Hashing

Try to implement the following **Abstract Data Type** using a Data Structure which makes each operation cost O(log n) in the average case.
Challenge: O(1) in the average case.

<u>Dictionary</u> {"Shaanan": "lecturer", "Liam": "tutor", "Jianzhong": "coordinator"}
(key, value)

- add(key, value) - add a value with key to the dictionary (d[key] = value)
- get(key) - get the value stored with key (d[key])
- remove(key) - delete a key value pair (del d[key])

array -- O(n) add (linear search), O(n) remove, O(n) contains (linear search)
sorted array -- O(n) add (insertionsort), O(n) remove, O(log n) contains (binary search)
linked list -- O(n) add (linear search), O(n) remove, O(n) contains (linear search)
balanced BST O(log n) for everything! -sort by key

Hash Table -- O(1) for everything!

Dictionaries (ADT)

- create_new()
- **insert**(D, key, item) (want O(1))
- item <- **search**(D, key) (want O(1))
- **delete**(D, key) (want O(1))
- free()

```
{
    "Pride and Prejudice": "Alice",
    "Wuthering Heights": "Alice",
    "Great Expectations": "John"
}
```

Dictionaries (ADT)

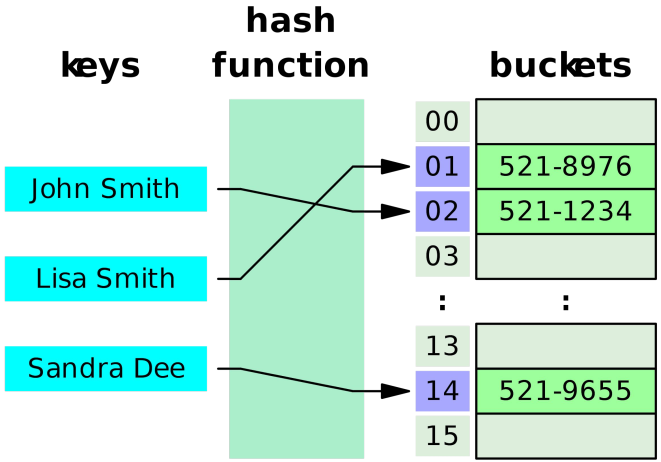
- create_new()
- **insert**(D, key, item)   (want O(1))
- item <- **search**(D, key) (want O(1))
- delete(D, key)   (want O(1))

```
{
    "Pride and Prejudice": "Alice",
    "Wuthering Heights": "Alice",
    "Great Expectations": "John"
}
```

Underlying data structure	Lookup		Insertion		Deletion		Ordered
	average	worst case	average	worst case	average	worst case	
Hash table	O(1)	O(*n*)	O(1)	O(*n*)	O(1)	O(*n*)	No
Self-balancing binary search tree	O(log *n*)	O(log *n*)	O(log *n*)	O(log *n*)	O(log *n*)	O(log *n*)	Yes
unbalanced binary search tree	O(log *n*)	O(*n*)	O(log *n*)	O(*n*)	O(log *n*)	O(*n*)	Yes
Sequential container of key–value pairs (e.g. association list)	O(*n*)	O(*n*)	O(1)	O(1)	O(*n*)	O(*n*)	No

Hash tables / hash map

- Maps **keys** to **values**.   (think python dictionary)
- Use a hash function to compute an **index** into an array of **buckets**
  - Ideally: each key is assigned into a unique bucket (**perfect hash function**)
  - We have **collisions**: hash function generates same index for >1 key.
  - There are ways to solve this!



	Average	Worst case
Insert	O(1)	O(n)
Search	O(1)	O(n)
Delete	O(1)	O(n)

Hash tables Issues

- **collisions**
  - hash function generates same index for multiple keys
  - can't store that element!
- **resizing**
  - run out of space in hashmap! => need to rehash everything => O(n)

h(x) = x%50
h(13) = 13
h(3) = 3
h(23) = 3

	Average	**Worst case**
Insert	O(1)	O(n)
Search	O(1)	O(n)
Delete	O(1)	O(n)

Load factor

$$\text{load factor} = \frac{n}{k}$$

`n   is the number of entries occupied in the hash table`

`k   is the number of buckets`

Higher load factor: greater chance of collision! (more buckets are full)
Lower load factor: wasted memory, and not necessarily any reduction in search cost

Java: aims for load factor of 0.75.
If the HashMap's load reaches 0.75, we **resize** the hashmap.

