COMP 333
BST Programming Project Part 3

In this part we will complete the BST program by adding the delete function. The function is complex mainly because of the number of cases to consider. One reason delete is more complex than insert is because a newly inserted node has no child nodes and becomes the child node of an existing node, with no impact on any other nodes, while a deleted node can have both a parent node and child nodes, requiring any link broken by the deletion to be repaired. We will organize the cases with the help of a few helper functions to cover specific cases and subcases.

Top-Level Function: Delete Value from BST: (delete t v)
The first parameter t is a reference to a bst, and the second parameter v is a value to find and delete. Like the insert function, delete will directly resolve a few special cases. For the general cases, it will call **(find t v)** to obtain a list of nodes that represent a path from the root to the node to be deleted, and will then call a helper function **(delete-node path)** to handle the rest of the work. In the case of delete the helper function will further rely on secondary helper functions to resolve cases and subcases.
Deletion logically breaks down into cases based on how many child nodes the node to delete has. Each of those cases has subcases. The large number of cases implies the need for a correspondingly larger set of test cases: a test case to trigger each deletion case.

Delete Cases
Before we look at the detailed code for delete in Racket, let's look at the general approach. As for some of the other functions, it will be convenient to create a top-level function (delete t v) and several helper functions that manage specific detailed cases. There will be some initial special cases that should be managed separately, before launching any lower level helper function.

Special cases for (delete t v):

- Tree t is empty: end the function because there is nothing to do.

At this point, call **(find t v)** to create the path from root to node to delete if it is present. Assume the list returned is bound to a symbol named "path". The node of interest will be the first node in the list:

(car path)

By examining node **(car path)**, you can determine if the value to delete is present or not.

- Value v is not in the tree: end the function because there is nothing to do.
- Value v is in the tree:
    - Is the **count** field > 1? Don't delete the node, simply decrement its count field.
    - Is the **count** field = 1? In this case, we must move forward to actually delete the node.
        - Is the node to delete the root of the tree?
            - Does the node have 0 or 1 child nodes? This corresponds to delete cases 0.1, 1.1, and 1.2. Delete the node and update the root of the tree.
    - If we reach this point – value v is in the tree, count field is 1, node is not the root if it has 0 or 1 child nodes – then the general delete case applies, call **(delete-node path)**

The remaining delete cases will be covered by the **(delete-node path)** helper function. Some general considerations for node n (node to delete), node p (parent of n), and node c (child of n).

- Is node n the left or right child of node p?
- If node n only has one child, node c, is node c the left child or right child of node n?
- If node n has two children, we will use an additional procedure to find the in-order predecessor of n, node iop. We will locate node iop, physically delete it, and copy its value and count fields into the original node, node n. This results in a "logical" but not "physical" delete of node n.

Below is a detailed enumeration of all delete cases based on child count. Subcases 0.1, 1.1, and 1.2 will be handled as special cases by the top-level delete function because these cases cause the root of the tree to change.

Since we are maintaining the result of (find t v) in "path" we can make the following assumptions for all remaining cases:

- Node to delete: (car path)
- Parent of node to delete: (car (cdr path)) = (cadr path)
- Child nodes of node to delete, if they exist:
    - (node-left (car path))
    - (node-right (car path))
- Path from n to in-order predecessor node: ioppath (only needed if n has two child nodes)
- In-order predecessor node: (car ioppath) (only needed if node n has two child nodes)

Don't confuse **path** (list of nodes from root of tree to n) with **ioppath** (list of nodes from n to iop)

Case 0: Child Count = 0

- Subcase 0.1: parent is null implying node is root (already covered in top-level function)
  - node being deleted is root of the tree and has 0 child nodes. Set root to null
- Subcase 0.2: node is left child of parent
  - set left child of parent to null
- Subcase 0.3: node is right child of parent
  - Set right child of parent to null, root doesn't change

