Section 03: Solutions

Review Problems

1. Code Analysis

For each of the following code blocks, what is the worst-case runtime? Give a big-Θ bound.

(a)
```
public IList<String> repeat(DoubleLinkedList<String> list, int n) {
    IList<String> result = new DoubleLinkedList<String>();
    for(String str : list) {
        for(int i = 0; i < n; i++) {
            result.add(str);
        }
    }
    return result;
}
```

Solution:

The runtime is $\Theta(nm)$, where m is the length of the input list and n is equal to the int `n` parameter.

One thing to note here is that unlike many of the methods we've analyzed before, we can't quite describe the runtime of this algorithm using just a single variable: we need two, one for each loop.

The other thing to remember is that in Java, foreach loops are converted into a while loop using iterators, which will influence the final runtime of our algorithm.

(b)
```
public void foo(int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 5; j < i; j++) {
            System.out.println("Hello!");
        }

        for (int j = i; j >= 0; j -= 2) {
            System.out.println("Hello!");
        }
    }
}
```

Solution:

The inner loop executes about $i - 5 + 1/2$ operations per loop. So we execute about

$$\sum_{i=0}^{n} i - 5 + i/2 = \frac{3}{2}\sum_{i=0}^{n} i - \sum_{i=0}^{n} 5 = \frac{3n(n-1)}{4} - 5n$$

$\Theta(n^2)$.

(c)
```java
public int num(int n){
    if (n < 10) {
        return n;
    } else if (n < 1000) {
        return num(n - 2);
    } else {
        return num(n / 2);
    }
}
```

Solution:

The answer is $\Theta\left(\log(n)\right)$.

One thing to note is that the second case effectively has no impact on the runtime. That second case occurs only for $n < 1000$ – when discussing asymptotic analysis, we only care what happens with the runtime as n grows large.

(d)
```java
public int foo(int n) {
    if (n <= 0) {
        return 3;
    }
    int x = foo(n - 1);
    System.out.println("hello");
    x += foo(n - 1);
    return x;
}
```

Solution:

The answer is $\Theta\left(2^n\right)$.

In order to determine that this is exponential, let's start by considering the following recurrence:

$$T(n) = \begin{cases} 1 & \text{If } n = 0 \\ 2T(n-1) + 1 & \text{Otherwise} \end{cases}$$

While we could unfold this to get an exact closed form, we can approximate the final asymptotic behavior by taking a step back and thinking on a higher level what this is doing.

Basically, what happens is we take the work done by $T(n-1)$ and multiply it by 2. If we ignore the $+1$ constant work done in the recursive case, the net effect is that we multiply 2 approximately n times. This simplifies to 2^n.

2. Binary Search Trees

(a) Write a method `validate` to validate a BST. Although the basic algorithm can be converted to any data structure and work in any format, if it helps, you may write this method for the `IntTree` class:

```
public class IntTree {
    private IntTreeNode overallRoot;

    // constructors and other methods omitted for clarity

    private class IntTreeNode {
        public int data;
        public IntTreeNode left;
        public IntTreeNode right;

        // constructors omitted for clarity
    }
}
```

Solution:

```
public boolean validate() {
    return validate(overallRoot, Integer.MIN_VALUE, Integer.MAX_VALUE);
}

private boolean validate(IntTreeNode root, int min, int max) {
    if (root == null) {
        return true;
    } else if (root.data > max || root.data < min) {
        return false;
    } else {
        return validate(root.left, min, root.data - 1) &&
            validate (root.right, root.data + 1, max);
    }
}
```

Section Problems

3. Recurrences

For each of the following recurrences, use the unfolding method to first convert the recurrence into a summation. Then, find a big-Θ bound on the function in terms of n. Assume all division operations are integer division.

