CSCI 136:
Data Structures
and
Advanced Programming

Lecture 25

Trees, part 4

Instructor: Dan Barowy

Williams

Topics

More BST methods

Tree balance

Big-O

Implicit BST

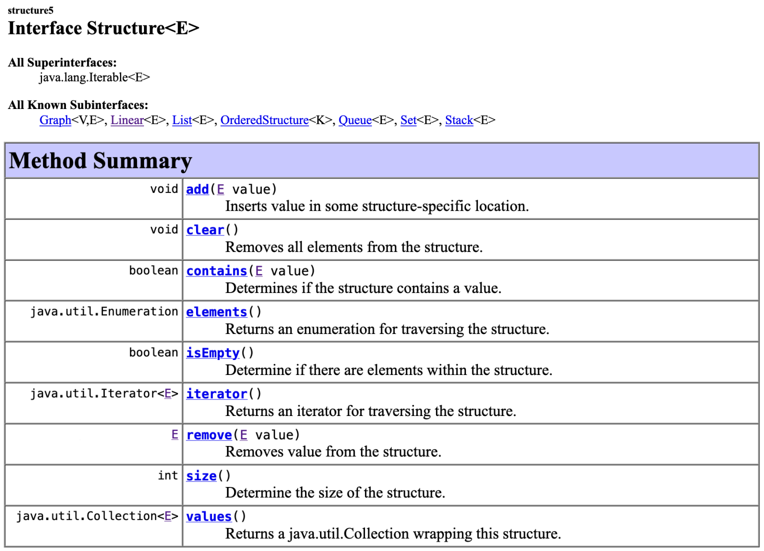
Your to-dos

1. Read **before Mon**: Bailey, Ch 15.15-3.
2. Lab 8 **(solo lab)**, **due Tuesday 11/15 by 10pm**.
3. Quiz, **open now**, due Saturday evening.

Binary Search Tree

Let's implement this.

Should it be a `structure`?



structure5

Interface Structure<E>

All Superinterfaces:
java.lang.Iterable<E>

All Known Subinterfaces:
Graph<V,E>, Linear<E>, List<E>, OrderedStructure<K>, Queue<E>, Set<E>, Stack<E>

Method Summary	
`void`	`add(E value)` Inserts value in some structure-specific location.
`void`	`clear()` Removes all elements from the structure.
`boolean`	`contains(E value)` Determines if the structure contains a value.
`java.util.Enumeration`	`elements()` Returns an enumeration for traversing the structure.
`boolean`	`isEmpty()` Determine if there are elements within the structure.
`java.util.Iterator<E>`	`iterator()` Returns an iterator for traversing the structure.
`E`	`remove(E value)` Removes value from the structure.
`int`	`size()` Determine the size of the structure.
`java.util.Collection<E>`	`values()` Returns a java.util.Collection wrapping this structure.

Binary Search Tree

At home: how is `remove` implemented?

Binary Search Tree

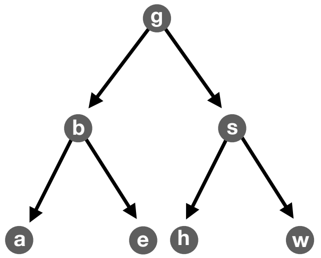
How might an iterator perform a given traversal?

Hint: use a stack!

Hint: the stack maintains all of the elements that still need to be traversed.

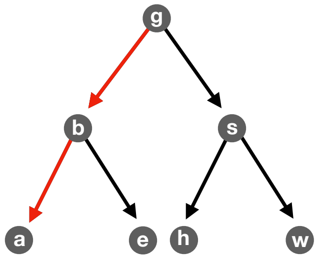
Tree balance

In the **worst case**, how long does it take to find an element in this binary search tree?



Suppose it is the letter ⓐ.

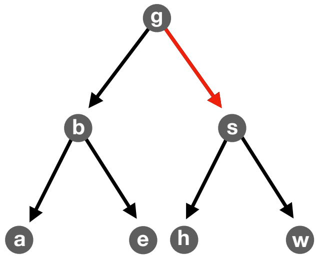
In the **worst case**, how long does it take to find an element in this binary search tree?



Suppose it is the letter ⓐ.

Finding ⓐ takes **two steps**.

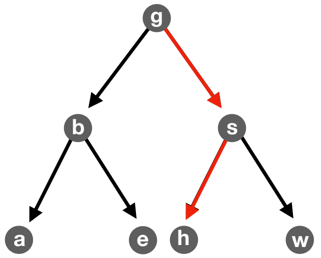
In the **worst case**, how long does it take to find an element in this binary search tree?



Suppose it is the letter ⓢ.

