

## Ejercicio 1:

Se puede modificar el siguiente código para que no sea necesario escribir -1 para finalizar la lista y baste con presionar Enter?

```
#include <iostream>
#include "LinkedList.h"

using namespace std;

int main() {
    LinkedList list;
    int value;

    cout << "Ingrese los valores de la lista (Escriba -1 para finalizar):
";

    while (true) {
        cin >> value;
        if (value == -1) {
            break;
        } else {
            list.append(value);
        }
    }

    // Esta parte del código no debe ser modificada
    cout << "Lista original: \n";
    list.printList();

    // Llamar al método reverse
    list.reverse();

    cout << "Lista invertida: \n";
    list.printList();
    cout << "Nuevo ";
    list.getHead();
    cout << "Nuevo ";
    list.getTail();

    return 0;
}
```

```
cpp Copiar Editar

#include <iostream>
#include <sstream>
#include "LinkedList.h"

using namespace std;

int main() {
    LinkedList list;
    string input;

    cout << "Ingrese los valores de la lista separados por espacios (Presione Enter para finaliza

    // Leer la línea completa de entrada
    getline(cin, input);

    // Crear un flujo de entrada a partir de la cadena
    istringstream stream(input);
    int value;

    // Extraer valores enteros del flujo
    while (stream >> value) {
        list.append(value);
    }

    // Esta parte del código no debe ser modificada
    cout << "Lista original: \n";
    list.printList();

    // Llamar al método reverse
    list.reverse();

    cout << "Lista invertida: \n";
    list.printList();
    cout << "Nuevo ";
    list.getHead();
    cout << "Nuevo ";
    list.getTail();

    return 0;
}
```

## Ejercicio 4:

Cómo se puede implementar en c++ un método string reverseString que devuelva una palabra que ingrese al revés usando stack?

Puedes implementar el método `reverseString` en C++ utilizando una pila (`stack`) de la biblioteca estándar `<stack>`. La idea es ir apilando cada carácter de la cadena y luego desapilarlo para formar la cadena invertida.

Aquí tienes una implementación en C++:

```
cpp Copiar Editar

#include <iostream>
#include <stack>
#include <string>

std::string reverseString(const std::string& input) {
    std::stack<char> charStack;

    // Apilar cada caracter de la cadena
    for (char ch : input) {
        charStack.push(ch);
    }

    std::string reversed;

    // Desapilar los caracteres para formar la cadena invertida
    while (!charStack.empty()) {
        reversed += charStack.top();
        charStack.pop();
    }

    return reversed;
}

int main() {
    std::string palabra;
    std::cout << "Ingrese una palabra: ";
    std::cin >> palabra;

    std::string palabraInvertida = reverseString(palabra);
    std::cout << "Palabra invertida: " << palabraInvertida << std::endl;

    return 0;
}
```

## Ejercicio 5:

Dame un ejemplo de código de c++ de una clase Pila que implementa una cola básica para que soporte prioridades dinámicas. Se deben tener los métodos siguientes: enqueue para insertar elementos con una prioridad específica manteniendo el orden según esta, updatePriority para actualizar la prioridad de un elemento ya existente, reinsertándolo en el lugar correcto, y dequeue y front para manejar siempre el ticket con mayor prioridad. No se permite el uso de estructuras de datos adicionales como arreglos o vectores, y todo debe implementarse utilizando la lista enlazada proporcionada. Además, el programa debe permitir la interacción por consola para gestionar tickets y mostrar el uso efectivo de la cola de prioridad.

Aquí tienes una implementación en C++ de una clase `PriorityQueue` utilizando una lista enlazada simple. Se incluyen los métodos `enqueue`, `updatePriority`, `dequeue`, y `front`, asegurando que la cola mantenga el orden según la prioridad de los elementos. También se incluye una interfaz de consola para gestionar tickets y probar la funcionalidad.

```
</> Priority Queue List

1  #include <iostream>
2  #include <limits>
3  using namespace std;
4
5  struct Node {
6      int data;
7      int priority;
8      Node* next;
9      Node(int d, int p) : data(d), priority(p), next(nullptr) {}
10 };
11
12 class PriorityQueue {
13 private:
14     Node* head;
```

Responde en el chat en su lugar >

Este código implementa una cola de prioridad con una lista enlazada y proporciona un menú interactivo para gestionar tickets. Puedes ejecutarlo y probar distintas operaciones como insertar, actualizar prioridad, procesar tickets y visualizar la cola.

