LINKED LIST

OBJECTIVES:

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

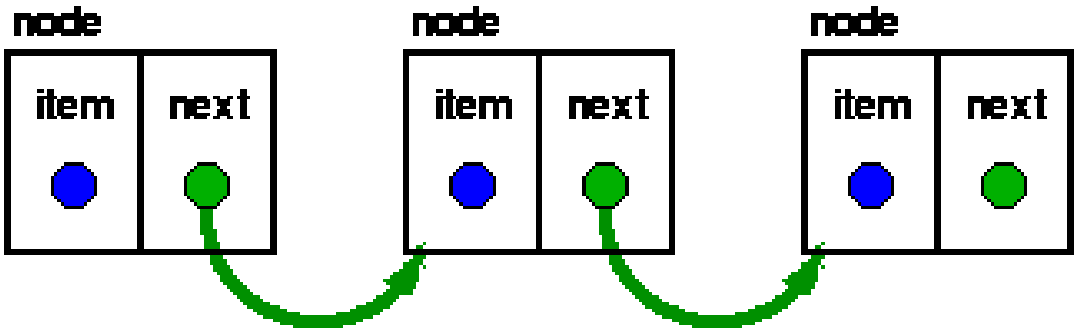
- Adding Nodes
- Deleting Nodes
- Searching Nodes
- Garbage Collection

LINKED LIST

- Linked lists and arrays are similar since they both store collections of data.
- The terminology is that arrays and linked lists store "elements" on behalf of "client" code.
- The specific type of element is not important since essentially the same structure works to store elements of any type.
- One way to think about linked lists is to look at how arrays work and think about alternate approaches.

LINKED LIST

- *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.*



Array Review

- Arrays are probably the most common data structure used to store collections of elements. In most languages, arrays are convenient to declare and the provide the handy [ ] syntax to access any element by its index number.
- The following example shows some typical array code and a drawing of how the array might look in memory. The code allocates an array int Scores[100], sets the first three elements set to contain the numbers 1, 2, 3 and leaves the rest of the array uninitialized...

```
void ArrayTest() {
int scores[100];
// operate on the elements of the scores array...
scores[0] = 1;
scores[1] = 2;
scores[2] = 3;
}
```

- 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.

Building 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

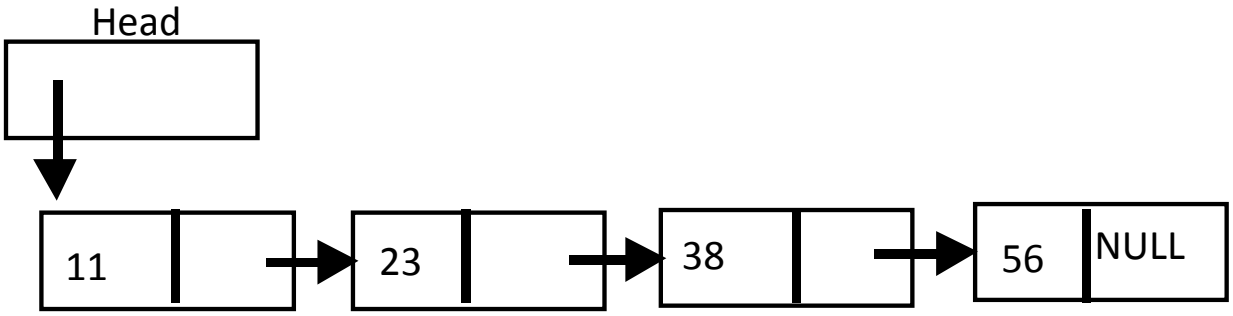
```
struct DataRecord{
        int Number;
        char Grade;
};
 struct Node{
                DataRecord Data;
                struct Node *Link;
};
 typedef struct Node *NodePointer;
void main()
{
        NodePointer Head;
}
```

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

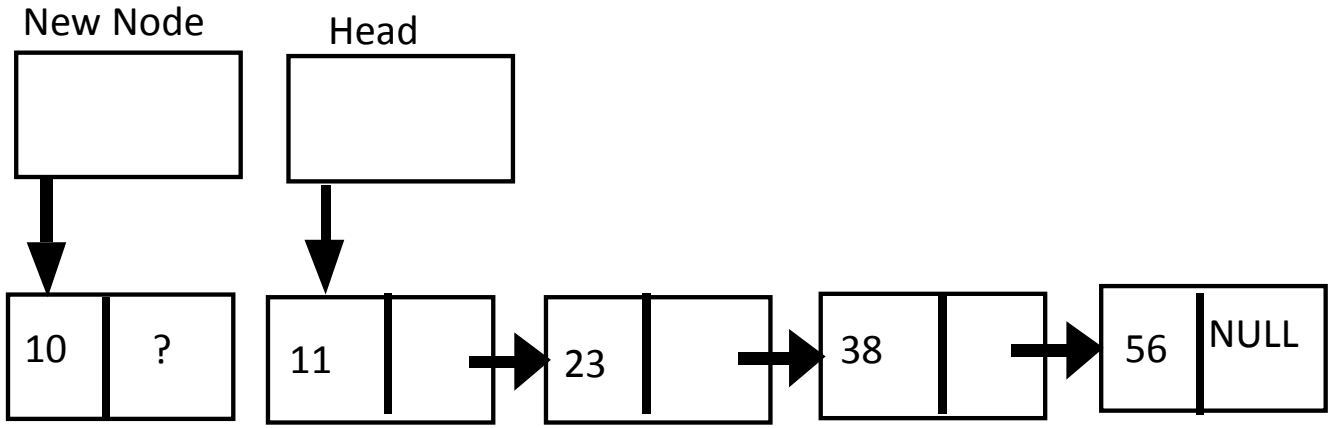
1. Adding Nodes To The Front Of The Linked List

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

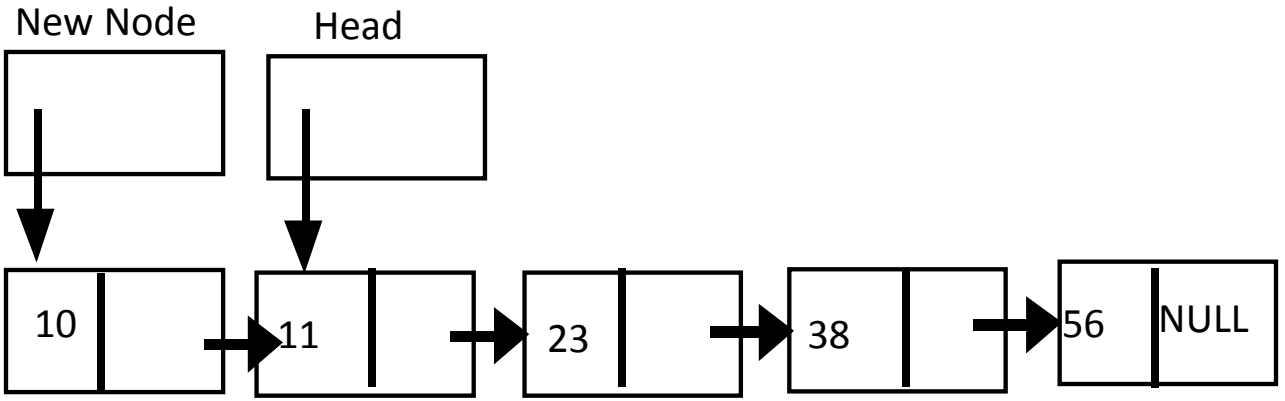
✓In order to have a large linked list, a program must be able to add nodes to the linked list in a systematic way. We next describe one simple way to insert nodes in a linked list. It will turn out that the function will work even if we start with an empty list. However, the process is clearer if we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



