# LINKED LISTS

## Chapter Objectives

At the completion of this chapter, you would have learnt:

? ? Adding Nodes;

? ? Deleting Nodes;

? ? Searching Nodes;

? ? Garbage Collection

*Definition : A linked list is a dynamic variable that consists of nodes containing valuable data and each node in the linked list is connected by pointers. A pointer variable called Head is used to point to the first node of the linked list.*

Structure illustrated below is called linked list.

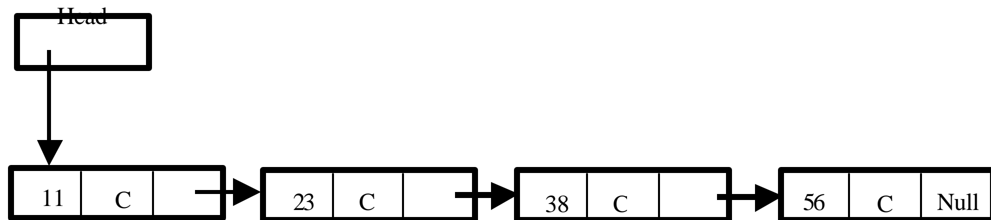



***Figure 3-1***

Each item in the list is called a ***node*** and contains two fields, an ***information*** field and a ***next address(pointer)*** field. The information field holds the actual element on the list. The next address field contains the address of the next node in the list. The entire linked list is accessed from an external pointer ***Head*** that points to the first node of the linked list. ***Head*** is NULL if the linked list is empty.

Linked lists are used for many of the same things that arrays are used for, namely for storing data in the list. Advantages of linked list compared to array is that the size of a linked list can change during program execution and also it is easier to insert and delete nodes in a linked list than in an array. For these reasons, linked lists are preferable to arrays for some applications.

# 3.1 Building a Linked List

## 3.1.1 Declaration

*To declare the pointers and node of a linked list :-*

***Declaration 1***

```
struct Node{
      int Data;
      struct Node *Link;
};

typedef struct Node *NodePointer;

void main()
{
      NodePointer Head;
}
```

***This Declaration contains a very simple information field known as Data which contains integer only.***

***Declaration 2***

```
struct DataRecord{
      int Number;
      char Grade;
};

struct Node{
             DataRecord Data;
             struct Node *Link;
};

typedef struct Node *NodePointer;

void main()
{
      NodePointer Head;
}
```

***This Declaration is more complex where the information field contains more than 1 field.***

Throughout the whole chapter, the coding will be based on declaration 1.

### 3.1.2 Adding Nodes To The Front Of The Linked List

*Purpose : To create a new node to be linked to the front of the list*

In order to have a large linked list, a program must be able to add nodes to the linked list in a systematic way. We next describe one simple way to insert nodes in a linked list. It will turn out that the function will work even if we start with an empty list. However, the process is clearer if we first assume that the list already has at least one node in it.

