CS 5004: Lecture 11

Northeastern University, Spring 2021

At the start of every lecture

1. Pull the latest code from the lecture-code repo
2. Open the Evening_lectures folder
3. Copy this week's folder somewhere else
   - So you can edit it without causing GitHub conflicts
4. Open the code:
   1. Find the build.gradle file in the folder called LectureX
   2. Double click it to open the project

Agenda

- More data structures
  - Implementation of graphs and trees
- Design of programs
- MVC architecture
- Design patterns intro

More data structures

Implementing graphs and trees in Java

Trees in Java

Java Collections – sets and maps

Set

`TreeSet<E>`

or

`HashSet<E>`

Map

`TreeMap<E>`

or

`HashMap<E>`

Both based on a self-balancing binary search tree (like a red-black tree)

Trees

- Already covered in 5008 Quick recap of terminology and key characteristics
- Walkthrough a generic binary search tree using interface `Comparable` and the `compareTo` method

What is a tree?

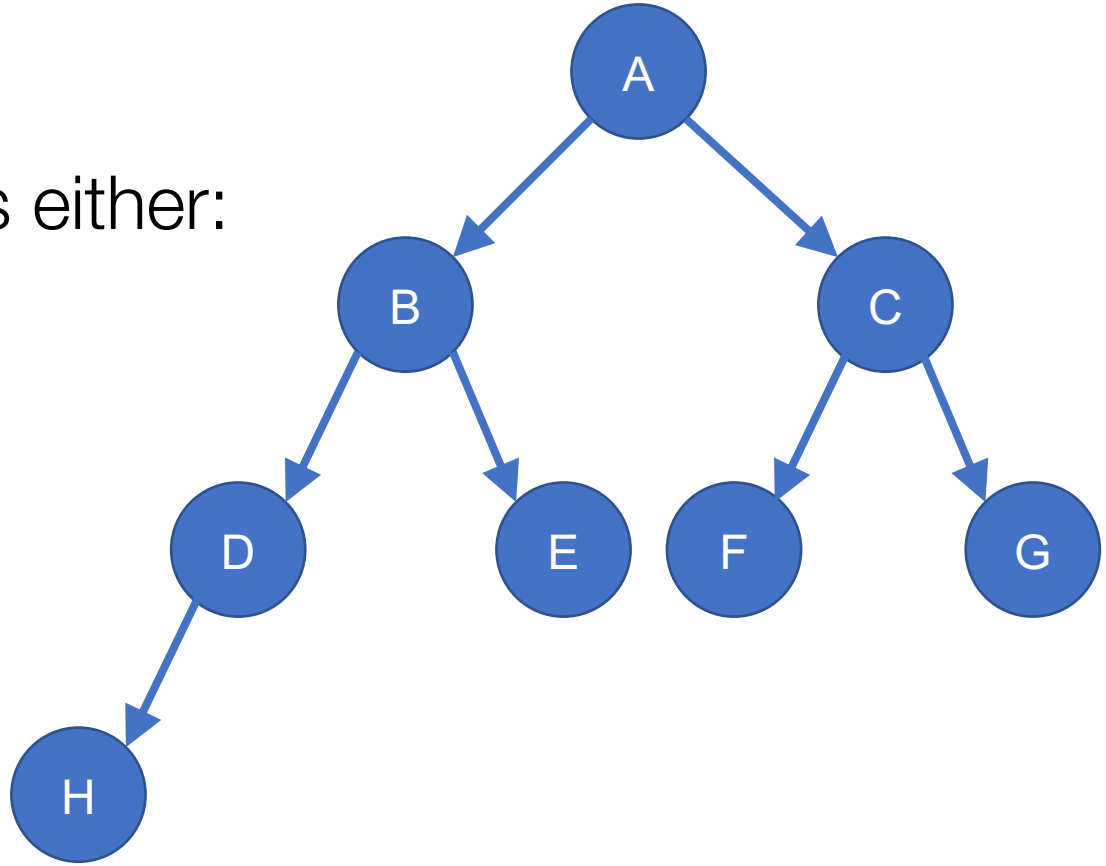
A **directed**, **acyclic** structure of linked nodes

- Directed – one-way links between nodes
- Acyclic – no path wraps back around to the same node twice

What is a tree?

Can be defined **recursively**. A tree is either:

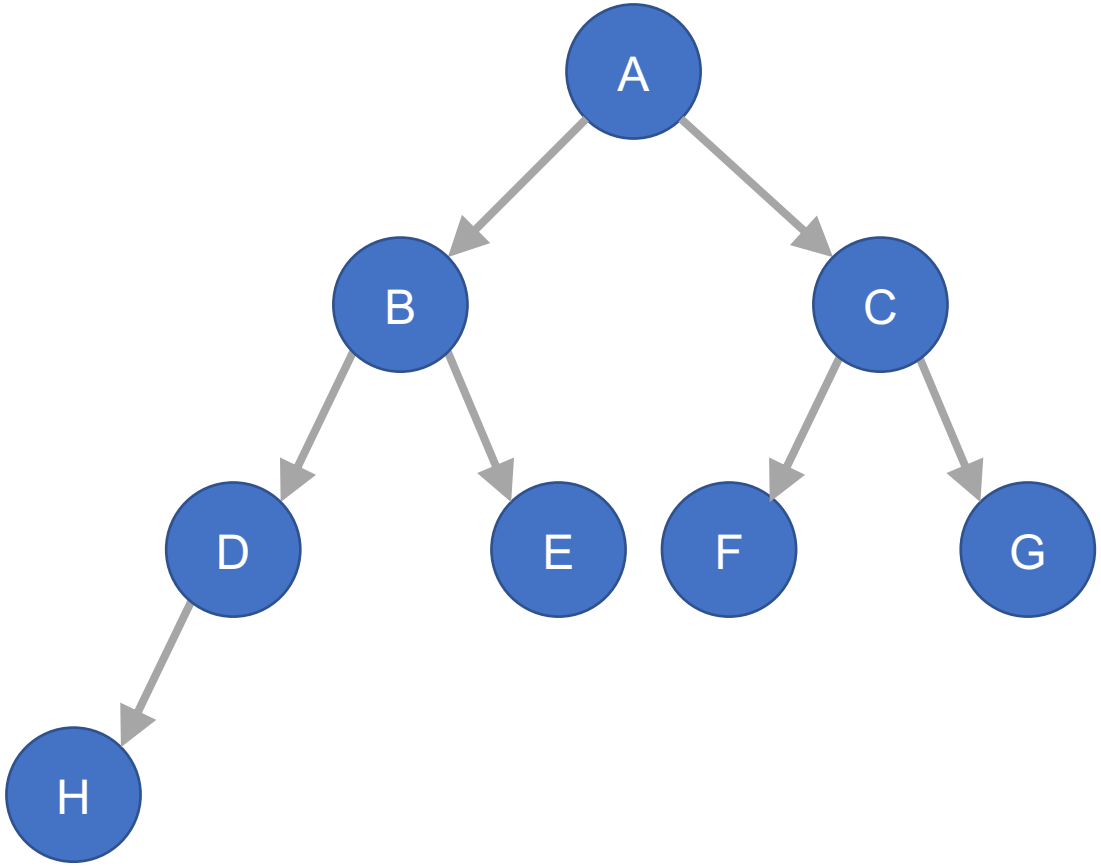
- Empty, or
- A root node that contains:
  - Data
  - A left subtree
  - A right subtree



Tree terminology

Nodes

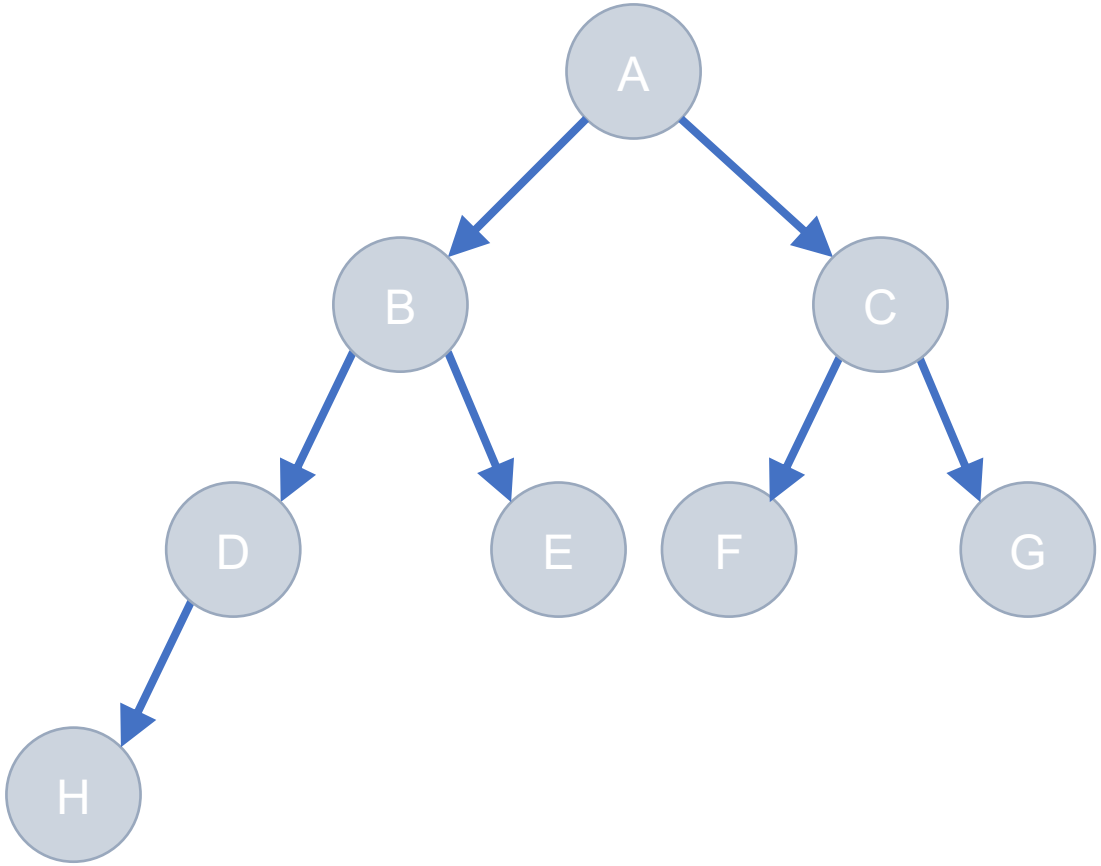
- Contain data
- Links to left/right children



Tree terminology

Edges

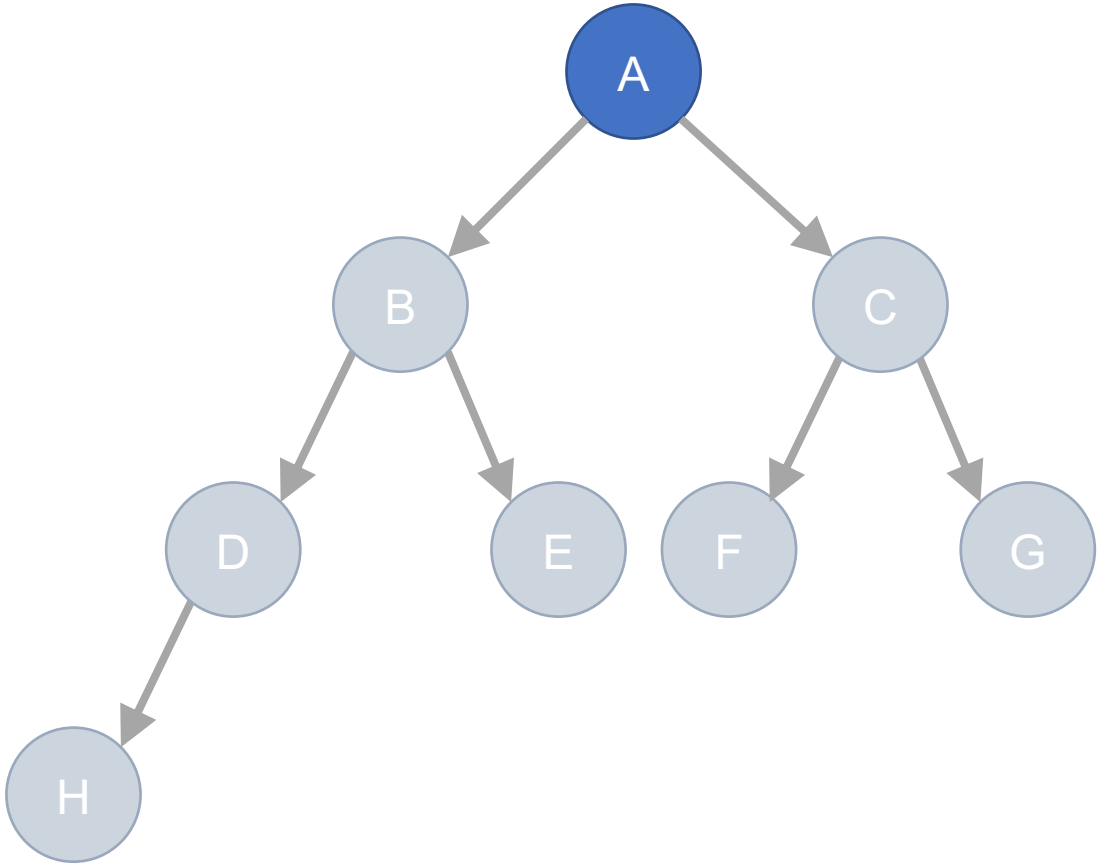
- Connect nodes



Tree terminology

Root

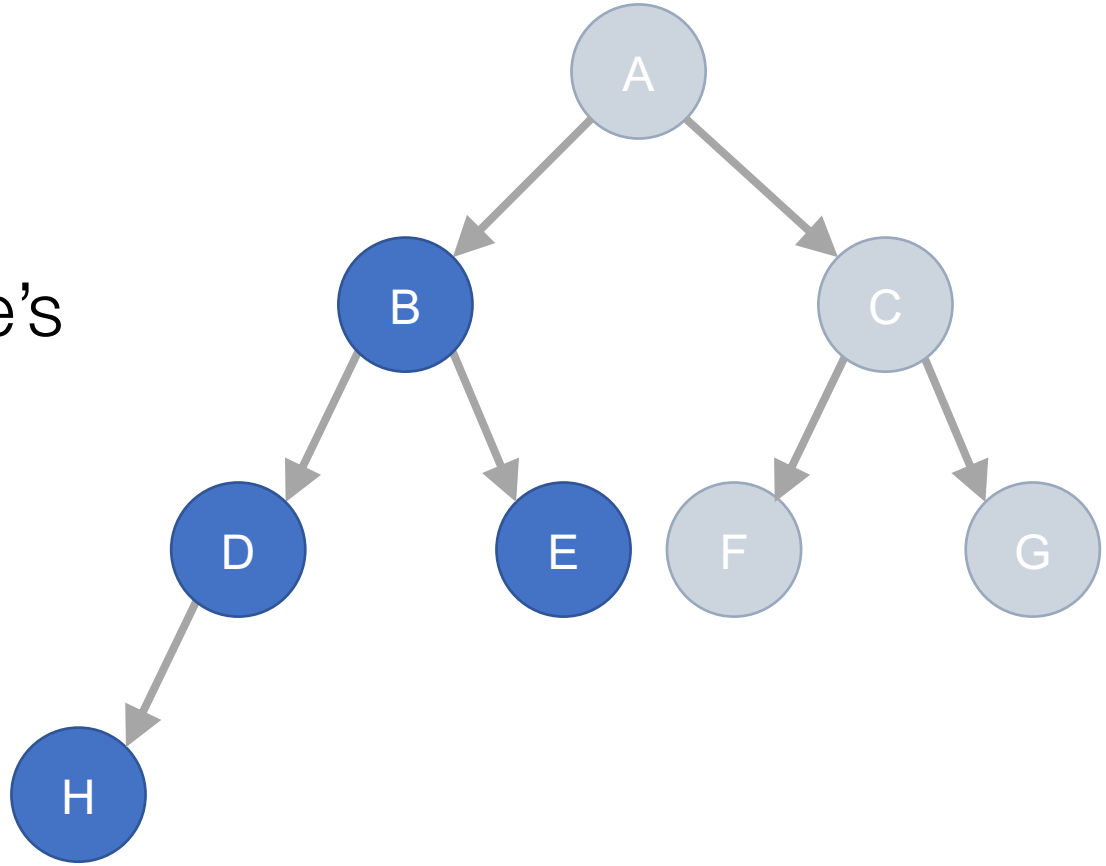
- The start of the tree



Tree terminology

Subtree

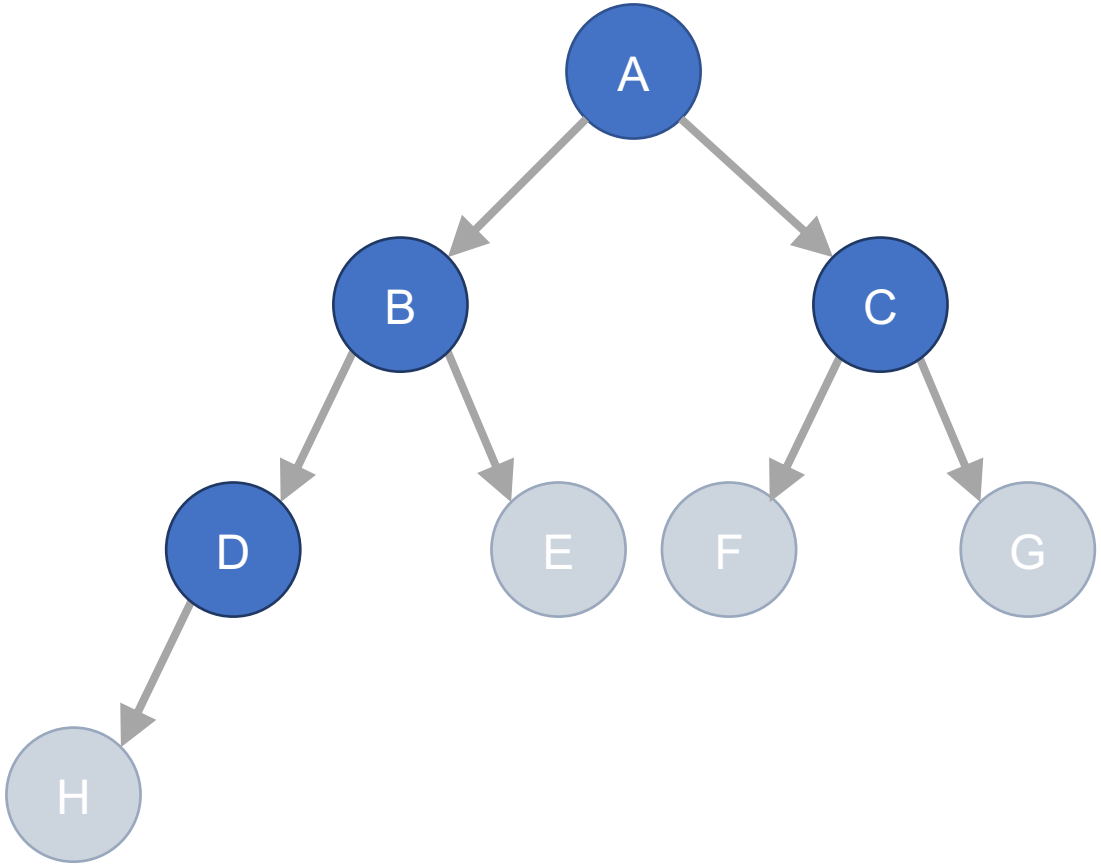
- The tree starting from one of a node's children
  - E.g. A's left subtree



Tree terminology

Parent/ancestor

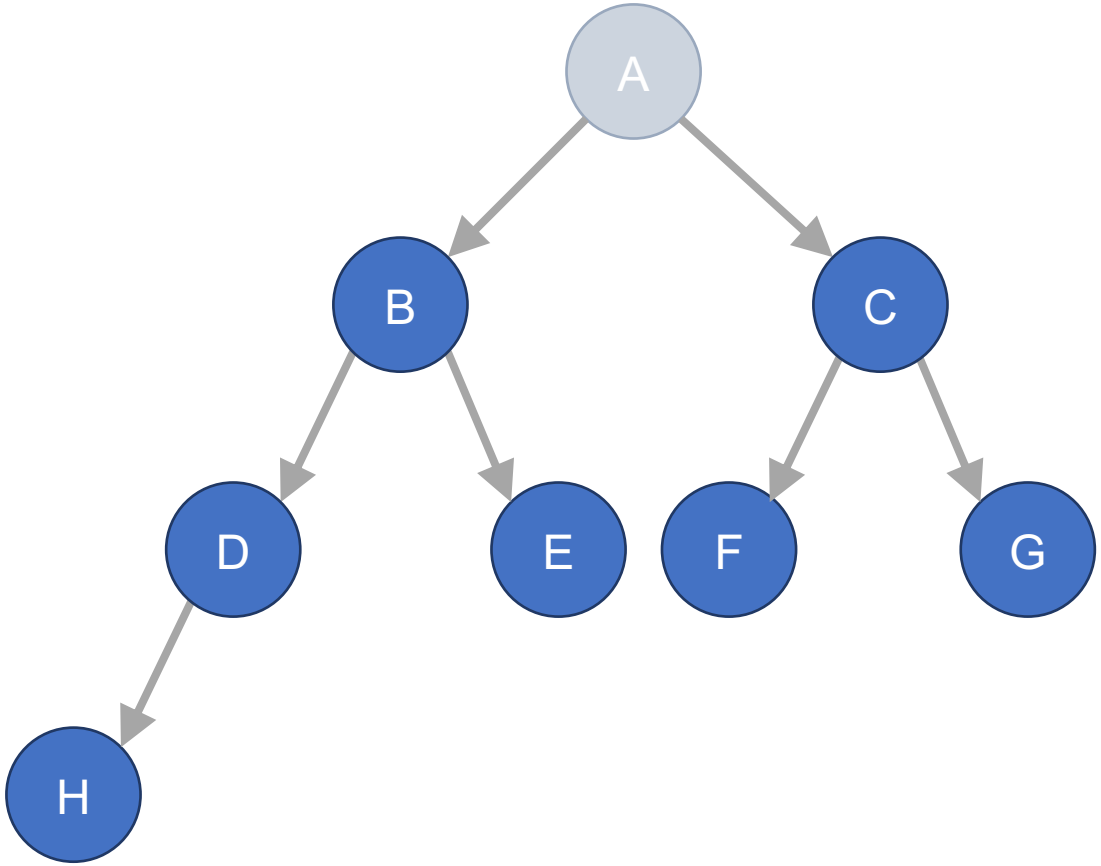
- Any node with at least one child
  - E.g. B is the parent of D and E



Tree terminology

Child

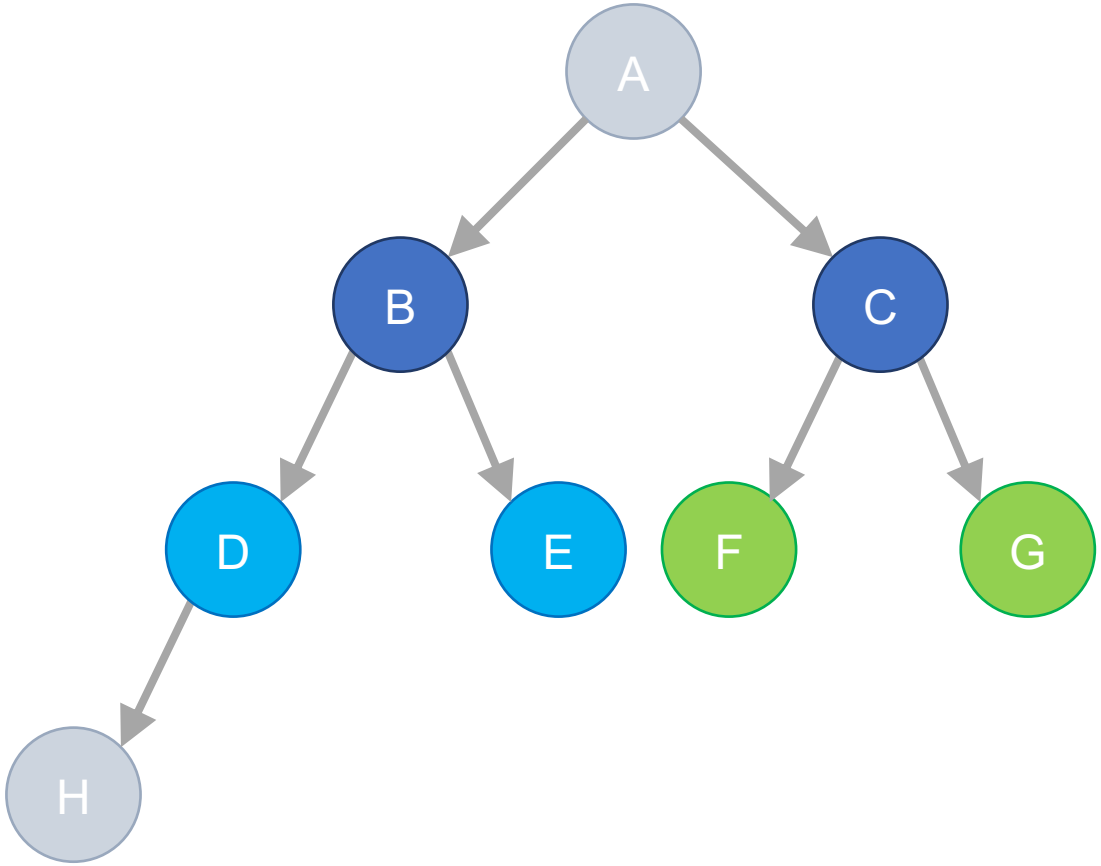
- Any node with a parent
  - E.g. B is the child of A



