[Date]

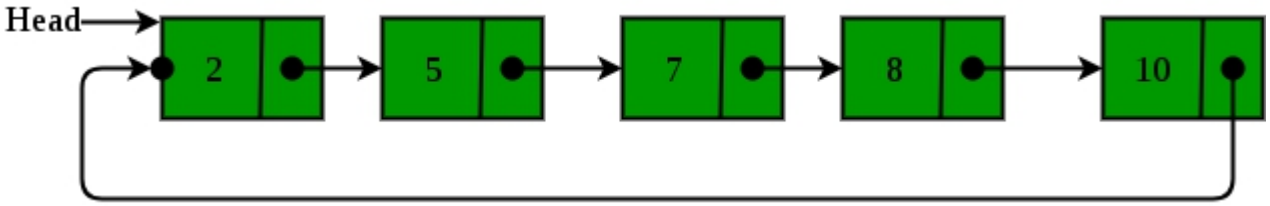
Complete placement course

Circular linked list

ANIL KUMAR DWIVEDI
BADA VIDYALAY

Circular Linked List :

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



Advantages of Circular Linked Lists:

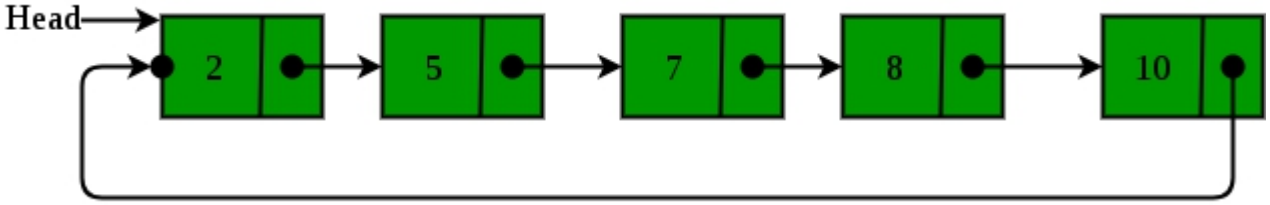
1) Any node can be a starting point. We can traverse the whole list by starting from any point. We just need to stop when the first visited node is visited again.

2) Useful for implementation of queue

3) Circular lists are useful in applications to repeatedly go around the list. For example, when multiple applications are running on a PC, it is common for the operating system to put the running applications on a list and then to cycle through them, giving each of them a slice of time to execute, and then making them wait while the CPU is given to another application. It is convenient for the operating system to use a circular list so that when it reaches the end of the list it can cycle around to the front of the list.

4) Circular Doubly Linked Lists are used for implementation of advanced data structures like Fibonacci Heap.

Circular Linked List | Set 2 (Traversal)



In a conventional linked list, we traverse the list from the head node and stop the traversal when we reach NULL. In a circular linked list, we stop traversal when we reach the first node again. Following is the C code for the linked list traversal.

```
/* Function to print nodes in
a given Circular linked list */
void printList(Node* head)
{
    Node* temp = head;

    // If linked list is not empty
    if (head != NULL) {

        // Print nodes till we reach first node again
        do {
            cout << temp->data << " ";
            temp = temp->next;
        } while (temp != head);
    }
}

// This code is contributed by rajsanghavi9

```

Complete program to demonstrate traversal. Following are complete programs to demonstrate traversal of circular linked list.

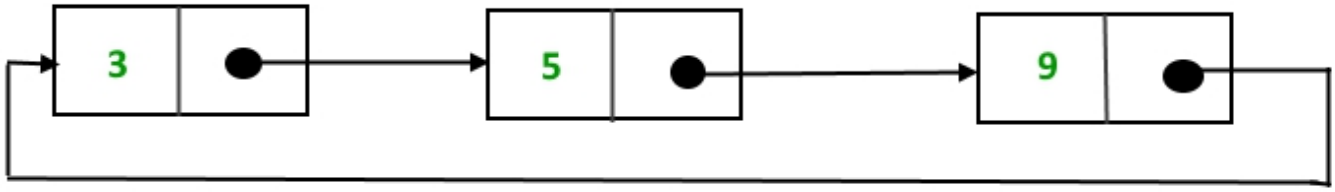
```
// C++ program to implement
// the above approach
#include <bits/stdc++.h>
using namespace std;

/* structure for a node */
class Node
{
    public:
    int data;
    Node *next;
};

/* Function to insert a node at the beginning
of a Circular linked list */
void push(Node **head_ref, int data)
{
    Node *ptr1 = new Node();
    Node *temp = *head_ref;
    ptr1->data = data;
    ptr1->next = *head_ref;

    /* If linked list is not NULL then
    set the next of last node */
    if (*head_ref != NULL)
    {
        while (temp->next != *head_ref)
```

```
            temp = temp->next;
        temp->next = ptr1;
    }
    else
        ptr1->next = ptr1; /*For the first node */

    *head_ref = ptr1;
}

/* Function to print nodes in
a given Circular linked list */
void printList(Node *head)
{
    Node *temp = head;
    if (head != NULL)
    {
        do
        {
            cout << temp->data << " ";
            temp = temp->next;
        }
        while (temp != head);
    }
}

/* Driver program to test above functions */
int main()
{
    /* Initialize lists as empty */
    Node *head = NULL;

    /* Created linked list will be 11->2->56->12 */
    push(&head, 12);
    push(&head, 56);
    push(&head, 2);
    push(&head, 11);

    cout << "Contents of Circular Linked List\n ";
    printList(head);

    return 0;
}

```

Output:

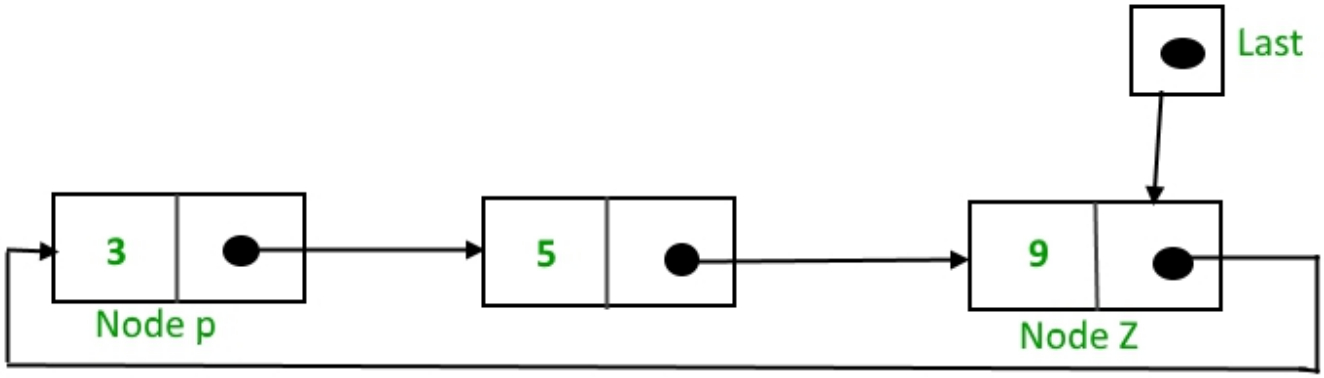
```
Contents of Circular Linked List
11 2 56 12
```

Circular Singly Linked List | Insertion



Implementation

To implement a circular singly linked list, we take an external pointer that points to the last node of the list. If we have a pointer last pointing to the last node, then last -> next will point to the first node.



The pointer *last* points to node Z and last -> next points to node P.

Why have we taken a pointer that points to the last node instead of the first node?

For the insertion of a node at the beginning, we need to traverse the whole list. Also, for insertion at the end, the whole list has to be traversed. If instead of *start* pointer, we take a pointer to the last node, then in both cases there won't be any need to traverse the whole list. So insertion at the beginning or at the end takes constant time, irrespective of the length of the list.

Insertion

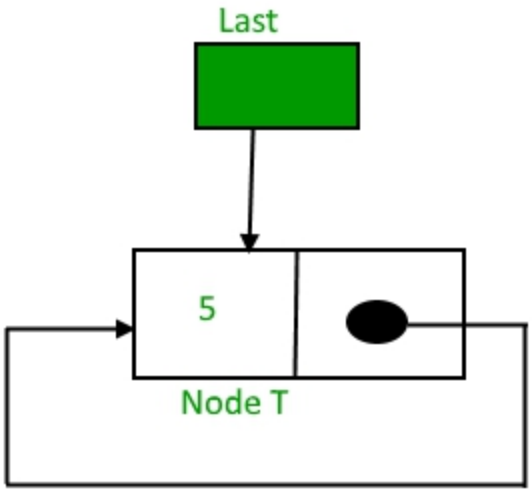
A node can be added in three ways:

- Insertion in an empty list
- Insertion at the beginning of the list
- Insertion at the end of the list
- Insertion in between the nodes

Insertion in an empty List

Initially, when the list is empty, the *last* pointer will be NULL.

