CS 60 - HW8

Homework 8: Binary Search Trees

In this assignment, we will implement functions that manipulate *binary search trees* (BSTs) in a variety of ways. Remember that:

- We never have duplicate keys in a BST.
- All keys in a left subtree are strictly less than the key at their root.
- All keys in a right subtree are strictly greater than the key at their root.

As we mentioned in the weekly, we will practice with BSTs in both Racket and in Java. Even though it can be difficult to switch back and forth between the two languages, we are asking you to context switch because working from different perspectives can help reinforce the core concepts beyond the specifics of coding.

You are welcome (and encouraged) to buddy or pair program again for all parts of this assignment! (The syllabus can remind you of the differences.) If you have not tried buddy or pair programming, give it a try! Feel free to use the "Search for Teammates" post on Piazza to identify potential programming buddies, or ask the teaching team to help you find a partner during lab hours.

We are really excited to help people on Piazza - so please ask questions early and often! Some more tips:

- For questions, please use Piazza with the **#hw8** tag.
- **Your code will be graded anonymously.** Please do not include your name anywhere in your submission.
- **If you choose to pair program, remember to add your partner to each Gradescope submission.**

— Your profs & your awesome Gruotoring team!

Problem 1: BSTs in Racket

- Learning Goal: Implement BST functions in Racket
- Prerequisites: Racket basics & BST algorithms
- Starter files: **`BST.rkt`**, **`BSTfunctions.rkt`**, **`BSTtests.rkt`**
- Submit: **`BSTfunctions.rkt`**, **`BSTtests.rkt`**, text submission (Part E)

Starter Files and Introduction

- BST.rkt - Contains the BST data structure. You do not need to modify this file or turn it in.
- BSTfunctions.rkt - Where you will do your work.
- BSTtests.rkt - Where you will write additional tests.

For this problem, we will keep things simple to focus on the core BST structure. Specifically, we will require that our BSTs contain integer keys (and no values). Because BSTs cannot contain duplicate keys, these BSTs implement a **set**. (Later, we will have BSTs store both keys and values, i.e. implement a **map**.)

Because Racket is a functional language, it does not have the ability to define object-oriented data structures.[1] To compensate for the lack of classes, we will represent trees using lists. Each node is represented by a list that contains the root key, the left subtree, and the right subtree. The subtrees can themselves be lists, giving rise to a nested list structure.

However, as good CS practitioners, we want to abstract away this underlying implementation from users! In the starter file **`BST.rkt`**, we have provided methods for constructing BSTs, accessing elements of a BST, and querying the structure of a BST:

- Construct a BST
  - **`(make-BST key left right)`**
  - **`(make-empty-BST)`**
  - **`(make-BST-leaf key)`**
- Access elements of a BST
  - **`(key tree)`**

[1] Technically, Racket provides functionality for classes in `racket/classes`, but it is not built-in to the base Racket library.

    - `(leftTree tree)`
    - `(rightTree tree)`
- Query the structure of a BST
    - `(emptyTree? tree)`
    - `(leaf? tree)`

Note that all functions except `emptyTree?` require that the input `tree` be non-empty.

Using these functions will make our code more readable and adaptable.

- This abstraction helps human readers (including ourselves) recognize when we are working with BSTs (and when we are not). For example, say we wanted to create a sorted list from a BST. If we were to use list functions to access the elements of our BSTs, it would not be clear to a human reader if a particular call to `first` or `cons` was working with a normal list, or with one of our special BST lists.
- This abstraction allows us to change the underlying implementation of a tree. For example, we could change the order of the elements to "left subtree, root key, right subtree". Perhaps more compellingly, we could make larger structural changes such as allowing the tree nodes to store both keys and values.

Important: Not using the provided helper functions will lose style points.

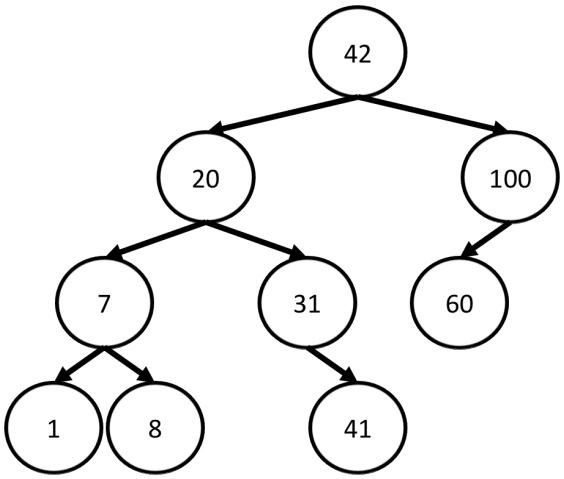
Read through the starter code to help you gain intuition for the rest of this problem.

