CSE332: Data Abstractions

Lecture 4: Priority Queues

Ruth Anderson
Winter 2011

Announcements

- **Project 1** – phase A due Wed Jan 12th, 11pm via catalyst
- **Homework 1** – due Friday Jan 14th at ***beginning*** of class

- Re-organization on course web page
- Info sheets?

Today

- Finish up Asymptotic Analysis
- New ADT! Priority Queues

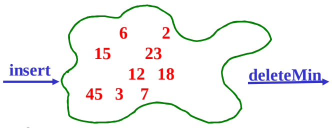
A new ADT: Priority Queue

- Textbook Chapter 6
  – We will go back to binary search trees (ch4) and hash tables (ch5) later
  – Nice to see a new and surprising data structure first
- A **priority queue** holds *compare-able data*
  – Unlike stacks and queues need to *compare items*
    • Given *x* and *y*, is *x* less than, equal to, or greater than *y*
    • What this means can depend on your data
  – Much of course will require comparable data: e.g. sorting
  – Integers are comparable, so will use them in examples
    • But the priority queue ADT is much more general

Priority Queue ADT

- Assume each item has a "priority"
  – The *lesser* item is the one with the *greater* priority
  – So "priority 1" is more important than "priority 4"
  – (Just a convention)

- Operations:
  – **`insert`**
  – **`deleteMin`**
  – **`create`**, **`is_empty`**, **`destroy`**



- Key property: **`deleteMin`** returns and deletes from the queue the item with greatest priority (lowest priority value)
  – Can resolve ties arbitrarily

Focusing on the numbers

- For simplicity in lecture, we'll often suppose items are just **`ints`** and the **`int`** is the priority
  – The same concepts without generic usefulness
  – So an operation sequence could be

```
insert 6
insert 5
x = deleteMin
```

  – **`int`** priorities are common, but really just need comparable
  – Not having "other data" is very rare
    • Example: print job is a priority *and* the file

Example

insert *x1* with priority *5*
insert *x2* with priority *3*
insert *x3* with priority *4*
a = **deleteMin**
b = **deleteMin**
insert *x4* with priority *2*
insert *x5* with priority *6*
c = **deleteMin**
d = **deleteMin**

- Analogy: **insert** is like **enqueue**, **deleteMin** is like **dequeue**
  - But the whole point is to use priorities instead of FIFO

Applications

Like all good ADTs, the priority queue arises often
- Sometimes "directly", sometimes less obvious

- Run multiple programs in the operating system
  - "critical" before "interactive" before "compute-intensive"
  - Maybe let users set priority level
- Treat hospital patients in order of severity (or triage)
- Select print jobs in order of decreasing length?
- Forward network packets in order of urgency
- Select most frequent symbols for data compression (cf. CSE143)
- Sort: **insert** all, then repeatedly **deleteMin**
  - Much like Project 1 uses a stack to implement reverse

More applications

- "Greedy" algorithms
  - Select the 'best-looking' choice at the moment
  - Will see an example when we study graphs in a few weeks
- Discrete event simulation (system modeling, virtual worlds, …)
  - Simulate how state changes when events fire
  - Each event *e* happens at some time t and generates new events *e1*, …, *en* at times *t+t1*, …, *t+tn*
  - Naïve approach: advance "clock" by 1 unit at a time and process any events that happen then
  - Better:
    - *Pending events* in a priority queue (priority = time happens)
    - Repeatedly: **deleteMin** and then **insert** new events
    - Effectively, "set clock ahead to next event"

Implementations of Priority Queue ADT

	insert	deleteMin
Unsorted Array		
Unsorted Linked-List		
Sorted Circular Array		
Sorted Linked-List		
Binary Search Tree (BST)		

Need a good data structure!

- Will show an efficient, non-obvious data structure for this ADT
  - But first let's analyze some "obvious" ideas for *n* data items
  - All times worst-case; assume arrays "have room"

data	*insert algorithm / time*		*deleteMin algorithm / time*	
unsorted array	add at end	$O(1)$	search	$O(n)$
unsorted linked list	add at front	$O(1)$	search	$O(n)$
sorted circular array	search / shift	$O(n)$	move front	$O(1)$
sorted linked list	put in right place	$O(n)$	remove at front	$O(1)$
binary search tree	put in right place	$O(n)$	leftmost	$O(n)$

More on possibilities

- If priorities are random, binary search tree will likely do better
  - $O($**log** $n)$ **insert** and $O($**log** $n)$ **deleteMin** on average

- But we are about to see a data structure called a "binary heap"
  - $O($**log** $n)$ **insert** and $O($**log** $n)$ **deleteMin** worst-case
  - Very good constant factors
  - If items arrive in random order, then **insert** is $O(1)$ on average

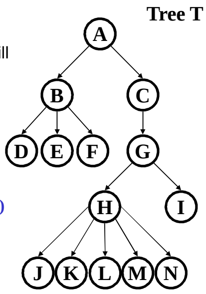
- One more idea: if priorities are 0, 1, …, *k* can use array of lists
  - **insert**: add to front of list at **arr[priority]**, $O(1)$
  - **deleteMin**: remove from lowest non-empty list $O(k)$

Tree terms (review?)

