Test 2 : Compsci 201

Owen Astrachan

April 5, 2019

Name: _____

NetID/Login: _____

Community standard acknowledgment (signature) _____

	value	grade
NetID	1 pt.	
Problem 1	24 pts.	
Problem 2	7 pts.	
Problem 3	17 pts.	
Problem 4	23 pts.	
Problem 5	12 pts.	
TOTAL:	84 pts.	

This test has 16 pages, be sure your test has them all. Write your NetID *clearly* on each page of this test (worth 1 point).

In writing code you do not need to worry about specifying the proper `import statements`. Don't worry about getting method names exactly right. Assume that all libraries and packages we've discussed are imported in any code you write. You can write any helper methods you would like in solving the problems. You should show your work on any analysis questions where it is asked for.

You may consult your six (6) note sheets and no other resources. You may not use any computers, calculators, cell phones, or other human beings. Any note sheets must be turned in with your test.

Common Recurrences and their solutions.

label	recurrence	solution
A	`T(n) = T(n/2) + O(1)`	$O(\log n)$
B	`T(n) = T(n/2) + O(n)`	$O(n)$
C	`T(n) = 2T(n/2) + O(1)`	$O(n)$
D	`T(n) = 2T(n/2) + O(n)`	$O(n \log n)$
E	`T(n) = T(n-1) + O(1)`	$O(n)$
F	`T(n) = T(n-1) + O(n)`	$O(n^2)$
G	`T(n) = 2T(n-1) + O(1)`	$O(2^n)$

`TreeNode` and `ListNode` classes as used on this test. In some problems the type of the `info` field may change from `int` to `String` and *vice versa*

```
public class TreeNode {
    String info;
    TreeNode left;
    TreeNode right;

    TreeNode(String x){
        info = x;
    }
    TreeNode(String x,TreeNode lNode,
                      TreeNode rNode){
        info = x;
        left = lNode;
        right = rNode;
    }
}
```

```
public class ListNode {
    int info;
    ListNode next;
    ListNode(int val) {
        info = val;
    }
    ListNode(int val,
             ListNode link){
        info = val;
        next = link;
    }
}
```

PROBLEM 1 : (*Wink Your Queen (24 points)*)

Consider the problem of removing duplicate elements from an array of strings creating a *unique* array. For example, given this array

```
["a", "b", "a", "b", "a", "b", "c", "d", "a"]
```

A method creating a unique array could return:

```
["a", "b", "d", "c"]
```

The order of the values in the unique array doesn't matter, but it should contain one of each unique value in the original array and no values not in the original array.

You're asked to reason about three methods that return a unique array. All work correctly. In answering questions using Big-Oh you can assume that all strings in the array parameter `a` are unique (this is likely the worst case).

Part A (8 points)

Method `rd2` returns a unique array using classes whose APIs are specified in the `java.util` package.

```
22    public String[] rd2(String[] a) {
23
24        HashSet<String> set =
25                new HashSet<>(Arrays.asList(a));
26
27        return set.toArray(new String[0]);
28    }
```

A.1 (2 points)

Do lines 24 and 25 together do the same thing as the code below? Answer *same* or *different* in the space to the right of the code.

```
HashSet<String> set = new HashSet<>();
for(String s : a) set.add(s);
```

A.2 (3 points)

What is the runtime complexity (using Big-Oh) of this code when parameter `a` has N elements? Explain your answer in general terms, be sure to account for all lines of code.

A.3 (3 points)

If `HashSet` is replaced by `TreeSet` in the code above the method is still correct. What is the runtime complexity (using Big-Oh) when `TreeSet` is used? Explain your answer.

Part B (7 points)

Method `rd3` returns a unique array using an `ArrayList`.

```
30    public String[] rd3(String[] a) {
31        ArrayList<String> list = new ArrayList<>();
32        for(String s : a) {
33            if (! list.contains(s)) {
34                list.add(s);
35            }
36        }
37        return list.toArray(new String[0]);
38    }
```

B.1 3 points

What is the runtime complexity (using Big-Oh) of this code when parameter `a` has N elements. Explain your answer in general terms, be sure to account for all lines of code.

