# CSC 321: Data Structures

Fall 2017

Hash tables

- HashSet & HashMap
- hash table, hash function
- collisions
  - ➢ linear probing, lazy deletion, clustering, rehashing
  - ➢ chaining
- Java hashCode method
- HW6: finite state machines

## 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 deliver 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

| | |
|---|---|
| "A" → 0 | 1 |
| "B" → 1 | 0 |
| "C" → 2 | 3 |
| | . . . |
| "Z" → 25 | 0 |

3

## 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!

4

## 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?

5

## 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)|`,
Pigeonhole Principle ensures *collisions* are possible ($v_1$ & $v_2$ → same index)

"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 ≤ 10 letters, so max sum of ASCII codes = 1,270 |
| | so most entries are mapped to first 13% of table |

6

## Better hash function

a good hash function for words 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;
}
```

7

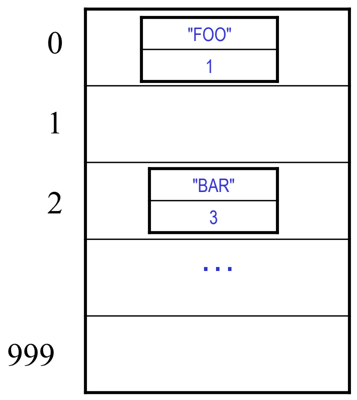
## Word frequency example

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?

8

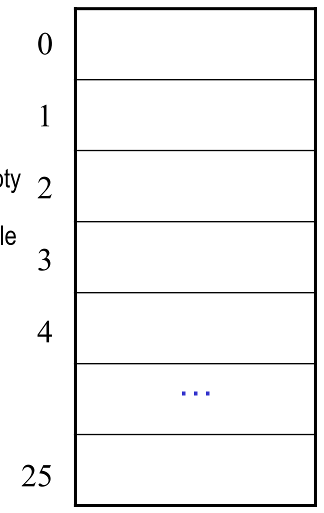
## Linear probing

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)

**example:** assume "first letter" hash function

- insert "BOO", "BAR", "COO", "BOW, …



linear probing requires "lazy deletion"

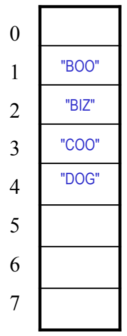
- when you delete an item, you can't just empty the location, since it would leave a hole
- subsequent searches would reach that whole and stop probing
- instead, leave a marker (a.k.a a tombstone) in that spot 0 can be overwritten but not skipped when probing

**example:** given above insertions

- delete "BAR", search for "COO"

9

## Clustering and load factor



in practice, probes are not independent

- as the table fills, clusters appear that degrade performance

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

average = 18/8 = 2.25 checks

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

empty table λ = 0 half full table λ = 0.5 full table λ = 1

THEOREM: assuming a reasonably large table, the average number of locations examined per insertion 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 | $(1 + 1/(1 - .9)^2)/2 = 50.5$ |

10

## Analysis of linear probing

the *load factor* $\lambda$ 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

