
CSE 331

Software Design & Implementation

Winter 2020

Section 4 – Graphs, Testing

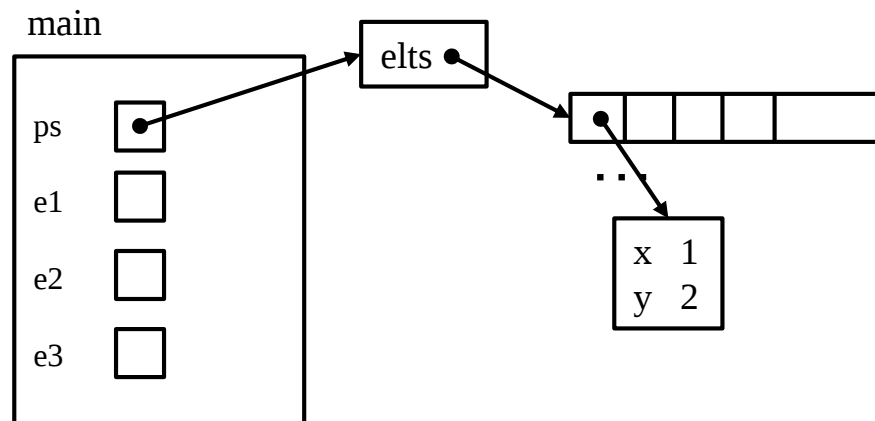
Administrivia

- Done with HW3!
- HW4 due today!
- HW5-1 and HW5-2 Spec out on the website
 - Always plan for work taking 3x longer than expected, so start early!
- Any questions?

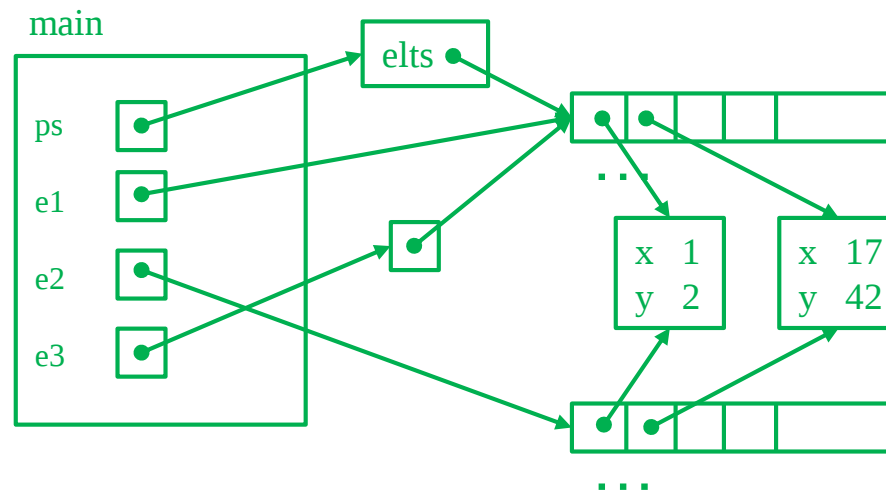
Agenda

- Rep-Exposure Exercise
- Graph concepts
- Testing in practice
 - Specification testing
 - Implementation testing

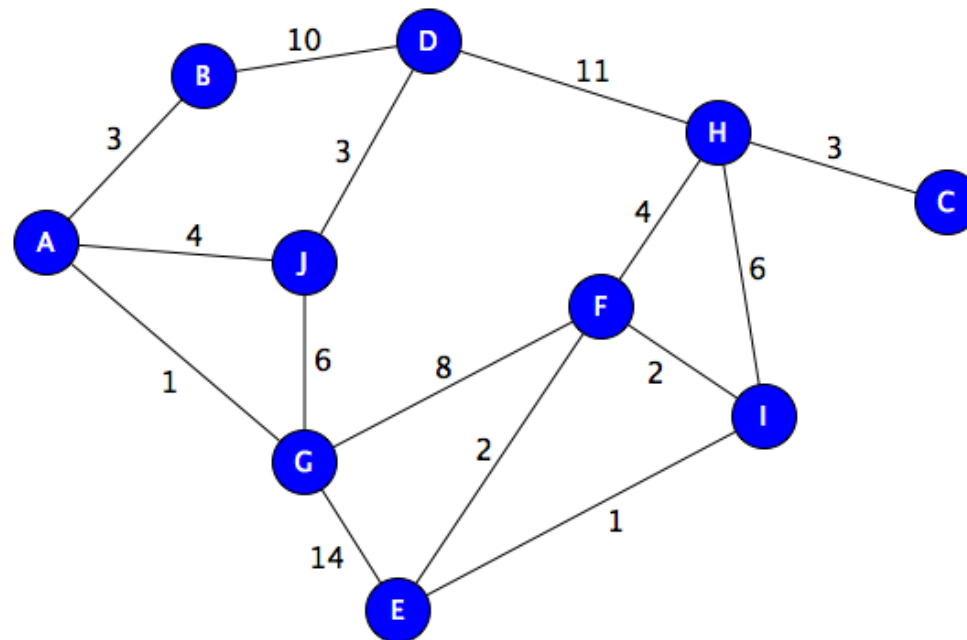
Rep-Exposure Exercise



Rep-Exposure Exercise (Solution)



