

CS51 Assignment 1: Trees, Trees, Trees

Due: Friday, 13 February 2009 at 5:00 PM

Total Points: 50 (including 10 style points)

In this assignment you will get practice with Scheme and recursion by working with trees in a couple different forms.

1 Setup

To complete this problem set you will need DrScheme 4.1.4 and the `asst1.scm` file that we have provided at <http://cs51.seas.harvard.edu/docs/asst1.scm>. Make sure the language is set to “Intermediate Student”. If you have any questions on how to work with DrScheme, please refer back to the zero’th assignment, post to the bulletin board, or ask a TF.

2 Style

For this and all subsequent assignments, a substantial portion of your grade is based on style. Remember to consult the CS 51 style guide on the course website at <http://cs51.seas.harvard.edu/docs/style.pdf> if you have any doubts about what constitutes clear, well-written code. Even if you don’t have specific questions, be sure to review the style guide to refresh your memory!

3 Testing your code

How do we know when a program is correct? That is, how can we ensure that the code we write will do what we intend it to do? We can read it carefully,¹ but that’s usually not enough. We can try to prove it correct, but for most programs that’s too difficult to do. We probably have to write some tests.

¹Or in some cases, ask someone else to read it carefully—but not in CS51.

Recall that we introduced the function `check-expect` in the last problem set. We expect that you will find it necessary to use tests to verify the correctness of your code. Please be sure to submit these tests with your assignment. Still unsure of what a test might look like? Suppose we want to test `inside-unit-circle?` from the previous problem set. Using `check-expect`, we might write:

```
(check-expect (inside-unit-circle? 0 0) true)
(check-expect (inside-unit-circle? .5 .4) true)
(check-expect (inside-unit-circle? 1 0) true)
(check-expect (inside-unit-circle? 0 -1) true)
(check-expect (inside-unit-circle? 'three' 'four') false)
(check-expect (inside-unit-circle? -3 -4) false)
```

This runs six tests, each of which evaluates an expression on the left (such as `(inside-unit-circle? 0 0)`) and compares it to the value of the expression on the right (such as `true`). If they are `equal?`, the test passes, but if they aren't, an informative error message is displayed.

Good tests exercise every branch of your code. In the example above, note that it tests a number of cases that the function might encounter: some with “normal” input (`0 0`, `.5 .4`, and `-3, -4`), two “corner” cases (`0 -1`, and `1, 0`), and a “bad” input (`'three' 'four'`). Note that a point with negative components is not a bad input. In general, your tests should cover every path through your program. This means that you will have to write tests with your specific code in mind. For example, suppose your Scheme function contains this piece of code:

```
(if (qux? obj)
    (handle-qux obj)
    (handle-nonqux obj))
```

A good set of tests for that code will provide cases where `obj` is not a `qux` and cases where it is, to ensure that both branches of the `if` work properly.

Try `check-expect` now. Start DrScheme and write a file that contains `(check-expect (- 2 1) 4)`. When you run your program, you will get a pop-up error message.

In this and future assignments, we may provide you with a number of tests for the code you write, and often we will require you to add tests of your own.

4 Trees

(39 points total)

In this problem set we'll implement and use two types of trees: *binary* and *n-ary* trees.

4.1 Binary Trees

In lecture, we saw how a binary tree is recursively defined. Each node has a value and two children, typically referred to as *left* and *right*, which are also binary trees. Of course, binary trees can be empty – a node with two empty children is referred to as a *leaf* node.

A binary search tree is a binary tree with an additional constraint on its nodes. Namely, if we have a node *N* that has a value *V*, then *N*'s left child and all of its descendants will have values less than *V* and *N*'s right child and all of its descendants will have values greater than *V*. This constraint allows us to efficiently traverse a tree when we are “searching” for a specific value since we know which child tree would contain the value. For simplicity, we'll only consider trees with integer values for now, and we'll assume that each value occurs only once.

Exercise 1.

