'\t' → for tab
'\n' → for newline

ascii Chart

0 — 48	A — 65	a — 97
1 — 49	B — 66	b — 98
2 — 50	C — 67	c — 99
3 — 51	D — 68	d — 100
4 — 52	E — 69	e — 101
:	F — 70	:
:	:	:
9 — 57	Z — 90	z — 122

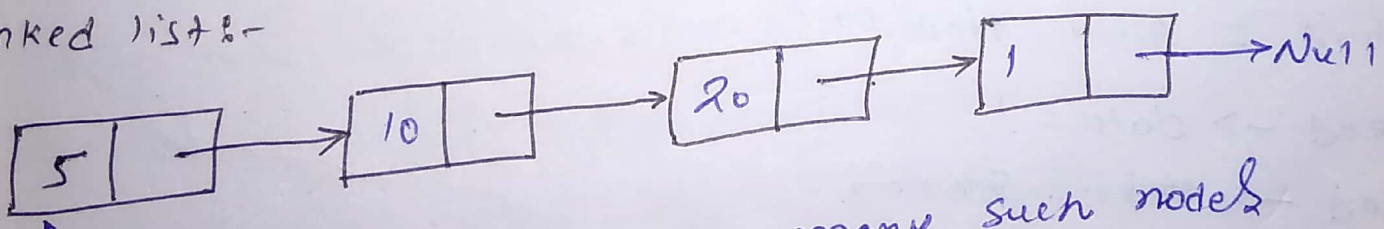
Palindrome :-
Word which is symmetrical from front and end.
9 1019
9 1419
Ompraksht shakarpno

Single linked list

it is a way to store a collection of elements. like an array these can be integers or characters. Each element in a linked list is stored in form of node.

Node :- | Data 10 | Next Pointer | } Node Collection of two sub-elements or parts.

Linked list :-



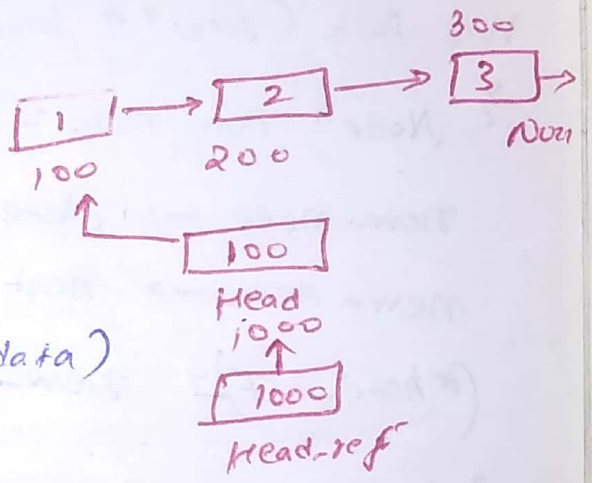
A linked list is formed when many such nodes are linked together to form a chain. First node is always used as a reference to traverse the list and is called Head. Last node points to null.

Creation of linked list with 3 Node:-

```
#include <bits/stdc++.h>
using namespace std;
class Node
{ public:
    int data;
    Node* next;
};
int main()
{
  Node* head = NULL;
  Node* second = Null;
  Node* third = NULL;
  // allocate three nodes in heap
  head = new Node();
  second = new Node();
  third = new Node();
  head -> data = 1;
  head -> next = second;
  second -> data = 2;
  second -> next = third;
  third -> data = 3;
  third -> next = NULL;
  return 0;        PrintList(head);
}
```

// for printing Contained of list write function
printlist ( )

```
Void printlist( Node * n)
{ while (n != NULL)
  {
  cout << n -> data << "  ";
  n = n -> next ;
  }
}
```



insert Node in linked list

1> At front of linked list

```
Void Push ( Node ** head_ref , int new_data)
{ /allocate node */
  Node * new_node = new Node ();
  // put data
  new_node -> data = new_data;
  new_node -> next = (* head_ref) ;
  (*head_ref) = new_node ;
```