The binary heap data structure implementing the priority queue ADT will be a *tree*, so worth establishing some terminology

root(tree)
children(node)
parent(node)
leaves(tree)
siblings(node)
ancestors(node)
descendents(node)
subtree(node)

depth(node)
height(tree)
degree(node)
branching factor(tree)

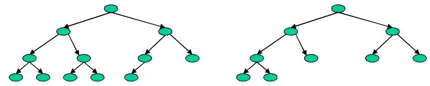


1/10/2011 13

Kinds of trees

Certain terms define trees with specific structure

- Binary tree: Each node has at most 2 children
- *n*-ary tree: Each node as at most *n* children
- Complete tree: Each row is completely full except maybe the bottom row, which is filled from left to right



Teaser: Later we'll learn a tree is a kind of directed graph with specific structure

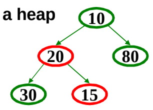
1/10/2011 14

Our data structure

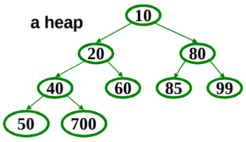
Finally, then, a *binary min-heap* (or just *binary heap* or just *heap*) is:

- A complete tree – the "structure property"
- For every (non-root) node the parent node's value is less than the node's value – the "heap order property" (not a binary search tree)

not a heap



a heap



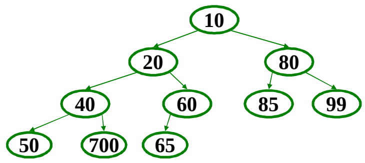
So:

- Where is the highest-priority item?
- What is the height of a heap with *n* items?

1/10/2011 15

Heap Operations

- findMin:
- deleteMin: percolate down.
- insert(val): percolate up.



1/10/2011 16

Heap – Deletemin

Basic Idea:

1. Remove root (that is always the min!)
2. Put "last" leaf node at root
3. Find smallest child of node
4. Swap node with its smallest child if needed.
5. Repeat steps 3 & 4 until no swaps needed.

1/10/2011 17

Heap – Insert(val)

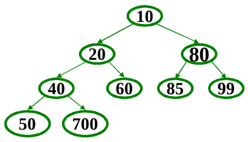
Basic Idea:

1. Put val at "next" leaf position
2. Repeatedly exchange node with its parent if needed

1/10/2011 18

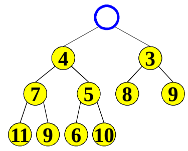
Operations: basic idea

- **`findMin`**: return **`root.data`**
- **`deleteMin`**:
  1. **`answer = root.data`**
  2. Move right-most node in last row to root to restore structure property
  3. "Percolate down" to restore heap property
- **`insert:`**
  1. Put new node in next position on bottom row to restore structure property
  2. "Percolate up" to restore heap property



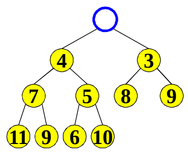
DeleteMin

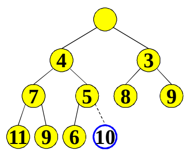
1. Delete (and return) value at root node



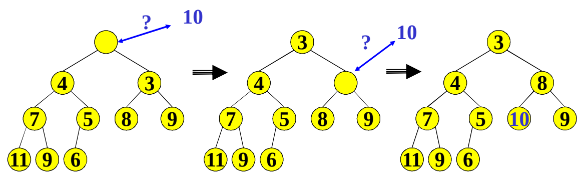
2. Restore the Structure Property

- We now have a "hole" at the root
  - Need to fill the hole with another value
- When we are done, the tree will have one less node and must still be complete





3. Restore the Heap Property



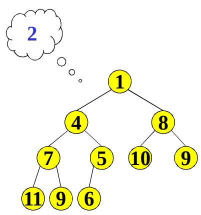
Percolate down:
- Keep comparing with both children
- Move smaller child up and go down one level
- Done if both children are ≥ item or reached a leaf node
- What is the run time?

DeleteMin: Run Time Analysis

- Run time is *O*(height of heap)
- A heap is a complete binary tree
- Height of a complete binary tree of *n* nodes?
  - height = $\lfloor \log_2(n) \rfloor$
- Run time of **`deleteMin`** is *O*(`log` *n*)

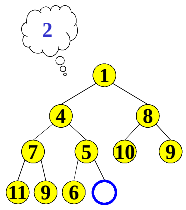
Insert

- Add a value to the tree
- Structure and heap order properties must still be correct afterwards

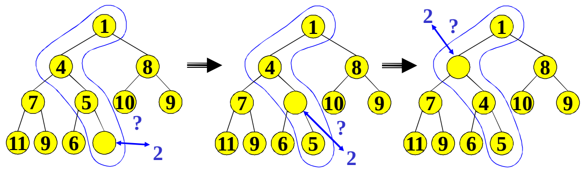


Insert: Maintain the Structure Property

- There is only one valid tree shape after we add one more node
- So put our new data there and then focus on restoring the heap property



Maintain the heap property



Percolate up:
- Put new data in new location
- If parent larger, swap with parent, and continue
- Done if parent ≤ item or reached root
- Run time?

Insert: Run Time Analysis

- Like **`deleteMin`**, worst-case time proportional to tree height
  - *O*(**`log`** *n*)
- But… **`deleteMin`** needs the "last used" complete-tree position and **`insert`** needs the "next to use" complete-tree position
  - If "keep a reference to there" then **`insert`** and **`deleteMin`** have to adjust that reference: *O*(**`log`** *n*) in worst case
  - Could calculate how to find it in *O*(**`log`** *n*) from the root given the size of the heap
    - But it's not easy
    - And then **`insert`** is always *O*(**`log`** *n*), promised *O*(1) on average (assuming random arrival of items)
- There's a "trick": don't represent complete trees with explicit edges! (see in next lecture)