CS 361 Data Structures & Algs Lecture 10

Prof. Tom Hayes
University of New Mexico
09-23-2010

Last Time

- Order Relations
- Sorting
    - Analysis of MergeSort, QuickSort
    - “Unrolling” recurrences to solve them
- Binary search
    - Problem: Ω(n) time to insert/delete
- Priority Queues

Today

Priority Queues & Heaps

Quiz #2

New Reading: secs 3.1 thru 3.4

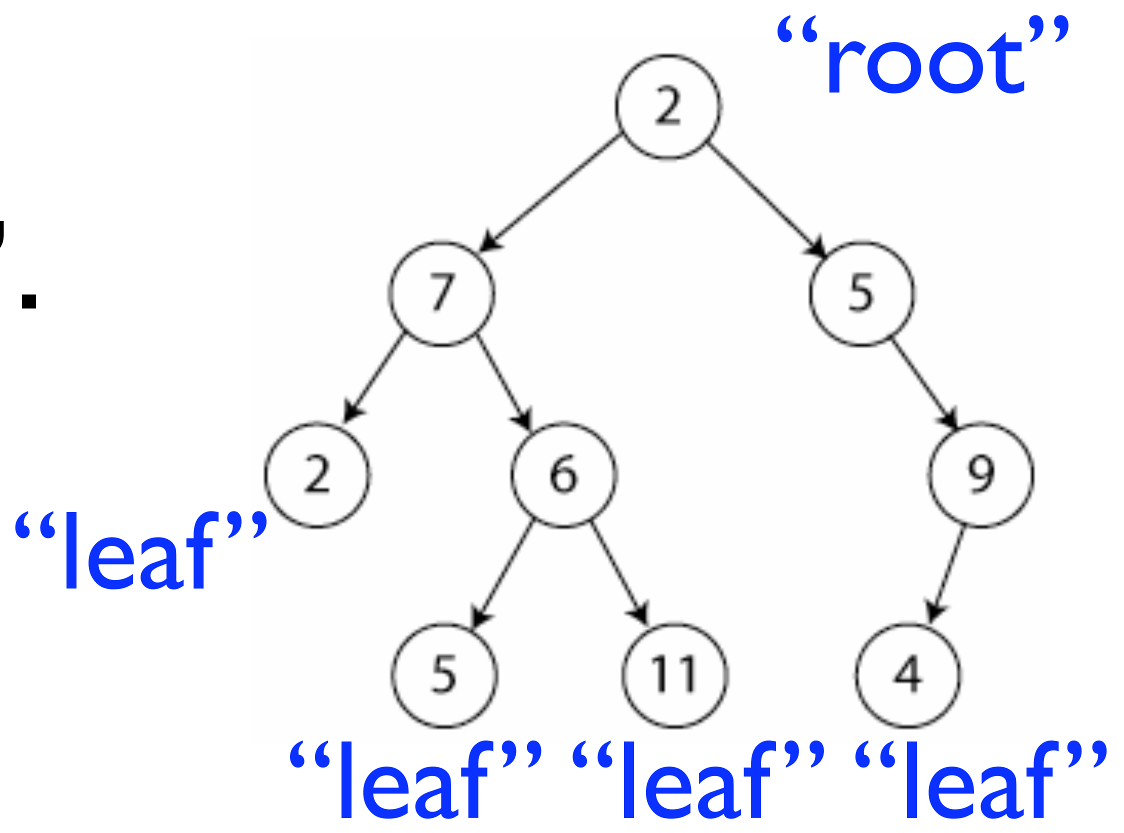
Priority Queues

Stores a collection of data

- Each data has a numeric “key value”
- operations supported: add, delete, extract_min.
- guarantees: O(log n) time per operation
- n = current size of collection.

WHiPFO, not FIFO.

Binary Trees



Data is stored in "nodes".

Each node has 4 fields:

data

parent (either a ref to a node, or "null")

left_child (reference to a node or null)

right_child (reference to a node or null)

Heap Order Property

We say a tree storing “key” values satisfies the (min-) heap order property if it is always the case that the parent of a node stores a value $\leq$ than the node does.

Such a tree is called a “heap.”

A heap is called “balanced” if every layer except perhaps the bottom one, has the maximum possible number of nodes.

Q: What is this number? 2^L for the L’th level away from the root. (“depth L”)

Heap Order Property

What can we do with a heap?

