CS61B Spring 2016 Guerrilla Section 3 Worksheet

12 March 2016

Directions: In groups of 4-5, work on the following exercises. Do not proceed to the next exercise until everyone in your group has the answer and *understands why the answer is what it is.* Of course, a topic appearing on this worksheet does not imply that the topic will appear on the midterm, nor does a topic not appearing on this worksheet imply that the topic will not appear on the midterm.

1 Asymptotic Approach

Find the run time of the following code in Θ notation:

```
int foo (int n) {
    if (n == 0)
        return 0;
    baz(n);
    return foo(n/3) + foo(n/3) + foo(n/3);
}

int baz (int n) {
    for (int i = 0; i < n; i++){
        System.out.println("Help me! I am trapped in a loop");
    }
    return n;
}
```

To help, please use the following table to organize your work:

Level	Number of Nodes	Work per Node	Total Amount of Work

$O(n \log(n))$
This runs in nlogn time as each baz call will run for n time and then each foo call will be called with 1/3 of the initial run time.

STOP!

Don't proceed until everyone in your group has finished and understands all exercises in this section!

2 Asymptotic Potpourri

For each of the following code snippets, give a bound for the run time with respect to the input n in Θ notation:

(a)

```
public void mystery1(int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            i = i*2;
            j = j*2;
        }
    }
}
```

$O(\log(n))$
Explanation: Both i and j are doubled in the inner for-loop, so it takes log(n) time for both i and j to be incremented to n.

(b)

```
public void mystery2(int n) {
    for (int i = n; i > 0; i = i/2) {
        for (int j = 0; j < i*2; j++) {
            System.out.println("Hello World");
        }
    }
}
```

$O(n)$
Explanation: Doubling i in the inner loop does nothing to the run time, it makes it similar to the one shown in lecture. This becomes $2n + n + \frac{n}{2} + \cdots + 2$

(c)

```
public void mystery3(int n) {
    for (int i = n; i > 0; i = i/2) {
        for (int j = 0; j < i*i; j++) {
            System.out.println("Hello World");
        }
    }
}
```

$O(n^2)$
Explanation: By squaring i in the inner loop, the run time will now be this sum: $n^2 + (n/2)^2 + \cdots + 1$

(d)

```
public void mystery4(int n) {
   int i = 1, s = 1;
   while (s <= n) {
        i++;
        s = s + i;
        System.out.println(s)
    }
}
```

$O(\sqrt{n})$
Explanation: Finishes when $s = 1 + 2 + 3 + \cdots + k - 2 + k - 1 + k + (k + 1) > n$, where k is number of iterations
Can think of if as:
$2s = 1+2+3+\cdots+k+(k+1)+(k+1)+(k)+(k-1)+\cdots+2+1 = (k+2)(k+1) \rightarrow s = \frac{k^2}{2} > n, k > sqrt(n)$
or shortcut explanation: $s = (1 + 2 + \cdots + k) < (k + k + \cdots + k) = k2$ as upper bound
Source: `https://www.quora.com/Why-is-the-runtime-of-this-function-O-sqrt-n`

STOP!
Don't proceed until everyone in your group has finished and understands all exercises in this section!

3 Stacks of Fun

Recall the SQueue from Discussion 5. The SQueue is a queue implemented using two stacks. Below is the code for a definition of this class:

```
public class SQueue {
    private Stack inStack;
    private Stack outStack;
    public SQueue() {
        inStack = new Stack();
        outStack = new Stack();
    }

    public void enqueue(int item) {
        inStack.push(item);
    }

    public void dequeue() {
        if (outStack.isEmpty()) {
            while (!inStack.isEmpty()) {
                outStack.push(inStack.pop());
                }
        }
        return outStack.pop();
    }
}
```

Now, suppose we construct an SQueue and enqueue 100 items:

(a) How many calls to push and pop result from the next call to dequeue?
201

(b) How many calls to push and pop result from each of the next 99 calls to dequeue?
1

(c) How many calls to push and pop (total) were required to dequeue 100 elements? How many operations is this per element dequeued?
300 operations, or 3 per dequeue

(d) What is the worst-case time to dequeue an item from an SQueue containing N elements? What is the runtime in the best case? Answer using big-theta notation. You may assume that both push and pop run in $\Theta(1)$.
Worst: $\Theta(N)$, Best: $\Theta(1)$

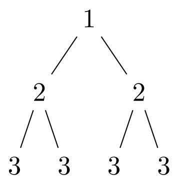
(e) What is the amortized time to dequeue an item from an SQueue containing N elements? Again, answer using Θ notation.
$\Theta(1)$

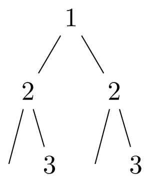
STOP!

Don't proceed until everyone in your group has finished and understands all exercises in this section!

4 Symmetric Tree

Given a binary tree, check whether it is a mirror of itself (ie, symmetric around its center).

