Welcome to daylight savings!
- A1 — we can re-run unit tests at a small discount...
- A2 — regextree.py now posted
- E3 — due Thurs.

CSC148 winter 2014
BSTs, big-Oh
week 9

Danny Heap
heap@cs.toronto.edu
BA4270 (behind elevators)
http://www.cdf.toronto.edu/~heap/148/W14/
416-978-5899

March 10, 2014
Outline

performance

big-o
instead of single tree class, separate node and bst classes:

class BTNode:
    """Binary Tree node."""

    def __init__(self: 'BTNode', data: object,
                 left: 'BTNode'=None,
                 right: 'BTNode'=None) -> None:
        """Create BT node with data, children left and right."""
        self.data, self.left, self.right = data, left, right
binary search tree

Add a condition: data in left subtree is less than that in the root, which in turn is less than that in right subtree. Now search is more efficient...

class BST:
    """Binary search tree."""

    def __init__(self: 'BST', root: BTNode=None) -> None:
        """Create BST with BTNode root."""
        self._root = root

also, possibly, record size other features.
deletion of data from BST rooted at node?

- what return value?

- what to do if node is None?

- what if data to delete is less than that at node?

- what if it’s more?

- what if the data equals this node’s data and...

  - this node has no left child

  - ... no right child?

  - both children?
recall 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?

linear (length $n$, $\mathcal{O}(n)$ steps)
BST 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 — nearly half in a well-balanced tree. 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
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 $O(n)$ — roughly proportional to the input size $n$. 
running time analysis

algorithm’s behaviour over large input (size $n$) is common way to compare performance — how does performance vary as $n$ changes?

- **constant:** $c \in \mathbb{R}^+$ (some positive number)
- **logarithmic:** $c \log n$ → binary search
- **linear:** $cn$ (probably not the same $c$) → linear search
- **quadratic:** $cn^2$
- **cubic:** $cn^3$
- **exponential:** $c2^n$
- **horrible:** $cn^n$ or $cn!$ → bogosort, matrix
running time analysis

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

- one algorithm runs in $(0.3365n^2 + 0.17n + 0.32)\mu s$

- 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 $O(n^2)$ or “order $n^2$” or “oh-n-squared” behaviour.
If any reasonable implementation of an algorithm, on any reasonable computer, runs in number of steps no more than $cg(n)$ (some constant $c$), we say the algorithm is $O(g(n))$. Graphing various examples where $g(n) = n^2$ shows how we ignore the constant $c$ as $n$ gets large (say $7n^2, 2n^2 + 1$ versus $43n + 2, n = 1297$).
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:

\[
2^{\log_2 n} = n = 10^{\log_{10} n} \implies \log_2 n = \log_2 10 \times \log_{10} n
\]

- so we just say \( O(\lg n) \).
hierarchy

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

\[ O(1) \subseteq O(\lg n) \subseteq O(n) \subseteq O(n^2) \subseteq O(n^3) \subseteq O(2^n) \subseteq O(n^n) \]