Homework 9 - CS60 - Fall 2015
Due: Tuesday, November 10 at 11:59pm

Homework Overview

The next few problems in this assignment ask you to implement functions that manipulate *binary search trees* (BSTs) in a variety of ways. Recall, all leaves within a left-hand subtree are strictly less than the root. All elements within a right-hand subtree are strictly greater than the root. Finally, we never have duplicates in a BST. If you're feeling uncertain about BSTs, you might watch some videos Colleen's YouTube channel :-)

- Video: Tree vocab, BST introduction and representations
- Video: Binary Search Trees (BSTs) - Insert and Remove Explained

We hope that it is illuminating to see BSTs represented in both Racket and Java! We know it is REALLY REALLY hard to switch back and forth between the two languages, but we hope getting to see two perspectives on the same problem helps you understand the core ideas (e.g. the BSTs) better and that the code is just one way of representing these.

Feel free to post questions to Piazza if you get stuck on anything! We're happy to help then and during our office hours: Mondays and Wednesday 4-5pm in the lecture hall and Tuesdays 11-12 and 4-5pm in the LAC computer lab.

- Colleen & Prof. Beth

Problem 1: BSTs in Racket

- Learning Goal: Implement BST functions in Racket
- Prerequisites: Racket basics and
- Submission: BST.rkt
- Points: 40

Part A: Download and read the starter file

Although often binary search trees (BSTs) keep track of a key and a value (i.e. implement a dictionary), we'll use BSTs of only integers for the following Racket problems (i.e., implementing a set).

In Racket, we don't have the ability to define our own data structures like we do in Java. To compensate for the lack of classes, we've defined a way we'll represent trees (i.e. a list of the key, left subtree, and right subtree). However, we don't want to have to reason about the order of these elements when we write code, we just want to be able to think about the key, the left subtree and the right subtree. In the starter file we provided methods for constructing BSTs, accessing elements of a BST, and a few boolean functions for BSTs.

- Create BSTs
  - `(make-BST key left right)`
  - `(make-empty-BST)`
  - `(make-BST-leaf key)`
- Access elements of a BST
  - `(key  tree)`
  - `(leftTree tree)`
  - `(rightTree tree)`
- Boolean functions for a BST
  - `(emptyTree? tree)`
  - `(leaf? tree)`

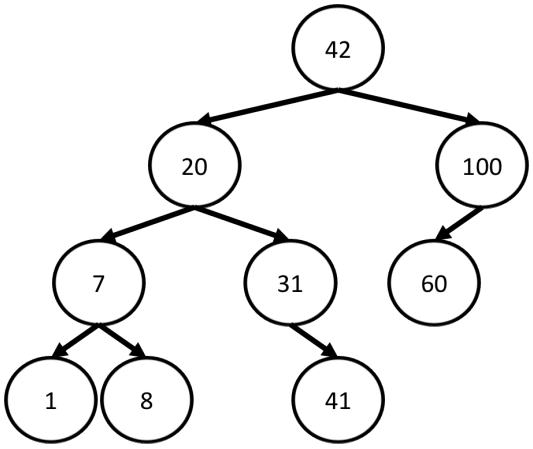
These functions will make the code more readable! For example, in one of the parts, you'll be creating a list -- If we used the list functions to access the contents of our BSTs, it wouldn't be clear if a particular call to `first` or `cons` was working with a list or a BST. These functions would also allow us to change the implementation of tree (i.e. change the order of the elements) and could make large changes a lot easier (e.g. imagine we wanted to add a value for each node). **We will take off up to 3 points per Part for not using the provided helper functions!**

- Download the starter code: BST.rkt

Part B: Practice using the constructors by writing test trees (5 points)

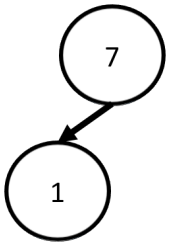
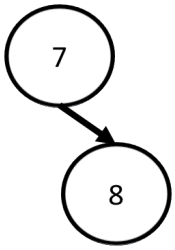
You're going to need to write a bunch of test cases in this assignment! Each one of these test cases is going to need a tree to work with. It'll be easier to write these test cases later on if you've created some trees and given them names so that you can refer to them in your test cases.

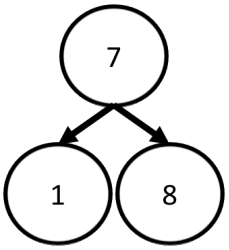
We've created (i.e. defined) a tree named `BigBST` that you can use in your test cases. It also appears in the starter code.




```
(define BigBST
   (make-BST 42
          (make-BST 20
                 (make-BST 7
                        (make-BST-leaf 1)
                        (make-BST-leaf 8))
                 (make-BST 31
                        (make-empty-BST)
                        (make-BST-leaf 41)))
          (make-BST 100
                 (make-BST-leaf 60)
                 (make-empty-BST))))
```

For Part B, you need to create the following four trees that you can use in your testing! The numbers can be anything you want as long as they're BSTs.

