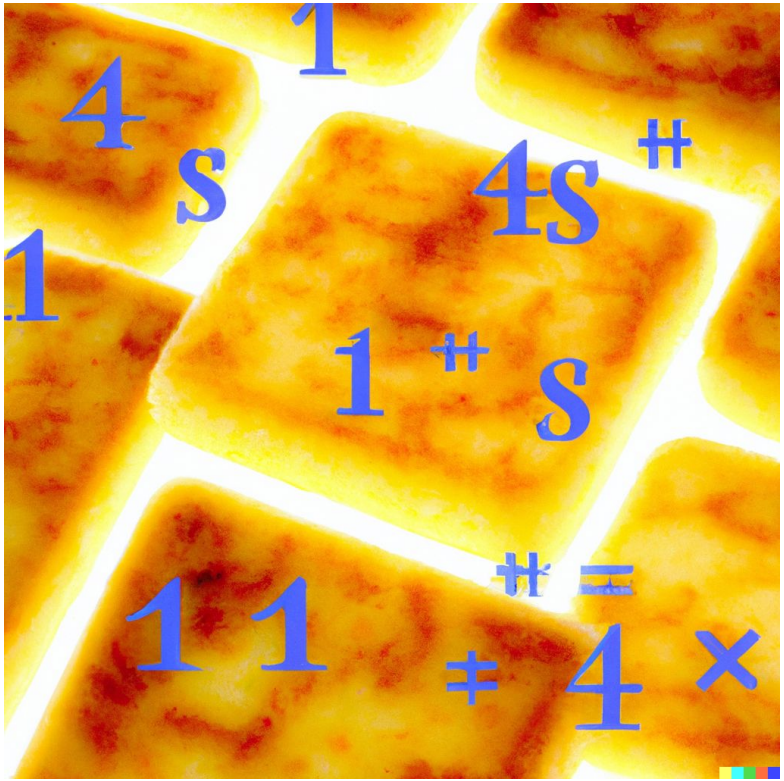
Lecture 20

Hash Tables II

CS61B, Spring 2024 @ UC Berkeley

Slides Credit: Josh Hug



Visualization for Some Basic Cases

Lecture 20, CS61B, Spring 2024

Visualization for Some Basic Cases

hashCode and Equals

- Why Custom Hash Functions?
- contains
- Duplicate Values

The Danger of Mutable Keys

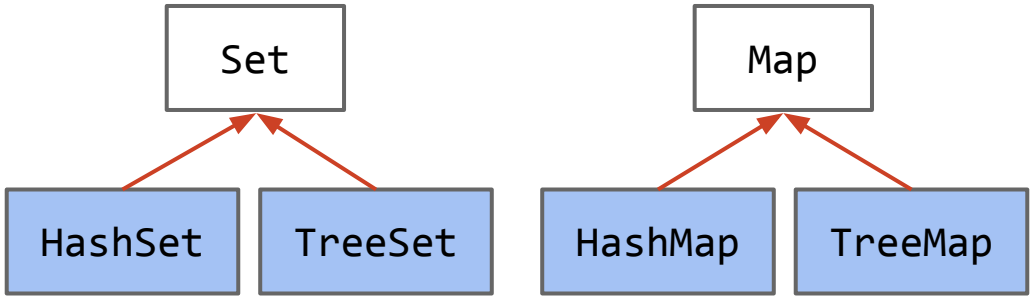
- Mutable vs. Immutable Types
- Mutable Hash Table Keys

A Peek into Java HashSets

Sets and Maps

We've now seen the two implementation philosophies for Sets and Maps.

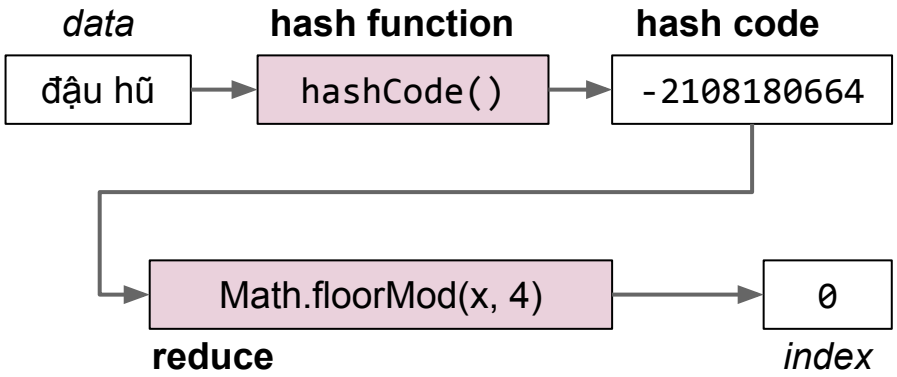
- Red black tree based approach: TreeSet/TreeMap.
  - Requires items to be comparable.
  - Logarithmic time operations.
- Hash table based approach: HashSet/HashMap.
  - Constant time operations if the hashCode spreads the items out nicely.



Hash Tables in Java: Recap

Hash tables:

- *Data* is converted into a hash code, the **hash code** is then **reduced** to a valid *index*.
- *Data* is then stored in a bucket corresponding to that *index*.
  - Each bucket is a “separate chain” of items.
- Resize when load factor N/M exceeds some constant.
- If items are spread out nicely, you get Θ(1) average runtime.



Hash based set ops

	contains(x)	add(x)
Bushy BSTs	Θ(log N)	Θ(log N)
Separate Chaining Hash Table With No Resizing	Θ(N)	Θ(N)
… With Resizing	Θ(1)†	Θ(1)*†

*: Indicates “on average”.
†: Assuming items are evenly spread.

HashTableVisualizer

Let’s play around with a hash table visualizer.

Goal, get a deeper understanding of:

- How hash codes affect the distribution of items.
- The interaction between equals and hashCode.
- Why hash tables are fast even though they use linked lists.

ColoredNumbers

The objects we're inserting into the HashTable are of type ColoredNumber. Each has 2 attributes:

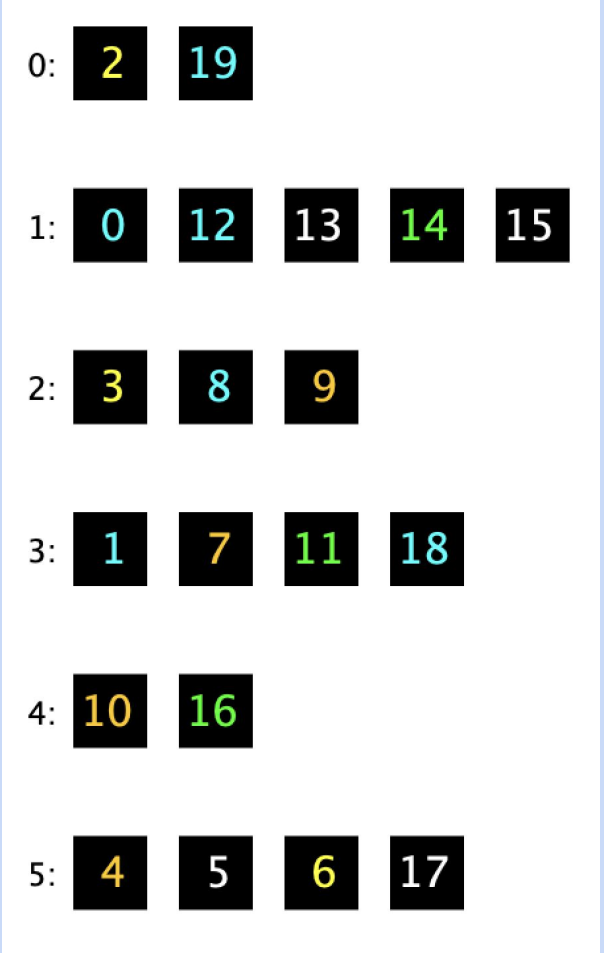
```
private int num;
private Color color;
```

Let's see what happens when we insert ColoredNumbers 0 through 19 into a hash table with 6 buckets.