19 points

First, we need to define a Scheme representation of a binary tree. We'll do this by defining a new structure, `tree`, with fields for the value, left child, and right child.

```
(define-struct tree (value left right))
```

In order to set the children fields of leaf nodes, we'll want a special tree that represents the empty tree. In order to satisfy the `tree?` predicate, it needs to be a `tree` struct, but must be different from all other trees. We'll set its `value`, `left`, and `right` fields to the empty list. We also define a predicate for checking if a tree is the empty tree. These are comparable to `empty` and `empty?`, except for trees instead of lists.

```
(define empty-tree (make-tree empty empty empty))
```

```
(define (empty-tree? t) (equal? empty-tree t))
```

(a) [3 points] Now write a predicate, `binary-tree?` that checks if a tree is a valid binary tree. Don't worry about checking whether the tree is a valid binary *search* tree, just check that it is a valid binary tree. Make sure you know what the difference is! Also remember what function(s) you get for free from Scheme when you define-struct tree...

To use our new binary tree as a binary search tree, we will need to define two functions. We will use the first function, `tree-insert`, to build up our tree. The second function, `tree-lookup`, will follow the structure of the tree to determine whether a specified value is in the tree. For this exercise, it is safe to assume that the arguments passed to `tree-insert` and `tree-lookup` are well-formed binary search trees, with all values being numbers.²

For a binary search tree to be efficient, it should be *balanced*. That is, at each node, roughly half of its descendants should descend from its left child, and the other half should descend from its right child. You do not need to implement balancing here, but you should think about why it is important.

²Think about how you might check whether a tree is a valid binary search tree!

(b) [6 points] Write `tree-insert`. It should take an existing binary search tree and a value, returning a new binary search tree with the value properly inserted.

(c) [6 points] Write `tree-lookup`. It should take a binary search tree and a value, returning whether or not the value is in the search tree.

(d) [4 points] We've provided a function `list->tree` which takes all the elements of a list and puts them into a binary search tree using `tree-insert`. Of course, `list->tree` won't work until `tree-insert` is implemented, so we've commented it out for now. Note: `list->tree` inserts the values into the tree in reverse order (last element in the list is inserted first)—this may be useful to know when debugging your code. We'll use this function to concisely build binary search trees, which will be useful for testing. We've written a couple tests for the functions you've defined above; add at least four of your own.

4.2 *n*-ary Trees

In nature, trees often have more than two branches. A computer scientist's tree can also have more than two branches. Some data structures even permit various nodes to have differing numbers of children. In this exercise we will define a structure that permit varying the number of children at each node.

Exercise 2.

20 points

In order to accommodate an arbitrary number of children, our *n*-ary tree struct has two fields, `value` and `children`. In this exercise, `value` will be restricted to strings; `children` is a list of children *n*-trees.

```
(define-struct n-tree (value children))
```

Unlike the binary search tree, there is no inherent way to organize an *n*-ary-tree. In other words, if we're looking for a specific node or key, there isn't necessarily a rule we can follow to find it. We can identify a specific node by its "path" that begins from the root node. We can write down the path to a node as the keys of the intermediate nodes. Naturally, we'll represent paths as lists of strings.

We're giving you some support functions to make it easier to construct hierarchical *n*-ary trees. Feel free to have a look at them to see how they work, but there are a few things that we haven't seen yet in class, so don't worry (or spend too much time on it) if you can't follow everything.

The only function we'll use, `construct-n-tree`, takes a list of paths and a root key and constructs an *n*-tree with nodes at the end (and intermediate) points of each of the paths.

(a) [10 points]

The DNS (Domain Name System) system translates human readable internet addresses (seas.harvard.edu) into actual IP addresses (140.247.51.248). If we reverse the addresses, we

notice that there is a hierarchy. All names that end with “edu” are servers that belong to educational institutions, all names that end with “harvard.edu” are servers that belong to Harvard, etc. Thus, we can represent all the DNS names in a hierarchy that has “com”, “edu”, “net” at the top with specific institutions underneath. (Ignore other possible DNS suffixes for now.)

