CS311 Data Structures
Lecture 09 — Hash

Jyh-Ming Lien

March 7, 2017
A Small Problem

- Small office building, 50 offices
- Office numbers 0-49
- Building owner wants to track which offices are occupied along with names of occupants
  - Office 32 Unoccupied
  - Office 43 Fakebook Inc
  - Office 19 Unoccupied
  - Office 9 Banana Corp
- **Suggest** a standard data structure and how one would manipulate it
Arrays Rock, except...

- Small office building, 50 offices
- Office numbers based on floor
  - Floor 1: 101, 102, 103,...,110
  - Floor 2: 201, 202, 203,...,210
- Building owner wants to track which are occupied/names of occupants
  - Office 402 Unoccupied
  - Office 503 Fakebook Inc
  - Office 209 Unoccupied
  - Office 109 Banana Corp
- Adapt the earlier approach with arrays: difficulties?

How about Reverse Lookup:
- ”Fakebook Inc” → Office 403
- ”Banana Corp” → Office 109
Hash Tables ≈ Dictionaries (Python)
Also called associative arrays, sometimes maps
Store objects in an array in a retrievable way
Involves computing a number for objects to be stored
Have $O(1)$ add(x)/remove(x) (sort of...)
Hash Tables are Simple

Succinctly
- Have \( x \) (object) to put in a hash table
- Compute integer \( xhc \) from \( x \)
  (hash code for \( x \) computed via a hash function provided by class of \( x \))
- Put \( x \) in array \( hta \) at index \( xhc \): \( hta[xhc] = x; \)
- \( x \) is now in the hash table

Things to consider
1. How do you compute \( xhc \)? Where should that code exist?
2. What if \( xhc \) is beyond of \( hta.length \)?
3. What if \( hta[xhc] \) is occupied?
Every object in Java has a `hashCode()` method

- Why?
- How are hash codes computed by default?
- Official Docs

Override `hashCode()`

- For your own classes, override default `hashCode()`
- Compute hash based on the internal data of an object
- Return an integer ”representing” the object
- Class is now ”hashable”
Computing a Hash Code

Hash Code from Hash Function
- An integer computed for an object
- Computed via a function provided by an object:
  ```java
  int hc = thing.hashCode();
  ```

Hash Contract
- If `x.equals(y)` is true, then `x.hashCode()==y.hashCode()`
- Equal object → Same hash code
- Important: If `x.equals(y)` is false, hash codes may be different or the same
  - May be `x.hashCode()==y.hashCode()`
  - May be `x.hashCode()!=y.hashCode()`
- Leads to collisions in a hash table
Goals of a Hash Function

1. Adhere to the Hash Contract
   - If \( x \) and \( y \) are equal, \textbf{must} have same hash code

2. Distribute different objects "fairly" across integers
   - If \( x \) and \( y \) not equal, \textbf{try} to make \( x \).hashCode() different from \( y \).hashCode()
   - Making hash codes different reduces collisions in hash tables

3. Compute \( x \).hashCode() as quickly as possible
   - Adding/looking up objects in a hash table requires computation of an object’s hash code
   - Reducing time spent on computing hash code improves performance

These three goals almost always involve \textbf{tradeoffs}
Discussion: Hash Codes for these Fine Fellows?

public int hashCode()

Ideas for hashCode() implementation of the following things

**Fundamental Types**
- Integer
- Long
- Character
- Boolean
- Float
- Double

**Custom Classes**
- class Initials{
  char first, last;
}
- class Coord{
  int row, col;
}
Hash Codes for 64-bit Primitives

Straight from the Java class library source code

```java
package java.lang;
public final class Double
    extends Number implements Comparable<Double>
{
    @Override
    public int hashCode() {
        return Double.hashCode(value);
    }

    public static int hashCode(double value) {
        long bits = doubleToLongBits(value);
        return (int)(bits ^ (bits >>> 32)); // ^ is XOR
    }
}

Why XOR? What does (int)(a long number) do?
```
class String {

    public int hashCode(){ .. }
    Returns a hash code for this string. The hash code for a String object is computed as

    \[ s[0] \times 31^{(n-1)} + s[1] \times 31^{(n-2)} + \ldots + s[n-1] \]

    using int arithmetic, where \( s[i] \) is the ith character of the string, \( n \) is the length of the string, and ^ indicates exponentiation.
}

