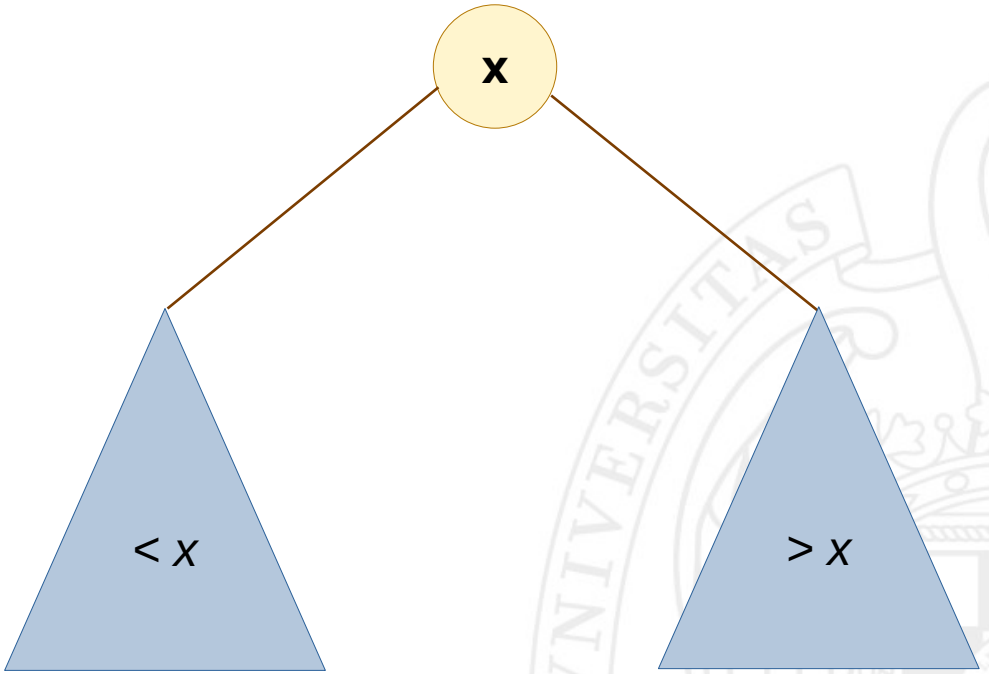
ESTRUCTURAS DE DATOS

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);
```

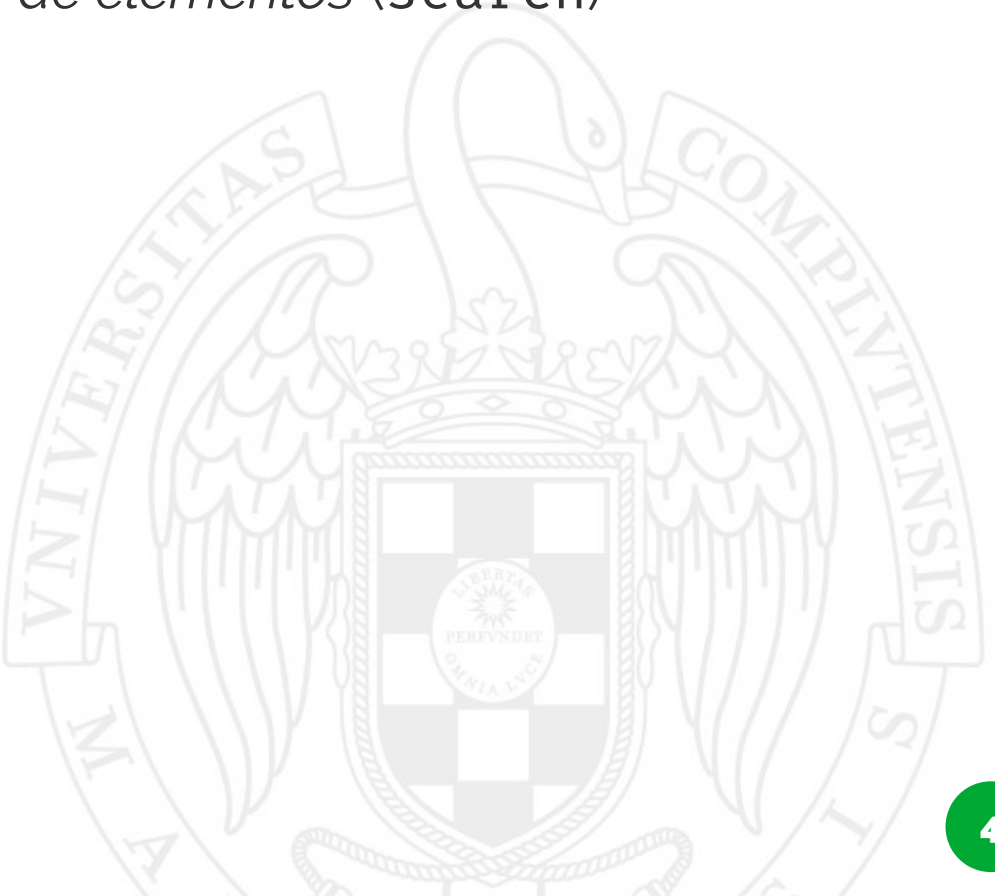
Dos fases

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 {

