Lab 12: Binary trees

Create a separate file for each question. Keep them in your “Labs” folder, with the name $l_i j q_k$ for Lab $i j$, Question $k$.

Download the headers for each function from the file $l_{12}$-interface.rkt.

After you have completed a question (except class exercises), including creating tests for it, you can obtain feedback by submitting it and requesting a public test. Follow the instructions given in the Style Guide.

This lab makes use of the following data definitions:

```scheme
(define-struct binode (op arg1 arg2))
;;;; A binary arithmetic expression internal node (BINode) is a
;;;; (make-binode (anyof ‘* ‘+ ‘/ ‘-) BinExp BinExp)

;;;; A binary arithmetic expression (BinExp) is one of:
;;;; * a Num
;;;; * a BINode
(define-struct node (key val left right))
;;;; A Node is a (make-node Nat Str BT BT)

;;;; A binary tree (BT) is either
;;;; * empty, or
;;;; * (make-node Nat Str BT BT)

;;;; A binary search tree (BST) is a BT which additionally satisfies
;;;; the ordering property recursively:
;;;; * every key in left is less than key
;;;; * every key in right is greater than key
```

Language level: Intermediate Student with lambda

1. [Class exercise with lab instructor assistance]

The nodes of a binary tree can be traversed (“visited”) in many different orders, including:

- In-order traversal: visits the root key in between the keys of the left and then right subtree.
- Pre-order traversal: visits the root key before (thus “pre”) the left and right subtrees.
- Post-order traversal: visits the root key after (thus “post”) the left and right subtrees.

Write a function `preorder` that consumes a BT, $bt$, and creates a list of the keys in $bt$ following the pre-order traversal.

For example:

```scheme
(preorder (make-node 1 "a" (make-node 2 "b" empty empty) (make-node 3 "c" empty empty)))
```

=> (list 1 2 3)
2. Create a function `count-ops` that consumes a binary arithmetic expression, `bin-exp`, and produces the number of operations in that expression.

   For example:
   
   `(count-ops (make-binode '+ 4 (make-binode '/
                  (make-binode '+ 8 3) 2)))` => 3

3. Create a function `odd-length` that consumes a BT, `bt`, and produces a new version of `bt` where each `Node` whose value is an odd length string, has its value replaced with the string “odd”.

   For example:
   
   `(odd-length
     (make-node 4 "four"
       (make-node 7 "seven"
         empty
         (make-node 2 "two" empty empty))
     (make-node 32 "thirty-two"
       empty
       empty))` => (make-node 4 "four"
   (make-node 7 "odd"
     empty
     (make-node 2 "odd" empty empty))
   (make-node 32 "thirty-two"
     empty
     empty))

4. Create a function `is-ancestor?` that consumes a BST, `bst`, and two keys `k1` and `k2`, and produces `true` if the `Node` with the key `k1` is an ancestor of a `Node` with the key `k2`. The function produces `false` otherwise. A `Node` (node-a) is an `ancestor` of another `Node` (node-b) if node-a is node-b (a `Node` is its own ancestor), or if node-b can be found in either the left or right subtrees of node-a. Be sure to use the ordering property of a BST where appropriate.

   For example:
   
   `(is-ancestor?
     (make-node 6 ""
       (make-node 1 "" empty empty)
     (make-node 13 ""
       empty
     (make-node 17 "" empty empty)))
   6 17)` => `true`
5. Create a function \textit{leftmost} that consumes a non-empty BST, \textit{bst}, and produces the leftmost node in the tree.

For example:
\begin{verbatim}
(leftmost
    (make-node 5 "five"
        (make-node 1 "one" empty empty)
        (make-node 6 "six"
            empty
            (make-node 9 "nine" empty empty)))
=> (make-node 1 "one" empty empty)
\end{verbatim}

6. \textit{Practice question (not to be submitted)}

Consider how to change the code from q1 to produce the in-order and post-order traversals. For a BST, the in-order traversal has a special property. This does not require a large change to q1.

For example (given the tree from q1):
\begin{verbatim}
(inorder q1-tree) => (list 2 1 3)
(postorder q1-tree) => (list 2 3 1)
\end{verbatim}

Optional open-ended questions

Consider the operation \textit{remove}, for a BST. Write a function \textit{remove}, that consumes a BST, \textit{t}, and a number, \textit{k}, and produces a new BST like \textit{t}, but with the key \textit{k} and its associated value removed. The goal is to keep as much of the new tree similar to the original tree as possible. See \textit{here} for hints.

Helpful tips

Writing and reading trees in Racket

If you format your trees as they are above, they are easier to read. Consider: if one follows along a certain level of indentation they will read through a “row” of the tree in order from left to right. If the tree has no empty, the final row will be the list of leaves.