

Lab Experiment No. 1

Aim:

Implementation and analysis of Arrays and its operations.

- Insertion
- Deletion
- Searching
- Traversing
- Updating

Background:

An array is a fundamental data structure that stores elements of the same data type in contiguous memory locations. Arrays provide constant-time access to elements using indexing but have a fixed size, meaning their size must be declared at the time of creation and cannot be dynamically changed.

Operations on Arrays

Arrays support various operations, including insertion, deletion, searching, traversing, and updation. Each operation has a specific behavior based on its implementation.

1. Insertion

Insertion involves adding a new element to the array at a specified index. If the insertion is at the end, it is efficient. However, inserting an element at the beginning or middle requires shifting elements, making the process more complex.

2. Deletion

Deletion removes an element from the array. If the element is at the end, the operation is straightforward. However, if it is at the beginning or middle, all subsequent elements need to be shifted to maintain the order.

3. Searching

Searching is used to find an element in an array. Two common algorithms are:

- Linear Search: Used for unsorted arrays. It checks each element sequentially.
- Binary Search: Used for sorted arrays. It follows the divide-and-conquer approach.

4. Traversing

Traversal means visiting each element in the array sequentially. This is required for processing or printing the elements.

5. Updation

Updating an element at a specific index is a direct operation because indexing provides constant-time access.

CODE

```
import java.util.Arrays;

public class ArrayOperations {
    private int[] arr;
    private int size;

    public ArrayOperations(int capacity) {
        arr = new int[capacity];
        size = 0;
    }
```

Insertion:

```
public void insert(int index, int value) {
    if (size == arr.length) {
        System.out.println("Array is full. Cannot insert.");
        return;
    }
    if (index < 0 || index > size) {
        System.out.println("Invalid index.");
        return;
    }
    for (int i = size - 1; i >= index; i--) {
        arr[i + 1] = arr[i];
    }
    arr[index] = value;
    size++;
}
```

Deletion

```
public void delete(int index) {
    if (index < 0 || index >= size) {
        System.out.println("Invalid index.");
        return;
    }
    for (int i = index; i < size - 1; i++) {
        arr[i] = arr[i + 1];
    }
    size--;
}
```

// Searching (Linear Search)

```
public int linearSearch(int value) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == value) {
            return i;
        }
    }
    return -1;
}
```

// Searching (Binary Search - Assumes Sorted Array)

```
public int binarySearch(int value) {
    int left = 0, right = size - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == value)
            return mid;
        else if (arr[mid] < value)
            left = mid + 1;
        else
            right = mid - 1;
    }
    return -1;
}
```

// Traversing

```
public void traverse() {
    for (int i = 0; i < size; i++) {
        System.out.print(arr[i] + " ");
    }
    System.out.println();
}
```

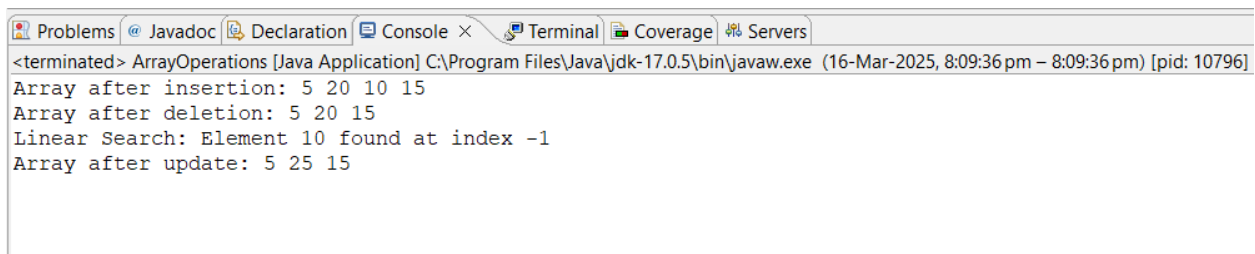
// Updating

```
public void update(int index, int value) {
    if (index < 0 || index >= size) {
        System.out.println("Invalid index.");
        return;
    }
    arr[index] = value;
}
```

TEST

```
public static void main(String[] args) {  
    ArrayOperations arrayOps = new ArrayOperations(10);  
    arrayOps.insert(0, 5);  
    arrayOps.insert(1, 10);  
    arrayOps.insert(2, 15);  
    arrayOps.insert(1, 20); // Insert at index 1  
  
    System.out.print("Array after insertion: ");  
    arrayOps.traverse();  
  
    arrayOps.delete(2);  
    System.out.print("Array after deletion: ");  
    arrayOps.traverse();  
  
    int searchIndex = arrayOps.linearSearch(10);  
    System.out.println("Linear Search: Element 10 found at index " + searchIndex);  
  
    arrayOps.update(1, 25);  
    System.out.print("Array after update: ");  
    arrayOps.traverse();  
}  
}
```

Output



The screenshot shows a Java IDE window with the 'Terminal' tab selected. The output of the program is as follows:

```
<terminated> ArrayOperations [Java Application] C:\Program Files\Java\jdk-17.0.5\bin\javaw.exe (16-Mar-2025, 8:09:36 pm – 8:09:36 pm) [pid: 10796]  
Array after insertion: 5 20 10 15  
Array after deletion: 5 20 15  
Linear Search: Element 10 found at index -1  
Array after update: 5 25 15
```

Fig 1.1 Arrays and its operations output

Performance Analysis:

OPERATION	BEST CASE	WORST CASE	AVERAGE CASE
Insertion	$O(1)$	$O(n)$	$O(n)$
Deletion	$O(1)$	$O(n)$	$O(n)$
Searching	$O(1)$	$O(n)$	$O(n)$
Traversing	$O(n)$	$O(n)$	$O(n)$
Updation	$O(1)$	$O(1)$	$O(1)$

Table1.1 Arrays and its operations. Performance Analysis

Conclusion:

- The experiment confirms that array operations vary in efficiency depending on the operation type and position of elements.
- Insertion and Deletion at the beginning or middle are costly due to shifting, reinforcing why linked lists are preferred for frequent insertions and deletions.
- Searching in an unsorted array is linear, while sorting the array beforehand allows efficient binary search.
- Updation is the most efficient operation since modifying an index is direct.
- The limitations of static arrays (fixed size, costly insertions/deletions) suggest the necessity of dynamic arrays or alternative data structures like linked lists, hash tables, or trees in practical applications.

Lab Experiment No. 2

Aim :

- Implementation and analysis of Linked Lists and their operations.
 - Singly Linked List
 - Insertion a new node
 - Deletion of a node
 - Traversing
 - Doubly Linked List
 - Insertion a new node
 - Deletion of a node
 - Traversing
 - Circular Linked List
 - Insertion a new node
 - Deletion of a node
 - Traversing

Background:

1. Singly Linked List

Definition

A **Singly Linked List (SLL)** is a linear data structure where each node contains data and a pointer to the next node in the sequence. The last node's pointer is set to null, indicating the end of the list.

Characteristics

- Each node has **two parts**:
 1. **Data**: The actual value stored in the node.
 2. **Next Pointer**: A reference to the next node in the list.
- The list starts from a **head node**.
- It **does not support backward traversal**.

Operations

1. **Insertion**: A new node can be inserted at the beginning, middle, or end of the list.
2. **Deletion**: A node can be removed by adjusting the next pointers of the previous node.
3. **Traversing**: The list is traversed from the head node to the last node using the next pointer.

Advantages

Uses memory efficiently as each node contains only one pointer.

Dynamic memory allocation prevents memory wastage.

Disadvantages

Searching is slow ($O(n)$) since we must traverse the list sequentially.

Cannot traverse backward since there is no reference to the previous node.

2. Doubly Linked List

Definition

A **Doubly Linked List (DLL)** is an extension of the singly linked list where each node contains two pointers:

1. **Next Pointer:** Points to the next node.
2. **Previous Pointer:** Points to the previous node.

Characteristics

- Each node has **three parts**:
 1. **Data**
 2. **Pointer to the Next Node**
 3. **Pointer to the Previous Node**
- The list starts with a **head** and ends with a **tail**.
- It allows **both forward and backward traversal**.

Operations

1. **Insertion:** A node can be added at the beginning, middle, or end, adjusting both next and prev pointers.
2. **Deletion:** A node can be removed by updating both pointers in neighboring nodes.
3. **Traversing:** The list can be traversed in both forward and backward directions.

Advantages

Faster traversal as it supports **both directions**.

More efficient deletion since we have direct access to the previous node.

Disadvantages

Requires **extra memory** due to the additional pointer in each node.

Insertion and deletion involve updating two pointers, making it slightly more complex.

3. Circular Linked List

Definition

A **Circular Linked List (CLL)** is a variation of the singly or doubly linked list where the last node points back to the first node instead of null, forming a **circular structure**.

Types of Circular Linked Lists

1. **Singly Circular Linked List:** The last node's next pointer connects to the head node.
2. **Doubly Circular Linked List:** Both next and prev pointers form a circular connection.

Characteristics

- There is **no null at the end**; the last node links back to the first.
- Can be implemented as **singly or doubly linked**.
- Traversal can start from **any node** and continue indefinitely.

