Linked List

Advantages of Array:

1. An array implementation allows Print to be carried out in linear time and Find K^{th} operation in constant time, which is good as can be expected
2. Random access is possible
3. Implementation of list using array is easier as compared to other implementations

Disadvantages of Array:

1. Elements of arrays are always stored in contiguous memory
    a. Inserting or deleting an element in an array may require all of its elements to be shifted
2. The size of array is always fixed
    a. You cannot add a new element beyond the end of the array
    b. Memory for the entire array is always reserved even though you use only part of the array
    c. You must guess the expected maximum size of the list in advance

Linked List

A list is a sequence of related data items. It can be an ordered collection of integer items like 3,40,75,98,100, or it can be a collection of structures like:

```
struct student{
                char name[20];
                char class[10];
                int grade;
        };
```

with each structure containing a set of related data items.
Stacks and queues are special type of lists. We call them ***restricted list*** because insertions and deletions are restricted to occur in these structures only at the ends of a list.

A **linked list** is a series of data items with each item containing a pointer giving a location of the next item in the list. A **linked list** is a collection of nodes, where each node consists of two parts

- **info**: the actual element to be stored in the list.
- **link**: one or two links that points to next and previous node in the list. also known as pointer.

The entire link list is accessed by an external pointer (say list) that points to the first node in the linked list.

Basic Operations on linked list

- Insertion

This operation creates a new node and inserts it into the list. A new node may be inserted at

- at the beginning of the list (insertHead)
- at the end of the list (insertTail)
- at the specified position (insertAfter, insertBefore,insertAtNthPos)
- if the list is empty, the node to be inserted will be the first node

- Deletion

This operation deletes a node (i.e. an item) from the linked list. A node may be deleted from the

- beginning of the list (deleteHead)
- end of the list (deleteTail)
- specified position (deleteAfter, deleteAtNthPos etc)

- Traversing

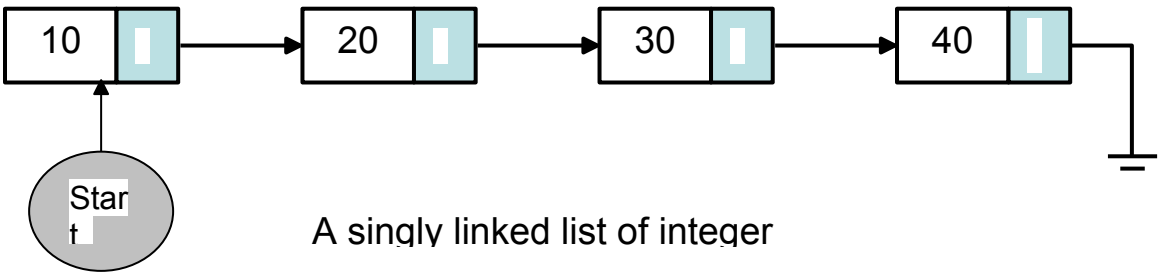
The process is visiting all the nodes of the linked list from one end to the other is termed as traversing the list.

- Searching

This operation checks if a particular item is present in the list.

-Display

This operation is used to print the content of the list.



A singly linked list of integer

i. The next address field of the **last node** in the list contains a special value, known as **null**, which is not a valid address. This null pointer is used to signal the end of a list.
ii. The list with **no nodes** on it is called the **empty list** or the null list.
iii. The nodes in a linked list are not stored contiguously in the computer's memory

iv. We don't have to **shift** any element while **inserting/deleting** in the list
v. Memory for each node can be allocated dynamically whenever the need arises
vi. The no of items in the list is only limited by the computer's memory

Creation of Linked List

Linked list can be created as:

```
struct list
{
  int item;
   struct list *next;
};
typedef struct list node;
node *ptr, *start;
ptr = (node *) malloc (sizeof(node));
```

The struct declaration merely describes the format of the nodes and does not allocate storage. Here, the type of the information to be stored is 'int'.next is a pointer to the parent structure type. This type of structure is called self-referential structures. Storage space for a node is created only when the function malloc() is called in the statement

ptr = (node *)malloc(sizeof(node)); - (1)
start = ptr; - (2)

