ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS ARBORESCENTES

Implementación de árboles binarios

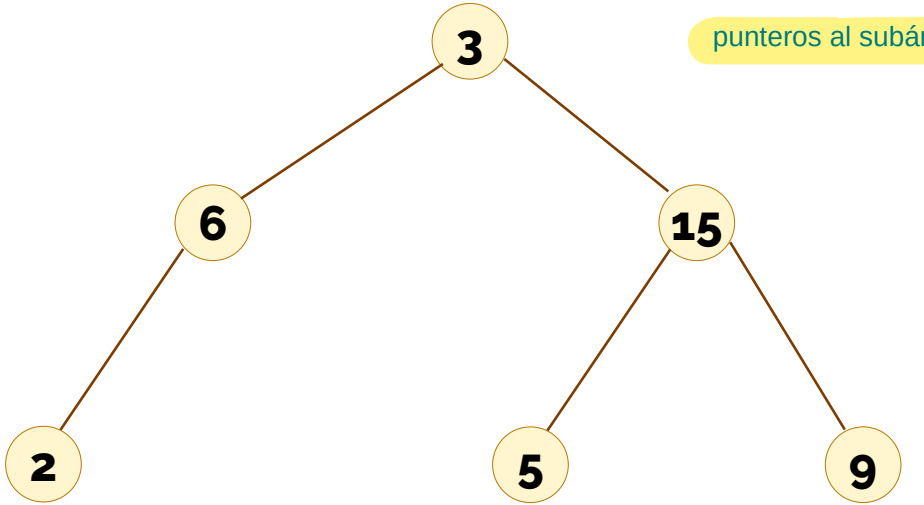
Manuel Montenegro Montes
Departamento de Sistemas Informáticos y Computación
Facultad de Informática – Universidad Complutense de Madrid

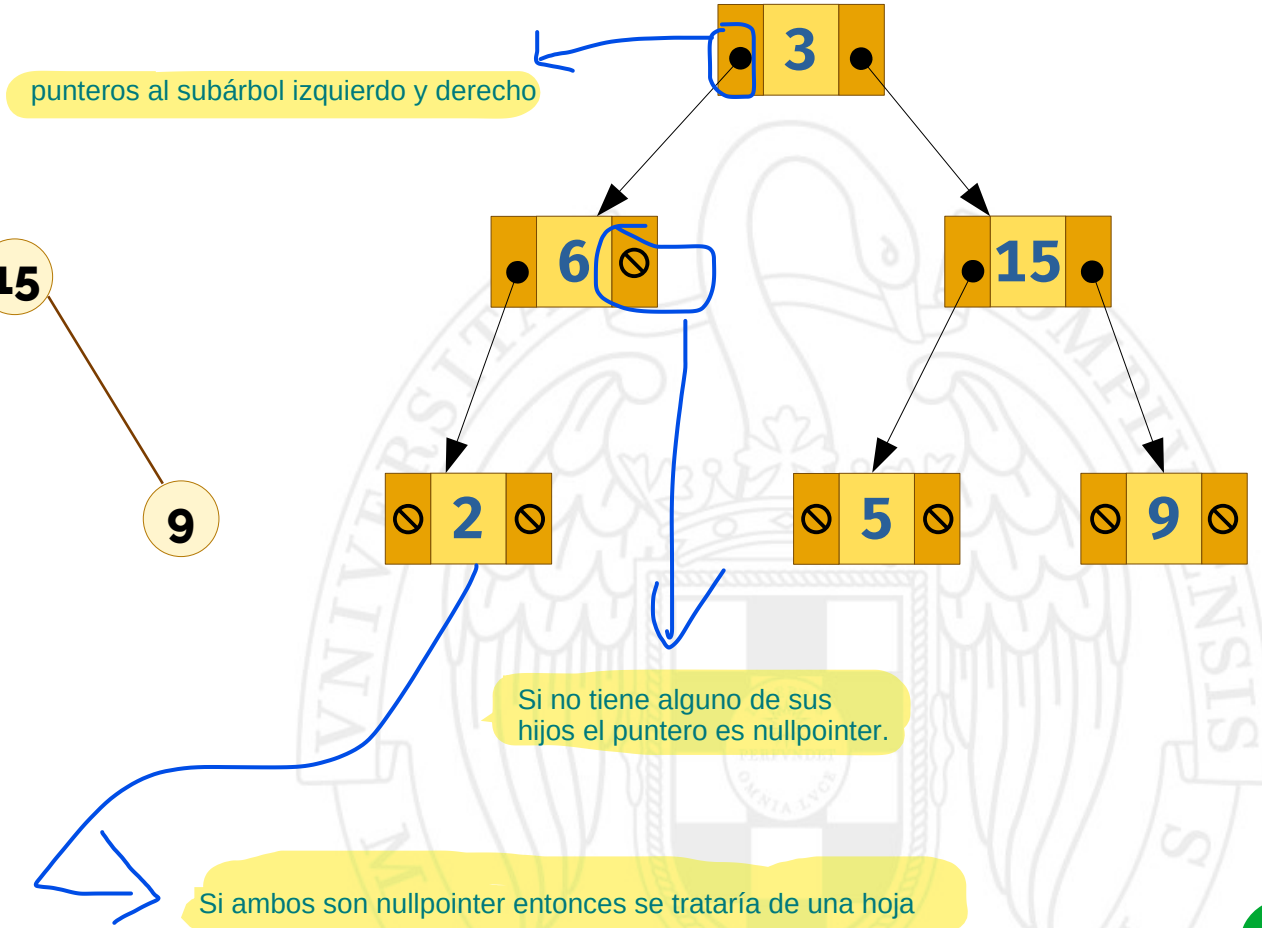
Hay varias formas de representar este TAD de árboles binarios, pero nosotros en este curso solamente nos centramos en una de ellas.

