The Hashing Technique

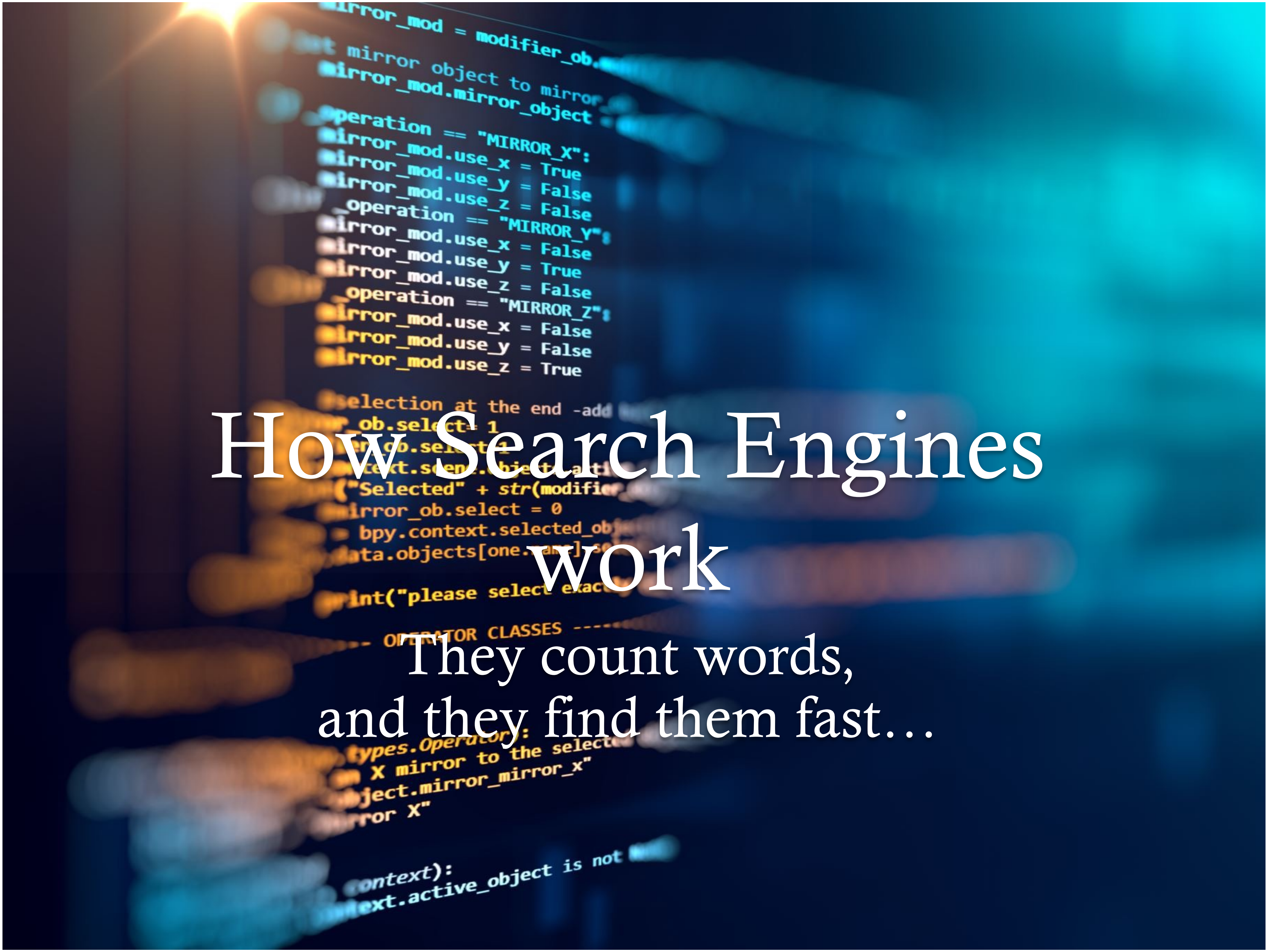
You have a huge number of items
and you want to search them. Fast!

