CMPT225, Spring 2021

Final Exam - Solutions

Name_____

SFU ID: |__|__|__|__|__|__|__|__|__|

Problem 1	
Problem 2	
Problem 3	
Problem 4	
TOTAL	

Instructions:

1. You should write your solutions directly in this word file, and submit it to Coursys. Submitting a pdf is also ok.
2. Submit your solutions to Coursys before April 20, 23:59. No late submissions, no exceptions
3. Write your name and SFU ID on the top of this page.
4. This is an open book exam. You may use textbooks, calculators, wiki, stack overflow, geeksforgeeks, etc. If you do, specify the references in your solutions.
5. Discussions with other students are not allowed. Posting questions online asking for solutions is not allowed.
6. The exam consists of four (4) problems. Each problem is worth 25 points.
7. Write your answers in the provided space.
8. You may use all classes in standard Java, and everything we have learned.
9. Explain all your answers.
10. **Really, explain all your answers.**

Good luck!

Problem 1 [25 points]

A. (15 points) In this question you need to design a data structure that supports PriorityQueue with deletions. Specifically, you need to write the class PriorityQueueWithDeletions. As a motivation you may think of a priority queue for a printer that allows adding a document, removing a document in some order (e.g. shorter documents are printed first), or a user can cancel the job. Specifically the class needs to support the following operations:
The running time of each operation must be O(log(size of the queue)).

```
public class PriorityQueueWithDeletions<T extends Comparable<T>> {

        public PriorityQueueWithDeletions() - creates an empty priority queue

        public Ticket add(T item) - adds a new element to the queue.
        Returns a ticket that can be used to remove your item from the queue.

        public T removeHighestPriority() - removes the element with the
                    highest priority from the priority queue and returns it.

        public T removeByTicket(Ticket t) - removes an element by ticket,
                                            and returns the removed element.

        public int size() - returns the number of elements in the queue

        public boolean isEmpty() - checks if the queue is empty
}
```

The running time of each operation must be O(log(n)), where n is the size of the priority queue.

Explain your answer in detail. Define the class Ticket and explain how you use it. Make sure Ticket does not allow the user to modify the queue adversarially.

IDEA: Use AVL tree for the Priority Queue.When adding a new element into the AVL tree, the Ticket will hold the pointer to the node with the inserted element. The removeByTicket() will simply remove the node from the tree using the O(log(n)) algorithm. removeHighestPriority() will find the minimum in the tree (by going all the way to the left), and removing this node.

```
public interface Ticket{} // will be held by the user so that no access
to the data is allowed.

public class PriorityQueueWithDeletions<T extends  Comparable<T>> {

    // node of the AVLNode<T>
    private class PQNode implements Ticket {
          AVLNode<T> avlNode;
    }

    We assume we have the standard methods of AVL: insert(item), remove(node),
    AVLTree<T> tree;

    public PriorityQueueWithDeletions() {
          tree = new AVLTree<T>();
    }

    public Ticket add(T item) {
    // add item to the AVL tree and return the node that holds the new item
    (remember the node implements Ticket)
          PQNode pqNode = new PQNode();
          pqNode.avlNode = tree.insert(item);
          return pqNode;
    }

    public T removeHighestPriority() {
          // goes all the way to the left, and removes the minimal node
          AVLNode<T> current = tree.getRoot();
          while (current.getLeftChild() != null)
                current = current.getLeftChld();
          return tree.remove(current);
    }

    public T removeByTicket(Ticket t)  {
          // remove the node in the ticket and return it
          PQNode pqNode = (PQNode).t;
          T ret = pqNode.avlNode.getData();
          tree.remove(pqNode.avlNode);
          return ret;

    }

    public int size() {
          return tree.size();
    }

    public boolean isEmpty() {
          return tree.isEmpty();
    }

}
```

Alternative idea: Priority Queue will be a minHeap with min defined using compareTo() of T. When adding a new element into the Heap, the Ticket will hold the pointer to the node with the inserted element.
The removeByTicket() method will remove the node from the heap. This is done as follows:

- remove the node in the ticket, and replace it with the last node in the heap.
- Then it will check if the replaced node satisfies the minHeap condition (it is smaller than the children, and larger than the parent), and apply siftUp or siftDown on it if needed,

B. (10 pts) Write a data structure that supports all operations of a stack, and in addition supports getMax. Specifically, you need to write the class StackWithMax.
The running time of each operation must be O(1).

```
public class StackWithMax<T extends Comparable<T>> {

        public StackWithMax() - creates an empty stack

        public void push(T item) - adds a new element to the stack.

        public T pop() - removes the element from top and returns it.

        public T getMax() - returns the maximum in the stack (without
        modifying the stack).

        public int size() - returns the number of elements in the stack

        public boolean isEmpty() - checks if the stack is empty

}
```

The running time of each operation must be O(1).

