CSE 250
Data Structures

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

Lec 33: Hash Tables

Announcements

- PA3 releasing tonight

Picking a Hash Function

What function could we use that would evenly distribute values to buckets?

Picking a Hash Function

What function could we use that would evenly distribute values to buckets?

Wacky Idea: Have $h(x)$ return a random value in **[0,N)**

(This makes apply impossible…but bear with me)

Random Hash Function

$$n = \text{number of elements in any bucket}$$

$$N = \text{number of buckets}$$

$$b_{i,j} = \begin{cases} 1 & \textbf{if } \text{element } i \text{ is assigned to bucket } j \\ 0 & otherwise \end{cases}$$

Random Hash Function

$$n = \text{number of elements in any bucket}$$

$$N = \text{number of buckets}$$

$$b_{i,j} = \begin{cases} 1 & \textbf{if } \text{element } i \text{ is assigned to bucket } j \\ 0 & otherwise \end{cases}$$

$$\mathbb{E}\left[b_{i,j}\right] = \frac{1}{N}$$

Random Hash Function

$$n = \text{number of elements in any bucket}$$

$$N = \text{number of buckets}$$

$$b_{i,j} = \begin{cases} 1 & \textbf{if } \text{element } i \text{ is assigned to bucket } j \\ 0 & otherwise \end{cases}$$

$$\mathbb{E}\left[\sum_{i=0}^{n} b_{i,j}\right] = \frac{n}{N}$$

Random Hash Function

$$n = \text{number of elements in any bucket}$$

$$N = \text{number of buckets}$$

$$b_{i,j} = \begin{cases} 1 & \textbf{if} \text{ element } i \text{ is assigned to bucket } j \\ 0 & otherwise \end{cases}$$

$$\mathbb{E}\left[\sum_{i=0}^{n} b_{i,j}\right] = \frac{n}{N}$$

Only true if $b_{i,j}$ and $b_{i',j}$ are uncorrelated for any i ≠ i'

(h(i) can't be related to h(i'))

The **expected** number of elements in any bucket j

Random Hash Function

$$n = \text{number of elements in any bucket}$$

$$N = \text{number of buckets}$$

$$b_{i,j} = \begin{cases} 1 & \textbf{if } \text{element } i \text{ is assigned to bucket } j \\ 0 & otherwise \end{cases}$$

Only true if $b_{i,j}$ and $b_{i',j}$ are uncorrelated for any $i \neq i'$

(h(i) can't be related to h(i'))

$$\mathbb{E}\left[\sum_{i=0}^{n} b_{i,j}\right] = \frac{n}{N}$$

The **expected** number of elements in any bucket j

...given this information, what do the runtimes of our operations look like?

Random Hash Function

$$n = \text{number of elements in any bucket}$$

$$N = \text{number of buckets}$$

$$b_{i,j} = \begin{cases} 1 & \textbf{if } \text{element } i \text{ is assigned to bucket } j \\ 0 & otherwise \end{cases}$$

Expected runtime of `add`, `contains`, `remove`: ***O*(*n*/*N*)**

Worst-Case runtime of `add`, `contains`, `remove`: ***O*(*n*)**

Hash Functions In the Real-World

Examples

- SHA256 ← Used by GIT
- MD5, BCRYPT ← Used by unix login, apt
- MurmurHash3 ← Used by Scala

hash(*x*) is pseudo-random

- **hash(*x*)** ~ uniform random value in [0, INT_MAX)
- **hash(*x*)** always returns the same value for the same ***x***
- **hash(*x*)** is uncorrelated with **hash(*y*)** for all x ≠ y

Hash Functions In the Real-World

Examples

- SHA256 ← Used by GIT
- MD5, BCRYPT ← Used by unix login, apt
- MurmurHash3 ← Used by Scala

We then use modulus to fit this random value into the size of our hash table

hash(*x*) is pseudo-random

- **hash(*x*)** ~ uniform random value in [0, INT_MAX)
- **hash(*x*)** always returns the same value for the same ***x***
- **hash(*x*)** is uncorrelated with **hash(*y*)** for all x ≠ y

Refresher on Modulus

The modulus function takes any integers ***n*** and ***d***, and returns a number ***r*** in the range [0, ***d***), such that ***n*** = ***q* * *d* + *r***.