Operations

1. **Insertion:** A new node can be added at any position while maintaining the circular structure.
2. **Deletion:** A node can be removed, adjusting pointers to maintain continuity.
3. **Traversing:** Since there is no null, traversal stops when we reach the starting node again.

Advantages

Efficient for **buffered or cyclic operations** (e.g., scheduling algorithms).

Can be traversed infinitely without needing to restart.

Disadvantages

More complex than singly and doubly linked lists.

Risk of **infinite loops** if not handled properly.

Comparison Table

Feature	Singly Linked List	Doubly Linked List	Circular Linked List
Memory Usage	Low (1 pointer per node)	Medium (2 pointers per node)	Medium (depends on type)
Traversal Direction	Forward Only	Forward & Backward	Forward (or both if doubly)
Insertion Complexity	O(1) (at head), O(n) (elsewhere)	O(1) (at head), O(n) (elsewhere)	O(1) (at head), O(n) (elsewhere)
Deletion Complexity	O(1) at head, O(n) elsewhere	O(1) if node reference is given	O(1) at head, O(n) elsewhere
Circular Structure	No	No	Yes
Best Use Cases	Simple, linear data storage	Doubly connected data traversal	Cyclic tasks, scheduling

Table 2.1 Comparison Table Linked Lists and their operations

CODE :

/ Singly Linked List Implementation

```

class SinglyLinkedList {
    class Node {
        int data;
        Node next;
        Node(int data) {
            this.data = data;
            this.next = null;
        }
    }

    private Node head;

    public void insert(int data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
            return;
        }
        Node temp = head;
        while (temp.next != null) {
            temp = temp.next;
        }
        temp.next = newNode;
    }

    public void delete(int key) {
        if (head == null) return;
    }
}

```



```

        if (head.data == key) {
            head = head.next;
            return;
        }
        Node temp = head;
        while (temp.next != null && temp.next.data != key) {
            temp = temp.next;
        }
        if (temp.next != null) temp.next = temp.next.next;
    }

    public void traverse() {
        Node temp = head;
        while (temp != null) {
            System.out.print(temp.data + " -> ");
            temp = temp.next;
        }
        System.out.println("NULL");
    }
}

```

// Doubly Linked List Implementation

```

class DoublyLinkedList {
    class Node {
        int data;
        Node prev, next;
        Node(int data) {
            this.data = data;
        }
    }

    private Node head;

    public void insert(int data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
            return;
        }
        Node temp = head;
        while (temp.next != null) {
            temp = temp.next;
        }
        temp.next = newNode;
        newNode.prev = temp;
    }

    public void delete(int key) {

```

```

    if (head == null) return;
    if (head.data == key) {
        head = head.next;
        if (head != null) head.prev = null;
        return;
    }
    Node temp = head;
    while (temp != null && temp.data != key) {
        temp = temp.next;
    }
    if (temp == null) return;
    if (temp.next != null) temp.next.prev = temp.prev;
    if (temp.prev != null) temp.prev.next = temp.next;
}

public void traverse() {
    Node temp = head;
    while (temp != null) {
        System.out.print(temp.data + " <-> ");
        temp = temp.next;
    }
    System.out.println("NULL");
}
}

```

// Circular Linked List Implementation

```

class CircularLinkedList {
    class Node {
        int data;
        Node next;
        Node(int data) {
            this.data = data;
        }
    }

    private Node last;

    public void insert(int data) {
        Node newNode = new Node(data);
        if (last == null) {
            last = newNode;
            last.next = last;
            return;
        }
        newNode.next = last.next;
        last.next = newNode;
        last = newNode;
    }
}

```

```

public void delete(int key) {
    if (last == null) return;
    if (last == last.next && last.data == key) {
        last = null;
        return;
    }
    Node temp = last;
    do {
        if (temp.next.data == key) {
            temp.next = temp.next.next;
            if (last.data == key) last = temp;
            return;
        }
        temp = temp.next;
    } while (temp != last);
}

public void traverse() {
    if (last == null) {
        System.out.println("List is empty");
        return;
    }
    Node temp = last.next;
    do {
        System.out.print(temp.data + " -> ");
        temp = temp.next;
    } while (temp != last.next);
    System.out.println("(back to head)");
}
}

```

TEST

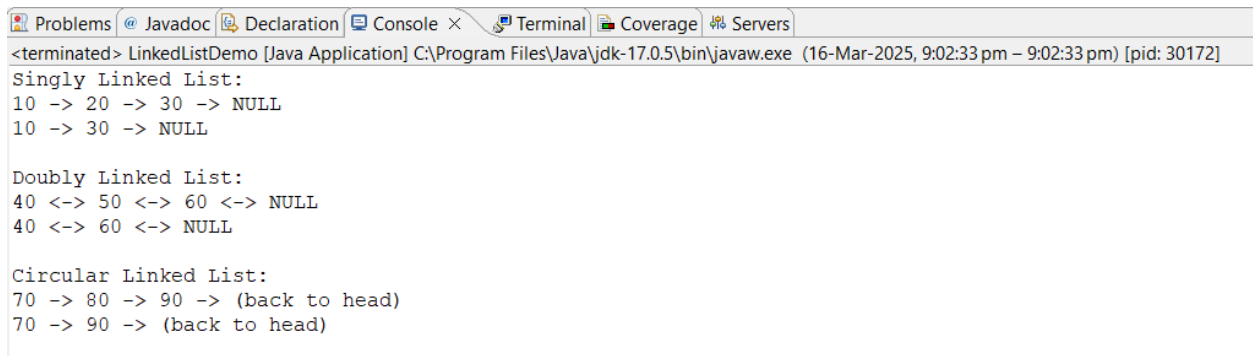
// Main Class to Test the Linked List Implementations

```
public class LinkedListDemo {
    public static void main(String[] args) {
        System.out.println("Singly Linked List:");
        SinglyLinkedList sll = new SinglyLinkedList();
        sll.insert(10);
        sll.insert(20);
        sll.insert(30);
        sll.traverse();
        sll.delete(20);
        sll.traverse();

        System.out.println("\nDoubly Linked List:");
        DoublyLinkedList dll = new DoublyLinkedList();
        dll.insert(40);
        dll.insert(50);
        dll.insert(60);
        dll.traverse();
        dll.delete(50);
        dll.traverse();

        System.out.println("\nCircular Linked List:");
        CircularLinkedList cll = new CircularLinkedList();
        cll.insert(70);
        cll.insert(80);
        cll.insert(90);
        cll.traverse();
        cll.delete(80);
        cll.traverse();
    }
}
```

OUTPUT :



```
<terminated> LinkedListDemo [Java Application] C:\Program Files\Java\jdk-17.0.5\bin\javaw.exe (16-Mar-2025, 9:02:33 pm - 9:02:33 pm) [pid: 30172]
Singly Linked List:
10 -> 20 -> 30 -> NULL
10 -> 30 -> NULL

Doubly Linked List:
40 <-> 50 <-> 60 <-> NULL
40 <-> 60 <-> NULL

Circular Linked List:
70 -> 80 -> 90 -> (back to head)
70 -> 90 -> (back to head)
```

Fig 2.1 Output Linked Lists and their operations

Performance Analysis:

Operation	Singly Linked List	Doubly Linked List	Circular Linked List
Insertion at beginning	O(1)	O(1)	O(1)
Insertion at end	O(n)	O(n)	O(1) (if tail is maintained)
Insertion at middle	O(n)	O(n)	O(n)
Deletion at beginning	O(1)	O(1)	O(1)
Deletion at end	O(n)	O(n)	O(n)
Deletion at middle	O(n)	O(n)	O(n)
Searching (Linear)	O(n)	O(n)	O(n)
Searching (Binary)	Not possible	Not possible	Not possible
Forward Traversal	O(n)	O(n)	O(n)
Backward Traversal	Not possible	O(n)	O(n) (Doubly Circular Only)

Table 2.2 Performance Analysis Linked Lists and their operations

Use Case	Recommended Linked List
Simple insertions/deletions at the start	Singly Linked List
Frequent forward and backward traversals	Doubly Linked List
Cyclic operations like CPU scheduling	Circular Linked List

Table 2.2 Use Case Linked Lists and their operations

Conclusion

Singly Linked List (SLL)

Efficient for **insertions and deletions at the beginning (O(1))**.

Inefficient for searching and inserting/deleting at the end (**O(n)**).

Best suited for **stack-like operations** (LIFO structures).

Doubly Linked List (DLL)

More versatile with **bidirectional traversal**.

Slightly **higher memory overhead** due to two pointers per node.

Best for **applications needing both forward and backward traversal** (e.g., browser history, undo/redo functionality).

Circular Linked List (CLL)

Eliminates null references and improves **end insertions (O(1))**.

Efficient in **cyclic applications** like **CPU scheduling, music playlists, and round-robin execution**.

Slightly **complex to implement** compared to SLL and DLL.

Lab Experiment No. 3

AIM: Compare Arrays with Linked Lists and analyse their performance in different scenarios.

Background:

An **Array** and a **Linked List** are two fundamental data structures used to store collections of elements.

- **Arrays** store elements in **contiguous memory locations** and allow **constant-time access** using indexing. However, their size is fixed once declared, making dynamic resizing costly.
- **Linked Lists** store elements dynamically using **nodes** that contain data and pointers to the next node. While they allow **efficient insertions and deletions**, they require extra memory for storing pointers and have **sequential access**, making searching slower.

