Searching Lists

- There are many instances when one is interested in storing and searching a list:
  - A phone company wants to provide caller ID: Given a phone number a name is returned.
  - Somebody who plays the *Lotto* thinks they can increase their chance to win if they keep track of the winning number from the last 3 years.

- Both of these have simple solutions using arrays, linked lists, binary trees, etc.

- Unfortunately, none of these solutions is adequate, as we will see.
Problematic List Searching

Here are some solutions to the problems:

• For caller ID:
  - Use an array indexed by phone number.
    * Requires one operation to return name.
    * An array of size 1,000,000,000 is needed.
  - Use a linked list.
    * This requires $O(n)$ time and space, where $n$ is the number of actual phone numbers.
    * This is the best we can do space-wise, but the time required is horrendous.
  - A balanced binary tree: $O(n)$ space and $O(\log n)$ time. Better, but not good enough.

• For the Lotto we could use the same solutions.
  - The number of possible lotto number is 15,890,700 for a 6/50 lotto.
  - The number of entries stored is on the order of 1000, assuming a daily lotto.
The Hash Table Solution

- A **Hash table** is similar to an array:
  - Has fixed size $m$.
  - An item with key $k$ goes into index $h(k)$, where $h$ is a function from the keyspace to \(\{0, 1, \ldots, m - 1\}\)

- The function $h$ is called a **hash function**.

- We call $h(k)$ the **hash value** of $k$.

- A **hash table** allows us to store values from a large set in a small array in such a way that searching is fast.

- The space required to store $n$ numbers in a hash table of size $m$ is $O(m + n)$. *Notice that this does not depend on the size of the keyspace.*

- The average time required to insert, delete, and search for an element in a hash table is $O(1)$.

- Sounds perfect. Then what’s the problem? Let’s look at an example.
Hash Table Example

- I have 5 friends whose phone numbers I want to store in a hash table of size 8. Their names and number are:

<table>
<thead>
<tr>
<th>Name</th>
<th>Phone Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Susy Olson</td>
<td>555-1212</td>
</tr>
<tr>
<td>Sarah Skillet</td>
<td>555-4321</td>
</tr>
<tr>
<td>Ryan Hamster</td>
<td>545-3241</td>
</tr>
<tr>
<td>Justin Case</td>
<td>555-6798</td>
</tr>
<tr>
<td>Chris Lindmeyer</td>
<td>535-7869</td>
</tr>
</tbody>
</table>

- I use as a hash function $h(k) = k \mod 8$, where $k$ is the phone number viewed as a 7-digit decimal number.

- Notice that

\[ 5554321 \mod 8 = 5453241 \mod 8 = 1. \]

But I can’t put *Sarah Skillet* and *Ryan Hamster* both in the position 1.

- Can we fix this problem?
Hash Table Problems

- The problem with hash tables is that two keys can have the same hash value. This is called collision.
- There are several ways to deal with this problem:
  - Pick the hash function \( h \) to minimize the number of collisions.
  - Implement the hash table in a way that allows keys with the same hash value to all be stored.
- The second method is almost always needed, even for very good hash functions. Why?
- We will talk about 2 collision resolution techniques:
  - Chaining
  - Open Addressing
Hash Functions

- Most hash functions assume the keys come from the set \( \{0, 1, \ldots\} \).

- If the keys are not natural numbers, some method must be used to convert them.
  - Phone number and ID numbers can be converted by removing the hyphens.
  - Characters can be converted using ASCII.
  - Strings can be converted by converting each character using ASCII, and then interpreting the string of natural numbers as if it were stored base 128.

- We need to choose a good hash function.

- A hash function is good if
  - it can be computed quickly, and
  - the keys are distributed uniformly throughout the table.
Some Good Hash Functions

- **The division method:**
  \[ h(k) = k \mod m \]
  - The modulus \( m \) must be chosen carefully.
  - Powers of 2 and 10 can be bad. Why?
  - Prime numbers not too close to powers of 2 are a good choice.
  - We can pick \( m \) by choosing an appropriate prime number that is close to the table size we want.

- **The multiplication method:** Let \( A \) be a constant with \( 0 < A < 1 \). Then we use
  \[ h(k) = \lfloor m(kA \mod 1) \rfloor, \]
  where \( "kA \mod 1" \) means the fractional part of \( kA \).
  - The choice of \( m \) is not as critical here.
  - We can choose \( m \) to make the implementation easy and/or fast.
Universal Hashing

- Let $x$ and $y$ be distinct keys.
- Let $\mathcal{H}$ be a set of hash functions.
- $\mathcal{H}$ is called universal if the number of functions $h \in \mathcal{H}$ for which $h(x) = h(y)$ is precisely $|\mathcal{H}|/m$.
- In other words, if we pick a random $h \in \mathcal{H}$, the probability that $x$ and $y$ collide under $h$ is $1/m$.
- Universal hashing can be useful in many situations.
  - You are asked to come up with a hashing technique for your boss.
  - Your boss tells you that after you are done, he will pick some keys to hash.
  - If you get too many collisions, he will fire you.
  - If you use universal hashing, the only way he can succeed is by getting lucky. Why?
Collision Resolution: Chaining

- With *chaining*, we set up an array of links, indexed by the hash values, to lists of items with the same hash value.

