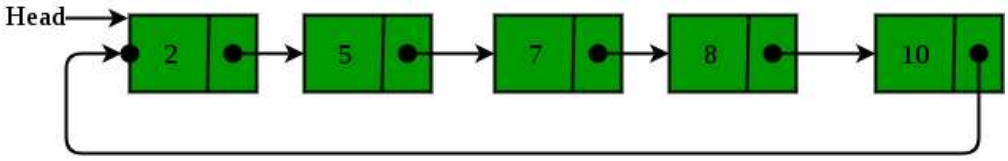
Lecture-10

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. Unlike this implementation, we don't need to maintain two pointers for front and rear if we use circular linked list. We can maintain a pointer to the last inserted node and front can always be obtained as next of last.

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.

Insertion in an empty List

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

After inserting a node T,

After insertion, T is the last node so pointer *last* points to node T. And Node T is first and last node, so T is pointing to itself.

Function to insert node in 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 *last =
          (struct Node*)malloc(sizeof(struct Node));

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

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

    return last;
}
```
Run on IDE

Insertion at the beginning of the list

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

After insertion,

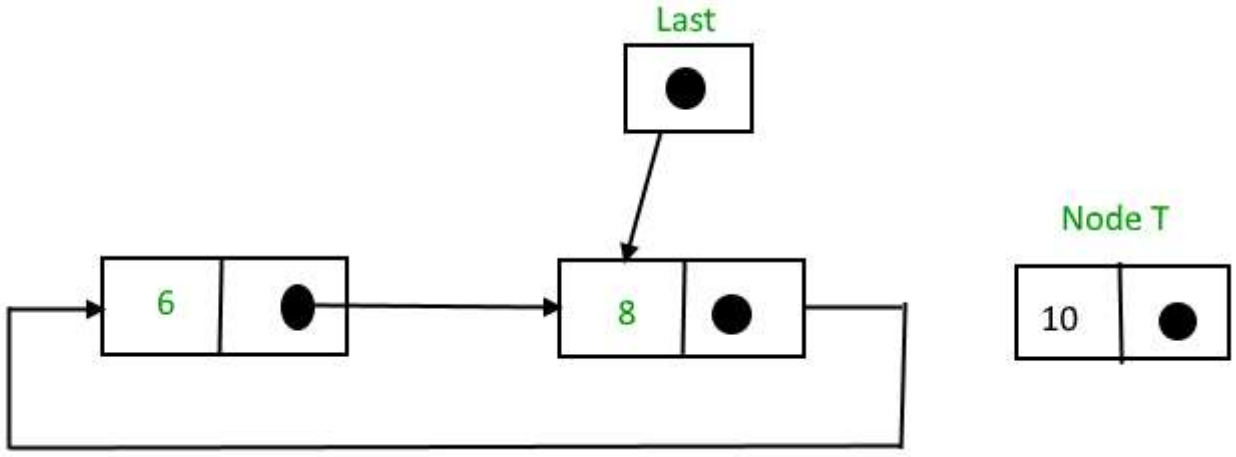
Function to insert node in the beginning of the List,

