ESTRUCTURAS DE DATOS

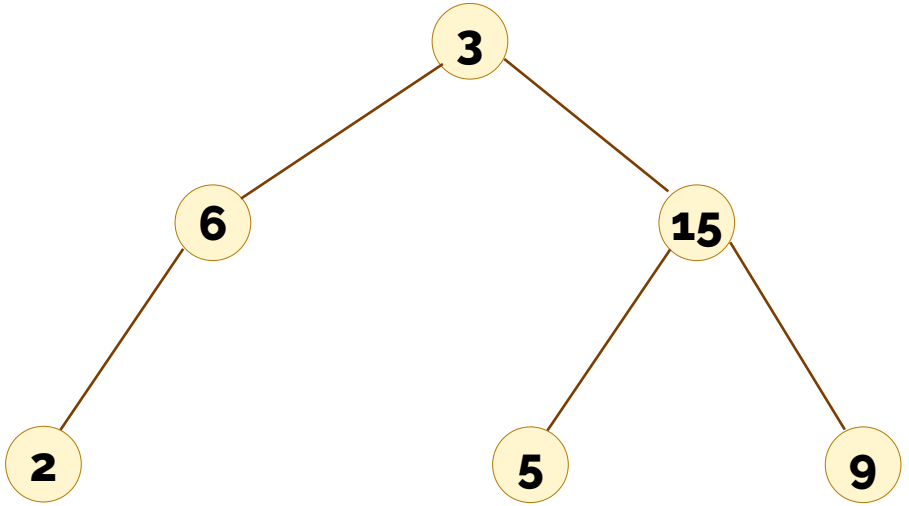
TIPOS ABSTRACTOS DE DATOS ARBORESCENTES

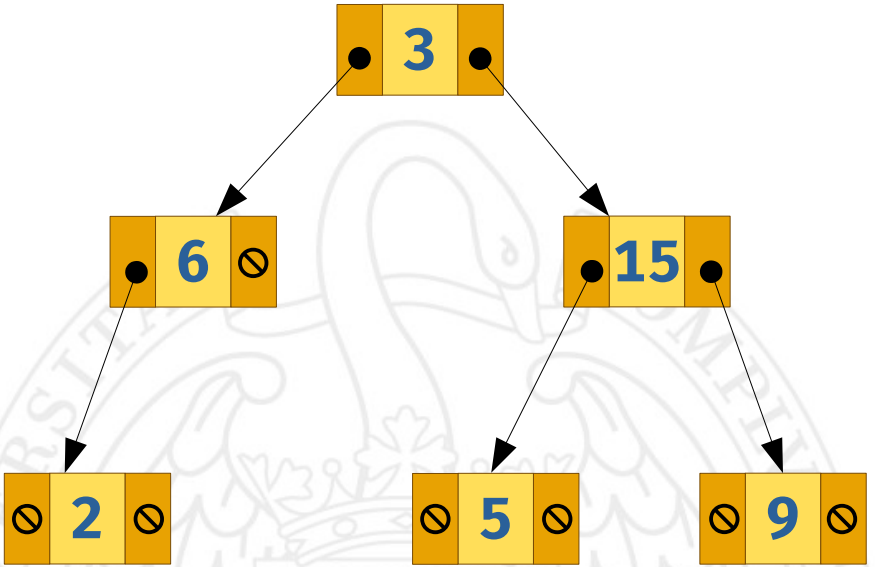
Implementación de árboles binarios

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

Representación mediante nodos

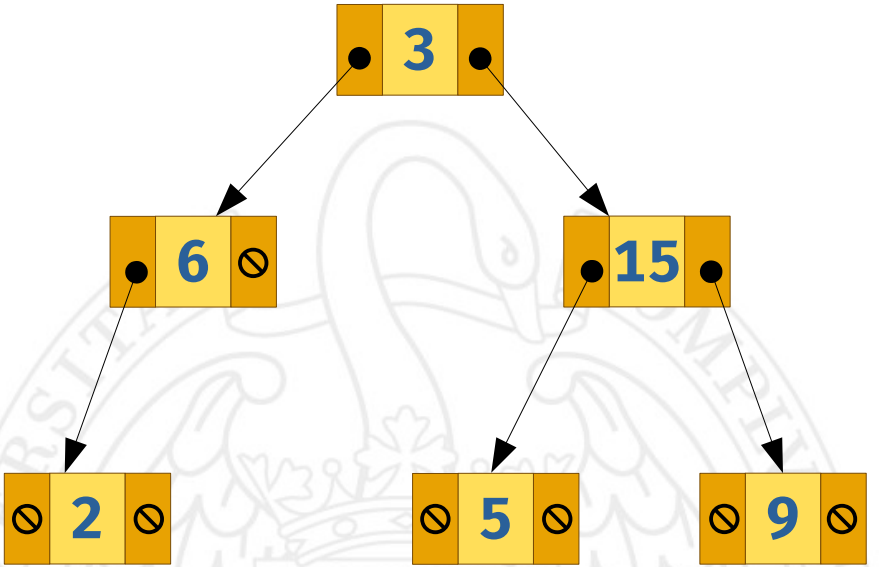
Representando árboles binarios





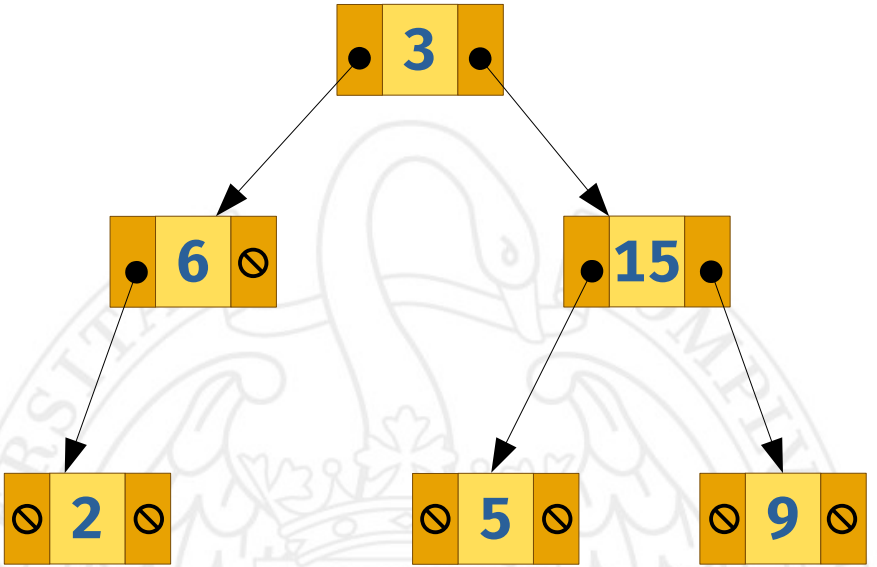
