## **LEC-4 Doubly LinkList**

# **Doubly Linked List: Overview**

A Doubly Linked List is a fundamental data structure used in computer science and programming. It's an extension of the basic Linked List, providing additional features and flexibility. In a Doubly Linked List, each element, known as a node, contains data and two pointers, one pointing to the previous node and the other pointing to the next node.

# **Advantages of Doubly Linked List**

Doubly Linked Lists offer several advantages over other data structures:

•**Bidirectional Traversal:** The presence of both previous and next pointers enables efficient traversal in both directions.

•**Insertions and Deletions:** Insertions and deletions are efficient in a Doubly Linked List compared to an array, as there's no need to shift elements.

•**Reversal:** Reversing a Doubly Linked List is straightforward, as each node has a pointer to its previous node.

# **Node Structure**

Each node in a Doubly Linked List consists of three components:

•**Data:** The actual value or information that the node holds.

•**Previous Pointer:** A pointer that points to the previous node in the list. For the first node, this pointer is usually set to null.

•**Next Pointer:** A pointer that points to the next node in the list. For the last node, this pointer is typically set to null.

# **Doubly Linked List Operations**

**1. Insertion**

There are three main cases to consider when inserting a new node into a Doubly Linked List:

**•Insertion at the Beginning:** Adjust pointers of the new node, the current first node, and update the list's head pointer.

**•Insertion in the Middle:** Adjust pointers of the previous node, new node, and the next node.

**•Insertion at the End:** Adjust pointers of the current last node, the new node, and update the list's tail pointer.

**2. Deletion**

Similar to insertion, deletion involves handling three main cases:

**•Deletion from the Beginning:** Update pointers of the new first node and the removed node's next node.

**•Deletion from the Middle:** Update pointers of the previous node, next node, and deallocate the removed node.

•**Deletion from the End:** Update pointers of the new last node and deallocate the removed node.

**3. Traversal**

Traversal of a Doubly Linked List can be performed in both directions: forward and backward, utilizing the previous and next pointers.

**4. Reversal**

Reversing a Doubly Linked List involves swapping the previous and next pointers of each node. Care should be taken to update the head and tail pointers.

# **Use Cases**

Doubly Linked Lists find applications in various scenarios:

**•Undo/Redo Functionality:** Doubly Linked Lists can be used to implement undo and redo functionality in applications.

•**LRU Cache:** Least Recently Used (LRU) cache eviction policies can be implemented efficiently using Doubly Linked Lists.

**•Text Editors:** Doubly Linked Lists can be employed to manage the characters in a text editor, facilitating quick insertions and deletions.

# **Conclusion**

In summary, a Doubly Linked List is a versatile data structure that offers bidirectional traversal, efficient insertions and deletions, and easy reversal. Its node structure with data, previous pointers, and next pointers provides the foundation for its operations and applications in various domains.



using namespace std;

#include <iostream>

class doublylinklist; // Forward declaration

class Node {

    friend class doublylinklist;

    int data;

    Node\* next;

    Node\* prev;

};

class doublylinklist {

private:

    Node\* head;

public:

    doublylinklist() {

        head = nullptr;

    }

    ~doublylinklist() {

        // TODO: Implement destructor to free memory used by the list nodes

    }

    bool insertatstart(int val) {

        Node\* newNode = new Node();

        newNode->data = val;

        newNode->next = head;

        newNode->prev = nullptr;

        if (head != nullptr) {

            head->prev = newNode;

        }

        head = newNode;

        return true;

    }

    bool insertatend(int val) {

        Node\* newNode = new Node();

        newNode->data = val;

        newNode->next = nullptr;

        if (head == nullptr) {

            newNode->prev = nullptr;

            head = newNode;

            return true;

        }

        Node\* current = head;

        while (current->next != nullptr) {

            current = current->next;

        }

        current->next = newNode;

        newNode->prev = current;

        return true;

    }

    bool sortedinsert(int val) {

        // TODO: Implement sorted insert

        // Insert the given value in a sorted manner in the linked list

        return false;

    }

    bool deletefromstart() {

        if (head == nullptr) {

            return false;

        }

        Node\* temp = head;

        head = head->next;

        if (head != nullptr) {

            head->prev = nullptr;

        }

        delete temp;

        return true;

    }

    bool deletefromend() {

        if (head == nullptr) {

            return false;

        }

        if (head->next == nullptr) {

            delete head;

            head = nullptr;

            return true;

        }

        Node\* current = head;

        while (current->next != nullptr) {

            current = current->next;

        }

        current->prev->next = nullptr;

        delete current;

        return true;

    }

    Node\* search(int val) {

        Node\* current = head;

        while (current != nullptr) {

            if (current->data == val) {

                return current;

            }

            current = current->next;

        }

        return nullptr;

    }

    bool deletespecificvalue(int val) {

        Node\* nodeToDelete = search(val);

        if (nodeToDelete == nullptr) {

            return false;

        }

        if (nodeToDelete == head) {

            head = head->next;

            if (head != nullptr) {

                head->prev = nullptr;

            }

        } else {

            nodeToDelete->prev->next = nodeToDelete->next;

            if (nodeToDelete->next != nullptr) {

                nodeToDelete->next->prev = nodeToDelete->prev;

            }

        }

        delete nodeToDelete;

        return true;

    }

    void display() {

        Node\* current = head;

        while (current != nullptr) {

            cout << current->data << " ";

            current = current->next;

        }

        cout << endl;

    }

};

int main() {

    doublylinklist list;

    list.insertatstart(3);

    list.insertatstart(2);

    list.insertatstart(1);

    list.insertatend(4);

    list.insertatend(5);

    cout << "Original List: ";

    list.display();

    list.deletefromstart();

    list.deletefromend();

    list.deletespecificvalue(3);

    cout << "Modified List: ";

    list.display();

    return 0;

}