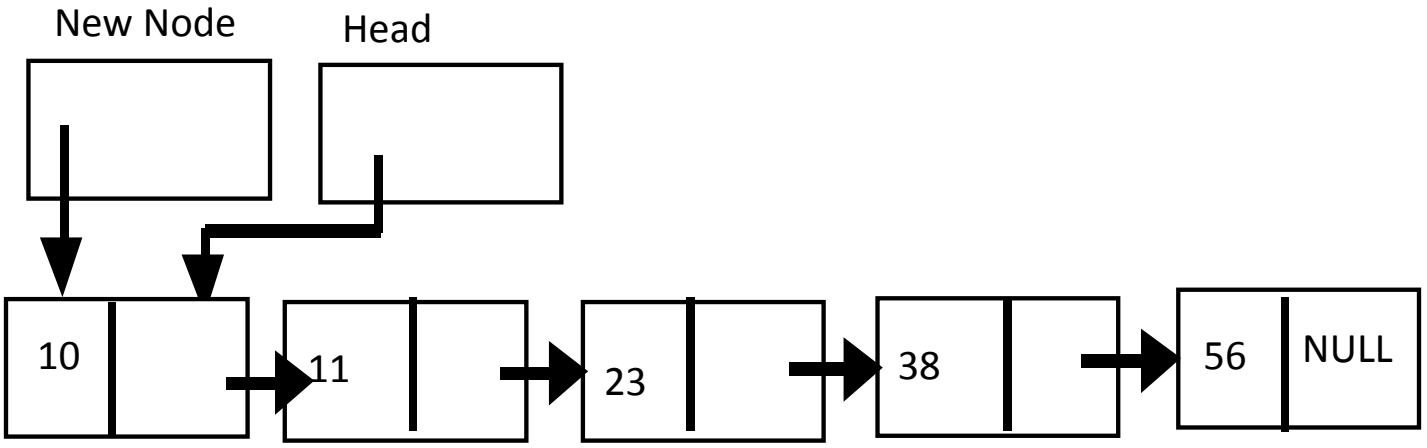
(a)



(b)


New Node
Head
10
11
23
38
56
NULL

(c)


New Node
Head
10
11
23
38
56
NULL

(d)

