Outline

- exam review
- recursion
- binary search
- stacks
- queues
- infix, prefix, and postfix expressions
- random numbers
- runtime analysis

exam review

- format similar to last exam
- material from lecture notes (including in-class exercises), book, assignments, quizzes
- for the book, all the material in Chapters 3, 4, and 5
- must also be familiar with the material presented before exam 1, i.e. review the material (and the exam)

review of recursion

- recursion is useful when we have a problem that:
  - has an easy solution for some base cases, and
  - for all other cases, has a solution that can be expressed in terms of solving a problem that is closer to the base case.
- the problem that is closer to the base case is often a smaller problem
  - e.g., smaller value of n
  - e.g., linked list shorter by one node

coding recursion

- recursion in Java (and in most languages) is only available at the level of methods
- the parameters of the method encode the problem, and the value of at least one of the parameters must be different on each call
  - otherwise, infinite recursion is guaranteed
- on each recursive call, the parameters must come closer to the base case

recursion is easier than loops for:

- anything that requires reversing the order of something:
  - in printing a number, the high-order digits should be printed first, but the low-order digits are more easily accessible (using modulo)
  - printing a linked list in reverse order.
- operation on recursive data structures such as linked lists (or trees)
- in-class exercise: write a recursive method to print a linked list in reverse order

in many cases recursion is equivalent to loops

- repeated operations can be usually implemented as either loops or recursive methods
- in most of these cases it doesn't matter much whether the solution is recursive or iterative (except as required on assignments and exams, or by your boss)
- operations on arrays
- binary search
- arbitrary (terminating) loops
- iterators do not benefit from recursion, but regular loops can be replaced by calls to recursive methods

infinite recursion

- each recursive call pushes on the stack the parameters of the call, and the return address
  - non-recursive calls do the same
- the stack is finite, and may only be a few megabytes
- if the stack is filled before the recursion ends, the program is terminated with stack overflow
- to avoid infinite recursion, each recursive call must come closer to a base case
- this is easy to guarantee on non-circular recursive data structures

binary search

- searching in a sorted array
- guaranteed logarithmic time
- look in the middle,
- select left or right half,
- and repeat until found
  - or guaranteed not found
- always need two indices to keep track of the start and end of the sub-array where the item may still be found
- recursive or iterative implementation: same performance, about same level of difficulty
- for exam, be able to understand and/or code binary search

stacks

- Last-In First-Out (LIFO) data structures
- two main implementations:
  - fixed-size (or variable-size) array: instance variables include array, index to top element
  - singly-linked list: pointer to first node, containing the top element of the stack
- be able to implement `push`, `pop`, and `size` for any of these
- main operations are constant time
  - the only linear time operation is to increase the array size when pushing new data on an array stack, and only if the stack grows as needed
- with the array size doubling whenever the stack grows, even on an array implementation of a stack, the main operations require amortized constant time

queues

- First-In First-Out (FIFO) data structures
- three main implementations:
  - singly-linked list: pointers to first and last node in queue
  - circular linked list: pointer to last node in queue
  - fixed-size circular array: start index, end index. Index arithmetic is done modulo the array size
- be able to implement offer and poll for at least the first and third
- the main operations are constant time
- double-ended queues have the operations of both stacks and queues
  - essentially, both add and remove at both ends of the Dequeue

infix, prefix, and postfix expressions

- infix: operator is between operands, needs precedence, parentheses
- prefix: operator is before operands
- postfix: operator is after operands
- implementing infix and prefix expressions requires an operand stack and an operator stack
- implementing postfix expressions only requires an operand stack: whenever an operator is encountered, pop the operands off the stack, execute the operation, and push the result
- remember that integer division rounds towards zero

random numbers

- truly random numbers require an unpredictable physical process
- pseudo random numbers are unpredictable rather than random
- for example, n' = (n * 45913 + 4137) % 65536 generates 16-bit random numbers: starting from 1, we get 50050, 61019, 36556, 22805, 46966, 23087, 18304, ...
- note in this example the last bit (even/odd) always alternates between 0 and 1

runtime analysis

- constant time: stack and queue insertion and deletion, linked list insertion or deletion at the front, linked list insertion at the back if a tail pointer is kept
- log time if doubling the size of the problem increases the time by a constant: binary search
  - soon we will see tree operations that take log time
- linear time: ordered linked list insertion or deletion, worst case tree insertion, deletion, or find

implementation

- exam questions may expect you to be able to implement methods covered in class, such as from classes:
  - ArrayStack.java
  - LinkedStack.java
  - BinarySearch.java (and the iterative equivalent)
  - LinkedListRec.java
- if any of these are asked, a description of the method would be given
- you may have to come up with the method return type and parameters. This is especially important for recursive methods