as long as you keep the load factor low (e.g., < 0.75), inserting, deleting and searching a hash table are all O(1) operations

if the table becomes too full, then must resize

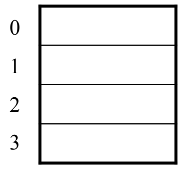
- create new table at least twice as big
- just copy over table entries to same locations???
- NO! when you resize, you have to rehash existing entries
  new table size $\rightarrow$ new hash function (+ different wraparound)

LET hashCode = word.length()

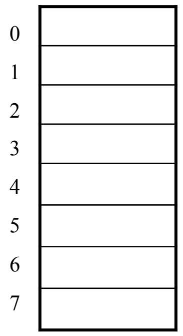
ADD "UP"

ADD "OUT"

ADD "YELLOW"



NOW RESIZE AND REHASH



## Chaining

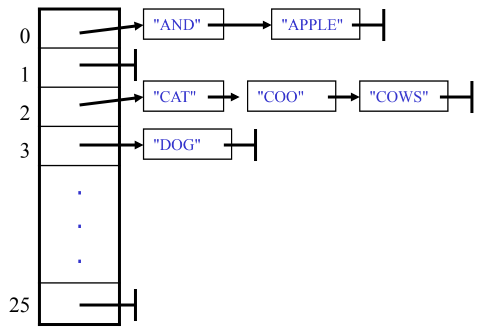
linear probing (or variants) were initially used when memory was expensive

- clustering, lazy deletion, and rehashing are all issues

modern languages like Java utilize a different approach

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



13

## 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'

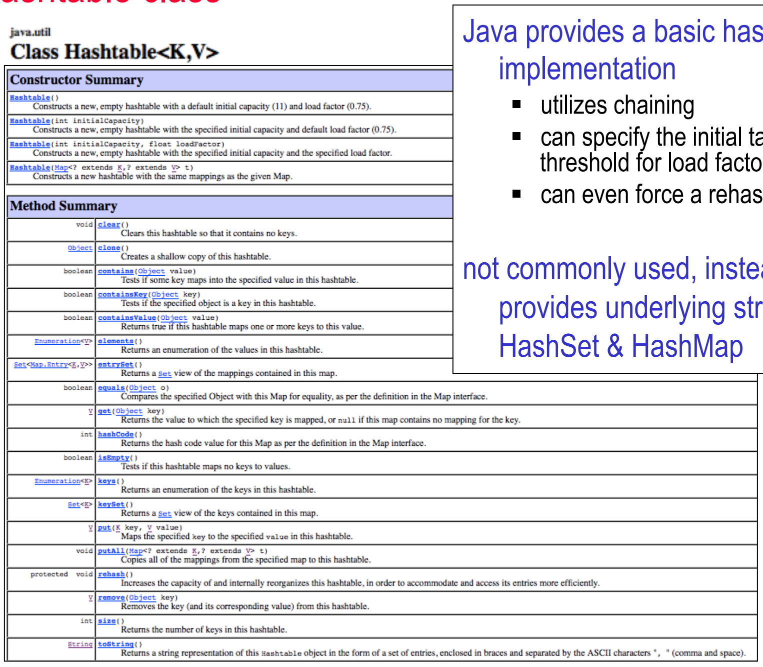
  if hash function is fair, then will have roughly `λ/tableSize` items in each bucket
  → average cost of a successful search is roughly `λ/(2*tableSize)`

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

chaining uses more memory – WHY?

14

## 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. |

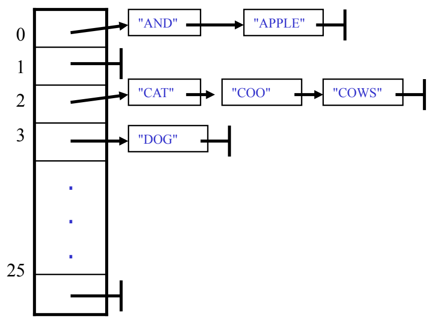
| **Method Summary** | |
|---|---|
| `void` | `clear()` Clears this hashtable so that it contains no keys. |
| `Object` | `clone()` Creates a shallow copy of this hashtable. |
| `boolean` | `contains(Object value)` Tests if some key maps into the specified value in this hashtable. |
| `boolean` | `containsKey(Object 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). |

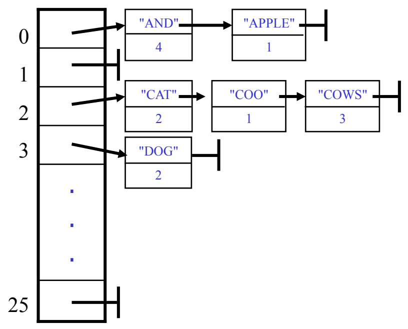
Java provides a basic hash table implementation
- utilizes chaining
- can specify the initial table size & threshold for load factor
- can even force a rehashing

not 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?

## Word frequencies (again)

### using HashMap instead of TreeMap

- containsKey, get & put operations are all O(1)*
- however, iterating over the keySet (and their values) does not guarantee any order

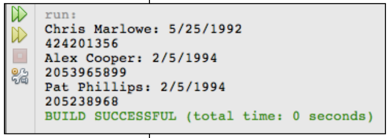
- if you really care about speed → use HashSet/HashMap
- if the data/keys are comparable & order matters → use TreeSet/TreeMap

