Poll Everywhere

PollEv.com/javabear text **javabear** to 22333

Implementing a Map by composing with which of these data structures results in a `get()` implementation that has a strictly better runtime complexity than `put()`?

An unsorted list	get(): O(N)	put(): O(N)	(A)
A sorted `DynamicArrayList`	get(): O(log N)	put(): O(N)	(B)
A sorted `SinglyLinkedList`	get(): O(N)	put(): O(N)	(C)
A BST	get(): O(H)	put(): O(H)	(D)



Lecture 20: Hashing

CS 2110

November 3, 2026

Today's Learning Outcomes

76. Identify the steps of hashing and describe the properties of a good hash function.

77. Visualize the state of a hash table after performing a given sequence of operations.

78. Contrast hash collision resolution by *chaining* and *linear probing*. Identify scenarios where each implementation might be preferable.

An Efficient Set Representation

Suppose we need to model a `Set<Integer>` in our program, and we know ahead of time that it will only store values between 0-9999. What representation can offer the most efficient `add()`, `remove()`, and `contains()` queries?

List? O(N) add()

Sorted List? O(N) add()

BST? O(logN) if balanced

Can we achieve O(1) complexity for all operations?

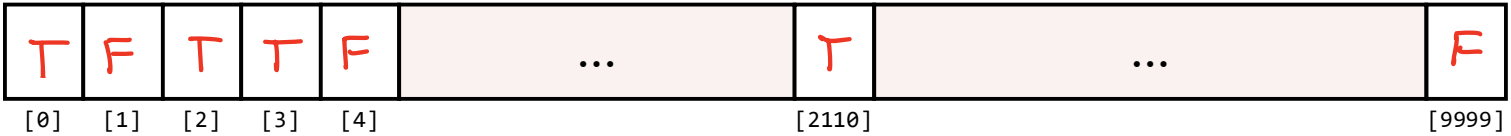
Modeling a Set with an Array

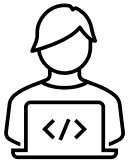
Suppose we need to model a `Set<Integer>` in our program, and we know ahead of time that it will only store values between 0-9999. What representation can offer the most efficient `add()`, `remove()`, and `contains()` queries?

Idea: Model using a `boolean[10000]`

arr[i] = does set contain i?

{0, 2, 3, 2110}





Each operation accesses/modifies only one array cell and does O(1) other work, so they have O(1) runtimes.

(but this uses a lot of space...)

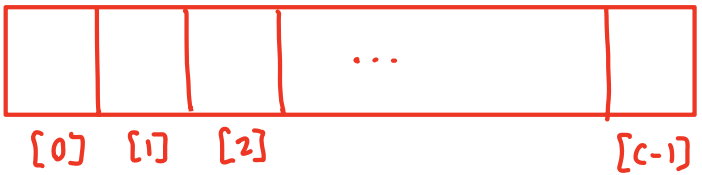
Motivating Hash Tables

Idea: Use the *values* of the objects as indices in a (dynamic) array where they are stored.

Combined with a random access guarantee, this lets us "jump directly" to that object (or where it belongs) in the collection.

We call the data structure that realizes this strategy a **hash table** and the array cells its **buckets**.

Capacity C = # of buckets



Most data we store isn't just numbers. We need a way to convert any object $\rightarrow \{0, 1, \ldots, C-1\}$

