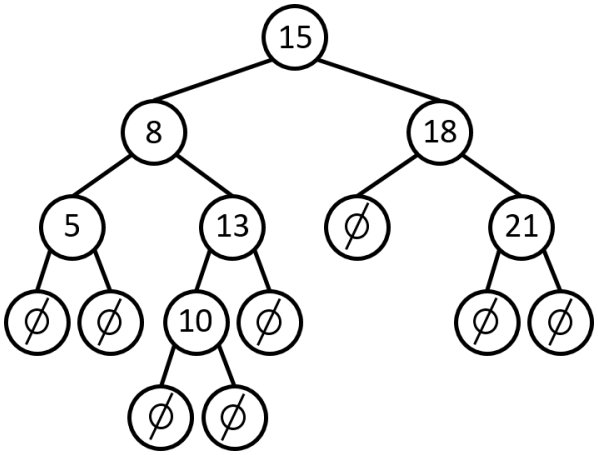
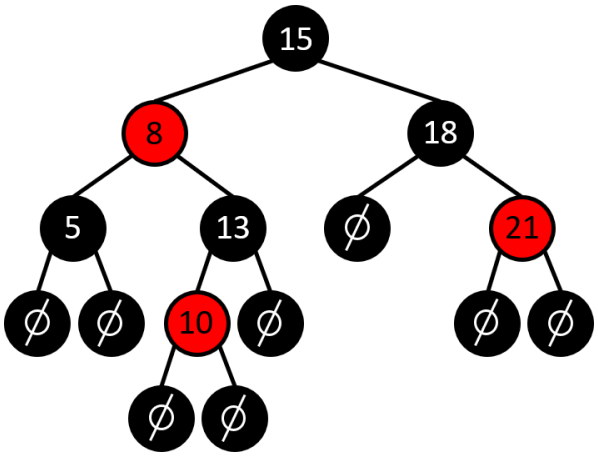
Data Structures
Spring 2021
Midterm 2 — Solutions

1 Red-Black Trees

Question: (a) [10 pts] Can the following binary search tree be colored to make it a red-black tree? If so, shade in the black nodes to make this a valid red-black tree. If not, explain what is wrong with the tree structurally.

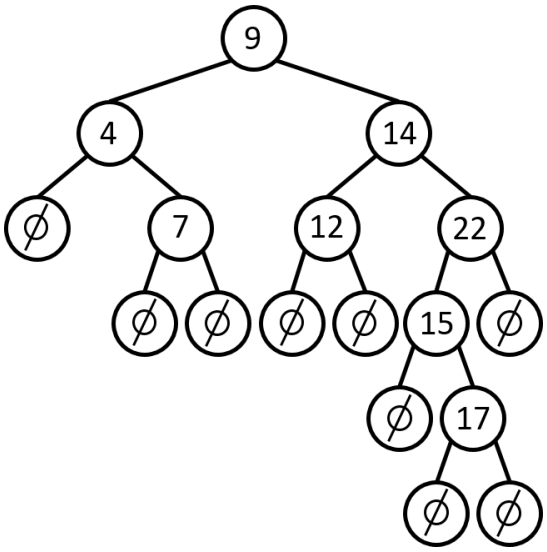


Answer: This tree can be colored as a red-black tree, as follows:



Discussion: Most people did fine on this one. A few students colored 18 red and 21 black; this will not work because then the path 15-18-null would have only 2 black nodes, rather than 3.

