Name: _____

Data Structures
Spring 2018
Final Exam

This is a 3-hour, closed-notes exam. **Please put away all notes, phones, laptops, etc.** There are 12 pages and 6 problems. Good luck!

Problem	**Score**
1	/18
2	/10
3	/15
4	/20
5	/16
6	/16
Total	/100

You get 5 points for writing your name at the top of this page.

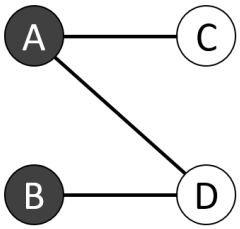
You do not need to show any work beyond your final answer, but you are welcome to do so if you feel it will help me understand your thought process.

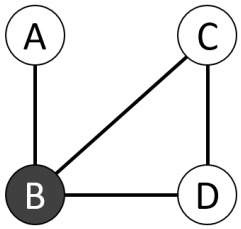
1. Depth-First Search.

(a) Consider the partial pseudocode for Depth-First Search below. Fill in each box to complete the pseudocode.

```
DFS(Graph G, starting vertex s)

    let list be a [          ] of unvisited vertices
                  (name of data structure)
    add s to list


    while list isn't empty

        vertex v = [          ]
                   (some operation on list)

        if v hasn't been visited

            mark v visited


            for each neighbor u of v (in alphabetical order)

                if [          ]
                   (condition on u and/or v)

                    set [     ]'s parent to [     ]

                    [          ]
                     (some operation on list)
```

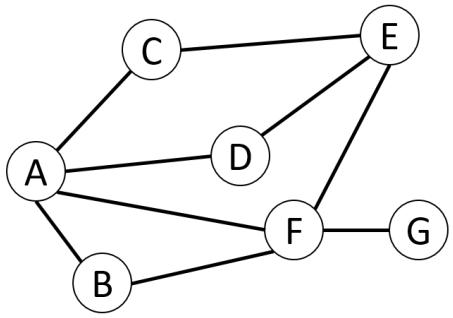
(b) A graph is said to be *bipartite* if its vertices can be split into two sets such that all edges are between pairs of vertices in opposite sets. Another way of thinking about this is that the vertices in a bipartite graph can be colored using two colors so that every edge connects two different-colored vertices. For example, the graph on the left below is bipartite, whereas the graph on the right below is not.





Augment the DFS pseudocode in part (a) so that your DFS correctly determines whether a graph is bipartite (you can also modify the vertex class if you need to). Clearly indicate what steps you are adding to DFS and where you are adding them.

(c) The following graph is *not* bipartite. Draw the DFS tree that results when your augmented DFS is run on this graph, starting at vertex A. You should indicate the point at which your augmented DFS identifies that the graph is not bipartite.



2. Binary Search Trees. Suppose you have an unbalanced Binary Search Tree that includes the following classes:

```
public class BST {
    Node root;

    ...
}
```

```
public class Node {

    String data;
    Node left;
    Node right;

    ...

}
```

Write a method inside the BST class that computes and returns the height of the BST. Your method should be recursive.

3. Union-Find.
(a) Draw a picture of the Union-Find data structure that results from the following sequence of operations (you do not need to show intermediate steps; if you do show intermediate steps your final answer should be clearly labeled). Use union by size. If there is a tie, break ties so that if `i < j`, `find(i)` gives the same result before and after calling `union(i,j)`.

```
makeset(1)
makeset(2)
...
makeset(7)
union(4,5)
union(2,6)
union(1,5)
union(7,3)
union(7,2)
```

(b) What are the following runtimes? (You do not need to justify your answers.)

(i) Worst-case runtime of union

(ii) Worst-case runtime of find

(iii) Amortized runtime of any sequence of m operations, including $n \leq m$ makesets

(c) Suppose you were dealing with an application in which unions were very common. How could you modify the Union-Find data structure so that the union operation is more efficient than the find operation (you can assume that you are unioning the list headers; do not include the cost of finding the headers in your union cost)? You do not need to write any code, just explain how your union and find operations would work. You are welcome to write code or pseudocode or draw pictures if you feel it will help make your explanation clearer.

What are the worst-case runtimes of your union and find (again, do not include the cost of finding the headers in your union cost)?

