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 ≥ any key stored in the left subtree, and
• must be < any key stored in the right subtree.

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

Your to-dos

1. Read before Wed: Bailey, Ch 15.4.
2. Lab 8 (solo lab), due Tuesday 11/15 by 10pm.
3. Quiz, due Saturday evening.
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 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?

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**?

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

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

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 \(\sim 0 \text{ steps}\)

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**.

Ouch!!!

Worst case: \(O(n)\)
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 \( \text{time} = \log_2(\# \text{ nodes}) \)

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: \( \text{height} \geq \lfloor \log_2(n) \rfloor \)

Intuition: $\log_2(n)$ is the number of times you can divide $n$ nodes in halves.
A map (also known as a dictionary, associative array, or key-value store) is an abstract data type that stores a collection of (key, value) pairs. Each key appears at most once in a collection. Maps support lookup, insert, and remove operations.

More formally, a map is a function with a finite domain.

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dan</td>
<td>C</td>
</tr>
<tr>
<td>Jeannie</td>
<td>A</td>
</tr>
<tr>
<td>Bill J</td>
<td>B</td>
</tr>
<tr>
<td>Iris</td>
<td>A</td>
</tr>
<tr>
<td>Sam</td>
<td>A+</td>
</tr>
</tbody>
</table>

You've seen something like this before (hint: SymbolTable)

---

**Map ADT**

**Map ADT (intuition)**

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dan</td>
<td>C</td>
</tr>
<tr>
<td>Jeannie</td>
<td>A</td>
</tr>
<tr>
<td>Bill J</td>
<td>B</td>
</tr>
<tr>
<td>Iris</td>
<td>A</td>
</tr>
<tr>
<td>Sam</td>
<td>A+</td>
</tr>
</tbody>
</table>

---

**structure5 Map interface**

```java
interface Map<K,V>
```

(I omitted some methods— see structure5 docs)

---

**structure5’s only Map implementation**

```java
public class MapList<K,V> extends java.lang.Object
 implements Map<K,V>
```

Associations establish a link between a key and a value. An associative array or map is a structure that allows a disjoint set of keys to become associated with an arbitrary set of values. The convenience of an associative array is that the values used to index the elements need not be comparable and their range need not be known ahead of time. Furthermore, there is no upper bound on the size of the structure. It is able to maintain an arbitrary number of different pieces of information simultaneously. Maps are sometimes called dictionaries because of the uniqueness of the association of words and definitions in a household dictionary.

This implementation is based on a list, so performance for most operations is linear.

What's the problem with this implementation?
Let's create a tree-backed Map

But first: how will it perform?

Recap & Next Class

**Today:**
- Map interface
- Tree backed map

**Next class:**
- Hash tables
- Collisions