Outline: exam review

- List, ArrayList, LinkedList
- parametrized types
- iterators
- invariants
- reminders: recursion, runtime analysis, binary search

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 2-2.9 and 5.**
- **must also be familiar with the material presented before exam 1, i.e. review the material (and the exam, quizzes, and homework assignments)**
  - classes and interfaces, runtime analysis, binary search.
- **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.

Collection and List

- **hierarchy of parametrized interfaces and classes**
  - including abstract classes and non-abstract (concrete) classes
  - List and Collection are interfaces, ArrayList and LinkedList are classes
- **the type parameter E indicates the types of objects stored by the collection**

```
interface List<E> extends Collection<E> { …
```

- **elements of a list are in order and have an index**

List

- **variable sized collection of objects in a particular sequence**
  - being in sequence, each element corresponds to a specific index in 0..list.size()-1
- **duplicate and null elements are permitted**
  - unless the list is special-purpose, e.g. in some of the assignments
- **the List interface defines required methods**
- **the AbstractList class helps in implementing new list classes**
- **some particularly useful methods of List:**
  - `boolean add(E e)` add at the end of a list
  - `void add(int index, E e)` add within the list
  - `Iterator<E> iterator()` create a new iterator
  - `E remove(int index)` remove and return the object at the given position
  - `int size()`

ArrayList

- **a list in which the elements are stored in an array, usually called `data`**
- **the array may have more elements than the list, so an instance variable (`size`) is needed to keep track of the size of the list**
    - `capacity()` returns `data.length`, the size of the underlying array
    - invariant: `size <= capacity()`
- **as the list grows, it may need a new array, so in the worst case adding at the end of the array is O(n)**
    - as long as the array grows by doubling the capacity, the time for adding n elements is O(n), meaning adding at the end takes <u>amortized</u> constant time
- **adding and removing at the beginning or in the middle of the array is always O(n) -- make sure you understand why**

LinkedList

- a list in which each element is stored in a node, and nodes are linked to each other
- the number of nodes is exactly the same as the number of elements
- each node has a reference to the value stored and a reference to the next node in the list: `item` or `value`, and `next`
- the end of the list is reached when the `next` field has the value `null`
- a linked list must have as instance variable the `head` of the list
- adding at the end of the linked list is faster if a `tail` pointer is kept -- O(1) instead of O(n)
- adding and removing at the front of a linked list is always O(1)

Circular and doubly-linked list

- **in a circular list, the `next` field of the last `LinkedNode` refers (back) to the head of the list**
- **in a doubly-linked list, each node has the `next` reference and also a reference to the node before it (`prev` or `previous`)**
    - the previous node of the head is null, and the next node of the tail is null
- **in a doubly-linked circular list, the tail is the previous node for the head, and the head is the next node for the tail**
    - prev and next are never null
-

Iterators

- **an iterator is an object that keeps track of the state of a traversal of a collection**
- **an iterator is like a bookmark for a book: there can be many bookmarks for a given book**
    - more than one of them might be on the same page
- **calling the iterator() method of a collection class is the common way to create a new iterator**
- **once the iterator exists, calls to hasNext() and next() may continue until there are no more elements**
    - or, can repeat calling next() until it throws an exception
- **examples:**

Using Iterators: Example 1/3

- calls to hasNext() and next() may continue until there are no more elements
- for example:

```
List<E> list = ...
Iterator<E> iter = list.iterator();
while (iter.hasNext()) {
  E variable = iter.next();
    // can use "variable" in the loop
}
```

Using Iterators: Example 2/3

- **can repeat calling next() until it throws an exception**

```
List<E> list = ...
Iterator<E> iter = list.iterator();
while (true) {
  try {
    E variable = iter.next();
      // can use "variable" in the loop
  } catch (NoSuchElementException e) {
    break;
  }
}
```

Using Iterators: foreach (Example 3/3)

- as an alternative to explicitly calling the iterator methods, use the foreach style:

```
for (E variable: list) {
  // can use "variable" in the loop
}
```

- `list` must implement `Iterable<E>` (or be an array)

Building an Iterator

- **for the exam, must understand and be able to implement at least the `hasNext()` and `next()` methods of LinkedListIterator.java and an iterator for ArrayLists, and of other simple iterators**
- **each iterator is its own separate class**
  - and may be internal to a collection class
- **the iterator needs access to the elements of the collection**
  - perhaps provided to the constructor
  - perhaps from the instance variables of the enclosing collection class
- **each iterator needs its own variables to keep track of its <u>state</u>, that is, where it is in the iteration**

Invariants

- **an invariant is something the programmer believes to be true about relationships among the program's variables**
- **for example: in binary search, if the object we are looking for is in the array, it <u>must</u> be between indices start..end**
    - this requires that the array must be sorted using the same comparison operation that is used in the binary search
- **in an object-oriented language such as Java, the most important relationships are among an object's instance variables (class data fields)**
- **if a public method is called with the invariant being true, the invariant <u>must</u> still be true at the end of the call**
- **a class invariant <u>must</u> always be true at the end of a public constructor**

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
- **recursion is a way of thinking as well as a programming technique**
  - mathematicians often think recursively

recursion examples on linked lists

- **the length of a linked list beginning with `node` is one more than the length of the list beginning with `node.next`:**

```
private long listLength(LinkedNode<E> node) {
  if (node == null) return 0;
  return 1 + listLength(node.next);
}
```

- **adding to or removing from a linked list is easy if we are willing to create new LinkedNodes for the backbone of the linked list:**

```
private LinkedNode<E> remove(LinkedNode<E> node, int index) {
  if (node == null) return null;
  if (index == 0) return node.next;   // remove this value
  return new LinkedNode<E>(node.value, remove(node.next, index - 1));
```

  - called as follows: `head = remove(head, index);`

runtime analysis

- **constant time: linked list insertion or deletion at the front, linked list insertion at the back if a tail pointer is kept**
  - amortized constant time for insertion at the end of an array list
- **log time if doubling the size of the problem increases the time by a constant: binary search**
  - later we will see tree operations that take log time
- **linear time: ordered list insertion or deletion, linear search**
- **quadratic time: removing n elements from the end of a singly-linked list or the front of an array list**
- **exponential time if adding one element doubles the time: towers of Hanoi, clear but inefficient computation of fibonacci numbers**