CS 5004: Lecture 10

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

- Codewalk logistics
- Lecture 9 final exercise recap
- Interfaces Comparable & Comparator
- Interfaces Iterable & Iterator
- Nested classes
- More data structures: graph & tree implementations

Codewalk logistics

There will be two types of codewalk:

- **In progress** – after 1 week with the assignment
- **Complete** – after 2 weeks with the assignment

Codewalk logistics

There will be two types of codewalk:

- **In progress** – after 1 week with the assignment
- **Complete** – after 2 weeks with the assignment

All groups will do **one in progress codewalk** and **one complete codewalk**

- Codewalks must be on different assignments

Expectations: All codewalks

- All group members participate
  - Please let us know ASAP if this will not be possible due to timezones
- You walk the audience through your code / design
  - Start with the UML
  - You can assume everyone knows what the assignment is about
- Strict time limit of 15 minutes
  - Please time yourselves!

Expectations: In progress codewalk

You should have:

- A complete design plan e.g. a UML showing the classes you'll create, their fields, and methods
- A partial implementation e.g. interfaces, one or two classes, but perhaps incomplete Javadoc or tests

You should be able to:

- Explain the reasoning behind your design
- Explain problems or confusions you're currently working on

Expectations: Complete codewalk

You should have:

- A complete, final UML showing your design
- A complete, polished implementation

You should be able to:

- Explain and justify all design choices
- Run your program to demo how it works (if there's a UI)

Codewalk dates

Assignment 8

- In progress – 3/31
- Complete – 4/7

Assignment 9

- In progress – 4/14
- Complete – 4/21

Codewalk dates

Assignment 8

- In progress – 3/31
- Complete – 4/7

If you sign up for this for your first codewalk...

Assignment 9

- In progress – 4/14
- Complete – 4/21

...your second codewalk will be...

Codewalk dates

Assignment 8

- In progress – 3/31
- Complete – 4/7

If you sign up for this for your first codewalk...

Assignment 9

- In progress – 4/14
- Complete – 4/21

...your second codewalk will be...

How to sign up (slots for HW8 will be opened tomorrow)

Only one group member needs to sign up

- Go to Canvas > Calendar
- Click the Find Appointment button
- Select this course
- View and reserve an available slot

Sign-ups are first come, first served. If all slots for your preferred session are faded, the session is full and you will have to choose a different session.

Lecture 9 recap

Last exercise

Interfaces `Comparable` & `Comparator`

Comparing objects

- We already know how to determine if two objects are **equal**

```
myObject.equals(otherObject);
```

- Override Java's built-in `equals` method to enable checking if two objects are the same based on their properties

Comparing objects

- What if we want to know if `myObject` is less than/greater than `otherObject`?
  - Why might we want to know this?

Comparing objects

- What if we want to know if `myObject` is less than/greater than `otherObject`?
  - Why might we want to know this?
  - Sorting collections of Objects… or other purposes:
    - e.g. two playing cards – which is worth more/less in a game

Comparing objects with `compareTo`

Many of Java's built-in objects (e.g. String, Integer) can be compared using **`compareTo`**

```
String abc = "ABC";
String lower = "abc";
abc.compareTo(lower);
```

→ **output?**

Comparing objects with `compareTo`

- Returns **negative int** if `this` is less than `other` e.g. `abc < lower`

```
String abc = "ABC";
String lower = "abc";
abc.compareTo(lower);
```

→ `output?`

Comparing objects with `compareTo`

- Returns **negative int** if `this` is less than `other` e.g. `abc < lower`
- Returns **zero** if `this` equals `other` e.g. `abc.equals(lower)`

```
String abc = "ABC";
String lower = "abc";
abc.compareTo(lower);
```

→ **output?**

Comparing objects with `compareTo`

- Returns **negative int** if `this` is less than `other` e.g. `abc < lower`
- Returns **zero** if `this` equals `other` e.g. `abc.equals(lower)`
- Returns **postive int** if `this` is greater than `other` e.g. `abc > lower`

```
String abc = "ABC";
String lower = "abc";
abc.compareTo(lower);
```

→ **output?**

Comparing objects with `compareTo`

- Returns **negative int** if `this` is less than `other` e.g. `abc < lower`
- Returns **zero** if `this` equals `other` e.g. `abc.equals(lower)`
- Returns **postive int** if `this` is greater than `other` e.g. `abc > lower`

```
String abc = "ABC";
String lower = "abc";
abc.compareTo(lower);
```

→ `-32: abc < lower`

Comparing objects with `compareTo`

Can be used as a test in if statements e.g.

```
if (abc.compareTo(lower) < 0) {
  // Do this if abc comes before lower
}
```

Implementing `compareTo` in your classes

Unlike **`equals`** & **`hashCode`**, there is no **`compareTo`** to override by default

Implementing `compareTo` in your classes

Unlike **`equals`** & **`hashCode`**, there is no **`compareTo`** to override by default

- Implement **`Comparable<T>`**
- Replace **`T`** with the class name

```
public class MyClass implements
                Comparable<MyClass> {

   …

}
```

Implementing `compareTo` in your classes

Implement `Comparable<T>`'s required method, `compareTo`

- If your header is correct (you changed T to the class name), IntelliJ will generate the method skeleton as shown
  - Otherwise `o` will be an `Object`

```
public class MyClass implements
                Comparable<MyClass> {
  public int compareTo(MyClass o) {
    …
  }
}
```

Implementing `compareTo` in your classes

Return:

- Negative (usually -1) if this is less than o
- Zero if this equals o
- Positive (usually 1) if this is greater than o

```
public class MyClass implements
                Comparable<MyClass> {
  public int compareTo(MyClass o) {
    …
  }
}
```

Rules for implementing `compareTo`

- If A is less than B, then B must be greater than A
- If A is less than B and B is less than C, then A must be less than C
- If A is not less than B and B is not less than A, then A and B must be equal
  - If `A.equals(B)` is `true`, `A.compareTo(B)` should return 0.

Exercise: making `Book` `Comparable`

- Take a look at the Book class in lecture-code:
  - Lecture 10 code > datatypes package
- How would you implement `compareTo`?:
  - When should a Book be **less than** another Book
  - When should a Book be **equal to** another Book
  - When should a Book be **greater than** another Book
- Try implementing `compareTo`

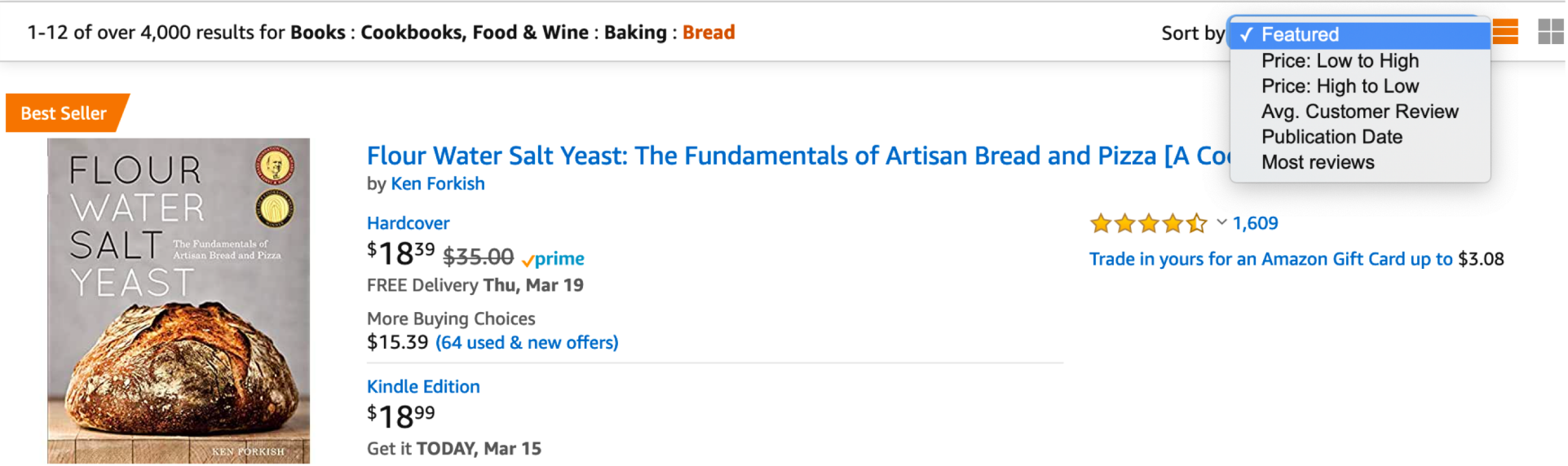
Making `Book` `Comparable`

One solution:

- Check **`equals`** first
- Then, use **`compareTo`** on the titles
- If titles are equal, use publication year as tie breaker

Interface `Comparator`

Makes it possible to sort custom objects in more than one way



Using `Comparator`

A <u>separate</u> class that implements `Comparator<T>`

→ Replace T with the type to compare

```
public class AuthorCompare implements Comparator<Book> {




}
```

Using `Comparator`

A <u>separate</u> class that implements `Comparator<T>`

→ Replace T with the type to compare

```
public class AuthorCompare implements Comparator<Book> {

  @Override
  public int compare(Book o1, Book o2) {
    // How to compare
  }
}
```

Required method

Return:

- < 0 if o1 < o2
- 0 if equal
- > 0 if o1 > o2

Sorting collections of Objects

Using `Collections.sort()`

Objects in collection <u>must</u> implement `Comparable<T>`

```
List<Book> books = new ArrayList<Book>();
// Add some books...
Collections.sort(books); // Won't compile if Book isn't
Comparable
```

...and/or pass a *Comparator* explaining how to sort

```
Collections.sort(books, new AuthorCompare());
```

Examples

- See lecture-code > Lecture10 > comparing
- Two Comparators
- Run main to see the effect of sorting with Comparators

Interfaces `Iterable` & `Iterator`

Example data structure: IListAdt

Generic custom list ADT (lecture 10 sample code)

- `void append(T item);`
- `T get(int index);`
- `void insert(T item, int index);`

Underlying data structure = generic linked list (lecture 8)

Iterating through a collection

Most of Java's built-in collections support iteration using a for-each loop

- How do we iterate our `IListADT`?

```
for (DataType elem: myCollection) {
  // do something with elem
}
```

Iterating through a collection

```
public static void main(String[] args) {
    Book lotr = new Book( title: "The Lord of the Rings",  author: "Tolkien, J. R. R.",  year: 1954);
    Book alchemist = new Book( title: "The Alchemist",  author: "Coelho, Paul",  year: 1988);
    Book littlePrince = new Book( title: "The Little Prince",  author: "Saint-Exupéry, Antoine de",  year: 1943);
    Book harryPotter = new Book( title: "Harry Potter and the Philosopher's Stone",  author: "Rowling, J. K.",  year: 1997);
    Book divingBell = new Book( title: "The Diving-Bell and the Butterfly",  author: "Bauby, Jean-Dominique",  year: 1997);

    IListAdt<Book> books = ListAdt.createEmpty();
    books.append(lotr);
    books.append(alchemist);
    books.append(littlePrince);
    books.append(harryPotter);
    books.append(divingBell);

    for (Book b: books) {
        System.out.println(b.toString());
    }
}
```

Iterating through a `IListADT`

- `IListADT` needs to extend Java's `Iterable<T>`
  - Enables iteration with a for-each loop
- Also need a separate implementation of Java's interface `Iterator<T>`

Interface `Iterable<T>`

- Implementing `Iterable<T>` allows an object to be traversed using a for-each loop
- Implementing classes must provide a method `Iterator<T> iterator()`

Modifier and Type	Method and Description
default void	`forEach(Consumer<? super T> action)` Performs the given action for each element of the `Iterable` until all elements have been processed or the action throws an exception.
`Iterator<T>`	`iterator()` Returns an iterator over elements of type T.
default `Spliterator<T>`	`spliterator()` Creates a `Spliterator` over the elements described by this `Iterable`.

Making `IListADT` iterable

Which class(es) should implement/extend `Iterable<T>`?

```
// Interface?
public interface IListAdt<T> {…}

// Implementing class?
public class ListAdt<T> implements IListAdt<T> {…}
```

Making `IListADT` iterable

Which class(es) should implement/extend `Iterable<T>`?

```
// Interface?
public interface IListAdt<T> {…}

// Implementing class?
public class ListAdt<T> implements IListAdt<T> {…}
```

Example: making `IListADT` Iterable

- Change `IListAdt` to extend `Iterable<T>`
- Create a placeholder for `Iterable`'s required method in the implementing class
  - `Iterator<T> iterator();`

Interface `Iterator<T>`

The class that implements `Iterator<T>` does the actual iterating from one element to the next

- for-each loop calls `Iterator`'s methods automatically → we just have to implement them

Iterator required methods:

```
// Are there more elements?
boolean hasNext();

// Return the next element

T next();
```

Implementing interface `Iterator<T>`

Multiple options:

- **In a separate class**
- In an inner class
  - Later today

Iterator required methods:

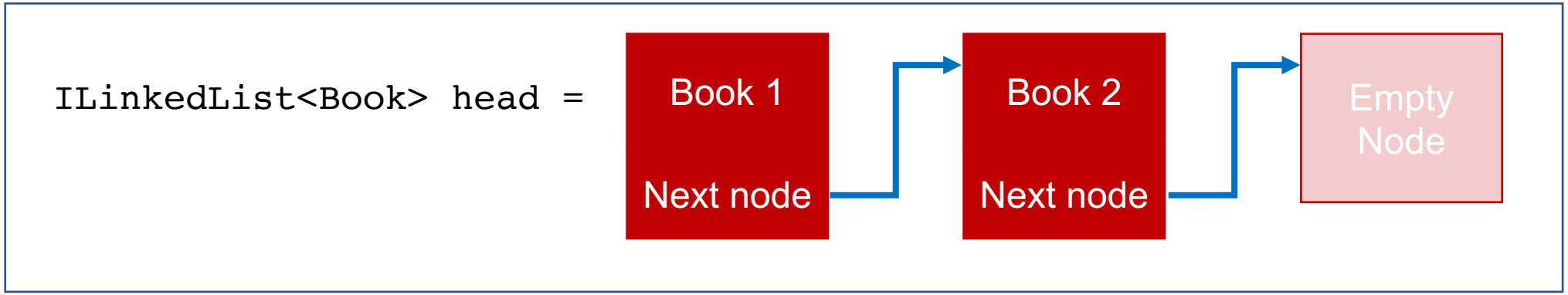
```
// Are there more elements?
boolean hasNext();

// Return the next element

T next();
```

How does `Iterator` work?

ListAdt returns an **`Iterator`** object because it implements Iterable

- Iterator shown in blue

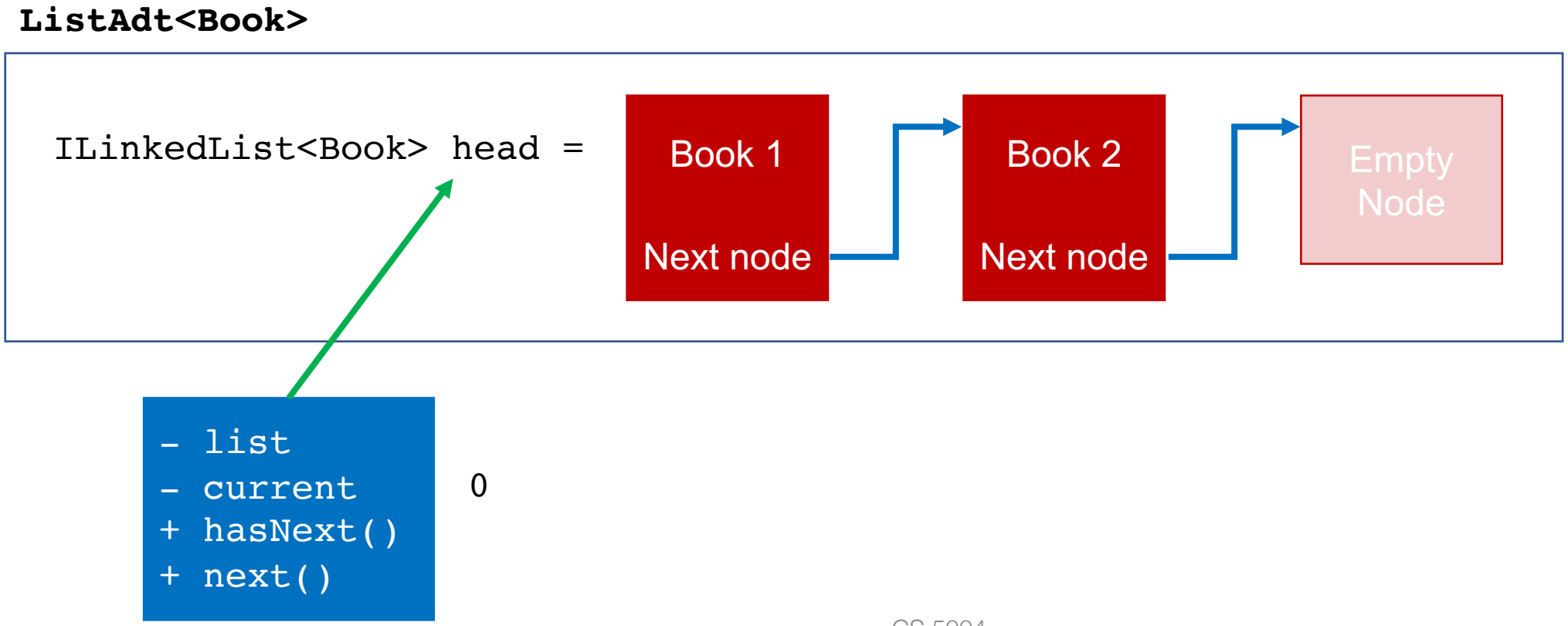
`ListAdt<Book>`




```
- list
- current
+ hasNext()
+ next()
```

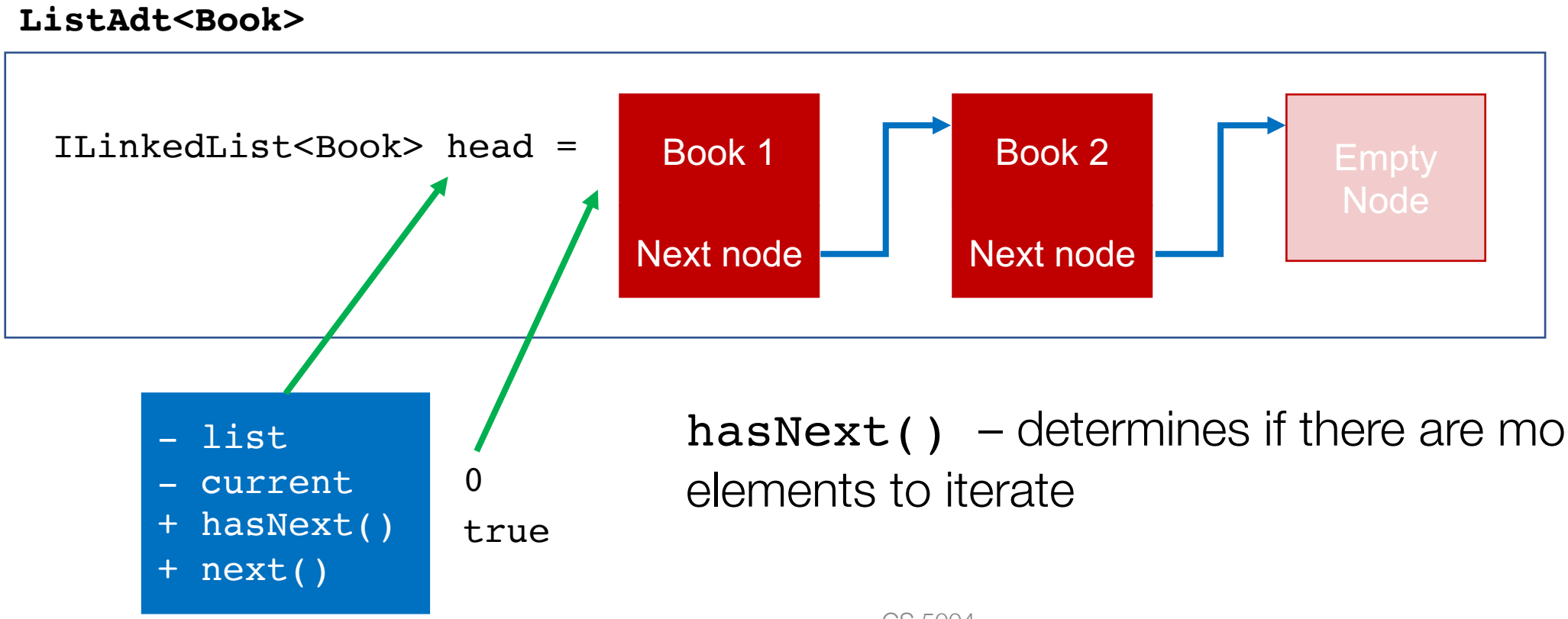
How does `Iterator` work?

`Iterator` contains a field referencing the list (`ListAdt<Book>`) and a field referencing the current index



How does `Iterator` work?

