CSM 61B
Spring 2019

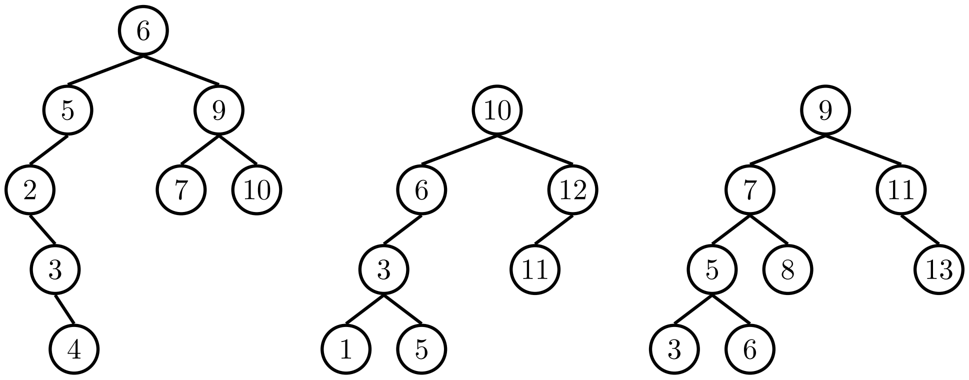
Trees & Hash

Mentoring 7: March 11, 2019

1 Binary Search Trees

```
public class BinarySearchTree<T extends Comparable<T>> {
    protected Node root;
    protected class Node {
        public T value;
        public Node left;
        public Node right;
    }
}
```

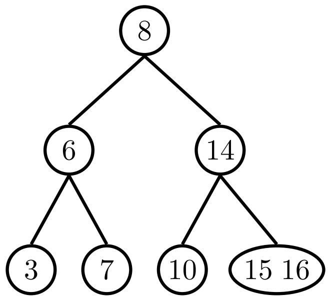
1.1 For each of the following binary search trees, determine if the height of the tree is the same as the height of the optimal tree with the same elements.



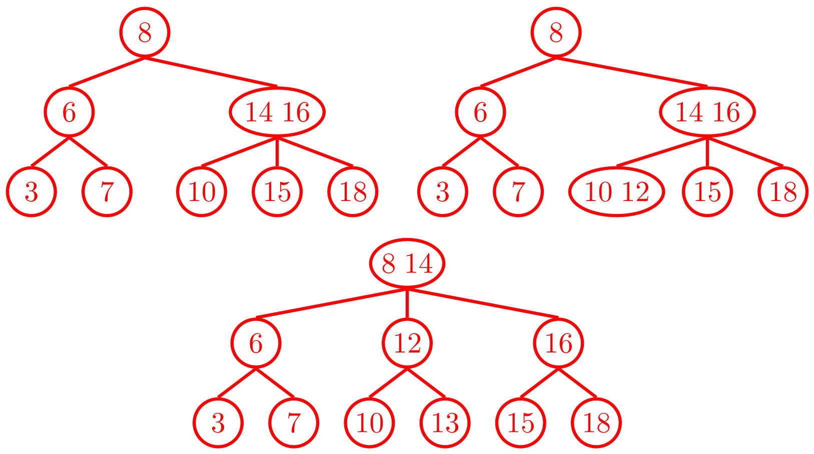
1. Not balanced
2. Not balanced
3. Balanced

Intuitively, if we can "stuff" all the elements in the lowest level into a higher level, then we can reduce the height of the tree and determine that the height of the tree is not the same as the height of the optimal binary tree.

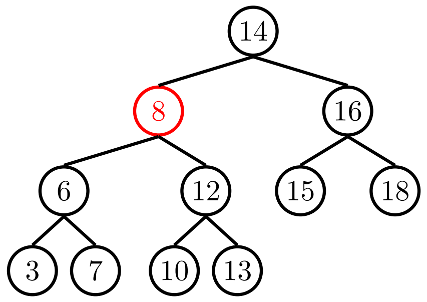
2 Balanced Trees



2.1 Draw what the 2-3 tree would look like after inserting 18, 12, and 13.



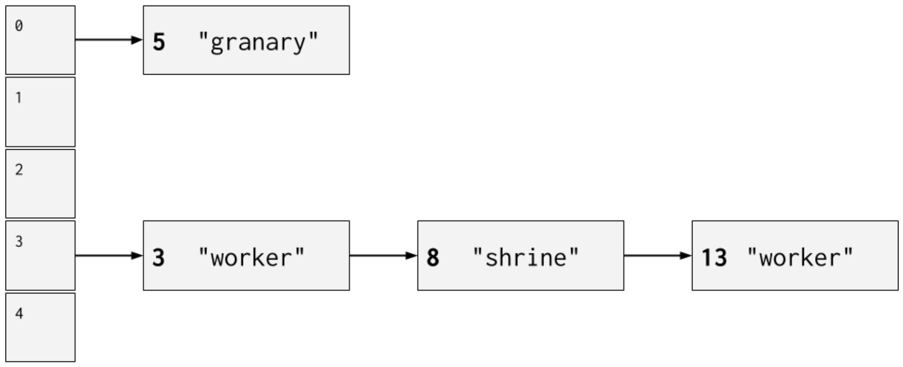
2.2 Now, convert the resulting 2-3 tree to a left-leaning red-black tree.



3 Hashing

3.1 (a) Draw the diagram that results from the following operations on a Java `HashMap`. `Integer::hashCode` returns the integer's value.

```
put(3, "monument");
put(8, "shrine");
put(3, "worker");
put(5, "granary");
put(13, "worker");
```



`"worker"` replaces `"monument"` as their keys are the same. Each `put` must iterate through the entire external chain to ensure that a key-update is not necessary.

