Heaps - Part II

Note Title 11/6/2007

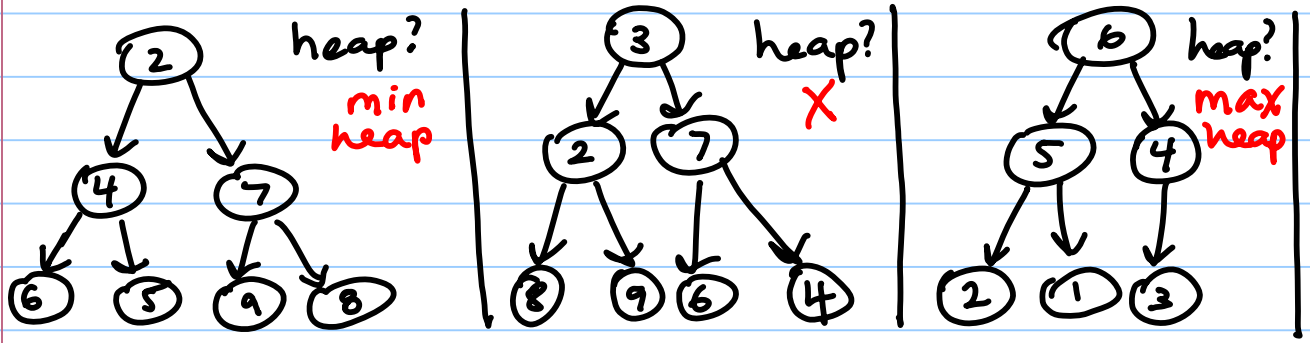
Heaps Recap

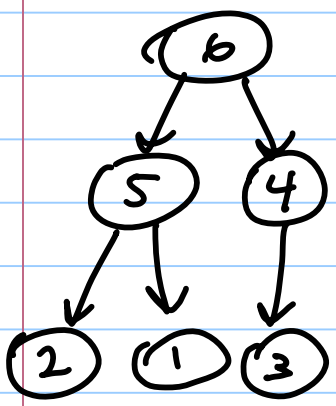
A heap is a completely filled binary tree
(ok, maybe not full leaves)
and either one of

– Max-Heap
For all nodes, value of parent ≥ value of self

OR

– Min Heap
For all nodes, value of parent ≤ value of self





How long to find (1) in the heap?

worst-case search: n

worst-case height: log n

What is easy to find in a max-heap?

max is O(1)

