LINKED LIST

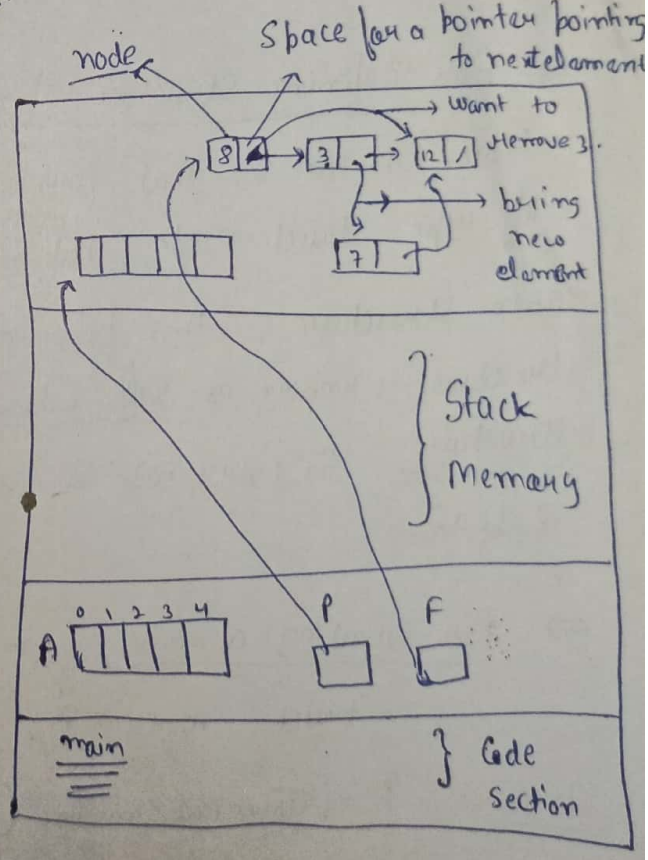
★ **Problem with Arrays** :- Arrays are of fixed size, this it main problem, bcz as we are programmers, we make Programs for our Users, so users will decide how many elements they will enter into our program, so At run time List size has to be decided, so, this is the problem with Arrays.

int A[100]; → fixed size.

int A[5];

In C++ :- int *P = new int [5] → fixed size

in heap



Now, we dont want Array bcz it has fixed size, we want a data structure of variable size, so that a user can input & remove as much elements he/she wants. For this, we must create memory in heap during run time & we will use Linking to make as a Linked list.

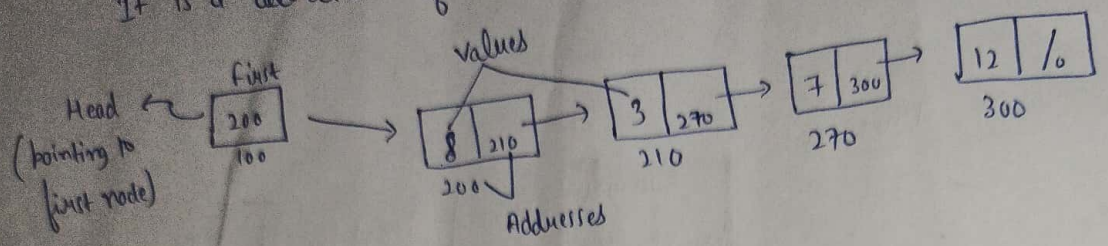
Array is like a bench of fixed size and Linked List is just like a chair, whoever wants to sit, bring a chair.

So, Linked List is more flexible than Array. We will Always create Linked List in heap. For each element, memory is differently Allocated.

Scanned with CamScanner

☆ About Linked List :-

It is a collection of nodes which contain data and pointer to next node

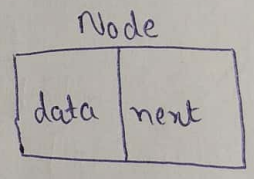


⇒ For Defining a Node, we have two things

- Data → int, float -- any type
- pointer → pointer is pointing to a node, so pointer is of node type.

⇒ For Defining structure of node,

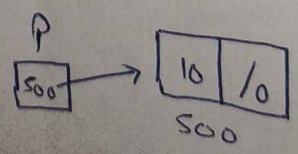
Here we can see that pointer is of type Struct Node, that is of own structure, this type of structure is known as Self-Referential Structure. So every node is consuming 8 bytes.




```
Struct Node → Size = 8 bytes
{ int data; → 4 bytes
  Struct Node * next; → 4 bytes
}
```

⇒ For Creating a Node, in heap

```
Struct Node *P;
```



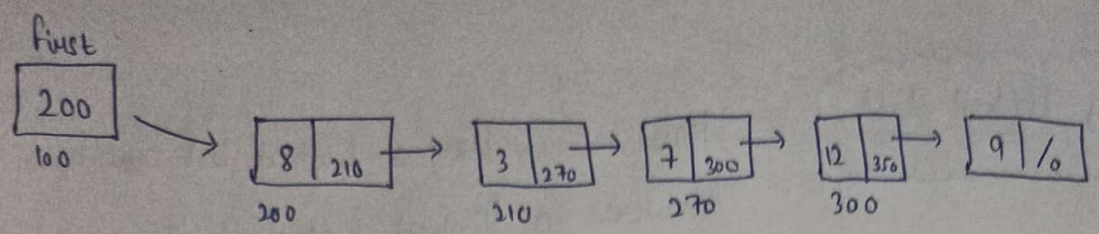
In C :- P = (struct Node *)malloc (size of (struct Node));

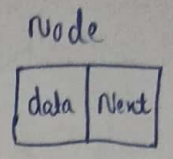
In C++ :- P = new Node;

⇒ For Accessing :- P → bcz P is a pointer

```
P → data = 10;
P → next = 0;
```

Scanned with CamScanner

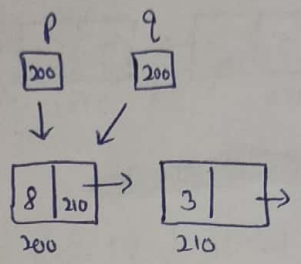




Struct Node *P, *q;

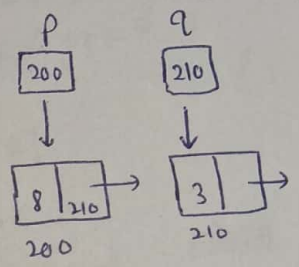
1.) P = q ;

q will also pointing on the Same Node



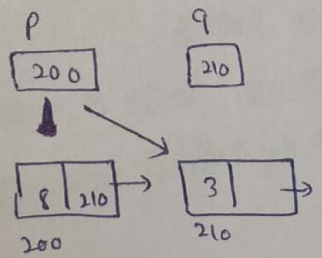
2.) q = P→next ;

Let q point on next Node of P



3.) P = P→next

P has move on to the next node, this is very useful Statement in Linked list



4.)

Struct Node *P = 0;

Struct Node *P = NULL;

Checking whether pointer is pointing to Any Node or not.

Conditions if pointer is Null.

if (P == NULL) → (A)

if (P == 0) → (A)

if (!P) → (A)

P = 0 [False]

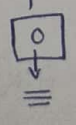
!P = True

if (P != NULL) → (B)

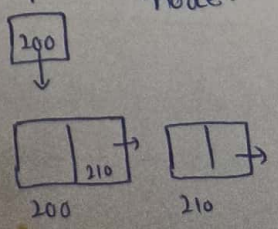
if (P != 0) → (B)

if (P) → (B)

(A) P → Not pointing to Any node.



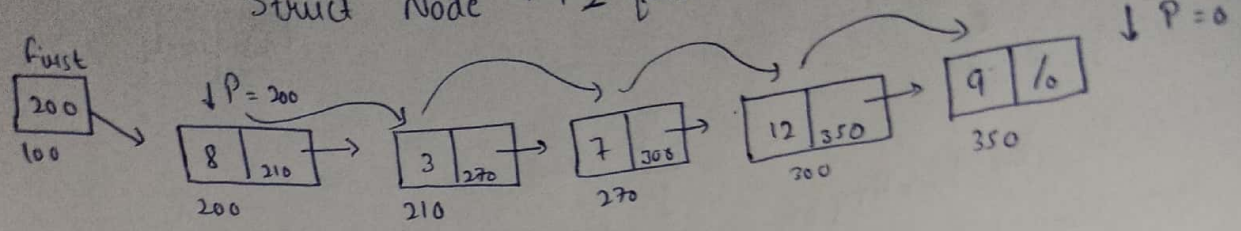
(B) P → pointing to Some node.



Scanned with CamScanner

★ Display Linked List :- How to display a linked list or how to traverse a linked list (i.e. Traversing)

Struct Node *P = first;




```
while ( P! =0)
{ printf ("%d", P→data);
  P = P→next;
}
```

```
while (P! = NULL)
{ printf ("%d", P→data
  P = P→next;
}
```

Code :-

```
Struct Node
{
  int data;
  Struct Node * ne
} *first = NULL;
```

```
int main ()
{
  int N, i;
  printf (" Enter Value of N: \n");
  Scanf ("%d", & N);
  int A[N];
  for (i= 0 ; i< N; i++)
  {
    printf (" Elements are : \n");
    Scanf ("%d", & A[i]);
  }
  Void Create (int A[], int n);
  Create (A, N);
  Void Display (struct Node * P);
  Display (first);
}
```

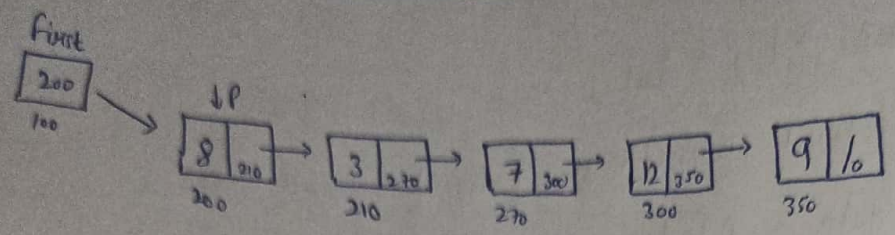
Scanned with CamScanner

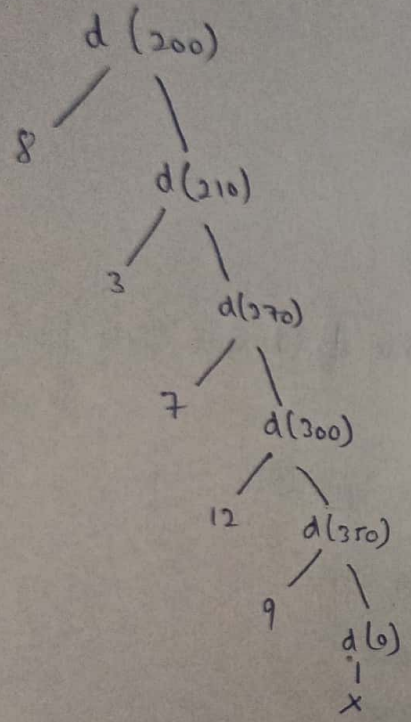
```
void create (int A[], int n)
{
    int i;
    struct Node *t, * last;
    first = (struct Node *) malloc (sizeof (struct Node));
    first -> data = A[0];
    first -> next = NULL;
    last = first;
    for (i=1 ; i<n ; i++)
    {
        t = (struct Node *) malloc (sizeof (struct Node));
        t -> data = A[i];
        t -> next = NULL;
        last -> next = t;
        last = t;
    }
}
void Display (struct Node *p) || P will become "first" Pointer.
{
    while (P != NULL)
    {
        printf ("%d -", P -> data);
        P = P -> next;
    }
}
```

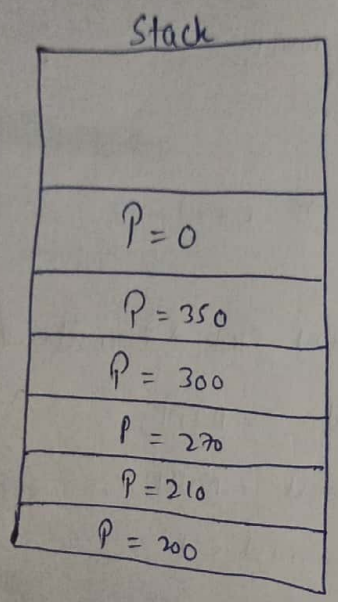
$\boxed{\text{Time Complexity} = O(n)}$

Scanned with CamScanner

⟹ Recursive Display linked list :-








```
void Display (struct Node *p)
{
    if (P! = NULL)
    {
        printf ("%d _", P→data
        Display (P→next);
    }
}
```

Display (first)

Time Complexity :- O(n)

bcz it prints only.

So, Loop & recursive function both have O(n) Time Complexity.

Space Complexity :- O(n) :- bcz (n+1) activation record forms.

