How does a hash table work?

I'm looking for an explanation of how a hashtable works - in plain English for a simpleton like me!

For example, I know it takes the key, calculates the hash (I am looking for an explanation how) and then performs some kind of modulo to work out where it lies in the array where the value is stored, but that's where my knowledge stops.

Could anyone clarify the process?

Edit: I'm not looking specifically about how hashcodes are calculated, but a general overview of how a hashtable works.

data-structures hash hashtable modulo

edited Oct 18 at 15:55
Student
161 2 17

asked Apr 8 '09 at 15:48
Arec Barrwin
17.9k 9 18 25

1 One simple explanation and implementation examples. – Mihai Limbășan Apr 8 '09 at 15:52

4 Recently, I have written this (en.algoritmy.net/article/50101/Hash-table) article describing several ways, how to store and lookup data, with accent on hash tables and their strategies (separate chaining, linear probing, double hashing) – malejpavouk Mar 27 '13 at 20:46

An exclusive article on hashing ...techieme.in/hashing-in-detail-part-one – – dharam Mar 8 at 18:16

15 Answers

Here's an explanation in layman's terms.

Let's assume you want to fill up a library of books and not just stuff them in there, but you want to be able to easily find them again when you need them.

So, you decide that if the person that wants to read a book knows the title of the book and the exact title to boot, then that's all it should take. With the title, the person, with the aid of the librarian, should be able to find the book easily and quickly.

So, how can you do that? Well, obviously you can keep some kind of list of where you put each book, but then you have the same problem as searching the library, you need to search the list. Granted, the list would be smaller and easier to search, but still you don't want to search sequentially from one end of the library (or list) to the other.

You want something that, with the title of the book, can give you the right spot at once, so all you have to do is just stroll over to the right shelf, and pick up the book.

But how can that be done? Well, with a bit of forethought when you fill up the library and a lot of work when you fill up the library.

Instead of just starting to fill up the library from one end to the other, you devise a clever little method. You take the title of the book, run it through a small computer program, which spits out a shelf number and a slot number on that shelf. This is where you place the book.

The beauty of this program is that later on, when a person comes back in to read the book, you feed the title through the program once more, and get back the same shelf number and slot number that you were originally given, and this is where the book is located.

The program, as others have already mentioned, is called a hash algorithm or hash computation and usually works by taking the data fed into it (the title of the book in this case) and calculates a number from it.

For simplicity, let's say that it just converts each letter and symbol into a number and sums them all up. In reality, it's a lot more complicated than that, but let's leave it at that for now.

The beauty of such an algorithm is that if you feed the same input into it again and again, it will keep spitting out the same number each time.

Ok, so that's basically how a hash table works.

Technical stuff follows.

First, there's the size of the number. Usually, the output of such a hash algorithm is inside a range of some large number, typically much larger than the space you have in your table. For instance, let's say that we have room for exactly one million books in the library. The output of the hash calculation could be in the range of 0 to one billion which is a lot higher.

So, what do we do? We use something called modulus calculation, which basically says that if you counted to the number you wanted (i.e. the one billion number) but wanted to stay inside a much smaller range, each time you hit the limit of that smaller range you started back at 0, but you have to keep track of how far in the big sequence you've come.

Say that the output of the hash algorithm is in the range of 0 to 20 and you get the value 17 from a particular title. If the size of the library is only 7 books, you count 1, 2, 3, 4, 5, 6, and when you get to 7, you start back at 0. Since we need to count 17 times, we have 1, 2, 3, 4, 5, 6, 0, 1, 2, 3, 4, 5, 6, 0, 1, 2, 3, and the final number is 3.

Of course modulus calculation isn't done like that, it's done with division and a remainder. The remainder of dividing 17 by 7 is 3 (7 goes 2 times into 17 at 14 and the difference between 17 and 14 is 3).

Thus, you put the book in slot number 3.

This leads to the next problem. Collisions. Since the algorithm has no way to space out the books so that they fill the library exactly (or the hash table if you will), it will invariably end up calculating a number that has been used before. In the library sense, when you get to the shelf and the slot number you wish to put a book in, there's already a book there.

