Hash Tables and Maps

How Search Engines Work

- Document IDs
- Search engine servers
- Index & Freq’s
- THE WEB
- crawl the web
- Create word index
- Rank results
- user query

Create word index
Creating Word Index: Word Frequency

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Hash Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Brian”</td>
<td>hash(“Brian”)</td>
</tr>
<tr>
<td>“Stella”</td>
<td>hash(“Stella”)</td>
</tr>
<tr>
<td>“Ellen”</td>
<td>hash(“Ellen”)</td>
</tr>
<tr>
<td>“Takis”</td>
<td>hash(“Takis”)</td>
</tr>
<tr>
<td>“Christine”</td>
<td>hash(“Christine”)</td>
</tr>
<tr>
<td>“Lyn”</td>
<td>hash(“Lyn”)</td>
</tr>
<tr>
<td>“Orit”</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>HashTable</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>12</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 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 $M = 1.5 \times N$

  • If you do not know $N$, provide for dynamic resizing:
    Create larger Hash Table
    Insert old elements into new
Hash Functions: Division

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

- Better:
  \[ h(\text{hashCode}) = ((a \times \text{hashCode} + b) \mod p) \mod M \]

  \( p: \text{prime} \gg N \)
  
  \( a, b: \text{positive integers} \)

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

“Brian” hash(“Brian”) 1
“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 \((M \gg N)\):

\[
\begin{align*}
\text{aa} & \quad \text{ab} & \quad \text{\ldots} & \quad \text{zz} \\
0 & & 1 & 2 & 3 & 4 & 673 & 674 & 675
\end{align*}
\]

• 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.

\[
\begin{align*}
\text{HA} & \quad \text{S} & \quad \text{H} & \quad \text{I} & \quad \text{N} & \quad \text{G} & \quad \text{I} & \quad \text{S} & \quad \text{F} & \quad \text{U} & \quad \text{N} \\
8 & & 1 & 19 & 8 & 9 & 14 & 7 & 9 & 19 & 6 & 21 & 14
\end{align*}
\]
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>A</td>
<td></td>
<td></td>
<td></td>
<td>H</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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
  - ...

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

![Graph showing frequency of word usage]
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);
```