CSC 321: Data Structures

Fall 2013

Hash tables

- HashSet & HashMap
- hash table, hash function
- collisions
  - linear probing, lazy deletion, primary clustering
  - quadratic probing, rehashing
  - chaining

HashSet & HashMap

recall: `TreeSet` & `TreeMap` use an underlying binary search tree (actually, a red-black tree) to store values

- as a result, add/put, contains/get, and remove are O(log N) operations
- iteration over the Set/Map can be done in O(N)

the other implementations of the `Set` & `Map` interfaces, `HashSet` & `HashMap`, use a "magic" data structure to provide O(1) operations*

***legal disclaimer:* performance can degrade to O(N) under bad/unlikely conditions
however, careful setup and maintenance can ensure O(1) in practice

the underlying data structure is known as a *Hash Table*

Hash tables

a hash table is a data structure that supports *constant time* insertion, deletion, and search *on average*

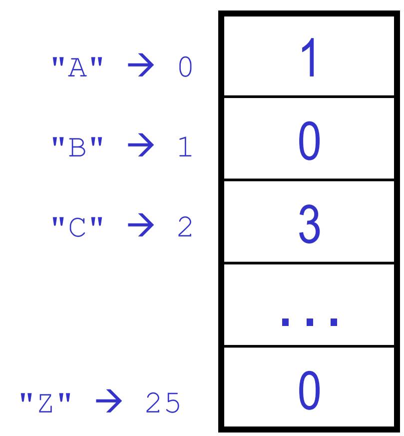
- degenerative performance is possible, but unlikely
- it may waste some storage
- iteration order is not defined (and may even change over time)

idea: data items are stored in a table, based on a key

- the key is mapped to an index in the table, where the data is stored/accessed

example: letter frequency

- want to count the number of occurrences of each letter in a file
- have an array of 26 counters, map each letter to an index
- to count a letter, map to its index and increment



Mapping examples

extension: word frequency

- must map entire words to indices, e.g.,

"A" → 0	"AA" → 26	"BA" → 52	. . .
"B" → 1	"AB" → 27	"BB" → 53	. . .
. . .	. . .	. . .	
"Z" → 25	"AZ" → 51	"BZ" → 77. . .	

- PROBLEM?

mapping each potential item to a unique index is generally not practical

of 1 letter words = 26
# of 2 letter words = 26^2 = 676
# of 3 letter words = 26^3 = 17,576
. . .

- even if you limit words to at most 8 characters, need a table of size 217,180,147,158
- for any given file, the table will be mostly empty!

Table size < data range

since the actual number of items stored is generally MUCH smaller than the number of potential values/keys:

- can have a smaller, more manageable table

  e.g., table size = 26
  possible mapping: map word based on first letter

  "A*" → 0   "B*" → 1   . . .   "Z*" → 25

  e.g., table size = 1000
  possible mapping: add ASCII values of letters, mod by 1000

  "AB" → 65 + 66 = 131

  "BANANA" → 66 + 65 + 78 + 65 + 78 + 65 = 417

  "BANANABANANABANANA" → 417 + 417 + 417 = 1251 % 1000 = 251

- POTENTIAL PROBLEMS?

Collisions

the mapping from a key to an index is called a *hash function*

- the hash function can be written independent of the table size
- if it maps to an index > table size, simply wrap-around (i.e., index % tableSize)

since `|range(hash function)| < |domain(hash function)|`,
can have multiple items map to the same index (i.e., a *collision*)

"ACT" → 67 + 65 + 84 = 216   "CAT" → 67 + 65 + 84 = 216

techniques exist for handling collisions, but they are costly (LATER)

it's best to avoid collisions as much as possible – HOW?