Distribution of Items

What do you notice about the distribution of items?

Why do you we get this distribution?



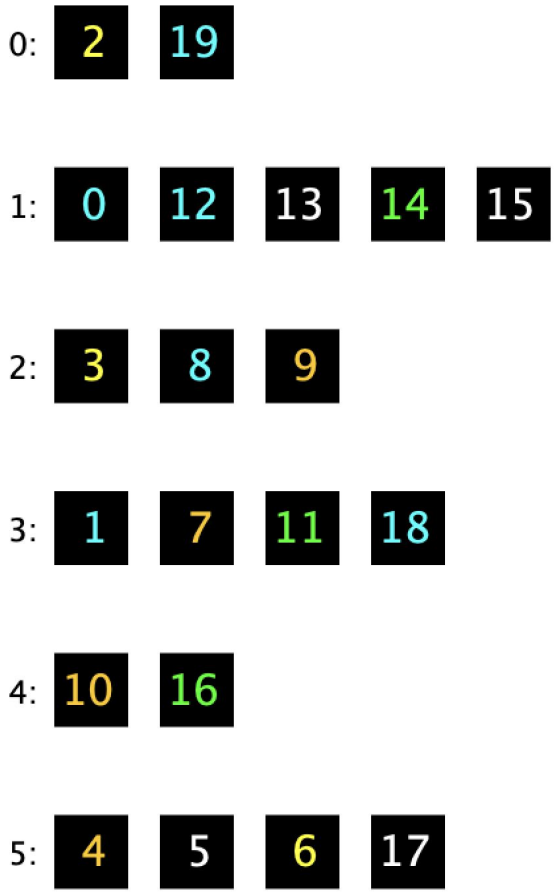
Distribution of Items (Your Answer)

What do you notice about the distribution of items?

- No bin has more than 5 items
- Odd indexes seem to have more items
- Hash doesn't seem to correspond well with the item

Why do you think we get this distribution?

- Average bin length is N/M
- No clear pattern, nondeterministic (across runs)



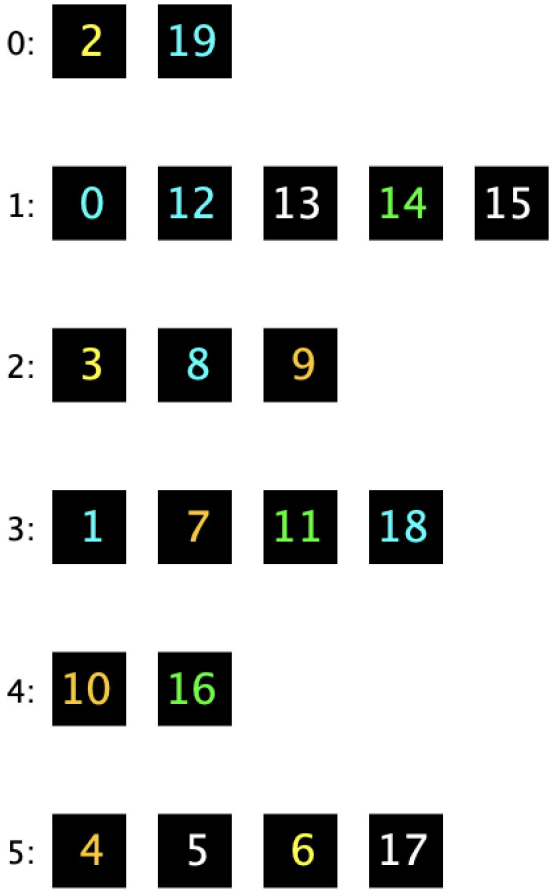
Creating a Custom hashCode Function

Suppose we create a hashCode function that returns 0.

```
@Override
public int hashCode() {
   return 0;
}
```

What distribution do we expect?

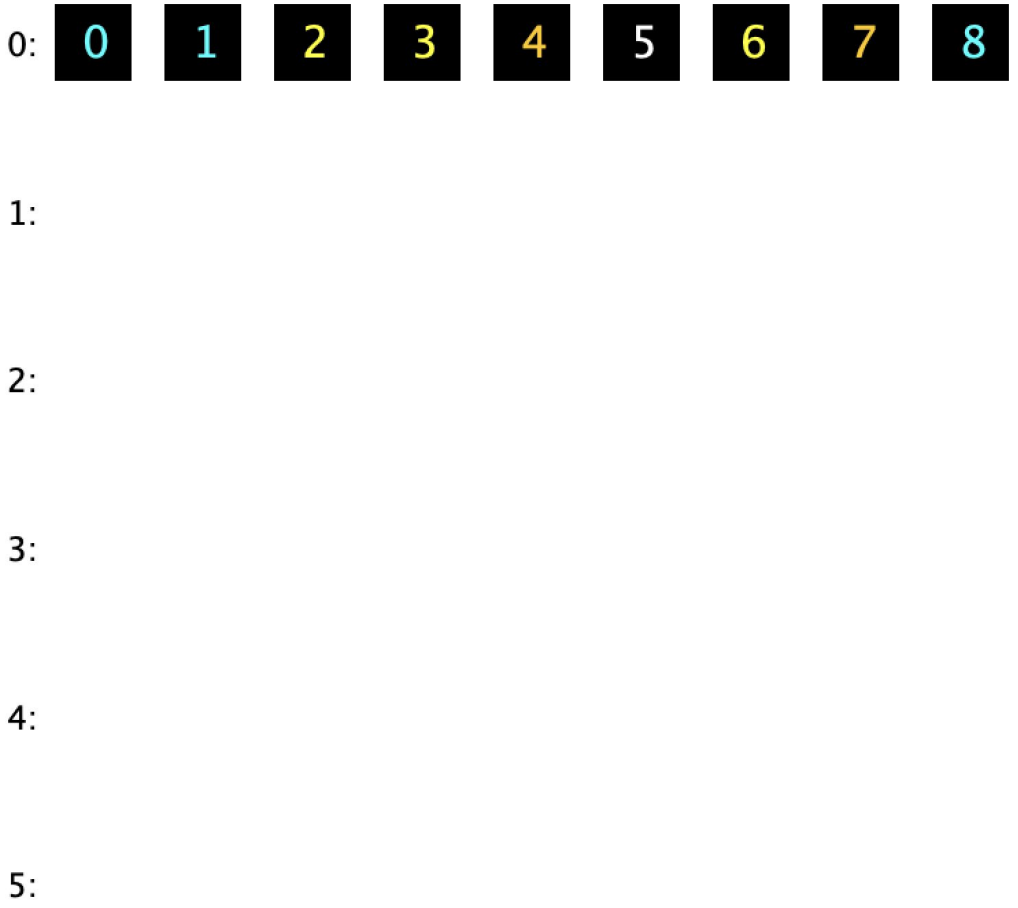
- In other words, how will the figure to the right change?



Let's try it out.

