Lab 5

Implementation of Linked List I

A **linked list** is a fundamental data structure in computer science. It mainly allows efficient **insertion** and **deletion** operations compared to arrays. Like arrays, it is also used to implement other data structures like stack, queue and deque.

A **linked list** is a linear data structure that consists of a series of nodes connected by pointers (in C or C++). Each node contains **data** and a **pointer**/**reference** to the next node in the list. Unlike **arrays, linked lists** allow for efficient **insertion** or **removal** of elements from any position in the list, as the nodes are not stored contiguously in memory.

- Templated Linked List Class:

The `List` class is a template, meaning it can hold data of any type (specified when an object of the class is created). This is done by defining the class with `template<class ItemType>`, where `ItemType` represents the data type that the list will hold.

```
//____________________________________________________
// CLASS TEMPLATE DEFINITION FOR LINKED LIST
//____________________________________________________
#include <iostream>
#include<conio.h>

using namespace std;

template<class ItemType>
class List
{
        protected:
                struct node {
                        ItemType info;
                        struct node *next;
                };
                typedef struct node *NODEPTR;
                NODEPTR listptr;
```

- Structure of a Node:

The class contains an internal structure `node`, which defines the elements of the linked list. Each `node` consists of:

- `info`: Stores the actual data of type `ItemType`.
- `next`: A pointer to the next node in the list.

A typedef `NODEPTR` is defined as a pointer to a `node` structure.

```
        public:
                List();
                ~List();
                ItemType emptyList();
                void insertafter(ItemType oldvalue, ItemType newvalue);
                void deleteItem(ItemType oldvalue);
                void push(ItemType newvalue);
                ItemType pop();
    };
```

```
//_______________________________________
// CLASS TEMPLATE IMPLEMENTATION
//_______________________________________

// Default Constructor that initializes a newly created list to empty list.
template<class ItemType>
List<ItemType>::List()
{
	listptr = 0;
}
```

- Class Member Variables:

- `listptr`: A pointer to the head of the linked list (i.e., the first node in the list).

- Constructor and Destructor:

- **Constructor (`List`)**: Initializes the `listptr` to `nullptr`, indicating that the list is empty when first created.

Destructor (`~List`): Deletes the entire list when the list object goes out of scope. It traverses through the list, deleting each node to free memory and prevent memory leaks.

```
template<class ItemType>
List<ItemType>::~List()
{
	NODEPTR p, q;
	if (emptyList())
		exit(0);
	for (p = listptr, q = p->next; p!=0; p = q, q = p->next)
		delete p;
}
```

- `emptyList()` Function:

This function checks if the list is empty by seeing if `listptr` is `nullptr`. It returns `true` if the list is empty, `false` otherwise.

➢ `insertafter()` Function:

This function searches the list for a node with a specified value (`oldvalue`) and inserts a new node with a new value (`newvalue`) right after it. If the node with `oldvalue` is not found, an error is displayed.

```
template<class ItemType>
void List<ItemType>::insertafter(ItemType oldvalue, ItemType newvalue)
{
        NODEPTR p, q;
        for (p = listptr; p != 0 && p->info != oldvalue; p = p->next)
                ;

        if (p == 0)
        {       cout << " ERROR: value sought is not in the list.";
                exit(1);
        }
        q = new node;
        q->info = newvalue;
        q->next = p->next;
        p->next = q;
}
```

```
// Determines if the list is empty.
template<class ItemType>
ItemType List<ItemType>::emptyList()
{
        return (listptr == 0);
}
```

- `push()` Function:

  This function adds a new node to the front of the list. It creates a new node, stores the data (`newvalue`) in the node, and links it as the new head of the list.

```
template<class ItemType>
void List<ItemType>::push(ItemType newvalue)
{
        NODEPTR p;
        p = new node;
        p->info = newvalue;
        p->next = listptr;
        listptr = p;
}
```

- `deleteItem()` Function:

This function searches for a node with the value `oldvalue` and deletes it from the list. It also handles special cases like deleting the head of the list.

```
template<class ItemType>
void List<ItemType>::deleteItem(ItemType oldvalue)
{
        NODEPTR p, q;
        for (q = 0, p = listptr; p != 0 && p->info != oldvalue; q = p, p = p->next)
                ;
        if (p == 0)
        {       cout << " ERROR: value sought is not in the list.";
                exit(1);
        }
        if (q == 0)     listptr = p->next;
        else            q->next = p->next;
        delete p;
}
```

- `pop()` Function:

This function removes the first node from the list and returns its value. It first checks if the list is empty (and throws an error if it is). Then it updates the head of the list to point to the second node, deletes the first node, and returns the value it contained

```
template<class ItemType>
ItemType List<ItemType>::pop()
{
        NODEPTR p;
        ItemType x;
        if (emptyList())
        {       cout << " ERROR: the list is empty.";
                exit(1);
        }
        p = listptr;
        listptr = p->next;
        x = p->info;
        delete p;
        return x;
}
```

- `main()` Function:

This function tests the list functionality by:

1. Pushing an integer (`87`) onto the list.
2. Popping the value and printing it.
3. It waits for user input before terminating.

```
int main()
{
        List<int> l;
        l.push(87);
        cout << l.pop()<<endl;
        getch();
        return 0;
}
```

Exercise 5.1

Write a menu driven program to test the linked list class

Exercise 5.2

Assume the following specifications of a node of linked structure and the class

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

class LinkedStr
{
private:
Node* ptr;


public:
Node* ptr;

// Constructor. Initiallize ptr to NULL.
LinkedStr();

// Destructor. Remove all the nodes from dynamic memory
~LinkedStr();

// Create a linked structure of length len pointed to by ptr.
// The values of the info part are input from the keyboard void
makeStr(int len);

// Display all the elements of the linked structure pointed to by ptr on the screen. void
displayStr();

// Remove the first element of the linked structure pointed to by ptr.
// If the structure is empty, do nothing void
removeFirst();

// Remove the first element of the linked structure pointed to by ptr.
// If the structure is empty, do nothing void
removeLast();
// Remove the first element of the linked structure with an info field equal to k.
// If no such element or the list is empty, do nothing void
remove(int k);

};
```

Write the implementation of the class LinkedStr. Write a driver program to test the implementation.

Output be like:

Values in the link list are :10 9 8 7 6 5 4 3 2 1

After removing first and last element:9 8 7 6 5 4 3 2

After removing the 5 element:9 8 7 6 4   3 2 1