CSE 331
Software Design & Implementation

Winter 2023

Section 4 – Graphs, Testing

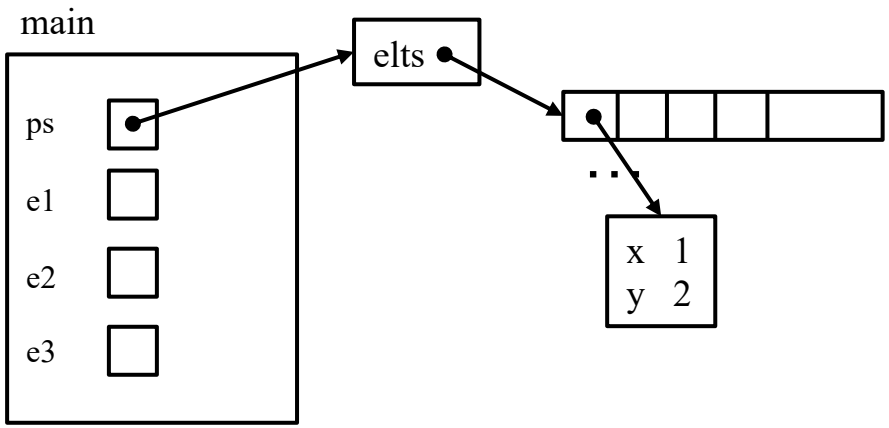
Administrivia

- Done with HW3!
- HW4 due today! **11 pm**
- HW5-1 Spec out on the website now
  - Always plan for work taking 3x longer than expected, so start early!
- Any questions?

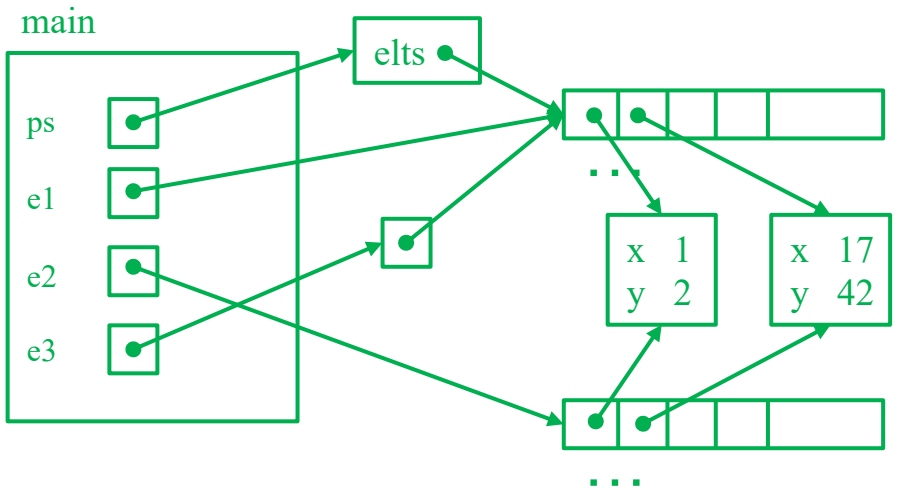
Agenda

- Rep-Exposure Exercise
- Graph concepts
- Testing in practice
  - Script testing
  - JUnit testing

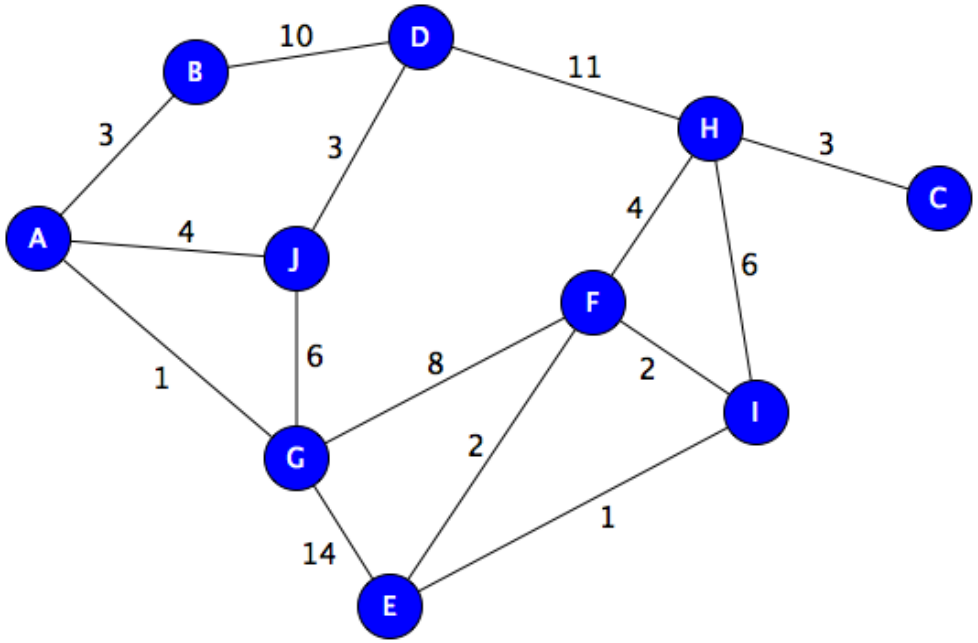
Rep-Exposure Exercise



Rep-Exposure Exercise (Solution)



