Object-oriented programming and data-structures

CS/ENGRD 2110
SUMMER 2018

Lecture 15: Hashing

http://courses.cs.cornell.edu/cs2110/2018su

Hash Functions

1 0 4 1 3

- Requirements:
  1) deterministic
  2) return a number in [0..n]

Hash Functions

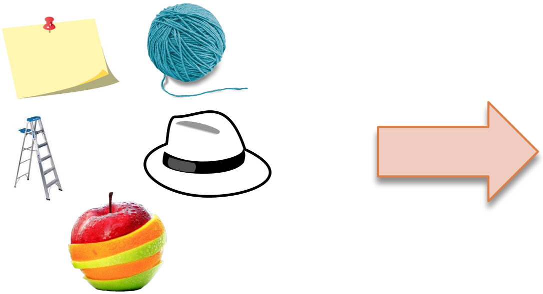
1 0 4 1 3

- Requirements:
  1) deterministic
  2) return a number in [0..n]

Which of the following functions f: Object -> int are hash functions:

a) f(x) = x
b) f(x) = x.hashCode()
c) f(x) = &x
d) f(x) = 0

Hash Functions



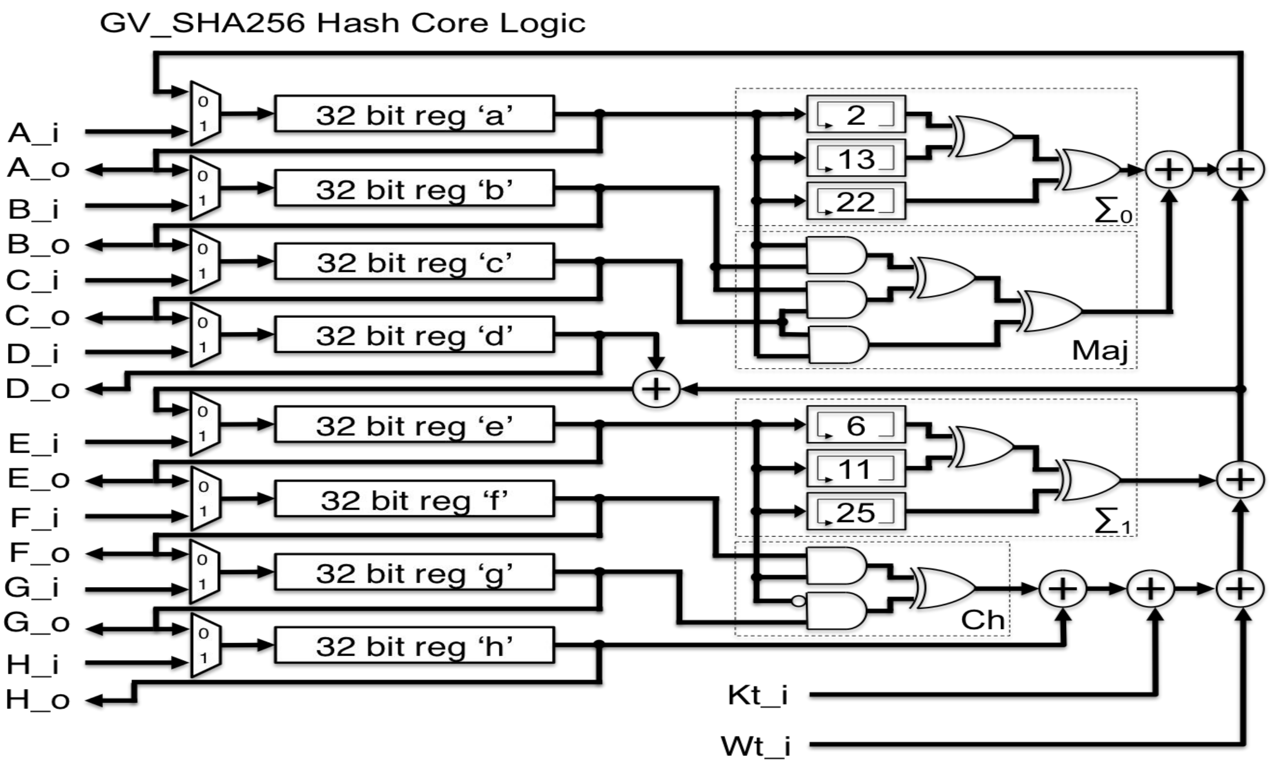
1 0 4 1 3

- Requirements:
  1) deterministic
  2) return a number in [0..n]
- Properties of a good hash:
  1) fast
  2) collision-resistant
  3) evenly distributed
  4) hard to invert

Example: hashCode()

- Method defined in java.lang.Object
- Default implementation: uses memory address of the object
  - If you override equals, you must override hashCode!
- String overrides hashCode()
  - s.hashCode() = s[0] * 31^(n-1) + s[1]*31^(n-2) + … + s[n-1]

Example: SHA-256

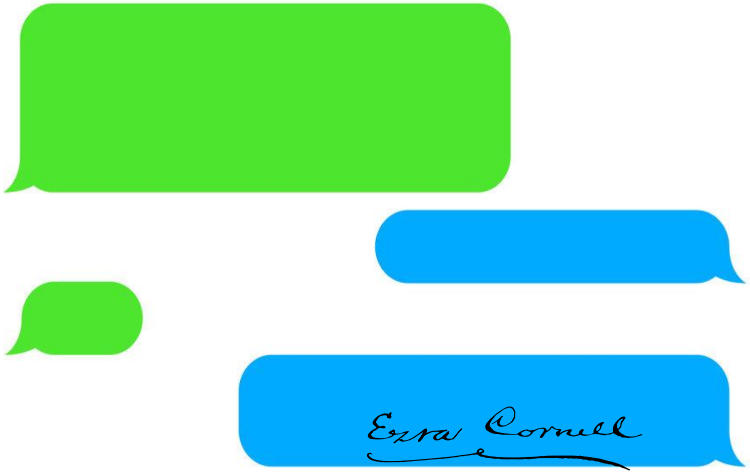


Application: Error Detection

Submitted	Date	By	Size	MD5 [What's this?]
A6GUI	April 10, 2018 04:28PM	[illegible]	10.82 kB	ca62dd8fc1273f51baa6f507efac1d2b

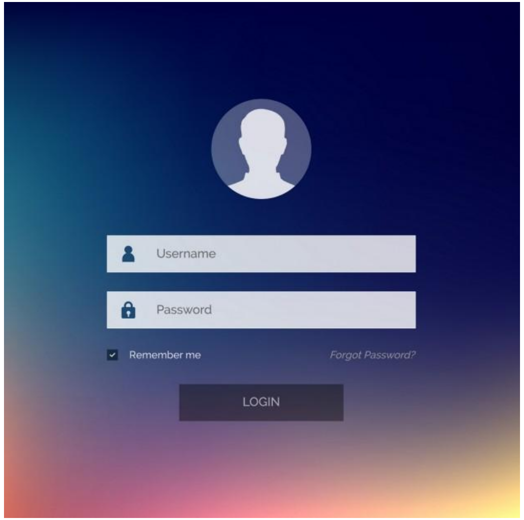
- Hash functions are used for error detection
- E.g., hash of uploaded file should be the same as hash of original file (if different, file was corrupted)

Application: Integrity



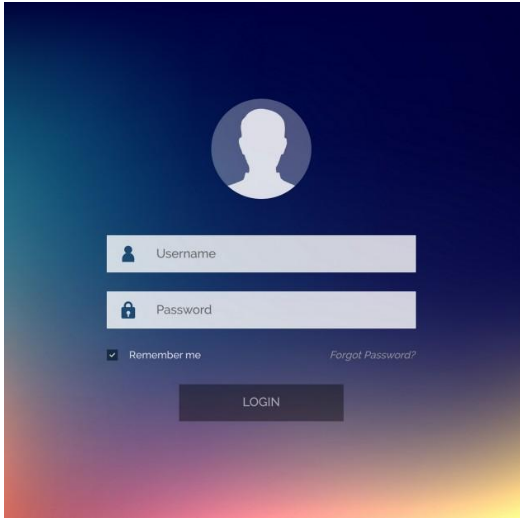
- Hash functions are used to "sign" messages
- Provides integrity guarantees in presence of an attacker
- Principals share some secret sk
- Send (m, h(m,sk))

Application: Password Storage



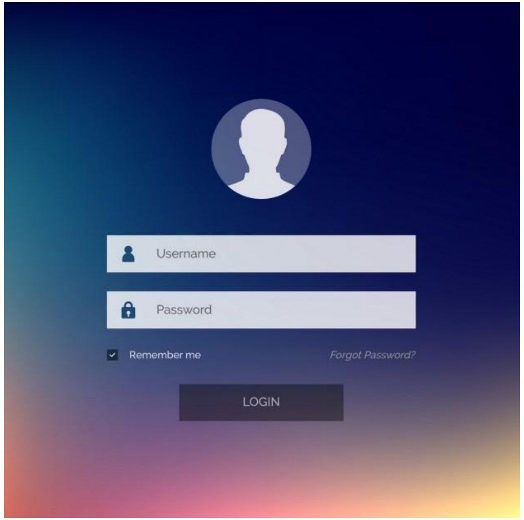
- Hash functions are used to store passwords

