J. P. DAWER INSTITUTE OF INFORMATION AND COMMUNICATION TECHNOLOGY

Project Submission of Data Structure Subject
(August 2024)

Linked List, Types of Linked List, Singly Linked List, Algorithms

...:::Guided By:::...

Maitri Hingu

...:::Prepared By:::...

Roll No	Name
1035	**Hetashvi Darji**
1047	**Mahek Dumasia**
1048	**Rahul Dusane**
1049	**Vaibhav Gabani**
1053	**Abhay Gami**

INDEX

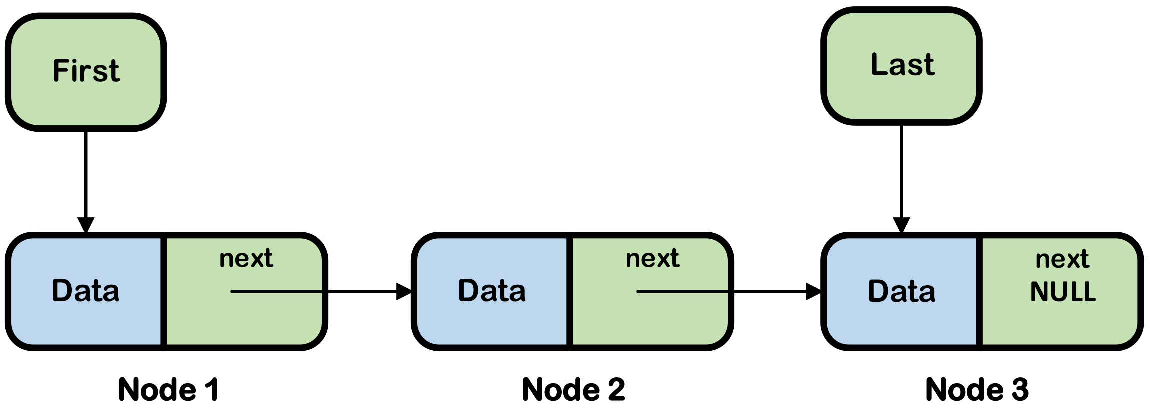
1. Introduction To Linked List

A linked list is a basic data structure that stores data in a sequence. Each unit of data, called a **Node**, has two parts: one part holds the actual data, and the other part holds a reference (or link) to the next node in the sequence.

This structure makes linked lists *flexible and easy to use when data needs to grow or shrink*, as they don't require memory to be stored in one continuous block like arrays do. The last node in the list points to a special value called NULL, which means there are no more nodes.

The linked list is usually controlled by a pointer called FIRST, which points to the first node. If FIRST is NULL, it means the list is empty. Because each node links to the next node of the same type, linked lists are called self-referential, which allows for efficient insertion and deletion operations.

There are also different types of linked lists, *each with its own characteristics and use cases*. The most common types are singly linked lists, doubly linked lists, circular linked lists and double circular linked list.



1.1 Basic Operations On Linked List

Insertion:

Insertion involves adding a new node into the linked list. Depending on where you want to add the new node, the insertion operation can be categorized as follows:

- **At the Beginning (First)**:

  This operation adds a new node before the current head of the list. The new node becomes the new head, and its next pointer points to the former head.

- **At the End (Last)**:

  This operation adds a new node after the current last node (tail) of the list.

- **In the Middle**:

  This operation inserts a new node after a specific node in the list.

Deletion:

Deletion involves removing a node from the linked list. Depending on where you want to remove the node, the deletion operation can be categorized as follows:

- **From the Beginning (First)**:

  This operation removes the head node from the list.

From the End (Last):

This operation removes the last node from the list.

From the Middle:

This operation removes a node that is neither at the head nor at the tail.

Traversal:

Traversal is the process of visiting each node in the linked list in sequence from the head to the end. It is essential for operations like searching and updating.

Search:

Search involves finding a node with a specific value or condition within the linked list.

Update:

Update involves changing the data stored in a node once it has been located.

1.2 Application of Linked List

1. **Dynamic Memory Allocation :**

   Linked lists allow for dynamic memory allocation by adding or removing nodes as needed, avoiding the need for a contiguous block of memory like arrays.

2. **Memory Management :**

   Used in memory management systems to track free and allocated memory blocks, helping to efficiently allocate and deallocate memory.

3. **Navigation Systems :**

   Useful for representing navigable structures like directories and files, where each node points to its contents, allowing easy traversal.

4. **Sparse Matrix Representation :**

   Efficiently represents matrices with many zero elements by storing only non-zero elements and their positions in a linked list, reducing memory usage.

5. **Undo Functionality in Software :**

   Stores previous states or changes in a linked list,enabling software to revert to previous states when undoing actions.

1.3 Advantages And Disadvantages

Advantages :

- Unlike arrays, linked lists can grow and shrink as needed. You don't have to decide on a fixed size in advance, which is helpful when dealing with unknown amounts of data.
- Adding or removing elements from a linked list is quicker when compared to arrays, especially at the beginning or in the middle. This is because you only need to change a few pointers instead of shifting elements, which can save time in large data sets.
- Linked lists only use as much memory as they need. Unlike arrays, which may reserve memory that might go unused, linked lists allocate memory as you add new elements, making them efficient in terms of space.

Disadvantages :

- Each element in a linked list needs extra memory to store a reference (or pointer) to the next element. This additional overhead means linked lists can use more memory than arrays when storing the same number of elements.
- To access an element in a linked list, you have to start from the beginning and move through each element one by one until you find what you're looking for. This sequential access is slower than arrays, where you can jump directly to any element using its index.
- Some operations, like searching for an element or reversing the list, are more complicated in linked lists than in arrays. These tasks often involve careful pointer manipulation and more steps to complete.

- Linked lists store elements in different memory locations. This scattered storage means that they do not take advantage of cache memory, which can make accessing data slower.

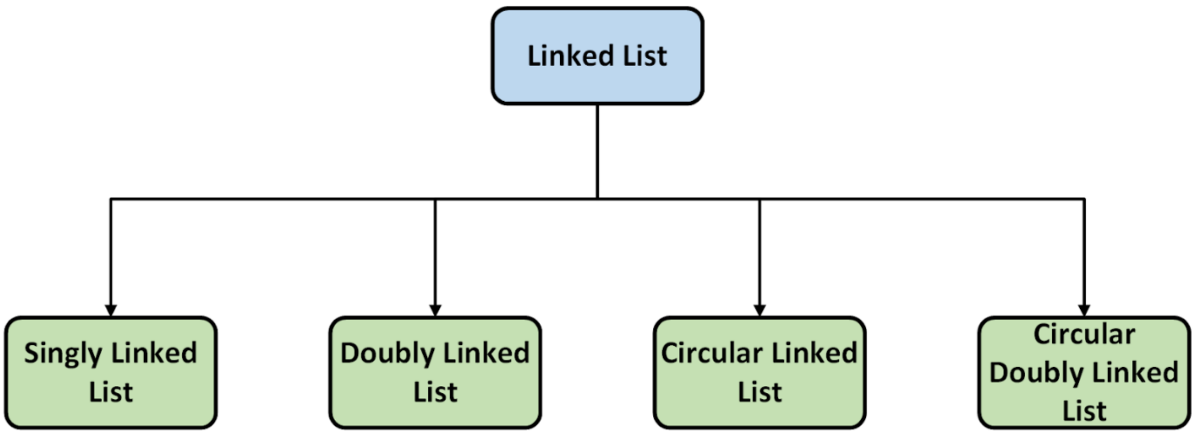
What is Difference between Array & Linked List ?

Array	Linked List
Arrays store elements in a contiguous block of memory. This means that all elements are placed sequentially in memory.	Linked lists consist of nodes where each node contains a reference to the next node. Nodes are scattered throughout memory and linked together via pointers.
The total amount of memory required must be allocated upfront. If the array's capacity is exceeded, resizing involves creating a new larger array and copying the elements from the old array.	Memory is allocated for each node individually as needed. There is no need to allocate a large block of memory initially.
Arrays allow for constant time access to elements because the address of each element can be calculated directly using the index.	Accessing an element in a linked list requires traversing the list from the head to the desired position, resulting in linear time O(n) access time.
Inserting or deleting elements (except at the end) generally involves shifting elements to maintain contiguous storage, leading to O(n) time complexity.	Inserting or deleting nodes can be done in constant time O(1) if you have a reference to the node (or its predecessor) where the operation should occur.

2. Types Of Linked List

2.1 Types Of Linked List

Basically, Linked List divides into 4 different types.



❶ Singly Linked List :

In a singly linked list, each node contains data & pointer where,

➜ **Data**: The value stored in the node.
➜ **Next pointer**: A reference to the next node in the list.

Nodes are connected in one direction, from the first node to the last node, where the last node points to NULL.

Types Of Linked List

➔Real life Example:

Fee Counter Queue:

People wait in line, each link to the next, reflecting a singly linked list.

