**44-542 Object Oriented Programming Name \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Exam 02 (100 points) KEY** *please print*

1. (10 pts) Consider the tree shown below.



* 1. How many leaves does this tree have? 8
  2. List all the descendants of C: G, H, L, N, O, P
  3. List the ancestors of L:H, C, A
  4. What is the height of this tree? 4
  5. List the siblings of C: B, F

1. (6 pts) Draw the binary search tree that results from inserting these elements into the tree in the order given here: 25, 3, 8, 1, 9, 17, 26

Draw tree here



1. (6 pts) Using the algorithm that we studied in class, remove 60 from the binary search tree shown below.



Draw resulting tree here



1. (12 pts) Assume we have a class **Node** defined as shown here:

**public class Node<E>**

**{**

**E data;**

**Node<E> nextNode;**

**public Node(E data){**

**this.data = data;**

**this.nextNode = null;**

**}**

**@Override**

**public String toString(){**

**return data.toString();**

**}**

**}**

Below is the code for a linked list class. It contains a constructor and one method. The constructor is complete – do not modify the constructor. Add code to implement the method **RemoveFirst**. Note that you should assume the list is non-empty. You do NOT need to check this – it is guaranteed.

**public class ALinkedList<E>**

**{**

**private Node<E> listStart;**

**private int listLength;**

**public ALinkedList(){**

**listStart = null;**

**listLength = 0;**

**}**

**/\*\***

**\* ASSUME THE LIST IS NOT EMPTY.**

**\* Removes the first object from the list.**

**\* @return the node that was removed from the list.**

**\*/**

**public Node<E> removeFirst(){**

**Node<E> temp = listStart;**

**listStart = listStart.nextNode;**

**listLength--;**

**return temp;**

**}**

**}**

1. (6 pts) Show the order in which the nodes are visited in a post order traversal of the tree shown below.

Place answer here: ­­­­­­­­­­­­­­50, 75, 70, 80, 60, 95, 90\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_



1. (6 pts) Show the tree that results from rotating left around 60 in the tree below.

Draw resulting tree here





1. (6 pts) Consider method **mystery** shown below.

**public static String mystery (String str, int n)**

**{**

**if(str.length() <= n)**

**{**

**return "";**

**} else**

**{**

**return str.substring(n).charAt(0) +**

**mystery (str.substring(n+1), n);**

**}**

**}**

**mystery("MERRYCHRISTMAS", 2) = \_\_\_RCIM\_\_\_\_\_\_\_\_\_\_**

1. (10 pts) Assume we have a class **Dog** and we are defining class **DogMap** as shown here. Add the code for methods **getRegNums** and **getDog**. Do not change existing code. Do not write on the back. Your code should easily fit in the space provided.

**public class DogMap**

**{**

**private TreeMap<String, Dog> myDogs;**

**public DogMap()**

**{**

**myDogs = new TreeMap<String, Dog>();**

**}**

**/\*\***

**\* Adds an entry to myDogs.**

**\* @param regNum The registration number, which is the key**

**\* for this mapping.**

**\* @param dog The dog object which is the value for this mapping.**

**\*/**

**public void addToDogMap(String regNum, Dog dog)**

**{**

**myDogs.put(regNum, dog);**

**}**

**/\*\***

**\* Returns the set of registration numbers used in myDogs.**

**\* @return the set of registration numbers used in myDogs.**

**\*/**

**public Set getRegNums()**

**{**

**return myDogs.keySet();**

**}**

**/\*\***

**\* Returns the dog associated with a specified registration**

**\* number.**

**\* @param regNum The registration number**

**\* @return the dog associated with a specified registration**

**\* number.**

**\*/**

**public Dog getDog(String regNum)**

**{**

**return myDogs.get(regNum);**

**}**

**}**

1. (6 pts) Suppose we are using the hash function **h(key) = key MOD 7** to store records with keys **14, 11, 13, 10, 17, 18**, and that we are using ***linear probing*** to resolve collisions.

You must apply **h** to the keys in the order in which they are listed, and store the key in the correct position in the table.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** |
| **14** | **18** |  | **10** | **11** | **17** | **13** |

1. (10 pts) Find the output of the following code segment:

**LinkedList<Integer> myList = new LinkedList<Integer>();**

**myList.add(33);**

**myList.add(45);**

**OUTPUT**

**[88, 33, 45, 55, 63, 73]**

**[33, 13, 55, 2, 73]**

**myList.addFirst(88);**

**myList.addLast(55);**

**myList.add(63);**

**myList.add(73);**

**System.out.println(myList);**

**myList.removeFirst();**

**myList.add(3, 2);**

**myList.set(1, 13);**

**myList.remove(4);**

**System.out.println(myList);**

1. (10 pts) Find the output of the following code segment:

**Stack<Integer> myStack = new Stack<Integer>();**

**myStack.push(99);**

**OUTPUT**

**77**

**77 55 88 99**

**myStack.push(88);**

**myStack.push(55);**

**myStack.push(77);**

**System.out.println(myStack.peek());**

**while(!myStack.isEmpty())**

**{**

**System.out.print(myStack.pop() + " ");**

**}**

1. (12 pts) Assume we have a class **Horse** defined as shown here:

**public class Horse implements Comparable<Horse>**

**{**

**String name;**

**String breed;**

**int yearOfBirth;**

**public Horse(String name, String breed, int yearOfBirth)**

**{**

**this.name = name;**

**this.breed = breed;**

**this.yearOfBirth = yearOfBirth;**

**}**

**public String getBreed()**

**{**

**return breed;**

**}**

**public int getYearOfBirth()**

**{**

**return yearOfBirth;**

**}**

**@Override**

**public String toString()**

**{**

**return name + " " + breed + " " + yearOfBirth;**

**}**

**@Override**

**public int compareTo(Horse otherHorse)**

**{**

**return this.name.compareTo(otherHorse.name);**

**}**

**}**

Assume we have a driver class where we have created an array list of horses, named **myHorses**. Complete the code below to create a comparator that will sort the horses by year of birth. Note that the code you are writing will be part of the driver class.

**Collections.sort(myHorses, new Comparator<Horse>() {**

**@Override**

**public int compare(Horse otherHorse01, Horse otherHorse02)**

**{**

**return otherHorse01.getYearOfBirth()-**

**otherHorse02.getYearOfBirth();**

**}**

**});**