Hash Tables and Maps
How Search Engines Work

Document IDs
- Crawl the web
- Create word index
- Index & Freq’s

Search engine servers
- Rank results

User query

THE WEB
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 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 English words

- The English language has *half-a-million* terms. Any given text, however, has only a *few thousand* words.

- Keeping an array of 500K words “just in case” is not good
  - What data structure should we use?

- **Hashing** is the idea that *order* is determined by some function of the *value* of the element to be stored

- Like throwing darts on a board
Let’s play darts (aka: let’s “hash the keys”)

What is the hash function?

- “Brian” → hash(“Brian”) → 0
- “Stella” → hash(“Stella”) → 1
- “Ellen” → hash(“Ellen”) → 2
- “Takis” → hash(“Takis”) → 3
- “Christine” → hash(“Christine”) → 4
- “Lyn” → hash(“Lyn”) → 5

HashTable

<table>
<thead>
<tr>
<th></th>
<th>“Brian”</th>
<th>“Stella”</th>
<th>“Ellen”</th>
<th>“Takis”</th>
<th>“Christine”</th>
<th>“Lyn”</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>“Brian”</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

“Orit” → ?
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 Table.

Can two or more keys may *collide* on the same index?
  - Then employ some method of *collision resolution*.

What are the Pros and Cons of using Hashing?
Pros and Cons

Pros
• Searching $O(\ )$
• Inserting $O(\ )$
• Deleting $O(\ )$

Cons
• You cannot keep adding new elements for ever!
  • Table size is fixed (like an array)
  • Needs expansion capabilities ($O(?)$)
• Would be nice to have a perfect hashing function but many items may end up on same location
• Collisions need resolution policy
Load Factor: 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 Hash Table
  Insert old elements into new
Hash Functions: Division

• Good:
  \[ h(\text{hashCode}) = \text{hashCode} \mod M \]
  M: prime

• Better:
  \[ h(\text{hashCode}) = ((a \times \text{hashCode} + b) \mod p) \mod M \]
  p: prime >> N
  a, b: positive integers
Hashing Functions: Mid-square

- The key is multiplied by itself and then “extract” some digits from the middle of the result.
- For example, if our key is 4321
  - Multiply the key by itself yielding 18671041
  - Extract the needed three digits
- It is critical that the same three digits be extracted each time.
- We may also extract bits and then reconstruct an index from the bits.
Even Object has its own hashing function in Java!

- The java.lang.Object class defines a method called `hashCode()` that returns an integer based on the memory location of the object
  - This 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
Resolving Collisions

- If we are able to develop a perfect hashing function, then we do not need to be concerned about collisions or table size

- However, often we do not know the size of the dataset and are not able to develop a perfect hashing function

- In these cases, we must decide how to resolve collisions
Resolving Collisions idea #1: Separate Chaining

“Brian”
“Stella”
“Ellen”
“Lyn”
“Takis”
“Orit”
Separate Chaining

\[ \text{hash(“Brian”) \rightarrow 1} \]

“Brian”

“Stella”

“Ellen”

“Lyn”

“Takis”

“Orit”
Resolving Collisions idea #2: Open Addressing

Look for another open position in the table other than the one to which the element is hashed.

• Open addressing ($M >> N$):

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>aa</td>
<td>ab</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>673</td>
<td>674</td>
<td>675</td>
<td>zz</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• How are *collisions* are resolved with this technique?
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>0</th>
<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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>H</td>
<td></td>
</tr>
</tbody>
</table>


The Java Hashtable<\texttt{K}, \texttt{V}> Class

- Located in \texttt{java.util}
- Methods
  - \texttt{int size()}
    \hspace{1em} // returns number of keys in table
  - \texttt{V get(\texttt{Object key})}
    \hspace{1em} // returns value to which specified key is mapped in table
  - \texttt{V put(\texttt{K key}, \texttt{V value})}
    \hspace{1em} // maps key to specified value in table
  - \texttt{boolean containsKey(\texttt{Object key})}
    \hspace{1em} // tests if the specified Object is a key in hash table
  - \texttt{V remove(\texttt{Object key})}
    \hspace{1em} // removes key and corresponding value from table
  - ...
import java.util.*;
import java.io.*;

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>Hashtable (Map t)</td>
<td>Constructs a new hash table with the same mappings as the given Map.</td>
</tr>
<tr>
<td>void</td>
<td>clear()</td>
<td>Clears this hash table so that it contains no keys.</td>
</tr>
<tr>
<td>Object</td>
<td>clone()</td>
<td>Creates a shallow copy of this hash table.</td>
</tr>
<tr>
<td>boolean</td>
<td>contains(Object value)</td>
<td>Tests if some key maps into the specified value in this hash table.</td>
</tr>
<tr>
<td>boolean</td>
<td>containsKey(Object key)</td>
<td>Tests if the specified object is a key in this hash table.</td>
</tr>
<tr>
<td>boolean</td>
<td>containsValue(Object value)</td>
<td>Returns true if this hash table maps one or more keys to this value.</td>
</tr>
<tr>
<td>Enumeration</td>
<td>elements()</td>
<td>Returns an enumeration of the values in this hash table.</td>
</tr>
<tr>
<td>Set</td>
<td>entrySet()</td>
<td>Returns a Set view of the entries contained in this hash table.</td>
</tr>
<tr>
<td>boolean</td>
<td>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>Object</td>
<td>get(Object key)</td>
<td>Returns the value to which the specified key is mapped in this hash table.</td>
</tr>
<tr>
<td>int</td>
<td>hashCode()</td>
<td>Returns the hash code value for this Map as per the definition in the Map interface.</td>
</tr>
<tr>
<td>boolean</td>
<td>isEmpty()</td>
<td>Tests if this hash table maps no keys to values.</td>
</tr>
<tr>
<td>Enumeration</td>
<td>keys()</td>
<td>Returns an enumeration of the keys in this hash table.</td>
</tr>
<tr>
<td>Set</td>
<td>keySet()</td>
<td>Returns a Set view of the keys contained in this hash table.</td>
</tr>
<tr>
<td>Object</td>
<td>put(Object key, Object value)</td>
<td>Maps the specified key to the specified value in this hash table.</td>
</tr>
<tr>
<td>void</td>
<td>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>protected</td>
<td>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>Object</td>
<td>remove(Object key)</td>
<td>Removes the key (and its corresponding value) from this hash table.</td>
</tr>
<tr>
<td>int</td>
<td>size()</td>
<td>Returns the number of keys in this hash table.</td>
</tr>
<tr>
<td>String</td>
<td>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>Collection</td>
<td>values()</td>
<td>Returns a Collection view of the values contained in this hash table.</td>
</tr>
</tbody>
</table>
Processing a directory of data files

```java
import java.io.*;

// args[0] is the name of a directory
dir = new File(args[0] + "/");
// dis 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);
```