University of Illinois at Urbana-Champaign
Department of Computer Science

Third Examination

CS 225 Data Structures and Software Principles
Sample Exam 2
75 minutes permitted

Print your name, netID, and lab section day/time neatly in the space provided below; print your name at the upper right corner of every page.

Name:
NetID:
Lab Section (Day/Time):

- This is a **closed book** and **closed notes** exam. In addition, you are not allowed to use any electronic aides of any kind.
- Do all 5 problems in this booklet. Read each question very carefully.
- You should have 7 sheets total (the cover sheet, plus numbered pages 1-12). The last sheet is scratch paper; you may detach it while taking the exam, but must turn it in with the exam when you leave.
- Unless otherwise stated in a problem, assume the best possible design of a particular implementation is being used.
- Unless the problem specifically says otherwise, (1) assume the code compiles, and thus any compiler error is an exam typo (though hopefully there are not any typos), and (2) assume you are NOT allowed to write any helper methods to help solve the problem, nor are you allowed to use additional arrays, lists, or other collection data structures unless we have said you can.

Problem	Points	Score	Grader
1	12		
2	30		
3	18		
4	15		
5	15		
Total	90		

1. **[Short Answer – 12 points (4 points each)].**

   (a) What two data structures are used to implement Kruskal's Algorithm?

   (b) Using big-$\mathcal{O}$ notation, indicate the running time of breadth-first search on a graph with `V` vertices and *no* edges.

   (c) Draw a directed graph that does not have a legal topological sort.

2. **[Algorithms – 30 points (6 points each)].**

   (a) Under what conditions will Dijkstra's Algorithm be more efficiently implemented using a heap than with a table, if your graph is implemented using an adjacency list? Justify your answer.

   (b) Justify the correctness of the depth-first-search version of our topological sort algorith. You don't need to justify the correctness of depth-first search – just explain why our use of it to perform topological sort *must* assign numbers to the vertices in such a way that the vertices are numbered in topological-sort order.

(c) In Prim's Algorithm, we said you choose an edge at each step, from among all edges that go from set `S` to set `N`. What do these two sets represent, and why would it be a problem to pick an edge that goes between two vertices in `S`?

(d) Under what conditions would running breadth-first search on an undirected graph, result in a breadth-first spanning forest of more than one tree, rather than a breadth-first spanning tree? That is, what kinds of undirected graphs would result in that kind of answer? Justify your answer.

(e) For the given graph, run Dijkstra's algorithm, indicating in the table below the distances at each vertex at the end of each step (d_v), and whether or not the vertex has been marked known yet at the end of each step (k_v). The initialization has already been done for you.

	A	B	C	D	E	F	G
A	0	8	3	0	0	0	0
B	0	0	0	0	0	16	18
C	0	3	0	0	0	0	20
D	0	0	0	0	0	0	19
E	5	0	12	2	0	0	0
F	0	0	0	0	0	0	0
G	0	0	0	0	0	10	0

V	d_v	k_v	d_v	k_v	d_v	k_v	d_v	k_v	d_v	k_v	d_v	k_v	d_v	k_v	d_v	k_v
A	∞	0														
B	∞	0														
C	∞	0														
D	∞	0														
E	0	0														
F	∞	0														
G	∞	0														
-	Start		Step 1		Step 2		Step 3		Step 4		Step 5		Step 6		Step 7	

3. **[Analysis – 18 points (9 points each)].**

   (a) What is the order of growth of the running time of finding the complement of a graph (the graph with the exact same vertices and the exact opposite set of edges), if you have a graph implemented with an adjaceny matrix? Express your answer in big-$\mathcal{O}$ notation and justify your answer. Assume your adjaceny matrix implementation *does* have a one-dimensional array of vertex information, and that your adjacency matrix itself does not have any "edge info" records, but rather, merely tells you if an edge exists or not.

(b) What is the order of growth of the running time of depth-first search on a graph with `V` vertices and `E` edges? Express your answer in big-$\mathcal{O}$ notation and justify your answer.

4. **[Breadth-First Spanning Tree – 15 points].**

   You are given the following class:

```
class TreeNode {
public:
   int vertexNumber;
   list<TreeNode*> subtrees; // initially an empty list; the function
                             //   push_back(item) will add item to
                             //   end of list
};
```

   In addition, you are given an adjacency matrix implementation of an unweighted, undirected, connected graph – such a graph, when you run breadth-first search on it, would produce a single spanning tree, rather than a spanning forest, no matter which vertex you start at. In this adjacency matrix implementation, the graph has vertices labelled with indices `0` through `n-1`, and you are given the value `n` and a two-dimensional array with `n` rows and `n` columns, both indexed from `0` through `n-1`. (In C++, a two-dimensional array is accessed via expressions such as `arr[i][j]` where `arr` is of type `int**`.) You want to write a method that takes those two values – the array and the `n` – as parameters, and returns a pointer to the root of the breadth-first-spanning-tree of the graph. You are allowed to use as many `Queues` as you like, as well as being allowed to create other one-dimensional arrays as well.

```
TreeNode* BreadthFirstSpanningTree(int** graph, int n) {
   // your code goes here
```

(Breadth-First Spanning Tree, continued)

5. **[Converting to undirected - 15 points].**

Suppose you have a directed weighted graph of n vertices, where the vertex numbers are 1 through n, and the graph implementation is an adjacency list. The adjacency list is represented with an Array, indexed from 1 to n, of pointers to the following type:

```
class EdgeNode {
public:
  int index;   //index of target vertex
  int weight;  //weight of edge
  EdgeNode* next;   // ptr to next edge
};
```

We want to convert this graph to an undirected weighted graph, by adding for each existent edge from u to v, the edge in the opposite direction, that is from v to u, with the same weight. That is, if the directed version had:

```
               5
 U ---------------------------> V
```

the undirected vertion will have:

```
               5
 U ---------------------------> V
               5
 V ---------------------------> U
```

You can assume that, initially, if there is an edge `<i, j>` in the graph, there is not an edge `<j, i>` of any weight. Write the function ConvertToUndirected, which receives as a parameter an Array of EdgeNode pointers – i.e. our adjacency list, as described above – by reference. The function will perform the conversion discussed above. Do not assume any edge list is specifically sorted in any way.

```
void ConvertToUndirected(Array<EdgeNode*>& graph) {
   // your code goes here
```

(Converting to undirected, continued)

.

(scratch paper)