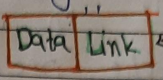
Subhash
Page :
Date :

# UNIT-3

## Linked List

Linked List can be defined as collection of data elements called nodes that are randomly stored in the memory.

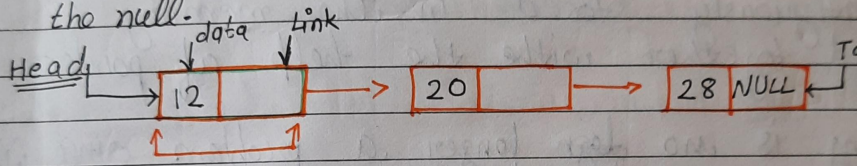
A node contain two fields



structure of node

→ data stored at that particular address
→ the pointer which contain the address of the next node in the memory.

- The last node of the list contains ~~address~~ pointer to the null.




### Uses of Linked List :⇒

1) The list is not required to be contiguously present in the memory. The node can be present/reside anywhere in the memory and linked together to make a list. This achieves optimized utilization of space.
2) List size is limited to memory size and doesn't need to be declare in advance.
3) Empty node can not be present in linked list
4) We can store values of primitive types in the singly linked list.

### Why use Linked List over array

Array contain the following limitations :⇒
1) Size of array must be known in advance before using it in the program.

Subhash
Page :
Date :

2. Increasing size of the array is time taking process. It is ~~also~~ almost impossible to expand the size of array at run time
(3) All the elements in array need to be contiguously stored in the memory, inserting any element in the array need shifting of all the predecessors.

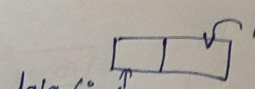
Linked list overcome all the limitations of an array. using linked list is useful bcoz
1) It allocates memory dynamically.
2) All the nodes of Linked List are non-contiguously stored in the memory & linked together with the help of pointer

3) Sizing is no ~~long~~ longer a problem since we do not need to define its size at the time of declaration. List grows as per the program's demand and limited to the available memory space.

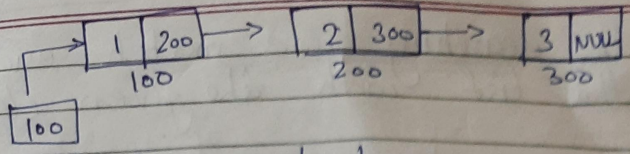
## Types

1) Singly / one way Linked List
2) Doubly or two way Linked List
3) Circular Linked List.
4) Doubly Circular Linked List.

(1) Singly linked list or one way linked list => This is the simplest type of Linked list. The singly linked list is a ds that contains two parts i.e., data part & other one is address part also known as a pointer.




Subhash
Page :
Date :




head or first or start

a) It use one external pointer ~~also~~ FIRST / HEAD for storing address of first node
b) The pointer of last node is set to NULL while other nodes contain the address of next nodes in their link.

Features => Contain only single link.
→ Only forward traversal is possible
→ We can't traverse in backward direction.

Representation of the node in a singly LinkedList

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

Operations on Singly Linked List :=>

1) Node Creation => A Linked List is formed when many nodes are linked together to form a chain.
2) Insertion => Adding an element to the Linked List.
3) Deletion => It involves deleting a node from the Linked List.
4) Traversing => Traversing means we simply visit each node of the list at least once in order to perform some operation.

Subhash
Page :
Date :

5) Searching => In searching we match each element of the list with the given element.

6) Sorting => To arrange nodes in a specific order.

