Student Name: Ganesh Nathan
Student ID: 2021BCS0104

ICS225 Data Strucutres-III
Lab - 4

1 Problem statement

Write a program to
(1) Insert n elements to a binary min-heap
(2) Delete an element
(3) Extract-min

2 Solution Description

A Min Heap is a data structure that ensures each node's value is smaller than both of its children. It is represented as a complete binary tree, where each level is filled from left to right.

The operations supported by a Min Heap include:

(1) **Insertion**: To add an element to the heap, it is appended at the end of the tree, similar to adding it to the last position of an array. Then, the element undergoes a process called *"Heapify Upwards"*. This process compares the node with its parent and swaps them if the node is smaller. This continues until the heap property is satisfied.

(2) **Deletion**: To remove an element from the heap, the node to be deleted is located and replaced with the last node in the heap. The heap is then reduced in size, and the replaced node undergoes a process called *"Heapify Downwards"*. This process compares the node with its children (if they exist) and swaps it with the smallest child that is smaller than the node. This continues until the heap property is satisfied.

(3) **Extract Minimum**: The minimum value in the heap is always the root node. To extract this minimum value, the *"Deletion"* operation is performed on the root, and the deleted value is returned as the extracted minimum.

3 Pseudocode

```
# Min-Heap Insertion
Insert(heap, value):
    heap.append(value)  # Add the new element to the end of the heap
    index = len(heap) - 1  # Index of the new element
    while index > 0:
        parent_index = (index - 1) // 2  # Calculate the parent index
        if heap[parent_index] > heap[index]:
            heap[parent_index], heap[index] = heap[index], heap[parent_index]  # Swap parent and child
```

```
            index = parent_index
        else:
            break

# Min-Heap Deletion
Delete(heap, index):
    if len(heap) == 0:
        return None
    if index > len(heap): # Value doesn't exist in the heap
        return None
    delete_value = heap[index]  # Store the minimum value to be returned
    heap[index] = heap[-1]  # Replace the root with the last element
    heap.pop()  # Remove the last element
    while True:
        child_index1 = (2 * index) + 1  # Index of the left child
        child_index2 = (2 * index) + 2  # Index of the right child
        smallest = index  # Assume the current element is the smallest
        if child_index1 < len(heap) and heap[child_index1] < heap[smallest]:
            smallest = child_index1
        if child_index2 < len(heap) and heap[child_index2] < heap[smallest]:
            smallest = child_index2
        if smallest != index:
            heap[index], heap[smallest] = heap[smallest], heap[index]  # Swap current
                element with smallest child
            index = smallest
        else:
            break
    return delete_value

# Extract Minimum
Extract_Min(heap):
    return delete(heap, 0)
```

4 Implementation

```
#include <iostream>
#include <vector>

using namespace std;

class MinHeap
{
private:
    vector<int> heap;

    int parent(int i) { return (i - 1) / 2; }
    int leftChild(int i) { return (2 * i) + 1; }
    int rightChild(int i) { return (2 * i) + 2; }

    void heapifyUp(int i)
    {
        while (i > 0 && heap[i] < heap[parent(i)])
```

```
        {
            swap(heap[i], heap[parent(i)]);
            i = parent(i);
        }
    }

    void heapifyDown(int i)
    {
        int smallest = i;
        int left = leftChild(i);
        int right = rightChild(i);

        if (left < heap.size() && heap[left] < heap[smallest])
            smallest = left;

        if (right < heap.size() && heap[right] < heap[smallest])
            smallest = right;

        if (smallest != i)
        {
            swap(heap[i], heap[smallest]);
            heapifyDown(smallest);
        }
    }

public:
    void insert(int value)
    {
        heap.push_back(value);
        int index = heap.size() - 1;
        heapifyUp(index);
    }

    void remove(int value)
    {
        int index = -1;
        for (int i = 0; i < heap.size(); i++)
        {
            if (heap[i] == value)
            {
                index = i;
                break;
            }
        }

        if (index == -1)
        {
            cout << "Element not found in the heap." << endl;
            return;
        }

        heap[index] = heap.back();
        heap.pop_back();

```

```cpp
        if (index < heap.size())
        {
            if (index > 0 && heap[index] < heap[parent(index)])
                heapifyUp(index);
            else
                heapifyDown(index);
        }
    }

    int extractMin()
    {
        if (heap.empty())
        {
            cout << "Heap is empty." << endl;
            return -1;
        }

        int min = heap[0];
        heap[0] = heap.back();
        heap.pop_back();
        heapifyDown(0);

        return min;
    }

    void display()
    {
        if (heap.empty())
        {
            cout << "Heap is empty." << endl;
            return;
        }

        cout << "Min-Heap: ";
        for (int i = 0; i < heap.size(); i++)
            cout << heap[i] << " ";

        cout << endl;
    }
};

int main()
{
    MinHeap minHeap;
    int n, value;

    cout << "Enter the number of elements to insert: ";
    cin >> n;

    cout << "Enter " << n << " elements: \n";
    for (int i = 0; i < n; i++)
    {
        cin >> value;
        minHeap.insert(value);
```

```
    }

    minHeap.display();
    int ch;
    int t;
    do
    {
        cout << "Choose an option: \n"
             << "1. Insert\n"
             << "2. Delete\n"
             << "3. Extract Min\n"
             << "Anything else to quit\n";
        cin >> ch;
        switch (ch)
        {
        case 1:
        {
            cout << "Enter value to insert: ";
            cin >> t;
            minHeap.insert(t);
            minHeap.display();
            break;
        }

        case 2:
        {
            cout << "Enter value to delete: ";
            cin >> t;
            minHeap.remove(t);
            minHeap.display();
            break;
        }
        case 3:
        {
            int min = minHeap.extractMin();
            if (min != -1)
                cout << "Extracted minimum element: " << min << endl;
            break;
        }

        default:
        {
            cout << "Wrong Choice!";
            break;
        }
        }
    } while (ch > 0 && ch < 4);

    return 0;
}
```

5 Algorithm Analysis

If n represents the number of nodes in the heap, the time and space complexity of the operations are as follows:

- Insertion: Time = $O(\log n)$, Space = $O(1)$
- Deletion: Time = $O(\log n)$, Space = $O(1)$
- Extract Minimum: Time = $O(\log n)$, Space = $O(1)$

6 Reference

- MIT Lecture on Binary Heaps
- CLRS (Introduction to Algorithms) for algorithms and data structures