After inserting node T,



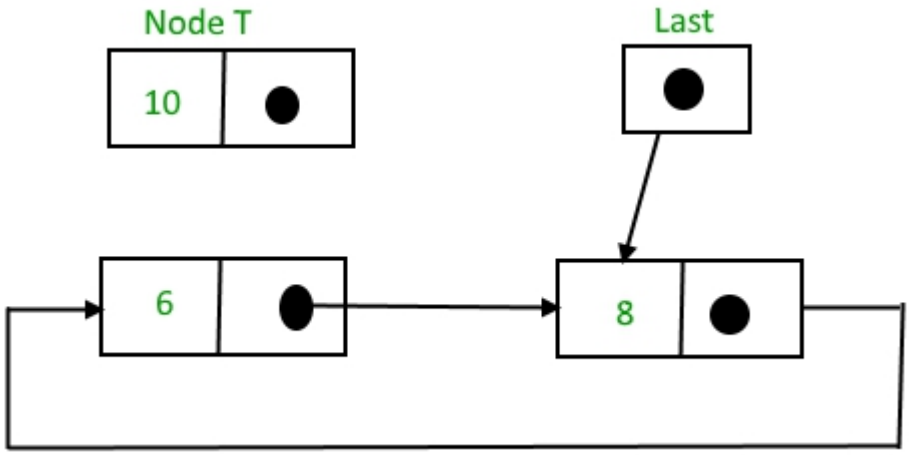
After insertion, T is the last node, so the pointer *last* points to node T. And Node T is the first and the last node, so T points to itself.
Function to insert a node into an empty list,

```
struct Node *addToEmpty(struct Node *last, int data)
{
    // This function is only for empty list
    if (last != NULL)
    return last;

    // Creating a node dynamically.
    struct Node *temp =
        (struct Node*)malloc(sizeof(struct Node));

    // Assigning the data.
    temp -> data = data;
    last = temp;
    // Note : list was empty. We link single node
    // to itself.
    temp -> next = last;

    return last;
}

```

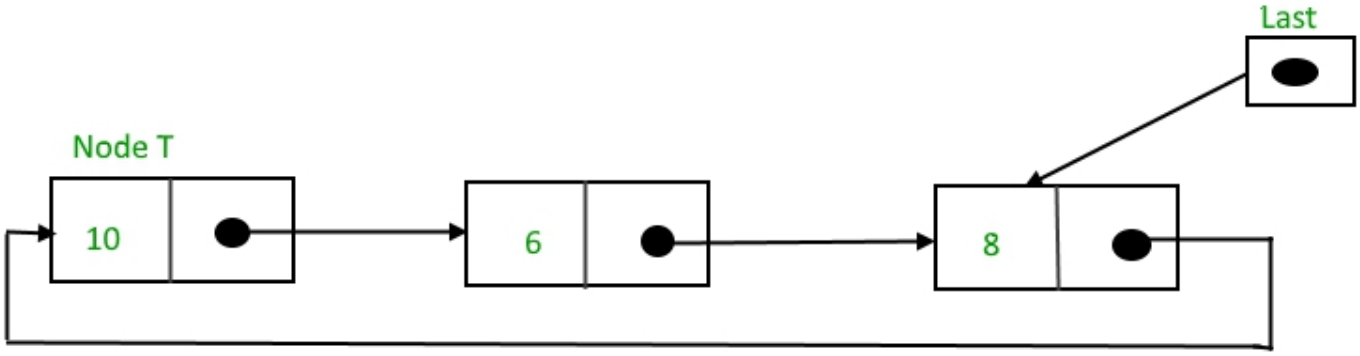
Insertion at the beginning of the list

To insert a node at the beginning of the list, follow these steps:
1. Create a node, say T.
2. Make T -> next = last -> next.

3. last -> next = T.



After insertion,



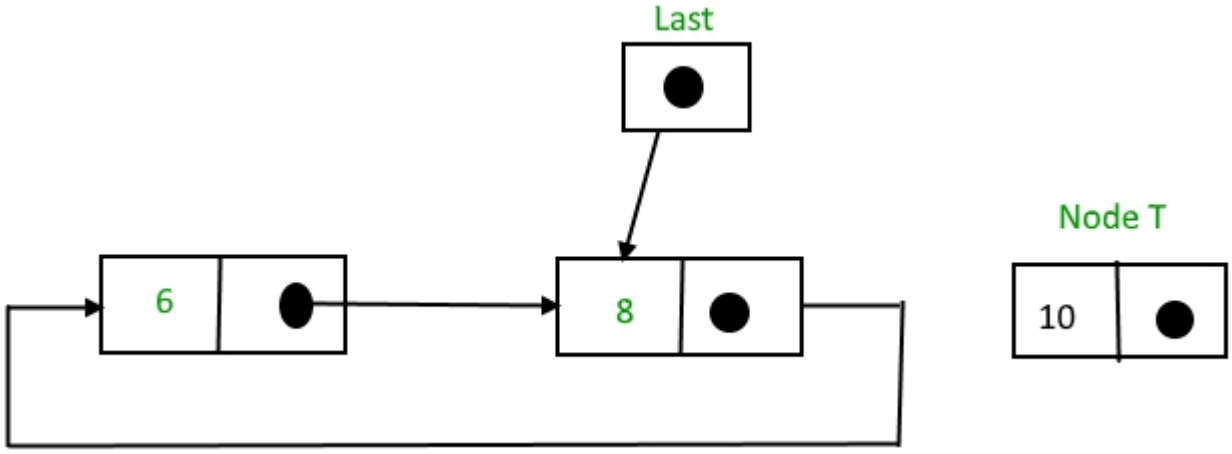
Function to insert nodes at the beginning of the list,

```
struct Node *addBegin(struct Node *last, int data)
{
if (last == NULL)
    return addToEmpty(last, data);

// Creating a node dynamically.
struct Node *temp
        = (struct Node *)malloc(sizeof(struct Node));

// Assigning the data.
temp -> data = data;

// Adjusting the links.
temp -> next = last -> next;
last -> next = temp;

return last;
}

```

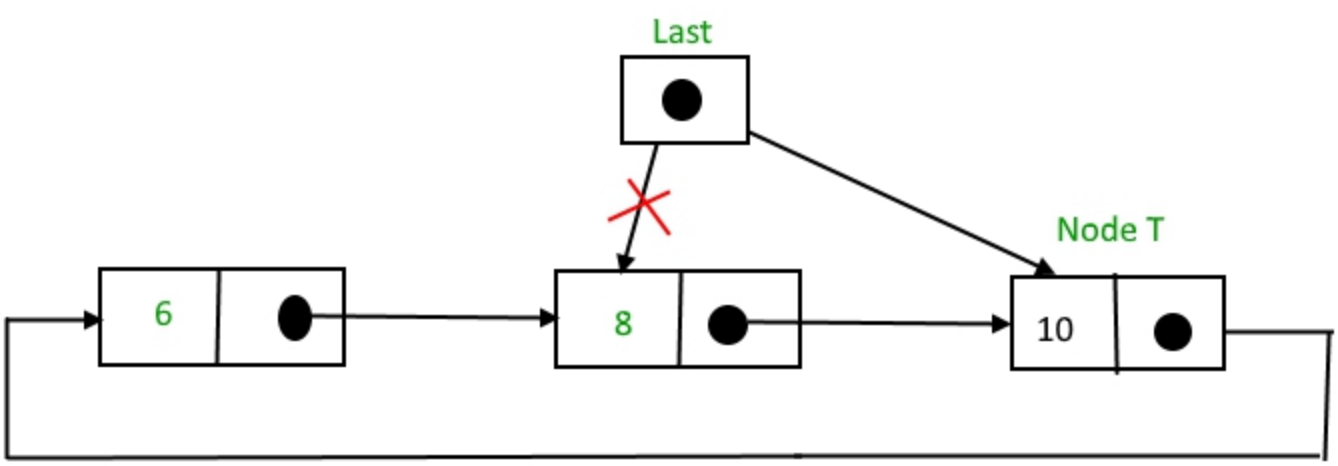
Insertion at the end of the list

To insert a node at the end of the list, follow these steps:

1. Create a node, say T.
2. Make T -> next = last -> next;
3. last -> next = T.
4. last = T.



After insertion,



Function to insert a node at the end of the List

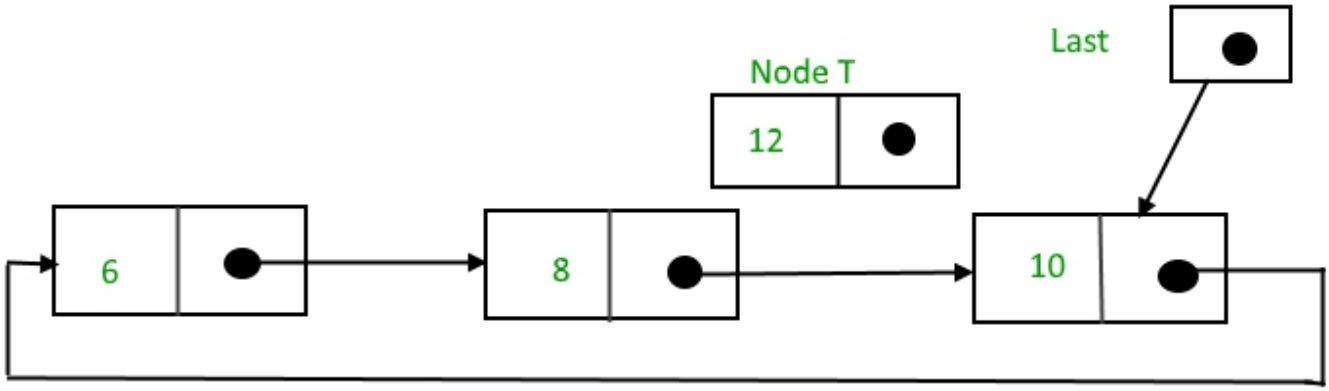
```c
struct Node *addEnd(struct Node *last, int data)
{
if (last == NULL)
    return addToEmpty(last, data);

// Creating a node dynamically.
struct Node *temp =
        (struct Node *)malloc(sizeof(struct Node));

// Assigning the data.
temp -> data = data;

// Adjusting the links.
temp -> next = last -> next;
last -> next = temp;
last = temp;

return last;
}
```

