CMSC 132:
Object-Oriented Programming II

Sets, Maps, Hashing

Department of Computer Science
University of Maryland, College Park
Overview

- Sets
- Maps
- Hashing
- Java equals and hashCode()
Set Data Structures

- No relationship between elements
- Types of sets
  - Set
  - Map
  - Hash Table
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
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) \rightarrow a == b`
  - `a`, `b` equal only if reference to same object
- Many classes have predefined `equals()` methods
  - `Integer.equals()` \rightarrow compares value of integer
  - `String.equals()` \rightarrow compares text of string
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(K key, V value)` // inserts element
- `V get(Object key)` // returns element
- `V remove(Object key)` // removes element
- `int size()` // key-value mappings
- `void clear()` // clears the map
- `boolean containsKey(Object key)` // looks for key
- `boolean containsValue(Object value)` // looks for value
- `boolean isEmpty()` // empty map?
- `Set<K> keySet()` // entire set of keys
- `Collection<V> values()` // values in the map
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
    - TreeMap
  - AbstractMap
    - HashMap
    - LinkedHashMap
Hashing

Approach

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

**Hash table h**

<table>
<thead>
<tr>
<th>Location</th>
<th>Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Λ</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>15</td>
</tr>
<tr>
<td>3</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>Λ</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
Hash Function

- Function for converting key into hash value

- For Java
  - Hash value \( \Rightarrow 32\text{-bit signed int} \)
  - Default hash function \( \Rightarrow \text{int hashCode( )} \)

- 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
Scattering Hash Values

- Multiplicative congruency method
  - Produces good hash values
  - Hash value = Math.abs((a * key.hashCode()) % N)
  - Where
    - N is table size
    - a is large prime number
Beware of % (Modulo Operator)

- The % operator is integer remainder
  \[ x \% y = x - y \times (x / y) \]

- 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
There is no “right” hashCode function

- Art involved in finding good hashCode function
- Also for finding hashCode to hashBucket function

From java.util.HashMap

```java
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

hashCode("apple") = 5
hashCode("watermelon") = 3
hashCode("grapes") = 8
hashCode("kiwi") = 0
hashCode("strawberry") = 9
hashCode("mango") = 6
hashCode("banana") = 2

Perfect hash function

Unique values for each key
Hash Function

Suppose now

\[
\begin{align*}
&\text{hashCode("apple") = 5} \\
&\text{hashCode("watermelon") = 3} \\
&\text{hashCode("grapes") = 8} \\
&\text{hashCode("kiwi") = 0} \\
&\text{hashCode("strawberry") = 9} \\
&\text{hashCode("mango") = 6} \\
&\text{hashCode("banana") = 2} \\
&\text{hashCode("orange") = 3}
\end{align*}
\]

Collision

- Same hash value for multiple keys
Hashing in Java

- Object class has built-in support for hashing
  - Method `int hashCode()` provides
  - Numerical hash value for any object
- `hashCode()` provides **pre-filter** for `equals()`
  - Check `equals()` only if `hashCode()` is identical
  - **Example**
    ```java
    if ( a.hashCode() == b.hashCode() )
    result = a.equals( b );
    else result = false;
    ```
  - Efficient if `hashCode()` is faster than `equals()`
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 $\rightarrow$ !a.equals(b) does not imply
    - a.hashCode( ) != b.hashCode( )
  - Though Java libraries may be more efficient
  - Converse is also not true $\rightarrow$ 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 $\rightarrow$ 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**
  - 1$^{\text{st}}$ letter of 1$^{\text{st}}$ str
  - 1$^{\text{st}}$ letter of 1$^{\text{st}}$ str + 1$^{\text{st}}$ letter of 2$^{\text{nd}}$ str
  - Length of 1$^{\text{st}}$ str + length of 2$^{\text{nd}}$ str
  - $\sum$ letter(s) of 1$^{\text{st}}$ str + $\sum$ letter(s) of 2$^{\text{nd}}$ str