CSE 450/551: Solutions for Homework 4, Spring 2018

Prof. Andréa W. Richa

1. **Answer:** For simplicity, assume all elements are distinct.

   JOIN(T_1, T_2): Find maximum element v in T_1. Splay(T_1, v). Make root of T_2 be the right child of v (node v does not have a right child since v is the maximum element in T_1).

   SPLIT(T, a): BST Search(T,a). Splay(T,a). Let T_1 be equal to a and the left subtree of a and T_2 be equal to the right subtree of a.

   Let n_1 and n_2 be the number of elements in T_1 and T_2 resp., and let $n = n_1 + n_2$. The amortized cost of JOIN is dominated by the amortized cost of the splay operation within it and hence it is equal to $O(\log n)$.

   The amortized cost of the SPLIT operation is dominated by the amortized cost of the splay operation within it and/or by the change in rank of node v (which will become the root of the combined tree), since after we join the two trees in the last step of this procedure, the rank of all nodes remains the same as right after the splay operation with the exception of node v. The rank of node v will now be $\log(n_1 + n_2) = \log n$ up from the rank of $\log n_1$ right after the splay operation, and hence will contribute an added amount of $\log n - \log n_1$ to the increase in potential after a split operation (which was not taken into account by the splay operation. Hence the total amortized cost of the SPLIT operation can be upper bounded by the amortized cost of the splay operation plus the final rank of node v, which gives us $O(\log n) + \log n = O(\log n)$.

   The JOIN and SPLIT operations cannot be guaranteed to have $O(\log n)$ worst-case running time on Red-Black trees, since in order to make sure that the resulting tree structure is a valid RB-tree, one would need to reposition more than a constant number of nodes, at a total cost of more than $O(\log n)$. Here is a simple counterexample: Let's say that T_1 consists of the (near-)complete binary tree with $\log n$ nodes, and T_2 consists of the (near-)complete binary tree with $n - \log n$ nodes, where all keys in T_1 are less than or equal to all keys in T_2 and all nodes are colored black. Let's say that we first delete the node v with maximum key in T_1 (following RB-Delete) and use it as the new root for the joint $T_1 \cup T_2$ tree, by making the roots of T_1 and T_2 the left and right children of v respectively. The resulting tree will not be a valid RB-tree, since the number of black nodes

on the paths from v to leaves in T_2 will be $k = \Theta(\log n)$ and the number of black nodes on the path from v to a leaf in T_1 would be $\Theta(\log \log n)$ asymptotically smaller than k; in order to fix this imbalance, one would need to move at least a linear number of nodes from T_2 into T_1 (maybe by re-inserting the nodes via RB-Insert into T_1 directly), or to re-insert all the nodes in T_1 into T_2 directly via RB-Insert, which would lead to a running time of at least $O(\log^2 n) > O(\log n)$. Similarly one can show that the cost of a split operation could be high in a RB-tree, since one may not be able to bring a to the root of the RB-tree (maybe having a as the root would produce a very imbalanced BST), and extracting all nodes with key smaller than that of A may take time proportional to $\log^2 n$.

2. **Answer: (a)** Performing a rotation involves two things: pointer manipulation and size recalculation. Refer the figures in Jeff Erickson's Note (lecture 6 page 4). Let's call the subtrees (triangles) $A, B, C, (D)$ from left to right and a be the root of A, b be the root of B, etc.

   case1: right rotation at x

   After rotation we can reassign the pointers so that p which was the parent of y is now the parent of x, x's right child is y, and y's left child is b, and y's parent is x. It takes only constant time.

   Size recalculation:

   $$size(x) = size(a) + size(b) + size(c) + 2$$
   $$size(y) = size(b) + size(c) + 1$$

   Hence size recalculation takes only constant time.

   case1: right roller-coaster at x

   We can observe reassigning the pointer for node x, y, z can be done in constant time.

   Size recalculation:

   $$size(x) = size(a) + size(b) + size(c) + size(d) + 3$$
   $$size(y) = size(b) + size(c) + size(d) + 2$$
   $$size(z) = size(c) + size(d) + 1$$

   Hence size recalculation takes only constant time.

   We can check all other cases using the same way. Thus a rotation in an augmented binary search tree takes constant time.

   (b)

   SPLAYSELECT(k, root)

   v = root

   SPLAY(SELECT(k, v))

   SELECT(k, v)

   if size[$v.leftchild$] = $k - 1$ then

return v

else if size[$v.leftchild$] + 1 $<$ k then

return SELECT($k-$ size[$v.leftchild$] -1 , v.rightchild)

else

return SELECT(k , v.leftchild)

Let x be the k-th smallest item in the tree. Then SPLAYSELECT(k, root) corresponds to the procedure SPLAYSEARCH(x,*root*), which searches for key x in a splay tree rooted at *root*. Since we saw in class that the amortized cost of any search operation on a splay tree is $O(\log n)$, this is also the amortized cost of SPLAYSELECT(k, root).