You don't have time to spare
but you have space to spare

How fast Search Engines search?

- They have a *huge* collection of N items
- How can they organize them so that they can search fast (in terms of O(?) or even in number of steps)?

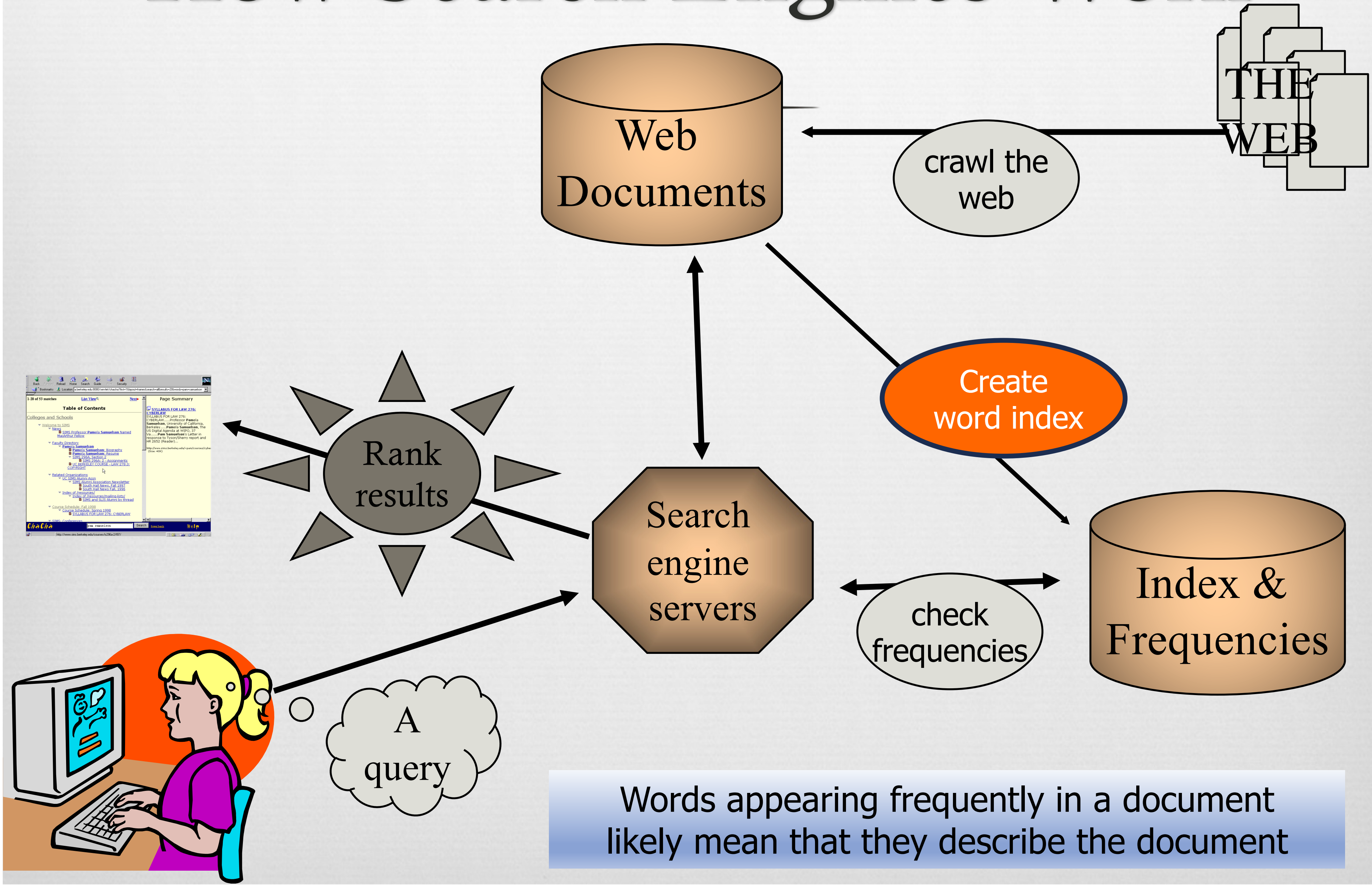
	Add	Search	Remove	Space
Unsorted Array				
Unsorted LinkedList				
Sorted Array				
Sorted LinkedList				