Structure of Node of Singly Linked List:

```
struct Node {
    int peopleId;
    struct Node* next;
};
```

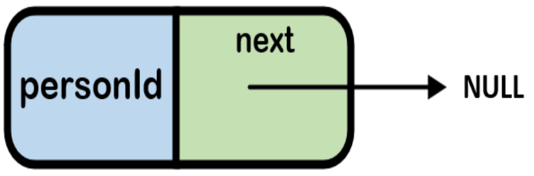
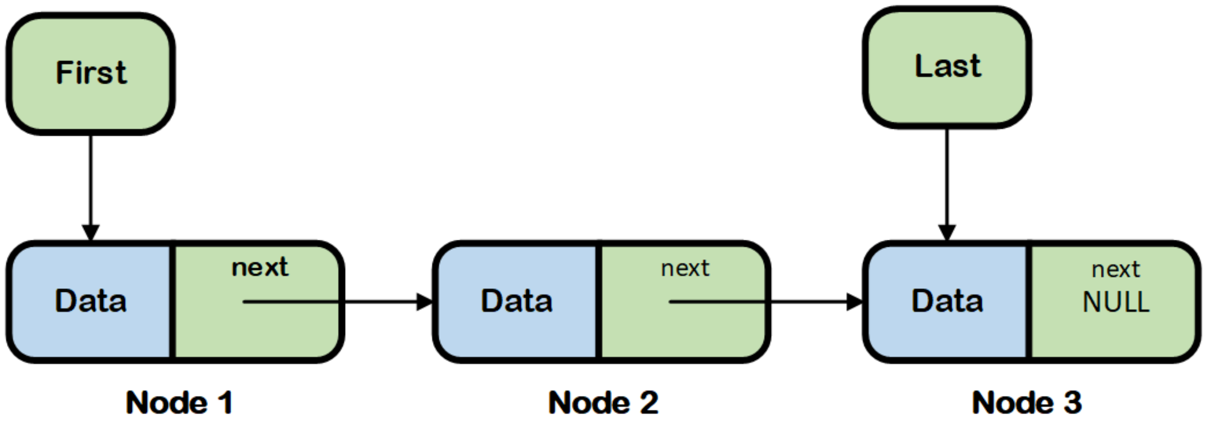


Diagram of Singly Linked List:



❷ Doubly Linked List:

In a doubly linked list, each node contains:

- **Data**: The value stored in the node.
- **Next pointer**: A reference to the next node.
- **Previous pointer**: A reference to the previous node.

  - Nodes are connected in both directions, allowing traversal forwards and backwards.

  - First Node's previous pointer and Last Node's next pointer both points to NULL.

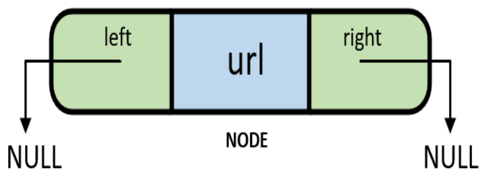
Real life Example:

Browser History:

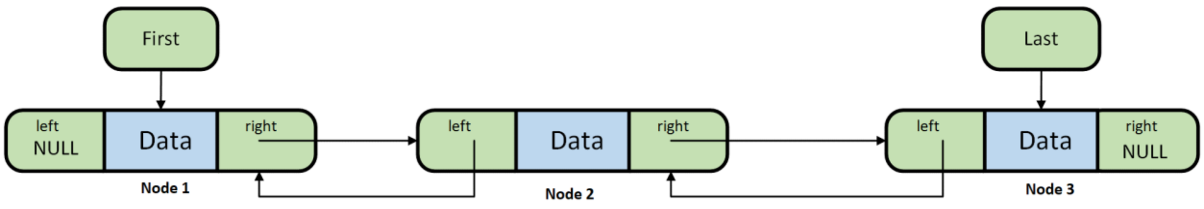
You can move forward and backward through the pages.

Structure of Node of Doubly Linked List:

```
struct Node {
    char url[25];
    struct Node* prev;
    struct Node* next;
};
```



✚ Diagram of Doubly Linked List:



❸ Circular Linked List:

In a Circular Linked List, each node contains:

➔ **Data:** The value stored in the node.

➔ **Next pointer:** A reference to the next node in the list.

- In this list, the last node's next pointer does not point to NULL, but rather to the first node (First), forming a circular loop.

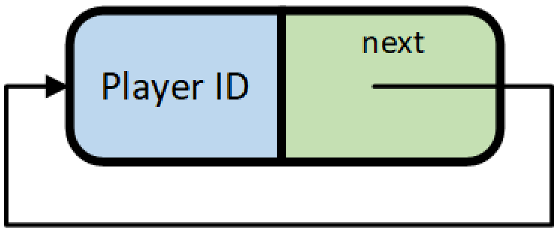
➔ Real life Example:

◉ Card Dealing in a Circle:

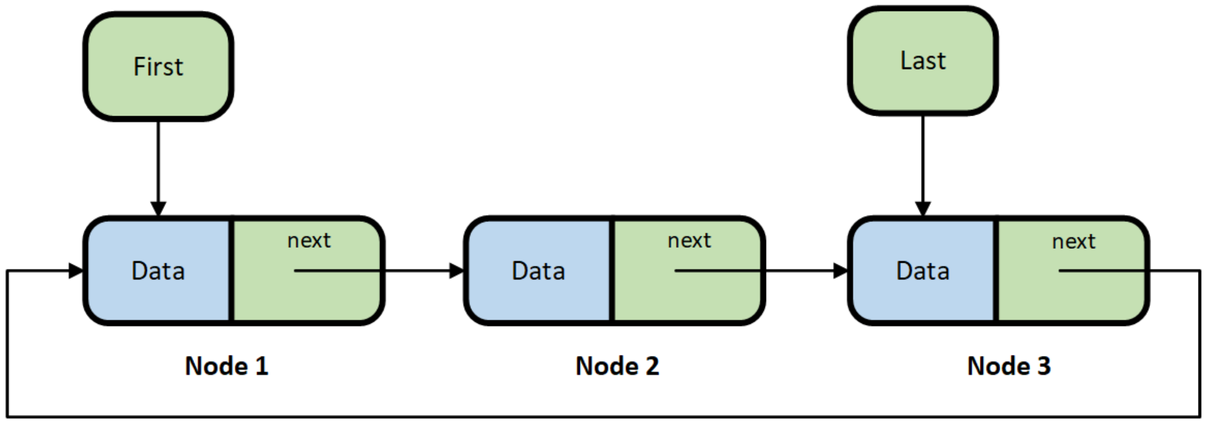
The dealer distributes cards in a circle, and after the last player, the next card goes to the first again, forming a loop.

Structure of Node of Circular Linked List:

```
struct Node {
    int playerID;
    struct Node* next;
};
```



✚ Diagram of Circular Linked List:



❹ Circular Doubly Linked List:

In a **Circular Doubly Linked List**, each node contains:

➜ **Data**: The value stored in the node.

➔ **Next pointer**: A reference to the next node in the list.

➔ **Previous pointer**: A reference to the previous node in the list.

- In this list, the last node's next pointer points to the first node, and the first node’s previous pointer points to the last node, forming a circular loop in both directions

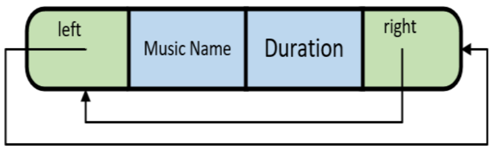
➔ Real Life Example:

Playlist Management:

In a music player with a playlist, you can move both forward and backward through the songs. After the last song, you loop back to the first, and vice versa.

