Warm Up

1. Write a recurrence for this piece of code (assume each node has exactly 1 or 2 children)

```java
private IntTreeNode doublePositivesHelper(IntTreeNode node) {
    if (node != null) {
        if (node.data > 0) {
            node.data *= 2; + C
        }
        node.left = doublePositivesHelper(node.left); T(n/2)
        node.right = doublePositivesHelper(node.right); T(n/2)
        return node; + C
    } else {
        return null; + C
    }
}
```

\[
T(n) = \begin{cases} 
    C_1 & \text{if } n < 1 \\
    C_2 + 2T \left( \frac{n}{2} \right) & \text{otherwise}
\end{cases}
\]

Extra Credit:
Go to PollEv.com/champk
Text CHAMPK to 22333 to join session, text "1" or "2" to select your answer
Warm Up Continued

\[ T(n) = \begin{cases} 
C_1 & \text{when } n < 1 \\
C_2 + 2T\left(\frac{n}{2}\right) & \text{otherwise} 
\end{cases} \]

<table>
<thead>
<tr>
<th>Level (i)</th>
<th>Number of Nodes</th>
<th>Input to Node</th>
<th>Work per Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>base</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Input = \( n \)
- work = \( C_2 \)
- \text{input} = \frac{n}{2}
- work = \( C_2 \)
Warm Up Continued

1. How many nodes on each branch level?
   \[2^i\]

2. How much work for each branch node?
   \[c_2\]

3. How much work per branch level?
   \[2^i c_2\]

4. How many branch levels?
   \[\frac{n}{2^i} < 1 \Rightarrow i = \log_2 n\]

5. How much work for each leaf node?
   \[c_1\]

6. How many leaf nodes?
   \[2^{\log_2 n} + 1 = 2n\]

\[T(n) = \begin{cases} 
  C_1 & \text{when } n < 1 \\
  C_2 + 2T\left(\frac{n}{2}\right) & \text{otherwise}
\end{cases}\]

<table>
<thead>
<tr>
<th>Level (i)</th>
<th>Number of Nodes</th>
<th>Input to Node</th>
<th>Work per Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2^0 = 1</td>
<td>(\frac{n}{2^0} = n)</td>
<td>(c_2)</td>
</tr>
<tr>
<td>1</td>
<td>2^1 = 2</td>
<td>(\frac{n}{2^1} = \frac{n}{2})</td>
<td>(c_2)</td>
</tr>
<tr>
<td>2</td>
<td>2^2 = 4</td>
<td>(\frac{n}{2^2} = \frac{n}{4})</td>
<td>(c_2)</td>
</tr>
<tr>
<td>3</td>
<td>2^3 = 8</td>
<td>(\frac{n}{2^3} = \frac{n}{8})</td>
<td>(c_2)</td>
</tr>
<tr>
<td>base</td>
<td>2^{\log_2 n} + 1</td>
<td>0</td>
<td>(C_1)</td>
</tr>
</tbody>
</table>

Combining it all together...

\[T(n) = \sum_{i=0}^{\log_2 n} 2^i c_2 + 2n c_1\]
Trees
Storing Sorted Items in an Array

get() – $O(\log n)$

put() – $O(n)$

remove() – $O(n)$

Can we do better with insertions and removals?
**Review: Trees!**

A **tree** is a collection of nodes
- Each node has at most 1 parent and 0 or more children

**Root node:** the single node with no parent, “top” of the tree

**Branch node:** a node with one or more children

**Leaf node:** a node with no children

**Edge:** a pointer from one node to another

**Subtree:** a node and all its descendants

**Height:** the number of edges contained in the longest path from root node to some leaf node
Tree Height

What is the height of the following trees?

- Overall height = 2
- Overall height = 0
- Overall height = -1 or NA
Traversals

**traversal**: An examination of the elements of a tree.
- A pattern used in many tree algorithms and methods

**Common orderings for traversals:**
- **pre-order**: process root node, then its left/right subtrees
  - 17 41 29 6 9 81 40
- **in-order**: process left subtree, then root node, then right
  - 29 41 6 17 81 9 40
- **post-order**: process left/right subtrees, then root node
  - 29 6 41 81 40 9 17

**Traversal Trick: Sailboat method**
- Trace a path around the tree.
- As you pass a node on the proper side, process it.
  - pre-order: left side
  - in-order: bottom
  - post-order: right side
A binary search tree is a binary tree that contains comparable items such that for every node, all children to the left contain smaller data and all children to the right contain larger data.
Implement Dictionary

Binary Search Trees allow us to:
- quickly find what we’re looking for
- add and remove values easily

Dictionary Operations:
Runtime in terms of height, “h”
get() – O(h)
put() – O(h)
remove() – O(h)

What do you replace the node with?
Largest in left sub tree or smallest in right sub tree
Height in terms of Nodes

For “balanced” trees $h \approx \log_c(n)$ where $c$ is the maximum number of children

Balanced binary trees $h \approx \log_2(n)$

Balanced trinary tree $h \approx \log_3(n)$

Thus for balanced trees operations take $\Theta(\log_c(n))$
Unbalanced Trees

Is this a valid Binary Search Tree?
Yes, but...

