Linked List

- It is a list or collection of data items that can be stored in scattered locations (positions) in memory.
- To store data in scattered locations in memory we have to make link between one data item to another. So, each data item or element must have two parts: one is data part another is link (pointer) part.
- Each data item of a linked list is called a node.
- Data part contains (holds) actual data (information) and the link part points to the next node of the list.
- To locate the list an external pointer is used that pints the first node of the list. The link part of the last node will not point to any node. So, it will be null. This type of list is called linear (one way) linked list or simply linked list.

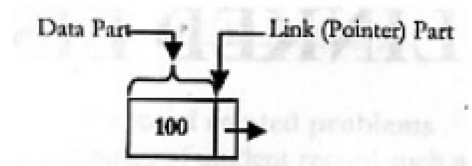


Figure 1: A single node

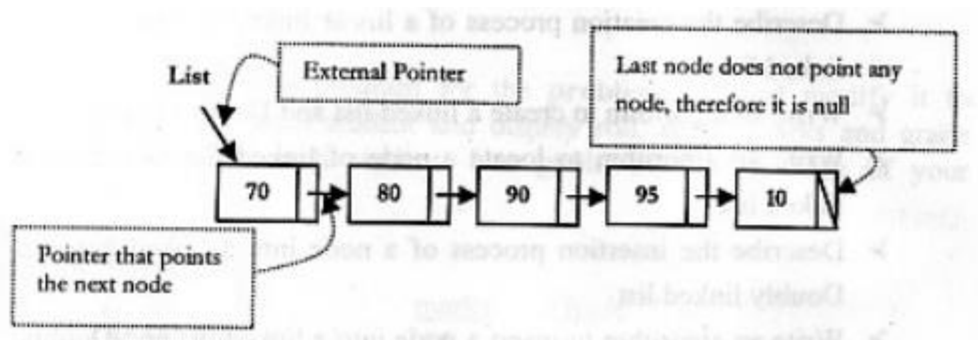
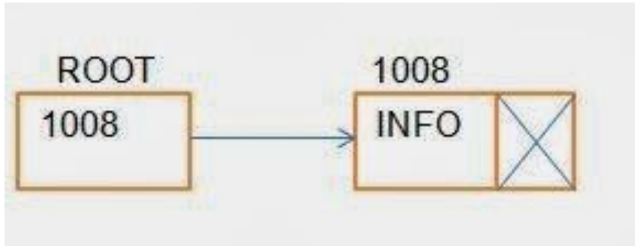


Figure 2: Graphical representation of a linear linked list

Memory representation of Linked List

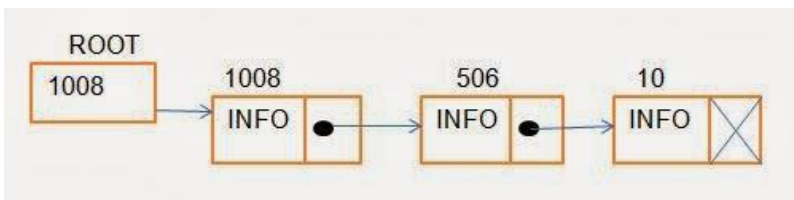
In memory the linked list is stored in scattered cells (locations). The memory for each node is allocated dynamically means as and when required. So the Linked List can increase as per the user wish and the size is not fixed, it can vary.

Suppose first node of linked list is allocated with an address 1008. Its graphical representation looks like the figure shown below:

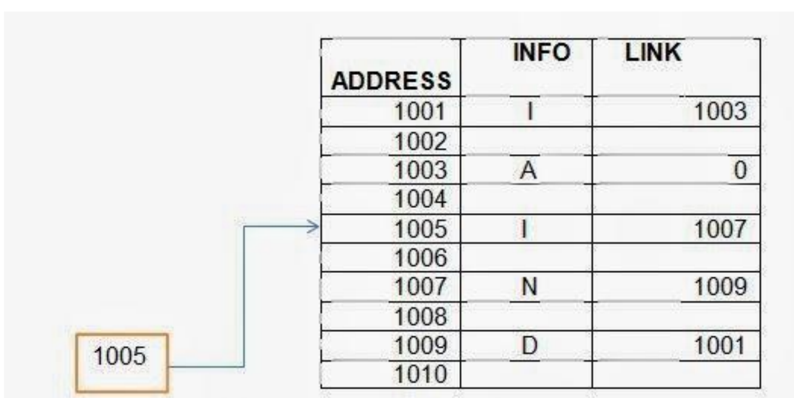


Suppose next node is allocated at an address 506, so the list becomes,

Suppose next node is allocated with an address with an address 10,s the list become,



