chapters - midterm

Prof Bill - created: Dec 2016

These are my 1-page summaries of chapters in our textbook:

Data Structures and Algorithms
by Goodrich, et al
wiley.com/college/goodrich

These are the chapters covered in the first half of 210. And the subject of our Midterm.
First half, bump up maps and hash tables:

- ➔ Ch 1, Ch 2 Java + OOP
- ➔ Ch 3, Ch 7 linked list + ArrayList + List ADT
- ➔ Ch 5 recursion
- ➔ Ch 6 stack and queue
- ➔ Ch 10 maps + hash tables + skip lists

Later in the course...
Second half, do graphs early:

- ➔ Ch 4 algorithm analysis
- ➔ Ch 14 graphs
- ➔ Ch 8 trees
- ➔ Ch 9 priority queue + heaps
- ➔ Ch 11 search tree
- ➔ Ch 12 search and sort

Optional

- ➔ Ch 13 text processing
- ➔ Ch 15 memory management

Missing:

- There's no GUI; use javafx, which is Ch 15 in Godfrey's book
- Streams API
- lambda exprs

Ch 1 Java Primer

/ I assume this is review material for 210 students. */*

Objects + **base types** = {boolean, char, byte, short, int, long, float, double}
In class, instance variables are private; **accessor**/getter and **mutator**/setter methods are public

Modifiers:

- **public**, **private**, **protected** - controls visibility to class variables and methods
- **abstract** - defines an interface, but no body/code
- **static** - makes a class variable or method (rather than instance)
- **final** - for variable, an initial value can never be changed; for method, it cannot be overridden

String class variables are immutable. Use **StringBuilder** to manipulate strings.

Simple I/O via console

- **System.out** is a **PrintStream** object, includes print() and println() methods
- Read from input stream using **Scanner** class with **System.in**

Section 1.7 An Example Program - review this

- **Notice (and copy) the structure!!!** instance variables; ctors; getters; update methods; main()
- private variables, public methods (why?)
- getter methods; no setters because variables are set in ctor and can't be changed after that
- printSummary is static, a class method (what's a better answer here?!?)

Just use **default package** for class
UML class diagram - a quick way to communicate class variables and methods

Javadoc - commenting standard used to produce documentation automagically (must use!); see page 51 example; the official Java API documentation is created using Javadoc, docs.oracle.com/javase/8/docs/api/

Consistent naming and indentation is part of quality code
Debugging = print statements or debugger

new operator "returns a reference to a newly-created object"; what's a "reference"?
method signature - the name parameters and return value of a method; this is the interface, not the body/code

What's the difference between an **instance variable** and a **class variable**? Method? How are these specified in Java?
ctor rules are complex: default ctor, ctor overloading, super, this, etc

Using Java from the **command line**: javac to compile, java to run your program
Scanner is nice for simple console input; see the 160/161 Muganda text for good examples
Just use **default package** for class; in larger projects, you'll use packages

Homework ideas: 1.8, 1.9, 1.10, 1.16, 1.21, 1.22, 1.26, 1.29 (all these are short programs)

Ch 2 OO Design

/ I assume this is review material for 210 students */*

design pattern - a common or “typical” solution to a design problem

polymorphism means “many forms” (example: Pet p = new Dog(“Brownie”);)

inheritance = is-a relationship
composition = has-a relationship

interface - code describing an API (methods)
abstract class - in between concrete class and interface, some methods are abstract
Interface is usually the starting point; sometimes you’ll do an abstract class to share snippets of code

exceptions - try, catch, throw, throws; exception hierarchy

generics - replace Object because “code became rampant with such explicit casts”

What is the **UML** representation for class, attributes, is-a relation, has-a relation? (see p 65)
The relations between classes is a critical design decision.
Some nice text/examples in Wikipedia: en.wikipedia.org/wiki/Class_diagram

For OOP, use public methods and private variables. Why?
Java supports **single inheritance**, but not multiple inheritance. Why?

Homework ideas: 2.12 (UML), 2.14 (exceptions)

- ❏ 2.33 is a fun program idea… use real books with expired copyrights, aka Project Project Gutenberg, http://www.gutenberg.org/
- ❏ 2.31… a fun time step simulation with bears and fish

Ch 3 Fundamental Data Structures

/ arrays, linked lists */*

3.1 Arrays

Array example: GameEntry, /* HEY! Always use {} for if() and loops */

- Declaration: GameEntry[] board;
- Allocation: board = new GameEntry[capacity];
- How many GameEntry objects have been created here? Why? Why is this necessary for arrays to work?

