```
! date
```

```
Tue May  3 08:28:06 UTC 2022
```

Please run the above line to refresh the date before your submission.

CSCI-SHU 210 Data Structures

Recitation 12 Maps (Dictionary) and Hash Tables

Part B: Understanding textbook code

```
from collections import MutableMapping     #linke to collection Module: https://docs.python.org/3/library/collections.abc.html

class MapBase(MutableMapping):
    """Our own abstract base class that includes a nonpublic _Item class."""

    #------------------------------- nested _Item class -------------------------------
    class _Item:
        """Lightweight composite to store key-value pairs as map items."""
        __slots__ = '_key', '_value'

        def __init__(self, k, v):
            self._key = k
            self._value = v

        def __eq__(self, other):
            return self._key == other._key     # compare items based on their keys

        def __ne__(self, other):
            return not (self == other)           # opposite of __eq__

        def __lt__(self, other):
            return self._key < other._key       # compare items based on their keys
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:1: DeprecationWarning: Using or importing the ABCs from 'collections' i
  """Entry point for launching an IPython kernel.
```

```
class UnsortedTableMap(MapBase):
    """Map implementation using an unordered list."""

    def __init__(self):
        """Create an empty map."""
        self._table = []                              # list of _Item's

    def __getitem__(self, k):
        """Return value associated with key k (raise KeyError if not found)."""
        for item in self._table:
            if k == item._key:
                return item._value
        raise KeyError('Key Error: ' + repr(k))

    def __setitem__(self, k, v):
        """Assign value v to key k, overwriting existing value if present."""
        for item in self._table:
            if k == item._key:                        # Found a match:
                item._value = v                         # reassign value
                return                                    # and quit
        # did not find match for key
```

```
            self._table.append(self._Item(k,v))

    def __delitem__(self, k):
        """Remove item associated with key k (raise KeyError if not found)."""
        for j in range(len(self._table)):
            if k == self._table[j]._key:                        # Found a match:
                self._table.pop(j)                                # remove item
                return                                              # and quit
        raise KeyError('Key Error: ' + repr(k))

    def __len__(self):
        """Return number of items in the map."""
        return len(self._table)

    def __iter__(self):
        """Generate iteration of the map's keys."""
        for item in self._table:
            yield item._key                                       # yield the KEY
```

```
from random import randrange                 # used to pick MAD parameters
class HashMapBase(MapBase):
    """Abstract base class for map using hash-table with MAD compression.

    ### Note: MAD: Multiply-Add-Divide
    From Mathematical analysis (group theory),
    this compression function will spread integer (more) evenly over the range [0..(N-1)]
    if we use a prime number for p.

    Keys must be hashable and non-None.
    """

    def __init__(self, cap=11, p=109345121):
        """Create an empty hash-table map.

        cap        initial table size (default 11)
        p           positive prime used for MAD (default 109345121)
        """
        self._table = cap * [ None ]
        self._n = 0                                                # number of entries in the map
        self._prime = p                                          # prime for MAD compression
        self._scale = 1 + randrange(p-1)                   # scale from 1 to p-1 for MAD
        self._shift = randrange(p)                          # shift from 0 to p-1 for MAD

    def _hash_function(self, k):
        return (hash(k)*self._scale + self._shift) % self._prime % len(self._table)

    def __len__(self):
        return self._n

    def __getitem__(self, k):
        j = self._hash_function(k)
        return self._bucket_getitem(j, k)                    # may raise KeyError

    def __setitem__(self, k, v):
        j = self._hash_function(k)
        self._bucket_setitem(j, k, v)                       # subroutine maintains self._n
        if self._n > len(self._table) // 2:
            self._resize(2 * len(self._table) - 1)         # number 2*len(self._table) - 1 is often prime

    def __delitem__(self, k):
        j = self._hash_function(k)
        self._bucket_delitem(j, k)                            # may raise KeyError
        self._n -= 1

    def _resize(self, c):
        """Resize bucket array to capacity c and rehash all items."""
        old = list(self.items())            # use iteration to record existing items
        self._table = c * [None]            # then reset table to desired capacity
```

```
        self._n = 0                                    # n recomputed during subsequent adds
        for (k,v) in old:
            self[k] = v                                # reinsert old key-value pair
```

```
class ChainHashMap(HashMapBase):
    """Hash map implemented with separate chaining for collision resolution."""

    def _bucket_getitem(self, j, k):
        bucket = self._table[j]
        if bucket is None:
            raise KeyError('Key Error: ' + repr(k))          # no match found
        return bucket[k]                                      # may raise KeyError

    def _bucket_setitem(self, j, k, v):
        if self._table[j] is None:
            self._table[j] = UnsortedTableMap()          # bucket is new to the table
        oldsize = len(self._table[j])
        self._table[j][k] = v
        if len(self._table[j]) > oldsize:                # key was new to the table
            self._n += 1                                      # increase overall map size

    def _bucket_delitem(self, j, k):
        bucket = self._table[j]
        if bucket is None:
            raise KeyError('Key Error: ' + repr(k))          # no match found
        del bucket[k]                                         # may raise KeyError

    def __iter__(self):
        for bucket in self._table:
            if bucket is not None:                            # a nonempty slot
                for key in bucket:
                    yield key
```

```
class ProbeHashMap(HashMapBase):
    """Hash map implemented with linear probing for collision resolution."""
    _AVAIL = object()          # sentinal marks locations of previous deletions
                               # python object() returns an empty object

    def _is_available(self, j):
        """Return True if index j is available in table."""
        return self._table[j] is None or self._table[j] is ProbeHashMap._AVAIL

    def _find_slot(self, j, k):
        """Search for key k in bucket at index j.

        Return (success, index) tuple, described as follows:
        If match was found, success is True and index denotes its location.
        If no match found, success is False and index denotes first available slot.
        """
        firstAvail = None
        while True:
            if self._is_available(j):
                if firstAvail is None:
                    firstAvail = j                            # mark this as first avail
                if self._table[j] is None:
                    return (False, firstAvail)                # search has failed
            elif k == self._table[j]._key:
                return (True, j)                              # found a match
            j = (j + 1) % len(self._table)                    # keep looking (cyclically)

    def _bucket_getitem(self, j, k):
        found, s = self._find_slot(j, k)
        if not found:
            raise KeyError('Key Error: ' + repr(k))          # no match found
        return self._table[s]._value

    def _bucket_setitem(self, j, k, v):
```

```
            found, s = self._find_slot(j, k)
            if not found:
                self._table[s] = self._Item(k,v)                    # insert new item
                self._n += 1                                         # size has increased
            else:
                self._table[s]._value = v                            # overwrite existing

    def _bucket_delitem(self, j, k):
        found, s = self._find_slot(j, k)
        if not found:
            raise KeyError('Key Error: ' + repr(k))          # no match found
        self._table[s] = ProbeHashMap._AVAIL                 # mark as vacated

    def __iter__(self):
        for j in range(len(self._table)):                    # scan entire table
            if not self._is_available(j):
                yield self._table[j]._key

    def __str__(self):
        result = []
        result.append("[")
        for j in range(len(self._table)):                    # scan entire table
            if not self._is_available(j):
                result.append("("+str(self._table[j]._key) + " " +str(self._table[j]._value) + "), ")
            else:
                result.append("None, ")
        result.append("]")
        return "".join(result)


def main():
    table = ProbeHashMap()
    values = ["Ezreal", "Blizcrank", "Annie", "Teemo", "Zed"]
    for i in range(len(values)):
        table[i] = values[i]   # __setitem__ in HashMapBase


    print(table)


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

