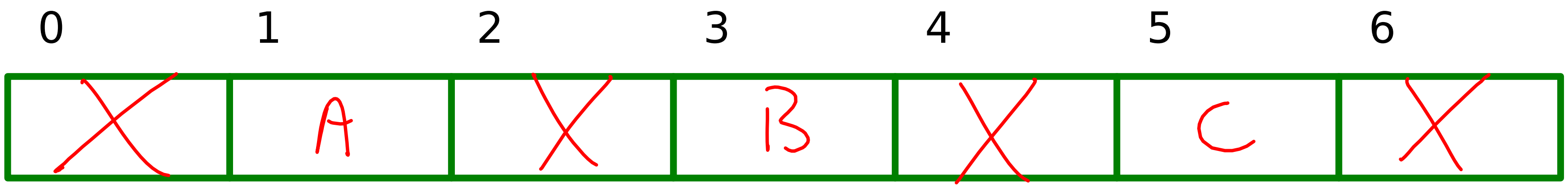
# Hash Tables

# Table naïve array implementation

- "Direct addressing"
- Worst case O(1) access cost
- But likely to waste space

| K | V |
|---|---|
| 1 | A |
| 3 | B |
| 5 | C |

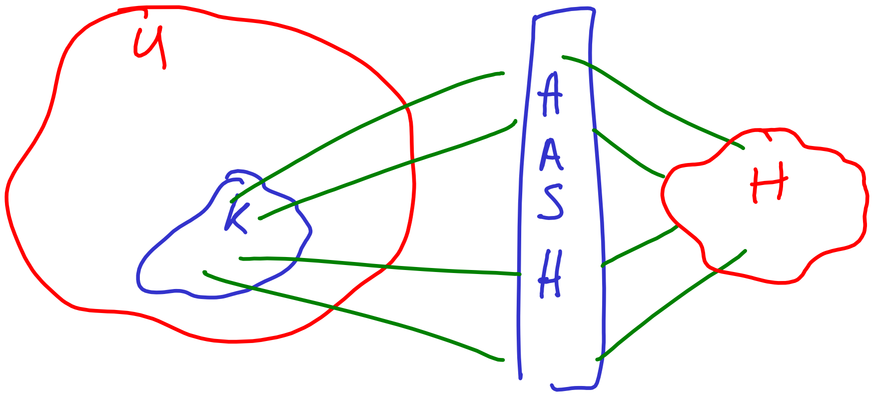


| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| X | A | X | B | X | C | X |

# Hashing

- U: Set of all possible keys
- H: Set of all possible hashes
- K: Set of actually used keys

- E.g. Division hash: h(x)=x mod y



size(U) > size(H)

| Key | Value |
|---|---|
| 2 | C |
| 3 | D |
| 8 | B |
| 9 | A |
| 4 | E |

y=4

| 0 | 1 | 2 | 3 |
|---|---|---|---|
| B<br>E | A | C | D |

O(1) lookup

Division is slow

# Hashing

- E.g. Multiplication hash: $h(x)=\lfloor k(ax \bmod 1)\rfloor$

$$h(x)=\lfloor \underset{\text{Constant}}{k}\underbrace{(\underset{\text{constant } 0<a<1}{a}x \bmod 1)}_{\text{Fractional part}} \rfloor$$

- Multiply key by a ⇒ F.P. number
- Take fractional part
- multiply by k

$k = 10{,}000$

$a = \frac{\sqrt{5}-1}{2}$

$h(123456) = \lfloor 41.155 \rfloor$

$= 41$

$h(123457) = 6221$

$h(123458) = 2401$

# Uniform Hashing

- Any analysis is going to be dependent on the hash function properties
- We usually consider uniform hashing:

A given key x has a hash h(x) which is equally probable to be any member of the set H

