**CMSC 132 Lecture Notes**

This file contains lecture notes for CMSC 132. This material need not be presented in the order given here.

**Table of Contents**

[1) Course Introduction 3](#_30j0zll)

[2) Introduction to Eclipse (Review) 3](#_1fob9te)

[3) Java Interfaces (Review) 3](#_3znysh7)

[4) Data and procedural abstraction 4](#_2et92p0)

[5) Java Collection Framework (Intro) 4](#_tyjcwt)

[6) Java wrapper classes: Integer, Float, Double, etc. (Review) 4](#_3dy6vkm)

[7) Enumerated types 4](#_1t3h5sf)

[8) Iterators 4](#_4d34og8)

[9) Inheritance 4](#_2s8eyo1)

[10) Object class (Review) 5](#_17dp8vu)

[11) “Type” vs. “class” 5](#_3rdcrjn)

[12) Type Casting 6](#_26in1rg)

[13) Multiple Inheritance 6](#_lnxbz9)

[14) Inheritance vs. Composition 6](#_35nkun2)

[15) “final” 6](#_1ksv4uv)

[16) Equals method (Review) 6](#_44sinio)

[17) Abstract classes 7](#_2jxsxqh)

[18) Comparable interface (Review) 7](#_z337ya)

[19) Comparator Interface 7](#_3j2qqm3)

[20) Programming errors (Review) 7](#_1y810tw)

[21) Testing (Review) 7](#_4i7ojhp)

[22) Code Coverage 8](#_2xcytpi)

[23) Exception Handling (Review) 8](#_1ci93xb)

[24) Inner classes 9](#_3whwml4)

[25) Nested classes 9](#_2bn6wsx)

[26) Deep vs. Shallow copy (Review) 9](#_qsh70q)

[27) Clone 9](#_3as4poj)

[28) Initialization blocks 10](#_1pxezwc)

[29) Garbage collector (Review) 10](#_49x2ik5)

[30) Software development process and systematic method design (Review) 10](#_2p2csry)

[31) Asymptotic Complexity 11](#_147n2zr)

[32) Writing classes with parameterized types 12](#_3o7alnk)

[33) Overview of Linear ADTs 12](#_23ckvvd)

[34) Linear Data Structures 13](#_ihv636)

[35) Java classes for Lists 14](#_32hioqz)

[36) Abstract Data Types related to Sets 14](#_1hmsyys)

[37) Hash Tables 14](#_41mghml)

[38) Java classes for Sets 16](#_2grqrue)

[39) Java classes for Maps 16](#_vx1227)

[40) Recursion (Review) 17](#_3fwokq0)

[41) Trees 17](#_1v1yuxt)

[42) Singleton Design Pattern 18](#_4f1mdlm)

[43) Heaps 18](#_2u6wntf)

[44) Graphs 19](#_19c6y18)

[45) Sorting Algorithms 21](#_3tbugp1)

[46) Concurrent programming 22](#_28h4qwu)

[47) Java I/O Streams 22](#_nmf14n)

[48) Introduction to Networking 23](#_37m2jsg)

[49) Algorithm Strategies 23](#_1mrcu09)

# Course Introduction

* 1. Instructor/TA info
  2. General expectations
  3. Schedule
  4. Purpose of course
  5. Class webpage
  6. Course tools
  7. Pre-requisite expectations (students should already know the bulk of the CMSC 131 content)
     1. Loops, conditional statements
     2. Arrays
     3. Java interfaces
     4. Exception handling
     5. Recursion
  8. Tips for Success
     1. Don’t miss class
     2. Ask questions
     3. Come to office hours
     4. Start projects early
     5. Review items missed on quizzes/exams
     6. Do all handouts from labs and practice questions provided
     7. Study constantly (not just the night before the exam)
  9. Overview of course content
     1. Programming
        1. Advanced Java features
        2. Testing/debugging
        3. Inheritance
        4. Concurrent programming
     2. Abstract Data types
     3. Data structures
     4. Algorithms and algorithm strategies
     5. Asymptotic complexity

# Introduction to Eclipse (Review)

* 1. Idea of Integrated Development Environment (IDE)
  2. Installation of Java (including the JRE and the JDK)
  3. Installation of Eclipse (including CVS plugin and UMCP plugin)
  4. Eclipse perspectives
  5. Connecting to student’s CVS repository
  6. Checking out and submitting projects (demonstration)

# Java Interfaces (Review)

* 1. Polymorphism

# Data and procedural abstraction

* 1. Encapsulation
  2. API

# Java Collection Framework (Intro)

* 1. Collection Interface (not class)
  2. (Review) ArrayList
  3. Generics
     1. Preferred because mistakes lead to syntax errors, not elusive runtime errors
  4. Collection class (not interface)
  5. (Review) for-each loops
  6. Iterator interface
     1. Supports remove (advantage over for-each loop)
     2. Note: for-each loop relies on iterators

