[Γα=Ω5]

Estructuras de Datos Lineales

Por Alan Martinez

¿Qué son?

Las estructuras de datos lineales son aquellas donde los elementos **se organizan secuencialmente**, uno tras otro. Cada elemento tiene un único predecesor (excepto el primero) y un único sucesor (excepto el último). Las operaciones típicas incluyen inserción, eliminación, búsqueda y actualización.

Arrays (Arreglos)

Un arreglo es una colección de elementos, todos del mismo tipo, organizados en posiciones contiguas de memoria. Los arreglos permiten acceso aleatorio, lo que significa que se puede acceder a cualquier elemento directamente a través de su índice. Desde c++ 11 existe la libreria `<array>` la cual incluye una implentación constante de array s.

Complejidad de operaciones comunes:

- Acceso y Modificación: O(1)
- Inserción/Eliminación en el medio: O(n) (donde n es el tamaño del arreglo)

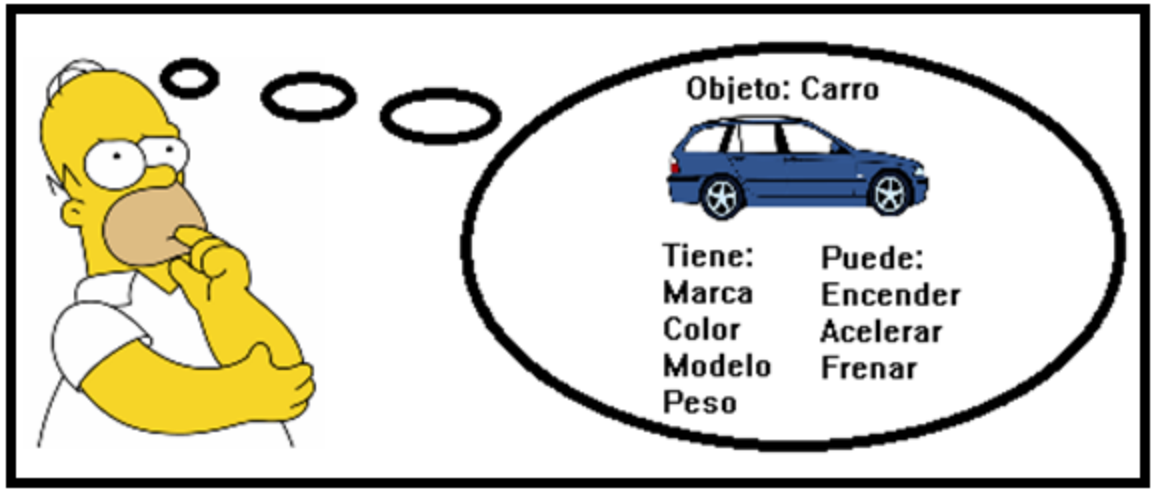
Ejemplo de uso de arrays

```cpp
// Array tradicional de C
int arr[5] = {10, 20, 30, 40, 50}; // Array de C

// Acceso directo a un elemento  O(1)
cout << "El tercer elemento del array tradicional es: " << arr[2] << endl; // Salida: 30, O(1)

// Modificación de un elemento en el array tradicional de C  O(1)
arr[2] = 35;
cout << "El nuevo tercer elemento del array tradicional es: " << arr[2] << endl; // Salida: 35, O(1)

// Usando std::array para un array constante
array<int, 5> arrConstante = {10, 20, 30, 40, 50}; // Declaración e inicialización

// Acceso a elementos en std::array  O(1)
cout << "El segundo elemento del std::array es: " << arrConstante.at(1) << endl; // Salida: 20, O(1)

// Modificación de un elemento en std::array  O(1)
arrConstante[3] = 45;
cout << "El nuevo cuarto elemento del std::array es: " << arrConstante[3] << endl; // Salida: 45, O(1)
```

Pero Antes...

Struct

Un `struct` (estructura) de C/C++ es un tipo de dato que agrupa diferentes variables bajo un solo nombre. A diferencia de los arreglos, los structs pueden contener **variables de diferentes tipos** (`int`, `float`, `char`, etc.). Son muy útiles cuando necesitamos representar **objetos** o **entidades del mundo real** con múltiples **atributos**.



