CSE 250
Data Structures

Dr. Eric Mikida
epmikida@buffalo.edu
208 Capen Hall

Lec 34: Hash Table Variants

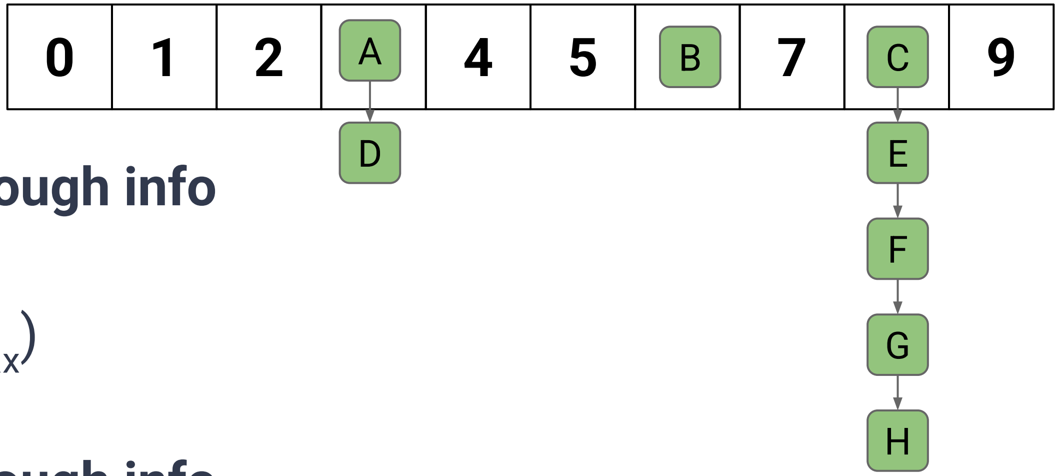
Warm-Up Question

What is the load factor (α) of this HashTable?

A: 5 B: 0.8 C: 0.5 D: Not enough info

What is the max load factor (α_{max}) of this HashTable?

A: 5 B: 0.8 C: 0.5 D: Not enough info



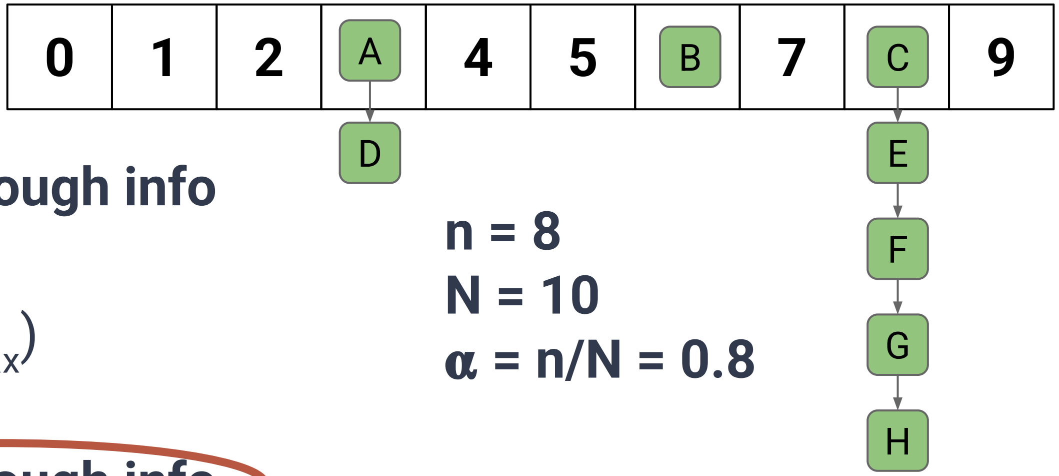
Warm-Up Question

What is the load factor (α) of this HashTable?

A: 5 B: 0.8 C: 0.5 D: Not enough info

What is the max load factor (α_{max}) of this HashTable?

A: 5 B: 0.8 C: 0.5 D: Not enough info



n = 8
N = 10
α = n/N = 0.8

Announcements

- PA3 is out now
  - Testing due this Sunday, 4/27
  - Implementation due next Sunday, 5/4
- WA5 releasing this Monday

Recap of HashTables (so far...)

Current Design: HashTable **with Chaining**

- Array of buckets
- Each bucket is the head of a linked list (a "chain" of elements)

Runtime for `contains(x)`

Expected Runtime:

Runtime for `contains(x)`

Expected Runtime:

1. Find the bucket (call our hash function): $O(c_{hash}) = O(1)$

Runtime for `contains(x)`

Expected Runtime:

1. Find the bucket (call our hash function): $\boldsymbol{O(c_{hash}) = O(1)}$
2. Find the record in the bucket: $\boldsymbol{O(\alpha \cdot c_{equality}) = O(1)}$

Runtime for `contains(x)`

Expected Runtime:

1. Find the bucket (call our hash function): $O(c_{hash}) = O(1)$
2. Find the record in the bucket: $O(\alpha \cdot c_{equality}) = O(1)$