```java
import java.util.Map;
import java.util.HashMap;
import java.util.Scanner;
import java.io.File;

public class WordFreq {
    private Map<String, Integer> words;

    public WordFreq() {
        words = new HashMap<String, Integer>();
    }

    public WordFreq(String filename) {
        this();
        try {
            Scanner infile = new Scanner(new File(filename));
            while (infile.hasNext()) {
                String nextWord = infile.next();
                this.add(nextWord);
            }
        }
        catch (java.io.FileNotFoundException e) {
            System.out.println("FILE NOT FOUND");
        }
    }

    public void add(String newWord) {
        String cleanWord = newWord.toLowerCase();
        if (words.containsKey(cleanWord)) {
            words.put(cleanWord, words.get(cleanWord)+1);
        }
        else {
            words.put(cleanWord, 1);
        }
    }

    public void showAll() {
        for (String str : words.keySet()) {
            System.out.println(str + ": " + words.get(str));
        }
    }
}
```

17

## hashCode function

```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 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());

    }
}
```



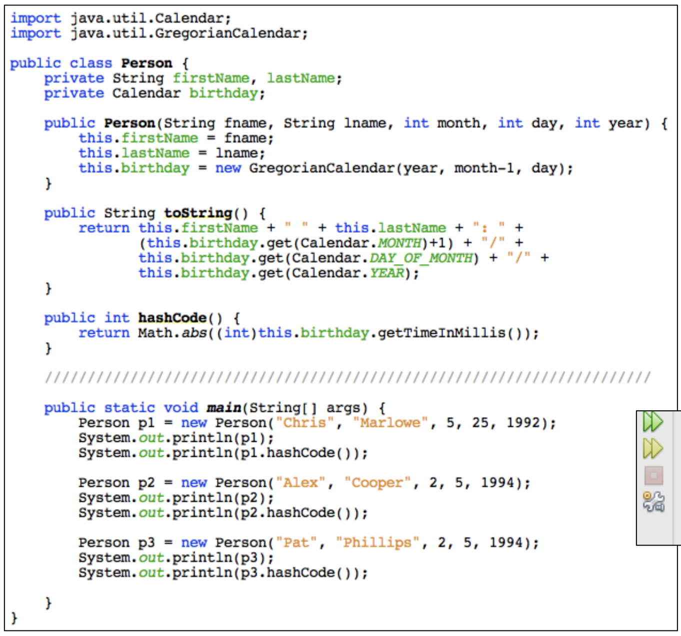
```
run:
Chris Marlowe: 5/25/1992
424201356
Alex Cooper: 2/5/1994
2053965899
Pat Phillips: 2/5/1994
205238968
BUILD SUCCESSFUL (total time: 0 seconds)
```

### a default hash function is defined for every `Object`

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

18

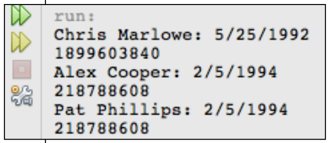
## overriding hashCode v.1



```
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());

    }
}
```

can override hashCode if more class-specific knowledge helps

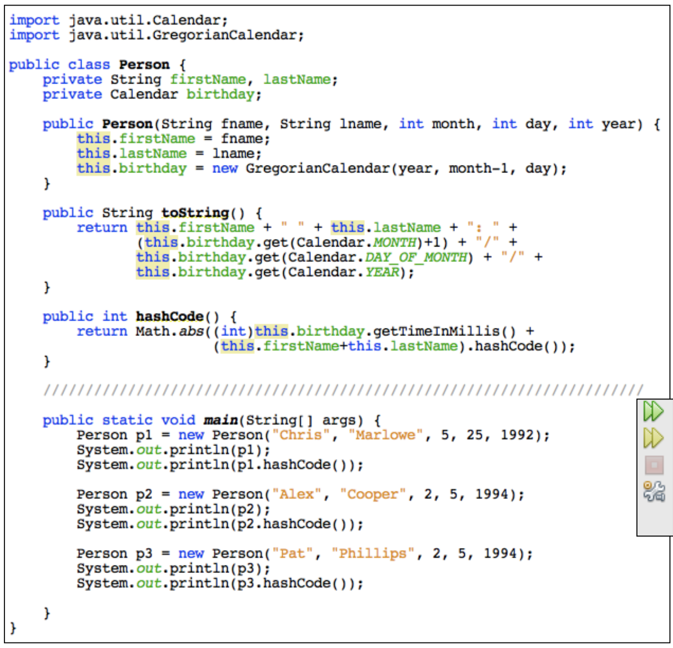
1. must consistently map the same object to the same index
2. must map equal objects to the same index



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

19

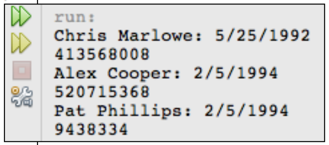
## overriding hashCode v.2