Both have their advantages and disadvantages, depending on the **operation** and **use case**.

Scenario Best - Choice

Random Access Required	- Array
Frequent Insertions/Deletions at Start/Middle	- Linked List
Searching Large Sorted Data-	- Array (Binary Search)
Memory-Efficient Structure	- Array
Dynamic Size Requirement	- Linked List

CODE:

```
import java.util.*;

public class ArrayVsLinkedList {
    public static void main(String[] args) {
        System.out.println("Performance Comparison: Array vs Linked List");

        // Array Operations
        System.out.println("\n--- ARRAY OPERATIONS ---");
        List<Integer> arrayList = new ArrayList<>();

        long startTime = System.nanoTime();
        arrayList.add(0, 10); // Insertion at the beginning (O(n))
        arrayList.add(20);    // Insertion at the end (O(1))
        arrayList.add(1, 15); // Insertion at middle (O(n))
        long endTime = System.nanoTime();
        System.out.println("Insertion Time (Array): " + (endTime - startTime) + " ns");

        startTime = System.nanoTime();
        arrayList.remove(0); // Deletion at the beginning (O(n))
        arrayList.remove(1); // Deletion in the middle (O(n))
        arrayList.remove(arrayList.size() - 1); // Deletion at the end (O(1))
        endTime = System.nanoTime();
        System.out.println("Deletion Time (Array): " + (endTime - startTime) + " ns");
    }
}
```

```

startTime = System.nanoTime();
arrayList.contains(15); // Searching (O(n))
endTime = System.nanoTime();
System.out.println("Search Time (Array): " + (endTime - startTime) + " ns");

// Linked List Operations
System.out.println("\n--- LINKED LIST OPERATIONS ---");
LinkedList<Integer> linkedList = new LinkedList<>();

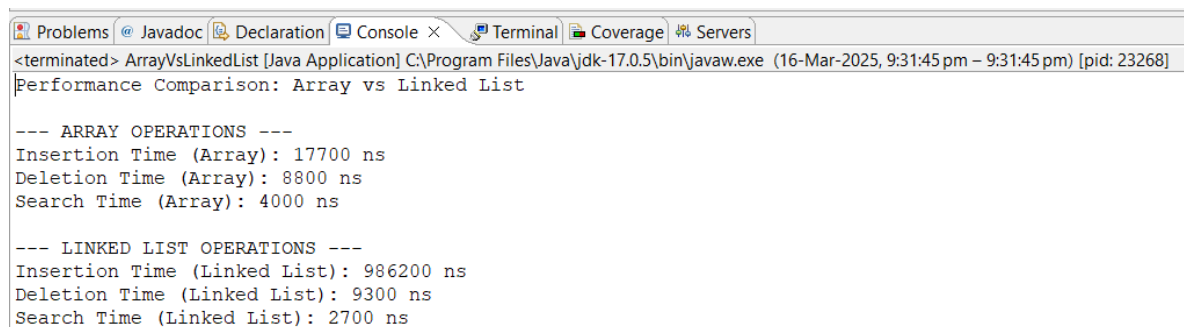
startTime = System.nanoTime();
linkedList.addFirst(10); // Insertion at the beginning (O(1))
linkedList.addLast(20); // Insertion at the end (O(1))
linkedList.add(1, 15); // Insertion at middle (O(n))
endTime = System.nanoTime();
System.out.println("Insertion Time (Linked List): " + (endTime - startTime) + " ns");

startTime = System.nanoTime();
linkedList.removeFirst(); // Deletion at the beginning (O(1))
linkedList.remove(1); // Deletion in the middle (O(n))
linkedList.removeLast(); // Deletion at the end (O(1))
endTime = System.nanoTime();
System.out.println("Deletion Time (Linked List): " + (endTime - startTime) + " ns");

startTime = System.nanoTime();
linkedList.contains(15); // Searching (O(n))
endTime = System.nanoTime();
System.out.println("Search Time (Linked List): " + (endTime - startTime) + " ns");
}
}

```

Output :



```

<terminated> ArrayVsLinkedList [Java Application] C:\Program Files\Java\jdk-17.0.5\bin\javaw.exe (16-Mar-2025, 9:31:45 pm - 9:31:45 pm) [pid: 23268]
Performance Comparison: Array vs Linked List

--- ARRAY OPERATIONS ---
Insertion Time (Array): 17700 ns
Deletion Time (Array): 8800 ns
Search Time (Array): 4000 ns

--- LINKED LIST OPERATIONS ---
Insertion Time (Linked List): 986200 ns
Deletion Time (Linked List): 9300 ns
Search Time (Linked List): 2700 ns

```

Fig 3.1 Output Compare Arrays with Linked Lists

Performance Analysis:

Operation	Array (Time Complexity)	Linked List (Time Complexity)	Remarks
Insertion at Beginning	$O(n)$ (shifting required)	$O(1)$	Linked List is faster
Insertion at End	$O(1)$ (if space available) / $O(n)$ (if resizing needed)	$O(n)$ (Singly LL), $O(1)$ (if tail pointer is maintained)	Arrays are faster if no resizing
Insertion at Middle	$O(n)$ (shifting required)	$O(n)$ (traversal required)	Similar performance
Deletion at Beginning	$O(n)$ (shifting required)	$O(1)$	Linked List is faster
Deletion at End	$O(1)$ (if no resizing needed)	$O(n)$ (Singly LL), $O(1)$ (Doubly LL with tail)	Arrays are generally better
Deletion at Middle	$O(n)$ (shifting required)	$O(n)$ (traversal required)	Similar performance
Accessing Elements (Search by Index)	$O(1)$ (direct indexing)	$O(n)$ (sequential search)	Arrays are much faster
Searching for an Element	$O(n)$ (Linear Search) / $O(\log n)$ (Binary Search for sorted array)	$O(n)$ (Linear Search)	Arrays are better for sorted data
Memory Usage	Fixed Size, No Overhead	Extra memory for pointers	Arrays are memory-efficient

Table 3.1 Performance Analysis of Arrays vs Linked Lists

Conclusion :

Array

Fast access to elements ($O(1)$) via indexing.

Efficient **for small datasets** where resizing is not frequent.

Ideal for **sorting and searching** (Binary Search is possible in sorted arrays).

Not ideal for dynamic operations due to high-cost insertions and deletions in the middle or beginning ($O(n)$).

Linked List

Fast insertions and deletions ($O(1)$) at the beginning or middle.

Efficient memory allocation since it grows dynamically.

Best for applications requiring frequent insertions and deletions, like queues, stacks, and scheduling systems.

Not ideal for searching and accessing elements since it takes **$O(n)$ time** for traversal.

Lab Experiment No. 4

Aim :

Implementation and analysis of Graph based single source shortest distance algorithms.

- Breadth first search
- Depth first search
- Dijkstra's algorithm
- Topological Sort
- Floyd–Warshall algorithm

Background:

A **graph** is a fundamental data structure used to model real-world relationships like road networks, social networks, and communication systems. Graph algorithms are essential for solving various shortest path problems. The key algorithms analyzed in this document are:

4.1 Breadth-First Search (BFS)

BFS is an algorithm used to traverse graphs level by level. It finds the shortest path in **unweighted graphs**. BFS is implemented using a **queue** and follows a **FIFO** approach.

4.2 Depth-First Search (DFS)

DFS explores as far as possible along a branch before backtracking. It is implemented using **recursion** or a **stack** and is useful in cycle detection, pathfinding, and connectivity checking.

4.3 Dijkstra's Algorithm

Dijkstra's Algorithm is used to find the shortest path from a **single source** to all other vertices in a graph with **non-negative weights**. It uses a **priority queue** (min-heap) for efficient processing.

4.4 Topological Sort

Topological Sorting is used for **Directed Acyclic Graphs (DAGs)** to determine the correct order of tasks or dependencies. It is commonly used in **task scheduling** and **course prerequisites resolution**.

4.5 Floyd–Warshall Algorithm

Floyd–Warshall Algorithm is an **all-pairs shortest path** algorithm that finds the shortest distance between **every pair of vertices** in a graph. It is a **dynamic programming** approach.