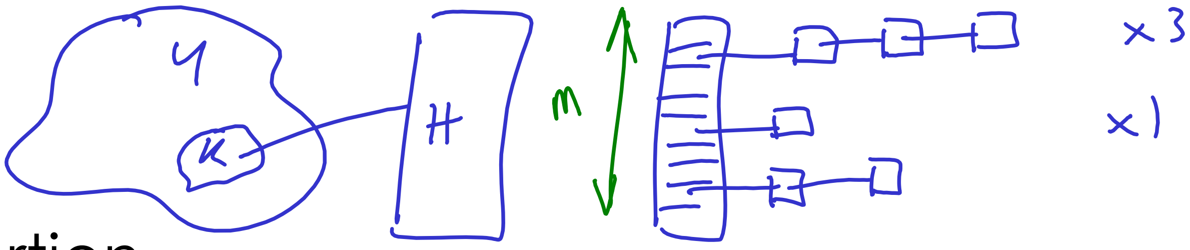
# Avoiding Collisions

- Since size(H)<size(U) multiple keys must map to the same hash value: Collisions
- Choose a good hash function
  - The more random the output, the less likely we will have collisions
  - Still going to happen though!

“good” depend. on input data.

# Chaining

- Each hash table slot is actually a linked list of keys



- Insertion
- Deletion
- Search

$O(1)$ add to end/start list.

splice out $O(\text{size of list})$

$\rightarrow O(\frac{n}{m})$

**Worst-case**

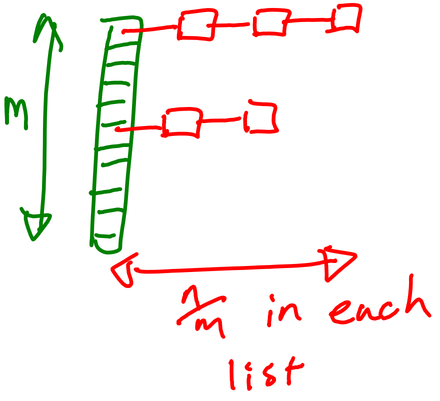
all hash to same value $\Rightarrow O(n)$

**Average**

Assume uniform hashing.

# Chaining

- Search



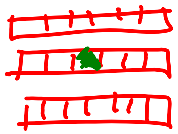
**Fails**

Time = time to traverse list + time to hash

$$\underbrace{\text{time to traverse list}}_{O\left(\frac{n}{m}\right)} + \underbrace{\text{time to hash}}_{O(1)}$$

Time $= O(1+\alpha)$ ↳ $\frac{n}{m}$

**Succeeds**



Imagine searching for the $i^{th}$ added key

$$\boxed{\frac{i-1}{m}+1}$$

$$\text{Average} = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{i-1}{m}+1\right)$$

$$= O(1+\alpha)$$

# Open Addressing

- Generate a global sequence of hash values and assign them in order
    - No link between the key and the hash

$\{1,6,3,4\}$

ABCD

$1 \rightarrow A$
$6 \rightarrow B$
$3 \rightarrow C$

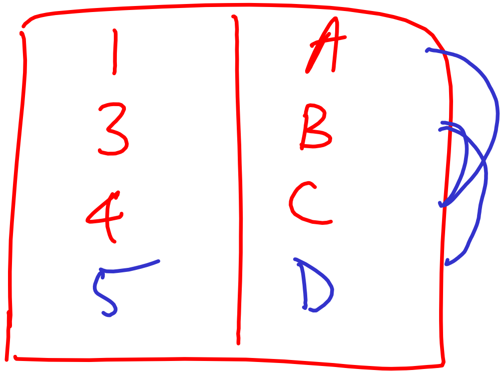
Search
$O(n)$

- Now associate an individual sequence of hashes with each key (there are m! sequences to choose amongst)

key generates sequence

E.g. A = 1, 3, 5, 2, 4
B = 3, 5, 2, 1, 4
C = 4, 2, 3, 1, 5
D = 1, 4, 3, 5, 2

ABCD



# Search

- Use key to generate probe sequence
- Follow sequence until:
  - Element found OR
  - Empty hash slot reached OR
  - Sequence finishes

