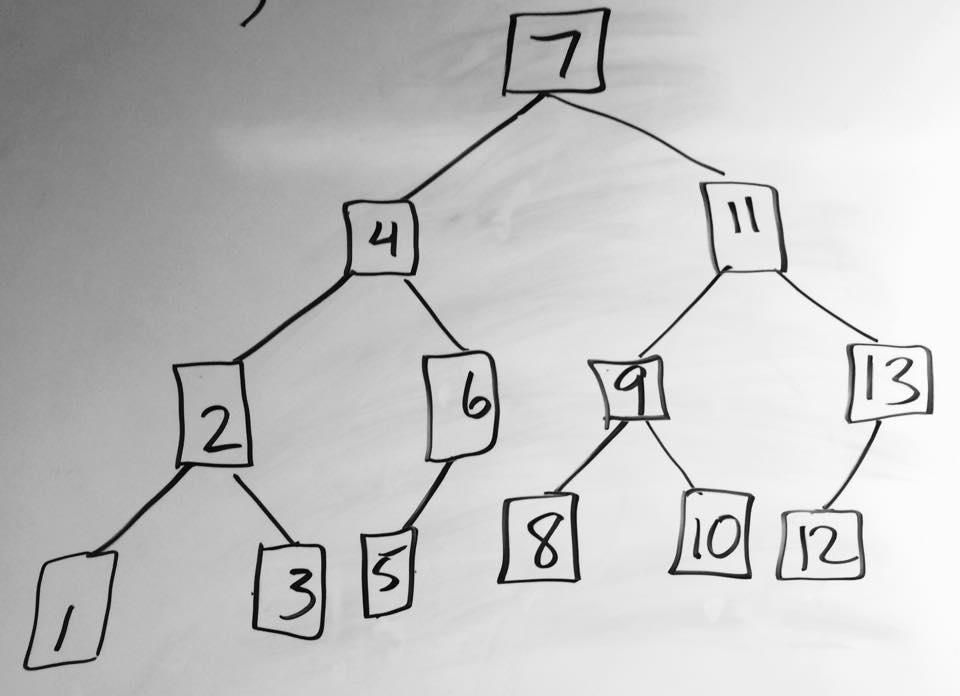
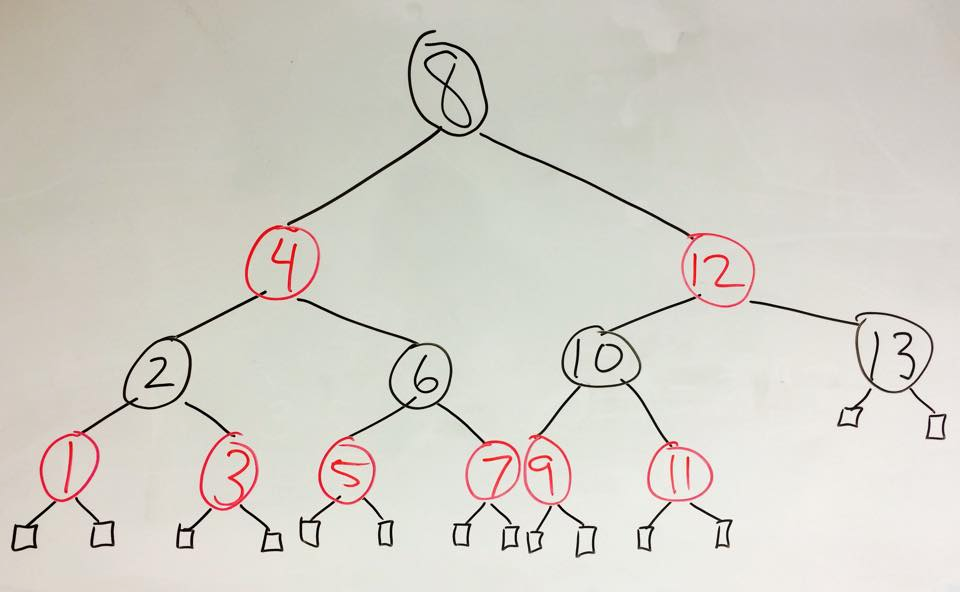
