

Pilas y Colas

Mauricio Avilés

Contenido

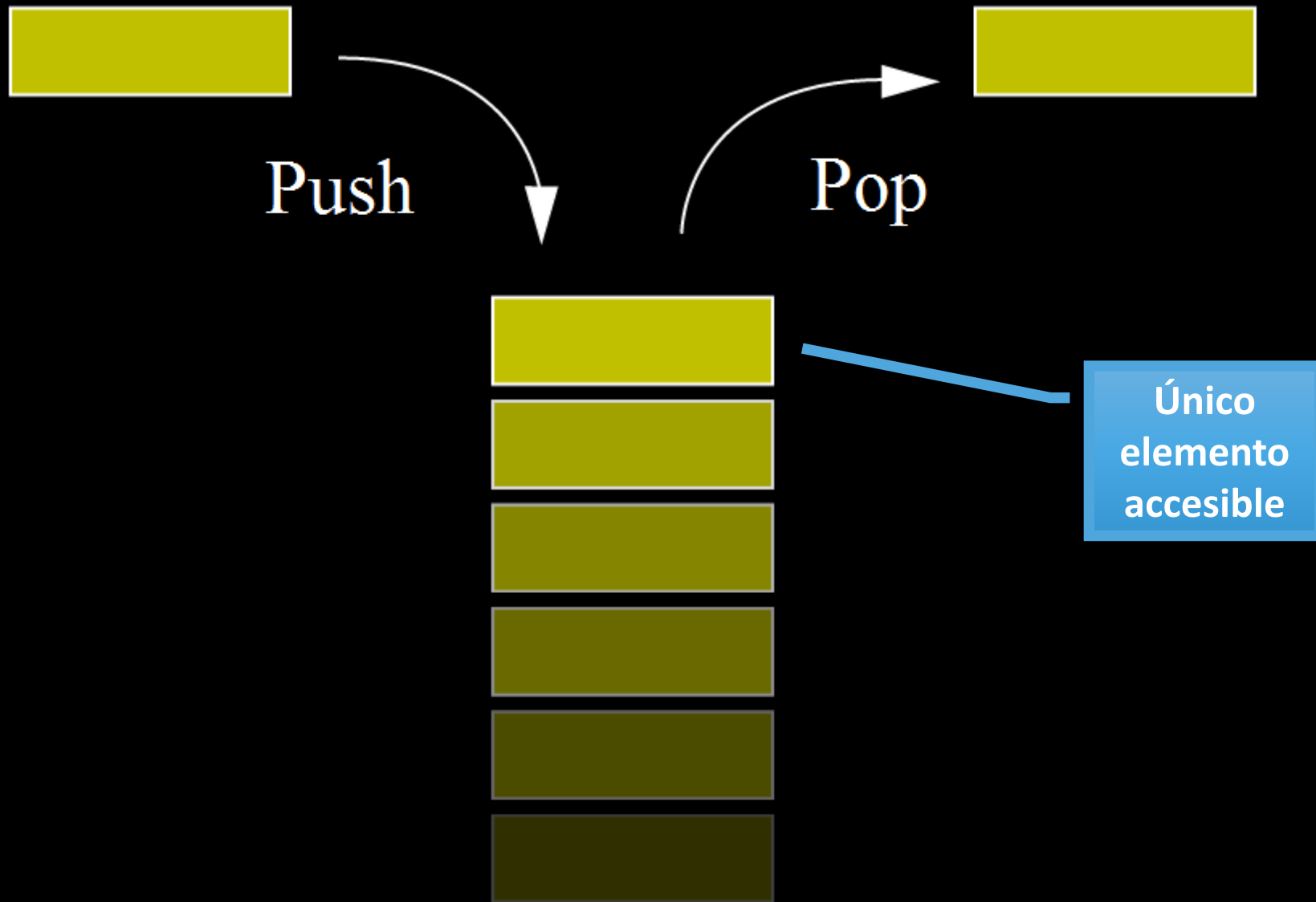
- Concepto de pilas
- Pilas basadas en arreglos
- Pilas basadas en enlaces
- Comparación de representaciones
- Colas basadas en arreglos
- Colas basadas en enlaces
- Comparación de representaciones

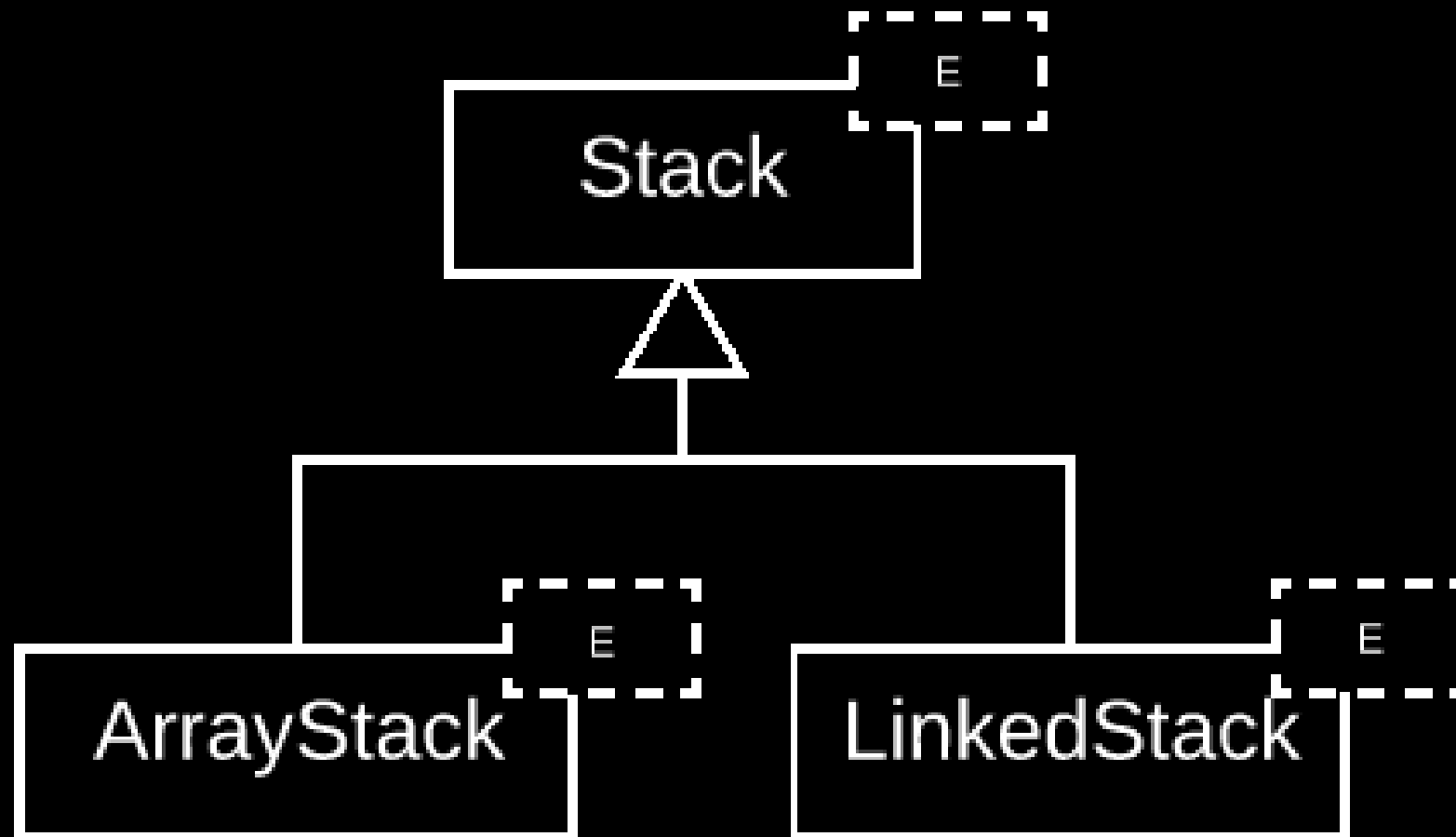
Lecturas

- Sección 4.2 y 4.3
 - Shaffer, C. A. "Data Structures & Algorithm Analysis in C++" (3rd ed., Dover). Mineola, NY. 2011.
- Capítulo 5
 - Goodrich, M. T., Tamassia, R., & Mount, D. M. "Data structures and algorithms in C++" (2nd ed., Wiley). Hoboken, NJ: Wiley. 2011.
- Capítulo 11 y 12
 - Joyanes, Aguilar, & Martínez. Estructura de datos en C ++. Madrid: McGraw-Hill Interamericana. 2007.

Pilas

- Estructura parecida a una lista
- Los elementos se insertan y se eliminan de un extremo de la lista
- Menos flexible que una lista
- Más eficiente y fácil de implementar
- LIFO





<u>Operación</u>	<u>Descripción</u>
push	Agrega un nuevo elemento al tope de la pila
pop	Elimina un elemento del tope de la pila
topValue	Retorna el valor del elemento en el tope de la pila
clear	Elimina todos los elementos de la pila
isEmpty	Dice si la pila se encuentra vacía
getSize	Retorna la cantidad de elementos en la pila

```
template <typename E>
class Stack {
private:
    void operator =(const Stack&) {}
    Stack(const Stack& obj) {}
public:
    Stack() {}
    virtual ~Stack() {}
    virtual void push(E element) = 0;
    virtual E pop() = 0;
    virtual E topValue() = 0;
    virtual void clear() = 0;
    virtual bool isEmpty() = 0;
    virtual int getSize() = 0;
};
```

Tipo genérico de los elementos a guardar en la pila.


```
template <typename E>
class Stack {
private:
    void operator =(const Stack&) {}
    Stack(const Stack& obj) {}
public:
    Stack() {}
    virtual ~Stack() {}
    virtual void push(E element) = 0;
    virtual E pop() = 0;
    virtual E topValue() = 0;
    virtual void clear() = 0;
    virtual bool isEmpty() = 0;
    virtual int getSize() = 0;
};
```

Protección de la asignación y del constructor de copia.

```
template <typename E>
class Stack {
private:
    void operator =(const Stack&) {}
    Stack(const Stack& obj) {}
public:
    Stack() {}
    virtual ~Stack() {}
    virtual void push(E element) = 0;
    virtual E pop() = 0;
    virtual E topValue() = 0;
    virtual void clear() = 0;
    virtual bool isEmpty() = 0;
    virtual int getSize() = 0;
};
```

Constructor vacío para la clase abstracta. Los constructores en C++ no se puede declarar virtual.

