CSE 100 - Homework 1

4.18

(a) Give a precise expression for the minimum number of nodes in an AVL tree of height h.

(b) What is the minimum number of nodes in an AVL tree of height 15?

Solution:

(a) Let $S(h)$ be the minimum number of nodes in an AVL tree T of height h. The subtrees of an AVL tree with mimimum number of nodes must also have minimum number of nodes. Also, at least one of the left and right subtrees of T is an AVL tree of height $h-1$. Since the height of left and right subtrees can differ by at most 1, the other subtree must have height $h-2$. Then, we have the following recurrence relation:

$$S(h) = S(h-1) + S(h-2) + 1. \tag{1}$$

We also know the base cases: $S(0) = 1$ and $S(1) = 2$.
One method to solve a recurrence relation is to guess the solution and prove it by induction. Observe that the recurrence relation in (1) is very similar to the recurrence relation of Fibonacci numbers. When we look at the first a few numbers of the sequence $S(h)$, it is not difficult to guess $S(h) = F(h+3) - 1$. Now, let's prove that $S(h) = F(h+3) - 1$ by induction.

Base cases:
$S(0) = 1$, $F(3) = 2$. So, $S(0) = F(3) - 1$.
$S(1) = 2$, $F(4) = 3$. So, $S(1) = F(4) - 1$.

Induction hyprothesis:
Assume that the hypothesis $S(h) = F(h+3) - 1$ is true for $h = 1, \cdots, k$.

Inductive step:
Prove that it is also true for $h = k+1$.

$$\begin{aligned} S(k+1) &= S(k) + S(k-1) + 1 \\ &= F(k+3) - 1 + F(k+2) - 1 + 1 \\ &= F(k+4) - 1. \end{aligned}$$

We replace $S(k)$ and $S(k-1)$ with their equivalence according to the hypothesis. Then, we get $S(k+1) = F(k+4) - 1$. Hypothesis is also true for $h = k+1$. Thus, it is true for all h.

(b) $S(15) = F(18) - 1$.

4.19
Show the result of inserting $2, 1, 4, 5, 9, 3, 6, 7$.

Solution:
See Figure 1.

4.20 The keys $1, 2, \cdots, 2^k - 1$ are inserted in order into an initially empty AVL tree. Prove that the resulting tree is perfectly balanced.

Solution:
We will prove the following more general statement by induction on k: The result of inserting any increasing sequence of $2^k - 1$ numbers into an initially empty AVL tree results in a perfectly balanced tree of height $k-1$.

Base case:
$k = 1$. Tree has only one node. This is clearly perfectly balanced.

Induction hypothesis: Assume hypothesis is true for $k = 1, 2, ..., h$.

Inductive step: Prove that it is true for $k = h + 1$, i.e. for the sequence $1, 2, \cdots, 2^{h+1} - 1$.

After the first $2^h - 1$ insertions, by the induction hypothesis, the tree is perfectly balanced, with height $h - 1$. 2^{h-1} is at the root; the left subtree is a perfectly balanced tree of height $h-2$, and the right subtree is a perfectly balanced tree containing the numbers $2^{h-1} + 1$ through $2^h - 1$, also of height $h - 2$. Each of the next 2^{h-1} insertions (2^h through $2^h + 2^{h-1} - 1$) are inserted into the right subtree, and the entire sequence of numbers in the right subtree (now $2^{h-1} + 1$ through $2^h + 2^{h-1} - 1$) were inserted in order and are a sequence of $2^h - 1$ nodes. By the induction hypothesis, they form a perfectly balanced tree of height $h - 1$. See Figure 2.

The next insertion, of the number $2^h + 2^{h-1}$, imbalances the tree at the root. The subsequent RR rotation forms a tree with root 2^h at the root, and a perfectly balanced left subtree of height $h - 1$. The right subtree consists of a perfectly balanced tree (of height $h - 2$), with the new node (containing $2^h + 2^{h-1}$ as the right child of its biggest element. Thus, the right subtree is as if the numbers $2^h + 1, \cdots 2^h + 2^{h-1}$ had been inserted in order. By the induction hypothesis, subsequently inserting the numbers $2^h + 2^{h-1} + 1$ through $2^{h+1} - 1$ nodes form a perfectly balanced subtree of height $h-1$. Since the left and right subtrees are perfectly balanced (height $h-1$), the whole tree is perfectly balanced.

4.36 Write a method to generate a perfectly balanced binary search tree of height h with keys 1 through $2^{h+1} - 1$. What is the running time of your method?

