Lab 4

Tiffany Nguyen

CS 5393-002

1. Heaps

    a. Design Documentation

    Objectives:

    The primary objective of this program is to implement a Priority Queue using a heap data structure with message handling for errors and successes. A PriorityQ class was created using a vector to design the heap, with private functions like parent, leftChild, rightChild, heapifyUp, and heapifyDown to manage the heap. Public functions such as empty(), size(), top(), push(), pop(), and print () allow interaction with the queue. The program inserts elements, prints the queue contents, and removes elements while displaying the status and handling empty queue exceptions.

    b. Output

```
Original dataset: 50 30 10 40 20 100 70 90 60 80

Inserting elements into the PriorityQ:
Inserted 50 into the PriorityQ.
PriorityQ contents: 50
Inserted 30 into the PriorityQ.
PriorityQ contents: 30 50
Inserted 10 into the PriorityQ.
PriorityQ contents: 10 50 30
Inserted 40 into the PriorityQ.
PriorityQ contents: 10 40 30 50
Inserted 20 into the PriorityQ.
PriorityQ contents: 10 20 30 50 40
Inserted 100 into the PriorityQ.
PriorityQ contents: 10 20 30 50 40 100
Inserted 70 into the PriorityQ.
PriorityQ contents: 10 20 30 50 40 100 70
Inserted 90 into the PriorityQ.
PriorityQ contents: 10 20 30 50 40 100 70 90
Inserted 60 into the PriorityQ.
PriorityQ contents: 10 20 30 50 40 100 70 90 60
Inserted 80 into the PriorityQ.
PriorityQ contents: 10 20 30 50 40 100 70 90 60 80

Size of PriorityQ: 10

Removing elements from PriorityQ:
PriorityQ contents: 10 20 30 50 40 100 70 90 60 80
Is PriorityQ Empty? No
PriorityQ contents: 20 40 30 50 80 100 70 90 60
Is PriorityQ Empty? No
PriorityQ contents: 30 40 60 50 80 100 70 90
Is PriorityQ Empty? No
PriorityQ contents: 40 50 60 90 80 100 70
Is PriorityQ Empty? No
PriorityQ contents: 50 70 60 90 80 100
Is PriorityQ Empty? No
PriorityQ contents: 60 70 100 90 80
Is PriorityQ Empty? No
PriorityQ contents: 70 80 100 90
Is PriorityQ Empty? No
PriorityQ contents: 80 90 100
Is PriorityQ Empty? No
PriorityQ contents: 90 100
Is PriorityQ Empty? No
PriorityQ contents: 100
Is PriorityQ Empty? Yes
Exception: Priority Q is empty.
```

c. What is the time complexity of the operations in (iii) and (v)?

The time complexity of the push function (iii) is O(log n), since the height of the heap is O(log n). When inserting the element into the heap, the heapify-up operation is used to maintain the heap property. For the pop function (v), when removing the top element from the heap, the heapify-down operation is used to restore the heap property. The element may need to move down through the height of the heap, which is O(log n)

d. What is the time complexity of the operations in (ii) and (iv)?

The time complexity for printing the dataset (ii) is O(n) time, where n is the number of elements in the dataset, as it prints the dataset iterating over all elements in the dataset. For printing the remaining queue (iv), it is like printing the entire queue, but only requires iterating through all elements currently in the queue. If there are n elements, it'll take O(n) time.

e. What is the space complexity of the PriorityQ implementation of n elements?

The space complexity of the PriorityQ implementation is determined by the storage used for the heap. Since the heap is implemented using a vector, it requires space proportional to the number of elements stored. Therefore, the space complexity is O(n), where n is the number of elements in the priority queue.

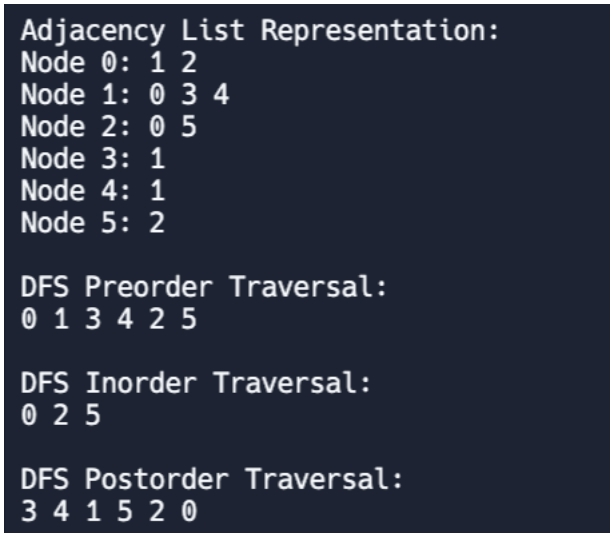
2. Graphs

    a. Design Documentation:

    Objectives:

    The objective of this program is to implement a graph with 6 nodes using two different representations: an adjacency list and an adjacency matrix. These two approaches will be compared for their efficiency and ease of use. Additionally, the program will include depth-first search (DFS) traversal methods: preorder, inorder, and postorder, implemented as member functions within a graph class. The DFS functions will explore the graph by visiting nodes in a specific order to demonstrate the traversal patterns for each method.

    b. Output:



c. What is the time complexity of DFS in a graph of n nodes?

The time complexity of depth-first search in a graph with n nodes is O(n+m), where m is the number of edges in the graph. This complexity arises because, in the worst case, DFS visits every node and explores every edge. Each node is processed once, and each edge is checked one when traversing to an adjacent node. Therefore, the total time taken is proportional to the sum of the number of nodes and number of edges.

d. What is the space complexity of a graph of n nodes?

The space complexity of a graph with n nodes can vary depending on the representation used. For an adjacency list representation, the space complexity is O(n+m), where m is the number of edges. This is because you need to store each node and its associated edges in the list. In constract, the adjacency matrix representation has a space complexity of $(O(n^2)$ because it requires a fixed-size matrix to represent connections between every pair of nodes, regardless of the number of edges.

e. What is the utility of Pre-, In- and Post-Order traversals?

In pre-order traversal, the node is processed before its children. This traversal method is useful for tasks where you need to create a copy of a tree or evaluate an expression tree, as it allows you to access the root node first. For instance, in prefix notation expressions, where the operator precedes its operands, pre-order traversal naturally represents the expression's structure, making it suitable for evaluating or parsing expressions. In-order traversal processes the left subtree, the node itself, and then the right subtree. This method is particularly useful for

binary search trees (BSTs), as it retrieves nodes in sorted order. If you need to output elements in a non-decreasing sequence, in-order traversal is the way to go. In the context of applications, in-order traversal can be instrumental in tasks like producing sorted output from a BST, as it directly reflects the properties of the tree. In post-order traversal, the node is processed after its children. This traversal is essential in situations where you need to delete or free nodes in a tree, as it ensures that you process child nodes before their parent nodes, thus preventing memory leaks. Additionally, post-order traversal is beneficial in evaluating expression trees where the operators are processed after their operands, which is vital in postfix notation.

References

ChatGPT.

GeeksforGeeks.