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;
}
```

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.

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, if 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.

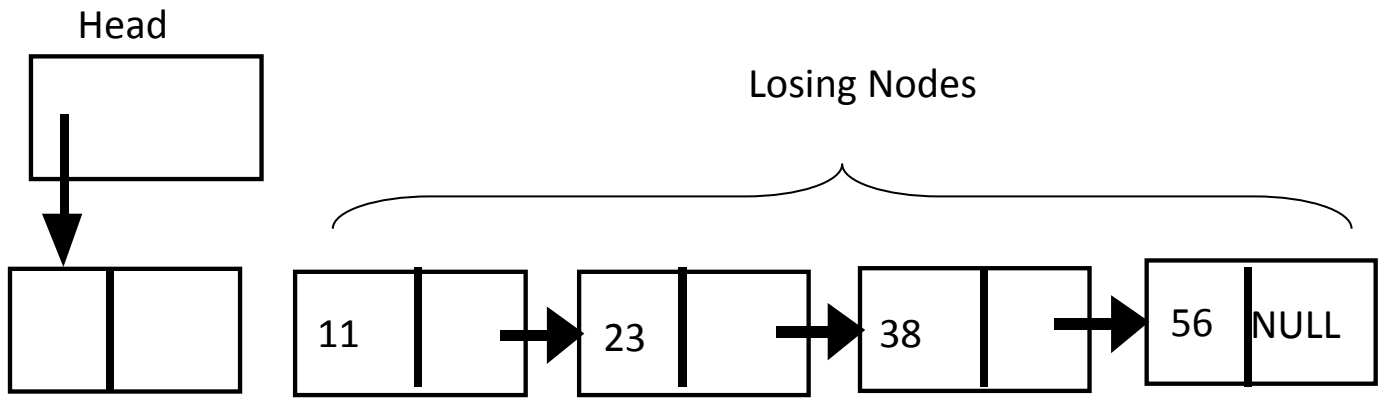
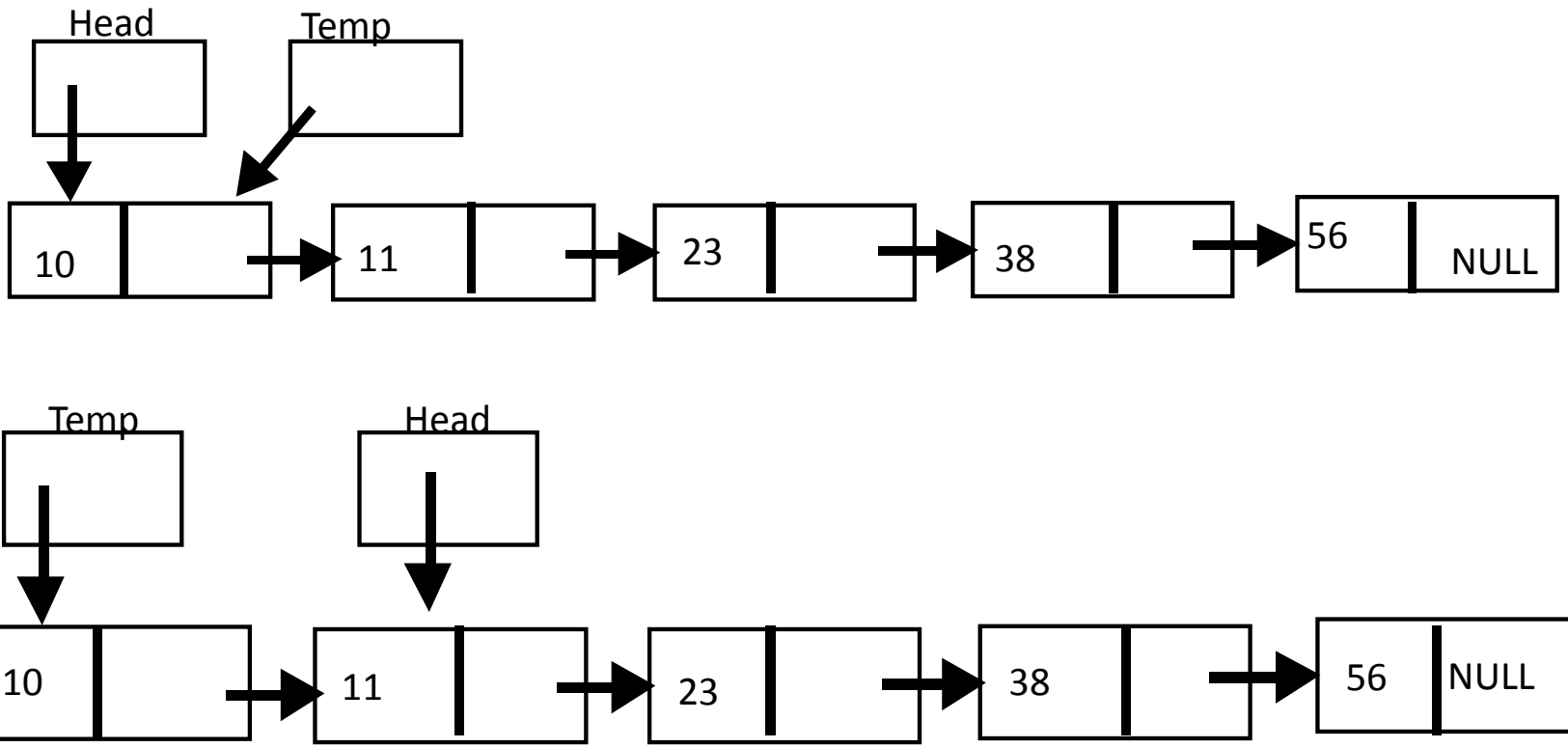


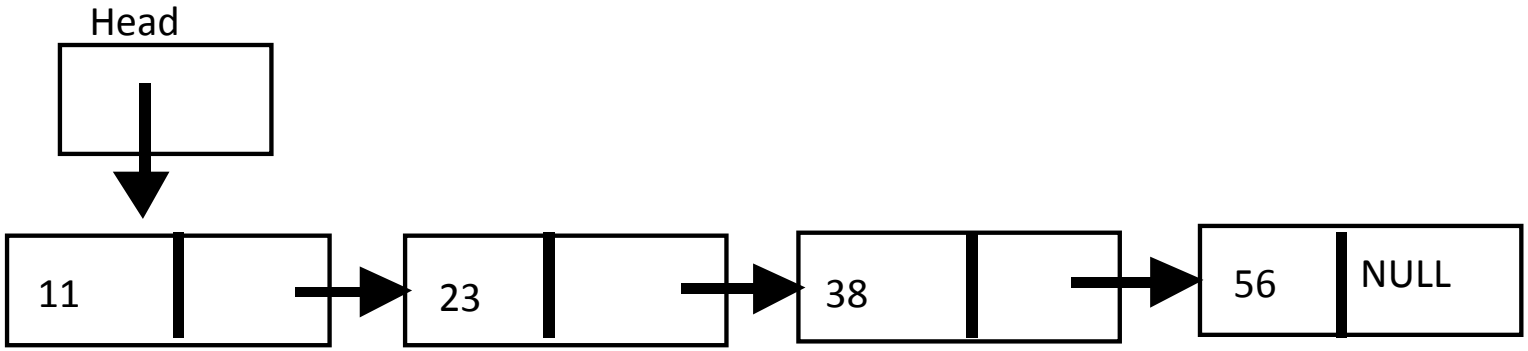
Figure 3.3

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.

2. Deleting Nodes at front of the list

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





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

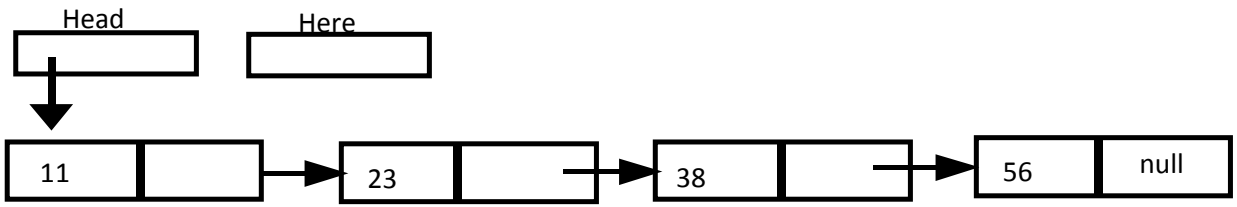
```
void DeleteFront(NodePointer &Head)
{
  NodePointer Temp;
  if (Head == NULL)
        printf("Linked list is empty..Cannot delete!");
  else
        {
              Temp = Head;
              Head = Head->Link;
              //Move Head to the next node
              free(Temp);
        }
}
```

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.

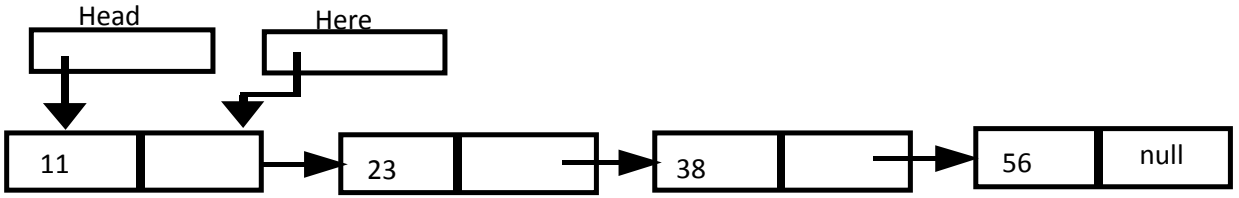
3. Searching a List

- ***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:-

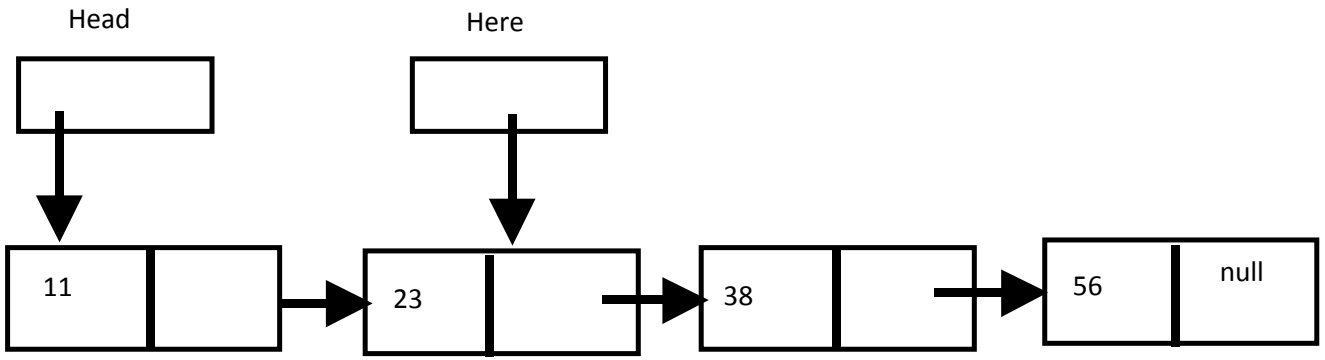
(a) Key = 38



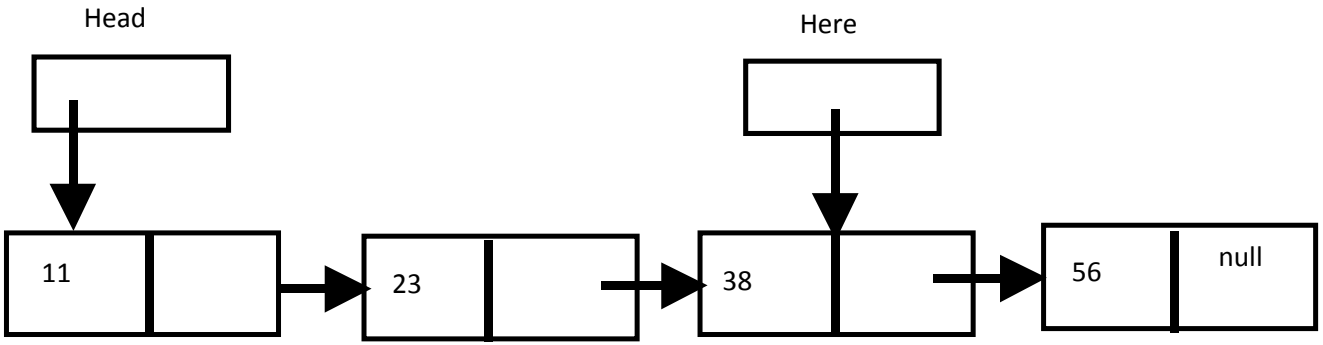
(b) Key = 38



(c) Key = 38



(d) Key = 38



ALGORITHMS

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.

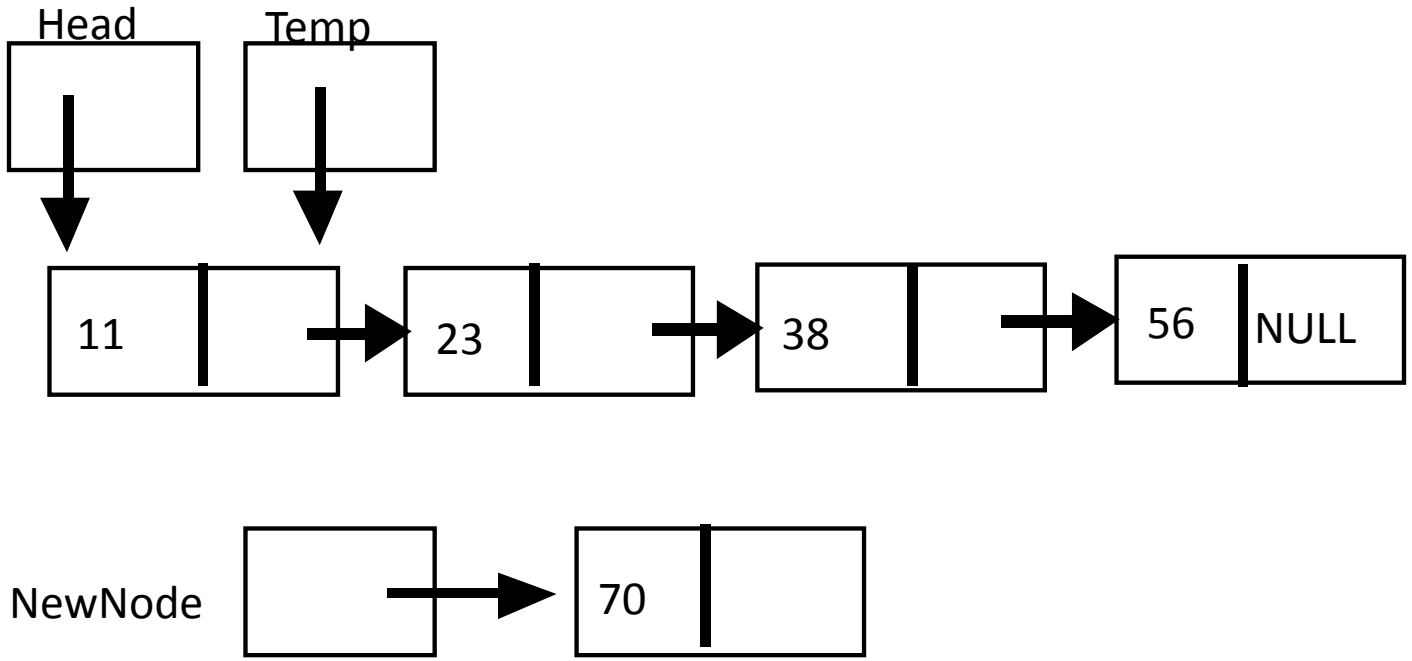
IMPLEMENTATION

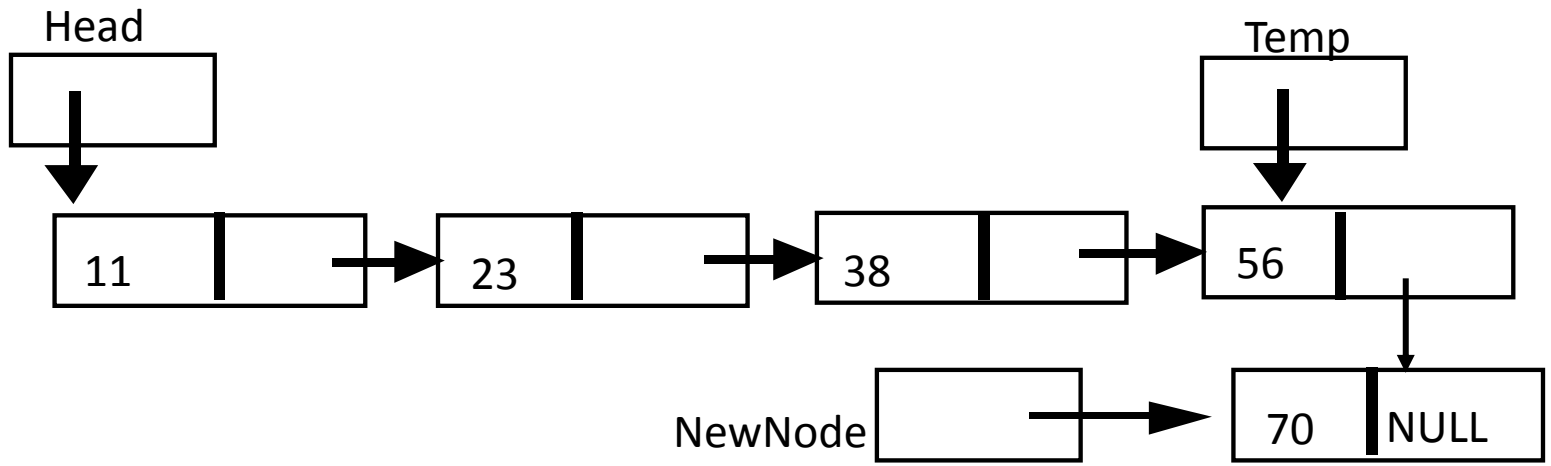
```
NodePointer Searching(NodePointer Head,int Key)
{
        NodePointer Here;
        Here = Head;
        while (Here != NULL)
                if (Here->Data == Key)
                        //Key is found in the list
                        return (Here);
                else
                    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

