Hashing

Jonatan Emil Simonsen

VGIS8, Aalborg University

1 INTRODUCTION

Hashing is a data structure for implementing dictionaries, where a key corresponds to an identifier in the language. O(n) is the worst case time, which does not differ from searching in a linked list. Performing hashing under reasonable conditions the average time for searching is $\Theta(1)$. The hash table uses directly addressing into an ordinary array, which makes determining the position of an element in an array O(1) time. The number of keys is relatively small compared to the size of the array index, which makes hashing a more effective alternative to directly addressing. The array does not use the key as an index, but computes the array index from the keys Cormen et al. (2009).

The written code used for the experimental results and the results themselves can be found in the GitHub repository: https://github.com/DGJonna97/ADSSEMiniProject.git

2 PSEUDOCODE AND THEORY

Throughout this chapter I will go through the pseudocode and related theory of hash tables, as well as chaining and open addressing for handling collisions with their underlying methods. All the theory and pseudocode in this chapter is taken form chapter 11 of the book Cormen et al. (2009).

2.1 Hash tables

Hash tables solves the problems of direct addressing, since it is not optimal for the cases where the amount of elements are large. Hash tables store less keys in a dictionary in relation to the amount of possible keys, and therefor requires less storage space than direct addressing. The benefit of hash tables is the time, since it takes O(1) like direct addressing for searching. The catch is the that for hash tables it is the average time, but for direct addressing it is the worst-case time.

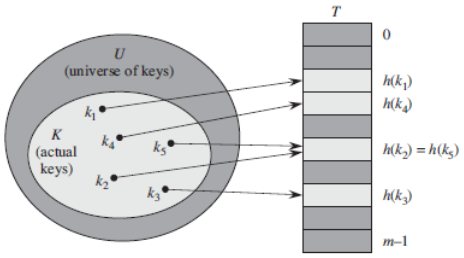

Figure 1. The figure shows how keys are mapped to a hash table.

Hash tables do not map single elements to a key, instead they are stored in h(k) slots, which is related to hash functions, which then decides the placement of the key in the hash table (Further information about the specific hash functions can be found in the sections: 2.2.1, 2.2.2, and 2.2.3). The main problem about hash tables, which can be seen in figure 1, is that multiple hash values can be mapped to the same slot and collide. This can happen since the value of m is less than —U—, which saves storage space, but also risks collisions. This problem can be solved by using the technique chaining.

2.1.1 Chaining

The technique chaining is a method for hash tables for solving the collision problem. When hash values are added to the same table slot, then chaining stores the hash values in the same slot. This is done by

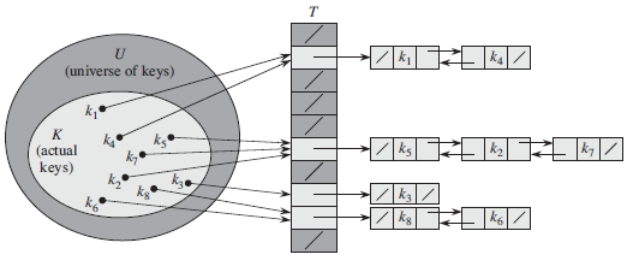

Figure 2. The figure shows how chaining solves the collision problem by adding the hash values to a linked list.

CHAINED-HASH-INSERT (T, x)
1 insert x at the head of list $T[h(x.key)]$

CHAINED-HASH-SEARCH (T, k)
1 search for an element with key k in list $T[h(k)]$

CHAINED-HASH-DELETE (T, x)
1 delete x from the list $T[h(x.key)]$

adding the hash values to the same linked list, so slot j contains a pointer, which points to a list of all the elements stored in the slot j.

Like direct addressing, chaining also have the three operations insert, search and delete, which serves the same purpose, but within the linked lists. The insert operation inserts the element into the assigned hash value, and the element is placed in that linked list. For the search the element is searched for in the linked list with the key k. The delete operation deletes an element from the linked list with the selected hash value.

2.2 Hash functions

What defines a good hash function is the assumption, that each key is equally likely to hash to any of the m slots in the table, and the selected m slot is not dependent on the other key slots. The next sections will describe the three different techniques, which is used to determine the selected slot of hash values.

2.2.1 The division method

The division method is found from the hash function:

$$h(k) = \text{k mod m}$$

In the function a key is mapped to a slot by dividing the current amount of keys by m, which is the length of the hash table. This way of assigning slots is effective since it only takes one operation to assign a slot to a key.

2.2.2 The multiplication method

The multiplication method for assigning a hash value is based on the function:

$$h(k) = \lfloor m\,(kA \bmod 1) \rfloor$$

This method takes the keys and multiply with a constant A, which is a positive integer between zero and one. This number is divided with an integer with the value of one. The next step of the function is to multiply the current value with m and then take the floor of the result, which provides the hash value for the key.