(a) $T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n/2) + 3 & \text{otherwise} \end{cases}$

Solution:

Unfolding a few levels to find a pattern:

$$\begin{aligned} T(n) &= T(n/2) + 3 \\ &= T(n/4) + 3 + 3 \\ &= T(n/8) + 3 + 3 + 3 \\ &= T(n/2^i) + 3i \end{aligned}$$

Setting $i = \log n$ to force the input into a base case, we get: $1 + \sum_{i=2}^{\log(n)+1} 3$. The big-$\Theta$ bound is $\Theta(\log(n))$.

Something you may notice is that depending on what exactly n is, the expression $\log(n) + 1$ may not evaluate to an integer. In that case, what does it mean to have $\log(n)+1$ as the upper limit of a summation?

What exactly this mean differs based on convention, but for the purposes of this class, we'll assume that i varies starting at 2 up to the largest possible integer that is $\leq \log(n) + 1$. We could write this more explicitly using floors: $1 + \sum_{i=2}^{\lfloor \log(n)+1 \rfloor} 3$.

(b) $T(n) = \begin{cases} 1 & \text{if } n = 0 \\ T(n-1) + 2 & \text{otherwise} \end{cases}$

Solution:

The summation is $1 + \sum_{i=1}^{n} 2$. The big-Θ bound is $\Theta(n)$.

(c) $T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(n/2) + n & \text{otherwise} \end{cases}$

Solution:

Unfolding a few levels to find a pattern sounds good, but this one is easier to see if we strategically choose an n such that we reach the base case easily, like $n = 8$:

$$\begin{aligned}
T(8) &= 2T(8/2) + 8 \\
T(8/2) &= 2T(8/(2*2)) + 8/2 \\
T(8) &= 2(2T(8/(2^2)) + 8/2) + 8 \\
&= 2^2T(8/(2^2)) + 8 + 8 \\
&= 2^2T(8/(2^2)) + 2*8 \\
T(8/(2^2)) &= 2T(8/(2^2*2)) + 8/(2^2) \\
T(8) &= 2^2(2T(8/(2^2*2)) + 8/(2^2))) + 2*8 \\
&= 2^3T(8/(2^3)) + 8 + 2*8 \\
&= 2^3T(8/(2^3)) + 3*8 \\
T(8/(2^3)) &= 1
\end{aligned}$$

It's becoming clearer now that we can generalize $T(n)$ in terms of i, such that $T(n) = 2^iT(n/2^i) + i*n$. Setting $i = \log(n)$ to force the input into a base case, we get: $\sum_{i=0}^{\log(n)} n = nlog(n) + n$. The big-$\Theta$ bound is $\Theta(n\log(n))$.

(d) $T(n) = \begin{cases} 1 & \text{if } n = 0 \\ T(n/3) + 4 & \text{otherwise} \end{cases}$

Solution:

The summation is $1 + \sum_{i=1}^{\log_3(n)+1} 4$. The big-$\Theta$ bound is $\Theta(\log n)$.

(e) $T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(n-1) + 1 & \text{otherwise} \end{cases}$

Solution:

Using a similar process, we get the following expression: $2^{n-1} + \sum_{k=0}^{n-2} 2^k = 2^{n-1} + 2^{n-1} - 1$. Both of these terms are $\Theta(2^n)$ (because $2^{n-1} = \frac{1}{2}2^n = \Theta(2^n)$) This ends up being in $\Theta(2^n)$.

(f) $T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(n/2) + 100 & \text{otherwise} \end{cases}$

Solution:

We first get $T(n) = 2^i T(n/2^i) + (2^i - 1)100$, then the summation is $-100 + \sum_{i=0}^{n} 101$.

Therefore, we have $\Theta(n)$.

4. Modeling recursive functions

(a) Consider the following method.

```java
public static int f(int n) {
    if (n == 0) {
        return 0;
    }

    int result = 0;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < i; j++) {
            result += j;
        }
    }
    return 5 * f(n / 2) + 3 * result + 2 * f(n / 2);
}
```

(i) Find a recurrence $T(n)$ modeling the worst-case runtime of `f(n)`.

