Midterm Exam 3, Compsci 201
Spring 2022, Duke University

April 19, 2023

General Directions

Before you begin, **make sure to write your name and NetID on the exam, read the Duke community statement, and sign indicating your understanding and agreement to these directions**. You are encouraged to write your NetID on every page of the exam where indicated in the event that pages become separated during scanning.

Every question on the exam will have a box indicating where you should write your answer. Answers outside of the corresponding box will not be graded.

For all problems you should assume that any necessary libraries (for example, from `java.util`) are imported. Where relevant, give the most tight analysis you can using big O notation. For example, if the running time is $O(N)$ then answering $O(N^2)$, while technically true, will not earn full credit.

You may not communicate with anyone while completing this exam. You may not discuss this exam with anyone else on the day of the exam. You may not access any electronic devices (including but not limited to phones, smartwatches, laptops, etc.) during the exam period. If you need to leave the exam room during the exam period, you should not communicate with anyone and should not access any electronic devices. You are allowed one 8.5x11 in. reference sheet.

Duke Community Standard

Duke University is a community dedicated to scholarship, leadership, and service and to the principles of honesty, fairness, respect, and accountability. Citizens of this community commit to reflect upon and uphold these principles in all academic and nonacademic endeavors, and to protect and promote a culture of integrity.

To uphold the Duke Community Standard:

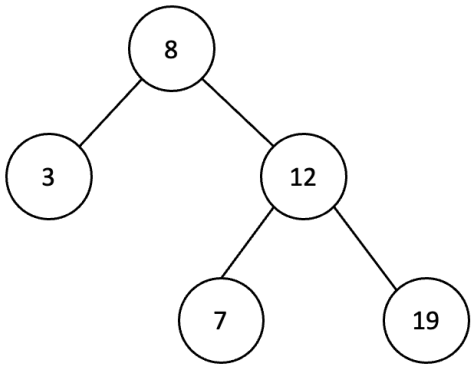
- I will not lie, cheat, or steal in my academic endeavors;
- I will conduct myself honorably in all my endeavors; and
- I will act if the Standard is compromised.

Print Name. Sample Solution **NetID.** _____

Signature. _____ **Date.** _____

```java
public class TreeNode {
    int info;
    TreeNode left;
    TreeNode right;
    TreeNode(int x){
        info = x;
    }
    TreeNode(int x, TreeNode lNode, TreeNode rNode){
        info = x;
        left = lNode;
        right = rNode;
    }
}
```

Figure 1: TreeNode class. Several questions that follow refer to the TreeNode class.



(a) Binary Tree

```java
public void traverse(TreeNode node) {
    if (node != null) {
        System.out.print(node.info + " ");
        traverse(node.left);
        traverse(node.right);
    }
}
```

(b) traverse method.

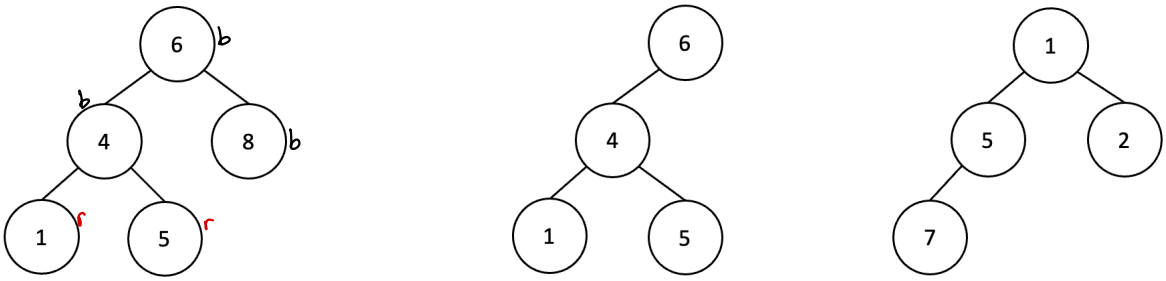
Figure 2

1. (3 points). Consider the binary tree shown in Figure 2a. Suppose we call the `traverse` method defined in Figure 2b on the root of this tree, the `8` node. What will be printed? You do not need to explain your answers.