2.2.3 Universal hashing

For Universal hashing it differs from the two previous hash functions, since it relies on randomising. Each key is provided a random hash function, also in cases, where elements have the same key. Since universal hashing uses random hash functions the average running time is not affected like the division and multiplication method, since no single value will evoke the worst case, and slow the search time.

2.3 Open addressing

Open addressing is a technique for handling collisions in hash tables, which is done by assigning a slot for all elements in the hash table. This means that all inserted elements have a slot, and the unassigned slots will contain NIL. When searching for an element in a hash table, which uses open addressing, each table slot will be searched until the desired element is found or it is concluded, that the table does not contain that element. By using the open addressing technique the programmer might experience, that the hash table can fill up, since the load factor cannot exceed 1. The pro of using this technique is the space requirement, since there is no need for storing pointers, like in chaining, and the unused memory space can be used for acquiring more available slots to prevent the hash table to fill up.

Since all elements is assigned their own slot, then we need to search for unused slots in case of collision, where an extra variable is added. This is the probe number, which ranges from zero to m - 1. How this probe value i is being utilised depends on the probing technique. I will comment on the three probing techniques in section 2.3.1, 2.3.2, and 2.3.3.

The pseudocode for hash-insert when using open addressing for collision handling:

```
HASH-INSERT(T, k)
1  i = 0
2  repeat
3      j = h(k, i)
4      if T[j] == NIL
5          T[j] = k
6          return j
7      else i = i + 1
8  until i == m
9  error "hash table overflow"
```

For hash-insert with open addressing the probe value i is set to zero, since we aim for the initial slot position. The selected slot is checked if it is already taken or set to NIL. If the slot is set to NIL, then the key is assigned to the slot. If there is a collision, then we add +1 to the probe value, which then points to a new slot, and then the algorithm will continue checking for available slots until the probe number reaches the length of the table, and an overflow error occurs, since no slots is available for the key.

The pseudocode for hash-search is also different when using open addressing for collision handling:

```
HASH-SEARCH(T, k)
1  i = 0
2  repeat
3      j = h(k, i)
4      if T[j] == k
5          return j
6      i = i + 1
7  until T[j] == NIL or i == m
8  return NIL
```

The principle of searching is similar to inserting, since the initial search of the key is checking the initially assigned slot. If the slot is taken by another key, then we add +1 to the probe number until the

desired key is found. In case that the probe number reaches m or a slot is assigned NIL, then the search function will return NIL.

For deleting values another value have to be inserted instead of NIL, since future hash-search functions might return NIL, because the former occupied slot now is occupied by NIL. This is solved by inserting a DELETE value after deleting a key. This allows the search to add +1 to the probe number and look through the rest of the hash table.

2.3.1 Linear probing

The linear probing method for open addressing uses the hash function:

$$h(k,i) = (h'(k) + \text{i}) \bmod \text{m}$$

This means, that for the hash-insert and hash-search functions, that if a slot is already assigned to another key, then one is added to the hash value and then divided by the length of the hash table to find the next slot, which then is check if it is available. This will continue until i == m. This results in the con of the method, which is primary clustering. This may result in increased average searching time in the cases where there are many occupied slots.

2.3.2 Quadratic probing

The quadratic probing method for open addressing uses the hash function:

$$h(k,i) = (h'(k) + \text{c1}*\text{i} + \text{c2}*i^2) \bmod \text{m}$$

For quadratic probing c1 and c2 are constants, which for my code has been set to c1 = 0 and c2 = 1. In this case the initial hash value determines the probe sequence, which might lead to secondary clustering, and the probe number might still go towards m, but for this to happen keys have to be assigned the same initial hash value.

2.3.3 Double hashing

The double hashing method for open addressing uses the hash function:

$$h(k,i) = (h1(k) + \text{i}*h2(k)) \bmod \text{m}$$

For double hashing the method relies on the key in two ways, since the h2(k) relies on the former h1(k) value, and besides that the h2(k) must be relatively prime to m for the whole hash table to be searched. For my code I have set m to be a prime and used the two following functions for h1(k) and h2(k):

$h1(k) = \text{k} \bmod \text{m}$
$h2(k) = 1 + (\text{k} \bmod \text{m'})$, where m' is m - 9 to make m a prime.

In this case the initial search would be at slot h1(k), if the slot is taken, every h2(k) slot would be checked until an available slot is found or an overflow error is thrown.