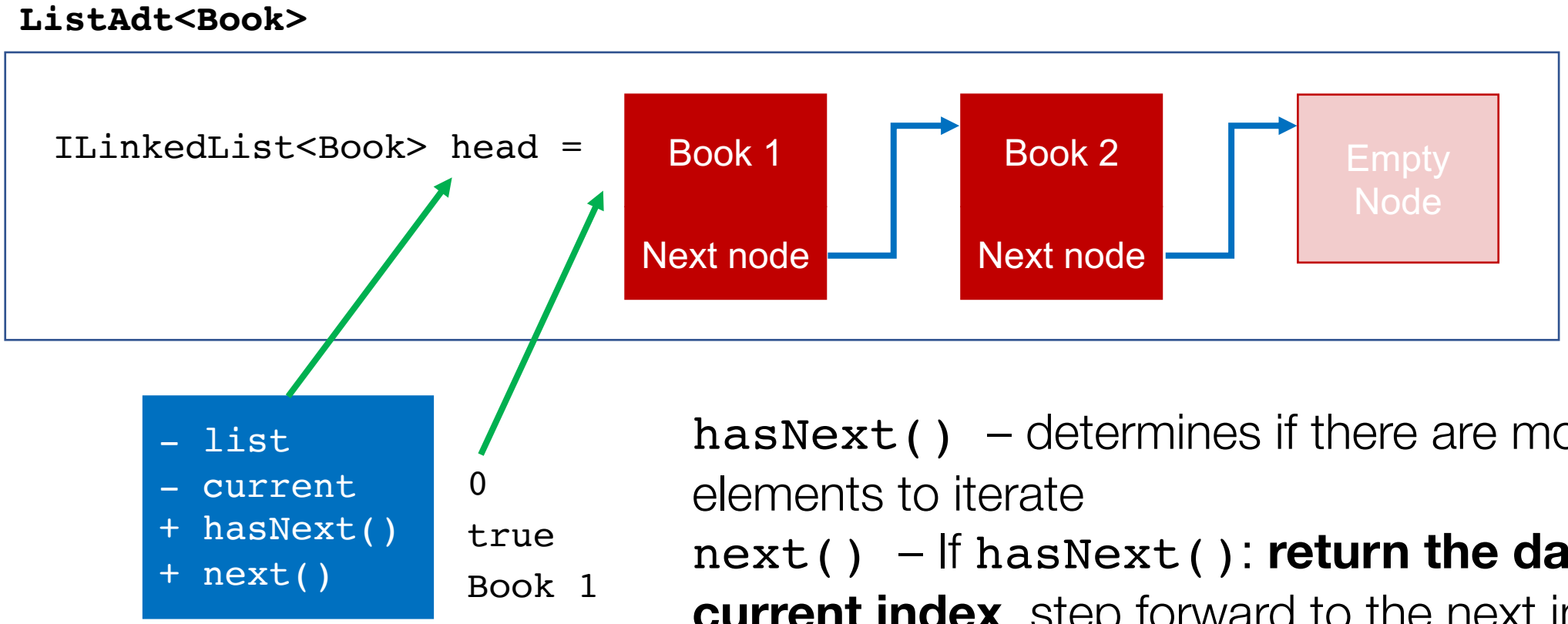
`Iterator` contains a field referencing the list (`ListAdt<Book>`) and a field referencing the current index



`hasNext()` – determines if there are more elements to iterate

How does `Iterator` work?

`Iterator` contains a field referencing the list (`ListAdt<Book>`) and a field referencing the current index

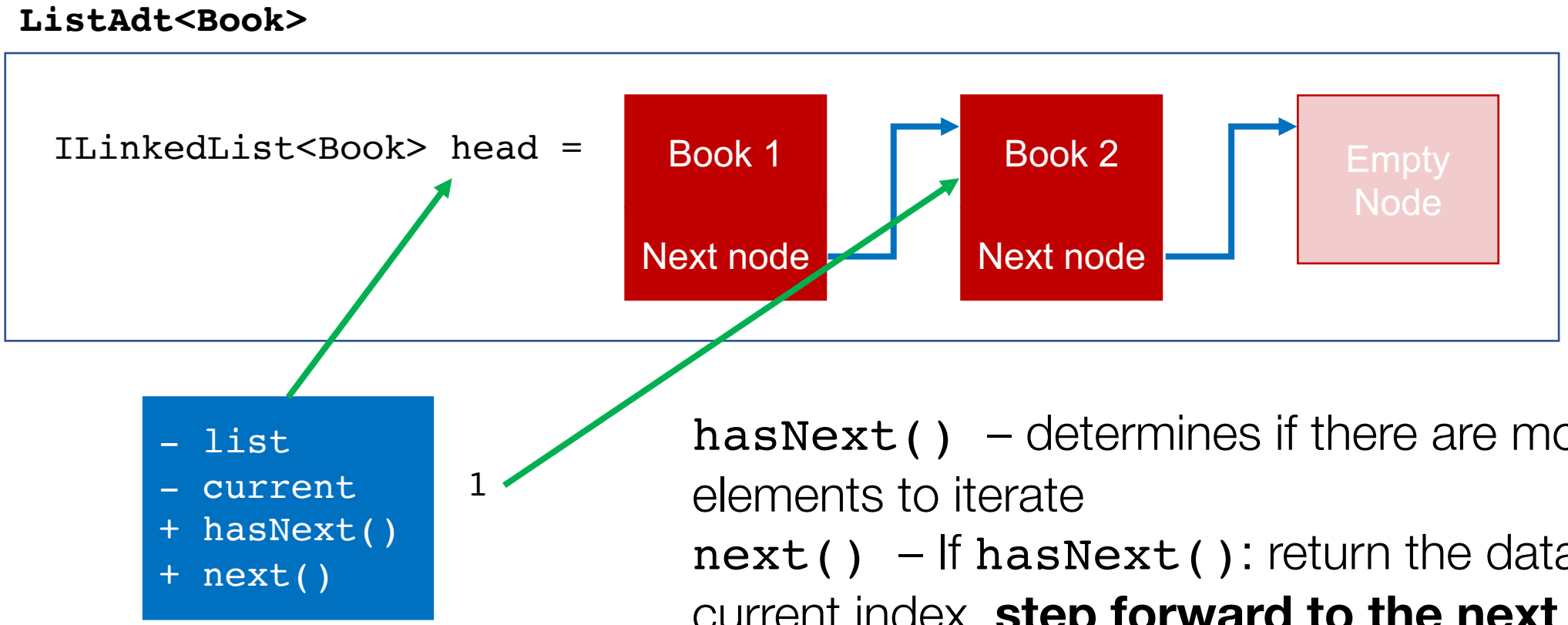


`hasNext()` – determines if there are more elements to iterate

`next()` – If `hasNext()`: **return the data at the current index**, step forward to the next index

How does `Iterator` work?

`Iterator` contains a field referencing the list (`ListAdt<Book>`) and a field referencing the current index

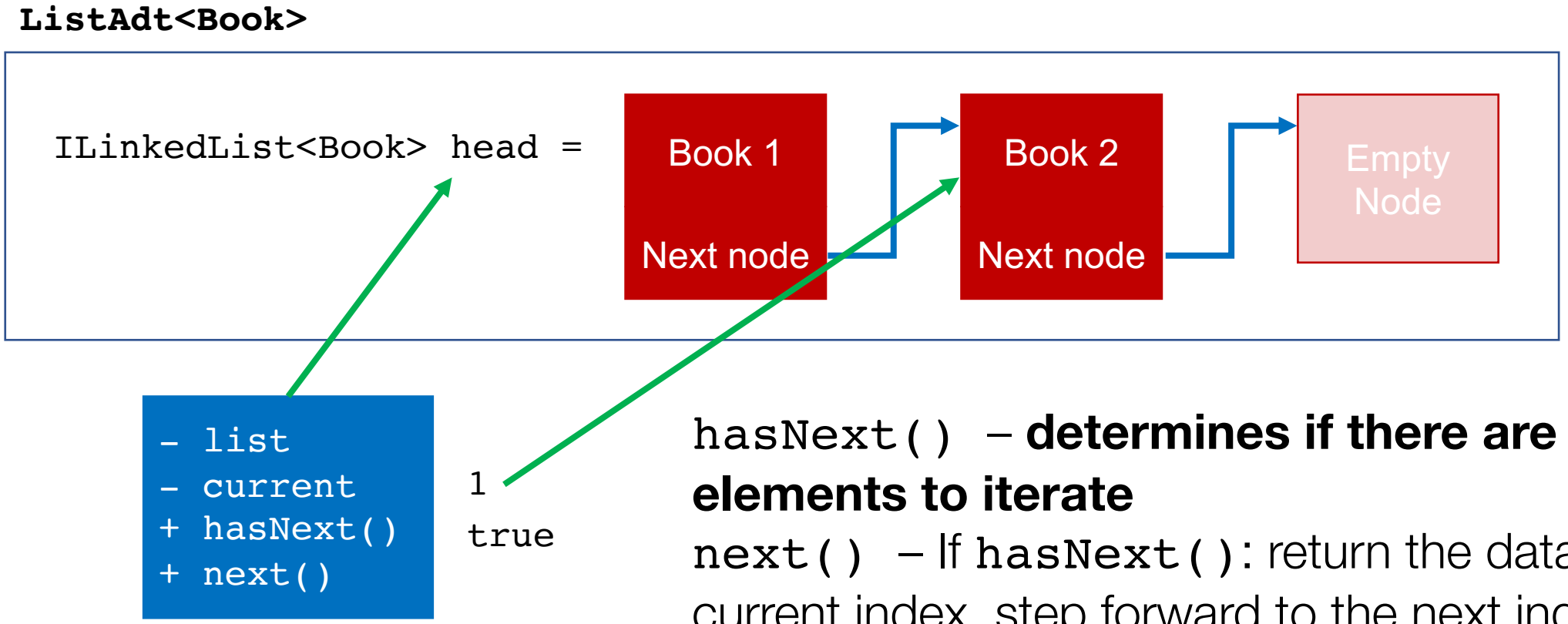


`hasNext()` – determines if there are more elements to iterate

`next()` – If `hasNext()`: return the data at the current index, **step forward to the next index**

How does `Iterator` work?

`Iterator` contains a field referencing the list (`ListAdt<Book>`) and a field referencing the current index

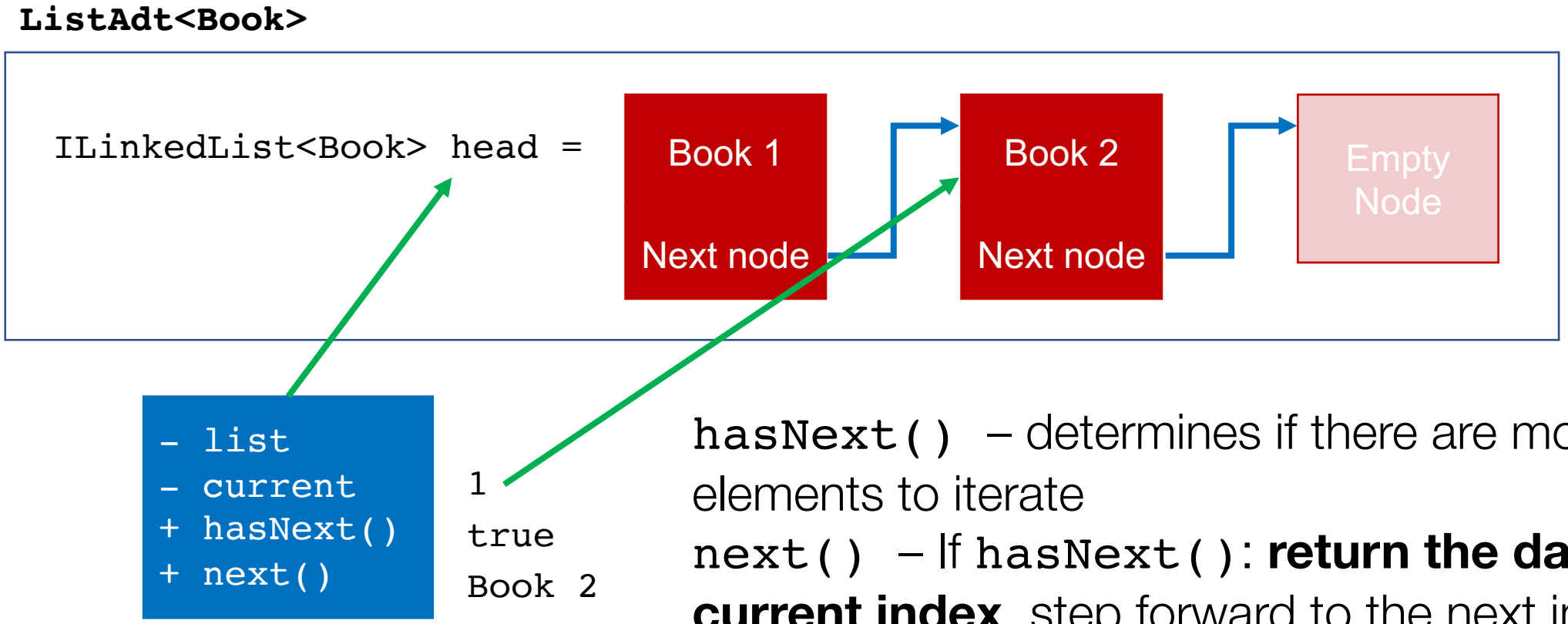


`hasNext()` – **determines if there are more elements to iterate**

`next()` – If `hasNext()`: return the data at the current index, step forward to the next index

How does `Iterator` work?

`Iterator` contains a field referencing the list (`ListAdt<Book>`) and a field referencing the current index

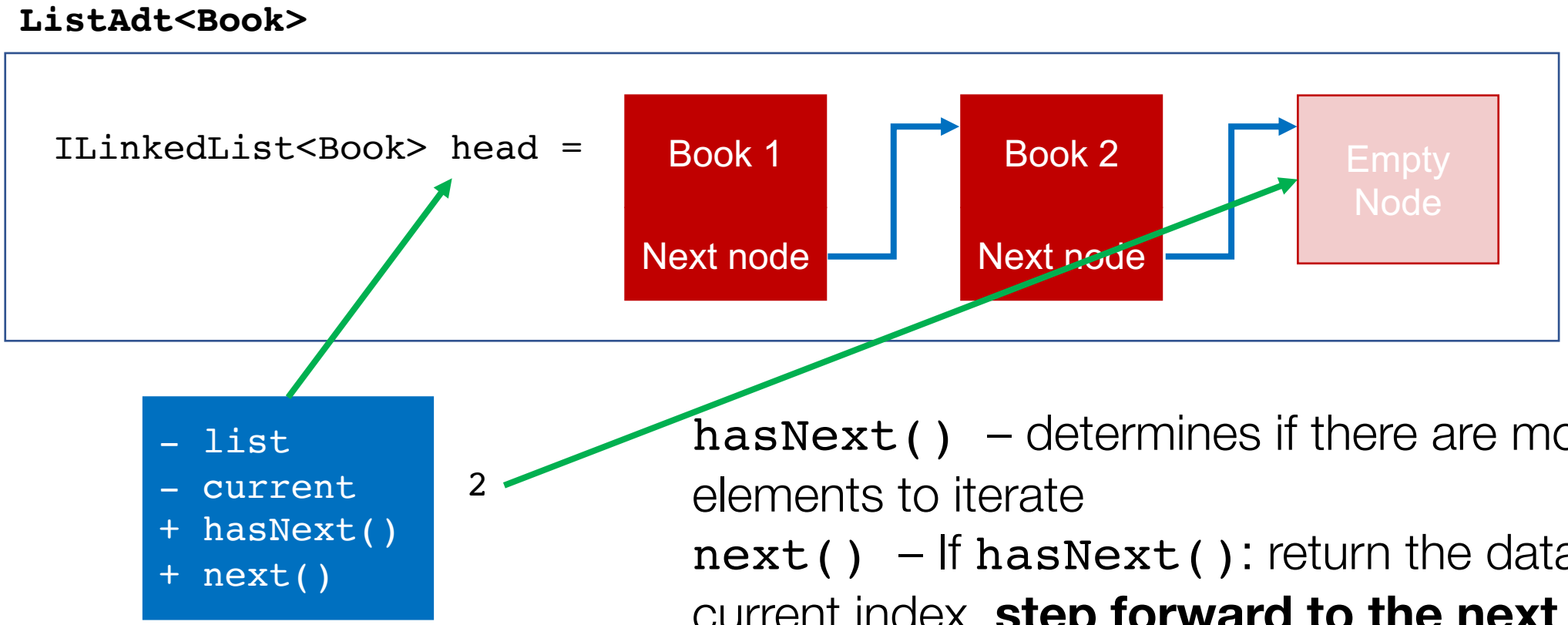


`hasNext()` – determines if there are more elements to iterate

`next()` – If `hasNext()`: **return the data at the current index**, step forward to the next index

How does `Iterator` work?

`Iterator` contains a field referencing the list (`ListAdt<Book>`) and a field referencing the current index

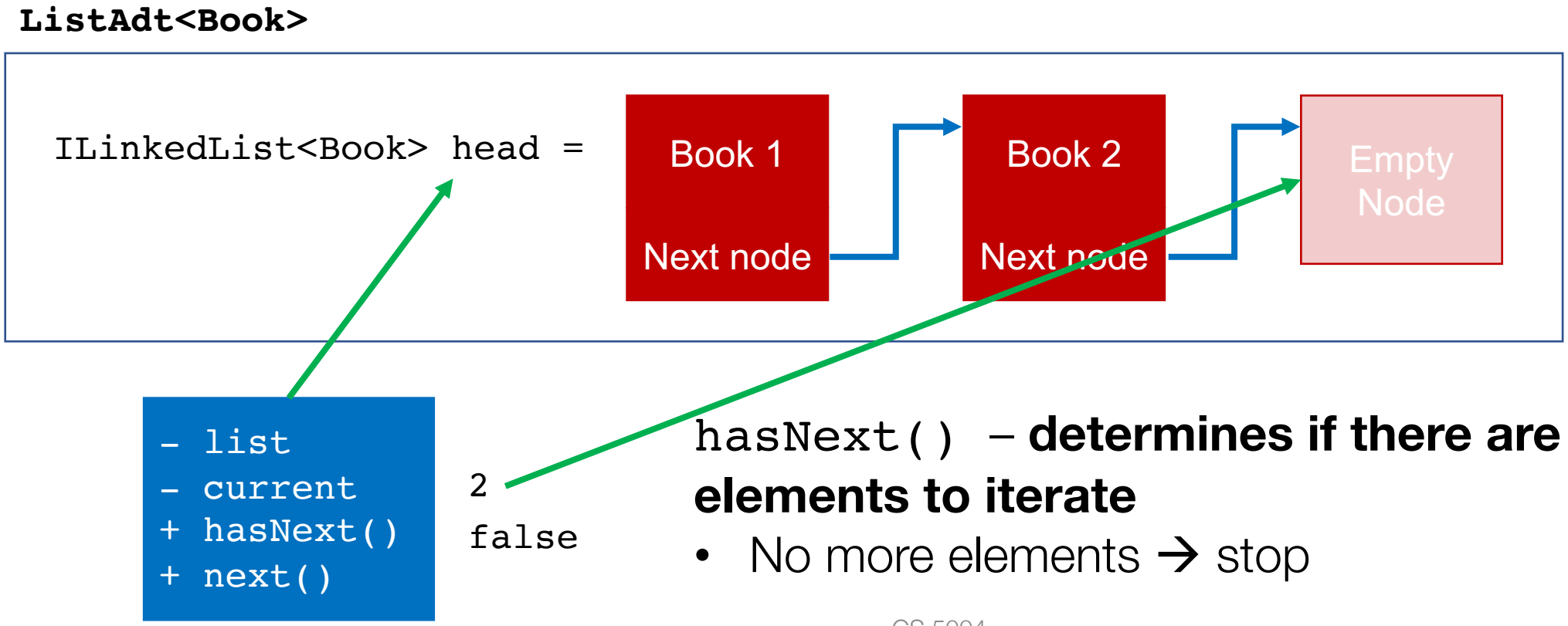


`hasNext()` – determines if there are more elements to iterate

`next()` – If `hasNext()`: return the data at the current index, **step forward to the next index**

How does `Iterator` work?

`Iterator` contains a field referencing the list (`ListAdt<Book>`) and a field referencing the current index



`hasNext()` – **determines if there are more elements to iterate**

- No more elements → stop

Walkthrough

- Lecture10 > iterating
  - ListAdt
  - ListIterator
  - Main

Break (10 mins)

Nested Classes

Nested Class

A class defined within some other class

```
public class Outer {
  ... fields & methods...
  private class Inner {
    ... fields & methods...
  }
}
```

Nested Class

Why use nested classes?