Utiliza los nodos que ya hemos visto

Representación mediante nodos

Representando árboles binarios





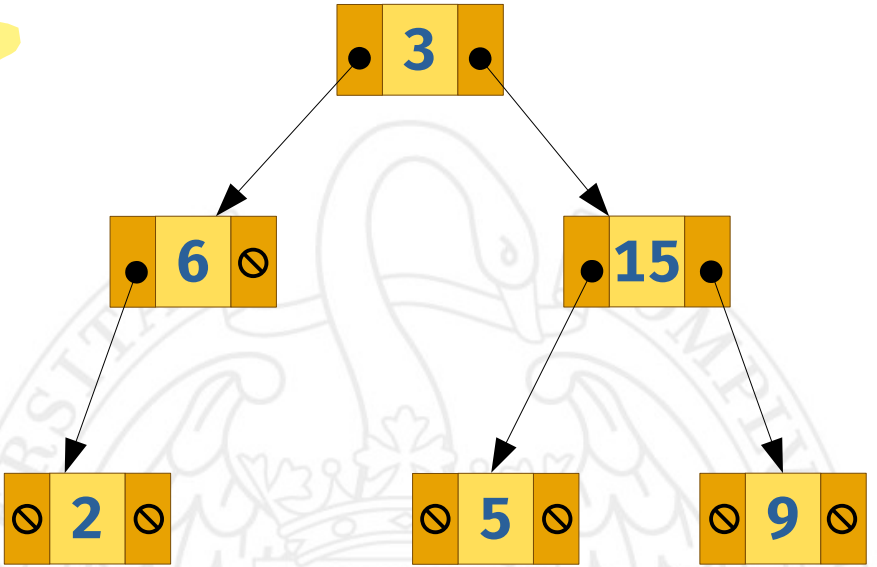
Representando árboles binarios

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

puntero al subárbol izquierdo

puntero al subárbol derecho.

Mediante un registro que nosotros vamos a llamar treenode.



