CSC148 winter 2014

BSTs, big-Oh

week 9

Danny Heap / Dustin Wehr
heap@cs.toronto.edu / dustin.wehr@utoronto.ca
BA4270 / SF4306D
http://www.cdf.toronto.edu/~heap/148/F13/

March 13, 2014

Outline

binary search tree — deletion

performance

big-oh

Term test remarking note

```
If you parsed
  return 1 + sum_or_max(...) if some_test else 0_or_1
as
  return 1 + (sum_or_max(...) if some_test else 0_or_1)
instead of
  (1 + sum_or_max(...)) if some_test else 0_or_1
but otherwise traced the function correctly,
you can probably get some points back.
```

A common mistake on A1

```
# Code here (before the main block) should NEVER
# reference variables such as x defined in the main block.
def f():
    print(x)    # BAD. Exception unless f is called
                # from the main block. If that is
                # the only way your code is meant
                # to be executed, you don't need
                # the main block.

if __name__ == '__main__':
    x = 5
    f(x)
```

last week...

Remember this and how it's an inorder.. wait I mean a postorder.. wait nevermind forget about it.

```
def _str(b: BTNode, i: str) -> str:
    """Return a string representing self inorder,
    indent by i"""
    return ((_str(b.right, i + '    ') if b.right else '') +
            i + str(b.data) + '\n' +
            (_str(b.left, i + '    ') if b.left else ''))
```

It is a reversed inorder traversal.

traversal task...

by hand

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

`[10, 6, 8, 12, 11, 15] (pre-order)`

`[8, 6, 12, 10, 11, 15] (inorder)`

recall: wrapper/node binary tree

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
```

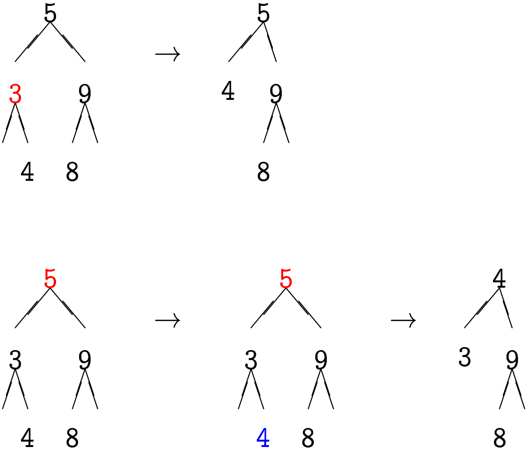
recall: binary search tree

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
```

deletion of data from BST



deletion of data from BST rooted at node?

- what return value?
  Consider case of deleting root: must return a *different* node.
- what to do if node is None?
  Return None. More generally, if data is not in tree, tree is unmodified and return the current root.
- what if data to delete is less than that at node?
  Try deleting data in left subtree. Then return this 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]
```

```
def search(x):
    for y in L:
        if x == y:
            return True
    return False
```

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

binary search of a sorted list

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

```
def search(L,x):
  if len(L) <= 1:
    return len(L) == 1 and x == L[0]
  mid = len(L)//2
  if x == L[mid]:
    return True
  elif x < L[mid]:
    return search(L[0:mid], x)
  else:
    return search(L[mid+1:len(L)], x)
```

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

BST efficiency?

Binary search of a list allowed us to ignore (roughly) half the list, and (roughly) half of the non-ignored sublist, and so on.

Searching a **binary search tree** allows us to ignore the left or right subtree — nearly half in a well-balanced tree, and then one of the subtrees of the non-ignored subtree, and so on.

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

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.

less-than-stellar sorting...

```
def sort(L:list):                  # Let's look at two sorting
    # some initializing            # algs of this form
    for i in range(len(L)):
        # do something
```

Selection sort:
What we've accomplished by the start of i-th iteration: We've put the smallest i elements of the list in their final places (in the first i positions of the list).
What we do next: select the smallest element from the remaining $n - i$ right-most positions, and swap it into position i.

Insertion sort:
What we've accomplished by the start of i-th iteration: The first i positions of the list are sorted, though the elements may not be in their final positions. The right-most $n - i$ elements are untouched.
What we do next: Take the next element (at position i) and insert it into its proper place in the left-most $i + 1$ positions.

less-than-stellar sorting...

Express some crude "number of steps" for these algorithms — ignore differences between steps that do not depend on the list size **n**

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.

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

algorithm's behaviour over large input (size **n**) is common way to compare performance — how does performance vary as **n** increases?

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 $cn!$

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.

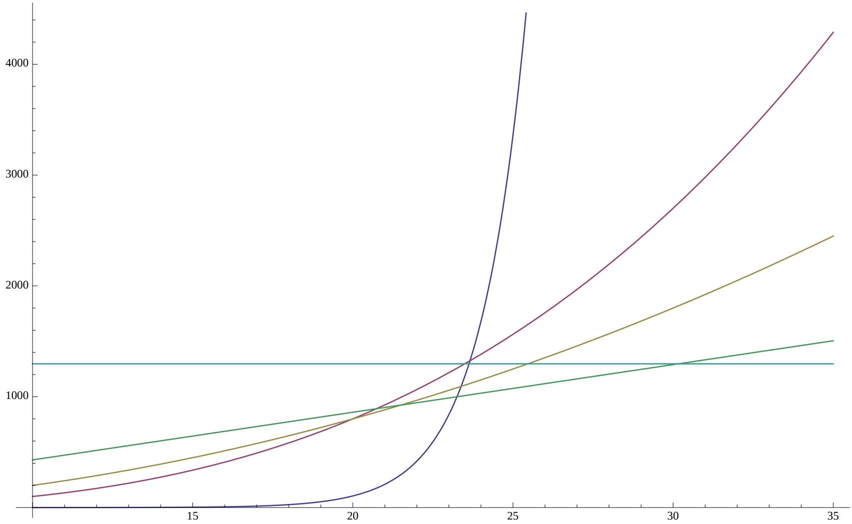
asymptotics

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 $\mathcal{O}(g(n))$.

Graphing various examples shows how we ignore the constant c as n gets large.

Compare

- $g(n) = .0001 \times 2^n$
- $g(n) = .1 \times n^3$
- $g(n) = 2n^2$
- $g(n) = 43n$
- $g(n) = 1297$



For $.0001 \times 2^n, .1 \times n^3, 2n^2, 43n$, and 1297, big-$\mathcal{O}$ takes over fully around $n = 30$.

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:

$$\begin{aligned} 2^{\log_2 n} = & \quad n & = 10^{\log_{10} n} \\ \Rightarrow \quad & \log_2 n & = \log_2(10^{\log_{10} n}) = \log_2 10 \times \log_{10} n \end{aligned}$$

[recall $\log_x y^z = (\log_x y) \times z$]

- so we just say $\mathcal{O}(\lg n)$.
- $\mathcal{O}(\lg n)$ is the run time of binary search of a sorted list, etc

hierarchy

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

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