Hash Tables and Maps

How Search Engines Work

- Document IDs
  - Index & Freq’s
  - Search engine servers
  - Rank results
  - crawl the web

User query

Create word index
Creating Word Index: Word Frequency

<table>
<thead>
<tr>
<th>Word</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>59</td>
</tr>
<tr>
<td>am</td>
<td>16</td>
</tr>
<tr>
<td>and</td>
<td>25</td>
</tr>
<tr>
<td>anywhere</td>
<td>8</td>
</tr>
<tr>
<td>are</td>
<td>2</td>
</tr>
<tr>
<td>be</td>
<td>4</td>
</tr>
<tr>
<td>boat</td>
<td>3</td>
</tr>
<tr>
<td>box</td>
<td>7</td>
</tr>
<tr>
<td>car</td>
<td>7</td>
</tr>
<tr>
<td>could</td>
<td>14</td>
</tr>
<tr>
<td>dark</td>
<td>7</td>
</tr>
<tr>
<td>do</td>
<td>37</td>
</tr>
<tr>
<td>eat</td>
<td>25</td>
</tr>
<tr>
<td>eggs</td>
<td>11</td>
</tr>
<tr>
<td>fox</td>
<td>7</td>
</tr>
<tr>
<td>goat</td>
<td>4</td>
</tr>
<tr>
<td>...</td>
<td>4</td>
</tr>
<tr>
<td>try</td>
<td>4</td>
</tr>
<tr>
<td>will</td>
<td>21</td>
</tr>
<tr>
<td>with</td>
<td>19</td>
</tr>
<tr>
<td>would</td>
<td>26</td>
</tr>
<tr>
<td>you</td>
<td>34</td>
</tr>
</tbody>
</table>

Challenges in counting English words

- In a document we read a word (e.g., "car")
  We need to increment a counter for "car".

- What data structure should we use?
  - Where do we store the counters?
  - How do we find the counter for “car” 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 good for Google
  - How long would it take to find an item in it?

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

- Like throwing darts and hitting bull’s eye every time!
Let’s play darts (aka: let’s “hash the keys”)

- Brian → hash("Brian") → 1
- Stella → hash("Stella") → 5
- Ellen → hash("Ellen") → 4
- Takis → hash("Takis") → 6
- Christine → hash("Christine") → 2
- Lyn → hash("Lyn") → 11

What is the hash function?

Any problems with our hash?

- 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 → ?

"Any problems with our hash?"
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.

• What if two or more keys collide on the same index?
  – Then employ some method of collision resolution.
  – (Catherine, Cibele, Christine)

What are the Pros and Cons of using Hashing?

Pro and Cons of using Hashing

Pros
• Searching $O(1)$
• Inserting $O(1)$
• Deleting $O(1)$

Cons
• You cannot keep adding new elements for ever!
  – HashTable is an array, size is fixed
  – Needs expansion capabilities ($O(?)$)

• Many items may end up hashing on same location, there is no perfect hashing function!
  – Collisions require 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 HashTable and insert old elements into new

---

Hash Functions: Division

- Good:
  \[ h(key) = key \mod M \]
  $M$: prime

- Better:
  \[ h(key) = ((a*key + b) \mod p) \mod M \]
  $p$: 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 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.
  - 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.
Implementing hash code: integers, booleans, and doubles

Java library implementations

```java
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;
    }
}
```

```java
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

Implementing hash code: strings

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;
    }
}
```

- 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”;
```java
int code = s.hashCode();
```

```
3045982 = 99 \cdot 31^1 + 97 \cdot 31^0 + 108 \cdot 31^2 + 108 \cdot 31^3
= 108 + 31 \cdot (108 + 31 \cdot (97 + 31 \cdot (99)))
(Horner’s method)
```
Inevitably, some keys will resolve to the same bucket regardless of the hash function we choose if there are fewer buckets than keys.

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

Resolving Collisions idea #1: **Separate Chaining**

“Brian”
“Stella”
“Ellen”
“Lyn”
“Takis”
“Orit”
Separate Chaining: HashTable of LinkedLists

- "Brian" \(\text{hash("Brian")}\) → 1
- "Stella"
- "Ellen"
- "Lyn"
- "Takis"
- "Orit"

Separate Chaining: HashTable of LinkedLists

- "Brian" \(\text{hash("Brian")}\) → 1
- "Stella" \(\text{hash("Stella")}\) → 4
- "Ellen"
- "Lyn"
- "Takis"
- "Orit"
Separate Chaining: HashTable of LinkedLists

- "Brian" hash("Brian") 1
- "Stella" hash("Stella") 4
- "Ellen" hash("Ellen") 4
- "Lyn" hash("Lyn") 4
- "Takis"
- "Orit"

Separate Chaining: HashTable of LinkedLists

- "Brian" hash("Brian") 1
- "Stella" hash("Stella") 4
- "Ellen" hash("Ellen") 4
- "Lyn" hash("Lyn") 4
- "Takis"
- "Orit"
Separate Chaining: HashTable of LinkedLists

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 ($M >> N$):

- 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, ...
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 code

```java
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.

**Hashtable (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 element();** Returns anenumeration of the values in this hash table.

**Set entrySet();** Returns a view of the entries contained in this hash table.

**Boolean equals(Object o);** Compares the specified object with this hash table for equality as per the definition in the Map interface.

**Object get(Object key);** Returns the value to which this specified key is mapped in this hash table.

**Object hashCode();** Returns the hash code value for this hash 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 keySet();** 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 m);** Copies all of the mappings from the specified map to this hash table. This operation will rehash 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 new, or more efficient, mappings.

**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.
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);