How Search Engines work

They count words,
and they find them fast…

How Search Engines Work



THE WEB

crawl the web

Web Documents

Create word index

Search engine servers

Rank results

check frequencies

Index & Frequencies

A query

Words appearing frequently in a document likely mean that they describe the document

Computing Word Index and Frequency

i am sam
i am sam
sam i am
that sam i am
that sam i am
i do not like
that sam i am
do you like green eggs and ham
i do not like them
sam i am
i do not like
green eggs and ham
would you like them
here or there
i would not like them
here or there
i would not like them
anywhere
i do not like
green eggs and ham
i do not like them
sam i am
would you like them
in a house
would you like them
with a mouse

i do not like them
in a house
i do not like them
with a mouse
i do not like them
here or there
i do not like them
Anywhere
i do not like green eggs and ham
i do not like them sam i am
would you eat them
in a box
would you eat them
with a fox
not in a box
not with a fox
not in a house
not with a mouse
i would not eat them here or there
i would not eat them anywhere
i would not eat green eggs and ham
i do not like them sam i am
would you could you
in a car
eat them eat them
here they are
i would not
could not
in a car

a :59
am :16
and :25
anywhere :8
are :2
be :4
boat :3
box :7
car :7
could :14
dark :7
do :37
eat :25
eggs :11
fox :7
goat :4
…
try :4
will :21
with :19
would :26
you :34

Challenges in counting words

- In a document we read a word (e.g., “eggs”)
  We need to keep a counter for every word
  and increment its counter.
- What data structure should we use?
  - Where do we store the counters?
  - How do we find the counter for “eggs” fast?
- Maybe a sorted array of words ordered lexicographically?
  - The English language has *half-a-million* words.
    Keeping a sorted array of 500K words is not fast for Google
  - How long would it take to find a word’s counter in it?
- With the **Hashing technique** the **order** is determined by some function of the **value** of the element to be stored

Let’s play darts (aka: let’s “hash the keys”)

Keys

Define **`hash(key)`**

“Brian” `hash(“Brian”)` → 1

“Stella” `hash(“Stella”)` → 5

“Ellen” `hash(“Ellen”)` → 4

“Takis” `hash(“Takis”)` → 6

“Christine” `hash(“Christine”)` → 2

“Lyn” `hash(“Lyn”)` → 11

“Orit” → ?

HashTable

0	
1	“Brian”
2	“Christine”
3	
4	“Ellen”
5	“Stella”
6	“Takis”
7	
8	
9	
10	
11	“Lyn”
12	

Hashing the keys

- To search for an entry in the table:

- Compute the *hash function* on the entry's key, then
- Use the value of the hash function as *an index* into the HashTable.

- Cool!! But: (Catherine, Caroline, Christine)
    - What if two or more keys *collide* on the same index?

- Then employ some method of ***collision resolution***.
    - Like what?

Load Factor **N items / capacity M**: When M is large enough?

- N/M = *load factor* of a hashtable
    - number of entries N in table
    - divided by the table capacity M.

- **Heuristics**:
    - If you know N, make M = 1.5 * N
    - If you do not know N, provide for **dynamic resizing:** Create larger HashTable and insert old elements into new

0	
1	“Brian”
2	“Christine”
3	
4	“Ellen”
5	“Stella”
6	“Takis”
7	
8	
9	
10	
11	“Lyn”
12	

Hash Functions: Mod-Division

- Good:

  hash(key) = f(key) % M

  M: capacity, a prime number

  f(): some function that produces a number,

  e.g., *f(key) = key.charAt(0) – 'A'*

- Better:

  *hash(key) = ((a * f(key) + b) % P) % M*

  prime $P >> N$ entries

  a, b: positive integers

What are the Pros and Cons of Hashing?