BIG array woes: shifting to insert/remove (see p 108), limit on num elements
BIG array plusses: O(1) access to the nth object, block memory is very efficient!
/* p 109 - another ArrayList groupie (ha!) */

Array example: insertion sort; can you describe in English?
Array example: Caesar cipher… notice the wraparound with modulo operator (%); also, note the main() test code included in the CaesarCipher class…
/* See that? your program can have multiple main() methods. You choose start class when running. */

Very useful nd practical Java API stuff, https://docs.oracle.com/javase/8/docs/api:

- **java.util.Arrays**, a useful bunch of static methods
- **java.util.Random**, useful pseudo-random methods (not static, get it?!?), seed

/* What is a seed in random number generation? This is so cool: www.random.org/bit-tally/ */

2D arrays, implemented as an array of arrays; google the syntax (ha)

3.2 Singly Linked Lists

nodes, head, tail
Draw the boxes! Critical… because the boxes translate directly to code!
Code for add_head, add_tail, remove_first, remove_all.
Example: nested Node class; user never sees the (private static) Node

3.3 Circular Linked Lists

store only tail pointer, calc head = tail.next   /* code a private getHead() method? */
/* only for people who dislike “== null” */

3.4 Doubly Linked Lists

sentinels + doubly-linked = slightly smaller/nicer code
/* notice no add/remove in the middle of list… why? */
Bad: More space for another pointer in each node
Good: Cleaner code; remove element in single v. doubly

3.5 Equivalence

4 options:

- ➔ a == b // pointer compare
- ➔ a.equals(b) // same as pointer compare because Array isn't a class
- ➔ Array.equals(a, b) // 1-D compare
- ➔ Array.deepEquals(a, b) // N-dimensional compare

3.6 Cloning

Object.clone() is shallow copy
Must override or it throws CloneNotSupportedException

Homework ideas: 3.2, 3.8, 3.12, 3.26

Ch 5 Recursion

/ pretty easy */*

5.1-3 Examples and analysis

A **recursive method** makes one or more calls to itself
A **recursive data structure** relies on smaller instance of the same structure, ala fractals, Mandelbrot set. Lots of recursive structures in nature: sunflowers, seashells, ferns, etc.
en.wikipedia.org/wiki/Mandelbrot_set ; www.nilsdougan.com/?page=writings/onfractals

Summation example: recursive, iterative, and equation
en.wikipedia.org/wiki/1_%2B_2_%2B_3_%2B_4_%2B_%E2%8B%AF

Factorial example: `n! = n * (n-1) * (n-2) * … 2 * 1`
...recursive and iterative solutions; en.wikipedia.org/wiki/Factorial

base case - fixed value, end of recursion
recursive case - function defined in terms of itself
activation frame - method call/return process, a stack of frames (ala CSC 220)

Binary search example: search sorted array (p 197 in text)
… recursive and iterative solution (iterative? how?); O(log N)

power, linear sum - bad examples of recursion

5.4 Designing recursive algorithms

Recursion is always more expensive than an iterative solution.
Elegance and simplicity of recursion solution may be worth it.
Key steps: Identify base case (end of recursion) and recursive case (to make problem smaller)

5.5 Recursion run amok

Element uniqueness problem (we'll discuss in chapter 4)

5.6 Eliminate tail recursion

Most linear recursion (tail recursion) can be easily be done with iteration.

Homework ideas: Implement 2 Fibonacci methods: recursive, iterative. For what N does each method take over 1 minute to run.

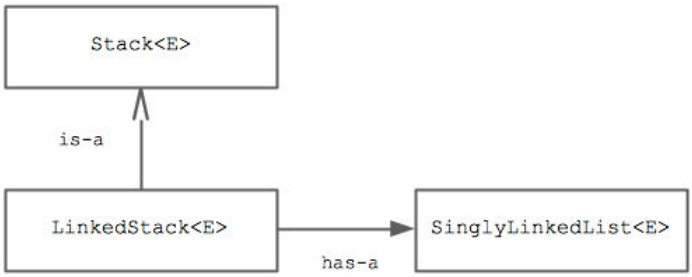
Ch 6 Stack, Queue, Deque

/ Easiest chapter in the book */*

6.1 Stack

Examples: Pez, plates, pancakes
LIFO = Last In, First Out
Stack ADT: push, pop to change Stack; top, size, isEmpty for info
Stack in **java.util.Stack** is old, V1.0; not consistent with new classes; use Deque instead

Array implementation: the classic; must know!; - fixed size; + O(1) and best for garbage collection because you don't have a zillion nodes like linked list
Linked list implementation: add/remove from head of list; O(1);
Adapter design pattern = is-a Stack<E>; has-a SinglyLinkedList



6.2 Queue

Examples: In line at the Jew-el, printer jobs
FIFO = First In, First Out
Queue ADT: enqueue, dequeue to change Queue; top, size, isEmpty for info
Queue in **java.util.Queue**

Array implementation: add to end, get from front; circular array using modulo (%); O(1)
Linked list implementation: add to end, get from front; easy!

