NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES

CL-218–DATA STRUCTURES LAB

Lab Session 05

Instructors: Ms. Aqsa Zahid

Outline:

- Single link 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.
- Linked List − A Linked List contains the connection link to the first link called First.

Why Linked List?

Arrays can be used to store linear data of similar types, but arrays have the following limitations.

1) The size of the arrays is fixed: So we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage.
2) Inserting a new element in an array of elements is expensive because the room has to be created for the new elements and to create room existing elements have to be shifted.

- For example, in a system, if we maintain a sorted list of IDs in an array id[].
- id[] = [1000, 1010, 1050, 2000, 2040].

And if we want to insert a new ID 1005, then to maintain the sorted order, we have to move all the elements after 1000 (excluding 1000).
Deletion is also expensive with arrays until unless some special techniques are used. For example, to delete 1010 in id[], everything after 1010 has to be moved.

Advantages over arrays

1) Dynamic size
2) Ease of insertion/deletion

Drawbacks:

1) Random access is not allowed. We have to access elements sequentially starting from the first node. So we cannot do binary search with linked lists efficiently with its default implementation. Read about it here.
2) Extra memory space for a pointer is required with each element of the list.
3) Not cache friendly. Since array elements are contiguous locations, there is locality of reference which is not there in case of linked lists.

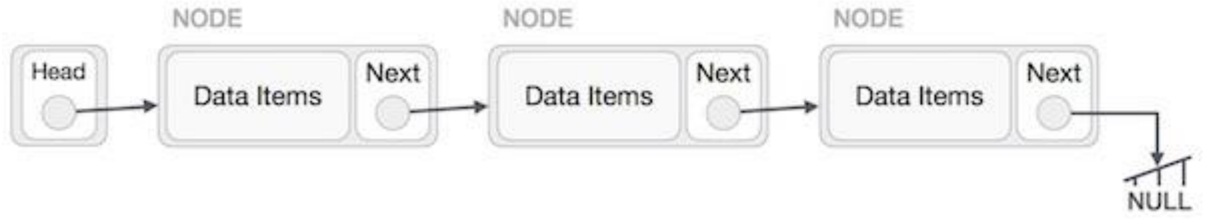
Representation:

A linked list is represented by a pointer to the first node of the linked list. The first node is called

the head. If the linked list is empty, then the value of the head is NULL.
Each node in a list consists of at least two parts:
1) data
2) Pointer (Or Reference) to the next node
In C, we can represent a node using structures. Below is an example of a linked list node with integer data.
In Java or C#, LinkedList can be represented as a class and a Node as a separate class. The LinkedList class contains a reference of Node class type.

Linked List Representation:

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



As per the above illustration, following are the important points to be considered.

- Linked List contains a link element called first.
- Each link carries a data field(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.

Types of Linked List:

Following are the various types of linked list.

- Simple Linked List − Item navigation is forward only.
- Doubly Linked List − Items can be navigated forward and backward.
- Circular Linked List − Last item contains link of the first element as next and the first element has a link to the last element as previous.

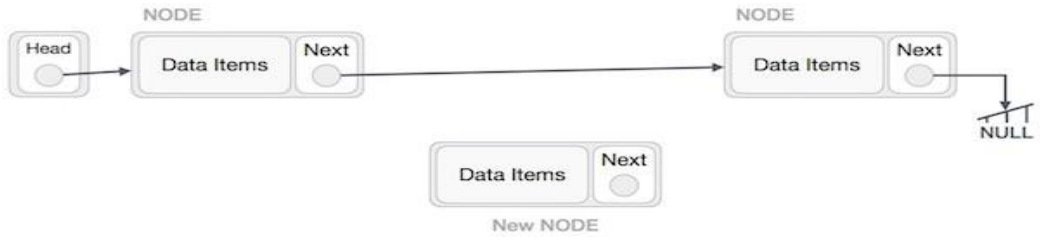
Basic Operations:

Following are the basic operations supported by a list.