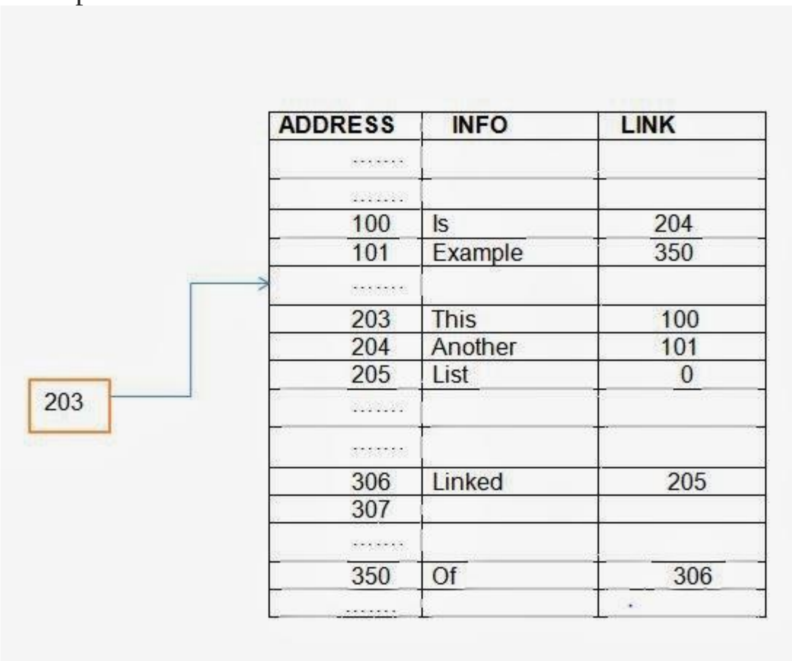
The other way to represent the linked list is as shown below:



ADDRESS	INFO	LINK
1001	I	1003
1002		
1003	A	0
1004		
1005	I	1007
1006		
1007	N	1009
1008		
1009	D	1001
1010		

In the above representation the data stored in the linked list is "INDIA", the information part of each node contains one character. The external pointer root points to first node's address 1005. The link part of the node containing information I contains 1007, the address of next node. The last node of the list contains an address 0, the invalid address or NULL address.

Example:



ADDRESS	INFO	LINK
....		
....		
100	Is	204
101	Example	350
....		
203	This	100
204	Another	101
205	List	0
....		
....		
306	Linked	205
307		
....		
350	Of	306
....		

The data stored in the linked list is: “The Is Another: Example Of Linked List”

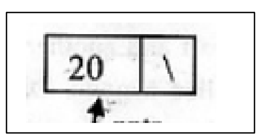
Node declaration and store data in a node

1. Node declaration

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

2. Data store:

```
node *nptr;
nptr -> data = 20;
nptr -> next = NULL;
```



Create a new node

1. Node declaration

```
struct node
        {
```

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

2. Declare variables (pointer type) that point to the node:

```
node *nptr;
```

3. Allocate memory for new node:

```
nptr = new(node);
```

4. Insert node value:

```
nptr -> data = 55;
nptr -> next = NULL;
```

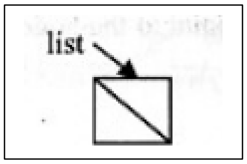
Algorithm (pseudo code) to create a linked list

1. Declare node and pointers (list, tptr, nptr):

```
i. struct node
   {
   int data;
   node *next;
   };
ii. node *list, *tptr, *nptr;
```

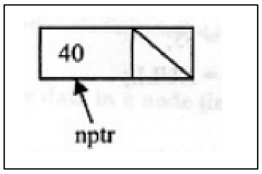
2. Create an empty list:

```
list = NULL;
```