Application: Password Storage



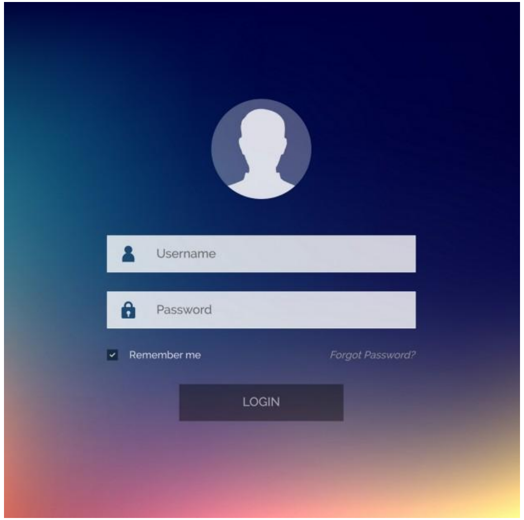
- Hash functions are used to store passwords
- Could store plaintext passwords
  - Problem: Password files get stolen

Application: Password Storage



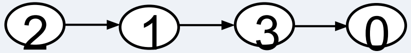
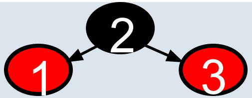
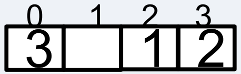
- Hash functions are used to store passwords
- Could store plaintext passwords
  - Problem: Password files get stolen
- Could store (username, h(password))
  - Problem: password reuse

Application: Password Storage

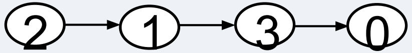
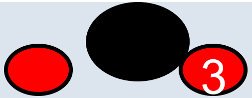
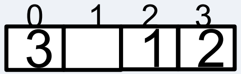


- Hash functions are used to store passwords
- Could store plaintext passwords
  - Problem: Password files get stolen
- Could store (username, h(password))
  - Problem: password reuse
- Instead, store
  - (username, s, h(password, s))

Application: Hash Set

Data Structure	add(val x)	lookup(int i)	find(val x)
ArrayList 	$O(n)$	$O(1)$	$O(n)$
LinkedList 	$O(1)$	$O(n)$	$O(n)$
TreeSet 	$O(\log n)$		$O(\log n)$
HashSet 	$O(1)$		$O(1)$

Application: Hash Set

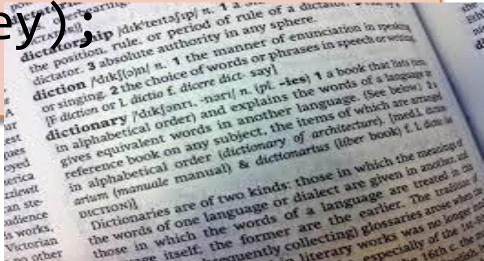
Data Structure	add(val x)	lookup(int i)	find(val x)
ArrayList 	$O(n)$	$O(1)$	$O(n)$
LinkedList 	$O(1)$	$O(n)$	$O(n)$
TreeSet 	$O(\log n)$		$O(\log n)$
HashSet 	$O(1)$		$O(1)$

Expected time
Worst-case: $O(n)$

HashSet and HashMap

```
Set<V>{

   boolean add(V value);

   boolean contains(V value);

   boolean rem
}
```

```
Map<K,V>{

   V put(K key, V value);

   V get(K key);

   V remove(K key);
}
```



Recall: Array Lists

- Finding an element in an ArrayList takes constant time when we know the index in the element
  - O(1)
- Unfortunately, if I want to determine whether “Donkey” is the set, I don’t know where “Donkey” could be”
  - So must search all the elements O(n)

Recall: Array Lists

- Finding an element in an ArrayList takes constant time when we know the index in the element
  - O(1)
- Unfortunately, if I want to determine whether "Donkey" is the set, I don't know where "Donkey" could be"
  - So must search all the elements O(n)
- Could hash functions somehow help us?

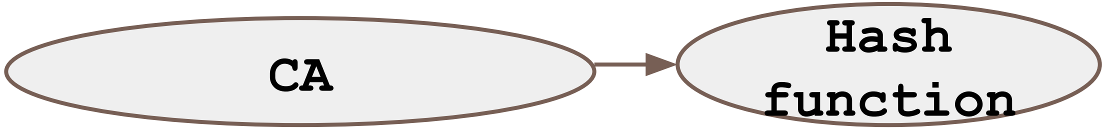
Hash Tables

- Finding an element in an array takes constant time when know which index is stored in.
- Recall that hash functions map objects to a number and are deterministic

Hash Tables

- Finding an element in an array takes constant time when know which index is stored in.

- Recall that hash functions map objects to a number and are deterministic

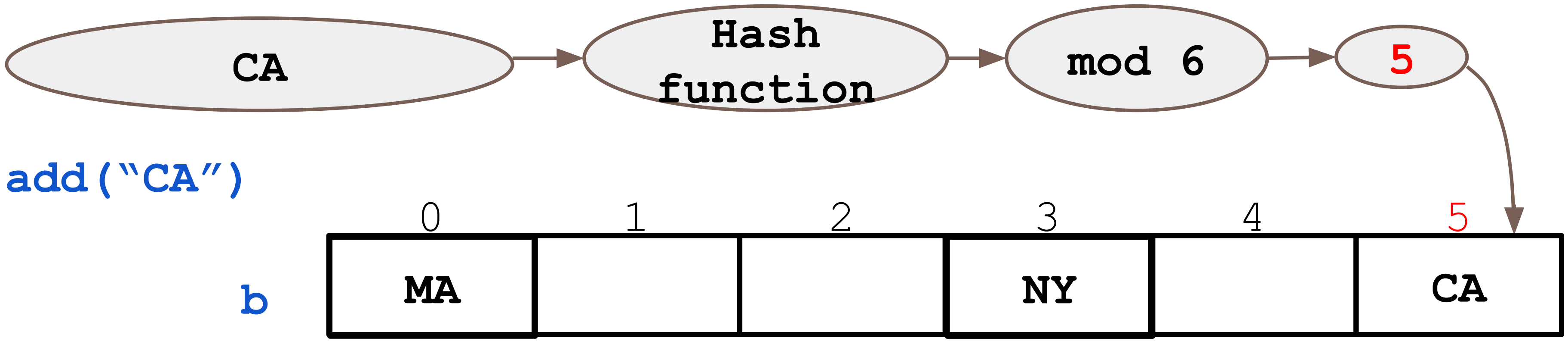


`add("CA")`

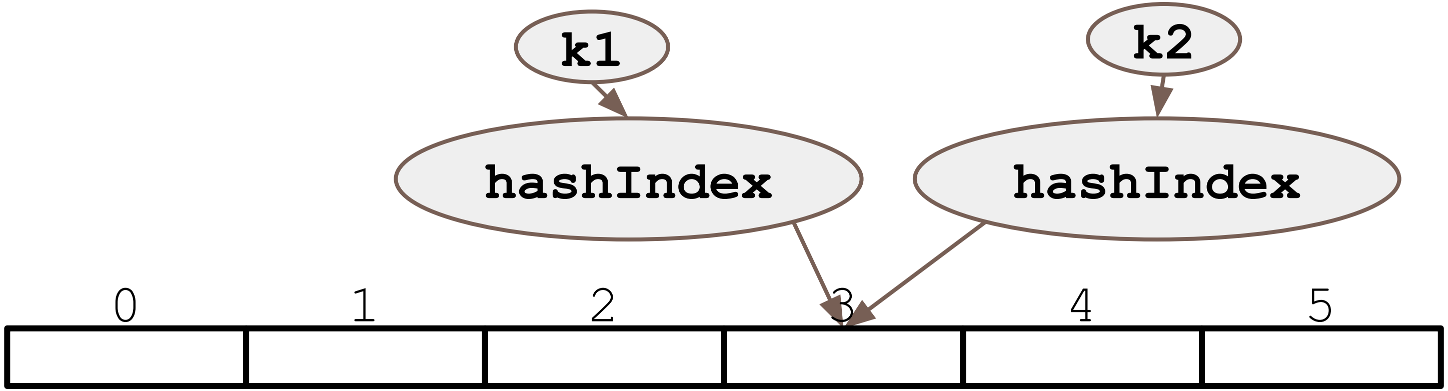
Hash Tables

- Finding an element in an array takes constant time when know which index is stored in.

- Recall that hash functions map objects to a number and are deterministic



So what goes wrong?



Can we have perfect hash functions?

- Perfect hash functions map each value to a different index in the hash table

Can we have perfect hash functions?