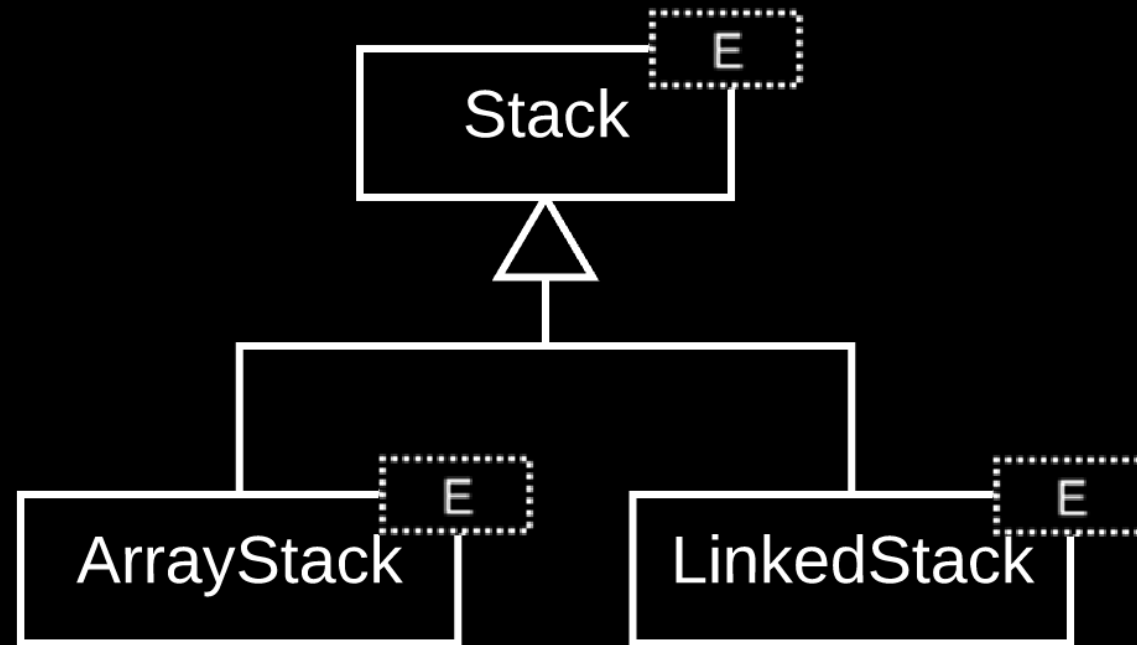
```
template <typename E>
class Stack {
private:
    void operator =(const Stack&) {}
    Stack(const Stack& obj) {}
public:
    Stack() {}
    virtual ~Stack() {}
    virtual void push(E element) = 0;
    virtual E pop() = 0;
    virtual E topValue() = 0;
    virtual void clear() = 0;
    virtual bool isEmpty() = 0;
    virtual int getSize() = 0;
};
```

Métodos que deben ser implementados por las subclases.

Implementaciones

Arreglos

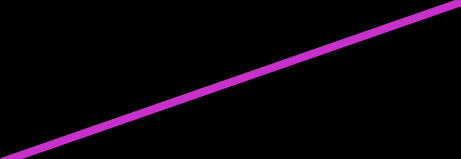
Listas
enlazadas



```
#include "Stack.h"  
#include <stdexcept>
```

```
using std::runtime_error;
```

```
template <typename E>  
class ArrayStack : public Stack<E> {  
private:  
    int maxSize;  
    int top;  
    E *elements;
```



Herencia pública de la clase Stack.

```
#include "Stack.h"  
#include <stdexcept>
```

```
using std::runtime_error;
```

```
template <typename E>  
class ArrayStack : public Stack<E> {  
private:  
    int maxSize;  
    int top;  
    E *elements;
```

Cantidad máxima de elementos en la pila.

```
#include "Stack.h"
#include <stdexcept>

using std::runtime_error;

template <typename E>
class ArrayStack : public Stack<E> {
private:
    int maxSize;
    int top;
    E *elements;
```

Primer espacio libre en el arreglo. También indica la cantidad de elementos en la pila.


```
#include "Stack.h"
#include <stdexcept>

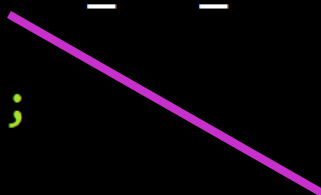
using std::runtime_error;

template <typename E>
class ArrayStack : public Stack<E> {
private:
    int maxSize;
    int top;
    E *elements;
```

Puntero al arreglo de elementos.

public:

```
ArrayStack(int maxSize = DEFAULT_MAX_SIZE) {  
    this->maxSize = maxSize;  
    elements = new E[maxSize];  
    top = 0;  
}  
~ArrayStack() {  
    delete [] elements;  
}
```



Constructor muy similar al constructor de la lista implementada con arreglos. Recibe la cantidad máxima de elementos.

public:

```
ArrayStack(int maxSize = DEFAULT_MAX_SIZE) {  
    this->maxSize = maxSize;  
    elements = new E[maxSize];  
    top = 0;  
}  
~ArrayStack() {  
    delete [] elements;  
}
```

Se inicializa el tamaño máximo y se crea el arreglo de elementos. Se inicializa el primer espacio libre en la posición 0.

public:

```
ArrayStack(int maxSize = DEFAULT_MAX_SIZE) {  
    this->maxSize = maxSize;  
    elements = new E[maxSize];  
    top = 0;  
}  
~ArrayStack() {  
    delete [] elements;  
}
```

Eliminar de memoria dinámica el arreglo de elementos.

El método para agregar elementos puede fallar si la pila ya se encuentra llena.

```
void push(E element) throw(runtime_error) {  
    if (top == maxSize) {  
        throw runtime_error("Stack overflow.");  
    }  
    elements[top++] = element;  
}  
E pop() throw(runtime_error) {  
    if (top == 0) {  
        throw runtime_error("Stack underflow.");  
    }  
    return elements[--top];  
}
```

```
void push(E element) throw(runtime_error) {  
    if (top == maxSize) {  
        throw runtime_error("Stack overflow.");  
    }  
    elements[top++] = element;  
}  
E pop() throw(runtime_error) {  
    if (top == 0) {  
        throw runtime_error("Stack underflow.");  
    }  
    return elements[--top];  
}
```

Se asigna el nuevo elemento en la primera posición libre y posteriormente se incrementa el valor de top.

```
void push(E element) throw(runtime_error) {  
    if (top == maxSize) {  
        throw runtime_error("Stack overflow.");  
    }  
    elements[top++] = element;  
}  
E pop() throw(runtime_error) {  
    if (top == 0) {  
        throw runtime_error("Stack underflow.");  
    }  
    return elements[--top];  
}
```

El método puede fallar si la pila se encuentra vacía.


```
void push(E element) throw(runtime_error) {  
    if (top == maxSize) {  
        throw runtime_error("Stack overflow.");  
    }  
    elements[top++] = element;  
}  
E pop() throw(runtime_error) {  
    if (top == 0) {  
        throw runtime_error("Stack underflow.");  
    }  
    return elements[--top];  
}
```

Primero se decrementa la posición del primer espacio libre, luego se retorna el elemento ubicado en esa posición.

El método puede fallar si la pila se encuentra vacía.

```
E topValue() throw(runtime_error) {  
    if (top == 0) {  
        throw runtime_error("Stack is empty.");  
    }  
    return elements[top-1];  
}  
void clear() {  
    top = 0;  
}
```

```
E topValue() throw(runtime_error) {  
    if (top == 0) {  
        throw runtime_error("Stack is empty.");  
    }  
    return elements[top-1];  
}  
void clear() {  
    top = 0;  
}
```



Se retorna el elemento en la última posición ocupada. Note que no se modifica el valor de top porque no se está agregando ni eliminando un elemento.

```
E topValue() throw(runtime_error) {  
    if (top == 0) {  
        throw runtime_error("Stack is empty.");  
    }  
    return elements[top-1];  
}  
void clear() {  
    top = 0;  
}
```

Borrado lógico. Los elementos siguen existiendo en el arreglo, pero serán sobrescritos cuando la estructura acepte nuevos elementos.

```
bool isEmpty() {  
    return top == 0;  
}  
int getSize() {  
    return top;  
}  
};
```

En este método se verifica si el valor del primer espacio libre en el arreglo es 0.

```
bool isEmpty() {  
    return top == 0;  
}  
int getSize() {  
    return top;  
}  
};
```

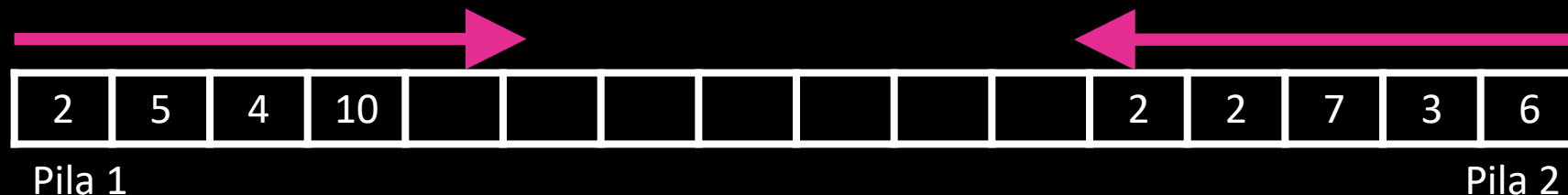
La posición del primer espacio libre también indica la cantidad de elementos en la estructura. Se retorna su valor.