- You **must** use the recursive structure of the tree to implement the missing functionality.
- You definitely should **not** (1) sort a list anywhere, nor (2) call `flatten`.

Part A: Write test trees (and practice making BSTs)

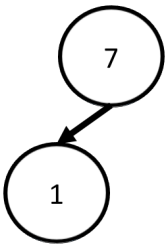
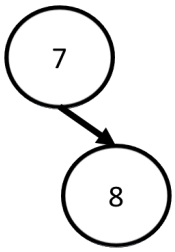
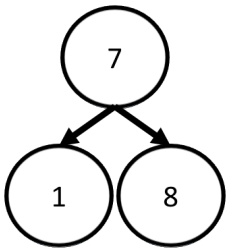
We want to test our BST functions! Each test case is going to need a tree to work with. Let us create some trees and give them names so that we can refer to them in our test cases.

We have defined a complex tree `bigBST` that you can use in your test cases. Here is its graphical representation:




```
None
(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))))
```

In **`BSTtests.rkt`**, create these four simpler trees to use in your testing, paying careful attention to the names given below:

			
`(define tree-wOneNode`	`(define tree-wLeftChild`	`(define tree-wRightChild`	`(define tree-wTwoChildren`

Before moving on, be sure the test cases for the BST data structure pass, in **`BSTtests.rkt`**.

Part B: Test and implement `height`

`(height tree)` takes in a binary search tree and outputs the number of edges in the longest path from the root of **`BST`** to any one of its leaves. Remember that we define the height of the empty binary search tree to be -1.

Example tests:

```
None
(check-equal? (height (emptyBST)) -1)
(check-equal? (height bigBST) 3)
```

Let us use test-driven development. That means you should:

1. First, uncomment the provided test cases in **`BSTtests.rkt`**.
2. Then, write test cases that check the output of **`height`** when called on each of the trees you created (i.e. **`tree-wOneNode`**, **`tree-wLeftChild`**, **`tree-wRightChild`**, and **`tree-wTwoChildren`**).
3. Finally, implement **`height`** in **`BSTfunctions.rkt`** to make the tests pass.

Helpful note: It has been awhile since we used Racket! As a reminder, you can submit in-progress work to Gradescope.

- Submit **`BSTtests.rkt`** to check that your tests pass our sample solution. If not, reread the function description again to make sure you understand the intended functionality.
- Submit **`BSTfunctions.rkt`** to check that your functions pass our tests (including hidden tests).

Be sure to keep the tags **`; provided tests`** and **`; student tests`**.

Part C: Test and implement `find-min`

`(find-min tree)` takes in a *non-empty* binary search tree and outputs the key of the smallest node in that binary search tree.

Example tests:

```
None
(check-equal? (find-min bigBST) 1)
```

For this part, you should:

1. First, uncomment the provided test case.
2. Then, write test cases that check the output of **`find-min`** when called on each of the trees you created.
3. Finally, implement **`find-min`** to make the tests pass.

Part D: Test and implement `in-order`

`(in-order tree)` takes in a binary search tree and outputs a list of all of the elements, in increasing order.

Hint: You can call **`in-order`** recursively on the left and right subtrees. If you did this, where would the root go?

Example tests:

```
None
(check-equal? (in-order emptyBST) '())
(check-equal? (in-order bigBST) '(1 7 8 20 31 41 42 60 100))
```

For this part, you should:

1. First, uncomment the provided test cases.
2. Then, write test cases that check the output of **`in-order`** when called on each of the trees you created.
3. Finally, implement **`in-order`** to make the tests pass.

Part E: Explain the bug in **`insert`**

The starter code provides two implementations of **`(insert e tree)`**. Both take in an element e to insert and a BST. **`insert`** correctly inserts the element, but **`insertWrong`** incorrectly inserts the element.