Tree terminology

Siblings

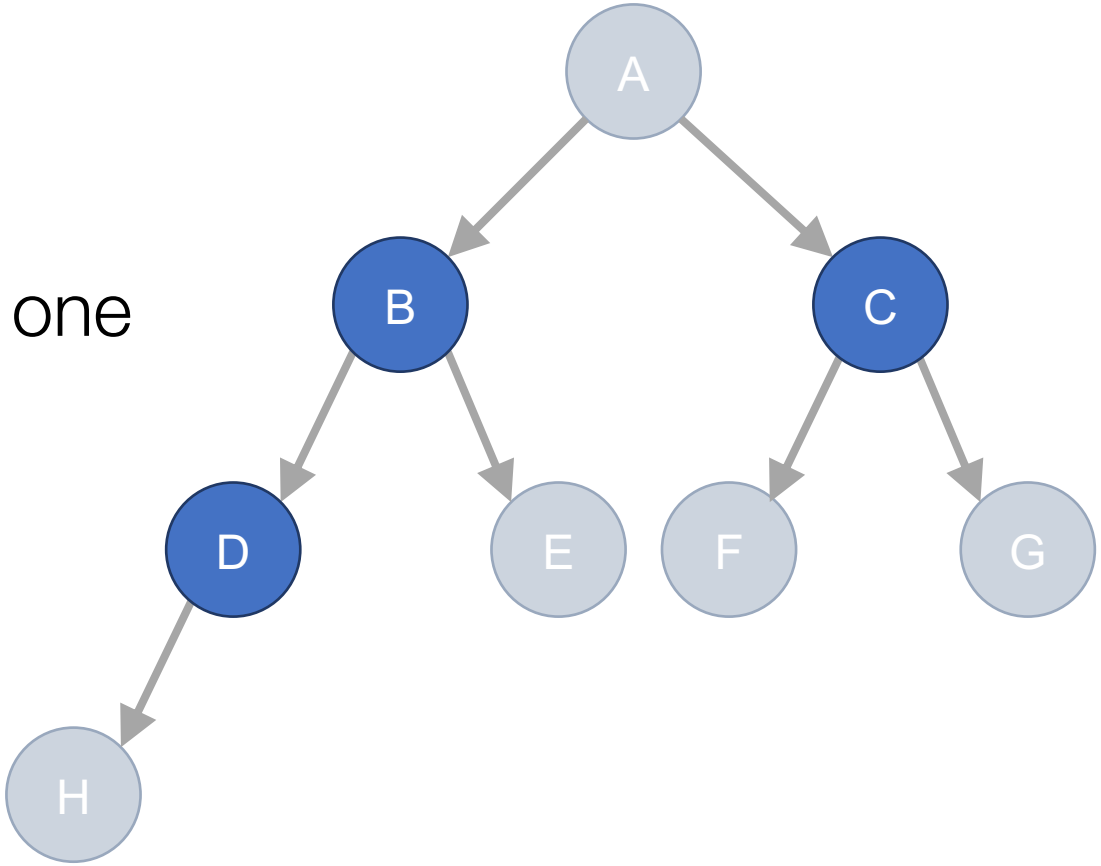
- Node with a common parent
  - B & C are siblings
  - D & E are siblings
  - F & G are siblings



Tree terminology

Branch / internal node

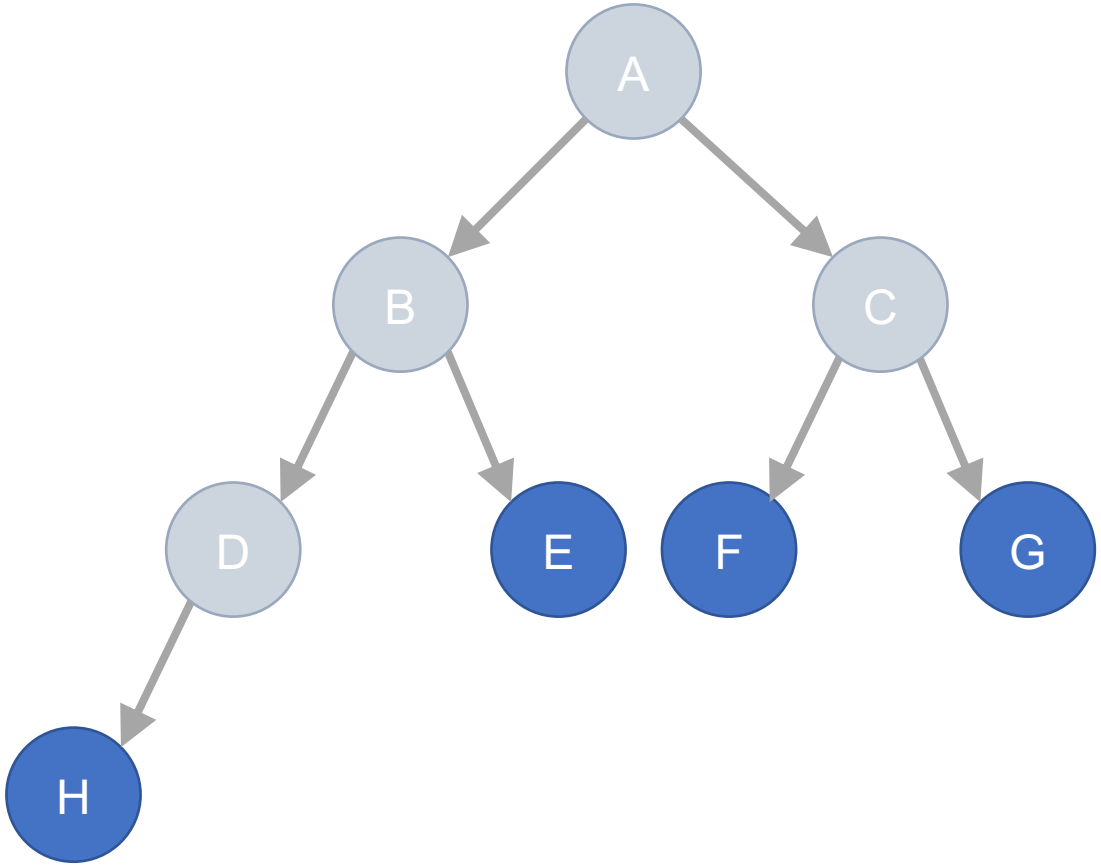
- Any node with a parent and at least one child



Tree terminology

Leaf / external node

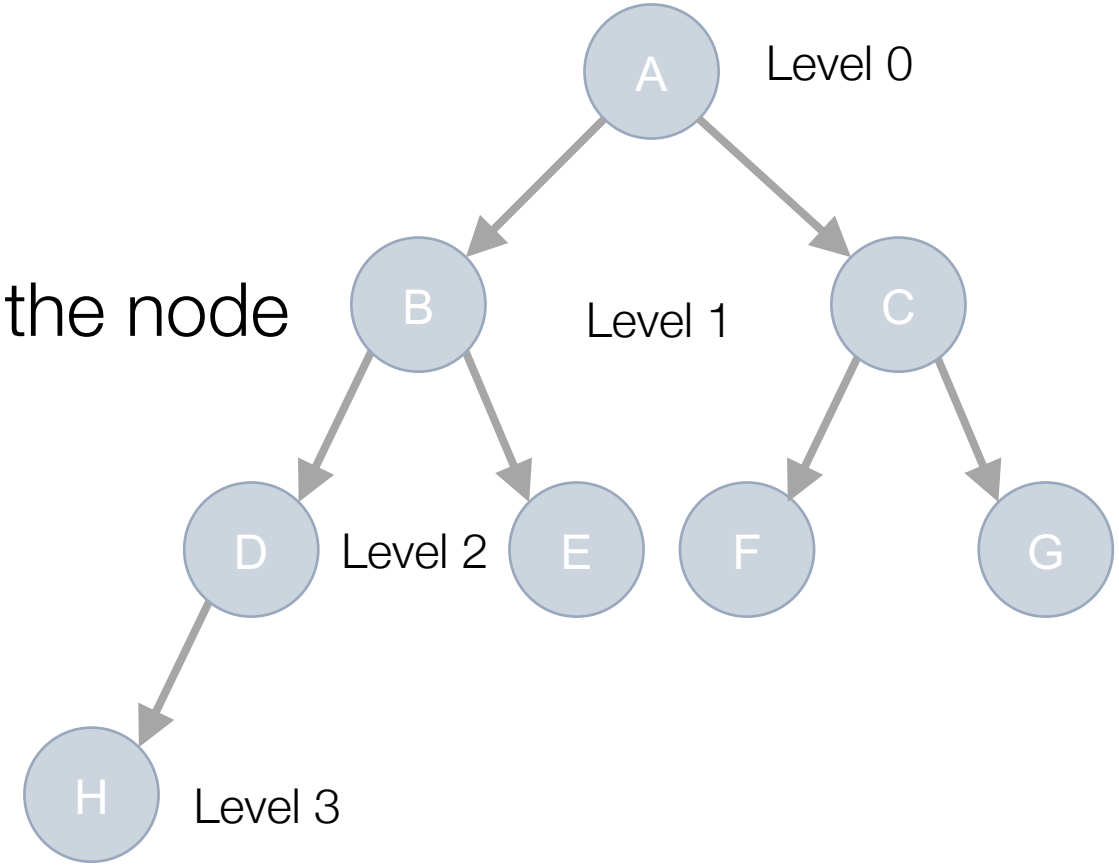
- A node with no children



Tree terminology

Level or **depth** of a node

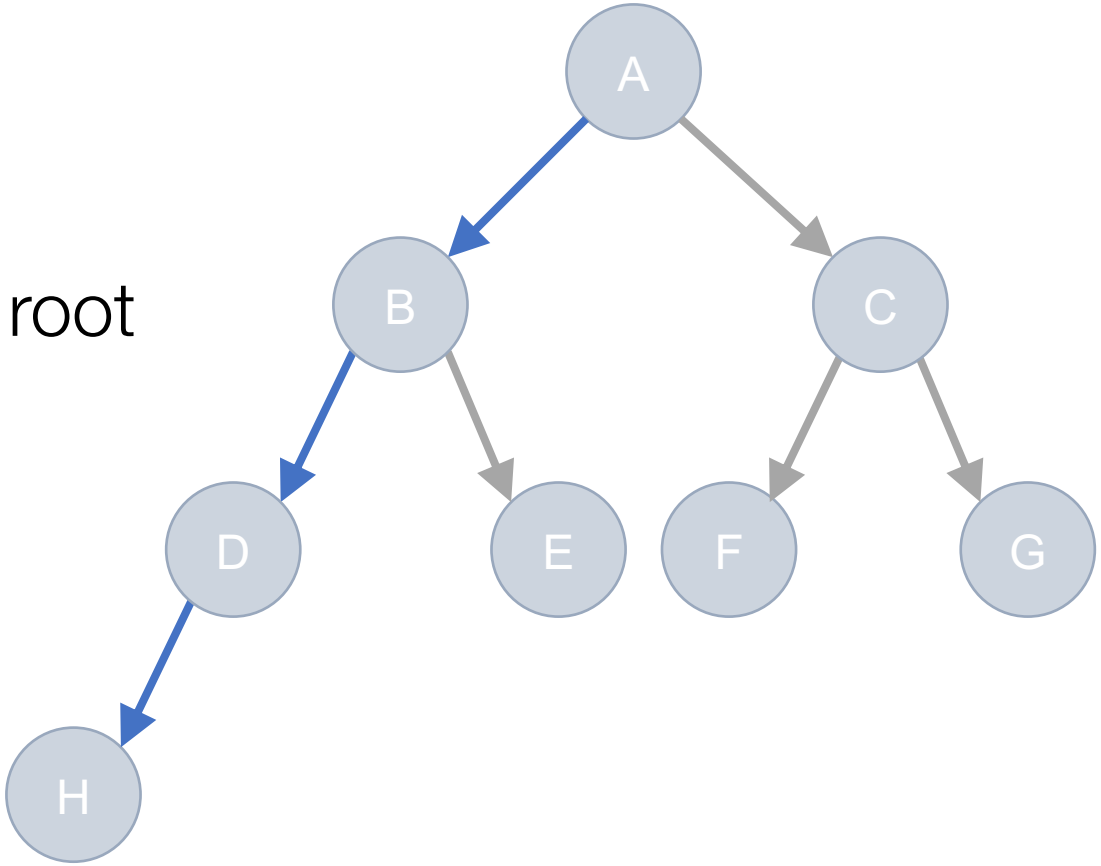
- Length of the path from the root to the node
- +1 if counting from 1 instead of 0



Tree terminology

Height of the tree

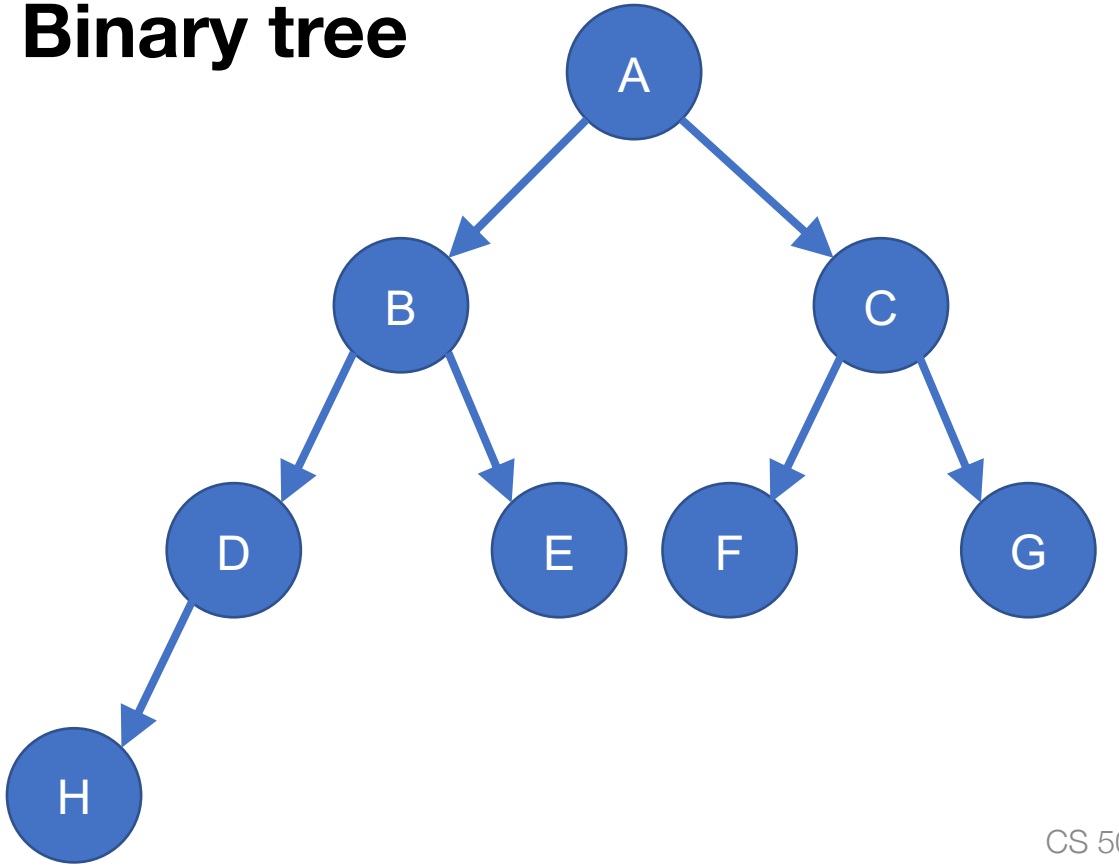
- Length of the longest path from the root to a leaf



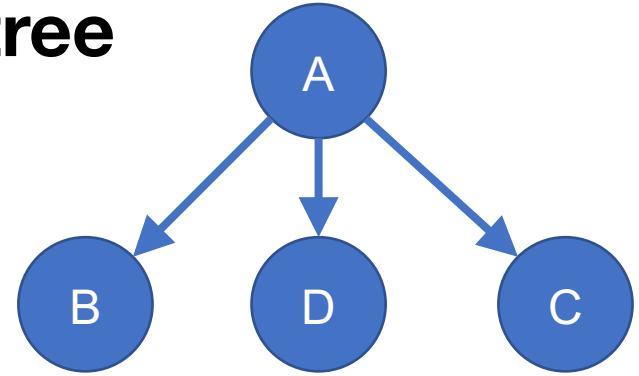
Binary Tree implementation

Binary tree – no node can have more than two children

Binary tree



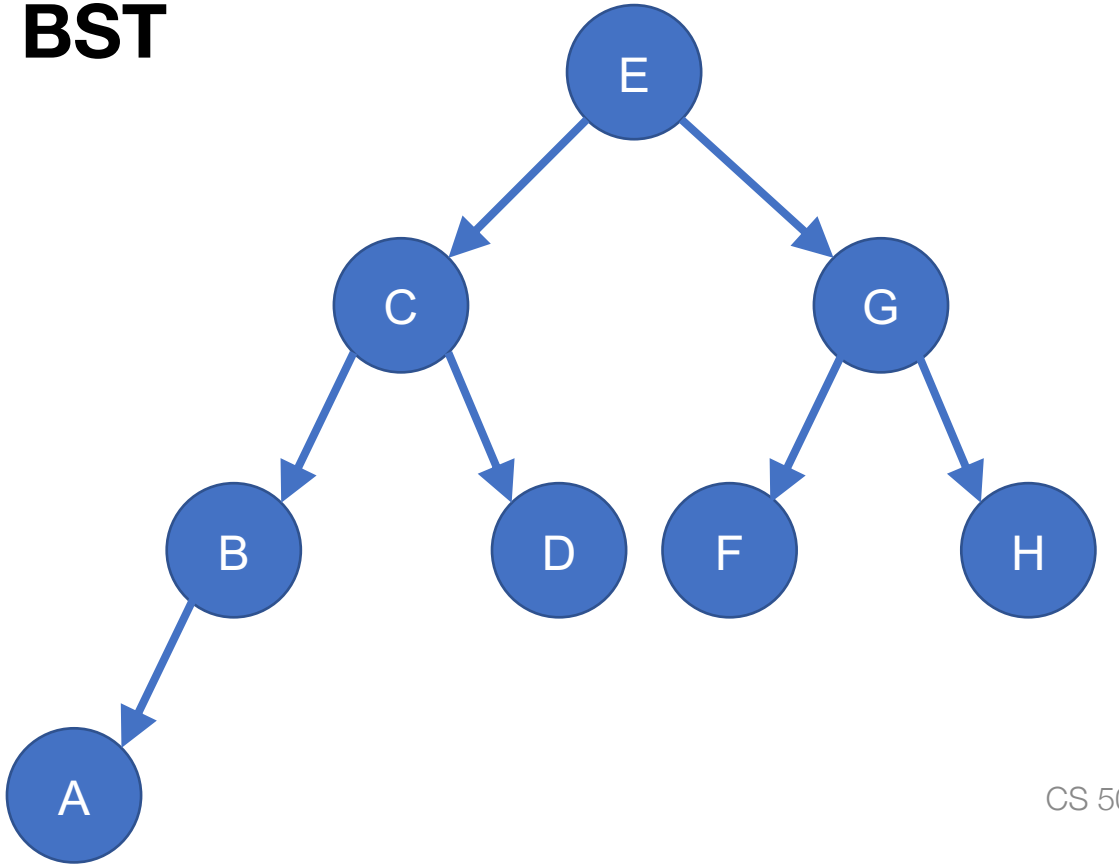
Not a binary tree



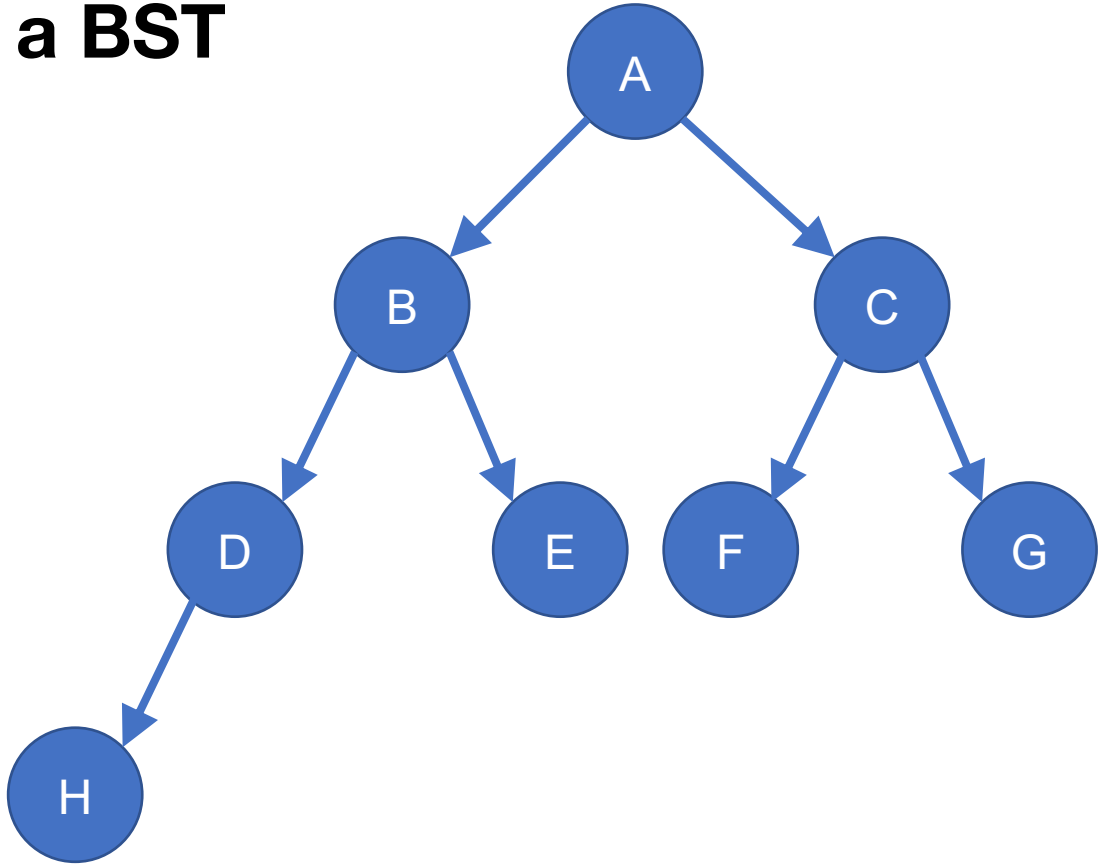
Implementing a binary search tree (BST)

Binary search tree – each node > all nodes in its left subtree and < all nodes in its right subtree

BST



Not a BST



Binary Search Tree Implementation

Two classes:

- Tree
  - Stores a reference to the root
  - The only class we want clients to interact with
- Node
  - Stores data and links to left and right subtree

Node class setup

```
public class Node<T> {
  private T data;
  private Node<T> left;
  private Node<T> right;

}
```

Node class setup

```
public class Node<T> {
  private T data;
  private Node<T> left;
  private Node<T> right;





}
```

Private fields to store the data & links to left & right subtrees (also Nodes)

Node class setup

```
public class Node<T> {
   ... fields ...
   public Node(T data) {
      this(data, null, null);
   }




}
```

Create a default constructor

- Client should not have to worry about connecting nodes

Node class setup

```
public class Node<T> {
   ... fields ...
   public Node(T data) {
      this(data, null, null);
   }
   public Node(T data, Node<T> left, Node<T> right) {
      this.data = data;
      this.left = left;
      this.right = right;
   }

}
```

Calls overloaded constructor with parameters filled in

Node class setup

```
public class Node<T> {
   ... fields ...
   ... constructors ...
   // Getters and setters for each field
}
```

Need public getters/setters that the Tree can access

Tree class setup

```
public class Tree<T extends Comparable<T>> {
  private Node<T> root;

  public Tree() {
    this.root = null;
  }
}
```

Tree class setup

```
public class Tree<T extends Comparable<T>> {
  private Node<T> root;

  public Tree() {
    this.root = null;
  }
}
```

Start with an empty Tree

Tree class setup

```
public class Tree<T extends Comparable<T>> {
  private Node<T> root;

  public Tree() {
    this.root = null;
  }
}
```

Binary Search Tree ADT

- `boolean contains(T x)`
- `T findMin()`
- `T findMax()`
- `void insert(T x)`
- `void remove(T x)`

Template for Tree methods

Often implemented recursively with a public and private version

- Public version only accepts data
  - Calls the private version and passes the node to process

```
public Type methodName(
                <public params>)
{
  this.methodName(root,
                        <params>);
}
```

Template for Tree methods

Often implemented recursively with a public and private version

- Public version only accepts data
- Private version accepts node to process as well

```
public Type methodName(
                <public params>)
{
  this.methodName(root,
                        <params>);
}

private Type methodName(root,
                    <params>) {
  ...
}
```

Example: `contains(T x)` public method

Clients don't care about the tree structure, only what it contains

```
public boolean contains(T x) {
    return this.contains(this.root, x);
}
```

Example: `contains(T x)` public method

Clients don't care about the tree structure, only what it contains

```
public boolean contains(T x) {
    return this.contains(this.root, x);
}
```

Tell the private method where to search from

Example: `contains(T x)` private method

```
private boolean contains(Node<T> node, T x) {
  // Base case – empty node
  if (node == null) { return false;}
}
```

Example: `contains(T x)` private method

```
private boolean contains(Node<T> node, T x) {
  if (node == null) { return false;}
  // Base case – this is what we're looking for
  if (node.getData().equals(x)) { return true;}
}
```

Example: `contains(T x)` private method

```
private boolean contains(Node<T> node, T x) {
  if (node == null) { return false;}
  if (node.getData().equals(x)) { return true;}
  // Recursive case – search left/right subtrees
  return this.contains(node.getLeft(), x) ||
                    this.contains(node.getRight(), x);
}
```

Walkthrough: BST with nested node class

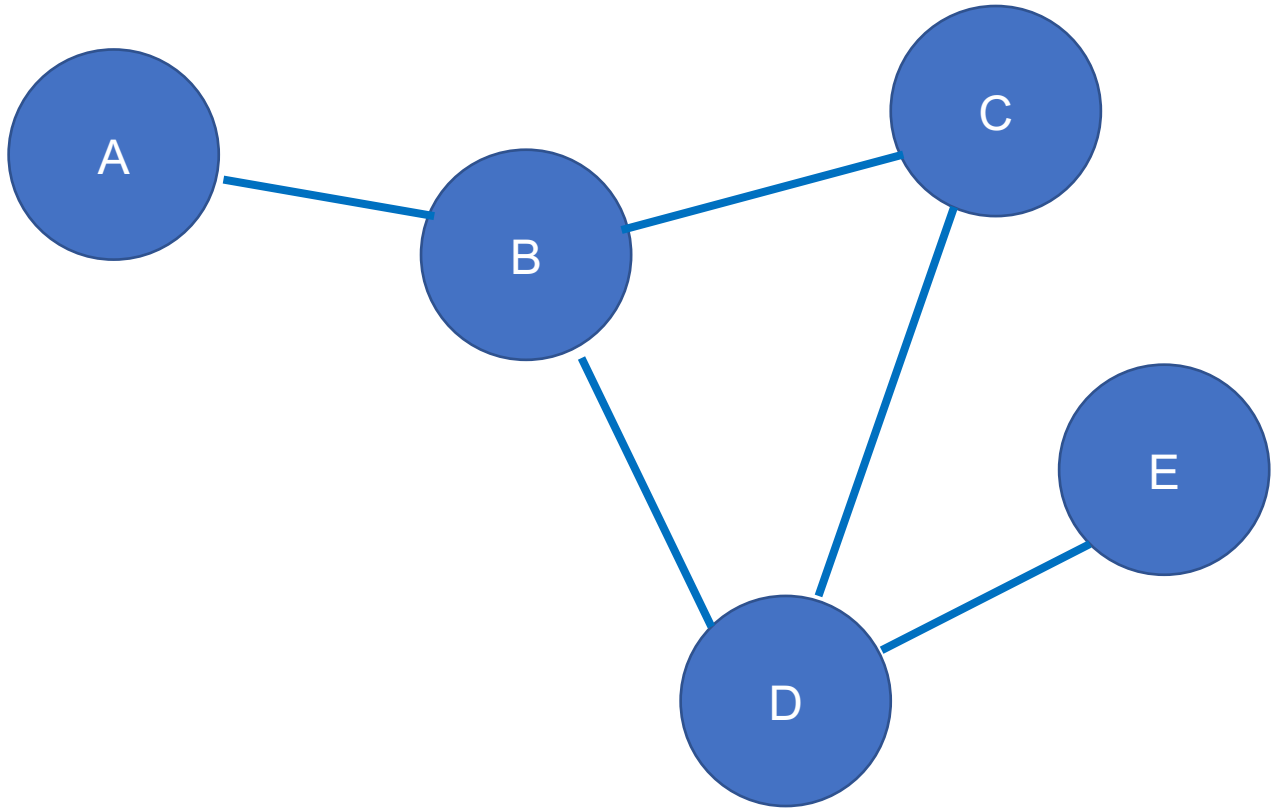
- Uses a private static nested NestedNode class instead of an external node class
- Keeps the nodes encapsulated in the tree
- Your tree does not have to be balanced
  - Existing implementation is not balanced

