COMP 333
BST Programming Project Part 1

BST (Binary Search Tree) Data Structure
In Java, class defs for BST might look like this:

```
class Node {
        int value;          // value maintained by node
        int count;          // number of insertions of that value (only store one node)
        Node left;          // node to left with left value < value (can be null)
        Node right;         // node to right with right value > value (can be null)
}
class BST {
        Node root;          // node at root of tree (can be null)
}
```

Class Node is the basic structure that describes the nodes that make up the BST. Class BST is the enclosing structure that manages the reference to the root node. Other fields can be added but this is a representative starting point.

In Racket, the structs corresponding to these classes might look like this:

```
(struct node (value count left right))
(struct bst (root))
```

We will also use the "mutable" and "transparent" attributes for these structures so that we can modify values after they have been constructed.

BST Operations

- Constructor: (bst null)
    - Initialize root to null
- Find: (find t v)
    - Generates path (list of nodes) in BST t from root to insertion point for value v.
    - If BST is empty (root of t is null) return null. Otherwise, add root to a new list of nodes. Then move downward through the tree recursively moving left or right to look for node with value v. Stop when either the node with value v is found. If node with value v is not found, stop at node that would be parent node of v (called the insertion point).
    - Organize path so that root is at the tail, and node or insertion point for node is at the head. In other words, lower nodes in the tree are appended to the head of the list, while higher nodes are placed deeper toward the tail of the list.

- Insert: (insert t v)
    - Add a new value v to existing BST t.
    - Start by calling (find t v) to get path (list of nodes) from root to node or insertion point.
    - If value of head of list equals v, then node with value v already exists in BST. In this case, don't insert a new node, increment the count field of existing node.
    - If value of head of list is not equal to v, then head of list is insertion point for new node with value v. Create a new node with value v and insert it as left or right child node of the insertion point.
    - Inserting value v into an initially empty BST is a special case. In this case, don't call find, instead create a new node and make it the root.
- Traverse: (traverse t)
    - Use recursive definition for in-order traversal to create a list of nodes. Note that a list of nodes is not the same as a BST. Traverse creates a list of references to all the nodes in the BST without modifying the BST.
    - Recursive definition for (traverse n)
        - If n is null, result is null
        - Else null is the append of traverse of (node-left n), (list n), and traverse of (node-right n).
    - Traversal of entire tree is obtained by starting traverse at the root.
- Delete: (delete t v)
    - Delete node in BST t with value v.
    - If root of t is empty, do nothing.
    - If root of t is not empty, use (find t v) to get path from root to node or insertion point.
    - If value of head of path is not equal to v, then node to delete is not in tree, do nothing.
    - If value of head of path equals v and count > 1, then don't delete the node, just decrement the count of the node.
    - If head of path equals v and count is 1, then delete the node. How to delete the node is broken down into cases and subcases, based on how many child nodes the node to delete has.
        - Child count is 0
        - Child count is 1
        - Child count is 2
    - Cases will be covered in detail in a later document.
    - Note that deleting the last node in a BST is a special case since it causes the root to change to null.

- Convert “List of Nodes” to “List of Values”: (nl-to-vl x)
    - To create a simple display of the values in a BST, we will do it in two steps
        - First, we use (traverse t) to generate a list of nodes from the BST.
        - Then we will use (nl-to-vl x) where x is a list of nodes we obtained from (traverse t). This function will convert a list of nodes into a list of values.
    - Typical usage will look like this:
        - (define b1 (bst null))   ; b1 is an initially empty BST
        - (insert b1 50)   ; insert values into b1
        - (insert b1 25)
        - (insert b1 75)
        - …
        - (display (nl-to-vl (traverse b1)))   ; display values in b1

Test Support Operations

A common way to test BST operations is to generate a list of random numbers then insert them into or delete them from a BST.

- Generate Random Number: (random-value r)
    - Generate an integer value on the range 0 through r-1
    - Use the built-in function (random) which generates a random value between 0 and 1. Multiply it by r to scale it, then use (floor x) and (inexact->exact x) to convert it to an integer.
- Generate List of N Random Numbers: (random-list n r)
    - Use (random-value r) to generate a list of n numbers

Later we will write additional test support functions to do aggregate insert and delete.

Find: (find t v)
The (find t v) function is an internal function that will be used by both insert in delete. (find t v) takes two arguments, t for the BST, a value v that represents a node value we are looking for in the tree (may or may not be there).

The result or value of find is a list of nodes that define a path from the root to the node we were looking for (if v is present in the tree), or the insertion point (if v is not present).