Graphs



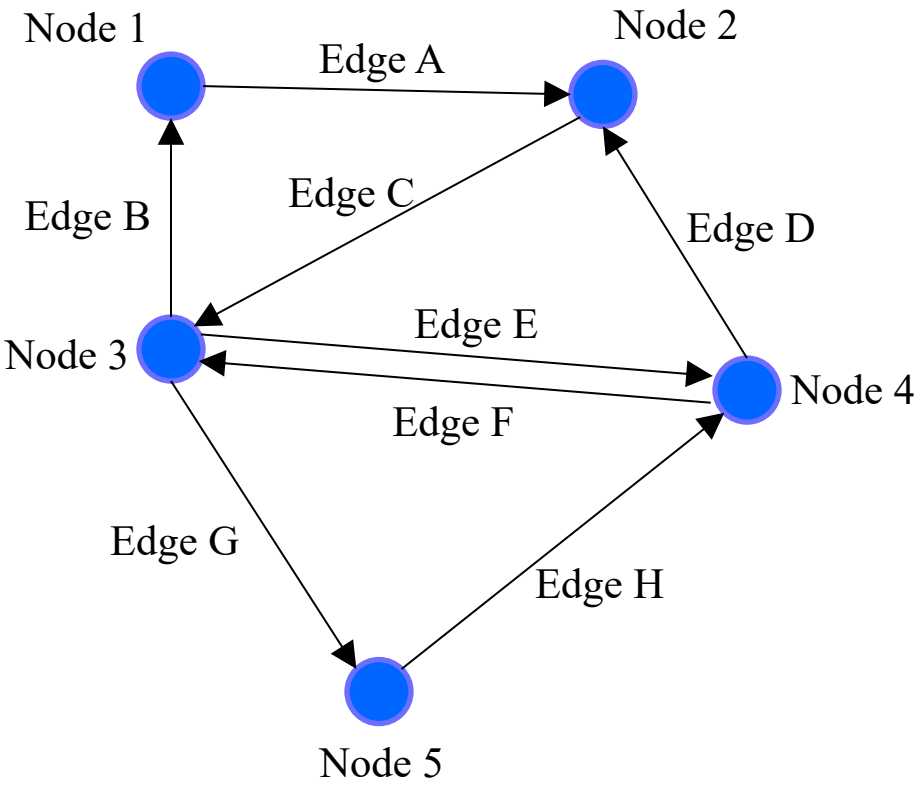
A graph represents relationships

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

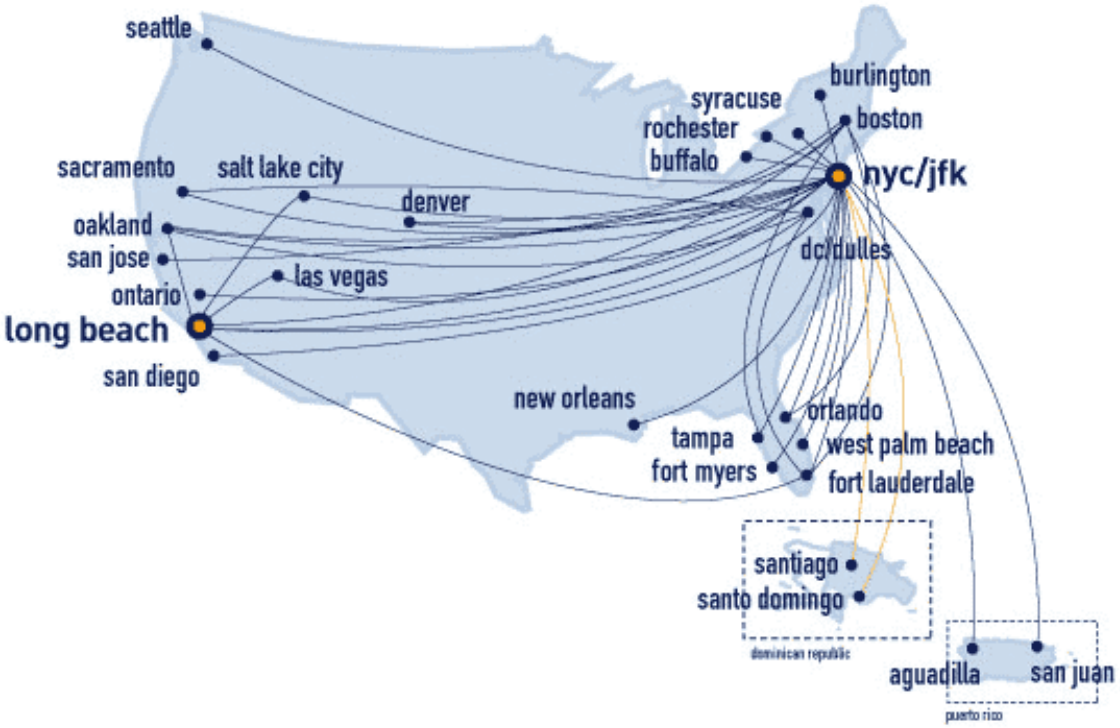
Nodes: intersections (cities)

Label: name/location

Edges: roads

Label: name/length

Example: airline flights



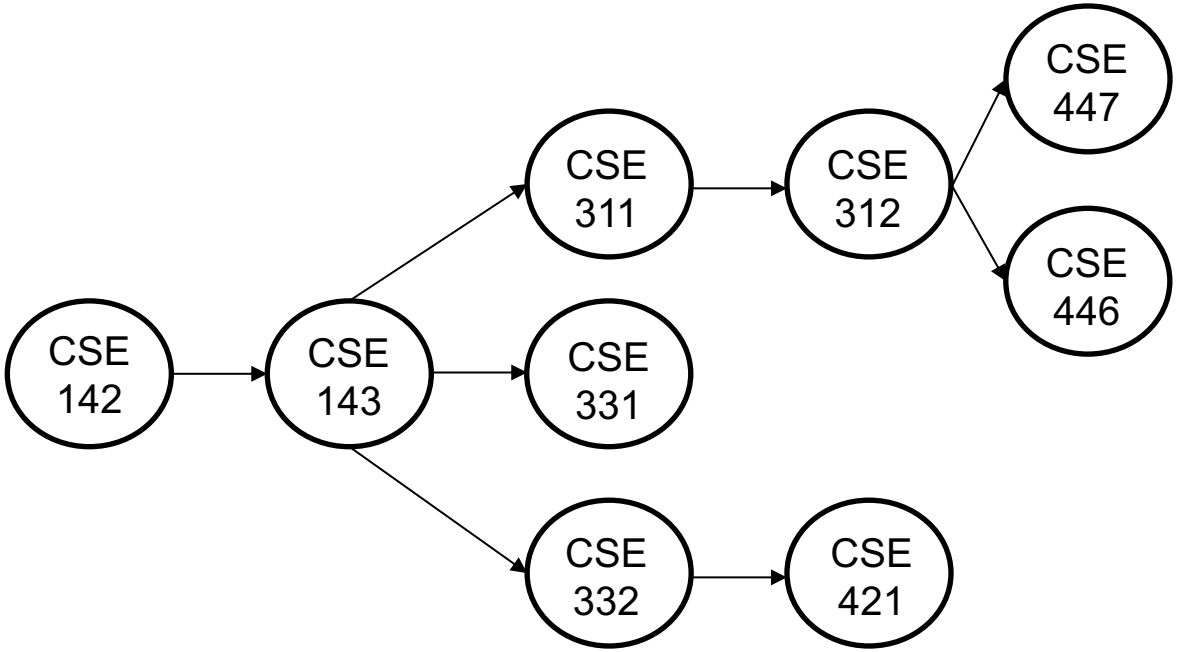
Nodes: airports

Label: airport code

Edges: flights

Label: cost/time

Example: CSE courses



Nodes: Courses

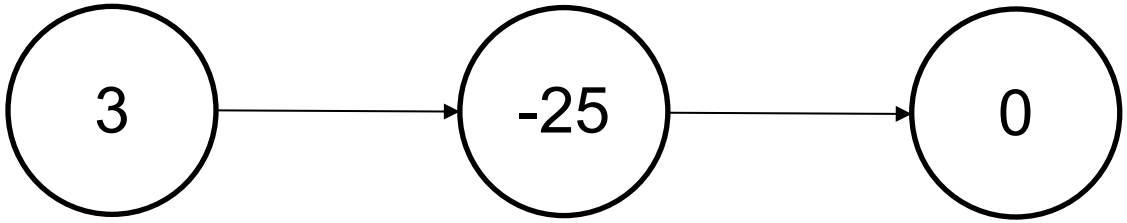
Label: Course name

Edges: pointer to next class

Label: none

You've used graphs before!