  }
}
```

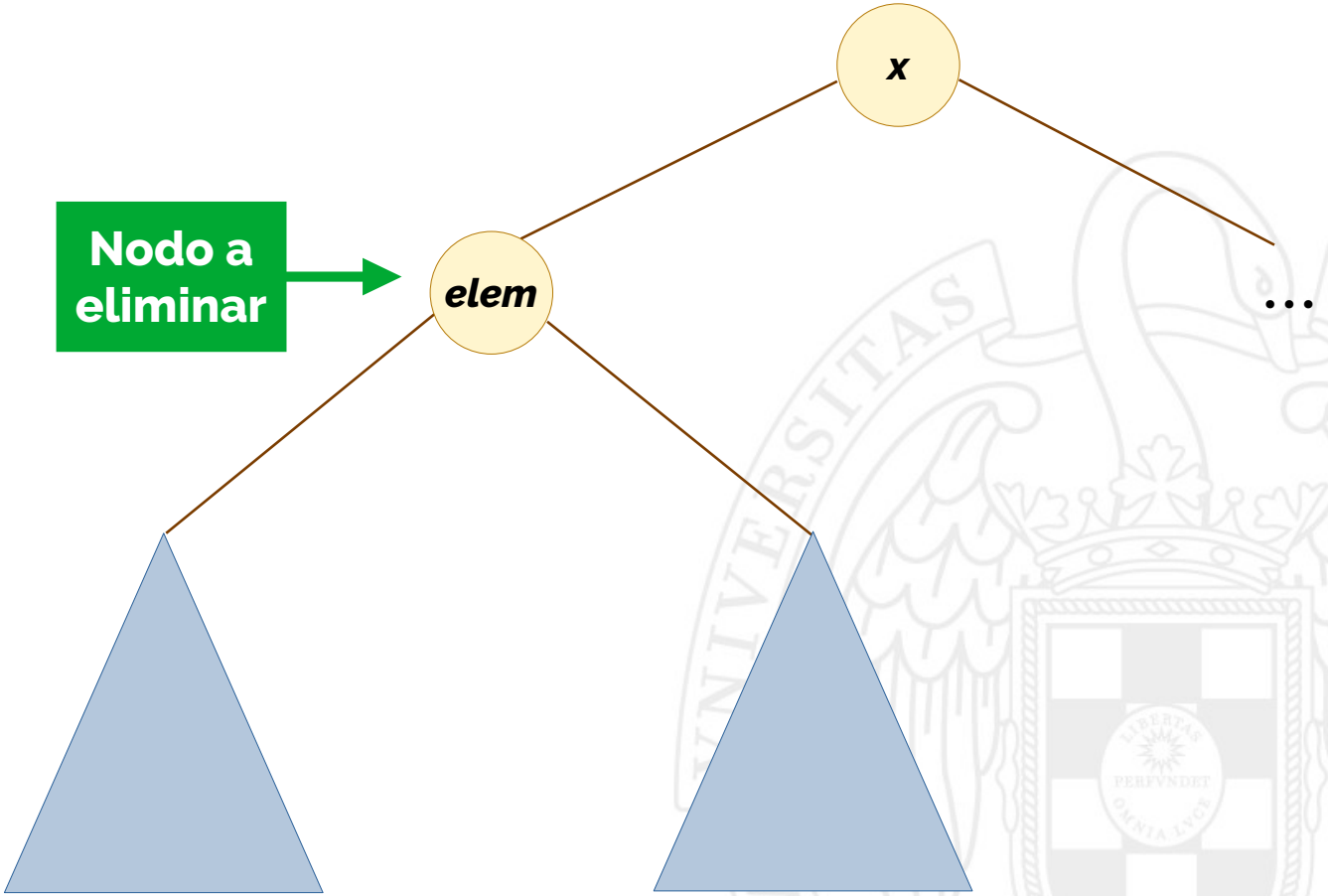
Borramos en el hijo derecho

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);
  }
}
```

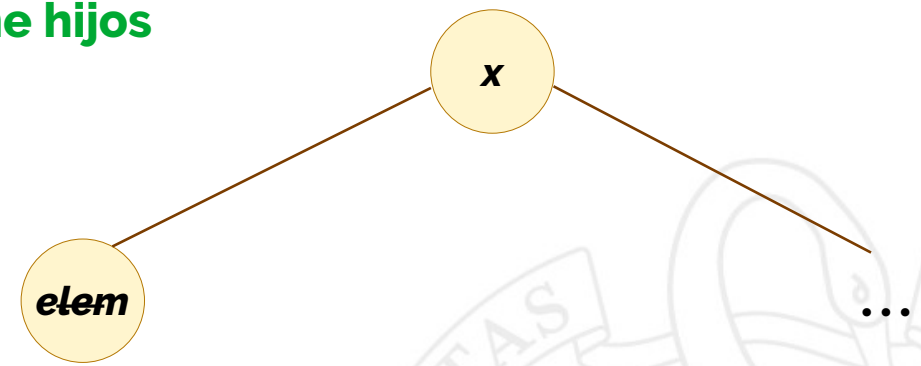
Caso `root→elem == elem`
Pasamos a fase 2

