4.0

Last week we've covered: Quicksort alg. that has a not so good worst ~~Quicksort~~ case behavior but a good average case behavior that works well in practice.

Then, we've seen several divide and conquer algs:

Binary search, Karatsuba's integer multiplication alg. and Strassen's matrix multiplication alg.

Today we are going to see yet another sorting alg. heapsort that is based on a neat data structure called heap. We will introduce heaps and discuss some of their properties.

Then we will prove a lower bound on comparison-based sorting algorithms like the one we've seen so far.

After that we will see a different sorting alg. that beats this lower bound. (because it is not based on comparisons!)

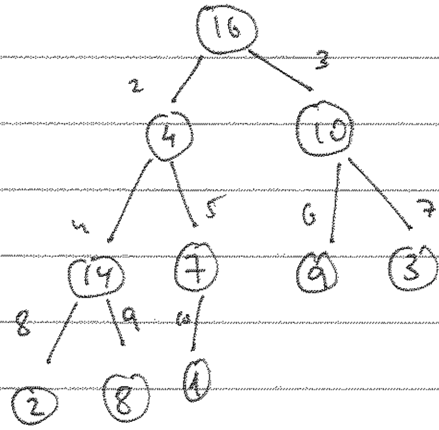
4.1

Heaps

A heap is a data structure – a way to organize your data – that has the form of a nearly complete binary tree.

That is, given an array A, each node of the tree corresponds to an element of the array.

For example, the array A=[16, 4, 10, 14, 7, 9, 3, 2, 8, 1) is represented by the tree



Note that the tree is completely filled in each level except, possibly, the last one, which is filled from the left up to a point.

Sometimes we will have a heap that represents only part of the array so we will have two parameters: A.length and A.heapsize (if we wish to consider only the first A.heapsize elements of A)

The root of the tree is always A[1], and given an index i, its left child is $2i$ and its right child is $2i+1$. i's parent is $\lfloor i/2 \rfloor$. Thus:

Parent(i)	Left(i)	Right(i)
1. Return $\lfloor i/2 \rfloor$	1. Return $2i$	1. Return $2i+1$

4.2

A maxheap (minheap) is a heap in which the values of the node satisfy a heap property. In maxheap the property is that $A[i] \le A[\text{Parent}(i)]$ and in minheap $A[i] \ge A[\text{Parent}(i)]$.

Note that in maxheap the largest element is always at the root. Maxheaps are useful for sorting algorithms and for various other applications.

Another important parameter of a heap/tree is the height of a node. This is the number of edges on the longest downwards path from the node to a leaf. In the example the height of 1 is 3, the height of 3 is 1 and the height of a leaf is 0.

The height of the root in an n-element array is $\Theta(\log n)$. We will see several procedures for obtaining and maintaining a maxheap.

Max heapify: a procedure running in $O(\log n)$ time that is important for creating and maintaining a max heap.

Build Maxheap: a linear time procedure for rearranging A to be a maxheap.

Heapsort: An $O(n \log n)$ time inplace sorting alg.

We will also see several procedures that allow us to insert and remove elements from a heap which are useful when dealing with priority queues.

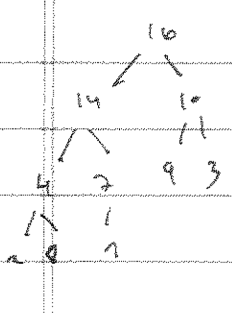
4.3

Let us start with Max heapify. This procedure gets an array A and an index i with the assumption that the trees rooted in i's children are maxheaps. It then let the value of A[i] "float down" to the right location to make the tree rooted at i also a maxheap.

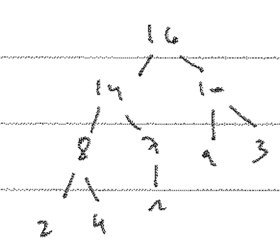
```
Max-heapify (A, i)
l = Left(i)
r = Right(i)
If   l ≤ A.heapsize and A[l] > A[i]
        largest = l
Else largest = i
If   r ≤ A.heapsize and A[r] > A[largest]
        largest = r
If   largest ≠ i
        exchange A[i] with A[largest].
        Max-heapify (A, largest)
```

The alg first find the child with the largest value (if it is not i), an then replaces ~~Akk~~ its value with A[i] and continues to "fix" the relevant subtree.

In our example if $i=2$ then largest $=4$ and we get



then largest $=9$ and we get



4.4

Runningtime: The first 9 lines take $\Theta(1)$ time and then we run the alg. again on a subtree. Thus if we measure the complexity in terms of the height of i we will get

$T(h) \leq T(h-1) + \Theta(1)$

Hence $T(h) = \Theta(h) = O(\log n)$

Correctness: It is easy to prove (say by induction on height) that Max-heapify is correct.

