CPSC-34000 Algorithms and Data Structures

Fall 2021

Final Exam

**Question-1 (10 points)** In a binary tree class, implement a Python method equal, that takes in another binary tree (other) as parameter and returns true if both binary trees are equal, otherwise the function returns False. What is the time complexity of your function?

class BinaryTree:

class \_Node:

def \_\_init\_\_(self, element, left = None, right = None):

self.\_left = left

self.\_right = right

self.\_element: int = element

def \_\_init\_\_(self):

self.\_root = None

self.\_size = 0

#this is my submission from PP8

def \_equal(self, root1, root2):

if root1 == None and root2 == None:

return True

if root1 != None and root2 != None:

return ((root1.\_element == root2.\_element), self.\_equal(root1.\_left, root2.\_left), self.\_equal(root1.\_right, root2.\_right))

return False

def equal(self, other):

return self.\_equal(self.\_root, other.\_root)

#time complexity:O(n) each node accessed once

**Question-2: (10 points)** Consider an array-based binary tree implementation, write a method find\_ansestors, that takes in an index i and returns all ancestors of node located at index i. What is the time complexity of your function?

class BinaryTree:

class ArrayBinaryTree:

def \_\_init\_\_(self):

self.\_heap = []

def find\_ancestors(self, i):

if (i != 0):

i = (i-1)/2

print(self.heap[i])

self.find\_ancestors(i)

else:

return 0

#time complexity O(n)per node + O(n-1/2) per ancestor

**Question-3 (5 points)** Show the binary tree that corresponds to the following array-based implementation of a binary tree. Does this represent a heap and/or binary search tree (i.e., exhibits heap property and/or binary search property)?

This is a heap binary tree. We can tell this immediately because of the rules of the binary search tree. Starting from 15 as the parent, we can see that the left child being smaller and the right being larger than the parent rule is broken, because nothing is larger than 15. When considering the type of heap tree, it is a max heap tree. So we can just follow that line of thinking to make the tree.

| 15 | 8 | 10 | 7 | 6 | 1 | 5 | 3 |
| --- | --- | --- | --- | --- | --- | --- | --- |

**15**

**8 10**

**7 6 1 5**

**3**

**Question-4 (5 points)** Given the sequence of numbers 1 through 15, what would be the order of insertion of these numbers into a binary search tree that would result in a binary search tree that exhibit worst and best search performance.

To optimize a BST the left node has to have as many nodes as the right node. The operations are optimized if the tree is balanced, making them usually O(logn). The worst case for a tree is based on the height of the tree. Therefore, if we made the left or right branch as long as possible, that would in theory be the worst case scenario.

results in worst-search performance

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

results in best-search performance

8 4 2 1 3 6 5 7 12 10 9 11 14 15

**8**

**4 12**

**2 6 10 14**

**1 3 5 7 9 11 13 15**

**Question-5 (10 points)** in binary search tree, write a function that takes in a root, p, and checks whether the tree rooted in p is a binary search tree or not. What is the time complexity of your function?

class BinaryTree:

class Node:

def \_\_init\_\_(self, element, left = None, right = None):

self.\_left = left

self.\_right = right

self.\_element: int = element

def \_\_init\_\_(self):

self.\_root = None

self.\_size = 0

return True

def \_is\_bst(self, p):

if (self != None):

if(\_is\_bst(self.\_left, p) == True):

return False

if (p != None and self.\_element <= p.\_element):

return False

p = self

return \_is\_bst(self.\_right, p)

return True

def is\_bst(self):

p = None

return \_is\_bst(self, p)

#time complexity: O(n)

**Question-6 (5 points)** Describe the difference between a tree and hash-based implementations of Map ADT, discuss their advantages, disadvantages and application scenarios where each should be used.

**Tree:** A binary tree map is assigned key and value data elements. The ordering, sorting and storing of the elements is natural. The elements are maintained in sorted order with no duplicates. This is commonly used to sort items stored in memory and to find elements stored in order.

