Problem 5 (8 pts) Graphs

A. (4 pts) Minimal spanning trees
   
a. (2 pts) Run Kruskal’s minimum spanning tree algorithm for the above graph. List the edges (e.g., AF) in the minimum spanning tree created, in the order they are added to the tree. DF, AC, CD, BE, CE

b. (2 pts) Run Prim’s minimum spanning tree algorithm for the above graph starting from vertex A. List the edges of the minimum spanning tree created, in the order they are added to the tree.
   AC, CD, DF, CE, EB

B. (4 pts) Single source shortest paths

Run Dijkstra’s shortest path algorithm on the previous graph using B as the start vertex. Show the entries in the following table after adding the first 3 nodes (B & 2 other nodes) to the set S of processed nodes (as defined by Dijkstra’s algorithm). Keep in mind that after adding a node to the set S you must adjust the cost/predecessor of the appropriate successor nodes.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>LowestCost</td>
<td>3</td>
<td>0</td>
<td>∞</td>
<td>2</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Predecessor</td>
<td>D</td>
<td>none</td>
<td>none</td>
<td>B</td>
<td>A</td>
<td>F</td>
</tr>
<tr>
<td>Order Added</td>
<td>3</td>
<td>1</td>
<td></td>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Problem 7 (6 pts) Multithreading and Synchronization

C. (3 pts) Multithreading
   a. Using multiple threads always improves program performance                F
   b. Using multiple threads can simplify the structure of a program            T
   c. Using multiple threads may cause intermittent bugs                      T

D. (3 pts) Synchronization
   Consider the following code if several MaybeRace objects are created and multiple threads execute their increment methods in parallel:

   ```java
   public class MaybeRace {
       static int x = 0;
       Object y = new Object();
       static Object z = new Object();
       public void inc1( ) { synchronized(y) { x = x + 1; } }
       public void inc2( ) { synchronized(z) { x = x + 1; } }
       public synchronized void inc3( ) { x = x + 1; }
   }
   
   a. Using inc1( ) is thread safe                                           F
   b. Using inc2( ) is thread safe                                           T
   c. Using inc3( ) is thread safe                                           F
E. (6 pts) We have a DenseBag class that is defined, in part as:

```java
public class DenseBag<E> {
    private Map<E, Integer> denseBagMap = new HashMap<E, Integer>();
    private int size;
    public boolean add(E o) {
        Integer value = (Integer) denseBagMap.get(o);
        if (value != null)
            denseBagMap.put(o, value + 1);
        else
            denseBagMap.put(o, 1);
        size++;
        return true;
    }
    ...
}
```

a) What can go wrong if multiple threads update a DenseBag at the same time? Give specific outcomes.

Answer: Since we are updating a `HashMap` from multiple threads without synchronization, almost anything. The `HashMap` data structure can be corrupted in various ways (such as introducing a cycle in the list of entries in a bucket). We also could find that the increment to `size` isn’t atomic, and that the two threads could simultaneously call `get` with the same value, see null, and put the value 1 into the map, with the result that the map contains 1 for that value rather than 2.

b) Mike proposes that we make the class thread safe by changing the Map to be a `ConcurrentHashMap`. Why isn’t this a good solution. Give specific examples of what can go wrong.

Answer: The updates to `size` still aren’t atomic, and neither is the sequence of calling `get` followed by `put` on the map. For example, after two threads call `add("x")`, the overall size field might contain either 1 or 2, and `denseBagMap.get("x")` might be either 1 or 2.

c) Propose a better solution to making DenseBag thread safe.

Answer: Just make the `add` method (and all other DenseBag methods) synchronized. We can leave the Map as a normal `HashMap`