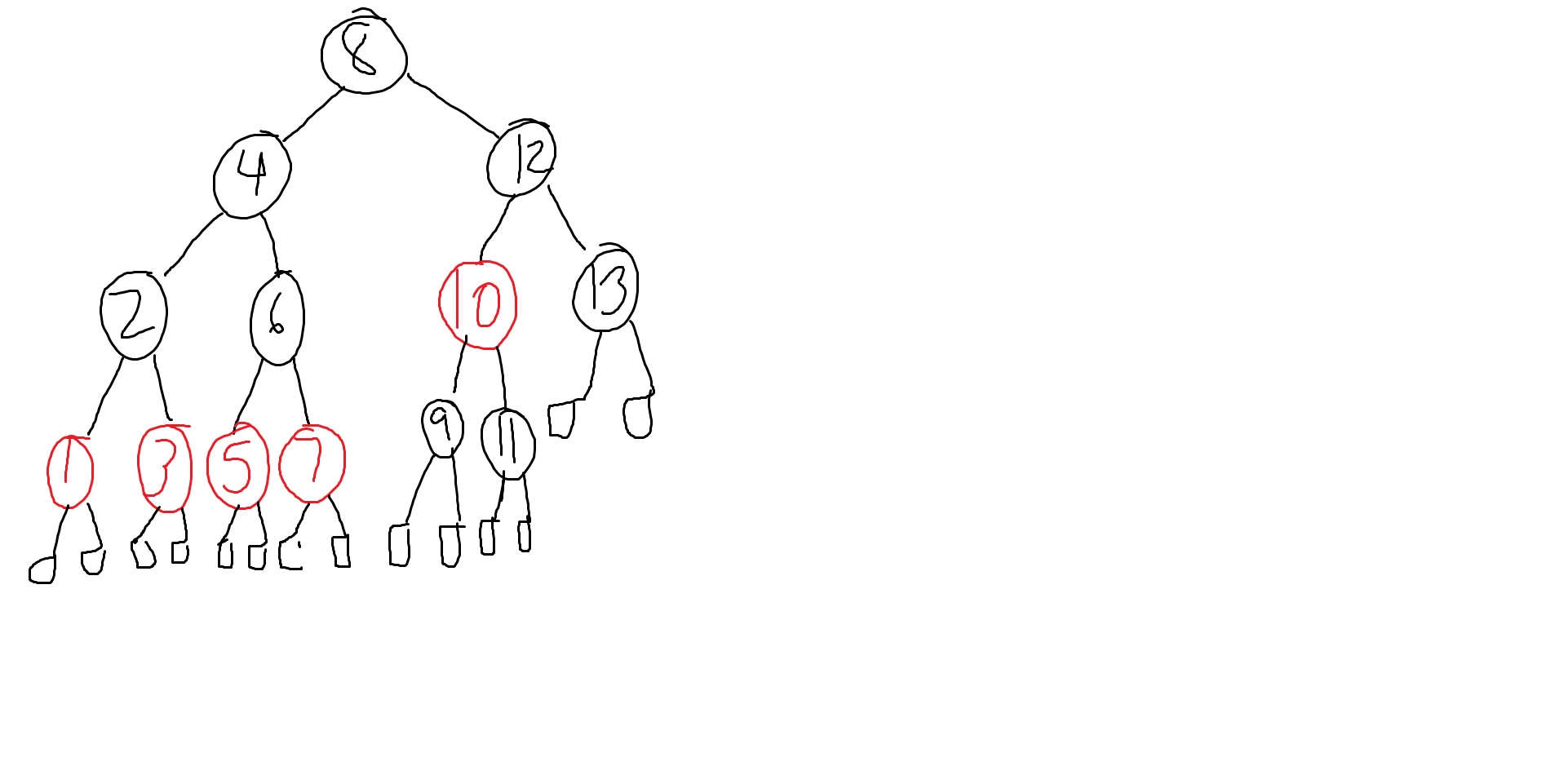
