Test 2: Compsei 100

Owen Astrachan

November 12, 2008

Name: ____________________________ (2 points)
Login: __________
Honor code acknowledgment (signature) __________________________________________

<table>
<thead>
<tr>
<th>Problem</th>
<th>Value</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem 1</td>
<td>15 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 2</td>
<td>24 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 3</td>
<td>16 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 4</td>
<td>8 pts.</td>
<td></td>
</tr>
<tr>
<td>TOTAL:</td>
<td>65 pts.</td>
<td></td>
</tr>
</tbody>
</table>

This test has 10 pages, be sure your test has them all. Do NOT spend too much time on one question — remember that this class lasts 75 minutes.

In writing code you do not need to worry about specifying the proper import statements. Assume that all libraries and packages we’ve discussed are imported in any code you write.

Unless indicated otherwise, the TreeNode class for this test is on the left. Some common recurrences and their solutions are on the right.

```java
public static class TreeNode {
    String info;
    TreeNode left;
    TreeNode right;
    TreeNode(String val, TreeNode lptr, TreeNode rptr) {
        info = val;
        left = lptr;
        right = rptr;
    }
}
```

<table>
<thead>
<tr>
<th>label</th>
<th>recurrence</th>
<th>solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>T(n) = T(n/2) + O(1)</td>
<td>O(log n)</td>
</tr>
<tr>
<td>B</td>
<td>T(n) = T(n/2) + O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>C</td>
<td>T(n) = 2T(n/2) + O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>D</td>
<td>T(n) = 2T(n/2) + O(n)</td>
<td>O(n log n)</td>
</tr>
<tr>
<td>E</td>
<td>T(n) = T(n-1) + O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>F</td>
<td>T(n) = T(n-1) + O(n)</td>
<td>O(n²)</td>
</tr>
<tr>
<td>G</td>
<td>T(n) = 2T(n-1) + O(1)</td>
<td>O(2^n)</td>
</tr>
</tbody>
</table>
Problem 1: (Frutti Tutti, List2Tree (15 points))

The method below returns a roughly balanced binary tree containing the same elements as in a doubly-linked list so that an inorder traversal of the binary tree visits the values in the same order as they’re stored from first to last in the linked list. In other words, if the linked list is in sorted/lexicographical order the binary tree will be a search tree.

```java
public TreeNode convert(DListNode list){
    if (list == null) return null;
    if (list.next == null){
        return new TreeNode(list.info,null,null);
    }
    DListNode first = list;
    DListNode middle = list;
    while (list != null && list.next != null){
        list = list.next.next;
        middle = middle.next;
    }
    DListNode second = middle.next;
    middle.prev.next = null;
    return new TreeNode(middle.info, convert(first), convert(second));
}
```

Part A (4 points)
What is the big-Oh runtime complexity of convert for an N-node list. Justify your answer.

Part B (2 points)
Briefly, what is the purpose of the line below in the method shown above.

```java
    middle.prev.next = null;
```

Part C (3 points)
The code above de-links nodes in the list and thus effectively “destroys” the list being converted to a tree. Explain how to add a few lines to reconstruct the list so that the method works, but the list isn’t de-linked/destroyed.

(this question continued)
Part D (3 points)
Explain why simply inserting nodes from a sorted linked list into a binary search tree using the code below results in $O(n^2)$ runtime for an n-node list.

```java
public TreeNode convert(DListNode list) {
    TreeNode root = null;
    while (list != null) {
        root = insert(root, list.info);
        list = list.next;
    }
    return root;
}

private TreeNode insert(TreeNode root, String info) {
    if (root == null) return new TreeNode(info, null, null);
    if (info.compareTo(root.info) <= 0) {
        root.left = insert(root.left, info);
    } else {
        root.right = insert(root.right, info);
    }
    return root;
}
```

Part E (3 points)
Explain both why changing the `insert` code above in Part D so that an AVL tree is created and added to (the `insert` code will change, it is not shown) will result in changing the complexity of the `convert` method and what the new complexity will be.
PROBLEM 2 :  (Finding Yourself (24 points))

Use the search tree below in answering questions for this problem.

Part A (3 points)
What is the postorder traversal of the subtree rooted at the narwhal node. List the post-order values in order, one per line. You should have six lines.

Part B (3 points)
Consider the code below for method size. What is returned by the call size(root) where root references the giraffe node in the diagram above?

Part C (3 points)
What is the running time of the method size below for an N-node tree. Use big-Oh and justify your answer.

```java
private int size(TreeNode root){
    if (root == null) return 0;
    return 1 + size(root.left) + size(root.right);
}
```

(this problem continued)
Part D (3 points)
Suppose the last value added to the tree above is jaguar. Before the addition of this value the tree is an AVL/height-balanced tree. After the addition the tree is no longer height-balanced. When the tree is rebalanced, lion will be the right child of giraffe. What value will be in the node that is the right child of lion after the tree is re-balanced to be an AVL tree after the addition of jaguar?