Fase 2: eliminación del nodo



Fase 2: eliminación del nodo

Caso 1: El nodo a eliminar no tiene hijos



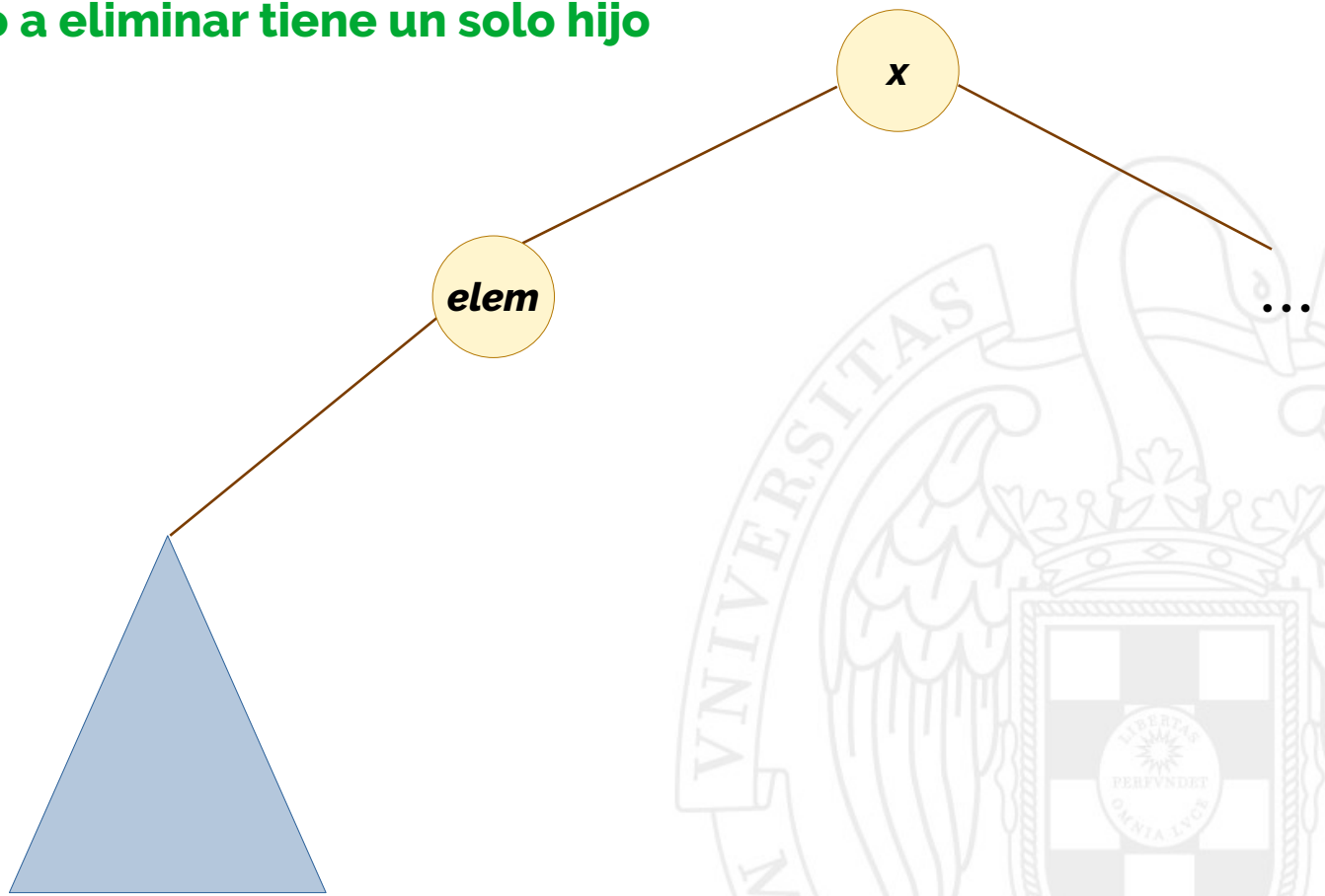
Fase 2: eliminación del nodo

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 {




