Racket Programming Project #2
Binary Search Tree (BST) Insert, Find, Traverse, Delete

Use the "struct" feature of Racket to define a node structure for building an unbalanced binary search tree (BST). Add functions to insert, find, traverse (in-order), and delete.

(struct bst-node (value left right count) #:mutable #:transparent)
This defines a structure to represent a node in a BST

- value: numerical value stored at the node
- left, right: references to other bst-node objects in the tree, if any
- count: number of insertions into the tree for this value; this is an alternative to inserting multiple nodes with identical values; initialized to 1 for first insertion, incremented during subsequent adds of the same value; decremented during subsequent deletions, unless count is 1, in which case the node is physically removed from the tree

The struct has two attributes:

- #:mutable allows instances of bst-node to be modified, default is immutable
- #:transparent allows contents of bst-node object to be printed or displayed

In functional programming, structs are immutable by default. Functions with immutable inputs and outputs are easier to test and even prove correct. But for large immutable data, small updates lead to high resource usage. For example, to insert a new node into an existing immutable tree, create a copy of the original with update, then replace original with update, and discard original. When programming in a functional language, programmer occasionally must choose between a pure functional style with high resource usage, or a hybrid functional-imperative style that has more efficient resource usage.

Note that the definition of a struct introduces automatically defined constructor, accessors, mutators, and membership test. You don't have to do anything to define them, they are defined automatically as a side-effect of the struct definition for bst-node.

- (bst-node v l r c): constructor
- (bst-node-value n): value accessor
- (bst-node-left n): left accessor
- (bst-node-right n): right accessor
- (bst-node-count n): count accessor
- (set-bst-node-value! n v): value mutator
- (set-bst-node-left! n l): left mutator
- (set-bst-node-right! n r): right mutator
- (set-bst-node-count! n c): count mutator
- (bst-node? n): tests n for membership in Sbst-node type

In keeping with the type system of untyped Racket, fields are left untyped. However, the struct bst-node definition introduces bst-node as a user-defined type. Functions defined later that expect arguments of type bst-node will generate errors if the argument is not of the appropriate type.

Add a value to a BST

Insert a value v into a BST with root node tree. For duplicate values, maintain a max of one node per value. Use the count field to indicate multiple insertions of a single value, don't add multiple nodes. Single node per value greatly simplifies need to balance the tree later on (not in this project).

(define (add-value-to-bst tree v)
(add-value-to-bst-subtree tree tree v))

The first function simply calls the helper function and introduces a 3rd argument.

(define (add-value-to-bst-subtree tree subtree v)

Arguments

tree: reference to bst-node at the root of the original tree
subtree: reference to bst-node currently being checked as the insertion location
v: numerical value to insert

Value

Final value is a reference to the root of the original tree

Algorithm

There are 5 cases

v < value at subtree and left of subtree is empty	insert new bst-node on left
v < value at subtree and left of subtree is not empty	recursive call on l of subtree
v > value at subtree and right of subtree is empty	insert new bst-node on right
v > value at subtree and right of subtree is not empty	recursive call on r of subtree
v = value at subtree	increment count

Insertion cases will use the **bst-node** constructor to create the new node to insert, and the accessors and mutators for individual fields (to check for empty, to update to a new value, etc.) Recursive calls will preserve the function argument for original value of the root node but will set the subtree argument to the appropriate subtree (left or right) to continue the descent into the tree to find the insertion position.

Simple Testing with bst-node Objects

Before writing any functions, you can do some simple testing by just creating a few bst-node objects and linking them together. You will need to use the bst-node constructor, and possibly accessors and mutators (which you don't need to define since they are defined automatically).

```
(define n1 (bst-node 5 empty empty 1))          call constructor and bind to n1
(define n2 (bst-node 10 empty empty 1))         call constructor and bind to n2
(define n3 (bst-node 15 empty empty 1))         call constructor and bind to n3
(set-bst-node-left! n2 n1)                      call mutator to link n1 as left of n2
(set-bst-node-right! n2 n3)                     call mutator to link n3 as right of n2
n2
        (bst-node 10 (bst-node 5 '() '() 1) (bst-node 15 '() '() 1) 1)
 (get-bst-value-list-inorder n2)
        '(5 10 15)
```

The output for the "n2" expression displays all fields of the n2 node. Note that the left and right field values of bst-node n2 are bound to bst-nodes n1 and n3. The display of the n2 node also displays the contents of n1 and n3, but nested within the n2 node.

