HashTables and Collisions

Hash Table/Map

A map is basically an ADT that contains a collection of <key, value> pairings. It basically maps a key to a value and enables O(1) insertion, searching and deletion. This is crazy, but that's how it is.

The main operations are:

- Retrieve data
- Insert data
- Delete data

A **direct addressing table** is the simplest form of a Hash Table. Basically, we directly use the value as the key without any hashing. The information on bus service 804 will go straight to index 804 of the DAT array.

To overcome limitations such as the need to keep a huge array for a potentially small set of <key, value> pairings, or the inability to store Strings as keys, hashing is required. Hashing helps to map larger integers to smaller integers, as well as non-integer keys to integers. This will be done with the help of Hash functions.

HashMap Implementation: Arrays (Not covered)

The simplest way to implement a HashMap is to index straight into an array with the hash key generated. This allows for O(1) lookup as well as O(1) insertion and deletion. The array will contain HashNodes in each slot, as we need to store both the keys and values together in the node.

Java's HashMap and HashTable are implemented using a special LinkedList (different from the usual LinkedList).

Hash Function Criteria:

A good hash function has the following criteria:

- Fast to compute
- Scatter keys evenly throughout the hash table
- Less collisions
- Needs less slots (space)

A bad hash function, on the other hand, depends on:

- Select Digits/Letters – e.g. choose the 4th and 8th digits of a phone number
    - hash(677**5**437**8**) = 58
    - hash(634**9**782**0**) = 90

A perfect hash function is a one-to-one mapping between keys and hash values. Hence no collision occurs. This is only possible if all keys are known. A minimal perfect hash function can create perfect hash tables, where the table size is the same as the number of keywords supplied.

A uniform hash function distributes keys evenly in the hash tables.

Hash Function Method

Flooring is a possible way to hash, by calculating a certain value's position with respect to the whole range of numbers, then finding out the whole number in the hash table that maps to that position. However, flooring is an expensive operation.

Multiplication is also possible, but it is also very expensive.

- Multiply the value by a constant real number A between 0 and 1
- Extract the fractional part
- Multiply by m, the hash table size
- hash(k) = floor(m(kA - floor(kA)))

The reciprocal of the golden ratio = (sqrt(5) - 1)/2 = 0.618033 seems to be a good choice for A (recommended by Knuth).

Modulus is a fast operation and is hence the **preferred method**. This is done through hash(k) = k%m.

An example of a hash function for a String:

```java
public int hashCode() {
  int h = hash;
  if (h == 0 && value.length > 0) {
      char val[] = value;
      for (int i = 0; i < value.length; i++) {
          h = 31 * h + val[i];
      }
      hash = h;
  }
  return h;
}
```

How do we decide on m (hash table size)? If *m* is power of two, say 2^n, then key modulo of m is the same as extracting the last n bits of the key. If m is 10^n, then our hash values is the last n digit of keys.

Rule of thumb: Pick a prime number close to a power of two to be m. Else, if there are many keys that have identical last two digits, then they will all be hashed to the same slot, resulting in many collisions.

Some really bad Hash Functions

Here are some bad hash functions from the tutorial:

- Multiplying by a factor before dividing by a multiple
  - h(key) = (key * 7) % 49
  - The keys will all be multiples of 7
- Using square root
  - The keys will be non-uniformly distributed, with more keys mapping to larger number slots
- Hashing with email addresses
- Using random in hashing – non-deterministic

Java HashMap

This class implements a hash map, which maps keys to values. Any non-null object can be used as a key or as a value e.g. we can create a hash map that maps people names to their ages. It uses the names as keys, and the ages as the values.

The AbstractMap is an abstract class that provides a skeletal implementation of the Map interface.

Generally, the default load factor (0.75) offers a good trade-off between time and space costs. The default HashMap capacity is 16.

Collision

Due to the fact that we are hashing a large number of items to a smaller set of keys, we may encounter collisions, which is when more than one item gets mapped to the same key.

