

## Chapter 4

# Lists and Linked Lists

### 4.1 Introduction

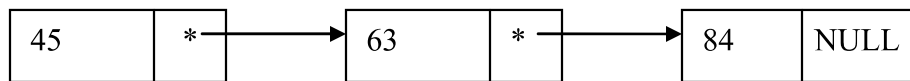
- A list is a sequence of elements. More formally, a list of items of type T is a finite sequence of elements of the set T.
- We might have a list of integers or something more complicated such as a list of records. The operations that can be performed on a list are:
  - ✓ Create the list, leaving it empty
  - ✓ Determine whether the list is empty or not
  - ✓ Determine whether the list is full or not
  - ✓ Find the size of the list
  - ✓ Add a new entry at the end of the list
  - ✓ Insert a new entry in a particular position in the list
  - ✓ Delete an entry from the list
  - ✓ Clear the list to make it empty
  - ✓ Traverse the list performing a given operation on each entry

### 4.2 Implementation of List

- There are two main ways of implementing lists:
  - a) Using contiguous storage (i.e. an array in which the elements are physically next to one another in adjacent memory locations) and
  - b) As a linked list
- The problem with contiguous storage implementation has been discussed before. These include:
  - a) Inserting in position required moving elements 'down' one position
  - b) Deletion required moving elements 'up' one position
- Linked implementation avoids the above problems.

### 4.3 What does a Linked List Look Like?

- A linked list is a collection of **nodes**, where each node contains a data along with information about the next node.
- A linked list uses non-contiguous memory locations and hence requires each node to remember where the next node is, if we are to handle all the nodes of the linked list.
- With a linked list, in addition to storing the data itself, we need to store a link (pointer) to the next element of the list. This link could be either an index to an array element (for array implementation) or a pointer variable containing the address of the next element. Such a combination of data and link is called a **NODE** of a list.



- The above linked list contains 3 nodes located at different memory locations each node has a pointer that points to the next node. The nodes that have a NULL, indicates the end of the list (last node).
- In C/C++ programs either NULL or '\0' can be used to indicate the end of the list

## 4.4 Array Implementation of a Linked List

- Memory needs to be allocated to both the elements of the list and the links. Consider the simple case when the elements are just integers then we might declare the following:

```
int data[10];
int link[10];
```

- We can now store a list up to 10 elements and their links i.e. 10 nodes
- Before we even consider programming let us imagine what the arrays might look like when holding just 4 nodes and the elements are in ascending numerical order:

Index	Data	Link
0	35	3
1	54	2

2	86	99
3	48	1

- For the first node, the data is 35 and the link is 3 meaning that the next node is located in element 3 in the array. So taking 0 as the start of the list and 99 as a dummy representing the end of the list if you follow the links you will get the sequence 35, 48, 54, 86.
- A simple program section to output the values if the list might be:

```
int i = 0;
do
{
    cout<<data[i];
    i = link[i];
}
while (i!=99)
```

- So far this just looks like a much more complicated way of storing a sequence in a simple array but watch!
- To insert the value 38 in the list does not require us to move all the elements down. We can place 38 at the end of the list and modify the links as follows:

Index	Data	Link
0	35	<b>4</b>
1	54	2
2	86	99
3	48	1
4	38	<b>3</b>

- So again, if you start at 0 and follow the links you will get the sequence 35, 38, 48, 54, 86.

- This has been accomplished by adding the new node at the end of the array and modifying of the elements down one position.
- Removing 54 from the list, which has its link the value 2, just means replacing the link to this node (i.e. 1) with this value.

Index	Data	Link
0	35	4
1	54	2
2	86	99
3	48	<b>2</b>
4	38	3

- So now, starting at 0 and following the links you would get 35, 38, 48, 86.
- The above notes do not contain functions to insert and delete but are intended to provide you with the concept of array implementation of a linked list.