Solution:

$$T(n) = \begin{cases} 1 & \text{When } n = 0 \\ \frac{n(n-1)(n-2)}{6} + 2T(n/2) & \text{Otherwise} \end{cases}$$

(ii) Find a recurrence $W(n)$ modeling the *returned integer output* of `f(n)`.

Solution:

$$W(n) = \begin{cases} 0 & \text{When } n = 0 \\ \frac{n(n-1)(n-2)}{2} + 7T(n/2) & \text{Otherwise} \end{cases}$$

(b) Consider the following method.

```
public static int g(n) {
    if (n <= 1) {
        return 1000;
    }
    if (g(n / 3) > 5) {
        for (int i = 0; i < n; i++) {
            System.out.println("Hello");
        }
        return 5 * g(n / 3);
    } else {
        for (int i = 0; i < n * n; i++) {
            System.out.println("World");
        }
        return 4 * g(n / 3);
    }
}
```

(i) Find a recurrence $S(n)$ modeling the worst-case runtime of `g(n)`.

Solution:

$$S(n) = \begin{cases} 1 & \text{When } n \leq 1 \\ 2S(n/3) + n & \text{Otherwise} \end{cases}$$

Important: note that the if statement contains a recursive call that must be evaluated for $n > 1$.

(ii) Find a recurrence $X(n)$ modeling the *returned integer output* of `g(n)`.

Solution:

$$X(n) = \begin{cases} 1000 & \text{When } n \leq 1 \\ 5T(n/3) & \text{Otherwise} \end{cases}$$

(iii) Find a recurrence $P(n)$ modeling the *printed output* of `g(n)`.

Solution:

$$P(n) = 2P(n/3) + n$$

(c) Consider the following set of recursive methods.

```
public int test(int n) {
    IDictionary<Integer, Integer> dict = new AvlDictionary<>();
    populate(n, dict);
    int counter = 0;
    for (int i = 0; i < n; i++) {
        counter += dict.get(i);
    }
    return counter;
}

private void populate(int k, IDictionary<Integer, Integer> dict) {
    if (k == 0) {
        dict.put(0, k);
    } else {
        for (int i = 0; i < k; i++) {
            dict.put(i, i);
        }
        populate(k / 2, dict);
    }
}
```

(i) Write a mathematical function representing the *worst-case runtime* of `test`.

You should write two functions, one for the runtime of `test` and one for the runtime of `populate`.

Solution:

The runtime of the `populate` method is:

$$P(k) = \begin{cases} \log(N) & \text{When } k = 0 \\ k\log(N) + P(k/2) & \text{Otherwise} \end{cases}$$

Here, N is the maximum possible value of n.

The runtime of the `test` method is then $R(n) = P(n) + nlog(n)$.

(ii) Write a mathematical function $Y(n)$ representing the *returned integer output* of `test`.

Solution:

$$Y(n) = \frac{n(n-1)}{2}$$

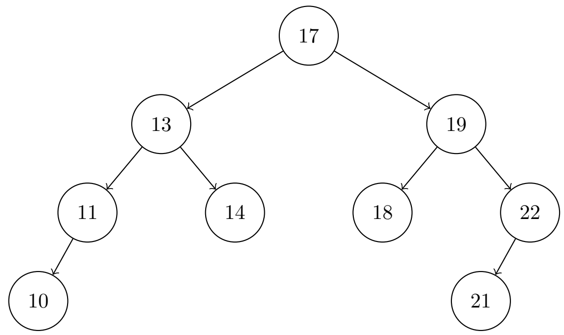
5. AVL Trees

(a) Draw an AVL Tree as each of the following keys are added in the order given. Show intermediate steps.