Various collision handling methods exist, including running the data into yet another calculation to get another spot in the table, or simply to find a space close to the one you were given (i.e. right next to the previous book). This would mean that you have some digging to do when you try to find the book later, but it's still better than simply starting at one end of the library.

Finally, at some point, you might want to put more books into the library than the library allows. In other words, you need to build a bigger library. Since the exact spot in the library was calculated using the exact and current size of the library, it goes to follow that if you resize the library you might end up having to find new spots for all the books since the calculation done to find their spots has changed.

I hope this explanation was a bit more down to earth than buckets and functions :)

edited Oct 18 at 16:19
Student
161 2 17

answered Apr 8 '09 at 16:33
Lasse V. Karlsen
191k 57 392 586

Thanks for such a great explanation. Do you know where I can find more technical details regarding how it's implemented in 4.x .Net framework? – Johnny_D Jan 26 at 10:14

1 Is that shelf and slot position a pointer? – CMCDragonkai Mar 12 at 13:31

No, it is just a number. You would just number each shelf and slot starting at 0 or 1 and increasing by 1 for each slot on that shelf, then continue numbering on the next shelf. – Lasse V. Karlsen Mar 12 at 13:40

This explanation is amazing. Thank you! – Joao Sousa Aug 22 at 9:07

'Various collision handling methods exist, including running the data into yet another calculation to get another spot in the table' - what do you mean by another calculation? It is just another algorithm? OK, so suppose we use another algorithm that outputs a different number based on the book name. Then later on, if I were to find that book, how would I know which algorithm to use? I'd use first algorithm, second algorithm and so on until I find the book whose title is the one I'm looking for? – user107986 Sep 6 at 9:45

This turns out to be a pretty deep area of theory, but the basic outline is simple.

Essentially, a hash function is just a function that takes things from one space (say strings of arbitrary length) and maps them to a space useful for indexing (unsigned integers, say).

If you only have a small space of things to hash, you might get away with just interpreting those things as integers, and you're done (e.g. 4 byte strings)

Usually, though, you've got a much larger space. If the space of things you allow as keys is bigger than the space of things you are using to index (your uint32's or whatever) then you can't possibly have a unique value for each one. When two or more things hash to the same result, you'll have to handle the redundancy in an appropriate way (this is usually referred to as a collision, and how you handle it or don't will depend a bit on what you are using the hash for).

This implies you want it to be unlikely to have the same result, and you probably also would really like the hash function to be fast.

Balancing these two properties (and a few others) has kept many people busy!

In practice you usually should be able to find a function that is known to work well for your application and use that.

Now to make this work as a hashtable: Imagine you didn't care about memory usage. Then you can create an array as long as your indexing set (all uint32's, for example). As you add something to the table, you hash it's key and look at the array at that index. If there is nothing there, you put your value there. If there is already something there, you add this new entry to a list of things at that address, along with enough information (your original key, or something clever) to find which entry actually belongs to which key.

So as you go a long, every entry in your hashtable (the array) is either empty, or contains one entry, or a list of entries. Retrieving is a simple as indexing into the array, and either returning the value, or walking the list of values and returning the right one.

Of course in practice you typically can't do this, it wastes too much memory. So you do everything based on a sparse array (where the only entries are the ones you actually use, everything else is implicitly null).

There are lots of schemes and tricks to make this work better, but that's the basics.

edited Jun 4 '09 at 17:35 answered Apr 8 '09 at 16:11
simon
5,830 1 19 24

Usage and Lingo:

1. **Hash tables** are used to quickly store and retrieve data (or records).
2. Records are stored in **buckets** using **hash keys**
3. **Hash keys** are calculated by applying a hashing algorithm to a chosen value contained within the record. This chosen value must be a common value to all the records.
4. Each **bucket** can have multiple records which are be organized in a particular order.

Real World Example:

Hash & Co., founded in 1803 and lacking any computer technology had a total of 300 filing cabinets to keep the detailed information (the records) for their approximately 30,000 clients. Each file folder were clearly identified with its unique number from 0 to 299.

The filing clerks of that time had to quickly fetch and store client records for the working staff. The staff had decided that it would be more efficient to use a hashing methodology to store and retrieve their records.

