
CS 225: DATA STRUCTURES

Homework 4

Group D1

Last Modified on March 19, 2022

Members	
Name	Student ID
Li Rong	3200110523
Zhong Tiantian	3200110643
Zhou Ruidi	3200111303
Jiang Wenhan	3200111016

PLEASE TURN OVER FOR OUR ANSWERS.

Ex. 1 **OUR ANSWER.**

- (i) Firstly we set the doubly linked list whose node's values follows the correct sequence. For operation `delete_min` or `delete_max`, we simply delete the first or last node of the linked lists. Then we set the value of previous node pointer of the second element to `NULL` or the value of next node pointer of the last but second element to `NULL`. For insert operation, for example in a max heap, we traverse from the value of the first node until we reach a node whose value is smaller than the insert one. Then we set its previous node's next pointer to the inserted node and set the current node's previous node pointer to the inserted node. Finally we set the next node pointer of the inserted node to the current node and the previous node pointer of the inserted node to the last traversed node. For `decrease(h, k)` operation, we traverse the doubly linked list and check its key value until we find the h 's node and decrease its value by k . Then we traverse the following nodes (for the sequence from big to small) to find the correct position for the node h , then we insert it to the correct position and delete the primal node.
- (ii) For insert and decrease operation, the time complexity for unsorted list is $O(1)$. For sorted list, the time complexity is $O(n)$. For delete operation, time complexity for both unsorted lists and sorted lists are $O(1)$.

Ex. 2 **OUR ANSWER.**

- (i) Each time we could compare the new element with the element in the lowest level. If the new element is smaller than the element in the heap, we could put this new element at the bottom of the heap. If the new element is bigger than the element in the heap, we should exchange the elements: put the new element in the original position of the element in the heap and move the original element in the heap to the bottom. Then we should compare the new element with the parent leaf in that position. Repeat this step until it meets a element in the heap which is bigger than the element or the new element is the biggest, we should put it at the top of the heap. For each new element we could repeat this process until we have finished inserting all the elements.

- (ii) **NOTE: We suspect that the given complexity is incorrect.**

We drew a figure for the inserting process. Please refer to the figure at the end of this document.

The labelled example is shown at the end of this exercise.

Initiation: There are n elements in `maxHeap` and k new elements to insert.

Step 1: First put the new elements at the bottom of `maxHeap`.

Step 2: Consider nodes with two children. Their ancestors are colored yellow except for a (since a is linked with l who has only one child).

Now we want to *heapify* with d, e , i.e. make the properties of a heap maintained in subtrees with root d and e . Do it also for all ancestors of d and e from the bottom to the top, so the left subtree of root a is maintained with the properties of a heap.

Because there are at most $\lfloor k/2 \rfloor$ two-children parent nodes for k new elements, the above process takes

$$\left\lfloor \frac{k}{2} \right\rfloor + \left\lfloor \frac{k}{4} \right\rfloor + \cdots = \sum_{i=1}^{\lceil \log(n+k) \rceil} \frac{k}{2^i} \approx k$$

steps, thus has time complexity of $O(k)$.

Step 3: We now calculate the father node with only one child node (the blue nodes). Note that they do not have other nodes to amortize the `siftDown` time. Thus there are at most $\log(n)$ nodes in the link (in the example, a, c and f), with each node taking $\log(n+k)$ time

to form a heap (so far, we cannot consider them as a heap as it is like a link without other branch nodes, as a result we can neither consider the time complexity as $O(\log(n))$). So the total time complexity is $\log(n) \log(n+k) \approx O(\log(n) \log(n+k))$.

Therefore, the final time complexity is $O(k + \log(n) \log(n+k))$ rather than $O(k + \log(n))$.

Inserting New Nodes. We append the new nodes to the binary heap, maintaining its properties as a complete binary tree. This indicates that there are at most one nodes whose number of children is 1.

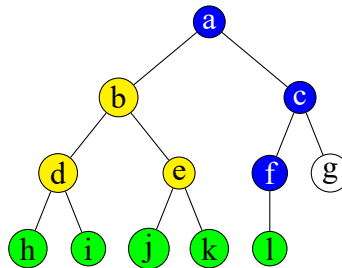


Figure 1: The example maxHeap. Green nodes are elements inserted. The blue ones are those linked with the node f , which has only one child. The yellow ones are linked with nodes d, e which have two children.

Ex. 3 **OUR ANSWER.**

- (i) The `siftUp` operation requires $\log(n)$ comparisons, each layer requires one. At each layer, compare the leaf node with parent node and swap the leaf node and parent node if needed. Therefore the time complexity is $O(\log n)$. For insert operation, we add the inserted node to the end of the heap. Then we apply `siftUp` operation to it until the property of heap is gained.
- (ii) Firstly we find the larger leaf node for each layer and it takes $\log n$ times of comparison. Then we set up a list consists of these values and the amount is $\log n$, then we apply binary search to find the proper position. It takes $O(\log \log n)$ steps to accomplish. Thus, the steps needed is $\log n + O(\log \log n)$.

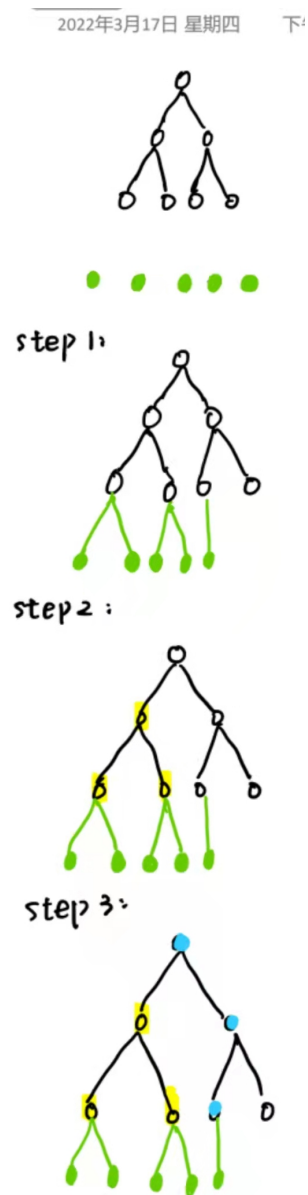


Figure 2: The hand-drawn example inserting process for Ex. 2.