Hash tables

Professors Clark F. Olson and Carol Zander

Hash tables

Binary search trees are data structures that allow us to perform many operations in O(log n) time on average for a collection of n objects (and balanced binary search trees can guarantee this in the worst case). However, if we are only concerned with insert/delete/retrieve operations (not sorting, searching, findMin, or findMax) then we can sometimes do even better. Hash tables are able to perform these operations in O(1) average time under some assumptions.

The basic idea is to use a **hash function** that maps the items we need to store into positions in an array where it can be stored. Since ASCII characters use seven bits, we could hash all three-letter strings according to the following function:

```
     hash(string3) = int(string3[0]) + int(string3[1]) * 128 + int(string3[2]) * 128 * 128
```

This guarantees that each three-letter string is hashed to a different number.

In the ideal case, you know every object that you could ever need to store in advance and you can devise a "perfect" hash function that maps each object to a different number and use each number from 0 to n-1, where n is the number of objects that you need to store. There are cases where this can be achieved. One example is hashing keywords in a programming language. You know all of the n keywords in advance and sometimes a function can be determined that maps these perfectly into the n values from 0 to n-1. This is very useful in implementing a compiler. After each word is read from a program file, you need to determine whether it is a keyword or a user-defined word. A perfect hash table allows you to look up the unique spot that the word would be stored if it was a keyword. You then check whether it matches the word stored there to see whether it is a keyword or not. This operation is O(1) in the number of keywords, rather than O(log n) to perform binary search.

Unfortunately, finding a perfect hashing function is not always possible. In particular, you may not know all of the possible values in advance. In addition, the range of possible values may be quite large. If we must consider all possible three-letter strings, then there are 1283 (more than 2 million) possibilities. The solution is to use a hash function that limits the values to a reasonable range. A common technique is to use the mod operator to take the remainder after dividing by the size of the table.

A minor problem is that you may not fill the entire array. This is minor problem because the array usually just stores pointers to objects, so not that much space is wasted for NULL pointers. A hash table might have 10,000 (or more) *buckets* (spaces to store objects). Even it is sparsely filled, it can waste at most 40,000 bytes, which is reasonable for a general-purpose computer.

A more significant problem is the possibility of **collisions**. If you can't guarantee the uniqueness of the hash function, then you might end up with two (or more) objects hashed to the same bucket. A good hash function has the following properties:

- Objects are well distributed in the hash table. Ideally, no two objects hash to bucket. If this can't be guaranteed, then we want buckets in the hash table to be equally likely when a new object is inserted.
- The hash function is efficient to compute. It should be O(1) with a small constant factor.

An example of a poor hashing function is the sum of the ASCII values in string that are about the same length. These would cluster strongly at (relatively) small values. A large table would not be evenly filled.

If you can't guarantee that objects hash to unique buckets, then handling collisions is the most important aspect of hashing. There are two general techniques for handling collisions: closed hashing (or sometimes just *hashing*) and open hashing (also called *separate chaining*).

Closed hashing (also called open addressing)

When hashing, multiple objects cannot be placed in the same bucket. This means that, when a collision occurs, another bucket for the new object to be stored in must be found. There are three common methods of doing this:

1. Linear probing
2. Quadratic probing
3. Double hashing

Each case modifies the bucket to examine after some number of collisions. The current attempt uses the hash function h(x) and a probing distance function D(i), where i is the number of collisions in the current insert/ retrieve/delete attempt. For a hash table with B buckets, the next bucket to examine is `( h(x) + D(i) ) mod B`. The size of the hash table, B, should be at least twice as big as the number of items you expect it to hold and be a prime number. Both characteristics aid in limiting collisions.

Linear probing

With linear probing, we look for the next open bucket to place the value in, i.e., the probing function D(i) = i. Formally, let's say that our hashing function is h(x). If when trying to insert the current object, we have collided with *i* filled buckets, then the next bucket that we try is:

```
        ( h(x) + i ) mod B
```

For example, we want to hash the following numbers in a hash table with 10 buckets using `h(x) = x mod 10`: `45 13 34 67 23 74`. (Note that the poor choice size of 10 – too small and not a prime number – is used for simplicity of arithmetic.) Using h(x) gives the following values:

h(45) = 5
h(13) = 3
h(34) = 4
h(67) = 7
h(23) = 3 [collision 1: try 3 + D(1) = 3+1 = 4; collision 2: try 3 + D(2) = 3+2 = 5; collision 3: try 3 + D(3) = 3+3 = 6]
h(74) = 4 [collision 1: try 4 + D(1) = 4+1 = 5; collision 2: try 4 + D(2) = 4+2 = 6; collision 3: try 4 + D(3) = 4+3 = 7; collision 4: try 4 + D(4) = 4+4 = 8]

