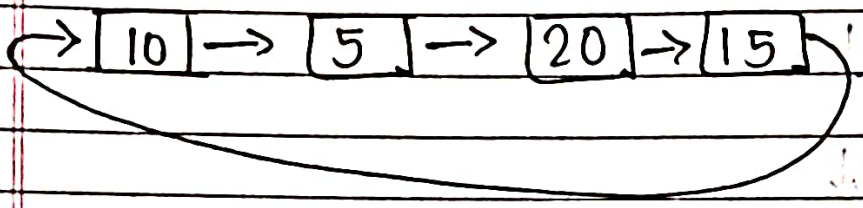
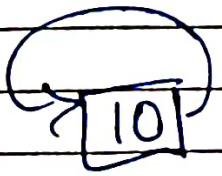
CIRCULAR LINKED LIST




```
struct Node {
int  data;
Node * next;
Node (int x) { int data = x;
                next = NULL; }
};
```

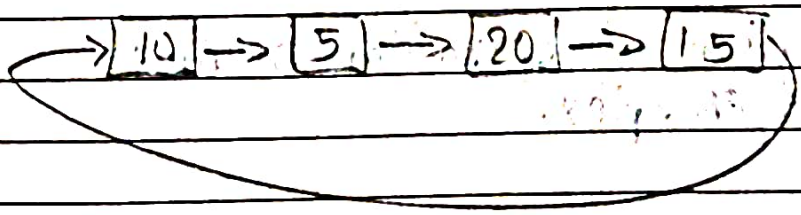
Circular linked list can be of singly linked list or a Doubly linked list.



head -> next = head

⇒

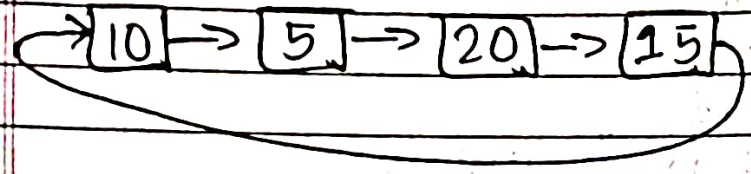
```
struct Node {
    int data;
    Node *next;
    Node (int d) { data = d;
                   next = NULL; }

};

int main() {
Node *head = new Node (10);
head -> next = new Node (5);
head -> next -> next = new Node (20);
head -> next -> next -> next = new Node (15);
head -> next -> next -> next -> next = head;
return 0; }
```



This is not the right way though.

Date
Page

Circular linked list (Advantages & Disadvantages)



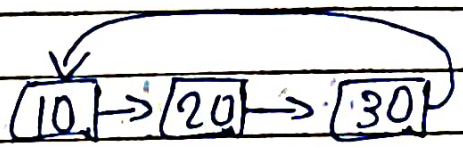
⇒ Advantages:-

1) We can traverse the whole list from any node.
2) Implementation of algorithms like round robin.
3) We can insert at the beginning & end by just maintaing one tail reference/ pointers

⇒ Disadvantages:-

1) Implementation of operation becomes ~~complicated~~ complex.

* Circular linked list Traversed :-

1) I/P :- 

O/P :- 10 20 30

2) I/P :- 

O/P :- 10

3) I/P :- NULL

O/P :-

⇒ Method I :- for loop.

```
void printlist (Node *head) {
  if (head == NULL) {return;}
     cout << (head -> data) << " ";
  for (Node *p = head -> next ; p != head; p = p -> next)
     { cout << (p -> data) << " " ; }
}
```

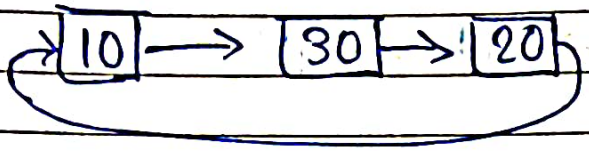
Time :- O(n) Space O(1)

⇒ Method 2 :- do while

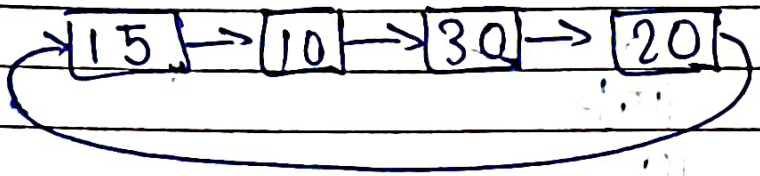
```
void printlist : (Node *head) {
  if (head == NULL) { return ; }
    Node *P = head;
do { cout << (P -> data) << " "; P = P -> next;}
while (p != head);
}
```