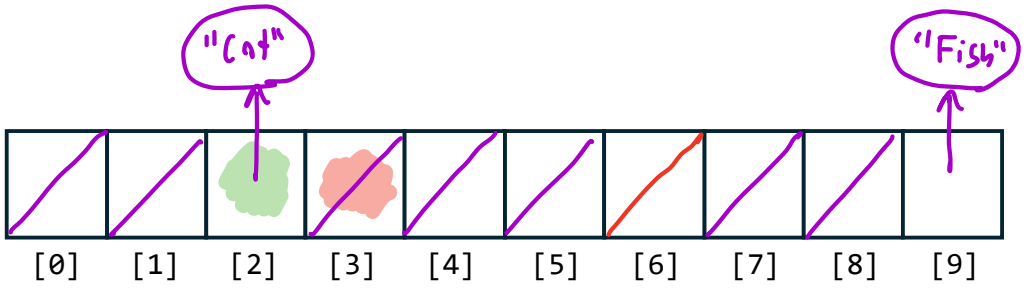
Illustrating Hash Table Operations

```
HashSet<String> set = new HashSet<>();
set.add("Cat");
set.add("Dog");
System.out.println(set.contains("Cat"));
System.out.println(set.contains("Pig"));
set.add("Fish");
set.remove("Dog");
```

→ true

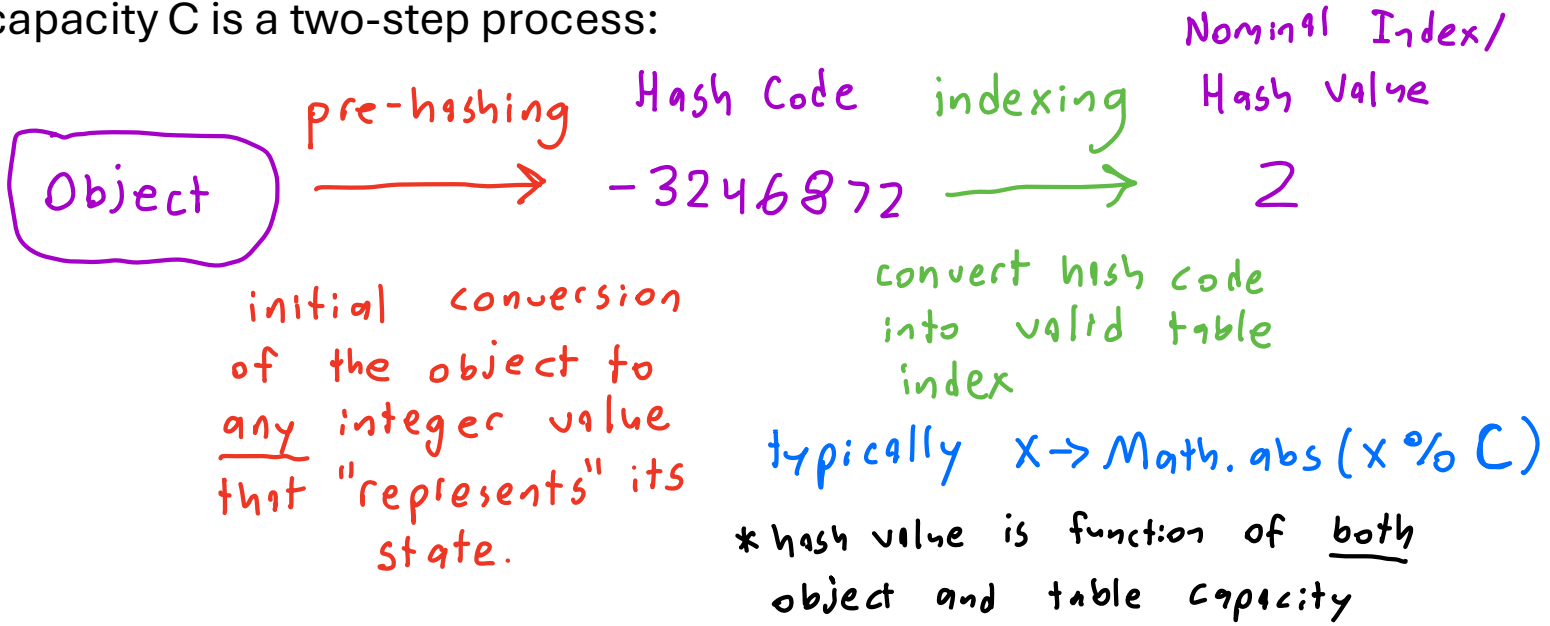
→ false

String	Hash Value
"Cat"	2
"Dog"	6
"Fish"	9
"Pig"	3



Hashing Objects

Computing the **"nominal index"** of an object in a hash table with capacity C is a two-step process:



The hashCode() Method

= A method of the `Object` class that is used for pre-hashing.

