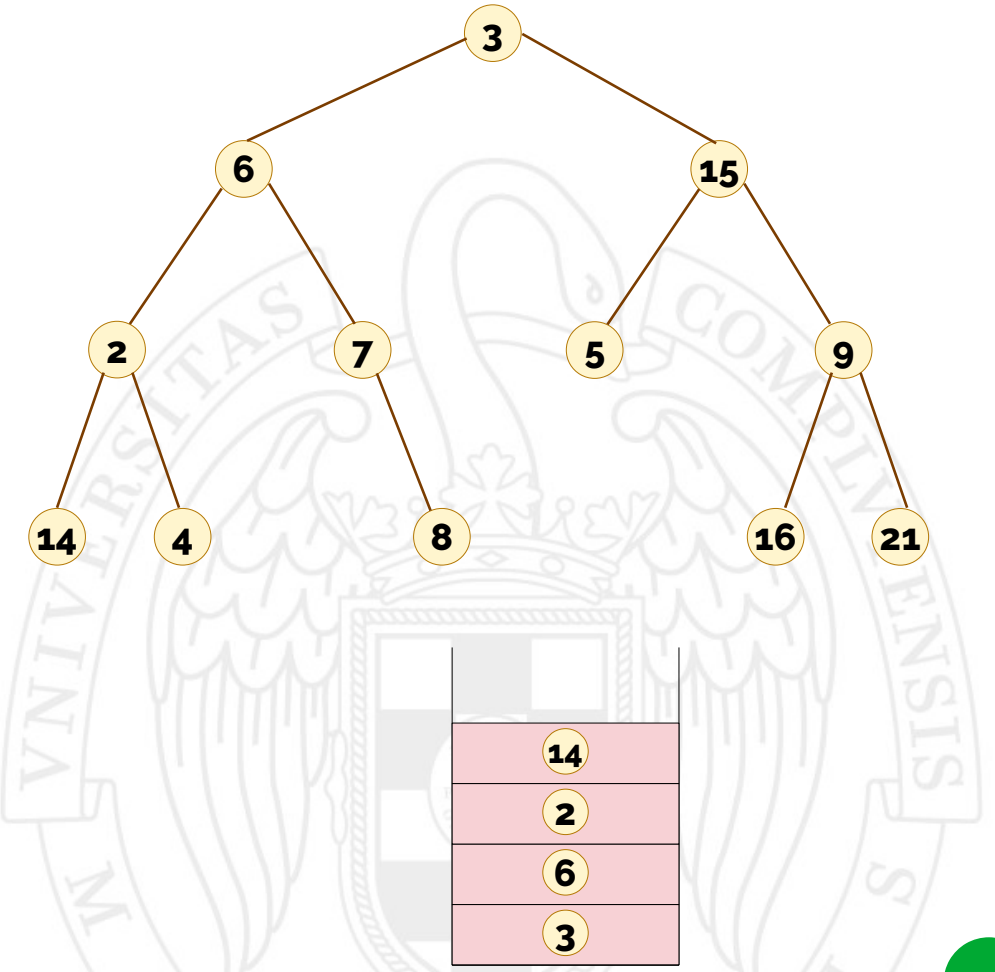




Algoritmo de recorrido

Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

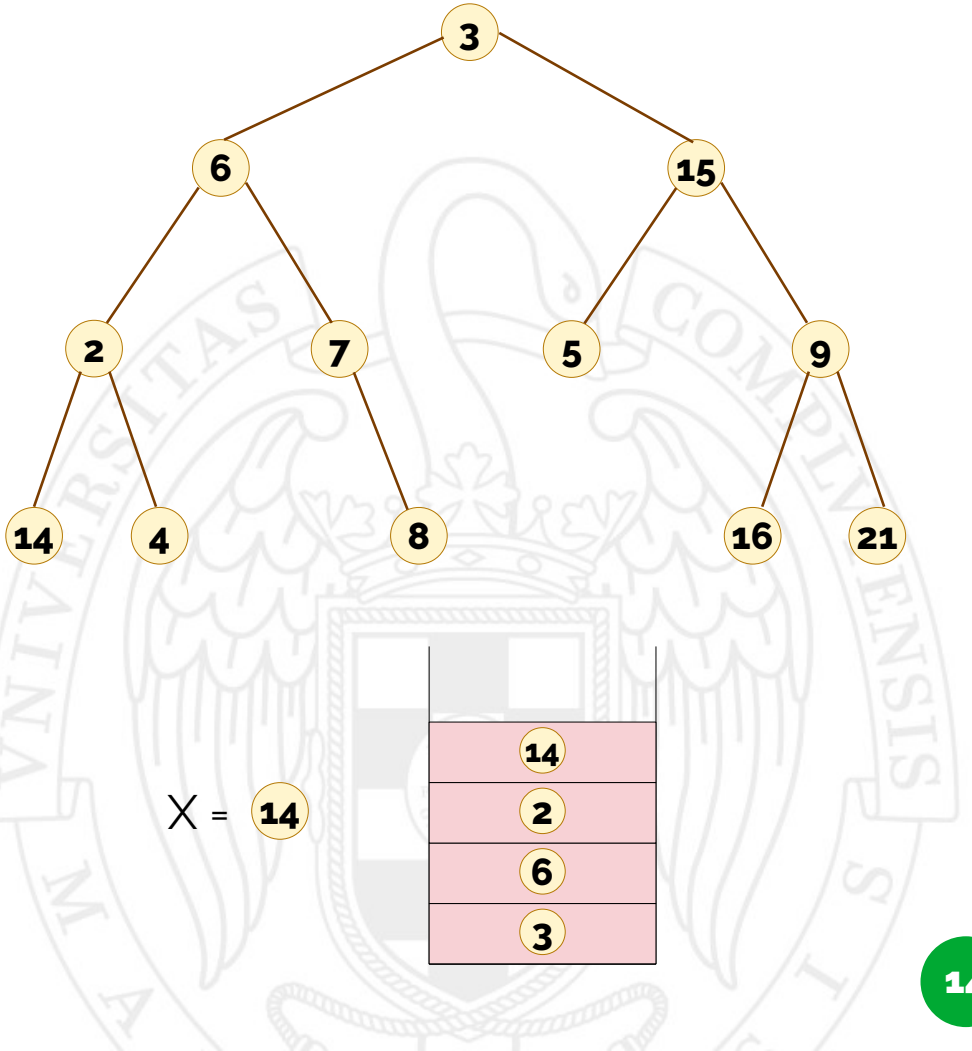


Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.

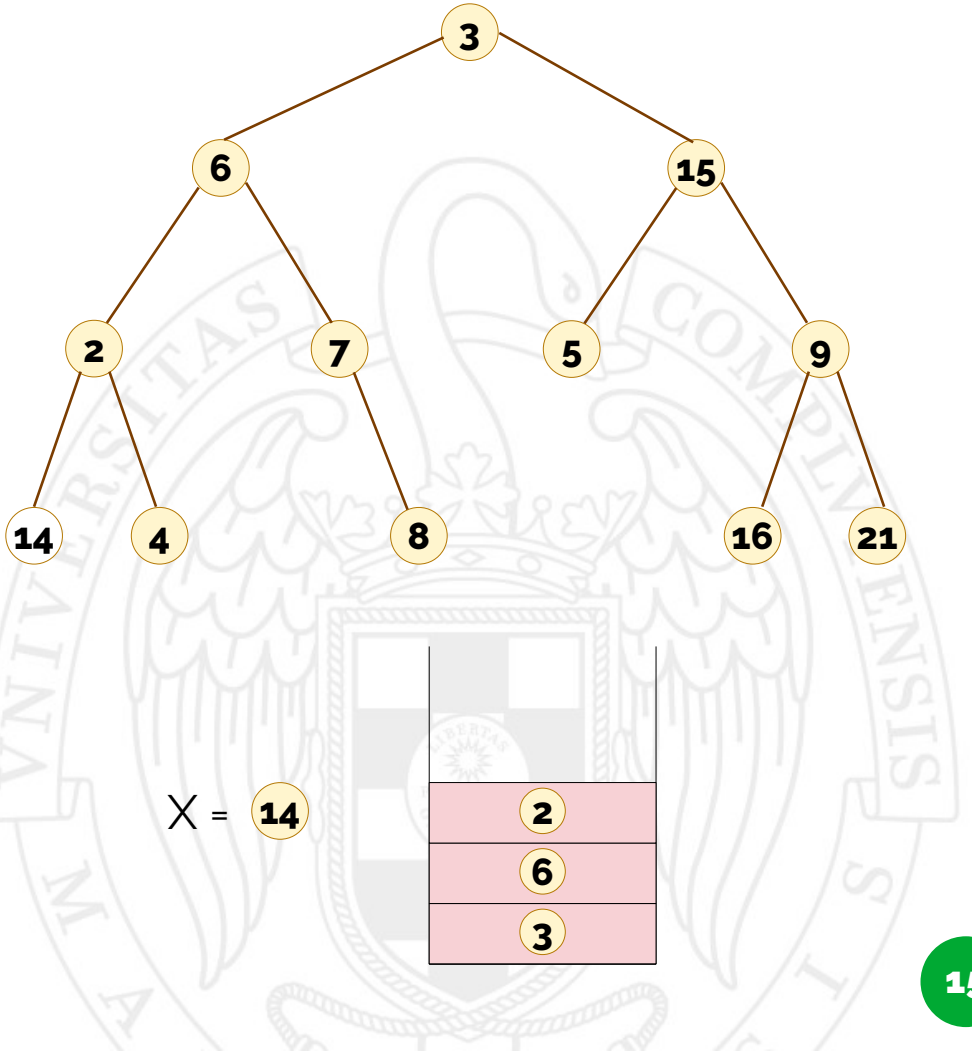


Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.

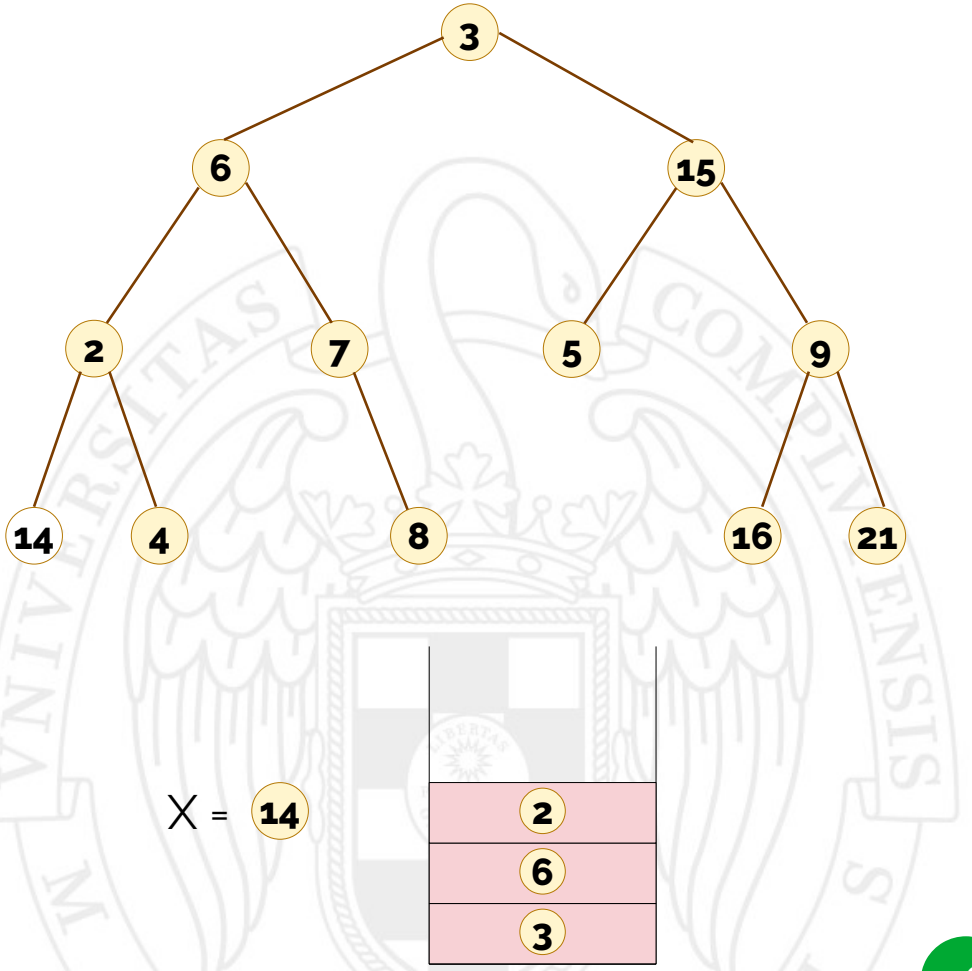


Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - ...
  - ...

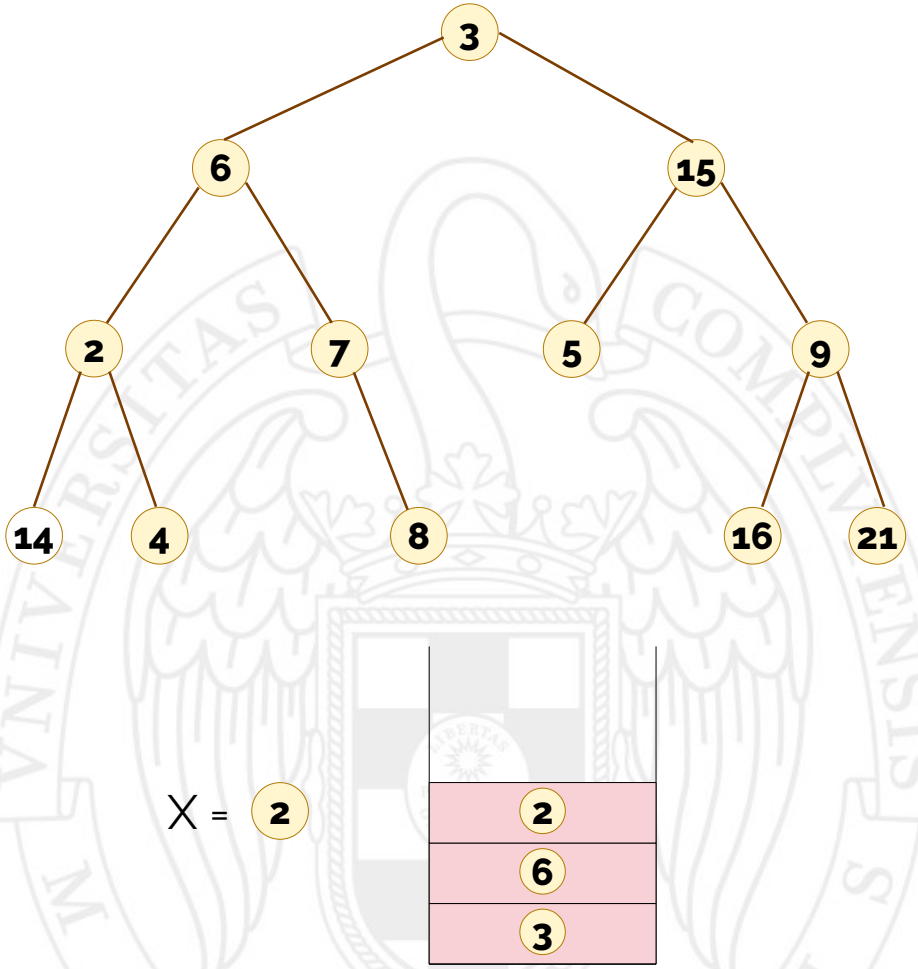


Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - ...
  - ...

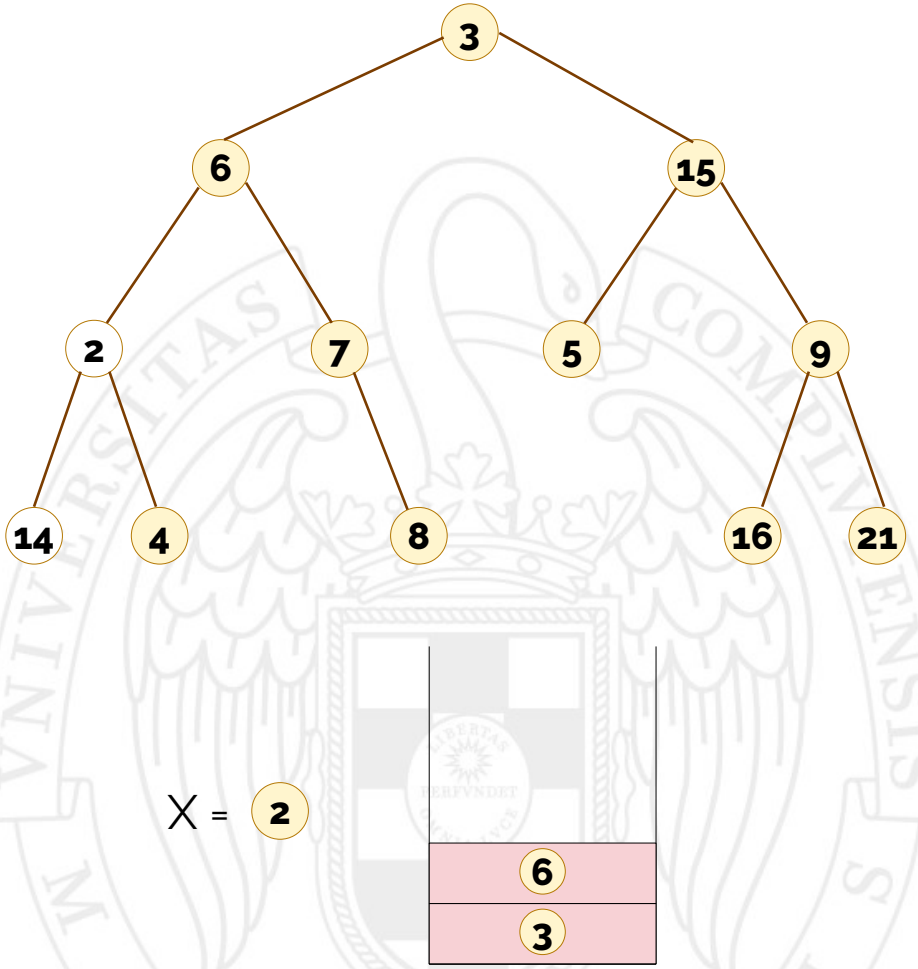


Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - ...
  - ...