Scanned with CamScanner

Code :-

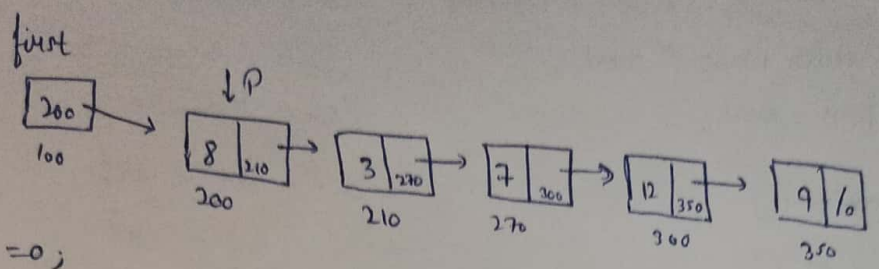
```
Struct Node
{
    int data;
    Struct Node * next;
} * first = NULL;
int main ( )
{
    int N, i ;
    printf (" Enter the Value of N: \n");
    Scanf ("%d", &N);
    int A[N];
    for (i = 0; i < N; i++)
    {
      printf (" Elements are : \n");
      Scanf ("%d", &A[i]);
    }
    void Create (int A[], int N);
      Create (A, N);
    Void Display (struct Node *P);
      Display (first);
}
void Create (int A[], int N)
{
    int i;
    Struct Node *t, * last;
    first = (struct Node *) malloc (sizeof (struct Node));
    first -> data = A[0];
    first -> next = NULL;
    last = first;
```

Scanned with CamScanner

```
for (i=0; i<n; i++)
  {
    t = (struct node *) malloc (size of (struct Node));
    t->data = A[i];
    t->next = NULL;
    last->next = t;
    last = t;
  }
}

Void Display (struct Node *p)
{
    if (p != NULL)
    {
      printf ("%d _", p->data);
      Display (p->next);
    }
}
```

Scanned with CamScanner

Counting No. of Nodes in a Linked List :-



Count = 0;
1+1+1+1 - -

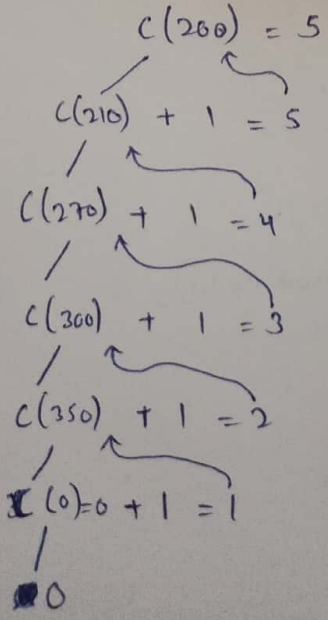
```
int Count (struct node * p)
{ int c = 0;
  while (P != 0)
  { c++;
    P = P -> net
  }
  return c;
}
```

Time Complexity :- $O(n)$

Space Complexity :- $O(1)$ → Constant

Recursively :-

```
int Count1 (struct node * p)
{ if (P == 0)
    return 0;
  else
    return Count (P->next) + 1;
}
```



Time Complexity :- $O(n)$

Space Complexity :- $O(n)$ → uses Stack

Scanned with CamScanner

Code :-

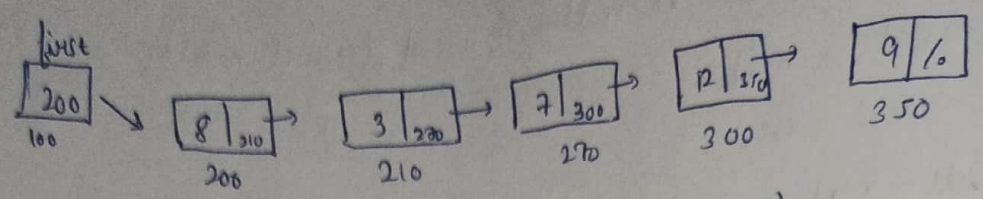
```
Struct Node
{
  int data;
  Struct Node * next;
} * first = NULL;

int main ()
{
  int N, i;
  printf (" Enter the value of N : \n");
  scanf ("%d", &N);
  int A[N];
  printf (" Elements are : \n");
  for (i=0 ; i<N ; i++)
    scanf ("%d", &A[i]);
  Void Create (int A[], int N);
  int Count (struct Node * P);
  int Count1 (struct Node * P);
  int c = Count(first);
  int m = Count1 (first);
  printf (" Total no. of Nodes : %d", c);
  printf (" Total no. of Nodes : %d", m);
}

Void Create ( int A[], int N)
{
  int i;
  struct Node *t, * last;
  first = (struct Node *) mallac (size of (struct Node));
  first -> data = A[0];
  first -> next = NULL;
  last = first;
```

Scanned with CamScanner

```
for (i=1; i<N; i++)
{
    t = (struct Node *) malloc (size of (struct node));
    t->data = A[i];
    t->next = NULL;
    last->next = t;
    last = t;
  }
}
int Count (struct Node *p)    || Recursively
{   if (p != NULL)
        return  Count (p->next) + 1;
    else
        return 0;
}

int Count1 (struct Node *p)
{   int c=0;
    while (p != NULL)
    {   c++;
        p = p->next;
    }
    return c;
}
```

Scanned with CamScanner

★ Sum of All Elements in a Linked List :-




```
int Add (struct Node *P)
{ int Sum = 0
  while (P)
  {
    Sum = Sum + P→data;
    P = P→next;
  }
  return Sum;
}
```

Time Complexity :- O(n)

- Recursively :-

```
int Add (struct Node *P)
{
  if (P == 0)
    return 0;
  else
    return Add (P→next) + P→data;
}
```

Time Complexity :- O(n)

Code :-

```
Struct Node
{
  int data;
  Struct Node * next;
} * first = NULL;
```

Scanned with CamScanner

```
int main ()
{
    int N, i;
    printf(" Enter no. of terms : \n");
    scanf ("%d", & N);
    int A[N];
    for (i=0; i<N; i++)
      scanf ("%d", & A[i]);
    void create (int A[], int N);
      create (A, N);
    int Sum1 (struct Node *p);
    int Sum2 (struct Node *p);
    int c = Sum1 (first);
    int m = Sum2 (first);
    printf(" Sum : %d \n", c);
    printf (" Sum : %d \n", m);
}

Void create (int A[], int N)
  {
    int i;
    Struct Node *t, * last;
    first = (struct Node *) malloc (size of (struct Node));
    first -> data = A[0];
    first -> next = NULL;
    last = first;
    for (i=1; i<N; i++)
      {
        t = (struct Node *) malloc (size of (struct node));
        t -> data = A[i];
        t -> next = NULL;
        last -> next = t;
        last = t;
  }
```

Scanned with CamScanner

```
int Sum1 (struct Node * p)
{
    int Sum = 0;
    while (P != NULL)
    {
        Sum = Sum + P -> data;
        P = P -> next;
    }
    return Sum;
}

int Sum2 (struct Node * p)   // Recursive function
{
    if (P == 0)
    return 0;
    else
    return Sum2 (P -> next) + P -> data;
}
```

Scanned with CamScanner

Maximum Element in a Linked List :-

```
Struct Node
{
   int data;
   Struct Node * next;
}* first = NULL;

int main ()
{
   int N, i;
   printf (" Enter the no. of terms : \n");
   Scanf (" %d", &n);
   int A [N];
   printf (" Elements are : \n");
   for (i = 0 ; i < N; i++)
      Scanf (" %d", & A[i]);
   Void create (int A[], int N);
   Create (A, N);
   int max ( Struct Node * P);
   int c = max (first);
   printf (" maximum Element : %d", c);
   int maxi (struct Node * P);
   int m = maxi (first);
   printf (" maximum Element : %d ", m);

}
```

Scanned with CamScanner

```
void Create (int A[], int n)
{
    int i;
    struct Node *t, *last;
    first = (struct Node *) malloc (sizeof (struct Node));
    first -> data = A[0];
    first -> next = NULL;
    last = first;
    for (i = 1; i < n; i++)
    {
        t = (struct Node *) malloc (sizeof (struct Node));
        t -> data = A[i];
        t -> next = NULL;
        last -> next = t;
        last = t;
    }
}

int max (struct Node *p)
{
    int max = P -> data;
    while (P != NULL)
    {
        if (max < P -> data)
            max = P -> data;
        P = P -> next;
    }
    return max;
}
```

Scanned with CamScanner

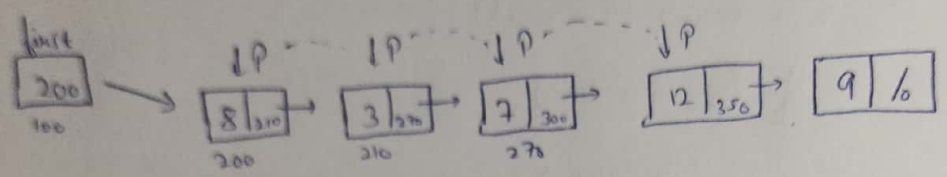
```
int max1 (struct Node *p)
{
    int x=0;
    Static int max = -34526;
    if (P==0)
        return max;
    else
    {
        x = max1 (P->next);
        if (x> P->data)
            return x;
        else
            return P->data;
    }
}
```

Scanned with CamScanner

★ Searching in a List :-

1. Linear Search ✓
2. Binary Search ✗

In a linked list, we cannot Perform Binary Search here, bcz in binary Search we directly go to the middle Element, but in linked list, we have to traversing, so binary Search is not possible. So we will perform only linear Search.




```
struct Node * Search ( struct Node *P, int key)
{
    while ( P != NULL)
    {
        if (key == P-> data)
            return P ;
        P = P-> next;
    }
    return NULL;
}
```

Recursively :-

```
struct Node * Search1 ( struct Node *P, int key)
{
    if (P == NULL)
        return NULL;
    if (key == P-> data)
        return P;
    return Search1 (P-> next, key);
}
```

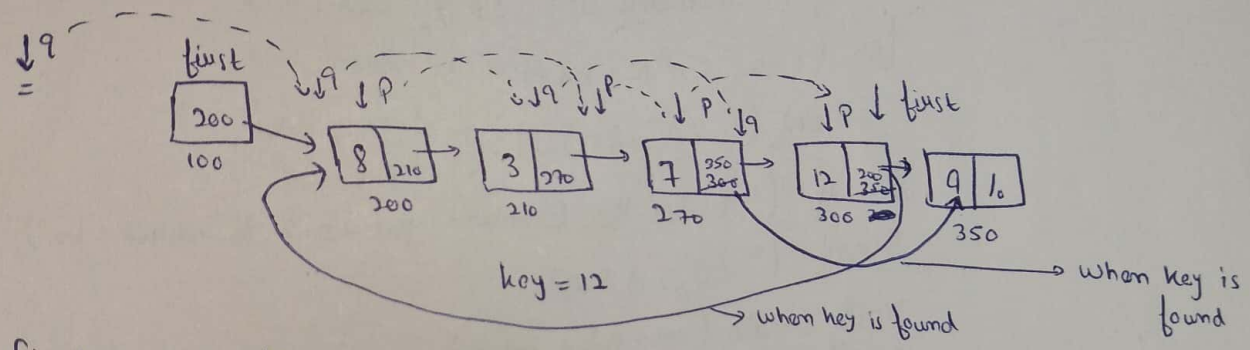
Scanned with CamScanner

⇒ Improving in Linear Searching :-

Two Methods :- we already discussed in Arrays.

1. Transposition. ✗
2. Move to Head. ✓

We will not do Transposition, bcz we dont want to change the values of two nodes, we will shift the required node to the first position So that it will Search in once, when we Search again. So we will Do Move to head.



First, we will take a pointer q which is NULL & it follows P

```
struct Node* Search (struct Node *P, int key)
{
    Node *q = NULL;
    while (P != NULL)
        if (key == P->data)
        {
            q->next = P->next;
            P->next = first;
            first = P;
        }
        q = P;
        P = P->next;
```

After Improving Searching :-

Scanned with CamScanner

Code :-

```
Struct Node
{
    int data;
    struct node * next;
} * first = NULL;

int main ()
{
    int N, i, key;
    Struct Node *temp, * prince;
    printf ("Enter the no. of terms : \n");
    Scanf ("%d", & N);
    int A[N];
    printf ("Elements are : \n");
    for (i=0; i< N; i++)
        Scanf (" %d", & A[i]);
    printf (" Enter the Element you want to Search : \n");
    Scanf (" %d", & key);
    Void create (int A[], int N);
    create (A, N);
    void Search1 (struct Node * P, int key);
    Search1 (first, key);
    Struct Node * Search2 (struct Node * P, int key);
    temp = Search2 (first, key);
    if (temp != NULL)
        printf (" key %d is found at %d location \n", temp->data, temp);
    Struct Node * Search 3 ( struct Node * P, int key);
    Prince = Search 3 (first, key);
    if (Prince != NULL)
    printf (" key %d is found at %d location \n", Prince -> da
}
```

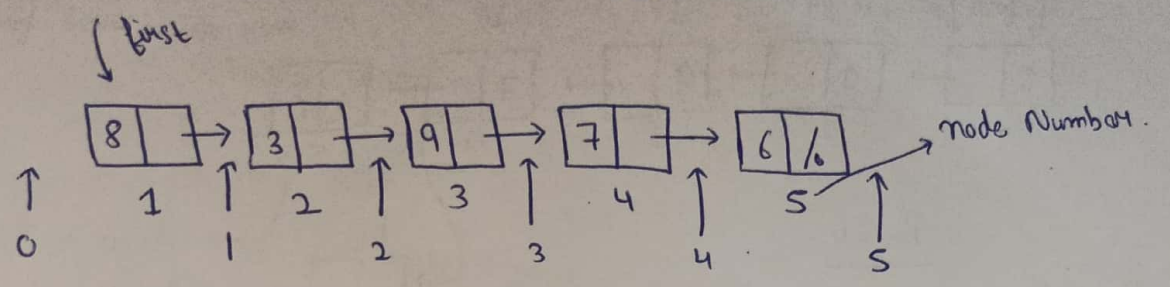
```
Void create (int A[], int N)
{
    int i;
    Struct Node * t, * last;
    first = (struct Node *) malloc (size of (struct Node));
    first -> data = A[0];
    first -> next = NULL;
    last = first;
    for (i=1 ; i<N; i++)
    {
        t = (struct Node *) malloc (size of (struct Node));
        t -> data = A[i];
        t -> next = NULL;
        last -> next = t;
        last = t;
    }
}
Void Search1 (Struct Node *P, int key)   || If a key is present at
                                            more than one location.
{
    while (P! = NULL)
    {
        if (key = = P-> data)
        printf (" key %d is found at %d location \n", P-> data, P);
        P = P-> next;
    }
}
```

Scanned with CamScanner

```
Struct Node * Search2 (struct Node * P, int key)   // Recursively
{
    if (P==NULL)
        Return NULL;
    if (key == P->data)
        Return P;
    Return Search2 (P->next, key);
}

Struct Node * Search3 (Struct Node *P, int key)   // Improving
{
    Struct Node *q = NULL;
    while (P != NULL)
    {
        if (key == P->data)
        {
            q->next = P->next;
            P->next = first;
            first = P;
            Return first;
        }
        q=P;
        P = P->next;
    }
}
```

Scanned with CamScanner

Inserting in a Linked List :- Introduce a new node in a linked list.

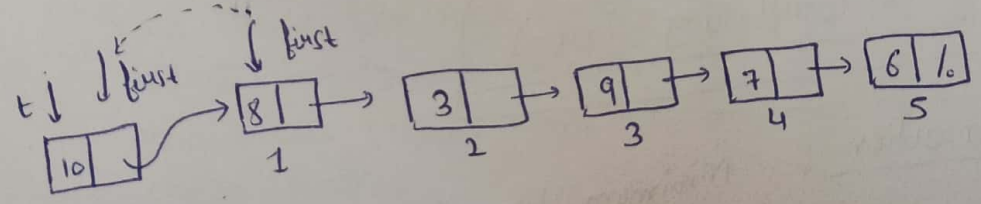


- If we say, we want to insert at before 1st node then index is 0.
- If we say, we want to insert after 3rd node, then index is 3.

