CSE2050 Sample Test - Final

Notes:

- The final may include more questions with different levels of difficulty.
- The final is closed book and closed notes.

Section 1:

- Amy needs to code a card game, where sometimes a card is drawn from the top of the deck, and sometimes a card is drawn from the bottom of the deck. Which of the following data structure is the best to represent the deck?

  a. List

  b. Stack

  c. Mapping(Dictionary)

  d. **Deque**

- For any node in an AVL tree, the height of its children can differ by at most 1. Therefore, when a new node is inserted to an AVL tree, it requires at most one rotation since the tree is already quite balanced. (True / **False**)

- The worst case running time for mergesort and quicksort are both O(n log n), where n is the number of the items to be sorted. (True / **False**)

Section 2:

2.1 The printout for the following code is:

```python
def fun(n):
    if n == 1:
        return n
    if n % 2 == 0:
        print(fun(n//2), end = ' ')
        return fun(n//2)
    else:
        print(fun(n-1), end = ' ')
        return fun(n-1)

fun(5)
```

a. 1 1 1

b. 1 1 1 1

c. 1

d**. 1 1 1 1 1 1 1**

2.2 Fill in the missing code and write down the output of the following code.

```
class Tree:
    def __init__(self, L):
        iterator = iter(L)
        self.data = next(iterator)
        self.children = [Tree(c) for c in iterator]
    def printpreorder(self):
        print(self.data, end = ' ')
        for child in self.children:
            child.printpreorder()
    def printpostorder(self):
        for child in self.children:
            child.printpostorder()
            print(self.data, end = ' ')
    def nonleafnum(self):
      if len(self.children) == 0:
        return 0
      count = 0
      for child in self.children:
        count += child.nonleafnum()
      count += 1
      return count


T = [0, [1, [2], [3]], [4, [5], [6], [7]],[8]]
tree = Tree(T)
print("Number of non-leaf nodes:", tree.nonleafnum())
print("Preorder traversal:")
tree.printpreorder()
print()
print("Postorder traversal:")
tree.printpostorder()
```

Output:

Number of non-leaf nodes: 3

Preorder traversal:

0 1 2 3 4 5 6 7 8

Postorder traversal:

1 1 0 4 4 4 0 0

Section 3:

3.1 Fill in missing code and write down the output of the following code.

```python
class Queue:
    def __init__(self):
        self._head = 0
        self._L = []
    def enqueue(self, item):
        self._L.append(item)
    def dequeue(self):
        item = self._L[self._head]
        self._head += 1
        return item
    def __len__(self):
        return len(self._L) - self._head
    def isempty(self):
        return len(self) == 0
class AdjacencySetGraph:
    def __init__(self, V, E):
        self._V = set()
        self._nbrs = {}
        for v in V: self.addvertex(v)
        for u, v in E: self.addedge(u, v)
    def vertices(self):
        return iter(self._V)
    def edges(self):
        for u in self._V:
            for v in self.nbrs(u):
                yield(u, v)
    def addvertex(self, v):
        self._V.add(v)
        self._nbrs[v] = set()
    def addedge(self, u, v):
        self._nbrs[u].add(v)
    def nbrs(self, v):
        return iter(self._nbrs[v])
    def bfs(self, v):
        tree = {}
        tovisit = Queue()
        tovisit.enqueue((None, v))
        while tovisit:
            a, b = tovisit.dequeue()
            if b not in tree:
                tree[b] = a
                for n in self.nbrs(b):
                    tovisit.enqueue((b, n))
        return tree
    def dfs(self, v):
        tree = {}
        tovisit = [(None, v)]
        while tovisit:
            a, b = tovisit.pop()
            if b not in tree:
                tree[b] = a
                for n in self.nbrs(b):
                    tovisit.append((b, n))
        return tree

V = {'A', 'B', 'C', 'D', 'E', 'F'}
E = {('A', 'B'), ('A', 'C'), ('A', 'D'),
('B', 'A'), ('B', 'C'),
('C', 'B'), ('C', 'A'), ('C', 'D'),
('D', 'C'), ('D', 'A'),
('E', 'F'),
('F', 'E')}
graph = AdjacencySetGraph(V, E)
tree = graph.dfs('A')
print(tree)
for v in V:
    if v not in tree:
        print("The shortest distance between A and ", v, ": inf")
    else:
        count = 0
        u = v
        while u is not 'A':
            count += 1
            u = tree[u]
        print("The shortest distance between A and ", v, ":", count)
```

