# CS224 Object Oriented Programming and Design Methodologies Lab Manual

# Lab 8- Linked List

Department of Computer Science

Dhanani School of Science and Engineering

Habib University

Fall 2025

# Contents

# Lab 8

# Linked List

## 8.1 Guidelines

1. Use of AI is strictly prohibited. This is not limited to the use of AI tools for code generation, debugging, or any form of assistance. If detected, it will result in immediate failure of the lab, student will be awarded 0 marks, reported to the academic integrity board and appropriate disciplinary action will be taken.
2. Absence in lab regardless of the submission status will result in 0 marks.
3. All assignments and lab work must be submitted by the specified deadline on Canvas. Late submissions will not be accepted.

## 8.2 Objectives

Following are the lab objectives of this lab:

1. To practice using pointers to create a Queue data structure in C++.
2. To understand how linked lists can be used to implement dynamic data structures.
3. To implement the FIFO (First-In-First-Out) principle using classes in C++.
4. To perform enqueue and dequeue operations using a linked list.
5. To handle edge cases in Queue operations such as empty queues.

## 8.3 Directory Structure

Labs will have following directory structure:

```
lab
├── manual
│   └── lab.pdf
├── src
│   ├── *.hpp
│   └── *.cpp
├── tests (optional)
│   └── *.cpp
└── makefile (optional)
```

`manual` will contain the lab manual pdf. `src` will contain the source code files, and `tests` (if present) will contain the test files. `makefile` (if present) will contain the makefile for testing and running.

## 8.4 Linked List in C++

A **Linked List** is a linear data structure that allows dynamic storage of data in non-contiguous memory locations. Unlike arrays, which require contiguous memory blocks, a linked list consists of a collection of nodes that are connected together using pointers. Each node typically contains two components:

- **Data:** The actual information to be stored.
- **Pointer:** A reference (or link) to the next node in the sequence.

In C++, a linked list can be efficiently implemented using **classes**. A common approach is to define a `Node` class that stores the data and a pointer to the next node, and a `LinkedList` class that manages the overall structure by maintaining a reference to the `head` node.

### Example: Implementation of a Simple Linked List in C++

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

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

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

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

public:
    LinkedList() : head(nullptr) {} // Constructor

    // Add node to the beginning
    void addToBeginning(int value) {
        Node* newNode = new Node(value);
        newNode->next = head;
        head = newNode;
    }

    // Add node to the end
    void addToEnd(int value) {
        Node* newNode = new Node(value);
        if (!head) {
            head = newNode;
        } else {
            Node* temp = head;
            while (temp->next != nullptr) {
                temp = temp->next;
            }
            temp->next = newNode;
        }
    }

    // Delete node from the beginning
    void deleteFromBeginning() {
        if (!head) {
            cout << "List is empty, nothing to delete.\n";
            return;
        }
        Node* temp = head;
        head = head->next;
        delete temp;
    }

    // Delete node from the end
    void deleteFromEnd() {
        if (!head) {
            cout << "List is empty, nothing to delete.\n";
            return;
        }
        if (!head->next) { // Only one element
            delete head;
            head = nullptr;
        } else {
            Node* temp = head;
            while (temp->next->next != nullptr) {
                temp = temp->next;
            }
            delete temp->next;
            temp->next = nullptr;
        }
    }

    // Display the linked list
    void display() {
        if (!head) {
            cout << "List is empty.\n";
            return;
        }
        Node* temp = head;
        while (temp != nullptr) {
            cout << temp->data << " -> ";
            temp = temp->next;
        }
        cout << "null\n";
    }
};

// Main function for testing
int main() {
    LinkedList list;

    // Test adding to the beginning
    cout << "Adding 10 and 20 to the beginning of the list:\n";
    list.addToBeginning(10);
    list.addToBeginning(20);
    list.display();

    // Test adding to the end
    cout << "\nAdding 30 and 40 to the end of the list:\n";
    list.addToEnd(30);
    list.addToEnd(40);
    list.display();

    // Test deleting from the beginning
    cout << "\nDeleting from the beginning:\n";
    list.deleteFromBeginning();
    list.display();

