Homework 8

OPTIONAL PROBLEMS

Due never, do now

"I understand why marriages break up over golf. I can't even talk about my own handicap because it's too upsetting." -*Shia Labeouf*

1 Written Problems

Problem 8.1

Moar Treaps

Prove (by strong induction) that any given collection $(k_1, p_1), \ldots, (k_n, p_n)$ of key-priority pairs, where all keys are distinct and all priorities are distinct, there is a unique treap T with n nodes, where each node contains a different key-priority pair. "Unique" means that there is only one way to arrange the treap for a given set of inputs.

Note: Strong induction works the same way as regular induction, except instead of assuming $P(k)$ and showing $P(k+1)$, you assume $P(i)$ for all $i \leq k$, and show that $P(k+1)$ follows from that.

Solution:

Base Case: $n = 0$. There is only one way a treap with no nodes can be constructed... it just won't have any nodes!

Inductive Assumption: Assume that there is only one way to construct a treap of size i for all $0 \leq i \leq k$.

Want to Show: There is only one way to construct a treap of size $k+1$.

Inductive Step: Given a treap of size $k+1$, there must be one unique element with the lowest priority value. To satisfy the heap condition, this node must be the root of T.

To satisfy the BST property, T_{left} must contain the remaining items whose keys are smaller than the root's and T_{right} must contain those whose keys are larger.

Both T_{left} and T_{right} must have between 0 and k elements each. By our inductive assumption, those subtrees are unique.

Since there is a unique root and a unique right and left subtree of the root, the treap of size $k+1$ must be unique.

Conclusion: We've proven that a treap of size 0 is unique. We've also proven that if all treaps of size $0 \leq i \leq k$ are unique, then all treaps of size $k+1$ must be unique. Therefore, we have proven that all treaps of size ≥ 0 are unique.

Problem 8.2

Sorting Nodes by Depth

Given a binary search tree, design an algorithm which creates a linked list of all the nodes at each depth. For example, if you have a tree with depth D, you'll have D linked lists. Your function should take in the root of the BST (which has pointers to any child nodes it may have), and return a list of linked lists.

Solution:

```
public List<LinkedList> makeDepthLinkedLists(TreeNode root):
        """makeDepthLinkedLists: TreeNode -> list
        Purpose: return a list of D linked lists of nodes at each depth
        """
        depthListList = List<LinkedList>;
        if root==null:
                return depthListList;
        prevList = LinkedList<TreeNode>();
        prevList.push(root);
        currList = LinkedList<TreeNode>();
        while prevList is not empty:
                currList = fillList(prevList, currList);
                depthListList.push(dList);
                prevList = currList;
                currList = LinkedList();
        return depthListList;



public LinkedList<TreeNode> fillList(LinkedList prevList, LinkedList currList):
        """fillList: LinkedLists prevList and currList -> LinkedList
        Purpose: return the LinkedList of nodes from the current depth level
        """
        for node in prevList:
                if curr has left child:
                        currList.push(curr.left)
```

```
                if curr has right child:
                        currList.push(curr.right)
        return currList
```

Problem 8.3

Rotated Array

Given a sorted array of n integers that has been rotated an unknown number of times, give an O(logn) algorithm that finds an element in the array. You may assume that the array was originally sorted in increasing order and there are no duplicates.

Solution:

Use a modified version of binary search!

```
function rotated(array, key):
    low = 0
    high = size of array -1
    while low <= high:
            mid = low + ((high - low) / 2)
            if array[mid] == key:
                    return mid
            if array[low] <= array[mid]:
                    if array[low] <= key and key < array[mid]:
                            high = mid -1
                    else:
                            low = mid +1

            else: //upper half is sorted
                    if array[mid] < key and key <= array[high]:
                            low = mid +1
                    else:
                            high = mid -1

    return -1
```

Problem 8.4

Rotated Array Episode II

It is given that in the array described above, the smallest integer has a value of 1, and the array contains only numerically consecutive integers. Write pseudocode to find how many times the original array was rotated.

Solution:

Iterate through the array until you find 1. The index of 1 is the number of times that the array has been rotated:

```
function rotate2(array):
        for i from 0 to end of array:
                if array[i] == 1:
                        return i
        return -1
```
