Hash tables

Suppose we want to store a set of numbers. How do we implement this?

possibilities: an array or a linked list

has size $n$

we'll see what hashing does — aims for $O(1)$ time for membership / insert / delete

How much space do we need?

linked list: $O(n)$
array: $O(n)$ — careful with the allocated space (reallocate if we are to exceed the current space)

How much time to check membership? Insert? Delete?

$O(n)$ $O(1)$ $O(n)$ — if have to find an element or possibly $O(1)$ if not have to find
Hash tables: bucket arrays

Bucket arrays

\[ S = \{ 4, 9, 17, 42, 53, 67 \} \quad n = 6 \]

Hash functions

\[ h(x) = x \mod N \]

- a hash function

Needs space: \( O(N) \)

We will try to have: \( N = O(n) \)

Is 24 in the set?

If no collisions, then operations are all \( O(1) \)

Goal: minimize collisions

Collisions

- If several elements compete for the same bucket

- We will discuss collision solving techniques
Hash tables: hash functions

In Java: hash function implemented in 2 parts:
- hash code - takes an object and returns an integer
- compression function: map hash code to range [0, ..., N-1]

Built-in `hashCode()` - inherited from `Object`
- returns a memory address of the object
- sometime useful to override: e.g. Java String
  - we want two objects with the same content to have the same `hashCode()` so that they are hashed to the same bucket and we can easily do membership testing etc.
Hash tables: hash codes

Typical possibilities:
- casting to an integer
    
    
    good strategy for Integer, Byte, Char, Float  \( h_c(x) = \text{(Integer)} \cdot x \)

- summing components
    
    what happens for double?

- polynomial hash codes
    
    choose \( a \) (any number - integer > 1)
    
    \[
    h_c_1(x_1, x_2, \ldots, x_k) = a^{k-1}x_k + \ldots + a^2x_3 + ax_2 + x_1 \]

- (cyclic shift)

- summing up components, e.g.,
  
  \( h_c(x) = \left\lfloor x \right\rfloor + \text{fractional part of } x \text{ viewed as an integer} \mod \text{MAXINT} \)

  suppose our elements are strings
  \[
  \begin{align*}
  h_c(s_1, s_2, \ldots, s_k) &= \text{sum of the hash codes of the characters } s_i \\
  h_c(\text{STOP}) &= s_1 s_2 \ldots s_k \\
  h_c(\text{POST}) \quad &\text{all the same hash code} \\
  h_c(\text{SPOT}) \ldots \quad &\text{want to avoid this}
  \end{align*}
  \]
Hash tables: compression functions

Typical possibilities:

- the division method

\[ h(x) = h_c(x) \mod N \]

- the MAD (multiply add and divide) method

Choose two numbers \( a, b \) and do:

\[ h(x) = (a \cdot h_c(x) + b) \mod N \]
Hash tables: collision handling

Hash-supported methods: put(), get(), remove()

Collision handling via separate chaining:

Load factor: \( \frac{n}{N} \) - want bounded by a constant < 1
Hash tables: collision handling

Collision handling via open addressing:

- linear probing

if $h(x)$ is taken,
go to $h(x) + 1 \mod N$, then $h(x) + 2 \mod N$, etc. until we find an empty space

Careful with deleting elements - e.g. if 7 is deleted
- want to replace it with 57
so that search for 10, 57 works fine

7 comes in, wants position 3
(the hash value of 7 is 3)

10 comes in, wants pos. 3

57 comes in, wants pos. 3
Collision handling via open addressing:

- quadratic probing

First try $h(x)$, then $(h(x) + 1) \mod N$

$17$

$5$

$13$

$7$

$57$

$10$

$9$

$7$ comes in and wants position $3$

$10$ comes in

$17$

$3$

$+9$

$+4$
Collision handling via open addressing:
- quadratic probing
- double hashing
Hash tables

Which one to use? (Which hash function? Which compression? Which collision handling?)
The load factor $\lambda$ is used as an indicator for the need to rehash.
Hash tables: application

Application: counting word frequencies in a text.