CSC148 fall 2013

binary search tree
week 8

Danny Heap
heap@cs.toronto.edu
BA4270 (behind elevators)
http://www.cdf.toronto.edu/~heap/148/F13/
416-978-5899

October 29, 2013

Computer Science
UNIVERSITY OF TORONTO

Outline

performance

binary search tree

big-oh

performance. . .

We want to measure **algorithm** performance, independent of hardware, programming language, random events

Focus on the **size** of the input, call it n. How does this affect the resources (e.g. processor time) required for the output? If the relationship is linear, our algorithm's complexity is $\mathcal{O}(n)$ — roughy proportional to the input size n.

list searching

You've already seen algorithms for seeing whether an element is contained in a `list`:

`[97, 36, 48, 73, 156, 947, 56, 236]`

What is the performance of these algorithms in terms of list size? What about the analogous algorithm for a tree?

a more efficient binary tree

We need to impose a sorting condition on binary trees. A **binary search tree** is:

- a binary tree
- left subtree of every node contains only values smaller than those of that node
- right subtree of every node contains only values greater than those of that node

efficiency?

Binary search of a list allowed us to ignore (roughly) half the list. Searching a binary search tree allows us to ignore the left or right subtree.

If we're searching the tree rooted at node n for value v, then one of three situations are possible:

- node n has value v
- v is less than node n's value, so we should search to the left
- v is more than node n's value, so we should search to the right

insert

Inserting is closely related to finding a node:

- if we find a node in our tree, no need to insert it!
- otherwise, we find the spot it should be, and insert it there.

deleting

deleting is a bit trickier, because there are several scenarios to consider, even after we've figured out which node we wish to delete:

- if the node we wish to delete is a leaf, just delete it
- if the node we wish to delete has exactly one child, replace it with the other
- if the node we wish to delete has two children, replace it with the largest child in its left subtree...

You should draw some diagrams until you understand these scenarios

running time analysis

worst-case.

algorithm's behaviour over large input (size n) is common way to compare performance

- constant: $c \in \mathbb{R}^+$ (some positive number) ← list dereference.
- logarithmic: $c \log n$ — binary search.
- linear: cn (probably not hte same c) → linear search.
- quadratic: cn^2 — bubble (other sorts)
- cubic: cn^3
- exponential: $c2^n$ → toah, — four-stools ≪ exponential
- horrible: cn^n or $cn!$

merge + quick.

running time analysis

deliberate sloppiness

abstract away difference between similar worst-case performance, e.g.

- one algorithm runs in $(0.3365n^2 + 0.17n + 0.32)\mu s$
  = micro-seconds
- another algorithm runs in $(0.47n^2 + 0.08n)\mu s$
- in both cases doubling n quadruples the run time. We say both algorithms are $\mathcal{O}(n^2)$ or "order n^2" or "oh-n-squared" behaviour.

asymptotics

If any reasonable implementation of an algorithm, on any reasonable computer, runs in time **no more than** $cg(n)$ (some constant c), we say the algorithm is $\mathcal{O}(g(n))$. Graphing various examples where $g(n) = n^2$ shows how we ignore the c as n gets large (say $7n^2, 2n^2 + 1$ versus $4n + 2$, $n = 12$).

case: lg n

this is the number of times you can divide n in half before reaching 1.

- refresher: $a^b = c$ means $\log_a c = b$.
- this runtime behaviour often occurs when we "divide and conquer" a problem (e.g. binary search)
- we usually assume lg n (log base 2), but the difference is only a constant:

$$2^{\log_2 n} = n = 10^{\log_{10} n} \Longrightarrow \log_2 n = \log_2 10 \times \log_{10} n$$

- so we just say $\mathcal{O}(\lg n)$.

hierarchy

Since big-oh is an **upper-bound** the various classes fit into a hierarchy:

$$\mathcal{O}(1) \subseteq \mathcal{O}(\lg n) \subseteq \mathcal{O}(n) \subseteq \mathcal{O}(n^2) \subseteq \mathcal{O}(n^3) \subseteq \mathcal{O}(2^n) \subseteq \mathcal{O}(n^n)$$