

Homework 09 - CS60 - Fall 2018

Due: Tuesday Tuesday November 20th at 11:59pm (100 points)

[Homework Overview](#)

[Problem 0: Optional and Encouraged Review](#)

[Problem 1: BSTs in Racket](#)

[Part A: Download and read the starter file](#)

[Part B: Practice using the constructors by writing test trees \(5 points\)](#)

[Part C: Write height \(2 points\)](#)

[Part D: Write find-min \(5 points\)](#)

[Part E: Write in-order \(10 points\)](#)

[Part F: Explain the bug in insert \(3 points\)](#)

[Part G: Write delete \(15 points\)](#)

[Problem 2: BinarySearchTree in Java](#)

[Part A: Download/read the starter file and explain put \(2 points\)](#)

[Part B: Style Changes \(3 points\)](#)

[Part C: Write getMinKey \(5 points\)](#)

[Part D: Write getAllKeysInOrder \(15 points\)](#)

[Part E: Write remove \(20 points\)](#)

[Part F: Revise the size method \(5 points\)](#)

[Problem 3: Big-O with BSTs](#)

[Part A: Download the starter file](#)

Homework Overview

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.

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 :-)

- [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!

- Colleen & your amazing grutors!

Problem 0: Optional and Encouraged Review

- You might want to watch these videos
 - [Video: Tree vocab, BST introduction and representations](#)
 - [Video: Binary Search Trees \(BSTs\) - Insert and Remove Explained](#)
- You might find it helpful to print out this document and fill it out - we did this in class, but fluency with these ideas will be super helpful on the homework! :-)
 - https://docs.google.com/document/d/1HIRF25cTicv7TYeXYv2JxAeK9rfN0PBd4dg6Jn_umHM/edit?usp=sharing + **Solution** ([HERE](#))
 - These are optional - so there isn't a place to upload them.

Problem 1: BSTs in Racket

- Learning Goal: Implement BST functions in Racket
- Prerequisites: Racket basics and
- Submission: BSTfunctions.rkt & Gradescope text submission
- 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)`

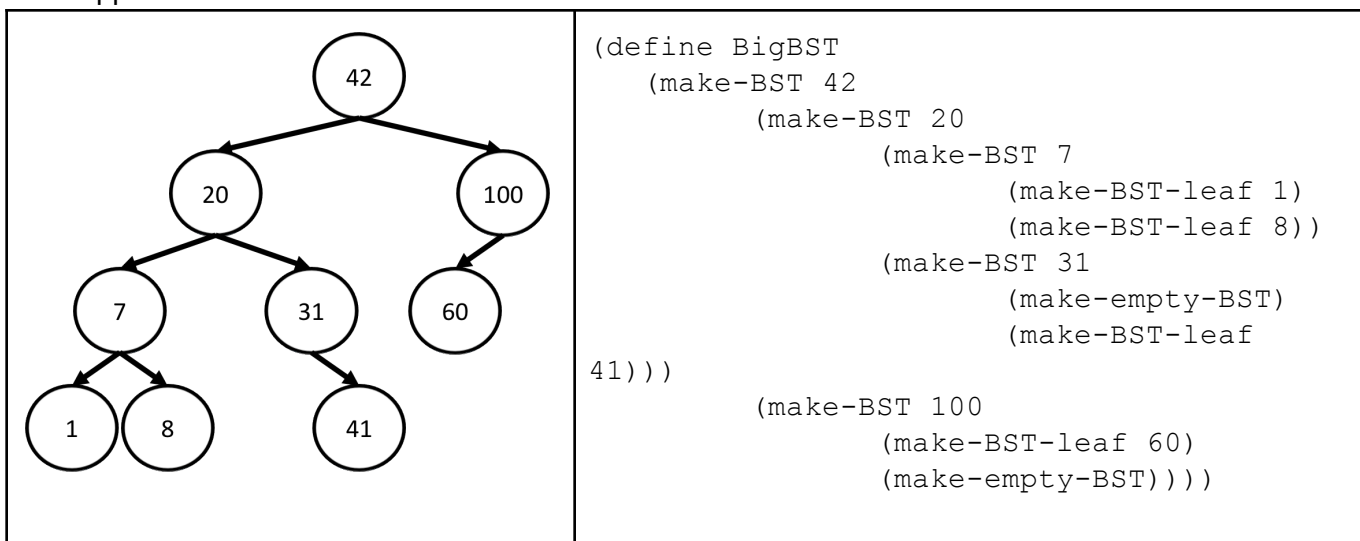
These functions will make the code more readable! For example, in one of the parts, you'll be creating a list -- If we used the list functions to access the contents of our BSTs, it wouldn't be clear if a particular call to `first` or `cons` was working with a list or a BST. These functions would also 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](#) (the BST interface; you won't need to modify this file or turn it in)
 - [BSTfunctions.rkt](#) (you do your work in this file)