Ejemplo de utilización

```
Stack<int> *s = new ArrayStack<int>();  
for (int i = 0; i < 100; i++) {  
    s->push(i);  
}  
while (s->getSize() > 10) {  
    cout << s->pop() << endl;  
}  
cout << s->getSize() << endl;  
s->clear();  
s->push(2343);  
cout << s->topValue() << endl;  
delete s;
```

Ejercicios con ArrayStack

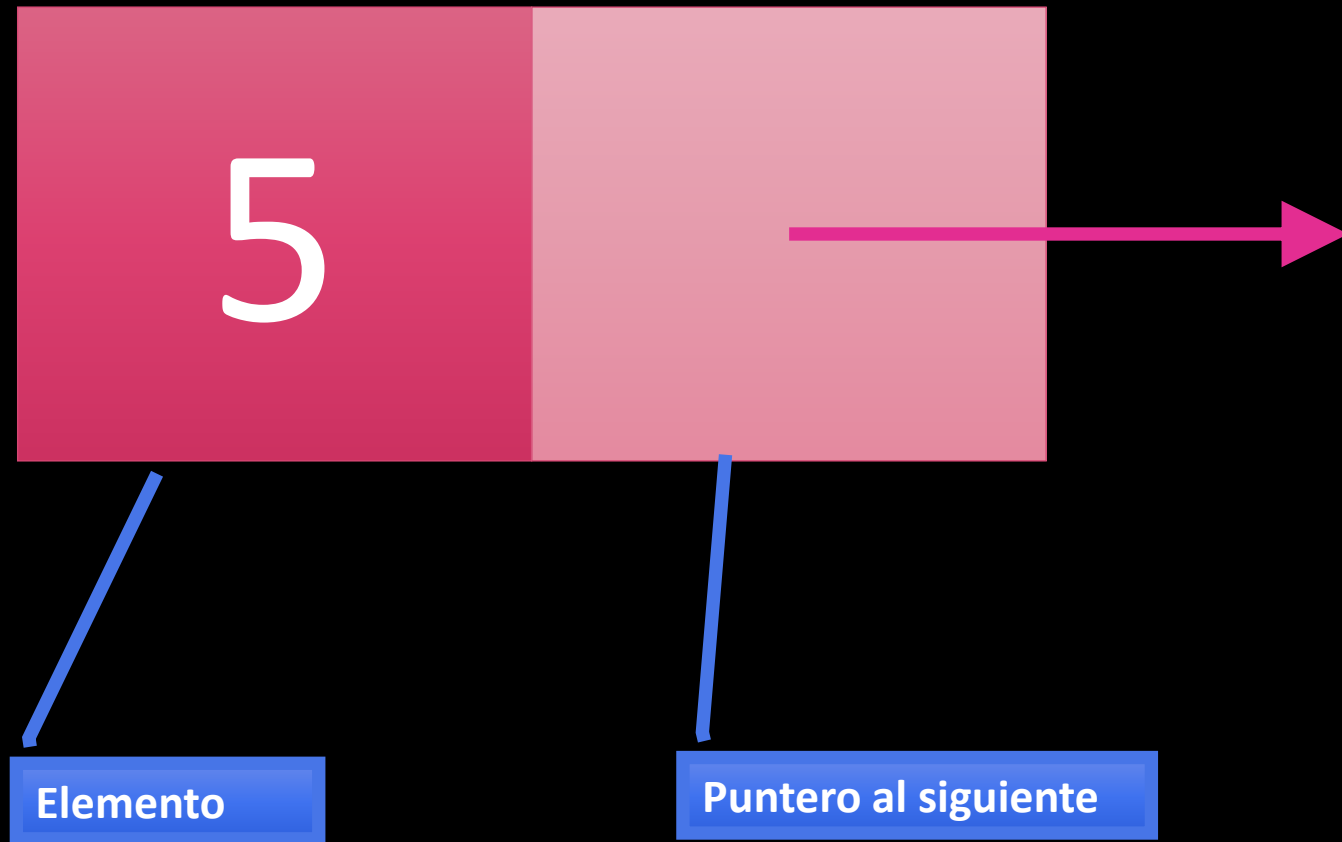
- Modificar el método **push** para que sea **dinámico**.
 - Si la pila está a punto de llenarse, sustituir el arreglo por uno más grande.
 - Mover los elementos al nuevo arreglo y eliminar el anterior de memoria dinámica.
- Implementar un método que se encargue de **imprimir** en pantalla los contenidos de la pila.
- Implementar un método **copiar** que reciba por parámetro otro objeto tipo Stack y que se encargue de copiar los contenidos de la pila actual en la pila enviada. La pila actual no debe cambiar sus elementos.
- El arreglo de un ArrayStack se utiliza más de un lado que de otro. Los primeros elementos se agregan al inicio y los espacios del final del arreglo tienden a estar siempre vacíos, a menos que la pila se llene. Implemente una **pila doble** que tenga dos pilas utilizando cada extremo del arreglo y así aprovechar el espacio.



Pilas basadas en punteros

- Implementación simple
- Los elementos se insertan y se eliminan de la **cabeza**
- Un nodo especial es **innecesario**

Nodo



```
#include <stdexcept>
#include "Node.h"

using std::runtime_error;

template <typename E>
class LinkedStack : public Stack<E> {
private:
    Node<E>* top;
    int size;
```

Puntero al elemento que se encuentra en el tope de la pila

```
#include <stdexcept>
#include "Node.h"

using std::runtime_error;

template <typename E>
class LinkedStack : public Stack<E> {
private:
    Node<E>* top;
    int size;
```

Cantidad de elementos en la pila.

public:

```
    LinkedStack() {  
        top = NULL;  
        size = 0;  
    }  
    ~LinkedStack() {  
        clear();  
    }
```

La pila se inicializa con el puntero top apuntando a nulo y la cantidad de elementos en cero.

```
public:
    LinkedStack() {
        top = NULL;
        size = 0;
    }
    ~LinkedStack() {
        clear();
    }
```

El destructor invoca al método clear para eliminar todos los elementos. Como no se hace uso de un nodo especial no es necesario hacer nada más.

```
void push(E element) {
    top = new Node<E>(element, top);
    size++;
}
E pop() throw(runtime_error) {
    if (top == NULL) {
        throw runtime_error("Stack underflow.");
    }
    E result = top->element;
    Node<E>* temp = top;
    top = top->next;
    delete temp;
    size--;
    return result;
}
```

La operación de insertar en la pila consiste en insertar un nuevo nodo al inicio de la lista. Se crea el nuevo nodo con el elemento y su puntero next se inicializa apuntando al que primer elemento actual. Luego se actualiza el puntero top para que apunte al nuevo nodo y se aumenta la cantidad de elementos.

```
void push(E element) {
    top = new Node<E>(element, top);
    size++;
}
E pop() throw(runtime_error) {
    if (top == NULL) {
        throw runtime_error("Stack underflow.");
    }
    E result = top->element;
    Node<E>* temp = top;
    top = top->next;
    delete temp;
    size--;
    return result;
}
```

Este método puede fallar si la pila se encuentra vacía. También se puede verificar si el tamaño es igual a cero.

```
void push(E element) {
    top = new Node<E>(element, top);
    size++;
}
E pop() throw(runtime_error) {
    if (top == NULL) {
        throw runtime_error("Stack underflow.");
    }
    E result = top->element;
    Node<E>* temp = top;
    top = top->next;
    delete temp;
    size--;
    return result;
}
```

Se guarda el valor del elemento que se va a eliminar.


```
void push(E element) {
    top = new Node<E>(element, top);
    size++;
}
E pop() throw(runtime_error) {
    if (top == NULL) {
        throw runtime_error("Stack underflow.");
    }
    E result = top->element;
    Node<E>* temp = top;
    top = top->next;
    delete temp;
    size--;
    return result;
}
```

Se utiliza un puntero temporal para apuntar al nodo que se eliminará.

```
void push(E element) {
    top = new Node<E>(element, top);
    size++;
}
E pop() throw(runtime_error) {
    if (top == NULL) {
        throw runtime_error("Stack underflow.");
    }
    E result = top->element;
    Node<E>* temp = top;
    top = top->next;
    delete temp;
    size--;
    return result;
}
```

Se actualiza el puntero top para que apunte al segundo nodo.

```
void push(E element) {
    top = new Node<E>(element, top);
    size++;
}
E pop() throw(runtime_error) {
    if (top == NULL) {
        throw runtime_error("Stack underflow.");
    }
    E result = top->element;
    Node<E>* temp = top;
    top = top->next;
    delete temp;
    size--;
    return result;
}
```

Se elimina el nodo apuntado por el puntero temporal, se disminuye el tamaño y se retorna el valor del elemento eliminado.

```
E topValue() throw(runtime_error) {  
    if (top == NULL) {  
        throw runtime_error("Stack is empty.");  
    }  
    return top->element;  
}  
void clear() {  
    Node<E>* temp;  
    while (top != NULL) {  
        temp = top;  
        top = top->next;  
        delete temp;  
    }  
    size = 0;  
}
```

Este método puede fallar si la pila se encuentra vacía. También se puede verificar la cantidad de elementos.

```
E topValue() throw(runtime_error) {  
    if (top == NULL) {  
        throw runtime_error("Stack is empty.");  
    }  
    return top->element;  
}  
  
void clear() {  
    Node<E>* temp;  
    while (top != NULL) {  
        temp = top;  
        top = top->next;  
        delete temp;  
    }  
    size = 0;  
}
```

Se retorna el valor del elemento al que apunta el puntero top.

```
E topValue() throw(runtime_error) {  
    if (top == NULL) {  
        throw runtime_error("Stack is empty.");  
    }  
    return top->element;  
}  
void clear() {  
    Node<E>* temp;  
    while (top != NULL) {  
        temp = top;  
        top = top->next;  
        delete temp;  
    }  
    size = 0;  
}
```

Se utiliza un puntero temporal para apuntar a los nodos que se van eliminando.

```
E topValue() throw(runtime_error) {  
    if (top == NULL) {  
        throw runtime_error("Stack is empty.");  
    }  
    return top->element;  
}  
void clear() {  
    Node<E>* temp;  
    while (top != NULL) {  
        temp = top;  
        top = top->next;  
        delete temp;  
    }  
    size = 0;  
}
```

Mientras hayan elementos en la lista, el puntero temporal apunta al primer nodo, se actualiza el puntero top para que apunte al nodo, y se elimina de memoria el nodo en el puntero temporal.

```
E topValue() throw(runtime_error) {  
    if (top == NULL) {  
        throw runtime_error("Stack is empty.");  
    }  
    return top->element;  
}  
void clear() {  
    Node<E>* temp;  
    while (top != NULL) {  
        temp = top;  
        top = top->next;  
        delete temp;  
    }  
    size = 0;  
}
```

Se actualiza la cantidad de elementos a cero.


```
bool isEmpty() {  
    return top == NULL;  
}  
int getSize() {  
    return size;  
}  
};
```

Para saber si la pila está vacía se verifica si el puntero top apunta a nulo. También se puede verificar la cantidad de elementos.

```
bool isEmpty() {  
    return top == NULL;  
}  
int getSize() {  
    return size;  
}  
};
```

Se retorna la cantidad de elementos en la estructura.