Output:

```
{'A': None, 'B': 'A', 'C': 'B', 'D': 'C'}

The shortest distance between A and  A : 0

The shortest distance between A and  C : 2

The shortest distance between A and  B : 1

The shortest distance between A and  F : inf

The shortest distance between A and  D : 3

The shortest distance between A and  E : inf
```

3.2 Write down the outputs of the following code. (Note there are print statements in method rotateleft and rotateright.

```
class Entry:
    def __init__(self, key, value):
        self.key = key
        self.value = value
    def __str__(self):
        return str(self.key) + ' : ' + str(self.value)
class Mapping:
    # Child class needs to implement this!
    def get(self, key):
        raise NotImplementedError
    # Child class needs to implement this!
    def put(self, key, value):
        raise NotImplementedError
    #   Child class needs to implement this!
    def __len__(self):
        raise NotImplementedError
    # Child class needs to implement this!
    def _entryiter(self):
        raise NotImplementedError
    def __iter__(self):
        return (e.key for e in self._entryiter())
    def values(self):
        return (e.value for e in self._entryiter())
    def items(self):
        return ((e.key, e.value) for e in self._entryiter())
    def __contains__(self, key):
    #   print(self, "contains", key)
        try:
            return (self.get(key) is not None)
        except KeyError:
            return False
    def __getitem__(self, key):
        return self.get(key)
    def __setitem__(self, key, value):
        self.put(key, value)
    def __str__(self):
        return "{%s}" % (", ".join([str(e) for e in self._entryiter()]))
class BSTNode:
    def __init__(self, key, value):
        self.key = key
        self.value = value
        self.left = None
        self.right = None
        self._length = 1
    def newnode(self, key,value):
        return BSTNode(key, value)
    def get(self, key):
        if key == self.key:
            return self
        elif key < self.key and self.left:
            return self.left.get(key)
        elif key > self.key and self.right:
            return self.right.get(key)
        else:
            raise KeyError
    def put(self, key, value):
        #print("Put in BSTNode", self.key, self.value, key, value)
        if key == self.key:
         self.value = value
        elif key < self.key:
          if self.left:
              self.left = self.left.put(key, value)
          else:
              self.left = self.newnode(key, value)
        elif key > self.key:
            if self.right:
                self.right = self.right.put(key, value)
            else:
                self.right = self.newnode(key, value)
        self.updatelength()
        return self
    def updatelength(self):
        len_left = len(self.left) if self.left else 0
        len_right = len(self.right) if self.right else 0
        self._length = 1 + len_left + len_right
    def floor(self, key):
        if key == self.key:
            return self
        elif key < self.key:
            return self.left.floor(key) if self.left else None
        elif key > self.key:
            return (self.right.floor(key) or self) if self.right else self
```

```python
def rotateright(self):
    print("rotateright", self.key)
    newroot = self.left
    self.left = newroot.right
    newroot.right = self
    self.updatelength()
    newroot.updatelength()
    return newroot
def rotateleft(self):
    print("rotateleft", self.key)
    newroot = self.right
    self.right = newroot.left
    newroot.left = self
    self.updatelength()
    newroot.updatelength()
    return newroot
def maxnode(self):
    return self.right.maxnode() if self.right else self
def _swapwith(self, other):
    ### Swap the key and value of a node.
    ### This operation has the potential to break the BST property.
    ### Use with caution!
    self.key, other.key = other.key, self.key
    self.value, other.value = other.value, self.value
def remove(self, key):
    if key == self.key:
        if self.left is None: return self.right
