CSCI 136: Data Structures and Advanced Programming
Lecture 26
Trees, part 4

Instructor: Dan Barowy Williams

Topics
More BST methods
Tree balance
Big-O
Implicit BST

Announcements

Spring pre-registration begins Wed, April 27 and runs until Fri, May 6.

The best way to get into the CS course you want is to pre-register.

Common “next steps” after CSCI 136:
CSCI 237: Computer Organization
CSCI 256: Algorithms
CSCI 334: Principles of Programming Languages
also, some electives.

Practice Quiz
Let's add `find` and `contains`.

**Binary Search Tree**

<table>
<thead>
<tr>
<th>Method Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>void add(E value)</strong></td>
</tr>
<tr>
<td><strong>void clear()</strong></td>
</tr>
<tr>
<td><strong>boolean contains(E value)</strong></td>
</tr>
<tr>
<td><strong>java.utilEnumeration elements()</strong></td>
</tr>
<tr>
<td><strong>boolean isEmpty()</strong></td>
</tr>
<tr>
<td><strong>java.util.Iterator&lt;E&gt; iterator()</strong></td>
</tr>
<tr>
<td><strong>E remove(E value)</strong></td>
</tr>
<tr>
<td><strong>int size()</strong></td>
</tr>
<tr>
<td><strong>java.util.Collection&lt;E&gt; values()</strong></td>
</tr>
</tbody>
</table>

**Should it be a **structure**?**

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.
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**.

Suppose it is the letter $s$.

Finding $s$ takes **one step**.
In the **worst case**, how long does it take to find an element in this binary search tree?

![Binary Search Tree Diagram]

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

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

![Binary Search Tree Diagram]

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)

\[
\text{isBalanced}(t): \\
\text{t is balanced if and only if} \\
\text{• t is empty, or} \\
\text{• all of the following} \\
\text{• isBalanced}(t\.left) \text{ is true and} \\
\text{• isBalanced}(t\.right) \text{ is true and} \\
\text{• }|\text{height}(t\.left) - \text{height}(t\.right)| \leq 1
\]

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)

![Image of Balanced Tree]

Clearly a balanced tree.

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

Time to access an element \(\sim 0\) 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

Balanced? Yes.
Max time to access an element: 2 steps

Balanced? Yes.
Max time to access an element: 2 steps

This looks like \( \text{time} = \log_2(\text{# nodes}) \)

But does this hold up?
Clearly not a balanced tree.

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

Worst case time is $O(\log_2(n))$ for a balanced binary tree. Why?

What is min. binary tree height needed to store $n$ nodes?

Cute theorem: height $\geq \lfloor \log_2(n) \rfloor$

Intuition: $\log_2(n)$ is the number of times you can divide $n$ nodes in halves.
Recall: binary search tree

A **binary search tree** is a binary tree that maintains the **binary search property** as elements are added or removed. In other words, the **key** in each node:

- must be $\geq$ any key stored in the left subtree, and
- must be $\leq$ any key stored in the right subtree.

As with other ordered structures, order is maintained on insertion.

BST is an ADT

Do we actually need a **tree** to store a **tree**?

No. We can use an **implicit data structure** instead.

Implicit data structure

A **implicit data structure** or **space-efficient data structure** is a data structure that stores only necessary information. Instead of explicitly representing relationships between elements of the structure using references, an implicit structure uses the relative positions of elements.

Implicit binary tree

A diagram of an implicit binary tree is shown, with nodes labeled from a to g, arranged in a way that reflects the relative positions of elements.
Implicit binary tree

```
Implicit relationship
```

leftChild(i) = 2 \times i + 1
rightChild(i) = 2 \times i + 2
parent(i) = \left\lfloor (i - 1) / 2 \right\rfloor

I will post an implementation on the course website.

Recap & Next Class

Today:
Tree balance
BST asymptotics
Implicit BST

Next class:
Maps