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

La busqueda search la inserción y la eliminación de nodos que hemos visto en los vídeos anteriores son la base de este TA

- 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:
   ...
};
```

default

copia

destructor

CONSTRUCTORES.

MUTADORAS

OBSERVADORAS (NO MODIFICAN)

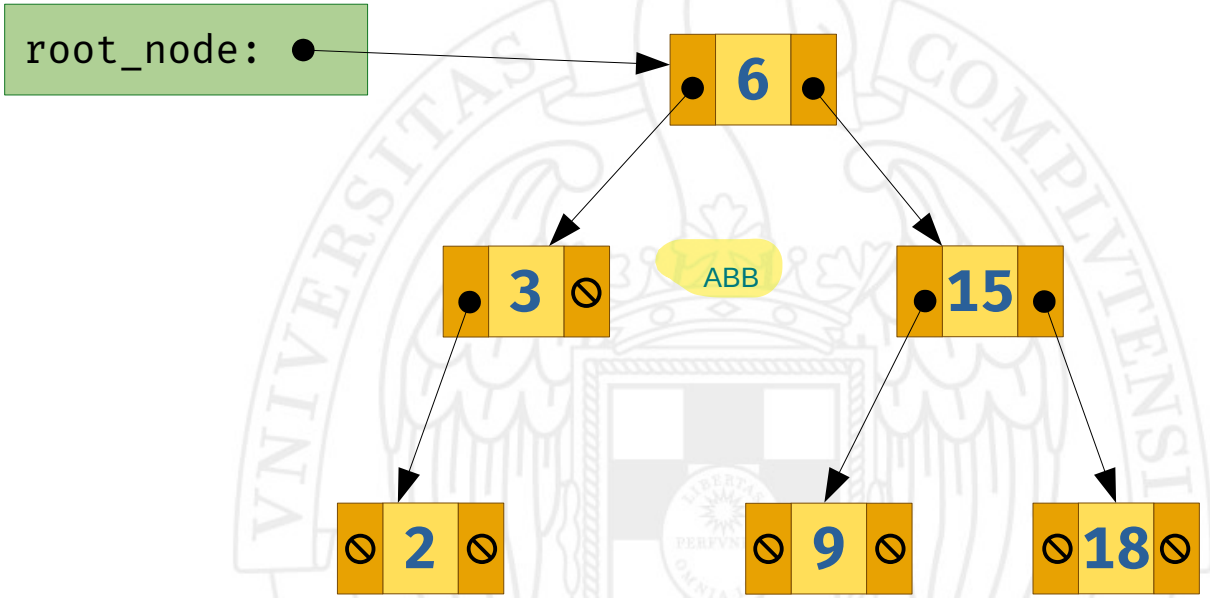
Implementación de SetTree

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

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

  Node *root_node;
};
```

PUNTERO AL NODO RAÍZ.

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

ESTO ES UNA MANERA.

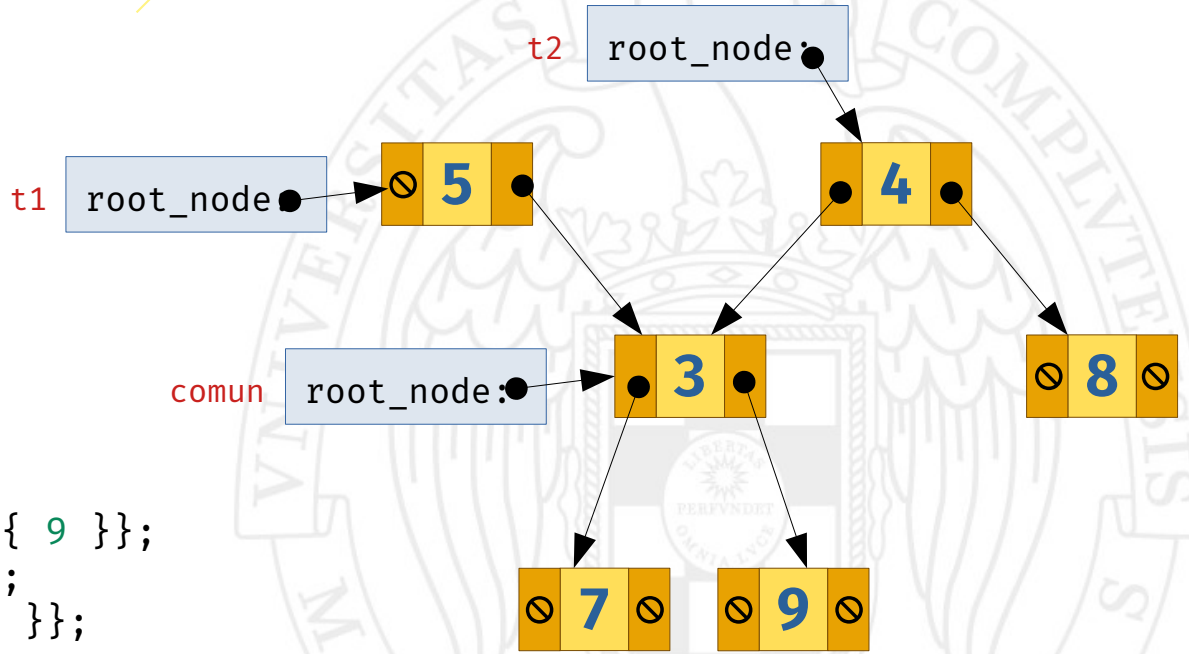


Sobre la compartición

Anteriormente...

