CSC 115
Midterm Exam #3:
Sections: A01 and A02
Thursday, December 3rd, 2020

Name:____KEY____(please print clearly!)

UVic ID number:____

Signature:____

Exam duration: 50 minutes

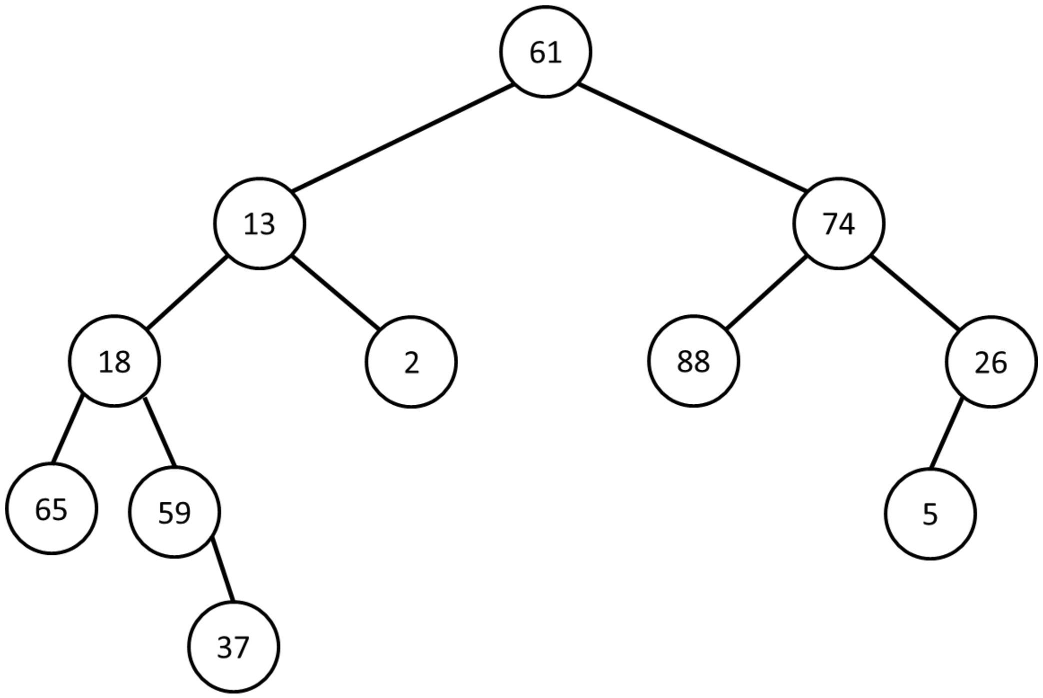
Instructor: Anthony Estey

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 (11) pages in this document, including this cover page.
- Page 11 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 (4 marks)

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



a) pre-order: 61, 13, 18, 65, 59, 37, 2, 74, 88, 26, 5

b) post-order: 65, 37, 59, 18, 2, 13, 88, 5, 26, 74, 61

c) in-order: 65, 18, 59, 37, 13, 2, 61, 88, 74, 5, 26

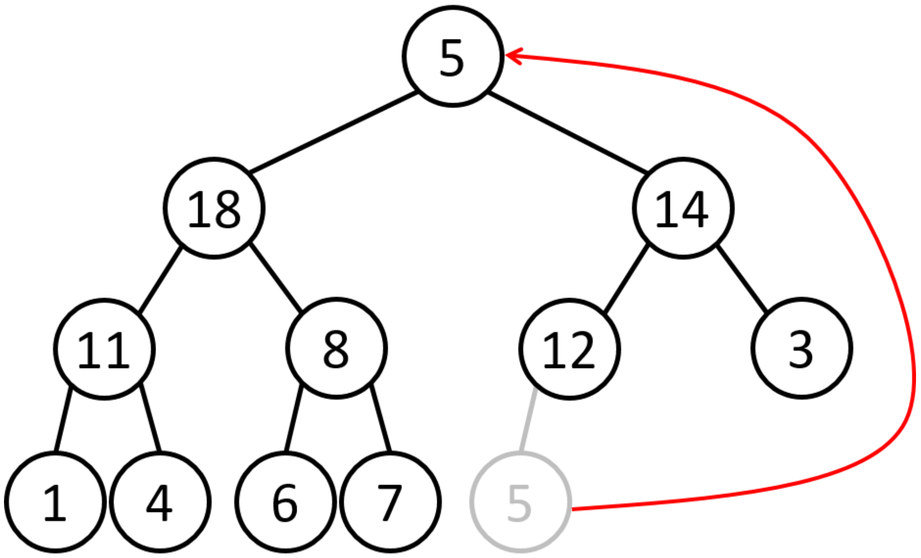
d) level order: 61, 13, 74, 18, 2, 88, 26, 65, 59, 5, 37

Part 2: Heaps (3 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 index numbers of the first 3 swaps that will occur during the bubbleDown process. If less than 3 swaps occur, fill in the blanks with a - instead of an index number.