Remember: we don't let α exceed a constant value

Runtime for `contains(x)`

Expected Runtime:

1. Find the bucket (call our hash function): $\boldsymbol{O(c_{hash}) = O(1)}$
2. Find the record in the bucket: $\boldsymbol{O(\alpha \cdot c_{equality}) = O(1)}$
3. **Total:** $\boldsymbol{O(c_{hash} + \alpha \cdot c_{equality}) = O(1)}$

Runtime for `contains(x)`

Expected Runtime:

1. Find the bucket (call our hash function): $O(c_{hash}) = O(1)$
2. Find the record in the bucket: $O(\alpha \cdot c_{equality}) = O(1)$
3. **Total:** $O(c_{hash} + \alpha \cdot c_{equality}) = O(1)$

Unqualified Worst-Case:

Runtime for `contains(x)`

Expected Runtime:

1. Find the bucket (call our hash function): $O(c_{hash}) = O(1)$
2. Find the record in the bucket: $O(\alpha \cdot c_{equality}) = O(1)$
3. **Total:** $O(c_{hash} + \alpha \cdot c_{equality}) = O(1)$

Unqualified Worst-Case:

1. Find the bucket (call our hash function): $O(c_{hash}) = O(1)$

Runtime for `contains(x)`

Expected Runtime:

1. Find the bucket (call our hash function): $O(c_{hash}) = O(1)$
2. Find the record in the bucket: $O(\alpha \cdot c_{equality}) = O(1)$
3. **Total:** $O(c_{hash} + \alpha \cdot c_{equality}) = O(1)$

Unqualified Worst-Case:

1. Find the bucket (call our hash function): $O(c_{hash}) = O(1)$
2. Find the record in the bucket: $O(n \cdot c_{equality}) = O(n)$

Runtime for `contains(x)`

<u>Expected Runtime:</u>

1. Find the bucket (call our hash function): $O(c_{hash}) = O(1)$
2. Find the record in the bucket: $O(\alpha \cdot c_{equality}) = O(1)$
3. **Total:** $O(c_{hash} + \alpha \cdot c_{equality}) = O(1)$

<u>Unqualified Worst-Case:</u>

1. Find the bucket (call our hash function): $O(c_{hash}) = O(1)$
2. Find the record in the bucket: $O(n \cdot c_{equality}) = O(n)$

Note: The expected number of equality checks and the worst-case number of equality checks are where these costs differ

Runtime for `contains(x)`

Expected Runtime:

1. Find the bucket (call our hash function): $O(c_{hash}) = O(1)$
2. Find the record in the bucket: $O(\alpha \cdot c_{equality}) = O(1)$
3. **Total:** $O(c_{hash} + \alpha \cdot c_{equality}) = O(1)$

Unqualified Worst-Case:

1. Find the bucket (call our hash function): $O(c_{hash}) = O(1)$
2. Find the record in the bucket: $O(n \cdot c_{equality}) = O(n)$
3. **Total:** $O(c_{hash} + n \cdot c_{equality}) = O(n)$

Runtime for `remove(x)`

Expected Runtime:

1. Find the bucket (call our hash function): $O(c_{hash}) = O(1)$
2. Find the record in the bucket: $O(\alpha \cdot c_{equality}) = O(1)$
3. Remove (by reference): $O(1)$
4. **Total:** $O(c_{hash} + \alpha \cdot c_{equality} + 1) = O(1)$

Unqualified Worst-Case:

1. Find the record in the bucket: $O(n \cdot c_{equality}) = O(n)$
2. **Total:** $O(c_{hash} + n \cdot c_{equality} + 1) = O(n)$

Runtime for `remove(x)`

Expected Runtime:

1. Find the bucket (call our hash function): $O(c_{hash}) = O(1)$
2. Find the record in the bucket: $O(\alpha \cdot c_{equality}) = O(1)$
3. Remove (by reference): $O(1)$
4. **Total:** $O(c_{hash} + \alpha \cdot c_{equality} + 1) = O(1)$

Only one extra constant-time step to remove

Unqualified Worst-Case:

1. Find the record in the bucket: $O(n \cdot c_{equality}) = O(n)$
2. **Total:** $O(c_{hash} + n \cdot c_{equality} + 1) = O(n)$

Runtime for `insert(x)`

Expected Runtime:

1. Find the bucket (call our hash function): $\boldsymbol{O(c_{hash}) = O(1)}$
2. Remove $\boldsymbol{x}$ from bucket if present: $\boldsymbol{O(\alpha \cdot c_{equality} + 1)}$
3. Prepend to bucket: $\boldsymbol{O(1)}$
4. Rehash if needed: $\boldsymbol{O(n \cdot c_{hash} + N)}$ **(amortized** $\boldsymbol{O(1)}$**)**
5. **Total:** $\boldsymbol{O(c_{hash} + \alpha \cdot c_{equality} + 3) = O(1)}$

Unqualified Worst-Case:

1. Remove $\boldsymbol{x}$ from bucket if present: $\boldsymbol{O(n \cdot c_{equality} + 1) = O(n)}$
2. **Total:** $\boldsymbol{O(c_{hash} + n \cdot c_{equality} + N) = O(n)}$

Runtime for `insert(x)`

Expected Runtime:

1. Find the bucket (call our hash function): $\boldsymbol{O(c_{hash}) = O(1)}$
2. Remove ***x*** from bucket if present: $\boldsymbol{O(\alpha \cdot c_{equality} + 1)}$
3. Prepend to bucket: $\boldsymbol{O(1)}$
4. Rehash if needed: $\boldsymbol{O(n \cdot c_{hash} + N)}$ **(amortized *O*(1))**
5. **Total:** $\boldsymbol{O(c_{hash} + \alpha \cdot c_{equality} + 3) = O(1)}$

One additional constant-time step to prepend, and then potentially the need to rehash, but that is amortized O(1)

Unqualified Worst-Case:

1. Remove ***x*** from bucket if present: $\boldsymbol{O(n \cdot c_{equality} + 1) = O(n)}$
2. **Total:** $\boldsymbol{O(c_{hash} + n \cdot c_{equality} + N) = O(n)}$

Quick Note on Java

- `Object::hashCode()` is a member function in Java that returns a pseudo-random integer for every object
    - When we define our own objects, we can also override this function (see `BZPair` in PA3)
- Small issue: `hashCode()` can return negative numbers
    - **Solution:** Use `Math.floorMod` instead of regular modulus

HashTable Drawbacks?

…So the expected runtime of all operations is ***O*(1)**

Why would you ever use any other data structure?

HashTable Drawbacks?

…So the expected runtime of all operations is $\boldsymbol{O(1)}$

Why would you ever use any other data structure?

- HashTables do not preserve ordering
- HashTables may waste a lot of memory
- Rehashing can be expensive
- Only **guarantee** on lookup time is that it is $\boldsymbol{O(n)}$

HashTable Drawbacks?

…So the expected runtime of all operations is ***O*(1)**

Why would you ever use any other data structure?

- HashTables do not preserve ordering
- HashTables may waste a lot of memory
- Rehashing can be expensive
- Only **guarantee** on lookup time is that it is ***O*(*n*)**

These can be partially addressed by some HashTable variations

Collision Resolution

When two records are assigned to the same bucket, it is called a **collision**

- With **chaining**, collisions are resolved by treating each bucket as a list
- May result in even more empty buckets (more wasted space)

Two more collision resolution techniques try to help with this issue

- Open Addressing
- Cuckoo Hashing

HashTables with Chaining

hash(A) = 4

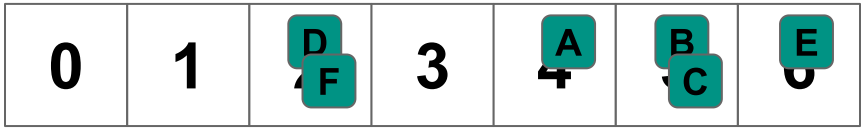
hash(B) = 5

hash(C) = 5

hash(D) = 2

hash(E) = 6

hash(F) = 2



HashTables with Chaining

hash(A) = 4

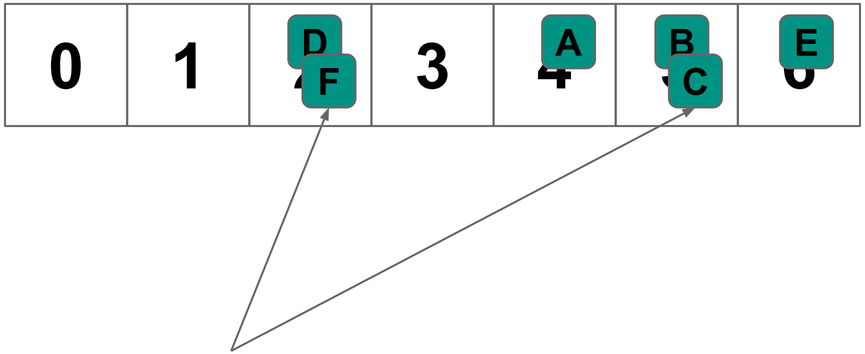
hash(B) = 5

hash(C) = 5

hash(D) = 2

hash(E) = 6

hash(F) = 2



Collisions are resolved by adding the element to the bucket's linked list

HashTables with Open Addressing

hash(A) = 4 ← no collision

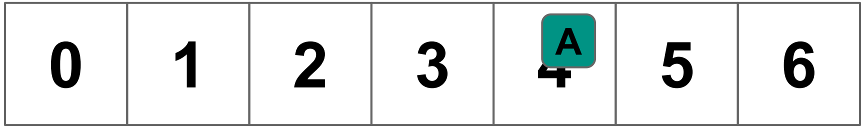
hash(B) = 5

hash(C) = 5

hash(D) = 2

hash(E) = 6

hash(F) = 4



With Open Addressing collisions are resolved by "cascading" to the next available bucket

