Analysis and Implementation of Heap Data Structure

In this implementation, I created two heap structures using arrays: MinHeapPriorityQueue and MaxHeapPriorityQueue. By implementing heaps with lists, memory efficiency is improved, and navigation of parent-child relationships through index calculation becomes easier.

The main functions implemented are sift_up and sift_down, which maintain the characteristic that parent nodes are always larger or smaller than their child nodes. Also, by storing task_id and task index in the task_position dictionary, I made it possible to instantly retrieve the position of a specific task. This allows for O(1) time complexity when searching for specific tasks, which is an important implementation for efficiently changing priorities, as without it, the operation would require O(n) time.

Additionally, operations such as insertion, extraction, and priority changes can all be executed with O(log n) time complexity.

The time complexity for each function is as follows:

1. is_empty(): O(1)

   A function that checks if the heap is empty.

2. insert(task): O(log n)

   Adds a task to the end of the heap and maintains the heap property using sift_up().

3. extract_min/max(): O(log n)

   Pops the minimum/maximum element and maintains the heap property using sift_down().

4. decrease/increase_key(): O(log n)

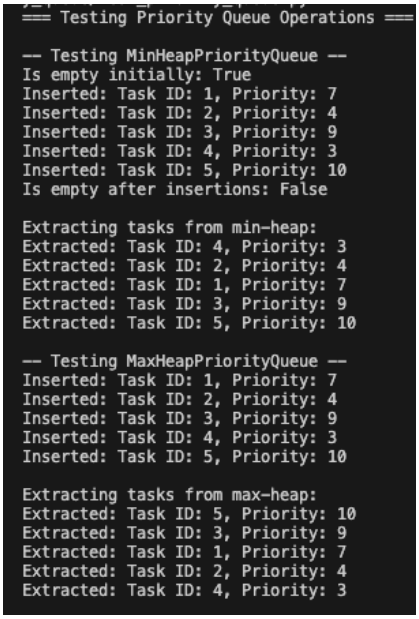
   Changes the priority of a task, then maintains the heap property using sift_up() or sift_down().

Where n is the number of elements in the heap.

For space complexity, the heap array is O(n), and the task_position dictionary is also O(n).

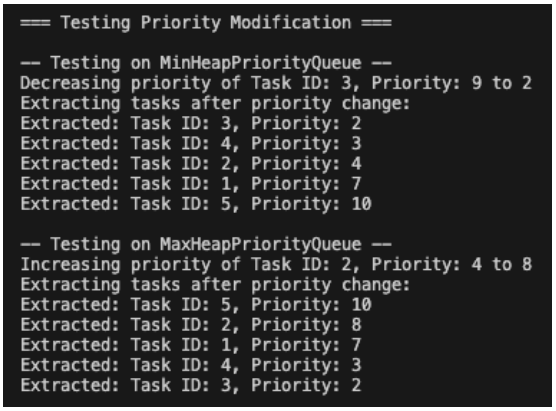
Therefore, the overall space complexity is O(n).

To verify the priority queue, I conducted test runs. First, for operational verification, I tested five tasks (priorities: 7, 4, 9, 3, 10):



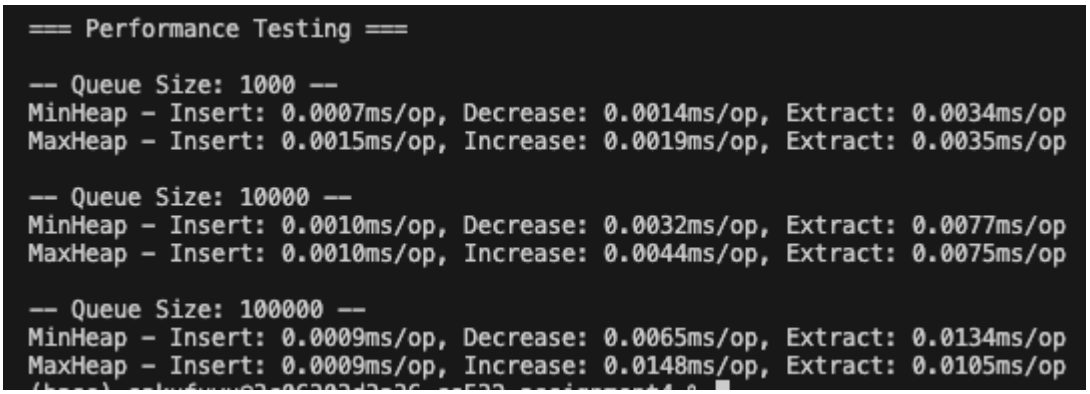
- **MinHeapPriorityQueue:** I confirmed that elements were extracted in order of low priority (3, 4, 7, 9, 10).
- **MaxHeapPriorityQueue:** I confirmed that elements were extracted in order of high priority (10, 9, 7, 4, 3).