Examples

> "a".hashCode() 97
> String s = "Hash!";
> s.hashCode() 69497011
> "b".hashCode() 98
> (31*31*31*31)*'H' + (31*31*31)*'a' +
> "ab".hashCode() 3105
> (31*31)*'s' + (31)*'h' + '!
> "ba".hashCode() 3135
> 69497011
Polynomial Hash Code Tricks

String uses a polynomial hash code

\[ a_0X^{n-1} + a_1X^{n-2} + a_2X^{n-3} + \cdots + a_{n-1}X^0 \]

31 is $X$ in the above
- 31 is not special
- Early java used 37 instead

A Trick

Can regroup a polynomial of any degree

**Example** of regrouping degree 3 polynomial

\[ a_0X^3 + a_1X^2 + a_2X^1 + a_3 \]

regrouped becomes

\[ (((a_0)X + a_1)X + a_2)X + a_3 \]
Implementations

Slow: Original

```
s[0] \times 31^{(n-1)} + s[1] \times 31^{(n-2)} + ... + s[n-1]
```

cchar s[];
public int hashCode() {
    int h = 0, i, n=s.length;
    for(i=0; i<n; i++){
        h += s[i] \times ((int) pow(31,n-i-1));
    }
    return h;
}
```

Faster: Exploit Regrouping

```
(...((s[0])*31
    + s[1])*31
    + s[2])*31
    + ...)
```

cchar s[];
public int hashCode() {
    int h = 0, i;
    for (i=0; i<s.length; i++){
        h = 31 \times h + s[i];
    }
    return h;
}
```

Examine parens carefully in expression
The Full Implementation uses Caching

Compute once, save for later

class String{
    private char[] str; // Chars of string
    private int hash;   // Default to 0

    public int hashCode() {
        // Check if the hash has already been computed
        if(this.hash!=0 || this.str.length==0){
            return this.hash;
        }
        // Hasn’t been computed, compute and store
        for(int i=0; i < this.str.length; i++) {
            this.hash = 31 * this.hash + this.str[i];
        }
        return this.hash;
    }
}

Not exactly how java.util.String looks but it’s the general idea
public int hashCode()

Ideas for `hashCode()` implementation of the following things

### Fundamental Types (Done)
- Integer
- Long
- Character
- Boolean
- Float
- Double

### Container Types
- Integer []
- Double []
- String []
- ArrayList<T>
- LinkedList<T>
- class Flurb{
  int x;
  double y;
  String s;
  int [] a;
}
class Flurb{
    int x;
    double y;
    String s;
    int [] a;

    public int hashCode(){
        int h = 0;
        h = h*31 + x;
        h = h*31 + (new Double(y)).hashCode();
        h = h*31 + s.hashCode();
        for(int i=0; i<a.length; i++){
            h = h*31 + a[i];
        }
        return h;
    }
}

Example: Flurb Class hashCode()
Basic hashCode() Strategy

Poor man’s strategy: x.toString().hashCode()
More thoroughly ...

**Fundamental Types**
- All have a fixed size in bytes
- `int` has 4 bytes
- Convert bytes of intrinsic to 4 bytes
- If shorter than 4 bytes like `Character`, done
- If 8 bytes like `Long`, `Double`, use XOR to reduce 8 to 4 bytes

**Container Types**
- Use String approach
- Polynomial hash code of elements
- For each element compute its hash code
- Update polynomial hash code
- Treat fields as part of the sequence
Two equal objects must have the same `hashCode()` and as much as possible unequal objects should have differing hashcodes.

Consequently, every class has a `hashCode()` method but should override it when overriding `equals()`.

Fundamental types with 32 bits or less like `Integer` are their own hash codes.

Fundamental types with more than 32 bits like `Long` can use XOR to combine 4-byte quantities to get a 32-bit hash.

Aggregate data like `String` often uses polynomial codes to calculate hash codes which differ when the order of constituents changes.

The same approach is used for other containers and custom classes that need the order of elements reflected in their hashcodes.