Module 3 Linked List and Memory Management

Self-Referential Structures, Dynamic Memory Allocation, Singly Linked List-Operations on Linked List. Doubly Linked List, Circular Linked List, Stacks and Queues using Linked List, Polynomial representation using Linked List
Memory allocation and de-Allocation-First-fit, Best-fit and Worst-fit allocation schemes

Self-Referential Structures

A list refers to a set of items organized sequentially. – An array is an example of a list.

The array index is used for accessing and manipulating array elements.

Problems with array:

The array size has to be specified at the beginning.

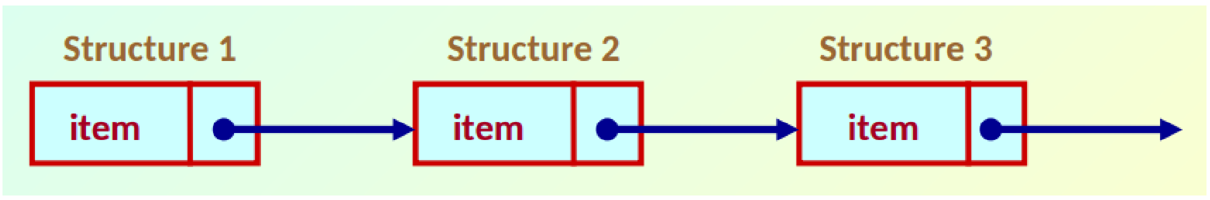
Deleting an element or inserting an element may require shifting of elements in the array.

A completely different way to represent a list:

– Make each item in the list part of a structure.

– The structure also contains a pointer or link to the structure containing the next item.

– This type of list is called a linked list.



Each structure of the list is called a node, and consists of two fields:

– One containing the data item(s).

– The other containing the address of the next item in the list (that is, a pointer).

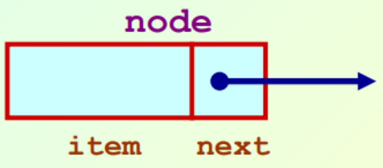
The data items comprising a linked list need not be contiguous in memory.

– They are ordered by logical links that are stored as part of the data in the structure itself.

– The link is a pointer to another structure of the same type.

Such a structure can be represented as:

```
struct node
 {
    int item;
    struct node  *next;
 }
```



Such structures that contain a member field pointing to the same structure type are called self-referential structures.

In general, a node may be represented as follows:

```
struct node_name
      {
         type  member1;
         type  member2;
          .........
         struct node_name *next;
      }
```

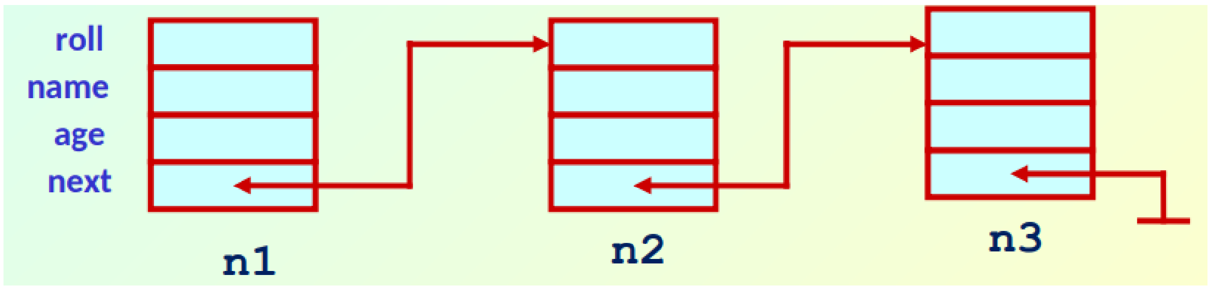
Consider the structure:

```
struct stud
        {
        int roll;
        char name[30];
        int age;
        struct stud *next;
        }
```

Also assume that the list consists of three nodes n1, n2 and n3. struct stud n1, n2, n3;
To create the links between nodes, we can write:

```
n1.next = &n2;
n2.next = &n3;
n3.next = NULL; /* No more nodes follow */
```

Now the list looks like:



Some important observations:

– The NULL pointer is used to indicate that no more nodes follow, that is, it is the end of the list.

– To use a linked list, we only need a pointer to the first element of the list.
– Following the chain of pointers, the successive elements of the list can be accessed by traversing the list.

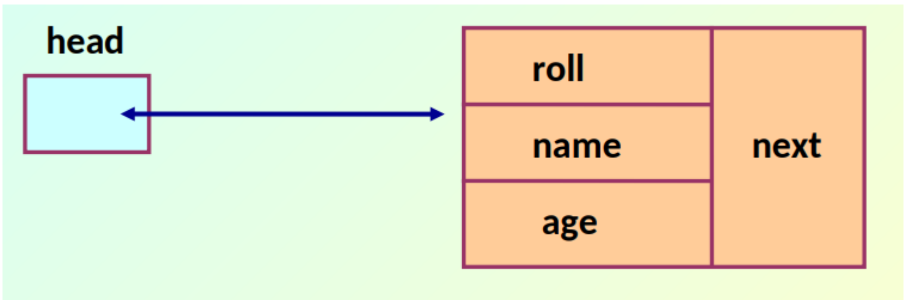
```
#include <stdio.h>
struct stud
 {
        int roll;
        char name[30];
        int age;
        struct stud *next;
 }
main()
{
        struct stud n1, n2, n3;
        struct stud *p;
        scanf (”%d %s %d”, &n1.roll, n1.name, &n1.age);
        scanf (”%d %s %d”, &n2.roll, n2.name, &n2.age);
        scanf (”%d %s %d”, &n3.roll, n3.name, &n3.age);
        n1.next = &n2;
        n2.next = &n3;
        n3.next = NULL;
         /* Now traverse the list and print the elements */
        p = &n1; /* point to 1st element */
        while (p != NULL)
        {
                printf (”\n %d %s %d”, p->roll, p->name, p->age);
                p = p->next;
        }
}
```

Dynamically allocate space for the nodes.

– Use malloc() or calloc() for allocating space for every individual nodes.

– No need for allocating additional space unnecessarily like in an array

head = (node *) malloc(sizeof(node));



```
node *create_list()
{
            int k, n;
            node *p, *head;
            printf ("\n How many elements to enter?");
            scanf ("%d", &n);
            for (k=0; k<n; k++)
            {
                        if (k == 0) {
                        head = (node *) malloc (sizeof(node));
                        p = head;
                        }
                        else {
                        p->next = (node *) malloc (sizeof(node));
                        p = p->next;
                        }

            scanf ("%d %s %d", &p->roll, p->name, &p->age);
            }
            p->next = NULL;
            return (head);
}
```

To be called from main() function as:

```
            node *head;
            ………
            head = create_list();
```

Traversing the List

Once the linked list has been constructed and head points to the first node of the list,

– Follow the pointers.
– Display the contents of the nodes as they are traversed.
– Stop when the next pointer points to NULL.

```
void display (node *head)
{
            int count = 1;
            node *p;

            p = head;
            while (p != NULL)
            {
                        printf ("\nNode %d: %d %s %d", count,
                        p->roll, p->name, p->age);
                        count++;
                        p = p->next;
            }
```

```
 printf ("\n");
}
```

To be called from main() function as:

```
node *head;
………
display (head);
```

LINKED LIST

If the memory is allocated before the execution of a program, it is fixed and cannot be changed. We have to adopt an alternative strategy to allocated memory only when it is required. There is a special data structure called linked list that provides a more flexible storage system and it does not require the use of array.

Linked lists are special list of some data elements linked to one another. The logical ordering is represented by having each element pointing to the next element. Each element is called a node, which has two parts, INFO part which stores the information and LINK which points to the next element.

Advantages

Linked list have many advantages. Some of the very important advantages are:

- *Linked Lists are dynamic data structure*: That is, they can grow or shrink during the execution of a program.
- *Efficient memory utilization:* Here, memory is not pre-allocated. Memory is allocated whenever it is required. And it is deallocated when it is no longer needed.
- *Insertion and deletions are easier and efficient:* Linked lists provide flexibility in inserting data item at a specified position and deletion of a data item from the given position.
- *Many complex applications can be easily carried out with linked lists*

Disadvantages

- ***More Memory***: If the numbers of fields are more, then more memory space is needed.
- Access to an arbitrary data item is little bit cumbersome and also time consuming.