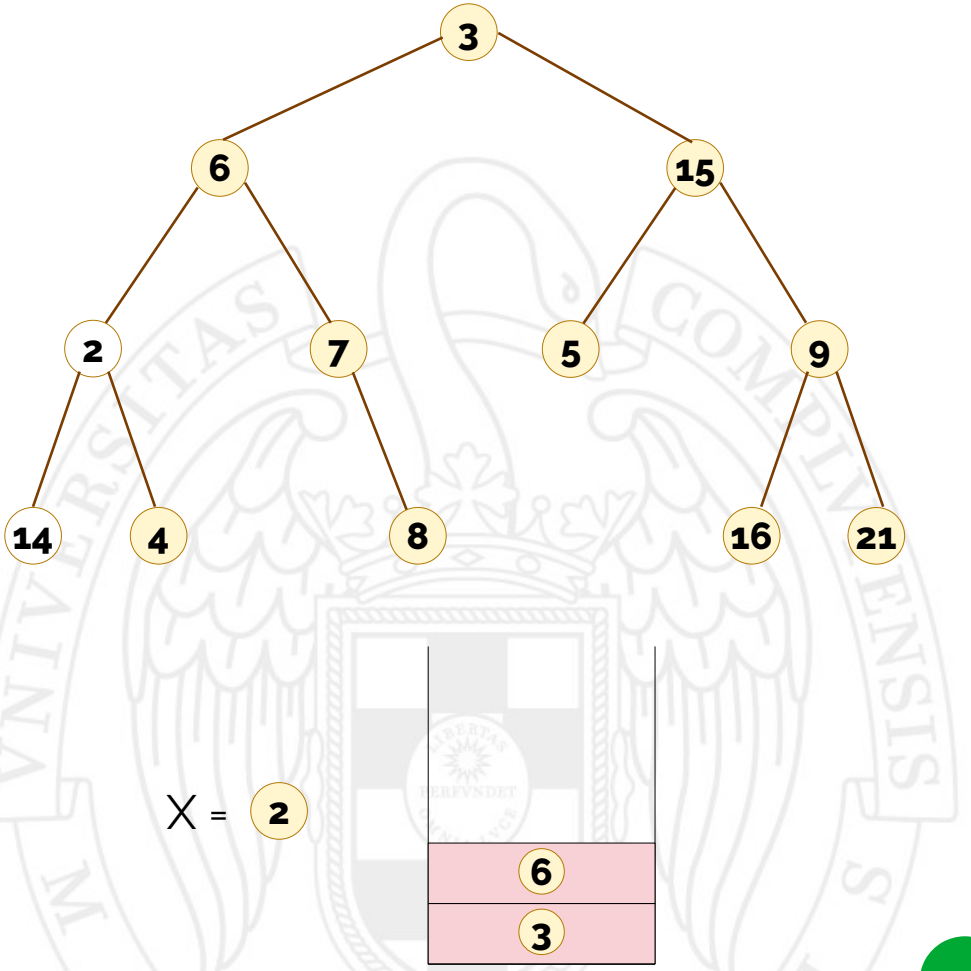


Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

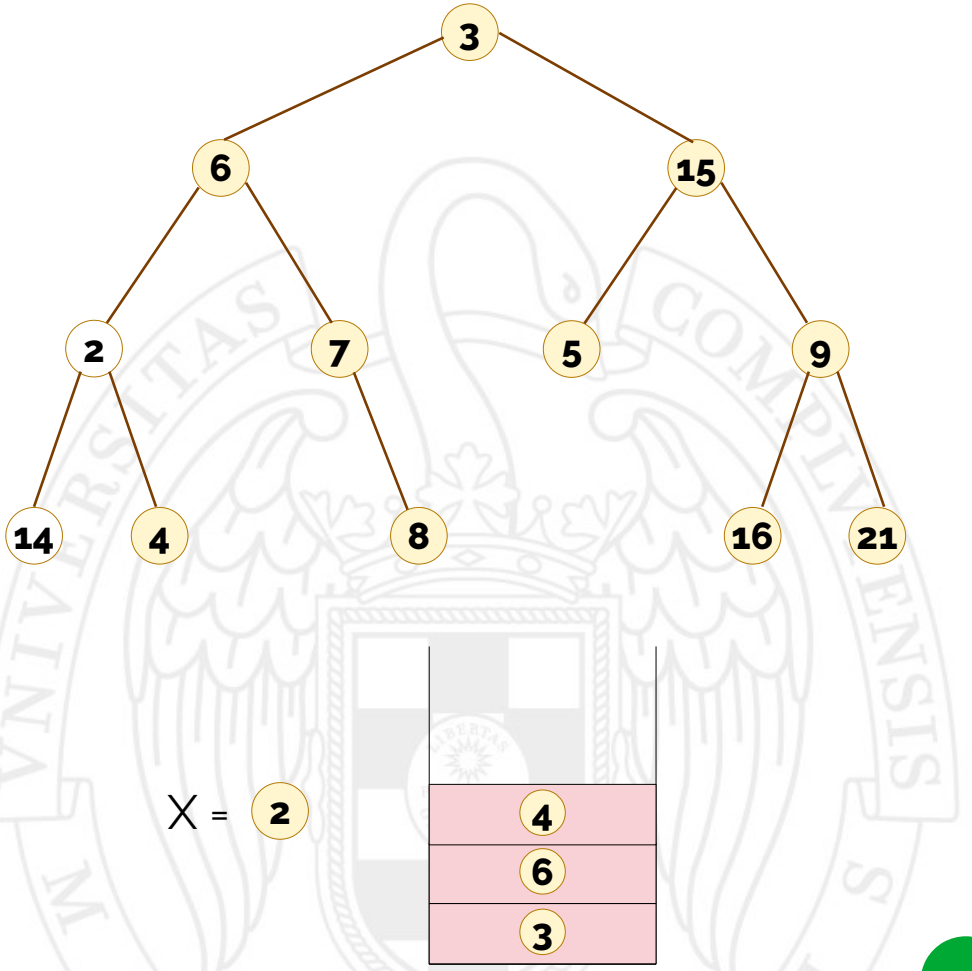


Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

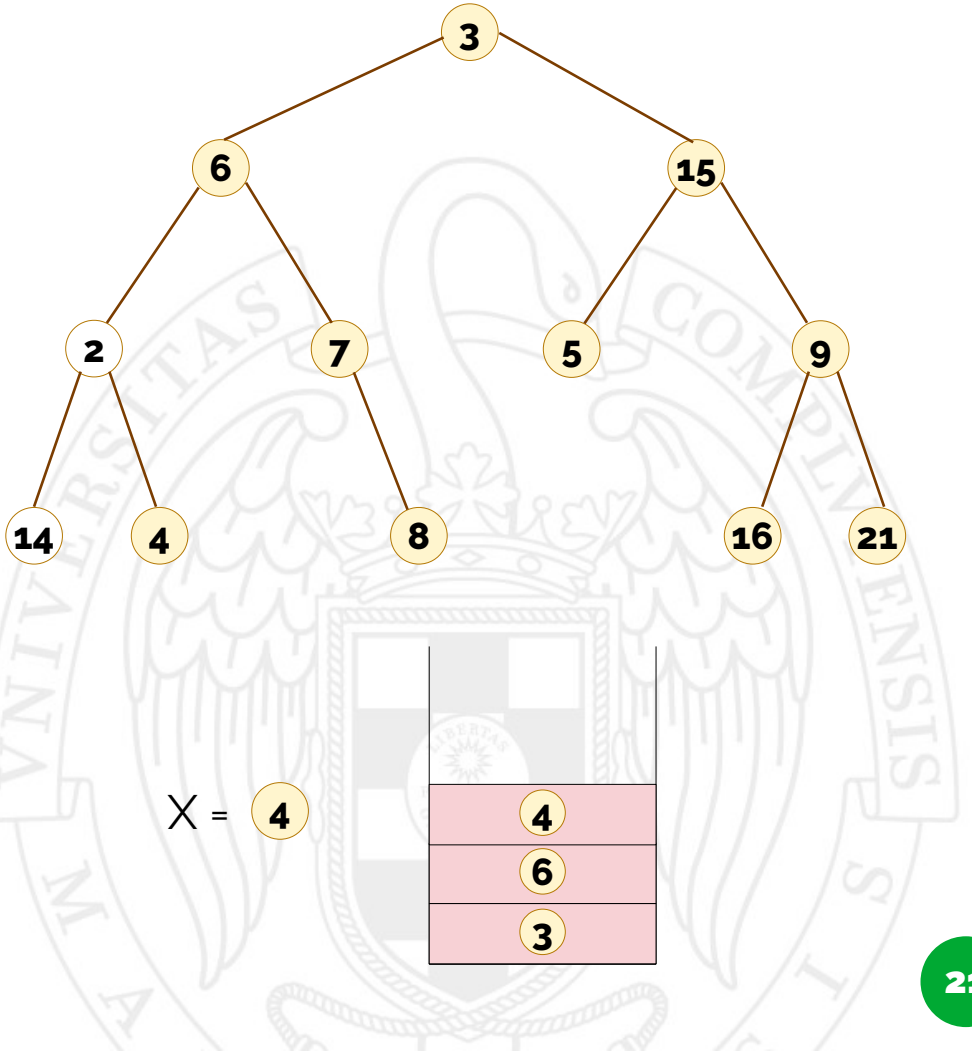


Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

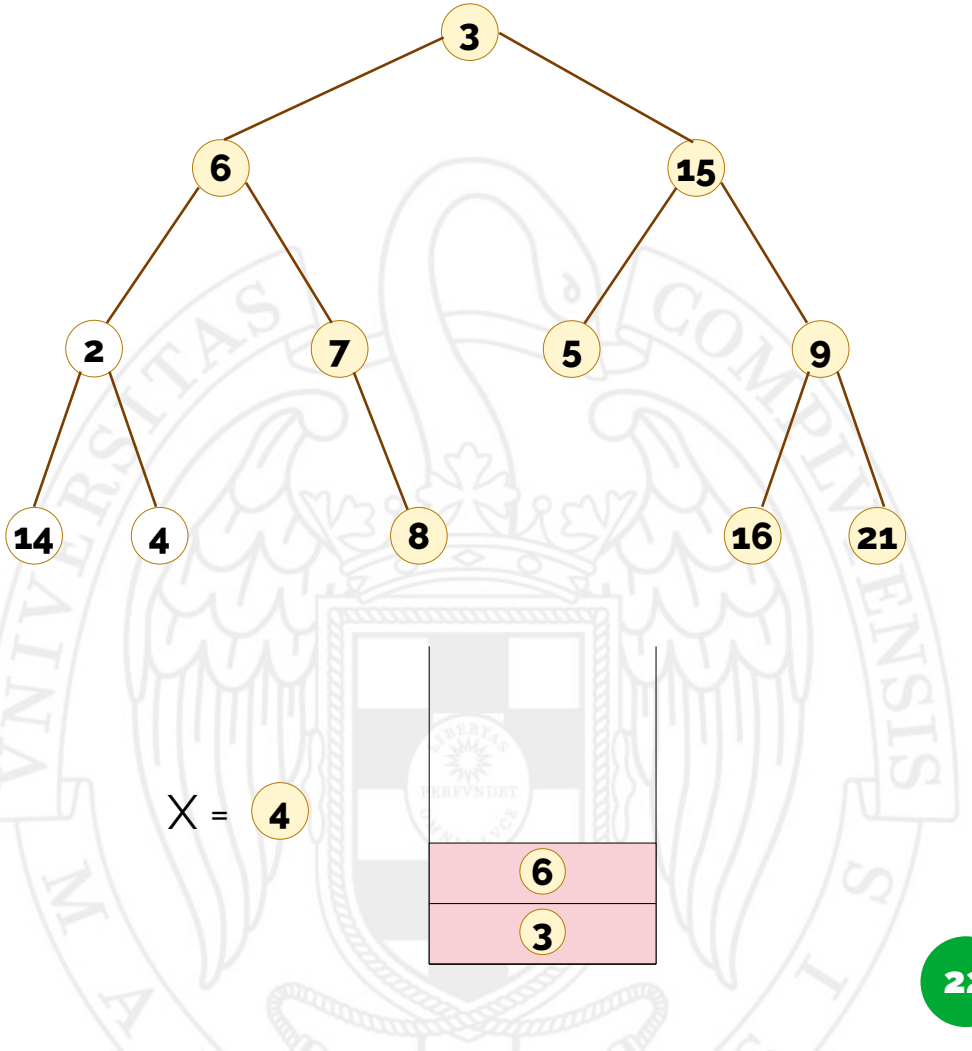


Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

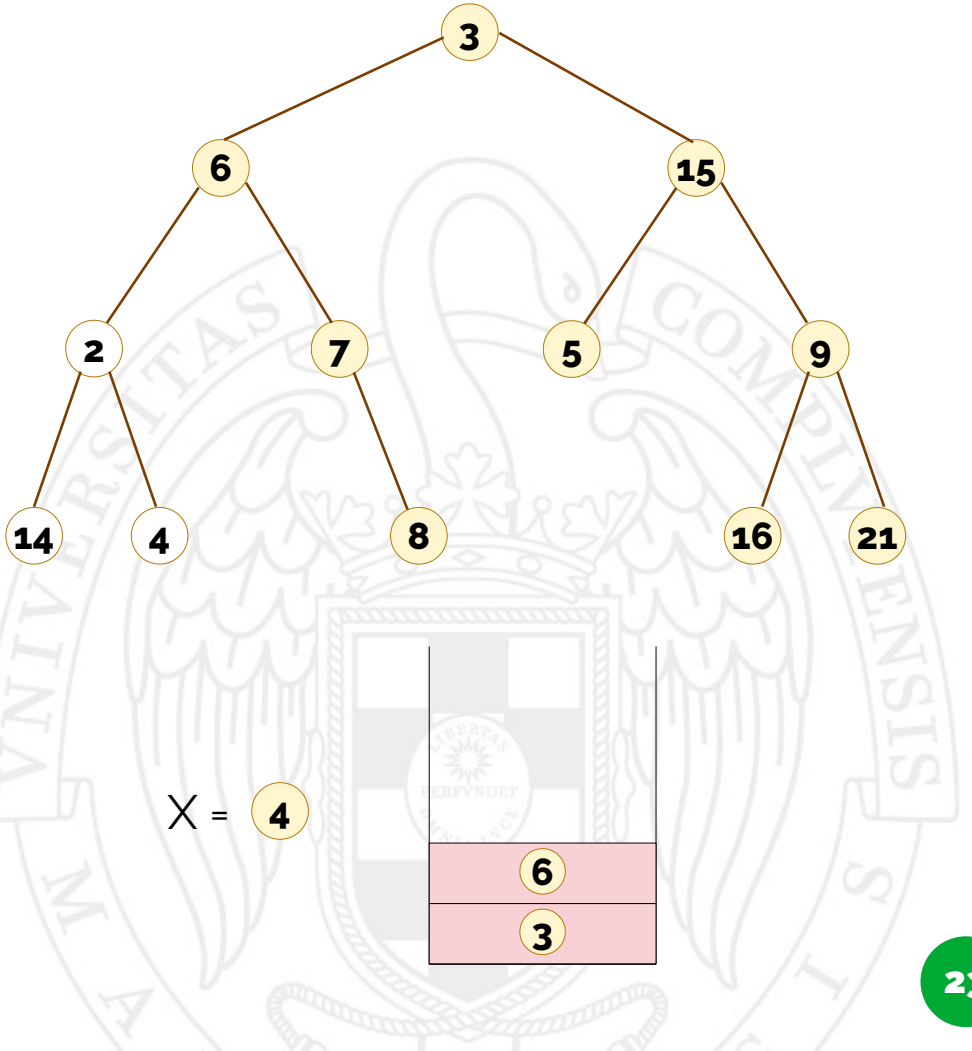


Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

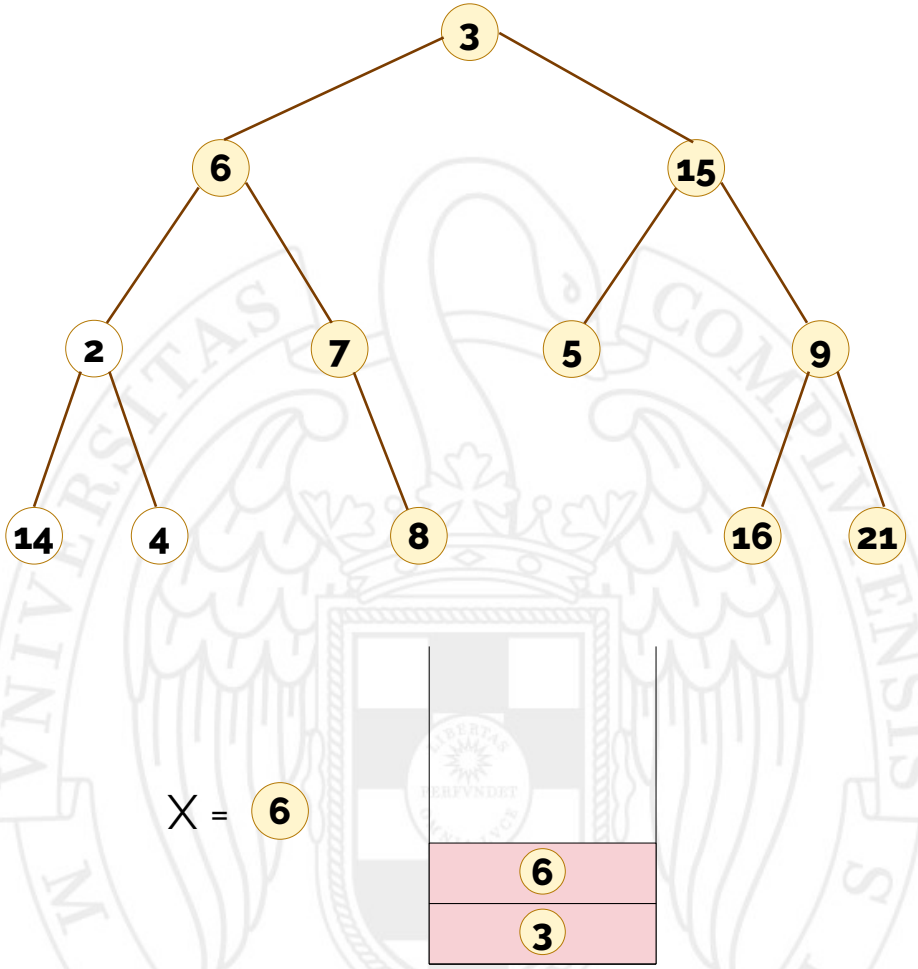


Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

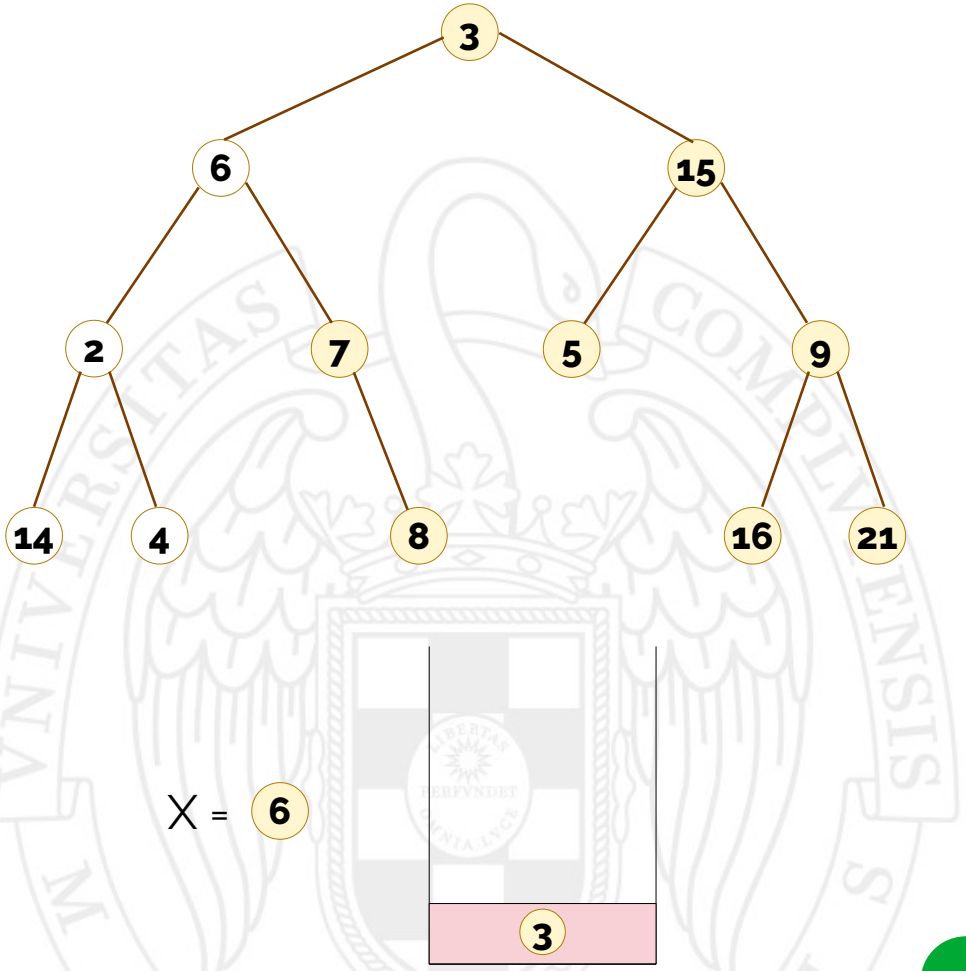


Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

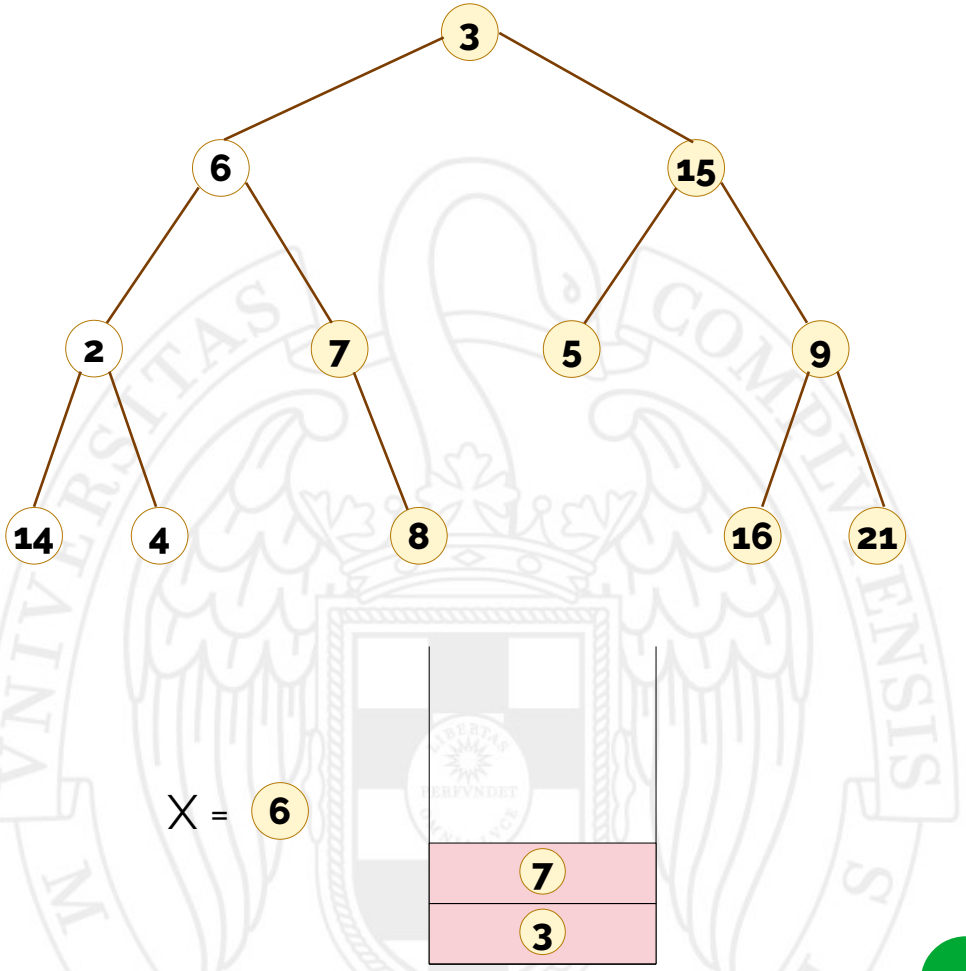


Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

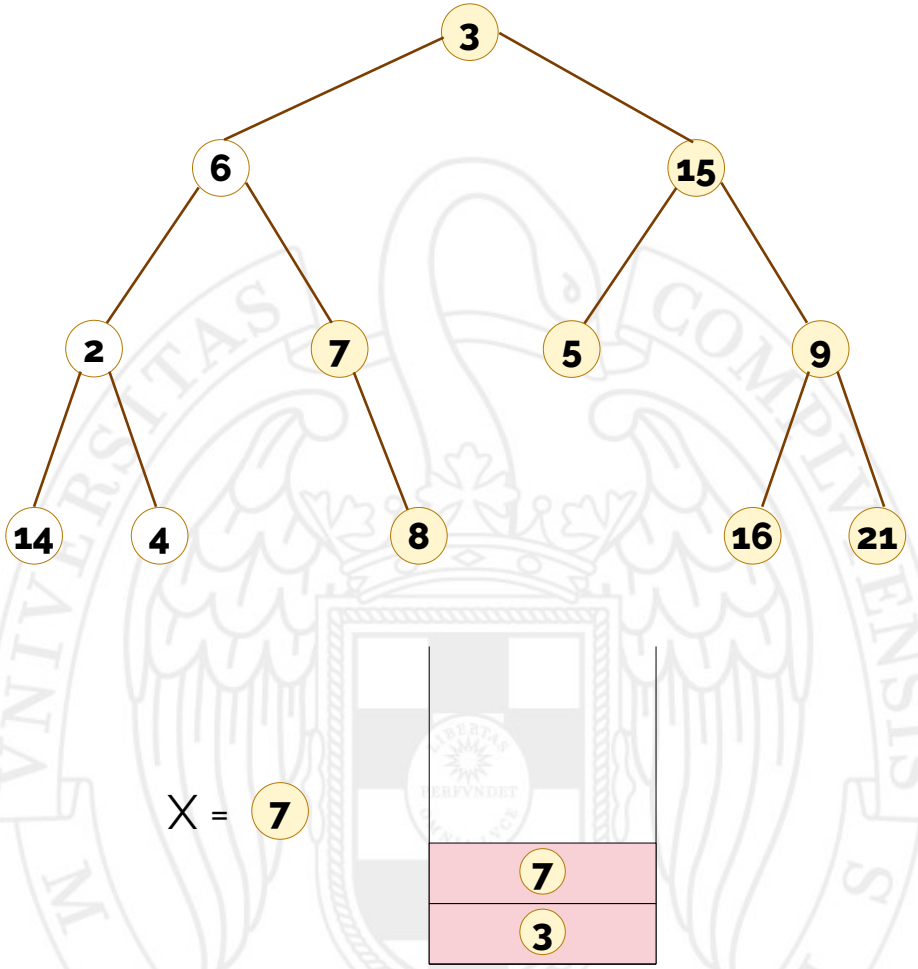


Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