HashTables with Open Addressing

hash(A) = 4

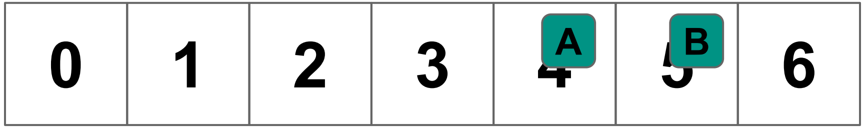
hash(B) = 5 ← no collision

hash(C) = 5

hash(D) = 2

hash(E) = 6

hash(F) = 4



With Open Addressing collisions are resolved by "cascading" to the next available bucket

HashTables with Open Addressing

hash(A) = 4

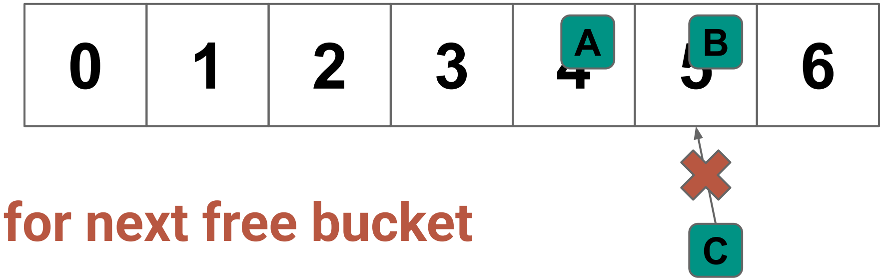
hash(B) = 5

hash(C) = 5 ← collision! Search for next free bucket

hash(D) = 2

hash(E) = 6

hash(F) = 4



With Open Addressing collisions are resolved by "cascading" to the next available bucket

HashTables with Open Addressing

hash(A) = 4

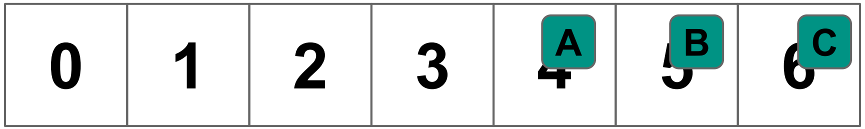
hash(B) = 5

hash(C) = 5 ← collision! Search for next free bucket

hash(D) = 2

hash(E) = 6

hash(F) = 4



With Open Addressing collisions are resolved by "cascading" to the next available bucket

HashTables with Open Addressing

hash(A) = 4

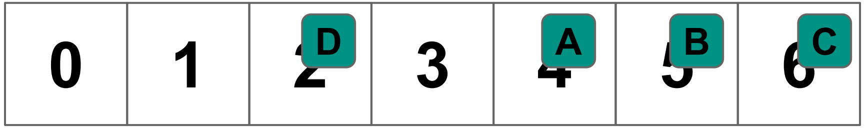
hash(B) = 5

hash(C) = 5

hash(D) = 2 ← no collision!

hash(E) = 6

hash(F) = 4



With Open Addressing collisions are resolved by "cascading" to the next available bucket

HashTables with Open Addressing

hash(A) = 4

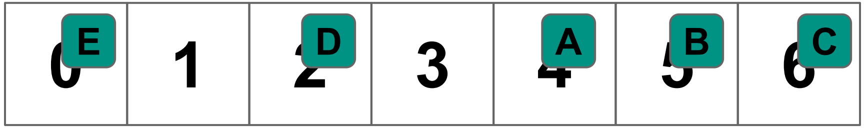
hash(B) = 5

hash(C) = 5

hash(D) = 2

hash(E) = 6 ← collision! cascade to 0

hash(F) = 4



With Open Addressing collisions are resolved by "cascading" to the next available bucket

HashTables with Open Addressing

hash(A) = 4

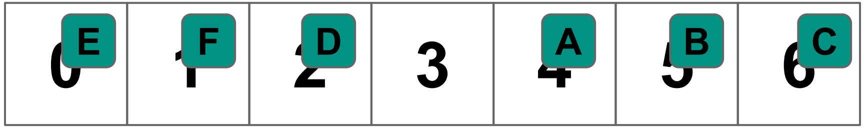
hash(B) = 5

hash(C) = 5

hash(D) = 2

hash(E) = 6

hash(F) = 4 ← collision! Cascade all the way to 1



With Open Addressing collisions are resolved by "cascading" to the next available bucket

HashTables with Open Addressing

hash(A) = 4

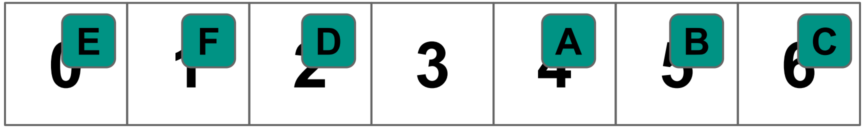
hash(B) = 5

hash(C) = 5

hash(D) = 2

hash(E) = 6

hash(F) = 4 ← collision! Cascade all the way to 1



With Open Addressing collisions are resolved by "cascading" to the next available bucket

How does lookup work?

