CMPSC 465 Data Structures & Algorithms
Fall 2025 Ke Chen and Yana Safonova

Worksheet 3

Monday, Sep 22, 2025

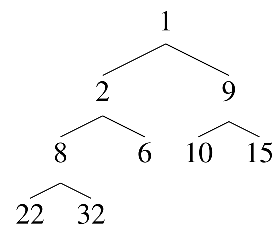
1. **Delete Min** Please consider the following array, which represents a min heap.

$$A = [1, 2, 9, 8, 6, 10, 15, 22, 32].$$

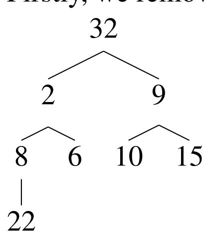
Suppose we remove the element at position 0 of the heap. How does the resulting heap look? Write both the array and tree representation of the heap.

Solution:

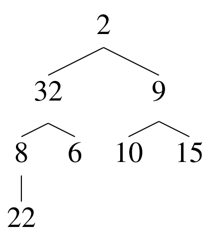
Our initial tree looks like this:



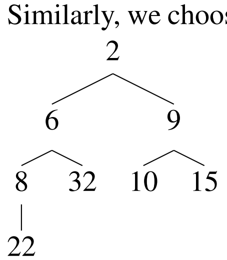
1. Firstly, we remove the node 1, and replace it with the last node i.e. 32.



2. Then we run Heapify-Down from the root. That is, we choose the smaller of the children, in this case between 2 and 9 is *2*. This element is now swapped with 32.



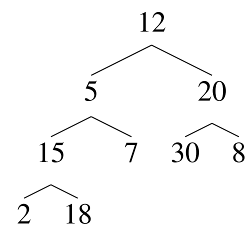
3. Similarly, we choose the smaller number between 8 and 6, then we swap 32 to give us the result.



The array representation for this heap is $[2,6,9,8,32,10,15,22]$.

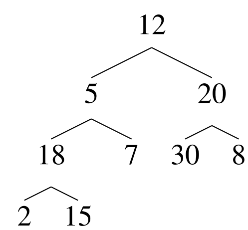
2. Heapify. How does the heap look like after we run the Build-heap function on the array $A[1...9] = [12,5,20,15,7,30,8,2,18]$?

Solution: Our initial tree looks like this:

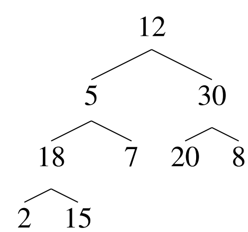


We run Max-Heapify-Down on this tree, starting from node $\lfloor \frac{n}{2} \rfloor$ down to 1 as follows:

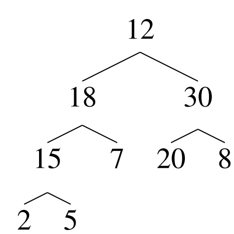
1. We swap node 15 and 18.



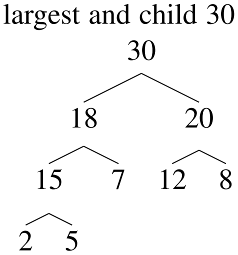
2. Then we swap 20 and 30.



3. Next, we swap 5 and 18 and later we swap 5 and 15 to obtain the following tree.



4. Finally, for the root 12, we first swap it with its largest and child 30 followed by swapping 12 with 20. The final tree:



We got the max-heap, so we do not need to do anything further.

3. **Node Count.** Show that there are at most $\lceil \frac{n}{2^{h+1}} \rceil$ nodes of height h in any n-element heap.

Proof: We prove this by induction on the height of the nodes. Let, n_h be the number of nodes at height h.

Base Case: The leaves of a heap ($h = 0$) are the nodes indexed by

$$\left\lfloor \frac{n}{2} \right\rfloor + 1,\ \left\lfloor \frac{n}{2} \right\rfloor + 2,\ \ldots,\ n,$$

which corresponds to the second half of the heap array (plus the middle element if n is odd). Hence the number of leaves is

$$n_0 = \left\lceil \frac{n}{2} \right\rceil.$$

So, the base case holds for $h = 0$.
Now assume the claim holds for height $h-1$. We show it holds for height h.

Case 1. If n_{h-1} is even, each node at height h has exactly two children. Thus

$$n_h = \frac{n_{h-1}}{2} = \left\lfloor \frac{n_{h-1}}{2} \right\rfloor.$$

Case 2. If n_{h-1} is odd, one node at height h has one child and the rest have two children. Thus

$$n_h = \left\lfloor \frac{n_{h-1}}{2} \right\rfloor + 1 = \left\lceil \frac{n_{h-1}}{2} \right\rceil.$$

In both cases,

$$n_h \leq \left\lceil \frac{n_{h-1}}{2} \right\rceil.$$

Applying the inductive hypothesis:

$$n_h \leq \left\lceil \frac{n_{h-1}}{2} \right\rceil \leq \left\lceil \frac{1}{2} \cdot \left\lceil \frac{n}{2^{h-1+1}} \right\rceil \right\rceil = \left\lceil \frac{n}{2^{h+1}} \right\rceil.$$

Thus the statement holds for all h. □

4. **K-th Largest Element.** Given an array A of n elements, find the k-th largest element.

Solution: We can solve this problem efficiently using a min-heap data structure. The main idea is to maintain a min-heap of size k that stores the k largest elements encountered so far.
Note that in the following algorithm, $H.peek()$ returns the root of the min-heap (minimum element).
Algorithm:

1. Create a min-heap data structure, H.
2. Iterate through the first k elements of the input array A and insert them into the min-heap.
3. Iterate through the remaining elements of the array, from index k+1 to n.
4. For each element A[i], compare it with the root of the min-heap (H.peek()).
5. If $A[i] > H.peek()$, then remove the root of the heap ($H.extract\_min()$) and insert $A[i]$ into the heap ($H.insert(A[i])$).

6. After iterating through all the elements, the root of the min-heap will be the k-th largest element. Return $H.peek()$.

Pseudocode:

```
function find_kth_largest(A, k):
    H = new MinHeap()
    for i from 1 to k:
        H.insert(A[i])
    for i from k+1 to n:
        if A[i] > H.peek():
            H.extract_min()
            H.insert(A[i])

    return H.peek()
```

Complexity Analysis:

- **Time Complexity:** Inserting the first k elements into the heap takes $O(k \log k)$ time. The loop for the remaining $n-k$ elements performs a constant number of heap operations (peek, extract_min, insert), each taking $O(\log k)$ time.The total time complexity is $O(k \log k + (n-k) \log k) = O(n \log k)$.
- **Space Complexity:** The min-heap stores at most k elements, so the space complexity is $O(k)$.