# Java wrapper classes: Integer, Float, Double, etc. (Review)

* + 1. Auto-boxing/unboxing

# Enumerated types

* 1. Syntax
  2. compareTo()
  3. values()
  4. == operator to compare values

# Iterators

* 1. Iterator interface
  2. Iterator is always “attached” to a particular collection
  3. One-time use (“disposable”)
  4. For-each loops rely on underlying iterators (but remove is not supported)
  5. Iterable interface (mention it now, implement Iterable classes later in the course)

# Inheritance

* 1. Example with inheritance diagram. Mention details:
     1. Inheritance
     2. Vocabulary:
        1. Superclass/base class
        2. Subclass/derived class
        3. “inherit”
        4. “IS-A”
     3. Transitivity
     4. Polymorphism. Given a polymorphic variable of a certain type:
        1. What kind of objects can be assigned? (Mention “subtype”)
        2. What methods can be invoked?
     5. Dynamic Dispatch
  2. Inheritance vs. Interfaces
     1. Both useful for polymorphic variables
     2. Both result in Subtypes
     3. Extensions provide inheritance (interfaces do not)
     4. Classes can extend only ONE class, but can implement many interfaces
  3. “this” and “super”
     1. Example with memory diagram
     2. Subclasses can also override methods (details lataer)
  4. Constructors
     1. Using “super” to invoke specific superclass constructors
  5. Method overriding
     1. Early/static binding vs. late/dynamic binding
     2. Compare to overloading with an example like this:

public class Base {

public void m (int x) { … }

}

public class Derived extends Base {

public void m (int x) { … }

public int m (int x) { … } // won’t compile! Why?

public void m (double d) { … }

}

Base b = new Base( );

Base d = new Derived( );

Derived e = new Derived( );

b.m (5);

d.m (6);

d.m (7.0);

e.m (8.0);

* 1. Protected visibility
  2. Shadowing variables (usually not desirable)

# Object class (Review)

* 1. Everything extends Object!
  2. Look at API documentation

# “Type” vs. “class”

* 1. An object is an instance of ONE class
  2. Objects may be several “types” (interfaces, superclasses)
  3. Instanceof operator (for checking type)

# Type Casting

* 1. Upcast (to superclass type)
     1. Always OK
     2. Done implicitly by Java when needed
     3. Just ignore extra features of subclass
  2. Downcast (to subtype)
     1. May throw IllegalCastException
     2. Must be done explicitly with casting operator
  3. Complex example like the following. Assume that Student and Faculty are extensions of Person.

Person p = new Person();

Student s = new Student();

Person tricky = new Student();

// For each of the following, decide whether it is OK,

// won’t compile, or compiles but throws exception

Student y = p;

Student y = (Student)p;

Student y = tricky;

Student y = (Student)tricky;

(Faculty)s;

(Faculty)tricky;

* 1. Safe downcasting using instanceof

# Multiple Inheritance

* 1. Definition
  2. Why doesn’t Java allow it? (“Diamond of Death” example.)

# Inheritance vs. Composition

* 1. Inheritance allows polymorphism
  2. Inheritance is a rigid design decision
  3. If composition is sufficient, favor it

# “final”

* 1. Variable (can’t be re-assigned)
  2. Method (can’t be overridden)
  3. Class (can’t be extended)

# Equals method (Review)

* 1. Introduce optimization comparing current object with parameter. Example:

public boolean equals(Object obj) {

if (this == obj) {

return true;

}

if ( ! (obj instanceof Cat)) {

return false;

}

Cat c = (Cat)obj;

return… // compare fields of c vs. this

}

# Abstract classes

* 1. Syntax for abstract class and abstract methods
  2. Do an example. Perhaps
     1. Shape is a class with state and some methods, but with an abstract method (calculateArea) that is implemented by concrete subclasses (Triangle, Circle, Square).
  3. Abstract class cannot be instantiated
  4. Can be used as a type (yields polymorphic variable)
  5. Compare with interfaces:
     1. Inheritance
     2. Classes can implement multiple interfaces (but can only extend one class)
     3. Favor interfaces

# Comparable interface (Review)

* 1. Needed for Collections.sort, Arrays.sort, Collection classes we’ll learn later (SortedMap, SortedSet)
  2. Defines “natural order”
  3. If your class implements Comparable then it should override equals (so that whenever compareTo returns 0, those objects are considered equal).