HashTables with Open Addressing

hash(A) = 4

hash(B) = 5

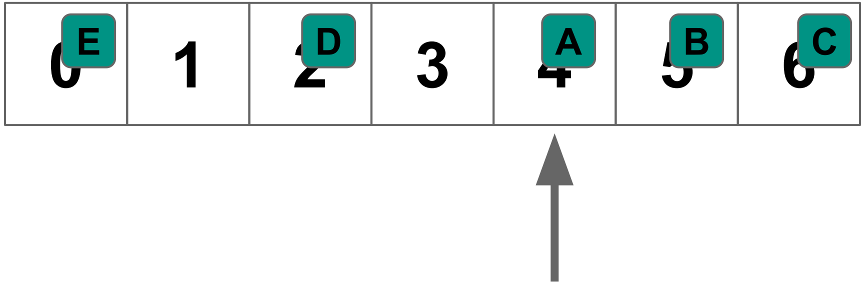
hash(C) = 5

hash(D) = 2

hash(E) = 6

hash(F) = 4

`contains(F)`



Bucket 4 does not contain F. Are we sure F does not exist?

HashTables with Open Addressing

hash(A) = 4

hash(B) = 5

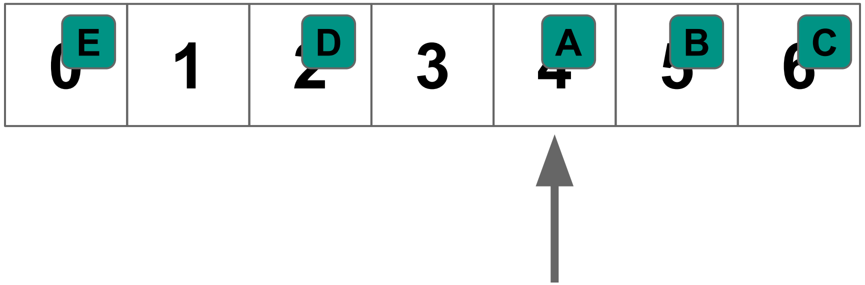
hash(C) = 5

hash(D) = 2

hash(E) = 6

hash(F) = 4

`contains(F)`



Bucket 4 does not contain F. Are we sure F does not exist? **No…it could have cascaded!**

HashTables with Open Addressing

hash(A) = 4

hash(B) = 5

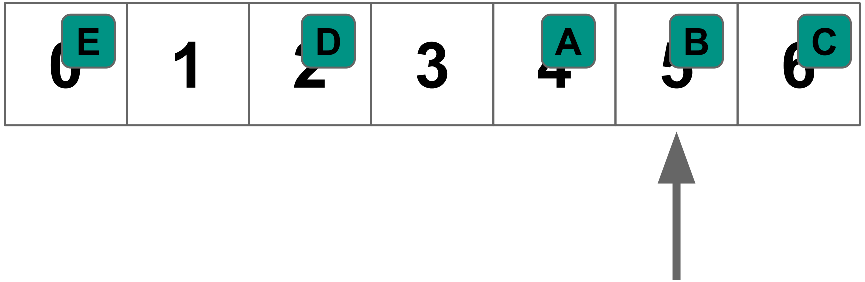
hash(C) = 5

hash(D) = 2

hash(E) = 6

hash(F) = 4

`contains(F)`



Bucket 5 does not contain F. Are we sure F does not exist? **No…it could have cascaded!**

HashTables with Open Addressing

hash(A) = 4

hash(B) = 5

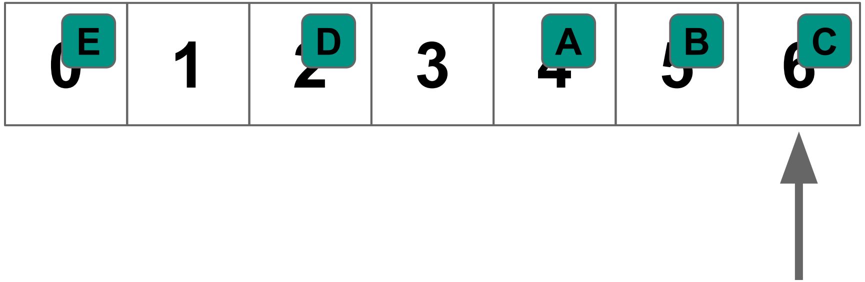
hash(C) = 5

hash(D) = 2

hash(E) = 6

hash(F) = 4

`contains(F)`



Bucket 6 does not contain F. Are we sure F does not exist? **No…it could have cascaded!**

