CS 1102

A term. 2017.

Assignment #4

**Due: Tuesday September 26, 2017 @ 9:00 p.m. via Canvas**

**Accepted until September 27, 2017 @ 1 p.m. with 25% late penalty**

## Fixes

1. Changes filenames for three parts to “part 1.rkt”, “part 2.rkt”, and “part 3.rkt”
2. You do not need check-expects for the random-widgets function
3. Reordered questions in part 2 (put find ahead of insert, but did not change the question content)
4. A few typography changes to fix grammar errors or to improve clarity.

## General assignment guidelines

1. Requested functions must have signatures and test cases. They must also be defined at the top level.
2. You may not change the signature or name of any of the functions. You are also forbidden to modify the structure definitions.
3. Pay particular attention to the requirements in Part 2 and Part 3.
4. As always, you may use any of your code from previous assignments, as well as any code we have posted in class. Typically you should credit others’ work, but if you are copying code posted on the Canvas site you may skip that step for this class.
5. You will use the Intermediate Student Language with Lambda, and may use all constructs in the language unless contradicted by the assignment.
6. You might want to watch the optional video on lambda posted on the course schedule.
7. Part 1 prepares you for Part 2; Part 2 prepares you for Part 3. Dividing things up by parts will not work well.

## Part 1. Basics of binary search trees

Your boss was impressed with your work in providing functionality for answering queries about widgets. However, there is one remaining problem: speed. Although at present few types of items are produced, your boss is looking to the future and possibly extending to millions of different widgets being manufactured. He has asked you to use a simplified version of the widget structure to build some infrastructure for quickly accessing the different types of widgets. Specifically, he wants you to implement functionality for a binary search tree in order to speed searching for a specific widget among the (eventually) many millions of different widgets the company manufactures.

(define-struct widget (name quantity price))

;; a widget is a (make-widget String Natural Number)

; same as assignment #3, except no parts component

(define-struct bst (widget left right))

;; a BST is either

;; false, or

;; a (make-bst widget bst bst)

For this part you will use ISL-lambda, and any constructs that exist. You are to develop the following functions:

1. random-widgets: Natural Natural 🡪 (listof Widget). The first parameter is the number of random widgets to create, and the second parameter is the maximum value the parameter can take on. For example:

> (random-widgets 5 9999)

(list

(make-widget "2407" 9711 7331)

(make-widget "8044" 7194 9666)

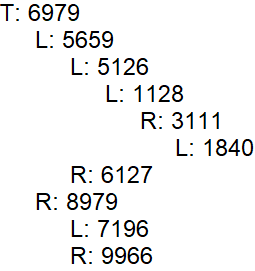
(make-widget "9430" 7040 6478)

(make-widget "4854" 9541 3171)

(make-widget "5789" 7782 7406))

While these widget names are not realistic, they are useful for testing your code without having to hand-generate a bunch of widgets. This functionality will come in handy later when you do performance analysis. When you generate random widgets, use a large value for the maximum value (e.g. 999999999) to greatly reduce the likelihood of having non-unique identifiers.

1. insert: Widget Bst 🡪 Bst. Adds the widget to an existing binary search tree in the correct position. Widgets are ordered according to their name. Specifically, you should use the string<=? function.
2. insert-all: (listof Widget) Bst 🡪 Bst. Inserts all of the widgets into the binary search tree. You may assume the widgets in the tree are ordered according to their name and the string<=? function.
3. find-name: String Bst 🡪 Widget | false. Returns the widget whose name corresponds to the given string, or false if that widget does not exist. If there are multiple copies of the same widget, returning either one is ok.
4. render: Bst 🡪 Image. Display an image of the binary tree using the “name” field of widget. Using an approach similar to assignment #3 is fine. The main purpose of the function is so you can inspect your tree visually for correctness. It should look something like:



T represents the top of the tree, and L and R represent left and right branches, respectively.

Hint: it should go without saying, but follow the template for binary search tree. You are not required to submit the template, but hopefully by now you don’t need to be told to start with it.

What to hand in: “part 1.rkt”

## Part 2. Writing functions that create functions

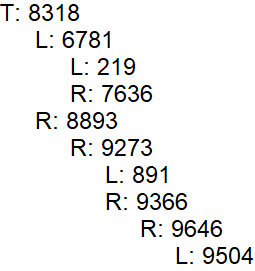
The Head of the Advanced Widget Engineering Section at the company has been impressed with your work, specifically your prior project on creating abstract functions to operate on widgets. To see if you’re cut out for AWES, she wants you to reimplement several functions from Part 1. However, **The Head is fairly strict and does not permit you to use the list template, or any recursion aside from the binary search tree template** (“let’s see if you really understand abstract functions.”).