  }
}
```

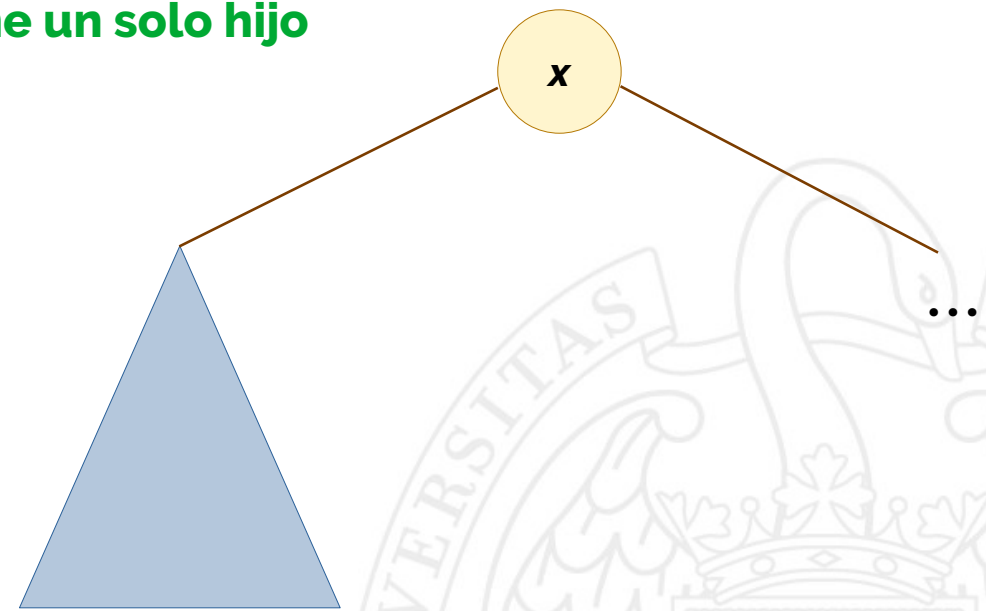
Fase 2: eliminación del nodo

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



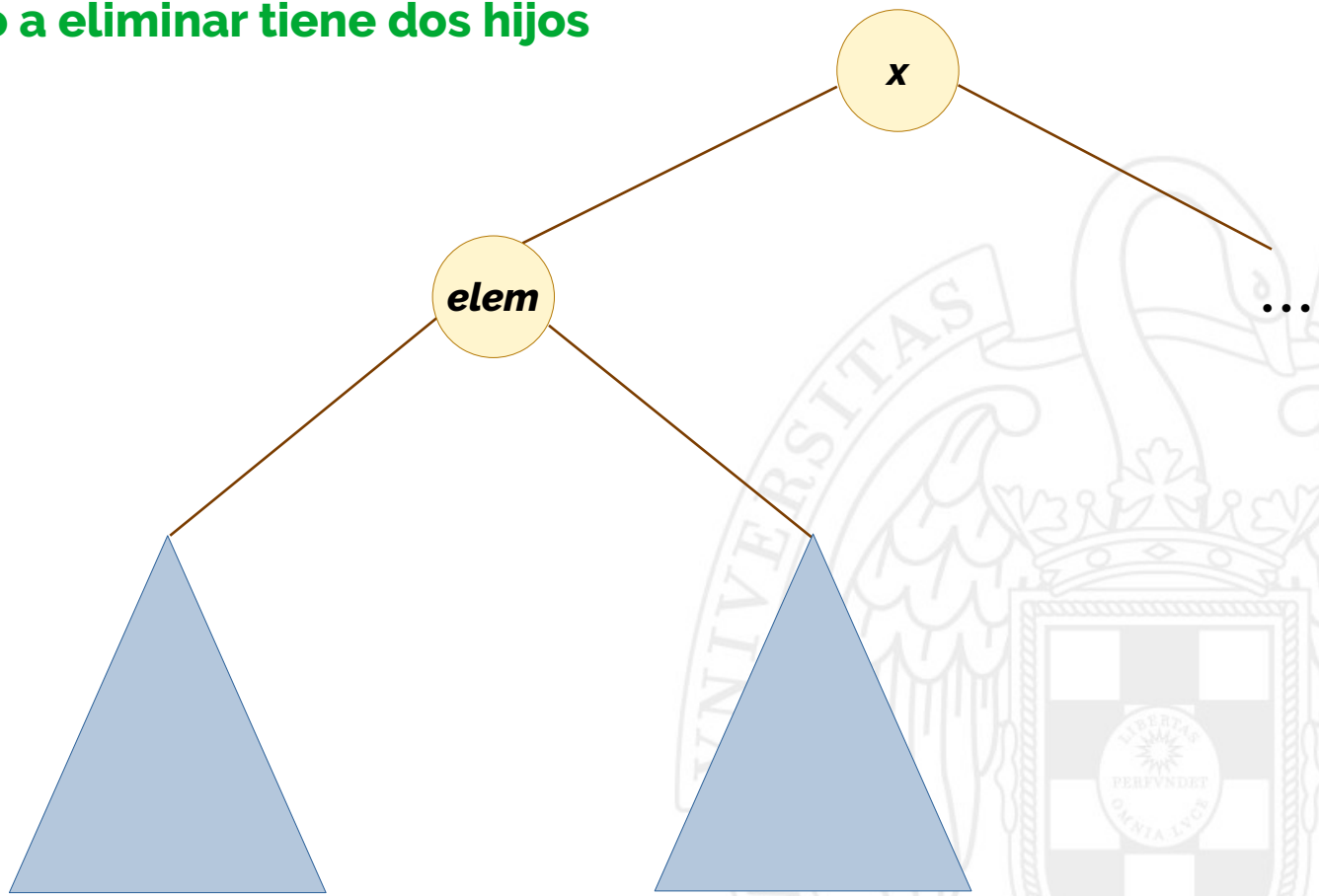
Fase 2: eliminación del nodo