- want to be sure that the hash function distributes the key evenly

- e.g., "sum of ASCII codes" hash function
  OK   if table size is 1000
  BAD   if table size is 10,000
  most words are <= 8 letters, so max sum of ASCII codes = 1,016
  so most entries are mapped to first 1/10th of table

Better hash function

a good hash function should

- produce an even spread, regardless of table size
- take order of letters into account (to handle anagrams)

- the hash function used by `java.util.String` multiplies the ASCII code for each character by a power of 31

$$\texttt{hashCode()} = char_0 * 31^{(len-1)} + char_1 * 31^{(len-2)} + char_2 * 31^{(len-3)} + \ldots + char_{(len-1)}$$

where `len = this.length()`, $char_i$ `= this.charAt(i)`:

```
/**
 * Hash code for java.util.String class
 *   @return an int used as the hash index for this string
 */
private int hashCode() {
    int hashIndex = 0;

    for (int i = 0; i < this.length(); i++) {
        hashIndex = (hashIndex*31 + this.charAt(i));
    }
    return hashIndex;
}
```

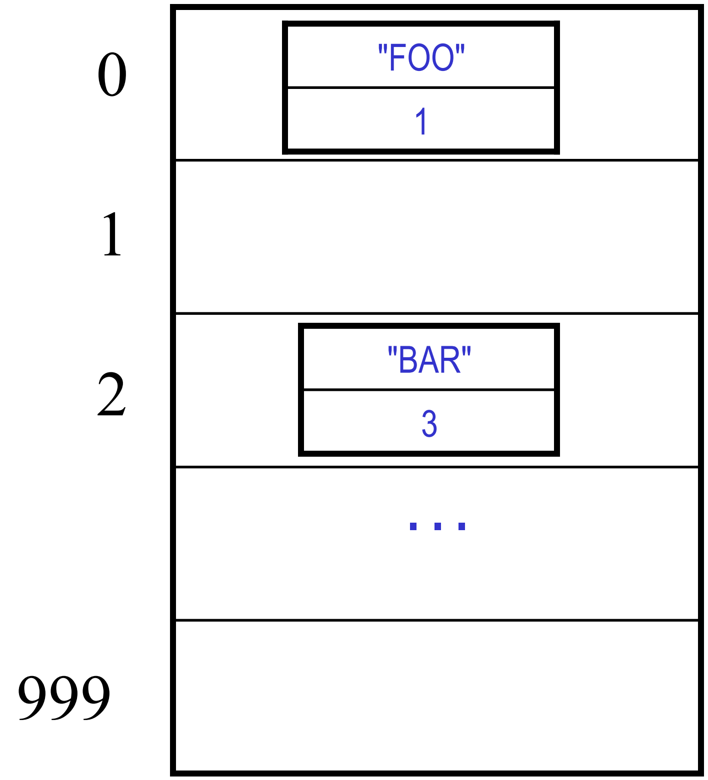
Word frequency example

returning to the word frequency problem

- pick a hash function
- pick a table size

- store word & associated count in the table

- as you read in words,
  map to an index using the hash function
  if an entry already exists, increment
  otherwise, create entry with count = 1



WHAT ABOUT COLLISIONS?

