Introduction to Algorithms and Data Structures

Lecture 11: Heaps

Mary Cryan

School of Informatics
University of Edinburgh

Remainder of semester 1

Hello everyone! I will take over the teaching for the remainder of semester 1, and also a large part of semester 2.

Plan for (rest of) semester 1:

11. The Heap data structure
12. BuildHeap and HeapSort: running-time
13. QuickSort
14. Graphs I: graph data structures, Breadth-first search
15. Graphs II: DFS, connected components,TopSort

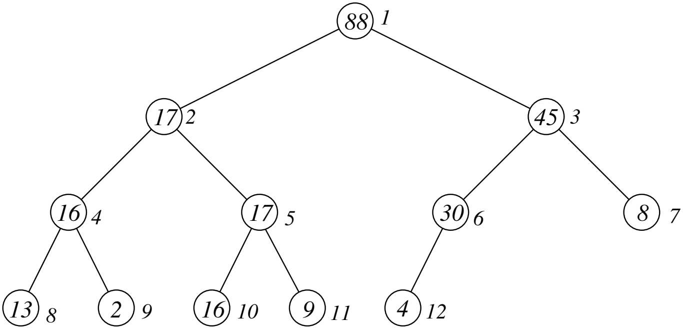
The Heap

Definition

A (max) heap is a "nearly complete" binary tree structure storing items in nodes, where every node is greater than or equal to each of its child nodes.

- The rule for parent/child key values is weaker over the tree as a whole than what we have for red-black trees, 2-3-4 trees or AVL trees (in those cases the tree encodes a total-ordering on the keys in the nodes).
- But ... the topology of a heap is more restricted than for those other tree structures - we have a binary tree with leaves appearing at depth h and depth $h-1$, and all depth-h leaves grouped together to the left.
- The heap does not (readily) carry total-order information, but is ideally set-up to efficiently answer "max" questions (suitable for priority queues).
- Neat structure of the topology means we can store the heap in an array.

Example heap



88	*17*	*45*	*16*	*17*	*30*	*8*	*13*	*2*	*16*	*9*	*4*

Direct mapping: k-th element of heap stored in index $k-1$.
Can use $(2^i - 1) + j - 1$ for index of jth element on level i.
(depends on "Almost-complete" property).

Heaps: height and size

A heap is an almost-complete binary tree:

- All leaves are either at depth $h-1$ or depth h (where h is height).
- The depth-h leaves all appear consecutively from left-to-right.

... A heap of height h has between 2^h and $2^{h+1}-1$ nodes.

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

Hence taking $\lg$ across this inequality, we see

$$h \leq \lg(n) < h+1.$$

This will put h in the range $[\lg(n)-1, \lg(n)]$, ie $\Theta(\lg(n))$.

Lots of our Heap algorithms have worst-case running-time *directly* related to the height of the Heap.

Main operations on a Heap

We imagine that the heap is stored in the array A.

Heap-Maximum Returns the max element of a Heap - $\Theta(1)$ time.

Max-Heapify Runs in $O(\lg(n))$ time and is used to maintain the (max) Heap property whenever some node/index i has violated the heap rule (but left subtree, right subtree are each legal Max Heaps).

Heap-Extract-Max Can return (and delete) the maximum item of a Heap in $O(\lg(n))$ time.

Max-Heap-Insert Can insert a new item (and maintain the heap property) in $O(\lg(n))$ time. Same for Heap-Increase-Key.

Build-Max-Heap Special one called Build-Max-Heap will run in $O(n)$ time to build a Heap from scratch from an unordered input array.

Max-Heapify and the other operations

The Max-Heapify operation (called at i) is used to "fix-up" a Heap where the left-subtree Left(i) is a Heap, and so is the right-subtree Right(i) ... but the value at i violates the Heap property.

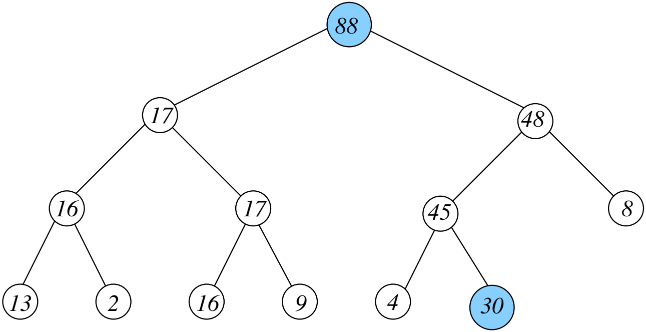
- We will show that Max-Heapify can be implemented in time $O(h)$ for the height h of the heap, which is $O(\lg(n))$.