The Zero HashCode Function

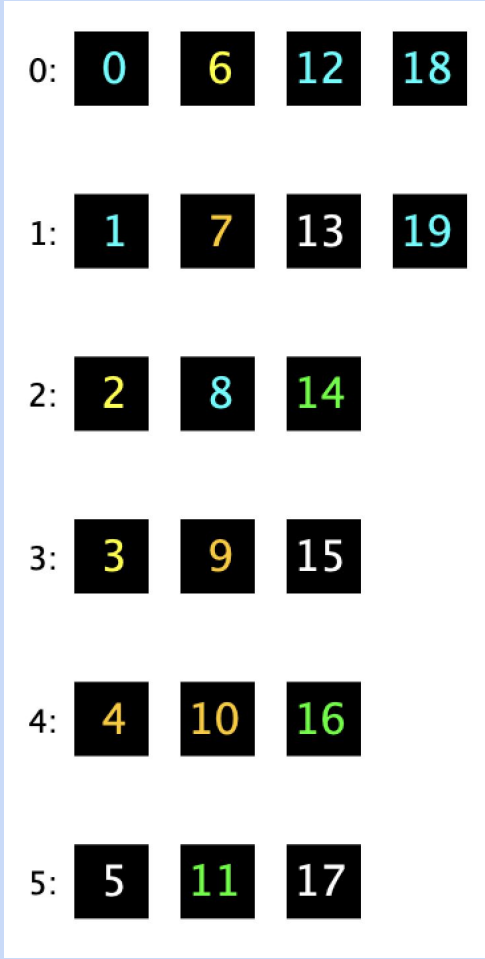
The resulting distribution is to the right.



Designing a Hash Function

What hash function will result in the distribution to the right?

```
private int num;
private Color color;
```



Designing a Hash Function

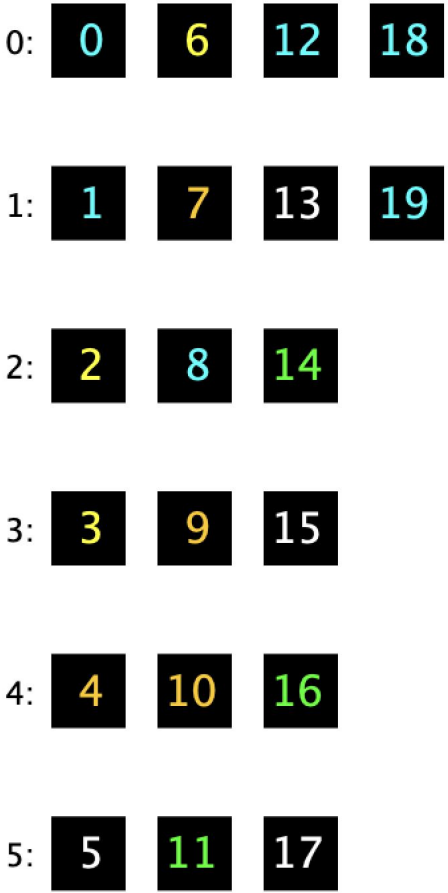
What hash function will result in the distribution to the right?

```
private int num;
private Color color;
```

Your answer?

- num % 6

Let's check it.



Other More Exotic Hash Functions

We can define whatever hash function we want:

- Leading digit.
- Sum of the digits.
- Length of the number.
- Something else?

Let’s try one. Which one do you want to try?

Why Custom Hash Functions?

Lecture 20, CS61B, Spring 2024

HashCode Comparison

We mentioned that the goal of a hash function is to try to spread items out evenly.

- No spread: Returning 0.
- Bad spread: Returning sum of the digits.
- Good spread: Returning num.

The Default hashCode: yellkey.com/part

We mentioned that the goal of a hash function is to try to spread items out evenly.

- No spread: Returning 0.
- Bad spread: Returning sum of the digits.
- Good spread: Returning num.

What do you think about the spread of the default hashCode, which returns the memory address?

A. No spread.
B. Bad spread.
C. Good spread.

The Default hashCode

We mentioned that the goal of a hash function is to try to spread items out evenly.

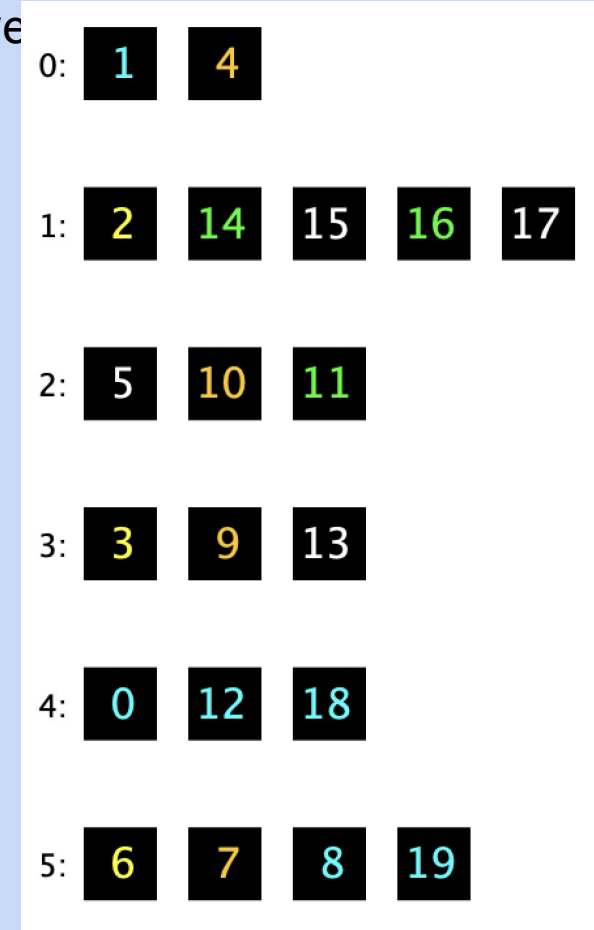
- No spread: Returning 0.
- Bad spread: Returning sum of the digits.
- Good spread: Returning num.

What do you think about the spread of the default hashCode, which returns the memory address?

A. No spread.
B. Bad spread.
C. **Good spread: The memory address is effectively random, so items should be evenly distributed.**

Hard Question

If the default hashCode achieves good spread, why do we even bother to create custom hash functions?



contains

Lecture 20, CS61B, Spring 2024

The equals Method for a ColoredNumber

Suppose the equals method for `ColoredNumber` is as below, i.e. two `ColoredNumbers` are equal if they have the same `num`.

- General principle: if two things are equal, they should act as if they are the same thing to outside observers

```
@Override
public boolean equals(Object o) {
    if (o instanceof ColoredNumber otherCn) {
        return this.num == otherCn.num;
    }
    return false;
}
```

HashSet Behavior

Suppose the equals method for `ColoredNumber` is on the previous slide, i.e. two `ColoredNumbers` are equal if they have the same num.

```
int N = 20;
HashSet<ColoredNumber> hs = new HashSet<>();
for (int i = 0; i < N; i += 1) {
   hs.add(new ColoredNumber(i));
}
```

Suppose we now check whether 12 is in the hash table.

```
ColoredNumber twelve = new ColoredNumber(12);
hs.contains(twelve); // returns ??
```

What do we expect to be returned by `contains`?

