2022310853 박연우

알고리즘개론 Assignment 1

<Min-Max-Median Priority Queue>

1. Objective

The objective of this assignment was to implement a Min-Max-Median Priority Queue in C. As the priority queue's name states, the goal was to create a data structure, which can efficiently find the minimum, maximum, and median integer elements. The median is defined as the smaller of the two middle elements when the number of elements is even.

2. Data Structure Implementation

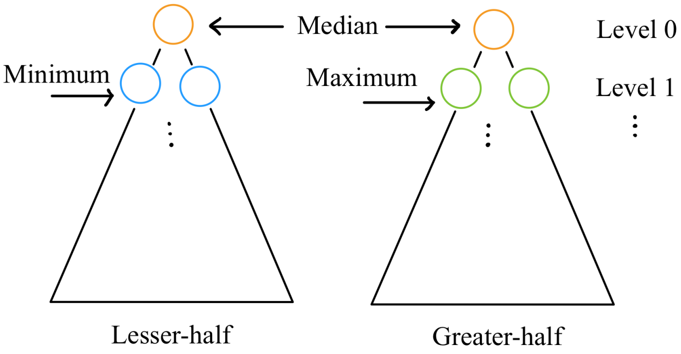
The Priority Queue I have designed in this assignment is composed of two heaps: a Min-Max Heap and a Max-Min Heap.

Before I dive into the structure of the Priority Queue, I will briefly explain the structure of Min-Max Heap. A Min-Max Heap is a complete binary tree data structure which combines structure between a Max Heap and a Min heap. In a Min-Max Heap, even levels consist of a Min Heap, while odd levels consist of a Max Heap. For instance, the root node (level 0) will hold the minimum element of the tree, the children of the root node (level 1) will hold the maximum element of their respective trees, and as we go down the heap, it continues to alternate between minimum and maximum levels. A Max-Min Heap is defined similarly to a Min-Max Heap: its maximum is on the even levels and the minimums on the odd levels.

In the Priority Queue designed for this project, the Min-Max Heap stores the greater-half of the elements entered and the Max-Min Heap stores the lesser-half of the elements entered. (Each heap will be referred to as the greater-half and lower-half from now on.)

- If the total number of elements is even, both heaps will have the same number of elements.
- If the total number of elements is odd, the greater-half will have one more element compared to that of the lesser-half.

If we structure the Priority Queue in such a way, we are able to store the minimum, median, and maximum values of the elements in an efficient manner. As shown in <Image 1>.



<Image 1> Visualization of the Priority Queue

In the Priority Queue, the minimum of the elements will be stored in the lesser-half. Since the lesser-half is a Max-Min heap, the minimum of the lesser-half will be stored in Level 1 (denoted by blue circles).

The median, denoted by orange circles, will be stored in the root of the lesser-half if the number of elements is even. If the number of elements is odd, the median will be at the root of the greater-half.

The maximum of the greater-half of elements will be the maximum for all elements. Since the greater-half is a Min-Max Heap, the maximum will be stored in Level 1 of the greater-half.

By implementing a Priority Queue consisting of a Min-Max Heap and Max-Min Heap, we can store the values of the elements and efficiently find the minimum, median, and maximum.

3. Performance Analysis

The Priority Queue needs to be able to do three types of operations efficiently: find, insert, delete. I will be reviewing the performance of each operation in terms of their time complexity. The overall performance summary of these operations is as follows:

	Average Case	**Worst Case**
Find	O(1)	O(1)
Insert	O(log n)	O(log n)
Delete	O(log n)	O(log n)

<Table 1> Time complexity of each operation in big O notation

1. **Find**

   Because of the structure of the Priority Queue, finding the minimum, median, and maximum only takes one comparison. To find the minimum, only one comparison of the two nodes on level 1 of the lesser-half is needed. To find the maximum. only one comparison on the two nodes of level 1 of the greater-half is needed. Lastly, to find the median, the number of elements decide whether the median will be in the root of the lesser-half or greater-half. Thus, finding the minimum, maximum, and median takes a constant amount of time, giving us O(1).