Explain your answer in detail.

IDEA: StackWithMax will be implemented using a LinkedList where we add/remove nodes to/from front.

Each node in the LinkedList will have (1) data and (2) max value in the stack from this node to the end.

When adding a new element, we compare the new element to max so far (in the first node), and set the maximum of them to be the max in the new node.

getMax() function simply returns max in the first node in the linked list.

```
public class StackWithMax<T sextends Comparable<T>> {

	private class TplusMax {
		T data;
		T max;
		public TplusMax(T data, T max) {
			this.data = data;
			this.max = max;
		}
	}

	LinkedList<TplusMax> list;

	public StackWithMax() {
		list = new LinkedList<TplusMax>();
	}

	public void push(T item) {
		T newMax = null;
		if (list.isEmpty());
			newMax= item;
		else
			maxSoFar = list.getFirst().max;
			newMax = item.compareTo(maxSoFar) > 0 ? item : maxSoFar;
		list.addFirst(new TplusMax(item, newMax));
	}

	public T pop() {
		return list.removeFirst().data;
	}

	public T getMax() {
		return list.getFirst().max;
	}

	public int size() {
		return list.size();
	}

	public boolean isEmpty() {
		return  list.isEmpty();
	}

}
```

Problem 2 [25 points]

In this problem use the following definition of Binary Tree.

```
public class BTNode<T> {
    private T data;
    private BTNode<T> leftChild;
    private BTNode<T> rightChild;
    private BTNode<T> parent;
}
public class BinaryTree<T> {
    private BTNode<T> root;
}
```

A. (7 pts) Write a method that gets a binary search tree and checks if it is an AVL tree. You may assume that the given tree is a binary search tree.
Explain your answer.
IDEA: Define the method height(BTNode node) that computes the height of the node. Then, for each node in the tree compare the heights of the left and the right children, and check if they satisfy the AVL property.

```
public boolean isAVLTree(BinaryTree<T> bst) {
        if (bst == null || bst.getRoot() == null)
                return true;

        // tree traversal
        Stack<BTNode<T>> stack = new Stack<BTNode<T>>();
        stack.add(bst.getRoot());
        while (!stack.isEmpty()) {
                BTNode<T> current = stack.pop();
                int hLeft = height(current.getLeftChild());
                int hRight = height(current.getRightChild());
                if (hLeft - hRight > 1 || hLeft - hRight < -1)
                        return false;
                if (current.getRightChild() != null)
                        stack.push(current.getRightChild());
                if (current.getLeftChild() != null)
                        stack.push(current.getLeftChild());
        }
        return true;
}

public int height(BTNode<T> node) {
        if (node == null)
                return -1;
        else
                return 1+ Math.max(height(node.getLeftChild()),
                                   height(node.getRightChild()));
}
```

B. (18 pts) For each of the following statements decide if they are true or false. Explain your answers. A correct T/F answer without a correct explanation will give 1 point.

1. (2 pts) Insertion into an AVL tree always increases the number of leaves.
   FALSE: For example, we start with only the root, that's 1 leaf. If we add one more node, we will get two nodes, but only one of them will be a leaf, so the number of leaves did not increase.

2. (4 pts) An AVL tree of height 3 has at least 7 nodes and at most 15 nodes
   TRUE:
   For upper bound: any binary tree of height 3 has at most 1+2+4+8=15 nodes.
   For lower bound:
   - AVL tree of height 0 has 1 node.
   - Any AVL tree of height 1 has at least 2 nodes.
   - Any AVL tree of height 2 has at least 1+1+2=4 nodes.
   - Any AVL tree of height 3 has at least 1+2+4=7 nodes.

3. (6 pts) If inserting a node into an AVL tree requires rebalancing, then the height of the entire tree does not change.
   TRUE: we saw that if we need to rebalance a node after insertion, then the height of this node does not change.
   If this rotated node is the root, then this is also the height of the tree.
   If not, then after rebalancing this node, there is no need to rebalance its ancestors since all heights stay the same.

4. (6 pts) When removing a node from an AVL tree with n nodes, then at most 3 vertices need to be rebalanced.
   FALSE: If you take a minimal AVL tree of height h = c*log(n), and remove from the shallowest branch, you will need to perform rebalancing $\Omega(\log(n))$ times.

Problem 3 [25 points]

A. (20 pts) Write a method that gets a 2d grid given as an array of ints with 0s and 1s. The goal is to find a path from the top left corner of the matrix to the bottom right using the minimal number of steps when you are only allowed to step on the 1s of the array. Write an algorithm that solves this problem and prints the path. In the end of the algorithm the input needs to be in the same state as it was in the beginning.