Ejemplo de utilización

```
LinkedList<char> s;  
string texto;  
cout << "Ingrese el texto a invertir: ";  
getline(cin, texto);  
for (int i = 0; i < texto.length(); i++) {  
    s.push(texto[i]);  
}  
cout << "Texto invertido: ";  
while (!s.isEmpty()) {  
    cout << s.pop();  
}  
cout << endl;
```

Ejercicios con LinkedStack

- Implementar un método que se encargue de **imprimir** en pantalla los contenidos de la pila.
- Implementar un método **copiar** que reciba por parámetro otro objeto tipo Stack y que se encargue de copiar los contenidos de la pila actual en la pila enviada. La pila actual no debe cambiar sus elementos.
- Cambiar la implementación de la pila enlazada para que utilice una **lista de nodos libres**, similar el ejercicio realizado con listas enlazadas.

Ejercicios con pilas: verificar palíndromo

- Escriba un programa que lea un string de la consola y verifique si es un **palíndromo**. Para esto puede usarse la estructura pila.
- Se lee el string de consola. Cada uno de los caracteres se inserta en la pila.
- Luego se empiezan a sacar los elementos de la pila y uno a uno se comparan con los elementos de la secuencia.
- Si todos son iguales, entonces el string es un palíndromo, de otro modo no lo es.

Ejercicio con pilas: símbolos de agrupamiento

- Escriba un programa que determine si una secuencia de caracteres de agrupamiento `()`, `[]`, `{}`, corresponden a una **secuencia válida**. Esto se logra utilizando una pila.
- Se leen los caracteres que se desean analizar. Si nos encontramos con un carácter de apertura `('`, `[`, `{`), entonces se inserta (push) en la pila. Si se encuentra un carácter de cierre `)`, `]`, `}`) entonces se elimina del tope (pop) y se verifica que el elemento eliminado corresponda con el carácter de cierre.
- Si en algún momento no corresponden, la secuencia no es válida. Si la pila está vacía y se encuentra un carácter de cierre, tampoco es válida. Si se llega al final de la secuencia y quedan elementos en la pila, tampoco es válida.
- La secuencia es válida si se llega al final y la pila está vacía.
 - `{([])}` → válido
 - `{(}` → no válido
 - `{([()())][{}]}` → válido

Comparación

- Con arreglos

- Acceso a elementos por medio de índices es eficiente
- No se requiere desplazar elementos porque sólo se agregan al final
- Requiere memoria aunque no se necesite
- Tiene un tamaño límite
- El borrado de todos los elementos no requiere ciclos

- Con punteros

- Sólo se accede el primer elemento, acceso es eficiente
- Implementación con punteros no realiza desplazamiento de elementos
- Se solicita memoria conforme crece la estructura
- No tiene tamaño límite
- El borrado de todos los elementos requiere ciclos

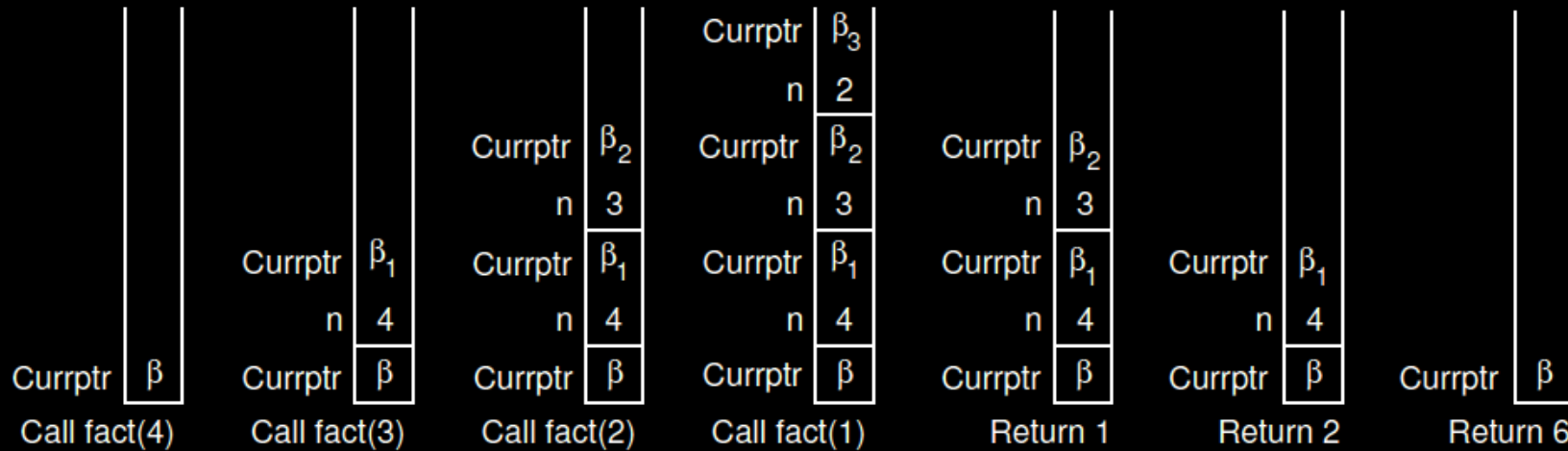
Implementación de recursión

- Una de las aplicaciones más comunes de las pilas
- Implementación de llamadas a subrutinas en lenguajes de programación
- Dirección de retorno, parámetros y variables locales → Registro de activación
- Cada retorno de una rutina saca el registro de activación de la pila


```

long fact(int n) {
    Assert((n >= 0) && (n <= 12), "Input out of range");
    if (n <= 1) return 1;
    return n * fact(n-1);
}