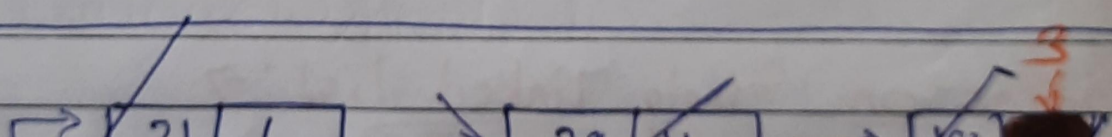
7) Updating => To update a node

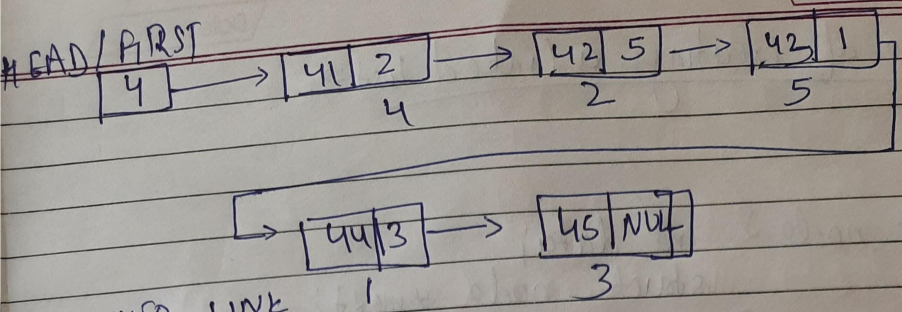
~~Insertion~~ => [Representation of Link List in memory :=>]

Linked List can be represented in memory by using two array => data/info and Link.

The Linked List can be represented in memory by

(1) In Linked List it is not necessary that the list must start from node[0] Rather, it can start from anywhere.






| | INFO | LINK |
|---|---|---|
| 1 | 44 | 3 |
| 2 | 42 | 5 |
| start 3 | 45 | NULL |
| 4 → 4 | 41 | 2 |
| 5 | 43 | 1 |
| 6 | | |

Let's create a node 'P'

```
# include <stdio.h>
# include <stdlib.h>
struct node {
        int data;
        struct node *next;
};
int main()
{
        struct node *prev, *head, *P;
P = (struct node*) malloc (sizeof (struct node));
(struct node*)
printf ("Enter the data");
scanf ("%d", &P->data);
P -> next = NULL;
return 0;
```

3.

Page :
Date :

## Example of creating a linked list =>

```
#
#
struct node { int data;
              struct node *next;
            };
void main(){
    struct node *prev, *head, *P;
    int n, i;
printf("enter the no. of elements");
scanf("%d", &n)
  head = NULL;
for( i=0; i<n; i++)
  {
    p = (struct node*) malloc(sizeof (struct node
    scanf("%d", &p->data);
      p -> next = NULL
      if(head == NULL)
        { head = P;
        }
        else
            prev -> next = p;
          { prev = p;
        }
}
```

## Dynamic memory allocation => malloc()

pointer = (cast type *) malloc ( byte-size)
int *ptr = (int *) malloc (sizeof (int));

Subhash
Page :
Date :

Traversing =>

Algorith => 1) set ptr = head;
2) if ptr = NULL
write "empty List"
goto step 6;
end of if

3. Repeat Step 4 & 5 until ptr != NULL

step 4: print ptr -> data
step 5: ptr = ptr -> next
(end of loop)

step 6: exit.

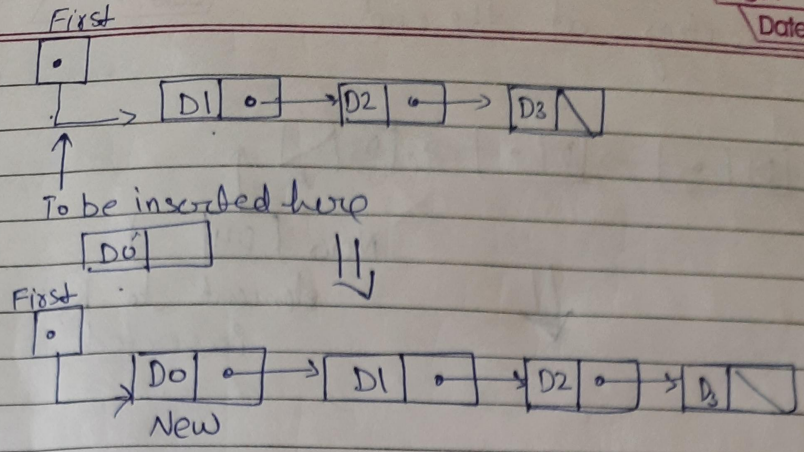
Code =>
```
#
#
struct node { int data;
              struct node* next;
            };
void display (struct node *head)
{ struct node *tmp;
  tmp = head;
  while(tmp != NULL)
  { printf("%d", tmp -> data);
    tmp = tmp -> next;
  }
}
void main() { struct node *prev, *
  int n,i;
```

```
printf(" enter the no. of elements");
scanf("%d", &n);
head = NULL;
for (i=0; i<n; i++)
{
p=(struct node*)malloc (sizeof (struct node));
scanf("%d", &p->data);
p->next = null;
if (head == =NULL)
{ head = P;
}
else { prev -> next = P; }
prev = P;
}
display (head);
}
```

