ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS LINEALES

Implementación del TAD Lista mediante arrays

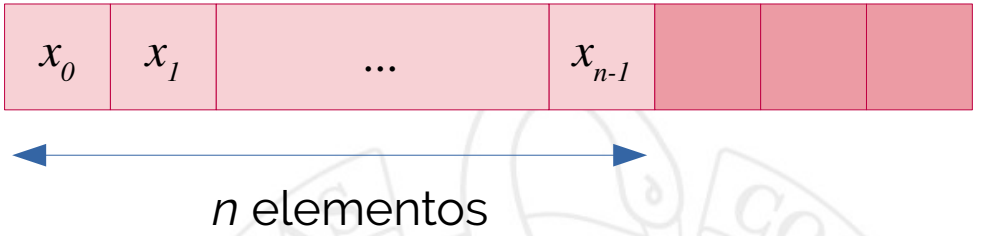
Manuel Montenegro Montes
Departamento de Sistemas Informáticos y Computación
Facultad de Informática – Universidad Complutense de Madrid

Recordatorio: operaciones del TAD Lista

- **Constructoras:**
  - Crear una lista vacía: ***create_empty***() → *L:List*
- **Mutadoras:**
  - Añadir un elemento al principio de la lista: ***push_front***(*x: elem, L: List*).
  - Añadir un elemento al final de la lista: ***push_back***(*x: elem, L: List).*
  - Eliminar el elemento del principio de la lista: ***pop_front***(*L: List).*
  - Eliminar el elemento del final de la lista: ***pop_back***(*L: List).*
- **Observadoras:**
  - Obtener el tamaño de la lista: ***size***(*L: List*) → *tam: int.*
  - Comprobar si la lista es vacía ***empty***(*L: List*) → *b: bool.*
  - Acceder al primer elemento de la lista ***front***(*L: List*) → *e: elem.*
  - Acceder al último elemento de la lista ***back***(*L: List*) → *e: elem.*
  - Acceder a un elemento que ocupa una posición determinada ***at***(*idx: int, L: List*) → *e: elem.*

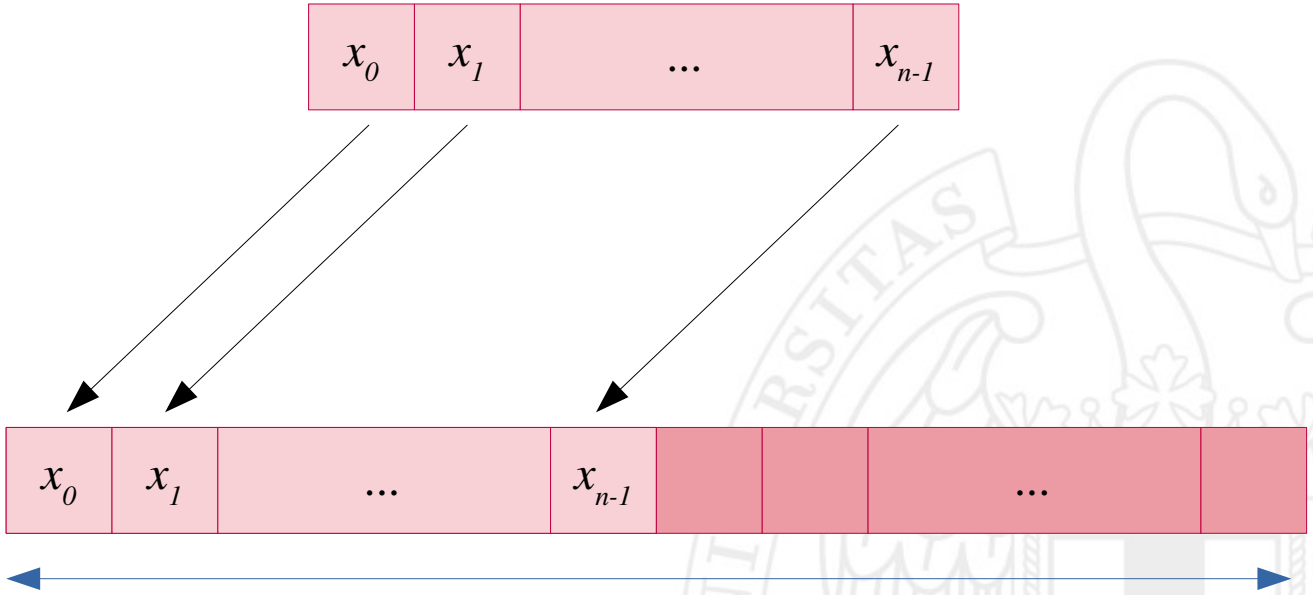
Implementación mediante arrays

$[\ x_0, x_1, \ldots, x_{n-1}\ ]$




```
class ListArray {
public:
    ...
private:
    int num_elems;
    std::string elems[MAX_CAPACITY];
};
```

