

# 31251 – Data Structures and Algorithms

Week 8

Luke Mathieson / Marco Tomamichel

## This time, in 31251 DSA:

- Maps
- Hashing
- Collisions

## Maps

# Multidimensional and non-numerical Data

- Contrary to first year programming examples, not all data are ints.<sup>1</sup>
- We often have multiple pieces of data we want to bundle together.
- Even when the data is unitary, it may not be numerical, let alone integer.
- Especially true in object oriented programming.

---

<sup>1</sup>Well, they are but...

- We really want to use arrays (or something very like them) though –
  - A lot of the fast algorithms are fast because they use arrays.
- However we can't address memory with a database record (mostly).
- What we want is an array that can be indexed by keys from a set of arbitrary type.

- Map is also called: associative array, symbol table, dictionary.
- It is an abstract data type that stores  $\langle key, value \rangle$  pairs.
- The *key* is unique (in theory at least).
- The *value* can of course be more complex than a single value.
- The *key* is the index for the entry *value*—so we can treat a map like an array.

- `add(key, value)` – insert a new element `value` into the map accessed by `key`.
- `get(key)` – retrieve the element indexed by `key`.
- `remove(key, value)` – delete the given pair from the map.

- There are a number of complications compared to a simple array:
  - What if we try to add two values with the same key?  
Overwrite, ignore the second? Sometimes reassignment of keys is implemented in a separate method.
  - Are `null` keys or values allowed?
  - What happens when we get a non-existent key?



Some other sensible ideas:

- `containsKey(key)` – check if `key` exists as a key in the map.
- `containsValue(value)` – check if `value` is an entry in the map.
- `getKey(value)` – retrieve the key that references `value`.
- Plus the usual `isEmpty()`, `size()`, constructors etc.

Depending on the exact circumstances, a number of strategies may be useful:

- If the key set is already small valued, positive integers, we could just use an array as the implementation.
- If there a small number of total entries, a linked list would be sufficient –  $\mathcal{O}(n)$  to do anything, but there's not much there.
- If the keys are totally ordered, we can extend binary search trees (or similar, more sophisticated data structures).

We mostly want a more general approach – one method is to use *hashing*.

# Hashing

- At their most general, *hash functions* (or *hashes*) are functions that take input of arbitrary length, and produce an output of fixed length.
- If the output length is  $k$  bits, we can interpret the output as an integer in  $[0, 2^k - 1]$ .
- So we can use this as a way of turning our key set into normal array indices.
  - Thus a map can be implemented as an array plus a hash function.

## Example hash functions - Division

- Array of size  $N$ .
- $h(K) = K \bmod N$ .
- If  $N = 20$  and  $K = 36$ , then  $h(K) = 16$
- This would send the item with key 36 to array cell 16.
- Easy to modify for non-numeric keys, just interpret the key as a binary number.
- Works best with arrays of prime length.

## Example hash functions - Folding

- Break the key up into parts, then combine arithmetically.
- e.g. Given a key 123456789 and an array of size 709, we could break it into 123, 456 and 789, add the three parts and take the modulus.
- $123 + 456 + 789 = 1368$  and  $1368 \bmod 709 = 659$ , so we send key 123456789 to array cell 659.

## Example hash functions - Mid squared

- Take the key, square it, and take the middle digits (how many is determined by the array size).
- e.g. Array size 1000 and  $K = 3121$ .
- Square the key and take the three middle digits  
 $K^2 = 9740641$ .
- Key 3121 goes to cell 406.

## Example hash functions - Extraction

- Only use part of the key.
- e.g. Array size 1000, and  $K = 542732346$ .
- We might take only the 4<sup>th</sup>, 6<sup>th</sup> and 7<sup>th</sup> digits.
- Then  $h(542732346) = 723$ .