HashSet Behavior

Suppose the equals method for `ColoredNumber` is on the previous slide, i.e. two `ColoredNumbers` are equal if they have the same num.

```
int N = 20;
HashSet<ColoredNumber> hs = new HashSet<>();
for (int i = 0; i < N; i += 1) {
   hs.add(new ColoredNumber(i));
}
```

Suppose we now check whether 12 is in the hash table.

```
ColoredNumber twelve = new ColoredNumber(12);
hs.contains(twelve); // returns true
```

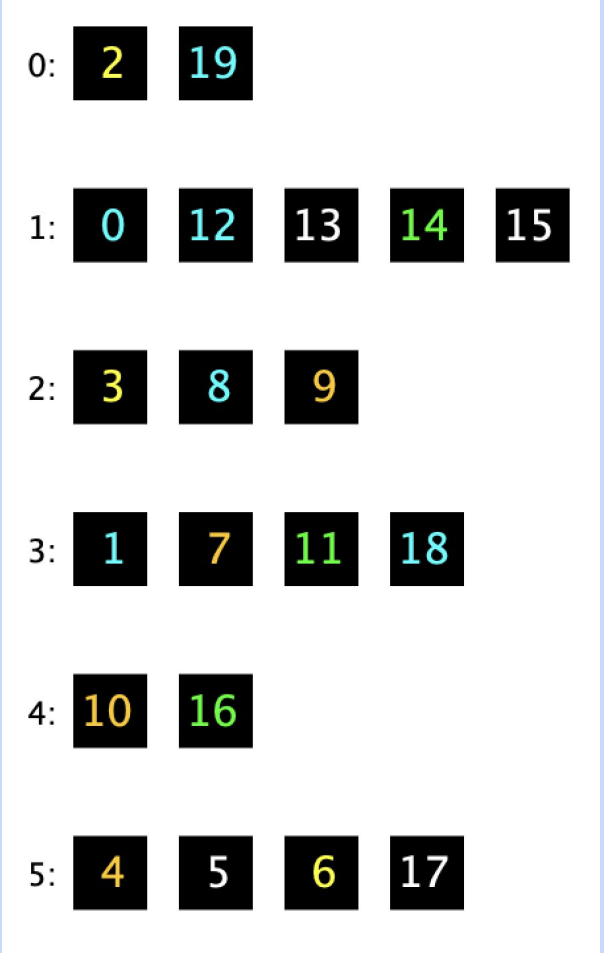
What do we expect to be returned by `contains`?

- We expect the contains call to be true, all 12s are created equal!

Finding an Item Using the Default HashCode

Suppose we are using the default hash function (uses memory address), which yields the table to the right.

```
int N = 20;
HashSet<ColoredNumber> hs = new HashSet<>();
for (int i = 0; i < N; i += 1) {
   hs.add(new ColoredNumber(i));
}
ColoredNumber twelve = new ColoredNumber(12);
hs.contains(twelve); // returns ??
```



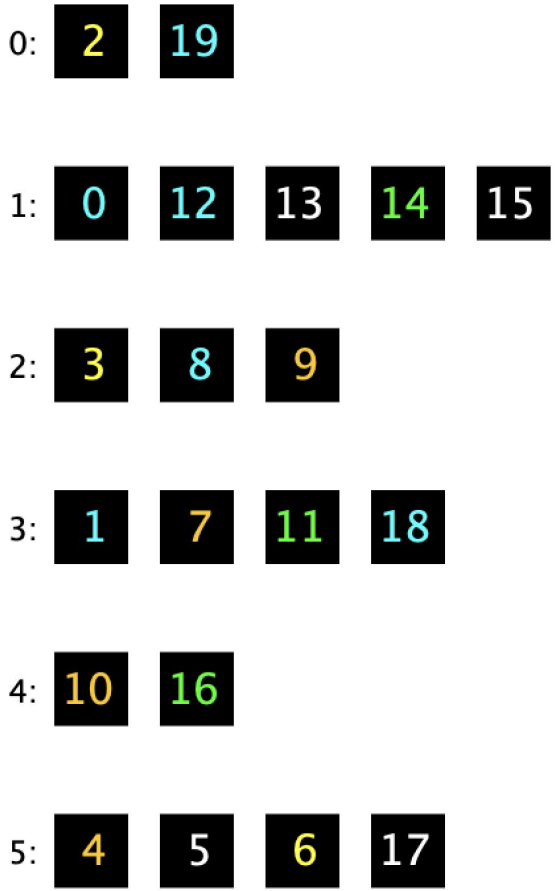
Suppose `equals` returns true if two `ColoredNumbers` have the same num (as on the previous slide).

What does the `contains` operation return. Why?

Finding an Item Using the Default HashCode

Suppose we are using the default hash function (uses memory address), which yields the table to the right.

```
int N = 20;
HashSet<ColoredNumber> hs = new HashSet<>();
for (int i = 0; i < N; i += 1) {
   hs.add(new ColoredNumber(i));
}
ColoredNumber twelve = new ColoredNumber(12);
hs.contains(twelve); // returns ??
```



Suppose `equals` returns true if two `ColoredNumbers` have the same num (as on the previous slide).

What does the `contains` operation return. Why?

- Returns false with probability 5/6ths.

Finding an Item Using the Default HashCode

hashCode: Based on memory address.

equals: Based on num.

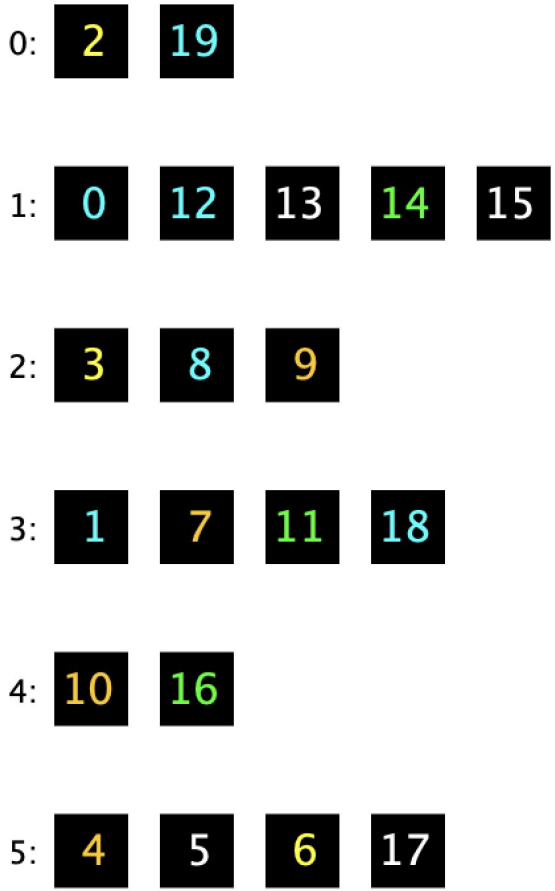
```
ColoredNumber twelve = new ColoredNumber(12);
hs.contains(twelve); // returns ??
```

There are two ColoredNumber objects with num = 12.

- One of them is in the HashSet.
- One of them was created by the code above.

Each memory address is random.

- Only 1/6th chance they % to the same bucket.



Finding an Item Using the Default HashCode

hashCode: Based on memory address.

equals: Based on num.