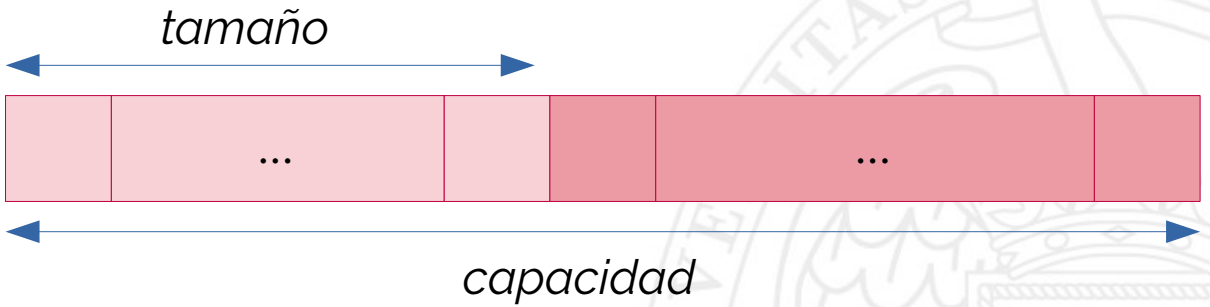
¿Y si el array se llena?



Nueva capacidad: $2 * n$

Tamaño vs. capacidad

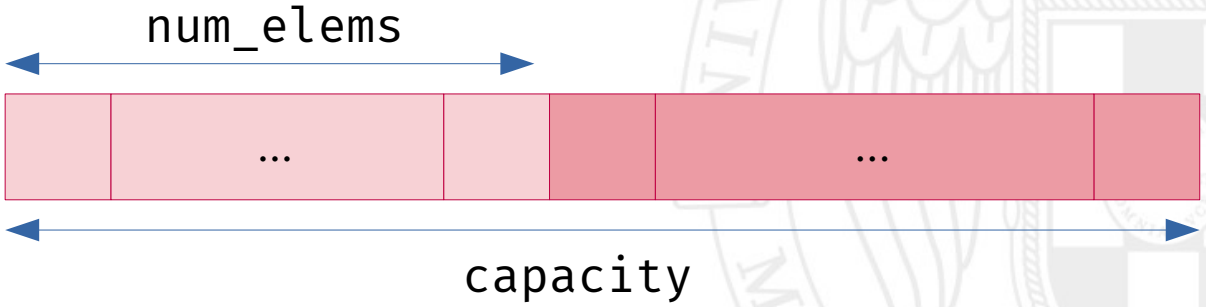
- **Capacidad de una lista:** Tamaño del array que contiene los elementos.
- **Tamaño de una lista:** Número de posiciones ocupadas por elementos.
  - Siempre se cumple *tamaño* ≤ *capacidad*.



Implementación con tamaño y capacidad

```
class ListArray {
public:
    …
private:
    int num_elems;
    int capacity;
    std::string *elems;
};
```

Array reservado dinámicamente



Invariante y modelo de representación

```
class ListArray {
public:
 …
private:
  int num_elems;
  int capacity;
  std::string *elems;
};
```

- Invariante de la representación:

  $I(x) = 0 \leq x.num\_elems \leq x.capacity$

- Función de abstracción:

  $f(x) = [\ x.elems[0],\ x.elems[1],\ \ldots,\ x.elems[x.num\_elems - 1]\ ]$

Creación y destrucción de una lista

```
const int DEFAULT_CAPACITY = 10;

class ListArray {

public:
  ListArray(int initial_capacity)
     : num_elems(0), capacity(initial_capacity), elems(new std::string[capacity]) { }

  ListArray()
     : ListArray(DEFAULT_CAPACITY) { }


  ~ListArray() { delete[] elems; }

   …

private:
  int num_elems;
  int capacity;
  std::string *elems;
};
```