Ejemplo:

```
struct Persona {
    string nombre;
    int edad;
    float altura;
};

int main() {
    Persona persona1; // Creamos una instancia de Persona
    persona1.nombre = "Juan";
    persona1.edad = 25;
    persona1.altura = 1.75;
    cout << "Nombre: " << persona1.nombre << endl;
    cout << "Edad: " << persona1.edad << endl;
    cout << "Altura: " << persona1.altura << " metros" << endl;
    return 0;
}
```

Arreglo de Structs

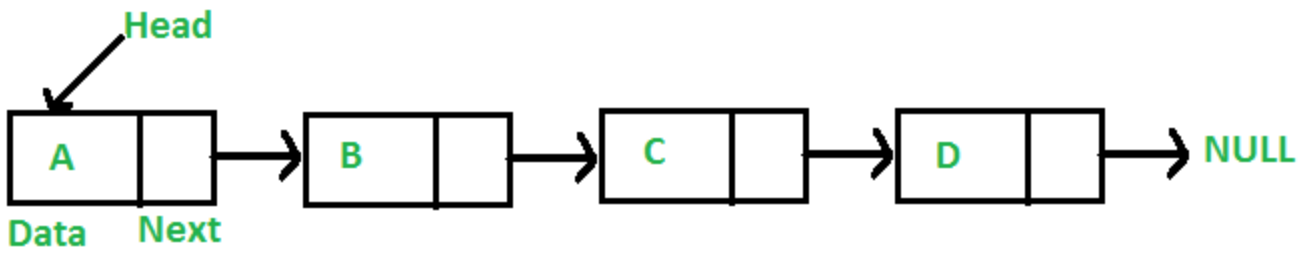
```
int main() {
    Persona personas[3] = {{"Ana", 20}, {"Luis", 22}, {"Carlos", 24}};

    for(int i = 0; i < 3; i++) {
        cout << "Nombre: " << personas[i].nombre << ", Edad: " << personas[i].edad << endl;
    }

    return 0;
}
```

Listas Enlazadas

Una **lista enlazada** es una estructura de datos lineal compuesta por una **colección de nodos**, donde cada nodo almacena dos componentes:

1. **Dato**: El valor que contiene el nodo.
2. **Referencia** (puntero): Un puntero que apunta al siguiente nodo en la secuencia.

A diferencia de los arrays, las listas enlazadas no tienen un tamaño fijo y permiten una **inserción** y **eliminación** eficiente O(1) en cualquier posición sin necesidad de reorganizar la memoria. Sin embargo, **no permiten acceso aleatorio** a los elementos, lo que implica que para acceder a un nodo específico, es necesario recorrer la lista desde el comienzo O(n).