We end up with:

			13	34	45	23	67	74	
0	1	2	3	4	5	6	7	8	9

Linear probing is not ideal because of the clustering effect that tends to fill up certain regions of the hash table (this is called primary clustering). If we assumed each probe is equally likely to result in a collision (an incorrect assumption owing to primary clustering), then the average number of cells examined in an insertion is 1/(1-λ), where λ is the load factor and is equal to the number of elements in the table divided by the number of buckets in the table, n/B. So, even with our assumption, this doesn't give us O(1) time. (We can transform 1/(1-λ) into B/(B-n).) A careful analysis that considers primary clustering indicates that the average number of probes for an insertion is actually $(1 + 1/(1 - \lambda)^2)/2$. This is also the average number of probes performed in an unsuccessful search. A successful search takes $(1 + 1/(1 - \lambda))/2$ probes on average.

Overall, with linear probing, it is important that the hash function distributes objects well and that the table is large enough to prevent primary clustering from causing many extra probes. This is usually the case for a table that is twice as large as the number of items it stores.

Quadratic probing
With quadratic probing, we use a quadratic probing function $D(i) = i^2$. So, after *i* collisions, we examine the bucket at $(h(x) + i^2) \bmod B$.

Using the same values as above, 45 13 34 67 23 74, gives:
h(45) = 5
h(13) = 3
h(34) = 4
h(67) = 7
h(23) = 3 [collision 1: try 3 + D(1) = $3+1^2$ = 4; collision 2: try 3 + D(2) = $3+2^2$ = 7;
collision 3: try 3 + D(3) = $3+3^2$ = 12, mod by 10 gives 2]
h(74) = 4 [collision 1: try 4 + D(1) = $4+1^2$ = 5; collision 2: try 4 + D(2) = $4+2^2$ = 8]

		23	13	34	45		67	74	
0	1	2	3	4	5	6	7	8	9

Quadratic probing usually ends up with fewer collisions, although second clustering can occur if many objects hash to the same bucket (before probing). However, whereas with linear probing a non-prime table size doesn't cause problems, with quadratic probing, the size of the hash table should be a prime number. The reason for this is that if the size is a non-prime, the sequence of buckets examined using the quadratic probing function may repeat before many of the buckets have been examined. In the above case, if we kept running into collisions, we would examine the following sequence of offsets:

1 4 9 6 5 6 9 4 1 0 1 4 9 6 5 6 9 4 1 0 …

Note that offsets 2, 3, 7, and 8 are never considered. A table size of 16 is even worse; only offsets of 0, 1, 4, and 9 would ever be considered.

When B is a prime, we are guaranteed that the first B / 2 probes will be to distinct buckets. This leads us to conclude that we should make sure that the size of the table is always at least twice the number of objects stored. If this is violated, we must expand the table and rehash all of the items. Moreover, prime numbers are good to use since performing addition modulo a prime tends to distribute objects in the hash table better.

Note that the quadratic probing buckets can be computed more efficiently than computing i^2 since $i^2 = (i-1)^2 + 2i - 1$. The analysis of the average number of probes required for quadratic probing is not completely understood, but it is better than linear probing.

Double hashing
To eliminate even secondary clustering, we can use double hashing. In this case, we use two hash functions, $h_1(x)$ and $h_2(x)$. After *i* collisions for the current object, we probe the bucket with index:

$(h_1(x) + D(i)$ where $D(i) = i * h_2(x)) \bmod B$

Typically, `h2(x) = r – (x mod r)` is used where r is a prime number smaller than B. Note that $h_2(x)$ can never be zero. A popular choice for `r` is `r = 7`. Using the same values, 45 13 34 67 23 74, with r = 7, yields:

$h_1(45) = 5$
$h_1(13) = 3$
$h_1(34) = 4$
$h_1(67) = 7$
$h_1(23) = 3$ [collision, $h_2(23) = 7 - (23 \bmod 7) = 7 – 2 = 5$;
collision 1: try $3 + D(1) = 3 + 1*5 = 8$]
$h_1(74) = 4$ [collision, $h_2(74) = 7 - (74 \bmod 7) = 7 – 4 = 3$;
collision 1: try $4 + D(1) = 4 + 1*3 = 7$;
collision 2: try $4 + D(2) = 4 + 2*3 = 10$; mod by 10 gives 0]

