

Implementación de conjuntos sobre ABB en C++

Algoritmos y Estructuras de Datos II

1.^{er} cuatrimestre de 2018

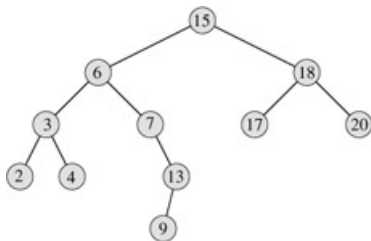
Introducción

- ▶ Vamos a implementar una interfaz de conjunto en C++
- ▶ La representación interna consistirá en un árbol binario de búsqueda (ABB)
- ▶ Utilizaremos memoria dinámica

Árboles binarios de búsqueda (ABB)

Un árbol binario es un ABB si es nil o satisface todas las siguientes condiciones:

- ▶ Todos los nodos del subárbol izquierdo son menores que la raíz.
- ▶ Todos los nodos del subárbol derecho son mayores que la raíz.
- ▶ Los subárboles izquierdo y derecho son ABBs.



Implementación en C++

- ▶ Vamos a implementar una clase `Conjunto<T>` paramétrica en un tipo `T` con un orden total estricto $<$
- ▶ Primero plantearemos el esquema de la clase
- ▶ Luego la parte pública (interfaz)
- ▶ Luego la parte privada (representación y operaciones auxiliares)
- ▶ Por último, la implementación de los métodos

Interfaz

- ▶ Queremos dotar a nuestra clase de una interfaz de conjunto
- ▶ ¿Qué operaciones serán visibles para el usuario?
En particular, para el taller, nos conformamos con:
 - ▶ Crear un conjunto nuevo (vacío)
 - ▶ Insertar un elemento
 - ▶ Decidir si un elemento pertenece al conjunto
 - ▶ Remover un elemento
 - ▶ Obtener la cantidad de elementos
 - ▶ Mostrar los elementos
- ▶ ¿Alguna otra operación que podría resultar útil? (dado que **T** tiene orden total estricto)
 - ▶ Obtener el mínimo
 - ▶ Obtener el máximo
 - ▶ Obtener el elemento siguiente a otro dado

Interfaz

```
template <class T>
class Conjunto {
    public:
        Conjunto();
        void insertar(const T&);
        bool pertenece(const T&) const;
        void remover(const T&);
        const T& minimo() const;
        const T& maximo() const;
        unsigned int cardinal() const;
        void mostrar(std::ostream&) const;
        const T& siguiente(const T&) const;
    private :
        /*...*/
};
```

- ¿Por qué mínimo y máximo devuelven Const T?

Representación de los nodos

- ▶ Definimos una estructura `Nodo` para representar los nodos del ABB
- ▶ La estructura estará en la parte privada de la clase ABB (no queremos exportarla)
- ▶ La estructura va a contener un valor del tipo `T` y dos punteros: uno al hijo izquierdo y el otro al hijo derecho
- ▶ La estructura tendrá un constructor que recibirá el valor de tipo `T` como único argumento e inicializará los dos punteros a `NULL`

Representación de los nodos

```
private:
    struct Nodo {
        T valor;
        Nodo* izq;
        Nodo* der;
        Nodo(const T& v) :
            valor(v), izq(NULL), der(NULL) {
        }
    };
    /*...*/
```


Representación de los nodos

```
private:
    struct Nodo {
        T valor;
        Nodo* izq;
        Nodo* der;
        Nodo(const T& v) :
            valor(v), izq(NULL), der(NULL) {
        }
    };
    Nodo* raiz;
```

`raiz` es la única variable de instancia y apunta al nodo raíz del ABB, o es `NULL` si el ABB no tiene nodos

Representación de los nodos

¿En qué se diferencia con la estructura de la lista doblemente enlazada?

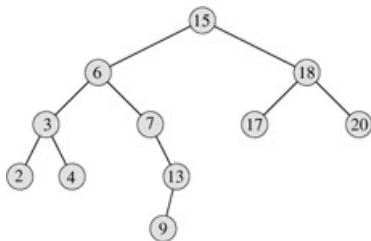
```
private:
    struct Nodo {
        T valor;
        Nodo* prev;
        Nodo* sig;
        Nodo(const T& v) :
            valor(v), prev(NULL), sig(NULL) {
        }
    };
    Nodo* cab;
```

¿Representan lo mismo? ¿Se comportan igual?

