

ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS LINEALES

Implementando el TAD Pila

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

Operaciones sobre pilas

Vamos a ver 2 implementaciones sobre el TAD pila

MEDIANTE VECTORES
MEDIANTE LISTAS ENLAZADAS.
HAY MÁS IMPLEMENTACIONES.

- **Constructoras:**

- Crear una pila vacía (***create_empty***).

- **Mutadoras:**

- Añadir elemento en la cima de la pila (***push***).
- Eliminar elemento en la cima de la pila (***pop***).

TODAS LAS OPERACIONES SE REALIZAN
SOBRE EL ELEMENTO DE LA CIMA

- **Observadoras:**

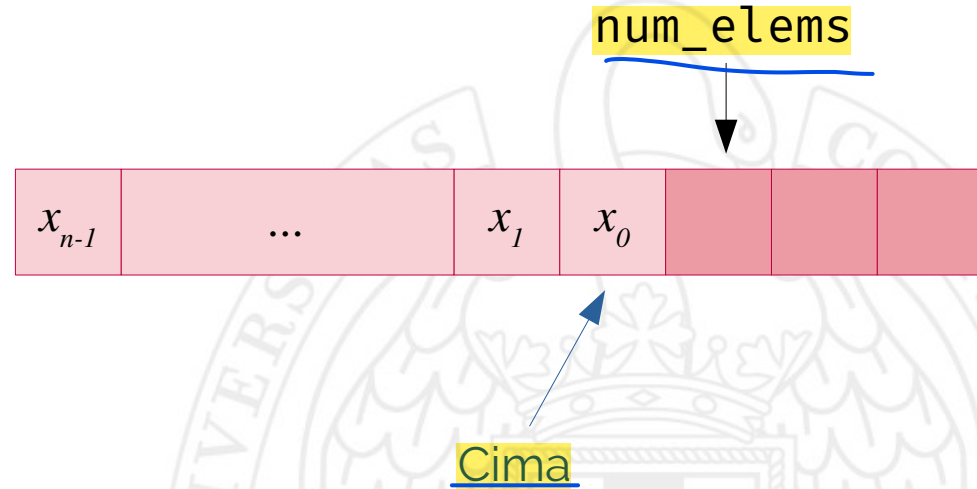
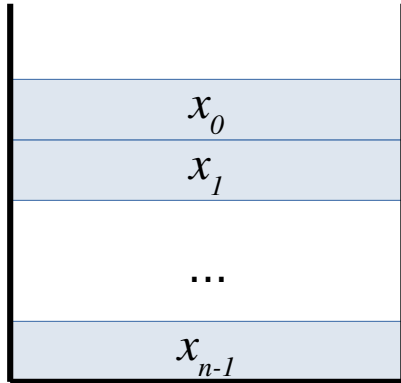
- Obtener el elemento en la cima de la pila (***top***).
- Saber si una pila está vacía (***empty***).

Implementación mediante vectores

Perjudica en el coste a la hora de hacer push()

O arrays

Implementación mediante vectores



Elementos se disponen de izquierda a derecha. Nosotros tenemos acceso al último elemento que ponemos

Implementación mediante vectores

```
template<typename T>
class StackArray {
public:
    ...
```

```
private:
```

