Double Hashing

Into & Live Coding

Hashing

- Arbitrary Size → Fix Size

Insert(0) 0 % 5 = 0

0				

Hashing

- Arbitrary Size → Fix Size

Insert(0)	0 % 5 = 0
Insert(5321)	5321 % 5 = 1

0	**5321**			

Hashing

- Arbitrary Size → Fix Size

Insert(0)	0 % 5 = 0
Insert(5321)	5321 % 5 = 1
Insert(-8002)	-8002 % 5 = 3

0	**5321**		**-8002**	

Hashing

- Arbitrary Size → Fix Size

Insert(0)	0 % 5 = 0
Insert(5321)	5321 % 5 = 1
Insert(-8002)	-8002 % 5 = 3
Insert(20000)	20000 % 5 = 0

0	**5321**		**-8002**	

Handling collisions

- Can we reduce the number of collisions?
  - Using a good hash function is a start
    - What makes a good hash function?
      1. Utilize the entire key
      2. Exploit differences between keys
      3. Uniform distribution of hash values should be produced

Modular hashing

- Overall a good simple, general approach to implement a hash map
- Basic formula:
    - h(x) = c(x) mod m
        - Where c(x) converts x into a (possibly) large integer
- Generally want m to be a prime number
    - Consider m = 100
    - Only the least significant digits matter
        - h(1) = h(401) = h(4372901)

Open Addressing

- I.e., if a pigeon's hole is taken, it has to find another
- If h(x) == h(y) == i
    - And x is stored at index i in an example hash table
    - If we want to insert y, we must try alternative indices
        - This means y will not be stored at HT[h(y)]
            - We must select alternatives in a consistent and predictable way so that they can be located later

Linear probing

- Insert:
    - If we cannot store a key at index i due to collision
        - Attempt to insert the key at index i+1
        - Then i+2 …
        - And so on …
        - mod m
        - Until an open space is found
- Search:
    - If another key is stored at index i
        - Check i+1, i+2, i+3 … until
            - Key is found
            - Empty location is found
            - We circle through the buffer back to i

Double hashing

- After a collision, instead of attempting to place the key x in i+1 mod m, look at i+h2(x) mod m
  - h2() is a second, different hash function
    - Should still follow the same general rules as h() to be considered good, but needs to be different from h()
      - h(x) == h(y) AND h2(x) == h2(y) should be very unlikely
        - Hence, it should be unlikely for two keys to use the same increment

What we will do

- Decide two hash functions.
- Try the first one, if we collide, increment using the second hash function.
- Must support insert and find.
- Bonus: resizable

Example

- https://www.cs.usfca.edu/~galles/visualization/ClosedHash.html