Refresher on Modulus

The modulus function takes any integers ***n*** and ***d***, and returns a number ***r*** in the range [0, ***d***), such that ***n*** = ***q* * *d* + *r***.

quotient divisor remainder

Refresher on Modulus

The modulus function takes any integers ***n*** and ***d***, and returns a number ***r*** in the range [0, ***d***), such that ***n*** = ***q * d + r***. (It returns the remainder of ***n / d***)

Refresher on Modulus

The modulus function takes any integers ***n*** and ***d***, and returns a number ***r*** in the range [0, ***d***), such that ***n*** = ***q * d + r***. (It returns the remainder of ***n / d***)

0	**1**	**2**	**3**	**4**	**5**	**6**

Refresher on Modulus

The modulus function takes any integers n and d, and returns a number r in the range $[0, d)$, such that $n = q * d + r$. (It returns the remainder of n / d)

0	1	2	3	4	5	6
0	1	2	3	4	5	6

Refresher on Modulus

The modulus function takes any integers ***n*** and ***d***, and returns a number ***r*** in the range [0, ***d***), such that ***n*** = ***q*** * ***d*** + ***r***. (It returns the remainder of ***n*** / ***d***)

0	1	2	3	4	5	6

0	1	2	3	4	5	6
7	8	9	10	11	12	13

Refresher on Modulus

The modulus function takes any integers ***n*** and ***d***, and returns a number ***r*** in the range [0, ***d***), such that ***n* = *q* * *d* + *r***. (It returns the remainder of ***n* / *d***)

0	1	2	3	4	5	6
0	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20

Refresher on Modulus

The modulus function takes any integers ***n*** and ***d***, and returns a number ***r*** in the range [0, ***d***), such that ***n*** = ***q * d + r***. (It returns the remainder of ***n / d***)

0	**1**	**2**	**3**	**4**	**5**	**6**

If my hash table has 7 buckets, and I insert an element with hash code 73, what bucket would it go in?

Refresher on Modulus

The modulus function takes any integers ***n*** and ***d***, and returns a number ***r*** in the range [0, ***d***), such that ***n*** = ***q * d + r***. (It returns the remainder of ***n / d***)

0	**1**	**2**	**3**	**4**	**5**	**6**

If my hash table has 7 buckets, and I insert an element with hash code 73, what bucket would it go in? **73 % 7 = 3**

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

Hash Function Recap

- We now have ***pseudo-random*** hash functions that run in ***O*(1)**

Hash Function Recap

- We now have ***pseudo-random*** hash functions that run in ***O*(1)**
  - They act as if they are uniformly random
    - Will evenly distribute elements to buckets
    - hash(***x***) is uncorrelated with hash(***y***)

Hash Function Recap

- We now have ***pseudo-random*** hash functions that run in ***O*(1)**
  - They act as if they are uniformly random
    - Will evenly distribute elements to buckets
    - hash(***x***) is uncorrelated with hash(***y***)
  - They are deterministic (hash(***x***) will always return the same value)

Hash Function Recap

- We now have ***pseudo-random*** hash functions that run in ***O*****(1)**
  - They act as if they are uniformly random
    - Will evenly distribute elements to buckets
    - hash(***x***) is uncorrelated with hash(***y***)
  - They are deterministic (hash(***x***) will always return the same value)
- We can use these hash functions to determine which bucket an arbitrary element belongs in in ***O*****(1)** time

Hash Function Recap

- We now have ***pseudo-random*** hash functions that run in ***O*****(1)**
  - They act as if they are uniformly random
    - Will evenly distribute elements to buckets
    - hash(***x***) is uncorrelated with hash(***y***)
  - They are deterministic (hash(***x***) will always return the same value)
- We can use these hash functions to determine which bucket an arbitrary element belongs in in ***O*****(1)** time
- There are expected to be ***n/N*** elements in that bucket
  - So runtime for all operations is **expected *O*(1) + *O*(*n*/*N*) = expected *O*(*n*)**

Hash Functions + Buckets

Everything is: $O\left(\frac{n}{N}\right)$

Let's call $\alpha = \frac{n}{N}$ the load factor.

Hash Functions + Buckets

Everything is: $O\left(\frac{n}{N}\right)$

Let's call $\alpha = \frac{n}{N}$ the load factor.