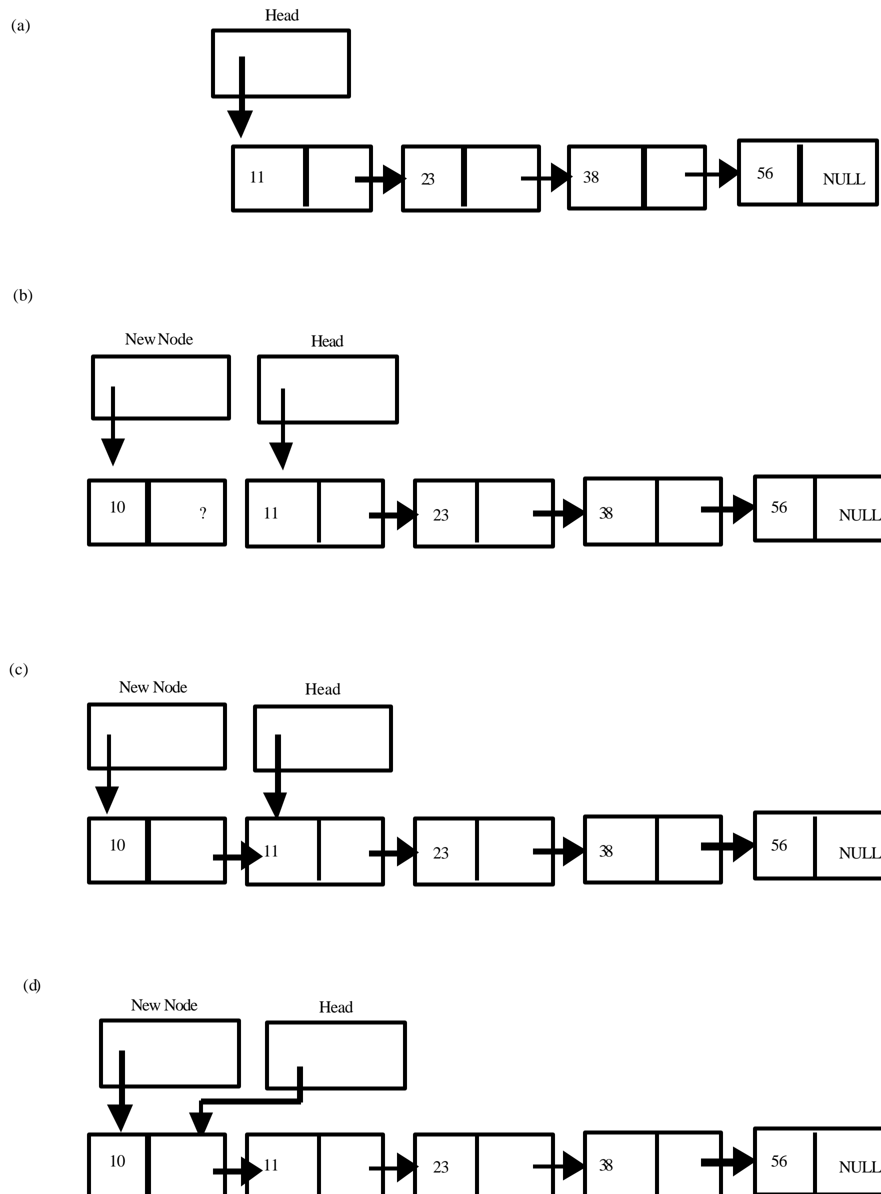



***Figure 3.2***

## Algorithm

In order to insert the data into the front of the linked list, the function will need to use *malloc* to create a new node. The data from Number is then copied into the new node, and the new node is inserted at the head of the list. Here, we are going to use a local pointer, NewNode to store the data.
The complete process can be summarized as follows :

1. Create a new dynamic variable pointed to by NewNode;
2. Place the data in this dynamic variable;
3. Make NewNode's link point to the head (first node) of the original linked list;
4. Make Head point to the node that NewNode is pointing

The figure above gives the algorithm in diagrammatic form.

## Implementation

```
void InsertInFront(NodePointer &Head,int Number)
{
   NodePointer NewNode;
   NewNode = (NodePointer)malloc(sizeof(struct Node));
   //Create a new memory location
   NewNode->Data = Number;
   //Store the new Number into the new location
   NewNode->Link = Head;
   //Link the NewNode's link to Head
   Head = NewNode;
}
```

### 3.1.3 The Empty List

A linked list is named by naming a pointer that points to the head of the list. To specify an empty list, the normal thing to do is to set this pointer equal to NULL:

Head = NULL;

Whenever you design a function for manipulating a linked list, you should check to see if it works on the empty list. If it does not, then it may be possible to add a special case for the empty list. If you cannot design the function to apply to the empty list, then the program must be designed to handle empty lists in some other way or to avoid them completely. One way to avoid empty lists is to add a dummy node that contains node real data but marks the end of the list and is never deleted.

### 3.1.4 Losing Nodes

You might be tempted to write the function InsertInFront using the pointer variable Head directly, instead of using the local pointer variable NewNode to construct a new node. If we were to try, we might start the function as follows :

```
Head = (NodePointer)malloc(sizeof(struct Node));
Head->Data = Number;
```

Now, f we do this way, Head will point to another new memory location and this causes the other nodes not pointed by any external pointer at all. These nodes are called losing nodes. Refer to figure 3.3 below. What happens to those losing nodes? They became useless memory locations which cannot be reused again. This will waste memory locations.

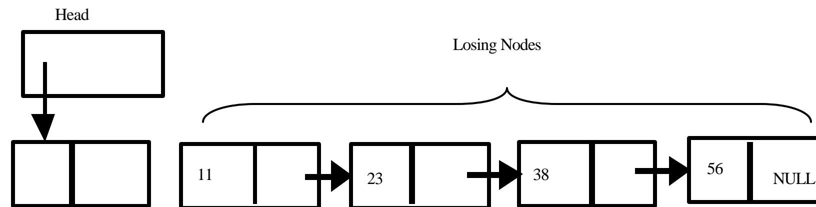



***Figure 3.3***

The same thing happen if step 4 of the algorithm is done before step 3. Therefore, it is important to re-link the pointers in the right order when manipulating a linked list.

## 3.1.5 Deleting Nodes at front of the list

*Purpose : To remove a node from the front of the list without loosing the other nodes.*

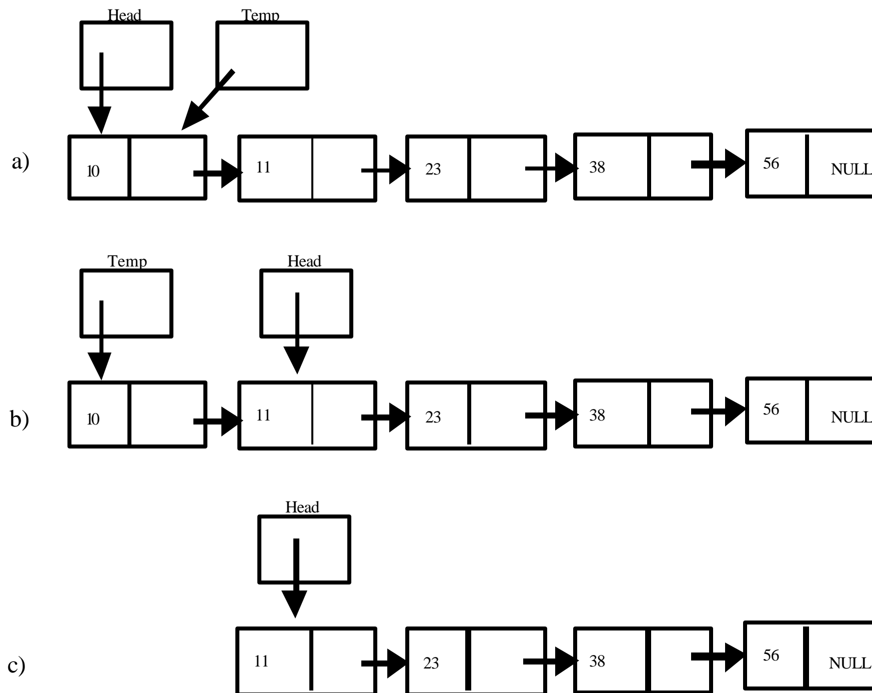



***figure 3.4***

### Algorithm

In order to delete the front node, we need another temporary pointer variable instead of directly free Head.