Now that we can build trees, let’s build a small DNS hierarchy. In order to give all addresses a common ancestor, we’ll add an empty string to the beginning of each path. This is relatively standard with DNS; see http://en.wikipedia.org/wiki/Root_nameserver for details.

```
(define dns-addresses (list
  (list "" "edu" "harvard" "fas")
  (list "" "edu" "mit" "csail")
  (list "" "edu" "harvard" "seas")
  (list "" "edu" "mit" "ll")
  (list "" "edu" "harvard" "seas" "cs51")
  (list "" "com" "apple" "itunes")
  (list "" "edu" "harvard" "my")
  (list "" "net" "cs50" "www")
  (list "" "com" "google" "www")
  (list "" "org" "slashdot" "www")
  (list "" "com" "digg" "www")
  (list "" "com" "facebook" "www")))

(define dns-tree (construct-n-tree "" dns-addresses))
```

Your job is to simulate a DNS lookup using an n-ary tree. In other words, given an address, you should check if it is in the DNS tree. To do this, write a function, `n-tree-lookup`, which checks if a given path leads to a node in the tree. If `path` is the empty list, `n-tree-lookup` should return true. Note that the path does not have to lead to a terminal node – for example,

```
> (n-tree-lookup (list "" "edu") dns-tree)
```

should be true. The order of the address has to be reversed, since this is the order of the hierarchy. That is, to search for `www.google.com`, you’ll have to pass

```
(list "" "com" "google" "www")
```

to `n-tree-lookup`. Feel free to write any helper functions necessary to implement `n-tree-lookup`. Finally, be sure to add some tests!

(b) [10 points]

In the last exercise, you followed a particular path down the tree. Now we will search for specific node and return the complete path. This tree places several Harvard professors in various departments and/or research groups.

```
(define professors
```

```
(list
  (list "Harvard" "Computer_Science" "GVI" "Hanspeter_Pfister")
  (list "Harvard" "Economics" "Macroeconomics" "Martin_Feldstein")
  (list "Harvard" "Government" "Michael_Sandel")
  (list "Harvard" "Computer_Science" "Theory" "Harry_Lewis")
  (list "Harvard" "Computer_Science" "Languages" "Greg_Morrisett")
  (list "Harvard" "Economics" "Amartya_Sen")
  (list "Harvard" "Government" "Harvey_Mansfield")
  (list "Harvard" "Computer_Science" "SYRAH" "Radhika_Nagpal")
  (list "Harvard" "Computer_Science" "SYRAH" "Matt_Welsh")
  (list "Harvard" "Economics" "Macroeconomics" "Greg_Mankiw")
  (list "Harvard" "Mathematics" "Noam_Elkies")
  (list "Harvard" "Computer_Science" "GVI" "Steven_Gortler")
  (list "Harvard" "Computer_Science" "Theory" "Michael_Mitzenmacher")
  (list "Harvard" "Computer_Science" "Theory" "Michael_Rabin")
  (list "Harvard" "History" "Drew_Gilpin_Faust")
  (list "Harvard" "Computer_Science" "SYRAH" "Margo_Seltzer"))

(define professors-tree (construct-n-tree "Harvard" professors))
```

Your task is to search `professors-tree` for a particular professor. To do this, write a function, `n-tree-search`, that takes a key and an n-tree. If the key is in the n-tree, you should return the path from the root node; otherwise, return false. For example

```
> (n-tree-search "Greg_Morrisett" professors-tree)
=> (list "Harvard" "Computer_Science"
        "Languages" "Greg_Morrisett")
> (n-tree-search "Larry_Summers" professors-tree)
=> false
```

You may assume that each key appears at most once in the tree. Don't forget to add tests!

5 Conclusion

(1 point total)

Exercise 3.

1 point

Please tell us how much time you spent on this problem set! The way we'd like you to do this is by defining a variable `minutes-spent` to be equal to the approximate number of minutes you spent working on this, from start to finish. Please don't forget this, or we'll think you finished in negative time, and students that fast clearly need more work. (But in seriousness, be honest - this is confidential and important for us to tune future problem sets.)