Design Documentation: Max-Heap Priority Queue in C++

Output:

```
50 Success: 50 has been added to the Priority Queue.
Priority Queue elements in descending order: 50
30 Success: 30 has been added to the Priority Queue.
Priority Queue elements in descending order: 50 30
10 Success: 10 has been added to the Priority Queue.
Priority Queue elements in descending order: 50 30 10
40 Success: 40 has been added to the Priority Queue.
Priority Queue elements in descending order: 50 40 30 10
20 Success: 20 has been added to the Priority Queue.
Priority Queue elements in descending order: 50 40 30 20 10
100 Success: 100 has been added to the Priority Queue.
Priority Queue elements in descending order: 100 50 40 30 20 10
70 Success: 70 has been added to the Priority Queue.
Priority Queue elements in descending order: 100 70 50 40 30 20 10
90 Success: 90 has been added to the Priority Queue.
Priority Queue elements in descending order: 100 90 70 50 40 30 20 10
60 Success: 60 has been added to the Priority Queue.
Priority Queue elements in descending order: 100 90 70 60 50 40 30 20 10
80 Success: 80 has been added to the Priority Queue.
Priority Queue elements in descending order: 100 90 80 70 60 50 40 30 20 10

10 elements in priority Q
Priority Q not empty.
Priority Queue elements in descending order: 100 90 80 70 60 50 40 30 20 10
Top element: 100
Popping
Success: Top element has been popped from the Priority Queue.
Priority Q not empty.
Priority Queue elements in descending order: 90 80 70 60 50 40 30 20 10
Top element: 90
Popping
Success: Top element has been popped from the Priority Queue.
Priority Q not empty.
Priority Queue elements in descending order: 80 70 60 50 40 30 20 10
Top element: 80
Popping
Success: Top element has been popped from the Priority Queue.
Priority Q not empty.
Priority Queue elements in descending order: 70 60 50 40 30 20 10
Top element: 70
Popping
Success: Top element has been popped from the Priority Queue.
Priority Q not empty.
Priority Queue elements in descending order: 60 50 40 30 20 10
Top element: 60
Popping
Success: Top element has been popped from the Priority Queue.
Priority Q not empty.
Priority Queue elements in descending order: 50 40 30 20 10
Top element: 50
Popping
Success: Top element has been popped from the Priority Queue.
Priority Q not empty.
Priority Queue elements in descending order: 40 30 20 10
Top element: 40
Popping
Success: Top element has been popped from the Priority Queue.
Priority Q not empty.
Priority Queue elements in descending order: 30 20 10
Top element: 30
Popping
Success: Top element has been popped from the Priority Queue.
Priority Q not empty.
Priority Queue elements in descending order: 20 10
Top element: 20
Popping
Success: Top element has been popped from the Priority Queue.
Priority Q not empty.
Priority Queue elements in descending order: 10
Top element: 10
Popping
Success: Top element has been popped from the Priority Queue.
Priority Q empty.
Top element anyway: Error: Priority Queue is empty.
-1
```

1. Introduction

This document outlines the design and implementation of a priority queue using a max-heap data structure in C++. The priority queue supports basic operations like insertion, removal of the maximum element, checking if the queue is empty, and printing its contents. The design ensures efficient operations with time complexities optimized for a heap-based priority queue.

---

2. System Overview

A **priority queue** is a data structure where each element has a priority associated with it. Elements with higher priority are dequeued before those with lower priority. In this implementation, a **max-heap** is used, which ensures that the highest-priority element (the maximum value) is always at the root of the heap.

The priority queue is implemented using a dynamic array (std::vector<int>) to store the heap elements and provides the following key functionalities:

- Inserting an element into the queue.
- Removing the maximum element (root).
- Printing all elements in descending order.
- Checking if the queue is empty.
- Retrieving the maximum element.

---

3. Class Design

Class: PriorityQ

The PriorityQ class encapsulates all operations and maintains the heap property. Below is the detailed breakdown of its members and methods:

Private Members:

- std::vector<int> heap: The dynamic array (vector) that holds the elements of the priority queue, arranged according to the max-heap property.

Private Helper Methods:

1. **int parent(int i)**:
    - Returns the index of the parent of the node at index i.

- Formula: (i - 1) / 2

2. **int leftChild(int i)**:
   - Returns the index of the left child of the node at index i.
   - Formula: 2 * i + 1
3. **int rightChild(int i)**:
   - Returns the index of the right child of the node at index i.
   - Formula: 2 * i + 2
4. **void heapifyUp(int i)**:
   - Recursively maintains the max-heap property by moving the node at index i upward if it violates the heap property (i.e., it is larger than its parent).
   - Time Complexity: O(log n)
5. **void heapifyDown(int i)**:
   - Recursively restores the max-heap property by moving the node at index i downward if it's smaller than its children.
   - Time Complexity: O(log n)

Public Methods:

1. **bool empty() const**:
   - Returns true if the heap is empty, otherwise false.
   - Time Complexity: O(1)
2. **int size() const**:
   - Returns the number of elements in the priority queue.
   - Time Complexity: O(1)
3. **int top()**:
   - Returns the maximum element (root of the heap). Throws an exception if the priority queue is empty.
   - Time Complexity: O(1)
4. **void push(int value)**:
   - Inserts a new element value into the heap, maintaining the max-heap property.
   - Steps:

        a. Add the new element at the end of the heap.
        b. Call heapifyUp() to maintain the heap property.
    - Time Complexity: O(log n)
5. **void pop()**:
    - Removes the maximum element (root) from the heap and re-heapifies.
    - Steps:
        a. Replace the root with the last element.
        b. Remove the last element.
        c. Call heapifyDown() to restore the max-heap property.
    - Time Complexity: O(log n)
6. **void print()**:
    - Prints the elements of the priority queue in descending order. To achieve this without modifying the original heap, the function creates a copy of the heap and removes elements one by one, re-heapifying as needed.
    - Time Complexity: O(n log n) (due to repeatedly heapifying as elements are removed)

---

4. Key Design Considerations

- **Dynamic Memory Allocation:**
    - The std::vector<int> is used to store the elements dynamically, ensuring flexibility in managing memory as the queue grows or shrinks.
- **Exception Handling:**
    - Proper exception handling is in place to handle edge cases like popping or accessing the top element from an empty queue (std::underflow_error).
- **Heapify Operations:**
    - Efficient reordering of elements during insertion (heapifyUp) and deletion (heapifyDown) ensures that the heap property is maintained. Both operations have a time complexity of O(log n), keeping the priority queue operations fast and efficient.
- **Print Function Optimization:**
    - The print() function creates a copy of the heap to avoid modifying the original heap. This allows safe inspection of the contents in sorted order. However, this results in a

time complexity of O(n log n), which could be further optimized depending on application needs.

---

5. Complexity Analysis

Operation	Time Complexity
Insert element (push)	O(log n)
Get the top element (top)	O(1)
Remove the top element (pop)	O(log n)
Check if empty (empty)	O(1)
Get size (size)	O(1)
Print all elements (print)	O(n log n)
Space Complexity	O(n)

- **Insert/Pop Time Complexity:** The heapify process during insertion and removal guarantees O(log n) performance, where n is the number of elements in the priority queue.
- **Top Time Complexity:** Accessing the top element takes constant time since the maximum element is always at the root of the heap.
- **Space Complexity:** The priority queue requires O(n) space where n is the number of elements stored in the heap.

---

6. Use Cases and Limitations

Use Cases:

- This priority queue implementation is suitable for scenarios where frequent insertion and deletion of the highest-priority element is needed, such as:
    - Job scheduling (e.g., highest-priority task execution).
    - Pathfinding algorithms (e.g., Dijkstra's or A* algorithms).
    - Event-driven simulations where events need to be processed based on their priority.

Limitations:

- The current print() method is not optimal for very large heaps as it creates a copy of the heap, leading to higher memory and computational overhead.
- This priority queue assumes that the higher the integer value, the higher its priority. For non-integer or custom objects, further modifications would be needed.

---

7. Future Enhancements

- **Custom Comparator:** Introduce support for custom comparators, enabling users to define priority based on criteria other than numeric value.
- **Performance Optimizations:** Further optimize the print() method to avoid copying the heap unnecessarily.
- **Thread Safety:** Add thread-safety mechanisms if the priority queue is to be used in a multi-threaded environment.
- **Dynamic Heap Size Visualization:** Provide a method for visualizing the heap structure during execution for debugging and educational purposes.

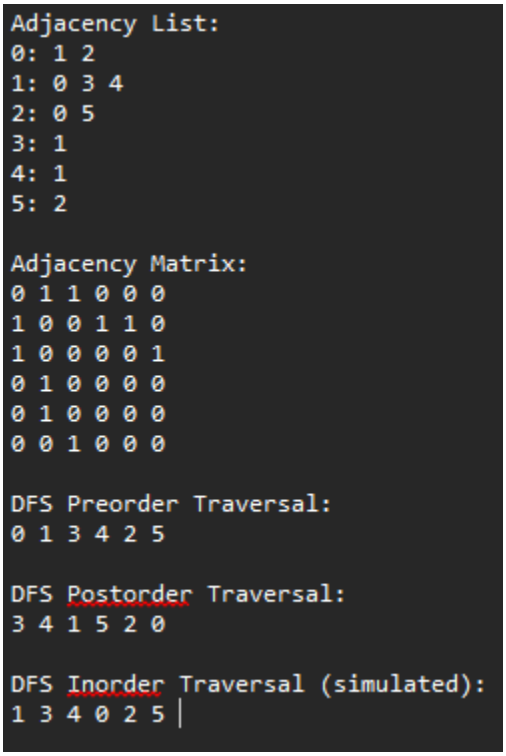
---

8. Conclusion

The PriorityQ class provides an efficient and flexible implementation of a max-heap priority queue. Its well-defined methods and exception handling make it robust for various use cases that require priority-based element access. Future enhancements can improve its performance and usability further.