The complete process can be summarized as follows :

a) Set Temp to point to Head
b) Move Head to the next node
c) Remove Temp

### Implementation

```
void DeleteFront(NodePointer &Head)
{
  NodePointer Temp;
  if (Head == NULL)
    printf("Linked list is empty..Cannot delete!");
  else
    {
          Temp = Head;
          Head = Head->Link;
          //Move Head to the next node
          free(Temp);
    }
}
```

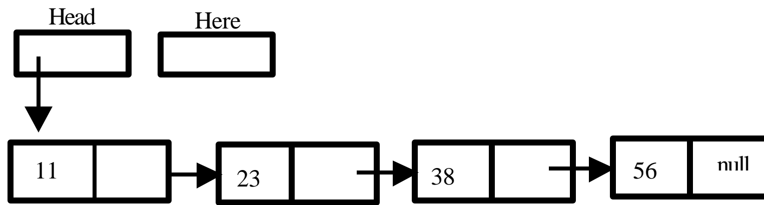
We, have to check for empty list. This is because, if we don't check for empty list, the statement *Head = Head->Link & free(Temp)* will cause a problem.

## 3.1.6 Searching a List

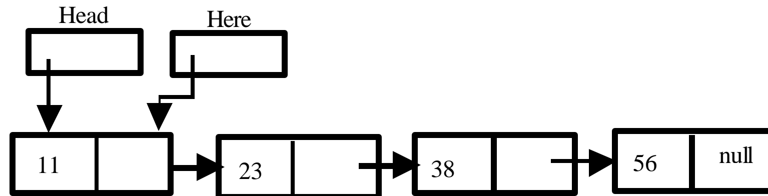
*Purpose : To search for a particular node in the list. Here, we will use Linear Search to search for the node.*

The algorithm is shown in the figure below:-

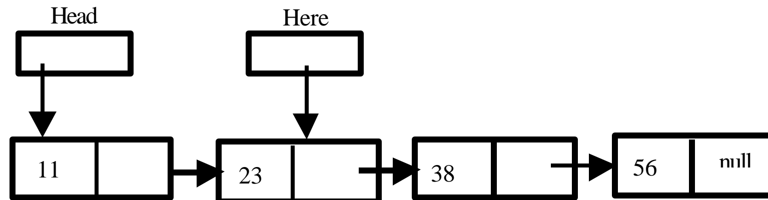
(a) Key = 38




(b) Key = 38




(c) Key = 38




(d) Key = 38

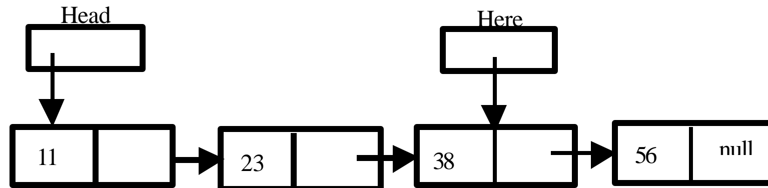



***figure 3-5***

## Algorithm

The only way to move around a linked list, or any other data structure made up of nodes and pointers, is to follow the arrows. So we will place the pointer Here at the first node and then move it from node to node, following the pointers until we find a node containing the integer Key or until we encounter the end of the list. The technique is diagrammed in the above figure from (b) to (d). Since empty lists present minor problems that clutter our discussion, we will first assume that the linked list contains at least one node. This search technique yields the following algorithm :

```
Make Here point to the Head (first node) in the list

While (Here is not pointing to a node containing Key) and (Here is not pointing
to the last node) do
        Make Here point to the next node in the list
```

## Implementation

```
NodePointer Searching(NodePointer Head,int Key)
{
      NodePointer Here;
      Here = Head;
      while (Here != NULL)
            if (Here->Data == Key)
                  //Key is found in the list
                  return (Here);
            else
                  //Move Here to the next node
                  Here = Here->Data;
      return (NULL);
      //return NULL if the node is not found
}
```

The function above will return a pointer out. The pointer return will either point to the node which is found or NULL(means not found). Key is the search argument and is used to compare with all the nodes in the list.