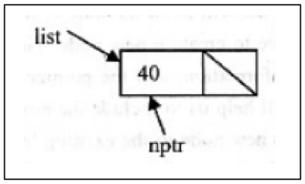
3. Create a new node:

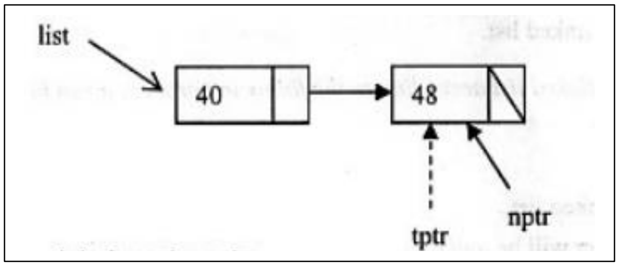
```
nptr = new(node);
nptr -> data = item;
nptr -> next = NULL;
```



4. Make link between the linked list and the new node:

```
if (list==NULL)
   {
   list = nptr;
   tptr = nptr;
   }
else
   {
   tptr -> next = nptr;
   tptr = nptr;
   }
```





4. Output linked list.

Note: Step 3 and 4 will be repeated again and again if two or more nodes are to be added in the list.

Comments: Here, **nptr** is a pointer that points to a new node, The **tptr** is the pointer that points the last node, which has been already added. **item** is a variable using which we shall enter data to the new node.

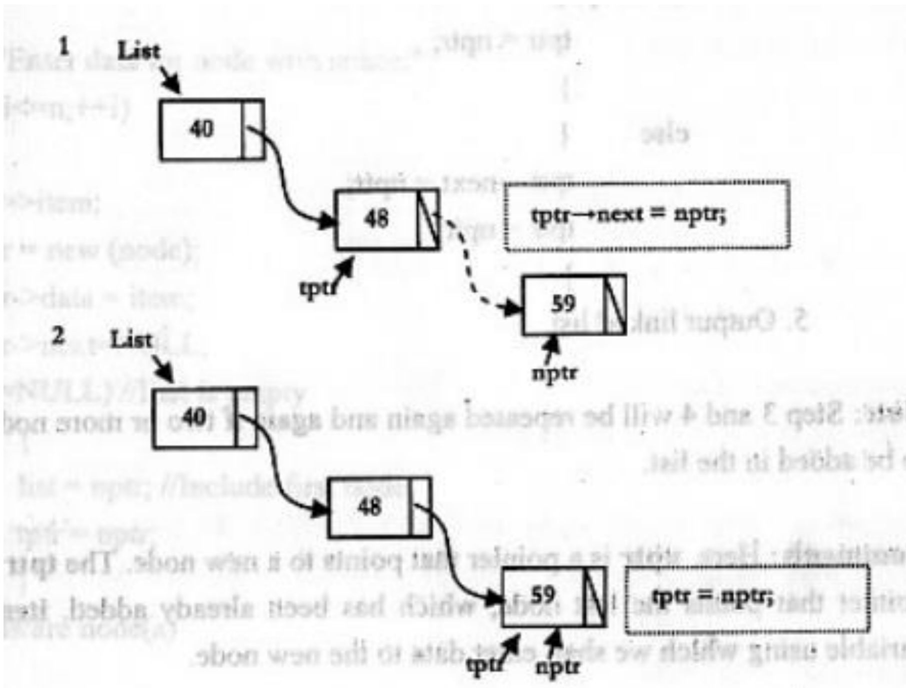


Figure 3: Addition of nodes to a linked list (pictorial view)

