

Work with your neighbor. (This will be graded for participation only.)

1. You are given the following hash function:

$$\text{hash}(\text{key}) = \text{key} \% 7$$

Insert the keys 15, 30, 28, 48 into the hash table below. Use **linear probing** to resolve collisions.

ANS:

0	1	2	3	4	5	6
28	15	30			23	48

- a) Now insert (put) 23 into the hash table. What is the probe sequence for this insertion?

ANS: 2, 1, 0, 6, 5

- b) If you lookup (get) 23 in the hash table, what is the probe sequence?

ANS: 2, 1, 0, 6, 5

*****Wait for lecture to continue before doing the next problems.**

2. The function `char_count(text)` below returns a dictionary containing a mapping of each character in `text` and the count of the number of occurrences in `text` for the character:

```
def char_count(text):
    cdict = {}
    for char in text:
        if char in cdict:
            cdict[char] = cdict[char] + 1
        else:
            cdict[char] = 1
    return cdict
```

Usage:

```
>>> char_count("to be or not to be")
{'t': 3, 'o': 4, ' ': 5, 'b': 2, 'e': 2, 'r': 1, 'n': 1}
```

- a. Write a new version of the function, `char_countv2(text)`, that returns a dictionary that maps each character in `text` to a list of integers, where the length of the list is the number of occurrences of the character in `text`.

Usage:

```
>>> char_countv2("to be or not to be")

{'t': [1, 1, 1], 'o': [1, 1, 1, 1], ' ': [1, 1, 1, 1, 1], 'b': [1, 1],
'e': [1, 1], 'r': [1], 'n': [1]}
```

ANS:

```
def char_countv2(text):
    cdict = {}
    for char in text:
        if char in cdict:
            val_list = cdict[char]    #get()
            val_list.append(1)
        else:
            cdict[char] = [1]         #put()

    return cdict
```

- b. Assume that you have a Dictionary ADT that holds key/value pairs and provides the following operations:
- `put(key, value)`
 - makes an entry for a key/value pair
 - assumes key is not already in the dictionary
 - `get(key)` looks up key in the dictionary
 - returns the value associated with key (and None if not found)
 - defines `__contains__()` so that
 - `in` returns True if a key is in the Dictionary and False otherwise.
 - Note: a Dictionary is of fixed sized and is set to its capacity when created
 - Usage:

```
>>> d = Dictionary(7)
>>> d.put('t', [1])
>>> 't' in d
True
>>> d.put('o', [1])
>>> val_list = d.get('t')
>>> val_list
[1]
```

Modify your function `char_countv2(text)` to use the Dictionary ADT defined above instead of a Python built-in dictionary. Call this version `char_countv3(text)`.

ANS:

```
'''
This version uses the Dictionary ADT.
'''
def char_countv3(text):
    cdict = Dictionary(31)

    for char in text:
        if char in cdict: # this works because of __contains__()
            val_list = cdict.get(char)
            val_list.append(1)
        else:

            cdict.put(char, [1])
```

3. We have 60,000 items to store in a hash table using open addressing with linear probing and we want a load factor of .75.

How big should the hash table be?

ANS: 80,000

Note: You get 80,000 by using the definition of load factor, which is N/M (number of keys / table size). But you would not actually use 80,000; rather you would use the first prime number larger than 80,000.

4. Use the hash function $\text{hash}(\text{key}) = \text{key} \% 7$ and **separate chaining** to insert the keys below into the hash table. On a collision, add to the **front** of the linked list.

15, 30, 16, 8

0	1	2	3	4	5	6
	8	16				
	15	30				