Graphs in Java

Graphs

- To be covered in depth in 5008
- Quick intro to terminology and key characteristics
- Walkthrough: Java implementation of a generic undirected graph

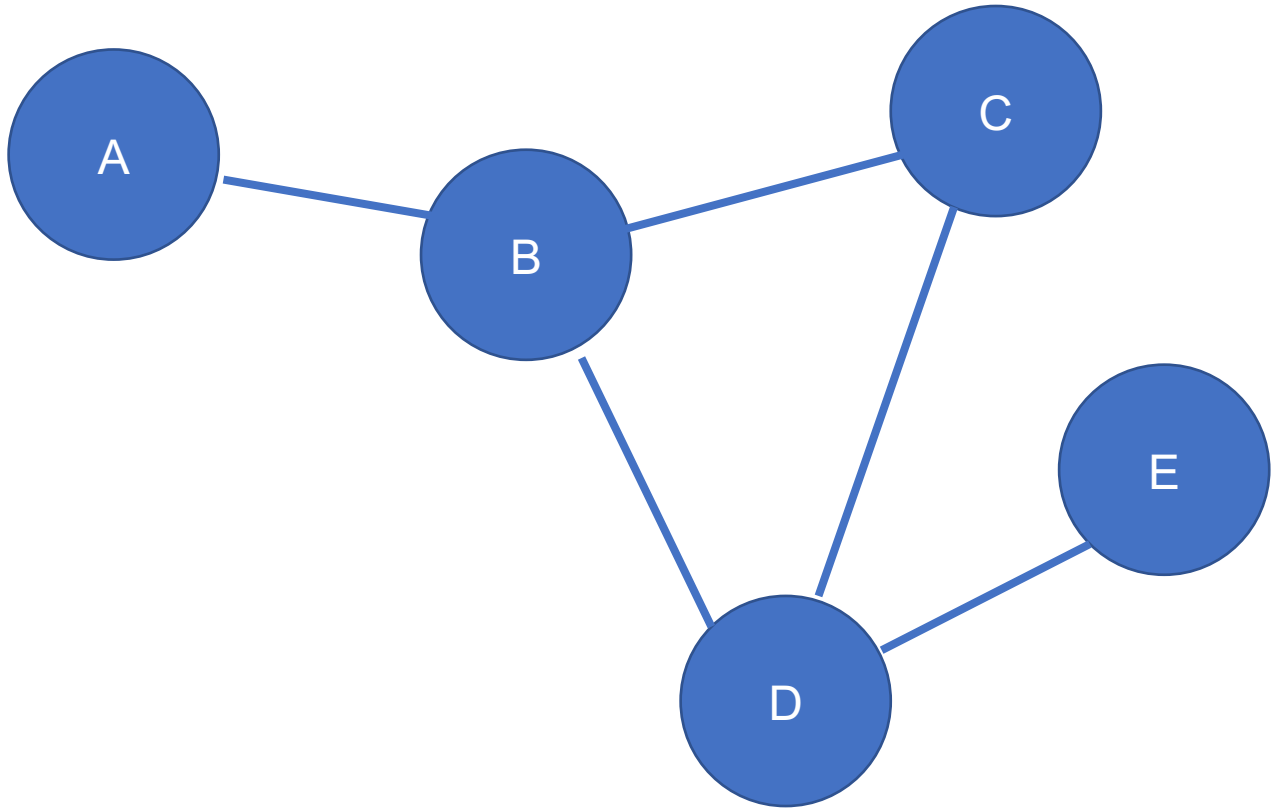
Graph data structure



Made up of:

- A set of vertices/nodes
- A set of edges

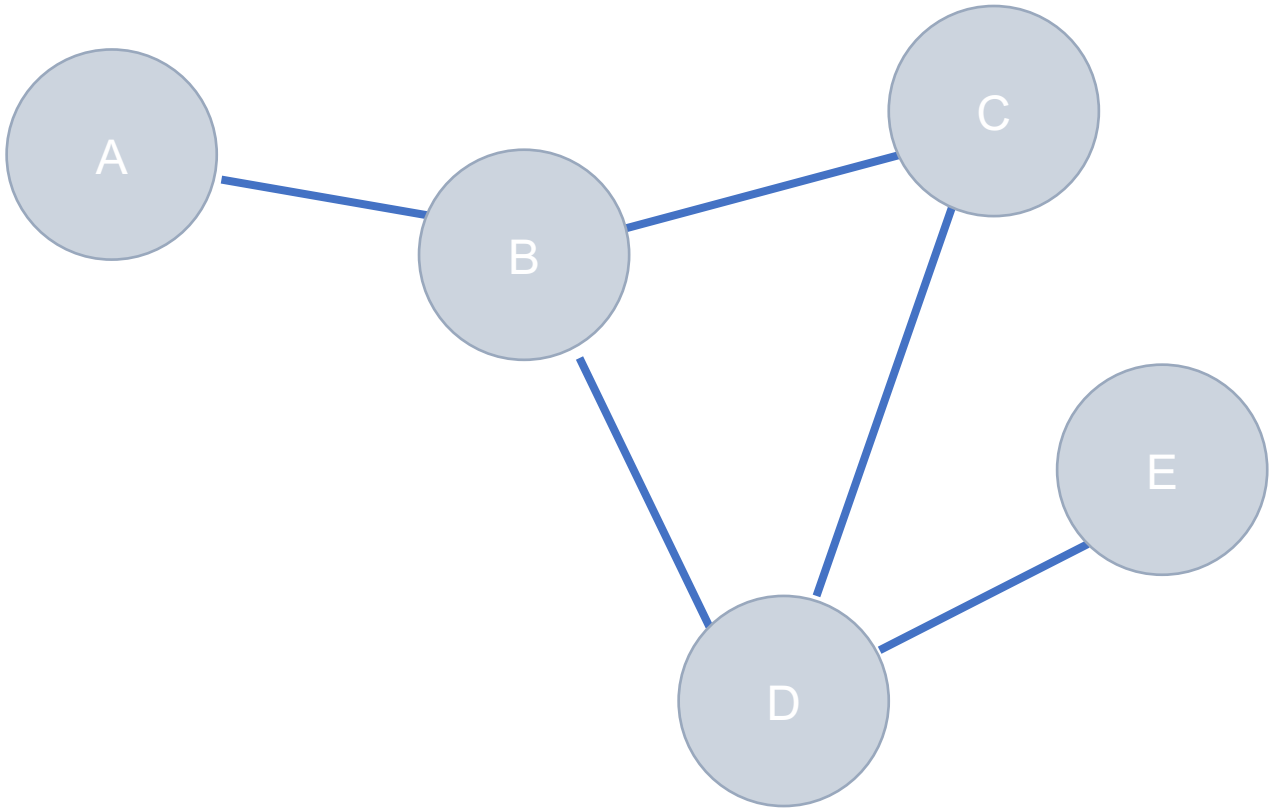
Graph data structure



Graphs are more general than trees

- Every tree is a graph but not every graph is a tree
- Linked lists are graphs too

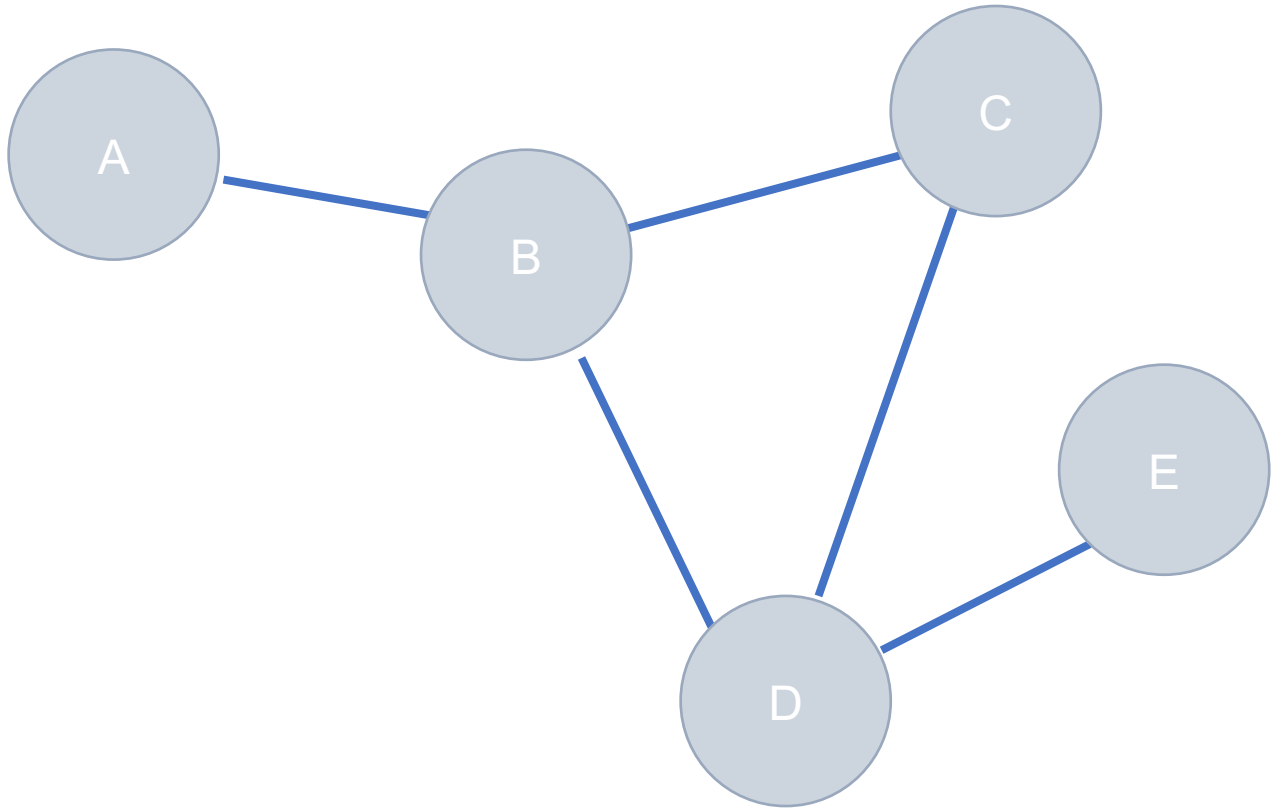
Graph data structure



Edges can be **undirected**:

- Node/vertex order doesn't matter
  - A to B == B to A

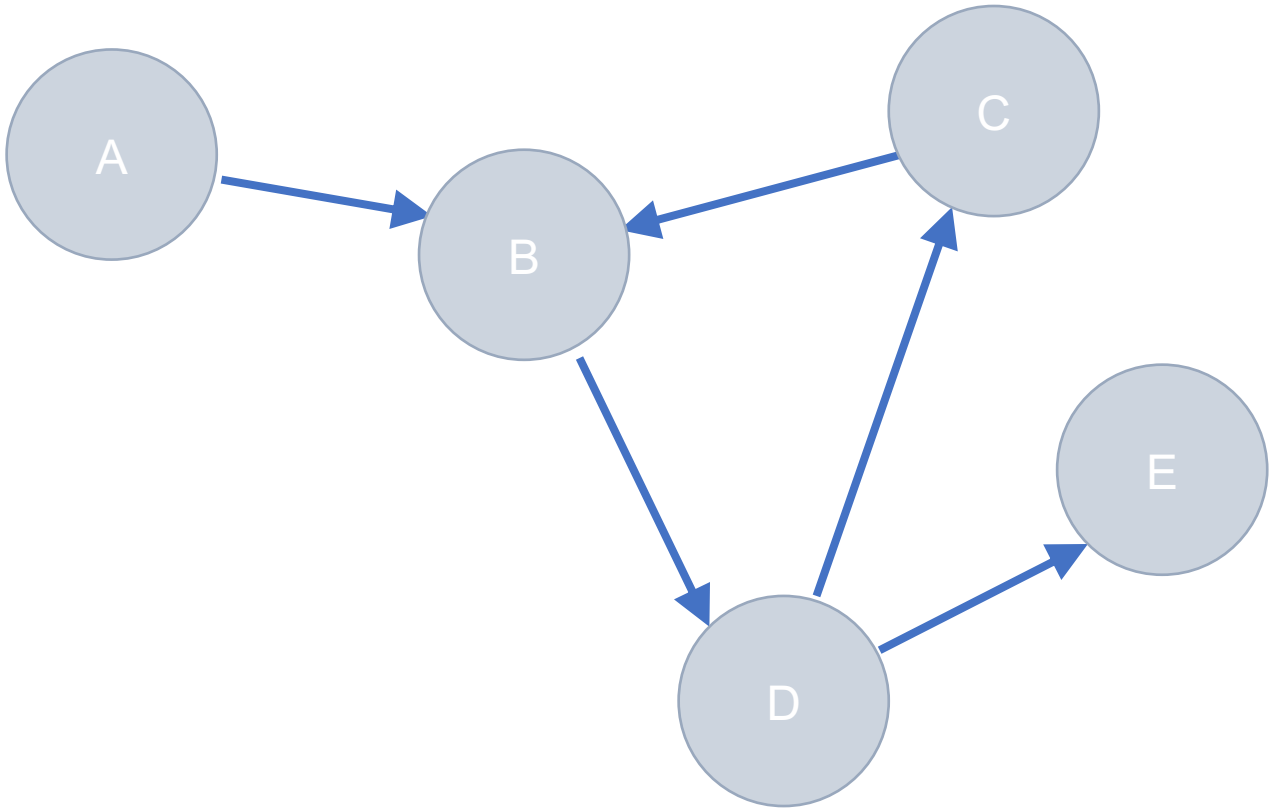
Graph data structure



Edges can be **undirected**:

- Node/vertex order doesn't matter
  - A to B == B to A
- Example application:
  - Social network

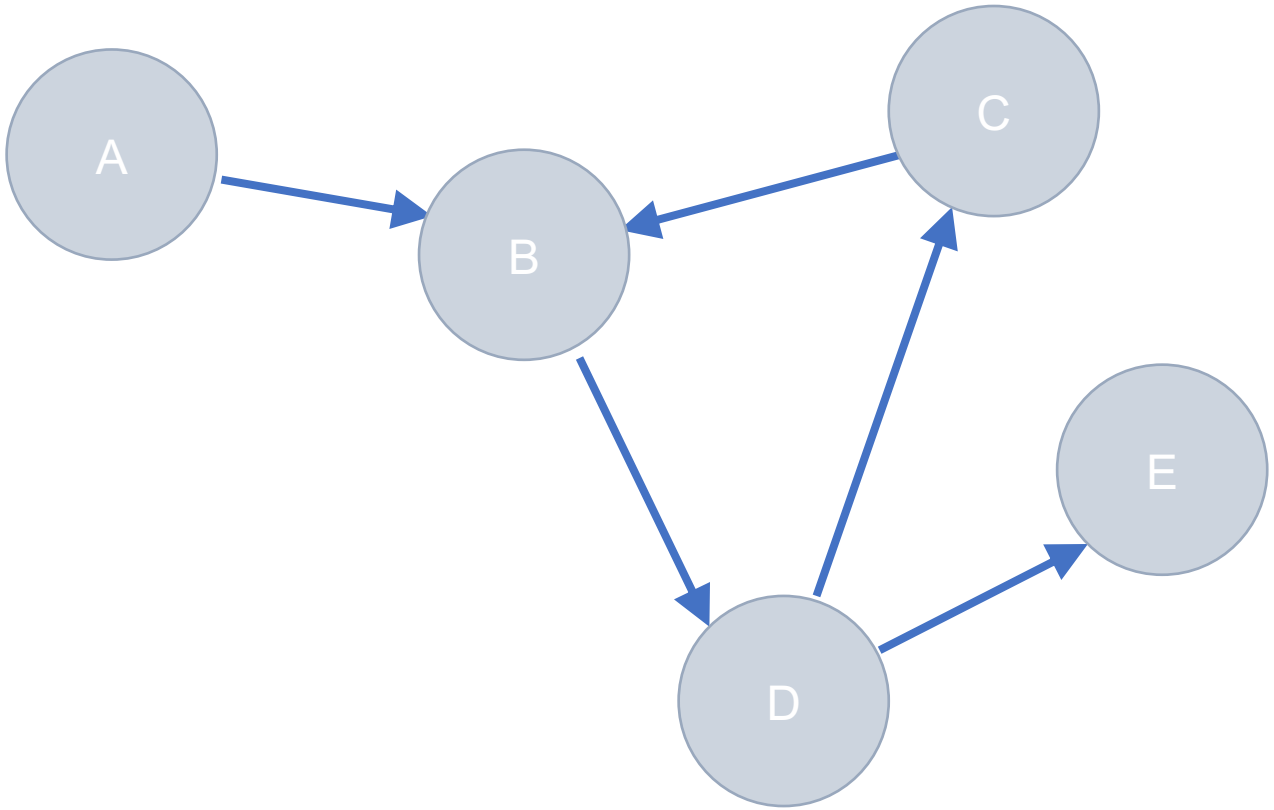
Graph data structure



Edges can be **directed**:

- Node/vertex order <u>does</u> matter
  - A to B but not B to A

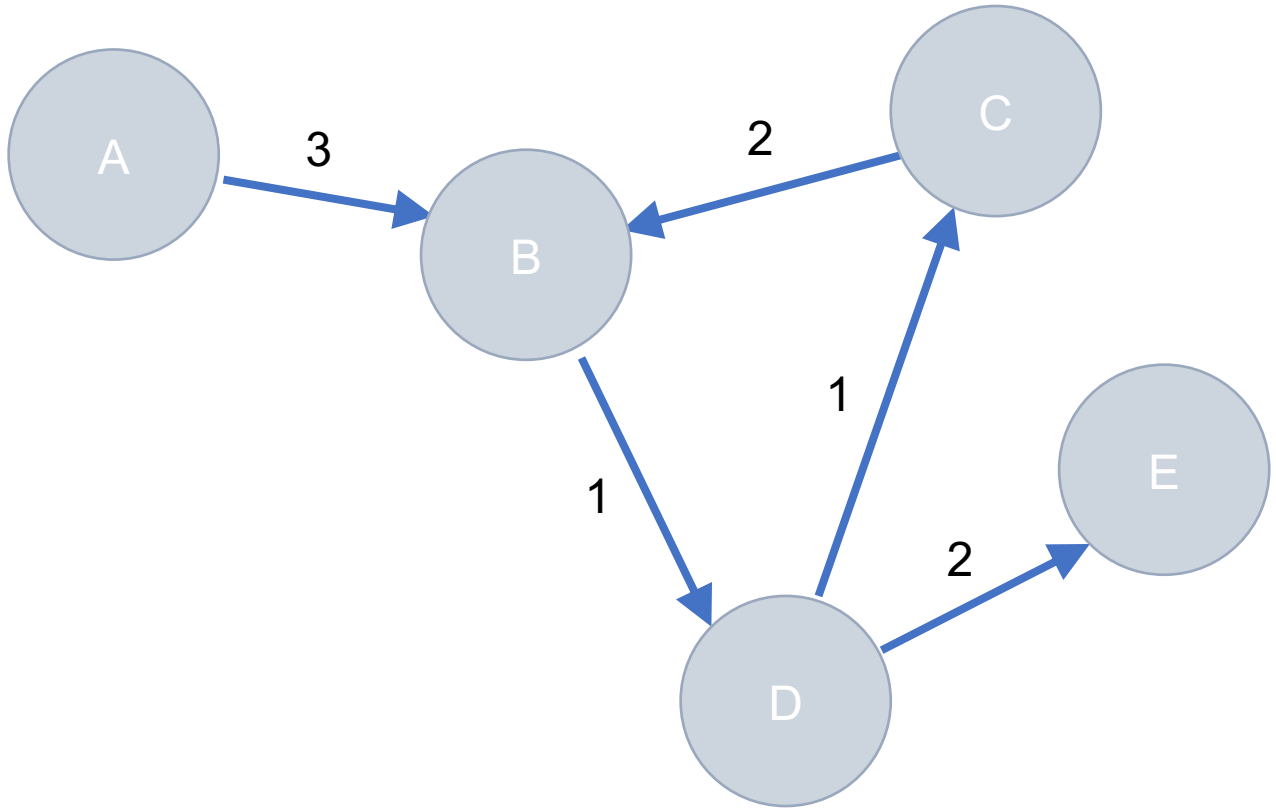
Graph data structure



Edges can be **directed**:

- Node/vertex order <u>does</u> matter
  - A to B but not B to A
- Example applications:
  - Path finding
  - Resource allocation

Graph data structure



Weighted graph:

- Edges have weights
- Directed or undirected
- Example application:
  - Path finding – weights can account for speed limits or traffic

Graph representation

Two primary options:

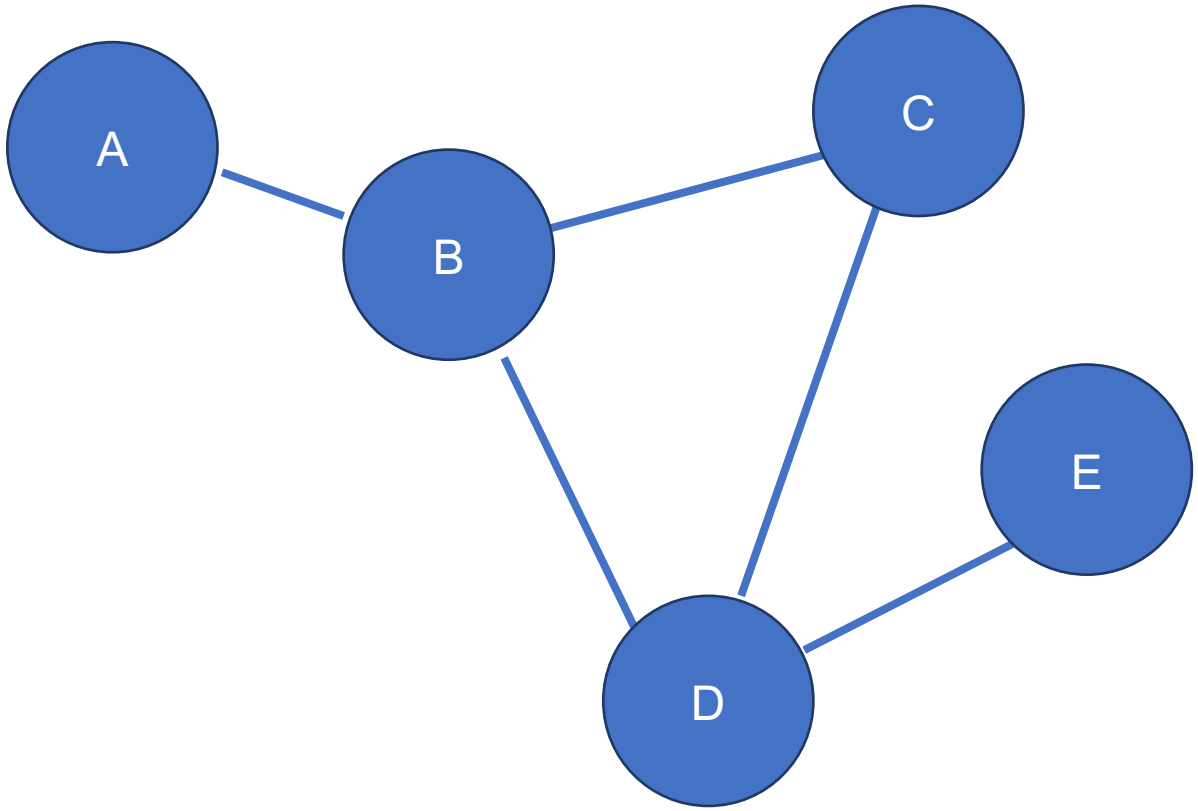
- Adjacency matrix
- Adjacency list

Graph representation

Adjacency Matrix - a 2D array *A*, where:

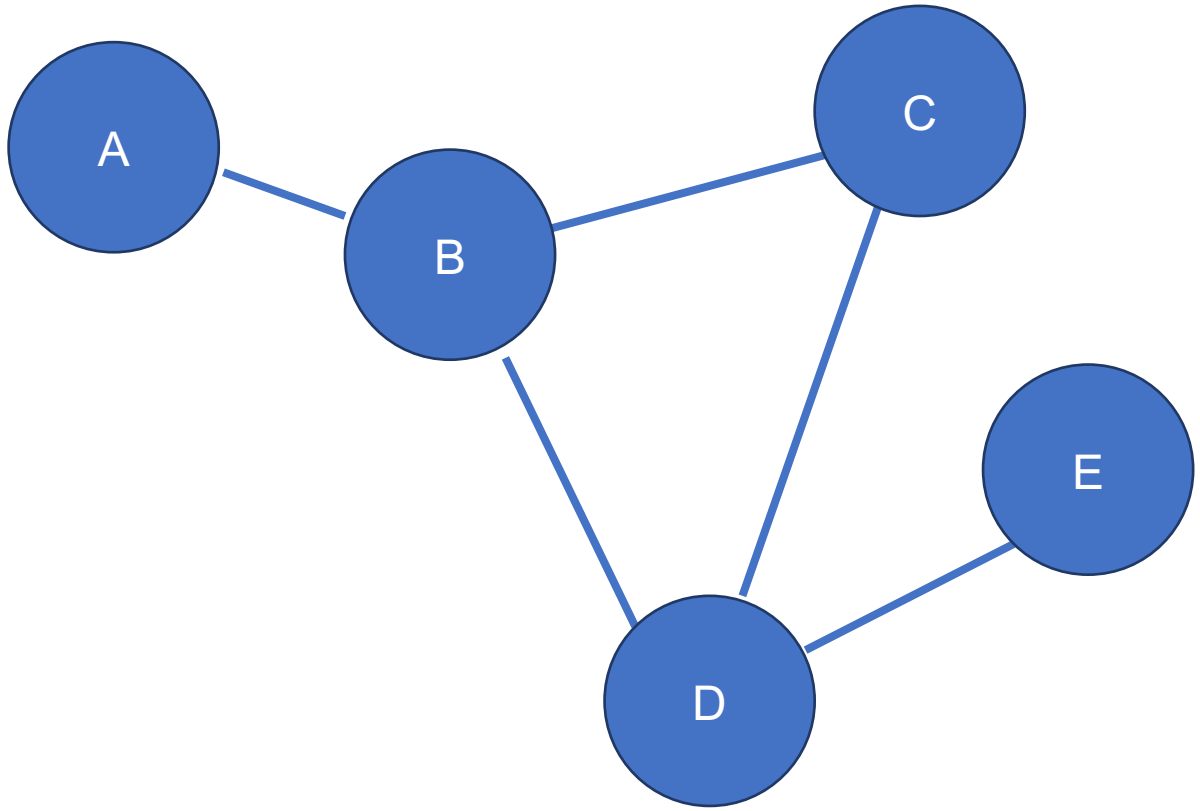
- If some edge *(u, v)* exists, set *A[u, v]* to 1 (true)
  - weighted graph: set to edge weight
- Else, set *A[u,v]* to 0 (false)
- Appropriate representation if a graph is dense
  - Nearly all possible edges are present

