15-~~111~~ 121 : Advanced Programming
Final Exam Summer 2008
June 27, 2008

Name: ____

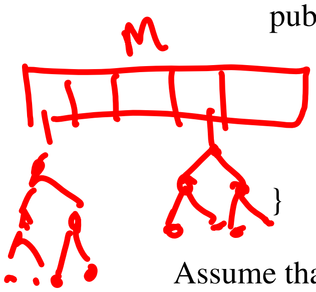
Andrew ID: _____________________________________

Answer the questions in the space provided following each question. We must be able to clearly understand your answer. If it is vague or confusing, it will be marked wrong. Be sure to read the directions to each question carefully. Answers to most questions are NOT long. Just carefully think about it before you write the answer. This exam is worth 20% of your final grade. You have ~~120~~ 80 minutes.

1. Runtime Complexity.

(a) Assume BST is an implemented class and you have a method height(), which has an optimal implementation, that finds the height of a balanced binary tree. What is the worst-case runtime complexity of the following code and why?

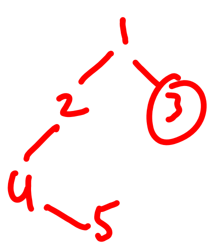
O(log n)



```
public int totalHeight(ArrayList<BST> list){
    int sum = 0;
    for each (tree in list)
        sum += height(tree);
    return sum;
}
```

$$height(T) = \begin{cases} -1 & T = \emptyset \\ 0 & |T| = 1 \\ 1 + max(h(T_L), h(T_R)) \end{cases}$$

Assume that list contains M elements, and each binary tree is *balanced* and contains N elements.



$h(1) = 1 + max(h(2), h(3))$

2 $\quad 1 + max(h(4), h(\emptyset))$

1 $\quad 1 + max(h(\emptyset), h(5))$

0

height = O(n)

O(M log N)

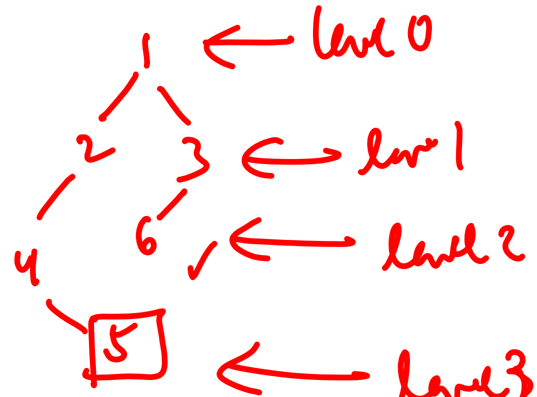
(b) You have a method minLeafLevel() that finds the level of the lowest leaf. The method is as implemented below. What is its worst-case runtime complexity and why? Justify your answer. Assume that a binary tree has *N* elements.

```
public int minLeafLevel() {
        return minLeafLevel(root);
}

private int minLeafLevel(Node p) {
        if (p == null) return -1;
        if (p.left == null)
            return 1 + minLeafLevel(p.right);
        if (p.right == null)
            return 1 + minLeafLevel(p.left);
        else
            return 1 + Math.min(minLeafLevel(p.left), minLeafLevel(p.right));
}
```