# Comparator Interface

* 1. Do a good example where instances of a class are sorted many ways

# Programming errors (Review)

* 1. Syntax
  2. Semantic
  3. Logical

# Testing (Review)

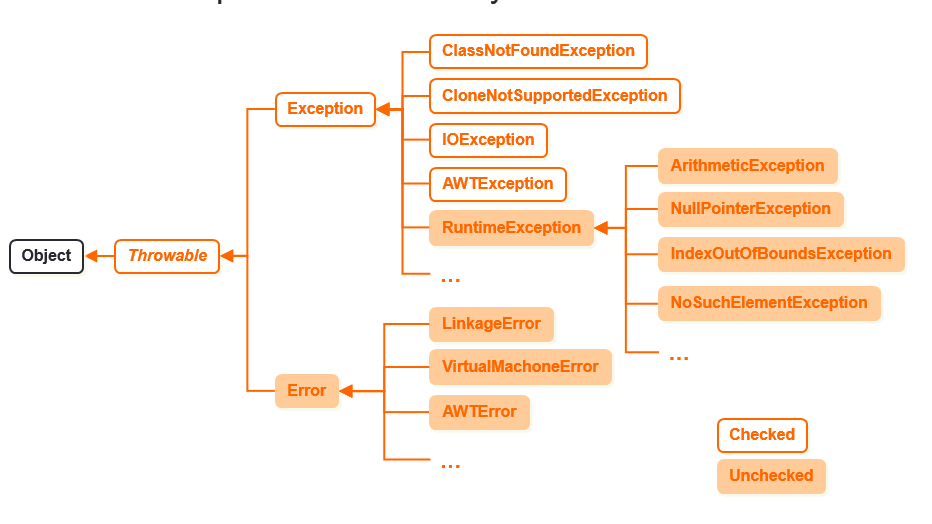
* 1. Integrated vs. Unit
  2. JUnit
     1. Syntax
     2. Demonstration with Eclipse

# Code Coverage

* 1. Statement coverage
  2. Conditional coverage

# Exception Handling (Review)

* 1. State examples of context and when to “throw” exceptions
  2. Do not use exception handling for “normal” program flow.
  3. Examples of exception classes
  4. What happens when exception is thrown?
     1. Draw Call stack and discuss
        1. Java looks for handler in current frame
        2. If found, exception is handled
        3. If not found, this frame is popped off the stack, now consider previous frame
  5. How do we write “handler”
     1. Try/catch
     2. Multiple catch blocks
     3. Finally block
  6. Checked vs. Unchecked exceptions
     1. Show inheritance diagram with picture of many exception classes:



* + 1. Unchecked
       1. Typically severe errors (such as “programming errors”) that you cannot recover from
       2. These should not be happening in a finished project.
    2. Checked
       1. More common problems that program should handle.
       2. Cannot be ignored.
          1. Handle locally OR
          2. Declare that your method “throws” the exception. (Allows it to pass on to the caller.)
  1. Catches should be for situation. Don’t do this: catch(Thowable e)

# Inner classes

* 1. Syntax for class declarations
  2. Each instance of the inner class is “tied to” a particular instance of the outer class.
  3. An outer class instance can have 0, 1, or multiple inner class instances “tied to” it.
  4. Inner class has access to all features of outer class (even private ones!)
  5. Syntax for instantiating inner class (object must be connected to existing outer class instance):

OuterClass outer = new OuterClass();

OuterClass.InnerClass inner = outer.new InnerClass();

* 1. Typical examples of inner class:
     1. Talk about Event handler attached to a GUI component
     2. Demonstrate with examples of a List implanting the Iterable interface.
        1. Iterator() method will return an instance of an inner class that implements Iterator.

# Nested classes

* 1. Syntax (like inner class, but “static”)
  2. No relationship between instances of inner and outer classes
  3. Inner class can access static members of outer class (even private ones)
  4. Typical use: We’ll use them like this:

public class LinkedList {

private static class node() {

}

}

# Deep vs. Shallow copy (Review)

* 1. Draw memory diagram and explain
     1. Shallow copy is probably fine when members are immutable
     2. Deep copy may be required when members are mutable

# Clone

* 1. Recall copy constructor (not always available)
  2. Clone method can be overridden
     1. Fast shallow copy
     2. Tricky contract
     3. Cloneable interface (“Marker interface”) used to indicate class can be safely cloned.
     4. Superclass must implement Clonable
     5. Members requiring deep copy must implement Clonable
     6. To override clone:
        1. Invoke super.clone()
        2. Does “correct” copy of base-class portion
        3. Does shallow copy of extended portion
        4. If deeper copy is needed (because members are mutable), call clone on members.
        5. Example:

public Student clone() throws CloneNotSupportedException {

Student theCopy = (Student)super.clone();

theCopy.address = (Address)address.clone();

return theCopy;