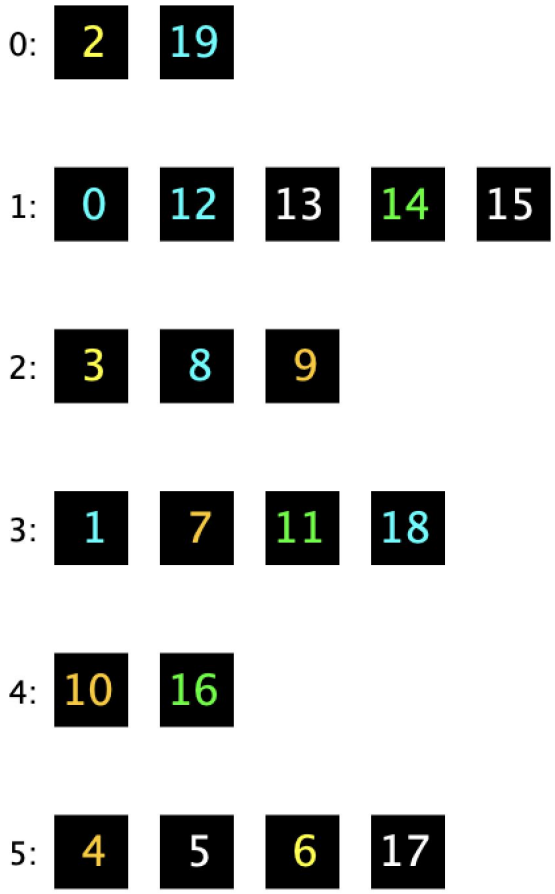
```
ColoredNumber twelve = new ColoredNumber(12);
hs.contains(twelve); // returns ??
```

There are two ColoredNumber objects with num = 12.

- One of them is in the HashSet.
- One of them was created by the code above.

Example: If object created by code above is in memory location 6000000, its hashCode % 6 is 0.

- HashSet looks in bucket zero, doesn't find 12.

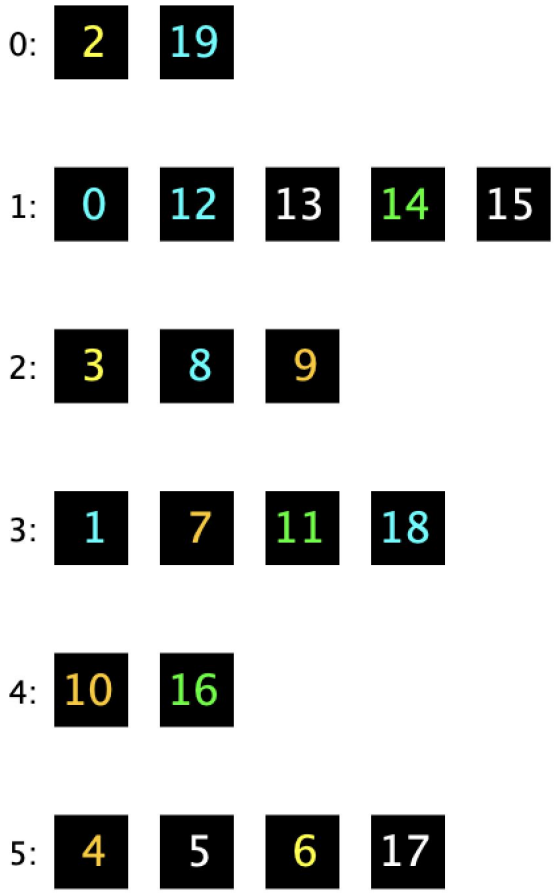


Hard Question

If the default hashCode achieves good spread, why do we even bother to create custom hash functions?

- Necessary to have consistency between `equals` and `hashCode` for basic operations to function.

Basic rule: If two objects are equal, they'd better have the same hashCode so the hash table can find it.



Duplicate Values

Lecture 20, CS61B, Spring 2024

Visualization for Some Basic Cases

hashCode and Equals

- Why Custom Hash Functions?
- contains
- **Duplicate Values**

The Danger of Mutable Keys

- Mutable vs. Immutable Types
- Mutable Hash Table Keys

A Peek into Java HashSets

Overriding Equals but Not HashCode: yellkey.com/wide

Suppose we have the same `equals` method (comparing `num`), but **we do not override hashCode**.

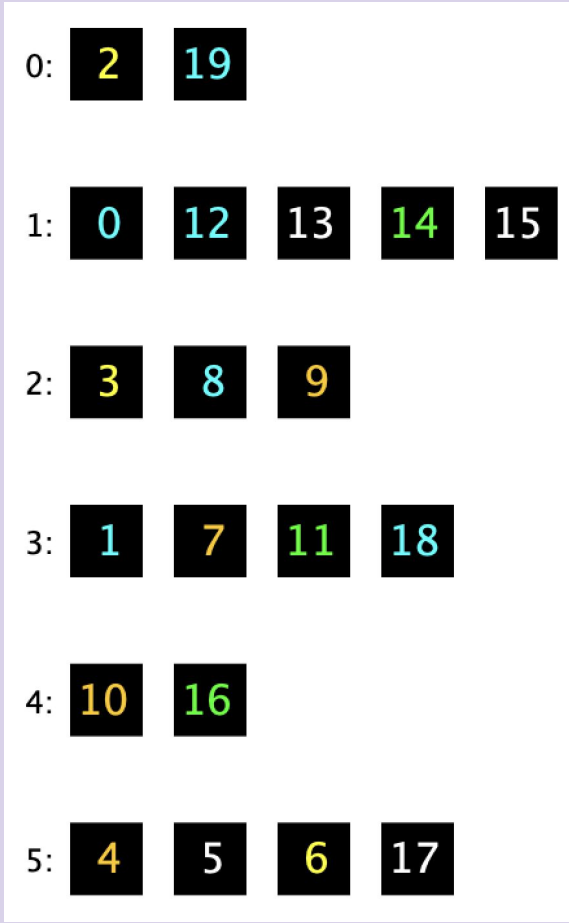
- Result of adding 0 through 19 is shown to the right.

```
public boolean equals(Object o) {
   ...  return this.num == otherCn.num; ...
}
```

```
ColoredNumber zero = new ColoredNumber(0);
hs.add(zero); // does another zero appear?
```

Which can happen when we call `add(zero)`?

A. We add another 0 to bin zero.
B. We add another 0 to bin one.
C. We add another 0 to some other bin.
D. We do not get a duplicate zero.

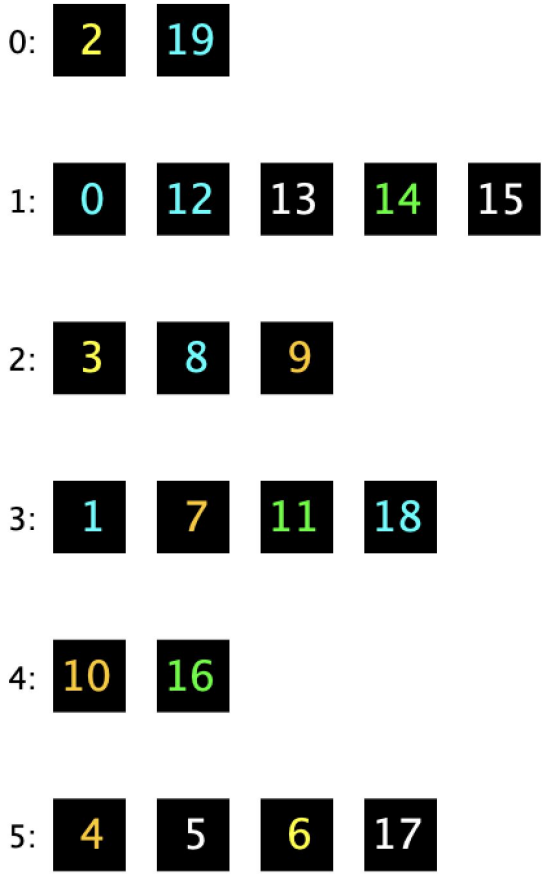


Overriding Equals but Not HashCode

Suppose we have the following `equals` method, but **we do not override hashCode**.

- Result of adding 0 through 19 is shown to the right.

```
@Override
public boolean equals(Object o) {
   if (o instanceof ColoredNumber otherCn) {
       return this.num == otherCn.num;
   }
   return false;
}
```



Which of the following can happen if we add 0 again?

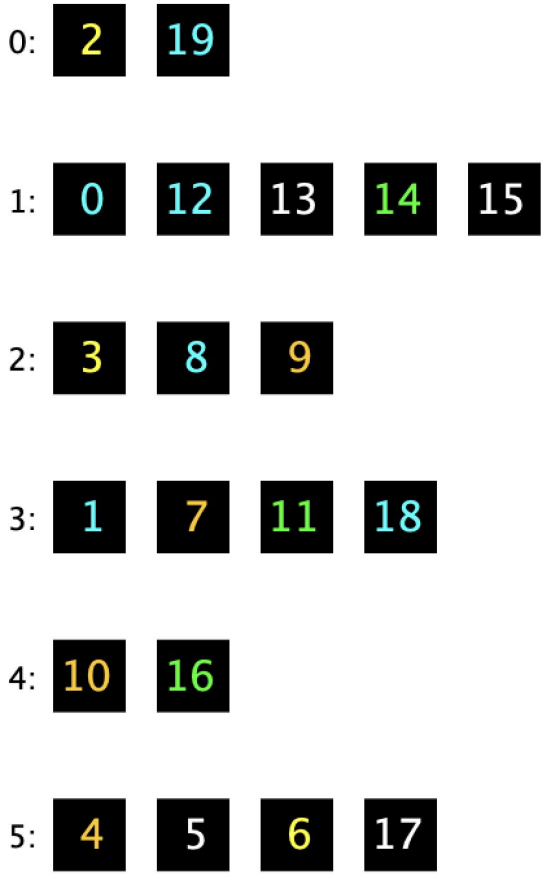
A. We add another 0 to bin zero.
B. We add another 0 to bin one.
C. We add another 0 to some other bin.
D. We do not get a duplicate zero.

Overriding Equals but Not HashCode

Suppose we have the following `equals` method, but **we do not override hashCode**.

- Result of adding 0 through 19 is shown to the right.

```
@Override
public boolean equals(Object o) {
   if (o instanceof ColoredNumber otherCn) {
       return this.num == otherCn.num;
   }
   return false;
}
```



Which of the following can happen if we add 0 again?

The new zero ends up in a random bin.

- 5/6ths chance: In bin 0, 2, 3, 4, or 5. Duplicate!
- 1/6 chance: In bin 1, no duplicate! (`equals` blocks it)

Takeaway: Equals and hashCode

Bottom line: If your class override `equals`, you should also override `hashCode` in a consistent manner.

- If two objects are `equals`, they must always have the same `hashCode`.

If you don’t everything breaks:

- Contains can’t find objects (unless it gets lucky).
- Add results in duplicates.

Mutable vs. Immutable Types

Lecture 20, CS61B, Spring 2024

Immutable Data Types

An immutable data type is one for which an instance cannot change in any observable way after instantiation.

Examples:

- Mutable: ArrayDeque, Percolation.
- Immutable: Integer, String, Date.

```
public class Date {
   public final int month;
   public final int day;
   public final int year;
   private boolean contrived = true;
   public Date(int m, int d, int y) {
       month = m; day = d; year = y;
   }
}
```

The ***final*** keyword will help the compiler ensure immutability.

- final variable means you may assign a value once (either in constructor of class or in initializer), but after it can never change.
- Final is neither sufficient nor necessary for a class to be immutable.

Which of the Classes Below Are Immutable?

Immutable: an instance cannot change in any observable way after instantiation.

```
public class Pebble {
  public int weight;
  public Pebble() {
    weight = 1;
  }
}
```

```
public class Rock {
  public final int weight;
  public Rock (int w) {
    weight = w;
  }
}
```

```
public class RocksBox {
  public final Rock[] rocks;
  public RocksBox (Rock[] rox) {
    rocks = rox;
  }
}
```

```
public class SecretRocksBox {
  private Rock[] rocks;
  public SecretRocksBox(Rock[] rox) {
    rocks = rox;
  }
}
```

Which of the Classes Below Are Immutable?

Immutable: an instance cannot change in any observable way after instantiation.

```
public class Pebble {
  public int weight;
  public Pebble() {
    weight = 1;
  }
}
```

Example mutation:

```
Pebble p = new Pebble();
p.weight = 2;
```

Which of the Classes Below Are Immutable?

Immutable: an instance cannot change in any observable way after instantiation.

```
public class Rock {
  public final int weight;
  public Rock (int w) {
    weight = w;
  }
}
```

No mutation possible.

Unless you use the special “Reflections” library which lets you disobey access modifiers.

Which of the Classes Below Are Immutable?

Immutable: an instance cannot change in any observable way after instantiation.

Example mutation:

```
Rock r1 = new Rock(10);
Rock r2 = new Rock(20);
Rock[] rox = {r1, r2};
RocksBox rb = new RocksBox(rox);
rb.rocks[1] = null;
```

```
public class RocksBox {
  public final Rock[] rocks;
  public RocksBox (Rock[] rox) {
    rocks = rox;
  }
}
```

Which of the Classes Below Are Immutable?

Immutable: an instance cannot change in any observable way after instantiation.

Example mutation:

```
Rock r1 = new Rock(10);
Rock r2 = new Rock(20);
Rock[] rox = {r1, r2};
SecretRocksBox rb = new SecretRocksBox(rox);
rox[0] = new Rock(-999);
```

```
public class SecretRocksBox {
  private Rock[] rocks;
  public SecretRocksBox(Rock[] rox) {
    rocks = rox;
  }
}
```

Which of the Classes Below Are Immutable?

Immutable: an instance cannot change in any observable way after instantiation.

```
public class Pebble {
  public int weight;
  public Pebble() {
    weight = 1;
  }
}
```

```
public class Rock {
  public final int weight;
  public Rock (int w) {
    weight = w;
  }
}
```

```
public class RocksBox {
  public final Rock[] rocks;
  public RocksBox (Rock[] rox) {
    rocks = rox;
  }
}
```

```
public class SecretRocksBox {
  private Rock[] rocks;
  public SecretRocksBox(Rock[] rox) {
    rocks = rox;
  }
}
```

How Would We Make SecretRocksBox Immutable?

Immutable: an instance cannot change in any observable way after instantiation.

Example mutation:

```
Rock r1 = new Rock(10);
Rock r2 = new Rock(20);
Rock[] rox = {r1, r2};
SecretRocksBox rb = new SecretRocksBox(rox);
rox[0] = new Rock(-999);
```

```
public class SecretRocksBox {
  private Rock[] rocks;
  public SecretRocksBox(Rock[] rox) {
    rocks = rox;
  }
}
```

How Would We Make SecretRocksBox Immutable?

To make SecretRocksBox immutable, we can make our own copy of the array.

- Example mutation fails!

```
Rock r1 = new Rock(10);
Rock r2 = new Rock(20);
Rock[] rox = {r1, r2};
SecretRocksBox rb = new SecretRocksBox(rox);
rox[0] = new Rock(-999);
```

```
public class SecretRocks {
    private Rock[] rocks;
    public SecretRocks(Rock[] rox) {
        rocks = new Rock[rox.length];
        System.arraycopy(rox, 0,
                         rocks, 0,
                         rox.length);
    }
}
```

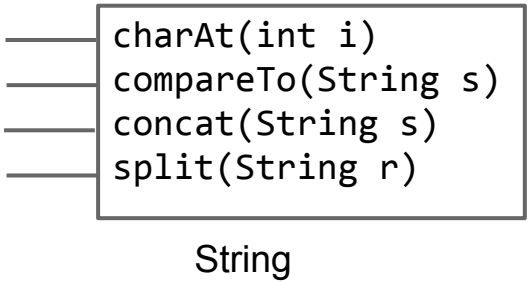
Immutability

Advantage: Less to think about: Avoids bugs and makes debugging easier.

