Lab Experiment No. 1:

Aim:

To implement and analyze the fundamental operations of Arrays, including:
● Insertion
● Deletion
● Searching
● Traversal
● Updation

Background:

An array is a linear data structure that stores elements of the same data type in contiguous memory locations. It offers constant-time access to elements via indexing. However, arrays have a fixed size, which must be defined during declaration and cannot be changed dynamically.

Operations on Arrays:
Arrays support multiple operations that serve various computational purposes:

1. **Insertion:**
   Adding a new element at a specified position. While insertion at the end is efficient, inserting in the middle or at the beginning requires shifting elements to accommodate the new value.

2. **Deletion:**
   Removing an element from a specified position. Similar to insertion, deletion from the middle or start involves shifting elements to maintain array structure.

3. **Searching:**
   Locating a particular element in the array. Common methods include:
   ● **Linear Search** – sequential checking, ideal for unsorted arrays
   ● **Binary Search** – efficient search in sorted arrays using divide-and-conquer

4. **Traversal:**
   Accessing and processing each element of the array sequentially, typically used for displaying or manipulating array contents.

5. **Updation:**
   Modifying the value of an element at a specified index. Due to direct indexing, this is a constant-time operation.

Algorithm:

1. Insertion in array:

1. Check if the array has space.
2. Identify the index for insertion.
3. Shift elements one position to the right from that index.
4. Insert the element.
5. Increase array size.

2. Deletion in array:

1. Check if the array is empty.
2. Identify the index to be deleted.
3. Shift subsequent elements one step to the left.
4. Decrease array size.

3. Searching in array:

1. Loop through the array from the start.
2. Compare each element with the target.
3. If found, return the index.
4. If not found, return "Not Found".

4. Traversal in array:

1. Start from the first element.
2. Access each element sequentially.
3. Perform desired operation (e.g., print).
4. Continue until the last element is processed.

5. Updation in array:

1. Locate the index of the element to be updated.
2. Replace the value at that index with a new value.

Code:

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

#define MAX_SIZE 100

// Function declarations
void insert(int arr[], int &size, int element, int position);
void remove(int arr[], int &size, int position);
int search(int arr[], int size, int element);
void traverse(int arr[], int size);
void update(int arr[], int size, int position, int new_value);

int main() {
    int arr[MAX_SIZE], size, element, position, new_value;

    cout << "Enter the number of elements in array: ";
    cin >> size;

    cout << "Enter " << size << " elements: ";
    for (int i = 0; i < size; i++) {
        cin >> arr[i];
    }
    cout << "\nInitial Array:\n";
    traverse(arr, size);

    // Insertion
    cout << "\nInsertion:\n";
    cout << "Enter element to insert: ";
    cin >> element;
    cout << "Enter position (1-based index): ";
    cin >> position;
    insert(arr, size, element, position);
    traverse(arr, size);

    // Deletion
    cout << "\nDeletion:\n";
    cout << "Enter position to delete (1-based index): ";
    cin >> position;
    remove(arr, size, position);
    traverse(arr, size);

    // Searching
    cout << "\nSearching:\n";
    cout << "Enter element to search: ";
    cin >> element;
    position = search(arr, size, element);
    if (position != -1)
        cout << "Element found at position " << position + 1 << endl;
```

```
    else
        cout << "Element not found\n";

    // Updation
    cout << "\nUpdation:\n";
    cout << "Enter position to update (1-
based index): ";
    cin >> position;
    cout << "Enter new value: ";
    cin >> new_value;
    update(arr, size, position, new_value);
    traverse(arr, size);

    cout << "\nAll operations completed
successfully!\n";

    return 0;
}

void insert(int arr[], int &size, int element,
int position) {
    if (size >= MAX_SIZE) {
        cout << "Array is full! Cannot
insert.\n";
        return;
    }
    if (position < 1 || position > size + 1) {
        cout << "Invalid position!\n";
        return;
    }
    for (int i = size; i >= position; i--) {
        arr[i] = arr[i - 1];
    }
    arr[position - 1] = element;
    size++;
    cout << "Element inserted
successfully!\n";
}

void remove(int arr[], int &size, int
position) {
    if (position < 1 || position > size) {
        cout << "Invalid position!\n";
        return;
    }
    for (int i = position - 1; i < size - 1; i++)
{
        arr[i] = arr[i + 1];
    }
    size--;
    cout << "Element deleted
successfully!\n";
}

int search(int arr[], int size, int element) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == element) {
            return i;
        }
    }
    return -1;
}

void traverse(int arr[], int size) {
    cout << "Current Array Elements: ";
    for (int i = 0; i < size; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}

void update(int arr[], int size, int position,
int new_value) {
    if (position < 1 || position > size) {
        cout << "Invalid position!\n";
        return;
    }
    arr[position - 1] = new_value;
    cout << "Element updated
successfully!\n";
}
```

Output:

```
Enter the number of elements in array: 4
Enter 4 elements: 1
2
3
4

Initial Array:
Current Array Elements: 1 2 3 4

Insertion:
Enter element to insert: 36
Enter position (1-based index): 4
Element inserted successfully!
Current Array Elements: 1 2 3 36 4

Deletion:
Enter position to delete (1-based index): 4
Element deleted successfully!
Current Array Elements: 1 2 3 4

Searching:
Enter element to search: 36
Element not found

Updation:
Enter position to update (1-based index): 1
Enter new value: 22
Element updated successfully!
Current Array Elements: 22 2 3 4

