Section 4: Graphs and Testing

Agenda

- Graphs (HW 5)
- JUnit Testing
- Test Script Language
- JavaDoc

Graphs

- Node
- Edge

Node
+ data item in a graph

Edge
+ connection between two nodes
Graphs

- **Directed** graph: edges have a *source* and *destination*
- Edges represented with arrows
- Parent/child nodes: related by an edge

Nodes: states or objects within the graph
Edges: connection between two nodes

Edges can be:
- Directed
- Undirected

What are some examples where each type of edge would be useful?

* Common term: Directed Acyclic Graph (DAG)
**Graphs**

**Children of A?**
- Nodes reached by an edge starting at node A

**Parents of D?**
- Nodes that have an edge ending at node D

**Children of A:**
- Nodes reached by an edge starting at node A

**Parents of D:**
- Nodes that have an edge ending at node D
**Graphs**

Paths from A to C:
a sequence or ordered list of edges starting at A and ending at C

Shortest path from A to C?

REMINDER: You’ve seen Graphs before!

Linked Lists

Binary Trees

Before we move on...

Read the wikipedia article in the spec!

(It has implementation hints!)
Testing

Internal vs. external

- Internal: JUnit
  - How you decide to implement the object
  - Checked with implementation tests

- External: test script
  - Your API and specifications
  - Testing against the specification
  - Checked with specification tests

A JUnit test class

- A method with @Test is flagged as a JUnit test
- All @Test methods run when JUnit runs

```java
import org.junit.*;
import static org.junit.Assert.*;

public class TestSuite {
    @Test
    public void Test1() { ... }
}
```

Using JUnit assertions

- Verifies that a value matches expectations
  - assertEquals(42, meaningOfLife());
  - assertTrue(list.isEmpty());

- If the assert fails:
  - Test immediately terminates
  - Other tests in the test class are still run as normal
  - Results show “details” of failed tests (We’ll get to this later)
Using JUnit assertions

<table>
<thead>
<tr>
<th>Assertion</th>
<th>Case for failure</th>
</tr>
</thead>
<tbody>
<tr>
<td>assertTrue(test)</td>
<td>the boolean test is false</td>
</tr>
<tr>
<td>assertFalse(test)</td>
<td>the boolean test is true</td>
</tr>
<tr>
<td>assertEquals(expected, actual)</td>
<td>the values are not equal</td>
</tr>
<tr>
<td>assertSame(expected, actual)</td>
<td>the values are not the same (by ==)</td>
</tr>
<tr>
<td>assertNotSame(expected, actual)</td>
<td>the values are the same (by ==)</td>
</tr>
<tr>
<td>assertNull(value)</td>
<td>the given value is not null</td>
</tr>
<tr>
<td>assertNotNull(value)</td>
<td>the given value is null</td>
</tr>
</tbody>
</table>

• And others: [https://junit.org/junit4/javadoc/4.11/org/junit/Assert.html](https://junit.org/junit4/javadoc/4.11/org/junit/Assert.html)

Each method can also be passed a string to display if it fails:

• assertEquals("message", expected, actual)

Using JUnit assertions

• When writing JUnit assertions, make sure to use the appropriate test

Ex: Testing Java’s List.size()

Use `assertEquals(list.size(), 1)`

Don’t use `assertTrue(list.size() == 1)`

Checking for exceptions

✗ Verify that a method throws an exception when it should:

✗ Passes only if specified exception is thrown

✗ Only time it’s OK to write a test without a form of asserts

@Test(expected=IndexOutOfBoundsException.class)
public void testGetEmptyList() {
    List<String> list = new ArrayList<String>();
    list.get(0);
}

Setup and teardown

✗ Methods to run before/after each test case method is called:

@Before
public void name() { ... }

@After
public void name() { ... }

✗ Methods to run once before/after the entire test class runs:

@BeforeClass
public static void name() { ... }

@AfterClass
public static void name() { ... }
Setup and teardown

```java
public class Example {
    List empty;

    @Before
    public void initialize() {
        empty = new ArrayList();
    }

    @Test
    public void size() {...}

    @Test
    public void remove() {...}
}
```