Time Complexity O(1)

```
}
```

2> insert b/w node

```
#include <bits/stdc++.h>
Using namespace std;
Class Node{
Public:
    int data;
    Node * next;
};
Void Push (Node ** head_ref , int new_data)
{ Node* new_node = new Node();
  new_node -> data = new_data;
  new_node -> next = (* head_ref);
  (*head_ref)= new_node;
};
Void insertAfter (Node * Prev_node , int new_data)
{ Node* new_node = new Node();
  if (Prev_node == NULL)
  { cout << " the given previous node cannot be NULL";
    return;
  }
new_node -> data = new_data;
new_node -> next = Prev_node -> next;
Prev_node -> next = new_node;
}
```

time :- O(1)

```
void append (Node ** head_ref, int new_data)
{ Node * new_node = new Node ();
  Node * last = *head_ref;
  new_node -> data = new_data;
  new_node -> next = NULL;
  if (*head_ref == NULL)
    { *head_ref = new_node;
      return;
    }
  while (last -> next != NULL)
      last = last -> next;
  last ->next = new_node;
  return;
  }
void printlist ( Node *node)
{ while ( node != NULL)
  { cout<< " " << node -> data;
    node = node -> next;
  }
}
int main ()
{ Node *head = NULL;
  append (&head, 6);
  Push (&head, 7); push (&head, 2);
  Append (&head, 4); insert ( head -> next, 8);
  Printlist (head); return 0;
}
```

Output:-
1 2 8 6 4

// function to delete linked list

```
void deletelist (Node** head_ref)
{
    Node* current = *head_ref;
    Node* next;
  while (current != NULL)
    { next = current->next;
      free(current);
      current = next;
    }*head_ref = NULL;
}
```

Length of a linked list (iterative & Recursive)

iterative methode ——

```
int getCount (Node* head)
{ int count = 0;
  Node* current = head;
  while (current != NULL)
    { count++;
      current = current->next;
    }
  return count;
}
```

Call:- getCount(head);

Recursive Soln :-

```
int getCount (Node * head)
{ if (head == NULL)
    return 0;
  Else
    return 1 + getCount (head -> next);
}
```

Search an element in a linked list

1> iterative Soln :-

```
bool search (Node* head, int x)
{ Node* current = head;
  while (current != NULL)
  { if (current -> data == x)
      return true;
    current = current -> next;
  }
  return false;
}
```

Recursive Soln :-

```
bool search (Node *head, int x)
{ if (head == NULL)
    return false;
Else if (head -> data == x)
    return true;
Else
    return search (head -> next, x);
}
```

Function to get Nth node in linked List

GetNth() function takes a linked list and integer index returns data valued stored in node at given index.

1) iterative Soln :-

```
int getNth (Node *head , int index)
{ Node * Current = head;
  int Count = 0;
  while (Current != NULL)
  { if (count == index)
        return (Current -> data);
    Count ++;
    Current = Current -> next;
  }
  assert(0);
}
```

2) Recursive Soln :-

```
int getNth (Node * head ,int index)
{   int Count=0;
    if (head == NULL)
        assert(0);
    else if (index == Count)
        return head -> data;
    else return GetNth (head -> next, index-1);
}
```

function to get Nth node from end of linked list

```
Void printNthfromlast ( Node * head, int n )
{ int len=0 , i;
  Node* temp = head;
  while ( temp !=NULL) {
    temp = temp -> next;
      len++;
  }
  if (len< n)
    return;
temp = head;
for (i=1 ; i< len- n+1; i++)
   temp  = temp -> next;
   Cout<< temp -> data;
return;
}
```

Time Complexity :- O(n)

method-2 (Use two pointers)

```
void printNthfromlast ( Node * head, int n)
{ Node * main_ptr = head;
  Node* ref_ptr = head;
  int Count=0
  if (head !=NULL)
  { while (Count <n)
    { if (ref_ptr == NULL)
      { Cout<<n<<"is greater than no. of nodes";
        return ;
```

```
        ref_ptr = ref_ptr -> next;
        count++;
    }
while (ref_ptr != NULL)
{ main_ptr = main_ptr -> next;
  ref_ptr = ref_ptr -> next;
}
cout << main_ptr -> data;
}}
```