- Perfect hash functions map each value to a different index in the hash table

- Impossible in practice
  - don't know size of the array
  - Number of possible values far far exceeds the array size
    - Want array size proportional to actual number of keys, not number of possible keys
  - no point in a perfect hash function if it takes too much time to compute

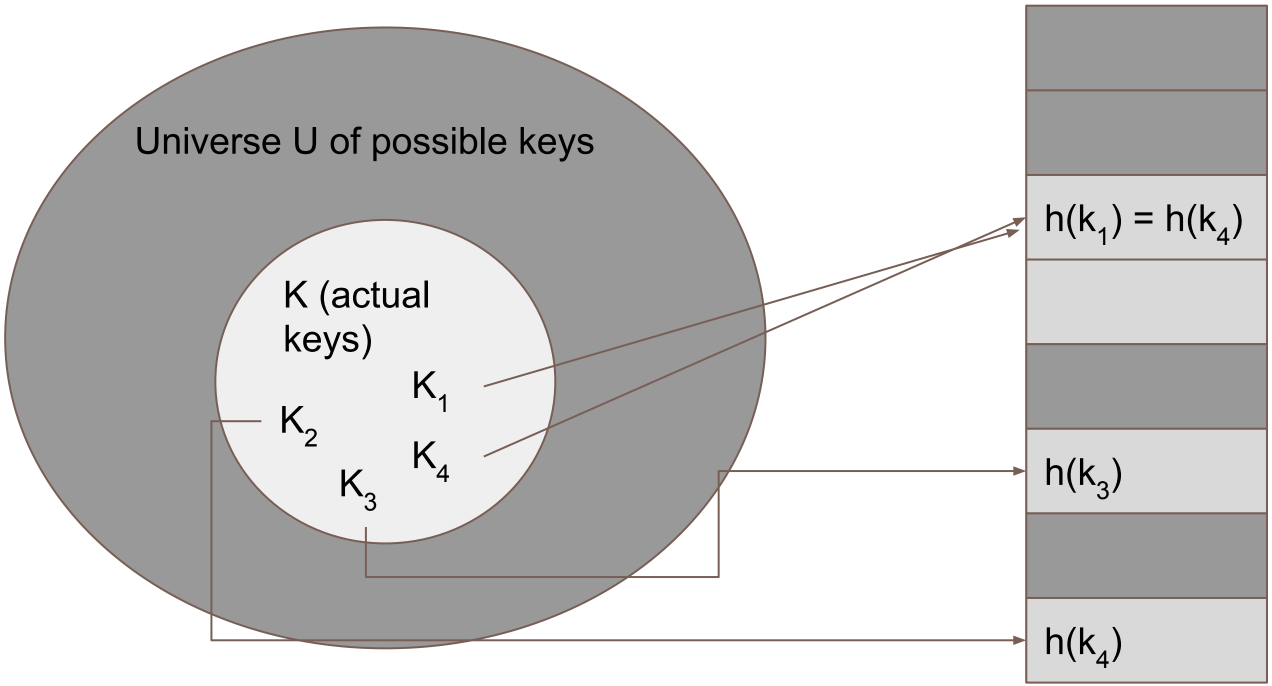
Can we have perfect hash functions?

- Perfect hash functions map each value to a different index in the hash table

- Impossible in practice
  - don’t know size of the array
  - Number of possible values far far exceeds the array size
    - Want array size proportional to actual number of keys, not number of possible keys
  - no point in a perfect hash function if it takes too much time to compute

- All hash functions will have **collisions**

Graphically



Want to minimise both the size of the array and the risk of collisions!

Load Factor

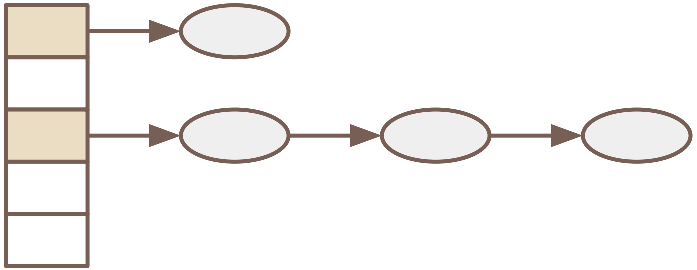
Load factor

$$\lambda = \frac{\#\ of\ entries}{length\ of\ array}$$

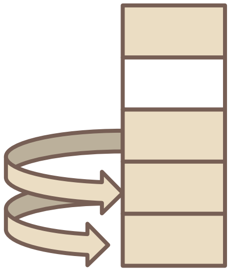
Collision Resolution

Two ways of handling collisions:

1. Chaining

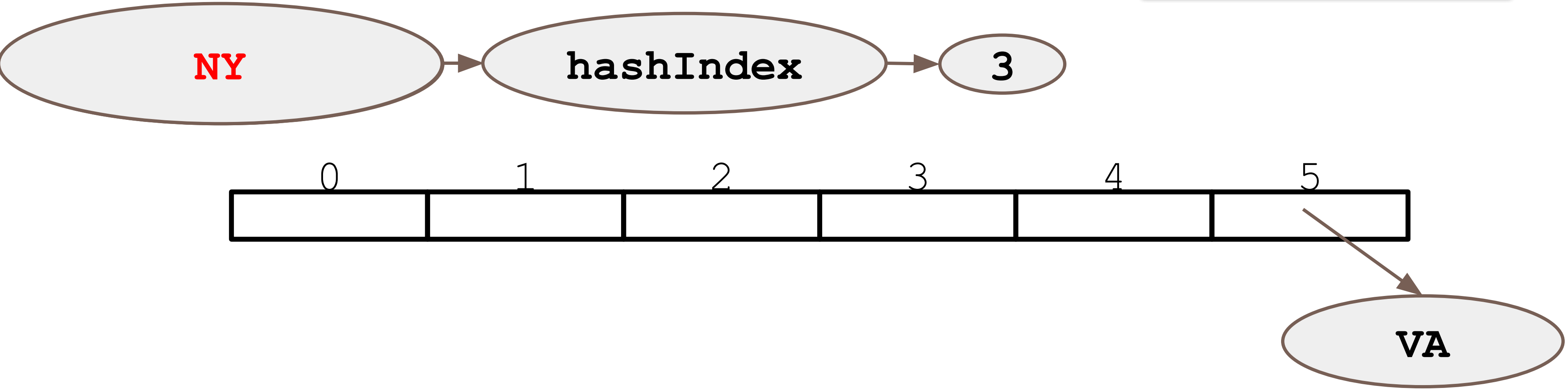


2. Open Addressing



Chaining

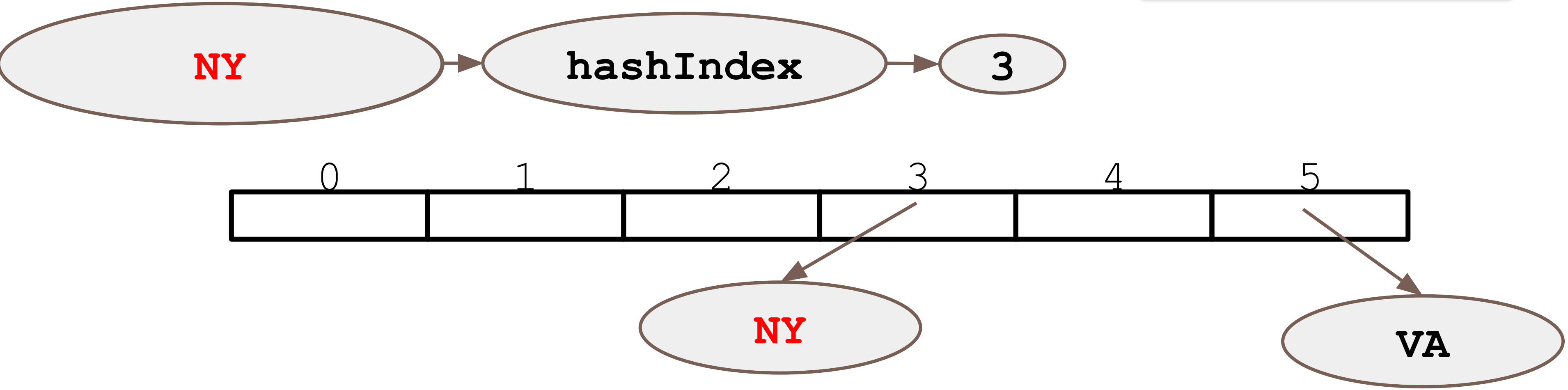
```
add("NY")
add("CA")
lookup("CA")
```



Place all the elements that hash to the same slot into the same linked list

Chaining

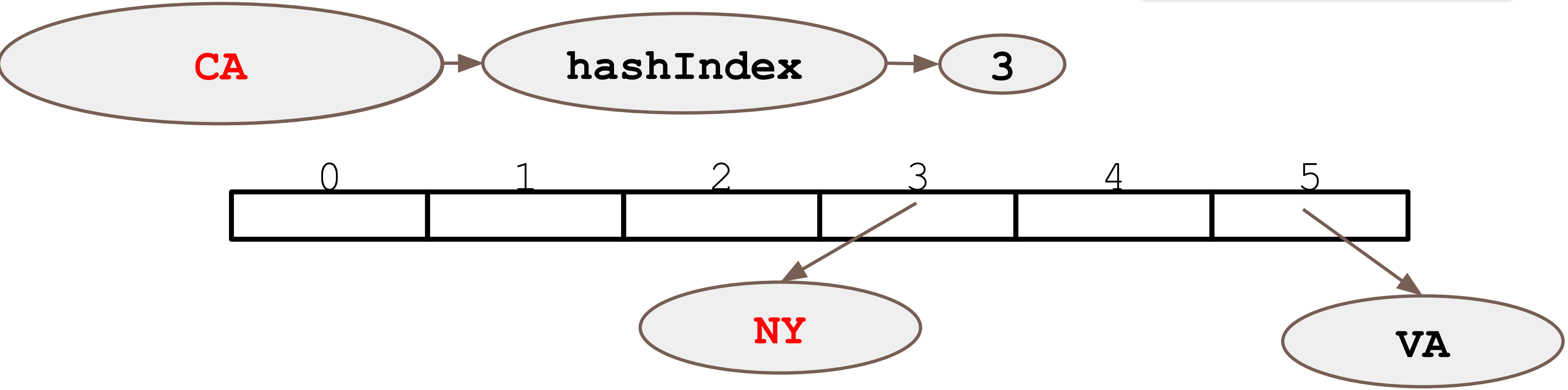
```
add("NY")
add("CA")
lookup("CA")
```



