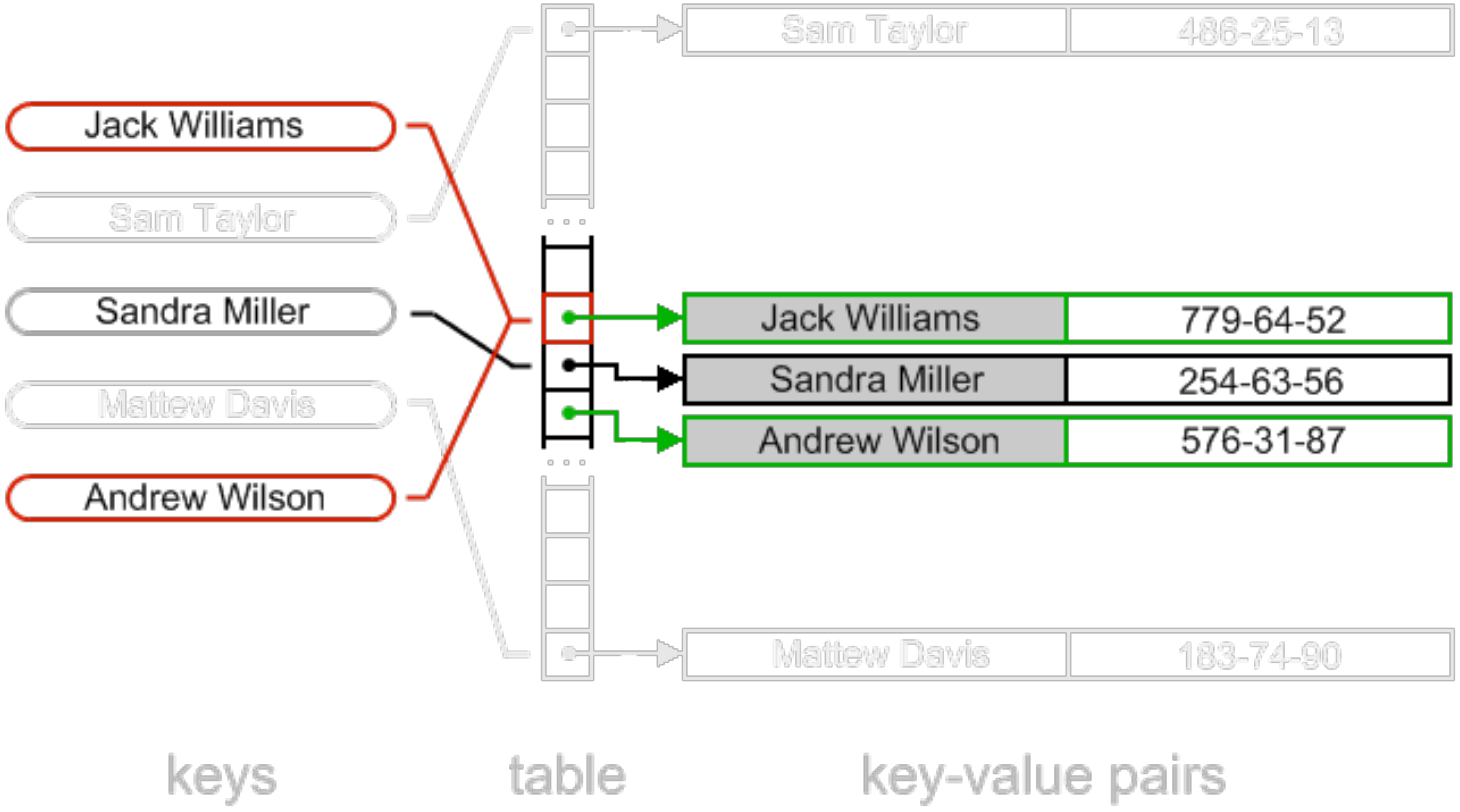
CS62 Class 21: Hash Tables (pt II)

Searching

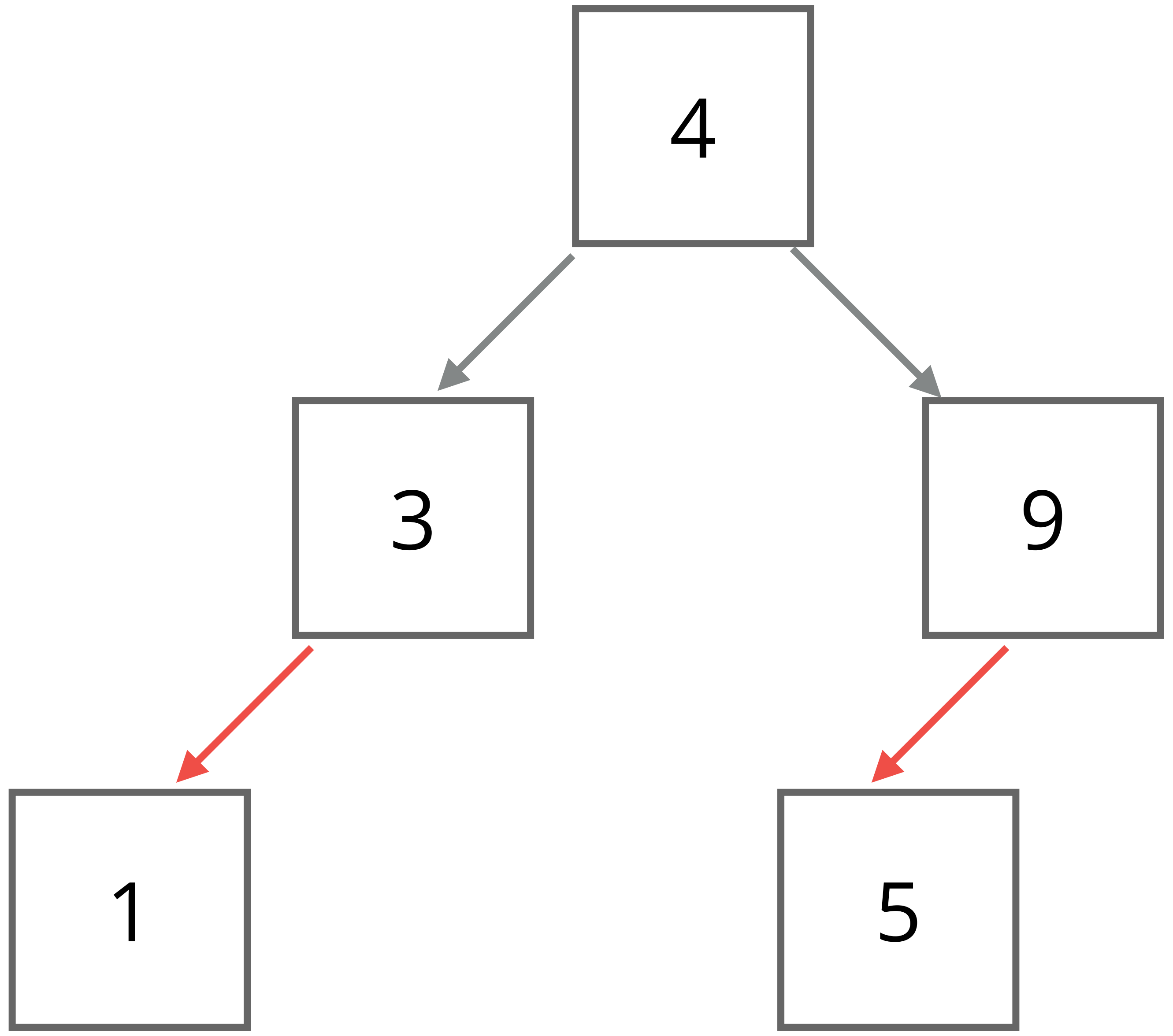


Open addressing hash table main idea: If there's a collision, just go to the next available bucket. No linked lists

