Sample Midterm

This sample midterm contains the kinds of questions that will be on the actual midterm examinations. It is much longer than the total size of the all midterm questions will be. At least one question on the (in-person) midterm will be all but identical to one here.

We will not post solutions to the sample midterm problems because, in our experience, people will be tempted to look at the solution before completely doing the problem. If anyone wishes to check their answers, they may give us a paper copy for review.

We will also try to answer questions by email, generally by giving hints or asking leading questions back.

1 Definitions and Motivation

The third part of the midterm (first half of lab time) will ask you to answer five questions of the following nature:

- What does the acronym "ADT" stand for? What is an ADT? What is the connection, if any, between an ADT and a data structure? Use an example to illustrate the answer to your last question. *See precorded lecture on ADTs.*
- What does `equals` do? Why must the method accept any object? What result must it return when given `null`? Why? *See precorded lecture on ADTs (Immutable ADTs: equals and hashcode).*
- When should a programmer use `equals` and when should they use `==` to compare two objects in Java? *See page 59 and 78 (3rd. ed. also 59 and 78).*
- What does it mean to throw an exception? How does one express throwing an exception in Java? Give two different ways program signal errors other than throw exceptions. Give one reason to use exceptions instead of these other ways. *See pre-recorded review lecture on exceptions.*
- In some of our linked list representations, we used "dummy nodes." What does this mean? What purpose does it serve? Also, how do dummy nodes affect how one implements iterators? *See pre-recorded lectures on dummy nodes and linked-list-varoants handout.*
- What does an iterator do? Why would we want an iterator for a sequence, rather than just using the sequence methods? What are advantages of each approach? Give examples. *See pre-recorded lecture on collectins and iterators.*
- What requirement must be established if you are going to dereference a variable? Explain how one can recognize a deference operation in the code, and then explain what the requirement for a safe dereference is and why. *See handout on dereferencing (Module 0).*
- Why might bugs be found by the JUnit tests, and NOT be found by the invariant checker? *See pre-recorded lecture on the Bag dynamic array invariant for information about the invariant checker (`wellFormed`).*
- Why might bugs only be found in the invariant checker rather than by the JUnit test case?
- What is a *model* field? How are they different from a normal field? What advantages do they have over regular fields? Disadvantages? *See pre-recorded lecture on Bag ADT (size).*

- What is an *interface*? Why do we use them? *See pre-recorded lecture on interfaces (Module 3).*
- What is an abstract class? Why do we have them? *See textbook chapter 13.4.*
- Explain `AbstractCollection`. How is it able to implement almost all the required methods? What can you say about the implementation? *See pre-recorded lecture on implementing collections.*
- Give three reasons why one would override a method in an abstract superclass. (Hint: when doing this sample midterm, you can empirically find the answer to this question by commenting out an overriding method in a homework and see what happens. Then put it back, and try another. What happens? Do the unit tests still pass? The efficiency tests? What goes on?) *See homework assignments 3 and 6.*
- Why is a nested class used to implement iterators not declared static?
- Explain the different flavors of linked lists that we have used in homework. *See handout on linked list variations.*
- What are *generic classes*? Why do we use them? Clearly explain the advantage(s). Give an example generic class from the standard Java collection library. *See pre-recorded lecture on generics (Module 6).*
- In some homework assignments we extended an abstract class rather than do all the work ourselves. Give a **specific example** of how this saved a lot of work. How is this **specific example** possible when the abstract class has no idea of how the data structure is implemented? The abstract class also included implementations that were not useful. Give a **specific example** and explain why the implementation was not used. How did we **specifically** avoid using it? *See homework assignments 3 and 6.*
- What distinguishes an *endogenous* list from an *exogenous* list? What peculiar properties do endogenous lists have? What benefits do they have? *See section on endogenous lists in linked-list variantion handout (Module 5).*
- A generic type parameter (such as `T`) can be used most places where a class name can be used. Where can it *not* be used? Why not? *See pre-recorded lecture on generics (Module 6).*
- Give an example of a bug that could be detected immediately by an invariant checker, but which could not be detected by a JUnit test for a while longer, if at all. Explain! Then give an example of a bug that could not possibly be detected by an invariant checker, but which could be detected immediately by a JUnit test. Explain this example too! Neither example should involve exceptions.
- Why do we say that nodes are "passive" unlike our regular classes? *See pre-recorded lecture on what linked lists are (Module 4).*
- Compare and contrast Stacks and Queues. *See pre-recorded lectures on stacks and queues (Module 7).*
- What are some classic situations where Stacks are used appropriately? Queues? *See testbook chapter 6 and 7.*