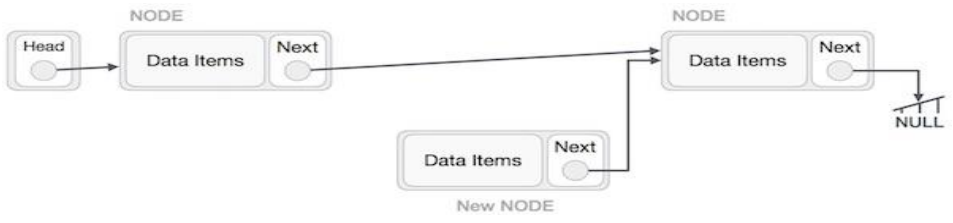
- Insertion − Adds an element at the beginning of the list.
- Deletion − Deletes an element at the beginning of the list.
- Display − Displays the complete list.
- Search − Searches an element using the given key.
- Delete − Deletes an element using the given key.

1. Insertion Operation:

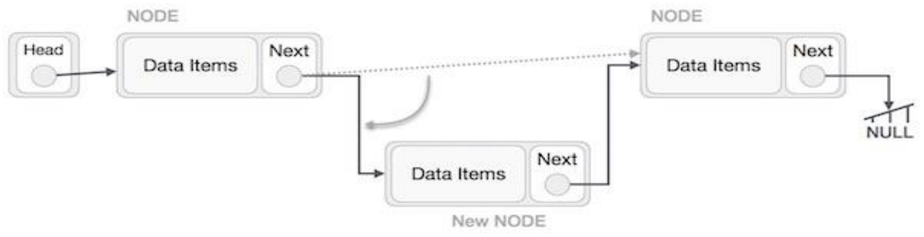
Adding a new node in linked list is a more than one step activity. We shall learn this with diagrams here. First, create a node using the same structure and find the location where it has to be inserted.



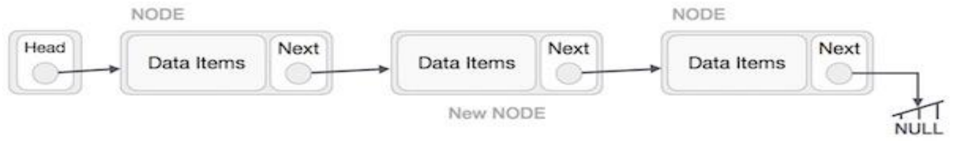
NewNode.next −> RightNode;



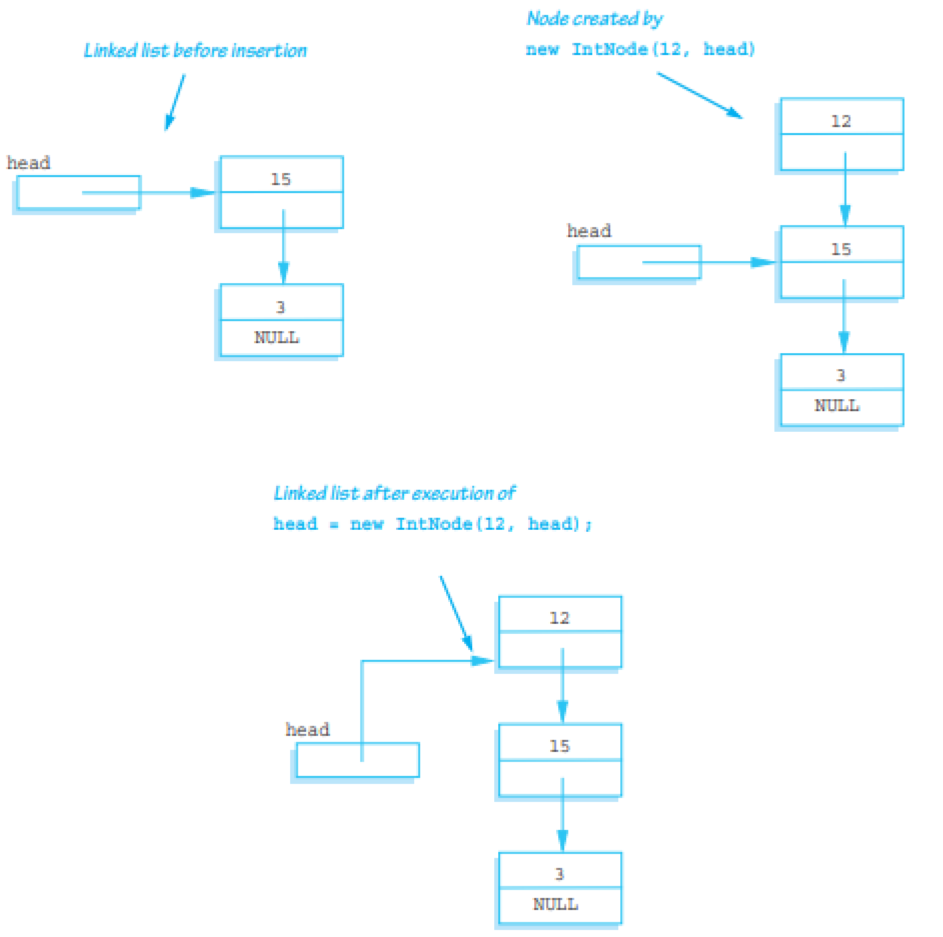
Now, the next node at the left should point to the new node.
LeftNode.next −> NewNode;



