Inf 2B: Heaps and Priority Queues

Lecture 6 of ADS thread

Kyriakos Kalorkoti

School of Informatics
University of Edinburgh

Stacks, Queues, and Priority Queues

Stacks, queues, and *priority queues* are all ADTs for storing collections of elements. They differ in their access policy:

Stacks: Last-in-first-out (LIFO)

Queues: First-in-first-out (FIFO)

Priority Queues: Elements have a *priority* associated with them. An element with highest priority gets out first.

The *PriorityQueue* ADT

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

  larger key means *higher* priority.

  Variant: *lower* key means *higher* priority.
  - Not really different, just define a new order $\leq^*$ on keys by

    $$k_1 \leq^* k_2 \iff k_1 \geq k_2,$$

    i.e., reverse existing order.

Different from *Dictionary*—here the meaning of a key is its relative value (in the collection).

The *PriorityQueue* ADT

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 (because k does not mean anything externally).

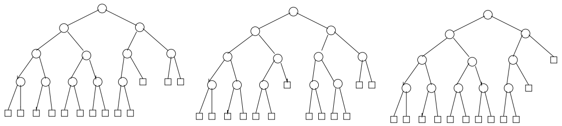
The Search Tree Implementation

Observation: The maximum key in a binary search tree is always stored in the rightmost interior vertex.

Therefore, all *Priority Queue* methods can be implemented on an AVL tree with running time $\Theta(\lg(n))$.

Could we do better?
maxElement() and removeMax() are *simpler* versions of the findElement() and removeItem() for *Dictionary*.

Example Binary Trees



Which of these are "Almost Complete"?

Answer: First one only.

Almost Complete Binary Trees

- All levels except maybe the last one have the maximum number of vertices.
- On the last level, all internal vertices are to the left of all leaves.

Height of an Almost Complete Tree

Theorem: An almost complete binary tree with n internal vertices has height

$$\lfloor \lg(n) \rfloor + 1.$$

(We automatically have $h = O(\lg n)$) WHY?
Proof: A *complete binary tree* of height h has $2^h - 1$ internal vertices (proof by easy induction on h).
For an *almost-complete tree*, of height h number of internal vertices n is:

- strictly more than number of internal vertices of a complete tree of height $h - 1$, so $n \geq (2^{h-1} - 1) + 1 = 2^{h-1}$;
- at most the number of internal vertices of a complete tree of height h, so $n \leq 2^h - 1 < 2^h$.

Thus $2^{h-1} \leq n < 2^h$. Hence

$$h - 1 \leq \lg n < h \Rightarrow h - 1 \leq \lfloor \lg n \rfloor < h$$
$$\Rightarrow h = \lfloor \lg n \rfloor + 1.$$

Abstract Heaps

Definition: A *heap* is an almost complete binary tree whose internal vertices store items such that the following *heap condition* is satisfied:

> (H) For every vertex v other than the root, the key stored at v is smaller than or equal to the key stored at the parent of v.

► So the maximum element is at the root.

The *last vertex* of a heap of height h is the rightmost internal vertex in the hth level.

Finding the Maximum

Algorithm maxElement()

return *root.element*

Runtime is $\Theta(1)$.

Insertion

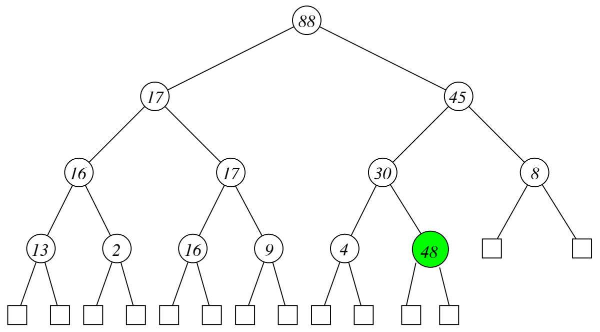
Algorithm insertItem(k, e)

Create new last vertex v.
while v is not the root **and** $k > v.parent.key$ **do**
    store the item stored at $v.parent$ at v
    $v \leftarrow v.parent$
store (k, e) at v

"Bubble" the item up the tree.
Basically swap v with $v.parent$ if v's key is bigger.