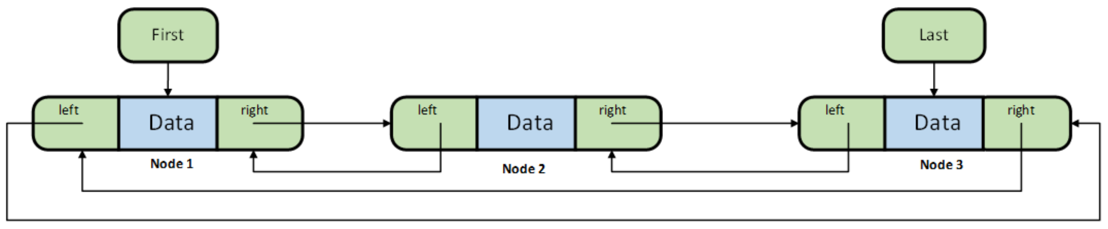
Structure of Node of Circular Doubly Linked List:

```
struct Node {
    char musicName[25];
    int duration;
    struct Node* next;
    struct Node* prev;
};
```



Types Of Linked List

✚ Diagram of Circular Doubly Linked List:

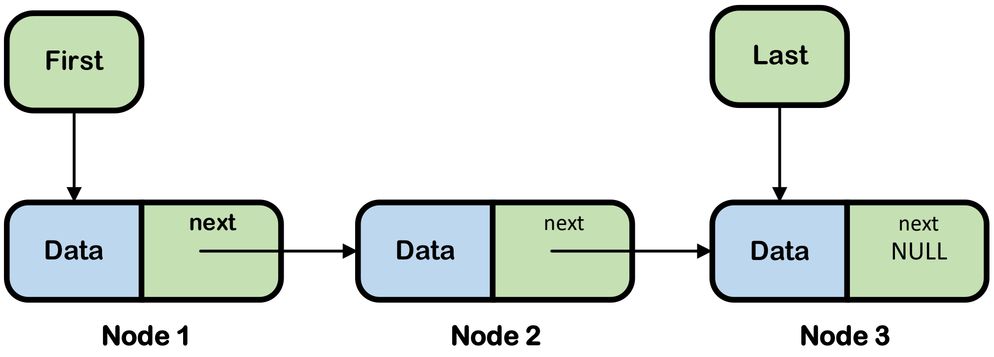


3. Singly linked list

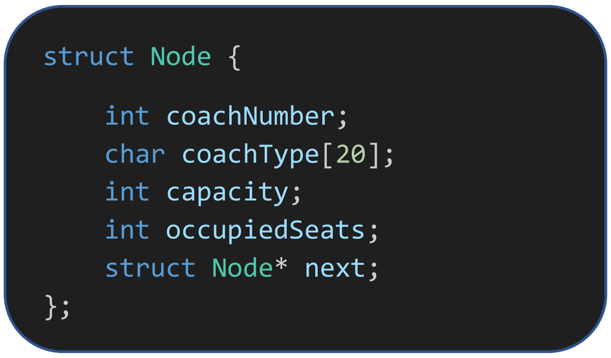
1) Singly linked list:

The node contains a pointer to the next node means that the node stores the address of the next node in the sequence. A singly linked list allows the traversal of data only in one way. The first node is called the FIRST; it points to the first node of the list and helps us access every other element in the list. The last node is called the LAST, points to NULL which helps us in determining when the list ends.

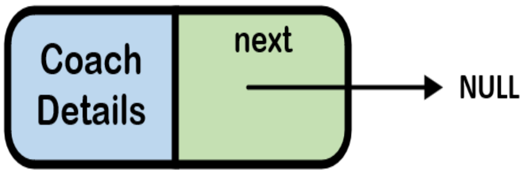
✚ Diagram:



➔ Example:



```
struct Node {
    int coachNumber;
    char coachType[20];
    int capacity;
    int occupiedSeats;
    struct Node* next;
};
```



Advantages Of Singly Linked List:

1. **Flexible Size:** Singly linked lists can grow and shrink easily, so you don't need to decide the size in advance like you do with arrays.
2. **Easy Insertion/Deletion:** Adding or removing elements in a singly linked list is simple, especially at the beginning or in the middle, without having to shift other elements.
3. **Smart Memory Use:** Memory is used only when a new node is created, which helps avoid wasting space.
4. **Great for Complex Structures:** Singly linked lists are easy to understand and use, making them a good choice for learning dynamic data structures.
5. **Steady Growth:** They don't need to be resized or moved to a bigger space as they grow.

Disadvantages Of Singly Linked List:

1. **Extra Memory:** Each node needs extra memory for storing the link to the next node, which can add up if the list is large.
2. **Slower Searches:** To find a specific element, you have to start from the first node and move one by one, making it slower than arrays where you can directly access any element.

3. **Complexity:** Simple tasks can be more complicated compared to using arrays.

4. **Slow Search:** Since elements aren't stored together, Finding an item in the list can be slow since you have to check each node one by one.

5. **One-Way Only:** You can only move forward through the list. If you need to go back, you'll have to use a different type of list, like a doubly linked list.

Tracing in Singly Linked List:

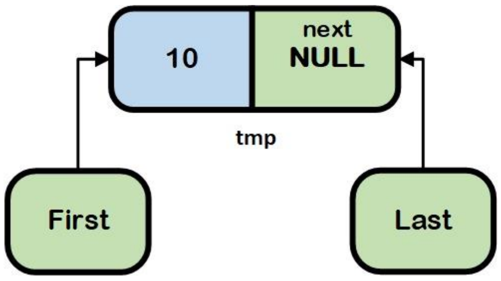
1. Insertion First Tracing:

```
void insertFirst() {
    tmp = (struct Node*)malloc(sizeof(struct Node));
    printf("Enter element You want to insert first = ");
    scanf("%d",&tmp->data);
    tmp->next = First;

    if(First == NULL) {
        First = Last = tmp;
    } else {
        First = tmp;
    }
}
```

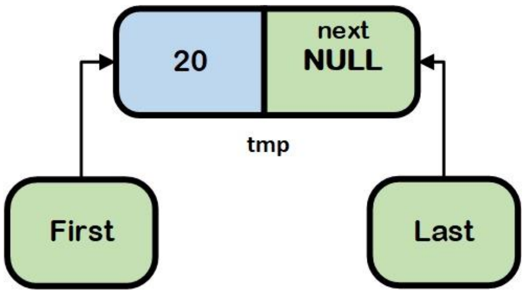
Case-1:

There is no data inside Linked list (suppose Linked list is NULL) at this situation let us perform insert first operation for
insert 10 first.

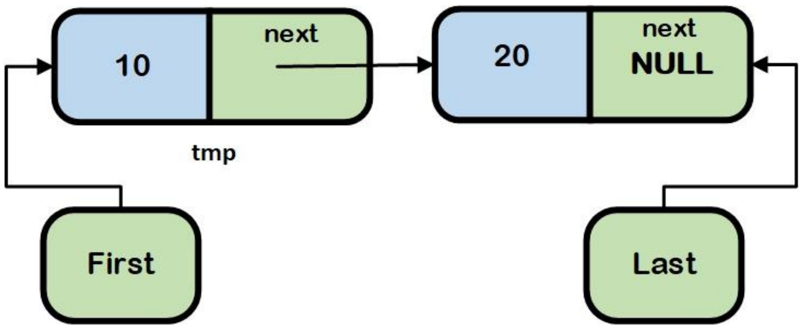


Case-2:

There is data inside Linked list ,



at this situation let us perform insert first operation for
insert 10 first.



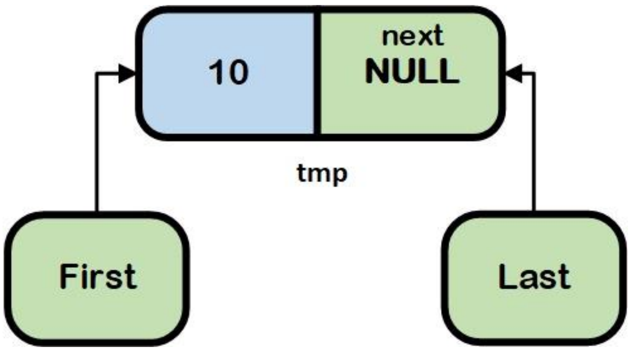
2. Insertion Last tracing:

```
void insertLast() {
    tmp = (struct Node*)malloc(sizeof(struct Node));
    printf("Enter element You want to insert Last = ");
    scanf("%d",&tmp->data);
    tmp->next = NULL;

    if(First == NULL) {
        First = Last = tmp;
    } else {
        Last->next = tmp;
        Last = tmp;
    }
```

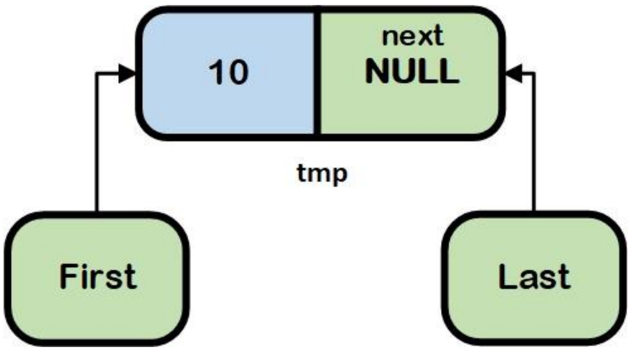
Case 1:

There is no data inside Linked list (suppose Linked list is NULL) at this situation let us perform insert last operation for **insert 10 last.**

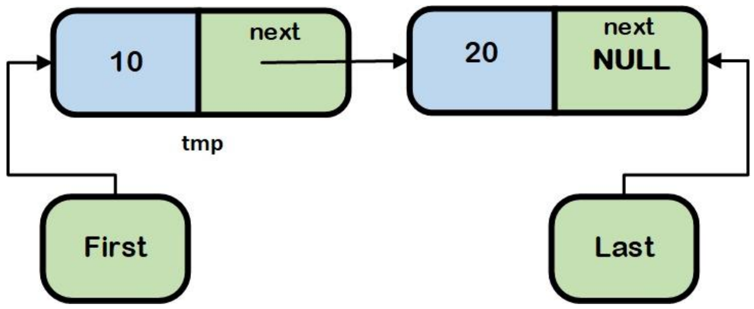


Case 2:

There is data inside Linked list ,



at this situation let us perform insert last operation for **insert 20 last.**



3. Insertion After tracing:

```c
void insertAfter() {
    if(First == NULL) {
        printf("\nLinked List is empty \nInsert After Operation can't be
done . ");
        return;
    }
    int index,cnt = 1;
    Curr = First;
    printf("\nAfter which index you want to insert data = ");
    scanf("%d",&index);
    while(Curr->next != NULL && cnt < index) {
        Curr = Curr->next;
        cnt ++;
    }
    if(cnt == index) {
        tmp = (struct Node*)malloc(sizeof(struct Node));
        printf("Enter element you want to insert = ");
        scanf("%d",&tmp->data);
        tmp->next = Curr->next;
        Curr->next = tmp;
        if(Curr == Last) {
            Last = tmp;
        }
    } else {
        printf("%d index is not found in Linked List . ",index);
    }
}
```

Case 1:

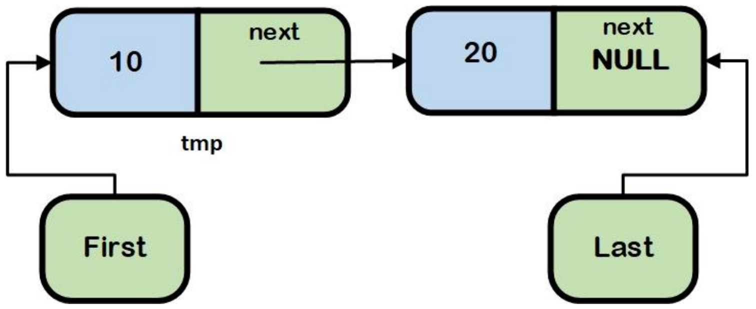
If Linked list is empty then insert after operation can’t be done because there is no index after which we can insert data.

Case 2:

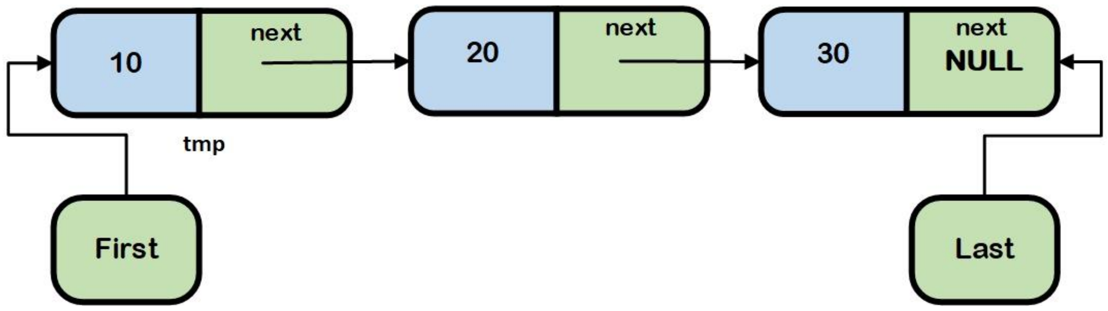
If user enters index number which is not existing in linked list (For example user wants to insert element after index 4 but Linked list having only 2 index) then insert after operation can’t be done.

Case 3:

If we want to perform insert after operation at after last index so suppose Linked List having data like

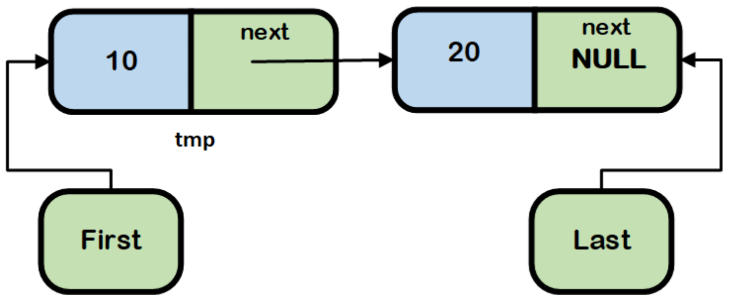


Insert 30 after index 2.

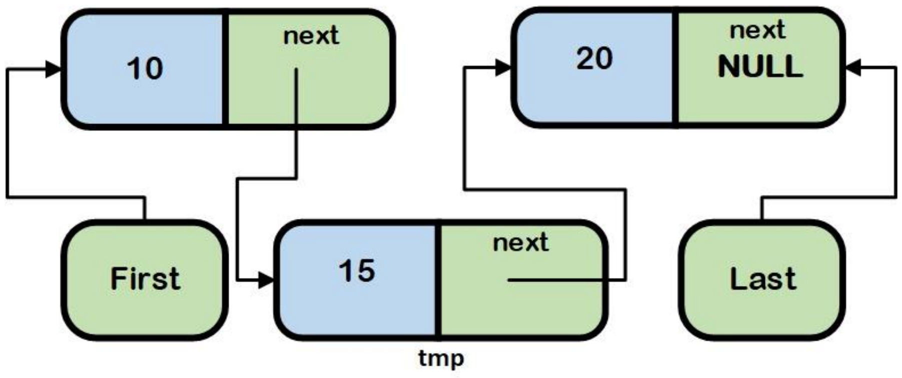


Case 4:

If we want to perform insert after operation between first and last index so suppose Linked List having data like



Insert 15 after index 1.



4. Delete First Tracing:

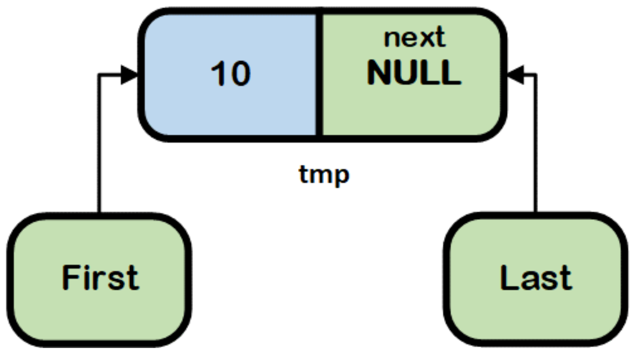
```c
void deleteFirst() {
    if(isempty()) {
        printf("\nLinked List is empty\nDelete First can't be done . ");
        return;
    }
    tmp = First;
    printf("\nData %d deleted from linked list ",tmp->data);
    if(First == Last) {
        First = Last = NULL;
    } else {
        First = First->next;
    }
    free(tmp);
}
```

Case 1:

If Linked List is empty then delete First operation can’t be done.

Case 2:

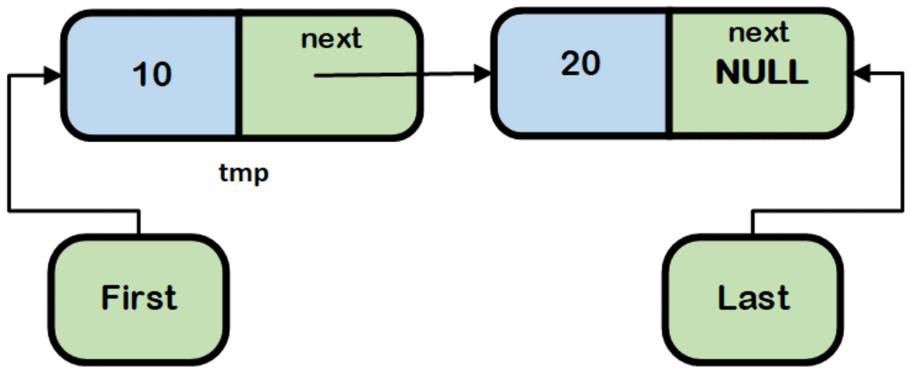
If Linked List having only one Node, then after performing Delete First operation Linked List will became empty.



After performing delete First operation both First and Last pointer will point to **NULL**.

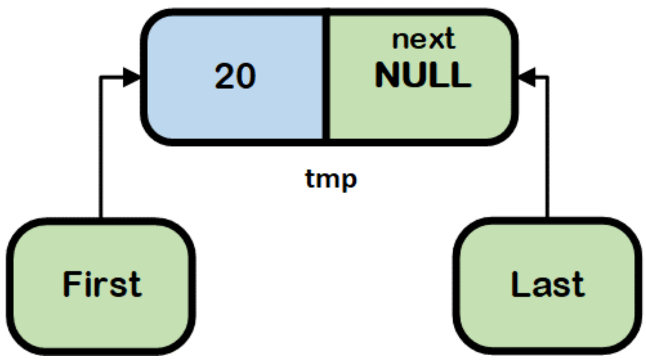
Case 3:

If Linked List Having more than one Node,



After performing Delete First operation Linked List will be as below.