Print middle of given linked list if odd length if even length print second middle element

Method 1 :- Using length print length/2 + 1 Node ~~if len Odd~~

Method 2 :- Two pointer method (increase one pointer by 2, increase other " by 1)

↓ this give middle elemen

```
void printmiddle (Node *head)
{ Node * slow_ptr = head;
  Node * fast_ptr = head;
  if (head != NULL)
  { while ( fast_ptr != NULL && fast_ptr -> next != NULL)
    { fast_ptr = fast_ptr -> next -> next;
      slow_ptr = slow_ptr -> next;
    }
    ~~printf( slow~~ cout << slow_ptr -> data;
}}
```

Method-3

```
Void PrintMiddle (Node * head)
{ int Count = 0;
  Node * mid = head;
  While (head != NULL)
  { if (Count%2)
        mid = mid -> next;
    ++Count;
    head = head -> next;
  }
  if (mid != NULL)
    cout << mid -> data;
}
```

Change mid if Count is odd so it takes half step of head.

Function Counts Number of times a given int occurs in linked list.

Method-1 iterative Soln :-

```
int Count (Node * head, int Search_int)
{ Node * Current = head;
  int Count = 0;
  while (Current != NULL)
  { if (Current -> data == Search_int)
        Count ++;
    Current = Current -> next;
  }
  Return Count;
}
```

Time Complexity O(n)

Auxiliary space O(1)

Method 2 :- Using Recursion

```
int frequency = 0;  → global variable declare
int Count (Node * head, int Search_element)
{ else if (head -> data == Search_element)
        frequency ++;
  if (head == NULL)
      Return frequency;
  else return Count (head -> next, Search_element);
}
```

without global variable :-

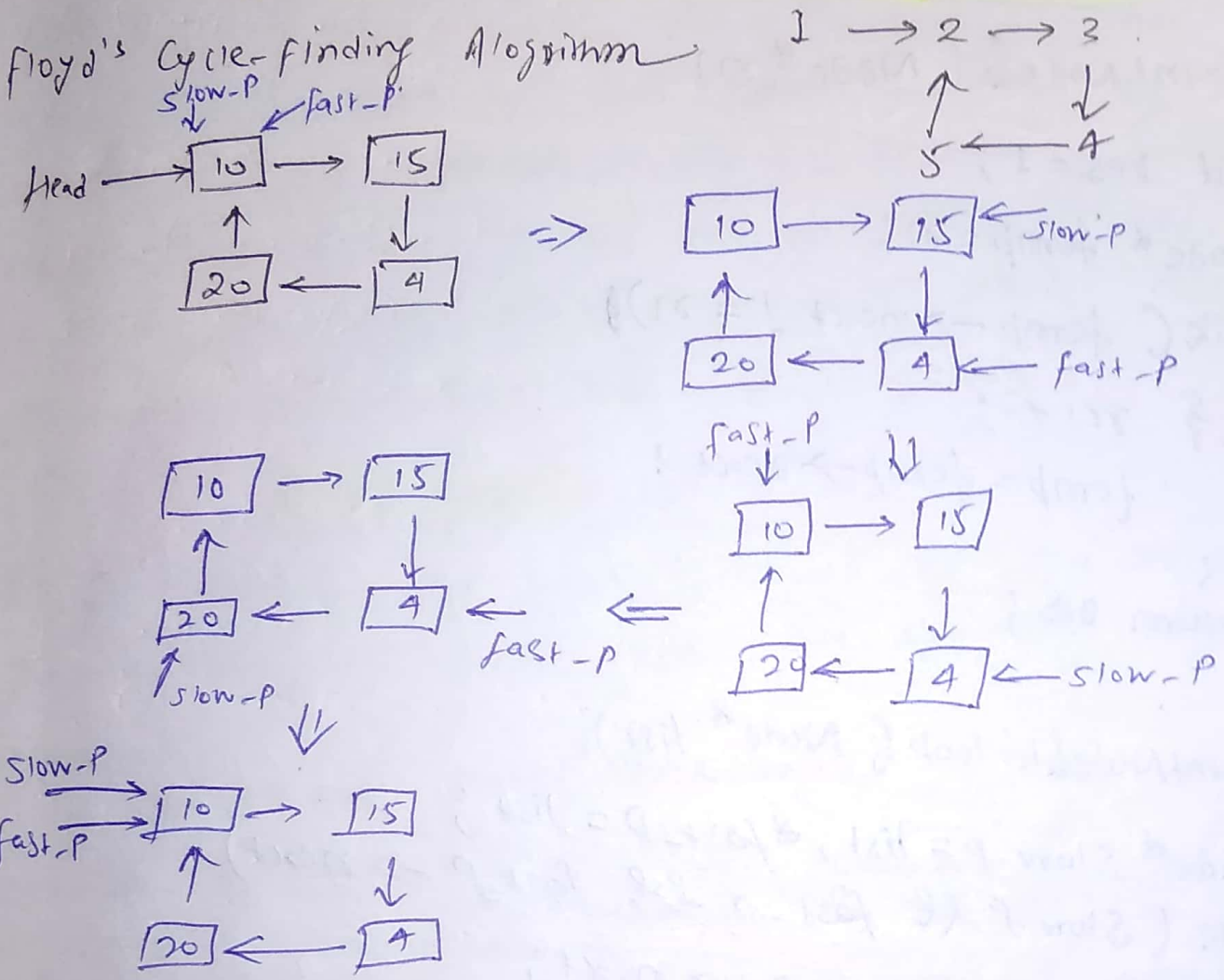
```
int Count (Node * head, int Search)
{ if (head == NULL)
      Return 0;
  Else if (head -> data == Search)
      return 1 + Count (head -> next, Search);
  Else
      return Count (head -> next, search);
}
```