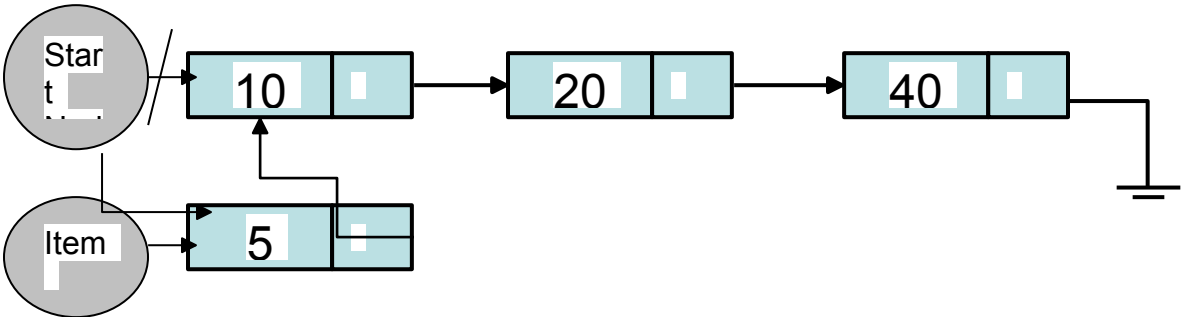
This statement (1) obtains a piece of memory that is sufficient to store a node and assigns (2) its address to the pointer variable **start**. This **start** pointer indicates the beginning of the linked list.

Insertion

To insert a node in a list,

- allocate required memory to a node
- assign the data in its information field
- adjust the pointers to link the node at the required place

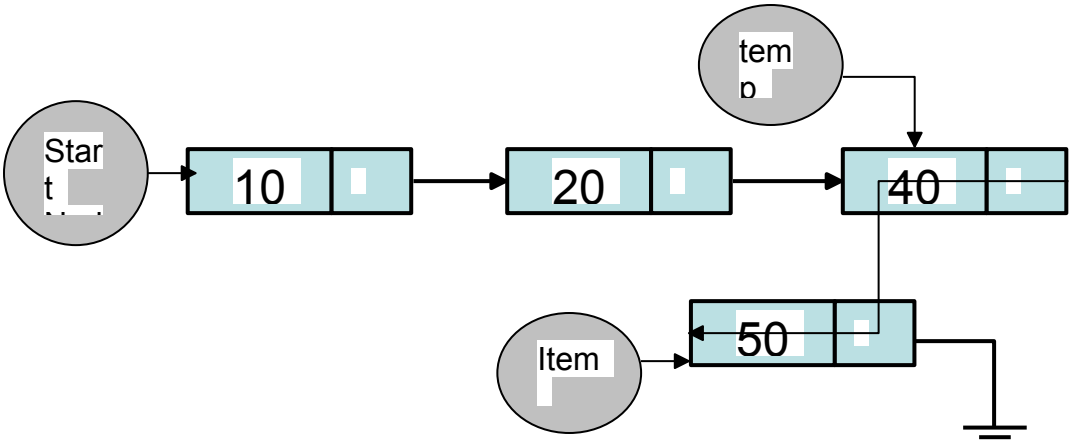
1. Algorithm for inserting a new node at the **beginning** of the list
   i. Allocate memory for the new node
   ii. Assign value
   iii. If the list is currently empty, assign NULL to the link field of the new node. Otherwise, make the link field of the new node to point to the starting node of the linked list.
   iv. Then, set the external pointer (which is pointing to the starting node) to point to the new node.



C Code

```
Void f_insert()
{
        node *item;
        item=(node *)malloc(sizeof(node));
        printf("Enter data to insert");
        scanf("%d",&item->info);
        if(start==NULL)
        {
                item->next=NULL;
                start=item;
        }
        else
        {
                item->next=start;
                start=item;
        }
}
```

2. Algorithm for inserting a new node at the **end** of the list
   i. Allocate memory for the new node
   ii. Assign value
   iii. Assign NULL to the link field of the new node.
   iv. Traverse the list until last node is reached. Then, insert the new node after the last node by storing the address of the newnode (item) to link field of the last node.