Time :- O(n)
Space :- O(1)

⊛ Insert at Begin of Circular linked list :-

1) I/P :- 

, x = 15

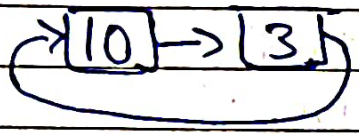
O/P :- 

2) I/P :- NULL , x = 10

O/P :- 

3) I/P :- 

, x = 3

O/P :- 

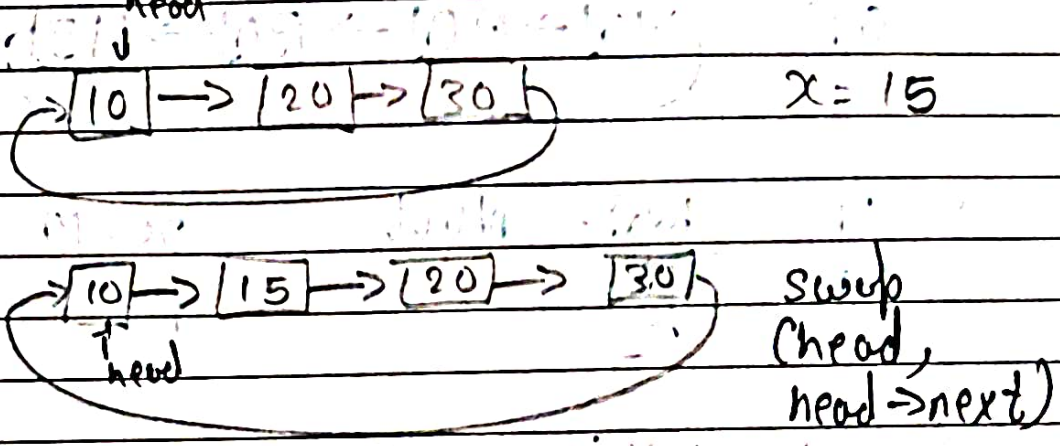
⇒ Naive :-

T.C :- O(n)

```
Node *insertBegin (Node *head, int x)
{ Node *temp = new Node (x);
  if (head == NULL)
      temp -> next = temp;
  else
  { Node *curr = head;
    while (curr -> next != head)
        {Curr = curr -> next; }
    curr -> next = temp;
    temp -> next = head;
  }
  return temp;
    }
```

⇒ Efficient Soln:-

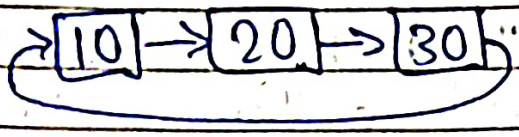
idea:- To insert the block just after the head or after first element & then swap with head.



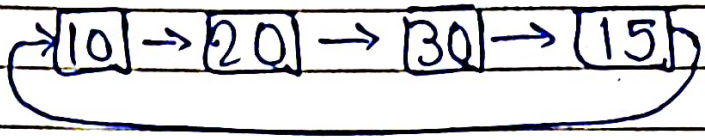
⇒

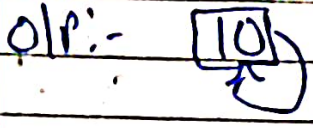
```
Node *insertBegin (Node *head, int x) {
Node *temp = new Node(x);
if (head == NULL)
  { temp->next = temp;
      return temp; }
else
   { temp->next = head->next;
     head->next = temp;
     int t = head->data;
     head->data = temp->data;
     temp->data = t;
     return head;
   }
}
```

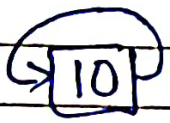
⊛ Insert at the End of Circular linked list :-

