Circular Linked Lists: Operations in C Language

codewithharry.com/videos/data-structures-and-algorithms-in-hindi-20

In the last tutorial, we learned about this new data structure, the circular linked lists. Additionally, we discussed the difference and similarities between a circular linked list and a linear linked list.

Let me quickly summarize some of the most important points:

1. Unlike singly-linked lists, a circular linked list has no node pointing to NULL. Hence it has no end. The last element points at the head node.
2. All the operations can still be done by maintaining an extra pointer fixed at the head node.
3. A circular linked list has a head node, but no starting node.

We even learned traversing through the circular linked list using the do-while approach. Today, we'll see one of the operations, insertion in a doubly-linked list with the help of C language.

Now, let's move on to the coding part. I have attached the snippet below. Refer to it while understanding the steps.

Creating the circular linked list:

1. Creating a circular linked list is no different from creating a singly linked list. One thing we do differently is that instead of having the last element to point to NULL, we'll make it point to the head.

2. Refer to those previous tutorials while creating these nodes and connecting them. This is the third time we are doing it, and I believe you must have gained that confidence.

```
struct Node
{
    int data;
    struct Node *next;
};
int main(){

    struct Node *head;
    struct Node *second;
    struct Node *third;
    struct Node *fourth;

    // Allocate memory for nodes in the linked list in Heap
    head = (struct Node *)malloc(sizeof(struct Node));
    second = (struct Node *)malloc(sizeof(struct Node));
    third = (struct Node *)malloc(sizeof(struct Node));
    fourth = (struct Node *)malloc(sizeof(struct Node));

    // Link first and second nodes
    head->data = 4;
    head->next = second;

    // Link second and third nodes
    second->data = 3;
    second->next = third;

    // Link third and fourth nodes
    third->data = 6;
    third->next = fourth;

    // Terminate the list at the third node
    fourth->data = 1;
    fourth->next = head;

    return 0;
}
```

Code Snippet 1: Creating the circular linked list

Traversing the circular linked list:

1. Create a void function *linkedListTraversal* and pass the head pointer of the linked list to the function.
2. In the function, create a pointer *ptr* pointing to the head.
3. Run a *do-while* loop until *ptr* reaches the last node, and ptr-> next becomes head, i.e. ptr->next = head. And keep printing the data of each node.

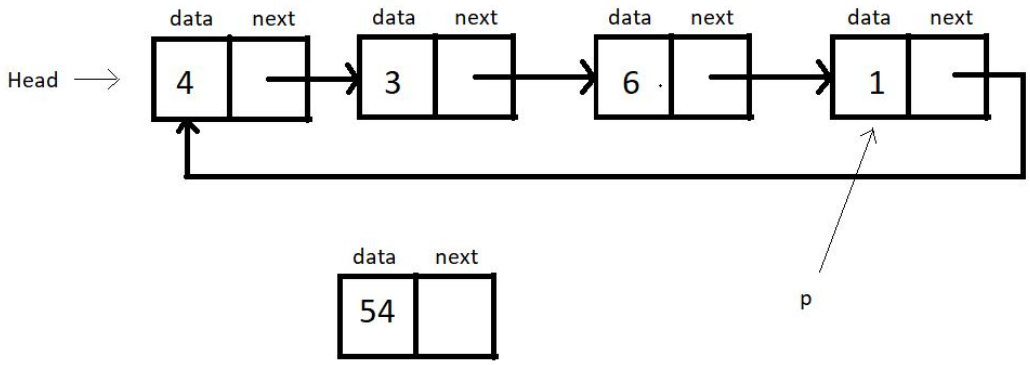
4. So, this is how we traverse through a circular linked list. And *do-while* was the key to make it possible.

```
void linkedListTraversal(struct Node *head){
    struct Node *ptr = head;
    do{
        printf("Element is %d\n", ptr->data);
        ptr = ptr->next;
    }while(ptr!=head);
}
```

Code Snippet 2: Traversing the circular linked list

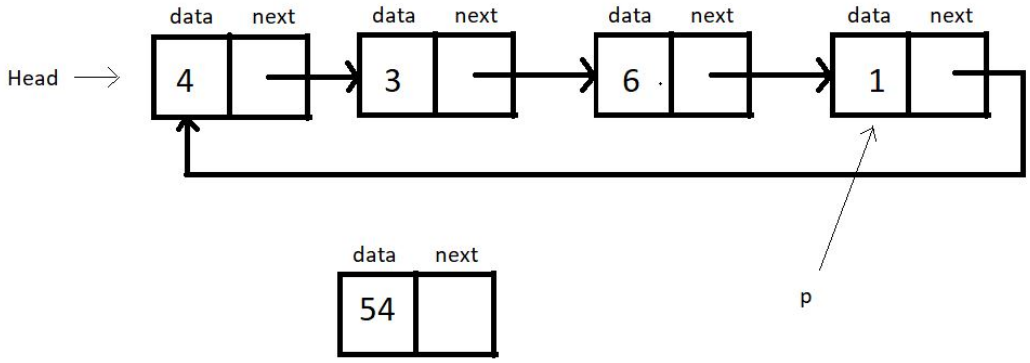
Inserting into a circular linked list:

1. I'll just cover the insertion part, and that too on the head. Rest of the variations, I believe, you'll be able to do yourselves. Things are very similar to that of singly-linked lists.
2. Create a struct Node* function *insertAtFirst* which will return the pointer to the new head.
3. We'll pass the current head pointer and the data to insert at the beginning, in the function.
4. Create a new struct Node* pointer *ptr*, and assign it a new memory location in the heap. This is our new node pointer. Make sure you don't forget to include the header file <stdlib.h>.
5. Create another struct node * pointer p pointing to the next of the head. *p = head->next*.
6. Run a *while* loop until the p pointer reaches the end element and p-> next becomes the head.



7. Now, assign *ptr* to the next of *p*, i.e.*p->next = ptr*. And *head* to the next of ptr, i.e. *ptr->next = head*.

8. Now, the new head becomes ptr. *head = ptr*.



9. Return head.

```
struct Node * insertAtFirst(struct Node *head, int data){
    struct Node * ptr = (struct Node *) malloc(sizeof(struct Node));
    ptr->data = data;

    struct Node * p = head->next;
    while(p->next != head){
        p = p->next;
    }
    // At this point p points to the last node of this circular linked list

    p->next = ptr;
    ptr->next = head;
    head = ptr;
    return head;

}
```

Code Snippet 3: Inserting into a circular linked list

Here is the whole source code:

```
#include<stdio.h>
#include<stdlib.h>

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

void linkedListTraversal(struct Node *head){
    struct Node *ptr = head;
    do{
        printf("Element is %d\n", ptr->data);
        ptr = ptr->next;
    }while(ptr!=head);
}

struct Node * insertAtFirst(struct Node *head, int data){
    struct Node * ptr = (struct Node *) malloc(sizeof(struct Node));
    ptr->data = data;

    struct Node * p = head->next;
    while(p->next != head){
        p = p->next;
    }
    // At this point p points to the last node of this circular linked list

    p->next = ptr;
    ptr->next = head;
    head = ptr;
    return head;

}

int main(){

    struct Node *head;
    struct Node *second;
    struct Node *third;
    struct Node *fourth;

    // Allocate memory for nodes in the linked list in Heap
    head = (struct Node *)malloc(sizeof(struct Node));
    second = (struct Node *)malloc(sizeof(struct Node));
    third = (struct Node *)malloc(sizeof(struct Node));
    fourth = (struct Node *)malloc(sizeof(struct Node));

    // Link first and second nodes
    head->data = 4;
    head->next = second;

    // Link second and third nodes
    second->data = 3;
    second->next = third;

    // Link third and fourth nodes
    third->data = 6;
    third->next = fourth;
```

```
    // Terminate the list at the third node
    fourth->data = 1;
    fourth->next = head;

    return 0;
}
```

Code Snippet 4: Insertion and traversal in a circular linked list

We'll now see whether the functions work accurately. Let's insert a few nodes at the beginning.

```
  printf("Circular Linked list before insertion\n");
    linkedListTraversal(head);
    head = insertAtFirst(head, 54);
    head = insertAtFirst(head, 58);
    head = insertAtFirst(head, 59);
    printf("Circular Linked list after insertion\n");
    linkedListTraversal(head);
```

Code snippet 5: Using the insertAtFirst function

Refer to the output below:

```
Circular Linked list before insertion
Element is 4
Element is 3
Element is 6
Element is 1
Circular Linked list after insertion
Element is 59
Element is 58
Element is 54
Element is 4
Element is 3
Element is 6
Element is 1
```

As you can see, all the elements we passed into the *insertAtFirst* function got added at the beginning. So, it is indeed working.

And this was all about a circular linked list. We won't go doing other operations. You should all carry out other operations yourselves. Believe me, it ain't a tough job. We have next in the series another variant of a linked list called the doubly linked lists.

Thank you for being with me throughout. I hope you enjoyed the tutorial. If you appreciate my work, please let your friends know about this course too. If you haven't checked out the whole playlist yet, move on to codewithharry.com or my YouTube channel to access it. See you all in the next tutorial, and we'll see another variant of linked lists, **Doubly Linked Lists**. Till then, keep learning.