Cosc 201
Algorithms and Data Structures
Lecture 13 (13/4/2026)
Hashing

Brendan McCane
brendan.mccane@otago.ac.nz
And
Michael Albert

Maps and sets revisited

- In computer science, a *map* consists of a set of *keys*, each of which has an associated *value*.
- So sets are a prerequisite, but also any map can be used to store a set simply by ignoring the values for each key (or, e.g., setting it to `null`)
- The fundamental map operations are usually considered to be:
  - *put*(k, v): add the mapping from k to v either by adding k if it's not already present, or by changing the associated value,
  - *get*(k): return the value associated with k if k is present, and
  - *remove*(k): remove the key k (and any associated value).
- We could implement a `Set` or `Map` implementation using BSTs where the cost of each operation is $O(\log n)$.
- But, this requires an underlying order on keys (and presupposes the order is relevant for e.g., inorder traversal).
- What if we don't care about an order. Can we get down to (or close to) $O(1)$? Worst-case? Amortised case?

A perfect world . . .

Let's dream. Keys come from a class `K`, values from a class `V`

- There are only 4000 possible keys.
- Each key, `k`, has a unique four-digit identifier that we can obtain in constant time as `k.id()`;

```
V[] map = new V[10000];

public V put(K key, V value) {
  V old = map[k.id()];
  map[k.id()] = value;
  return old; // To match the Java Map interface
}

public V get(K key) {
  return map[k.id()];
}

public V remove(K key) {
  V old = map[k.id()];
  map[k.id()] = null;
  return old;
}
```

Or is it?

That works, but . . .

- We always use storage for 10000 elements regardless of how many we actually have in the map.
- We require the magic `k.id()`.
- To be fair, this is a problem with most `Map` implementations.

So, what next?

- The main block is the `k.id()` issue.
- Avoiding wasting *too* much space would be nice too.
- If we want constant time access we're pretty much stuck with arrays so we have the usual static vs. dynamic tradeoff (and may need to resize occasionally).

Dealing with the space issue

Returning to our dream world, imagine that we knew somehow that we would never need maps for more than 30 values at a time. Why not try something like this?

```
V[] map = new V[53];

public V put(K key, V value) {
  V old = map[k.id() % map.length];
  map[k.id() % map.length] = value;
  return old; // To match the Java Map interface
}

public V get(K key) {
  return map[k.id() % map.length];
}

public V remove(K key) {
  V old = map[k.id() % map.length];
  map[k.id() % map.length] = null;
  return old;
}
```

What's the problem?

- The `id` of two different keys could produce the same value when we take the remainder modulo our array size.
- So, in `get` we might get the value for a different key and in `put` we might overwrite the value for a different key.
- We need to be able to deal with these kinds of *collisions*.
- And, what about the whole "magic ID" problem anyhow?

Solving the magic ID problem

- The solution to the magic ID problem is to make one up.
- A made up ID is called a *hash code*.
- More generally a *hash function* takes objects from a class as input and produces a value from a fixed finite set of values (in Java, a `int`).
- What properties should a hash function have?
  - It should be *very* fast to compute.
  - It should be *uniform* – no value should be more likely to occur than any other.
  - In fact, this should be true even when we take remainders modulo (any?) fixed number.

There is still a problem

- A hash code must be dependent on the contents of the object.
- In Java, if two objects are equal (according to the `equals` method) then their hash codes must be equal.
- For Objects, Java provides a default hash function that uses the memory address of the object.
- If you create a class in Java, but don't provide a `hashCode` method, the default one will be used.
- If you provide an `equals` method, but not a `hashCode` method, then the underlying assumption regarding `equals` and `hashCode` is violated, and hash maps in Java will not work as expected.
- So, if you implement an `equals` method in a class, then you should also implement a `hashCode` method.

Example hashCode functions

Java uses the following hashCode implementations:
For a `String`:

```
hashcode = s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]
```

For a `List`:

```
int hashCode = 1;
for (E e : list)
    hashCode = 31*hashCode + (e==null ? 0 : e.hashCode());
```

These hashCodes are almost guaranteed to be unique - different strings will have different hash codes.
But the codes will potentially be very large, and we don't want to create an array that big.

Universal hash functions

- If you have a website that uses hashing for it's main functionality, then if an adversary knows the hash function, they can create a lot of collisions and make your website slow and/or unusable.
- Rather than a fixed hash function, we can use a random hash function.
- A universal set of hash functions, H, is a set of hash functions such that if you pick keys k and j at random, and choose a hash function randomly from H, then the chance of $h(k) = h(j)$ is no more than $1/m$ (where m is the size of the hash table).

A Universal hash function

- Choose a prime number p big enough that every possible key $k < p$, and choose table size $m < p$.
- Let $h_{a,b}(k) = ((ak + b) \mod p) \mod m$.
- The parameters a and b are chosen randomly on program startup.
- Where $1 \leq a \leq p - 1$ and $0 \leq b \leq p - 1$.

Example universal hash function

- So, with $p = 17$ and $m = 6$,

$$\begin{aligned} h_{3,4}(8) &= ((3 \cdot 8 + 4)\%17)\%6 \\ &= (28\%17)\%6 \\ &= 11\%6 \\ &= 5. \end{aligned}$$

- Try $h_{4,5}$ with $k = 8$ and $k = 1$.
- and $h_{4,5}(8)$ but with $p = 19$ and $m = 7$,