B.2 2 points

Explain the purpose of line 37 in creating the unique array to return.

B.3 2 points

Are the elements in the array returned on line 37 in the same order relative to each other as they are in array parameter `a`? Explain your answer.

Part C (9 points)

Method `rd1` below returns a unique array by first sorting the array, then removing duplicate elelements (which are adjacent since the array is sorted).

```
 5    public String[] rd1(String[] a) {
 6        Arrays.sort(a);
 7        int lastUniqueIndex = 0;
 8        for(int k=1; k < a.length; k++) {
 9            if (! a[k].equals(a[lastUniqueIndex])) {
10                lastUniqueIndex++;
11                a[lastUniqueIndex] = a[k];
12            }
13        }
14        int size = lastUniqueIndex + 1;
15        String[] b = new String[size];
16        for(int k=0; k < size; k++) {
17            b[k] = a[k];
18        }
19        return b;
20    }
```

C.1 (3 points)

What is the runtime complexity (using Big-Oh) of this code when parameter `a` has N elements. Explain your answer in general terms, be sure to account for all lines of code.

C.2 (3 points)

Explain in general terms the code in lines 14-19 (not by explaining every line) in creating the unique array to return.

```
 5    public String[] rd1(String[] a) {
 6        Arrays.sort(a);
 7        int lastUniqueIndex = 0;
 8        for(int k=1; k < a.length; k++) {
 9            if (! a[k].equals(a[lastUniqueIndex])) {
10                lastUniqueIndex++;
11                a[lastUniqueIndex] = a[k];
12            }
13        }
14        int size = lastUniqueIndex + 1;
15        String[] b = new String[size];
16        for(int k=0; k < size; k++) {
17            b[k] = a[k];
18        }
19        return b;
20    }
```

C.3 (3 points)

The loop in lines 8-12 has one of the three statements below as an invariant. Choose which is an invariant: true each time the loop test `k < a.length` is evaluated. As is typical `[` and `]` indicate a closed/inclusive end-point and `(` and `)` indicate an open/non-inclusive end-point.

1. Elements in `[0..lastUnique]` are unique, there are no duplicates in this range.
2. Elements in `[0..lastUnique)` are unique, there are no duplicates in this range.
3. Elements in `(0..lastUnique]` are unique, there are no duplicates in this range.

You must provide three things for full credit:

- Which invariant above is true.
- Justify/explain briefly why the invariant is true the first time the loop test is evaluated.
- Justify/explain briefly why the invariant is true after lines 10 and 11 execute.

PROBLEM 2 : (*Summary Motion (7 points)*)

The method `sum` below correctly returns the sum of all the nodes in its linked-list parameter.

```
42      public int sum(ListNode list) {
43          if (list == null) return 0;
44          return list.info + sum(list.next);
45      }
```

Part A (3 points)

What is the recurrence relation for this method when `list` has n nodes? Use $T(n)$ as the time for `sum` to execute on an n-node list. What is the solution to this recurrence? You don't have to explain/justify, just write the recurrence and its solution.

$T(n) =$

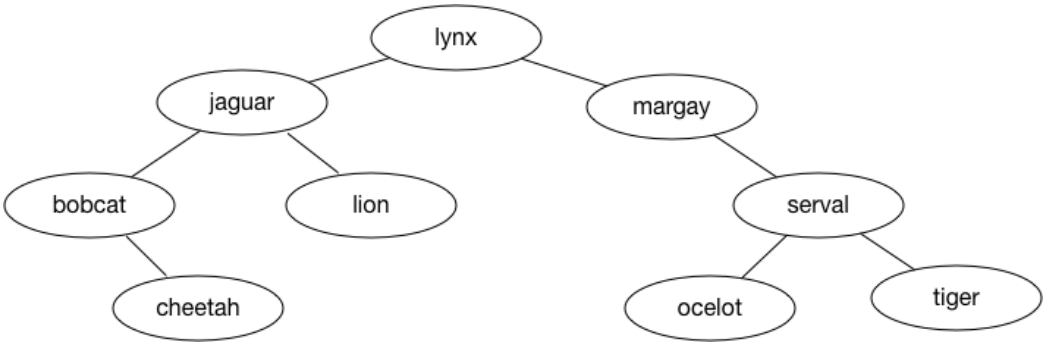
Part B (4 points)