(i)

$$\{13, 17, 14, 19, 22, 18, 11, 10, 21\}$$

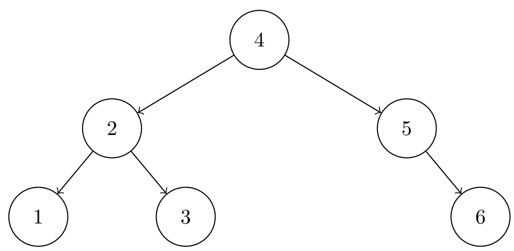
Solution:



(ii)

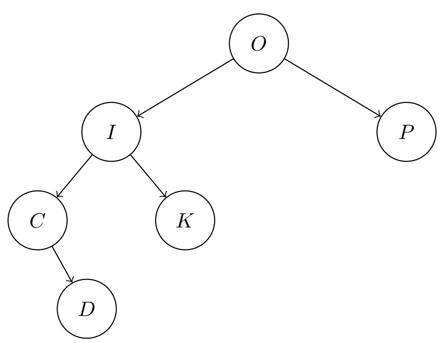
$$\{1, 2, 3, 4, 5, 6\}$$

Solution:



(b) Identify if the following trees are AVL trees. Explain your answer.

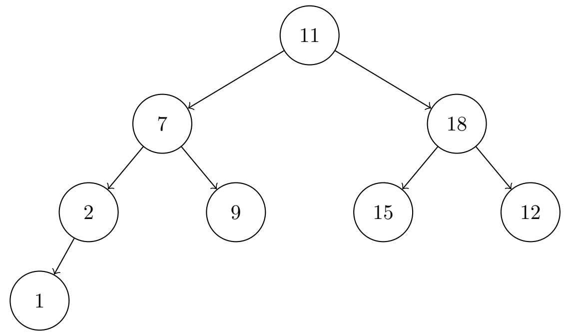
(i) Tree 1



Solution:

No, does not meet the balance property.

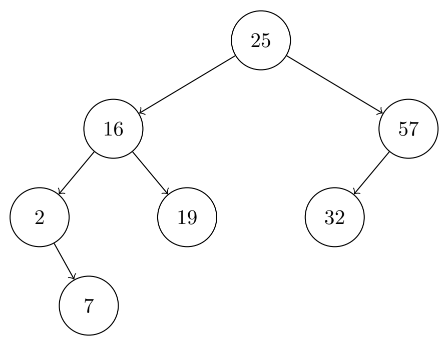
(ii) Tree 2



Solution:

No, does not meet the BST property. 12 is not greater than 18.

(iii) Tree 3



Solution:

Yes, it satisfies the balance and BST properties.

Food for thought

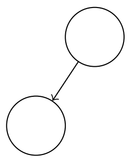
6. AVL trees

What is the minimum number of nodes in an AVL tree, given the following heights? Draw a picture of such a tree. (**Reminder**: an AVL tree's height is 0 for a tree with only 1 node in it!)

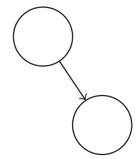
(a) 1

Solution:

Minimum number of nodes is 2, for a height of 1.



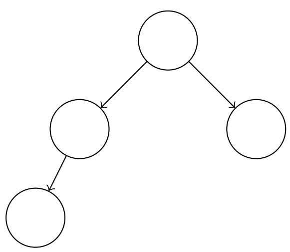
Solution:



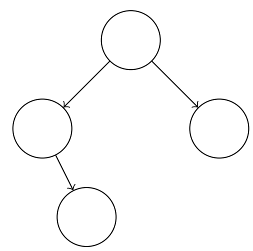
(b) 2

Solution:

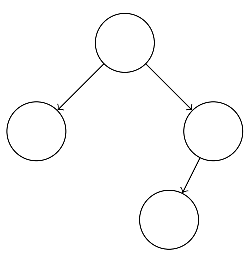
Minimum number of nodes is 4, for a height of 2.