| K | V |
|---|---|
| 1 | A |
| 2 | — |
| 3 | B |
| 4 | C |
| 5 | D |

Search D

probe: {1, 4, 3, 5, 2}

1 → A ×

4 → C ×

3 → B ×

5 → D ✓

Search E

Probe: {1, 3, 2, 5, 4}

1 → A ×

3 → B ×

2 → Empty

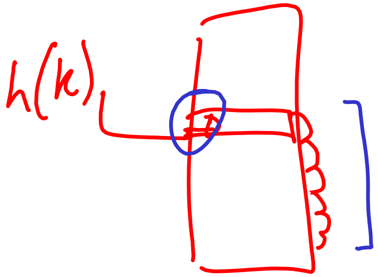
E is not in list

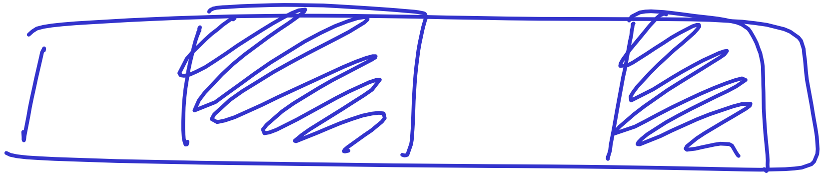
# Generating Sequences

Requirement: Every probe sequence should be a permutation of {0,1,...,m-1}

- Linear Probing

$$S_i(k) = \left(h(k) + i\right) \bmod m$$





- Basically 'randomises' the start of the sequence and then proceeds incrementally

X Long runs of slots "primary clustering"

X Two keys with same h(k) "secondary clustering"
same sequence

# Generating Sequences

Requirement: Every probe sequence should be a permutation of {0,1,...,m-1}

- Linear Probing

- Basically 'randomises' the start of the sequence and then proceeds incrementally

# Generating Sequences

- Quadratic Probing

$$S_i(k) = \left(h(k) + c_1 i + c_2 i^2\right) \bmod m$$

✓ No primary clustering

✗ secondary clustering

- Double Hashing

$$S_i(k) = \left(h(k) + i\,h_2(k)\right) \bmod m$$

✓ No primary

✓ No secondary

✗ Hard to find $h$ and $h_2$

# Number of Probes Needed

- Consider a search that performs i probes before it fails

# Open Addressing Performance

- Ave. number of probes in a failed search

$$\leq \underline{\frac{1}{1-\alpha}} \qquad \alpha = \frac{n}{m}$$

- Ave. Number of probes in a successful search

$$\underline{\frac{1}{\alpha} \ln \frac{1}{1-\alpha} + \frac{1}{\alpha}}$$

- If we can keep n/m ~constant, then the searches run in O(1) still

# Issues with Hash Tables

- Worst-case performance is dreadful — $O(n)$
- Deletion is slightly tricky if using open addressing

A : {1,2,3,4}
B : {1,3,4,2}
C : {1,4,2,3}
D : {1,3,2,4}

| k | v |
|---|---|
| 1 | A |
| 2 | D |
| 3 | ~~B~~ "DELETED" |
| 4 | C |

Search D

1 → A X

3 → empty → D is not in array

i) Forbid deletions

ii) Mark with "DELETED"

⇒ Search ignores node

⇒ Insert replace node

# Finishing with Tables

- You may remember we started all this trying to figure out how to implement the Table ADT
  - RB and AVL trees give us balanced trees that always give good performance and allow us to order the keys
  - B-trees are useful when the table becomes so big we needed to store it on the hard drive
  - Hash tables provide us with potentially really fast operations, but (like quicksort) terrible worst-case performance. They also don't sort the keys so you can't iterate over them in order

# Java Tables

- HashMap
  - Implements a hash table
  - You can specify an initial size and a load factor (n/m)

- TreeMap
  - Implements a red-black tree
  - Allows you to provide a Comparator for the keys so they are stored in sorted order
  - Therefore you can iterate over the values in a stable key order