Two Cases of Insertion :-
1. Insertion before first
2. Inserting after given position.

⇒ Case 1 :- Inserting a node before first :-

(i) Creating a New node
(ii) Then insert data
(iii) Then point this to first and move first to that node.



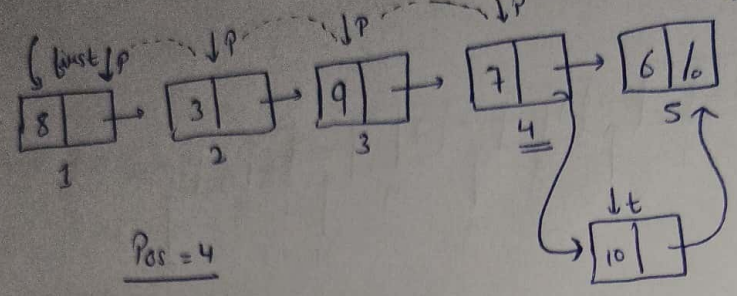

```
Struct Node *t = new Node;
t → data = x;
t → next = first;
first = t;
```

Time Complexity :- O(1)

Constant time.

Complete

⇒ Case - 2 :- Insert a New Node after Given position -



→ These are known as pointers they are used for Access

→ it is a link, bcz pointers used for linking are called as link.

Pos = 4

✓ • P should move for (Pos - 1) times.

```
Struct Node *t = new Node;
    t → data = x;
    P = first;
    for (i = 0 ; i < Pos-1 ; i++)
    {
        P = P → next;
    }
    t → next = P → next;
    P → next = t;
```

Time Complexity :- O(n)

For Insertion :- Minimum time taken :- O(1) → when we insert after 1st node.

Maximum time taken :- O(n) → when we insert after last node.

Scanned with CamScanner

Complete Code for Insertion by Combining Case 1 & Case 2 :-

```
Struct Node
{
  int data;
  Struct Node * next;
} * first = NULL;

int main ()
{
  int N, i;
  printf (" Enter the no. of terms : \n");
  Scanf ("%d", &N);
  int A[N];
  printf (" Elements are : \n");
  for (i=0 ; i<N ; i++)
   {  Scanf ("%d", &A[i]);
   }
  Void Create (int A[], int N);
  Void Display (struct Node *P);
  Void Insert (struct Node *P, int index, int x);
  Create (A,N);
  Display (first);
  printf (" \n After Insertion \n");
  Insert (first, 0, 10);
  Display (first);

}
```

Scanned with CamScanner

```
Void Create ( int A[], int n)
{
    int i;
    Struct Node *t, *last;
    first = (Struct Node *) malloc (size of (Struct Node));
    first -> data = A[0];
    first -> next = NULL;
    last = first;
    for(i=1; i<N; i++)
    {
        t = (Struct Node *) malloc (size of (Struct Node));
        t -> data = A[i];
        t -> next = NULL;
        last -> next = t;
        last = t;
    }
}

Void Display ( Struct Node *p)
{
    while (P != NULL)
    {
        printf ("%d _", P -> data);
        P = P -> next;
    }
}
```

Scanned with CamScanner

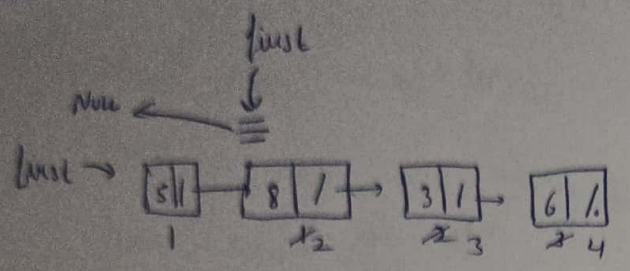
```
int Count (struct Node *p)
{ if (P != NULL)
      return Count (P->next) + 1;
  else
      return 0;
}

Void Insert (struct Node * P, int index, int x)
{
    struct Node *t;
    int i;
    if (index < 0 || index > Count (P))
          return; || Returning Nothing
    else
    {
        t = (struct Node *) malloc (sizeof (struct Node));
        t -> data = x;
        if (index == 0)
        {
            t -> next = first;
            first = t;
        }
        else
        {
            for (i = 0 ; i < index -1 && P ; i++)
            {
                P = P -> next;
            }
            t -> next = P -> next;
            P -> next = t;
        }
    }
}
```

Here P means that pointer pointing to some node (i.e Not NULL)

Scanned with CamScanner

☆ Creating a linked list using Insert :-

In this, We will not directly created linked list using create function.




```
Insert (0, 8);
Insert (1, 3);
Insert (2, 6);
Insert (0, 5);
```

Code :-

```
Struct Node
{
    int data;
    Struct Node * next;
}* first = NULL;
int main ()
{
    Void Display (Struct Node * p);
    Void Insert (Struct Node * p, int index, int x);
    printf (" After Insertion \n");
    Insert (first, 0, 10);
    Insert (first, 1, 5);
    Insert (first, 2, 14);
    Insert (first, 2, 9);
    Display (first);
}
```

Scanned with CamScanner

```
Void Display (struct Node *P)
  {
    while (P!=NULL)
      {
        printf ("%d_", P->data);
        P = P->next;
      }
  }

int Count (struct Node * P)
  {  if (P!=NULL)
       return Count (P->next) + 1;
     else
       return 0;
  }

Void Insert (struct Node *P, int index, int x)
  {
    Struct Node *t;
    int i;
    if (index < 0 || index > Count (P))
          return;
    t = (struct Node *) malloc (sizeof (struct Node));
    t->data = x;
    if (index ==0)
      {  t -> next = first;
         first = t;
      }
    else
      {  for (i=0 ; i< index -1 && P ; i++)
           {  P = P->next;
           }
         t->next = P->next;
         P->next = t;
      }
  }
```

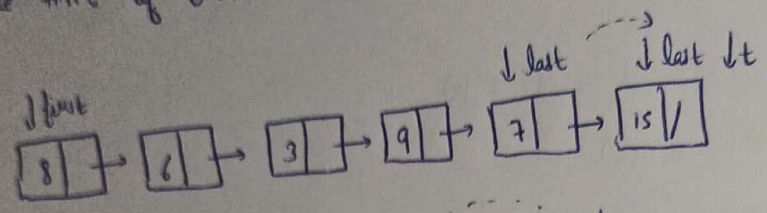
Scanned with CamScanner

★ Creating a linked list using insert at last

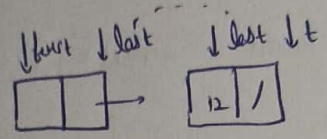
:- Always we want insert at last. repeat

Code

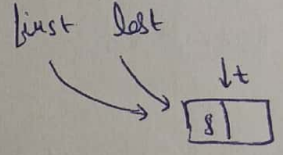
We know for inserting at last, we have to traverse through a list and it will take time of $O(n)$ but here we have different approach which will take time of $O(1)$.



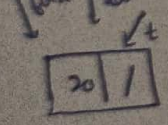
If there is only one node :-



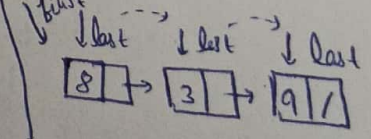
If there is no node present :- i.e. first = last = NULL then




```
Void Insert last ( int x)        → it will take a value
{
    Node *t = new Node;
      t → data = x;
      t → next = NULL;
      if (first == NULL)
          first = last = t;
      else
      {
          last→next = t;
          last = t;
      }
}
```



In Sert last (8)
Insert last (3)
Insert last (9)



This method is only used to insert at last, despite of index bcz it always insert at last and if the list is empty is will also work.

Scanned with CamScanner

Code :-

```
struct Node
{
    int data;
    Struct Node * next;
} * first = NULL;
int main ()
{
  int N, i;
  printf ("Enter the no. of terms : \n");
  Scanf ("%d", &N);
  int A[N];
  printf (" Elements are : \n");
  for (i=0; i< N, i++)
     Scanf ("%d", &A[i]);
  Void Create (int A[], int N);
  Void Display (Struct Node *P);
  Void Insert (Struct Node *P, int x);
  Create (A, N);
  Display (first);
  printf ("\n After Insertion \n");
  Insert (first, 98);
  Display (first);
}

Void Display ( Struct Node *P)
{
    while (P! = NULL)
      {
        printf ("%d", P→data);
        P=P→next;
      }
}
```

Scanned with CamScanner

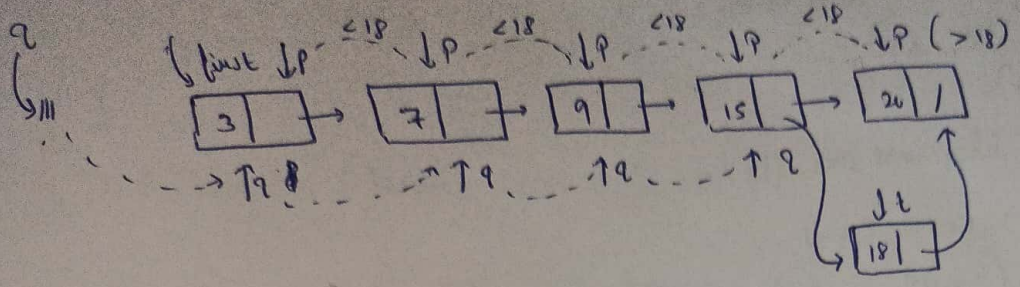
```
Void create (int A[], int N)
{
    int i;
    Struct Node *t, *last;
    first = (struct Node *) malloc (sizeof (struct Node));
    first -> data = A[0];
    first -> next = NULL;
    last = first;
    for (i=1; i<N; i++)
    {
        t = (Struct Node *) malloc (size of (Struct Node));
        t -> data = A[i];
        t -> next = NULL;
        last -> next = t;
        last = t;
    }
}
Void Insert ( Struct Node *P, int x)
{
    Struct Node *t, *last;
    t = (Struct Node *) malloc (size of (Struct Node));
    t -> data = x;
    t -> next = NULL;
    if (P == NULL)
        P = last = t;
    else
    {
        last = P; // last should be pointing an last Node to insert
        while (last -> next)
            last = last -> next;
        last -> next = t;
    }
}
```

a New node at the last.

Scanned with CamScanner

Inserting in a Sorted List :-

q → tailing Pointer




```
P = first;
q = NULL;
while (P && P→data < x)
{
    q = P;
    P = P→next;
}
t = new Node;
t→data = x;
t→next = q→next;
q→next = t;
```

Traversing ←

So, Time Complexity :- O(n)

Code :-

```
Struct Node
{
    int data;
    Struct Node * next;
} * first = NULL;
```

Scanned with CamScanner

```
int main ()
{
    int N, i;
    printf ("Enter no. of terms : \n");
    scanf ("%d", &N);
    int A[N];
    printf ("Elements are : \n");
    for (i=0; i<N; i++)
        scanf ("%d", &A[i]);
    void create (int A[], int n);
    void Display (struct Node * P);
    void Insert (struct Node *P, int x);
    create (A, N);
    Display (first);
    printf ("In After Insertion \n");
    Insert (first, 45);
    Display (first);
}

Void create (int A[], int N)
{
    int i;
    struct Node *t, last;
    first = (struct Node*) malloc (size of (struct Node));
    first->data = A[0];
    first -> next = NULL;
    last = first;
    for (i=1; i<N; i++)
    {
        t = (struct Node *) malloc (size of (struct Node));
        t -> data = A[i];
        t -> next = NULL;
        last -> next = t;
        last = t;
    }
}
```

Scanned with CamScanner

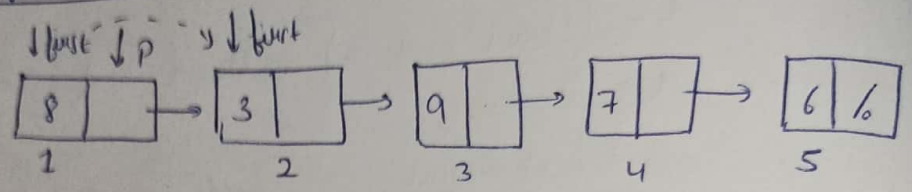
```
Void Display (struct Node *P)
{
  while (P != NULL)
   {
     printf ("%d _", P->data);
     P = P->next;
   }
}

Void Insert (struct Node *P, int x)
{
   struct Node *t, *q = NULL;
     t = (struct Node *) malloc (sizeof (struct Node));
     t->data = x;
     t->next = NULL;
     if (first == NULL)  // If list is Empty
          first = t;
     else
      { while (P && P->data < x)
          {  q = P;
             P = P->next;
          }
          if (P == first)  // If inserted Node is less than first Node
                           // data, then put it on left side of first
          {  t->next = first;  // Node
             first = t;
          }
          else
          {
             t->next = q->next;
             q->next = t;
          }
      }
}
```

Scanned with CamScanner

★ Deleting from Linked List :- Deleting a particular Node from combine

Two Cases :- ① Delete first Node
② Delete Node at Given position

Case-1 :- Deleting first Node :-

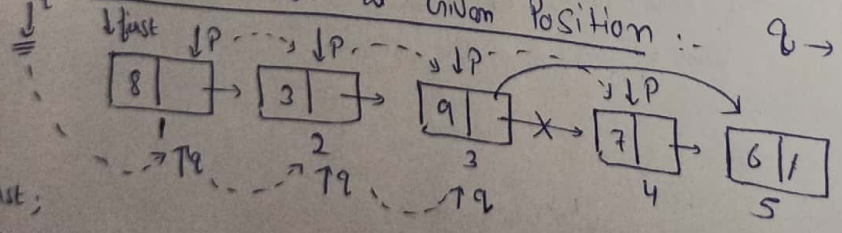


Note :- While deleting a Node, we have de-allocate it also, so that it will not occupy memory space. (i.e. Physically from Heap)

```
Node *P = first;
first = first -> next;
x = P -> data ; || Take out the value for further use & de allocation done.
delete P;
```

Time Complexity :- O(1)

Case-2 :- Deleting Node at Given Position :- q → Tail Pointer




```
Node *P = first;
Node *q = NULL;
for (i = 0; i < Pos-1; i++)
  {
    q = P;
    P = P -> next;
  }
q -> next = P -> next;
x = P -> data ; || Take out data
delete P;
```

Time Complexity :-
minimum → O(1)
maximum → O(n)

