Set and Map ADTs: Hash Tables

CSE 373 Winter 2020

Instructor: Hannah C. Tang

Teaching Assistants:

Aaron Johnston	Ethan Knutson	Nathan Lipiarski
Amanda Park	Farrell Fileas	Sam Long
Anish Velagapudi	Howard Xiao	Yifan Bai
Brian Chan	Jade Watkins	Yuma Tou
Elena Spasova	Lea Quan	

Poll Everywhere

pollev.com/uwcse373

- How long did HW4 take?

A. 0-2 Hours
B. 2-4 Hours
C. 4-6 Hours
D. 6-10 Hours
E. 10-14 Hours
F. 14+ Hours
G. I haven't finished yet / I don't want to say

Announcements

- Homework 5: k-d trees is released and due *next Friday*
  - This is the first of our “hard” homeworks
  - Suggestion: pretend it’s due Tuesday so you don’t panic while prepping for midterm. Start early!
  - Hint: start with a version that doesn’t prune; then implement a version that chooses good/bad sides; then finally a pruning version

- Midterm is *also* next Friday
  - If your student number ends in an odd number, go to KNE 2**1**0
  - If your student ends in an even number, go to KNE 2**2**0

Feedback from the Reading Quiz

- Is it possible to hash data without prior knowledge of its structure? To come up with a good hash function, it seems like we would need to know appropriate features of the data ahead of time to use as inputs to the hash function.

- How do we deal with collisions?

- When will we get to hash tables?

Lecture Outline

- **Hash Tables Introduction**

- Handling Collisions
  - Separate Chaining
  - Open Addressing

- Java-specific Gotchas

Review: Set and Map Data Structures

- We've seen several data structures implementing the Set or Map ADT

- Search Trees give good performance – log N – as long as the tree is reasonably balanced
  - Which doesn't occur with sorted or mostly-sorted input
  - So we invented two new categories of search trees whose heights are bounded:
    - **B-Trees**, which grow from the root and have L >= 2 children
    - **Balanced BSTs**, which grow from the leaves but rotate to stay balanced

	Find	Add	Remove
LLRB Tree Map	h = Θ(log N)	h = Θ(log N)	h = Θ(log N)
B-Tree Map	h = Θ(log N)	h = Θ(log N)	h = Θ(log N)
BST Map	h = Θ(N)	h = Θ(N)	h = Θ(N)
LinkedList Map	Θ(N)	Θ(N)	Θ(N)

Limits of Search-Tree-Based Sets and Maps

- We required items to be comparable
  - "Is X < Y?" isn't true of all types
  - Can we avoid the comparable requirement?

- Balanced search trees have excellent performance, but can we do *even better*?
  - Θ(log N) is *amazing:* 1 billion items is still only height ~30
  - Can we get even better performance than Θ(log N)?

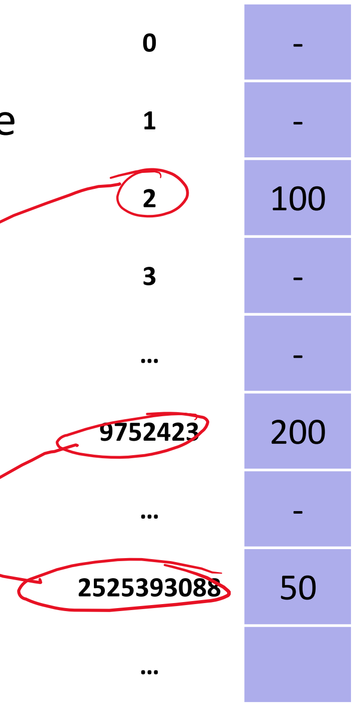
Basically: Can we do better than search trees?

Yes, We Can!

