Data Structure and Algorithms (JAVA)



**3rd Lab**

**Semester: Spring 2025**

**Software Engineering**

**Faculty of Information Technology, UCP Lahore, Pakistan**

Table of Contents

[Session 1: Common Operations and Complexity Analysis 4](#_Toc193516293)

[1.1 Insert Operation in Singly Linked List 4](#_Toc193516294)

[1.2 Delete Operation in Singly Linked List 5](#_Toc193516295)

[Session 2: Linked List Operations 6](#_Toc193516296)

[2.1 Traversing a Linked List 6](#_Toc193516297)

[2.2 Searching for an element in Linked List 7](#_Toc193516298)

[Session 3: Complexity Analysis and Practical Example 8](#_Toc193516299)

[3.1 Student Name Manager using Singly Linked List 8](#_Toc193516300)

[3.2 Complexity Analysis 9](#_Toc193516301)

[Basic Practice Task 10](#_Toc193516302)

[Scenario-Based Case Studies 10](#_Toc193516303)

[Case Study 1: Online Course Enrollment System 10](#_Toc193516304)

[Case Study 2: Music Streaming App (Playlist Management) 10](#_Toc193516305)

[Conclusion 10](#_Toc193516306)

**Lab Manual: Data Structures and Algorithms using Java**

Objective

In this lab, students will gain hands-on experience in implementing and performing operations on singly linked lists, understanding how they differ from array lists, and analyzing their time complexities.

By the end of this lab, students will be able to:

* **Understand** the fundamentals of linked lists and distinguish between linked lists and array lists.
* **Implement** a singly linked list and perform basic operations such as insertion, deletion, traversal, and searching.
* **Analyze** the time complexity of various singly linked list operations and understand the trade-offs compared to array-based implementations.
* **Apply** linked list concepts to real-world applications, such as managing a music playlist with forward and backward navigation.

This lab serves as a foundation for exploring more advanced linked list variations (e.g., doubly and circular linked lists) and their use in efficient data structure implementations.

## **Session 1: Common Operations and Complexity Analysis**

### **1.1 Insert Operation in Singly Linked List**

#### **Code Example: Inserting an Element in a Singly Linked List (at the end, a specific position, or beginning):**

import java.util.LinkedList;

public class LinkedListInsertion {

public static void main(String[] args) {

LinkedList<Integer> list = new LinkedList<>();

// Insert elements at the end (append)

list.add(10);

list.add(20);

list.add(30);

// Insert element at the beginning

list.addFirst(5);

// Insert element at a specific index

list.add(2, 15);

System.out.println("Linked List after insertions: " + list);

}

}

**Time Complexity:**

**O(1)** – Appending at the end of the list.

**O(1)** – Inserting at the head of the list.

**O(n)** – Insert at a specific index because you may need to traverse the list to find the position.

### **1.2 Delete Operation in Singly Linked List**

#### **Code Example: Deleting an Element from a Singly Linked List**

import java.util.LinkedList;

public class LinkedListDeletion {

public static void main(String[] args) {

LinkedList<String> list = new LinkedList<>();

list.add("A");

list.add("B");

list.add("C");

list.add("D");

// Remove element from the beginning

list.removeFirst();

// Remove element from the end

list.removeLast();

// Remove element at a specific index

list.remove(1);

System.out.println("Linked List after deletions: " + list);

}

}

**Time Complexity:**

removeFirst(): **O(1)** – Removing the head node.

removeLast(): **O(1)** – (Java’s built-in LinkedList optimizes this operation).

remove(index): **O(n)** – Need to traverse the list to locate the element at the given index.

## **Session 2: Linked List Operations**

### **2.1 Traversing a Linked List**

import java.util.LinkedList;

public class LinkedListTraversal {

public static void main(String[] args) {

LinkedList<Integer> list = new LinkedList<>();

list.add(1);

list.add(2);

list.add(3);

list.add(4);

System.out.println("Traversing using for-each:");

for (int num : list) {

System.out.println(num);

}

System.out.println("\nTraversing using basic for loop:");

for (int i = 0; i < list.size(); i++) {

System.out.println(list.get(i));

}

}

}

**Time Complexity:**

**Using for-each loop:**  
**O(n)** – Visit each node sequentially.

**Using basic for loop and get(index) method:**  
**O(n²)** – Each get() call takes O(n) due to sequential access, leading to nested traversal if you use get() inside a loop.

