COMP 333
BST Programming Project Part 2

Random Numbers: (random-value r) and (random-list n r)
We will need to generate random numbers so that we can use them to create larger nontrivial BSTs for testing. Racket provides a built-in function (random) that creates a random number between 0 and 1. We will start with this function, then do some scaling and type conversion to get an integer random number on a specific range. Finally we can create a list of n random numbers all on the same range. Here are a few built-in functions we can use:

```
(random)            ; generates an inexact (floating point) random value between 0 and 1
(floor x)           ; takes the floor of an inexact number
(inexact->exact x)  ; converts an inexact to exact (integer)
```

The first function (random-value r) uses r as an integer scale factor to return a random integer on the range 0 through r-1.

```
> (random-value 10)      2
> (random-value 10)      9
> (random-value 100)     15
> (random-value 100)     99
```

This function is not recursive, it is simply an expression to call (random) scale it, floor it, and convert it to an exact number.

The second function (random-list n r) uses (random-value r) to create a list of n numbers.

```
> (random-list 10 100)      '(38 82 70 10 47 30 60 97 69 34)
> (random-list 5 1000)      '(821 630 81 995 679)
```

This is a simple recursive function with recursion based on n. If n=1, then generate a list of one random number. Otherwise generate a random number, and append it to the result of the recursive call with n replaced with n-1.

Testing with Random Numbers

Using random numbers for testing BST is a big convenience. When the code is completed, each test can run with input based on a newly generated set of random numbers. Over time, the accumulation of many tests with each test using a different list of numbers and resulting in correct output, your confidence that the code is correct will improve.

In the early stages of testing, before functions are shown to be correct, you will need to use the random numbers in a different way. If an error is found for a particular set of random inputs, it is important to **capture** and **preserve** that specific list of random numbers, and temporarily change the testing strategy to repeatedly test with that exact same list of numbers, until the error is corrected. Once the error is corrected, switch back to running multiple tests with a different set of random inputs for each test.

So at the beginning of every random number test, begin by displaying the list of random numbers that were generated for the test. If there are no errors found during the test, you can discard that set of random numbers and move to the next set. But if an error was found, you will at least temporarily need to preserve that specific set of random numbers and run the same test repeatedly with the exact same list of numbers. In other words, that specific set of numbers has exposed an error that was not triggered by other inputs. Keep testing with the set of numbers that trigger the error until the error is corrected.

Review of Autogen Functions for (struct bst …) and (struct node …)
You will need to use the autogen functions of the structures to complete the code for BST:

```
(bst x)              ; constructor, create bst with root x
(bst? x)             ; test, is x bst?
(bst-root x)         ; get root of bst x
(set-bst-root! x y)  ; set root of bst x to y

(node x y z w)       ; constructor, create node with x y z w for value, count, left, right
(node? x)            ; test, is x a node?
(node-value x)       ; get value of node x
(node-count x)       ; get count of node x
(node-left x)        ; get left of node x)
(node-right x)       ; get right of node x)
(set-node-value! x y)  ; set value of node x to y
(set-node-count! x y)  ; set count of node x to y
(set-node-left! x y)   ; set left of node x to y
(set-node-right! x y)  ; set right of node x to y
```

Use these functions as needed in the BST functions whenever you need to manage the contents of bst and node structures. It is probably not worth the trouble to memorize the names of these functions, it would be a better use of your time to note the pattern of how the autogen function assigns names based on the name of the structure and the names of its fields.
Hint: in Java, the notation for modifying the value field of node n might look like this:

```
    n.value = 3;
or  n.setValue(3);
```

But in Racket, the same expression will look like this:

```
(set-node-value! n 3)
```

where set-node-value! is the mutator, n is the node, and 3 is the new value for the value field.

Another example: increment the count value of a node n

```
Java:  n.count++;
or     n.setCount(n.getCount()+1);

Racket: (set-node-count! n (+ 1 (node-count n)))
```

First, access the current value of count with (node-count n). Then add 1 to it with (+ 1 (node-count n)). Finally use the mutator to make this value the new updated value of count with (set-node-count! n (+ 1 (node-count n)))

BST Functions to Complete

Insert: (insert t v)
This function will insert a value v into an existing BST t. We will write a top-level function (insert t v) and a helper function (insert-node v s).

```
(define (insert t v)                ; top-level function, t is BST, v is value to insert
        (cond
                …                                   ; handle special case if BST is currently empty
                (else (insert-node v (find t v)))   ; use find and call insert-node
        ))

(define (insert-node v path)        ; v is value to insert, path is list of nodes to insertion point
        (cond
                …
        ))
```

Special Cases
For insertion, assuming we are not performing any rebalancing of imbalanced trees, the only situation in which the root changes is on insertion of the first node into previously empty tree. In this case, we must create a new node with value v and set that node to be the root of the tree. This special case is handled by the top-level (insert t v) function.

If the tree is not empty, then we call the (find t v) function to get the path from root to insertion point. We pass the value of path to the helper function (insert-node v path) and let it handle the rest of the cases. This function is simplified since we have already handled the case of the empty tree, which is the only case where the root changes. The helper function can then focus just on changes to links between nodes.

