Hash Tables
Table naïve array implementation

- “Direct addressing”
- Worst case $O(1)$ access cost
- But likely to waste space

<table>
<thead>
<tr>
<th>K</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>3</td>
<td>B</td>
</tr>
<tr>
<td>5</td>
<td>C</td>
</tr>
</tbody>
</table>

0 1 2 3 4 5 6
X A X B X C X
Hashing

- \( U \): Set of all possible keys
- \( H \): Set of all possible hashes
- \( K \): Set of actually used keys

- E.g. Division hash: \( h(x) = x \mod y \)

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>C</td>
</tr>
<tr>
<td>3</td>
<td>D</td>
</tr>
<tr>
<td>8</td>
<td>B</td>
</tr>
<tr>
<td>9</td>
<td>A</td>
</tr>
</tbody>
</table>

\( y = 4 \)

\[ 0 \ 1 \ 2 \ 3 \]

| B | A | C | D |

\( \text{size}(U) > \text{size}(H) \)

\( O(1) \) lookup

Division is slow
Hashing

- E.g. Multiplication hash: \( h(x) = \lfloor k(ax \mod 1) \rfloor \)
  - Multiply key by a \( \Rightarrow \) F.P. number
  - Take fractional part
  - Multiply by \( k \)

\[
\begin{align*}
h(123456) &= \lfloor 41.155 \rfloor \\
&= 41 \\
h(123457) &= 6221 \\
h(123458) &= 2401
\end{align*}
\]

- \( k = 10000 \)
- \( a = \frac{\sqrt{5} - 1}{2} \)
Uniform Hashing

- Any analysis is going to be dependent on the hash function properties
- We usually consider uniform hashing:

A given key \( x \) has a hash \( h(x) \) which is equally probable to be any member of the set \( H \)
Avoiding Collisions

- Since $\text{size}(H) < \text{size}(U)$ multiple keys must map to the same hash value: **Collisions**
- Choose a good hash function
  - The more random the output, the less likely we will have collisions
  - Still going to happen though!

"good" depend on input data.
Chaining

- Each hash table slot is actually a linked list of keys

- Insertion
  \[ O(1) \text{ add to end/start list.} \]
  \[ \text{splice out } O(\text{size of list}) \]
  \[ \Rightarrow O(\frac{n}{h}) \]

- Deletion

- Search

Worst-case: all hash to same value \( \Rightarrow O(n) \)

Average: Assume uniform hashing.
Chaining

- Search

**Fails**

\[
\text{Time} = \text{time to traverse list} + \text{time to hash}
\]

\[
= O\left(\frac{n}{m}\right) + O(1)
\]

\[
\Rightarrow \quad \text{Time} = O(1 + \alpha)
\]

\[
\Rightarrow \quad n = \frac{n}{m}
\]

**Succeeds**

Imagine searching for the \(i\)th added key.

\[
\text{Average} = \frac{1}{n} \sum_{i=1}^{n} \left(\frac{i-1}{m} + 1\right)
\]

\[
= O(1 + \alpha)
\]
Open Addressing

- Generate a global sequence of hash values and assign them in order
  - No link between the key and the hash
  
  \[
  \Sigma \{1, 3, 4, 5\}
  \]

  \[
  \begin{align*}
  \text{A} & \rightarrow \text{A} \\
  \text{B} & \rightarrow \text{B} \\
  \text{C} & \rightarrow \text{C}
  \end{align*}
  \]

- Now associate an individual sequence of hashes with each key (there are m! sequences to choose amongst)

  \[
  \text{key generates sequence}
  \]

  \[
  \begin{align*}
  \text{E.g.} & \quad A = 1, 3, 5, 2, 4 \\
  & \quad B = 3, 5, 2, 1, 4 \\
  & \quad C = 4, 2, 3, 1, 5 \\
  & \quad D = 1, 4, 3, 5, 2
  \end{align*}
  \]
Search

- Use key to generate **probe sequence**
- Follow sequence until:
  - Element found OR
  - Empty hash slot reached OR
  - Sequence finishes

<table>
<thead>
<tr>
<th>k</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
</tr>
<tr>
<td>4</td>
<td>D</td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

Search D
Probe: \{1, 4, 3, 5, 2\}

1 → A \times
4 → C \times
3 → B \times
5 → D \checkmark

Search E
Probe: \{1, 3, 2, 5, 4\}

1 → A \times
3 → B \times
2 → Empty \checkmark

E is not in list
Generating Sequences

Requirement: Every probe sequence should be a permutation of \( \{0, 1, \ldots, m-1\} \)

- Linear Probing

\[ S_i(k) = \left( h(k) + i \right) \mod m \]

- Basically 'randomises' the start of the sequence and then proceeds incrementally

  X Long runs of slots "primary clustering"

  X Two keys with same \( h(k) \) "secondary clustering"

  same sequence
Generating Sequences

Requirement: Every probe sequence should be a permutation of \{0, 1, ..., m-1\}

- Linear Probing

- Basically 'randomises' the start of the sequence and then proceeds incrementally
Generating Sequences

- Quadratic Probing
  \[ S_i(k) = (h(k) + c_1i + c_2i^2) \mod m \]
  \( \checkmark \) No primary clustering
  \( \times \) secondary clustering

- Double Hashing
  \[ S_i(k) = (h(k) + ih_2(k)) \mod m \]
  \( \checkmark \) No primary
  \( \checkmark \) No secondary
  \( \times \) Hard to find \( h \) and \( h_2 \)
Number of Probes Needed

- Consider a search that performs $i$ probes before it fails
Open Addressing Performance

- Ave. number of probes in a failed search
  \[ \alpha = \frac{n}{m} \]
  \[ \leq \frac{1}{1 - \alpha} \]

- Ave. Number of probes in a successful search
  \[ \frac{1}{\alpha} \ln \frac{1}{1 - \alpha} + \frac{1}{\alpha} \]

- If we can keep $n/m \sim$ constant, then the searches run in $O(1)$ still
Issues with Hash Tables

- Worst-case performance is dreadful \(- O(n)\)
- Deletion is slightly tricky if using open addressing

A: \{1, 2, 3, 4\}
B: \{1, 3, 4, 2\}
C: \{1, 4, 2, 3\}
D: \{1, 3, 2, 4\}

1) Forbid deletions
2) Mark with "DELETED"
   \(\Rightarrow\) Search ignores node
   \(\Rightarrow\) Insert replace node
Finishing with Tables

- You may remember we started all this trying to figure out how to implement the Table ADT
  - RB and AVL trees give us balanced trees that always give good performance and allow us to order the keys
  - B-trees are useful when the table becomes so big we needed to store it on the hard drive
  - Hash tables provide us with potentially really fast operations, but (like quicksort) terrible worst-case performance. They also don't sort the keys so you can't iterate over them in order
Java Tables

- **HashMap**
  - Implements a hash table
  - You can specify an initial size and a load factor \((n/m)\)

- **TreeMap**
  - Implements a red-black tree
  - Allows you to provide a Comparator for the keys so they are stored in sorted order
  - Therefore you can iterate over the values in a stable key order