Idea: Make α a constant

Hash Functions + Buckets

Everything is: $O\left(\frac{n}{N}\right)$ Let's call $\alpha = \frac{n}{N}$ the load factor.

Idea: Make α a constant

Fix an α_{max} and start requiring that $\alpha \leq \alpha_{max}$

Hash Functions + Buckets

Everything is: $O\left(\frac{n}{N}\right)$

Let's call $\alpha = \frac{n}{N}$ the load factor.

Idea: Make α a constant

Fix an α_{max} and start requiring that $\alpha \leq \alpha_{max}$

What do we do when this constraint is violated?

Hash Functions + Buckets

Everything is: $O\left(\frac{n}{N}\right)$

Let's call $\alpha = \frac{n}{N}$ the load factor.

Idea: Make α a constant

Fix an α_{max} and start requiring that $\alpha \leq \alpha_{max}$

What do we do when this constraint is violated? **Resize!**

Rehashing

When we insert an element that would exceed the load factor we:

1. Resize the underlying array from N_{old} to N_{new}
2. Rehash all of the elements from their old bucket to their new bucket
   a. Element x moves from hash(x) % N_{old} to hash(x) % N_{new}

Rehashing

Let's say we have a hash table of size 6, and hash(***x***) = 65

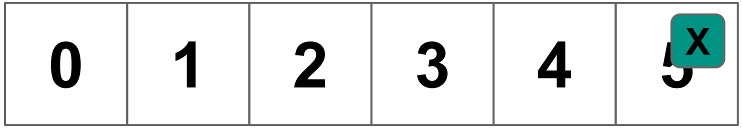
What bucket does it belong in?

0	**1**	**2**	**3**	**4**	**5**

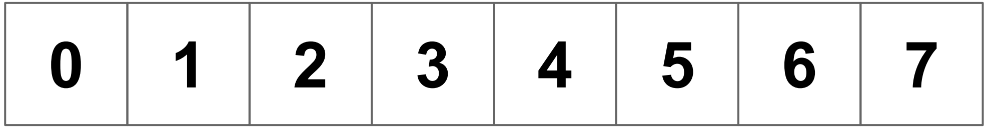
Rehashing

Let's say we have a hash table of size 6, and hash(***x***) = 65

What bucket does it belong in? 65 % 6 = 5



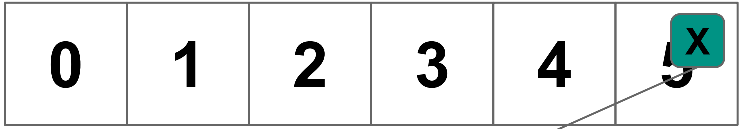
Now we want to resize the array to size 8. Where do we move ***x***?



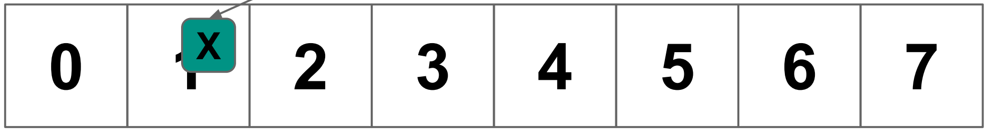
Rehashing

Let's say we have a hash table of size 6, and hash(***x***) = 65

What bucket does it belong in? 65 % 6 = 5



Now we want to resize the array to size 8. Where do we move ***x***? 65 % 8 = 1

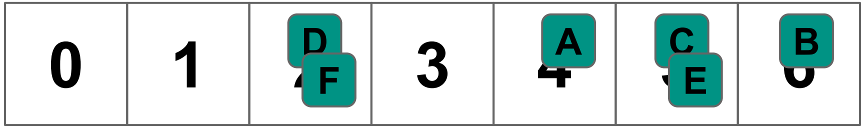


Rehashing

How long will it take to rehash every element after we resize?

