Lab Experiment No. 5

Objective

To implement and analyze Priority Queues using arrays, linked list and heaps.

Theory

A **priority queue** is an abstract data type where each element is associated with a **priority**, and elements are served based on **priority order**, not just insertion order.

- **Higher priority elements** are dequeued **before** lower priority ones.
- If priorities are the same, it can be FIFO or based on tie-breaking rules.

T-5.1. Implementation using Arrays

Working Principle

- Elements are stored in an array along with their priorities.
- During insertion, append the element.
- During deletion, **search for the element with highest priority** and remove it.

Operations

Operation	Description	Time Complexity
Insert	Add to the end of the array	O(1)
Pop	Linear scan for highest priority item	O(n)
Top	Linear scan to find top priority	O(n)

Pros

- Easy to implement
- Good for small data sizes

Cons

- Inefficient for large datasets (pop/top = O(n))

T-5.2. Implementation using Linked List

Working Principle

- Maintain a **sorted linked list** based on priority.
- Insertions ensure the list is sorted by priority (descending).
- The **head node** always contains the highest priority.

Operations

Operation	Description	Time Complexity
Insert	Traverse list to insert in order	O(n)
Pop	Remove head	O(1)
Top	Return head	O(1)

Pros

- Efficient removal (O(1))
- Better than arrays for frequent pop/top operations

Cons

- Slower insertions (O(n))
- Requires extra memory for pointers

T-5.3. Implementation using Binary Heaps

Working Principle

- Use a **binary heap** (min-heap or max-heap) to maintain priority.
- In a **max-heap**, parent nodes are always greater than or equal to children.

Operations

Operation	Description	Time Complexity
Insert	Add at end, heapify up	O(log n)
Pop	Remove root, heapify down	O(log n)
Top	Return root	O(1)

Pros

- Efficient for all operations
- Well-suited for dynamic datasets and large-scale applications

Cons

- Slightly more complex to implement
- Cannot search arbitrary elements in less than O(n).

Code

T-5.1. Implementation using Arrays. Depth-First Search (DFS)

```
#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++;
    counter++;
}

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;
        counter++;
        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->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;
}
```

```
    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; }

Sample 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

Complexity 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

In this lab, we implemented and analyzed Priority Queues using arrays, linked lists and heaps.