}

# Initialization blocks

* 1. Static initialization block
     1. Runs ONCE when class is loaded
  2. Non-static initialization block
     1. Runs each time an instance is created
     2. Runs before constructor

# Garbage collector (Review)

* 1. Explain with memory diagram
  2. Garbage collector typically runs when heap space is running out
  3. You can invoke it explicitly, but hardly ever need to

# Software development process and systematic method design (Review)

* 1. General software development process (iterative approach)
     1. Identify the components that make up the system.
     2. Figure out and analyze use cases that the system should be able to handle
     3. Determine the program modules (classes) and their APIs (interfaces). How will the components interact? Which components rely on each other?
     4. Build a basic prototype that features the essential/core components. (Just enough to be able to run and demonstrate the primary functionality)
     5. Iteratively refine this prototype so that it comes closer and closer to the desired product.
  2. Systematic Method (function) design
     1. Define data types/structures
        1. Figure out what kind of data must be represented
        2. Identify which primitives/classes/structures can be used for this kind of data
     2. Define Signature and contract
        1. Types of parameters for input
        2. Return type (or side effects) for output
        3. Document the “purpose” of the method. What does it do?
        4. Create a “stub” for the method (to be filled in)
     3. Come up with cases (examples)
        1. Determine the spectrum of input cases that may occur.
        2. Write down what the output should be for various inputs, E.g. foo(7) → orange
        3. Think about edge/corner cases
        4. Optionally, implement actual test cases (JUnit) now.
     4. Create an outline/template for the method
        1. Come up with an “inventory” of what is available at the start of the method. In Java, this may include parameters, (public) static data members, and the current object in an instance method.
        2. Write down (perhaps in pseudocode) the basic algorithm/steps the method should follow.
     5. Implementation
        1. Rely heavily on previous two steps
     6. Testing
        1. Based on cases from step iii

# Asymptotic Complexity

* 1. Intuition
     1. For a particular algorithm, consider a graph of time (T) as a function of n (size of dataset).
     2. Note that the graph will not be decreasing.
     3. Not every algorithm is “linear”
        1. Examples that run in linear time
        2. Examples that run in quadratic time
        3. Examples that run in log time
     4. Graphs tend to asymptotically approach some fixed (known) function. (A line, a parabola, an exponential curve, log curve, etc.)
     5. Considering graphs for two algorithms that solve the same problem, can we say which is “faster”?
        1. If the shapes are different (e.g. line vs. parabola) then we can say something specific: “For sufficiently large values of n, the linear algorithm will be faster than the quadratic algorithm”.
  2. Big-O notation
     1. Idea: f is O(g) means f is “not significantly slower” than g.
     2. Definition:

f is O(g) means:

there exist constants M, N such that:

for all n>N, M\*g(n) >= f(n)

* + 1. Easy “proofs” (from the definition):
       1. Example: Prove that 2n^2 + 15n + 20 is O(n^2).
          1. Find M that we can multiply n^2 by so that it dominates the other. (3 looks good.)
          2. Find N such that whenever n>N we have

2n^2 + 15n + 20 <= (3) n^2

* + - * 1. N = 1? N = 100? N = 1000 works.
        2. Carefully state what you have shown:

2n^2 + 15n + 20 is O(n^2) because

For all n > 1000, we have

2n^2 + 3n + 20 <= (3) n^2

* + 1. Using the definition of Big-O, functions can be placed into categories. Mention these explicitly:

O(1), O(log n), O(n), O(n^2), O(n^3)… O(n^1000000)… O(2^n), O(3^n)….. O(n!), O(n^n)….

* + 1. Familiarize students with notation by saying:
       1. All linear functions are O(n), O(n^2), O(n^1000)
       2. All logarithmic functions are O(log n), O(n), etc.
  1. Looking at code and determining runtimes
     1. Simple examples with one loop (Some O(n), some O(1), some O(log n)…).
     2. Examples with nested loops (Some O(n^2), some O(n), some O(1)…)
     3. Example with exponential time
     4. Example with log time
     5. Example with constant time
     6. Looking for “bottleneck”. (Worst section dictates performance of a whole sequence of sections.)
  2. Brief discussion of “case” analysis
     1. Best case (not usually very informative)
     2. Worst case (the function that measures the performance of the worst input case for each value of n – this can be useful to know.)
     3. Average case (the most useful analysis – the function that measures the average performance for each value of n.)