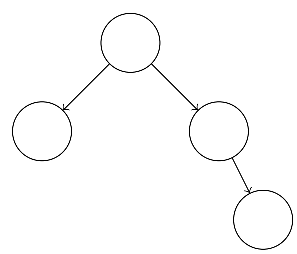
Solution:



Solution:



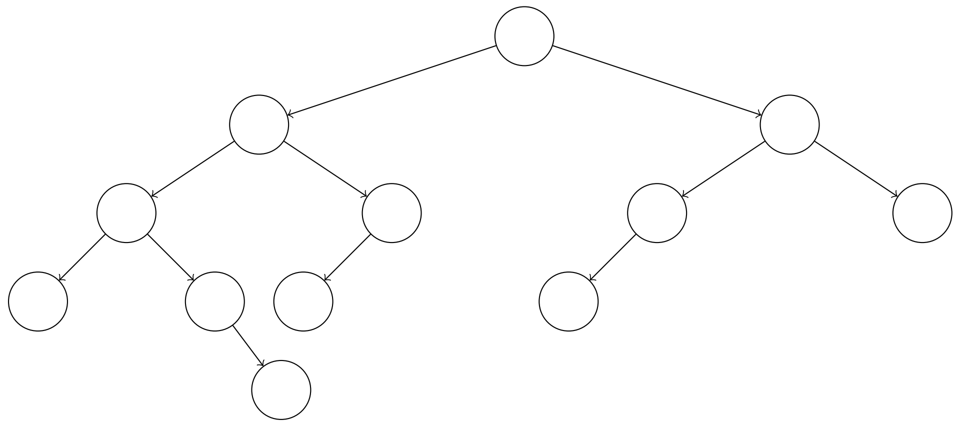
Solution:



(c) 4

Solution:

Minimum number of nodes is 12, for a height of 4. There are many possible solutions. Here is one!



7. Algorithm Design

(a) Given a binary search tree, describe how you could convert it into an AVL tree with worst-case time $\mathcal{O}(n \log(n))$. What is the best case runtime of your algorithm?

Solution:

Since we already have a BST, we can do an in-order traversal on the tree to get a sorted array of nodes. We could now simply insert all of these nodes back into an AVL tree using rotations which would give us an $\mathcal{O}(n \log(n))$ runtime.

(b) Given an AVL tree, describe how would you do a level order tree traversal. What is the worst-case runtime of your algorithm?

Solution:

Since an AVL tree is just a balanced BST, we can use a queue to add each node we visit. As we dequeue each node, we will add it's children to the queue. We would get an $\mathcal{O}(n)$ runtime.

Challenge Problems

8. Recurrences

(a) For the following recurrence, use the unfolding method to first convert the recurrence into a summation. Then, find a big- bound on the function in terms of n. Assume all division operations are integer division.

$$T(n) = \begin{cases} 1 & \text{if } n = 0 \\ 2T(n/3) + n & \text{otherwise} \end{cases}$$

Solution:

In order to determine what this expression looks like as a summation, it helps to first partially unroll it. When unfolding a recurrence like this it helps to

(i) Distribute the 2 (coefficient of the recursive call) at each step, to avoid having too many parentheses within parentheses.

(ii)

$$\begin{aligned} T(n) &= n + 2T\left(\frac{n}{3}\right) \\ &= n + 2\left(2T\left(\frac{n}{3^2}\right) + \frac{n}{3}\right) = n + \frac{2n}{3}2^2T\left(\frac{n}{3^2}\right) \\ &= n + \frac{2n}{3} + 2^2\left(2T\left(\frac{n}{3^3} + \frac{n}{3^2}\right)\right) = n + \frac{2n}{3} + \frac{2^2n}{3^2} + 2^3T\left(\frac{n}{3^3}\right) \\ &= n + \frac{2n}{3} + \frac{2^2n}{3^2} + 2^3\left(\frac{n}{3^3} + T\left(\frac{n}{3^4}\right)\right) = n + \frac{2n}{3} + \frac{2^2n}{3^2} + \frac{2^3n}{3^3} + 2^4T\left(\frac{n}{3^4}\right) \end{aligned}$$