La clase `PriorityQueue` implementa una cola de prioridad usando una lista enlazada simple; la función `enqueue` inserta un nuevo elemento en la cola manteniendo el orden según su prioridad, donde los valores más altos tienen mayor prioridad; la función `dequeue` elimina el elemento con la mayor prioridad; `front` permite obtener el valor del elemento con la mayor prioridad sin eliminarlo; y finalmente, `updatePriority` busca un elemento en la cola, lo elimina y lo reinserta con una nueva prioridad para mantener el orden adecuado dentro de la lista.

## Ejercicio 6:

Dame un ejemplo de código en c++ que implemente un método dentro de la clase HashTable. Este método debe devolver un vector con todas las claves almacenadas en la tabla hash, incluyendo las presentes en listas enlazadas debido a colisiones. Para lograrlo, se debe recorrer cada posición del arreglo dataMap y, si no está vacía, recorrer su lista enlazada para agregar cada clave al vector. Es necesario manejar correctamente los nodos de las listas enlazadas para evitar errores y garantizar que el vector incluya todas las claves almacenadas.

Aquí tienes un ejemplo de código en C++ que implementa una tabla hash con manejo de colisiones mediante listas enlazadas. Se incluye el método `getAllKeys()`, que devuelve un `std::vector<std::string>` con todas las claves almacenadas en la tabla, recorriendo tanto el arreglo `dataMap` como las listas enlazadas de cada posición.

### Implementación:

```
cpp Copiar Editar

#include <iostream>
#include <vector>
#include <list>
#include <string>

class HashTable {
private:
    static const int SIZE = 10; // Tamaño de la tabla hash
    std::vector<std::list<std::pair<std::string, int>>> dataMap; // Vector de Listas para manejar colisiones

    // Función hash para calcular el índice a partir de una clave
    int hashFunction(const std::string &key) {
        int hash = 0;
        for (char ch : key) {
            hash += ch; // Suma los valores ASCII de los caracteres
        }
        return hash % SIZE;
    }

public:
    // Constructor: inicializa la tabla hash con listas vacías
    HashTable() {
        dataMap.resize(SIZE);
    }

    // Método para insertar un par clave-valor en la tabla
    void insert(const std::string &key, int value) {
        int index = hashFunction(key);
        dataMap[index].push_back({key, value});
    }

    // Método para obtener todas las claves almacenadas en la tabla
    std::vector<std::string> getAllKeys() {
        std::vector<std::string> keys;
        for (const auto &bucket : dataMap) { // Recorre cada posición del array
            for (const auto &pair : bucket) { // Recorre la lista enlazada de cada posición
                keys.push_back(pair.first); // Agrega la clave al vector
            }
        }
        return keys;
    }
};
```

```
// Método para insertar un par clave-valor en la tabla
void insert(const std::string &key, int value) {
    int index = hashFunction(key);
    dataMap[index].push_back({key, value});
}

// Método para obtener todas las claves almacenadas en la tabla
std::vector<std::string> getAllKeys() {
    std::vector<std::string> keys;
    for (const auto &bucket : dataMap) { // Recorre cada posición del array
        for (const auto &pair : bucket) { // Recorre la lista enlazada de cada posición
            keys.push_back(pair.first); // Agrega la clave al vector
        }
    }
    return keys;
}
```

```

// Método para imprimir la tabla hash (solo para propósitos de prueba)
void printTable() {
    for (int i = 0; i < SIZE; i++) {
        std::cout << "Bucket " << i << ": ";
        for (const auto &pair : dataMap[i]) {
            std::cout << "[" << pair.first << ": " << pair.second << "]" -> " ";
        }
        std::cout << "NULL\n";
    }
}

};

// Función principal para probar la tabla hash
int main() {
    HashTable hashTable;
    hashTable.insert("apple", 100);
    hashTable.insert("banana", 200);
    hashTable.insert("cherry", 300);
    hashTable.insert("grape", 400);
    hashTable.insert("orange", 500);
    hashTable.insert("lemon", 600);
    hashTable.insert("melon", 700);
    hashTable.insert("berry", 800);
    hashTable.insert("peach", 900);
    hashTable.insert("plum", 1000);

    std::cout << "Tabla Hash:\n";
    hashTable.printTable();

    // Obtener todas las claves almacenadas
    std::vector<std::string> keys = hashTable.getAllKeys();
    std::cout << "\nClaves almacenadas en la tabla hash:\n";
    for (const std::string &key : keys) {
        std::cout << key << " ";
    }
    std::cout << std::endl;

    return 0;
}