- Why does a queue implementation using a linked list require a tail pointer but a stack implementation does not?
- Give a compelling reason for using a linked list data structure to implement a sequence ADT. *See pre-recorded lecture on why linked lists (Module 4).*
- Give a compelling reason for using a dynamic array data structure for a stack ADT.
- How can you identify a constant-time operation? A linear-time operation? *See chapter 1.2 in the textbook.*
- Is it OK if a given state of a data structure corresponds to multiple possible ADT states? If so, give an example. If not, what could go wrong? *(won't be on midterm)*
- Is it OK if a given ADT state corresponds to multiple possible data structure states? If so, give an example. If not, what could go wrong? *(won't be on midterm)*

2 Reading Code

The first part of the in-person midterm (held during the first lecture time) will be about one of the homework solutions. We will ask five questions similar to what you see here. You will be provided with the relevant code of the solution, with documentation and debugging information removed.

1.
   - Why is the "`cents`" field of `Money` declared `final` and yet is given no value? What does this mean?
   - What's the point of fields that can never change?
   - In the `Money` constructor taking a double, why do we multiply the amount by 100?
   - What's with the special case for `Long.MIN_VALUE`? This value is less than `-Long.MAX_VALUE`. Why isn't it handled by the error check at the beginning? (The answer to this question requires understanding the nature of `double` representation.)
   - The `add` method of `Money` doesn't check that the argument is not null. Why not check for null and throw a "null pointer exception" if it is?
   - Why does the `add` method use `Math.addExact`? How would it be inexact to simply use addition?
   - Why does the code for `div` look so different from the code for `mul`? (Compare that `sub` is just like `add`.)
   - What does `cents % 100` (see `toString`) do?
   - Why does `toString` have to special case when we have fewer than ten pennies?
   - The instructions for `toString` said not to use doubles, even though Java provides formatting of double with two decimal points. Why not use this build-in facility?
   - Why does the `Money`'s `equals` method take something of type `Object` (not of type `Money`)?
   - Why did we bother in overriding the `equals` method? The `Object` class already provides an implementation. What is wrong with it?
   - In the first constructor for `Account` (the one with just an owner parameter), it merely says

```
this(owner, Money.ZERO, Money.ZERO);
```

What does this mean? Why write the code this way?

- In the `adjust` method for `Account`, the code calls `newBalance.compareTo(minimum)`. What does that do?
- The code throws an `OverdraftException`, but never catches it. Why not? Who will catch it?
- The constructor for `Transaction` that takes four parameters only checks the memo for null. Why does it do that and not check the amount too? Is the amount allowed to be null?
- Explain the `hashCode` implementation for `Transaction`.
- The `perform` method starts with some strange code, including an "add" call in which the result is discarded. Why structure the code in that way?

2.
- Why is there a `wellFormed` method? It's private. Who calls it? Why? What does it accomplish? Whose purposes does it serve? The user, the client, or the developer?
- What does the syntax `this(INITIAL_CAPACITY);` mean? What is it doing?
- Why do we not assert the invariant at the beginning of the constructor `TransactionSeq(int)`?
- The `start` method has a strange line of code in it:

  ```
  isCurrent = (manyNodes > 0);
  ```

  What does that do?
- The `atEnd` method is supposed to return a boolean. How come we don't see it return `true` under one condition and `false` otherwise?
- Why does `advance` only move the `currentIndex` when `isCurrent` is true?
- Why does `removeCurrent` bother looping through the array? Why isn't it enough to just replace the current element with null and decrement the field `manyItems`?
- Why does `ensureCapacity` double the array's size. Isn't that wasteful? Doubling is OK if the array is small, but from 1000 to 2000 if we're just adding a single element?
- Why does `addBefore` always "ensure" the capacity? Why not wait until the array is full?
- In `clone()`, there is the line: `answer.data = data.clone();` What does it do? What would happen if it were omitted? How would the client (of the class) notice anything wrong?
- What is the `Spy` class for?