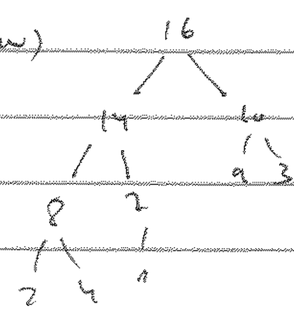
Given Max-heapify we can now give an alg. for building a maxheap from any array.

Notice that in a tree like ours (nearly complete) the elements in locations $A[\lfloor \frac{n}{2} \rfloor + 1, \ldots, n]$ are always leaves of the tree. So ~~they~~ each already forms a maxheap.

Build-Maxheap(A)

1. A.heapsize = A.length
2. For $i = \lfloor A.length/2 \rfloor$ to 1
3. Max-heapify(A, i)

In our example we will start from 5, Max-heapify will do nothing, the same with 4, 3. When we get to 2 we will get (as we saw)



then we go to 1 and stop.

4.5

To show why it works we use the following invariant:
At the start of each For iteration, each node $i+1, i+2, \dots, n$ is a root of a max-heap.

Initialization: $\frac{n}{2}+1, \dots, n$ are all leaves so invariant holds

Maintainance: Follows because Max-heapify works (and by invariant, Left(i), Right(i) are roots of max-heaps).

Termination: We stop when $i=0$ at this point node $i=1$ is a root of a max-heap.

Run time: trivial bound is $O(n \lg n)$ as we have $\frac{n}{2}$ iterations each taking $O(1) \cdot O(\log n)$.

Better analysis: when a node at level h is called, Max-heapify runs in time $O(h)$.

OK, so how many nodes are at height h?

Note that each such node has at least $2^{h-1}-1$ nodes below it. (a full tree at height $h-1$ has $2^{h}-1$ nodes below the root).

Thus, at most $\frac{n}{2^{h}}$ such nodes.

Therefore runtime $\leq \sum_{h=1}^{\log n} O(h) \cdot \frac{n}{2^{h}} = n \cdot O\left(\sum \frac{h}{2^{h}}\right)$

$= O(n)$

4.6

Now that we have an efficient way to construct maxheap we can give an efficient sorting alg.
The idea is that in a maxheap the largest element is in the root.
Thus we can exchange it with A[n], reduce the heapsize by 1, fix the maxheap (Max-heapify) and repeat.

Heapsort(A)
1. Build-Max-heap(A)
2. For i = A.length to 2
3. exchange A[1] with A[i]
4. A.heapsize = A.heapsize - 1
5. Max-heapify(A, 1)

Runtime: n-1 iterations each taking $O(\lg n)$. overall $\Theta(n \lg n)$
Correctness: Can prove that before for loop, elements in i+1 ... n are the n-i largest elements, sorted and that ~~Right(1), Left(1) are maxheaps A is a maxheap~~
A[1 ... A.heapsize] is a maxheap.
In other words A[1 ... i] is a maxheap.

4.7

Priority Queues

A popular application of heaps is in priority queues. For example, when you want to schedule jobs on a shared machine, you want to run them according to their priority so you keep the job ~~names~~ in a maxheap and then run the job at the root and rearrange the heap accordingly.

In general a priority queue is a data structure for maintaining a set S of elements, each with an associated ~~key~~ value called a key. (e.g. $A[i]$ is the value/key of element i). A max-priority queue supports the following operations:

Insert(S, x): insert the element x to S (i.e. $S \leftarrow S \cup \{x\}$)

Maximum(S): return the element with maximum key in S

Extract-Max(S): removes and returns the element in S with largest key

Increase-Key(S, x, k): increase the value/key of x to k

~~For us~~ We will implement a priority queue using heaps. Applications would require that we store at each $A[i]$ also the "name of the element in S" when $S \neq \{1 \ldots n\}$ (each pointer to relevant object etc.), but as this depends on the application we will ignore it.

So let us see how a maxheap can be used to support max-priority queue.

Heap_Maximum(A)

1. Return $A[1]$

4.8

Heap-Extract-Max (A)

1. If A.heap-size < 1
2. return "error". stop.
3. max = A[1]
4. A[1] = A[A.heapsize]
5. A.heapsize = A.heapsize - 1
6. Max-heapify (A, 1)
7. Return max

Heap-Increase-Key (A, i, key)

1. If key < A[i]
2. return "error". stop.
3. A[i] = key
4. while i > 1 and A[Parent(i)] < A[i]

   exchange A[i] with A[Parent(i)]

5. i = Parent(i)

Max-Heap-Insert (A, key)

1. A.heapsize = A.heapsize + 1
2. A[A.heapsize] = $-\infty$
3. Heap-Increase-key (A, A.heapsize, key)

Claim: algs. work and require $O(\lg n)$ time. ■