Insertion in between the nodes

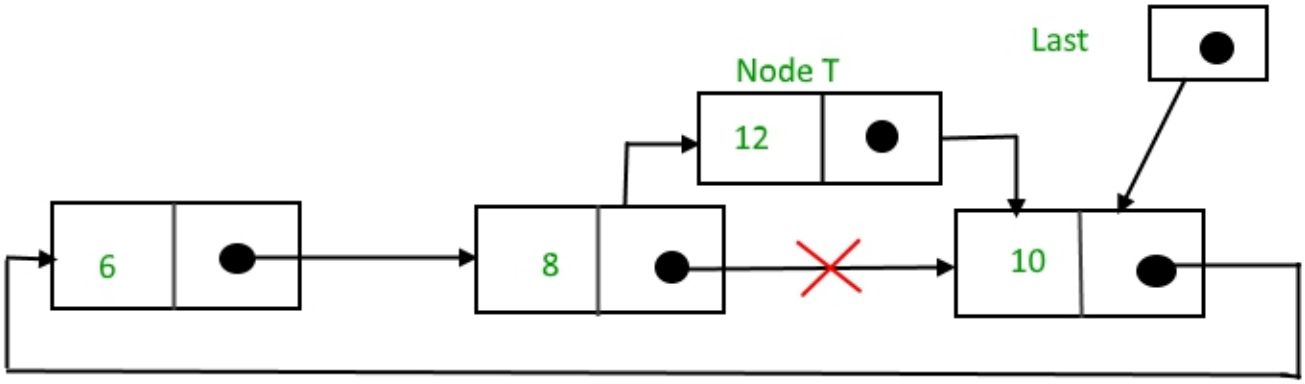
To insert a node in between the two nodes, follow these steps:

1. Create a node, say T.
2. Search for the node after which T needs to be inserted, say that node is P.
3. Make T -> next = P -> next;
4. P -> next = T.

Suppose 12 needs to be inserted after the node has the value 10,



After searching and insertion,



Function to insert a node at the end of the List,

```
struct Node *addAfter(struct Node *last, int data, int item)
{
    if (last == NULL)
    return NULL;

    struct Node *temp, *p;
    p = last -> next;

    // Searching the item.
    do
    {
        if (p ->data == item)
        {
            // Creating a node dynamically.
            temp = (struct Node *)malloc(sizeof(struct Node));

            // Assigning the data.
            temp -> data = data;

            // Adjusting the links.
            temp -> next = p -> next;

            // Adding newly allocated node after p.
            p -> next = temp;

            // Checking for the last node.
            if (p == last)
                last = temp;

            return last;
        }
        p = p -> next;
    } while (p != last -> next);

    cout << item << " not present in the list." << endl;
    return last;
}
```

The following is a complete program that uses all of the above methods to create a circular singly linked list.

```cpp
#include<bits/stdc++.h>
using namespace std;

struct Node
{
    int data;
    struct Node *next;
};

struct Node *addToEmpty(struct Node *last, int data)
{
    // This function is only for empty list
    if (last != NULL)
    return last;

    // Creating a node dynamically.
    struct Node *temp =
        (struct Node*)malloc(sizeof(struct Node));

    // Assigning the data.
    temp -> data = data;
    last = temp;

    // Creating the link.
    last -> next = last;

    return last;
}

struct Node *addBegin(struct Node *last, int data)
{
    if (last == NULL)
        return addToEmpty(last, data);

    struct Node *temp =
            (struct Node *)malloc(sizeof(struct Node));
```

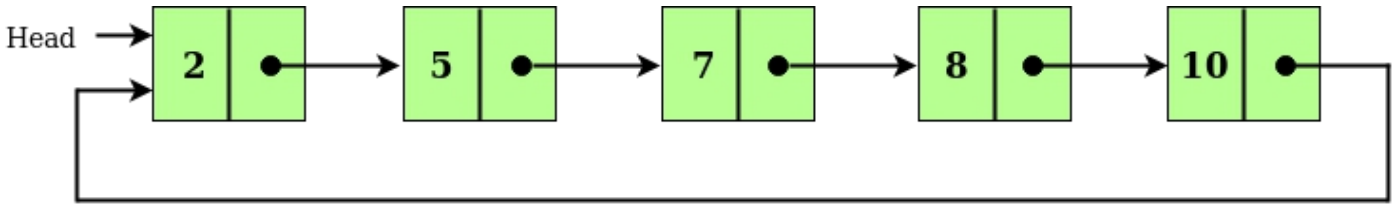
```
37
38      temp -> data = data;
39      temp -> next = last -> next;
40      last -> next = temp;
41
42      return last;
43  }
44
45  struct Node *addEnd(struct Node *last, int data)
46  {
47      if (last == NULL)
48          return addToEmpty(last, data);
49
50      struct Node *temp =
51          (struct Node *)malloc(sizeof(struct Node));
52
53      temp -> data = data;
54      temp -> next = last -> next;
55      last -> next = temp;
56      last = temp;
57
58      return last;
59  }
60
61  struct Node *addAfter(struct Node *last, int data, int item)
62  {
63      if (last == NULL)
64          return NULL;
65
66      struct Node *temp, *p;
67      p = last -> next;
68      do
69      {
70          if (p ->data == item)
71          {
72              temp = (struct Node *)malloc(sizeof(struct Node));
73              temp -> data = data;
74              temp -> next = p -> next;
75              p -> next = temp;
76
77              if (p == last)
78                  last = temp;
79              return last;
80          }
81          p = p -> next;
82      } while(p != last -> next);
83
84      cout << item << " not present in the list." << endl;
85      return last;
86
87  }
88
```

```
89  void traverse(struct Node *last)
90  {
91      struct Node *p;
92
93      // If list is empty, return.
94      if (last == NULL)
95      {
96          cout << "List is empty." << endl;
97          return;
98      }
99
100     // Pointing to first Node of the list.
101     p = last -> next;
102
103     // Traversing the list.
104     do
105     {
106         cout << p -> data << " ";
107         p = p -> next;
108
109     }
110     while(p != last->next);
111
112 }
113
114 // Driven Program
115 int main()
116 {
117     struct Node *last = NULL;
118
119     last = addToEmpty(last, 6);
120     last = addBegin(last, 4);
121     last = addBegin(last, 2);
122     last = addEnd(last, 8);
123     last = addEnd(last, 12);
124     last = addAfter(last, 10, 8);
125
126     traverse(last);
127
128     return 0;
129 }
```

Output:

```
2 4 6 8 10 12
```

Deletion from a Circular Linked List



We will be given a node and our task is to delete that node from the circular linked list.

Examples:

```
Input : 2->5->7->8->10->(head node)
        data = 5
Output : 2->7->8->10->(head node)

Input : 2->5->7->8->10->(head node)
         7
Output : 2->5->8->10->2(head node)
```

Algorithm

Case 1: List is empty.

Case 2:List is not empty

- If the list is not empty then we define two pointers **curr** and **prev** and initialize the pointer **curr** with the **head** node.
- Traverse the list using **curr** to find the node to be deleted and before moving to curr to the next node, every time set prev = curr.
- If the node is found, check if it is the only node in the list. If yes, set head = NULL and free(curr).
- If the list has more than one node, check if it is the first node of the list. Condition to check this(curr == head). If yes, then move prev until it reaches the last node. After prev reaches the last node, set head = head -> next and prev -> next = head. Delete curr.
- If curr is not the first node, we check if it is the last node in the list. Condition to check this is (curr -> next == head).
- If curr is the last node. Set prev -> next = head and delete the node curr by free(curr).

- If the node to be deleted is neither the first node nor the last node, then set prev -> next = curr -> next and delete curr.

Complete program to demonstrate deletion in Circular Linked List:

```cpp
// C++ program to delete a given key from
// linked list.
#include <bits/stdc++.h>
using namespace std;

/* structure for a node */
class Node {
public:
    int data;
    Node* next;
};

/* Function to insert a node at the beginning of
a Circular linked list */
void push(Node** head_ref, int data)
{
    // Create a new node and make head as next
    // of it.
    Node* ptr1 = new Node();
    ptr1->data = data;
    ptr1->next = *head_ref;

    /* If linked list is not NULL then set the
    next of last node */
    if (*head_ref != NULL)
    {
        // Find the node before head and update
        // next of it.
        Node* temp = *head_ref;
        while (temp->next != *head_ref)
            temp = temp->next;
        temp->next = ptr1;
    }
    else
        ptr1->next = ptr1; /*For the first node */

    *head_ref = ptr1;
}

/* Function to print nodes in a given
circular linked list */
void printList(Node* head)
{
    Node* temp = head;
    if (head != NULL) {
        do {
            cout << temp->data << " ";
```

```
            temp = temp->next;
        } while (temp != head);
    }

    cout << endl;
}

/* Function to delete a given node from the list */
void deleteNode(Node** head, int key)
{

    // If linked list is empty
    if (*head == NULL)
        return;

    // If the list contains only a single node
    if((*head)->data==key && (*head)->next==*head)
    {
        free(*head);
        *head=NULL;
        return;
    }

    Node *last=*head,*d;

    // If head is to be deleted
    if((*head)->data==key)
    {

        // Find the last node of the list
        while(last->next!=*head)
            last=last->next;

        // Point last node to the next of head i.e.
        // the second node of the list
        last->next=(*head)->next;
        free(*head);
        *head=last->next;
    }

    // Either the node to be deleted is not found
    // or the end of list is not reached
    while(last->next!=*head&&last->next->data!=key)
    {
        last=last->next;
    }
```

```
94
95      // If node to be deleted was found
96      if(last->next->data==key)
97      {
98          d=last->next;
99          last->next=d->next;
100         free(d);
101     }
102     else
103         cout<<"no such keyfound";
104     }
105
106 /* Driver code */
107 int main()
108 {
109     /* Initialize lists as empty */
110     Node* head = NULL;
111
112     /* Created linked list will be 2->5->7->8->10 */
113     push(&head, 2);
114     push(&head, 5);
115     push(&head, 7);
116     push(&head, 8);
117     push(&head, 10);
118
119     cout << "List Before Deletion: ";
120     printList(head);
121
122     deleteNode(&head, 7);
123
124     cout << "List After Deletion: ";
125     printList(head);
126
127     return 0;
128 }
```

Output:

```
List Before Deletion: 10 8 7 5 2
List After Deletion: 10 8 5 2
```

Circular Queue | Set 2 (Circular Linked List Implementation)

another method of circular queue implementation is discussed, using Circular Singly Linked List.

