**Hash Tables**

Table of Contents

[Hash Tables and Hash Functions 3](#_Toc48982735)

[Mapping Keys and Avoiding Collisions 4](#_Toc48982736)

[Collision Resolution 6](#_Toc48982737)

[Open Hashing 6](#_Toc48982738)

[Load Factor 7](#_Toc48982739)

[Open Addressing 8](#_Toc48982740)

[Linear Probing 8](#_Toc48982741)

[Quadratic Probing 9](#_Toc48982742)

[Double Hashing 10](#_Toc48982743)

We normally store data that can be tabulated in some sort of table. This entire table in turn could be stored in a variable or a structure. In the technique of hashing, this table is called a hash table.

Using the methods we have learnt so far, if we want to insert, delete or search for some item in the database, we would have to go through the list. Whatever process we use, the time complexity would be O(n) in the worst case, and O(log n) in the best case if using binary search. Hashing however, promises to perform any of the three operations in O(1). Hashing is the technique used to implement a hash table.

Hashing is used by compilers to store variables, by online spell checkers, by search engines to look for documents containing a given word and in many other places.

## Hash Tables and Hash Functions

In a hash table, the records are stored as key, value pairs. The value may consist of more than one piece of data. Each value is accessed using the key, and not directly an index like we did for arrays. The key may be a string, an integer, or almost anything at all. This sort of data structure is already present in C++, where it is called a hash map, and in Python, where it is called a dictionary.

The hash table has a fixed size, say tableSize. To keep things simple, we will assume that this is some fixed value. Dictionaries and hash maps also give hash tables a fixed value, but they are designed to dynamically allocate more space as more data is entered into the table. Each of the keys are mapped onto one of the indices within the range. A function is used to perform some calculations based on the key and decide which index a particular key should go to. Because all the functions related to hashing depend on this function to perform some calculations to figure out the exact index of the key, the time complexity of O(1) can be achieved. This function is called a hash function. For keys that are integers, we could have used the key itself as the index position of an array, but if the keys are very large numbers then we would need to use a very large array to store them, which is inefficient. The hash function allows us to find a much smaller index position for those large numbers.

The way we design our hash function must be very efficient so as to make the process fast. If we end up creating a function that takes more time to calculate the index for a key than it would to manually search for it, out endeavour would be pointless.

Note that hashing is not always a good method to use. When we frequently have to perform such functions as finding the maximum or minimum values within our data or printing the data in a sorted manner, hashing will give poor results. This is because the data in the hash table is not stored in any particular order, and thus cannot be looped over to easily retrieve the required information.

## Mapping Keys and Avoiding Collisions

A major problem we will face while designing our hash functions are collisions. This is when the function assigns a particular key to an index that is already occupied by another key. When this happens, an alternative position needs to be found for the new key. If this happens once or twice it is not a problem, but if it happens repeatedly, the extra calculations that will need to be done will become time consuming.

There are a few techniques used to map keys efficiently so as to avoid collisions. For example, it is common to find the indices for keys that are integers as index = key % tableSize. This is called the ‘key mod N’ method. As such, if tableSize is 10 and we have three keys, 1001, 1005 and 1003, they will be mapped onto the indices 1, 5 and 3 respectively. This method cannot be used in the keys have some properties that would cause problems. For example, if we have numbers ending in 0, using the above method they would all be given the index 0, which would cause collisions.

Common methods used to map integer keys include, but are not limited to:

* Key Mod N - index = key % tableSize
* Truncation - Select a few digits at specific positions from the number
* Folding - Divide key into parts and use sum of parts as index

- E.g. For 123456789 index = (123 + 456 + 789) % 1000 = 368

* Squaring - If normal truncation does not work, square the result

Mapping strings is a little more complicated, since they need to be converted to numerical values somehow. For example, we could use the sum of the ASCII values of each of the characters in the key as the index, but this would not be very efficient and would cause collisions frequently. For example, if we had two keys abc and cab, they would both give the same results and would be mapped onto the same index.

Alternatively, we could use index = sum (ASCII value \* pow(10, position)) which would give us better results and far fewer collisions. The recommended method for strings is to use a table size that is a prime number, since that avoids collisions. More methods for mapping string keys will be discussed later.

Hashing Function Example

## Collision Resolution

There are four methods to resolving collisions:

* Separate Chaining/Open Hashing
* Open Addressing/Closed Hashing
  + Linear Probing
  + Quadratic Probing
  + Double Hashing

### Open Hashing

When using separate chaining, the hash table does not actually store the keys, but rather stores pointers to the keys. When collisions occur, the new keys are stored in the same index, using the pointers to chain together keys that are on the same index, similar to how a linked list works.

