Computational Problem Solving CSCI-603
Chained Hash Tables Lab 7

07/26/2021

1 Introduction

In this lab, you will be revising the hash table implementation of a map to use chainning for handling collisions. You will also measure hashing performance with different hash functions.

1.1 Introduction to Chained Hashing

Chaining is an alternative to open addressing for handling collisions. Chaining handles collisions by using a hash table array that contains a list at each location. Each unique key that hashes to the same location is simply added to that list. See Figure 1 for an example.

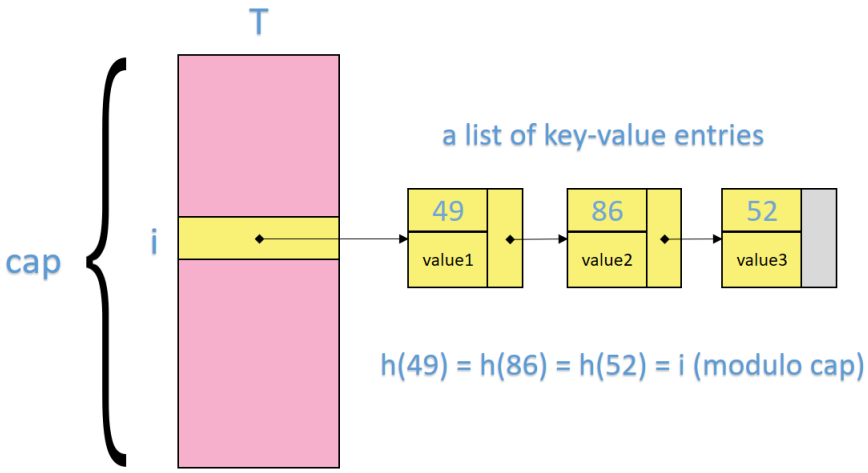

Figure 1: The use of *chaining* to resolve collisions. Here the integer keys 49, 86, and 52 collide because they all map to the same index i (location i in hash table T). To locate the record associated with key 52, a linear search will be needed after the hash function is used to find entry i. Note: the values associated with the keys are not displayed. *(This figure was adapted from slides by Erik Demaine and Charles Leiserson.)*

1.2 Measuring Fitness

In both open addressing and chaining, collisions degrade performance. Many keys mapping to the same location in a hash table result in a linear search. Clearly, good hash functions should minimize the number of collisions. How might the "goodness" or "badness" of a given hash function be measured by looking at the hash table after it has loaded its entries?

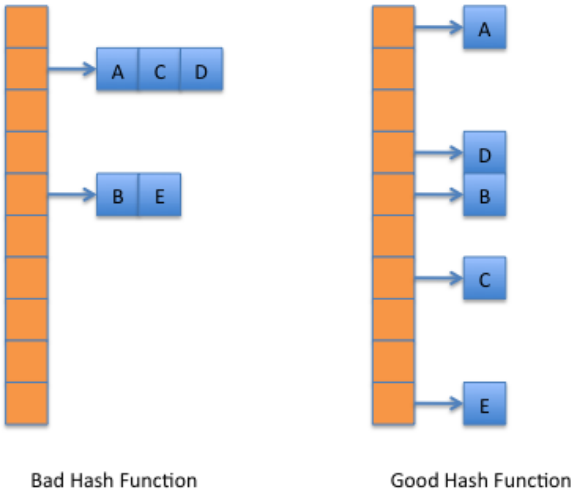

Figure 2: Two hypothetical hash functions have been applied to the same hash table array and the same sequence of keys put in the table.

The example in Figure 2 shows a program which entered 5 keys – A, B, C, D, and E – into a table of length 10 using two different hash functions. The second function provides better distribution, but how can you quantify that based on what you can measure in the tables?

2 Problem-Solving Session

1. Assume a chaining hash table of size 12 and an integer-to-string conversion function that simply adds their ordinal letter values together. Draw what your hash table would look like after putting the following `(key, value)` pairs into it. As an example of the encoding, here is how the first key converts to a number.

   'l','a','d' $\rightarrow 11 + 0 + 3 = 14$

   (a) ("lad", "English")
   (b) ("but", "English")
   (c) ("is", "Latin")
   (d) ("chin", "Dutch")
   (e) ("be", "Greek")
   (f) ("fun", "English")
   (g) ("blab", "German")
   (h) ("zoo", "Greek")

2. Show the order the entries `(key, value)` would be displayed if following the chains from top to bottom and left to right.

3. Write code that implements a hash function that sums up the ASCII values of the characters obtained using `ord` scaled by 31 to the power of the index at which that character occurs in the string, e.g.:

   'l','a','d' $\rightarrow$ `ord('l') + ord('a')*31 + ord('d')*31*31`.

4. Figure 1 illustrates the data structure design you will follow for implementing the chained hash table. Following this design, write pseudo-code for the `add(key,value)` and `remove(key)` operations.

3 Implementation and Answers to Questions

NOTE: Several questions are posed throughout the rest of this document. Enter your answers into a file called `answers.txt` *and submit it with your code.*

3.1 Chaining

The documentation for the module `hashmap.py` you will write is provided to you along with this document. Your data structure design is required to look like the diagram in Figure 1. That is, you build a chained hash table that contains a linked list at each location. Each unique key that hashes to the same location is simply added to that list. You may use the course version of a hash map for reference, but for the best learning experience, you should code your hash map class from scratch.

Note that you have to write an iterator for your hash map to iterate over the entries `(key, value)`. The file **sample_iter.py** contains examples to show you how to do it. There are many online resources as well.

3.1.1 Design Constraints

Your `add`, `contains`, `get`, and `remove` methods should all run in $O(1)$ time (assuming not much clustering due to excessive collisions). This means no linear searches besides the small chains of entries at specific "bucket" locations in the hash table. (Of course the iterator method is linear.)

You may not build, or use from the Python library, additional data structures beyond the list needed for the basic hash table implementation. For example, this means no parallel arrays, dictionaries or separate linked lists.

3.1.2 Testing

Thorough testing is critical for any data structure development, as there will be special cases that your code will have to consider. A significant portion of your grade will be given for a good test suite.

You must construct at least 3 additional non-trivial test cases and add them to the provided **tests.py** file. Each test must be distinct from the others in terms of what it demonstrates. There is the beginning of a test program file available for you to use for adding test functions. This file should be thoroughly commented to explain what each test is testing.

You have also been provided with a program called `word_count.py` which counts the number of times each different word appears in a given file. Make sure your hash map works with this program. You can run it using the provided files `mb.txt` and `atotc.txt` as inputs.

3.2 Comparing Hash Functions

This section involves measuring the original hashing, in Problem Solving question 1, the improved version in Problem Solving question 3, and answering 2 questions. To make the

measurements, we need to prevent the hash table from resizing. **Make sure to comment out the rehashing code from the `add` operation before answering the questions below**.

Add to your hash map a function named `imbalance` to test how well a hash function works. The function must compute the average length of all non-empty chains and then subtract 1. A perfectly balanced, N-entry table would have an imbalance number of 0; the worst case imbalance value would be $N - 1$, where N is the number of elements currently in the hash map.

- **Question 1**: What is the imbalance measurement value for the original *string* hash function from the question 1 of the problem solving session that simply adds the values together, when you run `word_count.py` using your hash map on the files `mb.txt` and `atotc.txt` and for each file use the table sizes of 100 and 1000?

Rerun the program so that it computes the hashcode of a string using the improved formula. You can simply provide the hash table with a different hash function when it is created. Recall that the formula for a string `s` is given by:

$$\sum_{\texttt{i=0}}^{\texttt{len(s)-1}} \texttt{ord(s[i])*31}^{i} = \texttt{ord(s[0])+ord(s[1])*31+...+ord(s[len(s)-1])*31}^{\texttt{len(s)-1}}$$

- **Question 2**: What is the imbalance value for the *improved* `hash_function` given above (i.e. the function from the PSS question 3)? Again, use table sizes of 100 and 1000.

Using any built-in hash function is forbidden.

4 Grading

Your grade will be determined as follows:

- 20%: results of problem solving
- 10%: Design
- 45%: Functionality
  - 5% each: `init, contains, get, iterator`
  - 10% each: `add, remove`
  - 5%: `imbalance`
- 15%: Testing
- 5%: Answer Questions.
- 5%: Code Style and Documentation

5 Submission Instructions

Create a ZIP file named lab7.zip containing the files `hashtable.py`, `tests.py`, and `answers.txt` to the two questions. Submit the ZIP file to the MyCourses assignment before the due date (if you submit another format, such as 7-Zip, WinRAR, or Tar, you will not receive credit for this lab). Do not alter other files to get it to work (i.e. `word_count.py`) as your code will be tested using the original files.