Hashing

Data Structures
Lecturer: Caio Fonseca

Introduction

- This section introduces the general principles of "hashing".
- Hashing is an address calculation technique that can be used to improve the speed and efficiency of access to data stored in data structures.
- Hashing offers an alternative to (inefficient) linear/sequential search techniques by calculating where data is going to be stored and going directly to that location for quick retrieval.

Hash Table

- A "hash table" is a central concept of hashing.
- A hash table is merely a fixed size array containing stored items/elements/buckets.
  - A linked list structure is generally inappropriate for hash tables as the direct access ability of arrays is crucial (for speed and efficiency, etc.).
- Example:
  A 10 element (or "bucket") hash table

Index	Value / Content
0	120
1	71
2	
…	
8	258
9	

Hash Function

- A "hash function" calculates the location (index) an item is stored in the hash table based on a variable "key".
- The idea is to apply a hash function that translates or maps a relatively large domain of possible key values to a relatively small range of hash table addresses.
- As a simple example, the hash function (H) for storing integer values on the previous slide using the values themselves as the key (K) is:

H(K) = K mod 10

```
def hash(k):
        return k%10;
```

Continued…

- There are many ways that the hash function can calculate the hashed value (i.e. location).
- For strings, for instance, the character codes (e.g. ASCII codes) could be added together before division remainder maps the total value down to a slot/bucket in the hash table.

```
def hash(k, hashTableSize):
        total=0
        for i in range(len(k))
                total+=k[i]
        return total%hashTableSize
```

Continued…

- It would also be possible to not use all characters in the string, but just sum/add the first X characters.
  - Once the hash function produces the same hashed value given the same key data this is absolutely fine.
- There are a variety of other hashing techniques that can be used (e.g. number folding) but the main thing is that the distribution of data should be as <u>even and uniform as possible</u> across all slots in the hash table.
  - Every slot should (ideally) have an equal chance of being mapped to by the hash function given a key.

Collisions

- As a hash function takes a comparatively large domain of keys and maps these down to a comparatively small range of hash table slots, a hash function is never one-to-one. In other words, different keys may map to the same hash table slot.
  - For instance, where "H(K) = K mod 10", key values of 81, 911, 151, etc. would all map to location 1.
  - Similarly, east, seat and eats would all map to the same slot (using the character code totals). So would many other strings with different letters but the same totals (e.g. datt).
- Two different keys that hash/map to the same slot/address/location/bucket are called synonyms.

Continued…

- When a key hashes/maps to a location that has already been taken or used, this is called a "collision".
- A hash table's load factor (LF), which ranges from 0.0 to 1.0, indicates how full it is.
  - Generally speaking, hashing only works well if the LF is no more than circa 0.7.
    - Otherwise, collisions become too frequent and collision resolution can be costly (negating any gain from hashing).
    - Some hashing approaches require smaller LFs still (e.g. no more than 0.5 for cuckoo hashing) to work well.

Collision Resolution

- Collisions are expected with hashing, but we have to be able to manage or resolve them.
- Various collision resolution strategies exist, including:
  - Open addressing techniques
    - Linear probing
    - Quadratic probing
    - Double hashing
  - Rehashing
  - Separate chaining
  - Cuckoo hashing

Open Addressing

- Open addressing techniques involve storing an item elsewhere in the hash table if a collision occurs and its hashed home location is occupied.
- Linear probing is a standard open addressing technique that "probes forward" (usually) one slot at a time from the occupied home location and takes the next free or available slot.
  - Linear probing will return/wraparound to the top of the hash table and continue probing if it reaches the end.
  - If the home location is revisited during probing the hash table is full (LF=1.0) and the insertion will fail.

Continued…

- Linear probing seeks to store items as close to their home locations as possible (if the home locations themselves are full).
  - This facilitates "locality of reference".
- However, linear probing can result in "primary clustering" as synonyms cluster near their shared home location.
- Other keys that hash to other locations nearby (that may be already taken by linear probing) also add to this primary clustering.
  - Open addressing techniques such as linear probing essentially take slots that other keys may subsequently hash to.

Continued…

- Quadratic probing is similar to linear probing but does not just "probe forward by 1 slot at a time" in an attempt to reduce primary clustering.
- Usually, quadratic probing probes forward (with wraparound) by n^2 with n incrementing. From home location H, it probes H+1, H+4, H+9, H+16, etc. (i.e. $H+1^2$, $H+2^2$, $H+3^2$, $H+4^2$, etc.)
- While quadratic probing helps avoid primary clustering, it does not provide the same degree of locality of reference (i.e. closeness to home location) as linear probing.
  - It can also result in secondary clustering (i.e. clustering at secondary/probed locations), but this does not tend to be problematic.
- It is important for the hash table size to be prime when using quadratic probing to maximise alternative locations.