Answer: 8 3 12 7 19

2. (3 points). Again consider the binary tree shown in Figure 2a. Is this a binary search tree? Briefly explain your answer.

Answer: Not a binary search tree. 7 is less than 8, but is in the subtree rooted at the right child of 8. Thus, for example, the binary search tree search algorithm would incorrectly report that 7 is not in this tree, because it would only search in the left subtree from the root.



(a) Tree A (b) Tree B (c) Tree C

Figure 3: Three Example Binary Trees

3. (4 points). Consider the binary trees shown Figures 3a and 3b. Are these red-black trees? If so, state a valid coloring (that is, for each node, say whether it is black or red, such that the overall result satisfies the red-black tree properties) If not, briefly explain why there is no valid coloring.

A. Tree A, shown in Figure 3a

Answer: YES, is a red-black tree.
For example: can color 6, 4, and 8 black and 1 and 5 red.

B. Tree B, shown in Figure 3b

Answer: NO, is not a red-black tree. The root (6) is required to be black, so there are 2 black nodes on the rightmost 6→null path (counting null as black). Because a red node cannot have a red child, at least 1 of the 4 or 1 nodes must be black, so the 6→4→1→null path must have at least 3 black nodes (again counting null as black). This would violate the path property that all root to null paths have the same number of black nodes.

4. (4 points). Consider the binary heap visualized in Figure 3c. We write the array representation of the heap using the 1-indexing convention that leaves an `X` at the beginning to mark the blank position 0. For example, we write the array representation of Figure 3c as: `[X, 1, 5, 2, 7]`.

A. Suppose you add 3 and then 1 to the binary heap, reestablishing the binary heap invariants after each addition. What would the array-representation of the resulting binary heap be? You do not need to explain your answer.

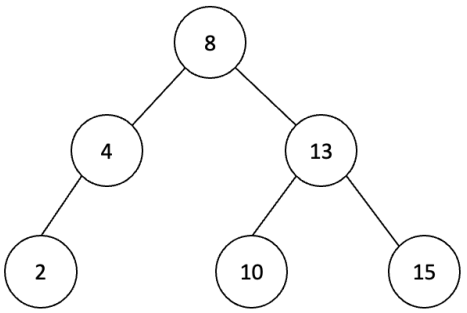
Answer: [X, 1, 3, 1, 7, 5, 2]

B. Suppose you remove the minimum value from the binary heap (as visualized in Figure 3c, without the additions from the previous problem). After reestablishing the binary heap invariants, what would the array-representation of the resulting binary heap be? You do not need to explain your answer.

Answer: [X, 2, 5, 7]

```
1  public int size(TreeNode root) {
2      if (root == null) { return 0; }
3      return 1+size(root.left)+size(root.right);
4  }
5
6  public int largestPerfectSubtree(TreeNode root) {
7      if (root == null) { return EXPR_1; }
8      int lPerf = largestPerfectSubtree(root.left);
9      int lSize = size(root.left);
10     int rPerf = largestPerfectSubtree(root.right);
11     int rSize = size(root.right);
12     if (lPerf==lSize && rPerf==rSize && lPerf==rPerf) {
13         return EXPR_2;
14     }
15     else if (lPerf > rPerf) {
16         return EXPR_3;
17     }
18     else {
19         return EXPR_4;
20     }
21 }
```

(a) size and largestPerfectSubtree



(b) Example Tree

Figure 4

5. (8 points). A binary tree is **perfect** if all interior (non-leaf) nodes have exactly two child nodes, and the leaf nodes all have the same depth. In the example tree shown in Figure 4b, the overall tree is not perfect because an interior node (the 4 node) has one child node. It still would not be perfect even if the 2 node were removed, because then the 4 node would be a leaf at a different depth than the 10 and 15 leaf nodes. It would be a perfect tree if we added a right child node to the 4 node.

Consider the code shown in Figure 4a. The `size` method is complete and correctly computes the number of nodes in a binary tree (including the `root` node). The `largestPerfectSubtree` method is partially complete. It should return the size of the largest perfect subtree. For example `largestPerfectSubtree(root)` where `root` refers to the 8 node in the example shown in Figure 4b should return 3, because the largest perfect subtree is the subtree rooted at the 13 node of size 3.