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) { }
};
```

a esto se le llamaba lista de inicialización

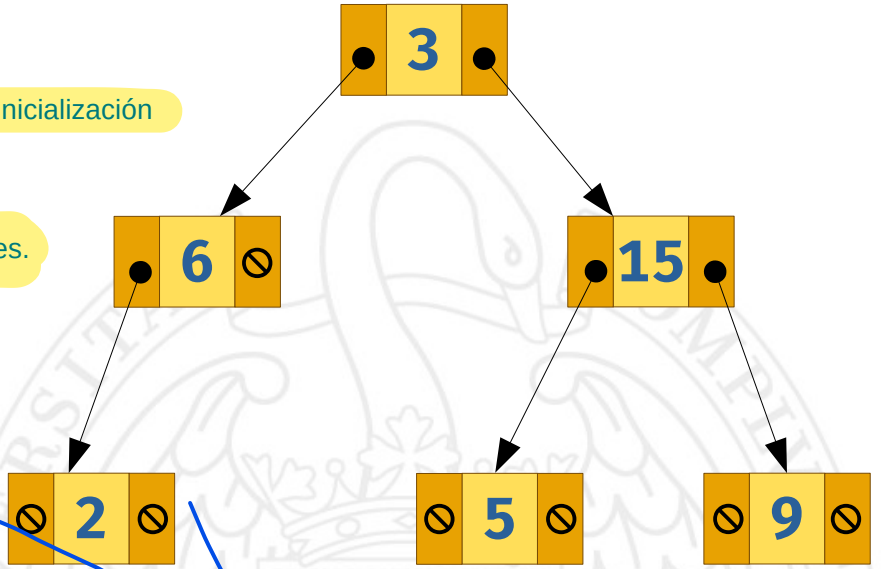
constructor de árboles.

Constructor para crear estos árboles binarios.

Inicializa los atributos a lo que recibe el constructor como parámetros.

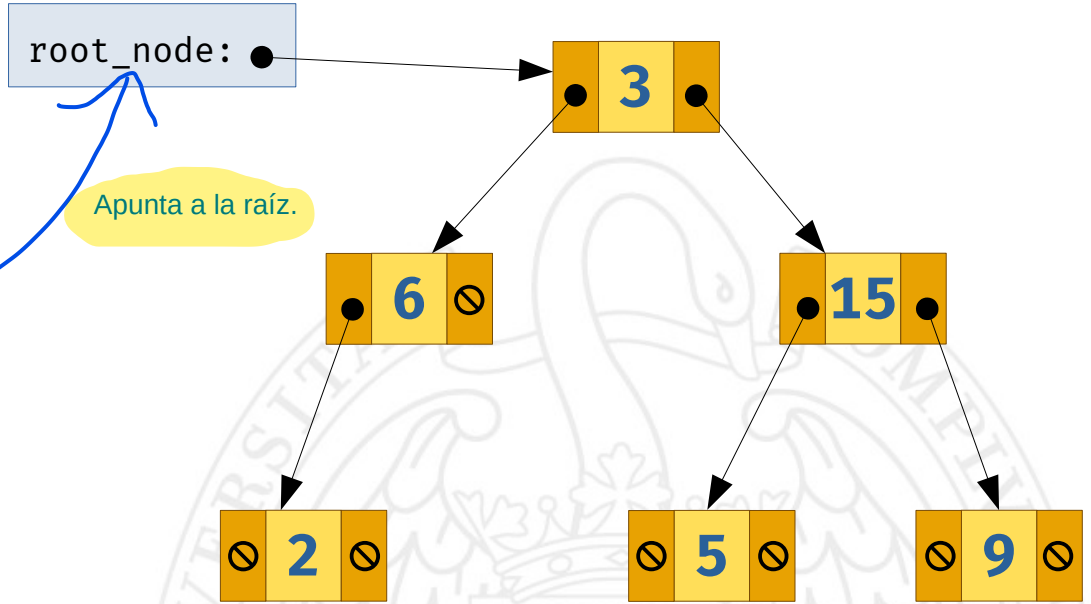
Recibe el puntero al hijo izquierdo, al elemento y al hijo derecho.

raíz del árbol



La clase BinTree

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



Apunta a la raíz.

En el caso en que el árbol es vacío:

`root_node == nullptr`

El registro que se ha implementado arriba.

No se si este root_node, que es un puntero, apuntará al nodo raíz de todos los árboles, tanto del árbol principal como de los subárboles.

Después de la representación vemos ahora como implementar las operaciones.

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;
};
```

Constructor vacío

Construir una hoja (create_leaf)

crear un árbol binario

obtener el elemento raíz de un árbol binario

árbol izquierdo

árbol derecho

si está vacío

todas const porque en teoría no se modifica ningún elemento del árbol.

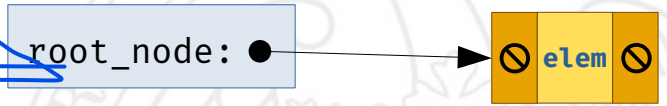
Son solamente métodos de acceso.

