Symbol Table and Hashing

- A (symbol) table is a set of table entries, \((K, V)\), each containing a unique key, \(K\), and a value (information), \(V\).

- Each key uniquely identifies its entry.

- Table searching: given a search key, \(K\), find the table entry, \((K, V)\).
  - Once such an entry is found, it may be retrieved, or its value, \(V\), may be updated, or the entire entry, \((K, V)\), may be removed from the table.
  - If no entry with key \(K\) exists in the table, a new table entry having \(K\) as its key may be inserted in the table.

- Hashing is a technique of storing values and searching for them in tables: linear, \(O(n)\), worst-case and extremely fast, \(O(1)\), average-case time.
Basic Features of Hashing

- Hashing computes an integer, called the hash code, for each object. The computation, called the hash function, \( h(K) \), maps objects (e.g., keys \( K \)) to array indices (e.g., \( 0, 1, \ldots, i_{\text{max}} \)). An object having a key value \( K \) should be stored at location \( h(K) \), and the hash function must always return a valid index for the array. A perfect hash function produces a different index value for every key. But such a function cannot be always found.

- **Collision**: if two distinct keys, \( K_1 \neq K_2 \), map to the same table address, \( h(K_1) = h(K_2) \).

- **Collision resolution policy**: how to find additional storage in which to store one of the collided table entries.
How common are collisions?

Von Mises Birthday Paradox:
if there are more than 23 people in a room, the chance is greater than 50% that two or more of them will have the same birthday.

Thus, in the table that is only 6.3% full (since $23/365 = 0.063$) there is better than 50–50 chance of a collision!

Probability $Q_N(n)$ that none of the $n$ items collides, being randomly tossed into a table with $N$ slots:

\[
Q_N(1) = 1; \quad Q_N(2) = Q_N(1) \cdot \frac{N-1}{N}; \\
Q_N(3) = Q_N(2) \cdot \frac{N-2}{N}; \ldots; \\
Q_N(n) = Q_N(n-1) \cdot \frac{N-n+1}{N}.
\]
Probability $P_N(n)$ of one or more collisions

\[
P_N(n) = 1 - Q_N(n)
= 1 - \frac{N(N-1)\cdots(N-n+1)}{N^n}
\]

<table>
<thead>
<tr>
<th>$n$</th>
<th>10</th>
<th>20</th>
<th>30</th>
<th>40</th>
<th>50</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_{365}(n)$</td>
<td>0.12</td>
<td>0.41</td>
<td>0.71</td>
<td>0.89</td>
<td>0.97</td>
</tr>
</tbody>
</table>
Open addressing with linear probing

The simplest collision resolution policy: to successively search for the first empty entry at a lower location (if no such entry, then “wrap around” the table).

<table>
<thead>
<tr>
<th>Pairs [key,value]</th>
<th>Hash code: key/10</th>
<th>Table address</th>
</tr>
</thead>
<tbody>
<tr>
<td>[20, A]</td>
<td>(h(20) = 2)</td>
<td>2</td>
</tr>
<tr>
<td>[15, B]</td>
<td>(h(15) = 1)</td>
<td>1</td>
</tr>
<tr>
<td>[45, C]</td>
<td>(h(45) = 4)</td>
<td>4</td>
</tr>
<tr>
<td>[87, D]</td>
<td>(h(87) = 8)</td>
<td>8</td>
</tr>
<tr>
<td>[39, E]</td>
<td>(h(39) = 3)</td>
<td>3</td>
</tr>
<tr>
<td>[31, F]</td>
<td>(h(31) = 3)</td>
<td>Collision!</td>
</tr>
</tbody>
</table>

Drawbacks: clustering of keys in the table.
Open addressing with double hashing

**Better collision resolution policy** reducing the likelihood of clustering: to hash the collided key again using a different hash function and use the result of the second hashing as an increment for probing table locations (including wraparound).

<table>
<thead>
<tr>
<th>Pairs [key,value]</th>
<th>Hash code: key/10</th>
<th>Table address</th>
<th>Hash address</th>
</tr>
</thead>
<tbody>
<tr>
<td>[20, A]</td>
<td>h(20) = 2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>[15, B]</td>
<td>h(15) = 1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>[45, C]</td>
<td>h(45) = 4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>[87, D]</td>
<td>h(87) = 8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>[39, E]</td>
<td>h(39) = 3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>[31, F]</td>
<td>h(31) = 3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

Collision!

