§ The English language has \textit{half-a-million} terms. Any given text, however, has only a \textit{few thousand} words.

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

\textbf{Hashing} is the idea that \textit{order} is determined by some function of the \textit{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>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brian</td>
<td>hash(&quot;Brian&quot;)</td>
<td>1</td>
</tr>
<tr>
<td>Stella</td>
<td>hash(&quot;Stella&quot;)</td>
<td>5</td>
</tr>
<tr>
<td>Ellen</td>
<td>hash(&quot;Ellen&quot;)</td>
<td>4</td>
</tr>
<tr>
<td>Takis</td>
<td>hash(&quot;Takis&quot;)</td>
<td>6</td>
</tr>
<tr>
<td>Christine</td>
<td>hash(&quot;Christine&quot;)</td>
<td>2</td>
</tr>
<tr>
<td>Lyn</td>
<td>hash(&quot;Lyn&quot;)</td>
<td>11</td>
</tr>
<tr>
<td>Orit</td>
<td></td>
<td>?</td>
</tr>
</tbody>
</table>

HashTable

0 1 2 3 4 5 6 7 8 9 10 11 12

“Brian”

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 $O(1)$
• 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?

• $N/M = load\ factor$ 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

Orit

<table>
<thead>
<tr>
<th>Name</th>
<th>Hash Function</th>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brian</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stella</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ellen</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Takis</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Christine</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lyn</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

0 1 2 3 4 5 6 7 8 9 10 11 12

“Brian”

“Christine”

“Ellen”

“Stella”

“Takis”

“Lyn”

“Orit”
Hash Functions: Division

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

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

• 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

• The \textit{java.lang.Object} class defines a method called \texttt{hashCode()} that returns an integer based on the memory location of the object
  * This is generally not very useful

• Classes derived from \texttt{Object} often override the inherited definition of \texttt{hashCode} to provide their own version

• For example, String and Integer define their own \texttt{hashCode} methods
  * These more specific \texttt{hashCode} functions are more effective

• If we are able to develop a \texttt{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

Separate Chaining

- “Brian”
- “Stella”
- “Ellen”
- “Lyn”
- “Takis”
- “Orit”

Resolving Collisions idea #2: Open Addressing

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>2</th>
<th>3</th>
<th>4</th>
<th>...</th>
<th>673</th>
<th>674</th>
<th>675</th>
</tr>
</thead>
</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, ...

```
H A S H I N G I S F U N
0 1 2 3 4 5 6 7 8 9 10

| A | | | | H |
```

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.*;

public class Frequency {
    public static void main(String[] args) {
        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();
    }
}
```

```
Frequency
```

```text
A
```
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);