CSC 115

Midterm Exam:
Thursday, 25 July 2019

Name:____________RUBRIC____(please print clearly!)

UVic ID number:____

Signature:____

Exam duration: 40 minutes

Instructor: Celina Berg

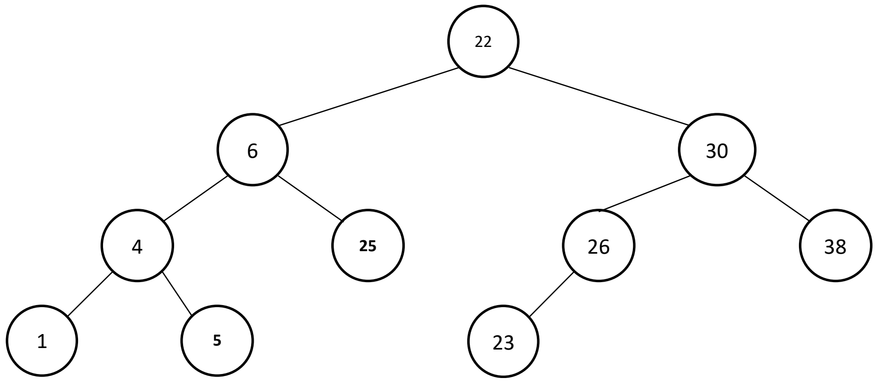
Students must check the number of pages in this examination paper before beginning to write, and report any discrepancy immediately.

- 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.
  Electronic devices are not permitted.
- The marks assigned to each question and to each part of a question are printed within brackets. Partial marks are available.
- There are ten (10) pages in this document, including this cover page.
- 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.

Question 1 (4 marks)

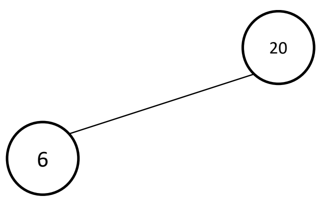
For each question, consider the tree shown and circle any of the terms given that describe that tree.

a)



I. Complete
II. Full
III. Binary search tree
IV. Balanced
V. Max Heap
VI. Binary tree

b)



I. Complete
II. Full
III. Binary search tree
IV. Balanced
V. Max Heap
VI. Binary tree

Grading:

½ mark off for every wrong choice

Question 2 (4 marks)
The following code is an implementation of a `HeapPriorityQueue`. You are to implement the methods commented with: // **TODO:**
TIP: Do not just go from memory of your assignment implementation, be sure to consider carefully the constructor and method implementation provided to you.
NOTE: You do not have to provide an implementation for any methods intentionally omitted.

```
public class HeapPriorityQueue implements PriorityQueue {
      private final static int DEFAULT_SIZE = 10000;

      private Comparable[] storage;
      private int currentSize;

      public HeapPriorityQueue () {
            this(DEFAULT_SIZE);
      }

      public HeapPriorityQueue( int size ) {
            storage = new Comparable[size + 1];
            currentSize = 0;
      }

      public void insert ( Comparable k  ) throws HeapFullException {
            if ( currentSize == storage.length )
                  throw new HeapFullException();

            currentSize++;
            storage[currentSize] = k;

            bubbleUp(currentSize);
      }

      private void bubbleUp ( int index ) {
            // implementation omitted intentionally
            // DO NOT implement
      }

      private void swapElement ( int pos1, int pos2 ) {
            // implementation omitted intentionally
            // DO NOT implement
      }

      private boolean hasLeft ( int pos ) {
            return (leftChild(pos) <= currentSize);
      }

      private boolean hasRight ( int pos ) {
            return ( rightChild(pos) <= currentSize);
      }
```

//continued on the following page…

```
    // TODO: complete implementation
    private int parent ( int pos ) {

        return pos/2;

    }

    // TODO: complete implementation
    private int leftChild ( int pos ) {

        return pos*2;

    }

    // TODO: Complete the implementation
    private int rightChild ( int pos )  {

        return pos*2 + 1;

    }

}
```

Grading
1 mark each
1 mark for taking root being at index 1 into account

Question 3 (10 marks)

The following code is an implementation of a Binary Search Tree. You are to implement the methods commented with: **// TODO:**

NOTE: You are free to use and add helper methods to support your implementation but you cannot change the signature of the public methods provided.

GRADING: A small portion of your grade will be allocated to the efficiency of your algorithm.

```
public class TreeException extends RuntimeException{

}// END of TreeException Class

public class TreeNode {
      private int value;
      private TreeNode left;
      private TreeNode right;

      public TreeNode(int value) {
            this.value = value;
            this.left = null;
            this.right = null;
      }
      public int getValue() {
            return this.value;
      }
      public void setValue(int value) {
            this.value = value;
      }
      public TreeNode getLeft() {
            return this.left;
      }
      public void setLeft(TreeNode newLeft) {
            this.left = newLeft;
      }
      public TreeNode getRight() {
            return this.right;
      }
      public void setRight(TreeNode newRight) {
            this.right = newRight;
      }
}// END of TreeNode Class

public class BinarySearchTree {
      private TreeNode root;

      public BinarySearchTree() {
            this.root = null;
      }
      // This method is left blank intentionally
      // You can assume it follows the expected behavior of
      // a Binary Search Tree.
      // You do NOT have to provide the implementation
      public void insert(int v) { .... }

      //continued on the following page…
```

```java
// TODO: Complete the implementation
// PURPOSE:  counts the number of elements in this
//           BinarySearchTree
// PARAMETERS: none
// RETURNS: (int) – the count
public int count() {

     return count(root);
}

public int count(TreeNode n) {
     if (n==null)
          return 0;
     else
          return 1 + count(n.getLeft()) + count(n.getRight());
}
```

Grading
1 mark helper call + signature
1 mark base case – if returns count, must pass as parameter
1 mark recursive calls
1 mark combine and return the result

```
// TODO: Complete the implementation
// PURPOSE:  counts the number of elements in this
//           BinarySearchTree that are above the given threshold
// PARAMETERS: int threshold
// RETURNS: (int) – the count
public int countAbove(int threshold) {

     return countAbove (root, threshold);
}



public int countAbove (TreeNode n, int threshold) {
     if (n==null)
          return 0;
     else
          if (n.getValue() <= threshold)
               return countAbove (n.getRight());
          else
               return 1  + countAbove(n.getLeft())
                         + countAbove(n.getRight());
}
```

Grading
1 mark helper call + signature
1 mark base case – if returns count, must pass it as a parameter
1 mark recursive calls on subtrees
1 mark for conditional recursive call to only the right subtree
2 marks combine and return the result in > condition

Question 4 (3 marks)
Below is the UML for a `BinaryTree` class that holds integer data. We have intentionally omitted the fields for this question.

BinaryTree	
...	
+ BinaryTree()	
+ insertValue(int):	void
+ findValue(int):	boolean
+ getOdds():	int[]

Imagine you are asked to write a `BinarySearchTree` class that extends `BinaryTree`.

Below we provide a short description of each method to augment the UML. For each method:
1) state whether your `BinarySearchTree` should **inherit** or **override** this method from `BinaryTree` to achieve algorithm correctness and efficiency.

2) provide a BRIEF reasoning for your decision

a) insertValue: inserts the given value into this tree

override – needs to ensure the invariant of a BST is maintained

b) findValue: determines whether the given value is in this tree

override – needs to take into account the BST invariant to make it most efficient

c) getOdds: creates an array of all the values in this tree that are odd numbers

inherit – BST invariant does not affect this method as we still have to traverse the whole tree.

Grading
1 mark each – must have inherit/override + some valid explanation