- Analogy: Immutable classes have some buttons you can press / windows you can look inside. Results are ALWAYS the same, no matter what.

Disadvantage: Must create a new object anytime anything changes.

- Example: String concatenation is slow!



Mutable Hash Table Keys

Lecture 20, CS61B, Spring 2024

Mutable HashSet Keys

In principle, we can create a `HashSet<List>`.

Weird stuff happens if:

- We insert a `List` into a `HashSet`.
- Later mutate that `List`.

Example: hashCode

Consider an ArrayList equal to [0, 1].

- Such a list has hashCode 962 (can compute using code shown).

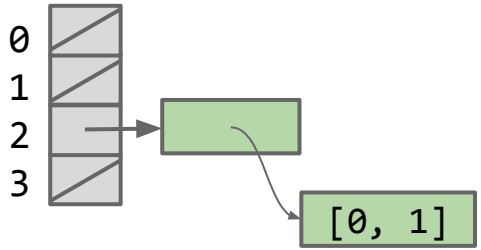
```
List<Integer> items = new ArrayList<>();
items.add(0);
items.add(1);
System.out.println(items.hashCode());
```

Example

Consider an ArrayList equal to [0, 1].

- Such a list has hashCode 962 (can compute using code shown).

If we add this list to a HashSet with 4 buckets, it lands in bucket 2 (962 % 4 = 2).



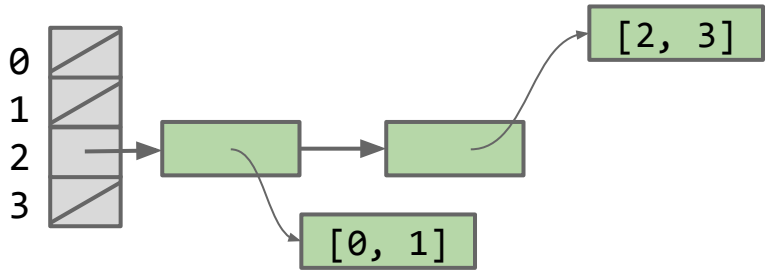

```
List<Integer> items = new ArrayList<>();
items.add(0);
items.add(1);
System.out.println(items.hashCode());

HashSet<List<Integer>> hs = new HashSet<>();
hs.add(items);
```

Example

First, we added [0, 1], which had hashCode 962, and landed in bucket 2.

- Suppose we now add the list [2, 3]. This list has hashCode 1026, which also lands in bucket 2.

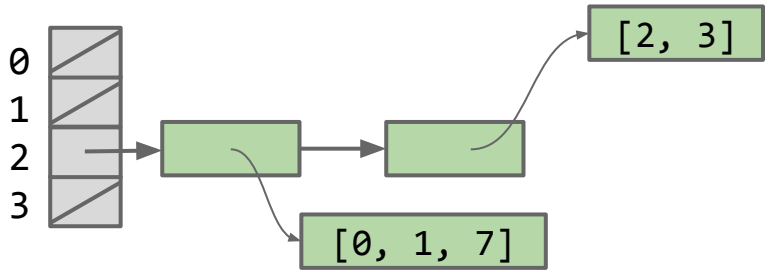



```
List<Integer> items = new ArrayList<>();
items.add(0);
items.add(1);
HashSet<List<Integer>> hs = new HashSet<>();
hs.add(items);
hs.add(List.of(2, 3));
```

Example

Suppose we add [0, 1], then [2, 3].

Now suppose we add the number 7 to items.




```
List<Integer> items = new ArrayList<>();
items.add(0);
items.add(1);
HashSet<List<Integer>> hs = new HashSet<>();
hs.add(items);
hs.add(List.of(2, 3));
items.add(7);
```

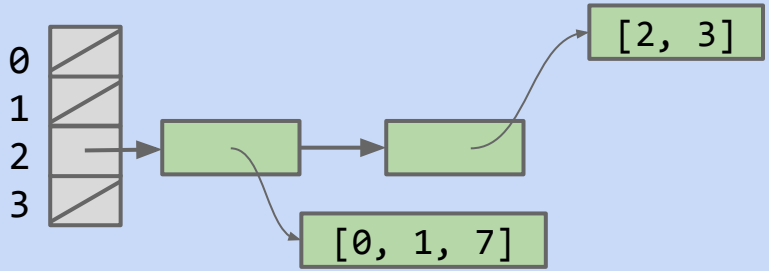
Example

Suppose we add [0, 1], then [2, 3].

- Now suppose we add the number 7 to items.

The hashCode of a list with [0, 1, 7] is 29829.

- What could go wrong?
- What method call will fail?




```
List<Integer> items = new ArrayList<>();
items.add(0);
items.add(1);
HashSet<List<Integer>> hs = new HashSet<>();
hs.add(items);
hs.add(List.of(2, 3));
items.add(7);
```

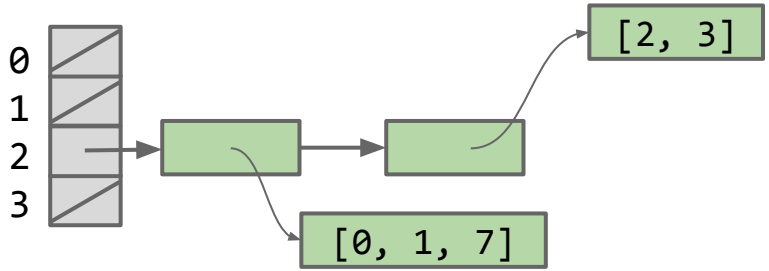
Example

Suppose we add [0, 1], then [2, 3].

- Now suppose we add the number 7 to items.

The hashCode of a list with [0, 1, 7] is 29829.

- What could go wrong?
- What method call will fail?
  - contains(items)




```
List<Integer> items = new ArrayList<>();
items.add(0);
items.add(1);
HashSet<List<Integer>> hs = new HashSet<>();
hs.add(items);
hs.add(List.of(2, 3));
items.add(7);
System.out.println(hs.contains(items));
```

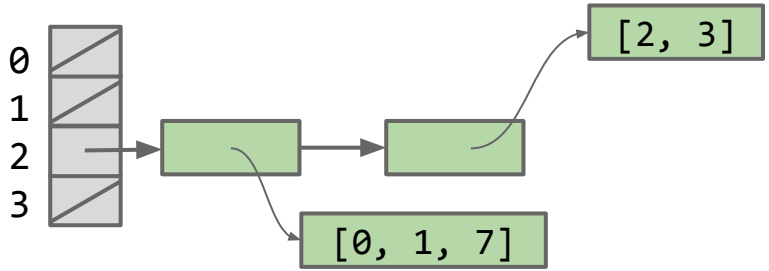
Example

Suppose we add [0, 1], then [2, 3].

- Now suppose we add the number 7 to items.
- The hashCode of a list with [0, 1, 7] is 29829.

If we call contains(items), we have a problem.

- hashCode of items is 29829 % 4 = 1.
- Hash table looks in bucket 1, empty!

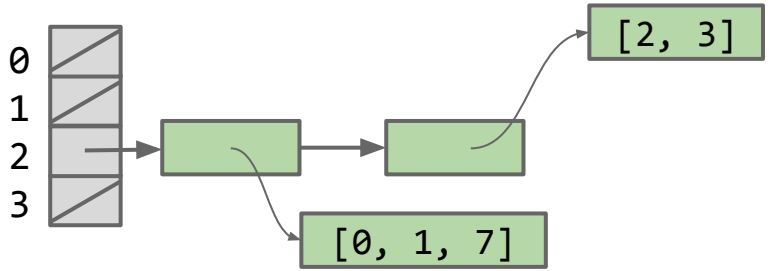



```
List<Integer> items = new ArrayList<>();
items.add(0);
items.add(1);
HashSet<List<Integer>> hs = new HashSet<>();
hs.add(items);
hs.add(List.of(2, 3));
items.add(7);
System.out.println(hs.contains(items));
```

Don’t Mutate Keys

Bottom line: Never mutate an Object being used as a key.

- Incorrect results, item gets lost.




```
List<Integer> items = new ArrayList<>();
items.add(0);
items.add(1);
HashSet<List<Integer>> hs = new HashSet<>();
hs.add(items);
hs.add(List.of(2, 3));
items.add(7);
System.out.println(hs.contains(items));
```

A Peek into Java HashSets

Lecture 20, CS61B, Spring 2024

A Peek into Java HashSets

HashSets in Java

We can look at the code that implements the HashSet in Java:

- https://github.com/openjdk/jdk/blob/master/src/java.base/share/classes/java/util/HashSet.java

It simply delegates all of its work to a `HashMap<K, Object>` and ignores the value.

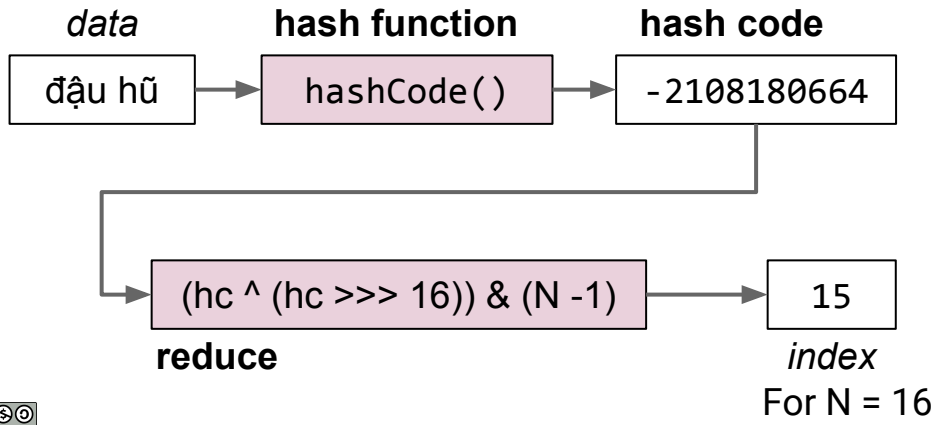
HashSets in Java

We can then look at the code that implements the HashMap in Java:

- https://github.com/openjdk/jdk/blob/master/src/java.base/share/classes/java/util/HashMap.java

Reading the code, we can see that:

- Hash table starts at size 16, then doubles every time N exceeds load factor which defaults to 0.75.
- The reduce function is a bit complicated. Come ask me at OH if you're curious.



HashSets in Java

Josh wrote a class called HashSet probe that uses the reflections library to print out the size of the array holding the buckets.

HashSets in Java

Josh wrote a class called HashSet probe that uses the reflections library to print out the size of the array holding the buckets.

- N/M is never more than 0.75.

```
Resize occurred, N = 13, hash table array size = 32
Resize occurred, N = 25, hash table array size = 64
Resize occurred, N = 49, hash table array size = 128
Resize occurred, N = 97, hash table array size = 256
Resize occurred, N = 193, hash table array size = 512
Resize occurred, N = 385, hash table array size = 1024
Resize occurred, N = 769, hash table array size = 2048
Resize occurred, N = 1537, hash table array size = 4096
```

Visualizing a Hash Table With Load Factor 0.75.

Let’s run a simulation to see what happens if the load factor is kept to 0.75 or less.

Visualizing a Hash Table With Load Factor 0.75.

Let's run a simulation to see what happens if the load factor is kept to 0.75 or less.

Longest bucket in the simulation looks like it's around ~5.

- In CS70, you'll derive a precise mathematical characterization of the length of the longest bucket.
  - This is the so called "balls into bins" problem.
  - https://en.wikipedia.org/wiki/Balls_into_bins_problem
  - Can use this math to show the worst case runtime is a bit worse than constant.

Another Interesting Optimization

If we ctrl-F for “red-black” we find that that if a bin gets too full, it is converted into a red-black tree!

- “This map usually acts as a binned (bucketed) hash table, but when bins get too large, they are transformed into bins of TreeNodes, each structured similarly to those in java.util.TreeMap. Most methods try to use normal bins, but relay to TreeNode methods when applicable (simply by checking instanceof a node). Bins of TreeNodes may be traversed and used like any others, but additionally support faster lookup when overpopulated. However, since the vast majority of bins in normal use are not overpopulated, checking for existence of tree bins may be delayed in the course of table methods.”
- This is well beyond the scope of our course.

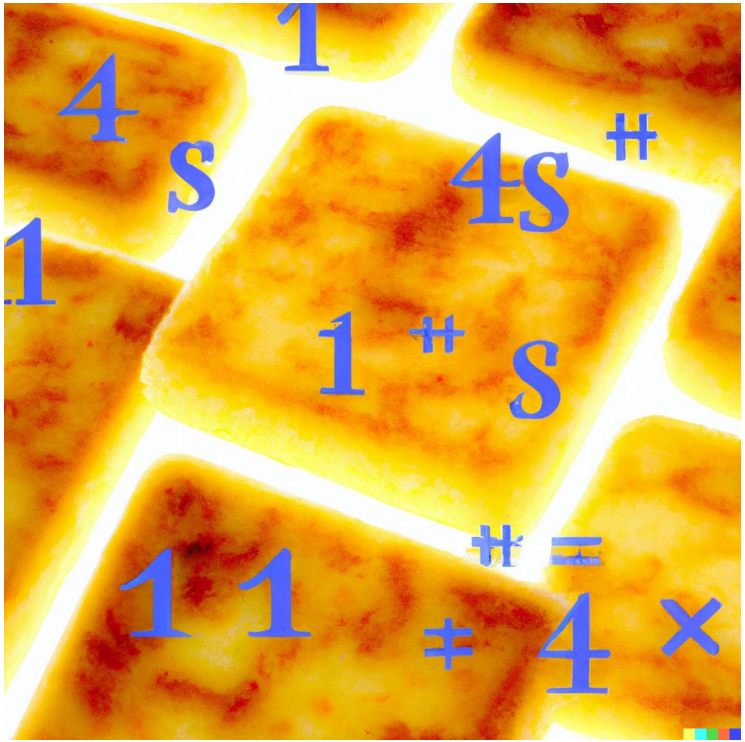
“The most useful algorithms are, unfortunately, not always the most beautiful.”

-Josh Hug

Citations

Hashbrowns in Cyberspace by Dall-E.

- Josh conjectures that H thing is the hashbrown key.



FAQ

What is the distinction between hash set, hash map, and hash table?

A hash set is an implementation of the Set ADT using the “hash table” as its engine.

A hash map is an implementation of the Map ADT using the “hash table” as its engine.

A “hash table” is a way of storing information, where you have M buckets that store N items. Each item has a “hashCode” that tells you which of M buckets to put that item in.