Place all the elements that hash to the same slot into the same linked list

Chaining

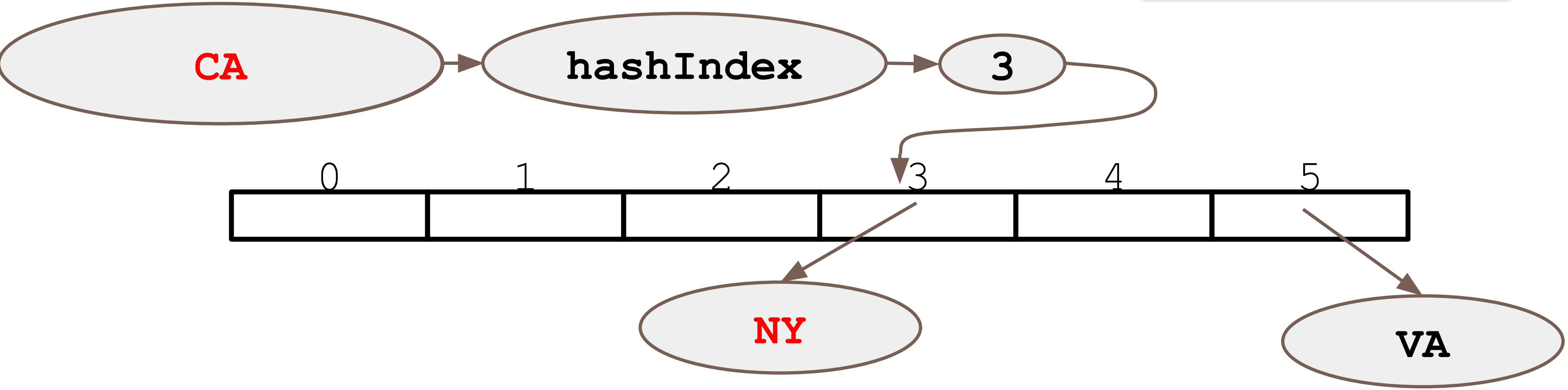
```
add("NY")
add("CA")
lookup("CA")
```



Place all the elements that hash to the same slot into the same linked list

Chaining

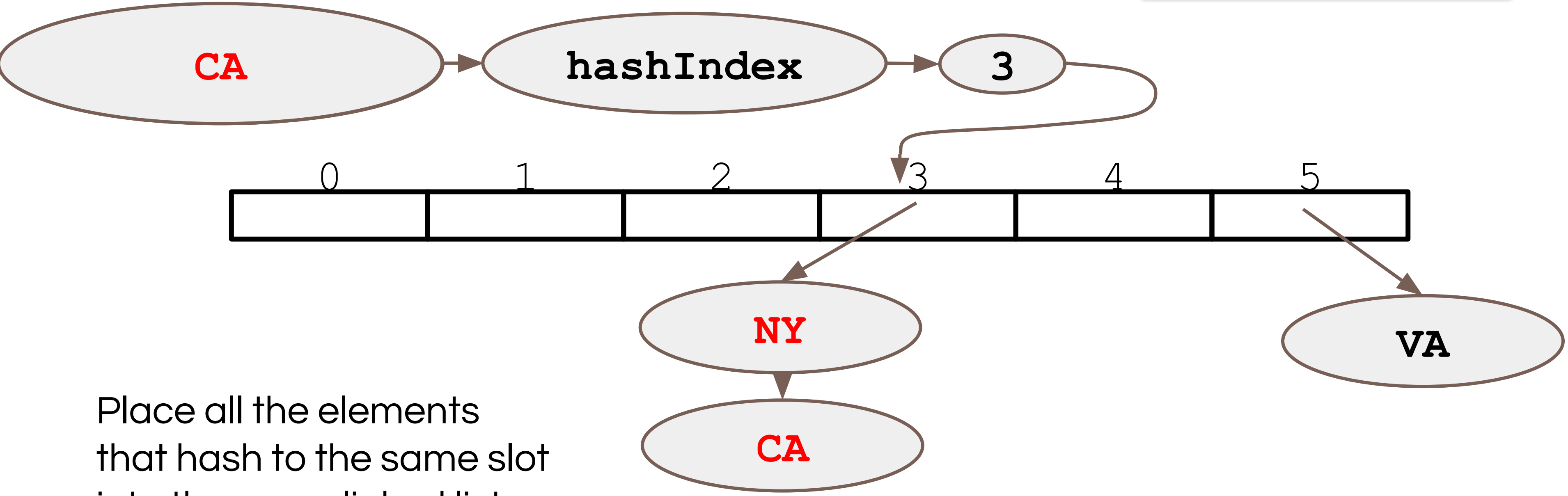
```
add("NY")
add("CA")
lookup("CA")
```



Place all the elements that hash to the same slot into the same linked list

Chaining

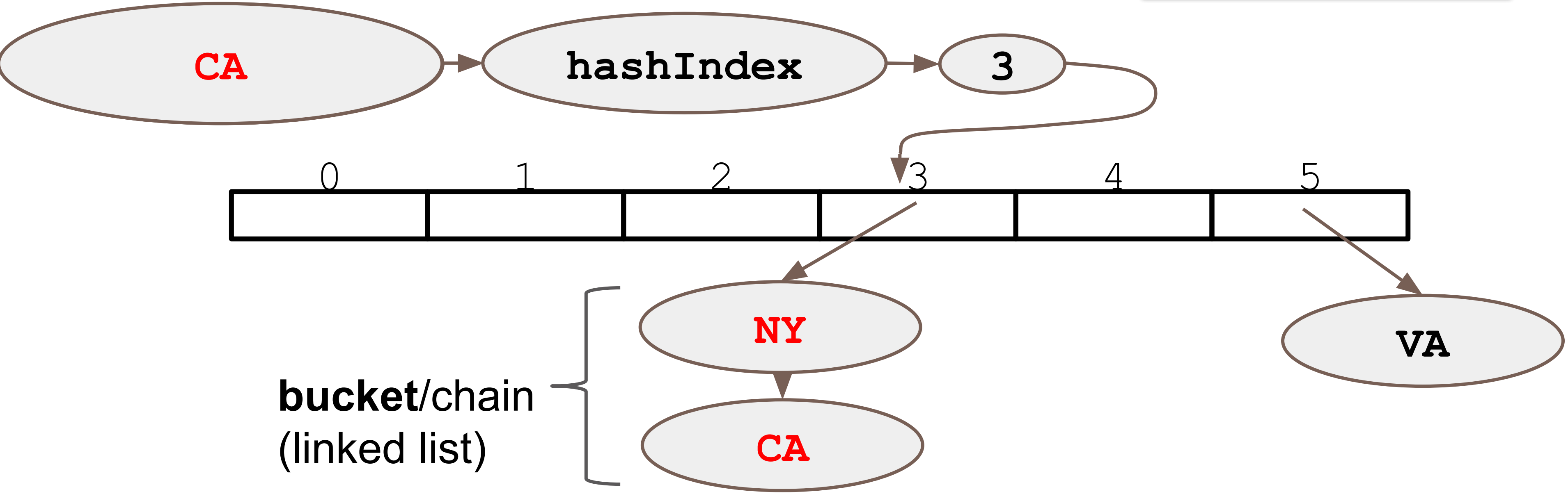
```
add("NY")
add("CA")
lookup("CA")
```



Place all the elements that hash to the same slot into the same linked list

Chaining

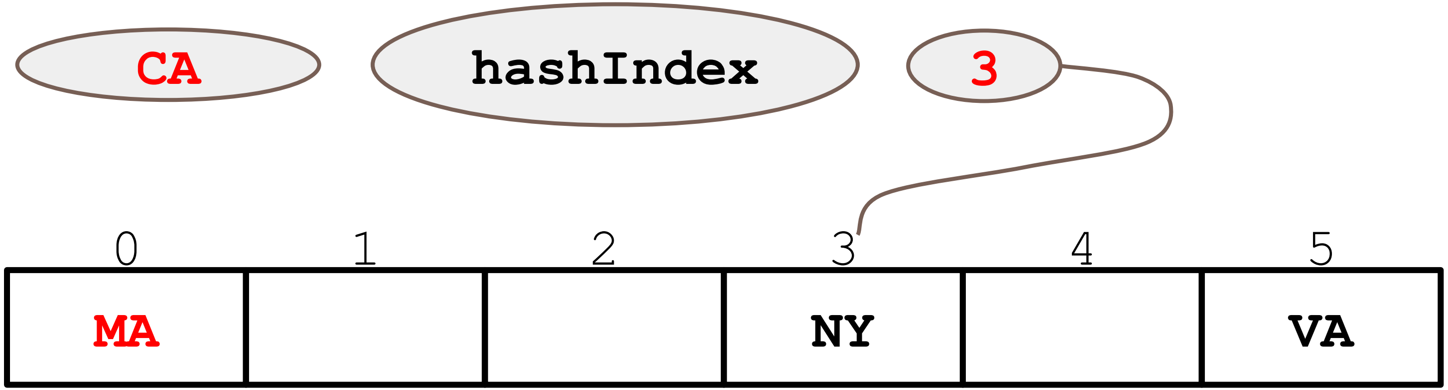
```
add("NY")
add("CA")
lookup("CA")
```