3 ANALYSIS

In this analysis I will go through the best and worst case time of the hash table hash-search, hash-insert and hash-delete functions. Besides that I will touch upon the hash-search and hash-insert function in case of collisions. For collisions I will also go through the best and worst case time of chaining and open addressing. All the best and worst cases is based on my code from the GitHub library: https://github.com/DGJonna97/ADSSEMiniProject.git

The hash-search, hash-insert, and hash-delete functions all have the same best and worst case time. This is due to them all containing one for-loop, which in best case has $\Omega(1)$ time, and worst case O(n) time. The best case is where a slot is found at once, and then only one operation is necessary. For the worst case a slot is not found before i = m.

As above it is the same case for chaining and open addressing. For open addressing it is the same as mention above, that an available slot for a key can in best case be found at once, and in worst case be found when i = m. For chaining it depends on the linked list instead. It only one key is assigned to a slot the best case is $\Omega(1)$ time, but in the worst case all elements are assigned to the same slot, which makes the linked list of length m, which results in O(n) time.

4 EXPERIMENTAL RESULTS

For the experimental results I will provide the computing time for inserting all the elements into the hash table, and the space requirements. The space requirements will be provided for usage of open addressing and chaining. The computing times for the three operations and the computing time will be provided for the three implemented hash functions with chaining for collisions, but also for open addressing with each of the three open addressing methods. I will then compare the open addressing functions against each other and compare them against chaining. These comparisons will be done with two different datasets with different sizes (Video_Games = 16.720 elements, and disney-voice-actors = 936 elements).

All of the running time results and space requirements can be found in the GitHub in a file called ExperimentResults.

The space requirements are the same for all of the cases, which is 0.15264129638671875 MB. This is the case since the size of the hash table m is the same for all the test runs, where m = 20.000. This decision was made based on the sizes of the datasets mentioned above. This would result in a smaller dataset with many available slots, and the second case with less than 4.000 available slots after the insertion of the dataset.

For the running time there is also some noticeable problems, since the running time is based on inserting all the element's key values, the open addressing functions are having problems. This happens in the form of primary clustering and secondary clustering for linear and quadratic probing. The functions can still be compared against each other, but in this case it would not necessarily provide an unbiased result when comparing chaining and open addressing. This would only be a problem for the $Video_Gamedataset$, since the amount of keys and the length of the hash table is fairly close to each other.

For the disney-voice-actor dataset the chaining technique for collisions are performing best with the multiplication hash function, where it performs 0.4 sec faster than the division hash function, and 0.9 sec faster than the universal hash function. For the open addressing functions all of them perform better with the multiplication hash function. An example of this is using linear probing for collisions and running it with the three different hash functions. For the division method the time is 2.19 sec, for multiplication it is 1.28 sec, and for universal hashing it is 2.47 sec. This is the same case for quadratic probing as well. For double hashing the times a closer together, where for division took 1.80 sec, multiplication 1.51 sec, and universal took 2.12. For double hashing the time for the division and multiplication method were closer, and the universal hashing method also performed faster.

I wanted to run the results with a much bigger dataset, so there would be less free spots in the hash table. This results in worse average search time, but I can compare the time for chaining and open addressing while there would be more collisions.

The results differs a lot from the last dataset, even though it is still the multiplication method with usage of chaining for collision handling, that is the fastest with 5.06 sec. The multiplication method with chaining is faster than the division method and universal method by at least 6.5 sec to the division method. The most interesting happens for the open addressing, where the multiplication method is not the fastest method consistently like for the former dataset. In the case of linear probing the multiplication method is the fastest by far. The multiplication takes 5.19 sec to compute, which is also close to the chaining method. For the division and universal methods the times are much slower, since the division method takes 24.63 sec, and universal takes 23.27 sec. For the quadratic collision technique the results were flipped. The division method was the fastest with a time of 15.17 sec. The universal method took 21.45 sec, and the slowest function were the multiplication method, which took 48.51 sec. This result is completely different from the former open addressing technique. For the last open addressing technique, double hashing, the

results were yet again different. The multiplication method took 9.71 sec, the division method took 18.90 sec, and the universal method took 25.36 sec.

5 CONCLUSION

Based on the gather results it can be concluded, that the multiplication method performs the best with chaining for collision handling, when a smaller dataset is inserted to a larger hash table. The same can be concluded for the larger dataset with the same amount of available slots.

When we only focus on open addressing for collision handling, it is best to use in combination with the multiplication method, when used in a case with many available slots. For the case with the larger dataset, linear probing performed better, when used with the multiplication method.

When putting the two collision methods against each other, the chaining method performs the fastest. This is the case for both datasets, which the running times have been conducted on.

REFERENCES

Cormen, T. H., Leiserson, Charles E. and, R. R. L., and Stein, C. (2009). *Introduction to algorithms (3rd Edition)*. MIT Press, Cambridge, MA, USA.