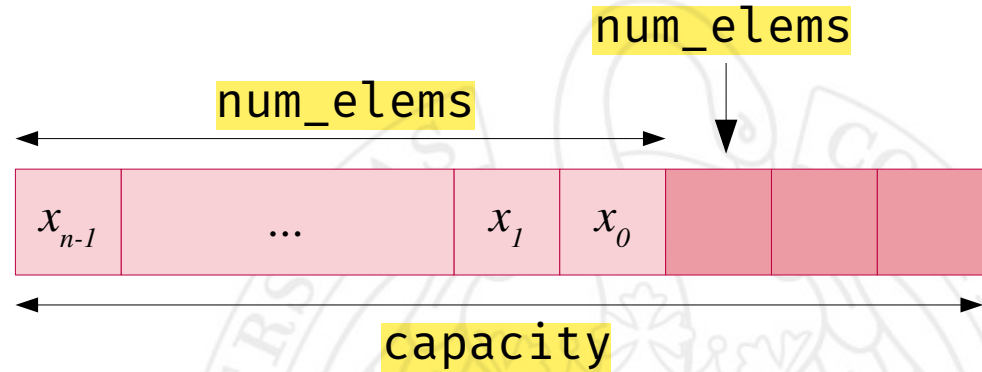
```
    int num_elems;
```

```
    int capacity;
```

```
    T *elems;
```

```
};
```

Genérico para el tipo de elementos



Primera posición del array que nosotros denotamos como vacía.

Interfaz pública de StackArray

```
template<typename T>
```

```
class StackArray {
```

```
public:
```

```
    StackArray(int initial_capacity = DEFAULT_CAPACITY);
```

```
    StackArray(const StackArray &other);
```

```
    ~StackArray();
```

Constructores y destructores.

```
    StackArray & operator=(const StackArray<T> &other);
```

Sobrecarga del operador igual.

```
    void push(const T &elem);
```

```
    void pop();
```

```
    const T & top() const;
```

```
    T & top();
```

```
    bool empty() const;
```

Primera devuelve una referencia constante, y la segunda referencias no constantes.

Vamos, que nos permitiría cambiar por el segundo método la cima de una pila si nosotros quisiéramos.

```
private:
```

```
    ...
```

```
};
```

Interfaz pública de StackArray

```
template<typename T>
class StackArray {
public:
    StackArray(int initial_capacity = DEFAULT_CAPACITY);
    StackArray(const StackArray &other);
    ~StackArray();

    StackArray & operator=(const StackArray<T> &other);

    void push(const T &elem);
    void pop();
    const T & top() const;
    T & top();
    bool empty() const;

private:
    ...
};
```

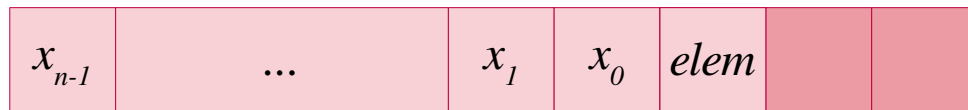
SÓLAMENTE VAMOS A IMPLEMENTAR LAS QUE ESTÁN MARCADAS/ SUBRAYADAS.

Métodos push() y pop()

```
void push(const T &elem) {  
    if (num_elems == capacity) {  
        resize_array(capacity * 2);  
    }  
    elems[num_elems] = elem;  
    num_elems++;  
}
```

PRIMERA POSICIÓN VACÍA

La función resize tiene coste lineal

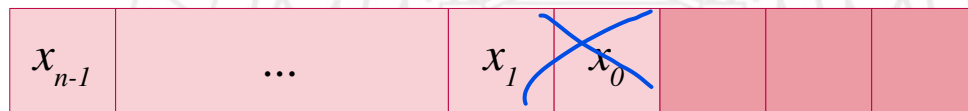


Numelems es la última posición vacía.

```
void pop() {  
    assert(num_elems > 0);  
    num_elems--;  
}
```

De modo que, en este caso, el valor de x_0 ya no se consideraría.

num_elems



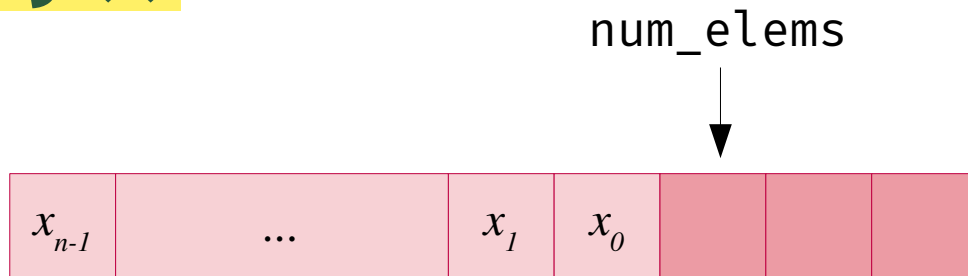
Métodos `top()` y `empty()`

```
const T & top() const {  
    assert(num_elems > 0);  
    return elems[num_elems - 1];  
}
```

La última posición NO VACÍA del array.

