Hashing

Iván Szabolcs

2017 tavasz

Cél:

- `Set[Key]`: `Key` típusú objektumok egy halmazát tárolni
  - `set.add( k : Key ) : Unit`
  - `set.contains( k : Key ) : Boolean`
  - `set.remove( k : Key ) : Unit` – esetleg
- `Map[Key,Value]`: `Key` kulcsokhoz rendelt `Value` értékeket tárolni
  - `map.contains( k : Key ) : Boolean`
  - `map.get( k : Key ) : Value`
  - `map.set( k : Key, v : Value ) : Unit`
  - `map.remove( k : Key ) : Unit` – esetleg

Ilyet már csináltunk, pl. piros-fekete fával rendezhető értékeknek (mondjuk `Int`eknek) a halmazát tudjuk tárolni

- Ha tudunk `Map`ot, akkor `Set`et is (a `Set` felfogható mint egy `Map[Key,Unit]`, ahol `Unit` valami ,,dummy" adattípus)
- Pl. PHP-ban az `array` is ,,asszociatív tömb", tkp. egy `Map[Any,Any]`

- Ha rendezhető a kulcstípus, akkor $O(\log n)$ időben megoldható legrosszabb / átlagos esetben
- Ennél szeretnénk gyakorlatban jobbat
- Nem mindig tudunk ,,értelmesen'' rendezni

A hash kód lényege: egy objektumhoz rendeljünk egy intet úgy, hogy különböző objektumoknak általában (jó eséllyel, stb) különböző legyen a hash kódja.

Mire jó, ha van hash függvényünk:

- Hash table – egy tömb
- Mondjuk `Set[Key]`t akarunk implementálni – egy `Array[Key]` tömb, mondjuk N méretű
- Olyan hash függvény kell, ami egy kulcsból készít egy $0 \ldots N-1$ közé eső intet
- A hash table mérete változhat menet közben
- Legegyszerűbb módszer: a hash függvény készít egy intet és `mod` N

A `k` kulcsot a tömb $\mathrm{hash}(\mathtt{k}) \bmod N$. slotjába írjuk be:

0	1	2	3	4	5	6	7

- `"bar".hashCode() = 97299, 97299%8 = 3, insert`

0	1	2	3	4	5	6	7
			true				

- `"red".hashCode() = 112785, 112785%8 = 1, contains?` false
- `insert("red")`

0	1	2	3	4	5	6	7
	true		true				

- `"bab".hashCode() = 97283, 97283%8 = 3, contains?` true – hopsz

Magát a kulcsot írjuk be a slotba:

0	1	2	3	4	5	6	7

- `"bar".hashCode()` = 97299, 97299%8 = 3, `insert`

0	1	2	3	4	5	6	7
			bar				

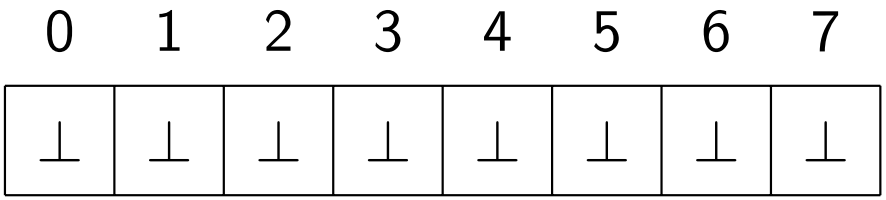
- `"red".hashCode()` = 112785, 112785%8 = 1, `contains?` false
- `insert("red")`