Solution:
One way to implement is the following:

```
public BSTNode genPerfectTree (int h)
{
    return genPefectTree (1, Math.pow(2, h + 1) - 1);
}

private BSTNode genPerfectTree (int low, int high)
{
   if (low==high)
       return new BSTNode(low, null, null);
   int root = (low + high) / 2;
   BSTNode left = genPerfectTree (low, root - 1);
   BSTNode right = genPerfectTree (root + 1, high);
   return new BSTNode(root, left, right);
}
```

The running time of this is $O(2^h) = O(n)$, where n is the number of nodes in the generated tree. Every node is visited just one time when it is generated.

4.47
Teo trees T_1 and T_2 are isomorphic if T_1 can be transformed into T_2 by swapping left and right children of (some of) nodes in T_1.

(a) Give a polynomial time algorithm to decide if two trees are isomorphic.

(b) What is the running time of your program?

Solution:
One way to implement is the following:

```
public boolean areIsomorphic (BinaryTreeNode root1, BinaryTreeNode root2)
{
   /* If both are null trees, they are isomorphic. */
   if (root1 == null && root2 == null)
      return true;

   /* If one is null and the other not, they are not isomorphic. */
   if (root1 == null || root2 == null)
      return false;

   /* If the elements at the roots are not the same, they are
   not isomorphic. */
   if (root1.element.compareTo(root2.element) != 0)
      return false;

   /* Now, to be isomorphic,
   root1.left must be isomorphic to root2.left AND
   root1.right must be iosmorphic to root2.right,
   OR
   root1.right must be isomorphic to root2.left AND
   root1.left must be iosmorphic to root2.right. */

   return (areIsomorphic(root1.left, root2.left) &&
      areIsomorphic(root1.right, root2.right))
      || (areIsomorphic(root1.right, root2.left) &&
      areIsomorphic(root1.left, root2.right));
}
```

Worst case time complexity, as a function of the height of the smaller of the two trees, can be written by the following recurrence relation.

$T(-1) = c1$ // height of -1 means an empty tree.
$T(h) = 4 \cdot T(h-1) + c2$// up to four recursive calls for trees smaller in height by 1.

We can solve this recurrence relation by back substitution.

$$\begin{aligned}
T(h) &= 4 \cdot T(h-1) + c2 \\
&= 4 \cdot (4 \cdot T(h-2) + c2) + c2 \\
&= 4 \cdot (4 \cdot (4 \cdot T(h-3) + c2) + c2) + c2 \\
&= 4^3 \cdot T(h-3) + c2 \cdot (1 + 4 + 4^2) \\
&= 4^i \cdot T(h-i) + c2 \sum_{k=0}^{i-1} 4^k \\
&= 4^i \cdot T(h-i) + c2 \cdot (4^i - 1)/(4-1) \\
&= 4^i \cdot T(h-i) + (c2/3) \cdot (4^i - 1)
\end{aligned}$$

Stop when $h - i = -1$ (the base case), i.e. $i = h + 1$.

$$\begin{aligned} T(h) &= 4^{h+1} \cdot T(-1) + (c2/3) \cdot (4^{h+1} - 1) \\ &= c1 \cdot 4^{h+1} + (c2/3) \cdot (4^{h+1} - 1) \\ &= (c1 + c2/3) \cdot 2^{2(h+1)} - (c2/3) \end{aligned}$$

Thus, the running time $T(h)$ is exponential in h, i.e. $O(2^h)$. However, if $h = O(logn)$, then it is $O(n^2)$.

Extra Problem
Prove that insertion into a binary search tree that is complete while maintaining completeness can not be done in $O(logn)$ time.

Solution:
We will show that for any tree height, we can create a perfectly balanced binary search tree that can not be inserted into in $O(logn)$ time, while leaving the tree perfectly balanced. For this problem, our perfectly balanced binary tree is a tree completely filled in, except possibly for the bottom level, which is filled from left to right.

First, we show that if a sequence of consecutive integers of length $2^k - 1$ starting at x is in a perfectly balanced tree, then the leaves of the tree are numbered $x, x + 2, x + 4, \cdots, x + 2^k - 2$. So, for example, if $k = 3$, and $x = 1$, the leaves are numbered $1, 3, 5$, and 7. See figure 3.

We will prove this by induction on k.

Base case: $k = 1$
Since there is just one integer x in the sequence, it must be a leaf.

Induction hypothesis:
Assume it is true for $k = 1, \cdots, h$.