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

Borraban de manera automática cada nodo de los árboles.



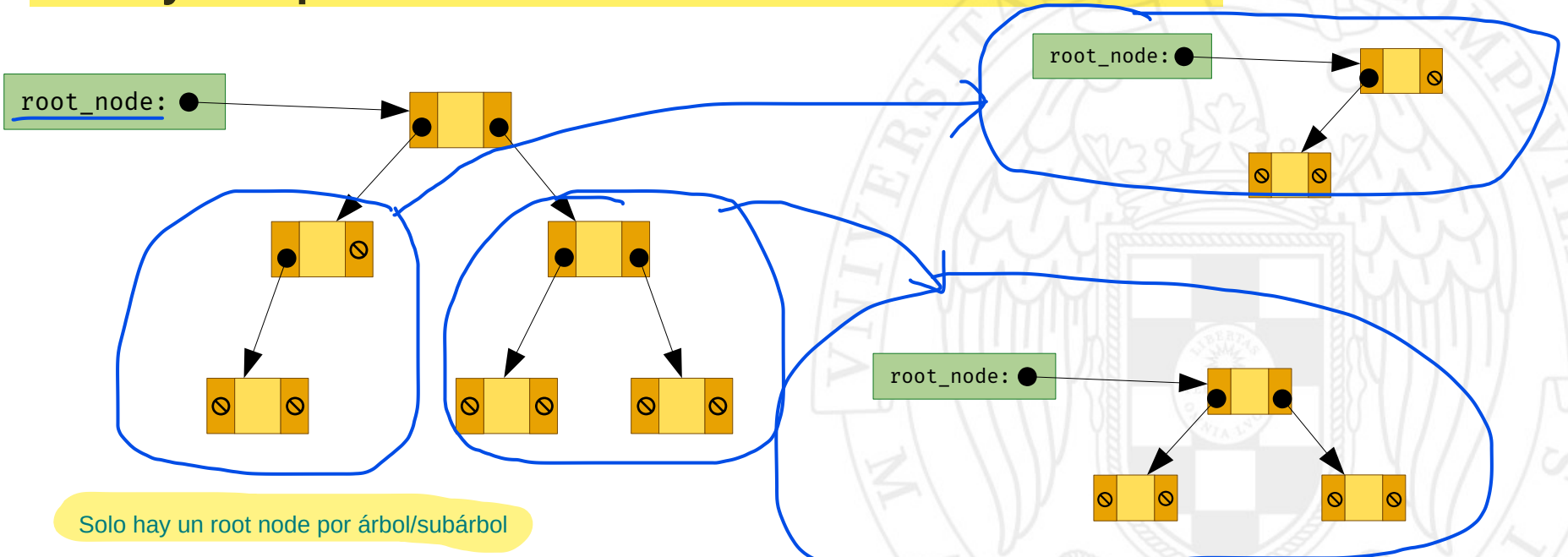

```
BinTree<int> comun = {{ 7 }, 3, { 9 }};
BinTree<int> t1 = {{}, 5, comun};
BinTree<int> t2 = {comun, 4, { 8 }};
```

Pero aquí...

- Implementamos un TAD para conjuntos.
- Cada objeto de la clase `SetTree` apunta a la raíz de su propio árbol de nodos.

cada objeto setTree tiene su propio árbol de nodos.

- **No hay compartición** entre los nodos de dos `SetTree` distintos.



Solo hay un root node por árbol/subárbol

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.

Con el destructor. Ya que NO utilizamos los punteros inteligentes.

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;
};
```

árbol vacío

raíz del conjunto que yo quiero copiar.

Copiaría el other.root_node y todos sus descendientes.

recibe la raíz y la elimina junto a todos sus descendientes.

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);
  ...
};
```

Estos son los que hemos definido en los árboles binarios de búsqueda.

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);
  ...
};
```

Hacen las búsquedas en los árboles binarios de búsqueda ABBs

si la raíz es nula

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()`

Ya que aunque sea equilibrado es lineal y nosotros pasamos por todos los nodos.

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.

de forma que no haría falta recorrer cada vez que queramos saber el size todo el árbol, si no que devolveríamos directamente el número de elementos. Esto tendría coste constante.

Parecido al TAD Lista que ya vimos. Pero ojo este tiene trampa.

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!

Puede que no esté el elemento que queremos borrar de forma que no lo podamos borrar. En ese caso aun así nos estaría restando 1 al número de elementos.

Porque si el elemento que queríamos insertar ya se encuentra en el conjunto, no podríamos añadirlo y sin embargo nos seguiría sumando 1 al número de elementos.

¿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.

no debería de aumentarlo

- En este caso, no tenemos que incrementar `num_elems`.
- Cambiamos la función auxiliar `insert`:

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

por:

Si lo ha encontrado o no lo ha encontrado.

```
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:

Si finalmente se ha eliminado.

```
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};
  }
}
```

se inserta.

raíz del árbol contiene el elemento a insertar. En erase es al contrario.

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;
  ...
};
```

si devuelve true entonces ahora sí incremento el número de elementos.

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`	$O(\log n)$	$O(n)$

en vez de ser siempre coste lineal

.count()

n = número de elementos del conjunto