Code :

4.1 Breadth-First Search (BFS)

```
import java.util.*;

public class BFS {
    public static void bfsTraversal(Map<Integer, List<Integer>> graph, int startNode) {
        Queue<Integer> queue = new LinkedList<>();
        Set<Integer> visited = new HashSet<>();

        queue.add(startNode);
        visited.add(startNode);

        while (!queue.isEmpty()) {
            int node = queue.poll();
            System.out.print(node + " ");

            for (int neighbor : graph.getOrDefault(node, new ArrayList<>())) {
                if (!visited.contains(neighbor)) {
                    queue.add(neighbor);
                    visited.add(neighbor);
                }
            }
        }
    }

    public static void main(String[] args) {
        Map<Integer, List<Integer>> graph = new HashMap<>();
        graph.put(0, Arrays.asList(1, 2));
        graph.put(1, Arrays.asList(3, 4));
        graph.put(2, Arrays.asList(5, 6));
        graph.put(3, Arrays.asList());
        graph.put(4, Arrays.asList());
        graph.put(5, Arrays.asList());
        graph.put(6, Arrays.asList());

        System.out.println("BFS Traversal: ");
        bfsTraversal(graph, 0);
    }
}
```

4.2 Depth-First Search (DFS)

```
import java.util.*;

public class DFS {
    public static void dfsTraversal(Map<Integer, List<Integer>> graph, int node, Set<Integer> visited) {
        if (visited.contains(node)) return;

        visited.add(node);
        System.out.print(node + " ");

        for (int neighbor : graph.getOrDefault(node, new ArrayList<>())) {
            dfsTraversal(graph, neighbor, visited);
        }
    }

    public static void main(String[] args) {
        Map<Integer, List<Integer>> graph = new HashMap<>();
        graph.put(0, Arrays.asList(1, 2));
        graph.put(1, Arrays.asList(3, 4));
        graph.put(2, Arrays.asList(5, 6));

        System.out.println("DFS Traversal: ");
        dfsTraversal(graph, 0, new HashSet<>());
    }
}
```

4.3 Dijkstra's Algorithm

```
import java.util.*;

class Dijkstra {
    public static void dijkstra(Map<Integer, List<int[]>> graph, int source) {
        PriorityQueue<int[]> pq = new PriorityQueue<>(Comparator.comparingInt(a -> a[1]));
        Map<Integer, Integer> distances = new HashMap<>();

        pq.add(new int[] {source, 0});
        distances.put(source, 0);

        while (!pq.isEmpty()) {
            int[] current = pq.poll();
            int node = current[0], cost = current[1];

            for (int[] neighbor : graph.getOrDefault(node, new ArrayList<>())) {
                int nextNode = neighbor[0], weight = neighbor[1];
                int newDist = cost + weight;
            }
        }
    }
}
```

```

        if (newDist < distances.getDefault(nextNode, Integer.MAX_VALUE)) {
            distances.put(nextNode, newDist);
            pq.add(new int[] {nextNode, newDist});
        }
    }
}

System.out.println("Shortest distances: " + distances);
}

public static void main(String[] args) {
    Map<Integer, List<int[]>> graph = new HashMap<>();
    graph.put(0, Arrays.asList(new int[] {1, 4}, new int[] {2, 1}));
    graph.put(1, Arrays.asList(new int[] {3, 1}));
    graph.put(2, Arrays.asList(new int[] {1, 2}, new int[] {3, 5}));

    dijkstra(graph, 0);
}
}

```

4.4 Topological Sort

```

import java.util.*;

public class TopologicalSort {
    public static void topologicalSort(Map<Integer, List<Integer>> graph, int vertices) {
        int[] inDegree = new int[vertices];
        for (List<Integer> edges : graph.values()) {
            for (int node : edges) {
                inDegree[node]++;
            }
        }

        Queue<Integer> queue = new LinkedList<>();
        for (int i = 0; i < vertices; i++) {
            if (inDegree[i] == 0) queue.add(i);
        }

        while (!queue.isEmpty()) {
            int node = queue.poll();
            System.out.print(node + " ");

            for (int neighbor : graph.getDefault(node, new ArrayList<>())) {
                if (--inDegree[neighbor] == 0) {
                    queue.add(neighbor);
                }
            }
        }
    }

    public static void main(String[] args) {

```

```

    Map<Integer, List<Integer>> graph = new HashMap<>();
    graph.put(0, Arrays.asList(1, 2));
    graph.put(1, Arrays.asList(3));
    graph.put(2, Arrays.asList(3));

    System.out.println("Topological Sort: ");
    topologicalSort(graph, 4);
}
}

```

4.5 Floyd–Warshall Algorithm

```

import java.util.*;

public class FloydWarshall {
    static final int INF = 99999;

    public static void floydWarshall(int[][] graph) {
        int V = graph.length;
        int[][] dist = Arrays.copyOf(graph, V);

        for (int k = 0; k < V; k++) {
            for (int i = 0; i < V; i++) {
                for (int j = 0; j < V; j++) {
                    if (dist[i][k] != INF && dist[k][j] != INF) {
                        dist[i][j] = Math.min(dist[i][j], dist[i][k] + dist[k][j]);
                    }
                }
            }
        }

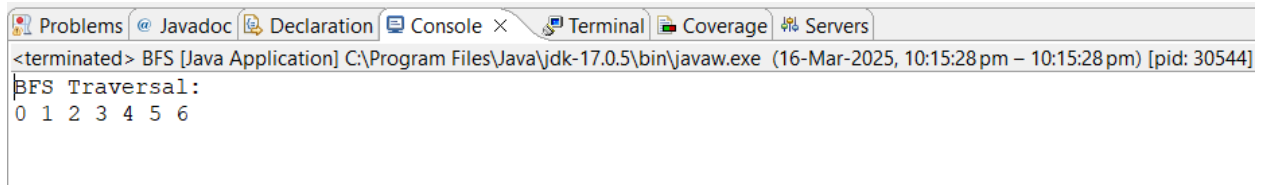
        System.out.println("Shortest distances: " + Arrays.deepToString(dist));
    }

    public static void main(String[] args) {
        int[][] graph = {
            {0, 3, INF, INF},
            {INF, 0, 1, INF},
            {INF, INF, 0, 2},
            {INF, INF, INF, 0}
        };

        floydWarshall(graph);
    }
}

```

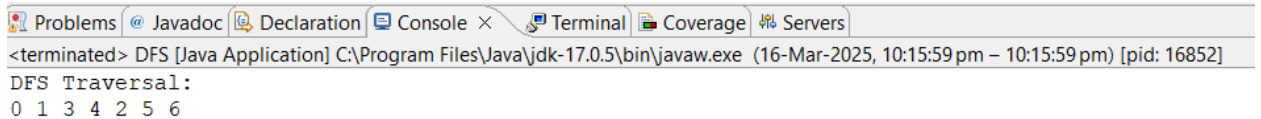
Output



The screenshot shows an IDE terminal window with the following tabs: Problems, Javadoc, Declaration, Console, Terminal, Coverage, and Servers. The terminal output is as follows:

```
<terminated> BFS [Java Application] C:\Program Files\Java\jdk-17.0.5\bin\javaw.exe (16-Mar-2025, 10:15:28 pm – 10:15:28 pm) [pid: 30544]  
BFS Traversal:  
0 1 2 3 4 5 6
```

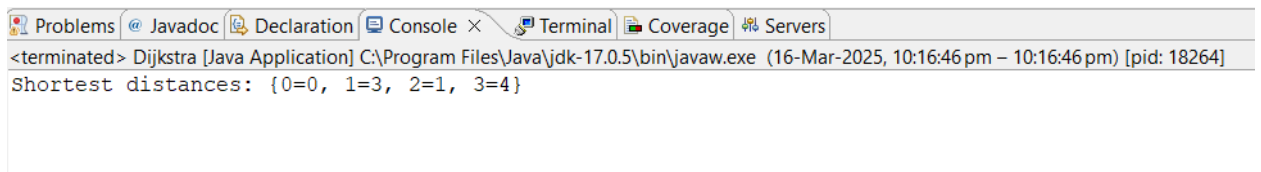
Fig4.1 Output Breadth-First Search (BFS)



The screenshot shows an IDE terminal window with the following tabs: Problems, Javadoc, Declaration, Console, Terminal, Coverage, and Servers. The terminal output is as follows:

```
<terminated> DFS [Java Application] C:\Program Files\Java\jdk-17.0.5\bin\javaw.exe (16-Mar-2025, 10:15:59 pm – 10:15:59 pm) [pid: 16852]  
DFS Traversal:  
0 1 3 4 2 5 6
```

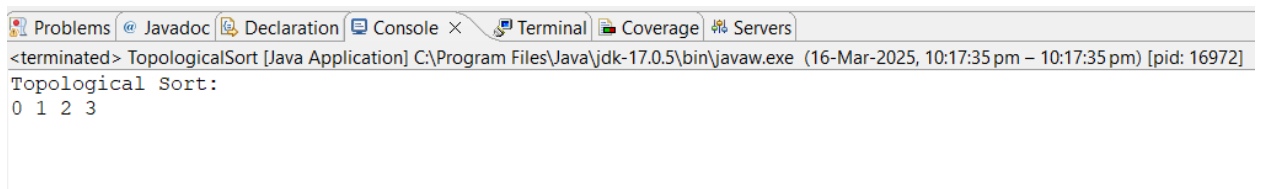
Fig 4.2 Output Depth-First Search (DFS)



The screenshot shows an IDE terminal window with the following tabs: Problems, Javadoc, Declaration, Console, Terminal, Coverage, and Servers. The terminal output is as follows:

```
<terminated> Dijkstra [Java Application] C:\Program Files\Java\jdk-17.0.5\bin\javaw.exe (16-Mar-2025, 10:16:46 pm – 10:16:46 pm) [pid: 18264]  
Shortest distances: {0=0, 1=3, 2=1, 3=4}
```

