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

- Constructoras:
  - Crear un conjunto vacío: ***create_empty***
- Mutadoras:
  - Añadir un elemento al conjunto: ***insert***
  - Eliminar un elemento del conjunto: ***erase***
- 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.

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

`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>
std::list<T>
```

Representación mediante listas ordenadas

- **Función de abstracción:**

  Si s es una instancia de la clase `SetList`:

  $f($`s`$)$ = `{ s.list_elems[i] | 0 ⩽ i < s.list_elems.size() }`

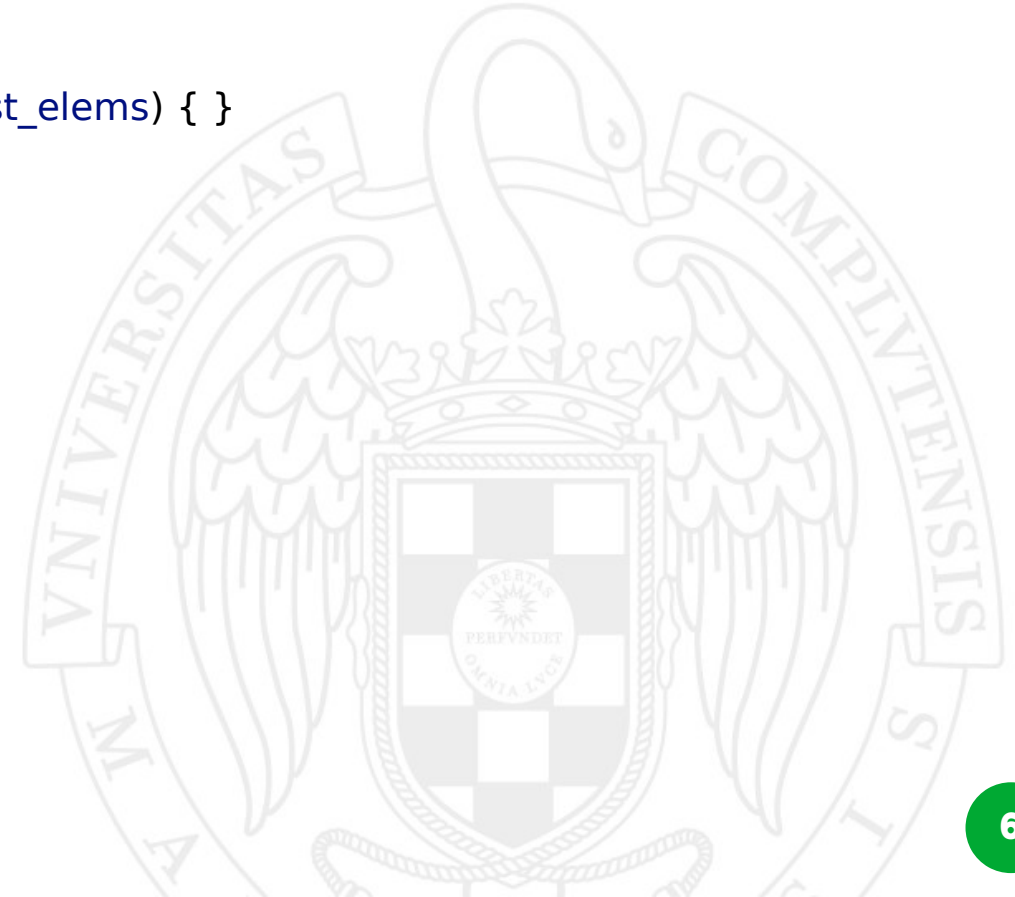
- **Invariante de representación**

  $I($`s`$) \equiv$ `∀i, j: 0 ⩽ i < j < s.list_elems.size()`

  $\Longrightarrow$ `s.list_elems[i] < s.list_elems[j]`

Operaciones constructoras

```
template <typename T>
class SetList {
public:
  SetList() { }
  SetList(const SetList &other): list_elems(other.list_elems) { }
  ~SetList() { }

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



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*

- Utilizamos una función de búsqueda binaria

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

  definida en `<algorithm>`

```
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 tenemos la función `lower_bound`:

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

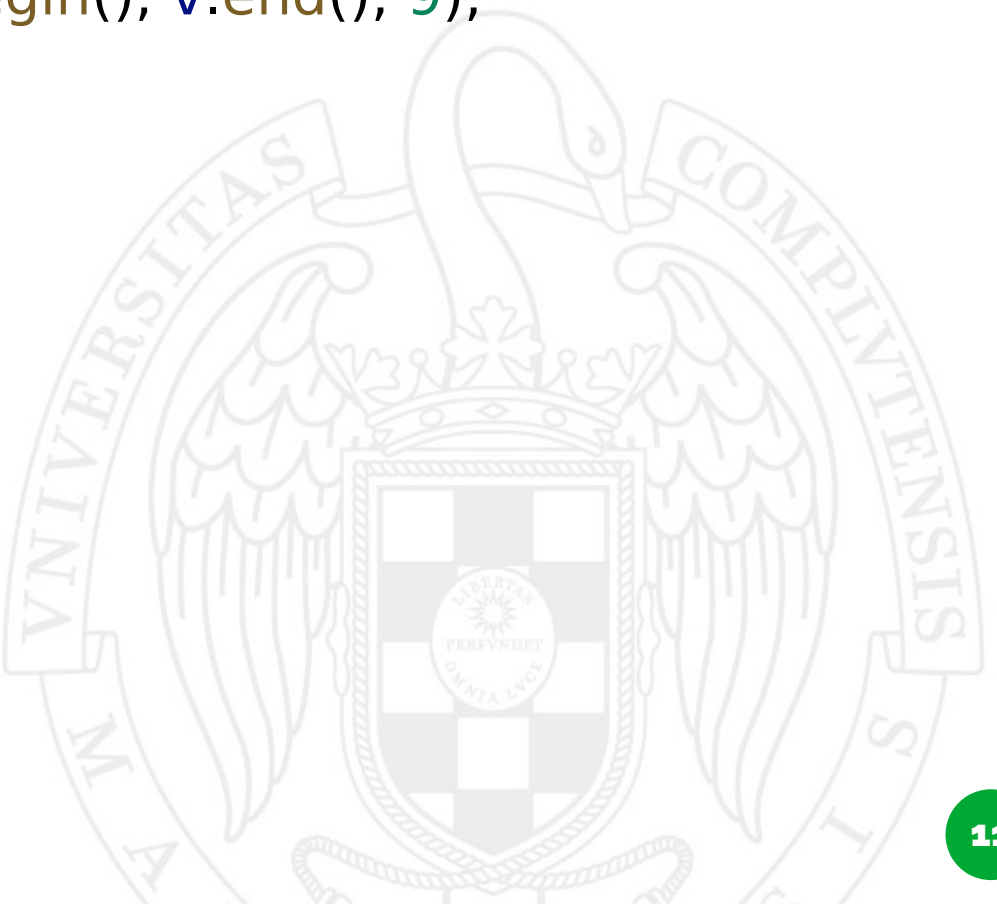
Devuelve un iterador al primer elemento contenido entre `begin` y `end` que no es estrictamente menor que `elem`.

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;
```

