Assignment-3

(Total Points: 70)

Instructions:

- Download the attached python file assignment_3.py from Blackboard.
- Follow the instructions and **replace** all TODO comments in the scaffolding code.
- Test your code as much as you can to make certain it is correct.
- Run flake8 in addition to testing your code; I expect professional and clear code with minimal flake8 warnings and having McCabe complexity (<10) from all of you.
- Fill in a PDF write up with formatted code and screenshots of your code, output, running the McCabe complexity command and error free console.
- Save the write-up as a PDF and submit it and a zipped file of your code as separate attachments well before the due date on blackboard.

  Note: Running flake 8

  flake8 path/to/your/file (for warnings and errors)

  flake8 --max-complexity 10 path/to/your/file (for complexity)

---

1. Implement Singly Linked List utilizing the scaffolding code. (Points 10)
   a) Your implementation should be able to add an element at the front of the linked list.
   b) It should have the "remove" functionality as well as "contains" functionality.
   c) You should also implement the repr function to display all the added elements in the linked list.

2. Implement a Binary Search Tree Dictionary. (Points 20)
   a) It should be able to insert, delete and retrieve items.
   b) It should display all the keys using pre-order, in order post-order traversals.

3. Implement Chained Hash Table utilizing the scaffolding code. (Points 20)
   a) Your implementation should be able to insert, retrieve and delete items in the hash table.
   b) Your implementation should be tested for different bin counts and load factor.
   c) You should test your implementation with the provided terrible_hash() to show ability for collision resolution.

4. Implement Open Address Hash Table utilizing the scaffolding code. (Points 20)
   a) Your implementation should be able to insert, retrieve and delete items in the hash table.
   b) Your implementation should be tested for different bin counts and load factor.
   c) You should test your implementation with the provided terrible_hash() to show ability for collision resolution.