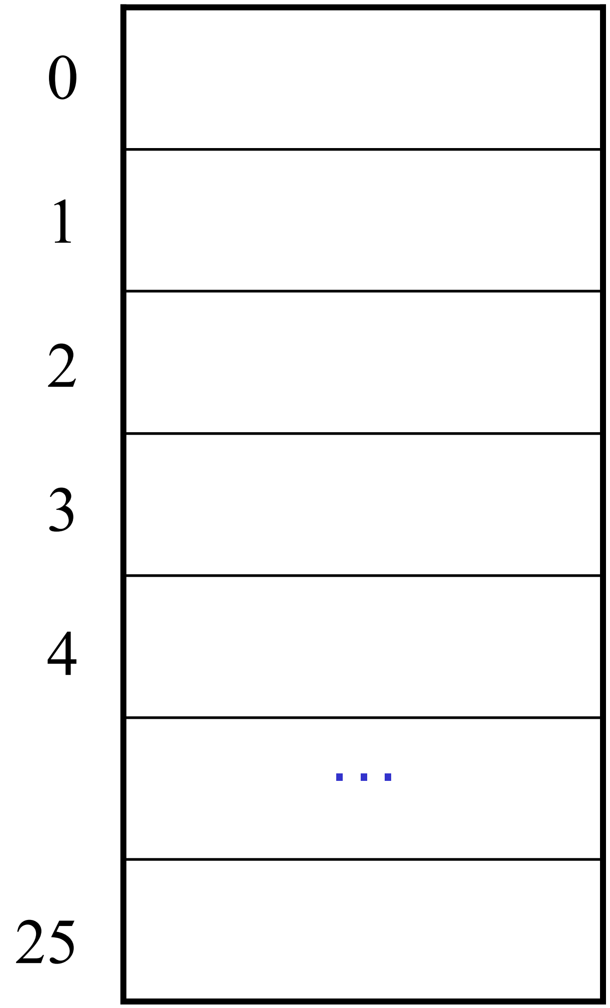
Linear probing

linear probing is a simple strategy for handling collisions

- if a collision occurs, try next index & keep looking until an empty one is found (wrap around to the beginning if necessary)

assume naïve "first letter" hash function

- insert "BOO"
- insert "COO"
- insert "BOW"
- insert "BAZ"
- insert "ZOO"
- insert "ZEBRA"



Linear probing (cont.)

with linear probing, will eventually find the item if stored, or an empty space to add it (if the table is not full)

what about deletions?

- delete "BIZ"

can the location be marked as empty?

can't delete an item since it holds a place for the linear probing

- search "COO"

0	"AND"
1	"BOO"
2	"BIZ"
3	"COO"
	. . .

Lazy deletion

when removing an entry

- mark the entry as being deleted (i.e., insert a "tombstone")
- subsequent searches must continue past tombstones (probe until desired item or an empty location is found)
- subsequent insertions can overwrite tombstones

ADD "BOO"

ADD "AND"

ADD "BIZ"

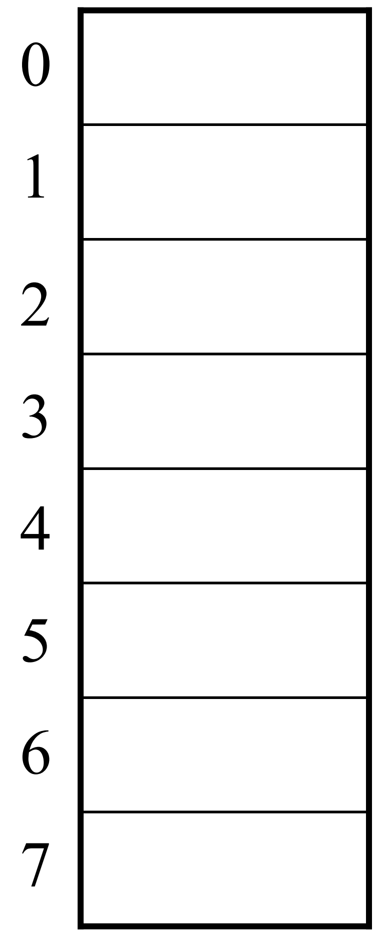
ADD "COO"

DELETE "BIZ"

SEARCH "COO"

ADD "COW"

SEARCH "COO"



Primary clustering

in practice, probes are not independent

- suppose table is half full

  maps to 4-7 require 1 check
  map to 3 requires 2 checks
  map to 2 requires 3 checks
  map to 1 requires 4 checks
  map to 0 requires 5 checks

  average = 18/8 = 2.25 checks

0	"AND"
1	"BOO"
2	"BIZ"
3	"COO"
4	
5	
6	
7	

using linear probing, clusters of occupied locations develop

- known as *primary clusters*

insertions into the clusters are expensive & increase the size of the cluster

Analysis of linear probing

the *load factor* λ is the fraction of the table that is full

empty table $\lambda = 0$ half full table $\lambda = 0.5$ full table $\lambda = 1$