Scanned with CamScanner

Combined Code of Deletion by Case 1 & Case 2 :-

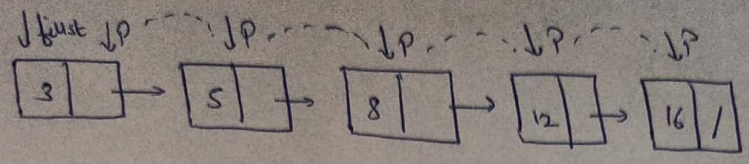
```
Struct Node
{
    int data;
    Struct Node * next;
} * first = NULL;
int main ()
{
    int N, i;
    printf (" Enter the No. of terms : \n");
    scanf ("%d", &N);
    int A[N];
    printf (" Elements are : \n");
    for (i = 0 ; i<N; i++)
      { scanf ("%d", &A[i]);
      }
    Void create ( int A[], int N);
    Void Display ( Struct Node *P);
    int delete (struct Node *P, int index);
    create (A,N);
    Display (first);
    printf (" After Deletion \n");
    printf (" Deleted Element : %d \n", Delete (first, 4));
    Display (first);
}

Void Display (Struct Node * P)
{
    while (P! = NULL)
      { printf (" %d _", P→data);
        P = P→next;
      }
}
```

Scanned with CamScanner

```
void create (int A[], int N)
{
    int i;
    struct Node *t, *last;
    first = (struct Node *) malloc (sizeof (struct Node));
    first->data = A[0];
    first->next = NULL;
    last = first;
    for (i=1; i<N; i++)
    {
        t = (struct Node *) malloc (sizeof (struct Node));
        t->data = A[i];
        t->next = NULL;
        last->next = t;
        last = t;
    }
}

int delete (struct Node *P, int index)
{
    struct Node *q = NULL;
    int x = -1, i;
    if (index == 1)
    {
        q = first;
        x = first->data;
        first = first->next;
        free (q);
        return x;
    }
    else
    {
        for (i=0; i< index-1 && P; i++)
        {
            q = P;
            P = P->next;
        }
        q->next = P->next;
        x = P->data;
        free (P);
        return x;
    }
}
```

Scanned with CamScanner

Checking the List is Sorted or not :-



$x = -32768$

~~3~~

~~5~~

~~8~~

~~12~~

16

```
int x = -32768;
Node *P = first;
while (P! =NULL)
  {
    if (P->data< x)
        return false; || List is not Sorted
    else
    {
        x = P->data;
        P = P->next;
    }
    return true;
```

Time Complexity :-

Max. time :- $O(n) \rightarrow$ Sorted

Min. time :- $O(1) \rightarrow$ Not Sorted

Code :-

```
Struct Node
{
    int data;
    Struct Node * next;
} * first = NULL;
```

Scanned with CamScanner

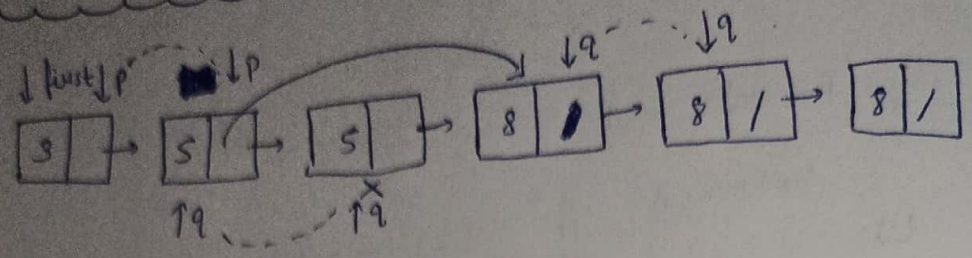
```
int main()
{
    int N, i;
    printf ("Enter the no. of terms : \n");
    Scanf ("%d", &N);
    int A[N];
    printf (" Elements are : \n");
    for (i=0 ; i< N; i++)
      Scanf ("%d", &A[i]);
void Create (int A[], int n);
int is Sorted (struct Node *P);
Create (A, N);
int m = issorted (first);
printf ("%d", m);
}

void create (int A[], int N)
{
    int i;
    Struct Node *t, * last;
    first = (struct Node *) malloc (size of (struct Node));
    first → data = A[0];
    first → next = NULL;
    last = first;
    for (i = 1; i < n ; i ++)
    {
        t = (struct Node *) malloc (size of (struct Node));
        t → data = A[i];
        t → next = NULL;
        last → next = t;
        last = t;
    }
}
```

Scanned with CamScanner

```
int isSorted (struct Node * P)
{ int x = -65536 ;   || minimum value of 4 Byte integer
    while (P != NULL)
    {
        if (P -> data < x)
            Return 0;   || List is not Sorted
        else
        {
            x = P -> data;
            P = P -> next;
        }
    }
    Return 1;   || List is Sorted.
}
```

Scanned with CamScanner

☆ Remove Duplicated from List :- From Sorted Linked List




```
Node *P = first ;
Node *q = first -> next ;
while (q != NULL)
    {
        if (P-> data != q -> data)
        {
            P = q ;
            q = q -> next ;
        }
        else
        {P -> next = q -> next ;
         delete q ;
         q = P -> next ;
        }
    }
```

Time Complexity :- O(n)

Code :-

```
Struct Node
{
    int data
    Struct Node * next ;
} * first = NULL ;
```

Scanned with CamScanner

```
int main ()
{
    int N, i;
    printf (" Enter the No. of terms : \n");
    scanf (" %d", &N);
    int A[N];
    printf (" Elements are : \n");
    for (i=0 ; i<N ; i++)
      scanf (" %d", &A[i]);
    Void Create (int A[], int N);
    Void Display ( Struct Node *P);
    Void Duplicate (Struct Node *P);
    Create (A, N);
    Display (first);
    printf (" \n After Removal \n");
    Duplicate (first);
    Display (first);

}

Void Display (Struct Node * P)
  {
    while (P! = NULL)
      {
        printf (" %d _", P->data);
        P = P -> next;
      }
  }
```

Scanned with CamScanner

```
void create (int A[], int N)
{
    int i;
    Struct Node *t, *last;
    first = (struct Node *) malloc (size of (struct Node));
    first -> data = A[0];
    first -> next = NULL;
    last = first;
    for (i = 1; i < N; i++)
    {
        t = (struct Node *) malloc (size of (struct Node));
        t -> data = A[i];
        t -> next = NULL;
        last -> next = t;
        last = t;
    }
}

Void Duplicate (Struct Node *P)
{
    [redacted] Struct Node *q = first -> next;
    while (q != NULL)
    {
        if (P -> data != q -> data)
        {
            P = q;
            q = q -> next;
        }
        else
        {
            P -> next = q -> next;
            free (q);
            q = P -> next;
        }
    }
}
```

Reversing

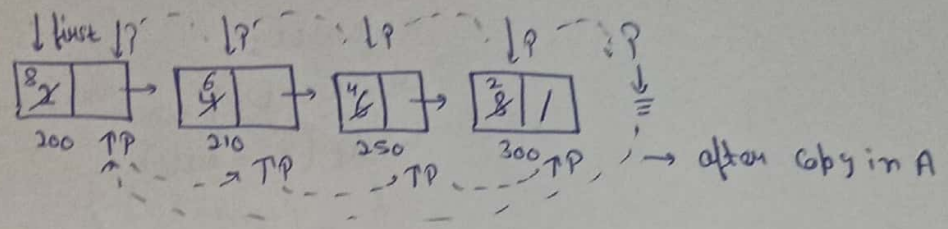
Scanned with CamScanner

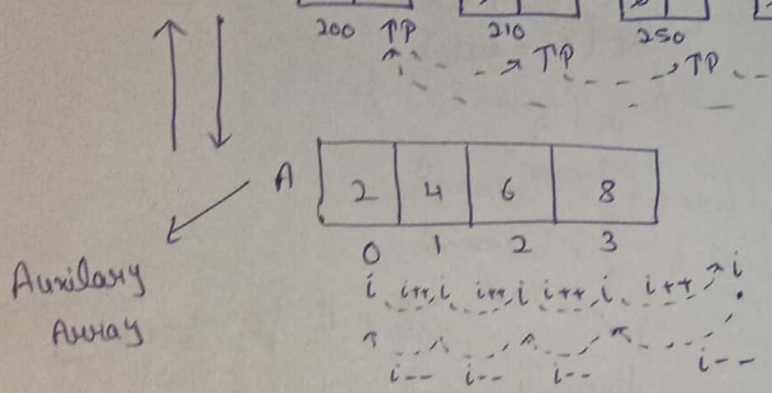
Reversing a List :-

METHOD - 1 :- Reversing Elements

METHOD - 2 :- Reversing links → we Prefer this more

⇒ **METHOD - 1** :- [redacted]






```
P = first;
i = 0;
while (P != NULL)
{
    A[i] = P->data;
    P = P->next;
} i++;
P = first;
i--;
while (P != NULL)
{
    P->data = A[i--];
    P = P->next;
}
```

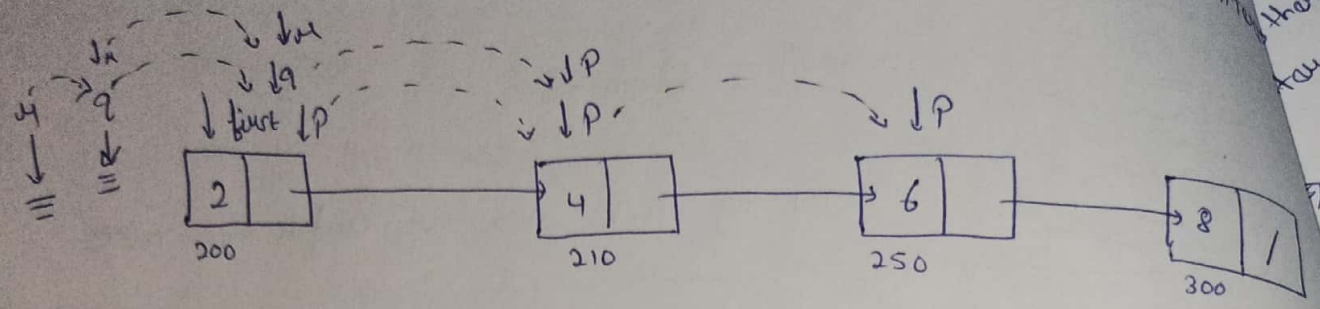
Time Complexity :- O(n)

In this method, we require an Array of size equal to linked list, so it will take more space thats why we Prefer a method reversing links.

Scanned with CamScanner

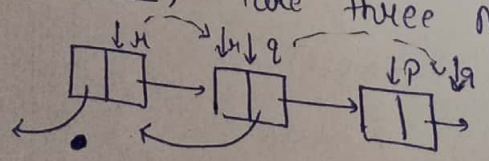
→ **Method - 2** :-

In this we will take two sliding Pointers that Reverse exchange




```
P = first;
q = NULL;
r = NULL;
while (P != NULL)
{
    r = q;
    q = P;
    P = P → next;
    q → next = r; // for Reversing
}
first = q;
```

<u>Now, To Reverse links</u>, Take three Nodes, we require 3 pointers

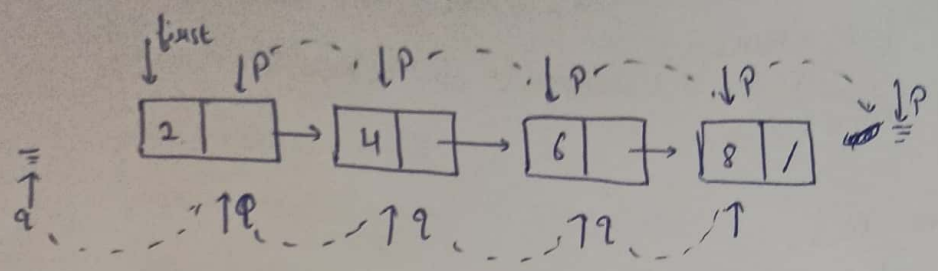


Procedure :- Is mein Pehle 'q' pointer lenge, jo ki 'r' pointer ko point krega and than q pointer 'r' ko point krwayengh, but next Node ke liye hamko Ek Aur pointer chahiye hoga, bcz agr next Node ka pointer nahi hoga than ham aagh proceed nahi krPayengh, Than ham q ko 'p' pointer pr point krengh & 'r' ko 'q' pointer pr & 'p' ko 'P → next' pr, which we Called as sliding. Siff "q → next" ko 'r' pr point krna is additional step for reversing.

Scanned with CamScanner

Another Reason why we Prefer Reversing links bcz we don't want to interchange the elements b/w Nodes.

⇒ **Recursive Reverse for Linked List :-**




```
Void Reverse ( Node *q, Node *p)
{
    if (p != NULL)
    {
        Reverse (p, p->next);   || Recursive Call
        p->next = q;   || Returning time
    }
    else
    {
        first = q;
    }
}
```

Scanned with CamScanner

Code :-

```
Struct Node
{
    int data;
    Struct Node * next;
}* first = NULL;

int main ()
{
    int N, i;
    printf (" Enter No. of Elements : \n");
    Scanf ("%d", &N);
    printf (" Elements are : \n");
    int A[N];
    for (i=0 ; i< N; i++)
        Scanf ("%d", &N);
    Void create (int A[], int N);
    Void Display (Struct Node *p);
    Void Reverse 1 (Struct Node *p);
    Void Reverse 2 (Struct Node *q, Struct Node *p);
    Create (A, N);
    Display (first);
    printf (" \n After Reversing \n");
    Reverse1 (first);
    Display (first);
    Reverse 2 [redacted] (NULL, first);
    Display (first);
}
```

Scanned with CamScanner

```
void create (int A[], int n)
{
    int i;
    struct Node *t, *last;
    first = (struct Node *) malloc (sizeof (struct Node));
    first -> data = A[0];
    first -> next = NULL;
    last = first;
    for (i = 1; i < n; i++)
    {
        t = (struct Node *) malloc (sizeof (struct Node));
        t -> data = A[i];
        t -> next = NULL;
        last -> next = t;
        last = t;
    }
}

void Display (struct Node *p)
{
    while (p != NULL)
    {
        printf ("%d_", p -> data);
        p = p -> next;
    }
}
```

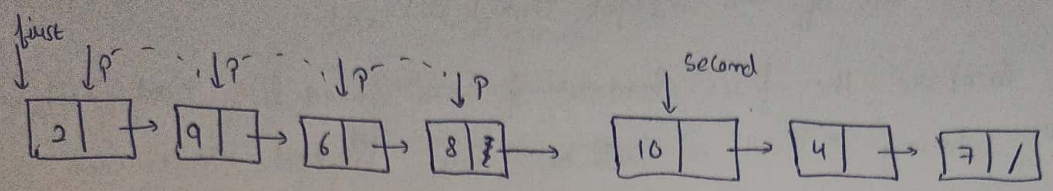
Scanned with CamScanner

```
Void Reverse1 (struct Node *p)
{
    p = first;
    struct Node *q = NULL;
    struct Node * r = NULL;
    while (p! = NULL)
    {
        r = q;
        q = p;
        p = p→next;
        q→next = r;
    }
    first = q;
}

Void Reverse2 (struct Node *q, struct Node *p)   // Recursively
{
    if (p)
    {
        Reverse2 (p, p→next);
        p→next = q;
    }
    else
    {
        first = q;
    }
}
```

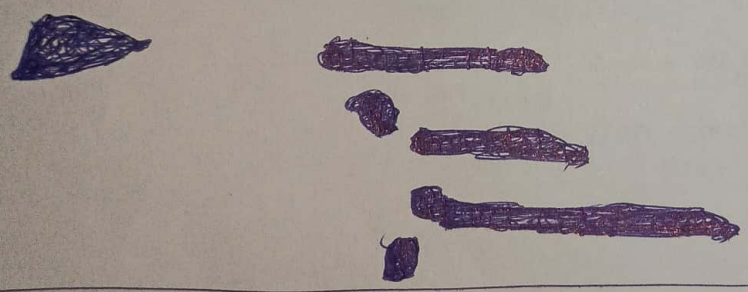
Scanned with CamScanner

Concatenating 2 Linked Lists :-

Joining two linked lists | Appending first list with second list.