There are four expressions, `EXPR_1`, `EXPR_2`, `EXPR_3`, and `EXPR_4` missing in the code shown. What should the code be for these so that the method will work correctly? You do not need to explain your answers.

A. `EXPR_1`:

0

B. `EXPR_2`:

lPerf + rPerf + 1 (or equivalent)

C. `EXPR_3`:

lPerf

D. `EXPR_4`:

rPerf

```
1  public int size(TreeNode root) {
2      if (root == null) { return 0; }
3      return 1+size(root.left)+size(root.right);
4  }
5
6  public int largestPerfectSubtree(TreeNode root) {
7      if (root == null) { return EXPR_1; }
8      int lPerf = largestPerfectSubtree(root.left);
9      int lSize = size(root.left);
10     int rPerf = largestPerfectSubtree(root.right);
11     int rSize = size(root.right);
12     if (lPerf==lSize && rPerf==rSize && lPerf==rPerf) {
13         return EXPR_2;
14     }
15     else if (lPerf > rPerf) {
16         return EXPR_3;
17     }
18     else {
19         return EXPR_4;
20     }
21 }
```

(a) size and largestPerfectSubtree

Recurrence	Solution
T(n) = T(n/2) + O(1)	O(log n)
T(n) = T(n-1) + O(1)	O(n)
T(n) = 2T(n/2) + O(1)	O(n)
T(n) = T(n/2) + O(n)	O(n)
T(n) = 2T(n/2) + O(n)	O(n log n)
T(n) = T(n-1) + O(n)	$O(n^2)$

(b) Recurrence Solutions

Figure 5

6. (6 points). Again consider the `size` and `largestPerfectSubtree` methods, the definitions of which are repeated on this page in Figure 5a. This question will ask about the asymptotic runtime complexity of the `largestPerfectSubtree` method as a function of N, where N is the number of nodes in the binary tree rooted at `root`. You may assume that the missing expressions all have constant runtime complexity. Note that solutions to common recurrence relations are provided in Figure 5b.

A. What is the asymptotic runtime complexity of `largestPerfectSubtree` on a **balanced** tree? Justify your answer by stating and explaining a recurrence relation, referencing the code.

Answer: $O(N \log(N))$, which is the solution to the recurrence $T(N) = 2T(N/2) + O(N)$.

- $T(N)$: runtime of largestPerfectSubtree on a balanced tree with N nodes.
- $2T(N/2)$: 2 recursive calls; runtime of 2 each recursive call, which is on a subtree of roughly half the size because the tree is balanced.
- $O(N)$: The runtime complexity of the size method.

B. What is the asymptotic runtime complexity of `largestPerfectSubtree` on an **unbalanced** tree? Justify your answer by stating and explaining a recurrence relation, referencing the code.

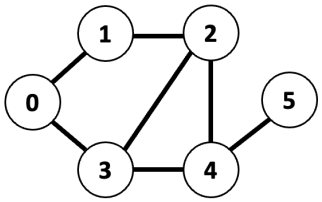
Answer: $O(N^2)$, which is the solution of the recurrence $T(N) = T(N-1) + O(N)$.

- $T(N)$: runtime of largestPerfectSubtree on an unbalanced with N nodes.
- $T(N-1)$: one recursive call on a subtree containing up to $N-1$ nodes. Fine but not required to also include an additional $T(1)$ term.
- $O(N)$: The runtime complexity of the size method.

```java
private boolean search(int start, int end) {
    if (start == end) {
        System.out.print(end + " ");
        return true;
    }
    if (!visited.contains(start)) {
        visited.add(start);
        for (int node: neighbors.get(start)) {
            if (search(node, end) == true) {
                System.out.print(start + " ");
                return true;
            }
        }
    }
    return false;
}
```

(a) search method

Node	Neighbors
0	[1, 3]
1	[0, 2]
2	[1, 3, 4]
3	[0, 2, 4]
4	[2, 3, 5]
5	[4]



(b) example graph with adjacency list representation

Figure 6

7. (8 points). Consider the `search` method defined in Figure 6a. The method works with an unweighted undirected graph where nodes have integer labels. The graph is represented by an adjacency list `Map<Integer, List<Integer>> neighbors`; assume this is an instance variable of the same class accessible to the `search` method. An example graph is drawn in Figure 6b along with its corresponding adjacency list representation. We also initialize an initially empty `HashSet<Integer> visited` as an instance variable accessible in `search` method.