Case 1: Child Count = 1

- Subcase 1.1: parent is null and child is left child of node (already covered in top-level)
  - node is root, set root to left child
- Subcase 1.2: parent is null and child is right child of node (already covered in top-level)
  - Node is root, set root to right child
- Subcase 1.3: node is left child of parent and child is left child of node
  - set left child of parent to left child of node
- Subcase 1.4: node is left child of parent and child is right child of node
  - set left child of parent to right child of node
- Subcase 1.5: node is right child of parent and child is left child of node
  - set right child of parent to left child of node
- Subcase 1.6: node is right child of parent and child is right child of node
  - set right child of parent to right child of node

The cases and subcases are similar, but still differ in the specifics. Take care to identify the correct case and respond with the correct implementation for that case. In the above cases, note that the root is the only node with no parent (if parent of n is null, then n is the root). Also note the deletion cases that cause a change in the root of the tree (0.1, 1.1, and 1.2) are coded directly in the top-level function.

Case 2: Child Count = 2

There is no easy way to delete a node with two child nodes. Instead we **substitute** the original problem with a problem that is easier to solve:

- Locate the in-order predecessor of the node to be deleted.
- Move the value and count fields from the in-order predecessor node to the original node, thereby erasing the previous value and count fields of the original node. This content substitution achieves a logical removal of the original node.
- Physically remove the in-order predecessor node
  - Case 2.1: predecessor node is left of parent
  - Case 2.2: predecessor node is right of parent

Finding the in-order predecessor, preserving its contents by copying into the original node, then physically deleting the in-order predecessor node, is a roundabout equivalent of deleting the original node. It solves the problem while preserving the BST property of all nodes.

Deletion of the in-order predecessor is guaranteed to be simpler with fewer cases to consider than for the original general deletion, which reduces the number of cases to consider:

- In-order predecessor cannot be the root of the tree
- In-order predecessor can have at most 1 child node because it cannot have a right child

The in-order successor can be used for this trick just as easily as the in-order predecessor. I'm arbitrarily picking the in-order predecessor for this implementation.

Finding the In-Order Predecessor of a BST Node with Two Children (left child not null)

- Locate the left child of the node.
- If the left child has no right child, then the left child is the in-order predecessor.
- If the left child has a right child, then move to the right child. Keep moving right until you reach a descendant node with no right child. This node is the in-order predecessor of the original node.
- Keep track of the path you follow from original node to its in-order predecessor by creating a list the nodes along the path. This path will be needed with deleting the in-order predecessor node.

This algorithm is tailored to find the in-order predecessor of **a node with a left child**. A more general algorithm for finding the in-order predecessor of any arbitrary node would have more cases.

Top Level Function for Delete: (delete t v)

This function starts the delete process. It calls **(find t v)** to obtain the path (list of nodes) from root to node n, the node to be deleted. Assume **(find t v)** is bound to symbol **path.** Also note that n is available as **(car path)**, the head of the list of nodes obtained from (find t v).

Special cases

- Tree t is empty (same as path is null): nothing to delete, do nothing
- Value of (car path) is not equal to v:
  - Node with value v is not present in tree, do nothing
- Value of (car path) equals v AND count of (car path) > 1:
  - Don't delete the node, just decrement count of (car path)
- Node value of (car path) = v (use (= a b) to compare numbers)
  AND count value of (car path) = 1
  AND (car path) equals (bst-root t) (use (equal? a b) to compare node references)
  AND (car path) has either 0 or 1 child nodes:
  - The node to be deleted is the root corresponding to delete subcases 0.1, 1.1, 1.2 listed above; update root of t to either null, left of (car path) or right of (car path)
- Else call helper function with stack as parameter (delete-node path); note that the node to be deleted is the first item in path, so there's no need to pass it as an additional parameter

Code Skeleton for (delete t v)

```
(define (delete t v)
        (let ((path (find t v)))          ; call find to get path from root to node to delete
                (cond
                        …                 ; implement top-level special cases here
                                          ; else call delete-node
                )
        )
)
```

When calling the helper function, the only parameter required is "path". The node to delete and its value are encoded into (car path) and (node-value (car path)).