2. **Insert**

   A Max-Min Heap has a tree height of O(log n), when n is the number of elements in the heap. If we insert an element to a Max-Min Heap, it will be appended as the last leaf node. In order to maintain the Max-Min Heap properties, the appended element needs to be pushed up through the Max-Min Heap into its correct position. Since the height of the Max-Min Heap is O(log n), it will take a maximum number of O(log n) swaps to maintain the heap property. The same applies to a Min-Max Heap taking O(log n) time since it has an analogous structure to a Max-Min Heap.

3. **Delete**

   In parallel to the complexity of insertion, deletion also takes similar steps. When deleting a node from the heap, it is swapped with the last node of the heap. After removing the last node from the heap, the swapped element needs to be pushed down the heap to its correct position to maintain the heap property. Since the height of the tree is O(log n), the maximum number of swaps needed will be O(log n).

4. Code Explanation

Data Structure:

1. **Min-Max Heap Structure**

   The MinMaxHeap structure works as a Min-Max Heap if min_max = 1, and works as a Max-Min heap is min_max = 0;

```
typedef struct MinMaxHeap{
        int arr[MAX_SIZE/2+1];
        int size;
        int min_max; //for min-max heap = 1 and max-min heap = 0
} Heap;
```

2. **Priority Queue Structure**

```
typedef struct PriorityQueue{
        Heap lesser; //max-min heap stores the lesser-half of values
        Heap greater; //min-max heap stores the greater-half of values
} PQ;
```

   The Priority Queue was set as a global variable: `PQ pq;`

Functions:

The functions for this Priority Queue consists of four parts: main function, Priority Queue functions, Heap functions, and helper functions.

1. **Main function**

   The main function takes in the number of operations that will be performed, and performs the operations based on the operation type and target. When the find operation is entered, it prints the found element (prints "NULL" if there are no elements). In cases where a delete operation is called when the Priority Queue is empty, the main function skips the operation.

2. **Priority Queue functions**

   The Priority Queue functions assume that the Priority Queue has at least one element since instances where the Priority Queue is empty will be handled in the main function.

   void initPriorityQueue()

   - Initializes priority queue. The size of both heaps are set to 0. The greater-heap's min_max is set to 1 to indicate it is a Min-Max Heap, and the lesser_heap's min_max is set to 0 to indicate it is a Max-Min Heap.

void insert(int element)

- Inserts an integer element into the priority queue. If the greater-half is empty, the element is added to the greater-half. If the element being inserted is larger than the root of the greater-half, it will be inserted into the greater-half. If not, the element will be inserted into the lesser-half. To keep the heap size difference less or equal to 1, if both greater-half and lesser-half are the same size and the element is being inserted into the lesser-half, the maximum of the lesser-half is removed and added to the greater-half. If the greater-half is larger than the lesser-half and the element is being added to the greater-half, the minimum of the greater-half is removed and added to the lesser-half.

int find min()

- Fetches but does not remove the minimum element. If the lesser-half contains elements, the function returns the smaller of the two elements at level 1 of the lesser-half. If there are no elements in the lesser-half the function returns the root of the greater-half.

int find median()

- Fetches but does not remove the median element. If the lesser-half and greater-half have the same size, the function returns the root node of the lesser-half. If the greater-half is larger than the lesser-half, it returns the root node of the greater-half.

int find max()

- Fetches but does not remove the maximum element. The function returns the larger of the two elements at level 1 of the greater-half.

int delete min()

- Deletes and returns the minimum element. If the lesser-half and greater-half are the same size, the minimum from the lesser-half is removed. If the greater-half is larger than the lesser-half in size, the minimum element is removed from the lesser half, and the minimum element from the greater-half is popped and inserted into the lesser-half.

int delete median()

- Deletes and returns the median element. If the lesser-half and the greater-half are the same size, the maximum element from the lesser-half is popped and returned. If the greater-half is larger than the lesser-half in size, the minimum element of the greater-half is removed and returned.

int delete max()

- Deletes and returns the maximum element. If both halves are the same size, the maximum element of the lesser-half is popped and inserted into the greater-half. Then, the maximum element of the greater-half is popped and returned.