Operations on Circular Queue:

- **Front:**Get the front item from queue.
- **Rear:** Get the last item from queue.
- **enQueue(value)** This function is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at Rear position.
-

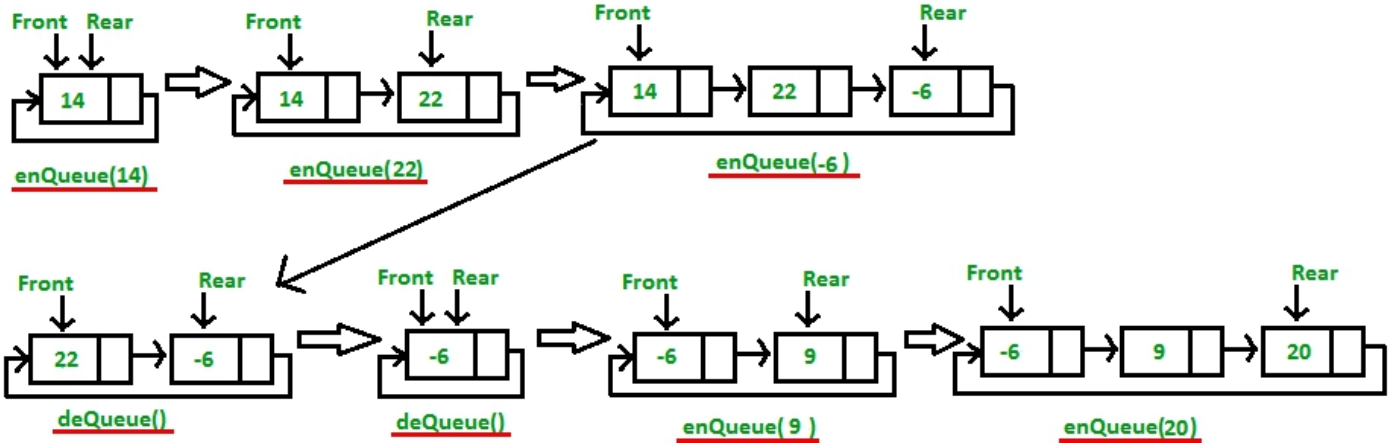
Steps:

1. Create a new node dynamically and insert value into it.
2. Check if front==NULL, if it is true then front = rear = (newly created node)
3. If it is false then rear=(newly created node) and rear node always contains the address of the front node.

deQueue() This function is used to delete an element from the circular queue. In a queue the element is always deleted from front position.

Steps:

1. Check whether queue is empty or not means front == NULL.
2. If it is empty then display Queue is empty. If queue is not empty then step 3
3. Check if (front==rear) if it is true then set front = rear = NULL else move the front forward in queue, update address of front in rear node and return the element.



```cpp
// C++ program for insertion and
// deletion in Circular Queue
#include <bits/stdc++.h>
using namespace std;

// Structure of a Node
struct Node {
    int data;
    struct Node* link;
};

struct Queue {
    struct Node *front, *rear;
};

// Function to create Circular queue
void enQueue(Queue* q, int value)
{
    struct Node* temp = new Node;
    temp->data = value;
    if (q->front == NULL)
        q->front = temp;
    else
        q->rear->link = temp;

    q->rear = temp;
    q->rear->link = q->front;
}

// Function to delete element from Circular Queue
int deQueue(Queue* q)
{
    if (q->front == NULL) {
        printf("Queue is empty");
        return INT_MIN;
    }

    // If this is the last node to be deleted
    int value; // Value to be dequeued
    if (q->front == q->rear) {
        value = q->front->data;
        free(q->front);
        q->front = NULL;
        q->rear = NULL;
    }
    else // There are more than one nodes
    {
        struct Node* temp = q->front;
        value = temp->data;
        q->front = q->front->link;
        q->rear->link = q->front;
        free(temp);
    }

    return value;
}
```

```
57
58  // Function displaying the elements of Circular Queue
59  void displayQueue(struct Queue* q)
60  {
61      struct Node* temp = q->front;
62      printf("\nElements in Circular Queue are: ");
63      while (temp->link != q->front) {
64          printf("%d ", temp->data);
65          temp = temp->link;
66      }
67      printf("%d", temp->data);
68  }
69
70  /* Driver of the program */
71  int main()
72  {
73      // Create a queue and initialize front and rear
74      Queue* q = new Queue;
75      q->front = q->rear = NULL;
76
77      // Inserting elements in Circular Queue
78      enQueue(q, 14);
79      enQueue(q, 22);
80      enQueue(q, 6);
81
82      // Display elements present in Circular Queue
83      displayQueue(q);
84
85      // Deleting elements from Circular Queue
86      printf("\nDeleted value = %d", deQueue(q));
87      printf("\nDeleted value = %d", deQueue(q));
88
89      // Remaining elements in Circular Queue
90      displayQueue(q);
91
92      enQueue(q, 9);
93      enQueue(q, 20);
94      displayQueue(q);
95
96      return 0;
97  }
```

Output:

```
Elements in Circular Queue are: 14 22 6
Deleted value = 14
Deleted value = 22
Elements in Circular Queue are: 6
Elements in Circular Queue are: 6 9 20
```

Time Complexity: Time complexity of enQueue(), deQueue() operation is O(1) as there is no loop in any of the operation.

Implementation of Deque using circular array

Operations on Deque:

Mainly the following four basic operations are performed on queue:

insetFront(): Adds an item at the front of Deque.

insertRear(): Adds an item at the rear of Deque.

deleteFront(): Deletes an item from front of Deque.

deleteRear(): Deletes an item from rear of Deque.

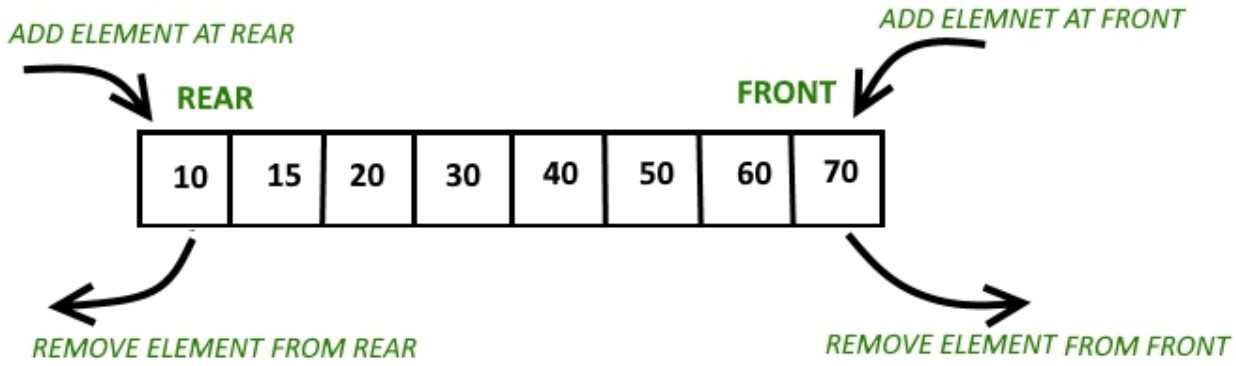
In addition to above operations, following operations are also supported

getFront(): Gets the front item from queue.

getRear(): Gets the last item from queue.

isEmpty(): Checks whether Deque is empty or not.

isFull(): Checks whether Deque is full or not.

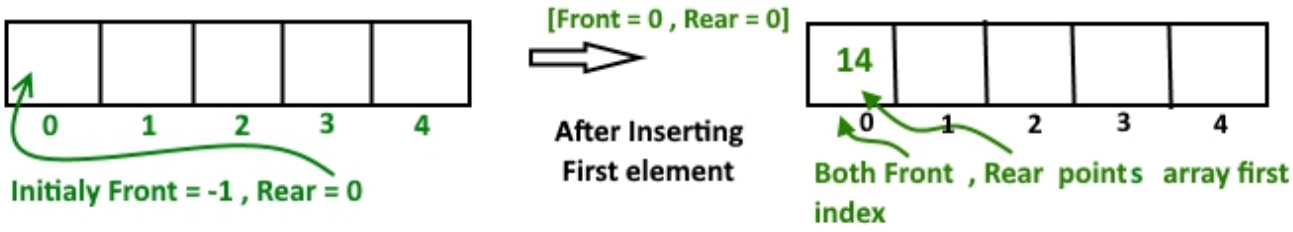


Circular array implementation deque

For implementing deque, we need to keep track of two indices, front and rear. We enqueue(push) an item at the rear or the front end of qedue and dequeue(pop) an item from both rear and front end.

Working

1. Create an empty array ‘arr’ of size ‘n’
initialize **front = -1** , **rear = 0**
Inserting First element in deque, at either front or rear will lead to the same result.



