CMPT225, Spring 2021

Midterm 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 March 12, 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. (3 pts each) For each sentence decide whether it is True or False.
Write a brief explanation.

1) Let $T(n) = 10n^4 + 5n + 3$. Then $T = \Omega(n^3)$.

Answer: True.
Because $T(n) > n^4 > n^3$ for all n. Therefore, by definition $T = \Omega(n^3)$.

2) Let $T(n) = 10^n$. Then $T = \Theta(2^n)$.

Answer: False.
Because $T(n)/2^n$ goes to infinity, and is not bounded by a constant. That is, T(n) is not $O(2^n)$

3) For all positive integers n, the function foo(n) will return 0.

```
public int foo(int n) {
    if (n>0)
        foo(n-1);
    return 0;
}
```

Answer: True (assuming the line is "return foo(n-1)")
Because there will be at n recursive calls, after which foo(0) will return 0.

The answer "will not compile" will also be accepted as correct.

4) For all integers n (positive or negative), the function bar(n) will return 0.

```
public int bar(int n) {
    if (n>0)
        return bar(2*n);
    return 0;
}
```

Answer: True.
If n<=0, this is clear. If we start with n>0, then eventually the 2*n will become higher than MAX_INT and the result will be a negative number

B. (5 pts) Use big-O notation to express the running time of foo() on an array of length n. **Explain your answer.**

```
public void foo(int array[]) {
        fooRec(array, 0, array.length-1);
}

public static void fooRec(int array[], int start, int end) {
        if (start == end)
                array[start]++;
        else {
                int mid = (end + start)/2;
                fooRec(array, start, mid);
                fooRec(array, mid+1, end);
                fooRec(array, mid+1, end);
        }
}
```

Answer: the running time can be written as $T(n)=3T(n/2) + O(1)$.
By Master Theorem we get that the running time is $O(n^{\log_2(3)})$, which is $O(n^{1.585})$.

C. (4 pts) Use big-O notation to express the running time of bar() on an array of length n. **Explain your answer.**

```
public static void bar(int array[]) {
        barRec(array, 0, array.length-1);
}

public static void barRec(int a[], int start, int end) {
        if (start <= end) {
                for (int i = start; i <= end; i++)
                        a[i] += 1;
                barRec(a, start+1, end-1);
        }
}
```

Answer: the running time can be written as $T(n)=T(n-2) + O(n)$. This gives
$T(n) < T(n-2) + Cn < T(n-4) + C(n-2) + Cn < C(n + (n-2) + (n-4) + (n-6)...)=O(n^2)$

(4 pts) Rewrite bar() so that it has the same functionality, but the running time is O(n). **Explain your answer.**

Answer: the function does the following:

- increases array[0] and array[length-1] by 1,
- increases array[1] and array[length-2] by 2,
- increases array[2] and array[length-3] by 3.
- etc...,

```
public static void bar(int array[]) {
        for (int i = 0; i <array.length/2;  i++)
                array[i] +=(i+1);
        for (int i = 0; i = array.length/2; i<array.length;  i++)
                array[i] +=(array.length-i);
  }
```

Pay attention to array[array.length/2]. Take care of length being odd/even.

Problem 2 [25 points]

A. (15 pts) Write a class StackReverse that supports the following operations, with running time O(1) for each operation.

```
public class StackReverse<T> {

	public StackReverse() - a constructor, creates an empty stack

	public void push(T item) - adds an element to the stack

	public T pop() - removes an element from the stack

	public void reverse() - reverses the order of the elements. That is,
		the element that was the last in the stack becomes the first,
		and vice versa

	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).

Before writing code, explain your answer.

Idea: use a doubly linked list (java.util.LinkedList in java), and a data field saying if we add/remove elements to/from front or to/front back. Clearly all operations run in O(1) time

```
public class StackReverse<T> {

	LinkedList<T> list;

	boolean front;


	public StackReverse() {
		list = new LinkedList<T>();
		front = true;
	}


	public void push(T item) {
		if (front)
			list.addFront(item);
		else
			list.addLast(item);
	}
```

```
        public T pop() {
                if (front)
                        return list.getFront();
                else
                        return list.getLast();
        }

        public void reverse() {
                front = !front
        }

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

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

}
```

B. (10 pts) Write an algorithm that gets an expression as a String in prefix notation and returns a String with the expression in postfix notation.
For example, on input "/ * 2 + 3 4 - 18 16" the method returns "2 3 4 + * 18 16 - /".
You may assume the input is always valid
Explain your answer.

Idea: probably the easiest way to do it is to use a recursive helper method.
The helper method gets an array of tokens and boundaries: start and end.
If start== end, then the expression is just a number.
Otherwise, the first token is an operator, and we apply recursion on each term.
Compute the first term (by counting how many numbers/operators there are).
Convert each term recursively, and add them to the output in the postfix order.

It is probably possible to do it using a stack adding the tokens right to left into the stack, but seems a bit messy to me.