Graphs



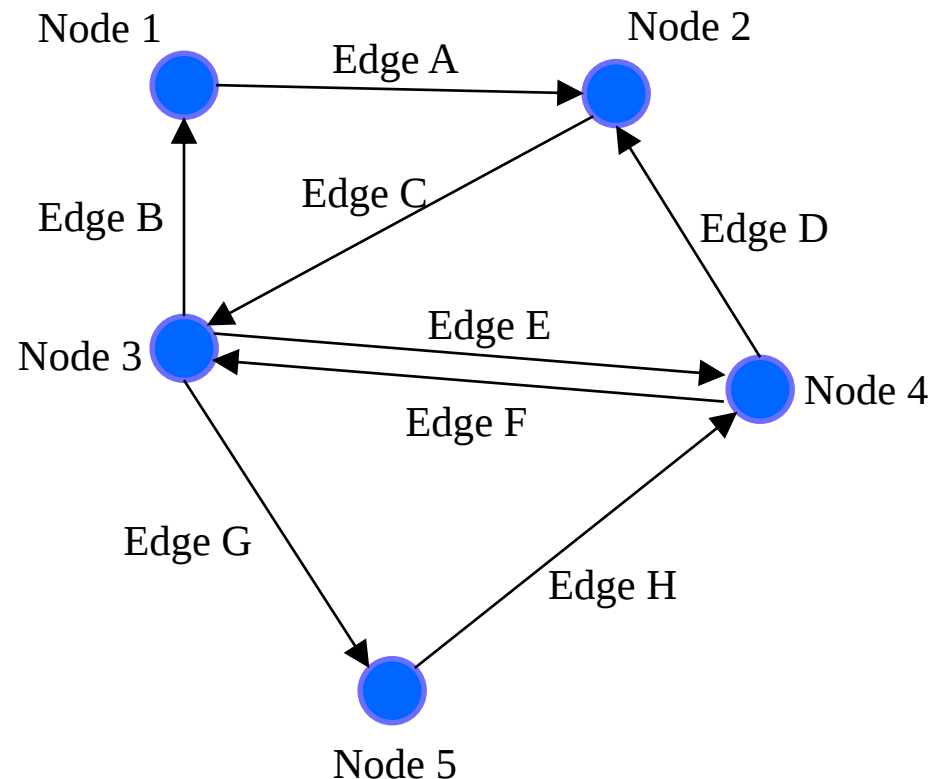
A graph represents relationships

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



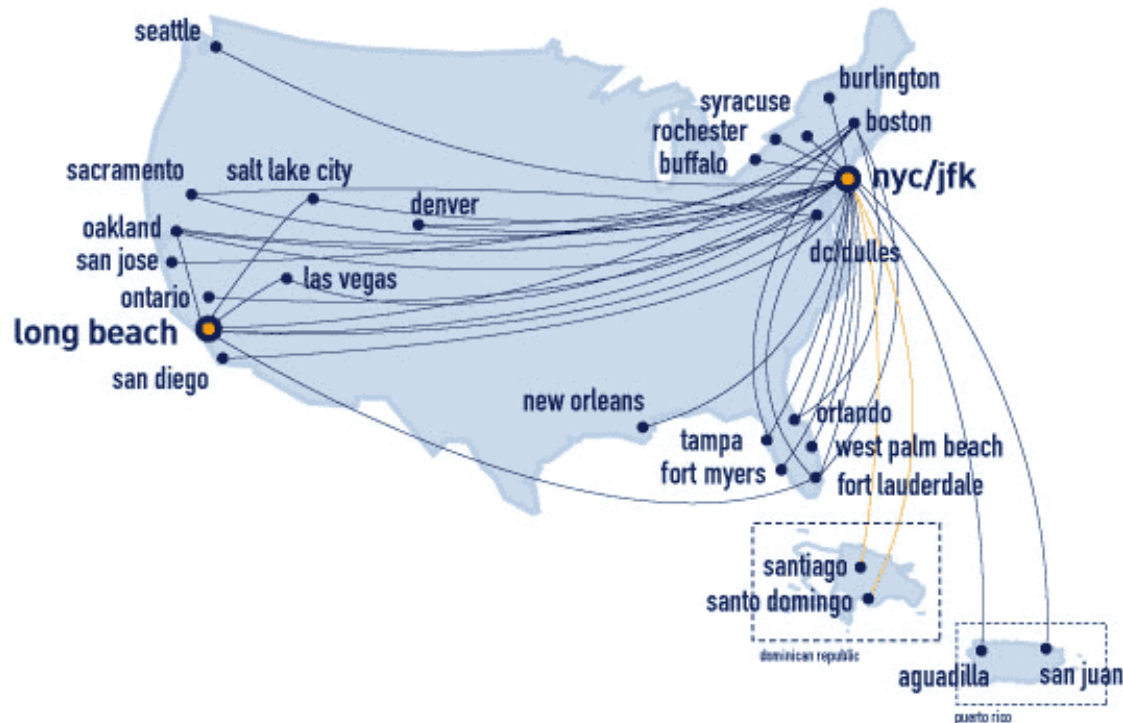
Nodes: intersections (cities)

Label: name/location

Edges: roads

Label: name/length

Example: airline flights



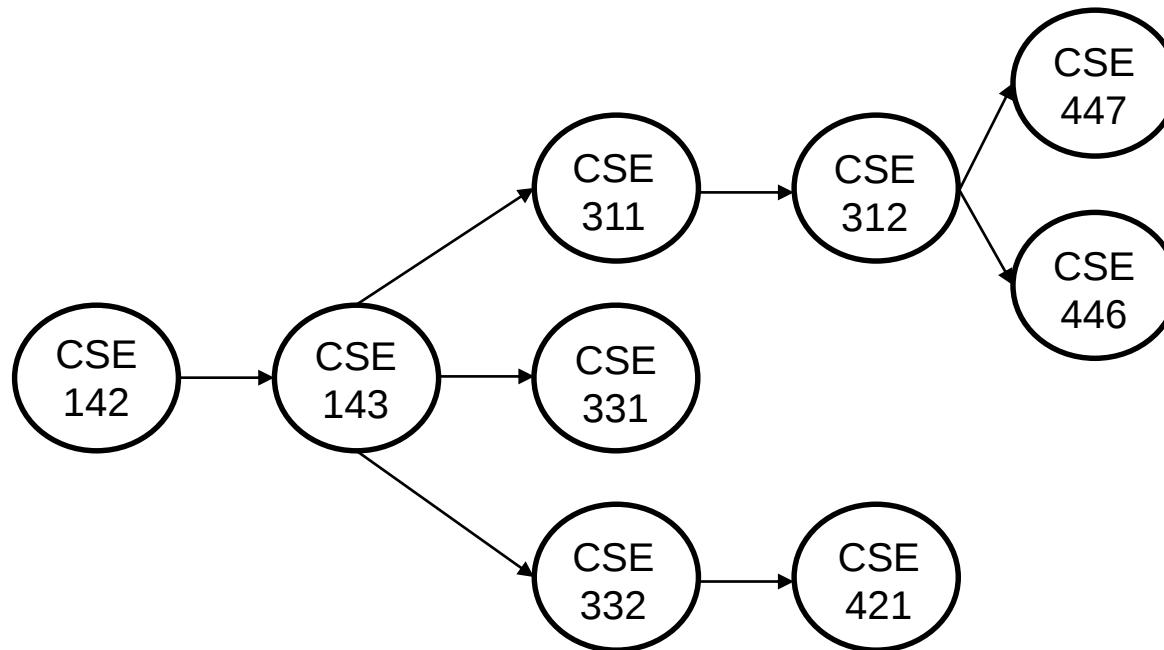
Nodes: airports

Label: airport code

Edges: flights

Label: cost/time

Example: CSE courses



Nodes: Courses

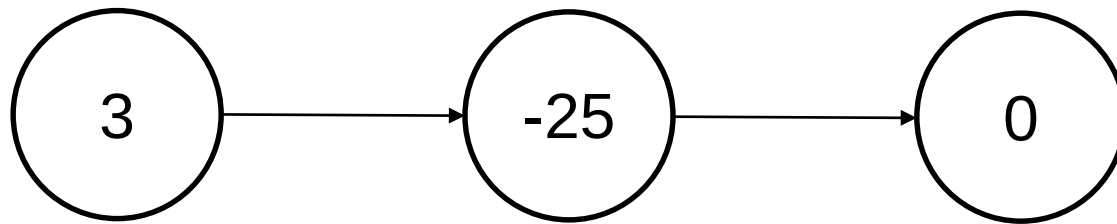
Label: Course name

Edges: pointer to next class

Label: none

You've used graphs before!

Singly linked Lists:



Nodes: Linked list node

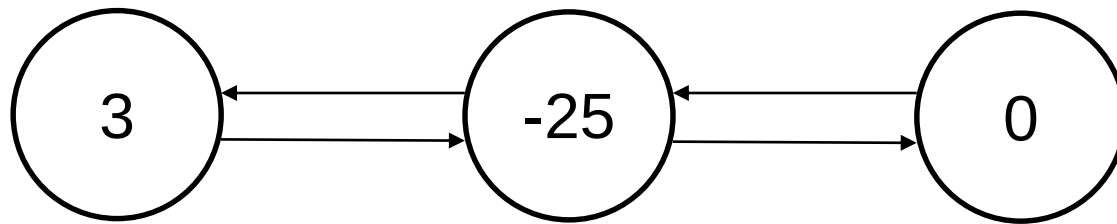
Label: integer

Edges: pointer to next node

Label: none

You've used graphs before!