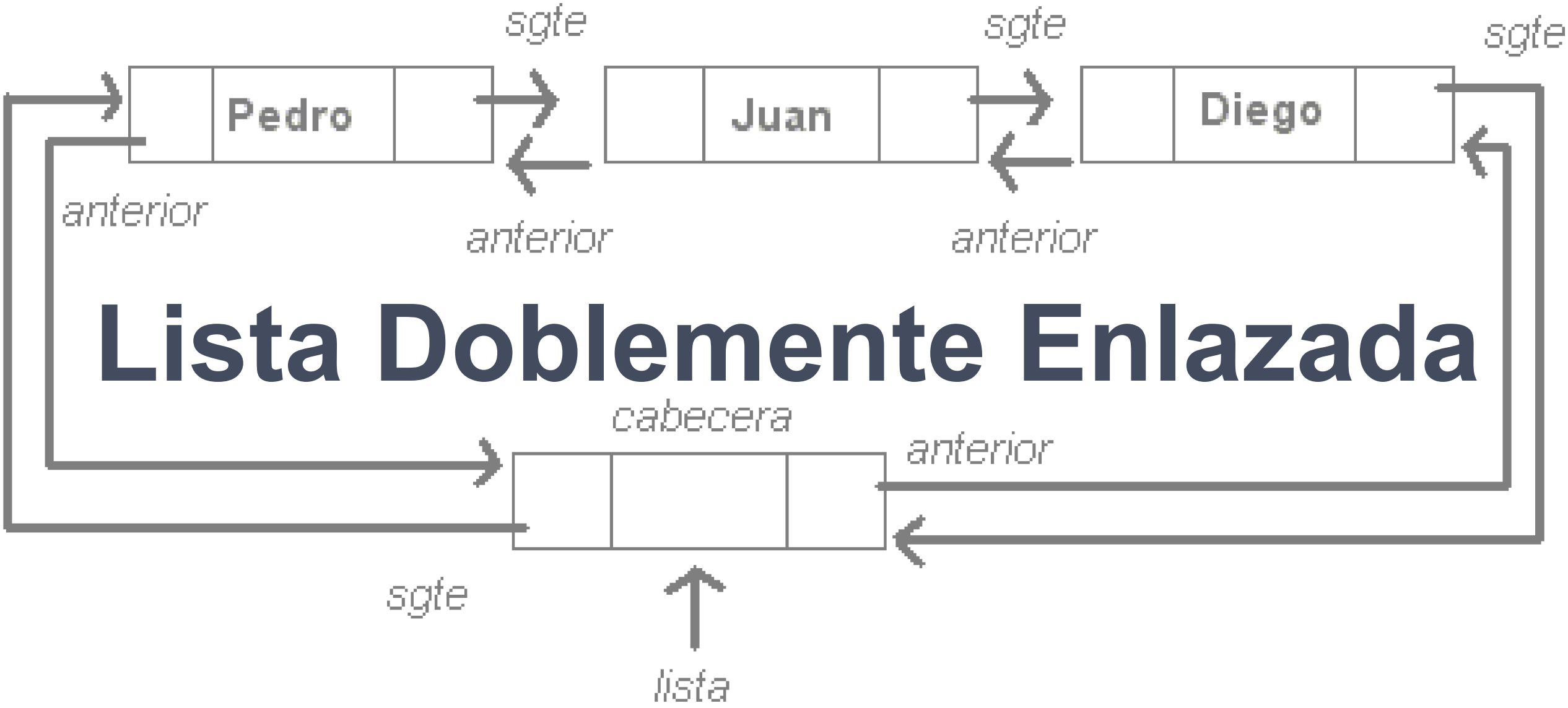
Implementación en C

```
struct Node {
    int data;
    Node* next;
};

int main() {
    Node* head = nullptrptr; // Inicializamos la lista vacía

    insert(&head, 10);
    insert(&head, 20);
    insert(&head, 30);
    cout << "Lista enlazada: ";
    printList(head); // Salida esperada: 30 20 10
    return 0;
}
```

Funciones Auxiliares

```
void insert(Node** head, int newData) {
    Node* newNode = new Node(); // Crear un nuevo nodo
    newNode->data = newData;    // Asignar el valor al nodo
    newNode->next = (*head);    // Apuntar al anterior primer nodo
    (*head) = newNode;          // Actualizar la cabeza de la lista
}
```

```
void printList(Node* node) {
    while (node != nullptr) {
        cout << node->data << " ";
        node = node->next; // Avanzar al siguiente nodo
    }
}
```


sgte
sgte
sgte
Pedro
Juan
Diego
anterior
anterior
anterior
Lista Doblemente Enlazada
cabecera
anterior
sgte
lista

```cpp
struct Node {
    int data;
    Node* next;
    Node* prev;
};

int main() {
    Node* head = nullptr;  // Inicializamos la lista vacía
    insertAtHead(&head, 10);
    insertAtHead(&head, 20);
    insertAtTail(&head, 5);
    insertAtTail(&head, 1);

    printListForward(head); // Salida: 20 10 5 1

    return 0;
}
```