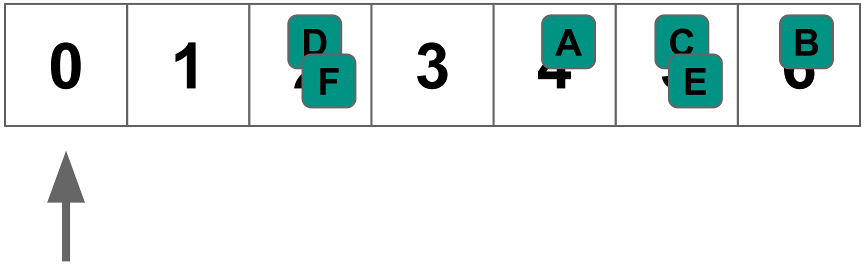
Related Question: *How do we iterate through a hash table?*

Iterating over a Hash Table



Iterating over a Hash Table

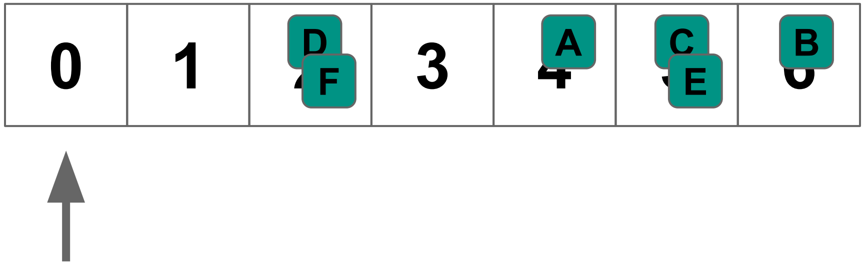
Start at the first bucket



Iterating over a Hash Table

Start at the first bucket

Iterate through that bucket

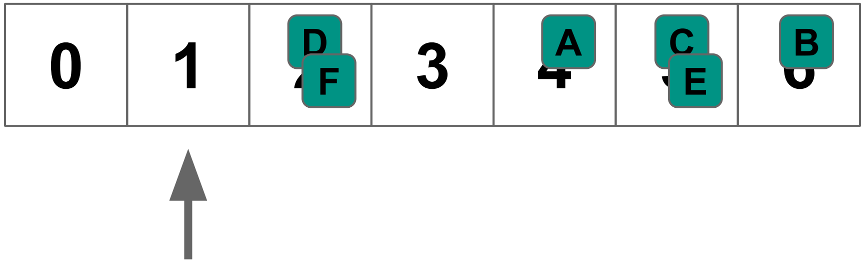


Iterating over a Hash Table

Start at the first bucket

Iterate through that bucket

Move to the next bucket



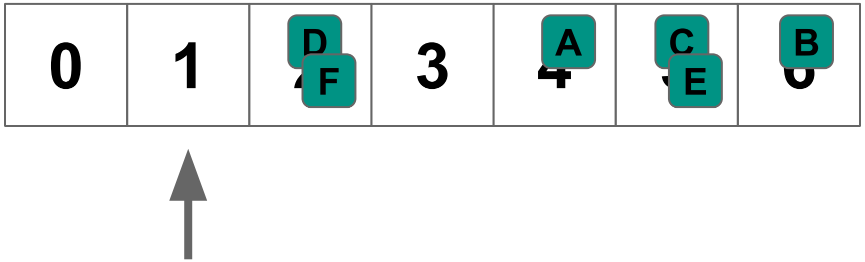
Iterating over a Hash Table