Time Complexity in each O(n).

Space " O(n)

Detect loop in linked list

Floyd's Cycle-finding Algorithm



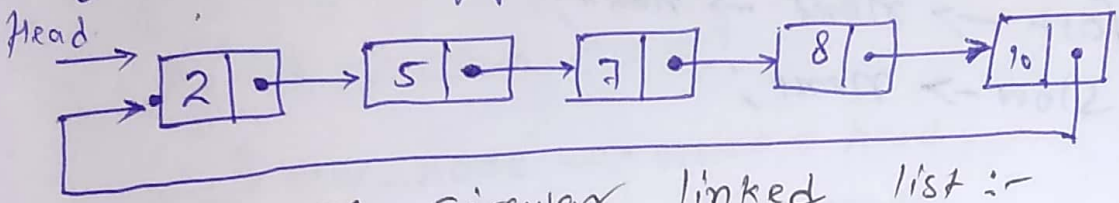

```
int detectloop (Node* list)
{ Node* slow_p = list, *fast_p = list;
  while (slow_p && Fast_p && fast_p -> next)
  { slow_p = slow_p -> next;
    fast_p = fast_p -> next -> next;
    if (slow_p == fast_p)
        Return 1;
  }
  return 0;
}
```

find length of loop in linked list

```
int countNodes(Node* n)
{ int res = 1;
  Node* temp = n;
  while(temp -> next != n)
    { res++;
      temp = temp -> next;
    }
  return res;
}
int countNodesinloop(Node* list)
{ Node* slow_p = list, *fast_p = list;
  while(slow_p && fast_p && fast_p -> next)
    { slow_p = slow_p -> next;
      fast_p = fast_p -> next -> next;
      if(slow_p == fast_p)
        return countNodes(slow_p);
    }
  return 0; // indicate that their is no loop
}
```

