31251 – Data Structures and Algorithms

Week 8, Autumn 2020

Xianzhi Wang

This time, in 31251 DSA:

- Maps
- Hashing
- Collisions

Maps

- Data Organisation by Position
  1. Array
  2. Lined List
  3. Sequential containers: `vector`, `list`, `deque`.
  4. Their combinations

  - Lookup by position: $O(1)$ or $O(n)$
  - lookup by value $O(\log n)$ or $O(n)$

- Data Organisation by Key
  1. Associative containers: `(unordered) set`, `(unordered) map`

  - Lookup by position: n/a or using `iterator`
  - lookup by value $O(\log n)$

But...

- We really want to use arrays for
  - Highly efficient algorithms
- What we want is an array that can be indexed by keys from a set of arbitrary type.

Remember how you retrieve a database record:

```
select ...
from ...
where id='123';
```

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

Basic Map Operations

- `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 `array`:

- What if we try to add two values with the same key?
    - Overwrite, ignore the second?
    - Reassignment of keys in a separate method.
- Are `null` keys or values allowed?
- What happens when we `get` a non-existent key?

Additional Map Operations

- `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`.
- `isEmpty()`, `size()`, constructors, etc.

Depending on the exact circumstances, a number of strategies:

- If the keys are 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

Hash Functions

- 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 $[0, 2^k)$.
- So we can use this as a way of turning our key set into normal array indices.
  - Thus an associative array can be implemented as `an array plus a hash function`.

- Given Array of size N, the hash function $h(K) := K \mod N$.

  If $N = 20$ and $K = 36$, then $h(K) = 36\%20 = 16$.

  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 lengths.

Hash Function (2) — Folding

- Break the key up into parts, then combine arithmetically.

  Given $N = 709$ *and* $K = 123456789$*, we break* K *into* $\{123, 456, 789\}$*, and then take the modulus of their sum:* $(123 + 456 + 789)\%709 = 659$*.*

  As such, we send key 123456789 *to array cell* 659*.*

Hash Function (3) — Mid Squared

- Take the key, square it, and take the middle digits—how many is determined by the array size.

Given $N = 1000$ *and* $K = 3121$*, square the key and take the three middle digits:* $K^2 = 97\mathbf{406}41$.

This will send key 3121 *to array cell* 406.

Hash Function (4) — Extraction

- Only use part of the key.

Given $N = 1000$, and $K = 542732346$, we might take only the 4^{th}, 6^{th} and 7^{th} digits.

This will send key 542732346 to array cell 723.

- Convert the key to a different base, then take the mod.

  Given $N = 97$, and $K = 345$, convert 345 to base 9 ($345_{10} = 423_9$) and take the mod: $423 \mod 97 = 35$.

  This will send key 345 to array cell 35.

- What did these all have in common?
  - Not really all that much — taking the `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

- Deterministic — always give the same hash for the same key.
- Efficient — be fast to calculate.
- Scalable* — can handle mapping to different sized ranges.
- Collision-avoiding* — can spread inputs evenly across outputs.

- `collision` — different keys being hashed to the same value.
- The last two properties on the previous slide address *collision avoidance*.
- Suppose a hash function has no collisions, it is called a "perfect hash function".
- In general, collisions are *inevitable*, so we need strategies for dealing with them.

Handling Collisions

- *Probing* (or *Closed Hashing* — when a collision occurs, find an alternative open spot in the array instead.
- It will decrease the hashing performance.

- The simplest – search sequentially along the array until finding somewhere free.
- So at the i^{th} attempt, we try cell $h(K) + i - 1$
  - -1 is just so we start at $h(K)$).

Considering an array of size 11 *and* $h(K) = K \mod 11$, *insert* $13, 26, 5, 37, 21, 16, 15$ *&* 31.

- At each step, instead of trying the next cell, we increase the gap — The i^{th} attempt is made at $h(K) + (-1)^{i-1} \cdot (\frac{i+1}{2})^2$.

Table: Sequence of attempts

1	$h(K)$
2	$h(K) + 1$
3	$h(K) - 1$
4	$h(K) + 4$
5	$h(K) - 4$
$\dots$	$\dots$

- Compare this method with the linear probing example.

Note: more complicated quadratic polynomials are available

Issues with Probing

- Can't just delete elements anymore (why?)
  - The hashed key of one key might have stored other elements.
  - Worse case — linear and quadratic risk reducing to linear search.

- Both linear and quadratic probing are 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, e.g., *double hashing* uses two hash functions, with the i^{th} probe being $h_1(K) + i \cdot h_2(K)$.

- `Chaining` (or `Open Hashing`): Instead of storing 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 linear 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.

- Extra space — need additional space to store references/lists/etc.
- Indirect access — Can't access the data directly from the array, which slows things down.
- Performance evaluation — probing-based strategies make it easy to tell when we should stop and resize — trickier to tell with chaining methods.

- *Bucket of values* — allocate a larger space to each array cell, big enough to store more than one element.
- 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.
- Basically impossible in Java (only works when you can address memory directly).

So What Does Hashing Offer

- If we have a good hash function, and relative few collisions:
  - $O(1)$ insertion.
  - $O(1)$ retrieval.
  - $O(1)$ deletion.
  - $O(n)$ search.
  - $O(n)$ space.
- If things go badly, these all reduce to $O(n)$.
- Hashmaps/hashtables form the core of many data-intensive applications.