Undirected graph – adjacency matrix



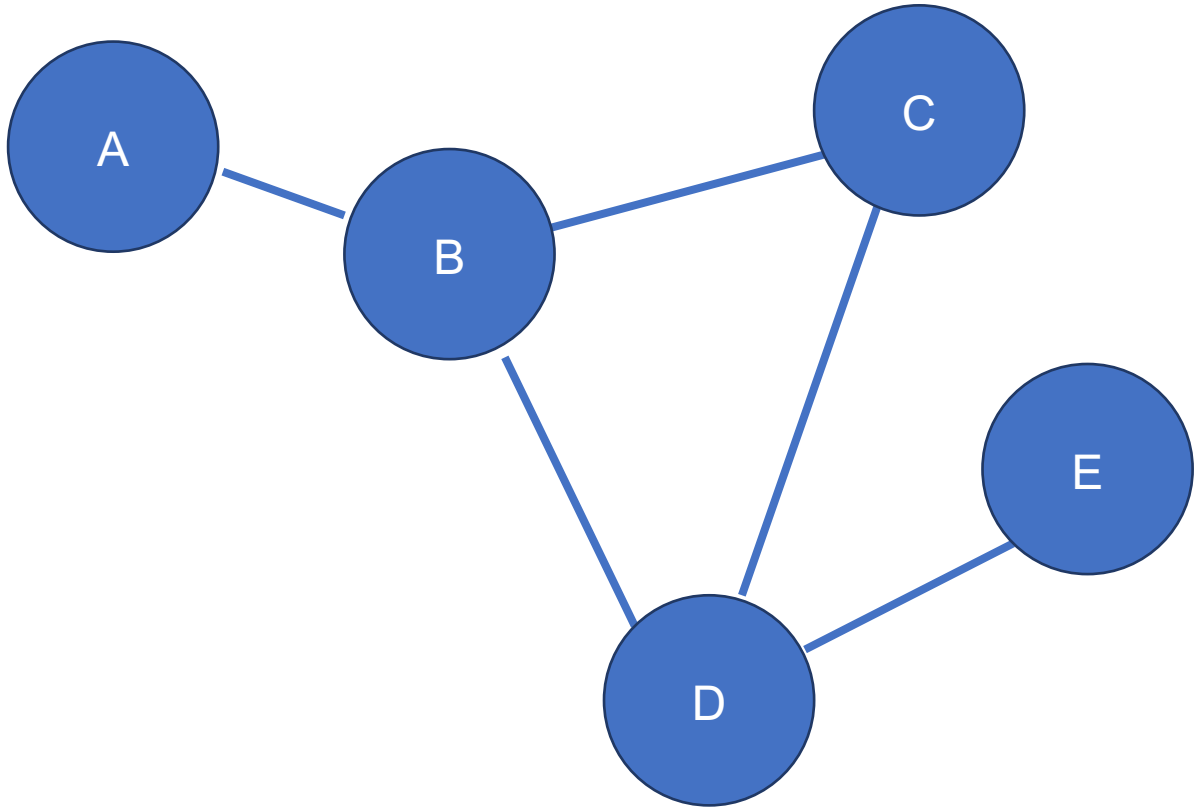
	0 (A)	1 (B)	2 (C)	3 (D)	4 (E)
0 (A)	0	1	0	0	0
1 (B)	1	0	1	1	0
2 (C)	0	1	0	1	0
3 (D)	0	1	1	0	1
4 (E)	0	0	0	1	0

Undirected graph – adjacency matrix



	0 (A)	1 (B)	2 (C)	3 (D)	4 (E)
0 (A)	0	1	0	0	0
1 (B)	1	0	1	1	0
2 (C)	0	1	0	1	0
3 (D)	0	1	1	0	1
4 (E)	0	0	0	1	0

Undirected graph – adjacency matrix

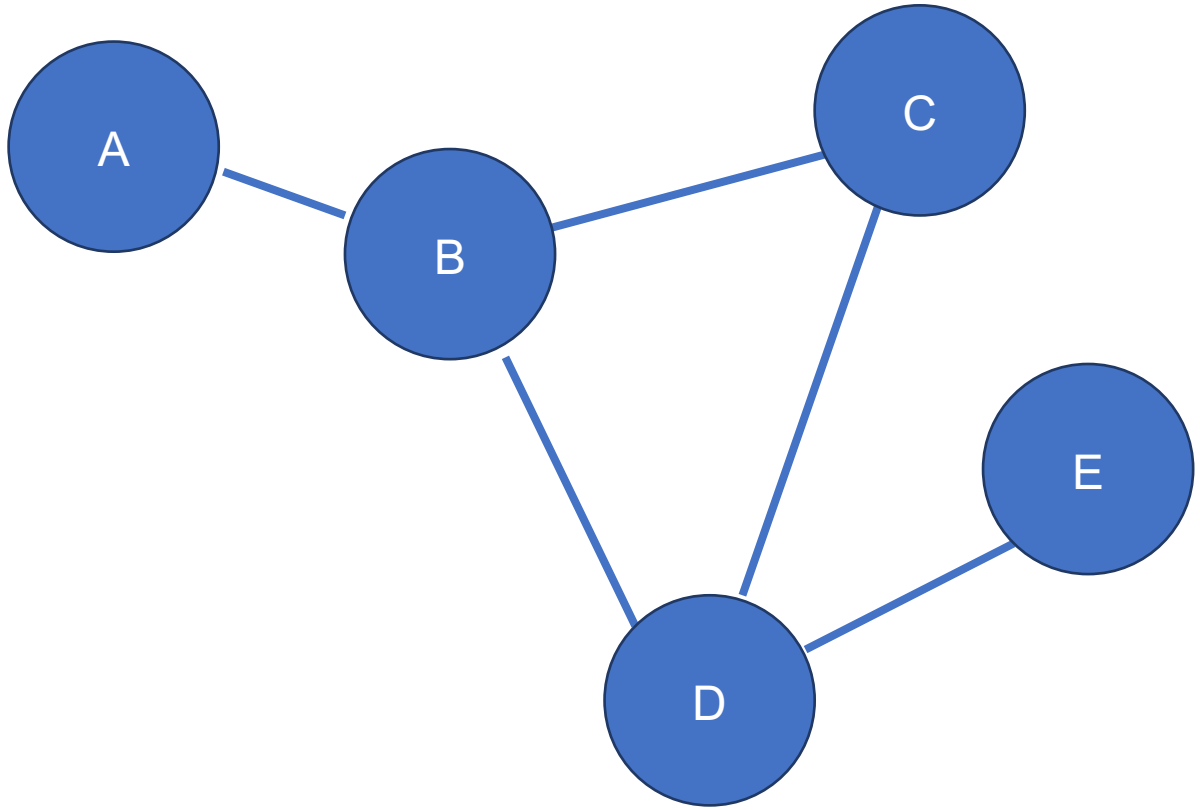


Create a 2D array:

```
int[][] graph
      = new int[5][5];
```

Length of each column = # of nodes/vertices

Undirected graph – adjacency matrix



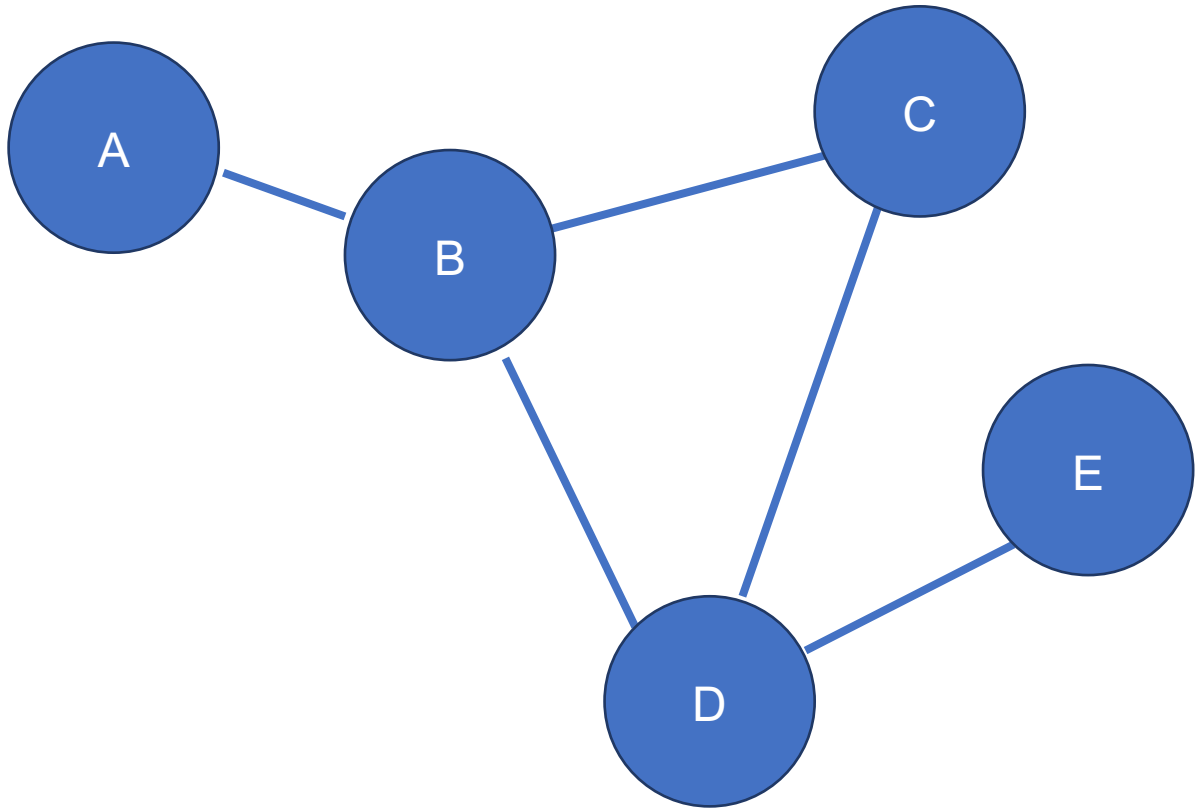
Assuming index 0 = A, 1 = B etc:

Connect A - B (& B – A)

```
graph[0][1] = 1;
graph[1][0] = 1;
```

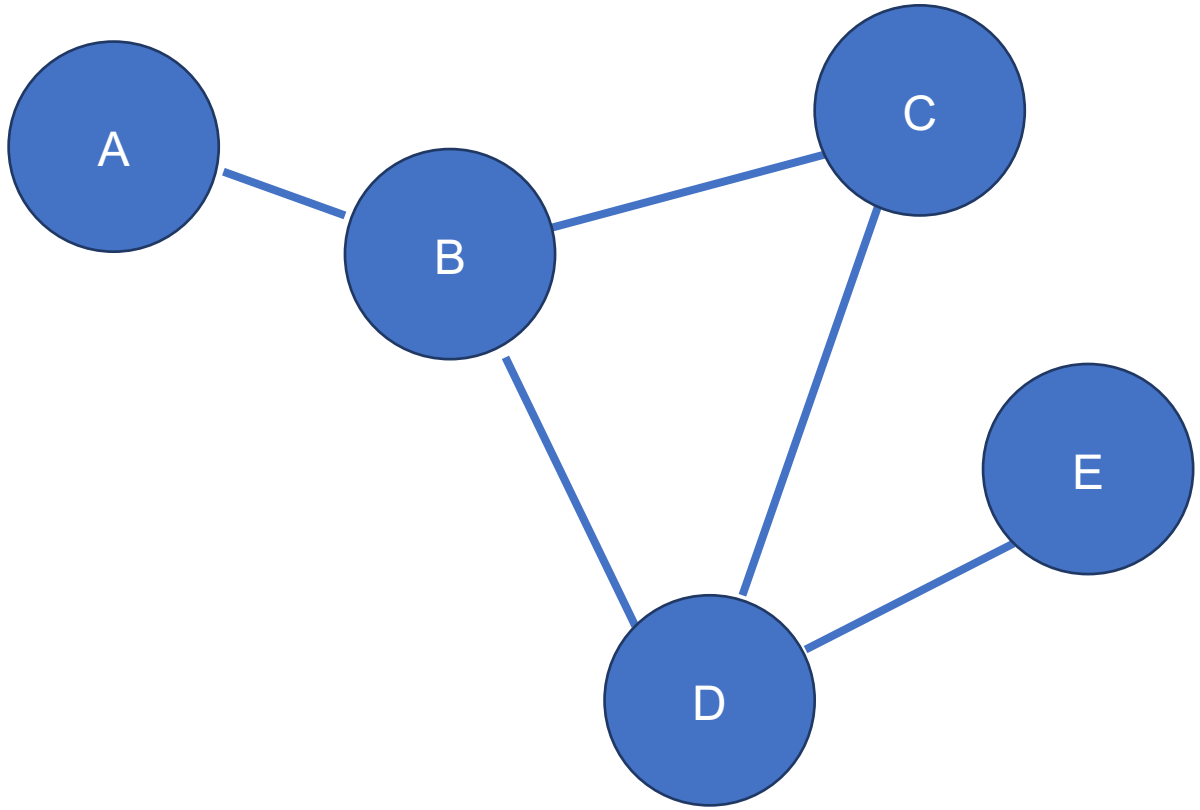
Connections go both ways

Undirected graph – adjacency matrix



	0 (A)	1 (B)	2 (C)	3 (D)	4 (E)
0 (A)	0	**1**	0	0	0
1 (B)	**1**	0	1	1	0
2 (C)	0	1	0	1	0
3 (D)	0	1	1	0	1
4 (E)	0	0	0	1	0

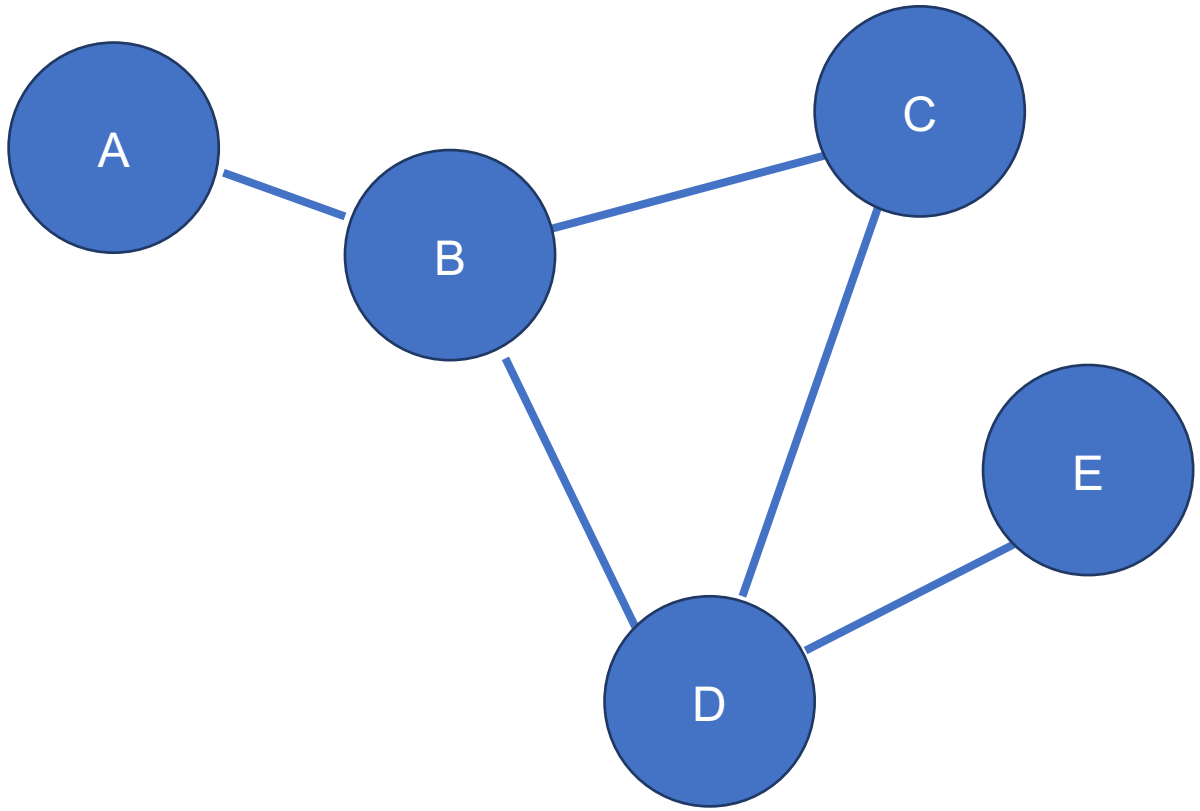
Undirected graph – adjacency matrix



Nodes are not connected to themselves (unless there's a loop)

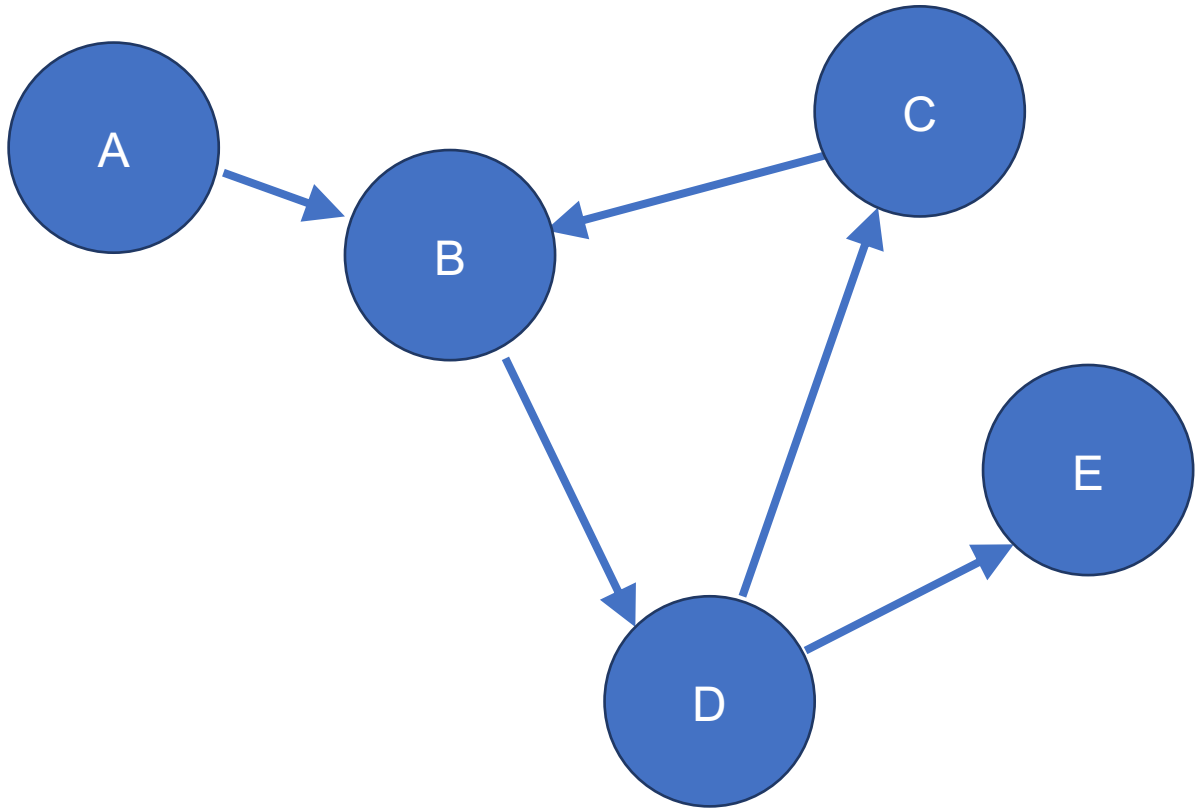
```
graph[0][0] = 0;
```

Undirected graph – adjacency matrix



	0 (A)	1 (B)	2 (C)	3 (D)	4 (E)
0 (A)	**0**	1	0	0	0
1 (B)	1	**0**	1	1	0
2 (C)	0	1	**0**	1	0
3 (D)	0	1	1	**0**	1
4 (E)	0	0	0	1	**0**

Directed graph – adjacency matrix



Connect in the arrow direction:

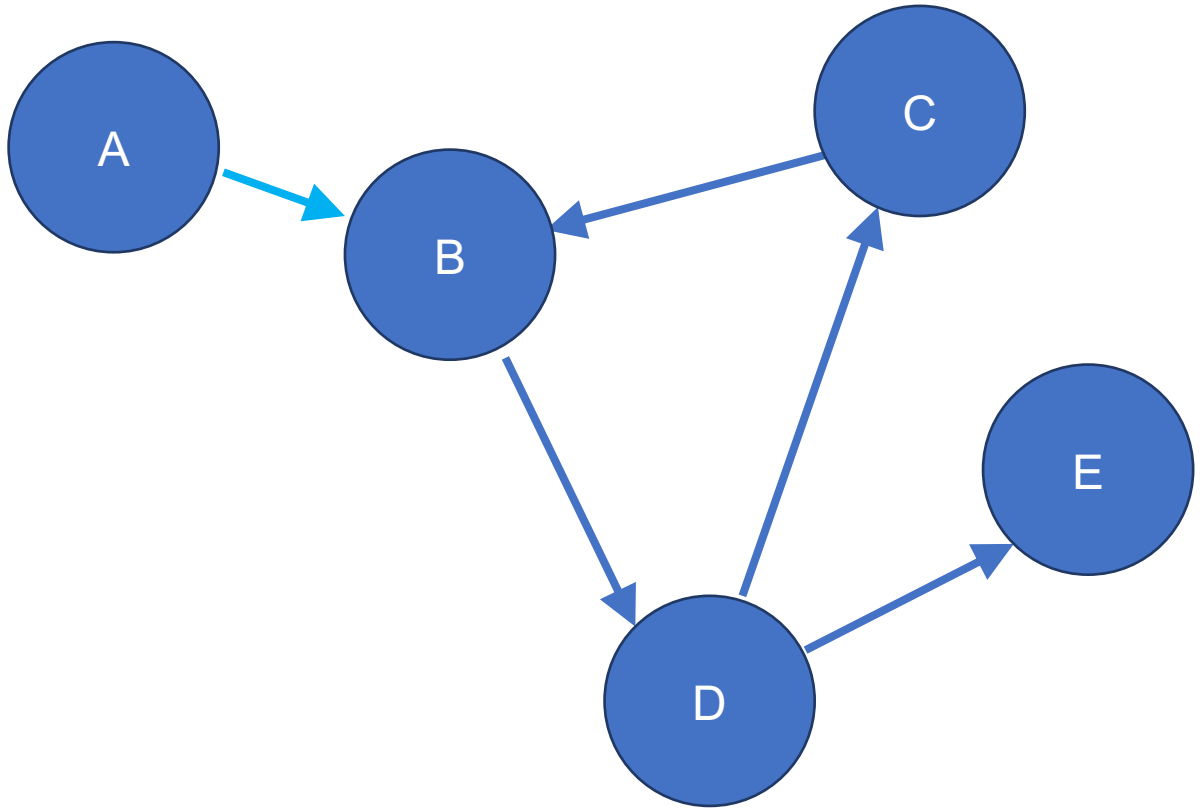
Connect A to B

```
graph[0][1] = 1;
```

But not B to A

```
graph[1][0] = 0;
```

Directed graph – adjacency matrix



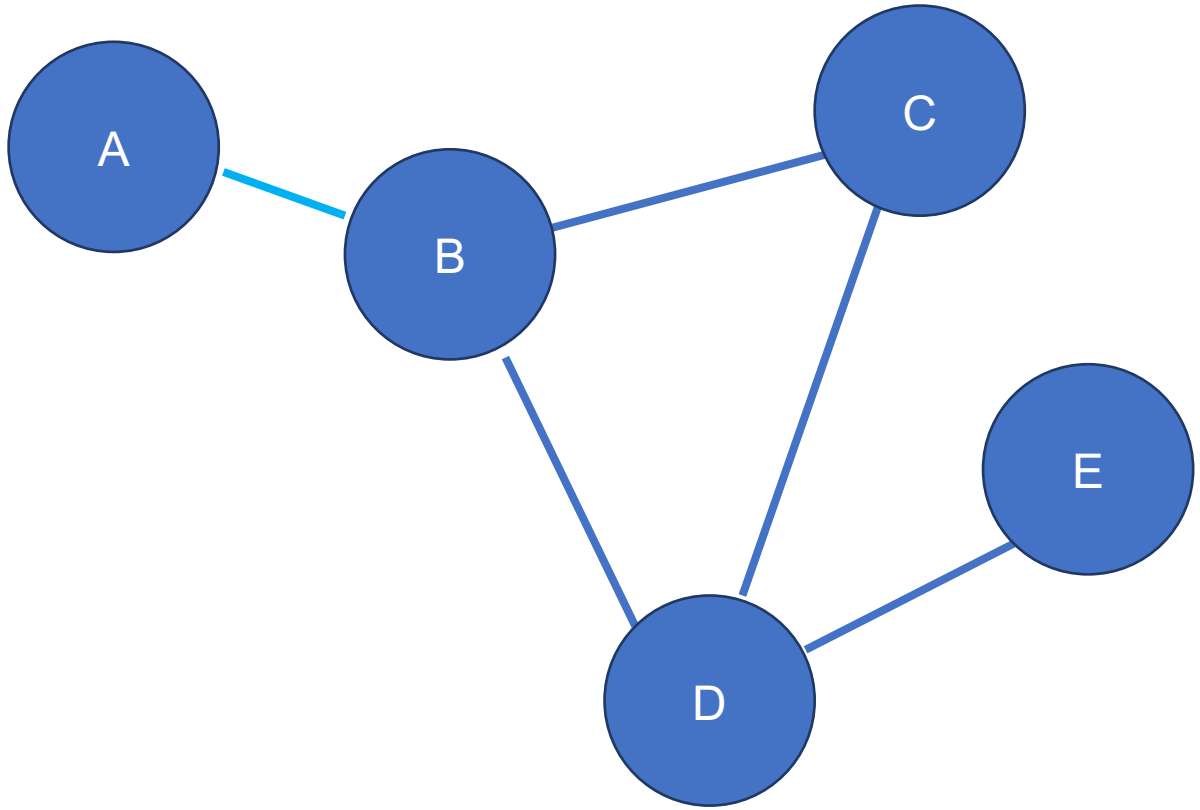
	0 (A)	1 (B)	2 (C)	3 (D)	4 (E)
0 (A)	0	1	0	0	0
1 (B)	0	0	0	1	0
2 (C)	0	1	0	0	0
3 (D)	0	0	1	0	1
4 (E)	0	0	0	0	0

