

Hash Tables

1 Key-Value Pairs

Suppose we want to store data consisting of pairs of **keys** and **values**. For example:

name	number
Smith	123512
Jones	174322
Jackson	192852

We want to be able to **look up** the values using the keys, and

- The total amount of data might be very large
- Our algorithms might perform many look ups

More

- It would be easier to find if the list was sorted, this would mean that a guess could be made, then use binary search to get closer to the correct data. This requires $\log(N)$ operations, rather than N , as you would with an unsorted list
- A good way to optimise this would be to take the first 3 letters of each name, and convert to the 3 numbers at the start of the name, then insert the data at that point. This would make finding the data easy as the conversion could just be done again

2 Hash Tables

- A **hash table** consists of a bucket array and a hash function
- A **bucket array** for a hash table is an array A of size N , where each cell of A is thought of as a **bucket** storing a collection of key-value pairs
- The size N of the array is called the **capacity** of the hash table
- A **hash function** h is a function mapping each key k to an integer in the range $[0, n-1]$, where N is the capacity of the hash table (To make an address for each key)
- The main idea is to use $h(k)$ as an index into the bucket array A . That is, we store the key-value pair (k, v) in the bucket $A[h(k)]$
- If there are two keys with the same hash value $h(k)$, then two different entries will be mapped to the same bucket in A . In this case, we say that a collision has occurred

Questions

- Can there be entries in the hash table with the **same key**? - No (Entries would be indistinguishable)
- Can there be entries in the hash table with the **same value**? - Yes
- Can there be entries in the hash table with the **same key and same value**?

How a hash function works:

- A hash function is usually specified as the composition of two functions:
 - hash code: keys to integers
 - compression function: integers to $[0, N-1]$
- The hash code is applied first, and the compression function is applied next onto the result

The goal of the hash function is to **disperse** the keys in an apparently random way

3 Hash functions

Ideal goal: Scramble the keys **uniformly**

More practically:

- Hash function should be **efficiently computable**
- Each table position **equally likely** for each key

Some compression functions:

- **division**: take integer mod N
- **multiply add and divide (MAD)** y maps to $ay + B \bmod N$. Where a and b are nonnegative integers

3.1 Collisions

- There are several ways to deal with **collisions**. Whichever method we choose to deal with them, a large number of collisions reduces the performance of the hash table
- A good hash function minimises the collisions as much as possible

We will discuss four different methods for handling collisions:

- Separate chaining
- Linear probing
- Quadratic probing
- Double hashing

3.2 Separate Chaining

Each bucket $A[i]$ stores a list holding the entries (k, v) such that $h(k) = i$

Separate chaining performance:

- Cost is proportional to length of list
- Average length = N/M (N is amount of data, M is size of array)
- Worst case: all keys hash to the same list

More

- Apply hash function (mod length) to each key
- Whenever two things go in the same place in the array, create a linked list inside the bucket

3.3 Linear Probing

The other three methods, called open addressing schemes, store at most one entry to each bucket.

- In linear probing, if we try to insert an entry (k, v) into a bucket $A[i]$ that is already occupied, where $i = h(k)$, then we try next at $A[(i + 1) \bmod N]$.
- If $A[(i + 1) \bmod N]$ is also occupied, then we try $A[(i + 2) \bmod N]$, and so on, until we find an empty bucket that can accept the new entry.
- The name linear probing comes from the fact that accessing a cell of the bucket array can be viewed as a probe.

Description of method:

- When you have a collision, just go to the next available place (Mod N , so loop round)

- In long lists, clusters of data are formed

Costs:

- Insert and search cost depend on length of cluster.
- Average length of cluster is N/M .
- Worst case: all keys hash to same cluster

3.4 Quadratic Probing

- Quadratic probing iteratively tries the buckets $A[(i + f(j)) \bmod N]$, for $j = 0, 1, 2, \dots$, where $f(j) = j^2$ until finding an empty bucket.
- Quadratic probing avoids clustering patterns that occur with linear probing. However, it creates its own kind of clustering, called secondary clustering.
- The quadratic probing strategy may not find an empty bucket in A even if one exists.

Method:

- Trying $i, i+1, i+9, \dots$
- Other than that it is the same as linear probing
- The motivation for this is to jump quickly out of clusters
- However this creates clusters in squares

3.5 Double Hashing

- In this approach, we choose a secondary hash function, h' , and if h maps some key k to a bucket $A[i]$, with $i = h(k)$, that is already occupied, then we iteratively try the buckets $A[(i + f(j)) \bmod N]$, for $j = 0, 1, 2, \dots$, where $f(j) = jh'(k)$
- A common choice of secondary hash function is

$$h'(k) = q - (k \bmod q)$$

for some prime number $q < N$.

- The secondary hash function should not evaluate to zero.

Method:

- First try i , then $i+h'(k)$ then $i+2 \times h'(k)$

3.6 Comparison

- The open addressing schemes are more memory efficient compared to separate chaining.
- Regarding running times, in experimental and theoretical analyses, the separate chaining method is either competitive or faster than the other methods.
- If memory space is not a major issue, the collision-handling method of choice is separate chaining.

3.7 Deletions

- **Deletions** from the hash table must not hinder future searches. If a bucket is simply left empty this will hinder future probes.
- But the bucket should not be left unusable.
- To solve these problems we use **tombstones**: a marker that is left in a bucket after a deletion.
- If a tombstone is encountered when searching along a probe sequence to find a key, we know to continue.
- If a tombstone is encountered during insertion, then we should continue probing (to avoid creating **duplicates**), but then the new record can be placed in the bucket where the tombstone was found.
- The use of tombstones lengthens the average probe sequence distance.
- Two possible remedies:
 - Local reorganization: after deleting a key, continue to follow the probe sequence of that key and move records into the vacated bucket. (This will not work for all collision resolution policies).
 - Periodically rehash the table by reinserting all records into a new hash table. And if you have a record of which keys are accessed most these can be placed where they will be found most easily.

This is describing a dictionary in Python