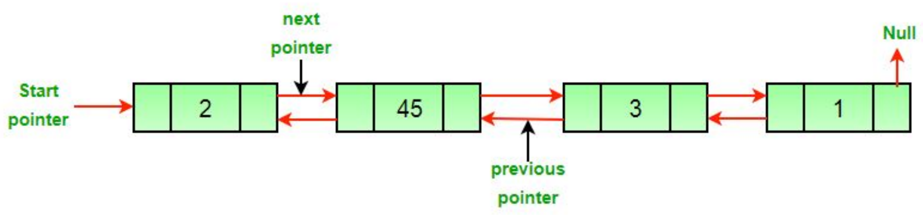
Delete Last of a Doubly Linked List

Pre-requisite: Doubly Link List Set 1| Introduction and Insertion

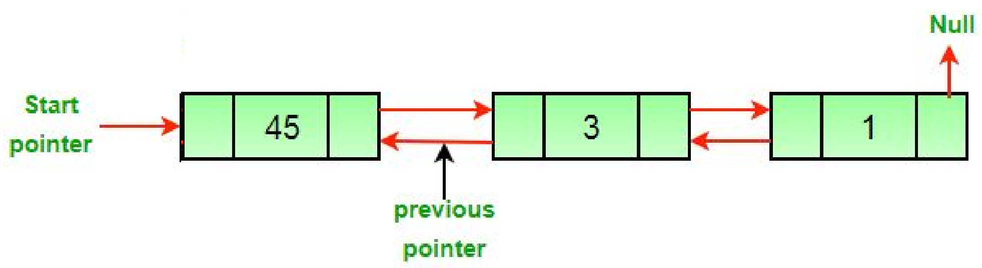
Write a function to delete a given node in a doubly-linked list.

Original Doubly Linked List

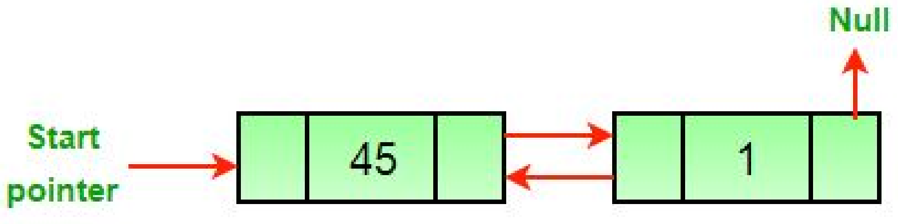


Approach: The deletion of a node in a doubly-linked list can be divided into three main categories:

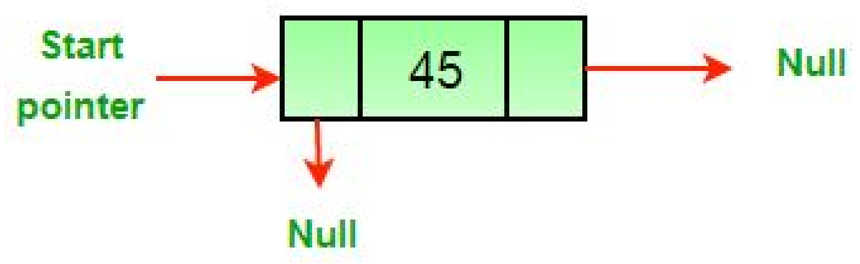
- **After the deletion of the head node.**



- **After the deletion of the middle node.**

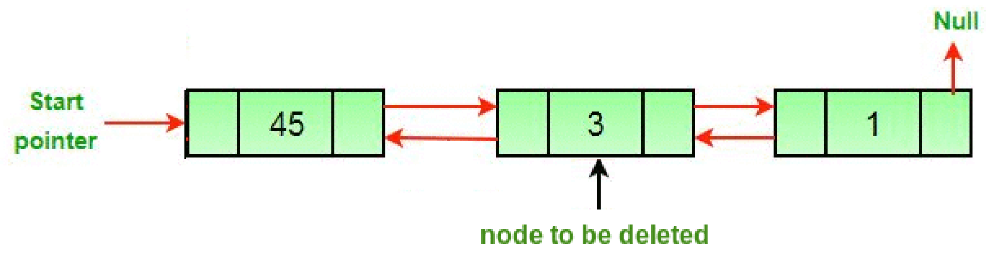


- **After the deletion of the last node.**



All three mentioned cases can be handled in two steps if the pointer of the node to be deleted and the head pointer is known.

1. If the node to be deleted is the head node then make the next node as head.
2. If a node is deleted, connect the next and previous node of the deleted node.



Algorithm

- Let the node to be deleted be *del*.
- If node to be deleted is head node, then change the head pointer to next current head.

```
if headnode == del then
headnode = del.nextNode
```

- Set prev of next to del, if next to del exists.

```
if del.nextNode != nonedel.nextNode.previousNode = del.prev
iousNode
```

- Set next of previous to del, if previous to del exists.

```
if del.previousNode !=nonedel.previousNode.nextNode =del.ne
xt
```

```
// C++ program to delete a node from
// Doubly Linked List
#include <bits/stdc++.h>
using namespace std;

/* a node of the doubly linked list */
class Node
{
    public:
    int data;
    Node* next;
    Node* prev;
};

/* Function to delete a node in a Doubly Linked List.
head_ref --> pointer to head node pointer.
del --> pointer to node to be deleted. */
void deleteNode(Node** head_ref, Node* del)
{
    /* base case */
    if (*head_ref == NULL || del == NULL)
        return;

    /* If node to be deleted is head node */
    if (*head_ref == del)
        *head_ref = del->next;

    /* Change next only if node to be
    deleted is NOT the last node */
    if (del->next != NULL)
        del->next->prev = del->prev;

    /* Change prev only if node to be
    deleted is NOT the first node */
```

```
    if (del->prev != NULL)
        del->prev->next = del->next;

    /* Finally, free the memory occupied by del*/
    free(del);
    return;
}

/* UTILITY FUNCTIONS */
/* Function to insert a node at the
beginning of the Doubly Linked List */
void push(Node** head_ref, int new_data)
{
    /* allocate node */
    Node* new_node = new Node();

    /* put in the data */
    new_node->data = new_data;

    /* since we are adding at the beginning,
    prev is always NULL */
    new_node->prev = NULL;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* change prev of head node to new node */
    if ((*head_ref) != NULL)
        (*head_ref)->prev = new_node;

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

/* Function to print nodes in a given doubly linked list
This function is same as printList() of singly linked list */
void printList(Node* node)
{
```

```
    while (node != NULL)
    {
        cout << node->data << " ";
        node = node->next;
    }
}

/* Driver code*/
int main()
{
    /* Start with the empty list */
    Node* head = NULL;

    /* Let us create the doubly linked list 10<->8<->4<->2 */
    push(&head, 2);
    push(&head, 4);
    push(&head, 8);
    push(&head, 10);

    cout << "Original Linked list ";
    printList(head);

    /* delete nodes from the doubly linked list */
    deleteNode(&head, head); /*delete first node*/
    deleteNode(&head, head->next); /*delete middle node*/
    deleteNode(&head, head->next); /*delete last node*/

    /* Modified linked list will be NULL<-8->NULL */
    cout << "\nModified Linked list ";
    printList(head);

    return 0;
}
```

Output:

```
Original Linked list 10 8 4 2
Modified Linked list 8
```

Complexity Analysis:

- **Time Complexity:** O(1). Since traversal of the linked list is not required so the time complexity is constant.
- **Space Complexity:** O(1). As no extra space is required, so the space complexity is constant.