To file a client record, filing clerks would use the unique client number written on the folder. Using this client number, they would modulate it by 300 (the **hash key**) in order to identify the filing cabinet it is contained in. When they opened the filing cabinet they would discover that it contained many folders ordered by client number. After identifying the correct location, they would simply slip it in.

To retrieve a client record, filing clerks would be given a client number on a slip of paper. Using this unique client number, they would modulate it by 300 (the **hash key**) in order to determine which filing cabinet had the clients folder. When they opened the filing cabinet they would discover that it contained many folders ordered by client number. Searching through the records they would quickly find the client folder and retrieve it.

In our real-world example, our **buckets** are **filing cabinets** and our **records** are **file folders**.

An important thing to remember is that computers (and their algorithms) deal with numbers better than with strings. So accessing a large array using an index is significantly much faster than accessing sequentially.

As Simon has mentioned which I believe to be **very important** is that the hashing part is to transform a large space (of arbitrary length, usually strings, etc) and mapping it to a small space (of known size, usually numbers) for indexing. This if very important to remember!

So in the example above, the 30,000 possible clients or so are mapped to a smaller space.

The main idea in this is to divide your entire data set into segments as to speed up the actual searching which is usually time consuming. In our example above, each of the 300 filing cabinet would (statistically) contain about 100 records. Searching (regardless the order) through 100 records is much faster than having to deal with 30,000.

You may have noticed that some actually already do this. But instead of devising a hashing methodology to generate a hash key, they will in most cases simply use the first letter of the last name. So if you have 26 filing cabinets each containing a letter from A to Z, you in theory have just segmented your data and enhanced the filing and retrieval process.

Hope this helps,

Jeach!

edited Dec 28 '14 at 19:53
David
111 1 10

answered Apr 8 '09 at 17:20
Jeach
2,392 4 26 44

You describe a specific type of hash table collision avoidance strategy, called variably "open addressing" or "closed addressing" (yes, sad but true) or "chaining". There's another type which doesn't use list buckets but instead stores the items "inline". – Konrad Rudolph Apr 8 '09 at 17:29

2 excellent description. except each filing cabinet would contain, on average, about `100` records (30k records / 300 cabinets = 100). Might be worth an edit. – Ryan Tuck Nov 7 '14 at 15:30

You guys are very close to explaining this fully, but missing a couple things. The hashtable is just an array. The array itself will contain something in each slot. At a minimum you will store the hashvalue or the value itself in this slot. In addition to this you could also store a linked/chained list of values that have collided on this slot, or you could use the open addressing method. You can also store a pointer or pointers to other data you want to retrieve out of this slot.

It's important to note that the hashvalue itself generally does not indicate the slot into which to place the value. For example, a hashvalue might be a negative integer value. Obviously a negative number cannot point to an array location. Additionally, hash values will tend to many times be larger numbers than the slots available. Thus another calculation needs to be performed by the hashtable itself to figure out which slot the value should go into. This is done with a modulus math operation like:

```
uint slotIndex = hashValue % hashTableSize;
```

This value is the slot the value will go into. In open addressing, if the slot is already filled with another hashvalue and/or other data, the modulus operation will be run once again to find the next slot:

```
slotIndex = (remainder + 1) % hashTableSize;
```

I suppose there may be other more advanced methods for determining slot index, but this is the common one I've seen... would be interested in any others that perform better.

With the modulus method, if you have a table of say size 1000, any hashvalue that is between 1 and 1000 will go into the corresponding slot. Any Negative values, and any values greater than 1000 will be potentially colliding slot values. The chances of that happening depend both on your hashing method, as well as how many total items you add to the hash table. Generally, it's best practice to make the size of the hashtable such that the total number of values added to it is only equal to about 70% of its size. If your hash function does a good job of even distribution, you will generally encounter very few to no bucket/slot collisions and it will perform very quickly for both lookup and write operations. If the total number of values to add is not known in advance, make a good guesstimate using whatever means, and then resize your hashtable once the number of elements added to it reaches 70% of capacity.

I hope this has helped.

PS - In C# the `GetHashCode()` method is pretty slow and results in actual value collisions under a lot of conditions I've tested. For some real fun, build your own hashfunction and try to get it to NEVER collide on the specific data you are hashing, run faster than GetHashCode, and have a fairly even distribution. I've done this using long instead of int size hashcode values and it's worked quite well on up to 32 million entires hashvalues in the hashtable with 0 collisions.