Test Writing Etiquette

1. Don’t Repeat Yourself
   - Use constants and helper methods

2. Be Descriptive
   - Take advantage of message, expected, and actual values
   - Ex: `testAddElementToEmptyList` instead of `testAdd`

3. Keep Tests Small
   - Isolate bugs one at a time; failing assertion halts test
   - Helps to catch bugs at the source

4. Be Thorough
   - Test big, small, boundaries, exceptions, errors

5. Order of Testing Matters
   - If `methodB()` relies on `methodA()` to work correctly, test `methodA()` first

Ground rules

Let’s put it all together!

```java
public class DateTest {
    // Test addDays when it causes a rollover between months
    @Test
    public void testAddDaysWrapToNextMonth() {
        Date actual = new Date(2050, 2, 15);
        actual.addDays(14);
        Date expected = new Date(2050, 3, 1);
        assertEquals("date after +14 days", expected, actual);
    }
}
```
How to create JUnit test classes

- Right-click hw5.test -> New -> JUnit Test Case
- **Important**: Follow naming guidelines we provide
- Demo

JUnit asserts vs. Java asserts

- We’ve just been discussing JUnit assertions so far
  - Tests for incorrect behavior
- Java itself has assertions
  - Tests for invalid states

```java
public class LitterBox {
    ArrayList<Kitten> kittens;
    public Kitten getKitten(int n) {
        assert(n >= 0);
        return kittens(n);
    }
}
```

Reminder: Enabling asserts in Eclipse

To enable asserts:
Go to Run -> Run Configurations… -> Arguments tab -> input `-ea` in VM arguments section

Don’t forgot your CheckReps!
Expensive CheckReps

- `ant validate` and staff grading will have assertions enabled
- But sometimes a checkRep can be expensive
  - For example, looking at each node in a Graph with a large number of nodes
- This could cause the grading scripts to timeout

Before your final commit, remove the checking of expensive parts of your checkRep or the checking of your checkRep entirely

Example: boolean flag and structure your checkRep as so:

```java
private void checkRep() {
    cheap-stuff
    if(DEBUG_FLAG) { // or can have this for entire checkRep
        expensive-stuff
    }
    cheap-stuff
    ...
```

External tests: Test script language

- Text file with one command listed per line
- First word is always the command name
- Remaining words are arguments
- Commands will correspond to methods in your code
Test script language

# Create a graph
CreateGraph graph1

# Add a pair of nodes
AddNode graph1 n1
AddNode graph1 n2

# Add an edge
AddEdge graph1 n1 n2 e1

# Print the nodes in the graph and the outgoing edges from n1
ListNodes graph1
ListChildren graph1 n1

Test script language

# Create a graph
CreateGraph graph1

# Add a pair of nodes
AddNode graph1 n1
AddNode graph1 n2

# Add an edge
AddEdge graph1 n1 n2 e1

# Print the nodes in the graph and the outgoing edges from n1
ListNodes graph1
ListChildren graph1 n1

Test script language

# Create a graph
CreateGraph graph1

# Add a pair of nodes
AddNode graph1 n1
AddNode graph1 n2

# Add an edge
AddEdge graph1 n1 n2 e1

# Print the nodes in the graph and the outgoing edges from n1
ListNodes graph1
ListChildren graph1 n1

How to create specification tests

✗ Create .test and .expected file pairs under hw5.test
✗ Implement parts of HWSTestDriver
   + driver connects commands from .test file to your Graph implementation to the output which is matched with .expected file
✗ Run all tests by running SpecificationTests.java
   + Note: staff will have our own .test and .expected pairs to run with your code
   + Do not hardcode .test/.expected pairs to pass, but instead make sure the format in hw5 instructions is correctly followed
Workflow for Specification Tests

1. Read in commands
2. Translate commands, apply them to your graph
3. Return results of commands
4. Formats output to match expected output style

Demo: Test script language

JavaDoc API

- Now you can generate the JavaDoc API for your code
- Instructions in the Editing/Compiling Handout
- Demo: Generate JavaDocs
- Demo steps are in spec