Funciones para Insertar (Cabeza)

```
void insertAtHead(Node** head, int data) {
    Node* newNode = new Node();  // Crear nuevo nodo
    newNode->data = data;        // Asignar valor
    newNode->next = (*head);     // El siguiente del nuevo nodo será el head actual
    newNode->prev = nullptr;        // No hay nodo anterior al head
    if ((*head) != nullptr) // Si la lista no está vacía
        (*head)->prev = newNode;
    (*head) = newNode; // Cambiamos el head para que apunte al nuevo nodo
}
```

Funciones para Insertar (Cola)

```
void insertAtTail(Node** head, int data) {
    Node* newNode = new Node();  // Crear nuevo nodo
    newNode->data = data;        // Asignar valor
    newNode->next = nullptr;        // El siguiente del nuevo nodo es nullptr porque será el último
    if (*head == nullptr) { // Si la lista está vacía
        newNode->prev = nullptr;
        (*head) = newNode;
        return;
    }
    Node* temp = *head; // Si no está vacía, recorremos hasta el final
    while (temp->next != nullptr)
        temp = temp->next;
    temp->next = newNode; // Enlazamos el nuevo nodo al final de la lista
    newNode->prev = temp;
}
```

Imprimir Lista:

```
void printListForward(Node* node) {
    cout << "Lista en orden hacia adelante: ";
    while (node != nullptr) {
        cout << node->data << " - ";
        node = node->next;
    }
    cout << endl;
}
```

Listas doblemente enlazada en C++

En C++, podemos utilizar `std::list` de la libreria `<list>` de la STL que proporciona una lista doblemente enlazada.

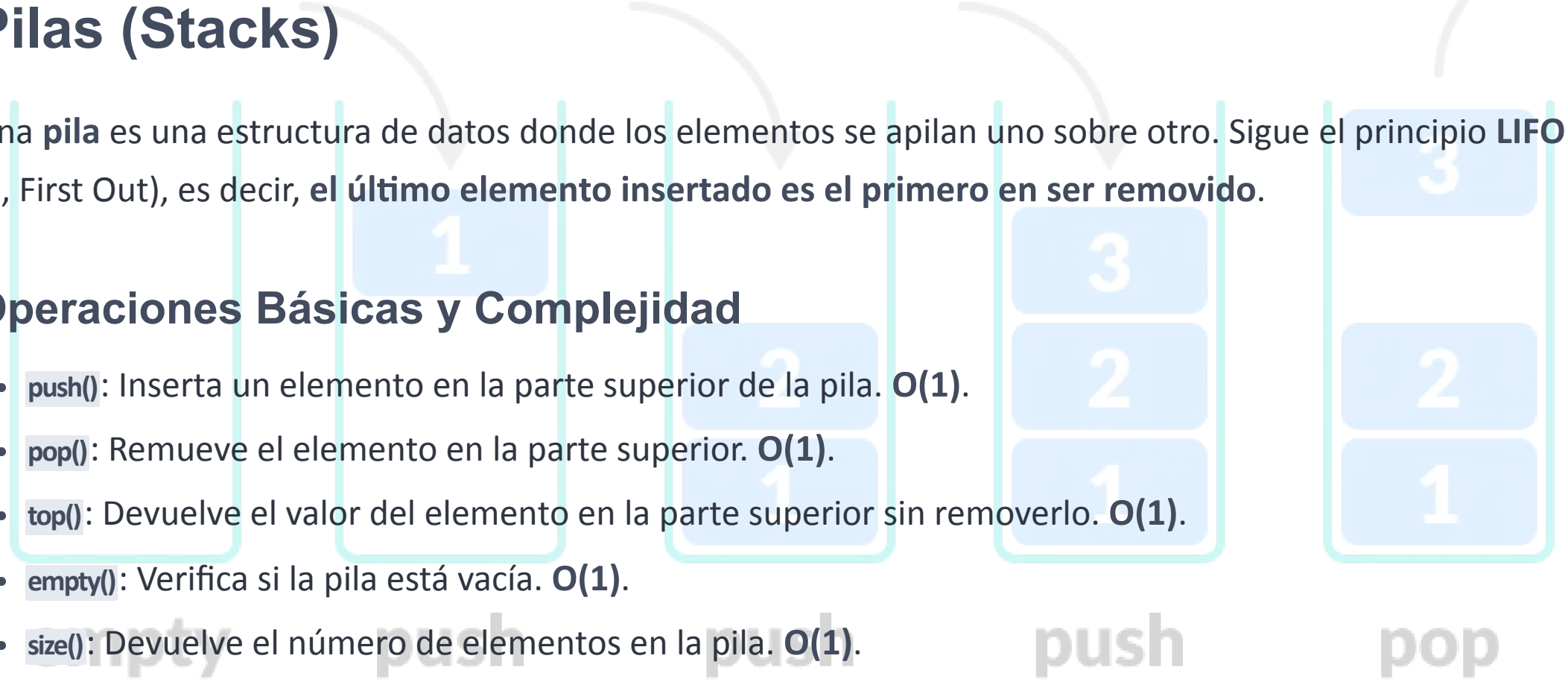
```
std::list<int> myList = {10, 20, 30}; // Crear una lista con elementos

myList.push_front(40); // O(1) - Insertar al frente
myList.push_back(50);  // O(1) - Insertar al final
myList.pop_front();    // O(1) - Eliminar el primer elemento
myList.pop_back();     // O(1) - Eliminar el último elemento

std::cout << "Lista final: ";
for (int n : myList)
    std::cout << n << " "; // O(n) - Recorrer la lista

std::cout << "\nTamaño de la lista: " << myList.size(); // O(n) - Consultar tamaño de la lista
```