All the cases are based on the head of the path variable. Insertion point will be found at

```
(car path)
```

The variable “path” contains the list of nodes returned earlier by “(find t v)”. The first node in path will be either

(1) a node with value v (if v has been inserted earlier), or
(2) the insertion point (if v has not been inserted previously).

There are three cases to check for:

- v = value of (car path): node with value v at (car path) already inserted
- v < value of (car path): new node with value v becomes left child of (car path)
- v > value of (car path): new node with value v becomes right child of (car path)

In the first case, if v = value of (car path), then the value we are trying to insert has already been inserted previously. In this case we don't insert a duplicate node, we increment the count field of (car path).

In the second case, if v < value of (car path), then there is no node with value v in the tree. The (car path) node is then the insertion point. In this case we create a new node with value v and make it the left node of (car path).

In the third case, if v > value of (car path), then similarly, we create a new node with value v and set it to be the right node of (car path).

Note that (insert-node v path) is not a recursive function. The top-level function "insert" called (find t v), which has done all the work required to locate the insertion point. The only work left for (insert-node …) to do is manage the three cases described above.

Other Functions for Test Support

Suppose we want to start with an empty BST and then insert many values from a list. There are several tasks to complete:

Generate a list of random numbers:

```
> (define rlist-01 (random-list 10 1000))
> rlist-01                          '(175 811 858 203 797 113 439 378 515 734)
```

Create an initially empty BST

```
> (define bst-01 (bst null))
> bst-01                            (bst '())
```

Insert values one at a time from rlist-01 into bst-01. After each insertion, use traverse and nl-to-vl to show the status of the tree.

```
> (insert bst-01 (list-ref rlist-01 0))        ; insert element 0 from list into BST
> (display (nl-to-vl (traverse bst-01)))       ; display result
(175)

> (insert bst-01 (list-ref rlist-01 1))        ; insert element 1 from list into BST
> (display (nl-to-vl (traverse bst-01)))       ; display result
(175 811)
…                                              ; and so on for all elements in list
```

But it would be easier to get map to do this for us:

```
> (map (lambda (x) (insert bst-01 x)(displayln (nl-to-vl (traverse bst-01)))) rlist-01)
(175)
(175 811)
(175 811 858)
(175 203 811 858)
(175 203 797 811 858)
(113 175 203 797 811 858)
(113 175 203 439 797 811 858)
(113 175 203 378 439 797 811 858)
(113 175 203 378 439 515 797 811 858)
(113 175 203 378 439 515 734 797 811 858)
'(#<void> #<void> #<void> #<void> #<void> #<void> #<void> #<void> #<void> #<void>)
>
```

We can turn this map expression into a function **(insert-from-list t y)** where t is an initially empty BST and y is a list of random numbers we want to insert. We will also set the final value of the map expression to (void) to suppress the list of void expressions at the end.

Add this definition to your code:

```
(define (insert-from-list t y)
   (map (lambda (x) (insert t x) (displayln (nl-to-vl (traverse t)))) y) (void)
)
```

The lambda does two things: (1) inserts a value into the BST, and (2) displays the BST. Once the map operation is complete, by default, map returns a list of results. But since the last function in the lambda is (displayln …) and the result of displayln is void, we get a list of void expressions as the return value from map. Since this list is not needed or helpful, we can suppress it by adding a final expression (void) to map that replaces the list of voids with (void) as the result of map.

Starting from an empty BST bst-01 and a list of random numbers rlist-01, here is the insert test case:

```
> (insert-from-list bst-01 rlist-01)
(175)
(175 811)
(175 811 858)
(175 203 811 858)
(175 203 797 811 858)
(113 175 203 797 811 858)
(113 175 203 439 797 811 858)
(113 175 203 378 439 797 811 858)
(113 175 203 378 439 515 797 811 858)
(113 175 203 378 439 515 734 797 811 858)
```

Once we implement delete, we will combine insert and delete tests to build a list starting from an empty tree, then tear it down and return to the empty tree starting point. We'll do that in Part 3.

Test Case 1 (Required)

Run the following test to confirm your code for Part 2 is complete:

```
(define btree (bst null))              ; create new bst named btree initialized to empty
btree                                  ; display its contents
(define rlist (random-list 15 1000))   ; create a random list rlist
rlist                                  ; display its contents
(insert-from-list btree rlist)         ; insert all values from rlist into btree
```

Here's an example trace of running the test. Note that each test will generate a different list of random numbers. But the the display of the traversal of the tree after each insertion should show the values inserted so far in sorted order.

