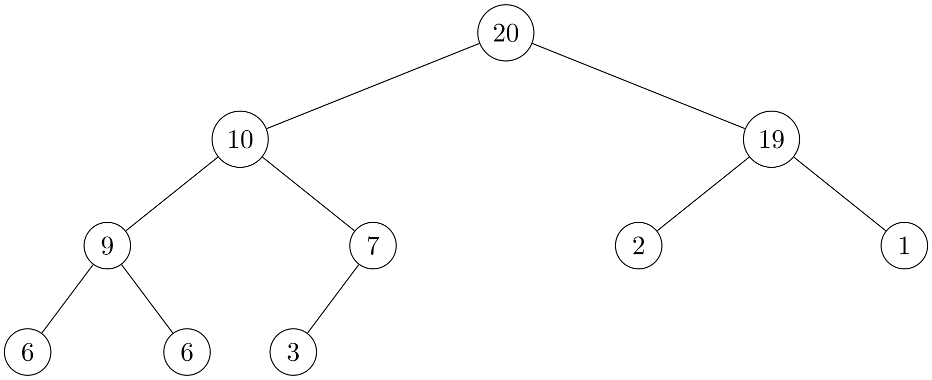
CS 240 — LECTURE 5

Bartosz Antczak *Instructor: Eric Schost* *January 17, 2017*

5.1 The Height of a Heap

Example 5.1.1. *Consider the following heap:*



We observe that the height of the heap is 3, and there are 10 nodes. There is/are:

- 1 node at level 0
- 2 nodes at level 1
- 4 nodes at level 2

We also observe that there are between 1 and 8 nodes at level 3. This means that for a heap of height $h = 3$, the number n of nodes satisfies

$$1 + 2 + 4 + 1 \leq n \leq 1 + 2 + 4 + 8$$

(i.e., the least number of nodes this heap can have is $1 + 2 + 4 + 1$, and th max number it can have is $1 + 2 + 4 + 8$).
In general, we have

$$1 + 2 + 4 + \cdots + 2^{h-1} + 1 \leq n \leq 1 + 2 + 4 + \cdots + 2^{h-1} + 2^h \tag{5.1}$$

$$2^h \leq n \leq 2^{h+1} - 1 \leq 2^{h+1} \tag{5.2}$$

$$h \leq \log_2(n) \leq h + 1 \tag{5.3}$$

$$\log_2(n) - 1 \leq h \leq \log_2(n) \tag{5.4}$$

On line 5.2, we used the identity $\sum_{i=1}^{n} 2^i = 2^{n+1} - 1$. From line 5.4, we see that the time it takes to traverse the height of a heap with n nodes is $\in \Theta(\log_2 n)$.

5.2 Building Heaps Using Arrays

To work with heaps in code, we store it in an array. Consider a heap H of n items. We'll create an array A of size n. We store the root in $A[0]$, and we continue with the elements level-by-level from top to bottom, in each level left to right. To access

- the left child of $A[i]$, go to $A[2i+1]$
- the right child of $A[i]$, go to $A[2i+2]$
- the parent of $A[i]$ ($i \neq 0$), go to $A[\lfloor \frac{i-1}{2} \rfloor]$

The array implementation of the heap from example 5.1.1 would be $A = [20, 10, 5, 9, 7, 2, 1, 6, 6, 3]$.

5.2.1 Various Algorithms to Implement Heaps

Problem: Given n items in $A[0 \cdots n-1]$, build a heap containing all of them.

Approach 1

This approach starts with an empty heap and inserts nodes one at a time. Analysing this algorithm, we see that the worst case running time will involve *bubbling-up* on every insertion. This means that inserting the ith element may take $\log_2(i)$ swaps (recall that this defines the time it takes to traverse the height of a heap), which means that the worst case running time is