Helper function (delete-node path)

```
(define (delete-node path)
        (let ((n (car path)))                                          ; n is node to delete
                (cond
                        ((= 0 (child-count n)) (delete-node-0 path))   ; n has 0 child nodes
                        ((= 1 (child-count n)) (delete-node-1 path))   ; n has 1 child node
                        (else (delete-node-2 path))                    ; n has 2 child nodes
        )))
```

Specific Helper Functions Based on Child Count

Write a helper function (child-count n) to calculate the number of child nodes of n, either 0, 1, or 2. Then, write the helper functions for each child count case. As an example of how to code the cases, consider subcase 1.4.

- n: node to delete: (car path)
- p: parent of n: (cadr path)

Note that p cannot be null, since this would imply that n is the root of the tree, and those cases were handled earlier. In case 1.4, n is the left child of p, and right child of n is not null.

What is the test to detect case 1.4?

```
(and
        (equal? (node-left p) n)                ; n is the left child of its parent
        (not (empty? (node-right n)))           ; child of n is its right child
)
```

In this case, n is deleted by setting left of p to right of n:

```
(set-node-left! p (node-right n))
```

So the code to add to the (delete-node-1 path) helper function is:

```
(define (delete-node-1 path))
        (let ((n (car path))(p (cadr path)))        ; use "let" to create local bindings for symbols
                (cond
                        …
                        ((and (equal? n (node-left p)) (not (empty? (node-right n))))       ; test for 1.4
                                (set-node-left! p (node-right n)))                          ; update for 1.4
                        …
                )
        )
)
```

Other cases are tested and updated similarly.

Child Count = 2

In this case we cannot directly delete a node with two child nodes since there is no easy way to restore the links that would be broken. Instead, we locate the in-order predecessor of the node to delete, and delete the predecessor instead.

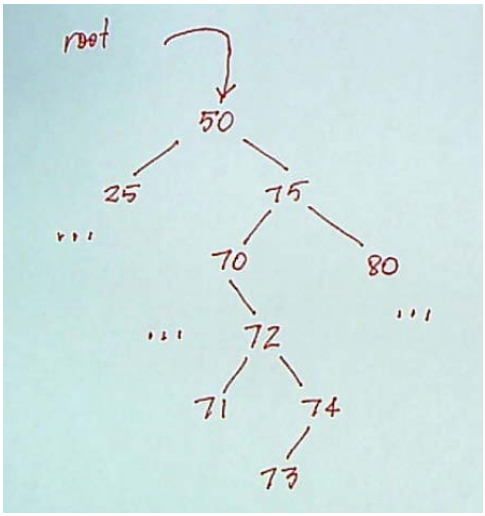
The in-order predecessor m of node n is the node whose value comes immediately before n if we look at all values in the tree in sorted order. Another way of describing it is to consider all nodes whose values are less than n, then find the largest of those values: the largest value in the tree that is less than n. If a node has two child nodes, then the nodes to its left have smaller values. And of those smaller values, the rightmost node is the largest of them.

The reason that deleting the predecessor will fix our deletion problem is that although we are going to physically delete that node, we are not going to delete its value. We are going to copy its value field into the value field of the original node. This causes the value of the original node to be replaced by the value of the predecessor, and effectively deletes the value of the original node without deleting the node itself. Since the predecessor is the largest value smaller than the original, if we position it in the tree in the same location where the original value was, then we have not violated any BST relationships between values. All values to the left of the predecessor are still smaller, and all values to the right of the predecessor are still larger.

We could equivalently choose the in-order **successor** instead. There's no real advantage of using one over the other, so we'll use the predecessor.

Example of In-Order Predecessor

Consider the following BST



We'll use node 75 as an example of a node with 2 child nodes.