```

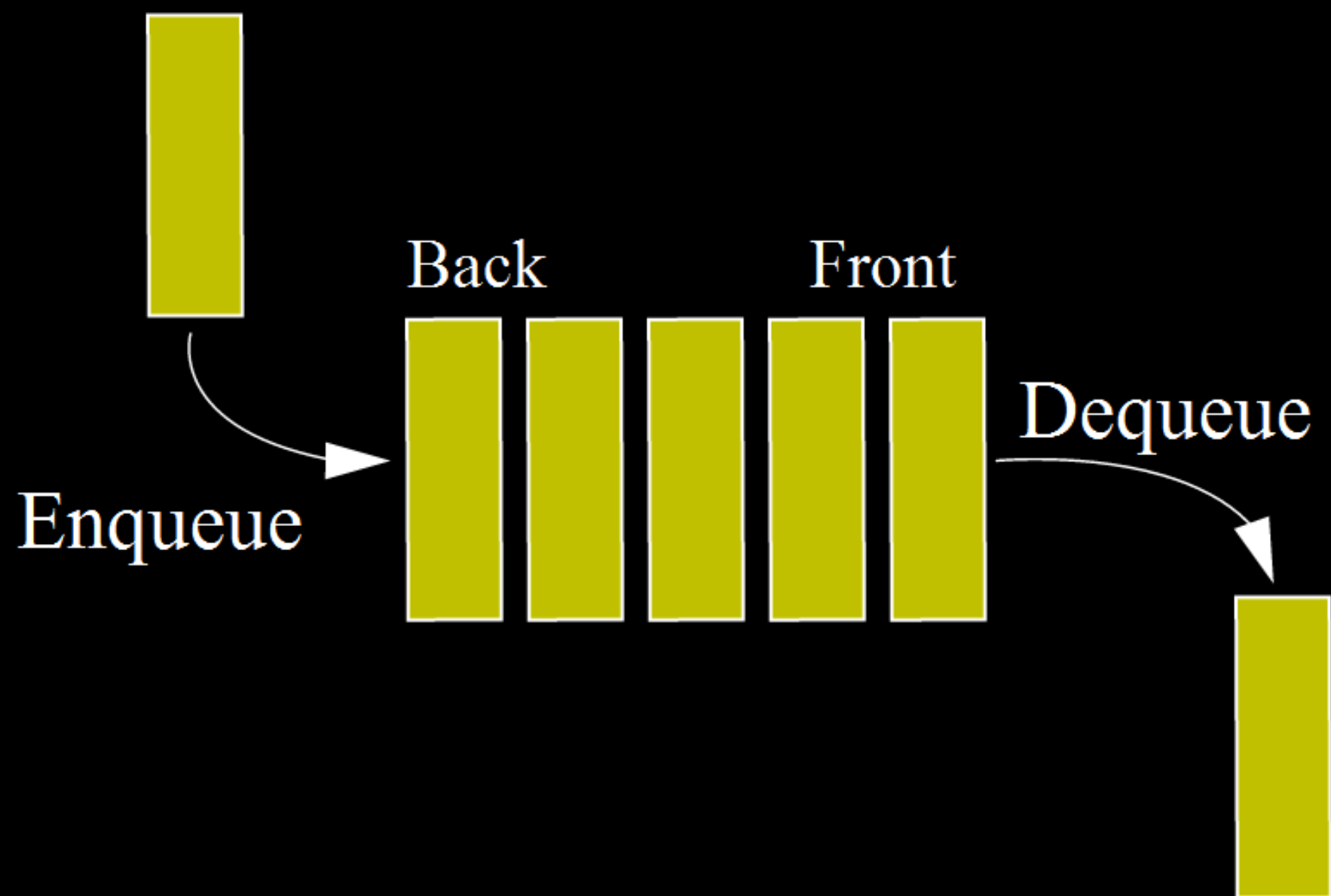


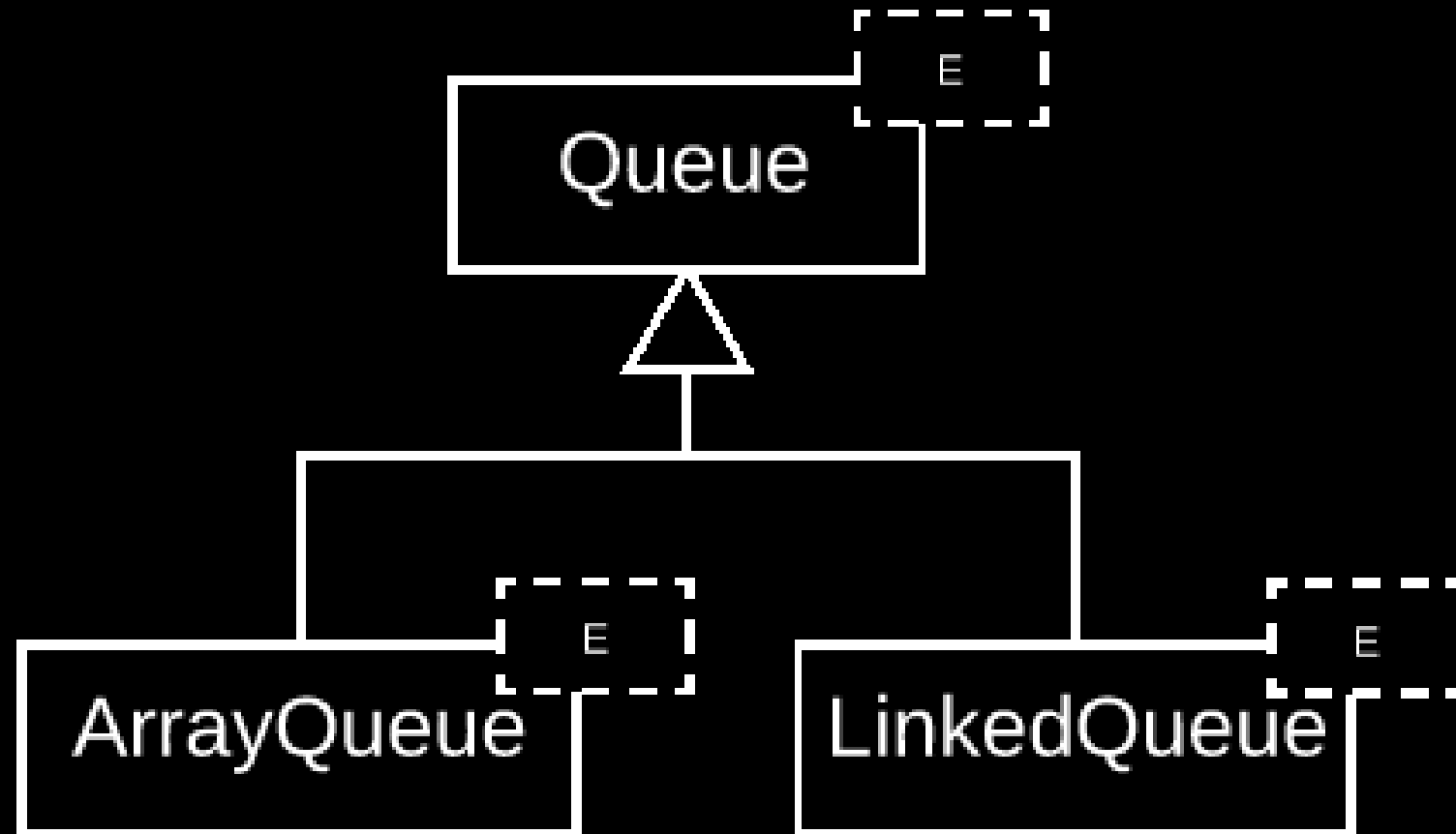
Return 24

- Aunque la recursión sirve para tratar de manera más simple algunos problemas, requiere mucho **overhead**
- Es más eficiente usar iteración
- Existen algoritmos que sólo se pueden implementar con recursión:
 - Mergesort
 - Quicksort
 - Recorridos en árboles
- Es posible **simular** recursión con una pila

Colas

- Estructura parecida a una lista
- Los elementos se insertan por un extremo y se remueven por el otro
- Funciona igual que una **fila** en la vida real (idealmente)
- **FIFO**





<u>Operación</u>	<u>Descripción</u>
enqueue	Agrega un nuevo elemento al final de la cola
dequeue	Elimina un elemento del inicio de la cola y retorna su valor
frontValue	Retorna el valor del elemento al inicio de la cola
clear	Elimina todos los elementos de la cola
isEmpty	Dice si la cola se encuentra vacía
getSize	Retorna la cantidad de elementos en la cola

```
template <typename E>
class Queue {
private:
    void operator =(const Queue&) {}
    Queue(const Queue& obj) {}

public:
    Queue() {}
    virtual ~Queue() {}
    virtual void enqueue(E element) = 0;
    virtual E dequeue() = 0;
    virtual E frontValue() = 0;
    virtual void clear() = 0;
    virtual bool isEmpty() = 0;
    virtual int getSize() = 0;
};
```

```
template <typename E>
class Queue {
private:
    void operator =(const Queue&) {}
    Queue(const Queue& obj) {}

public:
    Queue() {}
    virtual ~Queue() {}
    virtual void enqueue(E element) = 0;
    virtual E dequeue() = 0;
    virtual E frontValue() = 0;
    virtual void clear() = 0;
    virtual bool isEmpty() = 0;
    virtual int getSize() = 0;
};
```

Tipo genérico de los elementos a guardar en la cola.


```
template <typename E>
class Queue {
private:
    void operator =(const Queue&) {}
    Queue(const Queue& obj) {}

public:
    Queue() {}
    virtual ~Queue() {}
    virtual void enqueue(E element) = 0;
    virtual E dequeue() = 0;
    virtual E frontValue() = 0;
    virtual void clear() = 0;
    virtual bool isEmpty() = 0;
    virtual int getSize() = 0;
};
```

Protección del operador de asignación.

```
template <typename E>
class Queue {
private:
    void operator =(const Queue&) {}
    Queue(const Queue& obj) {}

public:
    Queue() {}
    virtual ~Queue() {}
    virtual void enqueue(E element) = 0;
    virtual E dequeue() = 0;
    virtual E frontValue() = 0;
    virtual void clear() = 0;
    virtual bool isEmpty() = 0;
    virtual int getSize() = 0;
};
```

Protección del operador de copia.

```
template <typename E>
class Queue {
private:
    void operator =(const Queue&) {}
    Queue(const Queue& obj) {}

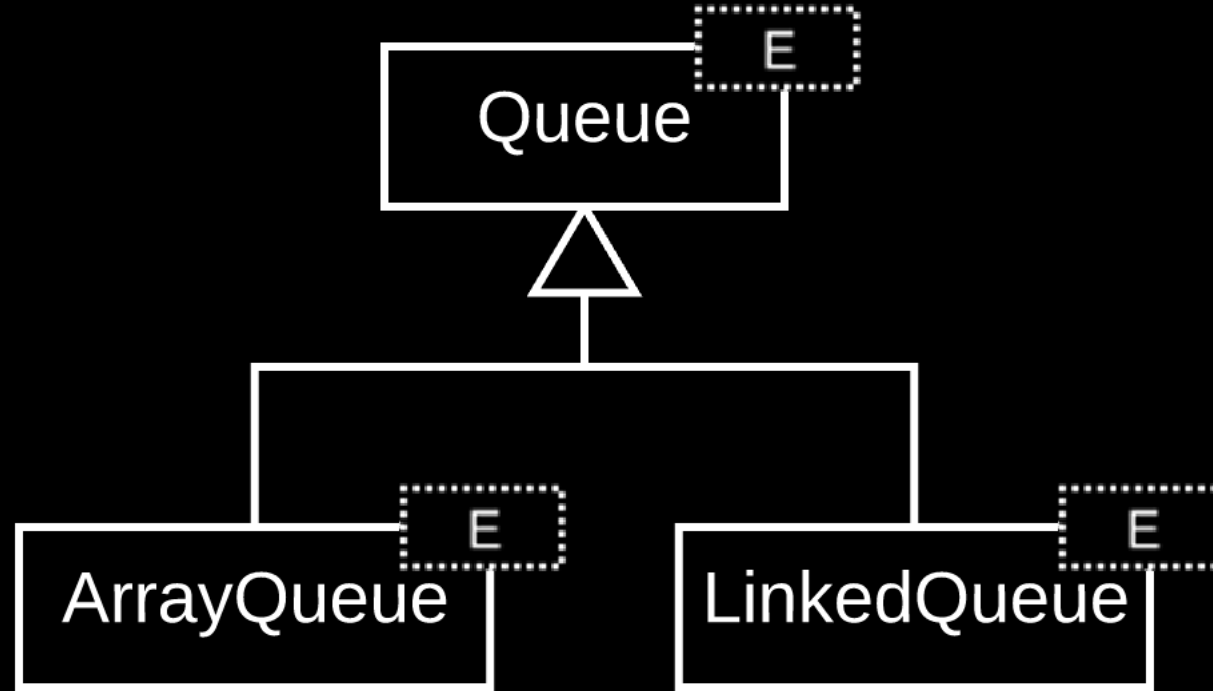
public:
    Queue() {}
    virtual ~Queue() {}
    virtual void enqueue(E element) = 0;
    virtual E dequeue() = 0;
    virtual E frontValue() = 0;
    virtual void clear() = 0;
    virtual bool isEmpty() = 0;
    virtual int getSize() = 0;
};
```

Métodos que deben implementar las subclases. En C++ los constructores no pueden ser marcados como virtuales.

