CSE 331
Software Design & Implementation
Section: Graphs; Testing
Reminders

• None!

Upcoming Deadlines

• HW4         due 11pm tonight (7/13)
• Prep. Quiz: HW5  due 11pm Tuesday (7/17)
Last Time...

- Specifications
- Abstract Data Types (ADTs)
  - Representation Invariants
  - Abstraction Functions
- Testing
  - Testing Heuristics
  - JUnit (section)

Today’s Agenda

- Graphs
- HW5
  - Specification tests
  - JUnit tests
- Review: Specifications
Graphs
Graphs
A graph represents relationships

A graph is a set of nodes and a set of edges between them.

Nodes may be labeled.

Edges may be labeled.

Edges may have a direction.
Example: Road Map

Nodes: intersections (cities)
Label: name/location

Edges: roads
Label: name/length
Example: Airline Flights

**Nodes:** airports

**Label:** airport code

**Edges:** flights

**Label:** cost/time
Example: CSE courses

**Nodes:** Courses

**Label:** Course name

**Edges:** pointer to next class

**Label:** none
You’ve used graphs before!

Singly linked Lists:

Nodes: Linked list node
Label: integer

Edges: pointer to next node
Label: none
You’ve used graphs before!

**Doubly linked Lists:**

- **Nodes:** Linked list node
- **Label:** integer
- **Edges:** pointers to prev/next nodes
- **Label:** none
You’ve used graphs before!

**Binary trees:**

- **Nodes:** Tree node
- **Label:** Integer

- **Edges:** Pointers to children
- **Label:** None

```plaintext
- 42
  - 8
    - -3
    - 40
  - 43
    - 98
```
An edge points from source to dest.

Each edge “points” from a source to a destination.
• **Outgoing** from source
• **Incoming** to destination

N.B.: We’re only dealing with directed graphs from here on out.
An edge points from source to dest.

Each edge “points” from a source to a destination.

- **Outgoing** from source
- **Incoming** to destination

Edge A is **Node 1 → Node 2**.

- Outgoing from **Node 1**
- Incoming to **Node 2**
An edge points from source to dest.

Each edge “points” from a source to a destination.
- **Outgoing** from source
- **Incoming** to destination

Edge C is **Node 2 → Node 3**.
- Outgoing from **Node 2**
- Incoming to **Node 3**
A node has children

A node’s outgoing edges point to its children.

- Potentially empty set
A node has children

A node’s outgoing edges point to its **children**.
- Potentially empty set

Node 3 has three children:
- Node 1
- Node 4
- Node 5
A node has children

A node’s outgoing edges point to its children.
- Potentially empty set

Node 2 has two children:
- Node 2
- Node 3
A node has parents

A node’s incoming edges point from its **parents**.
- Potentially empty set
A node has parents

A node’s incoming edges point from its **parents**.
- Potentially empty set

Node 4 has two parents:
- Node 3
- Node 5
A node has parents

A node’s incoming edges point from its **parents**.
- Potentially empty set

Node 5 has one parent:
- Node 3
A node has neighbors

A node’s **neighbors** are its children plus its parents.
- Potentially empty set
A node has neighbors

A node’s **neighbors** are its children plus its parents.

- Potentially empty set

Node 2 has four neighbors:

- Node 1 (parent)
- Node 2 (self-pointing)
- Node 3 (child)
- Node 4 (parent)
A node has neighbors

A node’s **neighbors** are its children plus its parents.
- Potentially empty set

Node 3 has four neighbors:
- Node 1 (child)
- Node 2 (parent)
- Node 4 (parent and child)
- Node 5 (child)
A path is a “chain” of edges from a source to a destination.
- Potentially empty sequence
- Might include a cycle
- Often want shortest
A path is a “chain” of edges from a source to a destination.
- Potentially empty sequence
- Might include a cycle
- Often want shortest

Path from Node 1 to Node 5:
1. Edge A : Node 1 → Node 2
2. Edge C : Node 2 → Node 3
3. Edge E : Node 3 → Node 4
4. Edge F : Node 4 → Node 3
5. Edge G : Node 3 → Node 5
A **path** is a “chain” of edges from a **source** to a **destination**.