Singly linked Lists:

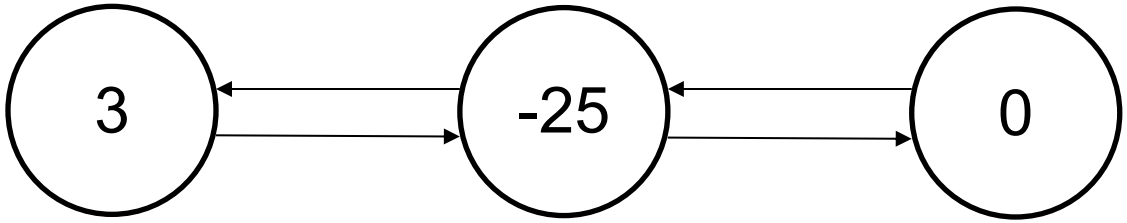


Nodes: Linked list node
Label: integer

Edges: pointer to next node
Label: none

You've used graphs before!

Doubly linked Lists:

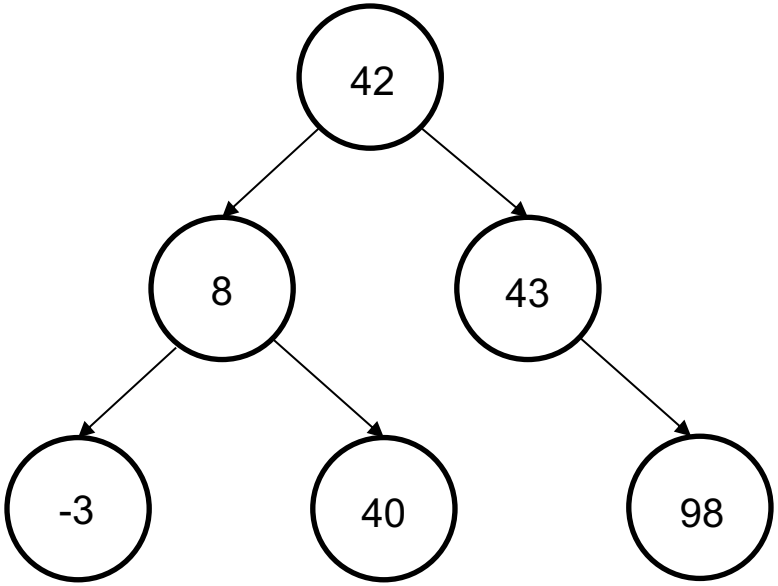


Nodes: Linked list node
Label: integer

Edges: pointers to prev/next nodes
Label: none

You've used graphs before!

Binary trees:



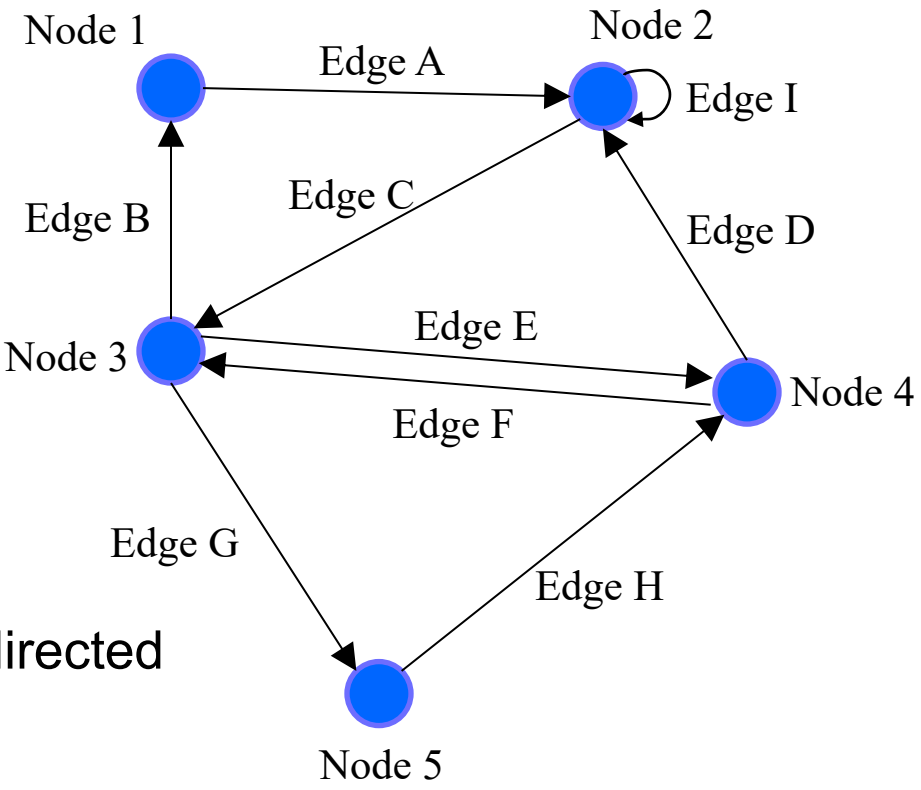
Nodes: Tree node
Label: Integer

Edges: pointers to children
Label: none

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.

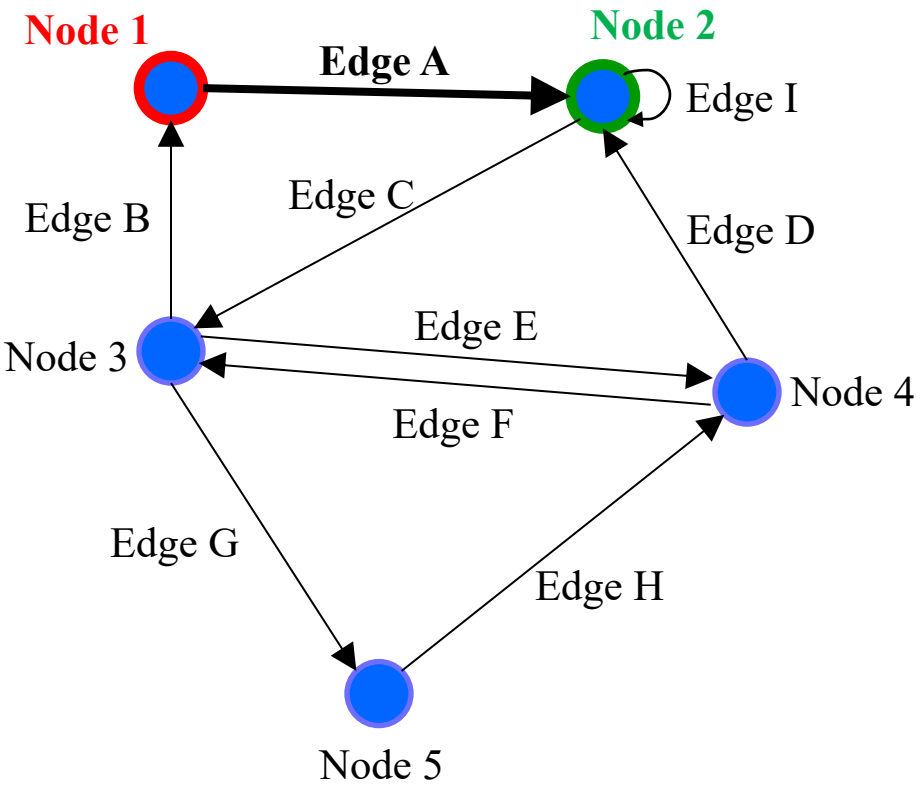
An edge points from source to dest.

