Insertion in a Linked List in C Language

codewithharry.com/videos/data-structures-and-algorithms-in-hindi-16

So, since we are already finished learning about all the cases one would have encountered while inserting a new node into a linked list, we can now code them individually in C language.

Before we code, let's recall all the cases:

1. Inserting at the beginning -> Time complexity: O(1)
2. Inserting in between -> Time complexity: O(n)
3. Inserting at the end -> Time complexity: O(n)
4. Inserting after a given Node -> Time complexity: O(1)

Let's now code. I have attached the snippet below. Refer to it while understanding the steps.

Understanding the snippet below:

1. So, the first thing would be to create a struct *Node*. This is a known thing to us. We have covered this in our traversal video.
2. Create the *linkedlistTraversal* function. Earlier tutorials can be referred to.
3. Do include the header file <stdlib.h>, since we'll be using malloc to reserve memory locations.
4. As we did last time, create the same four nodes, the first node being the *head*. Define a pointer to head node by *struct node* head*. And similarly for the other nodes. Request the memory location for each of these nodes from the heap via malloc. Link these nodes using the arrow operator.
5. Now that we have created a linked list, we can create functions according to the different cases.

Insertion at the beginning:

1. Create a struct Node* function *insertAtFirst* which will return the pointer to the new head.
2. We'll pass the current head pointer and the data to insert at the beginning, in the function.
3. Create a new struct Node* pointer *ptr*, and assign it a new memory location in the heap.
4. Assign head to the next member of the ptr structure using ptr-> next = head, and the given data to its data member.

5. Return this pointer *ptr*.

```
// Case 1
struct Node * insertAtFirst(struct Node *head, int data){
    struct Node * ptr = (struct Node *) malloc(sizeof(struct Node));
    ptr->data = data;

    ptr->next = head;
    return ptr;
}
```

Code Snippet 1: Implementing insertAtFirst.

Insertion in between:

1. Create a struct Node* function *insertAtIndex* which will return the pointer to the head.
2. We'll pass the current head pointer and the data to insert and the index where it will get inserted, in the function.
3. Create a new struct Node* pointer *ptr*, and assign it a new memory location in the heap.
4. Create a new struct Node* pointer pointing to *head*, and run a loop until this pointer reaches the index, where we are inserting a new node.
5. Assign p->next to the next member of the ptr structure using ptr-> next = p->next, and the given data to its data member.
6. Break the connection between p and p->next by assigning p->next the new pointer. That is, p->next = ptr.
7. Return head.

```
// Case 3
struct Node * insertAtEnd(struct Node *head, int data){
    struct Node * ptr = (struct Node *) malloc(sizeof(struct Node));
    ptr->data = data;
    struct Node * p = head;

    while(p->next!=NULL){
        p = p->next;
    }
    p->next = ptr;
    ptr->next = NULL;
    return head;
}
```

Code Snippet 2: Implementing insertAtIndex.

Insertion at the end:

1. Inserting at the end is very similar to inserting at any index. The difference holds in the limit of the while loop. Here we run a loop until the pointer reaches the end and points to NULL.
2. Assign NULL to the next member of the new ptr structure using ptr-> next = NULL, and the given data to its data member.

3. Break the connection between p and NULL by assigning p->next the new pointer. That is, p->next = ptr.
4. Return head.

```
// Case 2
struct Node * insertAtIndex(struct Node *head, int data, int index){
    struct Node * ptr = (struct Node *) malloc(sizeof(struct Node));
    struct Node * p = head;
    int i = 0;

    while (i!=index-1)
    {
        p = p->next;
        i++;
    }
    ptr->data = data;
    ptr->next = p->next;
    p->next = ptr;
    return head;
}
```

Code Snippet 3: Implementing insertAtEnd.

Insertion after a given node:

1. Here, we already have a struct Node* pointer to insert the new node just next to it.
2. Create a struct Node* function *insertAfterNode* which will return the pointer to the head.
3. Pass into this function, the head node, the previous node, and the data.
4. Create a new struct Node* pointer *ptr*, and assign it a new memory location in the heap.
5. Since we already have a struct Node* *prevNode* given as a parameter, use it as p we had in the previous functions.
6. Assign prevNode->next to the next member of the ptr structure using ptr-> next = prevNode->next, and the given data to its data member.
7. Break the connection between prevNode and prevNode->next by assigning prevNode->next the new pointer. That is, prevNode->next = ptr.
8. Return head.

```
// Case 4
struct Node * insertAfterNode(struct Node *head, struct Node *prevNode, int
data){
    struct Node * ptr = (struct Node *) malloc(sizeof(struct Node));
    ptr->data = data;

    ptr->next = prevNode->next;
    prevNode->next = ptr;

    return head;
}
```

Code Snippet 4: Implementing *insertAfterNode*.

So those were the cases we had in insertion. Below is the whole source code.

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

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

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

// Case 1
struct Node * insertAtFirst(struct Node *head, int data){
    struct Node * ptr = (struct Node *) malloc(sizeof(struct Node));
    ptr->data = data;

    ptr->next = head;
    return ptr;
}

// Case 2
struct Node * insertAtIndex(struct Node *head, int data, int index){
    struct Node * ptr = (struct Node *) malloc(sizeof(struct Node));
    struct Node * p = head;
    int i = 0;

    while (i!=index-1)
    {
        p = p->next;
        i++;
    }
    ptr->data = data;
    ptr->next = p->next;
    p->next = ptr;
    return head;
}

// Case 3
struct Node * insertAtEnd(struct Node *head, int data){
    struct Node * ptr = (struct Node *) malloc(sizeof(struct Node));
    ptr->data = data;
    struct Node * p = head;

    while(p->next!=NULL){
        p = p->next;
    }
    p->next = ptr;
    ptr->next = NULL;
    return head;
}

// Case 4
struct Node * insertAfterNode(struct Node *head, struct Node *prevNode, int data){
```

```
    struct Node * ptr = (struct Node *) malloc(sizeof(struct Node));
    ptr->data = data;

    ptr->next = prevNode->next;
    prevNode->next = 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 = 7;
    head->next = second;

    // Link second and third nodes
    second->data = 11;
    second->next = third;

    // Link third and fourth nodes
    third->data = 41;
    third->next = fourth;

    // Terminate the list at the third node
    fourth->data = 66;
    fourth->next = NULL;

    printf("Linked list before insertion\n");
    linkedListTraversal(head);
    // head = insertAtFirst(head, 56);
    // head = insertAtIndex(head, 56, 1);
    // head = insertAtEnd(head, 56);
    head = insertAfterNode(head, third, 45);
    printf("\nLinked list after insertion\n");
    linkedListTraversal(head);


    return 0;
}
```