Write an iterative solution, one in which no recursion is used. For every list the values of `sum(list)` and `summer(list)` should be the same.

```
public int summer(ListNode list) {



}
```

PROBLEM 3 : (*Catsup (17 points)*)

The tree below is a binary *search* tree.

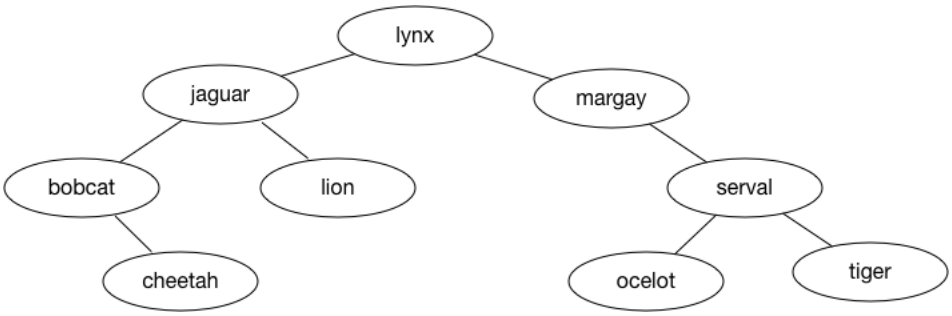


Part A (8 points)

Label/add the four strings *caracal*, *leopard*, *puma*, *panther*, and *snow leopard*, *in that order*, to the tree shown above. Be sure it's clear that each string added is a left- or right-child.

Part B (9 points)

In answering these questions use the tree *before values are added as in Part A*. This is reproduced here.



Consider method `toString` below.

```
19    public String toString(TreeNode root) {
20        if (root == null) return "";
21
22        String left = toString(root.left);
23        String right = toString(root.right);
24        String ret = left + " " + root.info + " " + right;
25        return ret.strip();
26    }
```

The value returned by `toString(root)`, where `root` points at the node with *lynx* above, is shown below.

```
bobcat cheetah jaguar lion lynx margay ocelot serval tiger
```

B.1 (3 points)

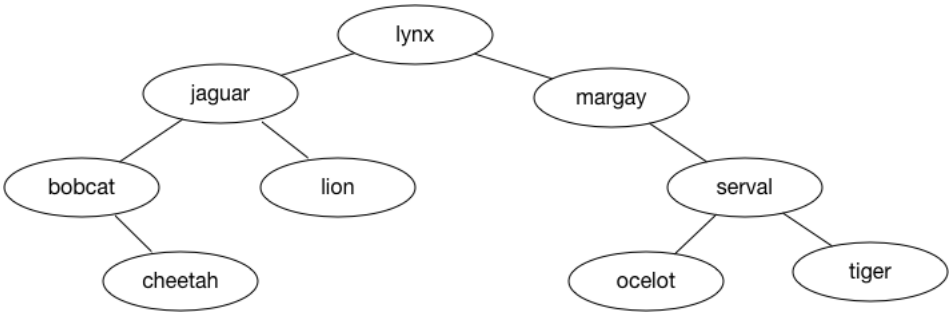
If line 24 in the method `toString` is changed to

```
String ret = root.info + " " + left + " " + right;
```

what is returned by the call `toString(root)`? The amount of whitespace between strings doesn't matter, just the order of the words.

