Batch:B2 Roll No.:16010121110

Experiment No. 6

Grade: AA / AB / BB / BC / CC / CD /DD

Title: Implementation of Linked List

Objective: To understand the use of linked list as data structures for various application.

Expected Outcome of Experiment:

CO	Outcome
CO 2	Apply linear and non-linear data structure in application development.

Books/ Journals/ Websites referred:

Citations mentioned in intext format.

Introduction:

Define Linked List

A linked list consists of nodes where each node contains a data field and a reference(link) to the next node in the list.

Source

https://www.geeksforgeeks.org/data-structures/linked-list/

Types of linked list:

There are four key types of linked lists:

1) Singly linked lists.
2) Doubly linked lists.
3) Circular linked lists.
4) Circular doubly linked lists.

Source - https://www.simplilearn.com/tutorials/data-structure-tutorial/types-of-linked-list

Algorithm for creation, insertion, deletion, traversal and searching an element in assigned linked list type:

2. LLType Insert(LLType Head, NodeType NewNode)

// This Algorithm adds a NewNode at the desired position in the linked list. Head is the pointer that points to the first node in the linked list

{ if (head==NULL) // first element in Queue

NewNode->Next = NULL;

head=NewNode;

Else // General case: insertion before head, in the end, in between

}
Algorithm LLType CreateLinkedList()

//This Algorithm creates and returns an empty Linked List, pointed by a pointer -head

{ createNode(head);

head=NULL;

}

3. Algorithm ElementType Delete(LLType Head, ElementType ele)

//This algorithm returns ElementType ele if it exists in the List, an error message otherwise. Temp and current are the a temproary nodes used in the delete process.

{ if (Head==NULL)

Print “Underflow”

exit;

Else

{0. search for element

1. element doesn't exist in list

2. deletion in unsorted list

3. deletion in sorted list

}

}
Algorithm NodeType Search(LLType Head,
ElementType Key)

//This algorithm returns NodeType node which contains the
'keyvalue" being searched.

{ if (Head==NULL)

Print "element doesn't exist"

exit;

Else{
return search(this - > next;
}

Implementation of an application using linked list:

```
/*****************************************************************
*********
Add two polynomials


******************************************************************
*********/
#include <iostream>


using namespace std;


class node{
   public:
   node* pointer;
   int value;
   int priority;
   node(){


   }
   public:
   node(node *pointer, int value,int priority){
      this->pointer=pointer;
      this->value=value;
      this->priority=priority;
   }
   void point(node *pointer){
      this->pointer=pointer;
```

```
    }
};
class linkedlist{
        public:

    node *top;
    node * bottom;
    linkedlist(){
       top= new node(NULL,0,0);
       bottom= new node(NULL,0,0);

    }
    void add(node *toBeAdded){
       if(top->pointer!=NULL)
       {top->pointer->point(toBeAdded);}
       if(bottom->pointer==NULL){
          bottom->point(toBeAdded);
       }
        top->point(toBeAdded);
    }
    void add(int value,int priority){
        add(new node(NULL,value,priority));
    }
     void display(){
         node temp;
```

```
        temp.point(bottom->pointer);
do{
    cout<<"\n"<<temp.pointer->value<<"*"<<"x^("<<temp.pointer->priority<<")";


         temp.point(temp.pointer->pointer);
}
      while(temp.pointer!=NULL);


    }


};


linkedlist* addPoly(linkedlist a,linkedlist b){
 linkedlist *c = new linkedlist();
 node temp1;
 temp1.point(a.bottom->pointer);
 node temp2;
 temp2.point(b.bottom->pointer);




 for (int i=0;0==0;i++){
 //Only for integral powers of x
 //assuming sorted polynomials
 int temp3=0;
```

```
 if(temp1.pointer==NULL && temp2.pointer==NULL)
 {
    break;
 }
 if(temp1.pointer!=NULL && temp1.pointer->priority==i){
 temp3=temp3+temp1.pointer->value;
 temp1.point(temp1.pointer->pointer);
 }

 if(temp2.pointer!=NULL && temp2.pointer->priority==i){
 temp3=temp3+temp2.pointer->value;
 temp2.point(temp2.pointer->pointer);
 //cout<<temp3;
 }
 c->add(temp3,i);

 }
//a.display();
 return c;
}

linkedlist* multiply(linkedlist a,  int constant){

  node* temp=a.bottom;
 linkedlist *c = new linkedlist();
```

```
    while(temp->pointer!=NULL){
        c->add(constant*(temp->pointer->value),temp->pointer->priority);
        temp->point(temp->pointer->pointer);
    }
    return c;


}
linkedlist* minusPoly(linkedlist a){
    node* temp=a.bottom;
 linkedlist *c = new linkedlist();
    while(temp->pointer!=NULL){
        c->add(-temp->pointer->value,temp->pointer->priority);
        temp->point(temp->pointer->pointer);
    }
    return c;
}

linkedlist* subPoly(linkedlist a,linkedlist b){

    return addPoly(a,*minusPoly(b));
}
int main()
{

linkedlist *mylink=new linkedlist();
```

```
mylink->add(2,1);
mylink->add(20,2);
mylink->add(14,4);
//mylink->display();

linkedlist *mylink2=new linkedlist();
mylink2->add(7,0);
mylink2->add(2,1);
mylink2->add(31,3);
mylink2->add(5,4);

cout<<"Hello";

//mylink2->display();
linkedlist *c = addPoly(*mylink,*mylink2);

c->display();



linkedlist *d = subPoly(*mylink,*mylink2);
d->display();

linkedlist *e = multiply(*d,3);
e->display();
```

```
    return 0;
}
```

