Estructuras enlazadas con asignación dinámica en memoria

Estructuras Lineales

Las **estructuras de datos lineales** son aquellas en las que los elementos ocupan lugares sucesivos en la estructura y cada uno de ellos tiene un único sucesor y un único predecesor, es decir, sus elementos están ubicados uno al lado del otro relacionados en forma lineal.

Vamos a trabajar con las siguientes:

- Pila
- Cola
- Lista

Estructura de Tipo Pila

Una pila es una estructura de datos lineal que sigue el principio 'último en entrar, primero en salir' (LIFO, por sus siglas en inglés). Similar a como “apilamos” libros. Se hace una pila de libros donde el último en apilar es el primero en salir si saco un libro.

```
#include <iostream>

// Definición de la estructura Nodo
struct Nodo {
    int dato;
    Nodo* siguiente;
};

// Función para crear un nuevo nodo
Nodo* crearNodo(int valor) {
    Nodo* nuevoNodo = new Nodo;
    nuevoNodo->dato = valor;
    nuevoNodo->siguiente = nullptr;
    return nuevoNodo;
}
```

```
// Función para insertar un elemento en la pila
void push(Nodo*& pila, int valor) {
    Nodo* nuevoNodo = crearNodo(valor);
    nuevoNodo->siguiente = pila;
    pila = nuevoNodo;
}

// Función para eliminar y obtener el elemento en la cima de la pila
int pop(Nodo*& pila) {
    if (pila == nullptr) {
        std::cout << "La pila está vacía." << std::endl;
        return -1; // Valor de error
    }
    int valor = pila->dato;
    Nodo* temp = pila;
    pila = pila->siguiente;
    delete temp;
    return valor;
}

// Función para verificar si la pila está vacía
bool isEmpty( Nodo* pila) {
    return pila == nullptr;
}

int main() {
    Nodo* pila = nullptr; // Inicializar la pila

    // Apilar elementos en la pila
    push(pila, 10);
    push(pila, 20);
    push(pila, 30);

    std::cout << "Elementos en la pila:" << std::endl;
    while (!isEmpty(pila)) {
        // Mostrar y eliminar el elemento en la cima de la pila
        int valor = pop(pila);
        std::cout << valor << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Estructura de Tipo Cola

Una cola es otra estructura de datos lineal, pero sigue el principio 'primero en entrar, primero en salir' (FIFO). Es decir, es como una cola del supermercado, el primero en entrar a la cola es el primero en salir.

```
#include <iostream>

// Definición de la estructura Nodo
struct Nodo {
    int dato;
    Nodo* siguiente;
};

// Función para crear un nuevo nodo
Nodo* crearNodo(int valor) {
    Nodo* nuevoNodo = new Nodo;
    nuevoNodo->dato = valor;
    nuevoNodo->siguiente = nullptr;
    return nuevoNodo;
}

// Función para insertar un elemento en la cola
void encolar(Nodo*& frente, Nodo*& final, int valor) {
    Nodo* nuevoNodo = crearNodo(valor);
    if (final == nullptr) {
        frente = final = nuevoNodo;
    } else {
        final->siguiente = nuevoNodo;
        final = nuevoNodo;
    }
}

// Función para eliminar y obtener el elemento del frente de la cola
int desencolar(Nodo*& frente, Nodo*& final) {
    if (frente == nullptr) {
        std::cerr << "La cola está vacía." << std::endl;
        return -1; // Valor de error
    }
    int valor = frente->dato;
    Nodo* temp = frente;
    frente = frente->siguiente;
    if (frente == nullptr) {
        final = nullptr; // Si se elimina el último elemento, actualizar 'final'
    }
    delete temp;
    return valor;
}

// Función para verificar si la cola está vacía
bool isEmpty(Nodo* frente) {
    return frente == nullptr;
}

int main() {
    Nodo* frente = nullptr;
    Nodo* final = nullptr;
```

```
    // Encolar elementos
    encolar(frente, final, 10);
    encolar(frente, final, 20);
    encolar(frente, final, 30);

    std::cout << "Elementos en la cola:" << std::endl;
    while (!isEmpty(frente)) {
        // Desencolar y mostrar el elemento del frente de la cola
        int valor = desencolar(frente, final);
        std::cout << valor << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