Can we search it quickly?

No. (whiteboard)

Can we extract_min quickly?

Well, we can find_min quickly. But if we extract it, we will have to replace the root.

What about adding an element? Stick it in a leaf. But it might violate the order property!

Reading: Heapify-Up, Heapify-Down

Goal: Fix a near-heap that has just one value out of place.

If it's smaller than its parent, swap it with its parent. Recurse! (Heapify-up)

If it's bigger than a child, swap it with the smaller child. Recurse! (Heapify-down)

Applet at http://people.ksp.sk/~kuko/bak/index.html

Adding to Heap

Place new item in leftmost unfilled spot in bottom level of tree. (Start new row if full.)

Run Heapify-Up to restore heap order prop.

Note 1: Heapify-up only does swaps, so it preserves the shape of the tree. Thus, tree remains balanced.

Note 2: “Wrong” key only moves up, so terminates in $\leq$ log n iterations.

Extract-min

Grab item from root. Now root is “empty.” Replace it with the rightmost item in bottom row of tree (remove this leaf).

Now the tree is the desired shape, but may violate H.O.P. at the root node (key too big).

Run Heapify-Down to restore H.O.P.

Note 1: Heapify-down only does swaps, so it preserves the shape of the tree. Thus, tree remains balanced.

Note 2: “Wrong” key only moves down, so terminates in $\leq \log n$ iterations.

Heapify-down

```
Heapify-down(Node v) { // key(v) may be too big.
    if (key(v) ≤ keys(all existing children of v))
        return    // nothing more to do.
    else {
        swap v with (child with smaller key)
        Heapify-down(that child)
    }
```

Delete

To delete an arbitrary data item, we need some way to locate it quickly in the heap.

Delete

To delete an arbitrary data item, we need some way to locate it quickly in the heap.

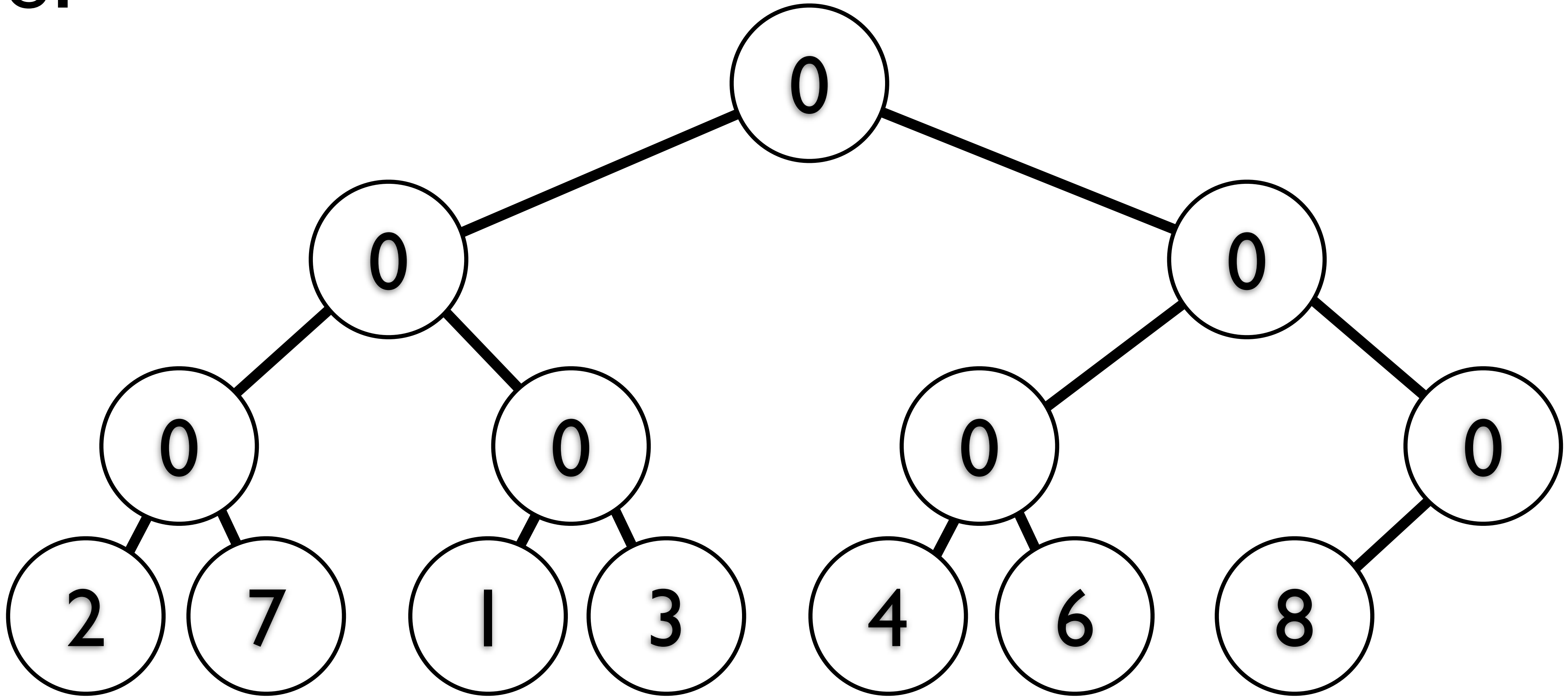
1) The heap itself is not going to let us do this.