Representando árboles binarios

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



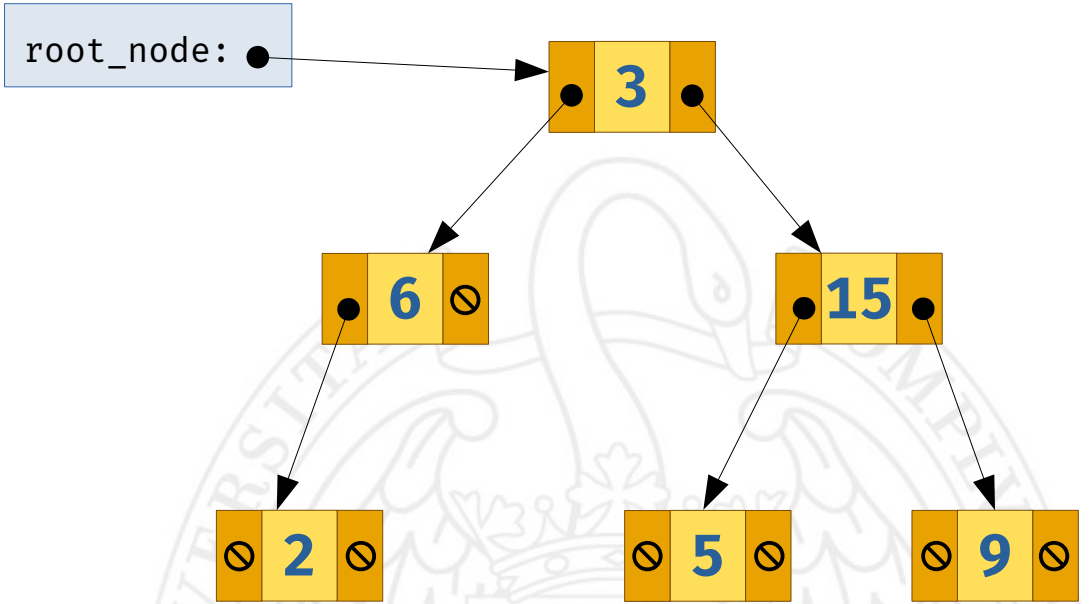
Representando árboles binarios

```
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

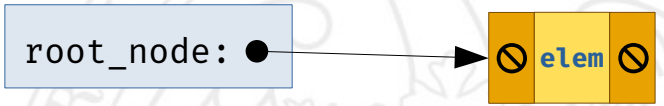
- Constructoras:
  - Crear un árbol vacío: ***create_empty***.
  - Crear una hoja: ***create_leaf***.