Graph representation

Adjacency List

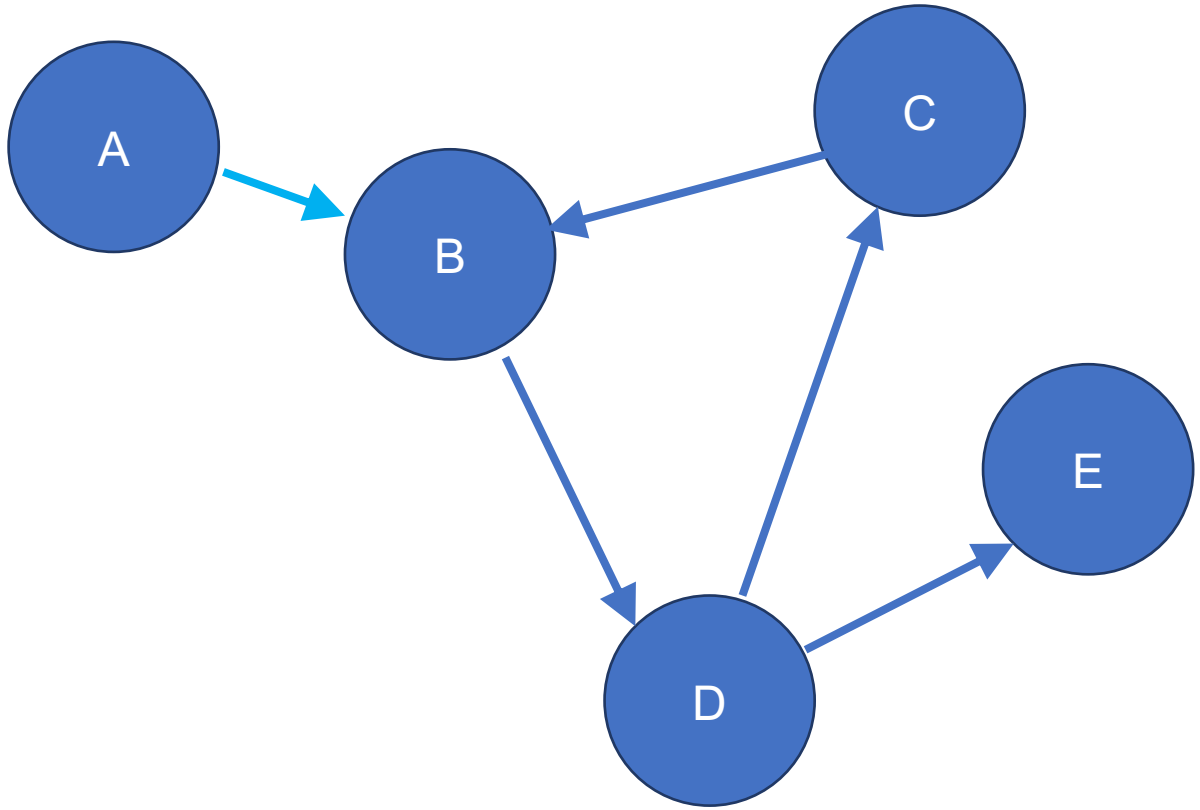
- For each node/vertex, keep a list of all adjacent vertices
- Can be implemented using various data collections (ArrayList, LinkedList, HashSet)
- Appropriate for sparse graphs

Undirected graph – adjacency list



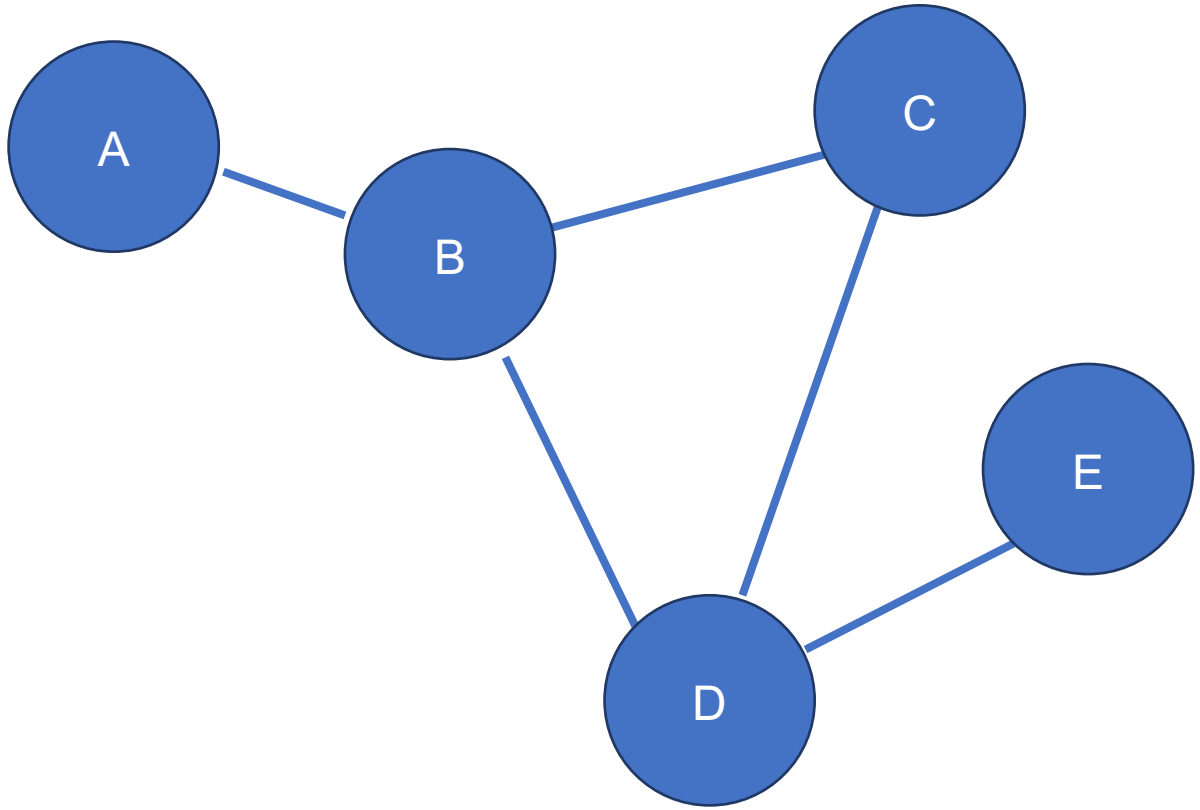
A	**B**
B	**A**, C, D
C	B, D
D	B, C, E
E	D

Directed graph – adjacency list



A	B
B	D
C	B
D	C, E
E	

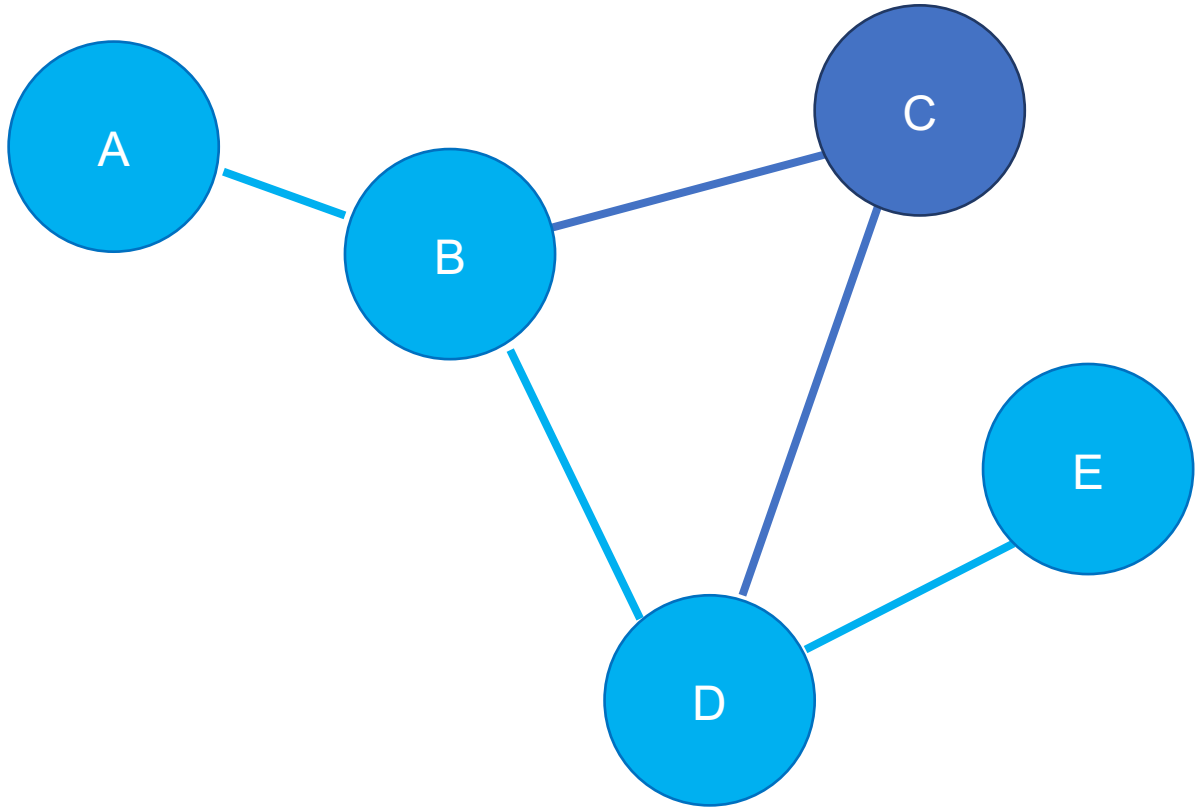
Paths in graphs



A set of edges connecting two vertices

- Neither edges nor vertices repeat
- Path length – the number of edges in the path

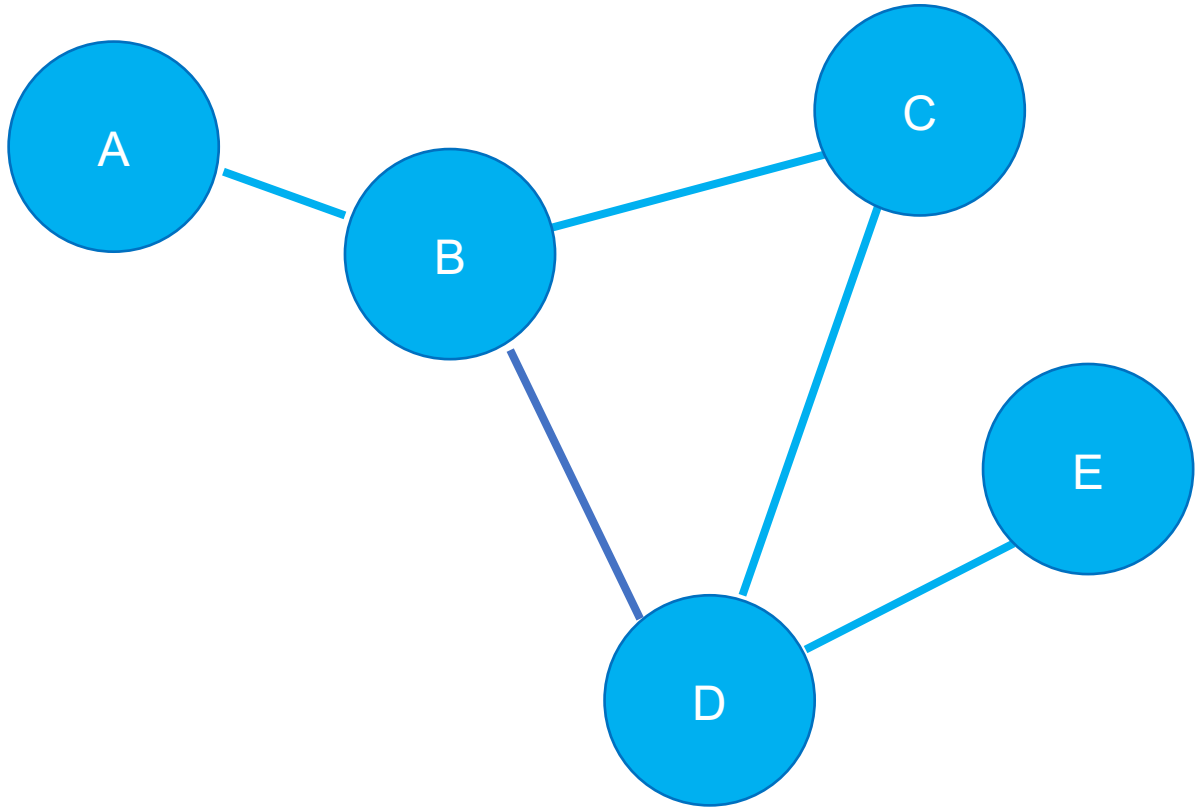
Paths in graphs – undirected example



Path from A to E (or E to A)

- Option 1: length 3

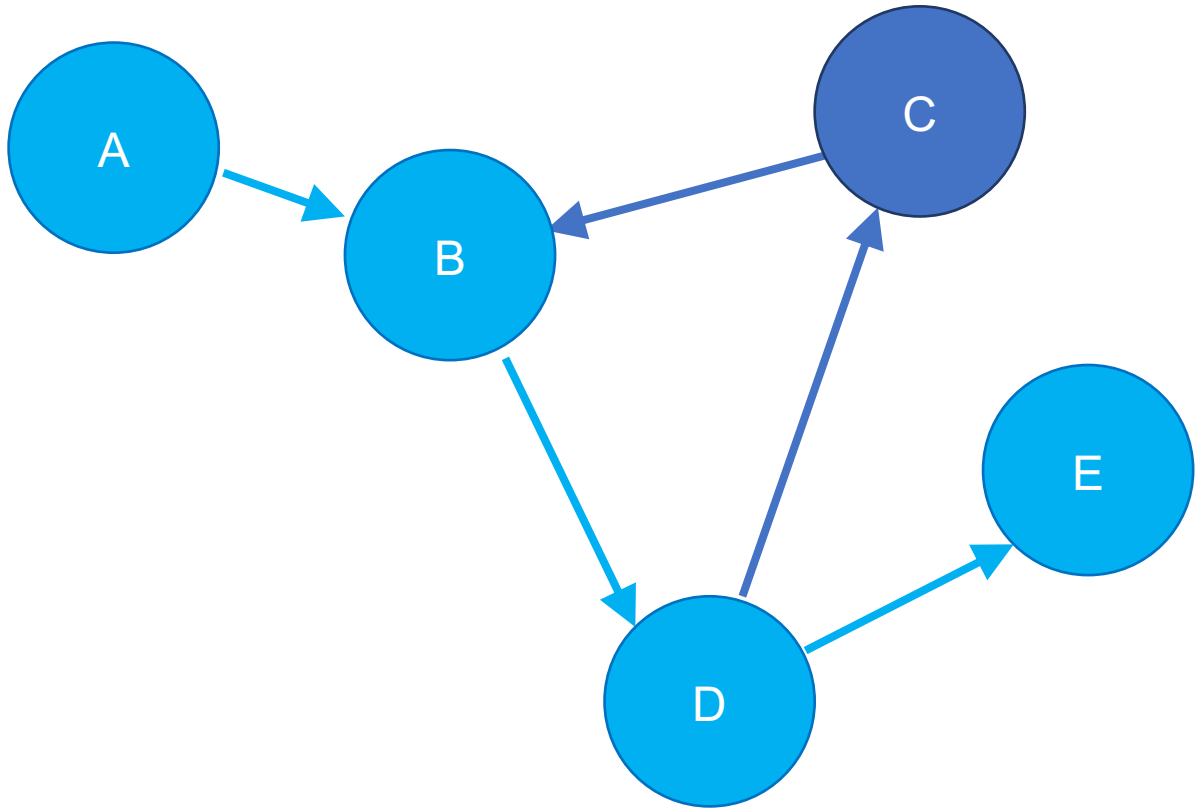
Paths in graphs – undirected example



Path from A to E (or E to A)

- Option 1: length 3
- Option 2: length 4

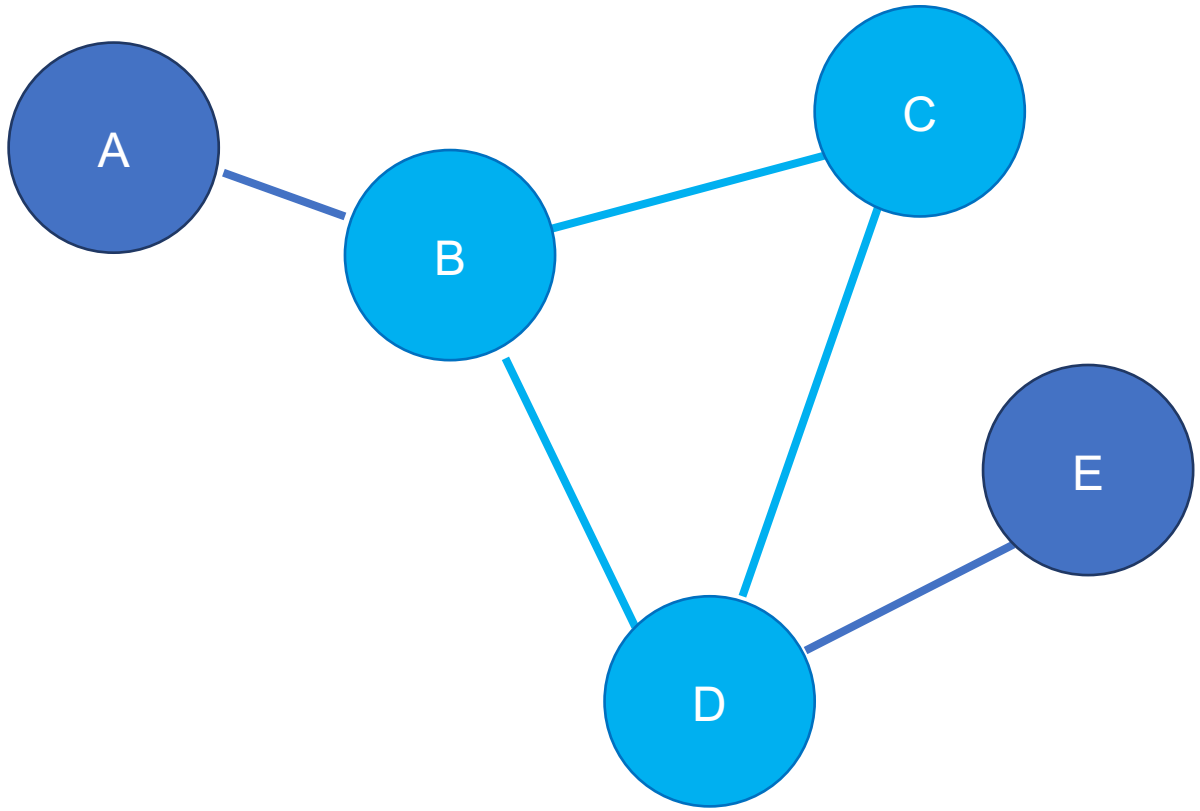
Paths in graphs – directed example



Path from A to E

- Only one option: length 3
- Can’t go from E to A

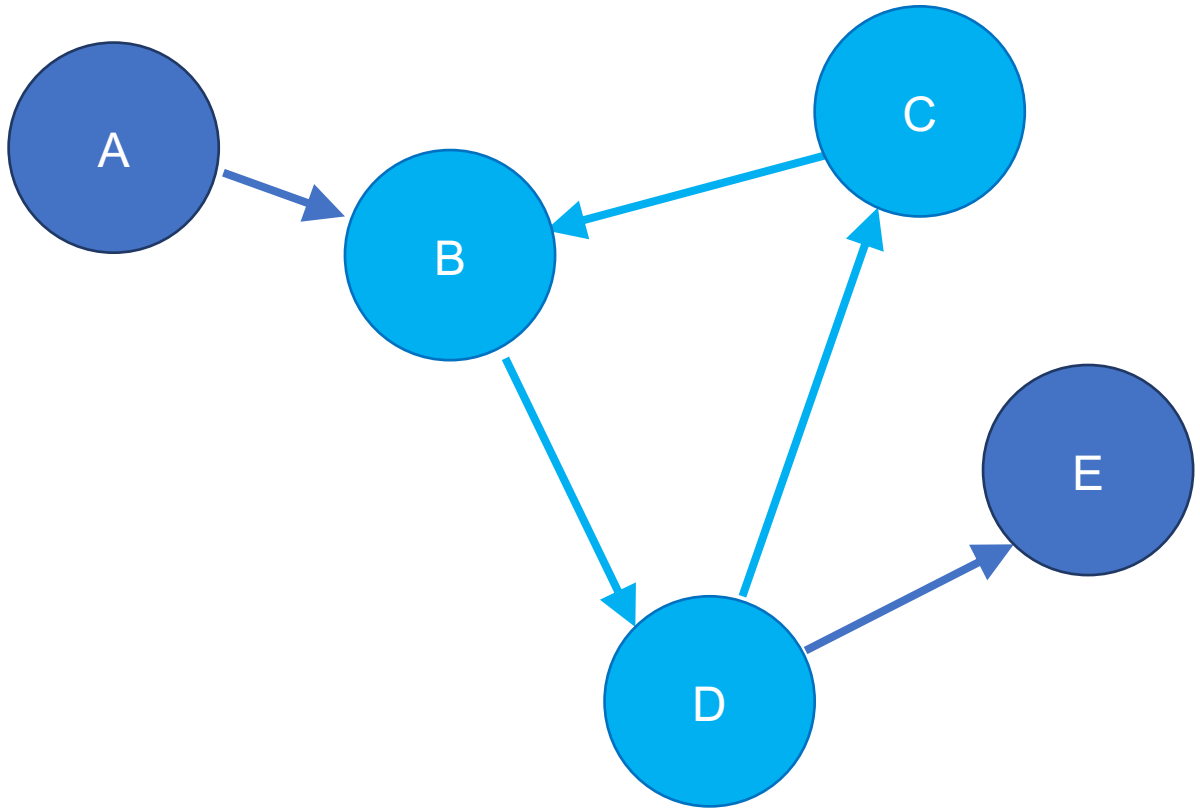
Cycles in graphs



A set of edges that starts and ends on the same vertex:

- B – C – D - B

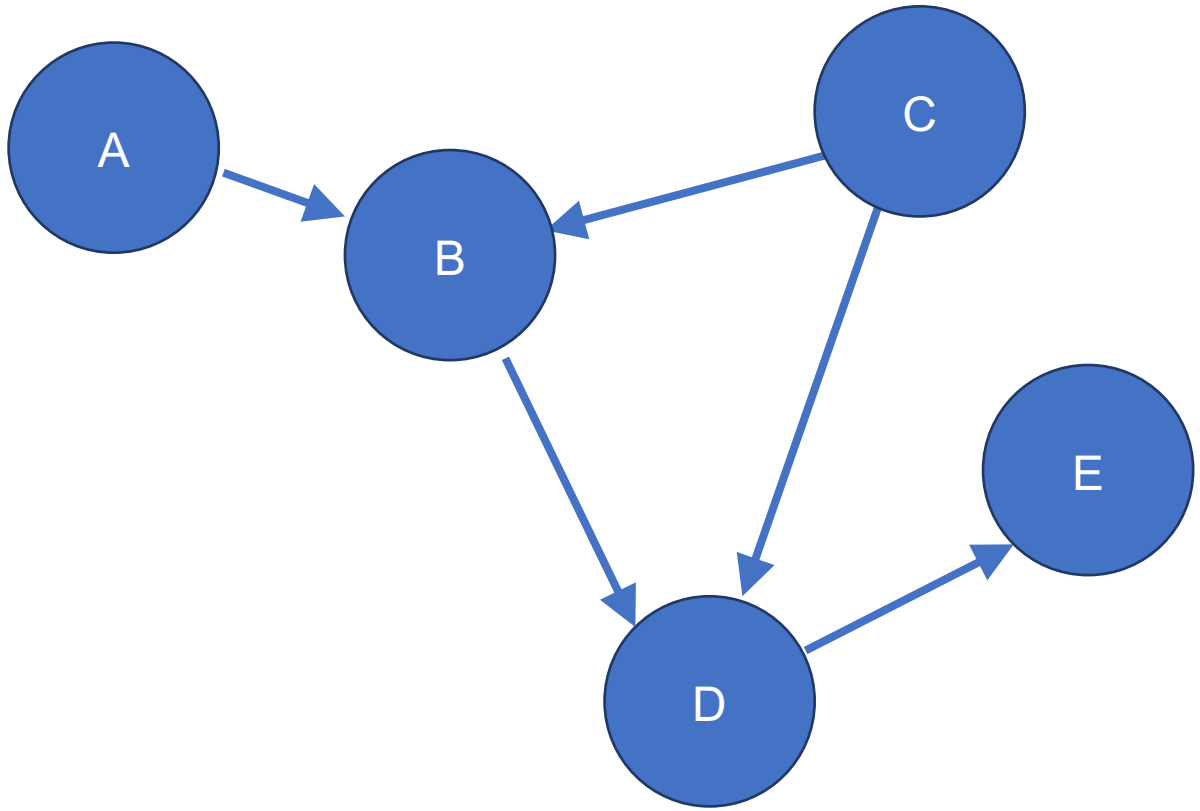
Cycles in graphs



A set of edges that starts and ends on the same vertex

- B – D – C - B

Cycles in graphs



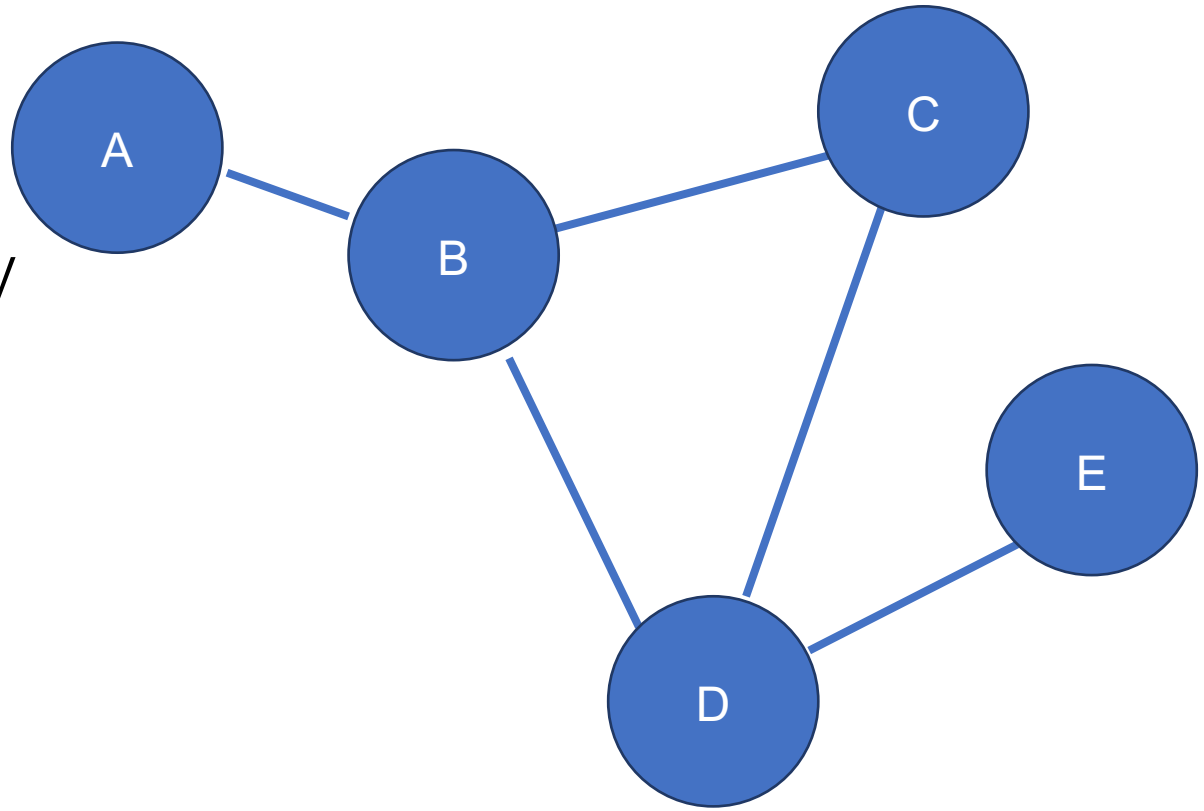
A set of edges that starts and ends on the same vertex

