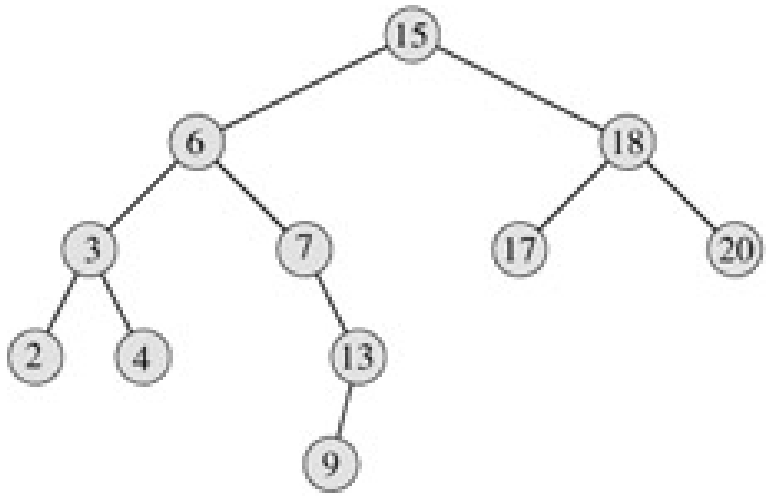
Introducción

- 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:

- la raíz es mayor que todos los nodos del subárbol izquierdo
- la raíz es menor que todos los nodos del subárbol derecho y
- 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 fcs. auxiliares)
- Por último, la implementación de los métodos

Esquema de la clase Conjunto<T>

```
#ifndef _CONJUNTO__H_
#define _CONJUNTO__H_

template <class T>
class Conjunto {
    public:
        /*...*/
    private:
        /*...*/
};

/* ... */
#endif
```

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

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;
    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?

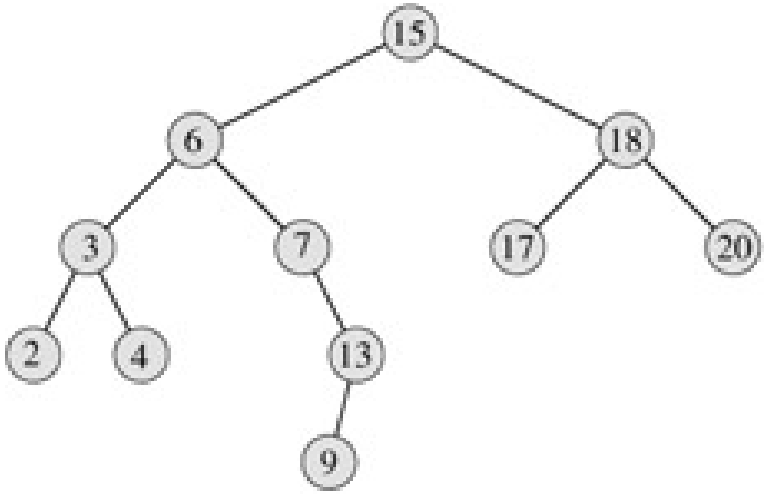
```
private:
    struct Nodo {
        T valor;
        Nodo* prev;
        Nodo* sig;
        Nodo(const T& v) :
          valor(v), prev(NULL), sig(NULL) {
        }
    };
    Nodo* cab;
```

¿Representan lo mismo? ¿Se comportan igual?

Los diferencia el invariante de representación (`rep`)

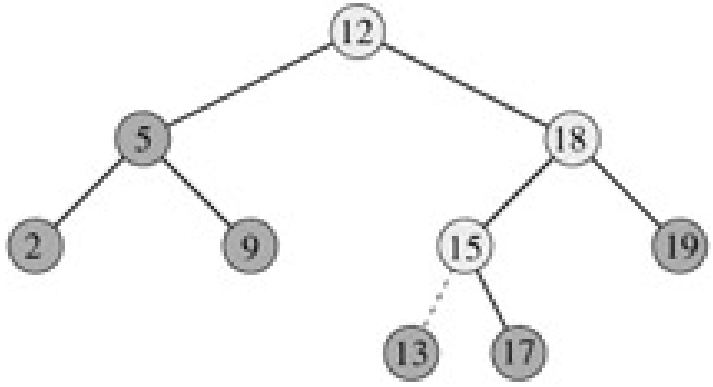
Pertenencia de un elemento

- 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



¡A programar!

En `Conjunto.hpp` está la declaración de la clase, su parte pública y la definición de `Nodo`. Completen (¡y prueben!) el constructor por defecto, `pertenece`, `insertar`, `mínimo` y `máximo`. Volvemos en una hora (o cuando terminen todos, lo que primero suceda).

Pertenece: Versión iterativa

```
template <class T>
bool Conjunto<T>::pertenece(
  const T& clave) const {
    Nodo* x = this->raiz;
    while (x != NULL && x->valor != clave)
        if (clave < x->valor)
            x = x->izq;
        else
            x = x->der;
    }
    return x != NULL;
}
```

Pertenece: Versión recursiva

También podemos implementarlo de manera recursiva. Para eso declaramos un método auxiliar:

```
private:
    /* ... */
    static bool pertenece_rec(Nodo*, const T&);
    /* ... */
```

Usamos el modicador `static` en la declaración. Recordemos que esto hace que el método no posea el parámetro implícito `this`. Veremos que efectivamente no lo necesitamos.