Creación y destrucción de una lista

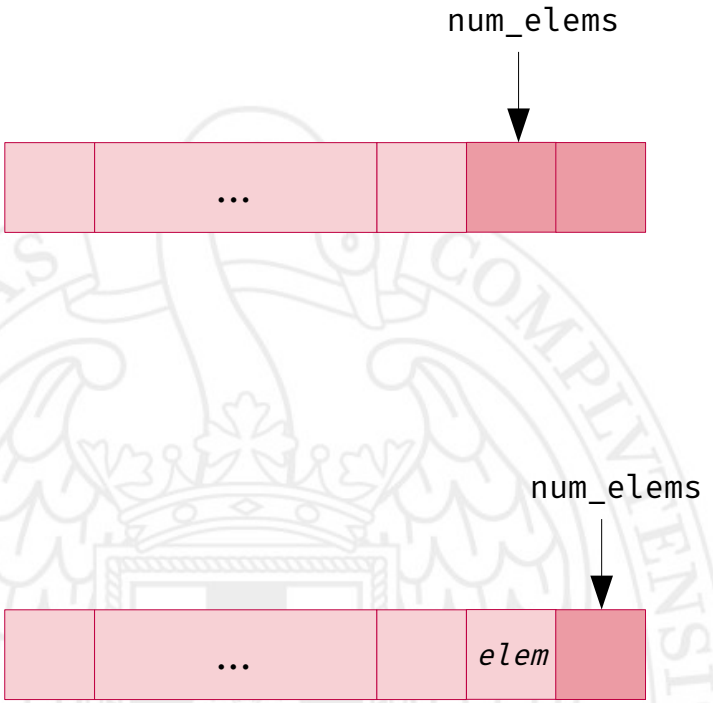
```
const int DEFAULT_CAPACITY = 10;

class ListArray {

public:
  ListArray(int initial_capacity = DEFAULT_CAPACITY)
     : num_elems(0), capacity(initial_capacity), elems(new std::string[capacity]) { }

  ListArray()
     : ListArray(DEFAULT_CAPACITY) { }


  ~ListArray() { delete[] elems; }

    …

private:
  int num_elems;
  int capacity;
  std::string *elems;
};
```

Valores por defecto para parámetros

Añadir elementos al final de una lista

```
class ListArray {
public:
   void push_back(const std::string &elem);
   …
private:
  int num_elems;
  int capacity;
  std::string *elems;
};
```

```
void ListArray::push_back(const std::string &elem) {
  if (num_elems == capacity) {
    resize_array(capacity * 2);
  }

  elems[num_elems] = elem;
  num_elems++;
}
```

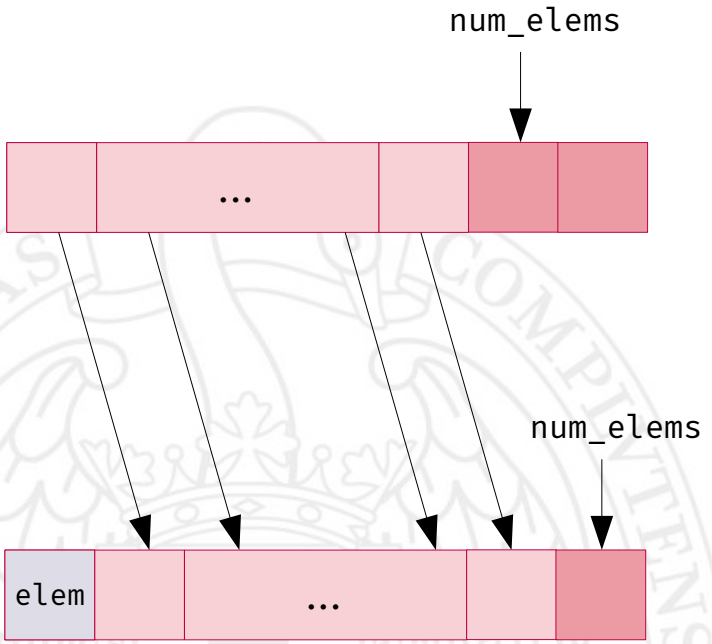


Redimensionar el array

```
class ListArray {
public:
   …
private:
  int num_elems;
  int capacity;
  std::string *elems;

  void resize_array(int new_capacity);
};

void ListArray::resize_array(int new_capacity) {
  std::string *new_elems = new std::string[new_capacity];
  for (int i = 0; i < num_elems; i++) {
    new_elems[i] = elems[i];
  }

  delete[] elems;
  elems = new_elems;
  capacity = new_capacity;
}
```

