CS W3134: Data Structures in Java
Lecture #20: Hashing II, Heaps
11/18/04
Janak J Parekh

Administrivia
- Grades should be available from website

Agenda
- Finish hashing
  - Let's look at the book’s code first to get an idea of how it works
- Heaps
Maps and sets, redux

- Since hashtables don’t store the data in linear order, they can’t work as a list
- Sets – insert and verify – works fine
- Maps – insert and lookup – also work fine
- Both trees and hash tables are great for this, but hash tables can potentially be faster

Hash functions

- What makes a good hash function?
  - Fast to compute
- Random keys?
  - If already random distribution, just mod it
- Non-random keys
  - Need to “compress” information
  - Use as much data as possible
  - Table size should be prime
- Book’s String example on page 565

Hash functions and efficiency

- Folding: Break into groups and add together – for example, SSN
  - 1000 cells => 3-digit numbers
- Efficiency?
  - All O(1) in theory, but…
  - Load factor: % of table actually used – directly affects performance
Hashing efficiency, cont’d.

- In general, quadratic probing and double hashing fare better than linear probing as the load factor goes up.
- Separate chaining: linear function of load factor (can be > 1, since multiple entries per cell).
- Generally want to avoid high loads…

What can’t you do?

- Specific ordering – it’s essentially random.
- Growable – can’t use a linked list and maintain performance metrics.
- Expect it to be automagically fast – need good hash functions.
- Although Java does have a number of hash functions built in… `hashCode()`.

Heaps

- More efficient way of implementing a priority queue as opposed to array.
- Modeled as binary tree, but usually implemented as an array.
- Not a binary search tree, but instead a binary tree that fulfills the **heap property**: a node is larger (or smaller, depending) than all nodes below it.
- Given a node n, left is 2n+1 and right is 2n+2; parent is (n-1)/2.
- Complete binary tree: we fill each level from left-to-right.
- Performance: $O(\log n)$ insert and remove.
Heap operations

- **Insert**
  - If root, simple
  - If not, put it at the “end”, i.e., next leaf, and then bubble up until we hit the appropriate node

- **Remove**
  - Always “remove” the root
  - Take the last element and put it into the root to replace the removed element
  - Then, bubble (trickle) down
  - Bubbling doesn’t require individual swaps...

Other operations

- **Key change**
  - Given an index and a new value
  - Then bubble up or bubble down, depending on the situation
  - Finding the index can be a problem if it’s not supplied

- **Expanding array**
  - Just like a list – don’t need to rehash

Tree-based heaps

- Can represent heaps as real trees
- Parent pointers needed
- Advantage: growable
- Disadvantage: finding last node is a problem
  - Convert index into bitstring, and ignore the first digit
  - Then, 0 is left, 1 is right
- Don’t need to move nodes around, just values (why?)
**Heapsort**

- If we insert N elements into a heap...
- Then remove N elements...
- We’ve got a sorted heap!
- Can we make it more efficient?
  - Don’t bubble up for each new insert; instead, add everything and then start trickling (**heapify**)
  - Don’t need to trickle leaf nodes, just intermediate nodes, e.g. start at n/2-1 and work backwards from there
  - Recursive: heapify right heap, heapify left heap, and then trickle ourselves down (stopping condition is a leaf)

**Heapsort (II)**

- Other optimizations
  - Work within the same array
  - First, heapify
  - Then, remove and put at bottom of array (since one less element in heap)
- Advantage over quicksort: less sensitive to distribution of data – always O(n log n) time

**Next time**

- Finish heaps
- Start graphs