Doubly linked Lists:



Nodes: Linked list node

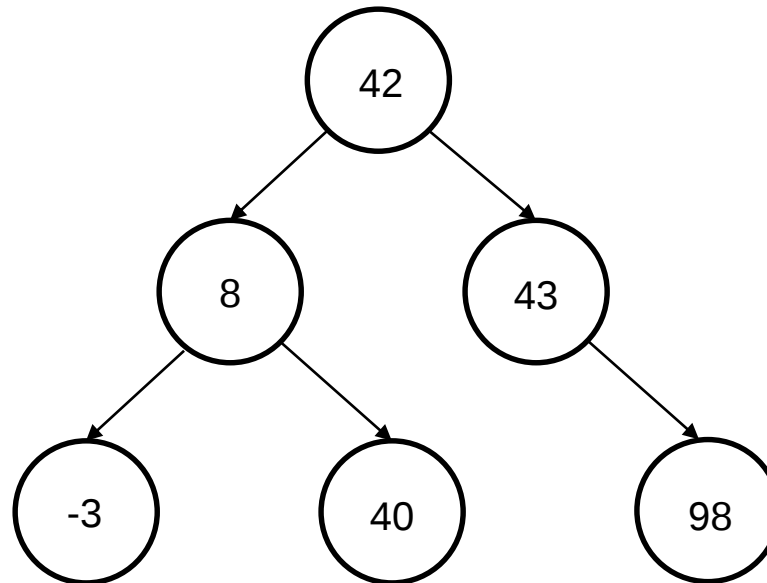
Label: integer

Edges: pointers to prev/next nodes

Label: none

You've used graphs before!

Binary trees:



Nodes: Tree node

Label: Integer

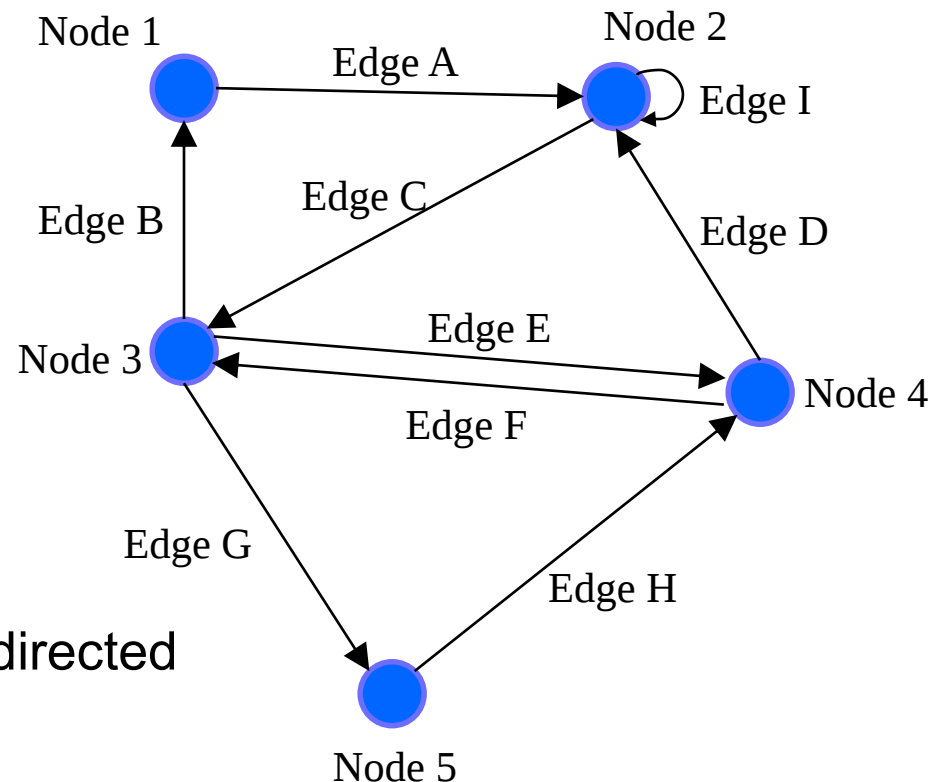
Edges: pointers to children

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.

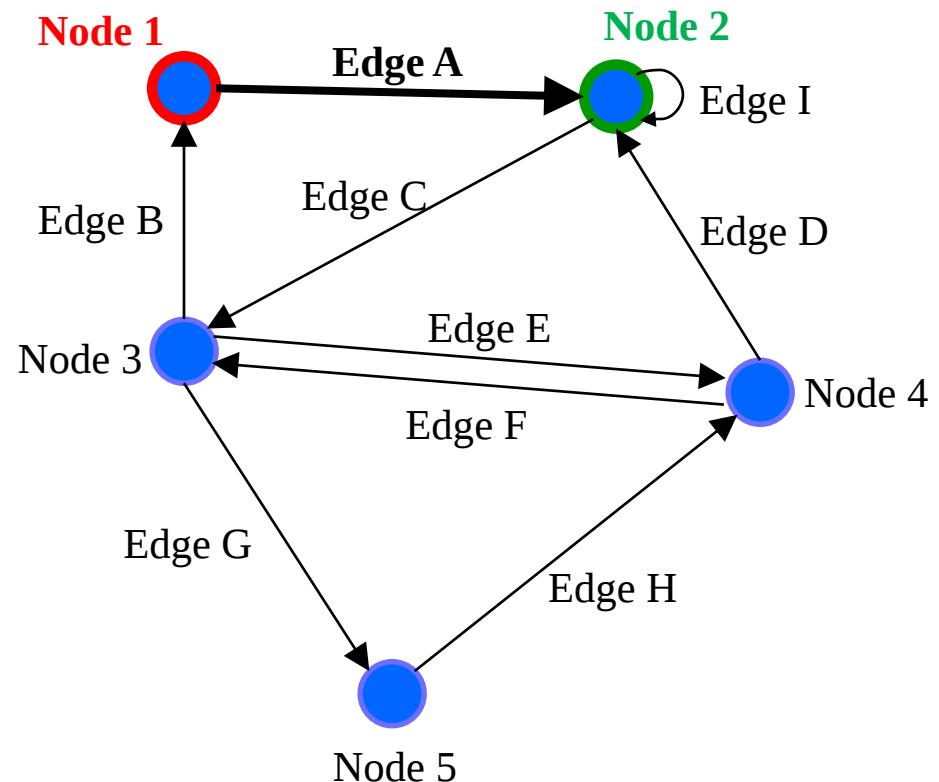
An edge points from source to dest.

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**



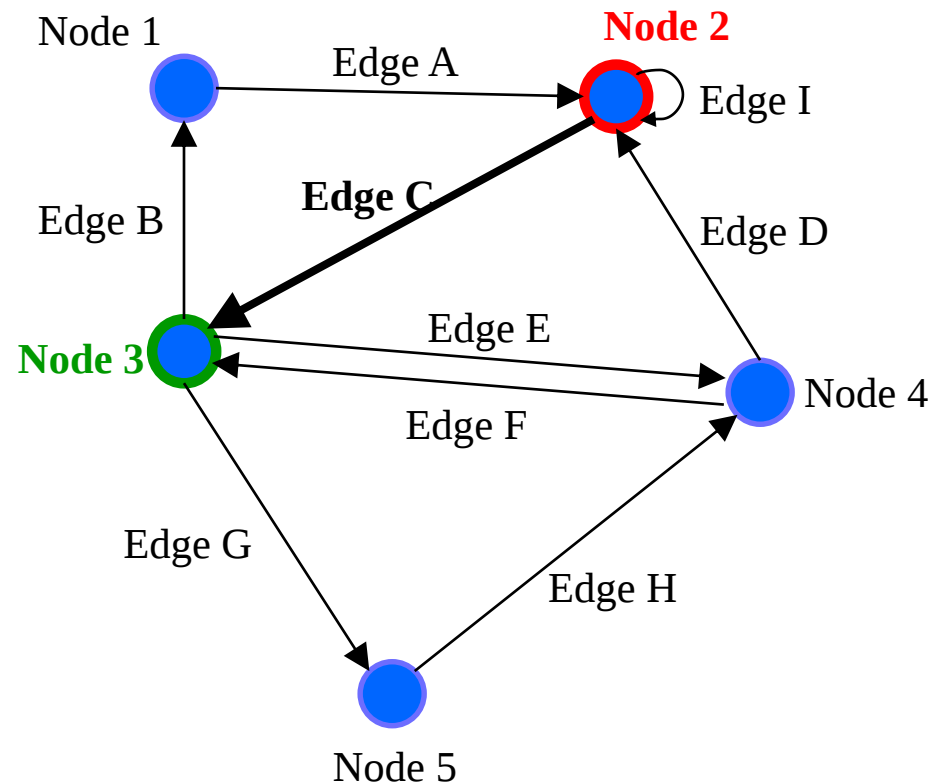
An edge points from source to dest.

