Solution Set 5: Advanced Data Structures

Problem 1: Joins on Red-Black Trees (15 points, courtesy of CLR, Problem 14-2, p. 278)
The join operation takes two dynamic sets S_1 and S_2 and an element x such that for any $x_1 \in S_1$ and $x_2 \in S_2$, we have $key[x_1] \le key[x] \le key[x_2]$. It returns a set $S = S_1 \cup \{x\} \cup S_2$. Here we'll investigate how to implement this join operation on red-black trees.

- Given a red-black tree T, we store its black-height as the field $bh[T]$. Argue that this field can be maintained by **`RB-Insert`** without requiring any extra storage in the tree and without increasing the asymptotic running times. Show that while descending through T, we can determine the black-height of each node we visit in $O(1)$ time per node visited.

  Recall that the black height of any node (not just the root—all nodes), is defined as the number of black nodes residing along any and all paths from the node to the fringe of its subtree, not including the node itself. Since the number of black nodes from the root to the fringe must be the same along all paths leading either left or right away from the root, the black height of the tree is only affected during insertions that ultimately cause the **root** to undergo either a rotation or to donate its blackness to its two children. Only in the latter case does **`RB-Insert`** actually increase the black-height of the tree, because the resulting red root is made to be black again and hence increases the black-height of every single path because all paths include it. Understand that while descending through the tree, the black-height of all the black nodes is one less than that of its parent, and the black height of a red node is the same as that of its parent. We can update the $bh[T]$ filed in $O(1)$ time if we encode what's necessary to detect a color change at the root, or we can brute-force recalculate the $bh[T]$ in $O(\lg n)$ by just following a path from the root to any leaf $O(\lg n)$ edges away.

We want to implement the operation **`RB-Join(`** T_1**`,`** x**`,`** T_2**`)`**, which destroys T_1 and T_2 and returns a red-black tree $T = T_1 \cup \{x\} \cup T_2$. Let n be the total number of nodes in T_1 and T_2.

- Assume without loss of generality that $bh[T_1] \ge bh[T_2]$. Describe an $O(\lg n)$-time algorithm that finds a black node y in T_1 with the largest key among all those nodes whose black-height is $bh[T_2]$.

Starting with $bh[T_1]$, follow the path from $root[T_1]$ toward the maximum key by travelling across right child pointers, but with the traversal of every black node toward the maximum, understand that the black-height of a black-node is one less than its parent (since the parent can include the black of its child in its black-height, but the child can't.) Keep iterating along the rightmost path until you land on the first black node with a black-height of $bh[T_2]$. Since the algorithm followed only one path, and this path is at most $O(\lg n)$ in length, the algorithm runs in $O(\lg n)$ time.

- Let T_y be the sub-tree rooted at y. Describe how T_y can be replaced by $T_y \{x\} T_2$ in $O(1)$ time without destroying the binary-search-tree property.

  Recall that $key[x_1] \le key[x] \le key[x_2]$, so local modifications can be made to replace T_y with x serving as parent of T_y and T_2. Clearly the binary search tree property won't be violated here, since all keys in T_y are less than the $key[x]$, and all keys in T_2 are greater than $key[x]$. Note that all keys in $T_y \{x\} T_2$ are greater than that of x's parent, since x (and previously everything in T_y) resides/resided in the right subtree of the parent.

- What color should we make x so that so that red-black properties 1, 2, and 4 are maintained? Describe how property 3 can be restored in $O(\lg n)$-time.

  Color it red so that none of the black-heights are affected. A red-red violation is introduced if the parent also happens to be red. (The root of T_y is y and was known to be black, so it's possible that its parent was black, too) The red-red violation can be treated as if it were introduced by the insertion of a node where a fringe red-red violation bubbled up toward the root and placed itself over x and its parent. It can be removed by calling **`RB-Fixup`** and fixed in $O(\lg n)$ time.

- Argue that the running time of **`RB-Join`** is $O(\lg n)$.

  Finding y takes $O(\lg n)$ time, replacing T_y with $T_y \{x\} T_2$ takes constant time, and removing the red-red violation takes $O(\lg n)$ time. Total running time: $O(\lg n)$! Woof!

