*Submitted By: Ravi Pandey (075BCT065)*

INTRODUCTION

List

A list (static list) is a collection of data structures much like a queue but unlike in a queue, items can be added, removed or accessed from any point at any time from a list.

Linked List

A linked list is a sequence of data structures, which are connected together via links.

Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list is the second most-used data structure after array. Following are the important terms to understand the concept of Linked List.

* **Link** − Each link of a linked list can store a data called an element.
* **Next** − Each link of a linked list contains a link to the next link called Next.
* **LinkedList** − A Linked List contains the connection link to the first link called First.

Linked List Representation

Linked list can be visualized as a chain of nodes, where every node points to the next node.

A linked list has the following components:

* Linked List contains a link element called start.
* Each link carries a data field(item/s) and a link field called next.
* Each link is linked with its next link using its next link.
* Last link carries a link as null to mark the end of the list.

LIST:

A header file “list.h” was created to use across multiple files.

*List.h:*

#include<iostream>

template<class T>

class List

{

    T\* elements;

    int maxSize;

    static const int increment=10;

    int size;

    void incSize();

public:

    class EMPTY{};

    class INVALID\_INDEX{};

    List(int n=10)

    {

        maxSize=n;

        elements = new T[maxSize];

        size = 0;

    }

    ~List()

    {

        delete elements;

    }

    void insert(T \_item);

    void insert(int index, T \_item);

    T remove();

    T remove(int index);

    int getsize();

    void display();

};

template<class T>

int List<T>::getsize()

{

    return size;

}

template<class T>

void List<T>::incSize()

    {

        maxSize+=increment;

        T\* temp = new T[maxSize];

        for(int i=0;i<size;i++)

            temp[i] = elements[i];

        delete elements;

        elements = temp;

    }

template<class T>

void List<T>::insert(int index ,T \_item)

    {

        if(index > size || index < 0)

            throw INVALID\_INDEX();

        if(size == maxSize)

            incSize();

        for(int i=size;i>index;i--)

            elements[i] = elements[i-1];

        size++;

        elements[index] = \_item;

    }

template<class T>

void List<T>::insert(T \_item)

    {

        if(size == maxSize)

            incSize();

        elements[size] = \_item;

        size++;

    }

template<class T>

T List<T>::remove(int index)

    {

        if(size == 0)

            throw EMPTY();

        if(index > size-1 || index < 0)

            throw INVALID\_INDEX();

        T temp = elements[index];

        for(int i=index;i<size-1;i++)

            elements[i] = elements[i+1];

        size--;

        return temp;

    }

template<class T>

T List<T>::remove()

    {

        if(size == 0)

            throw EMPTY();

        size--;

        return elements[size+1];

    }

template<class T>

void List<T>::display()

    {

        if(size == 0)

            throw EMPTY();

            //return;

        for(int i=0;i<size;i++)

            std::cout<<" "<<elements[i];

    }

1. WAP TO INSERT AND DELETE IN A LIST.

Program Code:

#include<iostream>

#include "list.h"

int main()

{

    List<int> i;

    int no,index,selection;

    while(true)

    {

        try

        {

            std::cout<<"\n\nEnter operation to perform: 1.INSERT  ";

            std::cout<<"2.INSERT-AT-INDEX  3.REMOVE  4.REMOVE-FROM-INDEX  5.EXIT  ";

            std::cin>>selection;

            if(selection == 1)

            {

                std::cout<<"\nEnter number to insert ";

                std::cin>>no;

                i.insert(no);

            }

            else if(selection == 2)

            {

                std::cout<<"\nEnter number to insert ";

                std::cin>>no;

                std::cout<<"Enter the index ";

                std::cin>>index;

                i.insert(index,no);

            }

            else if(selection == 3)

            {

                std::cout<<"\n";

                i.remove();

            }

            else if(selection == 4)

            {

                std::cout<<"\nEnter the index ";

                std::cin>>index;

                i.remove(index);

            }

            else{ break;}

            std::cout<<"\nLIST: ";

            i.display();

        }

        catch(List<int>::EMPTY)

        {

            std::cout<<"\nList is Empty!";

        }

        catch(List<int>::INVALID\_INDEX)