- Now the graph has no cycles…
- **Directed acyclic graph** (DAG)

Connected and complete graphs

Connected graph

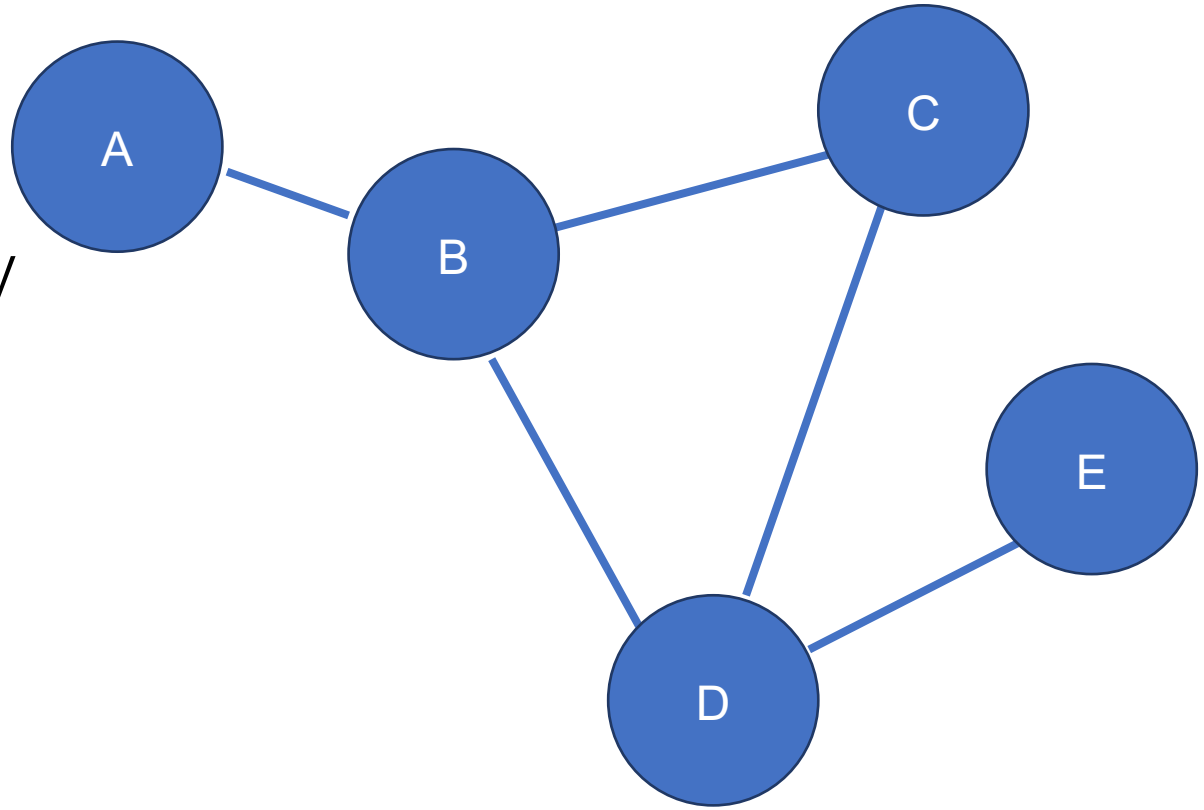
- undirected
- path from every vertex to every other vertex
- *Is this graph connected?*



Connected and complete graphs

Connected graph

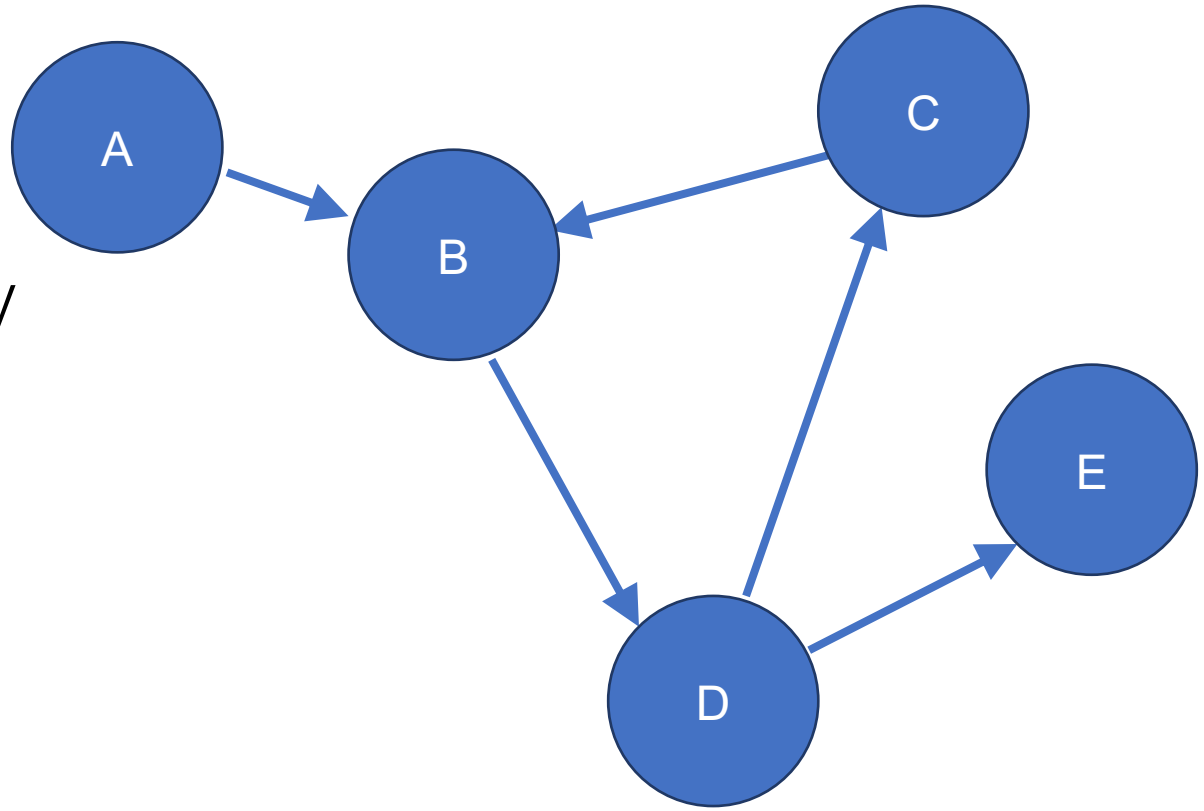
- undirected
- path from every vertex to every other vertex
- *Is this graph connected?*
  - *Yes*



Connected and complete graphs

Strongly connected graph

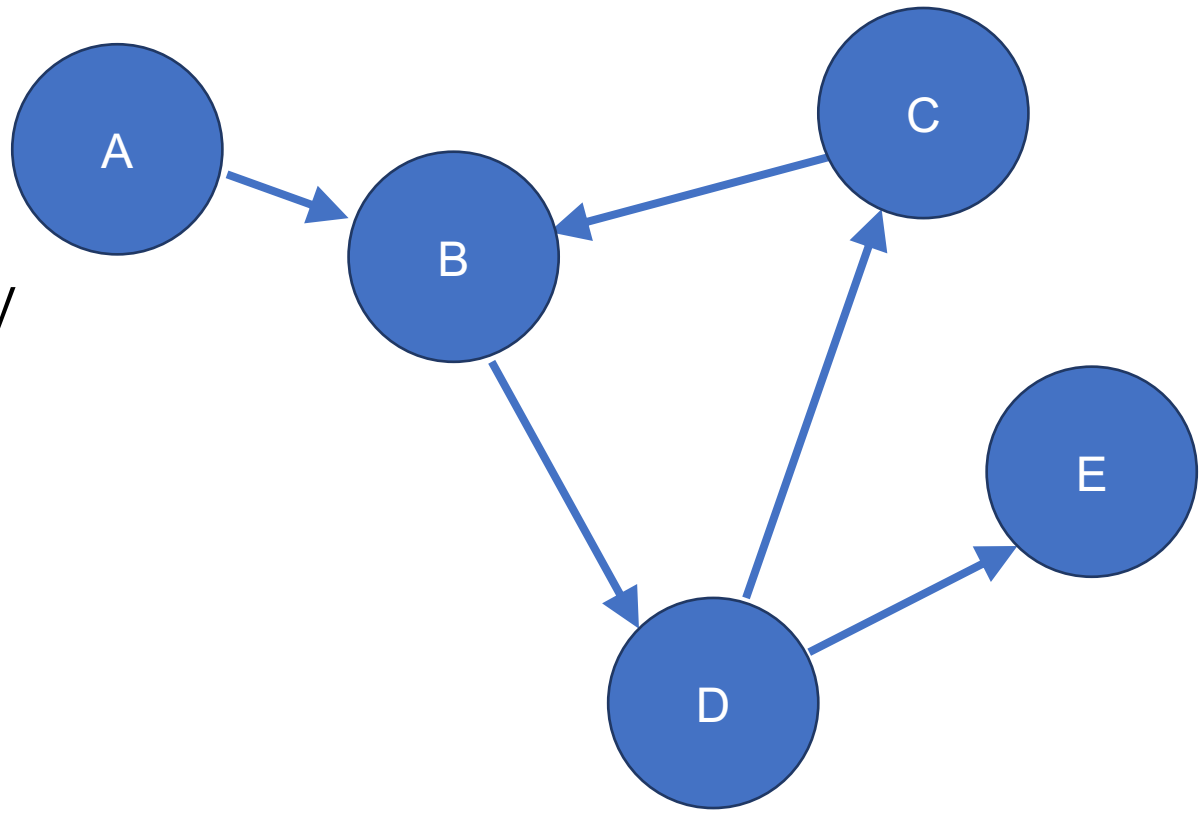
- directed
- path from every vertex to every other vertex
- *Is this graph strongly connected?*



Connected and complete graphs

Strongly connected graph

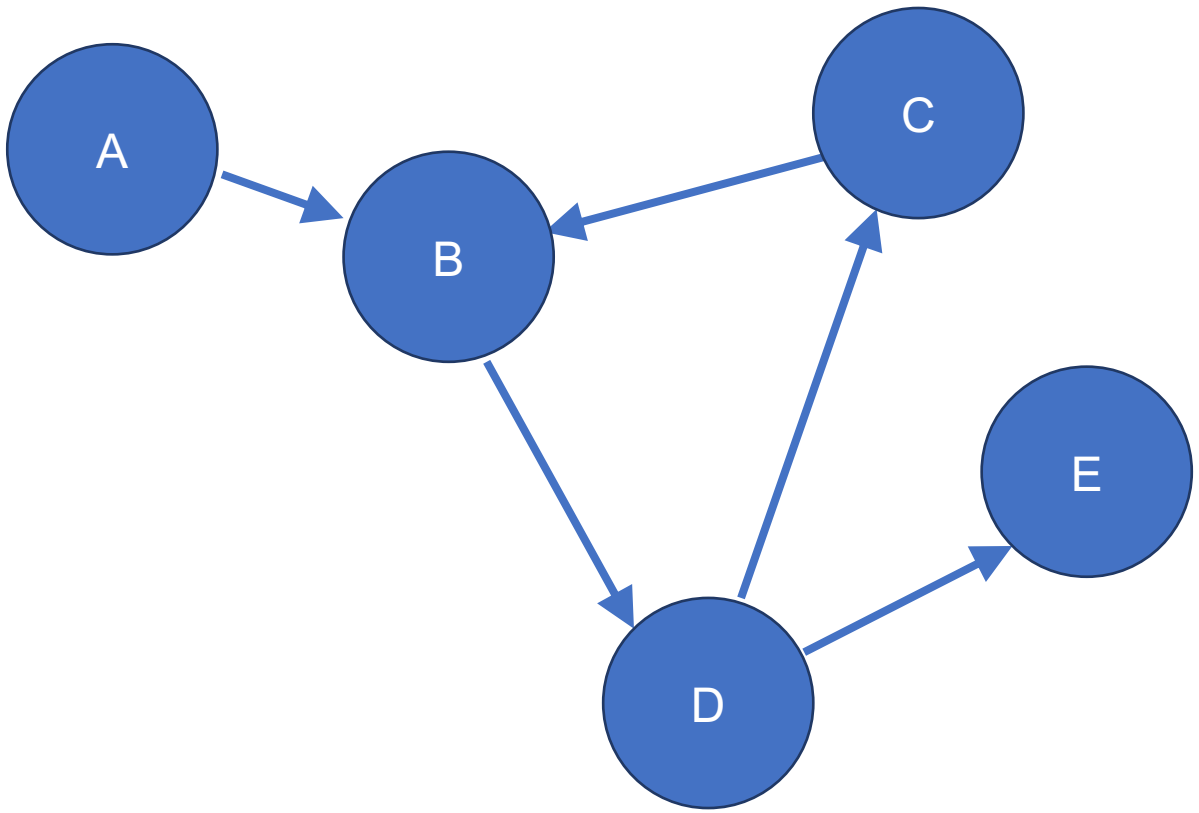
- directed
- path from every vertex to every other vertex
- *Is this graph strongly connected?*
  - *No. Many paths missing:*
    - *Any vertex to A*
    - *E to any vertex*



Connected and complete graphs

Weakly connected graph

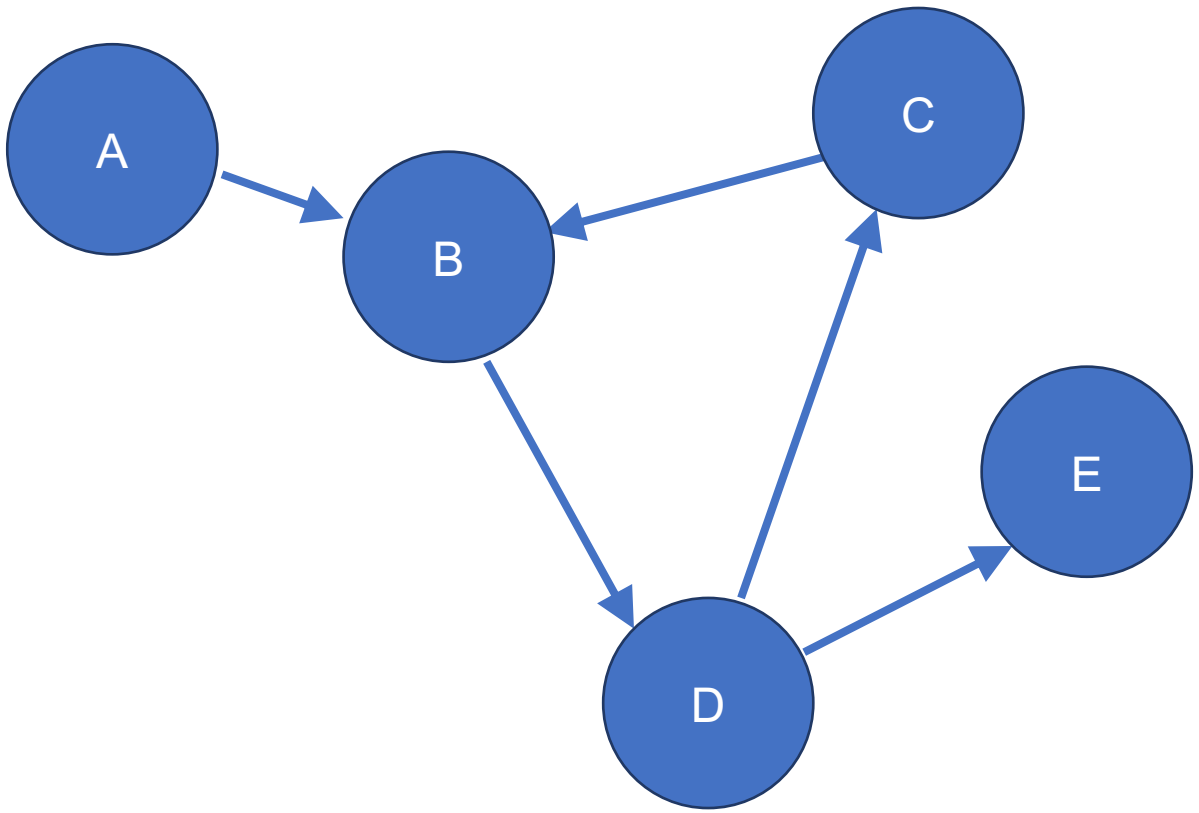
- directed
- not strongly connected but its underlying undirected graph (i.e. without the arrows) is connected
- *Is this graph weakly connected?*



Connected and complete graphs

Weakly connected graph

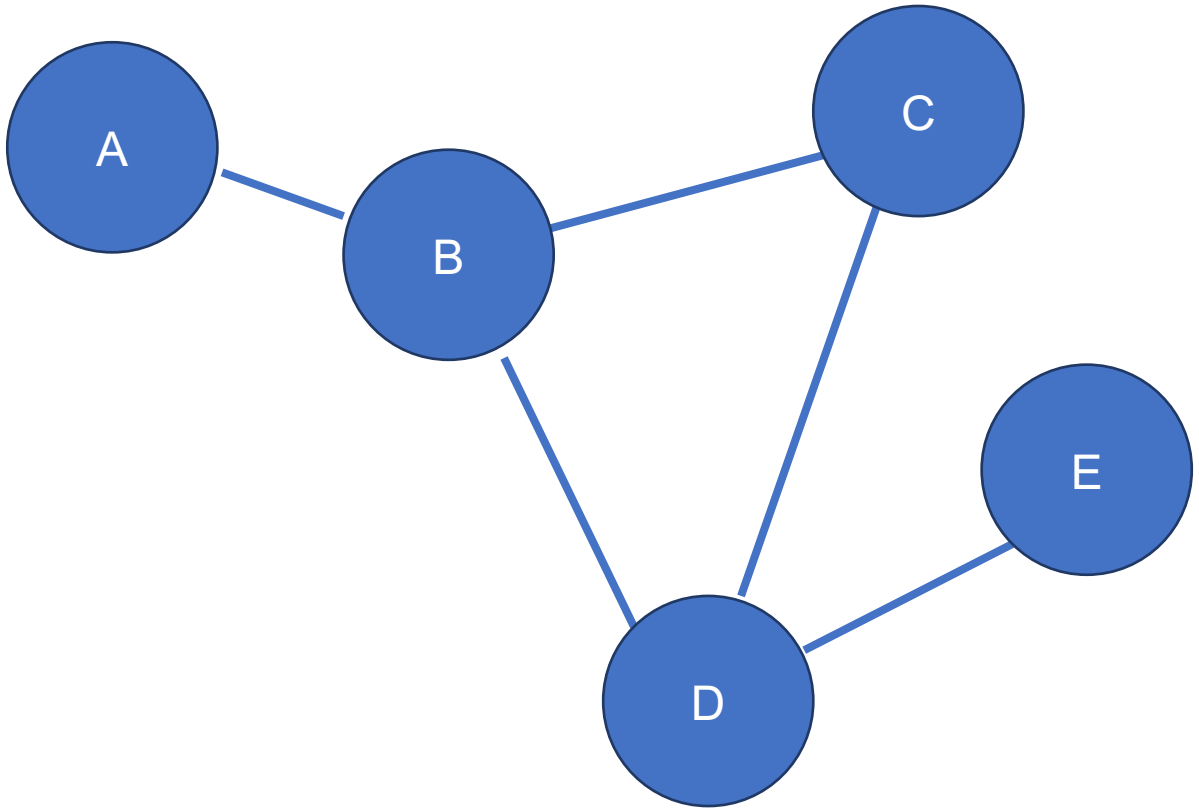
- directed
- not strongly connected but its underlying undirected graph (i.e. without the arrows) is connected
- *Is this graph weakly connected?*
  - *Yes*



Connected and complete graphs

Complete graph

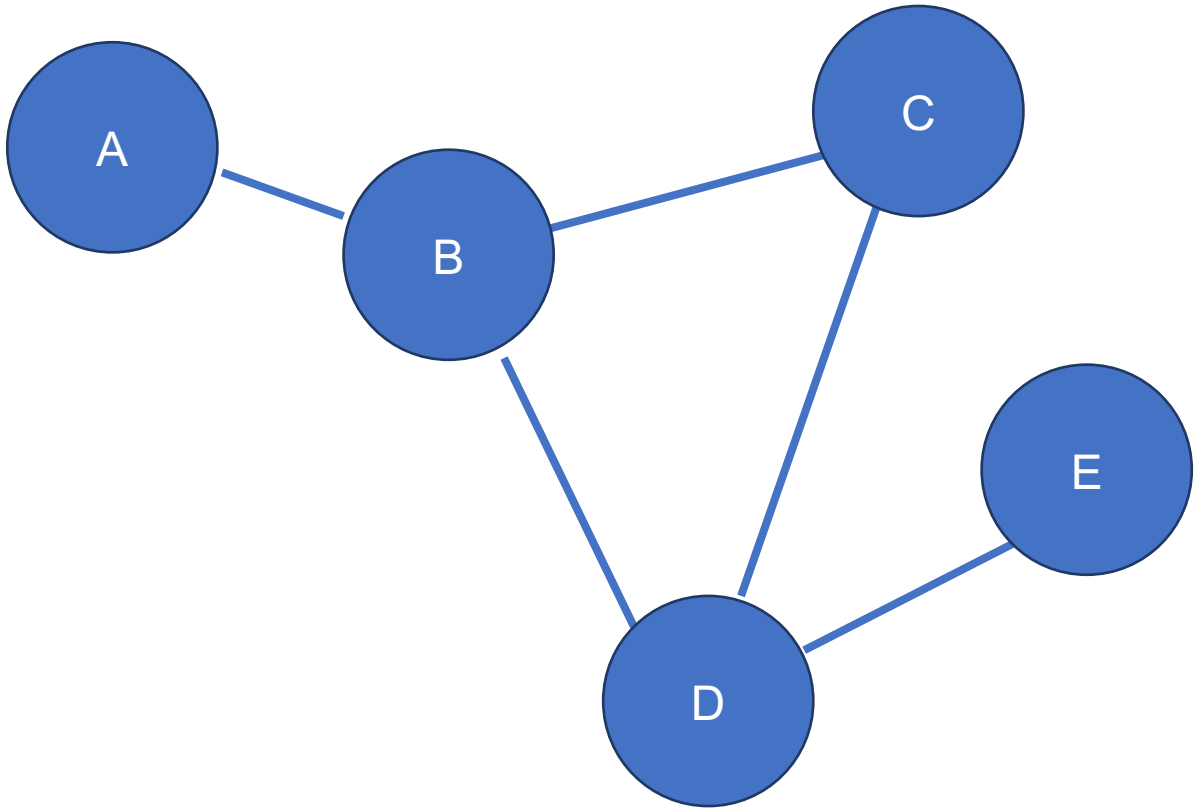
- undirected or directed
- an edge between every pair of vertices
- *Is this graph complete?*



Connected and complete graphs

Complete graph

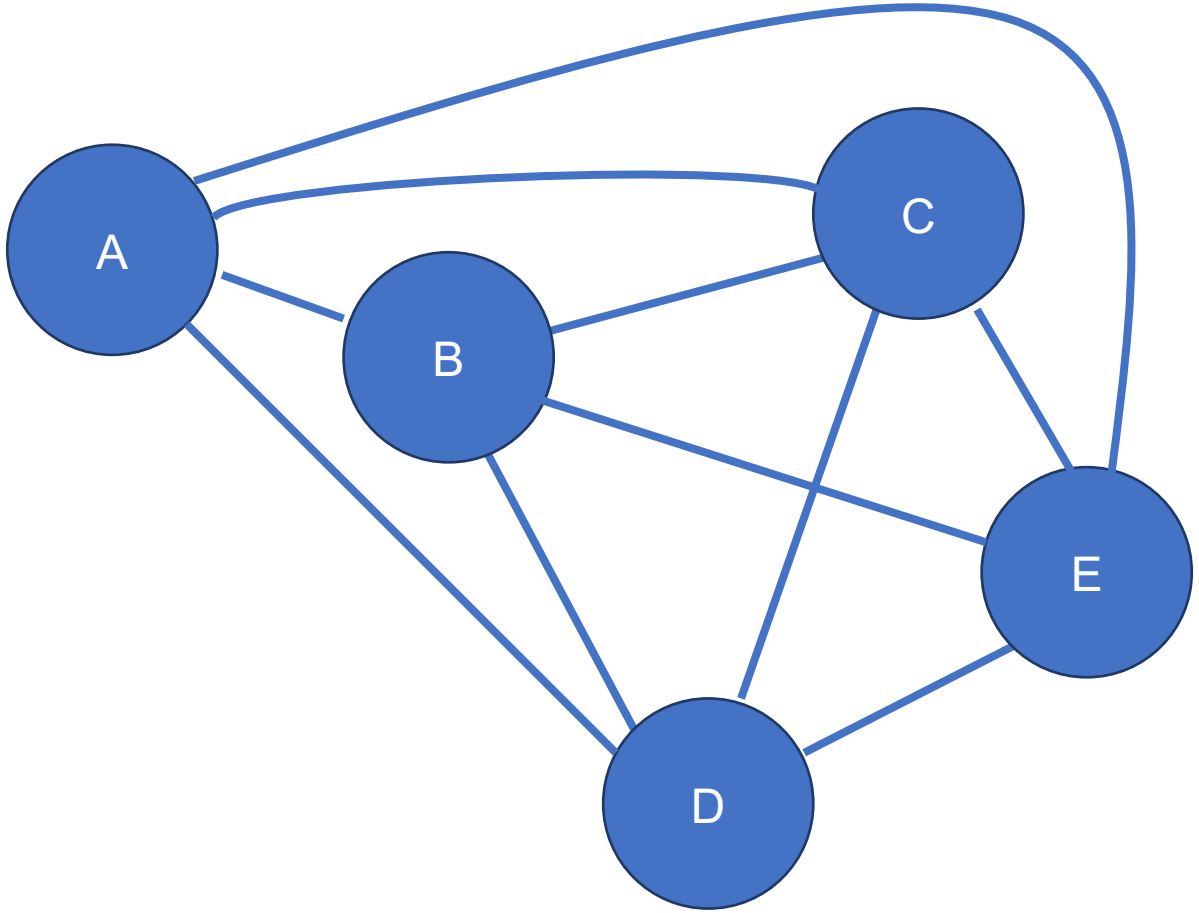
- undirected or directed
- an edge between every pair of vertices
- *Is this graph complete?*
  - *No. No direct connections between multiple pairs*



Connected and complete graphs

Complete graph

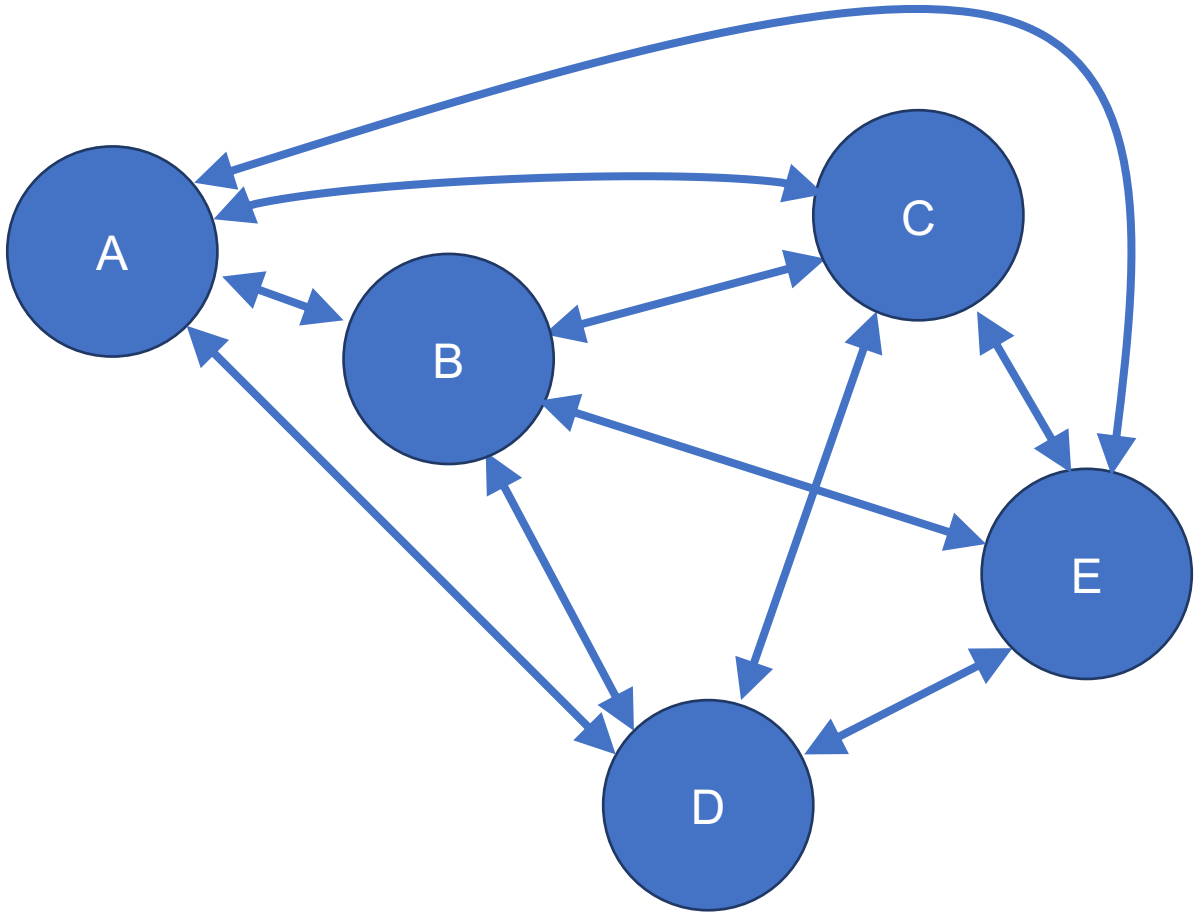
- …now it's complete



Connected and complete graphs

Complete directed graph

- Two-way connections



Walkthrough: Generic Graph ADT implementation

- `addVertex(T vert)`
- `addEdge(T from, T to)`
- `Iterable<T> getPath(T from, T to);`

Design of programs

Well designed (OOD) systems

- Object Oriented Design Principles
- SOLID principles

SOLID principles

- More design principles for object-oriented design of software
- Intended to make your code more:
  - Understandable
  - **Maintainable**
  - **Flexible**
- Widely used, especially in AGILE environments*

SOLID principles

Single responsibility

Open-closed

Liskov substitution

Interface segregation

Dependency inversion

SOLID principles

Single responsibility – related to encapsulation

Open-closed – related to abstraction and inheritance

Liskov substitution – related to polymorphism

Interface segregation – related to encapsulation

Dependency inversion – related to abstraction

Single responsibility principle

“A class should have one, and only one, reason to change.”