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 {




  }
}
```

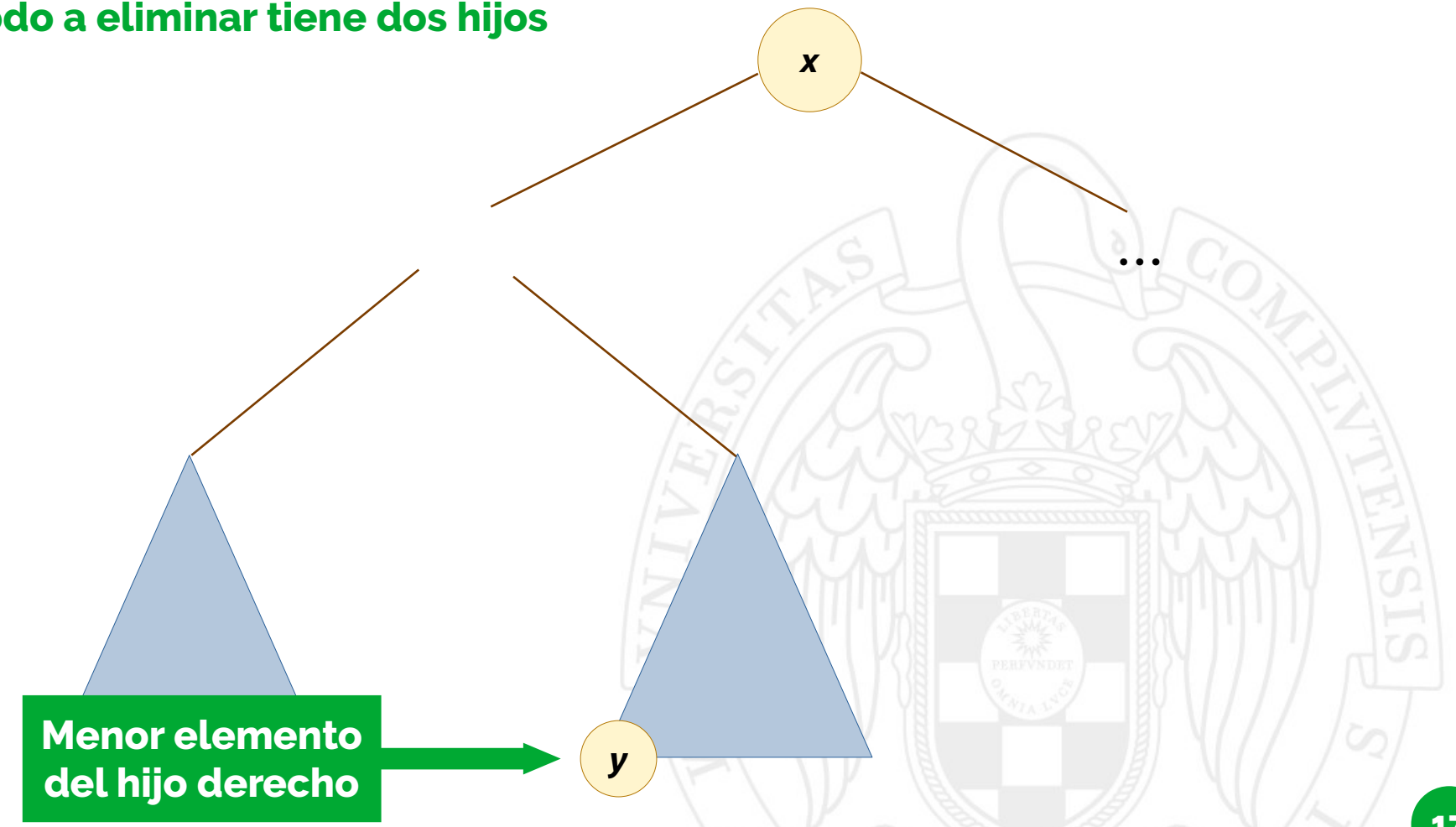
Fase 2: eliminación del nodo

Caso 3: El nodo a eliminar tiene dos hijos



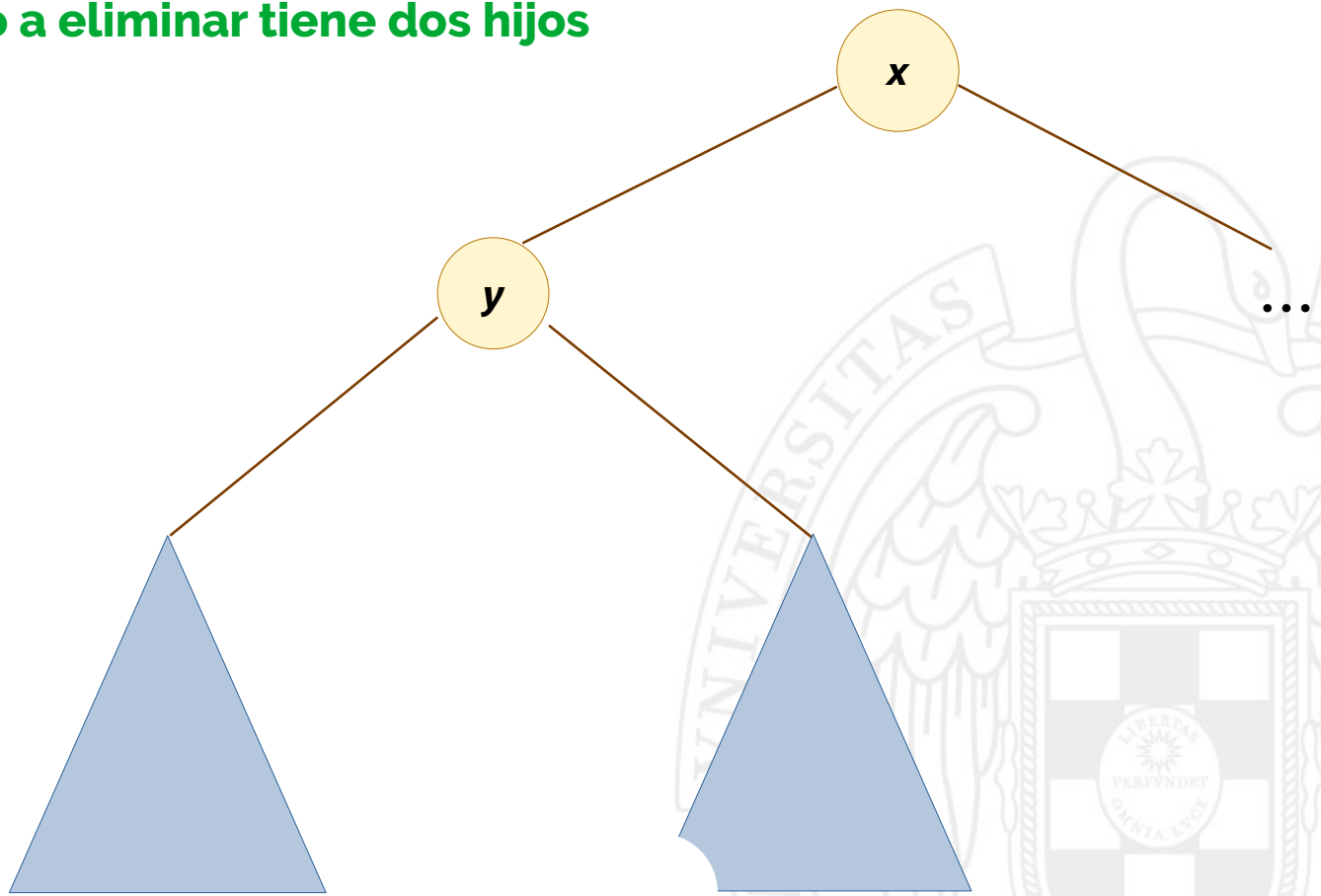
Fase 2: eliminación del nodo

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;
  }
}
```

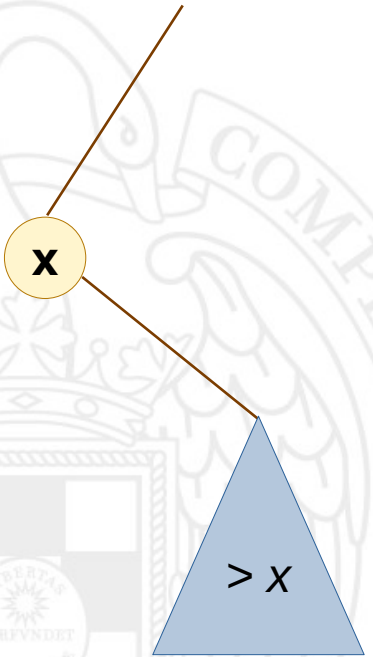
Eliminar el nodo más pequeño de un árbol

```
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.