0	1	2	3	4	5	6	7
	red		bar				

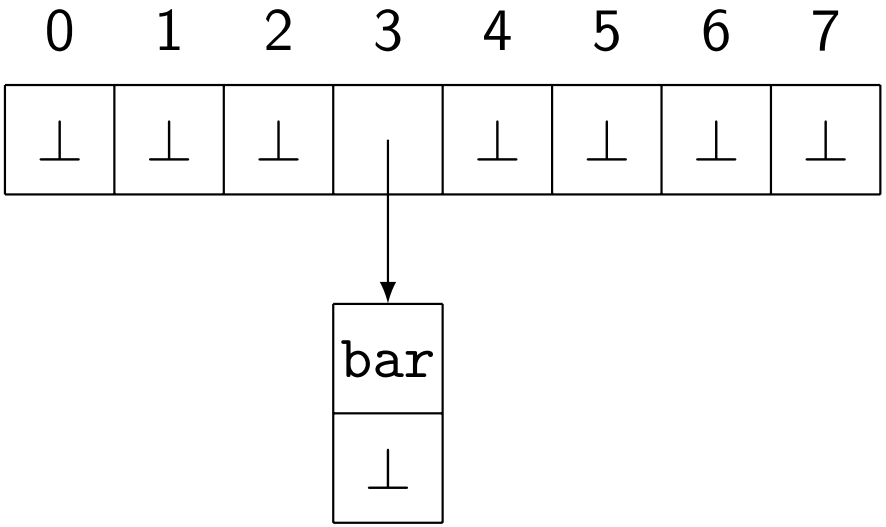
- `"bab".hashCode()` = 97283, 97283%8 = 3, `contains?` false
- `insert("bab")` – hopsz

A lehetséges ütközéseket (collision) fel kell oldjuk

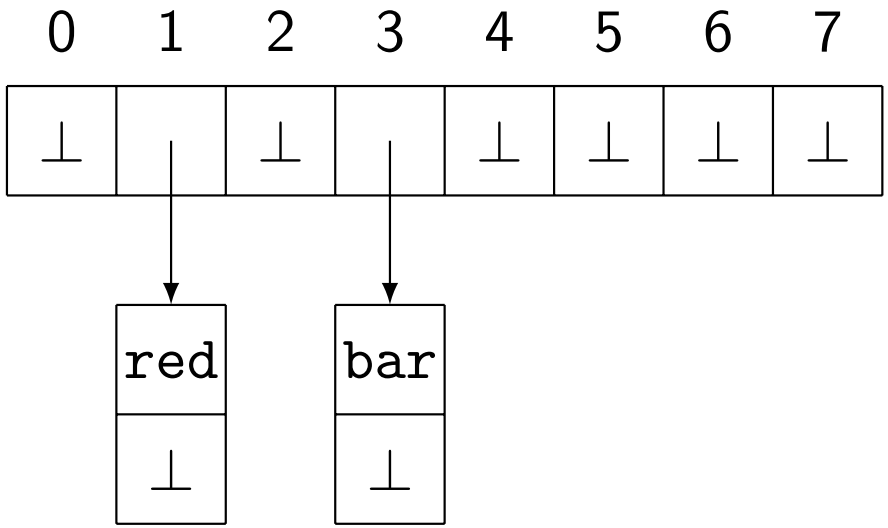
- Egy módszer: chaining
- A tömb egy cellájában egy pointer van egy láncolt listára, ami az oda eső kulcsokat tárolja



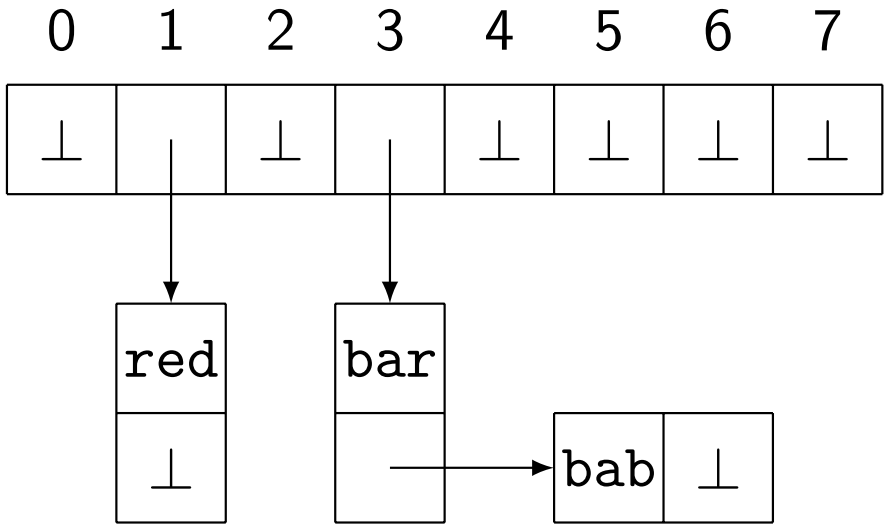
- `insert("bar")` (hash: 97299 $\Rightarrow$ 3)



