Data Structures & Programming

Hash Tables

Golnar Sheikhshab

Hash table: key-based address computation

There is a **bucket array** of **capacity** N.

There is a **hash function** that decides which bucket should contain the entry with **key k**.

A **good hash function** distributes (spreads) items evenly and avoids **collision** as much as possible.

There are also **collision handling** schemes because collision can not be completely avoided

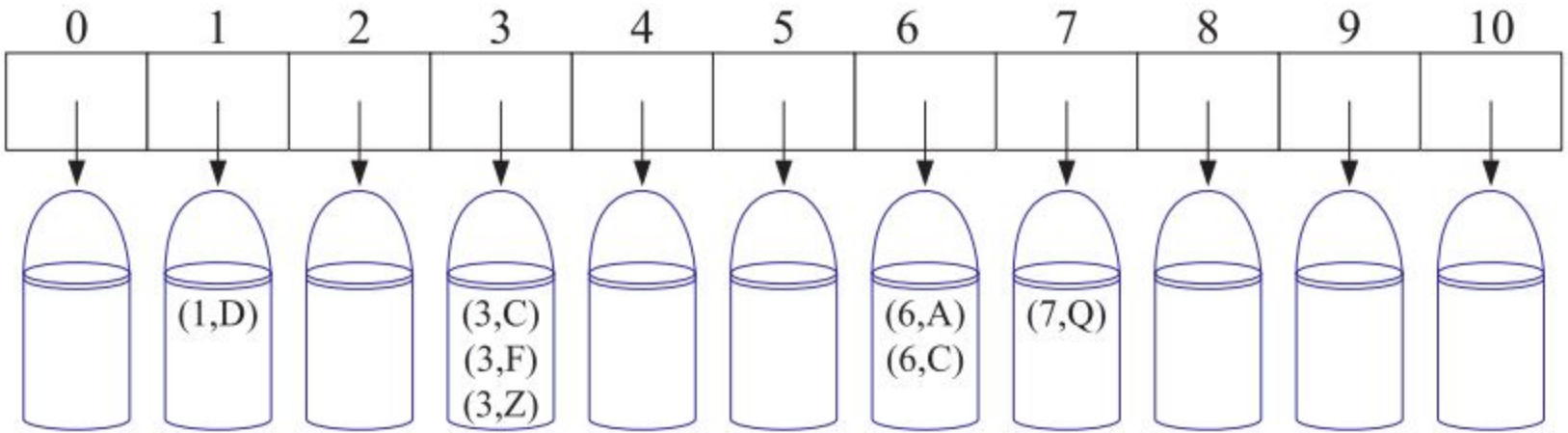

Figure 9.2: A bucket array of size 11 for the entries (1,D), (3,C), (3,F), (3,Z), (6,A), (6,C), and (7,Q).

Hash function

The second part of the hash table structure (the first part is bucket array)

Maps the key to [0, N-1] (an index in the bucket array)

The goal of hash function is to minimize **collision**

Has two parts:

- Hash code : that maps the key to an integer (unbounded)
- Compression : that maps the hash code into [0, N-1]