B.2 (3 points)

In answering these questions use the tree *before values are added as in Part A*. This is reproduced here.



Consider method `toString` below.

```
19    public String toString(TreeNode root) {
20        if (root == null) return "";
21
22        String left = toString(root.left);
23        String right = toString(root.right);
24        String ret = left + " " + root.info + " " + right;
25        return ret.strip();
26    }
```

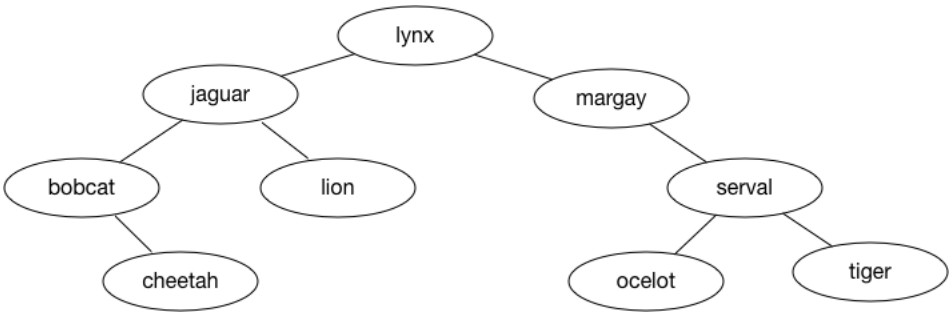
If line 24 in the method `toString` is changed to

```
String ret = right + " " + root.info + " " + left;
```

what is returned by the call `toString(root)`? The amount of whitespace between strings doesn't matter, just the order of the words.

B.3 (3 points)

In answering these questions use the tree *before values are added as in Part A*. This is reproduced here.



Consider method `toString` below.

```
19    public String toString(TreeNode root) {
20        if (root == null) return "";
21
22        String left = toString(root.left);
23        String right = toString(root.right);
24        String ret = left + " " + root.info + " " + right;
25        return ret.strip();
26    }
```

Indicate how to change line 24 so that the following is printed.

```
tiger ocelot serval  margay lion cheetah  bobcat jaguar lynx
```

You should write the code, you can only use the values of `left`, `right`, and `root.info` in the code you write. You should have some whitespace between each word, the amount of whitespace doesn't matter.

Write line 24 below:

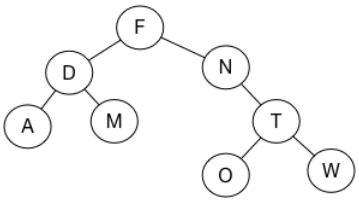
```
String ret =
```

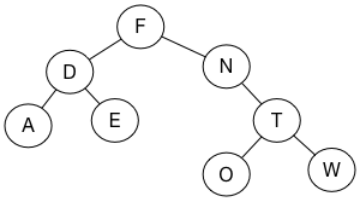
PROBLEM 4 : (*Needle in a Haystack? (23 points)*)

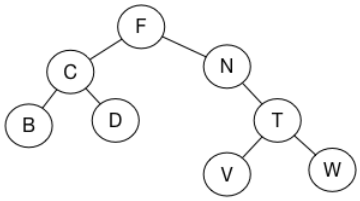
The method `searchTree` below correctly returns `true` if its parameter `root` is a binary search tree, and `false` if it is not a binary search tree.

```
29    public int max(TreeNode root) {
30        if (root == null) return Integer.MIN_VALUE;
31        return Math.max(root.info,
32                        Math.max(max(root.left), max(root.right)));
33    }
34
35    public int min(TreeNode root) {
36        if (root == null) return Integer.MAX_VALUE;
37        return Math.min(root.info,
38                        Math.min(min(root.left), min(root.right)));
39    }
40
41    public boolean searchTree(TreeNode root) {
42        if (root == null) return true;
43        if (! searchTree(root.left)) return false;
44        if (! searchTree(root.right)) return false;
45
46        if (root.info <= max(root.left)) return false;
47        if (root.info > min(root.right)) return false;
48
49        return true;
50    }
```

Consider each tree rooted at **F** below.