Implementaciones

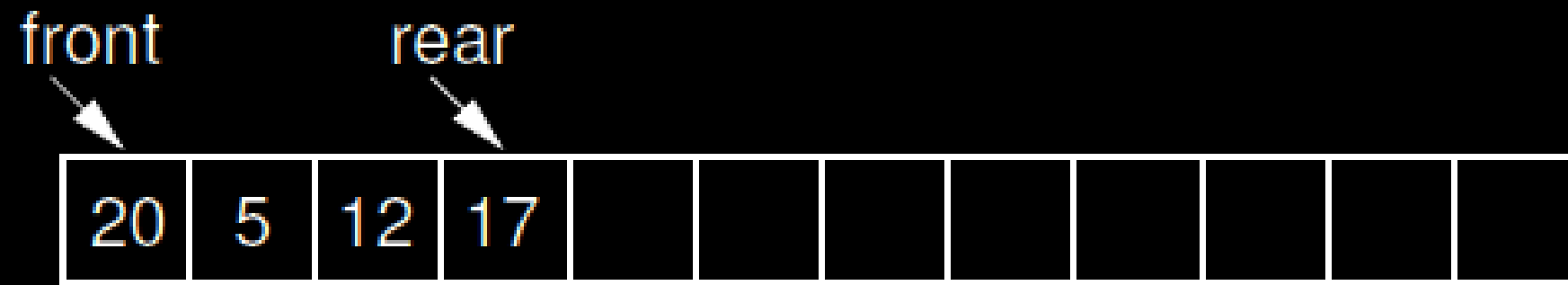
Arreglos

Listas
enlazadas

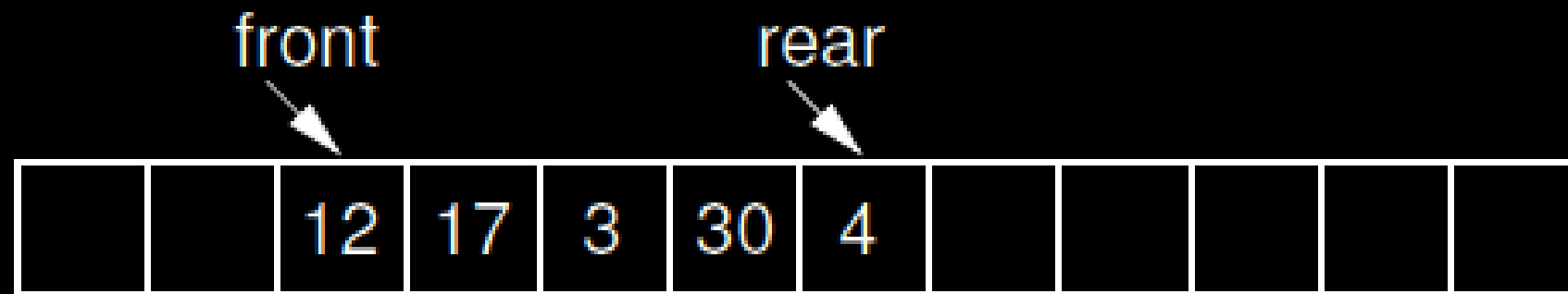


Colas basadas en arreglos

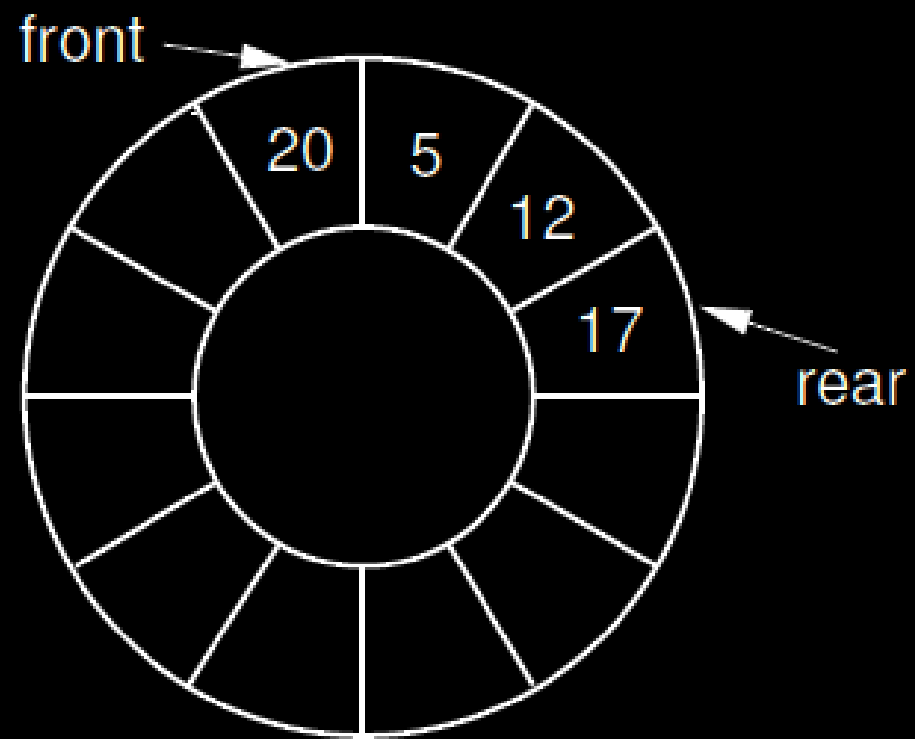
- Es **complicado** implementar colas en arreglos
- Si se inserta un elemento, todos los demás deben correrse un espacio
- Esto no es eficiente, por lo que debe buscarse una alternativa



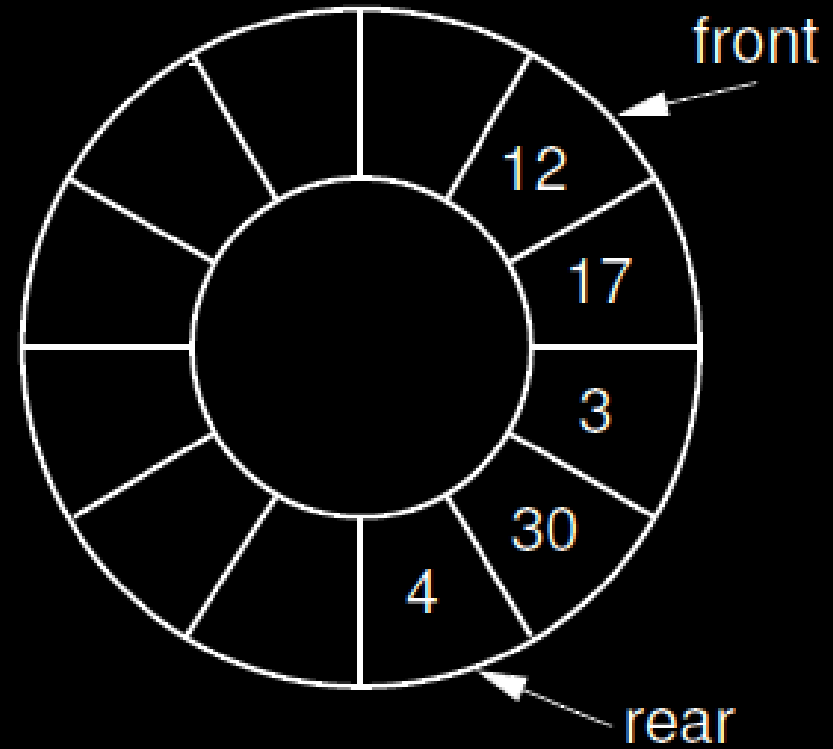
(a)



(b)



(a)



(b)

- Para manejar la lista como circular, se usa el operador de **módulo** (%)
- ¿Cómo diferenciar la lista llena de la vacía?
- Declarar un **espacio adicional** en el arreglo (Shaffer)
- Otra opción es utilizar un atributo de **tamaño**
- Se **simplifican** los métodos para saber si la lista está llena o vacía

```
#include <stdexcept>
#include "Queue.h"

using std::runtime_error;

template <typename E>
class ArrayQueue : public Queue<E> {
private:
    int front;
    int rear;
    int size;
    int maxSize;
    E* elements;
```

Herencia pública de la clase abstracta Queue,
determina cuáles métodos deben implementarse

```
#include <stdexcept>
#include "Queue.h"

using std::runtime_error;

template <typename E>
class ArrayQueue : public Queue<E> {
private:
    int front;
    int rear;
    int size;
    int maxSize;
    E* elements;
```

Posición del primer elemento de la cola.

```
#include <stdexcept>
#include "Queue.h"

using std::runtime_error;

template <typename E>
class ArrayQueue : public Queue<E> {
private:
    int front;
    int rear;
    int size;
    int maxSize;
    E* elements;
```

Posición del último elemento de la cola.

```
#include <stdexcept>
#include "Queue.h"

using std::runtime_error;

template <typename E>
class ArrayQueue : public Queue<E> {
private:
    int front;
    int rear;
    int size;
    int maxSize;
    E* elements;
```

Cantidad de elementos en la cola.

```
#include <stdexcept>
#include "Queue.h"

using std::runtime_error;

template <typename E>
class ArrayQueue : public Queue<E> {
private:
    int front;
    int rear;
    int size;
    int maxSize;
    E* elements;
```

Cantidad máxima de elementos en el arreglo.

```
#include <stdexcept>
#include "Queue.h"

using std::runtime_error;

template <typename E>
class ArrayQueue : public Queue<E> {
private:
    int front;
    int rear;
    int size;
    int maxSize;
    E* elements;
```

Puntero al arreglo de elementos.

Constructor recibe el tamaño del arreglo de elementos, que tiene un valor por defecto declarado en los define del compilador.

```
public:
    ArrayQueue(int maxSize = DEFAULT_MAX_SIZE) {
        this->maxSize = maxSize;
        elements = new E[maxSize];
        rear = 0;
        front = 1;
        size = 0;
    }
    ~ArrayQueue() {
        delete [] elements;
    }
```


public:

```
ArrayQueue(int maxSize = DEFAULT_MAX_SIZE) {  
    this->maxSize = maxSize;  
    elements = new E[maxSize];  
    rear = 0;  
    front = 1;  
    size = 0;  
}  
~ArrayQueue() {  
    delete [] elements;  
}
```

Se utiliza this para diferenciar el atributo del nombre del parámetro. Los otros atributos no lo necesitan porque no hay ambigüedad. Esto se podría evitar usando un nombre diferente en el parámetro.

public:

```
ArrayQueue(int maxSize = DEFAULT_MAX_SIZE) {  
    this->maxSize = maxSize;  
    elements = new E[maxSize];  
    rear = 0;  
    front = 1;  
    size = 0;  
}  
~ArrayQueue() {  
    delete [] elements;  
}
```

Se crea el arreglo de elementos del tamaño indicado y se el asigna al puntero elements.

public:

```
ArrayQueue(int maxSize = DEFAULT_MAX_SIZE) {  
    this->maxSize = maxSize;  
    elements = new E[maxSize];  
    rear = 0;  
    front = 1;  
    size = 0;  
}  
~ArrayQueue() {  
    delete [] elements;  
}
```

Se inicializa con rear menor que front porque la lista está vacía. Cuando haya al menos un elemento ambos apuntarán al mismo índice porque aumenta rear.

public:

```
ArrayQueue(int maxSize = DEFAULT_MAX_SIZE) {  
    this->maxSize = maxSize;  
    elements = new E[maxSize];  
    rear = 0;  
    front = 1;  
    size = 0;  
}  
~ArrayQueue() {  
    delete [] elements;  
}
```

Cantidad inicial de elementos.