## 4.5 Pointer Variable Implementation

- Consider the following:

```
struct element
{
    int number;
    int *next;
};
```

- This would appear to define a structure that consists of an integer and a pointer. It DOES, but the pointer can only point to an integer, not to a structure. The pointer must be able to point to the element structure.
- Thus we need:

```
struct node
{
```

```
int number;  
node *next;  
};
```

- Now we have a structure that can contain an integer and a pointer that can point to any variable of type “node”.
- In the following program block we created a linked list by adding elements to the front of the list.

```
node *head, *temp;  
int i,x;  
head = new node; //create the head node  
cin>>x;  
head->data = x;  
head->next = null;  
  
//creating a linked list of 5 nodes  
for(i=0; i<4; i++)  
{  
    temp = new node;  
    cin>>x;  
    temp->data = x;  
    temp->next = head;  
    head = temp;  
};
```

- We end up with a linked list where “head” is pointing to the first node and the last node having the “next” pointer equal to NULL.
- Printing out the contents could be accomplished with:

```
temp = head;  
while (temp != NULL)  
{
```

```
    cout<<temp->data;
    temp = temp->next;
}
```

- The full program could be:

```
//Program illustrating the creation of a linked list
//by adding elements to the front

#include<iostream.h>
struct node
{
    int data;
    node *next;
};

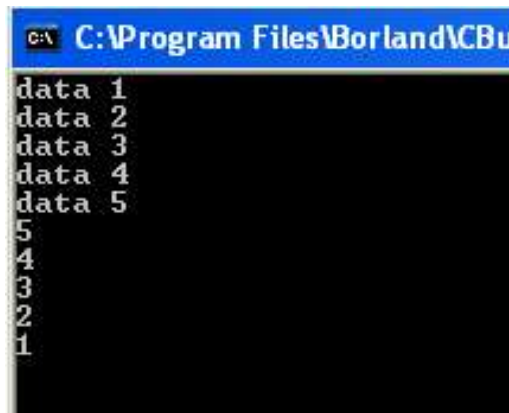
void main ()
{
    node *head, *temp;
    int i,x;

    //Creating the first node
    head = new node;
    cout<<"data ";
    cin>>x;

    head->data = x;
    head->next = NULL;
    for(i=0; i<4; i++)
    {
        temp=new node;
        cout<<"data ";
        cin>>x;

        temp->data = x;
        temp->next = head;
        head = temp;
    }
```

```
//Printing out the linked list
temp = head;
while (temp !=NULL)
{
    cout<<temp->data<<endl;
    temp = temp->next;
}
}
```



- Earlier we saw how to create a linked list by adding new nodes to the front of the list. Instead let us add new nodes to the end of the list.
- The only change is when we create a new node we make its pointer to the next element NULL (i.e. making it the end of the list) and making sure that the previous element points to the new one.

```
//Program illustrating the creation of a linked list
//by adding elements at the end at the sol 1

#include<iostream.h>
struct node
{
    int data;
    node *next;
};
```

```
void main ()
{
    node *head, *temp, *tail;
    int i,x;

    //Creating the first node
    head = new node;
    cout<<"data ";
    cin>>x;

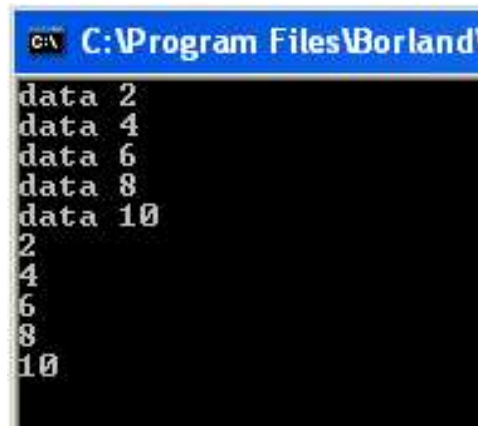
    head->data = x;
    head->next = NULL;
    tail = head;

    for(i=0; i<4; i++)
    {
        temp = new node;
        cout<<"data ";
        cin>>x;

        temp->data = x;
        temp->next = NULL;
        tail ->next = temp;
        tail = temp;
    }

    //Printing out the linked list
    temp = head;
    while (temp !=NULL)
    {
        cout<<temp->data<<endl;
        temp = temp->next;
    }
}
```





```
C:\Program Files\Borland\
data 2
data 4
data 6
data 8
data 10
2
4
6
8
10
```