Pros

- Searching O( )
- Adding O( )
- Removing O( )

Cons

- You cannot keep adding new elements for ever!
  - Hash Table is an array, its size is fixed
  - When it needs space expansion capabilities: O( )

- There is no perfect hashing function!
  - Many items may end up colliding on same location,
  - Collisions require resolution policy

Even *Object* in Java has its own hashing function!

- The `java.lang.Object` class defines a method called *hashcode()* that returns an integer based on the memory location of the object
  - Object's default method is generally not very useful
- Classes (derived from `Object`) often override the inherited definition of *hashcode()* to provide their own version
- For example, `String` and `Integer` define their own *hashcode* methods
  - These more specific *hashcode* functions are more effective

Java's hashCode() methods

Java library implementations

```
public final class Integer
{
   private final int value;
   ...

   public int hashCode()
   {  return value;  }
}
```

```
public final class Boolean
{
   private final boolean value;
   ...

   public int hashCode()
   {
      if (value) return 1231;
      else       return 1237;
   }
}
```

```
public final class Double
{
   private final double value;
   ...

   public int hashCode()
   {
      long bits = doubleToLongBits(value);
      return (int) (bits ^ (bits >>> 32));
   }
}
```

convert to IEEE 64-bit representation;
xor most significant 32-bits
with least significant 32-bits

Warning: -0.0 and +0.0 have different hash codes

Java's hashCode() methods

Java library implementation

```
public final class String
{
   private final char[] s;
   ...

   public int hashCode()
   {
      int hash = 0;
      for (int i = 0; i < length(); i++)
         hash = s[i] + (31 * hash);
      return hash;
   }
}
```

i^{th} character of s

char	Unicode
…	…
'a'	97
'b'	98
'c'	99
…	…

- Horner's method to hash string of length L: L multiplies/adds.
- Equivalent to $h = s[0] \cdot 31^{L-1} + \ldots + s[L-3] \cdot 31^2 + s[L-2] \cdot 31^1 + s[L-1] \cdot 31^0$.

Ex.

```
String s = "call";
int code = s.hashCode();
```

$\leftarrow$ $3045982 = 99 \cdot 31^3 + 97 \cdot 31^2 + 108 \cdot 31^1 + 108 \cdot 31^0$

$= 108 + 31 \cdot (108 + 31 \cdot (97 + 31 \cdot (99)))$

(Horner's method)

Resolving Collisions

- Inevitably, if there are fewer buckets than keys, some keys will resolve to the same location regardless of the hash function we choose.

- In these cases, we must decide how to resolve collisions

Resolving Collisions idea #1: Separate Chaining

"Brian"

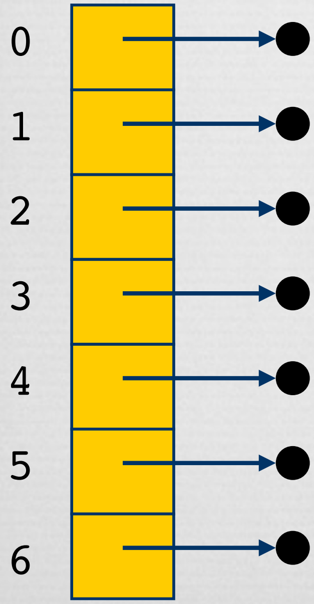
"Stella"

"Ellen"

"Lyn"

"Takis"

"Orit"



Resolve Open Addressing Collisions with Linear Probing

- When the index hashed to is occupied by a stranger, *probe* the next position.

- If that position is empty, we insert the entry, otherwise, we *probe* the next position and repeat.

