Outline

- exam 1 material: Java, ADTs, linked lists, run-time big-O, objects, references and pointers, iterators, invariants
- exam 2 material: generic types and container classes, stacks, queues, recursion, binary search
- binary trees, binary search trees, tree traversal, heaps, huffman coding, priority queues, hashing
- sorting: insertion, selection, bubble, mergesort, heapsort, quicksort

Java Concepts

- recursion
- references and pointers
- objects
- generic types and parametrized classes
- object equality
- object comparison
- iterators for collection classes

Java Concepts: recursion

- as a replacement for loops
- as a way of thinking of operations on lists, arrays (e.g. binary search)
- for tree operations, especially tree traversal
  - also for graph traversals, as long as you can guarantee termination
- recursion with return value to compute and return something
- recursion with void method to change or print something
- or a mix of these two

Java Concepts: references and pointers

- all variables in Java are either one of the 8 basic types, or
- a reference (pointer) to an object
- the reference may be `null`
- arguments (of a type other than the basic types) are passed by reference, so
- if they are modified (i.e. if their instance variables change), the caller can see the changes

Java Concepts: Objects and comparisons

- objects
- generic types and parametrized classes
  - generic types are only found as parameters to classes or interfaces
  - the corresponding actual type must be an object type
- object equality
- object comparison

Java Concepts: iterators

- iterators for collection classes
  - the iterator object must contain enough information to return all the contents of the collection class
    - one at a time
  - using an iterator is (and is meant to be) easy and convenient
  - implementing an iterator can be hard, e.g. for tree traversal

data structures 1/7
simple data structures

- arrays, ArrayLists, and Vectors
  - constant-time access
  - linear-time resizing
  - n-log-n sorting (mergesort, heapsort)
- stacks
- queues
- both stacks and queues can be implemented using either linked lists or arrays
  - for queues we use circular arrays
    - that is, index 0 is the index used after array.length – 1
- huffman trees (needed for huffman coding)

data structures 2/7
priority queues

- needs some way to compare objects
- many implementations:
  - heaps
    - heaps do not maintain order within a priority level
    - unless the key is a combination of priority and order
  - arrays of queues
    - only fast for relatively small number of priorities
  - ordered linked lists, arrays, etc
    - adding an element takes linear time

data structures 3/7
linked lists

- linked list class provides all the linked list operations: add, remove, size, toString, etc
- node class stores the value and next reference (recursive class, has a class variable of the same type as the class)
- the linked list object keeps a reference to the first node of a linked list
  - when we say “linked list”, sometimes we are referring to a linked list, and sometimes to a linked node
  - in-class exercise: in chained hashing, each array element would refer to linked list or to the first linked node?
- operations at the head (and at the tail, if a tail pointer is kept) take constant time
- most other operations take linear time
- ordered linked lists
  - need comparison to know when to insert, when to stop looking
  - search, insertion, deletion all take linear time

data structures 4/7
trees

- root, child, parent, sibling, etc
- tree nodes have zero or more children
- tree traversals: prefix, postfix, and, only for binary trees, infix
- most tree algorithms are well-suited to recursive implementation

data structures 5/7
binary trees

- logarithmic depth if the tree is balanced, for example, a heap
- otherwise, the worst case is linear depth
- many operations take time O(depth)
- other operations, e.g. tree traversal, take time O(nodes)

data structures 6/7
heaps

- a complete binary tree stored in an array
- heap property: each parent is greater (less) than either of its children
- when adding, add at the end (bottom) of the heap, then re-establish the heap property moving up the tree
- when removing, move the element at the end (bottom) of the heap to the top, then re-establish the heap property going down the tree

data structures 7/7
hash tables

- (best-case) constant-time access to keyed data
- hash function returns an int which is used as an index
- pseudo-randomness (hash function) is used to distribute data evenly
- different ways of handling collisions: increase array size, open addressing, chained hashing, separate storage

algorithms

- linked list operations: add, remove, search
- tree operations: add, remove, search
- ordered list insertion
- binary search
- heap insert and remove
- huffman coding
- prefix-to-infix-to-postfix and viceversa using stacks or using trees
- solution of different problems using priority queues (e.g. for huffman coding)

hashing algorithms

- hash functions: adding/XORing contributions from significant elements
  - cryptographic hash functions are slower, but hash much more thoroughly (and are still constant time)
- chained hashing (array of linked lists)
- open addressing: linear probing, quadratic probing, double hashing

sorting algorithms

- selection sort -- $O(n^2)$
- bubble sort -- $O(n^2)$, but $O(n)$ if array is sorted
- insertion sort -- $O(n^2)$, but $O(n)$ if the elements are at most a constant distance away from their correct position
- quick sort -- $O(n^2)$, but $O(n \log n)$ if the pivot is random and splits the array about evenly
- merge sort -- $O(n \log n)$
- heap sort -- $O(n \log n)$

run-time analysis

- a few typical functions:
  - O(log n): binary search
  - O(tree depth): going from the root to a leaf or from a leaf to a root: if the tree is balanced O(log n) – for example a heap, otherwise O(n)
  - O(n): traversing a data structure once, e.g. linked list removal, array insertion or expansion, linear search, best case for bubble sort and insertion sort
  - O(n log n): efficient sorting such as heap sort and merge sort, best case for quicksort
  - $O(n^2)$: most sorting algorithms, including worst case for insertion sort or quick sort, best and worst case for selection sort, worst and average case for bubble sort
- look at the loops and the changes in the loop variables
  - and the same for recursion
- might also have to consider big-O memory space usage

solving programming problems

- decompose the problem into reasonable elements
- use a class or method to implement each element
- combine these into a solution
- can analyze the solution for efficiency
- knowing data structures can help in decomposing the problem
- knowing algorithms can help if the problem is similar to one solved by an existing algorithm, or if the problem can be partially solved by an existing algorithm

concepts

- ADTs, and their representation by classes
- run-time analysis to determine big-O
- tree traversal
- invariants
- many different implementations possible for one interface, including for lists, queues, and stacks