(delete t 75)

The (find t 75) function will create a path from root to node 75

path = (find t 75) = (75 50)

We will define another path-building function (get-iop-path n) to start at node n and generate the path from n to its in-order predecessor.

ioppath = (get-iop-path (car path)) = (74 72 70 75)

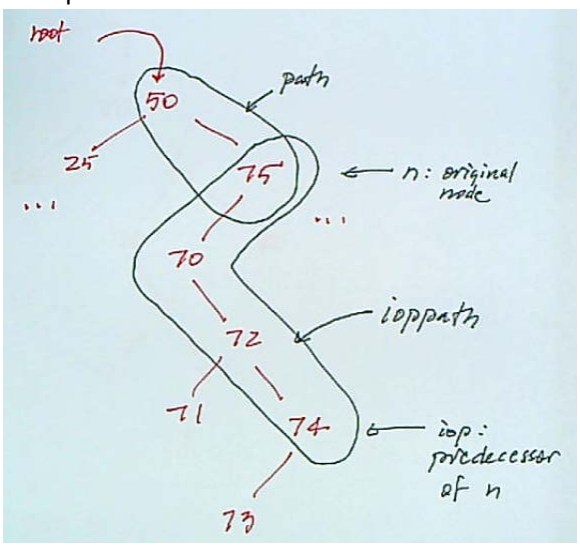
The in-order predecessor of node 75 is node 74, which is the first node in ioppath:

iop = in-order predecessor of node 75 = node 74 = (car ioppath)

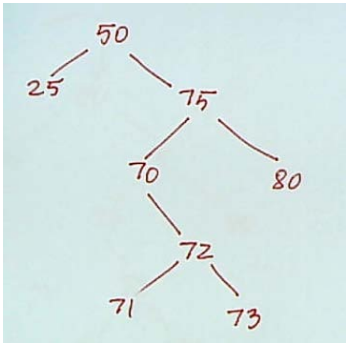
The parent of node 74 is node 72, which is (cadr ioppath)

piop = parent of iop = (cadr ioppath)

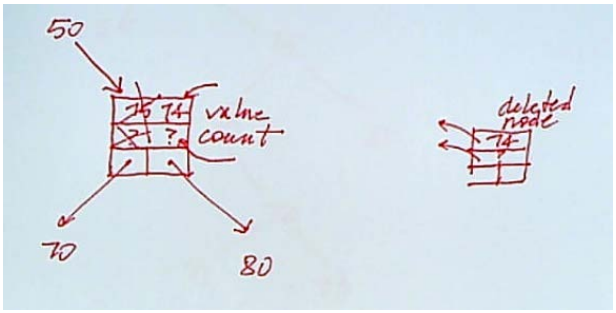
The updated picture of the tree is



Don't confuse **path**, which is a list of nodes from root of tree to node to delete, and **ioppath**, which is a list of nodes from node to delete to its predecessor. We now physically delete node 74, by setting right of 72 to left of 74, resulting in the modified tree:



Finally, we must do a little fix-up because the original task was to delete node 75, not node 74. We fix it up by replacing the value field of node 75 with the value 74.



So the 75 node was never actually deleted. Instead its value and count fields were replaced by the value and count fields from the iop node that we did physically delete. But since we put the value fields of the iop node back into the tree, the net effect is the same as if we'd deleted the 75 node and never touched the 74 node.

Generating ioppath

To find the path from node n to its iop, use the following algorithm:

- ioppath = (list (node-left n) n) ; initially ioppath is a list of node n and its left child
- Loop
    - If (car ioppath) has no right child, done.
    - Else add (node-right (car ioppath)) to head of ioppath

To implement, create a top-level function (get-iop-path n) that adds the first two nodes to the path. It then calls a helper function (get-iop-path-right path) that calls itself recursively to add the right child of the most recent node to the head of the list until it reaches a node with no right child.

Test Helper Functions

A detailed test to show that the tree has been updated correctly after a delete will need to show individual nodes in more detail. Rather than displaying just a node's value, we can write a modified version of "nl-to-vl" called "nl-to-all" which shows all fields – value, count, left and right – for each node.