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

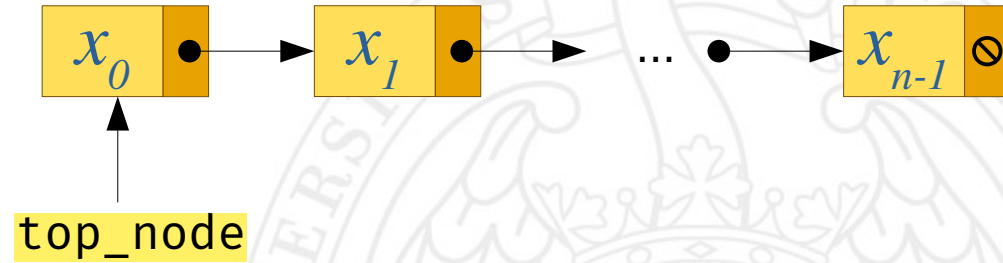
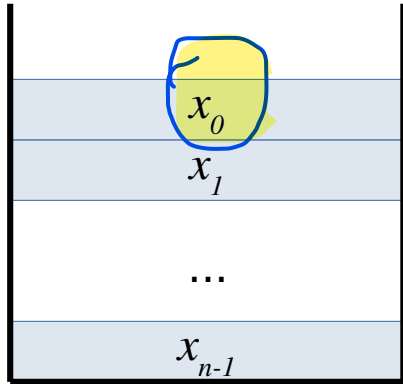
True si ocurre eso



Implementación mediante listas enlazadas simples

Con el modelo más sencillo, sin nodos fantasma ni nada.

Implementación mediante listas enlazadas



Añadimos nodos detrás del `top_node`, actualizamos el `top_node`, y para eliminar siempre vamos eliminando el `top_node`,

Implementación mediante listas enlazadas

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

```
    ...
private:
```

```
    struct Node {
        T value;
        Node *next;
    };
};
```

```
Node *top_node;
```

Puntero al nodo que representa la cima de la lista. Tenemos que el primer nodo apunta a nullptr. Importante supongo ir actualizando este.



Interfaz pública de StackLinkedList

```
template<typename T>
class StackLinkedList {
public:
    StackLinkedList();
    StackLinkedList(const StackLinkedList &other);
    ~StackLinkedList();

    StackLinkedList & operator=(const StackLinkedList<T> &other);

    void push(const T &elem);
    void pop();
    const T & top() const;
    T & top();
    bool empty() const;

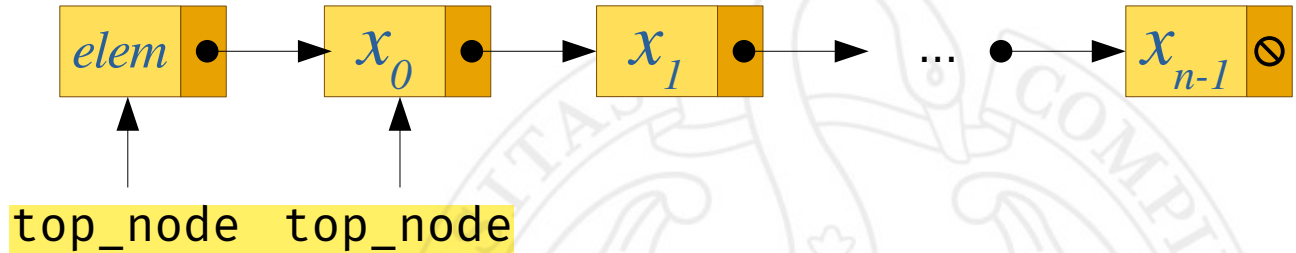
private:
    ...
};
```

Solo nos vamos a encargar de la implementación de los métodos marcados con subrayador.

