Sets, Maps, Hashing

Overview
- Sets
- Maps
- Hashing
  - Scattering Hash Values
  - Hash Function
- Hash Tables
  - Open Addressing
  - Chaining
- Java equals and hashCode()

Set Data Structures
- No relationship between elements
- Types of sets
  - Set
  - Map
  - Hash Table

How Do Sets Work in Java?
- Finding matching element is based on equals()
- To build a collection for a class
  - Need to define your own equals(Object) method
  - Default equals() uses reference comparison
    - I.e., a.equals(b) → a == b
    - a, b equal only if reference to same object
  - Many classes have predefined equals() methods
    - Integer.equals() → compares value of integer
    - String.equals() → compares text of string

Sets
- Properties
  - Collection of elements without duplicates
  - No ordering (i.e., no front or back)
  - Order in which elements added doesn't matter
- Implementation goal
  - Offer the ability to find/remove element quickly
  - Without searching through all elements

Set Concrete Classes
- HashSet
  - Elements must implement hashCode() method
- LinkedHashSet
  - HashSet supporting ordering of elements
  - Elements can be retrieved in order of insertion
- TreeSet
  - Elements must be comparable
  - Implement Comparable or provide Comparator
  - Guarantees elements in set are sorted
Map Definition

- Map (associative array)
  - Unordered collection of keys
  - For each key, an associated object
  - Can use key to retrieve object
- Can view as array indexed by any (key) value
  - Example
    A["key1"] = ...

Map Interface Methods

- Methods
  - `void put( Key, Object )` // inserts element
  - `Object get( Key )` // returns element
  - `void remove( Key )` // removes element
  - `Boolean containsKey( Key )` // looks for key
  - `Set keySet( )` // entire set of keys

Map Properties

- Map keys & map objects
  - Can also treat keys & values as collections
    - Access using keySet( ), values( )
  - Aliasing
    - Each key refers only a single object
    - But object may be referred to by multiple keys
  - Keys & values may be of complex type
    - `Map<Object Type1, Any Object Type2>
    - Including other collections, maps, etc...

Map Implementation

- Implementation approaches
  - Two parallel arrays
    - Unsorted
    - Sorted
  - Linked list
  - Binary search tree
  - Hash table
- Java Collections Framework
  - `TreeMap` → uses red-black (balanced) tree
  - `HashMap` → uses hash table

Java Collections Map Hierarchy

- `Map`
  - `SortedMap`
  - `AbstractMap`
    - `TreeMap`
    - `HashMap`
      - `LinkedHashMap`

Hashing

- Approach
  - Use hash function to convert key into number (hash value) used as index in hash table

Hash table h

\[
\begin{array}{cccccc}
V_1 & V_2 & V_3 & V_4 & \ldots & V_n \\
\hline
H(k_1) & H(k_2) & H(k_3) & H(k_4) & \ldots & H(k_n)
\end{array}
\]

Hash function f
# Hashing

- **Hash Table**
  - Array indexed using hash values
  - Hash table A with size N
  - Indices of A range from 0 to N-1
  - Store in A[hashValue % N]

<table>
<thead>
<tr>
<th>Hash table h</th>
<th>Location</th>
<th>Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>h[0]</td>
<td></td>
<td>Δ</td>
</tr>
<tr>
<td>h[1]</td>
<td></td>
<td>Δ</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>Δ</td>
</tr>
<tr>
<td>h[N − 1]</td>
<td></td>
<td>Δ</td>
</tr>
</tbody>
</table>

- **Hash Function**
  - Function for converting key into hash value
  - For Java
    - Hash value ⇒ 32-bit signed int
    - Default hash function ⇒ int hashCode()
  - For hash table of size N
    - Must reduce hash value to 0..N – 1
    - Can use modulo operator
      - Math.abs(hash value % N)

# Scattering Hash Values

- Hash function should scatter hash values uniformly across range of possible values
- Reduces likelihood of conflicts between keys
- Hash(<everything>) = 0
  - Satisfies definition of hash function
  - But not very useful (all keys at same location)
- Could use Math.abs(key.hashCode()) % N
  - Might not distribute values well
  - Particularly if N is a power of 2