The first function creates a list of the values of the child nodes of node n. If the child is not null, then the value of the node is used. If the child is null, then the symbol "X" is used.

```
(define (get-child-node-values n)
 (cond
  ((and(empty? (node-left n))(empty? (node-right n)))
   (list 'X 'X))
  ((and(empty? (node-left n))(not(empty? (node-right n))))
   (list 'X (node-value (node-right n))))
  ((and(not(empty? (node-left n)))(empty? (node-right n)))
   (list (node-value (node-left n)) 'X))
  (else
   (list (node-value(node-left n))(node-value(node-right n))))
  ))
```

Using this function, we can write a function that creates a list of all field values for node n:

```
(define (get-node-fields n)
 (cond
  ((empty? n) "X")
  (else (append (list (node-value n) (node-count n)) (get-child-node-values n)))
  ))
```

Finally, we can write a modified version of nl-to-vl that converts a list of nodes into a more detailed list of values, including value, count, left, and right fields for each node:

```
(define (nl-to-all x)
 (cond
  ((empty? x) (list 'X))
  (else (append (list (get-node-fields (car x))) (nl-to-all (cdr x))))
  ))
```

Add these function definitions to your source code when you're ready to run the required test cases below.

Required Test Case #1: (get-iop-path)

Test **(get-iop-path)** using example tree from above:

```
(define q (bst null))                          ; build tree
(define qlist '(50 25 75 70 80 72 71 74 73))
(insert-from-list q qlist)

(define n75 (car (find q 75)))                 ; find node 75
(node-value n75)

(nl-to-vl (get-iop-path n75))                  ; find path from 75 to iop
```

Run Result

```
(50)
(25 50)
(25 50 75)
(25 50 70 75)
(25 50 70 75 80)
(25 50 70 72 75 80)
(25 50 70 71 72 75 80)
(25 50 70 71 72 74 75 80)
(25 50 70 71 72 73 74 75 80)          ; result of build tree
75                                    ; value of node 75
'(74 72 70 75)                        ; ioppath from node 75 to node 74
```

