Doubly Linked List

CLO: 01,02

Objectives:

- You will understand the structure of a doubly linked list.
- You will be able to write programs to insert, delete, and display nodes.
- You will learn circular linked lists and how they differ from linear lists.

Doubly Linked Lists (DLL) Review

A **linked list** is a chain of boxes (nodes). Each box stores some data (like a number) and a pointer that tells where the next box is.

A **doubly linked list (DLL)** is a special chain where every box has **two pointers**:

- **Next:** points to the next node (to the right).
- **Prev:** points to the previous node (to the left).

Because of **next** and **prev**, you can move **forward** and **backward** through the list easily.

Node: One box that holds data and two pointers (next, prev).

First (head): Pointer to the first node of the list.

Last (tail): Pointer to the last node of the list.

Empty list: When first == NULL (no nodes).

LinkedList: A Linked List contains the connection link to the first link called First and to the last link called Last.

Representation:

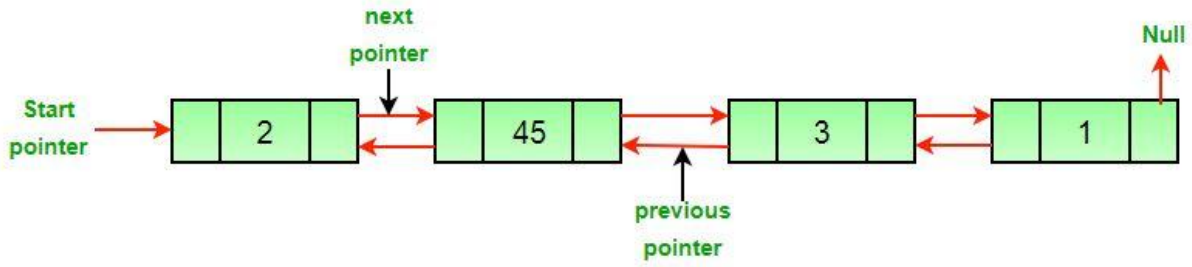


Figure 1. Doubly Linked List representation

Following are the basic operations supported by a list:

- **Insertion** − Adds an element at the beginning of the list.
- **Deletion** − Deletes an element at the beginning of the list.
- **Insert Last** − Adds an element at the end of the list.
- **Delete Last** − Deletes an element from the end of the list.
- **Insert After** − Adds an element after an item of the list.
- **Delete** − Deletes an element from the list using the key.
- **Display forward** − Displays the complete list in a forward manner.
- **Display backward** − Displays the complete list in a backward manner.

Following are advantages/disadvantages of doubly linked list over singly linked list

Advantages over singly linked list

Advantages

- You can go forward and backward easily.
- Deleting a node is faster if you already have a pointer to that node (you don't need to walk from the start to find the previous node).
- You can insert a node before a given node quickly.

Disadvantages

- Each node needs extra space for the **prev** pointer.
- You must update both next and **prev** pointers when inserting or deleting more work and more chances to make mistakes.

Creating a Doubly Linked List

We shall take the list to be initially empty, i.e.

L−>f i r s t = L−>last = NULL;

After allocating space for a new node and filling in the data (the value field), insertion of this node, pointed by p, is done as follows:

```
#include <iostream>
using namespace std;

class Node {
public:
   int data;
```

```
    Node* next;
    Node* prev;
    Node(int val) {
        data = val;
        next = NULL;
        prev = NULL;
    }
};
class DoublyLinkedList {
public:
    Node* head;

    DoublyLinkedList() {
        head = NULL;
    }
    // Insert at the beginning
    void insertAtStart(int val) {
        Node* newNode = new Node(val);

        if (head == NULL) {
            head = newNode;
        } else {
            newNode->next = head;
            head->prev = newNode;
            head = newNode;
        }
    }
    // Insert at the end
    void insertAtEnd(int val) {
        Node* newNode = new Node(val);
        if (head == NULL) {
```

