GRIFFITH COLLEGE

EXAMINATION COVER PAGE

Learner Name:	**Oluwatobi Babatunde James**
Learner Number:	**3025513**
Programme:	**Programming and data Structures**
Stage:	**Second stage**
Module Title:	**HDC**
Module Code:	**HDC-PDS**
Date:	1/02/2021
Time:	5:45pm

Questions Attempted:

Questions	**Attempted – Yes/No**
1	Yes
2	Yes
3	Yes
4	Yes
5	
6	
7	
8	
9	
10	
11	
12	

Question 4

Describe what a hash function is and what properties make a good hash function. Give a simple example how hashing could help store mobile phone numbers.

A hash function is a function that takes a set of inputs of any arbitrary size and fits them into a table or other data structure that contains fixed-size elements.

Simple Example
Mobile phone number: 0897362145

We take each number and use the hash function operation to it to determine what slot to position the value. The hash function takes each value mod 10

Our hash function here is to take each value mod 10.

0 mod 10 = 2, so it goes in slot 0.
8 mod 10 = 8, so it goes in slot 8.
9 mod 10 = 9, so it goes in slot 9.
7 mod 10 = 7, so it goes in slot 7.
3 mod 10 = 3, so it goes in slot 3.
6 mod 10 = 6, so it goes in slot 6.
2 mod 10 = 2, so it goes in slot 2.
1 mod 10 = 1, so it goes in slot 1.
4 mod 10 = 4, so it goes in slot 4.
5 mod 10 = 5, so it goes in slot 5.

Explain what double hashing is and where and when it is used.

Double Hashing is a technique used to avoid clustering or collusion on hash tables. It uses a secondary hash function $h'(key)$ on the keys to determine the increments to avoid the clustering problem.

Describe how separate chaining can handle collisions and how it is implemented.

Separate chaining places all entries with the same hash index in the same location, instead finding new location, however each location in the separate chaining operation uses a bucket to hold multiple entries, thereby handling the issue of collisions.

It can be implemented using an array, arraylist or linkedlist.

Implementation
Let us consider a simple hash function as "**key mod 7**" and sequence of keys as 50, 700, 76, 85.

Initial Empty Table

0	
1	
2	
3	
4	
5	
6	

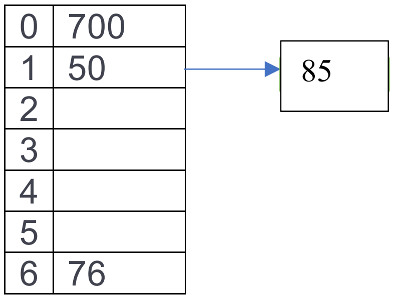
Insert 50

0	
1	50
2	
3	
4	
5	
6	

Insert 700 and 76

0	700
1	50
2	
3	
4	
5	
6	76

Insert 85: A collusion occurs therefore add a chain



0	700
1	50
2	
3	
4	
5	
6	76

In the implementation of the java.util.HashMap class in the Java API, the threshold 0.75 is used.

Is it important to keep the load factor under the threshold? If so explain why.

It's important because if the load factor exceeds the threshold, then you will need to increase the hash-table size and rehash all the entries in the map into a new larger hash table.

How can you reduce the likelihood of rehashing?

The likelihood of rehashing can be handled by doubling the hash table size.

Question 3

A

i) True

ii) False: Three parent nodes cannot point to one child

iii)False: This is simply because there is no parent node.

iv) False: A child node cannot point to a parent node in the same tree.

B

Outline the properties that define a red and black tree in detail. Include the Big O notation for searching within a red and black tree.

The properties for a red and black tree includes the following:

1. Every node is either red or black.
2. Every leaf (NULL) is black.
3. If a node is red, then both its children are black.
4. Every simple path from a node to a descendant leaf contains the same number of black nodes.

For searching within a red and black tree-The average and worst case is the same in searching within a red-black tree, which is O(logn).

Compare and contrast insertion into a Binary Search Tree and a Red and Black Tree.

Compare:

Both Binary search tree and a red and black tree insertion takes a running time of best case is O(log n).

They both maintain the binary search tree properties even during insertion.