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



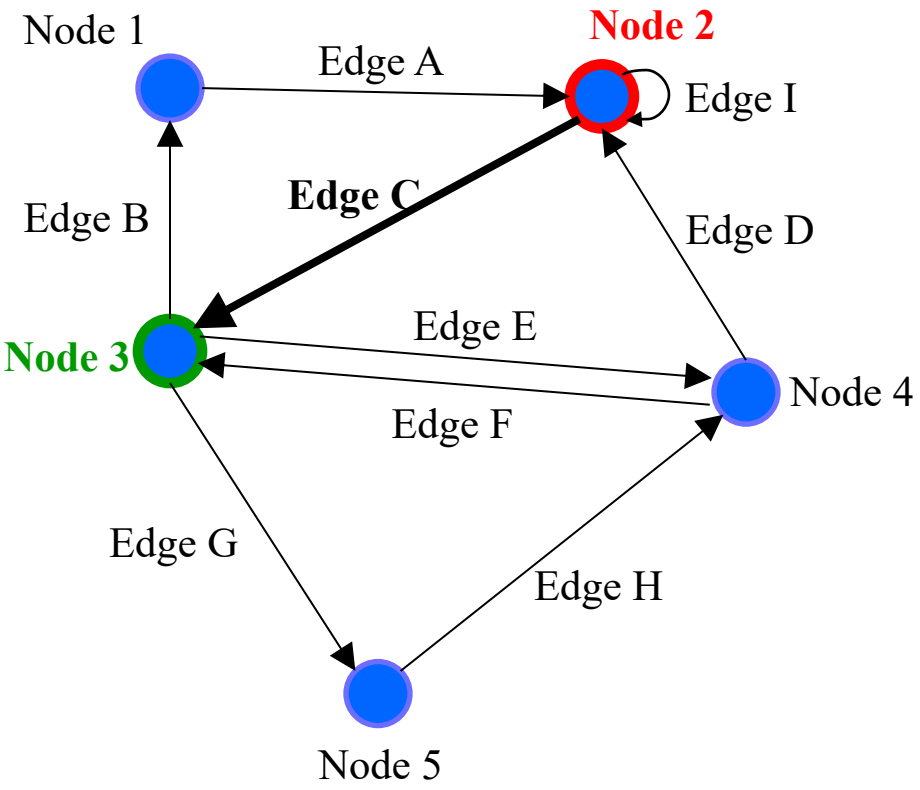
An edge points from source to dest.

Each edge "points" from a source to a destination.

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

Edge C is Node 2 → Node 3.

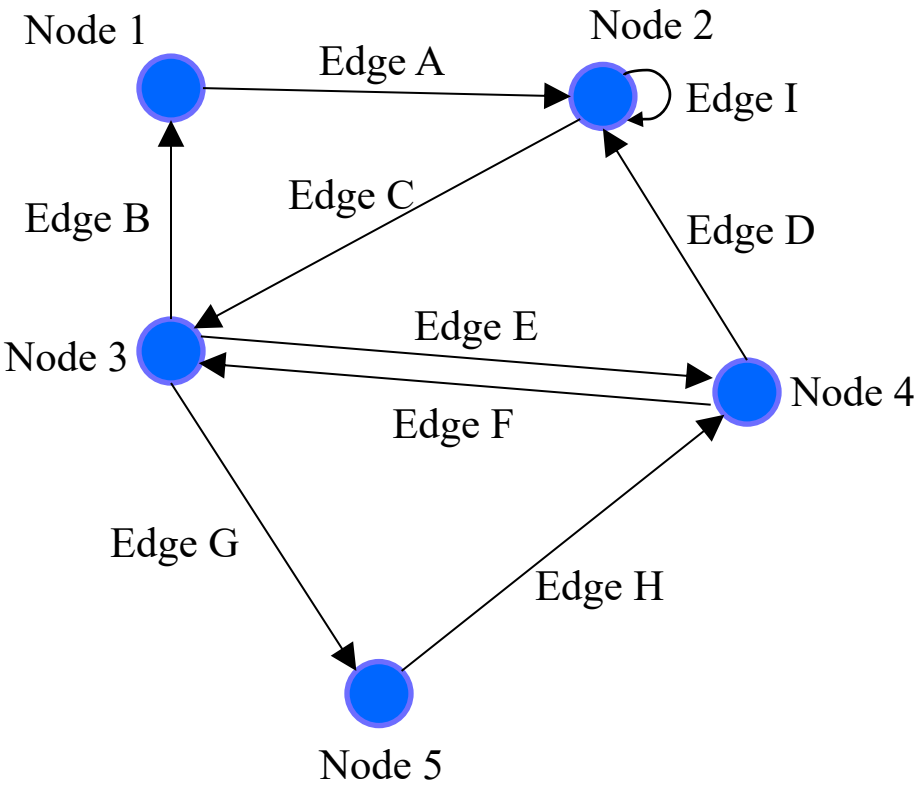
- Outgoing from Node 2
- Incoming to Node 3



A node has children

A node's outgoing edges point to its **children**.