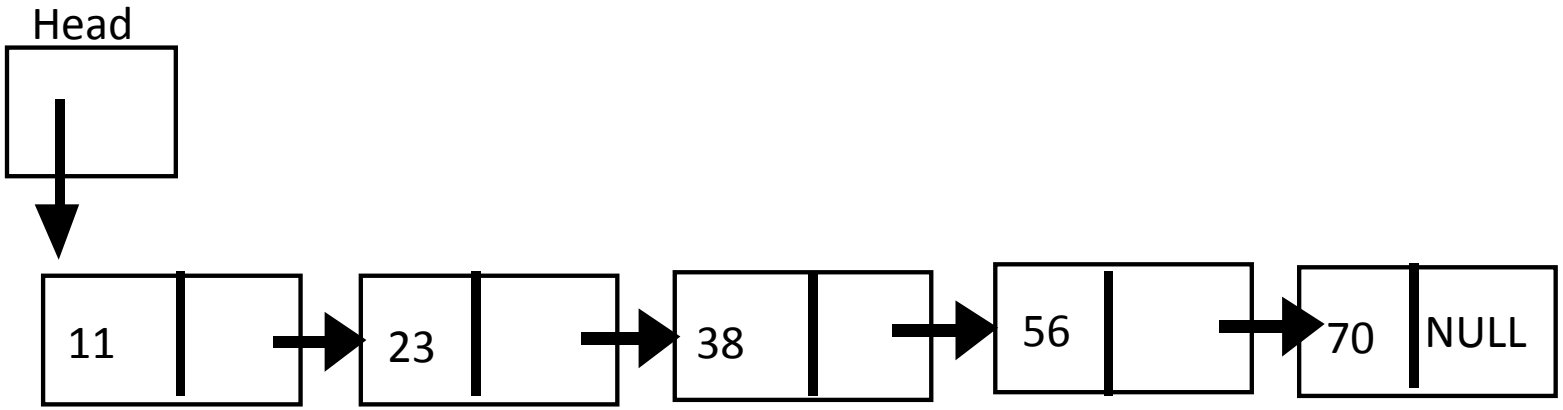
- The function above will return a pointer out. The pointer return will either point to the node which is found or NULL(means not found). Key is the search argument and is used to compare with all the nodes in the list.

