CMSC 132: OBJECT-ORIENTED PROGRAMMING II

Hashing

Department of Computer Science
University of Maryland, College Park

Announcements

- Video “What most schools don’t teach”
  - http://www.youtube.com/watch?v=nKIu9yen5nc

Introduction

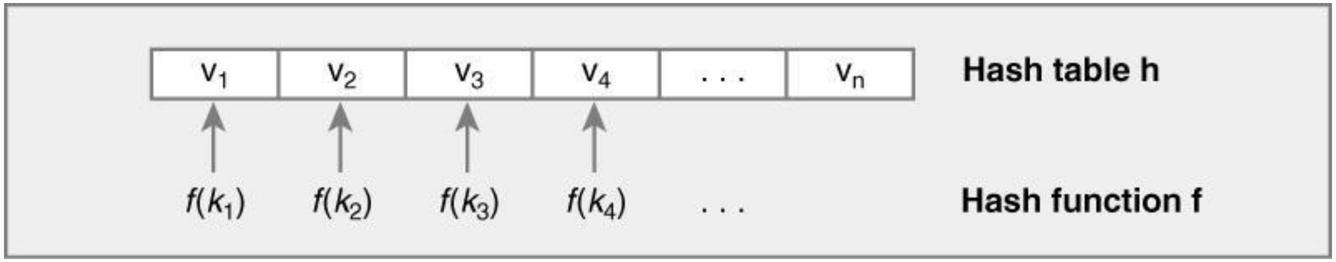
- If you need to find a value in a list what is the most efficient way to perform the search?
  - Linear search
  - Binary search
  - Can we have O(1)?

Hashing

- Remember that modulus allows us to map a number to a range
  - X % N → value between 0 and N - 1
- Suppose you have 4 parking spaces and need to assign each resident a space. How can we do it?
- parkingSpace(ssn) = ssn % 4
- Problems??
  - What if two residents are assigned the same spot?
- What if we want to use name instead of ssn?
  - Generate integer out of the name

Hashing

- Hashing
  - **Hashing function** → function that maps data to a value (e.g., integer)
  - **Hash Code/Hash Val**ue → value returned by a hash function
  - **Hash Table** → Array indexed using hash values
  - 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** (e.g., name, ssn) into number (hash Value) used as index in hash table (store in A[hashValue % N])



Hashing

- **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
- 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

0	kiwi
1	
2	banana
3	watermelon
4	
5	apple
6	mango
7	
8	grapes
9	strawberry

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

0	kiwi
1	
2	banana
3	watermelon
4	
5	apple
6	mango
7	
8	grapes
9	strawberry

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) 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

- **hashCode() method**
  - Part of the **Object** class
  - Provides hashing support by returning a hash value for any object
  - 32-bit signed int
- **Default hashCode( ) implementation** → Usually just address of object in memory
- **Using hashCode**

```
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);
}
```

- **If you override equals you need to make sure the** "hash code contract" is satisfied

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
  - **Example:** 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^{st} letter of 1^{st} str
    - 1^{st} letter of 1^{st} str + 1^{st} letter of 2^{nd} str
    - Length of 1^{st} str + length of 2^{nd} str
    - $\sum$ letter(s) of 1^{st} str + $\sum$ letter(s) of 2^{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

```
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);
```