

ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS ARBORESCENTES

Implementación del TAD Conjunto mediante listas ordenadas

Manuel Montenegro Montes
Departamento de Sistemas Informáticos y Computación
Facultad de Informática – Universidad Complutense de Madrid

Operaciones en el TAD Conjunto

Nos olvidamos de los árboles binarios de búsqueda y nos centramos en las listas ORDENADAS

- Constructoras:

Recordatorio

- Crear un conjunto vacío: **create_empty**

NO ES LA IMPLEMENTACIÓN MÁS IMPORTANTE. PERO NOS VA A SERVIR PARA CONOCER FUNCIONES DE LA LIBRERÍA DE C++

- Mutadoras:

- Añadir un elemento al conjunto: **insert**
- Eliminar un elemento del conjunto: **erase**

En realidad se utilizan los árboles binarios de búsqueda, estos no. Los explicó y dijo que los viéramos opcionalmente.

- Observadoras:

- Averiguar si un elemento está en el conjunto: **contains**
- Saber si el conjunto está vacío: **empty**
- Saber el tamaño del conjunto: **size**

Representación mediante listas ordenadas

- Clase que contiene un único atributo: `list_elems`, de tipo Lista.
- El atributo `list_elems` contiene los elementos del conjunto que se quiere representar de modo que:
 - `list_elems` almacena los elementos en orden ascendente.
 - `list_elems` no almacena duplicados.

Da igual si usar array o nodos.

$\{4, 5, 7, 3, 1\}$

`list_elems: [1, 3, 4, 5, 7]`

Representación mediante listas ordenadas

- Clase que contiene un único atributo: `list_elems`, de tipo Lista.
- El atributo `list_elems` contiene los elementos del conjunto que se quiere representar de modo que:
 - `list_elems` almacena los elementos en orden ascendente.
 - `list_elems` no almacena duplicados.

```
template <typename T>
class SetList {
public:
    ...
private:
    using List = ???;
    List list_elems;
};
```

`std::vector<T>` como si fuera un array
`std::list<T>` listas doblemente enlazadas.

Representación mediante listas ordenadas

- **Función de abstracción:**

Si s es una instancia de la clase SetList:

$$f(s) = \{ s.\text{list_elems}[i] \mid 0 \leq i < s.\text{list_elems}.\text{size}() \}$$

- **Invariante de representación**

$$I(s) \equiv \forall i, j: \quad 0 \leq i < j < s.\text{list_elems}.\text{size}() \\ \implies \underline{s.\text{list_elems}[i] < s.\text{list_elems}[j]}$$

Quiere decir que la lista está ordenada.

NO hay elementos repetidos.

Operaciones constructoras

```
template <typename T>
```

```
class SetList {
```

```
public:
```

X `SetList() { }` el atributo `list_elems` quedaría inicializado a una lista vacía.

`SetList(const SetList &other): list_elems(other.list_elems) { }` constructor de copia. Copia la lista de los elementos de la lista que le pasamos por parámetro a otra lista.

`~SetList() { }` Destructor llama al destructor de todos los atributos

```
private:
```

```
...
```

```
List list_elems;
```

```
};
```

NO HARÍA FALTA PONER NINGUNO PORQUE C++ PONDRÍA POR DEFECTO TODOS ELLOS AL SER CONSTRUCTORES BÁSICOS POR ASÍ DECIRLO.

Operaciones observadoras

```
template <typename T>
class SetList {
public:
    ...
    bool contains(const T &elem) const { ... }

    int size() const {
        return list_elems.size();
    }

    bool empty() const {;
        return list_elems.empty();
    }

private:
    ...
    List list_elems;
};
```



Operación *contains*

Para ver si está en la lista debemos hacer una búsqueda binaria, para que sea más eficiente.

- Utilizamos una función de búsqueda binaria

`bool binary_search(iterator first, iterator last, const T& val)`

Elemento que queremos buscar.

definida en `<algorithm>`

No lo implementamos nosotros porque sería una pérdida de tiempo, ya está implementada en las librerías de c++.

dos iteradores que contienen el inicio y el fin de la secuencia en la que queremos buscar.

```
template <typename T>
class SetList {
public:
```

```
    bool contains(const T &elem) const {
        return std::binary_search(list_elems.begin(), list_elems.end(), elem);
    }
```

```
    ...
};
```


Operaciones mutadoras

```
template <typename T>
class SetList {
public:
    ...
    void insert(const T &elem) { ... }
    void erase(const T &elem) { ... }

private:
    ...
    List list_elems;
};
```



Operación *insert*

- Necesitamos insertar el elemento en `list_elems` de modo que la lista permanezca ordenada.
- Podemos utilizar búsqueda binaria para saber dónde insertar el elemento.
- Problema: `binary_search` solamente indica si un elemento está en la lista o no. Pero la función `binary_search` NO nos dice cual es la posición del elemento. Nos dice si elemento está o no en una lista.