Contrast:

A regular Binary Search tree is not self-balancing, meaning depending on the order of insertions you will get different time complexities. While a red-black tree is operate using self-balancing.

A binary search tree best case running time is O(log n) but the worst case is O(n) while for the red-black tree the best case running time is O(log n) and the worst case is O(log n).

Question 1

a)

Function	**Name**
$(3/2)^n$	**Exponential**
1	**Constant**
$(3/2)n$	**Linear**
$2n^3$	**Polynomial**
2n	**Linear**
$3n^2$	**Polynomial**
1000	**Constant**
3n	**Linear**

b)

64, $\log_8 n$, $\log_2 n$, 4n, $n \log_6 n$, $n \log_2 n$, $8n^2$, $6n^3$, 8^{2n}

c)

i) Only one operation is performed here

ii) Time complexity : O(n).

iii) A big integer is used here because of the possibility that the sum or any of the variables might excess the limit of the value it can use if it is an int variable.

d)

Selection sort: O(n^2), Bubble sort: O(n^2), insertion sort: O(n^2)

Insertion and bubble sort meant be faster because if the base case occurs, which is O(n).

Question 2

a)

0- [1, 6, 8],

1- [0, 4, 6, 9],

2- [4, 6],

3- [4, 5, 8],

4- [1, 2, 3, 5, 9],

5- [3, 4],

6- [0, 1, 2],

7- [8, 9],

8- [0, 3, 7],

9- [1, 4, 7]

b)

c). There are two common algorithms for traversing graphs. Name them both and explain their differences.

Names: Depth first search and Breadth first search

Differences:

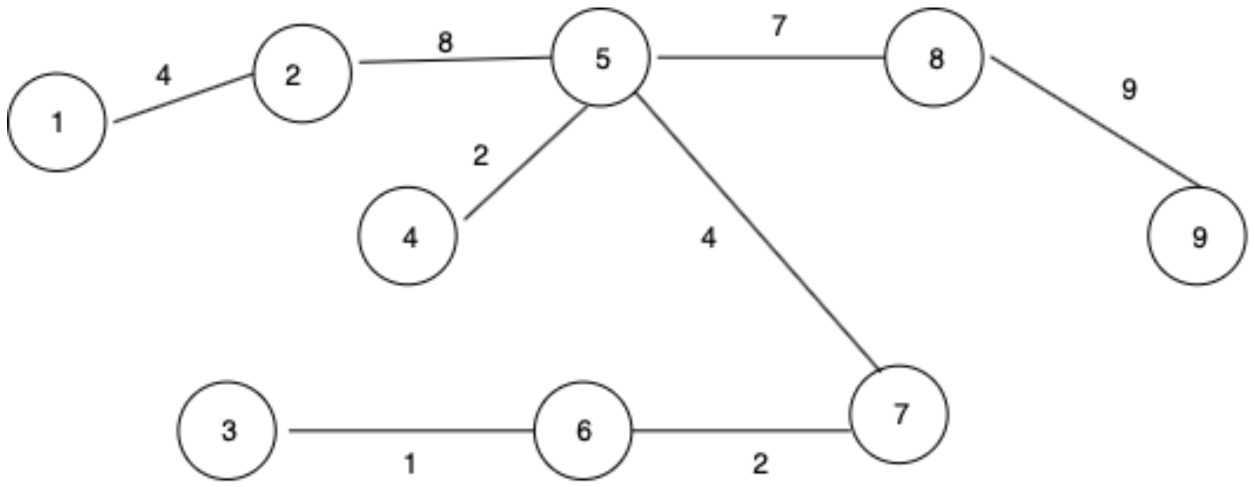
I) Depth first search uses stack data structure while the Breadth first search uses the queue data structure for to find the shortest path.

ii) The Time complexity of Depth First Search is also O(V + E) when Adjacency List is used and O(V^2) when Adjacency Matrix is used, where V stands for vertices and E stands for edges, while The Time complexity of Breadth First Search is O(V + E) when Adjacency List is used and O(V^2) when Adjacency Matrix is used, where V stands for vertices and E stands for edges.

iii) Depth First Search is better when dealing with solutions away from source while Breadth First Search is better for searching vertices which are closer to the given source.

d)

Minimum Spanning tree


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