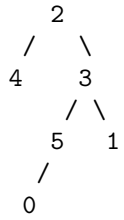
CS 61A Summer 2009 Homework 3

Topic: Tree and OOP

Lectures: Monday 7/6, Tuesday 7/7, Wednesday 7/8, Thursday 7/9

Reading: OOP: Above the Line Notes

1. (Graded by Corectness) Write a procedure `(tree-member? elem tree)` that takes in an element and a Tree (the abstract data type described in lecture) and checks whether the input element appears in the input Tree as datum anywhere. For instance, on the Tree represented below, call it `input-tree`:



`(tree-member? 3 input-tree)` would return `#t` while `(tree-member? 6 input-tree)` would return `#f`.

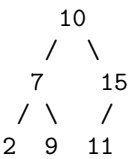
2. Using the `tree-member?` procedure, define a procedure `(ancestor? super sub tree)` that takes two elements and a Tree and checks to see if there exist a sub-tree whose datum is `super` and contains `sub`. For example, in the tree given as in the previous exercise, 2 is an ancestor of every datum in the tree, 3 is an ancester of 3,5,1,0, etc.

3. Write a procedure `shortest-length` that takes a Tree and checks to see what is the length of the shortest path that goes from the root to the leaves. For example, in the example Tree from the `tree-member?` problem, the `shortest-length` should return 2 since the path $(2 \rightarrow 4)$ is the shortest to a leaf and has length 2.

4. So far, we can find the shortest length from the root to a leaf in a Tree. Now we want to return the actual shortest path. Define a procedure `shortest-path` that takes a Tree and returns any one of the shortest path from the root to a leaf represented as a list. For instance, in the example Tree from the `tree-member?` problem, `shortest-path` might return `'(2 4)`.

5. A Binary Tree a Tree where every sub-Tree has at most 2 children. A Binary Search Tree is a binary Tree where in every single sub-Tree, every datum in the left sub-sub-Tree is smaller than the sub-Tree-root datum and every datum in the right sub-sub-Tree is bigger than the sub-Tree-root datum.

For example:



is a Binary Search Tree while

```
    /  \
   7    15
  / \    \
 2  11    12
```

Is NOT because 11 is on the left of 10 but is bigger and 12 is on the right of 15 but smaller.

Binary Trees are especially important in computer science because of Binary Search Tree. They are so special that we will treat them as a separate ADT.

1. Constructor: `(make-bt datum left right)`
2. Selectors: `(datum btree)`, `(left-child btree), (right-child btree)`

Now, write a procedure `(bst-insert val bst)` that takes a number and a Binary Search Tree and returns a new Binary Search Tree with the input number inserted into the Tree at the right position. Be careful that the tree you return is still a Binary Search Tree!

For example: inserting 7 into the following Binary Search Tree:

```
    8
   / \
  6  10
 /
4
```

should return

```
    8
   / \
  6  10
 / \
4  7
```

6. Write a procedure `(find val btree)` that returns `#t` if the input Binary Search Tree contains input number. Notice that you DO NOT and SHOULD NOT look through every single element in the Binary Search Tree because of the Binary Search Tree property. Try to make your `find` procedure as efficient as possible.

Note: The following questions involve OOP. To use the OOP language you must first

```
(load "~cs61a/lib/obj.scm")
```

before using `define-class`, etc.

7. For a statistical project you need to compute lots of random numbers in various ranges. (Recall that `(random 10)` returns a random number between 0 and 9.) Also, you need to keep track of *how many* random numbers are computed in each range. You decide to use object-oriented programming. Objects of the class `random-generator` will accept two messages. The message `number` means "give me a random number in your range" while `count` means "how many `number` requests have you had?" The class has an instantiation argument that specifies the range of random numbers for this object, so

```
(define r10 (instantiate random-generator 10))
```

will create an object such that `(ask r10 'number)` will return a random number between 0 and 9, while `(ask r10 'count)` will return the number of random numbers `r10` has created.

8. Define the class `coke-machine`. The instantiation arguments for a `coke-machine` are the number of Cokes that can fit in the machine and the price (in cents) of a Coke:

```
(define my-machine (instantiate coke-machine 80 70))
```

creates a machine that can hold 80 Cokes and will sell them for 70 cents each. The machine is initially empty. `Coke-machine` objects must accept the following messages:

`(ask my-machine 'deposit 25)` means deposit 25 cents. You can deposit several coins and the machine should remember the total.

`(ask my-machine 'coke)` means push the button for a Coke. This either gives a `Not enough money` or `Machine empty` error message or returns the amount of change you get.

`(ask my-machine 'fill 60)` means add 60 Cokes to the machine.

Here's an example:

```
(ask my-machine 'fill 60)
(ask my-machine 'deposit 25)
(ask my-machine 'coke)
NOT ENOUGH MONEY
(ask my-machine 'deposit 25)      ;; Now there's 50 cents in there.
(ask my-machine 'deposit 25)      ;; Now there's 75 cents.
(ask my-machine 'coke)
5                                 ;; return val is 5 cents change.
```

You may assume that the machine has an infinite supply of change.

9. We want to promote politeness among our objects. Write a class `miss-manners` that takes an object as its instantiation argument. The new `miss-manners` object should accept only one message, namely `please`. The arguments to the `please` message should be, first, a message understood by the original object, and second, an argument to that message. **(Assume that all messages to the original object require exactly one additional argument.)** Here is an example using the `person` class from the upcoming adventure game project:

```
> (define BH (instantiate person 'Brian BH-office))

> (ask BH 'go 'down)
BRIAN MOVED FROM BH-OFFICE TO SODA

> (define fussy-BH (instantiate miss-manners BH))

> (ask fussy-BH 'go 'east)
ERROR: NO METHOD GO

> (ask fussy-BH 'please 'go 'east)
BRIAN MOVED FROM SODA TO PSL
```