First swap: parent index:	1	child index:	2
Second swap: parent index:	2	child index:	4
Third swap: parent index:	-	child index:	-

Part 3: Inheritance (4 marks)

Examine the following three classes:

```
public class Dessert {
    private String name;

    public Dessert() {
        name = "dessert";
    }

    public Dessert(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public String toString() {
        return "Mmmmm... "+name;
    }
}

public class FrozenTreat extends Dessert {
    private String flavour;

    public FrozenTreat(String flavour) {
        this.flavour = flavour;
    }

    public FrozenTreat(String flavour, String name) {
        super(name);
        this.flavour = flavour;
    }
}

public class BakedGood extends Dessert {
    private String type;

    public BakedGood(String type) {
        this.type = type;
    }

    public BakedGood(String type, String name) {
        super(name);
        this.type = type;
    }
}
```

Notice that the BakedGood and FrozenTreat classes both extend the Dessert class.

Assume you have made an array of desserts and filled it with all three types of desserts, as shown in the image below.

array:

0	Dessert name: pie
1	FrozenTreat name: popsicle flavour: orange
2	FrozenTreat name: ice-cream flavour: strawberry
3	BakedGood name: cookie type: Oreo
4	Dessert: name: fudge
5	BakedGood: name: cake type: chocolate

Output Code (cannot be changed):

```
for (int i = 0; i < array.length; i++) {
    System.out.println(array[i]);
}
```

Current output:

```
Mmmmm... pie
Mmmmm... popsicle
Mmmmm... ice-cream
Mmmmm... cookie
Mmmmm... fudge
Mmmmm... cake
```

Target output:

```
Mmmmm... pie
Mmmmm... popsicle
Mmmmm... ice-cream
Yummy - Oreo cookie
Mmmmm... fudge
Yummy - chocolate cake
```

When using a loop to print the contents of the array, the output is the same for all different types of desserts, as shown in the "Current output" box.. You must modify the class code shown above so that the output produced matches the output shown to the right in "Target output" box in the image above.

To answer the question, indicate which class (or classes) you are modifying (Dessert, BakedGood and/or FrozenTreat), and then write the code to insert into the class to produce the target output.

Need to modify the BakedGood class

Within the BakedGood class, override the toString() method

Change it to:

```
public String toString() {
     return "Yummy - " + type + " " + super.getName();
}

// 1 - Should state or show that only the BakedGood class is updated
// 1 - override toString method
// 1 – method returns a string with baked good's type field
// 1 - call to getName() or super.getName()
       (0 if tried to access name field directly)
```

Part 4: Hash Tables (9 marks)

Given the hash table shown below, insert the elements 24, 36, 52, and 17 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:

	43	36	24	95	52	17
0	1	2	3	4	5	6

b) Quadratic probing:

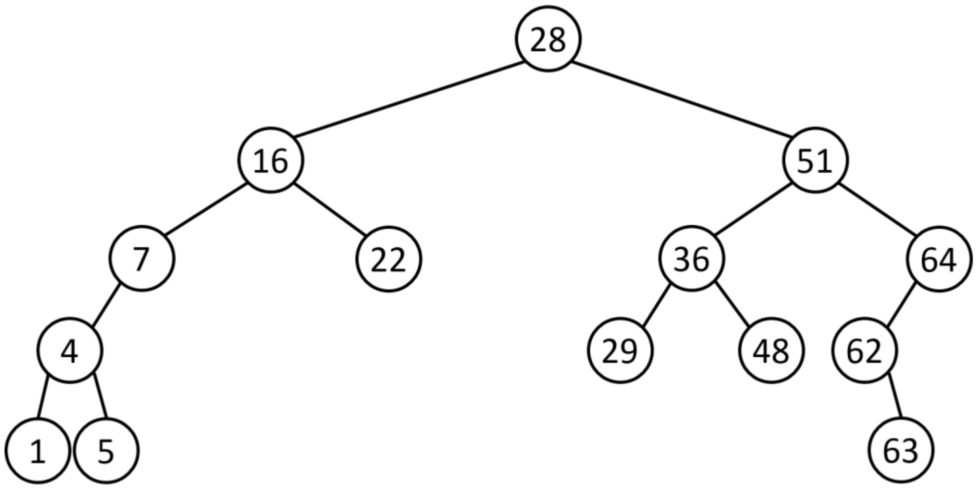
52	43	36	24	95	17	
0	1	2	3	4	5	6

c) Double hashing:

	43	17	24	95	36	52
0	1	2	3	4	5	6

Part 5: Tree Implementation (7 marks)

For this question you will complete a method that determines the number of nodes in a subtree rooted at a given node.



For example, in the BST shown above, the number of nodes in the tree rooted at the node with key 7 is 4 (the tree rooted a 7 has four nodes: 1, 4, 5, and 7). The number of nodes in the tree rooted at the node with key 51 is 7 (it has nodes 29, 36, 48, 51, 62, 63, 64).