        {

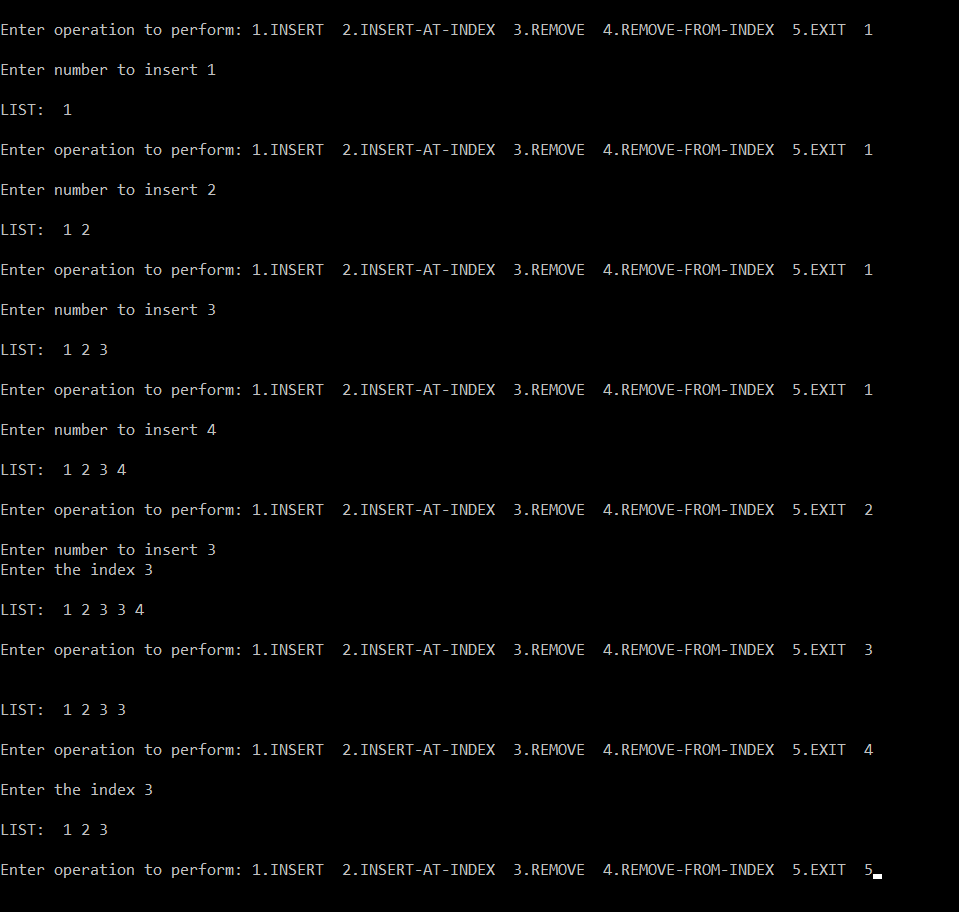
            std::cout<<"\nINVALID INDEX! ";

        }

    }

}

Output



2. WAP TO REPRESENT QUEUE AS A LIST.

Program Code:

#include<iostream>

#include"list.h"

template<class T>

class Queue

{

    List<T> items;

public:

    void enqueue(T \_item)

    {

        items.insert(\_item);

    }

    T dequeue()

    {

        return items.remove(0);

    }

    void display()

    {

        items.display();

    }

};

int main()

{

    Queue<int> iqueue;

    int no,selection;

    while(true)

    {

        try

        {

            std::cout<<"\nEnter operation to perform:";

            std::cout<<" 1.ENQUEUE  2.DEQUEUE 3.EXIT  ";

            std::cin>>selection;

            if(selection == 1)

            {

                std::cout<<"\nEnter a number to enqueue: ";

                std::cin>>no;

                iqueue.enqueue(no);

            }

            else if(selection == 2)

            {

                std::cout<<'\n'<<iqueue.dequeue()<<" dequeued. \n";

            }

            else

            {

                break;

            }

            std::cout<<"\nQUEUE: ";

            iqueue.display();

        }

        catch(List<int>::EMPTY)

        {

            std::cout<<"\nQueue is Empty!";