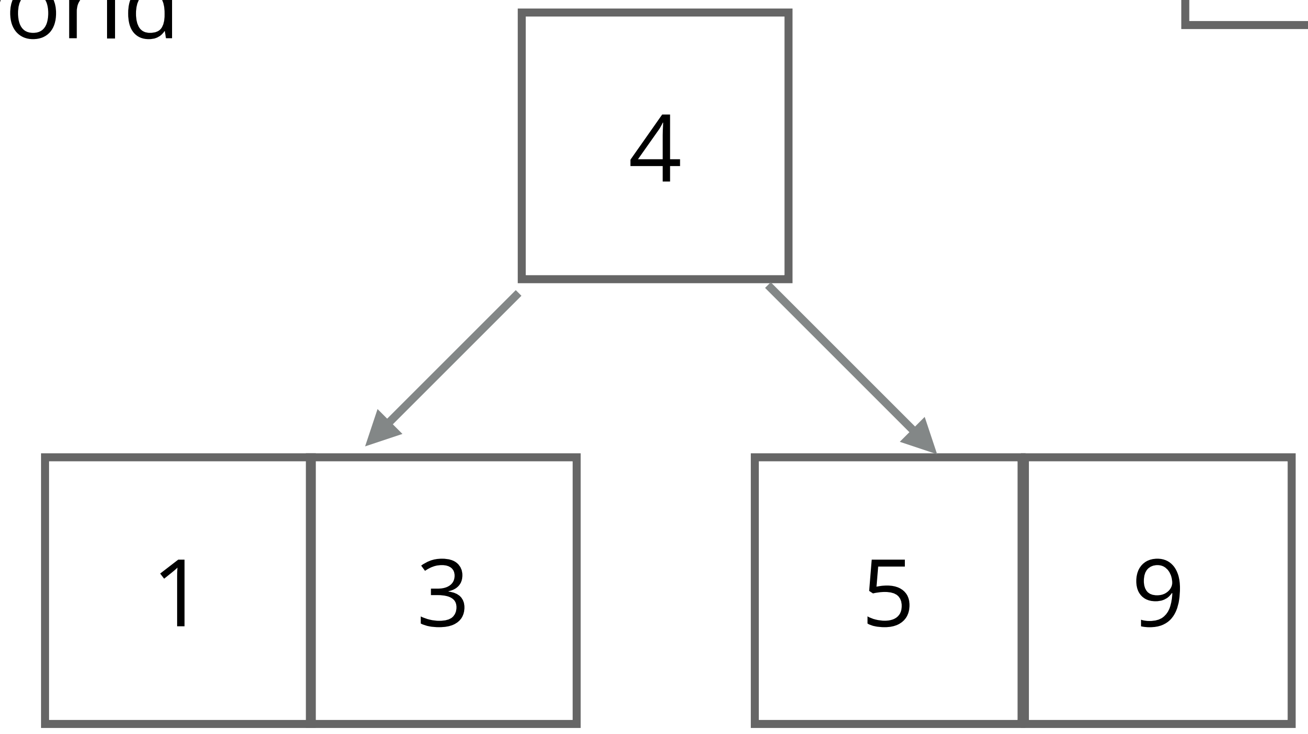
LLRB tree review

Insert the items 4, 5, 1, 9, 3 into a LLRB. (Hint: Draw it as a 2-3 tree).

LLRB world



2-3 world



Agenda

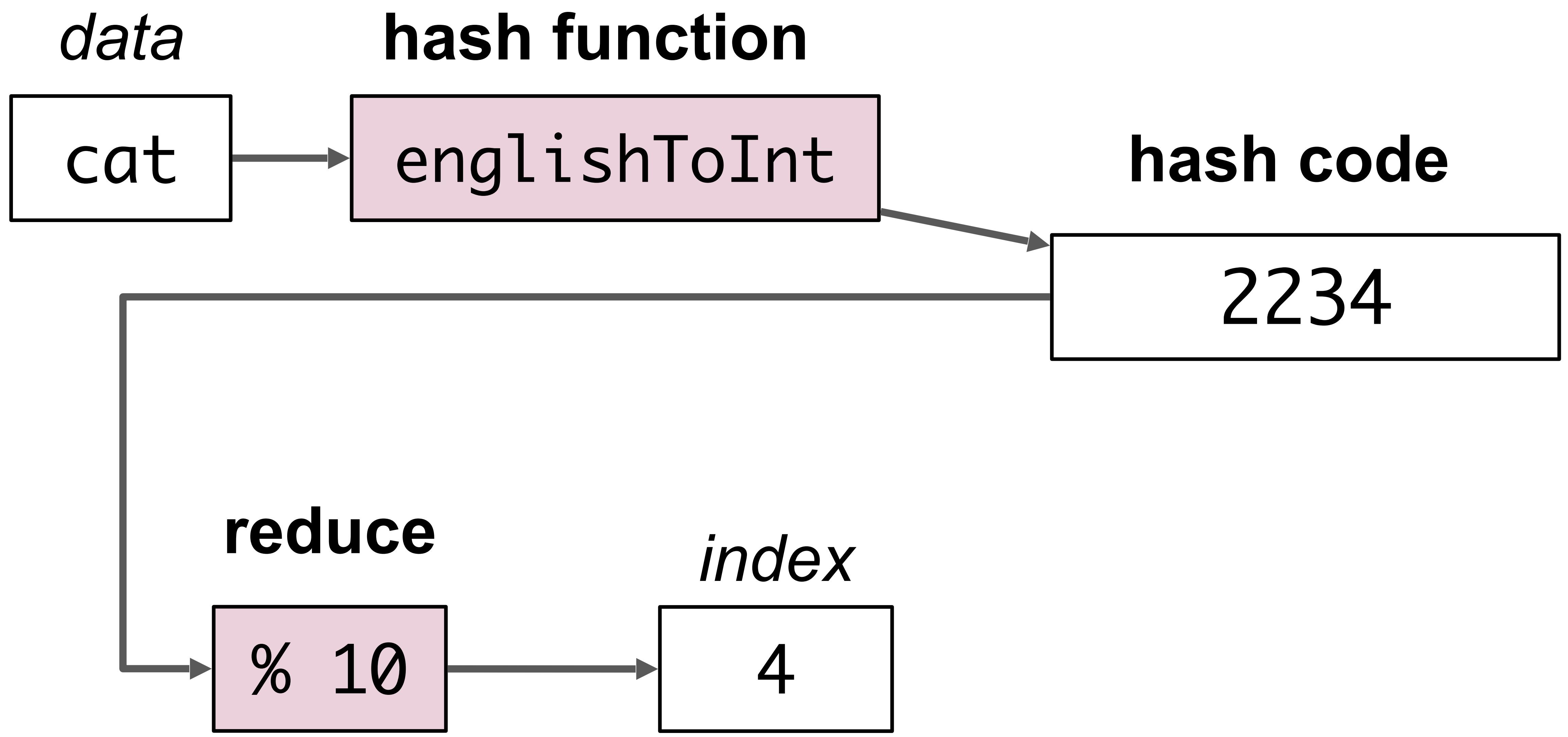
- Separate chaining review & analysis
- Open Addressing (linear & quadratic probing)
- More about equals and hashcodes
- Hashtables in Java

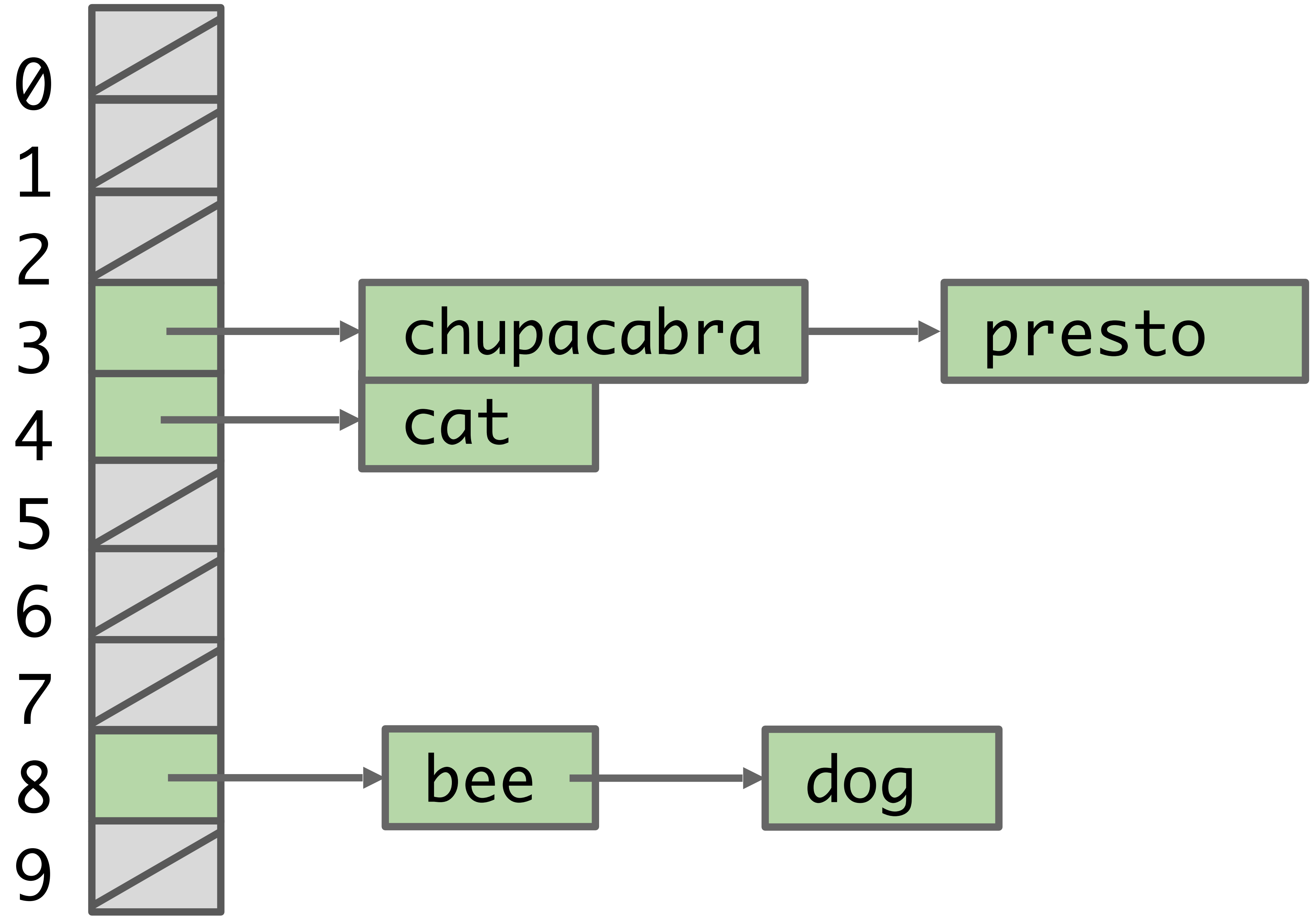
Last time: separate chaining hashtables

Separate Chaining Hash Table: review

A separate chaining hash table has *M* buckets which contain linked lists that store *N* data.

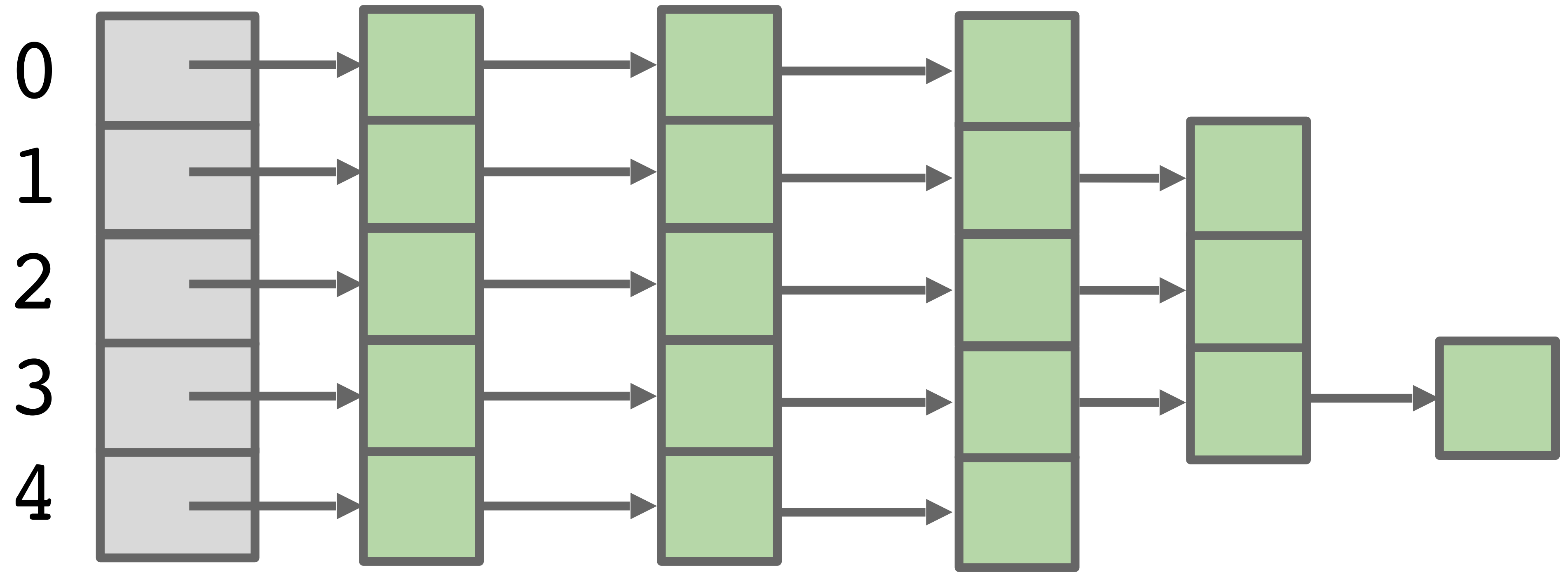
- *Data* is converted by a hash function into an integer representation called a hash code.
- The hash code is then reduced to a valid *index*, and data is stored in that bucket at that index.
- *Resize* when *load factor* N/M exceeds some constant.
- If items are spread out nicely, Θ(1) average runtime.





A hash table!

Hash Table Runtime with No Resizing



N = 19 M = 5 N / M = 3.8

Suppose we have:

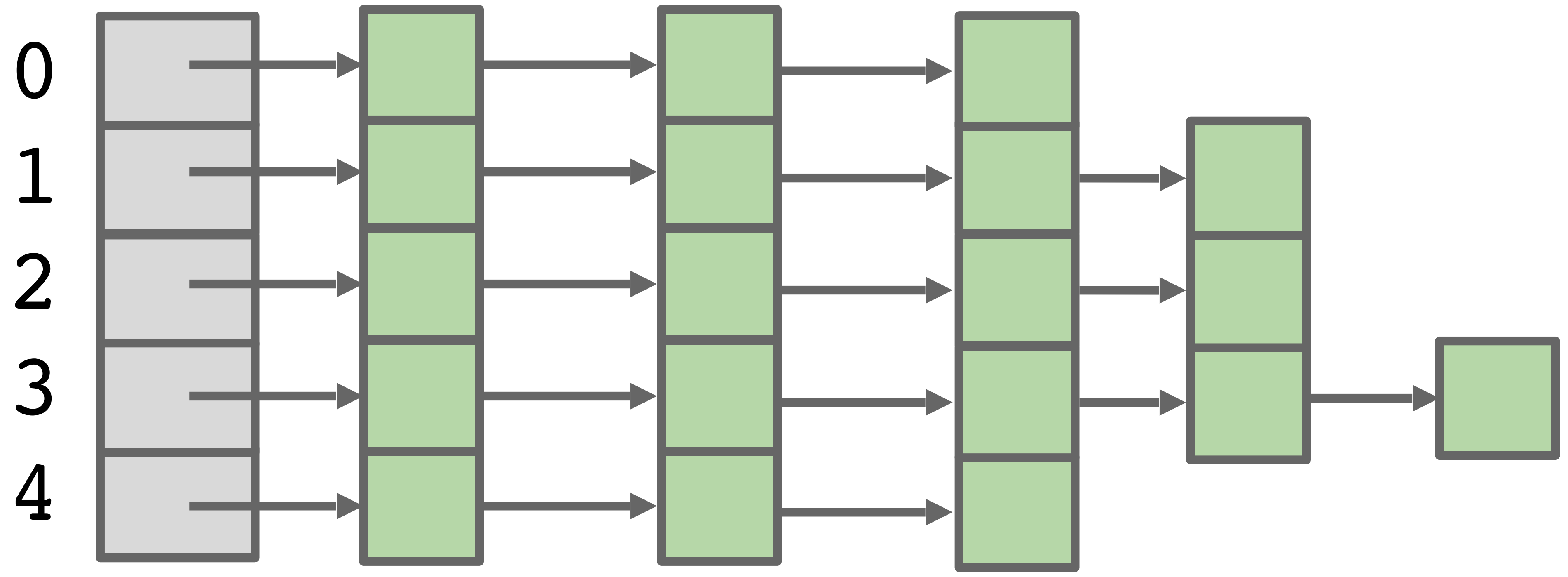
- An fixed number of buckets M.
- An increasing number of items N.

Average list is around N/M items

Even if items are spread out evenly, lists are of length Q = N/M.

- For M = 5, that means Q = Θ(N). Results in linear time operations.

Resizing Hash Table Runtime



N = 19   M = 5   N / M = 3.8

Suppose we have:

- An increasing number of buckets M.
- An increasing number of items N.

As long as M = Θ(N), then O(N/M) = O(1).

Assuming items are evenly distributed (as above), lists will be approximately N/M items long, resulting in Θ(N/M) runtimes.

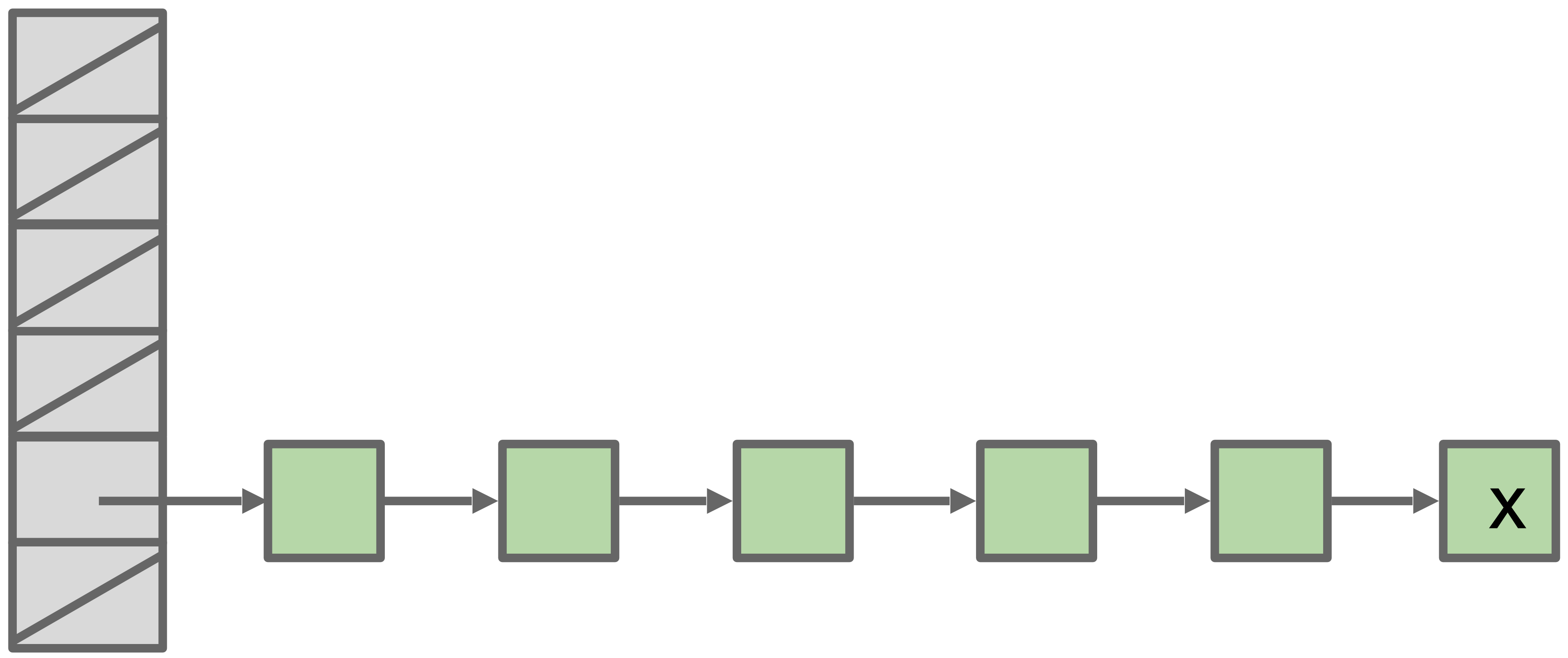
- By doubling every time N gets too big, we ensure that N/M = O(1).
- Thus, worst case runtime for all operations is Θ(N/M) = Θ(1).
  - ... unless that operation causes a resize.
    - If it causes a resize, what's the runtime of that specific operation?   O(N)
  - ... and again, we're assuming even distribution of items.

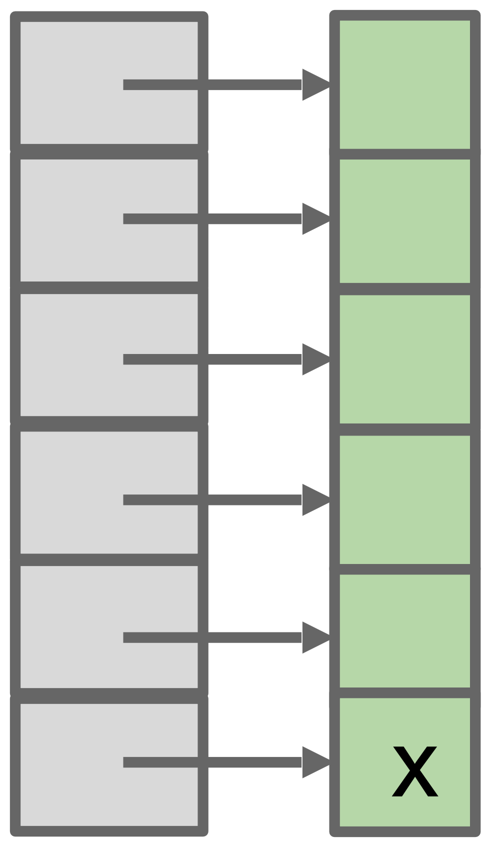
Regarding Even Distribution

Even distribution of item is critical for good hash table performance.

- Both tables below have load factor of N/M = 1.
- Left table is much worse!
  - Contains is Θ(N) for x.

How do we ensure an even distribution? A good scrambly hash function.





Uniform hashing assumption

- Uniform hashing assumption: Each key is equally likely to hash to an integer between 0 and $m - 1$.
- Mathematical model: balls & bins. Toss n balls uniformly at random into m bins.
- Bad news: Expect two balls in the same bin after ~$\sqrt{(\pi m/2)}$ tosses.
  - Birthday problem: In a random group of 23 or more people, more likely than not that two people will share the same birthday.
- Good news: load balancing
  - When $n \gg m$, the number of balls in each bin is “likely close” to n/m.

Going from hash code to hash value (index): avoiding negatives

- Hash code: an int between -2^{31} and $2^{31}-1$
- Hash value: an int between 0 and $m-1$, where m is the hash table size (typically a prime number/power of 2).
- The class that implements the dictionary of size m should implement a hash function. Examples:

```
private int hash (Key key){
    return key.hashCode() % m;
}
```

  - Bug! Might map to negative number (e.g., -1 % 16 = -1).

```
private int hash (Key key){
    return Math.abs(key.hashCode()) % m;
}
```

  - Very unlikely bug. For a hash code of -2^{31}, `Math.abs` will return a negative number!

```
private int hash (Key key){
    return (key.hashCode() & 0x7fffffff) % m;
}
```

  - Correct.

Parting thoughts about separate-chaining

- Deletion: Easy! Hash key, find its chain, search for a node that contains it and remove it.
- Ordered operations: not supported. Instead, look into (balanced) BSTs.
- Fastest and most widely used dictionary implementation for applications where key order is not important.

Open addressing

Open Addressing: An Alternate Strategy

Instead of using linked lists, an alternate strategy is “open addressing”.

- Map/set is stored as an **array of items**. Index tells you where to put the item.

If target location is already occupied, use a different location, e.g.

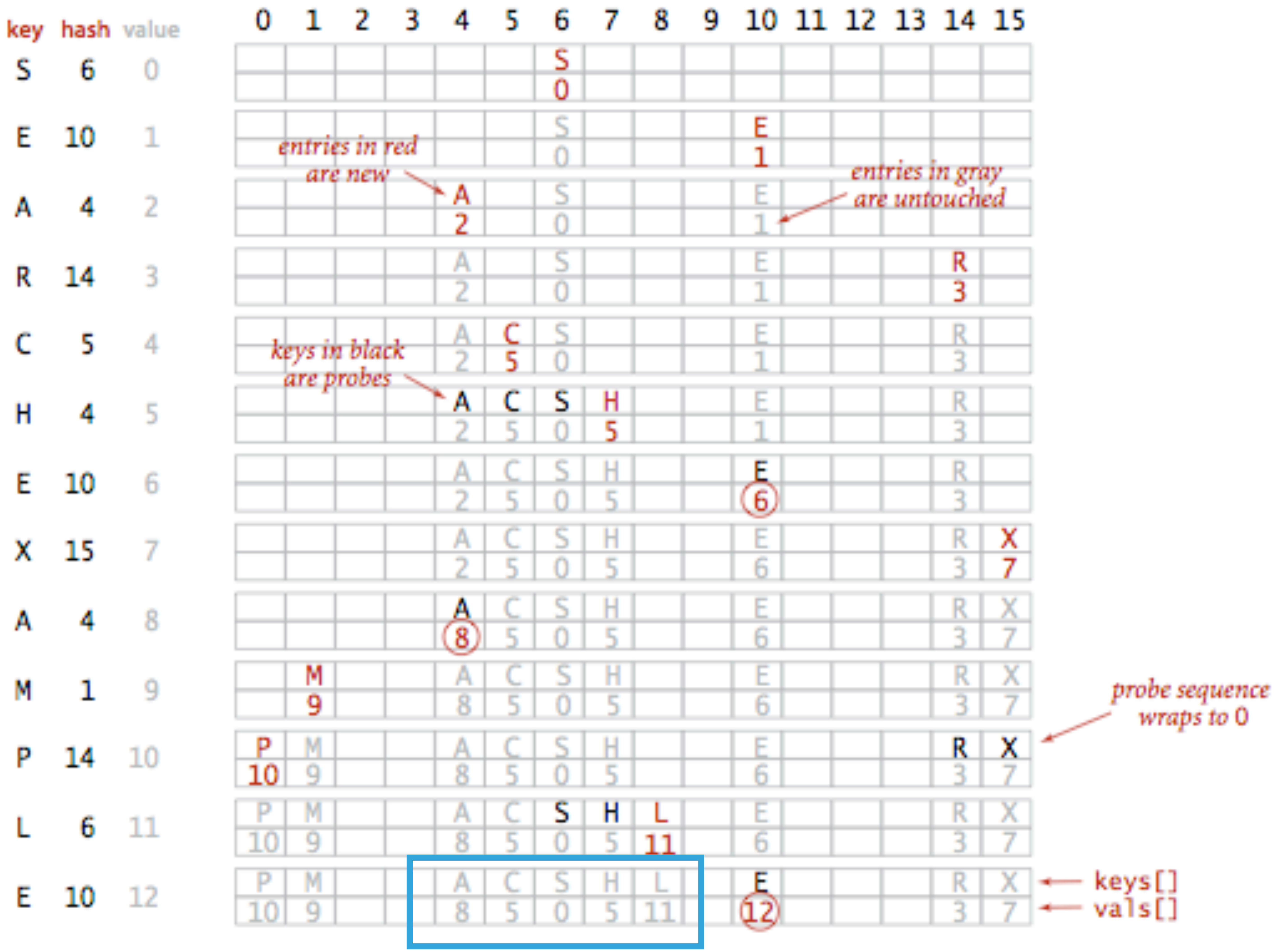
- **Linear probing:** Use next address, and if already occupied, just keep scanning one by one.
- **Quadratic probing**: Use next address, and if already occupied, try looking 4 ahead, then 9 ahead, then 16 ahead, ...

http://algs4.cs.princeton.edu

3.4 LINEAR PROBING DEMO

Linear Probing Example

- Hash: Map key to integer i between 0 and $m-1$.
- Insert: Put at index i if free. If not, try $i+1, i+2$, etc.
- Search: Search table index i. If occupied but no match, try $i+1, i+2$, etc
    - If you find a gap then you know that it does not exist.
- Table size m **must** be greater than the number of key-value pairs n.



Trace of linear-probing ST implementation for standard indexing client

primary clustering

Primary clustering

- Cluster: a contiguous block of keys.
- Observation: new keys likely to hash in middle of big clusters.
- Parameters:
  - m too large -> too many empty array entries.
  - m too small -> search time becomes too long.
  - Typical choice for load factor: $\alpha = n/m \sim 1/2$ -> constant time per operation.

Worksheet time!

- Assume a map implemented using hashing and linear probing for handling collisions.
- Let $m = 7$ be the hash table size.
- For simplicity, we will assume that keys are integers and that the hash value for each key k is calculated as $h(k) = k \,\%\, m$.
- Insert the key-value pairs (47, 0), (3, 1), (28, 2), (14, 3), (9,4), (47,5) and show the resulting map.

Worksheet answers

Key	Hash	Value	
47	5	0	
3	3	1	
28	0	2	
14	0	3	*collision*
9	2	4	
47	5	5	*updates value*

Keys	**28**	**14**	9	3		**47**	
Values	2	3	4	1		5	
Indices	0	1	2	3	4	5	6

Resizing in a linear probing hash table

- Goal: Load factor $n/m \leq 1/2$.
  - Double hash table size when $n/m \geq 1/2$.
  - Halve hash table size when $n/m \leq 1/8$.
  - Just like in separate chaining, need to rehash all keys when resizing (hash code does not change, but hash value changes as it depends on table size).
  - Deletion not straightforward.
    - Option 1: Delete but then re-insert everything in the cluster to close the gap
    - Option 2: Keep it in the table but flag it so it doesn't get searched for, and can be inserted over

Quadratic Probing

- Another open addressing technique that aims to reduce primary clustering by taking the original hash index and adding successive values of an arbitrary quadratic polynomial until an open slot is found.
- Modify the probe sequence so that $h(k, i) = (h(k) + c_1 i + c_2 i^2) \,\%\, m, c_2 \neq 0$, where i is the i-th time we have had a collision for the given index.
  - When $c_2 = 0$, then quadratic probing degrades to linear probing.
- Basically, at first collision, go to the next (1^2 = 1) slot in the array. If that's still a collision, go to the slot that's 2^2 = 4 away. If that's still a collision, go to the slot that's 3^2 = 9 away. If that's still a collision, go to the slot that's 4^2 = 16 away.
  - 1, 4, 9, 16, 25… away from the original hashed bucket

Quadratic probing – Example

- $h(k) = k \,\%\, m$ and $h(k, i) = (h(k) + i^2) \,\%\, m$.
- Assume $m = 13$, and key-value pairs to insert: (17,0), (33,1), (18,2), (20,3), (44,4), (11,5), (19,6), (7,7).

	0	1	2	3	4	5	6	7	8	9	10	11	12		
(17,0)					17										
(33,1)					17			33							
(18,2)					17	18		33							
(20,3)					17	18		33	20					Collision!	20 inserted 1 away
(44,4)					17	18	44	33	20					Collision!	44 inserted 1 away
(11,5)					17	18	44	33	20			11			
(19,6)					17	18	44	33	20		19	11		Collision!	19 inserted 4 away
(7,7)				7	17	18	44	33	20		19	11		Collision!	7 inserted 9 away

Summary for dictionary operations

	Worst case			Average case		
	Search	Insert	Delete	Search	Insert	Delete
BST	n	n	n	$\log n$	$\log n$	$\sqrt{n}$
balanced BST	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$
Separate chaining	n	n	n	1	1	1
Open addressing	n	n	n	1	1	1

Worst case when resizing hash table

All other operations

More about hashcode equality

ColoredNumbers

Let’s say we’re inserting ColoredNumber objects into a HashTable. 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.

Designing a Hash Function

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

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

A: Just num % size of table (6)

0: 0 6 12 18

1: 1 7 13 19

2: 2 8 14

3: 3 9 15

4: 4 10 16

5: 5 11 17

The default hash code

We mentioned that the goal of a hash function is to try to spread items out evenly. E.g., for an integer's .hashCode():

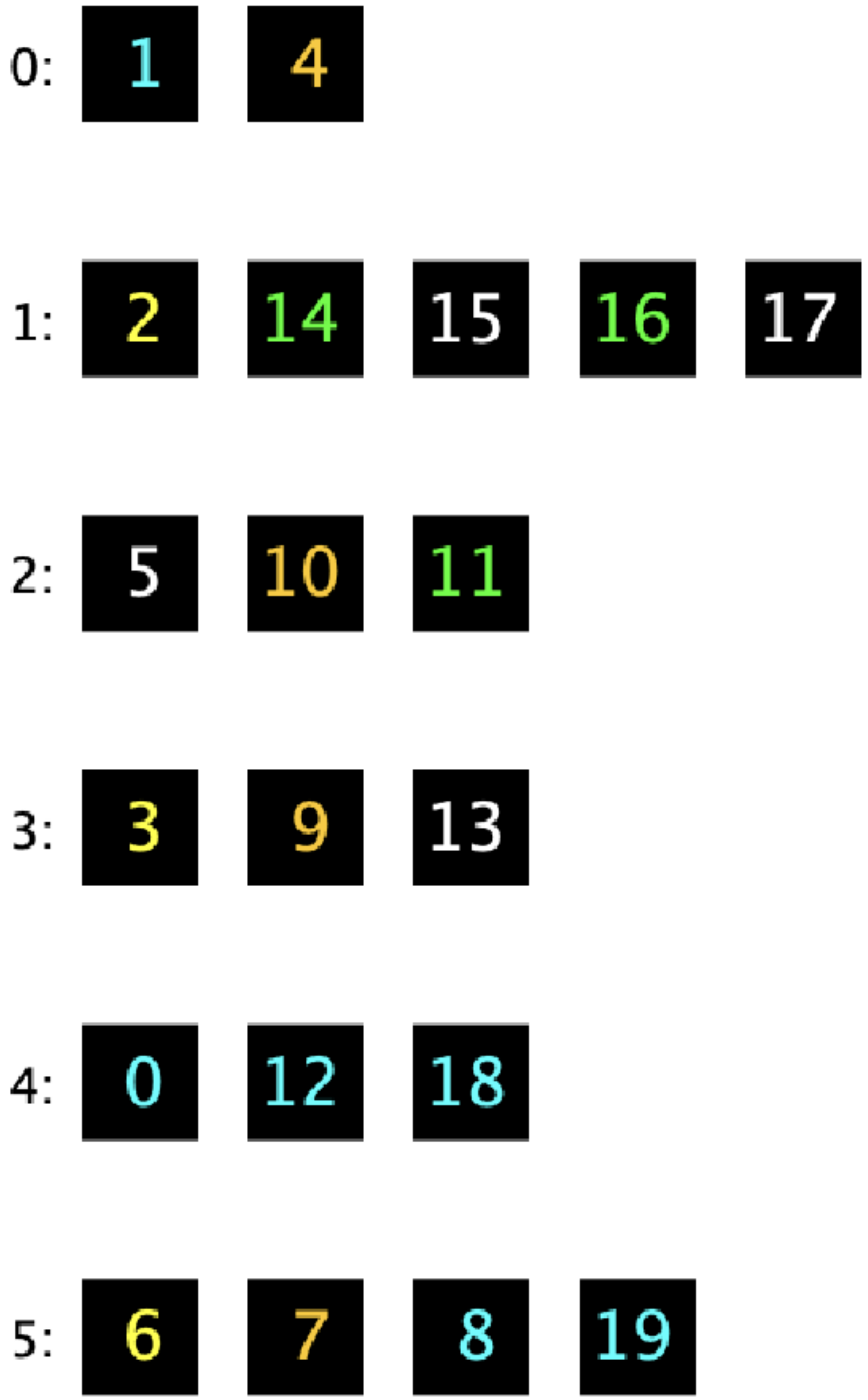
- No spread: Returning 0.
- Bad spread: Returning sum of its digits.
- Good spread: Returning the number itself.

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.

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



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.getClass() == this.getClass()) {
       return this.num == otherCn.num;
   }
   return false;
}
```

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 ??
```

0:	2	19			
1:	0	12	13	14	15
2:	3	8	9		
3:	1	7	11	18	
4:	10	16			
5:	4	5	6	17	

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

What does the `contains` operation return? (Note: contains() calls equals())

False with 5/6th probability

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.

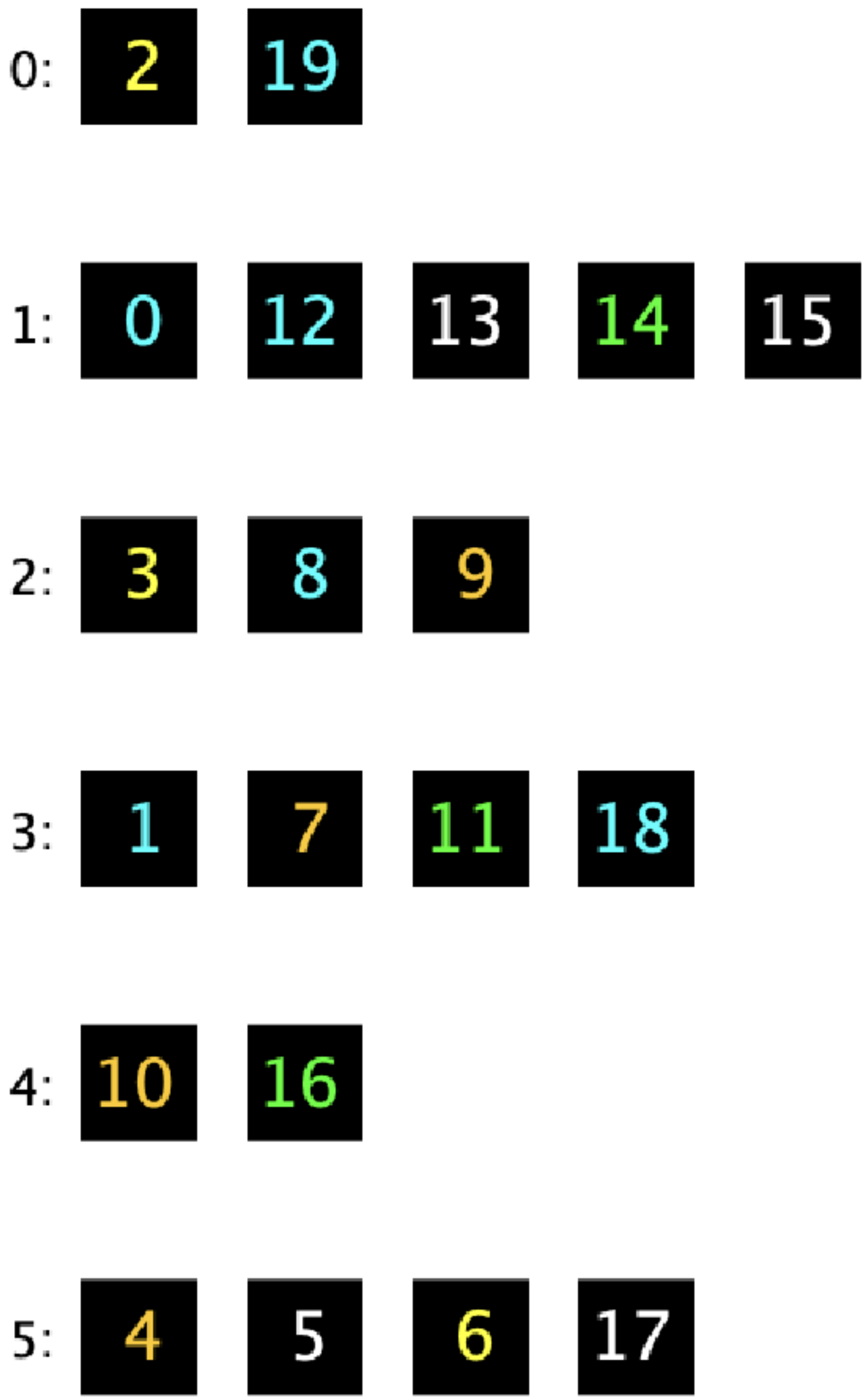
- One of them is in the HashSet.
- One of them was created by the code above.

Each memory address is random.

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

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 (in bucket 1).