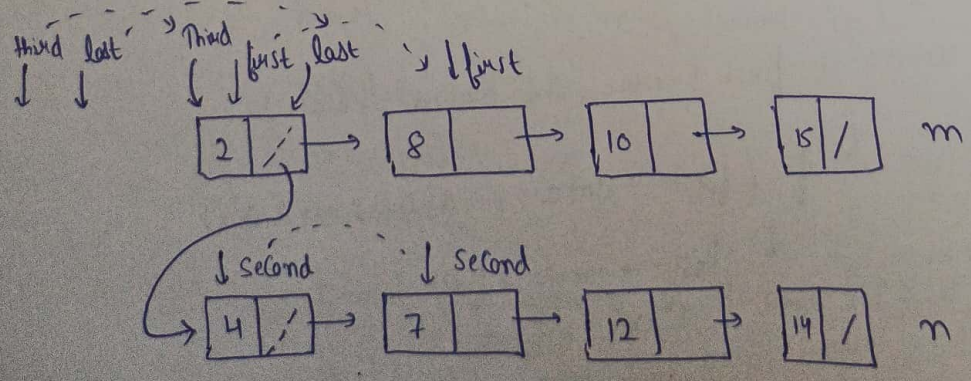



```
P = first;
while (P→next != NULL)
    {
      P = P→next;
    }
P→next = Second;
Second = NULL;   // Its our choice to make it NULL or not.
```



☆ Merging 2 Linked Lists :-

We can do merging in two sorted lists only.



Time Complexity :-

$O(m+n)$

Scanned with CamScanner

For merging we require one more pointer (i.e. third) and for that pointer we need a pointer (i.e last)

"Third" will pointing to first node of the linked list and "last" will to the last node of this merged linked list.

- Now Compare the first element of both linked list and check which is smaller, bring "Third" as well as "last" to that Node, then move "first" to next Node and make last → next = 'NULL'.

- Again Compare, Make "last → next" should point on Node whose data is small and bring last on that node and move "second" Node and make "last → next" as NULL.

- Then Continue Same Procedure.

```
if ( first → data < Second → data)
  {
    third = last = first ;
    first = first → next ;
    last → next = NULL ;
  }
else
  {
    third = last = Second ;
    Second = Second → next ;
    last → next = NULL ;
  }
while ( first != NULL && Second != NULL)
  {
    if (first → data < Second → data)
      {
        last → next = first ;
        last = first ;
        first = first → next ;
        last → next = NULL ;
      }
```

Scanned with CamScanner

```
else
  {
      last -> next = Second;
      last = Second;
      Second = Second -> next;
      last -> next = NULL;
  }
if (first ! = NULL)
      last -> next = first;
else
      last -> next = Second;
```

Code for Concatenating and Merging :-

```
Struct Node
{
    int data;
    Struct Node * next;
} *first = NULL, * Second = NULL, * third = NULL;

int main()
{
    int N, i;
    printf ("Enter the no. of terms : \n");
    Scanf ("%d", & N);
    int A[N];
    int B[N];
    printf (" Elements of A are : \n");
    for (i = 0 ; i < N; i++)
      {
        scanf ("%d", & A[i]);
      }
```

Scanned with CamScanner

```
printf ("Elements of B are : \n");
for (i=0 ; i< N; i++)
 {
   scanf ("%d", &B[i]);
 }
Void create (int A[], int N);
Void create2 (int B[], int N);
Void Concate ( struct Node * P, struct Node *q);
Void Merge (struct Node *P, struct Node *q);
Void Display (struct Node * P)
 create (A, N);
 Display (first);
 printf ("\n\n");
 create2 (B, N);
 Display (second);
 // printf (" \n after Concatination \n");
 // Concate (first, second);
 // Display (third);
 printf (" \n After merging \n");
 Merge (first, second);
}   Display (third);

Void Display (struct Node *P)
 {
   While (P! =NULL)
     {
       printf ("%d_", P→data);
       P = P→ next;
     }
 }
```

Scanned with CamScanner

```
void create (int A[], int N)
{
    int i;
    struct Node *t, * last;
    first = (struct node *) malloc (size of (struct Node));
    first -> data = A[0];
    first -> next = NULL;
      last = first;
    for (i=1; i<N; i++)
      {
        t = (struct Node *) malloc (size of (struct Node));
        t-> data = A[i];
        t -> next = NULL;
        last -> next = t;
          last = t;
      }
  }
void create 2 (int B[], int N)
  {  int i;
     struct Node *t, * last;
     Second = (struct Node *) malloc (size of (struct Node));
     Second -> data = B[0];
     Second -> next = NULL;
       last = Second;
     for (i=1; i<N; i++)
       {  t= (struct Node *) malloc (size of (struct Node));
          Second -> data = B[0];
          Second -> next = NULL;
            last -> next = t;
            last = t;
       }
  }
```

Scanned with CamScanner

```
void Concate (struct Node *P, struct Node *q)
{
    third = P;
    while ( P→ next ! = NULL)
      { P=P → next;
      }
    P→ next = q;
}
void Merge ( struct Node *P, struct Node *q)
{
    struct Node *last;
    if (P→ data < q →data)
      {
        third = last = P;
        P = P → next;
        third → next = NULL;
      }
    else
      {
        third = last = q;
        q = q → next;
        third → next = NULL;
      }
    while (P && q)
      {
        if (P→ data < q → data)
          {
            last → next = P;
            last = P;
            P = P→ next;
          }
        last → next = NULL;
```

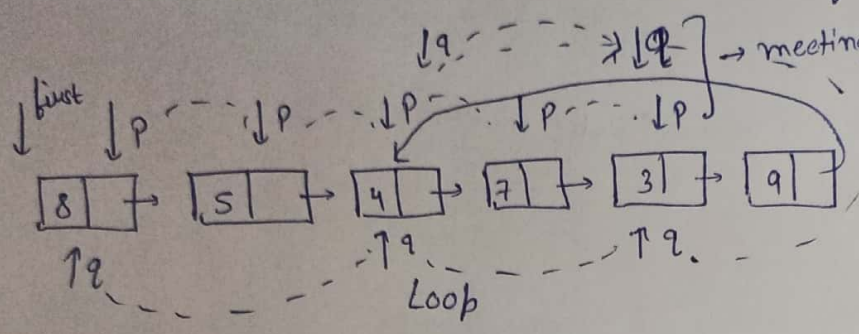
Scanned with CamScanner

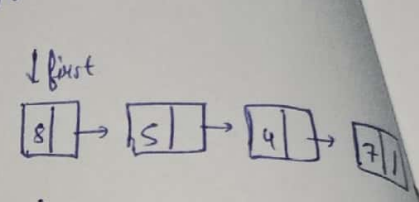
```
else
 {
      last -> next = q;
       last = q;
       q = q -> next;
        last -> next = NULL;
     }
}
 if (p)
  {
     last -> next = p;
   }
  if (q)
   {
     last -> next = q;
   }
}
```

Scanned with CamScanner

☆ Check for loop in List :-

If a last node is pointing on some node :- then it is forming loop.

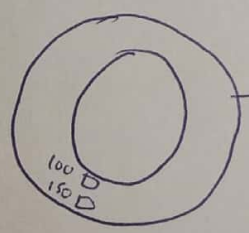




Linear :- while Traversing if it becomes Null or pointer becomes Null then it will be linear

- P will move by 1 step.
  q will move by 2 step. → if they meet again, then it will be loop there.

Eg :- Let us take two cars and circular track.



→ 15 laps (they will definetly meet at one pt.)

```
int isLoop (Node *f)
{
    Node *P, *q;
      P=q = f;
    Do
    {
        P= P→next;
        q =q→next;
        if (q)
        else q=q→next;
          q = NULL;
    } while (P && q && P!=q)
    if (P==q)
        return true;
    else
        return false;
}
```

→ q = q ? q → next : q;

Time Complexity :- O(n)

Code :-

```
Struct Node
{
    int data;
    Struct Node * next;
  } * first = NULL;

int main()
{
    Struct Node *t1, *t2;
    int A[4] = {10, 20, 30, 40};
    Void create (int A[], int N);
    int isLoop (struct Node * f);
    Create (A, 4);
    t1 = first -> next -> next; || Pointing to 3rd Node
    t2 = first -> next -> next -> next; || Pointing to last Node
    t2 -> next = t1; || forming loop
    int m = isloop (first);
    printf ("%d", m);
  }

Void create( int A[], int N)
  {
    int i;
    Struct Node *t, * last;
    first = (struct Node *) malloc ( size of (struct Node));
    first -> data = A [0];
    first -> next = NULL;
    last = first;
    for (i = 1; i < N; i ++)
      {  t = (struct Node *) malloc (size of (struct Node));
         t -> data = A[i];
         t -> next = NULL;
```

Scanned with CamScanner

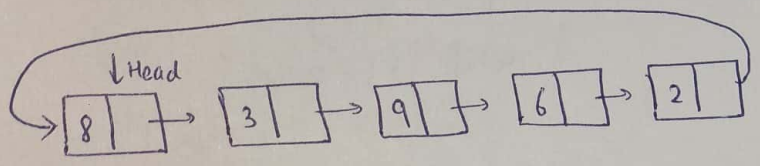
```
        last -> next = t;
        last = t;
    }
}

int isloop (struct Node *f)
{
    struct Node *p, *q;
    p = q = f;
    Do
    {
        p = p -> next;
        q = q -> next;
        if (q)
            q = q -> next;
        else
            q = NULL;
    } while (p && q && p != q);
    if (p == q)
        return 1;   || loop
    else
        return 0;   || linear
}
```

Scanned with CamScanner

Circularly Linked List :- A Linked List in which last Node is pointing on 1st Node then it is circular linked list. It is a Combination of Nodes linked in a circular manner, so there will be no first Node and No [redacted] last Node, so we used "Head" for one of the Node.

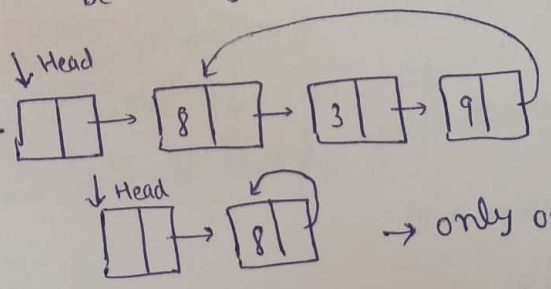
Eg :- Contacts list in mobile phone (old fashioned mobile)

⇒ **where to use :-** if we want only one directional Access and in a circular manner, then we use circular Linked List.



(i)

Aiguemant :- Circular Linked list Cannot be NULL, though it may be empty.



(ii)

→ only one Node

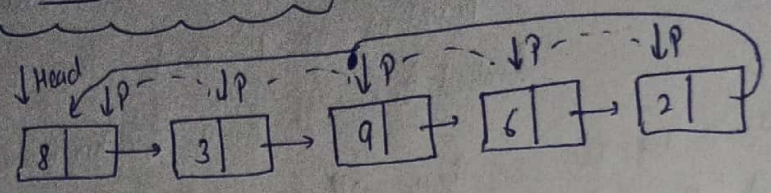
Empty ← it is pointing on one of the Node, so we Can Access all the Nodes.

Note :- we Can use representation (i) as well as (ii), Like in a Case if we want that "Head" Node as NULL, then go for (ii).

But we will study about (i).

Scanned with CamScanner

★ Display Circular Linked List :-



We use Do while bcz, Condition is $\boxed{P != Head}$, and this Condition occurs in first Calling, so it will not enter into loop that's why we use Do-while loop.

```
void Display (Node *P)
{
    Do
    {
        printf ("%d", P->data);
        P = P->next;
    } while (P != Head)
}
```

Recursive function :- For this, if we write Condition (P != Head) then it will again not enter, that's why we take a variable flag, if it is '0' then run, else not enter.

```
Void Display (Node *P)
{ static int flag = 0;
    if (P != Head || flag == 0)
    {
        flag = 1;
        printf ("%d", P->data);
        Display (P->next);
    }
} flag = 0; // when comes out of the loop
```

It will occupy more space, bcz stacks will be formed.