### 3.1.7 Insert a Node at the End of the list.

*Purpose : To create a new node to be inserted at the end of the list.*

The algorithm is shown in the figure below:-

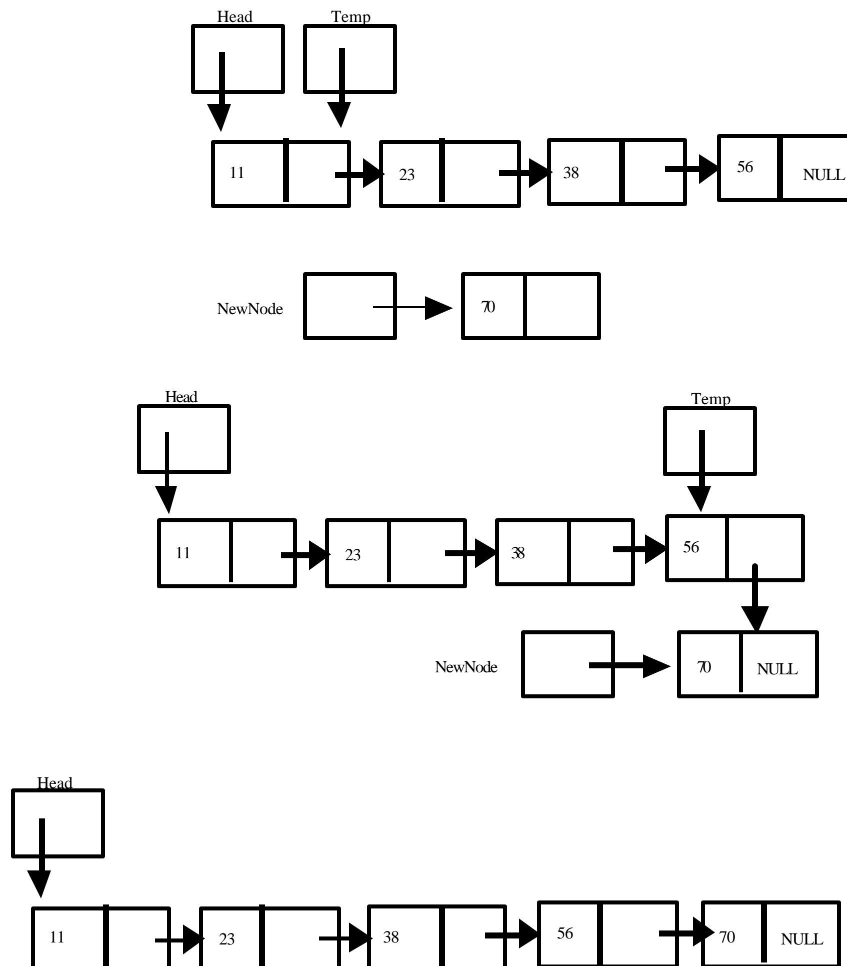



***figure 3.6***

## Algorithm

The concept here is very simple. We will use the last node's link to point to the new memory location. Here, we will need 2 temporary pointers variable. One called *NewNode* used to point to the new memory location to be inserted whereas the other one *Temp* is used to point to the last node in the list. To make Temp point to the last node, we will have to move Temp starting from Head to the last node. Since the memory location is supposed to be inserted at the end of the lst, make sure that the new node's link set to NULL to indicate the end od the list. The steps are shown below:-

a) Create a new node pointed by *NewNode*.
b) Assign *Temp* to Head
c) Move *Temp* to the last node
d) Assign *Temp*'s link to NewNode

## Implementation

```
void InsertAtEnd(NodePointer &Head, int Number)
{
   NodePointer NewNode,Temp;
   //Creating a new memory location to be inserted
   NewNode = (NodePointer)malloc(sizeof(struct Node));
   NewNode->Data = Number;
   NewNode->Link = NULL;
   //if list is empty, just assign Head to NewNode
   if (Head==NULL)
      Head = NewNode;
   else
      {
      //if list not empty, move Temp to the last
      //node.
            Temp = Head;
            while (Temp->Link!=NULL)
                  Temp = Temp->Link;
            //When Temp is at the last node, assign
            //Temp's link to NewNode
            Temp->Link = NewNode;
      }
}
```

### 3.1.8 Delete a Node at the End of the List

*Purpose :To remove the last node from the linked list.*

The algorithm is shown in the figure below:-

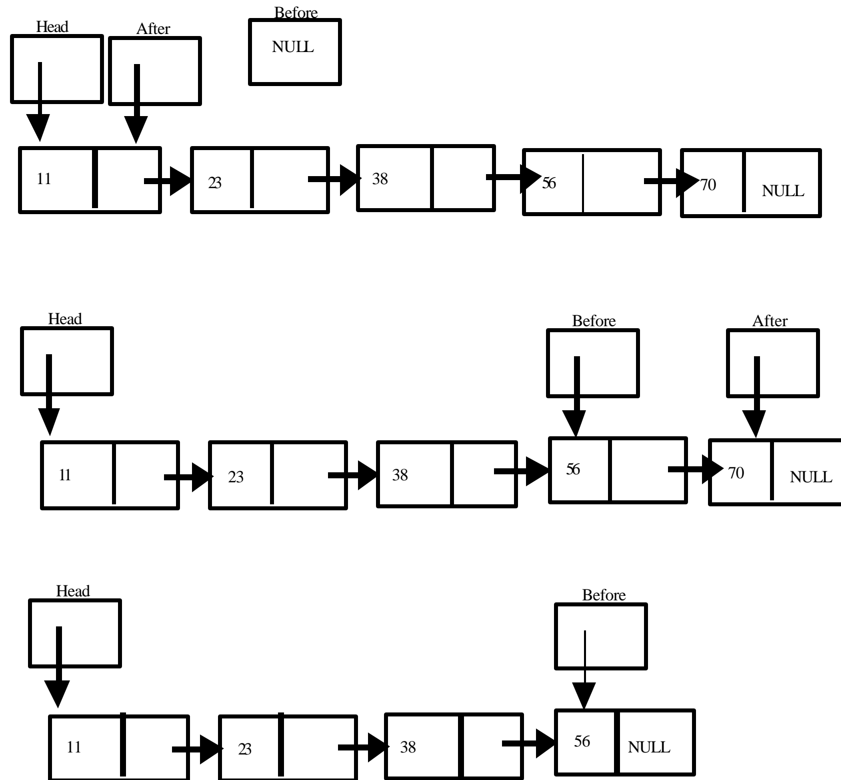



***figure3.7***

## Algorithm

Here, we need to 2 variables called *Before* and *After*. We will move the two pointers to the end of the list with *After* points to the last node and *Before* points to the second last node. Here we will remove the node pointed by *After* and with that the node pointed by *Before* is now becomes the last node so the Link field should be assigned to NULL. The steps are shown below:-

a) Assign After to Head and before to NULL
b) Move After to the last node and at the same time move Before too
c) Assign Before's Link to NULL
d) Remove After.