- Another solution to adding nodes to the end of the list would be:

```
//Program illustrating the creation of a linked list
//by adding elements at the end at the sol 2

#include<iostream.h>
struct node
```

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

void main ()
{
    node *head, *temp;
    int i,x;

    //Creating the first node
    head = new node;
    cout<<"data ";
    cin>>x;

    head->data = x;
    head->next = NULL;
    temp=head;

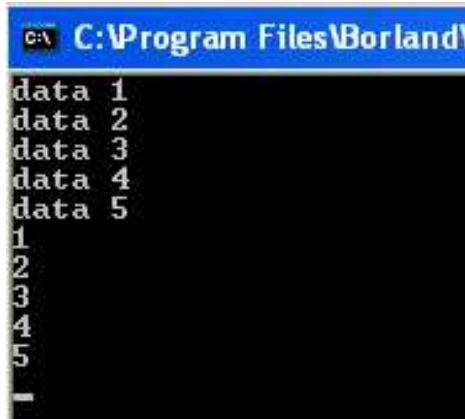
    for(i=0; i<4; i++)
    {
        temp->next = new node;
        temp = temp->next;

        cout<<"data ";
        cin>>x;

        temp->data = x;
        temp->next = NULL;
    }

    //Printing out the linked list
    temp = head;
    while (temp !=NULL)
    {
        cout<<temp->data<<endl;
        temp = temp->next;
    }
}
```

```
}
```



A screenshot of a DOS command prompt window with a blue title bar that reads "C:\Program Files\Borland\". The window has a black background with white text. It displays a linked list with 5 nodes. The first five lines are "data 1", "data 2", "data 3", "data 4", and "data 5". The next five lines are "1", "2", "3", "4", and "5", representing the next pointers of each node. The cursor is at the end of the fifth line.

## 4.6 Insertion and Deletion

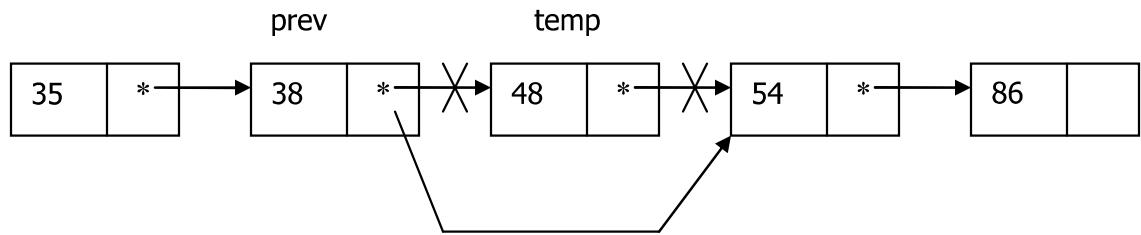
- There are two main operations which are required to maintain data in a linked list: insertion and deletion. While creating a linked list, we were always inserting data either at the end or at the beginning of the list.

### 4.6.1 Deletion

- If we need to insert an element between existing nodes, this needs to be done in a special way. Similarly, deletion also has to be handled separately. Once you are familiar with both of these, you can attempt to create a sorted list.
- Let us consider the following linked list, suppose we want to delete the node with data 48 (temp). We have to first consider its previous and succeeding node. Then, the idea of deletion would be simple if we make the node with data 38 (prev) points to the node with data 54, we will in fact exclude the node 48. Therefore, we have done the following:

```
prev -> next = temp -> next;
```

- The node with data 48 will still be in the memory, but it is out of the list, we can use delete to de-allocate it.



- The first task in the deletion program is to locate the previous node of the node to be deleted.