H	A	S	H	I	N	G	I	S	F	U	N
8	1	19	8	9	14	7	9	19	6	21	14

1	2	3	4	5	6	7	8	9	10
A							H		

There is a problem though: Clustering

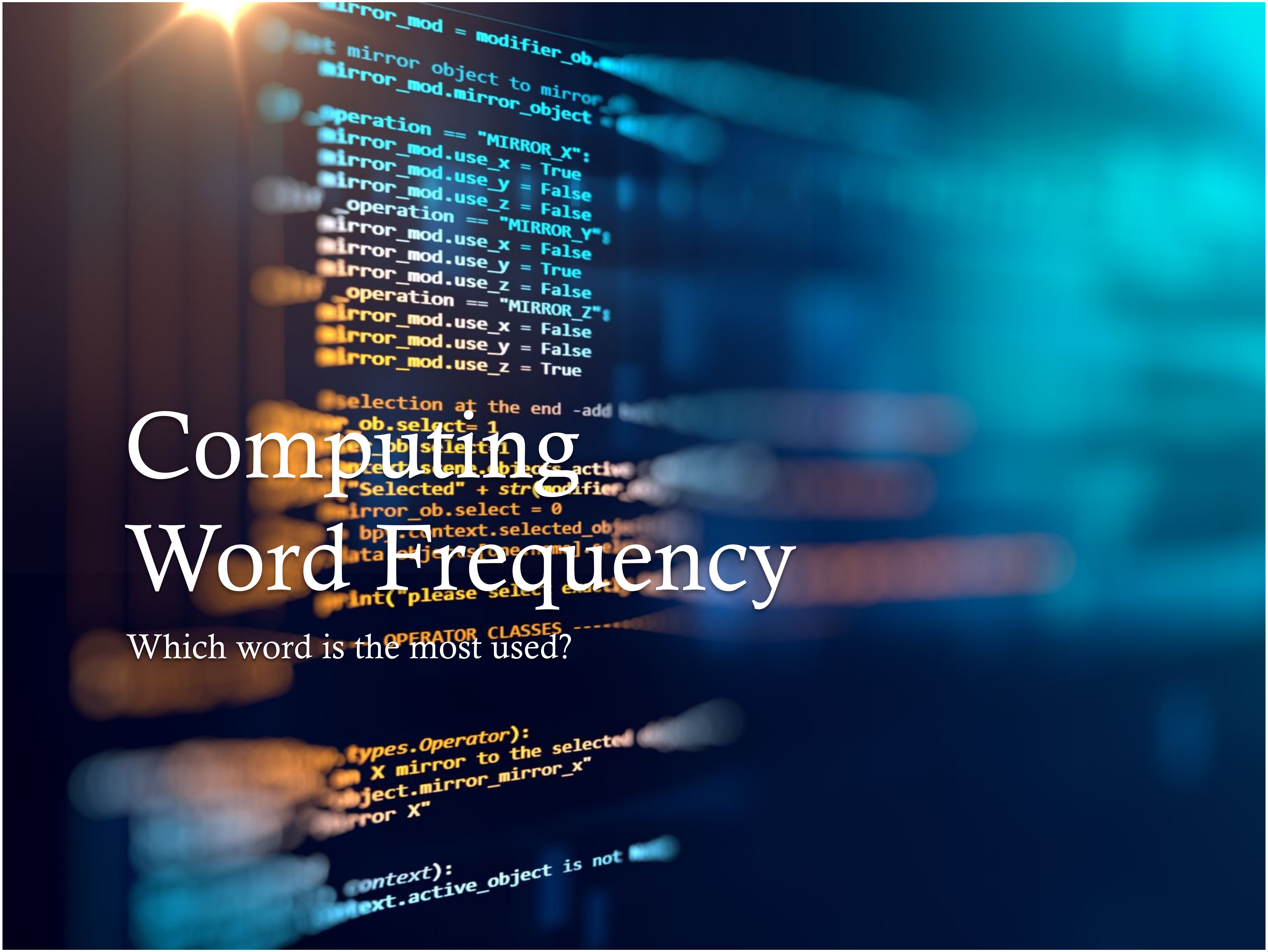
- As the table begins to fill up, more and more entries must be examined before the desired entry is found.

- Insertion of one entry may greatly increase the search time for others.

For example, consider H, S, H, I, ...

H	A	S	H	I	N	G	I	S	F	U	N
8	1	19	8	9	14	7	9	19	6	21	14

1	2	3	4	5	6	7	8	9	10
A							H		



Computing Word Frequency

Which word is the most used?