A. What will be **printed** by running `search(5, 2)` on the graph shown in Figure 6b? Assume `visited` is initially empty and that looping over the elements of a given `List` loops in the order shown in Figure 6b. You do not need to explain your answer.

Answer: 2 4 5

B. What will be **printed** by running `search(1, 4)` on the graph shown in Figure 6b? Assume `visited` is initially empty and that looping over the elements of a given `List` loops in the order shown in Figure 6b. You do not need to explain your answer.

Answer: 4 2 3 0 1

C. Assume there are N nodes in the graph and each node is adjacent to at most 10 other nodes (that is, the length of each `List` in the the adjacency list representation is at most 10). Assume that `neighbors` is a `HashMap` and `visited` is a `HashSet`. What is the asymptotic runtime complexity of the `search` method as a function of N? Briefly explain your answer.

Answer: O(N). The first time a node is recursed on, it is added to the visited set; if the algorithm recurses on the same node again it immediately returns without searching its neighbors. So we search the neighbors from a given node just once. Since a node has at most 10 neighbors in this problem, we recurse on each node at most 10 times. There are N nodes, and a give search method has O(1) complexity apart from the recursive calls, so we get O(N) overall.

```
1  public int degrees(Map<String, List<String>> friends,
       String start) {
2      Map<String, Integer> degrees = new HashMap<>();
3      MISSING_TYPE toConsider = MISSING_CONSTRUCTOR;
4      toConsider.add(start); degrees.put(start, 0);
5      while (toConsider.size() > 0) {
6          String name = toConsider.remove();
7          for (String other : friends.get(name)) {
8              if (EXPR_1) {
9                  degrees.put(other, EXPR_2);
10                 toConsider.add(other);
11             }
12         }
13     }
14     int maxDegree = 0;
15     for (String name : degrees.keySet()) {
16         maxDegree = Math.max(maxDegree, EXPR_3);
17     }
18     return maxDegree;
19 }
```

(a) degrees method

Key	Value
"Al"	["Bo", "Xi"]
"Bo"	["Al", "Xi"]
"Kat"	["Jen"]
"Jen"	["Xi", "Kat"]
"Xi"	["Al", "Bo", "Jen"]

(b) Example friends map

Figure 7

8. (8 points). Consider the `degrees` method outlined in Figure 7a. The input to the method is a `Map<String, List<String>> friends` where the value associated with a given key is the list of their friends. An example is given in Figure 7b. Note that the friend relationship is symmetric. You can assume that, as in the example, every string that appears in any value list is also a key in the `Map`.

The method should return the *minimum* degree k such that all keys in the `Map` are within k degrees of separation from `start`. A key `other` is within k degrees of separation from `start` if `start` is a *friend of* `someone` is a *friend of* `someone else` ... is a *friend of* `other`, where the *friend of* relationship is invoked at most k times. *Hint:* Recall the Erdos Numbers problem from discussion.

In the example given in Figure 7b, if `start` is `"Al"` then the method should return 3, because `"Kat"`, the most separated from `"Al"`, is 3 degrees of separation away (`"Al"` is a friend of `"Xi"` is a friend of `"Jen"` is a friend of `"Kat"`).

What should the missing code be so that the method works correctly? You do not need to explain your answers. You may assume that everyone is within some degree of separation from `start`.

A. `MISSING_TYPE` and `MISSING_CONSTRUCTOR`:

Queue<String> and new LinkedList<>()

B. `EXPR_1`:

!degrees.containsKey(other)

C. `EXPR_2`:

degrees.get(name) + 1

D. `EXPR_3`:

degrees.get(name)

This page left intentionally blank for scratch work