All operations completed successfully!
```

Fig 1.1 Array operations output

Performance Analysis:

Table 1.1 Analysis of time complexity for Array

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)

Conclusion:

- Array operations demonstrate varying efficiency depending on the type and location of the operation.
- Insertion and deletion in the middle or beginning require element shifting, making them less efficient.
- Searching in an unsorted array is linear, while binary search can optimize search operations if the array is sorted.
- Updation is fast and direct, thanks to index-based access.
- Static arrays are limited in size and flexibility, encouraging the use of dynamic structures like linked lists, hash tables, or trees for more demanding applications.

Lab Experiment No. 2:

Aim:

Implementation and analysis of Linked Lists and its operations.

- Singly Linked List
    - Insertion of a new node
    - Deletion of a node
    - Traversing
- Doubly Linked List
    - Insertion of a new node
    - Deletion of a node
    - Traversing
- Circular Linked List
    - Insertion of a new node
    - Deletion of a node
    - Traversing

Background:

A linked list is a linear data structure where each element (called a node) is stored in a separate object, and each node contains a reference (or link) to the next node in the sequence. Unlike arrays, linked lists allow dynamic memory allocation, making them more flexible when dealing with data of unknown or changing sizes.

There are three primary types of linked lists:

1. **Singly Linked List (SLL):** Each node contains data and a pointer to the next node. Traversal is unidirectional.
2. **Doubly Linked List (DLL):** Each node contains data, a pointer to the next node, and a pointer to the previous node, allowing bidirectional traversal.
3. **Circular Linked List (CLL):** A variation where the last node points back to the first node, forming a loop. It can be singly or doubly linked.

Operations:

- **Insertion:** Adding a new node to the list either at the beginning, middle, or end.
- **Deletion:** Removing a node from a specific position.
- **Traversal:** Visiting each node in the list to access or display data.

Algorithm:

1. Singly Linked List

Insertion:

1. Create a new node.
2. Set its link to the desired next node.
3. Adjust the previous node's link (if inserting in between or at end).

Deletion:

1. Traverse to the node before the one to be deleted.
2. Store the reference to the node to be deleted.

3. Update the previous node's link to skip the deleted node.
4. Free the memory of the deleted node.

Traversal:

1. Start from the head.
2. While the current node is not NULL: - Access the data. - Move to the next node.

2. Doubly Linked List

Insertion:

1. Create a new node.
2. Set its next and prev pointers.
3. Update the surrounding nodes' pointers accordingly.

Deletion:

1. Locate the node to delete.
2. Update the next of the previous node and the prev of the next node.
3. Free the memory.

Traversal:

- Forward traversal: Start from head, follow next.
- Backward traversal: Start from tail, follow prev.

3. Circular Linked List

Insertion:

1. Create a new node.
2. If list is empty, point new node's next to itself.
3. Otherwise, insert it after a specific node and adjust links.

Deletion:

1. Find the node and its previous node.
2. Update previous node's link to skip the deleted node.
3. Free memory. Handle case when deleting last node.

Traversal:

1. Start from any node.
2. Use a loop that stops when the starting node is reached again.

Code:

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

// --------- Singly Linked List ----------------

struct SinglyNode {
    int data;
    SinglyNode* next;
};

void insertSingly(SinglyNode*& head, int
value) {
    SinglyNode* newNode = new
SinglyNode{value, nullptr};
    if (!head) {
        head = newNode;
    } else {
        SinglyNode* temp = head;
        while (temp->next)
            temp = temp->next;
        temp->next = newNode;
    }
    cout << "Inserted " << value << " into
Singly Linked List." << endl;
}

void deleteSingly(SinglyNode*& head, int
value) {
    SinglyNode* temp = head;
    SinglyNode* prev = nullptr;
```

```
    while (temp && temp->data != value) {
        prev = temp;
        temp = temp->next;
    }

    if (!temp) {
        cout << "Value " << value << " not
found in Singly Linked List." << endl;
        return;
    }

    if (!prev) head = head->next;
    else prev->next = temp->next;

    delete temp;
    cout << "Deleted " << value << " from
Singly Linked List." << endl;
}

void traverseSingly(SinglyNode* head) {
    cout << "Singly Linked List: ";
    while (head) {
        cout << head->data;
        if (head->next) cout << " -> ";
        head = head->next;
    }
    cout << endl;
}

// ---------Doubly Linked List ----------------

struct DoublyNode {
    int data;
    DoublyNode* prev;
    DoublyNode* next;
};

void insertDoubly(DoublyNode*& head,
int value) {
    DoublyNode* newNode = new
DoublyNode{value, nullptr, nullptr};
    if (!head) {
        head = newNode;
    } else {
        DoublyNode* temp = head;
        while (temp->next)
            temp = temp->next;
        temp->next = newNode;
        newNode->prev = temp;
    }
    cout << "Inserted " << value << " into
Doubly Linked List." << endl;
}

void deleteDoubly(DoublyNode*& head,
int value) {
    DoublyNode* temp = head;

    while (temp && temp->data != value)
        temp = temp->next;

    if (!temp) {
        cout << "Value " << value << " not
found in Doubly Linked List." << endl;
        return;
    }

    if (temp->prev) temp->prev->next =
temp->next;
    else head = temp->next;

    if (temp->next) temp->next->prev =
temp->prev;

    delete temp;
    cout << "Deleted " << value << " from
Doubly Linked List." << endl;
}

void traverseDoubly(DoublyNode* head)
{
    cout << "Doubly Linked List: ";
    while (head) {
        cout << head->data;
        if (head->next) cout << " <-> ";
        head = head->next;
    }
    cout << endl;
}

// --------Circular Linked List ----------------

struct CircularNode {
    int data;
    CircularNode* next;
};

void insertCircular(CircularNode*& tail,
int value) {
```

```
    CircularNode* newNode = new
CircularNode{value, nullptr};
    if (!tail) {
        newNode->next = newNode;
        tail = newNode;
    } else {
        newNode->next = tail->next;
        tail->next = newNode;
        tail = newNode;
    }
    cout << "Inserted " << value << " into
Circular Linked List." << endl;
}

void deleteCircular(CircularNode*& tail,
int value) {
    if (!tail) {
        cout << "Circular Linked List is
empty." << endl;
        return;
    }

    CircularNode* curr = tail->next;
    CircularNode* prev = tail;
    bool found = false;

    do {
        if (curr->data == value) {
            found = true;
            break;
        }
        prev = curr;
        curr = curr->next;
    } while (curr != tail->next);

    if (!found) {
        cout << "Value " << value << " not
found in Circular Linked List." << endl;
        return;
    }

    if (curr == prev) {
        delete curr;
        tail = nullptr;
    } else {
        prev->next = curr->next;
        if (curr == tail) tail = prev;
        delete curr;
    }
    cout << "Deleted " << value << " from
Circular Linked List." << endl;
}

void traverseCircular(CircularNode* tail)
{
    cout << "Circular Linked List: ";
    if (!tail) {
        cout << "List is empty." << endl;
        return;
    }

    CircularNode* curr = tail->next;
    do {
        cout << curr->data;
        curr = curr->next;
        if (curr != tail->next) cout << " -> ";
    } while (curr != tail->next);
    cout << endl;
}

// ------------Main Function ------------------

int main() {
    SinglyNode* singlyHead = nullptr;
    DoublyNode* doublyHead = nullptr;
    CircularNode* circularTail = nullptr;

    // --- Singly Linked List  ---
    cout << "\n- Singly Linked List
Operations -----" << endl;
    insertSingly(singlyHead, 10);
    insertSingly(singlyHead, 20);
    insertSingly(singlyHead, 30);
    traverseSingly(singlyHead);
    deleteSingly(singlyHead, 20);
    traverseSingly(singlyHead);

    // --- Doubly Linked List  ---
    cout << "\n- Doubly Linked List
Operations -----" << endl;
    insertDoubly(doublyHead, 5);
    insertDoubly(doublyHead, 15);
    insertDoubly(doublyHead, 25);
    traverseDoubly(doublyHead);
    deleteDoubly(doublyHead, 15);
    traverseDoubly(doublyHead);

    // --- Circular Linked List  ---
```

```
    cout << "\n- Circular Linked List
Operations -----" << endl;
    insertCircular(circularTail, 100);
    insertCircular(circularTail, 200);
    insertCircular(circularTail, 300);
    traverseCircular(circularTail);
    deleteCircular(circularTail, 200);
    traverseCircular(circularTail);

    return 0;
}
```

Output:

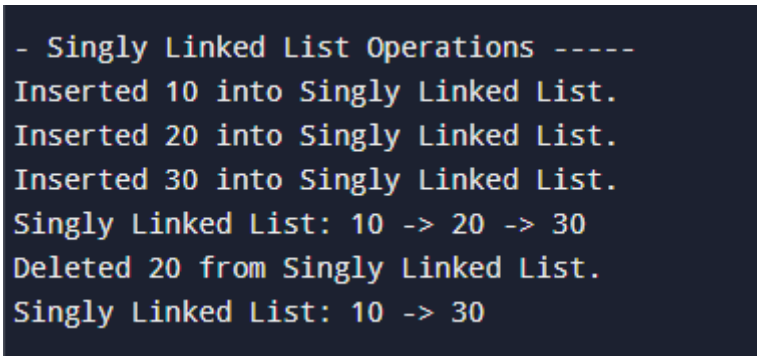

Fig 2.1 Singly Linked List operations output.

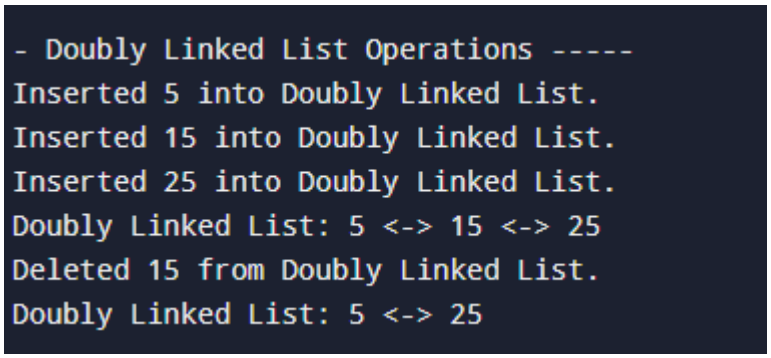

Fig 2.2 Doubly Linked List operations output.

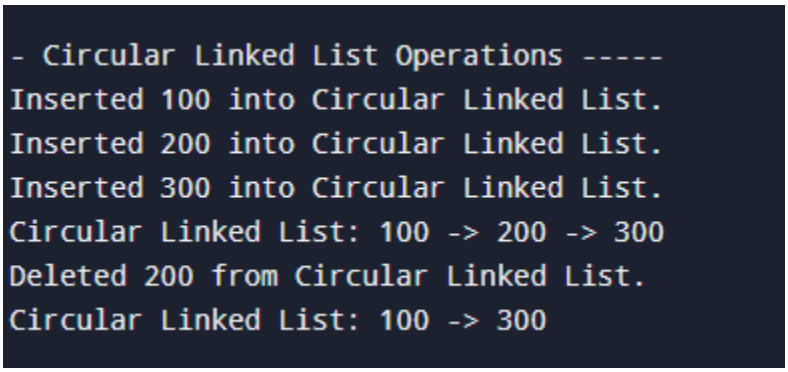

Fig 2.3 Circular Linked List operations output.