Conclusion:-

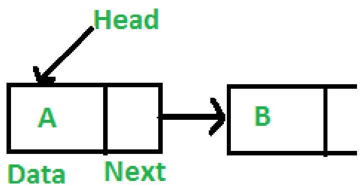
Thus we have understood the working of linkedl;ist and implemented application using linkedlist. A linked list is made up of nodes. We call every flower on this particular garland to be a node. And each of the node points to the next node in this list as well as it has data

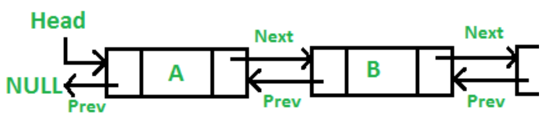
Post lab questions:

1. Compare and contrast SLL and DLL

Reference - https://www.geeksforgeeks.org/difference-between-singly-linked-list-and-doubly-linked-list/#:~:text=In%20SLL%2C%20the%20traversal%20can,directions%20(forward%20and%20backward).

Singly linked list (SLL)	Doubly linked list (DLL)
SLL nodes contains 2 field -data field and next link field.	DLL nodes contains 3 fields -data field, a previous link field and a next link field.





In SLL, the traversal can be done using the next node link only. Thus traversal is possible in one direction only.	In DLL, the traversal can be done using the previous node link or the next node link. Thus traversal is possible in both directions (forward and backward).
The SLL occupies less memory than DLL as it has only 2 fields.	The DLL occupies more memory than SLL as it has 3 fields.
Complexity of insertion and deletion at a given position is O(n).	Complexity of insertion and deletion at a given position is O(n / 2) = O(n) because traversal can be made from start or from the end.
Complexity of deletion with a given node is O(n), because the previous node needs to be known, and traversal takes O(n)	Complexity of deletion with a given node is O(1) because the previous node can be accessed easily
We mostly prefer to use singly linked list for the execution of stacks.	We can use a doubly linked list to execute heaps and stacks, binary trees.

When we do not need to perform any searching operation and we want to save memory, we prefer a singly linked list.	In case of better implementation, while searching, we prefer to use doubly linked list.
A singly linked list consumes less memory as compared to the doubly linked list.	The doubly linked list consumes more memory as compared to the singly linked list.