- `insert("red")`, hash: 112785 $\Rightarrow$ 1



- `insert("bab")`, hash: 97283 $\Rightarrow$ 3



- Jó, ha rövidek a láncok
- Load factor: vödrök (buckets) száma / elemek száma
- Ha a load factor túl nagy $\Rightarrow$ több vödör
- Kétszer (vagy konstansszor) annyi vödör $\Rightarrow$ amortizált konstans
- Ha a rehash nem túl drága

```
import scala.collection.mutable.HashSet

object Stringes extends App {
  val set = new HashSet[String]()
  val red = "red"
  val red2 = "red"
  set add red
  println( set contains red2 ) //true
  set add "bar"
  println( set contains "bar" ) //true
  println( set contains "bab" ) //false
}
```

Amikor a több vödör nem segít: ha ugyanaz a hash code

```
println( "AaBBAa".hashCode() ) //1952508096
println( "BBAaAa".hashCode() ) //1952508096
println( "BBBBAa".hashCode() ) //1952508096
println( "AaAaAa".hashCode() ) //1952508096
println( "AaBBBB".hashCode() ) //1952508096
println( "BBAaBB".hashCode() ) //1952508096
println( "BBBBBB".hashCode() ) //1952508096
println( "AaAaBB".hashCode() ) //1952508096
```

Java-ban a `String.hashCode()`:

$$s[0] \cdot 31^{n-1} + s[1] \cdot 31^{n-2} + \ldots + s[n-1]$$

$$\texttt{"Aa".hashCode()} == 31 * 65 + 97 == 2112$$
$$== 31 * 66 + 66 == \texttt{"BB".hashCode()}$$

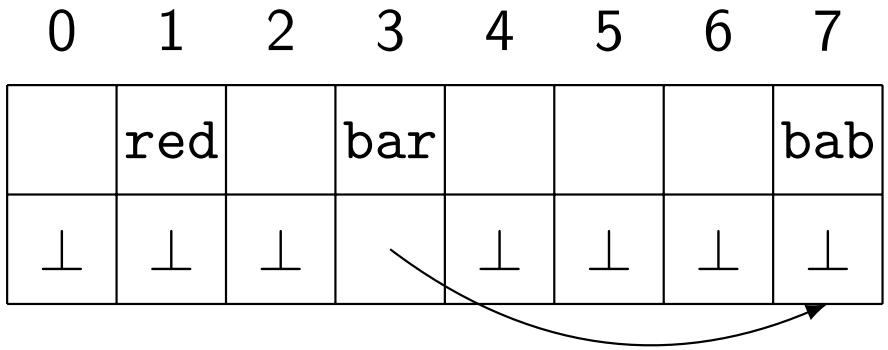
,,A Tomcat 6.0.32 server parses a 2 MB string of colliding keys in about 44 minutes of i7 CPU time, so an attacker with about 6 kbit/s can keep one i7 core constantly busy. If the attacker has a Gigabit connection, he can keep about 100.000 i7 cores busy”

- $\Rightarrow$ easy DOS támadás Tomcat webszerverek ellen (2011)
- ,,Fix”: max ötven paraméter

A Java 1.8 már láncolt lista helyett kiegyensúlyozott keresőfát használ a default HashSet implementációban

- Másik lehetőség láncolt lista helyett: open addressing
- Minden vödörbe max egy kulcsot teszünk
- A hash kód alapján számolt index csak kiindulópont
- Ha foglalt, akkor valamilyen módszerrel végigpróbálunk másokat
  - linear probing: `probe( h, k ) = h +` $c \cdot k$
  - quadratic probing: `probe( h, k ) = h +` $c_1 \cdot k + c_2 \cdot k^2$
  - double hashing: `probe( h, k ) =` $h + k \cdot h_2$
- A linear probingnél a ,,clustering" probléma, quadraticnál kevésbé, double hashingnál még kevésbé
- Pl. a Berkeley Fast File System-ben quadratic probing van
- Ha a load factor kb. 0.7re növekszik, mindegyik módszer meglassul

- A chaining ellen a plusz memória a fő érv
- Egy ,,hibrid" módszer: coalesced hashing
- Ütközésnél betesszük valahova máshova a táblában és odalinkelünk
- Kevesebb pointer



- Ahogy szokták: cellart használnak
  - csak az első N helyre hashelnek
  - a többi $M - N$ hely a ,,cellar", ezt tisztán ütközésfeloldásra tartják
  - a legtöbb load factorra a kb. $N = 0.86M$ közel optimális