(b) Suppose a resize occurs, doubling the array to size 10. What changes?

The value of `"shrine"` and `"granary"` will move. Specifically, the new length of the array is 10. A key of 8 will force `"shrine"` to be placed in the 8th index. A key of 5 will move `"granary"` to the 5th index. Everything else will remain the same.

4 Hash Codes

4.1 What does it mean for a hashcode to be valid?

- Consistency: Whenever it is invoked on the same object more than once, the `hashCode` method must consistently returns the same integer if nothing about the object changes.
- Equality constraint: If two objects are equal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects must produce the same integer result.
- It is not required that if two objects are unequal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.

4.2 Which of the following hashcodes are valid? Good?

```
class Point {
    private int x, y;
    private static int count = 0;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
        count += 1;
    }
}
```

(a)

```
public void hashCode() {
    System.out.print(this.x + this.y);
}
```

Invalid. Return type should be **`int`**. The method should override Java's `hashCode` method, but since it has the same signature with different return types, this would be a compile-time error

(b)

```
public int hashCode() {
    Random random = new Random();
    return random.nextInt();
}
```

Invalid. Not consistent. This method would compile because it does return an **`int`**. However, this does not obey the principle of consistency about returning the same integer every time, and so the `HashMap` would not work as expected. If we store an object at key 1, the next time around the object might have key 2, and we would be unable to find the object.

(c)

```
public int hashCode() {
    return this.x + this.y;
}
```

Valid, but certain inputs may cause a significant number of collisions. This method would work, since it obeys all bullets under the contract. However, this will cause many collisions. All points that sum up to the same number will be mapped to the same `hashCode` such as $(5,3),(4,4),(2,6),(6,2),(1,7),(7,1),\dots$

(d)

```
public int hashCode() {
    return count;
}
```

Invalid. Not consistent. This method would not error, because it does return an **`int`**, but it does not obey the principle of consistency. `count` is a static variable. Consider what happens when we create a **`new`** `Point (3, 4)` and call the `hashCode` method on it the first time. We would get 1. If we then create a **`new`** `Point(1, 2)`, calling the `hashCode` method on the earlier `Point(3, 4)` would now return 2, even though nothing about the original `Point` object changed.

(e)

```
public int hashCode() {
    return 4;
}
```

Valid, but causes collisions on all inputs. While this method is valid, it collides on any input.

5 Extra Practice: Trees

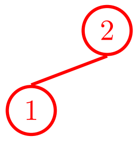
5.1 Given a node in a binary search tree (with parent pointers), implement `successor` which returns the next node in the in-order traversal of the BST. If there is no successor, return **`null`**.

```java
public class BinarySearchTree<T extends Comparable<T>> {
    protected Node root;
    protected class Node {
        public T value;
        public Node parent, left, right;
    }
    private Node successor(Node node) {
        if (node.right != null) {
            node = node.right;
            while (node.left != null) {
                node = node.left;
            }
            return node;
        } else {
            Node parent = node.parent;
            while (parent != null && parent.right == node) {
                node = parent;
                parent = parent.parent;
            }
            return parent;
        }
    }
}
```

There are two cases to find the next node in the in-order traversal. Let's think about the case specifically for Binary Search Trees. The next node in the in-order traversal is the same as the next biggest element in the Binary Search Tree.

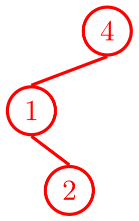
The first case is when the node passed in has a right branch. If it has a right branch, then the next largest element must lie in the sub-tree whose root is the right-child of the given node. Let's call this sub-tree T. The final thing to realize is that the element we are looking for is the smallest element in this sub-tree T. This is why we traverse left-wards, until we reach the left-bottom of T, which is the node we need to return.

The second case is when the node passed in has no right branch. To reason about this case, first consider a simple case with just 2 nodes.



Say that the node passed in is Node 1. Since the node doesn't have a right child, the next-largest node must be somewhere above it. In this example, we see that the answer is Node 2. As we traverse up from Node 1 to Node 2, we realize that the edge between 1 and 2 is a left-edge (1 is a left-child of 2). The general argument here is: keep traversing up the parents. As soon as we traverse a left-edge, we can stop and return the parent node (in this case, Node 2). Let's traverse up the parents, and name the first left-edge we encounter E, the parent on this edge P, and the child C. Why can we be sure that P is the node to return?

The first thing to note is that as we traverse up the tree, if we don't use a left-edge, then the child is greater than the parent. This is not what we want, since we want the next largest node. The next thing to notice is that going up a left-edge means that the parent is necessarily greater, by the definition of a left-edge in a Binary Search Tree. All that's left to argue now is that if node N is passed in, then $P > N$, and that there is no other node that is smaller than P, but greater than N. Consider the following example:



If the node that we're passed in is Node 2, then the first left-edge we will traverse while going up would have 4 as the parent. This example helps us see that the left-edge's parent will be greater, i.e., $P > N$, since it is guaranteed that N lies in the left-subtree of P. Note that we don't need to consider any of P's ancestors. If P has a parent through a right-edge, then that parent is bigger than P, which is not what we want. If P has a parent through a left-edge, then that parent is smaller than P, but it is also smaller than N, since N would lie in the left-subtree of P's parent.

The final argument is that there is no other node that is smaller than P, but greater than N. Let's look back at the example. First, we look at Node 1. Node 1 is smaller than Node 2, since it is a right-edge. In the example, Node 1 does not have any left-children, but if it did, we can be sure that they would all, too, be smaller than Node 2. So, any time we traverse a right-edge, we can ignore not only the right-edge's parent, but also be sure that everything in the left subtree of the right-edge can be safely ignored.

This concludes the argument, since we have shown that right-edges or their subtrees do not contain any nodes that are smaller than P but larger than N.