Required Test Case #2: Delete Subcases

The following set of twelve test cases tests each individual delete subcase by building a simple tree then deleting the node that exercises that case. For each test, the same tree is reused. It is emptied then repopulated with values needed for the next test. By using the “nl-to-all” function we can take a before and after snapshot of the tree and confirm that the correct node was deleted and the other links in the tree were updated correctly.

```
 (define bst-1 (bst null))

; --- delete cases that change root
; --- delete case 0.1
(displayln "") (displayln "delete case 0.1")
(set! bst-1 (bst null))
(insert-from-list bst-1 '(50))
(nl-to-all (traverse bst-1))          ; ((50 1 X X) X) (50 is root)
(delete bst-1 50)
(nl-to-all (traverse bst-1))          ; (X) (root is null)

; --- delete case 1.1
(displayln "") (displayln "delete case 1.1")
(set! bst-1 (bst null))
(insert-from-list bst-1 '(50 25))
(nl-to-all (traverse bst-1))          ; ((25 1 X X) (50 1 25 X) X) (50 is root)
(delete bst-1 50)
(nl-to-all (traverse bst-1))          ; ((25 1 X X) X) (25 is root)

; --- delete case 1.2
(displayln "") (displayln "delete case 1.2")
(set! bst-1 (bst null))
(insert-from-list bst-1 '(50 75))
(nl-to-all (traverse bst-1))          ; ((25 1 X X) (50 1 25 X) X) (50 is root)
(delete bst-1 50)
(nl-to-all (traverse bst-1))          ; ((25 1 X X) X) (25 is root)

; --- delete cases that don't change root
; --- delete case 0.2
(displayln "") (displayln "delete case 0.2")
(set! bst-1 (bst null))
(insert-from-list bst-1 '(50 25))
(nl-to-all (traverse bst-1))          ; ((25 1 X X) (50 1 25 X) X) (50 is root)
(delete bst-1 25)
(nl-to-all (traverse bst-1))          ; ((25 1 X X) X) (25 is root)

; --- delete case 0.3
(displayln "") (displayln "delete case 0.3")
(set! bst-1 (bst null))
(insert-from-list bst-1 '(50 75))
(nl-to-all (traverse bst-1))          ; ((25 1 X X) (50 1 25 X) X) (50 is root)
(delete bst-1 75)
(nl-to-all (traverse bst-1))          ; ((25 1 X X) X) (25 is root)

; --- delete case 1.3
(displayln "") (displayln "delete case 1.3")
```

```
(set! bst-1 (bst null))
(insert-from-list bst-1 '(50 25 12))
(nl-to-all (traverse bst-1))          ; ((25 1 X X) (50 1 25 X) X) (50 is root)
(delete bst-1 25)
(nl-to-all (traverse bst-1))          ; ((25 1 X X) X) (25 is root)

; --- delete case 1.4
(displayln "") (displayln "delete case 1.4")
(set! bst-1 (bst null))
(insert-from-list bst-1 '(50 25 35))
(nl-to-all (traverse bst-1))          ; ((25 1 X X) (50 1 25 X) X) (50 is root)
(delete bst-1 25)
(nl-to-all (traverse bst-1))          ; ((25 1 X X) X) (25 is root)

; --- delete case 1.5
(displayln "") (displayln "delete case 1.5")
(set! bst-1 (bst null))
(insert-from-list bst-1 '(50 75 60))
(nl-to-all (traverse bst-1))          ; ((25 1 X X) (50 1 25 X) X) (50 is root)
(delete bst-1 75)
(nl-to-all (traverse bst-1))          ; ((25 1 X X) X) (25 is root)

; --- delete case 1.6
(displayln "") (displayln "delete case 1.6")
(set! bst-1 (bst null))
(insert-from-list bst-1 '(50 75 85))
(nl-to-all (traverse bst-1))          ; ((25 1 X X) (50 1 25 X) X) (50 is root)
(delete bst-1 75)
(nl-to-all (traverse bst-1))          ; ((25 1 X X) X) (25 is root)

; --- delete case 2.1 #1
(displayln "") (displayln "delete case 2.1 #1")
(set! bst-1 (bst null))
(insert-from-list bst-1 '(50 25 75 12 30))
(nl-to-all (traverse bst-1))          ; ((25 1 X X) (50 1 25 X) X) (50 is root)
(delete bst-1 25)
(nl-to-all (traverse bst-1))          ; ((25 1 X X) X) (25 is root)

; --- delete case 2.1 #2
(displayln "") (displayln "delete case 2.1 #2")
(set! bst-1 (bst null))
(insert-from-list bst-1 '(50 25 75 12 30 10))
(nl-to-all (traverse bst-1))          ; ((25 1 X X) (50 1 25 X) X) (50 is root)
(delete bst-1 25)
(nl-to-all (traverse bst-1))          ; ((25 1 X X) X) (25 is root)

; --- delete case 2.2
(displayln "") (displayln "delete case 2.2")
(set! bst-1 (bst null))
(insert-from-list bst-1 '(50 25 75 12 30 10 13 14))
(nl-to-all (traverse bst-1))          ; ((25 1 X X) (50 1 25 X) X) (50 is root)
(delete bst-1 25)
(nl-to-all (traverse bst-1))          ; ((25 1 X X) X) (25 is root)
```