helper

f



$f(1) = 1 + \min(f(2), f(3))$

$1 + f(4)$

$1 + f(5)$

$1 + f(\phi)$

$1 + f(6)$

$1 + f(\phi)$

$O(n)$

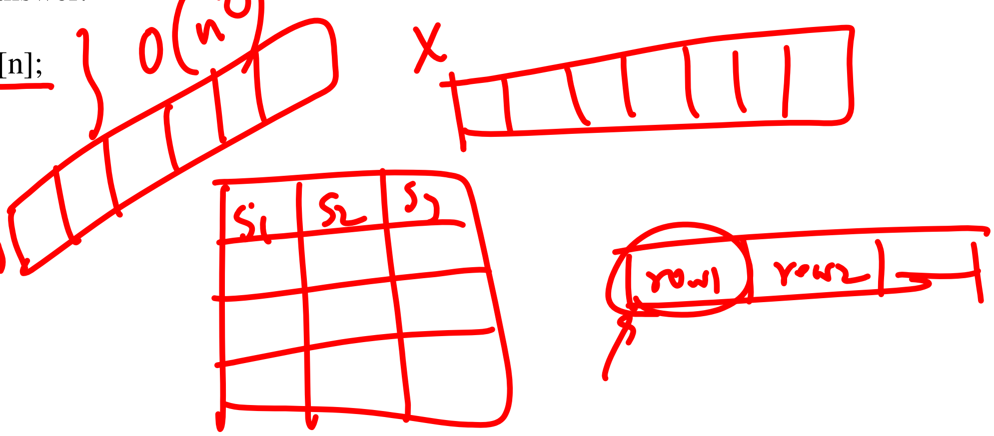
$= \boxed{2}$

(c) You have a method foobar(X) that iterates over each element in a one-dimensional array *X*. What is the time complexity of the following code? Describe in terms of O(1), O(n), O(n^2) or O(n^3). You need to justify the answer.

```
String [ ][ ] str = new String[n][n];
for (int k = 0; k < n; k++)
        foobar(str[k]);
```

$O(n^2)$

$O(n)$



2. Iterations.

Given two non-null strings, s1 and s2, write a method that finds the length of the longest contiguous string s3 such that both s1 and s2 contain s3. You assume that input strings are in lowercase.

public int substringMax(String s1, String s2)

String API

practice

s1 = "anandagunawardane"

s2 = "andygunafoo"

s3 = "guna"

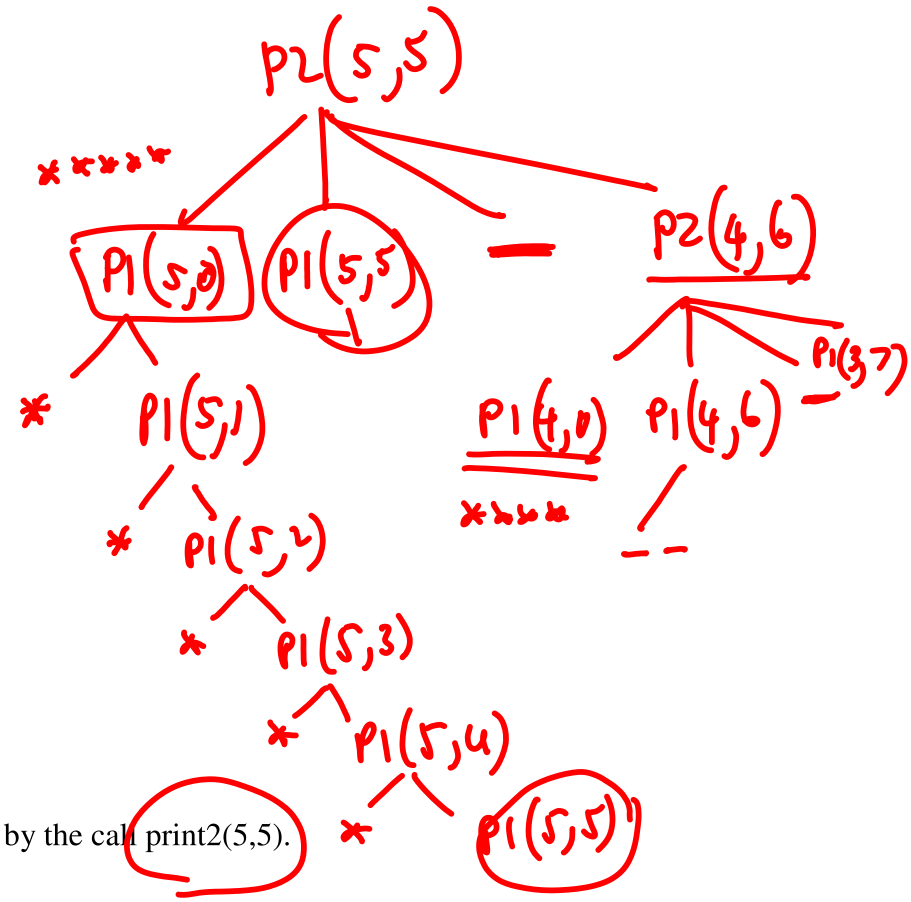
3. Recursion.

(a) Consider the following methods:

```
public void print1(int n, int m) {
        if (m < n) {
          System.out.print("*");
          print1(n, m+1);
        }
        else if (m > n) {
          System.out.print("-");
          print1(n, m-1);
        }
}

public void print2(int n, int m) {
  if (n > 0) {
     print1(n, 0);
     print1(n, m);
     System.out.println();
     print2(n-1, m+1);
  }
}
```

Show the output that would be produced by the call print2(5,5).