### Implementation

```
void DeleteEnd(NodePointer &Head)
{
   NodePointer Before,After;
   if (Head==NULL)
      printf("Linked List is empty!..Cannot Delete");
   else
   {
      Before = NULL;
      After  = Head;
      while (After->Link != NULL)
      {
            Before = After;
            After  = After->Link;
      }
      //Assign Before's link to NULL
      Before->Link = NULL;
      //Dispose After
      free(After);
   }
}
```

The above function works only is the list has more than 1 node. If the lst has only 1 node, then the statement *Before->Link* will give error because at that time, Before is NULL. So we need to change the above function to:-

```
void DeleteEnd(NodePointer &Head)
{
   NodePointer Before,After;
   if (Head==NULL)
      printf("Linked List is empty!..Cannot Delete");
   else
   {
      Before = NULL;
      After  = Head;
      while (After->Link != NULL)
      {
            Before = After;
            After  = After->Link;
      }
      if (Head->Link == NULL)
            //if the list has only 1 node. set the
            //Head to NULL
            Head = NULL;
      else
            Before->Link = NULL;
      free(After);
   }
}
```

### 3.1.9 Inserting Nodes Into A Sorted Linked List

We, next design a function to insert a node at a specified place in a linked list. Since we may want the nodes in some particular order, such as in numeric order, we cannot simply insert the node at the beginning (head) of the list nor at the end of the list. We will therefore design the function to insert a node between two specified nodes in a linked list. We assume that some other function or program part has placed two pointers called Before and After pointing to two nodes in the list, as shown in the Figure 3-8 below.

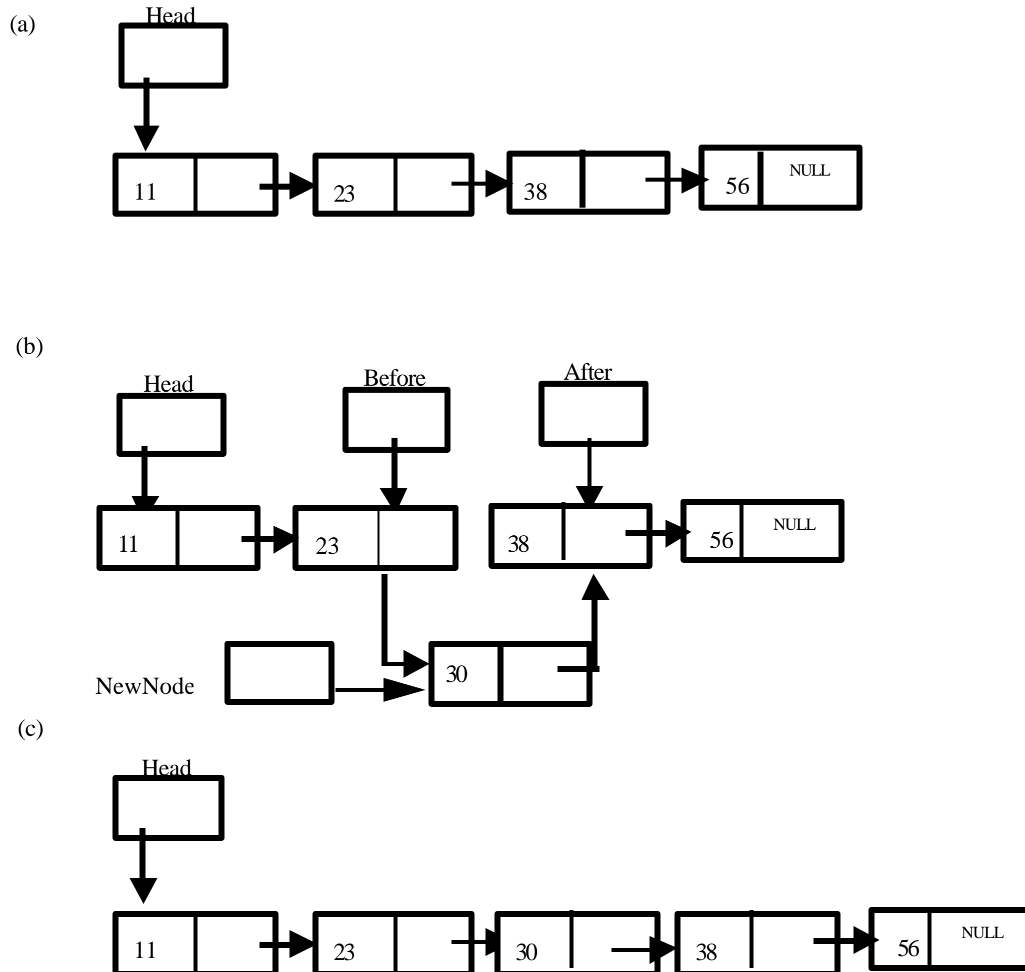



***Figure 3-8***

## Algorithm

Here, we will use two pointers, Before and After. Starting from Head, we will move *After* to a node where its value is greater than the new node's value. In the above case, *After* moves until it reaches the node where the value is 38 (greater than 30). *Before* will move together with *After*. When they are at the correct location, we will assign *Before*'s link to NewNode and assign *NewNode*'s link to *After*.

