CMSC 132: Object-Oriented Programming II

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

<table>
<thead>
<tr>
<th>v_1</th>
<th>v_2</th>
<th>v_3</th>
<th>v_4</th>
<th>...</th>
<th>v_n</th>
</tr>
</thead>
</table>

Hash table h

<table>
<thead>
<tr>
<th>f(k_1)</th>
<th>f(k_2)</th>
<th>f(k_3)</th>
<th>f(k_4)</th>
<th>...</th>
</tr>
</thead>
</table>

Hash function f
Hash Tables

**Hash Table**
- Array indexed using hash values
- Hash table $A$ with size $N$
- Indices of $A$ range from $0$ to $N-1$
- Store in $A[\text{hashValue} \mod N]$

```
<table>
<thead>
<tr>
<th>Location</th>
<th>Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$\Lambda$</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>$\Lambda$</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
```
Hash Buckets

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)

Collisions
- Assign 4 parking spaces to 5 people using
  \[ \text{bucket} = \text{keyValue} \mod 4 \]

Properties of a Good Hash Function
- Distributes (scatters) values nearly 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
- Ideally, most sets of keys encountered in practice will be somewhat uniformly distributed
- There always exists some set of keys that will all collide to the same bucket
  - You hope that such cases don't come up in practice
Mapping hashes to buckets

- Given a hash value of keyHash
- and an array of N hash buckets
- can use Math.abs(keyHash % N) for bucket
  - works well if keys are uniformly distributed
  - but they generally are not
- can use Math.abs(keyHash * largePrime % N)
  - works better (assuming N != largePrime)
- A more robust scattering function works for non-uniform hash functions
Mapping hashes to buckets

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**

  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

<table>
<thead>
<tr>
<th>0</th>
<th>kiwi</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>banana</td>
</tr>
<tr>
<td>3</td>
<td>watermelon</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>apple</td>
</tr>
<tr>
<td>6</td>
<td>mango</td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>grapes</td>
</tr>
<tr>
<td>9</td>
<td>strawberry</td>
</tr>
</tbody>
</table>
Hash Function

Suppose now

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

Collision

Same hash value for multiple keys

<table>
<thead>
<tr>
<th>Hash Value</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>apple</td>
</tr>
<tr>
<td>4</td>
<td>mango</td>
</tr>
<tr>
<td>5</td>
<td>grapes</td>
</tr>
<tr>
<td>6</td>
<td>strawberry</td>
</tr>
</tbody>
</table>
Dealing with collisions

- **Chaining**
  - each bucket is a linked list of entries that were hashed to the same bucket

- **Open addressing**
  - if the first bucket is full, go to "next" bucket
  - various algorithms for calculating "next" bucket to look at
  - linear probing ("next" = +1) is simplest, but doesn't work well in practice

- Collisions are bad if the hashtable is too full, or if the hash function is bad
Beware of % (Modulo Operator)

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

- if \( x \% y \) nonzero 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

- Safe if \( N \) is a power of 2
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

- Derived from address of object in memory
- but handles garbage collection (moves objects)

Can override with new user definition

- Must work with `equals()`
- Must satisfy the “hash code contract”
Java Hash Code Contract

- Java Hash Code Contract
  
  if a.equals(b) == true, then we must guarantee
  
a.hashCode() == b.hashCode()

- Inverse is not true

  !a.equals(b) does not imply
  
a.hashCode() != b.hashCode()

- Converse is also not true

  a.hashCode() == b.hashCode()
  
does not imply a.equals(b) == true

- hashCode()
  
  - must not change unless value changes
  
  - keys inserted into a hash table must not change
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
    - equal elements wind up in separate buckets
- 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 bad but valid hashCode( ) functions
  - For pair of Strings
    - 1\textsuperscript{st} letter of 1\textsuperscript{st} str
    - 1\textsuperscript{st} letter of 1\textsuperscript{st} str + 1\textsuperscript{st} letter of 2\textsuperscript{nd} str
    - Length of 1\textsuperscript{st} str + length of 2\textsuperscript{nd} str
    - $\sum$ letter(s) of 1\textsuperscript{st} str + $\sum$ letter(s) of 2\textsuperscript{nd} str