1) I/P :- 

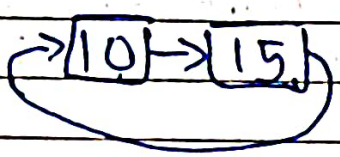
x = 15

O/P :- 

2) I/P :- head = NULL x = 10
O/P :- 

3) I/P :- 

x = 15

O/P :- 

→ Naive Soln :-

```
Node *insertEnd (Node *head, int x)
{ Node *temp = new Node (x);
   if(head == NULL) {
       temp->next = temp
        return temp; }
   else
   { Node *curr = head;
      while (curr -> next != head)
            { curr = curr ->next; }
      curr -> next = temp;
      temp -> next= head;
      return head;
    }
}
```

⇒ Efficient :-

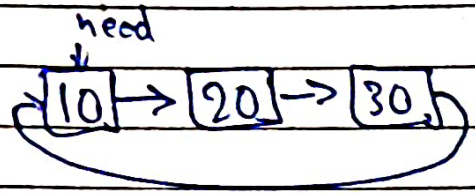
One way is to use a tail pointer.

Other way is :-

Is to insert the node just after head node once inserted.

⇒
```
Node *insertEnd (Node *head, int x) {
Node * temp = new Node (x);
if (head == NULL)
{ temp -> next = temp;   return temp; }
else
{ temp -> next = head -> next; } Insert temp after
  head -> next = temp;            head
  int t = temp -> data;
  temp -> data = head -> data; } Swapping.
  head -> data = t;
  return temp; } Temp is new head.
}
}
```

⊛ Delete head of circular linked list:-

1) I/P:- 

O/P:- 

2) I/P:- 

O/P:- NULL

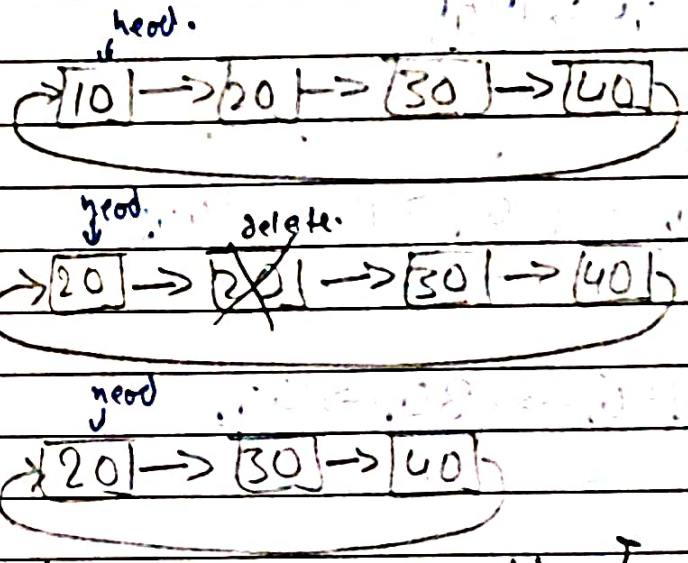
3) I/P:- head = NULL
O/P:- NULL.

⇒ Naive Soln:-

```
Node *delHead (Node *head) {
if(head == NULL) return NULL;
if(head->next == head)
{ delete head; return NULL; }
Node *curr = head;
while(curr->next != head)
        {curr = curr->next; }
curr->next = head->next;
 delete head;
 return (curr->next);
 }
```

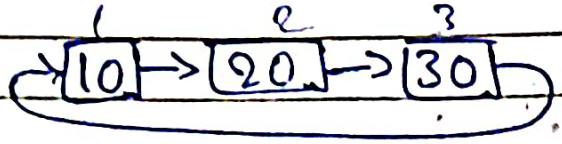
⇒ Efficient Soln:-

Copy the data of head->next to head once copied delete (head->next)

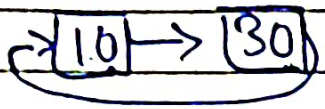


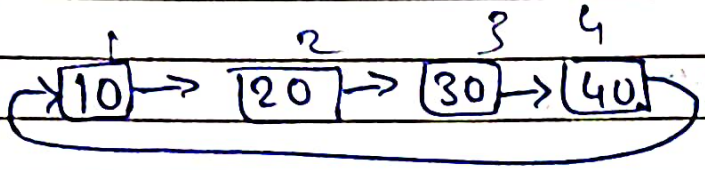

```
Node *delHead (Node *head) {
if(head == NULL)  return NULL;
if(head->next == head)
{ delete head;  return NULL; }
head->data = head->next->data;
Node *temp = head->next;
head->next = head->next->next;
delete temp;
return head;
}
```

