Even More ADTs

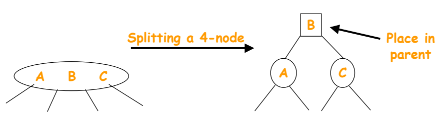
Lecture 18
CS211 – Fall 2006

Announcements & Reminders

- Prelim 2
  - Tuesday, Nov 14
  - 7:30-9:00PM
  - If you have a conflict
    - Contact Kelly Patwell (Course Administrator) soon!
- Dictionary operations
  - void insert (key, value)
  - void update (key, value)
  - value find (key)
  - void remove (key)
- Dictionary implementations
  - Linked lists & arrays
    - Too slow
  - Direct Address Tables
    - Limited usage
  - Hash Tables (using chaining and table doubling)
    - O(1) expected time
  - BSTs
    - O(log n) expected time for
      - Input in random order
      - No deletions
  - Balanced BSTs
    - O(log n) worst-case time

Example Balancing Scheme: 234-Trees

- Nodes have 2, 3, or 4 children (and contain 1, 2, or 3 keys, respectively)
- All leaves are at the same level
- Basic rule for insertion: We hate 4-nodes
  - Split a 4-node whenever you find one while coming down the tree
  - Note: this requires that parent is not a 4-node
- Delete is harder than insert
  - For delete, we hate 2-nodes
  - As in BSTs, cannot delete from a nonleaf so we use same BST trick: delete successor and recopy its data



234-Tree Analysis

- Time for insert or get is proportional to tree's height
- How big is tree's height h?
- Let n be the number of nodes in a tree of height h
  - n is large if all nodes are 4-nodes
  - n is small if all nodes are 2-nodes
- Can use this to show
  h = O(log n)

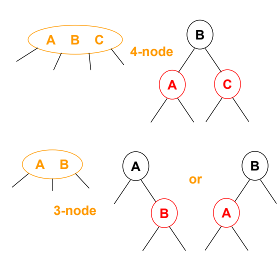
Analysis of tree height:

- Let N be the number of nodes, n be the number of items, and h be the height
- Define h so that a tree consisting of a single node is height 0
- It's easy to see $1+2+4+\ldots+2^h \leq N \leq 1+4+16+\ldots+4^h$
- It's also easy to see $N \leq n \leq 3N$
- Using the above, we have $n \geq 1+2+4+\ldots+2^h = 2^{h+1}-1$
- Rewriting, we have $h \leq \log(n+1) - 1$ or $h = O(\log n)$
- Thus, Dictionary operations on 234-trees take time O(log n) in the worst case

234-Tree Implementation

- Can implement all nodes as 4-nodes
  - Wasted space
- Can allow various node sizes
  - Requires recopying of data whenever a node changes size
- Can use BST nodes to emulate 2-, 3-, or 4-nodes

Using BSTs to Emulate 234-Trees



- A 2-node can be represented with a standard BST node
- A 4-node can be represented with three BST nodes
- A 3-node can be represented with two BST nodes (in two different ways)

Red-Black Trees

- We need a way to tell when an emulated 234-node starts and ends
- We mark the nodes
  - Black: "root" of 234-node
  - Red: belongs to parent
  - Requires one bit per node
- 234-tree rules become rules for *rotations* and color changes in red-black trees
- Result:
  - One black node per 234-node
  - Number of black nodes on path from root to leaf is same as height of 234-tree
  - On any path: at most one red node per black node
  - Thus tree height for red-black tree is O(log n)

Balanced Tree Schemes

- AVL trees [1962]
  - named for initials of Russian creators
  - uses rotations to ensure heights of child trees differ by at most 1
- 23-Trees [Hopcroft 1970]
  - similar to 234-tree, but repairs have to move back up the tree
- B-Trees [Bayer & McCreight 1972]
- Red-Black Trees [Bayer 1972]
  - not the original name
- Red-black convention & relation to 234-trees [Guibas & Stolfi 1978]
- Splay Trees [Sleator & Tarjan 1983]
- Skip Lists [Pugh 1990]
  - developed at Cornell

