Topic 4: Hash Tables

Eamonn Bell

eamonn.bell@durham.ac.uk

Key-Value Pairs

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

name	student number
Smith	123512
Jones	174322
Jackson	192852

Key-Value Pairs

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

name	student 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.

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 as a bucket storing a collection of key-value pairs.
- The size N of the array is called the capacity of the hash table

.

Hash Tables

- 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.
- 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.

Hash Tables

- Can there be entries in the hash table with the same key?

Hash Tables

- Can there be entries in the hash table with the same key?
- Can there be entries in the hash table with the same value?

Hash Tables

- 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 on the result,

Hash Tables

- 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 on the result,

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

Hash Functions

Ideal goal: scramble the keys uniformly.

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

Hash Functions: 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.

Hash Functions: 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

1. separate chaining
2. linear probing
3. quadratic probing
4. double hashing

Methods 2. through 4. are examples of open addressing policies for collision resolution.

Separate Chaining

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

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 same list.

Linear Probing

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

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 $A[(i+1) \bmod N]$.

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.

Linear Probing: 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.

Quadratic probing

- Quadratic probing iteratively tries the buckets

  $A[(i + f(j)) \bmod N]$, for $j = 0, 1, 2, \ldots$, 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.

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 $j = 0, 1, 2, \ldots,$ where $f(j) = j \cdot h'(k)$

- 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.

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.

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.

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.

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 placed in the bucket where the tombstone was found.

Deletions

- 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.