Hash Tables

15-121 Fall 2020

Margaret Reid-Miller

Today

- Sets and Maps review
- Hash Tables
- Next time
  - hashCodes
  - Priority Queues

List

- Sequence of elements
- Indexed starting at 0… (an index)
- A list can have duplicates.

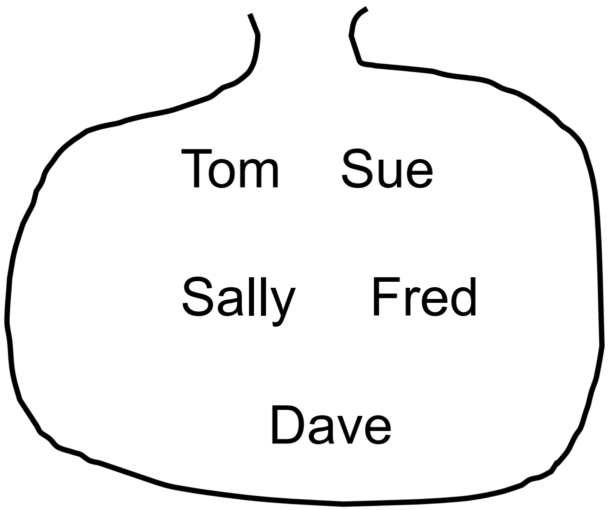
Set
(sometimes called a bag)

A set is "bag" of objects

- No duplicates with respect to `.equals()`
- Membership

Operations I want to be fast:

- Does the set **contain** this element?
- **Add** this element to the set
- **Remove** an element from the set



Map
(also called dictionary or associative array)

A map is a table of (key,value) pairs.

- Indexed by key (must be unique).
- Many keys can "map" to the same value.

Operations I want to be fast:

- **Get** Tom's section
- **Set** Dave's section to B
- **Remove** Fred from the class

Name	**Section**
Tom	A
Fred	B
Dave	A
Sally	C

TreeSet / TreeMap

- `TreeSet` is a class that implements a **sorted** `Set`.
- `TreeMap` is a class that implements a **sorted** `Map`.
- **Advantages:**
  - The `TreeSet /TreeMap` can be traversed (using an iterator) in order.
  - Subsets/submaps based on a range of values can be generated easily from a `TreeSet/TreeMap`.
- **Disadvantages:**
  - Contains, insert, and remove operations on the `treeMap` take O(log N) time for sets with N elements.

Use a TreeMap only when you need the keys in order

Both `TreeSet` & `TreeMap` use a balanced binary search tree called a “red-black tree".

Red-Black balanced binary search trees:

- The height of a red-black tree is guaranteed to be 2 log n.
- Every time you add or remove an element, the tree may be restructured to maintain balance.
- The runtime to rebalance the tree is worst case O(log n).

Thus

- Operations `contains/add/remove` for Sets and `get/set/remove` for map have worst-case O(log n) runtime.

Worst case O(log n) time!

Great! We're done!

The course is over! Yay!

Unless…

- Can we do better than worst-case O(log n)?
- What would be better?

    O(1)

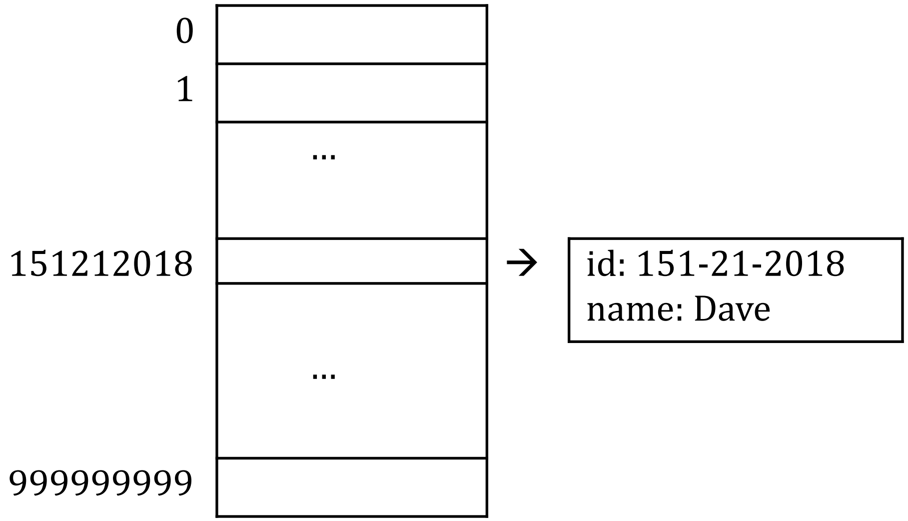
- What data structure do we know that usually gives O(1) time?

    arrays

- E.g., Suppose we want to maintain a set of students, where a student object has a 9-digit id and a student name.
- How can we use the student id to find a student in an array?