We call this a **degenerate tree**

For trees, depending on how balanced they are,
Operations at worst can be $O(n)$ and at best can be $O(\log n)$

How are degenerate trees formed?
- insert(10)
- insert(9)
- insert(7)
- insert(5)
Measuring Balance

Measuring balance:

For each node, compare the heights of its two sub trees

Balanced when the difference in height between sub trees is no greater than 1
Meet AVL Trees

**AVL Trees** must satisfy the following properties:

- **binary trees:** all nodes must have between 0 and 2 children
- **binary search tree:** for all nodes, all keys in the left subtree must be smaller and all keys in the right subtree must be larger than the root node
- **balanced:** for all nodes, there can be no more than a difference of 1 in the height of the left subtree from the right. \( \text{Math.abs(height(left subtree) – height(right subtree))} \leq 1 \)

AVL stands for Adelson-Velsky and Landis (the inventors of the data structure)
Is this a valid AVL tree?

Is it...
- Binary: yes
- BST: yes
- Balanced?: yes
Is this a valid AVL tree?

Is it...
- Binary  yes
- BST  yes
- Balanced?  no
Is this a valid AVL tree?

Is it...
- Binary: yes
- BST: no
- Balanced?: yes

9 > 8
Implementing an AVL tree dictionary

Dictionary Operations:

get() – same as BST
containsKey() – same as BST
put() - Add the node to keep BST, fix AVL property if necessary
remove() - Replace the node to keep BST, fix AVL property if necessary

Unbalanced!
Warm Up
Meet AVL Trees

**AVL Trees** must satisfy the following properties:
- **binary trees**: all nodes must have between 0 and 2 children
- **binary search tree**: for all nodes, all keys in the left subtree must be smaller and all keys in the right subtree must be larger than the root node
- **balanced**: for all nodes, there can be no more than a difference of 1 in the height of the left subtree from the right. Math.abs(height(left subtree) – height(right subtree)) ≤ 1

AVL stands for Adelson-Velsky and Landis (the inventors of the data structure)
Measuring Balance

Measuring balance:

For each node, compare the heights of its two sub trees

Balanced when the difference in height between sub trees is no greater than 1

Balanced

Unbalanced
Is this a valid AVL tree?

Is it...
- Binary: yes
- BST: yes
- Balanced?: yes
Is this a valid AVL tree?

Is it...
- Binary  yes
- BST      yes
- Balanced? no

Height = 0  Height = 2

2 Minutes
Is this a valid AVL tree?

Is it...
- Binary  yes
- BST  no
- Balanced? yes

-1  2  5
5

6

9  7

8

11

15

9 > 8
Implementing an AVL tree dictionary

Dictionary Operations:

get() – same as BST

containsKey() – same as BST

put() - Add the node to keep BST, fix AVL property if necessary

remove() - Replace the node to keep BST, fix AVL property if necessary

Unbalanced!
Rotations!

Insert ‘c’

Unbalanced!

Balanced!
Rotate Left

parent’s right becomes child’s left, child’s left becomes its parent

Insert ‘c’

Unbalanced!

Balanced!
Rotate Right
parent’s left becomes child’s right, child’s right becomes its parent

put(16);
Rotate Right
parent’s left becomes child’s right, child’s right becomes its parent

put(16);
So much can go wrong

Unbalanced!

1

2

3

Parent’s right becomes child’s left
Child’s left becomes its parent

Rotate Left

Unbalanced!

1

2

3

Parent’s left becomes child’s right
Child’s right becomes its parent

Rotate Right

Unbalanced!

1

2

2
Two AVL Cases

**Line Case**
Solve with 1 rotation

**Kink Case**
Solve with 2 rotations

- **Rotate Right**
  - Parent’s left becomes child’s right
  - Child’s right becomes its parent

- **Rotate Left**
  - Parent’s right becomes child’s left
  - Child’s left becomes its parent

- **Right Kink Resolution**
  - Rotate subtree left
  - Rotate root tree right

- **Left Kink Resolution**
  - Rotate subtree right
  - Rotate root tree left
Double Rotations 1

Insert ‘c’
Double Rotations 2

Unbalanced!