Performance Analysis:

Table 2.1 Analysis of time complexity for Linked List

Operation	**Singly Linked List**	**Doubly Linked List**	**Circular Linked List**
Insertion	O(1) at the beginning O(n) at end or specific positions	O(1) at the beginning O(n) at end or specific positions	O(1) at the beginning O(n) at end or specific positions
Deletion	O(1) at the beginning O(n) at end or specific positions	O(1) at the beginning O(n) at end or specific positions	O(1) at the beginning O(n) at end or specific positions
Traversing	O(n)	O(n)	O(n)

Conclusion:

- Linked lists provide efficient dynamic memory utilization compared to static arrays.
- Singly Linked Lists are simple and useful for forward traversal but lack reverse traversal capability.
- Doubly Linked Lists improve on SLL by enabling bidirectional traversal and easier deletion of nodes.
- Circular Linked Lists ensure that traversals can continue indefinitely and are useful in buffering applications.
- Though linked lists offer flexibility, operations like searching and random access are slower compared to arrays, making the choice of data structure highly application-dependent.

Lab Experiment 3:

Aim:

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

Background:

Arrays and Linked Lists are both linear data structures used to store collections of elements. However, they differ significantly in memory allocation, data access, and performance of operations such as insertion, deletion, and traversal.

Arrays:

- Contiguous memory allocation.
- Constant time (O(1)) access via indexing.
- Fixed size after declaration.
- Insertion and deletion require shifting elements.

Linked Lists:

- Nodes allocated dynamically and linked via pointers.
- Sequential access (no indexing).
- Can grow and shrink in size at runtime.
- Insertion and deletion are efficient if pointer is available.

Table 3.1 Key differences between Array and Linked List

Feature	Arrays	Linked Lists
Memory Allocation	Static: Memory is allocated at compile-time and cannot be resized.	Dynamic: Memory is allocated at runtime as nodes are created and freed.
Access Time	O(1): Direct access using an index.	O(n): Requires traversal from the head to the desired node.
Insertion (Middle)	O(n): All elements after the position must be shifted to the right.	O(1): If pointer to previous node is available, just adjust links.
Deletion (Middle)	O(n): Requires shifting all subsequent elements to the left.	O(1): If pointer to the previous node is known, unlink and delete the node.
Traversal	O(n): Access each element using a loop and index.	O(n): Traverse nodes one by one using pointers.
Memory Usage	Compact: Only stores data elements, with no extra space overhead.	Extra Overhead: Each node stores a pointer (or two in doubly linked lists).
Flexibility in Size	Fixed Size: Size must be defined at creation and cannot be changed.	Dynamic Size: Grows or shrinks as nodes are added or removed at runtime.

Algorithm:

1. Insertion
   - Array:
     1. Check if there is space in the array.
     2. Shift all elements from the desired position one step to the right.
     3. Insert the new element at the specified index.
     4. Increase the size of the array.
   - Linked List:
     1. Create a new node.
     2. Traverse the list to the desired position (or use a pointer to it).
     3. Adjust the next (and prev for doubly linked list) pointers to insert the node.
     4. Update the head/tail if needed.

2. Deletion
   - Array:
     1. Identify the index of the element to be deleted.
     2. Shift all elements after the index one step to the left.
     3. Decrease the size of the array.
   - Linked List:
     1. Traverse to the node to be deleted (or use a pointer to it).
     2. Adjust the pointers of the previous node to skip the target node.
     3. Deallocate (free) the memory of the deleted node.
     4. Update the head/tail if needed.

3. Traversal
   - Array:
     1. Use a for loop from index 0 to size - 1.
     2. Access and process each element using arr[i].
   - Linked List:
     1. Start from the head of the list.
     2. Use a while loop to traverse each node using pointers.
     3. Process the data in each node until the end (NULL or back to head for circular list).

4. Searching
   - Array:
     - Linear Search: Iterate through each element and compare with the target value - O(n).
     - Binary Search: Applicable only if the array is sorted - O(log n).
   - Linked List:
     - Linear Search Only: Traverse each node and compare the value - O(n).
     - Binary Search Not Applicable due to lack of random access.

Code:

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

#define MAX_SIZE 100

// -------------------- Array Operations --------------------
void arrayInsertion(int arr[], int &size, int element, int position) {
    if (size >= MAX_SIZE) {
        cout << "Array is full. Cannot insert.\n";
        return;
    }
    if (position < 1 || position > size + 1) {
        cout << "Invalid position!\n";
        return;
    }
    for (int i = size; i >= position; i--) {
        arr[i] = arr[i - 1];
    }
    arr[position - 1] = element;
    size++;
}

void arrayDeletion(int arr[], int &size, int position) {
    if (position < 1 || position > size) {
        cout << "Invalid position!\n";
        return;
    }
    for (int i = position - 1; i < size - 1; i++) {
        arr[i] = arr[i + 1];
    }
    size--;
}

void arrayTraversal(int arr[], int size) {
    cout << "Array Elements: ";
    for (int i = 0; i < size; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}

// -------------------- Singly Linked List --------------------
struct Node {
    int data;
    Node* next;
};

Node* head = NULL;

void linkedListInsertion(int element, int position) {
    Node* newNode = new Node();
    newNode->data = element;
    newNode->next = NULL;

    if (position < 1) {
        cout << "Invalid position!\n";
        return;
    }

    if (position == 1) {
        newNode->next = head;
        head = newNode;
        return;
    }

    Node* temp = head;
    for (int i = 1; temp != NULL && i < position - 1; i++) {
        temp = temp->next;
    }

    if (temp == NULL) {
        cout << "Position out of range!\n";
        return;
    }

    newNode->next = temp->next;
    temp->next = newNode;
}

void linkedListDeletion(int position) {
    if (head == NULL || position < 1) {
        cout << "Invalid position or empty list!\n";
        return;
    }

    Node* temp = head;
```

```
    if (position == 1) {
        head = head->next;
        delete temp;
        return;
    }

    for (int i = 1; temp != NULL && i < position - 1; i++) {
        temp = temp->next;
    }

    if (temp == NULL || temp->next == NULL) {
        cout << "Position out of range!\n";
        return;
    }

    Node* del = temp->next;
    temp->next = del->next;
    delete del;
}

void linkedListTraversal() {
    Node* temp = head;
    cout << "Linked List Elements: ";
    while (temp != NULL) {
        cout << temp->data;
        if (temp->next != NULL) cout << " -> ";
        temp = temp->next;
    }
    cout << endl;
}

// -------------------- Main --------------------
int main() {
    int arr[MAX_SIZE], arrSize = 0;
    int choice, element, position;

    cout << "Demonstrating Array Operations:\n";
    arrInsertionDemo:
    arrayInsertion(arr, arrSize, 10, 1);
    arrayInsertion(arr, arrSize, 20, 2);
    arrayInsertion(arr, arrSize, 30, 3);
    arrayTraversal(arr, arrSize);
    arrayDeletion(arr, arrSize, 2);
    arrayTraversal(arr, arrSize);

    cout << "\nDemonstrating Linked List Operations:\n";
    linkedListInsertion(10, 1);
    linkedListInsertion(20, 2);
    linkedListInsertion(30, 3);
    linkedListTraversal();
    linkedListDeletion(2);
    linkedListTraversal();

    return 0;
}
```

Output:

```
Demonstrating Array Operations:
Array Elements: 55 65 95
Array Elements: 55 95