- A way of logically grouping classes only used in one place
- Increased encapsulation
- More readable and maintainable code

```
public class Outer {
   ... fields & methods...
   private class Inner {
      ... fields & methods...
   }
}
```

Nested Class

Scope bounded by scope of outer class

- Inner class cannot exist independently of outer class

```
public class Outer {
  ... fields & methods...
  private class Inner {
    ... fields & methods...
  }
}
```

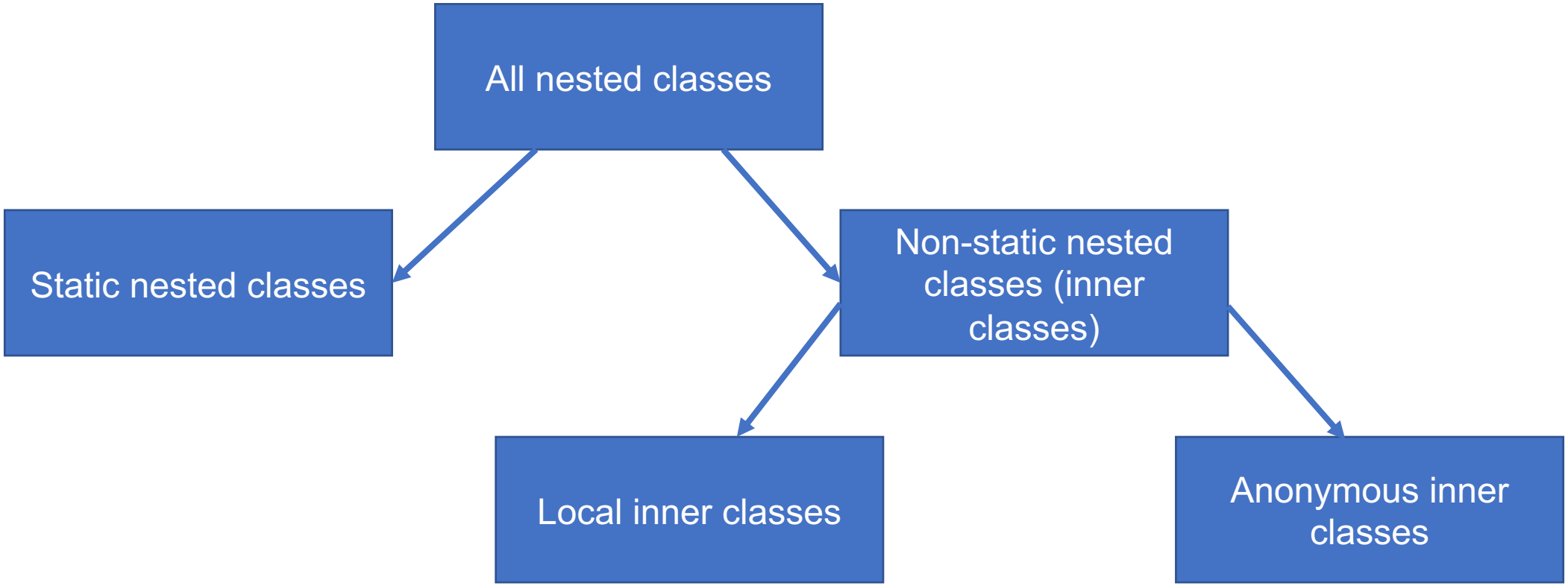
Nested Class

A member of its outer class

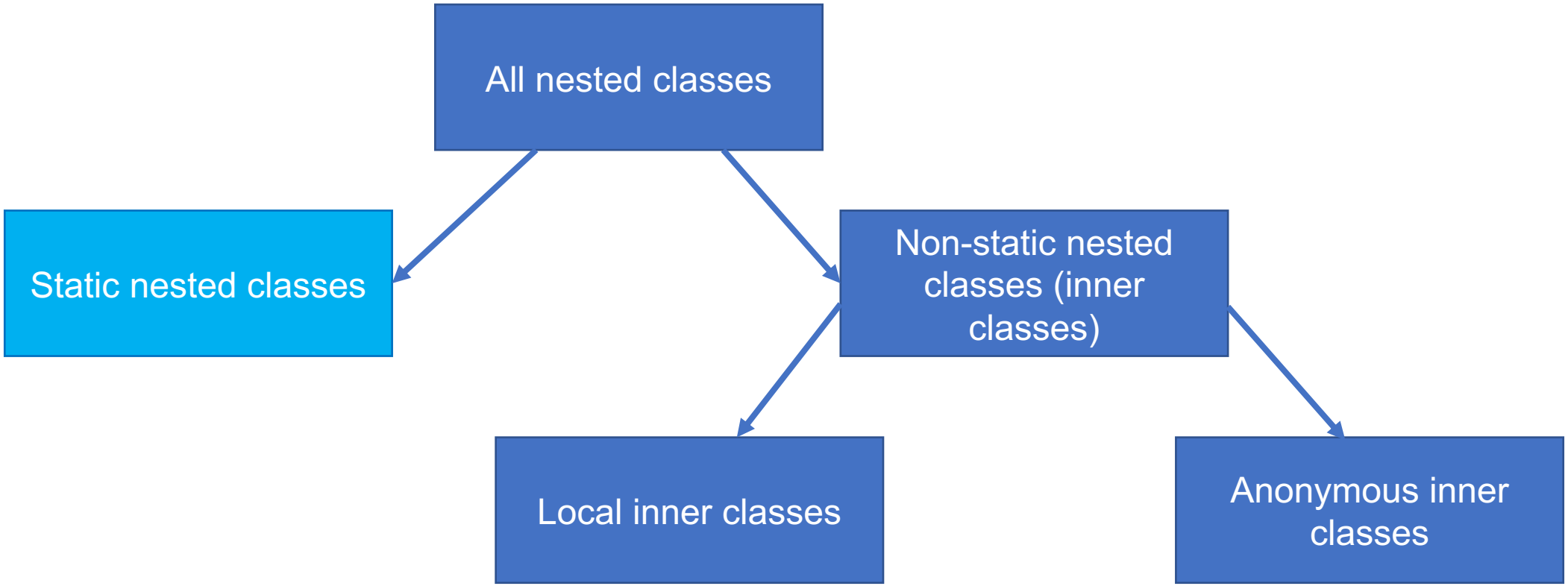
- Can be declared private, public, protected
- *Can have* access to all members of its outer class
- Outer class doesn't have access to members of the nested class

```
public class Outer {
   ... fields & methods...
   private class Inner {
      ... fields & methods...
   }
}
```

Terminology – types of nested classes



Terminology – types of nested classes



Static Nested Classes

```
public class Outer {
  private int x;
  private void myMethod() {…}
  private static class Inner {
    public void innerMethod() {



    }
  }
}
```

Static Nested Classes

Can't directly access non-static fields/methods of the outer class

```
public class Outer {
  private int x;
  private void myMethod() {…}
  private static class Inner {
    public void innerMethod() {
      x; → compile time error
      myMethod(); → compile time error
    }
  }
}
```

Static Nested Classes

…CAN directly access STATIC fields/methods of the outer class

```
public class Outer {
  private static int X;
  private static void myMethod() {…}
  private static class Inner {
    public void innerMethod() {
      X;
      myMethod();
    }
  }
}
```

Static Nested Classes

Outer class can use the static nested class like any other class

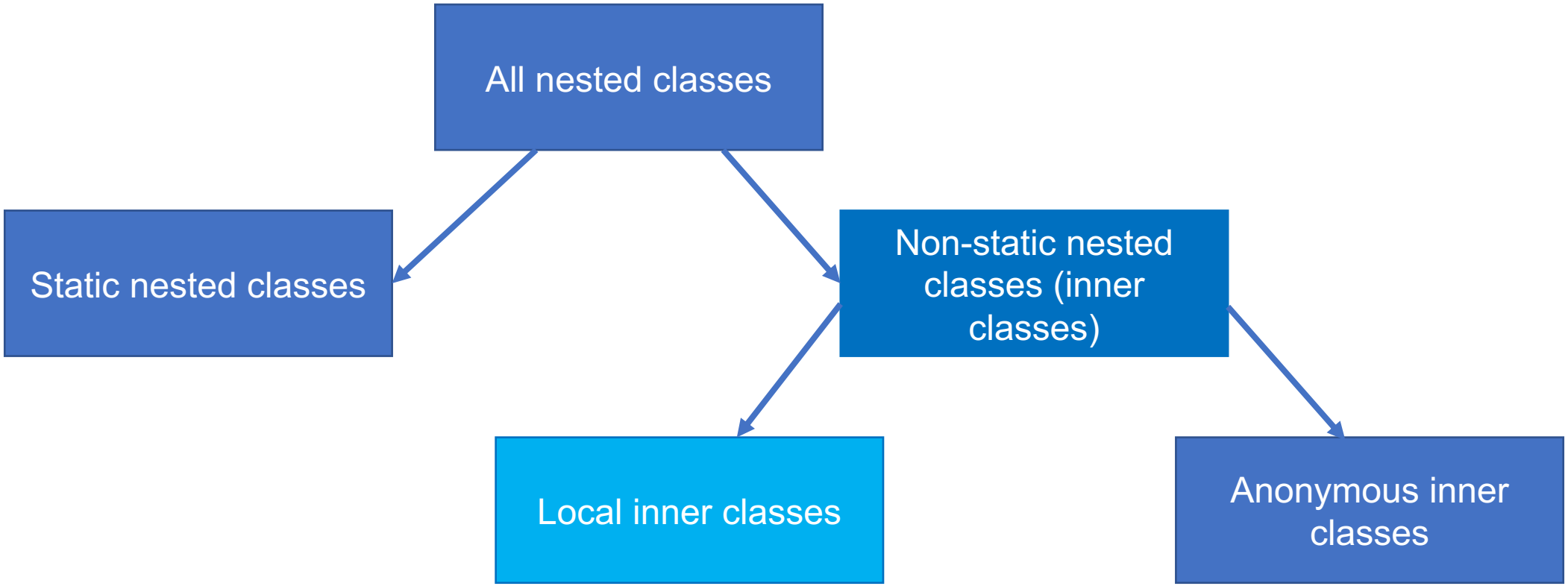
```
public class Outer {
  private Inner inner = new Inner();
  private void myMethod() {
    inner.innerMethod();
  }
  private static class Inner {
    public void innerMethod() {…}
  }
}
```

Static Nested Classes

Classes external to the outer class need to use the name of the outer class

- E.g. `Outer.Inner var = new Outer.Inner();`
- Note: only works if the static nested class has appropriate access (i.e not private)

Terminology – types of nested classes



Inner Classes

…are not static

```
public class Outer {
  private int x;
  private void myMethod() {…}
  private class Inner {
    public void innerMethod() {



    }
  }
}
```

Inner Classes

Can access all fields/methods of outer class. Don't use `this`

```
public class Outer {
  private int x;
  private void myMethod() {…}
  private class Inner {
    public void innerMethod() {
      x;
      myMethod();
    }
  }
}
```

Inner Classes

Outer class can use the inner class like any other class

```
public class Outer {
  private Inner inner = new Inner();
  private void myMethod() {
    inner.innerMethod();
  }
  private class Inner {
    public void innerMethod() {…}
  }
}
```

Inner Classes

Classes external to the outer class need to use the name of the outer class AND an object of the outer class

- E.g.
```
Outer outer = new Outer();
Outer.Inner var = outer.new Inner();
```
- Note: only works if the inner class has appropriate access (i.e not private)

Inner classes & shadowing

```
public class Outer {
  public String hello = “Hello";

  public class Inner {
    public String hello = “World”;
    public void sayHello() {System.out.println(hello);}
  }
}
```

Outer and inner instance variables have the same name

Inner classes & shadowing

```
public class Outer {
  public String hello = “Hello";

  public class Inner {
    public String hello = “World”;
    public void sayHello() {System.out.println(hello);}
  }
}
```

Higher-level variable will be overridden by lower-level variable

“World”

Inner classes & shadowing

```
public class Outer {
  public String hello = “Hello";

  public class Inner {
    public String hello = “World”;
    public void sayHello() {System.out.println(hello);}
  }
}
```

The higher level variable has been “shadowed” by the lower level variable

“World”

Inner classes & shadowing

Tends to lead to bugs…be careful

- Avoid accidental shadowing – don’t duplicate variable names
- Avoid shadowing on purpose (unless it’s really needed)

Walkthrough: A list with inner classes

- NestedList, implements the same IListAdt
- Instead of a separate ILinkedList implementation → inner classes for interfaces and nodes

`ListIterator` has redundant iteration

In the Iterator:

```
@Override
public T next() {
  T item
   = this.list.get(this.current);
  this.current++;
  return item;
}
```

ListAdt/NestedList get implementation

```
@Override
public T get(int index) throws ...
{
    // removed for space
    while (i != index) {
      current = current.getRest();
      i++;
    }
    return current.getItem();
}
```

Exercise: Remove redundancy

Instead of using `ListIterator`, make `ILinkedListIterable` iterable

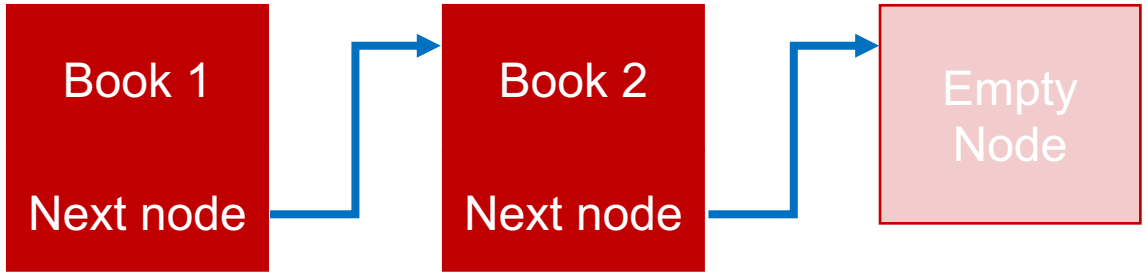
- Make a static inner `Iterator` class to iterate through the nodes
- Connect it to `NestedList`'s `iterator` method
- Run `main` to check it still works

We'll walk through the concepts before you get started

How does `Iterator` work?

The node at the head of the list returns an `Iterator` object

- Because it implements Iterable
- Iterator shown in blue




```
- current node
+ hasNext()
+ next()
```

Inner iterator to be implemented

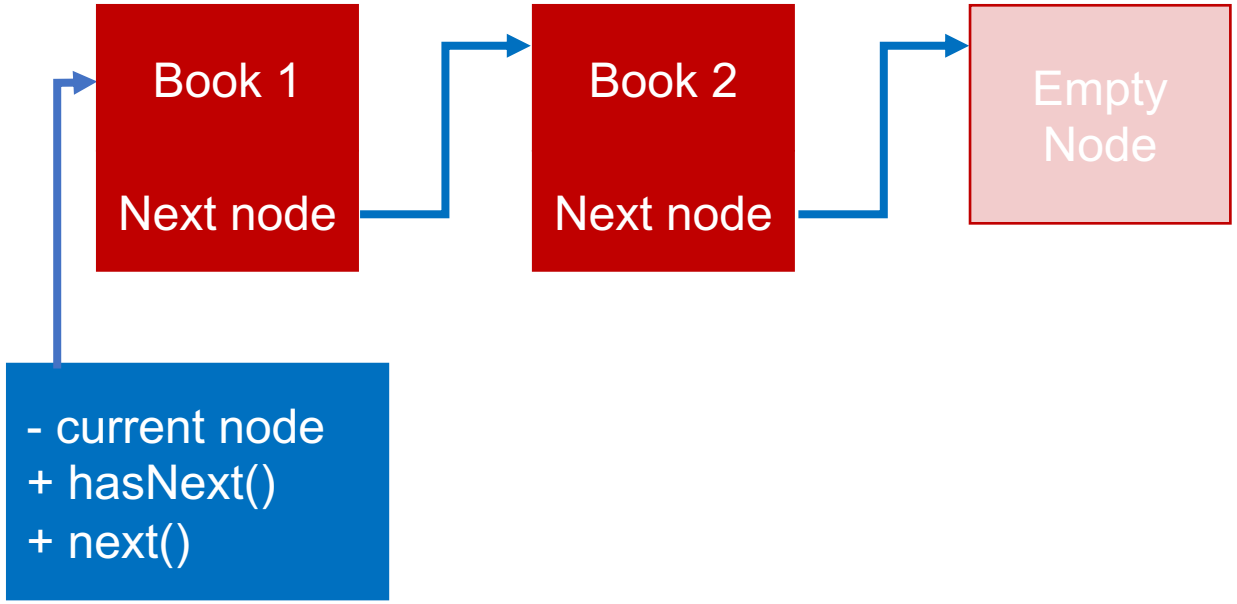
(Existing iterator, for reference)

```
- list
- current
+ hasNext()
+ next()
```

How does `Iterator` work?

`Iterator` contains a field referencing the current node

- Instead of list and current index



Inner iterator to be implemented

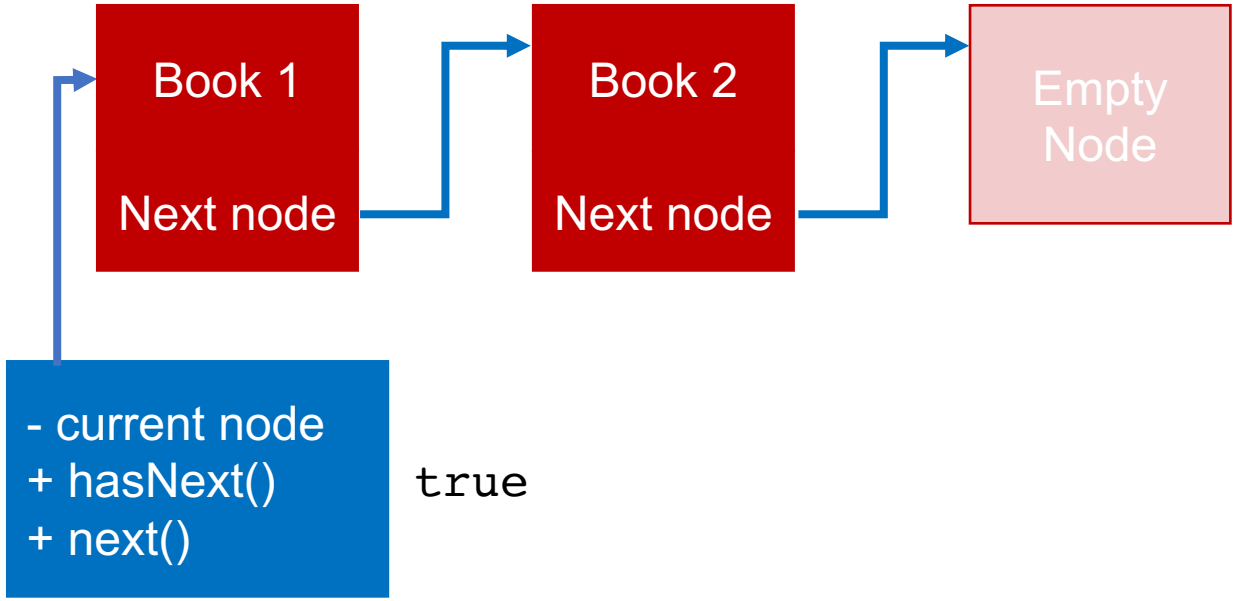
(Existing iterator, for reference)

```
- list
- current
+ hasNext()
+ next()
```

How does `Iterator` work?

`Iterator` contains a field referencing the current node

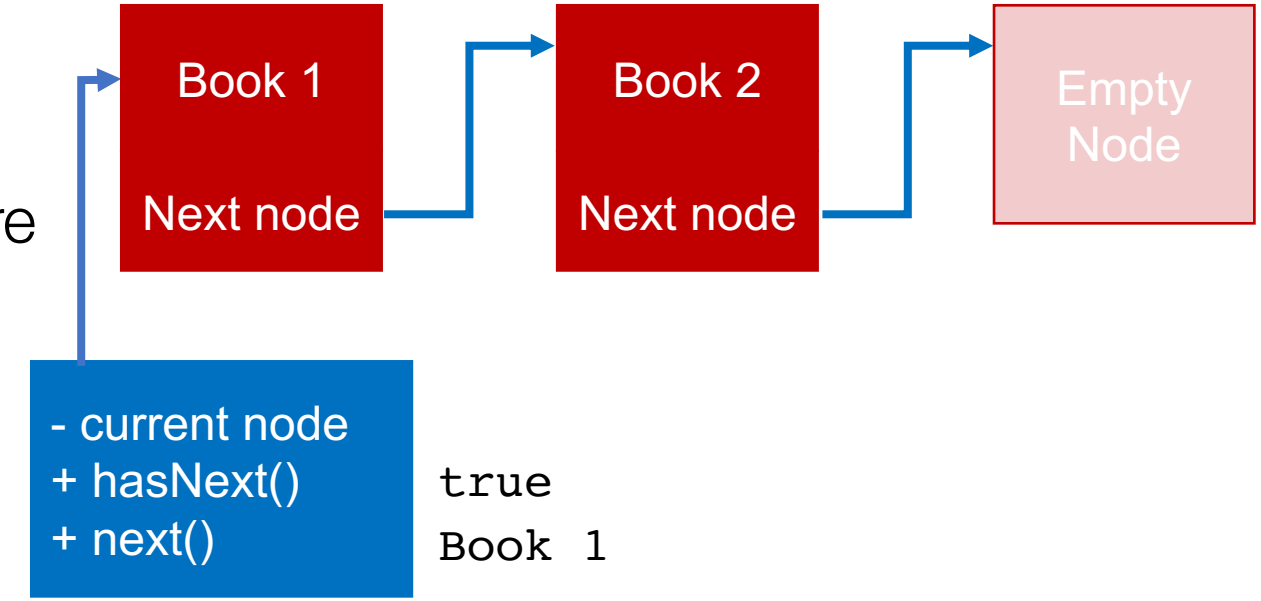
- **`hasNext()`** – determines if there are more elements to iterate



How does `Iterator` work?

`Iterator` contains a field referencing the current node

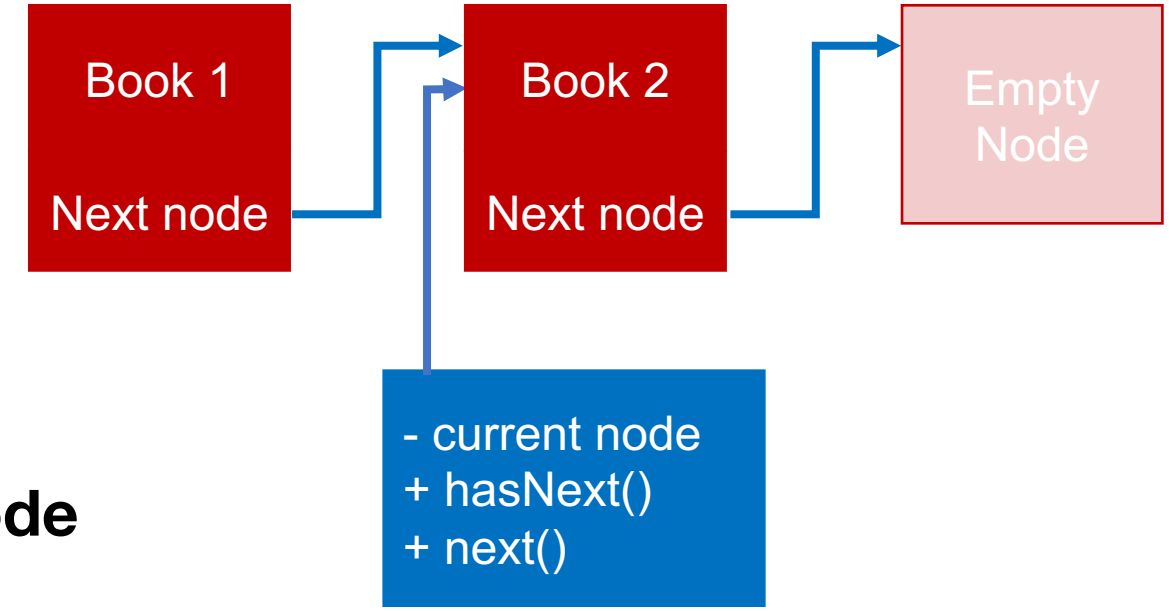
- `hasNext()` – determines if there are more elements to iterate
- `next()` – If `hasNext()`, **return the data at the current node**, step forward to the next node



How does `Iterator` work?