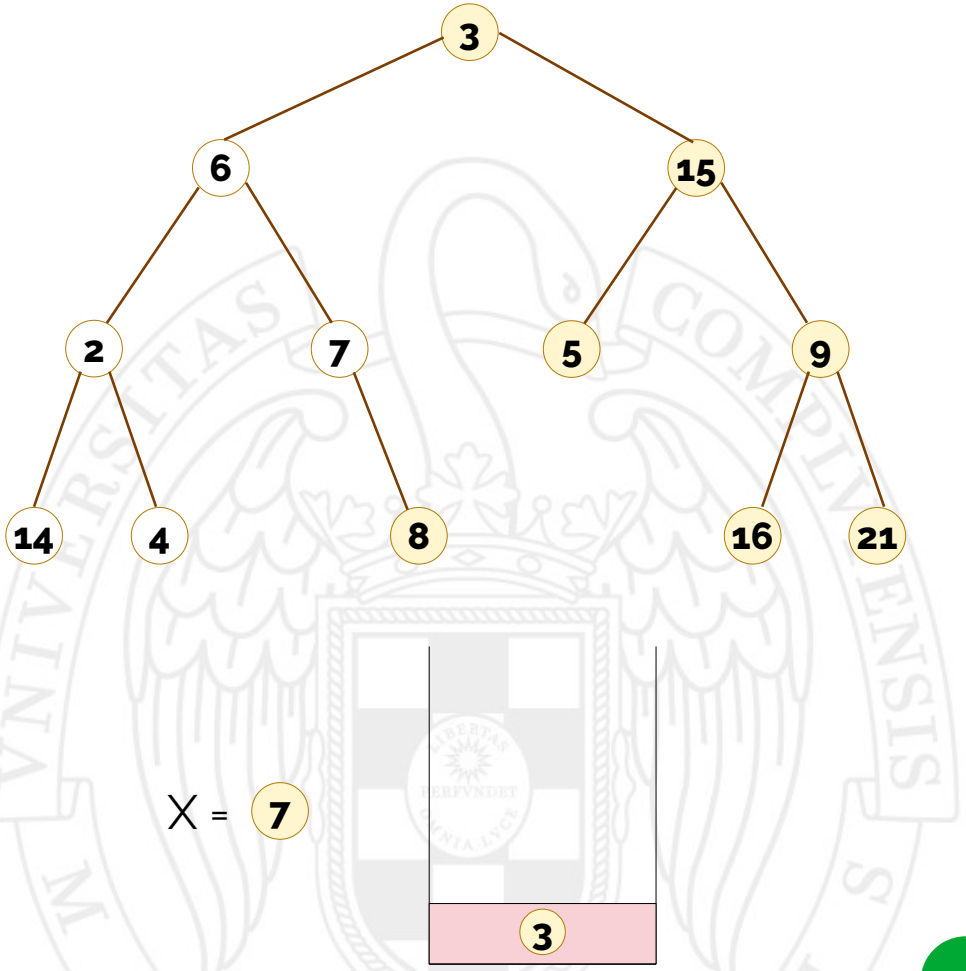


Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

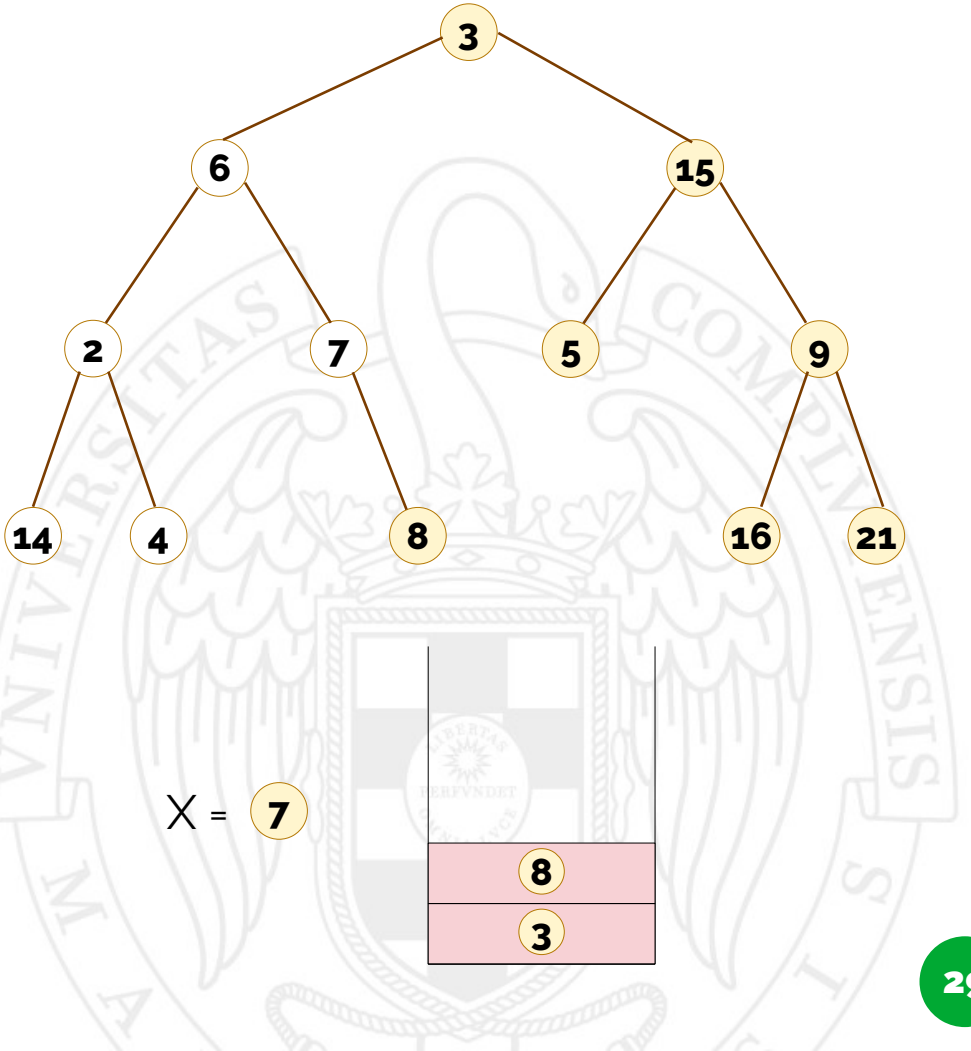


Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

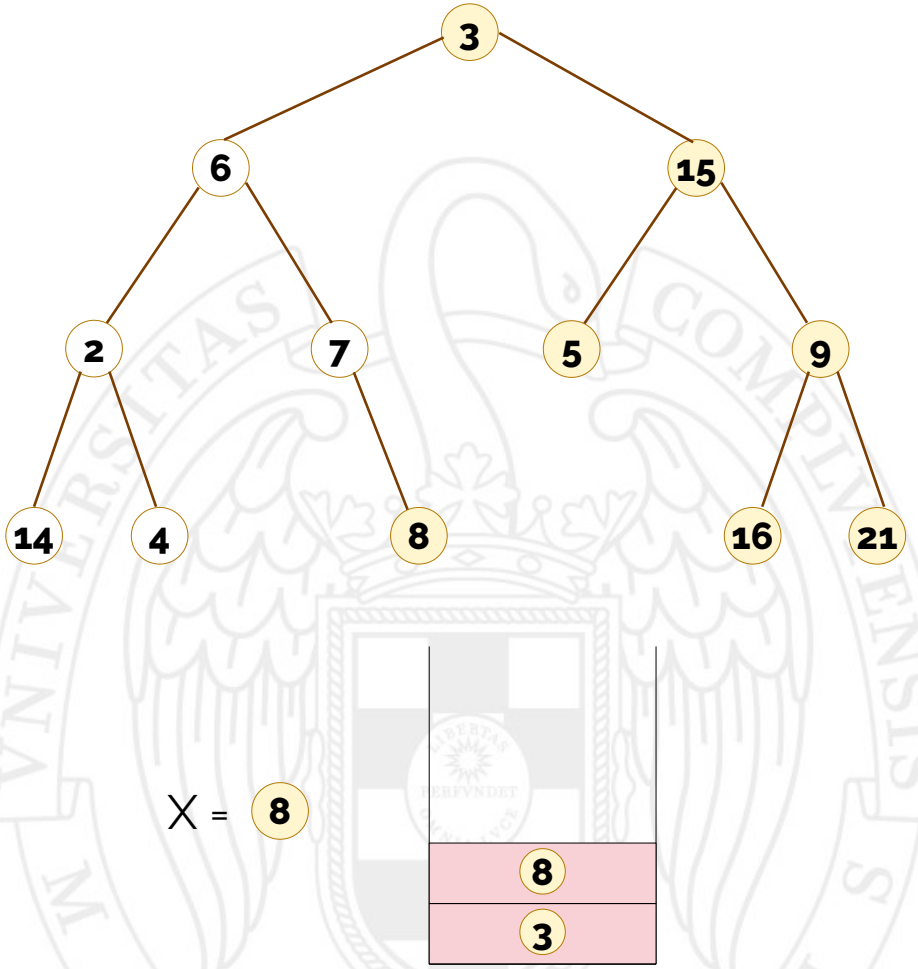


Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

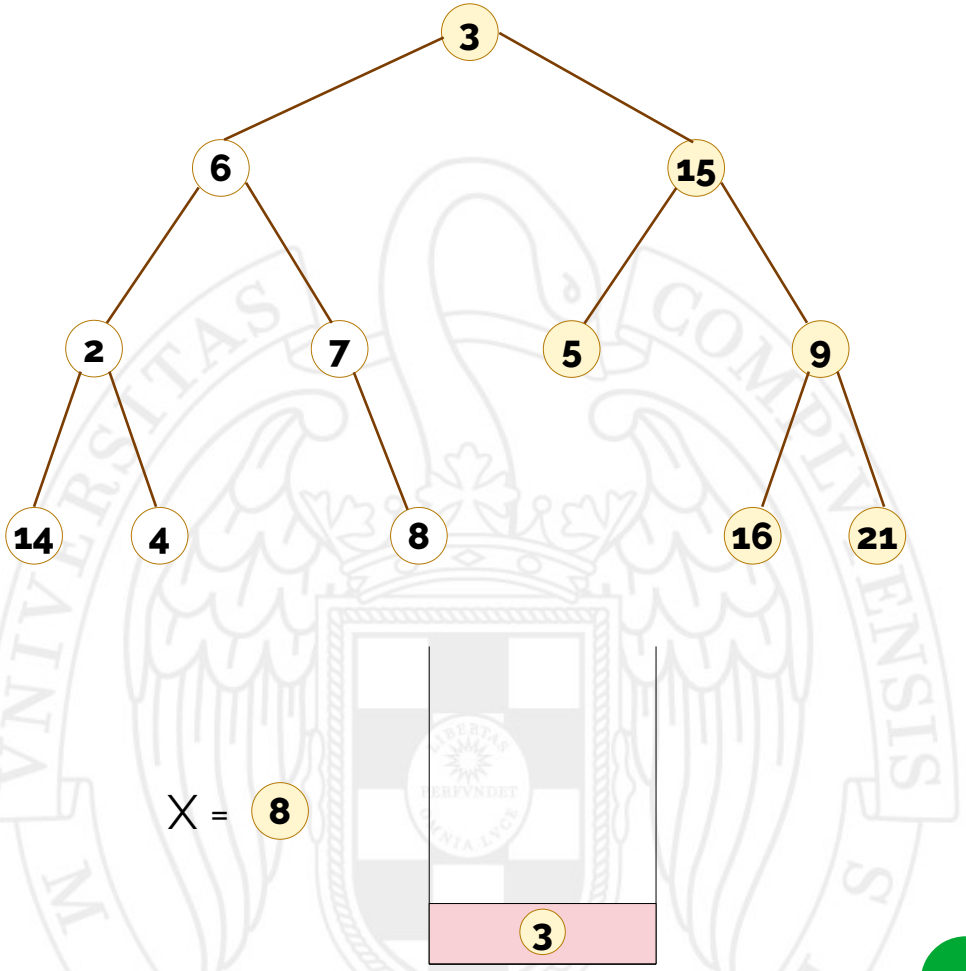


Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

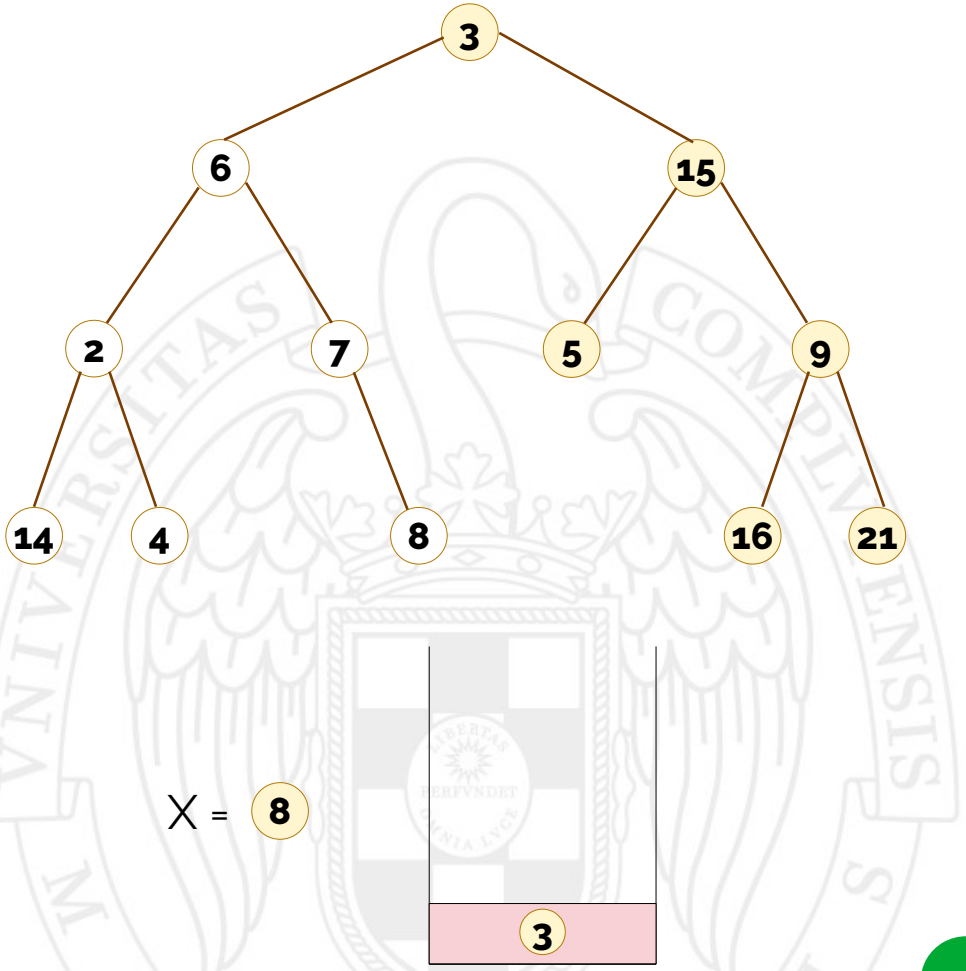


Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

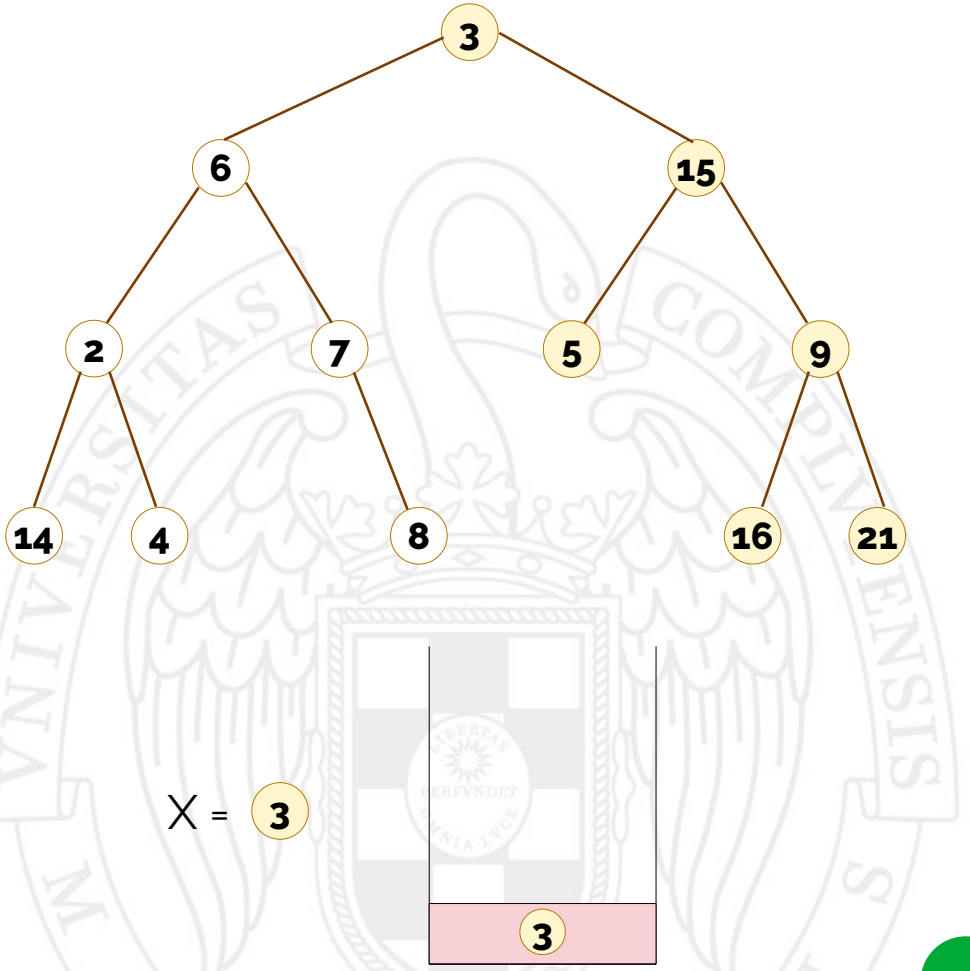


Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

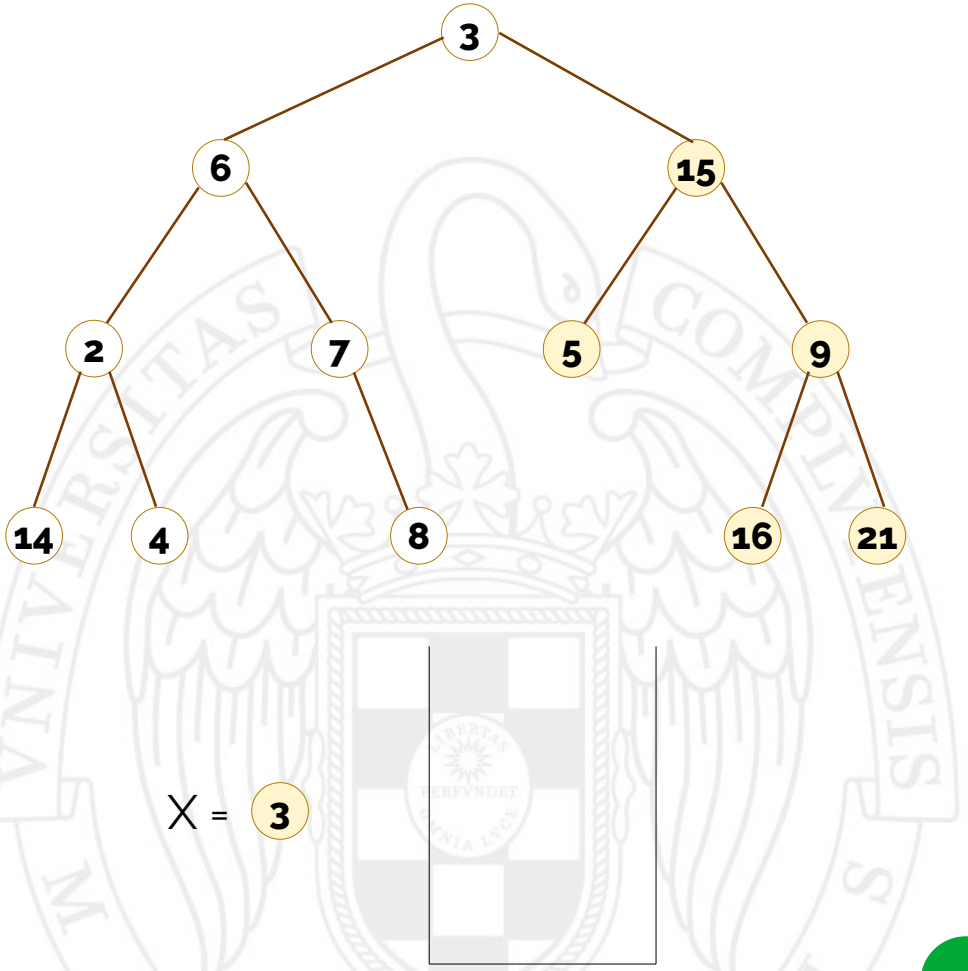


Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

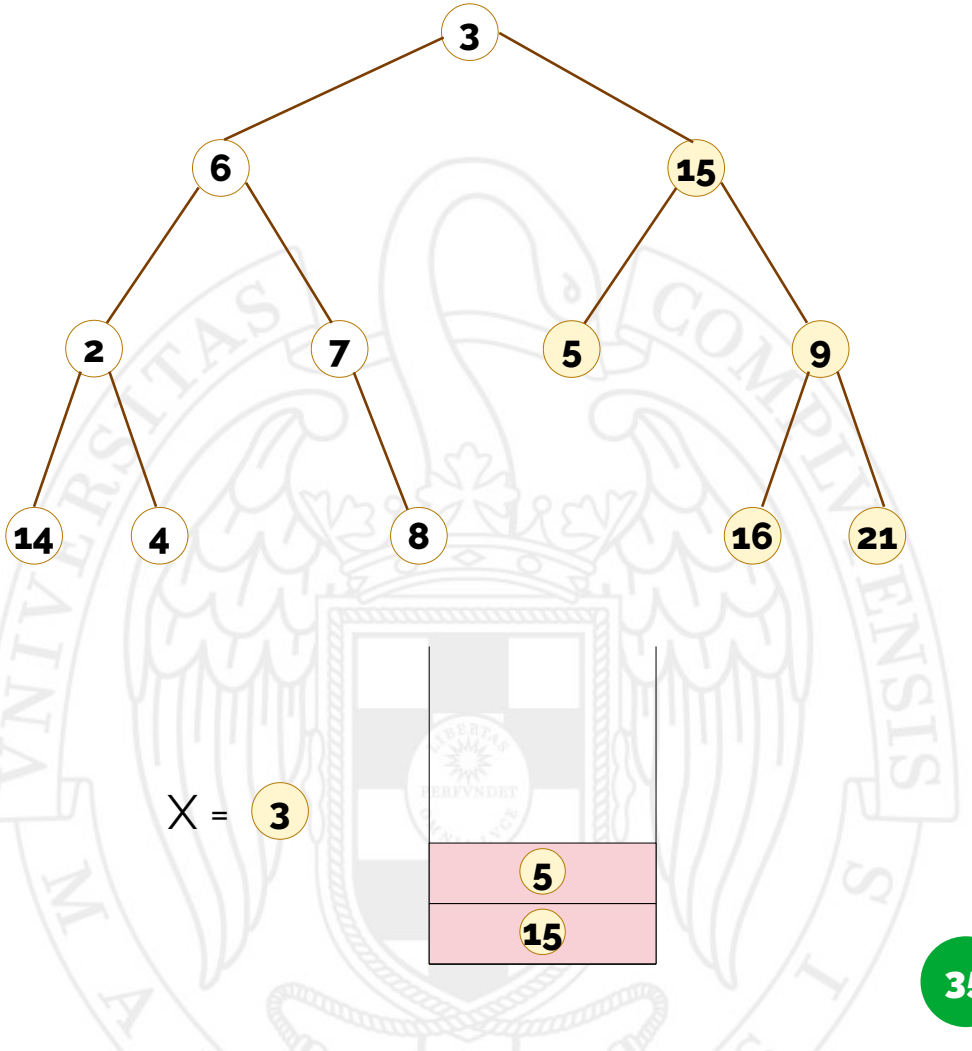


Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

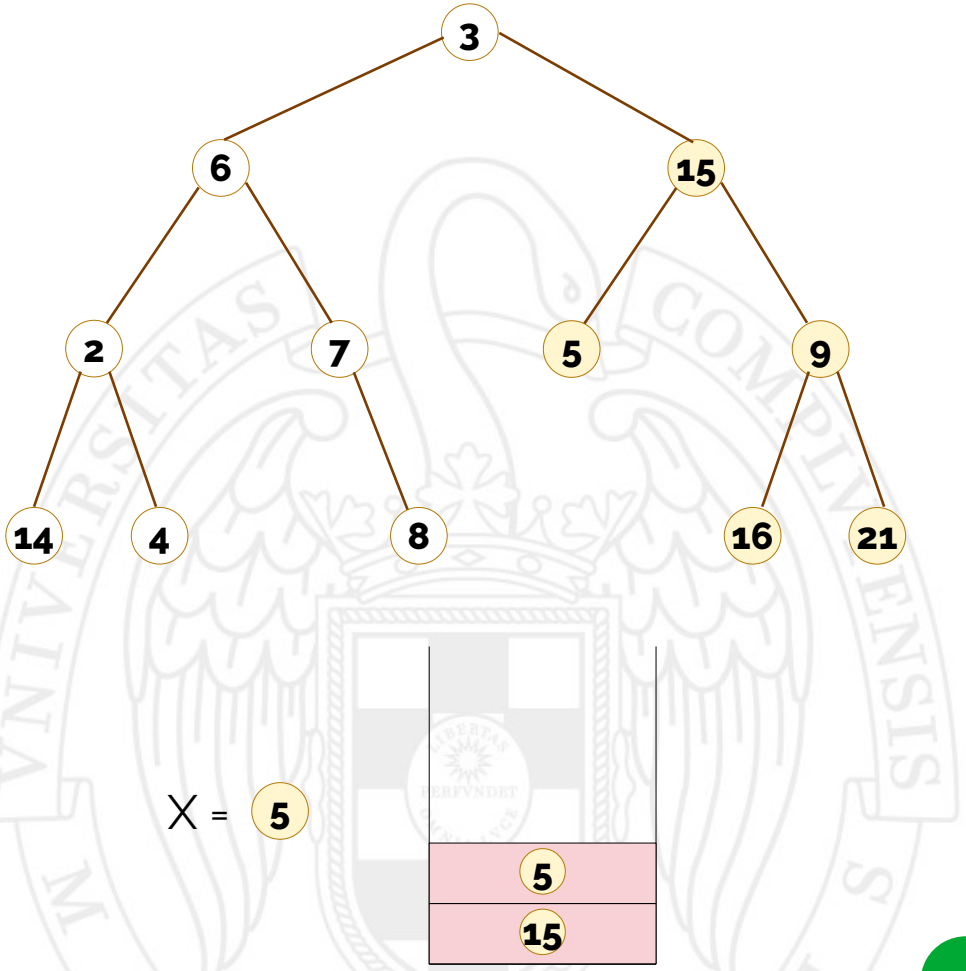


Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

