# Section 03: Solutions

## Section Problems

### 1. 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);
    }
}
```

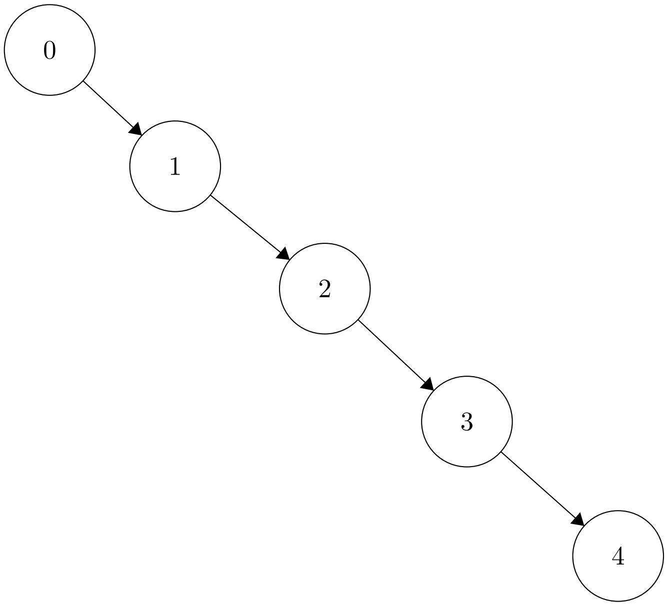
(b) Suppose we want to implement a method `findNode(V value)` that searches a binary search tree with $n$ nodes for a given value.

   (i) What is the worst case big-$\Theta$ runtime for `findNode`? Draw an example of a binary search tree with up to 4 nodes that would result in this worst-case runtime.

   **Solution:**

   The answer is $\Theta(n)$, because we could have a completely unbalanced tree and the value we are looking for could be at the very bottom.

   Example `findNode(4)`:

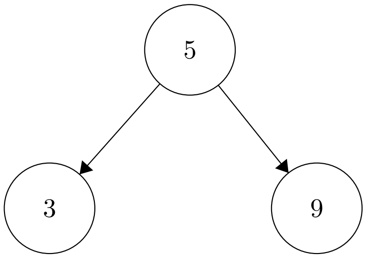


   (ii) What is the best case big-$\Theta$ runtime for `findNode`? Draw an example of a binary search tree with up to 4 nodes that would result in this best-case runtime.

   **Solution:**

   The answer is $\Theta(1)$, because the node we're searching for could be at the root.

   Example: `findNode(5)`:



## 2. Code Analysis

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

(a)

```
public List<String> repeat(List<String> list, int n) {
    List<String> result = new LinkedList<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.

(b)

```
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 $\Theta(\log(n))$.

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.

(c)

```
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 $\Theta(2^n)$.

If we visualized a tree that counted the number of calls to `foo`, we would see that exactly $2^{n+1} - 1$ calls are made. This is very similar to the example we saw in lecture 5 (https://courses.cs.washington.edu/courses/cse373/19au/lectures/05/)

(d)
```
public boolean isPrime(int n) {
    int toTest = 2;
    while (toTest < n) {
        if (n % toTest == 0) {
            return false;
        } else {
            toTest++;
        }
     }
     return true;
}
```

**Solution:**

There is no big-$\Theta$ bound for this function. Note that as $n$ grows very large, this function will occasionally run a single iteration. Remember that we only care about the worst case **as n grows larger and larger**.

Another way to think about this: if we thought about this as a graph, with $n$ on the x axis and the total number of operations on the y axis, we would see that as $n$ grows larger the y-axis value would be fluctuating. If we tried to come up with a function that lower-bounded this function as $n$ goes to infinity, the only lower bound we would be able to find would be $\Omega(1)$. Similarly, the tightest upper bound we could find would be $\mathcal{O}(n)$. Because the big-$\Omega$ and big-$O$ bounds do not match up in the worst case, there is no big-$\Theta$ bound.

## 3. “Tree method” walk-through

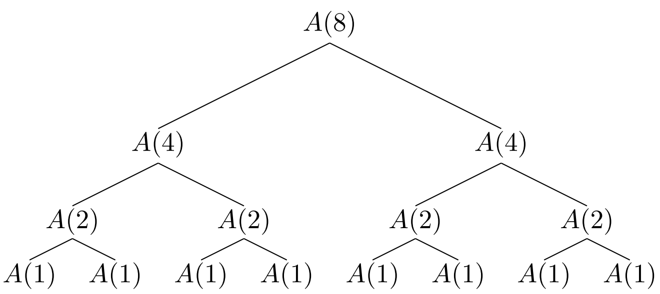
Consider the following method:

```
public int A(int n) {
  if (n <= 1) {
    System.out.println(”done!”);
    return 10;
  }
  for (int i = 0; i < n; i++) {
    System.out.println(”not done yet...”);
  }
  return A(n/2) + A(n/2);
}
```

We want to find an *exact* closed form of this method by using the tree method. Suppose we draw out the total work done by this method as a tree, as discussed in lecture. Let $n$ be the initial input to $A$.

(a) Draw a tree representing **the total number of calls** to $A$ for $n = 8$.

**Solution:**



(b) Consider the tree that would be generated representing $A$ called with any arbitrary $n$.

(i) What is the total amount of non-recursive work done at each **level** of the tree? Refer to the example you drew in the previous question to help you. Give your answer as an expression in terms of $n$.