(find t v) does not make any changes to the tree, it simply examines it and returns a list of nodes that define a path through the tree for use by other functions.

```
(define (find t v)
 (cond
       …                           ; cases go here
   ))
```

Cases to be added to the conditional:

```
If t is empty, the result is the empty list
else if value of root of t == v, then the result is a list containing root
else if v < value and left of root is empty, then result is list containing root
else if v < value and left of root is not empty
    then recursive call on left of root, append list root to the head of recursive call
else if v > value and right is empty, then result is list containing root
else if v > value and right of root is not empty
    then recursive call on right of root, append list root to the head of recursive call
```

The result will be a list of nodes that show the path between root of tree and destination node. We have to decide if the root is at the head of the list or at the tail. I suggest we put the root at the tail of the list and more recently visited nodes at the head. If you implemented a similar operation in Java, you would likely use a stack of nodes rather than a list.

Implementation of (find t v)
We can better organize the cases if we use a top-level function "find" to take care of special cases, and let it call a helper function "find-node" for the non-special cases. The reason this is an improvement is because the "find-node" function can more easily be written in a recursive style.

Option #1: (find-node n v stack)
Create a new parameter named "stack" that "find" will pass to "find-node" the helper function. This will be a list of nodes from root to insertion point. Initial value of "stack" is (list (bst-root t)), a list containing just the root of the tree.

- In the helper function, when the base cases are reached, "(cons n stack)" is the final value.
- For other cases, call "stack-node" function recursively with n replaced by left of n or right of n, and "stack" replaced by (cons n stack).

```
(define (find t v)                              ; top-level function
  (cond
    ((empty? (bst-root t)) null)                ; special case tree is empty
    (else (find-node (bst-root t) v null))      ; general case let find-node do the work
                                                ; initial value of stack is null
    ))

(define (find-node n v stack)                   ; helper function
  (cond
          …                                     ; cases go here
    ))
```

Option #2: (find-node n v)
Don't use extra parameter "stack".

- In the base case for "find-node", the result is (list n).
- In the recursive case, result is the append of recursive call with (list n). Be sure to put recursive call on the left and (list n) on the right, because we want the root at the tail of the list and the insertion point at the head.

```
(define (find t v)                              ; top-level function
  (cond
    ((empty? (bst-root t)) null)                ; special case tree is empty
    (else (find-node (bst-root t) v))           ; general case let find-node do the work
                                                ; no parameter "stack" in option #2
    ))

(define (find-node n v)                         ; helper function
  (cond
          …                                     ; cases go here
    ))
```

Brief Discussion of Implementation Options for Find

In Option #1, the approach for constructing the solution is to use a variable named “stack” which plays the role of an accumulator. This means that the variable’s value is modified over the course of the recursive calls and the final result is “accumulated” into the variable. During recursion, the value returned by the recursive call as the result expression for a recursive step is not important, and it is not used as a partial solution. Instead, the recursive call only needs to update the value of the accumulator value, which is more important for expressing the result when one of the base cases is reached.

In Option #2, there is no accumulator variable. Instead, the solution does use the result expression of the recursive call as a partial solution. In the recursive steps, rather than simply call the function recursively with an updated accumulator parameter value, the recursive call’s result expression is nested inside a follow on computation to compute the complete result expression.

Hard to say if one of these options is better or simpler than the other. Both are examples of general techniques for writing recursive functions. Try both and choose whichever one you think is more natural or intuitive.

Convert list of nodes to list of values: (nl-to-vl x)
When printing out results, it will be convenient to convert a list of nodes into a list of values. When nodes are displayed by the interpreter, each node is displayed along with any nested nodes that are connected to it, which is correct but hard to read on the screen. If we just want to check that the nodes are in the correct order, it's convenient to display a list of values of the nodes in the list, rather than the actual list of nodes. This function is merely for formatting convenience to print information out and display it.

```
(define (nl-to-vl x)
  (cond
        …                       ; base case, recursive case
   ))
```

```
If x only contains one node
    Take the value of the node and create a list of one item
Else
    Take the value of the head node, cons it to the result of the recursive call to (cdr x)
```

Traverse: (traverse t)
This function implements the recursive definition of BST in-order traversal described earlier. Its result is a list of nodes with the node values in ascending order. We will use the same trick we used for (find t v) and divide it into a top-level function and a recursive helper function.

```
(define (traverse t)            ; top-level function
  (traverse-node (bst-root t))  ; "traverse" calls "traverse-node" to do all the work
  )

(define (traverse-node n)       ; recursive helper function
  (cond
        …                       ; base case, recursive case
   ))
```

The top-level function "(traverse t)" will simply call the helper function "(traverse-node (bst-root t))" passing the root node as its parameter. In this way, the helper function can be simplified and written as a recursive function using only a node reference, without having to handle special cases involving the root of the BST structure.
For reference, here is the recursive definition of traverse that we will use for the traverse-node function.

```
If n is empty
    Return the empty list
Else
    Return the append of (1) traverse on left of n, (2) list of n, and (3) traverse of right of n
```

Brief Discussion of Implementation of Traverse

The reason for breaking up the definition of traverse into top-level and helper functions is related to the difference between interface and implementation in OOP.

