CSC 115 Practice Exam Questions - SOLUTION

Multiple Choice

1. In a recursive solution, the _____ terminates the recursive processing.
   a. local environment
   b. pivot item
   c. base case
   d. recurrence relation

2. The _____ keyword is used to call the constructor of the parent class.
   a. extends
   b. super
   c. this
   d. implements

3. What is the output of the following program…

```
public class A extends C {
     public A() {
          System.out.println("A constructor 1");
     }
     public A(int x) {
          System.out.println("A constructor 2");
     }
     public void foo() {
          System.out.println("A foo:" + x);
     }
}

public class C {
     int x;
     public C() {
          System.out.println("C constructor 1");
          this.x = 0;
     }
     public C(int x) {
          this.x = x;
          System.out.println("C constructor 2");
     }
     public void foo() {
          System.out.println("C foo:"  + x);
     }
}

public class FinalExamInheritance {
     public static void main(String[] args) {
          C myObject = new A(5);
          myObject.foo();

     }
}
```

   a. C constructor 1
      A constructor 2
      A foo:5
   b. C constructor 2
      A constructor 2
      A foo:0
   c. C constructor 1
      A constructor 2
      A foo:0
   d. C constructor 1
      A constructor 2
      A foo:5

4. A node directly below node n in a tree is called a _____ of node n.
    a. root
    b. leaf
    c. parent
    d. child

5. A node with no children is called a _____.
    a. root
    b. leaf
    c. parent
    d. child

6. In a tree, the children of the same parent are called _____.
    a. leafs
    b. siblings
    c. roots
    d. friends

7. Each node in a tree has _____.
    a. exactly one parent
    b. at most one parent
    c. exactly two parents
    d. at most two parents

8. The _____ of a tree is the number of nodes on the longest path from the root to a leaf.
    a. height
    b. length
    c. width
    d. depth

9. A _____ of height h is full down to level h – 1, with level h filled in from left to right.
    a. perfect binary tree
    b. complete binary tree
    c. balanced binary tree
    d. general tree

10. In a _____, the left and right subtrees of any node have heights that differ by at most 1.
    a. perfect binary tree
    b. complete binary tree
    c. balanced binary tree
    d. binary tree

11. The traversal of a binary search tree is _____.
    a. O(n)
    b. O(1)
    c. $O(n^2)$
    d. $O(\log_2 n)$

12. In an array based representation of a complete binary tree with the root value stored at index 0, which of the following represents the left child of node tree[i]?
    a. tree[i+2]
    b. tree[i–2]
    c. tree[2*i+1]
    d. tree[2*i+2]

13. In an array based representation of a complete binary tree with the root value stored at index 0, which of the following represents the parent of node tree[i]?
    a. tree[i–2]
    b. tree[(i–1)/2]
    c. tree[2*i–1]
    d. tree[2*i–2]

14. The following statement is **not** true
    a. A complete tree is a balanced tree
    b. A balanced tree is a complete tree
    c. A binary search tree is not always a balanced tree
    d. A perfect tree is a balanced tree

15. A heap in which the root contains the item with the smallest key is called a _____.
    a. minheap
    b. maxheap
    c. complete heap
    d. binary heap

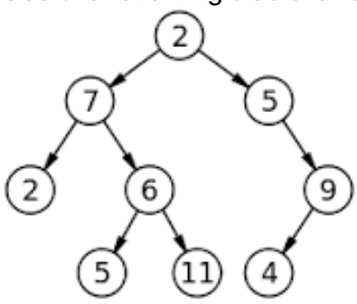
16. A heap is a _____.
    a. general tree
    b. queue
    c. perfect binary tree
    d. complete binary tree

17. What happens when a duplicate key is encountered during the put operation in a Map?
    a. A new entry is inserted, resulting in two entries with the same key
    b. A new key is generated, and an entry with the new key is inserted
    c. The value of the existing entry with the given key is updated
    d. The program crashes

18. Primary clustering is a problem associated with _____.
    a. separate chaining
    b. linear probing
    c. quadratic probing
    d. double hashing

19. With a separate chaining collision strategy the speed of the heap depends on _____.
    a. The length of the linked lists at each index of the hash table
    b. The largest probe sequence in the hash table
    c. The number of tombstones in the hash table
    d. The secondary hash function used when a collision occurs