**Insertion :=)** There are 3 different possibilities for inserting a node into a linked list.

→1. Inserting at the beginning of the list.
→2. Inserting at the end of the list
→3. Inserting a new node except the above mentioned position or we can say inbtw somewhere.

**Case i)** Insertion at the beginning.




Algorithm => 1) Make a new node & set the fields

NEW ← NODE
INFO(NEW) ← X

2) Add elements and reset first.

LINK(NEW) ← FIRST
FIRST ← NEW

3. RETURN.

Note

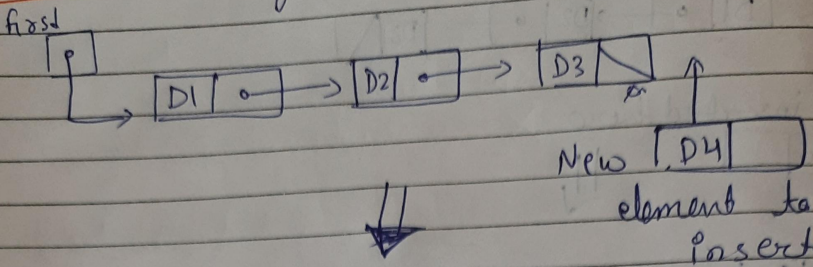
* FIRST => is the external pointer containing address of first node.
* NEW => A pointer containing address of NEW node.
* X => Data to be placed in new node.

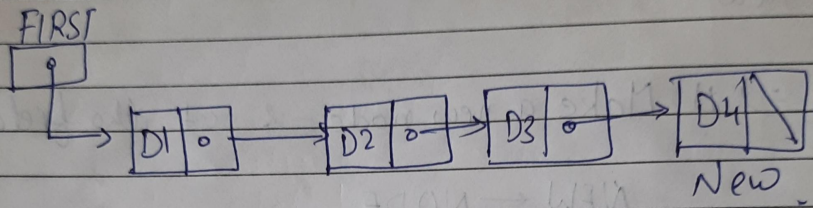
Code =>

```
void front( struct node *head, int value)
{ struct node *p;
  p = (struct node*) malloc(sizeof(struct node));
  p->data = value;
  p->next = head;
  head = p;
}
```

Subhash
Page :
Date :

## Case 2: Inserting at the end.




Algorithm:→ 1. Get a new Node & set the fields

```
New ← NODE
Info(New) ← X
Link(New) ← NULL
```

2. LOC ← FIRST.
3. // Pass all nodes & reach the last node.

```
while (LINK(LOC) != NULL) do
    LOC ← LINK(LOC)
end while.
```

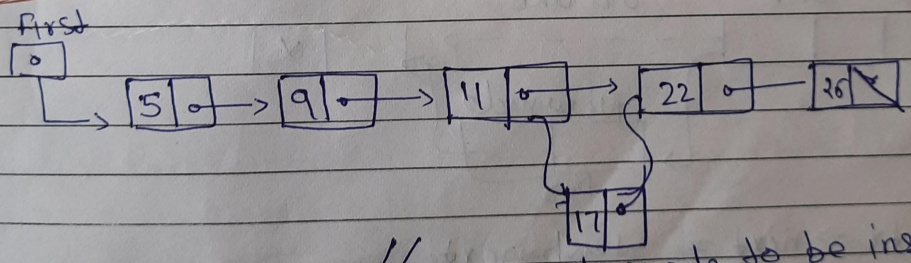
4. LINK (LOC) ← NEW // element Inserted.
5. Return.