Circular linked list

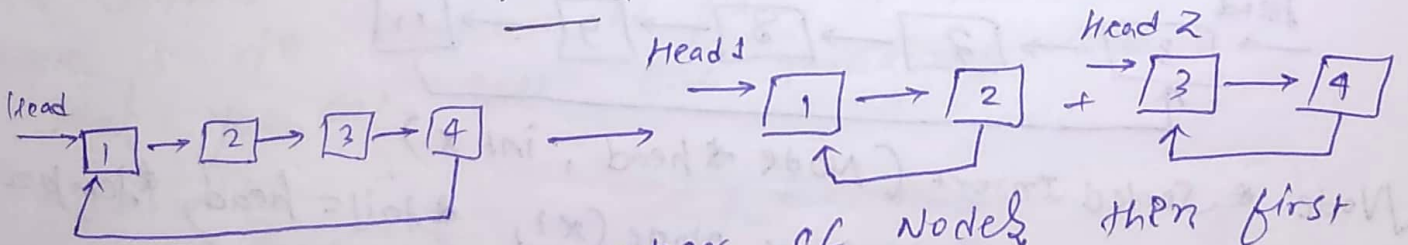
A linked list where all nodes are connected to form a circle. there is no NULL at end. It can be singly or doubly Circular linked list.



traversal of Circular linked list :-

```
Void printlist (struct Node * first)
{ struct Node * tmp = first;
  if (first != NULL)
  { printf ("%d", temp -> data);
    temp = temp -> next;
  while (temp != first)
  { printf ("%d", temp -> data);
    temp = temp -> next;
  }
  }
}
```

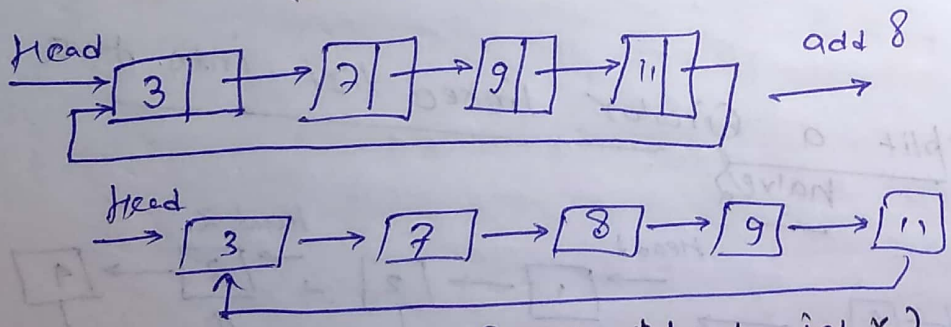
Split a Circular linked list into two halves



if there are odd number of Nodes then first list Contain One extra Nodes.

```
Void Split list ( Node *head, Node **head1_ref, Node **head2_ref)
{ Node *slow = head, *fast = head, *temp = NULL;
  while (fast -> next -> next != head && fast -> next != head)
  { fast = fast -> next -> next;
    slow = slow -> next;
  }
*head2_ref = slow->next;
if (fast -> next == head)
      fast -> next = *head2_ref;
  else fast -> next -> next = *head2_ref;
slow -> next = head;
*head1_ref = head;
}
```

Insert New data into Circular linked list




```
Node * Sorted Insert (Node *head , int x)
{ Node * new_node = new Node (x), *tail = head, *temp = head;
  while (tail -> next != head)
      tail = tail -> next;
```

ref) if (head -> data >= x)

{ new_node -> next = head;

head) tail -> next = new_node;

return new_node;

}

if (tail -> data <= x)

{ new_node -> next = head;

tail -> next = new_node;

return head;

}

while (temp->next != head)

{ if (temp -> next -> data >= x)

{ new_node -> next = temp -> next;

temp -> next = new_node;

return head;

}

temp = temp -> next;

}

Josephus Circle using Circular linked list

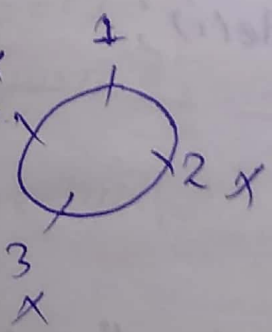
Given the total number of persons n in a circle and a number m which indicates that m person are skipped and m^{th} person is killed in circle. Task is to choose the place in initial circle so that you are last one remaining and so survive.

head;

input:- n=4

m=2

Output :- 1



input :- n=5

m=3

Output :- 4

i) Create a Circular linked list of size n.
ii) Traverse through linked list and one by one delete every mth node until there is one node left.
iii) Return value of only left node.