It should be every object's responsibility to report its hash code since it has full information about its own state.

Requirements:

Determinism: `x.hashCode()` returns the same value if `x` is not mutated.

Consistency: If `x.equals(y)`, then `x.hashCode() == y.hashCode()`

Desire:

`hashCode()` should "evenly spread out" objects across all `int` values

* See extended discussion about this last point in the lecture notes.



Coding Demo: Strange HashSet behavior



Poll Everywhere

PollEv.com/javabear text **javabear** to 22333

Why does this code print "false" even though list is in set?

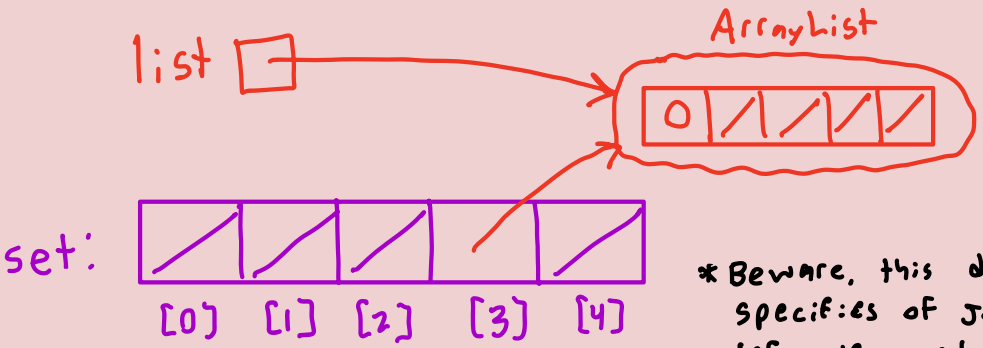
```
HashSet<ArrayList<Integer>> set = new HashSet<>();
ArrayList<Integer> list = new ArrayList<>();
set.add(list);
list.add(0);
System.out.println(set.contains(list));
```

(For example:)

hash value

empty list → 3

list with {0} → 1



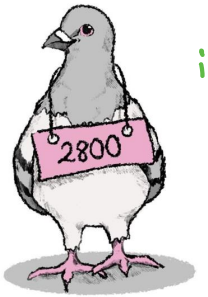
* Beware, this diagram abstracts away many specifics of Java's memory model, but keeps info we need to trace this issue.

Hash Collisions

There are only $2^{32} \approx 4$ billion int values in Java.

Most reference types have many more possible distinct states. For example, Strings, arrays, Lists, etc.

As a consequence (of the pigeonhole principle), sometimes distinct objects will have the same hash code.



Moreover, different hash codes will frequently be indexed to the same value.

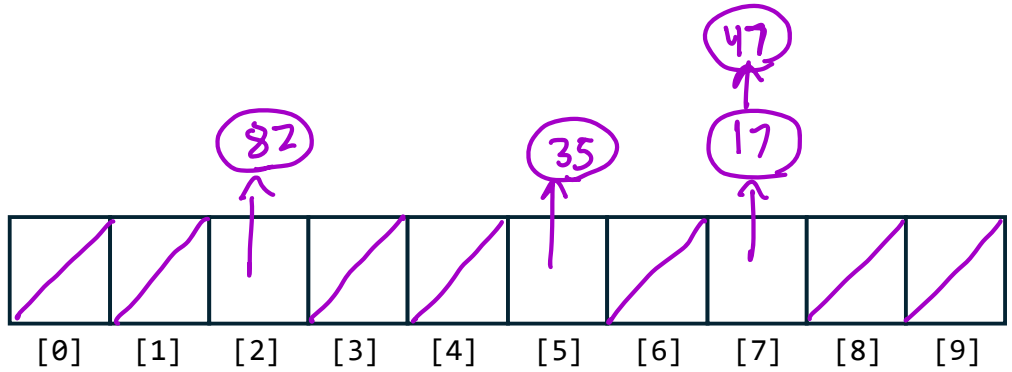
We need a way to deal with these collisions, where multiple elements "want" to be in the same bucket.

