Midterm 2: Compsci 201

Tabitha Peck

March 27, 2013

Print your name and NetID legibly in ALL CAPITAL letters. Make sure that we can clearly determine L vs. 1 and S vs. 5.

Name: ____

NetID/Login: _______________

Honor code acknowledgment (signature) ____

This test has 15 pages (with a help page and an APT Problem Statement at the end), be sure your test has them all. Do NOT spend too much time on any one question — remember that this class lasts 75 minutes.

Write clearly and legibly. If we cannot read it, we cannot grade it.

In writing code you do not need to worry about specifying the proper `import statements`. Don't worry about getting function or method names exactly right. Assume that all libraries and packages are imported in any code you write.

	value	grade
Problem 1	28 pts.	
Problem 2	26 pts.	
Problem 3	12 pts.	
Problem 4	12 pts.	
TOTAL:	78 pts.	

1 Short Answer: 28 points

1.1 Trees: 5 points

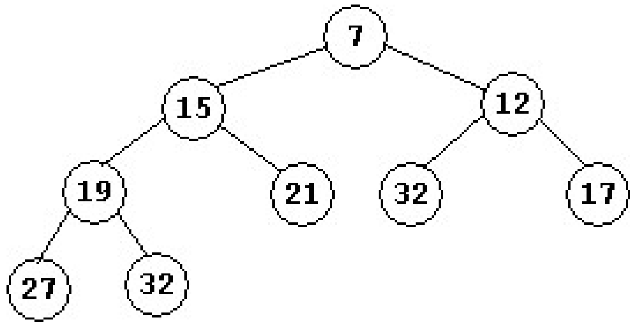
Show the resulting tree after EACH insertion of the following: **50, 78, 10, 54, 30** (in order) into an initially empty binary search tree. You should draw 5 trees, the resulting tree after EACH insertion.

1.2 Heaps: 5 points

Complete the array that stores the heap below such that the children of node k are located at $2k$ and $2k+1$. Note that no value is saved in index zero in the array.

	7	15	12	19	21	32	17	27	32

Show the resulting heap after EACH insertion of the values **17, 16, 11** into the min-heap below. You should draw 3 heaps, the resulting heap after EACH insertion.



1.3 Trees vs. Heaps I: 6 points

Complete the following table with the average running time (big-oh) for each operation on a binary search tree (BST) vs. a heap.

Operation	**BST**	**Min-Heap**
find the minimum value	$O(lgN)$	$O(1)$
find value x	$O(lg(N))$	$O(N)$
add value x	$O(lgN)$	$O(lgN)$

1.4 Trees vs. Heaps II: 6 points

Although heaps are often stored in arrays, not all binary trees are. Why wouldn't you want to store all binary trees in arrays such that the children of node k are located at $2k$ and $2k+1$?

If your tree is sparse or unbalanced then your array will have empty indices and this will waste memory.

Based on your reasoning above draw a binary tree that you would not want to store in an array:

1.5 Tree Traversals: 2 points

Give the post-order traversal of the binary heap (before you added the values *17, 16, and 11*) from **Question 1.2**.

27, 32, 19, 21, 15, 32, 17, 12, 7

1.6 Linked Lists vs. Trees: 4 points

Suppose you are adding the numbers 1, 2, 3, 4, ..., n into a non-balancing binary search tree. Discuss two approaches to adding the nodes into the BST, one that will produce a roughly balanced tree, and one that will produce the equivalent to a linked list. Make sure to discuss the running times for your two approaches.

Method 1: add the numbers to your tree in order from 1-n. Your tree will be a stick and equivalent to a linked-list. Each add will cost $O(N)$ and to make the whole tree will be $O(N^2)$.

Method 2: add the numbers in a random order. For large values of n your tree will be roughly balanced. Each add will cost $O(lgN)$, thus for the whole tree, $O(NlgN)$.

