Hash Tables

As a change from the structures we've been looking at so far, hash tables are almost refreshingly simple.

Think of a collection of labelled buckets, and as each fresh object comes along, we apply some clever hash function to the object which yields the label of the bucket we should put it in. Then putting stuff into this structure is easy, and finding an object merely involves looking in the correctly labelled bucket.

We assume that each object has an associated key, so that if object \( e \) has key \( k \), then \( e \) is stored in position \( f(k) \) of the hash table (possibly merely an array). To find \( e \), look in position \( f(k) \).

As examples:

1. Store students in chairs, \( f(k) = \) DNA code of student \( \Rightarrow \) lots of chairs!

2. Store students in chairs, \( f(k) = \) day/month of birthday \( \Rightarrow \) 367 chairs \& some collisions.

3. Store students in chairs, \( f(k) = \) weight to nearest pound \( \Rightarrow \) lots of collisions?

Clearly we don't want to have lots of wasted storage, but we also want to spread the distribution of objects as evenly as possible, and may also have to resolve collisions.
Suppose we are working with a table which can hold \( M \) objects (treat this as an array), so the available results from applying the hash function \( f \) to keys associated with these objects is an integer in \( 0, 1, 2, \ldots, M-1 \).

In an ideal sense, each answer would appear equally often as \( f \) is applied to all of our possible objects—that way, no one bucket is unfairly overloaded.

**Example 1:** let the keys be doubles with \( 0 \leq k < 1 \), then define

\[
f(k) := \text{Math. floor}(M \times k).
\]

If keys are doubles with \( a \leq k < b \), then define

\[
f(k) := \text{Math. floor}(\frac{M \times k}{b-a}).
\]

**Example 2:** let the keys be ints, then define

\[
f(k) := k \% M.
\]

**Example 3:** let the keys be Strings, then convert to an array of chars, rewrite each char as an ASCII int, then assemble into a big integer and apply example 2. E.g.,

\[
\text{cat} \rightarrow 99\,97\,116 \rightarrow 99(128)^2 + 97(128) + 116 \\
\rightarrow 1634548 \% M
\]
A little thought shows that using the modular hash function of examples 2 or 3 with \( M \) having lots of factors will lead to a very uneven distribution over the table. Far better here is to choose \( M \) near the desired size where \( M \) is actually prime. (Rather than constructing primes on the fly, have some useful ones stored in a convenient array.

Actually, the String hash function of example 3 could easily get really nasty since a longer string would easily waste space before we even got to the \( \% M \) part of the computation. Far better would be to continually reduce the running total mod \( M \), for example...

```java
int hash(String s, int M) {
    int h = 0, a = 127; // sneaky prime trick
    for (int i = 0; i < s.length(); i++) {
        int temp = Character.getNumericValue((char) s.charAt(i));
        h = (a * h + temp) \% M;
    }
    return h;
}
```

Given that we might have a decent hash function, we must still resolve the issue of collisions. There are many approaches to this, but the easiest (and the one of primary concern to us) is that of separate chaining.
This is a fancy name for doing what is really the most obvious thing... if a collision occurs, build a linked list at the site.

As an example, consider the collection...

<table>
<thead>
<tr>
<th>Object</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>...</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>...</td>
<td>23</td>
<td>24</td>
<td>25</td>
</tr>
</tbody>
</table>

with hash function... \( f(k) = k \mod 7 \)

applied to the text (space-less for illustration)... Once upon a time there was an amazingly yummy box of Leonidas chocolates which as if with a table of 7 'buckets', this produces...

```
1 - P - I - W - I - B - I - W - I - I -
2 - C - X - C - C - C - C -
3 - R - Y - Y - Y - D -
4 - E - E - E - E - S - E - L - L - E - S - L - E - S - S -
5 - T - M - T - M - M - M - T - F -
6 - N - U - N - N - N - G - U - N -
```

Try searching for 'L' and for 'O' — it's fast!