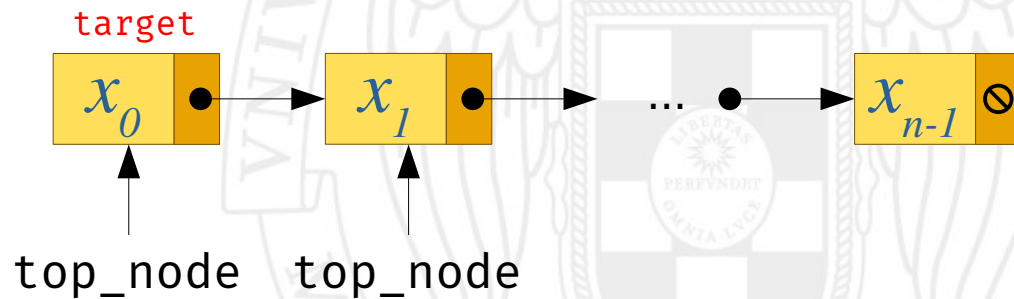
Operaciones push() y pop()

```
void push(const T &elem) {  
    top_node = new Node{ elem, top_node };  
}
```

Nodo que queremos insertar.



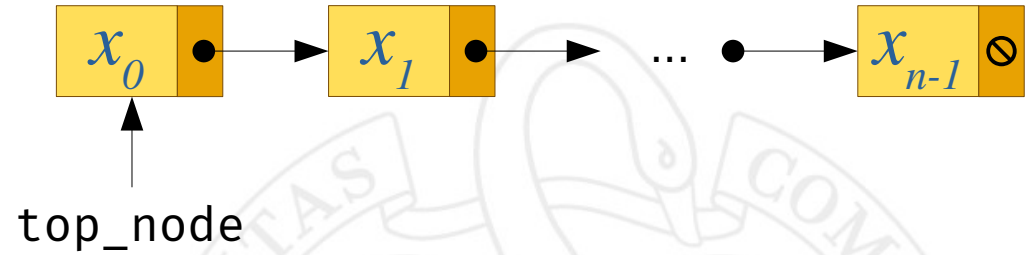
```
void pop() {  
    assert (top_node != nullptr);  
    Node *target = top_node;  
    top_node = top_node->next;  
    delete target;  
}
```



Operaciones `top()` y `empty()`

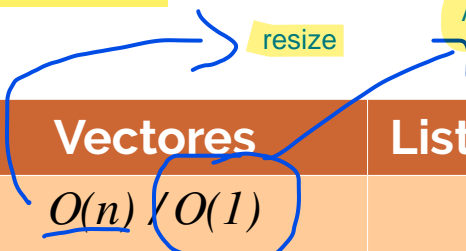
```
const T & top() const {  
    assert (top_node != nullptr);  
    return top_node->value;  
}
```

```
bool empty() const {  
    return (top_node == nullptr);  
}
```



Coste de las operaciones

COSTE AMORTIZADO, YA QUE NO SIEMPRE ES NECESARIO HACER EL RESIZE. LO VEREMOS MÁS ADELANTE, EN OTRO CURSO.



Operación	Vectores	Listas enlazadas
push	$O(n) / O(1)$	$O(1)$
pop	$O(1)$	$O(1)$
top	$O(1)$	$O(1)$
empty	$O(1)$	$O(1)$

n = número de elementos en la pila

Todo coste constante salvo la operación de apilar en el caso de los vectores, que es de coste constante.