- Thanks to hashing, we can convert objects to large integers
- Thanks to DataIndexed{Integer, Word}Set, we can use these large integers as array indices

```
WordToPriorityMap m;
m.add("cat", 100);
m.add("bee", 50);
m.add("dog", 200);
```

```
hashFunction("cat") == 2;
hashFunction("bee") == 2525393088;
hashFunction("dog") == 9752423;
```



Yes, We Can! (this time for sure)

- We're mapping strings to an integer
  - Hash the strings and use the hash value as an array index
  - To force our numbers to fit into a reasonably-sized array, we'll use the modulo operator (%)

```
WordToPriorityMap m;
m.add("cat", 100);
m.add("bee", 50);
m.add("dog", 200);
```

```
hashFunction("cat") == 2;
2 % 5 == 2
hashFunction("bee") == 2525393088;
2525393088 % 5 == 3
hashFunction("dog") == 9752423;
9752423 % 5 == 3
```

0	-
1	-
2	100
3	50
4	-

Poll Everywhere

pollev.com/uwcse373

How should we handle the “bee” and “dog” collision at index 3?

A. Somehow force “bee” and “dog” to share the same index
B. Overwrite “bee” with “dog”
C. Keep “bee” and ignore “dog”
D. Put “dog” in a different index, and somehow remember/find it later
E. Rebuild the hash table with a different size and/or hash function
F. I’m not sure …

Lecture Outline

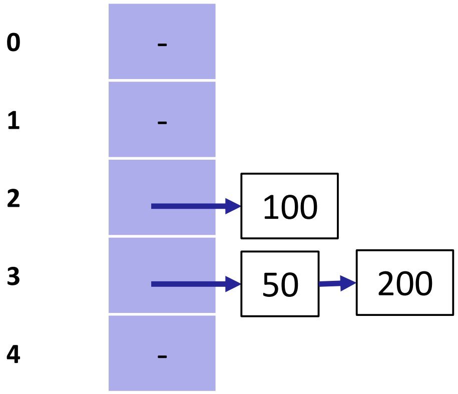
- Hash Tables Introduction
- **Handling Collisions**
  - **Separate Chaining**
  - Open Addressing
- Java-specific Gotchas

Yes, We Can! (third time's the charm)

- We're mapping strings to an integer
  - Hash the strings and use the hash value as an array index
  - To force our numbers to fit into a reasonably-sized array, we'll use the modulo operator (%)
  - Each entry in the array is an initially-empty linked list

```
WordToPriorityMap m;
m.add("cat", 100);
m.add("bee", 50);
m.add("dog", 200);
```

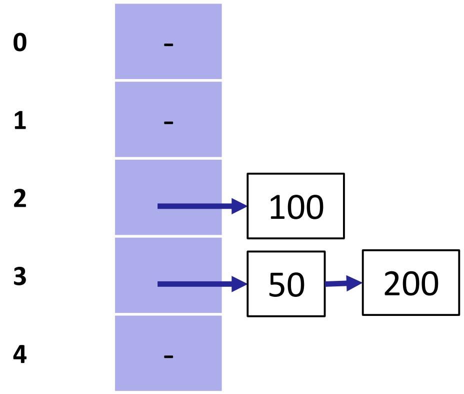
```
hashFunction("cat") == 2;
2 % 5 == 2
hashFunction("bee") == 2525393088;
2525393088 % 5 == 3
hashFunction("dog") == 9752423;
9752423 % 5 == 3
```



Separate Chaining

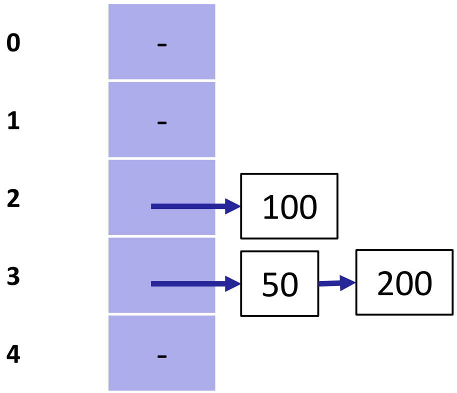
- Each index in our array is a "bucket". When an item x hashes to h:
  - If bucket h is empty: create a new list containing x
  - If bucket h is already a list: add x if it is not already present

- Bucket h is a "separate chain" of all items with hash code h



Separate Chaining: Performance

- The worst-case runtime is determined by the length of the longest list
  - Let's call the length of this worst-case list "Q"