`Iterator` contains a field referencing the current node

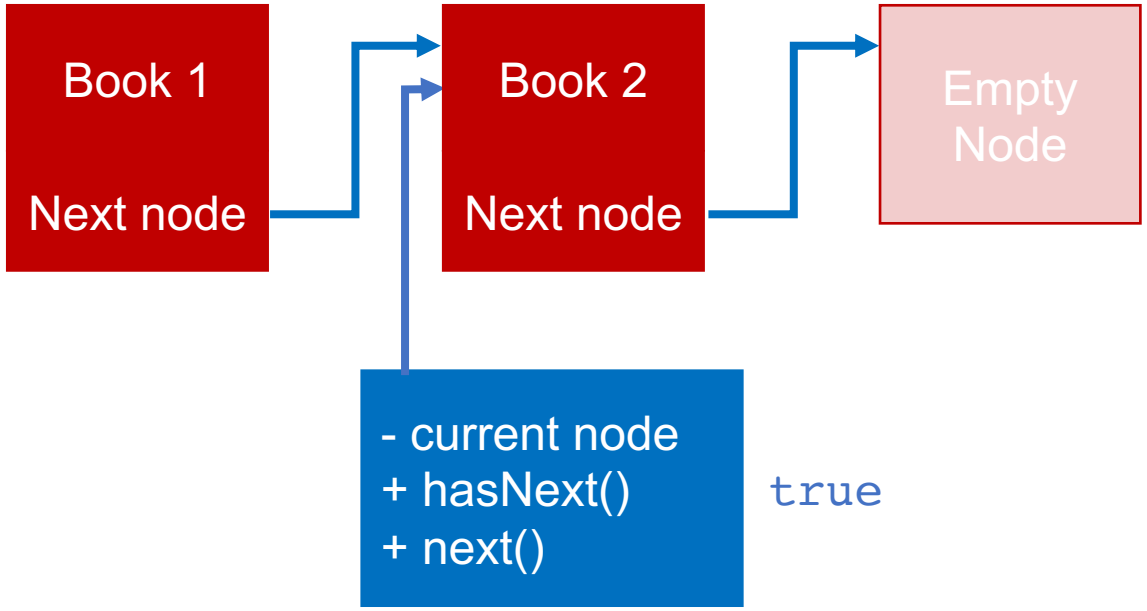
- `hasNext()` – determines if there are more elements to iterate
- `next()` – If `hasNext()`, return the data at the current node, **step forward to the next node**



How does `Iterator` work?

`Iterator` contains a field referencing the current node

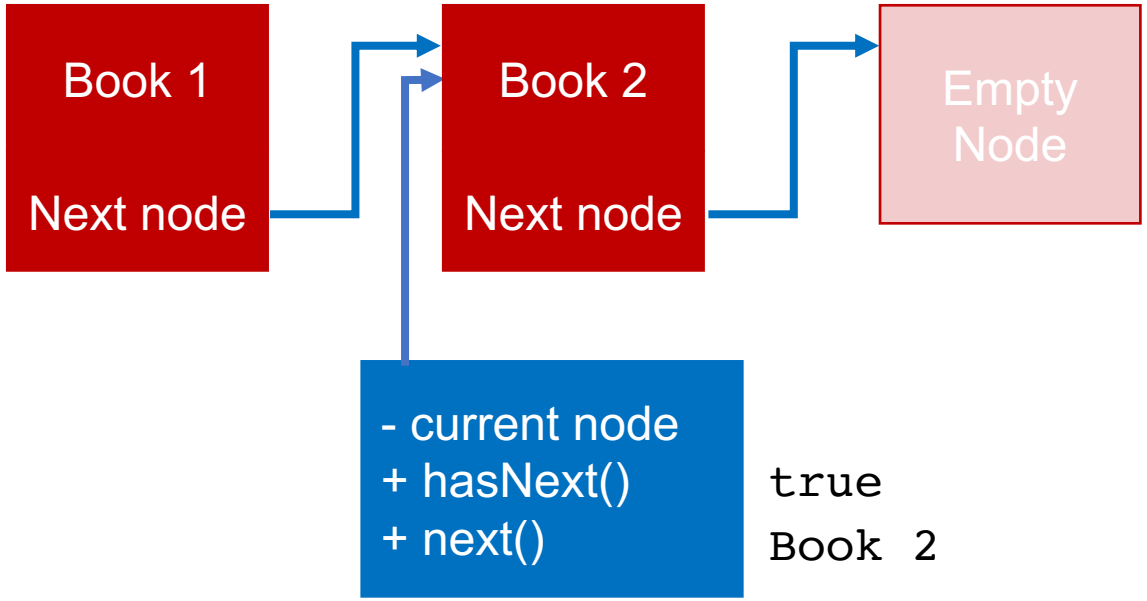
- `hasNext()` – **determines if there are more elements to iterate**
- `next()` – If `hasNext()`, return the data at the current node, step forward to the next node



How does `Iterator` work?

`Iterator` contains a field referencing the current node

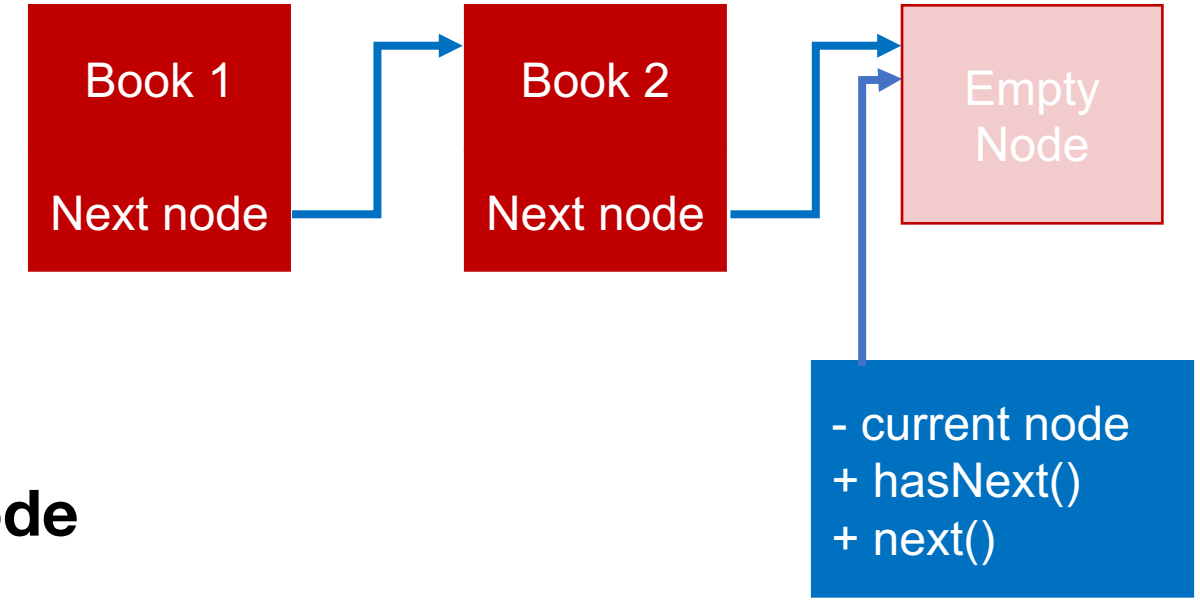
- `hasNext()` – determines if there are more elements to iterate
- `next()` – If `hasNext()`, **return the data at the current node**, step forward to the next node



How does `Iterator` work?

`Iterator` contains a field referencing the current node

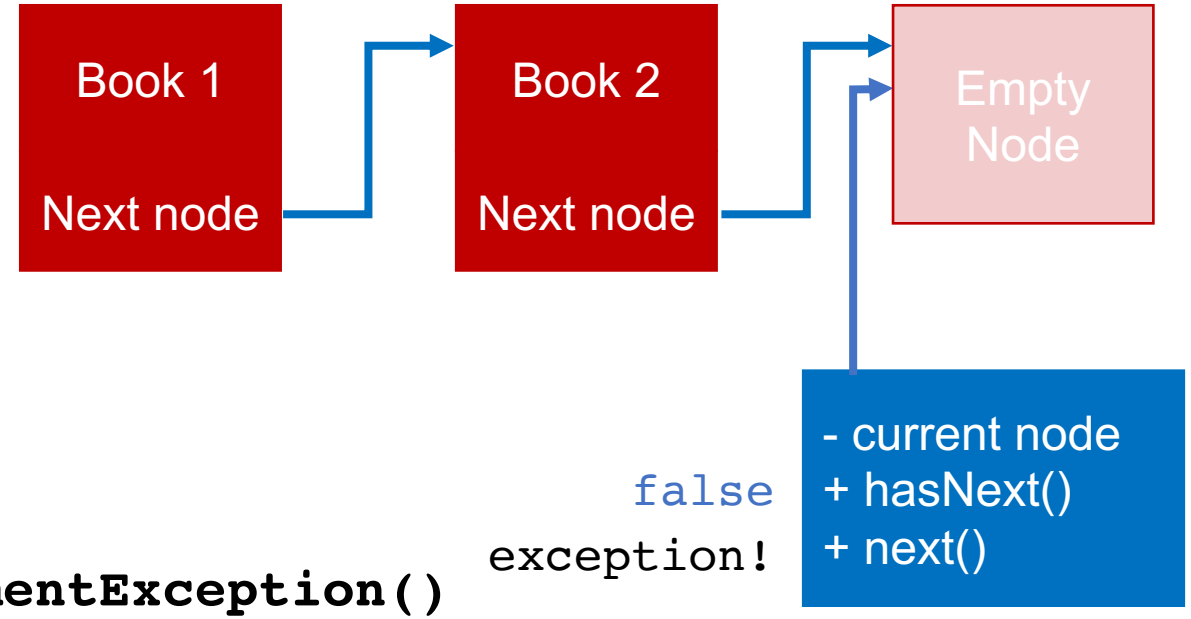
- `hasNext()` – determines if there are more elements to iterate
- `next()` – If `hasNext()`, return the data at the current node, **step forward to the next node**



How does `Iterator` work?

`Iterator` contains a field referencing the current node

- **`hasNext()`** – determines if there are more elements to iterate
- If **`hasNext()`** – return the data at the current node, step forward to the next node
  - Else stop & **`throw new NoSuchElementException()`**



Exercise: Remove redundancy

Instead of using `ListIterator`, make `ILinkedListIterable` iterable

- Make a static inner `Iterator` class to iterate through the nodes
- Connect it to `NestedList`'s `iterator` method
- Run `main` to check it still works

More data structures

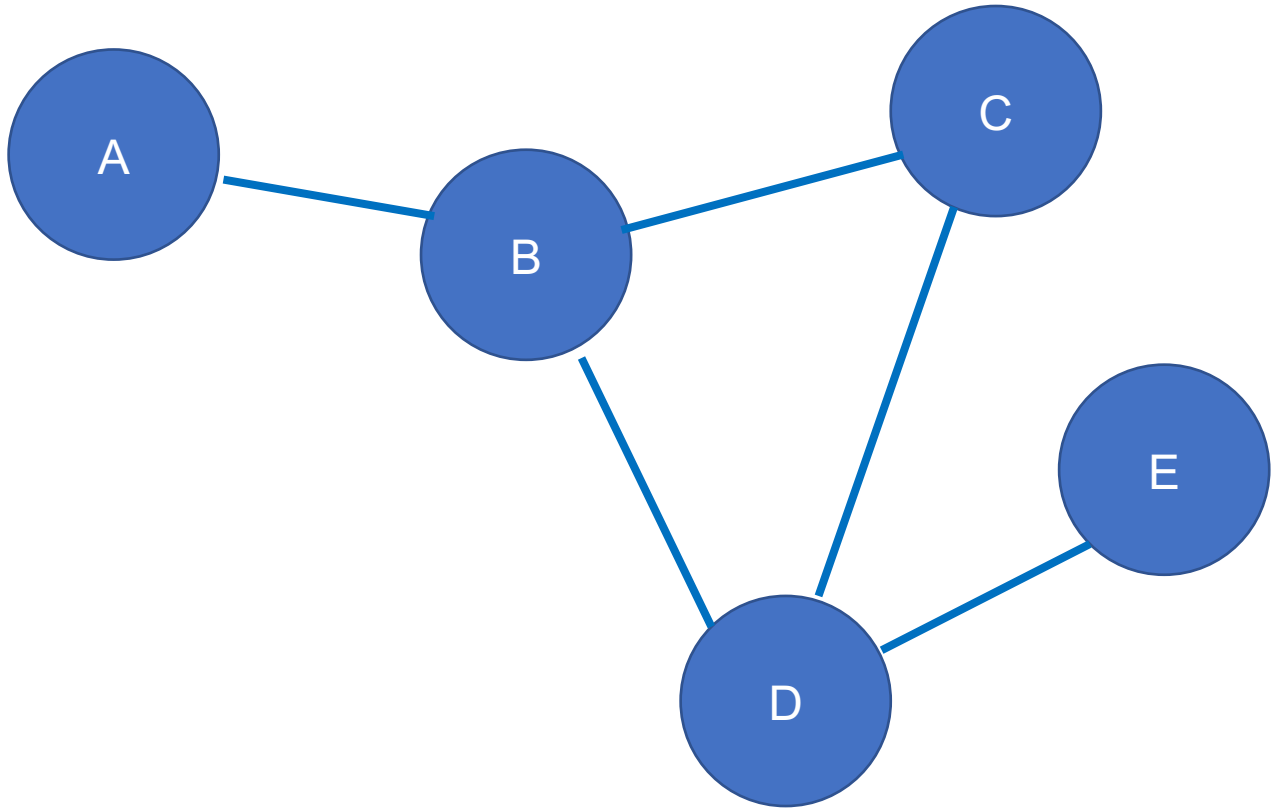
Implementing graphs and trees in Java

Graphs in Java

Graphs

- Already covered in 5008 (?)
- Quick recap of terminology and key characteristics
- Walkthrough: Java implementation of a generic undirected graph

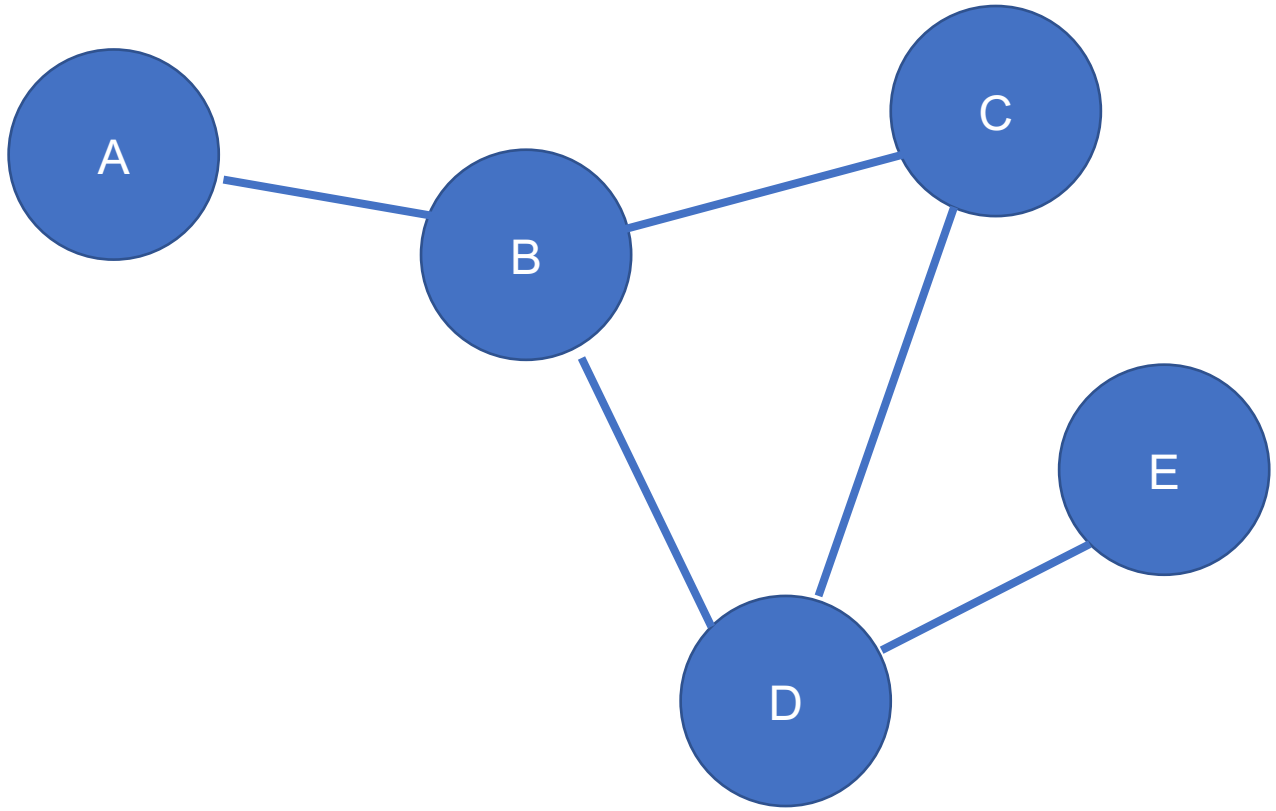
Graph data structure



Made up of:

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

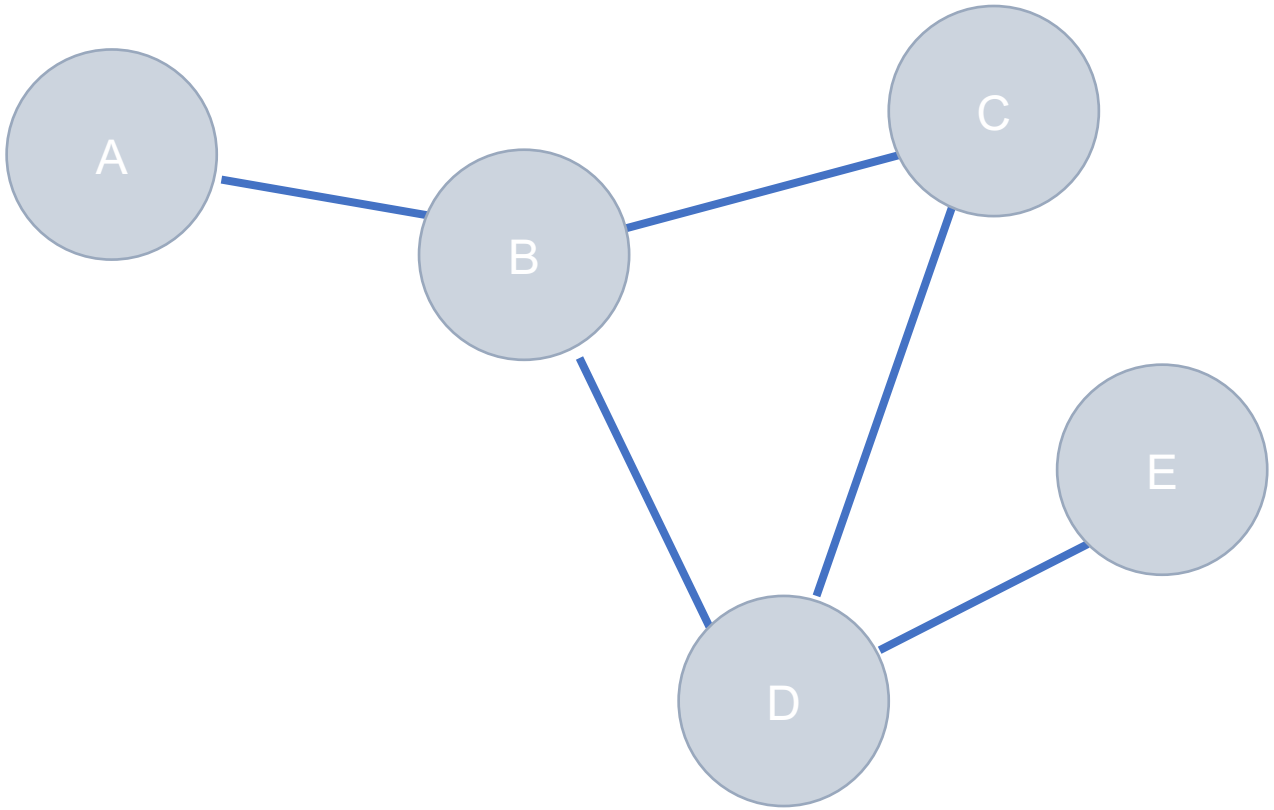
Graph data structure



Graphs are more general than trees

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

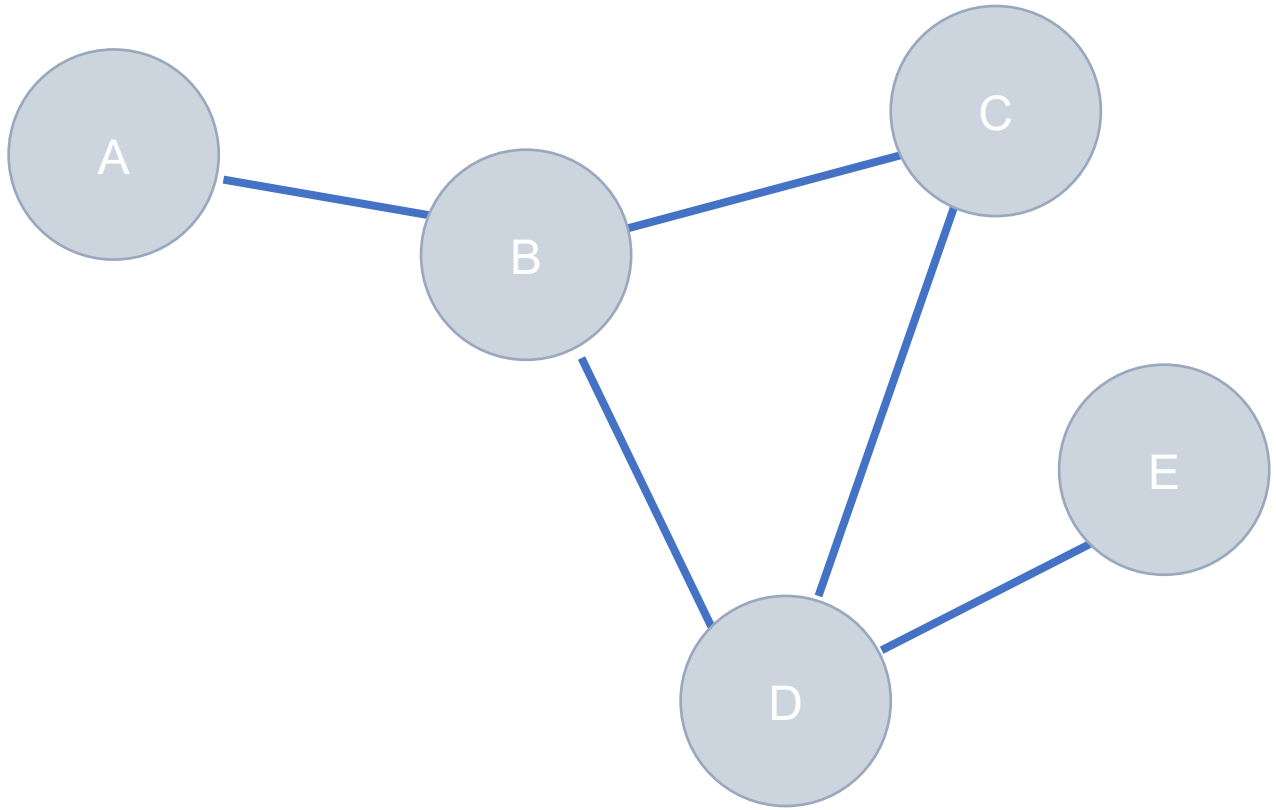
Graph data structure



Edges can be **undirected**:

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

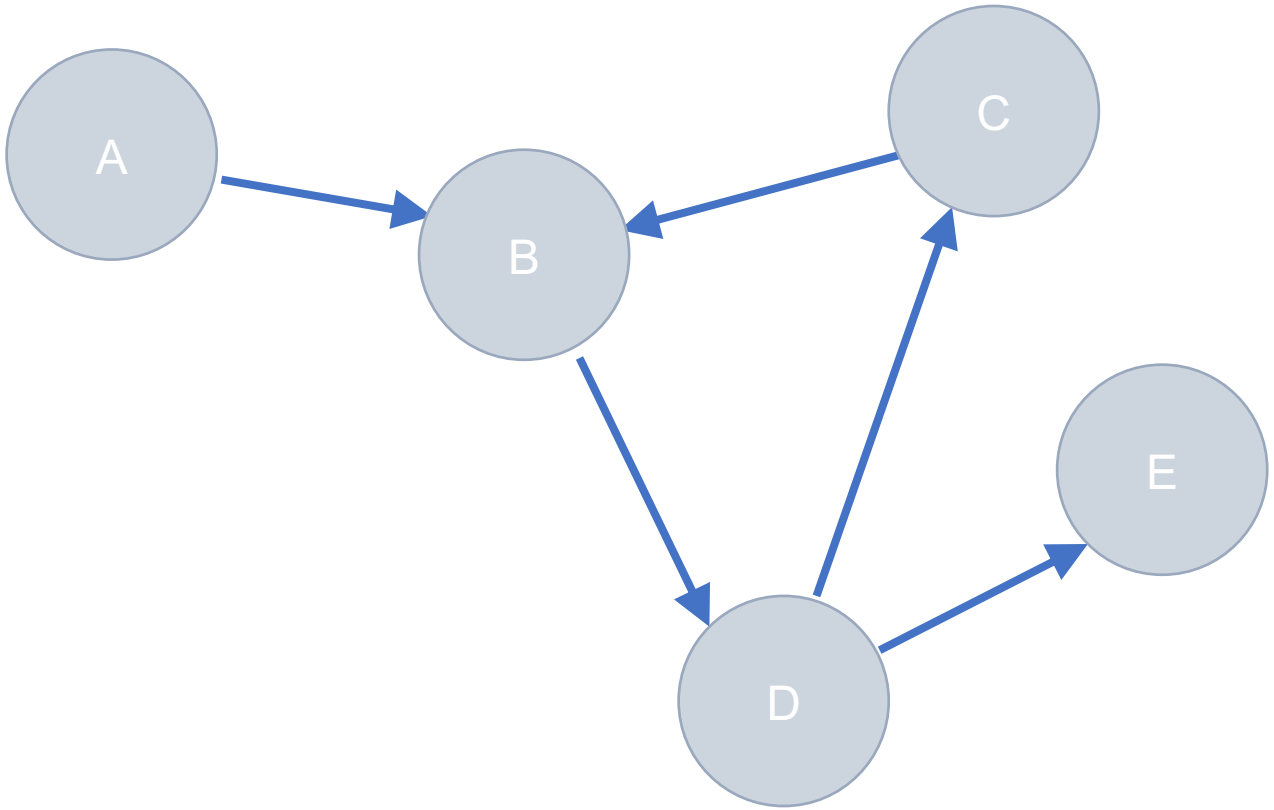
Graph data structure



Edges can be **undirected**:

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

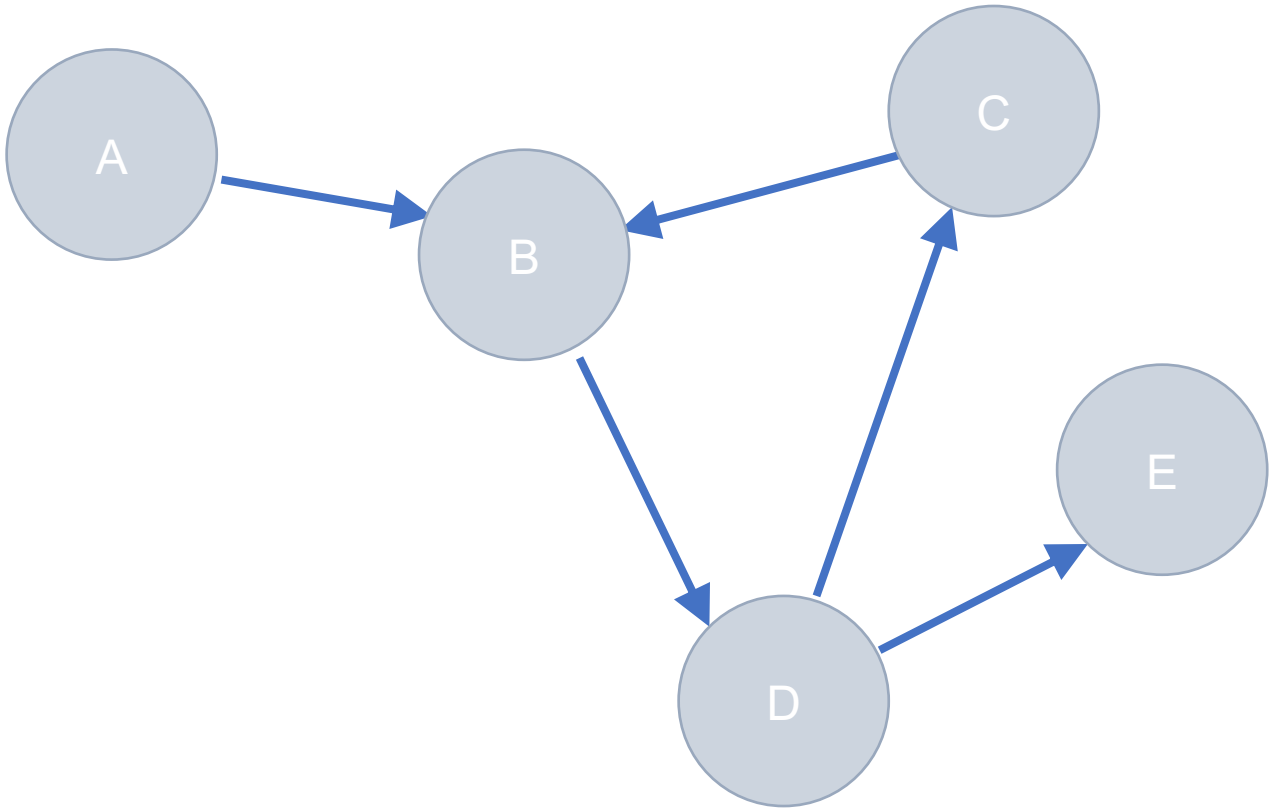
Graph data structure



Edges can be **directed**:

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

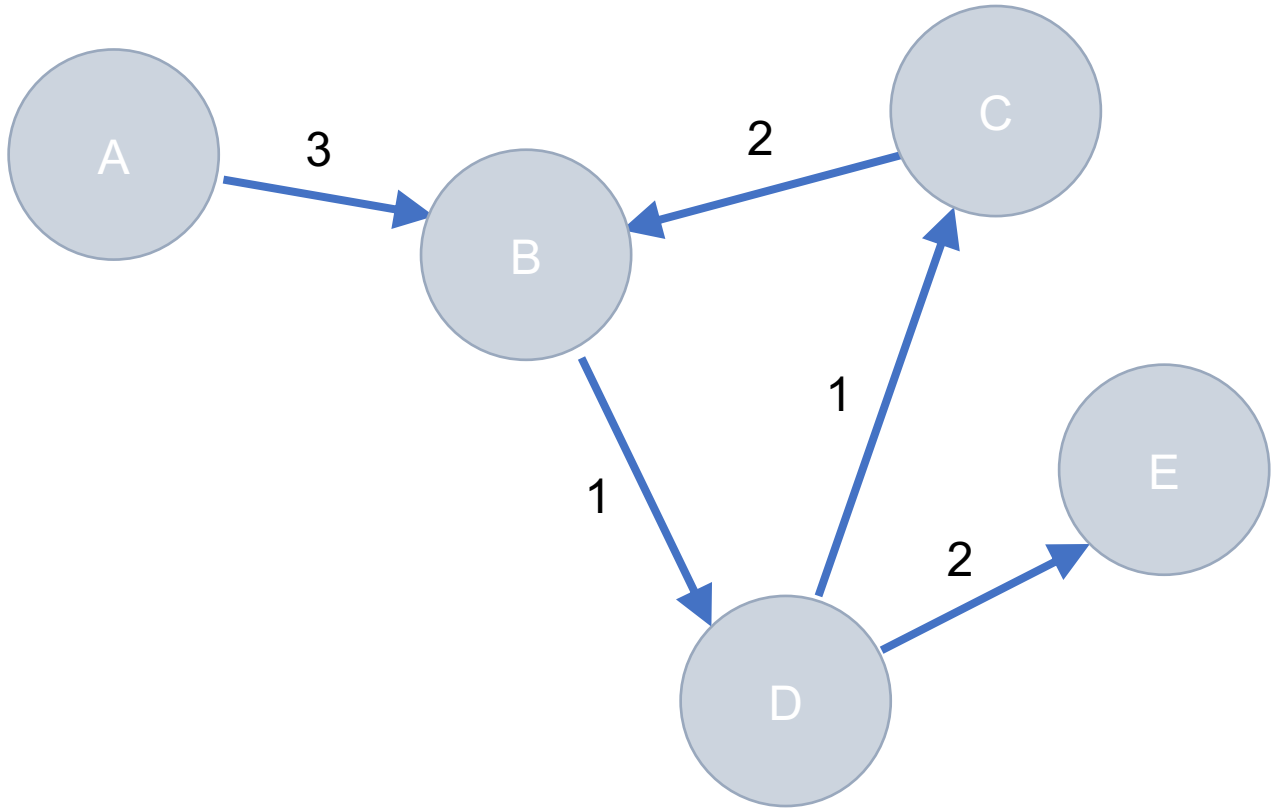
Graph data structure



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:

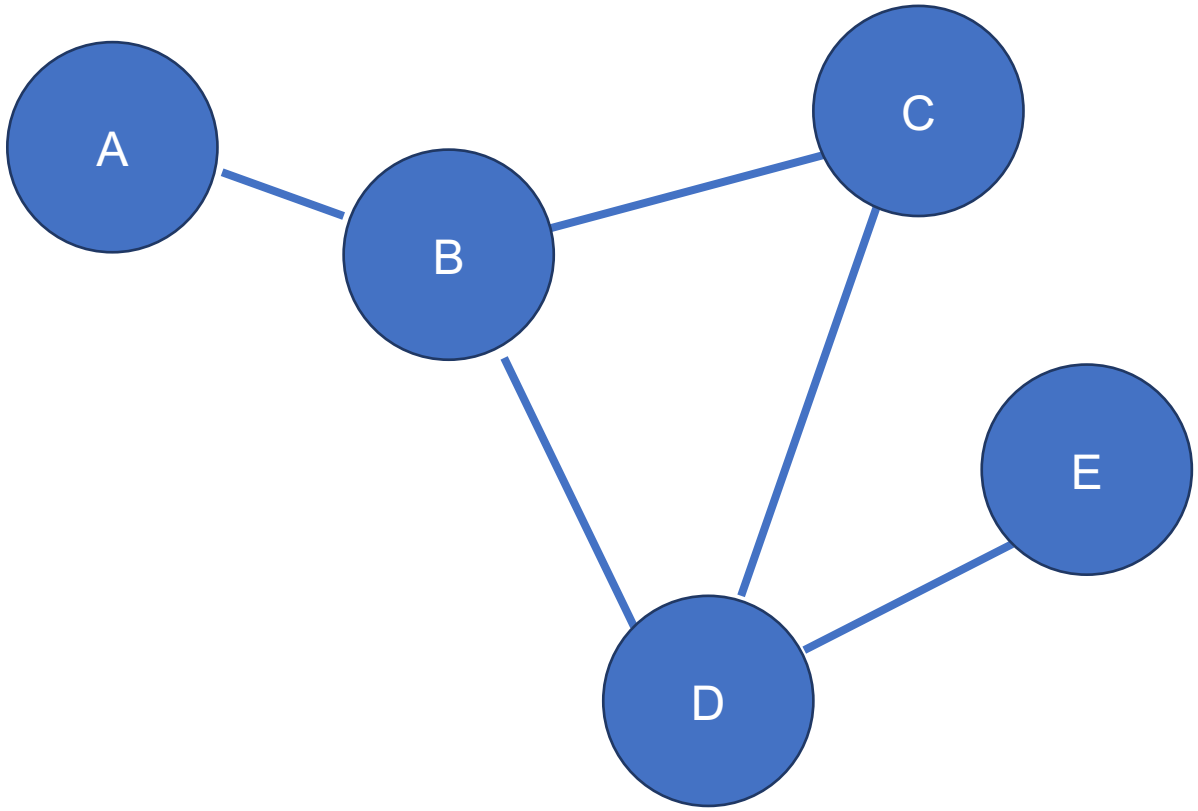
- Adjacency matrix
- Adjacency list

Graph representation

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

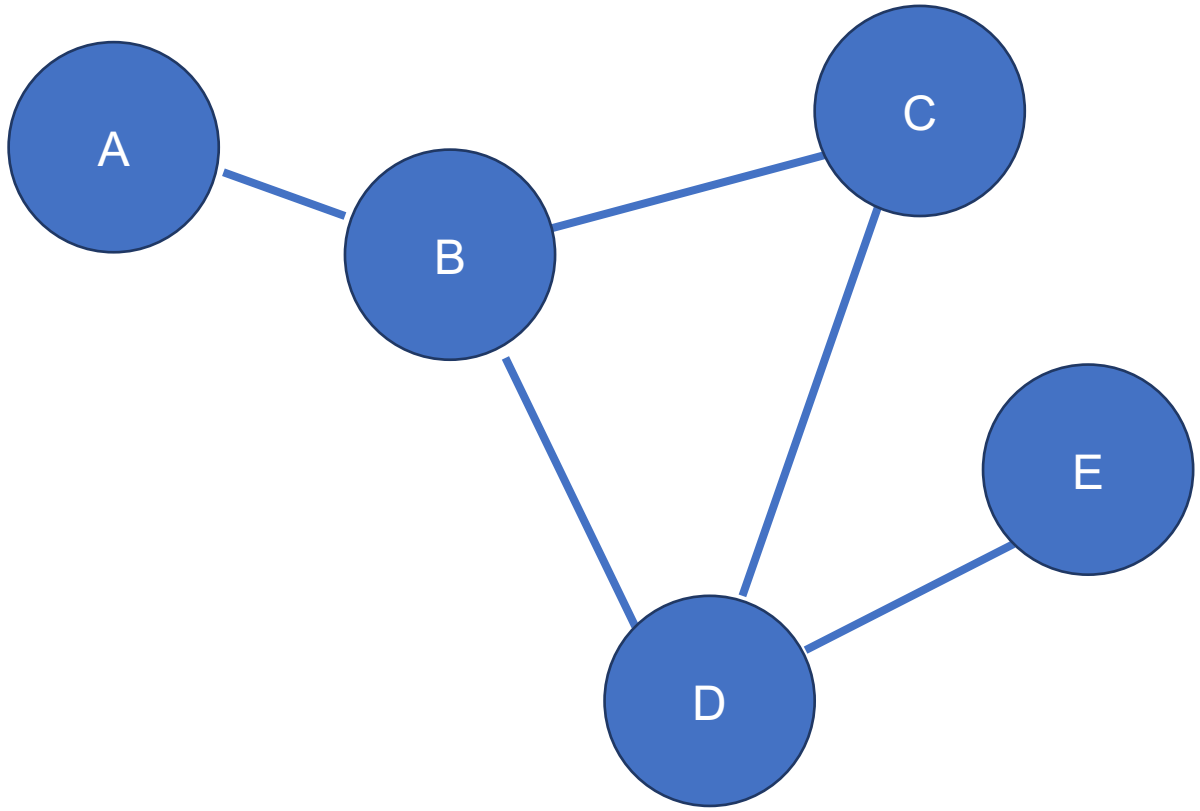
- If some edge *(u, v)* exists, set *A[u, v]* to 1 (true)
  - weighted graph: set to edge weight
- Else, set *A[u,v]* to 0 (false)
- Appropriate representation if a graph is dense
  - Nearly all possible edges are present

Undirected graph – adjacency matrix



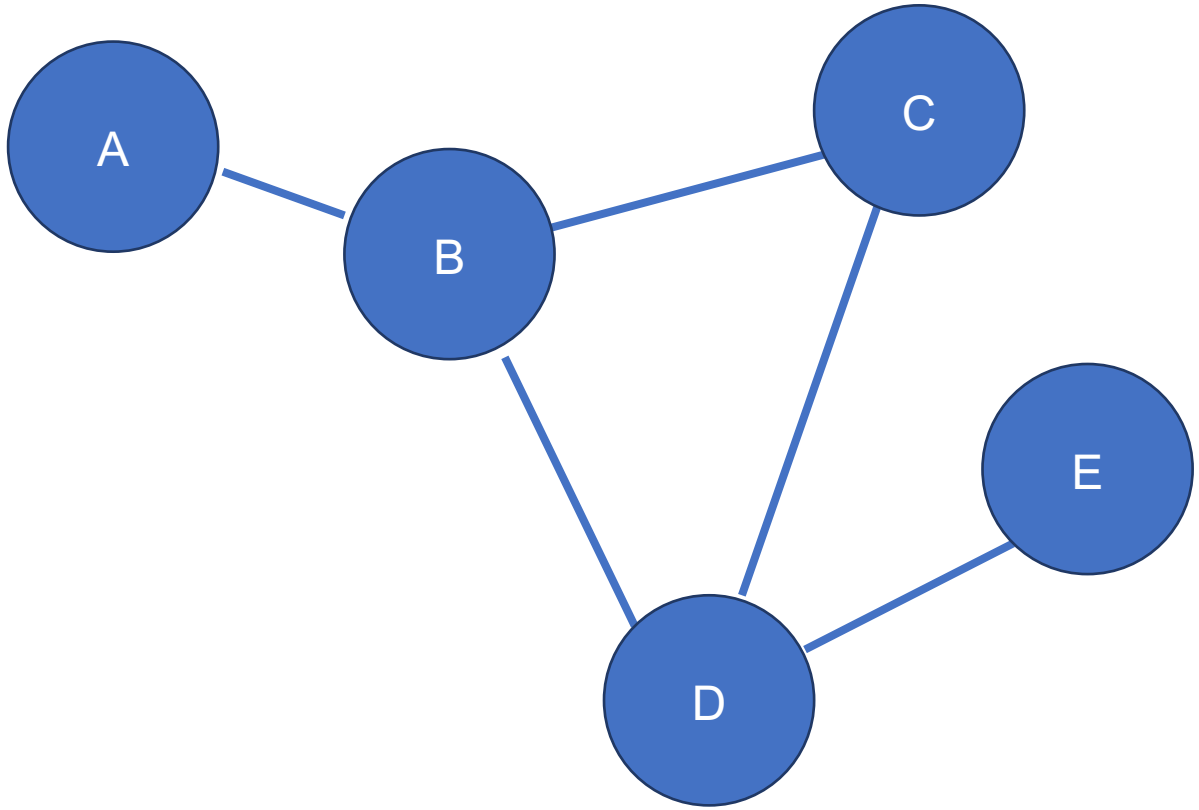
	0 (A)	1 (B)	2 (C)	3 (D)	4 (E)
0 (A)	0	1	0	0	0
1 (B)	1	0	1	1	0
2 (C)	0	1	0	1	0
3 (D)	0	1	1	0	1
4 (E)	0	0	0	1	0

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

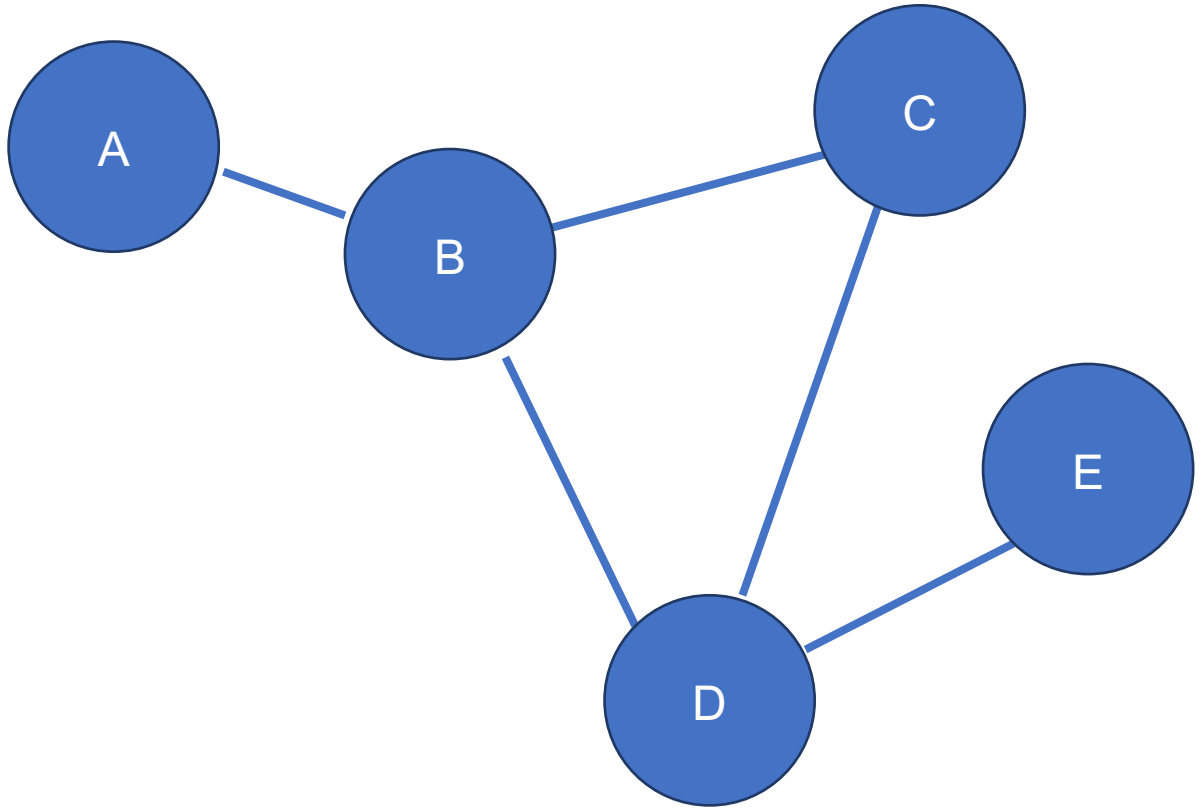


Create a 2D array:

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

Length of each column = # of nodes/vertices

Undirected graph – adjacency matrix



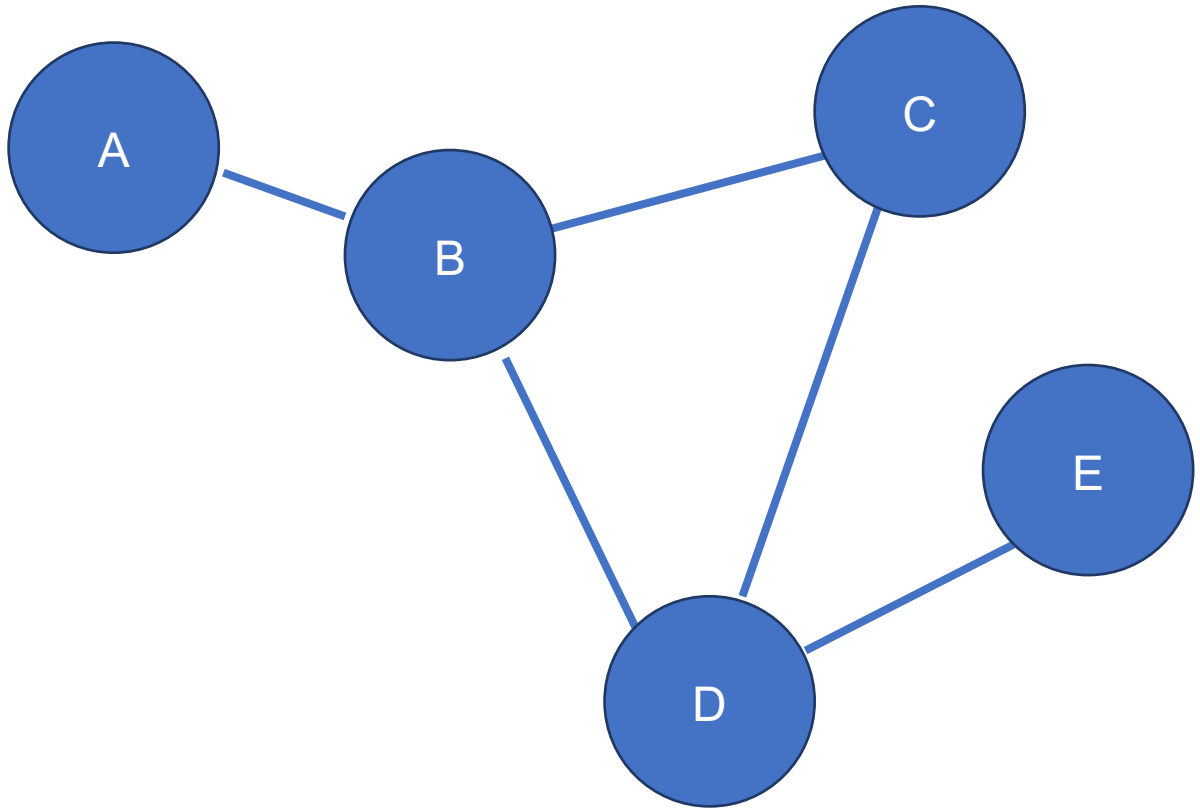
Assuming index 0 = A, 1 = B etc:

Connect A - B (& B – A)

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

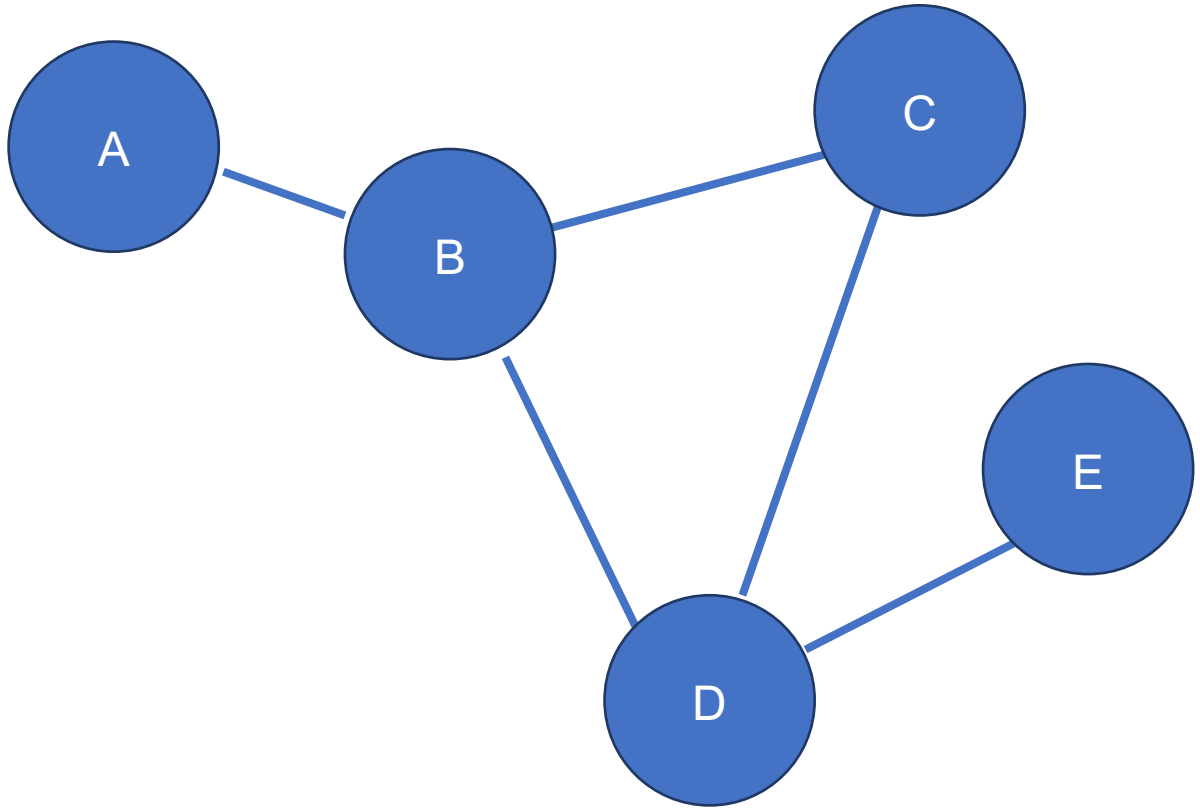
Connections go both ways

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

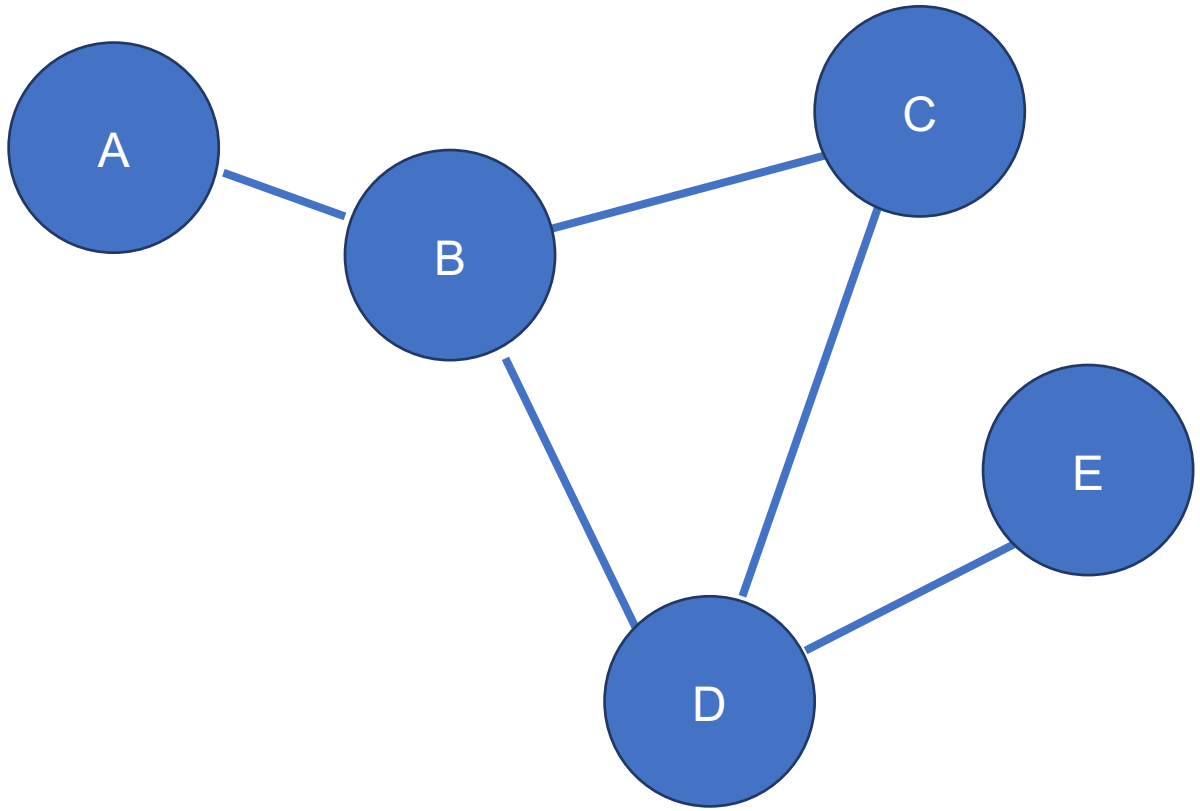
Undirected graph – adjacency matrix



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

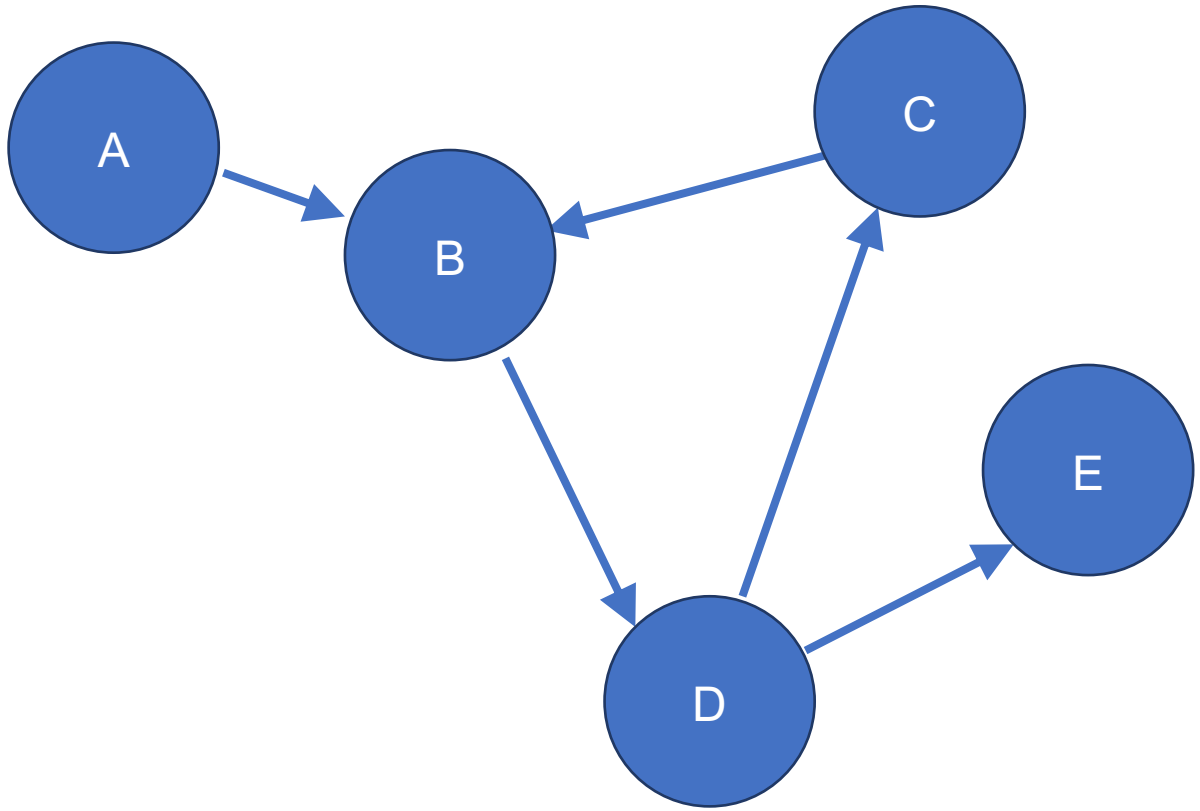
```
graph[0][0] = 0;
```

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:

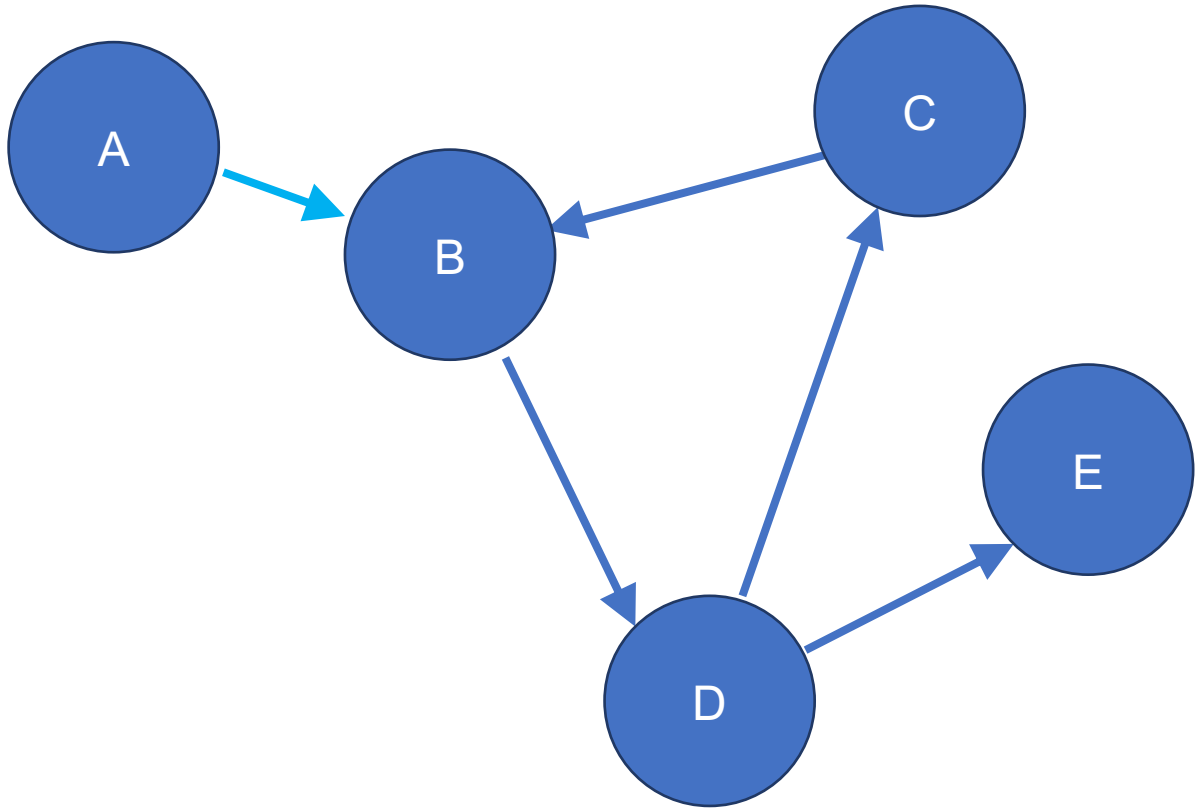
Connect A to B

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

But not B to A

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

Directed graph – adjacency matrix



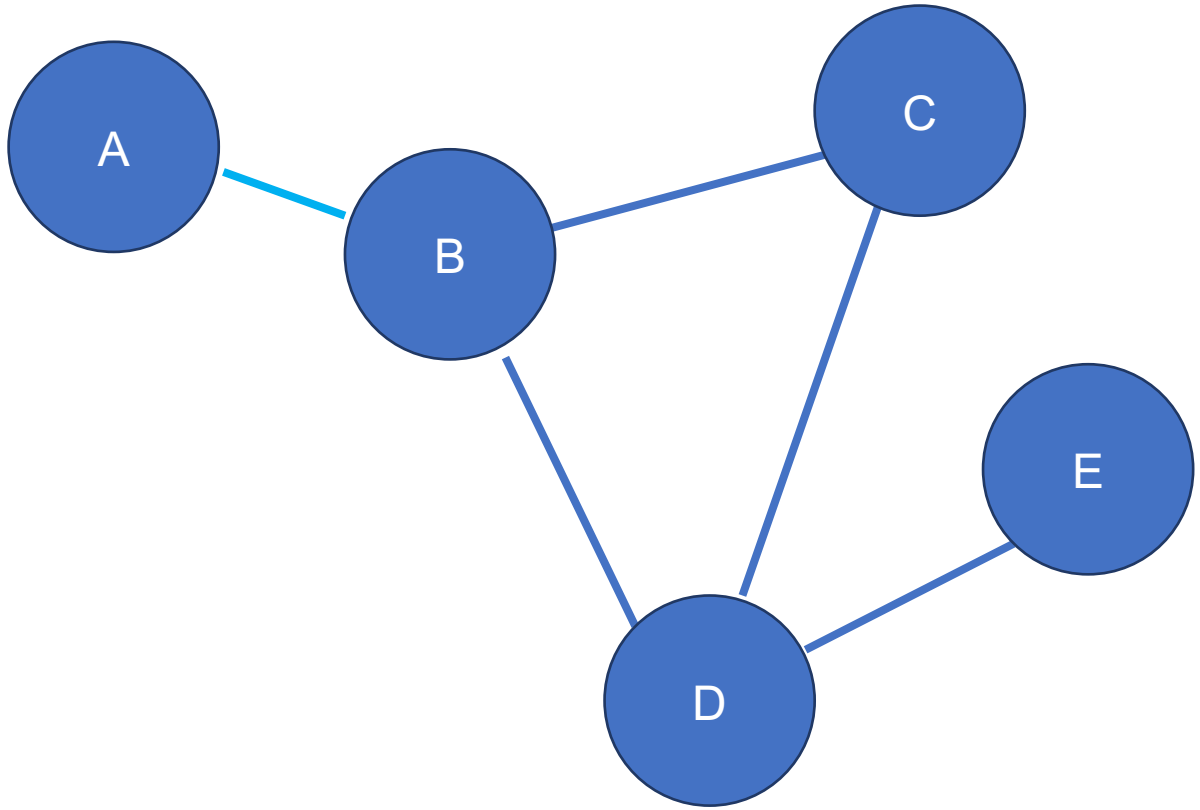
	0 (A)	1 (B)	2 (C)	3 (D)	4 (E)
0 (A)	0	1	0	0	0
1 (B)	0	0	0	1	0
2 (C)	0	1	0	0	0
3 (D)	0	0	1	0	1
4 (E)	0	0	0	0	0

Graph representation

Adjacency List

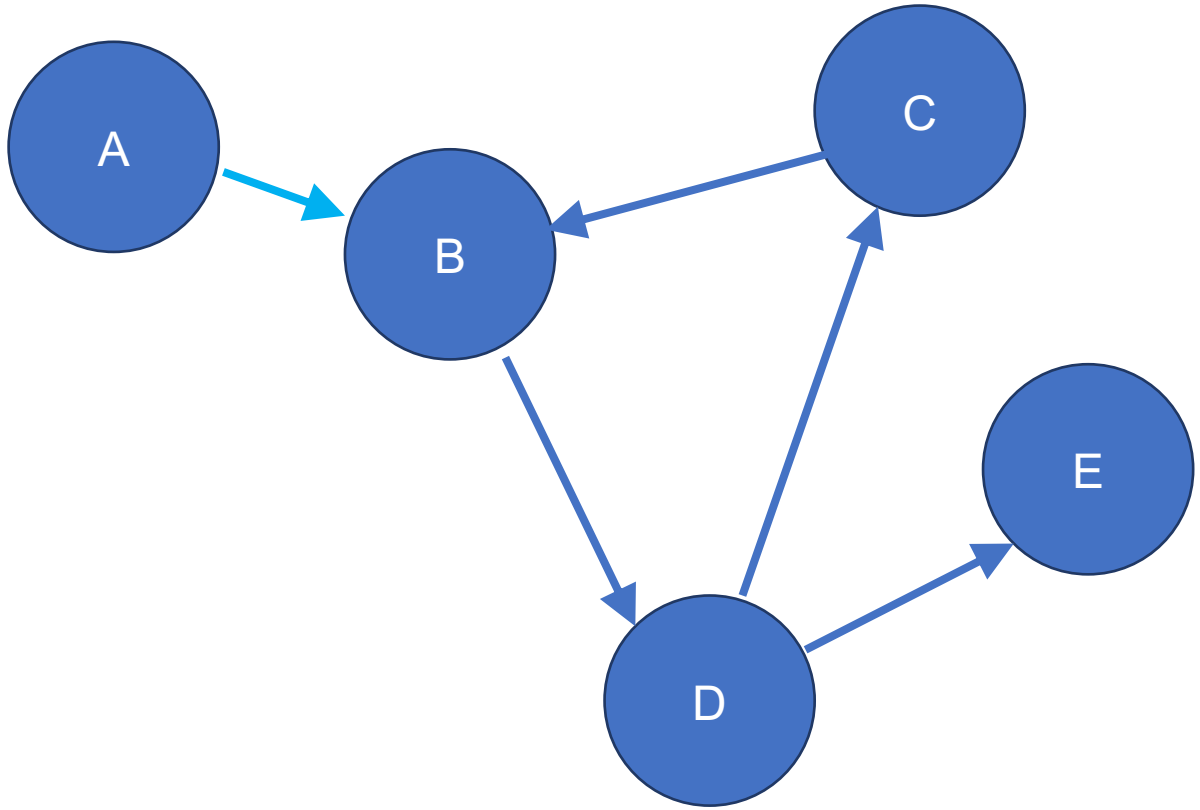
- For each node/vertex, keep a list of all adjacent vertices
- Can be implemented using various data collections (ArrayList, LinkedList, HashSet)
- Appropriate for sparse graphs