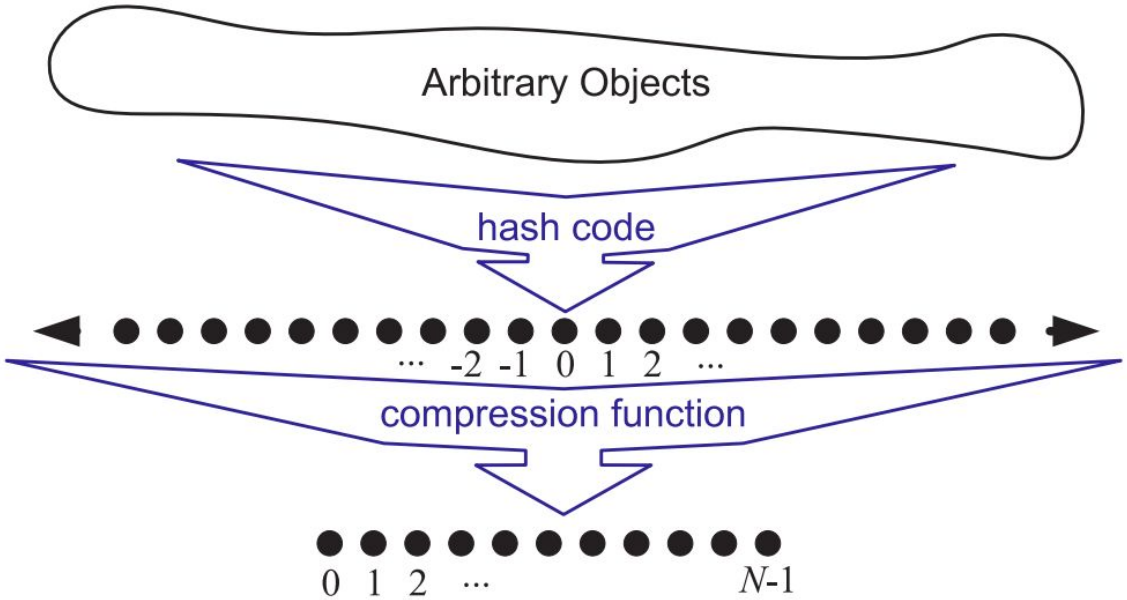

Figure 9.3: The two parts of a hash function: hash code and compression function.

Hash Code

The hash code of a key is an integer.

It doesn't need to be in [0, N-1]. It might even be negative!

If the hash code of two keys are the same, we can not avoid collision! So, hash codes must try really hard to avoid collision.

If k1 = k2 => hash code of k1 must be equal to hash code of k2

Possible hash codes

1. Convert to integer

    This is good for char, short, int. There won't be any collision.

    But for long? We will be ignoring half of the bits. (We are **casting down**.)

2. Sum of components

    The object is seen as a k-tuple of integers (X0, X1, ..., Xk-1) and hash code will be the sum of the components.

    Works better for long.

    Not so good for strings ("stop" and "tops" and "pots" will collide).

Possible hash codes (continue)

3. Polynomial Hash Codes

The object is still seen as a k-tuple of integers (X0, X1, ..., Xk-1).

h(k) = $x_0 a^{k-1} + x_1 a^{k-2} + \cdots + x_{k-2} a + x_{k-1}$

= $x_{k-1} + a(x_{k-2} + a(x_{k-3} + \cdots + a(x_2 + a(x_1 + a x_0)) \cdots))$.

a is a constant.

Bad choice for a? 2^14 or -2^15

Good choices for a? numbers with non-zero low-order bits
for example 33, 37, 39, and 41 (empirically found for over 50K English words)

Possible hash codes (continue)

4. Cyclic Shift Hash Codes

Assuming integer is 32 bits:

```
int hashCode(const char* p, int len) {          // hash a character array
  unsigned int h = 0;
  for (int i = 0; i < len; i++) {
    h = (h << 5) | (h >> 27);                   // 5-bit cyclic shift
    h += (unsigned int) p[i];                   // add in next character
  }
  return hashCode(int(h));
}
```

Shift	Collisions		Shift	Collisions	
	Total	Max		Total	Max
0	23739	86	9	18	2
1	10517	21	10	277	3
2	2254	6	11	453	4
3	448	3	12	43	2
4	89	2	13	13	2
5	4	2	14	135	3
6	6	2	15	1082	6
7	14	2	16	8760	9
8	105	2			

Table 9.1: Comparison of collision behavior for the cyclic shift variant of the polynomial hash code as applied to a list of just over 25,000 English words. The "Total" column records the total number of collisions and the "Max" column records the maximum number of collisions for any one hash code. Note that, with a cyclic shift of 0, this hash code reverts to the one that simply sums all the characters.

Possible hash codes (continue)

Hashing Floating-Point Quantities

Casting float to int? Bad! You'll be throwing information out even if they have the same number of bits.

