CS 1332R
WEEK 7

Map ADT

HashMaps

Collision Handling Methods

ANNOUNCEMENTS

- ❏

Map ADT

- Stores **key-value** pairs
- Keys are…
  - **unique**
  - **immutable** - you cannot change the key in a key-value pair once it has been added.

What are examples of valid key-value pairs?

phone number - person, GTID - student, SSN - citizen

All that matters is that the key is unique to the value.

Map ADT

```
Map<K, V>

V put (K key, V value)

V remove(K key)

Set<K> keySet()

List<V> values()
```

HashMap

- Most common implementation of the Map ADT
- Backed by an array
- Insert, access, and remove key-value pairs in ***O(1) time***
- Hash functions use some function to compute an integer value (the hashcode) for each **key** in a key-value pair. This value corresponds to the index in the backing array where the key-value pair will be placed.

HashMap: Computing the Hash

The hashcode is an integer representation of an object.

- In Java, every Object has a built-in `.hashCode()` function.
- Hash function should be efficient to compute to maintain O(1) operations.
- Multiple keys can have the same hash code

`If key1 == key2, then key1.hashCode() == key2.hashCode().`

Is the reverse true?

NO

A hash function aims to provide unique values to different objects, but it is not a *requirement*.

HashMap: Compression Function

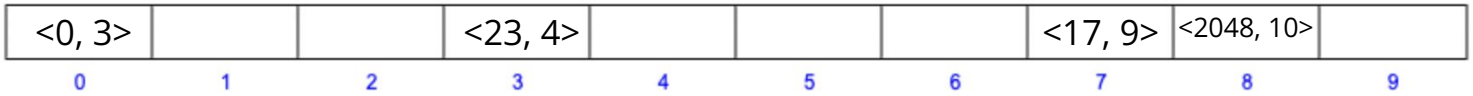
The hash corresponds to the index in the backing array where the key-value pair will be placed, but this value may be out of bounds.

COMPRESSION FUNCTION

```
index = Math.abs(key.hashCode() % backingArray.length)
```

Imagine we have the following ‹K, V› pairs and key.hashCode() = key. Where do each of the pairs end up in the backing array below?

<0, 3>, <23, 4>, <17, 9>, <2048, 10>

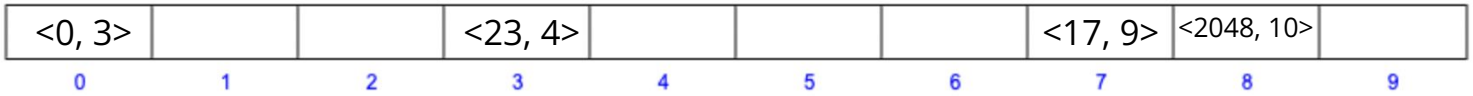


HashMap: Collisions

```
index = Math.abs(key.hashCode() % backingArray.length)
```

Imagine we have the following ‹K, V› pairs and key.hashCode() = key.

<0, 3>, <23, 4>, <17, 9>, <2048, 10>



What happens if we try to add ‹10, 9›?

COLLISION!

HashMap: Collision Handling Strategies

EXTERNAL CHAINING	**LINEAR PROBING**	**QUADRATIC PROBING**
closed addressing	open addressing	open addressing
The backing array has an **external data structure (e.g. list) at each index** holding all keys that map to that index.	Add **# of probes** to the original hash index until an available spot is reached.	Add **# of probes)^2** to the original hash index until an available spot is reached.

External Chaining HashMap: Add

PROCEDURE

1. Check for resize.

```
(size + 1.0) / length > MAX_LOAD_FACTOR
```

2. Compute the index.
3. If `backingArray[index] != null`, handle the collision →

EXTERNAL CHAINING

Iterate through the backing structure starting at `index` to check for duplicates.

2 CASES:

1. If no duplicate key is found, add the new key-value pair to the backing structure (at front or back) and increment `size`.
2. If a duplicate key is found, what do we do?
   replace the old value associated with that key with the new value

External Chaining HashMap: Remove

PROCEDURE

1. Compute the index.
2. If `backingArray[index].key != key` →

EXTERNAL CHAINING
(hard removal)

1. Iterate through the backing structure starting at `index` to find the key.
2. If found, remove the key from the backing structure, return the associated value, decrement `size`.

How do we know the key is not in the map?

1. A null index is reached.
2. The data structure does not contain the key.

HashMap: Resize

PROCEDURE

1. Create a new array of 2 * n + 1 backing length.
2. Going from 0 to backingArray.length, ***rehash*** all of the entries into the new array:

```
newIndex = Math.abs(key.hashCode() % newArray.length)
```

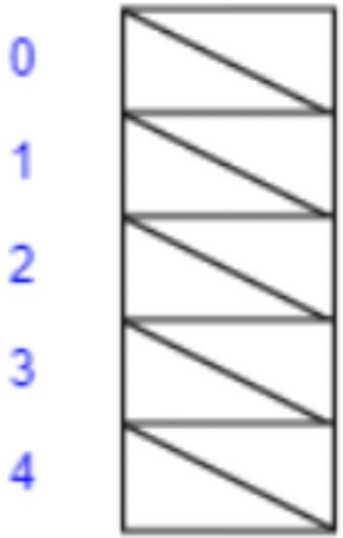
3. Set the backing array to the new array.