Chaining Hash Tables

Rather than storing its elements of type T in an array of type T[ ], a **chaining hash table** uses an array of collections storing type T.

- Most common: An array of `LinkedList<T>`s
- Every element is guaranteed to be stored at its nominal index.
- Once we reach that bucket, we need to search that "inner" collection.

add(35)
add(17)
add(82)
add(47)





Coding Demo: Chaining Hash Tables (Part 1)

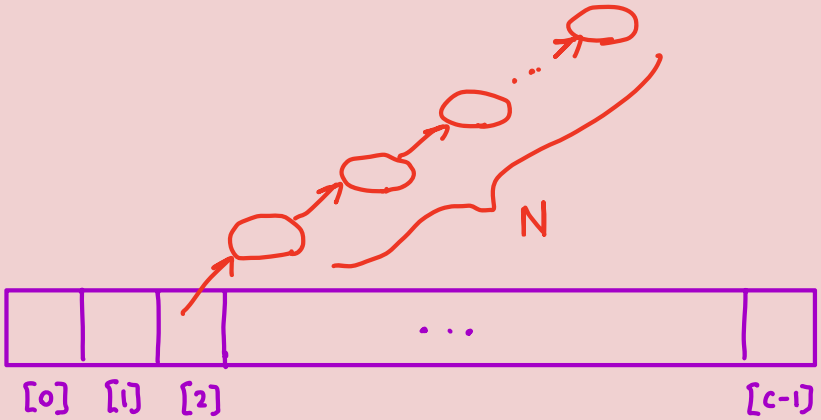


Poll Everywhere

PollEv.com/javabear text **javabear** to 22333

What is the worst-case runtime of add( ) in a chaining HashSet with capacity C that contains N elements?

Worst-case: Every element collides in same bucket



$O(1)$	**(A)**
$O(C)$	**(B)**
$O(N)$	**(C)**
$O(N/C)$	**(D)**

Expected Runtime Analysis

Simple Uniform Hashing Assumption: (SUHA)
Every object is equally likely to be hashed to each bucket of a hash table, and its placement is not affected by the objects already in the hash table.

If we add() N elements into C buckets, we expect them to be evenly distributed under SUHA, so each inner collection has size $\approx N/C$

Def. Load factor $\lambda = N/C$ = expected bucket size (of chaining HT)

All chaining hash table operations are expected $O(\lambda)$

If we keep C constant, $\lambda = O(N)$... not so good

Instead, grow C with N to keep λ constant.