# Beware of % (Modulo Operator)

- The % operator is integer remainder
  - x % y = x − y * (x / y)
- Result may be negative
  - −|y| < x % y < +|y|
- x % y has same sign as x
  - -3 % 2 = -1
  - -3 % -2 = -1
- Use Math.abs(x % N), not Math.abs(x) % N
  - Since Math.abs(Integer.MIN_VALUE) == Integer.MIN_VALUE
  - Will happen 1 in 2^{32} times (on average) for random int values

# Art and Magic of hashCode()

- There is no “right” hashCode function
  - Art involved in finding good hashCode function
  - Also for finding hashCode to hashBucket function

```java
From java.util.HashMap
static int hashBucket(Object x, int N) {
    int h = x.hashCode();
    h += ~(h << 9);
    h ^=  (h >>> 14);
    h +=  (h << 4);
    h ^=  (h >>> 10);
    return Math.abs(h % N);
}
```
Hash Function

**Example**
- \( \text{hashCode("apple")} = 5 \)
- \( \text{hashCode("watermelon")} = 3 \)
- \( \text{hashCode("grapes")} = 8 \)
- \( \text{hashCode("kiwi")} = 0 \)
- \( \text{hashCode("strawberry")} = 9 \)
- \( \text{hashCode("mango")} = 6 \)
- \( \text{hashCode("banana")} = 2 \)

**Perfect hash function**
- Unique values for each key

<table>
<thead>
<tr>
<th>Hash Code</th>
<th>Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>kiwi</td>
</tr>
<tr>
<td>1</td>
<td>banana</td>
</tr>
<tr>
<td>2</td>
<td>watermelon</td>
</tr>
<tr>
<td>3</td>
<td>grapes</td>
</tr>
<tr>
<td>4</td>
<td>mango</td>
</tr>
<tr>
<td>5</td>
<td>strawberry</td>
</tr>
</tbody>
</table>

Hash Function

**Suppose now**
- \( \text{hashCode("apple")} = 5 \)
- \( \text{hashCode("watermelon")} = 3 \)
- \( \text{hashCode("grapes")} = 8 \)
- \( \text{hashCode("kiwi")} = 0 \)
- \( \text{hashCode("strawberry")} = 9 \)
- \( \text{hashCode("mango")} = 6 \)
- \( \text{hashCode("banana")} = 2 \)
- \( \text{hashCode("orange")} = 3 \)

**Collision**
- Same hash value for multiple keys

<table>
<thead>
<tr>
<th>Hash Code</th>
<th>Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>kiwi</td>
</tr>
<tr>
<td>1</td>
<td>banana</td>
</tr>
<tr>
<td>2</td>
<td>watermelon</td>
</tr>
<tr>
<td>3</td>
<td>grapes</td>
</tr>
<tr>
<td>4</td>
<td>mango</td>
</tr>
<tr>
<td>5</td>
<td>strawberry</td>
</tr>
</tbody>
</table>

Types of Hash Tables

- **Open addressing**
  - Store objects in each table entry

- **Chaining (bucket hashing)**
  - Store lists of objects in each table entry

Open Addressing Hashing

**Approach**
- Hash table contains objects
- Probe ⇒ examine table entry
- Collision
  - Move \( K \) entries past current location
  - Wrap around table if necessary
- Find location for \( X \)
  1. Examine entry at \( A[\text{key}(X)] \)
  2. If entry = \( X \), found
  3. If entry = empty, \( X \) not in hash table
  4. Else increment location by \( K \), repeat

Open Addressing Hashing

**Approach**
- Linear probing
  - \( K = 1 \)
  - May form clusters of contiguous entries
- Deletions
  - Find location for \( X \)
  - If \( X \) inside cluster, leave non-empty marker
- Insertion
  - Find location for \( X \)
  - Insert if \( X \) not in hash table
  - Can insert \( X \) at first non-empty marker