- Még egy módszer: Cuckoo hashing
- Két hash függvény, h_1 és h_2
- Ha bármelyik üresre dobja $\Rightarrow$ oda tesszük
- Ha egyik sem $\Rightarrow$ a h_1 slotjába tesszük, onnan ,,kilökjük" az ott levőt
- A kilökött elemet a másik hash függvény szerint osztjuk be
- Ezt iteráljuk
- Ha ez egy $\log n$-es thresholdnál tovább tart, új hash függvényekkel újraépítjük az egészet
- Ha a load factor 0.5 alatt van, akkor ez várhatóan konstans sok lépés
- Ha három hash függvény van, akkor a load factor felmehet 0.91-re
- Ha egy helyre két elem fér, úgy pedig 0.8-ra
- Ha a tábla befér a cache-be, akkor ez $3 - 5\times$ gyorsabb, mint a chained hashing (2008)

Törlés: na az nem jó

```
import scala.collection.mutable.HashSet

class Coord( var x : Int, var y : Int ){
}

object Main extends App {

  val set = new HashSet[ Coord ]()

  val p = new Coord(100,100)
  set.add( p )
  println( set contains p )

  val q = new Coord(100,100)
  println( set contains q )

  val r = p
  println( set contains r )
}
```

```scala
import scala.collection.mutable.HashSet

class Coord( var x : Int, var y : Int ){
}

object Main extends App {

  val set = new HashSet[ Coord ]()

  val p = new Coord(100,100)
  set.add( p )
  println( set contains p ) //true

  val q = new Coord(100,100)
  println( set contains q ) //false

  val r = p
  println( set contains r ) //true
}
```

```
import scala.collection.mutable.HashSet

class Coord( var x : Int, var y : Int ){
  override def equals( that : Any ) = that match {
    case that : Coord => (that.x==this.x)&&(that.y==this.y)
    case _ => false
  }
}

object Main extends App {
  val set = new HashSet[ Coord ]()
  val p = new Coord(100,100)
  set.add( p )
  println( set contains p )
  val q = new Coord(100,100)
  println( set contains q )
}
```

```scala
import scala.collection.mutable.HashSet

class Coord( var x : Int, var y : Int ){
  override def equals( that : Any ) = that match {
    case that : Coord => (that.x==this.x)&&(that.y==this.y)
    case _ => false
  }
}

object Main extends App {
  val set = new HashSet[ Coord ]()
  val p = new Coord(100,100)
  set.add( p )
  println( set contains p ) //true
  val q = new Coord(100,100)
  println( set contains q ) //false
}
```

```
import scala.collection.mutable.HashSet

class Coord( var x : Int, var y : Int ){
  override def hashCode = x ^ y
}

object Main extends App {
  val set = new HashSet[ Coord ]()
  val p = new Coord(100,100)
  set.add( p )
  println( set contains p )
  val q = new Coord(100,100)
  println( set contains q )
}
```

```scala
import scala.collection.mutable.HashSet

class Coord( var x : Int, var y : Int ){
  override def hashCode = x ^ y
}

object Main extends App {
  val set = new HashSet[ Coord ]()
  val p = new Coord(100,100)
  set.add( p )
  println( set contains p ) //true
  val q = new Coord(100,100)
  println( set contains q ) //false
}
```

```scala
import scala.collection.mutable.HashSet

class Coord( var x : Int, var y : Int ){
  override def hashCode = x ^ y
  override def equals( that : Any ) = that match {
    case that : Coord => (that.x==this.x)&&(that.y==this.y)
    case _ => false
  }
}
object Main extends App {
  val set = new HashSet[ Coord ]()
  val p = new Coord(100,100)
  set.add( p )
  println( set contains p )
  val q = new Coord(100,100)
  println( set contains q )
}
```

```scala
import scala.collection.mutable.HashSet

class Coord( var x : Int, var y : Int ){
  override def hashCode = x ^ y
  override def equals( that : Any ) = that match {
    case that : Coord => (that.x==this.x)&&(that.y==this.y)
    case _ => false
  }
}
object Main extends App {
  val set = new HashSet[ Coord ]()
  val p = new Coord(100,100)
  set.add( p )
  println( set contains p ) //true
  val q = new Coord(100,100)
  println( set contains q ) //true! ^_^
}
```