```

## Explicación:

### 1. Estructura de datos:

- Se utiliza un vector de listas (`std::vector<std::list<std::pair<std::string, int>>>`) para manejar colisiones con encadenamiento.
- Cada lista almacena pares `<clave, valor>`.

### 2. Métodos principales:

- `hashFunction()`: Calcula el índice de almacenamiento basado en la suma de los valores ASCII de los caracteres de la clave.
- `insert()`: Inserta un par clave-valor en la tabla.
- `getAllKeys()`: Recorre la tabla y extrae todas las claves en un vector.
- `printTable()`: Muestra la tabla hash y las listas enlazadas en cada índice.

### 3. Ejemplo de uso:

- Se insertan varias claves en la tabla.
- Se imprimen los valores almacenados en cada índice.
- Se obtienen e imprimen todas las claves usando `getAllKeys()`.

Cómo puedo empezar para agregar el método `vector<string> keys()` a la siguiente clase:

```
#ifndef HASTABLEH // Preprocesador para evitar múltiples
inclusiones
#define HASHTABLE_H

#include <iostream>
#include "Node.h"

using namespace std;

class HashTable {
private:
    static const int SIZE = 7; // Tamaño fijo de la tabla hash
    Node* dataMap[SIZE]; // Arreglo de punteros a nodos, que
representa la tabla hash

    // Función hash: calcula un índice a partir de la clave
proporcionada
    int hash(string key) {
        int hash = 0;
        for (int i = 0; i < key.length(); i++) {
            int asciiValue = int(key[i]); // Obtiene el valor ASCII del
carácter
            hash = (hash + asciiValue * 23) % SIZE; // Calcula el hash
usando una constante multiplicadora y el módulo del tamaño de la
tabla
        }
        return hash; // Devuelve el índice calculado
    }

public:
    // -----
    // Destructor: libera la memoria de todos los nodos
    // -----
    ~HashTable() {
        for(int i = 0; i < SIZE; i++) {
            Node* head = dataMap[i]; // Apunta al primer nodo de
la lista en esa posición
            Node* temp = head;
            while (head) { // Recorre la lista y libera la memoria de
cada nodo
                head = head->next;
                delete ↓ ip;
            }
        }
    }
};
```

```
#endif // HASTABLE_H
```

Para agregar el método `vector<string> keys()` a la clase `HashTable`, debes recorrer cada posición del arreglo `dataMap` y, si hay elementos en la lista enlazada, recorrerla completamente para extraer cada clave.