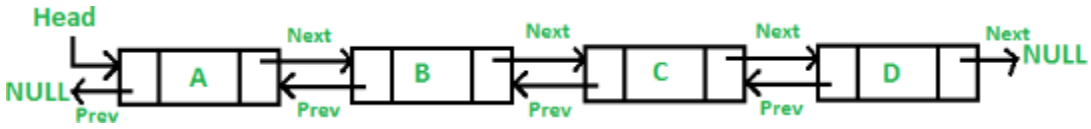
```
        head = newNode;
        return;
    }

    Node* temp = head;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = newNode;
    newNode->prev = temp;
}
// Delete a node (simple version)
void deleteNode(int val) {
    if (head == NULL) {
        cout << "List is empty.\n";
        return;
    }
    Node* temp = head;

    // Traverse to find node
    while (temp != NULL && temp->data != val) {
        temp = temp->next;
    }
    if (temp == NULL) {
        cout << "Node not found.\n";
        return;
    }
    // Adjust pointers
    if (temp->prev != NULL) {
        temp->prev->next = temp->next;
    } else {
```

```
            head = temp->next; // deleting first node
        }
        if (temp->next != NULL) {
            temp->next->prev = temp->prev;
        }

        delete temp;
        cout << "Node " << val << " deleted successfully.\n";
    }
    // Display list
    void display() {
        Node* temp = head;
        while (temp != NULL) {
            cout << temp->data << " <-> ";
            temp = temp->next;
        }
        cout << "NULL\n";
    }
};
int main() {
    DoublyLinkedList list;

    list.insertAtStart(10);
    list.insertAtEnd(20);
    list.insertAtEnd(30);
    list.display();

    list.deleteNode(20);
    list.display();
    return 0;
}
```

Practice Questions:

As you have learned how to insert and delete a node at different positions in the doubly linked list now you are required to perform the following tasks.

Create your own class of doubly linked list class, which will have the following functions:



1. Function called **Insert before given number** to add node at the beginning of list.

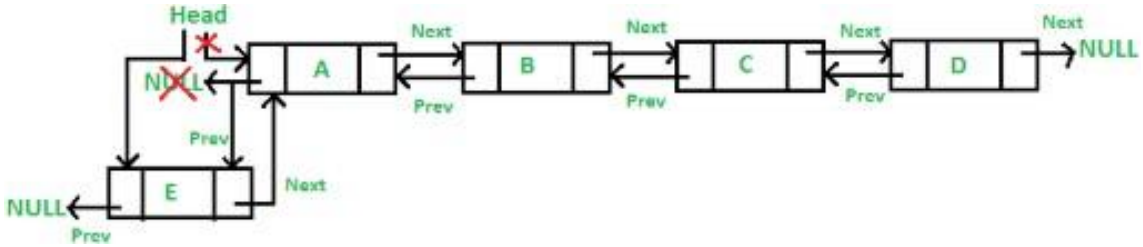
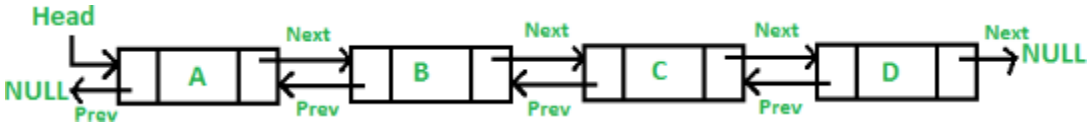


Figure 4. Insertion at beginning of the Doubly linked list



2. Function called **delete** to del node. Let say we want to delete node **C**. for this we have to copy Next pointer of node **C** in the Next pointer of node **B** and Previous pointer of node **C** in the Previous pointer of node **D**. Now our list will have only **A**, **B** and **D** nodes.
3. Function called **ReverseDisplay** to display values in reverse order.

Circular Linked List:

Circular linked list is a linked list where all nodes are connected to form a circle. There is no NULL at the end. A circular linked list can be a singly circular linked list or doubly circular linked list.

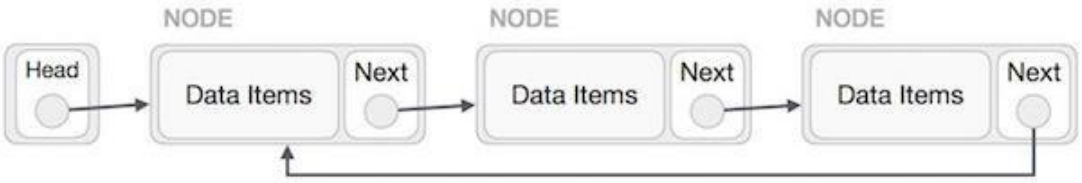


Figure 2. Singly List as Circular

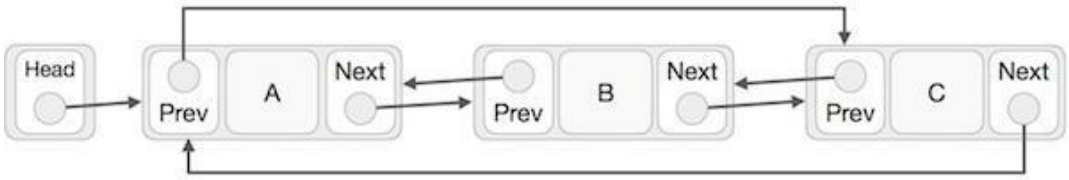


Figure 3. Doubly List as Circular

As per the above illustration, following are the important points to be considered.