Start at the first bucket

Iterate through that bucket

Move to the next bucket

…and repeat



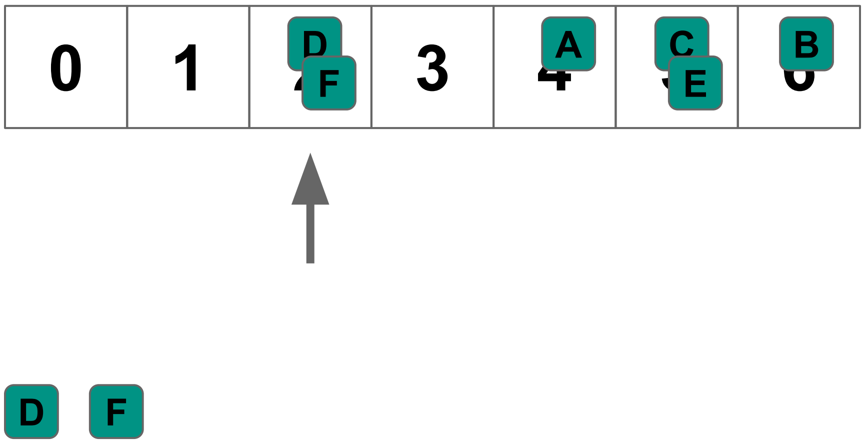
Iterating over a Hash Table

Start at the first bucket

Iterate through that bucket

Move to the next bucket

...and repeat



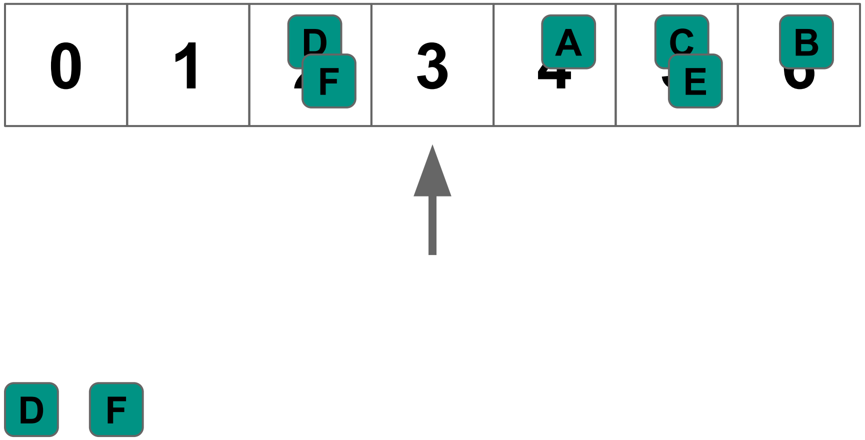
Iterating over a Hash Table

Start at the first bucket

Iterate through that bucket

Move to the next bucket

…and repeat



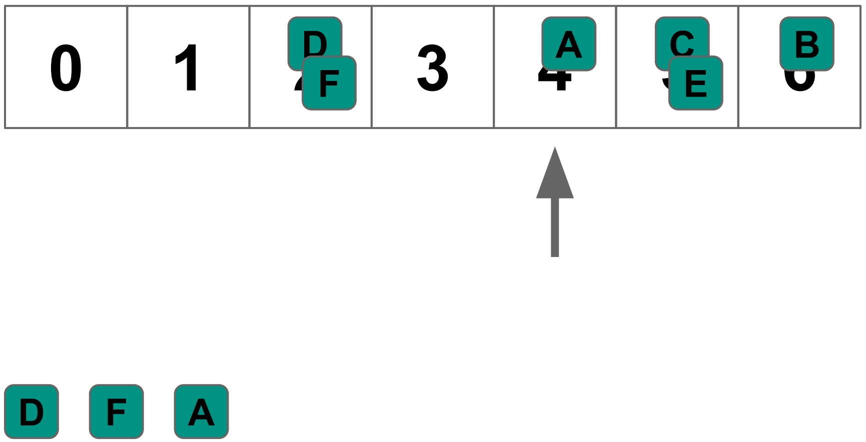
Iterating over a Hash Table

Start at the first bucket

Iterate through that bucket

