Data Structures for Problem Solving
Counting Words Using Hashing

Revision : 1.11

1 Problem

Suppose that we want to read a text file, count the number of times each word appears, and provide clients with quick answers to whether or not a word appears and how many times the word appears.

Let's review what data structures might be candidate choices for solving this problem.

One choice would be to use linked lists of (word, count) pairs. As we read through the text we could store a (word, count) pair in a linked list that is kept sorted by word. We could use a linear search to look for each word, and then either insert a new entry if the word is new, or update the count if the word is already in the list. The search for the word's entry is an $O(N)$ operation. After the search operation, inserting or updating an entry could be done in constant ($O(1)$) time.

Another choice would be to use arrays to store the (word, count) pairs. An array has efficient look-up; array access is an $O(1)$ operation independent of the array size or number of elements stored. If we could use a word as the index and keep the array sorted by word, a binary search and update of an existing entry would require $O(log\ N)$ time. However, inserting a new word would require $O(N)$ time to move existing entries. For example, to add a word at the front of the array, all of the (word, count) pairs must be moved forward one location to maintain the sorted order.

We would like to be able to quickly insert new words, update existing counts and search for words. For search, we need a fast way to get from a string, the word to its associated count. For insert or update, we need a fast way to add or change a (word, count) pair. In the general case, we would like to be able to delete entries fast, but deletion is not a capability required for a word counting program.

The type of data structure we want is known as a *map*, which stores a set of (key, value) pairs. For our problem, the (key, value) pairs are (word, count) pairs, where each word in the text is the *unique* key for a dictionary entry, and count is the value that counts the word occurrences.

The Python *dictionary* is an implementation of a *map*. We will study how a Python dictionary works and learn to design our own map data structures.

2 Solution Design and Analysis

2.1 The Python Dictionary (Review)

The key for a Python dictionary must be *hashable*. To be hashable, an element must be encodable into a constant value, typically an integer. The consequence of hashability is

that the element must not be mutable; otherwise, its encoding will not be constant.

The value in the (key, value) pair may be any value. Access, insertion, update and deletion of a value requires use of the key associated with that value.

In Python, we access a dictionary entry using indexing syntax. The key fills the place of an integer index. Entries in a dictionary are not stored in a 'normal' order; When iterating through a dictionary using `for`, the order in which the entries emerge appears random.

This summary of Python dictionary syntax reveals the principal operations of a *map*, plus some Python-specific utility operations.

```
wordCounts = dict()      # Creates a dictionary with no entries
wordCounts[ 'See' ] = 1  # Inserts entry (See, 1)
wordCounts[ 'spot' ] = 1 # Inserts entry (spot, 1)
wordCounts[ 'See' ] = 2  # Updates (See, 1) entry to (See, 2)
wordCounts[ ['b'] ] = 1  # TypeError: unhashable type: 'list'
wordCounts[ 'See' ]      # Accesses value for key See; returns 2
wordCounts[ 'Bad' ]      # Access value for key Bad; raises exception
'spot' in wordCounts     # Queries if key spot exists; returns True
1 in wordCounts          # Queries only searches key values; returns False
del wordCounts['See']    # Deletes the (See,2) entry
del wordCounts['Bad']    # Delete the (Bad,<value>) entry; raises KeyError
wordCounts.keys()        # Returns a sequence containing the keys
wordCounts.values()      # Returns a sequence containing the values
str( wordCounts )        # Returns string "{'See': 2, 'spot': 1}"
```

Python dictionaries use a *hash table* data structure to provide fast access, insertion and deletion. When properly designed, a hash table has an *expected* $O(1)$ access time.

Technically, a key element must have a `__hash__` function to be hashable in Python. The Python built-in function, `hash`, works like `str` and `len`. A call to `hash( obj )` searches for a *method* named `__hash__` in the `obj`'s class. The `__hash__` method returns an integer, which is the encoding of the object instance.