Selecting a Dictionary Scheme

- Use an unordered array for small sets (< 20 or so)
- Use a Hash Table if possible
  - Cannot efficiently do some ops that are easy with BSTs
  - Running times are expected rather than worst-case
- Use an ordered array if few changes after initialization
- B-Trees are best for large data sets, external storage
  - Widely used within data base software
- Otherwise, Red-Black Trees are current scheme of choice
- Skip Lists are supposed to be easier to implement
  - But shouldn't have to implement—use existing code
- Splay trees are useful if some items are accessed more often than others
  - But if you know which items are most-commonly accessed, use a separate data structure

Possible Priority Queue Implementations

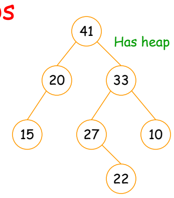
	Unordered List	Ordered List	Unordered Array	Ordered Array	BST*	Balanced BST
insert(item)	O(1)	O(n)	O(1)	O(n)	O(log n) expected	O(log n) worst-case
removeMax( )	O(n)	O(1)	O(n)	O(1)	O(log n) expected	O(log n) worst-case

* BST becomes unbalanced as PQ is used

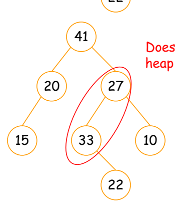
Can we do better than balanced trees?
Well no, not in terms of big-O bounds, but…

Heaps

- A heap is a tree that
  - Has a particular shape (we'll come back to this) and
  - Has the *heap property*
- *Heap property*
  - Each node's value (its *priority*) is ≤ the value of its parent
  - This version is for a max-heap (max value at the root)
    - There is a similar heap property for a min-heap (min at the root)



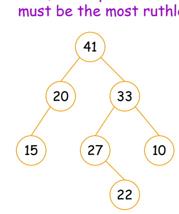
Has heap property



Does not have heap property

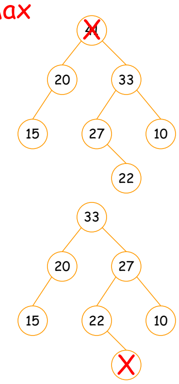
Heap Property Examples

- Ages of people in a family tree
  - Child is younger than parent
  - But an aunt can be younger than her niece
- Salaries of people in an organization
  - A boss makes more money than a subordinate
  - But a 2nd level manager in one region may make more than a 1st level manager in another region
- Crime family ordered by "ruthlessness" (measured by number of murders each member is responsible for)
  - Max, the top crime boss, must be the most ruthless



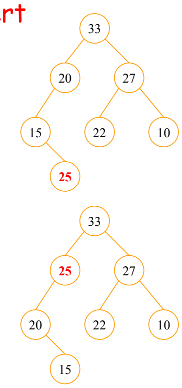
GetMax

- What would happen if someone were to "get" Max (the top boss)?
  - This leaves a hole at the root
  - We must maintain the heap property so…
    - The most ruthless subordinate moves up to fill the hole
  - This leaves another hole that we fill in the same way
  - We finally create an empty leaf which we delete



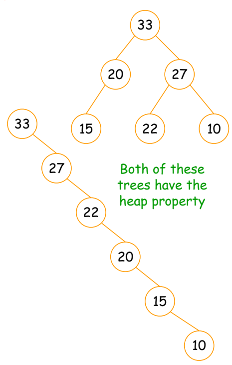
Insert

- What happens when "Fat Tony" arrives from Detroit?
  - He starts as a leaf
  - We must maintain the heap property, so…
    - If he is more ruthless than his boss, they swap positions



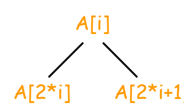
Heap Implementation

- This works great, but…
  - Operations insert and getMax can be slow if the tree is "skinny"
    - Both take linear time on a skinny tree and O(log n) time on a fat tree
- How can we ensure that our heap-tree is fat?



Heap Implementation (the Big Trick)

- Can avoid using pointers!