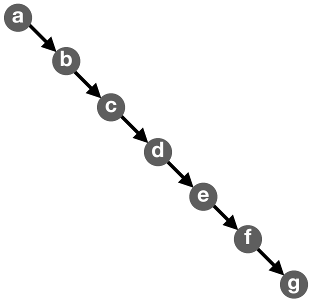
Finding ⓢ takes **one step**.

In the **worst case**, how long does it take to find an element in this binary search tree?



In the **worst case**, the time depends on the **length** of the **longest path**.

Suppose a friend gives you the following sequence of values: `[a,b,c,d,e,f,g]`



Ouch!!!

Worst case: O(n)

And asks you to store them in a binary tree to "make accessing them fast."

Is access **guaranteed** to be **fast**?

But what if your tree maintained the following property **on insertion**? (i.e., it is always true)

`isBalanced(t):`

`t` is balanced if and only if
- `t` is empty, or
- **all** of the following
  - `isBalanced(t.left)` is `true` **and**
  - `isBalanced(t.right)` is `true` **and**
  - `|height(t.left) - height(t.right)|` ≤ `1`

Keep in mind: we know that the worst case has something to do with **height**.

But what if your tree maintained the following property **on insertion**? (i.e., it is always true)

Clearly a balanced tree.

Yeah, sure, there's no tree. Details, details…

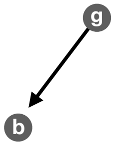
Time to access an element ~ **0 steps**

But what if your tree maintained the following property **on insertion**? (i.e., it is always true)

Balanced? **Yes.**

Max time to access an element ~ **0 steps**

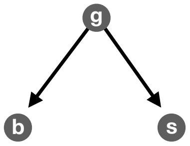
But what if your tree maintained the following property **on insertion**? (i.e., it is always true)



Balanced? **Yes.**

Max time to access an element: **1 step**

But what if your tree maintained the following property **on insertion**? (i.e., it is always true)

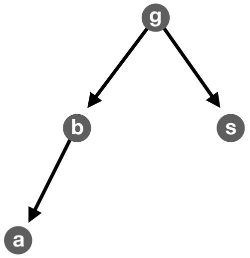


Balanced? **Yes.**

Changes nothing.

Max time to access an element: **1 step**

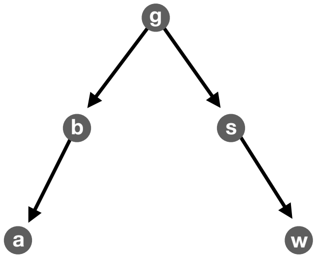
But what if your tree maintained the following property **on insertion**? (i.e., it is always true)



Balanced? **Yes.**

Max time to access an element: **2 steps**

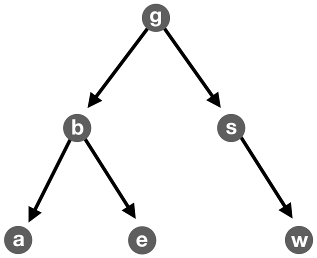
But what if your tree maintained the following property **on insertion**? (i.e., it is always true)



Balanced? **Yes.**

Max time to access an element: **2 steps**

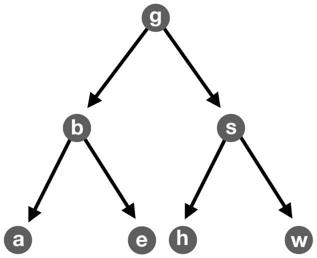
But what if your tree maintained the following property **on insertion**? (i.e., it is always true)



Balanced? **Yes.**

Max time to access an element: **2 steps**

But what if your tree maintained the following property **on insertion**? (i.e., it is always true)



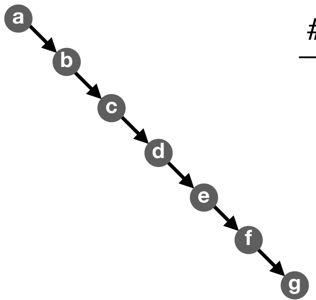
Balanced? **Yes.**

Max time to access an element: **2 steps**

# nodes	max time
1	**0 steps**
2	**1 step**
3	**1 step**
4	**2 steps**
5	**2 steps**
6	**2 steps**
7	**2 steps**
8	**3 steps**
…	**…**

This looks like **time** = log$_2$(# nodes)

But does this hold up?



# nodes	max time
7	**6 steps**

Clearly **not** a balanced tree.

Logarithmic worst-case access time has something to do with the **compactness** of a tree; **height matters**.

Recap & Next Class

Today:

- Tree balance
- BST asymptotics
- Implicit BST

Next class:

- Maps