2.2 Hash Table Operations, Pseudo-code and Design Issues/Decisions

Given that Python provides this map data structure based on hash tables, why would we want to know how to implement it ourselves? Here are several reasons:

- We can design hash functions for our own data structures to allow those classes to work as keys in dictionaries.
- We can implement our own version when a language or environment does not provide a map or dictionary data structure.
- We can implement a map to resolve design issues in different ways. These issues involve *space-time trade-offs*; learning how to resolve these will increase our skills.

In essence, **a hash table is an array that we access using strings or other *hashable* objects**. What makes a hash table's design interesting is the way that it converts objects

into array indices. We define a *hash function* to produce a mapping from keys to locations in an array, the hash table storage device. We convert words represented as strings to *hash codes* that can be transformed into indices for the hash table array. Initially the hash table array has a fixed number of locations, all of which are 'empty'. These empty locations might be `None` or some special, sentinel value. Here are the major operations on a hash table:

- `put( table, key, value ) -> NoneType` : Insert or update a (key,value) pair into the hash table. In Python, the insertion operation updates and replaces any value that previously may have been associated with the given key.
- `get( table, key ) -> value` : Access a value by supplying its key to the hash table.
- `contains( table, key ) -> Boolean` : Query whether or not a key exists in the table.
- `delete( table, key ) -> NoneType` : Delete the entry for a given key. (The `delete` operation may also carry the name `remove`.) (Our problem does not require implementing a delete operation.)

The pseudocode calls a *conceptual* `hash_function` to obtain an integer *hash code* that determines the index location of the entry in the hash table.

```
Function contains( hashTable, key )
    hashCode = hash_function( key )
    if there is a Valid entry at hashTable[ hashCode ],
        if the key matches the entry's key,
            return true
        otherwise
            Issue -- has there been a "collision"? this key has
                     hashed to a location that contains another entry.
    otherwise return false

Function get( hashTable, key )
    hashCode = hash_function( key )
    entry = hashTable[ hashCode ]
    if there is a Valid entry at hashTable[ hashCode ],
        if entry's key matches key,
            return entry's value
        otherwise
            Issue -- has there been a "collision"? this key has
                     hashed to a location that contains another entry.
    otherwise
        ERROR -- the requested key is not present

Function put( hashTable, key, value )
    hashCode = hash_function( key )
    if there is no Valid entry at the hashCode index in the hash table,
```

```
        put a new entry( key,value ) at hashTable[ hashCode ]
    otherwise
        entry = hashTable[ hashCode ]
        if entry's key matches key
            update entry's value to value
        otherwise
            Issue -- there has been a "collision" with another entry
```

In the pseudo-code, we see the use of a **hashing function** to scatter the entries throughout the array and the need to **handle collisions**. Designing a hash table data structure involves these activities: designing a hashing function that spreads out the data entries as evenly as possible, and deciding how to handle the collisions that may occur when more than one entry hashes to the same location.

2.3 Designing Hash Functions

2.3.1 Representing String Keys as Natural Numbers

Most hash functions are designed to produce members of the set of the natural numbers $\mathbb{N} = \{0, 1, 2, \ldots\}$, and we need a way to convert our words to natural numbers. As an example, let's just consider words consisting of exactly three lower-case letters. If we use the *ordinal* positions of the letters in the alphabet ('a' is 0, 'z' is 25), we can compute hashes for our 3-letter words as follows using a function we will call `numid`.

String s		$numid(s)$
aaa	$0 \times 26^2 + 0 \times 26^1 + 0 \times 26^0$	$= 0$
aab	$0 \times 26^2 + 0 \times 26^1 + 1 \times 26^0$	$= 1$
...		
baa	$1 \times 26^2 + 0 \times 26^1 + 0 \times 26^0$	$= 676$
bab	$1 \times 26^2 + 0 \times 26^1 + 1 \times 26^0$	$= 677$
bac	$1 \times 26^2 + 0 \times 26^1 + 2 \times 26^0$	$= 678$
...		
caa	$2 \times 26^2 + 0 \times 26^1 + 0 \times 26^0$	$= 1352$
cab	$2 \times 26^2 + 0 \times 26^1 + 1 \times 26^0$	$= 1353$
...		
zzz	$25 \times 26^2 + 25 \times 26^1 + 25 \times 26^0$	$= 17575$

The three-letter key set may be understood as a base-26 numbering system. As shown in the table above, we simply enumerate every possible string of three characters, increasing letter values from the right end of the string:

aaa, aab, aac, aad, . . . , aba, abb, . . . , baa, bab, . . . , zzz

For words of maximum length W, the number of possible lower case words is 26^W. For a length of $W = 10$ characters, there are 141,167,095,653,376 possible identifiers; that's *141 trillion*!

If we use the `numid` hash function with $W = 10$, the hash table needs one array location for every possible lower-case, ten character string to obtain optimal $O(1)$ search performance. However, it is clearly infeasible to allocate a table with trillions of entries, and it does not solve the problem of hashing a string of arbitrary length!

What if we generalize to strings of arbitrary length and choose a sub-sequence of three letters on which to apply our conversion function? Computing a number based on the last few letters would fail because many strings have the same last few letters. Using the first few letters is also a problem because many strings start with the same prefix.

Consequently, many good string hash functions use all the letters. A long string generates a *very* large number, and modular arithmetic limits values to the size of the table.

2.3.2 The Division Method for Hashing

One way to reduce storage needs is to shrink the capacity for the table, apply a function to divide a key's *hash code* by the capacity of the table, and produce the remainder as the word index (location). Using the *modulus* operator, we process the return value of `hash( word )`, the hash code, so that it will lie within the acceptable range of indices for the smaller, hash table array.

For example, given the word 'cab' as a key, the numeric value is 1353. If we create a hash table of capacity 1000, then:

$$h_{div}('cab') = numid('cab') \ \% \ 1000 = 1353 \ \% \ 1000 = 353$$

and the entry for the key 'cab' would be stored at index 353 in the hash table.

2.4 Handling Collisions

The shrunken table in the preceding example works as long as no other key hashes to location 353. If we insert an entry with the key value of 'anp' however, then

$$h_{div}('anp') = numid('anp') \ \% \ 1000 = (0 \times 26^2 + 13 \times 26^1 + 15 \times 26^0) \ \% \ 1000 = 353$$

and we have a collision.

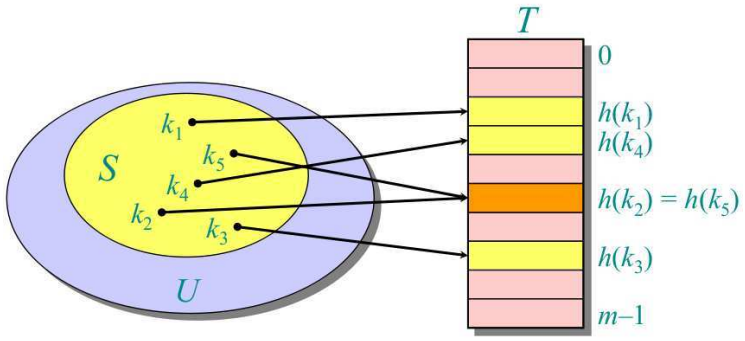

Figure 1: Example hash function $h()$. Here $h(k_i)$ maps key value k_i from the universe of possible keys (U) to an integer representing an index for hash table T of capacity m. The hash table T is a standard array with m elements. In the example, the k_i are unique words. S is the subset of keys from U that have been used to store elements in the hash table. For $h(k_2)$ and $h(k_5)$, the hash function maps a different key to the same location in the hash table, causing a collision. *(This figure was adapted from slides by Erik Demaine and Charles Leiserson.)*

One way of handling the collision issue is to start searching through the array from the *hashed-to, collision point* to the next available table location. This technique is called *open addressing*. Here are some consequences relevant to the choice of *open addressing* for collisions:

- If there are enough collisions in a region of the hash table, those collisions will produce a *cluster* of occupied locations that later queries will have to examine while searching for the desired entry.
- If the number of entries to store exceeds the capacity of the hash table array, a larger table is required. While the array is initially fixed, we can grow the hash table by *rehashing*. (See the **Rehashing** section.)
- A call to `contains` with a non-existent key may take extra time before the function reaches an empty location and is sure the key is not there. The search must cycle back to the start of the array when it reaches the end and stop searching if the search goes all the way through to the starting point of the search.
- If the application deletes entries from the table, a deletion operation must mark a deleted entry location as "available". Later search operations will interpret "available" as "not here; continue searching", and later insert operations will replace the "available" mark with a new entry. (See Figure 2.)