3. Heap functions

The following heap functions were created in reference to the pseudo code shown on Wikipedia's Min-max Heap webpage[1]. The iterative form was used to implement the functions.

void push_down(Heap* heap, int index)

- Pushes down the element at the given index down the given heap to its correct position. While the element being pushed down has children, if the element is at a minimum level and is larger than the smallest of its children and grandchildren, they are swapped. If the index at which the element was swapped is a grandchild of its original index, the element is once again swapped with the parent.

int pop_min(Heap* heap)

- Deletes and returns the minimum element of the given heap. The minimum element is determined and swapped with the last element. The last element, now swapped, is pushed down the heap to maintain Heap property.

int pop_max(Heap* heap)

- Deletes and returns the maximum element of the given heap. The maximum element is determined and swapped with the last element. The last element, now swapped, is pushed down the heap to maintain Heap property.

void push_up_max(Heap* heap, int index)

- Pushes up the element at the given index to the max levels of the given heap. The element is compared to its grandparent, and if it is larger, it is swapped until it no longer has a grandparent or is smaller than its grandparent.

void push_up_min(Heap* heap, int index)

- Pushes up the element at the given index to the min levels of the given heap. While the element has a grandparent and is smaller than their grandparent, it is swapped with its grandparent.

void push_up(Heap* heap, int index)

- Pushes up the element at the given index to the correct position of the given heap. If the index is not the root and at a min level, the element is compared to its parent. If it is larger than the parent, it is swapped and push_up_max is called to push it up to its correct position in the max levels. Otherwise, we push_up_min to push it up to its correct position in the min levels. Vice versa if the element is at a max level from the start.

void heap_insert(Heap* heap, int element)

- Inserts the given element to the given heap and increases the size of the heap by 1. It calls to the push-up functions to maintain the heap property.

[1] "Min-Max Heap." *Wikipedia*, Wikimedia Foundation, 26 June 2023, en.wikipedia.org/wiki/Min-max_heap.

4. Helper functions

The following functions were used to help the implementation of the Heap and Priority Queue functions:

void swap(int* a, int* b)

- Swaps the two elements passed.

int get_tree_level(int index)[2]

- Returns the level of the heap based on the index passed.

int at_min_level(Heap* heap, int index)

- Returns 1 if the node at given index is on a min level of the heap.

int smallest_descendant_index(Heap* heap, int index)

- Returns the index of the smallest child or grandchild of a given index node. If there are no children or grandchildren, -1 is returned.

int largest_descendant_index(Heap* heap, int index)

- Returns the index of the largest child or grandchild of a given index node. If there are no children or grandchildren, -1 is returned.

int is_a_grandchild(Heap* heap, int grandparent, int grandchild)

- Returns 1 is the grandchild index is a grandchild of the grandparent index of a given heap.

int find_heap_max_index(Heap* heap)

- Returns the maximum's index of the given heap. The maximum's index is the index of the larger of the level 1 node's index if greater half is passed, and 0 is the lesser-half is passed.

int find_heap_min_index(Heap* heap)

- Returns the minimums' index of the given heap. The minimum's index is 0 if the greater-half is passed, and smaller of the level 1 node's index if the lesser-half is passed.

5. Conclusion

The Priority Queue described in the assignment uses a Min-Max heap to store the greater half of values and a Max-Min heap to store the lesser-half of the values. It is able to efficiently find the minimum, median, and maximum in constant time. Inserting an element and deleting the minimum, median, or maximum is completed in O(log n) time.

[2] Hrehfeld, et al. "How Can I Calculate the Level of a Node in a Perfect Binary Tree from Its Depth-First Order Index?" Stack Overflow, 1 Sept. 1958, stackoverflow.com/questions/10721583/how-can-i-calculate-the-level-of-a-node-in-a-perfect-binary-tree-from-its-depth.

6. Work Cited

"Min-Max Heap." *Wikipedia*, Wikimedia Foundation, 26 June 2023, en.wikipedia.org/wiki/Min-max_heap.

Hrehfeld, et al. "How Can I Calculate the Level of a Node in a Perfect Binary Tree from Its Depth-First Order Index?" Stack Overflow, 1 Sept. 1958, stackoverflow.com/questions/10721583/how-can-i-calculate-the-level-of-a-node-in-a-perfect-binary-tree-from-its-depth.