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

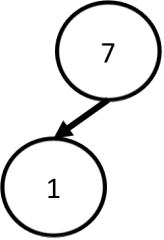
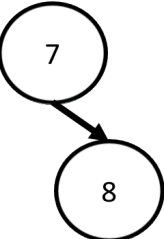
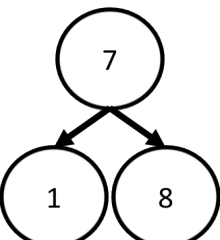
You're going to need to write a bunch of test cases in this assignment! Each one of these test cases is going to need a tree to work with. It'll be easier to write these test cases later on if you've created some trees and given them names so that you can refer to them in your test cases.

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.



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.

	<pre>(define tree-1-node</pre>	<pre>(7) - ascii art version</pre>
--	--------------------------------	------------------------------------

	<code>(define node-wLeftChild</code>	<code>(7) - ascii art version</code> <code>/</code> <code>(1)</code>
	<code>(define node-wRightChild</code>	<code>(7) - ascii art version</code> <code>\</code> <code>(8)</code>
	<code>(define node-wTwoChildren</code>	<code>(7) - ascii art version</code> <code>/ \</code> <code>(1) (8)</code>

Part C: Write height (2 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-equal? (height BigBST) 3) ;; using the tree defined above
(check-equal? (height (make-empty-BST)) -1)
(check-equal? (height (make-BST-leaf 42)) 0)
```

- Write test cases that 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-equal? (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. You definitely shouldn't call `flatten`.** (Note that you can call `in-order` recursively on the left and right subtrees. Where will the root go?)

For example,

```
(check-equal? (in-order BigBST) '(1 7 8 20 31 41 42 60 100))
(check-equal? (in-order tree-1-node) '(7))
```

- 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: Explain the bug in insert (3 points)

Try running the following **incorrect** version of `insert`. In gradescope, write a few sentences explaining why you need to make a new BST every time you call `insert`.

```
;; insert: create a new BST with a key inserted
;; inputs: e    - a key to add;
;;          BST - the BST to add it to
;; output: a new BST with e in it
(define (insert e BST)
  (cond [(emptyTree? BST) (make-BST-leaf e)]
        ; already have the element
        [(= e (key BST)) BST]
        ; insert to the LEFT
        [(< e (key BST)) (insert e (leftTree BST))]
        ; insert to the RIGHT
        [else (insert e (rightTree BST))]))
```

Correct code for `insert`:

```
;; insert: create a new BST with a key inserted
;; inputs: e    - a key to add;
;;          BST - the BST to add it to
```

```
;; output: a new BST with e in it
(define (insert e BST)
  (cond [(emptyTree? BST) (make-BST-leaf e)]
        ; already have the element
        [(= e (key BST)) BST]
        ; insert to the LEFT
        [(< e (key BST)) (make-BST (key BST)
                                   (insert e (leftTree BST))
                                   (rightTree BST))]
        ; insert to the RIGHT
        [else (make-BST (key BST)
                        (leftTree BST)
                        (insert e (rightTree BST)))])])
```

Part G: 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 `BST`, 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](#)

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-equal? (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-equal? (delete 42 BigBST) BigBST_without42)
```