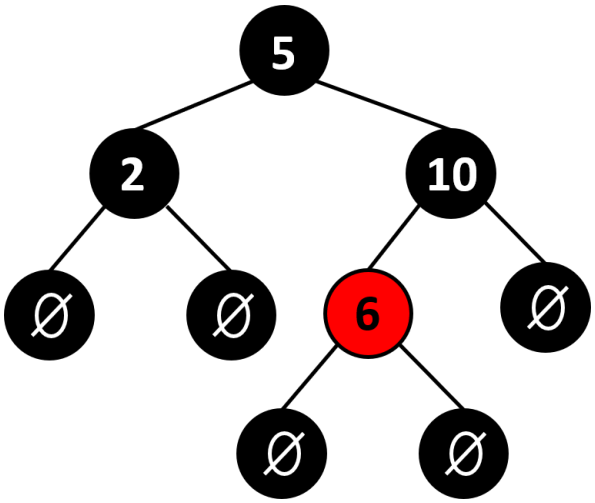
QUESTION: (b) [10 pts] Can the following binary search tree be colored to make it a red-black tree? If so, shade in the black nodes to make this a valid red-black tree. If not, explain what is wrong with the tree structurally.



ANSWER: This one can't be colored as a red-black tree. One problem is that the subtree rooted at 22 is too unbalanced. 22's right child is a null leaf, so 22 can have a black heght of at most 2 (if we color both 22 and its right child black). If we do this, we'd have to color both 15 and 17 red so that the path 22-15-17-null also only has 2 black nodes; this isn't allowed because a red node (15) cannot have a red child (17). Many other explanations are possible; there are several unbalanced subtrees within this tree.

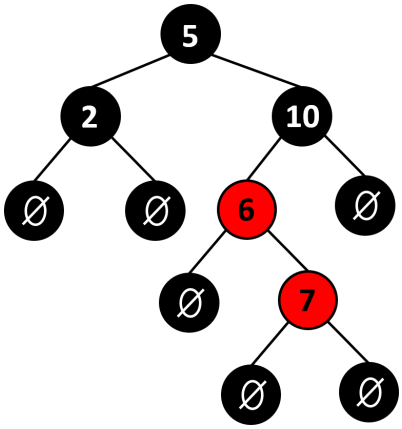
DISCUSSION: Everyone correctly identified that this tree cannot be colored as a red-black tree. The most common problem was not giving enough detail in the explanation; we asked you to explain what is wrong with the tree structurally. This meant that you either had to explain what goes wrong when you try to color the tree, or point out that the shortest path from the root to a null leaf is less than half the length of the longest path from the root to a null leaf.

QUESTION: (c) [10 pts] Here is a red-black tree:

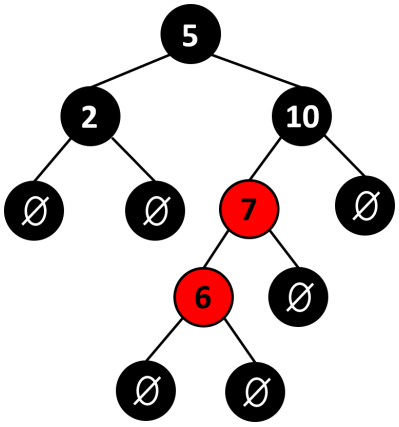


(In case you've printed in black and white and it isn't clear: 6 is the only node colored red in this tree.)

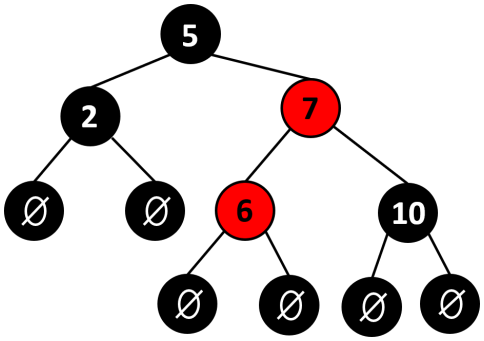
Show what the tree looks like after adding 7. Be sure to clearly indicate the color of each node in your final tree.
ANSWER: We'll show what the tree looks like after each intermediate step of the add. First, we do a normal BST add and color the new node, 7, red:



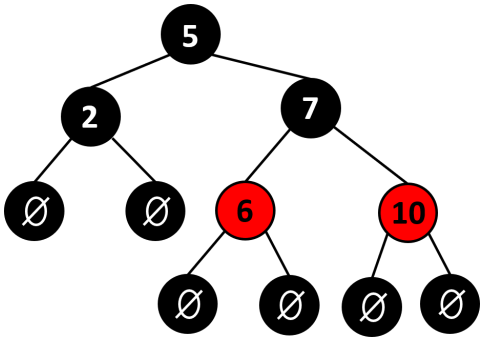
We observe that the new node's parent, 6, is red, and the parent's sibling, 10's null-leaf right child, is black. So we follow the fourth case of RBT-repair, in which we need to do some rotations to fix the structure of the tree. We begin with a left-rotation on 6:



Followed by a right-rotation on 10:



And finally, we finish it off by recoloring as follows:



DISCUSSION: Most people did fine with this. A few students rotated incorrectly, ending up with a tree that no longer satisfied the regular BST properties. The most common mistake was, at the end, coloring 7 red and 6 and 10 black. While this recoloring would give us a valid RBT in this case, it's not the correct recoloring to use in general because there is no guarantee that the parent of this subtree is black. That is, coloring 7 red after the rotations could have created a situation where we have a red node (7) with a red parent, which is not allowed.

2 Hash Tables

Consider the following hash table that uses *open addressing*:

index	0	1	2	3	4	5	6	7	8	9
key	50		72				26	107	66	

QUESTION: (a) [10 pts] If this hash table uses the hash function $h(k) = k\%m$, (where $m = 10$ for this array), and employs *linear probing* upon collision, **show how it will place a new key** when `add(46)` is performed. That is, indicate which positions are visited, in what order, and then show where the new key is placed.

ANSWER: Three collisions occur before finally placing the 46. The attempts occur as follows:

1. $h(46) = 46\%10 = 6$. The 26 stored in position 6 is a collision.
2. Linear probing dictates that we next attempt position 7, at which 107 causes a second collision.
3. We next attempt position 8, where a collision with 66 causes the third collision.
4. Finally, position 9 is unoccupied, and 46 is placed there.

DISCUSSION: Nearly everyone got this one. Not much to say.

QUESTION: (b) [10 pts] If this hash table employs *double hashing*, using the hash function $h(k) = (h_a(k) + ih_b(k))\%m$, where i is the iteration number (that is, the number of slots in which you've already attempted to place this key), $h_a(k) = k$, and $h_b(k) = 7 - (k\%7)$, **show how it will place a new key** when `add(46)` is performed.

ANSWER: There will be one collision before placing the 46:

1. On iteration 0, $h_a(46) = (6 + 0 \times h_b(46))\%10 = 6$, where the 26 will cause a collision.
2. On iteration 1, $h_a(46) = (6 + 1 \times (7 - (46\%7)))\%10 = (6 + 3)\%10 = 9$, which is unoccupied, and 46 is placed there.

DISCUSSION: A number of people considered the intitial attempt to be iteration $i = 1$, when it needs to be $i = 0$. Others simply miscalculated $h_b(k)$. Finally, a few people took the result of $h(k)$ at iteration $i = 1$ as an **addative** value for the initial index of 6 from the 0^{th} iteration; each $h(k)$ result, at each iteration, is its own index that does not need to be combined with previous results.

3 BST Size

QUESTION: [25 pts] Suppose you have a class `Node` that stores pointers to three `Node` objects `parent`, `left`, and `right`, and you have a class `BinarySearchTree` that stores a pointer to a `Node root`. Write a method in `BinarySearchTree` called `size` that takes a `Node` as input and returns an `int`. When called with the root as input, your `size` method should return the total number of nodes in the tree. Your method should be recursive.

ANSWER: Here's a method that will accomplish this:

```
public int size(Node current) {
    if (current == null) return 0;

    return 1 + size(current.left) + size(current.right);
}
```

DISCUSSION: Most people did quite well with this. Errors fell into a few broad categories. First, some of you had a base case that returns 1, rather than 0, if the input node is null. This doesn't quite work; it counts an entire level of nodes that does not exist below the leaves. Second, some of you included an extra input parameter that was meant to count how many nodes have been encountered so far. This extra parameter wasn't allowed (the question specifies that the method should take a Node as input) and was often used incorrectly. Third, some of you created a global variable to keep track of the size, and updated this size counter while doing a traversal of the three. This isn't technically incorrect, but it circumvents the need to actually do the computation recursively, which was the point of the question.

