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)?

<table>
<thead>
<tr>
<th></th>
<th>Add</th>
<th>Search</th>
<th>Remove</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted Array</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unsorted LinkedList</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sorted Array</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sorted LinkedList</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
How Search Engines work
They count words, and they find them fast...
How Search Engines Work

- **Web Documents**: Crawl the web to index documents.
- **Search engine servers**: Create word index.
- **Create word index**: Check frequencies.
- **Index & Frequencies**: Rank results.

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
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 eat them
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

... 
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”)

<table>
<thead>
<tr>
<th>Keys</th>
<th>Define hash(key)</th>
<th>HashTable</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Brian”</td>
<td>hash(“Brian”)</td>
<td>1</td>
</tr>
<tr>
<td>“Stella”</td>
<td>hash(“Stella”)</td>
<td>5</td>
</tr>
<tr>
<td>“Ellen”</td>
<td>hash(“Ellen”)</td>
<td>4</td>
</tr>
<tr>
<td>“Takis”</td>
<td>hash(“Takis”)</td>
<td>6</td>
</tr>
<tr>
<td>“Christine”</td>
<td>hash(“Christine”)</td>
<td>2</td>
</tr>
<tr>
<td>“Lyn”</td>
<td>hash(“Lyn”)</td>
<td>11</td>
</tr>
<tr>
<td>“Orit”</td>
<td></td>
<td>?</td>
</tr>
</tbody>
</table>
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 $\frac{N}{M}$ items / capacity $M$: When $M$ is large enough?

- $\frac{N}{M}$ = load factor of a hashtable
  - number of entries $N$ in table
  - divided by the table capacity $M$.

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

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

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

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

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

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

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

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

```java
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;
    }
}
```

<table>
<thead>
<tr>
<th>char</th>
<th>Unicode</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>'a'</td>
<td>97</td>
</tr>
<tr>
<td>'b'</td>
<td>98</td>
</tr>
<tr>
<td>'c'</td>
<td>99</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

• 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.**

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

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

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

(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.
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, ...

<table>
<thead>
<tr>
<th>H</th>
<th>A</th>
<th>S</th>
<th>H</th>
<th>I</th>
<th>N</th>
<th>G</th>
<th>I</th>
<th>S</th>
<th>F</th>
<th>U</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>1</td>
<td>19</td>
<td>8</td>
<td>9</td>
<td>14</td>
<td>7</td>
<td>9</td>
<td>19</td>
<td>6</td>
<td>21</td>
<td>14</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>H</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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
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
<table>
<thead>
<tr>
<th>Return Value</th>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Hashtable()</td>
<td>Constructs a new, empty hash table with a default initial capacity (11) and load factor, which is 0.75.</td>
</tr>
<tr>
<td></td>
<td>Hashtable(int initialCapacity)</td>
<td>Constructs a new, empty hash table with the specified initial capacity and default load factor, which is 0.75.</td>
</tr>
<tr>
<td></td>
<td>Hashtable(int initialCapacity, float loadFactor)</td>
<td>Constructs a new, empty hash table with the specified initial capacity and the specified load factor.</td>
</tr>
<tr>
<td></td>
<td>Hashable (Map t)</td>
<td>Constructs a new hash table with the same mappings as the given Map.</td>
</tr>
<tr>
<td></td>
<td>void clear()</td>
<td>Clears this hash table so that it contains no keys.</td>
</tr>
<tr>
<td></td>
<td>Object clone()</td>
<td>Creates a shallow copy of this hash table.</td>
</tr>
<tr>
<td></td>
<td>boolean contains(Object value)</td>
<td>Tests if some key maps into the specified value in this hash table.</td>
</tr>
<tr>
<td></td>
<td>boolean containsKey(Object key)</td>
<td>Tests if the specified object is a key in this hash table.</td>
</tr>
<tr>
<td></td>
<td>boolean containsValue (Object value)</td>
<td>Returns true if this hash table maps one or more keys to this value.</td>
</tr>
<tr>
<td></td>
<td>Enumeration elements()</td>
<td>Returns an enumeration of the values in this hash table.</td>
</tr>
<tr>
<td></td>
<td>Set entrySet()</td>
<td>Returns a Set view of the entries contained in this hash table.</td>
</tr>
<tr>
<td></td>
<td>boolean equals(Object o)</td>
<td>Compares the specified object with this Map for equality, as per the definition in the Map interface.</td>
</tr>
<tr>
<td></td>
<td>Object get(Object key)</td>
<td>Returns the value to which the specified key is mapped in this hash table.</td>
</tr>
<tr>
<td></td>
<td>int hashCode()</td>
<td>Returns the hash code value for this Map as per the definition in the Map interface.</td>
</tr>
<tr>
<td></td>
<td>boolean isEmpty()</td>
<td>Tests if this hash table maps no keys to values.</td>
</tr>
<tr>
<td></td>
<td>Enumeration keys()</td>
<td>Returns an enumeration of the keys in this hash table.</td>
</tr>
<tr>
<td></td>
<td>Set keySet()</td>
<td>Returns a Set view of the keys contained in this hash table.</td>
</tr>
<tr>
<td></td>
<td>Object put(Object key Object value)</td>
<td>Maps the specified key to the specified value in this hash table.</td>
</tr>
<tr>
<td></td>
<td>void putAll(Map t)</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>protected void rehash()</td>
<td>Increases the capacity of and internally reorganizes this hash table, in order to accommodate and access its entries more efficiently.</td>
</tr>
<tr>
<td></td>
<td>Object remove(Object key)</td>
<td>Removes the key (and its corresponding value) from this hash table.</td>
</tr>
<tr>
<td></td>
<td>int size()</td>
<td>Returns the number of keys in this hash table.</td>
</tr>
<tr>
<td></td>
<td>String toString()</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>Collection values()</td>
<td>Returns a Collection view of the values contained in this hash table.</td>
</tr>
</tbody>
</table>
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

What if there are many files

// args[0] is the name of a directory
// dir points to the directory’s contents