Algorithm (pseudo code) to search a node from a linked list

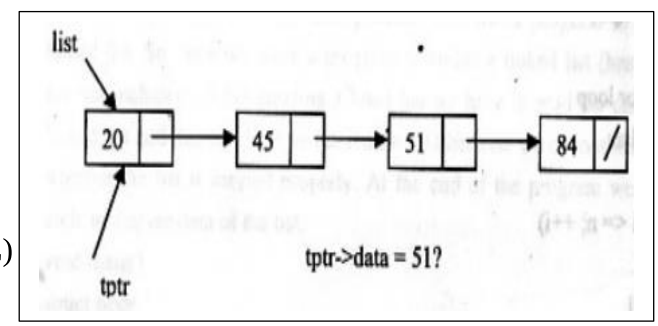
1. Declare node and tptr

2. Input the values to be located:

```
item = 51;
```

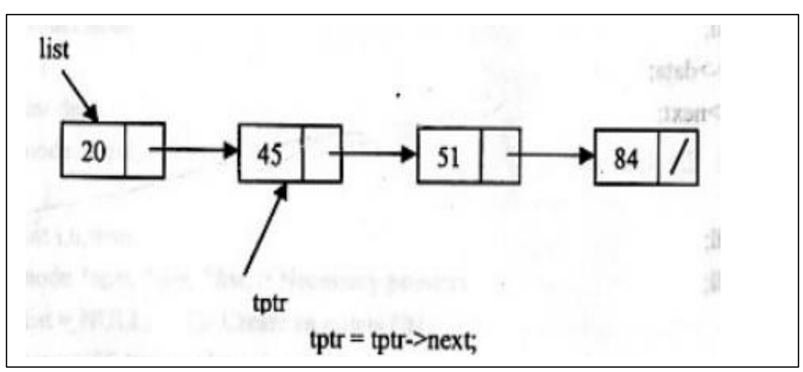
3. Search the item:

```
tptr = list;
while (tptr->data != item or tptr != NULL)
        {
        tptr = tptr->next;
        }
```



4. Output:

```
if (tptr -> data =  item)
        print "FOUND"
else print "NOT FOUND"
```



Algorithm (pseudo code) to insert a node into a linked list at beginning

1. Declare node and pointers (list, tptr, nptr)

2. Input linked list (we have to use an existing list)

3. Create a new node

```
nptr = new(node);
nptr -> data = item;
nptr -> next = NULL;
```

4. Make the link part of the new node pointing to the existing first node of the list.

```
nptr ->next = list;
```

5. Make the new node as the first node of the list.

```
list = nptr;
```

Algorithm (pseudo code) to insert a node into a linked list at end

1. Declare node and pointers (list, tptr, nptr)

2. Input linked list (we have to use an existing list)

3. Create a new node

```
nptr = new(node);
nptr -> data = item;
nptr -> next = NULL;
```

4. Declare a temporary pointer temp in order to traverse through the list

```
tptr = list;
```

5. traverse through the entire linked list

```
while (tptr ->next !=NULL)H
    {
        tptr = tptr -> next;
    }
```

6. Make the link part of the last node pointing to the new node of the list.

```
tptr =>next = nptr;
```

Algorithm to insert a node into linked list at given position

1. Declare node and pointers.

2. Input linked list, new node and the position of insertion (POSITION).

3. Find location to which the new node be inserted.

```
tptr=list;
for(i=1; i<POSITION; i++)
{
    if(tptr->next!=NULL)
    {
        tptr=tptr->next;
    }
}
```

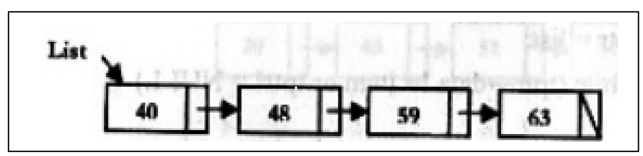
4. Make link between the new node with its next node and previous node.

```
nptr->next=tptr->next;
tptr->next=nptr;
```

5. Output: Updated linked list.

Algorithm (pseudo code) to insert a node into a linked list
[Here, we shall consider insertion after the first node or before the last node in an ascending list]

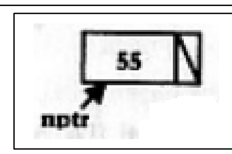
1. Declare node and pointers (list, tptr, nptr)



