21 Hashing

21.1 Introduction

In an ideal world, retrieving a value from a map would be done instantly by just examining the value's key. That is the goal of hashing, which uses a hash function to transform a key into an array index, thereby providing instantaneous access to the value stored in an array holding the key-value pairs in the map. This array is called a hash table.

> **Hash function**: A function that transforms a key into a value in the range of indices of a hash table.

> **Hash table**: An array holding key-value pairs whose locations are determined by a hash function.

Of course, there are a few details to work out.

21.2 The Hashing Problem

If a set of key-value pairs is small and we can allocate an array big enough to hold them all, we can always find a hash function that transforms keys to unique locations in a hash table. For example, in some old programming languages, identifiers consisted of an upper-case letter possibly followed by a digit. Suppose these are our keys. There are 286 of them, and it is not too hard to come up with a function that maps each key of this form to a unique value in the range 0..285. But usually the set of keys is too big to make a table to hold all the possibilities. For example, older versions of FORTRAN had identifiers that started with an upper-case letter, followed by up to five additional upper-case letters or digits. The number of such identifiers is 1,617,038,306, which is clearly too big for a hash table if we were to use these as keys.

A smaller table holding keys with a large range of values will inevitably require that the function transform several keys to the same table location. When two or more keys are mapped to the same table location by a hash function we have a collision. Mechanisms for dealing with them are called *collision resolution schemes*.

> **Collision**: The event that occurs when two or more keys are transformed to the same hash table location.

How serious is the collision problem? After all, if we have a fairly large table and a hash function that spreads keys out evenly across the table, collisions may be rare. In fact, however, collisions occur surprisingly often. To see why, lets consider the *birthday problem*, a famous problem from probability theory: what is the chance that a least two people in a group of *k* people have the same birthday? This turns out to be $p = 1-(365!/k!/365^k)$. Table 1 below lists some values for this expression. Surprisingly, in a group of only 23 people there is better than an even chance that two of them have the same birthday!

If we imagine that a hash table has 365 locations, and that these probabilities are the likelihoods that a hash function transforms two values to the same location (a collision), then we can see that we are almost certain to have a collision when the table holds 100 values, and very likely to have a collision with only about 40 values in the table. Forty is only about 11% of 365, so we see that collisions are very likely indeed. Collision resolution schemes are thus an essential part of making hashing work in practice.

k	*p*
5	0.027
10	0.117
15	0.253
20	0.411
22	0.476
23	0.507
25	0.569
30	0.706
40	0.891
50	0.970
60	0.994
100	0.9999997

Table 1: Probabilities in the Birthday Problem

An implementation of hashing thus requires two things:

- A hash function for transforming keys to hash table locations, ideally one that makes collisions less likely.
- A collision resolution scheme to deal with the collisions that are bound to occur.

We discuss each of these in turn.

21.3 Hash Functions

A hash function must transform a key into an integer in the range $0 \ldots t$, where t is the size of the hash table. A hash function should distribute the keys in the table as uniformly as possible to minimize collisions. Although many hash functions have been proposed and investigated, the best hash functions use the division method, which for numeric keys is the following.

$$hash(k) = k \ \% \ t$$

This function is simple, fast, and spreads out keys uniformly in the table. It works best when t is a prime number not close to a power of two. For this reason, hash table sizes should always be chosen to be a prime number not close to a power of two.

For non-numeric keys, there is usually a fairly simple way to convert the value to a number and then use the division method on it. For example, the following pseudocode illustrates a way to hash a string.

```
def hash_function(string, table_size)
  result = 0
  string.each_byte do |byte|
    result = (result * 151 + byte) % table_size
  end
  return result
end
```

Figure 1: A Ruby Hash Function for Strings

Making hash functions is not too onerous. Good rules of thumb are to use prime numbers whenever a constant is needed, and to test the function on a representative set of keys to ensure that it spreads them out evenly across the hash table.

21.4 Collision Resolution Schemes

There are two main kinds of collision resolution schemes, with many variations: chaining and open addressing. In each scheme, an important value to consider is the load factor, $\lambda = n/t$, where n is the number of elements in the hash table and t is the table size.

Chaining

In **chaining** (or **separate chaining**) records whose keys collide are formed into a linked list or chain whose head is in the hash table array. Figure 2 below shows a hash table with collisions resolved using chaining. For simplicity, only the keys are listed and not the values that go along with them (or, if you like, the key and the value are the same).