  - Crear un árbol a partir de una raíz y dos hijos: ***create_tree***.
- Observadoras:
  - Determinar si el árbol es vacío: ***empty***.
  - Obtener la raíz si el árbol no es vacío: ***root***.
  - 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;
};
```

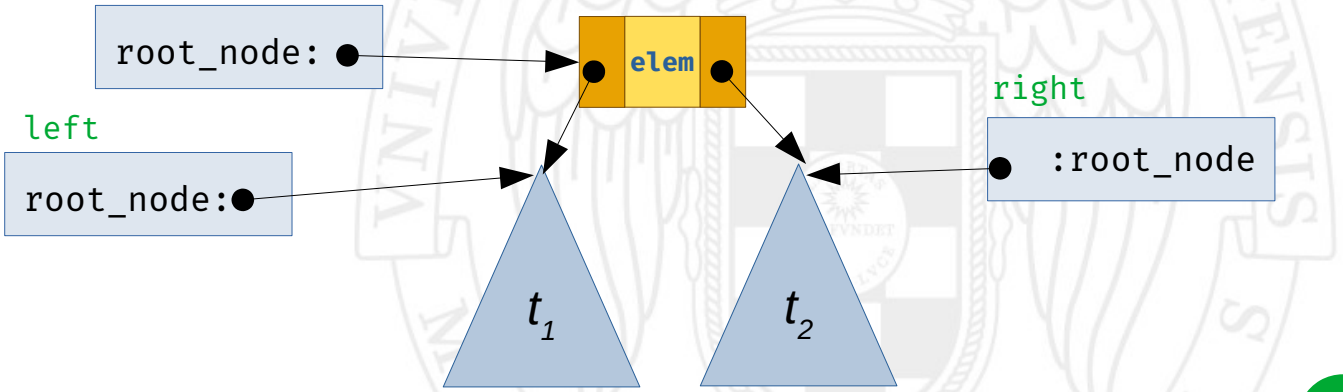
Creación de árboles

```
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;
};
```



