CSC 265: Enriched Data Structures **Fall 2018**

Tutorial 1: Heap Data Structure

Tutor: Lily Li *Date: 13 September*

1.1 Background

We want to implement priority queue (ADT) using a Max-Heap. Every priority queue P supports two main operations: Insert(P, x) and Extract-Min(P) or Extract-Max(P) depending on whether the largest or smallest value has highest priority. Both operations have running time $O(\log n)$, where $n = |P|$.

1.2 Maintaining the Heap Property

Heaps can be seen as a complete binary tree, but in practice the elements are stored in a list. The left and right children of the element at index i are at indices $2i$ and $2i + 1$ respectively. See Figure 1.1. The **height** of a tree is the length of its longest root-to-leaf path.

Q: What is the height of a tree with n nodes?
A: About $\log n$.
Q: What is the maximum and minimum number of elements in a heap of height h?
A: The maximum is $2^{h+1} - 1$ while the minimum is 2^h.

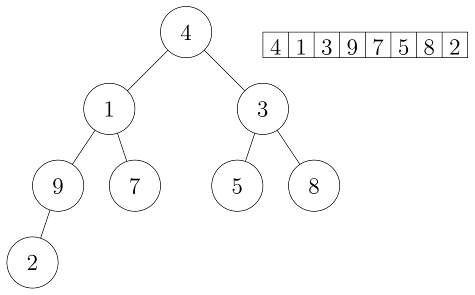

Figure 1.1: Consider the above example. Is this a heap? Is the tree rooted at the node 9 a heap?

Every heap maintains the heap property which states that a value stored at a non-root is at most the value of its parent. It follows then that the root has the maximum value, assuming that values in the heap are unique. Further any path from root to leaf is in non-increasing order.

We will use a function called Max-Heapify to fix the heap if it does not satisfy the heap property.

1.3 The Max-Heapify Procedure

The Max-Heapify(A, i) procedure assumes that the binary trees rooted at Left(i) and Right(i) are heaps. Upon termination, the sub-tree rooted at index i is a heap as well.

```
MAX-HEAPIFY($A, i$)
1   $l \leftarrow$ LEFT($i$)
2   $r \leftarrow$ RIGHT($i$)
3   $n \leftarrow$ HEAP-SIZE($A$)
4   if $l \leq n$ and $A[l] > A[i]$
5         $largest \leftarrow l$
6   else $largest \leftarrow i$
7   if $r \leq n$ and $A[r] > A[largest]$
8         $largest \leftarrow r$
9   if $largest \neq i$
10        SWAP($A[i], A[largest]$)
11        MAX-HEAPIFY($A, largest$)
```

The procedure compares the value at i to its left and right children. If a child of i has a greater value, then it is swapped with i and a recursive call is made one level below in the tree.

What is the worst-case running-time of MAX-HEAPIFY? Observe that at most h recursive calls are made, where h is the height of the sub-tree starting at i. Further, each call to MAX-HEAPIFY takes C steps for some constant C independent of the input. Therefore, for all inputs, *at most* Ch steps are needed. The worst case time complexity is $O(h)$ or $O(\log n)$ since h is at most the height of the entire tree.

1.4 Building a Max-Heap

With MAX-HEAPIFY at hand, given a list A, we can make it into a heap using the following algorithm

```
BUILD-MAX-HEAP($A$)
1   $n \leftarrow$ LENGTH($A$)
2   for $i \leftarrow \lfloor n/2 \rfloor$ downto 1
3         MAX-HEAPIFY($A, i$)
```

The idea is to apply MAX-HEAPIFY to all non-leaf nodes in the tree starting at the bottom.

A naïve analysis of the algorithm yields a worst-case running time of $O(n \log n)$ since there are $O(\log n)$ calls to MAX-HEAPIFY, but this is rather pessimistic. Lets look at the running time needed at each level of the tree. Observe that if we run MAX-HEAPIFY on a node whose sub-tree is of size h, then the running-time is $O(h)$ so we want to get a handle on the number of nodes with subtrees of a certain height. I claim that there are at most $\frac{n}{2^h}$ nodes in a complete binary tree with subtree of height h — it is a good exercise to try and prove this. Thus, a better bound on the running-time is as follows:

$$T(n) \leq \sum_{h=0}^{\lfloor \log n \rfloor} \frac{n}{2^h} O(h) = nC \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} \leq nC \sum_{h=0}^{\infty} \frac{h}{2^h}$$

To complete the analysis, we need to bound $\sum_{h=0}^{\infty} \frac{h}{2^h}$. Recall that the closed form of a geometric series is

$$\sum_{i=0}^{\infty} x^i = 1 + x + x^2 + \cdots = \frac{1}{1-x}$$

for $|x| < 1$. Let $x = \frac{1}{2}$ and make the following observation about our desired series

$$\begin{aligned}\sum_{h=0}^{\infty} hx^h &= 0 \cdot 1 + 1 \cdot x + 2x^2 + 3x^3 + 4x^4 \cdots \\ &= x\left(1 + 2x + 3x^2 + 4x^3 + \cdots\right) \\ &= \frac{xd(1 + x + x^2 + \cdots)}{dx} \\ &= x \cdot \frac{d\left((1-x)^{-1}\right)}{dx} \\ &= \frac{x}{(1-x)^2}\end{aligned}$$

For $x = \frac{1}{2}$ we have $\sum_{h=0}^{\infty} \frac{h}{2^h} \leq 2$ so we can bound the running-time of Build-Max-Heap by

$$T(n) \leq nC \sum_{h=0}^{\infty} \frac{h}{2^h} \leq 2nC.$$

It follows that $T(n) \in O(n)$. To show that the worst-case running time is $\Omega(n)$ we need to produce a instance of A which runs in time $\Omega(n)$. To that end, consider $A = [1, 2, ..., n]$. All told, the time complexity of Build-Max-Heap is $\Theta(n)$.

1.5 Application of Heaps

1.5.1 Heap-Sort

How can we use a heap to sort efficiently? What is the running-time and space requirement of your algorithm? If we had a heap, then it would be easy. Simply construct the sorted list by extracting the maximum element. So build a heap!... But we should do it carefully to reduced the space requirement.

```
Heap-Sort($A$)
1  Build-Max-Heap($A$)
2  for $i \leftarrow$ Length($A$) downto 2
3      Swap($A[1], A[i]$)
4      Heap-Size($A$) $\leftarrow$ Heap-Size($A$) $- 1$
5      Max-Heapify($A, 1$)
```

Consider the execution of the algorithm on list $A = [4, 1, 3, 9, 7, 5, 8, 2]$ as before.

1.5.2 Streaming Algorithm: Maintain Median

The problem set-up is as follows: you are going to get a stream of integers — you can assume these are unique — $x_1, ..., x_n$. At every instance i, I want to know the median of the first i numbers. Your task is to return the median in time $O(\log i)$.

The key idea is to use two heaps, a Max-Heap which store the smaller half of the elements and a Min-Heap which stores the larger half of the elements.