Cosc 201
Algorithms and Data Structures
Lecture 18 (30/4/2025)
Hashing and Collisions

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

Recap

- We will implement a map using an array.
- The array elements are called *buckets*.
- To add an item to the map, compute its hashcode and take the modulus with the size of the array.
- We usually use a universal hashing function to compute the array index:

$$h_{a,b}(k) = ((ak + b) \mod p) \mod m$$

- What if two keys have the same hashcode?
- Two possibilities: probing or chaining.

Linear Probing

- If we thought of the array as a sequence of rooms, then an obvious approach, if the room we want is full, is to use the next room.
- That's what we will call *linear probing*.
- It guarantees that, so long as there's still at least one space free, we won't fail to insert a new value (of course we circle back to the beginning when we reach the end).
- How frequently do we expect collisions to occur?
- How many spaces will we need to look through in order to find a free space?
- Both answers depend on the *load factor* of the map – the percentage of occupied slots.
- This load factor is frequently capped as part of the implementation – adding more elements then triggers a resizing.

Insertion

- The cost of an insertion is (proportional to) the length of the block of filled cells we need to search before finding a vacant spot.
- So long as the load is not too high (in particular if we have $< 75\%$ load) this is (at least on average) $O(1)$.
- This is not worst-case, since all the elements we inserted might have had the same hashcode (modulo the table size) and we'd have a single long block followed by vacant space.
- To search, we look where the element *ought* to be – if that cell is empty, then it's nowhere. If it's full but the wrong element, we follow the block from there, checking as we go until we either succeed, or hit an empty cell (and fail).

Resizing

- What do we do when we need to resize (because we've exceeded our load bound?)
- **Easy**. Just initialise a new table of a different size (typically about double) and insert all the elements of the current table into it.
- The cost is $\Theta(T)$ where T is the size of the original table, but since it was largely full, that's also $\Theta(n)$ where n is the number of elements in the table.
- This can be amortised across those elements to give $O(1)$ cost in the amortised sense.

A reminder about amortised cost

- The concept of *amortised cost* is an important one in dynamic data structures, particularly when using static support in the background.
- Effectively, it allows individual operations to borrow time from previous ones.
- Alternatively, we can think of previous observations as adding time to a bank that later ones can draw on.
- For example, the operations of a dynamic data structure have amortised time complexity of $O(1)$, if
  - there is a constant $C > 0$ such that,
  - for every positive integer k, and
  - any sequence of k operations on the data structure (from initialisation),
  - the average time per operation is less than C.

Removal

- It's not enough to search for the element we want to remove and simply free its spot.
- The reason is that elements might have been pushed further by it, from earlier positions where they belong, and now a search for them would fail (since it would hit an empty spot).
- Two solutions:
  - Replace it by a special tombstone marker. This counts as "full" for search purposes and load factor, but empty for insertion purposes (and inserting over a tombstone doesn't change the load factor).
  - Search forward from the element we're removing until we find something that belongs in that location or earlier – swap it back into this location and repeat until an empty cell is found.

Chaining

- Each bucket is a list of key-value pairs.
- When a mapping is added, check the bucket corresponding to its hashcode.
- If the bucket is empty just add it (or create it, and add it).
- If the bucket is not empty, check to see whether the given item belongs to its list.
  - If it does, change the associated value.
  - If it doesn't, add it.
- Getting and removing values can be handled similarly.

Chaining is used in Java for `Hashtable` and hence, inferentially at least, also for `HashMap` and `HashSet`.

Complexity

- For both chaining and linear probing:
  - The average cost of insertion, deletion and search is $O(1)$.
  - The worst case cost of insertion, deletion and search is $O(n)$.
  - The space complexity is $O(n)$.
- Compare to a balanced BST which has $O(\log n)$ for all operations.
- Can we do better?

Perfect hashing

- If we know the set of keys in advance, we can use a perfect hash function to map them to an array.
- This is a function that maps n keys to n slots with no collisions.
- The cost of insertion, deletion and search is $O(1)$.
- The space complexity is $O(n)$.

This is not practical for most applications, but it is useful for some applications where the set of keys is known in advance and does not change - for example any read-only memory (were often used in CD-ROMs).

Better than O(n)

- For probing, we can't do better than $O(n)$ in the worst case.
- For chaining, we can't do better than $O(n)$ in the worst case if the chain is a list.
- But, we could use a tree instead of a list for the chain - then our worst case is $O(\log n)$, but we still keep the average case at $O(1)$.
- We could also use a hash table of hash tables. If we do this right, then we can get the worst case down to $O(1)$.