6.3 Deque

General-purpose with insert/remove at front/end; pronounced "deck"
Deque ADT: addFirst, AddLast, removeFirst, removeLast to change Deque; first, last size, isEmpty for info
Deque in **java.util.Deque**

Homework ideas: Meh. Make sure you can code any of these structures with either an array or linked list.

Ch 7 List and Iterator ADT

/ ADT = Abstract Data Type */*

7.1 List ADT

The book's List is a simplified version of java.util.List from the Java Collections.

7.2 Array Lists

Same array problem with insertion and removal: shifting required
Array that automagically resizes, "algorithmic sleight of hand" in our text.
Understand resize() method on p. 264. Why an array of Objects? What happened to generic?!?
Performance analysis: resize cost is amortized over all operations, resulting in O(1) perf
Resize factor must be geometric; if arithmetic, then Big-Oh performance is impacted.

7.3 Positional Lists

Don't want to just make Node public. So add new object, Position.
P 274 - see Position<E> interface; just one method
p 277 - Node<E> is-a Position<E>; notice the sentinels used here; private method addBetween() is the key; notice how this method relies on sentinels

All this code/hassle is needed to handle one case: what if my Position gets removed?
p 280 - see remove() method; the trick: the Node's next/prev are set to null and that means he's been removed
p 280 - all operations O(1); but don't be tricked, they never search the list which would be O(N)

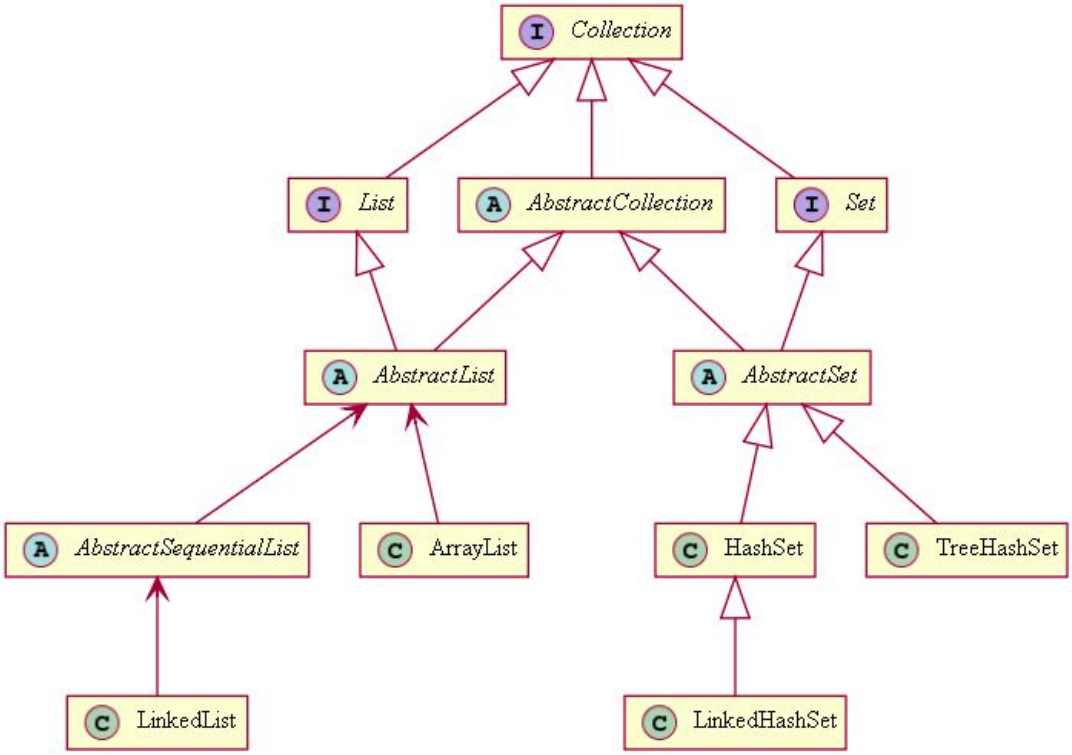
7.4 Iterators

Iterable interface; plugged into Java for-each syntax which is very nice
snapshot iterator - maintain private copy of sequence of elements
lazy iterator - no copying; remove() method tied to removable flag; this technique used in text

