

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.

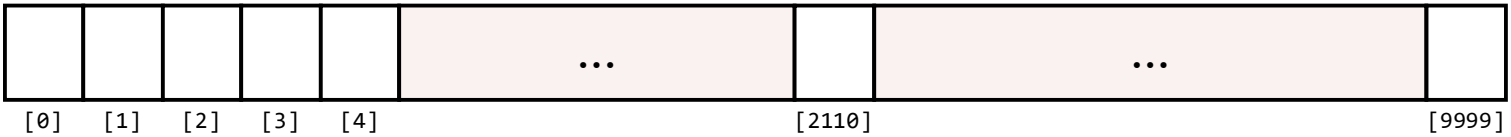
An Efficient Set Representation

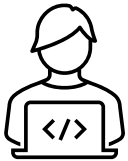
Suppose we need to model a `Set<Integer>` in our program, and we know ahead of time that it will only store values between 0-9999. What representation can offer the most efficient `add()`, `remove()`, and `contains()` queries?

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]`

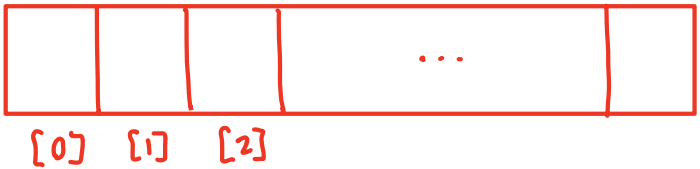




Motivating Hash Tables

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

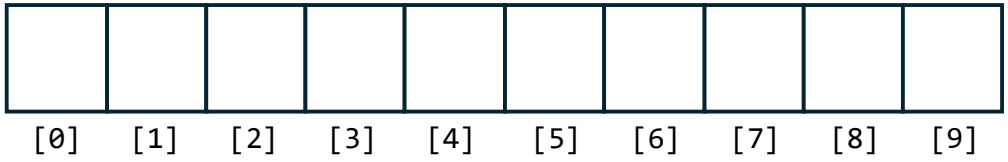
Combined with a random access guarantee, this lets us "jump directly" to that object (or where it belongs) in the collection.