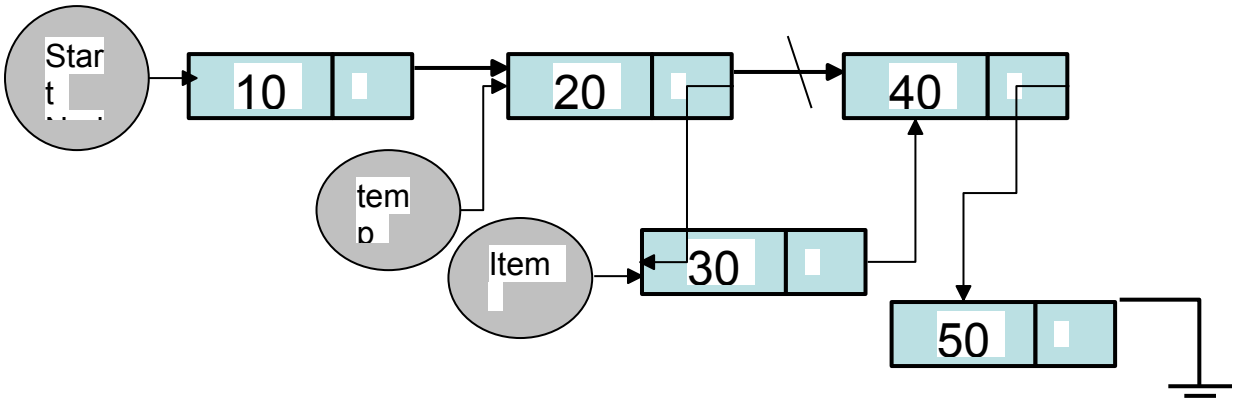

tem
p
Star
t
10
20
40
50
Item

C Code

```
void e_insert()
        {
                node *item,*temp;
                item=(node *)malloc(sizeof(node));
                printf("\nEnter data to insert:");
                scanf("%d",&item->info);
                item->next=NULL;
                if(start==NULL)
                start=item;
                else
                {
                        temp=start;
                        while(temp->next!=NULL)
                                temp=temp->next;
                        temp->next=item;
                }
        }
```

3. Algorithm for inserting a new node at specific position
   i. Allocate memory for the new node
   ii. Assign value
   iii. Assign next of the given node to the link field of the new node.
   iv. Assign newnode (item) to the link field of the given node

C Code



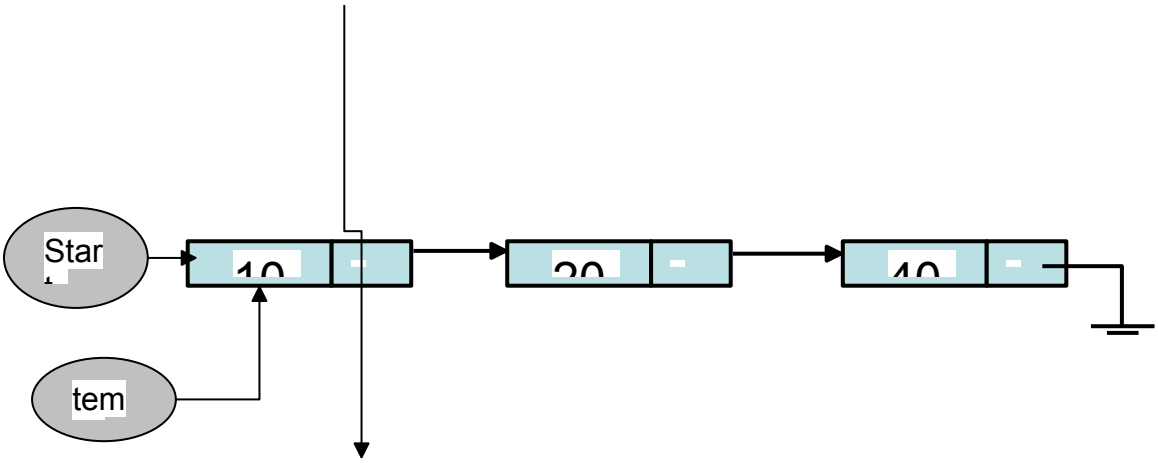

```
Void n_insert()
{
        int n,count=1;
        node *ptr,*temp;
        item=(node *)malloc(sizeof(node));
        printf("\nEnter the position");
        scanf("%d",&n);
        printf("enter the data to insert");
        scanf("%d",&item->info);
        temp=start;
        while(count<n)
        {
                temp=temp->next;
                count++;
        }
        item->next=temp->next;
        temp->next=temp;
}
```