public:

```
ArrayQueue(int maxSize = DEFAULT_MAX_SIZE) {  
    this->maxSize = maxSize;  
    elements = new E[maxSize];  
    rear = 0;  
    front = 1;  
    size = 0;  
}  
~ArrayQueue() {  
    delete [] elements;  
}
```

El destructor retorna la memoria del arreglo de elementos.

```
void enqueue(E element) throw(runtime_error) {  
    if (size == maxSize) {  
        throw runtime_error("Queue is full.");  
    }  
    rear = (rear+1) % maxSize;  
    elements[rear] = element;  
    size++;  
}  
E dequeue() throw(runtime_error) {  
    if (isEmpty()) {  
        throw runtime_error("Queue is empty.");  
    }  
    E result = elements[front];  
    front = (front+1) % maxSize;  
    size--;  
    return result;  
}
```

Recibe el elemento para agregar al inicio de la cola.
El método puede fallar si el arreglo ya se encuentra
lleno.

```
void enqueue(E element) throw(runtime_error) {
    if (size == maxSize) {
        throw runtime_error("Queue is full.");
    }
    rear = (rear+1) % maxSize;
    elements[rear] = element;
    size++;
}
E dequeue() throw(runtime_error) {
    if (isEmpty()) {
        throw runtime_error("Queue is empty.");
    }
    E result = elements[front];
    front = (front+1) % maxSize;
    size--;
    return result;
}
```

Aumenta en uno el valor del final de la cola. El módulo asegura que vuelva a cero si se encuentra al final del arreglo.

```
void enqueue(E element) throw(runtime_error) {  
    if (size == maxSize) {  
        throw runtime_error("Queue is full.");  
    }  
    rear = (rear+1) % maxSize;  
    elements[rear] = element;  
    size++;  
}  
E dequeue() throw(runtime_error) {  
    if (isEmpty()) {  
        throw runtime_error("Queue is empty.");  
    }  
    E result = elements[front];  
    front = (front+1) % maxSize;  
    size--;  
    return result;  
}
```

Se asigna el nuevo elemento en la nueva posición final y se incrementa la cantidad de elementos.



```
void enqueue(E element) throw(runtime_error) {  
    if (size == maxSize) {  
        throw runtime_error("Queue is full.");  
    }  
    rear = (rear+1) % maxSize;  
    elements[rear] = element;  
    size++;  
}  
E dequeue() throw(runtime_error) {  
    if (isEmpty()) {  
        throw runtime_error("Queue is empty.");  
    }  
    E result = elements[front];  
    front = (front+1) % maxSize;  
    size--;  
    return result;  
}
```

El método puede fallar si no hay elementos en la cola.

```
void enqueue(E element) throw(runtime_error) {
    if (size == maxSize) {
        throw runtime_error("Queue is full.");
    }
    rear = (rear+1) % maxSize;
    elements[rear] = element;
    size++;
}
E dequeue() throw(runtime_error) {
    if (isEmpty()) {
        throw runtime_error("Queue is empty.");
    }
    E result = elements[front];
    front = (front+1) % maxSize;
    size--;
    return result;
}
```

Se almacena en una variable temporal el valor del elemento que se va a eliminar.

```
void enqueue(E element) throw(runtime_error) {  
    if (size == maxSize) {  
        throw runtime_error("Queue is full.");  
    }  
    rear = (rear+1) % maxSize;  
    elements[rear] = element;  
    size++;  
}  
E dequeue() throw(runtime_error) {  
    if (isEmpty()) {  
        throw runtime_error("Queue is empty.");  
    }  
    E result = elements[front];  
    front = (front+1) % maxSize;  
    size--;  
    return result;  
}
```



Se aumenta el inicio de la cola para que no tome en cuenta el elemento. Se disminuye la cantidad de elementos y se retorna el valor.

```
E frontValue() throw(runtime_error) {  
    if (isEmpty()) {  
        throw runtime_error("Queue is empty.");  
    }  
    return elements[front];  
}  
void clear() {  
    delete [] elements;  
    elements = new E[maxSize];  
    rear = 0;  
    front = 1;  
    size = 0;  
}
```

El método puede fallar si no hay un elemento al inicio de la cola, es decir, si está vacía.

```
E frontValue() throw(runtime_error) {  
    if (isEmpty()) {  
        throw runtime_error("Queue is empty.");  
    }  
    return elements[front];  
}  
void clear() {  
    delete [] elements;  
    elements = new E[maxSize];  
    rear = 0;  
    front = 1;  
    size = 0;  
}
```

Simply se retorna el elemento que se encuentra en la posición indicada por front.

```
E frontValue() throw(runtime_error) {  
    if (isEmpty()) {  
        throw runtime_error("Queue is empty.");  
    }  
    return elements[front];  
}  
void clear() {  
    delete [] elements;  
    elements = new E[maxSize];  
    rear = 0;  
    front = 1;  
    size = 0;  
}
```

Al eliminar de memoria y volver a crear el arreglo de elementos se asegura que se invoque el destructor de dichos elementos, si es que lo tienen. Pero si se asume que la estructura va a ser utilizada con elementos de tipo nativo, entonces no es necesario este paso, simplemente se inicializan los índices.

```
E frontValue() throw(runtime_error) {  
    if (isEmpty()) {  
        throw runtime_error("Queue is empty.");  
    }  
    return elements[front];  
}  
void clear() {  
    delete [] elements;  
    elements = new E[maxSize];  
    rear = 0;  
    front = 1;  
    size = 0;  
}
```

Se inicializan los índices y el tamaño en cero.

```
bool isEmpty() {  
    return size == 0;  
}  
int getSize() {  
    return size;  
}  
};
```

Se verifica si la cantidad de elementos es cero.


```
bool isEmpty() {  
    return size == 0;  
}  
int getSize() {  
    return size;  
}  
};
```

Retornar la cantidad de elementos.

Ejemplo de utilización

```
Queue<int> *q = new ArrayQueue<int>();
srand(time(0));
for (int i = 0; i < 20; i++) {
    q->enqueue(rand());
}
cout << "Tamaño actual: " << q->getSize() << endl;
while (!q->isEmpty()) {
    cout << q->dequeue() << endl;
}
delete q;
```

Ejercicios con ArrayQueue

- Implementar un método que se encargue de **imprimir** en pantalla los contenidos de la cola.
- Implementar un método **copiar** que reciba por parámetro otro objeto tipo Queue y que se encargue de copiar los contenidos de la cola actual en la cola enviada. La cola actual no debe cambiar sus elementos.
- Una estructura **Deque** (double-ended queue) es una cola que permite operaciones de inserción y de borrado a **ambos extremos** de la cola. Modifique al ArrayQueue para que permita comportarse como una estructura Deque, implementando los siguientes métodos:
 - enqueueFront → insertar un elemento en el frente de la cola
 - dequeueRear → eliminar un elemento en el final de la cola
 - rearValue → retorna el elemento al final de la cola

Colas basadas en punteros

- Adaptación de la **lista enlazada**
- Front y rear son punteros al inicio y al final de la cola
- También se utiliza un **nodo especial**

```
#include <stdexcept>
#include "Queue.h"
#include "Node.h"

using std::runtime_error;
using std::cout;
using std::endl;

template <typename E>
class LinkedQueue : public Queue<E>{
private:
    Node<E>* front;
    Node<E>* rear;
    int size;
```

Puntero al inicio de la cola. Siempre apunta al nodo especial.

```
#include <stdexcept>
#include "Queue.h"
#include "Node.h"

using std::runtime_error;
using std::cout;
using std::endl;

template <typename E>
class LinkedQueue : public Queue<E>{
private:
    Node<E>* front;
    Node<E>* rear;
    int size;
```

Puntero al final de la cola. Siempre apunta al último elemento que se inserto. Inicialmente apunta al nodo especial.

```
#include <stdexcept>
#include "Queue.h"
#include "Node.h"

using std::runtime_error;
using std::cout;
using std::endl;

template <typename E>
class LinkedQueue : public Queue<E>{
private:
    Node<E>* front;
    Node<E>* rear;
    int size;
```

Cantidad de elementos en la cola.

public:

```
    LinkedQueue() {  
        front = rear = new Node<E>();  
        size = 0;  
    }  
    ~LinkedQueue() {  
        clear();  
        delete front;  
    }
```

La inicialización consiste en crear el nodo especial y que los punteros rear y front apunten hacia él. Se inicializa la cantidad de elementos en cero.


```
public:
    LinkedQueue() {
        front = rear = new Node<E>();
        size = 0;
    }
    ~LinkedQueue() {
        clear();
        delete front;
    }
```

Destructor similar al de la lista enlazada simple. Se eliminan los elementos y se borra el nodo especial.

```
void enqueue(E element) {  
    rear->next = new Node<E>(element);  
    rear = rear->next;  
    size++;  
}  
E dequeue() throw(runtime_error) {  
    if (front->next == NULL) {  
        throw runtime_error("Queue is empty.");  
    }  
    Node<E>* temp = front->next;  
    E result = temp->element;  
    front->next = front->next->next;  
    delete temp;  
    size--;  
    if (front->next == NULL) {  
        rear = front;  
    }  
    return result;  
}
```

Similar al método append de la lista enlazada simple. Se crea el nodo del nuevo elemento y se actualiza el último nodo para que apunte al nuevo. Se actualiza el puntero rear y se aumenta la cantidad de elementos.

```
void enqueue(E element) {
    rear->next = new Node<E>(element);
    rear = rear->next;
    size++;
}
E dequeue() throw(runtime_error) {
    if (front->next == NULL) {
        throw runtime_error("Queue is empty.");
    }
    Node<E>* temp = front->next;
    E result = temp->element;
    front->next = front->next->next;
    delete temp;
    size--;
    if (front->next == NULL) {
        rear = front;
    }
    return result;
}
```

El método puede fallar si la cola se encuentra vacía. Esta verificación también puede hacerse mediante el atributo size.

```
void enqueue(E element) {
    rear->next = new Node<E>(element);
    rear = rear->next;
    size++;
}
E dequeue() throw(runtime_error) {
    if (front->next == NULL) {
        throw runtime_error("Queue is empty.");
    }
    Node<E>* temp = front->next;
    E result = temp->element;
    front->next = front->next->next;
    delete temp;
    size--;
    if (front->next == NULL) {
        rear = front;
    }
    return result;
}
```

