Assignment 3  
Course: COMPSCI 590

Due Date: March 5, 2013

Problem 1:  We wish to perform the following two operations on a set \( X \) of real numbers:

- **INSERT** \((x)\): first delete from \( X \) all numbers not larger than \( x \) and then insert \( x \) into \( X \).
- **FIND-MIN**: return the smallest element of \( X \)

Describe a data structure that supports each of these operations in \( O(1) \) amortized time. *(Hint: Consider using a stack.)*

Problem 2:  Given a binary search tree, add to each node \( v \) an extra attribute \( v\.size \) indicating the number of keys stored in the subtree rooted at \( v \). Let \( \ell(v) \), \( r(v) \) denote the left and right child of \( v \), respectively, and let \( \alpha \) be a constant such that \( 1/2 \leq \alpha < 1 \). A node \( v \) is \( \alpha \)-balanced if \( \ell(v).size \leq \alpha \cdot v.size \) and \( r(v).size \leq \alpha \cdot v.size \). The binary search tree is \( \alpha \)-balanced if every node in the tree is \( \alpha \)-balanced.

In the following, assume that the constant \( \alpha \) satisfies \( 1/2 < \alpha < 1 \). Suppose that **INSERT** is implemented as usual for an \( n \)-node binary search tree, except that after every insertion, if any node in the tree is no longer \( \alpha \)-balanced, then we “rebuild” the subtree rooted at the highest such node in the tree so that it becomes 1/2-balanced. (Note: in this way, at most one “rebuild” is performed at each insertion or deletion)

We use the potential method to analyze the above rebuilding scheme. For a node \( v \) in a binary search tree \( T \), define \( \Delta(v) = |\ell(v).size - r(v).size| \), and define the potential of \( T \) as

\[
\Phi(T) = c \sum_{v \in T : \Delta(v) \geq 2} \Delta(v),
\]

where \( c \) is a sufficiently large constant that depends on \( \alpha \).

1. Argue that any binary search tree has nonnegative potential and a 1/2-balanced tree has potential 0.
2. Suppose that \( m \) units of potential can pay for rebuilding an \( m \)-node subtree. How large must \( c \) be in terms of \( \alpha \) in order for it to take \( O(1) \) amortized time to rebuild a subtree that is not \( \alpha \)-balanced?
3. Show that inserting an item into an \( n \)-node \( \alpha \)-balanced tree costs \( O(\log n) \) amortized time. *(Hint: Refer to [Er:15] for a different analysis of this algorithm.)*

Problem 3:  Any skip list \( \mathcal{L} \) can be transformed into a binary search tree \( T(\mathcal{L}) \) as follows. The root of \( T(\mathcal{L}) \) is the leftmost node on the highest non-empty level of \( \mathcal{L} \); the left and right subtrees are constructed recursively from the nodes to the left and to the right of the root. Let’s call the resulting tree \( T(\mathcal{L}) \) a skip list tree.
(1) Show that any search in $T(\mathcal{L})$ is no more expensive than the corresponding search in $\mathcal{L}$.

(2) Describe an algorithm to insert a new search key into the skip list tree in $O(\log n)$ expected time. Inserting key $x$ into $T(\mathcal{L})$ should produce exactly the same tree as inserting $x$ into $\mathcal{L}$ and then transforming $\mathcal{L}$ into a tree. (Hint: You will need to maintain some additional information in the tree nodes.)

Problem 4: In past lectures, we have seen disjoint-set data structures for maintaining a collection of disjoint sets which support the following two operations:

- $\text{UNION}(x, y)$: merges the sets that contain $x$ and $y$ into a new set that is the union of these two sets.
- $\text{FIND-SET}(x)$: returns a pointer to the representative of the (unique) set containing $x$.

Now suppose it is known that all union operations will be performed before all find-set operations. Describe an implementation of a disjoint-set data structure such that each of the UNION and FIND-SET operations takes $O(1)$ amortized time.

Problem 5: Let $X$ be a set of $n$ intervals on the real line. We say that a set $P$ of points stabs $X$ if every interval in $X$ contains at least one point in $P$. Describe and analyze an efficient algorithm to compute the smallest set of points that stabs $X$. Assume that your input consists of two arrays $X_L[1..n]$ and $X_R[1..n]$, representing the left and right endpoints of the intervals in $X$. 