We can start to see the pattern now: our summation is roughly of the form

$$n + \frac{2n}{3} + \frac{2^2n}{3^2} + \cdots + \frac{2^{i-1}n}{3^{i-1}} + 2^i\left(\frac{n}{3^i}\right)$$

Choosing $i = \log_3(n)$ buts us in the base case, and we get:

$$T(n) = n + \frac{2n}{3} + \frac{2^2n}{3^2} + \cdots + \frac{2^{\log_3(n)-1}n}{3^{\log_3(n)-1}} + 2^{\log_3(n)}$$

Notice that unlike with most of our previous recurrences, the term for the base case is not a constant. Because each of our recursive calls makes two other recursive calls, we hit the base case more than once (in fact $2^{\log_3(n)}$) times).

Rewriting to be in summation form:

$$\begin{aligned} T(n) &= \sum_{i=0}^{\log_3(n)-1} \frac{2^in}{3^i} + 2^{\log_3 n} \\ &= n\sum_{i=0}^{\log_3(n)-1} \left(\frac{2}{3}\right)^i + 2^{\log_3 n} \end{aligned}$$

Let's handle the summation first. We could evaluate the summation exactly, but that would be a bit tedious. Instead, let's show that it evaluates to some constant (which we can supress in the $\Theta()$ at the end). Each term is positive, and the first is $2/3$, so it is at least some constant. On the other hand, if instead of stopping when $i = \log_3(n) - 1$, we kept going to $i = \infty$, we would get an infinite geometric series, whose closed form is $\frac{1}{1-2/3} = 3$. Since this infinite series is larger than the finite one we care about, our series sums to at most 3. Thus it is between $2/3$ and 3, and definitely must be a constant.

Now let's figure out $2^{\log_3 n}$. Using logarithm identity 4 of the math review, we can change that number to $n^{\log_3(2)}$. So we have that

$$T(n) = \Theta(n) + n^{\log_3(2)}$$

.

Which of those is the dominating term? $\log_3(2)$ is the number which you raise 3 to to get 2. Since 3 is less than 2, $\log_3(2) < 1$. So the first term dominates and $\Theta(n)$ is our final answer.

9. Modeling recursive functions

Consider the following recursive function. You may assume that the input will be a multiple of 3.

```
public int test(int n) {
    if (n <= 6) {
        return 2;
    } else {
        int curr = 0;
        for (int i = 0; i < n * n; i++) {
            curr += 1;
        }
        return curr + test(n - 3);
    }
}
```

(a) Write a recurrence modeling the *worst-case runtime* of `test`.

Solution:

$$T(n) = \begin{cases} 1 & \text{When } n \leq 6 \\ n^2 + T(n-3) & \text{Otherwise} \end{cases}$$

(b) Unfold the recurrence into a summation (for $n \geq 6$).

Solution:

$$1 + \sum_{i=3}^{n/3} (3i)^2$$

Modeling this recurrence correctly is slightly challenging because we want to decrease n in increments of 3.

To do this, what we do is set the summation bounds to go up to $n/3$ instead of n, and multiply i on the inside by 3, simulating changing i in those increments.

We then also set the lower summation bound to be 3 instead of 0 or 1. That way, our summation will only consider numbers in the range 9 to n – if we set $i = 2$ or lower, our summation would double-cound $n \leq 6$, which should be handled by the base case.

Note: our model only works if n is a multiple of 3.

(c) Simplify the summation into a closed form (for $n \geq 6$).