4 Joining BSTs

As in question 3, suppose you have a class `Node` that stores pointers to three `Node` objects `parent`, `left`, and `right`, and also stores a key `K key` and a value `V value`. You also have a class `BinarySearchTree` that stores a pointer to a `Node root`. The `BinarySearchTree` class implements the `Dictionay` interface (i.e., it includes the methods `V add(K key)`, `V remove(K key)`, and `V lookup(K key)`.)

Suppose we want to *join* two BSTs. Specifically, suppose we start with two BSTs, `T1` and `T2`, where we know that all of the keys in `T1` are smaller than all of the keys in `T2`. We also know that `T1` contains n nodes, `T2` contains m nodes, and $n < m$.

QUESTION: (a) [15 pts] Add a method called `join` to the `BST` class that, when given a `BST` as an input parameter, returns a new `BST` that is the result of joining `T1` and `T2`. That is, your method should have the header:

```
public BST join(BST T2)
```

For example, if `T1` contains the keys 1,3,6 and `T2` contains the keys 8,9,13,14, then the call `T1.join(T2)` should return a single tree containing keys 1,3,6,8,9,13,14.

Your goal should be to write a `join` method that is as efficient as possible. That is, it should have the smallest big-O runtime in n and m that you can come up with. You will get partial credit if your solution is correct, but does not have the lowest possible big-O runtime. You will also get partial credit if you give a clear explanation of what your method would do, but do not provide the actual code.

A GOOD ANSWER:

```
public BST join (BST T2) {

    BST T1 = this;
    Node rightmost = T1.root;
    while (current.right != null) {
        rightmost = rightmost.right;
    }

    rightmost.right = T2.root;
    T2.root.parent = rightmost;
    return T1;

}
```

A BETTER ANSWER:

```
public BST join (BST T2) {

    BST T1 = this;
    Node rightmost = T1.root;
    while (current.right != null) {
        rightmost = rightmost.right;
    }

    rightmost.parent.right = rightmost.left;
    if (rightmost.left != null) {
        rightmost.left.parent = rightmost.parent;
    }

    Node newroot = rightmost;
    newroot.parent = null;
    newroot.left = T1.root;
    newroot.right = T2.root;

    return new BST(newroot);

}
```

DISCUSSION: Many people got the good answer; a few managed the even better answer of promoting the largest key in T1 to the root of the new tree, then making what's left of T1 its left child, and all of T2 its right child. Common mistakes with these approaches included a failure to update `parent` pointers, and a failure to return some `BST` pointer at the end.

The most common correct-but-not-wonderful answers involved traversing a tree (or both trees) to remove and insert the keys into the other tree (or a new tree). Similarly, some traversed the trees to extract the keys into an array, which was then sometimes sorted, and then construct a new tree out of it. A number of you performed the traversal by creating an managing their own stack. These approaches are slow and needlessly complex; notably, they fail to take advantage of the conspicuous assumption that all of T1's keys are less than all of T2's keys. It is during these overly involved answers that all kinds of detailed mistakes were made in extracting the keys, managing the arrays, and reconstructing the new trees.

QUESTION: (b) [10 pts] In terms of n and m, what is the average-case big-O runtime of your `join` method?

ANSWER: For the both versions, assuming reasonable balance $O(lg\ n)$ would be required to traverse the maximum key in `T1`. What remains is $O(1)$ for pointer updates.

DISCUSSION: The correctness of the answers to this question depended heavily on the answers to part (a). For those who traversed one or both tree, a correct accounting of the $O(n + m)$ nodes was critical. It was also important to observe that inserting keys into some final tree had to account for the $n + m$ total nodes in that tree at the end, and the average case $O(n + m)$ of each `add` operation. Likewise, for those who removed nodes from one tree while putting them into the other, you had to account for the lg time to perform each removal.