Inserting a Node at the End of the list.

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




Head
Temp
11
23
38
56
NewNode
70
NULL


Head
11
23
38
56
70
NULL

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 list, make sure that the new node's link set to NULL to indicate the end of the list.

The steps are shown below:-

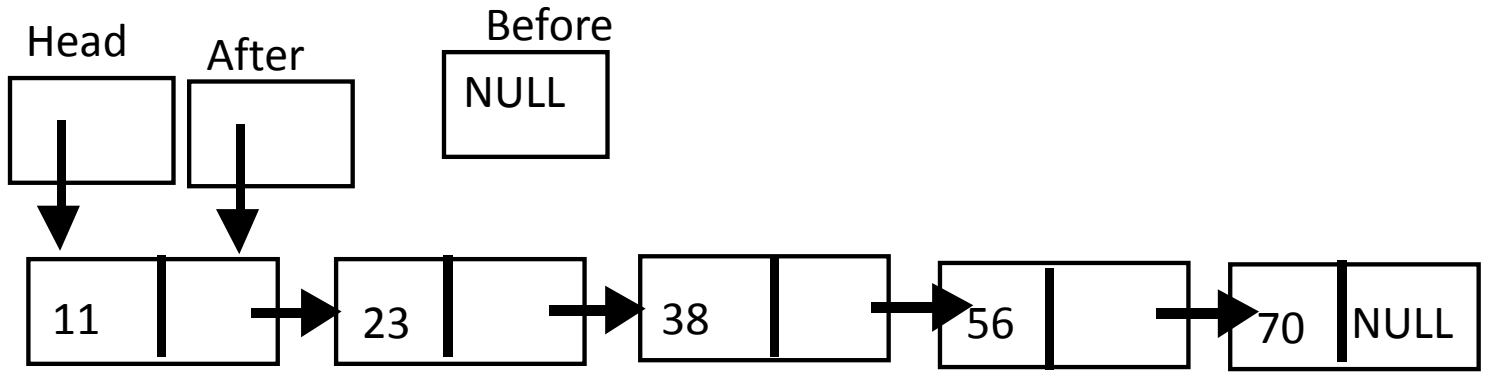
A. Create a new node pointed by *NewNode*.
B. Assign *Temp* to Head
C. Move *Temp* to the last node
D. Assign *Temp*'s link to NewNode

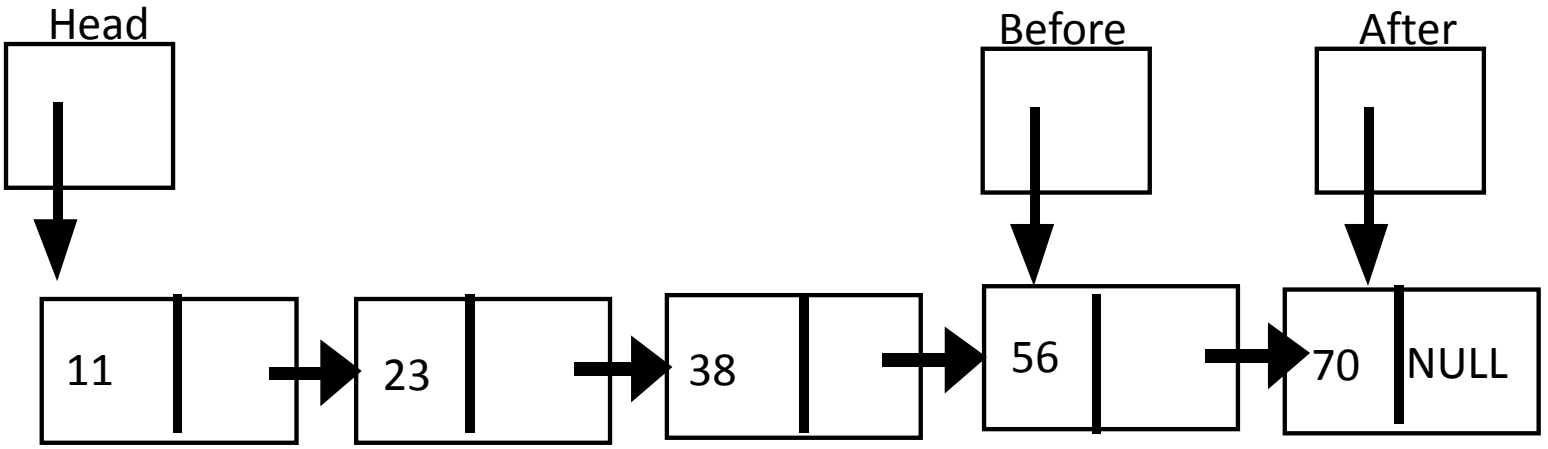
IMPLEMENTATION

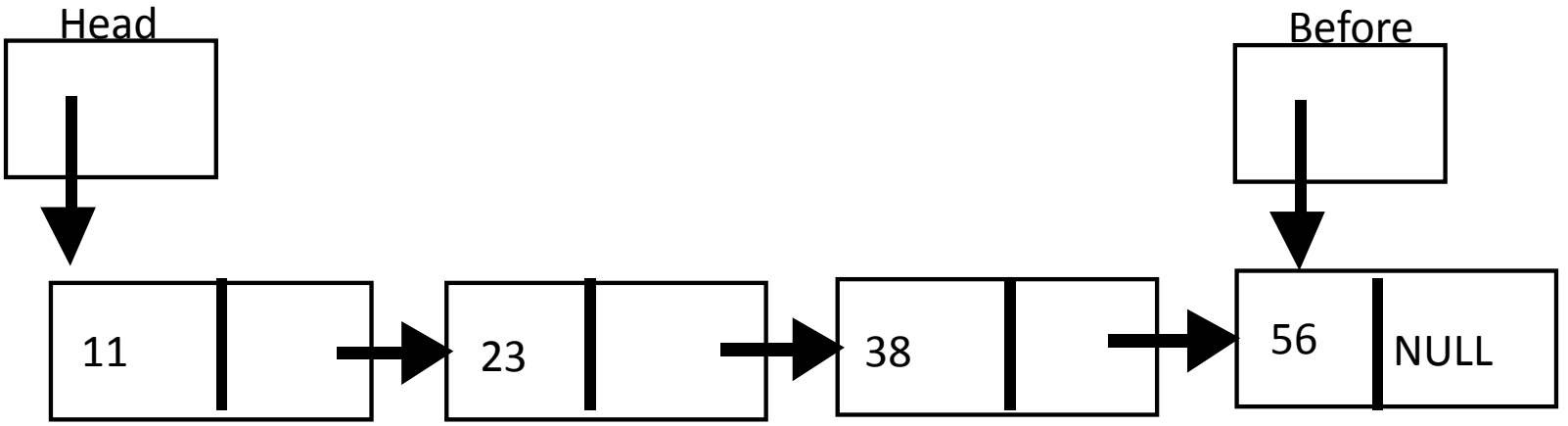
```
void InsertAtEnd(NodePointer &Head, int Number)
{
        NodePointer NewNode,Temp;
        //Creating a new memory location to be inserted
        NewNode = (NodePointer)malloc(sizeof(struct Node));
        NewNode->Data = Number;
        NewNode->Link = NULL;
        //if list is empty, just assign 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;
                }
}
```

