COMP90038 Algorithms and Complexity

Priority Queues, Heaps and Heapsort

Lars Kulik

Lecture 13

Semester 1, 2016

Heaps and Priority Queues

The heap is a very useful data structure for priority queues, used in many algorithms.

A priority queue is a set (or pool) of elements.

An element is injected into the priority queue together with a priority (often the key value itself) and elements are ejected according to priority.

We think of the heap as a partially ordered binary tree.

Since it can easily be maintained as a complete tree, the standard implementation uses an array to represent the tree.

The Priority Queue

As an abstract data type, the priority queue supports the following operations on a "pool" of elements (ordered by some linear order):

- find an item with maximal priority
- insert a new item with associated priority
- test whether a priority queue is empty
- eject the largest element

Other operations may be relevant, for example:

- replace the maximal item with some new item
- construct a priority queue from a list of items
- join two priority queues

Stacks and Queues as Priority Queues

Special instances are obtained when we use time for priority:

- If "large" means "late" we obtain the stack.
- If "large" means "early" we obtain the queue.

Some Uses of Priority Queues

- Job scheduling done by your operating system. The OS will usually have a notion of "importance" of different jobs.
- (Discrete event) simulation of complex systems (like traffic, or weather). Here priorities are typically event times.
- Numerical computations involving floating point numbers. Here priorities are measures of computational "error".

Many sophisticated algorithms make essential use of priority queues (Huffman encoding and many shortest-path algorithms, for example).

Possible Implementations of the Priority Queue

Assume priority $=$ key.

	$\textsc{Inject}(e)$	$\textsc{Eject}()$
Unsorted array or list		
Sorted array or list		
Heap	$O(\log n)$	$O(\log n)$

How is this accomplished?

The Heap

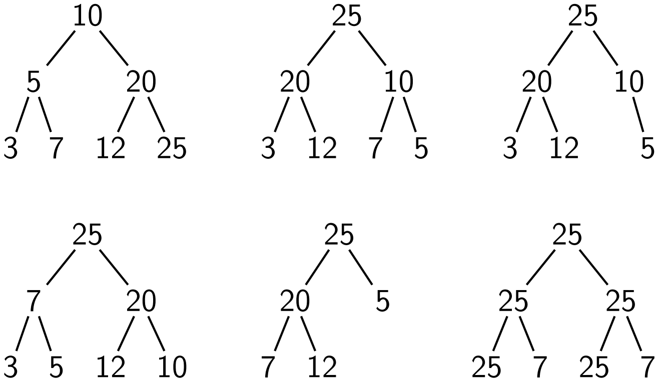
A heap is a complete binary tree which satisfies the heap condition:

Each child has a priority (key) which is no greater than its parent's.

This guarantees the the root of the tree is a maximal element.

(Sometimes we talk about this as a max-heap—one can equally well have min-heaps, in which each child is no smaller than its parent.)

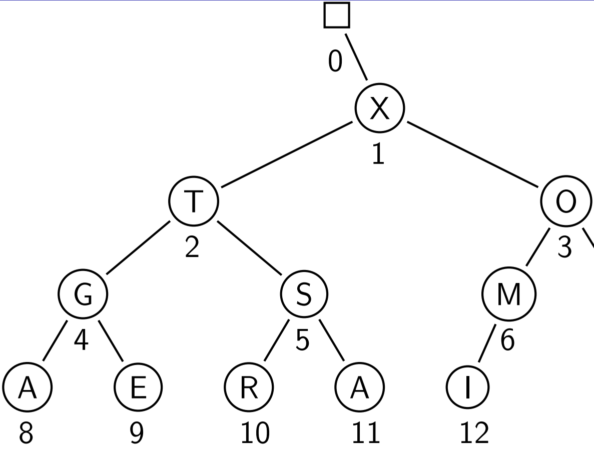
Which of these are heaps?



Heaps as Arrays

We can utilise the completeness of the tree and place its elements in level-order in an array H.

Note that the children of node i will be nodes $2i$ and $2i + 1$.



