BST Functions for Insert and Delete

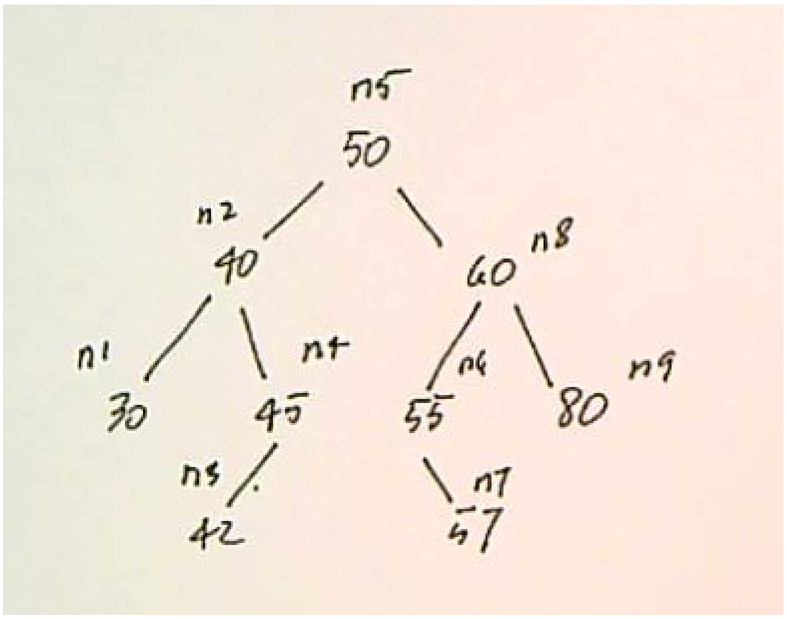
- To implement the add or insert function recursively, value should be the root node of the tree
- In an unbalanced BST, the first node inserted becomes root, and remains root unless there are rotations or deletions.
- So the root changes from empty to first node as a side effect of the first insertion
- Insertion should therefore update the tree and evaluate to root so that any external symbol for the root can also be set or updated.

- In Java, this issue is handled differently via OOD/OOP.
- You would normally provide class definitions for both Node and BST
- Root is a private instance variable of class BST and is updated internally if it changes
- In the current version of BST in Racket, we are not implementing a full OOD/OOP version.
- Changes to root are seen in the value of the add or insert function call

- Delete will also need to evaluate to the root, since deletion may change which node is root
- Any call to the insert or delete function can bind the value to the symbol that represents the original value of root
- If the root did not change, then the same value is bound again
- But if the root did change, the binding is updated

Ideas for Test Cases

Before the insertion and traversal functions are both ready to test one another, you can build a simple tree without using insertion, then test traversal in isolation.



To build the tree, first define all nodes as shown.

```
(define n1 (bst-node 30 empty empty 1))
(define n2 (bst-node 40 empty empty 1))
…
```

Then create all links using the left and right mutators as needed.

```
(set-bst-node-left! n2 n1)
(set-bst-node-right! n2 n4)
(set-bst-node-left! n4 n3)
…
```

Use node n5 as the root of the tree. You can then test traversal in isolation. Once you have traversal working, you can use it to test insertion.

Delete node from BST
Now define the function to update a tree by finding and deleting a node with a specific value if it exists.

First here are a couple of support functions that will be needed later

This function (implementation provided) is useful for testing. It converts a path (list of nodes) into a list of values. Displaying the list of values is easier to interpret than the display of a nested set of nodes.

```
(define (get-path-values p)
  (cond
   ((null? p) empty)
   (else (cons (bst-node-value (car p)) (get-path-values (cdr p))))
   ))
```

Get the number of child nodes of current node n.

```
(define (get-child-count n))
```

Arguments

    n: bst-node

Value

    0, 1 or 2

The next function will find the **inorder predecessor** of a given node. This will be needed in the case when a node should be deleted but the node has two child nodes. In that case, the solution requires deleting the inorder predecessor as a substitute. The function will find the inorder predecessor by constructing a path (a list of nodes) from the original node to the inorder predecessor and returning that path. Note that finding the inorder successor works equally well.