Delete a Node at the End of the List

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




Head
Before
After
11
23
38
56
70
NULL


Head
Before
11
23
38
56
NULL

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);
  }
}
```

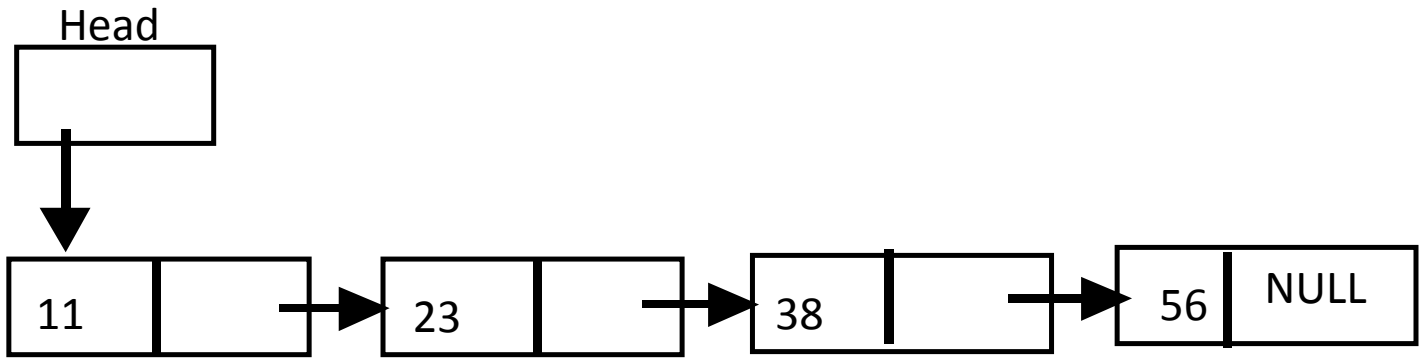
The above function works only is the list has more than 1 node. If the list has only 1 node, then the statement *Before->Link* will give error because at that time, Before is NULL. So we need to change the above function to:-

```
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);
  }
}
```

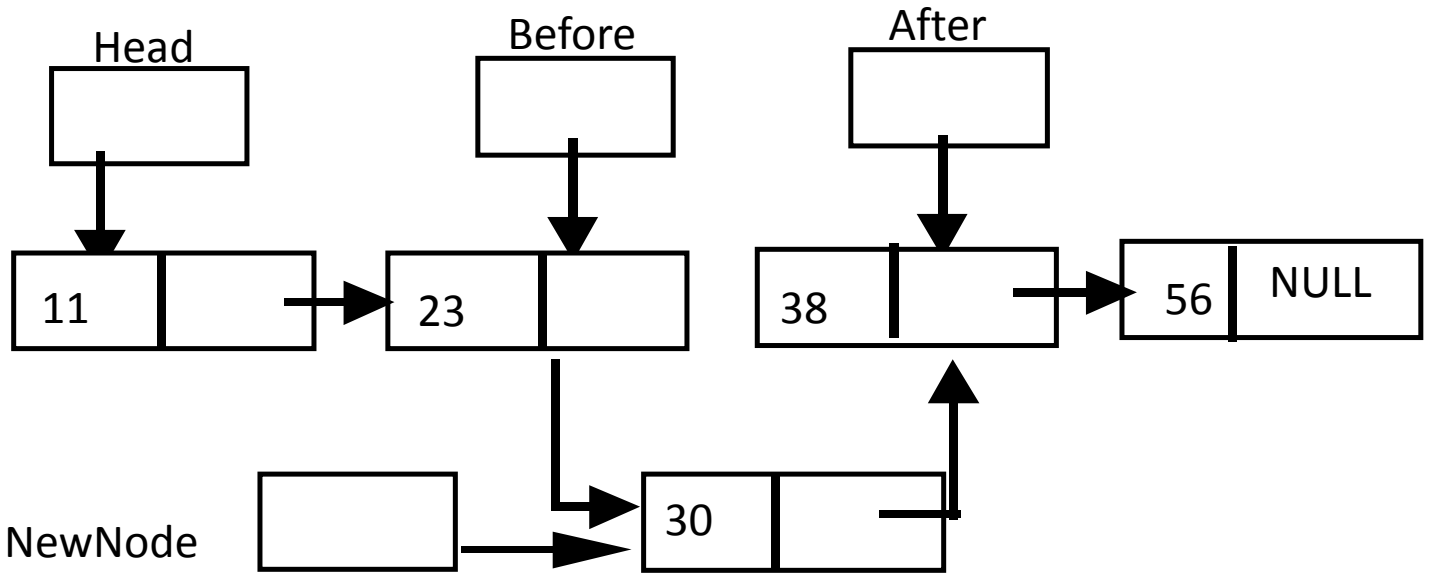
Inserting Nodes Into A Sorted Linked List

- 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 on the next slide.

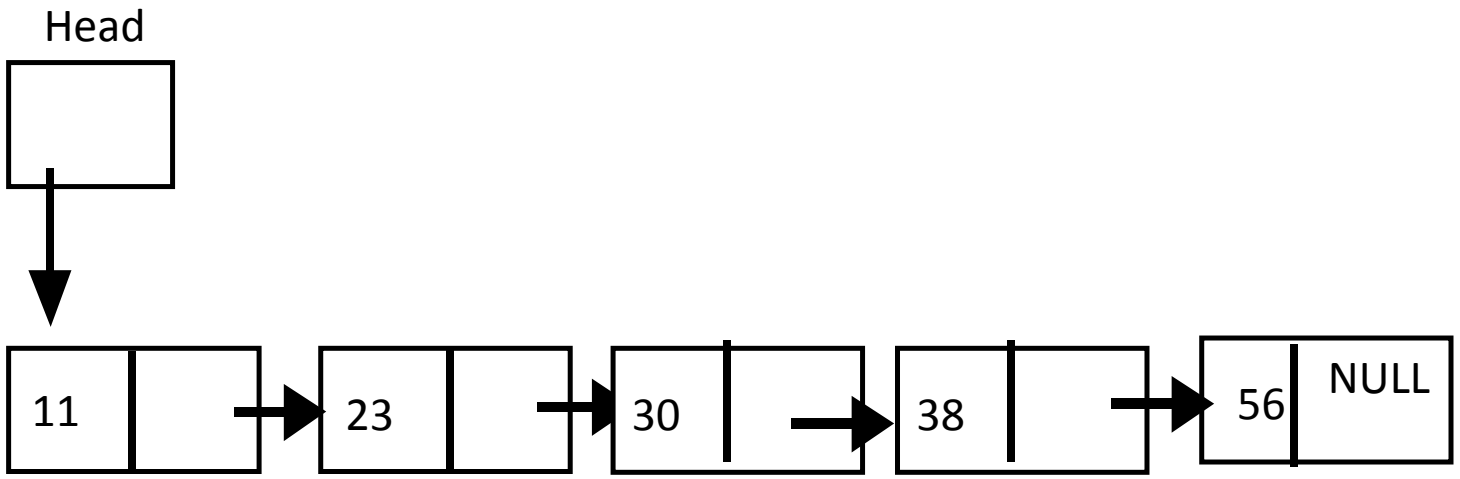
(a)


Head
11
23
38
56
NULL

(b)


Head
Before
After
11
23
38
56
NULL
NewNode
30

(c)



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;
}
```