- Uncomment the provided test case
- Before writing code for delete write the following seven 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: BinarySearchTree in Java

- Learning Goal: Practice with Java memory models
- Prerequisites: Java basics & BST algorithms
- Submission: BinarySearchTree.java & Gradescope text submission
- Points: 50

Part A: Download/read the starter file and explain put (2 points)

Download and import the starter files:

- [hw09.zip](#)

Make sure that you put these in the **package com.gradescope.hw09**. This should appear at the top of each of these java files. Having completed Problem 1 should be helpful to you in writing the Java portions. Feel free to refer back to these previous tasks!

Please start by reading put(), size(), containsKey(), and get()!!! In gradescope explain why we have lines like:

```
tempRoot.leftTree = put (inputKey, inputValue, tempRoot.leftTree);
```

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
- Not implemented methods
 - 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  
// //////////////////////////////////////
```

Please start by reading `put()`, `size()`, `containsKey()`, and `get()`!!!

Part B: Style Changes (3 points)

Revise the code in the first `BSTNode` constructor, so that it uses better programming practices. (Please post to piazza if you want a hint!) You can see how we avoided copy and pasting code in the `ListNode` constructors in [List.java](#).

If you haven't already, please start by reading `put`, `size()`, `containsKey()`, and `get()`!!!

Part C: Write `getMinKey` (5 points)

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

Note: One of the tests require that you throw an exception if someone tries to call `getMinKey` on an empty tree

```
@Test(expected=IllegalArgumentException.class)  
public void test_getMinKey0(){  
    // Test tree: empty  
    // ..... 42 .....  
    BinarySearchTree<Integer, String> myMap = new BinarySearchTree<Integer, String>();  
    myMap.getMinKey();  
}
```

Check out how the constructors for `BSTNode` do this. Here's a resource for learning a bit more about Exceptions: <http://beginnersbook.com/2013/04/throw-in-java/>

Part D: Write `getAllKeysInOrder` (15 points)

`toString` and `containsValue` call a method `getAllKeysInOrder`. Write `getAllKeysInOrder` so that the tests for the `toString` methods and the method that call `containsValue` pass. (At this point, everything except for the deletion tests should pass. Remember we want you testing as you go! :-D)

You might find 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.

```

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

```

Part E: Write remove (20 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 F: Revise the size method (5 points)

We provided a `size()` method that calculates the number of nodes in the tree, but the `BinarySearchTree` 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: Gradescope text submission
- Points: 10

Part A: Download the starter file

Assuming N is the number of nodes in a BST.

For these problems it might be helpful to imagine that N is 100000000 because when we do BigO calculations we're concerned about big input size and none of the answers involve only having a small number of nodes in the BST.

Q1: Explain why calling `nodeCount` on a BST requires $O(N)$ steps and does not depend upon the structure of the tree.

```

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

```

Q2: Explain how calling `findMin` on a BST could require $O(N)$ steps.

Q3: Explain how calling `findMin` on a BST could require $O(1)$ steps.

Q4: Explain how calling `findMin` on a BST could require $O(\log_2 N)$ steps.

Q5: In Problem 2 - Part F, we decided to store redundant information (i.e. we could always just calculate the size by traversing the tree so don't technically need to store it). It was also kind of a pain because we had to change multiple other methods. Explain why storing redundant information that we'd need to keep up to date in multiple other locations might be helpful.

Q6: If we wanted to modify `getMinKey` to be $O(1)$, what methods would we need to change?

Q7: Is it possible that inserting a key and value into a BST could take a constant number of steps? If yes, describe the tree and what you could insert.

Q8: Is it possible that inserting a key and value into a BST could take **$O(\log_2 N)$ steps**? If yes, describe the tree and what you could insert.

Q9: Is it possible that inserting a key and value into a BST could take **$O(N)$ steps**? If yes, describe the tree and what you could insert.

Q10: Is it possible that inserting into a BST could take **$O(N^2)$ steps**? If yes, describe the tree and what you could insert.