1. find: eq lt value bst 🡪 widget | false. Searches a binary search tree. eq is a function that consumes a value and a BST, and returns whether the (relevant property of the) current node is equal to the value passed in. lt (Less Than) is a function that consumes a value and a BST, and returns whether the value passed in is less than the (relevant property of the) current node. value is what is being searched for, and bst is the binary search tree. You may assume the tree is ordered according to the function lt.
2. insert: fn Widget Bst 🡪 Bst. Adds the widget to an existing binary search tree in the correct position. Widgets are ordered according to the function fn. All The Head will say for additional clarification is:
   1. You must pass in a function that compares a Widget and a node in a binary search tree.
   2. It must be capable of creating a tree based on any field of widget (name, quantity, or price).
   3. The ordering could be increasing, decreasing, based on the length of the string, or …
3. insert-all: fn field (listof Widget) Bst 🡪 Bst. Inserts all of the widgets into the binary search tree. Values will be ordered by the function fn. The function fn will be applied to part of the widget structure referred to as field. You **must** reuse your insert function from Part 2, question 1.

For example:

> (render (insert-all string<=? widget-name false (random-widgets 10 9999)))

Might produce



**Reminder:** you may not use the list template, and must use foldr here. You begin to object “But foldr takes a function with 2 parameters, while insert requires **3** parameters…” The Head smiles and says “Exactly. That’s why it’s a challenge.” As The Head walks away, you call out “But fn and field aren’t even a valid function! Am I expected to make one from them?” Unfortunately, you don’t receive a reply other than a cryptic smile.

1. You get a note from The Head saying “If you can do question #2 and #3, you obviously don’t need to waste your time recreating the render function. That’s a trivial tasks for a programmer of your caliber. Instead you should work on Part 3.”

Hints for question 3:

1. Conceptual complexity just jumped up from assignment #3.
2. Look at the provided example of insert-all and forget about foldr. Just think about how to *construct* a function from those components you could pass to insert to put a single element into the tree.
3. Once you have that working, then think about how to incorporate foldr to insert all of the elements at once.
4. You won’t need many lines of code. My solution is 8 lines long, and I’m fairly generous with line breaks.

What to hand in: A file called “part 2.rkt”. Make sure you have multiple test cases of your functions.

## Part 3. Performance measurements

The Head stops by and asks you to collect some timing data on how well your new codebase performs. Specifically, your job is to test it against filter (“Let’s see if you can beat a built-in primitive operation for finding things.”). Rather than just running a few tests, your job is to construct a set of functions that will generate the test cases to be run and time how long they take to complete.

Testing specifications:

* Test lists of length 10k, 100k, and 1 million (1 million may take a bit of time to create)
* For each list length, you should use built-in functions to time how long it takes to find a widget 25% of the way through the list, 50% of the way through the list, and 75% of the way through the list. You should also find a widget that is not in the list (there’s an easy way of picking a value guaranteed to not be in the list)
* You will compare the performance of using filter on a list vs. finding it in a binary search tree. You should construct the tree ahead of time to avoiding counting the time to generate the tree in the time to find an item.

The Head, impressed with your performance in part 2, gives you one additional constraint before you can join the Advanced Widget Engineering Section: you must write a function that generates the code that will execute your tests and time them. Realizing that description may have made little sense to you, she sketches out code you should be able to work with and puts it in starter.rkt for you.

Create a 4x3 table showing your timing results for filter. Create a similar table for using your binary search tree (label which is which). Your table does not have to be pretty – as long as we can clearly read what is going on that is fine. Having your results as a comment in your file is ok.

What to submit:

1. Your code.
2. You probably want to include the starter file with part 3 (you might want to include it for other parts as well).
3. The tables, as comments.

Submit a file called “part 3.rkt”

Hints:

1. Again, complexity and abstractness just increased. You are now working with functions that generate and return another function, and then having other code invoke the created function later.
2. As always, start less abstract. Figure out how to run a timing test on an element in the list using filter and using find in a BST.
3. A good next step is writing some code that can automatically extract the element 25% of the way through the list, and filter/find on the appropriate widget-name.
4. Then think about how to generalize what you’ve done to lambda functions.