Types of Linked List

Following are the various flavours of linked list.

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

Basic Operations

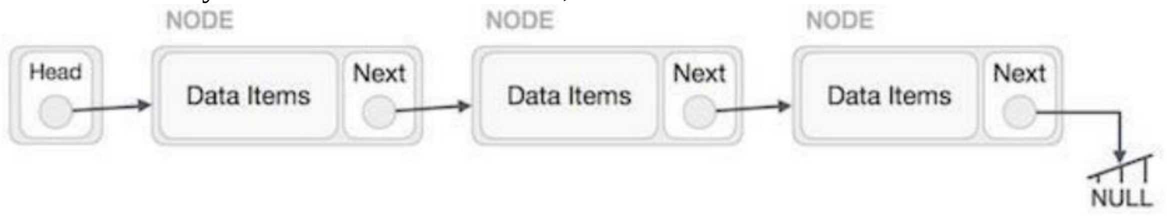
- Insertion − add an element at the beginning of the list.
- Display − displaying complete list.
- Search − search an element using given key.
- Delete - delete an element using given key

Singly Linked List

A linked list is a non-sequential collection of data items called nodes. These nodes in principles are structures containing fields. Each node in a linked list has basically two fields.

1. DATA field
2. NEXT field

The DATA field contains an actual value to be stored and processed. And, the NEXT field contains the address of the next data item in the linked list. The address used to access a particular node is known as a pointer. Therefore, the elements in a linked list are ordered not by their physical placement in memory but their logical links stored as part of the data within the node itself. Note that, the link field of the last node contains NULL rather than a valid address. It is a NULL pointer and indicates the end of the list. External pointer(HEAD) is a pointer to the very first node in the linked list, it enables us to access the entire linked list.



Basic Operations

Following are the basic operations supported by a list.

- **Insertion** –A new node may be inserted
    - ➢ At the beginning of the linked list
    - ➢ At the end of the linked list
    - ➢ At the specific position of the linked list
-

Deletion − delete an element from the
- ➢ Beginning of the linked list
- ➢ End of the linked list
- ➢ Specific position of the linked list

- **Display** – This operation is used to print each and every nodes information.
- **Traversing-**

It is a process of going through all the nodes of a linked list from one end to the other end. If we start traversing from the very first node towards the last node, it is called forward traversing. If the desired element is found, we signal operation “SUCCESSFUL”.

Steps to create a linked list

Step 1: Include Header File

Step 2: Define Node Structure

We are now defining the new global node which can be accessible through any of the function.

```
struct node
{  int data;
   struct node *next;
}*start=NULL;
```

Step 3 : Create Node using Dynamic Memory Allocation

Now we are creating one node dynamically using malloc function.

```
new_node=(struct node *)malloc(sizeof(struct node));
```

Step 4 : Fill Information in newly Created Node

Now we are accepting value from the user using scanf. Accepted Integer value is stored in the data field. Whenever we create new node , Make its Next Field as NULL.

```
printf("Enter the data : ");

scanf("%d",&new_node->data);

new_node->next=NULL;
```

Step 5 : Creating Very First Node

If node created in the above step is very first node then we need to assign it as starting node. If start is equal to null then we can identify node as first node .

```
start = NULL
if (start == NULL)
        {
        start = new_node;
        curr = new_node;
        }
```

Insert node at Start/First Position in Singly Linked List

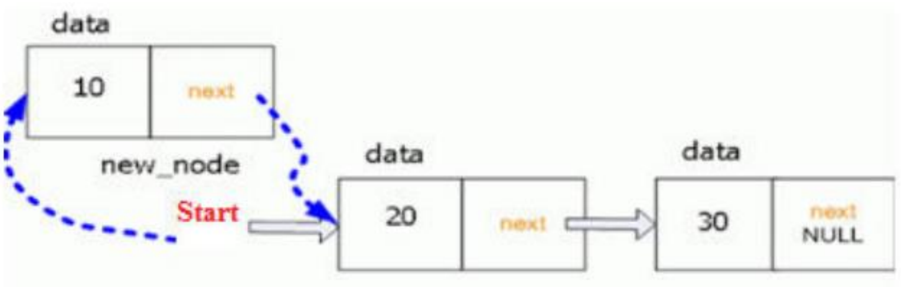
1. Allocate a memory for the new_node

2. Insert Data into the “Data Field“of new_node

3. If(start=NULL) list is empty then goto step 4 otherwise goto step 5

4. Start point to the new_node and set linked field of new_node to NULL

5. Linked field of the new_node pointed to start, then set start to new_node.



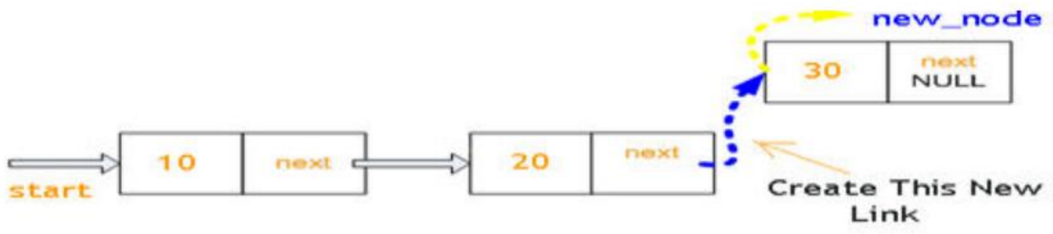
```
#include<stdio.h>
#include<conio.h>
struct node
{
int data;
struct node *ptr;
};
int main()
{
typedef struct node NODE;
NODE *start=NULL, *temp;
int item;
printf("\n\n\tSingle Linked List ");
temp=(NODE*)malloc((sizeof(NODE)));
printf("Enter the data to be inserted: ");
scanf("%d", &temp->data);
if(start==NULL)
{
temp->ptr=NULL;
start=temp;
}
else
{
temp->ptr=start;
start=temp;
}
getch();
}
```

Insert node at Last/End Position in Singly Linked List

Algorithm

1. Allocate a memory for the new_node

2. Insert Data into the “Data Field “of new_node and set “Linked field of new_node to NULL.
3. If(start=NULL) list is empty then goto step 4 otherwise goto step 5
4. Start point to the new_node and set linked field of new_node to NULL
5. Node is to be inserted at Last Position so we need to traverse SLL upto Last Node.
6. Linked field of last node pointed to the new node.



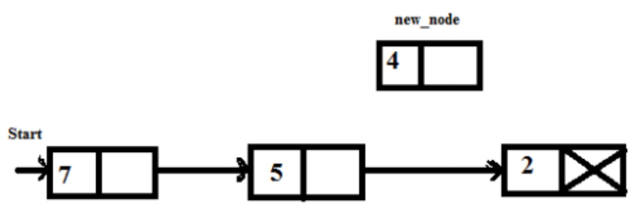
Program

```
#include<stdio.h>
#include<conio.h>
struct node
{
int data;
struct node *ptr;
};
int main()
{
typedef struct node NODE;
NODE *start=NULL, *temp, *p;
int item;
printf("\n\n\tSingle Linked List ");
printf("\n Insert into end");
temp=(NODE*)malloc((sizeof(NODE)));
printf("Enter the data to be inserted: ");
scanf("%d", &item);
item=temp->data;
if(start==NULL)
{
temp->ptr=NULL;
start=temp;
}
else
{
p=start;
while(p->ptr!=NULL)
{
p=p->ptr;
}
p->ptr=temp;
temp->ptr=NULL;
}
getch();
}
```

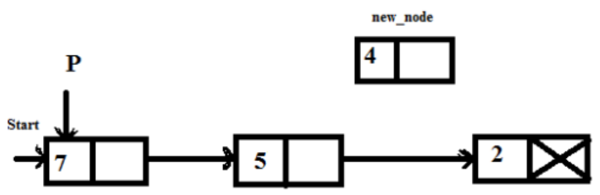
Insert node at Particular Position in Singly Linked List

Algorithm

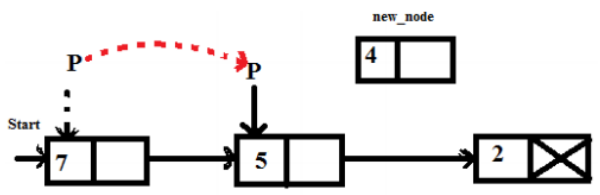
1. Allocate memory for the new node
2. Assign value to the data field of the new node
3. Find the position to insert
4. Suppose new node insert between node A and node B, make the link field of the new node point to the node B and make the link field of node A point to the new node



