Data Structures Lab

Lab # 04

Singly Linked Lists

Lab Engineer: Hilal Ahmad

Department of Computer Science,

National University of Computer and Emerging Sciences FAST Peshawar Campus

Table of Contents

1. Introduction to Linked Lists

A **linked list** is a dynamic data structure consisting of nodes. Each node contains:

- **Data**: Stores the value of the node.
- **Pointer**: Points to the next node in the list.

In contrast to arrays:

- Linked lists can grow or shrink dynamically.
- Memory does not need to be contiguous.

In this lab, we will focus on implementing **singly linked lists** using **classes**.

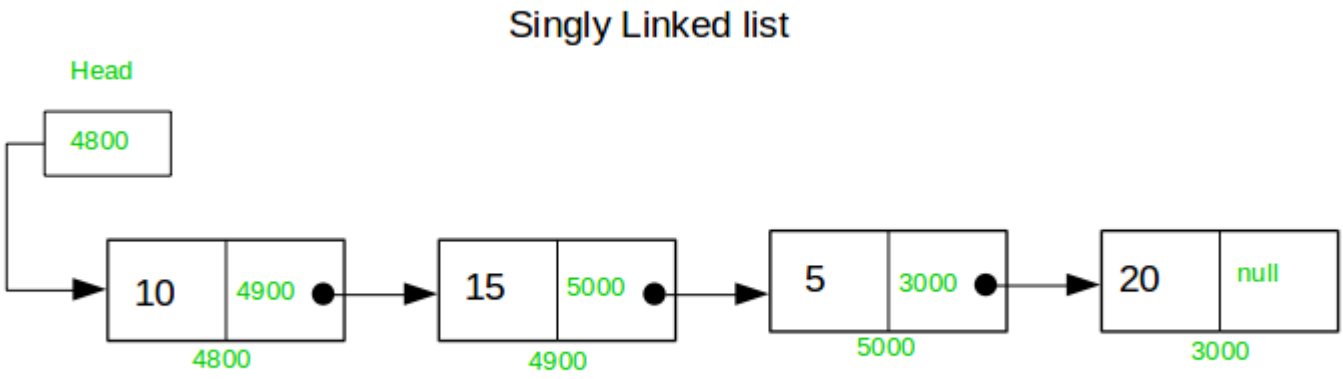

Fig.1: Structure of a Singly Linked List (Source: geeksforgeeks forum)

2. Singly Linked List Class Structure

Node Class:

```
class Node {
public:
    int data;      // Data part of the node
    Node* next;    // Pointer to the next node

    // Constructor
    Node(int value) : data(value), next(nullptr) {}
};
```

LinkedList Class:

The LinkedList class will handle all the operations such as insertion, deletion, and traversal of the list.

```
class LinkedList {
private:
    Node* head;  // Pointer to the first node (head)

public:
    // Constructor to initialize the list
    LinkedList() : head(nullptr) {}

    // Destructor to clean up memory
    ~LinkedList() {
        while (head) {
            deleteHead();
        }
    }

    // Function prototypes
    void insertAtHead(int value);
    void insertAtTail(int value);
    void deleteHead();
```

```
    void printList();
};
```

3. Operations on Singly Linked Lists

3.1 Insertion

We will implement insertion at two positions:

- **At the Head**: Insert a new node at the beginning.
- **At the Tail**: Insert a new node at the end.

Insertion at the Head:

```
void LinkedList::insertAtHead(int value) {
    Node* newNode = new Node(value);  // Create a new node
    newNode->next = head;             // Point new node to the current head
    head = newNode;                   // Update head to the new node
}
```

Insertion at the Tail:

```
void LinkedList::insertAtTail(int value) {
    Node* newNode = new Node(value);  // Create a new node

    if (head == nullptr) {
        head = newNode;  // If the list is empty, set newNode as head
    } else {
        Node* temp = head;
        while (temp->next != nullptr) {
            temp = temp->next;  // Traverse to the last node
        }
        temp->next = newNode;  // Attach newNode at the end
    }
}
```

3.2 Deletion

- **From the Head**: Remove the first node of the list.

Deletion from the Head:

```
void LinkedList::deleteHead() {
    if (head == nullptr) return;  // If the list is empty, return
    Node* temp = head;
    head = head->next;           // Move the head to the next node
    delete temp;                 // Free the memory of the old head
}
```

3.3 Traversal

To display the elements of the list, traverse from the head node to the end.

Traversal Example:

```
void LinkedList::printList() {
    Node* temp = head;
    while (temp != nullptr) {
        cout << temp->data << " ";  // Print data of the node
        temp = temp->next;          // Move to the next node
    }
    cout << endl;
}
```

4. Hands-On Coding Tasks

Task 1: Insertion at the Tail without Traversing Entire List

Optimize the **insertAtTail** function to store the tail node as part of the LinkedList class. Implement this version so that you don't have to traverse the list each time a new node is added at the end.

- **Note**: You can maintain a separate pointer to the tail node and update it during insertion.

Task 2: Insert at a Specific Position

Write a function to insert a node at a specific position in the list. The position can range from `1` (insert at head) to the total number of nodes + 1 (insert at tail). If the position is invalid, print an appropriate error message.

```
void insertAtPosition(int value, int position);
```

- **Note**: Consider boundary conditions (empty list, inserting at head/tail).

Task 3: Remove a Node by Value

Write a function that deletes a node from the linked list based on the value it contains. If the value is not present, display an appropriate message. You should handle deletion of the head, tail, and middle nodes.

```
void deleteByValue(int value);
```

- **Note**: Make sure to handle edge cases like deleting the only node or when the value does not exist in the list.

Task 4: Merge Two Sorted Linked Lists

Write a function to merge two sorted linked lists into one sorted list. Assume the lists are sorted in non-decreasing order.

```
LinkedList mergeSortedLists(LinkedList& other);
```