Solution:

$$\begin{aligned}
1 + \sum_{i=3}^{n/3} (3i)^2 &= 1 + \sum_{i=0}^{n/3} (3i)^2 - \sum_{i=0}^{2} (3i)^2 && \text{Adjusting summation bounds} \\
&= 1 + 9 \sum_{i=0}^{n/3} i^2 - \sum_{i=0}^{2} (3i)^2 && \text{Pulling out a constant} \\
&= 1 + 9 \sum_{i=0}^{n/3} i^2 - (0 + 9 + 36) && \text{Evaluating the summation} \\
&= 9 \frac{\frac{n}{3}\left(\frac{n}{3}+1\right)\left(\frac{2n}{3}+1\right)}{6} - 44 && \text{Sum of squares}
\end{aligned}$$

A "closed form", within the context of this class, is just any expression that does not contain a summation or is recursive. This means we can stop here without needing to further simplify the expression.

That said, if you wanted to continue simplifying, we could:

$$\begin{aligned}
9 \frac{\frac{n}{3}\left(\frac{n}{3}+1\right)\left(\frac{2n}{3}+1\right)}{6} - 44 &= \frac{9}{6}\left(\frac{n}{3}\left(\frac{n}{3}+1\right)\left(\frac{2n}{3}+1\right)\right) - 44 \\
&= \frac{1}{2}\left(n\left(\frac{n}{3}+1\right)\left(\frac{2n}{3}+1\right)\right) - 44 \\
&= \frac{1}{2}\left(n\left(\frac{2}{9}n^2 + n + 1\right)\right) - 44 \\
&= \frac{1}{9}n^3 + \frac{1}{2}n^2 + \frac{1}{2}n - 44
\end{aligned}$$

10. AVL Trees

(a) Is there a relationship between an AVL tree's height, and its minimum or maximum number of nodes? If so, what is it?

Solution:

There is a relationship for the minimum number of nodes, and it's recursive:

$$S_{min}(h) = \begin{cases} 1 & \text{When } h = 0 \\ 2 & \text{When } h = 1 \\ 1 + S_{min}(h-2) + S_{min}(h-1) & \text{Otherwise} \end{cases}$$

The maximum number of nodes is a little more straightforward: level i of the tree can have up to 2^i nodes in it. Summing across all levels we get:

$$S_{max}(h) = 2^{h+1} - 1$$

(b) Write a method `isAVLTree` to check if a given tree (which is guaranteed to be a valid BST) is a valid AVL tree. If it helps, you may write this method for this tree class, `HeightTree`, which keeps track of the height of a tree at each node:

```
public class HeightTree {
    private IntHeightNode overallRoot;

    // constructors and other methods omitted for clarity

    private class IntHeightNode {
        public int data;
        public int height;
        public IntHeightNode left;
        public IntHeightNode right;

        // constructors omitted for clarity
    }
}
```

Solution:

```
public boolean isAVLTree() {
    return isAVLTree(overallRoot);
}

private boolean isAVLTree(IntHeightNode root) {
    if (root == null || (root.left == null && root.right == null)) {
        return true;
    } else if (Math.abs(root.left.height = root.right.height) > 1) {
        return false;
    } else {
        return isAVLTree(root.left) && isAVLTree(root.right);
    }
}
```

(c) Now write `isAVLTree` *without* assuming that the tree is a valid BST.

Solution:

TODO

(d) Now write `isAVLTree` for the `IntTree` class (you may assume again that the tree is guaranteed to be a valid BST).

Solution:

```java
public boolean isAVLTree() {
    return isAVLTree(overallRoot);
}

private boolean isAVLTree(IntTreeNode root) {
    if (root == null || (root.left == null && root.right == null)) {
        return true;
    } else {
        return Math.abs(height(root.left) - height(root.right)) <= 1 &&
            isAVLTree(root.left) && isAVLTree(root.right);
    }
}

private int height(IntTreeNode root) {
    if (root == null || (root.left == null && root.right == null)) {
        return 0;
    } else {
        return 1 + Math.max(height(root.left), height(root.right));
    }
}
```