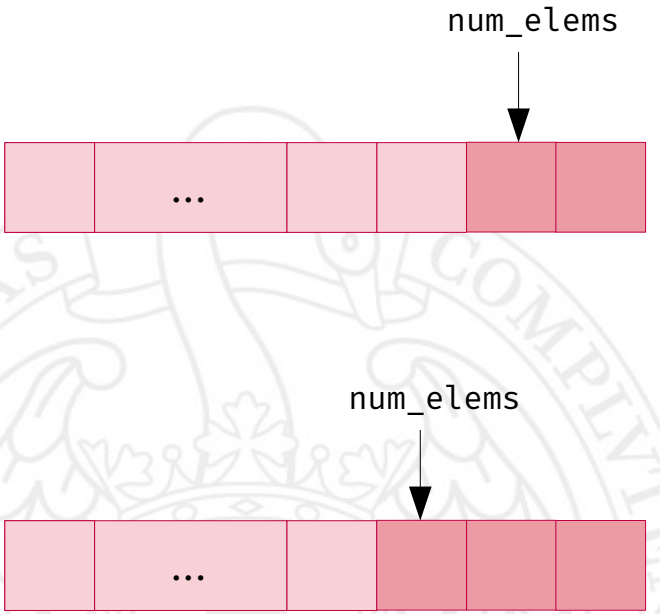
Añadir elementos al principio de una lista

```
class ListArray {
public:
  void push_front(const std::string &elem);
  …
};


void ListArray::push_front(const std::string &elem) {
  if (num_elems == capacity) {
    resize_array(capacity * 2);
  }

  for (int i = num_elems - 1; i ⩾ 0; i--) {
    elems[i + 1] = elems[i];
  }
  elems[0] = elem;
  num_elems++;
}
```



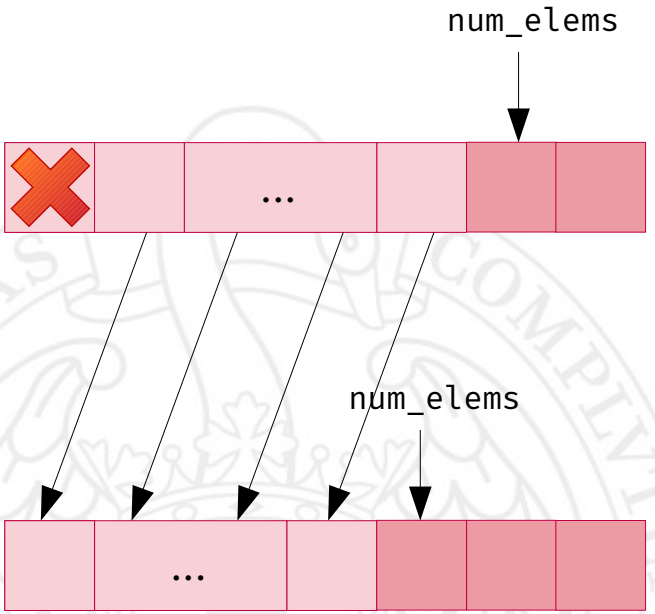
Eliminar elementos del final de una lista

```
class ListArray {
public:
  void pop_back();
  …
};


void ListArray::pop_back() {
  assert (num_elems > 0);
  num_elems--;
}
```



Eliminar elementos del principio de una lista

```
class ListArray {
public:
  void pop_front();
  …
};


void ListArray::pop_front() {
  assert (num_elems > 0);

  for (int i = 1; i < num_elems; i++) {
    elems[i - 1] = elems[i];
  }
  num_elems--;
}
```



Funciones observadoras (1)

```
class ListArray {
public:

  int size() const {
    return num_elems;
  }

  bool empty() const {
    return num_elems == 0;
  }

  std::string front() const {
    assert (num_elems > 0);
    return elems[0];
  }
  ...
};
```

Funciones observadoras (2)

```
class ListArray {
public:

  std::string back() const {
    assert (num_elems > 0);
    return elems[num_elems - 1];
  }

  std::string at(int index) const {
    assert (0 ⩽ index && index < num_elems);
    return elems[index];
  }
   ...
};
```

Mostrar el contenido de una lista

```
class ListArray {
public:
  void display() const;
  …
};


void ListArray::display() const {
  std::cout << "[";
  if (num_elems > 0) {
    std::cout << elems[0];
    for (int i = 1; i < num_elems; i++) {
      std::cout << ", " << elems[i];
    }
  }
  std::cout << "]";
}
```

Prueba de ejecución

```cpp
int main() {
  ListArray l;
  l.push_back("David");
  l.push_back("Maria");
  l.push_back("Elvira");
  l.display(); std::cout << std::endl;

  std::cout << "Elemento 1: " << l.at(1) << std::endl;

  l.pop_front();
  l.display(); std::cout << std::endl;

  return 0;
}
```

```
[David, Maria, Elvira]

Maria

[Maria, Elvira]
```