Pilas (Stacks)

Una **pila** es una estructura de datos donde los elementos se apilan uno sobre otro. Sigue el principio **LIFO** (Last In, First Out), es decir, **el último elemento insertado es el primero en ser removido**.

Operaciones Básicas y Complejidad

- `push()`: Inserta un elemento en la parte superior de la pila. **O(1)**.
- `pop()`: Remueve el elemento en la parte superior. **O(1)**.
- `top()`: Devuelve el valor del elemento en la parte superior sin removerlo. **O(1)**.
- `empty()`: Verifica si la pila está vacía. **O(1)**.
- `size()`: Devuelve el número de elementos en la pila. **O(1)**.



Implementación en C++

En C++ que podemos utilizar la clase `std::stack` de la libreria `<stack>`.

```
stack<int> s;

s.push(10);  // O(1) - Insertar 10
s.push(20);  // O(1) - Insertar 20
s.push(30);  // O(1) - Insertar 30

cout << "Elemento superior: " << s.top() << endl; // O(1) - Obtener el valor del elemento superior (Salida: 30)

s.pop();     // O(1) - Eliminar 30
cout << "Nuevo elemento superior: " << s.top() << endl; // O(1) - Obtener el nuevo elemento superior (Salida: 20)
```

Colas (Queues)

Una **cola** es una estructura de datos que sigue el principio **FIFO** (First In, First Out), es decir, **el primer elemento insertado es el primero en ser removido**. Las colas son útiles en situaciones donde se requiere un procesamiento en orden, como en sistemas de espera o procesamiento por turnos.

Operaciones Básicas y Complejidad

- `enqueue()` (`push()` en STL): Inserta un elemento al final de la cola. **O(1)**.
- `dequeue()` (`pop()` en STL): Remueve el primer elemento de la cola. **O(1)**.
- `front()`: Devuelve el valor del primer elemento sin removerlo. **O(1)**.
- `empty()`: Verifica si la cola está vacía. **O(1)**.
- `size()`: Devuelve el número de elementos en la cola. **O(1)**.