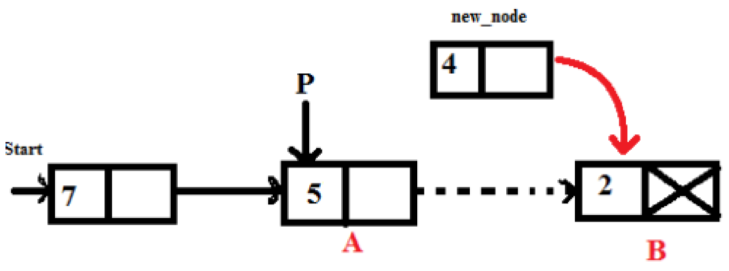
(a) Create a new node and insert the data into the data field



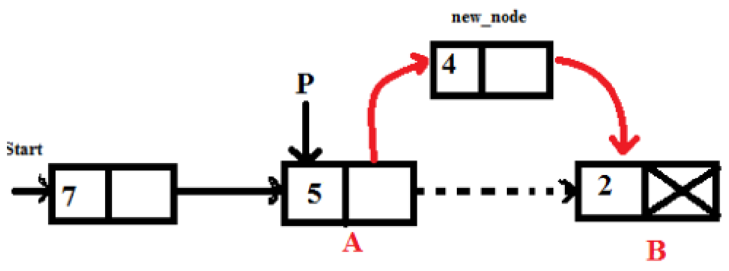
(b) p=start



(c) p=p->ptr



(d)new_node->ptr=p->ptr



Program

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
struct node
{
int data;
struct node*ptr;
};
void main()
{
typedef struct node NODE;
NODE *start=NULL,*temp,*p,*t;
int ch,item,pos,i;
printf("\nEnter the number: ");
scanf("%d",&item);
temp=(NODE*)malloc(sizeof(NODE));
temp->data=item;
printf("\nEnter the position: ");
scanf("%d",&pos);
p=start;
for(i=1;i<pos-1;i++)
{
p=p->ptr;
}
temp->ptr=p->ptr;
p->ptr=temp;
getch();
}
```

Display the elements in the linked list

1. If (start == Null) Display “Empty List”
2. Otherwise print elements from start to End

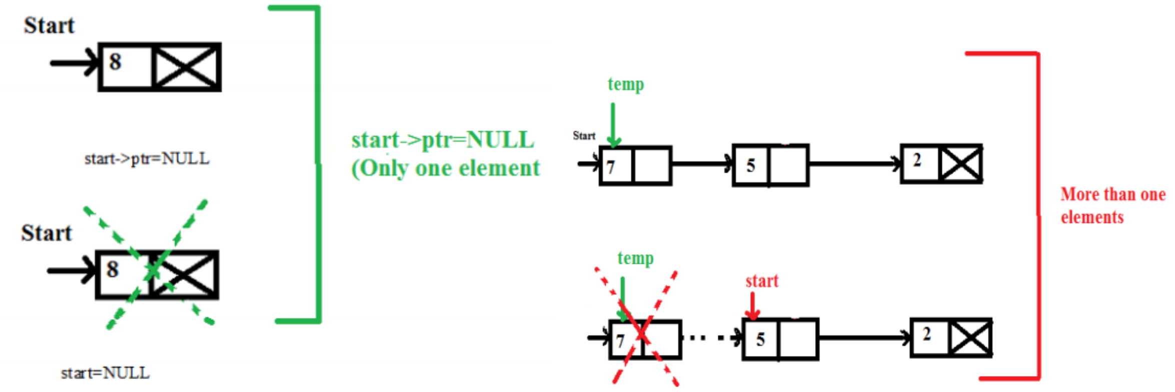
```
if(start==NULL)
printf("\nList is empty");
else
{
printf("\nElements are:");
for(p=start;p!=NULL;p=p->ptr)
printf(" %d",p->data);
 }
```

Delete node at Start/First Position in Singly Linked List

Algorithm

1. If list is empty, deletion is not possible.
2. If the list contains only one element, set external pointer to NULL

. Otherwise move the external pointer point to the second node and delete the first node



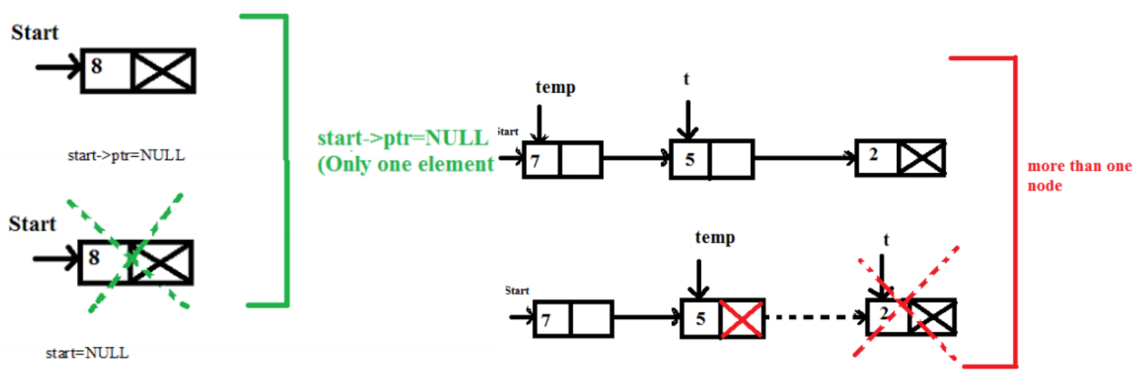
Program

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
struct node
{
int data;
struct node*ptr;
};
void main()
{
typedef struct node NODE;
NODE *start=NULL,*temp;
int item;
if(start==NULL)
printf("\nDeletion is not possible");
else if(start->ptr==NULL)
{
temp=start;
start=NULL;
printf("\nDeleted item is %d",temp->data);
free(temp);
}
else
{
temp=start;
start=start->ptr;
printf("\nDeleted item is %d",temp->data);
free(temp);
}
getch();
}
```

Delete node at End/Last Position in Singly Linked List

Algorithm

1. If the list is empty, deletion is not possible
2. If the list contain only one element, set external pointer to NULL
3. Otherwise go on traversing the second last node and set the link field point to NULL

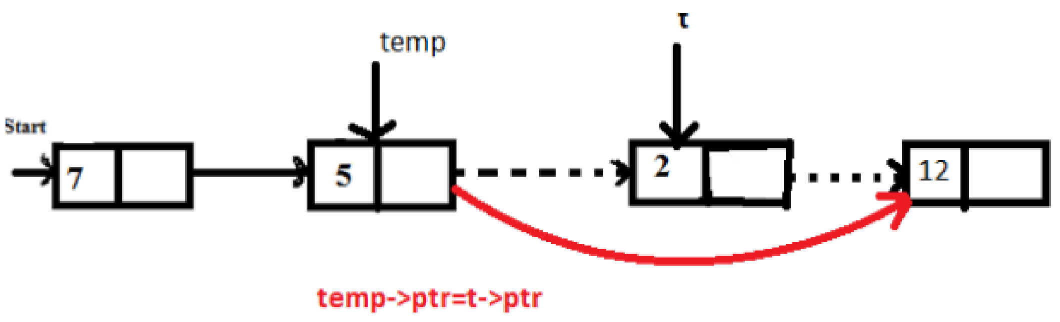


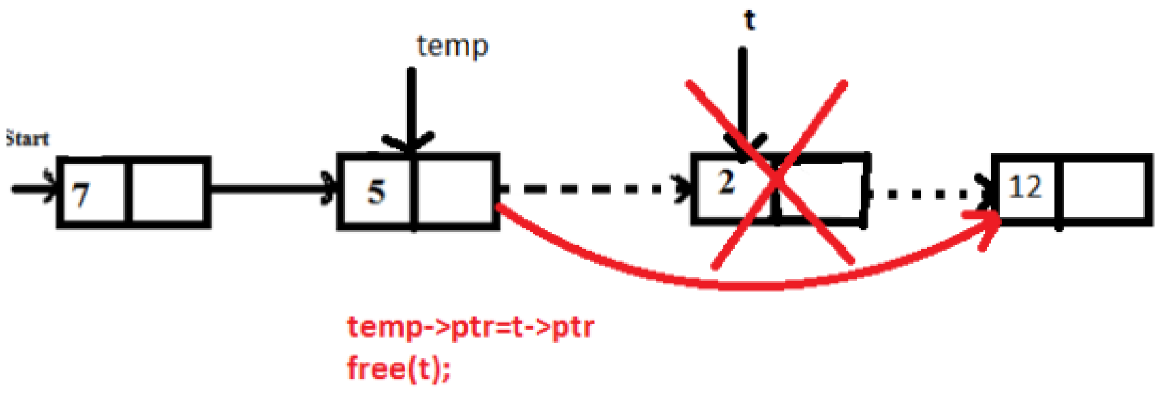
Program

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
struct node
{
{
typedef struct node NODE;
NODE *start=NULL,*temp,*p,*t;
int ch,item,pos,i;
if(start==NULL)
printf
("
\nDeletion is not possible");
else if(start->ptr==NULL)
{
temp=start;
start=NULL;
printf("\nDeleted item is %d",temp->data);
free(temp);
getch();
}
int data;
struct node*ptr;
};
void main()

