CSC 115
Midterm Exam #3:
Sections: A01 and A02
Thursday, July 30th, 2020

Name:____**KEY**_____(please print clearly!)

UVic ID number:_____

Signature:_____

Exam duration: 60 minutes

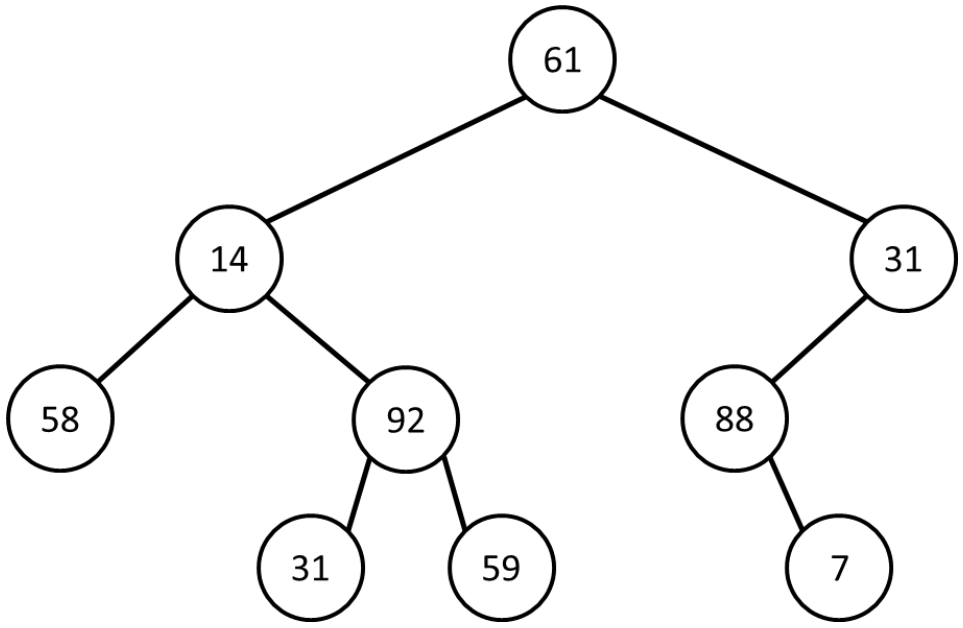
Instructor: Anthony Estey

Students must check the number of pages in this examination paper before beginning to write, and report any discrepancy immediately.

- We will not answer questions during the exam. If you feel there is an error or ambiguity, write your assumption and answer the question based on that assumption.
- Answer all questions on this exam paper.
- The exam is closed book. No books or notes are permitted.
  No electronic devices of any type are permitted.
- The marks assigned to each question and to each part of a question are printed within brackets. Partial marks are available.
- There are twelve (12) pages in this document, including this cover page.
- Page 12 is left blank for scratch work. If you write an answer on that page, clearly indicate this for the grader under the corresponding question.
- Clearly indicate only one answer to be graded. Questions with more than one answer will be given a **zero grade**.
- It is strongly recommended that you read the entire exam through from beginning to end before beginning to answer the questions.
- Please have your ID card available on the desk.

Part 1: Tree Traversals (8 marks)

Provide the pre-order, post-order, in-order and level-order traversals for the following binary tree:



a) pre-order: 61, 14, 58, 92, 31, 59, 31, 88, 7

b) post-order: 58, 31, 59, 92, 14, 7, 88, 31, 61

c) in-order: 58, 14, 31, 92, 59, 61, 88, 7, 31

d) level order: 61, 14, 31, 58, 92, 88, 31, 59, 7

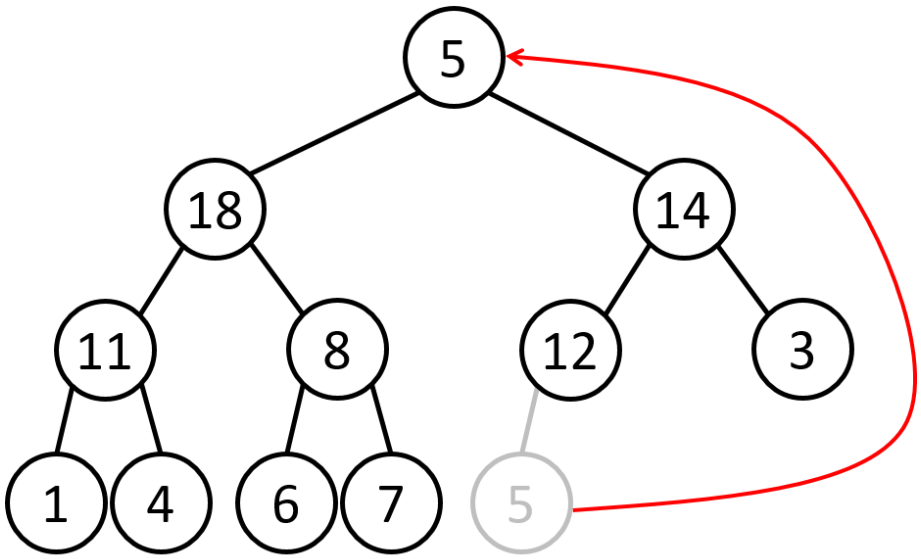
Part 2: Heaps (x marks)

When an element is removed from a heap, the element in the last position is swapped up to the root, as shown in the tree image above.

