CS 171 - Spring 2016

Final exam

Full Name: _____ **OPUS ID:** _____ **Section:** __ 000 (Vigfusson)
__ 001 (Fossati)
__ 002 (Wildani)

My answers to this exam are my own work. I understand that this exam is governed by the Emory Honor Code.

Signature: _____

Grading:

1a	**1b**	**1c**	**1d**	**1e**	**1f**	**1g**	**1h**	**1i**	**1j**	**2a**	**2b**	**2c**	**2d**			
2	2	2	2	2	2	2	2	2	2	4	4	4	4			

3a	**3b**	**4**	**5**	**6**	**7**	**8a**	**8b**	**9a**	**9b**							**Tot**
4	4	4	4	4	4	10	10	10	10							100

1. For each of the following statements, circle whether it is true or false, and briefly explain why. N is the number of elements in a data structure. For the big-O notation, consider the lowest possible bound for the worst case unless specified.

1.a) Sorting an array using merge sort takes O(N^2) time.

True **False** Explanation:

1.b) Inserting an element at the beginning of an array takes O(N) time.

True False Explanation:

1.c) Deleting 3 elements from the middle of a sorted array takes O(1) time.

True **False** Explanation:

1.d) Inserting an element to the front of a linked list takes O(1) time.

True False Explanation:

1.e) Inserting N elements from a random linked list into an empty binary search tree takes O(NlogN) time.

True False Explanation:

1.f) Finding an element in an extremely unbalanced binary search tree takes O(logN) time.

True **False** Explanation:

1.g) Finding a key-value pair in a hash table (using linear probing) that is almost full takes O(N) time.

True False Explanation:

1.h) Popping an element from a stack and subsequently inserting it into a queue takes O(N) time.

True **False** Explanation:

1.i) Checking if a queue contains an element takes O(N) time.

True False Explanation:

2. What is the run-time complexity in big-O notation of the following blocks of code? Justify your answer. Here are some potentially useful formulas:

1 + 2 + 3 + ... + n = n * (n+1) / 2

1 + x + x^2 + x^3 + ... + x^n = (x^(n+1) - 1) / (x - 1)

2.a)

```
for (int i = N; i < 2 * N; i++) {
      System.out.println("hey");
}
for (int j = 2 * N; j > N; j--) {
      System.out.println("you");
}
```

The first for loop repeats N times. After that, the second for loop also repeats N times. The total is O(N).

2.b)

```
int k = 0;
while (k < N) {
      k++;
      if (k == 5) {
            k = N;
      }
      System.out.println("wow");
}
```

The loop repeats exactly 5 times, which is O(1).

2.c)

```
public static int f(int k) {
      if (k <= 0)
            return 0;
      else
            return 1 + f(k - 1) + f(k - 2) + f(k - 3);
}

System.out.println(f(N)); // this is the call in the main method
```

Each method call causes three recursive calls, generating a recursion trace that develops like a ternary tree with N levels. Total number of recursive calls: O(3^N)

2.d)

```
BinarySearchTree bst = new BinarySearchTree();
for (int i = 0; i < N * N; i++) {
      // Math.random is O(1)
      int x = (int)(Math.random() * 100000000);
      bst.add(x);
}
```

The for loop repeats N^2 times. Each insertion of a random element in the tree in O(logN). Thus the total is O((N^2) * logN).

3. Consider the following solution for `addLast()` in Project 4 (Linked Deque). It compiles and runs, but it contains a logic error.

```
public class LinkedDeque<Item> implements Deque<Item> {
    int N;          // number of elements on deque
    Node first;     // beginning of deque
    Node last;      // end of deque

    class Node {
        Item item;
        Node next;
        Node previous;
    }

    public void addLast(Item item) {
        Node oldlast = last;
        last = new Node();
        last.item = item;
        last.next = null;
        last.previous = last;
        if (isEmpty())
            first = last;
        else
            oldlast.next = last;
        ++N;
    }
}
```

3.a) Explain the mistake in the method `addLast()`. Draw a picture that clearly shows what happens.

The .previous field of the new node is not correctly set to the node that was originally the last one, but it is set to itself.

3.b) How would you fix the code?

Replace the line last.previous = last with last.previous = oldlast

4. Consider the following Java code.

```
public static void replicateElements(Queue<Integer> q, int k) {
      for (int i = 0; i < q.size(); i++) {
            int x = q.remove();
            for (int j = 0; j < k; j++) {
                  q.add(x);
            }
      }
}
```

This method is supposed to replicate all the elements in the given queue `k` times. For example, if queue `q` contains `[1, 2, 3]`, after the call `replicateElements(q, 3)` it should contain `[1, 1, 1, 2, 2, 2, 3, 3, 3]`. However, the method contains an error. Explain the error and fix it.