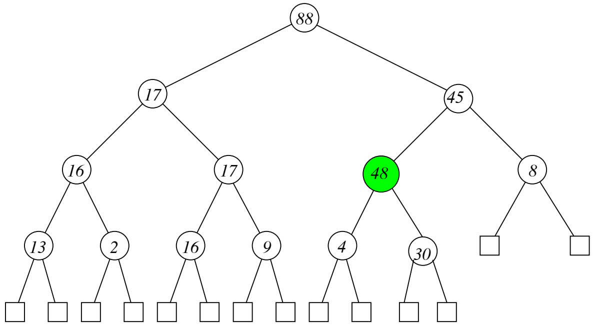
Takes $\Theta(1)$ for adding new last vertex (initially), and $\Theta(1)$ for every swap. Hence $\Theta(\lg n)$ worst-case in total.

insertItem



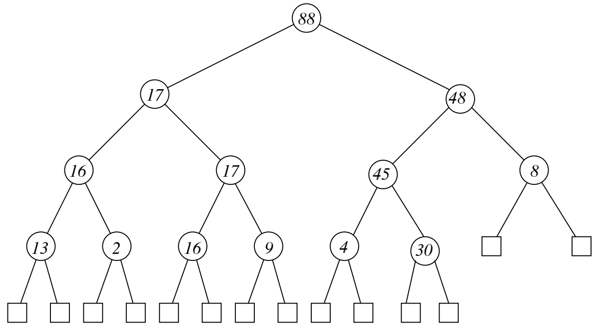
insertItem(48), first add at "last vertex".
Need to swap 48 with parent 30, because $48 > 30$.

insertItem



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

insertItem



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

Removing the Maximum

- Idea: Copy item in “last vertex" into root.
- Delete last vertex (*easy to delete at end of tree*).
- Now *parent greater than child* property might be false. Need to fix.
- New method Heapify(v):
  - Let s be $v.left$ or $v.right$ (whichever has max key).
  - Swap s and v.
  - Call Heapify() recursively.
- $\Theta(h) = \Theta(\lg n)$ time in total. Formal proof in notes.

Removing the Maximum

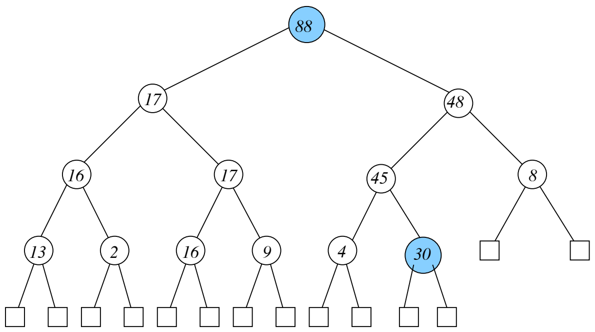
Algorithm removeMax()

```
e ← root.element
root.item ← last.item
delete last
heapify(root)
return e;
```

Algorithm heapify(v)

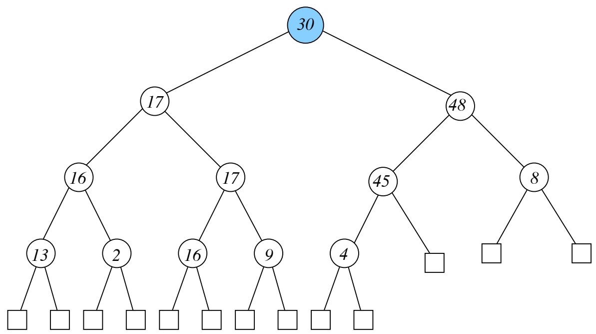
```
if v.left is an internal vertex
and v.left.key > v.key then
        s ← v.left
else
        s ← v
if v.right is an internal vertex
and v.right.key > s.key then
        s ← v.right
if s ≠ v then
        swap the items of v and s
        heapify(s)
```

removeMax



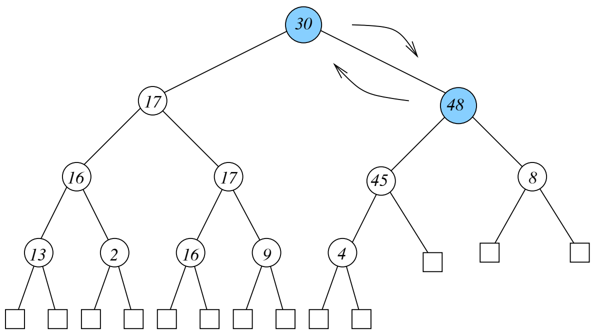
Need to copy over “last vertex” onto root.