}
else
{
temp=start;
t=start->ptr;
while(t->ptr!=NULL)
{
t=t->ptr;
temp=temp->ptr;
}
temp->ptr=NULL;
printf("\nDeleted item is %d",t->data);
free(t);
}
```

Delete node at Particular Position in Singly Linked List
1. Find the position to delete.
2. Suppose we delete node between node A & node B, set the link field of node A point to the node B



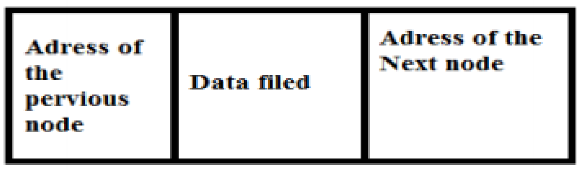


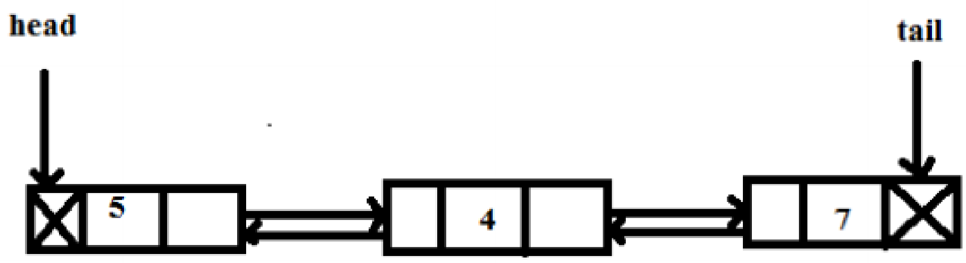
Program

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
struct node
{
int data;
struct node*ptr;
};
void main()
{
typedef struct node NODE;
NODE *start=NULL,*temp,*p,*t;
int ch,item,pos,i;
printf("\nEnter the position: ");
scanf("%d",&pos);
temp=start;
if(pos==1)
{
temp=start;
start=start->ptr;
printf("\nDeleted item is %d",temp->data);
free(temp);
}
else
{
for(i=1;i<pos-1;i++)
temp=temp->ptr;
t=temp->ptr;
temp->ptr=t->ptr;
printf("\nDeleted item is %d",t->data);
free(t);
}
getch()
}
```

DOUBLY LINKED LIST

A more sophisticated kind of linked list is a doubly-linked list or a two-way linked list. In a doubly linked list, each node has two links: one pointing to the previous node and one pointing to the next node.





head is the external pointer, used to point starting of the doubly linked list. Tail is used to denote end of the doubly linked list.

Representation of doubly linked list

```
struct node
{
int data;
struct node *prev,*next;
};
```

Insert node at Start/First Position in Doubly Linked List

1.Allocate memory for new node

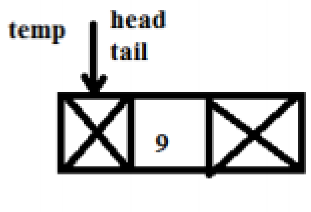
2. Assign value to the data field of the new node

3. If head=NULL then, Set head and tail pointer point to the new node, set previous node is NULL and next node is NULL

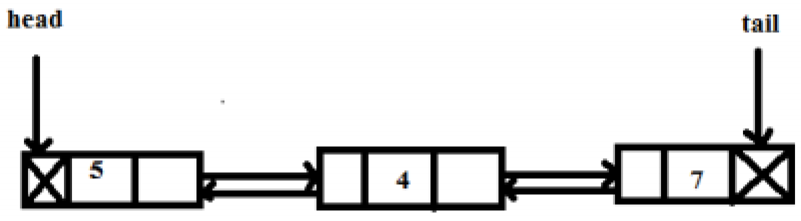
4. Otherwise

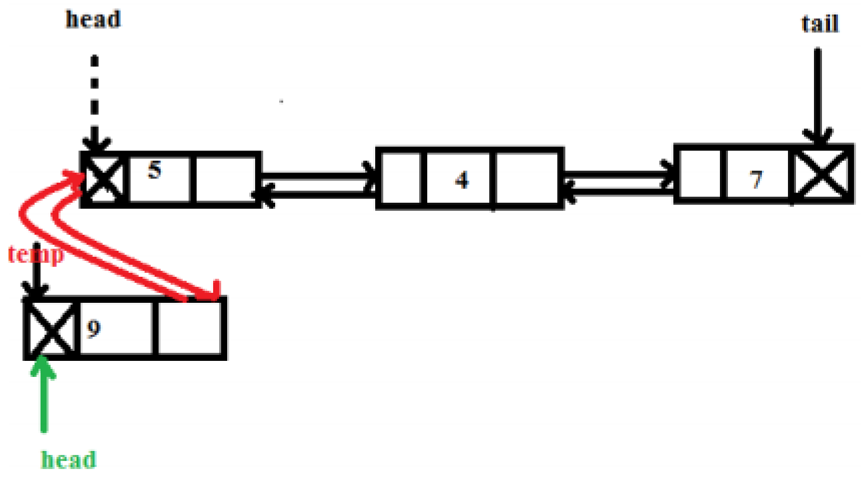
    Set temp→next=head and head→prev=temp and temp→prev=NULL

    Set external pointer head point to the new node



head=NULL

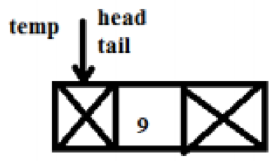




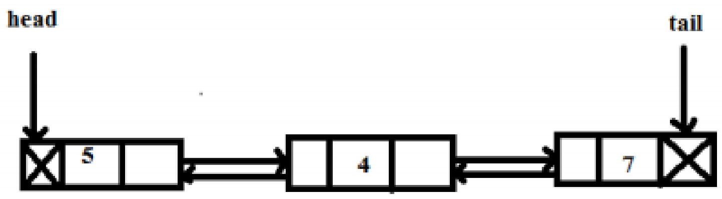
```
#include<stdio.h>
#include<conio.h>
#include<process.h>
struct node
{
int data;
        struct node *prev,*next;
};
void main()
{
typedef struct node NODE;
NODE *head=NULL,tail=NULL,*temp;
int no;
te
mp=(NODE*)malloc(sizeof(NODE));
printf("Enter the no: ");
scanf("%d",&no);
temp->data=no;
if(start==NULL)
{
temp->prev=NULL;
temp->next=NULL;
head=tail=temp;
}
else
{
temp->next=head;
head->prev=temp;
temp->prev=NULL;
head=temp;
}
if(head==NULL)
{
printf("No elements");
}
else
{
printf("\nElements are:");
for(p= head;p!=NULL;p=p->next)
{
printf(" %d",p->data);
}
}
getch();
}
```

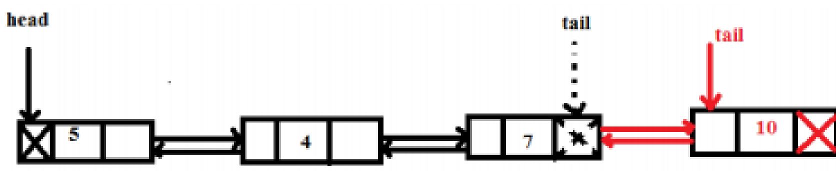
Insert node at End/Last Position in Doubly Linked List