NOTES:

- Probing: We do not need to add entries to the new array that have been removed.
- We do not have to deal with duplicates because we have handled that when adding.
- **EFFICIENCY*:*** Stop iterating when "size" entries have been rehashed.

HashMap: Practice

QUESTION #1: Assume a maximum load factor of 0.5 and external chaining.



put(2, D)
put(13, F)
put(0, A)
put(90, T)
put(13, C)
remove(2)
put(2, D)

Linear Probing HashMap: Add

PROCEDURE

1. Check for resize.

```
(size + 1.0) / length > MAX_LOAD_FACTOR
```

2. Compute the index.
3. If `backingArray[index] != null`, handle the collision →

MapEntry objects have an attribute marking whether they are removed or not.

LINEAR PROBING

Add **# of probes** to the original hash index until an available spot is reached.

Probe through the array starting at `index` to check for duplicates, keeping track of the first index where a pair was removed.
Stop iterating when 1) the key is found, 2) a null spot is found, or 3) `size` non-removed keys have been seen.

2 Cases:

1. If not found or **found and removed**, add key-value pair at the 1) first removed index, or 2) the first null index and increment `size`.
2. If **found and not removed**, replace the old value with the new value.

Quadratic Probing HashMap: Add

PROCEDURE

1. Check for resize.

```
(size + 1.0) / length > MAX_LOAD_FACTOR
```

2. Compute the index.
3. If `backingArray[index] != null`, handle the collision →

MapEntry objects have an attribute marking whether they are removed or not.

QUADRATIC PROBING

*** Add **# of probes squared** to the original hash index until an available spot is reached.***

Probe through the array starting at `index` to check for duplicates, keeping track of the first index where a pair was removed.
Stop iterating when 1) the key is found, 2) a null spot is found, or 3) you have probed `backingArray.length` times (resize, then re-probe).

2 Cases:

1. If not found or **found and removed**, add key-value pair at the 1) first removed index, or 2) the first null index and increment `size`.
2. If **found and not removed**, replace the old value with the new value.

Linear/Quadratic Probing HashMap: Remove

PROCEDURE

1. Compute the index.
2. If `backingArray[index].key != key` →

PROBING
(soft removal)

Iterate through the backing structure at `index` to find the key (following appropriate probing rule).

If **found and not removed**, mark the entry as removed, return the associated value and decrement `size`.

How do we know the key is not in the map?

1. The key is in the map and has been removed.
2. A null index is reached.
3. Probe limit is reached.

HashMap: Practice

QUESTION #2: Assume a maximum load factor of 0.67 and linear probing.

put(124, A)
put(17, B)
put(19, C)
put(55, D)
put(17, E)
put(39, F)
put(28, G)
remove(39)
contains(28)
put(72, H)
remove(17)
remove(39)
put(1, I)
put(2, J)

HashMap: Practice

QUESTION #2: Assume a maximum load factor of 0.67 and ***quadratic*** probing.

put(124, A)
put(17, B)
put(19, C)
put(55, D)
put(17, E)

csvistool.com

HashMap: Differences in Probing Strategies

LINEAR PROBING

- Susceptible to clustering of data
  - If you have a bad hash function, map effectively becomes a list.

QUADRATIC PROBING

- Spreads the data out a bit more
- Prevents clustering of data
- Prone to infinite probing, requires a probe limit

HashMap: Efficiencies

	Best/Average Case†	Worst Case††
Adding	***O(1)***	***O(n)***
Removing	***O(1)***	***O(n)***
Searching	***O(1)***	***O(n)***

† Note: best vs. worst case depends strongly on the hash function; bad hash functions yield poor performance (due to poorly spread data)

†† For probing, worst case add is $O(n^2)$ due to resizing; this does not apply to external chaining because you can just add to the front

HashMap: Practice

Give the big-O's of the following operations. Write the amortized complexity where appropriate and state that it is amortized. Try to provide a brief explanation for each big-O.

1. Runtime of adding to a HashMap with external chaining and a good hash function: O(1)
2. Runtime of adding to a HashMap with linear probing and a hash function that always returns 3: O(n)
3. Runtime of adding to a HashMap with quadratic probing and a hash function that always returns 3: O(n)
4. Average case of adding to a HashMap with external chaining backed by BSTs instead of LinkedLists: O(1)
5. Worst case of adding to a HashMap with external chaining backed by BSTs instead of LinkedLists: O(n)

LEETCODE PROBLEMS

15. 3sum

3. Longest Substring Without Repeating Characters

Any questions?

Name
Office Hours
Contact

Name
Office Hours
Contact

Let us know if there is anything specific you want out of recitation!