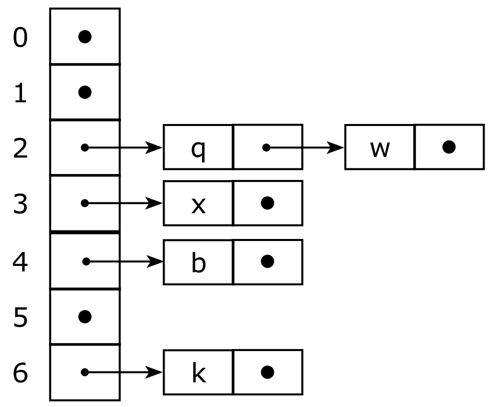

Figure 2: Hash Table with Chaining to Resolve Collisions

In this example, the table has seven locations. Two keys, *q* and *w*, collide at location two and they are placed in a linked list whose head is at location two. The keys *x*, *b*, and *k* are hashed to locations three, four, and six, respectively. This example uses an array of list heads, but an array of list nodes could have been used as well, with some special value in the data field to indicate when a node is unused.

The average chain length is λ. If the chain is unordered, on average successful searches require about 1+ λ/2 comparisons and unsuccessful searches about λ comparisons. If the chain is ordered, both successful and unsuccessful searches take about 1+ λ/2 comparisons, but insertions take longer. In the worse case, which occurs when all keys map to a single table location and the search key is at the end of the linked list or not in the list, searches require *n* comparisons. But this case is extremely unlikely.

More complex linked structures (like binary search trees), don't generally improve performance much, particularly if λ is kept fairly small. As a rule of thumb, λ should be kept less than about 10. But performance only degrades gradually as the number of items in the table grows, so hash tables that use chaining to resolve collisions can perform well on wide ranges of values of *n*.

Open Addressing

In the open addressing collision resolution scheme, records with colliding keys are stored at other free locations in the hash table found by *probing* the table for open locations. Different kinds of open addressing schemes use different *probe sequences*. In all cases, however, the table can only hold as many items as it has locations, so $n \le t$ and λ cannot exceed one; this constraint is not present for chaining.

Theoretical studies of random probe sequences have determined ideal levels of performance for open addressing. Performance declines sharply as the load factor approaches one. For example, with a load factor of 0.9, the number of comparisons for a successful search is about 2.6, and for an unsuccessful search is 20. Real open addressing schemes do not do even as well as this, so load factors must generally be kept below about 0.75.

Another point to understand about open addressing is that when a collision occurs, the algorithm proceeds through the probe sequence until either (a) the desired key is found, (b) an open location is found, or (c) the entire table is traversed. But this only works when a marker is left in slots where an element was deleted to indicate that the location may not have been empty before, and so the algorithm should proceed with the probe sequence. In a highly dynamic table there will be many markers and few empty slots, so the algorithm will need to follow long probe sequences, especially for unsuccessful searches, even when the load factor is low.

Linear probing is using a probe sequence that begins with the hash table index and increments it by a constant value modulo the table size. If the table size and increment are relatively prime, every slot in the table will appear in the probe sequence. Linear probing performance degrades sharply when load factors exceed 0.8. Linear probing is also subject to *primary clustering*, which occurs when clumps of filled locations accumulate around a location where a collision first occurs. Primary clustering increases the chances of collisions and greatly degrades performance.

Double hashing works by generating an increment for the probe sequence by applying a second hash function to the key. The second hash function should generate values quite different from the first so that two keys that collide will be mapped to different values by the second hash function, making the probe sequences for the two keys different. Double hashing eliminates primary clustering. The second hash function must always generate a number that is relatively prime to the table size. This is easy if the table size is a prime number. Double hashing works so well that its performances approximates that of a truly random probe sequence. It is thus the method of choice for generating probe sequences.

Figure 3 below shows an example of open addressing with double hashing. As before, the example only uses keys for simplicity, not key-value pairs. The main hash function is $f(x) = x \% 7$, and the hash function used to generate a constant for the probe sequence is $g(x) = (x \% 5)+1$. The values 8, 12, 9, 6, 25, and 22 are hashed into the table.

0	22
1	8
2	9
3	--
4	25
5	12
6	6

Figure 3: Hash Table with Open Addressing and Double Hashing to Resolve Collisions

The first five keys do not collide. But 22 % 7 is 1, so 22 collides with 8. The probe constant for double hashing is (22 % 5)+1 = 3. We add 3 to location 1, where the collision occurs, to obtain location 4. But 25 is already at this location, so we add 3 again to obtain location 0 (we wrap around to the start of the array using the table size: (4+3) % 7 = 0). Location 0 is not occupied, so that is where 22 is placed.

Note that some sort of special value must be placed in the unoccupied locations—in this example we used a double dash. A different value must be used when a value is removed from the table to indicate that the location is free, but that it was not before, so that searches must continue past this value when it is encountered during a probe sequence.

We have noted that when using open addressing to resolve collisions, performance degrades considerably as the load factor approaches one. In effect this means that hashing mechanisms that use open addressing must have a way to expand the table so they can lower the load factor and improve performance. A new, larger table can be created and filled by traversing the old table and inserting all records into the new table. Note that this involves hashing every key again because the hash function will generally use the table size, which has now changed. Consequently, this is a very expensive operation.