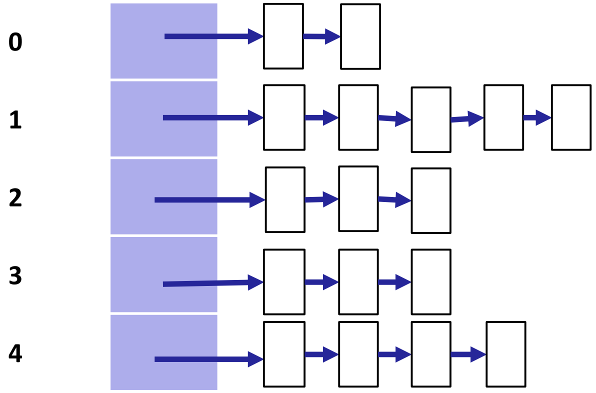
	Find	Add	Remove
LLRB Tree	h = Θ(log N)	h = Θ(log N)	h = Θ(log N)
Separate Chaining Hash Table	Q = Θ(??)	Q = Θ(??)	Q = Θ(??)
LinkedList Map	Θ(N)	Θ(N)	Θ(N)

Poll Everywhere

pollev.com/uwcse373

For this hash table with 5 buckets, give the order of growth for Q with respect to N

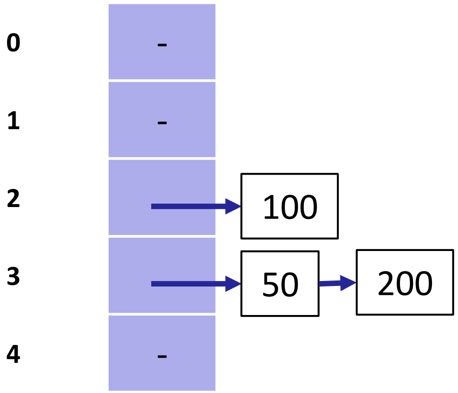
A. Q is Θ(1)
B. Q is Θ(log N)
C. Q is Θ(N)
D. Q is Θ(N log N)
E. I'm not sure …



Separate Chaining: Improving Performance for best/average case

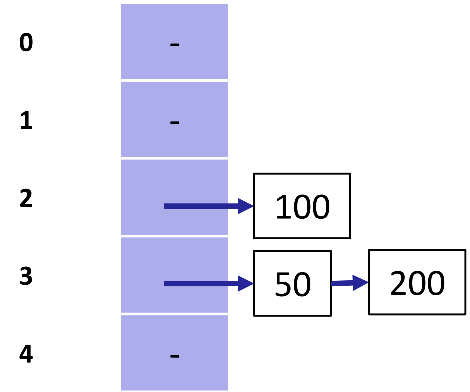
- Suppose we have:
  - A fixed number of buckets M
  - An increasing number of items N

- Even if the items are spread out evenly (ie, best and average cases), lists are of length λ = N/M
  - For M = 5, Q = Θ(N)
  - How can we improve our design to guarantee that N/M is Θ(log N) or even Θ(1)?



Separate Chaining: Improving Performance for best/average case

- Suppose we have:
  - An increasing number of buckets M
  - An increasing number of items N



- Even if the items are spread out evenly (ie, best and average cases), lists are of length λ = N/M
  - ~~For M = 5, Q = Θ(N)~~
  - ~~How can we improve our design to guarantee that N/M is Θ(log N) or even Θ(1)?~~

Make M a function of N

- Example strategy: when N/M >= 1.5, double M
  - This is called “resizing”
  - N/M is called the “load factor” and is often abbreviated λ