Singly Linked List



5. Delete Last tracing:

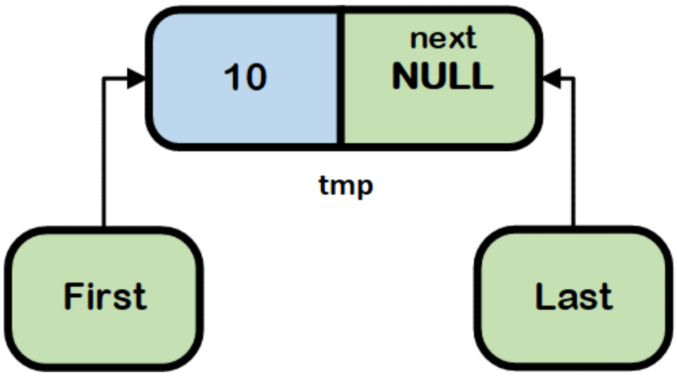
```c
void deleteLast() {
    if(isempty()) {
        printf("\nLinked List is empty\nDelete Last can't be done . ");
        return;
    }
    tmp = Last;
    printf("\nData %d deleted from linked list ",tmp->data);
    if(First == Last) {
        First = Last = NULL;
    } else {
        Curr = First;
        while(Curr->next != Last) {
            Curr = Curr->next;
        }
        Curr->next = NULL;
        Last = Curr;
    }
    free(tmp);
}
```

Case 1:

If Linked List is empty then delete Last operation can’t be done.

Case 2:

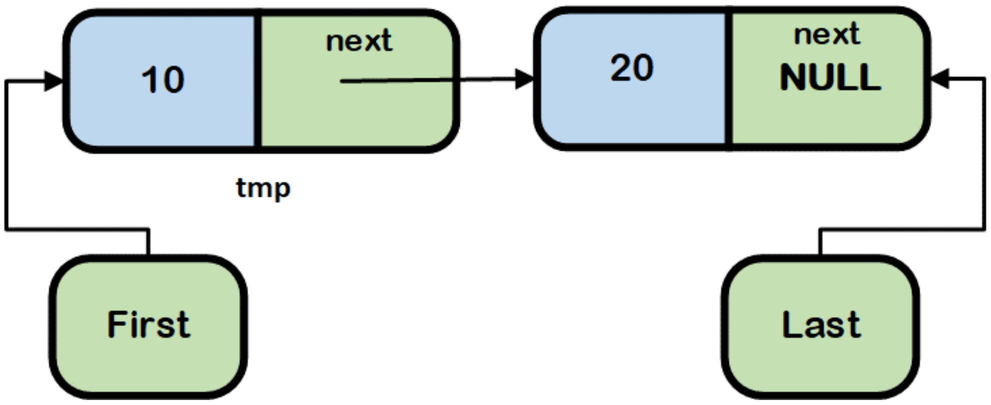
If Linked List having only one Node,



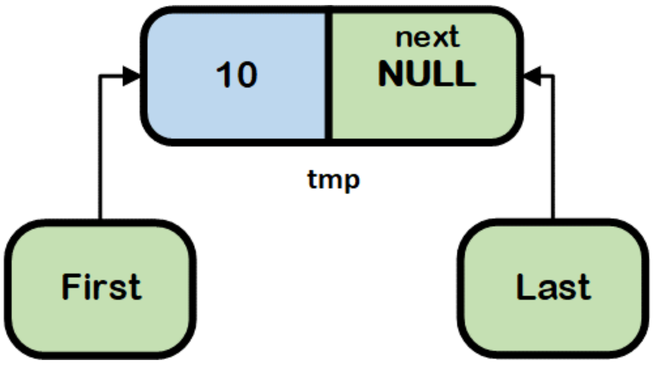
After performing Delete Last operation Linked List will became empty and both First and Last pointer will point to **NULL**.

Case 3:

If Linked List having more than one Nodes,



After performing Delete Last operation Linked List will be as below.



6.Delete Particular tracing:

```c
void deleteParticular() {
    if(isempty()) {
        printf("\nLinked List is empty \nElement can't be deleted . ");
        return;
    }
    int index , cnt = 1;
    printf("Enter which index you want to delete from Linked List = ");
    scanf("%d",&index);
    tmp = First;
    while(tmp->next != NULL && cnt < index) {
        cnt++ ;
        Curr = tmp ;
        tmp = tmp->next ;
    }
    if(cnt == index) {
        printf("\n%d element deleted from Linked List . ",tmp->data);
        if(First == Last) {
            First = Last = NULL;
        } else if(tmp == First) {
            First = First->next ;
        } else if(tmp == Last) {
            Curr->next = NULL;
            Last = Curr;
        } else {
            Curr->next = tmp->next;
        }
    } else {
        printf("\n%d index not found in Linked List . ",index);
    }
    free(tmp);
}
```

Case 1:

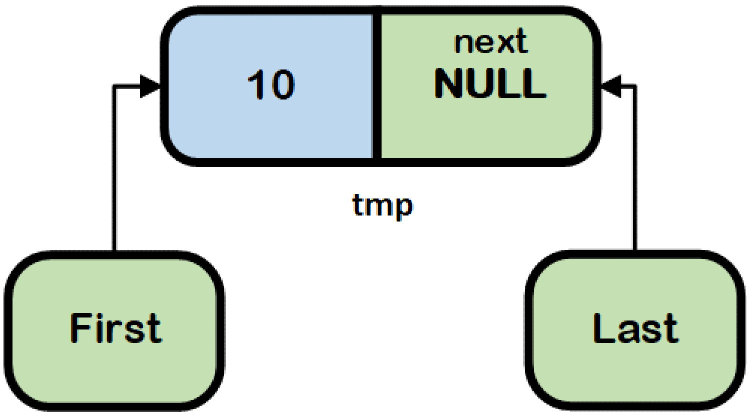
If Linked List is empty then Delete Particular operation can’t be done.

Case 2:

If index which user want to delete is not available in Linked List then Delete Particular operation can’t be done.

Case 3:

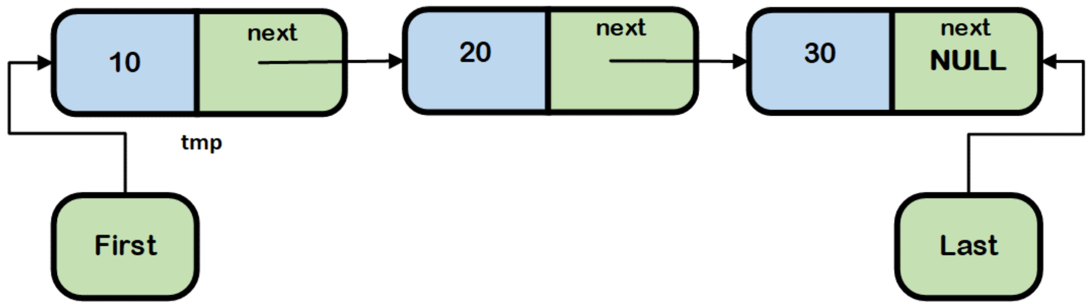
If index is available in linked List and Linked List having only one Node, (here index = 1)



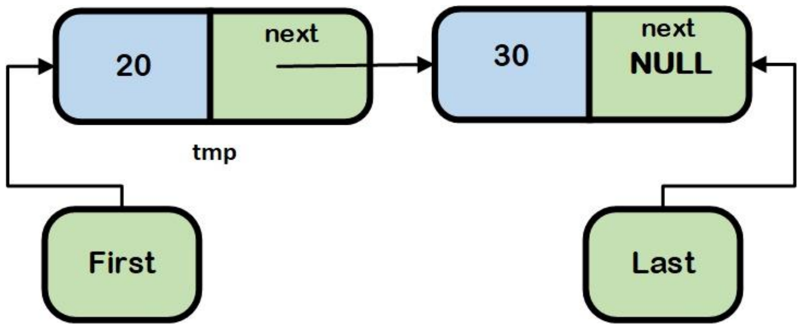
After Delete Particular operation, Linked List will became empty and both First and Last pointers will point to **NULL**.