```
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;
```

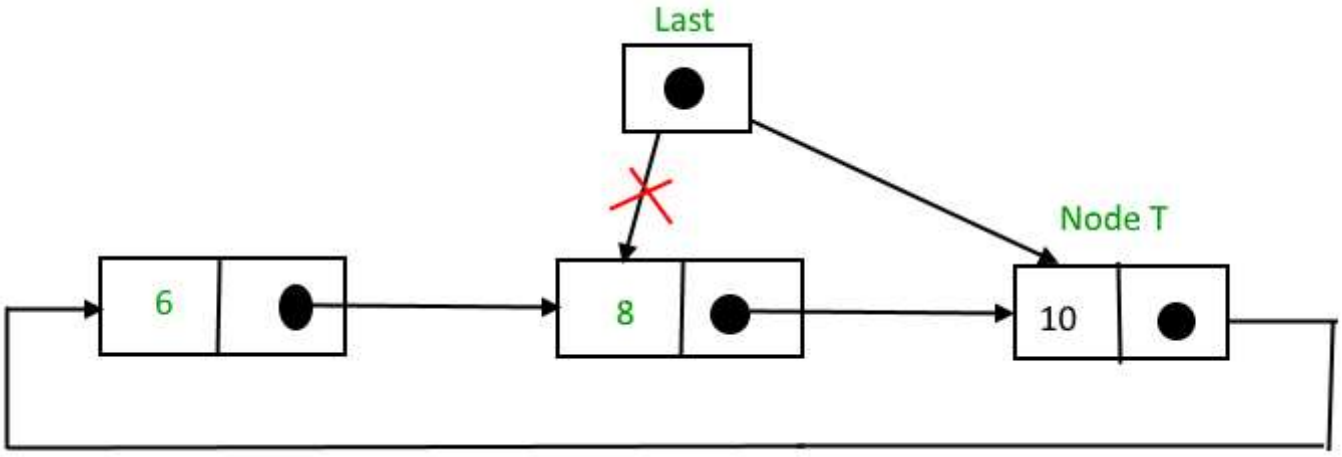
}

Insertion at the end of the list

To Insert a node at the end of the list, follow these step:
1. Create a node, say T.
2. Make T -> next = last -> next;
3. last -> next = T.
4. last = T.



After insertion,



Function to insert node in the end of the List,

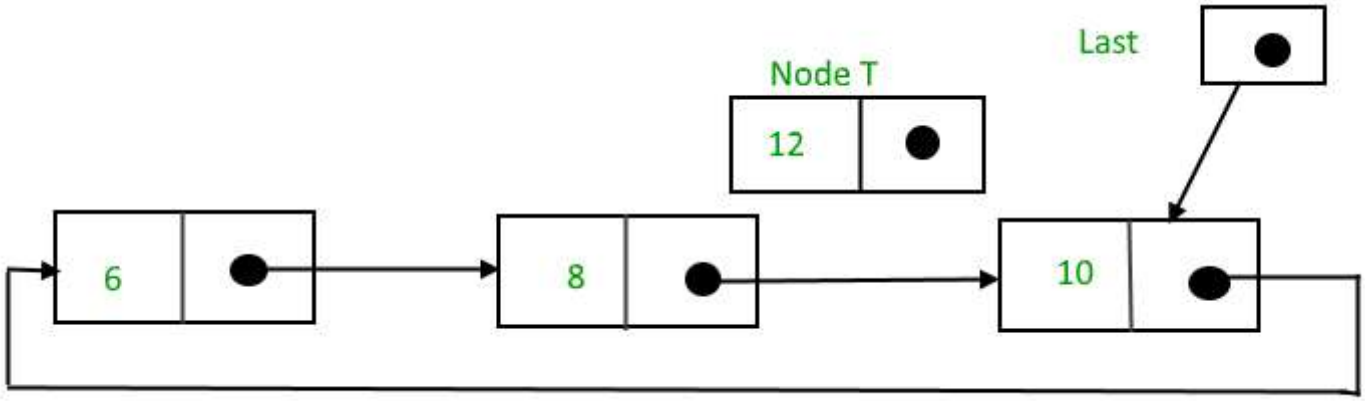
```
struct Node *addEnd(struct Node *last, int data)
{
  if (last == NULL)
     return addToEmpty(last, data);

  // 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;
}
```

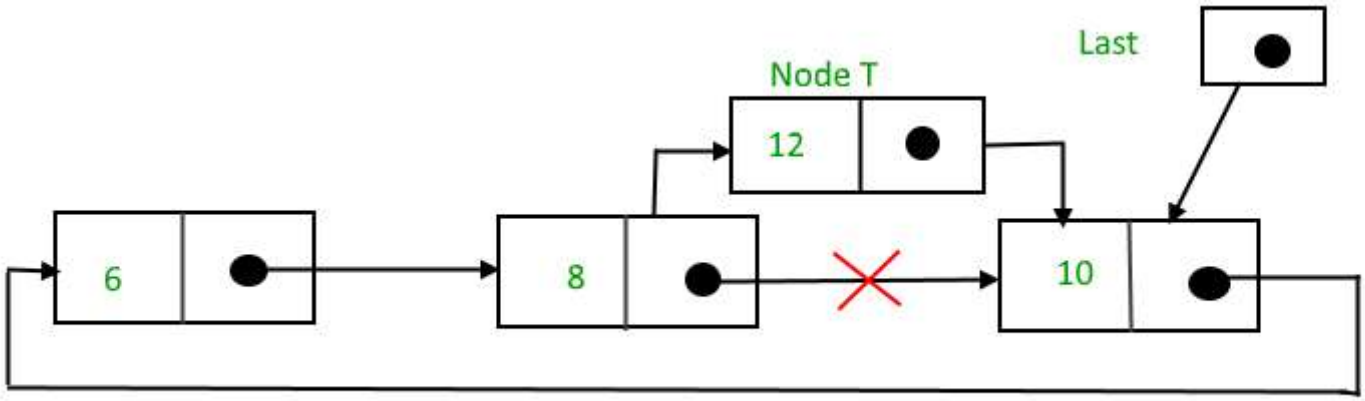
Insertion in between the nodes

To Insert a node at the end of the list, follow these step:
1. Create a node, say T.
2. Search the node after which T need to be insert, say that node be P.
3. Make T -> next = P -> next;
4. P -> next = T.

Suppose 12 need to be insert after node having value 10,



After searching and insertion,



Function to insert node in 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)
        {
            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;
}
```