```
[None, None, None, (0 Ezreal), (2 Annie), (4 Zed), None, None, None, (1 Blizcrank), (3 Teemo), ]
```

Part C: Modifying textbook code

Task 1. Modify "class HashMapBase(MapBase)". Our HashMapBase class maintains a load factor λ = 0.5. Can you locate this code? Modify load factor λ to 0.66.

```
class HashMapBase(MapBase):
    """Abstract base class for map using hash-table with MAD compression.

    ### Note: MAD: Multiply-Add-Divide
    From Mathematical analysis (group theory),
    this compression function will spread integer (more) evenly over the range [0..(N-1)]
    if we use a prime number for p.

    Keys must be hashable and non-None.
    """

    def __init__(self, cap=11, p=109345121):
        """Create an empty hash-table map.

        cap     initial table size (default 11)
        p       positive prime used for MAD (default 109345121)
        """
        self._table = cap * [ None ]
```

```
        self._n = 0                                             # number of entries in the map
        self._prime = p                                         # prime for MAD compression
        self._scale = 1 + randrange(p-1)                        # scale from 1 to p-1 for MAD
        self._shift = randrange(p)                              # shift from 0 to p-1 for MAD

    def _hash_function(self, k):
        return (hash(k)*self._scale + self._shift) % self._prime % len(self._table)

    def __len__(self):
        return self._n

    def __getitem__(self, k):
        j = self._hash_function(k)
        return self._bucket_getitem(j, k)                       # may raise KeyError

    ## Part C Task 1 ##
    def __setitem__(self, k, v):
        j = self._hash_function(k)
        self._bucket_setitem(j, k, v)                           # subroutine maintains self._n
        if self._n > 2 * len(self._table) // 3:                 # keep load factor <= 0.66
            self._resize(2 * len(self._table) - 1)              # number 2 * len(self._table) - 1 is often prime

    def __delitem__(self, k):
        j = self._hash_function(k)
        self._bucket_delitem(j, k)                              # may raise KeyError
        self._n -= 1

    def _resize(self, c):
        """Resize bucket array to capacity c and rehash all items."""
        old = list(self.items())           # use iteration to record existing items
        self._table = c * [None]           # then reset table to desired capacity
        self._n = 0                        # n recomputed during subsequent adds
        for (k,v) in old:
            self[k] = v                    # reinsert old key-value pair
```

Task 2. Modify "class ProbeHashMap(HashMapBase)". This file currently uses linear probing to handle collisions. After modification, your hashing should become quadratic probing instead of linear probing.