Case 4:

If index is available in Linked list and Linked List having more than one Nodes, at this situation if index is equals to Linked List’s Node’s index which is point by First pointer, (here index = 1)

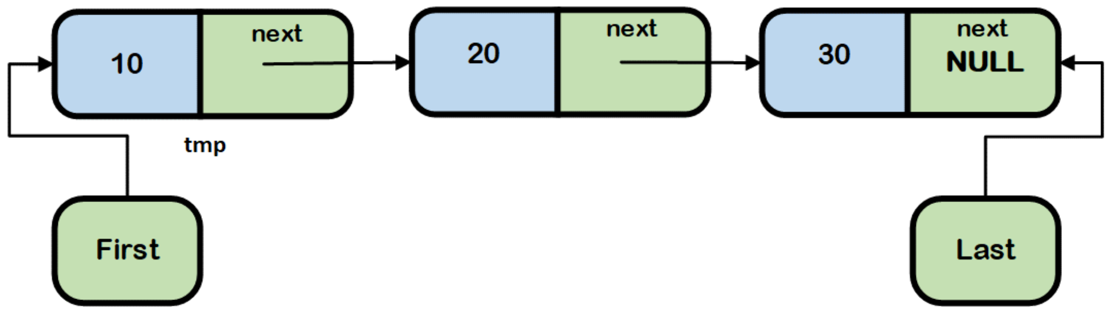


After performing Delete Particular operation Linked List will be as below.

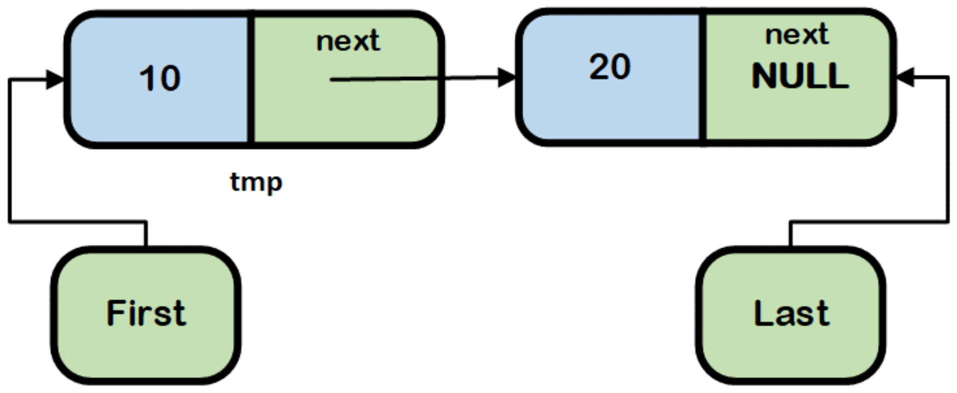


Case 5:

If index is available in Linked list and Linked List having more than one Nodes, at this situation if index is equals to Linked List's Node's index which is point by Last pointer, (here index = 3)

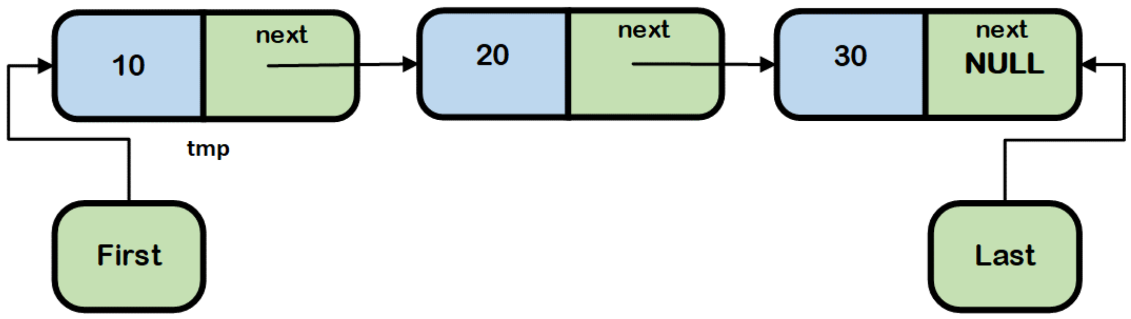


After performing Delete Particular operation Linked List will be as below.

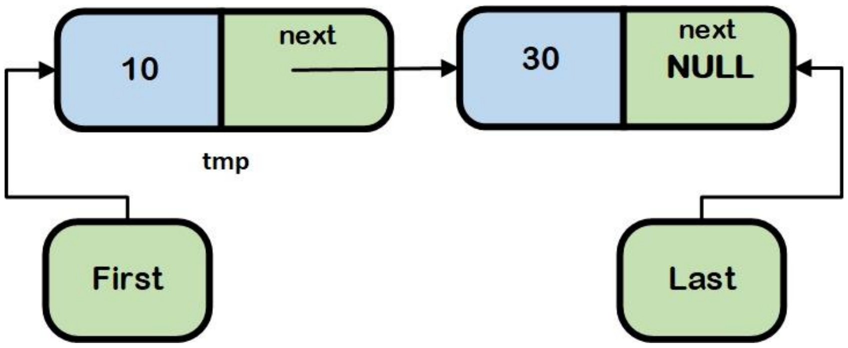


Case 6:

If index is available in Linked list and Linked List having more than one Nodes, at this situation if index is equals to Linked List’s Node’s index which is not point by First and Last pointer, (here index = 2)



After performing Delete Particular operation Linked List will be as below.



7.Linked List traversal tracing:

```
void traverse() {
    if(isempty()) {
        printf("\nLinked List is empty . ");
        return;
    }
    Curr = First;
    printf("\nLinked List : ");
    while(Curr != NULL) {
        printf("| %d |->",Curr->data);
        Curr = Curr->next;
    }
    printf(" /\n");
}
```

✚ Algorithms:

Algorithm for checking linked list empty or not:

```
Step 1: Start
Step 2: Check whether linked list empty or not.
         If FIRST pointer is pointing to NULL, then linked list is empty.
         Otherwise the linked list contains nodes.
         If(First == NULL)  {
             printf("List is empty");
         } else {
               printf("List is not empty");
         }
Step 3: End
```

Algorithm for Insert First :

```
Step 1: Start
Step 2: Allocate Dynamic memory for new node.
          tmp = (struct Node*) malloc (sizeof(struct Node));
step 3: Take input for new node.
           Assign user input to tmp->data;
step 4: Point the next of tmp to the first node.
           tmp -> next = First;
step 5: Check whether the linked list is empty or not.
          If linked list is empty then point FIRST and LAST pointer to    tmp.
           If (isempty()){
               First = Last = tmp;
           } else {
               goto 6;
           }
Step 6: If the list is not empty, Point first pointer to the new
         node (tmp).
         First = tmp;
Step 7: If you want to insert more element to first position then
         goto step 1;
Step 8: End.
```

Algorithm for Insert Last :

```
step 1: Start.

Step 2: Allocate Dynamic memory for new node.
        tmp = (struct Node*) malloc (sizeof(struct Node));

step 3: Take input for new node.
        Assign user input to tmp->data;

step 4: Set the next pointer of tmp to NULL cause this will be the
        last node.
        tmp -> next = NULL;

step 5: Check whether the linked list is empty or not.
        If linked list is empty then point First and Last pointer
        to tmp.
        If (isempty()){
            First = Last = tmp;
        } else {
            goto 6;
        }

Step 6: If the linked list is not empty, set nest pointer of Last
      to tmp.
      Last->next = tmp;
      Update the Last pointer to tmp.
      Last = tmp;

Step 7: If you want to insert more element to Last position
        then goto step 1;
```

Algorithm for Insert After :

```
Step 1: Start.

Step 2: Check whether the linked list is empty or not.
        If the linked list is empty,
  display "Linked List is empty. Insert After Operation can't
  be done".
        If (isempty()){
            printf("Linked List Is Empty…!\n");
            goto step 18;
        } else{
                goto Step 3.
        }

Step 3: Take input from the user for the index after which the data
        should be inserted.

Step 4: Set a counter cnt to 1 and point Curr to First.
        Curr = First;
        cnt = 1;

Step 5: Check if the next of Curr is not NULL and cnt is less than
  the index.
        If (Curr->next != NULL && cnt < index){
            goto Step 6.
        } else{
            goto Step 10.
        }
Step 6: Increment the cnt counter.
        cnt++;

Step 7: Move Curr to the next node.
        Curr = Curr->next;
```

```
Step 8: Go to Step 5.
        If (cnt == index)
            goto Step 11;
        Else
            "Index not found in Linked List"
            goto Step 3;

            Last = tmp;
        else
            goto Step 14;

Step 14: End.

Step 11: Take input from the user for the element to be inserted.
         Display the success message for the inserted element.

Step 12: Assign next of Curr to next of tmp.
         Assign tmp to Curr->next
         tmp->next = Curr->next;
         Curr->next = tmp;

Step 13: Check if Curr is pointing to Last.
        If (Curr == Last)
            Last = tmp;
        else
            goto Step 14;

Step 14: End.
```

Algorithm for Traverse :

```
Step 1: start

Step 2: Check if the linked list is empty or not If linked list
            is empty then display the “Linked List Empty”
        If (isempty()){
            printf(“Linked List Is Empty…!\n”);
        } else {
            goto step3;
        }

Step 3: Assign pointer Curr to Pointer First.
        Curr = first;

Step 4: Check if Curr is not Pointing to NULL then
            If(curr != NULL){
        goto step 5.
        } Else {
        goto step 8.
            }

Step 5: Display data of Curr(Curr->data)

Step 6: Point Curr to the next of Curr
        Curr = Curr->next;

Step 7:  Goto step 4

Step 8: End.
```