3.
- The `TransactionCollection` class extends `AbstractCollection<Transaction>`. What does this mean?
- What is the difference between a transaction collection and a transaction sequence?
- Why do we not include javadoc for some of the public methods? When is that allowed in our course?
- Why did we override `clear`? Wasn't there an inherited version we could have used? What about `size`?

- Why did we special-case the empty collection `clear`? What if we skipped that line?
- How come we don't override `addAll`? Will the operation work? How? (Or will it simply throw an exception?)
- ~~We used the inherited implementation of `AbstractCollection`'s `remove` method (the one with an `Object` parameter), rather than override it and implement our own. But the inherited method doesn't even know about our `version` field. How can it update the version correctly?~~
- Which sequences of method calls should lead to a `NoSuchElementException`? `IllegalStateException`? `ConcurrentModificationException`? Come up with actual examples (a sequence of method calls on newly created objects.)
- The `clone()` method does a `super.clone()` and then clones the data array as well. Why was that second step necessary? Doesn't the `data` already have the needed information?
- What are `version` and `colVersion`? What do they have to do with `ConcurrentModificationExceptio`
- Is the `addBefore` method constant-time? How do you know?
- Why does the iterator's `remove` method update both `version` and `colVersion`? Wouldn't leaving them alone accomplish the same thing?
- In the iterator's `remove` method, why do we increment the version but decrement the count? Why not decrement both?

4.
- We didn't follow the textbook in having a concrete `cursor` pointer. But a cursor is very important for implementing methods such as `getCurrent`. Why wasn't it inefficient to compute it on demand?
- We also didn't follow the textbook in having a concrete `tail` pointer. Instead we have it as a "model" field. What does that mean? How did this decision make the coding tasks more complex? How did it make coding easier ?
- When checking for cycles, why is it clear that if the fast and slow pointers point to the same node, we must have a cycle? (It's much less obvious why they will eventually be the same if there is a cycle.)
- Why all the loops and stuff to see if the precursor points to a node in the list? Why don't we just check `precursor instanceof Node` or it's null?
- The `size` method returns the value of the field `manyNodes`, only after checking the invariant which involves looping through the whole list counting all the nodes. This seems very slow. How does this not fail the efficiency test? Equivalently, how is this efficient?
- The `start` method sets `precursor` to null. Doesn't this mean there won't be a current?
- The `start` method has a strange line of code in it:

  ```
  isCurrent = manyNodes > 0;
  ```

  What does that do?
- When checking whether the cursor is at the end (in `atEnd`), it checks that the cursor is null. Explain this code! Why is it good that this code doesn't include any `if` statements?
- In the code for `advance`, it only moves `precursor` if there is a current. Won't that mean we get stuck there and can never get past not having a current?
- In `addBefore`, we sometimes execute:

```
head = new Node(element,head);
```

Doesn't that create a cycle? Why would we want that?

- In `addBefore`, we have an 'if' statement. Why couldn't we have just one case (no conditions)?
- In `removeCurrent`, the field `precursor` is not changed. Why not? Isn't it now pointing to the wrong node?
- How is assigning pointers amount to removing something?
- In the `removeCurrent` method, we sometimes change head. Why? Won't we lose our list?
- Is `removeCurrent` a *constant-time* operation? How can you tell? How does that compare with Homework #'2 implementation of the same method?
- At the beginning of `addAll` we check if the addend is empty. Is this special case just for convenience? Or just for efficiency? Or is it really needed?
- In Homework #'2 `addAll`, we cloned the addend only if we were adding ourselves. Here, we do it all the time. What would go wrong if we didn't clone an addend that was a different sequence?
- Is `addAll` a *constant-time operation*? How can you tell? What if `tail` were a concrete field?
- The `clone` method does a lot of extra work creating nodes. What if we skipped this part? What problem would result?
- What if we copied the list but didn't bother changing `result.precursor` ?
- Is the special casing of `head` being null in `clone` necessary? Why or why not?