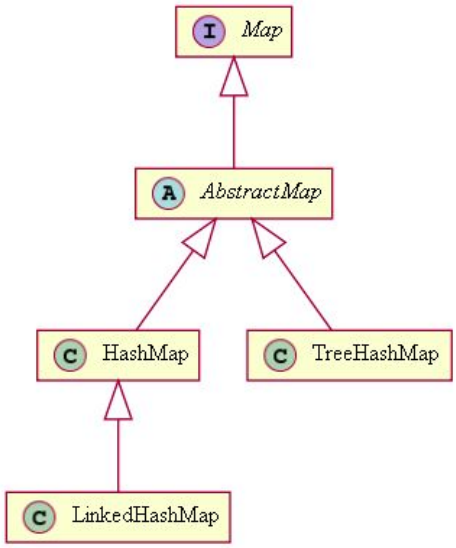
7.5 Java Collection Framework

Collection interface, we studied in this chapter.
Collections class, which is full of useful static methods like copy, min, max, shuffle, sort...
docs.oracle.com/javase/8/docs/api/java/util/Collections.html
Convert any Collection to an array with toArray() method.
Convert any array into a List using java.util.Arrays method asList(Array)

Here's a UML diagram.



Important to note the other universe in the Java Collections Framework: Maps!



Missing above: Collection is-a Iterable. Map is not.

Homework ideas:

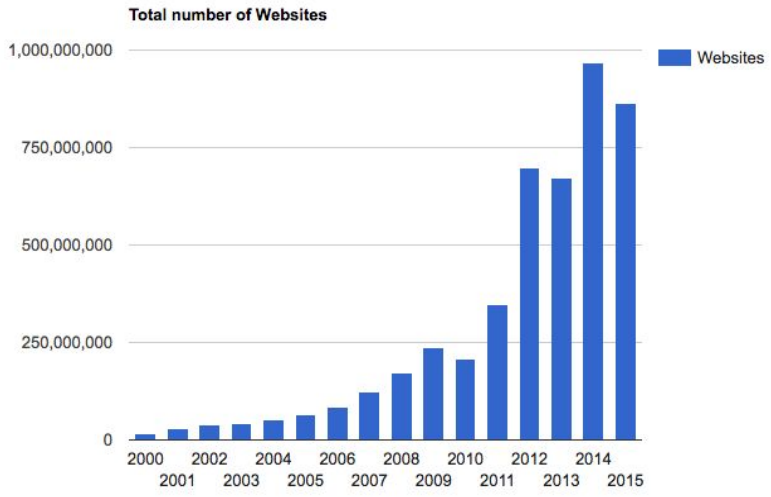
Ch 10 Maps, Hash Tables, and Skip Lists

/ Hash tables are key! This is a long, challenging chapter. */*

Map - efficiently store and retrieve values based on a key... **(key, value)** pairs

Hash table is a very popular map. And one of our most important data structures.
Example: Map website url to IP address.

www.internetlivestats.com/total-number-of-websites/



10.1 Map ADT

- **v get(k)** - return value associated with key
- **put(k, v)** - add (key, value) entry to map
- **remove(k)** - removes (key, value) entry from map
- **keySet()** - returns an iterable collection of keys in the map
- **values()** - returns an iterable collection of values in the map
- And blah blah… size(), isEmpty()

Java Collections Framework has a Map interface in **java.util.Map**
Maps are usually implemented with a hash table or a tree structure.
3 examples: counting word frequencies, AbstractMap, a Map using ArrayList

10.2 Hash tables

Hash table is-a Map. Arrays are soooo nice (fast and small). Use them for Map.
O(1) insertion and access = magic!
Help: en.wikipedia.org/wiki/Hash_table and www.visualgo.net/

Problem: how to convert any key into an integer in the range of [0-arraysize]?
Answer: **hash function**, modulo, so important that hashCode() is a method in Object
en.wikipedia.org/wiki/Java_hashCode()
Hash function **must be**: 1) easily calculated (no searching), 2) pseudo-random (no clustering), 3) reproducible (same result later for same object)

Hash functions: 1) use bit representation, 2) polynomial hash code, 3) cycle (bit) shift

Problem: how to handle **collisions** in the array?
Answer: Most common approaches

- **separate chaining** - buckets of collided keys; OK but extra space used
- **linear probing** - move down the array looking for an open spot; OK but tougher to remove items (leaves zombie spaces in array)
- **quadratic probing** - avoid clustering by adding k^2

> For a given hash value, the indices generated by linear probing are as follows:
>
> $H+1, H+2, H+3, H+4, \ldots, H+k$
>
> This method results in primary clustering, and as the cluster grows larger, the se
>
> An example sequence using quadratic probing is:
>
> $H+1^2, H+2^2, H+3^2, H+4^2, \ldots, H+k^2$

Source: en.wikipedia.org/wiki/Quadratic_probing

- **double hashing** - if you collide, hash again with diff function, then linear probing

Question: What if the hash table gets full and has lots of collisions?
Answer: Resize the array!