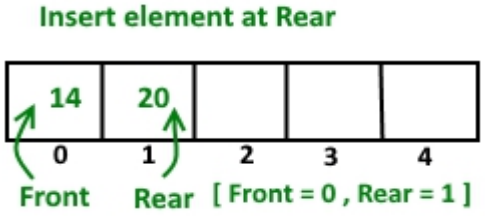
After insert **Front** Points = 0 and **Rear** points = 0

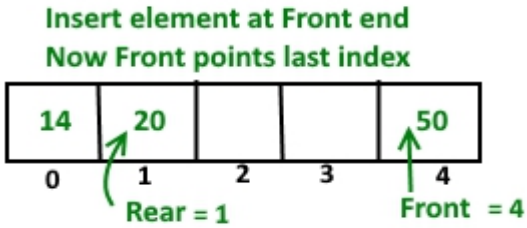
Insert Elements at Rear end

```
a). First we check deque if Full or Not
b). IF Rear == Size-1
       then reinitialize Rear = 0 ;
    Else increment Rear by '1'
    and push current key into Arr[ rear ] = key
Front remain same.
```

Insert Elements at Front end

```
a). First we check deque if Full or Not
b). IF Front == 0 || initial position, move Front
                     to points last index of array
       front = size - 1
    Else decremented front by '1' and push
         current key into Arr[ Front] = key
Rear remain same.
```



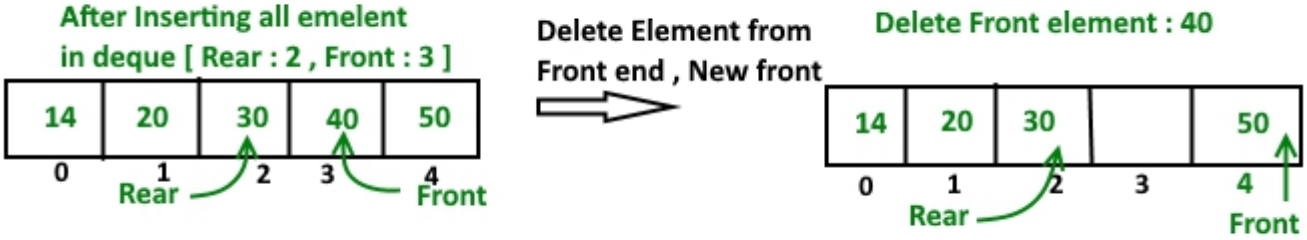


Delete Element From Rear end

```
a). first Check deque is Empty or Not
b).  If deque has only one element
        front = -1 ; rear =-1 ;
    Else IF Rear points to the first index of array
         it's means we have to move rear to points
         last index [ now first inserted element at
         front end become rear end ]
            rear = size-1 ;
    Else || decrease rear by '1'
            rear = rear-1;
```

Delete Element From Front end

```
a). first Check deque is Empty or Not
b).  If deque has only one element
            front = -1 ; rear =-1 ;
    Else IF front points to the last index of the array
         it's means we have no more elements in array so
          we move front to points first index of array
            front = 0 ;
    Else || increment Front by '1'
            front = front+1;
```



Below is the implementation of above idea.

```cpp
// C++ implementation of De-queue using circular
// array
#include<iostream>
using namespace std;

// Maximum size of array or Dequeue
#define MAX 100

// A structure to represent a Deque
class Deque
{
    int arr[MAX];
    int front;
    int rear;
    int size;
public :
    Deque(int size)
    {
        front = -1;
        rear = 0;
        this->size = size;
    }

    // Operations on Deque:
    void insertfront(int key);
    void insertrear(int key);
    void deletefront();
    void deleterear();
    bool isFull();
    bool isEmpty();
    int getFront();
    int getRear();
};

// Checks whether Deque is full or not.
bool Deque::isFull()
{
    return ((front == 0 && rear == size-1)||
            front == rear+1);
}

// Checks whether Deque is empty or not.
bool Deque::isEmpty ()
{
    return (front == -1);
}
```

```
47
48  // Inserts an element at front
49  void Deque::insertfront(int key)
50  {
51      // check whether Deque if full or not
52      if (isFull())
53      {
54          cout << "Overflow\n" << endl;
55          return;
56      }
57
58      // If queue is initially empty
59      if (front == -1)
60      {
61          front = 0;
62          rear = 0;
63      }
64
65      // front is at first position of queue
66      else if (front == 0)
67          front = size - 1 ;
68
69      else // decrement front end by '1'
70          front = front-1;
71
72      // insert current element into Deque
73      arr[front] = key ;
74  }
75
76  // function to inset element at rear end
77  // of Deque.
78  void Deque ::insertrear(int key)
79  {
80      if (isFull())
81      {
82          cout << " Overflow\n " << endl;
83          return;
84      }
85
86      // If queue is initially empty
87      if (front == -1)
88      {
89          front = 0;
90          rear = 0;
91      }
92
93       // rear is at last position of queue
94       else if (rear == size-1)
95           rear = 0;
96
97       // increment rear end by '1'
98       else
99           rear = rear+1;
```

```
100
101     // insert current element into Deque
102     arr[rear] = key ;
103 }
104
105 // Deletes element at front end of Deque
106 void Deque ::deletefront()
107 {
108     // check whether Deque is empty or not
109     if (isEmpty())
110     {
111         cout << "Queue Underflow\n" << endl;
112         return ;
113     }
114
115     // Deque has only one element
116     if (front == rear)
117     {
118         front = -1;
119         rear = -1;
120     }
121     else
122         // back to initial position
123         if (front == size -1)
124             front = 0;
125
126         else // increment front by '1' to remove current
127             // front value from Deque
128             front = front+1;
129 }
130
131 // Delete element at rear end of Deque
132 void Deque::deleterear()
133 {
134     if (isEmpty())
135     {
136         cout << " Underflow\n" << endl ;
137         return ;
138     }
139
140     // Deque has only one element
141     if (front == rear)
142     {
143         front = -1;
144         rear = -1;
145     }
```

```
146     else if (rear == 0)
147         rear = size-1;
148     else
149         rear = rear-1;
150 }
151
152 // Returns front element of Deque
153 int Deque::getFront()
154 {
155     // check whether Deque is empty or not
156     if (isEmpty())
157     {
158         cout << " Underflow\n" << endl;
159         return -1 ;
160     }
161     return arr[front];
162 }
163
164 // function return rear element of Deque
165 int Deque::getRear()
166 {
167     // check whether Deque is empty or not
168     if(isEmpty() || rear < 0)
169     {
170         cout << " Underflow\n" << endl;
171         return -1 ;
172     }
173     return arr[rear];
174 }
175
176 // Driver program to test above function
177 int main()
178 {
179     Deque dq(5);
180     cout << "Insert element at rear end : 5 \n";
181     dq.insertrear(5);
182
183     cout << "insert element at rear end : 10 \n";
184     dq.insertrear(10);
185
186     cout << "get rear element " << " "
187         << dq.getRear() << endl;
188
189     dq.deleterear();
190     cout << "After delete rear element new rear"
```

```
191         << " become " << dq.getRear() << endl;
192
193     cout << "inserting element at front end \n";
194     dq.insertfront(15);
195
196     cout << "get front element " << " "
197         << dq.getFront() << endl;
198
199     dq.deletefront();
200
201     cout << "After delete front element new "
202     << "front become " << dq.getFront() << endl;
203     return 0;
204 }
```

Output:

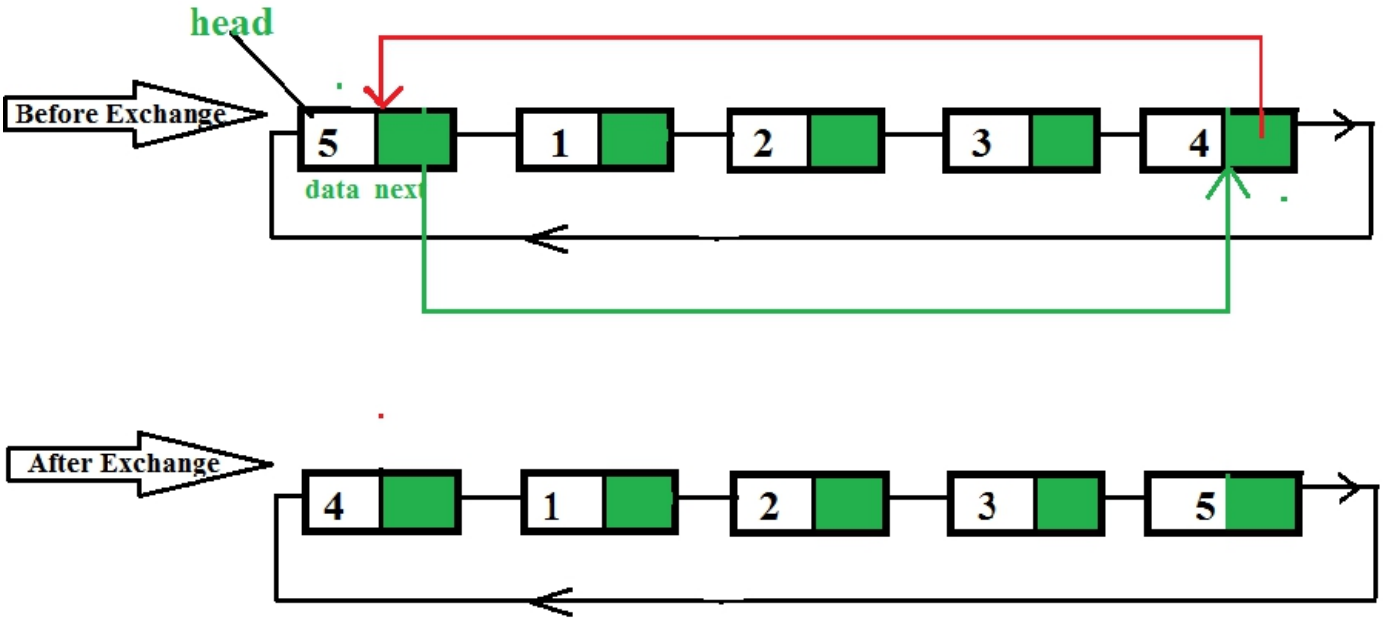
```
insert element at rear end  : 5
insert element at rear end : 10
get rear element : 10
After delete rear element new rear become : 5
inserting element at front end
get front element : 15
After delete front element new front become : 5
```

Time Complexity: Time complexity of all operations like insertfront(), insertlast(), deletefront(), deletelast()is O(1).