Use the following tree t for questions 40-45 …



20. Tree t is a min heap
    a. true
    b. false

21. An in-order traversal of t would visit nodes in the following order…
    a. 2 7 5 2 6 9 5 11 4
    b. 2 7 5 6 11 2 5 4 9
    c. 2 7 2 6 5 11 5 9 4
    d. 2 5 11 6 7 4 9 5 2
    e. None of the above

22. A level order traversal of t would visit nodes in the following order…
    a. 2 7 5 2 6 9 5 11 4
    b. 2 7 5 6 11 2 5 4 9
    c. 2 7 2 6 5 11 5 9 4
    d. 2 5 11 6 7 4 9 5 2
    e. None of the above

23. A pre-order traversal of t would visit nodes in the following order…
    a. 2 7 5 2 6 9 5 11 4
    b. 2 7 5 6 11 2 5 4 9
    c. 2 7 2 6 5 11 5 9 4
    d. 2 5 11 6 7 4 9 5 2
    e. None of the above

24. A post-order traversal of t would visit nodes in the following order…
    a. 2 7 5 2 6 9 5 11 4
    b. 2 7 5 6 11 2 5 4 9
    c. 2 7 2 6 5 11 5 9 4
    d. 2 5 11 6 7 4 9 5 2
    e. None of the above

25. Which traversal is the most efficient?
    a. Level order
    b. Pre order
    c. Post order
    d. In order
    e. None of the above

[15 marks] Heap Priority Queue

The following code is an implementation of a maximum HeapPriorityQueue. You are to implement the methods commented with: **`// PROVIDE IMPLEMENTATION`**

TIP: Do not just go from memory of your assignment implementation, be sure to consider carefully the method implementation provided to you here.

NOTE: The method `removeHigh()` is intentionally omitted and you **do not** have to provide an implementation for it.

```
public class HeapPriorityQueue implements PriorityQueue
{
	private final static int DEFAULT_SIZE = 10003;

	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();
	}

	private void swapElement ( int pos1, int pos2 )
	{
		Comparable e = storage[pos1];
		storage[pos1] = storage[pos2];
		storage[pos2] = e;
	}

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

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

	//continued on the following page
```

```
	// PROVIDE IMPLEMENTATION [2 marks]
	private int parent ( int pos )
	{
		return pos / 2;

	}

	// PROVIDE IMPLEMENTATION [2 marks]
	private int leftChild ( int pos )
	{
		return pos * 2;

	}

	// PROVIDE IMPLEMENTATION [2 marks]
	private int rightChild ( int pos )
	{
		return pos *2 + 1;

	}

	// PROVIDE IMPLEMENTATION [9 marks]
	private void bubbleUp ( )
	{
		int pos = currentSize;

		while (pos > 1) {
			if (storage[pos].compareTo(storage[parent(pos)]) > 0) {
				/* If higher than parent, we need to swap */
				swapElement ( pos, parent(pos) );
				pos = parent(pos);
			} else {
				return;
			}
		}
	}
}
```

Recursive Solution:

```
private void bubbleUp() {
	bubbleUp(currentSize);
}

private void bubbleUp(int pos) {
	if (pos <= 1) {
		return;
	}

	if (storage[pos].compareTo(storage[parent(pos)]) > 0) {
		/* If higher than parent, we need to swap */
		swapElement ( pos, parent(pos) );
		bubbleUp(parent(pos));
	}
}
```

[20 marks] Binary Search Trees

The following code is an implementation of a Binary Search Tree. You are to implement the methods commented with: **`// PROVIDE IMPLEMENTATION`**.
NOTE: You are free to add private helper methods to support your implementation but you cannot change the signature of the public methods provided. If you need to use the back of the paper for more space, please indicate so for the marker.

```
public class TreeException extends RuntimeException{
      public TreeException() { }
}// END of TreeException Class

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

      public TreeNode() {
            this.left = null;
            this.right = null;
      }
      public TreeNode(int key, int value) {
            this.key = key;
            this.value = value;
            this.left = null;
            this.right = null;
      }
      public int getKey() {
            return this.key;
      }
      public void setKey(int key) {
            this.key = key;
      }
      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;
      private int count;

      public BinarySearchTree() {
            this.root = null;
            count = 0;
      }
      // This method is left blank intentionally
      // You can assume it follows the expected behavior of a Binary
      // Search Tree and increments the count with each insertion.
      // You do NOT have to provide the implementation
      public void insert(int k, int v) { .... }
```

