Kramer Canfield
Professor Adam A. Smith
CS 361: Algorithms and Data Structures
18 May 2014

# Separate Chaining, Linear Probing, and Double Hashing Hash Tables (in Java)

## Overview

In a linear probing hash table, key-value pairs are effectively stored in one large array (or possibly another data structure) rather than multiple small sequential symbol tables as is the case with a separate chaining hash table. When putting a pair into the table, the hash code is generated for the key and the modulo operator is used to find a value between 0 and the length of the table. This result is the expected location of the pair in the table. For example, the key "banana" has a hash code of -1396355227. This result is problematic because the value is negative and negative values are not allowed as indices. Even by taking the absolute value, this number is potentially much larger than the table size. In order to accommodate these characteristics, the modulo operator is used to produce a much smaller number which fits inside the bounds of the table indices. However, collisions can result from this behavior, meaning that two distinct keys will try to be placed in the same spot in the table. When this occurs, whichever key is currently being added, and therefore chronologically second, is moved to the next available index in the table, meaning the next open space to the right* of the location where it is expected to be. If that location is occupied as well, the pair is moved until an index is found that is not occupied or is occupied by the same key as the pair to put in the table. Each move to the next spot is a "step" and how much the index increases each step is the "step-size." For a standard linear probing hash table, the step-size is 1. As the table gets increasingly full, the time it takes to add a new pair increases. This increasing time can be compared to α, the *load factor*. The load factor α, is defined as

$$\alpha = \frac{n}{m} .$$

where n is the number of key-value pairs in the table, and m is the total number of entries in the table. In order to keep the put() operation fast, α must be kept close to 0.5, although different implementations can vary slightly in their critical values of α. When α reaches its critical value, the table is "rehashed," meaning that the table is lengthened, and every key-value pair is put in the larger table as if the hash table is starting over. Because every existing pair must be put in the new table, rehashing is proportional to the number of pairs, and is therefore a linear operation.

In order to try to prevent clusters from forming, there is a variant of linear probing called double hashing. The main difference between the two is that double hashing calculates a step-size for each key with

$$s = (hashcode \ \% \ p) + 1$$

where s is the resulting step size and p is a small prime number that is relatively prime to the length of the table. The rest of the double hashing hash table is almost identical to the linear probing hash table, although get(key) and delete(key) are made slightly more complicated.

## Speed Tests

After implementing both a linear probing and a double hashing hash table, as well as bringing in a previously implemented separate chaining hash table, I generated a test for adding many pairs to each hash table. The keys were of type String and the values were of type Integer; the application of the hash table was to count word frequencies in a given text. The text used in the first tests was *Alice's Adventures In Wonderland* by Lewis Carroll[1]. The times were recorded in milliseconds for how long it took to add all of the words in the text to each hash table.

Results (`alice.txt`):

| Run | Separate chaining | Linear Probing | Double Hashing |
|---|---|---|---|
| **1** | 96 | 93 | 85 |
| **2** | 85 | 93 | 85 |
| **3** | 84 | 97 | 93 |
| **4** | 93 | 87 | 91 |
| **5** | 85 | 88 | 87 |
| **6** | 90 | 91 | 93 |
| **7** | 81 | 96 | 89 |
| **8** | 81 | 89 | 85 |
| **9** | 87 | 82 | 113 |
| **10** | 83 | 81 | 104 |
| **Mean Avg** | 86.5 | 89.7 | 92.5 |

As you can see from the results, the separate chaining hash table was faster than the linear probing hash table for adding by roughly 3 milliseconds. This may be explained by the lack of a need in separate chaining for rehashing. The linear probing hash table was faster than the double hashing hash table by roughly 3 milliseconds as well. There is a slight delay in double hashing due to calculating hash codes whenever an available index has to be found or the actual index of a key-value pair. Generally speaking for the put() method, all three hash tables are very close in the time to run, but perhaps with much larger data, these differences in time would change. To explore this, a larger data set was used, *A Tale Of Two Cities: A Story Of The French Revolution* by Charles Dickens[2]. This data set has almost ten thousand distinct words in it, rather than just over 2500 distinct words in *Alice's Adventures In Wonderland.* When running these tests, the separate chaining hash table had an average time of 238.5 ms, the linear probing hash table had an average time of 227.5 ms, and the double hashing hash table had an average time of 219.6 ms. It appears that with a larger data set, the double hashing hash table is now faster than the standard

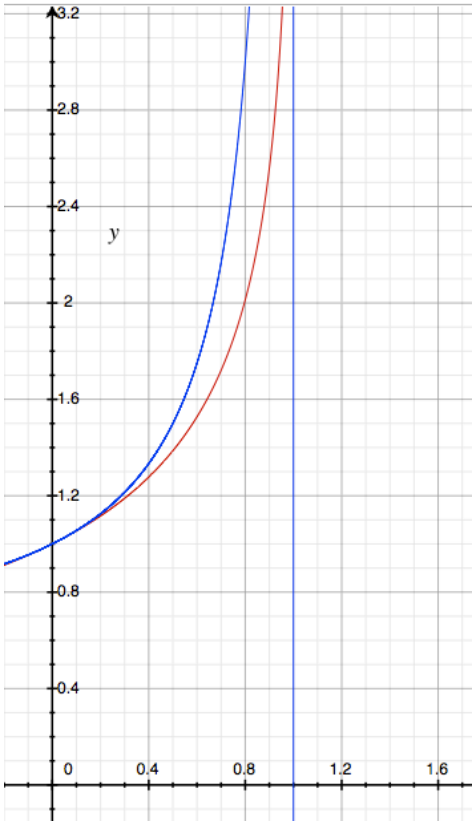
linear probing hash table (which is now faster than separate chaining). The differences in time have now reversed direction.

## Space and Caching

Linear probing and double hashing hash tables are much better for memory usage. With a large array (or multiple large arrays), keys and values are stored in blocks of contiguous memory, rather than having lists with pointers to different chunks of memory as is used with a separate chaining hash table. Arrays are also smaller data structures and they are much faster and easier to access and manipulate than ArrayLists or LinkedLists, which may be used in a separate chaining hash table. Lists can be slow and cumbersome because the standard get() operation is linear in time, so access is slower as well as more complicated due to the pointers needed for the data structure. When the benefits of List-type data structures, such as size flexibility, are not needed, the smaller size, very fast access, and caching benefits of arrays are favorable for most programs.

## Implementation Details

When calculating a step-size for any key, it is important to add 1 to the result after the hash-and-mod-by-a-small-prime because some keys have hash codes that would otherwise result in a step-size of zero. A step-size of zero would result in an infinite loop because every step would go nowhere; adding 1 removes this possibility.




With the delete() method, complexity increases in linear probing and double hashing. Because the get() method and findActualIndex() helper method rely on clusters for a stopping condition, the key-value pair cannot just be removed or set to null, but instead a placeholder must be used. The key can be null, but the value is set to a special Object called `pairWasDeleted` which marks where pairs used to be. If pairWasDeleted is encountered, rather than mistakenly stopping at the perceived end of a cluster, the hash table continues searching for the desired key-value pair as it should.

The only time that the load factor, α, is used in the non-separate chaining hash tables is in calculating whether or not a rehash is necessary. α is not stored, but calculated each time a check is made. You will notice that alpha has different upper bounds based on the kind of hash table. In linear probing, alpha is kept near 0.5 for optimization (although it could reach near 0.75 in usual practice), but in double hashing, alpha can approach a value much closer to 1.0. The reason for this difference is the varying step size and result of reduced clusters. With reduced clusters, the clusters that do form are smaller and so they are faster to traverse. If you look at the graph on the right, alpha is on the

x axis and a number proportional to the number of operations for a successful hit is on the y axis. Linear probing is shown in blue and double hashing is shown in red. Observe how for any alpha value larger than approximately 0.2, there is a difference in the number of operations; with double hashing, the alpha value can be larger while maintaining the same number of operations. According to Donald Knuth, a hit for standard linear probing (shown in blue) will have time complexity

$$\sim\frac{1}{2}\left(1+\left(\frac{1}{1-\alpha}\right)\right).$$

A hit for double hashing (shown in red) will have time complexity of roughly

$$\left(\frac{1}{\alpha}\right)\left(\ln\left(\frac{1}{1-\alpha}\right)\right)$$

(according to Steve Wolfman at the University of Washington)[3]. With the logarithm included in the second expression, it is easy to see that the number of operations will be greatly reduced.

## Conclusions

Although slightly slower or comparable in speed to separate chaining hash tables for smaller data sets, linear probing and double hashing hash tables can have great benefits in terms of memory and caching and great speed improvements for larger data sets. The need to rehash is less urgent with double hashing, further reducing time spent trying to maintain optimization. Although sometimes a headache to understand at first, it is clear that linear probing style hash tables carry great benefits. Double hashing hash tables do need to stop and calculate separate step sizes for each key, but the benefit of reducing clusters can be worth the added complexity in the end as demonstrated in real tests as well as mathematical theory.

* assuming that “going to the right” means increasing the indices

1 available at http://mathcs.pugetsound.edu/~adamasmith/cs361/alice.txt

2 available at http://mathcs.pugetsound.edu/~adamasmith/cs361/two-cities.txt

3 see lecture slide https://courses.cs.washington.edu/courses/cse326/00wi/handouts/lecture16/sld026.htm and course page https://courses.cs.washington.edu/courses/cse326/00wi/index.html