Undirected graph – adjacency list



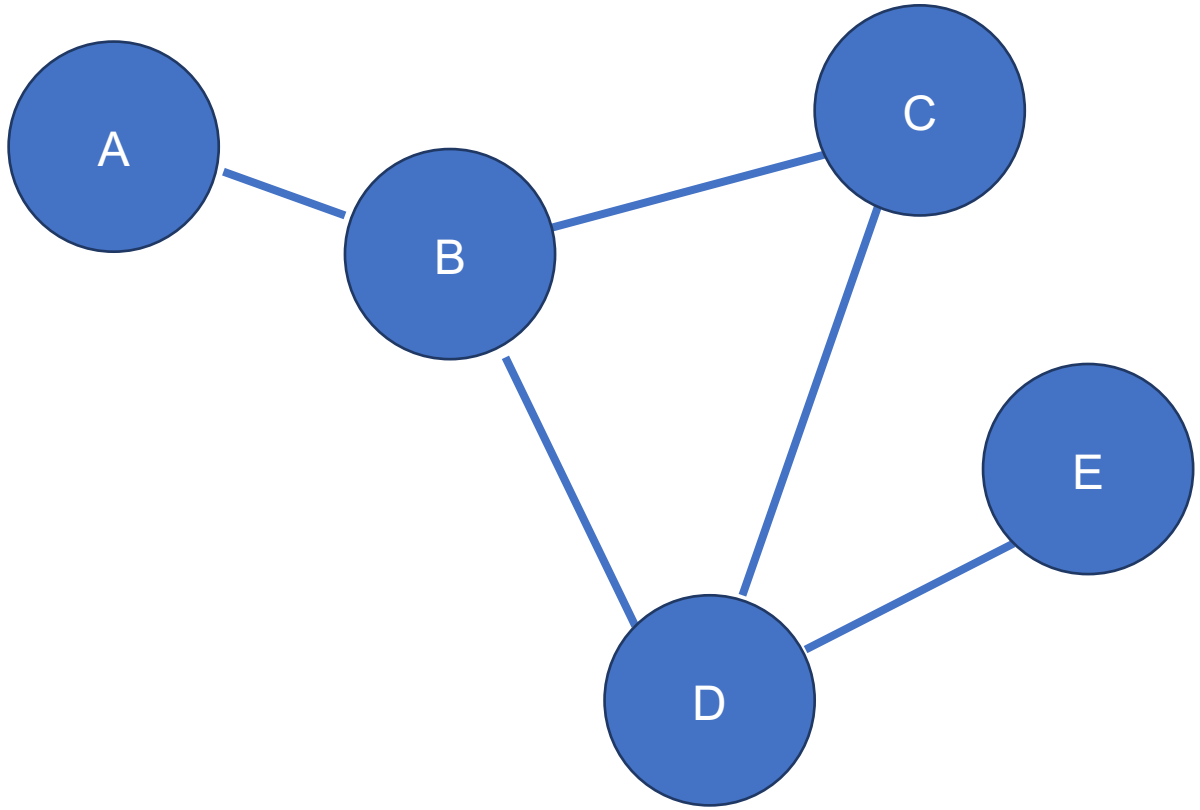
A	**B**
B	**A**, C, D
C	B, D
D	B, C, E
E	D

Directed graph – adjacency list



A	B
B	D
C	B
D	C, E
E	

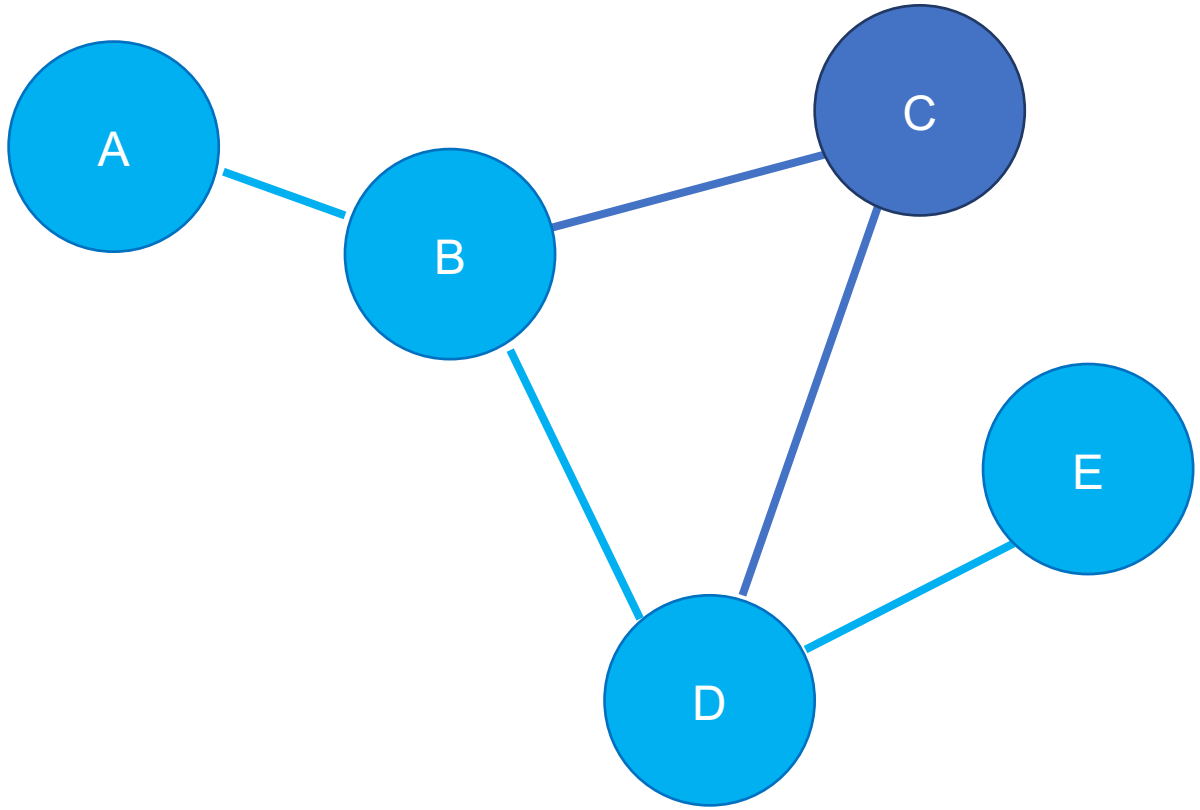
Paths in graphs



A set of edges connecting two vertices

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

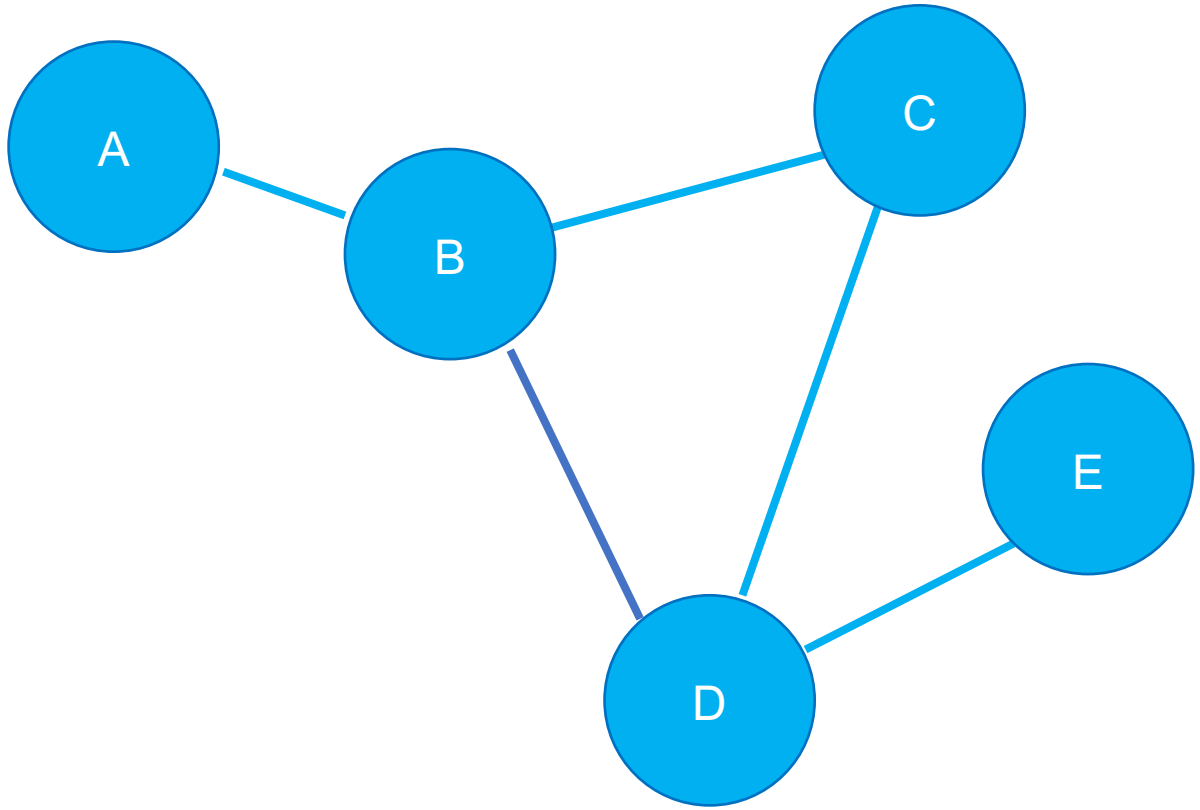
Paths in graphs – undirected example



Path from A to E (or E to A)

- Option 1: length 3

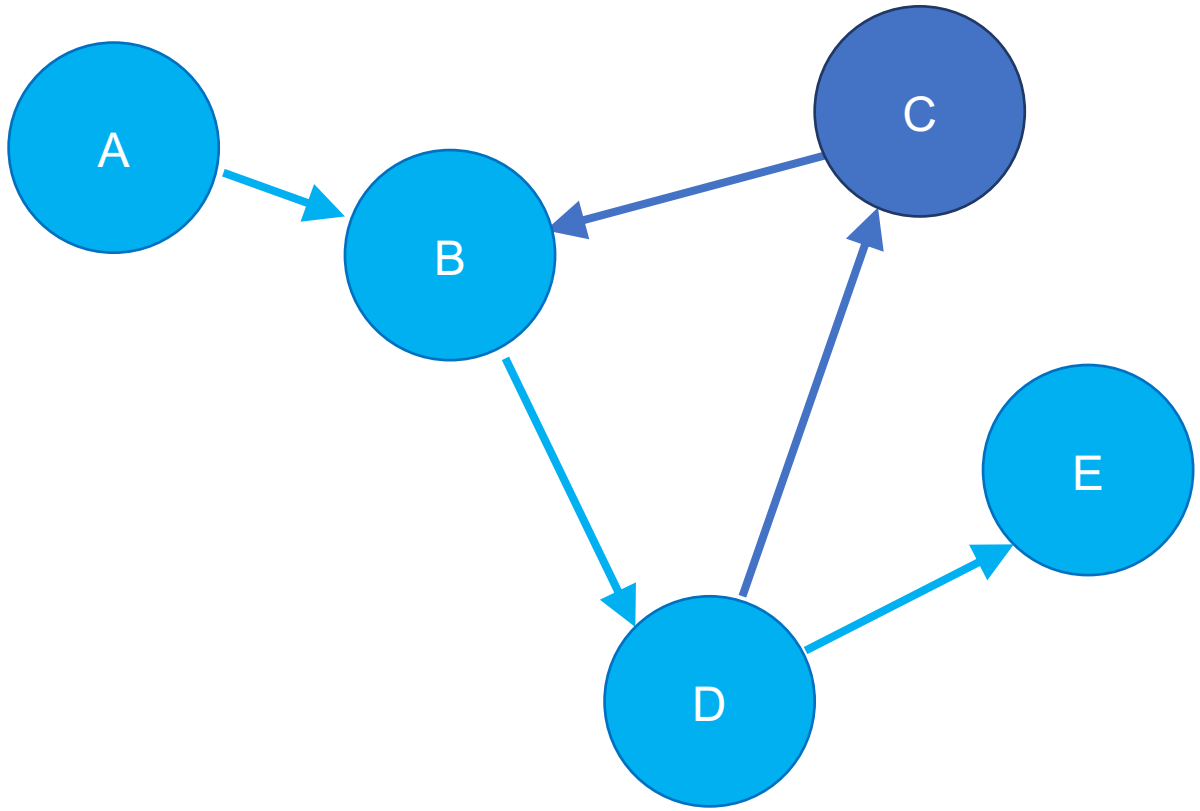
Paths in graphs – undirected example



Path from A to E (or E to A)

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

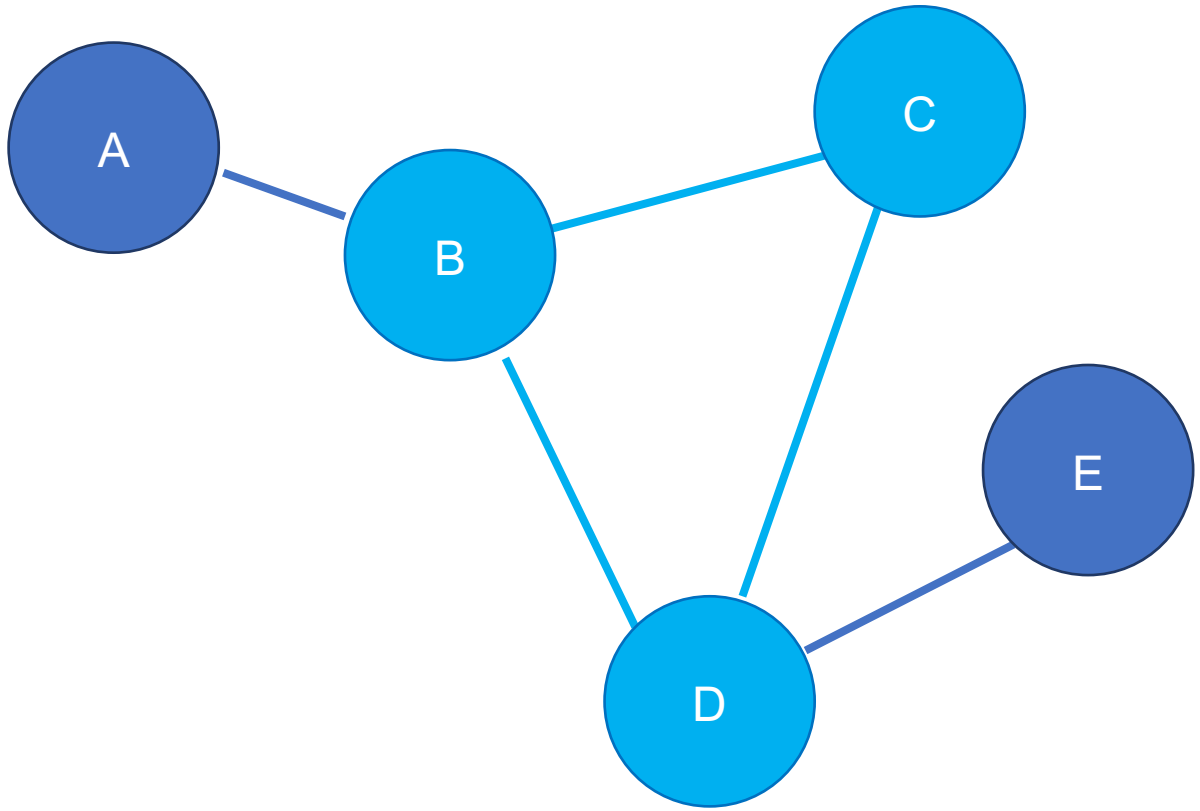
Paths in graphs – directed example



Path from A to E

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

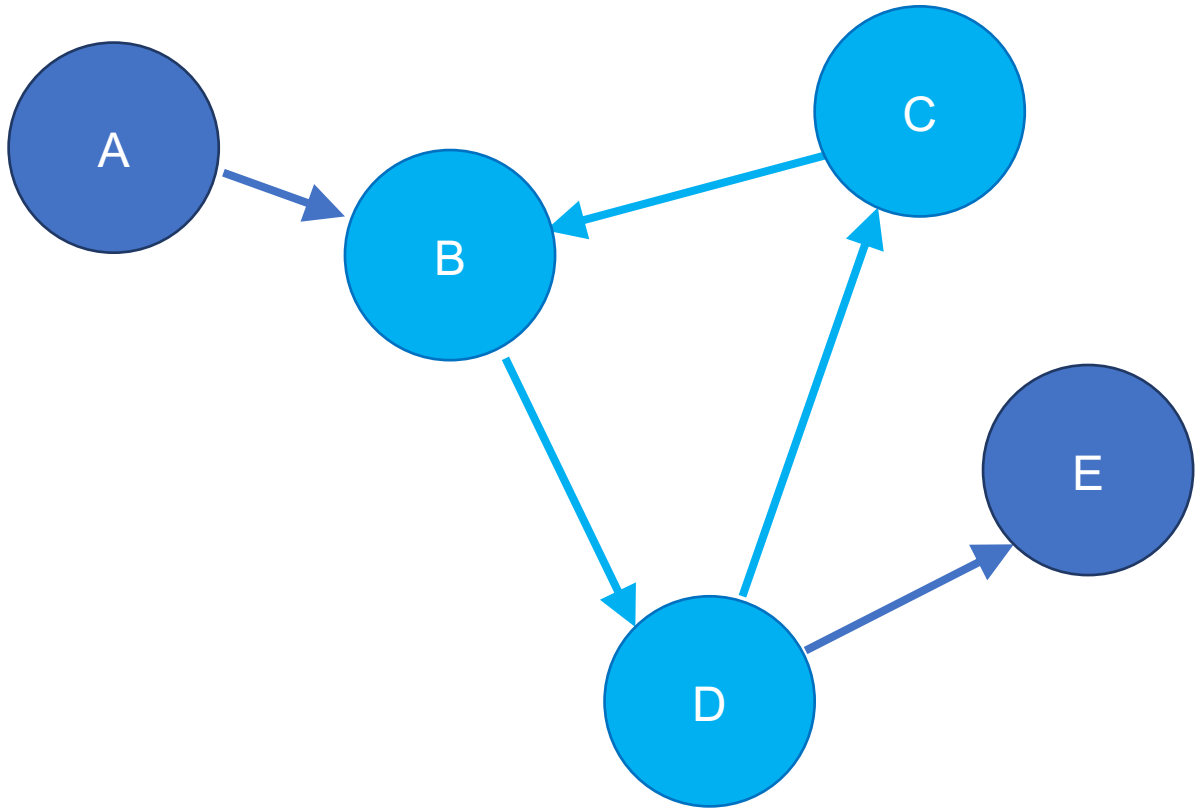
Cycles in graphs



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

- B – C – D - B

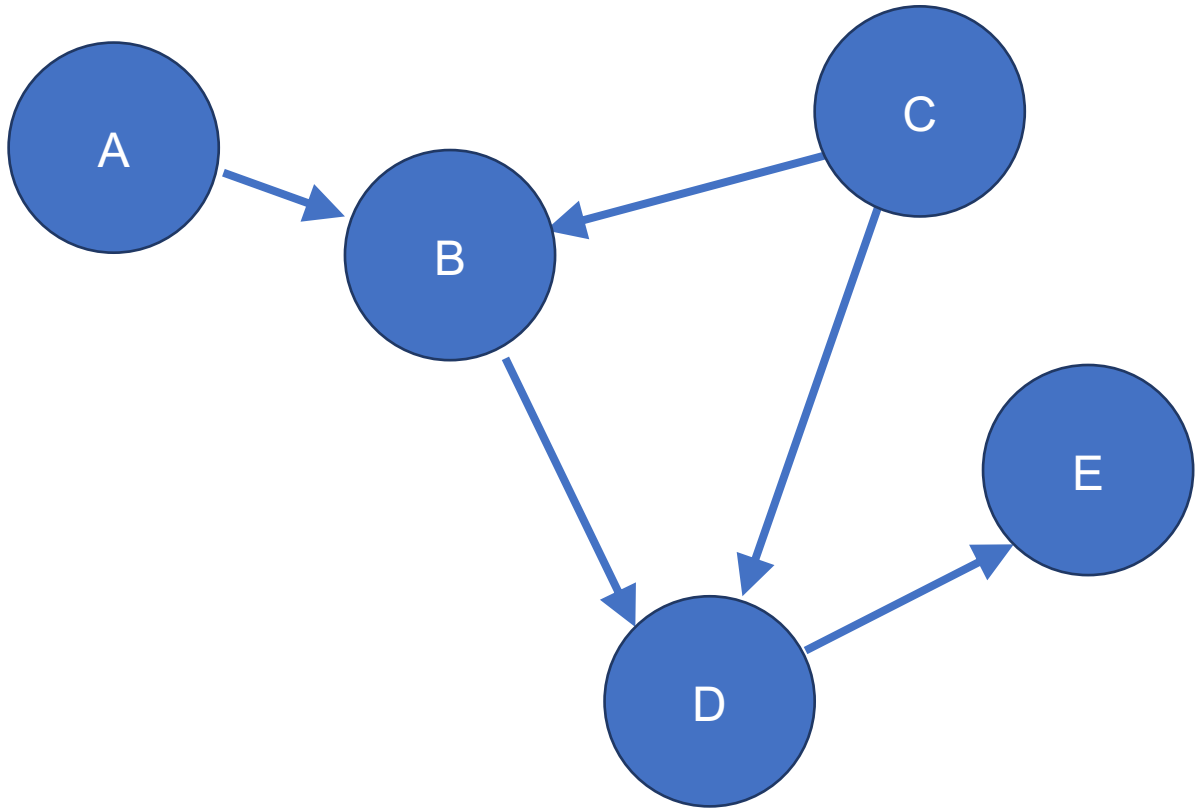
Cycles in graphs



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)