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");
```

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.



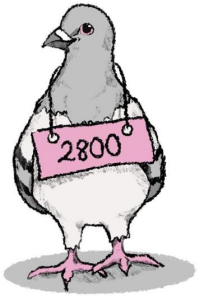
Coding Demo: Strange HashSet behavior



Hash Collisions

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

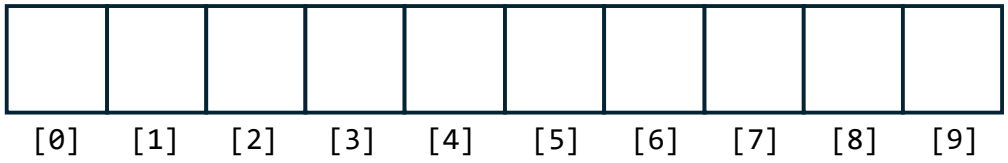
Most reference types have many more possible distinct states. For example, Strings, arrays, Lists, etc.



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.





Coding Demo: Chaining Hash Tables (Part 1)



Expected Runtime Analysis

Simple Uniform Hashing Assumption: (SUHA)

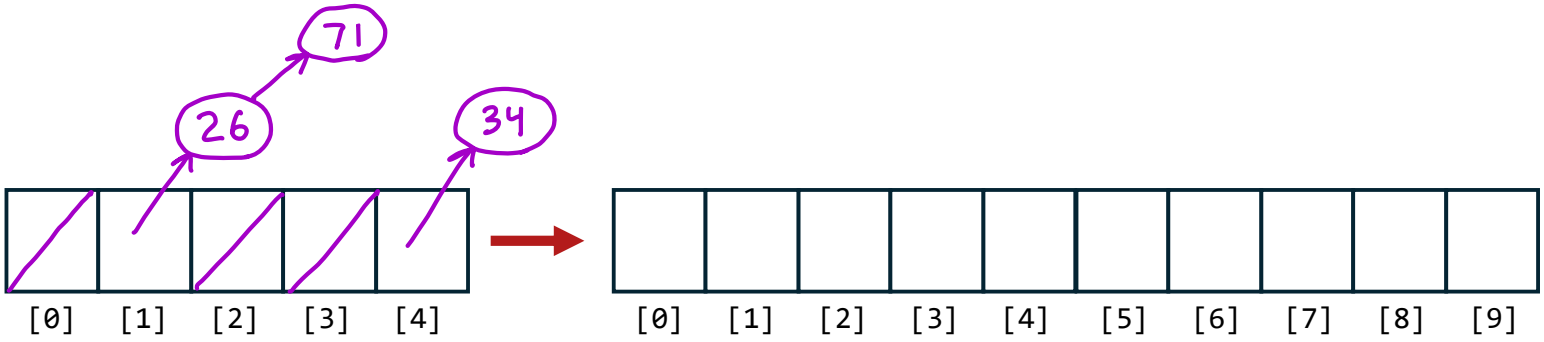
Every object is equally likely to be hashed to each bucket of a hash table, and its placement is not affected by the objects already in the hash table.

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 λ, which ensures good *amortized* complexity

Example: $\lambda \leq 0.75$





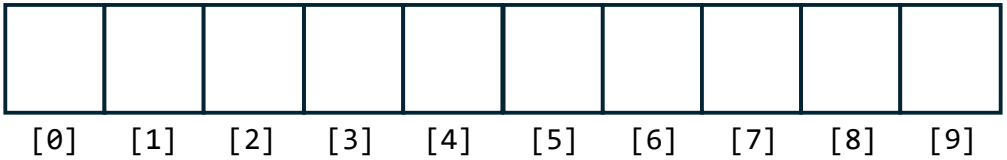
Coding Demo: Chaining Hash Tables (Part 2)



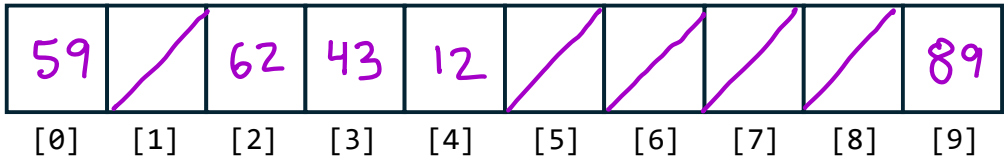
Probing Hash Tables

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

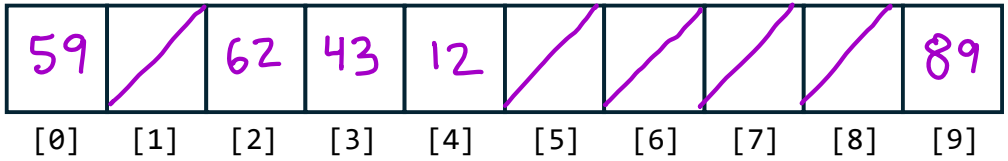
Collision Resolution: If an object's nominal bucket is full, check the next bucket, and keep "probing" until we find an empty bucket to place it.



`contains()` in Probing Hash Tables



remove() in Probing Hash Tables



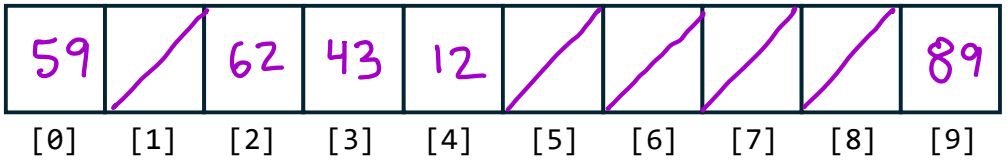
Just like other methods, start at nominal index and probe for element.

Incorporating Tombstones

When we add tombstones to our hash table in the `remove()` method, we'll need to account for them in our `add()` and `contains()` logic.

`add()`: Probe until we reach an empty bucket or a tombstone (which we can overwrite).

`contains()`: Probe past tombstones, since only empty buckets signal the end of a probing sequence.



Probing Hash Table Performance

The expected runtime of all operations in a probing hash table is also upper bounded by a function of λ, so it also amortizes to $O(1)$ with proper resizing.

Comparing Chaining vs. Probing:

- Probing has worse dependence on λ since nearby hash values can affect performance, not just exact collisions.
- Probing hash tables have smaller, centralized storage, so they can perform better due to hardware considerations like *cache locality* that fall outside of our scope.