Open Addressing

Probing: Find another available space in the array

`add("CA")`



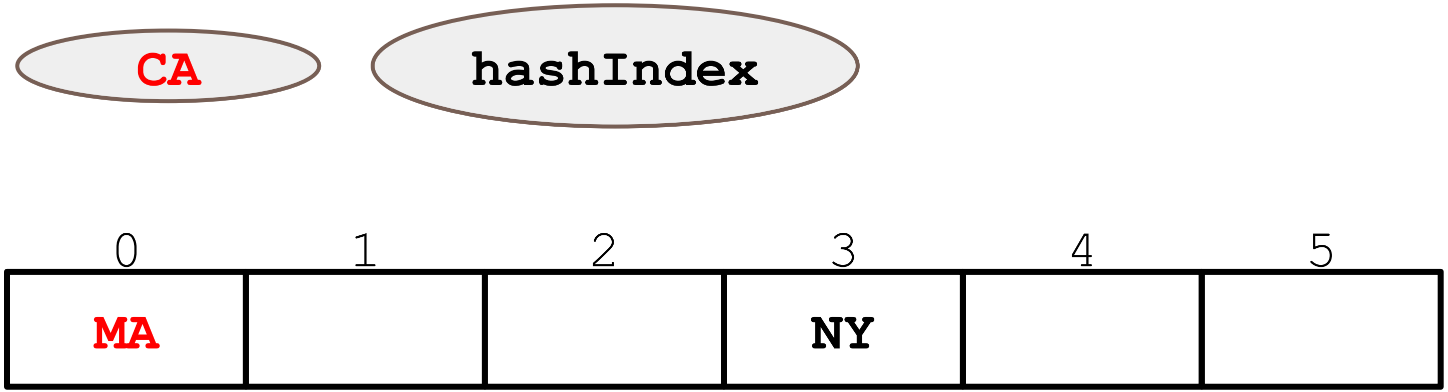
Open Addressing

- All elements occupy the hash table itself
- Each entry contains either an element of the set or NULL
- When searching for an element, systematically examine table slots until either we find the desired element, or know that the element is not in the set.
- No nodes are stored outside of the hash table, so table can fill up

Open Addressing

Probing: Successively probe the hash table until we find an empty slot in which to put the key.

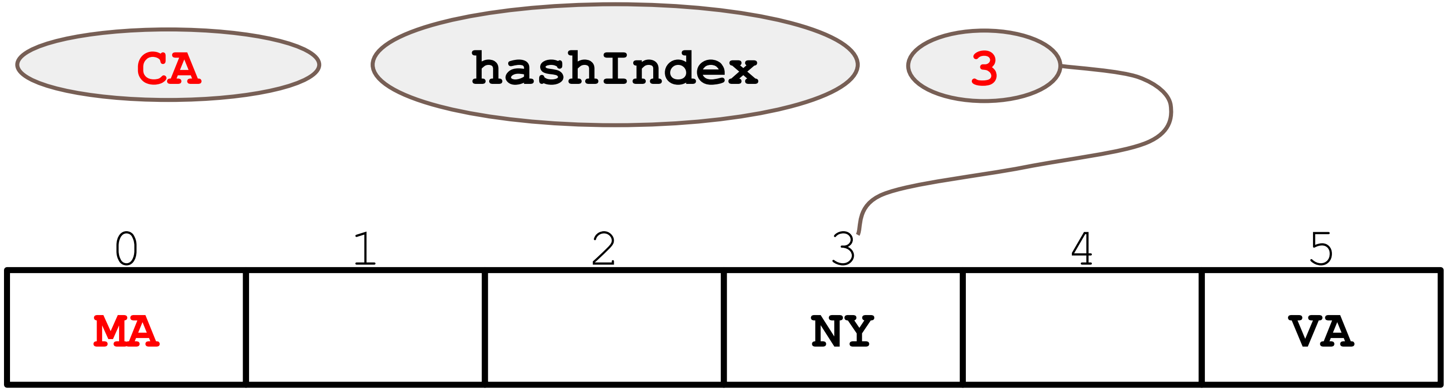
`add("CA")`



Open Addressing

Probing: Successively probe the hash table until we find an empty slot in which to put the key.

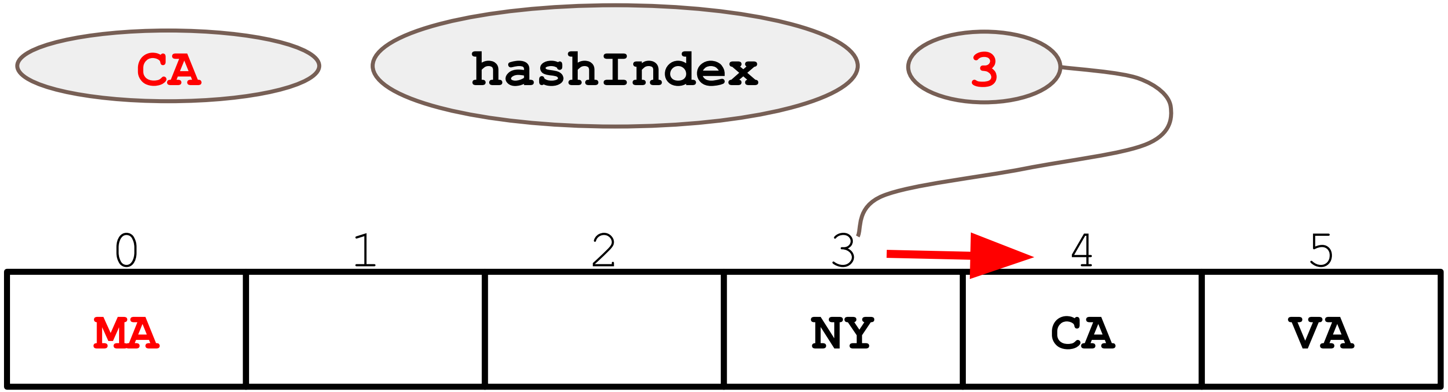
`add("CA")`



Open Addressing

Probing: Successively probe the hash table until we find an empty slot in which to put the key.

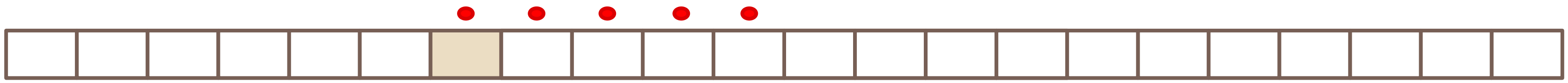
`add("CA")`



Different probing strategies

When a collision occurs, how do we search for an empty space?

***linear probing*:**
search the array in order, starting from h(x):
i, i+1, i+2, i+3 . . .

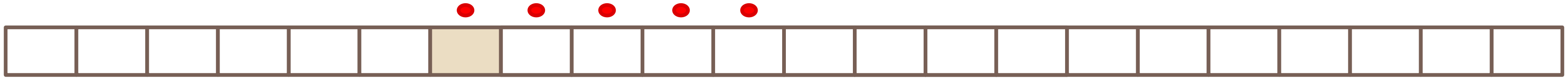


Different probing strategies

When a collision occurs, how do we search for an empty space?

***linear probing*:**
search the array in order, starting from h(x):
i, i+1, i+2, i+3 . . .

***Problem of clustering*:**
problem where nearby hashes have very similar probe sequence so we get more collisions



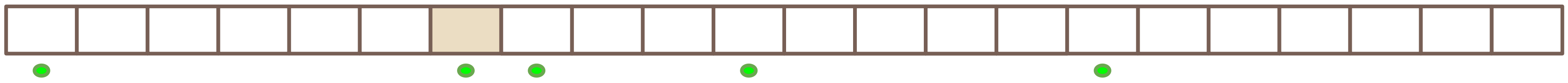
Long runs of occupied slots build up, increasing the average search time

The bigger the cluster gets, the faster it grows!

Different probing strategies

When a collision occurs, how do we search for an empty space?

***quadratic probing*:** search the array in nonlinear sequence:
$i, i+1^2, i+2^2, i+3^2$ . . .

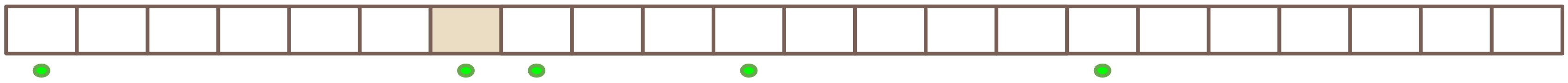


Different probing strategies

When a collision occurs, how do we search for an empty space?

***quadratic probing*:** search the array in nonlinear sequence:
$i, i+1^2, i+2^2, i+3^2 \ldots$

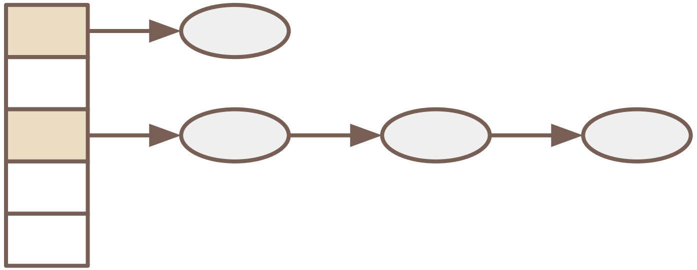
Idea is to probe more widely separated cells, instead of those adjacent to the primary hash site.



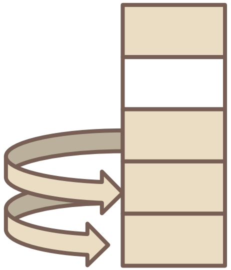
Collision Resolution

Two ways of handling collisions:

1. Chaining



2. Open Addressing



Load factor increases

Load factor

$$\lambda = \frac{\#\ of\ entries}{length\ of\ array}$$

- What happens when the load factor increases?

Load factor increases

Load factor

$$\lambda = \frac{\#\ of\ entries}{length\ of\ array}$$

- What happens when the load factor increases?
  - For the chaining method?

Load factor increases

Load factor

$$\lambda = \frac{\#\ of\ entries}{length\ of\ array}$$

- What happens when the load factor increases?
  - For the chaining method?
    - Always possible to insert new elements, but the chains become longer.
    - Operations slowdown

Load factor increases

Load factor

$$\lambda = \frac{\#\ of\ entries}{length\ of\ array}$$

- What happens when the load factor increases?
  - For the chaining method?
    - Always possible to insert new elements, but the chains become longer.
    - Operations slowdown
  - For the open addressing?