This will put the new node in the middle of the two. The new list should look like this −

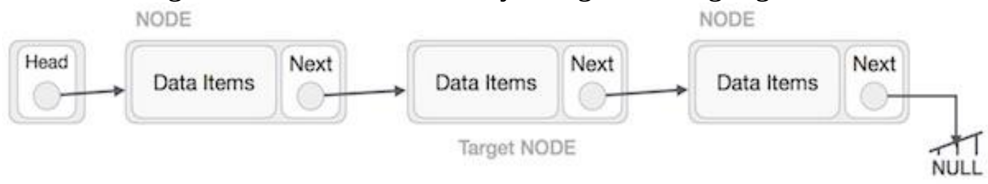


Display 17.3 Adding a Node to the Head of a Linked List

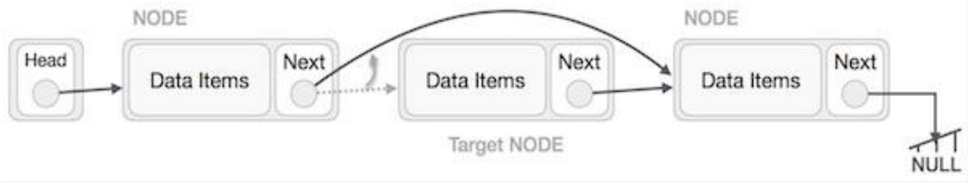

Linked list before insertion
head
15
3
NULL
Node created by
new IntNode(12, head)
12
head
15
3
NULL
Linked list after execution of
head = new IntNode(12, head);
12
head
15
3
NULL

2. Deletion Operation:

Deletion is also a more than one step process. We shall learn with pictorial representation. First, locate the target node to be removed, by using searching algorithms.

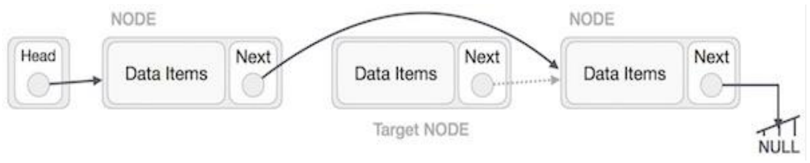


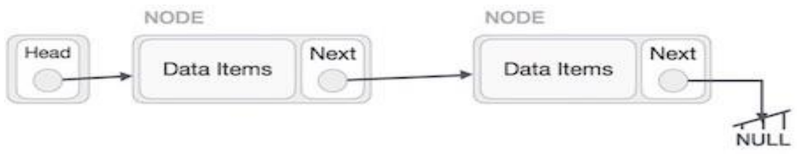
The left (previous) node of the target node now should point to the next node of the target node − LeftNode.next −> TargetNode.next;