Probe decrement Δ(31) = 4

[24, G]           | h(24) = 2        | Collision!    | Double hashing

Probe decrement Δ(24) = 6
Two more collision resolution techniques

Open addressing presents a problem if significant number of items need to be deleted, logically deleted items must remain in the table until the table can be reorganised.

Two techniques to attenuate this drawback:

• **Chaining**: all keys collided at a single hash address are placed on a linked list, or chain, started at that address.

• **Hash bucket**: a big hash table is divided into a number of small subtables, or buckets, and the hash function maps a key into one of the buckets; the keys are stored in each bucket sequentially in increasing order.
Universal classes of hash functions

Universal hashing – a random choice of the hash function from a large class of hash function to avoid bad performance on certain sets of input.

Let $K$ be the key set, let $N$ be the desired size of the range of the hash function, and let $H$ be a set of functions that map $K$ to $\{0, \ldots, N-1\}$. Then $H$ is said to be universal provided that for any distinct $k, \kappa \in K$,

$$\left| \left\{ h \in H : h(k) = h(\kappa) \right\} \right| \leq \frac{1}{N}.$$  

$H$ is a universal class if no pair of distinct keys collide under more than $\frac{1}{N}$ of the functions in the class.
Choosing a hash function

Four basic methods: *division*, *folding*, *middle-squaring*, and *truncation*.

**Division:**
- choose a prime number as the table size $N$,
- convert keys, $K$, into integers,
- use the remainder $h(K) = K \mod N$ as a hash value of the key $K$,
- find a double hashing decrement using the quotient, $\Delta(K) = \max\{1, \left(\frac{K}{N}\right) \mod N\}$

**Folding:**
- divide the integer key, $K$, into sections, – add, subtract, and/or multiply them together for combining into the final value, $h(K)$.

$K = 013402122$ — sections 013, 402, 122 — $h(K) = 013 + 402 + 122 = 537$
Choosing a hash function

**Middle-squaring:**
- choose a middle section of the integer key, $K$,
- square the chosed section,
- use a middle section of the result as $h(K)$.

$K = 013402122$ — middle: $402$ — $402^2 = 161404$ — middle: $h(K) = 6140$

**Truncation:**
- delete part of the key, $K$, — use the remaining digits (bits, characters) as $h(K)$.

$K = 013402122$ — last 3 digits: $h(K) = 122$

Notice that truncation does not spread keys uniformly into the table. Thus it is often used in conjunction with other methods.
Universal Class of Hash Functions by Division

Theorem:
Let the size of a key set, $K$, be a prime number: $|K| = M$. Let the members of $K$ be regarded as the integers $0, \ldots, M - 1$.

For any numbers $a \in \{1, \ldots, M - 1\}$ and $b \in \{0, \ldots, M - 1\}$ let

$$h_{a,b}(k) = ((a \cdot k + b) \mod M ) \mod M \ N.$$  

Then

$$H = \{h_{a,b} : 1 \leq a < M \text{ and } 0 \leq b < M\}$$

is a universal class.

In practice: let $M$ be the next prime number larger than the size of the key set. Then — choose randomly $a$ and $b$ such that $a > 0$ and use the hash function $h_{a,b}(k)$. 
Efficiency of search in hash tables

**Load factor** $\lambda$: if a table of size $N$ has exactly $M$ occupied entries, then $\lambda = \frac{M}{N}$.

Approximate average numbers of probe addresses examined for a successful ($S_\lambda$) and unsuccessful ($U_\lambda$) search:

<table>
<thead>
<tr>
<th></th>
<th>OALP</th>
<th>OADH</th>
<th>SC</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_\lambda$:</td>
<td>$0.5 \left(1 + \frac{1}{1-\lambda}\right)$</td>
<td>$\frac{1}{\lambda} \ln \left(\frac{1}{1-\lambda}\right)$</td>
<td>$1 + \frac{\lambda}{2}$</td>
</tr>
<tr>
<td>$U_\lambda$:</td>
<td>$0.5 \left(1 + \left(\frac{1}{1-\lambda}\right)^2\right)$</td>
<td>$\frac{1}{1-\lambda}$</td>
<td>$\lambda$</td>
</tr>
</tbody>
</table>