THEOREM: assuming a reasonably large table, the average number of locations examined per insertion (taking clustering into account) is roughly $(1 + 1/(1-\lambda)^2)/2$

empty table	$(1 + 1/(1 - 0)^2)/2 = 1$
half full	$(1 + 1/(1 - .5)^2)/2 = 2.5$
3/4 full	$(1 + 1/(1 - .75)^2)/2 = 8.5$
9/10 full	$(1 + 1/(1 - .9)^2)/2 = 50.5$

as long as the hash function is fair and the table is not too full, then inserting, deleting, and searching are all O(1) operations

Rehashing

it is imperative to keep the load factor below 0.75

if the table becomes three-quarters full, then must resize

- create new table at least twice as big
- just copy over table entries to same locations???
- NO! when you resize, you have to rehash existing entries
  new table size → new hash function (+ different wraparound)

LET hashCode = word.length()

ADD "UP"

ADD "OUT"

ADD "YELLOW"

0	
1	
2	
3	

NOW
RESIZE
AND
REHASH

0	
1	
2	
3	
4	
5	
6	
7	

Chaining

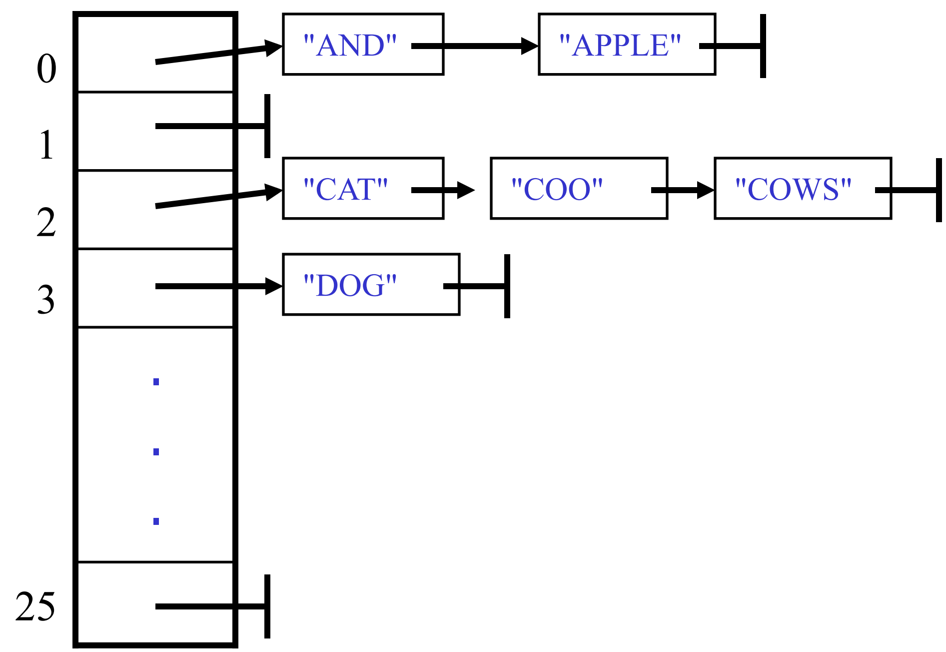
there are variations on linear probing that eliminate primary clustering

- e.g., quadratic probing increases index on each probe by square offset

Hash(key) → Hash(key) + 1 → Hash(key) + 4 → Hash(key) + 9 → Hash(key) + 16 → …

however, the most commonly used strategy for handling collisions is *chaining*

- each entry in the hash table is a bucket (list)
- when you add an entry, hash to correct index then add to bucket
- when you search for an entry, hash to correct index then search sequentially



Analysis of chaining

in practice, chaining is generally faster than probing

- cost of insertion is O(1) – simply map to index and add to list

- cost of search is proportional to number of items already mapped to same index
  e.g., using naïve "first letter" hash function, searching for "APPLE" might requires traversing a list of all words beginning with 'A'

  if hash function is fair, then will have roughly `λ/tableSize` items in each bucket
  → average cost of a successful search is roughly `λ/(2*tableSize)`

chaining is sensitive to the load factor, but not as much as probing – WHY?

