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

Test 2  
Thu 5 Apr 2018

Solutions

<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>

Name: ________________________________ NetID: ________________

Person to your left: ___________________ Person to your right: ___________________
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.”

Solution: insert

“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.”

Solution: remove max

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

Solution: build max heap

(Technically, the term “heapify” is used to describe the process of moving a single node to its proper location. However, we’ll allow it as an answer to this question.)

(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.

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

Solution:

\[
\begin{align*}
  a &= 4 \\
  b &= 2 \\
  \log_b a &= \log_2 4 = 2 \\
  f(n) &= n^2 \\
  This is Case 2, because f(n) = \Theta(n^{\log_2 4}). \\
  T(n) &= \Theta(n^2 \log n)
\end{align*}
\]

(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?

Solution: \(O(n)\)

A linked-list tree might cause this cost (where we need to search all the way to the end of the list).

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.

Solution: \(O(\log n)\)

“Amortized” time is the average over all of the operations. Thus, in a Splay Tree, while individual operations might be quite terrible, on average, the cost per operation is must less than the max.
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.)

Solution: The heights of the two child subtrees must differ by at most one.

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

Solution: A collision is when two keys hash to the same value, and so they are both trying to use the same slot in the array.

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

The root must be ______ black ______.

All leaves are black, and store ______ nothing ______.

All leaves have the same ______ black height ______.

No ______ red nodes ______ next to ______ red nodes ______.
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:

\begin{verbatim}
public class BSTNode {
    public int key;
    public BSTNode left,right;

    public BSTNode(int key) { ... }
}
\end{verbatim}

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.

\begin{solution}
\textbf{Solution: Instructor’s Note:} We will accept \texttt{public}, \texttt{private}, or neither.
Likewise, \texttt{static} is generally the correct designation for an \texttt{x=change(x)} method, since we can’t assume that there is any (non-null) node that would be the “current” object. However, we’ll accept this function without that keyword, no penalty.
The \texttt{type} that is returned, and the type of the parameter, however, are \textbf{very} important!
Also, we do \textbf{not} care about the order of the parameters in this method.

\begin{verbatim}
public static BSTNode insert(BSTNode root, int key) {
    if (root == null)
        return new BSTNode(key);
    if (root.key == key)
        return root;
    if (key < root.key)
        root.left = insert(root.left, key);
    else
        root.right = insert(root.right, key);
    return root;
}
\end{verbatim}
\end{solution}
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’.

Solution:

NOTE: The 2-3-4 nodes with two keys (45,52; 55,74) can be expressed as two different widgets (right-leaning or left-leaning). Either (or one of each) will be accepted.
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)
{
    Solution:
    if (node == null)
        return null;
    if (key < node.key1)
        return search(node.childA, key);
    if (key == node.key1)
        return node.val1;
    if (node.numKeys == 1 || key < node.key2)
        return search(node.childB, key);
    if (key == node.key2)
        return node.val2;
    return search(node.childC, 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.

Solution:

Instructor’s Note: As described in the notes during class, this question assumes that we are ignoring the “virtual nodes.” If this was not true, then there would be no such thing as red nodes!

Base: A single widget

A single red-black widget is made up of one to three nodes. However, only one of them is black; all other nodes in the widget are red leaves, and thus not counted; thus, the count is odd.

Thus, the base case holds.

Inductive:

Assume that the conjecture holds for trees with black height \( k \). We will prove that it also holds for trees with black height \( k + 1 \).

A red-black tree with black height \( k + 1 \) is a single root widget (1-3 nodes), with a certain number of subtrees, each with black height \( k \). By the I.H., all of the subtrees have a count which is odd.

We note that red leaves in the various subtrees are also red leaves in the larger tree - and thus are ignored in both the larger and smaller trees.

Moreover, all of the nodes in the root widget (red or black) are internal nodes, and thus must be included in the count.

There are 3 cases: the root widget may have 1, 2, or 3 nodes. In all three cases, the number of subtrees is one more than the number of nodes in that widget.

If there are an odd number of nodes in the root widget, then there are an even number of subtrees. So, the total count is:

\[
\text{count} = \text{rootWidget} + \text{subtrees} = \text{odd} + \text{even} = \text{odd}
\]

However, if there are exactly 2 nodes in the root widget, then there are exactly three subtrees, and the total count is:

\[
\text{count} = 2 + \text{subtree}_1 + \text{subtree}_2 + \text{subtree}_3 = \text{even} + \text{odd} + \text{odd} + \text{odd} = \text{odd}
\]

Thus, in all cases, the count in the \( k + 1 \) tree is odd.

Thus, the inductive step holds.
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:

![Example Tree Diagram]

Solution:

Base: The first key is splayed

In the first step \( k = 1 \), we find the minimum value in the entire tree, and splay it. It becomes the new root; it has no left child (because it is the minimum). Thus, the conjecture is trivially true for the base case.

Inductive:

Assume that the conjecture has held while we splay the first \( k \) values. We will now splay the \( k + 1 \) value, and demonstrate that the conjecture is still true.

We observe that the root of the entire tree, after the splay of the \( k \) value, must of course be the \( k \) value itself. Thus, the \( k + 1 \) value is in the right subtree - and we know that it must be the minimum value in that subtree.

When we splay this value, there may be an arbitrary number of Zig-Zig and Zig-Zag operations. The last step, however, will be either a Zig operation (the \( k + 1 \) value is currently the immediate right child of the root), or a Zig-Zag (the \( k + 1 \) value is the right-left child of the root). In either case, the value we’re splaying will become the new root of the tree; the \( k \) value will become its left child, and the entire rest of the left section of the tree will be attached as the left child of the \( k \) value.

Thus, the structure of the tree after the \( k + 1 \) is splayed still fulfills the conjecture.

Thus, the inductive step holds.
This page intentionally left blank.
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;
}