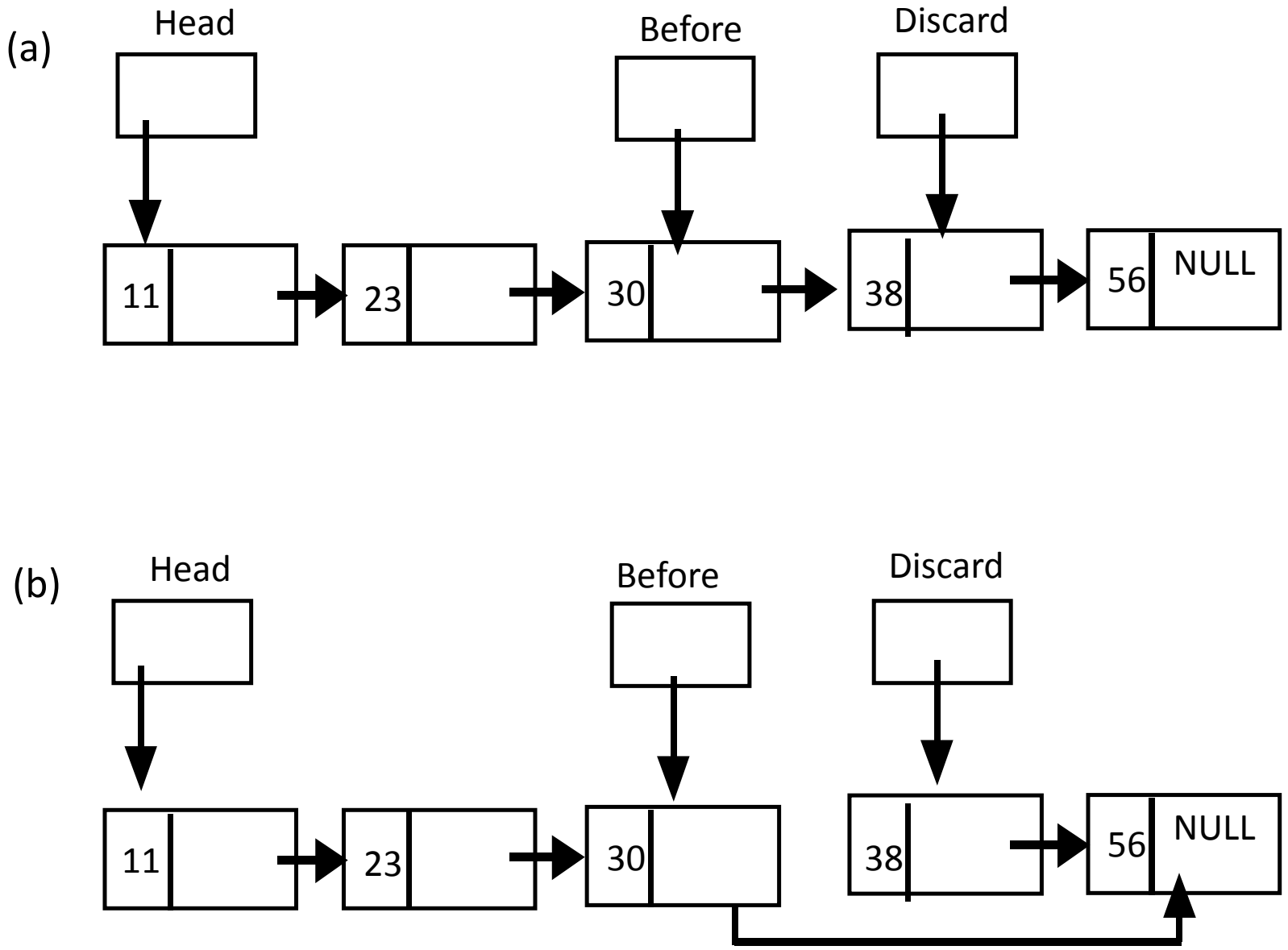
Comparison to Arrays

- 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.

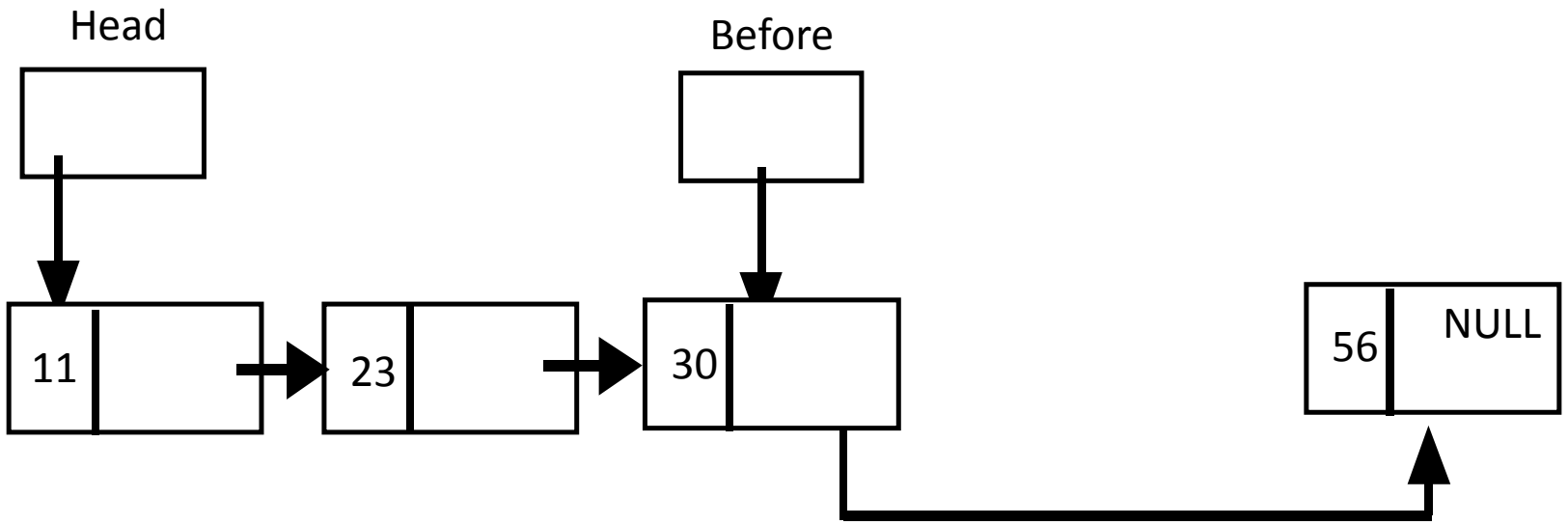
Deleting Nodes in Between 2 Nodes

- 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 :