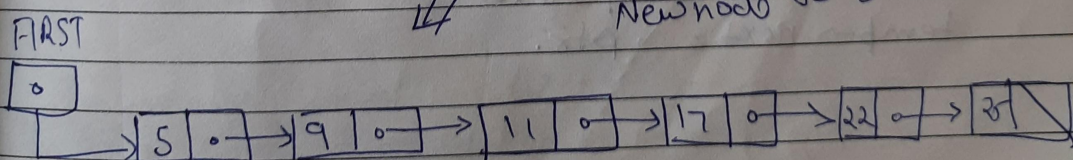
(*) LOC ⇒ Location of each node one by one.

Code :

```
void end(struct node *head, int value)
{
    struct node *p, *q;

p = (struct node *) malloc (sizeof(struct node));
  p->data = value;
  p->next = NULL;
    q = head;

  while (q->next != NULL)
      {
        q = q->next;

      }
      q->next = p;

}
```

* Case 3: Insertion at place.



Subhash®
Page :
Date :

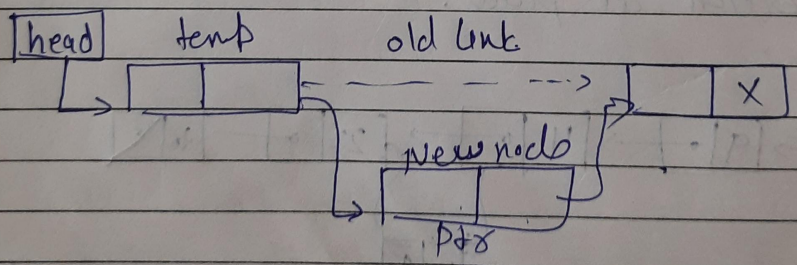
## Algorithm :-

Step1: Get new node & set fields
New ← ptr.
Info(New) ← X

2. Set temp = head
3) set i = 0;
Set 4) Repeat step 5 & 6 until loc. ptr = 1

5. temp = temp → next.
6) if temp = NULL
write desired node is not present
goto step 9.
end of if
end of loop.
step7: ptr → next = temp → next
temp → next = ptr.

step8 => set ptr = new
step 9: exit



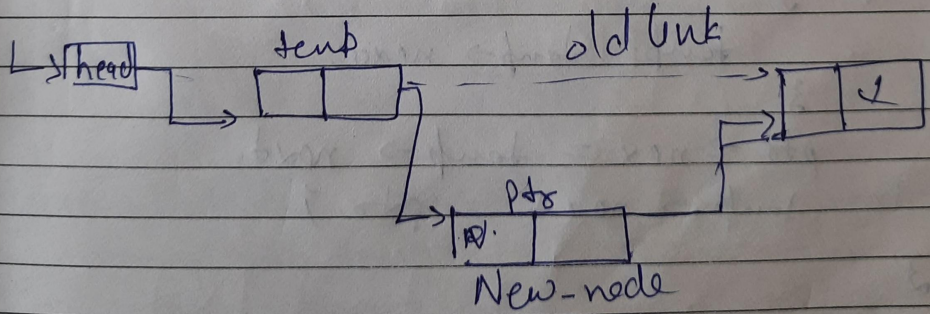

ptr → next = temp → next.
temp → next = ptr.

Subhash
Page :
Date :

# Algorithm: Inserting at nth node of list.

Step 1: ⇒ If ptr = NULL
write overflow
goto step
end of if

Step 2: set new_node = ptr.
step 3 new_node → data = data.
step 4: set temp = head
Step 5: i = 0 set
step 6: Repeat step 5 & 6 until i
step 7: temp = temp → next.
step 8: If temp = NULL
write desired node
is not found
goto step 12.
end of if
end of loop.
step 9: ptr → next = temp → next.
step 10: temp → next = ptr
Step 11: set ptr = new_node.
step 12: exit.




ptr → next = temp → next
temp → next = ptr.

