Hash Tables
Recall the Map ADT (§ 8.1)

Map ADT methods:

- **get(k):** if the map M has an entry with key k, return its associated value; else, return null
- **put(k, v):** insert entry \((k, v)\) into the map M; if key k is not already in M, then return null; else, return old value associated with k
- **remove(k):** if the map M has an entry with key k, remove it from M and return its associated value; else, return null
- **size(), isEmpty()**
- **keys():** return an iterator of the keys in M
- **values():** return an iterator of the values in M
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 map 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 $h(x) = \text{last four digits of } x$. 

\begin{align*}
0 & \quad \emptyset \\
1 & \quad 025-612-0001 \\
2 & \quad 981-101-0002 \\
3 & \quad 451-229-0004 \\
4 & \quad \ldots \\
9997 & \quad \emptyset \\
9998 & \quad 200-751-9998 \\
9999 & \quad \emptyset
\end{align*}
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 (§ 8.2.3)

- **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)
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 at most 6 collisions on a set of 50,000 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
  \[ p_0(z) = a_{n-1} \]
  \[ p_i(z) = a_{n-i-1} + z p_{i-1}(z) \]
  \((i = 1, 2, \ldots, n-1)\)
- We have \( p(z) = p_{n-1}(z) \)
Compression Functions
(§ 8.2.4)

- **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
(§ 8.2.5)

- 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

<table>
<thead>
<tr>
<th>Cell</th>
<th>Entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>∅</td>
</tr>
<tr>
<td>1</td>
<td>025-612-0001</td>
</tr>
<tr>
<td>2</td>
<td>∅</td>
</tr>
<tr>
<td>3</td>
<td>∅</td>
</tr>
<tr>
<td>4</td>
<td>451-229-0004 981-101-0004</td>
</tr>
</tbody>
</table>

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

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

**Algorithm** get\(k\):

*Output:* The value associated with the key \(k\) in the map, or \textbf{null} if there is no entry with key equal to \(k\) in the map

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

**Algorithm** put\((k,v)\):

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

\[
t = A[h(k)].\text{put}(k,v) \quad \{\text{delegate the put to the list-based map at } A[h(k)]\}
\]

\[
\text{if } t = \textbf{null} \text{ then } n = n + 1
\]

\[
\text{return } t
\]

**Algorithm** remove\((k)\):

*Output:* The (removed) value associated with key \(k\) in the map, or \textbf{null} if there is no entry with key equal to \(k\) in the map

\[
t = A[h(k)].\text{remove}(k) \quad \{\text{delegate the remove to the list-based map at } A[h(k)]\}
\]

\[
\text{if } t \neq \textbf{null} \text{ then } n = n - 1
\]

\[
\text{return } t
\]
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:**

- $h(x) = x \text{ mod } 13$
- Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order.

```
0 1 2 3 4 5 6 7 8 9 10 11 12
|   |   |   |   |   |   |   | 18 |44 |59 |32 |22 |31 |73 |
0 1 2 3 4 5 6 7 8 9 10 11 12
```

© 2004 Goodrich, Tamassia
Consider a hash table $A$ that uses linear probing.

**get($k$)**
- 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 get($k$)**

```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.

**put**\((k, o)\)
- 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 \(i\).

**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`.
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 $(i + jd(k)) \mod N$
  for $j = 0, 1, \ldots, N - 1$

- 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></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 $\alpha = \frac{n}{N}$ 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 $\frac{1}{1 - \alpha}$.
- 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
/** A hash table with linear probing and the MAD hash function */
public class HashTable implements Map {
    protected static class HashEntry implements Entry {
        Object key, value;
        HashEntry () {/* default constructor */}
        HashEntry(Object k, Object v) { key = k; value = v; }
        public Object key() { return key; }
        public Object value() { return value; }
        protected Object setValue(Object v) { // set a new value, returning old
            Object temp = value;
            value = v;
            return temp; // return old value
        }
    }
    /** Nested class for a default equality tester */
    protected static class DefaultEqualityTester implements EqualityTester {
        DefaultEqualityTester() {/* default constructor */}
        /** Returns whether the two objects are equal. */
        public boolean isEqualTo(Object a, Object b) { return a.equals(b); }
    }
    protected static Entry AVAILABLE = new HashEntry(null, null); // empty
    marker
    protected int n = 0; // number of entries in the dictionary
    protected int N; // capacity of the bucket array
    protected Entry[] A; // bucket array
    protected EqualityTester T; // the equality tester
    protected int scale, shift; // the shift and scaling factors
    /** Creates a hash table with initial capacity 1023. */
    public HashTable() {
        N = 1023; // default capacity
        A = new Entry[N];
        T = new DefaultEqualityTester(); // use the default equality tester
        java.util.Random rand = new java.util.Random();
        scale = rand.nextInt(N-1) + 1;
        shift = rand.nextInt(N);
    }
    /** Creates a hash table with the given capacity and equality tester. */
    public HashTable(int bN, EqualityTester tester) {
        N = bN;
        A = new Entry[N];
        T = tester;
        java.util.Random rand = new java.util.Random();
        scale = rand.nextInt(N-1) + 1;
        shift = rand.nextInt(N);
    }
/** Determines whether a key is valid. */
protected void checkKey(Object k) {
    if (k == null) throw new InvalidKeyException("Invalid key: null.");
}
/** Hash function applying MAD method to default hash code. */
public int hashCode(Object key) {
    return Math.abs(key.hashCode() * scale + shift) % N;
}
/** Returns the number of entries in the hash table. */
public int size() { return n; }
/** Returns whether or not the table is empty. */
public boolean isEmpty() { return (n == 0); }
/** Helper search method - returns index of found key or -index-1,
* where index is the index of an empty or available slot. */
protected int findEntry(Object key) throws InvalidKeyException {
    int avail = 0;
    checkKey(key);
    int i = hashValue(key);
    int j = i;
    do {
        if (A[i] == null) return -i - 1; // entry is not found
        if (A[i] == AVAILABLE) { // bucket is deactivated
            avail = i; // remember that this slot is available
            i = (i + 1) % N; // keep looking
        } else if (T.isEqualTo(key, A[i].key())) // we have found our entry
            return i;
        else // this slot is occupied--we must keep looking
            i = (i + 1) % N;
    } while (i != j);
    return -avail - 1; // entry is not found
}
/** Returns the value associated with a key. */
public Object get(Object key) throws InvalidKeyException {
    int i = findEntry(key); // helper method for finding a key
    if (i < 0) return null; // there is no value for this key
    return A[i].value(); // return the found value in this case
}
/** Put a key-value pair in the map, replacing previous one if it exists. */
public Object put(Object key, Object value) throws InvalidKeyException {
    if (n >= N/2) rehash(); // rehash to keep the load factor <= 0.5
    int i = findEntry(key); // find the appropriate spot
    if (i < 0) {
        A[++i] = new HashEntry(key, value); // convert to the proper index
    } else // this key has a previous value
        A[i] = setValue(value); // set new value & return old
}
/** Doubles the size of the hash table and rehashes all the entries. */
protected void rehash() {
    N = 2*N;
    Entry[] B = A;
    A = new Entry[N]; // allocate a new version of A twice as big as before
    java.util.Random rand = new java.util.Random();
    scale = rand.nextInt(N - 1) + 1; // new hash scaling factor
    shift = rand.nextInt(N); // new hash shifting factor
    for (int i = 0; i != B.length; i++)
        if ((B[i] != null) && (B[i] != AVAILABLE)) // if we have a valid entry
            A[i] = B[i]; // copy into the new array
    // ... values() is similar to keys() and is omitted here ...