CS 677: Assignment 4

Terence Henriod

March 17, 2014

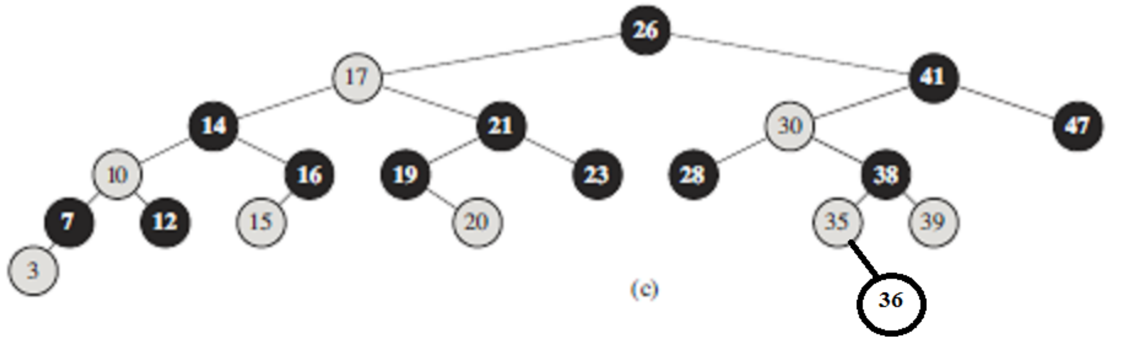
Abstract

In this assignment, operations related to Red-Black, Order-Statistic, and interval trees are considered.

1. Exercises 13.1-2 and 13.1-3 (page 311).

   (a) 13.1-2: Draw the red-black tree that results after TREE-INSERT is called on the tree in Figure 13.1 with key 36. If the inserted node is colored red, is the resulting tree a red-black tree? What if it is colored black?

   Solution:



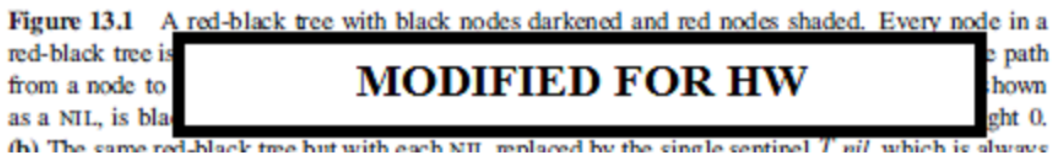


Figure 13.1 A red-black tree with black nodes darkened and red nodes shaded. Every node in a
red-black tree is   path
from a node to **MODIFIED FOR HW** hown
as a NIL, is bla   ght 0.
(b) The same red-black tree but with each NIL replaced by the single sentinel *T.nil*, which is always

Figure 1: The addition of a node with key 36 to the tree of Figure 13.1.

   Given that TREE-INSERT does not do anything to maintain any of the properties of a red-black, this insert case will not lead to a valid red-black tree. If the inserted node is red, then the resulting tree has two red nodes in a row, which is a violation of property #4, if the inserted node is colored black, this is still not a valid red-black tree because the black-height of all subtrees will not be the same without rotations to even out the black heights, violating property #5.

   (b) 13.1-3: Let us define a **relaxed red-black tree** as a binary search tree that satisfies redblack properties 1, 3, 4, and 5. In other words, the root may be either red or black. Consider a relaxed red-black tree T whose root is red. If we color the root of T black but make no other changes to T , is the resulting tree a red black tree?

   Solution: Yes. Because property #4 (if a node is red, both of its children must be black) is still maintained in the relaxed red-black in the red black tree, we can say that any children of the (red) root will be black. Changing the root to black will not violate properties #1 (each node will still be red or black), #3 (every leaf (nil) will still be black), #4 (any red nodes will still have black children), and #5 (the black heights at every node will be the dame for any of its sub-trees).

2. Exercise 13.3-5 (page 322): Consider a red-black tree formed by inserting n nodes with RB-INSERT. Argue that if $n > 1$, the tree has at least one red node.

   Solution: I will make my argument using induction:

   Inductive hypothesis
   The hypothesis is that inserting any number of nodes n, $\forall n > 1$, will leave at least one red node in the tree.

   Base Case
   Inserting one node will result in that node being painted black to comply with property #2 (the root must be black). Clearly, this does not result in at least one red node in the tree, so it cannot be our base case (which is fine, because we did not define our base case this way). The true base case involves the next node to be inserted (so actual base case is $n = 2$). This node will be inserted as a red node, and this node will not be re-painted or relocated because it is red or black (1), the root node will remain black (2), the leaves will remain black as long as the inserted red node is made to point to the NIL node as a child (3 and 4), and the new red node will not change the black height of any subtree (5), so since all properties hold, no changes need to be made.

