Lab 4

CMSC 15100 Introduction to Computer Science I

This lab is all about binary search trees. To get started, you will implement several polymorphic functions on BSTs. Then you will write some functions that work specifically with a dictionary of all English words. After you've finished the lab, you should show Instructor your work.

For this lab, you can use either the lab computers or your personal computer. Start by navigating to the folder CS151 on your computer of choice and creating a subfolder "lab4". You'll need the binary search tree implementation from class – download `BST.rkt` from Canvas and save it in "CS151/include".

BST ABC's

We already saw several generic functions on binary search trees, such as `BST-insert` and `BST-delete`. These methods are implemented for you to use in the file `BST.rkt`. To get practice working with this data structure, we'll implement a few more methods that will give us more flexibility when manipulating binary search trees.

Create a new file `lab4.rkt` inside the "lab4" folder where you will put your code for this lab. You should not need to modify BST.rkt for this assignment. In addition to the normal `require` statements we have at the top of every file, add the following line to the top of `lab4.rkt` to include the BST implementation,

```
(require "../include/BST.rkt")
```

You don't need to modify the code in `BST.rkt`, but you will need to understand how to use each of the functions in the file, as well as how the `BST` type is structured. It's recommended that you read the type annotations and purpose statements of each function, but not the function defintion, and if you have any questions on how to use these functions, ask Aritra. Some of the tasks below will be a lot easier if you use the existing BST functions in this file. Some of them will require you to write your own recursion on BSTs.

Task 0. Define variables for three different binary search trees of `Integer`s by using the `BST` and `Node` constructors.

These variables can be used in your tests. It's impossible for Racket to check if two functions are equal. When using `check-expect` with binary search trees, this implies that it's impossible for Racket to check if two objects of type `BST` are equal, because Racket would need to check if the comparison functions `lte?` are equal. Consequently, when you write tests below, use `check-expect` on the `bst` part of the binary search tree.

Task 1. Implement the following functions which make it easier to add and create binary search trees

```
(: singleton : (All (A) A -> (BST A)))
;; creates a BST with exactly one node with the given value

(: insert-all : (All (A) (Listof A) (BST A) -> (BST A)))
;; inserts an entire list of things into the BST

(: list->BST : (All (A) (Listof A) (A A -> Boolean) -> (BST A)))
;; given a list of values and a comparison function, create a BST with all the values
```

Task 2. Implement the following functions that compute some interesting statistics about a given binary search tree,

```
(: size : (All (A) (BST A) -> Integer))
;; returns the number of nodes in the BST

(: contains? : (All (A) (BST A) A -> Boolean))
;; returns whether or not this value is in the BST

(: count-value : (All (A) (BST A) Integer -> Integer))
;; counts the number of times the given value appears
;; Don't look in a left or right child if it's impossible for the value to be there!

(: count : (All (A) (BST A) (A -> Boolean) -> Integer))
;; counts the number of nodes that satisfy the given requirement function
;; This function can't take advantage of the BST structure (versus just a Tree)
;; and its runtime should be O(n).
```

Finding Words

Binary search trees are used with different types of ordered data. Here we'll work with a list of common English words, which are sorted in `lexicographic order` (also called alphabetic order).

The file `word-list.rkt` on Canvas contains the 20000 most common English words. Download this file and save it in your "CS151/include" directory, however it's not recommended that you open the file because it will make DrRacket slow due to how large the file is. Just add the following `require` statement to the top of your `lab4.rkt` file,

```
(require "../include/word-list.rkt")
```

`word-list.rkt` is a very boring file. It provides exactly one variable, `words-vec`, which is a `(Vectorof String)` containing about 20000 words. You can check that the file is imported correctly by running

```
(vector-length words-vec)
(vector-ref words-vec 8132)
```

in the interaction window.

To turn it into a BST, define a variable in your code,

```
(: common-words : (BST String))
(define common-words (list->BST (shuffle (vector->list words-vec)) string<=?))
```

The comparison function is `string<=?`, which is Racket's built-in function for the lexicographic order on strings. The `shuffle` function randomly rearranges the word list before converting it to a BST. This is done to ensure that the tree is balanced. You can test that if the call to `shuffle` is removed, even defining the `common-words` variable will take much longer to run.

You should not use the `words-vec` variable any more.

For these tasks, you don't have to write any `check-expect` tests. You can test your code by running the functions in the interaction window.

Task 3. Create a new BST of `Strings` called `word-and-drow` which contains all words in `common-words` and also all of their reverses. As a result, the size of `word-and-drow` should be around 40000.
You can use the following helper function `string-reverse`:

```
(: string-reverse : String -> String)
(define (string-reverse s)
  (list->string (reverse (string->list s))))
```

This problem can be solved without recursion, using purely the built-in BST functions and your functions above, and a call to `map`.

The final tasks will focus on finding the longest word within a certain range.

Task 4. Implement the following function

```
(: subtree-between : (All (A) (BST A) A A -> (BST A)))
;; return the nodes that lie between the first given value and the second given value
```

For example, `(subtree-between common-words "ca" "cb")` will produce a binary search tree with just the words that start with "ca".
It may be helpful to implement the function first for a binary search tree of `Strings`, and then convert the function to be polymorphic.

Finally, we would like to answer the question, "what is the longest word that starts with 'ca'"?

Task 5. Implement the following function

```
(: longest-between : (BST String) String String -> String)
;; returns the longest String in the BST between the given two strings
```

For this task, use a helper function `longest` which outputs the longest word in a BST.

In truth, though, long words may not be interesting words, and what we would really like to know is, "what is the most interesting word that starts with 'ca'"? One measure of the interestingness of a word is the *Scrabble score* of a word. Scrabble is a word-based board game where different points are awarded for different words. The score of a word is the sum of the scores of each letter, and each letter is worth a fixed number of numbers depending on which letter it is. For example, the word "TREE" is worth 4 points because each of T, R, and E are worth one point each. The word "SEARCH" is worth 8 points because S, E, A, R are worth one point each but C and H are worth two points each.

See the Wikipedia page `https://en.wikipedia.org/wiki/Scrabble_letter_distributions#English` to find the score of each letter.

Task 6. Implement a function `scrabble-score` which takes in a `String` and outputs its score in Scrabble.
Using the `scrabble-score` function, implement this function,

```
(: most-interesting-between : (BST String) String String -> String)
;; returns the word with the highest Scrabble score between the given two strings
```

To finish this lab, find some interesting words in the BST.