- Potentially empty sequence
- Might include a cycle
- Often want shortest

Path from **Node 1** to **Node 1**:
1. Edge A : Node 1 → Node 2
2. Edge C : Node 2 → Node 3
3. Edge B : Node 3 → **Node 1**
A path is a “chain” of edges from a source to a destination.
• Potentially empty sequence
• Might include a cycle
• Often want shortest

Path from Node 2 to Node 2:
1. Edge I: Node 2 → Node 2
Possible graph operations

Creators
• Construct an empty graph

Observers
• Look up node(s) by label, children of, parents of, neighbors of, ...
• Look up edge(s) by label, incoming to, outgoing from, ...
• Iterate through all nodes
• Iterate through all edges

Mutators
• Insert/remove a node
• Insert/remove an edge

You may not want to include all of these operations in your graph ADT design.

More observers
• Find path(s) from one node to another
• Find all reachable nodes
• Count indegree, outdegree
HW5: Preview
HW5: Design before implementation

- HW5: Building an ADT for labeled, directed graphs
  - Labeled: Nodes and edges have label values (Strings)
  - Directed: Edges have direction
  - Edges with same source and destination will have unique labels

- The exact interface of your Graph class is up to you
  - So no given JUnit tests bundled with the starter code
  - Reminder: Not a generic class.

- HW5 is just designing and specifying the ADT
  - HW6 will be implementing it
HW5: What’s Included

• Your submission for HW5 should include:
  – Java class(es) that represent your ADT
    • Each with method stubs
  – Specifications for all classes and methods
  – Tests for your ADT
    • JUnit and Script tests (coming soon...)

• Your submission for HW5 should not include:
  – Any implemented methods
  – Anything private (fields, methods, classes, etc.)
    • Including RI and AF
HW5: Specifications in JavaDoc

• Write class/method specifications in proper JavaDoc comments
  – See “Resources” → “Class and Method Specifications”

• You can generate nice HTML pages cleanly presenting all your JavaDoc specifications
  – Placed in “build/docs/javadoc/”

• This is a great way to verify the JavaDoc is formatted correctly
  – And to review/proofread your work...

• Let’s look at the JavaDoc from HW4... (demo)
JavaDoc Demo

- Run the “javadoc” gradle task (in the documentation folder)

- Locate `build/docs/javadoc/index.html`, right-click, **Open In** > a browser of your choice
  - Look for formatting errors or missing components!
HW5: Testing

• The design process includes crafting a good test suite
  – Script tests and JUnit tests

• **Script Tests** (*src/test/resources/testScripts/*)
  – Test script files `name.test` with corresponding `name.expected`
  – Validate behavior intrinsic to high-level concept (abstract meaning)
  – Tested properties should be expected of any solution to HW5

• **JUnit Tests** (*src/test/java/graph/junitTests/*)
  – JUnit test classes
  – Validate behavior that can't be tested with script tests.

• If you can validate a behavior using either test type, use a script test!
HW5: Script Tests

Each script test is expressed as text-based script `foo.test`
- One command per line, of the form: `Command arg_1 arg_2 ...
- Script’s output compared against `foo.expected`
- Precise details specified in the homework
- Match format **exactly**, including whitespace!

<table>
<thead>
<tr>
<th>Command (in <code>foo.test</code>)</th>
<th>Output (in <code>foo.expected</code>)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CreateGraph <code>name</code></td>
<td><code>created graph name</code></td>
</tr>
<tr>
<td>AddNode <code>graph label</code></td>
<td><code>added node label to graph</code></td>
</tr>
<tr>
<td>AddEdge <code>graph parent child label</code></td>
<td><code>added edge label from parent to child in graph</code></td>
</tr>
<tr>
<td>ListNodes <code>graph</code></td>
<td><code>graph contains: label_node ...</code></td>
</tr>
<tr>
<td>ListChildren <code>graph parent</code></td>
<td><code>the children of parent in graph are: child(label_edge) ...</code></td>
</tr>
<tr>
<td><code># This is comment text ...</code></td>
<td><code># This is comment text ...</code></td>
</tr>
</tbody>
</table>
HW5: `example.test`

```plaintext
# 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 all nodes in the graph
ListNodes graph1

# Print all child nodes of n1 with outgoing edge
ListChildren graph1 n1
```
# Create a graph
created graph graph1

# Add a pair of nodes
added node n1 to graph1
added node n2 to graph1

# Add an edge
added edge e1 from n1 to n2 in graph1

# Print all nodes in the graph
graph1 contains: n1 n2

# Print all child nodes of n1 with outgoing edge
the children of n1 in graph1 are: n2(e1)
HW5: Why Script Tests?

• Everyone’s implementation could (will!) be different, so we (staff) cannot write JUnit tests for everyone to use or to use for checking everyone’s code.

• We still need a way to test that you specify and implement the proper behavior, so we use script tests that work regardless of the implementation.