Deletions

To delete a node from a list
- Find out the node to be deleted.
- Adjust the pointers
- Free the node to be deleted

Algorithm for deleting the **first** node

1. If the list is empty, it is underflow.
2. Else,
   - move the start pointer to the second node
   - free the first node



C Code

```
void f_del()
{
        node *ptr;
        if(start==NULL)
        {
                printf("\n Empty list");
                getch();
        }
        else if(start->next==NULL)
        {
                printf("\nDeleted element is %d",start>info);
                getch();
                free(start);
                start=NULL;
        }
        else
        {
                ptr=start;
                start=start->next;
                printf("\n Deleted element is%d",ptr->info);
                getch();
                free(ptr);
        }
}
```

Algorithm for deleting the **last** node

1. If the list is empty, it is underflow

2. Else, traverse until the last node is encountered.
   - Also keep track of the node, which is before the current node.
3. make the second last node (i.e. temp1) to point to NULL and free the last node(i.e. current)

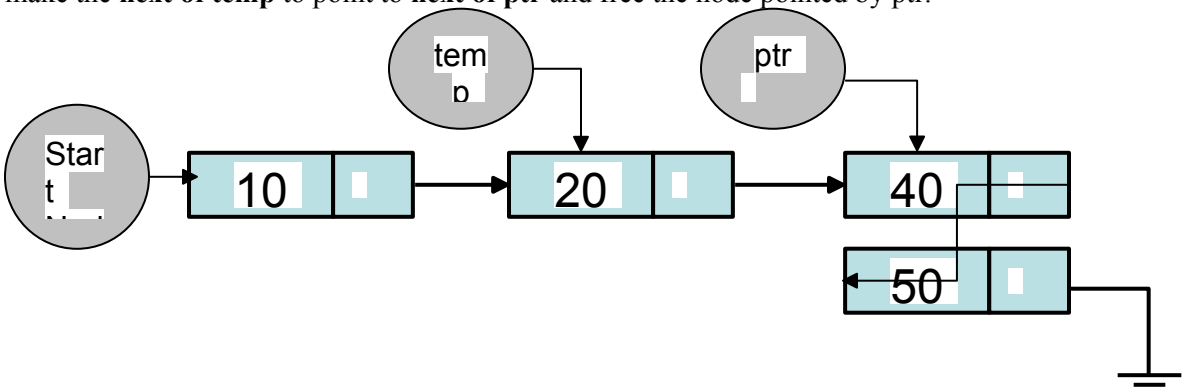
C Code

```
void e_del()
{
        node *ptr,*loc;
        if(start==NULL)
        {
                printf("\nEmpty list");
                getch();
        }
        else if(start->next==NULL)
        {
                printf("\nDeleted item is %d",start->info);
                free(start);
                start=NULL;
        }
        else
        {
                ptr=start;
                while(ptr->next!=NULL)
                {
                        loc=ptr;
                        ptr=ptr->next;
                }
                loc->next=NULL;
                printf("\nDeleted element is %d",ptr->info);
                getch();
                free(ptr);
        }
}
```

Algorithm for deleting a node at the **specified position (n)**

1. If the list is empty, it is underflow
2. Else, traverse until the specified node is encountered (ptr)
   - Also keep track of the node, which is before the specified node.

3. make the **next of temp** to point to **next of ptr** and free the node pointed by ptr.