Really Big Array (?)

Use the student id as the index into a really big array:
`contains`: O(1), `add`: O(1), `remove`: O(1). Yay!!!



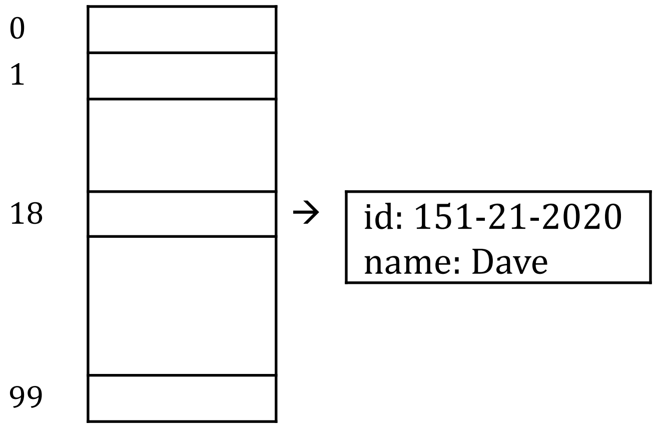
Problem: Memory hog: The range of student id values is independent on the number of students (size of the set).

Moderate Size Array (better)

Key Idea: Use the key **to compute** an index into a moderate size array.

- Want: contains, add, remove: O(1), memory: O(n)

Example: Use last two digits of the student id



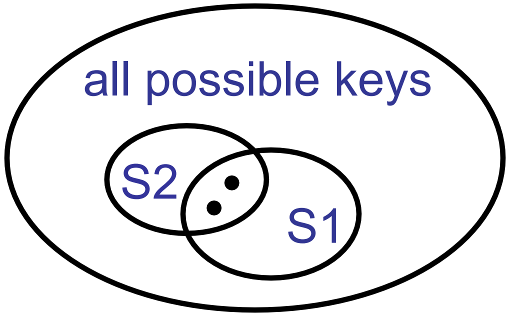
- **Problem:** Two or more students might have the same last two digits.

Hash Table

- **Hash Table** – An array that refers to elements in set/map
- **Hash Function** – A function that maps a key to an index in hash table
  - hash(key) -> index
- But if you want to allow for **any set** of student id values, then we have to deal with the fundamental problem of collisions.
- **Collision:** when some keys map to the same index:

$$x \neq y, \text{ but } hash(x) = hash(y).$$

Collisions



- *Can we prevent collisions when we don't know in advance which keys will be used in the set?*
  - No. Since the number of possible keys is much greater than the size of the hash table, there must be two keys that map to the same index.
  - Any set that contains those two keys will have a collision.

- **Pigeonhole Principle**: If you put more than n items into n bins, then at least one bin contains more than one item.

The Birthday Paradox

- *How likely are two keys going to hash to the same index?* Surprisingly likely!
- Probability that none of n people have the same birthday:

  p' = 1*(364/365)*(363/365)*…*((365- n +1)/365)
- Probability at least two people have the same birthday is p = 1 – p'

  When n = 23, p = 0.5.
  When n = 30, p = 0.7
  When n = 50, p = 0.97 !!

Hash Function

Desired properties of a hash function:

1. The hash function should be fast to compute: O(1)
2. Limited number of collisions:
   - Given two keys, the probability they hash to the same index is low.
   - When table has many keys they should be “evenly” distributed.

Examples of hash functions:

- If the key is an integer:
  `key % tablesize`
- If key is a String (or any Object):
  `key.hashCode() % tablesize`