- Potentially empty set



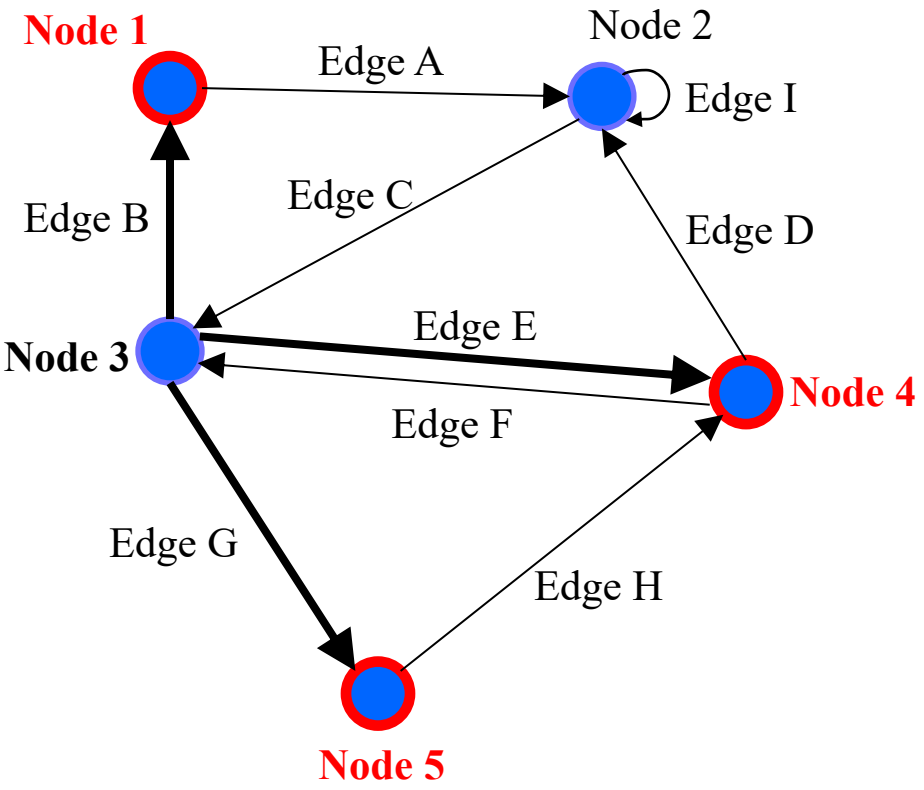
A node has children

A node's outgoing edges point to its **children**.

- Potentially empty set

Node 3 has three children:

- Node 1
- Node 4
- Node 5



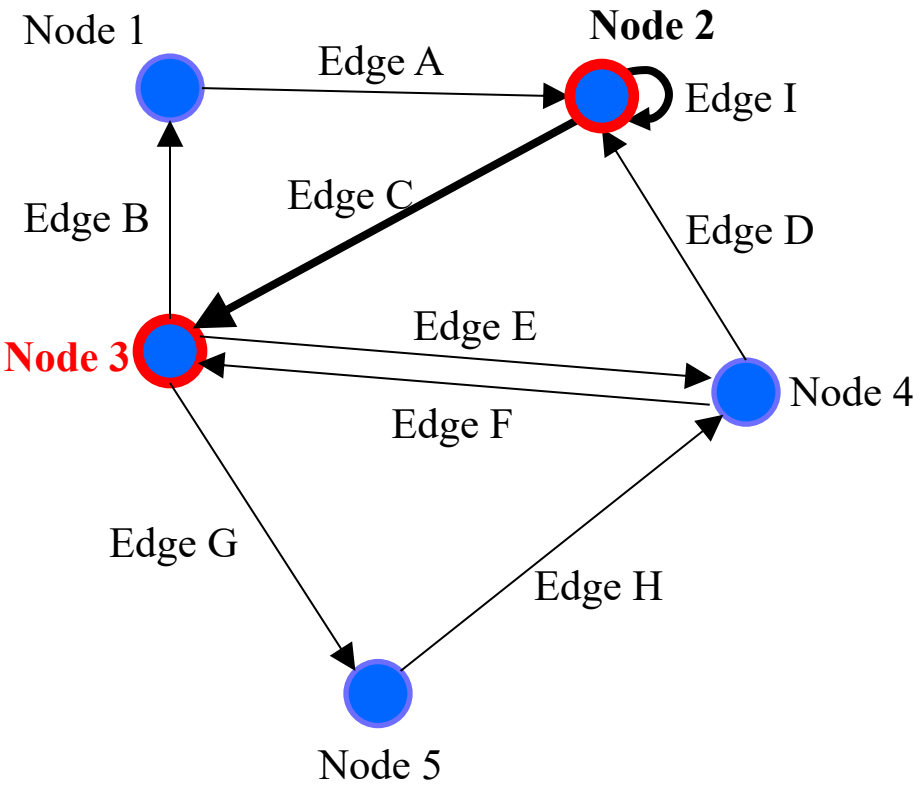
A node has children

A node's outgoing edges point to its **children**.

- Potentially empty set

Node 2 has two children:

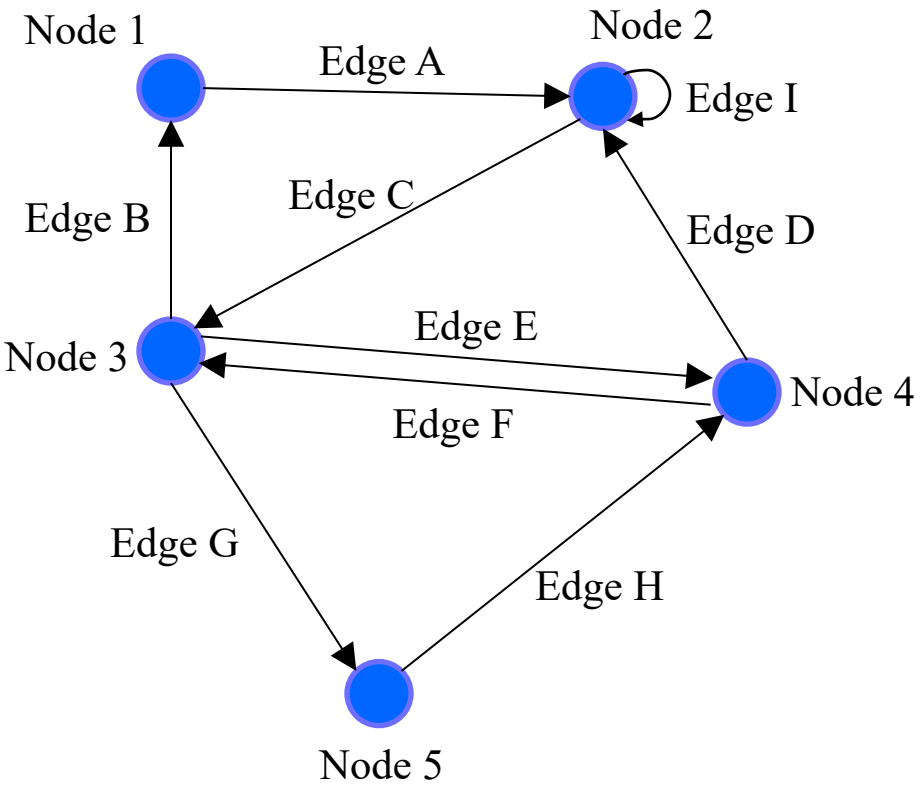
- Node 2
- Node 3



A node has parents

A node's incoming edges point from its **parents**.