Run the two versions (in a separate file or in the interactions window). **On Gradescope, complete the corresponding short-answer question**, briefly explaining (1-2 sentences) why **`insertWrong`** is incorrect, and why we need to make a new BST every time we call **`insert`**.

Part F: Test and implement `delete`

`(delete e tree)` takes as input an element **`e`** and a binary search tree **`tree`**. As with all things BST in Racket, **`e`** should be an integer. If **`e`** does not appear in **`tree`**, there is nothing to delete, and the function outputs **`tree`**. (More specifically, the function outputs a BST that is identical in structure to **`tree`**.) Otherwise, **`e`** *does* appear in **`tree`**, and the function outputs a BST that is identical to **`tree`** but with the node containing **`e`** deleted.

But wait, there's more! Generating this new tree requires adjustments to be made to the input **`tree`** ensuring that the result is a valid binary search tree. Let us remind ourselves of these adjustments:

- If the node to delete has zero children, it is straightforward to delete.
- Similarly, if the node has only one (non-empty) child, it is replaced by that child.
- When the node to be deleted has *two* non-empty children, we have to determine which of its children (or descendants) should take its place!
    - For the sake of this problem, the node that should take **`e`**'s place should be its successor, defined as the smallest element in the **`tree`** that is *greater* than **`e`**.
    - *Hint*: Use your **`find-min`** function!
    - If you are unsure where to start or get stuck, try reviewing this video from the weekly.

Here are some example tests:

```
None
(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)
```

For this part, you should:

1. First, uncomment the provided test cases.
2. Then, write the following test cases that check the output of **`delete`**. Use the trees that you created when possible, and define new trees when needed.
    - Remove X from a tree that does not contain X
    - Remove X from a tree where X is the only 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 was at the root & has only a right child
    - Remove X from a tree where X was at the root & has only a left child
    - Remove X from a tree where X was at the root & has two children
3. Finally, implement **`delete`** to make the tests pass.

Problem 2: BSTs in Java

- Learning Goal: Implement BST functions in Java
- Prerequisites: Java basics & BST algorithms
- Submit: **`BinarySearchTree.java`** & text submission (Part A)

Starter Files and Introduction

1. The starter files are in the [CS60 Github repository].
2. See the previous assignments for how to import these files into VSCode. Use **`hw8`** as the project name, and **`com.gradescope.hw8`** as the package name.

Have a look at the starter files. Notice that **`BinarySearchTree`** implements the [`Map` interface] and uses Generics for its keys and values. The **`Map`** interface provides some default methods[2], so a class that implements **`Map`** only needs to implement the abstract methods (click the appropriate tab under "Method Summary").

To help readability, we have broken up the code into sections, separated by labels, e.g.

```java
////////////////////////////////////////////////////////////////////
// Query Operations
// Methods: isEmpty, size, height, containsKey, containsValue, get, getMinKey
////////////////////////////////////////////////////////////////////
```

The sections contain the following public methods:

- Querying the tree
  - **`isEmpty`**
  - **`size`**
  - **`getHeight`**
  - **`containsKey`**
  - **`containsValue`**
  - **`get`**
  - **`getMinKey`**
- Modifying the tree

[2] These default methods are not covered in CS 60.

    - **`clear`**
    - **`put`**
    - **`putAll`**
    - **`remove`**
- Helper methods
    - **`inOrderKeys`**
    - **`getAllKeysInOrder`**

- Not-yet-implemented methods: These methods are required by the **`Map`** interface and so require a method stub. The details of the methods are not covered in CS 60 (but we are happy to talk more in office hours).
    - **`entrySet`**
    - **`keySet`**
    - **`values`**

A few notes:

- The instance variables have been shortened to **`root`**, **`left`**, and **`right`**. Be careful to differentiate between the instance variable **`this.root`** and the local variable **`root`** in helper methods. Helper methods should never access **`this.root`**.
- According to the **`Map`** interface, **`put`** returns the *previous* value associated with **`key`**, or **`null`** if there was no existing mapping for **`key`**. (The videos defined **`put`** differently and had it return the *new* value associated with **`key`**.)