4. Comparing data structures. I'm writing a website for a cool new game, and I need a data structure to keep track of all the players who are currently logged in. For each player, I keep track of `U`, their username, and `T`, the time at which they logged in. Let `n` be the number of players who are logged in. My data structure needs to support the following operations:

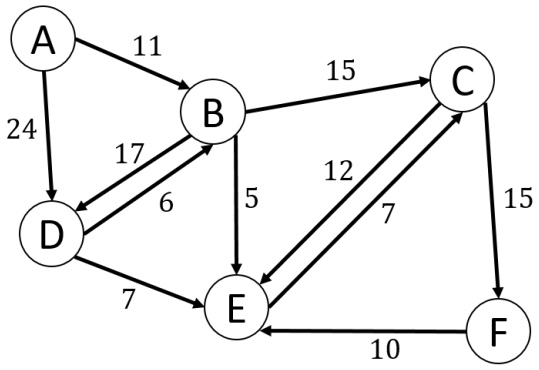
- `login(U,T)`: insert a player with username `U` and timestamp `T`
- `logout(U)`: remove the player with username `U`, returning that player's timestamp `T`
- `oldest()`: return (without removing) the username of the player with the oldest timestamp
- `multilogout(s)`: remove the `s` players who have been logged in the longest (i.e., the `s` players with smallest timestamps), returning an array of their usernames (if `s < n`, remove and return `n` players) (for some reason, my game keeps things interesting by placing a limit on the amount of time a player can stay logged in at one time)

(a) For each implementation in the table below, write the worst case runtime for each of the operations (do not include the cost of resizing an array). **Your runtimes should be in terms of `n` and `s`.** Feel free to write numbered footnotes to explain your reasoning (you can use the back if you run out of space).

	`login(U,T)`	`logout(U)`	`oldest()`	`multilogout(s)`
Array sorted by `U`				
Array sorted by `T`				
Red-Black tree ordered by `U`				
Red-Black tree ordered by `T`				

(b) Why would a hash table be a poor implementation choice for this application? Explain your reasoning in a few sentences.

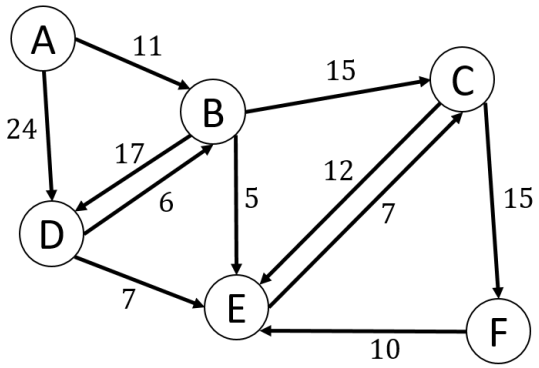
5. Shortest paths. Here is a graph:



(a) Draw the adjacency matrix representation of this graph.

(b) Fill in the table with the results of running Dijkstra's algorithm on this graph to find the shortest path from vertex `A` to every other vertex. Show the priority queue after each step of the algorithm (the priority queue at the start is shown).

Graph repeated for your reference:



	A	B	C	D	E	F
path length						
parent						

`(A, 0)`

`(B, ∞)`

`(C, ∞)`

`(D, ∞)`

`(E, ∞)`

`(F, ∞)`

6. Disjoint arrays. Every semester, the registrar's office has to schedule final exams so that no student has two exams scheduled at the same time. Suppose that for each course, we have an array that stores the names of all students enrolled in that course. We want to take two such arrays and determine whether the courses have any students in common. We can do this naively by writing a pair of nested for loops that, for each student in the first course, checks if the student's name matches any of the students in the second course:

```
for(int i = 0; i < course1.length; i++) {
    for(int j = 0; j < course2.length; j++) {
        if(course1[i].equals(course2[j])) return true;
    }
}
return false;
```

(a) Let n be the number of students in course 1, and let m be the number of students in course 2. In terms of n and m, how long does the above method take to determine if the courses have any overlap?

(b) We want to do better. Write a method to determine if two arrays have any elements in common that is *guaranteed* to have a better runtime than your result in part (a). You can use any of the data structures we've discussed this semester, and you can assume you have whatever implementation of this data structure you want (i.e., just call operations on your data structure, don't write the code for the data structure itself).

(c) In terms of n and m, how long does your method in part (b) take? In your answer, it should be clear what data structure(s) you used in your method, what implementation(s) you chose for those data structure(s), and the runtimes of the relevant operations.