Page :
Date :

Code

```
# include <stdio.h>
# include <conio.h>
struct node *head = NULL;
struct node {
        int data;
        struct node *next;
        };

void ins_at_pos_n(int data, int position)
    {
        struct node *ptr = (struct node*) malloc
                (sizeof(struct node));

    ptr -> data = data;
     int i;
    struct node *temp = head;
     if (position == 1)
        {   ptr -> next = temp;
                head = ptr
                return;
        }
    for (i=1; i< position; i++)
        {
            temp = temp -> next;
        }
        ptr -> nex = temp -> next.
        temp -> next = ptr;

}
void display ()
    {   struct node *temp = head;
        printf("\n List:");
```

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

int main ()
  { int i, n, pos, data
  struct node *temp, *prev, *head;
  printf(" \n Enter the no of node");
    scanf(" %d", &n)
      head = NULL;
printf(" Enter the elements");
  for (i=0; i<n; i++)
    {
  temp = (struct node *) malloc (size of (struct node));
    temp -> next = NULL;
      if (head = = NULL)
        { head = temp;
        }
      else { prev -> next = temp; }
          prev = temp;

  } // end of for

printf (" Enter the data you want to
    insert & position");

Scanf (" %d %d", &data, &pos);
    ins_at_pos_n(data, pos);
    display ();
    return 0; }.
```

Subhash
Page :
Date :

# Deletion operation :-

1) Delete at beginning.
2) Deletion at the end of the list.
3) Deletion after specified node.

(1) Deletion at beginning :-> It involves deletion of a node from the beginning of the list

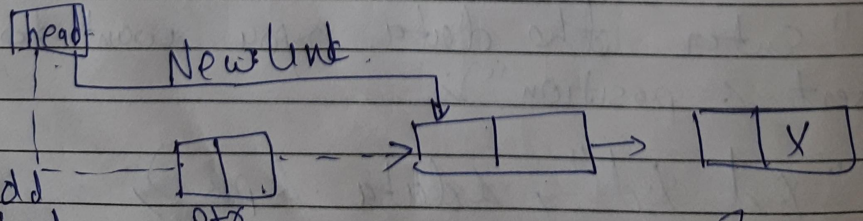
Logic =>
```
ptr = head
head = ptr -> next
free(ptr)
```

free the pointer ptr which were pointing to the head node of the list.

## Algorithm

Step1: if HEAD = NULL
write underflow
go to step 5
(end of if)

Step 2: Set ptr = Head.
step3: set head = ptr -> next
step 4 Free Ptr;
step5: EXIT.




```
ptr = head
head = ptr -> next
Free (ptr)
```

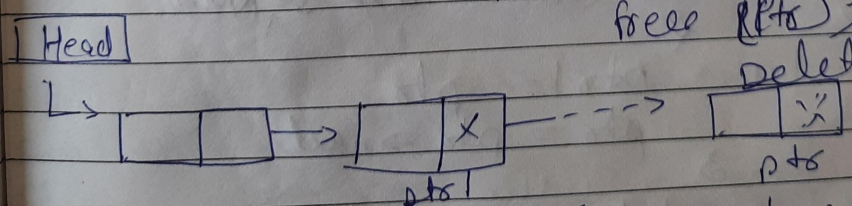
Subhash
Page :
Date :

Code
```
void begdelete()
{
  struct node *ptr;
  if(head == NULL)
  {
    printf("List is empty");
  }
  else {  ptr = head;
          head = ptr -> next;
          free(ptr);
       }
}
```

2 Deletion at the end of the List:=> It involves deletion of a node from the end of the List.

Logic