⑧ Delete Kth Node from a circular linked list

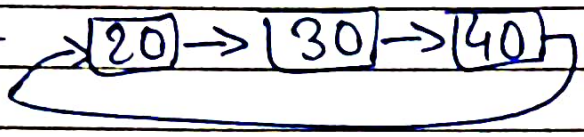
1) I/P:- 

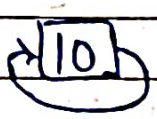
, K=2

O/P:- 

2) I/P:- 

K=1

O/P:- 

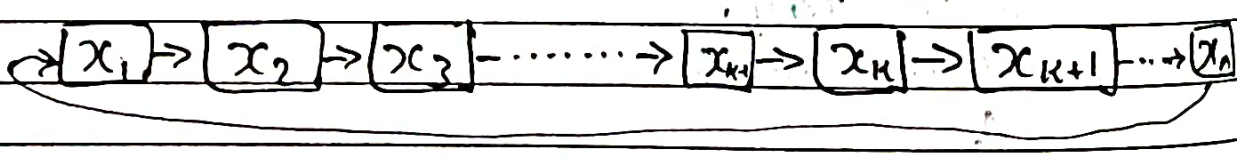
3) I/P:- 

K=1

O/P:- head = NULL.

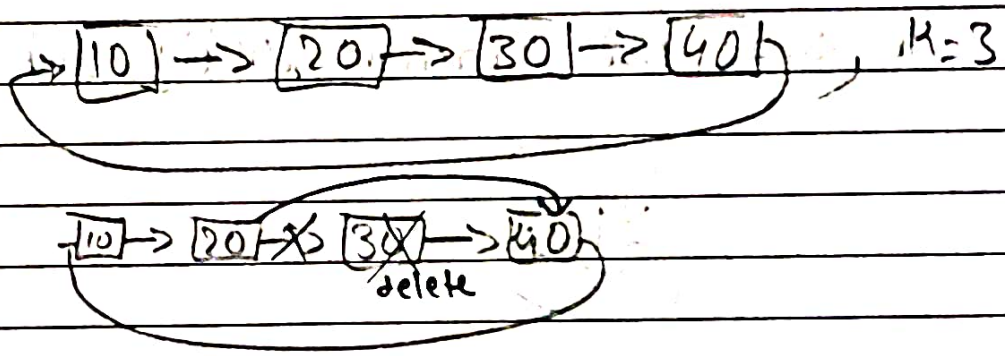
⇒ No. of node >= K

⇒ When K is not 1 :-

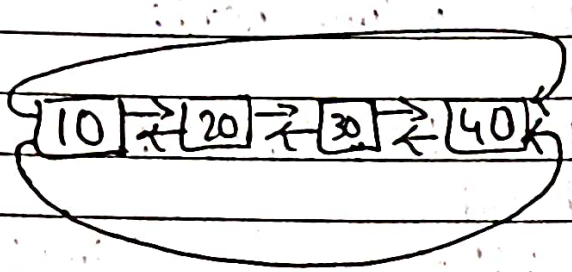


⇒

```
Node *deleteKth(Node *head, int k)
{ if (head == NULL) return head;
  if (k==1)
     {return deleteHead(head);}
Node *curr = head;
for (int i=0; i<k-2; i++)        Time :-
     {curr = curr->next;}
Node *temp = curr->next;
curr->next = curr->next->next;
delete temp;
return head; }
```



⑧ Circular Doubly linked list:-

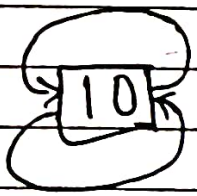


① Previous of head is last node.
② Next of last node is head.

An empty Circular Doubly linked list

head = null or NULL,

A single Node Circular Doubly link



① We get all advantages of circular & doubly linked list.

② We can access last node in constant time without maintaing extra tail pointer/references.

⊛ Insert at Head :-

```
Node *insertBegin (Node *Head, int x) {
  Node *temp. = new Node(x.);
  if (head == NULL)
    { temp -> next = temp;
      temp -> prev = temp;
      return temp; }

  head -> prev -> next = temp;
  temp -> next = head;
  head -> prev = temp;
   retun temp; }
```

⊛ Insert at End :-

To insert at the end some function just return head at last.