In other words:

- a class should only do one thing
- change = changes to the spec

Single responsibility principle

“A class should have one, and only one, reason to change.”

When is this particularly important?

- When multiple clients use the same class
- When a system is likely to change over time

What’s the problem with doing multiple things?

- Changes needed by one client cause changes to class needed by multiple clients → a big mess

Open-closed principle

“Classes should be open for extension but closed for modification.”

In other words:

- use abstraction (specifically, abstract classes)
- when requirements change, extend with new code, ***don’t modify code that works***

Liskov substitution principle

“Derived classes must be substitutable for their base classes.”

In other words:

- subtype polymorphism

Interface segregation principle

“Make fine-grained interfaces that are client specific.”

In other words:

- It is better to have many small interfaces than one big one

Dependency inversion principle

“Depend on abstractions, not on concretions.”

In other words:

- Use abstraction for common functionality
- Parent classes should know nothing of their children

Example: `Comparable<T>`

Interface segregation principle

- Does one very specific thing!
- Can be combined with other interfaces to make a complex object

Dependency inversion principle

(Depend on abstractions not on concretions)

- Doesn’t specify how something is comparable
- Doesn’t care what is comparable
  - Anything can be comparable if you make it so

SOLID principles

Single responsibility: “A class should have one, and only one, reason to change.”

Open-closed: “Classes should be open for extension but closed for modification.”

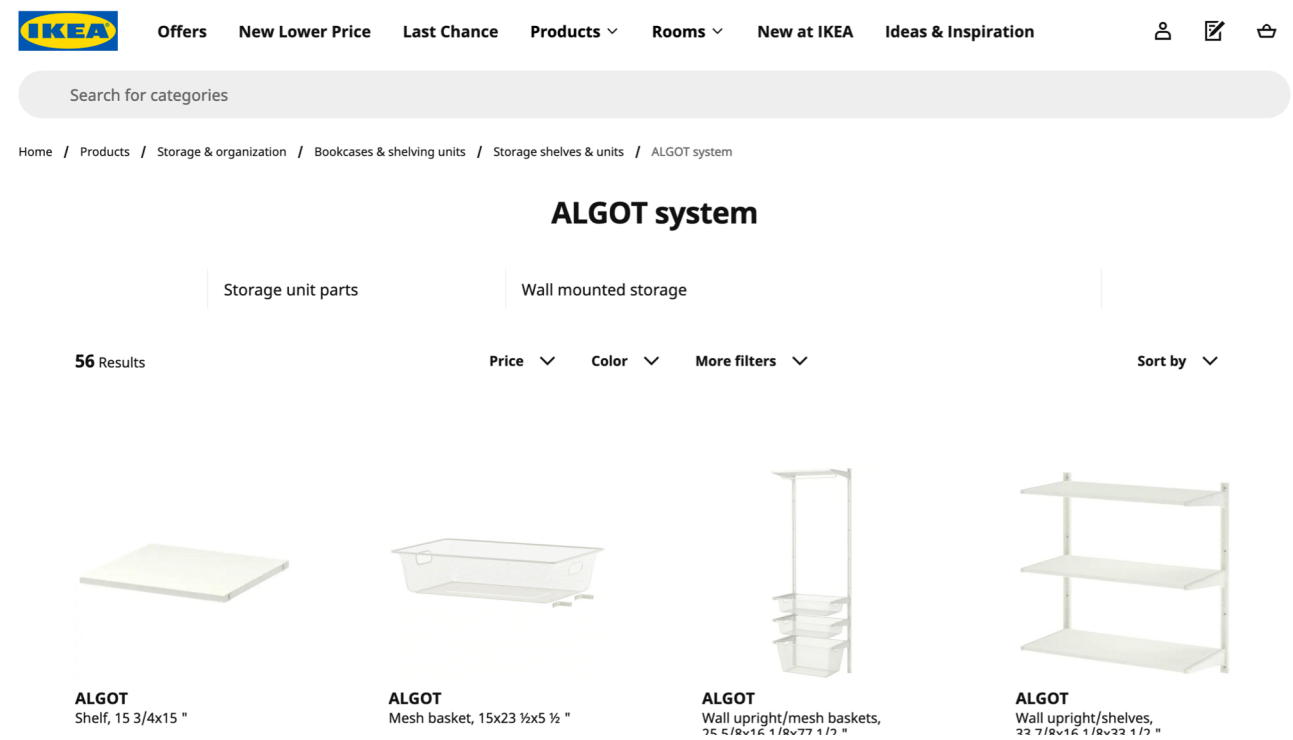
Liskov substitution: “Derived classes must be substitutable for their base classes.”

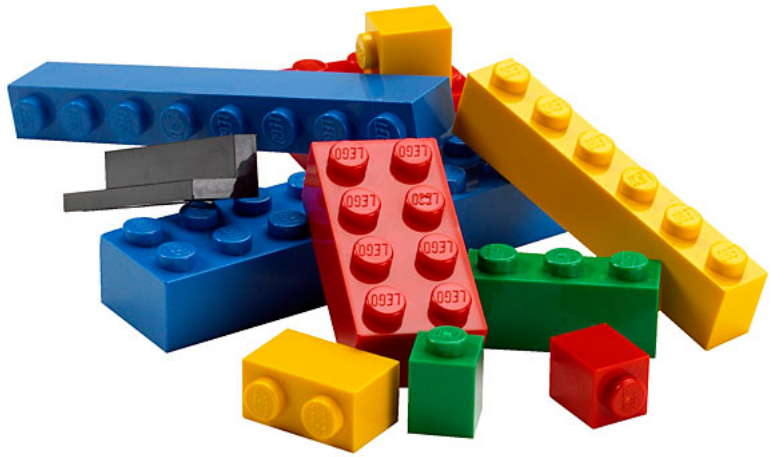
Interface segregation: “Make fine-grained interfaces that are client specific.”

Dependency inversion: “Depend on abstractions, not on concretions.”

Modular software

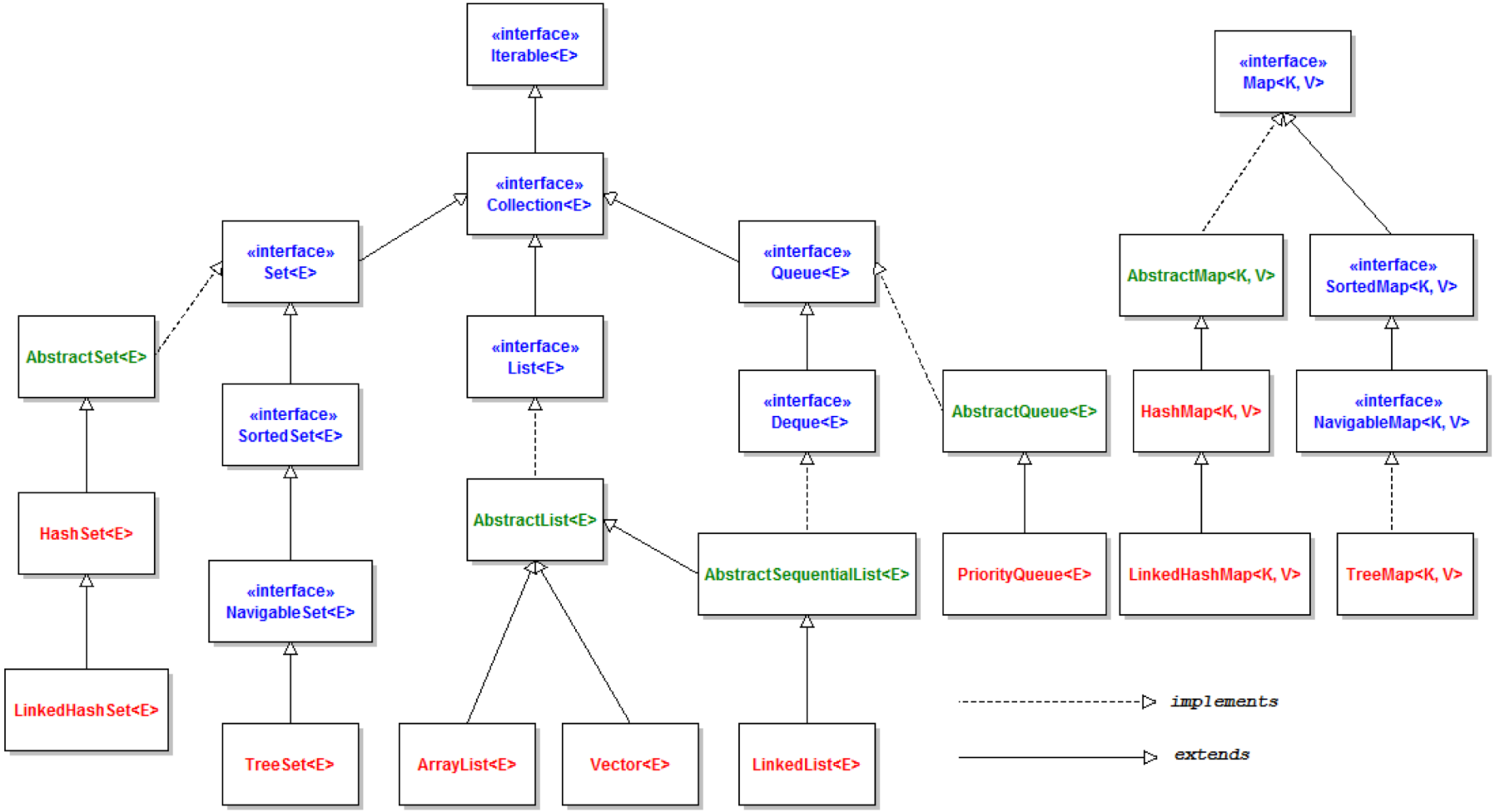
Split a program into interchangeable, reusable components





Example: UI components

Example: Java Collections

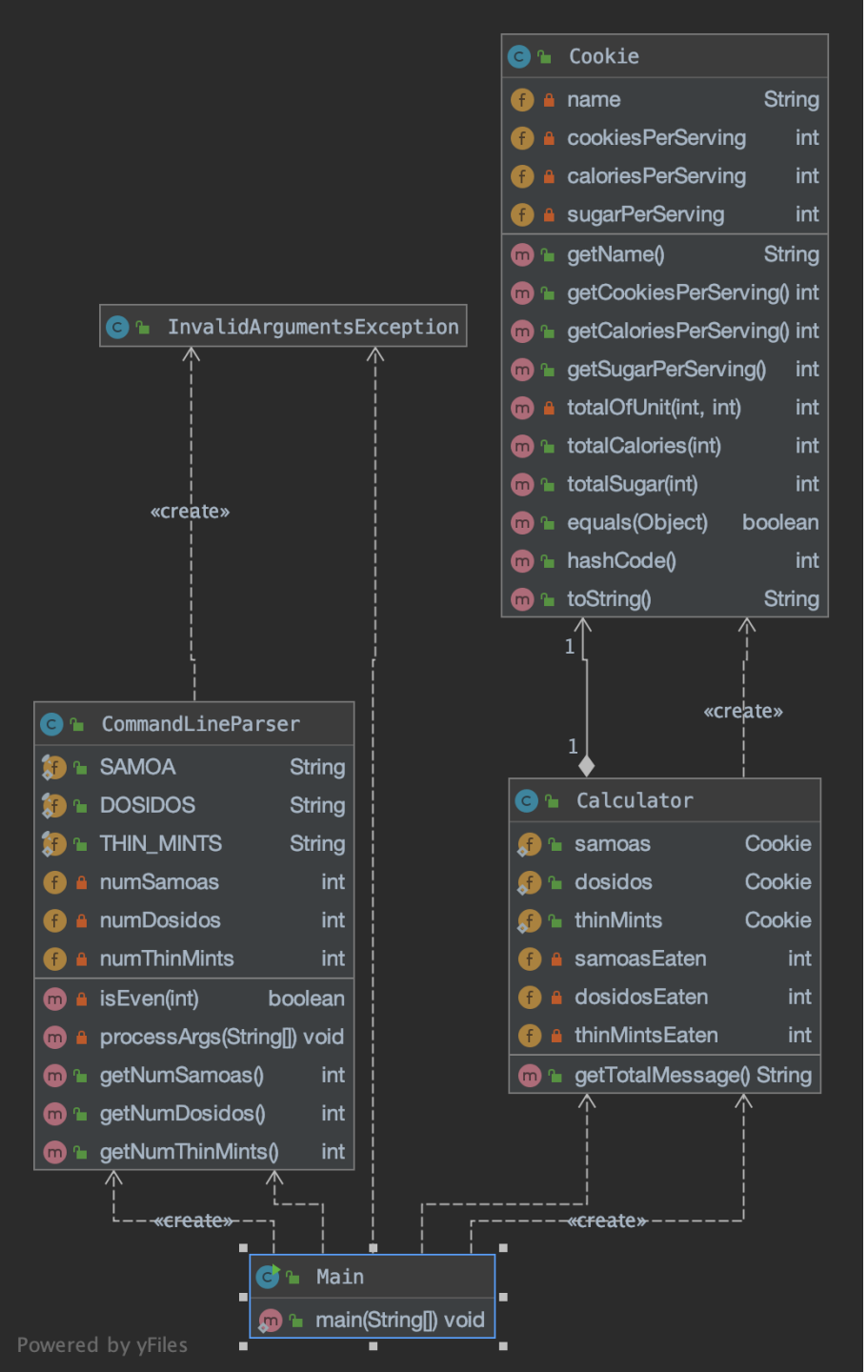


[Picture credit: http://www.codejava.net]

Quick activity: review the Cookie Counter example

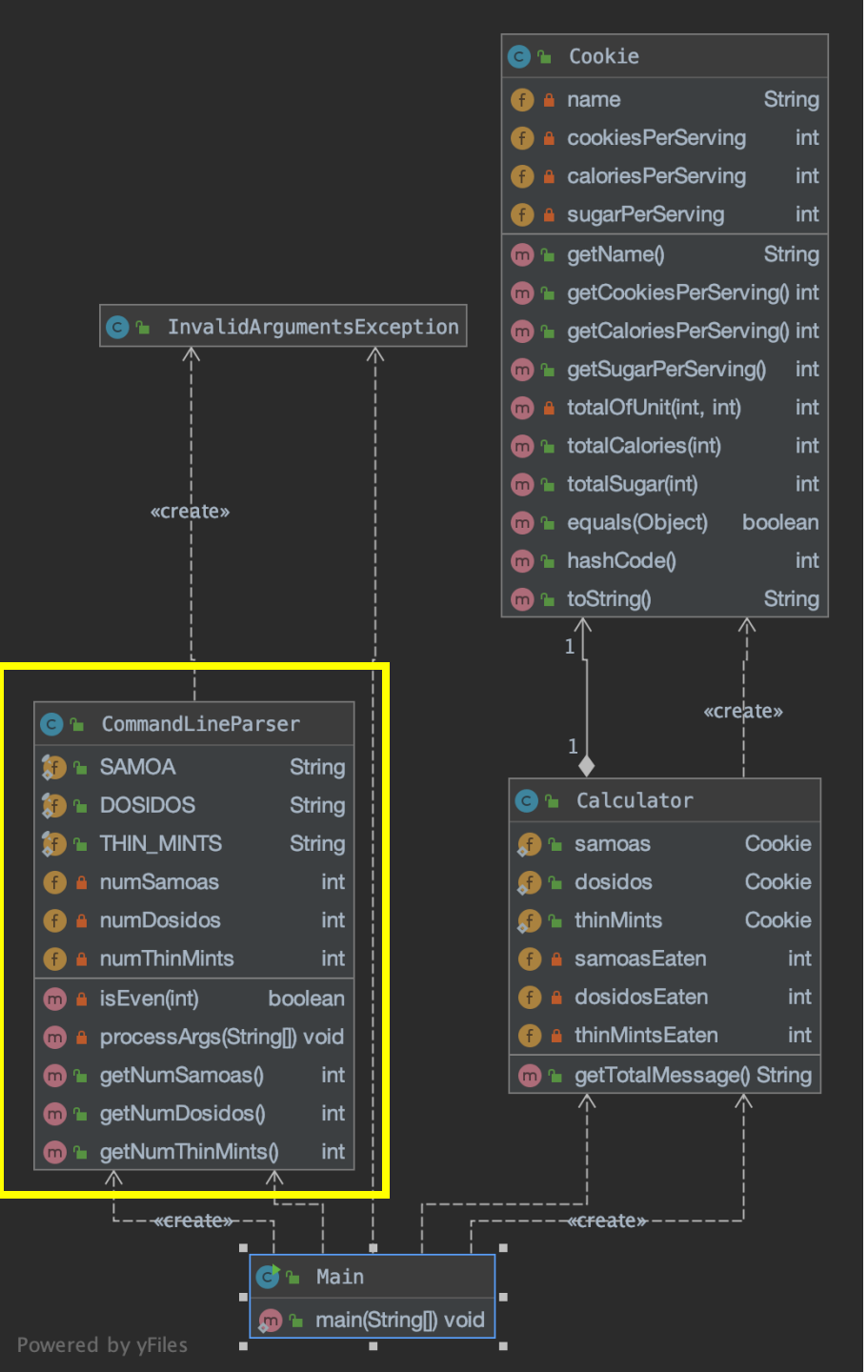
In Lecture9 > cookiecounter

- Is it modular? Spoiler alert: could do better
- Does it allow for easy extension e.g. more types of cookies?
- Is there functionality that would be useful in other situations?
  - E.g broader nutrition calculators?



Is the Cookie Counter modular enough?

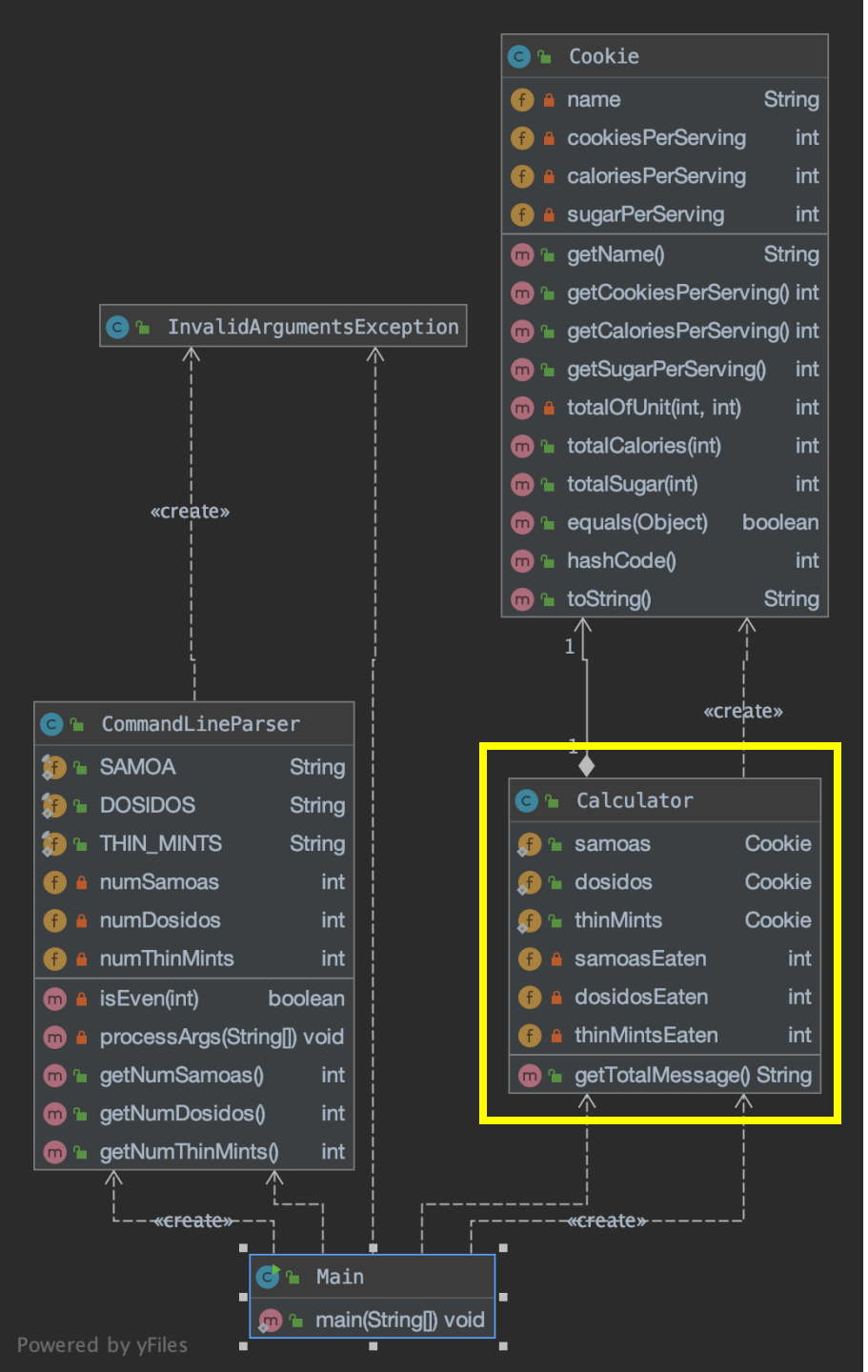
Weaknesses



Is the Cookie Counter modular enough?

Weaknesses

- CommandLineParser only works for this specific situation
  - Doesn't even handle all Girl Scout cookies



Is the Cookie Counter modular enough?

Weaknesses

- CommandLineParser only works for this specific situation
  - Doesn't even handle all Girl Scout cookies
- Calculator only works for this specific situation
  - Doesn't even handle all Girl Scout cookies

Break (10 mins)

Cohesion and coupling

OOD seeks to maximize cohesion and minimize coupling

Cohesion

- How well a module (class) encapsulates a single responsibility → does everything in the class belong in the class?

Coupling

- Interdependencies between classes → does a particular class depend on another class to function?

Cohesion / single responsibility issues

Do parsing / validating command line args and tracking number of cookies really belong together?

```
public class CommandLineParser {
    public static final String SAMOA = "--samoas";
    public static final String DOSIDOS = "--dosidos";
    public static final String THIN_MINTS = "--thin-mints";

    private int numSamoas = 0;
    private int numDosidos = 0;
    private int numThinMints = 0;
```

Probably not.

Cohesion / single responsibility issues

Should a nutrition calculator be responsible for creating new Cookies?