• They test what the methods are doing, they don’t care how the methods are doing it.
HW5: Creating a script test

1. Write test steps as script commands in a file `foo.test`
2. Write expected ("correct") output in a file `foo.expected`
   - ...taking care to match the output format exactly
3. Place both files under `src/test/resources/testScripts/`
4. Run all such tests via the Gradle task `scriptTests`
   - After class implemented and `GraphTestDriver` stubs filled
HW5: Test Commands vs Methods

- Your graph should not have the exact same interface as the script test commands
  - e.g. you should not have a method called `AddNode()` that adds a node to the graph and prints out/returns the string “added node n1 to graph1”
  - This wouldn’t make much sense for other graph clients!

- But you will need the ability to add a node!

- Later, we will need some way to map script test commands (`AddNode graph1 n1`) to some Java code that uses the methods of your graph class
  - This is part of HW6; do not worry about for now
HW5: Script tests vs. JUnit Tests

• Script tests will not cover every case for your graph:
  – What if you have additional methods that can’t be tested by our script test commands?
  – What about “bad” input for your graph?
  – What happens when you try to add the same node twice?
  – …

• We need some way to test cases that cannot be covered by our script tests

• For this, we use JUnit tests.
HW5: Creating JUnit tests

1. Create JUnit test class in src/test/java/graph/junitTests/

2. Write a test method for each unit test

3. Run all such tests via the Gradle task junitTests

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

/** Document class... */
public class FooTests {
    /** Document method... */
    @Test
    public void testBar() { /* JUnit assertions */ }
}
```
HW5: Creating JUnit tests

• Note: Your JUnit tests will fail in HW5, because you have not implemented the actual methods yet
  – The same goes for your script tests

• You will get them passing in HW6
Specifications
Specifications

Suppose we have a `BankAccount` class with instance variable balance. Consider the following specifications (ignore `@param`):

A. @effects decreases balance by amount
B. @requires amount >= 0 and amount <= balance
   @effects decreases balance by amount
C. @throws InsufficientFundsException if balance < amount
   @effects decreases balance by amount

Which specifications does this implementation meet?

```java
void withdraw(int amount) {
    balance -= amount;
}
```
Suppose we have a `BankAccount` class with instance variable `balance`. Consider the following specifications (ignore `@param`):

A. `@effects decreases balance by amount`
   - **✗**

B. `@requires amount >= 0 and amount <= balance`
   - `@effects decreases balance by amount`
   - **✓**

C. `@throws InsufficientFundsException if balance < amount`
   - `@effects decreases balance by amount`
   - **✗**

Which specifications does this implementation meet?

```java
void withdraw(int amount) {
    if (balance >= amount) balance-=amount;
}
```
Specifications

Suppose we have a `BankAccount` class with instance variable balance. Consider the following specifications (ignore `@param`):

A. `@effects` decreases balance by amount

B. `@requires` amount >= 0 and amount <= balance
   `@effects` decreases balance by amount

C. `@throws` InsufficientFundsException if balance < amount
   `@effects` decreases balance by amount

Which specifications does this implementation meet?

```java
void withdraw(int amount) {
    if (amount < 0) throw new IllegalArgumentException();
    balance -= amount;
}
```
Suppose we have a `BankAccount` class with instance variable `balance`. Consider the following specifications (ignore `@param`):

A. `@effects` decreases balance by amount  
   ![Checkmark]

B. `@requires` amount >= 0 and amount <= balance  
   `@effects` decreases balance by amount  
   ![Checkmark]

C. `@throws` `InsufficientFundsException` if balance < amount  
   `@effects` decreases balance by amount  
   ![Checkmark]

Which specifications does this implementation meet?

```java
void withdraw(int amount) throws InsufficientFundsException {
    if (balance < amount) throw new InsufficientFundsException();
    balance -= amount;
}
```
Testing

Consider the **BankAccount** class again. What are some good test cases?

```java
public class BankAccount {
   /** @return current balance of account */
   public void balance() { ... }

   /**
   * @param amount to withdraw
   * @requires amount >= 0
   * @throws InsufficientFundsException
   *         if balance < amount
   * @effects decreases balance by amount
   */
   public void withdraw(int amount) { ... }
}
```

- Specification test heuristic:
  - amount <= balance
  - amount > balance

- Boundary test heuristic:
  - amount = balance
  - amount > balance

- Others?

- Should we test amount < 0?
Before next lecture...

1. Do **HW4** by tonight! (reminder: deadline is 11pm)
   - Written portion (submit PDF on Gradescope)
   - Coding portion (push and tag on GitLab)

2. Review JUnit testing slides discussed in the last section.