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

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

Edge C is **Node 2** → **Node 3**.

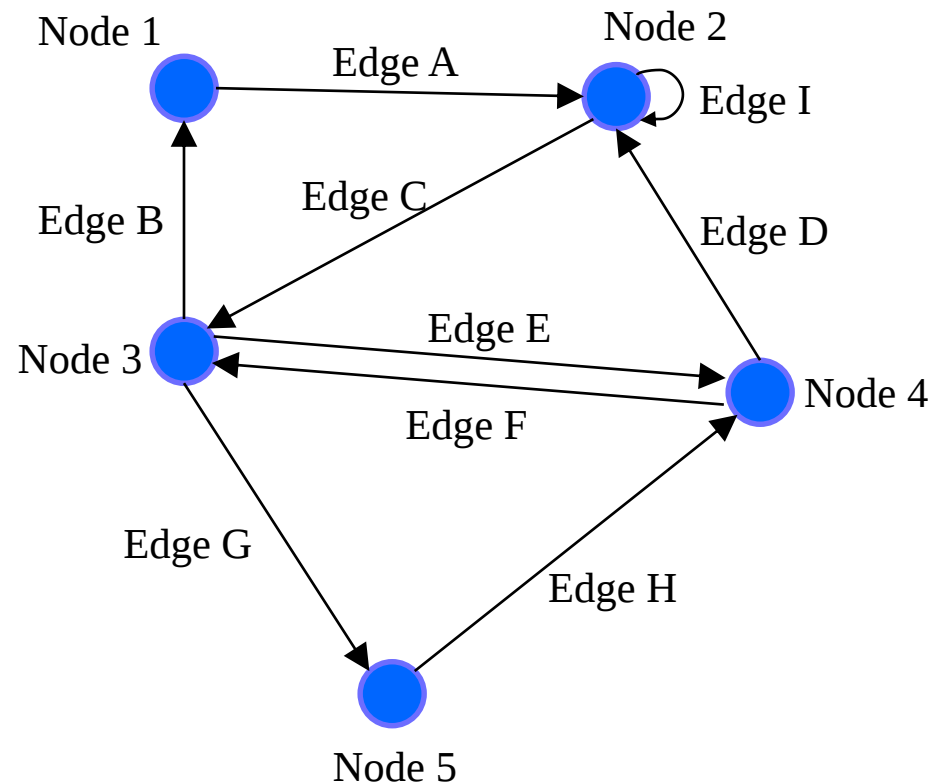
- Outgoing from **Node 2**
- Incoming to **Node 3**



A node has children

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

- Potentially empty set



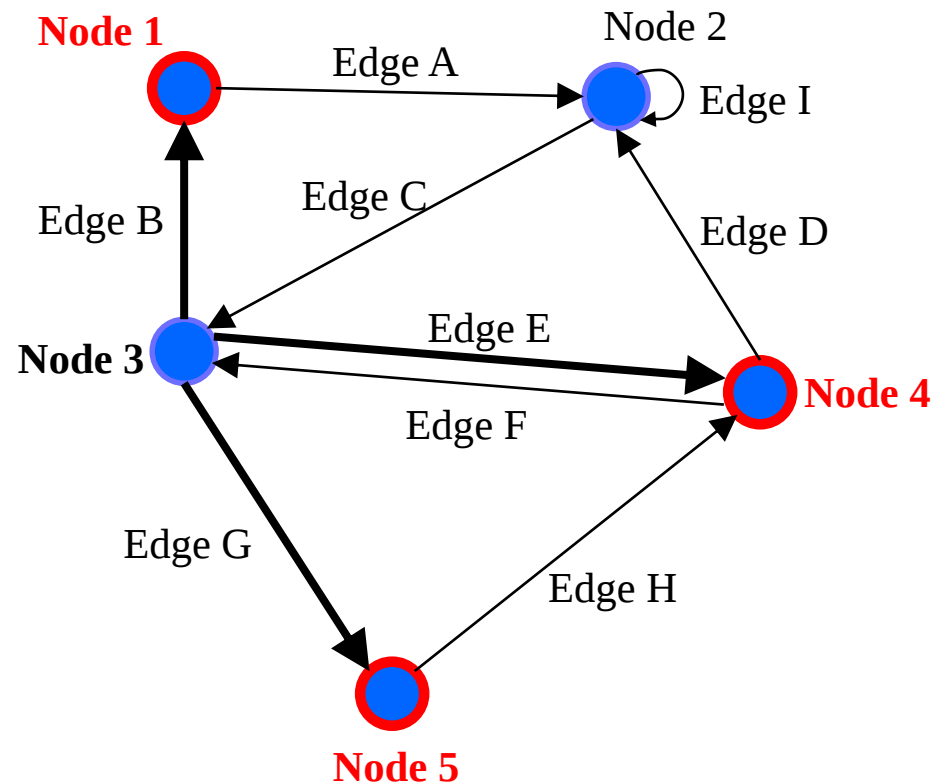
A node has children

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

- Potentially empty set

Node 3 has three children:

- Node 1
- Node 4
- Node 5



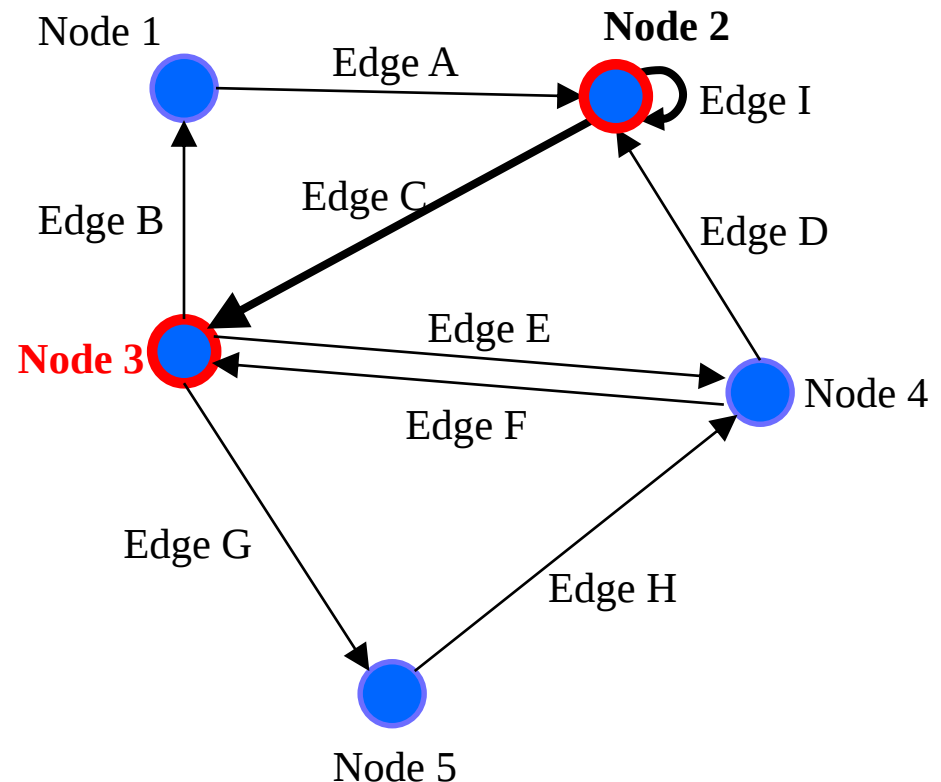
A node has children

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

- Potentially empty set

Node 2 has two children:

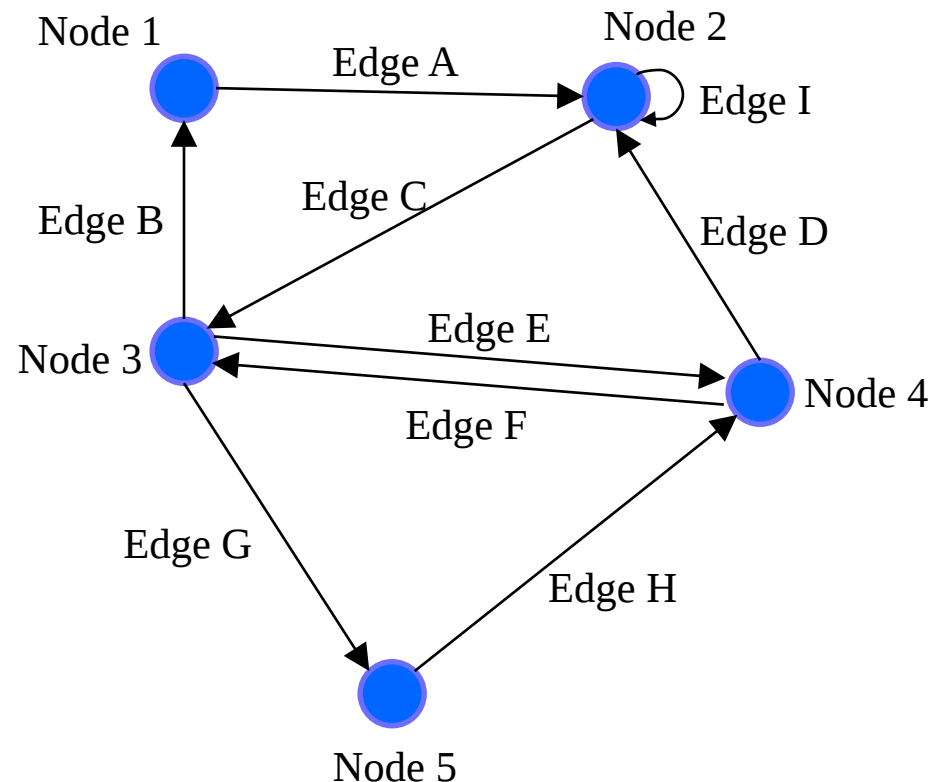
- Node 2
- Node 3



A node has parents

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

- Potentially empty set



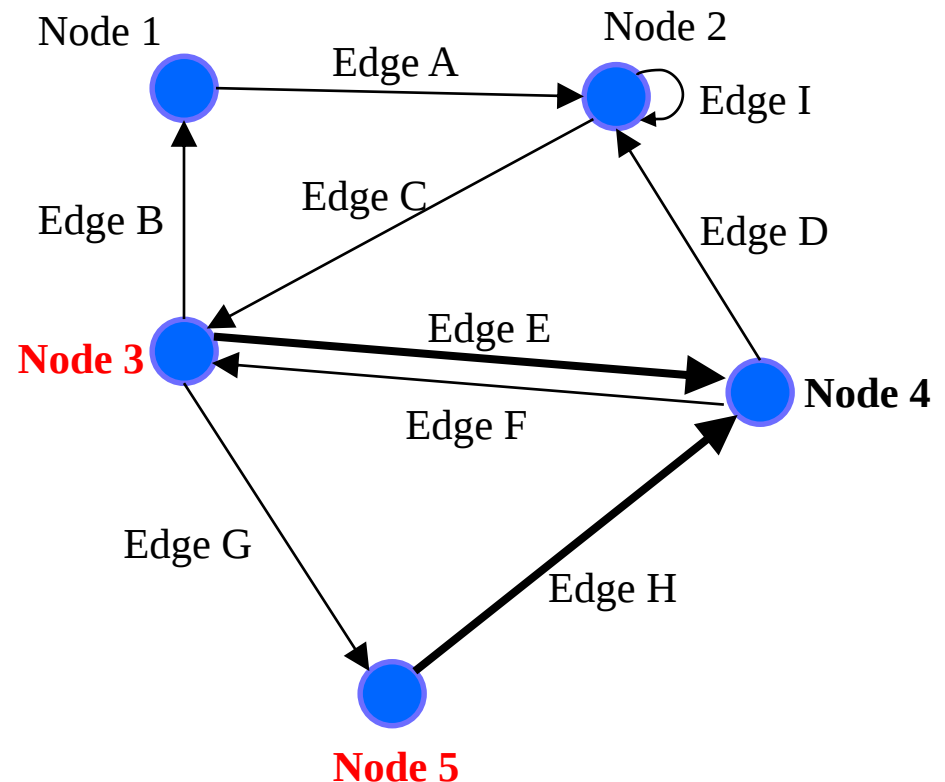
A node has parents

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

- Potentially empty set

Node 4 has two parents:

- **Node 3**
- **Node 5**



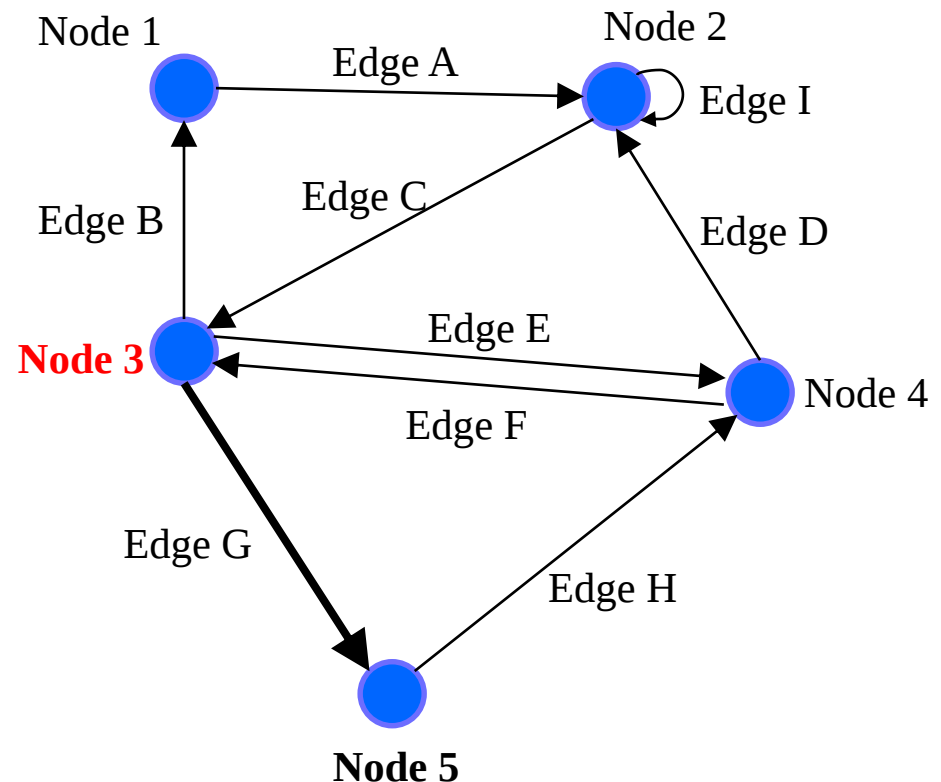
A node has parents

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

- Potentially empty set

Node 5 has one parent:

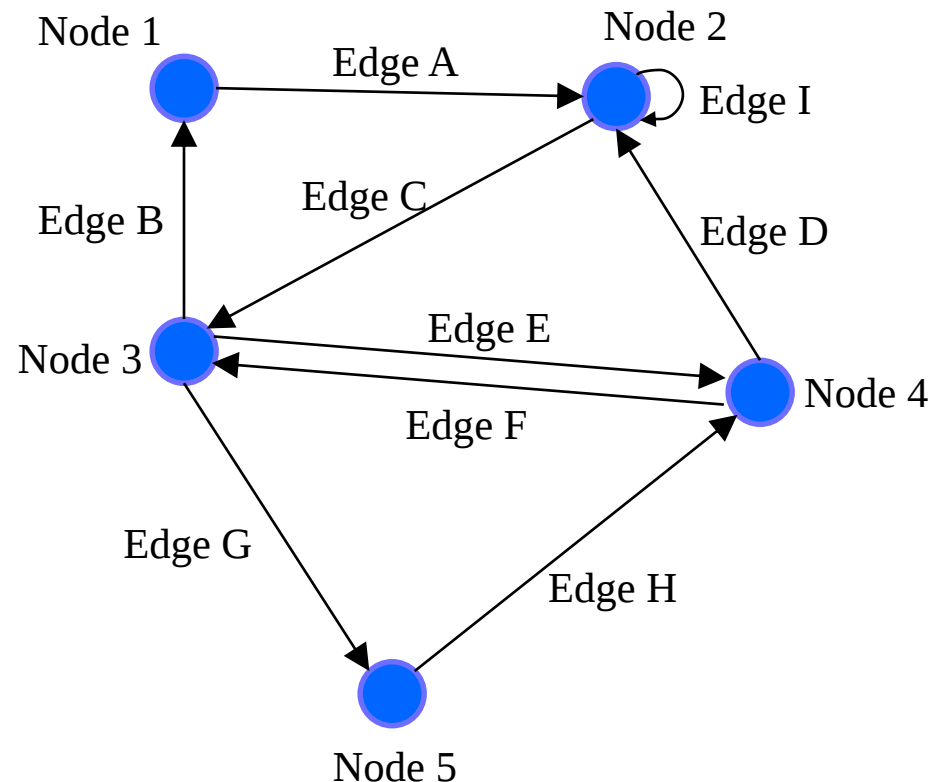
- **Node 3**



A node has neighbors

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

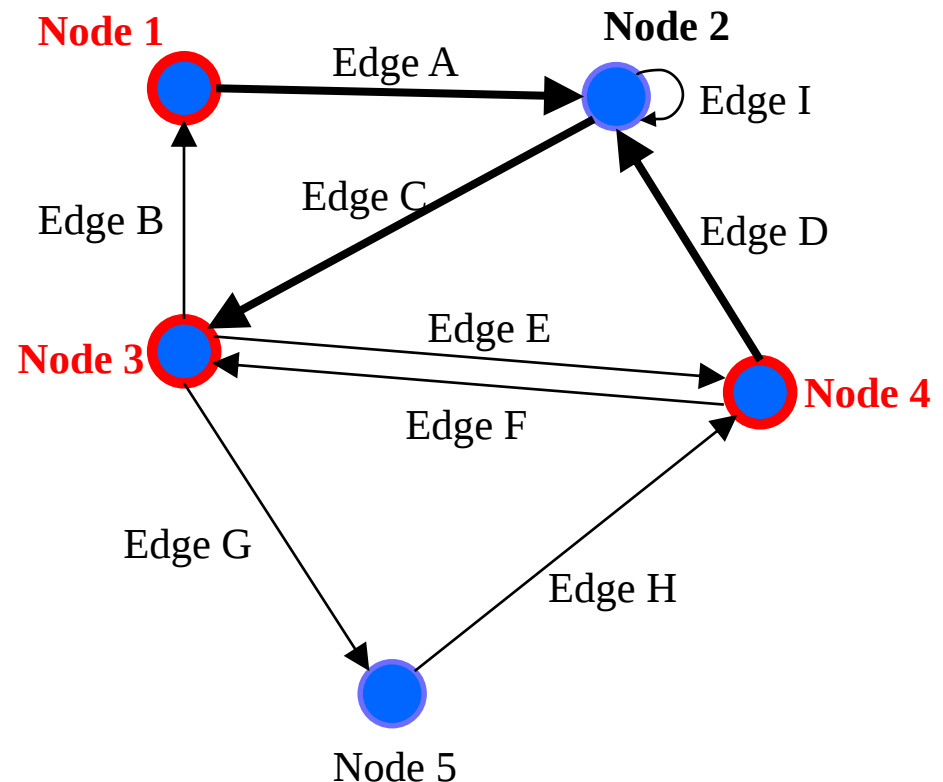
- Potentially empty set



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)

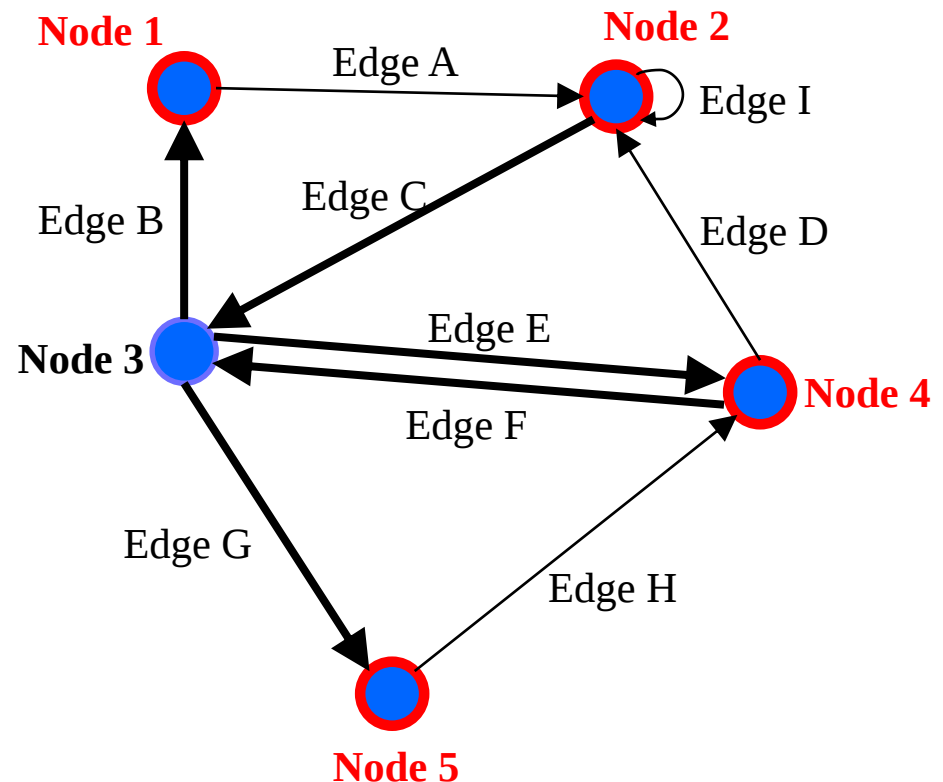
A node has neighbors

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

- Potentially empty set

Node 3 has four neighbors:

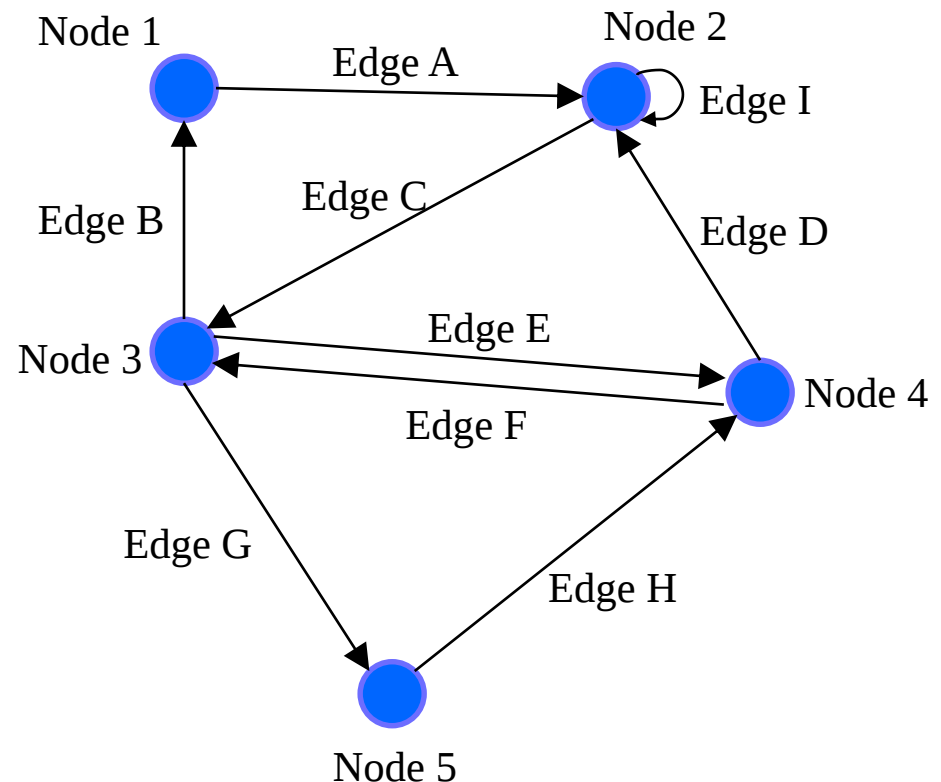
- **Node 1** (child)
- **Node 2** (parent)
- **Node 4** (parent *and* child)
- **Node 5** (child)



Paths between nodes

A **path** is a “chain” of edges from a **source** to a **destination**.

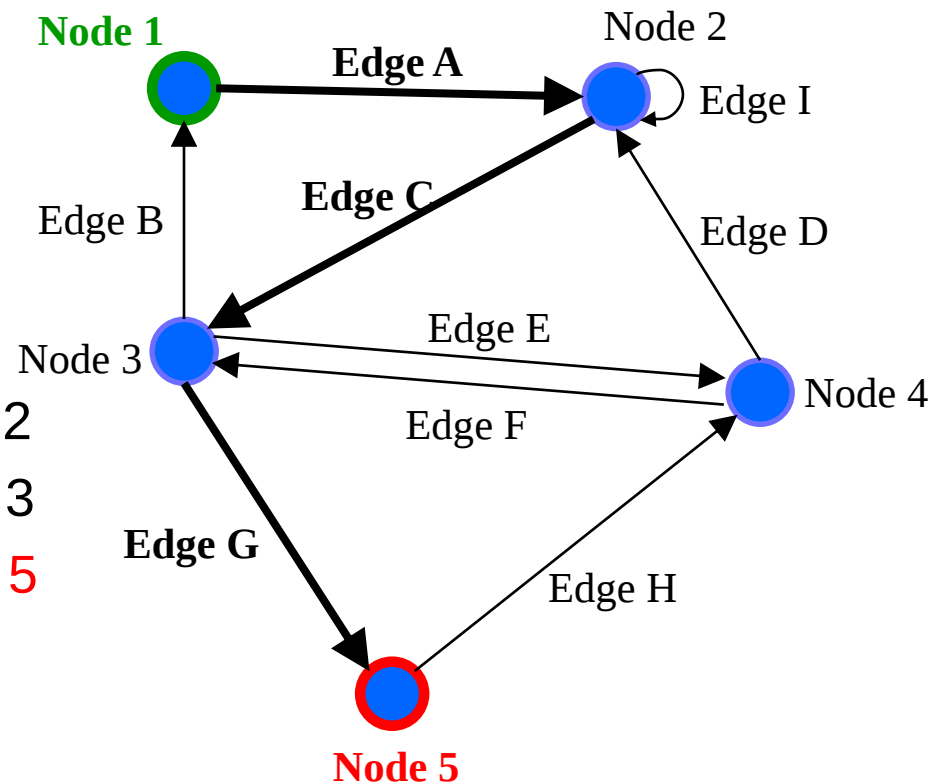
- Potentially empty sequence
- Might include a cycle
- Often want shortest



Paths between nodes

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 G : Node 3 → **Node 5**

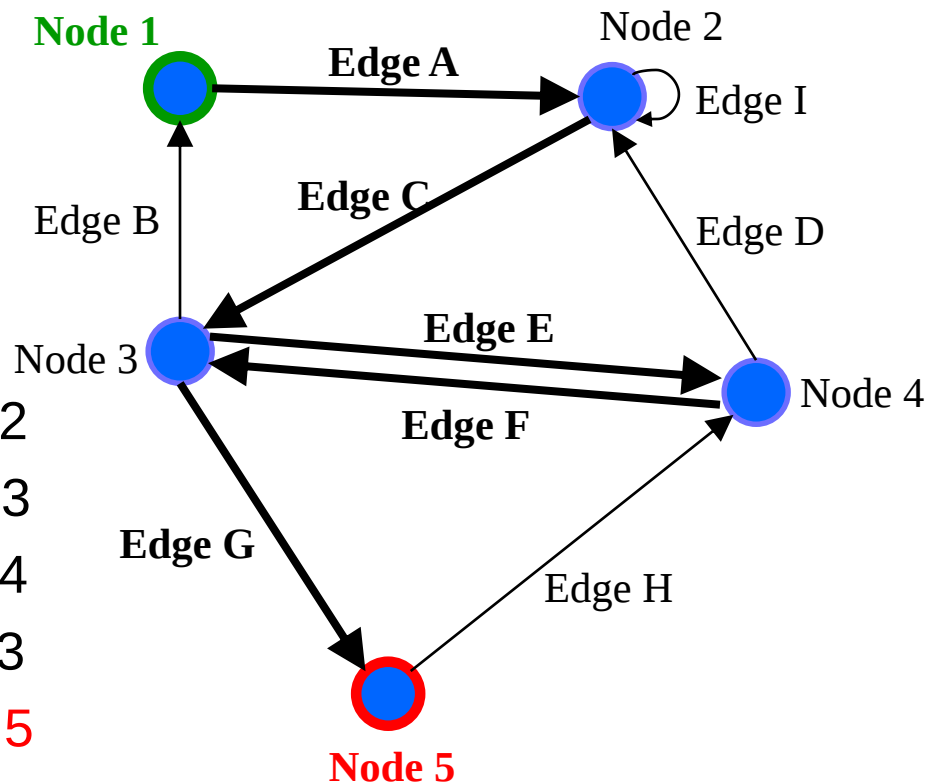
Paths between nodes

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**:

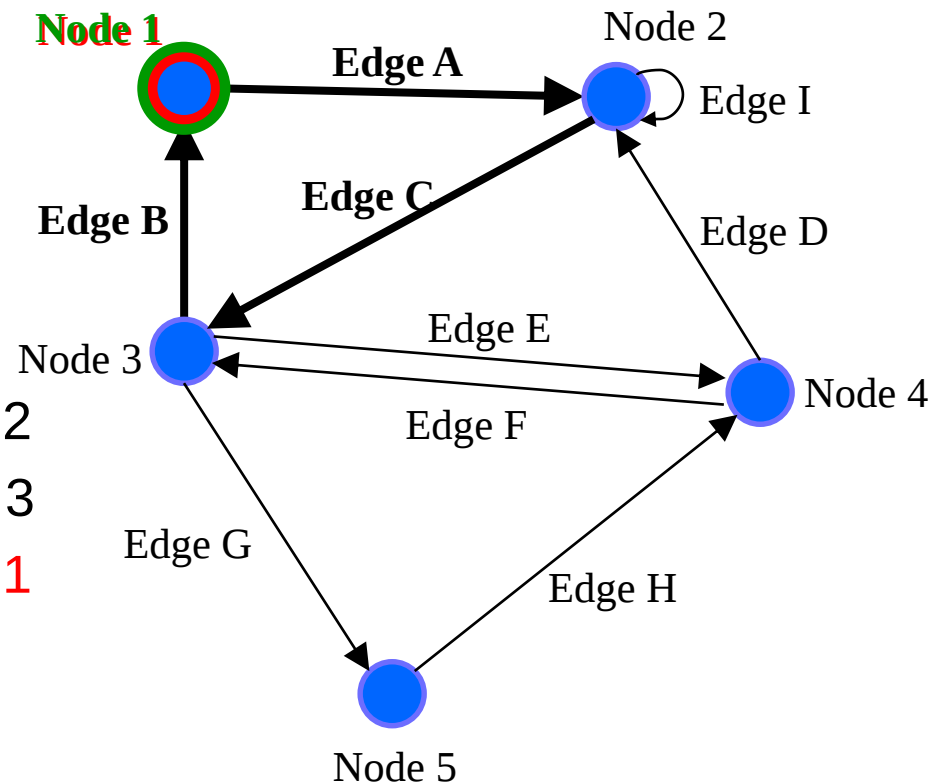
1. Edge A : **Node 1** → Node 2
2. Edge C : Node 2 → Node 3
3. Edge E : Node 3 → Node 4
4. Edge F : Node 4 → Node 3
5. Edge G : Node 3 → **Node 5**



Paths between nodes

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**

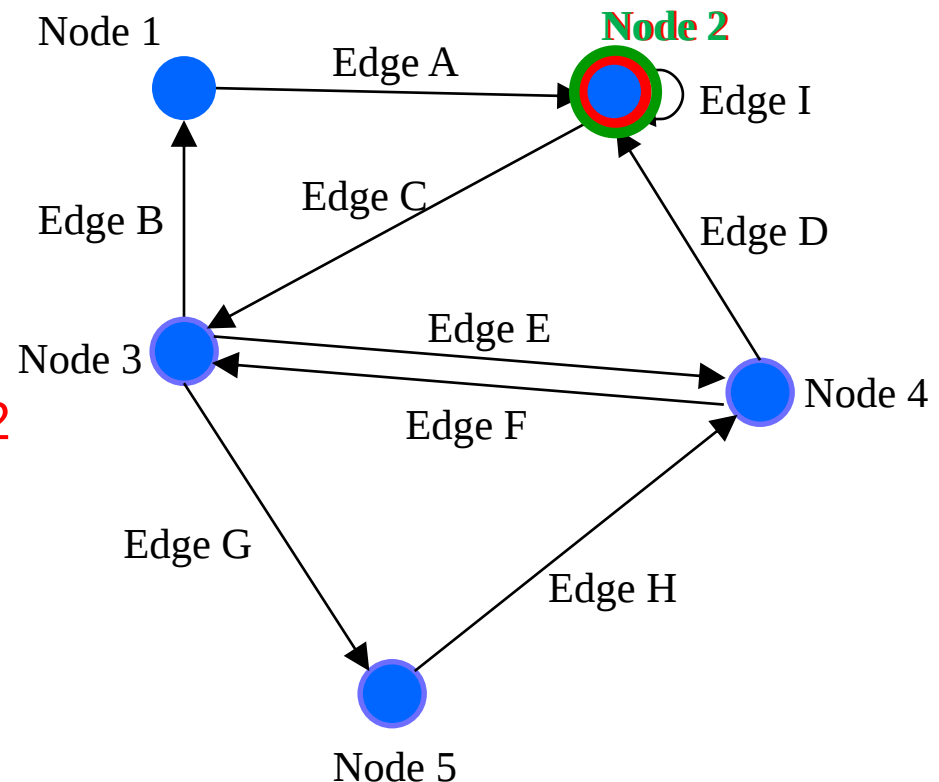
Paths between nodes

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

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: 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
- Let’s look at the JavaDoc from HW4... (demo)

HW5: Testing

- The design process includes crafting a good test suite
 - Specification tests and implementation tests
- **Specification tests** (`src/test/resources/specTests/`)
 - 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
- **Implementation tests** (`src/test/java/graph/implTest/`)
 - JUnit test classes
 - Validate behavior particular to your ADT design
- If you can validate a behavior using either test type, use a specification test!

HW5: Specification tests

Each specification test expressed as text-based script ***foo.test***

- One command per line, of the form: **Command** *arg₁* *arg₂* ...
- Script's output compared against ***foo.expected***
- Precise details specified in the homework
- Match format **exactly**, including whitespace!

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

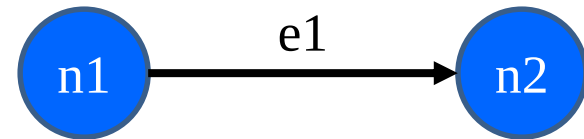
HW5: Why Specification tests?

- 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 specification 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: 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
```

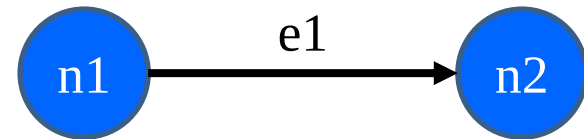
```
# Print all nodes in the graph
ListNodes graph1
```

```
# 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 specification 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/specTests/`
4. Run all such tests via the Gradle task `specTests`
 - After class implemented and **GraphTestDriver** stubs filled

HW5: Creating implementation tests

1. Create JUnit test class in `src/test/java/graph/implTest/`
2. Write a test method for each unit test
3. Run all such tests via the Gradle task `implTests`

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

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

HW5: Creating implementation tests

1. Note: Your implementation tests will fail in hw5 part 1, because you have not implemented the actual methods yet
 - The same goes for your specification tests
2. You will do that in part 2

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.

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
<code>assertTrue(test)</code>	<code>test == false</code>
<code>assertFalse(test)</code>	<code>test == true</code>
<code>assertEquals(expected, actual)</code>	<code>expected</code> and <code>actual</code> are not equal
<code>assertSame(expected, actual)</code>	<code>expected != actual</code>
<code>assertNotSame(expected, actual)</code>	<code>expected == actual</code>
<code>assertNull(value)</code>	<code>value != null</code>
<code>assertNotNull(value)</code>	<code>value == null</code>

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