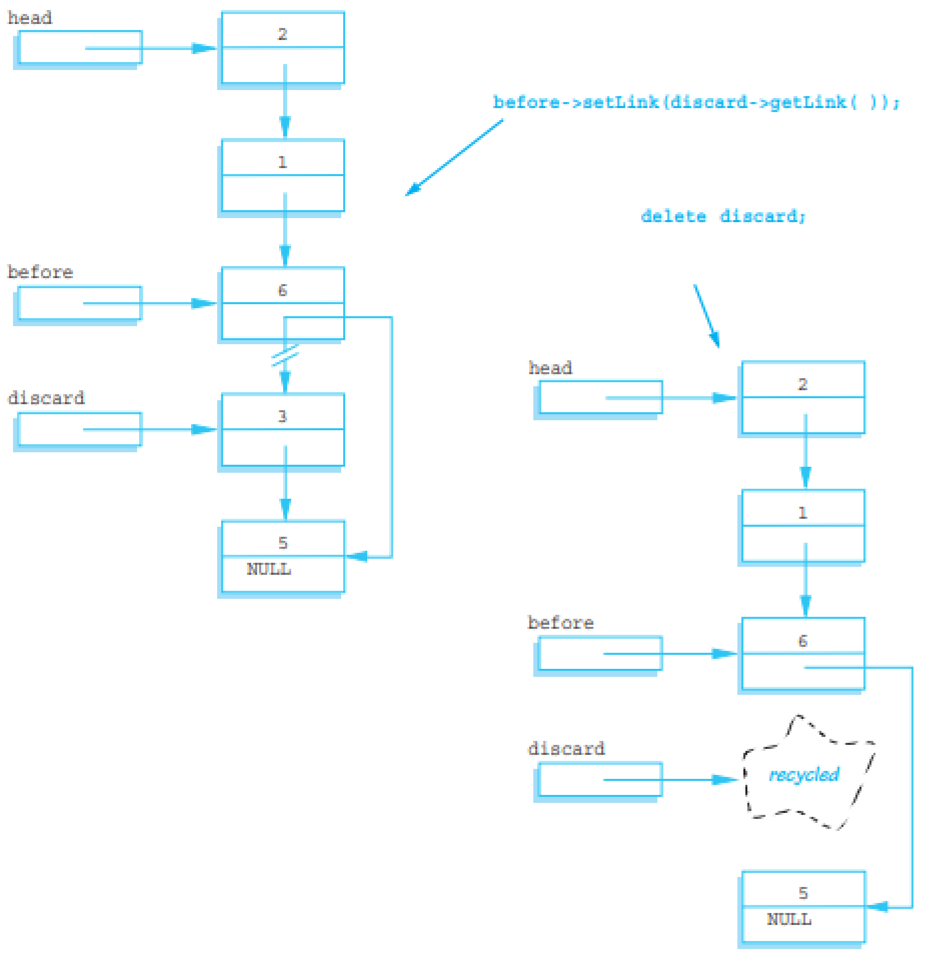
This will remove the link that was pointing to the target node. Now, using the following code, we will remove what the target node is pointing at.

```
TargetNode.next –> NULL;
```





Display 17.7 Removing a Node



```
before->setLink(discard->getLink( ));
```

```
delete discard;
```

This is sufficient to remove the node from the linked list. However, if you are not using this node for something else, you should destroy the node and return the memory it uses for recycling; you can do this with a call to `delete` as follows:

```
    delete discard;
```

The `delete` Operator

The `delete` operator eliminates a dynamic variable and returns the memory that the dynamic variable occupied to the freestore. The memory can then be reused to create new dynamic variables. For example, the following eliminates the dynamic variable pointed to by the pointer variable `p`:

```
delete p;
```

After a call to `delete`, the value of the pointer variable, like `p` just shown, is undefined.

algorithm

Pseudocode for `search` Function

```
Make the pointer variable here point to the head node (that is, first node) of the linked list.

while (here is not pointing to a node containing target
                          and here is not pointing to the last node)
{
      Make here point to the next node in the list.
}
if  (the node pointed to by here contains target)
      return here;
else
      return NULL;
```

