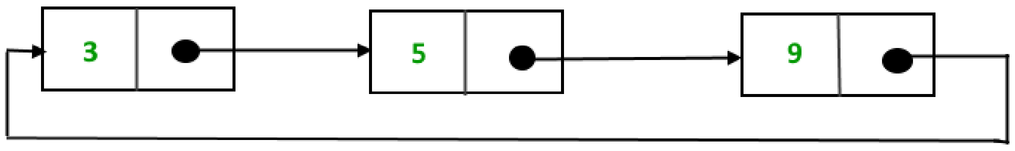
Insert at Begin of Circular Linked List

Why Circular? In a singly linked list, for accessing any node of the linked list, we start traversing from the first node. If we are at any node in the middle of the list, then it is not possible to access nodes that precede the given node. This problem can be solved by slightly altering the structure of a singly linked list. In a singly linked list, the next part (pointer to next node) is NULL. If we utilize this link to point to the first node, then we can reach the preceding nodes. Refer to this for more advantages of circular linked lists.

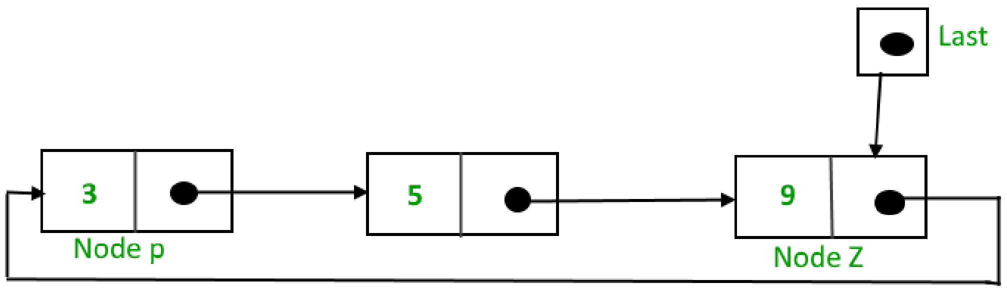
The structure thus formed is a circular singly linked list and looks like this:



In this post, the implementation and insertion of a node in a Circular Linked List using a singly linked list are explained.

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

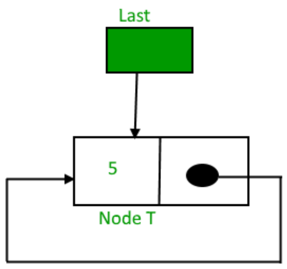
A node can be added in three ways:

- 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;
}
```

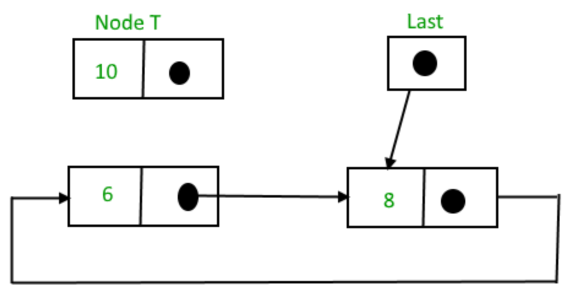
Insertion at the beginning of the list

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

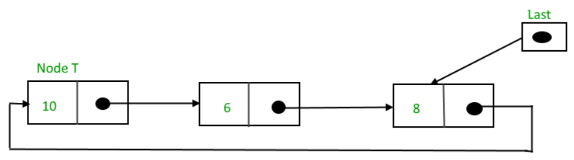
1. Create a node, say T.

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;
}
```

Time complexity : O(1) to insert node at beginning no need to traverse list it takes constant time

Space Complexity : O(1)

Insertion at the end of the list

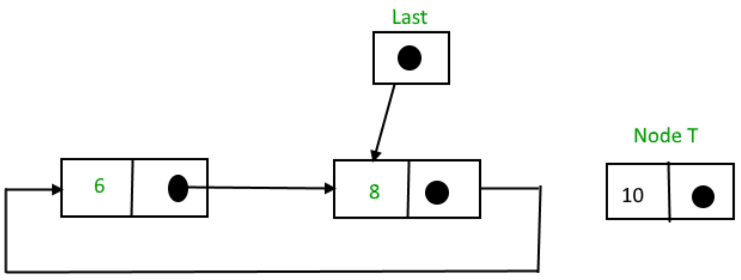
To insert a node at the end of the list, follow these steps:

1. Create a node, say T.

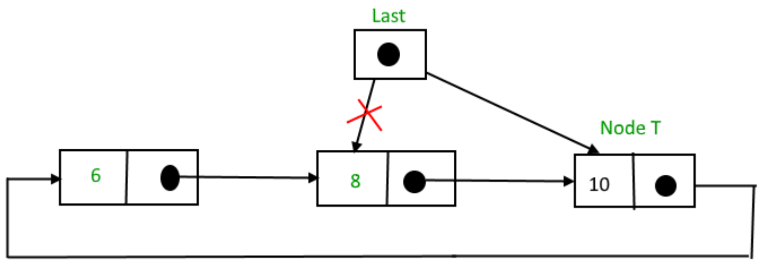
2. Make T -> next = last -> next;

3. last -> next = T.

4. last = T.



After insertion,



Function to insert a node at the end of the List

```
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;
}
```

Complexity analysis :

Time complexity : O(N) to insert at tail as it do not maintain any tail pointer so need to traverse till tail

Space Complexity : O(1)

Insertion in between the nodes

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

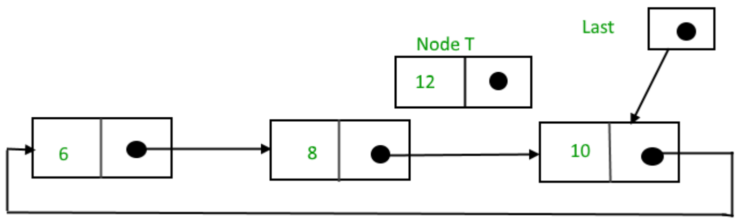
1. Create a node, say T.

2. Search for the node after which T needs to be inserted, say that node is P.

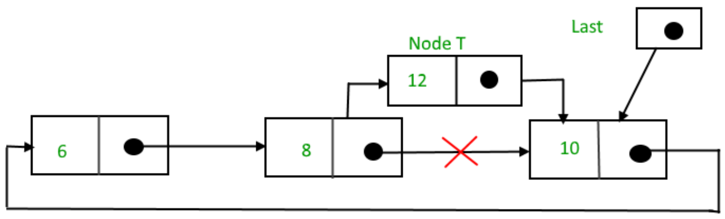
3. Make T -> next = P -> next;

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 specified position 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;
}
```

Complexity analysis :

Time Complexity : O(N)

Auxiliary Space : O(1)

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

```
#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));

    temp -> data = data;
    temp -> next = last -> next;
    last -> next = temp;

    return last;
}

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

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

    temp -> data = data;
    temp -> next = last -> next;
    last -> next = temp;
    last = temp;

    return last;
}

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

    struct Node *temp, *p;
    p = last -> next;
    do
    {
        if (p ->data == item)
        {
            temp = (struct Node *)malloc(sizeof(struct Node))
            temp -> data = data;
            temp -> next = p -> next;
            p -> next = temp;

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

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

}

void traverse(struct Node *last)
{
    struct Node *p;

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

    // Pointing to first Node of the list.
    p = last -> next;

    // Traversing the list.
    do
    {
        cout << p -> data << " ";
        p = p -> next;

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

}

// Driven Program
int main()
{
    struct Node *last = NULL;

    last = addToEmpty(last, 6);
    last = addBegin(last, 4);
    last = addBegin(last, 2);
    last = addEnd(last, 8);
    last = addEnd(last, 12);
    last = addAfter(last, 10, 8);

    traverse(last);

    return 0;
}
```

Output:

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

Complexity Analysis :

Time Complexity: O(N), for above code we need O(1) constant time to insert node at first position and to insert node at tail or end and in between list it will take O(N). so we ignore the constants so overall time complexity is **O(N)** for insertion operation in circular linked list.

Auxiliary Space: O(1), as we are not using any extra space.