Carleton University

Alina Shaikhet

# COMP 2402

part2

# Hash Tables

# Multiplicative Hashing

Dietzfelbinger et al. - 1997

$$\textbf{hash}(\boldsymbol{x}) = \left((\boldsymbol{z} \cdot \boldsymbol{x}) \ \textbf{mod} \ \mathbf{2}^{\boldsymbol{w}}\right) \ \textbf{div} \ \mathbf{2}^{\boldsymbol{w}-\boldsymbol{d}}$$

computer code: $\boldsymbol{z} * \boldsymbol{x} \gg (\boldsymbol{w} - \boldsymbol{d})$

right shift by $\boldsymbol{w} - \boldsymbol{d}$

- In multiplicative hashing, we use a hash table of size $\mathbf{2}^{\boldsymbol{d}}$ for some integer $\boldsymbol{d}$ (called the **dimension**): $\boldsymbol{t}.\textbf{length} = \mathbf{2}^{\boldsymbol{d}}$.
- $\boldsymbol{x}$ is an integer we want to hash: $\boldsymbol{x} \in \{\mathbf{0}, \dots, \mathbf{2}^{\boldsymbol{w}} - \mathbf{1}\}$
- $\boldsymbol{z}$ is a randomly chosen **odd** integer: $\boldsymbol{z} \in \{\mathbf{1}, \mathbf{3}, \mathbf{5}, \dots, \mathbf{2}^{\boldsymbol{w}} - \mathbf{1}\}$
- **div** operator = integer division (we discard the remainder):

  for any integers $\boldsymbol{a} \geq \mathbf{0}$, $\boldsymbol{b} \geq \mathbf{1}$, $\boldsymbol{a}\ \textbf{div}\ \boldsymbol{b} = \left\lfloor {}^{\boldsymbol{a}}/_{\boldsymbol{b}} \right\rfloor$
- Computers with **32**-bit arithmetic keep only the **lowest 32** bits of any number. So, by default, operations on integers are already done modulo $\mathbf{2}^{\boldsymbol{w}}$.

# Review

Binary multiplication:

$$\begin{array}{r} \times\; 1001 \\ 100 \;\leftarrow 2^2 \\ \hline 0000 \\ +\; 0000\phantom{0} \\ 1001\phantom{00} \\ \hline 100100 \end{array}$$

$$2^w = \underbrace{1000000\ldots0000}_{w \text{ bits}}$$

$$2^w - 1 = \underbrace{111111\ldots1111}_{w \text{ bits}}$$

right shift by 3

$$x \operatorname{div} 2^3 = 1001110 \leftarrow x \gg 3$$

$$x = 1001110101$$

$$2^8 = 100000000$$

$$x \cdot 2^8 = 100111010100000000 \leftarrow x \ll 8$$

left shift by 8

$$x = 1001110101$$

$$x \bmod 2^7 = 1110101$$

Base converter

# Example from the textbook

$$\text{hash}(x) = \big((z \cdot x) \bmod 2^{w}\big) \text{ div } 2^{w-d}$$

$$w = 32$$

$$d = 8$$

| | |
|---|---|
| $2^w$ (4294967296) | 100000000000000000000000000000000 |
| $z$ (4102541685) | 11110100100001111101000101110101 |
| $x$ (42) | 00000000000000000000000000101010 |
| $z \cdot x$ | 10100000011110010010000101110100110010 |
| $(z \cdot x) \bmod 2^w$ | 00011110010010000101110100110010 |
| $((z \cdot x) \bmod 2^w) \text{ div } 2^{w-d}$ | 00011110 |

# Multiplicative Hashing

$$\mathbf{hash}(x) = \left((z \cdot x) \ \mathbf{mod}\ 2^w\right) \ \mathbf{div}\ 2^{w-d}$$