        if self.right is None: return self.left
        self._swapwith(self.left.maxnode())
        self.left = self.left.remove(key)
    elif key < self.key and self.left:
         self.left = self.left.remove(key)
    elif key > self.key and self.right:
         self.right = self.right.remove(key)
    else:
        raise KeyError
    self.updatelength()
    return self
def __iter__(self):
    if self.left: yield from self.left
    yield Entry(self.key, self.value)
    if self.right: yield from self.right
    def preorder(self):
        yield self.key
        if self.left: yield from self.left.preorder()
        if self.right: yield from self.right.preorder()
    def __len__(self):
        return self._length
    def __str__(self):
        return str(self.key) + " : " + str(self.value)
class BSTMapping(Mapping):
    Node = BSTNode
    def __init__(self):
        self._root = None
    def get(self, key):
        if self._root is None: raise KeyError
        return self._root.get(key).value
    def put(self, key, value):
        #print("Put in BSTMapping", key, value)
        if self._root:
            self._root = self._root.put(key, value)
        else:
            self._root = self.Node(key, value)
    def floor(self, key):
        if self._root:
            floornode = self._root.floor(key)
        if floornode is not None:
            return floornode.key,floornode.value
        return None, None
    def remove(self, key):
        if self._root is None: raise KeyError
        self._root = self._root.remove(key)
    def _entryiter(self):
        if self._root:
            yield from self._root
    def preorder(self):
        if self._root:
            yield from self._root.preorder()
    def __len__(self):
        return len(self._root) if self._root else 0
    def __str__(self):
        return str(list(self.preorder()))
def height(node):
        return node.height if node else -1
```

```
def update(node):
        if node:
            node.updatelength()
            node.updateheight()
class AVLTreeNode(BSTNode):
    def __init__(self, key, value):
        BSTNode.__init__(self, key, value)
        self.updateheight()
    def newnode(self, key, value):
        return AVLTreeNode(key, value)
    def updateheight(self):
        self.height = 1 + max(height(self.left), height(self.right))
    def balance(self):
        return height(self.right) - height(self.left)
    def rebalance(self):
        #print('rebalance', self.key)
        bal = self.balance()
        if bal == -2:
            if self.left.balance() > 0:
                self.left = self.left.rotateleft()
            newroot = self.rotateright()
        elif bal == 2:
            if self.right.balance() < 0:
                self.right = self.right.rotateright()
            newroot = self.rotateleft()
        else:
            return self
        update(newroot.left)
        update(newroot.right)
        update(newroot)
        return newroot
    def put(self, key, value):
        #print("Put in AVLTreeNode:", self.key, self.value, key, value)
        newroot = BSTNode.put(self, key, value)
        #print('newroot:', newroot.key)
        update(newroot)
        return newroot.rebalance()
    def remove(self, key):
        newroot = BSTNode.remove(self, key)
        update(newroot)
        return newroot.rebalance() if newroot else None
class AVLTreeMapping(BSTMapping):
    Node = AVLTreeNode
map1 = BSTMapping()
print("BSTMapping:")
map1['W'] = 1
print(map1)
map1['E'] = 2
print(map1)
map1['I'] = 3
print(map1)
map1['I'] = 4
print(map1)
map1['S'] = 5
print(map1)
map1['G'] = 6
print(map1)
map1['R'] = 7
print(map1)
map1['E'] = 8
print(map1)
map1['A'] = 9
print(map1)
map1['T'] = 10
print(map1)
print("AVLTreeMapping:")
map1 = AVLTreeMapping()
map1['W'] = 1
print(map1)
map1['E'] = 2
print(map1)
map1['I'] = 3
print(map1)
map1['I'] = 4
print(map1)
map1['S'] = 5
print(map1)
map1['G'] = 6
print(map1)
map1['R'] = 7
print(map1)
map1['E'] = 8
print(map1)
map1['A'] = 9
```

```
print(map1)
map1['T'] = 10
print(map1)
print(map1.floor('Z'))
print("Number of nodes:", len(map1))
print("Sum of values:", sum(map1.values()))
```