Source code:

```
from __future__ import division


class SinglyLinkedNode(object):

    def __init__(self, item=None, next_link=None):
        super(SinglyLinkedNode, self).__init__()
        self._item = item
        self._next = next_link

    @property
    def item(self):
        return self._item

    @item.setter
    def item(self, item):
        self._item = item

    @property
    def next(self):
        return self._next

    @next.setter
    def next(self, next):
        self._next = next

    def __repr__(self):
        return str(self.item)


class SinglyLinkedList(object):

    def __init__(self):
        super(SinglyLinkedList, self).__init__()
        self.head = None
        self.currentNode = self.head
        pass

    def __len__(self):
        # return length of the list
        current = self.head
        length = 0
        while current is not None:
            length = length + 1
            current = current.next

        return length

    def __iter__(self):
        self.currentNode = self.head
        return self

    def next(self):
        if self.currentNode is not None:
            temp = self.currentNode
            self.currentNode = self.currentNode.next
            return temp.item
        else:
            raise StopIteration

    def __contains__(self, item):
        # checks list for item is found returns True else False
        current = self.head
        found = False
        while current is not None and not found:
            if current.item == item:
```

```python
                found = True
            else:
                current = current.next

        return found

    def remove(self, item):
        current = self.head
        if current is None or item is None:
            return False
        prev = None
        found = False
        while current is not None and not found:
            if (current.item == item) or \
                    (current.item.key is not None and
                        current.item.key == item):
                if prev is None:
                    self.head = current.next
                else:
                    prev.next = current.next
                found = True
            else:
                prev = current
                current = current.next

        return found

    def prepend(self, item):
        # add item to the front of the list and retrun True if
        # added successfully
        new = SinglyLinkedNode(item)
        new.next = self.head
        self.head = new
        return True

    def __repr__(self):
        s = "List:" + "->".join([str(item) for item in self])
        return s

    def __get__(self, key):
        node = self.head
        while(node is not None):
            if node.item is not None and node.item.key == key:
                return node.item
            node = node.next
        return None

    def __containsKeyValuePair__(self, kvp):
        # checks list for item is found returns True else False
        current = self.head
        found = False

        while current is not None and not found:
            if current.item.key == kvp:
                found = True
            else:
                current = current.next

        return found

# ===============================================================================


class BinaryTreeNode(object):
    def __init__(self, data=None, left=None, right=None, parent=None):
        super(BinaryTreeNode, self).__init__()
        self.data = data
        self.left = left
        self.right = right
        self.parent = parent
```

```python
class KeyValuePair(object):

    def __init__(self, key, value):
        object.__init__(self)
        self.key = key
        self.value = value

    def __repr__(self):
        return repr([self.key, self.value])


class BinarySearchTreeDict(object):

    def __init__(self):
        super(BinarySearchTreeDict, self).__init__()
        self.root = None
        self.size = 0
        pass

    def calcMaxHeightRecursive(self, node):
        if node is None:
            return -1
        else:
            return max(self.calcMaxHeightRecursive(node.left),
                       self.calcMaxHeightRecursive(node.right)) + 1

    @property
    def height(self):
        return self.calcMaxHeightRecursive(self.root)

    def inorder_keys(self):
        # Use the 'yield'  keyword and StopIteration exception
        return self.inorder_traverse(self.root)

    def inorder_traverse(self, tree_node):
        if tree_node.left is not None:
            for i in self.inorder_traverse(tree_node.left):
                yield i

        yield tree_node.data.key

        if tree_node.right is not None:
            for i in self.inorder_traverse(tree_node.right):
                yield i

    def inorder_traverse_pairs(self, tree_node):
        if tree_node.left is not None:
            for i in self.inorder_traverse_pairs(tree_node.left):
                yield i

        yield tree_node.data

        if tree_node.right is not None:
            for i in self.inorder_traverse_pairs(tree_node.right):
                yield i

    def postorder_keys(self):
        return self.postorder_traverse(self.root)

    def postorder_traverse(self, tree_node):

        if tree_node.left is not None:
            for i in self.postorder_traverse(tree_node.left):
                yield i

        if tree_node.right is not None:
            for i in self.postorder_traverse(tree_node.right):
                yield i

        yield tree_node.data.key
```

```python
def preorder_keys(self):
    return self.preorder_traverse(self.root)

def preorder_traverse(self, tree_node):

    yield tree_node.data.key

    if tree_node.left is not None:
        for i in self.preorder_traverse(tree_node.left):
            yield i

    if tree_node.right is not None:
        for i in self.preorder_traverse(tree_node.right):
            yield i

def _items(self):
    # Use 'yield' to return the items (key and value) using
    # an INORDER traversal.
    a = self.inorder_traverse_pairs(self.root)
    r = []
    for x in a:
        r.append([x.key, x.value])
    return r

def __getitem__(self, key):
    # Get the VALUE associated with key
    # return value associated with key in tree dictionary
    if self.tree_root:
        res = self.getValueForKey(key, self.tree_root)
        if res:
            return res.value
        else:
            return None
    else:
        return None
    pass

def getValueForKey(self, key, cur_node):
    if not cur_node:
        return None
    elif cur_node.data.key == key:
        return cur_node
    elif key < cur_node.data.key:
        return self.getValueForKey(key, cur_node.left)
    else:
        return self.getValueForKey(key, cur_node.right)

def __setitem__(self, key, value):
    # return True if set item successfully else False
    if (key is None):
        return None
    keyValuePair = KeyValuePair(key, value)
    if self.root is None:
        self.root = BinaryTreeNode(keyValuePair)
        self.size += 1
        return keyValuePair
    else:
        self.actualInsertLogic(self.root, keyValuePair)
        self.size += 1
        return keyValuePair

def actualInsertLogic(self, tree_node, keyValuePair):
    if (tree_node.data.key == keyValuePair.key):
        tree_node.data.value = keyValuePair.value
    if keyValuePair.key < tree_node.data.key:
        if tree_node.left is not None:
            self.actualInsertLogic(tree_node.left, keyValuePair)
        else:
            node = BinaryTreeNode(keyValuePair)
            tree_node.left = node
```

```
                node.parent = tree_node
        elif keyValuePair.key >= tree_node.data.key:
            if tree_node.right is not None:
                self.actualInsertLogic(tree_node.right, keyValuePair)
            else:
                node = BinaryTreeNode(keyValuePair)
                tree_node.right = node
                node.parent = tree_node
        pass

    def __delitem__(self, key):
        # return True/False
        if key is None:
            return False
        targetNode = self.searchInTree(self.root, key)
        if targetNode is not None:
            if targetNode.left is None:
                self.transplant(targetNode, targetNode.right)
            elif targetNode.right is None:
                self.transplant(targetNode, targetNode.left)
            else:
                treeMin = self.findTreeMin(targetNode.right)
                if treeMin.parent != targetNode:
                    self.transplant(treeMin, treeMin.right)
                    treeMin.right = targetNode.right
                    treeMin.right.parent = treeMin
                self.transplant(targetNode, treeMin)
                treeMin.left = targetNode.left
                treeMin.left.parent = treeMin
            return True
        else:
            return False

    def __contains__(self, key):
        # return True /False
        if self.getValueForKey(key, self.root):
            return True
        else:
            return False

    def __len__(self):
        # returns length of (total number of node in) tree
        return self.size

    @property
    def length(self):
        return self.size

    def display(self):
        # Print the keys using INORDER on one
        # line and PREORDER on the next
        l = []
        s = "Inorder:"
        flag = True
        for i in self.inorder_keys():
            if flag:
                flag = False
            else:
                s = s + "->"
            s = s + str(i)
        l.append(s)
        flag = True
        s = "Preorder:"
        for i in self.preorder_keys():
            if flag:
                flag = False
            else:
                s = s + "->"
            s = s + str(i)
        l.append(s)
        return l
```

```python
    def findTreeMin(self, tree_node):
        if tree_node.left is not None:
            return self.findTreeMin(tree_node.left)
        else:
            return tree_node

    def transplant(self, u, v):
        if u.parent is None:
            self.root = v
        elif u == u.parent.left:
            u.parent.left = v
        else:
            u.parent.right = v
        if v is not None:
            v.parent = u.parent

    def searchInTree(self, node, key):
        if node is None or key == node.data.key:
            return node
        elif key < node.data.key:
            return self.searchInTree(node.left, key)
        else:
            return self.searchInTree(node.right, key)


# ==============================================================================

class ChainedHashDict(object):

    def __init__(self, bin_count=10, max_load=0.7, hashfunc=hash):
        super(ChainedHashDict, self).__init__()
        self.binCount = bin_count
        self.maxLoad = max_load
        self.defaultFunction = hashfunc
        self.backingArr = []
        for i in range(self.binCount):
            self.backingArr.append(SinglyLinkedList())
        self.numOfElements = 0

    def customHash(self, inputNum):
        return self.defaultFunction(inputNum) % self.binCount

    @property
    def load_factor(self):
        return self.numOfElements / self.bin_count

    @property
    def bin_count(self):
        return self.binCount

    def rebuild(self, bincount):
        # Rebuild this hash table with a new bin count
        self.binCount = bincount
        tempbackingArr = []
        for i in range(self.binCount):
            tempbackingArr.append(SinglyLinkedList())
        for l in self.backingArr:
            node = l.head
            while(node is not None):
                hashedKey = self.customHash(node.item.key)
                (tempbackingArr[hashedKey]).prepend(KeyValuePair(
                    node.item.key, node.item.value))
                node = node.next
        for l in self.backingArr:
            del l
        self.backingArr = tempbackingArr

    def __getitem__(self, key):
        # Get the VALUE associated with key
        hashedKey = self.customHash(key)
```

```python
        if self.backingArr[hashedKey].__get__(key) is not None:
            if self.backingArr[hashedKey].__get__(key) is not None:
                return self.backingArr[hashedKey].__get__(key).value
            else:
                return None
        else:
            return None

    def __setitem__(self, key, value):
        if key is None:
            return False
        if self.load_factor > self.maxLoad:
            self.rebuild(self.binCount * 2)
        hashedKey = self.customHash(key)
        if not self.backingArr[hashedKey].__containsKeyValuePair__(key):
            (self.backingArr[hashedKey]).prepend(KeyValuePair(key, value))
            self.numOfElements = self.numOfElements + 1
        else:
            self.backingArr[hashedKey].__get__(key).value = value
        return True

    def __delitem__(self, key):
        if (key is None):
            return False
        hashedKey = self.customHash(key)
        if self.backingArr[hashedKey].__containsKeyValuePair__(key):
            self.numOfElements = self.numOfElements - 1
            self.backingArr[hashedKey].remove(key)
            return True
        else:
            return False

    def __contains__(self, key):
        # return True/ False
        hashedKey = self.customHash(key)
        return self.backingArr[hashedKey].__containsKeyValuePair__(key)

    @property
    def len(self):
        return self.numOfElements

    def __len__(self):
        return self.numOfElements

    def display(self):
        # Return a string showing the table with multiple lines
        # I want the string to show which items are in which bins
        s = ""
        for i in range(self.binCount):
            s = s+str(i)+str(self.backingArr[i])+"\n"
        return s[:len(s)-1]


class OpenAddressHashDict(object):

    def __init__(self, bin_count=10, max_load=0.7, hashfunc=hash):
        super(OpenAddressHashDict, self).__init__()
        self.binCount = bin_count
        self.maxLoad = max_load
        self.hashFunction = hashfunc
        self.backingArr = [None] * self.binCount
        self.numOfElements = 0
        self.delElements = 0
        self.DELETED = -float("inf")

    @property
    def load_factor(self):
        # returns current load factor of dictionary
        return (self.numOfElements + self.delElements) / self.bin_count

    @property
```

```python
def bin_count(self):
    # returns bin count for the dictionary
    return self.binCount

def rebuild(self, bincount):
    # Rebuild this hash table with a new bin count
    self.binCount = bincount
    prevArr = self.backingArr
    self.backingArr = [None] * bincount
    self.numOfElements = 0
    self.delElements = 0
    for prevValue in prevArr:
        if prevValue is not None and prevValue != self.DELETED:
            self.__setitem__(prevValue.key, prevValue.value)

def __getitem__(self, key):
    # Get the VALUE associated with key
    # returns value associated with the key is Dictionary else return None
    for index in range(self.binCount):
        probedKey = self.__linearProbing(key, index)
        element = self.backingArr[probedKey]
        if element is not None:
            if element != self.DELETED and element.key == key:
                return self.backingArr[probedKey].value
        else:
            return None
    return None

def __setitem__(self, key, value):
    if (key is None):
        return False
    if self.load_factor > self.maxLoad:
        self.rebuild(self.binCount * 2)
    for index in range(self.binCount):
        probedKey = self.__linearProbing(key, index)
        probedElem = self.backingArr[probedKey]
        if probedElem is None:
            self.backingArr[probedKey] = KeyValuePair(key, value)
            self.numOfElements = self.numOfElements + 1
            return True
        elif probedElem != self.DELETED and probedElem.key == key:
            self.backingArr[probedKey].value = value
            return True

def __delitem__(self, key):
    # deletes entry for key in dictionary
    # return True if deleted else False
    for index in range(self.binCount):
        probedKey = self.__linearProbing(key, index)
        element = self.backingArr[probedKey]
        if element is not None:
            if element != self.DELETED and element.key == key:
                self.backingArr[probedKey] = self.DELETED
                self.delElements = self.delElements + 1
                return True
        else:
            return False

def __contains__(self, key):
    # return True / False
    for index in range(self.binCount):
        probedKey = self.__linearProbing(key, index)
        probedElem = self.backingArr[probedKey]
        if probedElem is not None and probedElem != self.DELETED:
            if probedElem.key == key:
                return True
    return False

def __len__(self):
    # returns length of (total number of elements in) Dictionary
    return self.numOfElements - self.delElements
```

```python
    @property
    def len(self):
        return self.numOfElements - self.delElements

    def display(self):
        # Return a string showing the table with multiple lines
        # I want the string to show which items are in which bins
        s = ""
        for index in range(self.binCount):
            element = self.backingArr[index]
            if element is not None:
                if element == self.DELETED:
                    s = s+"bin "+str(index)+": " + "[None, None]\n"
                else:
                    s = s+"bin " + str(index) + ": " + element.__repr__()+"\n"
            else:
                s = s+"bin " + str(index) + ": " + "[None, None]\n"
        if len(s) > 0:
            return s[:len(s)-1]
        else:
            return s

    def __linearProbing(self, k, i):
        return (self.hashFunction(k) + i) % self.bin_count


def terrible_hash(bin):
    """A terrible hash function that can be used for testing.

    A hash function should produce unpredictable results,
    but it is useful to see what happens to a hash table when
    you use the worst-possible hash function.  The function
    returned from this factory function will always return
    the same number, regardless of the key.

    :param bin:
        The result of the hash function, regardless of which
        item is used.

    :return:
        A python function that can be passed into the constructor
        of a hash table to use for hashing objects.
    """
    def hashfunc(item):
        return bin
    return hashfunc


def main():
    # Thoroughly test your program and produce useful out.
    #
    # Do at least these kinds of tests:
    #  (1)  Check the boundary conditions (empty containers,
    #       full containers, etc)
    #  (2)  Test your hash tables for terrible hash functions
    #       that map to keys in the middle or ends of your
    #       table
    #  (3)  Check your table on 100s or randomly generated
    #       sets of keys to make sure they function
    #
    #  (4)  Make sure that no keys / items are lost, especially
    #       as a result of deleting another key
    pass


if __name__ == '__main__':
    main()
```

