Hash Tables
A hash function $h$ maps keys of a given type to integers in a fixed interval $[0, N - 1]$

Example:
$$h(x) = x \mod N$$
is a hash function for integer keys

The integer $h(x)$ is called the hash value of key $x$

A hash table for a given key type consists of
- Hash function $h$
- Array (called table) of size $N$

When implementing a map with a hash table, the goal is to store item $(k, o)$ at index $i = h(k)$
Example

We design a hash table for a dictionary storing entries as (SSN, Name), where SSN (social security number) is a nine-digit positive integer.

Our hash table uses an array of size $N = 10,000$ and the hash function $h(x) = \text{last four digits of } x$. 
Hash Functions

A hash function is usually specified as the composition of two functions:

**Hash code:**
- \( h_1: \text{keys} \rightarrow \text{integers} \)

**Compression function:**
- \( h_2: \text{integers} \rightarrow [0, N - 1] \)

The hash code is applied first, and the compression function is applied next on the result, i.e.,

\[
h(x) = h_2(h_1(x))
\]

The goal of the hash function is to “disperse” the keys in an apparently random way.
Hash Codes

- **Memory address:**
  - We reinterpret the memory address of the key object as an integer (default hash code of all Java objects)
  - Good in general, except for numeric and string keys

- **Integer cast:**
  - We reinterpret the bits of the key as an integer
  - Suitable for keys of length less than or equal to the number of bits of the integer type (e.g., byte, short, int and float in Java)

- **Component sum:**
  - We partition the bits of the key into components of fixed length (e.g., 16 or 32 bits) and we sum the components (ignoring overflows)
  - Suitable for numeric keys of fixed length greater than or equal to the number of bits of the integer type (e.g., long and double in Java)
Hash Codes (cont.)

Polynomial accumulation:
- We partition the bits of the key into a sequence of components of fixed length (e.g., 8, 16 or 32 bits)
  \( a_0 a_1 \ldots a_{n-1} \)
- We evaluate the polynomial
  \[ p(z) = a_0 + a_1 z + a_2 z^2 + \ldots + a_{n-1} z^{n-1} \]
  at a fixed value \( z \), ignoring overflows
- Especially suitable for strings (e.g., the choice \( z = 33 \) gives few collisions on an application using English words)

Polynomial \( p(z) \) can be evaluated in \( O(n) \) time using Horner’s rule:
- The following polynomials are successively computed, each from the previous one in \( O(1) \) time
  \[
  \begin{align*}
  p_0(z) &= a_{n-1} \\
  p_i(z) &= a_{n-i-1} + z p_{i-1}(z) \\
  (i &= 1, 2, \ldots, n-1)
  \end{align*}
  \]
- We have \( p(z) = p_{n-1}(z) \)
Compression Functions

Division:
- $h_2(y) = y \mod N$
- The size $N$ of the hash table is usually chosen to be a prime
- The reason has to do with number theory and is beyond the scope of this course

Multiply, Add and Divide (MAD):
- $h_2(y) = (ay + b) \mod N$
- $a$ and $b$ are nonnegative integers such that $a \mod N \neq 0$
- Otherwise, every integer would map to the same value $b$
Collision Handling

Collisions occur when different elements are mapped to the same cell

Separate Chaining:
let each cell in the table point to a linked list of entries that map there

Separate chaining is simple, but requires additional memory outside the table
Dictionary Methods with Separate Chaining used for Collisions

Delegate operations to a list-based dictionary at each cell:

**Algorithm** `find(k)`:

*Output:* The value associated with the key `k` in the dictionary, or `null` if there is no entry with key equal to `k` in the dictionary

```latex
return A[h(k)].get(k) \{delegate the get to the list-based map at A[h(k)]\}
```