Demonstrating Linked List Operations:
Linked List Elements: 55 -> 65 -> 95
Linked List Elements: 55 -> 95
```

Fig 3.1 Array and Linked List operations output.

Performance Analysis:

Table 3.2 Analysis of time complexity for comparison between Array and Linked List

Operation	Arrays (Time Complexity)	Linked Lists (Time Complexity)	Performance Comparison
Access	O(1) – Direct indexing	O(n) – Sequential traversal	Arrays are faster
Insertion (Beginning)	O(n) – Shifting elements required	O(1) – Direct pointer adjustment	Linked Lists are better
Insertion (Middle/End)	O(n)	O(n) – Need to traverse to position	Similar performance
Deletion (Beginning)	O(n) – Shifting elements required	O(1) – Adjust head pointer	Linked Lists are better
Deletion (Middle/End)	O(n)	O(n) – Need to traverse to position	Similar performance
Memory Usage	Efficient – Contiguous memory block	Extra memory required for pointers	Arrays use less memory
Cache Performance	Excellent – Better locality	Poor – Nodes scattered in memory	Arrays perform better in cache utilization

Conclusion:

- Arrays provide fast random access but suffer from fixed size and expensive insert/delete operations.
- Linked Lists are better suited for dynamic data where frequent insertion/deletion is required.
- In applications requiring constant resizing or pointer manipulation, linked lists are superior.
- For frequent access and minimal updates, arrays are more efficient due to constant-time indexing.
- Understanding the trade-offs between arrays and linked lists is essential for optimal data structure selection in real-world applications.

Lab Experiment 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:

Graphs are a fundamental data structure used to model relationships between pairs of objects. A graph is defined by a set of vertices (nodes) and a set of edges (connections). Based on their structure, graphs can be directed or undirected, weighted or unweighted.

Several algorithms are used to traverse or analyse graphs depending on the problem. Some of the most commonly used algorithms are:

1. Breadth First Search (BFS):
    - Explores nodes level by level using a queue.
    - Suitable for finding the shortest path in unweighted graphs.
    - Time Complexity: O(V + E)
2. Depth First Search (DFS):
    - Explores as far as possible along each branch using a stack (or recursion).
    - Useful in topological sort, cycle detection, and pathfinding.
    - Time Complexity: O(V + E)
3. Dijkstra's Algorithm:
    - Finds the shortest path from a single source to all other vertices in a graph with non-negative edge weights.
    - Uses a priority queue (min-heap) for efficiency.
    - Time Complexity: O((V + E) log V) using a min-heap
4. Topological Sort:
    - Applies to Directed Acyclic Graphs (DAGs).
    - Orders vertices linearly such that for every directed edge (u → v), u appears before v.
    - Time Complexity: O(V + E)
5. Floyd–Warshall Algorithm:
    - Computes shortest paths between all pairs of vertices.
    - Works on graphs with positive or negative weights (but no negative cycles).
    - Time Complexity: $O(V^3)$

Algorithm:

1. Breadth First Search (BFS)

1. Initialize a queue and a visited array.
2. Enqueue the starting node and mark as visited.
3. While the queue is not empty:
    - Dequeue a vertex, visit it.
    - Enqueue all unvisited adjacent vertices.

2. Depth First Search (DFS)
    1. Use a recursive function or a stack.
    2. Mark the current node as visited.
    3. Visit all unvisited adjacent vertices recursively.
3. Dijkstra's Algorithm
    1. Initialize distance of all vertices as infinity, except the source as 0.
    2. Use a priority queue to select the node with the smallest tentative distance.
    3. Update distances of adjacent vertices.
    4. Repeat until all vertices are processed.
4. Topological Sort (Using DFS)
    1. Initialize a visited array and a stack.
    2. For every unvisited vertex, perform DFS.
    3. After visiting all adjacent nodes, push the current node to the stack.
    4. Reverse the stack to get the topological order.
5. Floyd–Warshall Algorithm
    1. Create a distance matrix initialized to infinity, except for zeros on the diagonal.
    2. For each vertex k:
        - For each pair (i, j):
            - If distance[i][j] > distance[i][k] + distance[k][j], update it.

Code:

```
#include <iostream>
#include <vector>
#include <queue>
#include <stack>
#include <climits>
using namespace std;

const int MAX = 100;
int iteration_counter = 0;

// BFS
void BFS(vector<vector<int>>& graph, int
vertices, int start) {
   vector<bool> visited(vertices, false);
   queue<int> q;
   q.push(start);
   visited[start] = true;
   iteration_counter = 0;

   cout << "BFS Traversal: ";
   while (!q.empty()) {
      int node = q.front();
      q.pop();
      cout << node << " ";
      iteration_counter++;

      for (int i = 0; i < vertices; ++i) {
         if (graph[node][i] && !visited[i]) {
            q.push(i);
            visited[i] = true;
         }
      }
   }
   cout << "\nBFS Iterations: " <<
iteration_counter << "\n";
}

// DFS
void DFSUtil(vector<vector<int>>&
graph, int vertices, int node,
vector<bool>& visited) {
   visited[node] = true;
   cout << node << " ";
   iteration_counter++;

   for (int i = 0; i < vertices; ++i) {
      if (graph[node][i] && !visited[i]) {
         DFSUtil(graph, vertices, i, visited);
      }
   }
}
```

```
void DFS(vector<vector<int>>& graph,
int vertices, int start) {
   vector<bool> visited(vertices, false);
   iteration_counter = 0;

   cout << "DFS Traversal: ";
   DFSUtil(graph, vertices, start, visited);
   cout << "\nDFS Iterations: " <<
iteration_counter << "\n";
}

// Dijkstra
void Dijkstra(vector<vector<int>>&
graph, int vertices, int start) {
   vector<int> dist(vertices, INT_MAX);
   vector<bool> visited(vertices, false);
   dist[start] = 0;
   iteration_counter = 0;

   for (int count = 0; count < vertices - 1;
++count) {
      int min = INT_MAX, u = -1;

      for (int v = 0; v < vertices; ++v) {
         if (!visited[v] && dist[v] <= min) {
            min = dist[v];
            u = v;
         }
      }

      visited[u] = true;

      for (int v = 0; v < vertices; ++v) {
         if (!visited[v] && graph[u][v] &&
            dist[u] != INT_MAX &&
            dist[u] + graph[u][v] < dist[v]) {
            dist[v] = dist[u] + graph[u][v];
         }
         iteration_counter++;
      }
   }

   cout << "Dijkstra's Shortest Distances
from Source " << start << ":\n";
   for (int i = 0; i < vertices; ++i) {
      cout << "To " << i << ": " << dist[i]
<< "\n";
   }
   cout << "Dijkstra's Iterations: " <<
iteration_counter << "\n";
}

// Topological Sort (Using DFS)
void TopoSortUtil(vector<vector<int>>&
graph, int v, vector<bool>& visited,
stack<int>& Stack) {
   visited[v] = true;

   for (int i = 0; i < graph.size(); ++i) {
      if (graph[v][i] && !visited[i]) {
         TopoSortUtil(graph, i, visited,
Stack);
      }
      iteration_counter++;
   }

   Stack.push(v);
}

void
TopologicalSort(vector<vector<int>>&
graph, int vertices) {
   stack<int> Stack;
   vector<bool> visited(vertices, false);
   iteration_counter = 0;

   for (int i = 0; i < vertices; ++i) {
      if (!visited[i]) {
         TopoSortUtil(graph, i, visited,
Stack);
      }
   }

   cout << "Topological Order: ";
   while (!Stack.empty()) {
      cout << Stack.top() << " ";
      Stack.pop();
   }
   cout << "\nTopological Sort Iterations: "
<< iteration_counter << "\n";
}

// Floyd–Warshall
void
FloydWarshall(vector<vector<int>>&
graph, int vertices) {
   vector<vector<int>> dist = graph;
```

```
    iteration_counter = 0;

    for (int k = 0; k < vertices; ++k) {
        for (int i = 0; i < vertices; ++i) {
            for (int j = 0; j < vertices; ++j) {
                if (dist[i][k] != INT_MAX &&
dist[k][j] != INT_MAX &&
                    dist[i][k] + dist[k][j] <
dist[i][j]) {
                    dist[i][j] = dist[i][k] +
dist[k][j];
                }
                iteration_counter++;
            }
        }
    }

    cout << "Floyd–Warshall Shortest
Distance Matrix:\n";
    for (int i = 0; i < vertices; ++i) {
        for (int j = 0; j < vertices; ++j) {
            if (dist[i][j] == INT_MAX)
                cout << "INF ";
            else
                cout << dist[i][j] << " ";
        }
        cout << "\n";
    }
    cout << "Floyd–Warshall Iterations: "
<< iteration_counter << "\n";
}

int main() {
    int vertices = 6;
    vector<vector<int>> graph = {
        {0, 1, 0, 0, 0, 1},
        {1, 0, 1, 0, 0, 0},
        {0, 1, 0, 1, 1, 0},
        {0, 0, 1, 0, 1, 0},
        {0, 0, 1, 1, 0, 1},
        {1, 0, 0, 0, 1, 0}
    };

    cout << "Breadth First Search:\n";
    BFS(graph, vertices, 0);

    cout << "\nDepth First Search:\n";
    DFS(graph, vertices, 0);

    cout << "\nDijkstra's Algorithm:\n";
    Dijkstra(graph, vertices, 0);

    cout << "\nTopological Sort (Only valid
for DAGs):\n";
    TopologicalSort(graph, vertices);

    cout << "\nFloyd–Warshall
Algorithm:\n";
    // Replace 0s with INT_MAX for non-
edges
    vector<vector<int>> fw_graph(vertices,
vector<int>(vertices, INT_MAX));
    for (int i = 0; i < vertices; ++i) {
        for (int j = 0; j < vertices; ++j) {
            if (i == j) fw_graph[i][j] = 0;
            else if (graph[i][j]) fw_graph[i][j] =
graph[i][j];
        }
    }
    FloydWarshall(fw_graph, vertices);

    return 0;
}
```

Output:

```
Breadth First Search:
BFS Traversal: 0 1 5 2 4 3
BFS Iterations: 6

Depth First Search:
DFS Traversal: 0 1 2 3 4 5
DFS Iterations: 6

Dijkstra's Algorithm:
Dijkstra's Shortest Distances from Source 0:
To 0: 0
To 1: 1
To 2: 2
To 3: 3
To 4: 2
To 5: 1
Dijkstra's Iterations: 30

Topological Sort (Only valid for DAGs):
Topological Order: 0 1 2 3 4 5
Topological Sort Iterations: 36

Floyd-Warshall Algorithm:
Floyd-Warshall Shortest Distance Matrix:
0 1 2 3 2 1
1 0 1 2 2 2
2 1 0 1 1 2
3 2 1 0 1 2
2 2 1 1 0 1
1 2 2 2 1 0
Floyd-Warshall Iterations: 216
```

Fig 4.1 Graph based algorithms output

Performance Analysis:

Table 4.1 Analysis of time complexity for graph based algorithm

Algorithm	Time Complexity	Best Use Case
Breadth First Search	O(V + E)	Unweighted shortest path, layer-wise traversal
Depth First Search	O(V + E)	Cycle detection, Topological sort
Dijkstra's Algorithm	O((V + E) log V)	Shortest path in weighted graphs (non-negative)
Topological Sort	O(V + E)	Task scheduling, prerequisite resolution in DAGs
Floyd–Warshall	$O(V^3)$	All-pairs shortest paths in dense graphs

Conclusion:

- BFS and DFS are foundational graph traversal techniques used in many applications such as shortest path (BFS in unweighted graphs), cycle detection, and connected components.
- Dijkstra's algorithm is efficient for finding shortest paths from a single source in weighted graphs with non-negative weights.
- Topological sorting is useful when dealing with tasks that have dependencies, especially in scheduling problems.
- Floyd–Warshall algorithm provides an efficient way to find all-pairs shortest paths, especially when edge weights can be negative (but not forming cycles).
- The choice of algorithm depends on the graph type (weighted, unweighted, directed, cyclic) and the nature of the problem.

Lab Experiment 5:

Aim:

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

Background:

A Priority Queue is a special type of queue in which each element is associated with a priority. Elements are dequeued in order of their priority rather than the order in which they arrive. A higher-priority element is served before a lower-priority one.

There are various ways to implement a priority queue:

- Using Arrays: Elements are stored in an unsorted/sorted array. Insertion is straightforward but deletion (finding the highest-priority element) can be slow.
- Using Linked Lists: Nodes are inserted in a sorted manner according to their priority. Deletion is efficient.
- Using Heaps: The most efficient implementation for priority queues.The binary heap ensures that both insertion and deletion operations are performed in O(log n) time.A heap is a complete binary tree where the priority of the parent node is either greater than or less than that of its children, depending on whether a max-heap or min-heap is used.

Algorithm:

1. Priority Queue using Arrays

Insertion Algorithm (Array-Based):

1. Input: Element val and its priority pri.
2. Store the element and its priority at the next available index in the array.
3. Increment the size of the array.
4. End.

Deletion Algorithm (Array-Based):

1. Check if the array is empty. If yes, return an error or underflow.
2. Traverse the array to find the element with the highest priority.
3. Remove that element by shifting all the subsequent elements one position to the left.
4. Decrement the size of the array.
5. Return the deleted element.

2. Priority Queue using Linked Lists

Insertion Algorithm (Linked List-Based):

1. Input: Element val and its priority pri.
2. Create a new node containing the value and priority.
3. If the list is empty or the head node has lower priority than the new node:
    - Insert the new node at the beginning.
4. Else, traverse the list to find the correct position where the priority order is maintained.
5. Insert the new node in the correct position.
6. End.

Deletion Algorithm (Linked List-Based):

1. Check if the list is empty. If yes, return an error or underflow.
2. Remove the head node (which contains the highest priority element).
3. Update the head pointer to the next node.

4. Free the memory of the removed node.
5. Return the deleted element.

3. Priority Queue using Binary Heap
Insertion Algorithm (Heap-Based):
1. Input: Element val.
2. Insert the element at the end of the heap array.
3. Perform Heapify-Up operation:
    - Compare the inserted element with its parent.
    - If the heap property is violated (for Min-Heap: child < parent), swap them.
    - Repeat until the heap property is restored or the root is reached.
4. End.

Deletion Algorithm (Heap-Based):
1. Check if the heap is empty. If yes, return an error or underflow.
2. Remove the root element (which has the highest or lowest priority depending on heap type).
3. Replace the root with the last element in the heap.
4. Perform Heapify-Down operation:
    - Compare the new root with its children.
    - If the heap property is violated, swap with the smaller/larger child (for Min/Max Heap).
    - Repeat until the heap property is restored or a leaf node is reached.
5. Return the deleted element.

Code:

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

#define MAX 100

// Array-based Priority Queue
struct PriorityQueueArray {
    int data[MAX];
    int priority[MAX];
    int size;
};

// Linked List Node
struct Node {
    int data;
    int priority;
    Node* next;
};

// Heap-based Priority Queue
struct PriorityQueueHeap {
    int data[MAX];
    int priority[MAX];
    int size;
};

int counter = 0; // For analysis

// ---------------- Array Implementation ----------
void insertArray(PriorityQueueArray* pq, int val, int pri) {
    cout << "Inserting " << val << " with priority " << pri << " in Array\n";
    int i = pq->size;
    while (i > 0 && pq->priority[i - 1] > pri) {
        pq->data[i] = pq->data[i - 1];
        pq->priority[i] = pq->priority[i - 1];
        i--;
        counter++;
    }
    pq->data[i] = val;
    pq->priority[i] = pri;
    pq->size++;
}
```

```
void deleteArray(PriorityQueueArray* pq)
{
    if (pq->size == 0) {
        cout << "Array is empty\n";
        return;
    }
    cout << "Deleting " << pq->data[0] << "
from Array\n";
    for (int i = 0; i < pq->size - 1; i++) {
        pq->data[i] = pq->data[i + 1];
        pq->priority[i] = pq->priority[i + 1];
        counter++;
    }
    pq->size--;
}

// ---------------- Linked List
Implementation ----------------
void insertLinkedList(Node** head, int
val, int pri) {
    cout << "Inserting " << val << " with
priority " << pri << " in Linked List\n";
    Node* newNode = new Node{val, pri,
nullptr};
    if (*head == nullptr || (*head)->priority
> pri) {
        newNode->next = *head;
        *head = newNode;
        return;
    }
    Node* temp = *head;
    while (temp->next != nullptr && temp-
>next->priority <= pri) {
        temp = temp->next;
        counter++;
    }
    newNode->next = temp->next;
    temp->next = newNode;
}

void deleteLinkedList(Node** head) {
    if (*head == nullptr) {
        cout << "Linked List is empty\n";
        return;
    }
    cout << "Deleting " << (*head)->data
<< " from Linked List\n";
    Node* temp = *head;
    *head = (*head)->next;
    delete temp;
    counter++;
}

// ---------------- Heap Implementation ------
----------
void heapify(PriorityQueueHeap* pq, int i)
{
    int smallest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < pq->size && pq->priority[left]
< pq->priority[smallest])
        smallest = left;
    if (right < pq->size && pq-
>priority[right] < pq->priority[smallest])
        smallest = right;

    if (smallest != i) {
        swap(pq->data[i], pq-
>data[smallest]);
        swap(pq->priority[i], pq-
>priority[smallest]);
        counter++;
        heapify(pq, smallest);
    }
}

void insertHeap(PriorityQueueHeap* pq,
int val, int pri) {
    cout << "Inserting " << val << " with
priority " << pri << " in Heap\n";
    int i = pq->size++;
    pq->data[i] = val;
    pq->priority[i] = pri;
    while (i > 0 && pq->priority[(i - 1) / 2]
> pq->priority[i]) {
        swap(pq->data[i], pq->data[(i - 1) /
2]);
        swap(pq->priority[i], pq->priority[(i -
1) / 2]);
        i = (i - 1) / 2;
        counter++;
    }
}

void deleteHeap(PriorityQueueHeap* pq)
{
    if (pq->size == 0) {
```

```
        cout << "Heap is empty\n";
        return;
    }
    cout << "Deleting " << pq->data[0] << " from Heap\n";
    pq->data[0] = pq->data[--pq->size];
    pq->priority[0] = pq->priority[pq->size];
    heapify(pq, 0);
}

// ---------------- Main Function ----------------
int main() {
    PriorityQueueArray arrPQ = {.size = 0};
    Node* llPQ = nullptr;
    PriorityQueueHeap heapPQ = {.size = 0};

    // Array Implementation
    insertArray(&arrPQ, 15, 2);
    insertArray(&arrPQ, 25, 1);
    insertArray(&arrPQ, 35, 3);
    deleteArray(&arrPQ);
    cout << "Array Priority Queue iterations: " << counter << "\n";
    counter = 0;

    // Linked List Implementation
    insertLinkedList(&llPQ, 15, 2);
    insertLinkedList(&llPQ, 25, 1);
    insertLinkedList(&llPQ, 35, 3);
    deleteLinkedList(&llPQ);
    cout << "Linked List Priority Queue iterations: " << counter << "\n";
    counter = 0;

    // Heap Implementation
    insertHeap(&heapPQ, 15, 2);
    insertHeap(&heapPQ, 25, 1);
    insertHeap(&heapPQ, 35, 3);
    deleteHeap(&heapPQ);
    cout << "Heap Priority Queue iterations: " << counter << "\n";

    return 0;
}
```