Algorithm for Delete First :

```
Step 1: start.

Step 2: Check whether the linked list is empty or not.
        If linked list is empty then
        display “Linked List is Empty ”
        If (isempty()){
            printf(“Linked List Is Empty…!\n”);
            goto step 9;
        } else {
            goto step3;
        }

Step 3: Update pointer tmp to pointer First.
        tmp = First;

Step 4: display the data that is going to be deleted.

Step 5: check if First and Last pointing to the same node then
  set First and Last pointer with NULL.
        if (First == Last) {
            First = Last = NULL;
        }else {
            goto step 6;
        }

Step 6: Update First to point to the next Node after First.
        First = First -> next;

Step 7: Free the Allocated memory.
        free(tmp);

Step 8: If you want to delete more element form first then goto step 1;

Step 9: end.
```

Algorithm for Delete Last :

```
Step 1: start.

Step 2: Check whether the lined list is empty or not.
        If linked list is empty then
        display “Linked List is Empty ”
        If (isempty()){
            printf(“Linked List Is Empty…!\n”);
            goto step 14;
        } else {
            goto step3;
        }

Step 3: update pointer tmp to pointer Last.
        tmp = Last;

Step 4: display the data that is going to be deleted.
       (tmp->data)

Step 5: check if First and Last pointing to the same node then set
        First and Last pointer with NULL.
        if (First == Last) {
            First = Fast = NULL;
            Goto st ep 12;
        } else {
            goto step 6;
        }

Step 6: update Curr to point to First.
        Curr = First;

Step 7: check if the next of Curr is not equal to last then
        If(curr->next != Last)
            Goto Step 8;
        Else
            Goto Step 10;
```

```
Step 8: Point Curr to the next Node of Curr
        Curr = Curr-> next;

Step 9: goto step 7.

Step 10: Assign NULL to the next of Curr
        Curr->next = NULL;

Step 11: Point Last to the Curr
        Last = Curr;

Step 12: Free the allocated Memory.
        free(tmp);

Step 13: If you want to delete more element form first then goto
         step 1;

Step 14:end
```

Algorithm for Delete Particular :

```
Step 1: start.

Step 2: Check whether the linked list is empty or not.
        If linked list is empty then
        display “Linked List is Empty ”
        If (isempty()){
            printf(“Linked List Is Empty…!\n”);
            goto step 18;
        } else {
            goto step 3;
        }

Step 3: Take input from user of index which node should be deleted;
Step 4: Point tmp pointer to First and set a counter cnt to 1 .
        tmp = First;
        cnt = 1;

Step 5: check if next of tmp is not equal to NULL and counter is
less then index
        if(tmp->next != NULL && cnt<index)
            goto step 6;
        else
            goto step 10;

Step 6: increment cnt counter
        cnt++;

Step 7: assign tmp to the Curr
        Curr = tmp;

Step 8: point tmp to the next Node of tmp
        tmp = tmp->next;

Step 9: goto step 5

Step 10: Check if cnt is equals to index
        If(index == cnt)
            Goto step 11;
        Else
            Display "Index is not available in Linked List"
        Goto step 18;

Step 11: display data which is going to be deleted
        (tmp->data)

step 12: check if Linked List having only one Node than  assign NULL to
the First and Last
        if(First == Last)
            First = Last = NULL;
            Goto step 16;
        Else
            Goto step 13;
```

```
Step 13: Check if user want to delete Data which is point by
   First pointer than Point First to the next Node of First
        If (tmp == First)
            First = First->next;
            Goto step 16;
        Else
            Goto step 14;

Step 14: Check if user want to delete Data which is point by
           Last pointer than assign NULL to the next of Curr and
   Point Last to the Curr
        If (tmp == Last)
            Curr->next = NULL;
            Last = Curr;
            Goto step 16;
        Else
            Goto step 15;

Step 15: assign next of tmp to the next of Curr
        Curr->next = tmp->next;

Step 16: Free the allocated Memory.
        free(tmp);

step 17: If you want to delete more element than goto step 1

step 18: end
```

Code of whole linked list :

Click on run button to run code: Run Code

✚ Code :

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

struct Node {
    int data;
    struct Node* next;
} *First = NULL , *Last = NULL , *Curr = NULL , *tmp = NULL ;

bool isempty();
void insertFirst();
void insertLast();
void insertAfter();
void traverse();
void deleteFirst();
void deleteLast();
void deletePerticuler();

int main() {
    int ch = -1;
    while(ch != 0) {
        printf("\n\n+--------------------------+");
        printf("\n|        Choices          |");
        printf("\n+-------------------------+");
        printf("\n| 1).Insert First         |");
        printf("\n| 2).Insert Last          |");
        printf("\n| 3).Insert after         |");
        printf("\n| 4).Delete First         |");
        printf("\n| 5).Delete Last          |");
        printf("\n| 6).Delete perticular    |");
        printf("\n| 7).Traverse             |");
        printf("\n| 0).Exit                 |");
        printf("\n+-------------------------+");
        printf("\n\nEnter your choice = ");
        scanf("%d",&ch);

        switch (ch) {
        case 1 :
            insertFirst() ;
            break ;
        case 2:
            insertLast();
            break;

        case 3:
            insertAfter();
            break;

        case 4:
            deleteFirst();
            break;

        case 5:
            deleteLast();
            break;

        case 6:
            deletePerticuler();
            break;

        case 7:
            traverse();
            break;

        case 0:
            printf("\nExiting........\n");
            break;

        default:
            printf("\nEnter valid choice ! ");
            break;
        }
    }
    return 0;
}

bool isempty() {
    return First == NULL;
}
```

```
void insertFirst() {
    tmp = (struct Node*) malloc(sizeof(struct Node));
    printf("Enter element You want to insert first = ");
    scanf("%d", &tmp->data);
    printf("\nElement %d inserted successfully at First Pos.", tmp-
>data);
    printf("\n==============================================");
    tmp->next = First;

    if(isempty()) {
        First = Last = tmp;
    } else {
        First = tmp;
    }
}

void insertLast() {
    tmp = (struct Node*) malloc(sizeof(struct Node));
    printf("Enter element You want to insert Last = ");
    scanf("%d", &tmp->data);
    printf("\nElement %d inserted successfully at Last Pos.", tmp-
>data);
    printf("\n==============================================");
    tmp->next = NULL;

    if(isempty()) {
        First = Last = tmp;
    } else {
        Last->next = tmp;
        Last = tmp;
    }
}
```

```c
void insertAfter() {
    if(isempty()) {
        printf("\n+----------------------------------------+");
        printf("\n| Linked List is empty                   |");
        printf("\n| Insert After Operation can't be done.  |");
        printf("\n+----------------------------------------+");
        return;
    }
    int index, cnt = 1;
    Curr = First;
    printf("\nAfter which index you want to insert data = ");
    scanf("%d", &index);
    while(Curr->next != NULL && cnt < index) {
        Curr = Curr->next;
        cnt++;
    }
    if(cnt == index) {
        tmp = (struct Node*)malloc(sizeof(struct Node));
        printf("Enter element you want to insert = ");
        scanf("%d", &tmp->data);
        printf("\nElement %d inserted successfully at %d Pos.", tmp-
>data, index+1);
        printf("\n===============================================");
        tmp->next = Curr->next;
        Curr->next = tmp;
        if(Curr == Last) {
            Last = tmp;
        }
    } else {
        printf("\n%d index is not found in Linked List.", index);
        printf("\n=======================================");
    }
}
```

```
void traverse() {
    if(isempty()) {
        printf("\n+------------------------+");
        printf("\n| Linked List is empty.   |");
        printf("\n+------------------------+");
        return;
    }
    Curr = First;
    printf("\nLinked List: ");
    while(Curr != NULL) {
        printf("| %d |-->", Curr->data);
        Curr = Curr->next;
    }
    printf(" /\n");
}

void deleteFirst() {
    if(isempty()) {
        printf("\n+-----------------------------------------+");
        printf("\n| Linked List is empty                    |");
        printf("\n| Delete First Operation can't be done.   |");
        printf("\n+-----------------------------------------+");
        return;
    }
    tmp = First;
    printf("\nData %d deleted from linked list", tmp->data);
    printf("\n=================================");
    if(First == Last) {
        First = Last = NULL;
    } else {
        First = First->next;
    }
    free(tmp);
}
```