Hashtable class

java.util

Class Hashtable<K,V>

Constructor Summary
`Hashtable()` Constructs a new, empty hashtable with a default initial capacity (11) and load factor (0.75).
`Hashtable(int initialCapacity)` Constructs a new, empty hashtable with the specified initial capacity and default load factor (0.75).
`Hashtable(int initialCapacity, float loadFactor)` Constructs a new, empty hashtable with the specified initial capacity and the specified load factor.
`Hashtable(Map<? extends K,? extends V> t)` Constructs a new hashtable with the same mappings as the given Map.

Method Summary	
`void`	`clear()` Clears this hashtable so that it contains no keys.
`Object`	`clone()` Creates a shallow copy of this hashtable.
`boolean`	`contains(Object value)` Tests if some key maps into the specified value in this hashtable.
`boolean`	`containsKey(Object key)` Tests if the specified object is a key in this hashtable.
`boolean`	`containsValue(Object value)` Returns true if this hashtable maps one or more keys to this value.
`Enumeration<V>`	`elements()` Returns an enumeration of the values in this hashtable.
`Set<Map.Entry<K,V>>`	`entrySet()` Returns a `Set` view of the mappings contained in this map.
`boolean`	`equals(Object o)` Compares the specified Object with this Map for equality, as per the definition in the Map interface.
`V`	`get(Object key)` Returns the value to which the specified key is mapped, or `null` if this map contains no mapping for the key.
`int`	`hashCode()` Returns the hash code value for this Map as per the definition in the Map interface.
`boolean`	`isEmpty()` Tests if this hashtable maps no keys to values.
`Enumeration<K>`	`keys()` Returns an enumeration of the keys in this hashtable.
`Set<K>`	`keySet()` Returns a `Set` view of the keys contained in this map.
`V`	`put(K key, V value)` Maps the specified `key` to the specified `value` in this hashtable.
`void`	`putAll(Map<? extends K,? extends V> t)` Copies all of the mappings from the specified map to this hashtable.
`protected void`	`rehash()` Increases the capacity of and internally reorganizes this hashtable, in order to accommodate and access its entries more efficiently.
`V`	`remove(Object key)` Removes the key (and its corresponding value) from this hashtable.
`int`	`size()` Returns the number of keys in this hashtable.
`String`	`toString()` Returns a string representation of this `Hashtable` object in the form of a set of entries, enclosed in braces and separated by the ASCII characters "`, `" (comma and space).

