**CS234 Spring 2018 Assignment 3**

Due Friday, July 6\(^{th}\) at 4:00PM

Submit using MarkUs

Coverage: Module 9,10,11

**Programming Component**

*Note:* For all programming questions you must use Python 3.2.3 or higher. Please follow the Python style guide available from the course website.

*Note:* Correctness tests will include some marks for efficiency (that is, they will test with large inputs, causing inefficient solutions to time out)

**Q1. (12 Marks Correctness + 3 Marks Style)**

In this question you must implement a Set ADT using a hash table implementation using open addressing and the linear probe strategy. **Implementations that do not use this strategy will not receive any correctness marks.**

You must use the built-in Python hash function (it’s called `hash`) to compute hashes. You can’t go wrong if you resize when the load factor exceeds 2/3, but the exact choice is up to you (feel free to experiment with different values)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Effects</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__init__(self)</code></td>
<td>Initializes a new (empty) Set</td>
<td></td>
</tr>
<tr>
<td><code>__contains__(self,v)</code></td>
<td>Returns True if v is a member of self</td>
<td>Bool</td>
</tr>
<tr>
<td><code>__len__(self)</code></td>
<td>Returns the number of elements in self</td>
<td>Nat</td>
</tr>
<tr>
<td><code>add(self,v)</code></td>
<td>Adds v to self (if it is not already present)</td>
<td>None</td>
</tr>
<tr>
<td><code>remove(self,v)</code></td>
<td>Remove v from self (raises an exception if self does not contain v)</td>
<td>None</td>
</tr>
<tr>
<td><code>union(self,other)</code></td>
<td>Returns a new set that contains all elements in self or other (or both)</td>
<td>Set</td>
</tr>
<tr>
<td><code>intersection(self,other)</code></td>
<td>Returns a new set that contains all elements that are in both self and other</td>
<td>Set</td>
</tr>
<tr>
<td><code>subtract(self,other)</code></td>
<td>Returns a new set that contains all elements that occur in self, but do not occur in other</td>
<td>Set</td>
</tr>
</tbody>
</table>

Place your implementation of the Set class in `set.py`
Note: The magic methods have been documented. The other methods must be documented by you!
Please use your own words, don’t copy & paste the descriptions from the table!

**Example interaction**

```python
>>> S1 = Set()
>>> S1.add(3)
>>> S1.add(3)
>>> S1.add(4)
>>> S2 = Set()
>>> S2.add(2)
>>> S2.add(4)
>>> U = S1.union(S2)
>>> 2 in U
   True
>>> 3 in U
   True
>>> 4 in U
   True
>>> I = S1.intersection(S2)
>>> 2 in I
   False
>>> 3 in I
   False
>>> 4 in I
   True
>>> len(I)
   1
>>> I.remove(4)
>>> len(I)
   0
```
Q2. (10 marks correctness + 2 marks style)

In this question, you will use a typical Binary Tree implementation, using Nodes with the following 3 fields

```python
class BTNode:
    def __init__(self, item, left, right):
        self.item = item
        self.left = left
        self.right = right
```

Place your answers in `tree.py` (a starter file has been provided. Its node class has an `__repr__` function, just to make it easier on you to visualize your trees)

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Item</td>
<td>The value stored in this node</td>
</tr>
<tr>
<td>Left</td>
<td>A reference to the left child (or None if there is no left child)</td>
</tr>
<tr>
<td>Right</td>
<td>A reference to the right child (or None if there is no right child)</td>
</tr>
</tbody>
</table>

a) Write the Python function `traverse(root)` that consumes a `BTNode` (the root of a binary tree) and returns the in-order traversal. This function is **must not** be recursive. That is to say, it cannot call itself (directly, or indirectly, through mutual recursion). You **must not** use any recursive helper functions, either. *Hint: Use a Stack. You can use the lecture examples, or just use a Python list.* **A recursive solution will receive 0 (zero) correctness marks for part a.**

b) Write the Python function `balanced(root)` that consumes a `BTNode` (the root of a binary tree) and returns `True` if that tree is balanced, and returns `False` if not. For the purposes of this question, balanced means that for all nodes `V`, the height of `V.Left` and the height of `V.Right` are either equal, or differ by at most 1. (Recall that the height of None, the empty tree, is 0).
Examples:

A = None
B = BTNode (1, BTNode (6, None, None), BTNode (3, None, None))
C = BTNode (1, None, BTNode (2, None, BTNode (3, None, None)))

traverse(A) => []
traverse(B) => [6, 1, 3]
balanced(A) => True # vacuous truth: any “for all nodes” statement is true of the empty tree
balanced(C) => False

Written Component
Place your answers in A3.pdf

Although you are not required to, we recommend you use LaTeX to create your documents. Resources and links can be found on the course website.

Q3. (6 marks)
For the following two questions, you must determine time complexities from Question 1 and 2. Use n as the number of values in self, and m as the number of values in other. In all cases you must justify your answer.

a) What is the average case time complexity for your union operation from question 1? Give your answer in terms of n and m, where n is len(self) and m is len(other).

b) What is the worst case time complexity your subtract operation from question 1. Give your answer in terms of n and m, where n is len(self) and m is len(other).

b) What is the worst case time complexity of your traverse function in question 2. Give your answer in terms of n, the number of nodes in the tree.
**Q4. (6 marks)**

Imagine that the Set ADT from Q1 has been implementing using a Binary Search Tree (BST). There exists an algorithm to compute A union B in $O(n + m)$ time, where $n$ and $m$ are the sizes of sets A and B respectively.

Describe this algorithm. You may use a written description, give pseudo-code, or give Python code. You must explain how it works (meaning, you must argue that it is correct). You must also justify the time complexity.

**Q5. (6 marks)**

There is a variant on the open addressing hash table algorithm, known as “Robin Hood hashing”. In this approach, a “cost” is associated with each bucket. When a value $V$ is stored in a bucket, the “cost” of that bucket is set to be the number of failed probes that occurred prior to placing the value $V$. In other words, the “cost” is the value of $i$, the parameter to the probe function $p(i)$.

When probing for an empty bucket, if the algorithm is on probe number $i$, and probes a full bucket that currently has a cost < $i$, then the algorithm will “steal” this bucket for the current value, and then continue probing for a bucket to store the value that had its bucket stolen. This process can repeat many times (the value that lost its bucket can itself steal a bucket).

Example:

<table>
<thead>
<tr>
<th>cost</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>item</td>
<td>0</td>
<td>8</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

The above table uses Linear Probe and the identity hash function. 8 collided with 0 (mod 8) so its cost is 1, because it was placed using the probe function $p(i)$ when $i = 1$. The other values all have cost 0 since they were placed where the hash function indicates.

Using the regular Linear Probe strategy, if we insert 16, (congruent with 0 mod 8) it would end up at index 4, having probed indices 0, 1, 2, and 3.

With Robin Hood hashing, when it reaches index 2, the value of $i$ is 2, but the current cost of bucket 2 is 0. 2 > 0, so Robin Hood will “rob from the rich” (take the 0 cost bucket from the value 2) and “give to the poor” (so that the value 16 will only have a cost of 2, rather than of 4).
The value 2 will then be resume its own linear probe at i = 1. Since 1 > 0, 2 will now steal bucket 3 from the value 3.

<table>
<thead>
<tr>
<th>cost</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>item</td>
<td>0</td>
<td>8</td>
<td>16</td>
<td>2</td>
</tr>
</tbody>
</table>

Finally, the value 3 will resume its own linear probe at i = 1, and find an empty bucket at index 4

<table>
<thead>
<tr>
<th>cost</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>item</td>
<td>0</td>
<td>8</td>
<td>16</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

a) What effect would this strategy have on the average cost of a successful lookup? (In absolute terms, not asymptotically). Explain your reasoning.

b) What effect would this strategy have on the maximum cost of a successful lookup? Again, explain your reasoning.

c) What effect would this strategy have on the maximum cost of an unsuccessful lookup? Explain your reasoning.