Exchange first and last nodes in Circular Linked List

The task should be done with only one extra node, you can not declare more than one extra node, and also you are not allowed to declare any other temporary variable.

Note: Extra node means the need of a node to traverse a list.



Examples:

```
Input : 5 4 3 2 1
Output : 1 4 3 2 5

Input  : 6 1 2 3 4 5 6 7 8 9
Output : 9 1 2 3 4 5 6 7 8 6
```

Method 1: *(By Changing Links of First and Last Nodes)*
We first find a pointer to previous of last node. Let this node be p. Now we change the next links so that the last and first nodes are swapped.

Time Complexity:

O(n)

```cpp
// CPP program to exchange first and
// last node in circular linked list
#include <bits/stdc++.h>
using namespace std;

struct Node {
    int data;
    struct Node* next;
};

struct Node* addToEmpty(struct Node* head, int data)
{
    // This function is only for empty list
    if (head != NULL)
        return head;

    // Creating a node dynamically.
    struct Node* temp
        = (struct Node*)malloc(sizeof(struct Node));

    // Assigning the data.
    temp->data = data;
    head = temp;

    // Creating the link.
    head->next = head;

    return head;
}

struct Node* addBegin(struct Node* head, int data)
{
    if (head == NULL)
        return addToEmpty(head, data);

    struct Node* temp
        = (struct Node*)malloc(sizeof(struct Node));

    temp->data = data;
    temp->next = head->next;
    head->next = temp;

    return head;
}

/* function for traversing the list */
void traverse(struct Node* head)
{
    struct Node* p;

    // If list is empty, return.
    if (head == NULL) {
        cout << "List is empty." << endl;
        return;
    }
```

```
56
57      // Pointing to first Node of the list.
58      p = head;
59
60      // Traversing the list.
61      do {
62          cout << p->data << " ";
63          p = p->next;
64
65      } while (p != head);
66  }
67
68  /* Function to exchange first and last node*/
69  struct Node* exchangeNodes(struct Node* head)
70  {
71      // If list is of length 2
72      if (head->next->next == head) {
73          head = head->next;
74          return head;
75      }
76
77      // Find pointer to previous of last node
78      struct Node* p = head;
79      while (p->next->next != head)
80          p = p->next;
81
82      /* Exchange first and last nodes using
83      head and p */
84      p->next->next = head->next;
85      head->next = p->next;
86      p->next = head;
87      head = head->next;
88
89      return head;
90  }
91
92  // Driven Program
93  int main()
94  {
95      int i;
96      struct Node* head = NULL;
97      head = addToEmpty(head, 6);
98
99      for (i = 5; i > 0; i--)
100         head = addBegin(head, i);
101     cout << "List Before: ";
102      traverse(head);
103      cout << endl;
104
105      cout << "List After: ";
106      head = exchangeNodes(head);
107      traverse(head);
108
109      return 0;
110  }
```

Output

```
List Before: 6 1 2 3 4 5
List After: 5 1 2 3 4 6
```

Method 2: *(By Swapping Values of First and Last nodes)*
Algorithm:

1. Traverse the list and find the last node(tail).
2. Swap data of head and tail.

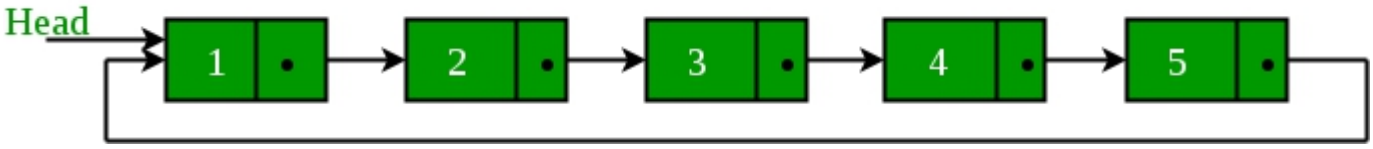
Below is the implementation of the algorithm:

```
/* Function to exchange first and last node*/
struct Node* exchangeNodes(struct Node* head)
{

    // If list is of length less than 2
    if (head == NULL || head->next == NULL) {
        return head;
    }
    Node* tail = head;

    // Find pointer to the last node
    while (tail->next != head) {
        tail = tail->next;
    }
    /* Exchange first and last nodes using
       head and p */

    // temporary variable to store
    // head data
    int temp = tail->data;
    tail->data = head->data;
    head->data = temp;
    return head;
}
```

Count nodes in Circular linked list

Given a circular linked list, count the number of nodes in it. For example, the output is 5

for the below list.




```
// c++ program to count number of nodes in
// a circular linked list.
#include <bits/stdc++.h>
using namespace std;

/*structure for a node*/

struct Node {
    int data;
    Node* next;
    Node(int x)
    {
        data = x;
        next = NULL;
    }
};
/* Function to insert a node at the beginning
of a Circular linked list */
struct Node* push(struct Node* last, int data)
{
    if (last == NULL) {
        struct Node* temp
            = (struct Node*)malloc(sizeof(struct Node));

        // Assigning the data.
        temp->data = data;
        last = temp;
        // Note : list was empty. We link single node
        // to itself.
        temp->next = last;

        return last;
    }

    // Creating a node dynamically.
    struct Node* temp
        = (struct Node*)malloc(sizeof(struct Node));

    // Assigning the data.
    temp->data = data;

    // Adjusting the links.
    temp->next = last->next;
    last->next = temp;

    return last;
}
```

```
48
49  /* Function to count nodes in a given Circular
50  linked list */
51
52  int countNodes(Node* head)
53  {
54      Node* temp = head;
55      int result = 0;
56      if (head != NULL) {
57          do {
58              temp = temp->next;
59              result++;
60          } while (temp != head);
61      }
62
63      return result;
64  }
65
66  /* Driver program to test above functions */
67  int main()
68  {
69      /* Initialize lists as empty */
70      Node* head = NULL;
71      head = push(head, 12);
72      head = push(head, 56);
73      head = push(head, 2);
74      head = push(head, 11);
75      cout << countNodes(head);
76      return 0;
77  }
78
```

Output:

```
4
```

Josephus Circle implementation using STL list

There are n people standing in a circle waiting to be executed. The counting out begins at some point in the circle and proceeds around the circle in a fixed direction. In each step, a certain number of people are skipped and the next person is executed. The elimination proceeds around the circle (which is becoming smaller and smaller as the executed people are removed), until only the last person remains, who is given freedom. Given the total number of persons n and a number k which indicates that k-1 persons are

skipped and k-th person is killed in circle. The task is to choose the place in the initial circle so that you are the last one remaining and so survive. (0 based indexing) .
Examples :

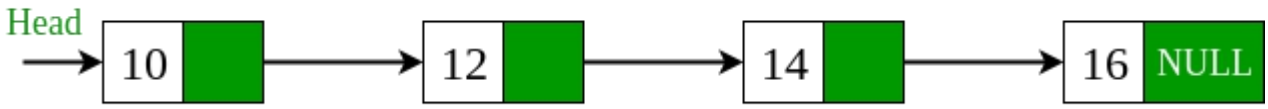
```
Input : Length of circle : n = 4
        Count to choose next : k = 2
Output : 0

Input : n = 5
        k = 3
Output : 3
```

```cpp
// CPP program to find last man standing
#include <bits/stdc++.h>
using namespace std;

int josephusCircle(int n, int k){
    list<int>l; //creates a doubly linked list using stl container//
    for(int i=0;i<n;i++)
        l.push_back(i); //pushes i to the end of the doubly linked list//

    auto it = l.begin();
    while(l.size()>1){

        for(int i=1;i<k;i++){
            it++;

            if(it==l.end()){
            //if iterator reahes the end,then we change it to begin of the list//
                it = l.begin();
            }
        }

        it = l.erase(it);

        if(it==l.end()){
        //if iterator reahes the end,then we change it to begin of the list//
                it = l.begin();
            }
    }

    return l.front(); //returns front element of the list//

}
/* Driver program to test above functions */
int main()
{
int n=14,k=2;

    cout<<josephusCircle(n,k)<<"\n";

    return 0;
}

```

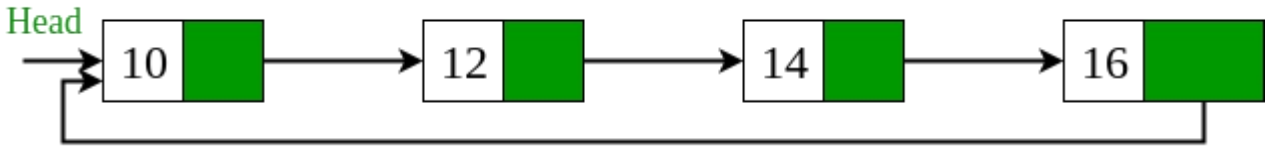
Time complexity: O(k * n)

Convert singly linked list into circular linked list

Given a singly linked list, we have to convert it into circular linked list. For example, we have been given a singly linked list with four nodes and we want to convert this singly
linked list into circular linked list.



The above singly linked list is converted into circular linked list.



Approach: The idea is to traverse the singly linked list and check if the node is the last node or not. If the node is the last node i.e pointing to NULL then make it point to the starting node i.e head node. Below is the implementation of this approach.