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;
};
```

Constructor vacío

Para una hoja

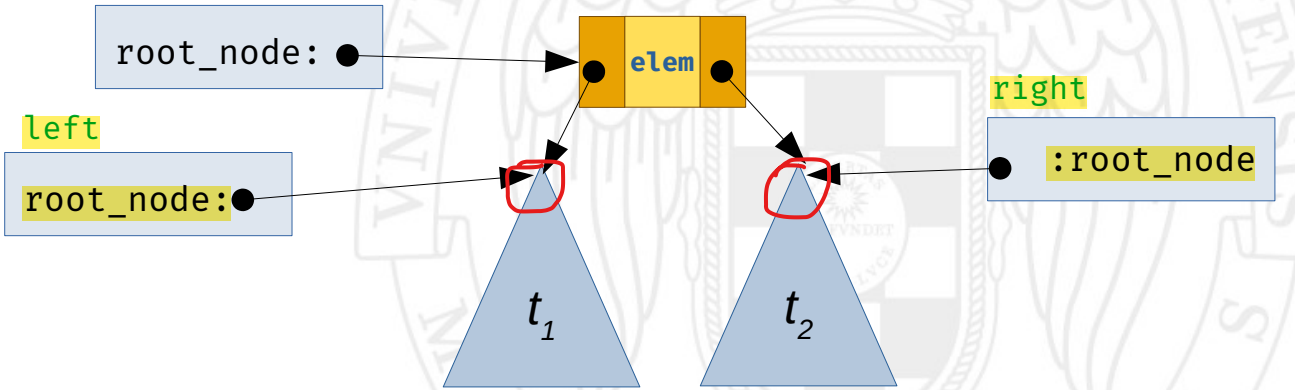


Creación de árboles

```
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;
};
```

raíz del árbol left

raíz del árbol right

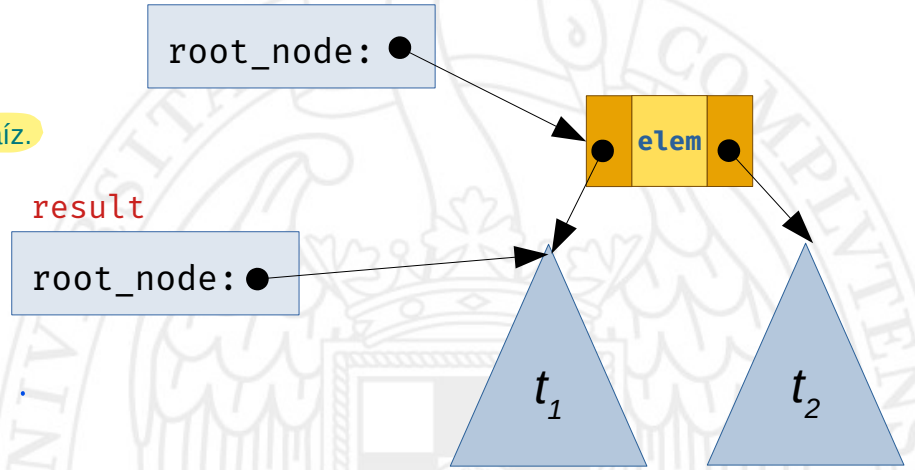


Operaciones observadoras