# Writing classes with parameterized types

* 1. Pair example
  2. MyList example
  3. Using type variable for variable declarations, return types, parameters
  4. Using “extends” in class declaration, e.g.: class MyList<T extends X>
  5. Example: class MyList<T extends <Comparable<T>>
  6. Syntax with arrays: T[]a = (T[]) Object[4];
  7. Is String[] a subtype of Object[]? YES. Show example where this causes problems.
  8. Is ArrayList<String> a subtype of ArrayList<Object>? NO. Error from compiler. (Good!)
  9. Using “? extends” for variables
     1. ArrayList<Athlete> a; Does NOT allow for subclasses of Athlete.
     2. ArrayList<? extends Athlete> a; DOES allow for subclasses of Athlete.
     3. Note: You cannot add elements to the collection.
     4. Usually used as a type for a parameter.

# Overview of Linear ADTs

* 1. List
     1. Mention vocabulary: Head, tail
     2. Add
     3. Insert
     4. Replace
     5. Remove
     6. Get by index
     7. Search (return index number, if found)
  2. Stack
     1. Push
     2. pop
  3. Queue
     1. enqueue
     2. dequeue

# Linear Data Structures

* 1. Arrays (contiguous allocation)
  2. Linked List (linked allocation)
     1. Nodes
        1. Reference to data
        2. Reference to next node
     2. Draw memory diagram
     3. Mention “head” reference
     4. Mention possibility of optional “tail” reference
     5. Mention last Node having null reference (typically)
     6. What would an empty linked list look like?
     7. Talk about how we traverse using a “cursor”
     8. Do a lengthy example of a typical linked list implementation (using a parameterized type)
  3. Comparing Array with LinkedLists
     1. Add
        1. Array
           1. O(n) if just one operation, or if adding to the front, or inserting in the middle.
           2. Add to the end can be O(1) in aggregate, by maintaining extra empty boxes, and multiplying the array by a fixed constant multiplier any time you run out of room.
        2. Linked List
           1. Add/insert “at cursor” – O(1)
           2. Add at head – O(1)
           3. Add at tail (with reference) – O(1)
           4. Add/insert in the middle without a cursor already there – O(n)
     2. Remove
        1. O(n) for array
        2. Linked List
           1. Remove at head – O(1)
           2. Remove from cursor – O(1) as long as it’s not the tail node. (How? Copy data from next node to current Node, copy next reference from next node to current node.)
           3. Remove from tail -- O(n)
           4. Remove with cursor to previous node – O(1)
     3. Replace
        1. O(1) for array
        2. Linked List
           1. Head/ at cursor (including tail reference) – O(1)
           2. Elsewhere – O(n)
     4. Random access (get by index)
        1. O(1) for array
        2. O(n) for Linked List
     5. Search is O(n) for either one (unless Array is sorted, then O(log n))
  4. Doubly Linked List
     1. Like Linked List but with another reference for previous node
     2. Draw memory diagram
     3. Show coding example (don’t spend much time on this)
     4. Performance compared to linked list?
        1. Delete last node is now O(1). Not much to gain.
        2. Disadvantage – extra memory taken up by prev reference
  5. Best data structure to implement a Stack? Linked List with “top” at head.
  6. Best data structure for Queue? Linked List – add to tail, remove from head.

# Java classes for Lists

* 1. List interface
  2. ArrayList
  3. LinkedList

# Abstract Data Types related to Sets

* 1. Set
     1. Add
     2. Contains? (Search)
     3. Remove
  2. Map
     1. What is it?
     2. put (key, value pair)
     3. Get value from key
     4. Remove
     5. containsKey
     6. containsValue
     7. getKeySet
     8. getValueSet

# Hash Tables

* 1. If a Set is implemented with an Array or Linked List, how fast is search?
     1. Linked List – O(n)
     2. Array – Could be as good as O(log n) if sorted
     3. Hash Table…. In many cases O(1).
     4. Downside of Hash tables? Cannot maintain a sorted/ordered collection.
  2. Example: Storing Univ. of MD students in a table using ID numbers as keys (ID numbers are sequences of 9 digits)
  3. Fastest way possible? Make an array of references to students of size 1,000,000,000 so that each ID number corresponds to a unique index number. Obviously very wasteful! (Most boxes unused)
  4. Another idea? Make an array of size 100,000 and then use IDNumber % 5 to pick a box. Works pretty well, but there will be collisions! How to handle collisions?
     1. Linear probing -- If box is already taken, try the next box, then the next one, until an empty one is found. Note that ID number must be stored as part of student record in order to locate a student later.
     2. Buckets – instead of an array of Student references, make it an array of references to a small List or Set (or Tree, learned later). Bucket will collect all students who have ID numbers with the same last 5 digits.
     3. What if the distribution is not uniform? (Many students have ID numbers that are clustered together, so we have many buckets that are empty while others are too big.)
        1. Multiply by a large prime, p first:

Bin number = (ID \* p) % 100,000

* 1. This is a hash table, but it will only work for data that has a field like an ID number (a whole number) What if we are storing data without such a field?
     1. A hash function is a function that maps objects to integers.
        1. Important: If the object is mutable, then care must be taken to always produce the same hash code. Otherwise the element can get lost in the table!
     2. Hash codes are the numbers that are assigned to each object.
     3. A good hash function will
        1. Scatter that data around the table (few elements have same hashCode)
        2. Be fast to calculate
     4. Example: Cat class where a Cat has a name (String) and a number of stripes (int).
        1. Could just use number of stripes as hashcode. (Terrible – very few unique values.)
        2. Could use ASCII code of first letter in name. (Only 26 hash codes)
        3. Could add together ASCII codes of the letters in the name. (Not too bad.)
           1. Still many collisions. The following names would generate the same hashcodes:

“EFG”, “GFE”, “FGE”… “DFH”, “FDH”, “HFD”, … “CFI”, “IFC”,… etc.

* + - 1. Could add together ASCII codes of the letters plus number of Stripes. (Even better).
      2. Why not multiply sum of ASCII codes by the number of stripes? (A lot of cats have 0 stripes!)
      3. How would a pro do it? In the String class, hashcodes are generated like this:

“fred” → ‘f’ \* 31^3 + ‘r’ \* 31^2 + ‘e’ \* 31^3 + ‘d’ \* 31^4

* + 1. Tips for obtaining a good hashing function
       1. Use as much of the immutable state as you can.
       2. Function should scatter values around as much as possible and minimize collisions
    2. What if you need to change state of object in table?
       1. Remove from table
       2. Change state
       3. Re-hash
    3. What happens when table gets too “full”?
       1. Define “load factor” (Number of elements / size of table)
       2. Table should be rehashed when load factor becomes very large
          1. Typically done for load factors > 2/3 or ¾
          2. Make a bigger table then rehash every element.
          3. Takes time O(n) if done a bit at a time
          4. In aggregate can be O(1) if done by multiplying size of table by a constant (like 1.3)
    4. Hashing in Java
       1. hashCode method is inherited from Object class (uses memory address of object to obtain hashCode)
       2. Return value can be any integer (positive, negative, or 0)
       3. Hashcode contract:
          1. If a.equals(b) then a.hashCode() must equal b.hashCode()
          2. Why? Inclusion in Java collections is based on equals. (Is there an element in the collection that is equal to this one I am holding?) If hashCodes for equal objects were different then we couldn’t check for inclusion properly.
       4. Important implication: If you override the equals method then you must override the hashCode method as well!
       5. How to go from hashCode to index in hash table (in Java):

Bucket # = Math.abs(x.hashCode() \* prime) % tableSize

* + - * 1. Why not do abs first, then multiply by prime? Because Math.abs(Integer.MIN\_VALUE) == INTEGER.MIN\_VALUE (overflow)

# Java classes for Sets

* 1. Set interface (Go over API)
  2. HashSet (backed by Hash Table)
     1. Elements must have hashCode implemented carefully!
  3. LinkedHashSet (backed by Hash Table overlaid with doubly linked list)
     1. Elements must have hashCode implemented carefully!
     2. Elements can be retrieved in order of insertion
     3. More overhead than HashSet (storing nodes with next/prev markers)
  4. TreeSet (Backed by a Tree , not a Hash Table– we’ll learn more about trees later)
     1. Elements must be Comparable
     2. Elements are sorted as they are inserted, so we can always iterate over the collection in sorted order
  5. Do some examples using the Java Set classes.

# Java classes for Maps

* 1. Map interface (Go over API)
  2. HashMap
     1. Keys must implement hashCode carefully
  3. LinkedHashMap
     1. Keys must implement hashCode carefully
     2. Key,Value pairs can be retrieved in order of insertion
  4. TreeMap
     1. Keys must implement hashCode carefully
     2. Key,value pairs can be retrieved in sorted order of keys
  5. Do some examples using the Java Map classes

# Recursion (Review)

* 1. Describe the “recursive” approach.
     1. If the dataset we’re working on is sufficiently small/simple, we provide an immediate answer.
     2. Otherwise, we take the dataset and break it into smaller pieces.
     3. We ask the method we are writing to provide solutions for these smaller parts, and we use this information to provide an answer for our original dataset.
  2. Do several preliminary examples of basic recursion. Some ideas are:
     1. factorial
     2. fibonacci
     3. exponentiation (as repeated multiplication)
  3. Do a long example of LinkedList implemented recursively.