If the input queue is not empty and k > 1, at each iteration the outer loop will remove one element but the inner loop will insert more than one. Thus, the index i will always be smaller than q.size() and the loop will never end. To fix it, add a line int n = q.size() before the first loop, and replace the condition i < q.size() with i < n.

5. You have a hash table that maps strings to strings. The table has 6 slots. It is managed with linear probing for conflict resolution. The hash function is the following:

```
H(key) = length(key) % num_slots
```

Fill in the full content of the table after each of these consecutive operations (for example, step 2 builds upon the result of step 1).

(1) put("cat", "A")

i	key	value
0		
1		
2		
3	**cat**	**A**
4		
5		

(2) put("elephant", "B")

i	key	value
0		
1		
2	**elephant**	**B**
3	**cat**	**A**
4		
5		

(3) put("dog", "C")

i	key	value
0		
1		
2	**elephant**	**B**
3	**cat**	**A**
4	**dog**	**C**
5		

(4) put("lizard", "D")

i	key	value
0	**lizard**	**D**
1		
2	**elephant**	**B**
3	**cat**	**A**
4	**dog**	**C**
5		

(5) put("fish", "E")

i	key	value
0	**lizard**	**D**
1		
2	**elephant**	**B**
3	**cat**	**A**
4	**dog**	**C**
5	**fish**	**E**

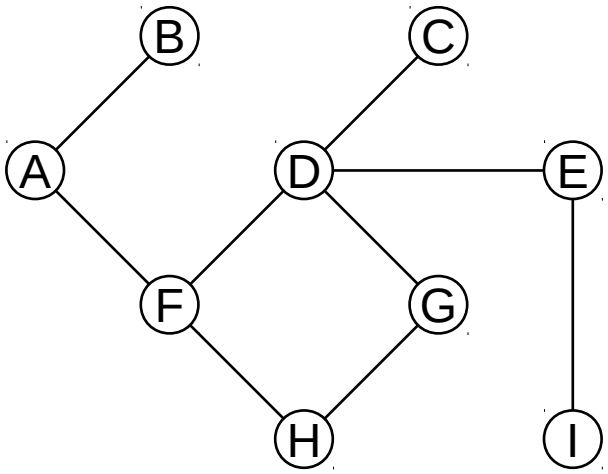
(6) put("dog", "F")

i	key	value
0	**lizard**	**D**
1		
2	**elephant**	**B**
3	**cat**	**A**
4	**dog**	**F**
5	**fish**	**E**

6. Sort the list of numbers 5, 7, 3, 2, 8, 1, 4, 6 using **selection sort**. Show all the steps.

1 7 3 2 8 5 4 6
1 2 3 7 8 5 4 6
1 2 3 7 8 5 4 6
1 2 3 4 8 5 7 6
1 2 3 4 5 8 7 6
1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8

7. Look at the following graph. List the order in which you visit all the vertices using Breadth First Search (BFS) starting from vertex A. Choose the selection order of the neighbors alphabetically.



A B F D H C E G I

8. A double-ended linked list keeps a reference to both the head and the tail nodes of the list. Complete the methods `addToTail()` and `sumMultiplesOf()` in the double-ended linked list class below.

```
class Node {
      int data;
      Node next;

      public Node(int data) {
            this.data = data;
      }
}

class LinkedList {
      Node head;
      Node tail;

      // (8.a) Inserts an element x at the end of this list.
      public void addToTail(int x) {

            Node n = new Node(x);
            if (head == null) {
                  head = n;
            } else {
                  tail.next = n;
            }
            tail = n;


      }
```

```
	// (8.b) Sums all the elements in the list that are multiple of k.
	// Returns the sum.
	public int sumMultiplesOf(int k) {

		int sum = 0;
		Node t = head;
		while (t != null) {
			if (t.data % k == 0) {
				sum += t.data;
			}
			t = t.next;
		}
		return sum;


	}
}
```

9. Complete the methods `areAllMultiples()` and `sumAll()` in the binary search tree class below. Hint: you can write additional private recursive helper methods if you need so.

```
class Node {
      int data;
      Node left;
      Node right;
}

public class BinarySearchTree {

      Node root;

      // (9.a) Returns true if all the values in the tree are multiple
      // of k, false otherwise. If the tree is empty, returns true.
      public boolean areAllMultiples(int k) {

            return areAllMultiples(root, k);
      }

      private boolean areAllMultiples(Node n, int k) {
            if (n == null) {
                  return true;
            } else {
                  return (n.data % k == 0) && areAllMultiples(n.left, k)
                                           && areAllMultiples(n.right, k);
            }
      }
```

```
	// (9.b) Sums all the values in the tree.
	// If the tree is empty, returns 0.
	public int sumAll() {

		return sumAll(root);
	}

	private int sumAll(Node n) {
		if (n == null) {
			return 0;
		} else {
			return n.data + sumAll(n.left) + sumAll(n.right);
		}
	}


}
```