## Implementation

```
void InsertInBetween(NodePointer &Head,
                     NodePointer After,
                     NodePointer Before)
{
   NodePointer NewNode;
   NewNode = (NodePointer)malloc(sizeof(struct Node));
   NewNode->Link = NULL;
   NewNode->Data = Number;
   NewNode->Link = After;
   Before->Link = NewNode;
}
```

## Comparison to Arrays

By using the function Insert, we can maintain the linked list in numerical order without rewriting existing nodes. We could squeeze a new node into the correct position simply by adjusting two pointers. Furthermore, this is true no matter how long the linked list is or where in the list we want the new record to go. If we had instead used an array of records, then much, and in extreme cases, all of the array would have to be copied over in order to make room for a new record in the correct spot. In spite of the overhead involved in positioning the pointers, inserting into a linked list is frequently more efficient than inserting into an array.

## 3.1.10 Deleting Nodes in Between 2 Nodes

Deleting a node from a linked list is also quite easy. Figure 3-9 illustrates the method. Once the pointers Before and Discard have been positioned (to position Discard, a search must be done to find the node that is to be deleted), all that is required to delete the node is the following C statement :

```
Before->Link = Discard->Link;
free(Discard);
```

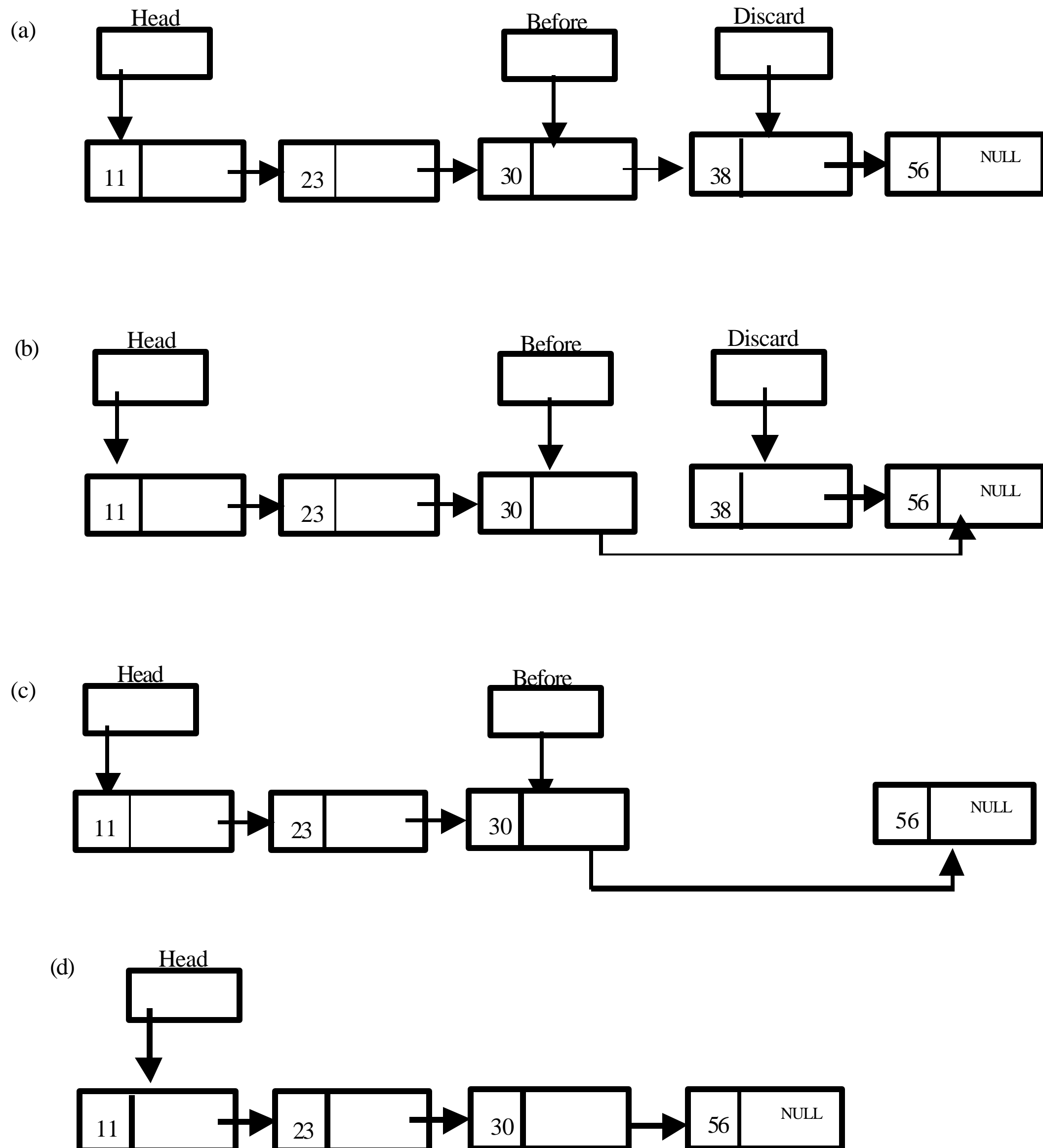



***Figure 3-9***

### Algorithm

Assuming that other functions had found the node which is to be deleted. We will now make sure that before the node is deleted the linked list is linked properly. Here we will use two pointers, *Before* and *Discard*. *Discard* will be pointing to the node to be deleted and *Before* will be pointing to the node before *Discard*.

### Implementation

```
void DeleteInBetween(NodePointer &Head,
                     NodePointer Discard,
                     NodePointer Before)
{
      Before->Link = Discard->Link;
      free(Discard);
}
```

## 3.2 Details of Implementation

In order to program in a high level language such as C, you do not need to know how the language is implemented, any more than you need to understand the workings of the human larynx or of the human brain in order to use the English language. C programs are implemented as the machine code produced by the C compiler, and all you need to know is that the machine code makes the input and output behave as we have described for the language C. Still, it is sometimes helpful, and invariably interesting, to know some details of the implementation. This is particularly true of pointers and dynamic variables. Since the description of their implementation is very much more concrete than the high level description of C pointers, many people find it easier to understand pointers in terms of their implementation. Additionally, it gives you a good idea of how the notion of pointers can be implemented in other programming languages, including many high level programming languages that do not have pointers as a basic predefined construction.

