Lecture 04

Approaches to building a heap

(A) sort the array using some other sorting algoritm - $O(n \log n)$

(B) start with empty heap, for each of the n items, insert into a heap - $O(n \log n)$

(C) Heapify/Build Heap - Put the items randomly in the array and then "correct". Once the items are in an array (in any order) we can consider them a heap that has the corsdrect shape and then we have to fix the order - $O(n)$

`heapify(A,i)`

Given array A representing a complete binary tree and element x at position i of $A(x = A[i])$ and assuming that the subtrees rooted at the children of x are valid heaps, bubble-down x such that the subtree rooted at x is a valid heap.

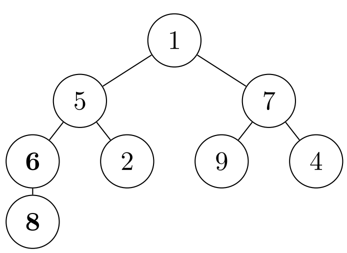
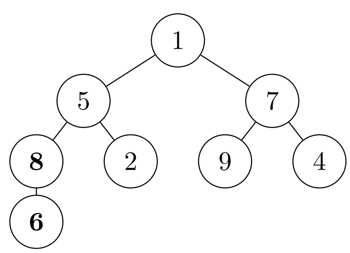
`build-heap(A)`

(1) put elements into array A in any order

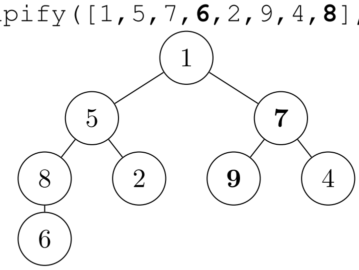
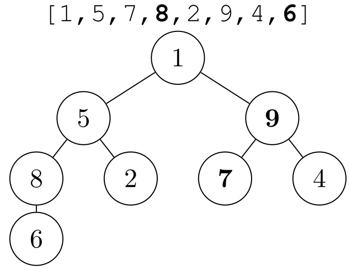
(2) call `build-heap(A)`

```
build-heap(A):
  heapsize ← size(A)
  for i in ⌊heapsize/2⌋...1:
    heapify(A,i)
```

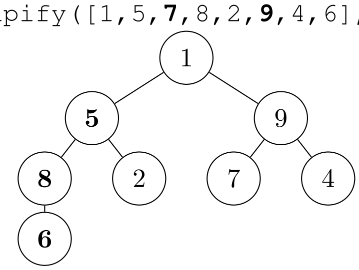
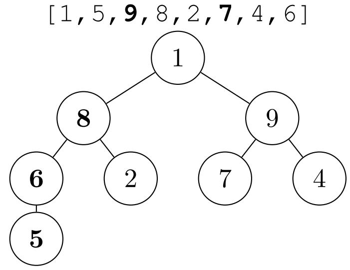
Because each item in the second half of the array is already a heap (it's a leaf), the preconditions for `heapify` are always met before each call. Trace `build-heap` on input $A = [1, 5, 7, 6, 2, 9, 4, 8]$:

 →  →

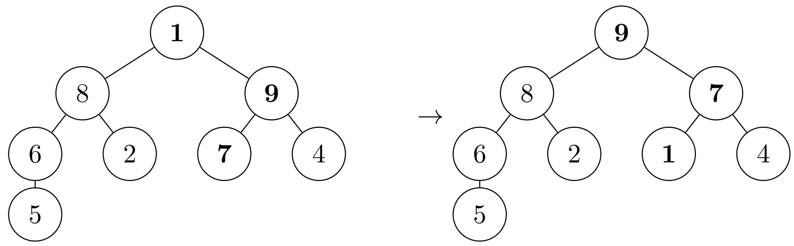
`heapify([1,5,7,`**`6`**`,2,9,4,`**`8`**`], 4)` → `[1,5,7,`**`8`**`,2,9,4,`**`6`**`]` →

 →  →

`heapify([1,5,`**`7`**`,8,2,`**`9`**`,4,6], 3)` → `[1,5,`**`9`**`,8,2,`**`7`**`,4,6]` →

 →  →

`heapify([1,`**`5`**`,9,`**`8`**`,2,7,4,`**`6`**`], 2)` → `[1,`**`8`**`,9,`**`6`**`,2,7,4,`**`5`**`]` →



`heapify([1,8,9,6,2,7,4,5], 1)` $\rightarrow$ `[9,8,7,6,2,1,4,5]`

Complexity of Build Heap

We make $O(n)$ calls to `heapify` and each takes $O(\log n)$ time, so we immidiately get a bound of $O(n \log n)$. But we can do better by analyzing more carefully. The running time of each call to `heapify` is proportional to the height of the tree on which it is called. So we get that the total time taken is

$$O\left(\sum_{h=1}^{\log n} h \cdot a\right) \text{ where } a = \text{ \# of subtrees of height } h$$

How many subtrees of each height?

1 node at height $\log(n)$
2 nodes at height $\log(n) - 1$
$\vdots$
$n/8$ nodes at height 2 (requiring at most 2 swaps)
$n/4$ nodes at height 1 (requiring at most 1 swap)
$n/2$ nodes at height 0 (require 0 swaps)

A complete tree with n nodes contains at most $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ subtrees of height n.

Since a complete tree with n nodes contains at most $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ nodes of height h, we get that the running time of `build-heap` is

$$O\left(\sum_{h=1}^{\log n} h \left\lceil \frac{n}{2^{h+1}} \right\rceil\right) \leq O\left(\frac{n}{2} \sum_{h=1}^{\infty} \frac{h}{2^h}\right) = O(n)$$

Note: $$\sum_{k=0}^{\infty} \frac{k}{2^k} = 2$$

The 2 in the denominator comes from the previous $h+1$ that was simplified to h but this doesn't matter in light of asymptotic analysis.

`heap-sort`

Starting from an array A in some arbitrary order, we must start by building a heap from A, using the process just described. Then, set heapsize == size of A and repeatedly:

- swap the root and the element at position heapsize (which means that the element that ends up at position heapsize is in the correct sorted position)
- decrement heapsize (since the last element is not part of the heap anymore)
- `heapify` starting at the root

This last repeated step should look familiar. It is simply calling `extractMax` repeatedly from the heap until it is empty.

The complexity of `heap-sort`

$$\Theta(n \log n)$$

since we `extractMax` n times and each call is $O(\log n)$.