

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

Podemos añadir iteradores en los árboles binarios y para ello nos podemos fijar en la versión iterativa de la versión en inorder de un árbol binario

```
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;  
    descend_and_push(node, st);
```

```
    auto it = tree.begin();
```

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

```
    visit(*it);
```

```
    ++it;
```

Implementar este bucle y que se mantenga lo que hace la implementación de arriba.

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

visitar el elemento apuntado por el iterador.


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;
```

```
};
```

constructores privados para que solo se puedan crear con begin y end()

pila de nodos. Es la misma que utilizamos en la versión iterativa del inorden.

```
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;  
};
```

obtener el elemento que está en la cima de la pila.

```
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);  
    }
```

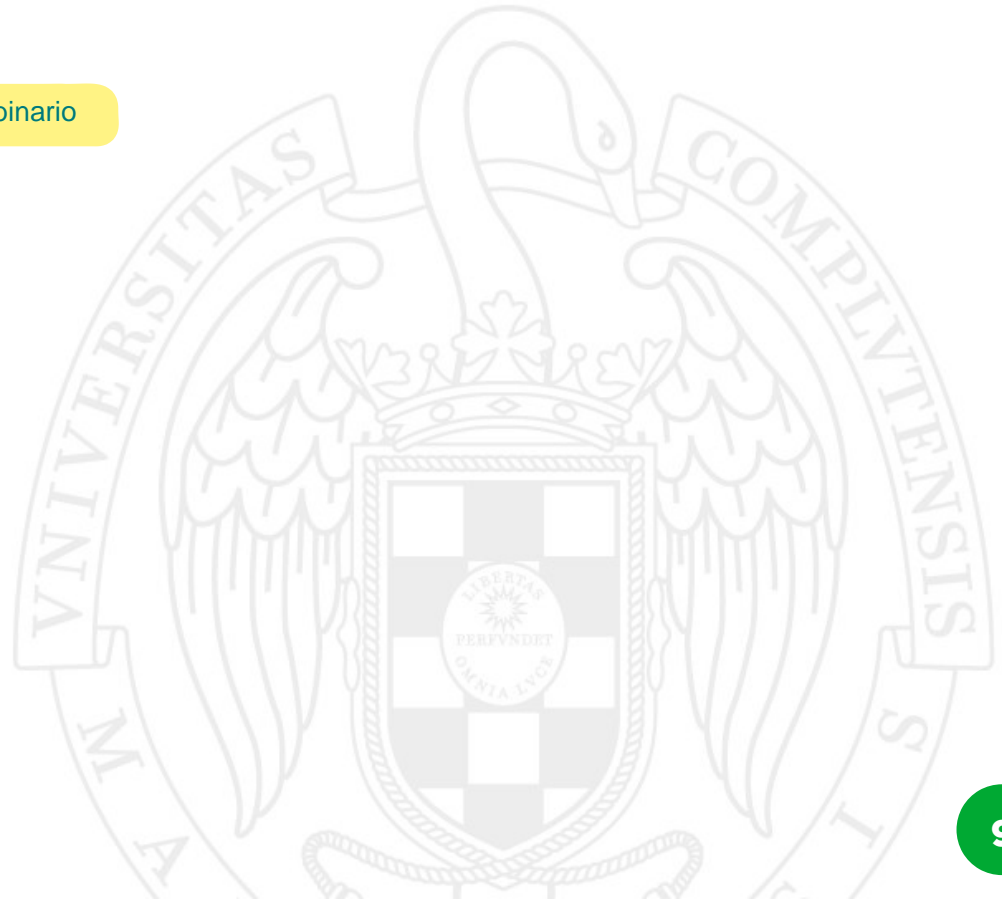
nodo raíz de mi árbol binario

```
    iterator end() {  
        return iterator();  
    }
```

Iterador con la pila vacía.

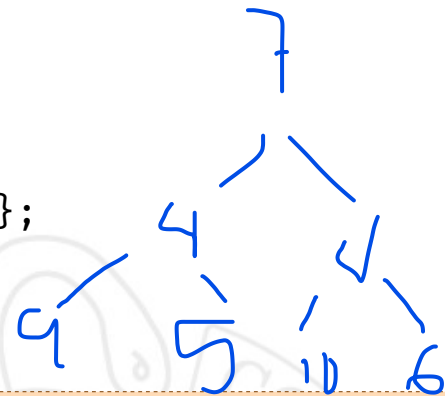
...

```
};
```



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

también se puede aplicar de esta manera

Posibles extensiones

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