You should interpret the bit sequence as int instead using **reinterpret cast**

warning: reinterpret cast is not portable (the result depends on the particular machine's encoding of types as a pattern of bits).

```
int hashCode(const float& x) {                    // hash a float
  int len = sizeof(x);
  const char* p = reinterpret_cast<const char*>(&x);
  return hashCode(p, len);
}
```

Compression Functions

Map the hash code to [0, N-1]

1. Division method + picking a **prime number** as **N**

$$h(k) = |k| \bmod N$$

2. The MAD method (multiply, add, and divide) + picking a **prime number** as **N**

$$h(k) = |ak + b| \bmod N$$

a (!=0) and b are constants that are chosen randomly

Dealing with collision

We can't avoid collision altogether. So, we have to deal with it.

We can't directly put (K, V) into A[h(K)] because there might be collision.

Ways to deal with the collision:

1. Separate Chaining
2. Open addressing (umbrella name for a set of methods)

Separate Chaining

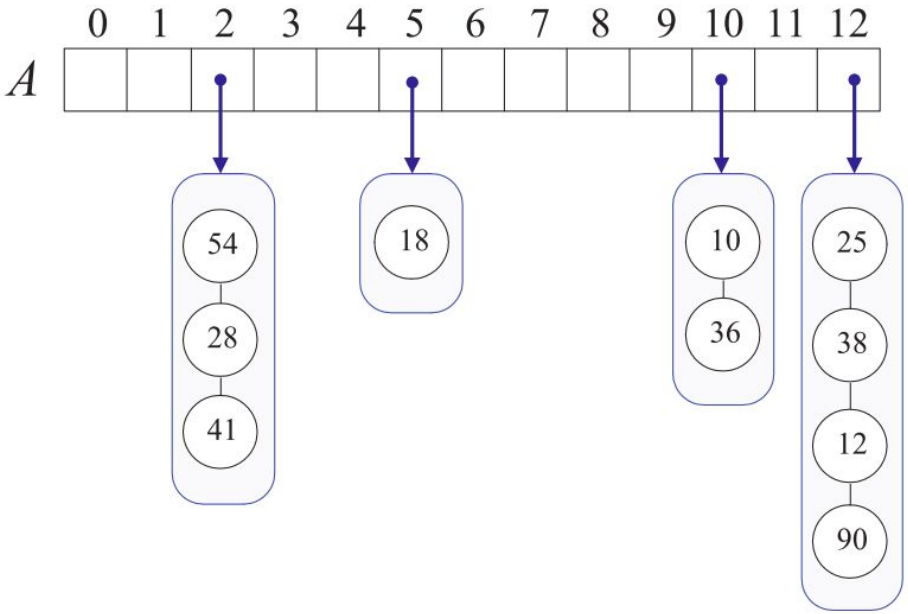

Figure 9.4: A hash table of size 13, storing 10 entries with integer keys, with collisions resolved by separate chaining. The compression function is $h(k) = k \bmod 13$. For simplicity, we do not show the values associated with the keys.

Algorithm find(k):

Output: The position of the matching entry of the map, or end if there is no key k in the map

return $A[h(k)]$.find(k) {delegate the find(k) to the list-based map at $A[h(k)]$}

Algorithm put(k, v):

$p \leftarrow A[h(k)]$.put(k, v) {delegate the put to the list-based map at $A[h(k)]$}
$n \leftarrow n + 1$
return p

Algorithm erase(k):

Output: None

$A[h(k)]$.erase(k) {delegate the erase to the list-based map at $A[h(k)]$}
$n \leftarrow n - 1$

Code Fragment 9.5: The fundamental functions of the map ADT, implemented with a hash table that uses separate chaining to resolve collisions among its n entries.

Load factor

When we use separate chaining, the maximum size of each bucket is called load factor. If the hash function is good, the expected load factor is n/N.

Load factor should be bounded by a small constant (preferably below 1).

The functions find, put, and erase in a map implemented with a hash table is O(load factor). So, if n=O(N), these functions will be of O(1).

Open Addressing

BucketArray is not an array of buckets anymore. It's an array of key-value pairs. We directly insert key-value pairs into it.