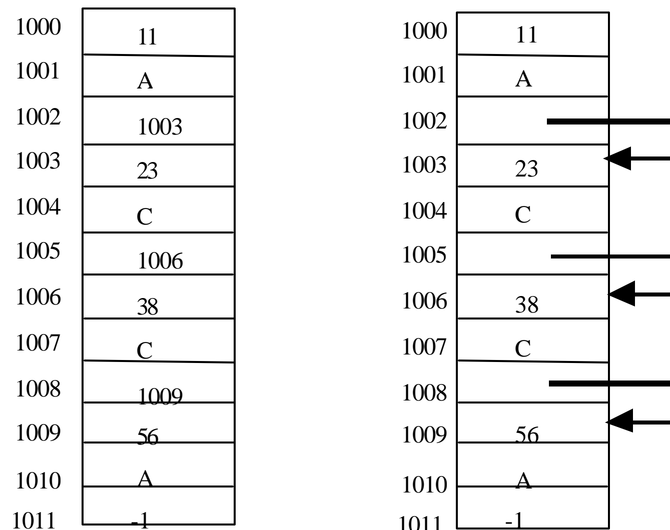



***Figure 3-10***

Recall that a computer's main memory consists of a very long sequence of numbered memory locations, and that each memory location can hold one string of binary digits, which we can interpret as a data value of some simple type, such as an integer or a character. A computer memory is structured much like a very large one-dimensional array.

To be concrete, let us say that we want to implement a linked list of the type shown in Figure 3-1. Each node will contain an integer field and a field of type char in addition to the one pointer field. One way to do this is to allocate three adjacent memory locations for each dynamic variable that is to serve as a node in the linked list. One of the three locations will hold the integer in the node, one will hold the character, and the third location will hold some integer that can be interpreted as a pointer to a node.

What can be interpreted as a pointer to one of these dynamic variables? These dynamic variables are implemented as three adjacent memory locations. Hence, one way to name one of these dynamic variables is to name the three addresses of these locations. That is exactly what we will do; however, we will use only the first address, since the other two are trivial to compute from the first one. In our implementation a dynamic variable of the type under discussion is just three adjacent memory locations : the first two hold the integer and character values, and the third holds the pointer. In this implementation the pointer is realized as the address of the first of the three memory locations that represent the dynamic variable that is pointed to.

By way of example, consider Figure 3-10. It shows a possible implementation of the linked list shown in Figure 31. The NULL pointer is indicated by the number minus one. Minus one is used for NULL because we know there is no location with that address. Any other negative number would do as well. The right-hand figure is an abstraction that ignores the particular address numbers used. Since the particular address numbers used are not important to the realization, that right-hand Figure is easier to deal with.

## Adding Nodes

As a program proceeds to add and delete nodes from a linked list, the picture of memory becomes a good deal more intricate. Suppose we wish to add a node containing the integer 30 and the character `B' in a position that will keep the nodes in the linked list in numeric order; the result should be as shown in Figure 3-8. Figure 3-11 shows the configuration of memory after the dynamic variable has been added. Note that the dynamic variable for this node was implemented with the next three available memory locations. The way to think about a linked list is as if the new node is squeezed in. However, in this implementation all the old nodes stay where they are. Only the values of the pointer fields change.

*Original List*

| Address | Value |
|---|---|
| 1000 | 11 |
| 1001 | A |
| 1002 | 1003 |
| 1003 | 23 |
| 1004 | C |
| 1005 | 1012 |
| 1006 | 56 |
| 1007 | C |
| 1008 | -1 |
| 1009 | Free |
| 1010 | Free |
| 1011 | Free |
| 1012 | 38 |
| 1013 | C |
| 1014 | 1006 |

*After adding a new node between node with integer number 23 and 38*

| Address | Value |
|---|---|
| 1000 | 11 |
| 1001 | A |
| 1002 | 1003 |
| 1003 | 23 |
| 1004 | C |
| 1005 | 1009 |
| 1006 | 56 |
| 1007 | A |
| 1008 | -1 |
| 1009 | 30 |
| 1010 | B |
| 1011 | 1012 |
| 1012 | 38 |
| 1013 | C |
| 1014 | 1006 |

***Figure 3-11***

## Deleting Nodes

Figure 3-12 shows the implementation of the same linked list after adding the node containing 30 and `B', as just described, and then deleting the node containing 38 and `C'.