	`(define tree-1-node`
	`(define node-wLeftChild`
	`(define node-wRightChild`

	`(define node-wTwoChildren`

Part C: Write height (5 points)

In this part, you'll write a function `(height BST)` whose input is a binary search tree and whose output is the number of edges in the longest path from the root of `BST` to any one of its bottom-level nodes, i.e., the height of the binary search tree. Note that in this case we are defining the height of the empty binary search tree as -1. For instance,

```
(check-expect (height BigBST) 3) ;; using the tree defined above
(check-expect (height (make-empty-BST)) -1)
(check-expect (height (make-BST-leaf 42)) 0)
```

- Write test cases that to check the height of the trees you created (i.e. `tree-1-node`, `node-wLeftChild`, `node-wRightChild`, and `node-wTwoChildren`).
- Write code for the function `height`.

Part D: Write find-min (5 points)

In this part, you'll write a function `(find-min BT)` whose input will always be a *non-empty* binary search tree and whose output is the value of the smallest node in that binary search tree. For instance,

```
(check-expect (find-min BigBST) 1) ;; using the tree defined above
```

- Uncomment the provided test case
- Write test cases that to check the output of `find-min` when called on each of the trees you created (i.e. `tree-1-node`, `node-wLeftChild`, `node-wRightChild`, and `node-wTwoChildren`).
- Write code for the function `find-min`.

Part E: Write in-order (10 points)

In this part, you'll write a function `(in-order BST)` whose input is any binary search tree and whose output is a list of all of the elements, in order, of the input. You must use the recursive structure of the tree to keep the elements in order. You will earn at most 2 points if your code relies on sorting the list. (Note that you can call `in-order` recursively on the left and right subtrees. Where will the root go?)

For example,

```
(check-expect (in-order BigBST) '(1 7 8 20 31 41 42 60 100))
```

- Uncomment the provided test case
- Write test cases that to check the output of `in-order` when called on each of the trees you created (i.e. `tree-1-node`, `node-wLeftChild`, `node-wRightChild`, and `node-wTwoChildren`).
- Write code for the function `in-order`.

Part F: Write delete (15 points)

The final binary search tree function this week is to write a function `(delete element BST)` whose input value is an integer and whose input `BsT` is a binary search tree. If `element` does not appear in `BST`, then a binary search tree identical to `BST` is returned. On the other hand, if `element` does appear in `BT`, then a tree similar to `BST` is returned, except with the node `element` deleted -- and other adjustments made, as necessary, to ensure that the result is a valid binary search tree. The next paragraph describes these adjustments.

If the value to delete has zero children, it is straightforward to delete. Similarly, if it has only one non-empty child, it is replaced by that child. When the value to be deleted has *two* non-empty children, however, it is more complicated which of its children (or descendants) are to take its place. For the sake of this problem, the node that should take value's place should be the smallest element in `BST` that is *greater* than value. (Use your `find-min` function!) The following YouTube video might be helpful too: [Video: Binary Search Trees (BSTs) - Insert and Remove Explained] You might also look at the remove method from `UnbalancedBSTMap`.

Here are two examples tests:

```
(define BigBST_without20
  (make-BST 42
            (make-BST 31
                      (make-BST 7
                                (make-BST-leaf 1)
                                (make-BST-leaf 8))
                      (make-BST-leaf 41))
            (make-BST 100
                      (make-BST-leaf 60)
                      (make-empty-BST))))

(check-expect (delete 20 BigBST) BigBST_without20)

(define BigBST_without42
  (make-BST 60
            (make-BST 20
                      (make-BST 7
                                (make-BST-leaf 1)
```

```
                                (make-BST-leaf 8))
                        (make-BST 31
                                (make-empty-BST)
                                (make-BST-leaf 41)))
                (make-BST-leaf 100)))

(check-expect (delete 42 BigBST) BigBST_without42)
```

- Uncomment the provided test case
- Before writing code for this problem, write the following six test cases (remember, you'll often need to define new trees!):
    - Remove X from a tree that does not contain X.
    - Remove X from a tree with only one node.
    - Remove X from a tree where X has no children & was in a left subtree
    - Remove X from a tree where X has no children & was in a right subtree
    - Remove X from a tree where X has only a right child
    - Remove X from a tree where X has only a left child
    - Remove X from a tree where X has two children

Problem 2: UnbalancedBSTMap in Java

- Learning Goal: Practice with Java memory models
- Prerequisites: Java basics & BST algorithms
- Submission: UnbalancedBSTMap.java
- Points: 50

Part A: Download the starter file:

Look through the code to make sure you understand the

- UnbalancedBSTMap.java
- UnbalancedBSTMapTest.java

Note: if you are seeing errors due to the @override annotations, you can try:

- Go to Project --> Properties --> Java Compiler, uncheck the box that says "Use compliance from execution environment 'Java SE.18 on the Java build path' " and change the compiler compliance level to 1.6
- or, remove the @override annotations

We've broken up the code into 3 sections (with the following public methods)