Scanned with CamScanner

ode :-

```
Struct Node
{
    int data;
    Struct Node * next;
} * Head = NULL;

int main( )
{
    int N, i;
    printf (" Enter No. of Elements: \n");
    Scanf ("%d", & N);
    int A [N];
    for( i = 0 ; i < N; i++)
     Scanf ("%d", & A[i]);
    Void Create (int A[], int N);
    Void Display ( struct Node *P);
    Void R Display (struct Node *P);
     Create (A, N);
     Display (Head);
     R Display (Head);

}

Void Display (Struct Node * P)
{
    Do
    {
        printf ("%d-", P → data);
        P = P → next;
    } while (P! = head)

}
```

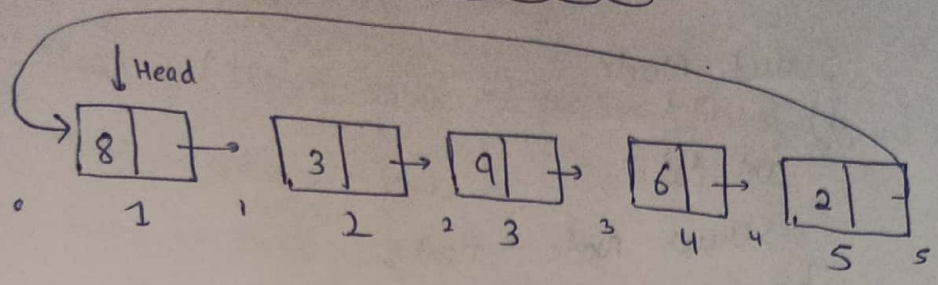
→ To See circular manner, we can See this in debug window.

Scanned with CamScanner

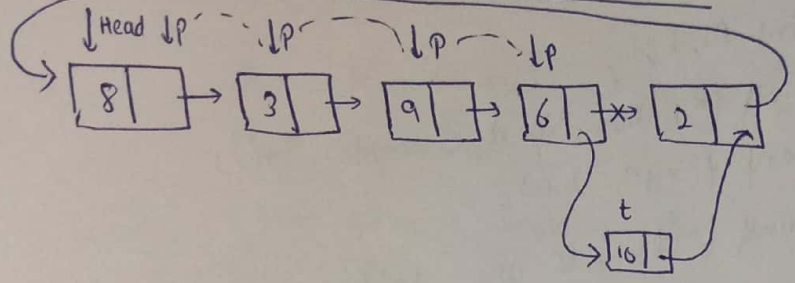
```
Void create (int A[], int N)
{
    int i;
    struct Node *t, * last;
    Head = (struct Node *) malloc (size of (struct Node));
    Head -> data = A[0];
    Head -> next = Head;
    last = Head;
    for (i=1; i<N; i++)
    {
        t = (struct Node *) malloc (size of (struct Node));
        t-> data= A[i];
        t-> next = last -> next;
        last -> next = t;
        last = t;
    }
}

Void R Display ( struct Node *P)
{
    Static int flag =0;
    if (P! = Head !! flag ==0)
    {
        flag = 1;
        printf ("%d _", P -> data);
        R Display (P -> next);
    }
    flag=0;
}
```

Scanned with CamScanner

Inserting in a circularly linked list :-

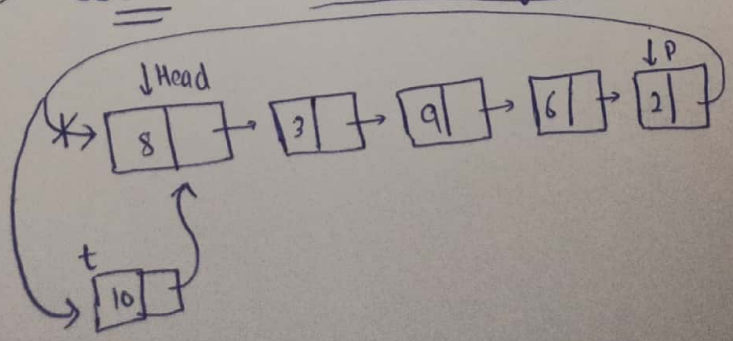


⇒ Case - 1 :- Inserting at any other position.




```
Node *t;
Node * P = Head;
for (i=0; Pos-1; i++)
    P=P→next;
t = new Node
t→data = x;
t→next = P→next;
P→nex = t;
```

⇒ Case - 2 :- Inserting before Head Node | Inserting after last node:- is same




```
Node *t = new Node;
t→data = x;
t→next = Head;
while (P→next != Head)
  {P = P→next;
  }
P→next = t;
Head = t;
```

Scanned with CamScanner

⇒ Code Containing Case-1 & Case-2 :-

```
Struct Node
{
    int data;
    Struct Node * next;
} * Head = NULL;
int main()
{
    int N, i;
    printf("Enter No. of elements : \n");
int A[n]; scanf("%d", &N);
    printf("Elements are : \n");
    for (i=0; i<N; i++)
        Scanf("%d", &A[i]);
Void create (int A[], int N);
Void Display (Struct Node *P);
Void Insert (Struct Node *P, int index, int x);
Create (A, N);
Insert (Head, 2, 10);
Display (Head);
}
```

Scanned with CamScanner

```
void create (int A[], int N)
{
    int i;
    Struct Node *t, *last;
    Head = (Struct Node *) malloc (Size of (Struct Node));
    Head -> data = A[0];
    Head -> next = Head;
        last = Head;
    for (i=1 ; i<N; i++)
        {
            t = (Struct Node *) malloc (Size of (struct Node));
            t-> data = A[i];
            t -> next = last -> next;
                last -> next = t;
                last = t;
        }
}
Void Display (Struct Node *P)
{
    DO {
        printf ("%d _", P-> data);
        P = P-> next;
    } while (P != Head)
}
```

Scanned with CamScanner

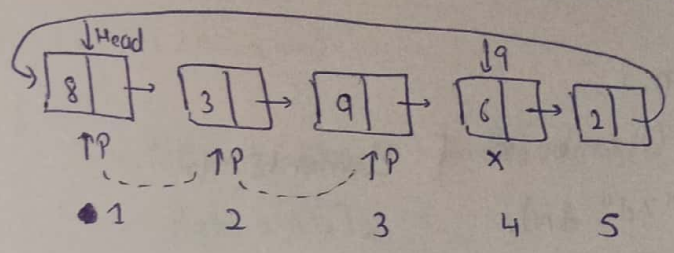
```
Void Insert (Struct Node *P, int index, int x)
{
    Struct Node *t;
    int i;
    if (index == 0)
    {
        t = (Struct Node *) malloc (Size of (Struct Node));
        t->data = x;
        if (Head == NULL)
        {
            Head = t;
            Head -> next = Head;
        }
        else
        {
            While (P->next != Head)
                P = P -> next;
            P->next = t;
            t->next = Head;
            Head = t;
        }
    }
    else
    {
        for (i=0 ; i< index-1 ; i++)
            P = P->next;
        t = (Struct Node *) malloc (Size of (Struct Node));
        t->data = x;
        t->next = P->next;
        P->next = t;
    }
}
```

Scanned with CamScanner

Deleting from circular Linked List

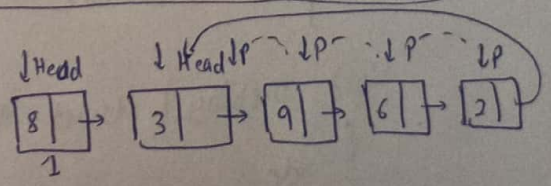
1. Deleting Head Node
2. Deleting a Node from given position

Case 1 :- Deleting a Node from given position :-




```
P = Head ;
for (i=0 ; i< Pos - 2 ; i++)
    { P= P→ next ;
    }
q = P→ next;
P→ next = q → next;
x = q → data;
delete q ;
```

Case-2 :- Deleting Head Node :-




```
P=Head ;
while ( P→ next != Head)
        P=P→ next;
P→ next = Head → next;
x = Head → data ;
delete Head;
Head = P→ next;
```

Scanned with CamScanner

⇒ **Code Containing Case-1 & Case-2** :-

```
Struct Node
{
    int data;
    Struct Node *next;
}* Head = NULL;

int main ()
{
  int n, i;
  printf ("Enter no. of elements : \n");
  Scanf ("%d", &N);
  int A[N];
  for (i=0 ; i<N; i++)
   Scanf ("%d", &A[i]);
  Void Create (int A[], int N);
  Void Display (Struct Node *P);
  int delete (Struct Node *P, int index);
  Create (A, N);
  printf (" Delete Element : %d \n", (delete (head, s)));
  Display (Head);
}

Void Display (Struct Node *P)
{
  do{
     printf ("%d", P→data);
     P = P→next;
  } while (P != Head);
}
```

Scanned with CamScanner

```
void Create (int A[], int N)
{
    int i;
    struct Node *t, *last;
    Head = (struct Node *) malloc (sizeof (struct Node));
    Head -> data = A[0];
    Head -> next = Head;
    last = Head;
    for (i=1; i<N; i++)
    {
        t = (struct Node *) malloc (sizeof (struct Node));
        t -> data = A[i];
        t -> next = last -> next;
        last -> next = t;
        last = t;
    }
}

int Delete (struct Node *P, int index)
{
    struct Node *q;
    int i, x;
    if (index ==1)
    {
        P = Head;
        while (P -> next != Head) P = P -> next;
        x = Head -> data;
        if (Head == P) // first and last Node
        {
            free (Head);
            Head = NULL;
        }
        else
        {
            P -> next = Head -> next;
            free (Head);
            Head = P -> next;
        }
    }
```

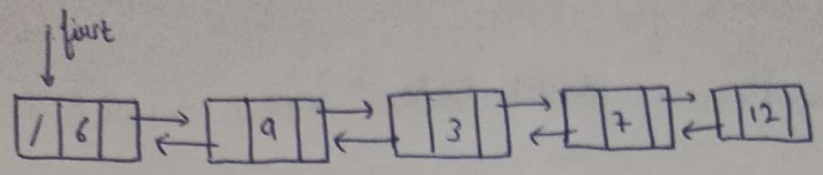
Scanned with CamScanner

```
else
{
    for(i=0 ; i< index - 2 ; i++)
    {
        p=p→ next;
    }
    q=p→ next;
    p→ next = q → next;
        x=q → data;
        free (q);
    }
    return x;
}
```

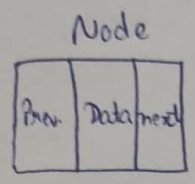
function closed.

Scanned with CamScanner

Doubly Linked List :-



A Node has a pointer on 'next node' as well as on 'Previous Node'.

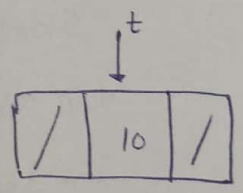


Structure for this Node :-

```
Struct Node
{
    Struct Node * Prev;
    int data;
    Struct Node * next;
}
```

Formation :-

```
struct Node *t;
t = new Node;
t→ Prev = NULL;
t→ data = 10;
t→ next = NULL;
```



"The main Benefit of using doubly linked list is we can access elements in either direction or we can say that doubly linked list is bi-directional.

Eg :- When we open browers pages, then the one page will link to Another and if we want to Access the previous page, then we can Access any page at any time, so Here doubly linked list is Stored.

Scanned with CamScanner

★ Display Doubly Linked List :-

```
Struct Node
{
    Struct Node *Prev;
    int data;
    Struct Node * next;
} * first = NULL;

int main ()
{
    int N, i;
    printf (" Enter No. of Elements : \n");
    Scanf ("%d", &N);
    int A [N];
    printf (" Elements Are : \n");
    for (i=0 ; i<n; i++)
        Scanf ("%d", & A[i]);
    Void Display (struct Node *P);
    int length (struct Node *P);
    Create (A, N);
    printf (" \n Length is : %d \n", length (first));
    Display (first);
}
Void Display (Struct Node *P)
{ while (P)
    {
        printf ("%d", P → data);
        P = P → next;
    }
}
```

Scanned with CamScanner

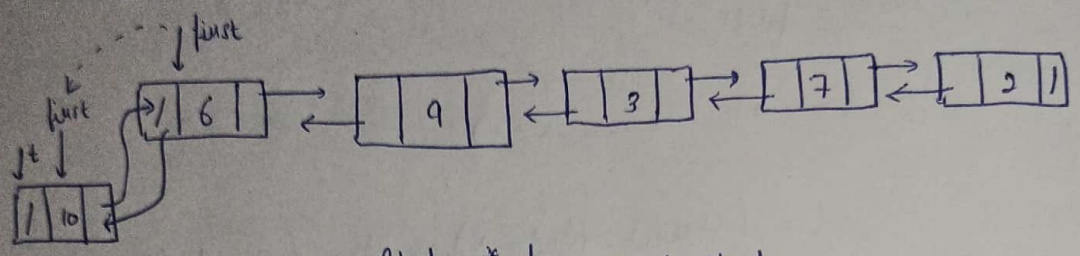
```
void create (int A[], int N)
{
    struct Node *t, *last;
    int i;
    first = (struct Node *) malloc (sizeof (struct Node));
    first -> data = A[0];
    first -> Prev = first -> next = NULL;
    last = first;
    for (i = 1; i < N ; i++)
    {
        t = (struct Node *) malloc (sizeof (struct Node));
        t -> data = A[i];
        t -> next = last -> next;
        t -> Prev = last;
        last -> next = t;
        last = t;
    }
}
int Length (struct Node *P)
{
    int len = 0;
    while (P)
    {
        len++;
        P = P -> next;
    }
    return len;
}
```

Scanned with CamScanner

☆ Inserting in a Doubly Linked List :-

1. Inserting before first Node
2. At any given index.

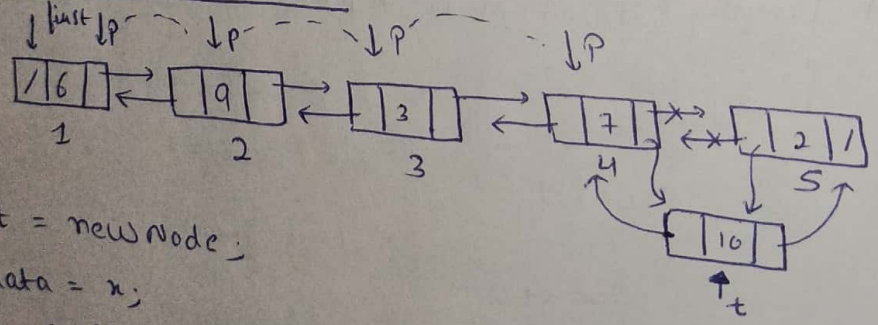
Case-1 :- Before first Node




```
Node * t = new Node;
t→data = x
t→Prev = NULL;
t→next = first;
first→Prev = t;
first = t;
```

Time Complexity :- O(1)

Case-2 :- At any Given index




```
Node * t = new Node;
t→data = x;
for(i=0; i<Pos-1; i++)
    P = P→next;
t→next = P→next;
t→Prev = P;
if (P→next)
    P→next→Prev = t;
P→next = t;
```

Time Complexity :-

Minimum :- O(1)

Maximum :- O(n)

