# CSE 331
# Software Design & Implementation

Autumn 2022

Section 5 – Graphs, Equals and Hashcode

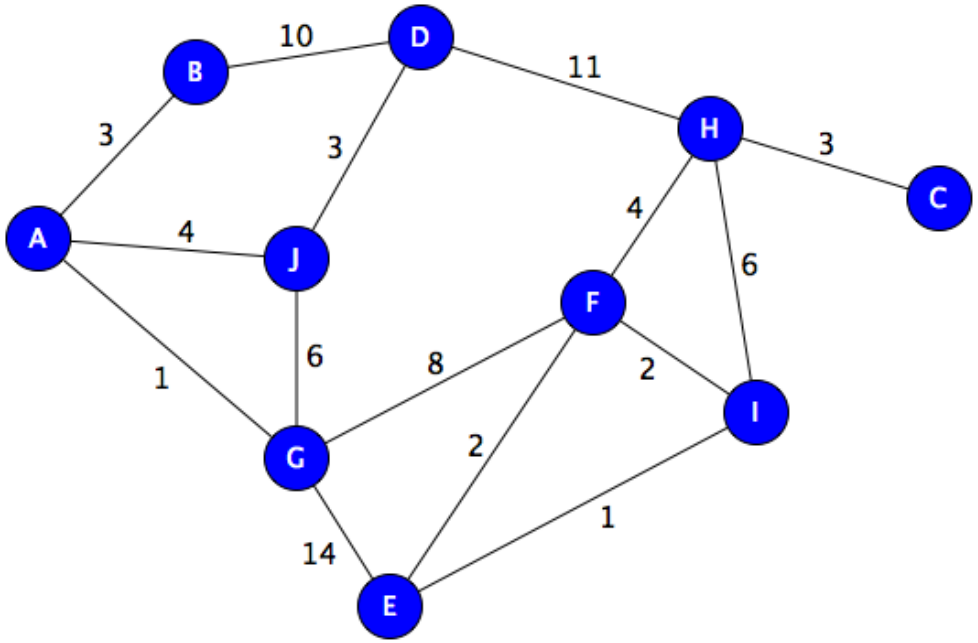
# Administrivia

- HW4 due yesterday!
- HW5 out now, due next Wednesday (11/2) at 11 pm!
- Any questions?