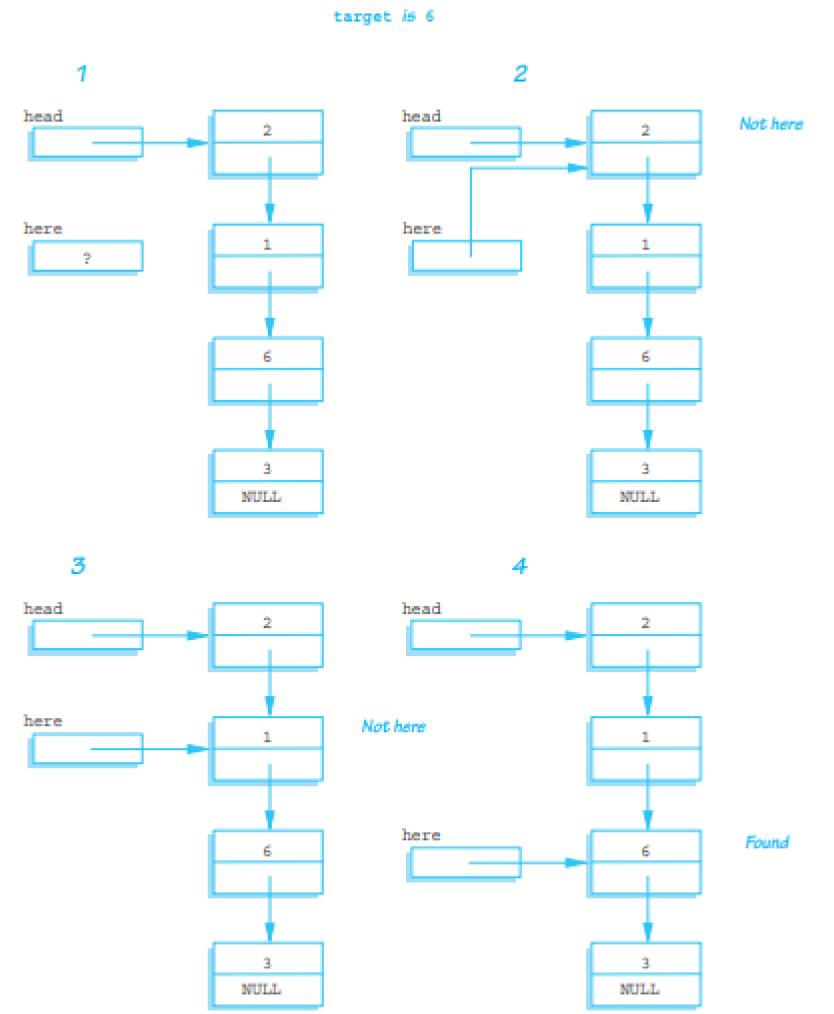
To move the pointer `here` to the next node, we must think in terms of the named pointers we have available. The next node is the one pointed to by the pointer member of the node currently pointed to by `here`. The pointer member of the node currently pointed to by `here` is given by the expression

```
here->getLink( )
```

To move `here` to the next node, we want to change `here` so that it points to the node that is pointed to by the above-named pointer. Hence, the following will move the pointer `here` to the next node in the list:

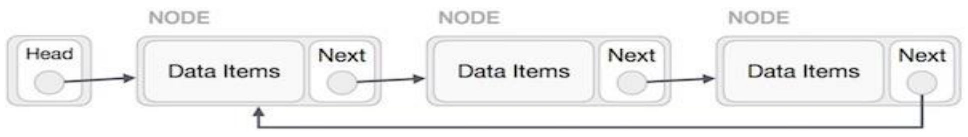
```
here = here->getLink( );
```

Display 17.8 Searching a Linked List



Singly Linked List as Circular:

In singly linked list, the next pointer of the last node points to the first node.



```
#include <iostream>
using namespace std;
//Declare Node
struct Node{
    int num;
    Node *next;
};
//Declare starting (Head) node
struct Node *head=NULL;
//Insert node at start
void insertNode(int n){
    struct Node *newNode=new Node;
    newNode->num=n;
    newNode->next=head;
    head=newNode;}
//Traverse/ display all nodes (print items)
void display(){
    if(head==NULL){
        cout<<"List is empty!"<<endl;
        return;}
    struct Node *temp=head;
    while(temp!=NULL){
        cout<<temp->num<<" ";
        temp=temp->next;}
    cout<<endl;
}
//delete node from start
void deleteItem(){
    if(head==NULL){
        cout<<"List is empty!"<<endl;
        return;
    }
    cout<<head->num<<" is removed."<<endl;
    head=head->next;
}
int main(){
    display();
    insertNode(10);
    insertNode(20);
    insertNode(30);
    insertNode(40);
    insertNode(50);
    display();
    deleteItem(); deleteItem(); deleteItem(); deleteItem(); deleteItem();
    deleteItem();
    display();
    return 0;
}
```

