[DOCUMENT TITLE]

CS 234

JIADONG MAI
20557203
Assignment 3

Q3

A)

The average case time complexity for my union operation: T(n) = O(max(n, m))

In the union function, I use two loop for putting the elements in self and other to a new set, called new_union, I also use the add operation in the class set, which is O(1) time complexity. Therefore, T(n) should be O(max(n, m))

B)

The worst case time complexity for my subtract operation: T(n) = O(n)

In the subtract operation, I use one loop to search all the elements in self, and use if statement to determining whether the elements show in self not show in the other. I use the add operation in the class set, which is O(1) time complexity. Therefore, T(n) should be 2* O(1) + O(2*n) = O(n)

C)

The worst case time complexity of my traverse function.T(n) = O(n)

In the traverse function, I use one loop to run through all the root, and store all the root and its children in the list of a. all the node in the root only run through one time. Therefore, T(n) should be O(n)

Q4

For question 4, it is similar to the traverse function we did in Q2. Self and other is BST, and has n and m nodes separately. Use the traverse() function we write in Q2.

```
def traverse(root):
  current = root
  a = []
  result = []
  done = 0
  if root ==None:
    return result

  while (not done):
    if current is not None:
      a.append(current)
      current = current.left
    else:
      if(len(a) > 0):
        current = a.pop()
        if current is not None:
          result.append(current.item)
        current = current.right
      else:
        if current is not None:
          result.append(current.item)
        done = 1
  return result
```

the traverse() function I post, the time complexity is O(n), n is equal to the node of the root

```
def union(self, other)
        List_self = traverse(self)              # O(n)
        List_other = traverse(other)            # O(m)
        for I in List_self:                     # O(n)
                if I in List_other:
                        List_self.remove(I)
        List_union = List_self + List_other     # O(n+m)
        return Sort(List_union)                 # O(n+m)
```

Therefore, the union function which use BST for input, the time complexity of this function is O(n+m)

Q5

a)

I suppose there are n number, the sum of this probes would be the sum of the cost + n. in the other words, the cost above each value is mean that the current position subtract original position. When I lookup a number through robin hook hash, I find the original position, if yes, the searching time is 1. If not, and continue to search, the searching times is 1 + cost.

Therefore, the average cost is equal to $\text{Average Cost} = \frac{(\sum_{k=0}^{n} Cost+n)}{n}$.

b)

if I add two more element on the example post on A3: 7 and 10, the new table should look like this:

Cost	0	1	2	1	2	2		0
Item	0	8	16	2	10	3	*	7

The maximum cost of a successful lookup is should be 1 + cost of that value. when the cost is 0, the position of the item is on the original position that p(i) should be. Else, the position of the item is on the original that p(i) plus the cost of that value.

Like in the example, if I search 16, the position that I first search is 16%8, which is 0, the maximum cost of a successful lookup should be 2+1, which is 3

If I search 3, the position is 3%8 is 3, the first search position is 3, which is 24, continue lookup until find the 3. Therefore, the maximum cost of a successful lookup should be 2+1, which is 3

If I search 7, the position is 7%8 is 7, the maximum cost of a successful lookup should be 1

c)

similar to the maximum, if I search a not exist number, like 6. The position is 6%8, which is 6, if find None, then it will stop, the maximum cost is 1

if I search a not exist number, like 4, keep looking until the item is None, then stop. Therefore, the maximum cost is 3.

Therefore, in the robin hood hash, find the item k, find the original position of k first, which is k% size of hash table, in the example, which is 8. Keep checking the whether the item is equal to k, until the item is None. Then stop.