CSCI 136:
Data Structures
and
Advanced Programming

Lecture 28

Hash collisions

Instructor: Kelly Shaw

Williams

Topics

Biased sampling

Hashcodes

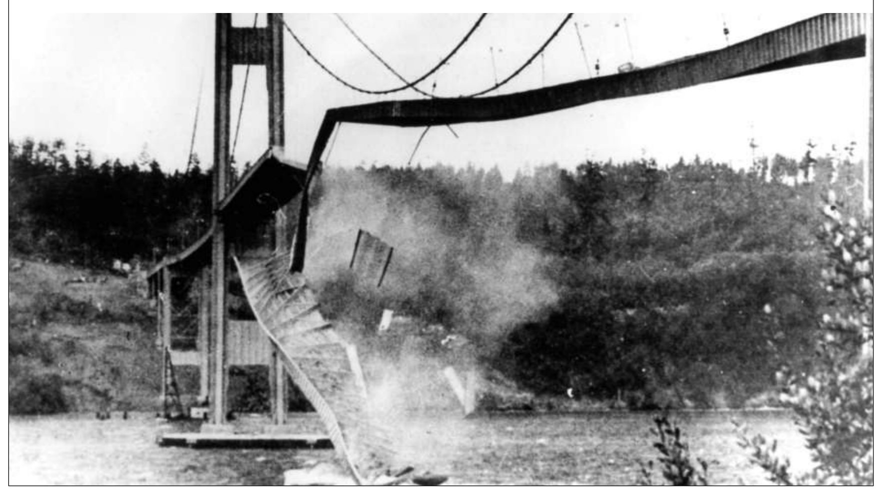
Hash collisions

Your to-dos

1. Read **before Mon**: *Bailey*, Ch. 16.3.
2. Lab 9 (partner lab), **due Tuesday 11/29 by 10pm**.
3. Quiz, **due Saturday evening**.

Study tip #4

"Failure is always an option."



Study tip #4

Adam Savage (MythBusters)

“'Failure is always an option' came up as a joke … when we were screwing something up over and over again, but it’s an awesome way to think about the scientific method. We tend to think about science as … a scientist saying, “I want to prove this thing,” and then coming up with an experiment to prove it. Nothing could be further from the truth.

Study tip #4

Adam Savage (MythBusters)

[In reality, a] scientist simply says, “I wonder if?” and then builds a methodology to test whether [the] theory is correct, or even to figure out what [the] theory might be. So to think that an experiment could “fail” is ludicrous. **Every experiment tells you something, even if it’s just don’t do that experiment the same way again.**”

Biased sampling

What does this program do?

```
Random r = new Random();
int num = r.nextInt(10);
```

Chooses a value between 0 and 9 inclusive with uniformly random probability.

I.e., all values are equally likely.

What if we want to specify the likelihood?

letter	likelihood
'a'	1
'b'	6
'c'	3

A naïve algorithm

'a'	'b'	'b'	'b'	'b'	'b'	'b'	'c'	'c'	'c'
0	1	2	3	4	5	6	7	8	9

```
char[] arr = new char[10];
// … code to fill array …
Random r = new Random();
int num = r.nextInt(10);
char c = arr[num];
```

A better algorithm

letter	likelihood
'a'	1
'b'	6
'c'	3

1. Compute the **sum of the likelihoods** (here: **10**).
2. Choose a number **n** between **0 … sum** (exclusive) uniformly randomly.
3. Set **soFar = 0**.
4. For each letter, add the **likelihood** to **soFar** and then check whether **n < soFar**. When **n < soFar** you've found the right letter.

Try it at home!

Notice that you get the **same answer** as using the naïve method.

'a'	'b'	'b'	'b'	'b'	'b'	'b'	'c'	'c'	'c'
0	1	2	3	4	5	6	7	8	9

1. Compute the **sum of the likelihoods** (here: **10**).
2. Choose a number **n** between **0 … sum** (exclusive) uniformly randomly.
3. Set **soFar = 0**.
4. For each letter, add the **likelihood** to **soFar** and then check whether **n < soFar**. When **n < soFar** you've found the right letter.

Hash codes

Hashing is so important that **every `Object` in Java** has a built-in hash function.

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`.

The general contract of `hashCode` is:

- Whenever it is invoked on the same object more than once during an execution of a Java application, the `hashCode` method must consistently return the same integer, provided no information used in `equals` comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
- If two objects are equal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects must produce the same integer result.
- It is *not* required that if two objects are unequal according to the `equals(java.lang.Object)` method, then calling the `hashCode` method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.

As much as is reasonably practical, the hashCode method defined by class `Object` does return distinct integers for distinct objects. (This is typically implemented by converting the internal address of the object into an integer, but this implementation technique is not required by the Java™ programming language.)

Returns:

a hash code value for this object.

See Also:

`equals(java.lang.Object)`, `System.identityHashCode(java.lang.Object)`

Hash codes

Good hash functions are already provided for **built-in types**.

Provide one for your own class by overriding **`hashCode`**.

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`.

The general contract of `hashCode` is:

- Whenever it is invoked on the same object more than once during an execution of a Java application, the `hashCode` method must consistently return the same integer, provided no information used in `equals` comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
- If two objects are equal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects must produce the same integer result.
- It is *not* required that if two objects are unequal according to the `equals(java.lang.Object)` method, then calling the `hashCode` method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.

As much as is reasonably practical, the hashCode method defined by class `Object` does return distinct integers for distinct objects. (This is typically implemented by converting the internal address of the object into an integer, but this implementation technique is not required by the Java™ programming language.)

Returns:

a hash code value for this object.

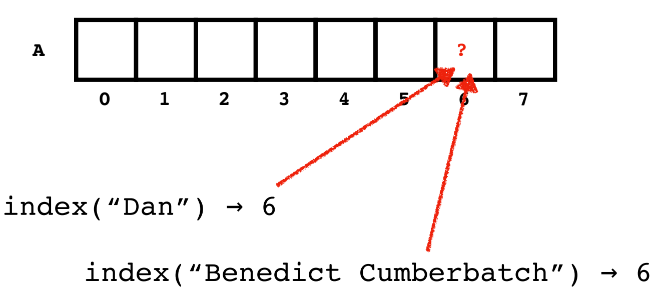
See Also:

`equals(java.lang.Object)`, `System.identityHashCode(java.lang.Object)`

Hash tables

Hash collisions

A **hash collision** is when **two or more distinct keys** have the **same hash value**.



Pigeonhole principle

Dealing with collisions

There are **two approaches** to dealing with collisions:

1. Change your **hash function**.
2. Change your **hash table design**.

The easier of the two approaches turns out to be #2.

We discuss **two hash table designs**: those that resolve collisions using **open addressing**, and those that resolve collisions using **external chaining**.

Open addressing

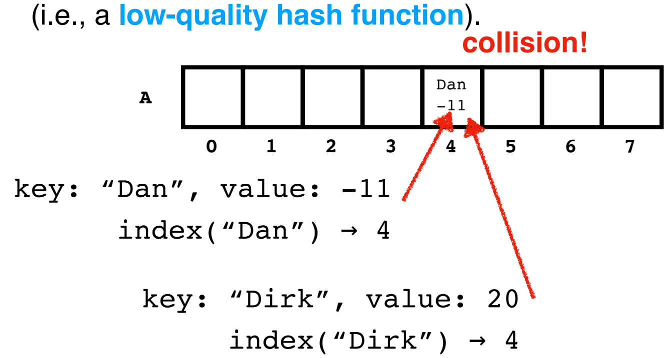
Open addressing is a method for resolving collisions in a hash table. Collisions are resolved by **probing**, which is a predetermined method for searching the hash table (aka **a probe sequence**). On **insertion**, probing finds the **first available bucket**. On **lookup**, probing searches until either the **key is found** or **an empty space** is found.

Linear probing

Suppose our keys are `Strings` and our hash function is

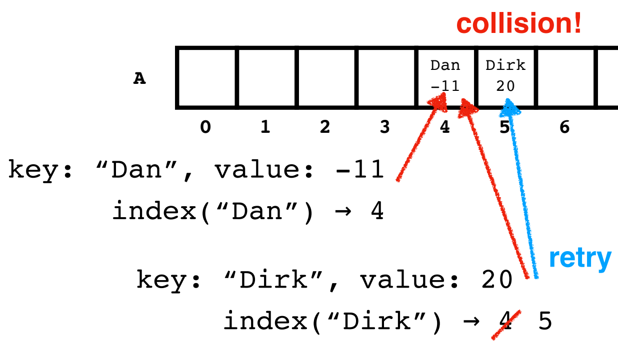
```
((int) key.charAt(0)) % A.length
```

(i.e., a **low-quality hash function**).



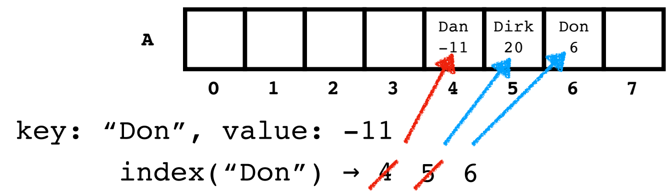
Linear probing

Linear probing works by scanning for $h(\texttt{key}) + c \times i$, where c is a constant (often 1) and i is the ith attempt.



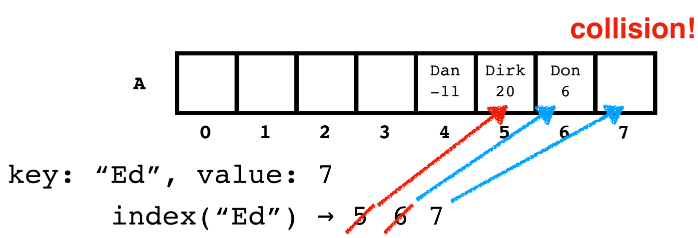
Linear probing

Linear probing works by scanning for $h(\texttt{key}) + c \times i$, where c is a constant (often 1) and i is the ith attempt.



Linear probing

Downside: values **cluster** around **collisions**.



Likelihood of collisions grows as cluster grows.

Our table is **still half empty**! This is **bad**!