Implementación de conjuntos sobre ABB en `C++`

Algoritmos y Estructuras de Datos II

2.do cuatrimestre de 2019

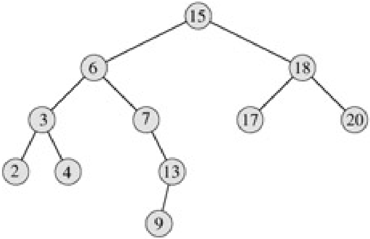
Introducción

- 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`)

Irep

- El nodo raiz es null

Irep

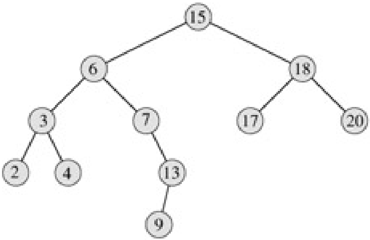
- El nodo raiz es null
- O bien
- Todos los nodos son alcanzables y
- No tiene ciclos y
- Todos los valores a la izq de la raiz son menores al valor en la raiz y
- Todos los valores a la der de la raiz son mayores al valor en la raiz

Irep

- El nodo raiz es null
- O bien
- Todos los nodos son alcanzables y
- No tiene ciclos y
- Todos los valores a la izq de la raiz son menores al valor en la raiz y
- Todos los valores a la der de la raiz son mayores al valor en la raiz y
- *Se cumple recursivamente el Irep para los dos subárboles*

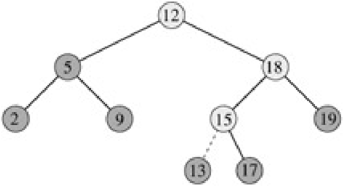
Pertenencia de un elemento

- 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 $\rightarrow$ Lo borramos.
  - El nodo que tenemos que borrar tiene un solo hijo $\rightarrow$ 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.

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.

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)
if right[x] ≠ NIL
    then return TREE-MINIMUM(right[x])
y ← p[x]
while y ≠ NIL and x = right[y]
      do x ← y
         y ← p[y]
return y
```

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

Algunas alternativas posibles para no tener que conocer el sucesor(...)

- (1) una pila.
- (2) Tener la cantidad de nodos precalculadas en el nodo.

https://www.geeksforgeeks.org/inorder-tree-traversal-without-recursion/

Inorden con Pila (1)

- (1) Crear una pila S.
- (2) Inicializar el nodo actual a la raiz.
- (3) Apilar el nodo actual y mover actual a la izquierda(actual = actual->izq) hasta que actual sea null.
- (4) Si actual no es null y S no esta vacio
  - (a) Imprimir la cima(c) de S.
  - (b) actual = c->der.
  - (b) volver al paso 3
- (5) si actual es null y S esta vacio, terminamos.

Inorden con Pila: Implementacion

```
void inOrder(struct Node *root) {
    stack<Node *> s;  Node *actual = root;
    while (actual != NULL || s.empty() == false)  {
        /* buscamos el nodo mas la izq */
        while (actual !=  NULL)  {
            /* apilamos antes de mover! */
            s.push(actual);
            actual = actual->izq;
        }
        actual = s.top();  /* actual deberia ser NULL  */
        s.pop();
        cout << actual->data << " ";
        /* ahora reccorremos el subarbol derecho! */
        actual = actual->der;
    }
}
```

Inorden con Pila: Ejemplo

```
        Abb b;
b.agregar(3);
b.agregar(1);
b.agregar(4);
b.agregar(0);
b.agregar(2);
b.inorden();
```

Inorden con Pila: Ejemplo

```
s = [3];
s = [3,1];
s = [3,1,0];
```

Inorden con Pila: Ejemplo

```
s = [3,1];
0;
s = [3];
1;
```

Inorden con Pila: Ejemplo

```
s = [3,2];
2;
s = [3];
3;
s = [];
```

Inorden con Pila: Ejemplo

```
s = [4];
4;
s = [];
```

Inorden con la cantidad de nodos precalculada (2)

```
private:
    struct Nodo {
        T valor;
        Nodo* izq;
        Nodo* der;
        int cant;
        Nodo(const T& v) :
          valor(v), izq(NULL), der(NULL), cant(0){
        }
    };
    /*...*/
```

Las hojas tienen 0 hijos

Inorden con la cantidad de nodos precalculada (2)

- La idea es:
- Actualizar cant al agregar/eliminar nodos en O(1)
- En inorden devolver un vector de nodos ordenados
  - La posicion se puede calculcar con la cantidad de nodos de cada subarbol en O(1)
  - El vector puede llenarse en O(n)

Inorden con cant: Implementación

```
void inOrder(vector<T>& v) {
        /* considero los nodas anteriores */
        int indice = cantIzq();
        v[indice] = info;

        if (izq != null) izq.inOrder(v);
        if (der != null) der.inOrder(v);
}
```

Inorden con cant: Implementación

```
void inOrder(vector<T>& v, int cantAnt) {
/* considero los nodas anteriores y los acumulados! */
        int indice = cantIzq() + cantAnt;
        v[indice] = info;

        if (izq != null) izq.inOrder(v, cantAnt);
        /* +1 me cuenta a mi mismo! */
        if (der != null) der.inOrder(v, indice + 1);
}
```

- Demotramos O(n) y correctitud por inducción en cantidad de nodos!
- En el caso iterativo -se puede- pero es mas dificil!

Destructor

¿Se puede aplicar la misma estregia para el destructor?

Destructor

```
template <class T>
void ABB<T>::destruir(nodo * n)    {
      if (n!=null)       {
       destruir(n->izq);
       delete n;
       destruir(n->der);
       }
}
```

Destructor

```
template <class T>
void ABB<T>::destruir(nodo * n)    {
    if (n!=null)      {
       destruir(n->izq);
       delete n;
       // ¡Cuidado,  n ya no existe!
       destruir(n->der);
       }
}
```

Destructor

```
template <class T>
void ABB<T>::destruir(nodo * n)    {
    if (n!=null)       {
       destruir(n->izq);
       destruir(n->der);
       delete n;
       }
}
```

¡A programar!

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