Solution Set

CIS 1210—Data Structures and Algorithms—Fall 2024

Heaps—Tuesday, October 1 / Wednesday, October 2

Readings

- Lecture Notes Chapter 14: Binary Heaps and Heapsort

Review: Heaps

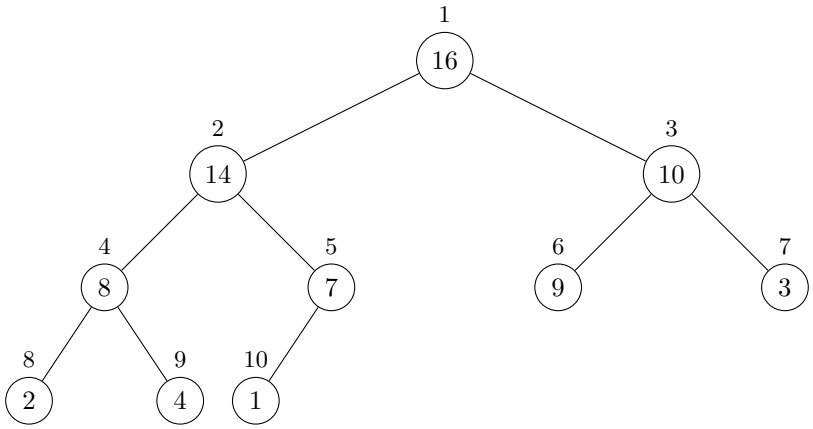
A heap is a tree-like data structure that implements the priority queue abstract data structure (ADT), which allows us to maintain a set of elements, each with an associated key, and select the element with the highest/lowest priority. Heaps satisfy the two following properties:

Heap Property: In a max-heap, for each node i, we have $A[\textsc{Parent}(i)] \geq A[i]$, so the maximum value is stored at the root. In a min-heap, for each node i, we have $A[\textsc{Parent}(i)] \leq A[i]$, so the minimum value is stored at the root.

Shape Property: A heap is an almost complete binary tree, meaning that every level of the tree is completely filled except for the last, which must be filled from left to right.

Implementation Details

Because a heap is an almost complete binary tree, we are able implement it using an array with 1-indexing as shown below:



Index	0	1	2	3	4	5	6	7	8	9	10
Value	`null`	16	14	10	8	7	9	3	2	4	1

Observe that we can populate the array from left to right by doing a level-order traversal of the tree, where we start from the root and go through each level of the tree from left to right. Additionally, because of the shape property, if the root is stored at index 1 of the array, given a node at index i, its left child can be found at index $2i$, its right child can be found at index $2i + 1$, and its parent can be found at index $\lfloor i/2 \rfloor$.

Operations (Max-Heaps)

Max-Heapify maintains the max-heap property at the node called on, so the entire subtree rooted at the node will now be a max-heap. It assumes the node's left and right subtrees are both valid max-heaps, and then allows the node to "float-down," swapping it with its larger child or terminating if the max-heap property holds. It runs in $O(h)$ time, where h is the height of the node, since in the worst case, the node must "float down" to the bottom of the tree. Since the height of any node is upper bounded by $\log n$, Max-Heapify runs in $O(\log n)$ for any node (though this bound may not be tight for some nodes, which is a property we leverage when analyzing the runtime of Build-Max-Heap).

Build-Max-Heap constructs a max-heap from an unsorted array by repeatedly calling Max-Heapify on nodes from the "bottom-up", starting at the nodes right above the leaves (which by definition are max-heaps!). It runs in $O(n)$ time; the mathematical proof of this upper-bound can be found here.

Extract-Max removes and returns the element with the maximum key. We remove the root, replace it with the right-most element in the bottom level/last element in the array, and then call Max-Heapify on the "new" temporary root to maintain the max-heap property. We perform constant work besides calling Max-Heapify, so it runs in $O(\log n)$ time.

Insert adds an element by first adding it to the end of the array/max-heap, and then allowing it to "float-up" to its correct position by repeatedly swapping it with its parent as necessary to maintain the max-heap property. It runs in $O(\log n)$ time, since the path it takes while it "floats-up" has length $O(\log n)$.

Peek returns the maximum element in the heap stored at the root. Since we implement a heap with an array, this runs in $O(1)$ time because we just index into the array.

Problems

Problem 1

Given a data stream of n test scores, design an $O(n \log k)$ time algorithm to find the k-th highest test score. Since PEFS provides minimal monetary resources, CIS 1210 Staff has limited access to storage space and can only afford you $O(k)$ space, where $k \ll n$.

Solution

Algorithm: Construct a min-heap from the first k tests, where tests are ordered by their score, by calling Build-Min-Heap. For each remaining test in the data stream: if its score is greater than the score at the root of the heap, remove the root by calling Extract-Min and then Insert the current test; otherwise, the score of the current test is less than or equal to the score at the root, so do nothing. After processing all tests in the data stream, return the score at the root of the heap by calling Peek.

Proof of Correctness: We will prove the algorithm's correctness by loop invariant. Namely, we will show that we maintain the invariant that the heap always contains the highest k scores we have seen thus far.
Initialization: By calling BuildHeap on the first k, the heap contains the only and thus highest k that have been seen in the stream.
Maintenance: Consider iteration i where we have an element e in the stream. Suppose that the heap contains the highest k until iteration $i-1$. Let m be the result of Peek (the minimum of the heap). There are two cases:

> Case 1: e is in the highest k seen up to iteration i. Thus the minimum of the heap (m) is not in the highest k and can be removed. Because of the loop invariant being held up to this point (highest k until iteration $i-1$), we know that the heap will contain the other $k-1$ highest elements. Thus, the algorithm removing m and inserting e will maintain the loop invariant.

Case 2: e is not in the highest k seen up to iteration i. $e < m$ because e is not in the highest k. The algorithm will not remove element m, which maintains the loop invariant.

Note that in both cases we also maintain the property that the heap holds exactly k scores.
Termination: After the algorithm processes all other $n - k$ elements in the stream, the heap will contain k elements which hold the above invariant.

At the end of the iteration, the heap will contain the k highest scores, the minimum of which will be the k-th highest. This is exactly the score desired.

Runtime Analysis: Constructing a min-heap from the first k tests by calling BUILD-MIN-HEAP takes $O(k)$ time. For each remaining test, we either do nothing, or we maintain the heap at size k by calling EXTRACT-MIN and then INSERT on the current score, which takes $O(\log k)$ time. Each test is inserted into the heap at most once, and since the data stream has n scores, our overall running time is $O(k + n \log k)$. Since $k \ll n$, our final running time is $O(n \log k)$.

Space Analysis: As a "pre-processing" step, we first construct a min-heap with the first k tests. We maintain the heap at size k because for each remaining test that we insert, we remove the root. Therefore, the space complexity of our algorithm is $O(k)$.