  ![Diagram of chaining]

- Let $n$ be the number of keys, and $m$ the size of the hash table. Define the **load factor** $\alpha = n/m$.

- Successful and unsuccessful searching both take time $\Theta(1 + \alpha)$ on average, assuming simple uniform hashing.

- By **simple uniform hashing** we mean that a key has equal probability to hash into any of the $m$ slots, independent of the other elements of the table.
Collision Resolution: Open Addressing

- With Open Addressing, all elements are stored in the hash table.
- This means several things
  - Less memory is used than with chaining since we don’t have to store pointers.
  - The hash table has an absolute size limit of $m$. Thus, planning ahead is important when using open addressing.
  - We must have a way to store multiple elements with the same hash value.
- Instead of a hash function, we need to use a probe sequence. That is, a sequence of hash values.
- We go through the sequence one by one until we find an empty position.
- For searching, we do the same thing, skipping values that do not match our key.
Probe Sequences

- We define our hash functions with an extra parameter, the probe number $i$. Thus our hash functions look like $h(k, i)$.
- We compute $h(k, 0)$, $h(k, 1)$, $\ldots$, $h(k, i)$ until $h(k, i)$ is empty.
- The hash function $h$ must be such that the probe sequence $\langle h(k, 0), \ldots h(k, m - 1) \rangle$ is a permutation of $\langle 0, 1, \ldots, m - 1 \rangle$. Why?
- Insertion and searching are fairly quick, assuming a good implementation.
- Deletion can be a problem because the probe sequence can be complex.
- We will discuss 2 ways of defining probe sequences:
  - Linear Probing
  - Double Hashing
Linear Probing

- Let $g$ be a hash function.
- When we use linear probing, the probe sequence is computed by
  \[ h(k, i) = (g(k) + i) \mod m. \]
- When we insert an element, we start with the hash value, and proceed element by element until we find an empty slot.
- For searching, we start with the hash value and proceed element by element until we find the key we are looking for.
- **Example:** Let $g(k) = k \mod 13$. We will insert the following keys into the hash table:
  18, 41, 22, 44, 59, 32, 31, 73
- **Problem:** The values in the table tend to *cluster.*
Double Hashing

- With **double hashing** we use two hash functions.
- Let $h_1$ and $h_2$ be hash functions.
- We define our probe sequence by

  $$ h(k, i) = (h_1(k) + i \times h_2(k)) \mod m. $$

- We must ensure that $m$ and $h_2(k)$ are relatively prime for all values of $k$. Why?
- One method to do this is to pick $m = 2^n$ for some $n$, and ensure that $h_2(k)$ is always odd.
- Double hashing tends to distribute keys more uniformly than linear probing.
Double Hashing Example

- Let $h_1 = k \mod 13$
- Let $h_2 = 1 + (k \mod 8)$, and
- Let the hash table have size 13.
- Then our probe sequence is defined by

$$h(k, i) = (h_1(k) + i \times h_2(k)) \mod 13$$

$$= (k \mod 13 + i(1 + (k \mod 8))) \mod 13$$

- Insert the following keys into the table:

18, 41, 22, 44, 59, 32, 31, 73
Open Addressing: Performance

- Again, we set $\alpha = n/m$. This is the average number of keys per array index.
- Note that $\alpha \leq 1$ for open addressing.
- We assume a good probe sequence has been used. That is, for any key, each permutation of $\langle 0, 1 \ldots, m - 1 \rangle$ is equally likely as a probe sequence.
- The average number of probes for insertion or unsuccessful search is at most $1/(1 - \alpha)$.
- The average number of probes for a successful search is at most

$$\frac{1}{\alpha} \ln \frac{1}{1 - \alpha} + \frac{1}{\alpha}.$$
Chaining or Open Addressing?

Hashing Performance: Chaining Verses Open Addressing

Ch: S
Ch: I
OA: I/US
OA: SS

\[ 1 + x \]
\[ \frac{1}{1-x} \]
\[ \frac{1}{x} \log \left( \frac{1}{1-x} \right) + \frac{1}{x} \]