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”)

```
<table>
<thead>
<tr>
<th>Name</th>
<th>Hash Function</th>
<th>Index</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>
</tbody>
</table>
```

What is the hash function?

Any problems with our hash?

```
<table>
<thead>
<tr>
<th>Name</th>
<th>Hash Function</th>
<th>Index</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>
</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 collide on the same index?
  - Then employ some method of collision resolution.
  - (Catherine, Cibele, Christine)

What are the Pros and Cons of using Hashing?

Pros and Cons

**Pros**
- Searching O(1)
- Inserting O(1)
- Deleting O(1)

**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?

- \( N/M = \text{load factor} \) 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:**
  \[ h(\text{hashCode}) = ((a \times \text{hashCode} + b) \mod p) \mod M \]
  \( p: \text{prime} \gg N \)
  \( a, b: \text{positive integers} \)

- 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

```java
static int hash(int h) {
    h ^= (h >>> 20) ^ (h >>> 12);
    return h ^ (h >>> 7) ^ (h >>> 4);
}
static int indexFor(int h, int length) {
    return h & (length-1);
}
public void put(K key, V value) {
    int hash = hash(key.hashCode());
    int i = indexFor(hash, table.length);
    // Store in bucket i of hashtable.
}
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.

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**

```
  0  1  2  3  4  5  6
```

```
Brian
Stella
Ellen
Lyn
Takis
Orit
```

Separate Chaining

```
  0  1  2  3  4  5  6
```

```
Brian hash("Brian")
Stella
Ellen
Lyn
Takis
Orit
```
Separate Chaining

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

Separate Chaining

“Brian” hash("Brian") 1
“Stella” hash("Stella") 4
“Ellen” hash("Ellen") 4
“Lyn” hash("Lyn") 4
“Takis” hash("Takis") 5
“Orit”

Separate Chaining

“Brian” hash("Brian") 1
“Stella” hash("Stella") 4
“Ellen” hash("Ellen") 4
“Lyn” hash("Lyn") 4
“Takis” hash("Takis") 5
“Orit”

Separate Chaining

“Brian” hash("Brian") 1
“Stella” hash("Stella") 4
“Ellen” hash("Ellen") 4
“Lyn” hash("Lyn") 4
“Takis” hash("Takis") 5
“Orit”

Separate Chaining

“Brian” hash("Brian") 1
“Stella” hash("Stella") 4
“Ellen” hash("Ellen") 4
“Lyn” hash("Lyn") 4
“Takis” hash("Takis") 5
“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>aa</th>
<th>ab</th>
<th></th>
<th></th>
<th>...</th>
<th>673</th>
<th>674</th>
<th>675</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></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.

<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>
<th>A</th>
<th>H</th>
</tr>
</thead>
</table>

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>
<th>A</th>
<th>H</th>
</tr>
</thead>
</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
  - ...
Basic Word Frequency code

```java
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();
```