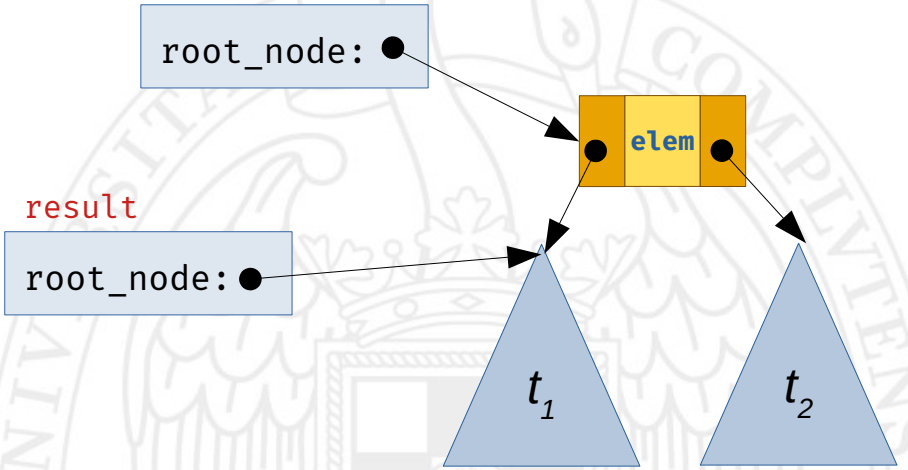
Creación de árboles

```
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;
};
```



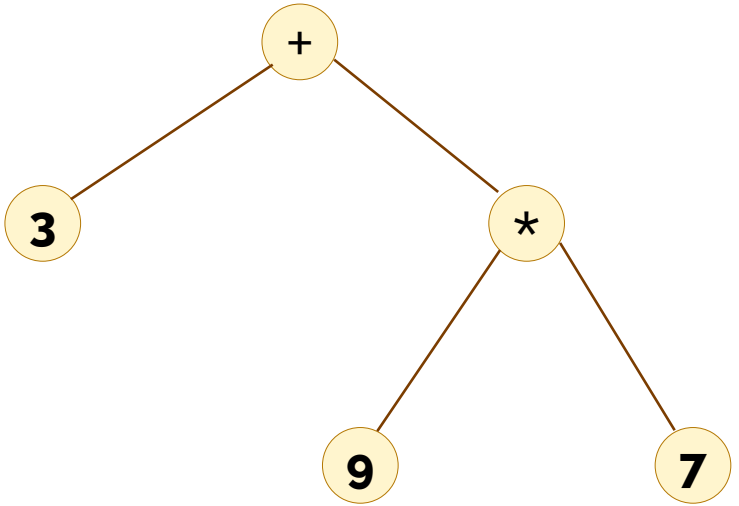
Operaciones observadoras

```
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

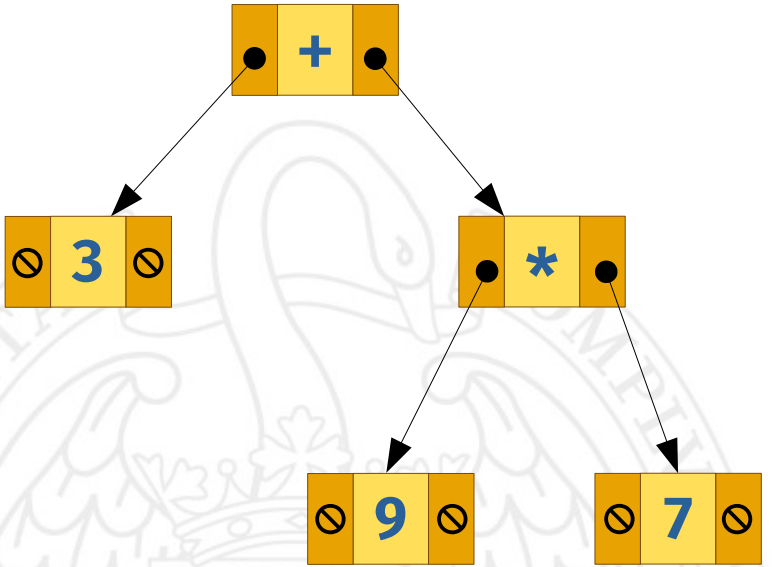