5.
- The data structure has a `last` pointer, but no `head`. How can we get the "head" pointer?
- The `wellFormed` method uses a variant of Floyd's algorithm (the Tortoise & Hare algorithm) to detect cycles. But I thought we *wanted* a cycle in the data structure? Why are we worried about cycles?
- We set the precursor to be null to start iteration. Why not rather put it on the `last` node? That would avoid a null pointer, and make the code simpler, wouldn't it?
- Why do we check the invariant twice in the `iterator` method?
- And why do we return `this`? Shouldn't we return a new iterator?
- The `hasNext` method looks too simple. Explain how it works if precursor is null.
- The `next` code has an 'if' with a special case. Why can't we use the general code ('else' branch) always?
- Before we return it, why do we set the "next" field of the result to null?
- Wait a minute, why are we reducing the count in `next` ?
- Why does `addBefore` just call `addAfter`? Don't they do different things? Adding before and after the current element?
- The `addAfter` code has four different cases. Show why no two of these case can use the same code.
- Why is the data structure used for this assignment so difficult to program? Why do we use a cycle, and not a nice normal null-terminated list?

- Why does the `toString` method get a comment `// implementation` when all the other ones are "required"? Was this method not required to pass the test?
- Why does `Register.add` not need to special case the contents being empty?
- Why does the loop add `f` after the cursor in the nested condition but before the cursor otherwise?
- Rewrite the `add` code to use a for-each loop.

6.
- The `Node` class has "prev" links as well as "next" links. Aren't these redundant? What use are they?
- What is going on with the warning suppression on the default constructor for a Node? Isn't that dangerous?
- The code for `wellFormed` doesn't use Floyd's algorithm (Tortoise & Hare). How can we sure there aren't any cycles?
- The dummy's data is required to be the dummy. What does that mean? How is that possible?
- On the other hand, in the loop we require that the data *not* be the node. Aren't these requirements contradictory?
- Why is the efficiency override for `clear` optional? Is the code given much faster than the code from `AbstractCollection` ? (Remind us how that is done!) Why won't we fail efficiency tests if we don't override `clear` ?
- The override for `addAll` is labeled "decorate." What does that mean?
- What special case do we handle and why? Why don't we use an iterator?
- What does `super.addAll(c)` do?
- Why does the `wellFormed` for the iterator return true if the versions do *not* match?
- The `next` method starts with three checks. What could cause each of these to fail? Is the order we check them important?
- The `remove` code does a lot of pointer changes. What's good about the way it is written (apart from being correct)?
- It makes sense to have cases for (1) adding at the head, (2) adding at the tail, (3) adding in the middle and (4) adding the very first element. How are these four cases handled by `addAfter` without any if's?

7. (Too soon before the mid-term examination)

3 Code

In the second part of the in-person midterm examination (second lecture time), you will be asked to code some simple task (on paper).

- Write a version of a new public member function `Bag#contains`, assuming `Bag` has the following fields:

```
private int count;
private Coin[] contents;
```

Fill the code in the following skeleton:

```
/** Return when a matching coin is found in the Bag
 * Return false otherwise.
 * @param item coin to match against (may be null)
 */
public boolean contains(Coin item)
{

  // Your code here!

}
```

- Do the same code assuming the following data structure:

```
private class Node {
  public Coin data;
  public Node next;
  public Node(Coin c, Node n) { data = c; next = n; }
}
private Node head;
public Bag() { head = null; }
```