//continued on the following page…

```
	// PROVIDE IMPLEMENTATION [10 marks]
	// PURPOSE:
	//    Finds the node with the minimum key and returns
	//    the value in that node.
	//
	// THROWS:
	//    TreeException if the tree is empty
	//
	// EXAMPLE:
	//    if t contains nodes with
	//    key:value pairs { 10:4, 20:7, 15:5, 25:6}
	//    getValOfMinKey returns 4
	//
	public int getValOfMinKey() throws TreeException
	{
		if (root == null) {
			throw new TreeException();
		}
		TreeNode cur = root;
		while(cur.getLeft() != null) {
			cur = cur.getLeft();
		}
		return cur.getValue();

	}

	// PROVIDE IMPLEMENTATION [10 marks]
	// PURPOSE:
	//    Calculates and returns the average of all
	//    values in the tree.
	//
	// THROWS:
	//    TreeException if the tree is empty
	//
	// EXAMPLE:
	//    if t contains nodes with the following
	//    key:value pairs { 10:2, 20:3, 15:5, 25:6}
	//    calculateAvgValue returns 4
	//
	public int calculateAvgValue() {

		return doSum(root)/count;

	}
	public int doSum(TreeNode t) {

	   if (t == null) {
	      return 0;
	   }

	   return t.getValue() + doSum(t.getLeft()) + doSum(t.getRight());

	}


}//END of BinarySearchTree Class
```

26. **[20 marks] HashTables**

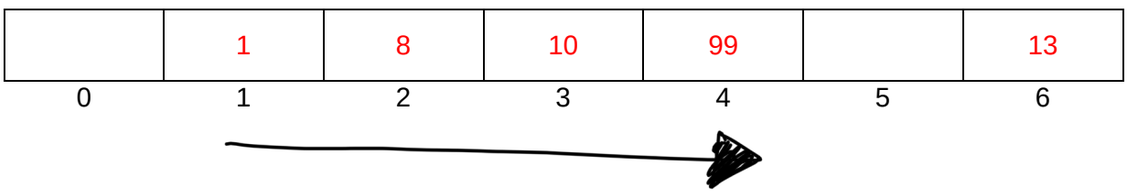
The following code is a Hash Table implementation with portions of code omitted intentionally.

```
public class HashTable {

	private static final int TABLE_SIZE = 7;

	public HashTable() {
		…
	}
	void insert (int key){
		…
	}
	int hash (int value) {
		return value % TABLE_SIZE;
	}
}
```

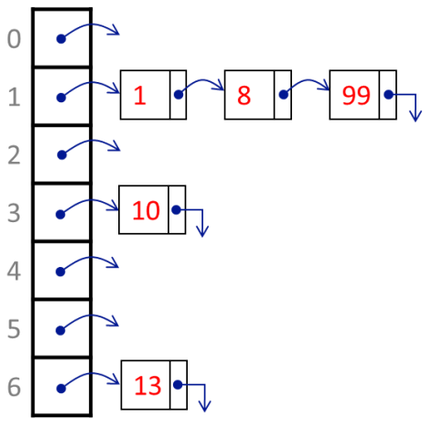
Given the following lines of code…

```
HashTable t = new HashTable();
t.insert(1);
t.insert(8);
t.insert(10);
t.insert(99);
t.insert(13);
```

a) Draw a picture of the hash table that results from the above code if the hash table is implemented using open addressing and linear probing. [4 marks]



b) Draw a picture of the hash table that results from the above code if the hash table is implemented using separate chaining. [4 marks]