Unfortunately I can't share the code as it belongs to my employer... but I can reveal it is possible for certain data domains. When you can achieve this, the hashtable is VERY fast. :)

edited Jan 5 at 23:09
voidHead
650 6 17

answered May 15 '10 at 1:41
Chris
149 1 3

i know the post is pretty old but can someone explain what (remainder + 1) means here – Hari Dec 29 '14 at 12:45

3 @Hari `remainder` refers to the result of the original modulo calculation, and we add 1 to it in order to find the next available slot. – voidHead Jan 5 at 22:52

This is how it works in my understanding:

Here's an example: picture the entire table as a series of buckets. Suppose you have an implementation with alpha-numeric hash-codes and have one bucket for each letter of the alphabet. This implementation puts each item whose hash code begins with a particular letter in the corresponding bucket.

Let's say you have 200 objects, but only 15 of them have hash codes that begin with the letter 'B.' The hash table would only need to look up and search through the 15 objects in the 'B' bucket, rather than all 200 objects.

As far as calculating the hash code, there is nothing magical about it. The goal is just to have different objects return different codes and for equal objects to return equal codes. You could write a class that always returns the same integer as a hash-code for all instances, but you would essentially destroy the usefulness of a hash-table, as it would just become one giant bucket.

answered Apr 8 '09 at 16:02
AndreiM
2,840 3 22 41

Short and sweet:

A hash table wraps up an array, lets call it `internalArray`. Items are inserted into the array in this way:

```
let insert key value =
    internalArray[hash(key) % internalArray.Length] <- (key, value)
    //oversimplified for educational purposes
```

Sometimes two keys will hash to the same index in the array, and you want to keep both values. I like to store both values in the same index, which is simple to code by making `internalArray` an array of linked lists:

```
let insert key value =
    internalArray[hash(key) % internalArray.Length].AddLast(key, value)
```

So, if I wanted to retrieve an item out of my hash table, I could write:

```
let get key =
    let linkedList = internalArray[hash(key) % internalArray.Length]
    for (testKey, value) in linkedList
        if (testKey = key) then return value
    return null
```

Delete operations are just as simple to write. As you can tell, inserts, lookups, and removal from our array of linked lists is *nearly* O(1).

When our internalArray gets too full, maybe at around 85% capacity, we can resize the internal array and move all of the items from the old array into the new array.

answered Apr 8 '09 at 17:24
Juliet
46.7k 30 149 203

It's even simpler than that.

A hashtable is nothing more than an array (usually sparse one) of vectors which contain key/value pairs. The maximum size of this array is typically smaller than the number of items in the set of possible values for the type of data being stored in the hashtable.

The hash algorithm is used to generate an index into that array based on the values of the item that will be stored in the array.

This is where storing vectors of key/value pairs in the array come in. Because the set of values that can be indexes in the array is typically smaller than the number of all possible values that the type can have, it is possible that your hash algorithm is going to generate the same value for two separate keys. A *good* hash algorithm will prevent this as much as possible (which is why it is relegated to the type usually because it has specific information which a general hash algorithm can't possibly know), but it's impossible to prevent.

Because of this, you can have multiple keys that will generate the same hash code. When that happens, the items in the vector are iterated through, and a direct comparison is done between the key in the vector and the key that is being looked up. If it is found, great and the value associated with the key is returned, otherwise, nothing is returned.

answered Apr 8 '09 at 16:04
casperOne ♦
54.6k 10 116 194

You take a bunch of things, and an array.

For each thing, you make up an index for it, called a hash. The important thing about the hash is that it 'scatter' a lot; you don't want two similar things to have similar hashes.

You put your things into the array at position indicated by the hash. More than one thing can wind up at a given hash, so you store the things in arrays or something else appropriate, which we generally call a bucket.

When you're looking things up in the hash, you go through the same steps, figuring out the hash value, then seeing what's in the bucket at that location and checking whether it's what you're looking for.

When your hashing is working well and your array is big enough, there will only be a few things at most at any particular index in the array, so you won't have to look at very much.

