Doubly Linked Lists Explained With Code in C Language

codewithharry.com/videos/data-structures-and-algorithms-in-hindi-21

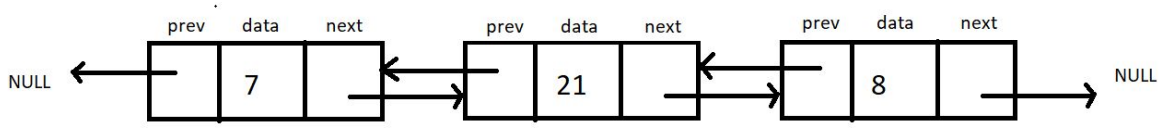
So, we have already talked about a lot of things under linked lists. We talked about the singly-linked lists, which had both a head node and the last node pointing to the NULL. We also talked about circular linked lists, which had no ending but an arbitrary head node. We also learned about all the basic operations (traversal, insertion, deletion, search) that we could do on both these variants of a linked list.

Our takeaway from all this is that we can perform all these operations on any variant of a linked list regardless of their structure and properties. We'll see one such thing today as well. We'll draw out similarities between the structure we'll handle today and the ones we did before.

What is a doubly-linked list?

Each node contains a data part and two pointers in a doubly-linked list, one for the previous node and the other for the next node.

Below illustrated is a doubly-linked list with three nodes. Both the end pointers point to the NULL.



How is it different from a singly linked list?

- A doubly linked list allows traversal in both directions. We have the addresses of both the next node and the previous node. So, at any node, we'll have the freedom to choose between going right or left.
- A node comprises three parts, the data, a pointer to the next node, and a pointer to the previous node.
- Head node has the pointer to the previous node pointing to NULL.

Implementation in C:

Let's try implementing a doubly linked list in our codes. We'll have a struct *Node* as before. The only information added to this struct *Node* is a struct *Node** pointer to the previous node. Let's name this *prev*.

This new information makes us travel in both directions, but using it follows the use of more memory space for a single node that now comprises three members. It is because of this we have a singly linked list.

```
struct Node {
    int data;
    Struct Node* next;
    Struct Node* prev;
};
```

Code Snippet 1: Implementation of a doubly linked list.

Operations on a Doubly Linked List:

The insertion and deletion on a doubly linked list can be performed by recurring pointer connections, just like we saw in a singly linked list.

The difference here lies in the fact that we need to adjust two-pointers (prev and next) instead of one (next) in the case of a doubly linked list. It very much follows the fact, "With great power, comes great responsibility." :)

Task: Try implementing a traversal algorithm to traverse once to the right and once to the left. Print the data in both cases.

So this was all we had in linked lists. It was a great segment. Let's give it an end here. We have so many things coming. So don't miss out on this ever. Keep revising things at regular intervals.

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. Don't forget to download the notes from the link given below. 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 where try to code these inserting methods. Till then, keep learning.

Download Notes here

Code as described/written in the video:

```
#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;

    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);
    return 0;
}
```