The simple way to describe the path to the inorder predecessor is (1) move one node down to the left, then move as many nodes as possible to the right. The final node is the inorder predecessor. Step one follows since all nodes smaller than the starting node must be either in the subtree of the left child (or in the tree above it to the left. But the left subtree nodes are closer in value than nodes above it. Step two follows since, of all nodes in the left child subtree, the node farthest down to the right is the largest. So the largest of the nodes that are less than the original node is the definition of inorder predecessor.

Note that this description assumes the original node has two child nodes. Finding the inorder predecessor of a node with 0 or 1 child nodes is more complex. For example, the minimum value in the tree has no left child (so 0 or 1 child nodes) and no inorder predecessor. The largest node in the tree has no right child, and its inorder predecessor is higher in the tree, its parent node.

```
(define (find-inorder-prev n)
  (cond
   ((null? (bst-node-right (bst-node-left n))) (list (bst-node-left n) n))
   (else (find-path-inorder-prev (bst-node-right (bst-node-left n)) (list (bst-node-left n) n)))
   ))
```

Argument

  n: original node to delete with two child nodes

Value

  Path or list of nodes from n to its inorder predecessor

Algorithm

  Base case is right of left of n is empty.
  In this case, the inorder successor is left of n
  So return a list of n and left of n

  The else clause calls the helper function with n replaced with left of n
  And the path initialized to left of n and n

Helper function continues building the path by moving to the right

(define (find-path-inorder-prev n p))

Arguments

  n: original node to delete
  p: a path (list of nodes) to the inorder predecessor (initially empty)

Value

  A path or list of nodes

Algorithm

  If right of n is null, then result is n added to head of p
  Else call function recursively, replace n with right of n, and append n to p

Delete method first finds a path from root to node to delete, if it exists, then calls helper function

```
(define (delete-value-from-bst v tree)
        (delete-node-from-bst (reverse (find-path v tree)) tree))
```

Arguments
        v: value to delete
        tree: node at root of tree
Value
        Evaluates to root node, since deletion may modify the root

For helper function
Arguments
        nl: path of nodes from root to node to delete, if it exists
        this was taken care of by original function
        tree: root of tree

```
(define (delete-node-from-bst nl tree)
```

Algorithm

```
        if node was not found, then do nothing
        if node was found and count > 1, decrement count
Else node to delete is first node in the path
Must check number of child nodes
  ; child count 0
  ; case 0.1:  node is root
  ;   tree is empty
  ; case 0.2:  node is left child of parent
  ;   set left child of parent to null
  ; case 0.3:  node is right child of parent
  ;   set right child of parent to null

  ; child count 1
  ; case 1.1:  node is root and child is left of node
  ;   set root to left child
  ; case 1.2:  node is root and child is right of node
  ;   set root to right child
  ; case 1.3:  node is left child of parent and child is left of node
  ;   set left of parent to child on left
  ; case 1.4:  node is left child of parent and child is right of node
  ;   set left of parent to child on right
  ; case 1.5:  node is right child of parent and child is left of node
  ;   set right of parent to child on left
  ; case 1.6:  node is right child of parent and child is right of node
  ;   set right of parent to child on right
```

```
    ; child count 2
    ; get inorder predecessor
    ; swap values between pred and node
    ; delete pred
    ; case 2.1:  pred has 0 children and pred is left child of its parent
    ; case 2.2:  pred has 0 children and pred is right child of its parent
    ; case 2.3:  pred has 1 child and pred is left child of its parent
    ; case 2.4:  pred has 1 child and pred is right child of its parent
```

Extra method for testing a series of deletes

```
(define (delete-value-list-from-bst x tree)
 (cond
  ((null? x) (void))
  (else
   (set! tree (delete-value-from-bst (car x) tree))
   (displayln (get-bst-value-list-inorder tree) (current-output-port))
   (delete-value-list-from-bst (cdr x) tree))
  ))
```