2 Trees and Recursion: 26 points

2.1 Recursion and Trees: 12 points

You want to add the method *isBST* to the class *BinaryTree* below. Assume that *BinaryTree* has a method for adding nodes to the tree. Complete the recursive method *isBST* that determines if the tree starting at *myRoot* is a binary search tree. The method *isBST* returns true if the binary tree is a binary *search tree* and false otherwise. Don't forget your base case. Note: assume that your binary tree has all unique values. In writing *isBST* you can (and should) call the functions *maxValue* and *minValue* shown below, these return the maximal and minimal values in a tree, respectively.

2.1.1 Code: 9 points

```
public class BinaryTree {
    Node myRoot = null;
    public class Node {
        public int myValue;
        public Node myLeft, myRight;
        public Node (int val) {
            myValue = val;
        }
    }
    public int maxValue(Node root) {
      if (root == null) return Integer.MIN_VALUE;  // negative infinity
      return Math.max(root.myValue,
                      Math.max(maxValue(root.myLeft),maxValue(root.myRight)));
    }

    public int minValue(Node root) {
      if (root == null) return Integer.MAX_VALUE;  // infinity
      return Math.min(root.myValue,
                      Math.max(minValue(root.myLeft),minValue(root.myRight)));
    }
    public boolean isBST(){ return isBST(myRoot); }

    public boolean isBST(Node cur){


        if(cur == null)
            return true;

        if(cur.myLeft != null  && maxValue(cur.myLeft) > cur.myValue)
            return false;

        if(cur.myRight != null && minValue(cur.myRight) < cur.myValue)
            return false;

        if(!isBST(cur.myLeft) || !isBST(cur.myRight))
            return false;

        return true;
    }
```

2.2 Trees and Recursive Backtracking: 14 points

Complete the method *getPathSum* which is added to the class *BinaryTree*, from above, by completing the code below as designated by the comments. The method *getPathSum* takes an integer, *target*, as a parameter and the current root, *cur*, in a tree. The method returns a stack of integer values representing a path from the root to a node in the tree that sum to *target*. The stack returned is empty if there is no root-to-node path that sums to target.

For example, *getPathSum(19)* on the original binary tree (the heap) in **Section 1.2** would build and return a stack containing *7, 12*. *getPathSum(43)* would return a stack containing 7,15,21, but *getPathSum(15)* would return an empty stack since no path starting at the root sums to 15.

Assume all values in the tree are positive.

Make sure to add the appropriate code within each of the designated comments.

```
    public Stack<Integer> getPathSum(int target){
        Stack<Integer> path = new Stack<Integer>();
        getPathSum(path,myRoot, target);
        return path;
    }



    private boolean getPathSum(Stack<Integer> path,Node cur, int target){
```

//**What are your base cases? 4 points**

```
        if(target == 0)
            return true;
        if(cur == null)
            return false;
        if(target < 0)
            return false;
```

//**Add the current value to *myPath*: 2 points**

```
    path.push(cur.myValue);
```

//Check if you can build a path left or right and return the appropriate booleans. 6 points

```
if(getPathSum(path, cur.myLeft, target-cur.myValue))
    return true;
if(getPathSum(path, cur.myRight, target-cur.myValue))
    return true;
```

//If you could not go left or right, you will need to backtrack. Add that code here. 2 points

```
path.pop();
return false;
```