Test Results:

➢ Running the McCabe complexity command and error free console

```
Terminal
C:\Python27>python -m flake8 "C:\Users\Tushar\Desktop\Algo assignment_3\tushar_pandey.py"

C:\Python27>python -m flake8 --max-complexity 10 "C:\Users\Tushar\Desktop\Algo assignment_3\tushar_pandey.py"

C:\Python27>
```

➢ Screenshots of output

```
tester_func2.7.py    tushar_pandey.py
Terminal
C:\Python27>python "C:\Users\Tushar\Desktop\Algo assignment_3\tester_func2.7.py"
For SinglyLinkedList ----------------
passed test case  1
passed test case  2
passed test case  3
passed test case  4
passed test case  5
passed test case  6
passed test case  7
passed test case  8
For BinarySearchTreeDict ----------------
passed test case  9
passed test case  10
passed test case  11
passed test case  12
passed test case  13
passed test case  14
passed test case  15
passed test case  16
passed test case  17
passed test case  18
passed test case  19
passed test case  20
passed test case  21
passed test case  22
passed test case  23
passed test case  24
```

```
For ChainedHashDict ----------------
passed test case  25
passed test case  26
passed test case  27
passed test case  28
passed test case  29
passed test case  30
passed test case  31
passed test case  32
passed test case  33
passed test case  34
passed test case  35
passed test case  36
passed test case  37
For OpenAddressHashDict ----------------
passed test case  38
passed test case  39
passed test case  40
passed test case  41
passed test case  42
passed test case  43
passed test case  44
passed test case  45
passed test case  46
passed test case  47
passed test case  48
passed test case  49
passed test case  50
```