- Pero tenemos la función `lower_bound`:

Está en `#include <algorithm>`

```
iterator lower_bound(iterator begin, iterator end, const T &elem)
```

Devuelve un iterador al primer elemento contenido entre `begin` y `end` que no es estrictamente menor que `elem`.

Ordenamos elementos de menor a mayor

Si todos son menores que `elem`, devuelve `end`.

Los elementos que hay entre `begin` y `end` han de estar ordenados.

Ejemplo: lower_bound

```
std::vector<int> v = {1, 5, 8, 10, 20};  
auto it_pos = std::lower_bound(v.begin(), v.end(), 9);  
std::cout << *it_pos << std::endl;
```

En este caso, el iterador va a devolver un diez porque es el primer número NO menor que 9

Operación *insert*

```
template <typename T>
class SetList {
public:
```

```
...
void insert(const T &elem) {
    auto position = std::lower_bound(list_elems.begin(), list_elems.end(), elem);
```

inserta en orden

```
    if (position == list_elems.end() || *position != elem) {
        list_elems.insert(position, elem);
    }
}
```

Comprobaciones: Si iterador es end y comprobación para que no haya duplicados

insertamos el elemento elem justo antes del elemento que me ha devuelto lower_bound

```
private:
```

```
...
List list_elems;
};
```

Operación *erase*

```
template <typename T>
class SetList {
public:
```

```
...
void erase(const T &elem) {
    auto position = std::lower_bound(list_elems.begin(), list_elems.end(), elem);
```

```
    if (position != list_elems.end() && *position == elem) {
        list_elems.erase(position);
    }
}
```

Recordamos que esto no apunta a ningún elemento en concreto.


Tiene que ser distinto de end porque ahí no hay ningún elemento y además tiene que ser igual a algún elemento de la lista.

```
private:
```

```
...
List list_elems;
};
```

¿Qué utilizo?

```
template <typename T>
class SetList {
public:
    ...
private:
    using List = ???
    List list_elems;
};
```



Dependiendo del coste de las operaciones que queramos utilizar.

Coste de las operaciones auxiliares

Operación	std::vector	std::list
binary_search	$O(\log n)$	$O(n)$ (no es búsq. binaria)
lower_bound	$O(\log n)$	$O(n)$ (no es búsq. binaria)
insert (en listas)	$O(n)$	$O(1)$
erase (en listas)	$O(n)$	$O(1)$

$n = \text{longitud de list_elems}$

Ya que en array tendríamos que desplazar los elementos a la izquierda, mientras que en enlazadas es mover punteros y es coste constante, igual para erase.

Coste de las operaciones

Operación	<code>std::vector</code>	<code>std::list</code>
<i>constructor</i>	$O(1)$	$O(1)$
<i>empty</i>	$O(1)$	$O(1)$
<i>size</i>	$O(1)$	$O(1)$
<i>contains</i>	$O(\log n)$	$O(n)$
<i>insert</i>	$O(\log n) + O(n)$	$O(n) + O(1)$
<i>erase</i>	$O(\log n) + O(n)$	$O(n) + O(1)$

el coste sería el máximo entre ambos, por tanto el coste lineal (es una regla)

PARA SIMPLIFICAR.

n = número de elementos del conjunto

Coste de las operaciones

Operación	std::vector	std::list
<i>constructor</i>	$O(1)$	$O(1)$
<i>empty</i>	$O(1)$	$O(1)$
<i>size</i>	$O(1)$	$O(1)$
<i>contains</i>	$O(\log n)$	$O(n)$
<i>insert</i>	$O(n)$	$O(n)$
<i>erase</i>	$O(n)$	$O(n)$

Esta operación marca la diferencia.

Utilizaremos la clase vector definida en el fichero de cabecera
`#include vector<T>` v siendo T el tipo de los elementos del conjunto.

n = número de elementos del conjunto

¿Qué utilizo?

```
template <typename T>
class SetList {
public:
    ...
private:
    using List = std::vector<T>;
    List list_elems;
};
```

ESTAS DAPOSITIVAS NO SON NECESARIAS DE APRENDER AL PIE DE LA LETRA.

No lo vamos a utilizar mucho. Vamos a utilizar bastante más los ÁRBOLES BINARIOS DE BÚSQUEDA.