Induction step:
Prove that it is true for $k = h + 1$. The numbers stored in the tree are $x, x + 1, \cdots, x + 2^{h+1} - 2$. The root stores the integer $((x + 2^(h + 1) - 2) + x)/2 = x + 2^h - 1$. The left subtree stores the numbers $x, x + 1, \cdots, x + 2^h - 2$. This tree has $(x + 2^h - 2) - x + 1 = 2^h - 1$ nodes, so the induction hypothesis applies, and the leaves of this tree are $x, x + 2, \cdots, x + 2^h - 2$.

The right subtree stores the numbers $x + 2^h, x + 2^h + 2, \cdots, x + 2^(h + 1) - 2$. This tree has $(x + 2^(h + 1) - 2) - (x + 2^h) + 1 = 2^h - 1$ nodes, so the induction hypothesis applies, and the leaves of this tree are $x + 2^h, x + 2^h + 2, \cdots, x + 2^(h + 1) - 2$.

Thus, the sequence of the leaves in the entire tree is $x, x+2, \cdots, x+2^h-2, x+2^h, x+2^h+2, \cdots, x+2^{h+1}-2$ and we have proven the statement for $k = h + 1$.

Now consider the perfectly balanced tree containing the numbers $1, 2, ..., 2^h - 1$. Its leaves will be numbered $1, 3, 5, \cdots, 2^h - 1$. Now remove the bottom right node from this tree (the one numbered $2^h - 1$), and insert into the tree the number 0. The numbers stored will be $0, 1, 2, \cdots, 2^h - 2$. Since there are still $2^h - 1$ nodes, the above proof applies, and the leaf nodes will be numbered $0, 2, 4, \cdots, 2^h - 2$. Since the leaf nodes are the only ones that have null pointers in a perfectly balanced tree, and since all of the leaf nodes are changing during the process of inserting 0, whatever algorithm does the insertion will have to visit all of the leaf nodes. Since there are $O(n)$ leaf nodes, the algorithm must have worst case $O(n)$ time complexity.

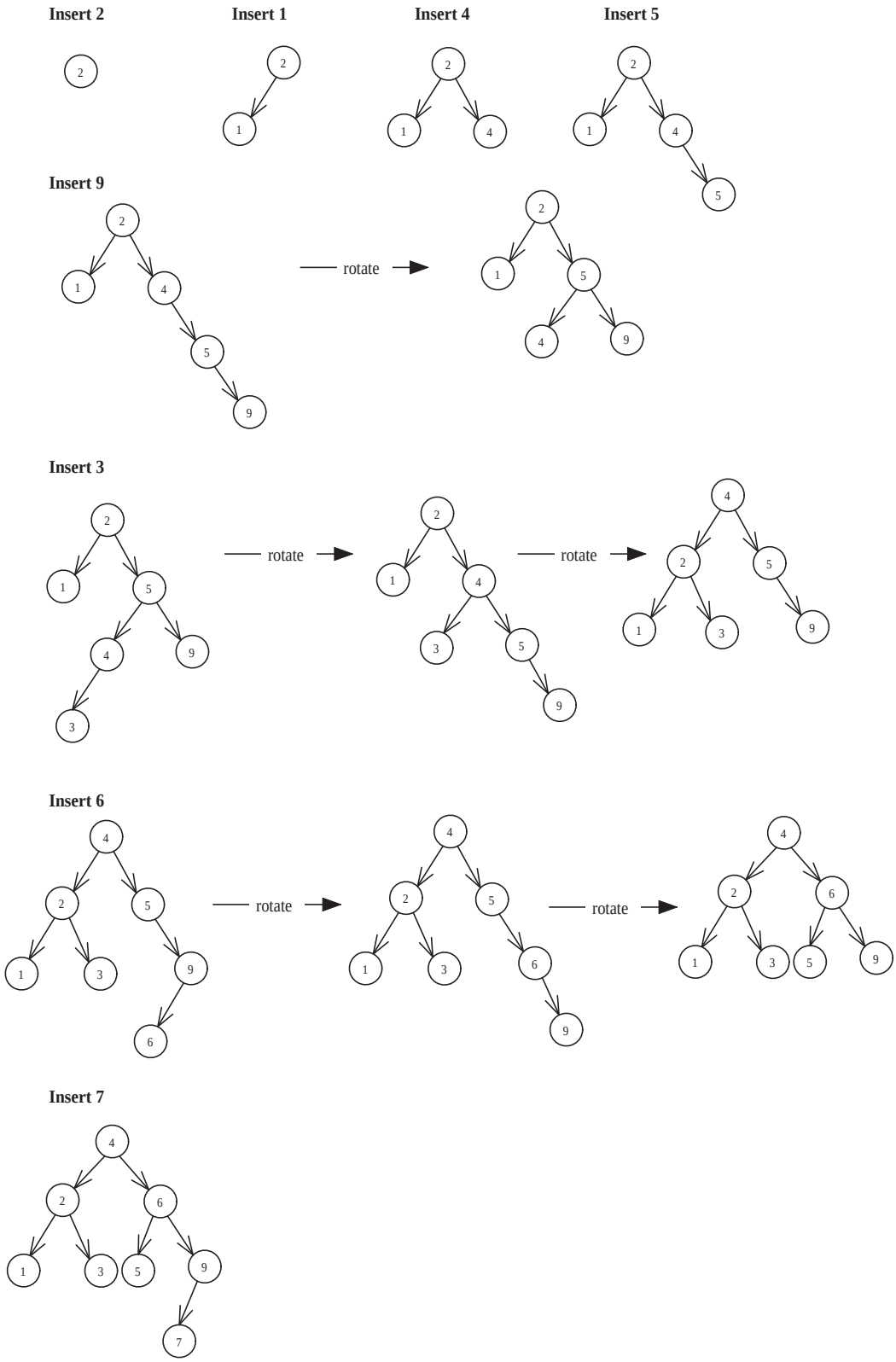
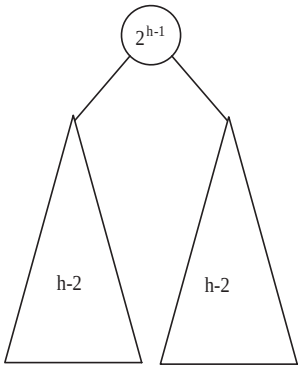
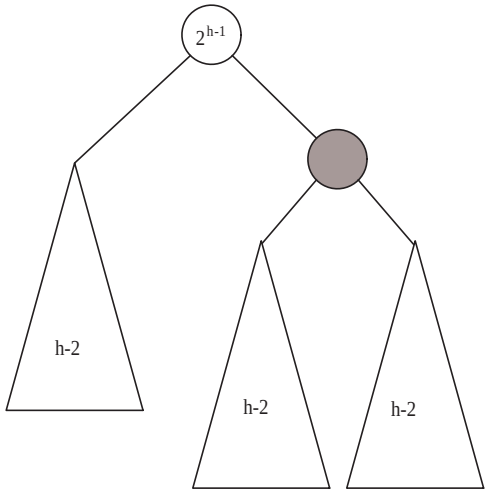