```
class ProbeHashMap(HashMapBase):
    """Hash map implemented with linear probing for collision resolution."""
    _AVAIL = object()          # sentinal marks locations of previous deletions
                               # python object() returns an empty object

    def _is_available(self, j):
        """Return True if index j is available in table."""
        return self._table[j] is None or self._table[j] is ProbeHashMap._AVAIL

    ## Part C Task 2 ##
    def _find_slot(self, j, k):
        """Search for key k in bucket at index j.

        Return (success, index) tuple, described as follows:
        If match was found, success is True and index denotes its location.
        If no match found, success is False and index denotes first available slot.
        """
        firstAvail = None
        i = 1
        start = j
        while True:
            if i > len(self._table):
                raise Exception("quadratic probing inifite looped ")
            if self._is_available(j):
                if firstAvail is None:
                    firstAvail = j                              # mark this as first avail
                if self._table[j] is None:
                    return (False, firstAvail)                  # search has failed
            elif k == self._table[j]._key:
```

```
                return (True, j)                              # found a match
            j = (start + i ** 2) % len(self._table)           # keep looking (cyclically)
            i += 1


    def _bucket_getitem(self, j, k):
        found, s = self._find_slot(j, k)
        if not found:
            raise KeyError('Key Error: ' + repr(k))           # no match found
        return self._table[s]._value

    def _bucket_setitem(self, j, k, v):
        found, s = self._find_slot(j, k)
        if not found:
            self._table[s] = self._Item(k,v)                  # insert new item
            self._n += 1                                      # size has increased
        else:
            self._table[s]._value = v                         # overwrite existing

    def _bucket_delitem(self, j, k):
        found, s = self._find_slot(j, k)
        if not found:
            raise KeyError('Key Error: ' + repr(k))           # no match found
        self._table[s] = ProbeHashMap._AVAIL                  # mark as vacated

    def __iter__(self):
        for j in range(len(self._table)):                     # scan entire table
            if not self._is_available(j):
                yield self._table[j]._key

    def __str__(self):
        result = []
        result.append("[")
        for j in range(len(self._table)):                     # scan entire table
            if not self._is_available(j):
                result.append("("+str(self._table[j]._key) + " " +str(self._table[j]._value) + "), ")
            else:
                result.append("None, ")
        result.append("]")
        return "".join(result)


def main():
    table = ProbeHashMap()
    values = ["Ezreal", "Blizcrank", "Annie", "Teemo", "Zed"]
    for i in range(len(values)):
        table[i] = values[i]   # __setitem__ in HashMapBase


    print(table)


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

```
[None, None, None, (3 Teemo), (0 Ezreal), None, None, (4 Zed), None, (1 Blizcrank), (2 Annie), ]
```

Task 3: Modify "class HashMapBase(MapBase)". In class HashMapBase(MapBase), find out the hash function, what type of compression method is it using? Change the compression method using Division method. Recall Division method: Index = k % N

```
class HashMapBase(MapBase):
    """Abstract base class for map using hash-table with MAD compression.

    ### Note: MAD: Multiply-Add-Divide
    From Mathematical analysis (group theory),
    this compression function will spread integer (more) evenly over the range [0..(N-1)]
    if we use a prime number for p.
```

```
    Keys must be hashable and non-None.
    """

    def __init__(self, cap=11, p=109345121):
        """Create an empty hash-table map.

        cap         initial table size (default 11)
        p           positive prime used for MAD (default 109345121)
        """
        self._table = cap * [ None ]
        self._n = 0                                        # number of entries in the map
        self._prime = p                                    # prime for MAD compression
        self._scale = 1 + randrange(p-1)                   # scale from 1 to p-1 for MAD
        self._shift = randrange(p)                         # shift from 0 to p-1 for MAD

    ## Part C Task 3 ##
    def _hash_function(self, k):
        # Division Method
        return hash(k) % len(self._table)
        ## MAD Method (Multiply, Add, and Divide)
        #return (hash(k)*self._scale + self._shift) % self._prime % len(self._table)

    def __len__(self):
        return self._n

    def __getitem__(self, k):
        j = self._hash_function(k)
        return self._bucket_getitem(j, k)                  # may raise KeyError

    def __setitem__(self, k, v):
        j = self._hash_function(k)
        self._bucket_setitem(j, k, v)                      # subroutine maintains self._n
        if self._n > len(self._table) // 2:
            self._resize(2 * len(self._table) - 1)         # number 2*len(self._table) - 1 is often prime

    def __delitem__(self, k):
        j = self._hash_function(k)
        self._bucket_delitem(j, k)                         # may raise KeyError
        self._n -= 1

    def _resize(self, c):
        """Resize bucket array to capacity c and rehash all items."""
        old = list(self.items())          # use iteration to record existing items
        self._table = c * [None]          # then reset table to desired capacity
        self._n = 0                       # n recomputed during subsequent adds
        for (k,v) in old:
            self[k] = v                   # reinsert old key-value pair
```

Part D: Coding & Problem solving

Task 4: implement function l1_contains_l2(l1, l2) below. This function checks if every element in l2 exists in l1. Use ProbeHashMap(Linear probing HashMap) to solve this problem. Assume all elements in l1, l2 are unique.

```
def l1_contains_l2(l1, l2):
    """ checks if every element in l2 exists in l1.
    :param l1: List[Int] -- the first python list
    :param l2: List[Int] -- the second python list

    return: True if all elements in l2 is exists in l1
    """
    # To do
    # O(len(l1) + len(l2)) == O(N)
    table = ProbeHashMap()
    for key in l1:          # O(len(l1)) expected
      table[key] = "not useful"
```

```
    for key in l2:          # O(len(l2)) expected
        if key not in table:
            return False
    return True

def main():
    l1 = [1,2,3,4,5,6,7,8,9,0]
    l2 = [5,2,8,9,0,1]
    l3 = [5,2,8,9,0,1,"haha"]
    print("l2 should be subset of l1........")
    print("Your result:", l1_contains_l2(l1, l2))
    print("l3 should not be subset of l1........")
    print("Your result:", l1_contains_l2(l1, l3))
main()
```

```
l2 should be subset of l1........
Your result: True
l3 should not be subset of l1........
Your result: False
```

Task 5: Complete the following code. This program counts words in a given text file, then output the most frequent word and its frequency.

```
from google.colab import files
uploaded = files.upload()
table = ProbeHashMap()
file = open("count_words.txt", "r")
# To do
for line in file:
    words = line.split()
    for word in words:
        if word in table:    # Expected O(1)
            table[word] += 1 # Expected O(1)
        else:
            table[word] = 1   # Expected O(1)

max_word = ""
max_count = 0
for key in table:
    if table[key] > max_count:
        max_word = key
        max_count = table[key]



print('The most frequent word is', max_word)
print('Its number of occurrences is', max_count)
```

选择文件 count_words.txt

- **count_words.txt**(text/plain) - 274 bytes, last modified: 2021/8/20 - 100% done

```
Saving count_words.txt to count_words.txt
The most frequent word is interest
Its number of occurrences is 7
```

Given two python lists, create union and intersection lists that contain union and intersection of the elements present in the given lists. Order of elements in output lists doesn’t matter.

Task 6: implement function union(l1, l2). This function returns a new list, that is the union of l1 and l2. Use Python dictionary to reduce runtime. Assume all elements in l1, l2 are unique.

Task 7: implement function intersection(l1, l2). This function returns a new list, that is the intersection of l1 and l2. Use Python dictionary to reduce runtime. Assume all elements in l1, l2 are unique.

```
def union(l1, l2):
    """
        :param l1: List[Int] -- the first python list
        :param l2: List[Int] -- the second python list

        Return a new python list of the union of l1 and l2.
        Order of elements in the output list doesn't matter.
        Use python built in dictionary for this question.

        return: union of l1 and l2, as a SingleLinkedList.
    """
    # To do
    d = {}
    output = l1.copy()
    for key in l1:
        d[key] = "not useful"

    for key in l2:
        if key not in d:
            output.append(key)
    return output


def intersection(l1, l2):
    """
        :param l1: List[Int] -- the first python list
        :param l2: List[Int] -- the second python list

        Return a new python list of the intersection of l1 and l2.
        Order of elements in the output list doesn't matter.
        Use python built in dictionary for this question.

        return: intersection of l1 and l2, as a SingleLinkedList.
    """
    # To do
    d = {}
    output = []
    for key in l1:
        d[key] = "not useful"

    for key in l2:
        if key in d:
            output.append(key)
    return output

def main():
    l1 = [10, 15, 4, 20]

    l2 = [8, 4, 2, 10]


    print(union(l1, l2), "||||should contain [10,15,4,20,8,2], order doesn't matter.")
    print(intersection(l1, l2), "||||should contain [4, 10], order doesn't matter.")

main()
```

```
[10, 15, 4, 20, 8, 2] ||||should contain [10,15,4,20,8,2], order doesn't matter.
[4, 10] ||||should contain [4, 10], order doesn't matter.
```