Se utiliza un puntero temporal para apuntar al nodo que se eliminará.

```
void enqueue(E element) {
    rear->next = new Node<E>(element);
    rear = rear->next;
    size++;
}
E dequeue() throw(runtime_error) {
    if (front->next == NULL) {
        throw runtime_error("Queue is empty.");
    }
    Node<E>* temp = front->next;
    E result = temp->element;
    front->next = front->next->next;
    delete temp;
    size--;
    if (front->next == NULL) {
        rear = front;
    }
    return result;
}
```

Se guarda el elemento del nodo para retornarlo al final del método.

```
void enqueue(E element) {
    rear->next = new Node<E>(element);
    rear = rear->next;
    size++;
}
E dequeue() throw(runtime_error) {
    if (front->next == NULL) {
        throw runtime_error("Queue is empty.");
    }
    Node<E>* temp = front->next;
    E result = temp->element;
    front->next = front->next->next;
    delete temp;
    size--;
    if (front->next == NULL) {
        rear = front;
    }
    return result;
}
```

Se actualiza el puntero next del nodo especial al inicio de la cola.

```
void enqueue(E element) {
    rear->next = new Node<E>(element);
    rear = rear->next;
    size++;
}
E dequeue() throw(runtime_error) {
    if (front->next == NULL) {
        throw runtime_error("Queue is empty.");
    }
    Node<E>* temp = front->next;
    E result = temp->element;
    front->next = front->next->next;
    delete temp;
    size--;
    if (front->next == NULL) {
        rear = front;
    }
    return result;
}
```

Se borra el nodo y se disminuye la cantidad de elementos en la cola.

```
void enqueue(E element) {
    rear->next = new Node<E>(element);
    rear = rear->next;
    size++;
}
E dequeue() throw(runtime_error) {
    if (front->next == NULL) {
        throw runtime_error("Queue is empty.");
    }
    Node<E>* temp = front->next;
    E result = temp->element;
    front->next = front->next->next;
    delete temp;
    size--;
    if (front->next == NULL) {
        rear = front;
    }
    return result;
}
```

En caso de que se haya eliminado el último elemento de la cola es necesario actualizar rear para que apunte al nodo especial.


```
E frontValue() throw(runtime_error) {  
    if (front->next == NULL) {  
        throw runtime_error("Queue is empty.");  
    }  
    return front->next->element;  
}  
void clear() {  
    Node<E>* temp = front->next;  
    while (temp != NULL) {  
        front->next = front->next->next;  
        delete temp;  
        temp = front->next;  
    }  
    rear = front;  
    size = 0;  
}
```

El método puede fallar si la cola se encuentra vacía.
También se puede verificar el atributo size.

```
E frontValue() throw(runtime_error) {  
    if (front->next == NULL) {  
        throw runtime_error("Queue is empty.");  
    }  
    return front->next->element;  
}  
void clear() {  
    Node<E>* temp = front->next;  
    while (temp != NULL) {  
        front->next = front->next->next;  
        delete temp;  
        temp = front->next;  
    }  
    rear = front;  
    size = 0;  
}
```

Se retorna el elemento almacenado en el primer nodo de la cola.

```
E frontValue() throw(runtime_error) {  
    if (front->next == NULL) {  
        throw runtime_error("Queue is empty.");  
    }  
    return front->next->element;  
}  
void clear() {  
    Node<E>* temp = front->next;  
    while (temp != NULL) {  
        front->next = front->next->next;  
        delete temp;  
        temp = front->next;  
    }  
    rear = front;  
    size = 0;  
}
```

Se utiliza un puntero temporal para apuntar a los nodos que se van borrando.


```
E frontValue() throw(runtime_error) {  
    if (front->next == NULL) {  
        throw runtime_error("Queue is empty.");  
    }  
    return front->next->element;  
}  
void clear() {  
    Node<E>* temp = front->next;  
    while (temp != NULL) {  
        front->next = front->next->next;  
        delete temp;  
        temp = front->next;  
    }  
    rear = front;  
    size = 0;  
}
```

Mientras la cola tenga elementos, se actualiza el puntero al primer nodo y se elimina el nodo apuntado por el temporal. Luego se actualiza temporal para que apunte al primer elemento nuevamente.

```
E frontValue() throw(runtime_error) {  
    if (front->next == NULL) {  
        throw runtime_error("Queue is empty.");  
    }  
    return front->next->element;  
}  
void clear() {  
    Node<E>* temp = front->next;  
    while (temp != NULL) {  
        front->next = front->next->next;  
        delete temp;  
        temp = front->next;  
    }  
    rear = front;  
    size = 0;  
}
```

Se actualiza rear para que apunte al nodo especial
y se actualiza la cantidad de elementos a cero.

```
bool isEmpty() {  
    return size == 0;  
}  
int getSize() {  
    return size;  
}  
};
```



Métodos sencillos que indican si la cola está vacía y el tamaño actual de la cola.

Ejemplo de utilización

```
LinkedQueue<string> q;  
string texto;  
do {  
    getline(cin, texto);  
    q.enqueue(texto);  
} while (texto != "salir");  
while (!q.isEmpty()) {  
    cout << q.dequeue() << endl;  
}
```

Ejercicios con LinkedQueue

- Implementar el método para **imprimir** los contenidos.
- Implementar un método **copiar** que reciba por parámetro otro objeto tipo Queue y que se encargue de copiar los contenidos de la cola actual en la cola enviada. La cola actual no debe cambiar sus elementos.
- Implementar la estructura **Deque** en la cola enlazada.
 - enqueueFront → insertar un elemento en el frente de la cola
 - dequeueRear → eliminar un elemento en el final de la cola
 - rearValue → retorna el elemento al final de la cola

Comparación

- Arreglos

- Acceso a los elementos eficiente
- Borrado de los elementos eficiente
- Tamaño del arreglo es fijo
- No hay *overhead*, índices

- Punteros

- Acceso a los elementos eficiente
- Borrado de los elementos eficiente
- Cola crece dinámicamente
- *Overhead* por manejo de punteros

Ejercicios con colas: Cola de prioridad

- Una cola de prioridad es una estructura que no aplica solamente el concepto de FIFO, si no que cada elemento **tiene asociado una prioridad**
- Se utiliza para guardar y obtener elementos en **orden de prioridad**
- Reglas
 - La prioridad es representada por un **número entero no negativo**
 - La prioridad **0 es la mayor prioridad**, entre más alto el número, menos prioridad
 - Cuando se inserta un elemento, debe **especificarse la prioridad** del mismo
 - Al atender un elemento (pop) se atiende el primer elemento que haya con la **mayor prioridad**
 - Los elementos con la **misma prioridad** deben atenderse en el **orden de llegada** (FIFO)
- Operaciones
 - Constructor: Recibe por parámetro un entero no negativo P que indica la prioridad máxima que va a tener la cola, de manera que las prioridades
 - void insert(E element, int priority): Inserta un nuevo elemento en la estructura con la prioridad indicada por parámetro
 - E min(): Retorna una copia del elemento de mayor prioridad en la cola
 - E removeMin(): Elimina y retorna el mismo elemento que retorna min(), el de mayor prioridad en la cola
 - int getSize(): Retorna la cantidad de elementos que se encuentran en la cola actualmente
 - bool isEmpty(): Retorna verdadero si la cantidad de elementos en la cola es cero, de otro modo retorna falso
- Implementación
 - La cola de prioridad se puede implementar de distintas maneras, pero una de las más eficientes consiste en tener una cola por cada prioridad posible
 - Dado que la cantidad de prioridades es un parámetro del constructor, es importante que la estructura sea dinámica
 - Debe crearse una cola por cada prioridad posible. Si la prioridad máxima es N, entonces se deben crear N+1 colas, desde 0 hasta N
 - Debe crearse un arreglo de colas, donde el índice de cada cola es precisamente el número de la prioridad
 - Cuando se inserta un elemento, se inserta como último elemento en la cola de la prioridad correspondiente
 - Cuando se utiliza el método min(), se busca la cola de menor prioridad que no esté vacía y se retorna el que se encuentra de primero
 - Cuando se utiliza el método removeMin(), también se busca la cola de menor prioridad que no esté vacía y se elimina y retorna el primer elemento de dicha cola

Ejemplo: crear una cola de prioridad con prioridad máxima 5. Observe que la estructura entonces cuenta con 6 colas.

Operaciones:

Insertar "BLB", prioridad 1

Insertar "GND", prioridad 5

Insertar "FRD", prioridad 5

Insertar "LGL", prioridad 0

Insertar "MRG", prioridad 1

Insertar "AWD", prioridad 3

Insertar "GYH", prioridad 0

Insertar "ARG", prioridad 1

Obtener mínimo -> retorna LGL

Al eliminar el mínimo repetidas veces hasta que la estructura está vacía, los elementos se eliminan en este orden:

LGL, GYH, BLB, MRG, ARG, AWD, GND, FRD

0	LGLGYH
1	BLBMRGARG
2	
3	AWD
4	
5	GNDFRD

- El ejercicio consiste en hacer dos implementaciones de la cola de prioridad:
 - **ArrayPriorityQueue**
 - Utilizando la clase **ArrayQueue**
 - **LinkedPriorityQueue**
 - Utilizando la clase **LinkedQueue**
- Ambas clases deben heredar de la clase abstracta **PriorityQueue**
- Ambas clases deben crear un **arreglo de colas** en memoria dinámica e implementar los distintos métodos
- **Programa de prueba**
 - Debe permitir al usuario **escoger** cuál de los dos tipos de cola **desea utilizar** y la cantidad de **prioridades** que desea utilizar
 - Los elementos deben ser **strings**
 - El programa solicita al usuario repetidas veces la operación que desea realizar (ejecutar cualquiera de los métodos) hasta que elija 'salir'
 - Debe mostrarse en pantalla cada vez los contenidos de la cola de prioridad

```
template <typename E>
class PriorityQueue {
private:
    void operator=(const PriorityQueue&) {}
    PriorityQueue(const PriorityQueue& other) {}
public:
    PriorityQueue() {}
    virtual ~PriorityQueue() {}
    virtual void insert(E element, int priority) = 0;
    virtual E min() = 0;
    virtual E removeMin() = 0;
    virtual int getSize() = 0;
    virtual bool isEmpty() = 0;
};
```



Pilas y Colas

Mauricio Avilés