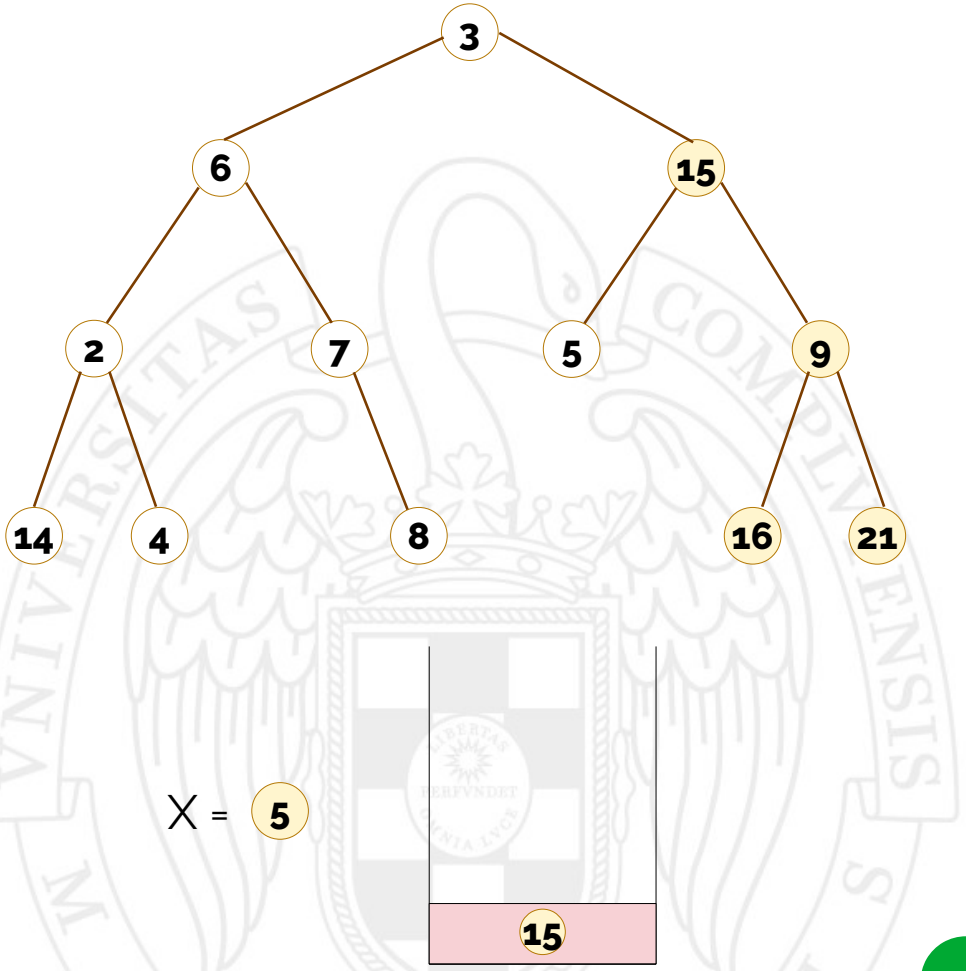


Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

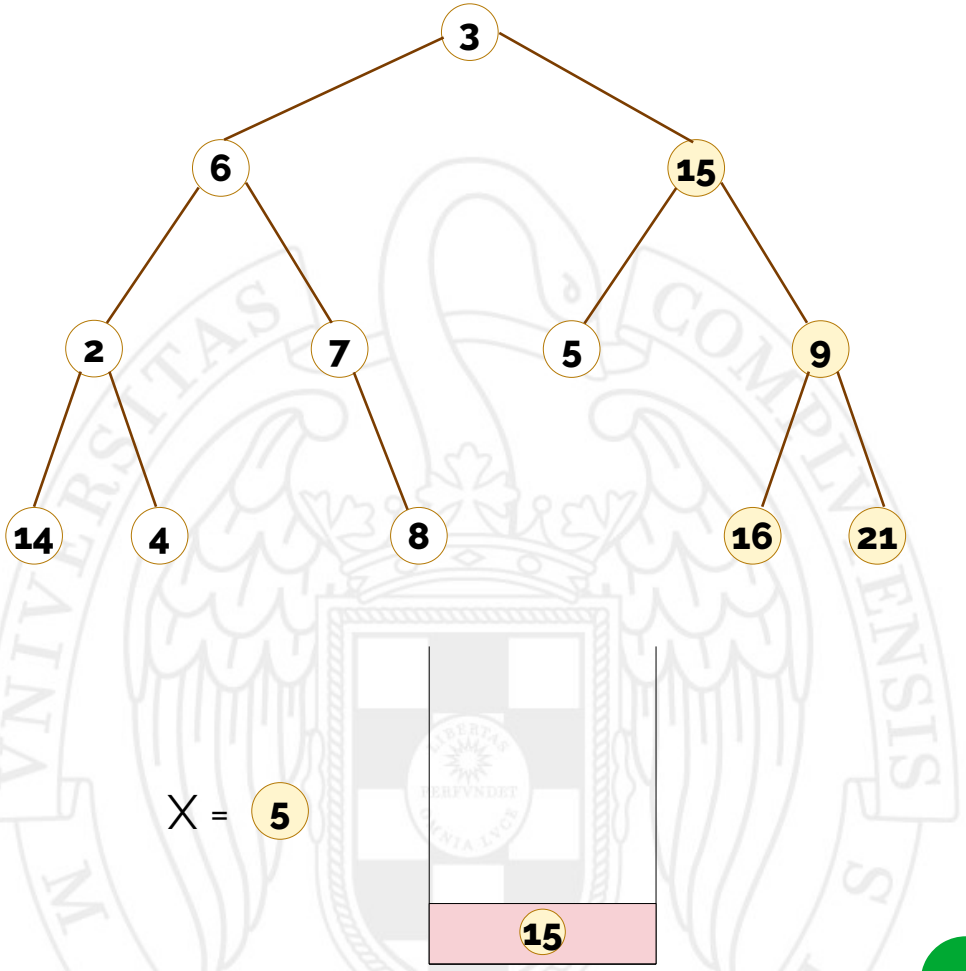


Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

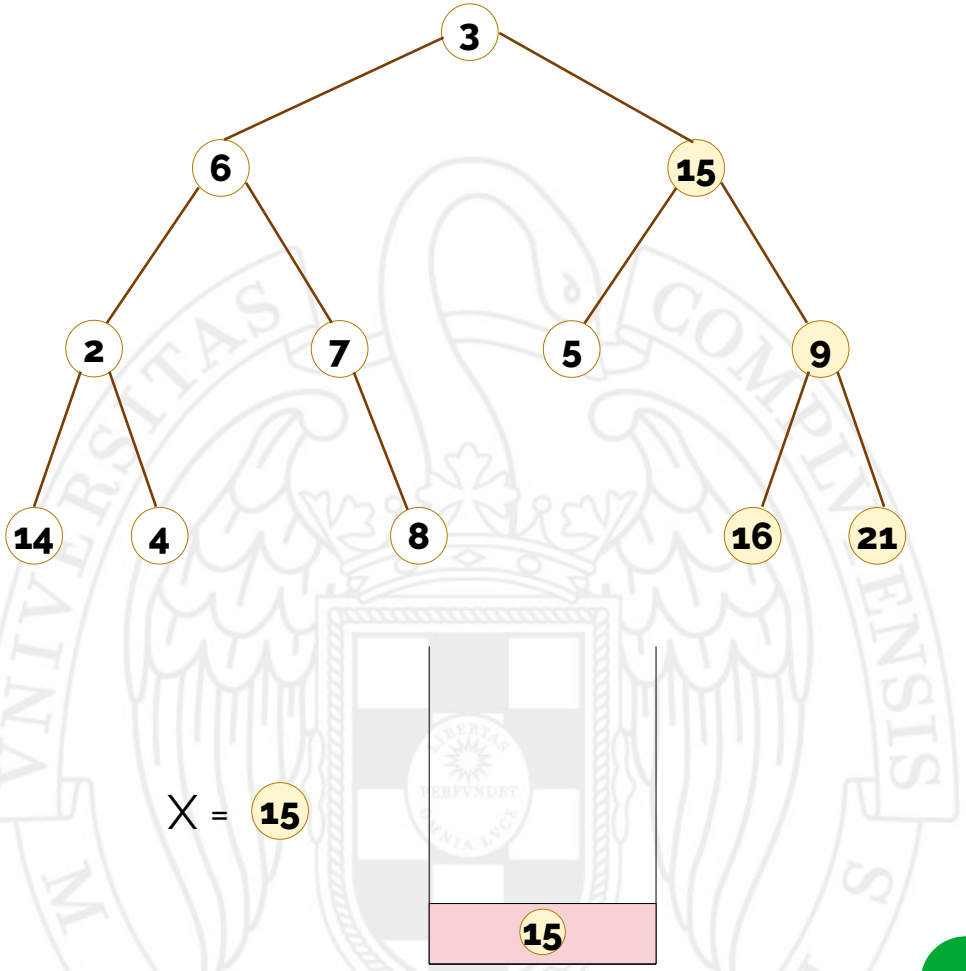


Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

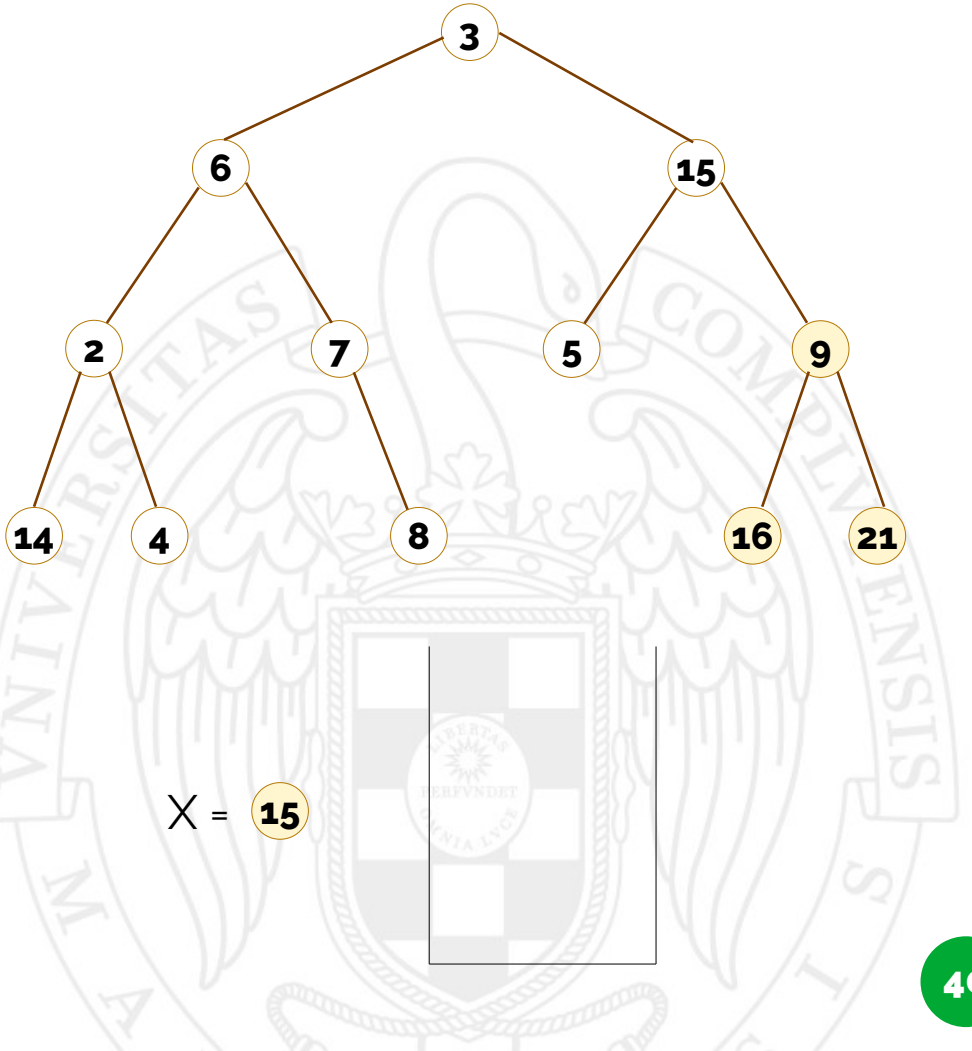


Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

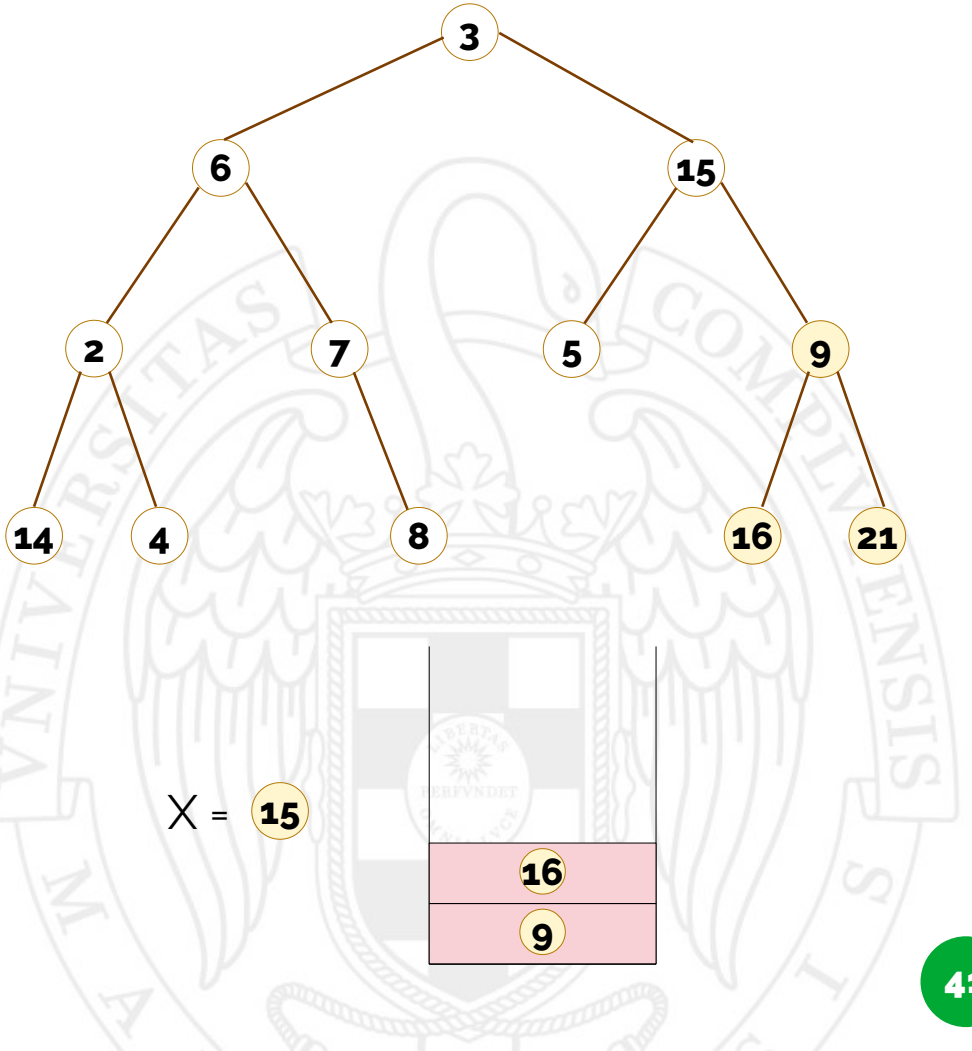


Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

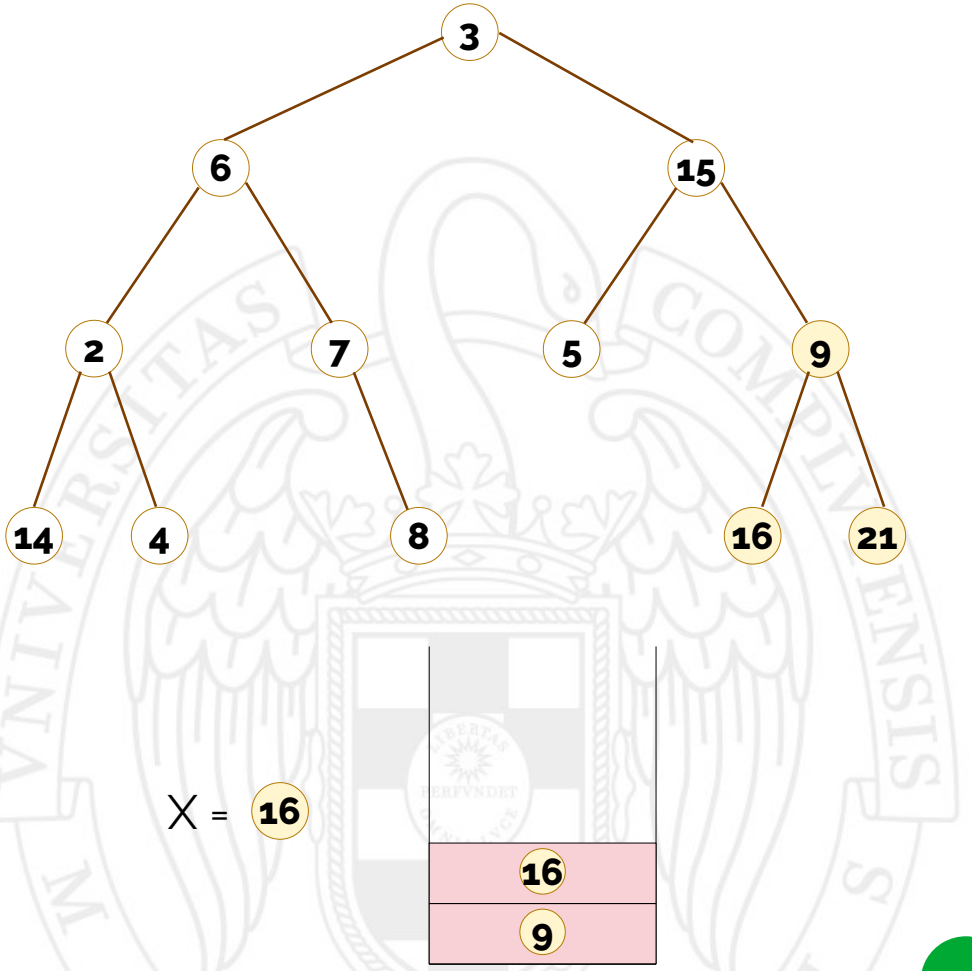


Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

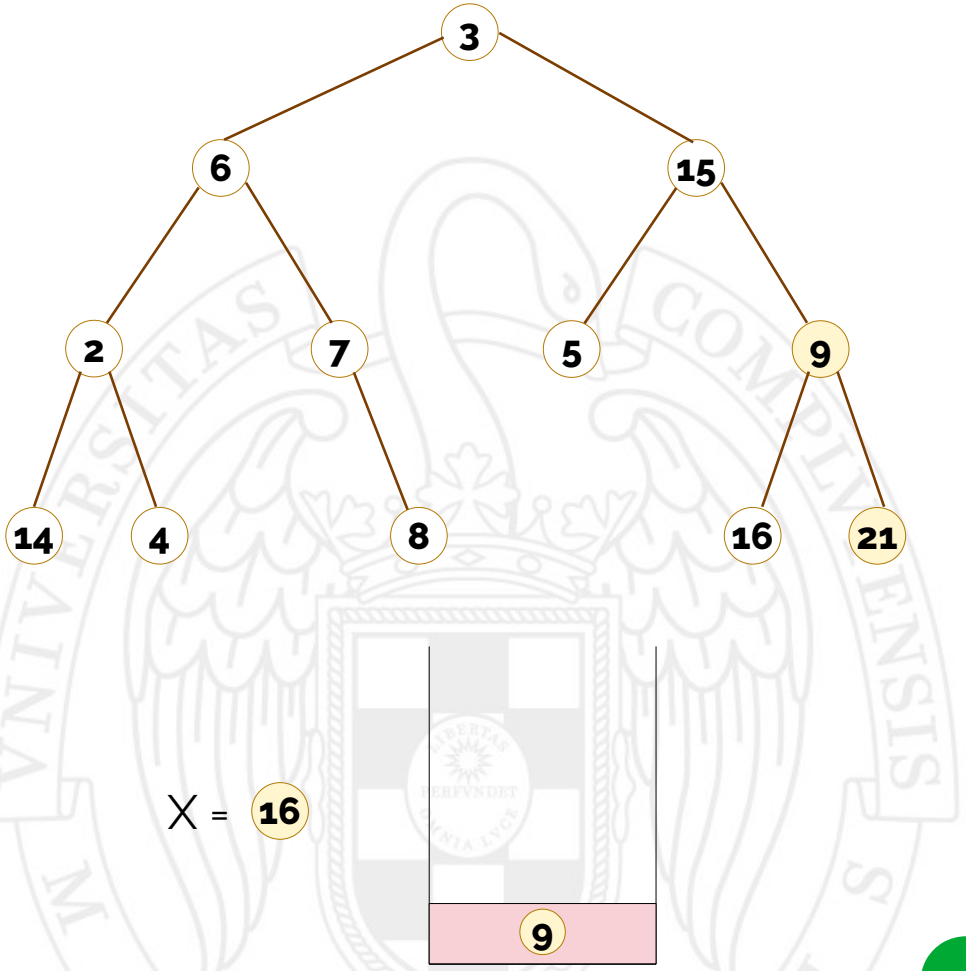


Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

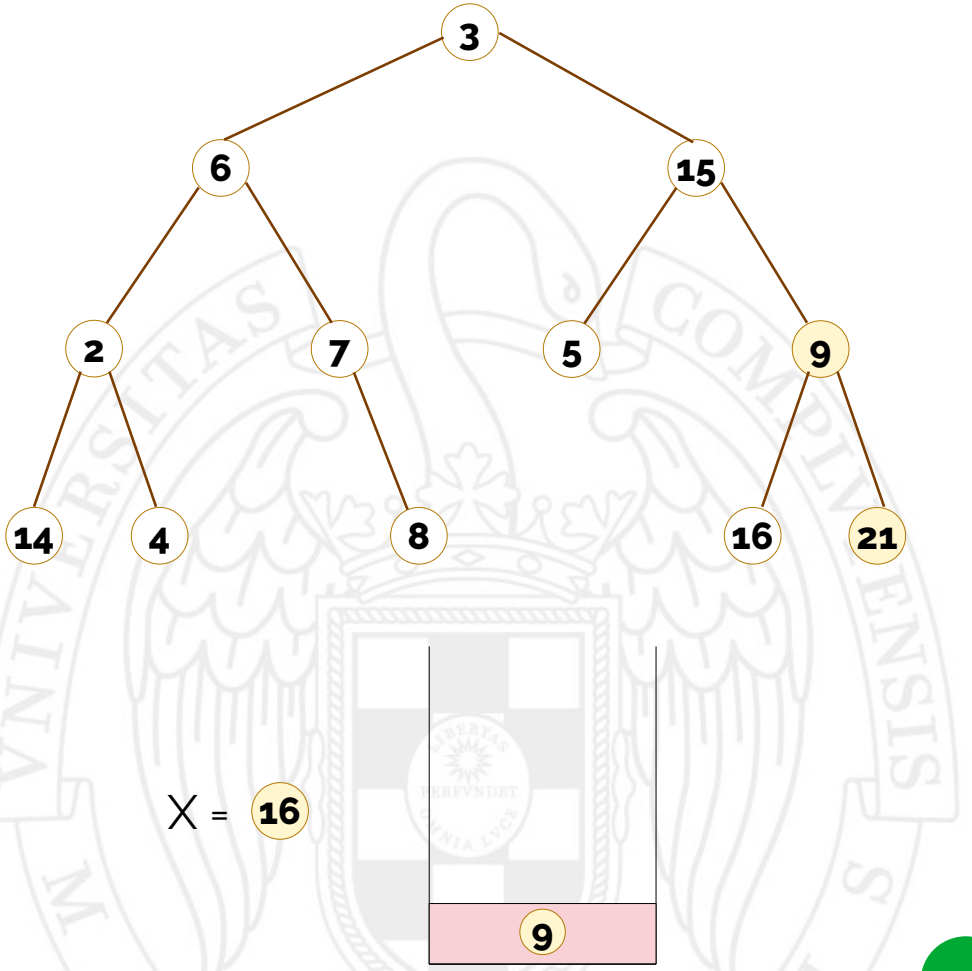


Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

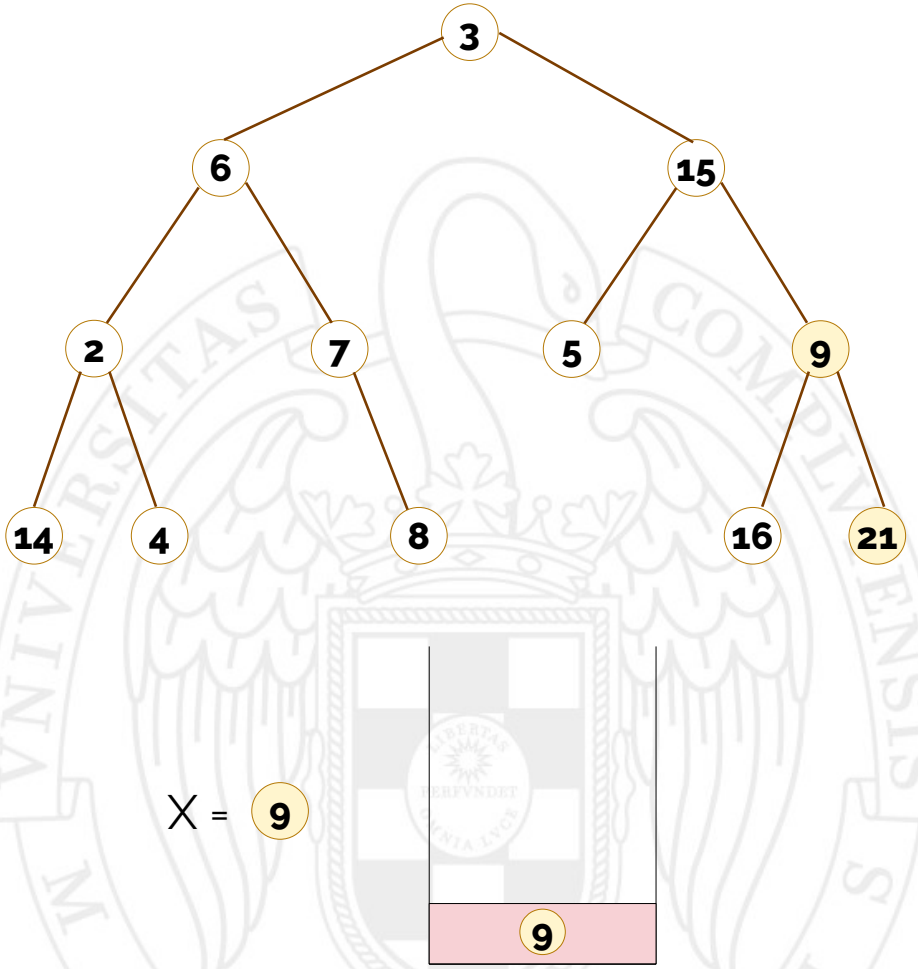


Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

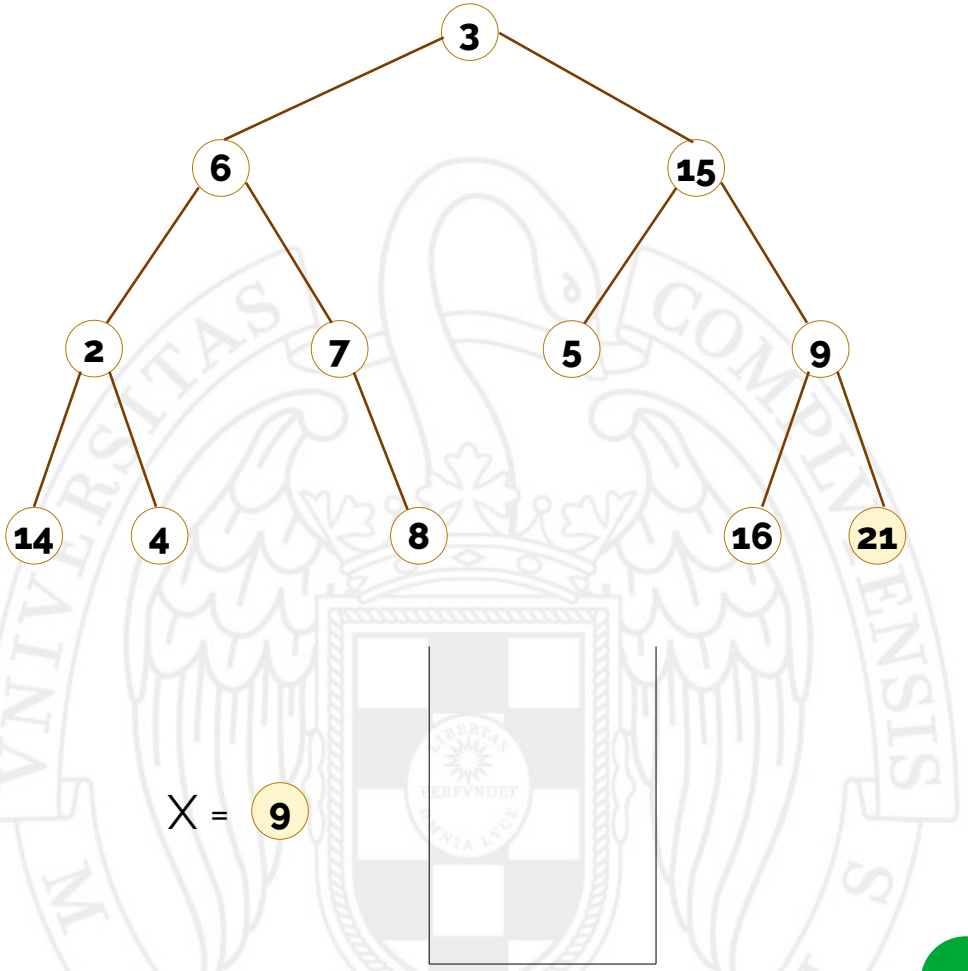


Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

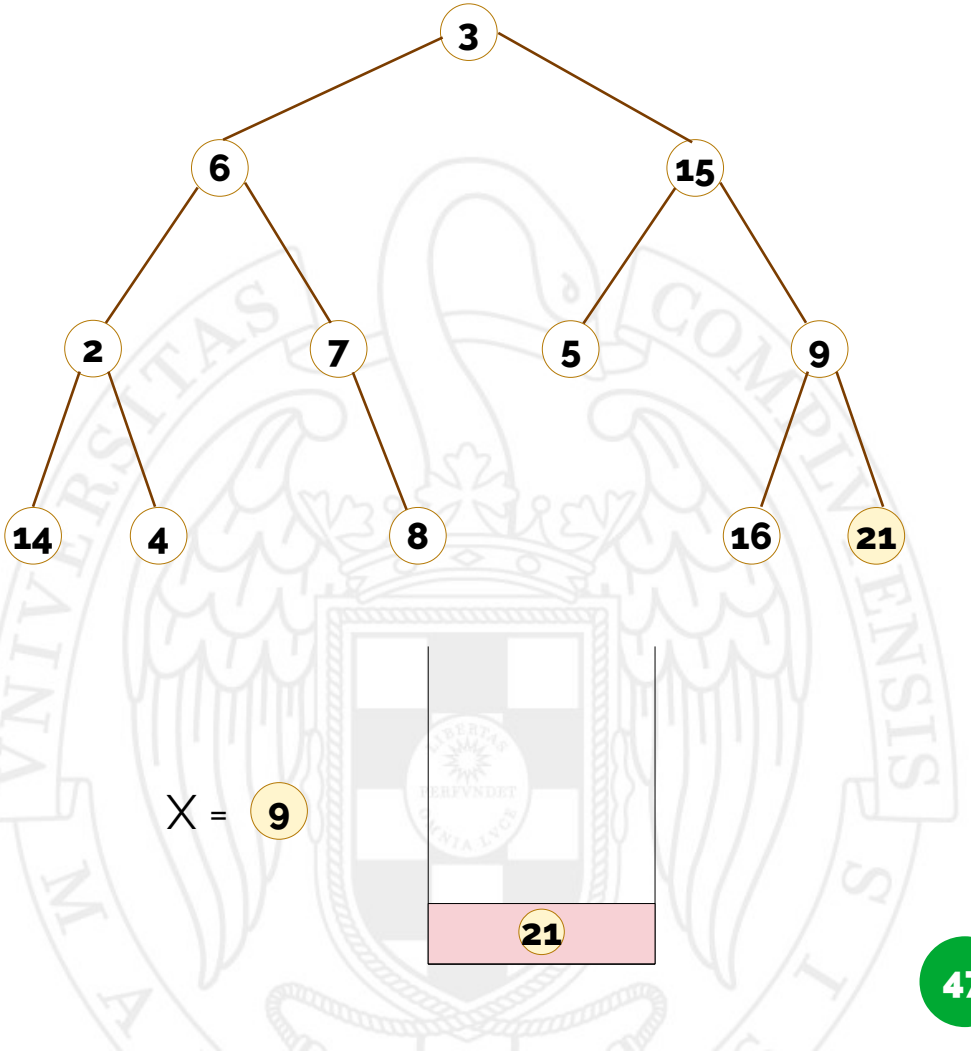


Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

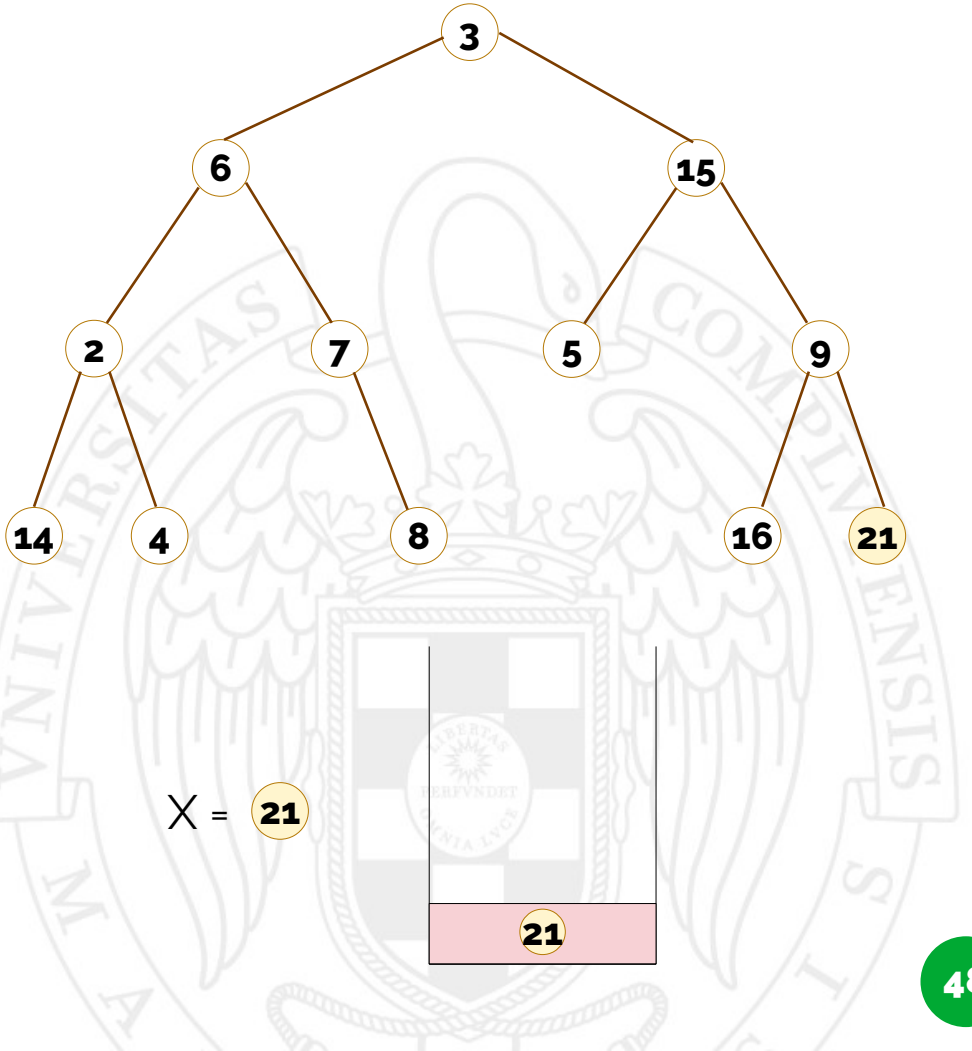


Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

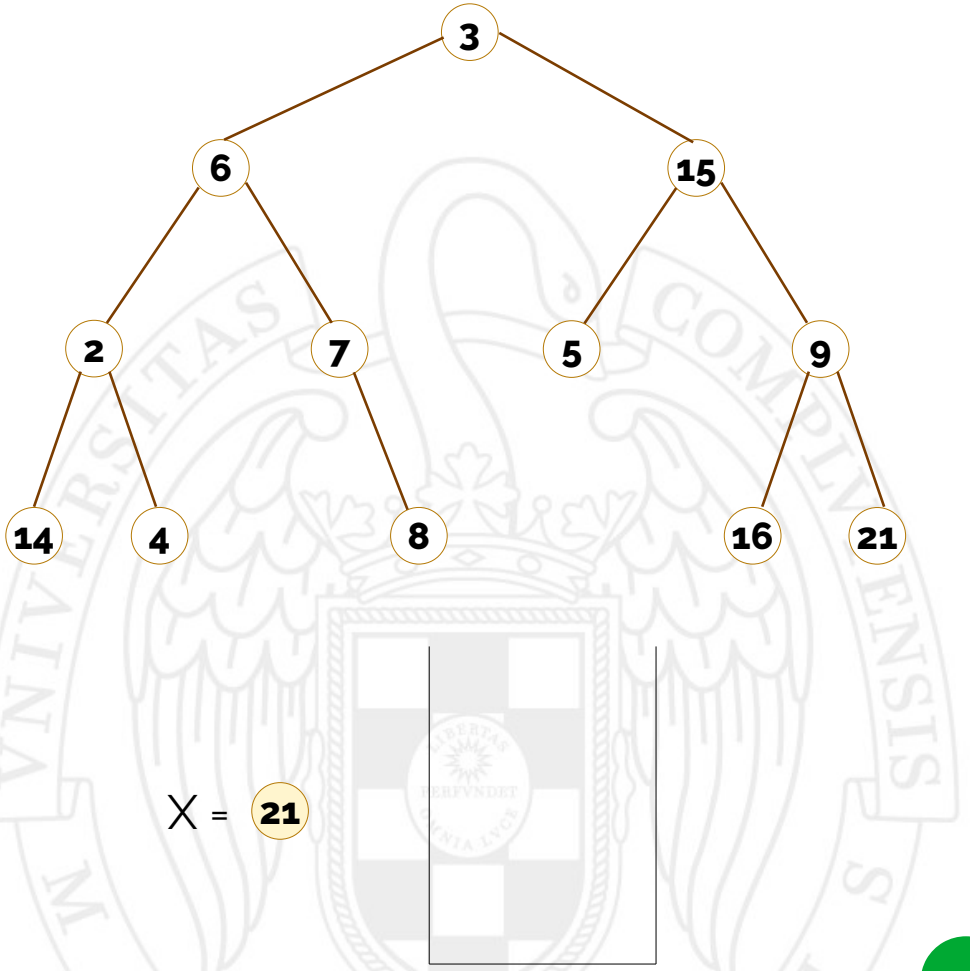


Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

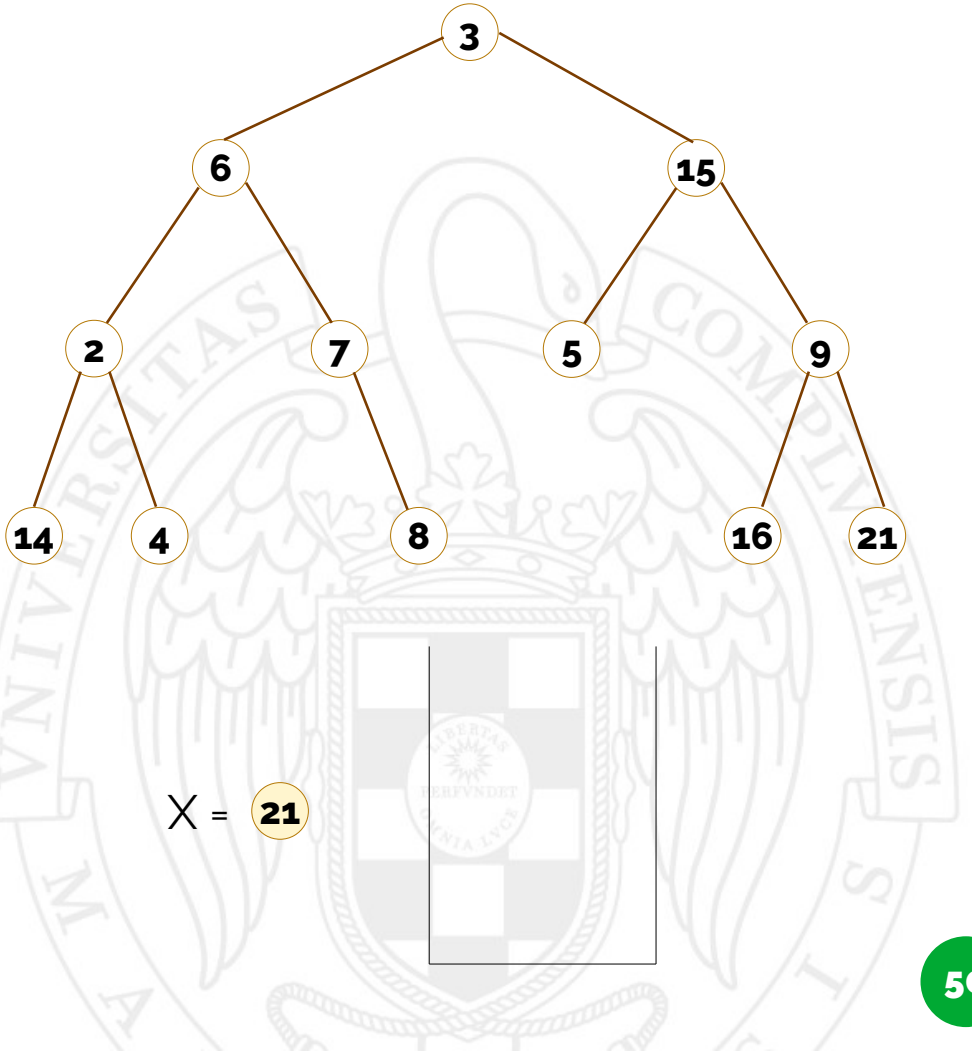


Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

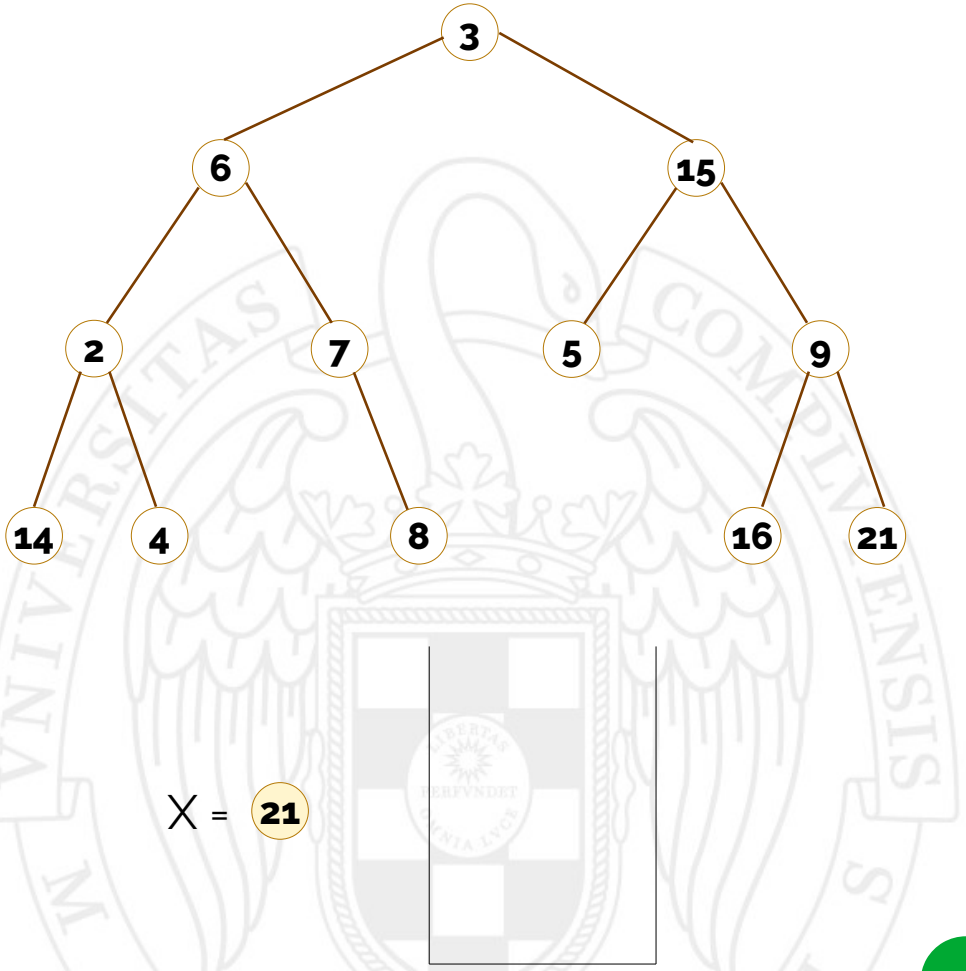


Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir mientras la pila no esté vacía:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.



Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir mientras la pila no esté vacía:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

```