Problem 2: Listing all Interval Intersections (10 points, courtesy of Exercise 15.3-4)

Given an interval tree T and an interval i, describe how all intervals in T that overlap i can be listed in $O(\min(n,k\lg n))$, where k is the number of intervals in the output list. You may temporarily delete entries from the interval tree if necessary, though a snazzier implementation will neither delete them nor mark previously listed nodes in any particular way.

Leverage off of the interval search from section 15.3 of the text, but each time you find an interval, print it out, and then delete it. Both the search, the deletion, and the insertion needed to put the interval back each take $O(\lg n)$ time. If there are in fact k such intervals, then the running time required to print all overlapping intervals is $O(k\lg n)$.

If however, the number of overlapping intervals $k = \omega(n/\lg n)$, then the running time would be superlinear, and we can't have that. In order to guarantee that the running time is $O(\min(n,k\lg n))$, we can keep a global count of the number of overlapping intervals found, and if it ever exceeds $n/\lg n$, we abort and map over the remaining tree structure via a brute-force inorder traversal, printing out intervals in the process if they overlap the interval of interest.

The snazzier implementation:

```
List-All-Overlaps-In-Tree(T, interval)

   numOverlaps = 0  // global count
   List-All-Overlaps(root[T], interval)

List-All-Overlaps(node, interval)

   if (node = NIL) return;
   if (int[node] overlaps interval)
      then  numOverlaps = numOverlaps + 1
            print int[node]

   if (left[node] != NIL and
       max[left[node]] > low[interval]) // reason to go left
      then numOverlapsPrior = numOverlaps
           List-All-Overlaps(left[x], interval)
           if (numOverlaps > numOverlapsPrior)
              then  List-All-Overlaps(right[x], interval);
      else  List-All-Overlaps(right[x], interval);
```

Problem 3: Minimum Gap between Keys (15 points, courtesy of Exercise 15.3-6)

Show how to maintain a dynamic set Q of numbers that supports the operation **`Min-Gap`**, which gives the magnitude of the difference between the two closest numbers in Q. For example, if $Q = \{1,5,9,15,18,22\}$, then **`Min-Gap(`** Q **`)`** would return 3, because 15 and 18 are nearest neighbors. Make the operations **`Insert`**, **`Delete`**, **`Search`**, and **`Min-Gap`** as efficient as possible, and analyze their running times.

- Underlying data structure:
  A red-black tree in which numbers in the set are simply stored as the key of the nodes. Search is then just the ordinary Tree-Search for binary search trees, which runs in $O(\lg n)$ time on red-black trees.

- Additional information:
  The red-black tree is augmented by the following fields in each node x.
  - **min-gap[x]** contains the minimum gap in the subtree rooted at x. It has the magnitude of the difference between the two closest numbers in the subtree rooted at x. If x is a leaf, then **min-gap[x]** = infinity.
  - **min-val[x]** contains the minimum value (key) in the subtree rooted at x.
  - **max-val[x]** contains the maximum value (key) in the subtree rooted at x.

- Maintaining the information:
  The three fields added to the tree can each be computed from the information in the node and its immediate children. Hence, by Theorem 15.1, they can be maintained during insertion and deletion without affecting the $O(\lg n)$ running time.

$$min\text{-}val[x] = \begin{cases} min\text{-}val[left[x]] & \text{if there's a left subtree} \\ key[x] & \text{otherwise} \end{cases}$$

$$max\text{-}val[x] = \begin{cases} max\text{-}val[right[x]] & \text{if there's a right subtree} \\ key[x] & \text{otherwise} \end{cases}$$

$$min\text{-}gap[x] = \min \begin{cases} min\text{-}gap[left[x]] & (\infty \text{ if no left subtree}) \\ min\text{-}gap[right[x]] & (\infty \text{ if no right subtree}) \\ key[x] - max\text{-}val[left[x]] & (\infty \text{ if no left subtree}) \\ min\text{-}val[right[x]] - key[x] & (\infty \text{ if no right subtree}) \end{cases}$$

  In fact, the reason for defining the **min-val** and **max-val** fields is to make it possible to compute the **min-gap** information stored at the node and its children.

- New operation: **Min-Gap** simply returns the min-gap value stored at the root. Thus, it runs in constant time.