HashTables with Open Addressing

hash(A) = 4

hash(B) = 5

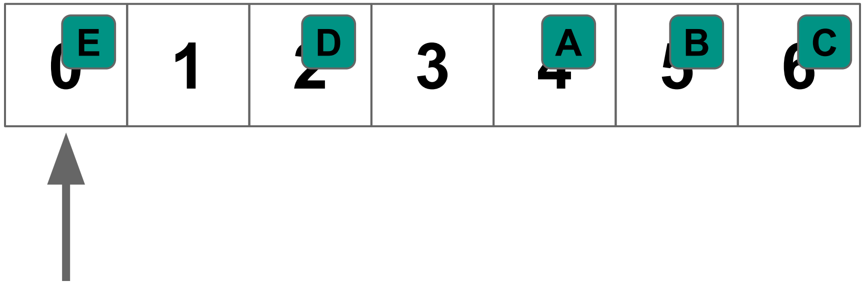
hash(C) = 5

hash(D) = 2

hash(E) = 6

hash(F) = 4

`contains(F)`



Bucket 0 does not contain F. Are we sure F does not exist? **No…it could have cascaded!**

HashTables with Open Addressing

hash(A) = 4

hash(B) = 5

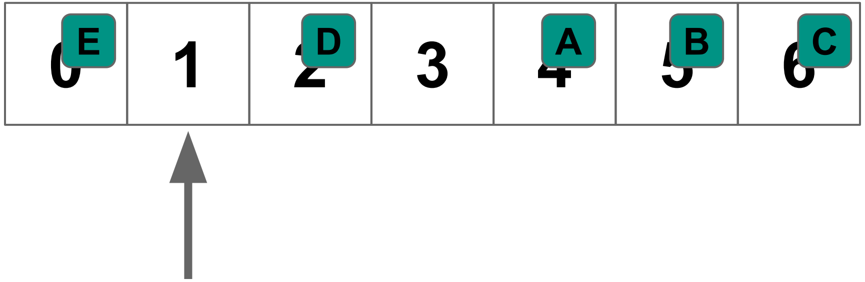
hash(C) = 5

hash(D) = 2

hash(E) = 6

hash(F) = 4

`contains(F)`



Bucket 1 does not contain F. Are we sure F does not exist? **Yes! If F existed it would be here, so `contains(F)` returns False.**

HashTables with Open Addressing

hash(A) = 4

hash(B) = 5

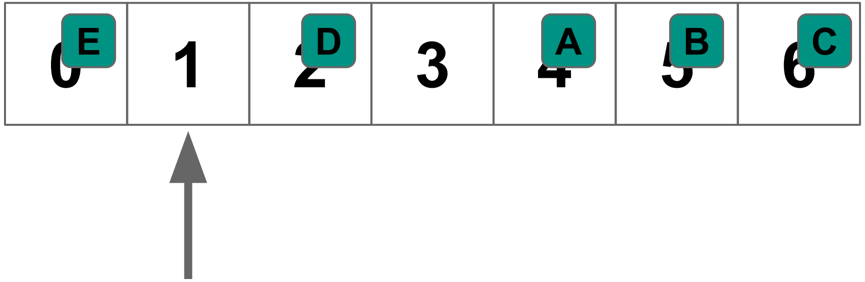
hash(C) = 5

hash(D) = 2

hash(E) = 6

hash(F) = 4

`contains(F)`



Bucket 1 does not contain F. Are we sure F does not exist? **Yes! If F existed it would be here, so `apply(F)` returns False.**

What if we insert F then remove E?

HashTables with Open Addressing

hash(A) = 4

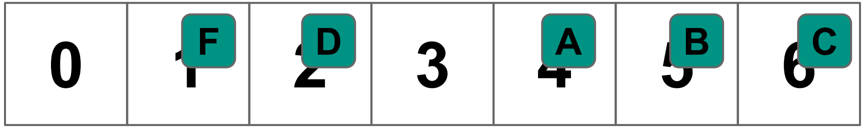
hash(B) = 5

hash(C) = 5

hash(D) = 2

hash(E) = 6

hash(F) = 4



`contains(F)` would fail in this case because it would check bucket 0 and conclude F doesn't exist!

Remove must also deal with potential cascading!

What if we insert F then remove E?

Removals with Open Addressing

To remove elements with Open Addressing:

1. First find the element (if it exists)
2. Remove the element
   a. Check all following elements in a contiguous block and move them up
   b. Don't move any element Y to a position that comes before hash(Y)

Open Addressing Runtime

Cascading to the next bucket(s) is called probing

- **Linear Probing:** If collision, cascade to hash(X) + ci
- **Quadratic Probing:** If collision, cascade to hash(X) + ci^2

Runtime Costs:

- Chaining is dominated by searching the chain
- Open Addressing is dominated by probing
  - In both cases, with low α we expect operations to be ***O*(1)**
  - Open addressing will occupy more buckets (waste less space)

Cuckoo Hashing

Open Addressing can have arbitrarily long chains

Can we reduce the chance of cascading for some operations?