74			13	34	45		67	23	
0	1	2	3	4	5	6	7	8	9

Attempting to Insert one more value, 58, demonstrates the problem with a non-prime table size:

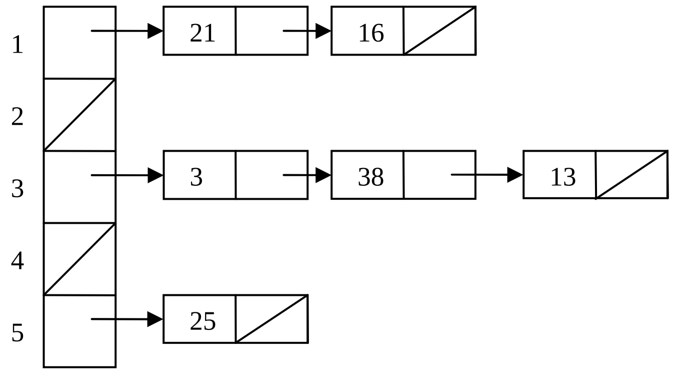
$h_1(58) = 8$ [collision, $h_2(58) = 7 - (58 \bmod 7) = 7 – 2 = 5$;
collision 1: try $8 + D(1) = 8 + 1*5 = 13$, mod by 10 gives 3;
collision 2: try $8 + D(2) = 8 + 2*5 = 18$, mod by 10 gives 8;
collision 3: try $8 + D(3) = 8 + 3*5 = 23$, mod by 10 gives 3;
collision 4: try $8 + D(4) = 8 + 4*5 = 28$, mod by 10 gives 8;
collision 5: try $8 + D(5) = 8 + 5*5 = 33$, mod by 10 gives 3;
collision 6: try $8 + D(6) = 8 + 6*5 = 38$, mod by 10 gives 8; And so on …]

Double hashing is a little more complex than quadratic probing and it is not always an improvement, given the additional overhead. Quadratic probing appears to be useful in most cases. The most significant drawback is the necessity for the hash table to be half empty. By using another function to resolve collisions, there are typically fewer collisions using double hashing.

Lazy deletion
When we perform a retrieve in a hash table, we must continue probing until we find the object or we find an empty cell. What happens if we delete an object that previously caused a collision? This would prevent us from finding the object that collided with the deleted object, since it would appear that it should be in the spot where the deleted object was. To fix this, we use a process called lazy deletion. This process marks items as deleted, rather than actually removing them from the table (using an extra data member for each bucket). This allows us to reconstruct the probing sequence for any object in the table, even if objects that it collided with have been deleted. It also allows that element to be used in a later insertion.

Open hashing (also called separate chaining)

In open hashing each bucket of the hash table stores a collection of the objects that hash to that bucket. This is an array of lists representation. The buckets can be other structures such as trees or even another hash table.



Collisions are relatively cheap with open hashing. In fact, we can allow the hash table to store more items than there are buckets without a big loss in performance. If the size of the table is O(n), then a good hashing function will yield O(1) time lookups (on average). We would still prefer to have the number of buckets be about as large as the number of objects in the table. Note that this doesn't actually reduce the complexity of table operations unless the size of the hash table grows with the number of objects stored, but it can still achieve a large constant factor improvement.

Summary

Hashing is very useful for quick retrieval. One use of hashing is to look up "dummy" objects of derived classes in a hash table according to some descriptor ("D" for deposit). The member operations of the dummy values (such as clone or create) can be used without worry about what is stored in the data members. This allows you to create on object of some derived class and refer to it with a base class pointer. Let's say that we have ten classes that derive from the same base class and we don't know which we want to create until, for example, we read a character from a file and that character specifies which type to create. One strategy would be to implement "create" member operations for each of the derived classes and store an object of each class in a table by hashing the type (using, for example, the letter that specifies the type). We can create our new object, by performing something like:

```
      BaseClass*  p = Table[hash(type)]->create();
```

This is called a factory. It is a factory and also a hash table that pumps out objects. This ties in nicely with the Open Closed Principle since this code does not need to change, even if we add/delete/change the types of objects that we use. This is also more efficient than checking each possible type to determine which kind of object we have.