After the swap occurs, the removal operation is not complete until both heap properties are restored.

Given that the heap is implemented using a 1-based array, fill in the contents of the array after the removal operation has completed (and the heap properties have been restored).

(I have put a - in index 0 as it will be empty in a 1-based array implementation)



-	18	11	14	5	8	12	3	1	4	6	7
0	1	2	3	4	5	6	7	8	9	10	11

Part 3: Inheritance (3 marks)

Given the code for classes A and B shown above, which of the following statements are true? Circle all that apply

```
class A {
    int x;
    public A() {
        System.out.println("A constructor 1");
        this.x = 0;
    }
    public A(int x) {
        this.x = x;
        System.out.println("A constructor 2");
    }

    public void doStuff() {
        System.out.println("A's stuff:" + x);
    }
}

public class B extends A implements Comparable<A> {
    public B() {
        System.out.println("B constructor 1");
    }
    public B(int x) {
        super(x);
        System.out.println("B constructor 2");
    }

    public void otherStuff() {
        System.out.println("B's other stuff: " + x);
    }

}
```

a) B inherits field x from A

b) B overrides the doStuff method from A

c) To compile without errors, class A needs to have an otherStuff method defined

d) To compile without errors, class B needs to have a doStuff method defined.

e) To compile without errors, class B needs to have a compareTo method defined

f) A ex1 = new A(1); ex1.otherStuff() will generate an error

g) B ex2 = new B(2); ex2.otherStuff() will generate an error

h) A ex3 = new B(); will call B's constructor 1 and none of A's constructors

i) A ex4 = new B(4) will call B's constructor 2 and A's constructor 2.

Part 4: Hash Tables (9 marks)

Given the hash table shown below, insert the elements 16, 38, 34, and 41 using the following open addressing collision handling schemes:

For all parts, assume the hash function is $h(k) = k\%7$.
The secondary hash function is $h_2(k) = 5 - k\%5$

a) Linear probing:

41	-	72	45	16	38	34
0	1	2	3	4	5	6

b) Quadratic probing:

34	41	72	45	38	-	16
0	1	2	3	4	5	6

c) Double hashing:

34	-	72	45	41	38	16
0	1	2	3	4	5	6

Part 5: Tree coding (6 marks)

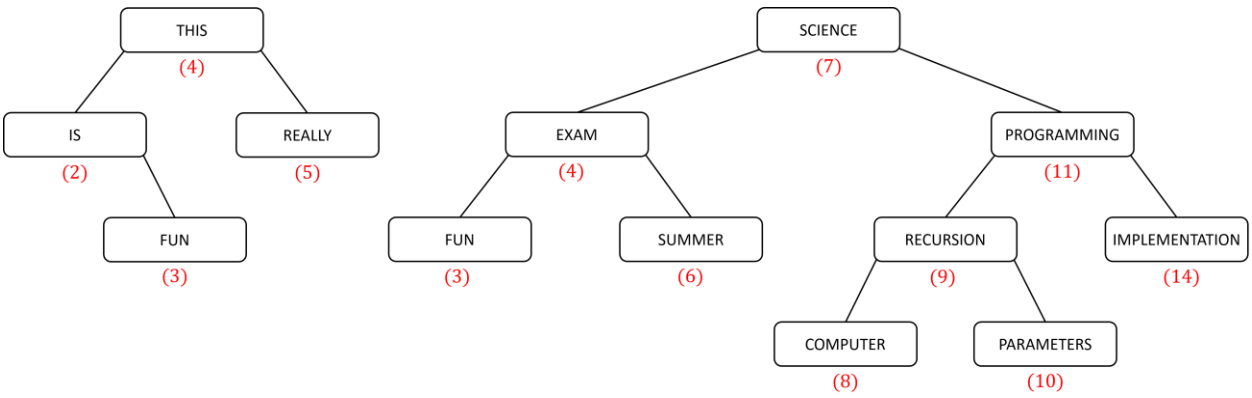
```
public class TreeNode {
    public String data;
    public TreeNode left;
    public TreeNode right;

    public TreeNode(String data) {
        this.data = data;
        left = null;
        right = null;
    }
}
```

In this problem you will make use of the TreeNode class defined above, where each node has a String data value. The TreeNode's form the Binary Search Tree for this problem, which is ordered by the length of data string. For all nodes in the tree:

- Nodes containing data strings with a length shorter than the node's are found in the node's left subtree
- Nodes containing data strings with a length longer than the node's are found in the node's right subtree.