**Algorithm** `insert(k, v)`:

*Output:* If there is an existing entry in our dictionary with key equal to `k`, then we return its value (replacing it with `v`); otherwise, we return `null`

```latex
t = A[h(k)].put(k, v) \{delegate the put to the list-based map at A[h(k)]\}
if t = null then
    \( n = n + 1 \)
return t
```

**Algorithm** `remove(k)`:

*Output:* The (removed) value associated with key `k` in the dictionary, or `null` if there is no entry with key equal to `k` in the dictionary

```latex
t = A[h(k)].remove(k) \{delegate the remove to the list-based map at A[h(k)]\}
if \( t \neq \text{null} \) then
    \( n = n - 1 \)
return t
```
Linear Probing

- **Open addressing**: the colliding item is placed in a different cell of the table.
- **Linear probing** handles collisions by placing the colliding item in the next (circularly) available table cell.
- Each table cell inspected is referred to as a “probe”.
- Colliding items lump together, causing future collisions to cause a longer sequence of probes.

**Example:**
- \[ h(x) = x \mod 13 \]
- Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order.

![Diagram of hash table with keys inserted using linear probing]

© 2004 Goodrich, Tamassia
Search with Linear Probing

Consider a hash table $A$ that uses linear probing

**find($k$)**
- We start at cell $h(k)$
- We probe consecutive locations until one of the following occurs
  - An item with key $k$ is found, or
  - An empty cell is found, or
  - $N$ cells have been unsuccessfully probed

**Algorithm find($k$)**

```plaintext
i ← h(k)  
p ← 0  
repeat  
c ← A[i]  
if c = ∅  
   return null  
else if c.key () = k  
   return c.element()  
else  
   i ← (i + 1) mod N  
p ← p + 1  
until p = N  
return null
```
Updates with Linear Probing

To handle insertions and deletions, we introduce a special object, called `AVAILABLE`, which replaces deleted elements.

**remove**(*k*)
- We search for an entry with key *k*.
- If such an entry (*k*, *o*) is found, we replace it with the special item `AVAILABLE` and we return element *o*.
- Else, we return `null`.

**insert**(*k*, *o*)
- We throw an exception if the table is full.
- We start at cell *h(k)*.
- We probe consecutive cells until one of the following occurs:
  - A cell *i* is found that is either empty or stores `AVAILABLE`, or
  - *N* cells have been unsuccessfully probed.
- We store entry (*k*, *o*) in cell *i*. 
Double Hashing

- Double hashing uses a secondary hash function $d(k)$ and handles collisions by placing an item in the first available cell of the series $(i + jd(k)) \mod N$
- The secondary hash function $d(k)$ cannot have zero values
- The table size $N$ must be a prime to allow probing of all the cells

Common choice of compression function for the secondary hash function:
\[ d_2(k) = q - k \mod q \]
where
- $q < N$
- $q$ is a prime

The possible values for $d_2(k)$ are $1, 2, \ldots, q$
Example of Double Hashing

Consider a hash table storing integer keys that handles collision with double hashing

- $N = 13$
- $h(k) = k \mod 13$
- $d(k) = 7 - k \mod 7$

Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

<table>
<thead>
<tr>
<th>$k$</th>
<th>$h(k)$</th>
<th>$d(k)$</th>
<th>Probes</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>5</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>41</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>22</td>
<td>9</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>44</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>59</td>
<td>7</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>32</td>
<td>6</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>31</td>
<td>5</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>73</td>
<td>8</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>
Performance of Hashing

- In the worst case, searches, insertions and removals on a hash table take $O(n)$ time.
- The worst case occurs when all the keys inserted into the map collide.
- The load factor $\alpha = n/N$ affects the performance of a hash table.
- Assuming that the hash values are like random numbers, it can be shown that the expected number of probes for an insertion with open addressing is $1 / (1 - \alpha)$.

- The expected running time of all the dictionary ADT operations in a hash table is $O(1)$.
- In practice, hashing is very fast provided the load factor is not close to 100%.

Applications of hash tables:
- small databases
- compilers
- browser caches