- The last link's next points to the first link of the list in both cases of singly as well as doubly linked list.
- The first link's previous points to the last of the list in case of doubly linked list.

How to implement Circular Doubly Linked List:

```
#include <iostream>
using namespace std;

class Node {
public:
   int data;
   Node* next;
   Node* prev;

   Node(int val) {
      data = val;
```

```
        next = NULL;
        prev = NULL;
    }
};

class CircularDoublyLinkedList {
public:
    Node* head;

    CircularDoublyLinkedList() {
        head = NULL;
    }

    // Insert at the beginning
    void insertAtStart(int val) {
        Node* newNode = new Node(val);

        if (head == NULL) {
            head = newNode;
            head->next = head;
            head->prev = head;
        } else {
            Node* tail = head->prev;
            newNode->next = head;
            newNode->prev = tail;
            tail->next = newNode;
            head->prev = newNode;
            head = newNode;
        }
    }

    // Insert at the end
    void insertAtEnd(int val) {
        Node* newNode = new Node(val);
```

```
    if (head == NULL) {
        head = newNode;
        head->next = head;
        head->prev = head;
    } else {
        Node* tail = head->prev;
        tail->next = newNode;
        newNode->prev = tail;
        newNode->next = head;
        head->prev = newNode;
    }
}

// Delete a node (simple version)
void deleteNode(int val) {
    if (head == NULL) {
        cout << "List is empty.\n";
        return;
    }
    Node* temp = head;

    // Traverse to find the node
    do {
        if (temp->data == val) {
            // Adjust links to skip this node
            temp->prev->next = temp->next;
            temp->next->prev = temp->prev;

            // If the deleted node is head, move head forward
            if (temp == head) {
                head = head->next;
            }
```

```
            delete temp;
            cout << "Node " << val << " deleted successfully.\n";
            return;
        }
        temp = temp->next;
    } while (temp != head);
    cout << "Node not found.\n";
}

// Display all nodes
void display() {
    if (head == NULL) {
        cout << "List is empty.\n";
        return;
    }

    Node* temp = head;
    do {
        cout << temp->data << " <-> ";
        temp = temp->next;
    } while (temp != head);
    cout << "(back to head)\n";
}
};
// Main function for testing
int main() {
    CircularDoublyLinkedList list;

    list.insertAtStart(10);
    list.insertAtEnd(20);
    list.insertAtEnd(30);
    list.display();

    list.deleteNode(20);
```

```
    list.display();
    return 0;
}
```

Practice Questions

1. Write a function that inserts a new node **before a specific node** in a **Circular Doubly Linked List**. The function should take two parameters:

   key → the value before which we want to insert.

   val → the new value to insert.

Example:

```
If the list is: 10 <-> 20 <-> 30 <-> (back to head)
and you call: insertBeforeValue(20, 15);
then the updated list should be: 10 <-> 15 <-> 20 <-> 30 <-> (back to head)
```

Hint:

- Traverse the list until you find the node with data equal to key.
- Connect the new node between key->prev and key.
- Don't forget to update both next and prev links properly.
- Handle if the node to insert before is the **head** (then update head pointer).

2. **Function Name:** deleteByValue(int key)

Write a function that deletes a node having a specific value (`key`) from a **Circular Doubly Linked List**.

Example:

```
If the list is: 10 <-> 20 <-> 30 <-> 40 <-> (back to head)
and you call: deleteByValue(30);
then the updated list should be: 10 <-> 20 <-> 40 <-> (back to head)
```

Hint:

- Traverse the list until you find the node that has **data == key**.
- Adjust the previous node's next pointer and next node's **prev** pointer to skip the deleted node.
- If the deleted node is the head, move head to the next node.
- Be careful that the list stays circular (head's **prev** points to the last node and last's next points to head).