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 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 answer questions by email, generally by giving hints or asking leading questions back.

1 Definitions and Motivation

- What does the acronym “ADT” stands 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.

- What does equals do? Why is it particularly useful for immutable classes with public constructors to override equals? Why must the method accept any object? What result must it return when given null? Why?

- When should a programmer using equals and when should they use == to compare two objects in Java?

- 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.

- How are exceptions useful for implementing libraries or generic ADT’s used by different applications?

- 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?

- Compare and contrast cursors and iterators. What are the advantages of each? Give examples.

- 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.

- Distinguish exogenous from endogenous linked lists.

- Why might bugs only be found in the invariant checker rather than by the driver? Why is it preferable that they are found in the invariant checker rather than the driver?

- What is an interface? Why do we use them?

- What is an abstract class? Why do we have them?

- Explain AbstractCollection. How is it able to implement almost all the required methods? What can you say about the implementation?
• 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?)

• 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.

• 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.

• 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?

• What distinguishes an endogenous list from an exogenous list? What peculiar properties do endogenous lists have? What benefits do they have?

• 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?

• Give an example of a bug that could be detected immediately by an invariant checker, but which could not be detected by a test driver 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 test driver. Explain this example too! Neither example should involve exceptions.

• Describe the insertion sort algorithm. Under what conditions does it do particularly well?

• Compare and contrast Stacks and Queues.

• What are some classic situations where Stacks are used appropriately? Queues?

• Why does a queue implementation using a linked list require a tail pointer but a stack implementation does not?

2 Code

• Given the following definition for a doubly-linked list collection class without cycles or dummy nodes:

```java
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 template 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.

3 Debugging

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

```java
public void insert(CarControl element) {
    ensureCapacity(2*contents.length+1);
    ++manyItems;
    if (currentIndex == manyItems) {
        contents[currentIndex] = element;
    } else if (currentIndex == 0) {
        for (int i=manyItems; i > 0; --i) {
```
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 driver that has lots of tests that stops after the first error. When you run the code with the driver, 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 driver passes all tests. But there are still two errors. The first was not found by the driver because it never tested inserting an element before the second element of a sequence with at least three elements. (This is fixed now!) What is the bug?

4. Why does the driver 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 driver not catch it?

---

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

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

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:

```java
class DoublyLinkedIntSeq {
    private class Node {
        int data;
        Node prev, next;
        Node(int d) { data = d; next = prev = null; }
    }
    private Node head, tail;
    private Node current;
    private boolean _wellFormed() { ... }
    public DoublyLinkedIntSeq() {
        head = tail = null;
        assert _wellFormed() : "Invariant false after constructor";
    }
    public void insert(int v) {
        assert _wellFormed() : "Invariant false before insert";
        if (current == null) {
            if (tail != null) {
                Node n = new Node(v);
                n.next = tail;
                tail.next = n;
                tail = tail.next;
            } else {
                head = tail = new Node(v);
            }
        } else {
            current = tail;
        } else {
            Node p = new Node(v);
            p.prev = current.prev;
            p.next = current;
            current.prev.next = p;
            current.prev = p;
            current = p;
        }
    }
}
```
assert _wellFormed() : "Invariant false after insert";
}

public void advance() { // no bugs here!
if (!hasCurrent()) throw new IllegalStateException("at end already");
current = current.next;
}

public boolean hasCurrent() { // no bugs here!
return current != null;
}

public void print() { // no bugs here!
for (Node p = head; p != null; p = p.next) {
if (p != head) System.out.print(",");
System.out.print(p.data);
}
System.out.println();
}

To test the code, we try the following test:

DoublyLinkedIntSeq a = new DoublyLinkedIntSeq();
a.insert(112);
a.insert(66);
a.print();

This code crashes on line 57:

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

DoublyLinkedIntSeq b = new DoublyLinkedIntSeq();
b.insert(42);
b.advance();
b.insert(88);
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. What is the job of wellFormed() and What problem could it have detected? Why doesn’t it always run? Explain the reasoning behind this convention!
– Fix the code.