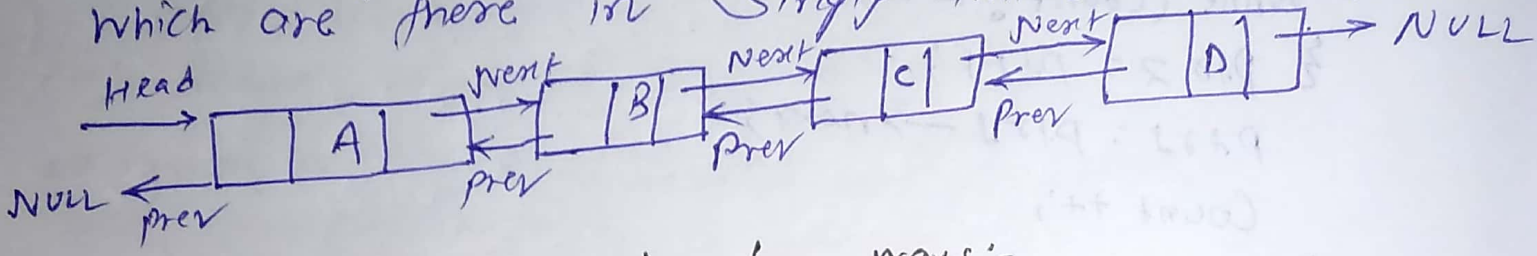
```
#include <bits/stdc++.h>
using namespace std;
Struct Node
{ int data;
  Struct Node * next;
};
Node *newNode (int data)
{ Node * temp = new Node;
  temp -> next = NULL;
  temp -> data = data;
  return temp;
}
Void getJosephusPosition ( int m, int n)
{ Node *head = newNode(1);
  Node *prev = head;
  for (int i=2; i<=n; i++)
    { prev -> next = newNode(i);
      Prev = prev -> next;
    }
  Prev -> next = head;
//while only one node is left in linked list
 Node *ptr1 = head, *ptr2 = head;
```

```
while (ptr1 -> next != ptr1)
{ //find m-th node
   int count=1;
   while (count != m)
   { ptr2 = ptr1;
     ptr1 = ptr1 -> next;
     count ++;
   }
  //Remove m-th node
  ptr2 -> next = ptr1 -> next;
  free (ptr1);
  ptr1 = ptr2 -> next;
}
Printf ("%d", ptr1->data);
}
int main ()
{  int n=14, m=2;
   getjosephusposition (m,n);
   return 0;
}
int jos (int n; int k)
{  if (n==1)
      return 0;
   else return (jos (n-1, k)+k) %n;
}
```

Time complexity : O(m*n)

Doubly Linked List (intro and insertion)

It contains an extra pointer, typically called previous pointer, together with next pointer and data which are there in singly linked list: next



A node can be added in four ways:-

i) At front of Double linked list.

ii) After a given node.

iii) At end of DLL.

iv) before a given node.

Add a Node at front (5 step process)

```
Void Push (struct Node ** head_ref, int new_data)
{ struct Node * new_node = new Node ;
    new_node -> data = new_data;
    new_node -> next = * head_ref;
    new_node -> prev = NULL;
    if ( (*head_ref) != NULL )
        *head_ref -> Prev = new_node;
    (*head_ref) = new_node;
}
```

Add a Node after a given node (7 step)

```
Void insertAfter ( struct Node * Prev_node, int new_data)
{   if ( Prev_node == NULL)
        Return ;
    Struct Node * new_node = new Node ;
```

```
new_node -> data = new_data;
new_node -> next = prev_node -> next;
new_node -> prev = prev_node;
prev_node -> next = new_node;
// change previous of (new_node's next node)
if (new_node -> next != NULL)
    new_node -> next -> prev = new_node;
}
```