Double Hashing

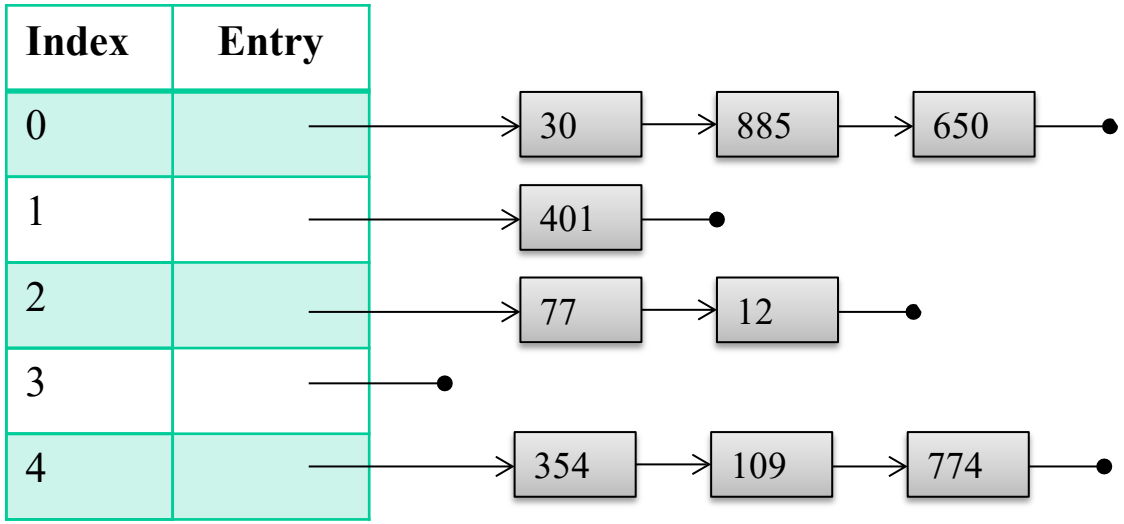
- Double hashing is a collision resolution technique that uses a second hash function in the event of a collision to calculate probe intervals/distances.
- It tries to avoid clustering (primary and secondary) and to spread entries evenly across the hash table.
  - Again, prime-sized hash tables tend to work best to maximise alternative locations on probing.
- Double hashing works as follows:
  - The first hash function H_1 calculates or maps to the home location as usual.
  - If the home location (H) is taken, a second hash function H_2 is applied to the key (K) to calculate the probe interval/distance. From home location H, it therefore probes $H+1H_2(K)$, $H+2H_2(K)$, $H+3H_2(K)$, etc.

Rehashing

- Rehashing is a strategy used when the hash table gets too full (i.e. its LF gets too high), resulting in frequent collisions and impacting performance overall.
- Rehashing essentially means that a new larger hash table is created and that all existing hash table entries are "rehashed" to the new hash table.
  - Using a new hash function appropriate to the new larger table.
- Rehashing is computationally expensive!
- Can choose to rehash when e.g. LF gets too high (exceeds a cut off point) or when an insertion fails.

Separate Chaining

- Separate chaining is an alternative collision resolution technique that allows for elements that hash to the same location (i.e. synonyms) to be stored in a linked list.
- The elements/slots/buckets in the hash table essentially contain/reference the heads of the separate linked lists ("synonym chains").



Continued…

- Under separate chaining, collisions can be easily managed as a new element can simply be inserted as the head of the relevant linked list.
  - Head insertion = fast and convenient.
- In order to find/access an element, the hash function is used to identify the home location and the associated linked list (chain) is searched sequentially.
  - We are back to linear search, but at least we are no longer dealing with one long list but rather N shorter lists where N is the hash table size.
  - All lists would be similarly sized too if the hash function works well.

Other Collision Resolution Strategies

- Various other eclectic collision resolution strategies exist for hashing that try and avoid issues such as clustering while maintaining locality of reference and efficiency/speed.
- One final example provided here is cuckoo hashing.
- Cuckoo hashing (there are variants on the description here)
  - Cuckoo hashing uses 2 or more hash tables with different hash functions.
  - Inserted items always hash to the first table; if the slot there is available, it is simply taken.
  - However, if the slot is already taken the item already there is "evicted" into the next hash table and the newly inserted element takes the now free slot.
  - The evicted element takes its hashed slot in the next hash table, evicting any element there to the next (or back to the first) hash table, and so on until all elements are stored across the hash tables and no more evictions are required.
    - A LF<0.5 is generally required to stop the evictions cycling / looping indefinitely.

The hash() Method in Python

- Hash is a very important concept in Computer Science, that's why many programming languages included a hash function as one of their built-in methods;
- Python included the hash() method and it can be called on any object. It will return an integer value for the object and this result is called hash code or hash value.
- It's important to notice that most objects are hashable, however not all the objects are. Mutable objects such as lists or tuples may not be hashable because when the object is changed or modified, its hash value may also change.

Hashing in Python

- Python also allows the programmer to have control over the hash codes by letting the programmer implement a __hash__ method on a class;
- By defining this __hash__ method in a class you can return hash values integers for the instances of the class you created.

Summary

- In this section we have looked at “hashing” as an approach to efficient data storage and access.
  - Trying to avoid linear/sequential access = key.
  - Hashing can be used in non-list-based data structures too.
- We have looked at key principles underpinning hashing including:
  - Hash tables.
  - Hash functions.
  - Collisions and various collision resolution strategies.