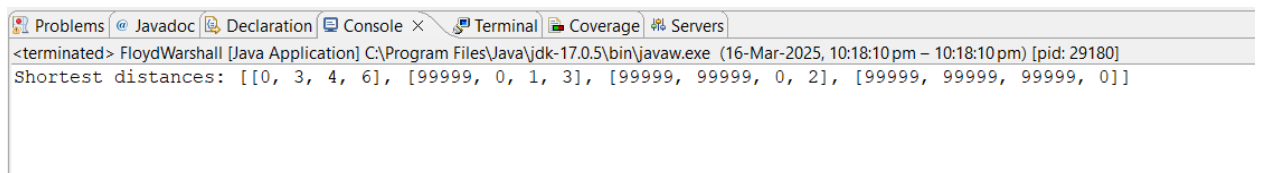
Fig 4.3 Output Dijkstra's Algorithm



The screenshot shows an IDE terminal window with the following tabs: Problems, Javadoc, Declaration, Console, Terminal, Coverage, and Servers. The terminal output is as follows:

```
<terminated> TopologicalSort [Java Application] C:\Program Files\Java\jdk-17.0.5\bin\javaw.exe (16-Mar-2025, 10:17:35 pm – 10:17:35 pm) [pid: 16972]  
Topological Sort:  
0 1 2 3
```

Fig 4.4 Output Topological Sort



The screenshot shows an IDE terminal window with the following tabs: Problems, Javadoc, Declaration, Console, Terminal, Coverage, and Servers. The terminal output is as follows:

```
<terminated> FloydWarshall [Java Application] C:\Program Files\Java\jdk-17.0.5\bin\javaw.exe (16-Mar-2025, 10:18:10 pm – 10:18:10 pm) [pid: 29180]  
Shortest distances: [[0, 3, 4, 6], [99999, 0, 1, 3], [99999, 99999, 0, 2], [99999, 99999, 99999, 0]]
```

Fig 4.5 Output Floyd–Warshall Algorithm

Performance Analysis

The performance of each algorithm depends on the type of graph (sparse/dense), the number of vertices (V), and the number of edges (E). Below is a comparative analysis of the time and space complexity of each algorithm:

Algorithm	Time Complexity	Space Complexity	Best Used For
BFS	$O(V + E)$	$O(V)$	Finding the shortest path in an unweighted graph
DFS	$O(V + E)$	$O(V)$	Cycle detection, connectivity checking, and pathfinding
Dijkstra's	$O((V + E) \log V)$	$O(V + E)$	Single-source shortest path with non-negative weights
Topological Sort	$O(V + E)$	$O(V)$	Ordering tasks in DAGs
Floyd-Warshall	$O(V^3)$	$O(V^2)$	Finding shortest paths between all pairs of vertices

Table 4.1 Breadth first search ,Depth first search ,Dijkstra's algorithm , Topological , Sort Floyd–Warshall algorithm Performance Analysis

Detailed Analysis

1. BFS & DFS
 - Best for unweighted graphs.
 - BFS is efficient for shortest path in such graphs.
 - DFS is useful for cycle detection, topological sorting, and traversing graphs.
2. Dijkstra's Algorithm
 - Uses a priority queue (Min Heap) for efficiency.
 - Best for graphs with non-negative edge weights.
 - Suitable for sparse graphs, but inefficient for dense graphs.
3. Topological Sorting
 - Used for directed acyclic graphs (DAGs).
 - Helps in scheduling problems like course prerequisites.
4. Floyd-Warshall Algorithm
 - Computes the shortest path between every pair of nodes.
 - Takes $O(V^3)$ time, making it impractical for large graphs.
 - Best for small dense graphs.

Conclusion :

- BFS and DFS are fundamental graph traversal techniques. BFS is better for shortest paths in unweighted graphs, while DFS is more suited for connectivity problems.
- Dijkstra's Algorithm is the best single-source shortest path algorithm for graphs with non-negative weights, but it struggles in dense graphs.
- Topological Sort is useful only for DAGs and helps in task scheduling.
- Floyd-Warshall is a brute-force approach for shortest paths between all pairs but is only practical for small graphs due to its $O(V^3)$ complexity.

Lab Experiment No. 5

Aim:

1. Implementation and analysis of Priority Queues using arrays, linked list and heaps.

Background :

A Priority Queue is an abstract data type where each element has a priority, and elements are served based on their priority rather than their order of insertion. It follows either Max-Priority (highest priority dequeued first) or Min-Priority (lowest priority dequeued first).

Priority Queues can be implemented in different ways:

1. Using an Array – Simple but inefficient for priority-based insertions and deletions.
2. Using a Linked List – Can maintain a sorted order but still has linear-time complexity for some operations.
3. Using a Heap – Most efficient method, offering logarithmic time complexity for insertions and deletions.

CODE :

1. Priority Queue using Arrays

```
import java.util.Arrays;

class PriorityQueueArray {
    private int[] arr;
    private int size;

    public PriorityQueueArray(int capacity) {
        arr = new int[capacity];
        size = 0;
    }

    public void insert(int value) {
        if (size == arr.length) {
            System.out.println("Queue is full");
            return;
        }
        arr[size++] = value;
    }

    public int deleteMax() { // Max-priority queue
        if (size == 0) return -1;
        int maxIdx = 0;

        for (int i = 1; i < size; i++) {
            if (arr[i] > arr[maxIdx]) {
                maxIdx = i;
            }
        }
    }
}
```



```

        int maxVal = arr[maxIdx];
        arr[maxIdx] = arr[--size]; // Replace max with last element
        return maxVal;
    }

    public void printQueue() {
        System.out.println("Queue: " + Arrays.toString(Arrays.copyOf(arr, size)));
    }

    public static void main(String[] args) {
        PriorityQueueArray pq = new PriorityQueueArray(5);
        pq.insert(3);
        pq.insert(5);
        pq.insert(1);
        pq.insert(4);

        pq.printQueue();
        System.out.println("Deleted max: " + pq.deleteMax());
        pq.printQueue();
    }
}

```

2. Priority Queue using Linked List

```

class Node {
    int data;
    Node next;

    public Node(int data) {
        this.data = data;
        this.next = null;
    }
}

class PriorityQueueLinkedList {
    private Node head; // Head always holds the highest priority element

    public void insert(int value) {
        Node newNode = new Node(value);
        if (head == null || value > head.data) { // Insert at head if priority is highest
            newNode.next = head;
            head = newNode;
        } else {
            Node temp = head;
            while (temp.next != null && temp.next.data > value) {
                temp = temp.next;
            }
            newNode.next = temp.next;
            temp.next = newNode;
        }
    }
}

```

```

public int deleteMax() {
    if (head == null) return -1;
    int maxVal = head.data;
    head = head.next; // Remove the highest priority element
    return maxVal;
}

public void printQueue() {
    Node temp = head;
    System.out.print("Queue: ");
    while (temp != null) {
        System.out.print(temp.data + " ");
        temp = temp.next;
    }
    System.out.println();
}

public static void main(String[] args) {
    PriorityQueueLinkedList pq = new PriorityQueueLinkedList();
    pq.insert(3);
    pq.insert(5);
    pq.insert(1);
    pq.insert(4);

    pq.printQueue();
    System.out.println("Deleted max: " + pq.deleteMax());
    pq.printQueue();
}
}

```

Priority Queue using Heap (Binary Heap - Min Heap)

```

import java.util.PriorityQueue;

public class PriorityQueueHeap {
    public static void main(String[] args) {
        PriorityQueue<Integer> pq = new PriorityQueue<>(); // Min-Heap

        pq.add(3);
        pq.add(5);
        pq.add(1);
        pq.add(4);

        System.out.println("Priority Queue: " + pq);
        System.out.println("Deleted min: " + pq.poll()); // Removes smallest element
        System.out.println("Priority Queue after deletion: " + pq);
    }
}

```

Output:

```
<terminated> PriorityQueueArray [Java Application] C:\Program Files\Java\jdk-17.0.5\bin\javaw.exe (16-Mar-2025, 11:40:59 pm – 11:40:59 pm) [pid: 8916]
Queue: [3, 5, 1, 4]
Deleted max: 5
Queue: [3, 4, 1]
```

Fig 5.1 Output Priority Queue using Arrays

```
Problems @ Javadoc Declaration Console × Terminal Coverage Servers
<terminated> PriorityQueueLinkedList [Java Application] C:\Program Files\Java\jdk-17.0.5\bin\javaw.exe (16-Mar-2025, 11:53:59 pm – 11:54:00 pm) [pid: 27636]
Queue: 5 4 3 1
Deleted max: 5
Queue: 4 3 1
```

Fig 5.2 Output Priority Queue using Linked List

```
Problems @ Javadoc Declaration Console × Terminal Coverage Servers
<terminated> PriorityQueueHeap [Java Application] C:\Program Files\Java\jdk-17.0.5\bin\javaw.exe (16-Mar-2025, 11:55:15 pm – 11:55:15 pm) [pid: 28040]
Priority Queue: [1, 4, 3, 5]
Deleted min: 1
Priority Queue after deletion: [3, 4, 5]
```

Fig 5.3. Output Priority Queue using Heap (Binary Heap - Min Heap)

Performance Analysis:

Implementation	Insertion Time Complexity	Deletion Time Complexity	Space Complexity
Array (Unsorted)	$O(1)$	$O(n)$	$O(n)$
Linked List	$O(n)$	$O(1)$	$O(n)$
Heap (Binary Heap)	$O(\log n)$	$O(\log n)$	$O(n)$

Table 5.1 Priority Queue Time Complexity Analysis

Observations:

1. **Array-based PQ** is **fastest for insertion** but **slowest for deletion** (since finding the max/min takes $O(n)$).
2. **Linked List PQ** allows **fast deletion ($O(1)$)** but **slow insertion ($O(n)$)**.
3. **Heap-based PQ** offers **efficient insertion and deletion** (both $O(\log n)$), making it **the best choice** for large-scale applications.