For example, for the input below, the path from grid[0][0] to grid[5][4] is marked in red, and consists of 10 ones. Note that you need to find the shortest path. If there are several shortest paths, your algorithm needs to print only one of them.

```
int[][] board = {
       {1,1,1,1,1},
       {1,0,0,0,1},
       {1,0,1,1,1},
       {1,1,1,0,0},
       {1,1,1,1,1},
       {0,0,1,0,1}
};
```

Before writing the algorithm, explain your answer.

```
public static void printPath(int[][] grid) {
```

IDEA: given an nxm array create a graph whose vertices are all points (i,j), and there is an edge between two points if and only if they are adjacent in the grid. That the number of vertices is n*m, and each vertex has degree at most 4. The corners have degree 2, and the vertices on the edges have degree 3.

Run BFS on that graph from (0,0) until reaching the point (n,m). When (n,m) is reached this will give the shortest path as per BFS algorithm, using the parent of each node. (Remember, each node added to the queue remembers its parent.)

```
}
```

B. (5 pts) What is the running time of your algorithm when the input is an nxn array? Explain your answer.

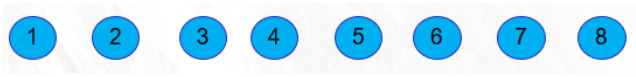
ANSWER: the running time is O(n*m) because the graph has n*m vertices, and at most 2nm edges, and the running time of BFS is O(|V|+|E|) = O(n*m).

Problem 4 [25 points]

A. (8 pts) Draw the representation of union-find data structure after the following sequence of operations. Draw some intermediate steps.

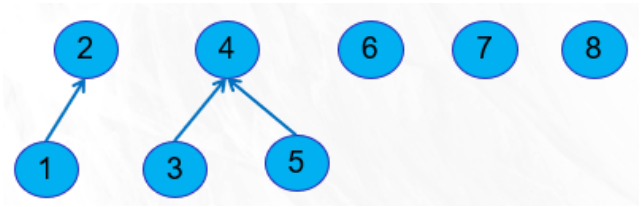
- For i=1...8

  makeset(i)

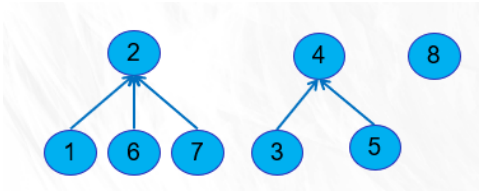


- union(1,2)

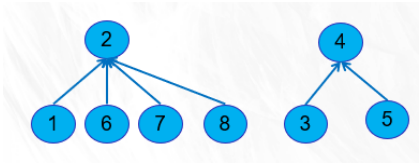
- union(3,4)
- union(3,5)



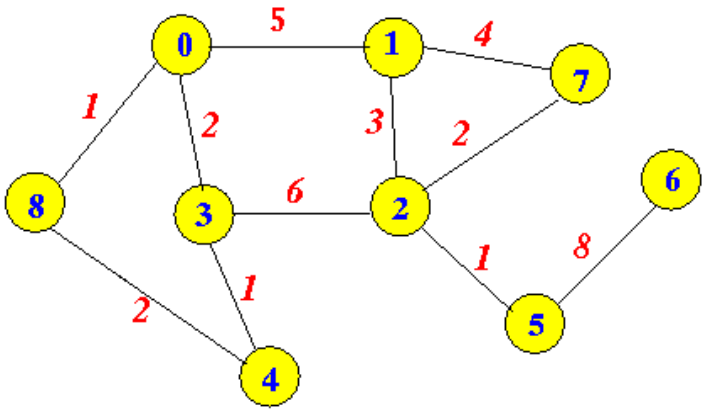
- union(1,6)
- union(1,7)



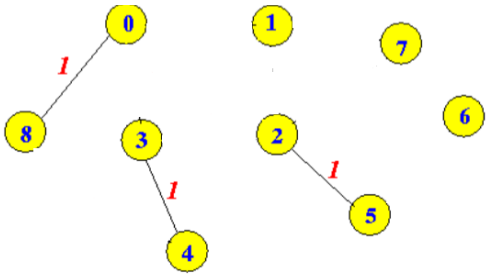
- union(8,1)



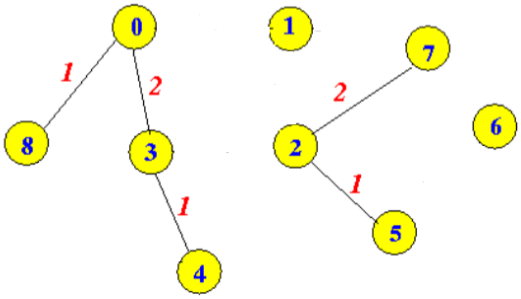
B. (7 pts) Show the execution of Kruskal's algorithm on the following graph.