```
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;
  }

};
```

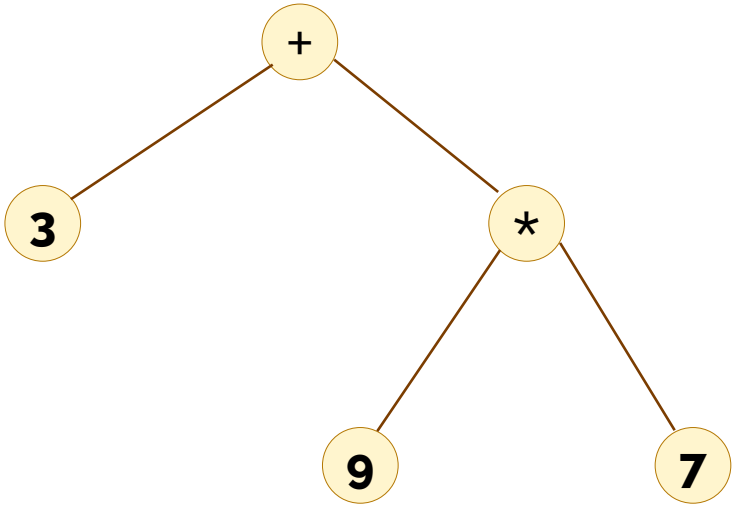
se limita a devolver el valor obtenido en el nodo raíz.

Mismo código para right.

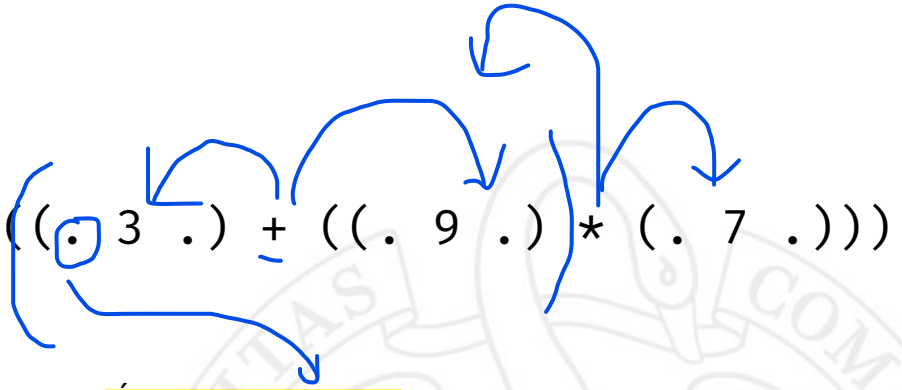


E/S de árboles

Representación textual de un árbol



Representación de los árboles



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

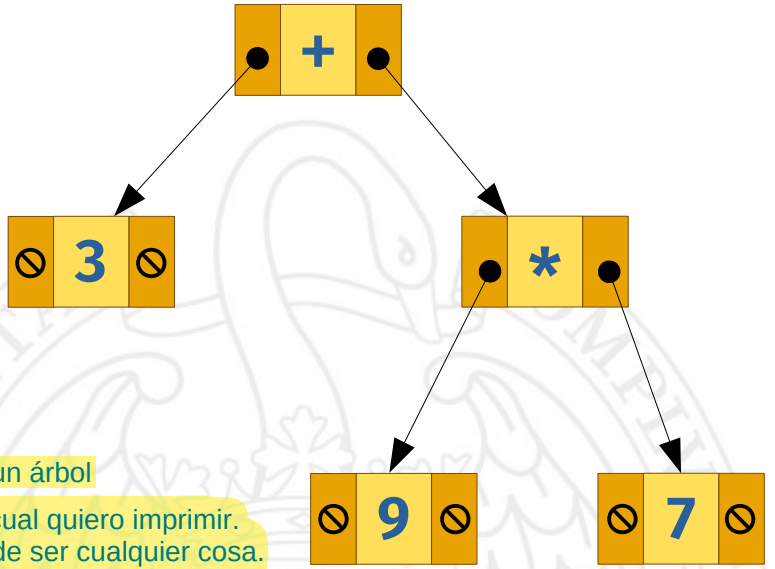
Mostrar un árbol por pantalla

```
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 << ")";
    }
  }
};
```

Puntero a un árbol

Manejador sobre el cual quiero imprimir.
Puede ser cout, puede ser cualquier cosa.

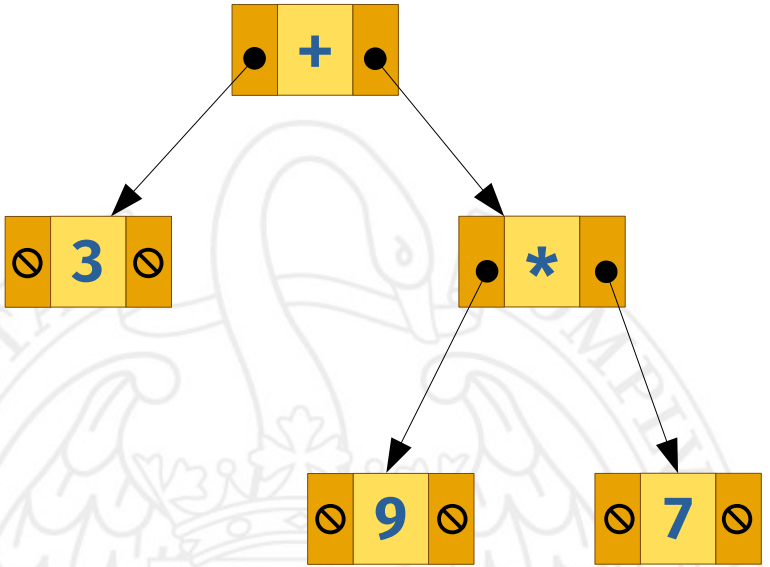
Llamada recursiva para el hijo izquierdo y el hijo derecho.



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;
}
```