For bonus points, make it so that when your hash table is accessed, it moves the thing found (if any) to the beginning of the bucket, so next time it's the first thing checked.

answered Apr 8 '09 at 16:22
chaos
80.3k 15 221 266

Hashing is a technique that promises to achieve ***O(1) data lookup***. This doesn't mean that only ***one comparison will be made***, but rather that the ***number of comparisons will remain the same***, no matter how large the data set. Compare this with the ***O(N) search time*** of simple linked lists or even O(log N) for binary search trees, and hashing starts to look very attractive.
Understanding Hashing
You may not even realize it, but chances are good you use concepts similar to hashing all the time. When you walk into a bookstore and head straight for the computer book section, you've just used a kind of hashing algorithm. When you are looking for a music CD by a particular artist, you no doubt go straight to the section containing CDs by artists with the same first letter of that artist's surname. Both these processes involve taking some property of the thing you are looking for—a book category or an artist's name—and using that to narrow down the search. In the case of the book, you know it is a computer book so you head straight for that section; in the case of the CD, you know the artist's name. Hashing begins with a hash function.

```
A hash function takes one value—a string, an object, a number,
and so on—and produces a hash value, usually an integer or some other numeric value.
```

This hash value is then used to locate a position within a hash table—a special kind of array. To give you an idea of how hashing works, the following example first shows you how to produce a hash value for strings; then it proceeds to use the hash value to store and locate the strings.

One of the simplest string hashing techniques involves adding letters. The resulting hash value can then be used as a position within the hash table for storing the string. The following code example shows the results of hashing the strings "elvis," "madonna," and "sting," assuming the letters of the alphabet are assigned the values 1 for a through 26 for z:

```
e + l + v + i + s = 5 + 12 + 22 + 9 + 19
= 67
m + a + d + o + n + n + a = 13 + 1 + 4 + 15 + 14 + 14 + 1
= 62
s + t + i + n + g = 19 + 20 + 9 + 14 + 7
= 69
```

Looking at the generated values, you can see that the string “elvis” would be placed into an array at position 67, “madonna” at position 62, and “sting” at position 69. Notice that the strings aren’t stored in any particular order. The positions seem random, and in fact hashing is sometimes referred to as randomizing. This is quite different from all of the other data structures and algorithms, which relied on `some kind of ordering to achieve adequate performance`. The hash function appears to be working satisfactorily. You can easily store values into unique locations and just as easily check for their existence. ***However, there are two major problems with this approach.*** Take another look at the generated values. If these were used as index positions within an array, then it would need to big enough to accommodate the largest position, 69. Having filled only 3 of the 70 positions available—that is, 0 to 69—you would still have 67 empty ones. Now imagine that the values had been 167, 162, and 169 instead—you’d end up with 167 empty slots. It would seem that this very simplistic hashing scheme is pretty inefficient in terms of storage. One way to solve this problem is to modify the hash function to produce only values within a certain range. Given the previous example, if the size of the hash table was restricted to, for example, ten positions, then the hash function could be modified to take the original result and use a modulus— the remainder after division—to find the remainder after division by 10, as shown in the following example:

```
e + l + v + i + s = 5 + 12 + 22 + 9 + 19
= 67 % 10
= 7
m + a + d + o + n + n + a = 13 + 1 + 4 + 15 + 14 + 14 + 1
= 62 % 10
= 2
s + t + i + n + g = 19 + 20 + 9 + 14 + 7
= 69 % 10
= 9
```

Now the addresses fall within the range 0 to 9 and can be stored in a hash table of size 10. So far so good. Unfortunately, there is still one more problem with the hash function as described: It suffers from a high rate of collisions—different values hashing to the same address. To illustrate what is meant by collision, the following code hashes the string “lives”. Notice that the result is the same address that was generated for “elvis” and therefore collides with an existing value:

```
l + i + v + e + s = 5 + 12 + 22 + 9 + 19
= 67 % 10
= 7
```