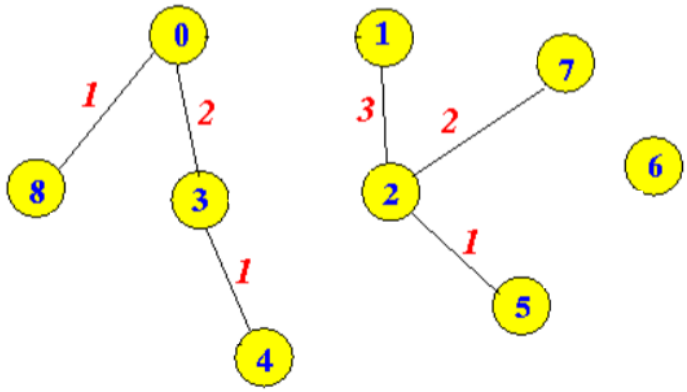
First we add the edges of cost 1



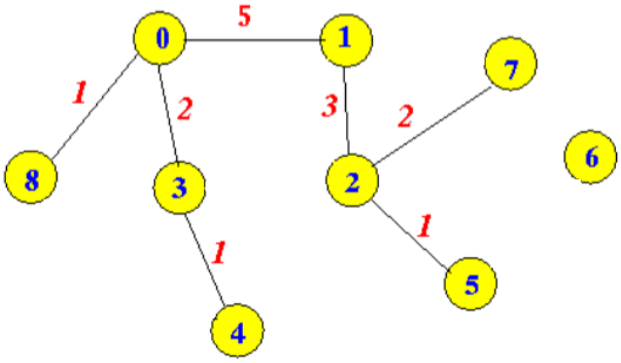
Then we add the edges (0.3) and (2,7) of cost 2.
[We could add (4,8) instead of (0,2)]



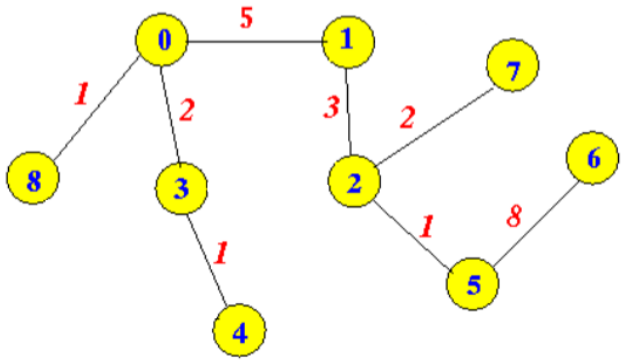
Next we must add the edge (1,2) of cost 3.



Then we add the edge (0,1) of cost 5 because it is the cheapest node between connected components (cheaper than 6 and 8).



Finally we add the edge (5,6) of cost 8 to connect the two connected components.



C. (10 pts) Write a function that gets an array A of length n of int, and an integer k, such that $0 \leq k \leq 2n/\log_2(n)$. The function needs to permute the elements of A so that the first k elements of A are the smallest numbers sorted in the non-decreasing order. The **running time** must be **O(n)** and **extra space used** should be **O(1)**.

For example, on input A =[3,1,8,2,6,11,4,12,5,7,10,9] and k=6 the resulting array A needs to have [**1,2,3,4,5,6**] as its first six elements, and the rest can be any permutation of the remaining elements. For example [**1,2,3,4,5,6**,12,7,9,8,10,11] The order of the elements after in the last n-k positions is not important.

Explain your idea before writing the code.

IDEA: This is very similar to problem 4D in the midterm. We create a minHeap from A . The minHeap is using the array structure. Then we remove k minimal elements from A, and add them to the end of A. Note that when we remove an element from the heap, it leaves us a free space in the end, and so we canada the minimum at that free space. In the end we need to reverse the array so that the minimal element appear in the beginning of the array

The algorithm is the following:

1. Let n = A.length
2. Given the array A, buildMinHeap(A)

   // put the k minimal elements in the end of the array
3. For i=0...k-1:
   a. int min =removeMinFromHeap(A)
   b. A[n-1-i] = min

   // reverse the array
4. For i=0...n/2
   a. int tmp = A[i]
   b. A[i] = A[n-1-i]
   c. A[n-1-i] = tmp

For the running time:

- Steps 1+2 run in O(n) time, using the buildHeap algorithm we saw in class.
- Each iteration in step 3 takes O(log(n)) time, and since there are k iterations, the total running time is O(k log(n)). Since k < 2n/log(n), we get that Step 3 runs in O(k log(n)) = O(n) time.
- Step 4 clearly takes O(n) time.
- Therefore, the total running time is O(n), as required.