2. Input linked list (we have to use an existing list)

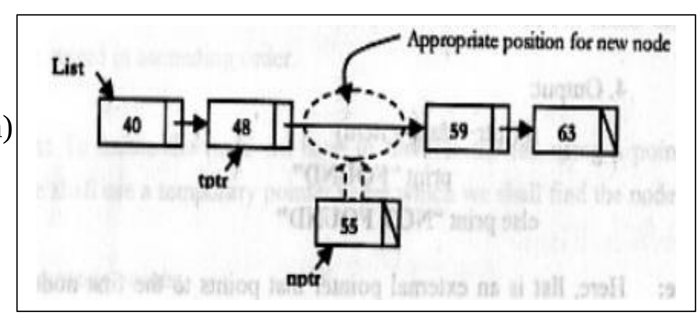
3. Create a new node

```
nptr = new(node);
nptr -> data = item;
nptr -> next = NULL;
```



4. Locate the appropriate position for the new node:

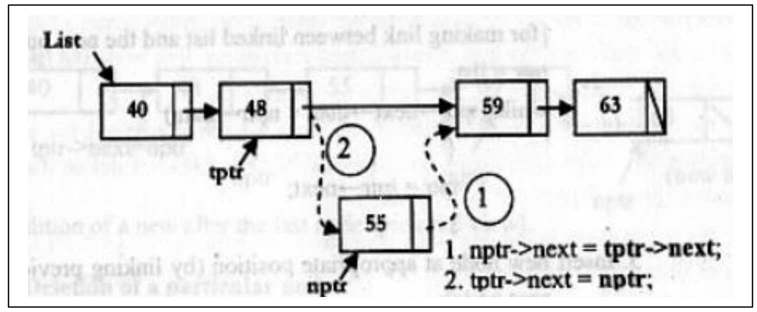
```
tptr = list;
while (tptr -> next -> data < nptr -> data)
{
tptr = tptr -> next;
}
```



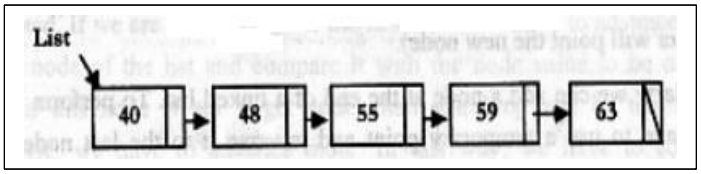
5. Insert new node at appropriate position

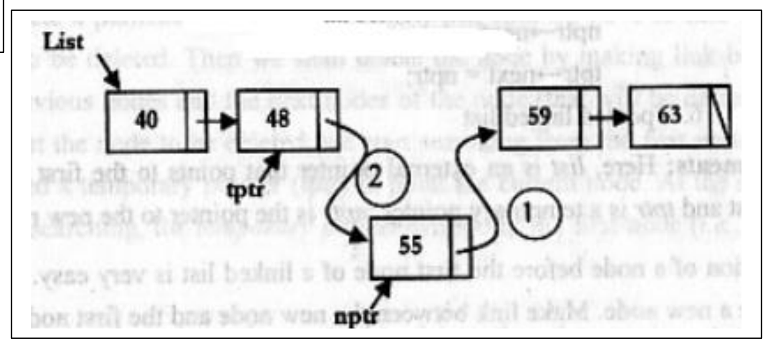
nptr -> next = tptr -> next;

tptr -> next = nptr;



6. Updated linked list.





Algorithm to delete a node from linked list at beginning

1. Declare node and pointers.

2. Input linked list.

3. Make the list, point to the next of the list.

tptr=list;

list=tptr->next;

free(tptr);

4. Output: Updated linked list.

Algorithm to delete a node from linked list at end

1. Declare node and pointers.

2. Input linked list.

3. Two Scenario: i. There is only one node in the list and that needs to be deleted. Ii. There are more than one node in the list and the last node of the list will be deleted.

i.

if(list->next=NULL)

{

```
        list=NULL;
        free(list);
    }
```

ii.

```
    tptr=list;
    while(tptr->next!=NULL)
    {
        pptr=tptr;
        tptr=tptr->next;
    }
```

4. Delete the node

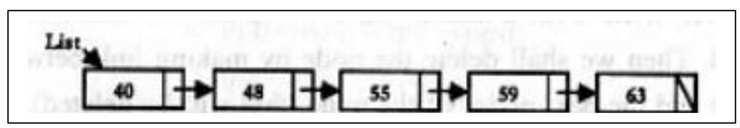
```
    pptr->next=NULL;
    free(tptr);
```