raíz



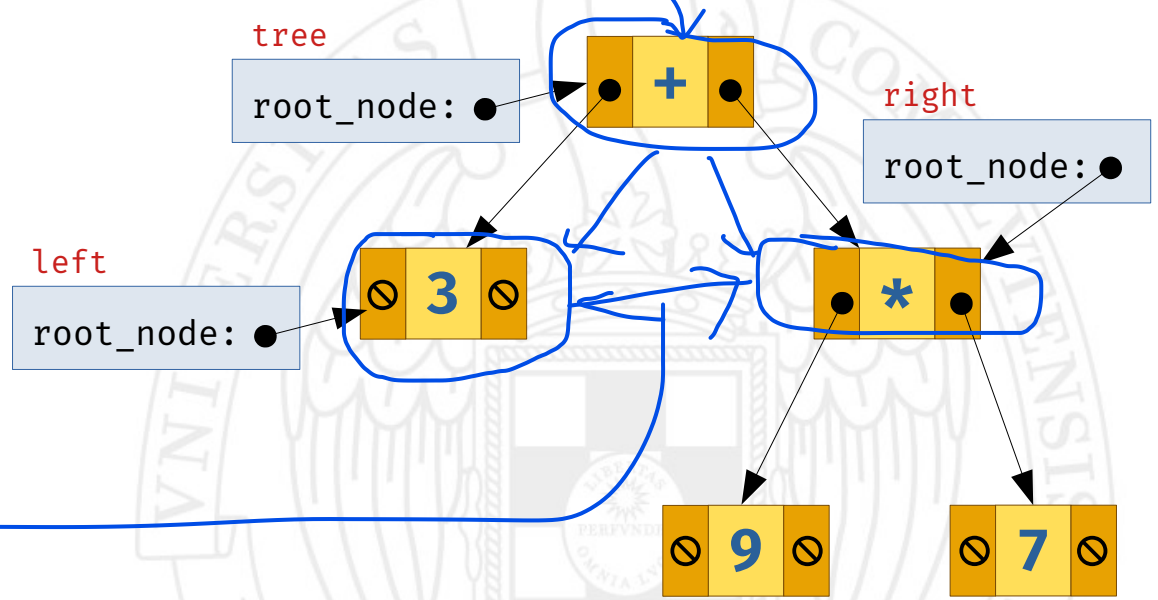
sobrecarga del operador de desplazamiento para poder hacer por ejemplo cout << t.
En este caso no hacemos más que llamar al método display.

Ejemplo

árbol left es un árbol cuya raíz es una hoja ya que la raíz 3 no tiene hijos.

```
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 .)))
```



Los cuadrados azules representan instancias de la clase BinTree. Mientras que los rodeados representan instancias de la clase TreeNode.

