4. Estructura de Datos: Lista Enlazada (Linked List)

Una **lista enlazada** es una colección de nodos donde cada nodo contiene un valor y un puntero al siguiente nodo en la secuencia. No requiere memoria contigua y permite inserciones y eliminaciones eficientes si se conoce la posición.

—

0.1. Características de las Listas Enlazadas

- **Tamaño Dinámico:** Crece o decrece en tiempo de ejecución.
- **Inserción/Eliminación en Tiempo Constante:** $\mathcal{O}(1)$ si se tiene el puntero al nodo previo.
- **Recorrido Secuencial:** Acceso por índice en $\mathcal{O}(n)$.
- **No Contiguo en Memoria:** Cada nodo se asigna dinámicamente.

—

0.2. Operaciones Básicas sobre Listas Enlazadas

1. **Recorrido:** Visitar nodos desde la cabeza hasta `nullptr`. $\mathcal{O}(n)$.
2. **Inserción al Frente:** Crear un nuevo nodo y actualizar la cabeza. $\mathcal{O}(1)$.
3. **Inserción al Final:** Recorrer hasta el último nodo y enlazar uno nuevo. $\mathcal{O}(n)$ (o $\mathcal{O}(1)$ si se guarda puntero a cola).
4. **Eliminación del Frente:** Actualizar la cabeza al siguiente nodo y liberar el antiguo. $\mathcal{O}(1)$.
5. **Eliminación en Medio:** Con puntero al nodo anterior, actualizar su `next` y liberar el nodo objetivo. $\mathcal{O}(1)$ tras búsqueda.
6. **Búsqueda:** Recorrer nodos comparando valores. $\mathcal{O}(n)$.

—

0.3. Implementación en C++

Lista Simple (Singly Linked List)

Listing 1: Implementacion básica de lista enlazada simple

```
1 #include <iostream>
2 using namespace std;
3
4 struct Node {
```

```
    int data;
    Node* next;
    Node(int x) : data(x), next(nullptr) {}
};

class LinkedList {
public:
    Node* head;
    LinkedList() : head(nullptr) {}

    // Insercion al frente (Se puede hacer directamente)
    void push_front(int x) {
        Node* nuevo = new Node(x);
        nuevo->next = head;
        head = nuevo;
    }

    // Insercion al final (Debe recorrerse la lista
    // entera para llegar al ultimo nodo. Si se quiere
    // anadir un nodo intermedio, el proceso es similar)
    void push_back(int x) {
        Node* nuevo = new Node(x);
        if (!head) {
            head = nuevo;
            return;
        }
        Node* cur = head;
        while (cur->next) cur = cur->next;
        cur->next = nuevo;
    }

    // Eliminacion del frente
    void pop_front() {
        if (!head) return;
        Node* tmp = head;
        head = head->next;
        delete tmp;
    }

    // Busqueda de un valor
    bool find(int x) {
        Node* cur = head;
        while (cur) {
            if (cur->data == x) return true;
            cur = cur->next;
        }
        return false;
    }

    // Recorrido e impresion
```

```
    void print() {
        Node* cur = head;
        while (cur) {
            cout << cur->data << " -> ";
            cur = cur->next;
        }
        cout << "nullptr\n";
    }
};

int main() {
    LinkedList lista;
    lista.push_back(10);
    lista.push_front(5);
    lista.push_back(20);
    lista.print();          // 5 -> 10 -> 20 -> nullptr
    cout << lista.find(10) << endl; // 1 (true)
    lista.pop_front();
    lista.print();          // 10 -> 20 -> nullptr
    return 0;
}
```

0.4. Ventajas y Desventajas

Ventajas:

- Inserciones y eliminaciones en $\mathcal{O}(1)$ con puntero previo.
- Tamaño dinámico sin prever capacidad.

Desventajas:

- Acceso secuencial lento: $\mathcal{O}(n)$. Esto puede reducirse usando una lista doblemente enlazada, que es igual pero cada nodo tiene un puntero al siguiente y al anterior. De esta forma, se puede empezar a recorrer la lista desde el principio o desde el final.
- Sobrecarga de memoria por punteros.
- Localidad de caché pobre.

0.5. Aplicaciones Comunes

- Implementación de pilas y colas basadas en nodos.
- Estructuras de listas de adyacencia en grafos.

- Gestión de memoria libre (free lists).
- Algoritmos con inserciones/eliminaciones frecuentes.