  (well, specifically, the height of the Heap rooted at i)

- We can then implement Heap-Extract-Max via the trick of just ...
  - Swapping $A[0]$ (the max element) with $A[A.heap\_size - 1]$ (the last item in the array, and decrementing $A.heap\_size$.
  - Then calling Max-Heapify(0) on the Heap to "fix" the error at the root.

- Max-Heapify is also key to the implementation of Build-Max-Heap.

Heap-Extract-Max



The main work is not returning the max element ($\Theta(1)$ time) but removing the max from the tree.

We copy over the "last node" onto the root, then call Max-Heapify to fix things.

Max-Heapify

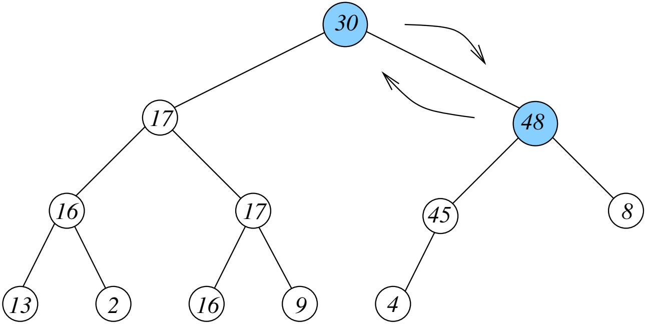
We assume that the “left-heap” Left(i) and the “right-heap” Left(i) are both accurate. Then Max-Heapify(i) will “patch-up” the heap from i.

Algorithm Max-Heapify(A, i)

$\ell \leftarrow$ Left(i)
$r \leftarrow$ Right(i)
$largest \leftarrow i$
if $\ell < A.heap\_size$ **and** $A[\ell] > A[i]$
    $largest \leftarrow \ell$
if $r < A.heap\_size$ **and** $A[r] > A[largest]$
    $largest \leftarrow r$
if $largest \neq i$
    exchange $A[i]$ with $A[largest]$
    Max-Heapify($A, largest$)

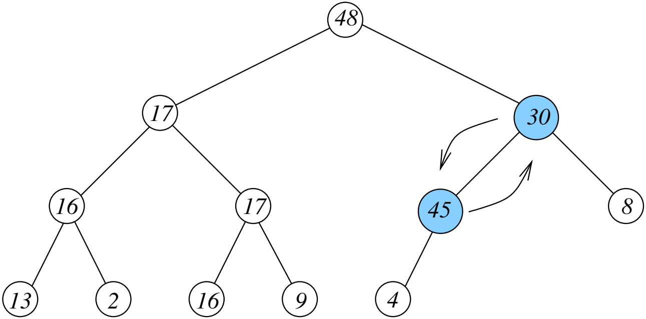
Max-Heapify

We are calling Max-Heapify from the root node.



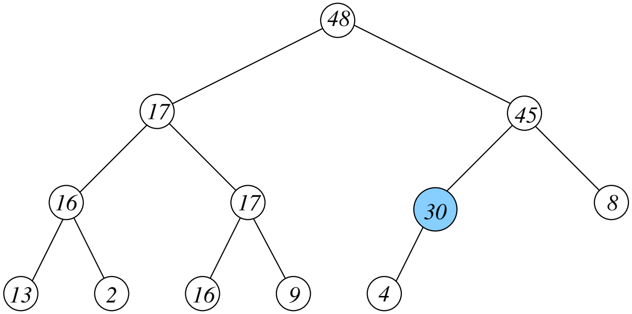
Max child of root is 48 on right, need to swap,
and then recursively call Max-Heapify on 30 as the child (as in line 10. of the Algorithm).

Max-Heapify . . .



Max child of 30 is 45 on left, need to swap,
and then call heapify on 30 as the child.

Max-Heapify ...



Max child of 30 is 4, less than 30. ok. Finish.