Resizing: double array size, need rehash all values => O(n) … bad!

Collision resolution

The way we handle collisions will erode the performance of the hash table.
We lose O(1)!

- Separate chaining
- Cuckoo hashing
- Linear probing

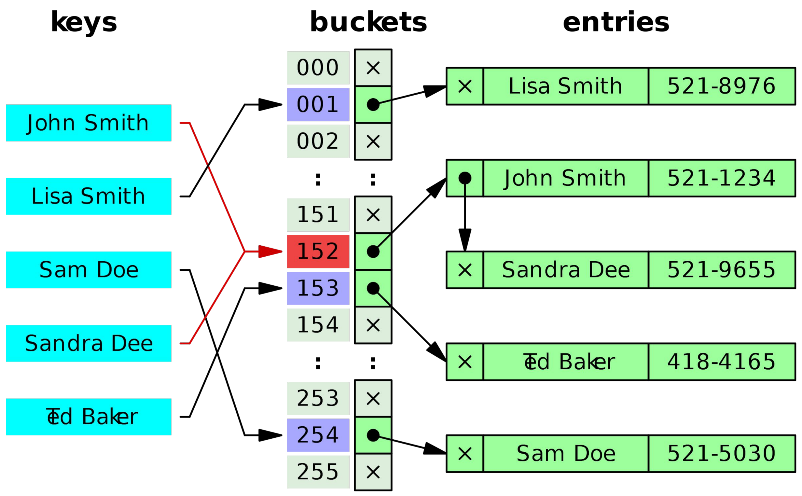
To consider: do we maintain O(1) insert/delete/search with these schemes?

Separate chaining

Instead of having buckets of values, have buckets of some **secondary data structure.**
(eg: linked list, array, another hashmap => all different properties)

- Each bucket has a list of entries with the same index.
- O(1) to find the correct bucket, then O(m) to search through the bucket:

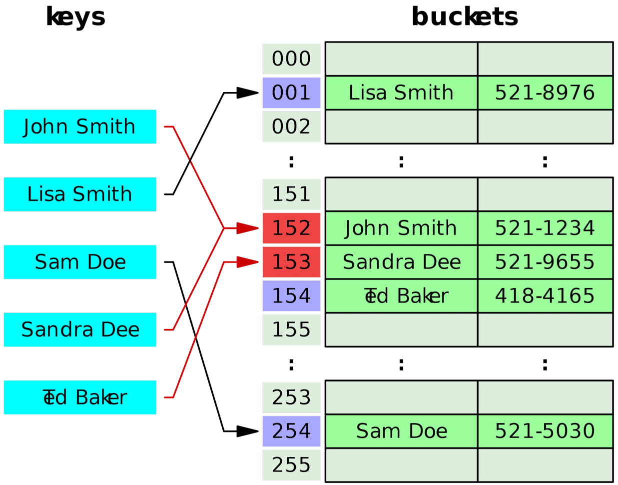
Searching through linked list: O(n)
Balanced binary tree: O(log n)
HashMap Java 8



Linear Probing

- Buckets are values as usual.
- If there is a collision, we **search linearly forward** until an unoccupied slot is found.   (eg: idx 100 full? go to 101. or 102. or 103...)

Time complexity: O(n) still, just resolves the collision part.
- How to delete elements? O(n) search. -- bad!