You’ve already seen one way to reduce the number of collisions: Increase the address space. By increasing the address space, you reduce the likelihood of a collision while at the same time increasing the amount of wasted memory. Most hashing algorithms are therefore a trade-off between efficiencies of space and time. Another way to reduce collisions is to be more selective in the choice of hash table size. It turns out that prime numbers give better results than nonprime numbers. By choosing a prime number close to the desired size, you reduce the amount of clustering and consequently the number of collisions. How and why this works is beyond the discussion of this stackoverflow thread. Ideally, though, you would like a perfect hashing algorithm —one that produces no collisions at all. Unfortunately, finding a perfect hashing algorithm is much harder than it might at first seem. For small sets of well-known input data, it may be possible to find one, but even with a very good hashing algorithm, the likelihood of finding one that produces no collisions at all is very small. A better solution is to try to reduce the number of collisions to something manageable and deal with them. The hash function discussed thus far is actually particularly poor when it comes to collisions. For one thing, the order of letters makes no difference. As you have seen already, “elvis” and “lives” both hash to the same address. In fact, anagrams—words with the same letters but in a different order—will always hash to the same value. What you need is an algorithm that somehow considers the order of the letters to be significant. An example of a fairly simple yet effective hashing algorithm is the one used in the ***String class of the JDK.*** The algorithm itself is grounded in very sound mathematics, but a proof is certainly beyond the discussion of this thread. Nonetheless, the actual implementation is pretty easy to understand. Most good hashing algorithms, like the one used in the ***JDK String class***, are based on sound mathematics. One such algorithm is the ***cyclic redundancy check (CRC)***. Many applications that compress or transmit files over networks use a CRC to ensure the integrity of the data. The CRC algorithm takes a stream of data and computes an integer hash value. One of the properties of a CRC calculation is that the ordering of the data is significant. This means that the two strings “elvis” and “lives” are almost guaranteed to hash to different values. I say “almost” here because the standard CRC isn’t perfect, and hence, there is still a nonzero (yet very small) chance of a collision. The idea is to add each letter just as you did before. This time, however, the working value is multiplied by 31 before adding each letter. The following equation shows what's involved in calculating the hash value for the string "elvis":

```
(((e * 31 + l) * 31 + v) * 31 + i) * 31 + s
```

Applying the same algorithm across all the example strings is demonstrated in the following code:

```
"elvis" = (((5 * 31 + 12) * 31 + 22) * 31 + 9) * 31 + 19
= 4996537
"madonna" = (((((13 * 31 + 1) * 31 + 4) * 31 + 15) * 31 + 14) * 31 + 14) * 31 + 1
= 11570331842
"sting" = (((19 * 31 + 20) * 31 + 9) * 31 + 14) * 31 + 7
= 18151809
"lives" = (((12 * 31 + 9) * 31 + 22) * 31 + 5) * 31 + 19
= 11371687
```

The values are wildly different from each other, and note that "elvis" and "lives" no longer collide. In addition, note how large the values are. Obviously, you aren't going to have a hash table containing 11,570,331,843 slots just to hold four strings, so just as before, you take the remainder after dividing the hash value by the hash table size (the modulus)—in this case, you'll use 11, the nearest prime number to 10—to generate the final address, as shown in the following code sample:

```
"elvis" = 4996537 % 11
= 7
"madonna" = 11570331842 % 11
= 3
"sting" = 18151809 % 11
= 5
"lives" = 11371687 % 11
= 8
```

The hash function now performs markedly better for the sample data. There are no longer any collisions, so the values can be safely stored into individual positions. At some point, though, the hash table will fill up and collisions will begin to occur. Even before it fills up, however, a collision is still very likely; this particular algorithm isn't perfect. As an example, imagine you want to add "fred" to the existing hash table. The hash value for this string is calculated: 196203 % 11 = 7, which collides with "elvis". The first option available to you is simply to increase (or possibly even decrease) the size of the hash table and recompute all the addresses using the new size. The following code shows the new hash values after resizing the hash table to 17 to store the new string "fred":

```
"elvis" = 4996537 % 17
= 16
"madonna" = 11570331842 % 17
= 7
"sting" = 18151809 % 17
= 8

"lives" = 11371687 % 17
= 13
"fred" = 196203 % 17
= 6
```

Now all of the strings have unique addresses and can be stored and retrieved successfully. However, a substantial price has been paid to maintain the uniqueness of addresses. In the preceding example, the size of the hash table was increased by 6 just to accommodate one more value. The hash table now has space for 17 strings, but only 5 have actually been stored. That's a utilization of only (5/17) * 100 = 29%. Not very impressive, and the problem can only get worse. The more strings you add, the larger the hash table needs to be in order to prevent collisions, resulting in more wasted space. As you can see, although resizing is partially useful to reduce the number of collisions, they still occur. Therefore, some other technique is required to manage the problem. The first solution to the problem of collision resolution is known as ***linear probing.*** ***Linear probing*** is a very simple technique that, on detecting a collision, searches linearly for the next available slot. And the other one being ***the bucket solution***.