The Java `Hashtable<K,V>` Class

- Located in java.util
- Methods

```
- int size()
     // returns number of keys in table

- V get(Object key)
     // returns value to which specified key is mapped in table

- V put(K key, V value)
     // maps key to specified value in table

- boolean containsKey(Object key)
     // tests if the specified Object is a key in hash table

– V remove(Object key)
     // removes key and corresponding value from table

- …
```

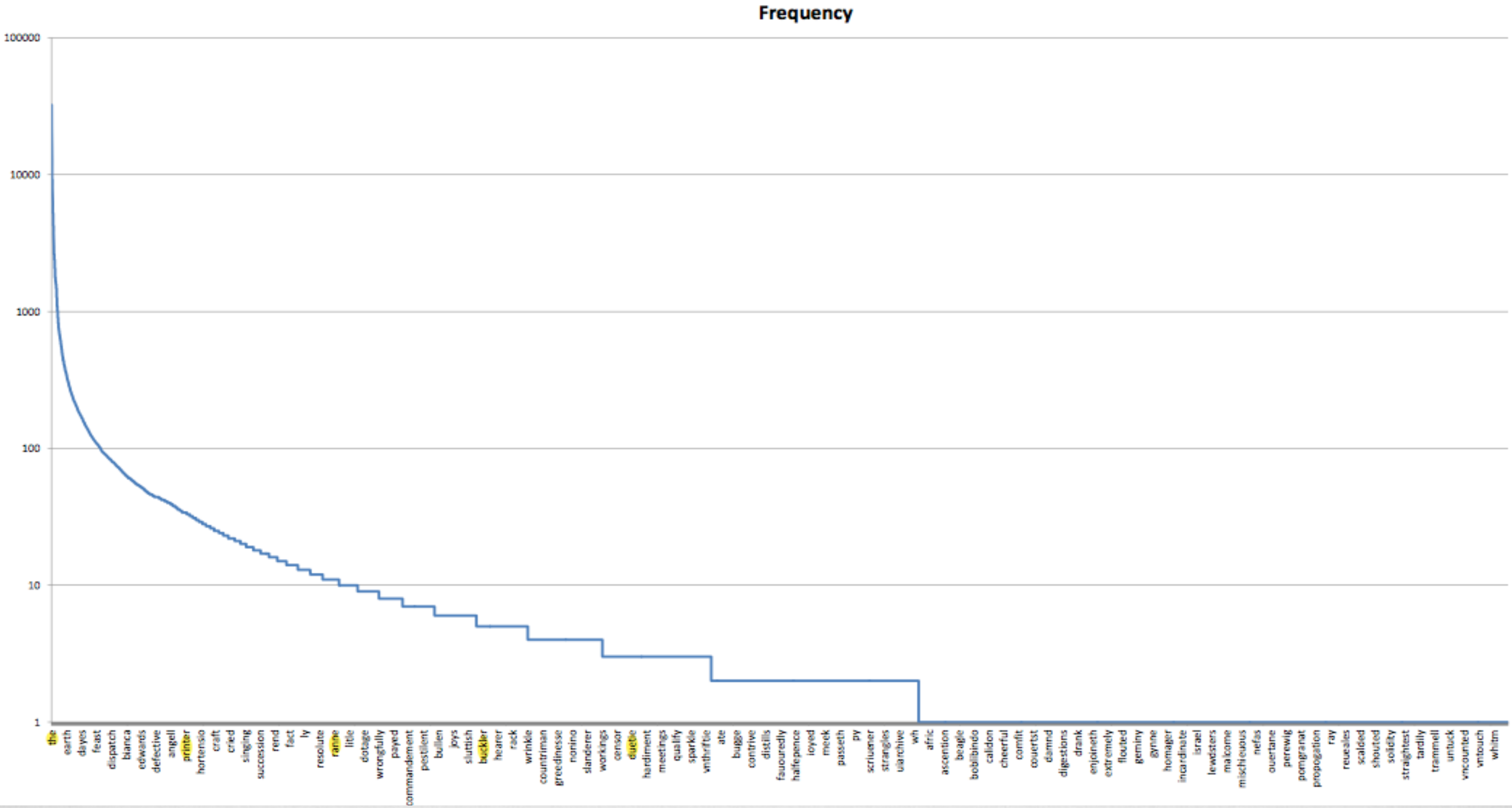
Basic Word Frequency pseudocode

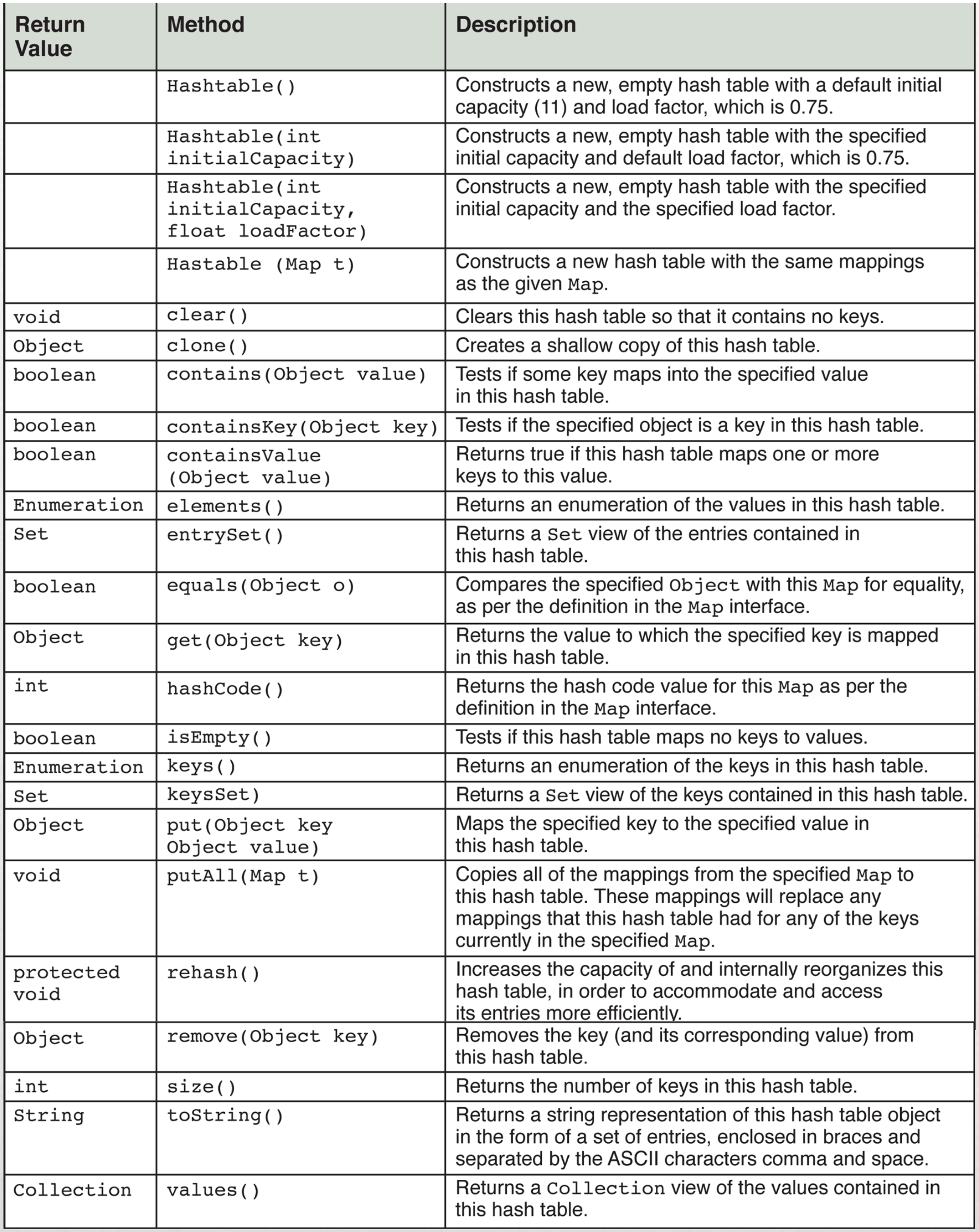
```
Count the number of times each word from an input document
appears in the document
       Define table = new Hashtable<String, Integer>();

      Start by reading the input document
      while (there are more words in the document) {
        read the next word
        if (the table contains already the word) {
          see how many times it has been seen before and
          add +1 to its frequency counter
        }
        else if it is the first time you've seen the word
        insert in the table a counter = 1 for this word
      }
At the end, we have counted all the frequencies of each word
```