Load factor increases

Load factor

$$\lambda = \frac{\#\ of\ entries}{length\ of\ array}$$

- What happens when the load factor increases?
  - For the chaining method?
    - Always possible to insert new elements, but the chains become longer.
    - Operations slowdown
  - For the open addressing?
    - Clustering causes operations to slowdown
    - Eventually impossible to insert

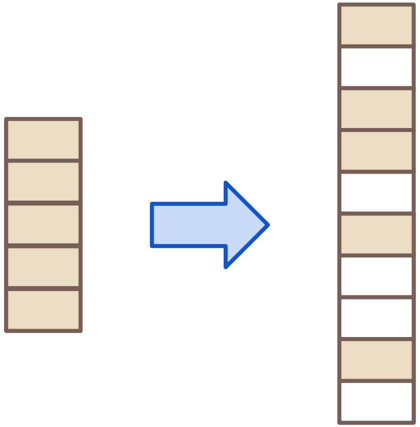
Resizing

Solution: ***Dynamic resizing***

Resizing

Solution: ***Dynamic resizing***

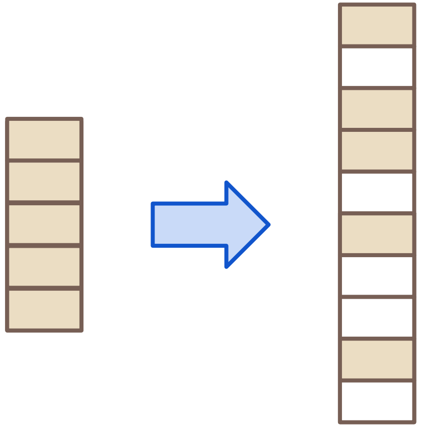
- Double the size.
- Reinsert / rehash all elements to new array



Resizing

Solution: ***Dynamic resizing***

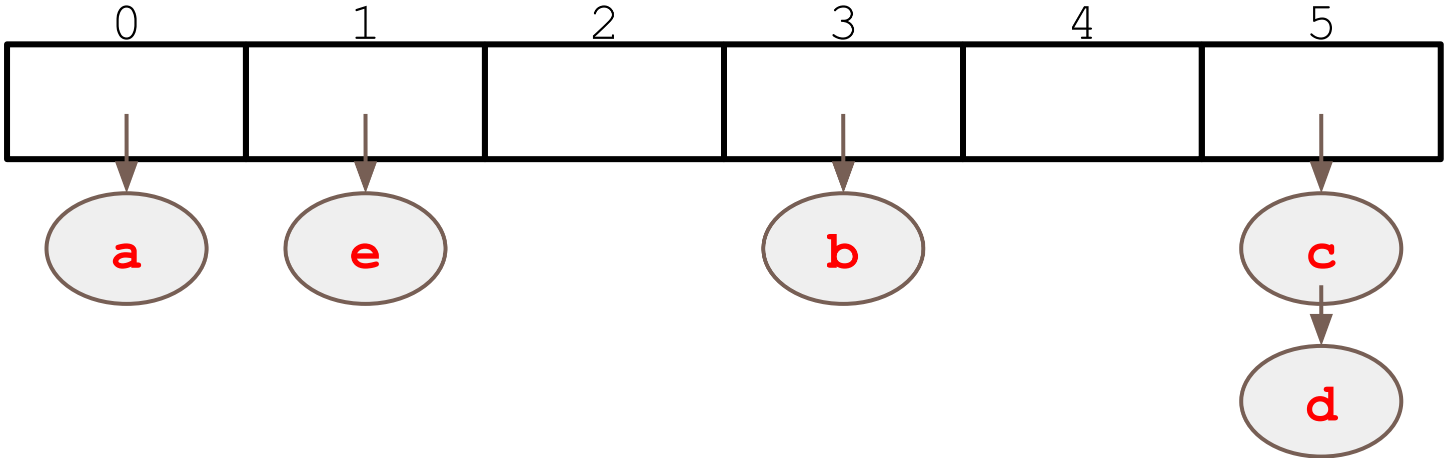
- Double the size.
- Reinsert / rehash all elements to new array
- Why not simply copy into first half?



Let's try it

Insert the following elements (in order) into an array of size 6:

element	a	b	c	d	e
hashCode	0	9	17	11	19



Let's try it

Insert the following elements (in order) into an array of size 6:

element	a	b	c	d	e
hashCode	0	9	17	11	19

0	1	2	3	4	5
a	d	e	b		c

Note: Using linear probing, no resizing

Let's try it

Insert the following elements (in order) into an array of size 6:

element	a	b	c	d	e
hashCode	0	9	17	11	19

0	1	2	3	4	5

What is the final state of the hash table if you use open addressing with quadratic probing (assume no resizing)?

Let's try it

Insert the following elements (in order) into an array of size 6:

element	a	b	c	d	e
hashCode	0	9	17	11	19

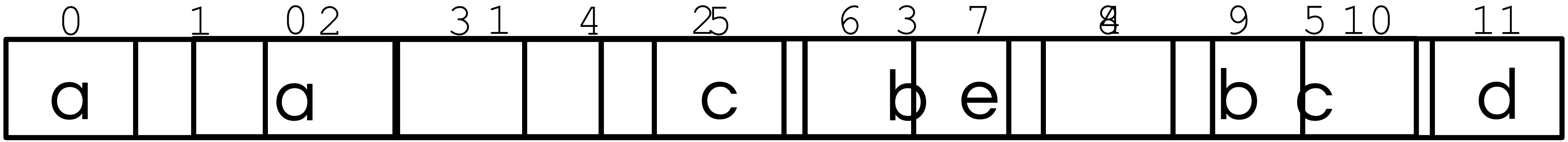
0	1	2	3	4	5
a	e	d	b		c

Note: Using quadratic probing, no resizing

Let's try it

Insert the following elements (in order) into an array of size 6:

element	a	b	c	d	e
hashCode	0	9	17	11	19



Note: Using quadratic probing, resizing if load > ½

Worst Case Time Complexity

Collision Handling	put(v)	get(v)	remove(v)
Chaining			
Open Addressing			

Worst Case Time Complexity

Collision Handling	put(v)	get(v)	remove(v)
Chaining	O(1)	O(n)	O(n)
Open Addressing	O(n)	O(n)	O(n)

Weren't hashsets designed to improve complexity? No better than a linked list!

Worst Case Time Complexity

Collision Handling	put(v)	get(v)	remove(v)
Chaining	O(1)	O(n)	O(n)
Open Addressing	O(n)	O(n)	O(n)

Weren't hashsets designed to improve complexity? No better than a linked list!

Hashsets are an example of a datastructure where we care about **average time complexity**, not worst time.

Recall: Load Factor

Load factor

$$\lambda = \frac{\#\ of\ entries}{length\ of\ array}$$

Gold Standard for Hash Function

- A good hash function satisfies (approximately) the assumption of **simple uniform hashing**:
  - Each key is equally likely to hash to any of the m slots, independently of where any other key has hashed to

Gold Standard for Hash Function

- A good hash function satisfies (approximately) the assumption of **simple uniform hashing**:
  - Each key is equally likely to hash to any of the m slots, independently of where any other key has hashed to

- Unfortunately:
  - Hard to check
  - Rarely know the key distribution

Average Complexity of Chaining

- How do we compute the average complexity of chaining?

Average Complexity of Chaining

- How do we compute the average complexity of chaining?

- Complexity of get/remove is:
  - The cost of computing the hash function
  - The cost of finding the element in the chain

Average Complexity of Chaining

- How do we compute the average complexity of chaining?

- Complexity of get/remove is:
  - The cost of computing the hash function O(1)
  - The cost of finding the element in the chain O (avg length of chain)

Average Complexity of Chaining

- How do we compute the average complexity of chaining?

- Complexity of get/remove is:
  - The cost of computing the hash function O(1)
  - The cost of finding the element in the chain O (avg length of chain)

- What is the average length of the chain?

Average Complexity of Chaining

- How do we compute the average complexity of chaining?

- Complexity of get/remove is:
  - The cost of computing the hash function O(1)
  - The cost of finding the element in the chain O (avg length of chain)

- What is the average length of the chain?
  - Assume uniform hashing: every entry equally likely to end up in a slot in the array

Average Complexity of Chaining

- How do we compute the average complexity of chaining?

- Complexity of get/remove is:
  - The cost of computing the hash function O(1)
  - The cost of finding the element in the chain O (avg length of chain)

- What is the average length of the chain?
  - Assume uniform hashing: every entry equally likely to end up in a slot in the array
  - If m slots and n entries, uniform distribution with probability n/m

Average Complexity of Chaining

- How do we compute the average complexity of chaining?

- Complexity of get/remove is:
  - The cost of computing the hash function O(1)
  - The cost of finding the element in the chain O (avg length of chain)

- What is the average length of the chain?
  - Assume uniform hashing: every entry equally likely to end up in a slot in the array
  - If m slots and n entries, uniform distribution with probability n/m
  - Length of chain is the **expectation** of a uniform distribution

Average Complexity of Chaining

- How do we compute the average complexity of chaining?
- Complexity of get/remove is:
  - The cost of computing the hash function O(1)
  - The cost of finding the element in the chain O (avg length of chain)
- What is the average length of the chain?
  - Assume uniform hashing: every entry equally likely to end up in a slot in the array
  - If m slots and n entries, uniform distribution with probability n/m
  - Length of chain is the **expectation** of a uniform distribution
  - Expectation is n/m, so expectation is ƛ

Average Time Complexity

Collision Handling	put(v)	get(v)	remove(v)
Chaining	O(1)	O(1 + ƛ)	O(1 + ƛ)
Open Addressing			

(Ignoring Resizing)

Average Complexity of OpenAddr

- How do we compute the average complexity of chaining?
  - Must compute the average number of probes.