Open Addressing Example

**Hash codes**
- \( H(A) = 6 \)
- \( H(C) = 6 \)
- \( H(B) = 7 \)
- \( H(D) = 7 \)

**Hash table**
- Size = 8 elements
- \( \Lambda = \) empty entry
- \( * = \) non-empty marker

**Linear probing**
- Collision ⇒ move 1 entry past current location
Open Addressing Example

Operations
- Insert A, Insert B, Insert C, Insert D

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

Find A, Find B, Find C, Find D

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

Delete A, Delete C, Find D, Insert C

<table>
<thead>
<tr>
<th>D</th>
<th>D</th>
<th>D</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
</tbody>
</table>

Efficiency of Open Hashing

Load factor = entries / table size
Hashing is efficient for load factor < 90%

<table>
<thead>
<tr>
<th>Number of Comparisons</th>
<th>Approximate Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>O(1)</td>
</tr>
<tr>
<td>0.2</td>
<td>1.13</td>
</tr>
<tr>
<td>0.3</td>
<td>1.21</td>
</tr>
<tr>
<td>0.4</td>
<td>1.33</td>
</tr>
<tr>
<td>0.5</td>
<td>1.50</td>
</tr>
<tr>
<td>0.6</td>
<td>1.75</td>
</tr>
<tr>
<td>0.7</td>
<td>2.17</td>
</tr>
<tr>
<td>0.8</td>
<td>3.00</td>
</tr>
<tr>
<td>0.9</td>
<td>5.50</td>
</tr>
<tr>
<td>0.95</td>
<td>10.5</td>
</tr>
<tr>
<td>0.99</td>
<td>26.5</td>
</tr>
<tr>
<td>0.99</td>
<td>O(N)</td>
</tr>
</tbody>
</table>

Chaining (Bucket Hashing)

Approach
- Hash table contains lists of objects
- Find location for X
  - Find hash code key for X
  - Examine list at table entry A[ key ]
- Collision
  - Multiple entries in list for entry

Chaining Example

Hash codes
- H(A) = 6
- H(B) = 7
- H(C) = 6
- H(D) = 7

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>
Chaining Example

- **Operations**
  - Insert A, Insert B, Insert C
  
  - Insertion order:
    - 1 A
    - 2 A
    - 3 A
    - 4 A
    - 5 A
    - 6 A
    - 7 A
    - 8 A

- **Find B, Find A**
  
  - Retrieval order:
    - 1 A
    - 2 A
    - 3 A
    - 4 A
    - 5 A
    - 6 A
    - 7 A
    - 8 A

Efficiency of Chaining

- **Load factor** = entries / table size
- **Average case**
  - Evenly scattered entries
  - Operations = O( load factor )
- **Worse case**
  - Entries mostly have same hash value
  - Operations = O( entries )

Hashing in Java

- **Default hashCode( ) implementation**
  - Usually just address of object in memory
- **Can override with new user definition**
  - Must work with equals( )
  - Following Java “hashcode contract”

Java Hash Code Contract

- **hashCode( )**
  - Must return same value for object in each execution, provided information used in equals( ) comparisons on the object is not modified
- **equals( )**
  - if a.equals(b) == true, then must guarantee a.hashCode( ) == b.hashCode( )
  - Inverse is not true → !a.equals(b) does not imply a.hashCode( ) != b.hashCode( )
  - Though Java libraries may be more efficient
  - Converse is also not true → a.hashCode( ) == b.hashCode( ) does not imply a.equals(b) == true
Java `hashCode()`

- **Implementing `hashCode()`**
  - Include only information used by `equals()`
  - Else 2 “equal” objects → different hash values
  - Using all / more of information used by `equals()`
    - Help avoid same hash value for unequal objects

- **Example `hashCode()` functions**
  - For pair of Strings
    - 1st letter of 1st str
    - 1st letter of 1st str + 1st letter of 2nd str
    - Length of 1st str + length of 2nd str
    - \( \sum \) letter(s) of 1st str + \( \sum \) letter(s) of 2nd str