

# 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 12, 2014

# Outline

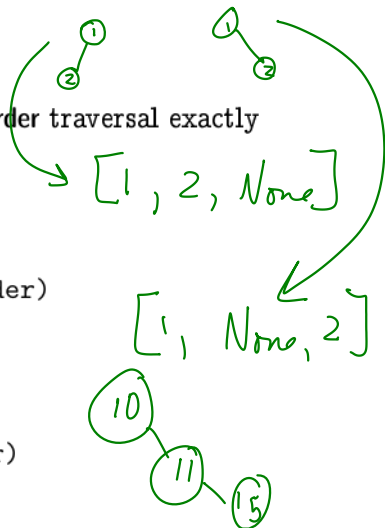
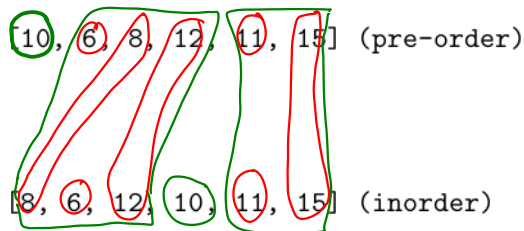
performance

big-oh

# traversal task...

by hand

on its own, neither a preorder nor inorder traversal exactly specify a tree, but together...



## wrapper/node binary tree

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

```
class BTreeNode:
    """Binary Tree node."""

    def __init__(self: 'BTreeNode', data: object,
                 left: 'BTreeNode'=None,
                 right: 'BTreeNode'=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
```



## 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?





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  $\mathcal{O}(n)$  — roughly proportional to the input size  $n$ .

# running time analysis

$$\frac{n^2 + n}{2} - 1$$

$$\begin{array}{ccccccc}
 1 & + & 2 & + & 3 & + & \dots & + & n \\
 n & + & n-1 & + & n-2 & + & & + & 1 \\
 \hline
 1 & + & 2 & + & 3 & + & \dots & + & n \\
 \hline
 1 & + & 2 & + & 3 & + & \dots & + & n
 \end{array}
 = \frac{n(n+1)}{2}$$

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:**  $c \log n$

**linear:**  $cn$  (probably not the same  $c$ )

**quadratic:**  $cn^2$

**cubic:**  $cn^3$

**exponential:**  $c2^n$

"horrible":  $cn^n$  or  $cn!$   
 "beaut, f.w."

## less-than-stellar sorting...

$$\frac{n^2 + n}{2} + n - 1$$

express some crude “number of steps” for these algorithms — ignore differences between steps that do not depend on the list size  $n$

$$1 + \dots + n - 1 + n = \frac{n(n+1)}{2} + \underline{(n-1)}$$

**selection sort:** for each list position from 0 to  $n-2$ , linear-search the remaining elements to find the minimum, and if it is smaller than the element at the current position, swap them.

$$1 + 2 + 3 + \dots + n - 1 = \frac{n(n+1)}{2} - n$$

**insertion sort:** for each list position from 1 to the end of the list, compare it to each previous element until you find one that is not larger than it, and insert element there.

## 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  $\mathcal{O}(n^2)$  or “order  $n^2$ ” or “oh-n-squared” behaviour.

# asymptotics

If any reasonable implementation of an algorithm, on any reasonable computer, runs in number of steps no more than  $cg(n)$  (some constant  $c$ ), we say the algorithm is  $\mathcal{O}(g)$ .

Graphing various examples where  $g(n) = n^2$  shows why 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  $\mathcal{O}(\lg n)$ .

# hierarchy

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

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