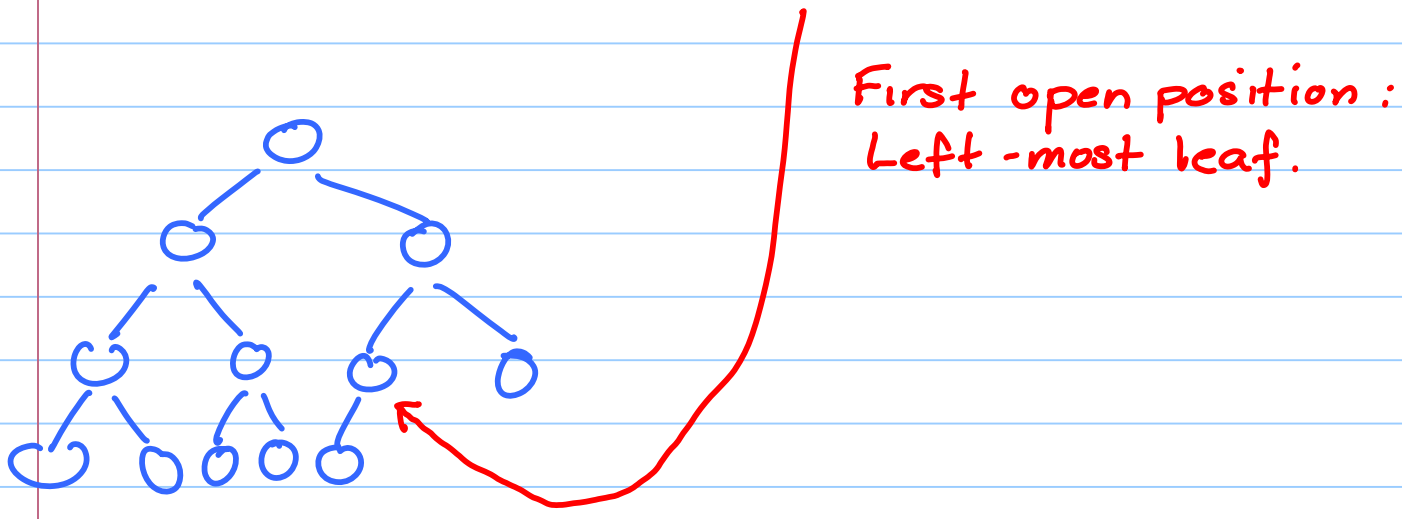
Typical Heap Questions

- What is the height of the Heap?
- How long does it take to find a value?
- How long does it take to insert a value?
- How long does it take to delete a value?

- log n for first question
- n for second question
- what about the last two?

Heap: Insert (val)

Due to the structure of the heap, you always know where the next new node will be placed.



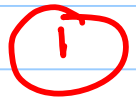
The challenge is to maintain [in O(log n) time] the heap's overall property of subtrees having contents less than (or for a MinHeap greater than) the local root node.

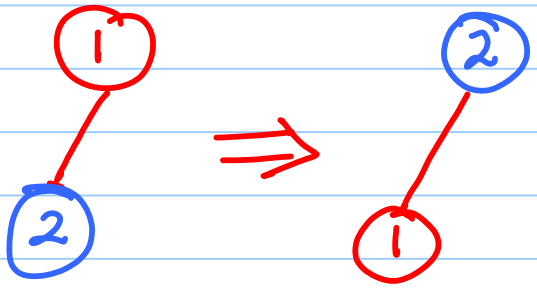
Let's look at how MaxHeap would be built using the values:

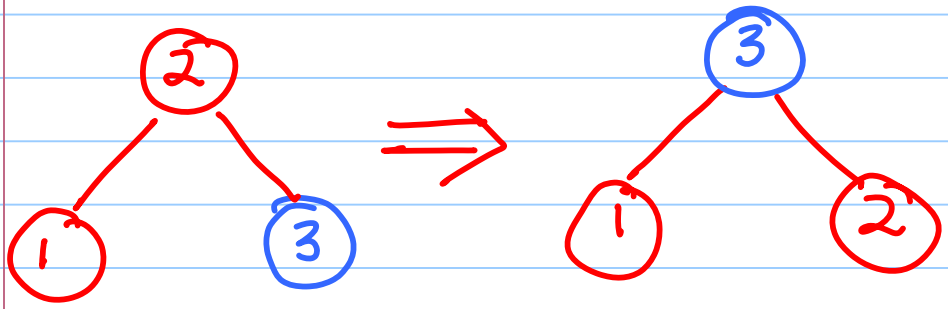
1, 2, 3, 4, 5, 6

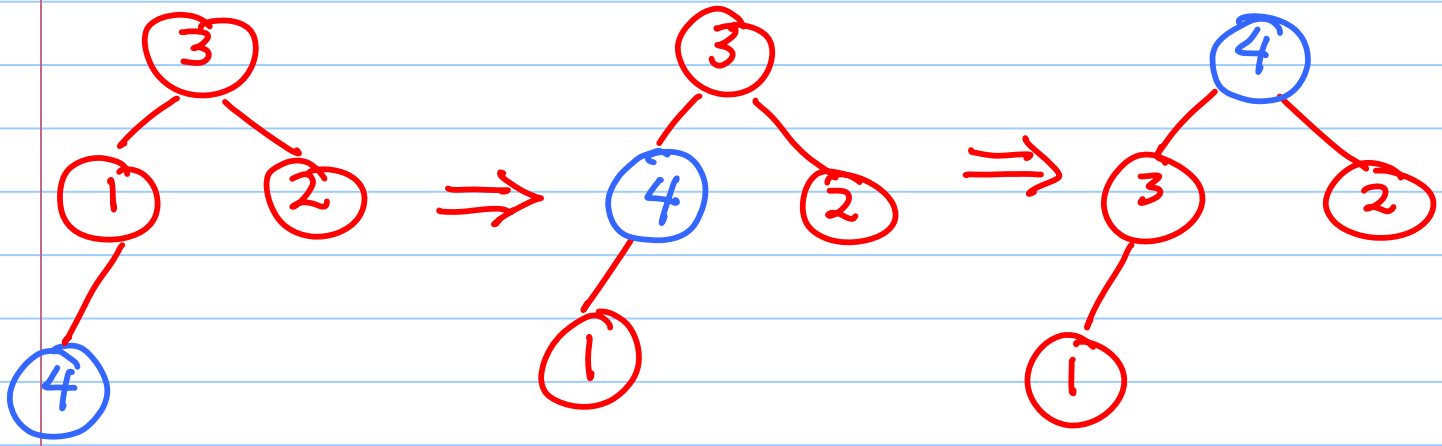
Build-Max Heap

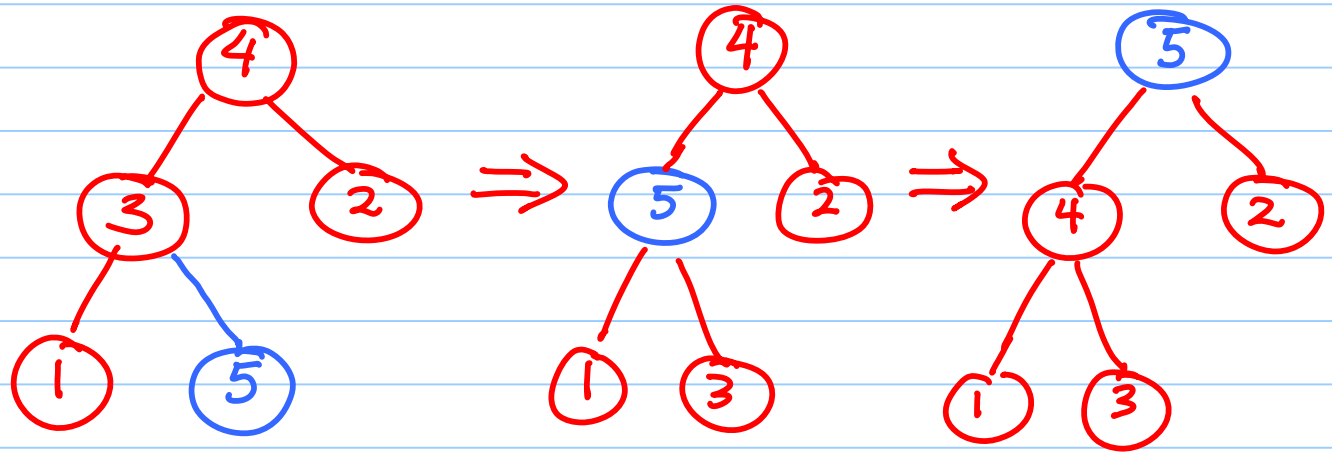
Input: 1, 2, 3, 4, 5, 6











- Note that the key to maintaining the max-heap after each insertion is to "bubble" the newly added value upward.

- How long does "bubbling" take?
  $O(\log n)$

- How many times do we "bubble"?
  $O(n)$

- So Build-Max Heap is $O(n \log n)$.
  We'll look at this again shortly.

- What about Insert?
  Just "bubble" once: $O(\log n)$

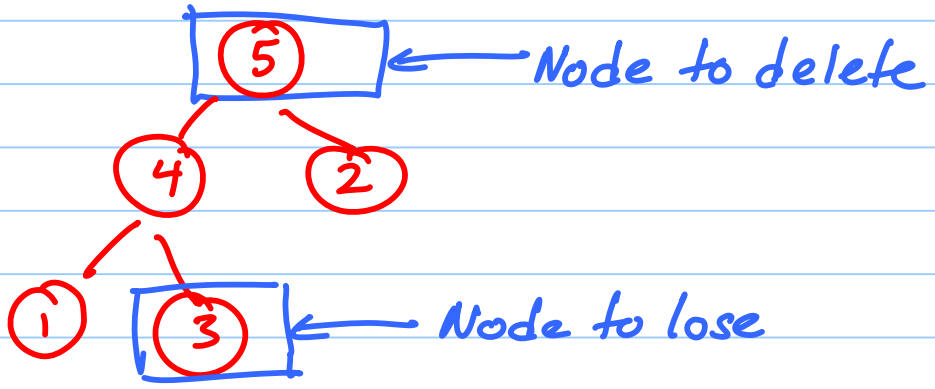
What about best case?