4. Estructura de Tipo Lista

Las listas enlazadas son estructuras de datos que consisten en nodos que se conectan unos a otros para formar una secuencia. Es decir, cada elemento conoce a su siguiente en la lista.

```
#include <iostream>

// Definición de la estructura Nodo para listas enlazadas simples
struct Nodo {
    int dato;
    Nodo* siguiente;
};

// Función para crear un nuevo nodo
Nodo* crearNodo(int valor) {
    Nodo* nuevoNodo = new Nodo;
    nuevoNodo->dato = valor;
    nuevoNodo->siguiente = nullptr;
    return nuevoNodo;
}

// Función para imprimir una lista enlazada simple
void imprimirListaSimple(Nodo* inicio) {
    Nodo* actual = inicio;
    while (actual != nullptr) {
        std::cout << actual->dato << " ";
        actual = actual->siguiente;
    }
    std::cout << std::endl;
}
```

```
int main() {
    // Crear una lista enlazada simple
    Nodo* listaSimple = crearNodo(10);
    listaSimple->siguiente = crearNodo(20);
    listaSimple->siguiente->siguiente = crearNodo(30);

    // Imprimir la lista enlazada simple
    std::cout << "Lista Simple: ";
    imprimirListaSimple(listaSimple);

    // Liberar la memoria al final del programa
    delete listaSimple->siguiente->siguiente;
    delete listaSimple->siguiente;
    delete listaSimple;

    return 0;
}
```

5. Patrones de Operación en Listas

Claro, a continuación, te proporcionaré ejemplos de operaciones comunes en listas enlazadas simples: carga sin restricciones, carga sin repetir, búsqueda, recorrido y eliminación de nodos.

Carga sin restricciones:
Cargar elementos en una lista enlazada simple sin restricciones significa agregar nodos al final de la lista de acuerdo con la secuencia en la que se ingresan los elementos.

```
Nodo* cargarLista(Nodo* inicio, int valor) {
    Nodo* nuevoNodo = crearNodo(valor);
    if (inicio == nullptr) {
        inicio = nuevoNodo;
    } else {
        Nodo* actual = inicio;
        while (actual->siguiente != nullptr) {
            actual = actual->siguiente;
        }
        actual->siguiente = nuevoNodo;
    }
    return inicio;
}
```

Carga sin repetir:
Cargar elementos en una lista enlazada simple sin repetir significa verificar si el

elemento ya existe en la lista antes de agregarlo nuevamente.

```
bool existeElemento(Nodo* inicio, int valor) {
    Nodo* actual = inicio;
    while (actual != nullptr) {
        if (actual->dato == valor) {
            return true;
        }
        actual = actual->siguiente;
    }
    return false;
}

Nodo* cargarSinRepetir(Nodo* inicio, int valor) {
    if (!existeElemento(inicio, valor)) {
        return cargarLista(inicio, valor);
    }
    return inicio;
}
```

Búsqueda

Buscar un elemento en una lista enlazada simple implica recorrer la lista para encontrar el valor deseado.

```
Nodo* buscarElemento(Nodo* inicio, int valor) {
    Nodo* actual = inicio;
    while (actual != nullptr) {
        if (actual->dato == valor) {
            return actual; // Devuelve el nodo que contiene el valor
        }
        actual = actual->siguiente;
    }
    return nullptr; // El valor no se encontró en la lista
}
```

Recorrido:

Recorrer una lista enlazada simple significa visitar cada elemento de la lista y realizar alguna acción, como mostrar su valor.

```
void recorrerLista(Nodo* inicio) {
    Nodo* actual = inicio;
    while (actual != nullptr) {
        std::cout << actual->dato << " ";
        actual = actual->siguiente;
```

```
    }
    std::cout << std::endl;
}
```

Eliminación de nodos:

Eliminar un nodo en una lista enlazada simple significa eliminar un elemento específico de la lista. Aquí se muestra cómo eliminar un nodo con un valor dado:

```
Nodo* eliminarNodo(Nodo* inicio, int valor) {
    if (inicio == nullptr) {
        return nullptr; // Lista vacía, no hay nada que eliminar
    }
    if (inicio->dato == valor) {
        Nodo* temp = inicio;
        inicio = inicio->siguiente;
        delete temp;
        return inicio; // Devuelve la lista actualizada
    }
    Nodo* actual = inicio;
    while (actual->siguiente != nullptr) {
        if (actual->siguiente->dato == valor) {
            Nodo* temp = actual->siguiente;
            actual->siguiente = actual->siguiente->siguiente;
            delete temp;
            return inicio; // Devuelve la lista actualizada
        }
        actual = actual->siguiente;
    }
    return inicio; // Valor no encontrado en la lista
}
```

Insertar ordenado

```
// Función para insertar un elemento de manera ordenada en la lista
Nodo* insertarOrdenado(Nodo* inicio, int valor) {
    Nodo* nuevoNodo = crearNodo(valor);

    // Caso especial: si la lista está vacía o el valor es menor que el primer elemento
    if (inicio == nullptr || valor < inicio->dato) {
        nuevoNodo->siguiente = inicio;
        inicio = nuevoNodo;
        return inicio;
    }

    Nodo* actual = inicio;
    while (actual->siguiente != nullptr && actual->siguiente->dato < valor) {
        actual = actual->siguiente;
```

```
    }

    nuevoNodo->siguiente = actual->siguiente;
    actual->siguiente = nuevoNodo;
    return inicio;
}
```