C Code

```
void n_del()
{
    node *temp,*ptr
    int n,count=1;
    if(start==NULL)
    {
        printf("\nEmpty list");
        getch();
    }
    else
    {
        printf("\n Enter a position");
        scanf("%d",&n);
        ptr=start;
        while(count<n)
        {
                count++;
                temp=ptr;
                ptr=ptr->next;
        }
        temp->next=ptr->next;
        printf("\nDeleted item is %d",ptr->info);
        getch();
        free(ptr);
    }
}
```

Traversing a list and displaying information stored

As has been stated, list traversal is the process of visiting all the nodes in the list. Traversal is generally needed to print the items stores in each node of the list. The following function, traverse the given list and prints the item stored in each of them.

Algorithm for **traversing** and **displaying**

1. If the list is empty, print no element exist.
2. Else, traverse until the temp is not NULL (temp!=NULL)
   -While traversing print the info part each time.

C Code

```
void display()
{
        node *temp;
        printf("\nElements of linked list are");
        temp=start;

        while(temp!=NULL)
        {
                printf("%d\t",temp->info);
                temp=temp->next;
        }
        getch();
}
```

Searching for an item in a list

In order to find an item in a given list, we visit the nodes one after another until we find the item.

1. Start with the first node in the list, as the temp node and keep track of count .
2. Check the info at the current node with the search item
   If the items match, print the value with its position (counter).
   Otherwise, move to the next node and repeat the process
3. If no match is found till we reach the end of the list, print search unsuccessful.

C Code

```
void search()
{
        node *temp;
        int num,count=0;
        printf("\n Enter a number to search");
        scanf("%d",&num);
        temp=start;
        do
        {
                count++;
                if(temp->info==num)
                {
                        printf("\nThis number is at %d position",count);
                        getch();
                        temp=NULL;
                        return;
                }
                else
                        temp=temp->next;
        }
        while(temp!=NULL);
        printf("\nThis number doesnot exist in list");
        getch();
}
```

Destroying a List

When the list is not needed any more, we delete all the nodes of the list by releasing the memory occupied by them.
For this, delete the nodes one by one from the beginning till the last node.

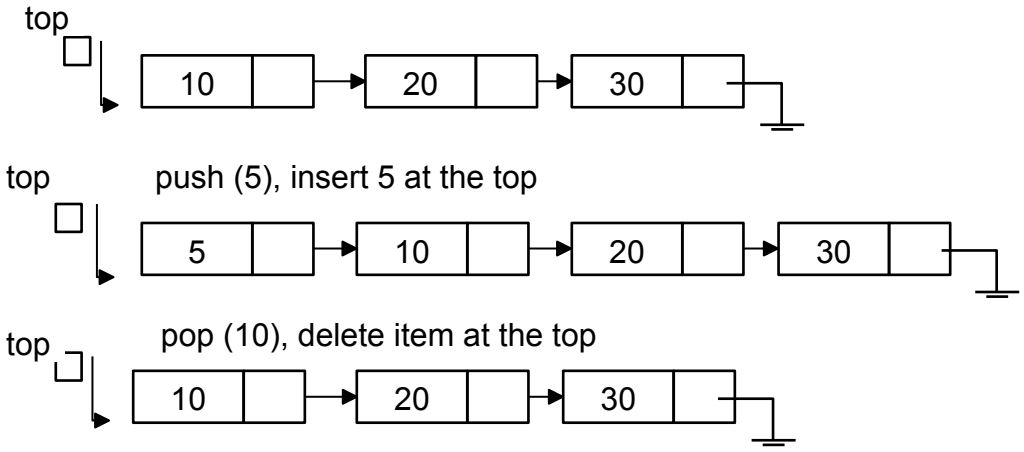
C Code

```
void destroy()
{
        struct node *temp;
        temp = NULL;
        while (start != NULL)
        {
                temp = start;              // Store current node, before deleting
                 start = start->next;      // Move to next node
                free(temp);               // Release memory occupied by temp
        }
}
```

Implementation of Stack using Linked List

As has already been studied, stack is a collection of items in which addition of new items and deletion of existing items are done from only one end known as top of stack. Since, linked list is also a list of items, concepts of stack can be implemented using linked lists instead of an array.