```
((. 3 .) + ((. 9 .) * (. 7 .)))
```

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

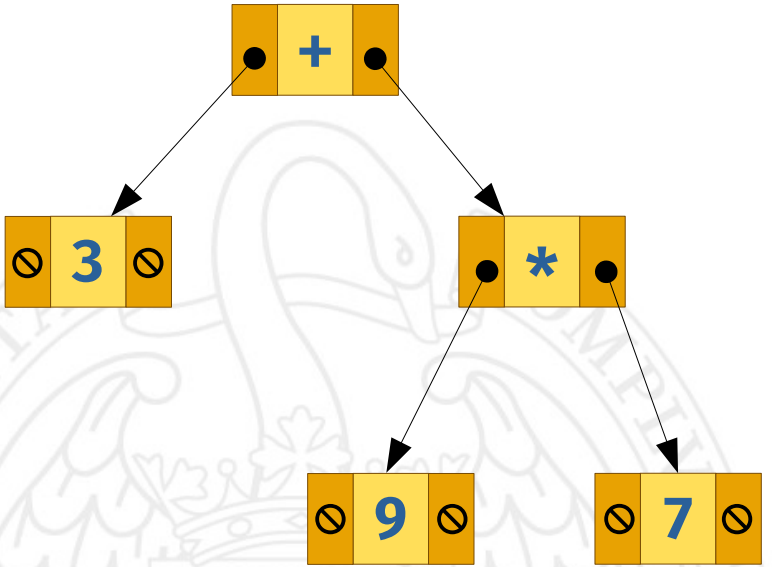
Mostrar un árbol por pantalla

```
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

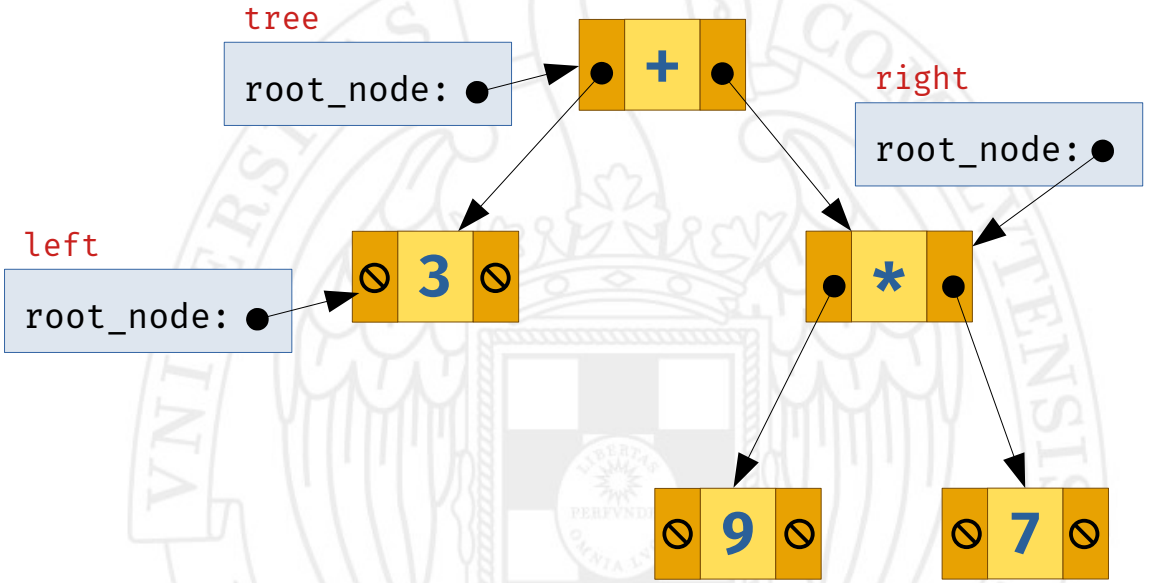
```
template<class T>
class BinTree {
public:
   ...

  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;
}
```