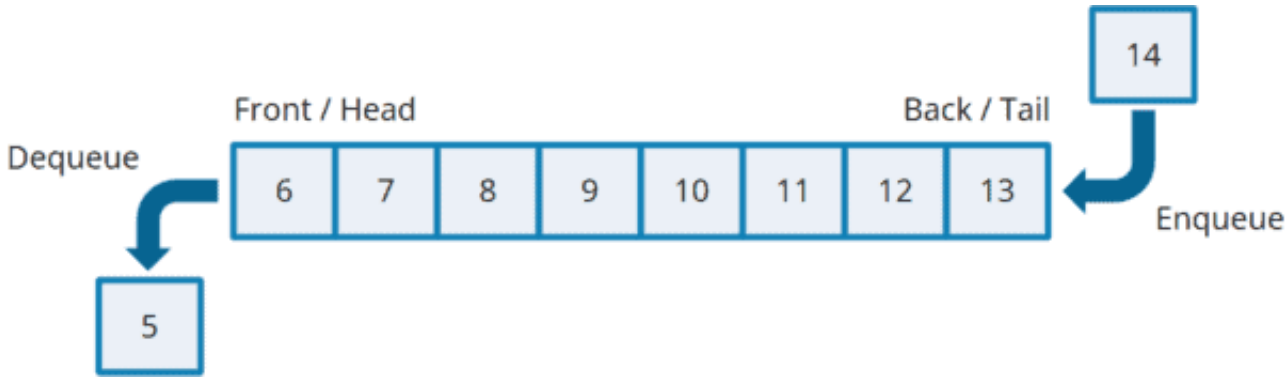
Implementación en C++

En C++ que podemos utilizar la clase `std::queue` de la libreria `<queue>`.

```
queue<int> q;

q.push(10);  // O(1) - Insertar 10 al final de la cola
q.push(20);  // O(1) - Insertar 20 al final de la cola
q.push(30);  // O(1) - Insertar 30 al final de la cola

cout << "Primer elemento: " << q.front() << endl; // O(1) - Obtener el primer elemento (Salida: 10)

q.pop();     // O(1) - Eliminar el primer elemento (10)
cout << "Nuevo primer elemento: " << q.front() << endl; // O(1) - Obtener el nuevo primer elemento (Salida: 20)
```

Deques (Double-Ended Queues)

Un **deque** es una estructura de datos que permite la inserción y eliminación de elementos **tanto por el principio como por el final**. Es una **combinación de una pila y una cola**, ofreciendo flexibilidad para operar en ambos extremos.



Operaciones Básicas

- `push_back()` / `push_front()`: **O(1)**
- `pop_back()` / `pop_front()`: **O(1)**.
- `front()` / `back()`: **O(1)**.
- `size()`: **O(1)**.
- `empty()`: **O(1)**.

Implementación en C++

En C++ que podemos utilizar la clase `std::deque` de la libreria `<deque>`.

```
deque<int> d;

d.push_back(10);  // O(1) - Insertar 10 al final
d.push_back(20);  // O(1) - Insertar 20 al final
d.push_front(30); // O(1) - Insertar 30 al inicio

cout << "Primer elemento: " << d.front() << endl; // O(1) - Obtener el primer elemento (Salida: 30)
cout << "Último elemento: " << d.back() << endl;  // O(1) - Obtener el último elemento (Salida: 20)

d.pop_front(); // O(1) - Eliminar el primer elemento (30)
cout << "Nuevo primer elemento: " << d.front() << endl; // O(1) - Obtener el nuevo primer elemento (Salida: 10)
```

Problemas

- **673 A** Bear and Game
- **236 A** Boy or Girl

Referencias

- GeeksforGeeks. (2023). *Deque in C++ Standard Template Library (STL)*. Recuperado de https://www.geeksforgeeks.org/deque-cpp-stl/
- GeeksforGeeks. (2024) *Stack in C++ STL*. Recuperado de https://www.geeksforgeeks.org/stack-in-cpp-stl/
- GeeksforGeeks. (2024). *List in C++ Standard Template Library (STL)*. Recuperado de https://www.geeksforgeeks.org/list-cpp-stl/
- GeeksforGeeks. (2024). *Queue in C++ Standard Template Library (STL)*. Recuperado de https://www.geeksforgeeks.org/queue-cpp-stl/
- harendrakumar123. (2024). *Doubly Linked List Tutorial*. Recuperado de https://www.geeksforgeeks.org/doubly-linked-list/
- harendrakumar123. (2024). *Singly Linked List Tutorial*. Recuperado de https://www.geeksforgeeks.org/singly-linked-list-tutorial/
- sharma, S. (2022). *STD::array in C++*. Recuperado de https://www.geeksforgeeks.org/stdarray-in-cpp/