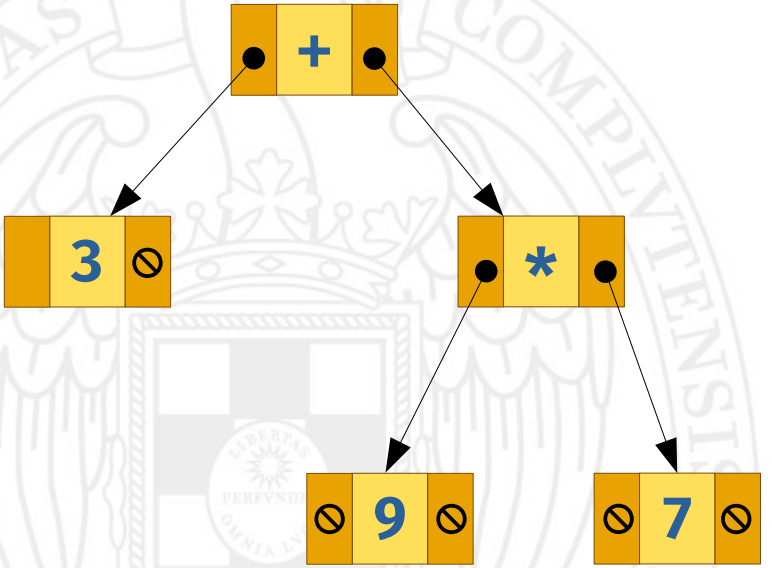
Ejemplo

Llama al constructor de un parámetro

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

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

  return 0;
}
```

También se puede inicializar de esta manera

```
((. 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;
  }
}
```

Lectura recursiva

árbol vacío.

lees recursivamente el hijo izquierdo

Luego leemos el elemento

Luego leemos el hijo derecho

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

Destrucción de memoria

Problema importante

Veremos en otro vídeo como podemos hacer esta liberación. Aquí solo va a explicar cual es el problema que se nos presenta.

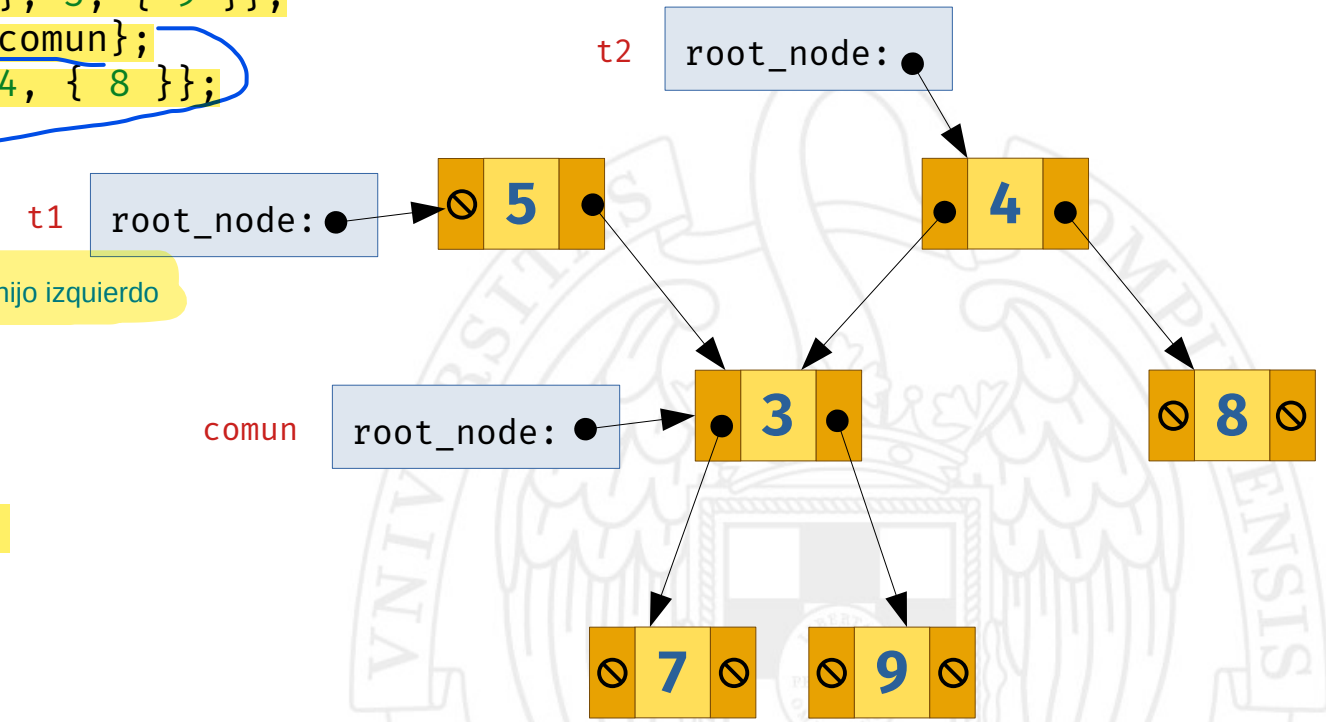
- ¡No estamos liberando la memoria ocupada por los nodos!
- Hay que hacerlo con cuidado...

El problema de la compartición en árboles

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

comun es el hijo derecho

comun es el hijo izquierdo



¿Cómo liberamos la memoria ocupada por los nodos?