The **interface** is the subset of the code that is exposed to the client. In OOP, these are the functions declared **public**. The **implementation** is the rest of the code that only used internally to complete the requirements of the application. In OOP, these functions are declared **private**.

If the client wants to perform a traverse, then the interface should be made simple and straightforward:

```
(define b (bst null))       ; create bst named b
…                           ; series of inserts/deletes/etc
(traverse b)                ; "good" interface definition for traverse, based on b
```

In this example, the traverse function accepts one parameter of type bst. This interface is consistent with the client's view of the structure.

But the implementer will notice that the traverse function is easier to implement if the parameter is of type node rather than bst. It is possible and not difficult to obtain the root node from the bst reference.

```
(bst-root b)
```

But we don't want the client to have to know implementation details of the bst in order to call the traverse function

```
(traverse (bst-root b))
```

Therefore it is often convenient to organize the code into a top-level function with the needs of the client in mind, which calls a helper function with extra information plugged in as needed by the implementation.

```
(define (traverse t) (traverse-node (bst-root t)))       ; interface used by the client

(define (traverse-node n)                                ; implementation used only internally
    (cond
        …
    )
)
```

Recap

After completing this exercise, you should have completed the following functions that work on the provided test cases.

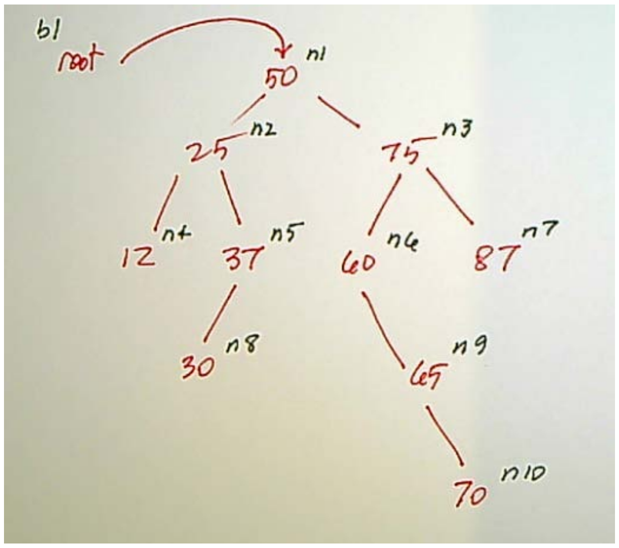
```
(struct node … )               ; struct to define a single node
(struct bst …)                 ; struct to define a BST using root to reference node at root

(define (find t v) … )         ; result is list of nodes along path from root to node with value v
(define (traverse t) …)        ; result is list of all nodes in BST t
(define (nl-to-vl x) … )       ; argument is list of nodes, result is a list of values of those nodes
```

Testing
When first starting to code a project, it is hard to find good test cases because not all of the needed functions are completed. For example, for this project we have not yet discussed how to implement insert and delete. Without an insert function, it will be harder to create a tree to use as a test case for simpler functions we are implementing first (traverse, find, and nl-to-vl).

But there is a simple fix. For now, we can build a few BSTs manually using only the autogen functions like the node constructor. Once we have a manually built tree, we can use it to unit test our first set of functions. Note that this is only a temporary measure to support testing. After we implement insert in the next section, we won't need to build trees manually.

How to Build a BST Manually
First, create some random numbers, then draw a sketch of the resulting BST. Don't use too many nodes, here is one with 10 nodes. Each node has been labeled with the symbolic name n1 through n10.



Now call the "node" constructor, and bind the result to a named symbol. Note that we need to build the nodes **from the bottom up** so that names are defined before we use them as links in another node.

```
(define n10 (node 70 1 null null))
(define n09 (node 65 1 null n10))
(define n08 (node 30 1 null null))
(define n07 (node 87 1 null null))
(define n06 (node 60 1 null n09))
(define n05 (node 37 1 n08 null))
(define n04 (node 12 1 null null))
(define n03 (node 75 1 n06 n07))
(define n02 (node 25 1 n04 n05))
(define n01 (node 50 1 n02 n03))
```

Finally, create a new BST with root at node n01.

```
(define b01 (bst n01))
```

You can try some simple test cases using the autogen functions based on the nodes you just defined.

```
(node? n10)                 ; test for membership for node n10
(node-value n10)            ; access of value field of node n10
(node-count n08)            ; access of count field of node n08
(bst? b01)                  ; test for bst membership for b01
(empty? (bst-root b01)) ; check if root of b01 is empty
```