- Implement the following function (using the array data structure)

```
/** Remove and discard all coins matching the argument.
 * @param item coin to match against (may be null)
 * @return number of coins discarded
 */
public int removeAll(Coin item)
{

  // Your code here!

}
```

- Do the previous problem, but with a linked list data structure.
- Write the object invariant for a singly-linked list with a head pointer, a tail pointer and a count of items.
- Implement insertion sort with a singly-linked list of integers: insert in backward order and then destructively reverse the result.
- Do the last code with generics and a comparator instead.
- Given the following definition for a doubly-linked list collection class without cycles or dummy nodes:

```
class StringList extends AbstractCollection<String> {
    private class Node {
        public String data;
        public Node prev, next;
        public Node(String d, Node p, Node n) { data = d; prev = p; next = n; }
    }
    private Node head, tail;
    private int numItems;
    private int version;

    private class MyIterator implements Iterator<String> {
        private Node nextNode = head;
        private boolean canRemove = false;
        private int myVersion = version;

        public String next() {
            ... // error checking (not shown)
            String result = nextNode.data;
            nextNode = nextNode.next;
            canRemove = true;
            return result;
        }
        public void remove() {
            // TODO
        }
    }
}
```

Write the `remove` method without using any other methods. Ignore invariants. Don't forget to implement "fail-fast" semantics.

- Implement a Queue generic class using the Java standard class `ArrayList`. You should implement the five functions: `isEmpty`, `size`, `enqueue`, `front`, and `dequeue`.
- Do the same for Stack with methods `isEmpty`, `size`, `push`, `peek`, and `pop`.
- Repeat the last two questions but use a circular doubly-linked list ADT with a dummy node.
- Describe (no code!) how an array can be effectively used to implement a Queue.

4 Debugging

The fourth and last part of the in-person mid-term examination will be in the lab in which you are given a repo with code that is fulty. You will ask to dagnose problems, answering questions such as:

1. How do you know that's there's a problem?
2. What is the example that the code fails to work with? Explain, drawing pictures if appropriate.

3. What is the problem? Why is the expectation correct? Why is what the program did wrong?
4. What did the program do wrong? Point to the section of code that does something wrong (or omits to do something required).
5. Fix the code.

- For an actual example from a previous semester: follow this invite link:

  https://classroom.github.com/a/glMd-fcd

  Then answer the five questions above.

- In an array-based Seq implementation, a friend wrote the following code for an insert method:

```
116 public void insert(CarControl element)
117 {
118   ensureCapacity(2*contents.length+1);
119
120   ++manyItems;
121
122   if (currentIndex == manyItems) {
123      contents[currentIndex] = element;
124   } else if (currentIndex == 0) {
125     for (int i=manyItems; i > 0; --i) {
126        contents[i] = contents[i-1];
127     }
128   } else {
129     for (int i=currentIndex; i < manyItems; ++i) {
130        contents[i+1] = contents[i];
131     }
132   }
133
134   contents[currentIndex] = element;
135
136 }
```

1. As soon as you see the code, without even looking at the actual details of the logic, you can see that it is likely to have bugs. Why? Explain!
2. There is a JUnit test suite that has lots of tests that stops after the first error. When you run the code with the JUnit test suite, the result is:

```
java.lang.ArrayIndexOutOfBoundsException: 1
at edu.uwm.cs351.DiskSeq.insert(DiskSeq.java:126)
at Driver.testDiskSeq(Driver.java:182)
at Driver.main(Driver.java:27)
Initial tests
Passed 112 tests.
Failed 1 test.
```

Why would "1" be a bad array index in the given code? Surely the array has at least one element?

3. You fix a problem on line 125, and now the code passes all tests. But there are still two errors. The first was not found by the test suite because it never tested inserting an element before the second element of a sequence with at least three elements. What is the bug?
4. Why does the test suite not detect the problem even though it inserts at all locations in a two element sequence (at the start, in the middle and at the end?)
5. There is another problem which happens when one inserts 28 elements in the sequence. It gets slower and slower and then there is an `OutOfMemoryError`. What is this error? Why did the test suite not catch it?