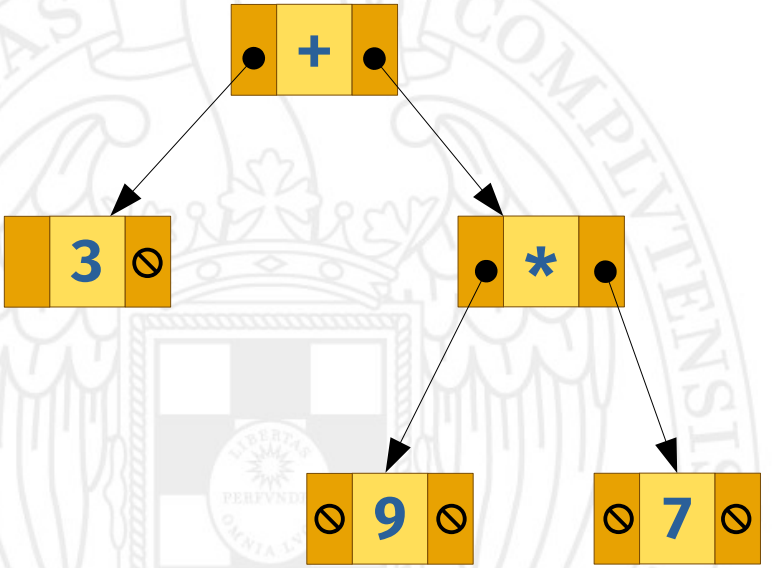
```
((. 3 .) + ((. 9 .) * (. 7 .)))
```



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 .)))
```

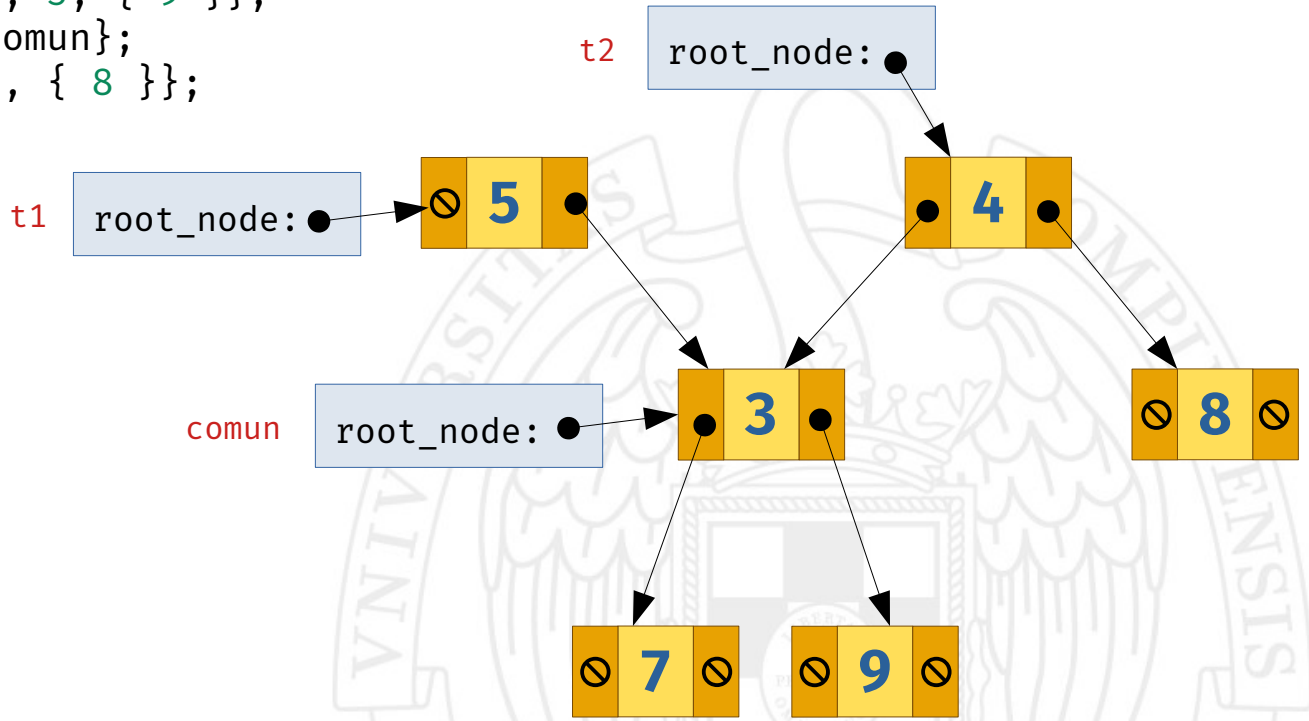