# Agenda

- Graph concepts
- HW5
- Script Testing
- Equals and Hashcode

# Graphs



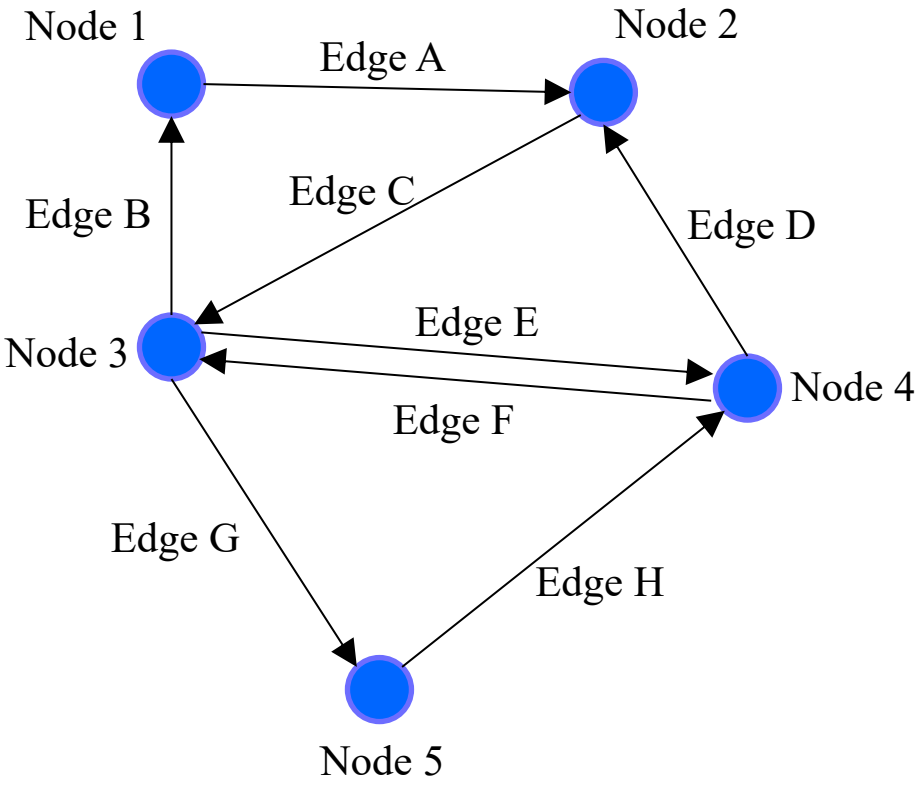
# A graph represents relationships

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

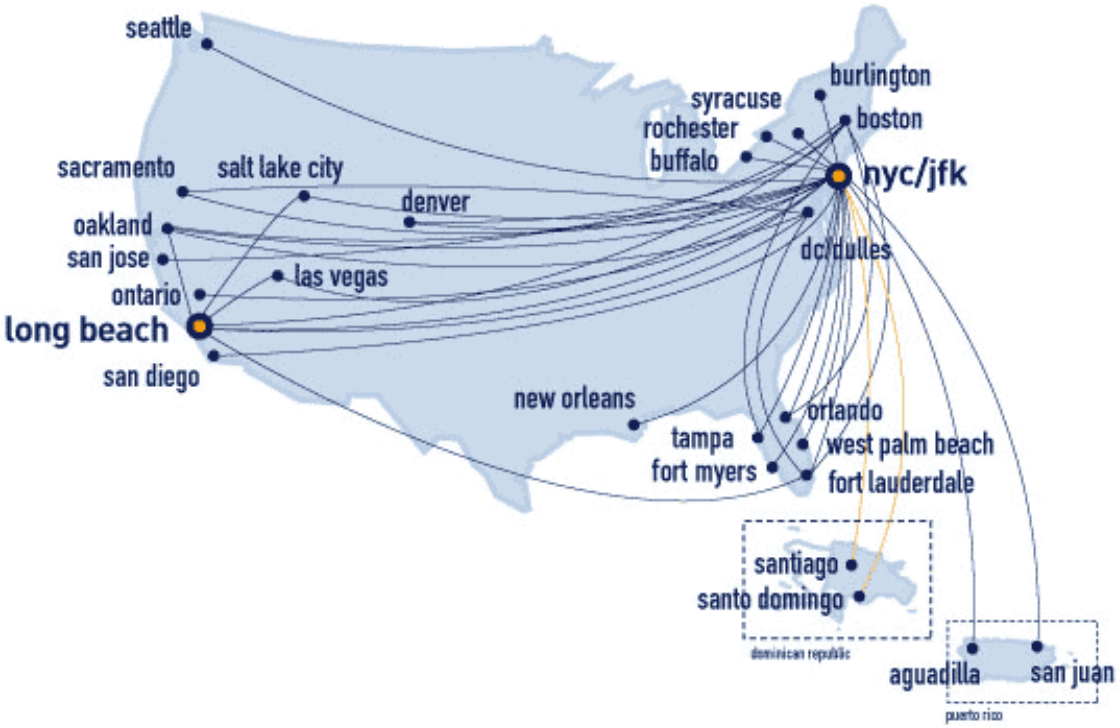

**Nodes:** intersections (cities)

**Label:** name/location

**Edges:** roads

**Label:** name/length

# Example: Airline Flights



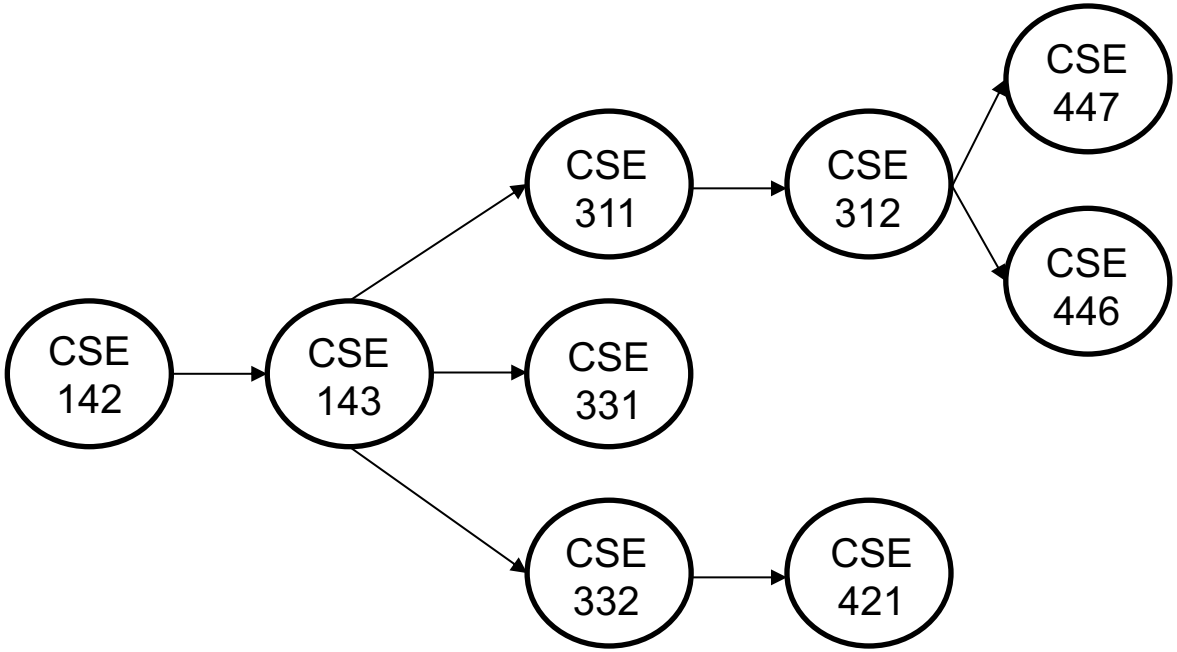

**Nodes:** airports

**Label:** airport code

**Edges:** flights

**Label:** cost/time

# Example: CSE courses




**Nodes:** Courses

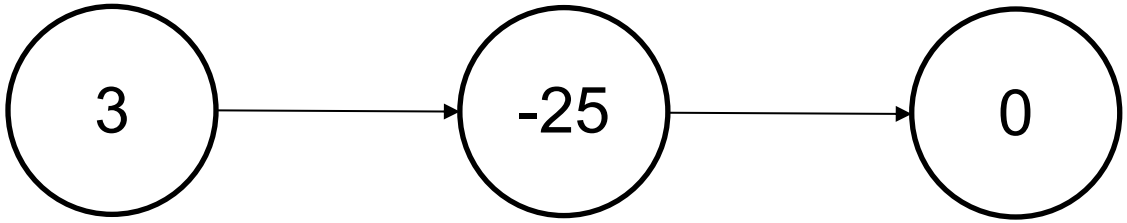
**Label:** Course name

**Edges:** pointer to next class

**Label:** none

# You've used graphs before!

**Singly linked Lists:**

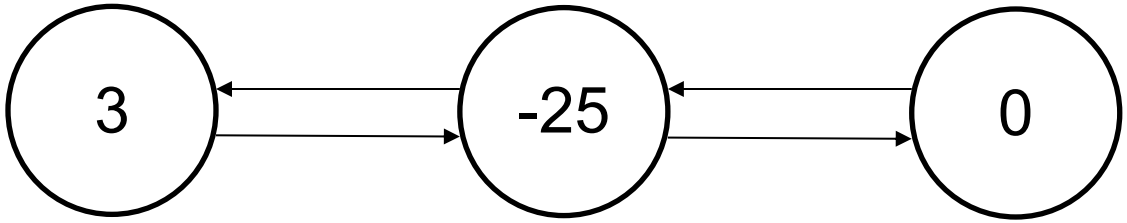