1. Allocate memory for new node
2. Assign value to the new node
3. If head=NULL then,Set external pointer to the new node, set previous node is NULL and next node is NULL
4. Otherwise, tail->next=temp, temp->prev=tail and next pointer of new node set to NULL. tail point to the new node



head=NULL

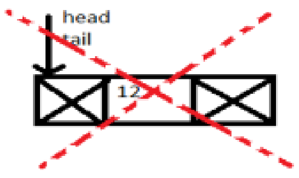




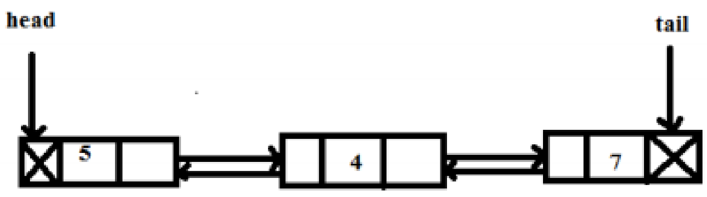
```
#include<stdio.h>
#include<conio.h>
#include<process.h>
struct node
{
int data;
struct node *prev,*next;
};
void main()
{
int ch,no;
typedef struct node NODE;
NODE *head=NULL,*tail=NULL,*temp;
temp=(NODE*)malloc(sizeof(NODE));
printf("Enter the no: ");
scanf("%d",&no);
temp->data=no;
if(start==NULL)
{
temp->prev=NULL;
temp->next=NULL;
head=tail=temp;
}
else
{
tail->next=temp;
temp->prev=tail;
temp->next=NULL;
tail=temp;
}
if(head==NULL)
{
printf("No elements");
}
else
{
printf("\nElements are:");
for(p= head;p!=NULL;p=p->next)
{
printf(" %d",p->data);
}
}
getch();
}
```

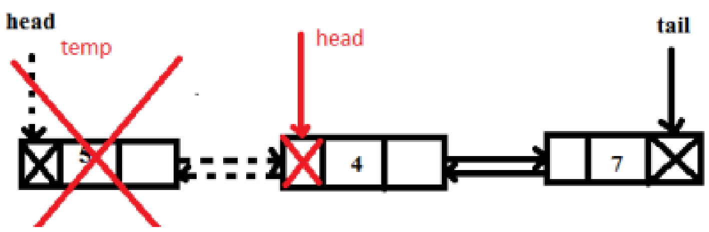
Delete a node at Start/First Position in Doubly Linked List

1. If start=NULL then, Print deletion is not possible
2. If the list contain only one element, set external pointer to NULL
3. Otherwise move the external pointer point to the second node and delete first node



head=tail=NULL

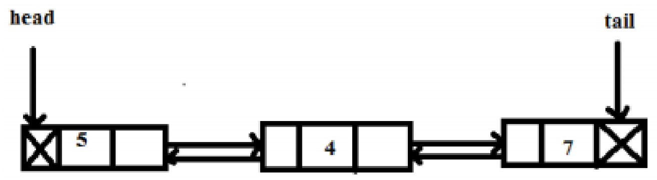


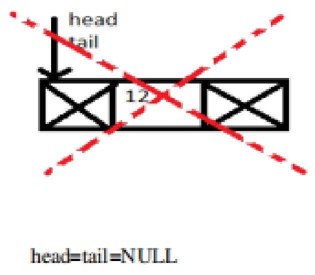


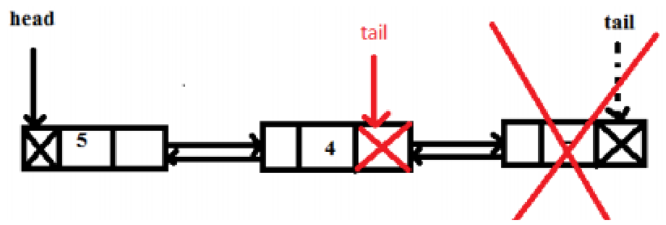
```
#include<stdio.h>
#include<conio.h>
struct node
{
int data;
struct node *prev,*next;
};
void main()
{
int no;
typedef struct node NODE;
NODE *head=NULL,*temp;
if(head==NULL)
{
printf("Deletion is not possible");
}
else if(head->next==NULL)
{
temp=start;
head=tail=NULL;
printf("Deleted element is: %d",temp-
>data);
free(temp);
}
else
{
temp=head;
head=temp->next;
head->prev=NULL;
printf("Deleted element is: %d",temp-
>data);
free(temp);
}
if(head==NULL)
{
printf("No elements");
}
else
{
printf("\nElements are:");
for(p=head;p!=NULL;p=p->next)
{
printf(" %d",p->data);
}
}
getch();
}
```

Delete a node at End/Last Position in Doubly Linked List

1. If the list is empty, deletion is not possible
2. If the list contain only one element, set external pointer to NULL
3. Otherwise go on traversing the last and set next field of second last node to NULL



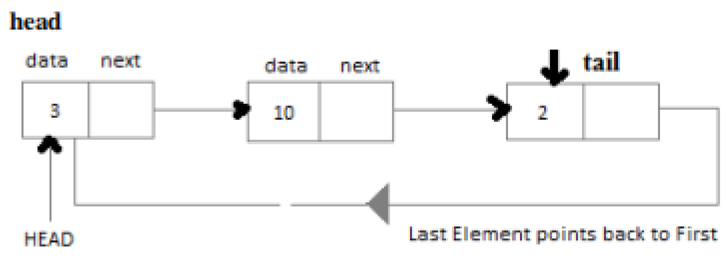




```
#include<stdio.h>
#include<conio.h>
struct node
{
int data;
struct node *prev,*next;
};
void main()
{
int no;
typedef struct node NODE;
NODE *head=NULL,tail=NULL,*temp;
if(head==NULL)
{
printf("Deletion is not possible");
}
else if(head->next==NULL)
{
temp=start;
head=tail=NULL;
printf("Deleted element is: %d",temp-
>data);
free(temp);
}
else
{
temp=tail;
tail=tail->prev;
free(temp);
}
if(head==NULL)
{
printf("No elements");
}
else
{
printf("\nElements are:");
for(p=head;p!=NULL;p=p->next)
{
printf(" %d",p->data);
}
}
getch();
}
```

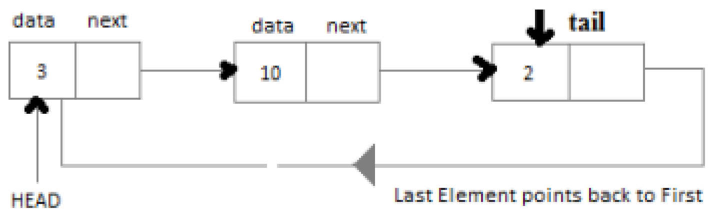
CIRCULAR LINKED LIST

Circular Linked List is little more complicated linked data structure. In the circular linked list we can insert elements anywhere in the list where as in the array we cannot insert element anywhere in the list because it is in the contiguous memory. In the circular linked list the previous element stores the address of the next element and the last element stores the address of the starting element. The elements points to each other in a circular way which forms a circular chain. The circular linked list has a dynamic size which means the memory can be allocated when it is required. A circular linked list has no end.