```
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



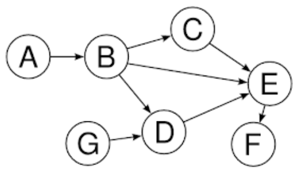
```
run:
Chris Marlowe: 5/25/1992
413568008
Alex Cooper: 2/5/1994
520715368
Pat Phillips: 2/5/1994
9438334
```

20

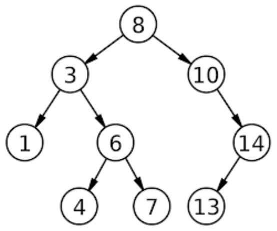
## Graphs (sneak peek)

trees are special instances of the more general data structure: graphs

- informally, a graph is a collection of nodes/data elements with connections


$$\begin{tikzcd} & & C \arrow[dr] & \\ A \arrow[r] & B \arrow[ur] \arrow[rr] \arrow[d] & & E \arrow[d] \\ G \arrow[r] & D \arrow[urr] & & F \end{tikzcd}$$

a tree is a graph in which one node has no edges coming into it (the root) and no cycles


$$\begin{tikzcd} & & 8 \arrow[dl] \arrow[dr] & \\ & 3 \arrow[dl] \arrow[d] & & 10 \arrow[d] \\ 1 & 6 \arrow[dl] \arrow[d] & & 14 \arrow[dl] \\ 4 & 7 & 13 & \end{tikzcd}$$

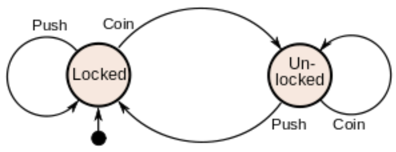
21

## Finite State Machines (FSMs)

many useful problems can be defined using simple graphs

- a *Finite State Machine* (a.k.a. *Finite Automaton*) defines a finite set of states (i.e., nodes) along with transitions between those states (i.e., edges)

e.g., the logic controlling a coin-operated turnstile



can be in one of two states: locked or unlocked

| | |
|---|---|
| ▪ if locked, | pushing → it does not allow passage & stays locked |
| | inserting coin → unlocks it |
| ▪ if unlocked, | pushing → allows passage & then relocks |
| | inserting coin → keeps it unlocked |

22

## Other examples

Claude Shannon used a FSM to show constraints on Morse code

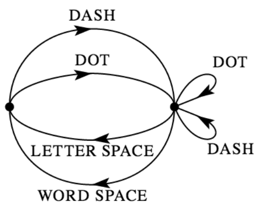
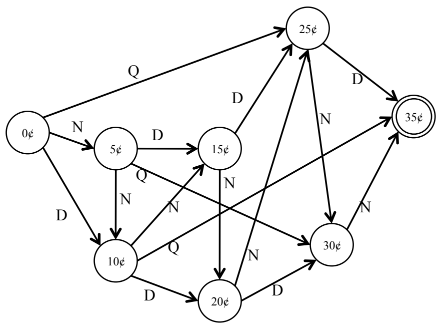


Fig. 2 — Graphical representation of the constraints on telegraph symbols.



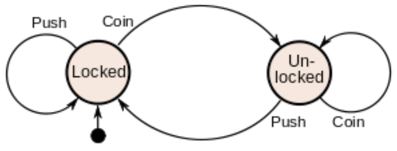
can use a FSM to specify the behavior of a vending machine

adding a coin (Q, D, N) changes the state

23

## HW6: Simulate a FSM

will read in a file that specifies the edges in a FSM



```
locked push locked
locked coin unlocked
unlocked push locked
unlocked coin unlocked
```

then be able to trace paths within that FSM:

```
Enter FSM file: turnstile.txt

Enter a start state (* to end): locked
Enter a step sequence (separated with whitespace): coin push
End state: locked
```

24

## FSM class

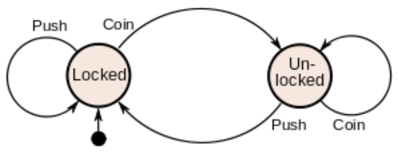
your FSM class will utilize a 2-level hash table

```
private HashMap<String, HashMap<String, String>> table;
```

- the key to the table is the start node of an edge
- the value is another map, which maps edge labels to the end states

e.g., table.get("locked") →a map containing edges coming out of "locked"

table.get("locked").get("coin") →"unlocked"


Push
Coin
Locked
Un-
locked
Push
Coin