**Nodes:** Linked list node
**Label:** integer

**Edges:** pointer to next node
**Label:** none

# You've used graphs before!

**Doubly linked Lists:**

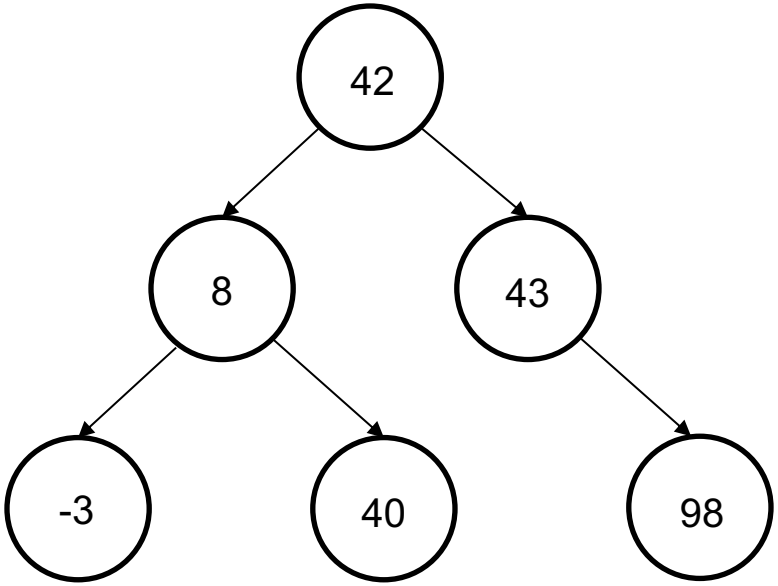



**Nodes:** Linked list node
**Label:** integer

**Edges:** pointers to prev/next nodes
**Label:** none

# You've used graphs before!

**Binary trees:**



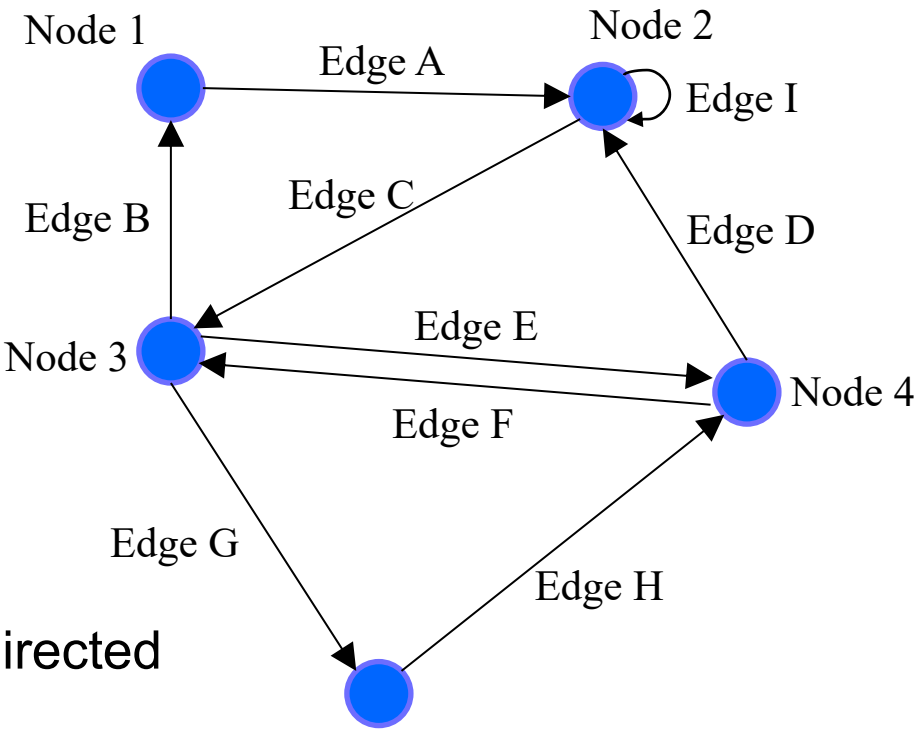

**Nodes:** Tree node
**Label:** Integer

**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

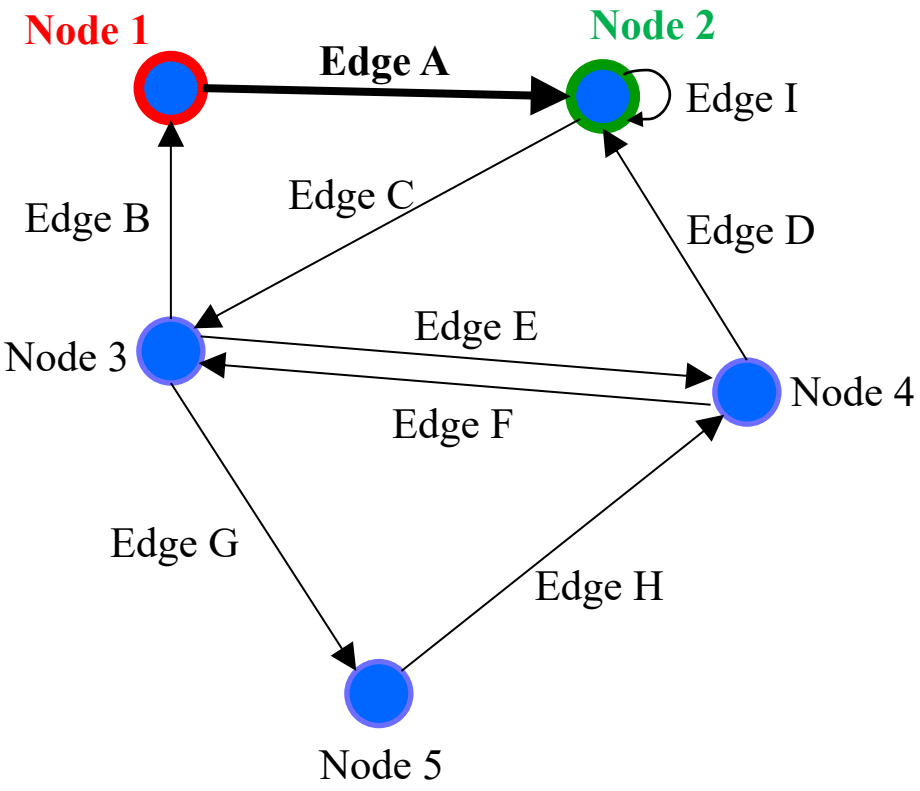



N.B.: We're only dealing with directed graphs from here on out.

# 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

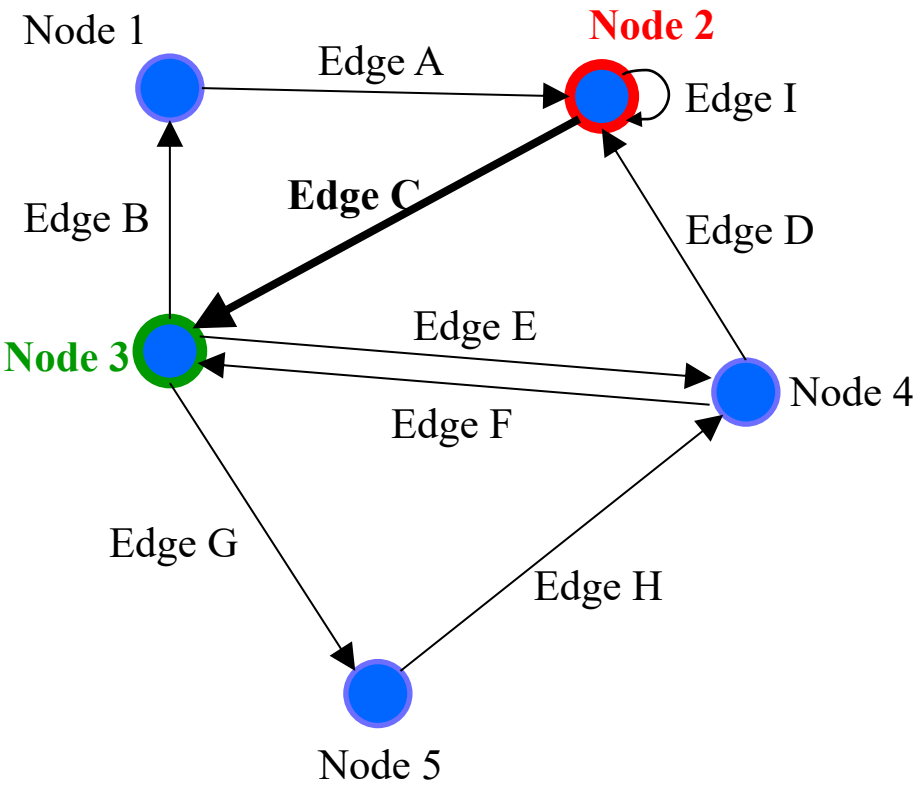
# An edge points from source to dest.

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

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

Edge C is Node 2 → Node 3.