Output:

```
Inserting 15 with priority 2 in Array
Inserting 25 with priority 1 in Array
Inserting 35 with priority 3 in Array
Deleting 25 from Array
Array Priority Queue iterations: 3
Inserting 15 with priority 2 in Linked List
Inserting 25 with priority 1 in Linked List
Inserting 35 with priority 3 in Linked List
Deleting 25 from Linked List
Linked List Priority Queue iterations: 2
Inserting 15 with priority 2 in Heap
Inserting 25 with priority 1 in Heap
Inserting 35 with priority 3 in Heap
Deleting 25 from Heap
Heap Priority Queue iterations: 2
```

Fig 5.1 Priority Queue implementation output.

Performance Analysis:

Table 5.1 Analysis of time complexity for Priority Queue

Operation	Array-based (Unsorted)	Array-based (Sorted)	Linked List-based	Heap-based
Insertion	O(1)	O(n)	O(n)	O(log n)
Deletion (max/min)	O(n)	O(1)	O(1)	O(log n)
Access to max/min	O(n)	O(1)	O(1)	O(1)
Memory Usage	Efficient	Less efficient	Less efficient	Efficient

Conclusion:

- **Array-based Priority Queue:** Best for simple applications but inefficient for large data sets, especially when sorting is involved.
- **Linked List-based Priority Queue:** Offers more dynamic memory management but suffers from slower insertion times.
- **Heap-based Priority Queue:** Most efficient for performance, especially when operations like insertion and deletion need to be fast. This is the preferred choice for implementing priority queues in most scenarios.

Lab Experiment 6:

Aim:

Implementation and analysis of Skip Lists and comparison with basic data structures.

Background:

A Skip List is a probabilistic data structure that allows fast search, insertion, and deletion operations - similar to balanced trees like AVL or Red-Black trees. It uses multiple levels of linked lists where higher levels act as "express lanes" to skip over many elements, allowing logarithmic average time complexity. The basic idea is to augment a sorted linked list with additional layers. Each layer is a subset of the layer below, where each element appears in a higher layer with some fixed probability. This randomness gives the Skip List its probabilistic balancing.

Key Characteristics:

- Skip lists offer average-case time complexities of O(log n) for search, insert, and delete.
- The structure is easier to implement and maintain than balanced trees.
- Space complexity is O(n log n) in the worst case due to multiple levels.

Algorithm:

1. Search (Locate a Key in Skip List)
    1. Initialize a pointer at the top-most level of the header node.
    2. While at the current level:
        - Traverse forward as long as the next node's key is less than the target key.
        - If the next key is greater or null, move one level down.
    3. Continue the process until you either:
        - Find a node with the target key - return success.
        - Reach level 0 and the key is not found - return failure.

2. Insertion (Insert a New Element)
    1. Initialize a path tracker (update[]) to store the last visited node at each level.
    2. Starting from the highest level of the header:
        - Traverse forward until the next key is greater than or equal to the target key.
        - Record the last visited node at each level before moving down.
    3. At level 0, insert the new key after the appropriate node.
    4. Randomly generate a level for the new node using a probabilistic method.
    5. For each level up to the generated level:
        - Create forward pointers and adjust the update[] nodes to link to the new node.
    6. If the new node exceeds the current maximum level, update the list's level accordingly.

3. Deletion (Remove an Element by Key)
    1. Use a similar traversal process as in insertion to track the update[] array.
    2. At level 0, check if the key is present in the list.
        - If not found, return failure.

3. If the node exists:
    - Unlink the node at every level it appears in by updating the forward pointers.
    - Deallocate the node's memory.
4. After deletion, check if the highest level in the skip list has become empty.
    - If yes, reduce the skip list level accordingly.

Code:

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <climits>
#include <vector>

#define MAX_LEVEL 16

class Node {
public:
    int key, value;
    std::vector<Node*> forward;

    Node(int k, int v, int level) : key(k),
value(v), forward(level + 1, nullptr) {}
};

class SkipList {
private:
    int level;
    Node* header;

    int randomLevel() {
        int lvl = 0;
        while (rand() < RAND_MAX / 2 &&
lvl < MAX_LEVEL) lvl++;
        return lvl;
    }

public:
    SkipList() {
        level = 0;
        header = new Node(INT_MIN, 0,
MAX_LEVEL);
    }

    ~SkipList() {
        Node* curr = header->forward[0];
        while (curr) {
            Node* next = curr->forward[0];
            delete curr;
            curr = next;
        }
        delete header;
    }

    void insert(int key, int value) {
        std::vector<Node*>
update(MAX_LEVEL + 1);
        Node* curr = header;

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

        curr = curr->forward[0];

        if (!curr || curr->key != key) {
            int newLevel = randomLevel();

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

            Node* newNode = new Node(key,
value, newLevel);
            for (int i = 0; i <= newLevel; i++) {
                newNode->forward[i] =
update[i]->forward[i];
                update[i]->forward[i] =
newNode;
            }
        } else {
            curr->value = value;
        }
```

```
    }

    Node* search(int key) {
        Node* curr = header;

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

        if (curr && curr->key == key) return curr;
        return nullptr;
    }

    void remove(int key) {
        std::vector<Node*> update(MAX_LEVEL + 1);
        Node* curr = header;

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

        curr = curr->forward[0];

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

            while (level > 0 && !header->forward[level]) level--;
        }
    }

    void display() {
        std::cout << "Skip List (Level " << level << "):\n";
        for (int i = 0; i <= level; i++) {
            Node* node = header->forward[i];
            std::cout << "Level " << i << ": ";
            while (node) {
                std::cout << node->key << " -> ";
                node = node->forward[i];
            }
            std::cout << "NULL\n";
        }
    }
};

int main() {
    srand(time(0));

    SkipList list;
    list.insert(3, 30);
    list.insert(6, 60);
    list.insert(7, 70);
    list.insert(9, 90);
    list.insert(12, 120);
    list.insert(19, 190);
    list.insert(17, 170);
    list.insert(26, 260);
    list.insert(21, 210);
    list.insert(25, 250);

    list.display();

    int key = 12;
    Node* result = list.search(key);
    if (result) {
        std::cout << "Found key " << key << " with value " << result->value << "\n";
    } else {
        std::cout << "Key " << key << " not found\n";
    }

    std::cout << "Deleting key 21\n";
    list.remove(26);
    list.display();

    return 0;
}
```

Output:

```
Skip List (Level 4):
Level 0: 3 -> 6 -> 7 -> 9 -> 12 -> 17 -> 19 -> 21 -> 25 -> 26 -> NULL
Level 1: 3 -> 7 -> 9 -> 12 -> 17 -> 26 -> NULL
Level 2: 3 -> 9 -> 26 -> NULL
Level 3: 3 -> 9 -> 26 -> NULL
Level 4: 3 -> NULL
Found key 12 with value 1200
Deleting key 26
Skip List (Level 4):
Level 0: 3 -> 6 -> 7 -> 9 -> 12 -> 17 -> 19 -> 21 -> 25 -> NULL
Level 1: 3 -> 7 -> 9 -> 12 -> 17 -> NULL
Level 2: 3 -> 9 -> NULL
Level 3: 3 -> 9 -> NULL
Level 4: 3 -> NULL
```

Fig 6.1 Skip List implementation

Performance Analysis:

Table 6.1 Analysis of time complexity for Skip List

Operation	Average Case	Worst Case	Space Usage
Search	O(log n)	O(n)	O(n log n)
Insert	O(log n)	O(n)	O(n log n)
Delete	O(log n)	O(n)	O(n log n)

Conclusion:

- Skip Lists are an efficient alternative to balanced trees for ordered data structures.
- With logarithmic expected time complexity, they provide fast search, insertion, and deletion.
- Their probabilistic nature simplifies implementation compared to trees requiring strict balancing.
- In many practical applications, Skip Lists match or exceed performance of traditional balanced trees.
- They are especially useful in concurrent and distributed systems like databases and memory indexing.

Lab Experiment 7:

Aim:

Implementation and analysis of Splay Trees and comparison with basic data structures.

Background:

Splay Trees are a type of self-adjusting binary search tree where recently accessed elements are moved to the root through a process called **splaying**. This ensures that frequently accessed elements are quick to access again, improving performance in real-world scenarios with non-uniform access patterns.

Each operation (search, insertion, deletion) involves a **splay** step to bring the accessed node to the root. This adaptive nature can lead to amortized time complexities of **O(log n)**.

Splay Trees are often compared to:

- Binary Search Trees (BST)
- AVL Trees
- Red-Black Trees because of their dynamic balancing characteristics.

Algorithm:

1. Splay Operation (Core Mechanism)
Input: Node x to be accessed
Output: Node x becomes the new root
Steps:

1. While x is not the root:
    - Let p be the parent of x and g the grandparent of x (if it exists).
    - Depending on the relationship between x, p, and g, perform one of the following:
        - **Zig** (single rotation): x is child of root (p is root).
        - **Zig-Zig** (double rotation): x and p are both left or both right children.
        - **Zig-Zag** (double rotation): x is a left child and p is a right child, or vice versa.
2. Continue until x is the root.

2. Search Operation
Input: Key k
Output: Node with key k, or NULL if not found.
Steps:

1. Traverse the tree as in a standard BST search.
2. If node with key k is found or a leaf is reached:
    - **Splay** the last accessed node to the root.
3. Return the result.

3. Insert Operation

Input: Key k, Value v
Output: Updated tree with new node inserted.
Steps:

1. If tree is empty, insert the node as root.
2. Else:
    - Traverse as in a BST to find the insertion point.
    - Insert the new node as a leaf.
    - **Splay** the newly inserted node to the root.

4. Delete Operation
Input: Key k
Output: Updated tree with node removed.
Steps:

1. Search for the node with key k.
2. If found:
    - **Splay** the node to the root.
    - If the left and right subtrees exist:
        - Save a reference to the right subtree.
        - Remove the root.
        - Splay the maximum node of the left subtree to the root.
        - Attach the saved right subtree as the new root's right child.
    - If only one subtree exists, make it the new root.
3. If node not found, exit.

Code:

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

// Node structure
struct Node {
    int key;
    Node* left;
    Node* right;
    Node(int k) : key(k), left(nullptr),
right(nullptr) {}
};

// Right rotate
Node* rightRotate(Node* x) {
    Node* y = x->left;
    x->left = y->right;
    y->right = x;
    return y;
}

// Left rotate
Node* leftRotate(Node* x) {
    Node* y = x->right;
    x->right = y->left;
    y->left = x;
    return y;
}

// Splay function
Node* splay(Node* root, int key) {
    if (!root || root->key == key)
        return root;

    // Left subtree
    if (key < root->key) {
        if (!root->left) return root;

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

```
            root->left->right = splay(root->left->right, key);
            if (root->left->right)
                root->left = leftRotate(root->left);
        }

        return root->left ? rightRotate(root) : root;
    }
    // Right subtree
    else {
        if (!root->right) return root;

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

        return root->right ? leftRotate(root) : root;
    }
}

// Insert function
Node* insert(Node* root, int key) {
    if (!root) return new Node(key);

    root = splay(root, key);
    if (root->key == key) return root;  // Duplicate not inserted

    Node* newNode = new Node(key);
    if (key < root->key) {
        newNode->right = root;
        newNode->left = root->left;
        root->left = nullptr;
    } else {
        newNode->left = root;
        newNode->right = root->right;
        root->right = nullptr;
    }
    return newNode;
}

// Delete function
Node* deleteKey(Node* root, int key) {
    if (!root) return nullptr;

    root = splay(root, key);
    if (root->key != key) return root;  // Key not found

    Node* temp;
    if (!root->left) {
        temp = root->right;
    } else {
        temp = splay(root->left, key);
        temp->right = root->right;
    }
    delete root;
    return temp;
}

// Inorder traversal
void inorder(Node* root) {
    if (!root) return;
    inorder(root->left);
    cout << root->key << " ";
    inorder(root->right);
}

// Structured test
int main() {
    Node* root = nullptr;

    cout << "\n--- Splay Tree Operations ---\n";
    cout << "\nInserting elements: 10, 20, 30, 40, 50, 25\n";
    int keys[] = {10, 20, 30, 40, 50, 25};
    for (int key : keys) {
        root = insert(root, key);
        cout << "Inserted: " << key << "\n";
    }

    cout << "\nInorder traversal after insertions:\n";
    inorder(root);
    cout << "\n";
```

```
    cout << "\nSearching for key 30:\n";
    root = splay(root, 30);
    cout << "Root after splay: " << root->key << "\n";

    cout << "\nDeleting key 20:\n";
    root = deleteKey(root, 20);
    inorder(root);
    cout << "\n";

    cout << "\nDeleting key 10:\n";
    root = deleteKey(root, 10);
    inorder(root);
    cout << "\n";

    return 0;
}
```

Output:

```
--- Splay Tree Operations ---

Inserting elements: 10, 20, 30, 40, 50, 25
Inserted: 50
Inserted: 60
Inserted: 70
Inserted: 80
Inserted: 90
Inserted: 250