- Potentially empty set



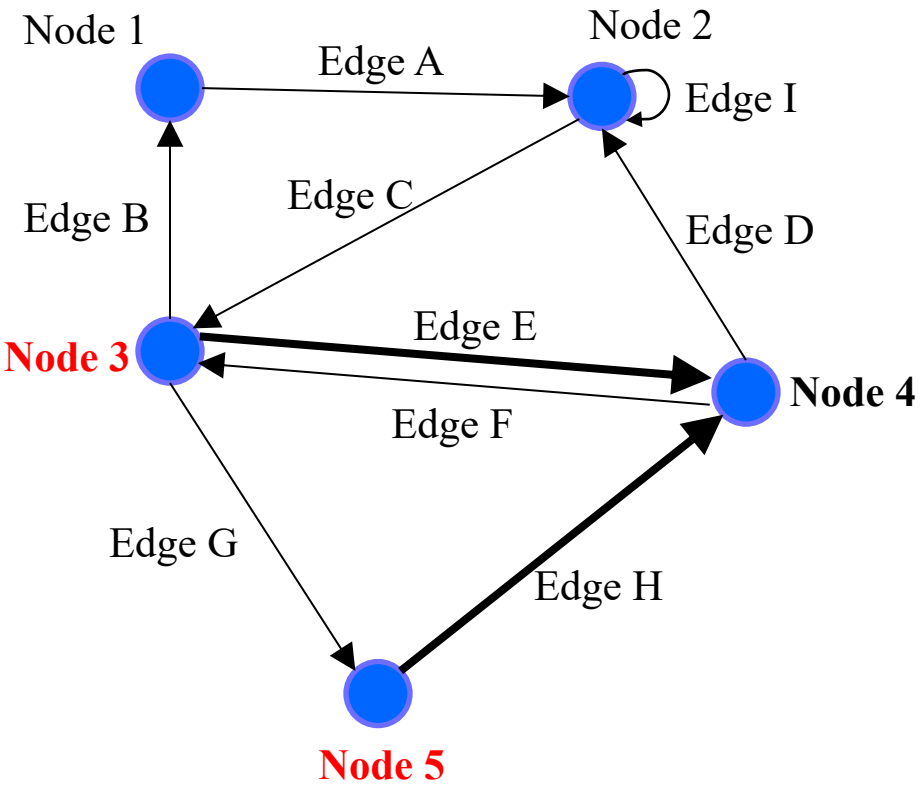
A node has parents

A node's incoming edges point from its **parents**.

- Potentially empty set

Node 4 has two parents:

- Node 3
- Node 5



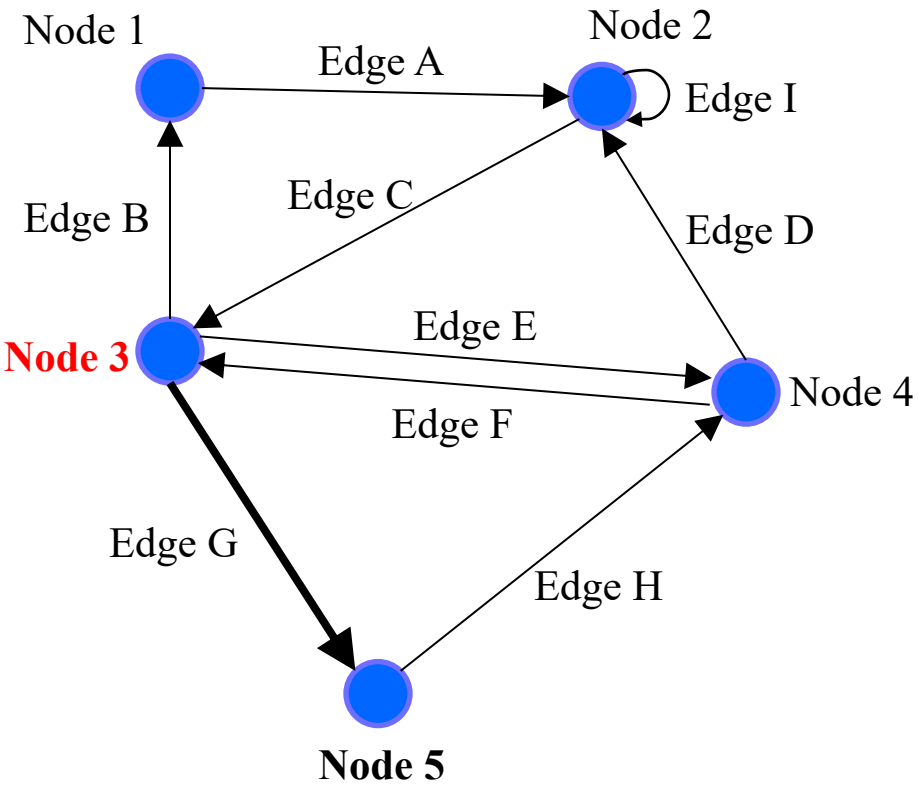
A node has parents

A node's incoming edges point from its **parents**.

- Potentially empty set

Node 5 has one parent:

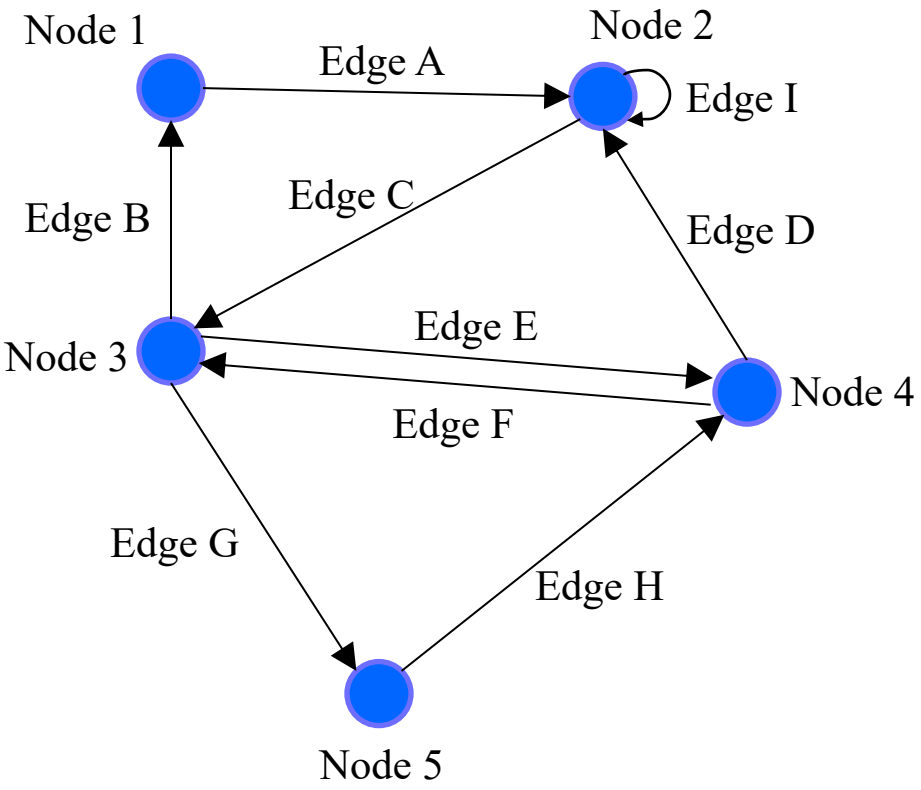
- Node 3



A node has neighbors

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

- Potentially empty set



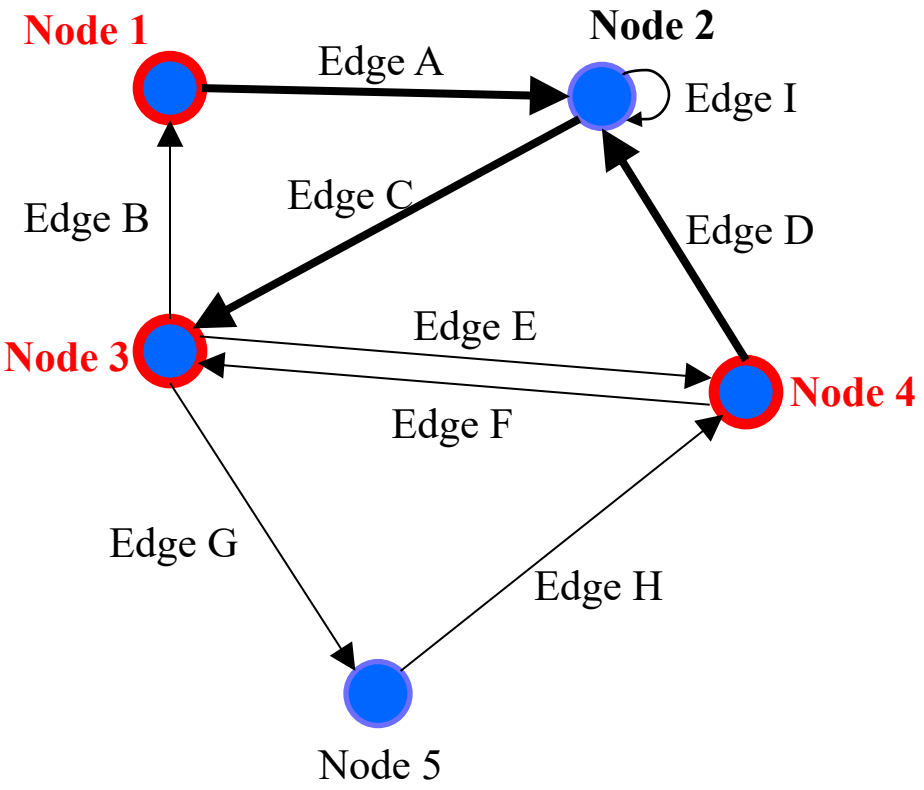
A node has neighbors

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)



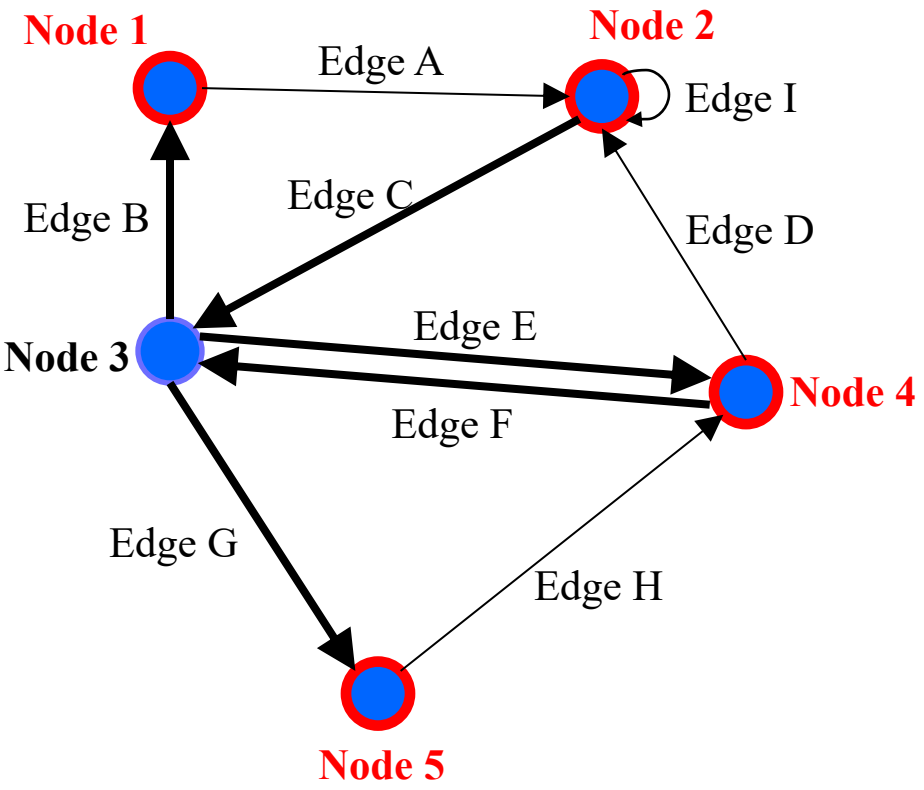
A node has neighbors

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)



Possible graph operations

Creators

- Construct an empty graph

You *might or might not* want to include all of these operations in your graph ADT design.

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

More observers

- Find path(s) from one node to another
- Find all reachable nodes
- Count indegree, outdegree

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
  - Advice: Look ahead at HW6 and consider its likely needs
  - Reminder: *Not a generic class.*

- HW5 split into 2 parts
  1. **Design and specify a graph ADT**
  2. Implement that ADT specification

HW5-1: What's Included

- Your submission for HW5-1 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 HW5-1 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…
- Make sure to look at your JavaDoc before submitting!

HW5: Testing

- How might you test a dishwasher?
  - Maybe we put in a dirty plate, run it, and we expect there to be a clean plate
  - Maybe we fill it with dirty silverware, run it, and the silverware should be clean
  - …

- Tests consist of some **input/setup**, and we compare the results to some **expected output**

HW5: Testing

- Now, how can we (the course staff) test your graph ADT?
  - Given that you will all have different designs with different methods

- To answer this, we must first answer: what does a graph need to be able to do?
  - We need the ability to add nodes and edges
  - And we need to be able to see what nodes exist and the relationships between those nodes
  - Otherwise, it wouldn't be a useful graph!