Find Tests

```
(find b01 70)
(find b01 50)
```

Better to use nl-to-vl to convert result of find from list of nodes to list of values

```
(nl-to-vl (find b01 70))
(nl-to-vl (find b01 50))
```

Traverse Tests

```
(traverse b01)
```

Similarly, it's better convert result of traverse to list of values to view the results:

```
(nl-to-vl (traverse b01))
```

Using Map for Aggregate Tests

Let's apply find to every value in the tree b01. We can get a list of all values with

```
(nl-to-vl (traverse b01))
```

Then we can use map to use find to get the path from root to each value

```
(map (lambda (x) (find b01 x)) y)
```

where y is the list of values. Substituting the (nl-to-vl …) expression for y we get

```
(map (lambda (x) (find b01 x)) (nl-to-vl (traverse b01)))
```

(map …) will take each value one at a time from y and pass it as x to the lambda expression.

Finally we should also apply nl-to-vl to the result of find

```
(map (lambda (x) (nl-to-vl (find b01 x))) (nl-to-vl (traverse b01)))
```

Compare the path for each value to the sketch we created above to confirm the path is correct. For example, the path for (find b01 70) is (70 65 60 75 50), where 70 is the head (left end) of the list and 50 is the tail (right end) of the list.

Example Test Results

```
> (find b01 70)
             (list
             (node 70 1 '() '())
             (node 65 1 '() (node 70 1 '() '()))
             (node 60 1 '() (node 65 1 '() (node 70 1 '() '())))
             (node 75 1 (node 60 1 '() (node 65 1 '() (node 70 1 '() '()))) (node 87 1 '() '()))
             (node
              50
              1
              (node 25 1 (node 12 1 '() '()) (node 37 1 (node 30 1 '() '()) '()))
              (node 75 1 (node 60 1 '() (node 65 1 '() (node 70 1 '() '()))) (node 87 1 '() '()))))
> (find b01 50)
            (list
             (node
              50
              1
              (node 25 1 (node 12 1 '() '()) (node 37 1 (node 30 1 '() '()) '()))
              (node 75 1 (node 60 1 '() (node 65 1 '() (node 70 1 '() '()))) (node 87 1 '() '()))))
```

In these examples, (find …) returns a list of nodes displayed in a nested format that is difficult to read.

```
> (nl-to-vl (find b01 70))
            '(70 65 60 75 50)
 > (nl-to-vl (find b01 50))
            '(50)
```

If instead we use (nl-to-vl …) to convert a list of nodes to a list of values, the list of values is easier to read when displayed by the interpreter.

```
> (traverse b01)
            (list
             (node 12 1 '() '())
             (node 25 1 (node 12 1 '() '()) (node 37 1 (node 30 1 '() '()) '()))
             (node 30 1 '() '())
             (node 37 1 (node 30 1 '() '()) '())
             (node
              50
              1
              (node 25 1 (node 12 1 '() '()) (node 37 1 (node 30 1 '() '()) '()))
              (node 75 1 (node 60 1 '() (node 65 1 '() (node 70 1 '() '()))) (node 87 1 '() '())))
             (node 60 1 '() (node 65 1 '() (node 70 1 '() '())))
             (node 65 1 '() (node 70 1 '() '()))
             (node 70 1 '() '())
             (node 75 1 (node 60 1 '() (node 65 1 '() (node 70 1 '() '()))) (node 87 1 '() '()))
             (node 87 1 '() '()))
> (nl-to-vl (traverse b01))
            '(12 25 30 37 50 60 65 70 75 87)
```

Similarly with traverse, it's better to convert the result of traverse from a list of nodes to a list of values.

```
> (nl-to-vl (traverse b01))  ; list of values to pass to map
            '(12 25 30 37 50 60 65 70 75 87)
> (map (lambda (x) (nl-to-vl (find b01 x))) (nl-to-vl (traverse b01))) ; run lambda on list of values
            '((12 25 50) (25 50) (30 37 25 50) (37 25 50) (50) (60 75 50) (65 60 75 50) (70 65 60 75 50) (75 50)
            (87 75 50))
```

Finally, once we have composed an expression we want to test for multiple inputs, we can use map by giving it the test expression and a list of inputs to run it on. For this last output, each element of the list returned by map is a path from root to the corresponding value in the tree.

Study each path by comparing it to the sketch of the tree above. For example, the path for value 30 is (30 37 25 50). Make sure you recognize the path of nodes that is traced starting at the root 50 in order to reach 30. Also note that 30 is at the head of the list and 50 is at the tail.

Next: In Parts 2 and 3, we'll implement insert and delete functions (with many test cases) to complete the project.