> (define btree (bst null))
> btree
(bst '())
> (define rlist (random-list 15 1000))
> rlist
'(794 840 402 146 544 830 26 93 975 293 342 57 273 493 542)
> (insert-from-list btree rlist)
(794)
(794 840)
(402 794 840)
(146 402 794 840)
(146 402 544 794 840)
(146 402 544 794 830 840)
(26 146 402 544 794 830 840)
(26 93 146 402 544 794 830 840)
(26 93 146 402 544 794 830 840 975)
(26 93 146 293 402 544 794 830 840 975)
(26 93 146 293 342 402 544 794 830 840 975)
(26 57 93 146 293 342 402 544 794 830 840 975)
(26 57 93 146 273 293 342 402 544 794 830 840 975)
(26 57 93 146 273 293 342 402 493 544 794 830 840 975)
(26 57 93 146 273 293 342 402 493 542 544 794 830 840 975)
>

Test Case 2 (Required)
Also run the following test case to confirm that your code handles insertion of duplicate values correctly.

> (define btree-2 (bst null))
> btree-2
(bst '())
> (define list-2 '(50 25 75 12 37 30 45 60 80 12 60))
> list-2
'(50 25 75 12 37 30 45 60 80 12 60)
> (insert-from-list btree-2 list-2)
(50)
(25 50)
(25 50 75)
(12 25 50 75)
(12 25 37 50 75)
(12 25 30 37 50 75)
(12 25 30 37 45 50 75)
(12 25 30 37 45 50 60 75)
(12 25 30 37 45 50 60 75 80)
(12 25 30 37 45 50 60 75 80) *; no new node added for duplicate value 12*
(12 25 30 37 45 50 60 75 80) *; no new node added for duplicate value 60*

You can see from the output that the insertions of the last two values 12 and 60 do not grow the list of nodes, since the values are duplicates and do not cause additional nodes to be added to the tree.

To get a more explicit confirmation that duplicate values are being handled correctly, write a modified version of nl-to-vl called **nl-to-vcl** which shows both the value and the count field for each node in the list. Then use that function to display the contents of the tree with duplicate values.

> (nl-to-vcl (traverse btree-2))
'((12 2) (25 1) (30 1) (37 1) (45 1) (50 1) (60 2) (75 1) (80 1))

Now you can see explicitly that although we performed 11 insertions, we wind up with only 9 nodes, 7 of which have counts of 1, and two of which have counts of 2.

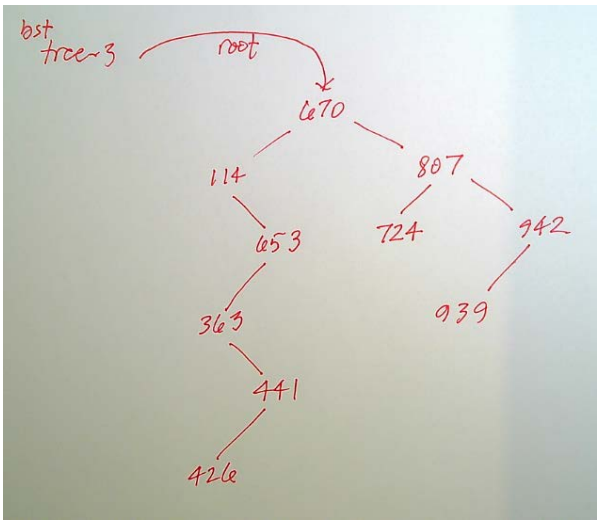
Test Case 3 (Optional)

The (find t v) function was implemented and tested in Part 1. Now that the (insert t v) function has been implemented, you can return to the find function and test it on any tree created with insert.

> (define tree-3 (bst null))
> tree-3 (bst '())
> (define list-3 (random-list 10 1000))
> list-3 '(670 114 807 653 363 942 939 724 441 426)
> (insert-from-list tree-3 list-3)

(670)
(114 670)
(114 670 807)
(114 653 670 807)
(114 363 653 670 807)
(114 363 653 670 807 942)
(114 363 653 670 807 939 942)
(114 363 653 670 724 807 939 942)
(114 363 441 653 670 724 807 939 942)
(114 363 426 441 653 670 724 807 939 942)

A sketch of the resulting bst tree-3 looks like this:



This is also a good example of how a general set of random numbers may result in a tree with its height greater than optimal, hence the need for rebalancing in the general case to keep the height of the tree at its minimal value.

Now use map and apply (find t v) to each value in the tree, and confirm that the indicated path from v to root is correct.

> (map (lambda (x) (displayln (nl-to-vl (find tree-3 x)))) list-3)

(670)
(114 670)
(807 670)
(653 114 670)
(363 653 114 670)
(942 807 670)
(939 942 807 670)
(724 807 670)
(441 363 653 114 670)
(426 441 363 653 114 670)
'(#<void> #<void> #<void> #<void> #<void> #<void> #<void> #<void> #<void> #<void>)

You can now check the path of each value from leaf to root to confirm that “find” creates the correct path for that value.

The list of void values which is the result of applying map repeatedly to a function that returns void can be suppressed by adding a (void) expression to a sequence after calling map:

> (begin (map (lambda (x) (displayln (nl-to-vl (find tree-3 x)))) list-3) (void))

(670)
(114 670)
(807 670)
(653 114 670)
(363 653 114 670)
(942 807 670)
(939 942 807 670)
(724 807 670)
(441 363 653 114 670)
(426 441 363 653 114 670)