For any randomly chosen **odd** integer $z \in \{1, 3, 5, \dots, 2^w - 1\}$,
and for any $x, y \in \{0, \dots, 2^w - 1\}$, $x \neq y$,

$$Pr(\mathbf{hash}(x) = \mathbf{hash}(y)) \leq \frac{2}{2^d}$$

$t.\mathrm{length}$

Alina Shaikhet – COMP2402 – Carleton University

# Expected length of a list

$$Pr(\text{hash}(x) = \text{hash}(y)) \le \frac{2}{t.\text{length}}$$

$$n \le t.\text{length}$$

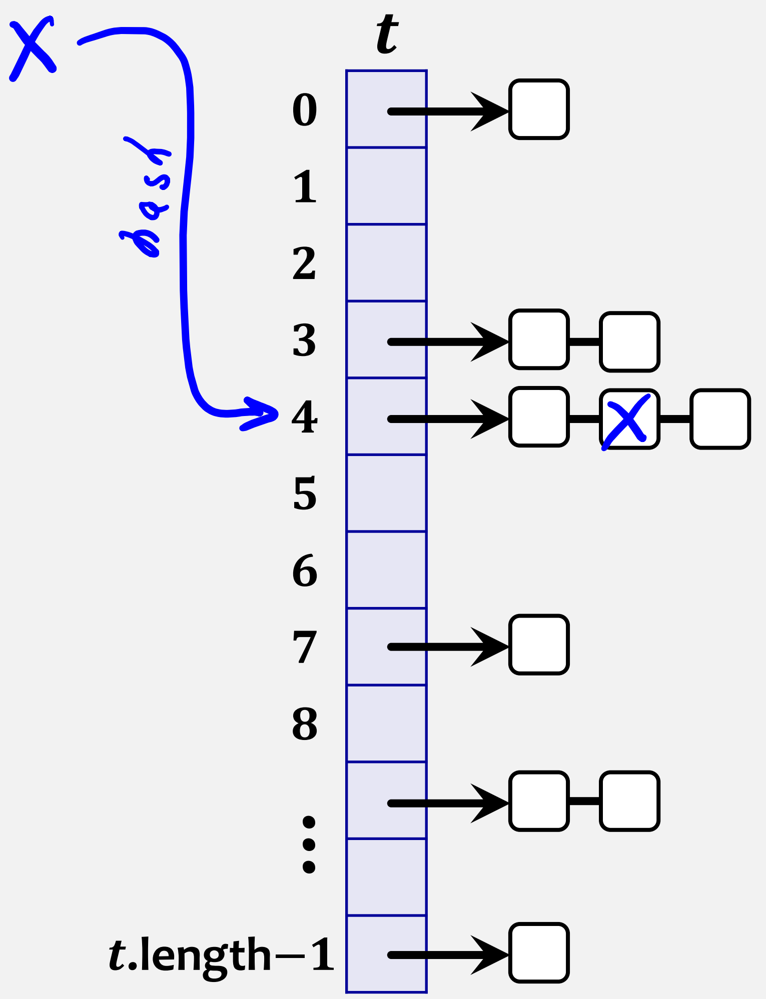
Suppose a Chained Hash Table stores a set $S \subseteq \{0, \dots, 2^w - 1\}$ of size $n$, then:

1. for any $x \in S$, the expected length of the list that contains $x$:

$$E(t[\text{hash}(x)].\text{size}()) \le 1 + \frac{2(n-1)}{t.\text{length}} \le 3$$

2. for any $x \notin S$, the expected length of the list that $x$ hashes to:

$$E(t[\text{hash}(x)].\text{size}()) \le \frac{2n}{t.\text{length}} \le 2$$




$$n_{\text{hash}(x)} \le 3$$

# Theorem 5.1

A **ChainedHashTable** implements the **USet** interface. Ignoring the cost of calls to resize(), a **ChainedHashTable** supports the operations add($\boldsymbol{x}$), remove($\boldsymbol{x}$), and contains($\boldsymbol{x}$) in $\boldsymbol{O}(\mathbf{1})$ **expected** time per operation.
Furthermore, beginning with an empty **ChainedHashTable**, any sequence of $\boldsymbol{m}$ add($\boldsymbol{x}$) and remove($\boldsymbol{x}$) operations results in a total of $\boldsymbol{O}(\boldsymbol{m})$ time spent during all calls to resize().