Conclusion

- For small data sizes, an **array-based priority queue** is simple to implement.
- If **deletion operations are more frequent**, a **linked list** implementation is preferred.
- For **large-scale applications**, a **heap-based priority queue** is the most efficient choice due to its logarithmic performance.

Lab Experiment No. 6

Aim:

Analyse the applications aspects of Skip Lists and compare with basic data structures..

Background :

A Skip List is a probabilistic data structure that allows fast searching, insertion, and deletion operations, making it an alternative to balanced trees like AVL trees or Red-Black trees. It extends a linked list by adding multiple layers that enable efficient binary search-like traversal.

Key Characteristics:

- Skip lists maintain multiple levels of linked lists, where higher levels "skip" more elements.
- The expected time complexity for search, insertion, and deletion is $O(\log n)$.
- They provide an alternative to self-balancing trees with a simpler implementation.

Applications of Skip Lists

Skip Lists are particularly useful in:

1. Databases (Indexing & Caching) – Used in database indexing (e.g., Redis) to store sorted key-value pairs.
2. Networking – Used in distributed systems (e.g., P2P networks) for routing.
3. Concurrent Programming – Skip lists allow efficient concurrent access compared to trees.
4. Graph Algorithms – Efficient for maintaining dynamic connectivity in graphs.
5. Memory Management – Used in garbage collection algorithms for efficient memory management.

Data Structure	Search Complexity	Insertion Complexity	Deletion Complexity	Space Complexity	Best Use Case
Skip List	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$	Efficient ordered list operations
Array	$O(n)$	$O(n)$	$O(n)$	$O(n)$	Simple but slow for dynamic operations
Linked List	$O(n)$	$O(1)$ (at head)	$O(1)$ (at head)	$O(n)$	Fast insertion/removal at ends
BST (Balanced)	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$	Fast searching and modifications
Hash Table	$O(1)$ (average), $O(n)$ (worst)	$O(1)$ (avg)	$O(1)$ (avg)	$O(n)$	Unordered key-value lookups

Table.6.1 Comparison with Other Data Structures

Observations

- Skip Lists and Balanced Trees (like AVL, Red-Black Trees) have $O(\log n)$ complexity, but Skip Lists are simpler to implement.
- Skip Lists outperform Linked Lists for searching but use more space.
- Hash Tables offer $O(1)$ search on average but lack order, making Skip Lists better for sorted data.

CODE :

```
import java.util.Random;

class SkipListNode {
    int key;
    SkipListNode[] forward;

    public SkipListNode(int key, int level) {
        this.key = key;
        this.forward = new SkipListNode[level + 1]; // Array of forward pointers
    }
}

class SkipList {
    private static final int MAX_LEVEL = 16;
    private final double P = 0.5; // Probability factor
    private final Random random;
    private SkipListNode head;
    private int level;

    public SkipList() {
        this.head = new SkipListNode(-1, MAX_LEVEL);
        this.level = 0;
        this.random = new Random();
    }

    // Generate random level for a new node
    private int randomLevel() {
        int lvl = 0;
        while (random.nextDouble() < P && lvl < MAX_LEVEL) {
            lvl++;
        }
        return lvl;
    }

    // Insert a key into the Skip List
    public void insert(int key) {
        SkipListNode[] update = new SkipListNode[MAX_LEVEL + 1];
        SkipListNode curr = head;

        for (int i = level; i >= 0; i--) {
            while (curr.forward[i] != null && curr.forward[i].key < key) {
                curr = curr.forward[i];
            }
            update[i] = curr;
        }
    }
}
```

```

int newLevel = randomLevel();
if (newLevel > level) {
    for (int i = level + 1; i <= newLevel; i++) {
        update[i] = head;
    }
    level = newLevel;
}

SkipListNode newNode = new SkipListNode(key, newLevel);
for (int i = 0; i <= newLevel; i++) {
    newNode.forward[i] = update[i].forward[i];
    update[i].forward[i] = newNode;
}
}

// Search for a key in the Skip List
public boolean search(int key) {
    SkipListNode curr = head;
    for (int i = level; i >= 0; i--) {
        while (curr.forward[i] != null && curr.forward[i].key < key) {
            curr = curr.forward[i];
        }
    }
    curr = curr.forward[0];
    return curr != null && curr.key == key;
}

// Delete a key from the Skip List
public void delete(int key) {
    SkipListNode[] update = new SkipListNode[MAX_LEVEL + 1];
    SkipListNode curr = head;

    for (int i = level; i >= 0; i--) {
        while (curr.forward[i] != null && curr.forward[i].key < key) {
            curr = curr.forward[i];
        }
        update[i] = curr;
    }

    curr = curr.forward[0];
    if (curr != null && curr.key == key) {
        for (int i = 0; i <= level; i++) {
            if (update[i].forward[i] != curr) break;
            update[i].forward[i] = curr.forward[i];
        }

        while (level > 0 && head.forward[level] == null) {

```

```

        level--;
    }
}

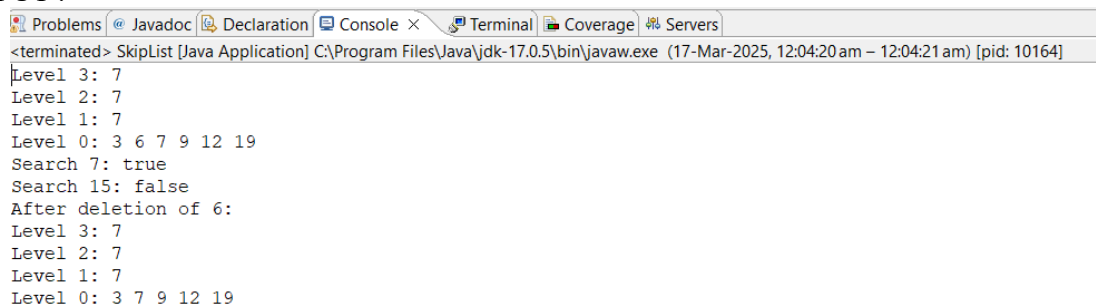
// Print the Skip List
public void printList() {
    for (int i = level; i >= 0; i--) {
        SkipListNode node = head.forward[i];
        System.out.print("Level " + i + ": ");
        while (node != null) {
            System.out.print(node.key + " ");
            node = node.forward[i];
        }
        System.out.println();
    }
}

public static void main(String[] args) {
    SkipList skipList = new SkipList();
    skipList.insert(3);
    skipList.insert(6);
    skipList.insert(7);
    skipList.insert(9);
    skipList.insert(12);
    skipList.insert(19);
    skipList.printList();
    System.out.println("Search 7: " + skipList.search(7));
    System.out.println("Search 15: " + skipList.search(15));

    skipList.delete(6);
    System.out.println("After deletion of 6:");
    skipList.printList();
}
}

```

OUTPUT :



```

<terminated> SkipList [Java Application] C:\Program Files\Java\jdk-17.0.5\bin\javaw.exe (17-Mar-2025, 12:04:20 am – 12:04:21 am) [pid: 10164]
Level 3: 7
Level 2: 7
Level 1: 7
Level 0: 3 6 7 9 12 19
Search 7: true
Search 15: false
After deletion of 6:
Level 3: 7
Level 2: 7
Level 1: 7
Level 0: 3 7 9 12 19

```

Fig 6.1 Output Skip List in Java

Performance Analysis of Skip Lists

Operation	Average Complexity	Worst-Case Complexity
Search	$O(\log n)$	$O(n)$
Insertion	$O(\log n)$	$O(n)$
Deletion	$O(\log n)$	$O(n)$

Table 6.1 Analysis of Skip Lists

Average case: $O(\log n)$ due to the layered structure that allows efficient traversal.

Worst case: $O(n)$ occurs when the randomization fails (rare case).

Conclusion

- **Skip Lists** are an alternative to balanced trees, providing efficient $O(\log n)$ performance for search, insert, and delete.
- They **require extra space** due to multiple layers but have simpler implementation than AVL or Red-Black trees.
- **Best used for** scenarios where ordered data retrieval is needed (e.g., **databases, caches, distributed systems**).
- **Not ideal** when memory is a constraint, as it requires extra pointers.

Lab Experiment No. 7

Aim:

Analyse the applications aspects of Splay Trees and compare with basic data structures

Background :

A Splay Tree is a self-adjusting binary search tree (BST) that improves performance by moving frequently accessed elements closer to the root through a process called splaying. This helps in optimizing the access time for frequently used elements, making Splay Trees useful for caching, memory management, and dynamic sets.

Key Characteristics:

- Self-adjusting BST: Every time a node is accessed, it is moved to the root using splaying.
- No explicit balancing like AVL or Red-Black Trees.
- Faster access to recently used elements (temporal locality).
- Amortized time complexity: $O(\log n)$ for search, insertion, and deletion.