4. Sorting.

N

Given a million objects in memory (assume we have no memory constraints) all of type HighSchoolStudent:

14 – 19

```
public class HighSchoolStudent {
      private String name;
      private int numberOfYearsOld;
}
```

O(N)

What sorting algorithm would you use, and why would you use it, to sort these objects by the second field *numberOfYearsOld*? Consider all sorting algorithms we discussed and justify why you would use a particular algorithm. Pay careful attention to type of the sort key.

5. Stacks and Queues.

(a) Given two FIFO queues with data in sorted order, move all data to a third queue so that the third queue ends up with data again in sorted order. Implement your algorithm and give its worst-case runtime complexity. You may use all the standard queue operations. No data structure other than Queue can be used.

```
public Queue merge(Queue Q1, Queue Q2){
```

$Q_1 = 1 \quad 5 \quad 7$

$Q_2 = 2 \quad 4$

$Q_3 = 1, \ 2 \quad 4 \quad 5 \quad 7$

(b) Discuss two advantages of the Queue implementation based on a linked list over the Queue implementation based on an array.

6. Linked Lists.

(a) Is it possible to search a sorted, singly-linked list in worst-case $O(\log n)$? If it is possible, please explain an approach. If not, please explain your intuition.

(b) If it is likely that certain items within a linked list are more frequently used than others, the average-case performance of searching a linked list can often be improved by moving an item to the head of the list each time it is accessed. Please explain the intuition behind this technique.

7. Binary Search Tree.

(a) Given an array of numbers: 19, 6, 8, 11, 4, 13, 5, 27, 43, 49, 31, 25

Draw a binary search tree by inserting the above numbers from left to right and then show the resulting tree after 19 is removed. Assume the BST class given in Listing 2.

(b) Given an array of numbers: 19, 6, 8, 11, 4, 13, 5, 27, 43, 49, 31, 25

Create an AVL tree by inserting and balancing the tree using single or double rotations. Show all steps.

(c) Draw a binary tree T such that each node stores a single number and a preorder traversal of T yields 15,7,4,9,12,10,17,22,20,18,25 and an inorder traversal of T yields 4,7,9,10,12,15,17,18,20,22,25.

8. Hashing

(a) Exactly when would you use a hashtable to store a set of data?

(b) Given the keys, 97, 98, 67, 48, 98, 65, 45 and a table of size 10, show how the keys will be stored if (use the function key mod 10)

(i) Linear Probing is used:

(ii) Quadratic probing is used:

Listing 1: The Linked List Class.

```
import java.util.*;
public class LinkedList {
private Node head;
/**
* Constructs an empty list
*/
public LinkedList() {
head = null;
}
/**
* Returns true if the list is empty
*/
public boolean isEmpty() {
return head == null;
}
/**
* Inserts a new node at the beginning of this list.
*/
public void addFirst(Object item) {
head = new Node(item, head);
}
/**
* Returns a string representation
*/
public String toString() {
StringBuffer result = new StringBuffer();
for(Node tmp = head; tmp != null; tmp = tmp.next)
result.append(tmp.data + " ");
return result.toString();
}
private static class Node {
private Object data;
private Node next;
public Node(Object data, Node next) {
this.data = data;
this.next = next;
}
}
}
```

Listing 2: The Binary Search Tree Class.

```
import java.util.*;
public class BST{
private Node root;
public BST(){
root = null;
}
/**
* Inserts a new item
*/
public void insert(Comparable data){
root = insert(root, data);
}
private Node insert(Node p, Comparable toInsert){
if (p == null)
return new Node(toInsert);
if (toInsert.compareTo(p.data) == 0)
return p;
if (toInsert.compareTo(p.data) < 0)
p.left = insert(p.left, toInsert);
else
p.right = insert(p.right, toInsert);
return p;
}
/**
* Searches for an item
*/
public boolean search(Comparable toSearch){
```

```
return search(root, toSearch);
}
private boolean search(Node p, Comparable toSearch){
if (p == null)
return false;
else if (toSearch.compareTo(p.data) == 0)
return true;
else if (toSearch.compareTo(p.data) < 0)
return search(p.left, toSearch);
else
return search(p.right, toSearch);
}


private static class Node {
private Comparable data;
private Node left, right;
public Node(Comparable data) {
left = right = null;
this.data = data;
}
public Node(Comparable data, Node l, Node r){
left = l; right = r;
this.data = data;
}
public String toString() {
return data.toString();
}
}
}
```