Discussion to be continued only if you are interested further.

answered May 25 at 8:31
Shirgill Farhan Ansari
1,241 1 6 18

+1 nice explanation. If someone finds a way to allocate buckets non-contiguously (a prophecy!!), storage inflation can be resolved. Storing them in minimal space would then be a miracle. – Nirav Bhatt Oct 7 at 11:30

All of the answers so far are good, and get at different aspects of how a hashtable works. Here is a simple example that might be helpful. Lets say we want to store some items with lower case alphabetic strings as a keys.

As simon explained, the hash function is used to map from a large space to a small space. A simple, naive implementation of a hash function for our example could take the first letter of the string, and map it to an integer, so "alligator" has a hash code of 0, "bee" has a hash code of 1, "zebra" would be 25, etc.

Next we have an array of 26 buckets (could be ArrayLists in Java), and we put the item in the bucket that matches the hash code of our key. If we have more than one item that has a key that begins with the same letter, they will have the same hash code, so would all go in the bucket for that hash code so a linear search would have to be made in the bucket to find a particular item.

In our example, if we just had a few dozen items with keys spanning the alphabet, it would work very well. However, if we had a million items or all the keys all started with 'a' or 'b', then our hash table would not be ideal. To get better performance, we would need a different hash function and/or more buckets.

answered Apr 8 '09 at 16:41
Greg Graham
356 3 9

Here's another way to look at it.

I assume you understand the concept of an array A. That's something that supports the operation of indexing, where you can get to the Ith element, A[I], in one step, no matter how large A is.

So, for example, if you want to store information about a group of people who all happen to have different ages, a simple way would be to have an array that is large enough, and use each person's age as an index into the array. Thay way, you could have one-step access to any person's information.

But of course there could be more than one person with the same age, so what you put in the array at each entry is a list of all the people who have that age. So you can get to an individual person's information in one step plus a little bit of search in that list (called a "bucket"). It only slows down if there are so many people that the buckets get big. Then you need a larger array, and some other way to get more identifying information about the person, like the first few letters of their surname, instead of using age.

That's the basic idea. Instead of using age, any function of the person that produces a good spread of values can be used. That's the hash function. Like you could take every third bit of the ASCII representation of the person's name, scrambled in some order. All that matters is that you don't want too many people to hash to the same bucket, because the speed depends on the buckets remaining small.

answered Apr 8 '09 at 17:44
Mike Dunlavey
29.8k 7 59 90

Lots of answers, but none of them are very *visual*, and hash tables can easily "click" when visualised.

Hash tables are often implemented as arrays of linked lists. If we imagine a table storing peoples' names, after a few insertions it might be laid out in memory as below, where `()` -enclosed numbers being hash values of the text.

```
bucket#  bucket content / linked list

[0]      --> "sue"(780) --> null
[1]      null
[2]      --> "fred"(42) --> "bill"(9282) --> "jane"(42) --> null
[3]      --> "mary"(73) --> null
[4]      null
[5]      --> "masayuki"(75) --> "sarwar"(105) --> null
[6]      --> "margaret"(2626) --> null
[7]      null
[8]      --> "bob"(308) --> null
[9]      null
```

A few points:

- each of the array indices (`[0]` , `[1]`) is known as a ***bucket***, and starts a - possibly empty - linked list of values
- each value (e.g. `"fred"` with hash `42`) is linked from bucket `[hash % number_of_buckets]` e.g. `42 % 10 == [2]` ; `%` is the modulus operator - the remainder when divided by the number of buckets
- multiple data values may ***collide*** and be linked at the same bucket, most often because their hash values collide after the modulus operation (e.g. `42 % 10 == [2]` , and `9282 % 10 == [2]`), but occasionally because the hash values are the same (e.g. `"fred"` and `"jane"` both shown with hash `42` above)
    - most hash tables handle collisions - with slightly reduced performance but no functional confusion - by comparing the full text of a key being sought or inserted to each key already in the linked list