Some table expansion schemes work incrementally by keeping the old table around and making all insertions in the new table, all deletions from the old table, and perhaps moving records gradually from the old table to the new in the course of doing other operations. Eventually the old table becomes empty and can be discarded.

21.5 Summary and Conclusion

Hashing uses a hash function to transform a key into a hash table location, thus providing almost instantaneous access to values though their keys. Unfortunately, it is inevitable that more than one key will be hashed to each table location, causing a collision and requiring some way to store more than one value associated with a single table location.

The two main approaches to collision resolution are chaining and open addressing. Chaining uses linked lists of key-value pairs that start in hash table locations. Open addressing uses probe sequences to look through the table for an open spot to store a key-value pair and then later to find it again. Chaining is very robust and has good performance for a wide range of load factors, but it requires extra space for the links in the list nodes. Open addressing uses space efficiently, but its performance degrades quickly as the load factor approaches one; expanding the table is very expensive.

No matter how hashing is implemented, however, average performance for insertions, deletions, and searches is $O(1)$. Worst case performance is $O(n)$ for chaining collision resolution, but this only occurs in the very unlikely event that the keys are hashed to a single table location. Worst case performance for open addressing is a function of the load factor that gets very large when λ is near one, but if λ is kept below about 0.8, $W(n)$ is less than 10.

21.6 Review Questions

1. What happens when two or more keys are mapped to the same location in a hash table?
2. If a hash table has 365 locations and 50 records are placed in the table at random, what is the probability that there will be at least one collision?
3. What is a good size for hash tables when a hash function using the division method is used?
4. What is a load factor? Under what conditions can a load actor exceed one?
5. What is a probe sequence? Which is better: linear probing or double hashing?

21.7 Exercises

1. Why does the example of open addressing and double hashing in the text use the hash function $g(x) = (x \% 5)+1$ rather than $g(x) = x \% 5$ to generate probe sequences?
2. Suppose a hash table has 11 locations and the simple division method hash function $f(x) = x \% 11$ is used to map keys into the table. Compute the locations where the following keys would be stored: 0, 12, 42, 18, 6, 22, 8, 105, 97. Do any of these keys collide? What is the load factor of this table if all the keys are placed into it?
3. Suppose a hash table has 11 locations, keys are placed in the table using the hash function $f(x) = x \% 11$, and linear chaining is used to resolve collisions. Draw a picture similar to Figure 2 of the result of storing the following keys in the table: 0, 12, 42, 18, 6, 22, 8, 105, 97.
4. Modify with the diagram you made for Exercise 3 to show what happens when 18 and 42 are removed from the hash table.
5. Suppose a hash table has 11 locations, keys are mapped into the table using the hash function $f(x) = x \% 11$, and collisions are resolved using open addressing and linear probing with a constant of three to generate the probe sequence. Draw a picture of the result of storing the following keys in the table: 0, 12, 42, 18, 6, 22, 8, 105, 97.
6. List the probe sequence (the table indices) used to search for 97 in the diagram you drew for problem 5. List the probe sequence used when searching for 75.
7. Starting with the diagram you made for Exercise 5, show the result of removing 18 from the table. List the probe sequence used to search for 97. How do you guarantee that 97 is found even though 18 is no longer encountered in the probe sequence?
8. Suppose a hash table has 11 locations, keys are mapped into the table using the hash function $f(x) = x \% 11$, and collisions are resolved using double hashing with the hash function $g(x) = (x \% 3)+1$ to generate the probe sequence. Draw a picture of the result of storing the following keys in the table: 0, 12, 42, 18, 6, 22, 8, 105, 97.
9. List the probe sequences used to search for 97 and 75 using the diagram you drew for Exercise 8. In what way are these sequences different from the probe sequences generated in Exercise 6?

21.8 Review Question Answers

1. When two or more keys are mapped to the same location in a hash table, they are said to *collide*, and some action must be taken, called *collision resolution*, so that records containing colliding keys can be stored in the table.
2. If 50 values are added at random to a hash table with 365 locations, the probability that there will be at least one collision is 0.97, according to the Table 1.
3. A good size for hash tables when a division method hash function is used is a prime number not close to a power of two.

4. The load factor of a hash table is the ratio of the number of key-value pairs in the table to the table size. In open addressing, the load factor cannot exceed one, but with chaining, because in effect more than one key-value pair can be stored in each location, the load factor can exceed one.
5. A probe sequence is a list of table locations checked when elements are stored or retrieved from a hash table that resolves collisions with open addressing. Linear probing is subject to primary clustering, which decreases performance, but double hashing as been shown to be as good as choosing increments for probe sequences at random.