Poll Everywhere

pollev.com/uwcse373

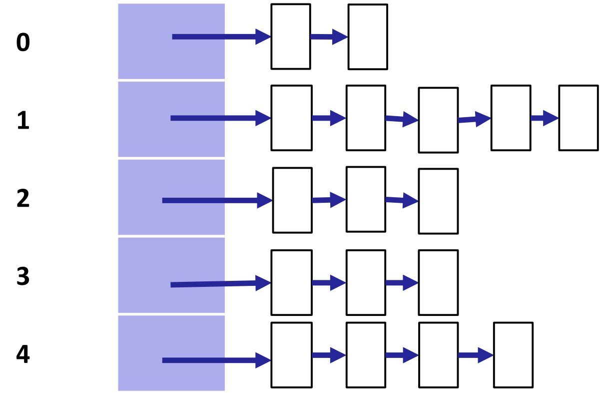
- Demo: https://docs.google.com/presentation/d/1QevjelsyVO8Ea375VRhIf-o--MIMDYB83OxBbXnbQZU/edit#slide=id.g52624185f6_2_2823

- Where will the bucket go? 

A. Index 0
B. Index 1
C. Index 3
D. Index 4
E. Index 7
F. I’m not sure …

Separate Chaining: Runtime Analysis for best/average case

- As long as M ∈ Θ(N), O(λ) ∈ Θ(1)
- *Assuming items are evenly spaced*, lists will be λ items long, resulting in Θ(λ) ∈ Θ(1) runtimes
- What's the cost of a resize?
  - Resizing takes Θ(N) time to redistribute all items
  - However, most add operations (specifically: λ_{target}M adds) will be Θ(1)
- Similar to our resizing arrays, as long as we resize by a multiplicative factor the average runtime will still be Θ(1)



Separate Chaining: Performance

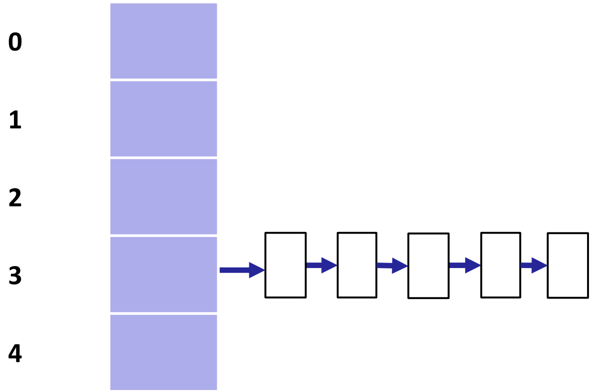
	Find	Add	Remove
LLRB Tree	h = Θ(log N)	h = Θ(log N)	h = Θ(log N)
Resizing Separate Chaining Hash Table *(worst case)*	Q= Θ(N)	Q = Θ(N)	Q = Θ(N)
Resizing Separate Chaining Hash Table *(best/average cases)* +	λ = Θ(1)	λ = Θ(1)*	λ = Θ(1)*
LinkedList Map	Θ(N)	Θ(N)	Θ(N)

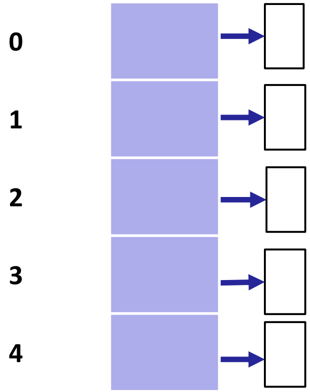
*: Indicates average case

+: Assuming items are evenly spaced

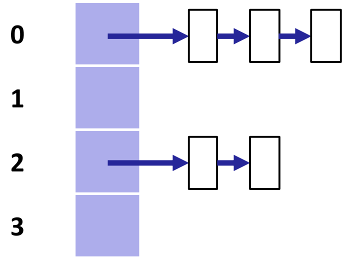
"Assuming items are evenly spaced"

- Hash function uniformity is critical to avoiding worst case





- Hash table size is also critical; it must be relatively prime to the hash function's clusters (if any)
  - Eg, if hash function only returns even numbers, an even-sized hash table would cause clusters



Lecture Outline

- Hash Tables Introduction
- **Handling Collisions**
  - Separate Chaining
  - **Open Addressing**
- Java-specific Gotchas

Yes, We Can! (fourth time's the boon)