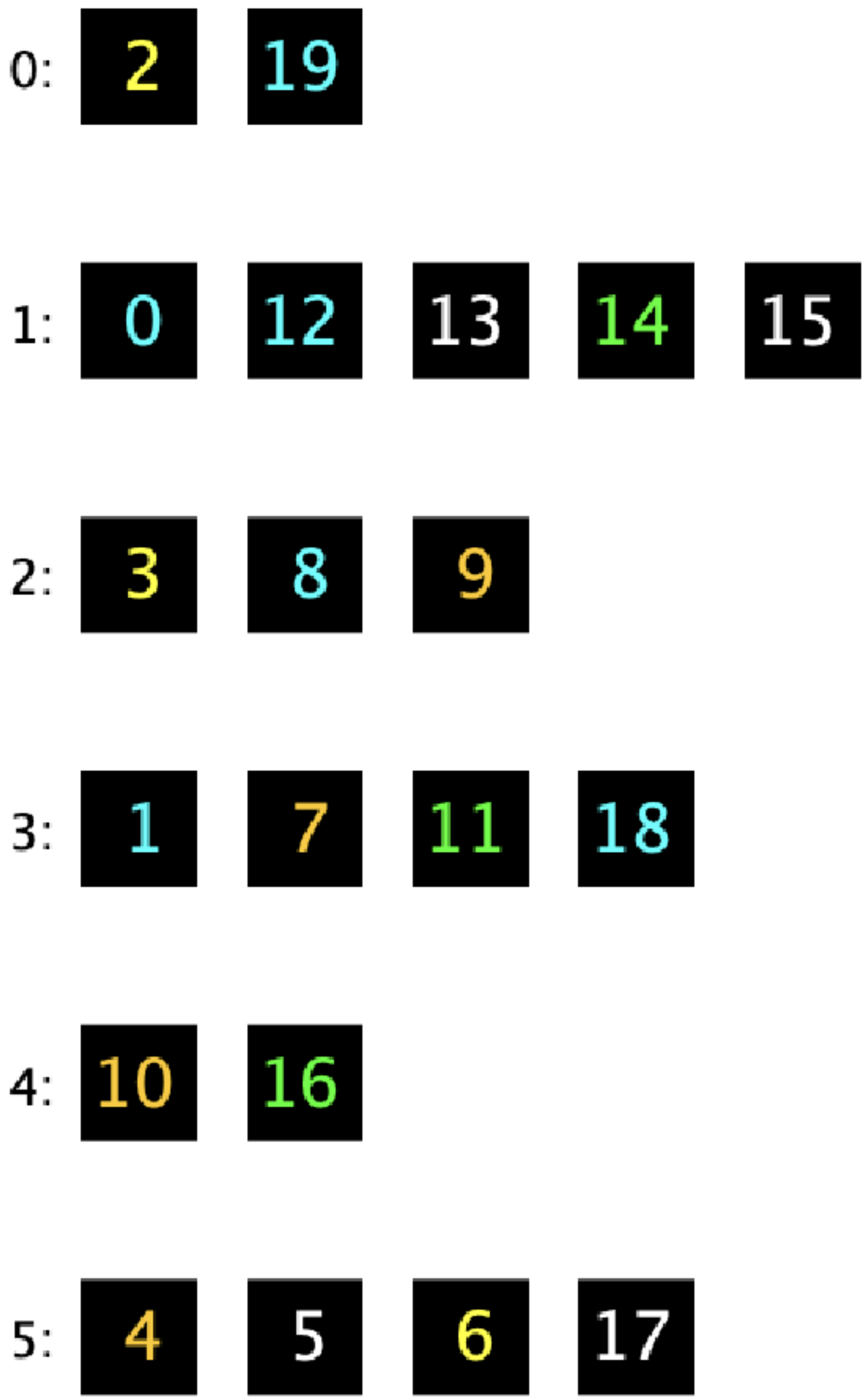


Consistency between equals and hashCode

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.



Worksheet time!

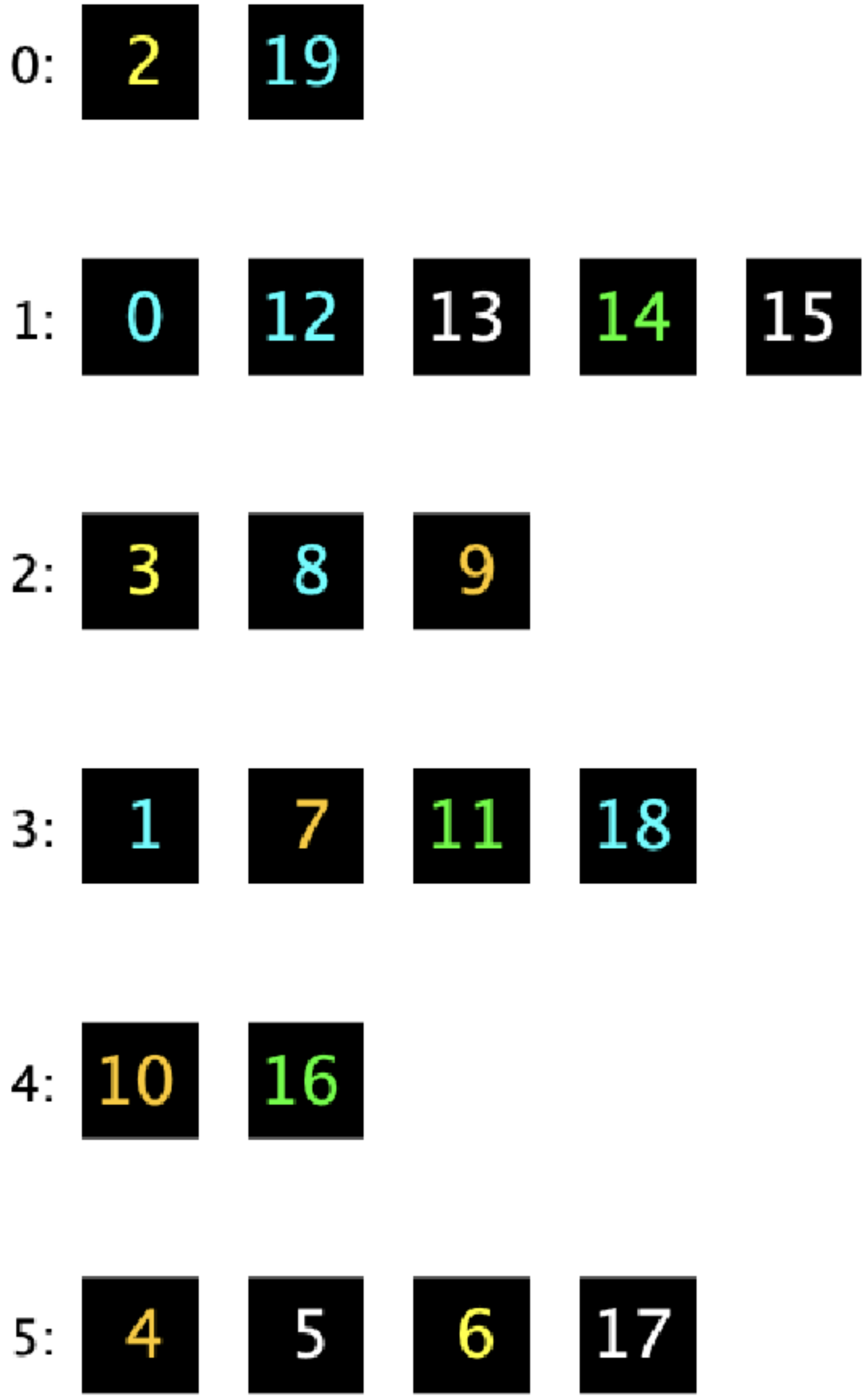
Suppose we have the same equals method (comparing num), but we do not override hashCode.

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

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

What 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.



Worksheet answer

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

What 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.

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)

0: 2 19

1: 0 12 13 14 15

2: 3 8 9

3: 1 7 11 18

4: 10 16

5: 4 5 6 17

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.

Hash Tables in Java

The Ubiquity of Hash Tables

In Java, implemented as `java.util.HashMap` and `java.util.HashSet`.

- How does a HashMap know how to compute each object's hash code?
  - Good news: It's not "`implements Hashable`".
  - Instead, all objects in Java must implement a `.hashCode()` method.

Object Methods

All classes are hyponyms of `Object`.

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

Default implementation of `hashCode` for an Object returns its **memory address**.

.hashCode()

Hash Codes in Java: More specific types

Java's actual hashCode function for Strings below (code cleaned up slightly):

- "横田誠司" and "±EreWn" map to 839,611,422.

```
public int hashCode(String s) {
       int intRep = 0;
       for (int i = 0; i < s.length(); i += 1) {
        intRep = intRep * 31;
        intRep = intRep + s.charAt(i);
       }
       return intRep;
}
```

That is, the two calls below both return 839,611,422.

- "横田誠司".hashCode()
- "±EreWn".hashCode()

Note: for integers, the hashcode is just the integer value.

For booleans, true's hashcode is 1231 and false's is 1237. Why? They're both large prime numbers (primes = better to avoid collisions.)

Implementating of hashCode() for user-defined types

```
public class Date {
     private int month;
     private int day;
     private int year;
     …
     public int hashCode() {
          int hash = 1;
          hash = 31*hash + ((Integer) month).hashCode();
          hash = 31*hash + ((Integer) day).hashCode();
          hash = 31*hash + ((Integer) year).hashCode();
          return hash;
          //could be also written as
          //return Objects.hash(month, day, year);
     }
}
```

$\boxed{31x+y \text{ rule}}$

Why 31? It's a small prime to ensure all bits of all the fields play a role in creating the hash code.

General hash code recipe in Java

- Combine each significant field using the 31x+y rule.
- Shortcut 1: use `Objects.hash()` for all fields (except arrays).
- Shortcut 2: use `Arrays.hashCode()` for primitive arrays.
- Shortcut 3: use `Arrays.deepHashCode()` for object arrays.
  - But make sure the objects are immutable!

Avoid items getting lost

Warning #1: Never store objects that can change in a HashSet or HashMap!

- Such objects are also called “mutable” objects, e.g. they can change.
  - Example: You’d never want to make a HashSet<List<Integer>>.
- **If an object’s variables changes, then its hashCode changes. May result in items getting lost.**

Warning #2: Never override equals without also overriding hashCode.

- Can also lead to items getting lost and generally weird behavior.
- HashMaps and HashSets use **equals()** to determine if an item exists in a particular bucket. (Not recalling .hashCode()!) (We just did this example.)

.equals()

More on warning #2: if 2 objects are “equal”, they should have the same hashCode

- Requirement: If `x.equals(y)` then it should be `x.hashCode()==y.hashCode()`.
- Ideally (but not necessarily): If !`x.equals(y)` then it should be `x.hashCode()!=y.hashCode()`.
- Need to override *both* `equals()` and `hashCode()` for custom types.
  - Already done for us for `Integer`, `Double`, etc.

Equality test in Java

- Requirement: For any objects x, y, and `z`.
  - Reflexive: `x.equals(x)` is `true`.
  - Symmetric: `x.equals(y)` iff `y.equals(x)`.
  - Transitive: if `x.equals(y)` and `y.equals(z)` then `x.equals(z)`.
  - Non-null: if `x.equals(null)` is `false`.
- If you don't override it, the default implementation of .equals() checks whether x and y refer to **the same object in memory.**

Overriding `equals()` for user-defined types

```
public class Date {
    private int month;
    private int day;
    private int year;
    …
    public boolean equals(Object y) {
        if (y == this){ return true;}
        if (y == null){ return false;}
        if (y.getClass() != this.getClass()){ return false;}
        Date that = (Date) y;
        return (this.day == that.day &&
                this.month == that.month &&
                this.year == that.year);
    }
}
```

signature: public boolean equals(Object objToCompare)

same memory location

non-null requirement

same class

cast the obj to be compared as the same class

and compare specific attributes
(of primitive types)

General equality test recipe in Java: `x.equals(y)`

- Optimization for reference equality.
  - `if (y == this) {return true;}`
- Check against null.
  - `if (y == null) {return false;}`
- Check that two objects are of the same type.
  - `if (y.getClass() != this.getClass()) {return false;}`
- Cast them.
  - `Date that = (Date) y;`
- Compare each significant field (i.e. instance variable).
  - `return (this.day == that.day && this.month == that.month && this.year == that.year);`
  - If a field is a primitive type, use ==.
  - If a field is an object, use equals().
  - If field is an array of primitives, use Arrays.equals().
  - If field is an area of objects, use Arrays.deepEquals().
    - But make sure the objects are immutable!

Separate Chaining implementation

```java
public class SeparateChainingLiteHashST<Key, Value> {
    private static final int INIT_CAPACITY = 128;
    private static final int LOAD_FACTOR_THRESHOLD = 4;

    private int m;                      // number of buckets
    private int n;                      // number of key-value pairs
    private Node[] table;             // array of linked-list chains - the table itself

    private class Node {
        Key key;
        Value val;
        Node next;

        public Node(Key key, Value val, Node next) {
            this.key = key;
            this.val = val;
            this.next = next;
        }
    }

    public SeparateChainingLiteHashST() {
        this(INIT_CAPACITY);
    }

    public SeparateChainingLiteHashST(int capacity) {
        m = capacity;
        table = (Node[]) new SeparateChainingLiteHashST.Node[m];
        n = 0;
    }

    private int hash(Key key) {
        return (key.hashCode() & 0x7fffffff) % m;
    }
```

Linear probing also has code, but we won't go over it in lecture (see code on website)

Worksheet time!

Fill in the blanks to implement get() in a separate chaining hash table. You can assume you have access to the hash() method, and an instance variable called table which is an array of Nodes, where Nodes contain a key, value, and next pointer (they are Nodes in a SLL).

```
public Value get(Key key) {
    int i = ______________________; //hash the key
    for (______________________________________) { //go through linked list
        if (_____________________) { //if the keys match
            return ______________; //return the value
        }
    }
    return null;
}
```

Worksheet solution

```
public Value get(Key key) {
    int i = hash(key);
    for (Node x = table[i]; x != null; x = x.next) {
        if (key.equals(x.key)) {
            return x.val;
        }
    }
    return null;
}
```

remember we use .equals() to compare keys!

Final notes

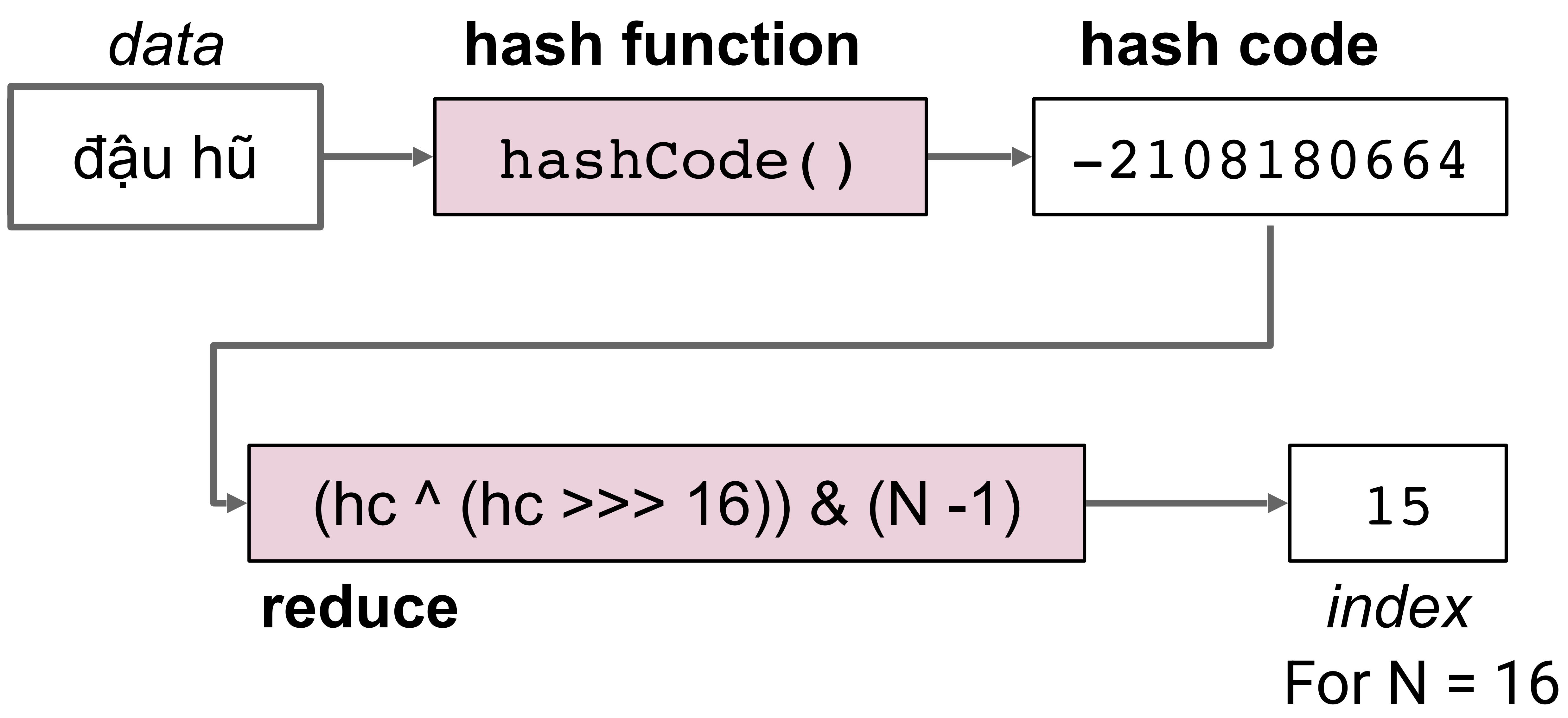
HashMap source code

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 using bitwise operations you'll learn in CS105.



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

Lecture 21 wrap-up

- Checkpoint 2 regrades due by lecture next Thurs.
  - Please submit them **digitally** on Gradescope instead of on paper.
- HW8: Hex-A-Pawn due tonight 11:59pm
- HW9: Transplant Manager released, due next Tues 11:59pm (partners OK)
- Reminder: quiz in lab tomorrow!

Resources

- Hashtable history of supporting the Holocaust & Japanese Internment camps: https://cs.pomona.edu/classes/cs62/history/hashtables/
- Reading from textbook: Chapter 3.4 (Pages 458-477); https://algs4.cs.princeton.edu/34hash/
- Hashtable visualization: https://visualgo.net/en/hashtable
- Practice problem behind this slide

Problem 1

- $h(k) = k \,\%\, m$ and $h(k, i) = (h(k) + i^2) \,\%\, m$.
- Assume $m = 9$, and key-value pairs to insert: (3,0), (9,1), (18,2), (0,3), (4,4), (36,5) in a quadratic probing hash table.

Answer 1

- $h(k) = k \,\%\, m$ and $h(k, i) = (h(k) + i^2) \,\%\, m$.
- Assume $m = 9$, and key-value pairs to insert: (3,0), (9,1), (18,2), (0,3), (4,4), (36,5).