Some guidelines:

- Your task in the rest of this problem is to implement the missing functionality in the **`BinarySearchTree`** class, helpfully marked with TODO.
    - Do not do it all now -- the sub-problems provide some extra context.
    - Some TODOs require you to implement missing functionality. Other TODOs are there to warn developers (you) that methods will not work until other functionality is implemented.
    - You do not have to implement the “not-yet-implemented” methods.
- Each method that you implement in Java corresponds to a procedure you implemented in Racket. Use your Racket code to help out!
    - As in Racket, you **must** use the recursive structure of the tree to implement the missing BST methods. You definitely should **not** sort a list anywhere.
- Remember to test as you go!

Part A: Explain code in `put`

Carefully read the methods `size()`, `containsKey(...)`, `get(...)`, and `put(...)`.

Then, **on Gradescope, complete the short-answer question** briefly explaining (1-2 sentences) why we have a private helper version of **`put`**, and why we need to use it in the public version of **`put`**.

```java
public ValueType put(KeyType key, ValueType value)
private BSTNode put(KeyType key, ValueType value, BSTNode root)
```

Part B: Refactor the `BSTNode` constructor

Let us start off by refactoring the code, which

- Restructures existing code without changing its behavior
- Improves design, structure, and/or implementation while preserving functionality

Refactor the two-parameter `BSTNode` constructor to remove duplicated code.

Part C: Implement `getMinKey`

Familiarize yourself with the tests for **`getMinKey`**, then implement **`getMinKey`** to make these test cases pass.

If you have to throw an exception, be sure to pass a helpful message to the **`Exception`** constructor. Think of yourself as a user of the **`BinarySearchTree`** class. What message would help you understand what went wrong when calling the **`getMinKey`** method? If you would like a refresher, our **`LinkedList`** class in HW6 throws exceptions.

Note that exceptions are passed "up" method calls; that is, if **`method1`** throws an exception, and **`method2`** calls **`method1`**, then **`method2`** will throw the same exception. That means you should only have to throw one exception in your implementation.

Part D: Implement `addKeysToArrayList`

Familiarize yourself with the tests for **`toString`** and **`containsValue`**. In the **`BinarySearchTree`** class, note that these methods call a private helper method **`getAllKeysInOrder`**, which in turn calls **`addKeysToArrayList`**.

Implement **`addKeysToArrayList`** so that the test cases pass. The tests check **`toString`**, **`containsValue`**, and **`put`** where the value for an existing key is replaced. You might find the Java ArrayList API helpful, or you can search the internet for "example how to use Java ArrayList".

Part E: Implement `remove`

Familiarize yourself with the tests for **`remove`**, then implement **`remove`** to make these test cases pass. We recommend that you consult the Racket and Java code for **`insert`** / **`put`**, and your Racket code for **`delete`**. Remember that, in Java, we modify trees rather than build new ones.

Part F: Refactor to store and use an instance variable `size`

In our **`LinkedList`** class, we explicitly stored the size of the list in an instance variable **`mySize`**. What are the trade-offs of storing the size explicitly?

Let us modify our **`BinarySearchTree`** to also explicitly store its size (number of nodes).

- Add an instance variable **`size`**. (Nifty! Java can differentiate between the instance variable **`size`** and the method **`size()`**.)
- Modify the method **`size()`** to return **`this.size`** instead of calculating the size.
- Update the **`size`** variable in the **`BinarySearchTree`** methods as appropriate so that all of the test cases pass.

Problem 3: Big O with BSTs

Note: This part is its own assignment on Gradescope.

Assume N is the number of nodes in a binary search tree.

Notes

- Remember that when we use Big O, we care about asymptotic growth, i.e. what happens as the input size tends to infinity. Therefore, for these questions, you might find it helpful to imagine that N is a **very big** number.
- Refer to earlier problems for the BST functions as needed.
- *All short answer questions can be successfully answered in a few sentences at most.*

Analyzing BSTs using Racket

1. Explain why calling `node-count` on a BST takes O(N) steps and does not depend upon the structure of the tree.

```
(define (node-count tree)
    (cond [(emptyTree? tree) 0]
          [(leaf? tree) 1]
          [else (+ 1 (node-count (leftTree tree))
                     (node-count (rightTree tree)))]))
```

Describe a scenario in which calling `find-min` on a BST would take the given number of steps.