Heap: DeleteRoot

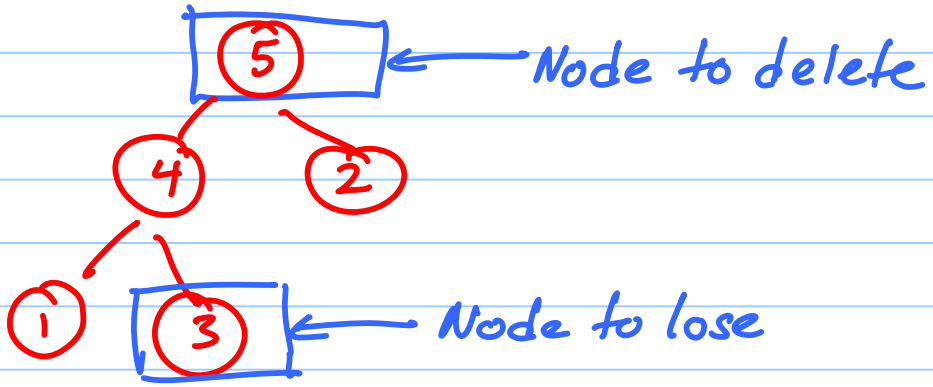
Due to the structure of the heap, you always know where the next new node (the root) will be dropped.

Once again, the challenge is to maintain [in O(log n) time] the heap's overall data property.

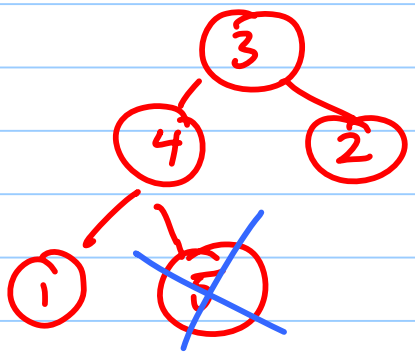
Let's look at what it would be like to delete the root of Max Heap.



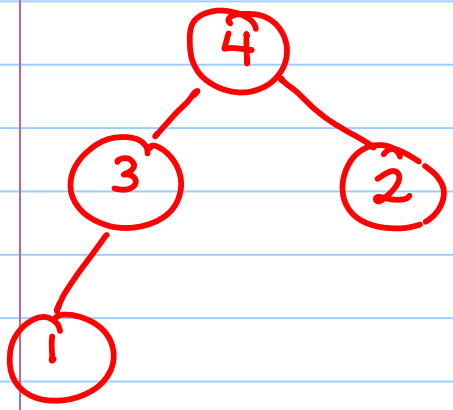
Can't just delete root!



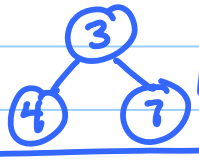
Start by swapping "3" with root + drop node



Now "bubble" the root downward until it is in a non-violating spot.



- When we bubble down, there is a choice!
- What if we had 3 Instead of 2?



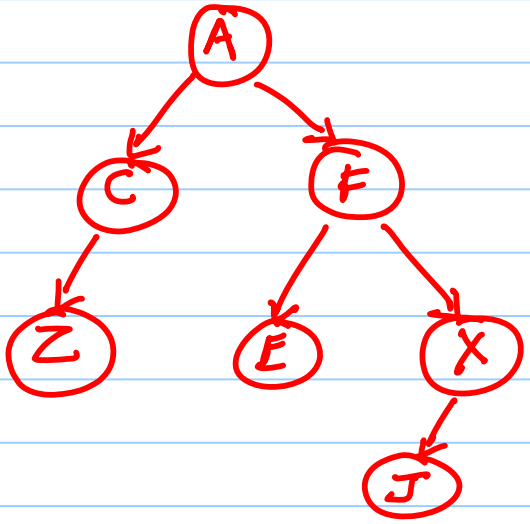
How much time does "bubbling" take?

2 comparisons
log n height
} 2 × log n

$O(\log n)$

We will refer to this "bubble downward" operation as Max-Heapify (p. 130 of book)
And we will use it again in Build Max Heap and HeapSort

Array Storage Example



Is this a heap?

