CSc 345: Analysis of Discrete Structures
Spring 2018 (Lewis)

Test 2
Thu 5 Apr 2018

Name: ___________________________________________ NetID: ___________________

Person to your left: __________________________ Person to your right: __________________

<table>
<thead>
<tr>
<th>Question</th>
<th>Points</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short Answer, Page 1</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>Short Answer, Page 2</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>insert()</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>Pretty Pictures, Page 1</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>Pretty Pictures, Page 2</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>search()</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>Induction (choose 1 of 2)</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>Total:</td>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>
1. For each question below, give a short answer - a few words or symbols, maybe a sentence or two.

(a) (9 points) The following descriptions each describe an operation that we might perform on a max heap. Name each operation.

“Append a value to end of the array, then bubble it up into the proper location.”

“Swap the first and last values in the array; then bubble the value which is now at [0] down, to its proper location. At the end, remove the last element from the array.”

“Starting at the middle of the array (and iterating toward the start of the array), bubble each value down to its proper location.”

(b) (5 points) Use the Master Method to solve the following recurrence. If the recurrence cannot be solved by the Master Method, give a short explanation why not.

\[ T(n) = 4T\left(\frac{n}{2}\right) + n^2 \]

(c) (6 points) A Splay Tree is a BST which can make guarantees about the “amortized” time of its operations. What is the absolute (not amortized) worst-case cost of an insert in a Splay Tree? What sort of tree shape would cause this to come about?

What is the amortized time cost for an insert in a Splay Tree? Explain what it means that this is less than the non-amortized time.
2. (a) (5 points) What is the AVL invariant - that is, what is true about every node in an AVL tree? (Other than the basic BST structure.)

(b) (5 points) What is a “collision” in a hash table? (Don’t worry about explaining how to handle this condition.)

(c) (10 points) The four rules for a red-black tree are below; fill in the blanks for each one.

The root must be ________________.

All leaves are black, and store ________________.

All leaves have the same ________________.

No ________________ next to ________________.
3. (10 points) Using the \texttt{x=change(x)} style, give the code for a Java method named \texttt{insert()}, which performs an insertion of a new key into a BST.

- The key type should be \texttt{int}; do not have a satellite value.
- If the key is a duplicate, do not change the tree; also \textbf{do not} throw any exception. (It’s just a NOP.)

The class for the node is as follows:

```java
public class BSTNode {
    public int key;
    public BSTNode left,right;

    public BSTNode(int key) { ... }
}
```

Do not write the class that contains this method, and don’t worry about how the method is called on the root of the tree. Instead, assume that there is code somewhere which does this already.
4. (a) (5 points) For each of the AVL trees below, a new node has just been inserted (it is highlighted), and now the AVL property is violated. Draw each tree after the proper rebalancing operations have been performed.

(b) (5 points) Insert the value 37 into the following 2-3-4 tree; draw the new tree. Use the top-down insertion method.
5. (15 points) Convert the following 2-3-4 tree to the red-black tree that emulates it. Mark each red node with the letter ‘R’.

![2-3-4 Tree Diagram]

- 42
  - 30
    - 6, 12, 24
    - 40
    - 45, 52
  - 55, 74
    - 61, 68, 72
    - 85, 91, 92
6. (15 points) **NOTE**: This problem is about 2-3 trees, **NOT** 2-3-4 trees!

   Give a **recursive** definition for `search()`, in a 2-3 tree. This takes a node reference (which might be `null`), and a key to search for; return the `String` associated with the key, or `null` if the key does not exist.

   I’ve adapted the node class that you used in Project 4 to have one less key and one less value, but otherwise it works the same way. I’ve provided the source for this at the end of this exam.

   Your algorithm **must** make use of the `numKeys` field.

   ```java
   String search(Node23 node, int key)
   {
   ```
Choose only one of the induction problems to do. Do not do both - if you do, we will only grade one of them!

7. (10 points) Consider a (non-empty) red-black tree. In this proof, we will count all of the nodes except for any red leaves. That is, we’ll count all of the black leaves, and all internal nodes of either color.

Using structural induction, prove that the number of these nodes is odd.

HINT 1: Use the black height (not the normal height!) as the induction variable.
HINT 2: I suggest the root+subtrees method - but use induction over the black height. That is, add an entire widget at the root in your inductive step; you will have several cases to consider.
Choose only one of the induction problems to do. Do not do both - if you do, we will only grade one of them!

Consider a Splay Tree, with $n$ keys inside it. You now search for each of the keys, in order - that is, you search for the first, then the second, and then the third. As normal, each search operation causes that value to be splayed to the root.

Prove the following conjecture using induction:

“After you have searched for the first $k$ keys, those keys are all arranged as a linked list, going left from the root of the tree. (The right subtree might have any shape at all; we don’t prove anything about it.)”

EXAMPLE TREE:
public class Node23
{
    // this tells us how many keys are stored in the node. In internal
    // nodes, it’s redundant, but it’s critical in leaf nodes. And it’s
    // very helpful in the internal nodes, as well, so we might as well
    // keep it.
    //
    // Possible values: 1,2
    // *NOT* 3

    public int numKeys;

    // these are the keys themselves; ignore fields which are not
    // valid, according to numKeys above.

    public int key1,key2;

    // each key is paired with a matching value variable (which must not
    // be null for a valid key)

    public String val1,val2;

    // if this is a leaf node, then *ALL* of these pointers must be
    // null. If not, then the number of non-nulls should be exactly
    // one more than the number of keys above.

    public Node23 childA,childB,childC;
}