Complete the implementation of the sizeOfSubtree method shown below for the BinarySearchTree class.

```
/*
 * Purpose: return the number of nodes in the tree rooted
 *          at the node with the given key
 * Parameters: K key - the key to search for
 * Returns: int - the number of nodes in the subtree,
 *                0 if no node in the tree has the given key
 */
public int sizeOfSubtree(K key) {
    // write the code that goes here
}
```

Any key can be given to the method, including keys not found in the tree. So, given BinarySearchTree b shown above, b.sizeOfSubtree(28) returns 14, whereas b.sizeOfSubtree(55) returns 0.

Remember the sizeOfSubtree method is in the BinarySearchTree class, so you have access to the tree's root, and are expected to make use of the fact that it is a Binary Search Tree. You may want to create helper methods to solve this problem.

```
public int sizeOfSubtree(K key) {
      TreeNode<K> cur = root;
      while (cur != null) {
            if (cur.key.compareTo(key) == 0) {
                  return sizeOfSubTreeRec(cur);
            } else if (cur.key.compareTo(key) < 0) {
                  cur = cur.right;
            } else {
                  cur = cur.left;
            }
      }
      return 0;
}

public int sizeOfSubTreeRec(TreeNode<K> cur) {
      if (cur == null) {
            return 0;
      }
      return 1 + sizeOfSubTreeRec(cur.left) +
                        sizeOfSubTreeRec(cur.right);
}


// Grading rubric:

// 1 - uses compareTo/equals instead of ==
// 2 - use BST property to locate node with key (left/right)
// 1 - return 0 if key not found
// 1 - return size if found
// 2 - calculate number of nodes rooted at tree
```

Part 6: Hash Table Implementation (7 marks)

In this problem you will be working with a hash table that uses separate chaining. You will implement a method that determines the size of the largest linked list across all indexes in the hash table. For example:

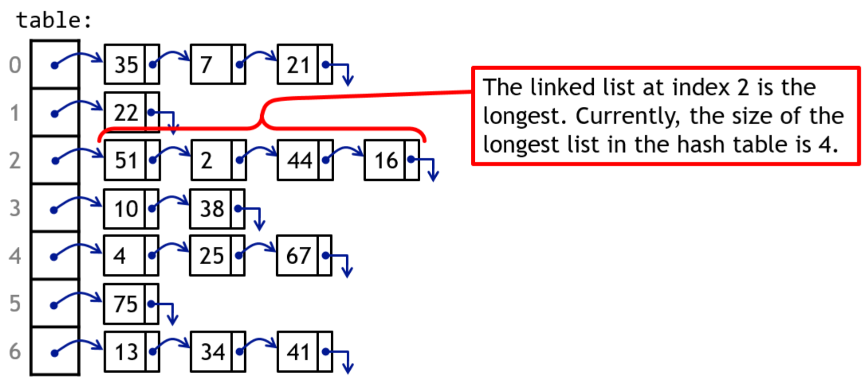


table:
0
1
2
3
4
5
6
35
7
21
22
51
2
44
16
10
38
4
25
67
75
13
34
41
The linked list at index 2 is the longest. Currently, the size of the longest list in the hash table is 4.

The HashTable class is shown below:

```
public class HashTable {
    private static final int m = 7; // prime number
    private List<List<Integer>> table;

    public HashTable() {
        table = new ArrayList<List<Integer>>(m);
        for (int i = 0; i < m; i++) {
            table.add(new LinkedList<Integer>());
        }
    }

    public void insert (int key){
        int index = key%m;
        List<Integer> listAtIndex = table.get(index);
        if (!listAtIndex.contains(key)) {
            listAtIndex.add(key);
        }
    }

    /*
     * Purpose: Find the size of the largest linked list
     *          across all indexes in the hash table
     * Parameters: none
     * Returns: int - size of the largest linked list
     */
    public int largestList() {
        // implement this
    }
}
```

Finish the implementation of the largestList method, which finds the largest linked list across all indexes of the hash table.

<u>Note:</u> To receive full marks, you **must** use an iterator in your solution. You will still get partial marks for a correct solution that does not use an iterator.

```
public int mostCollisionsAtIndex() {

      Iterator<List<Integer>> iter = table.iterator();
      int max = 0;

      while (iter.hasNext()) {
            List<Integer> listAtIndex = iter.next();
            if (listAtIndex.size() > max) {
                  max = listAtIndex.size();
            }
      }
      return max;
}

// Alternate solution:
public int largestList() {
      int max = 0;
      int count = 0;
      for (int i = 0; i < m; i++) {
            List<Integer> list = table.get(i);
            Iterator<Integer> iter = list.iterator();
            count = 0;
            while (iter.hasNext()) {
                  count++;
                  iter.next();
            }
            if (count > max) {
                  max = count;
            }
      }
      return max;
}

// Grading rubric:
// 2- iterate through all indexes in table
// 2- calculate length of list at index
// 3- compare, update, return max
// NOTE: an iterator MUST be used in solution.
```

… Left blank for scratch work…

END OF EXAM

Question	**Value**	**Mark**
Part 1	4	
Part 2	3	
Part 3	4	
Part 4	9	
Part 5	7	
Part 6	7	
Total	**34**	