i. The two operations that we apply on a stack are push (inserting item on the top) and pop (deleting item at the top).
ii. The **push** operation is similar to **adding a new node at the beginning of the list** (insertHead()).
iii. A stack items can be accessed only through its top, and a list items can be accessed from the pointer to the first node.
iv. Similarly, **removing the first node** from a linked list (deleteHead()) is similar to **popping** an item from a stack. In both cases, the only immediately accessible item of a collection is removed from that collection, and the next item becomes immediately accessible one.
v. The first node of the list is the top of the stack



Implementing Queue using Linked List

A queue is a special type of list that allows insertion at one end, called the front of queue, and deletion at other end, called the rear of queue.

i. To implement a queue in a linked list, we make the first node of the list as the front and the last node as the rear. So, we will use two pointer variables – front and rear. 'front' will point to the beginning of the list (like start) while 'rear' will point to the last node of the list .
ii. Initially, when the queue is empty, both rear and front will be NULL.
iii. To **enqueue** a new item, we insert it after the rear node and update rear to point to new node. This function will work similarly as insertTail(), because both are the process to add new item after the last node. In the case of Queue, we insert a new item after the node pointed by 'rear'.

iv. To **dequeue** an item, we remove the front node and update front to point to the next node. This function will work similarly as deleteHead() of general list or pop() operation of stack.

Advantages of Linked List over Contiguous List

i. A Linked list is a dynamic data structure. Therefore, the primary advantage of linked lists over arrays is that linked lists can grow or shrink in size during the execution of a program. A linked list can be made just as long as required.
ii. Another advantage is that a linked list doesn't waste memory space. It uses the memory that is just needed for the list at any point of time. This is because it is not necessary to specify the number of nodes to be used in the list.
iii. The third and the more important advantage is that the linked lists provide flexibility in allowing the items to be rearranged efficiently; it is easier to insert or delete items by rearranging the links i.e. the items can be inserted randomly at any position without too much difficulty.

Array Vs List

i. The number of elements of an array is fixed when the array is created. In contrast, the number of elements in a list can be varied by adding or removing elements.
ii. The index of a component of an array is unique and fixed. In contrast, the position of an element in a list can vary over time, e.g. if we remove the first element of a list then the element which was previously second in the list now becomes the first.
iii. All components of an array can be accessed easily in constant time, but there is no statement that the same is true for the elements of a list

Limitations of Linked Lists

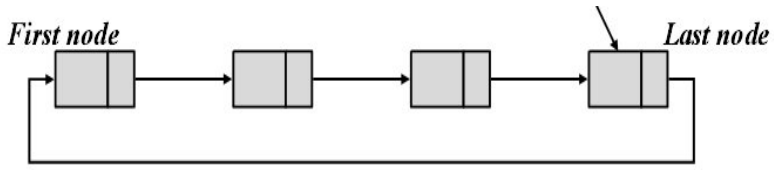
i. Whenever we deal with a fixed length list, it would be better to use an array rather than a linked list.
ii. We must also note that a linked list will use more storage than an array with the same number of items. This is because each item has an additional link field.
iii. To visit a particular list we have to start right from the starting node.

Circular Linked List

This problem can be solved by using a circular linked list. In such a list it is possible to reach any other point in the list. A circular linked list is a list, where the next link of the last node contains a pointer to the first node rather than a null pointer. If we begin at a given node and traverse the entire list, we ultimately end up at the starting point.

In a circular list, we don't have a first or a last node. But, following a convention, we assume that the external pointer points to a last node. The node following this node will be called the first node.

last



This convention allows access to the last and the first node (by referencing last->next)
Also, this provides the advantage of being able to add or remove an element conveniently from either the front or the rear of the list where as in single linked list, to add a node at the end we have to start right from the beginning to reach the last node.
For an empty circular list is, last = NULL

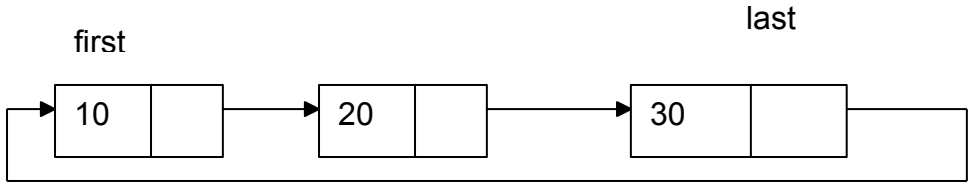


Fig: Circular Linked list

Primitive operations in Circular List
Most of the operations on circular lists are similar to singly linked lists. However, care has to be taken for non-empty lists, as there is no null pointer to signify the end of the list.