Applications of Splay Trees

Splay Trees are used in scenarios where frequent access to recently used elements is required, including:

1. Memory Caching – Used in operating systems (OS) and web browsers to manage frequently accessed data efficiently.
2. Garbage Collection – Helps in tracking memory allocation and deallocation.
3. Network Routing – Used in dynamic network routing for quick updates of frequently accessed routes.
4. Database Systems – Improves efficiency of indexing and key-value storage.
5. File Systems – Used in Unix file system page management for fast access to recently used files.

Comparison with Other Data Structures

Data Structure	Search Complexity	Insertion Complexity	Deletion Complexity	Self-Balancing?	Best Use Case
Splay Tree	$O(\log n)$ (amortized)	$O(\log n)$ (amortized)	$O(\log n)$ (amortized)	Yes (self-adjusting)	Caching & frequently accessed elements
AVL Tree	$O(\log n)$	$O(\log n)$	$O(\log n)$	Yes (strict balancing)	Search-heavy applications
Red-Black Tree	$O(\log n)$	$O(\log n)$	$O(\log n)$	Yes (looser balancing)	Dynamic ordered data operations
Binary Search Tree (BST)	$O(n)$ worst	$O(n)$ worst	$O(n)$ worst	No	Simple implementation
Skip List	$O(\log n)$	$O(\log n)$	$O(\log n)$	No explicit balancing	Probabilistic alternative to trees
Hash Table	$O(1)$ (average), $O(n)$ worst	$O(1)$ avg	$O(1)$ avg	No	Fast key-value lookup

Table 7.1 Comparison with Other Data Structures

Observations:

- Splay Trees provide $O(\log n)$ complexity in amortized cases, making them faster than regular BSTs but slightly less predictable than AVL trees.
- Unlike AVL and Red-Black Trees, Splay Trees do not maintain strict balancing, but they self-adjust based on access patterns.
- Skip Lists and Hash Tables are alternatives but do not support order-based operations efficiently.

CODE:

```
class SplayTree {
    class Node {
        int key;
        Node left, right;

        public Node(int key) {
            this.key = key;
            this.left = this.right = null;
        }
    }

    private Node root;

    // Right Rotation
    private Node rightRotate(Node x) {
        Node y = x.left;
        x.left = y.right;
        y.right = x;
        return y;
    }

    // Left Rotation
    private Node leftRotate(Node x) {
        Node y = x.right;
        x.right = y.left;
        y.left = x;
        return y;
    }

    // Splaying operation: Moves a given key to the root
    private Node splay(Node root, int key) {
        if (root == null || root.key == key) return root;

        // Left subtree case
        if (key < root.key) {
            if (root.left == null) return root;

            if (key < root.left.key) { // Zig-Zig (Left Left)
                root.left.left = splay(root.left.left, key);
                root = rightRotate(root);
            } else if (key > root.left.key) { // Zig-Zag (Left Right)
                root.left.right = splay(root.left.right, key);
                if (root.left.right != null)
```

```

        root.left = leftRotate(root.left);
    }
    return (root.left == null) ? root : rightRotate(root);
}
// Right subtree case
else {
    if (root.right == null) return root;

    if (key > root.right.key) { // Zag-Zag (Right Right)
        root.right.right = splay(root.right.right, key);
        root = leftRotate(root);
    } else if (key < root.right.key) { // Zag-Zig (Right Left)
        root.right.left = splay(root.right.left, key);
        if (root.right.left != null)
            root.right = rightRotate(root.right);
    }
    return (root.right == null) ? root : leftRotate(root);
}
}

// Insert operation
public void insert(int key) {
    if (root == null) {
        root = new Node(key);
        return;
    }
    root = splay(root, key);
    if (root.key == key) return;

    Node newNode = new Node(key);
    if (key < root.key) {
        newNode.right = root;
        newNode.left = root.left;
        root.left = null;
    } else {
        newNode.left = root;
        newNode.right = root.right;
        root.right = null;
    }
    root = newNode;
}

// Search operation
public boolean search(int key) {
    root = splay(root, key);
    return root != null && root.key == key;
}

// Delete operation
public void delete(int key) {
    if (root == null) return;
    root = splay(root, key);

```

```

        if (root.key != key) return;

        if (root.left == null) {
            root = root.right;
        } else {
            Node temp = root.right;
            root = splay(root.left, key);
            root.right = temp;
        }
    }
}

// Preorder traversal
public void preOrder(Node root) {
    if (root != null) {
        System.out.print(root.key + " ");
        preOrder(root.left);
        preOrder(root.right);
    }
}

public void printTree() {
    preOrder(root);
    System.out.println();
}

public static void main(String[] args) {
    SplayTree tree = new SplayTree();
    tree.insert(10);
    tree.insert(20);
    tree.insert(30);
    tree.insert(40);
    tree.insert(50);
    tree.insert(25);

    System.out.println("Splay Tree after insertions:");
    tree.printTree();

    System.out.println("Search for 30: " + tree.search(30));
    tree.printTree();

    System.out.println("Deleting 20...");
    tree.delete(20);
    tree.printTree();
}
}

```

Output:

```
<terminated> SplayTree [Java Application] C:\Program Files\Java\jdk-17.0.5\bin\javaw.exe (17-Mar-2025, 12:18:07 am - 12:18:07 am) [pid: 8596]
Splay Tree after insertions:
25 20 10 40 30 50
Search for 30: true
30 25 20 10 40 50
Deleting 20...
10 25 30 40 50
```

Fig7.1Output splay Tree operation

Performance Analysis of Splay Trees

Operation	Average Complexity	Worst-Case Complexity
Search	$O(\log n)$ (amortized)	$O(n)$ (single access)
Insertion	$O(\log n)$ (amortized)	$O(n)$ (worst)
Deletion	$O(\log n)$ (amortized)	$O(n)$ (worst)

Table 7.1 Performance Analysis of Splay Trees

- **Amortized time complexity** is $O(\log n)$ for most operations due to self-adjustment.
- **Worst-case complexity** is $O(n)$ if access patterns are unfavorable (like a sorted insertion sequence without rotations).

Conclusion:

- **Splay Trees** are useful for **frequently accessed elements** due to self-adjusting behavior.
- They are **simpler than AVL trees** but lack strict balancing.
- **Ideal for caching, dynamic operations, and memory management.**
- **Not ideal for uniformly random access patterns**, where AVL or Red-Black Trees may be better.

Lab Experiment No. 8

Aim:

Implementation and analysis of dynamic Arrays and its operations:

- Insertion
- Deletion
- Searching
- Traversing
- Updation

Background :

A **Dynamic Array** is an array that **automatically resizes** itself when elements are added or removed. Unlike normal arrays with a fixed size, dynamic arrays grow (or shrink) as needed, maintaining **efficient** insertions and deletions.

Key Characteristics:

- Elements are stored in **contiguous memory** locations.
- **Automatic resizing** when the array is full (usually doubling the size).
- **Direct access** to elements using indexing (like a normal array).
- Backed by **static arrays** internally.

Operations on Dynamic Arrays

1. **Insertion:** Add an element; if full, resize the array.
2. **Deletion:** Remove an element and shift remaining elements.
3. **Searching:** Find an element (linear search or binary search if sorted).
4. **Traversing:** Visit and display all elements.

CODE:

```
package dynamic_arrays;
import java.util.Arrays;
class DynamicArray {
    private int[] array;
    private int size;
    private int capacity;
    public DynamicArray() {
        array = new int[2]; // Start with small initial capacity
        size = 0;
        capacity = 2;
    }
    // Resize array when needed
    private void resize() {
        capacity = capacity * 2;
        array = Arrays.copyOf(array, capacity);
    }
    // Insert element
    public void insert(int value) {
        if (size == capacity) {
            resize();
        }
    }
}
```

```

array[size++] = value;
}
// Delete element at index
public void delete(int index) {
    if (index < 0 || index >= size) {
        System.out.println("Invalid index");
        return;
    }
    for (int i = index; i < size - 1; i++) {
        array[i] = array[i + 1];
    }
    size--;
}
// Search for an element
public int search(int value) {
    for (int i = 0; i < size; i++) {
        if (array[i] == value) {
            return i;
        }
    }
    return -1;
}
// Traverse and display elements
public void traverse() {
    for (int i = 0; i < size; i++) {
        System.out.print(array[i] + " ");
    }
    System.out.println();
}
// Update element at a specific index
public void update(int index, int value) {
    if (index < 0 || index >= size) {
        System.out.println("Invalid index");
        return;
    }
    array[index] = value;
}
public static void main(String[] args) {
    DynamicArray da = new DynamicArray();
    da.insert(10);
    da.insert(20);
    da.insert(30);
    da.insert(40);
    System.out.print("Array after insertion: ");
    da.traverse();
    da.delete(1);
    System.out.print("Array after deletion at index 1: ");

```

```

da.traverse();
int index = da.search(30);
System.out.println("Searching 30: Found at index " + index);
da.update(1, 35);
System.out.print("Array after updating index 1 to 35: ");
da.traverse();
}
}

```

Output :

```

<terminated> DynamicArray [Java Application] C:\Program Files\Java\jdk-17.0.5\bin\javaw.exe (28-Apr-2025, 11:58:
Array after insertion: 10 20 30 40
Array after deletion at index 1: 10 30 40
Searching 30: Found at index 1
Array after updating index 1 to 35: 10 35 40

```

Fig 8.1 Output Dynamic Arrays operation

Performance Analysis Dynamic Arrays

Operation	Average Time Complexity	Worst-Case Time Complexity
Insertion	$O(1)$ (amortized)	$O(n)$ (if resizing needed)
Deletion	$O(n)$	$O(n)$
Searching	$O(n)$	$O(n)$
Traversing	$O(n)$	$O(n)$
Updation	$O(1)$	$O(1)$

table 8.1 Performance Analysis of Dynamic Arrays

Important Points:

- **Insertion** is $O(1)$ **amortized** because resizing happens infrequently.
- **Deletion** and **searching** require shifting or scanning, making them $O(n)$.
- **Updation** is $O(1)$ because of direct indexing.