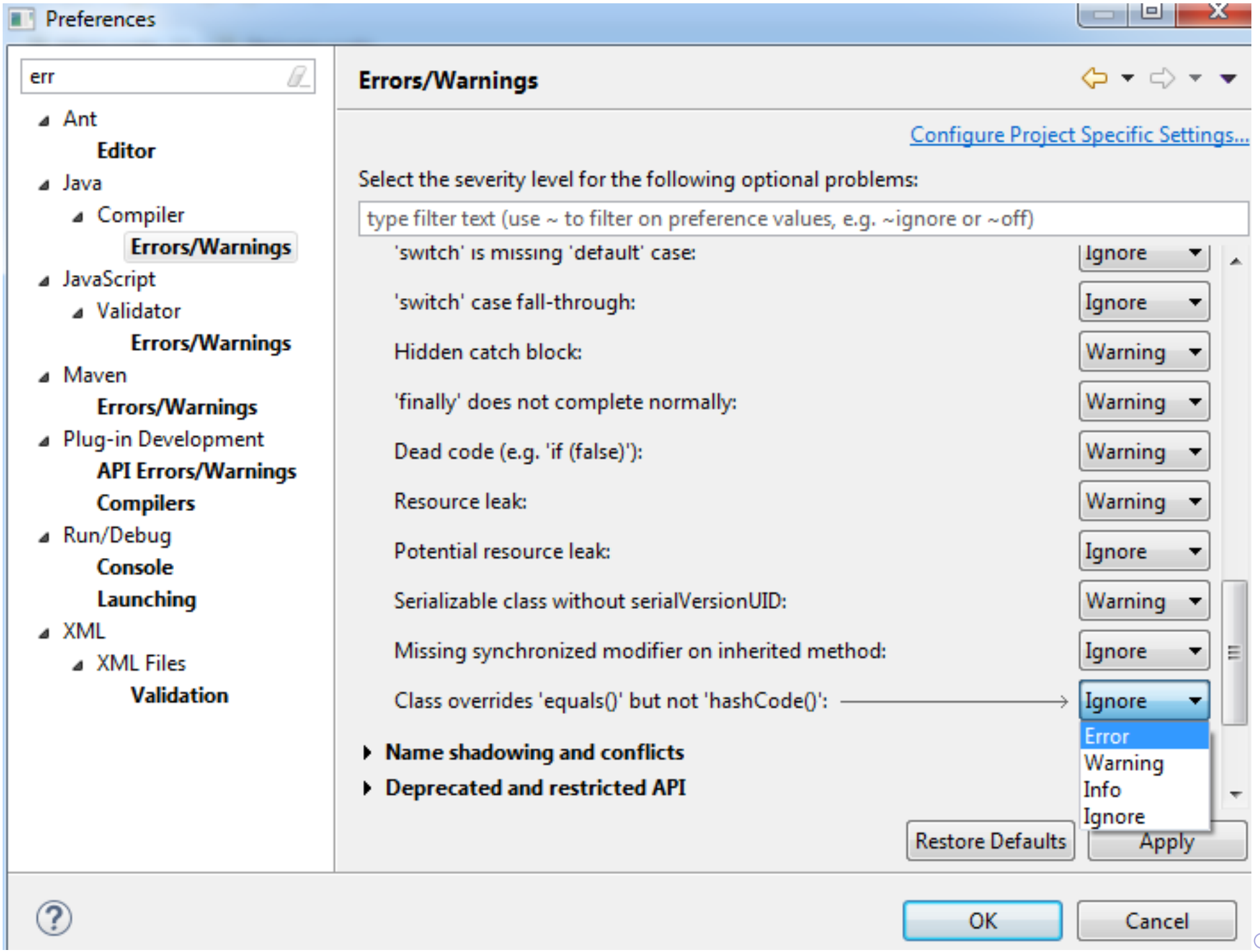
```scala
import scala.collection.mutable.HashSet
class Coord( var x : Int, var y : Int ){
  override def hashCode = x ^ y
  override def equals( that : Any ) = that match {
    case that : Coord => (that.x==this.x)&&(that.y==this.y)
    case _ => false
  }
}
object Main extends App {
  val set = new HashSet[ Coord ]()
  val p = new Coord(100,100)
  set.add( p )
  println( set contains p ) //true
  p.x = 50
  println( set contains p )
  val q = new Coord(50,100)
  println( set contains q )
  val r = new Coord(100,100)
  println( set contains r )
}
```

```scala
import scala.collection.mutable.HashSet
class Coord( var x : Int, var y : Int ){
  override def hashCode = x ^ y
  override def equals( that : Any ) = that match {
    case that : Coord => (that.x==this.x)&&(that.y==this.y)
    case _ => false
  }
}
object Main extends App {
  val set = new HashSet[ Coord ]()
  val p = new Coord(100,100)
  set.add( p )
  println( set contains p ) //true
  p.x = 50
  println( set contains p ) //false
  val q = new Coord(50,100)
  println( set contains q ) //false
  val r = new Coord(100,100)
  println( set contains r ) //false dafuq
}
```