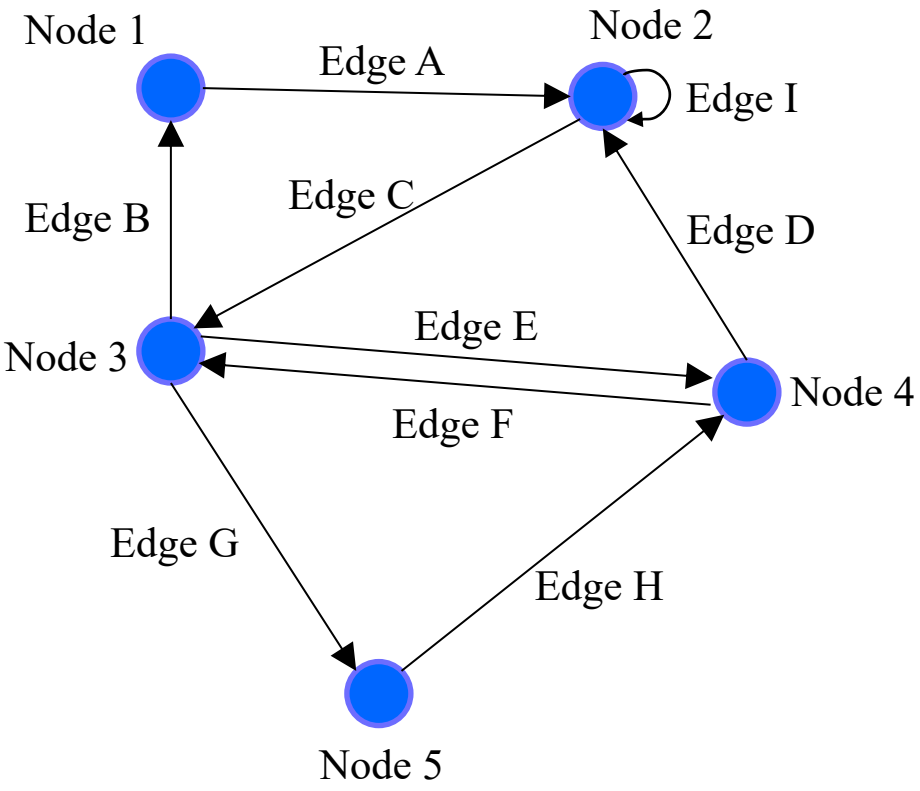
- Outgoing from Node 2
- Incoming to Node 3



# A node has children

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

- Potentially empty set



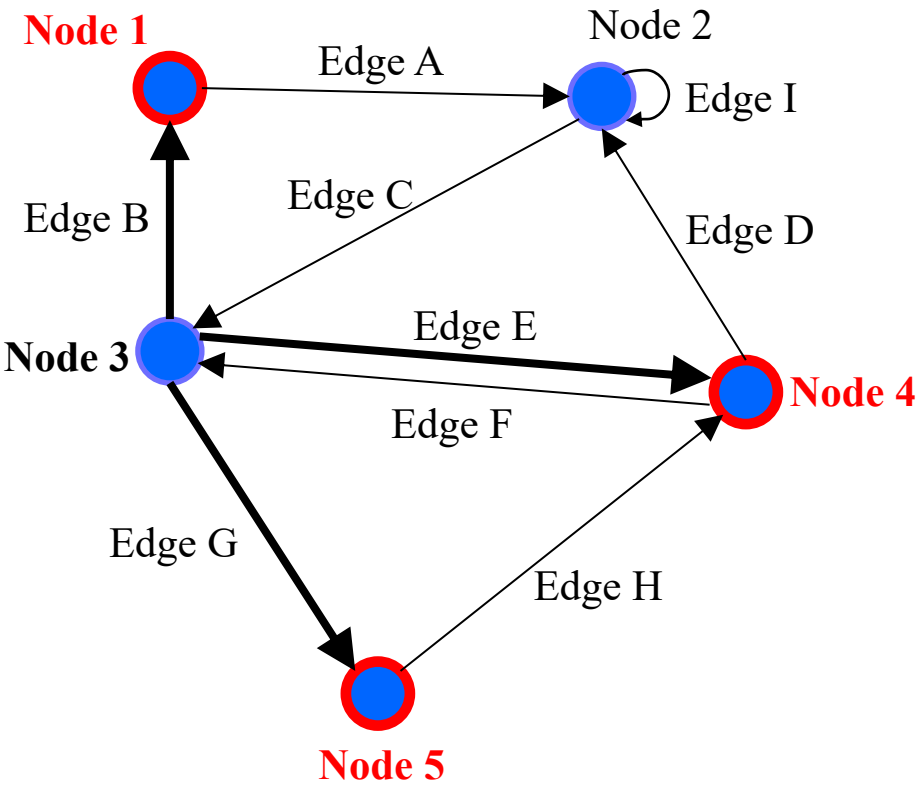
# A node has children

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

- Potentially empty set

Node 3 has three children:

- Node 1
- Node 4
- Node 5



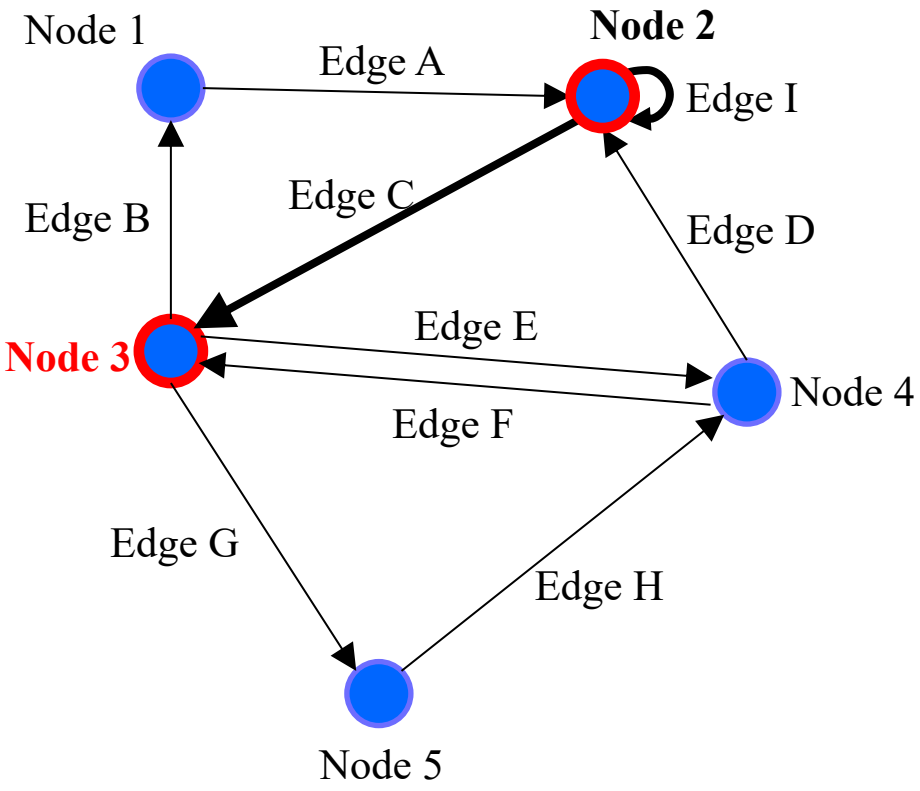
# A node has children

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

- Potentially empty set

Node 2 has two children:

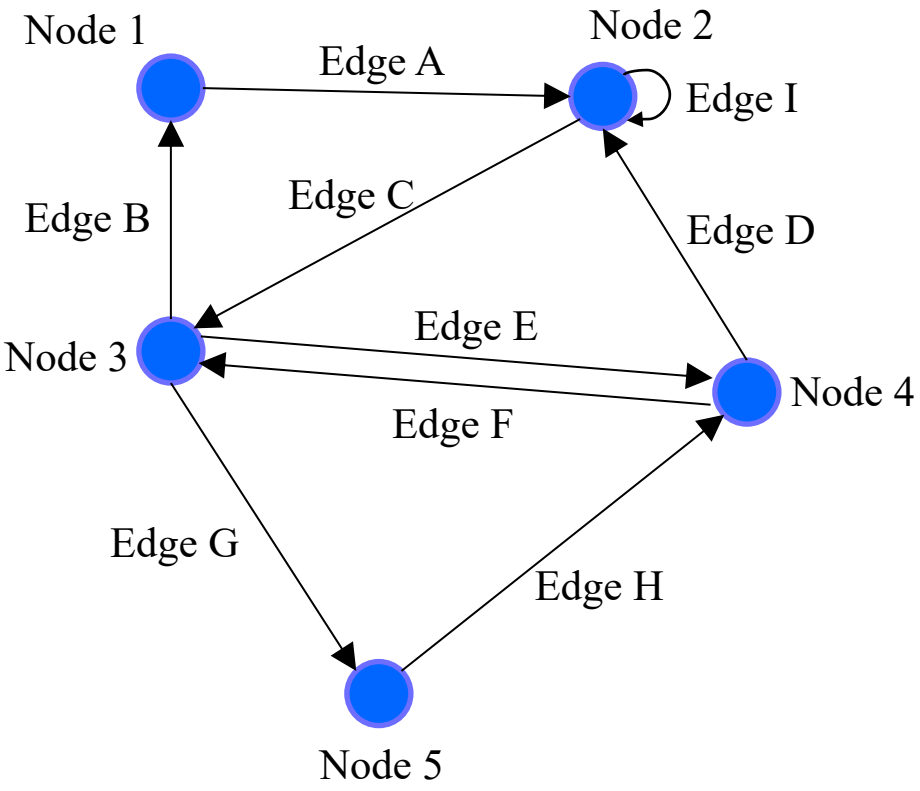
- Node 2
- Node 3



# A node has parents

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

- Potentially empty set



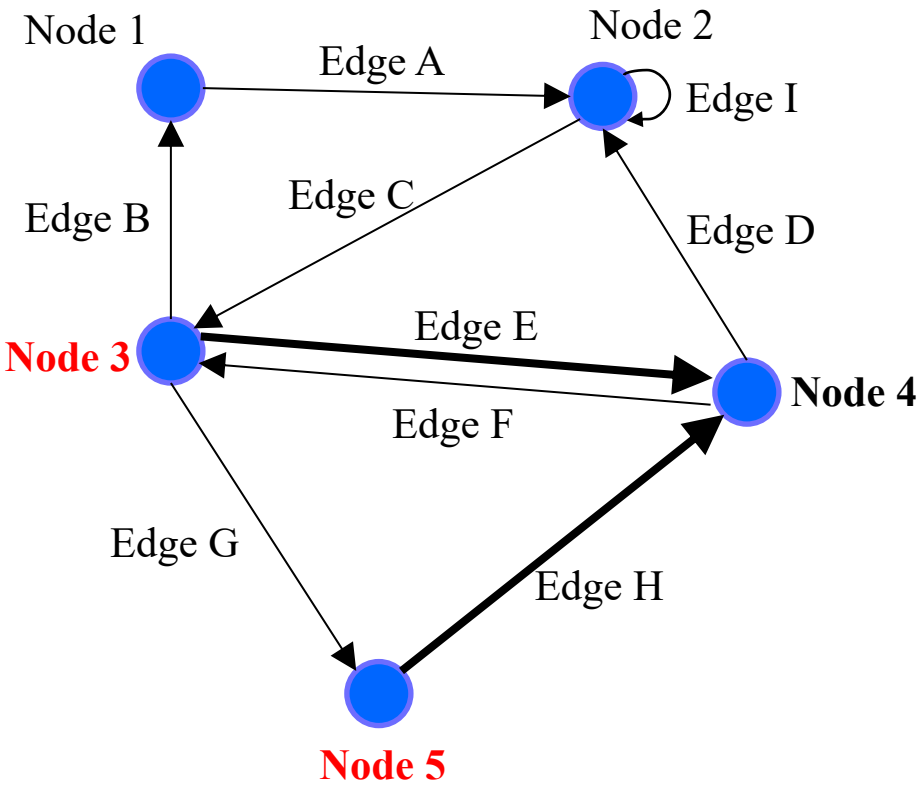
# A node has parents

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

- Potentially empty set

Node 4 has two parents:

- Node 3
- Node 5



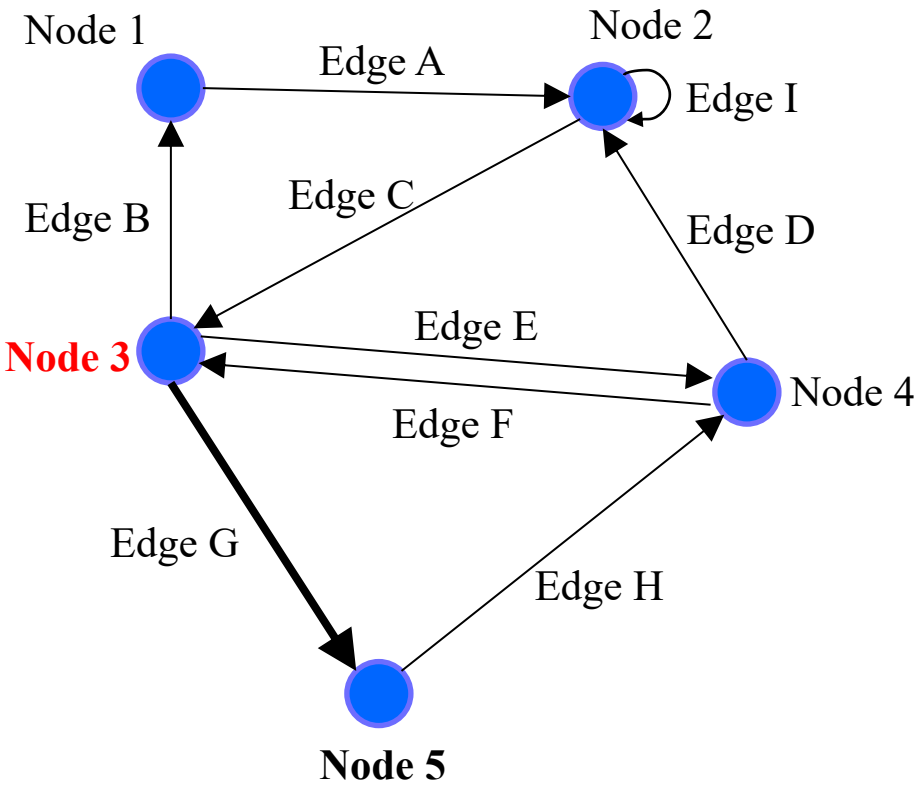
# A node has parents

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

- Potentially empty set

Node 5 has one parent:

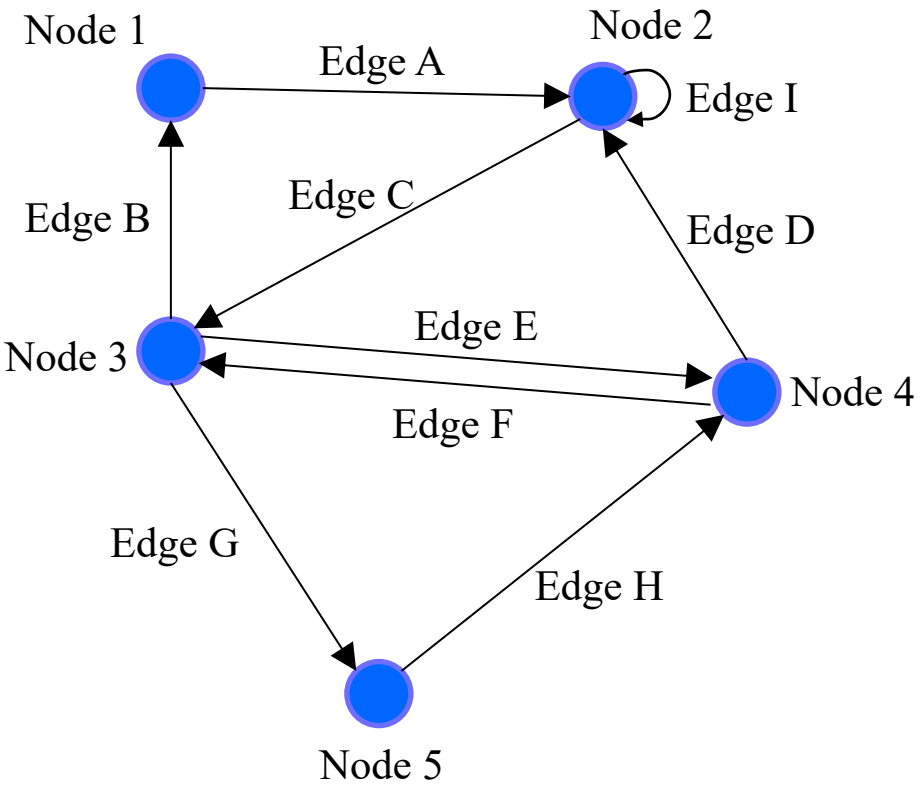
- Node 3



# A node has neighbors

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

- Potentially empty set



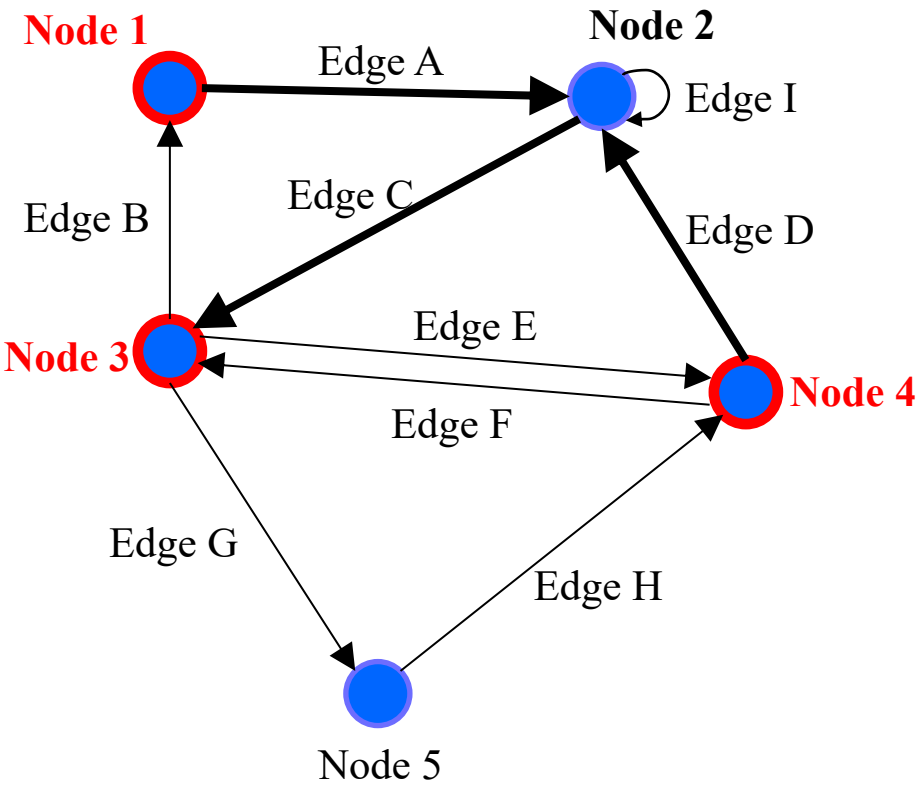
# A node has neighbors

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)



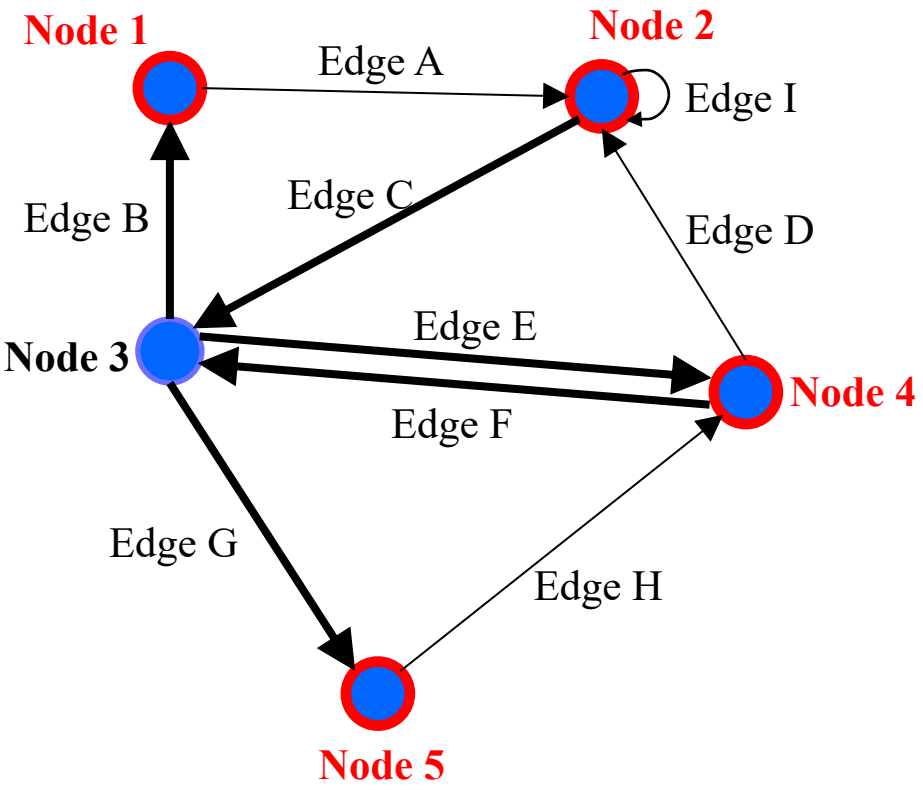
# A node has neighbors

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

- Potentially empty set

Node 3 has four neighbors:

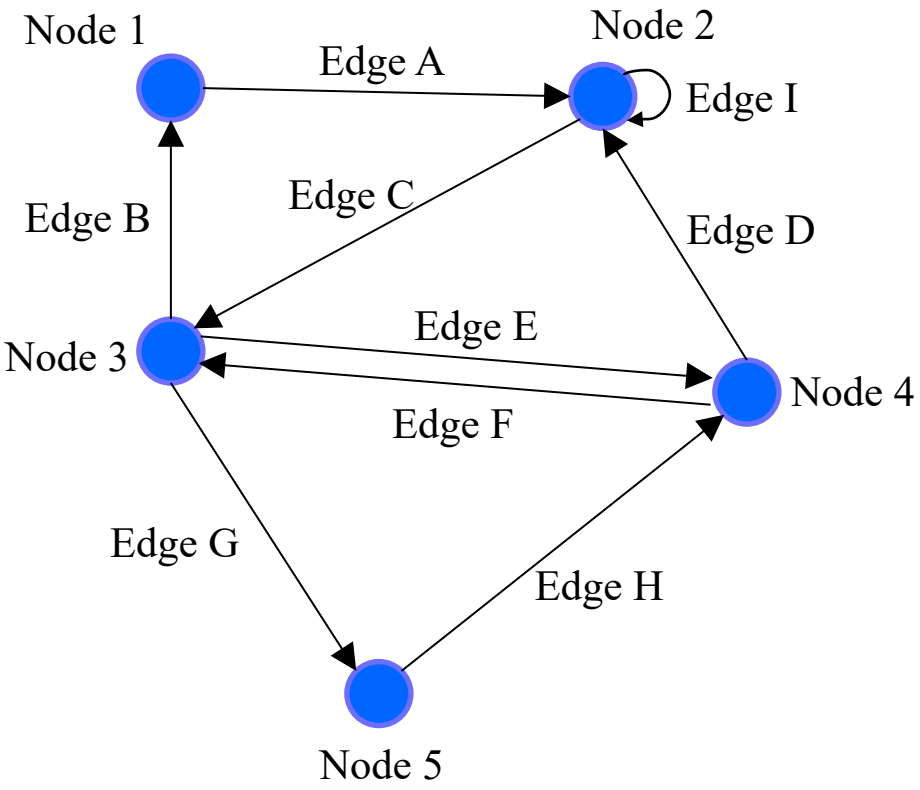
- Node 1 (child)
- Node 2 (parent)
- Node 4 (parent *and* child)
- Node 5 (child)



# Paths between nodes

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

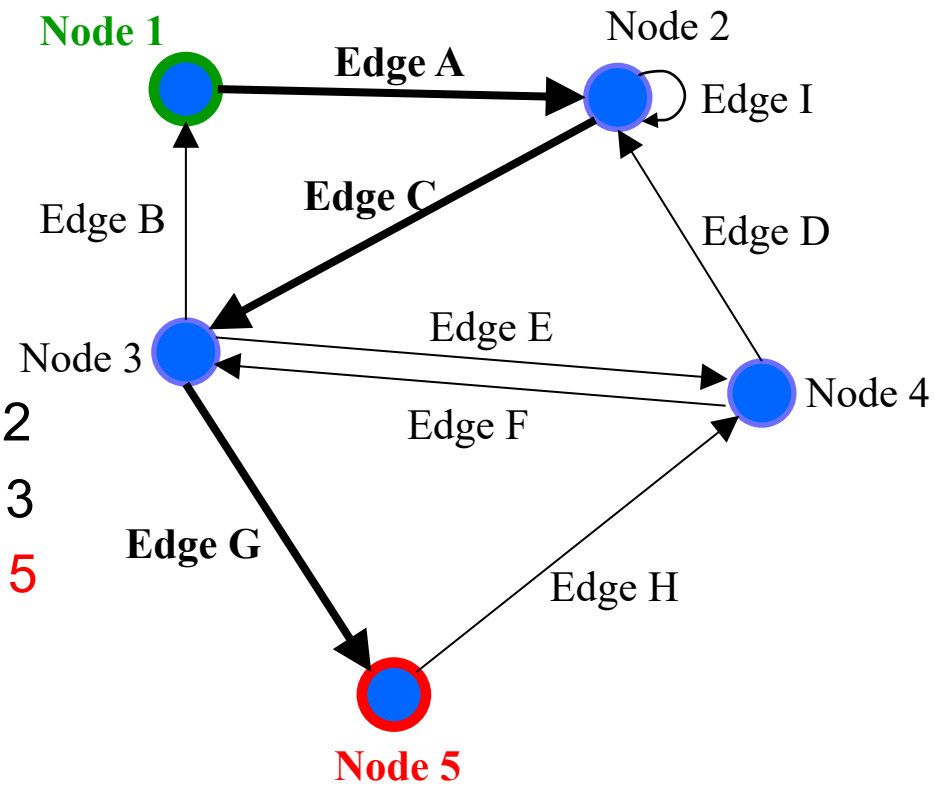
- Potentially empty sequence
- Might include a cycle
- Often want shortest



# 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



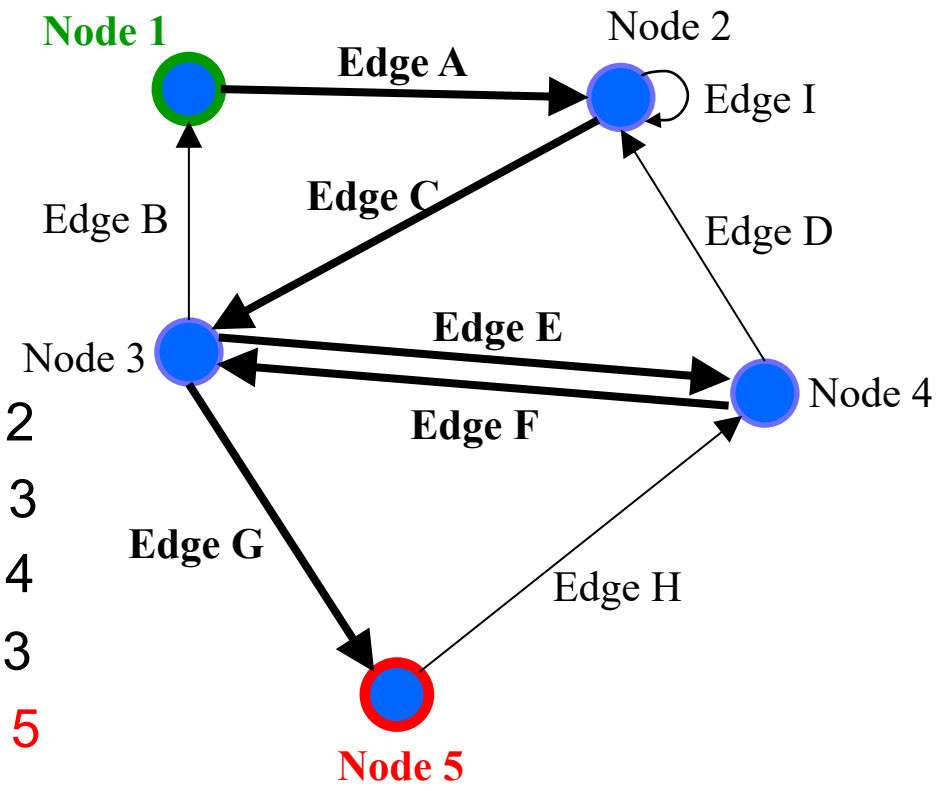

Path from Node 1 to Node 5:

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



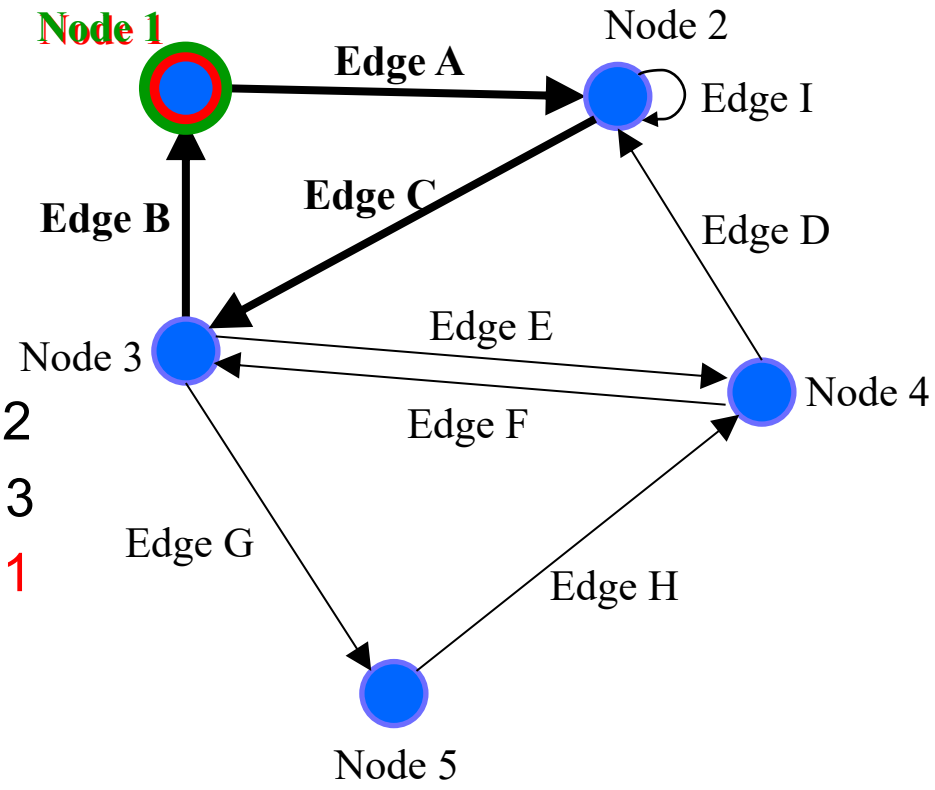

Path from Node 1 to Node 5:

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



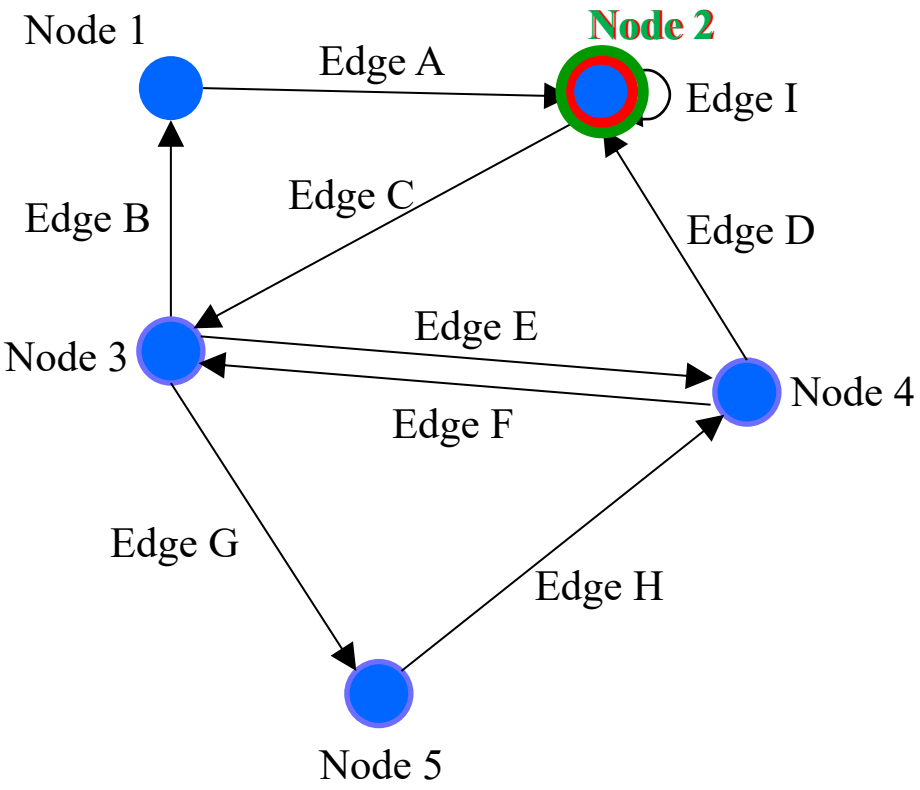

Path from Node 1 to Node 1:

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 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 (**`String`**s)
  - 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!

| 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`

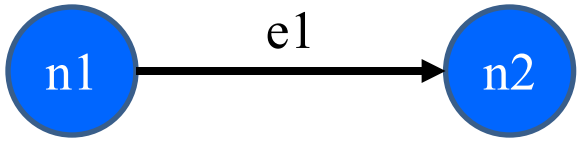
```
# Create a graph
CreateGraph graph1

# 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`

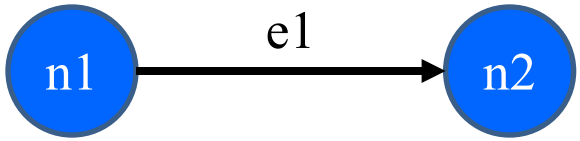
```
# Create a graph
created graph graph1

# 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: 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: `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.

- These two commands have output in a specific format and in sorted order
  - But your methods should not return things in this format or in sorted order
  - Instead, your methods should return the necessary information in collections

# 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

- 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

# Equals and Hashcode

# The `equals` method (review)

- Specification mandates several properties:
  - *Reflexive*: `x.equals(x)` is `true`
  - *Symmetric*: `x.equals(y)` $\Leftrightarrow$ `y.equals(x)`
  - *Transitive*: `x.equals(y)` $\wedge$ `y.equals(z)` $\Rightarrow$ `x.equals(z)`
  - *Consistent*: `x.equals(y)` shouldn't change, unless perhaps `x` or `y` did
  - *Null uniqueness*: `x.equals(null)` is `false`

- Several notions of equality:
  - *Referential*: literally the same object in memory
  - *Behavioral*: no sequence of operations could tell apart (excluding `==`)
  - *Observational*: no sequence of observer operations could tell apart (excluding `==`)

# The `hashCode` method (review)

- Specification mandates several properties:
  - *Self-consistent*: `x.hashCode()` shouldn't change, unless `x` did
  - *Equality-consistent*: `x.equals(y)` ⇒ `x.hashCode() == y.hashCode()`

- Equal objects *must* have the same hash code.
  - Implementations of `equals` and `hashCode` work together for this
  - If you override `equals`, you ***must*** override `hashCode` as well

- Ideally a good `hashCode` method returns different values for unequal objects, but the contract does not require this.

# Overriding `equals` and `hashCode`

- A subclass method overrides a superclass method, when…
  - They have the exact same name
  - They have the exact same argument types
- An overriding method should satisfy the overridden method's spec.
- Always use `@override` tag when overriding `equals` and `hashCode` (or any other overridden method)
- Note: Method overloading is not the same as overriding
  - Same name but distinguished by different argument types
- Keep these details in mind if you override `equals` and `hashCode`.

# `equals` and `hashCode` worksheet

- Let's practice…