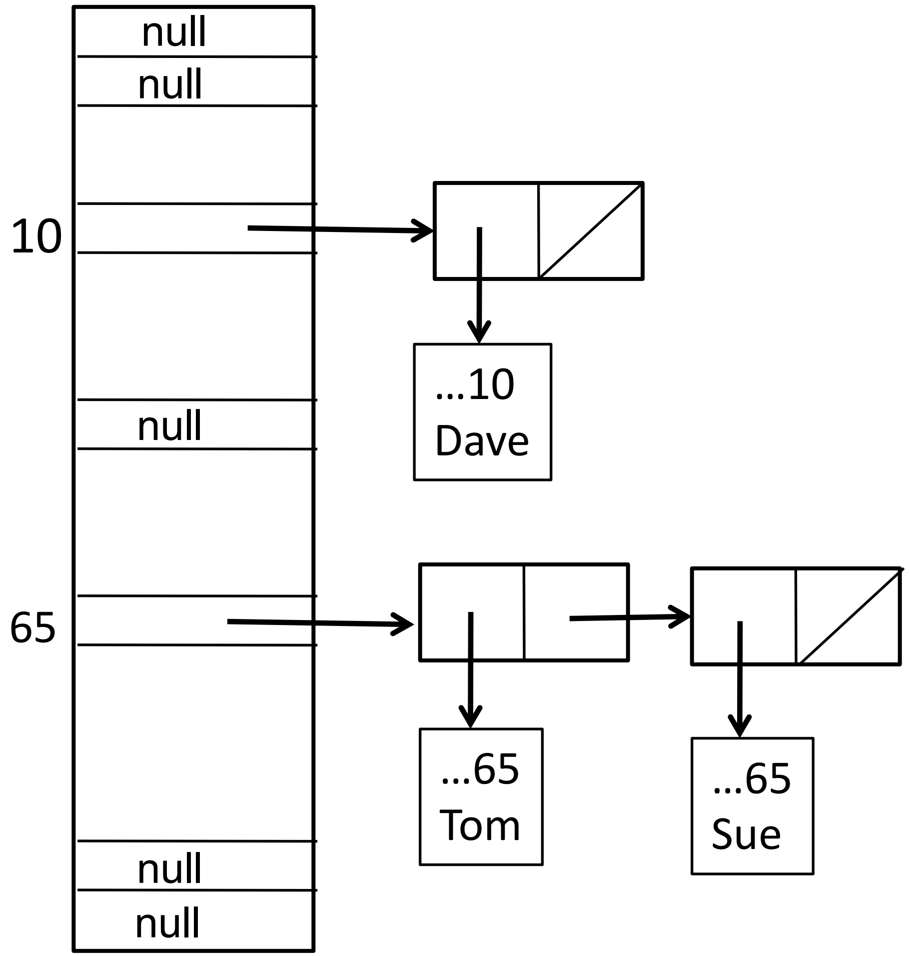
Handling Collisions

1. **Open Addressing** (topic for 15-451)
2. **Separate chaining** – Each index of array contains all the elements that hash to that index (called a **bucket**)

What data structure should we use to maintain a bucket?

- Often a linked list because:
  - Buckets are small (few collisions)
  - Linked lists easy to implement
  - Many buckets can be empty and empty linked lists take no storage
  - No additional constraints such as Comparable

Separate Chaining using a linked list



Set operations using Separate Chaining

`contains(obj):`

- Find the index in the array using the hash function on `obj`
- Check if any element in the bucket equals `obj`

`add(obj):`

- Find the index using the hash function on `obj`
- If no element in the bucket equals `obj`, add `obj` to the bucket

`remove(obj):`

- Find the index using the hash function on `obj`
- Remove `obj` from bucket, if it exists

Runtime

What is the worst-case runtime for contains, add, remove?

O(n) – all the keys hash to the same index

What is the best-case runtime?

O(1) – only a few keys map to any one index

What is the expected runtime?

O(1) – assuming the hash function is good, and the hash table is not too full

Load Factor

Load Factor: (number of elements) / (length of array)

What is the expected size of a bucket?

The load factor

What is a good load factor?

A small constant so that the linked list stay short, even the longest ones.

Java uses a default value of 0.75

Can the load factor be larger than 1? Yes

Space vs Time

What if we keep adding elements and the load factor increases?

- The probability of a collision increases.
- Linked lists can get long and runtimes go up.
- Even worse, the longest list linked list may be much larger than the average length.

Space vs time trade-off:

- Decrease array size
  - more collisions – slower contains, add, remove
- Increase array size
  - fewer collisions – faster contains, add, remove

Rehashing

If the load factor gets too big what can we do?

Create a larger table.

Can we just copy the elements to a new larger table?

NO! We need to reinsert each element of the old table in the new table using a **new hash function**.

How much bigger should we create the array?

Approximately twice the size (adds only O(1) amortize time)

Hashing in Java

Every Java object inherits from the Object class:

```
boolean equals(Object obj)
String toString()
int hashCode()
```

How can you use these methods to implement hashing?

Step 1. Use the hashCode method of the object to get a (random-like) integer of it. (For a map use the hashCode of the key.)

What range of values can it return?

-2.1 billion to 2.1 billion

Hashing in Java

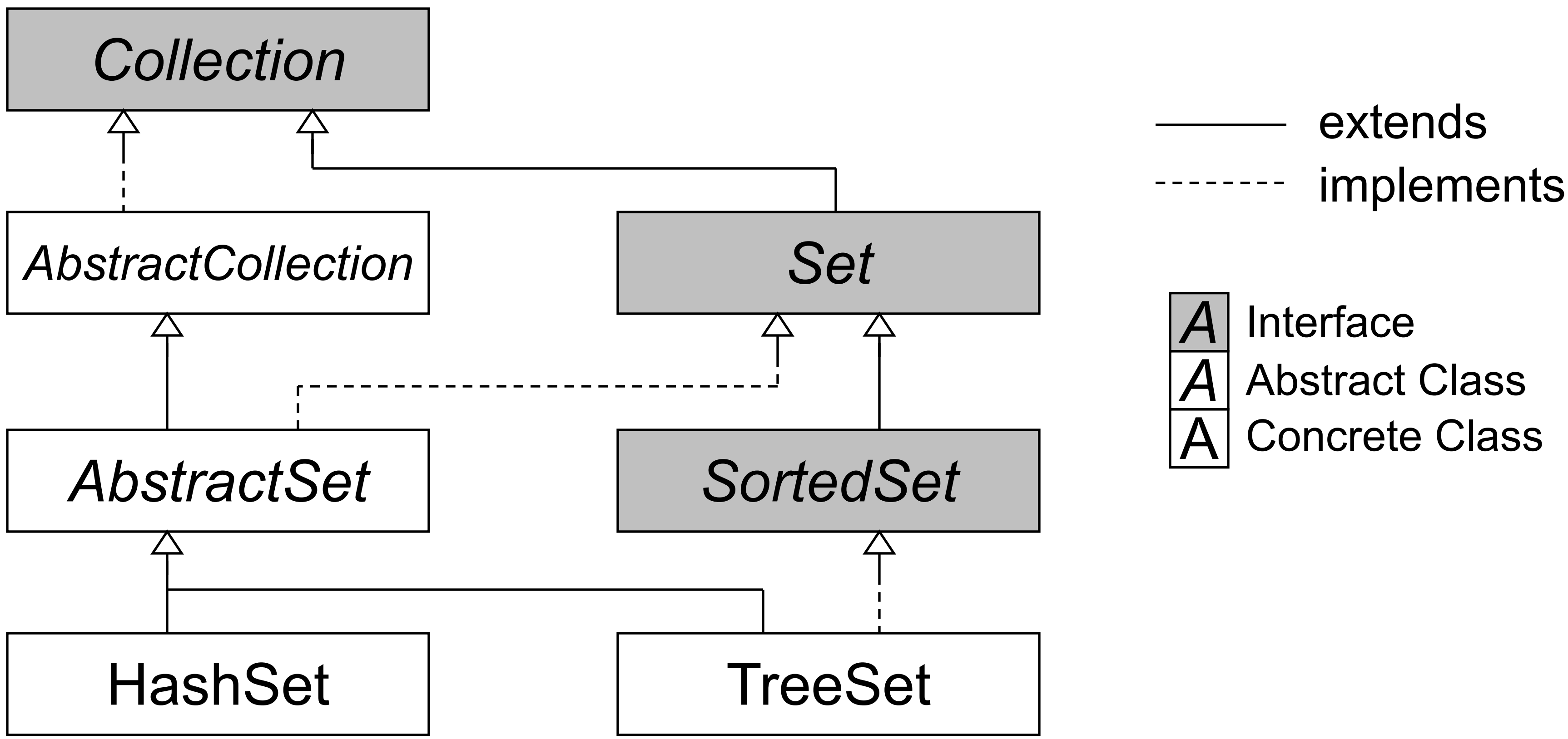
Step 2: To get an index in the range of the array take modulus of the hash with the length of the array. Mod will spread all possible hashCode values evenly.

```
Math.abs(obj.hashcode() % (array.length));
```