5. Output: Updated linked list.

Algorithm (pseudo code) to delete a node from a linked list
[Here, we shall not consider deletion process of the first node and the last node in the list]

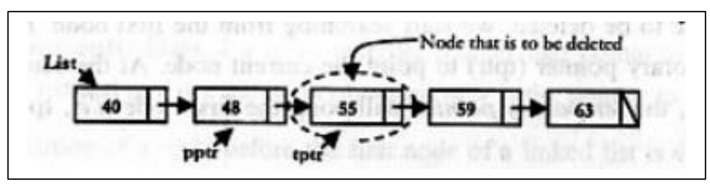
1. Declare node and pointers (list, tptr, pptr)

2. Input linked list and the item (that is to be deleted)



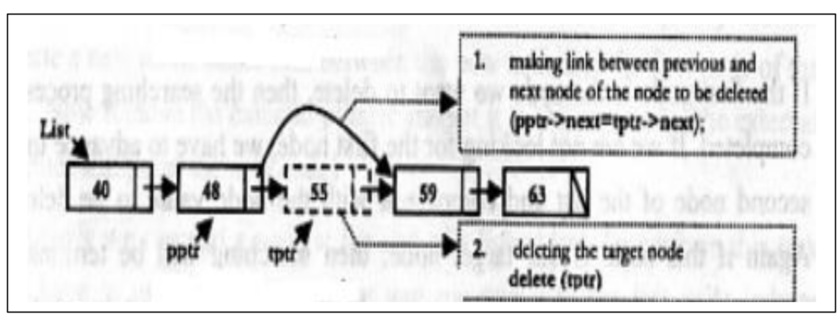
3. Search the item to be deleted in the list:

```
        tptr = list;
        while (tptr -> data !=item)
            {
            pptr =tptr;
            tptr = tptr -> next;
            }
```

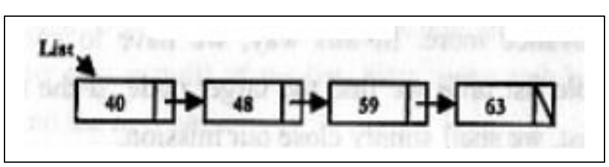


4. Delete the node

```
        pptr -> next = tptr -> next;
        delete (tptr);
```



5. Output: Updated linked lists.



Doubly Linked List

- A doubly or two way linked list is a list where each node has three parts.
- One is link or pointer to the previous (backward) node and one is data part to hold the data and another is link or pointer to the following (forward) node.
- There is an external pointer to the first node of the list.
- Doubly linked list is also called two-way linked list.

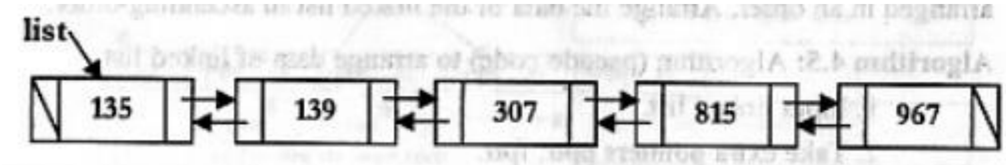


Figure 4: Graphical representation of a doubly linked list

Declare a node of a doubly linked list

1. Node declaration

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

Create a node

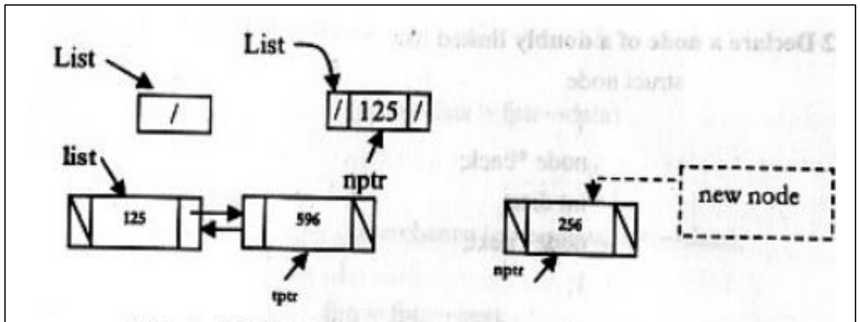
```
struct node
        {
        node *back;
        int data;
        node *next;
        };
node *nptr;
nptr = new(node);
nptr -> back = NULL;
nptr -> data = item;
nptr -> next = NULL;
```

