Hashing

Department of Computer Science
University of Maryland, College Park
Introduction

• If you need to find a value in a list what is the most efficient way to perform the search?
  • Linear search
  • Binary search
  • Can we have O(1)?
# Hashing

- **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**

![Hash Table and Hash Function Diagram](image-url)
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 $A[\text{hashValue} \mod N]$
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
- 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

\[
\begin{array}{c|c}
0 & kiwi \\
1 & \\
2 & banana \\
3 & watermelon \\
4 & \\
5 & apple \\
6 & mango \\
7 & \\
8 & grapes \\
9 & strawberry \\
\end{array}
\]
### 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></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>grapes</td>
</tr>
<tr>
<td>8</td>
<td>strawberry</td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>
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 \text{Math.abs}(x \% N)\ and not \text{Math.abs}(x) \% N
- About absolute value in Java
  - \text{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

• 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()
  (Though Java libraries may be more efficient)

• Converse is also not true
  a.hashCode() == b.hashCode()
  does not imply a.equals(b) == 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 $\rightarrow$ 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
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);
}
```