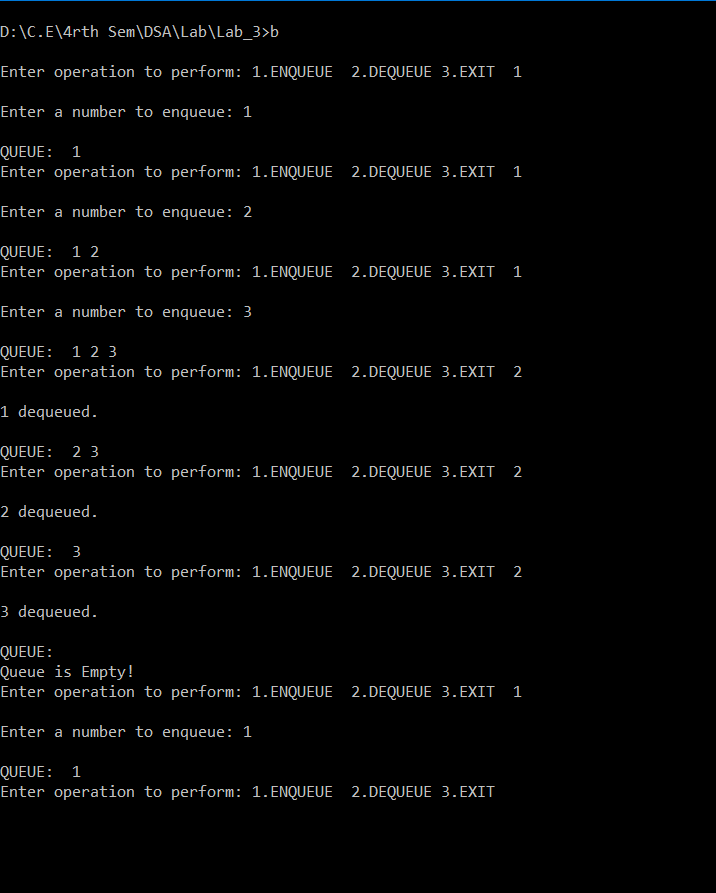
        }

    }

    return 0;

}

Output



LINKED LIST:

Write an algorithm and program for following cases in singly linear linked list:

ALGORITHM

*1)INSERTIONS OPERATIONS*

1. INSERT NODE AT BEGINNING OF THE LIST:
2. Get input from user
3. Create a new node to represent given item
4. Check if the list is empty. If so, then set start pointer of list to point the new node
5. Else set the next pointer of the new node to start and start pointer to the new node.
6. INSERT NODE AT END OF THE LIST:
7. Get input from user
8. Create a new node to represent given item
9. If the list is empty then set start pointer of list to point the new node
10. Else traverse the list to the end node (until next pointer of the node points to null) and set the next pointer of the end node to the new node.
11. INSERT NODE AFTER SPECIFIC NODE:
12. Get item to be inserted and the specific node after which item is to be inserted
13. Create a new node to represent given item
14. Traverse the list to get to the specific node
15. Set next pointer of the new node to next node of the given node and next pointer of the given node to the new node.
16. INSERT NODE BEFORE SPECIFIC NODE:
17. Get item to be inserted and the specific node before which item is to be inserted
18. Create a new node to represent given item
19. Traverse the list to get the node just before given node (until next pointer of the node point to given node)
20. Set next pointer of the new node to given node and next pointer of the node just before the specific node to the new node

*2)DELETION OPERATIONS*

1. DELETE NODE FROM BEGINNING OF THE LIST:
2. Display empty message or terminate the program if the list is empty
3. Else set start to point to the next node of current starting node and free the starting node.
4. DELETE THE NODE FROM END OF THE LIST:
5. Display empty message or terminate the program if the list is empty
6. Else traverse the list to end
7. Then, set the next pointer of the node just before the end node to null and free the end node.
8. DELETE NODE AFTER SPECIFIC NODE:
9. Get the pointer to the specific node from user
10. Set the next pointer of given node to the next pointer of its next node and free the space of the removed node.

Program Code:

#include<iostream>

template<class T>

class Node

{

public:

    T item;

    Node<T>\* next;

};

template<class T>

class List

{

    Node<T>\* start;

public:

    class EMPTY{};

    class INVALID\_INDEX{};

    List()

    {

        start = nullptr;

    }

    ~List()

    {

      if(start != nullptr)

       {

           Node<T>\* p = start;

           Node<T>\* q = p->next;

           if( q == nullptr)

                delete p;

           else

           for(; q != nullptr ; p = q , q = q->next)

              delete p;

       }

    }

 void insertAtBeg(T \_item);

 void insertAtEnd(T \_item);

 void insertAfter(Node<T>\* spNode,T \_item);

 void insertBefore(Node<T>\* spNode,T \_item);

 T deleteAfter(Node<T>\* spNode);

 T deleteFromBeg();

 T deleteFromEnd();

 Node<T>\* getNode(int index);

 void display();

};

template<class T>

void List<T>::insertAtBeg(T \_item)

    {

        Node<T>\* newNode = new Node<T>();

        newNode->item = \_item;

        if (start == nullptr)

        {

            newNode->next = nullptr;

            start = newNode;

        }

        else

        {

            newNode->next = start;

            start = newNode;

        }

    }

template<class T>

void List<T>::insertAtEnd(T \_item)

    {

        Node<T>\* newNode = new Node<T>();

        newNode->item = \_item;

        newNode->next = nullptr;

        if (start == nullptr)