Note the BST is NOT ordered alphabetically. The ordering relationship is illustrated in the two trees shown in the following image, where each node contains a data string (and the lengths of the strings are shown in red below the node):



Complete the find method specified below for a binary search tree with the ordering described and shown above. For full marks, your solution must utilize the ordering (ie. you should not need to search through all nodes in the tree to determine if a String exists).

```java
/*
 * Purpose: determines whether the tree contains a node
 *          with the given data value
 * Parameters: String data - the data value to find
 * Returns: boolean - true if found, false otherwise
 * Note: The order property for this tree is that for any node n,
 *   all elements in n's left subtree contain string's with fewer
 *   characters than n's string, and and all elements in n's right
 *   subtree contain strings with more characters than n's string.
 */
public boolean find(String data){
    //TODO: implement this
}
```

Assume the tree contains at most one element with a data string of any given length (ie. no two nodes will have a data string with the same number of characters). You may create a helper function if you wish.

```
Recursive solution:
public boolean find(String data){
      return find(root, data);
}

public boolean find(TreeNode cur, String data) {
      if (cur==null) {
            return false;                  // could also be:
      } else if (cur.data.equals(data)) { //cur.data.compareTo(data)==0
            return true;
      } else if (cur.data.length() < data.length()) {
            return find(cur.right, data);
      } else {
            return find(cur.left, data);
      }
}

Iterative solution:
public boolean find(String data){
      TreeNode cur = root;
      while (cur != null) {
            if (cur.data.equals(data)) {
                  return true;
            } else if (cur.data.length() < data.length()) {
                  cur = cur.right;
            } else {
                  cur = cur.left;
            }
      }
      return false;
}
```

Grade breakdown:
6 marks total: -2 per mistake

- missing/incorrect check if node==null then return false
- missing/incorrect searching left if data's length is less than nodes
- missing/incorrect searching right if data's length is greater nodes
- missing/incorrect return true under the following circumstances:
  - cur.data.equals(data)
  - cur.length == data.length()
  - cur.data.compareTo(data)==0

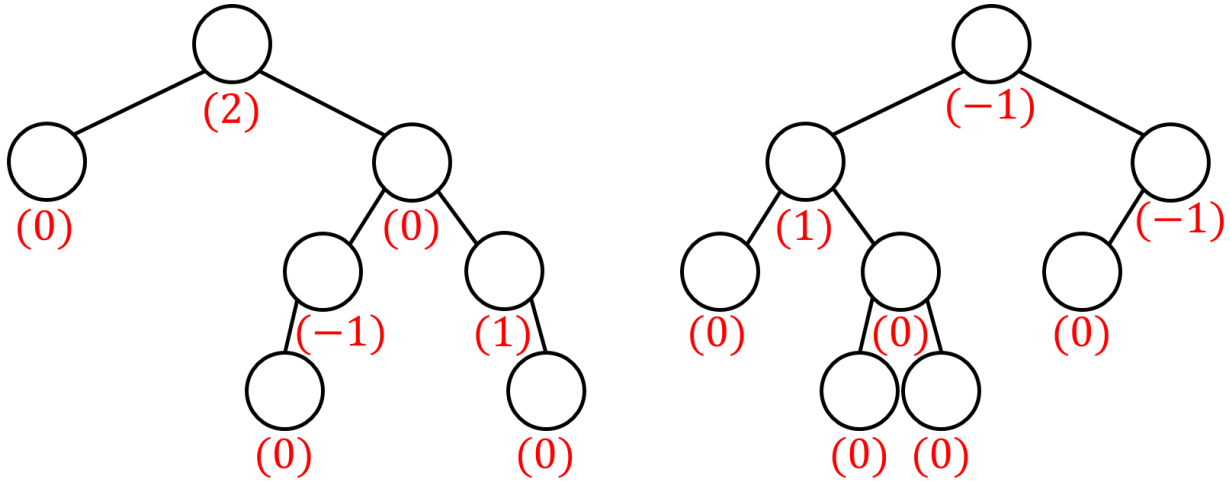
Part 6: More tree coding (6 marks)

In this problem you will determine the imbalance value of a node in a tree:

- If a node is null, or if a node's left and right subtrees are the same height, its imbalance value is 0.
- If a node's left subtree's height is greater than its right subtree's height, it has a negative imbalance value equal to the number of levels higher the left subtree is than the right subtree.
- If a node's right subtree's height is greater than its left subtree's height, it has a positive imbalance value equal to the number of levels higher the right subtree is than the left subtree.