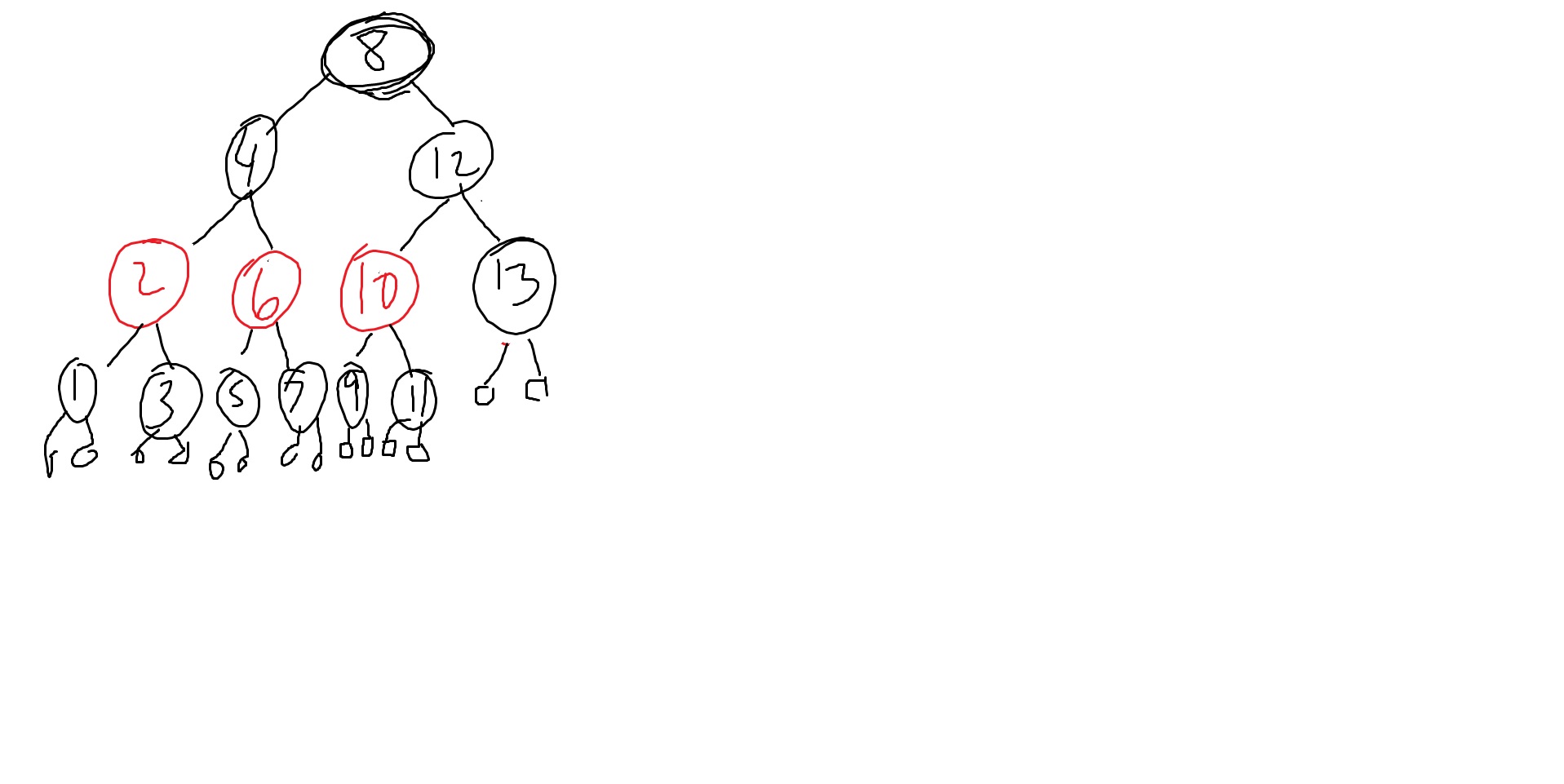
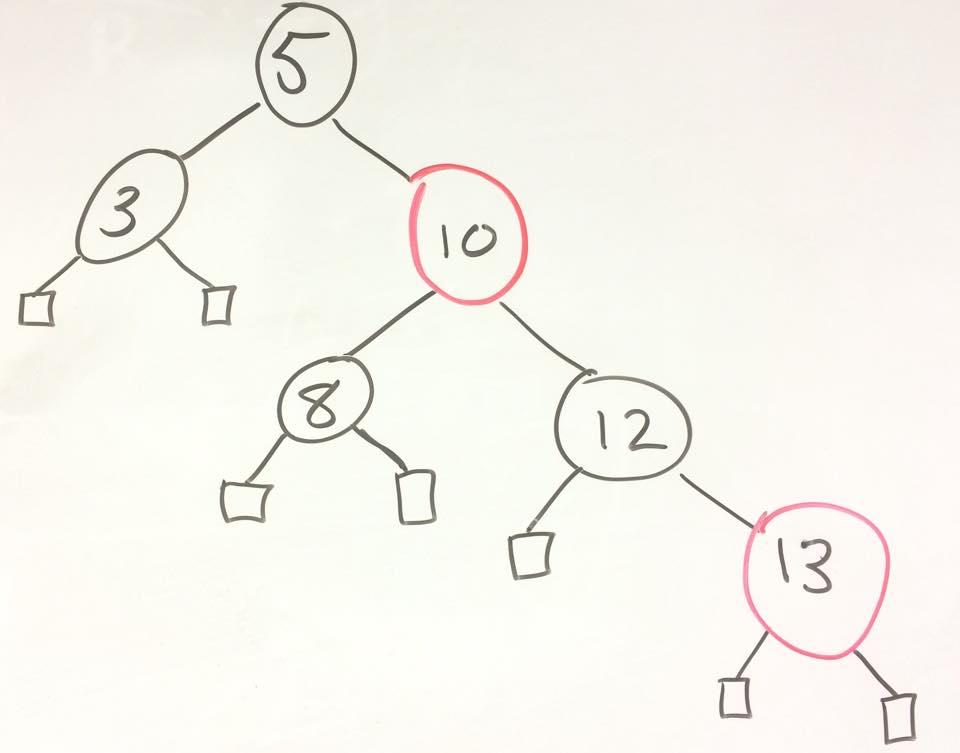
HW5