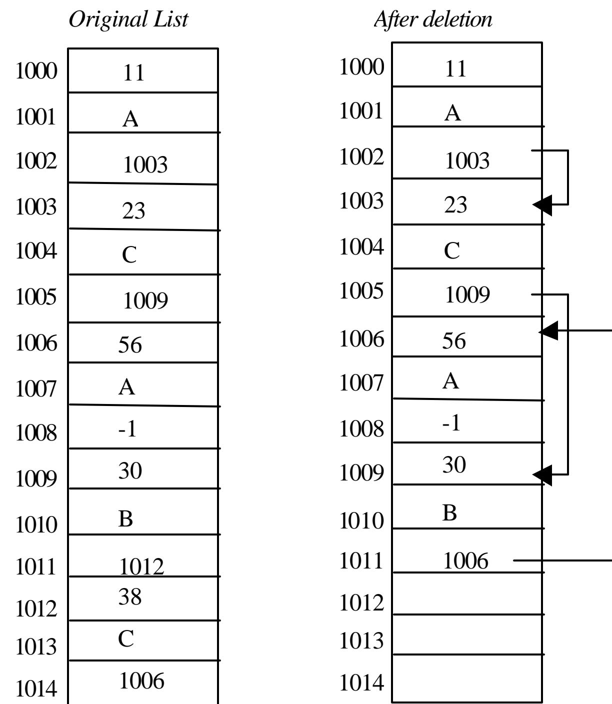



***Figure 3-12***

## Garbage Collection

Notice that as we add nodes, the pattern of arrows gets to be rather messy, but that need not concern us. Ordinarily, we need not be concerned with the actual memory addresses when we use pointers and dynamic variables. We need only think in terms of an abstraction of the pointer structure that ignores the actual locations of the dynamic variables. If we have enough memory, we can get away with thinking only on an abstract level, which ignores all the details of the particular memory addresses used. Unfortunately, there is some danger of wasting memory if we think exclusively on this abstract level. Look again at the memory configuration shown in Figure 3-13. Notice that the dynamic variable in locations 1009, 1010, and 1011 still has an integer, a letter, and a pointer in it. Yet the node represented by that dynamic variable is no longer on the linked list. The program will never be able to use the dynamic variable stored in locations 1009 through 1010, and so those memory locations should be made available for other uses. Yet if we continue to add dynamic variables at the bottom of memory, then locations 1006 through 1008 will never be reused. Locations like 1009, 1010, and 1011 are frequently referred to by the technical term garbage -- not a very dignified word, but a descriptive one and the one that is generally used. A good implementation would keep track of these garbage memory locations and reuse them. Locating such garbage memory locations so that they can be reused is called, appropriately enough, garbage collection.

Many implementations of C do not have very good garbage collection, and the system must be given some help in order to perform this task. Specifically, the system must be told which dynamic variables are garbage. This is what the free command does.

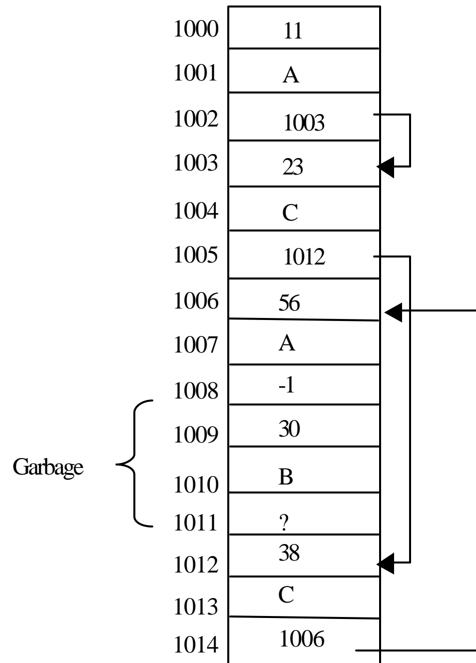



***Figure 3-13***

# Exercises for Linked Lists

1. What is the major advantage of using pointer representations of lists instead of using arrays to store list-like structures?
2. When an element is deleted from a linked list represented using pointers, it is automatically returned to the heap. True/False?
3. All pointers to a node that is returned to the heap are automatically reset to NULL so they cannot reference the node returned to the heap. True/False?
4. Define the term `Linked List'. How are the list nodes connected?
5. Why would the following code fragment cause a run-time error? Assume Link is a pointer and Data is an integer. How could it be fixed?

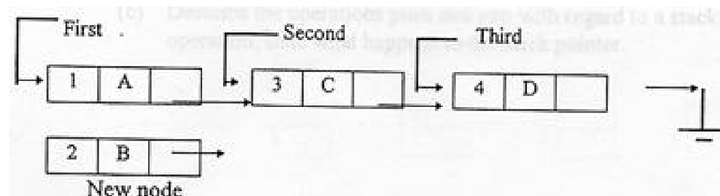
```
while (Head!=NULL)
      Head = Head->Link;
printf("Last data item in list is: ",Head->Data);
```

6. When is it advantageous to use a linked list data structure? When would an array be a better choice?
7. Write a C function to interchange pointers p and q, so that after the function is performed, p will point to the node to which q formerly pointed, and vice versa.
8. Write a function that will delete of the nodes in the linked list where the Data's value in the node is equals to 30. Make sure the list does not lose any nodes.
9. Write a function ClearList that disposes of all the nodes in a linked list and sets the list to be empty.
10. (a) What is meant by the term empty list?
    (b) Briefly explain how a new item is added to the front of the linked list
11. When items are added to and removed from linked lists it is possible for garbage to be created. Briefly explain this statement and briefly describe the process known as garbage collection. State what is done in the C language to help with garbage collection.
12. What function is performed by the statement Head = Head->Link;?
13. Given

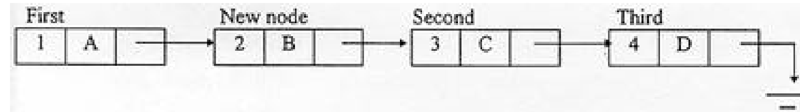
```
struct Node{
                int Number;
                char Grade;
            }
```

Why can this not be used to make a node in a linked list? What must be added to it to make a structure which could be used?

14. The new node shown below is to be added between the First and Second nodes. Give C codes to show how this could be done.




After adding the new node the linked list looks like this:




Give C codes to show how you would delete the New node without causing other nodes to be lost.

15. Describe the process necessary to insert an item at the correct place in a sorted list..

16. What is an empty list? Write a C statement which would set a list to be empty and a function will return 0 when the list is empty otherwise return 1