Output:

```
['W']
['W', 'E']
['W', 'E', 'I']
['W', 'E', 'I']
['W', 'E', 'I', 'S']
['W', 'E', 'I', 'G', 'S']
['W', 'E', 'I', 'G', 'S', 'R']
['W', 'E', 'I', 'G', 'S', 'R']
['W', 'E', 'A', 'I', 'G', 'S', 'R']
['W', 'E', 'A', 'I', 'G', 'S', 'R', 'T']
AVLTreeMapping:
['W']
['W', 'E']
rotateleft E
rotateright W
['I', 'E', 'W']
['I', 'E', 'W']
['I', 'E', 'W', 'S']
['I', 'E', 'G', 'W', 'S']
rotateright W
['I', 'E', 'G', 'S', 'R', 'W']
['I', 'E', 'G', 'S', 'R', 'W']
['I', 'E', 'A', 'G', 'S', 'R', 'W']
['I', 'E', 'A', 'G', 'S', 'R', 'W', 'T']
('W', 1)
Number of nodes: 8
Sum of values: 50
```

3.3

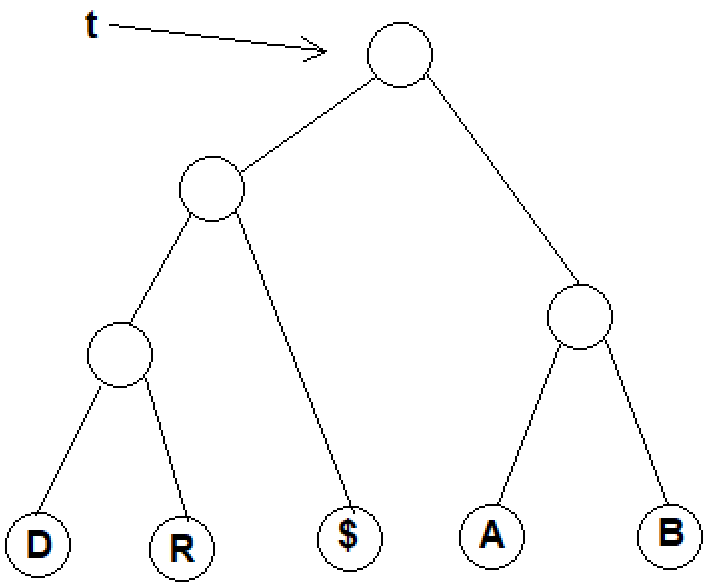
Given the following program:

```
class BinaryTree:
    def __init__(self,data, left = None, right = None):
        self.data = data
        self.left = left
        self.right = right

t1 = BinaryTree('', BinaryTree('D'), BinaryTree('R'))
t1 = BinaryTree('', t1, BinaryTree('$'))
t2 = BinaryTree('', BinaryTree('A'), BinaryTree('B'))
t = BinaryTree('', t1, t2)
print(t.getCode('R'))
```

3.3.1

Draw the binary tree that this program creates



3.3.2

Notice that the tree in 3.3.1 contains empty strings in the internal nodes and one-character string in the leaves. A path from the root node to one of the leaves creates a binary code for the character in the corresponding leaf. The code is a string of zeros and ones that is built by traversing the path from the root to the leaf. Every time we go to the left in the tree we append 0 to the string and every time we go to the right in the tree we append one. For example, the code for the character 'R' should be '001' and for the character 'A' is '10'.

Write a recursive method for the class BinaryTree called getCode. This method receives a character as a parameter and returns the code of that character. After adding your method to the code above and executing that code, the output should be '001'.

Note: This kind of tree is called Huffman tree used in compression algorithms.

```
class BinaryTree:
    def __init__(self,data, left = None, right = None):
        self.data = data
        self.left = left
        self.right = right
    def getCode(self, char):
        if self.left == None and self.right == None:
            if self.data == char:
                return ''
            else:
                return None
        if self.left != None:
            LT = self.left.getCode(char)
            if LT != None:
                return '0' + LT
        if self.right != None:
            RT = self.right.getCode(char)
            if RT != None:
                return '1' + RT
        return None

t1 = BinaryTree('', BinaryTree('D'), BinaryTree('R'))
t1 = BinaryTree('', t1, BinaryTree('$'))
t2 = BinaryTree('', BinaryTree('A'), BinaryTree('B'))
t = BinaryTree('', t1, t2)
print(t.getCode('R'))
```

3.4

The following program builds a general tree. The method **showLevel** prints the tree elements at level k. For example, if k is 0 it will print the root. In this specific program the output will be: p n t n y z because all these elements are at level 2, and the **showLevel** method is executed for k = 2. Complete the **showLevel** method.

```
class Tree:
    def __init__(self, L):
        it = iter(L)
        self.data = next(it)
        self.children = [Tree(c) for c in it]
    def showLevel(self, k):
        if k == 0:
           print(self.data, end='')
           return
        for child in self.children:
           child.showLevel(k-1)


T = ['c', ['a', ['p'], ['n'], ['t']], ['o', ['n', [‘m’]]],['x',
['y'],['z']]]

t = Tree(T)
t.showLevel(2)
```