Max-Heap-Insert

Algorithm Max-Heap-Insert(A, k)

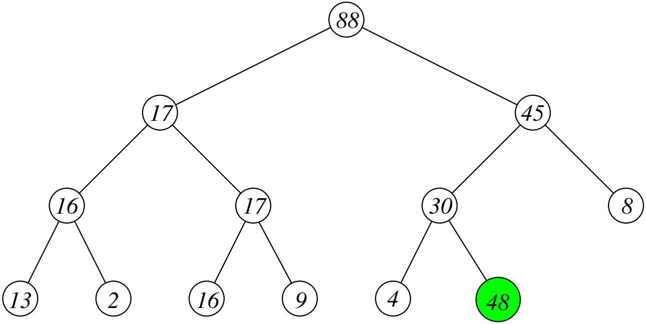
$A.heap\_size \leftarrow A.heap\_size + 1$
$A[heap\_size - 1] \leftarrow k$.
$j \leftarrow heap\_size - 1$
while $(j \neq 0$ **and** $A[j] > A[\text{Parent}(j)])$ **do**
    exchange $A[j]$ and $A[\text{Parent}(j)]$
    $j \leftarrow \text{Parent}(j)$

"Bubble" the item up the tree.
Basically swap k with A[Parent(j)] if k is bigger.

Why is this correct??

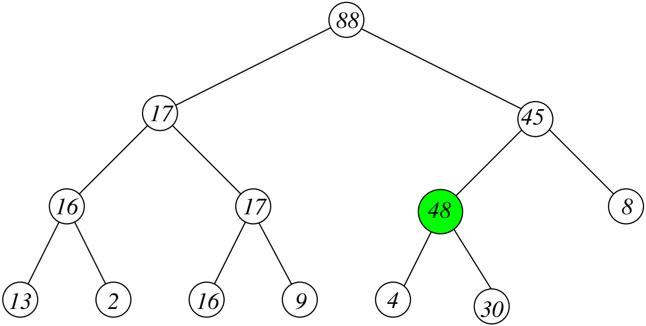
Takes $\Theta(1)$ for adding new last node (initially), and $\Theta(1)$ for every swap. Hence $\Theta(\lg n)$ worst-case in total.

Max-Heap-Insert



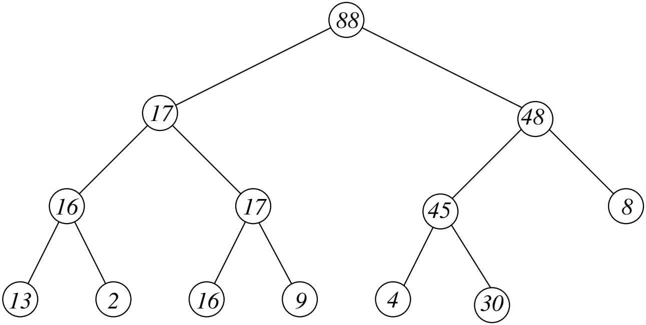
Max-Heap-Insert(48), first add at "last node".
Need to swap 48 with parent 30, because $48 > 30$.

Max-Heap-Insert



48 has now moved-up
Now need to swap 48 with parent 45, because 48 $>$ 45.

Max-Heap-Insert



Done. 48 is less than root 88, no swap needed.

Priority Queues

A Priority queue is a Data Structure for storing collections of elements. They differ in their access policy compared to Lists, Stacks and Queues:

- Every element is associated with a *key*, which is taken from some linearly ordered set, such as the integers.
- Keys represent priorities:

  A larger key means a higher priority.

Classic application is for access to resources like printers, when different users may have varying priority levels.

Priority Queue operations

Methods of *PriorityQueue*:

- insertItem(k, e): Insert element e with key k.
- maxElement$()$: Return an element with maximum key; an error occurs if the priority queue is empty.
- removeMax$()$: Return and remove an element with maximum key; an error occurs if the priority queue is empty.
- isEmpty$()$: Return TRUE if the priority queue is empty and FALSE otherwise.

No findElement(k) or removeItem(k) methods.

Implementations of Priority Queues

Observation:

> *The maximum key in a binary search tree (like a Red-Black tree) is always stored in the rightmost leaf.*

Therefore, all Priority Queue methods can be implemented on an Red-Black tree with running time $\Theta(\lg(n))$ (except isEmpty which is $\Theta(1)$).

However, using a Max Heap we can implement maxElement with Heap-Maximum in $\Theta(1)$ time, while still having insertItem (via Max-Heap-Insert) and removeMax (via Heap-Extract-Max) in $\Theta(\lg(n))$ time.

Note Balanced Search trees can be "tweaked" to maintain a direct pointer to the rightmost leaf, to give $\Theta(1)$ for maxElement.

Reading Material

This lecture used content from sections 6.1, 6.2 and 6.3 of [CLRS]:

- I did Max-Heap-Insert more directly than the book.
- I didn't write the details of Parent, Left, Right on slides (tutorial qn).

In lecture 12, I will cover:

- The method Build-Heap
- The asymptotic analysis of the running-time of the Heap algorithms (6.1-6.3 of [CLRS])
- Heapsort and its running time (6.4 of [CLRS])