Algorithm (pseudo code) to create a doubly linked list

1. Declare node and pointers
```
        a. struct node
                {
                node *back;
                int data;
                node *next;
                };
        b. node *list, *tptr;
```

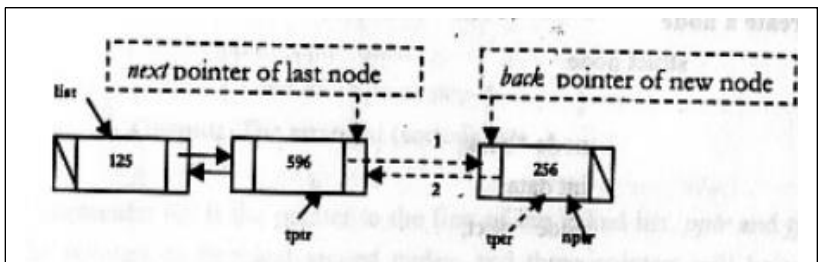
2. Create an empty list:
```
        list = NULL;
```

3. Create a new node:
```
        nptr = new(node);
        nptr -> back = NULL;
        nptr -> data = item;
        nptr -> next = NULL;
```



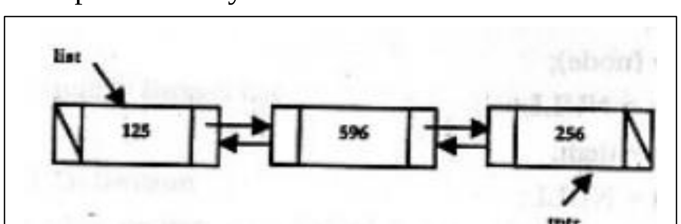
(a) A doubly linked list and new node

4. Make link between the last node of the list and the new node:
```
        if (list==NULL)
                {
                list = nptr;
                tptr = nptr;
                }
        else
                {
                tptr -> next = nptr;
                nptr -> back = tptr;
                tptr = nptr;
                }
```



(b) Linking the last node of the list and the new node

5. Output: a doubly linked list.



(c) Doubly linked list after making link

Circular Linked List

- A circular linked list is a list where each node has two parts.
- One is data part to hold the data and another is link or pointer part that points the next node and the last node's pointer points the first node of the list.
- Like other linked list there is an external pointer to the list to pint the first node.

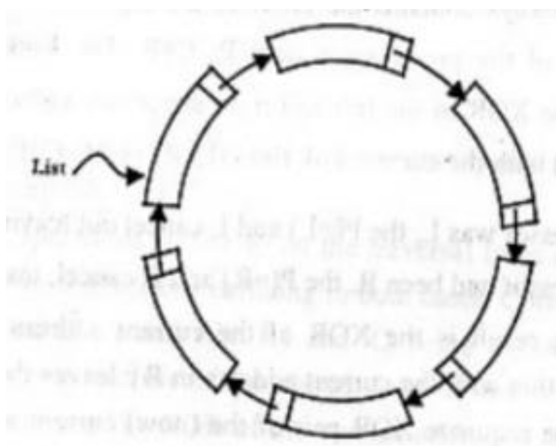


Figure 5: Pictorial view of a circular linked list (a circular diagram)

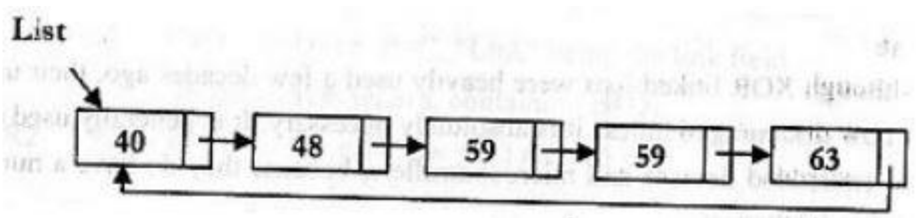


Figure 6: A circular linked list (linear diagram)