Complete the definition of the get-bst-value-list-inorder function early on, it will help with testing by displaying only a list of values for the tree.

Once you complete both the insertion and traversal methods, you can try out larger scale testing. If you have not yet defined a random number generator function, you can create your own lists of random numbers for initial testing.

```
> (define value-list-1 '(250 473 125 901 22 35 607))
> value-list-1
        '(250 473 125 901 22 35 607)
> (define tree-5 (add-value-list-to-bst empty value-list-1))
> tree-5
        (bst-node
         250
         (bst-node 125 (bst-node 22 '() (bst-node 35 '() '() 1) 1) '() 1)
         (bst-node 473 '() (bst-node 901 (bst-node 607 '() '() 1) '() 1) 1)
         1)
> (get-bst-value-list-inorder tree-5)
        '(22 35 125 250 473 607 901)
> (sort value-list-1 <)
        '(22 35 125 250 473 607 901)
>
```

You'll still need the random number generator in order to test your code by building a tree with thousands of inserted nodes.

Inorder Traversal of BST

The definition of inorder traversal of a BST is to start at the root and then for each node in the tree recursively, (1) visit the **left subtree** of the node, (2) visit the **node**, and (3) visit the **right subtree** of the node. It is a property of BSTs that listing node values in the order of the inorder traversal will result in a list of values in ascending sorted order. This will be a useful indirect test of correctness of the other functions (insert and delete).

(define (get-bst-value-list-inorder n))

Arguments

n: bst-node to traverse
initially n is the root of the tree
for recursive calls may be a deeper node

Value

list of numbers representing the inorder traversal of nodes starting at n

Algorithm

If n is empty, value is the empty list
If left and right of n are both empty, value is value of n as a list
If only left field is empty, value is value of n cons'ed to recursive call to right
If only right field is empty, value is value of n cons'ed to recursive call to left
If neither is empty, value is recursive call to left, appended to value of n as list,
appended to recursive call to right

The term "visit a node" is intentionally vague. It can be as simple as display the value at the node, or it could be some complex computation. The traversal is therefore simply a template for ordering the visits to the nodes and is independent of whatever function is performed to an individual node during the visit. Here we just need the simplest possible version of "visit" which is to note the value of the node and add it to the list for the entire tree.

Using Random Numbers for Test Cases

BST nodes can be defined to hold any type of data. But without loss of generality, we assume integer data for simplicity. In this case we can generate random numbers to support testing.

Generate random integer on range 0 to rng-1 (completed function shown)

```
(define (get-random-in-range rng)
 (inexact->exact (remainder (floor (* (random) (expt 2 31))) rng))
 )
```

This function uses the built-in Racket function **(random)** to generate a random value between 0.0 and 1.0. It then scales and mods it to limit range from 0 to rng – 1. Feel free to experiment.

Use the previous function to generate a **list** of random numbers, number of numbers is count, range is 0 to rng – 1.

(define (get-random-list-in-range rng count))

Arguments

rng: range of values (0 to rng – 1)

count: number of values (length of the list)

Value

list of the generated random numbers

Algorithm

Call the **get-random-in-range** function **count** times and generate a list of the numbers as value

Use recursion and decrement count until it reaches zero for the base case

Test to find if a list contains a specific value

(define (contains? x n))

Arguments

x: list of numbers

n: number to search for in list

Value

#t if n is contained in x

#f if n is not contained in x

Algorithm

If x is empty, value is #f

If car of x equals n, value is #f

Else value is recursive call with x replaced by cdr of x

This function uses the idea of **contains?** but checks each value in list against all other values in list. Its value is #f is the list contains no duplicates, #t if it does. Note that this is inherently an $O(n^2)$ algorithm.