        {

            start = newNode;

        }

        else

        {

            Node<T>\* p = start;

            Node<T>\* q = p->next;

                for(; q != nullptr ; p = q , q = p->next) ;

            p->next = newNode;

        }

    }

template<class T>

void List<T>::insertAfter(Node<T>\* spNode,T \_item)

    {

        Node<T>\* newNode = new Node<T>();

        newNode->item = \_item;

        newNode->next = spNode->next;

        spNode->next = newNode;

    }

template<class T>

void List<T>::insertBefore(Node<T>\* spNode,T \_item)

    {

        Node<T>\* newNode = new Node<T>();

        newNode->item = \_item;

        if(spNode == start)

        {

            insertAtBeg(\_item);

            return;

        }

        Node<T>\* p = start;

        Node<T>\* q = p->next;

            for(; q != spNode ; p = q , q = p->next) ;

        newNode->next = p->next;

        p->next = newNode;

    }

template<class T>

   T List<T>::deleteFromBeg()

    {

        if(start == nullptr)

            throw EMPTY();

        Node<T>\* temp = start;

        start = start->next;

        delete temp;

    }

template<class T>

T List<T>::deleteFromEnd()

    {

        if(start == nullptr)

            throw EMPTY();

        T temp;

        Node<T>\* p = start;

        Node<T>\* q = nullptr;

            for(;p->next != nullptr; q = p, p = p->next) ;

        if(q == nullptr)

        {

            start = nullptr;

        }

        else

        {

            q->next = nullptr;

        }

        temp = p->item;

        delete p;

        return  temp;

    }

template<class T>

T List<T>::deleteAfter(Node<T>\* spNode)

    {

        T temp;

        Node<T>\* tempNode;

        tempNode = spNode->next;

        temp = tempNode->item;

        spNode->next = tempNode->next;

        delete tempNode;

        return  temp;

    }

template<class T>

Node<T>\* List<T>::getNode(int index)

{

    if(start == nullptr)

        throw EMPTY();

    Node<T>\* p = start;

    for(int i=0;i<index;i++)

    {

        if(p == nullptr)

            throw INVALID\_INDEX();

        p = p->next;

    }

    return p;

}

template<class T>

void List<T>::display()

{

    if(start == nullptr)

        throw EMPTY();

    Node<T>\* p = start;

    for(; p != nullptr ; p = p->next)

        std::cout<<" "<<p->item;

}

int main()

{

    List<int> ilist;

    int no ,index,selection;

    Node<int>\* ptr;

    while(true)

    {

        try

        {

            std::cout<<"\nEnter operation to perform: 1.INSERT  2.DELETE 3.EXIT  ";

            std::cin>>selection;

            if(selection == 1)

            {

                std::cout<<"\nWhere to insert: 1.BEGINING  2.BEFORE-INDEX  3.AFTER-INDEX  4.END ";

                std::cin>>selection;

                std::cout<<"\nEnter a number to insert: ";

                std::cin>>no;

                if(selection == 1)

                    ilist.insertAtBeg(no);

                else if(selection == 4)

                    ilist.insertAtEnd(no);

                else if(selection == 2)

                {

                    std::cout<<"Enter index ";

                    std::cin>>index;

                    ptr = ilist.getNode(index);

                    ilist.insertBefore(ptr,no);

                }

                else if(selection == 3)

                {

                    std::cout<<"Enter index ";

                    std::cin>>index;

                    ptr = ilist.getNode(index);

                    ilist.insertAfter(ptr,no);

                }

                else

                {

                    break;

                }

            }

            else if(selection == 2)

            {

                std::cout<<"\nWhere to delete from: 1.BEGINING  2.AFTER-INDEX  3.END ";

                std::cin>>selection;

                if(selection == 1)

                {

                    std::cout<<"\n";

                    ilist.deleteFromBeg();

                }

                else if(selection == 3)

                {

                    std::cout<<"\n";

                    ilist.deleteFromEnd();

                }

                else if(selection == 2)

                {

                    std::cout<<"\nEnter index ";

                    std::cin>>index;

                    ptr = ilist.getNode(index);

                    ilist.deleteAfter(ptr);

                }

                else

                {

                    break;

                }

            }

            else

            {

                break;

            }

           std::cout<<"\nLIST: ";

           ilist.display();

        }

        catch(List<int>::EMPTY)

        {

            std::cout<<"List is Empty! ";

        }

        catch(List<int>::INVALID\_INDEX)

        {

            std::cout<<" INVALID INDEX!! ";

        }

    }

return 0;

}

Output