3 Homework: 12 points

3.1 WordLadder: 6 points

At the end of the exam you will find the **APT Problem Statement** for **Word Ladder I**. The code below is an almost working solution, however your friend forgot one small thing. The code currently passes 13 out of 15 cases, and fails on the two cases, one of which is shown below. Answer the question on the following page.

```
expected
 "none"
got
 "ladder"
: ["hot", "dog", "log" ] "hog" "hig"
```

```
public class WordLadder {
    public String ladderExists(String[] words, String from, String to) {
        ArrayList<String> list = new ArrayList<String>(Arrays.asList(words));
        if(isLadder(list, from, to))
            return "ladder";
        return "none";
    }
    public boolean isLadder(ArrayList<String> words, String from, String to){
        if(isStep(from, to))
            return true;
        if(words.isEmpty())
            return false;
        for(String s: words){
            if(isStep(from, s)){
                ArrayList<String> copy = new ArrayList(words);
                copy.remove(s);
                if(isLadder(copy, s, to))
                    return true;
                copy.add(s);
            }
        }
        return false;
    }
    private boolean isStep(String w1, String w2){
        char[] c2= w2.toCharArray();
        char[] c1 = w1.toCharArray();
        int inCommon = 0;
        for (int i=0; i<c1.length; i++){
            if(c2[i] == c1[i]){
                inCommon++;
            }
        }
        if(inCommon == (c2.length-1))
            return true;
        else
            return false;
    }
  }
```

Word Ladder: Explain your friend's error and clearly explain how you would fix the error.

My friend forgot to make sure that the ladder had at least one interior rung. You can fix this by adding a parameter to the method call that keeps track of the number of steps taken to find the ladder. You will need to add a base case that makes sure that there is at least one interior rung.

3.2 DNA: 6 points

Complete the following method from the DNA assignment that reverses a LinkStrand. You can use the methods *public IDnaStrand append(String dna)* and *public IDnaStrand appendBeginning(String dna)* that append nodes at the back and front of the LinkeStrand respectively.

```
public IDnaStrand reverse() {

    if(myHead == null)
        return this;

    LinkStrand reverseStrand = new LinkStrand();
    StringBuilder data = new StringBuilder(myHead.myData);
    data = data.reverse();
    reverseStrand.append(data.toString());


    Node forwardsList = myHead.myNext;
    while(forwardsList != null) {
        StringBuilder data1 = new StringBuilder(forwardsList.myData);
        data1 = data1.reverse();
        reverseStrand.appendBeginning(data1.toString());
        forwardsList = forwardsList.myNext;
    }
    return reverseStrand;
```

4 Sorting Algorithms: 12 points

The code for three sorting algorithms that were discussed in class can be found on the next page. Answer the following questions based on the code for the three sorting algorithms.

4.1 6 points

What is the running time for each algorithm on a sorted list? **Explain your answer**.

a. Algorithm 1 $O(N)$ The inner for loop will not run because temp ¡ a[j].

b. Algorithm 2 $O(NlgN)$ Split list in half until size is 1 (lgN) then merge all N elements together

c. Algorithm 3 $O(NlgN)$ remove N elements from a priority queue. Remove from a heap costs $O(lgN)$

4.2 6 points

What is the running time for each algorithm on a list in random order? **Explain your answer**.

a. Algorithm 1 $O(N^2)$ The inner for loop could run 1 + 2 + 3 + ... + N times. The outer for-loop runs N times.

b. Algorithm 2 $O(NlgN)$ Split list in half until size is 1 (lgN) then merge all N elements together

c. Algorithm 3 $O(NlgN)$ remove N elements from a priority queue. Remove from a heap costs $O(lgN)$

4.3 Extra Credit

What are the names of each of the three sorting algorithms?

a. Algorithm 1
Insertion sort

b. Algorithm 2
Merge sort

c. Algorithm 3
Heap sort

Algorithm 1

```
public void sort1(int[] a){
    for(int i=1; i < a.length; i++){
        int temp = a[i];
        int j;
        for(j= i-1; (j>= 0 && temp < a[j]); j--)
            a[j+1] = a[j];
        a[j+1] = temp;
    }
}
```