2. O(N) steps
3. O(1) steps
4. O($\log_2$N) steps

Modifying the `BinarySearchTree` class in Java

5. Reread HW8, Problem 2, Part F. In this task, we decided to store the size of the tree in an instance variable so that we would not need to traverse the tree each time we wanted to retrieve its size. However, this design decision requires changing multiple other methods. Explain why storing redundant information can be helpful, even if doing so requires keeping variable values up-to-date in multiple other locations.

6. How could we modify the BST class so that `getMinKey` takes O(1) steps? That is, what instance variable(s) would we need to add, and what would we change about `getMinKey`? What other methods would we need to change?

Inserting a node into a BST

Is there a scenario in which inserting a node into a BST would take the given number of steps? If yes, describe the tree and node.

7. O(1) steps
8. $O(\log_2 N)$ steps
9. O(N) steps
10. $O(N^2)$ steps

Rubric

#	Name	Autograder	Functionality	Other	Style	Testing	Total
1	Racket BSTs	16	0	3	5	10	34
2	Java BSTs	15	4	3	5	0	27
3	Big O with BSTs			19			19
		37.5%	5%	32.5%	12.5%	12.5%	80

We will check for the following elements of **bad style**:

- Comments
  - Complicated lines of code are not explained.
  - Too many comments can be just as confusing for a reader as no comments at all. We want to save inline comments for particularly tricky pieces of code and do not need one for every line.
  - [Racket] Missing a function-level comment, or the function-level comment is missing a description of the function and its inputs and outputs.
  - [Java] Missing a description of a non-private class / variable / method, or the function-level comment does not conform to Javadoc style. (Exception: Test methods)
- Naming
  - Variable or method names are not helpful or do not follow language-specific conventions.
- Formatting
  - Function(s) are written in a way that makes it hard to read (e.g. lines too long so text wraps, missing meaningful indentation).
  - [Racket] Racket convention is to put all closing parentheses on the same line.
  - [Java] Curly braces should follow style guide (opening braces same line as function declaration / control structure, closing braces on own line except when followed by `else` or `else if`, function and control structure bodies enclosed in curly braces).
- Redundancy
  - Function(s) have copy-pasted code rather than using helper functions.
  - Function(s) explicitly returns `true`/`false` rather than returning predicates directly, have redundant predicates rather than using `else`, or use unnecessary `else` after `return` statements in `if` / `else` if blocks.
  - Function(s) are overly complicated, e.g. include extra base case or other unnecessary code.
  - Avoid unused imports.
- Helper Functions

    - Code is not “self-documenting”, e.g. does not use built-in / provided functions where possible. (Exception is if the assignment instructions specifically forbid the use of such functions.)
- Control Structures
    - [Racket] Good style uses `if` over `cond` for single predicates (plus possibly an `else` case), and uses `cond` over `if` to avoid nested `if`s (unless doing so would repeat predicates).
    - [Java] Good style uses `for` loops over `while` loops whenever possible.
- Class Design [Classes only]
    - Classes / variables / methods have inappropriate access modifiers (`private` vs `protected` vs `public`).
    - Variables / methods have inappropriate `static` modifier and should belong to an instance of the class (non-`static`) or the whole class (`static`).
    - Variables / methods have inappropriate `final` modifier and should be changeable (non-`final`) or unchangeable (`final`).
    - Instance variables and methods should be referenced using the keyword `this`, i.e. `this.fieldName` and `this.methodName(possibly arguments here)`.
    - Class variables and methods should be referenced using the class name, i.e. `ClassName.fieldName` or `ClassName.methodName(possibly arguments here)`.
    - Mark methods with `@Override` when (1) a class method overrides a superclass method, (2) a class method implements an interface method, or (3) an interface method respecifies a superinterface method.
- Tests
    - Use appropriate flavors to test against booleans / test for equality. Racket `check-true` / `check-false` / `check-equal`, Java: `assertTrue` / `assertFalse` / `assertEqual`.
    - [Java] Put `@Test` on its own line before all test functions.
    - It is generally better practice to test one functionality per test method! This potentially means multiple tests for a single function. This approach isolates exactly what functionality is failing without having to retest multiple times (and helps prevent cases in which we fix one bug only to discover another).
- Miscellaneous
    - Files contain startup notes such as “TODOs” or “delete this”.
    - Files contain debugging statements, including unnecessary print statements or output trace statements.
    - Files contain student name(s). We grade everything anonymously in CS60 so that things are graded as fairly as possible.