- Use a *complete* binary tree stored in an array
  - Definition: *Complete* means that each level of the tree is filled except possibly the last, which is filled from left to right
- For A[i]
  - left child = 2 * i
  - right child = 2 * i + 1
  - parent = $\lfloor i / 2 \rfloor$

Insert and GetMax Pseudocode

```
insert (item):
  Place item in a leaf (= next empty position in array);
  while (item > parent) {Swap item with parent;} // BubbleUp

getMax ():
  max = root.value;
  Swap root with last item (call it v) in heap;     // Unchanging heap-shape
  Decrease heap size by 1 (i.e., access less of the array);
  while (v < one of its children)                   // BubbleDown
          {Swap v with its largest child;}
  return max;
```

To Build a Heap

- How long to construct a heap, given the items?
- Worst-case time for insert() is O(log n)
- Total time to build heap using insert() is
  O(log 1) + O(log 2) + … + O(log n)
  or O(n log n)

Can we do better?

- We had two heap-fixing methods
  - bubbleUp: move up the tree as long as we're > our parent
  - bubbleDown: move down the tree as long as we're < one of our children
- If we build the heap from the bottom-up using bubbleDown then we can build it in time O(n) (Wow!)

Efficient Heap Building

- Build from the bottom-up
- If there are n items in the heap then...
  - There are about n/2 mini-heaps of height 1
  - There are about n/4 mini-heaps of height 2
  - There are about n/8 mini-heaps of height 3 and so on
- The time to fix up a mini-heap is O(its height)
- Total time spent fixing heaps is thus bounded by
  $n/2 + 2n/4 + 3n/8 + \ldots$
- This can be rewritten as
  $n(1/2 + 2/4 + \ldots + i/2^i + \ldots)$
  $= n(2)$
- Thus total heap-building time (using the bottom-up method) is O(n)

HeapSort

- Given a Comparable[ ] array of length n,
  - Put all n elements into a heap: O(n) or O(n log n)
  - Repeatedly get the min: O(n log n)

```
public static void heapSort(Comparable[] a) {
  PriorityQueue<Comparable> pq = new PriorityQueue<Comparable>();
  for (Comparable x : a) { pq.put(x); }
  for (int i = 0; i < a.length; i++) { a[i] = pq.get(); }
}
```

PQ Application: Simulation

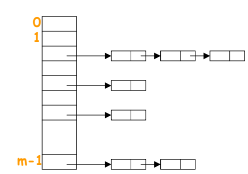
- Example: Given a probabilistic model of bank-customer arrival times and transaction times, how many tellers are needed
  - Assume we have a way to generate random inter-arrival times
  - Assume we have a way to generate transaction times
  - Can simulate the bank to get some idea of how long customers must wait

Time-Driven Simulation
- Check at each *tick* to see if any event occurs

Event-Driven Simulation
- Advance clock to next event, skipping intervening *ticks*
- This uses a PQ!

Another PQ Implementation

- If there are only a few possible priorities then can use an array of queues
  - Each array position represents a priority (0..m-1 where m is the array size)
  - Each queue holds all items that have that priority
- One text [Skiena] calls this a *bounded height priority queue*
- Time for insert: O(1)
- Time for getMax:
  - O(m) in the worst-case
- Example: airline check-in



Other PQ Operations

delete
a particular item

update
an item (change its priority)

join
two priority queues

- For delete and update, we need to be able to find the item
  - One way to do this: Use a Dictionary to keep track of the item's position in the heap
- Efficient joining of 2 Priority Queues requires another data structure
  - Skew Heaps or Pairing Heaps
    - Chapter 23 in text
    - Not part of 211

Selecting a Priority Queue Scheme

- Use an unordered array for small sets (< 20 or so)
- Use a sorted array or sorted linked list if few insertions are expected
- Use an array of linked lists if there are few priorities
  - Each linked list is a queue of equal-priority items
  - Very easy to implement
- Otherwise, use a Heap if you can
- Heap + Hashtable
  - Allow *change-priority* operation to be done in O(log n) expected time
- Balanced tree schemes
  - Useful and practical
- There are a number of alternate implementations that allow additional operations
  - Skew heaps
  - Pairing heaps
  - Fibonacci heaps
  - ...