CS2040S: Data Structures and Algorithms

Tutorial Problems for Week 7 Monday: Heaps and Priority Queues

For: 26 September 2022, Tutorial 5

Solution: Secret! Shhhh... This is the solutions sheet.

Problem 1. True or False?

For each of the following, determine if the statement is True or False, justifying your answer with appropriate explanation.

a) The smallest element in a min heap is always the root.

b) The second largest element in a max heap with more than two elements (all elements are unique) is always one of the children of the root.

c) When a heap is stored in an array, finding the parent of a node takes at least $O(\log n)$ time.

d) Every node in the heap except the leaves has exactly 2 children.

e) We can obtain a sorted sequence of the heap elements in $O(n)$ time.

Solution:

a) True. Applying min heap property, all descendants of the root must be greater than the root.

b) True. Suppose second largest element is not the child of the root. Since it cannot be the root, it must be a descendant of the children of the root. However, this violates the max heap property. (Proof by contradiction)

c) False. Only requires $O(1)$ time by using the index. For a node at index i, parent is at index $\lfloor \frac{i}{2} \rfloor$ (using 1-indexed array).

d) False. A simple example is a heap with 4 elements. The left child of the root is not a leaf, but has only one child.

e) False. Heap sort takes $O(n \log n)$. (Also, note a common misconception that heapify sorts the array. This is **NOT TRUE**.)

Problem 2. Greater than x

Give an algorithm to find all vertices bigger than some value x in a **max heap** that runs in $O(k)$ time where k is the number of vertices in the output.
(This is different from most algorithms you have encountered which are dependent on the size of the input, instead of the size of the output. We sometimes call this output sensitive algorithms).

meaning elements inside it still have a chance of being larger than x but they will also be < key

Solution: Perform a pre-order traversal of the max heap starting from the root. At each node, check if node key is $> x$. If yes, output node and continue traversal. Otherwise terminate traversal on subtree rooted at current node, return to parent and continue traversal.

Algorithm 1 Solution to Problem 2

```
1: procedure FindNodesBiggerThanX(node, x)
2:     if node.key ≥ x then
3:         output node.key
4:         FindNodesBiggerThanX(node.left, x)
5:         FindNodesBiggerThanX(node.right, x)
6:     else
7:         return output (or terminate algorithm)
8:     end if
9: end procedure
```

Since the traversal terminates when it encounters that a node's key $\leq x$. In the worst case, it encounters $2k$ number of such nodes. That is, each of the left and right child of a valid node (node with key $> x$) are invalid. It will not process any invalid node other than those $2k$ nodes. It will process all k valid nodes, since there cannot be any valid nodes in the subtrees not traversed (due to the heap property). Thus the traversal encounters $O(k + 2k) = O(k)$ number of nodes in order to output the k valid nodes.

Problem 3. Updating a heap

Give an algorithm for the `update(int old_key, int new_key)` operation, which updates the value of `old_key` in a binary heap (max or min) with `new_key` in the binary heap in $O(\log n)$ time, which does not change the time complexity of the other operations. You are required to modify the other operations in the binary heap if needed, including any additional data structures used. You may assume all values in the heap will be unique. (Additional: What if the keys are not unique?)

Solution: Without any auxiliary data structure, in order to find `old_key` in the heap, you will need to search through the entire heap which will take $O(n)$ time. To support update in $O(\log n)$ time, we will need to use an extra Hash Table h. The key and value pair *(k, i)* for h will be the key k in the heap, and the index i of k in the heap (because array implementation) respectively.

We start by filling h with the corresponding key-value pairs by going through the entire input array. Now, whenever we call `swap` that is in `shiftUp` or `shiftDown`, we need to update both the values in the array, and the key-value pairs in h. Since heap creation, insert and remove are all dependent on `shiftUp` and `shiftDown`, there is no further modification required. For update, because we have The Hash Table h, we can search for the index of `old_key` in $O(1)$ time thus it's location in the heap. We update this key value pair: (`old_key`, i) $\Longrightarrow$ ((`new_key`, i)). Nowe we perform both `shiftUp` and `shiftDown` on `new_key`. Since the shift operations are still in $O(\log n)$ time, all the operations, including update, are in $O(\log n)$ time.

sliding window

Problem 4. Sorted? Almost. (Adapted from CS2040 AY19/20 Sem1 Final Exam)

A array A_k of n unique floating point values of no fixed precision is partially sorted when each value differs from its correct position in the sorted array by no more than k positions, where k is a positive integer. For example, an array A_2 could be $[1, 4, 3, 2, 6, 5, 7]$, where the sorted output should be $[1, 2, 3, 4, 5, 6, 7]$.

