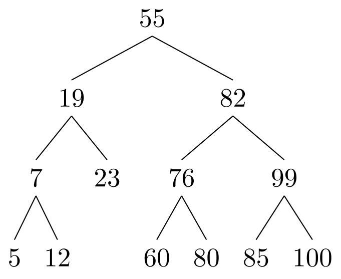
CS 1332 Exam 2 - not all topics are represented here. See the list of major topics uploaded as a T-Square Announcement. This is only a sample! (A bit more has been added to the end.)

Taken from Spring Semester 2014

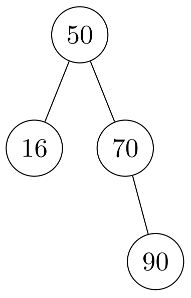
1. You are given the following tree.



   (a) (5 points) Perform a post-order traversal printing the data of the tree depicted above. Write the numbers here:

   (b) (5 points) Using the BST depicted above, add the data values 70 (add this first) and then 77 (add this second) to the BST (without rebalancing). Do not redraw the tree that is given for this answer, simply add the value 70 and then 77 where they would be in the existing drawing above. (Note these new numbers have nothing to do with the traversal.)

2. (5 points) You are given the following AVL tree. First add 99 (literally) to the tree that is given to you (showing it unbalanced). Secondly, using the space below and on the right, redraw the balanced AVL tree that would result from this add of 99 and subsequent rebalancing according to AVL rules.



3. Matching. Place the number of the answer that BEST matches. Choices can be used once, more than once, or not at all. In all cases, n refers to the number of data items.

1. O(1)	4. O(log n)
2. O(2^n)	5. O(n^2)
3. O(n)	6. O(n log n)

(a) (3 points) ______ Big O of worse case for searching a BST that is balanced.

(b) (3 points) ______ Big O of worst case for searching a BST that is unbalanced.

(c) (3 points) ______ Big O of best case for searching a BST regardless of whether it is balanced or not.

(d) (3 points) ______ Big O of level-order traversing of a BST.

(e) (3 points) ______ Big O of preorder traversing of a BST.

(f) (3 points) ______ Big O of worst case for searching an AVL tree.

(g) (3 points) ______ Big O of worst case add for an AVL tree.

(h) (3 points) ______ Big O of level-order traversing of an AVL.

(i) (3 points) ______ Big O of hash table lookup when using an array with external chaining for collision resolution when the hashtable has had 99% collisions.

(j) (3 points) ______ Big O of hash table lookup when using an appropriately sized baking array with external chaining for collision resolution when the hashing function works well.

(k) (3 points) ______ Big O of worst case hash table lookup when using an array with linear probing as the collision resolution strategy.

(l) (3 points) ______ Big O to rebalance an AVL tree that becomes unbalanced during an add. Assume the rebalancing is part of the add method.

4. (8 points) Hash Tables. Your hashing function sums the individual digits of a number.

The hash table size is 5.

Using each data item in the order given below, add the data item to the hash table.

You must use the hash function as described and take into account the size of the hash table.

Your hash table is to use external chaining. (Feel free to add to the end of each linked list to make the drawing/writing easier.) Be sure to show all data items.

Hash Function: Sum of the individual digits of the number.
Note: The table size is 5. Do not resize the array.

```
Data: 55, 2110, 0004, 133, 322, 91, 78, 51
```

hashtable[0] →

hashtable[1] →

hashtable[2] →

hashtable[3] →

hashtable[4] →

5. (30 points) BST (Binary Search Tree) instance method **add(int)**. Using the classes given below, complete the instance method named **add** having int data as a parameter. Your code is to add the data to the BST maintaining the BST properties. **Feel free to write a helper method if you wish.**

   You are to assume that root will be null if the original BST is empty. Also, all nodes existing in the BST have an int data value. Left and/or right will be null if not referencing a node in the tree.

   Do not ask what to do if the parameter is null. The parameter is literally an int which is a primitive type.

   Notice that Node is a private inner class and that BST's instance method add follows it below.

```
public class BST {
    private Node root;

    private class Node {  // Note this is a private inner class.
        private int data;
        private Node left, right;
        public Node(int data) {
             this.data = data;
            left = null;
            right = null;
        }
    } // end of private inner class Node

    public void add(int data) {
```

NEW STUFF STARTS HERE

6. Be sure you are familiar with the Big O of all typical operations for all the data structures we have studied. Adding, removing, searching would be just the start for each and every data structure we study. If other operations are asked about, you should be familiar enough with Big O to surmise an answer even if it is a novel operation we have not literally talked about. This should not be a matter of memorization!

7. Be sure you can perform the typical operations by hand for all the data structures we have studied. Adding (with or without balancing as appropriate for the data structure of interest), removing, searching of any data structure we have studied.

8. Less likely, but still possible, if an operation is described involving a data structure, you might be expected to show how it would work - even if it is not a classic operation that we have studied.

9. Implement a hash table with an array of size 10 as the backbone and external chaining as the resolution strategy. Include a constructor, an add method, and a search method. You can assume the data will be ints. Make up a hashing function of your choice.

10. Implement a hash table with an array of size 10 as the backbone and open addressing with linear probing as the resolution strategy. Include a constructor, an add method, and a search method. You can assume the data will be ints. Make up a hashing function of your choice.

11. Assume you have a MaxHeap class with a size instance variable and an array instance variable. *size* and *array* are the names of these two instance fields. ~~Note the length of the array might be less than the size of the heap.~~ **Note the length of the array might be greater than the size of the heap.** Assume the class already has a removeMax method that returns the maximum of the heap and also adjusts the max heap to reflect the "removal" of that max value from the logical heap portion of the array. The class already has an add instance method implemented to add a new value to the heap (although you should not need that for the method you will be writing.)

    Assume slot 0 of the array remains unused to help simplify the indexing within the max heap.

    Write an instance method heapSort. It is to work in place and result in the data in the instance field *array* being transformed into a increasing-order array. (You can assume there are never any duplicates.) The steps you perform must adhere to the classic technique used in heapSort and not by calling nor writing some other sort.

    In the end, the *size* of the heap will have become 0.

    Do not make nor use an extra copy of the array.

    (For the purposes of this question, you can just ignore any stray values that might have been in the array that were past the portion that represented the heap based on *size*. Those extra non-heap values can just sit there and not be sorted.)