```c
void deleteLast() {
    if(isempty()) {
        printf("\n+----------------------------------------+");
        printf("\n| Linked List is empty                   |");
        printf("\n| Delete Last Operation can't be done.   |");
        printf("\n+----------------------------------------+");
        return;
    }
    tmp = Last;
    printf("\nData %d deleted from linked list", tmp->data);
    printf("\n=================================");
    if(First == Last) {
        First = Last = NULL;
    } else {
        Curr = First;
        while(Curr->next != Last) {
            Curr = Curr->next;
        }
        Curr->next = NULL;
        Last = Curr;
    }
    free(tmp);
}

void deletePerticuler() {
    if(isempty()) {
        printf("\n+---------------------------------------------+");
        printf("\n| Linked List is empty                        |");
        printf("\n| Delete Perticuler Operation can't be done.  |");
        printf("\n+---------------------------------------------+");
        return;
    }
    int index, cnt = 1;
    printf("Enter which index you want to delete from Linked List = ");
    scanf("%d", &index);
    tmp = First;
    while(tmp->next != NULL && cnt < index) {
        cnt++;
        Curr = tmp;
        tmp = tmp->next;
        }
            if(cnt == index) {
            printf("\nElement %d deleted from Linked List . ",tmp->data);
            printf("\n======================================");
            if(First == Last) {
                First = Last = NULL ;
            } else if(tmp == First) {
                First = First->next ;
            } else if(tmp == Last) {
                Curr->next = NULL ;
                Last = Curr ;
            } else {
                Curr->next = tmp->next ;
            }
        } else {
            printf("\n%d index not found in Linked List . ",index);
            printf("\n====================================");
        }
        free(tmp);

    }
```

Code of real life example of linked list :

Click on this to run code LIVE : Run Code

```c
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
#include <string.h>

// Define the structure for train coach node
struct Node {
    int coachNumber;           // Unique coach number
    char coachType[20];        // Type of coach (AC, Sleeper, General)
    int capacity;              // Total number of seats in the coach
    int occupiedSeats;         // Currently occupied seats
    struct Node* next;         // Pointer to the next coach in the train
} *First = NULL, *Last = NULL, *Curr = NULL, *tmp = NULL;

// Function declarations
bool isempty();
bool isCoachNumberUnique(int number);
bool isValidCoachType(char *type);
void insertFirst();
void insertLast();
void traverse();
void deleteFirst();
void deleteLast();

int main() {
    int ch = -1;
    while (ch != 0) {
        printf("\n\n+------------------------+");
        printf("\n|        Choices         |");
        printf("\n+------------------------+");
        printf("\n| 1).Insert First        |");
        printf("\n| 2).Insert Last         |");
        printf("\n| 3).Delete First        |");
        printf("\n| 4).Delete Last         |");
        printf("\n| 5).Traverse            |");
        printf("\n| 0).Exit                |");
        printf("\n+------------------------+");
        printf("\n\nEnter your choice = ");
        scanf("%d", &ch);
        switch (ch) {
        case 1:
            insertFirst();
            break;
        case 2:
            insertLast();
            break;
        case 3:
            deleteFirst();
            break;
        case 4:
            deleteLast();
            break;
        case 5:
            traverse();
            break;
        case 0:
            printf("\nExiting........\n");
            break;
        default:
            printf("\nEnter valid choice!\n");
            break;
        }
    }
    return 0;
}

// Check if the list is empty
bool isempty() {
    return First == NULL;
}

// Check if the coach number is unique
bool isCoachNumberUnique(int number) {
    Curr = First;
    while (Curr != NULL) {
        if (Curr->coachNumber == number) {
            return false;  // Coach number is not unique
        }
        Curr = Curr->next;
    }
```

```c
    return true;  // Coach number is unique
}

// Check if the coach type is valid
bool isValidCoachType(char *type) {
    return (strcmp(type, "AC") == 0 || strcmp(type, "Sleeper") == 0 ||
strcmp(type, "General") == 0);
}

// Insert a coach at the beginning
void insertFirst() {
    // Allocate memory for the new node
    tmp = (struct Node*)malloc(sizeof(struct Node));

    // Take input for the new node's fields directly into tmp
    do {
        printf("Enter coach number: ");
        scanf("%d", &tmp->coachNumber);
        if (!isCoachNumberUnique(tmp->coachNumber)) {
            printf("\nError: Coach number must be unique!\n");
        }
    } while (!isCoachNumberUnique(tmp->coachNumber));

    do {
        printf("Enter coach type (AC, Sleeper, General): ");
        scanf("%s", tmp->coachType);
        if (!isValidCoachType(tmp->coachType)) {
            printf("\nError: Invalid coach type! Please enter 'AC',
'Sleeper', or 'General'.\n");
        }
    } while (!isValidCoachType(tmp->coachType));

    do {
        printf("Enter coach capacity (0-70): ");
        scanf("%d", &tmp->capacity);
        if (tmp->capacity < 0 || tmp->capacity > 70) {
            printf("\nError: Coach capacity must be between 0 and
70.\n");
        }
    } while (tmp->capacity < 0 || tmp->capacity > 70);
    do {
        printf("Enter number of occupied seats: ");
        scanf("%d", &tmp->occupiedSeats);
        if (tmp->occupiedSeats < 0 || tmp->occupiedSeats > tmp-
>capacity) {
            printf("\nError: Occupied seats must be non-negative and
less than or equal to the capacity.\n");
        }
    } while (tmp->occupiedSeats < 0 || tmp->occupiedSeats > tmp-
>capacity);

    // Insert the node at the beginning
    tmp->next = First;
    if (isempty()) {
        First = Last = tmp;
    } else {
        First = tmp;
    }

    printf("\nCoach %d inserted successfully at First Pos.\n", tmp-
>coachNumber);
    printf("===============================================");
}

// Insert a coach at the end
void insertLast() {
    // Allocate memory for the new node
    tmp = (struct Node*)malloc(sizeof(struct Node));

    // Take input for the new node's fields directly into tmp
    do {
        printf("Enter coach number: ");
        scanf("%d", &tmp->coachNumber);
        if (!isCoachNumberUnique(tmp->coachNumber)) {
            printf("\nError: Coach number must be unique!\n");
        }
    } while (!isCoachNumberUnique(tmp->coachNumber));
    do {
        printf("Enter coach capacity (0-70): ");
        scanf("%d", &tmp->capacity);
        if (tmp->capacity < 0 || tmp->capacity > 70) {
            printf("\nError: Coach capacity must be between 0 and
70.\n");
        }
    } while (tmp->capacity < 0 || tmp->capacity > 70);

    do {
        printf("Enter number of occupied seats: ");
        scanf("%d", &tmp->occupiedSeats);
        if (tmp->occupiedSeats < 0 || tmp->occupiedSeats > tmp-
>capacity) {
            printf("\nError: Occupied seats must be non-negative and
less than or equal to the capacity.\n");
        }
    } while (tmp->occupiedSeats < 0 || tmp->occupiedSeats > tmp-
>capacity);

    // Insert the node at the end
    tmp->next = NULL;
    if (isempty()) {
        First = Last = tmp;
    } else {
        Last->next = tmp;
        Last = tmp;
    }

    printf("\nCoach %d inserted successfully at Last Pos.\n", tmp-
>coachNumber);
    printf("===============================================");
}

// Traverse and display all coaches
void traverse() {
    if (isempty()) {
        printf("\nLinked List is empty.\n");
        return;
    }
    Curr = First;
    printf("\nLinked List: ");
    while (Curr != NULL) {
        printf("\n| Coach Number: %d, Type: %s, Capacity: %d, Occupied:
%d |",
            Curr->coachNumber, Curr->coachType, Curr->capacity, Curr-
>occupiedSeats);
        printf("\n\t\t\t||");
        printf("\n\t\t\t\\/");  // Visual representation of the linked
list traversal
        Curr = Curr->next;
    }
    printf("\n\t\t\tNULL\n"); // End of the linked list
}

// Delete the first coach
void deleteFirst() {
    if (isempty()) {
        printf("\nNo coaches to delete.\n");
        return;
    }

    tmp = First;
    printf("\nDeleting Coach Number %d...\n", First->coachNumber);
    if (First == Last) {  // If there's only one node
        First = Last = NULL;
    } else {
        First = First->next;
    }
    free(tmp);

    printf("First coach deleted successfully.\n");
    printf("===============================================");
}

// Delete the last coach
void deleteLast() {
    if (isempty()) {
        printf("\nNo coaches to delete.\n");
        return;
    }
    if (First == Last) {  // If there's only one node
        printf("\nDeleting Coach Number %d...\n", First->coachNumber);
        free(First);
        First = Last = NULL;
    } else {
        Curr = First;
        while (Curr->next != Last) {  // Traverse to the second last
node
            Curr = Curr->next;
        }
        printf("\nDeleting Coach Number %d...\n", Last->coachNumber);
        free(Last);
        Last = Curr;
        Last->next = NULL;
    }

    printf("Last coach deleted successfully.\n");
    printf("===============================================");
}
```

Source of Material:

Website:-

www.geeksforgeeks.org

Books:-

Data Structures Using C by Balagurusamy

"Data Structures Using C" by Reema Thareja(oxford university press

THANKS