Move to the next bucket

…and repeat



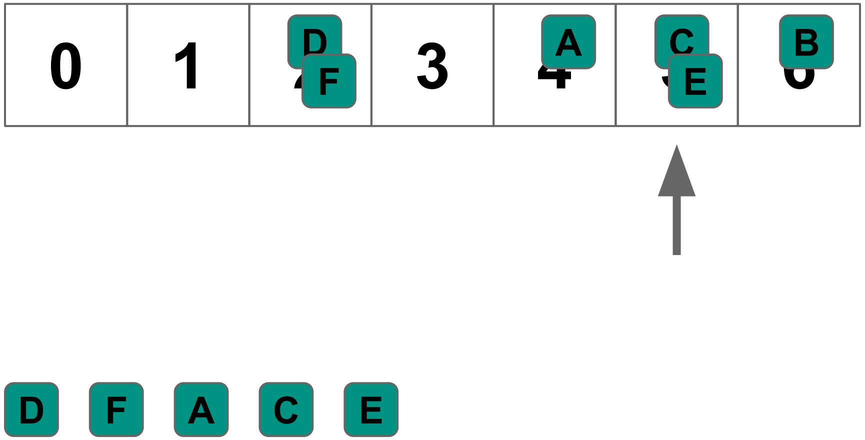
Iterating over a Hash Table

Start at the first bucket

Iterate through that bucket

Move to the next bucket

…and repeat



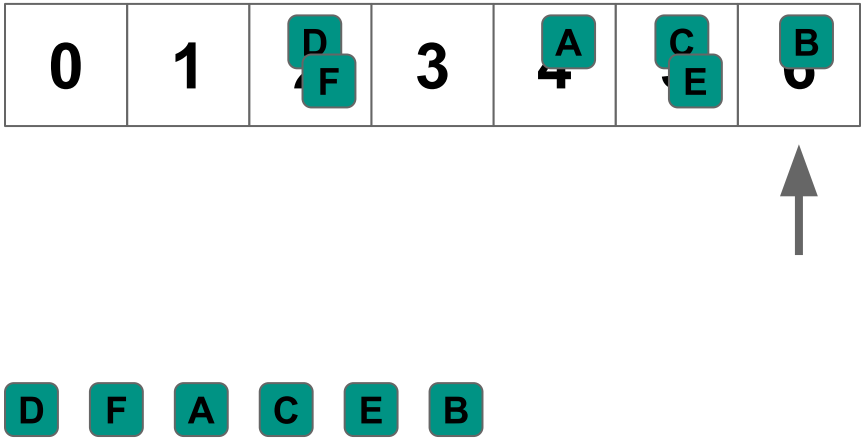
Iterating over a Hash Table

Start at the first bucket

Iterate through that bucket

Move to the next bucket

…and repeat



Iterating over a Hash Table

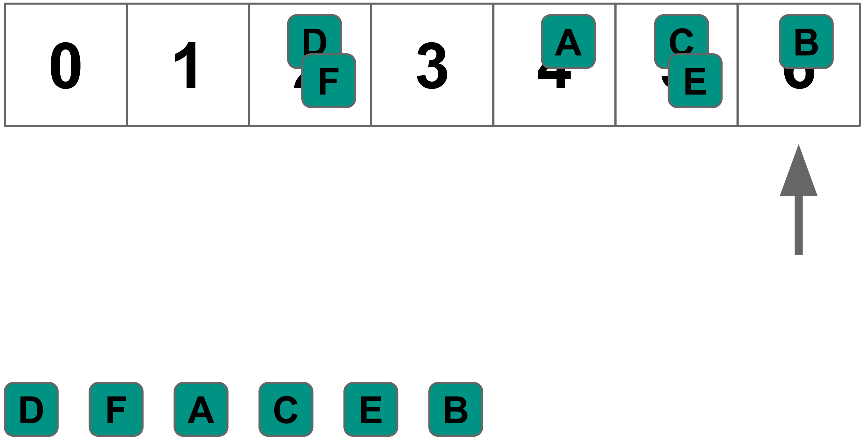
Start at the first bucket

Iterate through that bucket

Move to the next bucket

…and repeat

How long does it take?