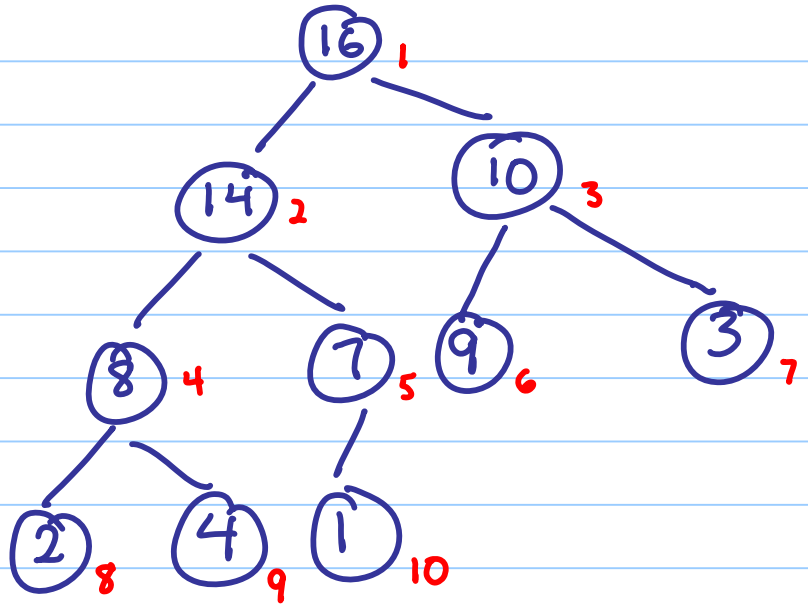
A	C	F	Z	-1	E	X	-1	-1	-1	-1	-1	-1	J	-1	....

When used with complete trees such as heaps, there are no gaps within the tree — only empty spaces on the extreme right.

Example

Max Heap Viewed as . . .

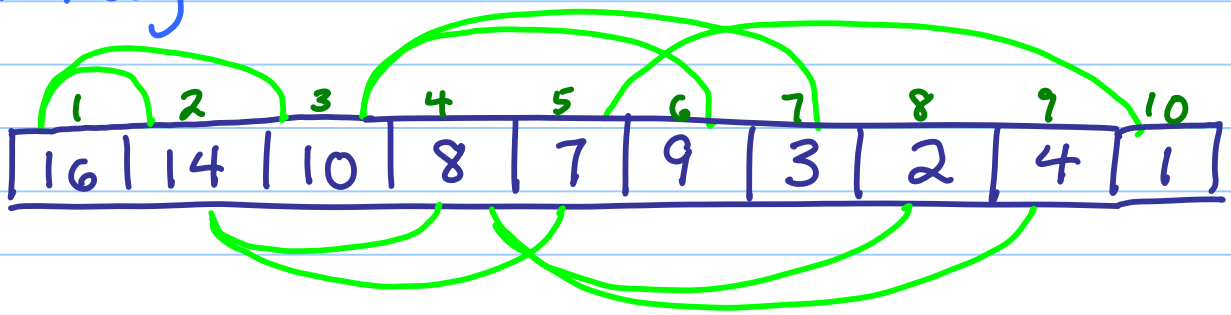
Binary Tree



Number in circle is value at that node

Number in red is is the corresponding index in the array

Array



Index of left child = 2 x index of parent
Index of right child = (2 x index of parent) + 1

Note the "immediate access" benefit of this choice of storage structure.

↓

Heap as Priority Queue

One use of a heap is that of Priority Queue. It has good run-times for insertion and deletion, and provides instant access to the next item in the queue.

Example: Printer Queue.

Start printing top job right away (and then fix the structure after deleting).

HeapSort

Since the heap provides an easy way to extract values in their relative order, it is trivial to create an $O(n \log n)$ sorting algorithm using a heap.