# Hash Codes

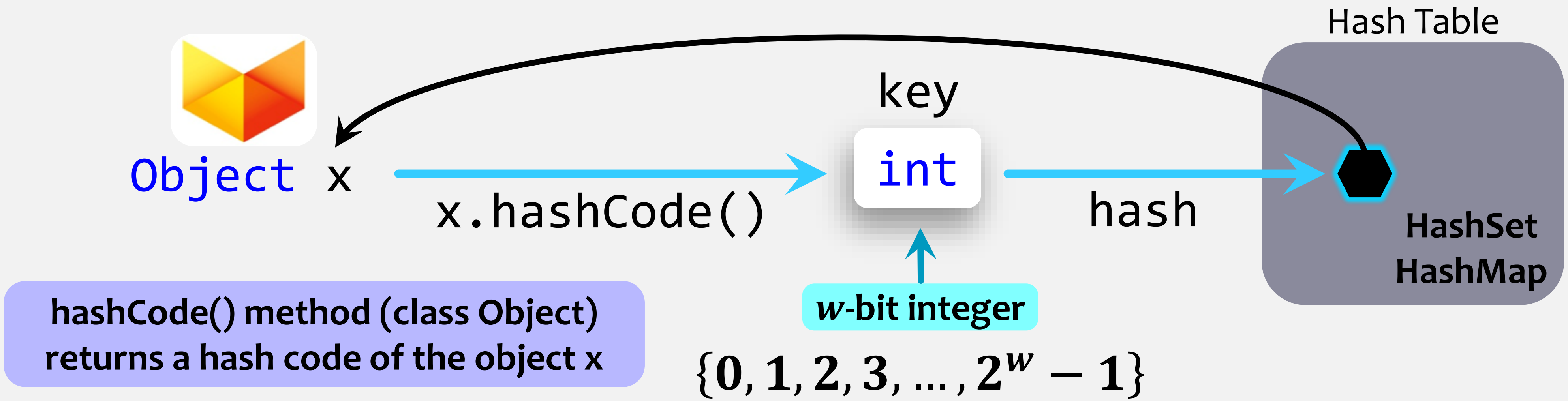
Very often hash tables store types of data that are **not integers**. They may be strings, objects, arrays, or other compound structures.

We must **map** these data types to $\boldsymbol{w}$-bit hash codes:

hashCode() returns a $\boldsymbol{w}$-bit integer in $\{\mathbf{0}, \dots, \mathbf{2^w - 1}\}$.

1. **Require:** if $x$.equals($y$) then $x$.hashCode() == $y$.hashCode()
2. **Desire:** if !$x$.equals($y$) then $x$.hashCode() != $y$.hashCode()

$$Pr(x.\texttt{hashCode()} == y.\texttt{hashCode()}) \le \frac{1}{2^w}$$



# Hashing & Java Data Types

| DATA TYPE | SIZE | DESCRIPTION |
|---|---|---|
| byte | 8 bits | Stores whole numbers from $-\mathbf{128}$ to $\mathbf{127}$ |
| char | 16 bits | Stores a single character/letter or ASCII values |
| short | 16 bits | Stores whole numbers from $-\mathbf{32,768}$ to $\mathbf{32,767}$ |
| int | 32 bits | Stores whole numbers from $-\mathbf{2,147,483,648}$ to $\mathbf{2,147,483,647}$ |
| float | 32 bits | Stores fractional numbers. Sufficient for storing $\mathbf{6}$ to $\mathbf{7}$ decimal digits |
| long | 64 bits | Stores whole numbers from $-\mathbf{9,223,372,036,854,775,808}$ to $\mathbf{9,223,372,036,854,775,807}$ |
| double | 64 bits | Stores fractional numbers. Sufficient for storing $\mathbf{15}$ decimal digits |

# Hashing & Java Data Types

**Easy**

treat these bits as the representation of an **integer** in the range $\{0, \dots, 2^{32} - 1\}$

| WRAPPER CLASSES | DATA TYPE | SIZE | BINARY REPRESENTATION |
| --- | --- | --- | --- |
| Byte | byte | 8 bits | $\{0, \dots, 2^8 - 1\} = \{0, \dots, 255\}$ |
| Character | char | 16 bits | $\{0, \dots, 2^{16} - 1\} = \{0, \dots, 65535\}$ |
| Short | short | 16 bits | $\{0, \dots, 2^{16} - 1\} = \{0, \dots, 65535\}$ |
| Integer | int | 32 bits | $\{0, \dots, 2^{32} - 1\} = \{0, \dots, 4,294,967,296\}$ |
| Float | float | 32 bits | $\{0, \dots, 2^{32} - 1\} = \{0, \dots, 4,294,967,296\}$ |
| Long | long | 64 bits | $\{0, \dots, 2^{64} - 1\}$ |
| Double | double | 64 bits | $\{0, \dots, 2^{64} - 1\}$ |