This occurs more frequently than we would intuitively expect, as seen from Von Mises Paradox, where you only need 23 in a room to have >50% chance of at least 2 people sharing the same birthday.

In this case, we will need to resolve this collision in a way that still ensures efficiency.

Criteria of a Good Collision Resolution Method

- Minimize clustering
- Always find an empty spot if it exists
- Give different probe sequences when two initial probes are the same (i.e. no secondary clustering)
- Fast

Separate Chaining

The most straightforward method is to keep storing the collided keys into the same spot through the use of a Linked List. The slot in the array would be containing the head of a Linked List, where new <key, value> pairings can be inserted accordingly.

Separate Chaining Implementation: Array of Linked Lists

So why an array of linked lists, and not an array of arrays? This is because a LinkedList allows for overall faster insertion, searching and deletion.

Let's say there are x items stored at the same hash slot. We would thus need to iterate through all x items to find a specific item for linked lists, while we can use binary search for array, allowing us to find the item faster.

However, the maintenance of a sorted array would be very difficult due to the need to constantly shift items back and forth. On the other hand, linked list's insertion and deletion is a lot faster.

Furthermore, linked list allows us to not waste space unnecessarily, which may occur when we initialize arrays without more than 1 element inside.

The average size of the linked lists would be thus the load factor α = number of items/size of table.

Rehashing due to Decreasing Efficiency

To ensure that the efficiency does not worsen past a certain point, aka the load factor does not exceed a certain value, we can reconstruct the whole hash table and rehash all keys into a bigger table.

Linear Probing

Basically, when a slot is already occupied, keep adding 1 to the slot's index until we find an empty slot. Finding a specific key will also use the same method.

Deletion

Due to the fact that our finding algorithm will keep searching linearly until an empty space is found before stopping, we cannot simply remove a key from the hash table. A way to indicate that a specific key is "removed" will be required.

Lazy Deletion

This is also known as lazy deletion, where we use 3 different states:

- Occupied
- Occupied but deleted
- Empty

Primary Clustering

Due to the above logic, consecutive occupied slots may appear, also known as primary clustering. This increases the time needed for finding, inserting and deleting.

Modified Linear Probing

To avoid primary clustering, we can use a probe sequence as follows:
hash(key)
*(hash(key)+1*d)%m*
*(hash(key)+2*d)%m*
*(hash(key)+3*d)%m*
where d is some constant integer >1 and is co-prime to m.
Note: Since *d* and *m* are co-primes, the probe sequence covers all the slots in the hash table.

However, any insertion into the same starting position will result in the same probing sequence.

Quadratic Probing

This is just like linear probing except that instead of linear increments to the index, we add square numbers instead.

Theorem of Quadratic Probing

The theorem states that if $\alpha < 0.5$, and m is prime, then we can always find an empty slot (m is the table size and α is the load factor). This means that the hash table is less than half full.

However, insertion into a hash table that is more than half full may be impossible, and results in non-terminating probing.

Non-terminating Probing Solution

The solution to this is to stop probing after x probes or use another probing method when the load factor is >0.5. Another way is to resize the table.

Secondary Clustering

If two keys have the same initial position, their probe sequences are identical. This results in secondary clustering, but it is still not as bad as linear probing. The solution to this issue is the same as that of primary clustering.

Double Hashing

Double Hashing uses a secondary hash function to calculate the number of slots to jump each time a collision occurs.
hash(key)
*(hash(key)+1*hash$_2$(key))%m*
*(hash(key)+2*hash$_2$(key))%m*
*(hash(key)+3*hash$_2$(key))%m*

The secondary hash function **must not** evaluate to 0. This results in probing of the same slot over and over again. The solution is to change the hash function to be as such:
hash$_2$(key) = 5–(key%5)

Analysis

If the result of the secondary hash function evaluates to 1, then it is no different from linear probing. Any other evaluation is the same as modified linear probing. The difference is that double hashing allows every key to have a different modified linear probing process, thus reducing any clustering to the minimum.