Scanned with CamScanner

ode :-

```
Struct Node
{
   Struct Node * Prev;
   int data;
   Struct Node * next;
}* first = NULL;

int main()
{
   int N, i;
   printf(" Enter No. of elements : \n");
   Scanf("%d", &N);
   int A[N];
   printf(" Elements are : \n");
   for (i=0; i<N; i++)
        Scanf("%d", &A[i]);
   void Display (struct Node *P);
   Void Length (struct Node *P);
   void Insert (Struct Node *P, int index, int x);
   Create (A,N);
   Display (first);
   printf("\n");
   printf(" After Insertion");
   Insert (first, 4, 10);
   Display (first);
}

Void Display (Struct Node *P)
{
    while (P)
    {
       printf ("%d", P→data);
       P=P→next;
    }

}
```

Scanned with CamScanner

```
Void create (int A[], int N)
{
    Struct Node *t, *last;
    int i;
    first = (Struct Node *) malloc (Size of (Struct Node));
    first -> data = A[0];
    first -> Prev = first -> next = NULL;
    last = first;
    for (i=1; i<N; i++)
    {
        t = (Struct Node *) malloc (Size of (Struct Node));
        t-> data = A[i];
        t-> next = last -> next;
        t-> Prev = last;
        last -> next = t;
        last = t;
    }
}

int length (Struct Node * P)
{
    int len = 0;
    while(P)
    {
        len++;
        P = P-> next;
    }
}
```

Scanned with CamScanner

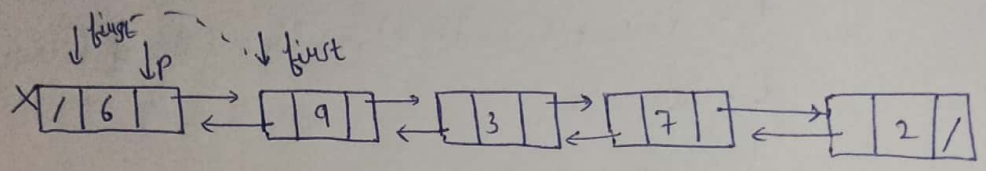
```
void Insert( struct Node *P, int index, int x)
{
    struct Node *t;
    int i;
    if (index<0 || index > length(P))
        return;
    if (index ==0)
    {
        t = (struct Node *) malloc (size of (struct Node));
        t→data = x;
        t→Prev = NULL;
        t→next = first;
        first→Prev = t;
        first = t;
    }
    else
    {
        for (i=0; i<index-1; i++)
        {
            P= P→next;
        }
        t= (struct Node *) malloc (size of (struct Node));
        t→data = x;
        t→Prev = P;
        t→next = P→next;
        if (P→next)    || Conditional but mandatory
            P→next →Prev=t;
        P→next =t;
    }
}
```

Scanned with CamScanner

☆ Delete in a Doubly Linked List :-

① Delete First Node
② Delete from Given index.

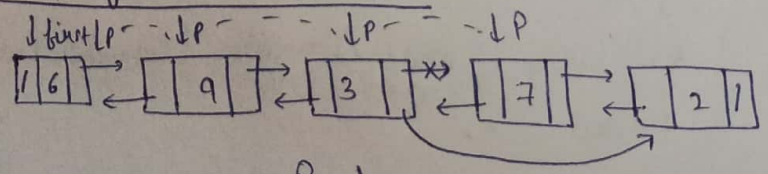
Case-1 :- Delete First Node




```
P = first;
first = first -> next;
x = P -> data;
delete P;
if (first != NULL)
    first -> Prev = NULL;
```

Time Complexity :- O(1)

Case-2 :- Delete from Given position




```
P = first;
for (i=0; i< pos-1 ; i++)
    P = P -> next;
P -> Prev -> next = P -> next
if (P -> next != NULL)   || If No node after node 'which have to be deleted', is present or not.
P -> next -> Prev = P -> Prev;
x = P -> data;
delete P;
```

Time Complexity :-

Maximum :- O(1)
Minimum :- O(n)

Scanned with CamScanner

Code :-

```
Struct Node
{
    Struct Node * Prev;
    int data;
    Struct Node * Next;
}* first = NULL;

int. main ()
{
    int N, i;
    printf ("Enter No. of Elements: \n");
    scanf ("%d", & N);
    int A[N];
    printf (" Elements are: \n");
    for (i = 0; i < N; i++)
        Scanf ("%d", & A[i]);
    Void Display (Struct Node *P);
    Void create ( int A[], int N);
    int length (struct Node *P);
    Void delete (Struct Node *P, int index);
    Create (A, N);
    Display (first);
    printf ("\n");
    printf (" After Deletion \n");
    Delete (first, 1);
    Display (first);

}
```

Scanned with CamScanner

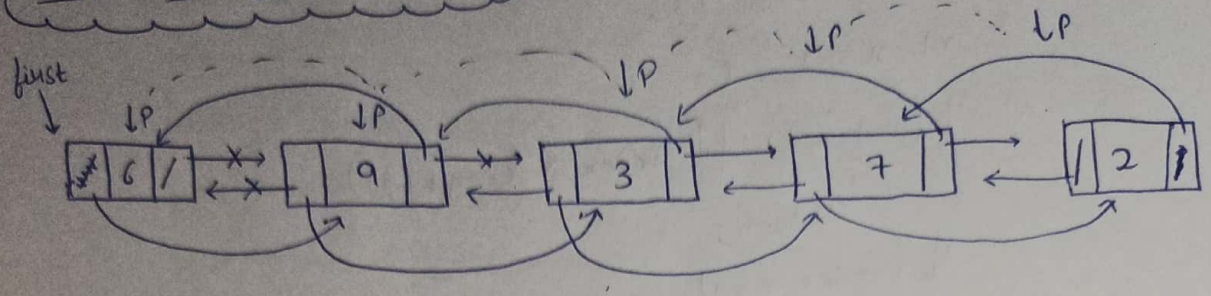
```
void create (int A[], int N)
{
    struct Node *t, *last;
    int i;
    first = (struct Node *) malloc (sizeof (struct Node));
    first -> data = A[0];
    first -> Prev = first -> next = NULL;
    last = first;
    for (i=1; i<N; i++)
    {
        t = (struct Node *) malloc (sizeof (struct Node));
        t -> data = A[i];
        t -> next = last -> next;
        t -> Prev = last;
        last -> next = t;
        last = t;
    }
}

void Display (struct Node *P)
{
    while(P)
    {
        printf ("%d", P -> data);
        P = P -> next;
    }
}

int length (struct Node *P)
{ int len = 0;
    while (P)
    {
        len ++;
        P = P -> next;
    }
    return len;
}
```

Scanned with CamScanner

```
void delete( struct Node *p, int index)
{
    struct Node *q;
    int x=-1 , i;
    if (index <1 || index > length(p))
        Return;
    if (index ==1)
    {
        q = first;
        first = first -> next;
        x = q -> data;
        free (q);
        if (first != NULL)
            first -> Prev = NULL;
    }
    else
    {
        q = first;
        for (i=0 ; i< index-1 ; i++)
        { q = q -> next;
        }
        q -> prev -> next = q -> next;
        if (q -> next != NULL)
        {
            q -> next -> Prev = q -> Prev;
        }
        x = q -> data;
        free(q);
    }
}
```

Scanned with CamScanner

★ Reverse a Doubly Linked List :-




```
P = first;
while (P != NULL)
{
    temp = P->next;
    P->next = P->Prev;   } -> swapping of pointers
    P->Prev = temp;
    P = P->prev;   || bcz links are already change
    if (P->next == NULL)
        first = P;
}
```

Code :-

```
Struct Node
{
    Struct Node * Prev;
    int data;
    Struct Node * Next;
}* first = NULL;
int main ()
{
    int N, i;
    printf (" Enter No. of Elements :\n");
    Scanf (" %d", &N);
    int A[N];
    printf (" Elements are:\n");
    for (i=0 ; i<N; i++)
        Scanf (" %d", &A[i]);
```

Scanned with CamScanner

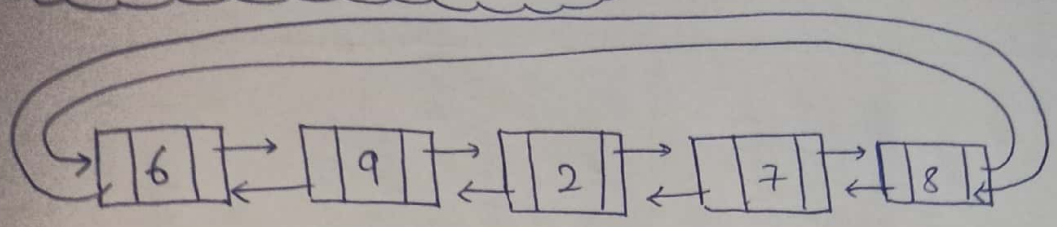
```
void Display (struct Node *P);
void Create (int A[], int N);
void Reverse (struct Node *P);
  Create (A,N);
  Display (first);
  printf ("In After Reversal :In");
   Reversal (first);
    Display (first);

}
void create (int A[], int N)
  {
     Struct Node *t , *last;
      int i;
      first = (struct Node *) malloc (size of (struct Node));
      first -> data = A[0];
      first -> Prev = first -> next = NULL;
      last = first;
      for (i=1 ; i<N; i++)
      {
          t = (struct Node *) malloc (size of (struct Node));
          t-> data = A[i];
          t-> next = last -> next;
          t-> Prev = last;
          last -> next = t;
           last = t;
      }
  }
```

Scanned with CamScanner

```
Void Display (struct Node *P)
{
    while (P)
    {
        printf ("%d", P→data);
        P = P→next;
    }
}

Void Reversal (struct Node *P)
{
    struct Node * temp;
    P = first;
    while (P)
    {
        temp = P→next;
        P→next = P→Prev;
        P→Prev = temp;
        P = P→Prev;
        if (P! = NULL && P→next==  NULL)
            first = P;
    }
}
```

Scanned with CamScanner

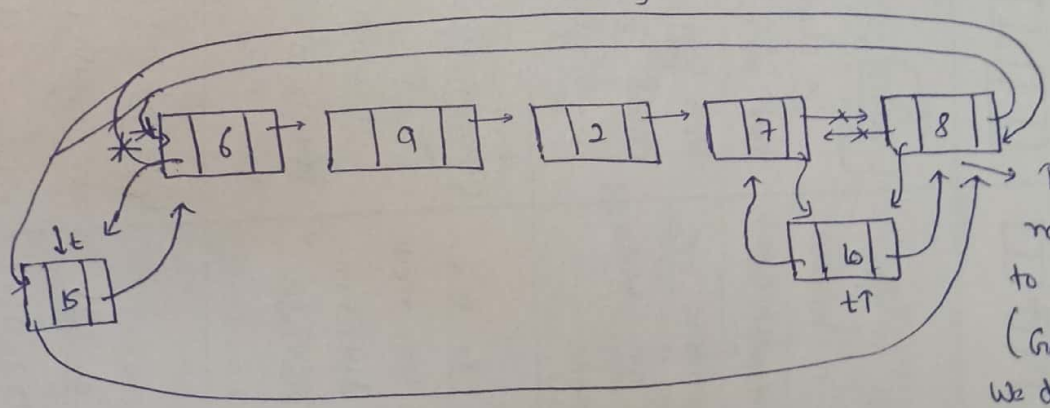
Circular doubly Linked List :-



Display :-

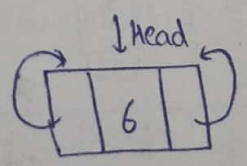
```
P = Head ;
do
{
    printf ("%d", P→data);
    P = P→ next ;
} while (P! = Head);
```

Insertion :- "Inserting at any position after Head will be the Same as in doubly linked list."



This is a node previous to Head Node (Go to that node we dont have to traverse the list Just [redacted] take this as previous to head).

If there is only one node then

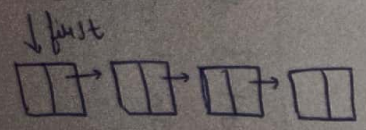
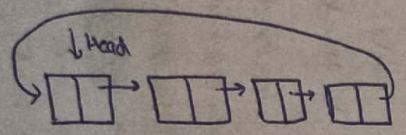
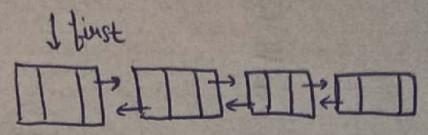
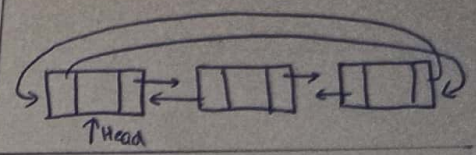


It previous & next should point to Same node.

Q- Write Insert & delete in Revision time.

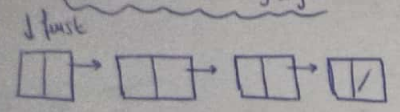
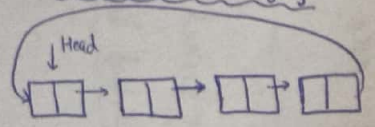
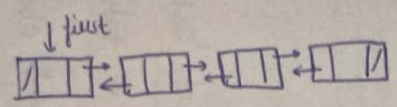
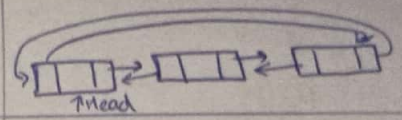
Scanned with CamScanner

☆ Comparison of linked lists :-

Linear Singly	Circular Singly	Linear Doubly	Circular Doubly
			
Space :- For 'n' elements, 'n' extra pointers are required so, it contains one value and one pointer in every node.	**Space** :- It is also require 'n' pointers for 'n' elements, so it also contains one value and one pointer in every node.	**Space** :- In this, for 'n' elements, '2n' pointers are required, so we can say that it contains double pointers that was in singly linked list.	**Space** :- It also contains '2n' pointers for 'n' elements, so it also contains double pointers than that of singly linked list.
Traversing :- In this we can move in only forward direction.	**Traversing** :- In this we can move in forward direction as well as in circular manner.	**Traversing** :- In this we can move in either direction. i.e. forward as well as backward.	**Traversing** :- In this we can move in either direction. i.e forward as well as backward but also in circular manner.
Insert :- ☆ Inserting at left • Time Complexity :- $O(1)$ • only one link is modified	**Insert** :- ☆ Inserting at left • Time Complexity :- $O(n)$ • Two links are modified	**Insert** :- ☆ Inserting at left • Time Complexity :- $O(1)$ • Three links are modified	**Insert** :- ☆ Inserting at left • Time Complexity :- $O(1)$ • Four links are modified.