   Assumption
   Assume that inserting n nodes will produce at least 1 red node in the tree.

   To Show
   If an additional node is inserted, it will be inserted as red, and one of 2 things can occur:

   - The new node is inserted as the child of a black node.

   - The new node is inserted as the child of a red node.

   In the first case, inserting a red node will not alter/violate any of the 5 red-black tree properties, so no transformations to fix the tree will be made, leaving the red node in the tree, and the hypothesis holds.

   In the second case, transformations must be made to alter the tree to undo the violation of property #4 so that no red node has a red child. There are 3 such transformations used to remedy 3 different cases of property violating insertions. The three transformations are:

   - Node re-painting to "push a red node up the tree"

   - A left rotation of a particular pair of nodes

   - A right rotation of a perticular pair of nodes

   The three cases where any of these transformations might be applied are:

   - Case 1: When a red node is inserted as the child of a red node whose sibling is also red

   - Case 2: When a red node is inserted as the child of a red node whose sibling is black

- Case 3: When a red node is inserted as the "inside" child of a red node whose sibling is black

Note that of the three transformations, only a re-paint will alter the number of red nodes in the tree. This re-paint only occurs when "Case 1" appears, the case when a black parent node with two red children, and one red grandchild, is detected. In this case, the re-paint results in two red nodes (the children) and a black one (the parent) being exchanged for two blacks (again, the children) and a red one (the parent). The red grandchild is left unaltered. So even though the re-paint operation essentially merges two reds, there will still be two red nodes left after the operation. Even if the red node gets pushed up to become the root node and it gets repainted black, the red grandchild will still remain, so at least one red node (usually more) will always remain in a red-black tree with 2 or more nodes.

3. Exercise 14.1-6 (page 345): Observe that whenever we reference the size attribute of a node in either OS-SELECT or OS-RANK, we use it only to compute a rank. Accordingly, suppose we store in each node its rank in the subtree of which it is the root. Show how to maintain this information during insertion and deletion. (Remember that these two operations can cause rotations.)

   Solution: In general, ranks can be adjusted as the tree is descended in the search for insertion/deletion points. Because rank information refers only to a node's rank within its sub-tree, not all ranks need to be updated every time the tree is altered. Each node's rank information depends only on the size of its right sub-tree.

   Insert
   During insertion, as the tree is descended, if the search path needs to go right, then the current node's rank should be incremented before the search goes down the right path. Once the insertion point is found, the new node should be given a rank of 1. During fix-up, should a right rotation occur, then the node that was on the left (the one that will be rotated "upward") will need to be given a rank equal to the rank it originally had plus the rank of the right node (the one that is rotated "downward"), the size of the left node's right sub-tree. In a left rotation, the right node that is moved upward will keep it's rank. The left node that is moved downward will get a rank equal to its new right child's (the right node's former left child) rank plus one.

   Delete
   When deleting a node in the modified OS-tree, when searching for the node to be deleted, as the search descends the tree to the right, the rank of the current node should be decremented before the search descends to the right, similar to the way ranks were incremented in the insertion process. The node to be deleted is either removed, or its in-order predecessor is removed in the same manner via recursion. This will work for all 3 deletion cases. Again, should a rotation occur, in the case that it is a right rotation, the left node will get the rank it had originally plus the right node's rank; the right node's rank will remain unchanged. In the case of a left rotation, the left node that is moved downward will receive a rank of its new right child plus one; the right node will keep the same rank it had originally because it's right sub-tree will not have changed.

4. Exercise 14.3-3 (page 353): Describe an efficient algorithm that, given an interval i, returns an interval overlapping i that has the minimum low endpoint, or $T.nil$ if no such interval exists.

   Solution: An efficient algorithm that would accomplish this should ideally do so in $\Theta\big(\lg n\big)$ time. Since interval trees are sorted by the low end point of their intervals, this could be accomplished by searching for an interval with that has a low value that overlaps with i, then by moving leftwards, down the tree to find the interval that still overlaps with interval i but that has the lowest endpoint. Such an algorithm would look like:

   Algorithm *FIND-LOWEST-INTERVAL*(T, i)

```
1.  $x \leftarrow T.root$
2.  (* Find the first overlapping interval *)
3.  while $x \neq T.nil$ and $(i.high < x.low)$ and $(i.low > x.high)$
4.      do
5.          if $i.high < x.low$
6.             then
7.                  $x \leftarrow x.right$
8.             else
9.                  if $i.low > x.high$
10.                    then $x \leftarrow x.left$
11. (* Find the overlapping interval with the lowest end point *)
12. while $x.left \neq T.nil$ and $i.high \geq x.low$
13.     do $x \leftarrow x.left$
14. (* Return a pointer to the found node (or nil) *)
15. return $x$
```

   This algorithm is basically a modified binary search that spends constant time at each node, and only visits one node per level of the tree, so it intutitively is a $\Theta\big(\lg n\big)$ algorithm.