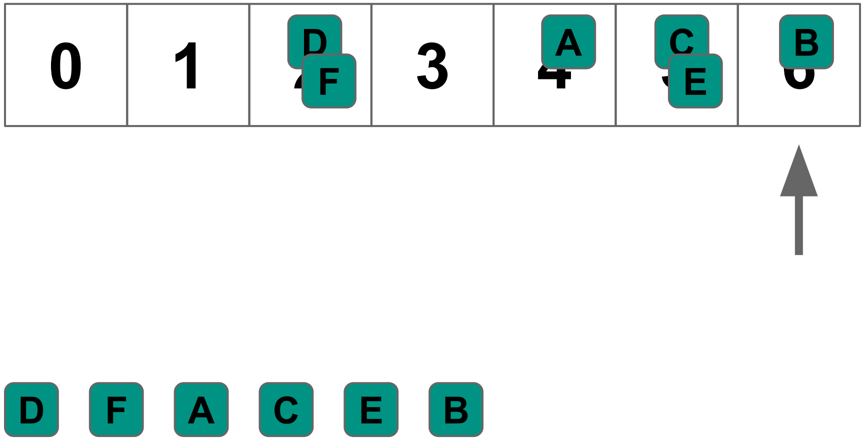
Iterating over a Hash Table

Start at the first bucket

Iterate through that bucket

Move to the next bucket

…and repeat



How long does it take? ***O*(*N* + *n*)**

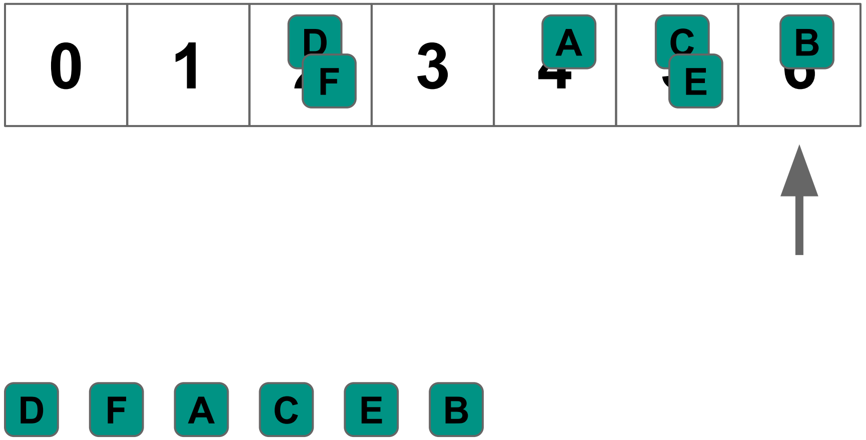
Iterating over a Hash Table

Start at the first bucket

Iterate through that bucket

Move to the next bucket

…and repeat



How long does it take? ***O*(*N* + *n*)**

Visit every bucket

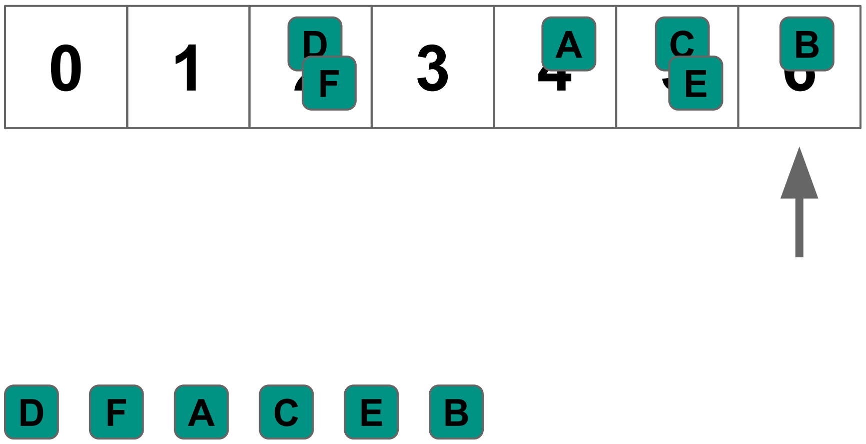
Iterating over a Hash Table

Start at the first bucket

Iterate through that bucket

Move to the next bucket

…and repeat



How long does it take? ***O*(*N* + *n*)**

Visit every bucket

Visit every element in each bucket

Rehashing

So how long does it take to rehash an entire hash table with ***n*** elements and ***N*** buckets?

Rehashing

So how long does it take to rehash an entire hash table with ***n*** elements and ***N*** buckets?

Rehashing an individual element costs ***O*(1)**

Rehashing

So how long does it take to rehash an entire hash table with ***n*** elements and ***N*** buckets?

Rehashing an individual element costs $\boldsymbol{O(1)}$

Iterating through each element costs $\boldsymbol{O(N + n)}$

Rehashing

So how long does it take to rehash an entire hash table with ***n*** elements and ***N*** buckets?

Rehashing an individual element costs $\boldsymbol{O(1)}$

Iterating through each element costs $\boldsymbol{O(N + n)}$

Rehashing costs: $\boldsymbol{O(N + n)}$

Rehashing

When we insert an element that would exceed the load factor we:

1. Resize the underlying array from N_{old} to N_{new}
2. Rehash all of the elements from their old bucket to their new bucket
   a. Element ***x*** moves from hash(***x***) % N_{old} to hash(***x***) % N_{new}

Rehashing

When we insert an element that would exceed the load factor we:

1. Resize the underlying array from N_{old} to N_{new}
2. Rehash all of the elements from their old bucket to their new bucket
   a. Element ***x*** moves from hash(***x***) % N_{old} to hash(***x***) % N_{new}

How long does this take?

