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.

Let's implement this.
Should it be a **structure**?

**Method Summary**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void add(E value)</td>
<td>Inserts value in some structure-specific location.</td>
</tr>
<tr>
<td>void clear()</td>
<td>Removes all elements from the structure.</td>
</tr>
<tr>
<td>boolean contains(E value)</td>
<td>Determines if the structure contains a value.</td>
</tr>
<tr>
<td>java.utilEnumeration elements()</td>
<td>Returns an enumeration for traversing the structure.</td>
</tr>
<tr>
<td>boolean isEmpty()</td>
<td>Determines if there are elements within the structure.</td>
</tr>
<tr>
<td>java.util.Iterator&lt;E&gt; iterator()</td>
<td>Returns an iterator for traversing the structure.</td>
</tr>
<tr>
<td>E remove(E value)</td>
<td>Removes value from the structure.</td>
</tr>
<tr>
<td>int size()</td>
<td>Determines the size of the structure.</td>
</tr>
<tr>
<td>java.util.Collection&lt;E&gt; values()</td>
<td>Returns a java.util.Collection wrapping this structure.</td>
</tr>
</tbody>
</table>

Binary Search Tree

At home: how is `remove` implemented?

**Hint:** use a stack!

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

Binary Search Tree

How might an iterator perform a given traversal?

**Hint:** use a stack!
In the **worst case**, how long does it take to find an element in this binary search tree?

Suppose it is the letter **a**.

Finding **a** 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 **s**.

Finding **s** takes **one step**.

In the **worst case**, the time depends on the **length** of the longest path.
Suppose a friend gives you the following sequence of values: \([a, b, c, d, e, f, g]\)

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

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

Ouch!!!

Worst case: \(O(n)\)

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
  - \(|\text{height}(t.left) - \text{height}(t.right)| \leq 1\)

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

<table>
<thead>
<tr>
<th># nodes</th>
<th>max time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0 steps</td>
</tr>
<tr>
<td>2</td>
<td>1 step</td>
</tr>
<tr>
<td>3</td>
<td>1 step</td>
</tr>
<tr>
<td>4</td>
<td>2 steps</td>
</tr>
<tr>
<td>5</td>
<td>2 steps</td>
</tr>
<tr>
<td>6</td>
<td>2 steps</td>
</tr>
<tr>
<td>7</td>
<td>2 steps</td>
</tr>
<tr>
<td>8</td>
<td>3 steps</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

This looks like $time = \log_2(#\ nodes)$

But does this hold up?

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