DATA STRUCTURE AND ALGORITHM

Lab 5

Doubly Linked Linear List

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.

Doubly Linked List

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

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

* **Element** − Each link can store a data called an element or item.
* **Next** − Each link contains a link to the next link called Next.
* **Previous –** Each link contains a link to the previous link called Previous.
* **Start** – A Doubly Linked List contains the connection link to the first link called Start. It’s previous points to null.

WRITE AN ALGORITHM AND PROGRAM FOR FOLLOWING CASES IN DOUBLY 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, set previous pointer of new node to null pointer.
6. Then, set start pointer to the new node.
7. INSERT NODE AT END OF THE LIST:
8. Get input from user
9. Create a new node to represent given item
10. If the list is empty then set start pointer of list to point the new node
11. 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 and the previous of new node to that end node.
12. INSERT NODE AFTER SPECIFIC NODE:
13. Get item to be inserted and the specific node after which item is to be inserted
14. Create a new node to represent given item
15. Traverse the list to get to the specific node
16. Set next pointer of the new node to next node of the specific node and set its previous pointer to the specific node
17. Set the next pointer of the specific node to the new node.
18. INSERT NODE BEFORE SPECIFIC NODE:
19. G et item to be inserted and the specific node before which item is to be inserted
20. Create a new node to represent given item
21. Traverse the list to get the specific node
22. Set next pointer of the new node to specific node and previous of new node to previous pointer of the node pointed by previous of specific node
23. If specific node is start, then set start=new node else set next of the node before specific node to new node
24. Set previous of specific node equal 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
4. Check if the list becomes empty or not, if not so, set previous of the new starting node to null pointer and free the earlier starting node.
5. DELETE THE NODE FROM END OF THE LIST:
6. Display empty message or terminate the program if the list is empty
7. If the list has only one node, free the node and set start to null pointer
8. Else traverse the list to end
9. Then, set the next pointer of the node just before the end node to null and free the end node.
10. DELETE NODE BEFORE SPECIFIC NODE:
11. Get the pointer to the specific node from user
12. Display error message if list is empty or invalid index is given by user
13. Create a temporary node resembling the node before given node
14. Set next of node before temporary node to point to specific node and previous of specific node= previous of temporary node
15. Free the space of removed node.
16. DELETE NODE AFTER SPECIFIC NODE:
17. Get the pointer to the specific node from user
18. Display error message if list is empty or invalid index is given by user
19. Create a temporary node resembling the node after given node
20. Set the previous of node after temporary node equal to given node
21. Then, set the next pointer of the given node to node after temporary node
22. Free the space of the removed node.

Program Code:

#include <iostream>

template <class t>

struct NODE

{

    t item;

    NODE<t>\* next;

    NODE<t>\* prev;

};

template <class t>

class DoublyLinkedList

{

    typedef NODE<t> Node;

    Node \* start;

    public:

    class UNDERFLOW{};

    class INVALID\_INDEX{};

    DoublyLinkedList()

    {

        start=nullptr;

    }

    ~DoublyLinkedList()

    {

        if(start==nullptr)

            delete start;

        else

        {

            Node \* p=start;

            Node \* q =nullptr;

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

            delete p;

        }

    }

    void display();

    Node\* getNodeAtIndex(int);

    void insertAtFront(t);

    void insertAtEnd(t);

    void insertAfter(Node\*,t);

    void insertBefore(Node\*,t);

    t deleteFromFront();

    t deleteFromEnd();

    t deleteBefore(Node \*);

    t deleteAfter(Node \*);

};

template <class t>

void DoublyLinkedList<t>::display()

{

    if(start==nullptr)

        throw UNDERFLOW();

    std::cout<<"Data in List:\t";

    for(Node\* n=start;n!=nullptr;n=n->next)

    {

        std::cout<<'\t'<<n->item;

    }

    std::cout<<"\n";

}

template <class t>

NODE<t> \* DoublyLinkedList<t>::getNodeAtIndex(int index)

{

    if(start==nullptr)

        throw UNDERFLOW();

    if(index<0)

        throw UNDERFLOW();

    Node\* node = start;

    for(int i=0;i<index;i++)

    {

        node=node->next;

        if(node==nullptr)

            throw INVALID\_INDEX();

    }

    return node;

}

template <class t>

void DoublyLinkedList<t>::insertAtFront(t data)

{

    Node \* newNode=new Node();

    newNode->item=data;

    newNode->prev=nullptr;

    if(start==nullptr)

    {

        newNode->next=nullptr;

        start=newNode;

    }

    else

    {

        start->prev = newNode;

        newNode->next = start;

        start = newNode;

    }

}

template <class t>

void DoublyLinkedList<t>::insertAtEnd(t data)

{

    Node \* newNode=new Node();

    newNode->item=data;

    newNode->next=nullptr;

    if(start==nullptr)

    {

        newNode->prev=nullptr;

        start=newNode;