Average Complexity of OpenAddr

- How do we compute the average complexity of chaining?
  - Must compute the average number of probes.
- How many probes do we do?

Average Complexity of OpenAddr

- How do we compute the average complexity of chaining?
  - Must compute the average number of probes.
- How many probes do we do?
  - We always have to probe the first location

Average Complexity of OpenAddr

- How do we compute the average complexity of chaining?
  - Must compute the average number of probes.
- How many probes do we do?
  - We always have to probe the first location
  - With probability ƛ, first location is full, have to probe again

Average Complexity of OpenAddr

- How do we compute the average complexity of chaining?
  - Must compute the average number of probes.

- How many probes do we do?
  - We always have to probe the first location
  - With probability ƛ, first location is full, have to probe again
  - With probability ƛ^2, second location is also have, have to probe yet again
  - …

Average Complexity of OpenAddr

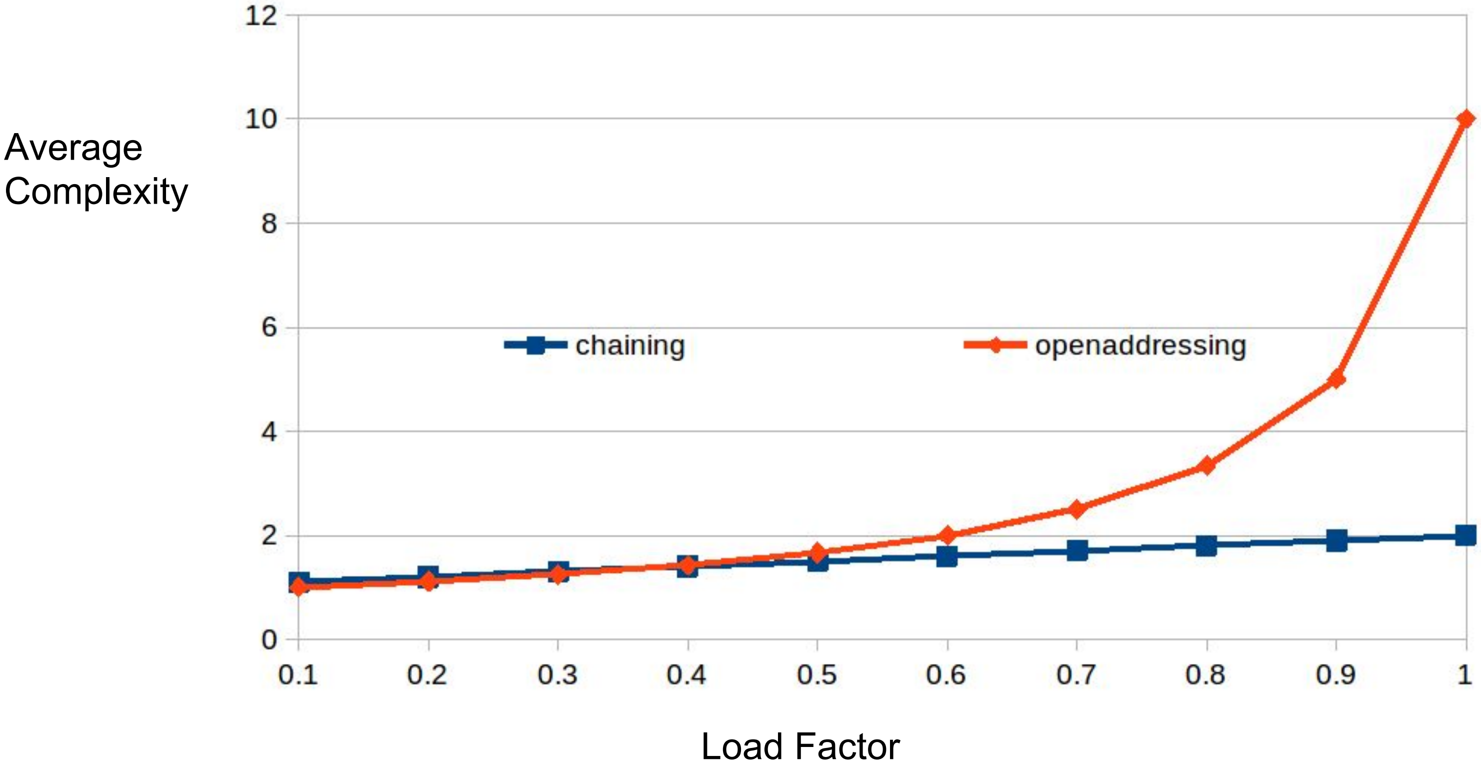
- How do we compute the average complexity of chaining?
  - Must compute the average number of probes.
- How many probes do we do?
  - We always have to probe the first location
  - With probability ƛ, first location is full, have to probe again
  - With probability ƛ^2, first two locations are full, have to probe yet again
  - …
- Expected number of probes = 1 + ƛ + ƛ^2 + ƛ^3 … = 1 / (1 - ƛ)

Average Time Complexity

Collision Handling	put(v)	get(v)	remove(v)
Chaining	O(1)	O(1 + λ)	O(1 + λ)
Open Addressing	O(1+ 1/1-λ)	O(1+ 1/1-λ)	O(1+ 1/1-λ)

(Ignoring Resizing)

Average Complexity Compared



Still no resizing!

Collision Resolution Summary

Chaining

- store entries in separate chains (linked lists)
- can have higher load factor/degrades gracefully as load factor increases

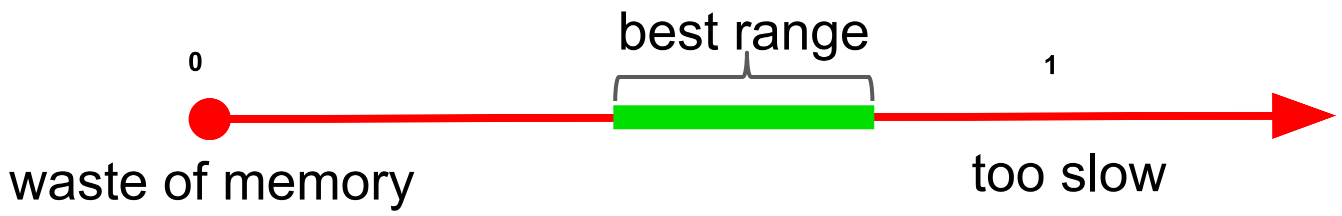
Open Addressing

- store all entries in table
- use linear or quadratic probing to place items
- uses less memory
- clustering can be a problem — need to be more careful with choice of hash function

Ideal Load Factor

Load factor

$$\lambda = \frac{\#\ of\ entries}{length\ of\ array}$$



Assume Constant Load Factor!

Collision Handling	put(v)	get(v)	remove(v)
Chaining	O(1)	O(1)	O(1)
Open Addressing	O(1)	O(1)	O(1)

If we assume constant load factor, then all operations take constant time.

But assuming constant load factor requires **resizing the array**, and this does not take constant time!

Amortised Analysis to the rescue!

- In an **amortised analysis**, the time required to perform a sequence of operations is averaged over all the operations

- Can be used to calculate the **average cost** of an operation

gimme! coffee

VS.

Amortised Analysis to the rescue!

- Assume dynamic resizing with load factor ƛ = 1/2
- Most put operations take (expected) time O(1)
- If i = 2^j, put takes time O(i)
  - Start with an array of size 2, and then double every time reaches half full
- Total time to perform n put operations is
  - N * O(1) + O(2^0 + 2^1 + 2^2 + … + 2^j)
- Average time to perform 1 put operation is
  - O(1) + O(1/ 2^j + 1/ 2^(j-1) + … + ¼ + ½ + 1) = O(1)

Amortised Analysis (with resize)

Collision Handling	put(v)	get(v)	remove(v)
Chaining	O(1)	O(1)	O(1)
Open Addressing	O(1)	O(1)	O(1)

Can we do better?

Collision Handling	put(v)	get(v)	remove(v)
Chaining	O(1)	O(1)	O(1)
Open Addressing	O(1)	O(1)	O(1)

Can we somehow **bound** the worst case of put/get?

What if?

- We had more than just one hash function
  - Use two hash functions, and place the element in the bucket that is the **least loaded**
  - **Second-Choice Hashing**

What if?

- We had more than just one hash function
  - Use two hash functions to compute two buckets, and place the element in the bucket that is the **least loaded**
  - **Second-Choice Hashing**
  - Still insufficient to get past $O(1 + \lambda)$

What if?