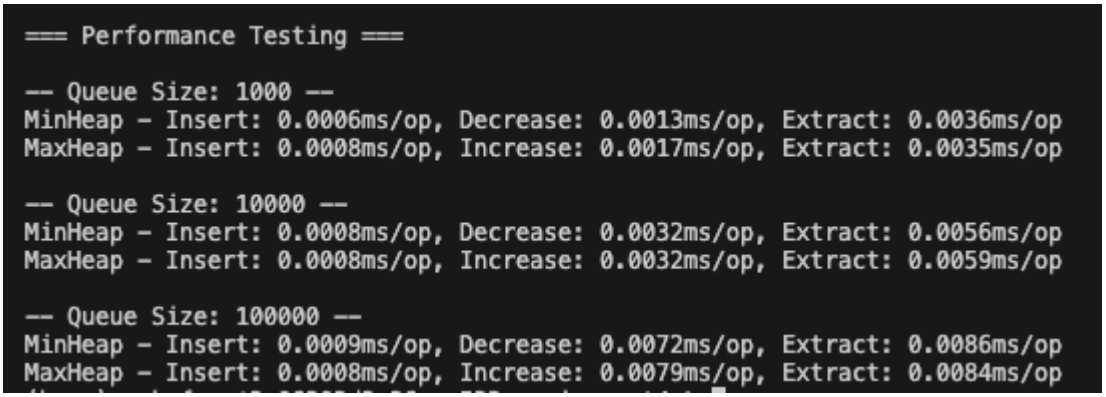
Next, the results of priority change verification are as follows:



- **MinHeapPriorityQueue:** When Task 3's priority was changed from 9 to 2, I confirmed that the extraction order became (2, 3, 4, 7, 10), correctly extracting in order of low priority after the change.
- **MaxHeapPriorityQueue:** When Task 2's priority was changed from 4 to 8, I confirmed that the extraction order became (10, 8, 7, 3, 2), correctly extracting in order of high priority after the change.

For performance analysis, I measured the average execution time for various operations with queue sizes of 1,000, 10,000, and 100,000.





MinHeap Performance:

Queue Size	Insert (ms/op)	Decrease (ms/op)	Extract (ms/op)
1,000	0.0006	0.0013	0.0036
10,000	0.0008	0.0032	0.0056
100,000	0.0009	0.0072	0.0086

MaxHeap Performance:

Queue Size	Insert (ms/op)	Increase (ms/op)	Extract (ms/op)
1,000	0.0008	0.0017	0.0035
10,000	0.0008	0.0032	0.0059
100,000	0.0008	0.0079	0.0084

The theoretical growth rate of the logarithmic function when the queue size increases by 10 times is:

- $log_2(10000)/log_2(1000) = 1.33333333...$
- $log_2(10000)/log_2(100000) = 1.25$

Looking at the experimental results, insert operations are fast across all data sizes and run at speeds close to the theoretical log(n) speed. In MaxHeap, execution speed remains constant across all queue sizes, indicating very efficient operation.

However, increase/decrease_key() operations are quite slow, with speed increasing nearly twice every time the queue size increases by 10 times in both MinHeap and MaxHeap, which is a much worse rate of increase than the theoretical logarithmic speed. It's possible that the heap reconstruction after key changes is more complex than expected.

For extract operations, they become approximately 1.5 times slower every time the queue size increases by 10 times, which is slightly higher than but close to the theoretical logarithmic growth rate.