OALP – open addressing with linear probing
OADH – open addressing with double hashing
SC – separate chaining

OALP and OADH: $\lambda \leq 0.7$;
SC: $\lambda$ may be higher than 1
Theoretical vs. Experimental Efficiency of Search

\( N = 997 \), average of 50 trials

**Successful search** \( S_\lambda \)

<table>
<thead>
<tr>
<th>( \lambda )</th>
<th>SC</th>
<th>OALP</th>
<th>OADH</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.10</td>
<td>1.05/1.04</td>
<td>1.06/1.05</td>
<td>1.05/1.05</td>
</tr>
<tr>
<td>0.25</td>
<td>1.12/1.12</td>
<td>1.17/1.16</td>
<td>1.15/1.15</td>
</tr>
<tr>
<td>0.50</td>
<td>1.25/1.25</td>
<td>1.50/1.46</td>
<td>1.39/1.37</td>
</tr>
<tr>
<td>0.75</td>
<td>1.37/1.36</td>
<td>2.50/2.42</td>
<td>1.85/1.85</td>
</tr>
<tr>
<td>0.90</td>
<td>1.45/1.44</td>
<td>5.50/4.94</td>
<td>2.56/2.63</td>
</tr>
<tr>
<td>0.99</td>
<td>1.49/1.49</td>
<td>50.5/16.4</td>
<td>4.65/4.79</td>
</tr>
</tbody>
</table>

**Unsuccessful search** \( U_\lambda \)

<table>
<thead>
<tr>
<th>( \lambda )</th>
<th>SC</th>
<th>OALP</th>
<th>OADH</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.10</td>
<td>0.10/0.10</td>
<td>1.12/1.11</td>
<td>1.11/1.11</td>
</tr>
<tr>
<td>0.25</td>
<td>0.25/0.21</td>
<td>1.39/1.37</td>
<td>1.33/1.33</td>
</tr>
<tr>
<td>0.50</td>
<td>0.50/0.47</td>
<td>2.50/2.38</td>
<td>2.00/2.01</td>
</tr>
<tr>
<td>0.75</td>
<td>0.75/0.80</td>
<td>8.50/8.36</td>
<td>4.00/4.10</td>
</tr>
<tr>
<td>0.90</td>
<td>0.90/0.93</td>
<td>50.5/39.1</td>
<td>10.0/10.9</td>
</tr>
<tr>
<td>0.99</td>
<td>0.99/0.97</td>
<td>5000/360.9</td>
<td>100.0/98.5</td>
</tr>
</tbody>
</table>
Java: hash tables

java.util package:
• Java 1.0 and 1.1: class Hashtable
• Java 1.2: classes HashMap and HashSet

Hashtable implementation:

• Any object can be stored in a hash table

• Objects used as keys must implement equals() and hashCode() methods:
  o if a.equals(b)
    then a.hashCode() must be equal to b.hashCode()

• Object get(Object key) —
  returns the object keyed by key

• Object put(Object key, Object value) —
  puts value to the hash table and returns old object
  keyed by key

Table has size and load factor: when number of entries
is greater than $\lambda \cdot$ size, the table is resized.

Collisions are resolved by chaining.
Class Hashtable

In Java 1.2 this class has been retrofitted to implement Map interface, so that it becomes a part of Java Collections Framework.

An instance of Hashtable has two parameters: initial capacity and load factor (default: 0.75).

Example: a hash table of numbers with their names as keys:

Hashtable numbers = new Hashtable();

numbers.put(‘one’, new Integer(1));
numbers.put(‘two’, new Integer(2));
numbers.put(‘ten’, new Integer(10));

To retrieve a number:

Integer n = (Integer)numbers.get(‘two’);
if ( n != null )
    System.out.println( ‘two = ’ + n );
Efficiency of Search for Table ADT Implementations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Sorted array</th>
<th>AVL tree</th>
<th>Hash table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize</td>
<td>$O(N)$</td>
<td>$O(1)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>Is full?</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Search*)</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Insert</td>
<td>$O(N)$</td>
<td>$O(\log N)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Delete</td>
<td>$O(N)$</td>
<td>$O(\log N)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Enumerate</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
<td>$O(N \log N)$</td>
</tr>
</tbody>
</table>

*) also: Retrive, Update

To enumerate a hash table, entries must first be sorted in ascending order of their keys that takes $O(N \log N)$ time