For example, If a node's right subtree has a height of 8 and its left subtree has a height of 3, its imbalance value is 5. If a node's left subtree has a height of 4 and its right subtree a height of 3, its imbalance value is -1.

The following image shows two trees, where each node is labeled with its imbalance value:



Complete the balance method shown in the code below:

```
/*
 * Purpose: Get the imbalance value of the given node
 * Parameters: TreeNode n - the node
 * Returns: int - the imbalance value
 */
public int imbalance(TreeNode n) {
    // TODO: implement this
}
```

Similar to TreeNodes we have worked with throughout the term, you can access each TreeNode's left and right child. You will likely need to make a helper method to implement the imbalance method correctly.

```
public int imbalance(TreeNode n) {
      if (n == null) {
            return 0;
      } else {
            return height(n.right) - height(n.left);
      }
}

private int height(TreeNode n) {
      if (t == null) {
            return -1;
      } else {
            return 1 + Math.max(height(t.left), height(t.right));
      }
}
```

Grade breakdown:

6 marks total:
-2 per mistake, -1 if attempt but not correct:

- missing/incorrect check if node is null before calculating left/right heights
- missing/incorrect returning difference of right minus left heights
- missing/incorrect helper method to determine node height

Part 7: Hash Table coding (6 marks)

The following code shows an implementation of the put method from assignment 7. It compiles, but fails some tests:

```
public class HashMap<K extends Comparable<K>, V> {

    private List<List<Entry<K,V>>>  table;
    private int numElements;

    public HashMap() { // assume this works correctly
        table = new ArrayList<List<Entry<K,V>>>(1000003);
        for (int i = 0; i < tableSize; i++) {
            table.add(new LinkedList<Entry<K,V>>());
        }
        numElements = 0;
    }

    public boolean containsKey(K key) {
        ... // assume this works correctly
    }

    public V get (K key) throws KeyNotFoundException {
        ... // assume this works correctly
    }

    public void put (K key, V value){
        int index = Math.abs(key.hashCode())%tableSize;
        List <Entry<K,V>> listAtIndex = table.get(index);

        if (listAtIndex.isEmpty()) {
            listAtIndex.add(new Entry<K,V> (key, value));
            numElements++;
        } else {
            for (int i=0; i<listAtIndex.size(); i++) {
                Entry<K,V> cur = listAtIndex.get(i);
                if (cur.key.equals(key)) {
                    cur.value = value;
                } else {
                    listAtIndex.add(new Entry<K,V> (key, value));
                    numElements++;
                }
            }
        }
    }
}
```

Describe the problems with the implementation of put shown above, and explain how to fix them so that put works correctly.

Clearly indicate what the problem is, and how to fix it. You can provide code examples if you wish, but it is not necessary; a written explanation is fine.

The main problem with the code above is the else statement within the for-loop.

The for-loop is supposed to search the linked list of elements that also hashed to the same index (as this is obviously usually separate chaining). The idea is to first determine if there is a matching key, and if so, update the value associated with the key (which is what happens if the if-statement is true). If there are no elements with a matching key, then a new data entry is created and added to the back of the linked list.

The problem is that as you iterate through every item in the linked list inside the for-loop, a new entry is created every time an element's key does not match the key passed as a parameter. This can result in multiple new elements added to the linked list, instead of just adding a single element if no matching key is found in the list.

There are multiple ways to fix this, most involve moving the code in the else statement outside the for-loop. As long as the code updates the value if a matching key is found, and inserts a new key-value entry if no key is found, then that's fine.

Example 1:

```
boolean found = false;
for (int i = 0; i < listAtIndex.size(); i++) {
      Entry<K,V> cur = listAtIndex.get(i);
      if (cur.key.equals(key) {
            cur.value = value;
            found = true;
            break;
      }
}
if (found == true) {
      listAtIndex.add(new Entry<K,V> (key, value);
      numElements++;
}
```

Example 2:

```
for (int i = 0; i < listAtIndex.size(); i++) {
      Entry<K,V> cur = listAtIndex.get(i);
      if (cur.key.equals(key) {
            cur.value = value;
            return;
      }
}
listAtIndex.add(new Entry<K,V> (key, value);
numElements++;
```

... Left blank for scratch work...

END OF EXAM

Question	Value	Mark
Part 1	8	
Part 2	6	
Part 3	4	
Part 4	9	
Part 5	6	
Part 6	6	
Part 7	6	
Total	**45**	