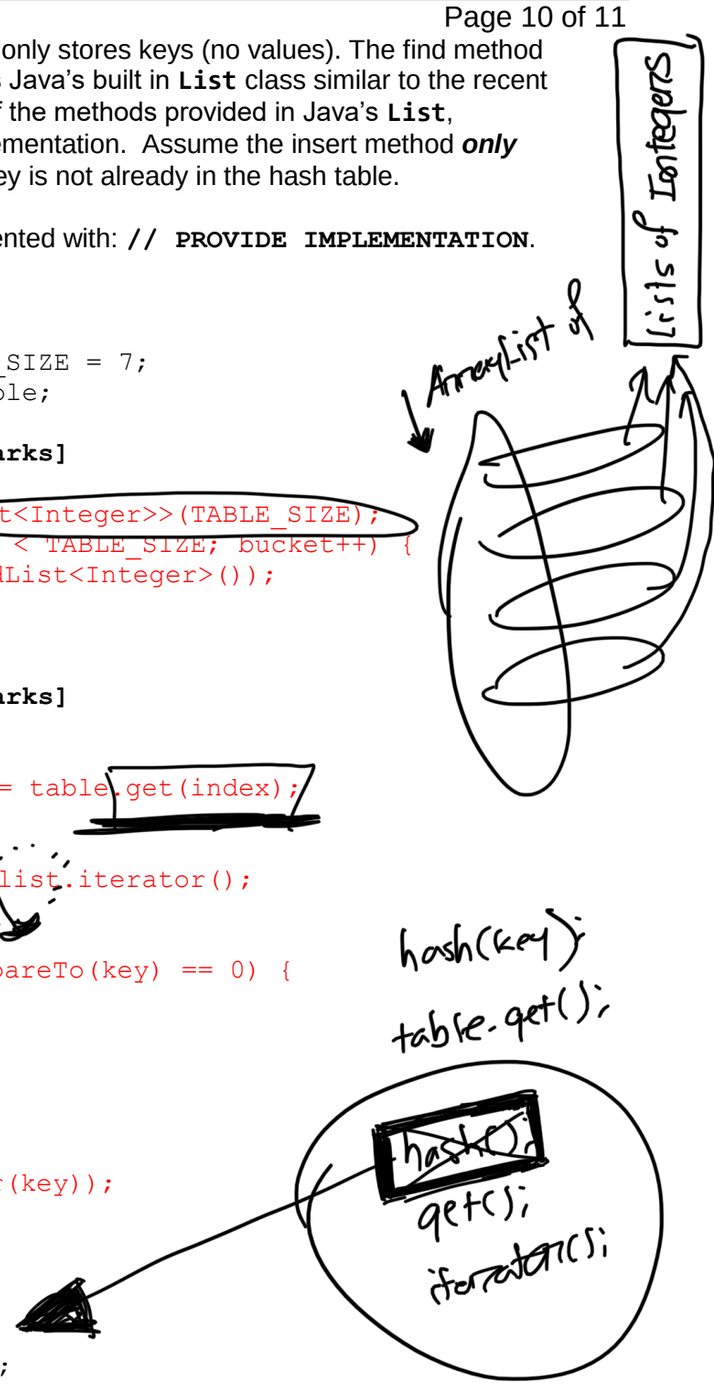
c) Insert and find are expected to be constant time operations (*O(1)*).
What condition must hold to get this expected constant-time performance? [2 marks]

no collisions

d) The following implementation of a Hash Table only stores keys (no values). The find method is omitted intentionally. This implementation uses Java's built in **`List`** class similar to the recent labs and assignments. You are free to use any of the methods provided in Java's **`List`**, **`ArrayList`**, or **`LinkedList`** classes in your implementation. Assume the insert method ***only*** inserts a new element into the hash table if the key is not already in the hash table.

You are to provide implementation where commented with: **`// PROVIDE IMPLEMENTATION`**.

```
public class HashTable {

	private static final int TABLE_SIZE = 7;
	private List<List<Integer>> table;

	// PROVIDE IMPLEMENTATION [5 marks]
	public HashTable() {
		table = new ArrayList<List<Integer>>(TABLE_SIZE);
		for (int bucket=0; bucket < TABLE_SIZE; bucket++) {
			table.add(new LinkedList<Integer>());
		}
	}

	// PROVIDE IMPLEMENTATION [5 marks]
	void insert (int key) {
		int index = hash(key);
		LinkedList<Integer> list = table.get(index);
		boolean found = false;
		Iterator<Integer> iter = list.iterator();
		while (iter.hasNext()) {
			if (iter.next().compareTo(key) == 0) {
				found = true;
				break;
			}
		}

		if (!found) {
			list.add(new Integer(key));
		}
	}


	int hash (int value) {
		return value % TABLE_SIZE;
	}

}
```



THE END