Basic Word Frequency code

```
import java.util.Hashtable;
import java.io.File;
   Hashtable<String, Integer> table =
       new Hashtable<String, Integer>();

      Scanner reader = new Scanner(new File(filename));
      while (reader.hasNext()) {
        String word = reader.next();
        if (table.containsKey(word)) {
          int previousCount = table.get(word);
          table.put(word, previousCount+1);
        }
        else table.put(word, 1);
        totalWords++;
      }
      reader.close();
```

Words popular with Shakespeare





Return Value	Method	Description
	`Hashtable()`	Constructs a new, empty hash table with a default initial capacity (11) and load factor, which is 0.75.
	`Hashtable(int initialCapacity)`	Constructs a new, empty hash table with the specified initial capacity and default load factor, which is 0.75.
	`Hashtable(int initialCapacity, float loadFactor)`	Constructs a new, empty hash table with the specified initial capacity and the specified load factor.
	`Hastable (Map t)`	Constructs a new hash table with the same mappings as the given `Map`.
`void`	`clear()`	Clears this hash table so that it contains no keys.
`Object`	`clone()`	Creates a shallow copy of this hash table.
`boolean`	`contains(Object value)`	Tests if some key maps into the specified value in this hash table.
`boolean`	`containsKey(Object key)`	Tests if the specified object is a key in this hash table.
`boolean`	`containsValue (Object value)`	Returns true if this hash table maps one or more keys to this value.
`Enumeration`	`elements()`	Returns an enumeration of the values in this hash table.
`Set`	`entrySet()`	Returns a `Set` view of the entries contained in this hash table.
`boolean`	`equals(Object o)`	Compares the specified `Object` with this `Map` for equality, as per the definition in the `Map` interface.
`Object`	`get(Object key)`	Returns the value to which the specified key is mapped in this hash table.
`int`	`hashCode()`	Returns the hash code value for this `Map` as per the definition in the `Map` interface.
`boolean`	`isEmpty()`	Tests if this hash table maps no keys to values.
`Enumeration`	`keys()`	Returns an enumeration of the keys in this hash table.
`Set`	`keysSet)`	Returns a `Set` view of the keys contained in this hash table.
`Object`	`put(Object key Object value)`	Maps the specified key to the specified value in this hash table.
`void`	`putAll(Map t)`	Copies all of the mappings from the specified `Map` to this hash table. These mappings will replace any mappings that this hash table had for any of the keys currently in the specified `Map`.
`protected void`	`rehash()`	Increases the capacity of and internally reorganizes this hash table, in order to accommodate and access its entries more efficiently.
`Object`	`remove(Object key)`	Removes the key (and its corresponding value) from this hash table.
`int`	`size()`	Returns the number of keys in this hash table.
`String`	`toString()`	Returns a string representation of this hash table object in the form of a set of entries, enclosed in braces and separated by the ASCII characters comma and space.
`Collection`	`values()`	Returns a `Collection` view of the values contained in this hash table.

What if there are many files

```
import java.io.File;


// args[0] is the name of a directory
dir= new File(args[0] + "/");
// dir points to the directory’s contents
File[]files= dir.listFiles();
System.out.println(files.length + "files");
for(File f:files)
  if(!f.isHidden())
    process(f); // i.e. count word frequencies
```