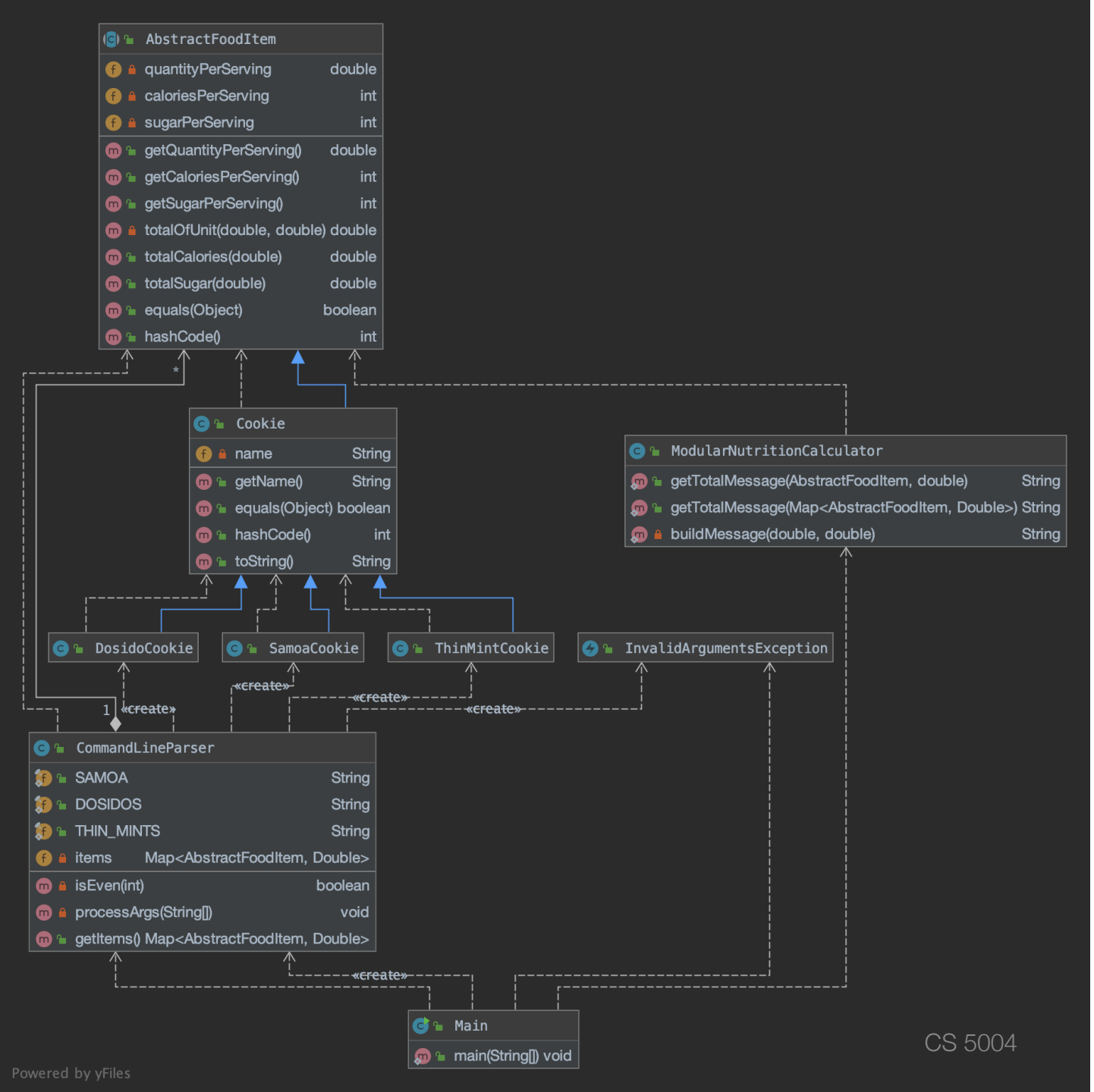
```java
/**
 * Constructor for the Calculator class.
 * @param samoasEaten
 * @param dosidosEaten
 * @param thinMintsEaten
 */
public Calculator(int samoasEaten, int dosidosEaten, int thinMintsEaten) {
    // populate cookies - values should really be constants...
    samoas = new Cookie("Samoas",2, 150, 11);
    dosidos = new Cookie("Do-si-dos", 3, 160, 11);
    thinMints = new Cookie("Thin Mints", 4, 160, 10);
    this.samoasEaten = samoasEaten;
    this.dosidosEaten = dosidosEaten;
    this.thinMintsEaten = thinMintsEaten;
}
```

Probably not.

Related coupling issues

- Command line processor can only function in this specific case
- Calculator depends on 3 specific cookies

***Can we do better?* Yes**



Some improvements

Food data types add more flexibility

Calculator is also more flexible

(CommandLineParser is still a problem)

Decoupling and design

- Examine dependencies before coding (design phase)
- Avoid unnecessary coupling
  - e.g. command line parsing and data storage

Coding without first analyzing dependencies is likely to lead to unnecessary coupling

- A method needs access to information from another object
- Hasty, naïve solutions lead to tight coupling
- Code winds up harder to understand, less reusable

Different types of coupling

Content coupling

- One class is highly dependent on another and may modify its internal state

Common coupling

- Multiple classes share common data

Control coupling

- Using flags / logic to allow client code to control how a method behaves

Stamp coupling

- Another object is passed as a parameter

Data coupling

- Too many parameters

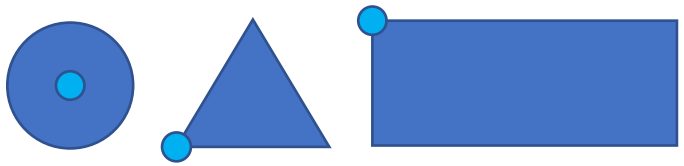
Content coupling example

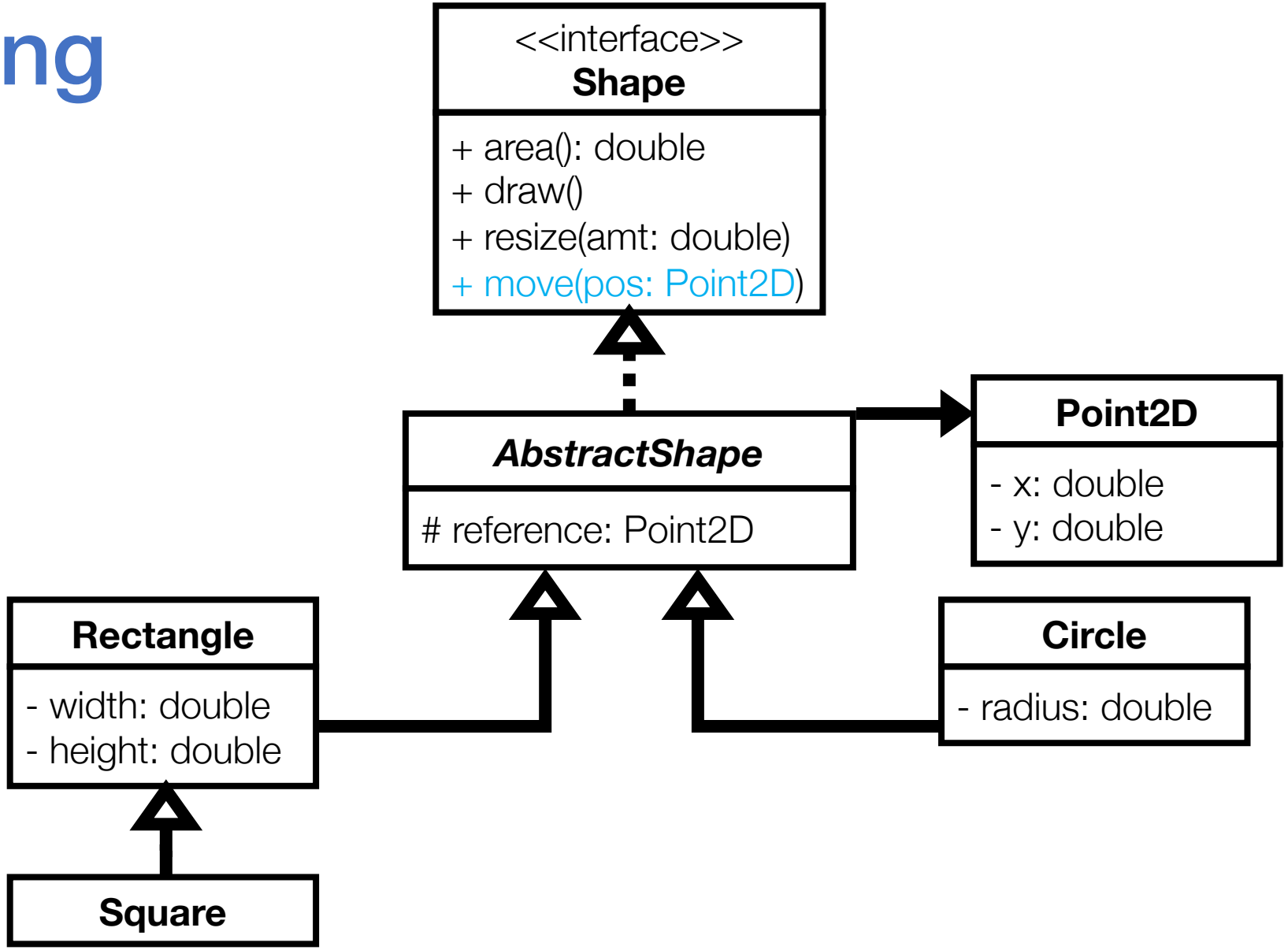
One class is highly dependent on another and may modify its internal state. **Always bad!**

Fix: Don’t modify the internal state of other objects

Content coupling example

`reference` = the point used to start drawing or resizing.





Content coupling example

```
public abstract class AbstractShape implements IShape {
  protected Point2D reference;
  // constructors, other required methods

  public void move(Point2D pos) {
    this.reference.setX(pos.x);
    this.reference.setY(pos.y);
  }
}
```

Content coupling example

```
public abstract class AbstractShape implements IShape {
  protected Point2D reference;
  // constructors, other required methods

  public void move(Point2D pos) {
    this.reference.setX(pos.x);
    this.reference.setY(pos.y);
  }
}
```

Changing the internal state of another object

Content coupling example

```
public abstract class AbstractShape implements IShape {
  protected Point2D reference;
  // constructors, other required methods

  public void move(Point2D pos) {
    this.reference = pos;
  }
}
```

Looser coupling

Content coupling example

```
public abstract class AbstractShape implements IShape {
  protected Point2D reference;
  // constructors, other required methods

  public void move(double x, double y) {
    this.reference = new Point2D(x, y);
  }
}
```

Even looser

Common coupling

Multiple classes share common data (global variables)

- Changes to the common data affects all classes that use it
- Not always a problem

```
fruits = [“apples”, “pears”, “oranges”]
```

Common coupling

Multiple classes share common data (global variables)

- Changes to the common data affects all classes that use it
- Not always a problem

```
fruits = [“apples”, “pears”, “oranges”]

Collections.sort(fruits)
```

Order changes for A & B

Control coupling

Flags / logic used to allow client code to control how a method behaves.

→ Client code needs to know about the method's implementation

```
public String getName(boolean fullName) {
  if (fullName)
     return this.firstName + " " + this.lastName;
  else
     return this.firstName;
}
```

Control coupling

Flags / logic used to allow client code to control how a method behaves.
→ Client code needs to know about the method's implementation

```
public String getFirstName() {
  return this.firstName;
}

public String getFullName() {
 return this.firstName + " " + this.lastName;
}
```

Fix: Separate methods (or polymorphism if applicable)

Stamp Coupling

Another object is passed as a parameter

```
public abstract class AbstractShape implements IShape {
  protected Point2D reference;
  // constructors, other required methods

  public void move(Point2D pos) {
    this.reference = pos;
  }
}
```

AbstractShape is dependent on Point2D

Stamp Coupling

Another object is passed as a parameter

```
public abstract class AbstractShape implements IShape {
  protected Point2D reference;
  // constructors, other required methods

  public void move(Point2D pos) {
    this.reference = pos;
  }
}
```

AbstractShape is dependent on Point2D
→ Changes to Point2D will affect AbstractShape

Stamp Coupling

Another object is passed as a parameter

```
public abstract class AbstractShape implements IShape {
  protected Point2D reference;
  // constructors, other required methods

  public void move(double x, double y) {
    this.reference = new Point2D(x, y);
  }
}
```

Fix: Use primitives where possible

Data coupling

Too many parameters. Rule of thumb: > 3 is too many.

Fix: Encapsulate related parameters into an object

(Yes, this potentially causes stamp coupling)

Benefits of loose coupling

- Design decisions in one class will not affect others
- Correctness is easier to demonstrate through testing
- Increased reusability
- Easier maintenance
  - Less likely that changes will impact other parts of a program
  - More robust to errors.

Takeaways

- Avoid content coupling
- For other forms of coupling:
  - Be aware that they exist
  - Reduce / remove if it makes sense
  - If can't be reduced / removed → take steps to avoid side effects

General good design tips

Methods: ...should do one thing well

- Compute a value, but let the client decide what to do with it
  - e.g. don't calculate a value and put it in a String
- Observe or mutate, but not both
- "Flag" variables to control behavior are often a symptom of poor design
  - *Often* bad, not *always* bad
  - Can you use polymorphism to remove the need for a flag?
  - ...or split the method into separate, smaller, more specific methods?

General good design tips

Methods: ...design signatures carefully

- Avoid long parameter lists → error prone and confusing
- Be careful with overloading
  - Only overload if methods really are related

General good design tips

Instance variables

A variable should be made into an *instance* variable if and only if:

- It is part of the inherent internal state of the object
- It has a value that is meaningful for the object's entire life
- Its state must persist past the end of any one method

All other variables can be local to the methods in which they are only used

- If it's only used in one method, it should be a local variable / constant
- Don't use instance variables to avoid parameter passing

General good design tips

Constructors

- Object should be completely initialized after the constructor is called
  - Client code should not be forced to use other methods to finish initialization
- Constructors can be private / protected
  - Use when there is reason for tighter controls on object initialization

Model-View-Controller architecture

Model-View-Controller (MVC) Architecture

- Used for data-driven user* applications
  - *Users of applications… not other engineers who are users of your code

Model-View-Controller (MVC) Architecture

- Apps have three main tasks:
  - Loading/storing data
  - Creating the user interface (UI)
  - Interpreting user actions—modifying data or modifying the UI

Model-View-Controller (MVC) Architecture

- Apps have three main tasks:
  - Loading/storing data
  - Creating the user interface (UI)
  - Interpreting user actions—modifying data or modifying the UI
- These tasks are largely independent→ separate into 3 components: model, view, and controller
  - Doesn't necessarily mean literally 3 classes: Model, View, Controller
  - May have more than one class for each component

Model

- Talks to data source to retrieve and store data e.g.
  - A database table
  - A file
  - Some external API

View

- Asks model for data, and presents if for the user e.g.
  - Draws an application window with controls for interacting with the data
  - Takes user input and sends it to the controller

Controller

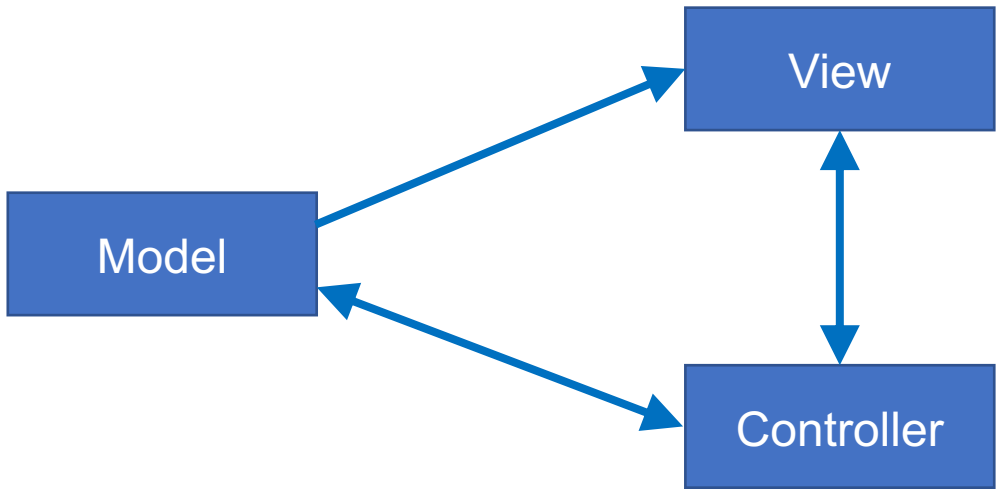
- Listens for the user to change data or state in the UI, notifying the model or view accordingly e.g.
  - User clicks “next” button – tell view to load next screen
  - User submits form input – tell model to update the data

Benefits of the MVC architecture

- **Code organization**
  - Maintainable, easy to find what you need
- **Ease of development**
  - Build and test components independently
- **Flexibility**
  - Swap out views for different presentations of the same data
  - Swap out models to change data storage without affecting user

MVC flow in theory

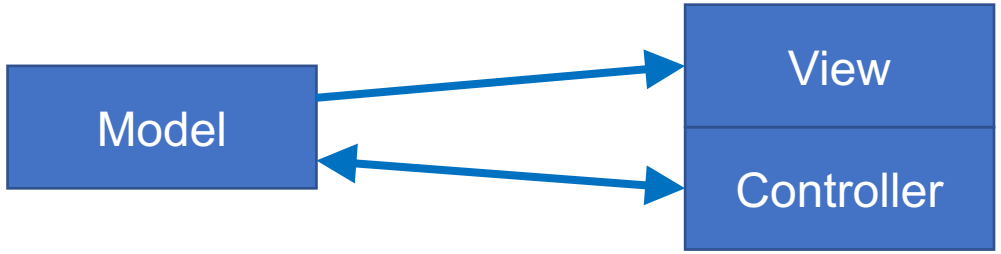
How the application behaves in response to inputs (controller) is independent of the visuals of the UI (view)



MVC flow in practice

View and controller are very intertwined

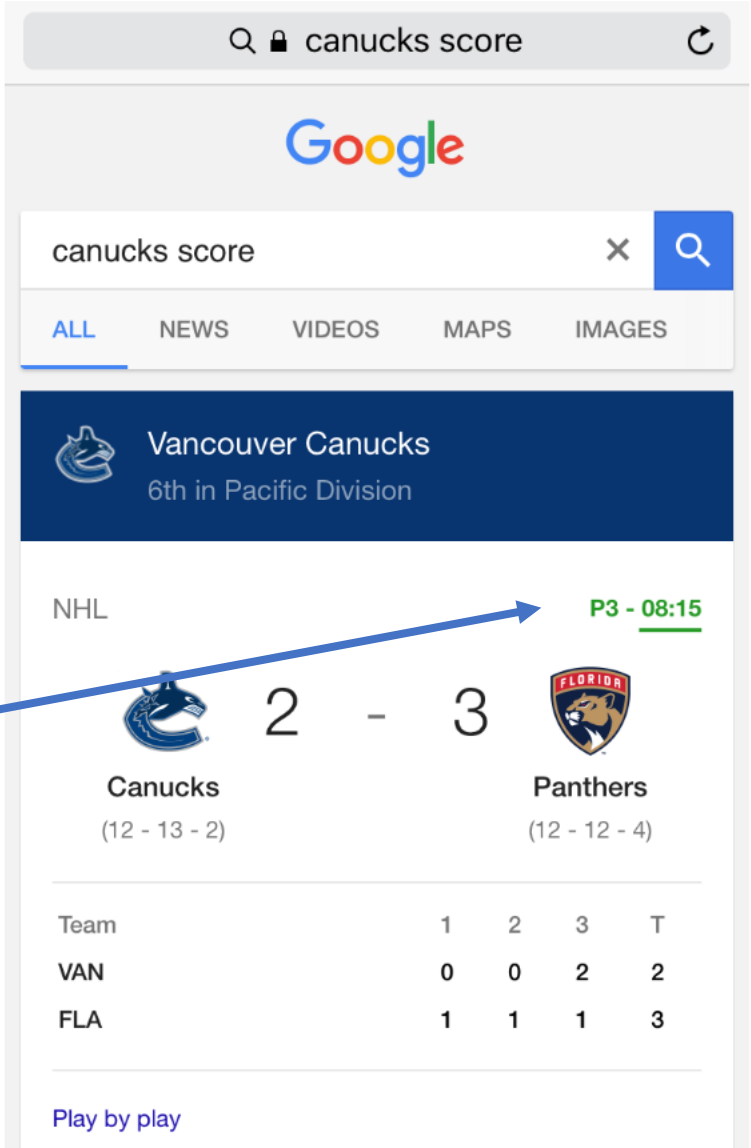
- Many architectures combine the two



Push vs. Pull

Push architecture

- As soon as model changes, views are notified
- Guaranteed to have the latest data in the view
- Example: live updating sports scores



Push vs. Pull

Pull architecture

- When a view needs to be updated, it asks the model for new data
- Avoid unnecessary updates, not nearly as intensive on the view
- Example: wait for user to ask for score updates

MVC example – Traffic signal

MVC example – Traffic signal

Component	Model	View	Controller
Detect cars waiting to enter intersection			X
Traffic lights to direct car traffic		X	
Decide to change the light's status	X		
Manual override for particular lights			X
Detect pedestrians waiting to cross			X
Pedestrian signals to direct pedestrians		X	
External timer which triggers changes at set interval			X

MVC example – Traffic signal

Model

- Stores current state of traffic flow
- Stores whether there are cars/pedestrians waiting

View

- Conveys information to cars and pedestrians (the lights)

Controller

- Aware of model's current direction
- Triggers methods to notify model that state should change

MVC example – Traffic signal

Model

- `TrafficModel` - keeps track of which lights should be on and off

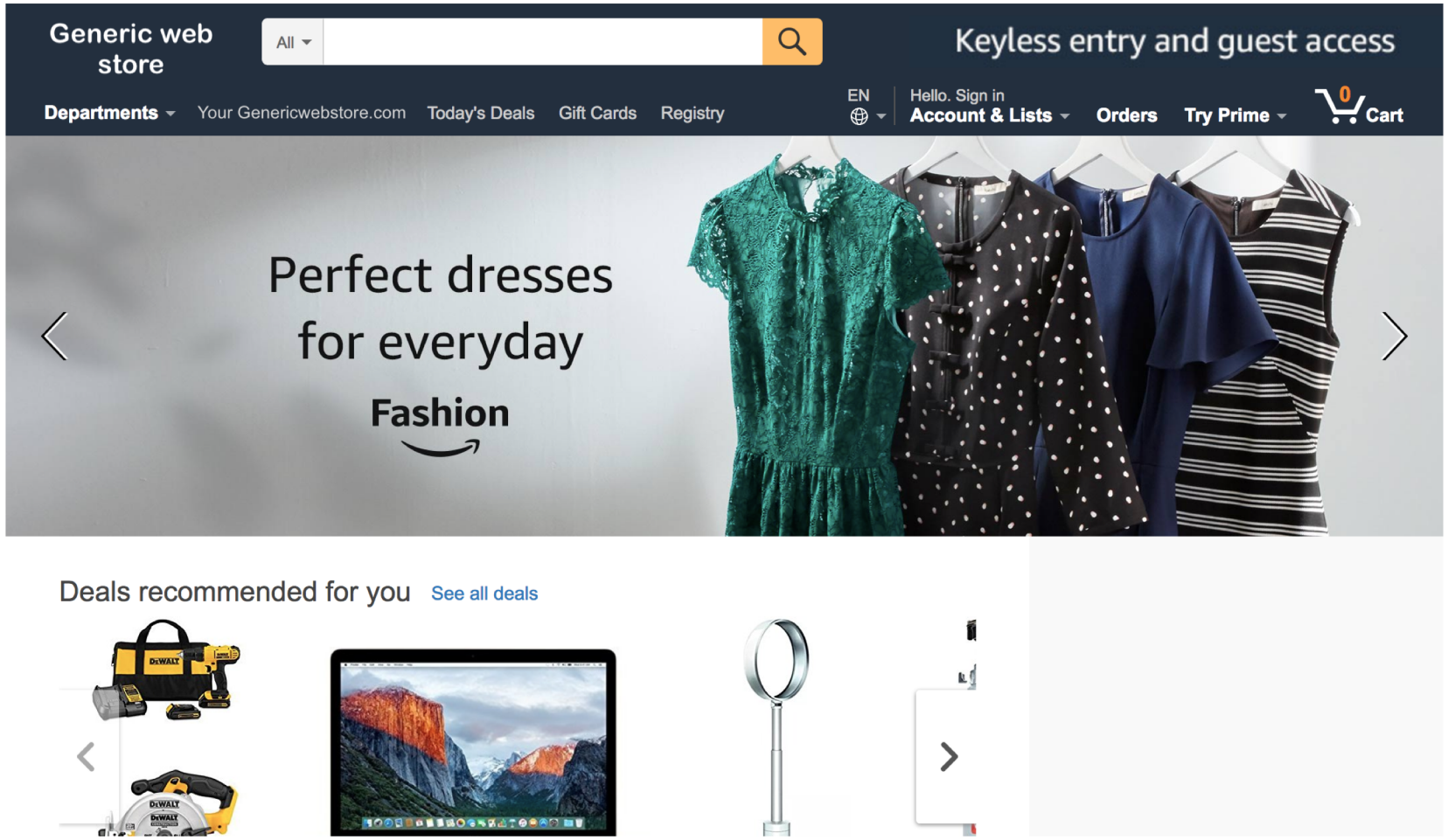
View

- `CarLight` – shows relevant state of `TrafficModel` to cars
- `WalkLight` – shows relevant state of `TrafficModel` to pedestrians

Controller

- `WalkButton` – notifies `TrafficModel` that pedestrian is waiting
- `CarDetector` – notifies `TrafficModel` that car is waiting
- `LightSwitch` – enables/disables `CarLight, WalkLight`

MVC example – online store



MVC example – online store

Component	Model	View	Controller
Store product/inventory details			
Display price/details of a product			
User sign-in			
Authenticate user sign-in attempt			
Check user credentials			
Update user's shopping cart			
Purchase items in shopping cart			
Maintain record of customer transactions			

MVC example – online store

Component	Model	View	Controller
Store product/inventory details?			
Display price/details of a product			
User sign-in			
Authenticate user sign-in attempt			
Check user credentials			
Update user's shopping cart			
Purchase items in shopping cart			
Maintain record of customer transactions			

MVC example – online store

Component	Model	View	Controller
Store product/inventory details	X		
Display price/details of a product			
User sign-in			
Authenticate user sign-in attempt			
Check user credentials			
Update user's shopping cart			
Purchase items in shopping cart			
Maintain record of customer transactions			

MVC example – online store

Component	Model	View	Controller
Store product/inventory details	X		
Display price/details of a product?			
User sign-in			
Authenticate user sign-in attempt			
Check user credentials			
Update user's shopping cart			
Purchase items in shopping cart			
Maintain record of customer transactions			

MVC example – online store

Component	Model	View	Controller
Store product/inventory details	X		
Display price/details of a product		X	
User sign-in			
Authenticate user sign-in attempt			
Check user credentials			
Update user's shopping cart			
Purchase items in shopping cart			
Maintain record of customer transactions			

MVC example – online store

Component	Model	View	Controller
Store product/inventory details	X		
Display price/details of a product		X	
User sign-in?			
Authenticate user sign-in attempt?			
Check user credentials?			
Update user's shopping cart			
Purchase items in shopping cart			
Maintain record of customer transactions			

MVC example – online store

Component	Model	View	Controller
Store product/inventory details	X		
Display price/details of a product		X	
User sign-in		X	
Authenticate user sign-in attempt			X
Check user credentials	X		
Update user's shopping cart			
Purchase items in shopping cart			
Maintain record of customer transactions			

MVC example – online store

Component	Model	View	Controller
Store product/inventory details	X		
Display price/details of a product		X	
User sign-in		X	
Authenticate user sign-in attempt			X
Check user credentials	X		
Update user's shopping cart?			
Purchase items in shopping cart?			
Maintain record of customer transactions?			

MVC example – online store

Component	Model	View	Controller
Store product/inventory details	X		
Display price/details of a product		X	
User sign-in		X	
Authenticate user sign-in attempt			X
Check user credentials	X		
Update user's shopping cart			X
Purchase items in shopping cart			X
Maintain record of customer transactions	X		

Exercise: Tic-tac-toe game

MV* implementation of Tic-Tac-Toe

- Two players (“X”, “O”)
- Interactive (use Scanner)
- First player to get 3-in-a-row wins
  - Horizontal or
  - Vertical or
  - Diagonal

```
X's turn! Enter your move in the format: row col
0 2
    0   1   2
  -------------
0 |   | O | X |
1 |   | X |   |
2 |   | X | O |
  -------------
```

- Code to draw the board is provided
- More info in Canvas