**Difficult**

Chapter 5.3.2

# Hashing & Java Data Types

| $p$ | $q$ | $p \oplus q$ |
|---|---|---|
| 1 | 1 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 0 | 0 | 0 |

Exclusive OR (XOR)

- Class Long

```
public int hashCode()
   return (int)(this.longValue()^(this.longValue()>>>32))
```

(unsigned)
bitwise right shift operator

**This is bad**
**when first 32 bits are the same as last 32 bits**

- Class Double

```
public int hashCode()
   long v = Double.doubleToLongBits(this.doubleValue());
   return (int)(v^(v>>>32));
```

https://docs.oracle.com/javase/8/docs/api/java/lang/Long.html#hashCode--
https://docs.oracle.com/javase/8/docs/api/java/lang/Double.html#hashCode--

# Hashing & Java Data Types

- Interface Map.Entry<K,V>

two entries are the same if their keys are the same and their values are the same

```
boolean equals(Object o)

    return (e1.getKey()==null ?
             e2.getKey()==null : e1.getKey().equals(e2.getKey()))  &&
            (e1.getValue()==null ?
             e2.getValue()==null : e1.getValue().equals(e2.getValue()))
```

```
int hashCode()

    return (e.getKey()==null   ? 0 : e.getKey().hashCode()) ^
          (e.getValue()==null ? 0 : e.getValue().hashCode())
```

https://docs.oracle.com/javase/8/docs/api/java/util/Map.Entry.html#equals-java.lang.Object-

# BadExamples.java

```java
long y = 1L << 32L;  // y = 2^32 = 4294967296
```

HashSet is just a HashMap in Java:

http://hg.openjdk.java.net/jdk8/jdk8/jdk/file/687fd7c7986d/src/share/classes/java/util/HashSet.java

Fix with BSTs:

https://hg.openjdk.java.net/jdk8/jdk8/jdk/file/687fd7c7986d/src/share/classes/java/util/HashMap.java

```java
public class BadExamples {
  static class Point {
    int x, y;
    Point(int x, int y) {
      this.x = x;
      this.y = y;
    }

    public boolean equals(Object o) {
      if (o == this) {
          return true;
      }
      if (!(o instanceof Point)) {
          return false;
      }
      Point c = (Point) o;
      return c.x == x && c.y == y;
    }

    public int hashCode() {
      return x ^ y
    }
}
```

# Good Hash Code

For a compound object, we want to create a hash code by combining the individual hash codes of the object's constituent parts.

**hardcoded**

Change this every time you restart your app, otherwise someone can design an attack.

```
public int hashCode() {
    // random numbers from rand.org
    long[] r = {0x2058cc50L, 0xcb19137eL};
    long rr = 0xbea0107e5067d19dL;

    // convert (unsigned) hashcodes to long
    long h0 = x & ((1L<<32)-1);
    long h1 = y & ((1L<<32)-1);

    return (int)(((r[0]*h0 + r[1]*h1)*rr) >>> 32);
}
```

# Hash Codes for Compound Objects

Given an object made up of $r \geq 1$ parts whose hash codes are $x_0, x_1, \dots, x_{r-1}$

- choose mutually independent random $w$-bit integers $z_0, z_1, \dots, z_{r-1}$
- choose a random $2w$-bit **odd** integer $z$
- compute a hash code for our object:

$$\text{hash}(x) = \text{hash}(x_0, \dots, x_{r-1}) = \left( \left( z \cdot \sum_{i=0}^{r-1} z_i x_i \right) \bmod 2^{2w} \right) \operatorname{div} 2^w$$

Given different objects $x$ and $y$, assume $x_i \neq y_i$ for at least one index $i \in \{0, \dots, r-1\}$.

Then

$$Pr(\text{hash}(x_0, \dots, x_{r-1}) = \text{hash}(y_0, \dots, y_{r-1})) \leq \frac{3}{2^w}$$