CISC-235*
Test #3
March 13, 2020

Student Number (Required) _____

Name (Optional) _____

This is a closed book test. You may not refer to any resources.

This is a 50 minute test.

Please write your answers in ink. **Pencil answers will be marked, but will not be re-marked under any circumstances.**

The test will be marked out of 50.

Question 1	/12
Question 2	/12
Question 3	/12
Question 4	/12
Question 5	/2
TOTAL	/50

☐ By writing my initials in this box, I authorize the disposal of this test paper if I have not picked it up by April 15, 2020.

"The product is a final hash, and all my books are botches"

~ Herman Melville (author of Moby-Dick)

Question 1 : (12 Marks)

Write an algorithm that returns the second-largest value in a Binary Search Tree of **positive integers**. You may assume that all the values in the tree are different (no duplicate values). If the tree does not contain a second-largest value, your algorithm should return -1.

Your algorithm may be iterative or recursive. The definitions we have used for Binary Search Trees are given on the last page of this test. You may detach that page.

Solution:

Solution 1: Traverse the tree

```
def largest(v):                 # method in the Binary_Search_Tree class
    if v == nil:
        return -1
    elif v.right_child == nil:
        return v.value
    else:
        return largest(v.right_child)

def second_largest():       # method in the Binary_Search_Tree class
    if self.root == nil:
        return -1
    elif self.root.right_child == nil:
        return largest(self.root.left_child)
    else:
        parent = self.root
        current = self.root.right_child
        while current.right_child != nil):
            parent = current
            current = current.right_child
```

```
    if current.left_child == nil:
        return parent.value
    else:
        return largest(current.left_child)
```

Solution 2: use the tree to build a sorted list of values, return 2nd largest

```
def in_order():
    if self.root == nil:
        return empty list
    else:
        return rec_in_order(self.root)

def rec_in_order(current, L):
        t_list = an empty list
        if current.left_child != nil:
            t_list.append(rec_in_order(current.left_child))
        t_list.append(current.value)
        if current.right_child != nil:
            t_list.append(rec_in_order(current.right_child))
        return t_list

def second_largest():
        value_list = in_order(self.root)
        if length(value_list) < 2:
            return -1
        else:
            return second-last element of value_list
```

Other correct solutions are possible.

Students do not have to write out the details of the in-order traversal if they take that approach. I'm satisfied if they say something like "let In_Order(T) be a function that performs an in-order traversal of the tree and returns the values in an array" (or in a list) This is ok because the code for in-order traversal was given in the notes.

So this approach can be satisfactorily expressed as

```
let In_Order(T) be a function that performs an in-order traversal of the
tree and returns the values in a list
L = In_Order(T)
if length(L) < 2:
    return -1
else:
    return the second last element of L
```

Similarly, the other approach can be written as

```
find the largest value in the tree
if it has a left child:
    return the largest value in the left subtree
else if it has a parent:
    return parent.value
else:
    return -1
```

Again, this is ok because the algorithm for finding the largest value in a tree was given in the notes, so it can be referred to here.

Marking:

Any valid solution, expressed with enough clarity to show that it works in all cases	*12*
A valid solution that does not handle an empty tree or a tree with just one vertex	*8*
A solution that goes astray by searching in the wrong place for the second-largest value	*6*
A solution that shows understanding of Binary Search Trees, but no approach to this question	*3*

Question 2 : (12 Marks)

Suppose we need to store a set of n values, where $n \leq 10,000$

In what circumstances would a Red-Black tree be a better choice than an array?

In what circumstances would an array be a better choice than a Red-Black tree?

(Hint: you may want to consider such things as the operations we need to do on the set.)

Solution:

A Red-Black tree would be a better choice than an array if we need to perform search, insert and delete operations on the set after it is defined. These operations are all in O(log n) on RB-trees. For an array, search is in O(n) if the array is unsorted, and insert and delete are both in O(n) if the array is sorted.

An array would be a better choice if only search is required after the set is initially stored. Searching a sorted array takes at most log n steps, whereas searching a RB-tree can take up to 2*log n steps.

An array would also be a better choice if we are implementing a stack containing the values

Marking

6 marks for stating a situation where a RB-tree would be more suitable (there may be other situations than I have suggested).

6 marks for stating a situation where an array would be more suitable (ditto).

3 marks for showing an understanding of the RB-tree structure but not stating a situation where it would be more suitable.

3 marks for showing an understanding of the array structure but not stating a situation where it would be more suitable.

Question 3: (12 Marks)

Here is an insert function for a hash table:

```
insert(k):
   v = h'(k)      # h'(k) is the hashing function
   i = 0
   while (i < m):
      a = (v + 2*i) % m
      if T[a] == 'empty' or T[a] == 'deleted':
         T[a] = k
         break
      else:
         i += 1
   if (i == m):
      report "insert fail"
```

This is simply linear probing using a step size of 2 instead of 1. You may assume that m (the size of the hash table) is odd.

Does this reduce primary clustering, when compared to standard linear probing? (Hint: what do the probe sequences look like?)

Solution:

It does not reduce primary clustering. All the probe sequences are rotations of 0,2,4,6,....,m-1,1,3,5, ...

Marking:

For a correct answer with a valid explanation	*12*
For a correct answer with a poor explanation (such as "There are two probe sequences, one for odds and one for evens")	*9*
For a correct answer with no explanation	*7*
For an incorrect answer that shows an understanding of primary clustering	*5*
For an incorrect answer that shows poor understanding of primary clustering	*3*

Question 4 : (12 Marks)

Suppose we are hashing a set of 10,000 items, each identified by a 9-digit integer key (ie key values in the range [000000000 ... 999999999]) , storing them in a table with $m = 10,500$ addresses

Explain why $h'(k) = "sum\ of\ digits" \pmod m$ with quadratic probing is not a good hashing strategy for this situation.

Solution:

Since the largest sum of digits we can get is 9*9 = 81, all 10,000 keys will hash into the first 82 addresses in the table – over 100 keys on average for each of those addresses. Since we are using quadratic probing, the probe sequence for a key is completely determined by the initial location for that key. Secondary clustering will be a very large problem for this system.

Marking:

Stating that this function only maps keys onto a small subset of the addresses in T, which will cause many collisions (not necessary to use the phrase "secondary clustering"). Note: full marks even if they also state other, less valid points	*12*
Stating that this function only maps keys onto a small subset of the addresses in T, without saying why this is a problem	*9*
Stating that the main problem is secondary clustering (this is true), but not stating the reason why this is particularly bad in this example	*8*
Stating that the main problem is that the probing sequences may not examine all addresses	*7*
Stating that the main problem is something irrelevant, such as 10,000 and 10,500 are not relatively prime	*3*

Question 5: (2 marks)

Tommy "Red" Black coached the Xerox PARC table tennis team from 1963 to 1977. Guibas and Sedgewick named Red-Black Trees in his honour.

True **<u>False</u>**

Solution: False

Marking:

Correct answer (False)	*2*
Incorrect or no answer	*0*