**Solution:**

The answer is $n$ work for each level of this tree. Note that the root level does $n$ work because of the for loop. When we split this node into two children on the next level, we see that each of these children nodes will perform $\frac{n}{2}$ work. Since there are two child nodes, we get $\frac{n}{2} + \frac{n}{2} = n$ work on the second level. We can extend this idea to see that every level performs $n$ work.

(ii) What is the height of the tree? Give your answer as an expression in terms of $n$.

**Solution:**

The answer is $\log_2(n)$ because we continually divide n by 2 until we hit the base case of 1.

Another way to think about this would be how many times do I have to divide n by two to get 1? If we express that as a mathematical formula we could say $\frac{n}{2^i} = 1$ where $i$ is the number of times we need to divide $n$ by two to reach one. If we solve for this equation we see it's exactly when $i = \log_2(n)$.

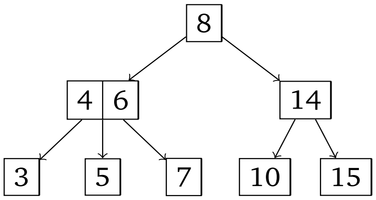
(iii) Give a big-$\Theta$ runtime for $A$ given an arbitrary $n$. Hint: use your answers from the previous parts to help you solve this question.

**Solution:**

The answer is $\Theta(n \log(n))$. This is because we have $\log(n)$ levels and at each level we do exactly $n$ work.

## 4. B-Trees

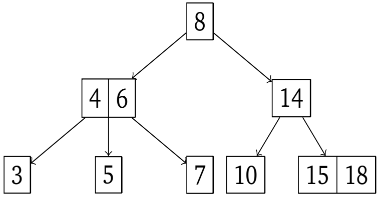
(a) Draw what the following 2-3 tree would look like after inserting 18, 38, 12, 13, and 20.



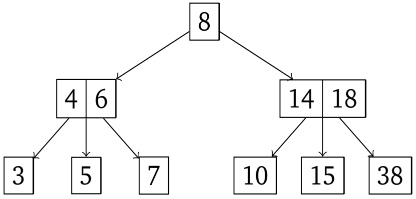
**Solution:**

The tree is redrawn after each insertion, but only the final tree matters in this case. Remember that doing something like this to show your work can help you earn partial credit in an exam setting!

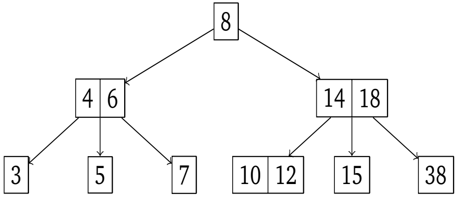
After inserting 18...



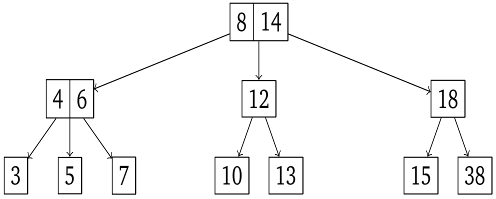
After inserting 38...



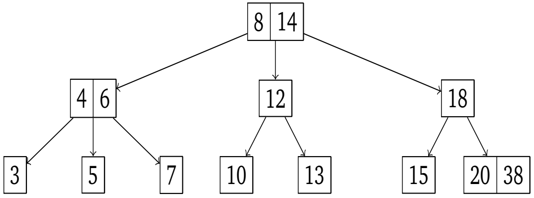
After inserting 12...



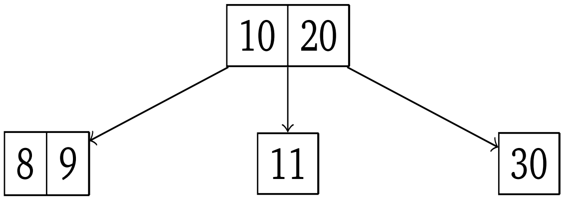
After inserting 13...



After inserting 20...

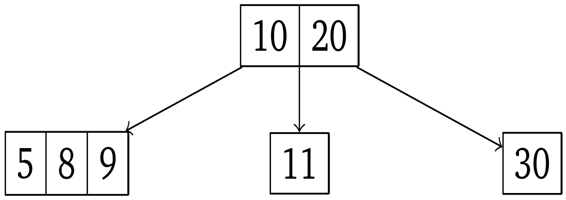


(b) Given the following initial 2-3-4 tree, draw the result of performing each operation.



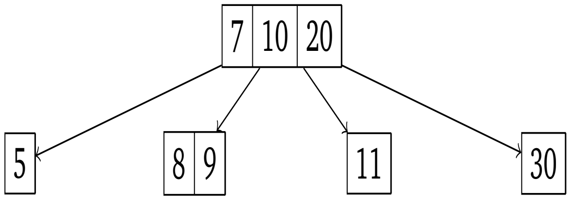
(i) Insert 5 into this tree.

**Solution:**



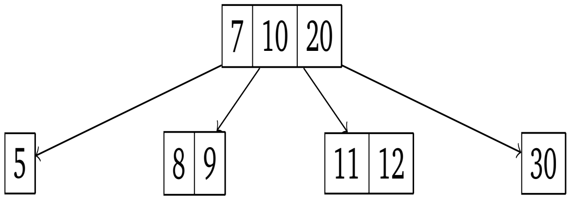
(ii) Insert 7 into the resulting tree.

**Solution:**



(iii) Insert 12 into the resulting tree.

**Solution:**



(c) Suppose the keys 1, 2, 3, 4, 5, 6, 7, 8, 9, and 10 are inserted sequentially into an initially empty 2-3-4 tree. Which insertions cause a split to take place?

**Solution:**

4, 6, 8, 10