1. Allocate the new array: $O(1)$
2. Rehash every element from the old array to the new: $O(N_{old} + n)$
3. Free the old array: $O(1)$

Total: $O(N_{old} + n)$

Rehashing

When we insert an element that would exceed the load factor we:

1. Resize the underlying array from N_{old} to N_{new}
2. Rehash all of the elements from their old bucket to their new bucket
   a. Element x moves from hash(x) % N_{old} to hash(x) % N_{new}

How long does this take?

How do we pick N_{new}?

1. Allocate the new array: $O(1)$
2. Rehash every element from the old array to the new: $O(N_{old} + n)$
3. Free the old array: $O(1)$

Total: $O(N_{old} + n)$

Rehashing

Whenever $\alpha > \alpha_{max}$, double the size of the array (remember ArrayLists)

If we start with ***N*** buckets and insert ***n*** elements:

1. First rehash happens at $n_1 = \alpha_{max} \times N$: goes from ***N*** to **2*N***

Rehashing

Whenever $\alpha > \alpha_{max}$, double the size of the array (remember ArrayLists)

If we start with ***N*** buckets and insert ***n*** elements:

1. First rehash happens at $n_1 = \alpha_{max} \times N$: goes from ***N*** to **2*N***
2. Second rehash happens at $n_2 = \alpha_{max} \times 2N$: goes from **2*N*** to **4*N***

Rehashing

Whenever $\alpha > \alpha_{max}$, double the size of the array (remember ArrayLists)

If we start with ***N*** buckets and insert ***n*** elements:

1. First rehash happens at $n_1 = \alpha_{max} \times N$: goes from ***N*** to **2*N***
2. Second rehash happens at $n_2 = \alpha_{max} \times 2N$: goes from **2*N*** to **4*N***
3. Third rehash happens at $n_3 = \alpha_{max} \times 4N$: goes from **4*N*** to **8*N***

Rehashing

Whenever $\alpha > \alpha_{max}$, double the size of the array (remember ArrayLists)

If we start with ***N*** buckets and insert ***n*** elements:

1. First rehash happens at $n_1 = \alpha_{max} \times N$: goes from N to $2N$
2. Second rehash happens at $n_2 = \alpha_{max} \times 2N$: goes from $2N$ to $4N$
3. Third rehash happens at $n_3 = \alpha_{max} \times 4N$: goes from $4N$ to $8N$

   …

j. jth rehash happens at $n_j = \alpha_{max} \times 2^{j-1}N$: goes from $2^{j-1}N$ to 2^jN

Total Work

With ***n*** insertions, choose ***j*** s.t. $n = 2^j\alpha_{max}$

$$2^j = n / \alpha_{max}$$

$$j = \log(n / \alpha_{max})$$

$$j = \log(n) - \log(\alpha_{max})$$

$$j \le \log(n)$$ ← **Number of rehashes**

Total Work

Rehashes required: **≤ log(n)**

The ith rehash: ***O*(2^{i}N)**

$$\sum_{i=0}^{\log(n)} O(2^i N) = O\left(N \sum_{i=0}^{\log(n)} 2^i\right) = O(2^{\log(n)+1} - 1) = O(n)$$

So ***O*(*n*)** work is required to do ***n*** insertions → Insert cost is **amortized *O*(1)**

Runtime for `contains(x)`

Expected Runtime:

Runtime for `contains(x)`

<u>Expected Runtime:</u>

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

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

Unqualified Worst-Case:

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:** $\boldsymbol{O(c_{hash} + \alpha \cdot c_{equality}) = O(1)}$

Unqualified Worst-Case:

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

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)`

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: $O(n \cdot c_{equality}) = O(n)$

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)`

<u>Expected Runtime:</u>

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

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

1. Remove x from bucket if present: $O(n \cdot c_{equality} + 1) = O(n)$
2. **Total:** $O(c_{hash} + n \cdot c_{equality} + 3) = O(n)$

Runtime for `insert(x)`

Expected Runtime:

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

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: $O(n \cdot c_{equality} + 1) = O(n)$
2. **Total:** $O(c_{hash} + n \cdot c_{equality} + 3) = O(n)$