```
node *head, *temp, *prev;
temp = head;

while ( (temp !=NULL) && (temp->data!=48) )
{
    prev = temp;
    temp = temp->next;
}
```

- Then we can delete it with the following instruction

```
if (temp != NULL)
{
    prev->next = temp->next;
    delete temp;
}
```

- The condition `temp != NULL` is done to make sure that the node to be deleted exist in the list.
- The program to delete node is as follows:

```
//Program illustrating the deletion of a node from
// a linked list

#include<iostream.h>
struct node
```

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

void main ()
{
    clrscr();
    node *head, *temp, *prev;
    int i,x,y;

    //Creating the first node
    head = new node;
    cout<<"data ";
    cin>>x;

    head->data = x;
    head->next = NULL;
    temp = head;

    for(i=0; i<4; i++)
    {
        temp->next = new node;
        temp = temp->next;

        cout<<"data ";
        cin>>x;

        temp->data = x;
        temp->next = NULL;
    }

    cout<<"Enter data to be deleted:";
    cin>>y;

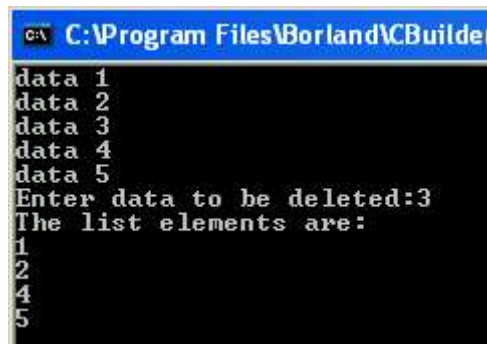
    //Deleting a node
    temp = head;
    while((temp != NULL) && (temp->data != y))
    {
```

```
        prev = temp;
        temp = temp->next;
    };

    if(temp != NULL)
    {
        prev->next = temp->next;
        delete temp;
    }

    //Printing out the linked list
    cout<<"The list elements are:"<<endl;
    temp = head;

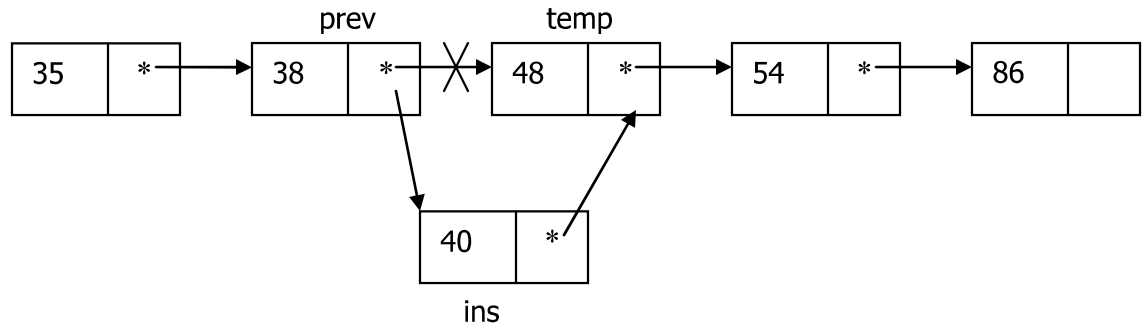
    while (temp != NULL)
    {
        cout<<temp->data<<endl;
        temp = temp->next;
    };
}
```



```
C:\Program Files\Borland\CBuilde
data 1
data 2
data 3
data 4
data 5
Enter data to be deleted:3
The list elements are:
1
2
4
5
```

### 4.6.2 Insertion

- Insertion can be specified before or after a node which has specific data. For example let us consider the case below:



- Suppose we want to insert a node with data 40 before the node with data 48 and after the node 38.
- The next node for 38(prev) is 48(temp). This should become 40(ins), and to maintain the continuity, the next node of the inserted node (40) should be 48.
- This can be done syntactically as follows:

```
ins -> next = temp;
prev -> next = ins;
```

- Here is the full program to insert a node in a list.

```
//Program illustrating the insertion of a node to a
//linked list

#include<iostream.h>
struct node
{
    int data;
    node *next;
};
```

```
void main ()
{
    clrscr();
    node *head, *temp, *prev, *ins;
    int i,x,y,z;

    //Creating the first node
    head = new node;
    cout<<"data ";
    cin>>x;

    head->data = x;
    head->next = NULL;
    temp = head;

    for(i=0; i<4; i++)
    {
        temp->next = new node;
        temp = temp->next;

        cout<<"data ";
        cin>>x;

        temp->data = x;
        temp->next = NULL;
    }

    cout<<"Enter data to be inserted:";
    cin>>y;

    cout<<"The new data should be inserted before which
        node?";
    cin>>z;

    //Inserting a node
    temp = head;
    while( (temp != NULL) && (temp->data != z) )
    {
        prev = temp;
        temp = temp->next;
```

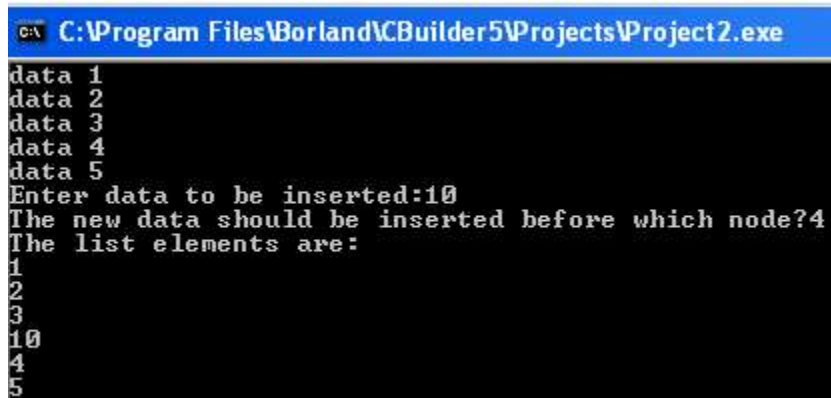


```
};

if(temp != NULL)
{
    ins = new node;
    ins->data = y;
    ins->next = temp;
    prev->next = ins;
}

//Printing out the linked list
cout<<"The list elements are:"<<endl;
temp = head;

while(temp != NULL)
{
    cout<<temp->data<<endl;
    temp = temp->next;
};
}
```



```
C:\Program Files\Borland\VCBuilder5\Projects\Project2.exe
data 1
data 2
data 3
data 4
data 5
Enter data to be inserted:10
The new data should be inserted before which node?4
The list elements are:
1
2
3
10
4
5
```

### 4.6.3 The special case of the head node

- Suppose we want to delete the head node or we want to insert a new node before the head node. There are two special things about the head node. The head node once assigned is usually not changed. Also for the head node there is no previous node. Hence the case of the head node needs to be handled differently.
- For deleting the head node, we simply copy the contents of the second node to the head node.

```
temp = head;
head = head->next;
delete temp;
```

- For inserting before the head node, we first create a temp node in which we copy the head node data. Then we write the new data into the head node and insert the temp node before the head node. If the data to be inserted is x, then we will have:

```
temp = new node;
temp->data = x;
temp->next = head;
head = temp;
```

## 4.7 Doubly Linked List

- A linked list where the nodes only contain one link is actually called a singly linked list. With such a list you can only traverse in one direction.
- To be able to traverse in both directions requires a doubly linked list. Pictorially it would look as follows:

