Hash Tables

CS 2860: Algorithms and Complexity

Magnus Wahlström and Gregory Gutin

February 10, 2018

Recap: Hash tables

- Hash tables: "Almost perfect" data structure for unordered Sets/Maps
- Map: Stores mappings (associations) (`key` $\mapsto$ `value`)
- Constant ($O(1)$) average time for basic operations (`get`, `put`, `remove`, membership test)
- Does not support any ordering operations (no fast `min`, no fast `successor`, iteration in "unknown order" only)

The plan

- Saw:
  - Implementation
  - Hash buckets, collision strategies
- Today:
  - Hash functions
  - The hashCode/equals contract
  - Other hash applications

Hash Functions

Hash functions

- Cryptographic hash functions – convert file (sequence/block of bytes) to "fingerprint" bitstring
  - Different inputs should practically always give different hashes
  - Ideally infeasible to find even single pair with same hash value
  - Ideally uninvertible: Cannot guess input from hash value; data fully scrambled, differences exaggerated
- Applications:
  - Digital signatures: use short signature from trusted source to verify nature of large files from untrusted sources
  - Detect whether file has been changed/corrupted
  - File identifiers – git, svn, backup systems

Hash functions

- Our hashes are not cryptographic
- Compared to cryptographic hashes, what we need is weaker:
  - Our hashes are 32 bits, not (say) 256 bits
  - Collisions are annoying, but no disasters
  - Computation should be fast
- Another difference:
  - Cryptographic hash depends on bit-level raw data
  - Our hashes depend on logical structure of data
  - (See hashCode/equals contract, later)

Hash functions

- Our hashes are not cryptographic
- Compared to cryptographic hashes, what we need is weaker:
  - Our hashes are 32 bits, not (say) 256 bits
  - Collisions are annoying, but no disasters
  - Computation should be fast
- Another difference:
  - Cryptographic hash depends on bit-level raw data
  - Our hashes depend on logical structure of data
  - (See hashCode/equals contract, later)

Hash functions

- Our hashes are not cryptographic
- Compared to cryptographic hashes, what we need is weaker:
  - Our hashes are 32 bits, not (say) 256 bits
  - Collisions are annoying, but no disasters
  - Computation should be fast
- Another difference:
  - Cryptographic hash depends on bit-level raw data
  - Our hashes depend on logical structure of data
  - (See hashCode/equals contract, later)

Hash functions

- Our hashes are not cryptographic
- Compared to cryptographic hashes, what we need is weaker:
  - Our hashes are 32 bits, not (say) 256 bits
  - Collisions are annoying, but no disasters
  - Computation should be fast
- Another difference:
  - Cryptographic hash depends on bit-level raw data
  - Our hashes depend on logical structure of data
  - (See hashCode/equals contract, later)

★Large numbers

- The number of 32-bit hashes is 2^{32}
- For most types of data, collisions are unavoidable:
  - The number of 10-character strings made from letters a–z is
    $$26^{10} \approx 2^{47}$$
  - There are at least $2^{15} > 30,000$ different 10-letter words that all map to the same hash code (no matter what hash function)
  - The number of files of 1,024 bytes is
    $$256^{1024} = 2^{8192}$$
  - The number of different 1K-files with the same hash code is unimaginably large, no matter what hash function you use
- Most of these strings/files/... will never be created
  1. If we have an adversary, they may maliciously craft colliding strings (e.g., as a part of a DoS-attack against a web server)
  2. If we're not worried about that scenario, we'll be fine

★Large numbers

- The number of 32-bit hashes is 2^{32}
- For most types of data, collisions are unavoidable:
  - The number of 10-character strings made from letters a–z is

$$26^{10} \approx 2^{47}$$

  - There are at least $2^{15} > 30,000$ different 10-letter words that all map to the same hash code (no matter what hash function)
  - The number of files of 1,024 bytes is

$$256^{1024} = 2^{8192}$$

  - The number of different 1K-files with the same hash code is unimaginably large, no matter what hash function you use
- Most of these strings/files/... will never be created
  1. If we have an adversary, they may maliciously craft colliding strings (e.g., as a part of a DoS-attack against a web server)
  2. If we're not worried about that scenario, we'll be fine

★Large numbers

- The number of 32-bit hashes is 2^{32}
- For most types of data, collisions are unavoidable:
  - The number of 10-character strings made from letters a–z is

$$26^{10} \approx 2^{47}$$

  - There are at least $2^{15} > 30,000$ different 10-letter words that all map to the same hash code (no matter what hash function)
  - The number of files of 1,024 bytes is

$$256^{1024} = 2^{8192}$$

  - The number of different 1K-files with the same hash code is unimaginably large, no matter what hash function you use
- Most of these strings/files/... will never be created
  1. If we have an adversary, they may maliciously craft colliding strings (e.g., as a part of a DoS-attack against a web server)
  2. If we're not worried about that scenario, we'll be fine

★Large numbers

- The number of 32-bit hashes is 2^{32}
- For most types of data, collisions are unavoidable:
  - The number of 10-character strings made from letters a–z is

$$26^{10} \approx 2^{47}$$

  - There are at least $2^{15} > 30,000$ different 10-letter words that all map to the same hash code (no matter what hash function)
  - The number of files of 1,024 bytes is

$$256^{1024} = 2^{8192}$$

  - The number of different 1K-files with the same hash code is unimaginably large, no matter what hash function you use
- Most of these strings/files/...will never be created
  1. If we have an adversary, they may maliciously craft colliding strings (e.g., as a part of a DoS-attack against a web server)
  2. If we're not worried about that scenario, we'll be fine

★Large numbers

- The number of 32-bit hashes is 2^{32}
- For most types of data, collisions are unavoidable:
  - The number of 10-character strings made from letters a–z is

$$26^{10} \approx 2^{47}$$

  - There are at least $2^{15} > 30,000$ different 10-letter words that all map to the same hash code (no matter what hash function)
  - The number of files of 1,024 bytes is

$$256^{1024} = 2^{8192}$$

  - The number of different 1K-files with the same hash code is unimaginably large, no matter what hash function you use
- Most of these strings/files/…will never be created
  1. If we have an adversary, they may maliciously craft colliding strings (e.g., as a part of a DoS-attack against a web server)
  2. If we're not worried about that scenario, we'll be fine

★Behaviour of random keys

- Assume a table of capacity m, containing n random items
- At what value of n does each of the following occur?
    1. The first collision:
    2. Table half-full:
    3. Table 90% full:
    4. Table entirely full:

★Behaviour of random keys

- Assume a table of capacity m, containing n random items
- At what value of n does each of the following occur?
  1. The first collision: At $n = \Omega(\sqrt{m})$ (birthday paradox)
  2. Table half-full:
  3. Table 90% full:
  4. Table entirely full:

★Behaviour of random keys

- Assume a table of capacity m, containing n random items
- At what value of n does each of the following occur?
  1. The first collision: At $n = \Omega(\sqrt{m})$ (birthday paradox)
  2. Table half-full: At 70% use ($n = 0.7m$)
  3. Table 90% full:
  4. Table entirely full:

★Behaviour of random keys

- Assume a table of capacity m, containing n random items
- At what value of n does each of the following occur?
  1. The first collision: At $n = \Omega(\sqrt{m})$ (birthday paradox)
  2. Table half-full: At 70% use ($n = 0.7m$)
  3. Table 90% full: At $\approx$ 200% use
  4. Table entirely full:

★Behaviour of random keys

- Assume a table of capacity m, containing n random items
- At what value of n does each of the following occur?
  1. The first collision: At $n = \Omega(\sqrt{m})$ (birthday paradox)
  2. Table half-full: At 70% use ($n = 0.7m$)
  3. Table 90% full: At $\approx$ 200% use
  4. Table entirely full: At $n = \Omega(m \log m)$ (coupon collection)

★Behaviour of random keys

- Assume a table of capacity m, containing n random items
- At what value of n does each of the following occur?
  1. The first collision: At $n = \Omega(\sqrt{m})$ (birthday paradox)
  2. Table half-full: At 70% use ($n = 0.7m$)
  3. Table 90% full: At $\approx$ 200% use
  4. Table entirely full: At $n = \Omega(m \log m)$ (coupon collection)
- When $n = m$, the fraction of free slots is approximately

$$(1 - \tfrac{1}{m})^n = (1 - \tfrac{1}{m})^m \approx 1/e \approx 37\%$$

★Behaviour of random keys

- Assume a table of capacity m, containing n random items
- At what value of n does each of the following occur?
  1. The first collision: At $n = \Omega(\sqrt{m})$ (birthday paradox)
  2. Table half-full: At 70% use ($n = 0.7m$)
  3. Table 90% full: At $\approx$ 200% use
  4. Table entirely full: At $n = \Omega(m \log m)$ (coupon collection)
- When $n = m$, the fraction of free slots is approximately

$$(1 - \tfrac{1}{m})^n = (1 - \tfrac{1}{m})^m \approx 1/e \approx 37\%$$

- The "longest run" when $n = m$ is probably $O(\log n)$, but most slots are short (0/1/2 elements)

★Behaviour of random keys

- Assume a table of capacity m, containing n random items
- At what value of n does each of the following occur?
  1. The first collision: At $n = \Omega(\sqrt{m})$ (birthday paradox)
  2. Table half-full: At 70% use ($n = 0.7m$)
  3. Table 90% full: At $\approx$ 200% use
  4. Table entirely full: At $n = \Omega(m \log m)$ (coupon collection)
- When $n = m$, the fraction of free slots is approximately

$$(1 - \tfrac{1}{m})^n = (1 - \tfrac{1}{m})^m \approx 1/e \approx 37\%$$

- The "longest run" when $n = m$ is probably $O(\log n)$, but most slots are short (0/1/2 elements)
- Common strategy: Move to larger table at $n = 0.7m$

Desired properties of "our" hash functions

- Easy to compute (e.g., $O(n)$ time if object has size n)
- Reasonably irregular (few collisions)
- Few collisions even in the face of realistic, non-random use patterns
- Consistent with equals

Desired properties of "our" hash functions

- Easy to compute (e.g., $O(n)$ time if object has size n)
- Reasonably irregular (few collisions)
- Few collisions even in the face of realistic, non-random use patterns
- Consistent with equals

Desired properties of “our” hash functions

- Easy to compute (e.g., $O(n)$ time if object has size n)
- Reasonably irregular (few collisions)
- Few collisions even in the face of realistic, non-random use patterns
- Consistent with equals

Desired properties of "our" hash functions

- Easy to compute (e.g., $O(n)$ time if object has size n)
- Reasonably irregular (few collisions)
- Few collisions even in the face of realistic, non-random use patterns
- Consistent with equals

The hashCode/equals contract

hashCode and equals

- Fact: All objects in Java have the following two methods:
  - object.hashCode(): return hash value
  - object1.equals(object2): equivalence test
- There's an important contract between these two:
  1. If you change one of these methods (e.g., equals), you also need to change the other!
  2. If you do, your methods must be consistent with each other!
- Why?

hashCode and equals

- Fact: All objects in Java have the following two methods:
  - object.hashCode(): return hash value
  - object1.equals(object2): equivalence test
- There's an important contract between these two:
  1. If you change one of these methods (e.g., equals), you also need to change the other!
  2. If you do, your methods must be consistent with each other!
- Why?

hashCode and equals

- Fact: All objects in Java have the following two methods:
  - object.hashCode(): return hash value
  - object1.equals(object2): equivalence test
- There's an important contract between these two:
  1. If you change one of these methods (e.g., equals), you also need to change the other!
  2. If you do, your methods must be consistent with each other!
- Why?
- Consistency:
  - If x.equals(y) then x.hashCode()==y.hashCode()!
  - Converse: If x.hashCode()!=y.hashCode(), then !x.equals(y).

Testing object equivalence

- Recall: Want to test (for Strings, Sets, Arrays...) whether two distinct objects have equivalent contents
- Example: Two different strings both contain "rabbit"
- Example: Two different Sets contain the same items (but in different orders in the tree/in the hash buckets!)
- So to test whether two Sets are equal, we cannot use "bit-level identity" (as a cryptographic hash code would give us). We have to do:

Testing object equivalence

- Recall: Want to test (for Strings, Sets, Arrays...) whether two distinct objects have equivalent contents
- Example: Two different strings both contain "rabbit"
- Example: Two different Sets contain the same items (but in different orders in the tree/in the hash buckets!)
- So to test whether two Sets are equal, we cannot use "bit-level identity" (as a cryptographic hash code would give us). We have to do:

Testing object equivalence

- Recall: Want to test (for Strings, Sets, Arrays...) whether two distinct objects have equivalent contents
- Example: Two different strings both contain "rabbit"
- Example: Two different Sets contain the same items (but in different orders in the tree/in the hash buckets!)
- So to test whether two Sets are equal, we cannot use "bit-level identity" (as a cryptographic hash code would give us). We have to do:
  1. If set1.size != set2.size, they are different
  2. Otherwise, for every object x in set1:
     2.1 If set2 does not contain x (or an object equivalent to x!) the sets are different[1]
  3. If the loop terminates, the sets are equivalent

[1] The test "contains an object equivalent to x" is just set2.contains(x)

Testing object equivalence

- Recall: Want to test (for Strings, Sets, Arrays...) whether two distinct objects have equivalent contents
- Example: Two different strings both contain "rabbit"
- Example: Two different Sets contain the same items (but in different orders in the tree/in the hash buckets!)
- So to test whether two Sets are equal, we cannot use "bit-level identity" (as a cryptographic hash code would give us). We have to do:
    1. If set1.size != set2.size, they are different
    2. Otherwise, for every object x in set1:
        - 2.1 If set2 does not contain x (or an object equivalent to x!) the sets are different[1]
    3. If the loop terminates, the sets are equivalent
- So since a Set must implement its own equals code, it must also implement a new hashCode function

[1] The test "contains an object equivalent to x" is just set2.contains(x)

Default operations

- The default rules (for Object) are:
  - object.hashCode() returns the physical memory address
  - object1.equals(object2) tests object1 == object2?
- Consistent?

Default operations

- The default rules (for Object) are:
  - object.hashCode() returns the physical memory address
  - object1.equals(object2) tests object1 == object2?
- Consistent?

Default operations

- The default rules (for Object) are:
  - object.hashCode() returns the physical memory address
  - object1.equals(object2) tests object1 == object2?
- Consistent? Yes.

Default operations

- The default rules (for Object) are:
  - object.hashCode() returns the physical memory address
  - object1.equals(object2) tests object1 == object2?
- Consistent? Yes.
- But it's also a very "boring" equality test.
- If you subclass directly from Object, this is your default functionality

A bad hash function for Strings

- Recall:
  1. String objects test equality by character-by-character comparisons
  2. Every character in a String maps to a unicode code page (a number 0–2^{32}, usually 0–255)
- What about the following hash code for strings?
  1. Initialise sum=0
  2. For every character c in the string:
     2.1 Add the numerical value of c to sum
  3. Return resulting sum
- Is it (1) consistent? (2) good?

A bad hash function for Strings

- Recall:
    1. String objects test equality by character-by-character comparisons
    2. Every character in a String maps to a unicode code page (a number 0–2^{32}, usually 0–255)
- What about the following hash code for strings?
    1. Initialise sum=0
    2. For every character c in the string:
        2.1 Add the numerical value of c to sum
    3. Return resulting sum
- Is it (1) consistent? (2) good?

A bad hash function for Strings

- Recall:
  1. String objects test equality by character-by-character comparisons
  2. Every character in a String maps to a unicode code page (a number 0–2^{32}, usually 0–255)
- What about the following hash code for strings?
  1. Initialise sum=0
  2. For every character c in the string:
     2.1 Add the numerical value of c to sum
  3. Return resulting sum
- Is it (1) consistent? (2) good?

A bad hash function for Strings

- Recall:
  1. String objects test equality by character-by-character comparisons
  2. Every character in a String maps to a unicode code page (a number 0–2^{32}, usually 0–255)
- What about the following hash code for strings?
  1. Initialise sum=0
  2. For every character c in the string:
     2.1 Add the numerical value of c to sum
  3. Return resulting sum
- Is it (1) consistent? (2) good?
  1. Yes: Character-by-character equality tests on bit-level.

A bad hash function for Strings

- Recall:
  1. String objects test equality by character-by-character comparisons
  2. Every character in a String maps to a unicode code page (a number 0–2^{32}, usually 0–255)
- What about the following hash code for strings?
  1. Initialise sum=0
  2. For every character c in the string:
     2.1 Add the numerical value of c to sum
  3. Return resulting sum
- Is it (1) consistent? (2) good?
  1. Yes: Character-by-character equality tests on bit-level.
  2. High quality?
     - hash(god)==hash(dog), hash(cat)==hash(act)
     - hash(reset)==hash(trees)==hash(steer)==hash(terse)

A bad hash function for Strings

- Recall:
  1. String objects test equality by character-by-character comparisons
  2. Every character in a String maps to a unicode code page (a number 0–2^{32}, usually 0–255)
- What about the following hash code for strings?
  1. Initialise sum=0
  2. For every character c in the string:
     2.1 Add the numerical value of c to sum
  3. Return resulting sum
- Is it (1) consistent? (2) good?
  1. Yes: Character-by-character equality tests on bit-level.
  2. High quality?
     - hash(god)==hash(dog), hash(cat)==hash(act)
     - hash(reset)==hash(trees)==hash(steer)==hash(terse)
- Hash code is low-quality on common use patterns, because English text contains many anagrams
- Also, it "throws away" the dependency on order

Immutable objects

- A common programming trick is to make objects immutable – unchangeable
- Example String objects:
  - Cannot change one character inside a String
  - Can return a different String with one character changed
- Consequence for hashes:
  - Once we have computed a hashCode value, we may remember the hash value in a member variable and never compute it again
- (Caching hash values is also possible for mutable objects, if you're careful to forget the hash every time it changes)
- Note: Computing hashCode should take at least $\Omega(n)$ time (but hopefully not more)

Immutable objects

- A common programming trick is to make objects immutable – unchangeable
- Example String objects:
  - Cannot change one character inside a String
  - Can return a different String with one character changed
- Consequence for hashes:
  - Once we have computed a hashCode value, we may remember the hash value in a member variable and never compute it again
- (Caching hash values is also possible for mutable objects, if you're careful to forget the hash every time it changes)
- Note: Computing hashCode should take at least $\Omega(n)$ time (but hopefully not more)

Immutable objects

- A common programming trick is to make objects immutable – unchangeable
- Example String objects:
  - Cannot change one character inside a String
  - Can return a different String with one character changed
- Consequence for hashes:
  - Once we have computed a hashCode value, we may remember the hash value in a member variable and never compute it again
- (Caching hash values is also possible for mutable objects, if you're careful to forget the hash every time it changes)
- Note: Computing hashCode should take at least $\Omega(n)$ time (but hopefully not more)

Immutable objects

- A common programming trick is to make objects immutable – unchangeable
- Example String objects:
  - Cannot change one character inside a String
  - Can return a different String with one character changed
- Consequence for hashes:
  - Once we have computed a hashCode value, we may remember the hash value in a member variable and never compute it again
- (Caching hash values is also possible for mutable objects, if you're careful to forget the hash every time it changes)
- Note: Computing hashCode should take at least $\Omega(n)$ time (but hopefully not more)

Mutable/immutable

- Suggestions?

Mutable/immutable

- Suggestions?
- Necessarily mutable: Array, Set, Map...

Mutable/immutable

- Suggestions?
- Necessarily mutable: Array, Set, Map...
- Easily immutable: Single-linked lists (with some programming discipline)
- Same principle for binary trees without parent links

Immutable / mutable keys

Mutable key types

Warning: Never modify an object after it has been used as the key of a hash table!

- Many objects (String, Integer, ...) are immutable: They cannot be modified after creation
  - You cannot change the value of 3 to 5, only "rebind" a single variable from 3 to 5; similarly with strings
- However, some modifiable data types (e.g., arrays/lists) are still usable as hash table key types
- Modifying these after insertion is a bug!

Immutable / mutable keys

Mutable key types

Warning: Never modify an object after it has been used as the key of a hash table!

- Many objects (String, Integer, ...) are immutable: They cannot be modified after creation
  - You cannot change the value of 3 to 5, only "rebind" a single variable from 3 to 5; similarly with strings
- However, some modifiable data types (e.g., arrays/lists) are still usable as hash table key types
- Modifying these after insertion is a bug!

Immutable / mutable keys

Mutable key types

Warning: Never modify an object after it has been used as the key of a hash table!

- Many objects (String, Integer, ...) are immutable: They cannot be modified after creation
  - You cannot change the value of 3 to 5, only “rebind” a single variable from 3 to 5; similarly with strings
- However, some modifiable data types (e.g., arrays/lists) are still usable as hash table key types
- Modifying these after insertion is a bug!

Immutable / mutable keys

Mutable key types

Warning: Never modify an object after it has been used as the key of a hash table!

- Many objects (String, Integer, ...) are immutable: They cannot be modified after creation
  - You cannot change the value of 3 to 5, only "rebind" a single variable from 3 to 5; similarly with strings
- However, some modifiable data types (e.g., arrays/lists) are still usable as hash table key types
- Modifying these after insertion is a bug!

Immutable keys warning explained

- Lookup procedure: Consider call `lookup(item)`
  1. Let `hash1 = item.hashCode()`
  2. Compute hash bucket `bucket = (hash1 % m)` (table capacity m)
  3. For each element `item2` stored in slot `bucket`:
     - 3.1 If `item2.hashCode() != hash1`: Continue to next object
     - 3.2 If `item2.hashCode() == hash1`, test `item2.equals(item)`
     - 3.3 If positive, return YES (found a copy of item in the hash table)
- What happens if the "old" item `item2` was modified after insertion?
  - The new and the old versions can have different hash codes: The item would be in the wrong place!
  - Even a test `hashtable.contains(item2)` would fail after modification!

Immutable keys warning explained

- Lookup procedure: Consider call `lookup(item)`
  1. Let `hash1 = item.hashCode()`
  2. Compute hash bucket `bucket = (hash1 % m)` (table capacity m)
  3. For each element `item2` stored in slot `bucket`:
     3.1 If `item2.hashCode() != hash1`: Continue to next object
     3.2 If `item2.hashCode() == hash1`, test `item2.equals(item)`
     3.3 If positive, return YES (found a copy of item in the hash table)
- What happens if the “old” item `item2` was modified after insertion?
  - The new and the old versions can have different hash codes: The item would be in the wrong place!
  - Even a test `hashtable.contains(item2)` would fail after modification!

Hash functions in Java

- Integer: hash(i)=i
- Single character: hash(c)=Unicode code point of c
- String S, length n:
  - `s[0]`$\cdot 31^{n-1}$ + `s[1]`$\cdot 31^{n-2}$ + ...+ `s[n-1]`$\cdot$ 1
- Array/ordered collection A:
  - hash(A[0])*31^{n-1} + hash(A[1])*31^{n-2} + ...
- Unordered collection (set) S:
  - Sum of hash(x) over all x in S

Hash functions in Java

- Integer: hash(i)=i
- Single character: hash(c)=Unicode code point of c
- String S, length n:
  - s[0]$\cdot 31^{n-1}$ + s[1]$\cdot 31^{n-2}$ + ...+ s[n-1]$\cdot$ 1
- Array/ordered collection A:
  - hash(A[0])*31^{n-1} + hash(A[1])*31^{n-2} + ...
- Unordered collection (set) S:
  - Sum of hash(x) over all x in S

Hash functions in Java

- Integer: hash(i)=i
- Single character: hash(c)=Unicode code point of c
- String S, length n:
  - `s[0]`$\cdot 31^{n-1}$ `+ s[1]`$\cdot 31^{n-2}$ `+ ...+ s[n-1]`$\cdot$ `1`
- Array/ordered collection A:
  - hash(A[0])*31^{n-1} + hash(A[1])*31^{n-2} + ...
- Unordered collection (set) S:
  - Sum of hash(x) over all x in S

Hash functions in Java

- Integer: hash(i)=i
- Single character: hash(c)=Unicode code point of c
- String S, length n:
  - `s[0]`$\cdot 31^{n-1}$ `+ s[1]`$\cdot 31^{n-2}$ `+ ...+ s[n-1]`$\cdot$ `1`
- Array/ordered collection A:
  - hash(A[0])*31^{n-1} + hash(A[1])*31^{n-2} + ...
- Unordered collection (set) S:
  - Sum of hash(x) over all x in S

Other hash applications (v. briefly)

Applications

1. To speed up equivalence testing (if hashCode has been cached)
2. More clever string comparison (Rabin-Karp algorithm)
   2.1 Compute hash(pattern) length m
   2.2 Compute hash(string[0 . . . m-1]); compare hash values
   2.3 Update to hash(string[1 . . . m]); compare values
   2.4 Update to hash(string[2 . . . m]); compare values . . .
3. Bloom filters
   3.1 When working with very large objects, instead of a full hash table, implement only the Boolean array solution
   3.2 Can (probabilistically!) answer question "have I seen this object before?"
   3.3 Space $O(n)$ for n objects of size S, much better than $n \cdot S$
   3.4 Example: "Does computer ... in my distributed network have a copy of this file?"

Applications

1. To speed up equivalence testing (if hashCode has been cached)
2. More clever string comparison (Rabin-Karp algorithm)
    2.1 Compute hash(pattern) length m
    2.2 Compute hash(string[0 ... m-1]); compare hash values
    2.3 Update to hash(string[1 ... m]); compare values
    2.4 Update to hash(string[2 ... m]); compare values ...
3. Bloom filters
    3.1 When working with very large objects, instead of a full hash table, implement only the Boolean array solution
    3.2 Can (probabilistically!) answer question “have I seen this object before?”
    3.3 Space $O(n)$ for n objects of size S, much better than $n \cdot S$
    3.4 Example: “Does computer ... in my distributed network have a copy of this file?”

Applications

1. To speed up equivalence testing (if hashCode has been cached)
2. More clever string comparison (Rabin-Karp algorithm)
   2.1 Compute hash(pattern) length m
   2.2 Compute hash(string[0 . . . m-1]); compare hash values
   2.3 Update to hash(string[1 . . . m]); compare values
   2.4 Update to hash(string[2 . . . m]); compare values . . .
3. Bloom filters
   3.1 When working with very large objects, instead of a full hash table, implement only the Boolean array solution
   3.2 Can (probabilistically!) answer question "have I seen this object before?"
   3.3 Space $O(n)$ for n objects of size S, much better than $n \cdot S$
   3.4 Example: "Does computer ... in my distributed network have a copy of this file?"

Applications

1. To speed up equivalence testing (if hashCode has been cached)
2. More clever string comparison (Rabin-Karp algorithm)
   2.1 Compute hash(pattern) length m
   2.2 Compute hash(string[0 ... m-1]); compare hash values
   2.3 Update to hash(string[1 ... m]); compare values
   2.4 Update to hash(string[2 ... m]); compare values ...
3. Bloom filters
   3.1 When working with very large objects, instead of a full hash table, implement only the Boolean array solution
   3.2 Can (probabilistically!) answer question "have I seen this object before?"
   3.3 Space $O(n)$ for n objects of size S, much better than $n \cdot S$
   3.4 Example: "Does computer ... in my distributed network have a copy of this file?"

Applications

1. To speed up equivalence testing (if hashCode has been cached)
2. More clever string comparison (Rabin-Karp algorithm)
   2.1 Compute hash(pattern) length m
   2.2 Compute hash(string[0 ... m-1]); compare hash values
   2.3 Update to hash(string[1 ... m]); compare values
   2.4 Update to hash(string[2 ... m]); compare values ...
3. Bloom filters
   3.1 When working with very large objects, instead of a full hash table, implement only the Boolean array solution
   3.2 Can (probabilistically!) answer question "have I seen this object before?"
   3.3 Space $O(n)$ for n objects of size S, much better than $n \cdot S$
   3.4 Example: "Does computer ... in my distributed network have a copy of this file?"

Summary

We saw:

- Properties of hash functions and random numbers
- The hashCode/equals contract
- A few samples of “other algorithmic applications” of hash functions (may return to these)