Add a node at end (7 step)

```
Void append (struct Node **head_ref, int data)
{ struct Node * new_node = new Node;
  new_node -> data = data;
  new_node -> next = NULL;
  struct Node * last = *head_ref;
  if (*head_ref == NULL)  // if linked list is empty.
                          //make new node as head;
  { new_node -> prev = NULL;
    *head_ref = new_node;
    return;
  }
  while (last -> next != NULL)
      last = last -> next;

  last -> next = new_node;
  new_node -> prev = last;
  return;
}
```

Add a node before a given node.

```
Void insertbefore (struct Node **head_ref, struct Node *next_node,
                   int new_data)
{
  if (next_node == NULL)
      return;
  Struct Node * new_node = new Node;
  new_node -> data = new_data;
  new_node -> prev = next_node -> prev;
  next_node -> prev = new_node;
  new_node -> next = next_node;
  if (new_node -> prev != NULL)
      new_node -> prev -> next = new_node;
  Else
      *head_ref = new_node;
}
```

Delete a node in Doubly linked list

Deletion of node in doubly linked list divided in three main Categories :-

i) Deletion of head node
ii) Deletion of middle "
iii) Deletion " end "

```
Void deleteNode (Node **head_ref, Node *del)
{
  if (del == *head_ref)
    { *head_ref = pos -> next;
      pos -> next -> prev = NULL;
      free (pos); return; }
```

```
if (pos -> next == NULL)
    { pos -> prev -> next = NULL;
      free(pos);
      return;
    }
pos -> prev -> next = pos -> next;
pos -> next -> prev = pos -> prev;
free(pos);
return;
}
```

node,

Reverse a doubly linked list

```
Void reverse (Node **head_ref)
{ Node *temp = NULL;
  Node *Current = *head_ref;
  while (Current != NULL)
  { temp = Current -> prev;
    Current -> prev = Current -> next;
    Current -> next = temp;
    Current = Current -> prev;
  }
  // Before Changing the head, check for case like empty
     // list and list with only one node.
  if (temp != NULL)
      *head_ref = temp -> prev;
}
```

Here Concept is to Swap prev and next pointers of all nodes, Change prev of head (or start) and Change head pointer in end.

Merge Sort for Doubly linked list

```
Node *merge (Node *first, Node *second)
{ if (first == NULL)
      return second;
  if (second == NULL)
      return first;
  if (first -> data < second -> data)
  { first -> next = merge (first->next, second);
      first -> next -> prev = first;
      first -> prev = NULL;
      return first;
  }
  else
  { second -> next = merge (first, second -> next);
    second -> next -> prev = second;
    second -> prev = NULL;
    return second;
  }
Node *split (Node *head)
  { Node *fast = head, *slow = head;
    while (fast -> next && fast -> next -> next)
    { fast = fast -> next -> next;
      slow = slow -> next;
    }
    Node *temp = slow -> next;
      slow -> next = NULL;
      return temp;
  }
```

```
Node *mergesort(Node *head)
{   if (!head || !head->next)
        return head;
    Node *second = split(head);
    head = mergesort(head);
    second = mergesort(second);
    return merge(head, second);
}
```

Find Pairs with given sum in doubly linked list

Given a Sorted doubly linked list of positive distinct elements, the task is to find pairs in doubly LL whose sum is equal to given X. without using an extra space.

Input :- 1 <-> 2 <-> 4 <-> 5 <-> 6 <-> 8 <-> 9

X = 7

Output :- (6,1), (2,5)

Expected time complexity :- O(n)

Auxiliary space :- O(1)

```
Void pairSum(Node *head, int x)
{   struct Node *first = head;
    "    "    *second = head;
    while(second->next != NULL)
        second = second->next;
    bool found = false;
    while(first != NULL && second != NULL &&
          first != second && second->next != first)
    {   if((first->data + second->data) == x)
        {   found = true;
            cout << first->data << ", " << second->data
                 << endl;
            first = first->next;
```

```
        second = second -> prev;
    }
    else
    { if ((first -> data + second -> data) < x)
            first = first -> next;
      Else
            second = second -> prev;
    }
}
if (found == false)
    cout << "No Pair found";
}
```