Conclusion

- **Dynamic Arrays** are highly efficient when frequent **inserts at the end** are required.
- They are best when **random access ($O(1)$)** is needed.
- **Deletion and insertion at arbitrary positions** are **costly** due to shifting elements.
- **Resizing** can be expensive at times but happens rarely.
- **Ideal for** applications where **read-heavy operations** dominate, like lists, logs, and history managers.

Lab Experiment No. 9

Aim:

Implementation and analysis of hashing technique

Background :

Hashing is a technique to uniquely identify a specific object from a group of similar objects. It maps data to a fixed-size value (called hash code) that determines where the data will be stored or found.

In Java, HashMap and HashSet are the most popular hashing-based data structures.

1. HashMap

Definition

A **HashMap** is a part of Java's Collection Framework that stores data in **key-value pairs**. It uses a **hashing mechanism** to compute an index where the value will be stored or retrieved.

- Internally, it uses an array of buckets and a hash function.
- Each key's hashCode is computed, then compressed into an array index.
- If two keys have the same index (**collision**), they are stored together using a **Linked List** (or **Tree** structure after Java 8 for large chains).

Characteristics

- **Allows null keys** (only one null key) and multiple null values.
- **No ordering** (entries are unordered).
- **Constant time** operations (on average) for insertion, search, and deletion.

Applications

- Fast lookups in **caches, databases, and associative arrays**.

2. HashSet

Definition

A **HashSet** is a collection that contains **only unique elements** and is backed internally by a **HashMap**.

- When you insert an element, it becomes the key in the internal HashMap with a dummy constant value.
- It uses the hashCode of the object to ensure uniqueness.

Characteristics

- **No duplicates allowed.**
- **No order** is maintained.
- Efficient for **checking existence** and **unique data collection**.

Applications

- Removing duplicates from a collection.
- Efficient implementation of **sets** in mathematics or database systems.

3. Hashing Strings

Definition

- The number 31 is chosen because it is an **odd prime** and helps in better distribution of hash codes.
- The result of hashCode() is used when strings are keys in hash-based structures like **HashMap** and **HashSet**.

Characteristics

- Fast computation using integer arithmetic.
- Helps efficiently store and retrieve **string keys**.

Applications

- **Symbol tables** in compilers.
- **Caching of web pages**, sessions.
- **Username lookups** in systems.

4. Hashing Objects (with equals() and hashCode())

Definition

When you store **custom objects** (your own classes) in hash-based collections, **you must override** both equals() and hashCode().

- hashCode() decides **which bucket** the object goes to.
- equals() checks **actual equality** between two objects inside the same bucket.

Golden Rules

- **If two objects are equal according to equals(), then they must have the same hashCode().**
- **If two objects have the same hashCode(), they may or may not be equal (collision is possible).**

Characteristics

- Correct and efficient behavior of **HashMap**, **HashSet**, and other hash-based collections depends on this.
- Helps prevent **duplicate objects**.
- Ensures **fast lookups** and **efficient storage**.

Applications

- Storing **employee records**, **students' data**, **bank accounts** where custom equality checking is needed.
- Key management in **distributed systems**.

Technique	Description
HashMap	Key-value pair storage with constant-time access.
HashSet	Stores unique elements internally backed by a HashMap.
Hashing Strings	Compute a hashCode for string data.
Hashing Objects with equals() and hashCode()	Custom classes need proper equals() and hashCode() for correct hashing behavior.

Table 9.1 Hashing Techniques Explained

Code :**HashMap Implementation**

```
import java.util.HashMap;

public class HashMapExample {
    public static void main(String[] args) {
        HashMap<Integer, String> map = new HashMap<>();

        // Insertion
        map.put(1, "Apple");
        map.put(2, "Banana");
        map.put(3, "Cherry");

        // Traversal
        System.out.println("HashMap Elements:");
        for (Integer key : map.keySet()) {
            System.out.println(key + " => " + map.get(key));
        }

        // Searching
        System.out.println("Contains Key 2: " + map.containsKey(2));
        System.out.println("Contains Value 'Mango': " + map.containsValue("Mango"));

        // Deletion
        map.remove(2);
        System.out.println("After removing key 2: " + map);
    }
}
```

Output :

```
class Udemy > HashMapExample [Java Application] C:\Program Files\Java\jdk-17.0.5\bin\javaw.exe (29-Apr-2025, 12:14:22 am – 12:14:22 am) [pid: 32804]
HashMap Elements:
1 => Apple
2 => Banana
3 => Cherry
Contains Key 2: true
Contains Value 'Mango': false
After removing key 2: {1=Apple, 3=Cherry}
```

Fig 9.1 Output HashMap Implementation**HashSet Implementation**

```
public class HashSetExample {
    public static void main(String[] args) {
```

```

HashSet<String> set = new HashSet<>();
// Insertion
set.add("Red");
set.add("Green");
set.add("Blue");
// Traversal
System.out.println("HashSet Elements:");
for (String color : set) {
    System.out.println(color);
}
// Searching
System.out.println("Contains 'Green': " + set.contains("Green"));
// Deletion
set.remove("Green");
System.out.println("After removing 'Green': " + set);
}
}
<terminated> HashSetExample [Java Application] C:\Program Files\Java\jdk-17.0.5\bin\javaw.exe (29-Apr-2025, 12:15:26 am – 12:15:26 am) [pid: 32724]
HashSet Elements:
Red
Blue
Green
Contains 'Green': true
After removing 'Green': [Red, Blue]

```

Fig 9.2 Output HashMap Implementation

Hashing Strings

```

package hasing_techniques;
public class StringHashing {
    public static void main(String[] args) {
        String str = "Hello";
        System.out.println("HashCode of '" + str + "': " + str.hashCode());
        String anotherStr = "World";
        System.out.println("HashCode of '" + anotherStr + "': " + anotherStr.hashCode());
    }
}
<terminated> StringHashing [Java Application] C:\Program Files\Java\jdk-17.0.5\bin\javaw.exe (29-Apr-2025, 12:16:48 am – 12:16:48 am) [pid: 20124]
HashCode of 'Hello': 69609650
HashCode of 'World': 83766130

```

Fig 9.3 Output HashMap Implementation

Hashing Custom Objects with equals() and hashCode()

```

import java.util.HashSet;
import java.util.Objects;

```

```

class Student {
    int id;
    String name;

    public Student(int id, String name) {
        this.id = id;
        this.name = name;
    }

    // Override equals()
    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null || getClass() != obj.getClass()) return false;
        Student student = (Student) obj;
        return id == student.id && Objects.equals(name, student.name);
    }

    // Override hashCode()
    @Override
    public int hashCode() {
        return Objects.hash(id, name);
    }
}

public class CustomObjectHashing {
    public static void main(String[] args) {
        HashSet<Student> students = new HashSet<>();
        students.add(new Student(1, "Shivam"));
        students.add(new Student(2, "Singh"));
        students.add(new Student(1, "Shivam")); // Duplicate based on id and name

        System.out.println("Students in HashSet:");
        for (Student student : students) {
            System.out.println(student.id + " " + student.name);
        }
    }
}

```

Output

```
Problems Javadoc Declaration Console × Terminal Coverage Servers Debug
<terminated> CustomObjectHashing [Java Application] C:\Program Files\Java\jdk-17.0.5\bin\javaw.exe (29-Apr-2025, 12:19:04 am – 12:19:04 am) [pid: 22756]
Students in HashSet:
1 Shivam
2 Singh
```

Fig 9.4 Output HashMap Implementation

Performance Analysis

Operation	HashMap	HashSet	String Hashing	Object Hashing
Insertion	O(1) average	O(1) average	O(1)	O(1)
Deletion	O(1) average	O(1) average	-	O(1)
Searching	O(1) average	O(1) average	O(1)	O(1)
Traversal	O(n)	O(n)	-	O(n)

Table 9.2 Performance Analysis

Conclusion

- **HashMap** is best for key-value pair storage with fast access.
- **HashSet** ensures **uniqueness** of elements.
- **String hashing** is used in many internal data structures (like symbol tables, caching).
- **Custom object hashing** requires correct implementation of **equals()** and **hashCode()** to avoid unexpected behavior.
- **Hashing** enables **constant-time** average complexity for insertion, deletion, and search operations — making it highly efficient for **large datasets**.