# Trees

* 1. Draw memory diagram of tree (using linked nodes)
  2. Explain that there are never cycles
  3. Go over vocabulary/concepts:
     1. Node
     2. Root
     3. Leaves
     4. Interior nodes
     5. Parent
     6. Child
     7. Subtree
     8. Level
     9. Height
  4. What might implementation of a tree look like?

public class Tree<T> {

Node root;

private class Node<T> {

T data

Set<Node> children;

}

}

* 1. Define “binary tree”
     1. At most 2 children per node
     2. Left and right children are distinguishable (left and right, not a set of two things)
     3. Binary Tree Traversal
        1. Depth-First
           1. Pre-order
           2. In-order
           3. Post-order
        2. Breadth-First (using a Queue)
     4. Degenerate vs. well-balanced binary tree
     5. Binary Search Tree
        1. Binary tree
        2. Data must be comparable
        3. At every node:
           1. All values in left subtree are smaller than current node
           2. All values in right subtree are larger than current node
        4. Show trees that are examples of BST’s (and non-examples)
        5. Note that In-Order traversal yields elements in sorted order! This is actually a sorting algorithm students should know: Tree Sort.
           1. Start with empty tree
           2. Add all elements to tree
           3. Do an in-order Traversal
        6. Go over easy algorithm for search
           1. Time O(n) for degenerate tree
           2. Time O(log n) for well-balanced tree
        7. Go over algorithm for inserting
           1. O(log n) for well-balanced tree
           2. Insertions may unbalance the tree

There are self-balancing trees

Mention “rotations”, but we won’t cover it

* + - 1. Go over algorithm for deleting (also O(log n) if well balanced; also may unbalance tree)
      2. Binary Search Trees to implement a Map. (Go over an implementation like this)

public class BinarySearchTree <K extends Comparable<K>, V> {

private class Node {

private K key;

private V data;

private Node left, right;

public Node(K key, V data) {

this.key = key;

this.data = data;

}

}

private Node root;

}

# Singleton Design Pattern

* 1. Only one instance of class will ever exist
  2. Example – Polymorphic binary tree with two kinds of nodes:
     1. Non-Empty Nodes (normal)
     2. EmptyNodes (These replace “null” when left/right child is missing in Non-Empty Node)
        1. Implemented as Singleton

# Heaps

* 1. API
     1. Insert
     2. Remove smallest
  2. Define “complete” tree (all levels filled except last level that is filled from left-to right up to a certain position.)
  3. Draw examples of heaps as a tree
     1. Complete binary tree
     2. At each node, value is smaller than all descendants
     3. Self-balancing
  4. Go over insert algorithm
     1. Add to bottom of tree
     2. While smaller than parent, swap with parent
     3. Time O(log n).
  5. Go over getSmallest algorithm
     1. Copy root to reference
     2. Move last element up to root position
     3. While greater than either child
        1. Swap with smaller child
     4. Return reference to original root
  6. Is a tree (with linked nodes) the best data structure for heap? NO! Array is best!
     1. Show how heap data can be stored in an array
     2. Formulas for accessing parent and child nodes of node at index i:
        1. Parent: floor((i – 1) / 2)
        2. Left child: 2i + 1
        3. Right child: 2i + 2
     3. Go through algorithms for insert/getSmallest using these formulas, as needed.
        1. No wasted space
        2. Easier than keeping nodes (nodes would have to know where parent is – complex book-keeping)
  7. Max-heap is an obvious variation where the operation is removeLargest.
  8. Applications
     1. Heap Sort
        1. Start with empty heap
        2. Add all elements to heap
        3. While heap is not empty, removeSmallest
        4. Running time O(n log n)
     2. Priority Queue
        1. Enqueue
        2. Dequeue element with highest priority

# Graphs

* 1. Define vertex / edge
  2. Undirected graph (draw picture)
  3. Directed graph (draw picture)
     1. Include at least one pair of nodes with arrows going both ways
     2. Include a node that is connected to itself
     3. Predecessor
     4. Successor
  4. Show a graph with (optional) weights
  5. Define path
  6. Define cycle
  7. Define “connected” graph
  8. Graph traversal (visit each vertex exactly once)
     1. Breadth-first search (BFS)
        1. Describe with picture first
        2. Go over code using Queue for discovered vertices that have not been visited yet
     2. Depth-first search (DFS)
        1. Describe with picture first
        2. Go over code using Stack for discovered vertices that have not been visited yet
        3. Go over recursive implementation
  9. Graph implementation
     1. Idea #1 – A graph is basically a set of vertices. Each vertex keeps track of its neighbors.

class Graph<V> {

class Node {

V vertex;

Set<Node> adjacencies;

*OR*

Map<Node, Integer> adjacencies; // if weighted

}

Set<Node> nodes;

}

*// Might want to add*

class Edge<V> {

V from, to;

int weight; // optional, if weighted

}

Set<Edge> edges;