void descend_and_push(const NodePointer &node,
                      std::stack<NodePointer> &st) {
  NodePointer current = node;
  while (current ≠ nullptr) {
    st.push(current);
    current = current→left;
  }
}
```

Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir mientras la pila no esté vacía:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

```
template <typename U>
void inorder(U func) const {

  std::stack<NodePointer> st;
  descend_and_push(root_node, st);

  while (!st.empty()) {
    NodePointer x = st.top();
    st.pop();
    func(x→elem);
    descend_and_push(x→right, st);
  }

}
```

ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS ARBORESCENTES

Recorrido en inorden iterativo (2)

Manuel Montenegro Montes
Departamento de Sistemas Informáticos y Computación
Facultad de Informática – Universidad Complutense de Madrid

Objetivo

- Aplicar técnicas de transformación de programas:

```
void inorder(NodePointer &node) {
  if (node ≠ nullptr) {
    inorder(node→left);
    visit(node→elem);
    inorder(node→right);
  }
}
```

```
void inorder(NodePointer &node) {

  std::stack<NodePointer> st;
  descend_and_push(node, st);

  while (!st.empty()) {
    NodePointer x = st.top();
    st.pop();
    visit(x→elem);
    descend_and_push(x→right, st);
  }

}
```

Transformación de funciones recursivas finales

Recordatorio: funciones recursivas

- Esquema general de una función recursiva simple:

```
void f(x) {
  previo();
  if (es_caso_base(x)) {
    caso_base();
  } else {
    pre_recursivo();
    f(x);
    post_recursivo();
  }
}
```

- Una función es **recursiva final** (o recursiva de cola) si finaliza justo después de la llamada recursiva.
- Es decir, si no se realiza ninguna acción en *post_recursivo()*.