**Advantages:** All elements are inserted in a sorted order.

**Disadvantages:** Time complexity for the map and methods of the map are usually O(nlog)

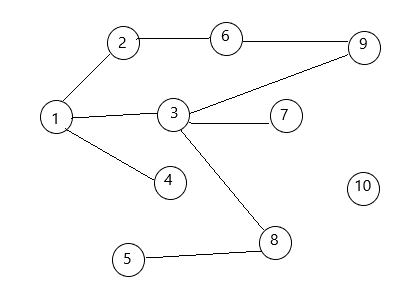
**Hash-Based Map ADT**: A hash map indices data structures. Almost any data type can be indexed, given a key, and then put into a ‘slot’ in the hash map. The key is immutable. These are commonly used when a programmer wants to assign a key with a value and store that data, as it is easily accessible using the hash system. It is a dictionary data type, meaning it is not ordered and can be changed.

**Advantages:** Most methods only take O(1) time.

**Disadvantages:** The order of the map will not remain constant. There is not a convenient way to find the size of the map because it can store abstract data types.

**Question-7 (15 points)** Study the below graph, and answer the following questions:

1. What is the corresponding adjacency matrix
2. Show order of nodes if a Depth-First-Search is invoked on node 1
3. Show order of visiting nodes, if a Breadth-First-Search is invoked at node 1



Adjacency matrix

|  | Node-1 | Node-2 | Node-3 | Node-4 | Node-5 | Node-6 | Node-7 | Node-8 | Node-9 | Node-10 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Node-1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| Node-2 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| Node-3 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 |
| Node-4 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Node-5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| Node-6 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| Node-7 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Node-8 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| Node-9 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| Node-10 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

**Depth-First-Search traversal – start at node 1**

Algorithm DFS(G,u): {We assume u has already been marked as visited}

Input: A graph G and a vertex u of G

Output: A collection of vertices reachable from u, with their discovery edges

for each outgoing edge e = (u,v) of u do

if vertex v has not been visited then

Mark vertex v as visited (via edge e).

Recursively call DFS(G,v).

| 1 | 2 | 6 | 9 | 3 | 7 | 8 | 5 | 4 | 10 |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 3 | 9 | 6 | 2 | 7 | 8 | 5 | 4 | 10 |  |  |  |  |  |  |
|  |  |  |  | 8 | 5 | 7 | 4 | 10 |  |  |  |  |  |  |  |
|  | 7 | 8 | 5 | 9 | 6 | 2 | 4 | 10 |  |  |  |  |  |  |  |
|  | ^ | 9 | 6 | 2 | 8 | 5 | 4 | 10 |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | 8 | 5 | 7 | 9 | 6 | 2 | 4 | 10 |  |  |  |  |  |  |  |
|  |  | 9 | 6 | 2 | 7 | 4 | 10 |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

1. 1-2-6-9-3-7-8-5-4-10
2. 1 - 3 - 9 - 6 - 2 - 7 - 8 - 5 - 4 - 10

8-5-7-4-10

7 -8-5-9-6-2-4-10

9-6-2-8-5-4-10

1. 8- 5 -7-9-6-2-4-10

9-6-2-7-4-10

| 1 | 4 | 3 | 8 | 5 | 7 | 9 | 6 | 2 | 10 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

1-4-3-8-5-7-9-6-2-10

**Breadth-First-Search traversal – start at node 1**

def BFS(g, s, discovered):

level = [s]

while len(level) > 0

next level = [ ]

for u in level:

for e in g.incident edges(u):

v = e.opposite(u)

if v not in discovered:

discovered[v] = e

next level.append(v)

level = next level

Code Fragment 14.8: Implementation of breadth-first search on a graph, starting at a designated vertex s.

1. 1-2-3-4-6-7-8-9-5-10
2. 1-3-4-2-8-7-9-6-5-10

| 1 | 2 | 3 | 4 | 6 | 7 | 8 | 9 | 5 | 10 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 3 | 4 | 2 | 8 | 7 | 9 | 6 | 5 | 10 |