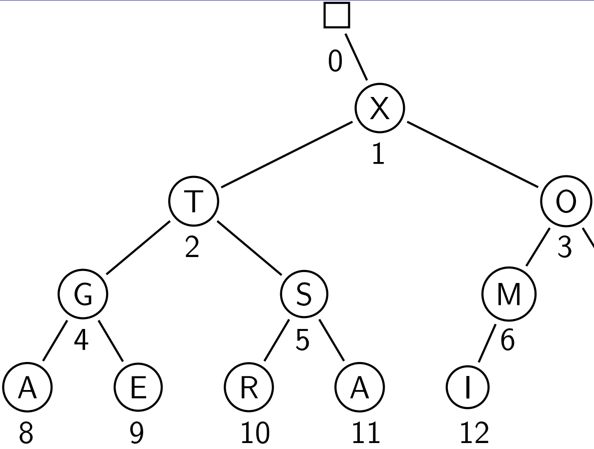
H:

	X	T	O	G	S	M	N	A	E	R	A	I
0	1	2	3	4	5	6	7	8	9	10	11	12

Heaps as Arrays

This way, the heap condition is very simple:

For all $i \in \{0, 1, \ldots, n\}$, we must have $H[i] \leq H[i/2]$.



H:

	X	T	O	G	S	M	N	A	E	R	A	I
0	1	2	3	4	5	6	7	8	9	10	11	12

Properties of the Heap

The root of the tree $H[1]$ holds a maximal item; the cost of $\textsc{Eject}$ is $O(1)$ plus time to restore the heap.

The height of the heap is $\lfloor \log_2 n \rfloor$.

Each subtree is also a heap.

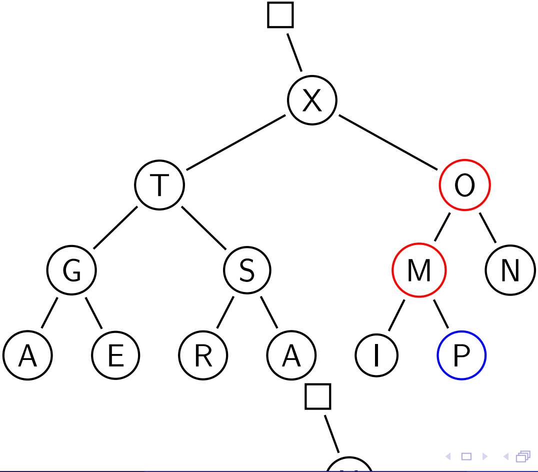
The children of node i are $2i$ and $2i + 1$.

The nodes which happen to be parents are in array positions 1 to $\lfloor n/2 \rfloor$.

It is easier to understand the heap operations if we think of the heap as a tree.

Injecting a New Item

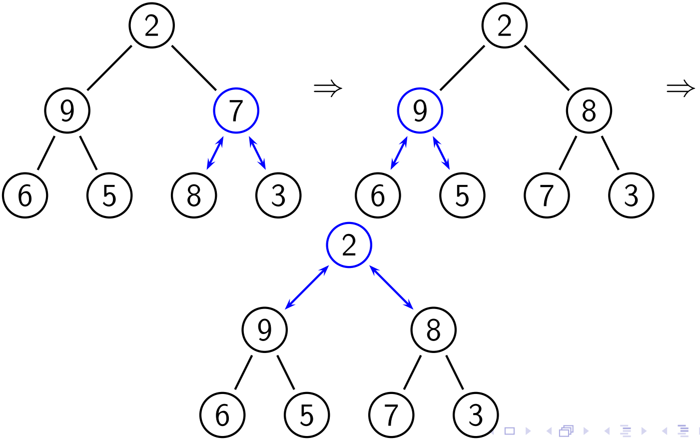
Place the new item at the end; then let it "climb up", repeatedly swapping with parents that are smaller:



$\Rightarrow$

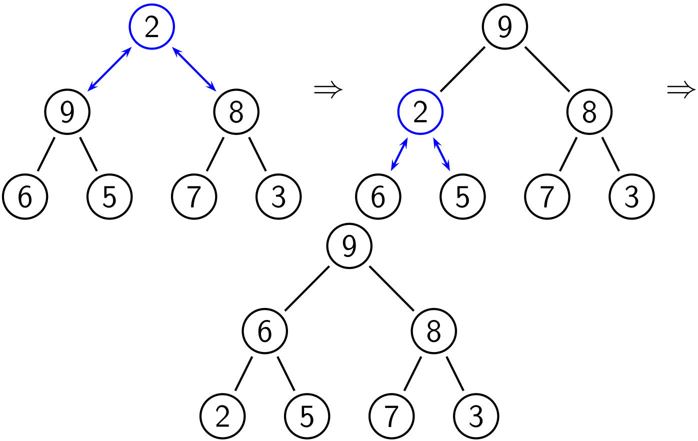
Building a Heap Bottom-Up

To construct a heap from an arbitrary set of elements, we can just use the inject operation repeatedly. The construction cost will be $n \log n$. But there is a better way:



Building a Heap Bottom-Up: Sifting Down

Whenever a parent is found to be out of order, let it "sift down" until both children are smaller:



Algorithm to Turn $H[1..n]$ into a Heap, Bottom-Up

```
for i ← ⌊n/2⌋ downto 1 do
    k ← i
    v ← H[k]
    heap ← False
    while not heap and 2 × k ≤ n do
        j ← 2 × k
        if j < n then
            if H[j] < H[j + 1] then
                j ← j + 1
        if v ≥ H[j] then
            heap ← True
        else
            H[k] ← H[j]
            k ← j
    H[k] ← v
```

Analysis of Bottom-Up Heap Creation

For simplicity, assume the heap is a full binary tree: $n = 2^{h+1} - 1$. Here is an upper bound on the number of "down-sifts" needed (consider the root to be at level h, so leaves are at level 0):

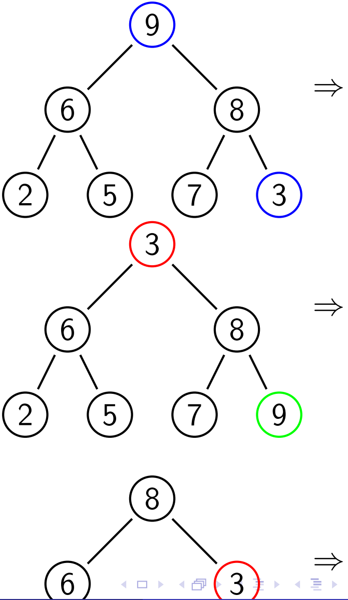
$$\sum_{i=1}^{h} \sum_{\text{nodes at level } h-i} i \quad = \quad \sum_{i=1}^{h} i \cdot 2^{h-i} \quad = \quad 2^{h+1} - h - 2$$

The last equation is easily proved by mathematical induction.

Note that $2^{h+1} - h - 2 < n$, so we perform at most a linear number of down-sift operations. Each down-sift is preceded by two key comparisons, so the number of comparisons is also linear.

Hence we have a linear-time algorithm for heap creation.

Ejecting a Maximal Element from a Heap



Here the idea is to swap the root with the last item z in the heap, and then let z "sift down" to its proper place.

After this, the last element (here shown in green) is no longer considered part of the heap, that is, n is

Exercise: Build and Then Deplete a Heap

First build a heap from the items S, O, R, T, I, N, G.

Then repeatedly eject the largest, placing it at the end of the heap.
✎

Exercise: Build and Then Deplete a Heap

First build a heap from the items S, O, R, T, I, N, G.

Then repeatedly eject the largest, placing it at the end of the heap.
✎

Anything interesting to notice about the tree that used to hold a heap?

Heapsort

Heapsort is a $\Theta(n \log n)$ sorting algorithm, based on the idea from this exercise.

Given unsorted array $H[1..n]$:

Step 1 Turn H into a heap.

Step 2 Apply the eject operation $n - 1$ times.

Properties of Heapsort

On average slower than quicksort, but stronger performance guarantee.

Truly in-place.

Not stable.

Coming Up Next

We will look at the “transform and conquer” paradigm.