Data Structures and Algorithms — CSCI 230
Chapter 4 — Review Solutions

1. Suppose each non-leaf node of a m-ary has exactly m children — in other words, each node of the tree has either exactly 0 or exactly m children. Suppose we build such a tree of height $h \geq 0$ having the **minimum** possible number of nodes. What is the number of nodes in this "minimal" tree? Prove your answer using mathematical induction.

   SOLUTION:
   The answer is

   $$mh + 1.$$

   The proof is by induction on $h \geq 0$.

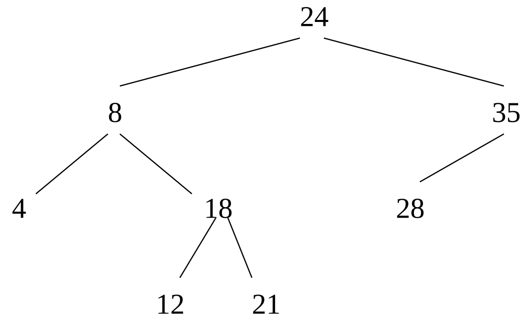
   Basis case: For $h = 0$, the formula becomes $m \cdot 0 + 1 = 1$. Since there is only one node in any tree of height 0, the basis case is established.

   Induction step: For all $h > 0$, if there are $m(k-1)+1$ nodes in a minimal m-ary tree of height $0 \leq k < h$, then consider a minimal m-ary tree of height h. Remove the root of this tree. This creates m new trees, $m-1$ of which have 1 node and one of which is a minimal m-ary of height $h-1$. If either statement were false, the original tree would not be minimal — it would have nodes that could be removed from it and it would still satisfy the conditions. Therefore, by the induction hypothesis, the original tree had

   $$m(h-1) + 1 + (m-1) + 1 = m(h-1) + m + 1 = mh + 1$$

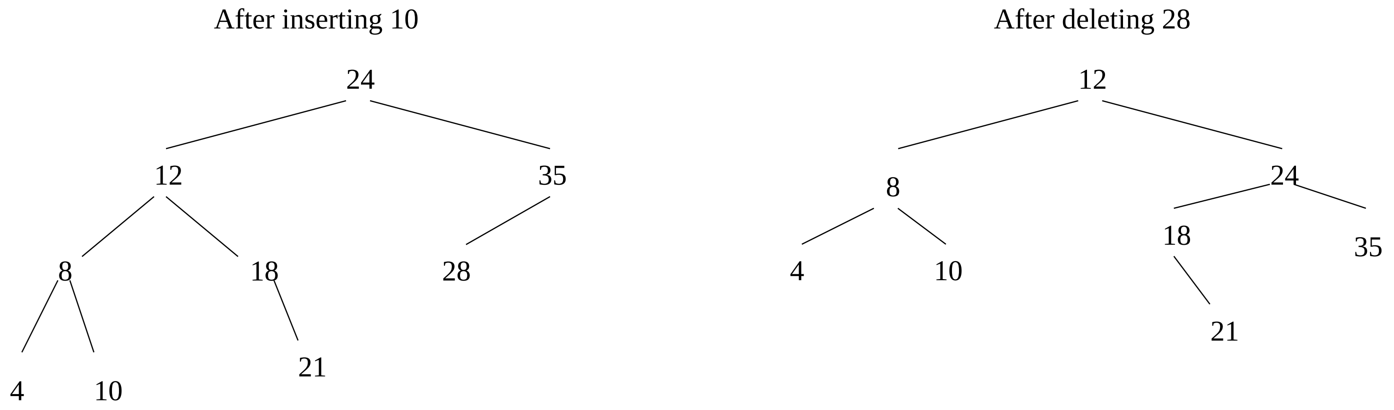
   nodes.