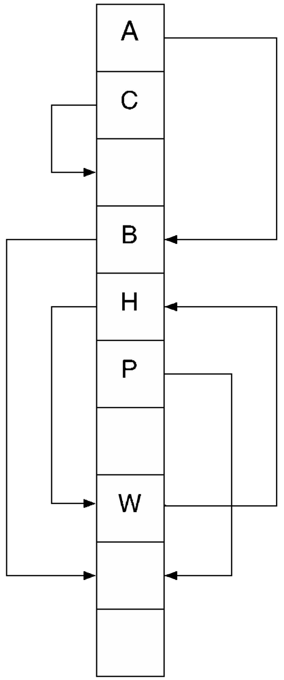
Cuckoo hashing

- Use multiple hash functions.
- Lookup: use hash function 1
  - If not found, use hash function 2
  - Etc
- Insertion: use hash function 1.
  - If collision, use hash function 2
  - Etc
- Deletion, same as lookup.

Worst case: O(1)
Careful with cycles!

$$h(6) = 6 \mod 11 = 6$$

$$h'(6) = \left\lfloor \frac{6}{11} \right\rfloor \mod 11 = 0$$



Cuckoo hashing - more details

```
on collision: displace the previous object to allow the new one to
be in its right spot, then re-insert it using a different hash function

  -   Use multiple hash functions.
  -   Lookup: use hash function 1
        -   If not found, use hash function 2.
  -   Insertion: use hash function 1.
        -   If collision, use hash function 2
  -   Deletion, same as lookup.
Worst case: O(1)
Efficient; but careful with cycles!
```

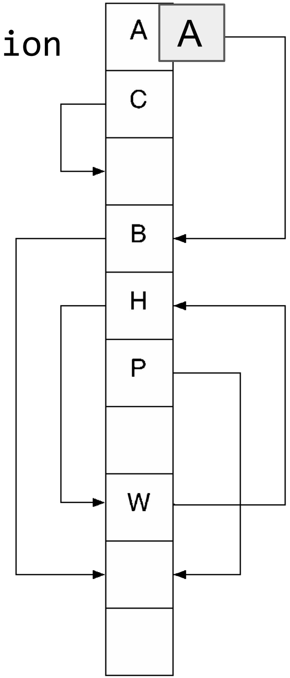
$$h(6) = 6 \mod 11 = 6$$

$$h'(6) = \left\lfloor \frac{6}{11} \right\rfloor \mod 11 = 0$$

insert A:
h(A) = 4   (H[4]=B)
h'(A) = 8 (H[8]=/)
H[8] := A

insert A:
h(A) = 4   (H[4]=B)
h'(B) = 8 (H[8]=/)
H[8] := B

6 hashes to itself...



Issues with hashing

- Collisions!
    - The way we handle collisions will erode the performance of the hash table. Loses O(1)!
    - Want to avoid collisions.

- If the hash table is nearly full -> more likely to have a collision!
    - If array is full -> guaranteed for collision!

- If the hash table is nearly empty -> wasting space!
    - Similar to having an array of size 1000.
      We want to start small and grow.

Solution:
If hash table is at <u>75%</u> capacity - resize it.
How to resize?
    Double the size of the array, AND need to **rehash** every element. Time?

Hash tables / hash map

- Maps **keys** to **values.**   (think python dictionary)
  - in memory, same as an array ("associative array")

- Use a <u>hash function</u> to compute an **index** into an array of **buckets**
- Ideally: each key is assigned into a unique bucket (**perfect hash function**)
  - eg: hash ints: h(x) = x % 10   (set N_BUCKETS = k = 10)

% modulo (remainder)

k=10 n=0

Insert into hash table: ~~2~~, 7, 85, 8, 28, 6.

13 / 10 = remainder? 3

k: # buckets, n: # elements in map

H			2	13		85		7	8 |28	
idx	0	1	2	3	4	5	6	7	8	9

Hash tables / hash map

- Use a <u>hash function</u> to compute an **index** into an array of **buckets**
- Ideally: each key is assigned into a unique bucket (**perfect hash function**)
  - eg: hash ints: h(x) = x % N_BUCKETS (set N_BUCKETS = k = 10)

```
h1(x) = x % 10;      k: # buckets;  n = num elements
h2(x) = x % 20;    chance of collision: 1/(k - n)
```

Insert into hash table: 2, 85, 8, 28, 6, 17. h(28) = 28 % 10 = 8

2 5 8 8 6 7 17

H			2			85	6		8 28	
								17		

idx	0	1	2	3	4	5	6	7	8	9