HW5: Testing

- For example: we want to test that if we get all the nodes in an empty graph we just created, we get zero nodes.

- How do we test this on **each** of your graphs?
  - We won't write new JUnit tests for each one of you!

- We need to define a test input and output format that is independent of your graph specification and implementation
  - We call these script tests!

- Ex:

Input	Output
`CreateGraph g`	`created graph g`
`ListNodes g`	`g contains:`

HW5: Script Tests

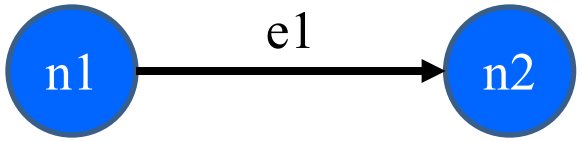
Each script test is expressed as text-based script file **`foo.test`**

- Sequence of commands, 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!

Command (in `foo.test`)	Output (in `foo.expected`)
`CreateGraph` *name*	**`created graph`** *name*
`AddNode` *graph label*	**`added node`** *label* **`to`** *graph*
`AddEdge` *graph parent child label*	**`added edge`** *label* **`from`** *parent* **`to`** *child* **`in`** *graph*
`ListNodes` *graph*	*graph* **`contains:`** $label_{node}$ …
`ListChildren` *graph parent*	**`the children of`** *parent* **`in`** *graph* **`are:`** *child* **`(`**$label_{edge}$**`)`** …
`#` *This is comment text …*	**`#`** *This is comment text …*

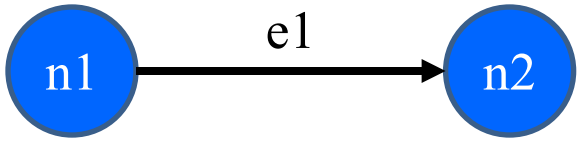
HW5: `example.test`

```
# 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
```



HW5: `example.expected`

```
# 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: 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`**
   - But only after the class is implemented and **`GraphTestDriver`** stubs filled in (in HW5 part 2)

HW5: More Script Tests

- How do these commands call methods in our graph class?
  - They don't! (for now…)
  - We'll deal with that in HW5 Part 2—don't worry for now.

- These script tests will work regardless of your graph design or implementation.

- Commands are **not** the same as the methods in your graph
  - 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!

HW5: `ListNodes` and `ListChildren`

- **`ListNodes`** and **`ListChildren`** are the only commands where the output depends on the state of your graph
  - The rest have output that repeats inputs (e.g. name of graph)

- Thus, every test should have either **`ListNodes`** or **`ListChildren`** to validate the graph state is as you expect.

- These two commands have output in a specific format and in sorted order
  - But your graph methods **should not** return things in this format or in sorted order
  - Instead, your methods should return the necessary information in unsorted collections (when implemented in HW5 part 2)

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`

```
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

1. Note: Your JUnit tests will fail in hw5 part 1, because you have not implemented the actual methods yet
   - The same goes for your script tests

2. You will do that in part 2

HW5: Testing Summary

- 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!

JUnit for test authors

The following slides are included for reference and add additional material that you'll need to write tests for HW 5.

Reminder: In CSE 331 this quarter, we're using Junit 4, not Junit 5. If you've used Junit 5 elsewhere, some details are different (especially tests for methods that should throw an exception)

Writing tests with JUnit

Annotate a method with `@Test` to flag it as a JUnit test

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

/** Unit tests for my Foo ADT implementation */
public class FooTests {
  @Test
  public void testBar() {
    ... /* use JUnit assertions in here */
  }
}
```

Common JUnit assertions

JUnit's documentation has a full list, but these are the most common assertions.

Assertion	Failure condition
`assertTrue(`*`test`*`)`	*`test`* `== false`
`assertFalse(`*`test`*`)`	*`test`* `== true`
`assertEquals(`*`expected, actual`*`)`	*`expected`* and *`actual`* are not equal
`assertSame(`*`expected, actual`*`)`	*`expected`* `!=` *`actual`*
`assertNotSame(`*`expected, actual`*`)`	*`expected`* `==` *`actual`*
`assertNull(`*`value`*`)`	*`value`* `!= null`
`assertNotNull(`*`value`*`)`	*`value`* `== null`

Any JUnit assertion can also take a string to show in case of failure, *e.g.*, `assertEquals("helpful message", expected, actual)`.

Always* use >= 1 JUnit Assertion

- If you don't use any JUnit assertions, you are only checking that no exception/error occurs

- That's a pretty weak notion of passing a test; rarely the best test you could write

- Having more than one JUnit assertion in a test may make sense, but one is the most common scenario
  - "Each test should test one (new) thing" (most of the time)

* = Special-case coming in a couple slides ☺️

JUnit assertions vs Java's assert

- Use JUnit assertions **only in JUnit test code**
  - JUnit assertions have names like **`assertEquals`**, **`assertNotNull`**, **`assertTrue`**
  - Part of JUnit framework used to report test results
    - Accessed via **`import org.junit`**….
  - **Don't** use in ordinary Java code (***never*** **`import org.junit`**…. in non-JUnit code)

- Use Java's **`assert`** statement in ordinary Java code
  - Use liberally to annotate/check "must be true" / "must not happen" / etc. conditions
  - Use in **`checkRep()`** to detect failure if problem(s) found
  - **Do not** use in JUnit tests to check test result – does not interact properly with JUnit framework to report results

Checking for a thrown exception

- Need to test that your code throws exceptions as specified

- This kind of test method fails if its body does *not* throw an exception of the named class
  - May not need any JUnit assertions inside the test method

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

Test ordering, setup, clean-up

JUnit does not promise to run tests in any particular order.

However, JUnit can run helper methods for common setup/cleanup

- Run before/after *each* test method in the class:

```
@Before
public void m() { ... }
@After
public void m() { ... }
```

- Run once before/after *all* test methods in the class:

```
@BeforeClass
public static void m() { ... }
@AfterClass
public static void m() { ... }
```

Tips for effective testing

- Use constants instead of hard-coded values
  - Makes change easier later on
- Take advantage of assertion messages
- Give a descriptive name to each unit test (method)
  - Verbose but clear is better than short and inscrutable
  - Don't go overboard, though :-)
- Write tests with a simple structure
  - Isolate bugs one at a time with successive assertions
  - Helps avoid bugs in your tests too!
- Aim for thorough test coverage
  - Big/small inputs, common/edge cases, exceptions, ...

Test Design Worksheet

- Work in small groups
- Give logic of the tests, not actual code
- Only test the operations provided on the worksheet
- More details in lecture if additional information/review needed