Outline: exam review

- iterators (also in homework 8)
- the collection interface
- recursive linked list methods (homework 9)
- stacks (also in homework 10)
- infix, prefix, and postfix expressions, expression trees
- queues
- trees, including binary search trees (also in homework 11)

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 2.10, 4, 6.1-6.3 and 6.5.
- must also be familiar with the material presented before exams 1 and 2, i.e. review the material (and the exams, quizzes, and homework assignments)
  - classes and interfaces, runtime analysis, binary search, recursion, lists, iterators, parametrized classes, invariants, algorithm runtime analysis
- review all the code posted on the course web page. Understand this code well enough to be able to code similar programs
  - the actual question may ask for something similar, and the corresponding code may be the same or different.

iterators

- an iterator is an object that implements the `Iterator` interface
- with methods `E next()` and `boolean hasNext()`
- iterators are created by calling the `iterator()` method of an `Iterable` object
- calling `next()` repeatedly returns all the objects in the collection
- a <u>foreach</u> loop transparently/implicitly/invisibly calls `iterator()` to create an iterator, and the `next()` and `hasNext()` methods to get successive objects:

```
for (E value: collection) { // the keyword is for, not foreach
  // can use value in the body of the loop
}
```

- a `ListIterator` can go backwards as well as forwards

collection interface

- parametrized interface with many implementing classes
    - AbstractCollection/List/SequentialList make it easier to implement specific types of collections
- many sub-interfaces, including Set, Stack, Queue, Deque
- sets do not add duplicate elements
    - equality is determined by the .equals method
- Map<K, V> stores objects of type V (value) as indexed by objects of type K (key)
    - a map must always store both the key and its value
-

recursive linked list methods

- generally:
  - have at least one parameter called `node` of type `LinkedNode<E>`
  - base case is when `node == null`
    - but the base case might be different, or there could be multiple base cases
  - recursive case takes `node.next` as parameter
- in most cases, return value contributes to the result
  - in which case the return type should be the type of the return value
- can do work before the recursive call
  - e.g. check for base cases
- and/or after the recursive call
  - e.g. add 1 to the result

stacks

- stacks are a fundamental data structure in computer science, used:
  - in the implementation of method calls,
  - in the evaluation of arithmetic expressions, and
  - in checking correctness of parentheses
- a stack is a LIFO data structure where only the top element is accessible
  - pushing a number of elements in order, pops them in reverse order
- `push(E value)` adds its parameter to the stack
- `pop()` removes and returns the top element of the stack

infix, prefix, postfix expressions, and expression trees

- normal infix expressions sometimes require precedence, associativity, and parentheses to correctly show which operations are evaluated first

  25 - (2 + 3) * 4 + 7

  - parentheses evaluate 2+3 first, precedence evaluates * next, left-to-right associativity evaluates – before +

- in a prefix expression, an operator is followed by two operands, each of which might in turn be a prefix expression

  + - 25 * + 2 3 4 7

- in a postfix expression, two operands (each of which might be postfix expressions) are followed by an operator

  25 2 3 + 4 * - 7 +

- must know (from book section 4.4) algorithms to evaluate postfix expressions using stacks

trees

- tree nomenclature: root, child, parent, leaf, interior node, level/height/depth of a node, height/depth of a tree
- <u>complete binary trees</u> and <u>perfect binary trees</u>
- expression trees
- binary search trees
- trees are a recursive data structure suitable for recursive methods

binary search trees

- tree is sorted so all values less than the value in a node are in the left subtree and all values greater than the value in the node are in the right subtree
- find, add, remove, take time O(depth)
- unless the tree is guaranteed to be balanced, O(depth) is O(n)
- find, add, and remove can be implemented recursively

exam review

- make sure your answer addresses the question
- reminders for coding questions:
  - read and understand the entire question
  - write clearly
  - only call outside methods if it is clearly appropriate
- a question asking for runtime is asking for <u>worst-case</u> runtime