EXMAPLE CODE:

```
#include <iostream>

using namespace std;

struct node
{
   int data;
   node *next;
};

class linked_list
{
private:
   node *head,*tail;
public:
   linked_list()
```

```
    {
        head = NULL;
        tail = NULL;
    }

    void add_node(int n)
    {
        node *tmp = new node;
        tmp->data = n;
        tmp->next = NULL;

        if(head == NULL)
        {
            head = tmp;
            tail = tmp;
        }
        else
        {
            tail->next = tmp;
            tail = tail->next;
        }
    }
};

int main()
{
    linked_list a;
    a.add_node(1);
    a.add_node(2);
    return 0;
}
```

We will first check if the 'head' is NULL or not. If the 'head' is NULL, it means that there is no linked list yet and our current node(tmp) will be the 'head'.

```
if(head == NULL)
 {
   head = tmp;
   tail = tmp;
 }
```

If 'head' is NULL, our current node (tmp) is the first node of the linked list and this it will be 'head' and 'tail' both (as it is also the last element right now).

If ‘head’ is not NULL, it means that we have a linked list and we just have to add the node at the end of the linked list.

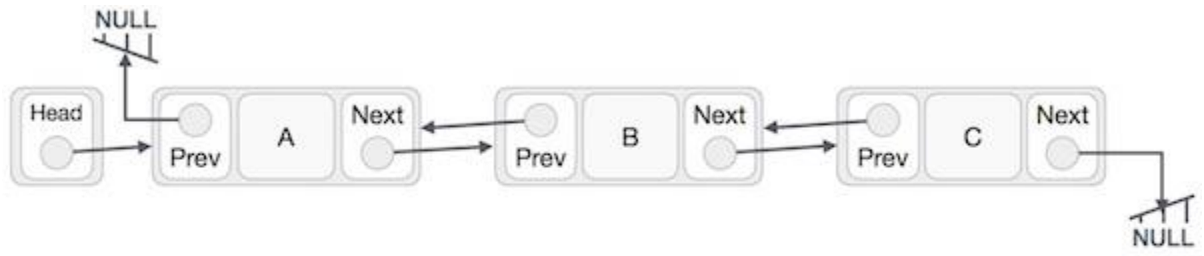
```
else
 {
   tail->next = tmp;
   tail = tail->next;
 }
```

The new node (tmp) will go after the ‘tail’ and then we are changing the tail because the new node is the new ‘tail’.

Doubly Linked List:
Doubly Linked List is a variation of Linked list in which navigation is possible in both ways, either forward and backward easily as compared to Single Linked List. Following is the important term to understand the concept of doubly linked list.

- Prev − Each link of a linked list contains a link to the previous link called Prev.

Doubly Linked List Representation



```
struct node
{
int value;
struct node* next;
struct node* prev;
};
struct node* head;
struct node* tail;
void init()
{
   head=NULL;
   tail=NULL;
}
```

Insertion at First:

```
void insertFirst(int element)
{
   struct node* newItem;
   newItem=new node;
```

```
    if(head==NULL)
    {
        head=newItem;
        newItem->prev=NULL;
        newItem->value=element;
        newItem->next=NULL;
        tail=newItem;
    }
    else
    {
        newItem->next=head;
        newItem->value=element;
        newItem->prev=NULL;
        head->prev=newItem;
        head=newItem;
    }
}
```

Deletion at First:

```
void deleteFirst()
{
    if(head==NULL)
    {
        return;
    }
    if(head==tail)///one element in the list
    {
        struct node* cur;
        cur=head;
        head=NULL;
        tail=NULL;
        delete cur;
        return;
    }
    else
    {
        struct node* cur;
        cur=head;
        head=head->next;
        head->prev=NULL;
        delete cur;
    }
}
```