Tanulságok:

- A `hashCode` és az `equals` metódusokat mindig párban felülírni
- A `hashCode` soha ne változhasson az objektum életciklusa alatt
- Ha két objektum lehet `equals` bármikor élettartamuk során, akkor a `hashCode`-juk ugyanaz kell legyen

Tehát `hashCode`eot csak immutable mezőkből szabad számítani, különben ez lesz


Preferences
err
Ant
Editor
Java
Compiler
Errors/Warnings
JavaScript
Validator
Errors/Warnings
Maven
Errors/Warnings
Plug-in Development
API Errors/Warnings
Compilers
Run/Debug
Console
Launching
XML
XML Files
Validation
Errors/Warnings
Configure Project Specific Settings...
Select the severity level for the following optional problems:
type filter text (use ~ to filter on preference values, e.g. ~ignore or ~off)
'switch' is missing 'default' case: Ignore
'switch' case fall-through: Ignore
Hidden catch block: Warning
'finally' does not complete normally: Warning
Dead code (e.g. 'if (false)'): Warning
Resource leak: Warning
Potential resource leak: Ignore
Serializable class without serialVersionUID: Warning
Missing synchronized modifier on inherited method: Ignore
Class overrides 'equals()' but not 'hashCode()': Ignore
Error
Warning
Info
Ignore
Name shadowing and conflicts
Deprecated and restricted API
Restore Defaults
Apply
OK
Cancel

- a hashCode contract Javában:

  > Objects that are equal must have the same hash code within a running process.

- Tehát: ugyanannak az alkalmazásnak két különböző futtatásakor is lehet különböző hashkódja ugyanannak az objektumnak!
- Pl. a Google ProtocolBuffer-ei ilyenek
- $\Rightarrow$ ne használj hashkódot elosztott alkalmazásokban!

A ,,jó" hash függvény milyen?

Lesz matek is, de előbb

- `Objects.hash(Object[] objects)` : mint a Stringekre, 31-es alapú számrendszerbeli összegzés
- `Arrays.hashCode()` : same

Ha gyorsan kell egy hash függvény, akkor ez algebrai adattípusokra gyorsan ad egy jó megoldást

```
class Coord( var x : Int, var y : Int ){
   override def hashCode = Objects.hash( x, y )
   override def equals( that : Any ) = that match {
      case that : Coord => (that.x==this.x)&&(that.y==this.y)
      case _ => false
   }
}
```

Algebrai adattípus?

- veszünk pár ,,alap" immutable adattípust
- típusok szorzata: egy olyan típus, aminek a megadott típusok adják egy-egy mezőjét

Pl. a `Coord` az egy `Int x Int` típus

- típusok összege: egy olyan típus, aminek értékkészlete a megadott típusok értékkészletének a diszjunkt uniója

Tipikusan erre való Javában az `interface`: az interface típus értékkészlete az összes őt implementáló osztály lehetséges értékeinek uniója

- Rekurzív típusdefiníció megengedett

Példa: lista

- List = Nil + (int x List)
- Nil = () (üres szorzat)

```
interface List;
class Nil implements List {}
class Link implements List{
  public final int head;
  public final List tail;
  public Link( int head, List tail ){
    this.head = head; this.tail = tail;
  }
}
```

```
List list142 = new Link( 1, new Link(4, new Link(2, new Nil())));
System.out.println( ((Link)list142).head ) //1
System.out.println( (Link)(((Link)list142).tail).head ) //4
```

Példa: fa

- `Tree  = Empty + (Tree x int x Tree)`
- `Empty = ()`

```
interface Tree;
class Empty implements Tree {} // should be ofc singleton btw
class Nonempty implements Tree{
  public final int data;
  public final Tree left, right;
  public Link( int data, Tree left, Tree right ){
    this.data = data; this.left = left; this.right = right;
  }
}
```