For example, this binary tree is symmetric: 

But this tree is not: 

Definition of TreeNode:

```
public class TreeNode {
     int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int x) { val = x; }
}

public class Symmetric {
    public boolean isSymmetric(TreeNode root) {
        // your code here
        /* The following is the solution */
        if (root == null) return true;
        return isSymmetric(root.left,root.right);
    }

    public boolean isSymmetric(TreeNode a, TreeNode b){
        if (a == null) return b == null;
        if (b == null) return false; // when only one of them is null
        if (a.val != b.val) return false;
        return isSymmetric(a.left, b.right) && isSymmetric(a.right, b.left);
    }
}
```

STOP!

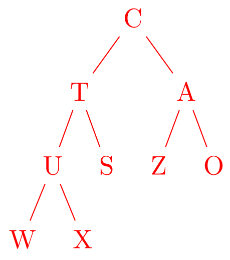
Don't proceed until everyone in your group has finished and understands all exercises in this section!

5 Greetings, Tree Traveler

(a) Draw a full binary tree that has the following preorder and postorder. Each node should contain exactly one letter. A full binary tree is a tree such that all nodes except leaf nodes have exactly 2 children.

(1) Pre-order: C T U W X S A Z O

(2) Post-order: W X U S T Z O A C



(b) What is the inorder of this tree?
W U X T S C Z A O

(c) Can a tree have the same in-order and post-order traversals? If so, what can you say about the tree?
Yes. The tree can only have left children. (In-order gives left-root-right, post-order gives left-right-root).

(d) What about a tree with the same pre-order and post-order traversals?
Yes, but only if the tree has one element or is empty.

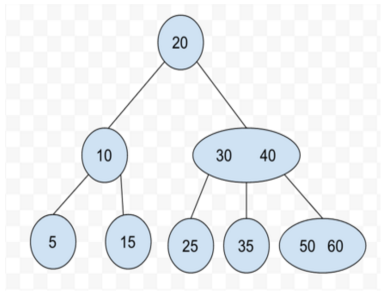
STOP!

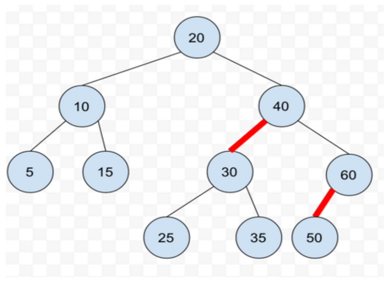
Don't proceed until everyone in your group has finished and understands all exercises in this section!

6 Balanced Search Tree Mechanics

(a) Insert the following numbers in order into a 2-3 tree: 20, 10, 35, 40, 50, 5, 25, 15, 30, 60

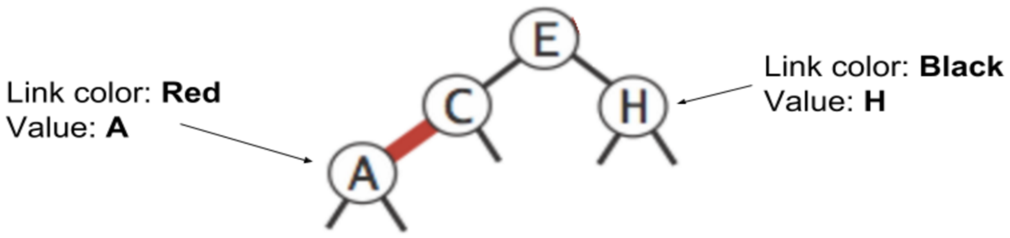
(b) Draw a red-black tree that corresponds to the 2-3 tree you drew in part a.

Part (a) soln 

Part (b) soln 

7 What color am I?

For each of the situations below in a valid LLRB tree, indicate whether the node?s link to its parent is red or black



Options:

(a) Red

(b) Black

(c) Either red or black

Questions:

(1) _B_ The largest value in a tree with more than one node.
The largest value is always a right child, so must be black

(2) _C_ The smallest value in a tree with more than one node.
A two node 2-3 tree has a red smallest child. A three node (insert 1,2,3) 2-3 tree has a black smallest child

(3) _B_ A node whose parent is red (parent's link color is red).
We cannot have two consecutive red nodes

(4) _C_ A node whose children are the same color.
A node with two children of the same (which must be black) can be either red or black

(5) _C_ A freshly inserted node after the insertion operation is completed.
When the insertion operation for a BST has completed, the node may be either red or black depending on rotations and color flips

STOP!

DON'T PROCEED UNTIL EVERYONE IN YOUR GROUP HAS FINISHED AND UNDERSTANDS ALL EXERCISES IN THIS SECTION!

8 Weight Times

A Quick Union data structure is used to handle set union and membership operations. The supported methods are:

(i) connect(a, b) - connects the set of a to the set of b

(ii) isConnected(a, b) - returns true if a and b are in the same set

Example:

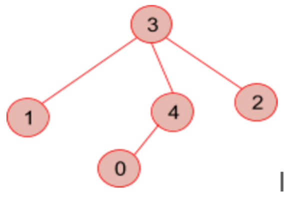
```
connect(a, b)
connect(b, c)
connect(a, d)
isConnected(c, d) //returns true
isConnected(d, b) //returns true
```

Internally, a Quick Union?s sets are represented using trees. Sets can be connected by adding one set?s tree to the root of another set?s tree. Note that Weighted Quick Union data structures are similar to Quick Union data structures, except that a Weighted Quick Union will always add the shorter tree to the root of the taller tree during connect operations.