- ➔ **load factor** - percentage where resize happens
- ➔ Must re-hash every object because of modulo N!

Review: Code 10.7 abstract class, Code 10.8 sep chaining, Code 10.9 linear probing

Ch 10 continued...

10.3 Sorted Maps

This is **not** a major structure like hash table. It's just a sorted array. With binary search.
Fixes a major weakness in hash table: no sorting or search values in order.
Example: timestamp app, find entry closest to a time

Java Collections Framework: **java.util.SortedMap** and java.util.NavigableMap.
SortedMap ADT:

- `Entry<K,V> firstEntry(), lastEntry()`- return first/last entry
- `Entry<K,V> ceilingEntry(k)`- return entry with smallest key >= k
- `Entry<K,V> floorEntry(k)`- return entry with greatest key <= k
- lowerEntry and higherEntry same as above, but not equal to key k
- `Iterable<Entry<K,V>> subMap( k1, k2)`- return all entries with keys >= k1 and < k2

O(log N) access when using binary search
Review: page 430, Code frag 10.11-12 for `SortedTableMap`

10.5 Sets, Multisets, and Multimaps

set - unordered, no duplicates, like a set in Maths
Set ADT: add(e), remove(e), contains(e), iterator(). Also… union, intersection, subtraction
Java Collections Framework: **java.util.Set** interface. There is java.util.SortedSet too.

Implementation: Map with only keys, values unimportant. In Java Collections Framework: java.util.HashSet (hash table), java.util.ConcurrentSkipList (skip list), and java.util.TreeSet (balanced search tree)

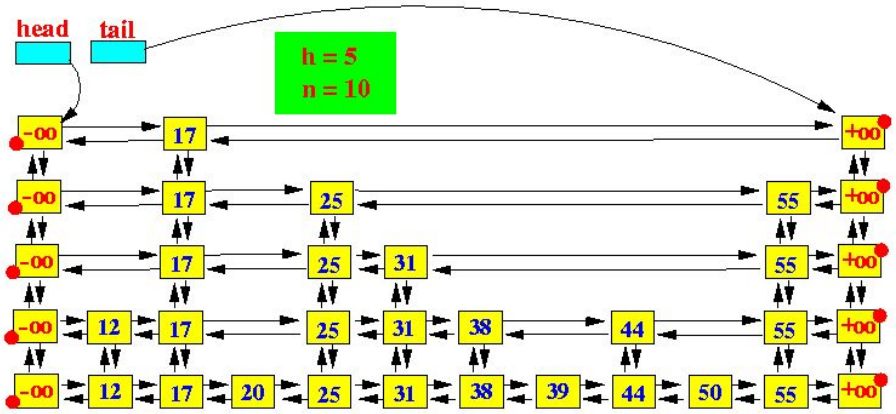
multiset - a bag, a set that allows duplicates
Multiset ADT: add(e), contains(e), count(e), remove(e), remove(e, n), size(), iterator()
Use a Map, where value is number of occurrences of the key

multimap - a map where a key can have multiple values
Not in Java Collections Framework

10.4 Skip Lists

Sort of like a binary search using a collection of linked lists.
Average performance = O(log n). Worst performance = O(n).

The basic structure: a doubly-linked list of doubly-linked lists. Use sentinels. Create levels with random “coin flip”. Each node has 4 pointers: next, prev, up, above, below.
Horizontal **levels**. Vertical **towers**.
In example below: h = 5, tower height or number of levels; n = 10, num items



Search pseudo-code (page 439 of our text):

```
Input: search key k;
Output: position p in bottom list S0 such that key(p) <= k
SkipSearch( k) {
  p = s   // start at head or top level
  while p.below != null {
    p = p.below
    while k >= key( p.next) {
      p = p.next
    }
  }
  return p
}
```

Insert pseudo-code (page 440 of our text)... this is a smushed version:

```
SkipInsert( k, v) {
  p = SkipSearch( k)  // above, return S0 node key <= k
  insert node for (k, v) after p
  while random == tails
    add level above
}
```