Destrucción de memoria

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?

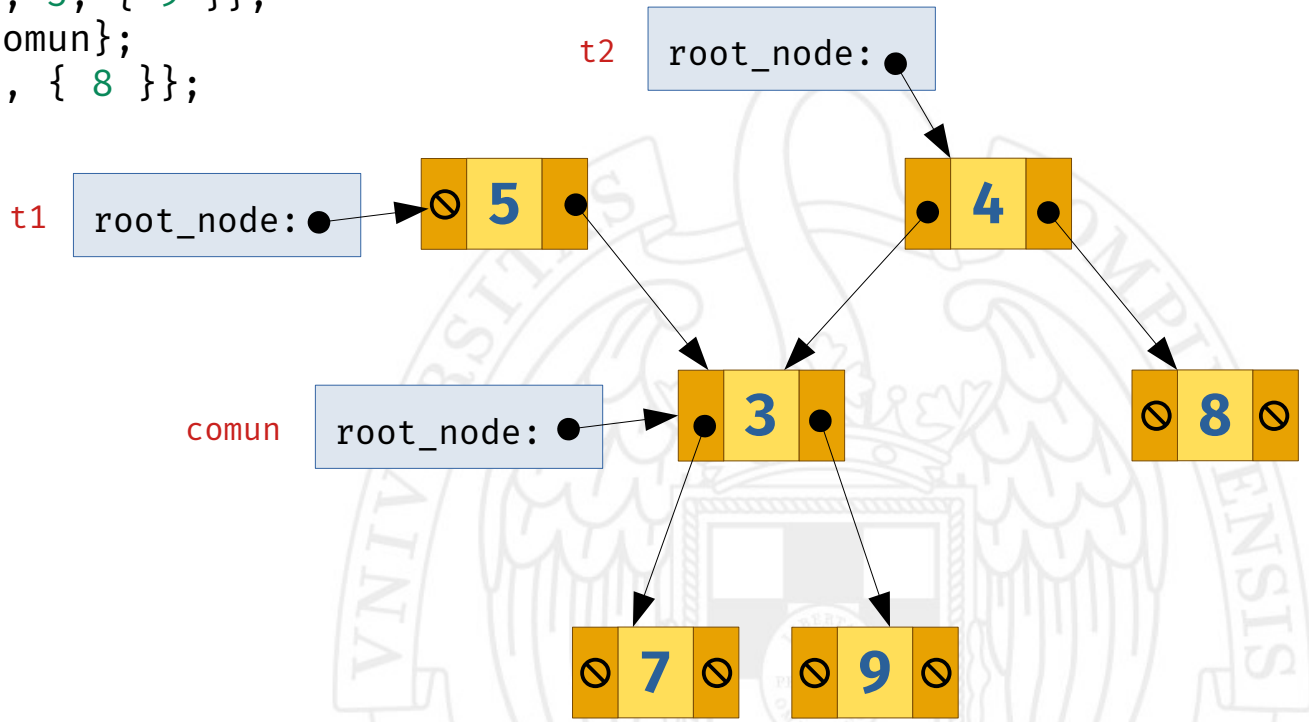
Intento fallido de destructor

```
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.