For example, if the hashing function is and three keys 0, 1 and 81 are given, the 0th index of the hash table points to the key 0, which in turn points to NULL and the 1st index of the hash table points to the key 1, which in turn points to NULL. When the key 81 is found and it is determined that it should also reside in the 1st index, the 1st index points to the key 81, which points to key 1, which points to NULL. 81 is now said to be the head of the 1st index.

Separate chaining is a very simple method that maintains a time complexity of O(1) for insertions. It also has good space utilization, since the table essentially never fills up. However, if we want to search for or delete an element, it may take a long time, since we would need to sequentially go over every element at a particular index. The time complexity for searching and deletion is not guaranteed to be O(1).

Disadvantages of separate chaining include the need for extra space to store the pointers, and the fact that space is also wasted sometimes since there may be spots on the table that are never used at all.

### Load Factor

An important parameter to consider when deciding which collision resolution method to use is the load factor, . The load factor is a measure of the number of elements per node on average, i.e. , where is the total number of elements in the table. It is good practice to keep the load factor close to 1. If there is too little storage, the load factor will be higher and there will be more frequent collisions.

Considering this information, the separate chaining method begins to look bad. For one thing, the load factor for separate chaining can be much, much greater than 1, since we have no limits on the maximum number of elements that a table can store. For an unsuccessful search, we would need to go to a specific node, which would take constant time, but then we would need to search along the entire length of that node. Say the node length is . Then the time complexity becomes O(1 + ). Even for a successful search, if we assume that we have to go through about half the length of a node on average, the time complexity becomes O(1 + ). In the worst case, the time complexity can even become O(n), which makes the entire process pointless. As such, separate chaining is only ever used when the number of elements that needs to be added is unknown, or the frequency with which insertions or deletions will take place is unknown. The three closed hashing methods, which all limit the load factor to 0.5, are better choices.

### Open Addressing

Under the open addressing method, all the keys go directly into the table and there are no chains. The load factor must be kept below 0.5, and as such, the table size needs to be at least twice the size needed to store all the keys. Remember that using this method does not guarantee that collisions will not occur, but when collisions do occur, alternative cells are found for the new keys.

#### Linear Probing

The linear probing method suggests trying to use the spot given by the hash function, and if the spot is not empty, moving to the immediate next available spot.

For example, say we have a table of size 10, and our hash function is . If the first key is 89, it is given the 9th index. If the next key is 49, it is also given the 9th index. However, since this spot is occupied, 49 must be placed in the next available spot. Since the table size is 10, there is no 10th index, so 49 must circle back and use the 0th index. If that index were also occupied it would keep going through the indices until it found an empty spot.

Our overall hash function becomes . is the collision resolution strategy. In our particular case, we are resolving collisions by sequentially searching the array until an empty cell is found. Thus, , where initially. If we get another key, 59, initially so . This is occupied, so , and . This is also occupied so and . This is empty so 59 goes to the 1st index.

The average number of cells examined during insertion is roughly . The proof behind this formula is beyond the scope of this course. Both successful and unsuccessful searches cost the same number of cell examinations as an insertion does.

The main problem with linear probing is that collisions cause clusters to form, since indices are occupied linearly by the new keys. This is called the primary clustering problem. Even if there is a huge amount of space available, if we have similar keys, they will all try to go to the same indices and will traverse linearly over occupied spaces. This problem is solved by the next method of closed hashing.

#### Quadratic Probing

Quadratic probing takes the exact same approach as linear probing, except that it changes the collision resolution strategy from to . This reduces the primary clustering problem. However, we need to ensure that the load factor is kept below 0.5, since otherwise, there is a good chance that keys that cause collisions will be unable to find an alternative spot. tableSize should also be a prime number, since it has been found that the problem will become more severe if it is not.

To keep the load factor below 0.5 as new keys are added, the table is dynamically expanded as approaches 0.5. This is called re-hashing. The new table size must also be a prime number.

Copying the current table into a new table might take linear time, but it is also possible to do this in constant time. This is done by inserting keys into the current table as well as into the new table as begins to approach 0.5, in preparation for re-hashing. The details of how this is done is an advanced topic. The only problem with this is that it uses a large amount of memory.

Quadratic probing has its own problems as well. Elements that are initially given the same index will still probe to the same alternative cells due to how the collision resolution strategy is set up. This results in another type of clustering called the secondary clustering problem, which results in the same problem we faced with linear probing for similar keys.

#### Double Hashing

Under double hashing, the collision resolution strategy is itself another hashing function. Thus, where .

The only rule that we must follow for the second hashing function is that it can never be 0 for any of the keys. For example, if and we are looking for an alternative position for the key 99, . As such, regardless of how high goes, we will always get the same index position, essentially entering an infinite loop.

A good function to use as the collision resolution strategy under double hashing is , where is a prime constant. This gives the best results. If we are looking for an alternative position for the key 59, and we have decided that , then . As such, .