If the table size grows, hash tables implemented as above tend to resize themselves to keep the ratio of elements to buckets (aka *load factor*) somewhere in the 0.5 to 1.0 range. With load factor 1 and a cryptographic strength hash function, 36.8% of buckets will tend to be empty, another 36.8% have one element, 18.4% two elements, 6.1% three elements, 1.5% four elements, .3% five etc.. - the list lengths *average* 2.0 elements no matter how many elements are in the table, which is why we say lookup/insert/erase are O(1) constant time operations.

(Notes: not all hash tables use linked lists, but most general purpose ones do, as closed hashing (aka open addressing), particularly with erase operations supported, has less stable performance properties with collision-prone keys/hash functions).

answered Jun 1 at 6:59
Tony D
64.2k 7 73 133

How the hash is computed does usually not depend on the hashtable, but on the items added to it. In frameworks/base class libraries such as .net and Java, each object has a GetHashCode() (or similar) method returning a hash code for this object. The ideal hash code algorithm and the exact implementation depends on the data represented by in the object.

answered Apr 8 '09 at 15:52
Lucero
41.8k 4 61 109

A hash table totally works on the fact that practical computation follows random access machine model i.e. value at any address in memory can be accessed in O(1) time or constant time.

So, if I have a universe of keys (set of all possible keys that I can use in a application, e.g. roll no. for student, if it's 4 digit then this universe is a set of numbers from 1 to 9999), and a way to map them to a finite set of numbers of size I can allocate memory in my system, theoretically my hash table is ready.

Generally, in applications the size of universe of keys is very large than number of elements I want to add to the hash table(I don't wanna waste a 1 GB memory to hash ,say, 10000 or 100000 integer values because they are 32 bit long in binary reprsentaion). So, we use this hashing. It's sort of a mixing kind of "mathematical" operation, which maps my large universe to a small set of values that I can accomodate in memory. In practical cases, often space of a hash table is of the same "order"(big-O) as the (number of elements *size of each element), So, we don't waste much memory.

Now, a large set mapped to a small set, mapping must be many-to-one. So, different keys will be alloted the same space(?? not fair). There are a few ways to handle this, I just know the popular two of them:

- Use the space that was to be allocated to the value as a reference to a linked list. This linked list will store one or more values, that come to reside in same slot in many to one mapping. The linked list also contains keys to help someone who comes searching. It's like many people in same apartment, when a delivery-man comes, he goes to the room and asks specifically for the guy.
- Use a double hash function in an array which gives the same sequence of values every time rather than a single value. When I go to store a value, I see whether the required memory location is free or occupied. If it's free, I can store my value there, if it's occupied I take next value from the sequence and so on until I find a free location and I store my value there. When searching or retreiving the value, I go back on same path as given by the sequence and at each location ask for the vaue if it's there until I find it or search all possible locations

in the array.

Introduction to Algorithms by CLRS provides a very good insight on the topic.

answered Jun 12 at 5:19
div
176 4

For all those looking for programming parlance, here is how it works. Internal implementation of advanced hashtables has many intricacies and optimisations for storage allocation/deallocation and search, but top-level idea will be very much the same.

```
(void) addValue : (object) value
{
   int bucket = calculate_bucket_from_val(value);
   if (bucket)
   {
       //do nothing, just overwrite
   }
   else   //create bucket
   {
      create_extra_space_for_bucket();
   }
   put_value_into_bucket(bucket,value);
}

(bool) exists : (object) value
{
   int bucket = calculate_bucket_from_val(value);
   return bucket;
}
```

where `calculate_bucket_from_val()` is the hashing function where all the uniqueness magic must happen.

The rule of thumb is: **For a given value to be inserted, bucket must be UNIQUE & DERIVABLE FROM THE VALUE that it is supposed to STORE.**

Bucket is any space where the values are stored - for here I have kept it int as an array index, but it maybe a memory location as well.

answered Oct 7 at 11:11
Nirav Bhatt
3,951 2 20 55