Design Documentation for Graph DFS Traversals

Output:



1. Overview

This program implements a graph data structure that supports adjacency list and adjacency matrix representations. It also provides Depth-First Search (DFS) traversals: Preorder, Postorder, and a simulated Inorder traversal, applicable for general graphs. The graph is undirected, and the traversal results are printed to the console.

---

2. Classes and Methods

Class: Graph

This class represents the graph and contains the following key components:

- **Private Members:**
    - numNodes: Stores the number of nodes in the graph.
    - adjList: The adjacency list representation of the graph.
    - adjMatrix: The adjacency matrix representation of the graph.
    - nodes: A vector containing the nodes (represented by integers for simplicity).
- **Public Methods:**

i. **Graph(vector<int> nodes)**

- **Description**: Constructor that initializes the graph with n nodes and resizes both adjacency list and matrix.
- **Parameters**:
  - nodes: Vector of integers representing nodes.

ii. **void addEdge(int u, int v)**

- **Description**: Adds an undirected edge between two nodes u and v by updating both the adjacency list and matrix.
- **Time Complexity**: O(1) for both list and matrix updates.
- **Parameters**:
  - u: The first node.
  - v: The second node.

iii. **int getNodeIndex(int node)**

- **Description**: Returns the index of a node in the nodes vector.
- **Time Complexity**: O(V) where V is the number of nodes.

iv. **void printAdjList()**

- **Description**: Prints the adjacency list representation of the graph.
- **Time Complexity**: O(V + E), where V is the number of vertices and E is the number of edges.

v. **void printAdjMatrix()**

- **Description**: Prints the adjacency matrix representation of the graph.
- **Time Complexity**: O(V^2) for iterating through the matrix.

vi. **void dfsPreorder(int startNode)**

- **Description**: Initiates DFS Preorder traversal from the given startNode.
- **Time Complexity**: O(V + E).

vii. **void dfsPostorder(int startNode)**

- **Description**: Initiates DFS Postorder traversal from the given startNode.
- **Time Complexity**: O(V + E).

viii. **void dfsInorder(int startNode)**

        - **Description**: Initiates a simulated DFS Inorder traversal. The neighbors of each node are sorted and traversed in order.
        - **Time Complexity**: O(V log V + E) due to the sorting step for neighbors.
- **Private Helper Methods**:
    - **void dfsPreorderUtil(int nodeIdx, vector<bool>& visited)**: Recursively explores nodes in preorder.
    - **void dfsPostorderUtil(int nodeIdx, vector<bool>& visited)**: Recursively explores nodes in postorder.
    - **void dfsInorderUtil(int nodeIdx, vector<bool>& visited)**: Recursively explores nodes in a simulated inorder manner by sorting neighbors.

---

3. Graph Representation

- **Adjacency List**:
    - Each node has a list of its neighbors.
    - **Space Complexity**: O(V + E).
    - **Time Complexity** (for adding edges): O(1).
- **Adjacency Matrix**:
    - A matrix where matrix[i][j] = 1 represents an edge between nodes i and j.
    - **Space Complexity**: O(V^2).
    - **Time Complexity** (for adding edges): O(1).

---

4. DFS Traversal Techniques

The program implements three variations of DFS traversal:

1. DFS Preorder Traversal:

- **Order**: Node → Neighbors.
- **Use Case**: Useful when you need to process each node as soon as it is encountered.
- **Time Complexity**: O(V + E).
- **Space Complexity**: O(V) for the visited array.

2. DFS Postorder Traversal:

- **Order**: Neighbors → Node.

- **Use Case**: Important for scenarios like dependency resolution where child nodes must be processed before their parent.
- **Time Complexity**: O(V + E).
- **Space Complexity**: O(V) for the visited array.

3. DFS Inorder Traversal (Simulated):

- **Order**: First Half of Neighbors → Node → Remaining Neighbors.
- **Use Case**: This is a custom traversal designed to simulate an inorder traversal for graphs. Typically used in binary trees, the neighbors are sorted to achieve a similar effect.
- **Time Complexity**: O(V log V + E) due to sorting.
- **Space Complexity**: O(V) for the visited array.

---

5. Time and Space Complexity Summary

Operation	Time Complexity	Space Complexity
Adjacency List Creation	O(V + E)	O(V + E)
Adjacency Matrix Creation	O(V^2)	O(V^2)
DFS Preorder Traversal	O(V + E)	O(V)
DFS Postorder Traversal	O(V + E)	O(V)
DFS Inorder Traversal (Simulated)	O(V log V + E)	O(V)

---

6. Conclusion

This implementation of a graph supports DFS traversals in preorder, postorder, and a simulated inorder traversal using adjacency list and adjacency matrix representations. The DFS algorithms have practical applications such as hierarchical structure exploration, dependency resolution, and optimization problems.