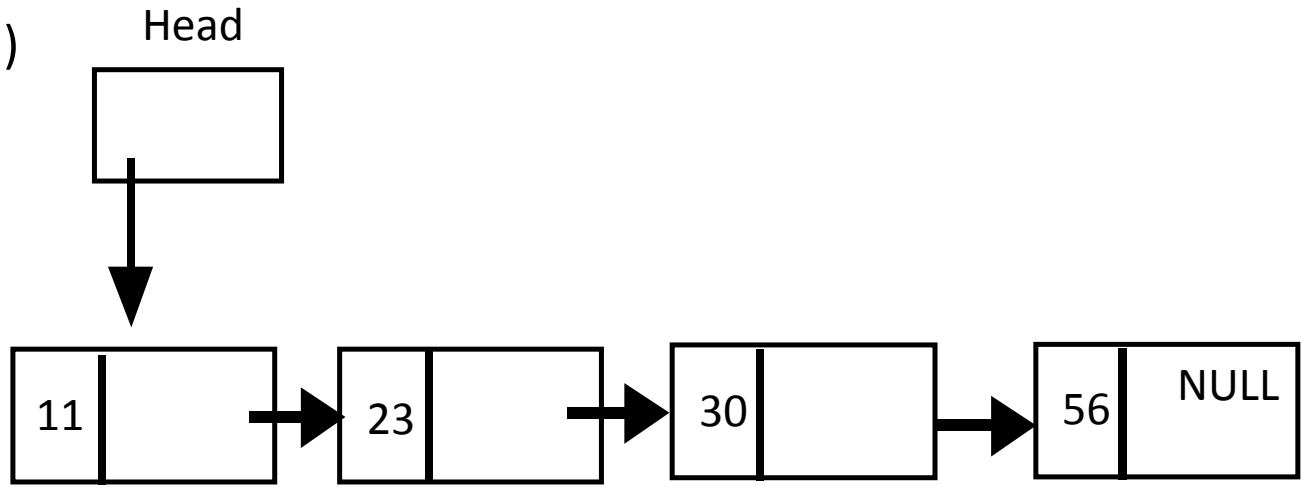
```
Before->Link = Discard->Link;
free(Discard);
```


(a)
Head
Before
Discard
11
23
30
38
56
NULL
(b)
Head
Before
Discard
11
23
30
38
56
NULL

(c)



(d)



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);
}
```

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

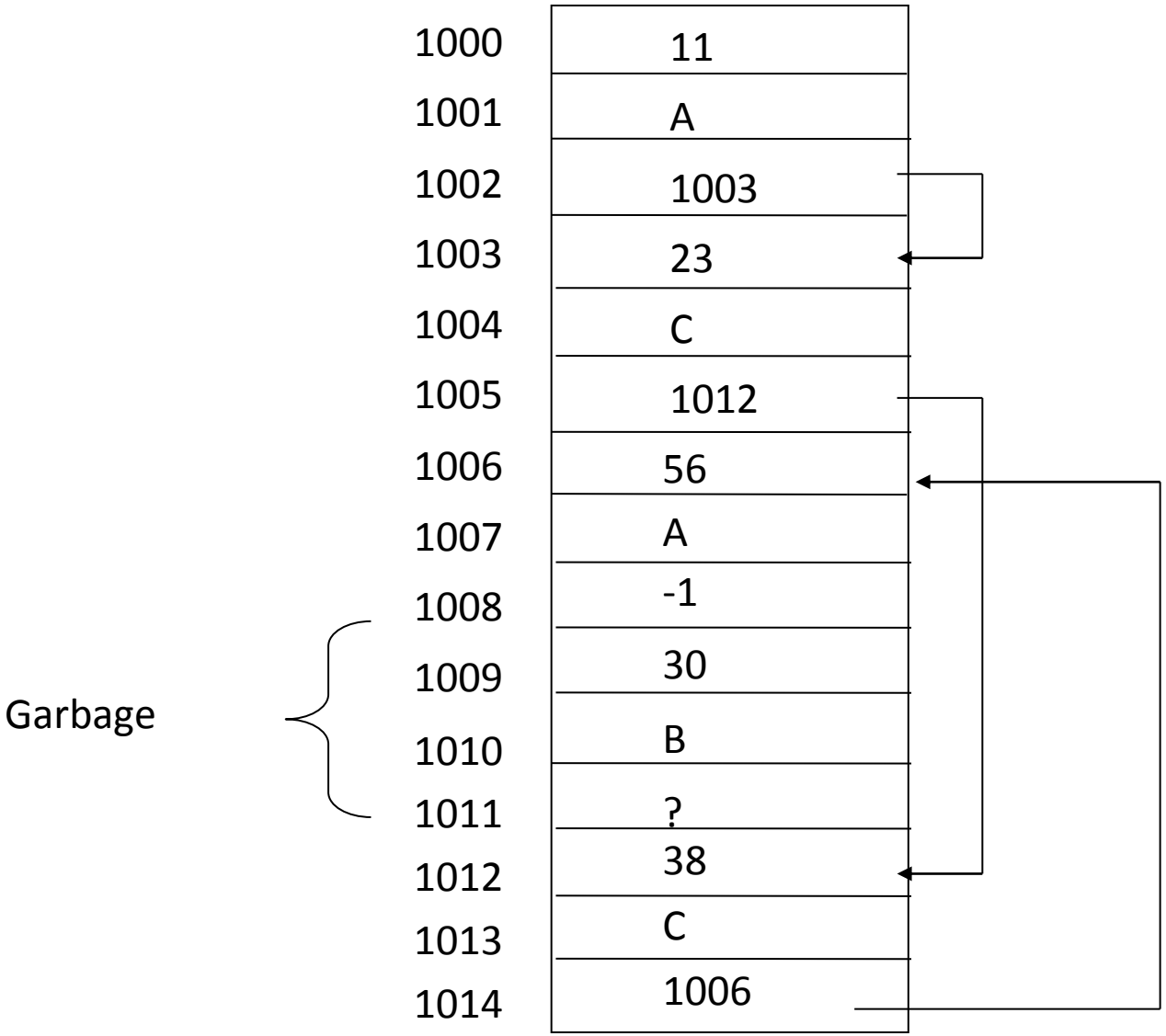

1000
11
1001
A
1002
1003
1003
23
1004
C
1005
1012
1006
56
1007
A
1008
-1
1009
30
1010
B
1011
?
1012
38
1013
C
1014
1006
Garbage

Figure 3-13

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>

struct Number
{
      int num;
      Number *next;
};

void linkedList(Number **list, int num)
{
    Number *temp;
    temp = (Number *) malloc(sizeof(Number));
    temp -> num = num;
    temp -> next = 0;
    Number *temp2 = *list;
    if( *list == 0 )
    {
       *list = temp;
    }
    else
```

```
{
        while(temp2 -> next != 0)
        {
                temp2 = temp2 -> next;
        }
        temp2 -> next = temp;
        }
}

void displayLink(Number **list)
{
    Number *temp = *list;

    while(temp != 0)
    {
            printf("\n%i",temp -> num);
            temp = temp -> next;
    }
}
```

```
main()
{
    Number *list = 0;

    linkedList(&list,90);
    linkedList(&list,100);
    linkedList(&list,100);
    linkedList(&list,80);
    displayLink(&list);
    printf("\n");
    linkedList(&list,70);
    displayLink(&list);
    getch();
}
```