* + 1. Idea #2 – A graph is just a map from a vertex to a set of it’s adjacencies.

class Graph<V> {

Map<V, Set<V>> map;

*OR*

Map<V, Map<V, Integer>> map; // if weighted

}

* + 1. Idea #3 -- Adjacency matrix
       1. Draw a picture of the idea
       2. Potential implementation:

List<V> vertices // to give them index numbers

boolean[][] matrix;

OR

int[][] matrix; // if weighted graph

* 1. Djikstra’s algorithm
     1. Define problem of having a weighted graph and finding the path from one node to another with lowest cost.
     2. Explain that Djikstra’s algorithm actually does more than that: Given a particular Starting vertex, S, Djikstra’s algorithm will simultaneously find the lowest cost path from S to every other vertex on the graph.
     3. Go over Djikstra’s algorithm
        1. Show the pseudocode
        2. Trace it by maintaining a table for the cost and predecessors of each vertex
        3. At the end, show the simple algorithm for recovering the lowest cost path from S to any vertex on the graph.

# Sorting Algorithms

* 1. Define “comparison-based sort”
  2. State theorem (without proof) that fastest possible comparison-based sort is O(n log n).
  3. Are faster sorts possible? Yes, as fast as O(n), but they rely on properties beyond just “comparable” data.
  4. Comparison-based sorts
     1. Recall: Tree Sort
        1. Requires BST.
        2. Runs in time O( n log n) if tree is well-balanced
     2. Recall Heap Sort
        1. Requires Heap
        2. Runs in time O(n log n)
     3. Bubble Sort
        1. Demonstrate algorithm
        2. O(n^2)
     4. Selection Sort
        1. Demonstrate algorithm
        2. O(n^2)
        3. But tends to be fast when n is small (like < 20)
     5. Merge Sort
        1. Demonstrate
        2. O(n log n)
     6. Quick sort
        1. Demonstrate by selecting random pivots
        2. Worst case: O(n^2)
        3. Average case: O(n log n)
  5. Sorts based on more than just comparisons
     1. Counting sort
        1. Go over algorithm
        2. Only fast if range is O(n), but then runs in time O(n)
     2. Radix sort
        1. Go over algorithm
        2. Fast if number of digits and number of symbols possible is “small” compared to n. For a fixed number of digits (with a fixed number of symbols possible), it will be O(n).

# Concurrent programming

* 1. Define “process”
     1. Show “task manager” on Windows computer
     2. Talk about “multi-processing” and how CPU can only run one process per core.
     3. Processes typically do not share resources. How can they communicate?
        1. Files
        2. Network I/O
        3. Operating system (e.g. pipes in Unix)
  2. Threads
     1. “lightweight” processes
     2. A single process may spawn multiple threads
     3. Threads can share resources
     4. In Java, if your program spawns multiple threads, each has its own call-stack, but they share the same heap.
     5. Examples of programs that are typically multi-threaded:
        1. Web server (many clients connecting simultaneously)
        2. Web browser (threads will simultaneously GET documents like photos)
        3. GUI programming. In Java the Event Dispatching Thread is separate from the main thread that runs your code. So event handlers are running independently of the main thread.
  3. Java Thread class
  4. Runnable interface
  5. Example(s) of multi-threaded programs
  6. join
  7. Data races
  8. Synchronization
     1. Synchronized blocks
     2. Synchronized instance methods
  9. Deadlock

# Java I/O Streams

* 1. Byte Streams
     1. Overview
     2. InputStream, OutputStream, BufferedInputStream, BufferedOutputStream, FileInputStream, FileOutputStream
  2. Character Streams
     1. Overview
     2. Reader, Writer, BufferedReader, BufferedWriter, FileReader, FileWriter
  3. Data Streams
     1. Overview
     2. DataInputStream, DataOutputStream
  4. Object Streams
     1. Overview
     2. ObjectInputStream, ObjectOutputStream
  5. Parsing with the Scanner
  6. Files and Directories (using File class)

# Introduction to Networking

* 1. What are networks?
  2. I.P. Addresses
  3. Port numbers
  4. Packets
  5. UDP
  6. TCP
  7. Client/Server model
  8. Java Networking classes
     1. InetAddress
     2. DatagramSocket
     3. DatagramPacket
     4. Socket
     5. ServerSocket
     6. URLConnection
  9. Examples
     1. TCP client/server application
     2. UDP client/server application
     3. URL reader (reads contents of a web page)
     4. Multi-threaded Web server (process GET request and serves page)

# Algorithm Strategies

* 1. Divide and Conquer
  2. Dynamic Programming
  3. Greedy
  4. Brute Force
  5. Branch and Bound
  6. Backtracking
  7. Using heuristics