2. Consider the following binary search tree.



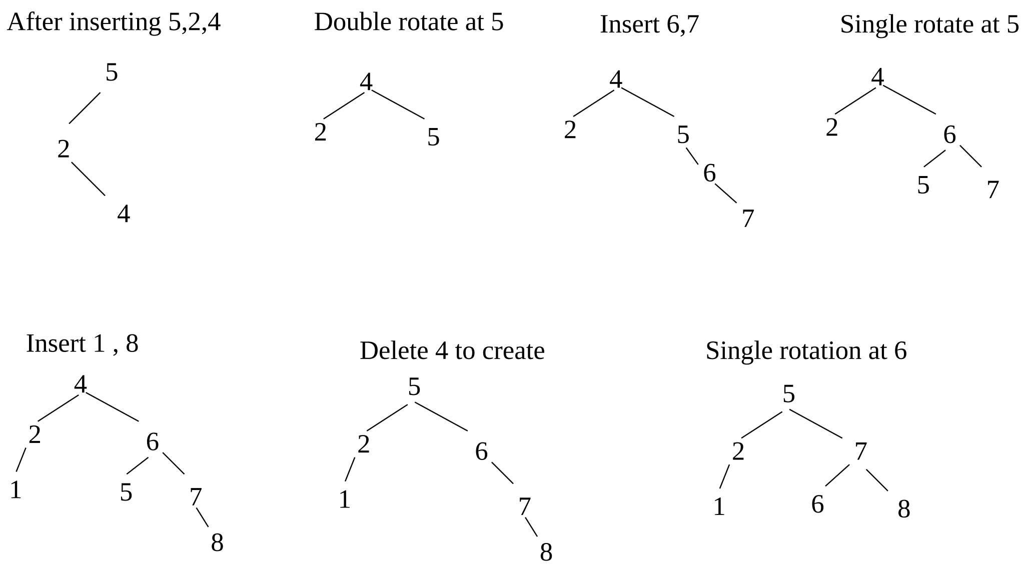
   Assuming it is an AVL tree, show the state of the tree after inserting 10 and then again after deleting 28.

SOLUTION:



3. Given an empty AVL tree of integers, show the structure of the tree after each of the values 5, 2, 4, 6, 7, 1, 8 is inserted and then show the change to the resulting tree when 4 is deleted. Indicate where rotations are done to rebalance the tree.

SOLUTION:



4. Given a binary search tree containing n floating point values and having a height that is $O(\log n)$ (i.e. it is balanced), write an algorithm to count the number of nodes storing values greater than `x0` and less than `x1`. What

is the running time of your algorithm? (More credit will be given for an efficient algorithm.)

You may assume the following declaration for tree nodes:

```
struct TreeNode {
   float x;
   TreeNode * left;
   TreeNode * right;
}
```

Start from the following prototype and assume the function is initially passed a pointer to the root:

```
int Count( TreeNode * T, float x0, float x1 )
```

Solution:

```
{
   if ( T == NULL )
      return 0;
   else if ( T->x <= x0 )
      return Count( T->right, x0, x1);
   else if ( T->x >= x1 )
      return Count( T->left, x0, x1);
   else
      return 1 + Count( T->left, x0, x1 ) +
                 Count( T->right, x0, x1 );
}
```

Showing that this is in fact $O(\log n + k)$ is a bit tricky, which is why I didn't require you to do this. The basic idea is to count the recursive calls. For each value between `x0` and `x1` two recursive calls are made. These are calls made at the `else` statement, resulting in the k part of the $O(\log n + k)$. There are at most $O(\log n)$ recursive calls made following the `else if` statements. These correspond to walking down the left and right boundaries of the region of the tree containing the values between `x0` and `x1`.

5. It is possible to turn a sorted array of N floating point values into an AVL tree that is as balanced as possible in $O(N)$ time. Write a function to do this. (Hint: think about which value should be stored at the root.) Assume the following structure declaration for AVL nodes.

```
struct Avl_Node {
  float Element;
  Avl_Node *Left;
  Avl_Node *Right;
  int Height;
};
```

Your function should calculate the heights of the nodes and it should return a pointer to the root of the tree. You do not need to prove that the result is an AVL tree and you do not need to prove the function requires $O(N)$ time. Here is a prototype

```
Avl_Node * ArrayToAVLTree( float values[], int N  )
```

Solution:

```
Avl_Node * ArrayToAVLTree( float values[], int N  )
{
   return ArrayToAVLTree( values, 0, N-1 );
}

Avl_Node * ArrayToAVLTree( float values[], int low, int high )
{
    if ( low > high ) return NULL;
    int mid = (low+high)/2;
    Avl_Node * node = new Avl_Node;
    node->Element = values[mid];
    node->Left = ArrayToAVLTree( values, low, mid-1 );
    node->Right = ArrayToAVLTree( value, mid+1, high );
    int h1 = ( node->Left == NULL ? -1 : node->Left->Height );
    int h2 = ( node->Right == NULL ? -1 : node->Right->Height );
    node->Height = 1 + max(h1,h2);
    return node;
}
```