Practical Exercise 8 – Graphs

Overall Objective

To design and implement applications using graphs.

Background

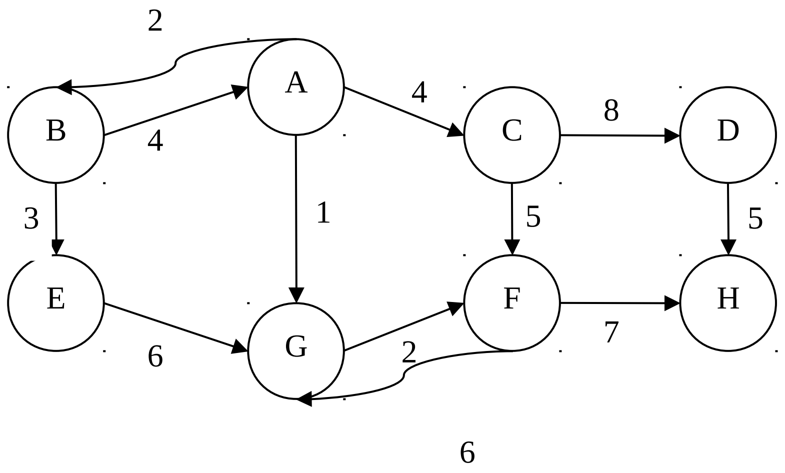
You will need to know:

1. basic Java programming knowledge
2. classes and interfaces
3. generics
4. graph concept

Description

Part 1: Discussion

1. The following graph are given:



1. Construct an ***adjacency matrix*** representation for the graph given above.

	A (in)	B	C	D	E	F	G	H
A (out)	0	2	A→C 4	0	0	0	1	0
B	4	0	0	0	3	0	0	0
C	0	0	0	8	0	5	0	0
D	0	0	0	0	0	0	0	5
E	6	0	0	0	0	0	0	0
F	0	0	0	0	0	0	6	7
G	0	0	0	0	0	2	0	0
H	0	0	0	0	0	0	0	0

2. Construct an ***adjacency list*** representation for the graph given above.

Note: number in brackets means the edge weight

A → B(2)→ C(4) → G(1) → null
(this means A is pointing to B, C and G. Don't misunderstood)

B → A(4) → E(3) → null
C → D(8) → F(5) → null
D → H(5) → null
E → G(6) → null
F → G(6) → H(7) → null
G → F(2) → null
H → null

3. Determine whether there is any ***cycle*** or ***loop*** in the graph given above.

<u>A cycle is a path consisting of at least 3 vertices that starts and ends with the same vertex.</u>

<u>A loop is a special case of a cycle in which a single arc begins and ends with the same vertex.</u>

Based on the definition, there is no cycle and no loop in the graph.

4. Start at A, trace a ***depth-first traversal*** through the above graph. You are required to show the ***stack*** contents as you work your way down the graph and then as you back out.

Depth-first search means go as deep as possible.

NOTE: In this case, we will follow the alphabetical order (in actual case it does not matter where you start and go).

Another NOTE: Mr. Wong said everyone's answer will be different, so no worries.

How to do?

Step 1: Push A into stack
stack: A

Step 2: Pop the stack and push adjacent of the popped item (which is A) into stack
stack: B, C, G (Why B, C, G? Because follow alphabetical order. Of course, you could reverse it if you don't like this order)

Repeat Step 2.

Step 3: stack: B, C, F
Step 4: stack: B, C, H
Step 5: stack: B, C,
Step 6: stack: B, D
Step 7: stack: B
Step 8: stack: E

Result: A, G, F, H, C, D, B, E (Take the last item of each stack)

5. Start at A, trace a ***breadth-first traversal*** through the above graph. You are required to show the ***queue*** contents as you work your way down the graph.

Breadth-first search means look at all adjacent vertex of each vertex before going to another vertex.

Assume we follow alphabetical order.

Step 1: Queue: A

Step 2: Dequeue the queue and enqueue the adjacent of the dequeued item (which is A in this case)
Queue: B, C, G

Step 3: Repeat step 2
Queue: C, G, E

Step 4: Queue: G, E, D, F
Step 5: Queue: E, D, F
Step 6: Queue: D, F
Step 7: Queue: F, H
Step 8: Queue: H

Result (take the first item of each queue): A, B, C, G, E, D, F, H

6. Start at A, develop a minimum spanning tree using Prim's algorithm for the graph.

Minumum spanning tree can be find by starting from the minimum weight in the graph. NOTE: - no cycle is allowed

7. Suppose the source vertex is A, develop a shortest path for the graph using Dijkstra's algorithm.

<u>ANSWER</u> **What is needed?** We need to find the shortest path from A to each node. (means A → B, A → C, A → D …) **How to do it?** Path 1: From A → B *List down possible paths:* Since the graph is directed (got arrow), the only path from A → B is A → B, and the cost is 2. Path 2: From A → C

Part 2: Programming Exercise

Graphs

Refer to lecture slide/main textbook (Ch30, Liang 9th edi.), define the following interface and classes for the graph:

1. Define `Graph<V>` interface (refer to slide 35).

2. Define `AbstractGraph<V>` abstract class that implements `Graph<V>` interface (refer to slide 35).
   - Define an inner class `Tree` `for depth-first search and breadth-first search`

3. Define `UnweightedGraph<V>` concrete class that extends `AbstractGraph<V>` abstract class (refer to slide 36).

4. Write the test program that builds a graph with 12 cities and their edges. Then, the program prints out all the edges of each city in the graph. Also, the program prints a DFS and a BFS for the graph

Hash Tables

1. Using only the modulo division method and linear probing, store the keys shown below in an array with 19 elements.

224562	137456	214562
140145	214576	162145
144467	199645	234534

2. Change your collision resolution in question 1 by using linked list method.

Answer for 1:

Why we use 19? We always use a prime number. And also depends on the data size.

In linear probing, in case of collision, we just put the element to the next available key.

Hashing Calculation	Hashed Table (Linear Probing)	
224562 % 19 = 1	0	10 - 137456
137456 % 19 = 10	1 - 224562	11 - 144467
214562 % 19 = 14	2 - 140145	12 - 199645
140145 % 19 = 1	3	13
214576 % 19 = 9	4	14 - 214562
162145 % 19 = 18	5	15
144467 % 19 = 10	6	16
199645 % 19 = 12	7	17 - 234534
234534 % 19 = 17	8	18 - 162145
	9 - 214576	

Answer for 2:

In chaining resolution, in case of collusion, we just append the element at the list.

Hashing Calculation	Hashed Table (Chaining resolution)	
224562 % 19 = 1	0	10 – 137456 → 144467
137456 % 19 = 10	1 – 224562 → 140145	11 -
214562 % 19 = 14	2 -	12 - 199645
140145 % 19 = 1	3	13
214576 % 19 = 9	4	14 - 214562
162145 % 19 = 18	5	15
144467 % 19 = 10	6	16
199645 % 19 = 12	7	17 - 234534
234534 % 19 = 17	8	18 - 162145
	9 - 214576	