Hashing

Department of Computer Science
University of Maryland, College Park
Hashing

Hashing function → function that maps data to a value (e.g., integer)

Hash Code/Hash Value → value returned by a hash function

Hash functions can be used to speed up data access

We can achieve O(1) data access using 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]

\[
\begin{array}{|c|c|}
\hline
\text{Location} & \text{Key} \\
\hline
0 & \Lambda \\
1 & 10 \\
2 & 15 \\
3 & 20 \\
4 & \Lambda \\
\ldots & \ldots \\
\hline
\end{array}
\]

\[\text{Hash table } h\]

- h[0] = \Lambda
- h[1] = \Lambda
- \ldots
- h[N - 1] = \Lambda
Hash Function

- Hash Function → Function for converting key into hash value
- For hash table of size N
  - Must reduce hash value to 0..N – 1
  - Can use modulo operator → hash value = Math.abs(keyValue % N)
- Example Problem
  - Assign 4 parking spaces to 4 people using
    - h(key) = keyValue % 4
  - What happens if we have 4 spaces and 8 people?
    - Collision → Same hash value for multiple keys
- Bucket
  - Each table entry can be referred to as a bucket
  - In some implementations the bucket is represented by a list (those elements hashing to the same bucket are placed in the same list)
- Properties of a Good Hash Function
  - Distributes (scatters) values uniformly across range of possible values
  - It is not expensive to compute
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(keyValue % 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 * keyValue) % N)
- Where
  - N is table size
  - a is large prime number
Hash Function

Example

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

Perfect hash function

- Unique values for each key
Hash Function

Suppose now

<table>
<thead>
<tr>
<th>Hash Value</th>
<th>Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>apple</td>
</tr>
<tr>
<td>3</td>
<td>watermelon</td>
</tr>
<tr>
<td>8</td>
<td>grapes</td>
</tr>
<tr>
<td>0</td>
<td>kiwi</td>
</tr>
<tr>
<td>9</td>
<td>strawberry</td>
</tr>
<tr>
<td>6</td>
<td>mango</td>
</tr>
<tr>
<td>2</td>
<td>banana</td>
</tr>
<tr>
<td>3</td>
<td>orange</td>
</tr>
</tbody>
</table>

Collision

- Same hash value for multiple keys
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)` and not `Math.abs(x) % N`

About absolute value in Java
- `Math.abs(Integer.MIN_VALUE) == Integer.MIN_VALUE`!
- Will happen 1 in \(2^{32}\) times (on average) for random int values
Hashing in Java

- Object class has built-in support for hashing
  - Method `int hashCode()` provides
    - Numerical hash value for any object
    - 32-bit signed int

- Default `hashCode()` implementation
  - Usually just address of object in memory

- Can override with new user definition
  - Must work with `equals()`
  - Must satisfy the “hash code contract”
Java Hash Code Contract

if \( a.equals(b) == \text{true}, \) then we must guarantee
\[ a.hashCode() == b.hashCode() \]

Inverse is not true

\( \neg a.equals(b) \) does not imply
\[ a.hashCode() \neq b.hashCode() \]
(Though Java libraries may be more efficient)

Converse is also not true

\[ a.hashCode() == b.hashCode() \]
does not imply \( a.equals(b) == \text{true} \)

hashCode()

Must return same value for object in each execution, provided information used in equals( ) comparisons on the object is not modified
When to Override hashCode

- You must write classes that satisfy the Java Hash Code Contract
- You will run into problems if you don’t satisfy the Java Hash Code Contract and use classes that rely on hashing (e.g., HashMap, HashSet)
  - Possible problem – You add an element to a set but cannot find it during a lookup operation
  - See code distribution example
- Does the default equals and hashCode satisfy the contract? Yes!
- If you implement the Comparable interface you should provide the appropriate equals method which leads to the appropriate hashCode method
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
  - ∑ letter(s) of 1st str + ∑ letter(s) of 2nd str
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

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);
}
```
Pre-filtering with hashCode

- hashCode( ) provides **pre-filter** for equals( )
  - Check equals( ) only if hashCode( ) is identical
  - Example
    - if ( a.hashCode( ) == b.hashCode( ) )
      result = a.equals( b );
    else result = false;
  - Efficient if hashCode( ) is faster than equals( )