- We had more than just one hash function
  - Use two hash functions to compute two buckets, and place the element in the bucket that is the **least loaded**
  - **Second-Choice Hashing**
  - Still insufficient to get past $O(1 + \lambda)$

- We could move keys after they're placed
  - Still insufficient to bound the worst case lookup
  - It does however reduce variance

Robin-Hood Hashing

- Variation of open-addressing where keys can be moved after they're placed

- **Key Idea:** when a key is already present during an insertion that is closer to its "base" location than the new key, it is displaced to make room for new key
  - Decreases variance in the expected number of lookups

a	b	z	x	c	d	e			

Robin-Hood Hashing

- Variation of open-addressing where keys can be moved after they're placed

- **Key Idea:** when a key is already present during an insertion that is closer to its "base" location than the new key, it is displaced to make room for new key
  - Decreases variance in the expected number of lookups

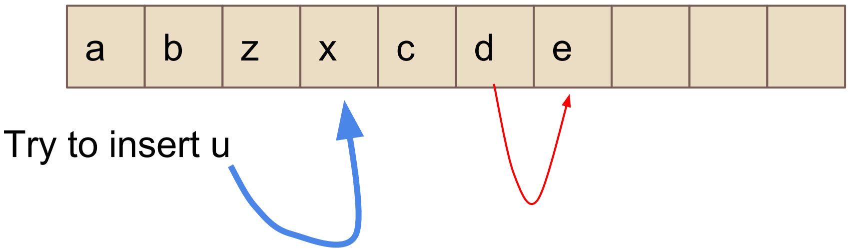
a	b	z	x	c	d	e			

Probe count for e is 1

Robin-Hood Hashing

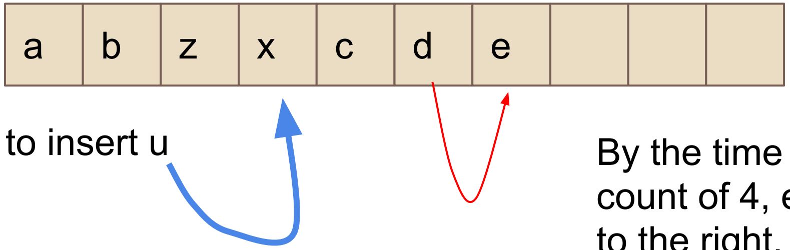
- Variation of open-addressing where keys can be moved after they're placed

- **Key Idea:** when a key is already present during an insertion that is closer to its "base" location than the new key, it is displaced to make room for new key
  - Decreases variance in the expected number of lookups



Robin-Hood Hashing

- Variation of open-addressing where keys can be moved after they're placed
- **Key Idea:** when a key is already present during an insertion that is closer to its "base" location than the new key, it is displaced to make room for new key
  - Decreases variance in the expected number of lookups



Try to insert u

By the time reach e, u has a probe count of 4, e only of 1, so displace e to the right, and insert u at e's spot

Cuckoo Hashing

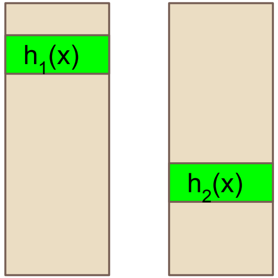
- Cuckoo hashing combines both ideas

- Hashing scheme where
  - Lookups are **worst-case O(1)**
  - Deletions are **worst-case O(1)**
  - Insertions are **expected O(1)**

(Analysis is quite complicated, we won't see it in class)

Cuckoo Hashing

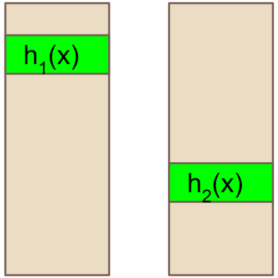
- Maintains two tables, each of which has m elements
- Choose to hash functions h_1 and h_2
- Maintains invariant:
  - every element will be either at position $h_1(x)$ in the first table or $h_2(x)$ in the second



Cuckoo Hashing

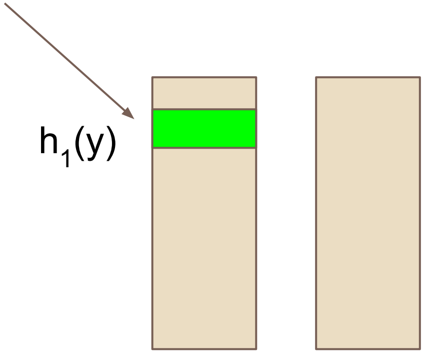
- Lookups take time O(1) because only two locations must be checked

- Deletions take time O(1) because only tw locations must be checked



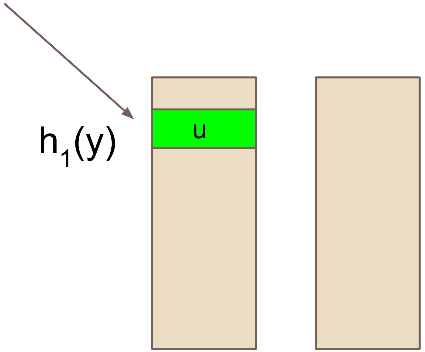
Cuckoo Hashing

- To insert an element y, first try table 1:
  - If $h_1(y)$ is empty, place y there.



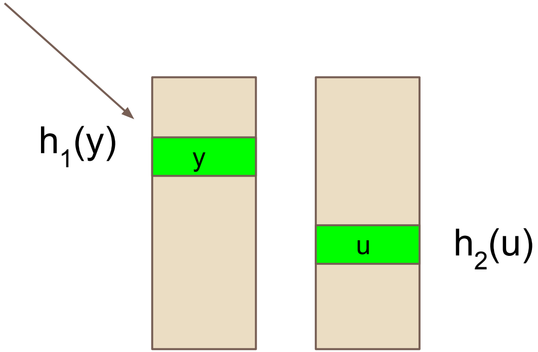
Cuckoo Hashing

- To insert an element y, first try table 1:
  - If $h_1(y)$ is empty, place y there.
  - If $h_1(y)$ contains an element **u**, place y there but then try to place y into table 2



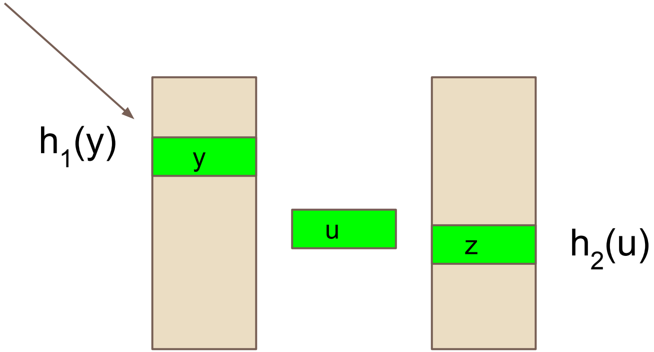
Cuckoo Hashing

- To insert an element y, first try table 1:
  - If h_1(y) is empty, place y there.
  - If h_1(y) contains an element **u**, place y there but then try to place y into table 2



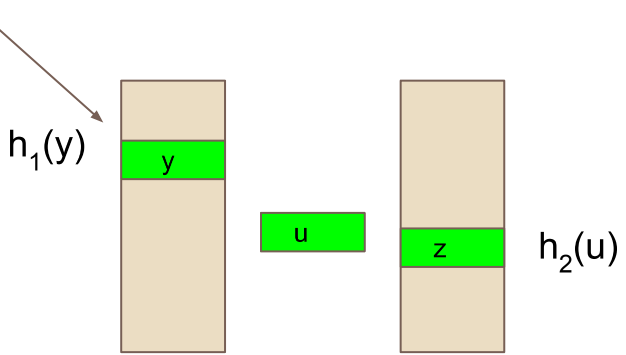
Cuckoo Hashing

- What if table 2 had an element **z** at $h_2(u)$?



Cuckoo Hashing

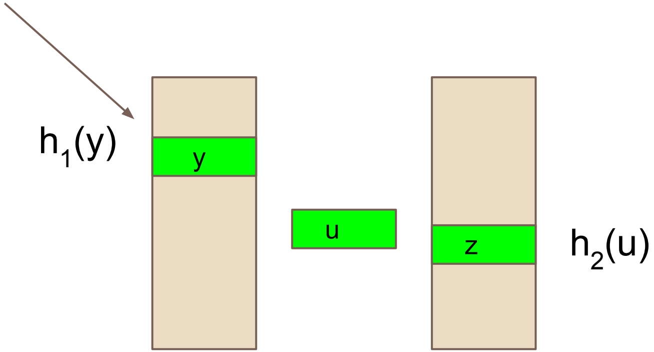
- What if table 2 had an element **z** at $h_2(u)$?
  - Then evict z, and place $h_2(z)$ in the first table



Cuckoo Hashing

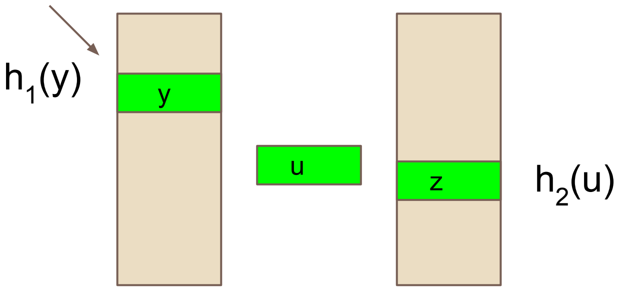
- What if table 2 had an element **z** at $h_2(u)$?
  - Then evict z, and place $h_2(z)$ in the first table

- Keep going until detect that there is a cycle



Cuckoo Hashing

- What if table 2 had an element **z** at $h_2(u)$?
  - Then evict z, and place $h_2(z)$ in the first table
- Keep going until detect that there is a cycle (revisit same sot with the same slot to insert)
  - At which point **rehash the table** choosing new hash functions h_1 and h_2



Cuckoo Hashing

- What if table 2 had an element **z** at $h_2(u)$?
  - Then evict z, and place $h_2(z)$ in the first table
- Keep going until detect that there is a cycle (revisit same sot with the same slot to insert)
  - At which point **rehash the table** choosing new hash functions h_1 and h_2

Proofs rely on bipartite graphs and strongly connected components!