HW1 – Part I

Algorithms and Computation

Name: Robert Rice
Date: December 4, 2025
Student ID: 657588340
Language: Java

---

1. Introduction

The goal of this Homework is to build a **priority queue** using a **binary min heap**. In this part, the focus is on the *data structure* itself, not on the full graph algorithm.

The main requirements are as follows:

- Store items with a separate priority value.
- Always return the item with the **smallest** priority quickly.
- Support changes to the priority of an existing item.
- Support deletions of any item from the heap.
- Keep all operations efficient;

2. Design Overview

I implemented a class called `MinHeapPriorityQueue`. It is a **min heap**, which means:

- Each node has a priority.
- The node at the top (the root) always has the **smallest** priority.
- Every child node has a priority that is greater than or equal to its parent.

The heap is stored in an **array** using 1-based indexing, so:

- The root is at index 1.
- The left child of index i is at index $2i$.
- The right child of the index i is at the index $2i + 1$.

To support fast lookup for `Delete(item)` and `ChangePriority(item, newPriority)`, I also use a **map** (`HashMap`) that remembers where each item is stored in the array.

3. Data Structures

3.1 HeapNode

I created an inner class called `HeapNode` with two fields:

- `int item` – the item itself (for example, a vertex id in a graph).
- `int priority` – the value used to order items in the heap (for example, a distance).

This lets me keep the *data* and its *priority* separate, which is important later.

3.2 Heap Array

- `HeapNode[] heap` – stores all the nodes.
- I use indices from 1 to `size`. Index 0 is unused.

Using a complete binary tree structure allows the heap to fit nicely into this array.

3.3 Position Map

- `Map<Integer, Integer> position` – maps `item` → `index in heap array`.

The position map is crucial for the following:

- Finding an item's index in $O(1)$ average time.
- Implementing `Delete(item)` and `ChangePriority(item, newPriority)` in $O(\log n)$ time, because once I know the index, I can fix the heap using `Heapify_Up` or `Heapify_Down`.

4. Implemented Operations

The following is a summary of the main methods and how they work.

4.1 `StartHeap(N)`

Method: `StartHeap(int capacity)` (also called from the constructor)

This method:

- Allocates the heap array with size `capacity + 1`.
- Set `size = 0`.
- Creates a new empty `position` map.

Set up an empty heap that can hold up to N elements. The running time is $O(N)$ to allocate and initialize the array.

4.2 `Heapify_Up(index)`

Methods `private void Heapify_Up(int index)`

This method is used when an item's priority becomes **smaller** (better), such as after an insert or a priority decrease. While the node's priority is smaller than its parent's priority, I:

- Swap the node with its parent.
- Update the `position` map for both nodes.
- Continue moving up the tree.

This restores the heap property from a leaf toward the root. The running time is $O(\log n)$.

4.3 `Heapify_Down(index)`

Methods `private void Heapify_Down(int index)`

This method is used when an item's priority becomes **larger** (worse), or when we move the last node into a hole created by a deletion. Repeatedly:

- Look at the left and right children.
- Find the child with the smallest priority.
- If that child has a smaller priority than the current node, swap them.
- Update the `position` map and continue downward.

This restores the heap property from the root to the leaves. The running time is $O(\log n)$.

4.4 `Insert(item, value)`

Methods `public void Insert(int item, int priority)`

Steps:

1. Check if there is room in the array.
2. Check if the item is not already in the heap.
3. Increase `size` by 1.
4. Create a new `heap node(item, priority)` at `heap[size]`.
5. Add `position.put(item, size)`.
6. Call `Heapify_Up(size)` to restore the min-heap property.

The running time is $O(\log n)$.

4.5 `FindMin()`

Methods `public HeapNode FindMin()`

- If `size == 0`, return `null`.
- Otherwise, return `heap[1]`.

Because the minimum is always stored at the root, this operation runs in $O(1)$ time.

4.6 `Delete(index)`

Method: implemented as `private void DeleteIndex(int index)`

This is the internal version of `Delete(index)` from the handout.

Steps:

1. If the index is invalid, return.
2. If `index == size` (last element):
    - Remove it from the `position` map.
    - Set `heap[index] = null` and decrease `the size`.

3. Otherwise:
    - Save the node that we are deleting.
    - Save the last node at `heap[size]`.
    - Move the last node to `heap[index]`.
    - Remove the deleted item from `position`.
    - Update `the position` for the last node to the new index.
    - Reduce `size`.
    - If the priority of the new node is lower than that of its parent, call `Heapify_Up(index)`. Otherwise, call `Heapify_Down(index)`.

The running time is $O(\log n)$.

4.7 `ExtractMin()`

Methods `public HeapNode ExtractMin()`

- If the heap is empty, return `null`.
- Otherwise:
    - Save `heap[1]` as the minimum.
    - Call `DeleteIndex(1)`.
    - Return the saved node.

This removes and returns the smallest-priority item. The running time is $O(\log n)$.

4.8 `Delete(item)`

Methods `public void Delete(int item)`

Steps:

1. Look up `index = position.get(item)`.
2. If `the index` is `null`, the item is not in the heap.
3. Otherwise, call `DeleteIndex(index)`.

The position map gives an average lookup of $O(1)$, and `the DeleteIndex` is $O(\log n)$, so the total running time is $O(\log n)$.

4.9 `ChangePriority(item, newPriority)`

Methods `public void ChangePriority(int item, int newPriority)`

Steps:

1. Look up `index = position.get(item)`.
2. If `index` is `null`, the item is not in the heap; return.
3. Save `oldPriority = heap[index].priority`.
4. Set `heap[index].priority = newPriority`.
5. If `newPriority < oldPriority`, call `Heapify_Up(index)`.

6. Else if `newPriority > oldPriority`, call `Heapify_Down(index)`.
7. If the priority is unchanged, no heapify is needed.

The running time is $O(\log n)$. This method is especially important for the Dijkstra algorithm, where we often improve the distance (priority) of a vertex.

5. Time Complexity Summary

For a pile of size n:

- `StartHeap(N)`: $O(N)$
- `Insert(item, priority)`: $O(\log n)$
- `FindMin()`: $O(1)$
- `ExtractMin()`: $O(\log n)$
- `Delete(index)` (via `DeleteIndex`): $O(\log n)$
- `Delete(item)`: $O(\log n)$
- `ChangePriority(item, newPriority)`: $O(\log n)$

These match the expected Big-Oh efficiencies.

6. Conclusion

In this Homework, I implemented a priority queue using a binary min heap with an additional position map. The data structure supports the following.

- Inserting items with priorities.
- Finding and extracting the minimum item.
- Deleting arbitrary items.
- Change the priority of any item and fix the heap correctly.

All main operations are executed in $O(\log n)$ time or better, in accordance with assignment requirements.

This priority queue is now ready to be used in the latest assignment, where I will implement and adapt Dijkstra's algorithm for the Almost Shortest Path problem.