- Consider the following code to insert an item in a sorted linked list without a dummy node.

```
60  public void insert(T t) {
61    Node p;
62    for (p = head; head != null; p = p.next)
63      if (p.data.compareTo(t) > 0) break;
64    if (p == head) p = new Node(t, head);
65    else p.next = new Node(t,p);
66  }
```

1. If we insert the value 5 into an empty list, nothing happens. The list stays empty. Why did we get this problem? Fix the bug.
2. Now assume we have successfully inserted 5 into an empty sequence. If we then try to insert the value 7, it crashes with a "null pointer exception" on line 63. In the debugger it says that `p` is null. There's a simple thing wrong with the loop header on line 62. What? Fix it. (This doesn't fix the full problem, of course.)
3. Once we fix that problem, it still crashes when we attempt to insert 7 into the list, this time on the 'else' line (line 65). Again the debugger says that `p` is null. What happened? Fix the bug. This will require a bigger change.
4. Depending on how you fixed the previous bug, there may still be nasty bugs.

- Consider the following buggy definition for a doubly-linked list:

```
 1  class DoublyLinkedIntSeq {
 2    private class Node {
 3      int data;
 4      Node prev, next;
 5      Node(int d) { data = d; next = prev = null; }
 6    };
 7    private Node head, tail;
 8    private Node current;

15    private boolean wellFormed() { ... }

35    public DoublyLinkedIntSeq() {
```

```
36      head = tail = null;
37      assert wellFormed() : "Invariant false after constructor";
38    }


41    public void insert(int v) {
42      assert wellFormed() : "Invariant false before insert";
43      if (current == null) {
44        if (tail != null) {
45          Node n = new Node(v);
46          n.next = tail;
47          tail.next = n;
48          tail = tail.next;
49        } else {
50          head = tail = new Node(v);
51        }
52        current = tail;
53      } else {
54        Node p = new Node(v);
55        p.prev = current.prev;
56        p.next = current;
57        current.prev.next = p;
58        current.prev = p;
59        current = p;
60      }
61      assert wellFormed() : "Invariant false after insert";
62    }
63
64    public void advance() { // no bugs here!
65      if (!hasCurrent()) throw new IllegalStateException("at end already");
66      current = current.next;
67    }
68
69    public boolean hasCurrent() { // no bugs here!
70      return current != null;
71    }
72
73    public void print() { // no bugs here!
74      for (Node p = head; p != null; p = p.next) {
75        if (p != head) System.out.print(",");
76        System.out.print(p.data);
77      }
78      System.out.println();
79    }
```

To test the code, we try the following test:

```
82      DoublyLinkedIntSeq a = new DoublyLinkedIntSeq();
```

```
83      a.insert(112);
84      a.insert(66);
85      a.print();
```

This code crashes on line 57:

```
Exception in thread "main" java.lang.NullPointerException
        at DoublyLinkedIntSeq.insert(DoublyLinkedIntSeq.java:57)
        at DoublyLinkedIntSeq.main(Driver.java:75)
```

- In class we talked about rules for dereferencing. How was this rule violated on line 57?
- If we do the simplest thing to follow the rule and avoid the crash, the print call on rule 76 will only print `112`. What should be on line 57? (Fix the whole bug)
- In an independent test, someone does:

  ```
  87      DoublyLinkedIntSeq b = new DoublyLinkedIntSeq();
  88      b.insert(42);
  89      b.advance();
  90      b.insert(88);
  91      b.print();
  ```

  Whether or not the first bug (b) is fixed, this will print `42,8842,8842,8842,...` ad infinitum. Why? Explain!
- This problem would have been detected *before* the infinite loop by `wellFormed()` but it didn't get a chance because of the way that the programmer ran the test case. What is the job of `wellFormed()` and What problem could it have detected? Why doesn't it always run? (How is the invariant checking controlled by the tester?) Explain the reasoning behind this convention!
- Fix the code.