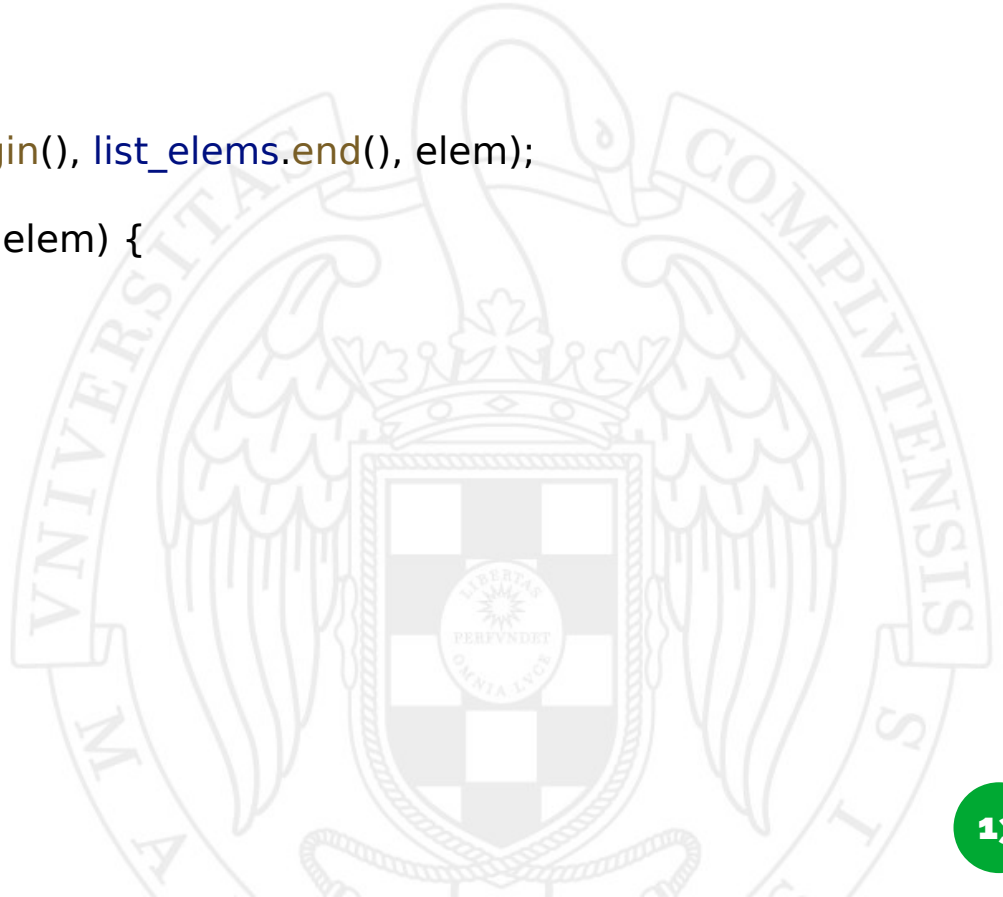


Operación *insert*

```
template <typename T>
class SetList {
public:
  ...
  void insert(const T &elem) {
    auto position = std::lower_bound(list_elems.begin(), list_elems.end(), elem);

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


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);
    }
  }

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



¿Qué utilizo?

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

```
std::vector<T>
std::list<T>
```

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 = longitud de list_elems

Coste de las operaciones

Operación	std::vector	std::list
constructor	$O(1)$	$O(1)$
`empty`	$O(1)$	$O(1)$
`size`	$O(1)$	$O(1)$
`contains`	$O(\log n)$	$O(n)$
`insert`	$O(\log n) + O(n)$	$O(n) + O(1)$
`erase`	$O(\log n) + O(n)$	$O(n) + O(1)$

n = número de elementos del conjunto

Coste de las operaciones

Operación	std::vector	std::list
constructor	$O(1)$	$O(1)$
`empty`	$O(1)$	$O(1)$
`size`	$O(1)$	$O(1)$
`contains`	$O(\log n)$	$O(n)$
`insert`	$O(n)$	$O(n)$
`erase`	$O(n)$	$O(n)$

n = número de elementos del conjunto

¿Qué utilizo?

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