removeMax



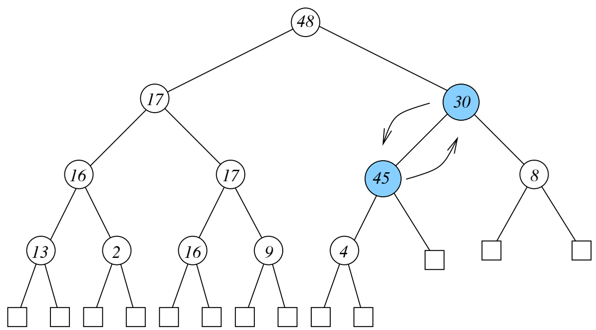
Now we call heapify(*root*).

removeMax



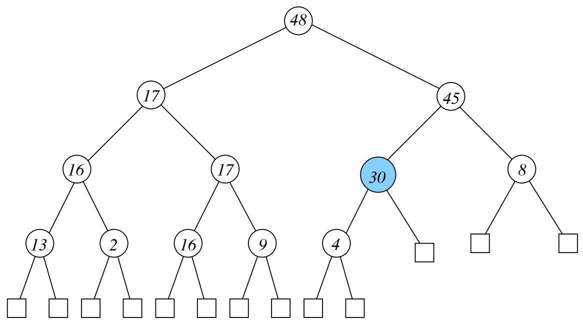
Max child of root is 48 on right, need to swap,
and then call heapify on 30 as the child.

removeMax



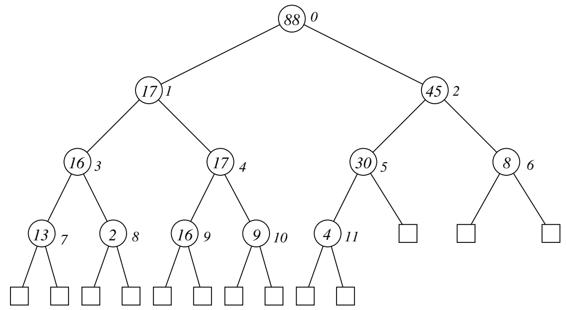
Max child of 30 is 45 on left, need to swap,
and then call heapify on 30 as the child.

removeMax



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

Storing Heaps in Arrays



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

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

Working on Heaps as Arrays

- maxElement(): Just look at index 0 of array.
- insertItem(k, e): Insert into index *size*.
  - $size \leftarrow size + 1$.
  - Do “bubbling" using array structure:
    - v's left child is in index $2v + 1$;
    - right child in index $2v + 2$.
- removeMax():
  - Copy item at $size - 1$ into index 0.
  - $size \leftarrow size - 1$.
  - Do “swapping" using array structure.
- Using dynamic arrays get $\Theta(\lg n)$ amortised time for insertItem(k, e) and removeMax().

Turning an Array into a Heap

Algorithm buildHeap(H)

1. $n \leftarrow H.length$
2. **for** $v \leftarrow \lfloor \frac{n-2}{2} \rfloor$ **downto** 0 **do**
3. heapify(v)

Theorem: The running time of buildHeap is $\Theta(n)$, where n is the length of the array H.

Resources

- The Java Collections Framework has an implementation of *PriorityQueue* (using heaps) in its `java.util` package:
  `http://java.sun.com/j2se/1.5.0/docs/`
  `api/java/util/PriorityQueue.html`
- If you have [GT]: read the “Priority Queues" chapter
- If you have [CLRS]: look at the “Heapsort" chapter (but ignore the sorting for now).