Aquí está la implementación del método `keys()`:

Código actualizado con `keys()`:

```
cpp Copiar Editar

#ifndef HASTABLE_H // Preprocesador para evitar múltiples inclusiones
#define HASTABLE_H

#include <iostream>
#include <vector> // Para usar vector
#include "Node.h"

using namespace std;

class HashTable {
private:
    static const int SIZE = 7; // Tamaño fijo de la tabla hash
    Node* dataMap[SIZE] = {nullptr}; // Inicializa la tabla hash con punteros nulos

    // Función hash: calcula un índice a partir de la clave proporcionada
    int hash(string key) {
        int hash = 0;
        for (int i = 0; i < key.length(); i++) {
            int asciiValue = int(key[i]); // Obtiene el valor ASCII del carácter
            hash = (hash + asciiValue * 23) % SIZE; // Calcula el hash usando una constante
        }
        return hash; // Devuelve el índice calculado
    }

public:
    // Destructor: libera la memoria de todos los nodos
    ~HashTable() {
        for (int i = 0; i < SIZE; i++) {
            Node* head = dataMap[i]; // Apunta al primer nodo de la lista en esa posición
            Node* temp = head;
            while (head) { // Recorre la lista y libera la memoria de cada nodo
                head = head->next;
                delete temp;
            }
        }
    }
};
```

 Copiar
  Editar

```

        temp = head;
    }
}

// Método para imprimir el contenido de La tabla hash
void printTable() {
    for(int i = 0; i < SIZE; i++) {
        cout << i << ":" << endl;
        if(dataMap[i]) { // Si hay elementos en esa posición de La tabla
            Node* temp = dataMap[i];
            while (temp) { // Recorre La Lista y muestra Los pares clave-valor
                cout << "  {" << temp->key << ", " << temp->value << "}" << endl;
                temp = temp->next;
            }
        }
    }
}

// Método para agregar un nuevo par clave-valor a La tabla hash
void set(string key, int value) {
    int index = hash(key); // Calcula el índice usando La función hash
    Node* newNode = new Node(key, value); // Crea un nuevo nodo con La clave y el va
    if (dataMap[index] == nullptr) { // Si no hay elementos en esa posición, agrega
        dataMap[index] = newNode;
    } else { // Si ya hay una Lista en esa posición, se agrega el nuevo nodo al fina
        Node* temp = dataMap[index];
        while (temp->next != nullptr) { // Recorre La Lista hasta el último nodo
            temp = temp->next;
        }
        temp->next = newNode; // Agrega el nuevo nodo al final de La Lista
    }
}

// Método para obtener el valor asociado a una clave
int get(string key) {
    int index = hash(key); // Calcula el índice usando La función hash
    Node* temp = dataMap[index];
    while (temp != nullptr) { // Recorre La Lista en esa posición
        if (temp->key == key) return temp->value; // Si encuentra La clave, devuelve
        temp = temp->next;
    }
    return 0; // Si no encuentra La clave, devuelve 0 (valor indicativo)
}

// Método para obtener todas Las claves almacenadas en La tabla hash

```





```

// Método para obtener todas las claves almacenadas en la tabla hash
vector<string> keys() {
    vector<string> keyList;
    for (int i = 0; i < SIZE; i++) { // Recorre cada posición de la tabla
        Node* temp = dataMap[i];
        while (temp != nullptr) { // Recorre la lista enlazada en la posición actual
            keyList.push_back(temp->key); // Agrega la clave al vector
            temp = temp->next; // Avanza al siguiente nodo
        }
    }
    return keyList;
}

};

#endif // HASTABLE_H

```

### Explicación del método `keys()`

1. Recorrer toda la tabla hash (`dataMap[]`):
  - Se usa un bucle `for` para recorrer cada índice de la tabla (de `0` a `SIZE-1`).
2. Recorrer las listas enlazadas:
  - Si hay nodos en `dataMap[i]`, se recorre la lista enlazada con un puntero `temp`.
  - En cada iteración, se agrega la clave (`temp->key`) al vector `keyList`.
3. Devolver el vector con todas las claves:
  - Una vez recorridos todos los índices y listas, se devuelve `keyList`.

El método `keys()` recupera las claves que han sido almacenadas en la tabla hash. Se recorre cada índice del arreglo y en cada posición itera sobre la lista enlazada, para extraer la clave de cada nodo y agregarla al vector. Al final se devuelve el vector con todas las claves y es posible visualizarlas o hacer uso de ellas.