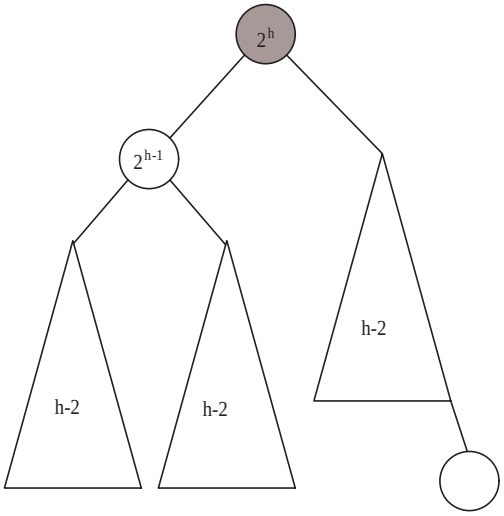
Figure 1: Problem 4.19



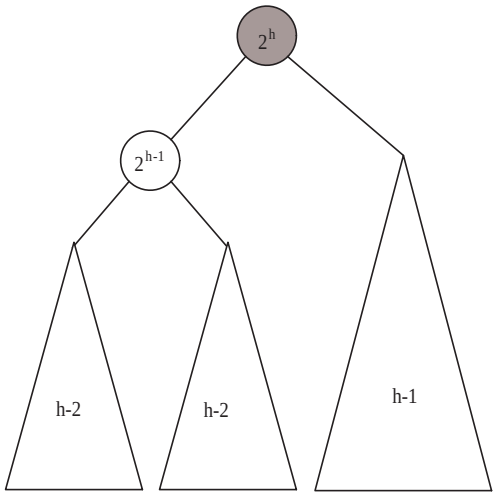
after inserting 1, 2, ..., 2^h-1.



after inserting 1, 2, ..., 2^h-1, ..., 2^h+2^{h-1}-1.



after inserting 1, 2, ..., 2^h-1, ..., 2^h+2^{h-1}.



after inserting 1, 2, ..., 2^h-1, ..., 2^h+2^{h-1}, ... 2^{h+1}-1.

Figure 2: Problem 4.20