Cuckoo Hashing

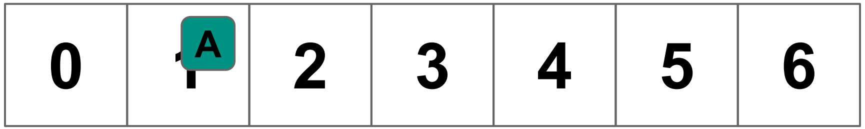
Idea: Use two hash functions, $hash_1$ and $hash_2$

To insert a record ***X***:

1. If $hash_1$(***X***) and $hash_2$(***X***) are both available, pick one at arbitrarily
2. If only one of those buckets is available, pick the available bucket
3. If neither is available, pick one arbitrarily and evict the record there
    a. Insert ***X*** in this bucket
    b. Insert the evicted record following the same procedure

HashTables with Cuckoo Hashing

$hash_1(A) = 1$	$hash_2(A) = 3$
$hash_1(B) = 2$	$hash_2(B) = 4$
$hash_1(C) = 2$	$hash_2(C) = 1$
$hash_1(D) = 4$	$hash_2(D) = 6$
$hash_1(E) = 3$	$hash_2(E) = 4$



HashTables with Cuckoo Hashing

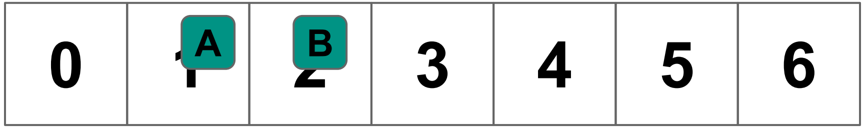
$hash_1(A) = 1$ $hash_2(A) = 3$

$hash_1(B) = 2$ $hash_2(B) = 4$

$hash_1(C) = 2$ $hash_2(C) = 1$

$hash_1(D) = 4$ $hash_2(D) = 6$

$hash_1(E) = 3$ $hash_2(E) = 4$



HashTables with Cuckoo Hashing

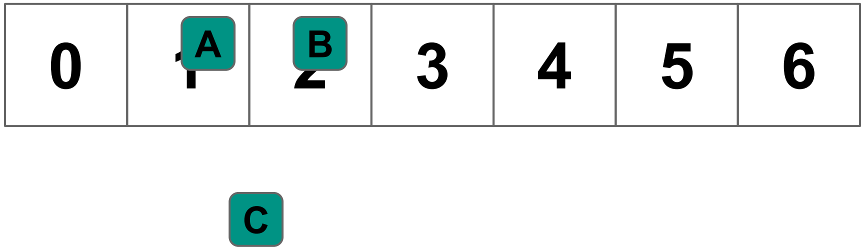
$hash_1(A) = 1$ $hash_2(A) = 3$

$hash_1(B) = 2$ $hash_2(B) = 4$

$hash_1(C) = 2$ $hash_2(C) = 1$

$hash_1(D) = 4$ $hash_2(D) = 6$

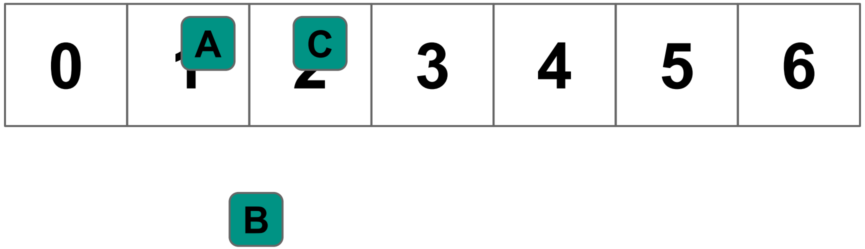
$hash_1(E) = 3$ $hash_2(E) = 4$



C can't go in either bucket, so evict one at random (let's say ***B***) and reinsert the evicted element

HashTables with Cuckoo Hashing

$hash_1(A) = 1$	$hash_2(A) = 3$
$\mathbf{hash_1(B) = 2}$	$\mathbf{hash_2(B) = 4}$
$hash_1(C) = 2$	$hash_2(C) = 1$
$hash_1(D) = 4$	$hash_2(D) = 6$
$hash_1(E) = 3$	$hash_2(E) = 4$