Intento fallido de destructor

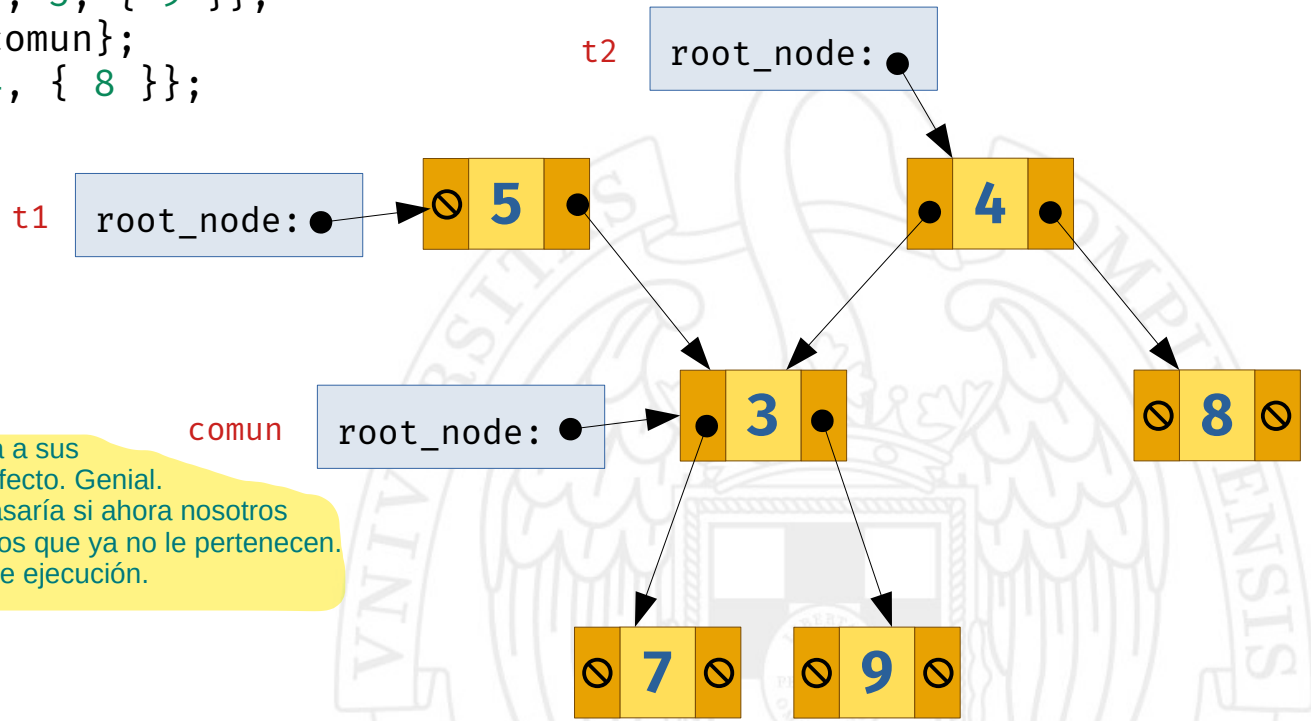
```
template<class T>
class BinTree {
public:
  ...
   ~BinTree() {
     delete_with_children(root_node);
   }

private:

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

Podemos pensar que el árbol es responsable de borrar sus nodos

ESTO NO FUNCIONARIA

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?

Imagina que liberamos el nodo raiz de t1, se llamaría a sus descendientes y eliminaríamos el nodo 5, 3, 7,9. Perfecto. Genial.
Pero no tanto como nosotros nos pensamos. Qué pasaría si ahora nosotros queremos borrar el nodo 4? Se pondría a borrar nodos que ya no le pertenecen.
Lo cual provocará seguramente un error en tiempo de ejecución.

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. Hacer copias de un árbol entero puede llegar a ser bastante costoso.

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

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

Mecanismo para liberar memoria.