Transformación a iterativo

```
void f(x) {
   previo();
   if (es_caso_base(x)) {
      caso_base();
   } else {
      pre_recursivo();
      f(x);
   }
}
```

Transformación a iterativo

```
previo();
```

```
void f(x) {
  previo();
  if (es_caso_base(x)) {
    caso_base();
  } else {
    pre_recursivo();
    f(x);
  }
}
```

Transformación a iterativo

previo();
{ suponemos ¬es_caso_base(x)*}*

```
void f(x) {
  previo();
  if (es_caso_base(x)) {
    caso_base();
  } else {
    pre_recursivo();
    f(x);
  }
}
```

Transformación a iterativo

```
previo();
{ suponemos ¬es_caso_base(x) }
pre_recursivo();
```

```
void f(x) {
  previo();
  if (es_caso_base(x)) {
    caso_base();
  } else {
    pre_recursivo();
    f(x);
  }
}
```

Transformación a iterativo

```
previo();
{ suponemos ¬es_caso_base(x) }
pre_recursivo();
```

```
void f(x) {
  previo();
  if (es_caso_base(x)) {
    caso_base();
  } else {
    pre_recursivo();
    f(x);
  }
}
```

Transformación a iterativo

```
previo();
{suponemos ¬es_caso_base(x)}
pre_recursivo();
previo();
```

```
void f(x) {
  previo();
  if (es_caso_base(x)) {
    caso_base();
  } else {
    pre_recursivo();
    f(x);
  }
}
```

Transformación a iterativo

```
previo();
{ suponemos ¬es_caso_base(x) }
pre_recursivo();
previo();
{ suponemos ¬es_caso_base(x) }
```

```
void f(x) {
  previo();
  if (es_caso_base(x)) {
    caso_base();
  } else {
    pre_recursivo();
    f(x);
  }
}
```

Transformación a iterativo

```
previo();
{ suponemos ¬es_caso_base(x) }
pre_recursivo();
previo();
{ suponemos ¬es_caso_base(x) }
pre_recursivo();
```

```
void f(x) {
  previo();
  if (es_caso_base(x)) {
    caso_base();
  } else {
    pre_recursivo();
    f(x);
  }
}
```

Transformación a iterativo

```
previo();
{suponemos ¬es_caso_base(x)}
pre_recursivo();
previo();
{suponemos ¬es_caso_base(x)}
pre_recursivo();
```

```
void f(x) {
  previo();
  if (es_caso_base(x)) {
    caso_base();
  } else {
    pre_recursivo();
    f(x);
  }
}
```

Transformación a iterativo

```
previo();
{suponemos ¬es_caso_base(x)}
pre_recursivo();
previo();
{suponemos ¬es_caso_base(x)}
pre_recursivo();
previo();
```

```
void f(x) {
  previo();
  if (es_caso_base(x)) {
    caso_base();
  } else {
    pre_recursivo();
    f(x);
  }
}
```

Transformación a iterativo

```
previo();
{suponemos ¬es_caso_base(x)}
pre_recursivo();
previo();
{suponemos ¬es_caso_base(x)}
pre_recursivo();
previo();
{suponemos ¬es_caso_base(x)}
```

```
void f(x) {
  previo();
  if (es_caso_base(x)) {
    caso_base();
  } else {
    pre_recursivo();
    f(x);
  }
}
```

Transformación a iterativo

```
previo();
{suponemos ¬es_caso_base(x)}
pre_recursivo();
previo();
{suponemos ¬es_caso_base(x)}
pre_recursivo();
previo();
{suponemos ¬es_caso_base(x)}
pre_recursivo();
```

```
void f(x) {
  previo();
  if (es_caso_base(x)) {
    caso_base();
  } else {
    pre_recursivo();
    f(x);
  }
}
```

Transformación a iterativo

```
previo();
{suponemos ¬es_caso_base(x)}
pre_recursivo();
previo();
{suponemos ¬es_caso_base(x)}
pre_recursivo();
previo();
{suponemos ¬es_caso_base(x)}
pre_recursivo();
```

```
void f(x) {
  previo();
  if (es_caso_base(x)) {
    caso_base();
  } else {
    pre_recursivo();
    f(x);
  }
}
```

Transformación a iterativo

```
previo();
{suponemos ¬es_caso_base(x)}
pre_recursivo();
previo();
{suponemos ¬es_caso_base(x)}
pre_recursivo();
previo();
{suponemos ¬es_caso_base(x)}
pre_recursivo();
previo();
```

```
void f(x) {
  previo();
  if (es_caso_base(x)) {
    caso_base();
  } else {
    pre_recursivo();
    f(x);
  }
}
```

Transformación a iterativo

```
previo();
{suponemos ¬es_caso_base(x)}
pre_recursivo();
previo();
{suponemos ¬es_caso_base(x)}
pre_recursivo();
previo();
{suponemos ¬es_caso_base(x)}
pre_recursivo();
previo();
{suponemos es_caso_base(x)}
```

```
void f(x) {
  previo();
  if (es_caso_base(x)) {
    caso_base();
  } else {
    pre_recursivo();
    f(x);
  }
}
```

Transformación a iterativo

```
previo();
{suponemos ¬es_caso_base(x)}
pre_recursivo();
previo();
{suponemos ¬es_caso_base(x)}
pre_recursivo();
previo();
{suponemos ¬es_caso_base(x)}
pre_recursivo();
previo();
{suponemos es_caso_base(x)}
caso_base();
```

```
void f(x) {
  previo();
  if (es_caso_base(x)) {
    caso_base();
  } else {
    pre_recursivo();
    f(x);
  }
}
```

Transformación a iterativo

```
previo();
{suponemos ¬es_caso_base(x)}
pre_recursivo();
previo();
{suponemos ¬es_caso_base(x)}
pre_recursivo();
previo();
{suponemos ¬es_caso_base(x)}
pre_recursivo();
previo();
{suponemos es_caso_base(x)}
caso_base();
```

```
void f(x) {
  previo();
  if (es_caso_base(x)) {
    caso_base();
  } else {
    pre_recursivo();
    f(x);
  }
}
```

Transformación a iterativo

```
previo();
{suponemos ¬es_caso_base(x)}
pre_recursivo();
previo();
{suponemos ¬es_caso_base(x)}
pre_recursivo();
previo();
{suponemos ¬es_caso_base(x)}
pre_recursivo();
previo();
{suponemos es_caso_base(x)}
caso_base();
```

```
void f(x) {
  previo();
  while (!es_caso_base(x)) {
    pre_recursivo();
    previo();
  }
  caso_base();
}
```

Transformación a iterativo

```
void f(x) {
  previo();
  if (es_caso_base(x)) {
    caso_base();
  } else {
    pre_recursivo();
    f(x);
  }
}
```

```
void f(x) {
  previo();
  while (!es_caso_base(x)) {
    pre_recursivo();
    previo();
  }
  caso_base();
}
```

Transformación de `inorder`

Recorrido en inorden

```
void inorder(NodePointer node) {
  if (node ≠ nullptr) {
    inorder(node→left);
    visit(node);
    inorder(node→right);
  }
}
```

Dos funciones auxiliares

`inorder_stack(stack<NodePointer> &st)`

Desapila todos los elementos de `st`, y para cada uno de ellos:

- Visita su raíz.
- Realiza un recorrido en inorden de su hijo derecho

```
void inorder_stack(stack<NodePointer> &st) {
  if (!st.empty()) {
    NodePointer current = st.top();
    st.pop();
    visit(current);
    inorder(current→right);
    inorder_stack(st);
  }
}
```

Dos funciones auxiliares

`inorder_gen(NodePointer node, stack<NodePointer> &st)`

- Realiza un recorrido en inorden de **`node`**.
- Llama a **`inorder_stack`** pasándole **`st`** como parámetro.

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {
  inorder(node);
  inorder_stack(st);
}
```

Si **`st`** es una pila vacía, entonces **`inorder_gen()`** hace lo mismo que **`inorder()`**

Transformando inorder_gen

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {
  inorder(node);
  inorder_stack(st);
}
```

Transformando inorder_gen

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {
  inorder(node);
  inorder_stack(st);
}
```

```
void inorder(NodePointer node) {
  if (node ≠ nullptr) {
    inorder(node→left);
    visit(node);
    inorder(node→right);
  }
}
```

Transformando inorder_gen

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {
  if (node ≠ nullptr) {
    inorder(node→left);
    visit(node);
    inorder(node→right);
  }
  inorder_stack(st);
}
```

```
void inorder(NodePointer node) {
  if (node ≠ nullptr) {
    inorder(node→left);
    visit(node);
    inorder(node→right);
  }
}
```

Transformando inorder_gen

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {
  if (node ≠ nullptr) {
    inorder(node→left);
    visit(node);
    inorder(node→right);
  }
  inorder_stack(st);
}
```

Transformando inorder_gen

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {
  if (node ≠ nullptr) {
    inorder(node→left);
    visit(node);
    inorder(node→right);
  } else {

  }
  inorder_stack(st);
}
```

Transformando `inorder_gen`

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {
  if (node ≠ nullptr) {
    inorder(node→left);
    visit(node);                  } st.push(node);
    inorder(node→right);
    inorder_stack(st);
  } else {
    inorder_stack(st);
  }
}
```

`inorder_stack(stack<NodePointer> st)`

Desapila todos los elementos de `st`, y para cada uno de ellos:

- Visita su raíz.
- Realiza un recorrido en inorden de su hijo derecho

Transformando inorder_gen

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {
  if (node ≠ nullptr) {
    inorder(node→left);
    st.push(node);
    inorder_stack(st);
  } else {
    inorder_stack(st);
  }
}
```

Transformando `inorder_gen`

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {
  if (node ≠ nullptr) {
    st.push(node);
    inorder(node→left);
    inorder_stack(st);
  } else {
    inorder_stack(st);
  }
}
```

`inorder_gen(node→left, st);`

`inorder_gen(NodePointer node, stack<NodePointer> st)`

- Realiza un recorrido en inorden de **`node`**.
- Llama a **`inorder_stack`** pasándole **`st`** como parámetro.

Transformando `inorder_gen`

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {
  if (node ≠ nullptr) {
    st.push(node);
    inorder_gen(node→left, st);
  } else {
    inorder_stack(st);
  }
}
```

¡Es recursiva final!

Transformando inorder_gen

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {
  if (node ≠ nullptr) {
    st.push(node);
    node = node→left;
    inorder_gen(node, st);
  } else {
    inorder_stack(st);
  }
}
```

Transformando inorder_gen

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {
  if (node == nullptr) {
    inorder_stack(st);
  } else {
    st.push(node);
    node = node→left;
    inorder_gen(node, st);
  }
}
```

Transformando inorder_gen

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {
  if (node == nullptr) {
    inorder_stack(st);
  } else {
    st.push(node);
    node = node→left;
    inorder_gen(node, st);
  }
}
```

```
void f(x) {
  previo();
  if (es_caso_base(x)) {
    caso_base();
  } else {
    pre_recursivo();
    f(x);
  }
}
```

```
void f(x) {
  previo();
  while (!es_caso_base(x)) {
    pre_recursivo();
    previo();
  }
  caso_base();
}
```

Transformando inorder_gen

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {
  while (node ≠ nullptr) {
    st.push(node);
    node = node→left;
  }
  inorder_stack(st);
}
```

```
void f(x) {
  previo();
  if (es_caso_base(x)) {
    caso_base();
  } else {
    pre_recursivo();
    f(x);
  }
}
```

```
void f(x) {
  previo();
  while (!es_caso_base(x)) {
    pre_recursivo();
    previo();
  }
  caso_base();
}
```

Transformando inorder_gen

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {
  while (node ≠ nullptr) {
    st.push(node);
    node = node→left;
  }
  inorder_stack(st);
}
```

Transformando inorder_gen

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {
  while (node ≠ nullptr) {
    st.push(node);
    node = node→left;
  }
  inorder_stack(st);
}
```

```
void inorder_stack(stack<NodePointer> &st) {
  if (!st.empty()) {
    NodePointer current = st.top();
    st.pop();
    visit(current);
    inorder(current→right);
    inorder_stack(st);
  }
}
```

Transformando inorder_gen

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {
  while (node ≠ nullptr) {
    st.push(node);
    node = node→left;
  }
  if (!st.empty()) {
    NodePointer current = st.top();
    st.pop();
    visit(current);
    inorder(current→right);
    inorder_stack(st);
  }
}
```

```
void inorder_stack(stack<NodePointer> &st) {
  if (!st.empty()) {
    NodePointer current = st.top();
    st.pop();
    visit(current);
    inorder(current→right);
    inorder_stack(st);
  }
}
```

Transformando inorder_gen

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {
  while (node ≠ nullptr) {
    st.push(node);
    node = node→left;
  }
  if (!st.empty()) {
    NodePointer current = st.top();
    st.pop();
    visit(current);
    inorder(current→right);
    inorder_stack(st);
  }
}
```

`inorder(current→right);` `inorder_stack(st);` } `inorder_gen(current→right, st);`

`inorder_gen(NodePointer node, stack<NodePointer> st)`

- Realiza un recorrido en inorden de **node**.
- Llama a **inorder_stack** pasándole **st** como parámetro.

Transformando inorder_gen

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {
  while (node ≠ nullptr) {
    st.push(node);
    node = node→left;
  }
  if (!st.empty()) {
    NodePointer current = st.top();
    st.pop();
    visit(current);
    inorder_gen(current→right, st);
  }
}
```

¡Es recursiva final!

Transformando `inorder_gen`

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {
  while (node ≠ nullptr) {
    st.push(node);
    node = node→left;
  }
  if (!st.empty()) {
    NodePointer current = st.top();
    st.pop();
    visit(current);
    node = current→right;
    inorder_gen(node, st);
  }
}
```

¡Es recursiva final!

Transformando inorder_gen

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {
  while (node ≠ nullptr) {
    st.push(node);
    node = node→left;
  }
  if (st.empty()) {

  } else {
    NodePointer current = st.top();
    st.pop();
    visit(current);
    node = current→right;
    inorder_gen(node, st);
  }
}
```

```
void f(x) {
  previo();
  if (es_caso_base(x)) {
    caso_base();
  } else {
    pre_recursivo();
    f(x);
  }
}
```

Transformando inorder_gen

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {
  while (node ≠ nullptr) {
    st.push(node);
    node = node→left;
  }
  while (!st.empty()) {
    NodePointer current = st.top();
    st.pop();
    visit(current);
    node = current→right;
    while (node ≠ nullptr) {
      st.push(node);
      node = node→left;
    }
  }
}
```

¡Es iterativa!

Versión iterativa

```
stack<NodePointer> st;
NodePointer node = root;

while (node ≠ nullptr) {
  st.push(node);
  node = node→left;
}
while (!st.empty()) {
  NodePointer current = st.top();
  st.pop();
  visit(current);
  node = current→right;
  while (node ≠ nullptr) {
    st.push(node);
    node = node→left;
  }
}
```

Comparación

```
stack<NodePointer> st;
NodePointer node = root;

while (node ≠ nullptr) {
  st.push(node);
  node = node→left;
}
while (!st.empty()) {
  NodePointer current = st.top();
  st.pop();
  visit(current);
  node = current→right;
  while (node ≠ nullptr) {
    st.push(node);
    node = node→left;
  }
}
```

```
stack<NodePointer> st;
descend_and_push(root, st);

while (!st.empty()) {
  NodePointer x = st.top();
  st.pop();
  visit(x);
  descend_and_push(x→right, st);
}
```

ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS ARBORESCENTES

Iteradores en árboles

Manuel Montenegro Montes
Departamento de Sistemas Informáticos y Computación
Facultad de Informática – Universidad Complutense de Madrid

Recordatorio

```
void inorder(NodePointer &node) {

  std::stack<NodePointer> st;
  descend_and_push(node, st);

  while (!st.empty()) {
    NodePointer x = st.top();
    visit(x→elem);
    st.pop();
    descend_and_push(x→right, st);
  }

}
```

¿Cómo implementar un iterador?

- Un iterador debe simular este recorrido, pero «por partes».

```
void inorder(NodePointer &node) {

  std::stack<NodePointer> st;                          auto it = tree.begin();
  descend_and_push(node, st);

  while (!st.empty()) {
    NodePointer x = st.top();                                    visit(*it);
    visit(x->elem);
    st.pop();                                                      ++it;
    descend_and_push(x->right, st);
  }

}
```

```
for (auto it = tree.begin(); it != tree.end(); ++it) {
  visit(*it);
}
```

Interfaz de iteradores

```
template<class T>
class BinTree {
public:

  iterator begin();
  iterator end();

  class iterator {
  public:
    T & operator*() const;
    iterator & operator++();
    bool operator==(const iterator &other);
    bool operator≠(const iterator &other);

     ...
  };

};
```

Implementación privada de iteradores

```
class iterator {
public:
  T & operator*() const;
  iterator & operator++();
  bool operator==(const iterator &other);
  bool operator≠(const iterator &other);

private:
  iterator();
  iterator(const NodePointer &root);

  std::stack<NodePointer> st;
};
```

```
void inorder(NodePointer &node) {

  std::stack<NodePointer> st;
  descend_and_push(node, st);

  while (!st.empty()) {
    NodePointer x = st.top();
    visit(x→elem);
    st.pop();
    descend_and_push(x→right, st);
  }

}
```

Implementación privada de iteradores

```
class iterator {
public:
  T & operator*() const;
  iterator & operator++();
  bool operator==(const iterator &other);
  bool operator≠(const iterator &other);

private:
  iterator() { }
  iterator(const NodePointer &root) {
    BinTree::descend_and_push(root, st);
  }

  std::stack<NodePointer> st;
};
```

```
void inorder(NodePointer &node) {

  std::stack<NodePointer> st;
  descend_and_push(node, st);

  while (!st.empty()) {
    NodePointer x = st.top();
    visit(x→elem);
    st.pop();
    descend_and_push(x→right, st);
  }

}
```

Implementación privada de iteradores

```
class iterator {
public:
  T & operator*() const {
    assert(!st.empty());
    return st.top()→elem;
  }
  iterator & operator++();
  bool operator==(const iterator &other);
  bool operator≠(const iterator &other);

private:
  iterator();
  iterator(const NodePointer &root);

  std::stack<NodePointer> st;
};
```

```
void inorder(NodePointer &node) {

  std::stack<NodePointer> st;
  descend_and_push(node, st);

  while (!st.empty()) {
    NodePointer x = st.top();
    visit(x→elem);
    st.pop();
    descend_and_push(x→right, st);
  }

}
```

Implementación privada de iteradores

```
class iterator {
public:
  T & operator*() const;

  iterator & operator++() {
    assert(!st.empty());
    NodePointer top = st.top();
    st.pop();
    BinTree::descend_and_push(top→right, st);
    return *this;
  }

  bool operator==(const iterator &other);
  bool operator≠(const iterator &other);

private:
  iterator();
  iterator(const NodePointer &root);

  std::stack<NodePointer> st;
};
```

```
void inorder(NodePointer &node) {

  std::stack<NodePointer> st;
  descend_and_push(node, st);

  while (!st.empty()) {
    NodePointer x = st.top();
    visit(x→elem);
    st.pop();
    descend_and_push(x→right, st);
  }

}
```

Creación de iteradores

```
template<class T>
class BinTree {
public:

  iterator begin() {
    return iterator(root_node);
  }

  iterator end() {
    return iterator();
  }

   ...


};
```

Ejemplo

```
int main() {
  BinTree<int> tree {{{ 9 }, 4, { 5 }}, 7, {{ 10 }, 4, { 6 }}};

  for (auto it = tree.begin(); it =/= tree.end(); ++it) {
    cout << *it << " ";
  }

  return 0;
}
```

```
9 4 5 7 10 4 6
```

Ejemplo

```
int main() {
  BinTree<int> tree {{{ 9 }, 4, { 5 }}, 7, {{ 10 }, 4, { 6 }}};

  for (int x: tree) {
    cout << x << " ";
  }

  return 0;
}
```

```
9 4 5 7 10 4 6
```

Posibles extensiones

- Iteradores constantes: `cbegin()`, `cend()`, etc.
- Diferencia entre postincremento (`it++`) y preincremento (`++it`).
- Aplicación a `SetTree` y `MapTree`.