Scanned with CamScanner

Comparison of Linked Lists :-

Linear Singly	Circular Singly	Linear Doubly	Circular Doubly
			
★ At any Position • Time Complexity :- maximum :- O(n) minimum :- O(1) • Two links are modified	★ At any Position • Time Complexity :- maximum :- O(n) minimum :- O(1) • Two links are modified	★ At any Position • Time Complexity :- maximum :- O(n) minimum :- O(1) • four links are modified but for last, only three links are modified	★ At any Position • Time Complexity :- maximum :- O(n) minimum :- O(1) • four links have to be modified.
Delete :- ★ Deleting Head • Time Complexity :- O(1) • No links have to modified.	Delete :- ★ Deleting Head • Time Complexity :- O(n) • one link is modified.	Delete :- ★ Deleting Head • Time Complexity :- O(1) • one links modified.	Delete :- ★ Deleting Head • Time Complexity :- • Two links modified.
★ At any position • Time Complexity :- maximum :- O(n) minimum :- O(1) • one link is modified	★ At any position • Time Complexity :- maximum :- O(n) minimum :- O(1) • one link is modified	★ At any position • Time Complexity :- maximum :- O(n) minimum :- O(1) • Two links are modified but if we deleting at last then one link will be modified.	★ At any position • Time Complexity :- maximum :- O(n) minimum :- O(1) • Two links are modified.

Scanned with CamScanner

☆ When and which Linked List has to be used :-

- If we want to Access data one by one and dont have to Come back to previous data then use Linear Singly Linked List.
- If we want to Access data in forward direction but also in Circular manner, then use circular Singly Linked List.
- If we want to Access data bi-directionally then use Linear doubly linked list.

Imp

Now, The circular doubly linked list is the best linked list as it has all possible features of all linked list. but it takes lots of space bcz of extra pointers, So if we worried about space then dont use it else use this, bcz now a days space is not a Concern.

⇒ **Note** :- For languages like Java, it has inbuilt Classes for linked list and that linked list is circular doubly linked list. bcz now a days, space is not an issue.

Scanned with CamScanner

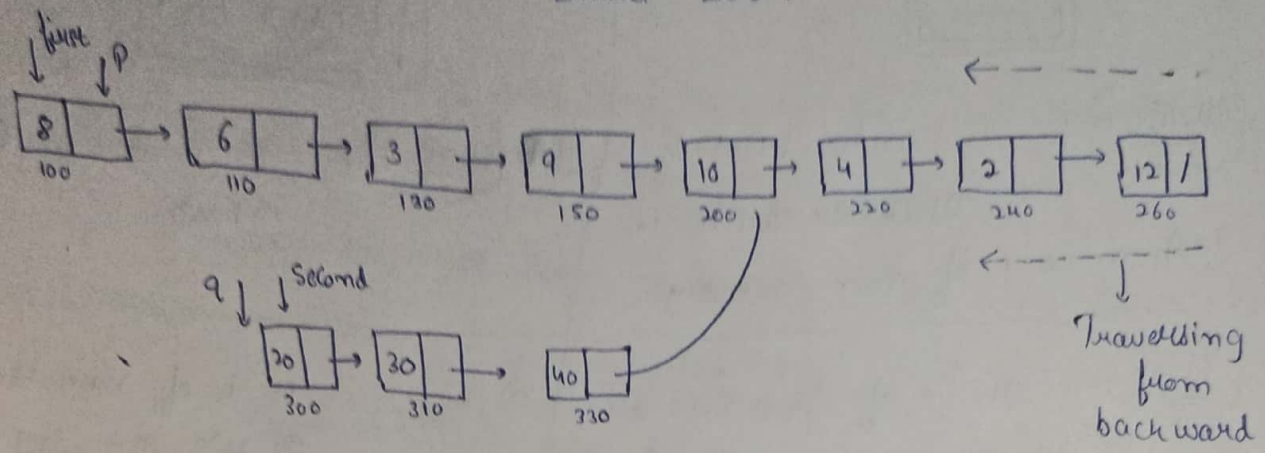
Comparison of Array with Linked List :-

ARRAY	LINKED LIST
① **Creation** :- It can be created in stack or can be in heap memory of main memory.	① **Creation** :- It is always created in heap.
② **Size** :- It is of fixed size or we can say it has fixed length.	② **Size** :- It is of variable size or we can say it has variable length.
③ chances of perfectly utilization is less as it may be large or may be small.	③ chances of perfectly utilization is high as it has variable size depending upon requirement
④ **When to use** :- when we know the no. of elements / length, then use Array.	④ **When to use** :- when we don't know about length then use Linked List.
⑤ **Space Consumption** :- It takes space only to store data (ie. int → 4 bytes, float → 4 bytes, etc)	⑤ **Space Consumption** :- It takes space for storing data as well as for storing pointers which points to next node, so it occupies more space
⑥ **Accessibility** :- we can access any element directly without any traversing. It can be accessed faster bcz it is formed in stack, but if it is formed in heap, then it will also take time but faster than list.	⑥ **Accessibility** :- we have to traverse through link then we can access that particular elements.

Scanned with CamScanner

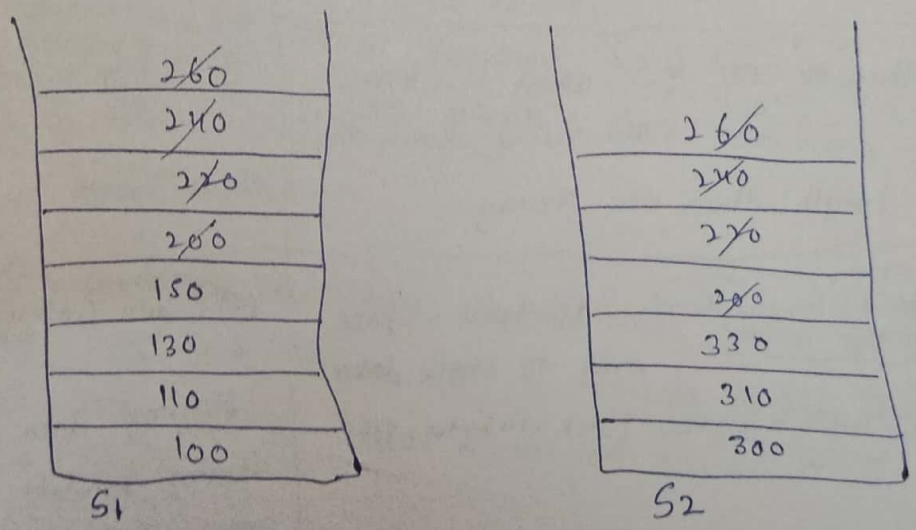
V.V.V.V. Imp

Student challenge - 1 :- Finding Intersecting point of Two Linked List.



Now, we know that we have to traverse backward but we can't traverse backward in Linear Singly Linked List, So, if we want to traverse, it can only be done by using Stacks, So that we get the Address of intersecting Node.

Pushing Addresses into Stack ← then poping out the Addresses



Poping out the Addresses means that whenever the Address is same delete it but also make a copy of that one and when the Addresses will be different, a previous Address from that Address was the intersecting point of two Linked lists, i.e. from this point two linked lists will be different.

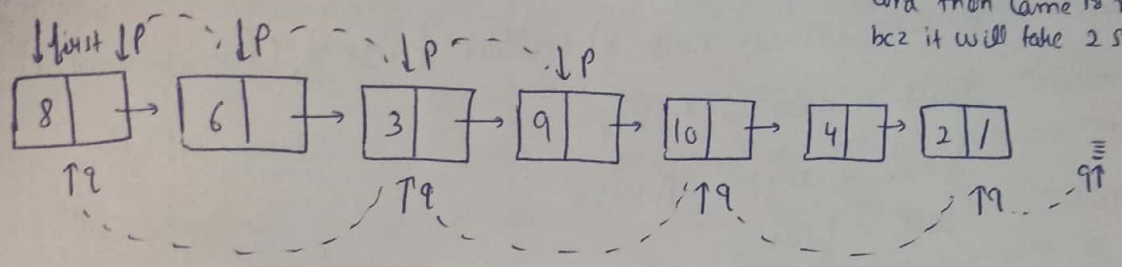
Scanned with CamScanner

Two de :-

Scanned with CamScanner

Scanned with CamScanner

dent challange -2 :- Finding middle Element of a Linked List. (*Best method) [instead of finding length and then came to middle bcz it will take 2 scans.]



- Move 'q' by two Nodes & P by one node, so distance travelled by 'p' is half of 'q', so 'P' will be in middle.

Code :-

```
Struct Node
{
  int data;
  Struct Node * next;
}* first = NULL;
int main()
{
  int N, i;
  printf (" Enter the no. of terms : \n");
  Scanf ("%d", &N);
  int A[N];
  printf ("Elements are : \n");
  for (i=0; i < N; i++)
    Scanf ("%d", &A[i]);
  Void create (int A[], int N);
  Void middle (struct Node *P);
  Create (A, N);
  middle (first);
}
```

Scanned with CamScanner

```
Void create (int A[], int N)
{ int i;
    Struct Node *t, *last;
    first = (struct Node *) malloc (sizeof (struct Node));
    first -> data = A[0];
    first -> next = NULL;
    last = first;
    for (i=1; i< N; i++)
      {
        t = (struct Node *) malloc (sizeof (struct Node));
        t -> data = A[i];
        t -> next = NULL;
        last -> next = t;
        last = t;
      }
  }
Void middle (Struct Node *P)
  {
      Struct Node *x, *q;
        x = q = first;
        while (q != NULL)
        {  q = q -> next;
           if (q != NULL)
              q = q -> next;
           if (q != NULL)
              P = P -> next;
        }
        printf (" Middle Element : %d", P -> data);
  }
```

Scanned with CamScanner

PRACTISE SET

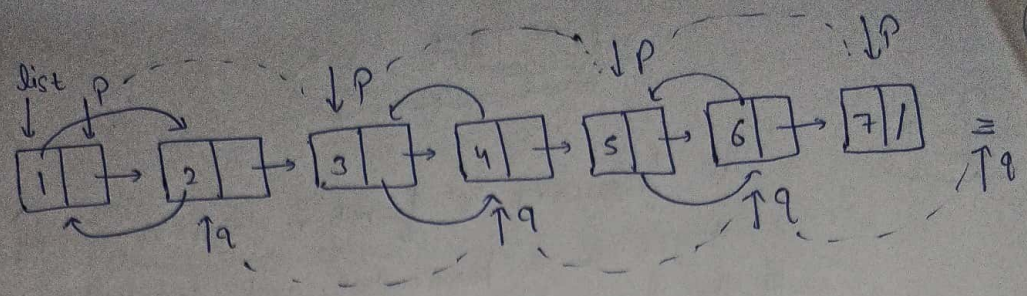
Ques - 1 :- The following 'c' function takes a Singly-Linked List of integers as a parameter and rearranges the elements of the list. The list is represented as pointer to a structure. The function is called with the list containing the integers 1, 2, 3, 4, 5, 6, 7 in the given order. what will be the contents of the list after the function completes execution?

```
Struct Node
{
    int value;
    Struct Node *next;
};
void rearrange (struct Node * list)
{
    Struct node *p, *q;
    int temp;
    if (! list || ! list -> next)
        return;
    p = list;
    q = list -> next;
    while (q)
    {
        temp = p -> value;
        p -> value = q -> value;
        q -> value = temp;
        p = q -> next;
        q = p ? p -> next : 0;
    }
}
```

→ it means if (p! = NULL)
q = p -> next;
else
q = NULL;

Scanned with CamScanner

Ans :- 1 :-



So, the ans. is 2, 1, 4, 3, 6, 5, 7

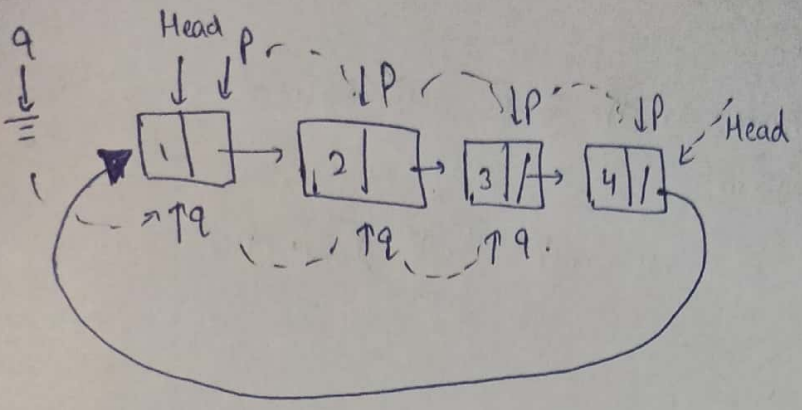
Ques-2 :- The following 'c' function takes a singly - linked list as input argument. It modified the list by moving the last element to the front of the list and returns the modified List. Some part of the Code is left blank.

```
type of struct Node
{
    int value;
    struct Node * next;
} node;

Node * node_to_front ( Node * Head)
{
    Node *P, *q;
    if ((head == NULL) || (head -> next == NULL))
            return head;
    q = NULL;
    P = head;
    while (P -> next != NULL)
            q = P;
            P = P -> next;
}
_____
    return Head;
}
```

Scanned with CamScanner

Choose the Correct Alternative to replace the blank line.

Answer-2 :-



(a) q = NULL; P→next = Head; Head = P;

(b) q→next = NULL; Head = P; P→next = Head;

(c) Head = P; P→next = q; q→next = NULL;

✓(d) q→Next = NULL; P→next = Head; Head = P;

Ques-3 :- Linked Lists are not Suitable data structures of which one of the following Problems?

Ans-3 :- (a) Insertion Sort (c) Radix Sort

✓(b) Binary Search (d) Polynomial manipulation

Ques-4 :- Consider the function f defined below.

```
struct item
{
  int data;
  struct item * next;
};
int f (struct item *P)
{
    Return ((P==NULL)|| (P→next == NULL) ||
    ((P→data <= P→ next→data) && f (P→next)));
}
```

For a given linked list P, the function f returns 1 if and only if.

Scanned with CamScanner

(a) The list is Empty or has exactly one element.

It means increasing order

(b) The elements in the list are sorted in non- decreasing order of data value.

(c) The Elements in the list are sorted in non- increasing order of data value.

(d) Not all Elements in the list have the same data value.

Scanned with CamScanner