- We're mapping strings to an integer
  - Hash the strings and use the hash value as an array index
  - To force our numbers to fit into a reasonably-sized array, we'll use the modulo operator (%)
  - "Probe" for a different bucket

```
WordToPriorityMap m;
m.add("cat", 100);
m.add("bee", 50);
m.add("dog", 200);
```

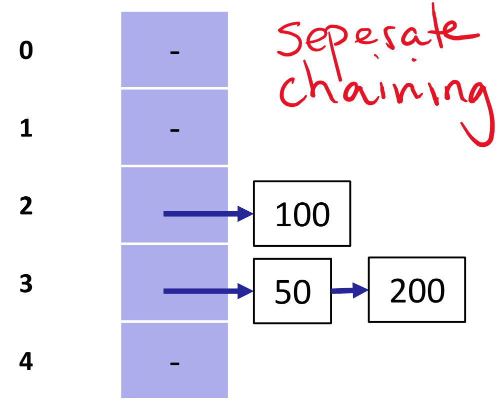
```
hashFunction("cat") == 2;
2 % 5 == 2
hashFunction("bee") == 2525393088;
2525393088 % 5 == 3
hashFunction("dog") == 9752423;
9752423 % 5 == 3
```

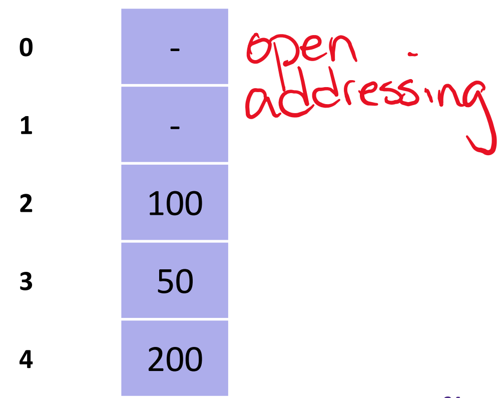
0	-
1	-
2	100
3	50
4	200

try here

Open Addressing

- Linear probing
  - Add one to the index. If already occupied, keep incrementing
  - Demo: http://goo.gl/o5EDvb
- Quadratic probing
  - Add one to the index. If already occupied, look 4 ahead, then 9 ahead, then 16 ahead, then …
- Many other possibilities, but not often used in practice
  - Load factor λ must be carefully managed to prevent excessive (or infinite) time spent probing





Lecture Outline

- Hash Tables Introduction
- Handling Collisions
  - Separate Chaining
  - Open Addressing
- **Java-specific Gotchas**

Java Gotchas (1 of 2)

- Java's hash table implementation is the HashSet/HashMap
  - The hash function is Object's hashCode(), which is a 32-bit number
  - Java's equals() method is implemented as *memory address* equality

- **Warning #1**: Don't override equals() without also overriding hashCode()
  - Leads to items getting lost and other weird behavior
  - HashMaps/HashSets use equals() to determine if an item exists in a particular bucket, but hashCode() to find the item in the bucket

Java Gotchas (2 of 2)

- **Warning #2**: Don't store objects that can change in a HashSet/HashMap!
  - If an object's members can change, then its hashCode() changes. Again, items may get lost.

- **Warning #3**: Most cryptographic hashes consider 32-bits substantially too small
  - But do you need cryptographic-quality hashing?

tl;dr

- Hash Tables combine hashing and data-indexed arrays
  - Collision resolution is tricky!
  - Managing load factor λ and smart resizing yields Θ(1) runtime

	Find	Add	Remove
Resizing Separate Chaining Hash Table *(worst case)*	Q = Θ(N)	Q = Θ(N)	Q = Θ(N)
Resizing Separate Chaining Hash Table *(best/average cases)* +	Θ(1)	Θ(1)*	Θ(1)*
LLRB Tree	h = Θ(log N)	h = Θ(log N)	h = Θ(log N)
B-Tree	h = Θ(log N)	h = Θ(log N)	h = Θ(log N)
BST	h = Θ(N)	h = Θ(N)	h = Θ(N)
LinkedList	Θ(N)	Θ(N)	Θ(N)

*: Indicates average case
+: Assuming items are evenly spaced