$$\Theta\left(\sum_{i=1}^{n} \log_2(i)\right)$$

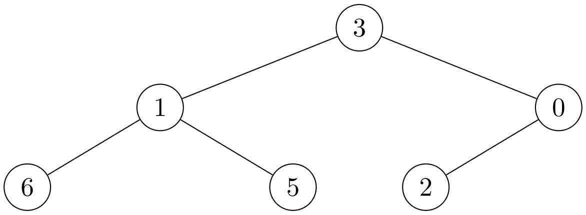
$$\Theta(\log_2(n!))$$

$$\Theta(n \log_2 n)$$

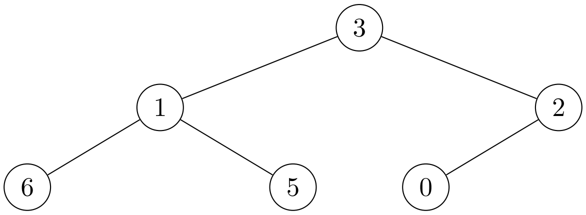
Approach 2

Remember that we're working with heaps structured as arrays. So instead of manually inserting one node at a time, it would seem faster to simply take an arbitrary array, and apply the *bubble-down* algorithm on the first $\frac{n}{2}$ nodes, starting with the node at index $\lfloor \frac{n}{2} \rfloor$, going down to the root node (index 0). Applying this bubble-down implementation grantees that this array will be a proper heap.

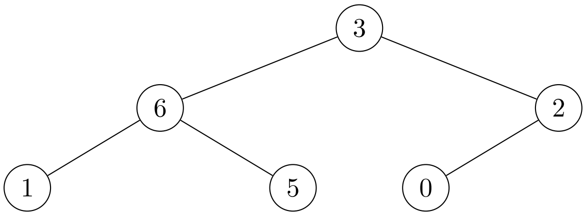
Example 5.2.1. *Applying this algorithm on an arbitrary array (in heap form):*



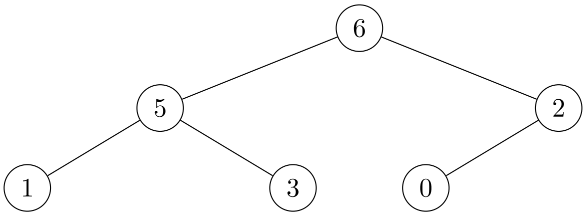
1. We start with an arbitrary heap.



2. Bubble down on index index two (which is $\lfloor \frac{n}{2} \rfloor$ — node "0")



3. Bubble down on index one — node "1"



4. Finally, bubble-down on index zero — node "3"

Analyzing this algorithm on a tree of height 3, we see that in the worst case running time

- On level $h = 3$, we perform 0 swaps (since all nodes are leaves in this level)
- On level $h - 1 = 2$, we perform 4×1 swaps
- On level $h - 2 = 1$, we perform 2×2 swaps
- On level $h - 3 = 0$, we perform $h = 3$ swaps

In general, the number of swaps is at most

$$(1 \cdot h) + (2 \cdot (h-1)) + (4 \cdot (h-2)) + \cdots + (2^{h-1} \cdot 1)$$

which can be simplified to

$$\begin{aligned}
&= (2^0 \cdot (h-0)) + (2^1 \cdot (h-1)) + (2^2 \cdot (h-2)) + \cdots + (2^{h-1} \cdot (h-(h-1))) \\
&= \sum_{i=0}^{h-1} 2^i \cdot (h-i) \\
&= 2^h \sum_{i=0}^{h-1} 2^{i-h} \cdot (h-i) \\
&= 2^h \sum_{i=0}^{h-1} \frac{h-i}{2^{h-i}} \\
&\leq 2 \leq 2 \cdot 2^h \leq 2n && \text{(From equation 5.4)}
\end{aligned}$$

Thus, the runtime of this algorithm is $\in \Theta(n)$.

5.3 Intro to Selection

The **selection problem** states:

> *Given an array A of n numbers, and $0 \leq k \leq n$, find the element in position k of the sorted array (a.k.a. the k-th largest number in A)*

Consider input array $A = [3, 2, 8, 7, 6, 11, 12, 22, 1]$

5.3.1 Quick-select and Quick-sort

These two algorithms are used to sort an array. They rely on two important subroutines (in linear time):

- choose-pivot
- partition

More on this in the next lecture