    }

    else

    {

        Node \* p= start;

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

        p->next = newNode;

        newNode->prev=p;

    }

}

template <class t>

void DoublyLinkedList<t>::insertAfter(Node\* spNode,t data)

{

    Node\* newNode = new Node();

    newNode->item=data;

    newNode->next=spNode->next;

    newNode->prev=spNode;

    spNode->next=newNode;

}

template <class t>

void DoublyLinkedList<t>::insertBefore(Node\* spNode,t data)

{

    Node \* newNode= new Node();

    newNode->item = data;

    newNode->prev = spNode->prev;

    newNode->next = spNode;

    if(spNode!=start)

        (spNode->prev)->next = newNode;

    else

                start = newNode;

    spNode->prev= newNode;

}

template <class t>

t DoublyLinkedList<t>::deleteFromFront()

{

    if(start==nullptr)

        throw UNDERFLOW();

    Node\* tempNode = start;

    t temp = tempNode->item;

    start=start->next;

    if(start!=nullptr)

        start->prev=nullptr;

    delete tempNode;

    return temp;

}

template <class t>

t DoublyLinkedList<t>::deleteFromEnd()

{

    bool loopEntered=false;     //is set to true if list has mode than 1 nodes

    if(start==nullptr)

        throw UNDERFLOW();

    Node \* tempNode = start;

    for(;tempNode->next!=nullptr;tempNode=tempNode->next)

    {

        loopEntered=true;

    }

    t temp = tempNode->item;

    if(!loopEntered)

    {

        start=nullptr;

    }

    else

    {

        (tempNode->prev)->next=nullptr; //disconnectiong last node

    }

    delete tempNode;

    return temp;

}

template <class t>

t DoublyLinkedList<t>::deleteBefore(NODE<t>\* spNode)

{

    if(spNode == start)

        throw INVALID\_INDEX();

    t temp;

    Node \* tempNode;

    tempNode = spNode->prev;

    temp = tempNode->item;

    if(spNode->prev==start)

    {

        start = spNode;

        start->prev=nullptr;

    }

    else

    {

        (tempNode->prev)->next=spNode;

         spNode->prev=tempNode->prev;

    }

    delete tempNode;

    return temp;

}

template <class t>

t DoublyLinkedList<t>::deleteAfter(Node \* spNode)

{

    if(spNode->next == nullptr)

        throw INVALID\_INDEX();

    t temp;

    Node \* tempNode;

    tempNode = spNode->next;

    temp = tempNode->item;

    if(tempNode->next!=nullptr)

        (tempNode->next)->prev = spNode;

    spNode->next=tempNode->next;

    delete tempNode;

    return temp;

}

class INVALID\_SELECTION{};

int main()

{

    int sel;

    DoublyLinkedList<int> dl;

    int num,index;

    while(1)

    {

        try

        {

            std::cout<<"\nEnter Operation:\t1)Insert\t2)Delete\t3)Exit\t";

            std::cin>>sel;

            if(sel==1)

            {

                std::cout<<"Insert at:\t\t1)Beginning\t2)End\t3)Before index\t4)After index\t";

                std::cin>>sel;

                if(sel<=0 || sel>4)

                    throw INVALID\_SELECTION();

                std::cout<<"Enter the number:\t";// to be inserted:\t";

                std::cin>>num;

                if(sel==1)

                    dl.insertAtFront(num);

                else if(sel==2)

                    dl.insertAtEnd(num);

                else if(sel==3 || sel==4)

                {

                    std::cout<<"Enter index:\t\t";

                    std::cin>>index;

                    NODE<int>\* node = dl.getNodeAtIndex(index);

                    if(sel==3)

                        dl.insertBefore(node,num);

                    else

                        dl.insertAfter(node,num);

                }

            }

            else if(sel==2)

            {

                int data;

                std::cout<<"Delete from:\t\t1)Beginning\t2)End\t3)Before index\t4)After index\t";

                std::cin>>sel;

                if(sel==1)

                    data = dl.deleteFromFront();

                else if(sel==2)

                    data = dl.deleteFromEnd();

                else if(sel==3 || sel==4)

                {

                    std::cout<<"Enter index:\t\t";

                    std::cin>>index;

                    NODE<int>\* node = dl.getNodeAtIndex(index);

                    if(sel==3)

                        data = dl.deleteBefore(node);

                    else

                        data = dl.deleteAfter(node);

                }

                else

                {

                    throw INVALID\_SELECTION();

                }

                std::cout<<data<<" deleted.\n";

            }

            else

            {

                exit(0);

            }

            dl.display();

        }

        catch(DoublyLinkedList<int>::UNDERFLOW)

        {

            std::cerr<<"List is empty!\n";

        }

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

        {

            std::cerr<<"Inavlid index selected!\n";

        }

        catch(INVALID\_SELECTION)

        {

            std::cerr<<"Invalid Option Selected!\n";

        }

    }

    return 0;

}

Output:



