Bangladesh University of Engineering and Technology

Department of Computer Science and Engineering

Academic Year 2021 - 2022

CSE 208
-Data Structures and Algorithms II Sessional-

Offline No. 5
Study of Collision Resolution Performances for Different Hashing Algorithms

Name: Anwarul Bashir Shuaib

Roll: 1805010

Section: A1

Date of Submission: June 10, 2021

Study of Collision Resolution Performances for Different Hashing Algorithms

Abstract

In this experiment, the famous Polynomial Hashing Algorithm[1] and djb2[2] algorithm were used to compare their collision resolution performances and average number of probes. A separate double hashing and a custom hashing methods were created using an auxiliary hashing algorithm and similar performance tests were done. The test consisted of randomly generating 10000 words and inserting those in the custom hash table, and calculating the number of collisions alongside. After insertion, 1000 words were randomly selected and searched across the tables for calculating average number of probes. Finally all the performance metrics were given in a tabular form to compare the algorithms side by side.

Experimental Setup:

We created a hash table which can accept a custom hash function for string distribution. We used the *Polynomial Hashing Algorithm* and *djb2* algorithm for this purpose.

The polynomial hashing function is computed as follows:

Listing 1: Polynomial Hash Algorithm

```
1 private int polyHash(String k) {
2     int multiplier = 263;
3     long hash = 0;
4     for (int i = 0; i < k.length(); i++){
5         hash = (hash * multiplier + k.charAt(i)) % 1000000007;
6     }
7     return (int) hash % m;
8 };
```

Here, we take the modulo in line 7 to make sure that the hash value doesn't overflow the table length.
The *djb2* hashing algorithm is given as follows:

Listing 2: djb2 Hash Algorithm

```
private int djb2Hash(String k) {
    long hash = 5381;
    for (int i = 0; i < k.length(); i++) {
        hash = (((hash << 5) + hash) + k.charAt(i));
    }
    int retVal = (int) hash % m;
    return retVal < 0 ? (retVal + m) % m : retVal;
};
```

Similarly, we take the hash value modulo length of the table on line 6. Since there's no prime modulo in our main for loop, we have to make sure that our hash value doesn't end up being negative. That's why we made that check in line 7.

We also used a double hashing method and a custom hashing method, defined by our auxiliary hash function:

Listing 3: Auxiliary Hash Algorithm

```java
private int auxHash(String k) {
    int multiplier = 0xDEADBEEF;
    long hash = 0;
    for (int i = 0; i < k.length(); i++) hash = (hash + k.charAt(i) ^
       multiplier) % 1000000009;
    return (int) (hash % m);
}
```

Performance Measurement:

We calculated the number of collisions over 10000 randomly generated 7 character long word insertion. After that, we randomly searched 1000 words across the hash table and calculated average number of probes, i.e, the average number of look-ups needed to find the value of a given key. For a load factor $\alpha = \frac{10000}{10007} = 0.9993$

Hash / Method	Polynomial Hash		djb2 Hash	
	Collisions	Avg probes	Collisions	Avg probes
Separate chaining	3654	1.49	3742	1.47
Double Hashing	62067	7.65	62627	4.79
Custom Probing	62650	5.62	65814	11.78

Table 1: Statistics for table size, m = 10007

Again, for a load factor of $\alpha = \frac{10000}{80309} = 0.125$,

Hash / Method	Polynomial Hash		djb2 Hash	
	Collisions	Avg probes	Collisions	Avg probes
Separate chaining	643	1.07	592	1.07
Double Hashing	735	1.06	664	1.07
Custom Probing	713	1.06	650	1.06

Table 2: Statistics for table size, m = 80309

Conclusion

We conclude our analysis by observing that separate chaining method performed the best out of three methods. The number of collisions were also found to be lower than the other two. This is because in separate chaining, we considered a collision count only when two different strings produced the same hash value. But when probing along the linked lists in each cell, we did not increment our collision counter. Again, the load factor was quite high for double and custom hashing to provide a good distribution of strings. Decreasing the load factor α to .125 dramatically boosted the performances of the last two methods as well as decreased average number of probes.

References

[1] Wikipedia contributors, "Rolling hash — Wikipedia, the free encyclopedia." https://en.wikipedia.org/w/index.php?title=Rolling_hash&oldid=1032375032, 2021. [Online; accessed 18-January-2022].

[2] York University, "Hash functions." http://www.cse.yorku.ca/~oz/hash.html, 2021. [Online; accessed 18-January-2022].