Java provides a basic hash table implementation

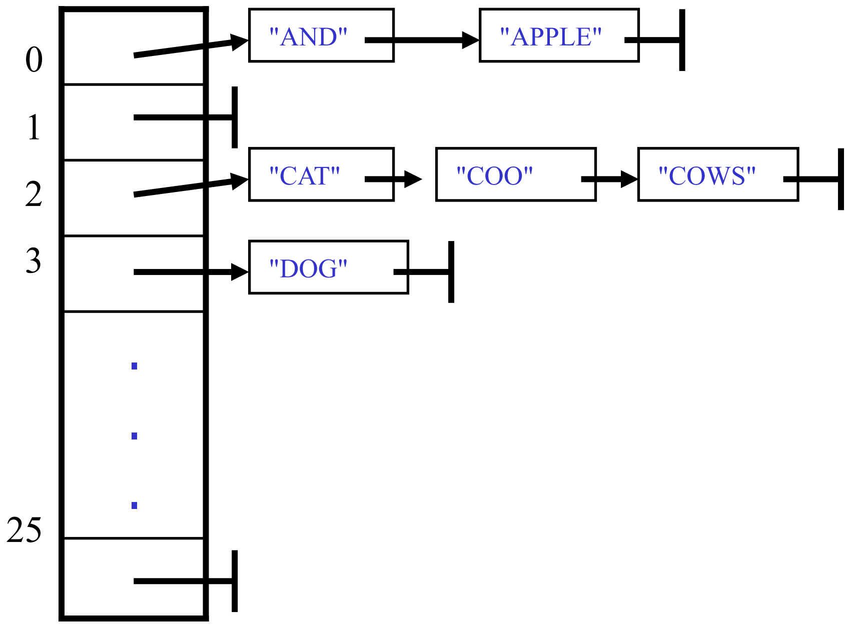
- utilizes chaining
- can specify the initial table size & threshold for load factor
- can even force a rehashing

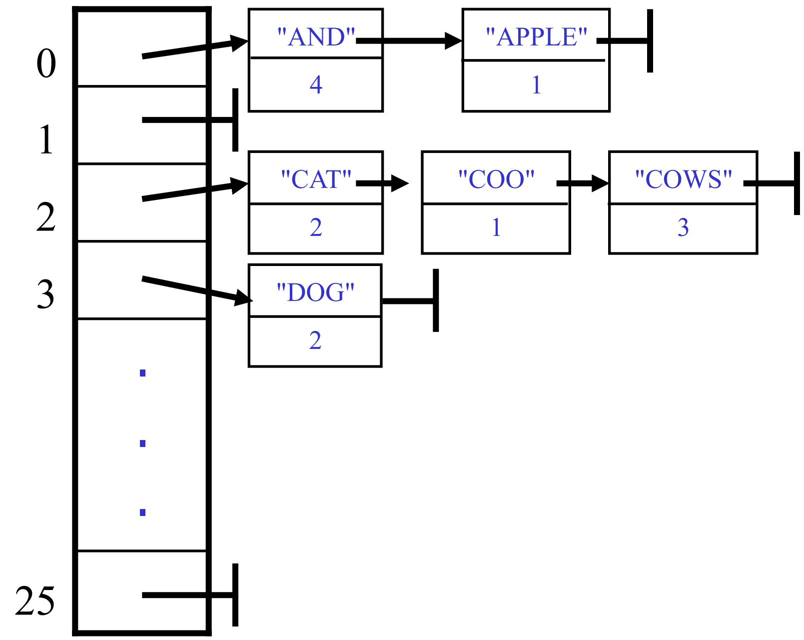
note commonly used, instead provides underlying structure for HashSet & HashMap

HashSet & HashMap

java.util.HashSet and java.util.HashMap use hash table w/ chaining

- e.g., HashSet<String> HashMap<String, Integer>





- defaults: table size = 16, max capacity before rehash = 75%
  can override these defaults in the HashSet/HashMap constructor call

note: iterating over a HashSet or HashMap is: O(num stored + table size) WHY?

hashCode function

```
import java.util.Calendar;
import java.util.GregorianCalendar;

public class Person {
    private String firstName, lastName;
    private Calendar birthday;

    public Person(String fname, String lname, int month, int day, int year) {
        this.firstName = fname;
        this.lastName = lname;
        this.birthday = new GregorianCalendar(year, month-1, day);
    }

    public String toString() {
        return this.firstName + " " + this.lastName + ": " +
               (this.birthday.get(Calendar.MONTH)+1) + "/" +
               this.birthday.get(Calendar.DAY_OF_MONTH) + "/" +
               this.birthday.get(Calendar.YEAR);
    }

    ////////////////////////////////////////////////////////////////////////

    public static void main(String[] args) {
        Person p1 = new Person("Chris", "Marlowe", 5, 25, 1992);
        System.out.println(p1);
        System.out.println(p1.hashCode());

        Person p2 = new Person("Alex", "Cooper", 2, 5, 1994);
        System.out.println(p2);
        System.out.println(p2.hashCode());

        Person p3 = new Person("Pat", "Phillips", 2, 5, 1994);
        System.out.println(p3);
        System.out.println(p3.hashCode());

    }
}
```

```
run:
Chris Marlowe: 5/25/1992
424201356
Alex Cooper: 2/5/1994
2053965899
Pat Phillips: 2/5/1994
205238968
BUILD SUCCESSFUL (total time: 0 seconds)
```