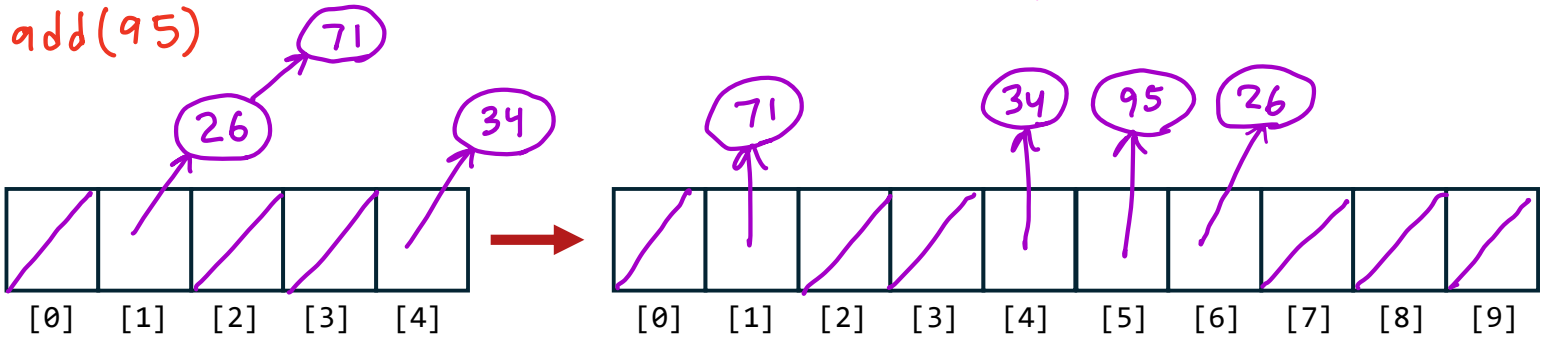
Hash Table Resizing

We want to ensure that $\lambda = {}^{N}/_{C}$ remains upper bounded by some constant.

- Every time we add an element to a hash table, λ increases.
- When it exceeds our bound, we must make C bigger to reduce λ.
- Our strategy: double C, which ensures good *amortized* complexity

Example: $\lambda \leq 0.75$

* Need to re-hash all elems during resize since hash value changes with capacity!





Coding Demo: Chaining Hash Tables (Part 2)

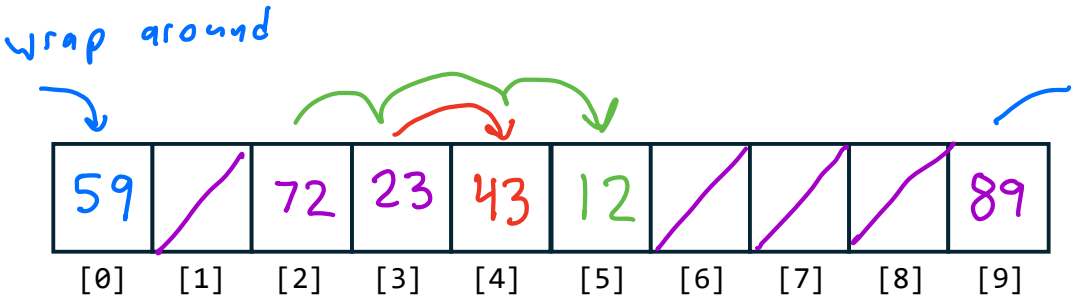


Probing Hash Tables

Unlike chaining hash tables, each bucket can only hold up to one element of type T.

Collision Resolution: If an object's nominal bucket is full, check the next bucket, and keep "probing" until we find an empty bucket to place it.

add(23)
add(72)
add(43)
add(12)
add(89)
add(59)

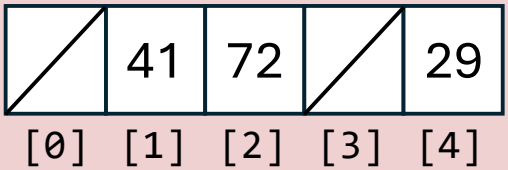


Poll Everywhere

PollEv.com/javabear text **javabear** to 22333

Suppose that we call `add(36)` on a probing `HashSet` with max load factor 0.75 and the following state:

* this will trigger a resize and re-hash

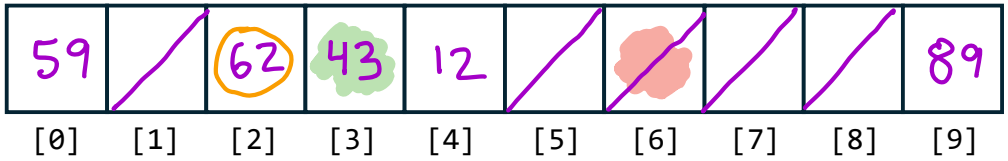


no collisions after

At which index will 36 reside after this operation?

1 **(A)**	3 **(C)**
2 **(B)**	6 **(D)**

contains() in Probing Hash Tables



Always start at nominal index:

Best case: Nominal bucket contains element contains(43) == true

Nominal bucket is empty contains(16) == false

If nominal bucket contains different element, we have more work to do. contains(12) can't just look at bucket 2 since 12 probed to bucket 4.

We'll pick up from here next lecture.