If we implement a hash table with a *sparsely populated* array, we can get good open addressing performance with the trade-off of over-allocated memory.

(We will examine other techniques for handling collisions in the lab.)

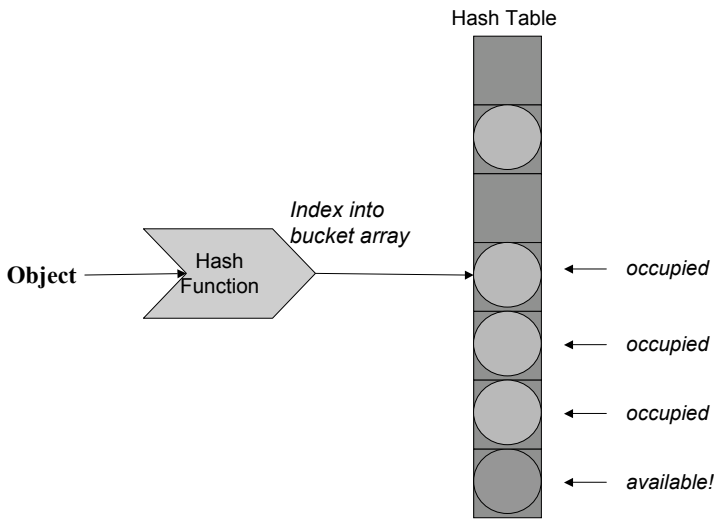

Figure 2: Open Addressing with *linear probing* examines each location in order until finding an empty or "available" location.

2.5 Rehashing

As a hash table with open addressing becomes full, the probability of collisions rises, and hash tables start behaving poorly.

A common solution is to monitor the hash table's *load level*. We calculate the load as the number of keys already in the table divided by the table's total capacity (i.e. number of locations). When the load exceeds a certain threshold, the hash table data structure must perform a `rehash` operation to enlarge the size of the hash table and repopulate a new array with a *rehashing* of the existing entries.

```
Function rehash( hashtable )

    allocate a larger array to use as new hash table storage
    for each entry in the original hashtable
        # put() uses new hash values based on a larger hash table capacity
        put( new array, entry key, entry value )
    change the hashtable's array to be the new, larger array
```

The `rehash` operation is an $O(N)$, linear time operation on the size of the table, because the loop executes once for each entry in the original table.

2.6 Implementation

The `pycount.py` program uses the built-in Python dictionary to store the counts of occurrences for each word. The program prompts for a text file, reads all the words, and allows the user to make queries.

The `word_count.py` program uses an open-addressing hash table module, found in `hashtable.py`, to count word occurrences. Rather than defining a custom hash function for strings, the hashtable module uses the `hash` defined by the `str` class.

3 Testing

We should test our `numid` function using various strings, e.g. using those shown in the table above ('aaa,' 'zzz' and various others), checking that word counts are produced properly at the different string positions.

We then need to test the `contains`, `get` and `put` functions. When putting new words and existing words, check that when existing words are 'put' that their values are updated appropriately. The `contains` function should return true only if we provide a key that has already been put in the table. For `get`, we need to check that existing words have their count correctly returned.

Finally, we would need to check our implementation for `countWords`, using one or more small test files such as `default.txt`. This file includes words that appear once and others that appear more than once.

To test the handling of collisions, we need to generate keys whose hashcodes are the same. Unfortunately, that may be difficult to find keys whose `hash_function` produce the same hashcode.

Testing is easiest using small table capacities to start, as done in the code provided. A function that prints the contents of these tables is helpful for debugging.