- Queries about the tree
  - `isEmpty`
  - `size`
  - `containsKey`
  - `containsValue`
  - `get`
  - `getMinKey`
  - `getHeight`
- Modifications to the tree
  - `clear`
  - `put`
  - `putAll`
  - `remove`
- Helper Methods
  - `inOrderKeys`
  - `getAllKeysInOrder`
- Extra Credit
  - `entrySet`
  - `keySet`
  - `values`

Look for the following style of labels for these sections within both of the starter files:

```
// ////////////////////////////////////////////////////////////////
// *** Modifications to the tree ***
// Methods: clear, put, putAll, remove
// ////////////////////////////////////////////////////////////////
```

Part B: Style Changes (5 points)

- Revise the style used in the constructors of BSTNode. (Please post to piazza if you want a hint!)
- Colleen always makes spelling mistakes and has written `ValueTpye` instead of `ValueType`. Right click on `ValueTpye`, click “Refactor” -> “Rename” and modify the name to be `ValueType`
    - The point of this part is to get you familiar with the Refactoring tool in Java & to make it clear that `ValueTpye` is a variable and not referring to a specific class until the object is created!
- Note - No new test cases will pass based upon this, but please pay attention to style throughout your assignment :)

Part C: Write getMinKey (5 points)

Familiarize yourself with the tests for `getMinKey` and write code to make these test cases pass.

Part D: Write getHeight (5 points)

Familiarize yourself with the tests for `getHeight` and write code to make these test cases pass.

Part E: Write getAllKeysInOrder (15 points)

`toString` and `containsValue` call a method `getAllKeysInOrder`. Write `getAllKeysInOrder` so that the `toString` methods and method containing `containsValue` pass. You might from the Java ArrayList API helpful and you might also find it helpful to search the internet for examples of how to use an ArrayList in Java. You should not use sort for this method. It should be recursive like the Racket function. (However, we added this note late, so won’t take off points).

```java
/* *************** */
// toString
/* *************** */
@Override
// returns a String containing an ordered list of keys
public String toString() {
    ArrayList<KeyType> allKeys = this.getAllKeysInOrder();
    return allKeys.toString();
}
```

Part F: Write remove (15 points)

Familiarize yourself with the tests for `remove` and write code to make these test cases pass. You might find the Racket solution helpful, but remember - in Java you'll be modifying the tree. See above for additional resources about how remove from a BST works.

Part G: Revise the size method (5 points)

We provided a `size()` method that calculates the number of nods in the tree, but the `UnbalancedBSTMap` should be able to keep track of the size without having to calculate it every time.

- add an instance variable treeSize
- modify the method size() to return this.treeSize instead of calculating the size
- update the treeSize variable as appropriate so that all of the test cases pass

Problem 3: Big-O with BSTs

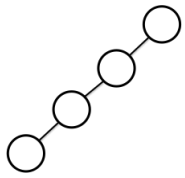
- Learning Goal: Use recurrence relations to calculate the Big-O runtime of a Racket BST
- Prerequisites: recurrence relations and BSTs
- Submission: hw9pr3_bigO.txt
- Points: 10

Part A: Download the starter file

- Download the starter file: hw9pr3_bigO.txt
- Prerequisites: recurrence relations and BSTs

Part B: Determine the Big-O runtime of `find-min` in Racket for a BST with a particular shape (5 points)

Suppose you have a BST for which every non-leaf node has exactly one child subtree: its left subtree. See the example BST to the right. Derive the Big-O runtime for the `find-min` function you wrote in part D of Problem 1.



Your answer should:

- Define any variables you'll use
- Give the recurrence relation
- Give at least 3 generalizations of the recurrence (your "cheat sheet") to substitute into the recurrence
- Show at least 3 substitutions into the recurrence to demonstrate a pattern
- Give the result of the pattern
- State the Big-O runtime

Be sure to define any variables you use.

Part C: Reason about a "worst case" BST (5 points)

Now we want you to reason about the scenario of a BST with nodes having exactly one child!

To this end, answer the following questions:

- Provide a sequence of at least 5 inserts into an empty BST that would yield a BST with all non-leaf nodes having exactly one child.
- Notice how for the function `find-min`, the Big-O runtime was worse than for a perfectly balanced BST. Would this difference also arise for the function `nodeCount`? Explain your answer in a few sentences. Hint: think about the recursive calls in the two functions!

```
(define (nodeCount tree)
   (cond
      [(emptyTree? tree) 0]
      [(leaf? tree) 1]
      [else (+ 1 (nodeCount (leftTree tree))
```

```
(nodeCount (rightTree tree)))]))
```

Problem 4: Extra Challenging Credit - Java BST

- Learning Goal: Work with mutable sets
- Prerequisites: Searching for information about Java on Google
- Submission: UnbalancedBSTMap.java under the label Hw09 Extra Credit
- Points: Up to 15 bonus points

Part A: Implement the three unimplemented methods in UnbalancedBSTMap

- You might find the following open-source version of TreeMap helpful:
    - http://www.opensource.apple.com/source/gcc3/gcc3-1175/libjava/java/util/TreeMap.java
- Note: These methods require that the set that is returned be modifiable and in modifying it modify the underlying representation
- Within your code, acknowledge any sources you have used.