```
public static String prefix2postfix(String prefix) {
	String[] tokens = prefix.split(" ");
	return prefix2postfix(tokens, 0, tokens.length-1);
}

public static String prefix2postfix(String[] prefix, int start, int end) {
	if (start == end)
		return prefix[start];
	else {
		int end1 = endOfTerm(prefix, start+1);
		String term1 = prefix2postfix(prefix, start+1, end1);
		String term2 = prefix2postfix(prefix, end1+1, end);
		return term1 + " " + term2 + " " + prefix[start];
	}
}

public static int endOfTerm(String[] prefix, int start) {
	int i=start;
	int count = 0;
	while (count != 1) {
		if (isOperation(prefix[i]))
			count--;
		else if (isNumber(prefix[i]))
			count++;
		i++;
	}
	return i-1;
}
```

Problem 3 [25 points]

In this problem use the following definition of Binary Tree. You may assume the classes have the standard getters/setters.

```
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. (10 pts) Write the method equals(Object other) for the class Binary Tree. The method returns true if the argument is a BinaryTree with the same data. You should compare the data in different nodes using equals() method in the class T.

The running time should be O(n) in the worst case, where n is the size of the smaller tree. For example, if one tree has n vertices, and the other has n^2 vertices, the running time should be O(n). **Explain your algorithm and running time**.

Idea: recursively compare the trees by comparing the root, and then left child, and then right child. Stop the first time you see inequality. Clearly the running time is linea in the size of the smaller of the trees

```
public boolean equals(Object other) {
              if (this == other)

                    return true;
              if (other == false)
                    return false;
              if (!(other instanceof BinaryTree))
                    return false;
              // compare the nodes.
              BTNode<T> thisRoot = this.getRoot();
              BTNode<T> otherRoot = ((BinaryTree<T>)other).getRoot();
              return deepCompare(thisRoot, otherRoot);
}

public boolean deepCompare(BTNode<?> root1, BTNode<?> root2) {
       if (root1 == root2)                    return true;
       if (root1 == null && root2 != null)    return false;
       if (root2 == null && root1 != null)    return false;
       return (root1.getData().equals(root2.getData())
              && deepCompare(root1.getLeftChild(), root2.getLeftChild())
              && deepCompare(root1.getRightChild(), root2.getRightChild()));
}
```

(10 pts) Write a method for the class Binary Tree that returns the depth of the shallowest leaf in the tree. That is, among all leaves, you need to return the smallest depth of a leaf. What is the running time of your algorithm?
Explain your algorithm and running time.

Idea: recursively look at the left and the right subtrees, and compute the minimum of them and add 1 to it.
Note that if a node has only 1 child, then we should call recursion only on that one child, and not take minimum.
The running time of the algorithm is O(size of the tree)

Alternatively, it is possible to run BFS, and count layers until reaching a leaf for the first time. This would be more efficient that the recursive approach if the tree is large but has a leaf at a low depth

```
	public int depthOfShallowestLeaf() {
		if (this.root==null)
			return -1;
		else
			return depthOfShallowestLeafRec(this.root);
	}

	public int depthOfShallowestLeafRec(BTNode<?> root) {
		if (root.isLeaf())
			return 0;
		else if (root.getLeftChild() == null)
			// not leaf, so right child must be not null
			return depthOfShallowestLeafRec(root.getRightChild());
		else if (root.getRightChild() == null)
			// not leaf, so left child must be not null
			return depthOfShallowestLeafRec(root.getLeftChild());
		else
			return 1+Math.min(
					depthOfShallowestLeafRec(root.getLeftChild()),
					depthOfShallowestLeafRec(root.getRightChild()));
}
```

B. (5 pts) Write a definition of a Ternary Tree in Java. Each node has data of a generic type, a pointer to the parent, and at most three children: left, middle, and right.

Idea: same a binary tree, but with three children:

```
public class TernaryTree<T> {
	private TernaryNode<T> root;

	public TernaryTree(TernaryNode<T> root) {
		this.root = root;
	}

	+ getters/setters
}



public class TernaryNode<T> {
	private T data;
	private TernaryNode<T> leftChild;
	private TernaryNode<T> middleChild;
	private TernaryNode<T> rightChild;
	private TernaryTNode<T> parent;
		+ constructor/getters/setters
}
```

Problem 4 [25 points]