- The tree on the left is *not a search tree* because although the left-subtree of the root is a search tree, and the right-subtree of the root is a search tree, one of the nodes, **M**, in the left-subtree is greater than the root.
- The tree in the middle *is a search tree* because for each node, the subtree rooted at that node is a search tree and each node is greater than all values in its right subtree and less than or equal to all values in its left subtree.
- The tree on the right is *not a search tree* because the subtree rooted at **T** is not a search tree since **V** is not less than or equal to **T**.

Part A (7 points)

Methods `min` and `max` shown above each have a recurrence of $M(n) = 2M(n/2) + O(1)$.

Let $T(n)$ be the time for `searchTree` to run when the tree parameter has n nodes. *Label each line of the method* `searchTree` with an expression using T, or with a big-Oh expression. Assume trees are roughly balanced. Using these labels complete the recurrence:

$T(n) =$

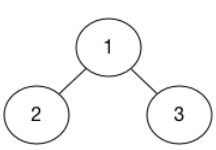
What is the solution to this recurrence?

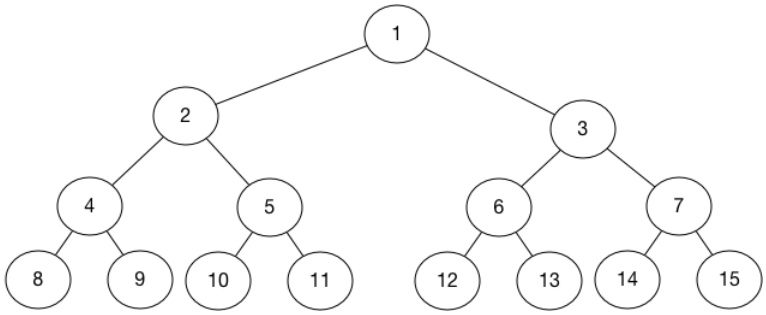
Part B (8 points)

In this problem `TreeNode.info` has type `int`, so stores an integer value.

A *full tree* has 2^{n-1} nodes at level n where the root is at level one, its children at level two, and in general the children of a node at level k are at level $k+1$. The nodes at level n are numbered starting with 2^{n-1} and incrementing by one, from left-to-right, until the last node on level n which is $2^n - 1$

In the trees below, the tree on the left is a full-tree with two levels and the tree on the right is a full tree with four levels. We call these 2-level and 4-level full trees, respectively.





B.1

What is the total number of nodes in a 4-level tree, i.e., the tree on the right above?

B.2

What is the total number of nodes in an n-level full tree?

B.3

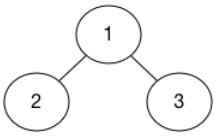
What is the number of *non-leaf* nodes in a 3-level full tree?

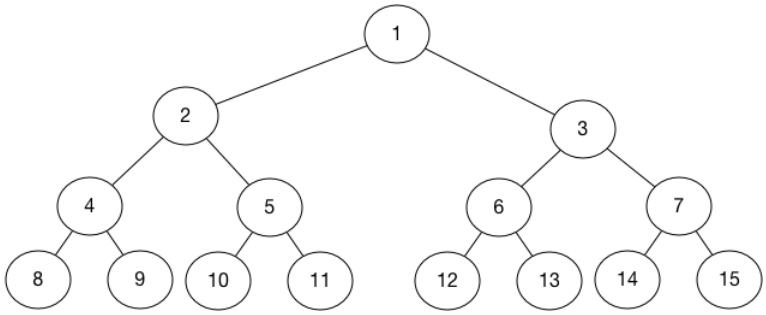
B.3

What is the number of *non-leaf* nodes in a n-level full tree?

Part C (8 points)

The method `createFull` below creates and returns a full tree with the specified number of levels so that the call `createFull(2)` returns the tree on the left and the call `createFull(4)` returns the tree on the right.