```
ptr = head
while ( ptr -> next != NULL)
  { ptr1 = ptr
    ptr = ptr -> next;
  }
ptr1 -> next = NULL;
free (Ptr);
```




ptr1 -> next = NULL
free(ptr)

Page :
Date :

Algorith

Step1: if head = NULL
write underflow
goto step 8
(end of if)

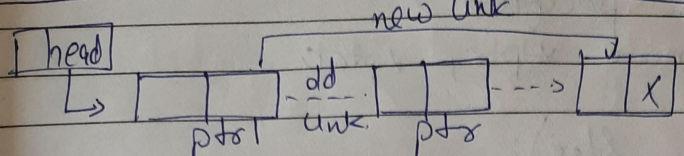
Step2: set ptr = head
Step3: ~~repet~~ repeat step 4 & 5 while ptr -> next != NULL
Step 4: set ptr1 = ptr
step 5: set ptr = ptr -> next
(end of loop)
step 6: set ptr1 -> next = NULL
step 7: free PTR.
step 8: exit.

## Code

```
void end_delete () { struct node *ptr, *p
    if(head ==NULL)
    { printf ("List is empty");
    }
    else if (head -> next == NULL)
    else
    { ptr = head
      while (ptr -> next != NULL)
      {
        ptr1 = ptr
        ptr = ptr -> next;
      }
      ptr1 -> next = NULL;
      free (ptr);
      printf(" Deleted node from li
    }
```

Subhash
Page :
Date :

## Deletion a node from specified position




ptr1 -> next = ptr -> next

free (ptr)

Algorithm : step1 : if head = NULL
write underflow
goto step 10
end of if.
step2: set temp = head
step3: set i = 0
step4: repeat step 5 to 8 until i
step5: temp1 = temp
step 6: temp = temp -> next
step7: if temp = NULL
write " desired node not present "
goto step 12
end of if
step8: i = i+1
end of loop.
step9: temp1 -> next = temp -> next
step 10: free temp
step 11: exit.

Date :

Code

```
void delete_specified()
{ struct node *ptr, *ptr1;
  int loc, i;
scanf("%d", &loc);
 ptr = head;
for(i=0; i<loc; i++)
   { ptr1 = ptr;
     ptr = ptr->next;
     if(ptr == NULL)
      { printf("There are less than %d elements in the list");
     return;
   }
}
ptr1->next = ptr->next
   free(ptr)
   printf("\n Deleted node is %d", loc);
}.
```

## Applications of Linked List:->

Linked lists are used in wide variety of applications because of their characteristics. They save wastage of memory. Some common application of Linked list are:->

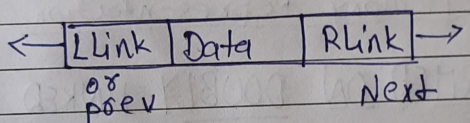
a) in computer Science:->

1.) Implementing Stack.
2) Implementing Queue.
3) Implementing non-Linear structure
4) Maintaining directories
5) Performing arithmatic operations on

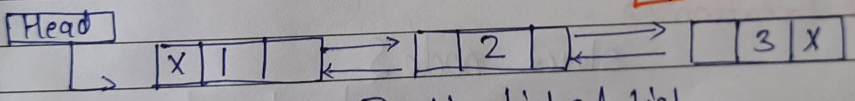
long integers.
6) Manipulating polynomials.
7) Representing sparse matrices etc.

b) in real world :=>
1) Image viewer.
2) previous & next page in web browser.
3) Music player etc.

#) Doubly Linked List :=> : Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence.
A node consist of 3 parts.




The Prev part of first node & the Next part of the last node will always contain NULL indicates end of each direction.



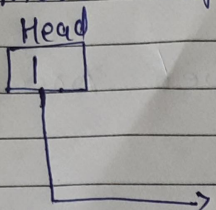

Doubly Linked List.

In C, structure doubly linked list can be given as :=>

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

## Memory Representation of a doubly Linked List =>

Generally, doubly linked list contain more space for every node & therefore causes more expansive basic operations.




| | Data | Prev | Next |
|---|---|---|---|
| 1 | 13 | -1 | 4 |
| 2 | | | |
| 3 | | | |
| 4 | 15 | 1 | 6 |
| 5 | | | |
| 6 | 19 | 4 | 8 |
| 7 | | | |
| 8 | 57 | 6 | -1 |

Memory Representation

## OPERATIONS PERFORMED ON DOUBLY LINKED LIST :=>

### 1) Node Creation :=>

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