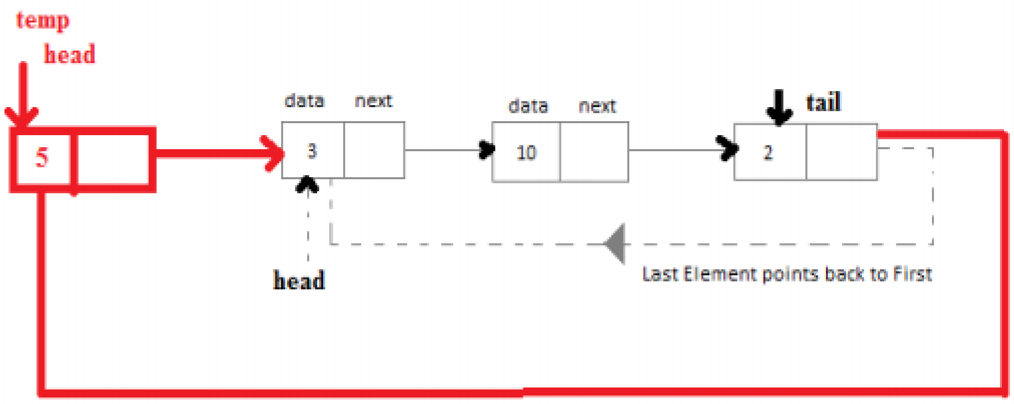


Inserting a node at the beginning

1. Allocate a memory for new node
2. If list is empty then head and tail point to the new node. And linked field of tail pointed to head.
3. If the list is not empty then
    a) New node pointed to head
    b) Head point to new node
    c) Linked field of tail pointed to head



After insertion



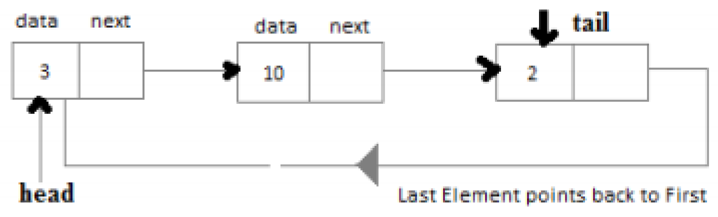
We declare the structure for the circular linked list in the same way as we declare it for the linear linked lists

```
struct node
{
int data;
struct node *next;
};
typedef struct node NODE;
NODE *head=NULL;
NODE *tail=NULL;
NODE *temp;
temp=(struct
NODE*)malloc(sizeof(NODE));
printf("Enter the element to be inserted")
scanf("%d",&item);
```

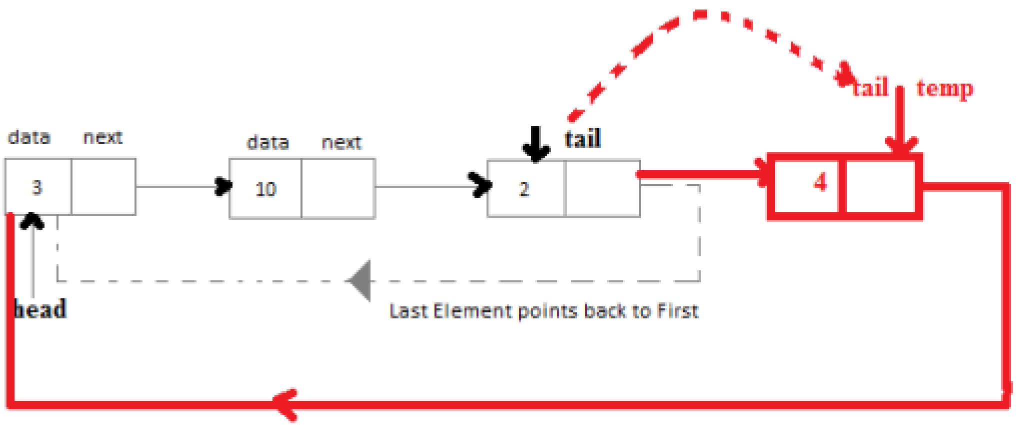
```
temp->data=item;
if(head==NULL)
{
head=tail=temp;
tail->next=head;
}
else
{
temp->next=head;
head=temp;
tail->next=head;
}
```

Inserting a node at the End

1. Allocate a memory for new node
2. If list is empty then head and tail point to the new node. And linked field of tail pointed to head.
3. If the list is not empty then
    a) Linked field of tail point to the new node
    b) tail point to new node



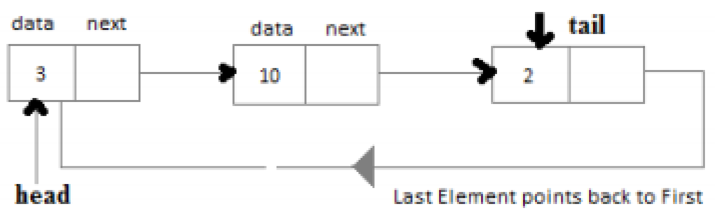
After insertion

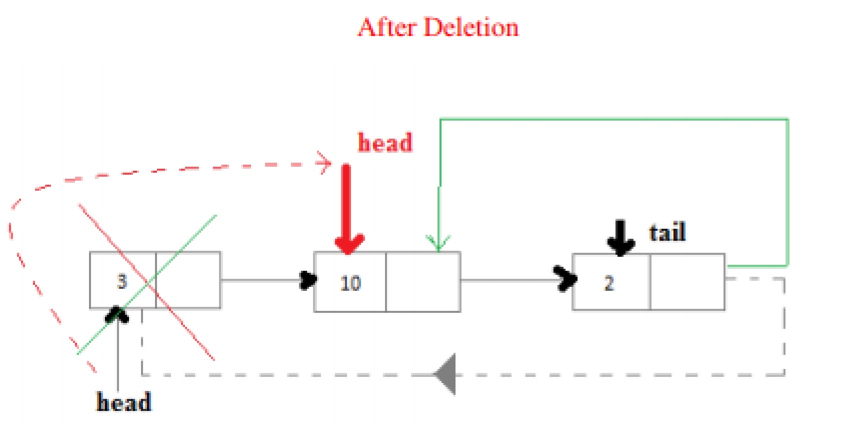


```
struct node
{
int data;
struct node *next;
};
typedef struct node NODE;
NODE *head=NULL;
NODE *tail=NULL;
NODE *temp;
temp=(struct
NODE*)malloc(sizeof(NODE));
printf("Enter the element
to be inserted")
scanf("%d",&item);
temp->data=item;
if(head==NULL)
{
head=tail=temp;
tail->next=head;
}
else
{
tail->next=temp;
tail=temp;
tail->next=head;
}
```

Delete a node from the beginning

1. If list is empty then print deletion is not possible
2. If the list contain only on element the set head and tail to NULL
3. If the list contain more than one element then
    a) temp pointer point to the head node
    b) head move to the next node
    c) Linked field of tail pointed to head

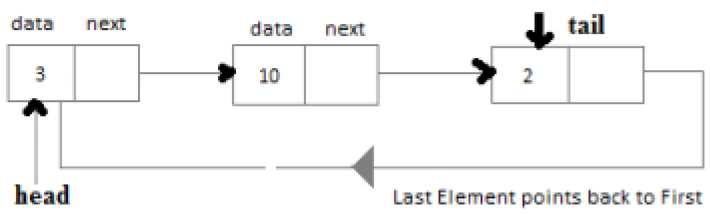


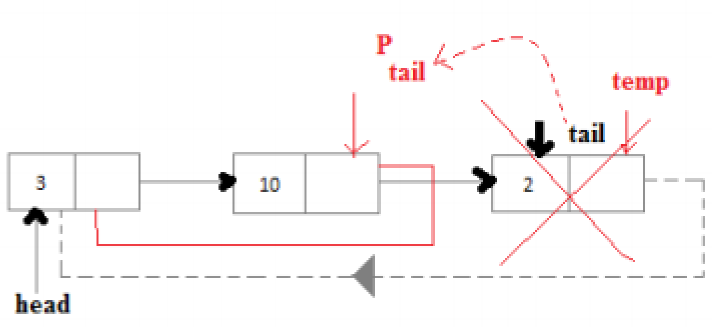


```
if(head==NULL) //List is empty
{
printf(Deletion is not possible\n”);
}
else if(head==tail) //List contains only one node
{
free(head);
head=tail=NULL;
}
else
{
temp=head;
head=head->next;
tail->next=head;
free(temp);
}
```

Delete a node from the beginning

1. If list is empty then print deletion is not possible
2. If the list contain only on element the set head and tail to NULL
3. If the list contain more than one element then
    - a) p pointer point to the second last node and temp pointer points to the last node
    - b) tail move to the p
    - c) Linked field of tail pointed to head
    - d) Delete the last node





```
if(head==NULL) //List is empty
{
printf(Deletion is not possible\n”);
}
else if(head==tail) //List contains only one node
{
free(head);
head=tail=NULL;
}
else
{
p=head;
while(p->next!=tail)
{
p-p->next;
}
temp=p->next;
tail=p;
tail->next=head;
free(temp);
}
```

Application of linked list-Polynomial Representation

Linked list are widely used to represent and manipulate polynomials. Polynomials are the expressions containing number of terms with nonzero coefficient and exponents.In the linked representation of polynomials, each term is considered as a node. And such a node contains three fields

- Coefficient field
- Exponent field
- Link field

The coefficient field holds the value of the coefficient of a term and the exponent field contains the exponent value of the term. And the link field contains the address of the next term in the polynomial. The polynomial node structure is

Coefficient(coeff)	**Exponent(expo)**	**Address of the next node(next)**

Algorithm

Two polynomials can be added. And the steps involved in adding two polynomials are given below

1.Read the number of terms in the first polynomial P

2. Read the coefficient and exponent of the first polynomial

3. Read the number of terms in the second polynomial Q

4. Read the coefficient and exponent of the second polynomial

5. Set the temporary pointers p and q to traverse the two polynomials respectively

6. Compare the exponents of two polynomials starting from the first nodes

    a) If both exponents are equal then add the coefficient and store it in the resultant linked list

    b) If the exponent of the current term in the first polynomial P is less than the exponent of the current term of the second polynomial then added the second term to the resultant linked list. And, move the pointer q to point to the next node in the second polynomial Q.

    c) If the exponent of the current term in the first polynomial P is greater than the exponent of the current term in the second polynomial Q, then the current term of the first polynomial is added to the resultant linked list. And move the pointer p to the next node.