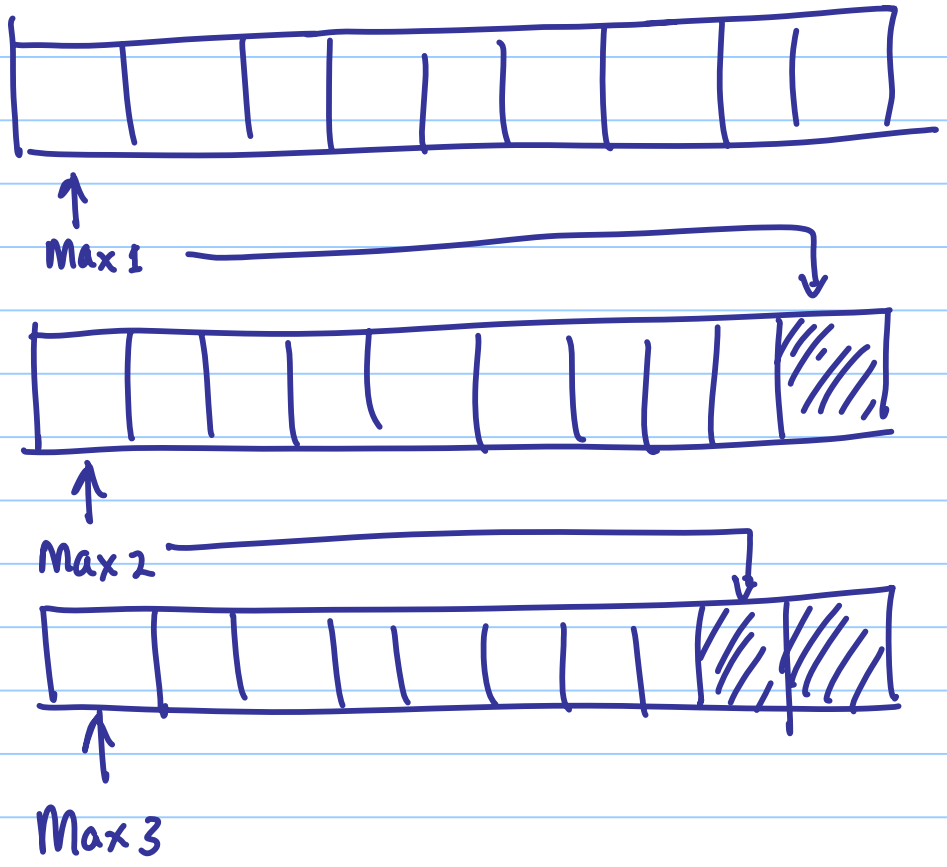
- Build the Heap (Build-Max Heap)
- Harvest the heap (Take Max, Heapify, Take Max, Heapify, Take Max ...)

Some questions to consider:

- Can we sort an array of values without using a linear amount of extra memory?

An "array trick" allows us to build in place.

Array Trick



etc.

- Is the worst-case runtime actually better than $n \log n$?

Let's re-examine Build-Max Heap using Max-Heapify

↓

Max-Heapify

- Let's assume MaxHeap.
- Suppose you have a complete tree where the root is the only thing violating the heap property. (This should remind you of how we implemented DeleteRoot, where we put the right-most leaf node into the root position.)
- You can "bubble" that value down to a valid position.
  - If the value of the root is smaller than either child, swap it with the larger of the children.
  - Repeat until it is no longer smaller than either of its two children.
- What is the runtime of this?

  ↓

  → Height at which you run it determines the runtime: $1 \leftrightarrow \log n$

Build-Max Heap (Revisited)

- Can we use Max-Heapify (instead of "bubbling upward") to Build-Max Heap in an array, in-place?

↓

YES!

- Build-Max Heap in place, starting at the bottom of complete binary tree, call Max-Heapify as you move upward.

Note: Each call to Max-Heapify has a cost of $O(\log n)$

- Leaves represent subtrees with heap property satisfied, So we can run Max-Heapify on others "bottom up"

↑ or does it?

Build-Max Heap Analysis – the "easy version"

- Easy to call in an array.
- Process $\frac{n}{2}$ nodes (no need to process leaves)
- Each call to Max Heapify is $O(\log n)$
- So $O(\frac{n}{2})$ calls, each $O(\log n) \Rightarrow O(n \log n)$

Is it really this bad ??

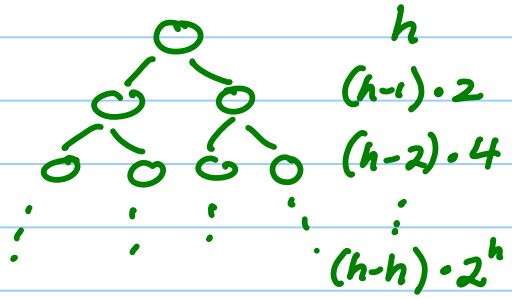
Note: Does not take advantage of the places where we call Max-Heapify on trees with a very small height.