Algorithm (pseudo code) to create a circular linked list

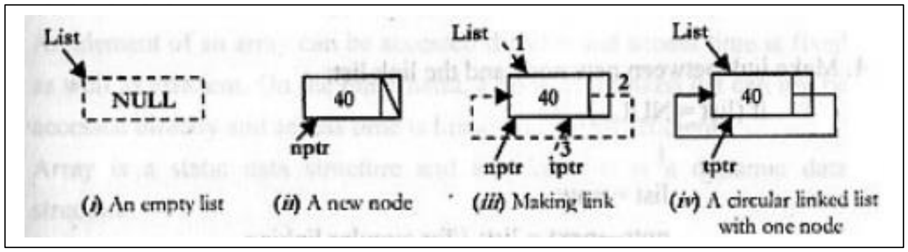
1. Declare node and pointers (list, tptr, nptr)

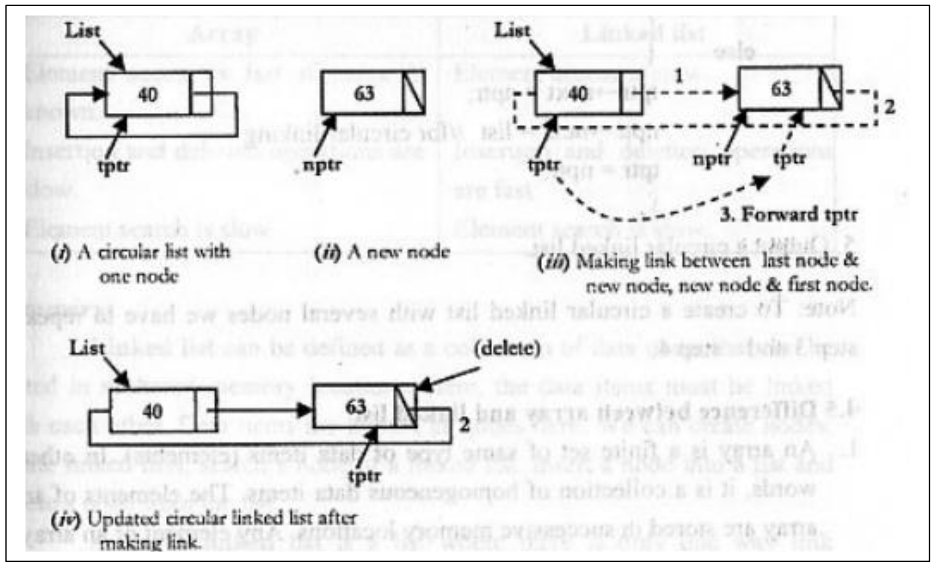
2. Create an empty list:
    list = NULL;

3. Create a new node with data:
    nptr = new(node);
    nptr -> data = item;
    nptr -> next = NULL;

4. Make link between the new node

```
if (list==NULL)
        {
        list = nptr;
        nptr -> next = list;
        tptr = nptr;
        }
else
        {
        tptr -> next = nptr;
        nptr -> next = list;
        tptr = nptr;
        }
```





5. Output: a circular linked list.

Differences between array and linked list

- An array is the data structure that contains a collection of similar type data elements whereas the Linked list is considered as non-primitive data structure contains a collection of unordered linked elements known as nodes.
- Random access is possible in array but in linked list, random access is not allowed
- Accessing an element in an array is fast, while Linked list takes linear time, so it is quite a bit slower.
- Operations like insertion and deletion in arrays consume a lot of time. On the other hand, the performance of these operations in Linked lists is fast.
- Arrays are of fixed size. In contrast, Linked lists are dynamic and flexible and can expand and contract its size.
- In an array, memory is assigned during compile time while in a Linked list it is allocated during execution or runtime.
- Elements are stored consecutively in arrays whereas it is stored randomly in Linked lists.

- The requirement of memory is less due to actual data being stored within the index in the array. As against, there is a need for more memory in Linked Lists due to storage of additional next and previous referencing elements.
- In addition memory utilization is inefficient in the array. Conversely, memory utilization is efficient in the linked list.