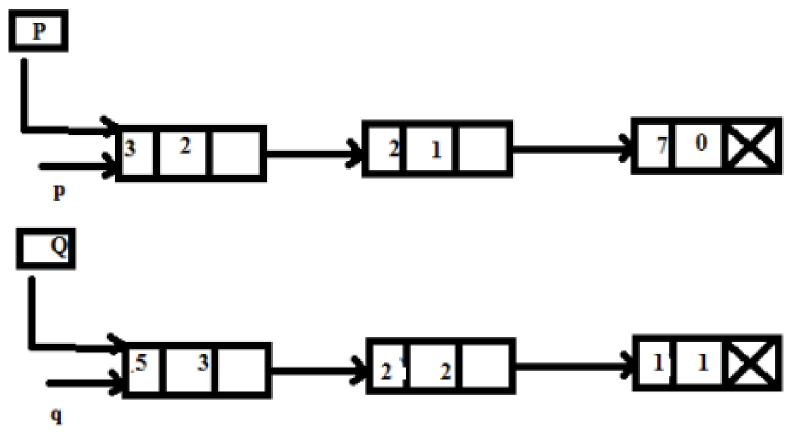
    d) Append the remaining nodes of either of the polynomials to the resultant linked list.

Let us illustrate the way the two polynomials are added. Let p and q be two polynomials having three terms each.

$P=3x^2+2x+7$

$Q=5x^3+2x^2+x$

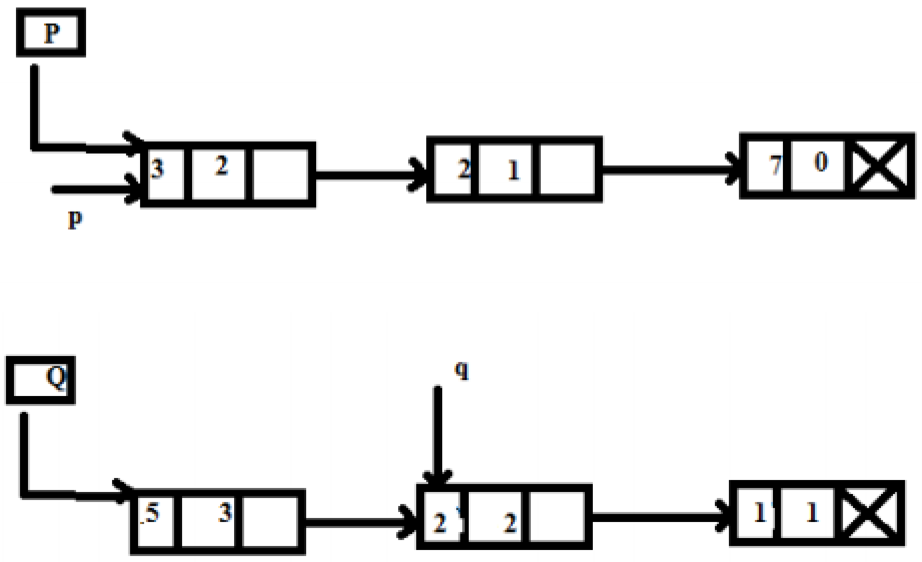
These two polynomial can be represented as



Step 1. Compare the exponent of p and the corresponding exponent of q. Here,

expo(p)<expo(q)

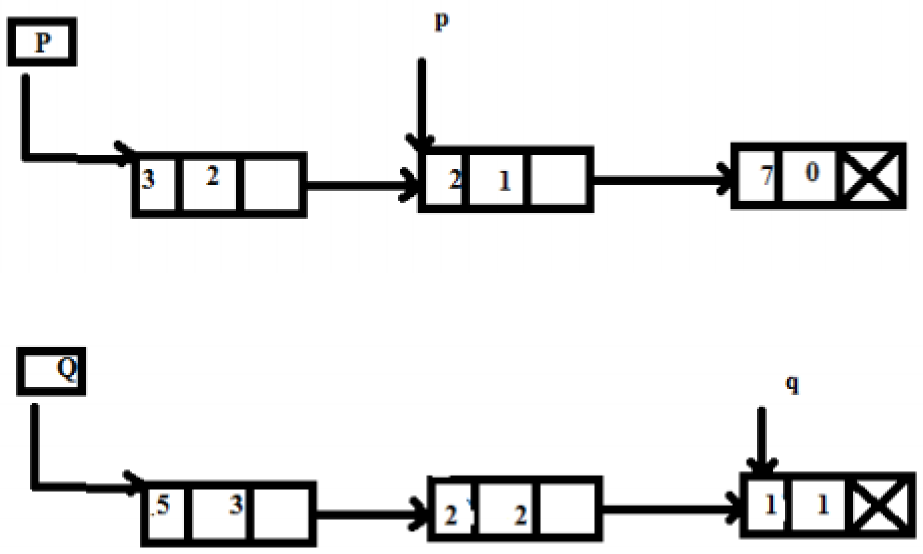
So, add the terms pointed to by q to the resultant list. And now advance the q pointer.



Step 2. Compare the exponent of the current terms. Here,

expo(p)=expo(q)

So, add the coefficients of these two terms and link this to the resultant list. And, advance the pointers p and q to their next nodes.

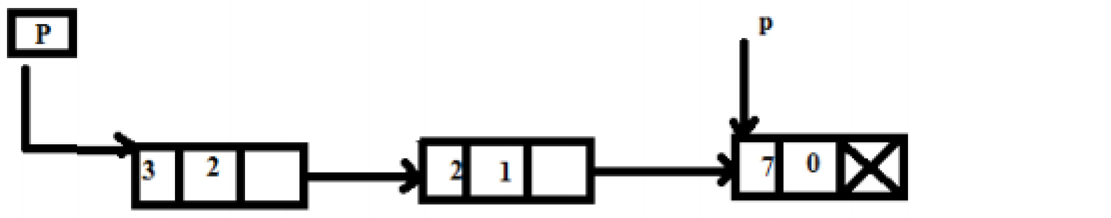


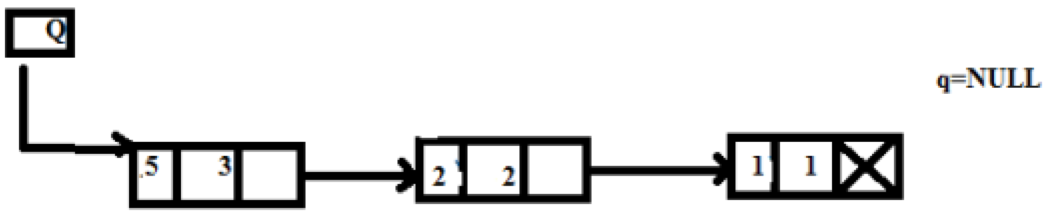
Step 3 : Compare the exponents of the current terms again

expo(p)=expo(q)

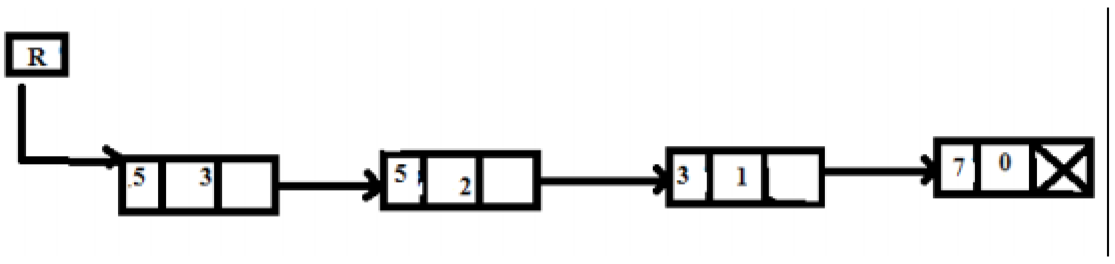
So, add the coefficients of these two terms and link this to the resultant linked list. And, advance the pointers to their next nodes. Q reaches the NULL and p points the last node.