Delete

To delete an arbitrary data item, we need some way to locate it quickly in the heap.

1) The heap itself is not going to help us do this.

Example:



Delete

To delete an arbitrary data item, we need some way to locate it quickly in the heap.

1) The heap itself is not going to help us do this.

Thus, we need to store some additional data. Let's keep an index array, that, for each element, keeps a reference to the node storing it in the heap.

* Extra bookkeeping. When doing swap operations, must update this array.

Delete

To delete an arbitrary data item, we need some way to locate it quickly in the heap.

1) The heap itself is not going to help us do this.

Thus, we need to store some additional data. Let's keep an index array, that, for each element, keeps a reference to the node storing it in the heap.

* Extra bookkeeping. When doing swap operations, must update this array.

Assumes: items come from a small set. Why?

Delete

```
Delete(item i) {
    Node v = Position[i]
    Remove rightmost leaf from bottom row, placing
    its item in node v;
    Update Position array;
    Heapify-up(v);
    Heapify-down(v);   // why both?
}
```

Running Times

These 3 operations: Add, Extract-min, Delete, always run in time O(log n) for heap of size n.

Why? Would be time O(1) except for the calls to Heapify-Up and Heapify-Down. For these, the function body runs in time O(1), but this needs to be multiplied by the number of recursive calls that might occur.

Since the “bad node” always moves in the same up/down direction, and the tree is balanced, this is ≤ log n times.

Additional Operations

Find-min: (peek) Returns value of minimum key, but doesn’t remove it. Runs in ?

Additional Operations

Find-min: (peek) Returns value of minimum key, but doesn't remove it. Runs in O(1) time.

Additional Operations

Find-min: (peek) Returns value of minimum key, but doesn't remove it. Runs in O(1) time.

Change-key(item i): give item i a new key value. Runs in ?

Additional Operations

Find-min: (peek) Returns value of minimum key, but doesn’t remove it. Runs in O(1) time.

Change-key(item i): give item i a new key value. Runs in O(log n) time. We can just Delete item i, then Add it again.

Additional Operations

Find-min: (peek) Returns value of minimum key, but doesn't remove it. Runs in O(1) time.

Change-key(item i): give item i a new key value. Runs in O(log n) time. We can just Delete item i, then Add it again.

Faster: change its key, then run Heapify-up and Heapify-down on it again. Need to use Position[ ] to find it quickly in the first place.

Hack: Tree in an Array

We've been thinking of a tree in an Object-Oriented way. Each Node stores its:

parent, left-child, right-child, data.

Here's a slick hack that works when:

(a) we know the tree is going to be balanced,

and (b) we know how big it will be.

Hack: Tree in an Array

For a tree of depth ≤ d, store in an array of length 2^(d+1).

Root is in array position 0.

Left-child of i is in position (2*i + 1).

Right-child of i is in position (2*i + 2).

Parent of i is in position (i-1)/2 (integer division), except for root.

Null data corresponds to “node not present”

(see whiteboard)

Reading Assignment:

Read Sections 3.1 to 3.4

Topics: Graphs, Trees (unrooted), Connectedness, Criteria for Being a Tree,

Graph Traversal (DFS, BFS),

Implementation of Graph Traversal using Queues and Stacks

Testing Bipartiteness