Los diferencia el invariante de representación (rep)

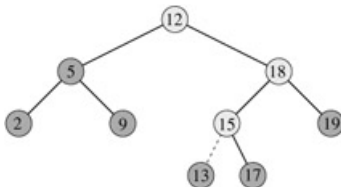
Pertenencia de un elemento

- ▶ Empezamos en la raíz, si existe, si no devolver False
- ▶ Si el elemento está en la raíz, devolvemos True
- ▶ Si no, decidimos en qué nodo continuar en base a $<$ (gracias al *invariante de representación* de los ABB).
 - ▶ Consideramos a este nodo como la raíz del subárbol correspondiente y repetimos.



Insertar un elemento

- ▶ Buscamos en qué lugar del árbol debe ir la nueva clave
- ▶ Para ello hacemos una búsqueda de la clave en el árbol
- ▶ Si la búsqueda es exitosa, la clave ya pertenece al conjunto y no hacemos nada
- ▶ Si la búsqueda fracasa, se debe insertar un nuevo nodo como hijo del último nodo de la búsqueda



Borrar un elemento

- ▶ Buscamos el nodo que tenemos que borrar.
- ▶ Tenemos 3 casos:
 - ▶ El nodo que tenemos que borrar es una hoja → Lo borramos.
 - ▶ El nodo que tenemos que borrar tiene un solo hijo → El hijo pasa a ocupar el lugar del padre.
 - ▶ El nodo que tenemos que borrar tiene dos hijos.
 - ▶ ¿Qué nodos pueden ocupar su lugar?
 - ▶ El inmediato sucesor. ¿Dónde está?
 - ▶ El inmediato predecesor. ¿Dónde está?

Discusión

► ¿Qué complejidad tienen las siguientes operaciones?

- Pertenece $\rightarrow \mathcal{O}(N)$
- Insertar $\rightarrow \mathcal{O}(N)$
- Borrar $\rightarrow \mathcal{O}(N)$
- Mínimo/Máximo $\rightarrow \mathcal{O}(N) / \mathcal{O}(1)$

donde N es la cantidad de elementos que tiene el conjunto.

¡Ojo! ¡No depende sólo de la estructura en este caso!

¡Depende de si los datos fueron ingresados de manera uniforme o no!

Iteración

Problema: Dar un algoritmo para recorrer todos los nodos de un árbol ...

- ▶ ... en tiempo lineal (i.e. en $\mathcal{O}(n)$)
- ▶ ... iterativo
 - ▶ ¿Por qué, si ya conocemos recorridos recursivos?
Para (después) poder implementar iteradores sobre árboles.

Dado que los nodos no tienen un puntero a su padre, necesitamos saber *a dónde subir* para no tener que repetir recorridos innecesarios.

- ▶ Para eso usamos una estructura de datos auxiliar: por ejemplo, una pila.
- ▶ Elijamos una forma de recorrer el árbol: por ejemplo, *InOrder*.

InOrder

- ▶ Repaso: $\text{inorder}(\text{Bin}(i, r, d)) \equiv \text{inorder}(i) \ \& \ \langle r \rangle \ \& \ \text{inorder}(d)$,
con lo cual, si el árbol es un ABB, esto está ordenado.
- ▶ Observación: el primer elemento que tenemos que devolver es el mínimo. Y ya sabemos cómo encontrarlo: yendo siempre hacia la izquierda.

Tenemos que hallar el sucesor del mínimo.

Sucesor de un elemento

- ▶ Buscamos el nodo en el árbol que tiene el elemento del que buscamos el sucesor.
- ▶ (caso A) Si el nodo tiene un subárbol derecho, devolvemos el mínimo elemento de dicho subárbol.
- ▶ (caso B) Si el nodo no tiene subárbol derecho, hay que subir en el árbol:
 - ▶ (caso B.1) Si el nodo es un hijo izquierdo, devolvemos el elemento del padre.
 - ▶ (caso B.2) Si el nodo es un hijo derecho, subimos en el árbol hasta llegar a un nodo por su rama izquierda, y devolvemos ese elemento.

Sucesor de un elemento, según el Cormen

TREE-SUCCESSOR(x)

```
1  if  $right[x] \neq \text{NIL}$ 
2      then return TREE-MINIMUM( $right[x]$ )
3   $y \leftarrow p[x]$ 
4  while  $y \neq \text{NIL}$  and  $x = right[y]$ 
5      do  $x \leftarrow y$ 
6           $y \leftarrow p[y]$ 
7  return  $y$ 
```

Pueden encontrar los algoritmos para árboles binarios de búsqueda (BST en inglés) en el capítulo 12 del Cormen.

¡A programar!

En `Conjunto.hpp` está la declaración de la clase, su parte pública y la definición de `Nodo`.