1. See below
   1. The algorithm would search list B for the b values that, when added with a, produce a zero result. (This search would be of complexity O(log n).) This would be done as you look at each element of list A. (This would be of complexity O(n).)  
        
      for (int i = 0; i < size(A); i++) { // -> O(n)  
       searchTree(B); // in this function, we would determine if the list had any  
       // values that satisfied a + b = 0 -> O(log n)  
      }
   2. We can generalize this as containing (n-1) for loops and then searching the nth tree for values that satisfy the given condition (a + b + . . . + n = 0). This would increase the complexity to O(n(n-1)log n).  
        
      For each of these lists, we would have (n-1) for loops (nested), where n represents the lists (A, B, C, D). That is, for k = 3 lists, we would have a for loop that iterates from i = 0 to size(A), then a second loop that would iterate from j = 0 to size(B). Then we would do a search through the 3rd tree for the elements that satisfied the condition (a + b + c = 0). This would provide a complexity of O(n2log n). For k = 4 lists, we would have 3 for loops (nested). Within the innermost for loop, we would then search the 4th tree for the elements that satisfied the condition (a + b + c + d = 0). This would provide a complexity of O(n3log n).  
        
      **k = 3**  
      for (int i = 0; i < size(A); i++) { // -> O(n)  
       for (int j = 0; j < size(B); j++) { // -> O(n)  
       searchTree(C); // see the comment from part a) but keep in mind  
       // that we would be searching for values that made  
       // a + b + c = 0 true -> O(log n)  
       }  
      }  
        
      **k = 4**  
      for (int i = 0; i < size(A); i++) { // -> O(n)  
       for (int j = 0; j < size(B); j++) { // -> O(n)  
       for (int k = 0; k < size(C); k++) { // -> O(n)  
       searchTree(D); // see the comment from part a) but keep in  
       // mind that we would be searching for  
       // values that made a + b + c + d = 0 true  
       // -> O(log n)  
       }  
       }  
      }
2. See below
   1. See the image below
   2. See the image below
   3. See images below



1. See the image below

This red-black tree is not an AVL tree because the external (leaf) nodes are not all on the same level. However, It is a valid red-black tree because it satisfies all properties of a red-black tree. The root is black. Every node is either red or black. The black height from the root to any of the external NIL nodes are equal (3). None of the red nodes have a red child node, and all external nodes are black.

1. Since we are looking at a general binary search tree (BST) rather than a special case such as a red-black tree, we can assume external nodes are nodes that contain data and have zero children. Given any BST, we can convert it into a chain of nodes all along one side. This tree would have a length of n, and it could be created by no more than O(n) rotations in the same direction on the original BST. (See image below).

If a node in the tree has a left subtree, we can rotate it to the right until all nodes are along the right side of the tree. This can also be done in the reverse direction, i.e., given a node with a right subtree, we can rotate it to the left until all nodes of the BST are in a left-going chain of length n, using at most O(n) rotations. Therefore, we can transform any BST of n nodes into any other BST of the same size by using O(n) rotations.