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

```
Struct list
{
        int info;
        struct linked_list *next;
}
typedef struct list node;
node *start=NULL;
node *last=NULL;
```

Algorithm for **inserting** a node **at the beginning**

```
1.      ptr=(node *)malloc(sizeof(node))
2.      set ptr->info=item
3.      if start==NULL then
                set ptr->next=ptr
                set start=ptr
                set last=ptr
        else follow step 4 to  6
4.              set ptr->next=start
```

5. set start=ptr
6. set last->next=ptr

Algorithm for **inserting** a new node **at the end**

1. ptr=(node *)malloc(sizeof(node))
2. set ptr->info=item
3. if start==NULL
   set ptr->next=ptr
   set start=ptr
   set last=ptr
   else follow step 4 to 6
4. set last->next=ptr
5. set last=ptr
6. set last->next=start

Algorithm for **deleting** the node from the **beginning**

1. if start==NULL
   print “circular list is empty”
2. else follow step 2
   if start->next=start
   print “Deleted element is start->info”
   free(start)
   start=NULL
   last=NULL
   else follow step 3 to 7
3. set ptr = start
4. set start = start->next
5. print “deleted element is ptr->info”
6. set last->next=start
7. free(ptr)

Algorithm for **deleting** the node from the **end**

1. if start==NULL
   print “circular list is empty”
   else follow step 2
2. if start->next=start
   print “Delement element is start->info”
   free(start)
   start=NULL
   last=NULL
   else follow step 3 to 10
3. set ptr=start
4. Repeat step 5 and 6 until ptr->next=start

```
5.                  set temp=ptr
6.                  set ptr=ptr->next
7.          print "the deleted item is ptr->info"
8.          set temp->next=ptr->next
9.          set last=temp
10.         free(ptr)
```

Doubly Linked List

In both, singly list and a circular list, list can be traversed only in forward direction. But in doubly list, we can traverse the list in both directions. This has been made possible by defining two address fields, one of them points to the previous(left) node while the other one points to the next(right) node.

The external pointer will point to the first node, as in singly list.
While performing the basic operations, we need to update both(left and right) address fields.

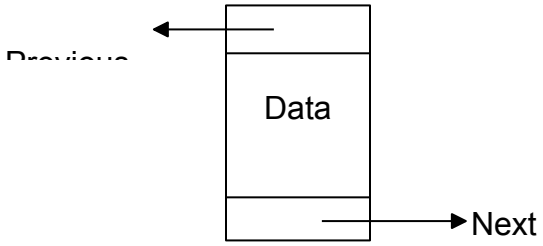


Fig: A node representation with doubly

We declare the structure for the doubly linked as:

```
struct list
{
        int info;
        struct list *prev;
        struct list *next;
};
typedef struct list node;
node *start=NULL;
```

Algorithm for **inserting** a node **at the beginning**

1. ptr=(node *)malloc(sizeof(node))
2. set ptr->info=item

3. if(start==NULL) then
   set ptr->prev = ptr->next = NULL
   set start = ptr
   else follow step 4 to 7
4. set ptr->prev=NULL
5. set ptr->next=start
6. set start->prev=ptr
7. set start=ptr

Algorithm for **inserting** a node **at the end**

1. ptr=(node *) malloc (sizeof(node))
2. ptr->info=item
3. if(start==NULL) then
   set ptr->prev = ptr->next = NULL
   set start = ptr
   else follow step 4 to 7
4. set ptr->next=NULL
5. ptr->prev=last
6. last->next=ptr
7. last=ptr