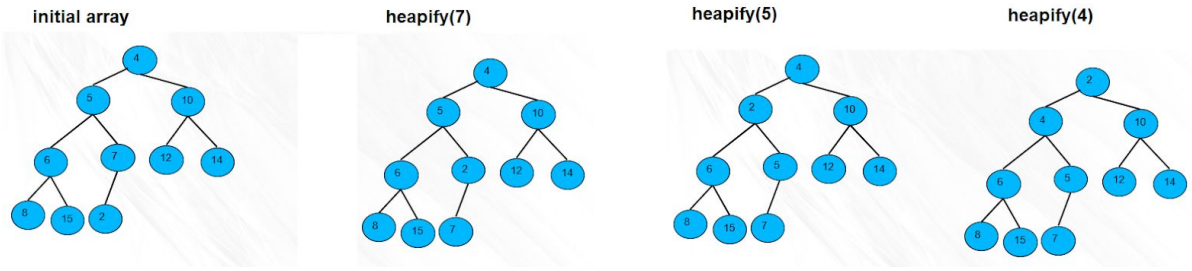
A. (6 pts) Let A = [4, 5, 10, 6 , 7, 12, 14, 8, 15, 2].

- Apply build-minHeap algorithm on A using the linear time algorithm we saw in class.
- Draw the tree representation of the heap.
- Draw the array representing the heap.
- **Draw the intermediate steps.**

Recall the buildHeap algorithm:

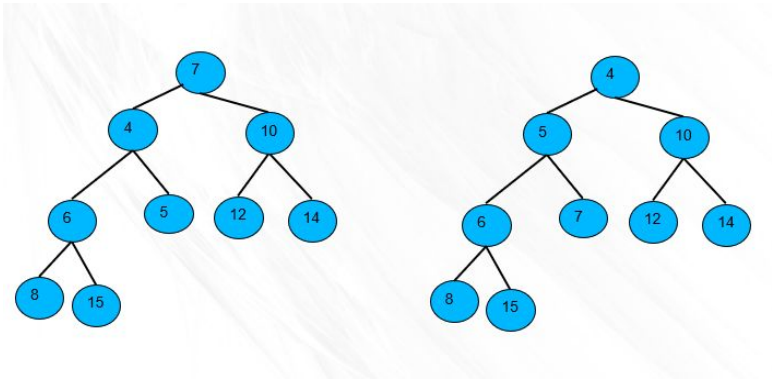
```
Treat the array as a complete binary tree
For each vertex v starting from the bottom
    Apply heapify(v)
```

The sequence will be as follows:
Heapifying all others doesn't affect the tree.



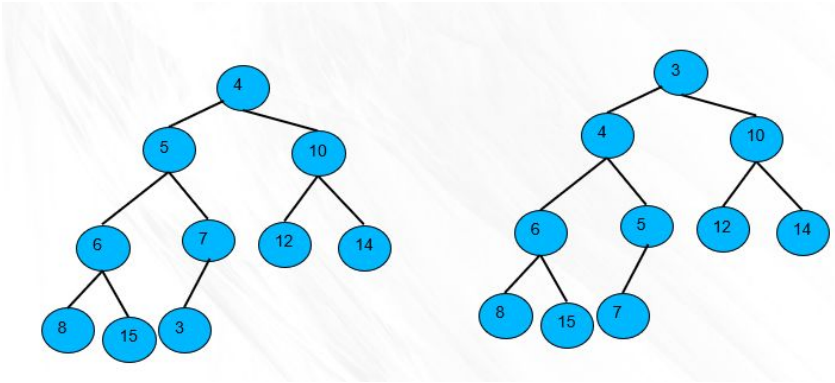
B. (3 pts) Apply removeMin() on the heap obtained in part A, and draw the resulting heap.

Remove 2 and return it. Move 7 to be the root, and apply heapify() on the root.



C. (3 pts) Add 3 to the heap obtained in part B, and draw the resulting heap.

Add 3 as the last node, and then push it up



D. (13 pts) Write a function that gets an array A of length n of integers, and $0 \le k \le n$, and returns an array B of length k containing the smallest k elements in A. In the end A must be in the same state as in the beginning.
The **running time** must be **O(n log(k))** and **extra space used** should be **O(k)**.

For example, on input A =[4,1,5,7,2,3,1,3] and k=4 the output should be B = [1,1,2,3]. The order of the elements in B is not important

Explain your idea before writing the code.
Idea: we will use maxHeap of size at most k+1. In particular each operation on the heap takes O(log(k)) time.

The algorithm is the following:

- Add first k elements of A to the maxHeap
- Then for the remaining n-k elements of A do the following:
  - Add it to the maxHeap
  - Remove maximal element from the heap
- Return the k elements in the heap

The running time is clearly O(n log(k)) since each operation on the heap takes O(log(k)) time. Also, the extra space is O(k) - this is the space used by the heap.

For correctness, it is easy to see by induction that in each iteration the heap contains the minimal k elements so far.