CS W3134: Data Structures in Java

Lecture #19: Trees
11/16/04
Janak J Parekh

Administrivia

- HW#3 returned today
- HW#5 out
  - Due two weeks from *today*, because of Thanksgiving
  - "Looks forward" a little; most (all?) will be covered today

Agenda

- Huffman trees
- Hashing

Huffman trees

- Goal: form trees that let us figure out short binary string prefixes for each letter
  - We can then represent each letter with fewer # of bits
  - Ordinarily, each letter eats 8 or 16 bits (what's a bit?)
- Procedure
  - Create unit trees with each character and its frequency
  - Put all of these in a priority queue sorted by frequency

Huffman trees (II)

- Procedure (cont'd)
  - While there's more than one element in the priority queue…
    - Pull off two elements
    - Combine them with a "blank" parent node, whose frequency is the sum of the two children
    - Push back onto priority queue
  - When priority queue has one element, pop it; that's the Huffman tree
- Navigating the tree
  - Left == 0, Right == 1

Quick review

- We've learned…
  - Array Lists
  - Linked Lists
  - Stacks
  - Queues
  - Trees
- Various performance metrics?
- We can do better on a number of them!

Hash Table

- Believe it or not, we can build a data structure that has O(1) performance for insert, search, remove
- Several disadvantages
  - Array-based, so sometimes difficult to expand
  - Performance can suffer based on various parameters
  - Can't visit items in order

Keys?

- In general, we want to make lookup by keys very fast
- In an array, the *index number* is the key
  - Not useful as a "real" key, as this number may change
  - But numbers are very fast.
- OK, so how do we use a "word" as a key?
  - We convert it to a number somehow

Here's a simple one…

- Take the numeric value of all the letters
  - a = 1, b = 2, … , z = 26
  - Add them together
  - Put the word in that cell
    - cats == 43
- How well would this work?
  - What's the minimum value?
  - What's the maximum value for a 10-letter word?
  - How many words could be in between?

A bit more sophisticated

- For each character, multiply it by 26 to the position
  - Always produces unique number for each word
- cats == $3 * 26^3 + 1 * 26^2 + 20 * 26^1 + 19 * 26^0$
- What's the minimum value?
- What's the maximum value for a 10-letter word?
- Why is this so inefficient?
- Need to *hash* this large value into a smaller one
  - How about % arraySize?
  - This is one of the simplest hash functions

Collisions

- All of this would be good if we could come up with a *perfect hash* function: one that maps every possible entry into a different cell
- Guess what? We usually can't, unless we know precisely what data we'll be inputting
- Several different methodologies to deal with this

Separate chaining

- Make each hash cell a "bucket" for multiple entries
- Use a linked list or array or similar construct to store the entries
- Must make sure lists don't get too long: good hash function
  - But much less sensitive to load factors than open addressing

Linear probing

- "Open addressing": Just put the result in another cell
- *Linear probing*: put it in the very next cell
  - Leads to "clusters" making the hash table very inefficient
- *Quadratic probing*: space 'em out
  - x+1, x+4, x+9, x+16, x+25
  - Wraparound if necessary
  - Has other clustering properties

Double hashing

- Another form of open addressing
- Hash the key using a different function, and use that result as a step size (x+y)
  - Hash function must *never* return a zero, and should not be the same as the first hash function
  - stepSize = constant – (key % constant)
  - (constant is a prime less than table size)
- Table size must be prime

Double hashing, cont'd

- Other considerations
  - Duplicates are a problem with this method
  - Deletes?
  - Consider expanding the array: rehashing required
    - Load factor of the hash table very important

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…

Next time

- Heaps