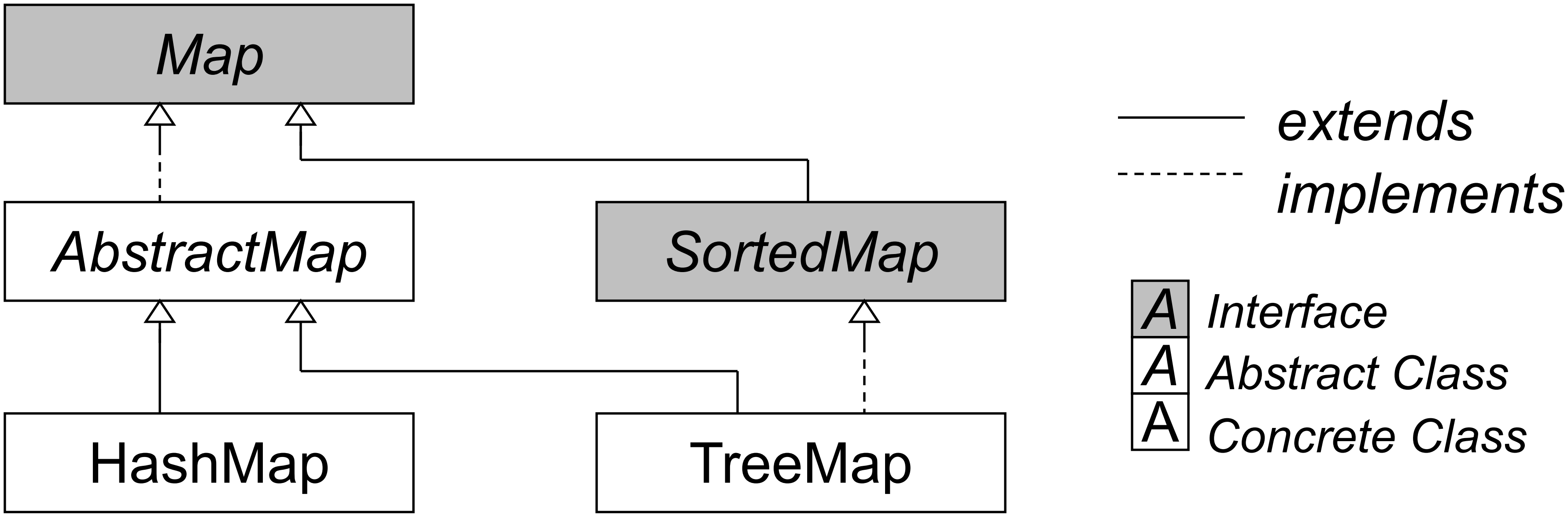
Why do we need to take the absolute value?

Step 3: Use `.equals` to determine if an element is in the bucket at that index.

Sets in the Java API



Maps in the Java API



`HashSet` is a class that implements a set

The elements of the set are stored using a hash table.

- elements' class must override `equals()` and `hashCode()` (more about this soon).

Advantages:

- The `HashSet` supports search, insert, and remove operations in O(1) expected time.

Disadvantages:

- Traversals cannot be done in a meaningful way with a HashSet.

If the order of the elements is unimportant, use a `HashSet`. **It's fast.**

HashSet Example

```
Set<Integer> a = new HashSet<Integer>();
Set<Integer> b = new HashSet<Integer>(10);
a.add(1);
a.add(5);
b.add(1);
b.add(9);
b.add(0);
a.addAll(b);
for (Integer i : a)
   System.out.println(i);
```

Initial capacity (points to the argument 10)

Iterator used here accesses each element of set **in no particular order** since the set is implemented with a hash table. (More about this soon.)

`HashMap` is a class that implements a map

- The (key,value) pairs of the map are stored using a hash table. Again, keys must override `hashcode()` (more about this soon).
- **Advantages:**
  - The `HashMap` supports search, insert, and remove operations in O(1) expected time.
- **Disadvantages:**
  - Traversals (using an iterator) cannot be done in a meaningful way with a HashMap.

If key order is unimportant, use a HashMap. It's fast.

HashMap Example

Key	Value
K1	V1
K2	V2
K3	V3
K4	V4

```
Map<String, String> tvShowMap
              = new HashMap<String, String>();
tvShowMap.put("The Simpsons", "FOX");
tvshowMap.put("Grey's Anatomy", "ABC");
tvshowMap.put("How I Met Your Mother", "CBS");
...
System.out.println("The Simpsons is on " +
                 tvShowMap.get("The Simpsons"));

System.out.println("CSI changes networks!");
String oldNetwork = tvShowMap.put("CSI","NBC");
```