1. (1 Point) TRUE or FALSE: Recall that a rotation can be implemented with 6 pointer assignments. Suppose a binary search tree maintains successor and predecessor links (denoted \texttt{u.succ} and \texttt{u.pred} in the text). Now rotation requires 12 pointer assignments.

\textbf{SOLUTION} \hspace{1em} FALSE. Rotation does not affect the successor and predecessor links of a binary search tree.

\textbf{Comments:}

2. (4 Points) The text gave a conventional algorithm for successor of a node in a BST. Give the rotation-based version of the successor algorithm.

\textbf{SOLUTION} \hspace{1em} Suppose we want to find the successor of node \texttt{u}. There are two cases:
\begin{enumerate}
\item[(A)] \texttt{u} has a non-nil right child, \texttt{v = u.right}. Then as long as \texttt{v.left} \neq \texttt{nil}, we rotate \texttt{v.left}. When \texttt{v.left} = \texttt{nil}, we return \texttt{v} as the successor of \texttt{u}.
\item[(B)] Otherwise, either \texttt{u} has no successor or it is the tip of the left-spine of its successor. How to tell which is the case? Well, there are two easy cases:
\item[(B1)] If \texttt{u} is the root, then \texttt{u} must have no successor
\item[(B2)] If \texttt{u} is a left child, then \texttt{u.parent} must be the successor.
\end{enumerate}
To see why (B1) and (B2) are true, recall that \texttt{u} has no right child. Here is the simple algorithm to decide which of the two cases hold. It amounts to producing the situation where (B1) or (B2) holds:
As long as \texttt{u} is a right child, we keep rotating \texttt{u}. Recall that rotation does not change successor relations. When we stop, either (B1) or (B2) holds. or (B2) \texttt{u} is a right child of its parent \texttt{p = u.parent}. If (B1), return \texttt{nil}, and if (B2), return \texttt{u.parent}.

\textbf{Comments:} Why would we be interested in the rotational version of successor? Well, we can obtain amortized complexity bounds using this approach (not exactly as we described, but using Splay tree concepts from Chapter 6).

3. (3 Points) Give the in-order, pre-order and post-order listing of the nodes in the tree in Figure 15 (Lecture II).

\textbf{SOLUTION} \hspace{1em} INORDER: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16.
PRE-ORDER: 12, 5, 3, 1, 2, 4, 8, 7, 6, 10, 9, 11, 14, 13, 15, 16.
POST-ORDER: 2, 1, 4, 3, 6, 7, 9, 11, 10, 8, 5, 13, 16, 15, 14, 12.

\textbf{Comments:}

4. (6 Points) Give a recursive routine called \texttt{CheckBST(u)} which checks whether the binary tree \texttt{T_u} rooted at a node \texttt{u} is a binary search tree (BST). You must figure out the information to be returned by \texttt{CheckBST(u)}; this information should also tell you whether \texttt{T_u} is BST or not. Assume that each non-nil node \texttt{u} has the three fields, \texttt{u.key}, \texttt{u.left}, \texttt{u.right}.
SOLUTION Let $\max\{T_u\}$ denote the maximum key in $T_u$ (similarly for $\min\{T_u\}$). Then $T_u$ is a BST iff $u = nil$ or:

1. $T_u.left$ is a BST
2. $T_u.right$ is a BST
3. $\max\{T_u.left\} < u.key \leq \min\{T_u.right\}$

The problem with (3) is that we need to make these inequalities true even when $u.left = nil$ or $u.right = nil$. To achieve this, we define $\max\{T_u.left\} = -\infty$ when $u.left = nil$ and define $\min\{T_u.right\} = -\infty$ when $u.right = nil$.

We use postorder traversal. Let $CheckBST(u)$ take a node $u$ as argument, and returns a pair $[minkey, maxkey]$ of numbers. When $T_u$ is a BST, $minkey$ is the smallest key in the subtree $T_u$ and $maxkey$ is the largest key in $T_u$. There are two special cases: in case $u = nil$, we return $[minkey, maxkey] = [+\infty, -\infty]$. In case $T_u$ is not a BST, we return $[minkey, maxkey] = [-\infty, +\infty]$. Then our algorithm is:

\[
\text{CheckBST}(u) \\
\hspace{1em} \triangleright \text{Base Case:} \\
\hspace{2em} \text{if (} u = \text{nil) \ Return(}[+\infty, -\infty]) \\
\hspace{1em} \triangleright \text{Recursive Case:} \\
\hspace{2em} [Lmin, Lmax] \leftarrow \text{CheckBST}(u.left) \\
\hspace{2em} [Rmin, Rmax] \leftarrow \text{CheckBST}(u.right) \\
\hspace{1em} \triangleright \text{If not BST:} \\
\hspace{2em} \text{if (} (Lmax > u.key) \text{ or } (Rmin \leq u.key) \text{)} \\
\hspace{3em} \text{else Return(}[-\infty, +\infty]) \\
\hspace{2em} \text{else} \\
\hspace{3em} \triangleright \text{If BST:} \\
\hspace{4em} \text{Return}(\min\{Lmin, u.key\}, \max\{Rmax, u.key\}) \\
\hspace{5em} \triangleright \text{This takes care of the cases } Lmin = +\infty \text{ and } Rmax = -\infty
\]

Comments: REMARKS:

1. Do not make the standard mistake of thinking that a BST only has to satisfy the relationship $u.left.key \leq u.key < u.right.key$ for all node $u$.
2. Second, we said that most properties about binary trees ought to be done recursively by structural induction. So a post order traversal is the best solution.
3. As for the base case, we feel that it is always better to use nil. The alternative is to treat a tree with only one node as base case.
4. Try to re-write $CheckBST(u)$ in the style of our postorder traversal algorithm, based on the shell routines BASE and VISIT.

5. (4 Points) Draw an AVL $T$ with minimum number of nodes such that the following is true: there is a node $x$ in $T$ such that if you delete this node, the AVL rebalancing will require two rebalancing acts. Note that a double-rotation counts as one, not two, rebalancing act. Draw $T$ and the node $x$.

SOLUTION $T$ has height 4, and has 12 nodes.

Comments: This was in your midterm. So if you had done hw4 before the midterm (since this hw was intended to give some practice for midterm), you would have had an easier time.

6. (6 Points) Insert into an initially empty AVL tree the following sequence of keys: $1, 2, 3, \ldots, 14, 15$.

(a) Draw the trees at the end of each insertion as well as after each rotation or double-rotation. [View double-rotation as an indivisible operation].

(b) Prove the following: if we continue in this manner, we will have a complete binary tree at the end of inserting key $2^n - 1$ for all $n \geq 1$.
(a) Omitted.

(b) After we inserting the keys $1, 2, \ldots, 2^n - 1$ into the tree, we claim that the result is a complete binary tree of height $n - 1$. The basis cases hold, when $n = 0, 1$.

Inductively: suppose that the result is true for $2^n - 1$ ($n \geq 1$). We must prove the result for $2^{n+1} - 1$. Consider what happens when we insert the next $2^n$ keys into the perfect binary tree $T_n$ containing the keys $1, 2, \ldots, 2^n - 1$: view $T_n$ as a root $v$ with two subtrees, $A$ and $B$, where $A$ and $B$ are complete binary trees of heights $n - 2$. We suggest you draw diagrams to follow along this argument:

- After the next $2^{n-1}$ keys are inserted, by induction, the tree $B$ will be transformed into a perfect binary tree of height $n - 1$. Let this subtree be denote $B'$. Now, view $B'$ as a root $u$ with two subtrees $C$ and $D$, each perfect binary tree with height $n - 2$. Assume $A < C < D$ (i.e., the keys in $A$ are less than the keys in $C$, etc). At this point, the tree size is $2^n - 1 + 2^{n-1}$.

- When we insert the next item, this will transform $D$ into $D'$ with height $n - 1$. That means the root $v$ of the entire tree (which now has $2^n + 2^{n-1}$ keys) is unbalanced. This will force a rotation at $u$. After rotation, $u$ is the root, and $v$ the left child.

- Now, if we continue with inserting the next $2^{n-1} - 1$ keys, by induction, the tree $D'$ will become a perfect balanced tree of height $n - 1$. Call this tree $E$.

- At this point, the tree is rooted at $u$, left child is $v$. The left subtree of $v$ is $A$ and right subtree of $v$ is $C$. The right subtree of $u$ is $E$. This is a perfect binary tree of height $n$, with $2^{n+1} - 1$ keys.

Comments:

7. (4 Points) Draw two AVL trees, with $n$ keys each: the two trees must have different heights. Make $n$ as small as you can.

SOLUTION You just try small values of $n$ – e.g., for $n = 4$, all AVL trees with 4 keys must have height 2. You will soon discover that the smallest possible value of $n$ is $n = 7$. We could get AVL trees of heights either 2 or 3, as shown in Figure 1.

Comments:

SOLUTION FIGURE

Figure 1: Two AVL trees with 7 nodes: with heights 2 and 3, respectively

8. (12 Points) Relaxed AVL Trees

Let us define AVL(2) balance condition to mean that at each node $u$ in the binary tree, $|balance(u)| \leq 2$.