Open Addressing is a collective name for

- Linear Probing
- Quadratic Probing
- Double Hashing

Linear probing

If A[h(k)] is occupied, check A[(h(k)+i) mod N] for i=1....N until you find the key or and empty cell.

Do this for both find and put functions.

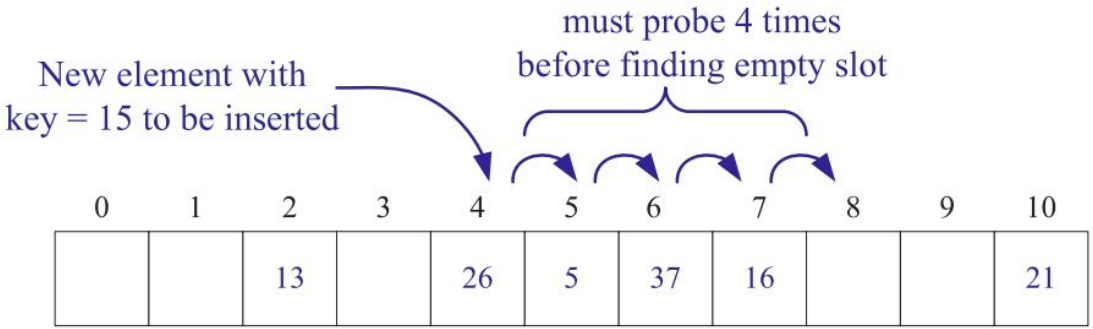

Figure 9.5: An insertion into a hash table using linear probing to resolve collisions. Here we use the compression function $h(k) = k \bmod 11$.

Quadratic probing

Linear probing leads to entry clustering. A collision increase the chance of another collision and the next collision increases the chance of the next collision even more.

To reduce this effect, we do quadratic probing.

Everything is the same as before but we check A[h(k)+j^2] for j=1...N if A[h(k)] is occupied.

Clustering still happens but not as bad as in linear clustering.

Double Hashing

To avoid clustering altogether, we throw in another hash function h' as follows:

If A[h(k)] is already occupied, we check A[h(k)+j*h'(k)] for j=1....N

As a result, the number of cells we skip depends on the key k.

h' should not be 0 for any k.

A common choice for h'(k) = q−(k mod q), for some prime number q < N.

Deleting in open addressing

In remove, should we try shifting entries? If so, which entries? What if they were inserted into the right place? This is too difficult.

Instead, we put a special "available" symbol there.

When we are searching for a key, we treat the "available" symbol as an occupied cell. But when we insert, we treat it as an empty cell.

This means the put function (which should replace or insert if key was absent) should remember where it saw an "available" cell while searching for the key.

Open addressing saves space but it complicates removal

Separate Chaining vs. Open Addressing

Open addressing saves space but is not necessarily faster.

In experimental settings separate chaining is competitive or faster (depending on the load factor).

So, if space is not scarce, opt for separate chaining.

Load Factors and Rehashing

If load factor is too high the map operations start being too slow O(N). For example in quadratic probing, if load factor is > 0.5 you may need to check N/2 cells (and that will be all the cells you can check) to find a key.

In these cases, **rehashing** into a new table and with a new hash function could be a good idea. The new table is usually doubles in size.

Rehashing involves computing the hash function for all items in the hash table and inserting them again. This by itself is O(N) but just like in expandable arrays this cost will be **amortized** over time when new put, find, remove functions take expected time of O(1).

STL map vs. STL unordered_map

STL map is based on a balanced binary search. The keys in STL map are always ordered but the functions are O(log n).

STL unordered_map however, is based on hash functions and the expected time complexity is amortized O(1).

When using STL map, all that matters is that the key has an operator<.

When using STL unordered_map, a hash function must be defined for the key type. Therefore, you can't use unordered_map with a key of type pair<int,int> unless you define a hash function for this type.

Because I don't want you to give up at a certain point when you can go further:

https://www.youtube.com/watch?v=-sUKoKQlEC4

Reading Material

Sections 9.2.1 -- 9.2.6 of the textbook