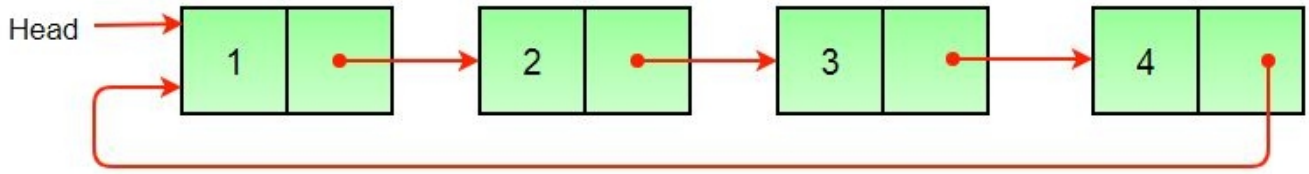
```cpp
// Program for converting singly linked list
// into circular linked list.
#include <bits/stdc++.h>

/* Linked list node */
struct Node {
    int data;
    struct Node* next;
};

// Function that convert singly linked list
// into circular linked list.
struct Node* circular(struct Node* head)
{
    // declare a node variable start and
    // assign head node into start node.
    struct Node* start = head;

    // check that while head->next not equal
    // to NULL then head points to next node.
    while (head->next != NULL)
        head = head->next;

    // if head->next points to NULL then
    // start assign to the head->next node.
    head->next = start;
    return start;
}

void push(struct Node** head, int data)
{
    // Allocate dynamic memory for newNode.
    struct Node* newNode = (struct Node*)malloc
                           (sizeof(struct Node));

    // Assign the data into newNode.
    newNode->data = data;

    // newNode->next assign the address of
    // head node.
    newNode->next = (*head);

    // newNode become the headNode.
    (*head) = newNode;
}

```

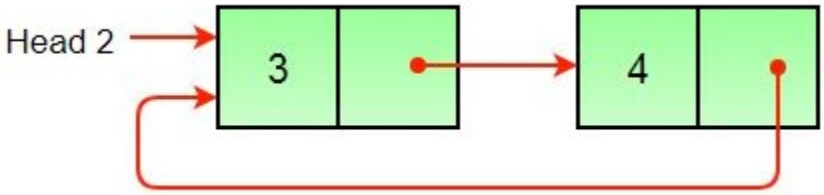
```c
// Function that display the elements of
// circular linked list.
void displayList(struct Node* node)
{
    struct Node* start = node;

    while (node->next != start) {
        printf("%d ", node->data);
        node = node->next;
    }

    // Display the last node of circular
    // linked list.
    printf("%d ", node->data);
}

// Driver program to test the functions
int main()
{
    // Start with empty list
    struct Node* head = NULL;

    // Using push() function to construct
    // singly linked list
    // 17->22->13->14->15
    push(&head, 15);
    push(&head, 14);
    push(&head, 13);
    push(&head, 22);
    push(&head, 17);

    // Call the circular_list function that
    // convert singly linked list to circular
    // linked list.
    circular(head);

    printf("Display list: \n");
    displayList(head);

    return 0;
}
```

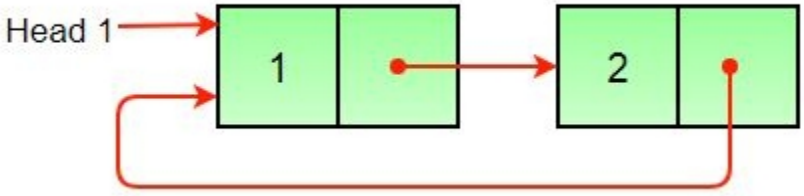
Output:

```
Display list:
17 22 13 14 15
```

Split a Circular Linked List into two halves







If there are **odd number of nodes**, then first list should contain one extra.

1) Store the mid and last pointers of the circular linked list using tortoise and hare algorithm.
2) Make the second half circular.
3) Make the first half circular.
4) Set head (or start) pointers of the two linked lists

```cpp
// Program to split a circular linked list
// into two halves
#include <bits/stdc++.h>
using namespace std;

/* structure for a node */
class Node
{
    public:
    int data;
    Node *next;
};

/* Function to split a list (starting with head)
into two lists. head1_ref and head2_ref are
references to head nodes of the two resultant
linked lists */
void splitList(Node *head, Node **head1_ref,
                           Node **head2_ref)
{
    Node *slow_ptr = head;
    Node *fast_ptr = head;

    if(head == NULL)
        return;

    /* If there are odd nodes in the circular list then
    fast_ptr->next becomes head and for even nodes
    fast_ptr->next->next becomes head */
    while(fast_ptr->next != head &&
        fast_ptr->next->next != head)
    {
        fast_ptr = fast_ptr->next->next;
        slow_ptr = slow_ptr->next;
    }

    /* If there are even elements in list
    then move fast_ptr */
    if(fast_ptr->next->next == head)
        fast_ptr = fast_ptr->next;

    /* Set the head pointer of first half */
    *head1_ref = head;

    /* Set the head pointer of second half */
    if(head->next != head)
        *head2_ref = slow_ptr->next;

    /* Make second half circular */
    fast_ptr->next = slow_ptr->next;

    /* Make first half circular */
    slow_ptr->next = head;
}

```

```
/* UTILITY FUNCTIONS */
/* Function to insert a node at
the beginning of a Circular linked list */
void push(Node **head_ref, int data)
{
    Node *ptr1 = new Node();
    Node *temp = *head_ref;
    ptr1->data = data;
    ptr1->next = *head_ref;

    /* If linked list is not NULL then
    set the next of last node */
    if(*head_ref != NULL)
    {
        while(temp->next != *head_ref)
        temp = temp->next;
        temp->next = ptr1;
    }
    else
        ptr1->next = ptr1; /*For the first node */

    *head_ref = ptr1;
}

/* Function to print nodes in
a given Circular linked list */
void printList(Node *head)
{
    Node *temp = head;
    if(head != NULL)
    {
        cout << endl;
        do {
        cout << temp->data << " ";
        temp = temp->next;
        } while(temp != head);
    }
}

// Driver Code
int main()
{
    int list_size, i;

    /* Initialize lists as empty */
    Node *head = NULL;
    Node *head1 = NULL;
    Node *head2 = NULL;

    /* Created linked list will be 12->56->2->11 */
    push(&head, 12);
    push(&head, 56);
    push(&head, 2);
    push(&head, 11);
```

```
110
111     cout << "Original Circular Linked List";
112     printList(head);
113
114     /* Split the list */
115     splitList(head, &head1, &head2);
116
117     cout << "\nFirst Circular Linked List";
118     printList(head1);
119
120     cout << "\nSecond Circular Linked List";
121     printList(head2);
122     return 0;
123 }
124
```

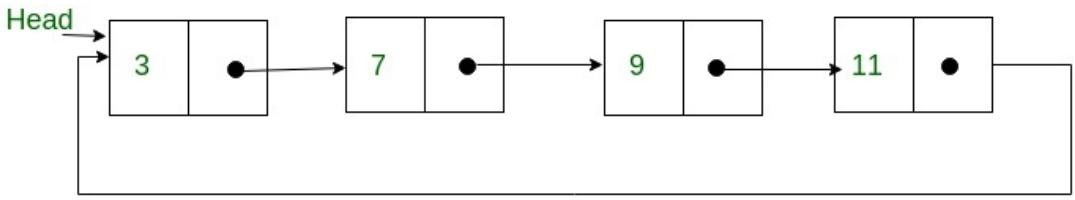
Output:

```
Original Circular Linked List
11 2 56 12
First Circular Linked List
11 2
Second Circular Linked List
56 12
```

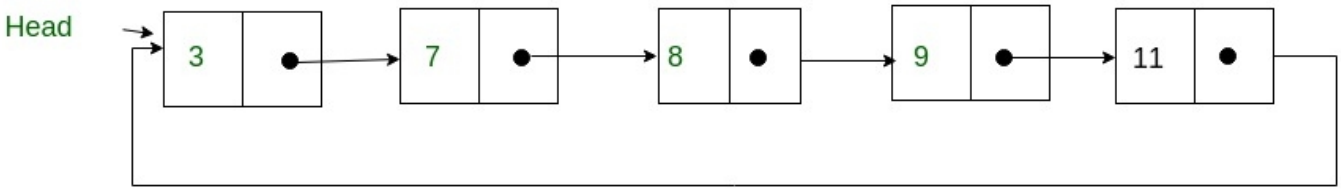
Time Complexity: **O(n)**

Sorted insert for circular linked list

Difficulty Level: Rookie
Write a C++ function to insert a new value in a sorted Circular Linked List (CLL). For example, if the input CLL is following.