### **2.2 Searching for an element in Linked List**

import java.util.LinkedList;

public class LinkedListSearching {

public static void main(String[] args) {

LinkedList<String> list = new LinkedList<>();

list.add("Apple");

list.add("Banana");

list.add("Cherry");

list.add("Date");

// Searching using contains() method

String searchItem = "Cherry";

if (list.contains(searchItem)) {

System.out.println(searchItem + " is found in the list.");

} else {

System.out.println(searchItem + " is not found in the list.");

}

}

}

**Time Complexity:**

Search for an element **(contains(element))**:

**O(n)** – The list may need to be traversed to find the element.

## **Session 3: Complexity Analysis and Practical Example**

### **3.1 Student Name Manager using Singly Linked List**

#### **Code Example:**

import java.util.LinkedList;

public class StudentManager {

public static void main(String[] args) {

LinkedList<String> studentList = new LinkedList<>(); // Singly Linked List

// Adding students to the list

studentList.add("Ali");

studentList.add("Babar");

studentList.add("Sana");

studentList.add("Daud");

System.out.println("Initial Student List:");

displayList(studentList);

// Remove a student

System.out.println("\nRemoving 'Charlie' from the list...");

studentList.remove("Charlie");

displayList(studentList);

// Searching for a student

String searchName = "Ali";

if (studentList.contains(searchName)) {

System.out.println("\n" + searchName + " is found in the list!");

} else {

System.out.println("\n" + searchName + " is not in the list.");

}

// Adding a new student at a specific index

studentList.add(2, "Eve");

System.out.println("\nAfter adding 'Eve' at index 2:");

displayList(studentList);

}

// Method to display the student list

public static void displayList(LinkedList<String> list) {

for (String student : list) {

System.out.println(student);

}

}

}

### **3.2 Complexity Analysis**

|  |  |  |
| --- | --- | --- |
| **Operation** | **Method** | **Time Complexity** |
| Add at the end | add(element) | O(1) |
| Add at a specific index | add(index, element) | O(n) |
| Remove an element | remove(element) | O(n) |
| Check if list contains element | contains(element) | O(n) |
| Traverse the list | for-each loop | O(n) |

## **Basic Practice Task**

1. **Insert an element in a sorted linked list while maintaining the sorted order**

* Objective: Implement a method to insert an element in the correct position of a sorted singly linked list.
* Skills Focused: Traversal, insertion, maintaining list order.
* Challenge: Ensure the list remains sorted after the insertion.

1. **Reverse a singly linked list**

* Objective: Implement a method to reverse the given singly linked list without using extra space.
* Skills Focused: Pointer manipulation, understanding of next references.
* Challenge: Handle the list reversal iteratively (or recursively if they are more advanced).

## **Scenario-Based Case Studies**

### **Case Study 1: Online Course Enrollment System**

**Scenario:**

An online learning platform stores a list of students enrolled in a course using a single linked list.

**Tasks:**

1. Add a new student at the end of the list when they enroll.
2. Remove a student from the list if they drop the course.
3. Search for a student by name to check if they are enrolled.
4. Sort the list alphabetically by student names.
5. Find the student with the highest grade and return their details.

## **Case Study 2: Music Streaming App (Playlist Management)**

**Scenario:**

A music streaming app stores songs in a singly linked list. Users can perform actions like adding, deleting, or finding songs in their playlists. **Tasks:**

1. Add a new song at the end of the playlist.
2. Delete a song from the playlist based on its title.
3. Search for a song in the playlist to check if it is already added.
4. Sort the playlist alphabetically by song titles.
5. Find the longest song (in duration) and return its title.

### **Conclusion**

In this lab, students explored the core concepts and operations of singly linked lists, including insertion, deletion, traversal, and searching. Additionally, students analyzed the time complexities of linked list operations and understood real-world applications such as Student Name Manager, etc.