Program Output from Detailed Delete Cases

delete case 0.1
(50)
'((50 1 X X) X)
'(X)

delete case 1.1
(50)
(25 50)
'((25 1 X X) (50 1 25 X) X)
'((25 1 X X) X)

delete case 1.2
(50)
(50 75)
'((50 1 X 75) (75 1 X X) X)
'((75 1 X X) X)

delete case 0.2
(50)
(25 50)
'((25 1 X X) (50 1 25 X) X)
'((50 1 X X) X)

delete case 0.3
(50)
(50 75)
'((50 1 X 75) (75 1 X X) X)
'((50 1 X X) X)

delete case 1.3
(50)
(25 50)
(12 25 50)
'((12 1 X X) (25 1 12 X) (50 1 25 X) X)
'((12 1 X X) (50 1 12 X) X)

delete case 1.4
(50)
(25 50)
(25 35 50)
'((25 1 X 35) (35 1 X X) (50 1 25 X) X)
'((35 1 X X) (50 1 35 X) X)

delete case 1.5
(50)
(50 75)
(50 60 75)
'((50 1 X 75) (60 1 X X) (75 1 60 X) X)
'((50 1 X 60) (60 1 X X) X)

delete case 1.6
(50)
(50 75)

(50 75 85)
'((50 1 X 75) (75 1 X 85) (85 1 X X) X)
'((50 1 X 85) (85 1 X X) X)

delete case 2.1 #1
(50)
(25 50)
(25 50 75)
(12 25 50 75)
(12 25 30 50 75)
'((12 1 X X) (25 1 12 30) (30 1 X X) (50 1 25 75) (75 1 X X) X)
'((12 1 X 30) (30 1 X X) (50 1 12 75) (75 1 X X) X)

delete case 2.1 #2
(50)
(25 50)
(25 50 75)
(12 25 50 75)
(12 25 30 50 75)
(10 12 25 30 50 75)
'((10 1 X X) (12 1 10 X) (25 1 12 30) (30 1 X X) (50 1 25 75) (75 1 X X) X)
'((10 1 X X) (12 1 10 30) (30 1 X X) (50 1 12 75) (75 1 X X) X)

delete case 2.2
(50)
(25 50)
(25 50 75)
(12 25 50 75)
(12 25 30 50 75)
(10 12 25 30 50 75)
(10 12 13 25 30 50 75)
(10 12 13 14 25 30 50 75)
'((10 1 X X) (12 1 10 13) (13 1 X 14) (14 1 X X) (25 1 12 30) (30 1 X X) (50 1 25 75) (75 1 X X) X)
'((10 1 X X) (12 1 10 13) (13 1 X X) (14 1 12 30) (30 1 X X) (50 1 14 75) (75 1 X X) X)

Required Test Case #3: Combined Insert/Delete Cases

Once all functions are complete, we can generate a list of random numbers, insert them, then delete them, and show a snapshot of a traversal of the tree after every operation. For this test, we don't need the verbose node snapshot, we can simply show the value of the node to represent it. However, you could substitute nl-to-all to get a more detailed snapshot of the tree, which could be helpful if you need more insight into errors in your code.

```
(define bst-1 (bst null))                    ; initially empty bst

(define list-1 (random-list 10 1000))        ; list of 10 random numbers
(displayln list-1)

(insert-from-list bst-1 list-1)              ; insert numbers from list into bst
(nl-to-vl (traverse bst-1))                  ; snapshot of tree

(delete-from-list bst-1 list-1)              ; delete numbers from list from bst
(nl-to-vl (traverse bst-1))                  ; snapshot of tree
```

Example Run Result

```
(954 897 907 925 449 157 454 829 559 980)    ; list

(954)                                        ; insert test
(897 954)
(897 907 954)
(897 907 925 954)
(449 897 907 925 954)
(157 449 897 907 925 954)
(157 449 454 897 907 925 954)
(157 449 454 829 897 907 925 954)
(157 449 454 559 829 897 907 925 954)
(157 449 454 559 829 897 907 925 954 980)

'(157 449 454 559 829 897 907 925 954 980)   ; traversal after all insertions

(157 449 454 559 829 897 907 925 954)        ; delete test
(157 449 454 829 897 907 925 954)
(157 449 454 897 907 925 954)
(157 449 897 907 925 954)
(449 897 907 925 954)
(897 907 925 954)
(897 907 954)
(897 954)
(954)
()

'()                                          ; traversal after all deletions
```

Results show that we start with empty bst, insert values, delete them, and end with empty bst.