Step 4. There is no node in the second polynomial to compare with. So, the last node in the first polynomial is added to the end of the resultant linked list.





Step 5. Display the resultant linked list. The resultant linked list is pointed to by the pointer R



Queue Using Linked List

The major problem with the queue implemented using array is, It will work for only fixed number of data. That means, the amount of data must be specified in the beginning itself. Queue using array is not suitable when we don't know the size of data which we are going to use. A queue data structure can be implemented using linked list data structure. The queue which is implemented using linked list can work for unlimited number of values. That means, queue using linked list can work for variable size of data (No need to fix the size at beginning of the implementation). The Queue implemented using linked list can organize as many data values as we want.

In linked list implementation of a queue, the last inserted node is always pointed by ' **rear** ' and the first node is always pointed by ' **front** '.

To implement queue using linked list, we need to set the following things before implementing actual operations.

Step 1: Include all the header files which are used in the program. And declare all the user defined functions.

Step 2: Define a 'Node' structure with two member’s data and next.

Step 3: Define two Node pointers 'front' and 'rear' and set both to NULL.

Step 4: Implement the main method by displaying Menu of list of operations and make suitable function calls in the main method to perform user selected operation.

enQueue(value) - Inserting an element into the Queue

We can use the following steps to insert a new node into the queue...

Step 1: Create a newNode with given value and set 'newNode → next' to NULL.
Step 2: Check whether queue is Empty (rear == NULL)
Step 3: If it is Empty then, set front = newNode and rear = newNode.
Step 4: If it is Not Empty then, set rear → next = newNode and rear = newNode.

deQueue() - Deleting an Element from Queue
We can use the following steps to delete a node from the queue...
Step 1: Check whether queue is Empty (front == NULL).
Step 2: If it is Empty, then display "Queue is Empty!!! Deletion is not possible!!!" and terminate from the function
Step 4: Then set 'front = front → next' and delete 'temp' (free(temp)).

display() - Displaying the elements of Queue
We can use the following steps to display the elements (nodes) of a queue...
Step 1: Check whether queue is Empty (front == NULL).
Step 2: If it is Empty then, display 'Queue is Empty!!!' and terminate the function.
Step 3: If it is Not Empty then, define a Node pointer 'temp' and initialize with front.
Step 4: Display 'temp → data --->' and move it to the next node. Repeat the same until 'temp' reaches to 'rear' (temp → next !=NULL).
Step 4: Finally! Display 'temp → data --->
NULL'.

Program

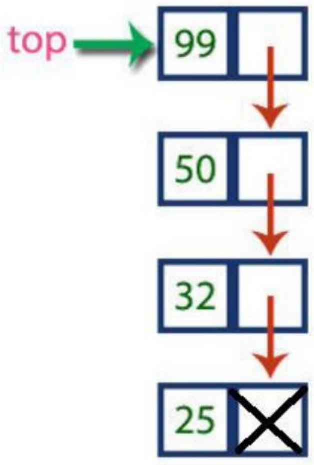
```
#include<stdio.h>
struct node
{
int data;
struct node *ptr;
};
int main()
{
typedef struct node NODE;
NODE *front=NULL,*rear=NULL,
*temp, *p;
int i, ch, pos, item;
while(1)
{
printf("\n\n\tQueue Using Linked
List\n1.Insert\n2.Delete\n3.Display\n4.
Exit\n\nEnter your
choice: ");
scanf("%d", &ch);
switch(ch)
{
case 1:
temp=(NODE*)malloc((sizeof(NODE)));
printf("Enter the data to be inserted: ");
scanf("%d", &temp->data);
if(rear==NULL)
{
temp->ptr=NULL;
front=rear=temp;
}
else
{
p=front;
while(p!=rear)
{
p=p->ptr;
}
p->ptr=temp;
temp->ptr=NULL;
rear=temp;
}
break;
case 2:
if(front==NULL)
{
printf("No elements to delete!");
}
else
{
```

```
if(front->ptr==NULL)
{
temp=front;
item=temp->data;
printf("Deleted element is %d", temp->data);
free(temp);
front=rear=NULL;
}
else
{
temp=front;
item=temp->data;
front=front->ptr;
free(temp);
printf("Deleted element: %d", item);
}
}
break;
case 3:
if(front==NULL)
{
printf("No elements");
}
else
{
printf("\nElements are: ")
p=front;
while(p!=rear)
{
printf("%d ", p->data);
p=p->ptr;
}
printf("%d", p->data);
}
break;
case 4:
exit(0);
}
}
getch();
}
```

Stack Using Linked LIST

The major problem with the stack implemented using array is, it works only for fixed number of data values. That means the amount of data must be specified at the beginning of the implementation itself. Stack implemented using array is not suitable, when we don't know the size of data which we are going to use. A stack data structure can be implemented by using linked list data structure. The stack implemented using linked list can work for unlimited number of values. That means, stack implemented using linked list works for variable size of data. So, there is no need to fix the size at the beginning of the implementation. The Stack implemented using linked list can organize as many data values as we want.

In linked list implementation of a stack, every new element is inserted as '**top**' element. That means every newly inserted element is pointed by '**top**'. Whenever we want to remove an element from the stack, simply remove the node which is pointed by '**top**' by moving '**top**' to its next node in the list. The **next**field of the first element must be always **NULL**



In above example, the last inserted node is 99 and the first inserted node is 25. The order of elements inserted is 25, 32,50 and 99.

push(value) - Inserting an element into the Stack

We can use the following steps to insert a new node into the stack...
Step 1: Create a newNode with given value.
Step 2: Check whether stack is Empty (top == NULL)
Step 3: If it is Empty, then set newNode → next = NULL.
Step 4: If it is Not Empty, then set newNode → next = top.
Step 5: Finally, set top = newNode.

pop() - Deleting an Element from a Stack

We can use the following steps to delete a node from the stack

Step 1: Check whether stack is Empty (top == NULL).
Step 2: If it is Empty, then display "Stack is Empty!!! Deletion is not possible!!!" and terminate the function
Step 3: If it is Not Empty, then define a Node pointer 'temp' and set it to 'top'.
Step 4: Then set 'top = top → next'.
Step 5: Finally, delete 'temp' (free(temp))

Program

```
#include<stdio.h>
#include<conio.h>
struct node
{
int data;
struct node *ptr;
};
void main()
{
typedef struct node NODE;
NODE *top=NULL,*temp,*t;
int ch,item;
clrscr();
while(1)
{
printf("\nMENU:\n1.Push\n2.Pop\n3.Display\
n4.Exit\nEnter your choice: ");
scanf("%d",&ch);
switch(ch)
{
case 1:
temp=(NODE*)malloc(sizeof(NODE));
printf("\nEnter the item: ");
scanf("%d",&item);
temp->data=item;
if(top==NULL)
{
temp->ptr=NULL;
top=temp;
}
else
{
temp->ptr=top;
top=temp;
}
break;
case 2:
if(top==NULL)
printf("\nDeletion is not possible");
else if(top->ptr==NULL)
{
temp=top;
top=NULL;
printf("\nPoped item is %d",temp->data);
free(temp);
}
Else
{
temp=top;
top=top->ptr;
printf("\nPoped item is %d",temp->data);
free(temp);
}
break;
case 3:
if(start==NULL)
printf("\nStack is empty");
else
{
printf("\nElements are:");
for(t=top;t!=NULL;t=t->ptr)
printf(" %d",t->data);
}
break;
case 4:
exit(0);
default:
printf("\nWrong Choice");
break;
}
getch();
}
}
```