## Example hash functions - Radix

- Convert the key to a different base, then take the mod.
- e.g. Array size 97,  $K = 345$ .
- Convert 345 to base 9 –  $345_{10} = 423_9$ .
- Then  $423 \bmod 97 = 35$ , i.e.  $h(345) = 35$ .

- What did these all have in common?
- Not really all that much – taking the  $\text{mod}$  is pretty standard.
- Hash functions are often application dependent – if your data has special structure, you can exploit this to produce a better/faster hash function.
- But there are some desirable properties in general.

## Some properties of good hash functions

To make the hash function useful it should:

- Be fast to calculate.
- Be deterministic (we should always get the same hash from the same key).
- Be able to handle mapping to different sized ranges.
- Spread the inputs evenly across the outputs.

- Clearly, even with unique keys, some keys must hash to the same value.
- This is called a *collision*.
- The last two properties on the previous slide address collision avoidance.
- If a hash function has no collisions it is called a *perfect* hash function.
- In general, collisions are unavoidable, so we need strategies for dealing with them.

## Handling collisions

- Confusingly also called *Closed Hashing*.
- When a collision occurs, we search for an alternative open spot in the array.
- This is called *probing*.
- The way the probing is performed changes the performance of the hash.

- The simplest – just search linearly along the array until you find somewhere free.
- So at the  $i^{th}$  attempt, we try cell  $h(K) + i - 1$  (the  $-1$  is just so we start at  $h(K)$ ).
- Example – let  $h(K) = K \bmod 11$ , with an array of size 11, insert 13, 26, 5, 37, 21, 16, 15 & 31.

- At each step, instead of just moving to the next cell, we increase the gap.
- The  $i^{th}$  attempt is made at  $h(K) + (-1)^{i-1} \cdot (\frac{i+1}{2})^2$ .
- The sequence of attempts is then  $h(K), h(K) + 1, h(K) - 1, h(K) + 4, h(K) - 4, \dots, h(K) + \frac{i^2}{4}, h(K) - \frac{i^2}{4}, \dots$
- Compare this method with the linear probing example.
- More complicated quadratic polynomials can be used.



- Both linear and quadratic probing suffer from clustering problems - reducing them to linear search.
- Both sensitive to table load, performance gets worse as the array fills up.
- Quadratic probing is sensitive to load and table size – if it's more than half full and not of prime size, it's possible that no open position can be found.

Other probing strategies exist — i.e. *double hashing* uses two hash functions, with the  $i^{th}$  probe being  $h_1(K) + i \cdot h_2(K)$ .

- But one problem remains for all probing strategies: we can't just delete elements anymore (why?).

- Chaining is also called *open hashing*.
- Instead of storing the elements directly, the array stores secondary data structures.
- *Separate Chaining* – each array entry is a linked list of elements with that hash.
- *Scatter Chaining* – each array entry is a table of pointers/references to elements (not as applicable in Java).
- *Coalesced Chaining* – combines chaining and probing:
  - Store colliding entries in the last available position in the array.
  - Can set aside a special section of the array to be the *cellar* where all the chained elements are.

- Need to additional space to store references/lists/etc.
- Can't access the data directly from the array – slows things down.
- Probing based strategies make it easy to tell when we should stop and resize – trickier to tell with chaining methods.

- Sort of a hybrid approach.
- Allocate a larger amount of space to each array cell – enough to store more than one element.
- Each entry is a *bucket* of values.
- Essentially an array of arrays – we keep as much of the benefit of arrays as possible, but can still chain a limited number of elements.
- Can add an overflow as the final entry.

## So what does hashing offer

- If we have a good hash function, and relative few collisions:
  - $O(1)$  insertion.
  - $O(1)$  retrieval (key known).
  - $O(1)$  deletion.
  - $O(n)$  search (key unknown).
  - $O(n)$  space.
- If things go badly, these all reduce to  $O(n)$ .
- Remember, this is all with arbitrary key data – this is why hashmaps/hashtables/etc. form the core of many data-access-heavy applications.