(define (contains-duplicates? x))

Arguments

x: list of numbers to check for duplicates

Value

#t if contains at least one duplicate

#f if contains no duplicates

Algorithm

If list is empty or singleton, value is #f

If the cdr of x contains car x, value is #t

Else call function recursively on cdr x

Now that we have functions to check for duplicates within a list, we can use them to generate lists of random values with no duplicates. The approach is to assume the existence of a list with no duplicates then add a unique value to it. There is a functional style issue to consider. One way is to create a new random value and bind it locally using “let”. If it is a unique value, then add it to the existing list. If it is a duplicate, then discard it and try again by calling the function recursively. The second way is to not use the local binding, but rather create a helper function with arguments that represent the original list and the list with a new value already added, without yet knowing if the added value is unique or not. If the list with the new value contains no duplicates, then that list is the value for the current function. If the list contains duplicates, then the new value is ignored and the function is called recursively to generate another potential unique value.

The first function calls the helper function and introduces a new argument that represents the original list, with a new random value added. When called, it is not yet known if the new random value is unique or not.

```
(define (add-random-to-list-unique x rng)
  (add-random-to-list-unique-1 x (cons (get-random-in-range rng) x) rng)
  )
(define (add-random-to-list-unique-1 x y rng)
  (cond
    ((not (contains-duplicates? y)) y)
    (else (add-random-to-list-unique-1 x (cons (get-random-in-range rng) (cdr y)) rng))
    ))
```

To complete the function, we can now define a function to generate a list of unique random numbers. It works by repeatedly calling the function that adds one random number to an initially unique list.

```
(define (get-random-list-in-range-unique rng count)
  (cond
    ((= count 1) (add-random-to-list-unique empty rng))
    (else (append (add-random-to-list-unique (get-random-list-in-range-unique rng (- count 1)) rng)))
    ))
```

In the case of extreme values of the arguments, this way of generating unique random numbers will fail due to infinite recursion, specifically if count > rng. If count is only slightly less than range, function will terminate, but performance will be poor. However, these extremes aren't needed in practice.

In the case when count equals range or count only slightly less than range, a better alternative is to treat the function as a permutation generator rather than a traditional random number generator.

Example: generate 5 random numbers on the range 0 to 4.

> The only way to satisfy the request is to create a permutation of the numbers 0, 1, 2, 3, 4. Generating random numbers on the range 0 to 4 over and over until all values are generated is grossly inefficient.

In this case, we simply want to generate a random permutation of the original set of values. Ignore range, generate all values from 0 to count – 1, store in an array or vector, shuffle the contents of the vector, then convert to a list. The shuffle requires a relatively large number of swaps. For each swap, generate a pair of random integers that represent two distinct indexes into the vector, then exchange the contents of values at those indexes. An effective shuffle requires a large number of swaps, but there is no opportunity for non-termination.

In many testing applications, the interleaving of random values is more important than the absolute range of the values. There are advantages and disadvantages to both methods (random values vs random permutation), which we will discuss in class.

Define a function to check if a list of numbers is in ascending order.

(define (is-sorted? x))

Arguments

x: list of numbers

Value

#t if x is in ascending order, #f otherwise

Algorithm

If x is empty or singleton, value is #t

Else value is comparison of first two items in the list (car of x and car of cdr of x),
ANDed with recursive call, with x replaced by cdr x

Note that **predefined Racket function (sort)** can be used to sort lists

> (sort '(5 4 10 9 7) <)
'(4 5 7 9 10)

The 1st argument to sort is a list of numbers, and 2nd argument is the comparison operation used to sort, in this case "<" for ascending sort.

Sorting can be useful during testing. For example:

- Generate a list of unique random numbers and bind it to a symbol for reference
- Insert the list into an initially empty BST
- Generate the inorder traversal of the tree
- Compare inorder traversal to the sort of the original random list (the two lists should be identical).

Find value in tree

Find or search a value to find out if the value is in the tree or not. Rather than evaluating to true/false, evaluates to the path (path is a list of bst-nodes) from root to node with that value. Knowing the path to a node with a given value will be useful if the node is to be deleted later. Note that creating a list of bst-nodes does not modify any of the node fields, the nodes continue to hold values that form the tree that contains them. The path is merely a list of references to selected nodes from the tree.

(define (find-path v n)
(find-path-1 v n empty))

First function simply calls 2nd function with 2nd argument added to represent path.