5. Exercise 9.1-1 (page 215): Show that the second smallest of n elements can be found with $n + \lceil \lg n \rceil - 2$ comparisons in the worst case. (*Hint*: Also find the smallest element.)

   Solution: It is known that it takes $n-1$ comparisons to find the smallest element in a set by conducting a "tournament" where elements are compared using a divide and conquer strategy. This method is often represented graphically using a tree. If an element "beats" another, it advances to a higher level of the tree to be compared to another element of the set, and the "losing" element is no longer compared to any other elements. It can be seen that $n-1$ comparisons were made because a complete binary tree has at most $n-1$ internal nodes (nodes that are not leaves). This is can be seen by solving the sum $\sum_{i=0}^{\lg n} 2^i = 2n - 1$ which gives the number of all the nodes in a full binary tree, and then subtracting the n leaves gives $n-1$ remaining nodes (for a FULL binary tree). The number of internal nodes is significant because it counts the number of comparison results. For a less than full binary tree, there will be fewer comparisons because "byes" will appear in our tournament.

   The second smallest element would have lost to the smallest element somewhere in the tournament. The number of elements that were compared to the smallest element would be at most $\lceil \lg n \rceil$ elements, or one for each level of the the tree formed by the tournament (not counting the root level, where there are no more elements to compare with the smallest elemtent), since the height of a complete binary tree is $\lceil \lg n \rceil + 1$. To find the second smallest element, one only needs to gather all the elements that were compared with the smallest element in the first tournament and hold a sort of "consolation" tournament. The winner of this second tournament will be the second smallest element of the set. Since it is known that holding a tournament with n elements requires $n - 1$ comparisons (see above), then a tournament of $\lg n$ nodes would require $\lg n - 1$ comparisons.

   Thus, combining the costs of the tournaments results in $n - 1 + \lceil \lg n \lceil -1 = n + \lceil \lg n \rceil - 2$ comparisons.

   I feel is should be noted that while this algortithm might technically be optimal in terms of comparisons, it does invlove the overhead of storing lists of elements for the participant elements of the second tournament. This is not ideal if space is a concern. An "in-place" iterative algorithm would be ideal for space usage, and could perform the same operation in at most $2n - 3$ comparisons, which is $\mathrm{O}(n)$ time, similar to the algortithm above. But, since the question only asked about comparisons, I won't go in to this further.

6. Exercise 13.2-4 (page 314): Show that any arbitrary n-node binary search tree can be transformed into any other arbitrary n-node binary search tree using $\mathrm{O}\big(n\big)$ rotations. (*Hint*: First show that at most $n-1$ right rotations suffice to transform the tree into a right-going chain.)

   Solution: Using the hint, consider how to turn any tree into a right-going chain using at most $n-1$ right rotations. If one were to use right rotations that were proximal to the *right spine* of the tree, that is right rotations involving one node already in the right spine of the tree and one node exactly one leftward branch away from the spine. Each right rotation would move one node into the right spine and push the other node involved down a level in the right spine of the tree. The net result is that 1 node was moved into the right spine of the tree, and none were moved out of the right spine. If we were to consider the root node as already being in the right spine, then even in the worst case where the other $n-1$ nodes are not in the right spine, they can be moved over one at a time.

   Let any sequence of right rotations that transform a binary tree to a right-going chain be denoted as R_i. Note that the effects of any rightward rotation can be reversed with a leftward rotation. Let a sequence that transforms a right-going chain back to an arbitrary binary tree by reversing the right rotations of sequence R_i be denoted by L_i.

   Considering these two facts, suppose we have some arbitrarily constructed binary tree T_1. Then we can transform this tree into a right-going chain by shifting all of it's nodes into a right spine, call this sequence of rotations R_1. Now suppose we have a second tree, T_2, that can be transformed to a right-going chain by the sequence of right rotations R_2. Then we could transform T_1 to T_2 by applying rotation sequences R_1 and L_2, and vice versa. The size of each rotation sequence is at most $n-1$, so even by fully using both rotation sequences, at most $2*(n-1) = 2n-2$ rotations are performed, which, of course, is $\mathrm{O}\big(n\big)$ rotations.