(a) Draw the Weighted Quick Union object that results after the following four method calls:

```
connect(1, 3)
connect(0, 4)
connect(0, 1)
connect(0, 2)
```

There are many possible solutions for this problem. It?s most important to note that, when connecting sets, we're only connecting at the root.



(b) What is the resulting array of the Weighted Quick Union after the calls in (a) are executed?
|3|3|3|3|3| (again, this is one of many possible solutions)

STOP!

DON'T PROCEED UNTIL EVERYONE IN YOUR GROUP HAS FINISHED AND UNDERSTANDS ALL EXERCISES IN THIS SECTION!

9 Weighted Quick Union

(a) In terms of runtime, what is the worst way to place the integers 1,2,3,4, and 5 into the same set? Your answer should be in the form of a series of calls to the connect method.

```
connect(1,2)
connect(2,3)
connect(3,4)
connect(4,5)
```

(b) Assume a single node has a height of 0, what is the shortest and tallest height possible for a Quick Union object with 10 elements?
Shortest: 1, Tallest: 9

(c) In general, what are the shortest and tallest heights possible for a Quick Union with k elements? What does this mean for the best and worst case runtimes for isConnected and connect?
Shortest: 1, Tallest: k-1
isConnected $\rightarrow$ Worst: $\Theta(n)$, Best: $\Theta(1)$
connect $\rightarrow$ Worst: $\Theta(n)$, Best: $\Theta(1)$

(d) What is the shortest and tallest height possible for a Weighted Quick Union with 10 elements? How about a Weighted Quick Union with N elements? What does this mean for the best and worst-case runtimes for isConnected and connect?
10 items Shortest: 1, Tallest: 3
N items $\rightarrow$ Shortest: 1, Tallest: $floor(logk)$
isConnected $\rightarrow$ Worst: $\Theta(log(n))$, Best: $\Theta(1)$
connect $\rightarrow$ Worst: $\Theta(log(n))$, Best: $\Theta(1)$

STOP!

Don't proceed until everyone in your group has finished and understands all exercises in this section!

10 Mutual Recursion

Give a tight bound for the run-time of foo(p, q) in terms of p and q using θ notation. Assume p and q are positive.

```
public static int foo(int n, int m) {
    if (m == 0) {
        return bar(n - 1, n);
    }
    return foo(n, m - 1);
}
public static int bar(int x, int y) {
    if (x == 0) {
        return 1;
    }
    return foo(x, 3*y);
}
```

$O(p^2 + q)$
Explanation: The first call to `foo` recurses m times before calling `bar`. Then `bar` calls `foo` p times total, each time using $3q$ as the argument for m in `foo`. Each of these calls takes $3p$ time since p was passed into `bar` the first time as the argument for q. This comes out to a total of $O(q)$ time for the first call to `foo`, plus $O(p^2)$ time for the remaining recursive calls.

STOP!

DON'T PROCEED UNTIL EVERYONE IN YOUR GROUP HAS FINISHED AND UNDERSTANDS ALL EXERCISES IN THIS SECTION!

11 Your Node is my Node

Consider the Node class below. We've seen how a simple class like this can be used to power very versatile data structures, including Doubly Linked Lists and Binary Search Trees. Since both of these data structures use the same Node types, it is actually possible to transform an instance of a BST into a sorted DLL and vice versa. Fill in the function treeToList below such that it takes in the root Node of a BST and transforms it destructively into a DLL with the same ordering. You may use the extra space to implement helper functions as needed.

```java
public class Node {
    public int item;
    public Node left;
    public Node right;
    public Node(Node left, int item, Node right) {
        this.left = left;
        this.item = item;
        this.right = right;
    }
    public Node(int item) {
        this(null, item, null);
    }
}
```

```java
public class TreeList {
    /* Complete the treeToList method below. */
    public static Node treeToList(Node root) {
        if (root == null) { return null; }

        /* Transform the left and right sides recursively */
        Node leftList = treeToList(root.left);
        Node rightList = treeToList(root.right);

        /* Transform the root into a Doubly Linked List */
        root.left = root;
        root.right = root;

        /* Extend the left list to include the root */
        leftList = extend(leftList, root);

        /* Extend the left list the include the right list */
        leftList = extend(leftList, rightList)

        return leftList;
    }

    /* Put any helper methods you need here */

    /* Extends the leftList with the elements of the rightList.
       Assumes that the given Nodes represent circular DLLs */
    private static Node extend(Node leftList, Node rightList) {
        if (leftList == null)  { return rightList; }
        if (rightList == null) { return leftList; }
        Node endOfLeft = leftList.left;
        Node endOfRight = rightList.left;
        link(endOfLeft, rightList);
        link(endOfRight, leftList);
        return leftList;
    }

    /* Makes b the next node of a, and a the previous node of b */
    private static void link(Node a, Node b) {
        a.right = b;
        b.left = a;
    }
}
```

STOP!

Don't proceed until everyone in your group has finished and understands all exercises in this section!