Can we get an asymptotic improvement with a more thorough analysis?

- We should be more careful with our analysis
- MaxHeapify is technically not always $\log n$.
- It is whatever height (h) we are at!

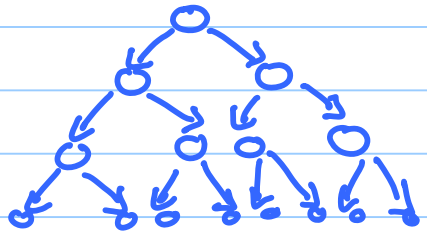
Build-Max Heap Runtime

- Build-Max Heap: Multiple calls to MaxHeapify
- The runtime of Max-Heapify is based on the current height of the heap.
- How much work would be done if we took an unordered array of this size and "heapified" it?
  - At full height we are looking at one node.
  - At height h-1, we are looking at two nodes
  - At height h-2, we are looking at four nodes



Try real example

$n = 15$



$h=0$, # leaves $= 8 = \left\lceil \frac{n}{2^{0+1}} \right\rceil$

$h=1$, # nodes $= 4 = \left\lceil \frac{n}{2^{1+1}} \right\rceil$

$h=2$, # nodes $= 2 = \left\lceil \frac{n}{2^{2+1}} \right\rceil$

$h=3$, # nodes $= 1 = \left\lceil \frac{n}{2^{3+1}} \right\rceil$

RT of Build-Max Heap

$$\sum_{h=0}^{\log n} \left(\begin{array}{c}\#\text{ nodes in a level w/ height } h\end{array}\right) O(h)$$

MaxHeapify

$$\sum_{h=0}^{\log n} \frac{n}{2^{h+1}} \cdot O(h)$$

↓

$$n \sum_{h=0}^{\log n} \frac{h}{2^{h+1}}$$

$$= n \sum_{h=0}^{\log n} \left(\frac{1}{2}\right)\left(\frac{h}{2^h}\right)$$

$$= \frac{1}{2} n \sum_{h=0}^{\log n} \left(\frac{h}{2^h}\right)$$

$$\leq n \sum_{h=0}^{\infty} \frac{h}{2^h}$$

how do we solve?

We can use formula (derived next page)

$$\sum_{j=0}^{\infty} j \cdot x^j = \frac{x}{(1-x)^2}$$

with $x = \frac{1}{2}$

$$\sum_{j=0}^{\infty} \frac{j}{2^j} = \frac{\frac{1}{2}}{(1-\frac{1}{2})^2}$$

$$n \cdot \left(\frac{\frac{1}{2}}{(1-\frac{1}{2})^2}\right)$$

$$= n \cdot \frac{\frac{1}{2}}{\frac{1}{4}}$$

$$= 2n \in O(n)$$

$O(n)$ runtime for Build-Max Heap.
More careful analysis paid off!

Math Detour

Recall for $k<1$ $\quad \sum_{j=0}^{\infty} x^j = \frac{1}{1-x}$

Take derivative of both sides (use chain rule)

$$\sum_{j=0}^{\infty} j \cdot x^{j-1} = \frac{1}{(1-x)^2}$$

$f(x) = \frac{1}{1-x}$
$g(x) = 1-x$
$h(x) = x^{-1}$
$f(x) = h(g(x))$
$f'(x) = h'(g(x)) \cdot g'(x)$

Multiply both sides by x.

$$\sum_{j=0}^{\infty} j \cdot x^j = \frac{x}{(1-x)^2}$$

Use this formula in previous derivation, setting $x = \frac{1}{2}$.

$O(n)$ runtime for Build-Max-Heap

What is Ω ?

n – trivial lower bound (Need to examine each node at least once)

$\Theta(n)$ Amortized Max Heapify

So Runtime of Build Max Heap is $\Theta(n)$

Back to the Question about HeapSort

- Recall that we first Build-Max Heap
- Now we have to harvest the heap
- How much time to harvest the heap in order?

- $O(n)$ to call Build-Max Heap
- $\underbrace{n-1 \text{ calls}}_{O(n-1)}$ to $\underbrace{\text{Max-Heapify}}_{O(\log n)}$ during "harvest" stage.

So the sorting takes $O(n) + O(n \log n)$