La implementación de `pertenece` nos queda así:

```
template <class T>
bool Conjunto<T>::pertenece(const T& clave) cons
    return pertenece_rec(this->raiz, clave);
}
```

Pertenece: Versión recursiva

```
template <class T>
bool Conjunto<T>::pertenece_rec(
  Nodo* x, const T& clave) {
    if (x != NULL && x->valor != clave) {
        if (clave < x->valor)
            return pertenece_rec(x->izq, clave);
        else
            return pertenece_rec(x->der, clave);
    } else
        return x != NULL;
}
```

Insertar un elemento: Búsqueda

```
template <class T>
void Conjunto<T>::insertar(const T& clave) {
    Nodo* y = NULL;
    Nodo* x = this->raiz;
    while (x != NULL && x->valor != clave) {
        y = x;
        if (clave < x->valor)
            x = x->izq;
        else
            x = x->der;
    }
    /* ... */
}
```

El puntero `x` recorre el árbol hasta encontrar la `clave` o volverse `NULL`.

El puntero `y` apunta al padre de `x` o a `NULL` si `x == this->raiz`.

Insertar un elemento: Inserción

```
template <class T>
void Conjunto<T>::insertar(const T& clave) {
    /* ... */
    if (x == NULL) {
        Nodo* z = new Nodo(clave);
        if (y == NULL)
            this->raiz = z;
        else if (clave < y->valor)
            y->izq = z;
        else
            y->der = z;
    }
}
```

Si la búsqueda de `clave` fracasa (es decir, `x == NULL`) se inserta la `clave` en el árbol, ya sea en la raíz (si `y == NULL`) o como hijo de `y`.

Insertar un elemento: Versión recursiva

También podemos implementarlo de manera recursiva. Para eso declaramos un método auxiliar:

```
private:
    /* ... */
    void insertar_rec(Nodo*, Nodo*, const T&);
    /* ... */
```

(En este caso no utilizamos el modicador `static` en la declaración. Notar que en la implementación usamos el parámetro implícito `this`.)

La implementación de `insertar` nos queda así:

```
template <class T>
void Conjunto<T>::insertar(const T& clave) {
    insertar_rec(NULL, this->raiz, clave);
}
```

Insertar un elemento: Versión recursiva

```
template <class T>
void Conjunto<T>::insertar_rec(
  Nodo* y, Nodo* x, const T& clave) {
    if (x != NULL && x->valor != clave) {
        if (clave < x->valor)
          insertar_rec(x, x->izq, clave);
        else
          insertar_rec(x, x->der, clave);
    } else if (x == NULL) {
        Nodo* z = new Nodo(clave);
        if (y == NULL)
            this->raiz = z;
        else if (clave < y->valor)
            y->izq = z;
        else
            y->der = z;
    }
}
```

Discusión

- ¿Qué complejidad tienen las siguientes operaciones?
  - Insertar $\rightarrow \mathcal{O}(N)$
  - Pertenece $\rightarrow \mathcal{O}(N)$
  - 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!

¿Afecta en algo que el tipo T tenga orden total ($<$)? ¿Y si fuese un AVL? ¿Cambia algo?

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 no 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: `inorder(Bin(i, r, d))` $\equiv$ `inorder(i) & <r> & 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.
- Observación': el siguiente elemento que tenemos que devolver es el que le sigue al mínimo.
  - Si el nodo tiene hijo derecho, será el mínimo de ese subárbol.
  - Si no, será el padre del mínimo.
- Idea (para imprimr los nodos en orden): Bajar siempre hacia la izquierda. Cuando esto no sea posible, imprimir el último nodo al que llegamos. Continuar haciendo lo mismo sobre el subárbol derecho si existe y si no, subir imprimiendo cada nodo por el que pasamos hasta encontrar alguno con hijo derecho.

Ejemplos y Borrador

En el pizarrón.

¡A programar! (bis)

Completar la implementación

- Agregarle al módulo de lista interfaz de pila, es decir, las operaciones `push` (`apilar`) y `pop` (`desapilar`) .
- Implementar `mostrar` y `cardinal`.
- No debemos perder memoria, por lo tanto no se olviden del destructor.
- De tarea: implementar `remover`. Tienen que considerar 3 casos. Pueden ayudarse con las transparencias de la clase teórica.
- Para pensar: ¿Cómo cambia el código para recorrer el árbol en *PreOrder*? ¿Y en *PostOrder*? ¿Y si en vez de usar una pila usáramos una cola, qué recorrido podríamos obtener?

InOrder - Solución

```
template <class T>
void Conjunto<T>::mostrar(std::ostream& os) const {
    Pila<Nodo*> pila;
    Nodo* sig = this->raiz;
    do {
        if (sig != NULL) {
            pila.push(sig);
            sig = sig -> izq;
        } else {
            sig = pila.pop();
            os << sig -> valor << " ";
            sig = sig -> der;
        }
    } while (!pila.vacia() || sig != NULL);
    os << endl;
}
```