a default hash function is defined for every `Object`

- uses *native code* to access & return the address of the object

overriding hashCode v.1

```java
import java.util.Calendar;
import java.util.GregorianCalendar;

public class Person {
    private String firstName, lastName;
    private Calendar birthday;

    public Person(String fname, String lname, int month, int day, int year) {
        this.firstName = fname;
        this.lastName = lname;
        this.birthday = new GregorianCalendar(year, month-1, day);
    }

    public String toString() {
        return this.firstName + " " + this.lastName + ": " +
                 (this.birthday.get(Calendar.MONTH)+1) + "/" +
                 this.birthday.get(Calendar.DAY_OF_MONTH) + "/" +
                 this.birthday.get(Calendar.YEAR);
    }

    public int hashCode() {
        return Math.abs((int)this.birthday.getTimeInMillis());
    }

    ////////////////////////////////////////////////////////////////////////

    public static void main(String[] args) {
        Person p1 = new Person("Chris", "Marlowe", 5, 25, 1992);
        System.out.println(p1);
        System.out.println(p1.hashCode());

        Person p2 = new Person("Alex", "Cooper", 2, 5, 1994);
        System.out.println(p2);
        System.out.println(p2.hashCode());

        Person p3 = new Person("Pat", "Phillips", 2, 5, 1994);
        System.out.println(p3);
        System.out.println(p3.hashCode());

    }
}
```

```
run:
Chris Marlowe: 5/25/1992
1899603840
Alex Cooper: 2/5/1994
218788608
Pat Phillips: 2/5/1994
218788608
```

can override hashCode if more class-specific knowledge helps

1. must consistently map the same object to the same index
2. must map equal objects to the same index

overriding hashCode v.2

```
import java.util.Calendar;
import java.util.GregorianCalendar;

public class Person {
    private String firstName, lastName;
    private Calendar birthday;

    public Person(String fname, String lname, int month, int day, int year) {
        this.firstName = fname;
        this.lastName = lname;
        this.birthday = new GregorianCalendar(year, month-1, day);
    }

    public String toString() {
        return this.firstName + " " + this.lastName + ": " +
               (this.birthday.get(Calendar.MONTH)+1) + "/" +
               this.birthday.get(Calendar.DAY_OF_MONTH) + "/" +
               this.birthday.get(Calendar.YEAR);
    }

    public int hashCode() {
        return Math.abs((int)this.birthday.getTimeInMillis() +
                        (this.firstName+this.lastName).hashCode());
    }

    //////////////////////////////////////////////////////////////////////

    public static void main(String[] args) {
        Person p1 = new Person("Chris", "Marlowe", 5, 25, 1992);
        System.out.println(p1);
        System.out.println(p1.hashCode());

        Person p2 = new Person("Alex", "Cooper", 2, 5, 1994);
        System.out.println(p2);
        System.out.println(p2.hashCode());

        Person p3 = new Person("Pat", "Phillips", 2, 5, 1994);
        System.out.println(p3);
        System.out.println(p3.hashCode());

    }
}
```

to avoid birthday collisions, can also incorporate the names

- utilize the String hashCode method

```
run:
Chris Marlowe: 5/25/1992
413568008
Alex Cooper: 2/5/1994
520715368
Pat Phillips: 2/5/1994
9438334
```