(define (find-path-1 v n p))

Arguments

v: value searched for in the bst tree
n: the bst tree being searched
p: path from root of tree to current search point

Value

list of bst-nodes from root to bst-node with value

Algorithm

If n is empty, value is current path p
If v = value of n, value is cons of n and p
If x < value of n, cons n onto value of recursive call with n replaced by left of n
If x > value of n, cons n onto value of recursive call with n replaced by right of n

Note that the find-path function does not evaluate to a simple #t/#f. You can write a separate function contains? that calls find-path and evaluates to #t/#f

(define (bst-contains? v n))

Uses find-path to find if tree rooted at n contains value v.
If value of node at the tail of the path equals v, then value is #t, else value is #f.

Given a path of bst-nodes, create a list of just the values of the bst-nodes. Useful for debugging by suppressing the display of other fields in the bst-nodes in the path

(define (get-path-values p))

Arguments

p: a path or list of bst-node references

Value

list of values of the original list of bst-node

Algorithm

If p is empty, value is empty list
else cons value of car of p onto recursive call with x replaced by cdr of x

Delete a value from a BST (draft)

(define (delete-value-from-bst v tree))

Find path from root to value
If count of node to delete > 1, decrement count
Else must physically delete node
Child count of node to be deleted = 0
Three subcases (assume n is node to delete, p is parent of n, p can be null)
n is root (p is null)
n is the only node in the tree, after deletion, tree is empty
n is not root, n is left child of p
set left of p to null
n is not root, n is right child of p
set right of p to null
Child count = 1
Six subcases (assume n is node to delete, p is parent of n, p can be null, c is child of n)
n is root (p is null), c is left child of n
set root to left of n
n is root (p is null), c is right child of n
set root to right of n
n is not root, n is left child of p, c is left child of n
set left of p to left of n
n is not root, n is left child of p, c is right child of n
set left of p to right of n
n is not root, n is right child of p, c is left child of n
set right of p to left of n
n is not root, n is right child of p, c is right child of n
set right of p to right of n

Child count = 2

No easy way to delete n directly without rotations or other adjustments, easier to find inorder predecessor of n, swap values between n and pred, and delete pred

Child count of pred = 0

Two subcases

pred is left child of parent

set left of parent to empty

pred is right child of its parent

set right of parent to empty

Child count of pred = 1

Two subcases

pred is left child of its parent

set left of parent to left of pred

pred is right child of its parent

set right of parent to left of pred

note that in both of these cases, child of pred must be left child

Child count of pred = 2

Excluded, not possible for pred to have 2 child nodes

More detail TBD.

Example Test Cases

```
(define rlist-1 (get-random-list-unique-values 1000 10))
(define rlist-1-sorted (sort rlist-1 <))
(displayln (format "~a original list" rlist-1))
        (524 118 257 423 238 234 543 339 623 4) original list
 (displayln (format "~a sorted list" rlist-1-sorted))
        (4 118 234 238 257 339 423 524 543 623) sorted list
 (define tree-1 (add-value-list-to-bst empty rlist-1))
(displayln (format "~a inorder traversal of tree" (get-bst-value-list-inorder tree-1)))
        (4 118 234 238 257 339 423 524 543 623) inorder traversal of tree
 (delete-value-list-from-bst rlist-1 tree-1)
```

Note that the display of the sorted version of the list of inserted random numbers should be identical to the display of the inorder traversal of the tree.

Detailed Delete Trace

```
#value=524# #child count 2, root# #path =#(423 257 118 524) #actual delete value=423
(4 118 234 238 257 339 423 543 623)
#value=118# #child count 2, not root# #path =#(4 118) #actual delete value=4
(4 234 238 257 339 423 543 623)
#value=257# #child count 2, not root# #path =#(238 257) #actual delete value=238
(4 234 238 339 423 543 623)
#value=423# #child count 2, root# #path =#(339 238 4 423) #actual delete value=339
(4 234 238 339 543 623)
#value=238# #child count 1, not root#
(4 234 339 543 623)
#value=234# #child count 0, not root#
(4 339 543 623)
#value=543# #child count 1, not root#
(4 339 623)
#value=339# #child count 2, root# #path =#(4 339) #actual delete value=4
(4 623)
#value=623# #child count 0, not root#
(4)
#value=4# #child count 0, root
()
```