Inorder traversal after insertions:
50 60 70 80 90 250

Searching for key 30:
Root after splay: 50

Deleting key 20:
50 60 70 80 90 250

Deleting key 10:
50 60 70 80 90 250
```

Fig 7.1 Splay Tree implementation output.

Performance Analysis:

Table 7.1 Analysis of time complexity for Splay Tree.

Operation	Average Case	Worst Case
Search	O(log n)	O(n)
Insert	O(log n)	O(n)
Delete	O(log n)	O(n)

Conclusion:

Splay Trees provide an efficient and elegant solution for dynamic set operations where frequent access patterns are skewed. Though the worst-case time for individual operations is linear, their amortized complexity remains logarithmic, making them practical in many scenarios. Compared to strictly balanced trees like AVL or Red-Black Trees, Splay Trees often offer better real-world performance without explicit balancing logic.

Lab Experiment No. 8:

Aim:
Implementation and analysis of Dynamic Arrays and its operations.

- Insertion
- Deletion
- Searching
- Traversing
- Updation

Background:

A dynamic array is a data structure that allows elements to be added or removed and resizes itself automatically when capacity is exceeded. Unlike static arrays with fixed size, dynamic arrays grow or shrink as needed. They provide random access like arrays and are useful when the number of elements is not known in advance.
Operations typically supported by dynamic arrays:

- Insertion: Adding a new element, possibly resizing the array.
- Deletion: Removing an element and shifting others.
- Searching: Finding an element by value or index.
- Traversing: Visiting all elements.
- Updation: Changing the value at a specific index.

Algorithm:

1. Insertion
    1. Check if the current size equals capacity.
    2. If yes, double the capacity and reallocate memory.
    3. Insert the new element at the current size index.
    4. Increment the size.
2. Deletion
    1. Check if index is valid.
    2. Shift all elements after the index to the left by one.
    3. Decrement the size.
    4. Optionally, shrink the capacity if size is too small.
3. Searching
    1. Iterate over the array.
    2. If the element matches the target, return the index.
    3. If not found, return -1.
4. Traversing
    1. Loop from 0 to size-1.
    2. Print each element.
5. Updation
    1. Check if index is valid.
    2. Replace the element at the index with the new value.

Code:

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

class DynamicArray {
private:
    int* arr;
    int size;
    int capacity;

    void resize(int newCapacity) {
        int* temp = new
int[newCapacity];
        for (int i = 0; i < size; ++i)
            temp[i] = arr[i];
        delete[] arr;
        arr = temp;
        capacity = newCapacity;
    }

public:
    DynamicArray() {
        size = 0;
        capacity = 2;
        arr = new int[capacity];
    }

    void insert(int value) {
        if (size == capacity)
            resize(capacity * 2);
        arr[size++] = value;
        cout << "Inserted: " << value
<< "\n";
    }

    void deleteAt(int index) {
        if (index < 0 || index >= size)
{
            cout << "Invalid index\n";
            return;
        }
        cout << "Deleted: " <<
arr[index] << "\n";
        for (int i = index; i < size - 1;
++i)
            arr[i] = arr[i + 1];
        size--;
        if (size < capacity / 2 &&
capacity > 2)
            resize(capacity / 2);
    }

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

    void update(int index, int value)
{
        if (index < 0 || index >= size)
{
            cout << "Invalid index\n";
            return;
        }
        cout << "Updated index " <<
index << " from " << arr[index] <<
" to " << value << "\n";
        arr[index] = value;
    }

    void traverse() {
        cout << "Array Elements: ";
        for (int i = 0; i < size; ++i)
            cout << arr[i] << " ";
        cout << "\n";
    }

    ~DynamicArray() {
        delete[] arr;
    }
};

int main() {
    DynamicArray da;

    da.insert(10);
    da.insert(20);
    da.insert(30);
    da.traverse();

    da.deleteAt(1);
```

```
    da.traverse();

    int index = da.search(30);
    if (index != -1)
        cout << "Element 30 found at
index " << index << "\n";
    else
        cout << "Element 30 not
found\n";

    da.update(0, 99);
    da.traverse();

    return 0;
}
```

Output:

```
Inserted: 67
Inserted: 97
Inserted: 134
Array Elements: 67 97 134
Deleted: 97
Array Elements: 67 134
Element 30 not found
Updated index 0 from 67 to 99
Array Elements: 99 134
```

Fig. 8.1 Dynamic Array Implementation Output.

Performance Analysis:

Table 8.1 Analysis of time complexity for Dynamic Array.

Operation	**Best Case**	**Worst Case**	**Average Case**
Insertion	O(1) (at end, no resize)	O(n) (with shifting/resizing)	O(n) (in middle or beginning)
Deletion	O(1) (from end)	O(n) (with shifting)	O(n) (from middle or beginning)
Searching	O(1) (if element at known index)	O(n) (linear search)	O(n) (not found)
Traversal	O(n)	O(n)	O(n)

Conclusion:

Dynamic Arrays are a flexible alternative to static arrays with the ability to resize dynamically. They support efficient insertion and updates, although deletion and searching remain linear-time operations. This experiment demonstrates core dynamic array operations and their performance characteristics compared to basic static structures.

Lab Experiment No. 9:

Aim:
Implementation and analysis of Hashing techniques.

Background:

Hashing is a technique used for fast data retrieval. It involves mapping data to a hash table using a hash function. The primary objective of hashing is to optimize search, insertion, and deletion operations by achieving constant average time complexity, O(1). However, collisions can occur when multiple keys map to the same index. To handle such cases, various collision resolution techniques are used such as:

- **Open Addressing** (Linear Probing, Quadratic Probing, Double Hashing).
- **Separate Chaining.**

Algorithm:

1. Hash Function
    1. Input: Key k, Table size m
    2. Compute: index = k % m
    3. Output: Index position in the hash table

2. Insertion (Using Separate Chaining)
    1. Input: Key k, Value v
    2. Compute hash index using the hash function.
    3. If the bucket at that index is empty, create a new linked list and insert the key-value pair.
    4. If a list already exists, append the key-value pair to the list (or update if key exists).
    5. End.

3. Search
    1. Input: Key k
    2. Compute the hash index using the hash function.
    3. Traverse the linked list at that index.
    4. If the key is found, return the value.
    5. Else, return “Key not found”.
    6. End.

4. Deletion
    1. Input: Key k
    2. Compute the hash index using the hash function.
    3. Traverse the linked list at that index.
    4. If the key is found, delete the corresponding node.
    5. If not found, return "Key not found".
    6. End.

5. Traversal
    1. For each index i from 0 to m-1:
        - Print index i.
        - If a list exists at that index, print all key-value pairs.
    2. End.

6. Updation
    1. Input: Key k, New value v
    2. Compute the hash index using the hash function.
    3. Traverse the list at that index to find the key.
    4. If found, update the value to v.
    5. Else, return "Key not found".
    6. End.

Code:

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

const int TABLE_SIZE = 10;

class HashTable {
private:
   list<pair<int, string>>
table[TABLE_SIZE];

   int hashFunction(int key) {
      return key % TABLE_SIZE;
   }

public:
   void insert(int key, const string& value)
{
      cout << "Insert " << key << endl;
      int index = hashFunction(key);
      table[index].push_back({key,
value});
   }

   void remove(int key) {
      cout << "Delete operation\n";
      int index = hashFunction(key);
      for (auto it = table[index].begin(); it
!= table[index].end(); ++it) {
         if (it->first == key) {
            table[index].erase(it);
            return;
         }
      }
   }

   void search(int key) {
      cout << "Search operation\n";
      int index = hashFunction(key);
      for (auto& element : table[index]) {
         if (element.first == key) {
            cout << "Found " << key << ": "
<< element.second << endl;
            return;
         }
      }
      cout << "Key not found.\n";
   }

   void display() {
      cout << "Display operation\n";
      bool hasElements = false;
      for (int i = 0; i < TABLE_SIZE; i++)
{
         if (!table[i].empty()) {
            hasElements = true;
            cout << "Index " << i << ": ";
            for (auto& pair : table[i]) {
               cout << pair.first << "->" <<
pair.second << " ";
            }
            cout << endl;
         }
      }
      if (!hasElements) {
         cout << "Hash Table is empty.\n";
      }
   }
```

```
};

int main() {
    HashTable ht;

    ht.insert(1, "Apple");
    ht.insert(11, "Banana");
    ht.insert(21, "Mango");

    ht.display();

    ht.search(11);
    ht.remove(11);
    ht.search(11);

    ht.display();

    return 0;
}
```

Output:

```
Insert 1
Insert 11
Insert 21
Display operation
Index 1: 1->Yash  11->Arora 21->Captain
Search operation
Found 11: Arora
Delete operation
Search operation
Key not found.
Display operation
Index 1: 1->Yash  21->Captain
```

Fig 9.1 Hashing Operation Implementation Output.

Performance Analysis:

Table 9.1 Analysis of time complexity for Hashing Operation.

OPERATION	AVERAGE CASE	WORST CASE
Insertion	O(1)	O(n)
Search	O(1)	O(n)
Deletion	O(1)	O(n)

Conclusion:

- Hashing provides efficient data access, with average-case constant time for insertion, deletion, and search operations.
- Collision resolution using separate chaining effectively handles cases where multiple keys map to the same index.
- Performance heavily depends on the quality of the hash function - a good hash function ensures uniform distribution and minimizes collisions.
- Table size selection is crucial - choosing a prime number and maintaining a suitable load factor helps in maintaining optimal performance.
- The experiment demonstrates that hashing outperforms linear data structures (like arrays and linked lists) for search-based operations, especially in large datasets.