B can only go in 4 now, but 4 is free

HashTables with Cuckoo Hashing

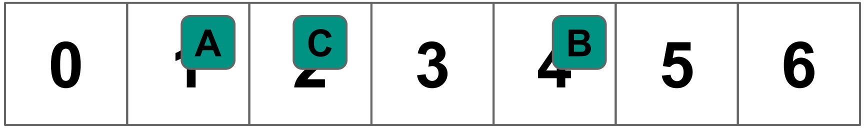
$hash_1(A) = 1$ $hash_2(A) = 3$

$hash_1(B) = 2$ $hash_2(B) = 4$

$hash_1(C) = 2$ $hash_2(C) = 1$

$hash_1(D) = 4$ $hash_2(D) = 6$

$hash_1(E) = 3$ $hash_2(E) = 4$



B can only go in 4 now, but 4 is free

HashTables with Cuckoo Hashing

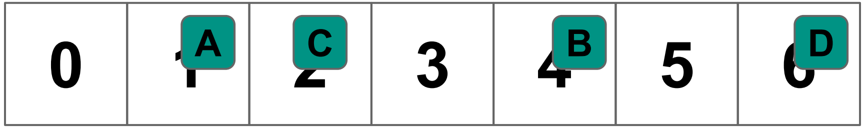
$hash_1(A) = 1$ $hash_2(A) = 3$

$hash_1(B) = 2$ $hash_2(B) = 4$

$hash_1(C) = 2$ $hash_2(C) = 1$

$hash_1(D) = 4$ **$hash_2(D) = 6$**

$hash_1(E) = 3$ $hash_2(E) = 4$



HashTables with Cuckoo Hashing

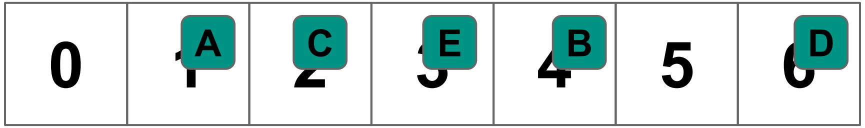
$hash_1(A) = 1$ $hash_2(A) = 3$

$hash_1(B) = 2$ $hash_2(B) = 4$

$hash_1(C) = 2$ $hash_2(C) = 1$

$hash_1(D) = 4$ $hash_2(D) = 6$

$hash_1(E) = 3$ $hash_2(E) = 4$



HashTables with Cuckoo Hashing

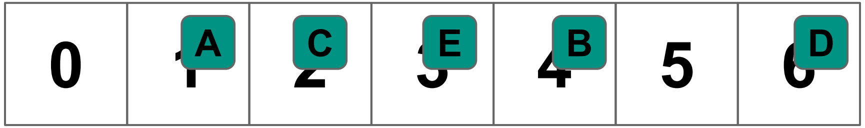
$hash_1(A) = 1$ $hash_2(A) = 3$

$hash_1(B) = 2$ $hash_2(B) = 4$

$hash_1(C) = 2$ $hash_2(C) = 1$

$hash_1(D) = 4$ $hash_2(D) = 6$

$hash_1(E) = 3$ $hash_2(E) = 4$



What if we try to insert ***F*** which hashes to either 1 or 3?

HashTables with Cuckoo Hashing

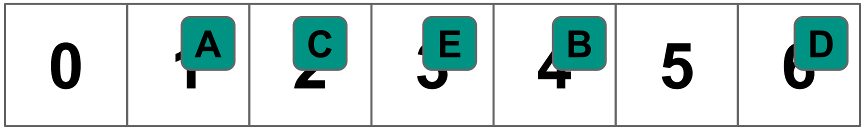
$hash_1(A) = 1$ $hash_2(A) = 3$

$hash_1(B) = 2$ $hash_2(B) = 4$

$hash_1(C) = 2$ $hash_2(C) = 1$

$hash_1(D) = 4$ $hash_2(D) = 6$

$hash_1(E) = 3$ $hash_2(E) = 4$



What if we try to insert ***F*** which hashes to either 1 or 3? **We will loop infinitely trying to evict…so limit the number of eviction attempts then do a full rehash**

Cuckoo Hashing

So with Cuckoo Hashing, we may have to rehash early, and may follow long chains of evictions inserting, but…

What is the runtime of contains/remove?

Cuckoo Hashing

So with Cuckoo Hashing, we may have to rehash early, and may follow long chains of evictions inserting, but…

What is the runtime of contains/remove?

1. Check 2 different buckets: ***O*(1)**
2. That's it…no chaining, cascading etc…

Apply and remove are <u>GUARANTEED</u> *O*(1) with Cuckoo Hashing

HashTables as Sets

We've now seen HashTable's as an implementation of Sets

- `HashSet` in Java -> Expected O(1) runtime for `add`, `contains`, `remove`

What about `HashMap`? What is a map??

HashTables as Sets

We've now seen HashTable's as an implementation of Sets

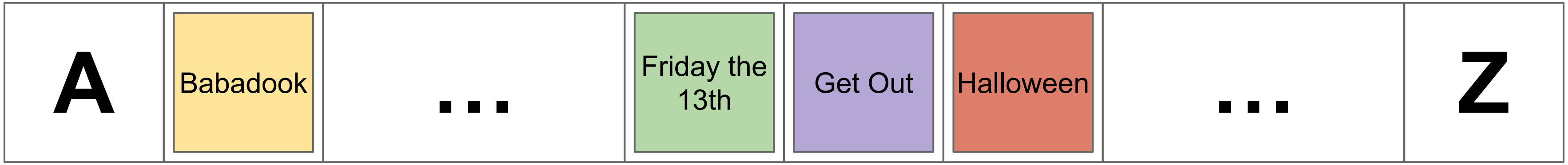
- `HashSet` in Java -> Expected O(1) runtime for `add`, `contains`, `remove`

What about `HashMap`? What is a map??

- A map IS a set. It is a set of key-value pairs!

HashSets vs HashMaps

This was an example of a **`HashSet`** that stored movie titles (with a bad hash function…but ignore that for now)



HashSets vs HashMaps

This is an example of a **`HashMap`** that stores key value pairs where the key is a movie title and the value is the movie object associated with that title