Part E (6 points)
The $k^{th}$ largest value in a collection of items is the smallest value that is larger than $k$ elements. The $0^{th}$ largest is the smallest, it’s larger than no other element. In a sorted array $a$ the $k^{th}$ largest is simply $a[k]$. The method findKthArray below returns the $k^{th}$ largest value in a binary search tree.

For example, in the tree at the beginning of this problem fox is the $3^{rd}$ largest element and monkey is the $8^{th}$ largest.

Briefly explain both why the method works correctly and what its running time is to find the $k^{th}$ largest from a binary tree of $N$ elements that is roughly balanced. Use big-Oh, justify your answer which should be in terms of $N$ and $k$ (or just one of them).

```java
private void fill(TreeNode root, ArrayList<String> list){
    if (root != null){
        fill(root.left,list);
        list.add(root.info);
        fill(root.right,list);
    }
}

public String findKthArray(TreeNode root, int k){
    ArrayList<String> list = new ArrayList<String>();
    fill(root,list);
    return list.get(k);
}
```

(this problem continued)
Part F (6 points)
A student proposes the method below to find the $k^{th}$ largest element in a binary tree of $N$ elements. It works correctly.

Briefly explain both why the method works correctly and what its running time is to find the $k^{th}$ largest from a binary tree of $N$ elements that is roughly balanced. Use big-Oh, justify your answer which should be in terms of $N$ and $k$ (or just one of them).

```java
public String findKth(TreeNode root, int k) {
    if (root == null) return null;

    int leftCount = size(root.left);
    if (leftCount == k) {
        return root.info;
    }
    else if (k < leftCount) {
        return findKth(root.left, k);
    }
    else {
        return findKth(root.right, k - leftCount - 1);
    }
}
```
The Program Schedule APT is available as the last page(s) of this test. For this question you’ll be asked to evaluate two different solutions to this problem. Both solutions are all-green, meaning they solve the problem correctly.

**Solution A**

```java
public class Pojo implements Comparable<Pojo>{
    int setup;
    int runtime;
    public Pojo(int s, int r){
        setup = s; runtime = r;
    }
    public int compareTo(Pojo o) {
        return o.runtime - runtime;
    }
}

public int schedule(int[] setUp, int[] runTime){
    int start = 0;
    int finish = 0;
    Pojo[] all = new Pojo[setUp.length];
    for(int k=0; k < setUp.length; k++){
        all[k] = new Pojo(setUp[k],runTime[k]);
    }
    Arrays.sort(all);
    for(Pojo p : all) {
        start = start+p.setup;
        finish = Math.max(finish,start+p.runtime);
    }
    return finish;
}
```

**Solution B**

```java
public int schedule(int[] setup, int[] runtime) {
    for(int j=setup.length-1; j >= 0; j--){
        for(int k=0; k < j; k++) {
            if (runtime[k] > runtime[k+1]) {
                int temp = setup[k];
                setup[k] = setup[k+1];
                setup[k+1] = temp;
                temp = runtime[k];
                runtime[k] = runtime[k+1];
                runtime[k+1] = temp;
            }
        }
    }
    int start = 0, finish = 0;
    for(int k=setup.length-1; k >= 0; k--){
        start += setup[k];
        finish = Math.max(finish,start+runtime[k]);
    }
    return finish;
}
```

**Part A (4 points)**

Both solutions choose to run the program that has the longest runtime of the programs that haven’t run. Circle the one line in each solution that is “most responsible” for sorting the runtimes in order so that eventually the maximal runtime is chosen to run first. You should circle the line that causes the sort to “do the right thing”.

(problem continued)
Part B (4 points)
What is the big-Oh runtime of Solution A for N-element arrays. Justify your answer briefly.

Part C (4 points)
What is the big-Oh runtime of Solution B for N-element arrays. Justify your answer briefly.

Part D (4 points)
A student claims that Solution B uses bubble sort and that using selection sort instead would result in a much faster program. Justify or refute the student’s claim.
PROBLEM 4:  

(Setup (8 points))

The method `intersect` below returns a set that represents the intersection of two sets; the intersection is the elements that are in both sets.

```java
public TreeSet<String> intersect(TreeSet<String> a, TreeSet<String> b){
    TreeSet<String> result = new TreeSet<String>();
    for(String sa : a){
        if (b.contains(sa)){
            result.add(sa);
        }
    }
    return result;
}
```

Part A (4 points)

In calling `intersect` as shown below the two sets `inter1` and `inter2` will contain the same values.

```java
TreeSet<String> inter1 = intersect(seta,setb);
TreeSet<String> inter2 = intersect(setb,seta);
```

Explain how the running times of the two calls could be drastically different. In particular, if set `seta` contains \( A \) elements and `setb` contains \( B \) elements express the running time of each call in terms of both \( A \) and \( B \) (use big-Oh, justify your answer).

Part B (4 points)

Will the running times of the two calls above be different if TreeSet is replaced everywhere by HashSet? Explain your answer.
(nothing on this page)