CS 61BL Lab 13

Ryan Purpura

Announcements

- Gitlet due Friday at 11:59 pm!
- Fill out the Mid-semester Survey!
  - If 85% fill it out by the end of the day Friday, the whole class will get one extra credit point.
  - Please help us make the class better going forward and for future semesters!
- Midterm next Monday!

Maps

- A map is a structure that maps keys to values.
- Want fast insertions and fast lookups.
- Keys must be unique.

Key	Maps to	Value
Alice	→	0.85
Bob	→	0.81
Charlie	→	0.65
Daniel	→	0.71
Eve	→	0.67

Sets

- A set is a collection of items where there are no duplicates.
- You can think of a set as a map where only the keys matter (in fact, Sets in Java are just maps with keys mapping to dummy values)

Conceptually: {Alice, Bob, Charlie, Dan, Eve}

Using a map:

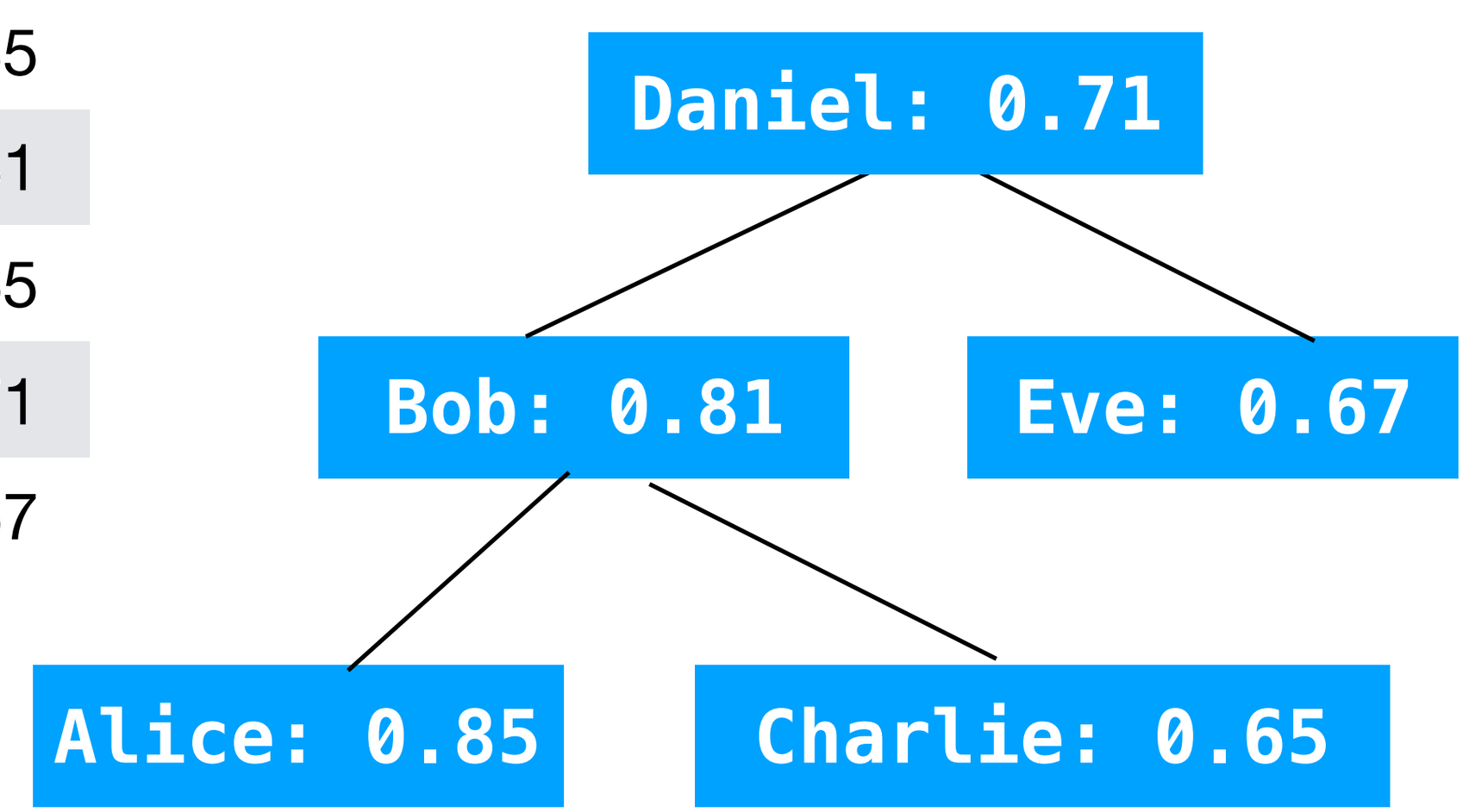
Key	Maps to	Value
Alice	→	null
Bob	→	null
Charlie	→	null
Daniel	→	null
Eve	→	null

How to implement

Idea 1: Use (balanced) binary search trees

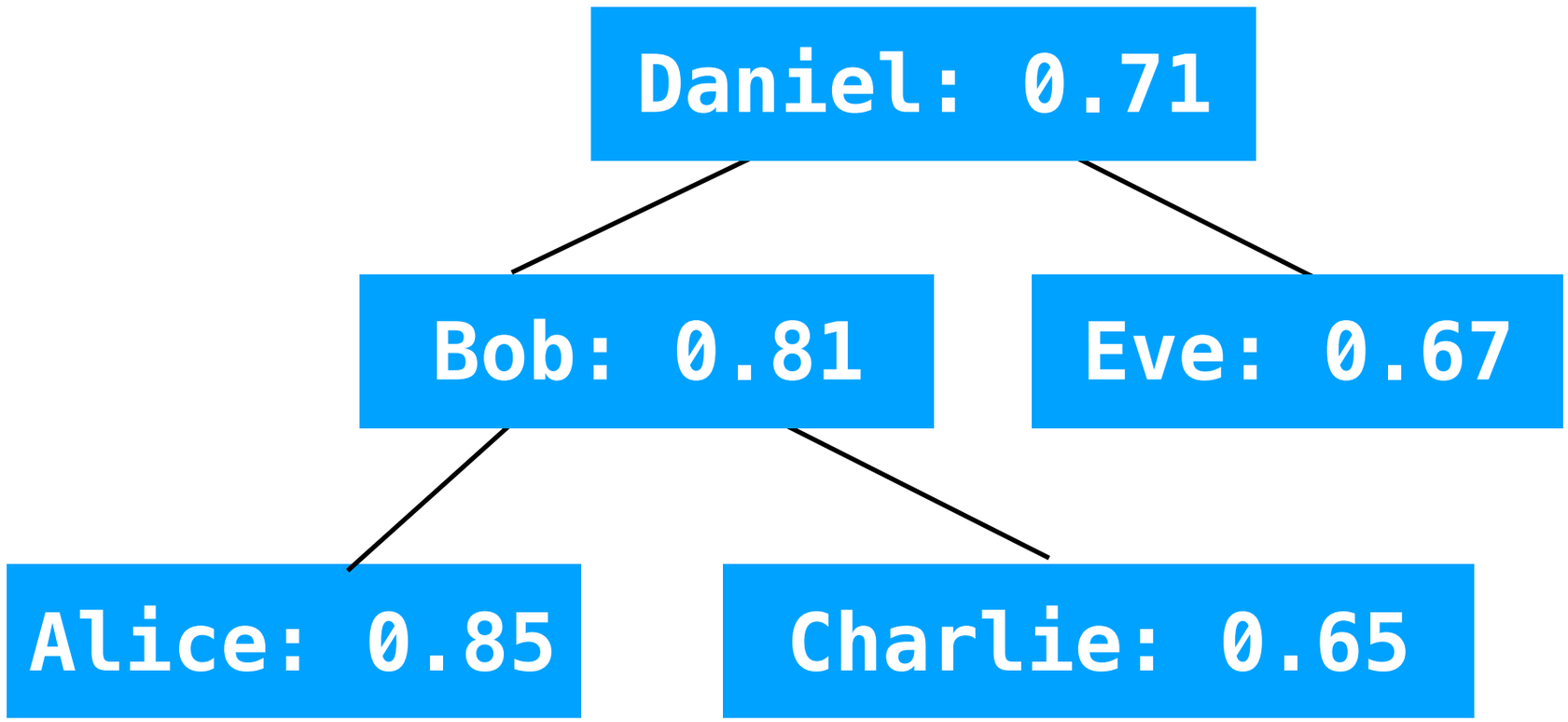
- Make an Entry class that is sorted by the search key.

Alice	→	0.85
Bob	→	0.81
Charlie	→	0.65
Dan	→	0.71
Eve	→	0.67



Tree Maps

- Insertions and lookups take $O(\log N)$ time: pretty good!
- Can use in-order traversal to obtain keys in sorted order!
- But not everything is Comparable or easy to write a Comparator for.
- $O(\log N)$ is good, but can we do it in $\Theta(1)$ time?



Constant-time Lookups

- What's a data structure we've already learned that has constant time lookups?
- Arrays!
- But arrays are indexed by **`int`**s.
- Ideas on how to use arrays to implement a map?

Idea: Convert key to an integer

- If we have some way to convert our search key into an integer, we can use that as an index to the array. (This is called ***hashing*** and the resulting integer is called a ***hash value***)
- For example, for our names (String), we could sum the chatAt values of all the letters.

- E.g. Alice becomes 65 + 108 + 105 + 99 + 101 = 478

Key (name)	Maps to	Value
Alice	→	0.85
Bob	→	0.81
Charlie	→	0.65
Daniel	→	0.71
Eve	→	0.67

Name	Index
Alice	478
Bob	275
Charlie	696
Daniel	589
Eve	288

Allocate a Massive Array

Key	Maps to	Value
Alice	→	0.85
Bob	→	0.81
Charlie	→	0.65
Daniel	→	0.71
Eve	→	0.67

Name	Index
Alice	478
Bob	275
Charlie	696
Daniel	589
Eve	288

0	null
1	null
2	null
...	...
275	Bob → 0.81
276	null
...	...
288	Eve → 0.67
...	...

Analysis

- Constant time lookups!
- But we're wasting a lot of memory.
- Let's use a smaller array and make indices "wrap around": we can do this with the modulo operator (%) *

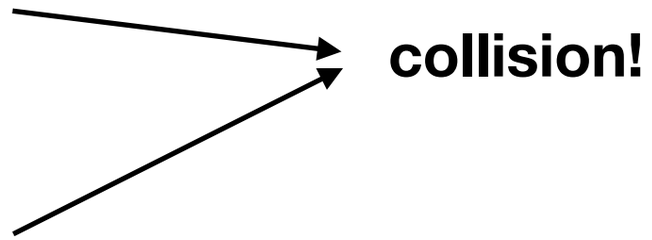
* We should use **`Math.floorMod`** in practice because we might have negative hashes.

0	null
1	null
2	null
...	...
275	Bob → 0.81
276	null
...	...
288	Eve → 0.67
...	...

Smaller Array

- Let's use a size 8 array instead.
- Calculating new array indices:

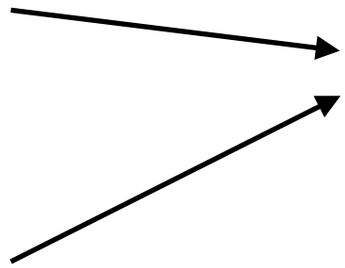
Name	Index
Alice	478 % 8 = 6
Bob	275 % 8 = 3
Charlie	696 % 8 = 0
Daniel	589 % 8 = 5
Eve	288 % 8 = 0



Hash Collisions

- We've run into a problem: when we try to map our hashes to fit into a small array, we will get overlaps.
- Ideas?

Name	Index
Alice	478 % 8 = 6
Bob	275 % 8 = 3
Charlie	696 % 8 = 0
Daniel	589 % 8 = 5
Eve	288 % 8 = 0

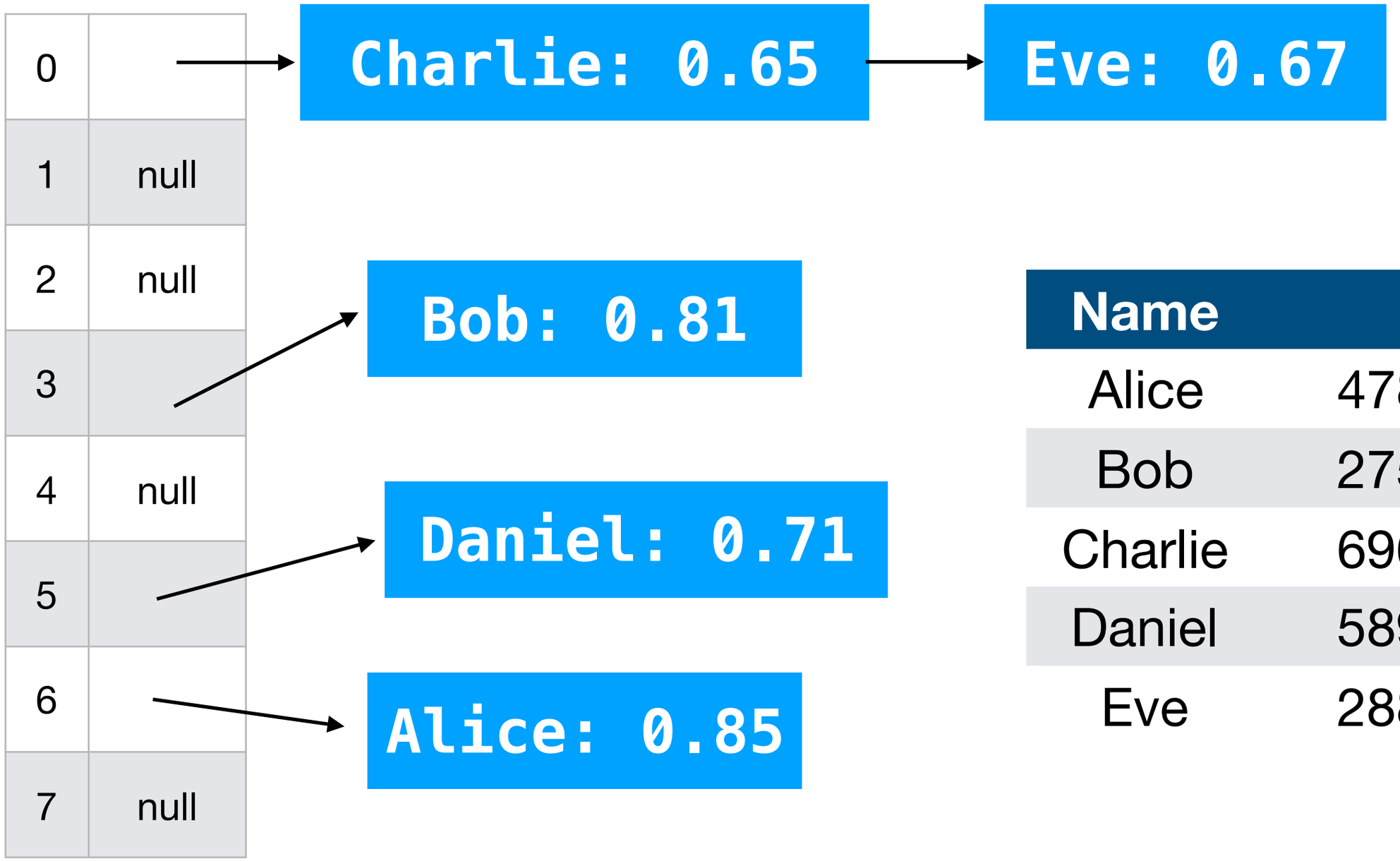


collision!

External Chaining

- The approach we'll use is called "external chaining"
- Each element in the array will point to another data structure (e.g. LinkedList), which will hold all of the Entries that correspond to that index.
- We'll call these "buckets".

External Chaining

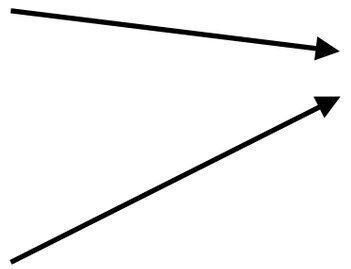


Name	Index
Alice	478 % 8 = 6
Bob	275 % 8 = 3
Charlie	696 % 8 = 0
Daniel	589 % 8 = 5
Eve	288 % 8 = 0

Resizing

- What if we have too few buckets?
- Our solution will be similar to ArrayLists: if ratio of elements to buckets exceeds some fill factor (e.g. 0.75), we'll resize.
- Do we need to calculate new indices?

Name	Old Index	New Index
Alice	478 % 8 = 6	478 % 16 = 14
Bob	275 % 8 = 3	275 % 16 = 3
Charlie	696 % 8 = 0	696 % 16 = 8
Daniel	589 % 8 = 5	589 % 16 = 13
Eve	288 % 8 = 0	288 % 16 = 0



no more collision!

Hash Tables

- This structure is called a "Hash Table"
- Used in Java's HashMap and HashSet
- Gives us average insertion and lookup time of $\Theta(1)$!
- But what is the worst case runtime?

- What case(s) cause the the worst case runtime?

Picking good hash functions

- First of all, your hash function must be *valid*
  - If two objects are equal by **`.equals()`**, they *must have the same hash value*
  - An unmutated object must always produce the same hash value (a.k.a. no randomness)
- Your hash function should be (but is not required to be) "good"
  - Minimizes collisions and roughly evenly distributed
  - Quick to compute

hashCode()

- All Java objects have a `hashCode` method.
- By default, returns the memory address of the object.
- Conveniently, the default implementation of .equals() compares memory addresses, so by default hashCode is valid.
- But what if we override `.equals()`?

hashCode

```
public int hashCode()
```

Returns a hash code value for the object. This method is supported for the benefit of hash tables such as those provided by `HashMap`.