After inserting 8, the above CLL should be changed to the following



Algorithm:
Allocate memory for the newly inserted node and put data in the newly allocated node. Let the pointer to the new node be new_node. After memory allocation, following are the three cases that need to be handled.

```
1) Linked List is empty:
    a)  since new_node is the only node in CLL, make a self loop.
          new_node->next = new_node;
    b) change the head pointer to point to new node.
          *head_ref = new_node;
2) New node is to be inserted just before the head node:
  (a) Find out the last node using a loop.
          while(current->next != *head_ref)
             current = current->next;
  (b) Change the next of last node.
          current->next = new_node;
  (c) Change next of new node to point to head.
          new_node->next = *head_ref;
  (d) change the head pointer to point to new node.
          *head_ref = new_node;
3) New node is to be  inserted somewhere after the head:
   (a) Locate the node after which new node is to be inserted.
          while ( current->next!= *head_ref &&
              current->next->data data)
          {   current = current->next;   }
```

```
(b) Make next of new_node as next of the located pointer
      new_node->next = current->next;
(c) Change the next of the located pointer
      current->next = new_node;
```

```cpp
// C++ program for sorted insert
// in circular linked list
#include <bits/stdc++.h>
using namespace std;

/* structure for a node */
class Node
{
    public:
    int data;
    Node *next;
};

/* function to insert a new_node in a list in sorted way.
Note that this function expects a pointer to head node
as this can modify the head of the input linked list */
void sortedInsert(Node** head_ref, Node* new_node)
{
    Node* current = *head_ref;

    // Case 1 of the above algo
    if (current == NULL)
    {
        new_node->next = new_node;
        *head_ref = new_node;
    }

    // Case 2 of the above algo
    else if (current->data >= new_node->data)
    {
        /* If value is smaller than head's value then
        we need to change next of last node */
        while(current->next != *head_ref)
            current = current->next;
        current->next = new_node;
        new_node->next = *head_ref;
        *head_ref = new_node;
    }

    // Case 3 of the above algo
    else
    {
        /* Locate the node before the point of insertion */
        while (current->next!= *head_ref &&
            current->next->data < new_node->data)
        current = current->next;

```

```
48          new_node->next = current->next;
49          current->next = new_node;
50      }
51  }
52
53  /* Function to print nodes in a given linked list */
54  void printList(Node *start)
55  {
56      Node *temp;
57
58      if(start != NULL)
59      {
60          temp = start;
61          do {
62          cout<<temp->data<<" ";
63          temp = temp->next;
64          } while(temp != start);
65      }
66  }
67
68  /* Driver code */
69  int main()
70  {
71      int arr[] = {12, 56, 2, 11, 1, 90};
72      int list_size, i;
73
74      /* start with empty linked list */
75      Node *start = NULL;
76      Node *temp;
77
78      /* Create linked list from the array arr[].
79          Created linked list will be 1->2->11->12->56->90 */
80      for (i = 0; i< 6; i++)
81      {
82          temp = new Node();
83          temp->data = arr[i];
84          sortedInsert(&start, temp);
85      }
86
87      printList(start);
88
89      return 0;
90  }
91
```

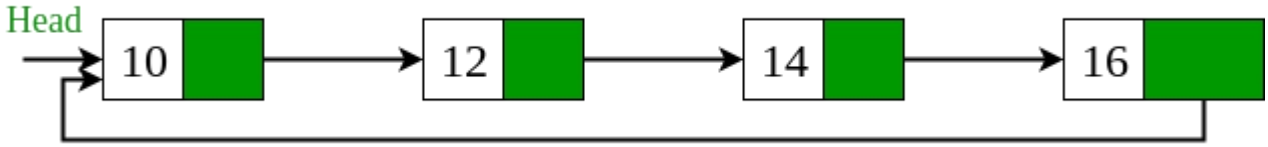
Output

```
1 2 11 12 56 90
```

Time Complexity: O(n) where n is the number of nodes in the given linked list.

Check if a linked list is Circular Linked List

A linked list is called circular if it is not NULL-terminated and all nodes are connected in the form of a cycle. Below is an example of a circular linked list.



An empty linked list is considered as circular.

```cpp
// C++ program to check if linked list is circular
#include<bits/stdc++.h>
using namespace std;

/* Link list Node */
struct Node
{
    int data;
    struct Node* next;
};

/* This function returns true if given linked
list is circular, else false. */
bool isCircular(struct Node *head)
{
    // An empty linked list is circular
    if (head == NULL)
    return true;

    // Next of head
    struct Node *node = head->next;

    // This loop would stop in both cases (1) If
    // Circular (2) Not circular
    while (node != NULL && node != head)
    node = node->next;

    // If loop stopped because of circular
    // condition
    return (node == head);
}
```

```
32
33  // Utility function to create a new node.
34  Node *newNode(int data)
35  {
36      struct Node *temp = new Node;
37      temp->data = data;
38      temp->next = NULL;
39      return temp;
40  }
41
42  /* Driver program to test above function*/
43  int main()
44  {
45      /* Start with the empty list */
46      struct Node* head = newNode(1);
47      head->next = newNode(2);
48      head->next->next = newNode(3);
49      head->next->next->next = newNode(4);
50
51      isCircular(head)? cout << "Yes\n" :
52                        cout << "No\n" ;
53
54      // Making linked list circular
55      head->next->next->next->next = head;
56
57      isCircular(head)? cout << "Yes\n" :
58                        cout << "No\n" ;
59
60      return 0;
61  }
```

Output

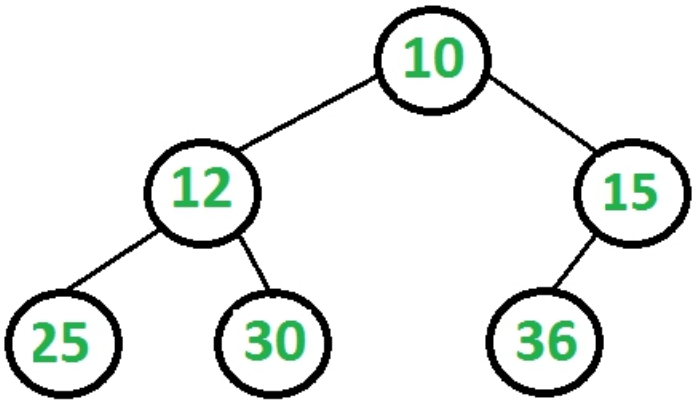
```
No
Yes
```

Convert a Binary Tree to a Circular Doubly Link List

Given a Binary Tree, convert it to a Circular Doubly Linked List (In-Place).

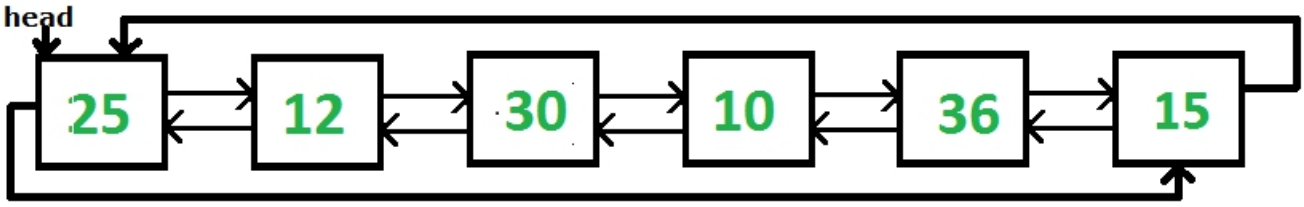
- The left and right pointers in nodes are to be used as previous and next pointers respectively in converted Circular Linked List.
- The order of nodes in the List must be the same as in Inorder for the given Binary Tree.
- The first node of Inorder traversal must be the head node of the Circular List.

Example:



The above tree should be in-place converted to following

Circular Doubly Linked List



The idea can be described using the below steps.

1) Write a general-purpose function that concatenates two given circular doubly lists (This function is explained below).

2) Now traverse the given tree
….a) Recursively convert left subtree to a circular DLL. Let the converted list be leftList.
….a) Recursively convert right subtree to a circular DLL. Let the converted list be rightList.
….c) Make a circular linked list of root of the tree, make left and right of root to point to itself.
….d) Concatenate leftList with the list of the single root node.
….e) Concatenate the list produced in the step above (d) with rightList.

Note that the above code traverses the tree in Postorder fashion. We can traverse in an inorder fashion also. We can first concatenate left subtree and root, then recur for the right subtree and concatenate the result with left-root concatenation.

How to Concatenate two circular DLLs?

-

Get the last node of the left list. Retrieving the last node is an O(1) operation since the prev pointer of the head points to the last node of the list.

- Connect it with the first node of the right list
- Get the last node of the second list
- Connect it with the head of the list.

Below are implementations of the above idea.

```cpp
// C++ Program to convert a Binary Tree
// to a Circular Doubly Linked List
#include<iostream>
using namespace std;

// To represents a node of a Binary Tree
struct Node
{
    struct Node *left, *right;
    int data;
};

// A function that appends rightList at the end
// of leftList.
Node *concatenate(Node *leftList, Node *rightList)
{
    // If either of the list is empty
    // then return the other list
    if (leftList == NULL)
        return rightList;
    if (rightList == NULL)
        return leftList;

    // Store the last Node of left List
    Node *leftLast = leftList->left;

    // Store the last Node of right List
    Node *rightLast = rightList->left;

    // Connect the last node of Left List
    // with the first Node of the right List
    leftLast->right = rightList;
    rightList->left = leftLast;

    // Left of first node points to
    // the last node in the list
    leftList->left = rightLast;

    // Right of last node refers to the first
    // node of the List
    rightLast->right = leftList;

    return leftList;
}

// Function converts a tree to a circular Linked List
// and then returns the head of the Linked List
```

```
Node *bTreeToCList(Node *root)
{
    if (root == NULL)
        return NULL;

    // Recursively convert left and right subtrees
    Node *left = bTreeToCList(root->left);
    Node *right = bTreeToCList(root->right);

    // Make a circular linked list of single node
    // (or root). To do so, make the right and
    // left pointers of this node point to itself
    root->left = root->right = root;

    // Step 1 (concatenate the left list with the list
    //         with single node, i.e., current node)
    // Step 2 (concatenate the returned list with the
    //         right List)
    return concatenate(concatenate(left, root), right);
}

// Display Circular Link List
void displayCList(Node *head)
{
    cout << "Circular Linked List is :\n";
    Node *itr = head;
    do
    {
        cout << itr->data <<" ";
        itr = itr->right;
    } while (head!=itr);
    cout << "\n";
}


// Create a new Node and return its address
Node *newNode(int data)
{
    Node *temp = new Node();
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// Driver Program to test above function
int main()
{
    Node *root = newNode(10);
    root->left = newNode(12);
    root->right = newNode(15);
    root->left->left = newNode(25);
    root->left->right = newNode(30);
    root->right->left = newNode(36);

    Node *head = bTreeToCList(root);
    displayCList(head);

    return 0;
}
```

Output

To be continued..

More questions will be added and you will get updated one ,later on..

Learn…. practice practice practice practice practice practice practice only practice……