Problem 4.a. Give an algorithm to sort a partially sorted array in $O(n)$ time, where $k = 1$.

Solution: Iterate from the beginning of the array A_2 to the end. At each step i, check whether the current element at $A[i] > A[i+1]$. If yes, swap $A[i]$ and $A[i+1]$, else increment i. Essentially, this is insertion sort or 'single-pass' bubble sort.

Problem 4.b. Give an algorithm to sort a partially sorted array in $O(n \log k)$ time, where k can be any positive integer smaller than n.

Solution: The key observation here is that since every element is at most k away from its actual sorted position. What this means is that the element that should be placed in $A[i]$ (the i^{th} index of $A[1..n]$), is the smallest element in $A[i..(i+k)]$.

We use a min-heap, of size $k+1$ that keeps all elements in the sub-array $A[i..(i+k+1)]$ (Initialised with the first $k+1$ elements). Initially, i starts off at index 1, and we increase the index i by 1 as we iterate. Every time we do, we first extract the minimum value from the min-heap, since we know that this must be the value that should be at the i^{th} index. Then, we add the item at index $i+k+2$. Intuitively, you can view this as "sliding" a heap over a window of size $k+1$. An edge case is at the last $k+1$ elements. Since there will be no more elements to add after that, we simply repeatedly extract the minimum from the heap without adding anymore values into the min-heap.

to create the min heap

The algorithm requires $O(k)$ auxiliary space from the min-heap of size $k+1$. Heapifying the first $k+1$ elements take $O(k)$ time, and we do at most n sets of extract-min and insertion operations that are each in $O(\log k)$ time. The total runtime is $O(n \log k + k)$, or $O(n \log k)$ since $k < n$.

Problem 5. Which number to pick? sliding window

The Great Overlord of Arithmetic has a challenge for you! He gives you a stack of n integers, and allows you to repeatedly perform the following operation:

- From any of the top k integers in the stack, you may remove one of them and add it to the Fundamental Pool of Arithmetic.

The Great Overlord of Arithmetic wants you to calculate the maximum total value of numbers that can possibly be placed in the Fundamental Pool of Arithmetic (starting from 0). Give an efficient algorithm to do this, and state the time complexity.

this is same sliding window concept as qn 4

extract first k elements to create a max heap and then keep extracting max to add to the pool then pop another value from the stack to add into the max heap

For example, for $k = 2$, where integers in the stack are $[2, -10, 2, -6, 5]$, the output of your algorithm should be 4. Another example for $k = 5$, where integers in the stack are $[-1, -1, -1, -1, -1, 10]$, the output of your algorithm should be 9.

Solution: Since we can only look at the top k elements in the stack, and we want to maximise the total value of elements chosen, we make use of a maximum heap of size k. We need to maintain two values, the current / running sum, and the historical highest sum achieved.

the sum in the pool and the historical highest we've found

We will remove the largest element in the heap, followed by inserting the next element in the stack. This simulates the top k elements in the stack at any one point in time. When removing the largest element, we will also add that value to the current sum, and update the historical highest sum when necessary. Note that since it is possible that all the values in the heap of size k are negative, the current sum can be reduced. This is why we need to keep track of the historical highest sum. We repeat this step for $n-k$ times until the last element is added to the heap.

Note that since there are k elements remaining that could have positive values, we also need to check the remaining elements in the heap and update the sums as necessary.

Algorithm 2 Solution to Problem 5

```
1: Let A be the stack of integers
2: Initialise maximum heap D
3: for i = 1 to k do
4:     Insert A.pop() into D
5: end for
6: max_sum = 0
7: current_sum = 0
8: for i = k + 1 to n do
9:     current_sum = current_sum + D.extractMax()
10:    Insert A.pop() into D
11:    if current_sum > max_sum then
12:        max_sum = current_sum
13:    end if
14: end for
15: while D is not empty and D.getMax() > 0 do
16:    current_sum = current_sum + D.extractMax()
17:    if current_sum > max_sum then
18:        max_sum = current_sum
19:    end if
20: end while
21: return max_sum
```

extractMax()

Each of the n values in the stack will at most be inserted once and removed once from the heap of size k. Since each operation takes $O(\log k)$, the total time complexity of $O(n \log k)$. The algorithm requires $O(k)$ auxiliary space for the maximum heap of size k.