Algorithm 2

```
public void sort2(int[] a){
    if(a.length > 1){
        int half = a.length/2;
        int[] a1 = Arrays.copyOfRange(a, 0, half);
        int[] a2 = Arrays.copyOfRange(a, half, a.length);
        sort2(a1);
        sort2(a2);
        sort2Helper(a, a1, a2);
    }
}
private void sort2Helper(int[] array, int[] a1, int[] a2){
    int len1 = a1.length;  int len2 = a2.length;
    int it1 = 0;  int it2 = 0;
    for(int i=0; i < array.length; i++){
        if(it2==len2 || (it1 < len1 && a1[it1] < a2[it2])){
            array[i] = a1[it1];
            it1++;
        }
        else{
            array[i] = a2[it2];
            it2++;
        }
    }
}
```

Algorithm 3

```
public void sort3(int[] a){
    PriorityQueue<Integer> q = new PriorityQueue<Integer>();
    for(int i: a)
        q.add(i);
    int i = 0;
    while(!q.isEmpty()){
        a[i] = q.remove();
        i++;
    }
}
```

```
String
```

- .length() Get the length of the String. $O(1)$.
- .charAt(i) Get the char at index i. $O(1)$.
- .split(" ") Split a string by spaces and store it in a string[].
- .substring(i, j) Get the substring between indices i and j. Index i is *inclusive*, and index j is *exclusive*. $O(1)$. For example:

```
String x = "abcdefg";
String y = x.substring(2, 4);
// y now has the value "cd"
```

```
ArrayList<T>  // Where T is a type, like String or Integer
```

- .add(i, X) Add element X to the list at index i. If no i is provided, add an element to the end of the list. Adding to the end runs in $O(1)$.
- .get(i) Get the element at position i. Runs in $O(1)$.
- .set(i, X) Set the element at position i to the value X. $O(1)$.
- .size() Get the number of elements. $O(1)$.

```
HashSet<T>  // Where T is a type, like String or Integer
```

- .size() Compute the size. $O(1)$.
- .add(X) Add the value X to the set. If it's already in the set, do nothing. $O(1)$.
- .contains(X) Return a boolean indicating if X is in the set. $O(1)$.
- .remove(X) Remove X from the set. If X was not in the set, do nothing. $O(1)$.

```
HashMap<K, V>  // Where K and V are the key and value types, respectively.
```

- .size() Compute the size. $O(1)$.
- .containsKey(X) Determines if the map contains a value for the key X. To get that value, use .get(). $O(1)$.
- .get(X) Gets the value for the key X. If X is not in the map, return null. $O(1)$.
- .put(k, v) Map the key k to the value v. If there was already a value for k, replace it. $O(1)$.
- .keySet() Return a Set containing the keys in the map. Useful for iterating over. $O(1)$.

To iterate over a HashSet<T>, use

```
for (T v : nameOfSet) {
  // v is the current element of the set.
}
```

This can be combined with HashMap's .keySet() to iterate over a HashMap.

APT: Word Ladder I

Problem Statement

A *word ladder* is a sequence of words in which each word can be transformed into the next word by changing one letter. For example, the word ladder below changes 'lot' to 'log'.

```
lot dot dog log
```

This is not the shortest word-ladder between 'lot' and 'log' since the former can be immediately changed to the latter yielding a word ladder of length two:

```
lot log
```

The first and last words in a word ladder are the *anchor rungs* of the ladder. Any other words are *interior rungs*. For example, there are three interior rungs in the ladder below between 'smile' and 'evote'.

```
smile smite smote emote evote
```

In this problem you'll write a method that has parameters representing potential interior rungs: an array of strings (these may be nonsense or English words), and the anchor rungs --- two strings. Your code must determine whether there exists any ladder between the exterior rungs that uses at least one interior rung. If there is any ladder the method returns `"ladder"`, otherwise it should return `"none"`.

Class

```
public class WordLadder {
  public String ladderExists(String[] words,
                             String from, String to) {
       // fill in code here
  }
}
```

Notes and Constraints

- The parameters `from` and `to` are the anchor rungs, they must be connected by at least one interior rung from `words` or there are no valid word ladders.
- `words` contains at most 50 words.
- All strings contain only lowercase, alphabetic characters.
- All strings in `word` are the same length and are the same length as `from` and `to`.