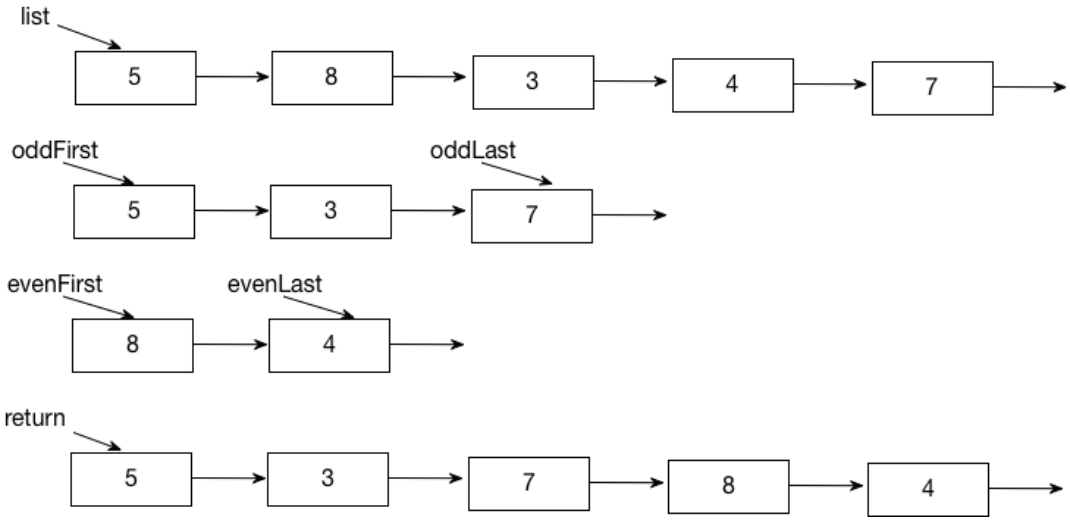
The method uses a private helper method that you should complete using at most 10 lines of code.

```
public TreeNode createFull(int levels) {
    return fullHelper(1,levels);
}

/**
 * Return a full-tree with the specified number of levels
 * in which the root contains nodeValue
 * @param nodeValue is the value of the root of the tree returned
 * @param levels is the number of levels in the full tree returned
 * @return a full tree
 */
private TreeNode fullHelper(int nodeValue, int levels) {
    if (levels == 1) {
        return new TreeNode(nodeValue);
    }









}
```

PROBLEM 5 : (*The Great Divide (12 points)*)

Write method `divide` which alters a linked list and returns the altered list so that all nodes with an odd index appear in the same order as in the original list followed by all nodes with an even index in the same order. Odd and even nodes are determined *by the index of the node in the original list* with the first node as node number 1, the next node as node number 2 and so on.

For example, the original list is shown at the top of the diagram and the returned list at the bottom.



Your code should not create any new nodes, but instead alter next fields. Your code should run in $O(n)$ time where n is the number of nodes in the original list.

You can write the code in any way.

You may find the following ideas useful in writing a solution.

- Check if `list` is null or has one node, in which case it can be returned without changing any pointers. This ensures that the code you write deals with lists of two or more nodes and thus contains both an odd node and an even node.
- Create a variable `boolean odd = true` before the loop you write. In the loop you write use the statement `odd = !odd` to change the value from `true` to `false` and *vice versa*.

One idea that will work **and that you're strongly encouraged to use** is to maintain pointers to the first and last nodes of two lists: an odd-list and an even-list as shown in the diagram. As the original list is traversed, nodes are *alternately added to the end of the odd-list or even-list.*

If you write the code to maintain four such pointers, the last statements you might write are:

```
oddLast.next = evenFirst;
evenLast.next = null;
return oddFirst;
```

You'll need to initialize these four variables before the loop and access some of them in the body of the loop that you write.

Complete the code on the next page.

```
public ListNode divide(ListNode list) {

}
```