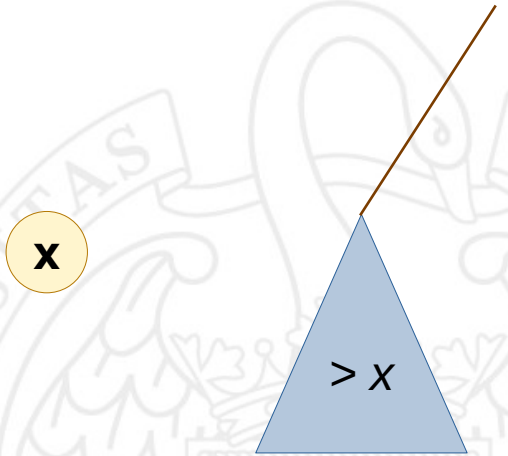
Eliminar el nodo más pequeño de un árbol

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




  }
}
```



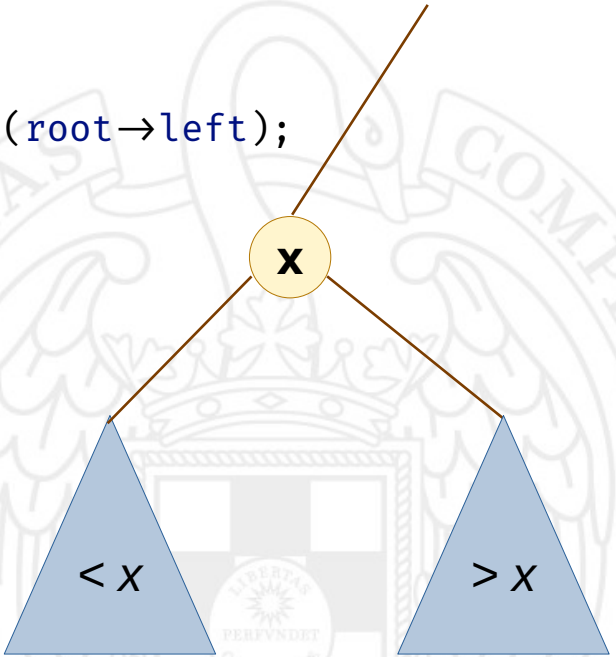
Eliminar el nodo más pequeño de un árbol

```
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)`

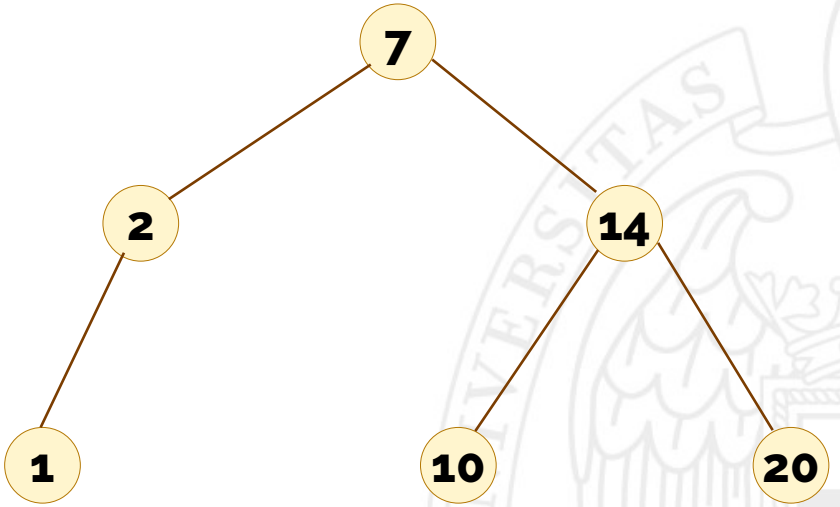
  Elimina la raíz del árbol, devolviendo la nueva raíz. Si la raíz tiene dos hijos, la nueva raíz es el nodo con el menor valor del hijo derecho. Se llama a `remove_lowest` para obtener este último nodo.