Példa: formula

- `Formula = Var + Or + Not + And`
- `Var = String`
- `Or  = Formula x Formula`
- `And = Formula x Formula`
- `Not = Formula`

vagy elsőrendben

- `Term     = Var + (Symbol x TermList)`
- `Var      = String`
- `Symbol   = String`
- `TermList = Nil + (Term x TermList)`
- `Nil      = ()`
- `Formula  = Atom + Or + Not + And + Forall`
- `Atom     = Symbol x TermList`

```
interface Tree;
class Empty implements Tree {}
class Nonempty implements Tree{
  public final int data;
  public final Tree left, right;
  public Link( int data, Tree left, Tree right ){
    this.data = data; this.left = left; this.right = right;
  }
  public int hashCode(){
    return Objects.hash( data, left, right );
  }
  public boolean equals( Object that ){
    if( this == that ) return true;
    if(!(that instanceof Link)) return false;
    thatLink = (Link)that;
    return( thatLink.data == this.data &&
            Objects.equals( thatLink.left,this.left ) &&
            Objects.equals( thatLink.right,this.right) );
  }
}
```

```
class Empty implements Tree{
  private static final Empty theInstance = new Empty();
  private Empty() {}
  public static Empty getInstance() = { return theInstance; }
  //singleton => equals, hashCode are fine
}
```

```
trait Tree
case object Empty extends Tree
case class Nonempty(data:Int, left:Tree, right:Tree) extends Tree
```

Bloom filter

Scenario:

- van egy nagy halmazunk (szótár etc.), ami nem fér be a memóriába
- `add`, `contains`
- minimalizálni akarjuk a lemezműveletek számát
- ehhez használunk a memóriában:
  - egy M méretű bittömböt: T
  - k darab független hash függvényt: $h_1, \ldots, h_k$, amik $0 \ldots M-1$-be képzik az objektumokat
- Egy o objektum beszúrásakor T-ben az összes $h_1(o)$, $h_2(o)$,..., $h_k(o)$ indexet 1-re állítjuk
- `contains(o)`: ha $T[h_1(o)], \ldots, T[h_k(o)]$ bármelyike 0, akkor nincs benne a halmazban, különben megnézzük a lemezt

Bloom filter analízis

- Egy o objektum beszúrásakor $h_j(o) \neq i$: $1 - \frac{1}{M}$.
- Egy o objektum beszúrásakor semelyik h_j hash függvény nem képez i-re: $\left(1 - \frac{1}{M}\right)^k$.
- N darab beszúrás után az i. bit még mindig 0: $\left(1 - \frac{1}{M}\right)^{kN}$.
- N darab beszúrás után az i. bit már 1: $1 - \left(1 - \frac{1}{M}\right)^{kN}$.
- Hogy ekkor egy lekérdezés csupa 1-est talál: $\left(1 - \left(1 - \frac{1}{M}\right)^{kN}\right)^k$.
- Ez kb: $(1 - e^{-kN/M})^k$.
- Optimalizálva k-t: $k = \frac{M}{N} \ln 2$.
- Pl. 1%-ot célozva: 9.6 bit/elem, $6 - 7$ hash függvény
- 0.0001-et: 19.2 bit/elem, $13 - 14$ hash függvény

Bloom filter hol van

- Akamai (felhőszolgáltató): csak akkor cachel egy web-objektumot, ha már a második lekérdezés érkezik rá
- Google BigTable, Apache HBase, Postgresql: adatbázis query kulcsokra
- Chrome böngésző: malicious URL-ek azonosítására, lokális Bloom filtert használt
- Bitcoin: wallet synchronization
- SPIN: az elérhető állapotok halmazának trackelésére
- Medium (tartalomszolgáltató): ne ajánljon usernek olyan cikket, amit már olvasott

Honnan lesz hash függvényünk?

- Sok ,,különböző'' hash függvényt kaphatunk egy ,,elég hosszú'' kimenetű hash függvény outputját feldarabolva
- MD5, SHA – jó és lassú
- Merkle-Damgard: blokkokra vágás, internal state, mixing függvény
- Pszeudorandom belső állapotból indulva jó mixing függvénnyel más-más hasheket kapunk

something like

```
uint32_t mix( uint32_t internal_state, uint32_t message_block )
{
  return (internal_state * message_block) ^
         (internal_state << 3 + message_block >> 2);
}
```