LAB TASK

Question No. 1:

Write a Count () function that counts the number of times a given int occurs in a list. The code for this has the classic list traversal structure as demonstrated in Length ().

```
void CountTest() {
List myList = BuildOneTwoThree(); // build {1, 2, 3}
int count = Count(myList, 2); // returns 1 since there's 1 '2' in the list
}
/*
 Given a list and an int, return the number of times that int occurs
 in the list.
*/
int Count(struct node* head, int searchFor) {
// Your code
```

Question No. 2:

Menu driven program for all operations on singly linked list

Operations to be performed:

- **traverse():** To see the contents of the linked list, it is necessary to traverse the given linked list. The given traverse() function traverses and prints the content of the linked list.
- **insertAtFront():** This function simply inserts an element at the front/beginning of the linked list.
- **insertAtEnd():** This function inserts an element at the end of the linked list.
- **insertAtPosition():** This function inserts an element at a specified position in the linked list.
- **deleteFirst():** This function simply deletes an element from the front/beginning of the linked list.
- **deleteEnd():** This function simply deletes an element from the end of the linked list.
- **deletePosition():** This function deletes an element from a specified position in the linked list.
- **maximum():** This function finds the maximum element in a linked list.
- **mean():** This function finds the mean of the elements in a linked list.
- **sort():** This function sort the given linked list in ascending order.
- **reverseLL():** This function reverses the given linked list.

Question No. 3:

Write a program that prompts the users to enter 12 numbers. This program reads the numbers into a linked list. Make another Linked list that will store the average of numbers.

The two lists will be passed to a function for and the average will be calculated by

a. Take First Four nodes of "numbers list" calculate their average and store at first node in "Average linked list".

b. Next time skip the first node of "numbers list" and average the next 4 nodes and store at second node in Average linked list.

c. And this procedure will continue, until Average for all will be stored in Averagelist.

Question No. 4:

Given a singly Linked List, the task is to swap the first odd valued node from the beginning and the first even valued node from the end of the Linked List. If the list contains node values of a single parity, then no modifications are required.

Examples:
Input: 4 -> 3 -> 5 -> 2 -> 3 -> NULL
Output: 4 -> 2 -> 5 -> 3 -> 3 -> NULL
Explanation:
4 -> **3** -> 5 -> **2** -> 3 -> NULL ===> 4 -> **2** -> 5 -> **3** -> 3 -> NULL
The first odd value in any node from the beginning is **3**.
The first even value in any node from the end is **2**.
After swapping the above two node values, the linked list modifies to 4 -> 2 -> 5 -> 3 -> 3 -> NULL.
Input: LL: 2 -> 6 -> 8 -> 2 -> NULL
Output: 2 -> 6 -> 8 -> 2 -> NULL

Question No. 5:

Given a singly linked list containing **N** nodes, the task is to find the mean of all the distinct nodes from the list whose data value is an odd Fibonacci number. **Write function to Generate Fibonacci series**
Examples:
Input: LL = 5 -> 21 -> 8 ->12-> 3 -> 13 ->144 -> 6
Output 10.5
Explanation:
Fibonacci Nodes present in the Linked List are {5, 21, 8, 3, 13, 144}
Odd Fibonacci Nodes present in the List are {5, 21, 3, 13}
Count of Odd Fibonacci Nodes is 4
Therefore , Mean of Odd Fibonacci Node Values = (5 + 21 + 3 + 13) / 4 = 10.5
Input: LL = 55 -> 3 -> 91 -> 89 -> 76 -> 233 -> 34 -> 87 -> 5 -> 100
Output:77
Explanation:
Fibonacci Nodes present in the Linked List are {55, 3, 89, 233, 34, 5}
Odd Fibonacci Nodes present in the Linked List are {55, 3, 89, 233, 5}
Count of Odd Fibonacci Nodes is 5
Therefore , Mean of Odd Fibonacci Node Values = (55 + 5 + 3 + 89 + 233) / 5 = 77