(a) Derive an upper bound on the height of a AVL(2) tree on $n$ nodes.

(b) Give an insertion algorithm that preserves AVL(2) trees. Try to follow the original AVL insertion as much as possible; but point out differences from the original insertion.

(c) Give the deletion algorithm for AVL(2) trees. SOLUTION FIGURE
SOLUTION

(a) Imitating the lecture notes, define $\mu(h)$ to be the minimum number of nodes in a relaxed AVL tree of height $h$. In general, we have the recurrence

$$\mu(h) = 1 + \mu(h-1) + \mu(h-3).$$  \hspace{1cm} (1)

We assume the recurrence holds for $h \geq 2$. The boundary conditions are

$$\mu(-1) = 0, \mu(0) = 1, \mu(1) = 2.$$  

Thus, $\mu(2) = 3$ and $\mu(3) = 5$. It is easy enough to get a rough lower bound for $\mu(h)$: from (1), we see that

$$\mu(h) > 2\mu(h-3).$$

Thus, the domain variable $h$ drops 3 units at each step. So in $h/3$ steps, you should reach 0. Each step give you a factor of 2, so this shows

$$\mu(h) > 2^{h/3}.$$  

For basis, you may check that this is true for $h \leq 3$.

You may have noticed that what we are doing is basically domain and range transformation.

Now we address the question: suppose $T$ has $n$ nodes and height $h$. Then we know that $n \geq \mu(h) > 2^{h/3}$. Taking logs, we get $\lg n > h/3$ or $h < 3 \lg n$. 
(b) As in AVL trees, the insertion has two phases: standard insertion phase followed by rebalancing phase. In rebalancing, we move along the root path starting from the newly inserted node, and looking for an unbalanced node. Let \( u \) be the first unbalanced node. Thus, one of its children has height \( h - 1 \) and the other child \( v \) has height \( h + 2 \), for some \( h \geq 0 \). So \( u \) itself has height \( h + 3 \). It is clear the original height of \( u \) was \( h + 2 \). Let \( w \) be the child of \( v \) of height \( h + 1 \). It is clear that the sibling of \( w \) has height \( h \pm \delta \) for some \( \delta = 0, 1 \). We consider two cases, (I.a) and (I.b), that are very similar to the original AVL tree insertion cases. See Figure 2.

Case (I.a): \( w \) is an outer grandchild of \( u \). After rotating at \( v \), we would achieve balance at \( v \) and the height of \( v \) is \( h + 2 \). So this is a terminal case.

Case (I.b): \( w \) is an inner grandchild of \( u \). Let the two children of \( w \) have heights \( h - \delta' \) and \( h - \delta'' \) where \( \delta', \delta'' \in \{0, 1, 2\} \). Moreover, \( \delta' \delta'' = 0 \). After a double rotation at \( w \), we find that \( w \) takes the place of \( u \) as the root the previously unbalanced subtree. But \( w \) is now balanced and has height \( h + 2 \). Again this is terminal.

Note that there is basically NO qualitative difference in the behavior of this algorithm and the standard AVL insertion.

(c) Again, as in AVL trees, the two cases of insertion translates into an identical analysis for deletion, with the exception that these cases are non-terminal: one has to continue checking if other nodes in the root path are unbalanced. Call these Cases (D.a) and (D.b), respectively. But now there is a third possibility, Case (D.c): \( u \) is unbalanced, one child of \( u \) has height \( h - 1 \) and the other child \( v \) has height \( h + 2 \). Moreover, the two children of \( v \) have heights \( h + 1 \). In this case, we perform a single rotation at \( v \) and terminate.

COMMENTS: For (a), you can get better bounds, replacing \( \sqrt[3]{2} \) by \( \phi \) that is the positive root of the polynomial \( x^3 - x^2 - 1 \), analogous to the Golden ratio. For (b), it is somewhat surprising to see that the case analysis turns out to mirror the standard AVL tree algorithms for both insertions and deletions!

9. (0 Points) Let \( T \) be the AVL tree in Figure 3(a) (page 11, Lect.II). This calls for hand-simulation of the insertion and deletion algorithms. Show intermediate trees after each rotation, not just the final tree.
   (a) Delete the key 10 from \( T \).
   (b) Insert the key 2.5 into \( T \). This question is independent of part (a).

   Re-do parts (a) and (b), but using the AVL tree in Figure 3(b) instead.

10. (0 Points) Describe what changes is needed in our binary search tree algorithms for the exogenous case.