    // Test deleting from the end
    cout << "\nDeleting from the end:\n";
    list.deleteFromEnd();
    list.display();

    return 0;
}
```

Listing 8.1: Example: Basic Linked List Implementation

### Applications

Linked lists are widely used in:

- Implementing dynamic data structures such as stacks and queues.
- Memory management systems to keep track of free and allocated memory.
- Graph representation using adjacency lists.
- Undo/redo functionality in text editors.

## 8.5 Exercises

### Question 1 — Linked List Manipulations

Write a C++ program `linked_list.cpp` that performs the following operations on a singly linked list:

- Define a `Node` class with integer `data` and a pointer to the next node.
- Define a `LinkedList` class with:
  - `print()` — prints all elements of the list by traversing it.
  - `length()` — calculates and returns the length of the linked list.
  - `insertBefore(int target, int newValue)` — inserts a new node **before** a given element.
  - `insertAfter(int target, int newValue)` — inserts a new node **after** a given element.
  - `deleteBefore(int target)` — deletes the node **before** a given element.
  - `deleteAfter(int target)` — deletes the node **after** a given element.
- Demonstrate all operations through a `main()` function.

#### Sample Test Case

```
Initial list: 10 -> 20 -> 30 -> 40 -> null
Length of list: 4

Insert 15 before 20:
10 -> 15 -> 20 -> 30 -> 40 -> null

Insert 35 after 30:
10 -> 15 -> 20 -> 30 -> 35 -> 40 -> null

Delete node before 20:
10 -> 20 -> 30 -> 35 -> 40 -> null

Delete node after 30:
10 -> 20 -> 30 -> 40 -> null
```

—

## Question 2 — Queue Implementation using Linked List

Implement a Queue using a Linked List in a C++ program named `linked_queue.cpp`.

A Queue follows the First-In-First-Out (FIFO) principle:

- `enqueue()` inserts an element at the rear.
- `dequeue()` removes an element from the front.
- Define a class `Node` with a data variable and a pointer to the next node.
- Define a class `Queue` with private pointers `front` and `rear`.
- Implement member functions:
  - `enqueue(int)` — add elements at the rear of the queue.
  - `dequeue()` — remove elements from the front of the queue.
  - `print_queue()` — display the elements of the queue.
- Prompt the user for the number of elements to enqueue and the elements themselves.
- Print the queue after enqueue operations.
- Prompt the user for the number of elements to dequeue and perform the operations.
- Print the queue after dequeue operations.
- Handle edge cases such as attempting to dequeue from an empty queue.

### Sample Test Case

```
>>> Enter number of elements to enqueue: 5
>>> Enter elements: 1 2 3 4 5
Queue after enqueue: 1 2 3 4 5
>>> Enter number of elements to dequeue: 3
Queue after dequeue: 4 5
```

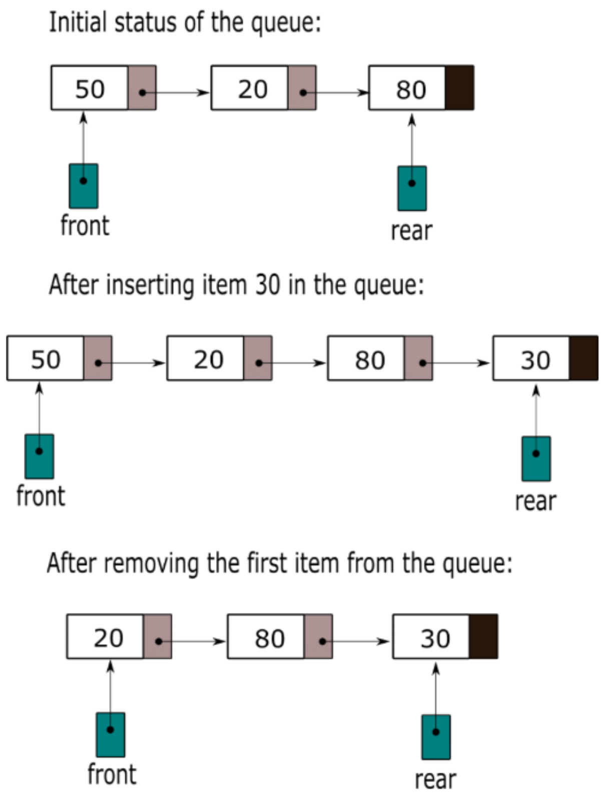
Initial status of the queue:
50
20
80
front
rear
After inserting item 30 in the queue:
50
20
80
30
front
rear
After removing the first item from the queue:
20
80
30
front
rear

Figure 8.1: Enqueue and Dequeue operations in a Linked List Queue

## Sample Test Case 2

```
>>> Enter number of elements to enqueue: 4
>>> Enter elements: 1 2 2 3
Queue after enqueue: 1 2 2 3
>>> Enter number of elements to dequeue: 2
Queue after dequeue: 2 3
```