- `remove_lowest(root)`

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

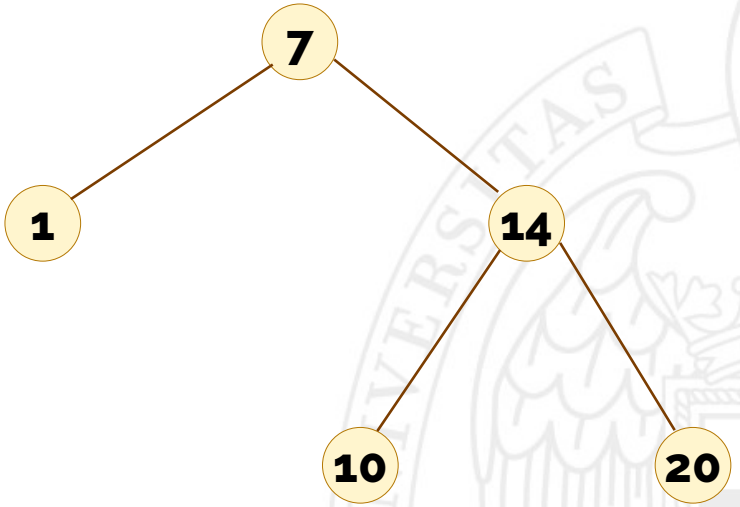
Ejemplo

- Eliminar el valor **2**



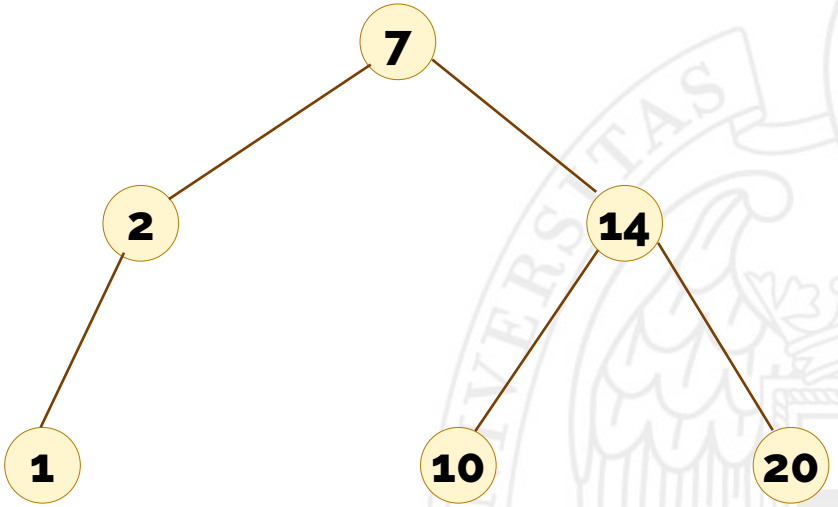
Ejemplo

- Eliminar el valor **2**



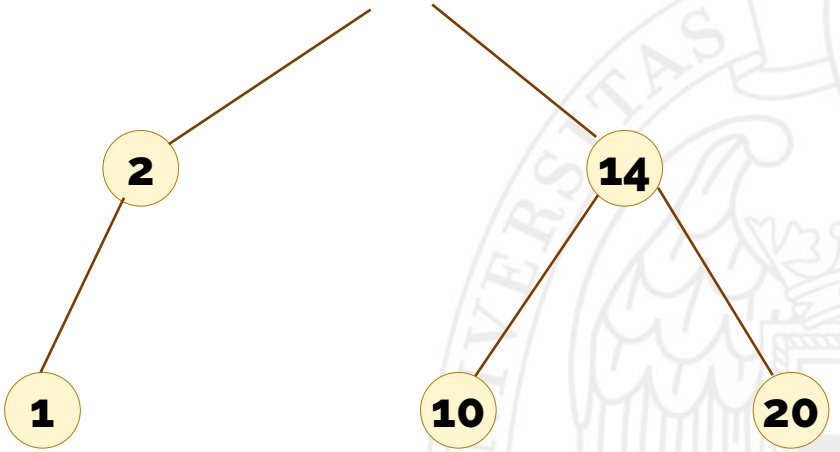
Ejemplo

- Eliminar el valor **7**



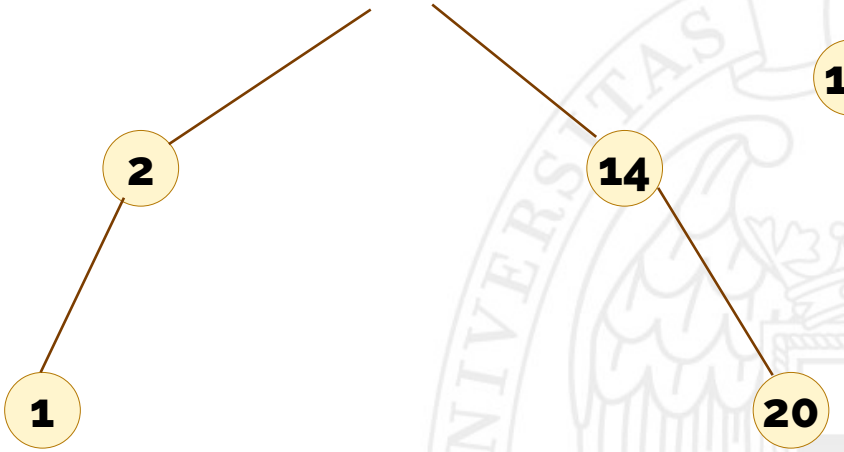
Ejemplo

- Eliminar el valor **7**



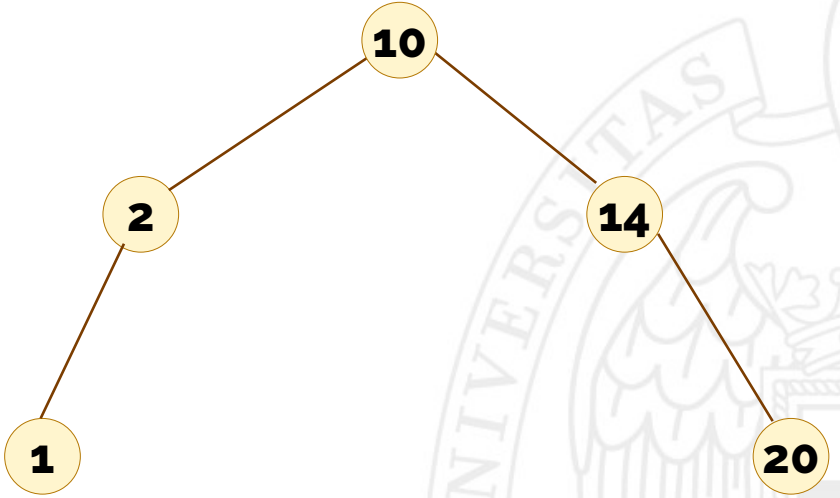
Ejemplo

- 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.