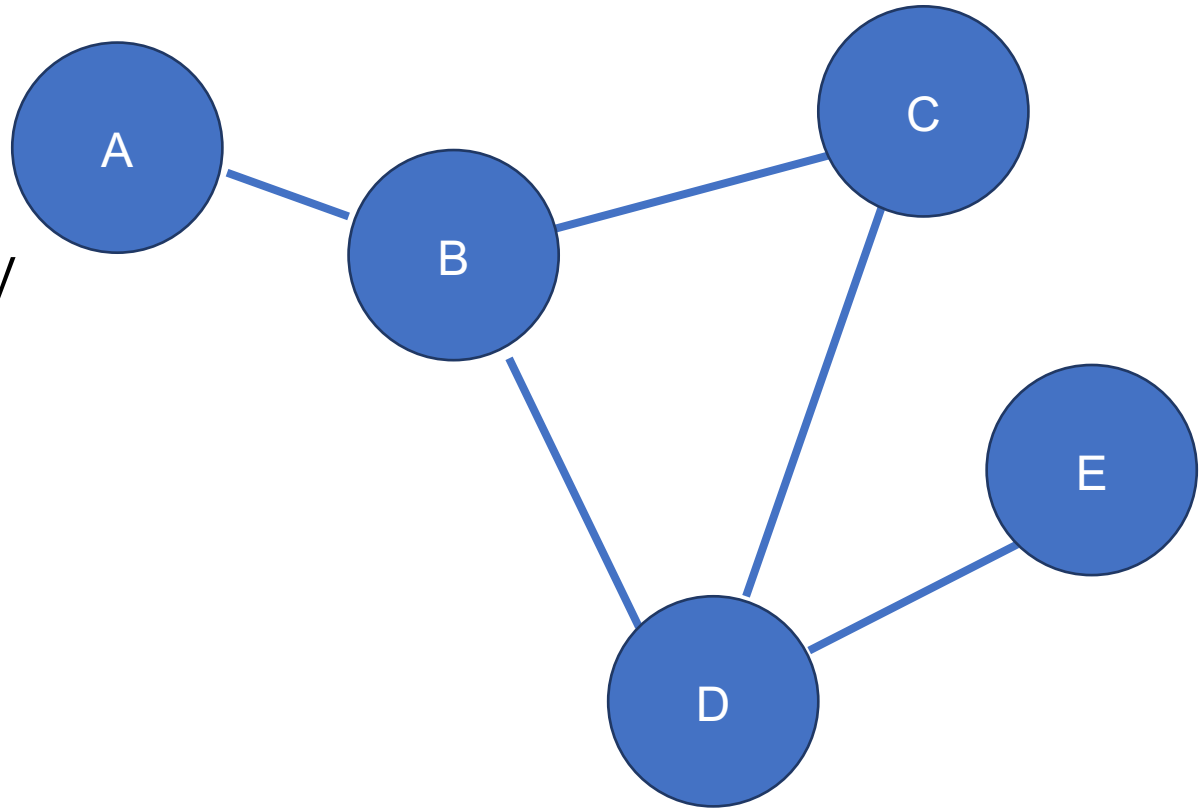
Other less common terms

- **Walk** - a path that can repeat either vertices or edges
- **Trail** – a path that can repeat vertices, but not edges
- **Circuit** – a trail that starts and ends at the same vertex

Connected and complete graphs

Connected graph

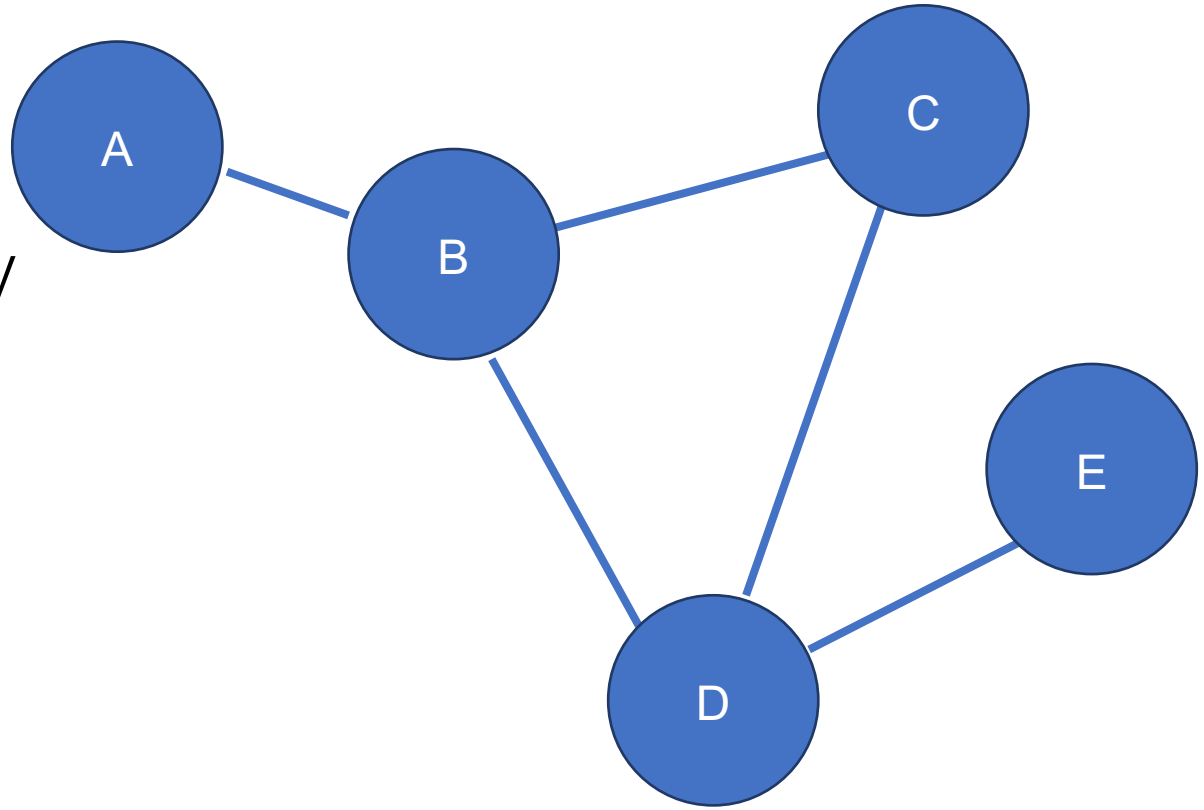
- undirected
- path from every vertex to every other vertex
- *Is this graph connected?*



Connected and complete graphs

Connected graph

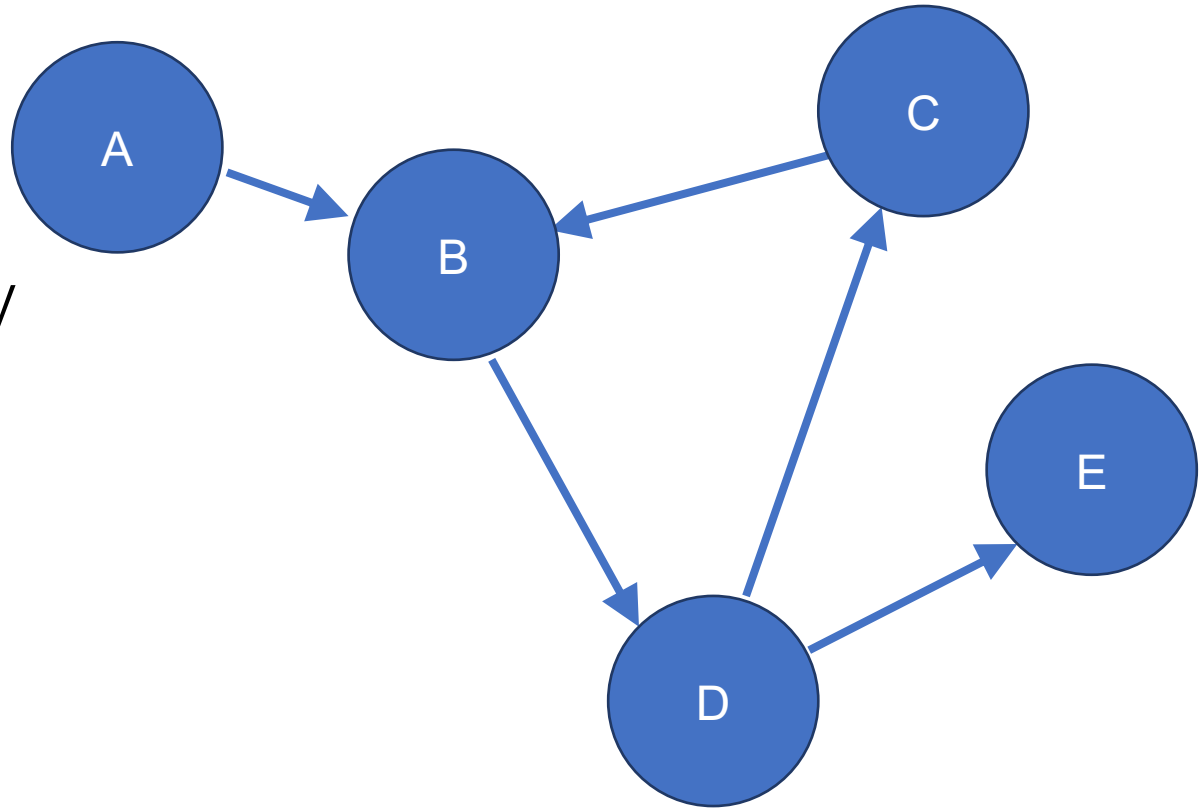
- undirected
- path from every vertex to every other vertex
- *Is this graph connected?*
  - *Yes*



Connected and complete graphs

Strongly connected graph

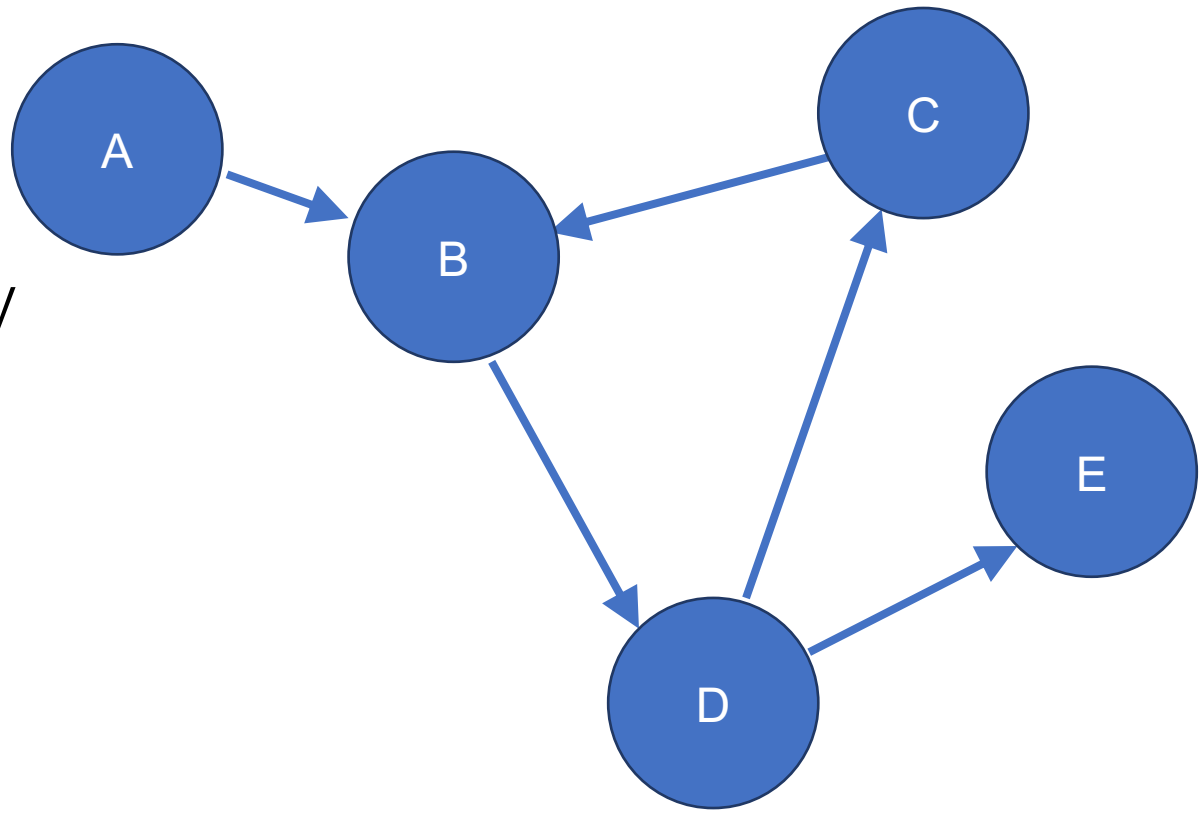
- directed
- path from every vertex to every other vertex
- *Is this graph strongly connected?*



Connected and complete graphs

Strongly connected graph

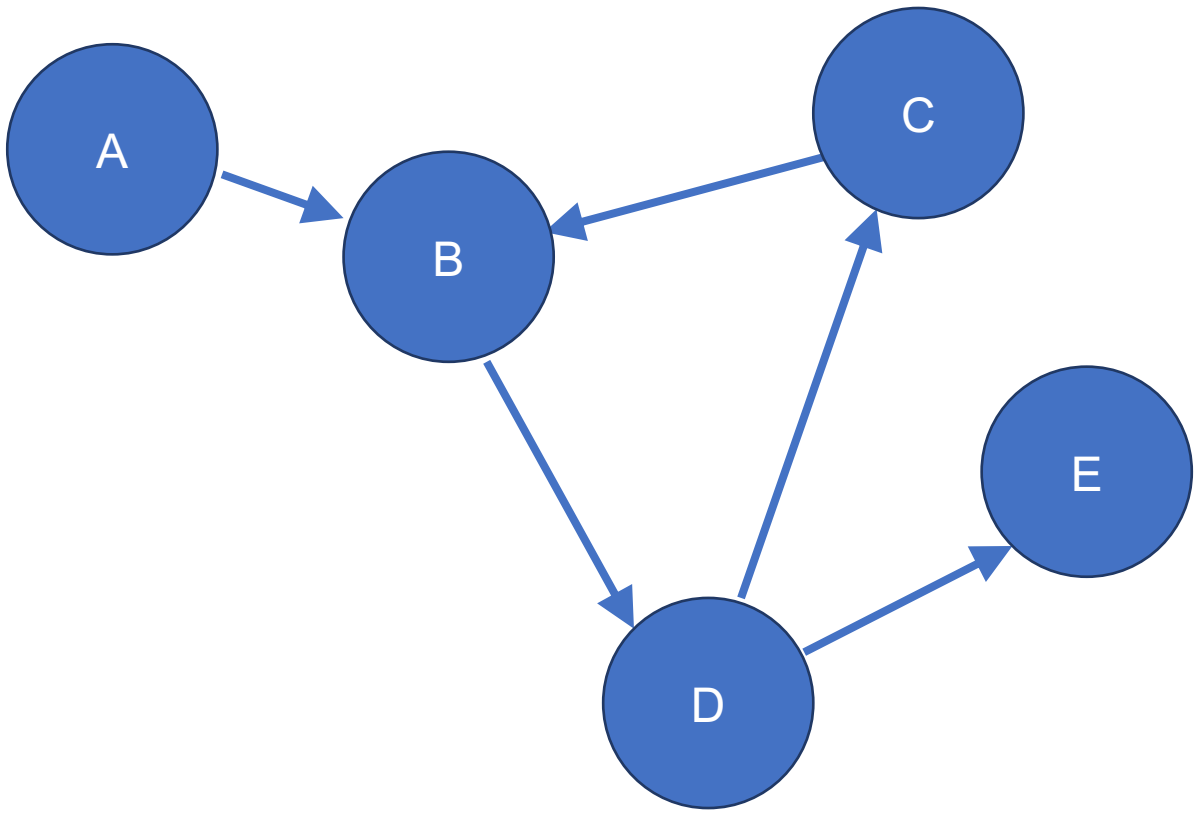
- directed
- path from every vertex to every other vertex
- *Is this graph strongly connected?*
  - *No. Many paths missing:*
    - *Any vertex to A*
    - *E to any vertex*



Connected and complete graphs

Weakly connected graph

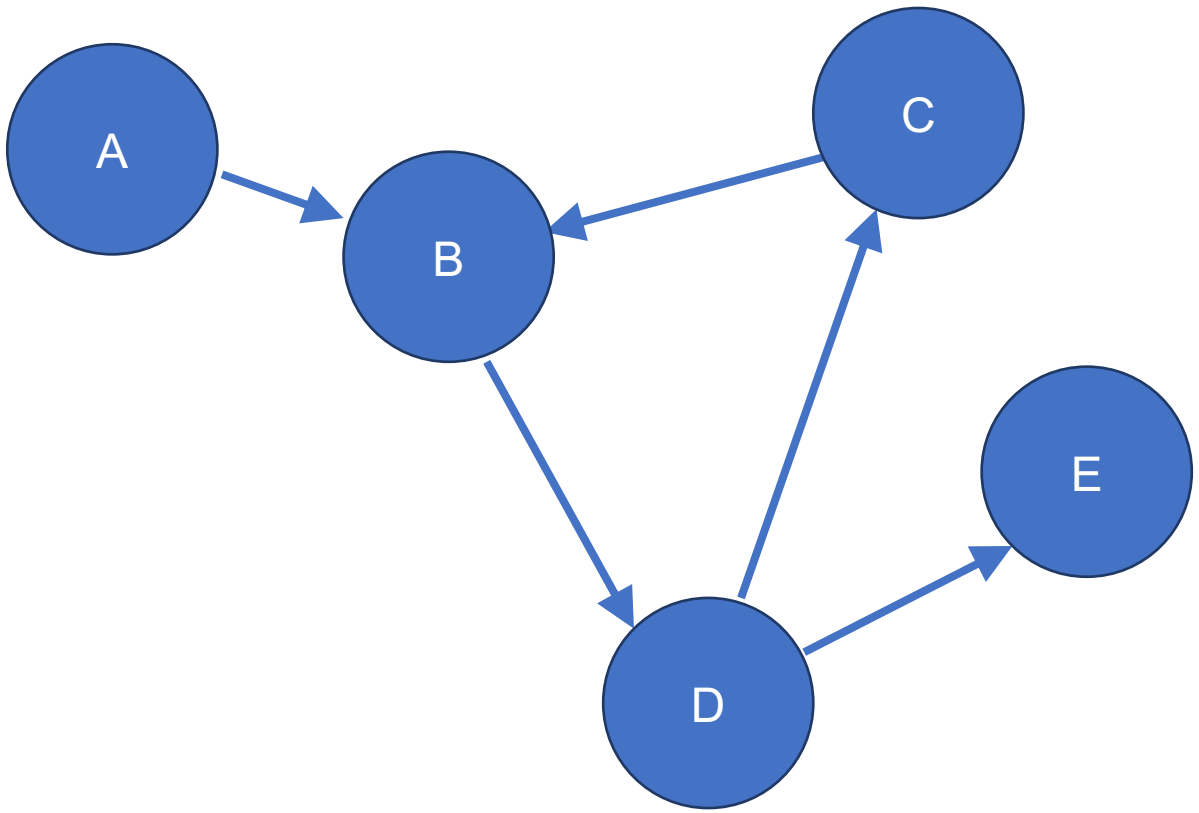
- directed
- not strongly connected but its underlying undirected graph (i.e. without the arrows) is connected
- *Is this graph weakly connected?*



Connected and complete graphs

Weakly connected graph

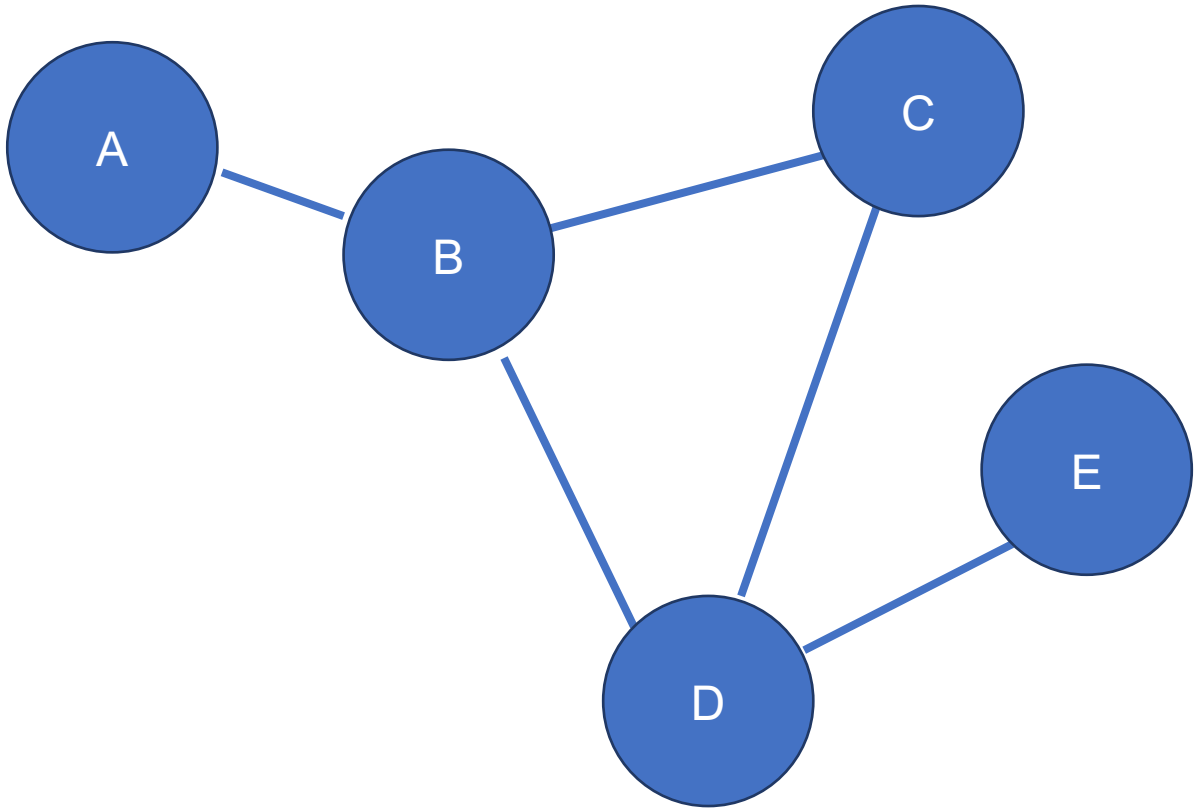
- directed
- not strongly connected but its underlying undirected graph (i.e. without the arrows) is connected
- *Is this graph weakly connected?*
  - *Yes*



Connected and complete graphs

Complete graph

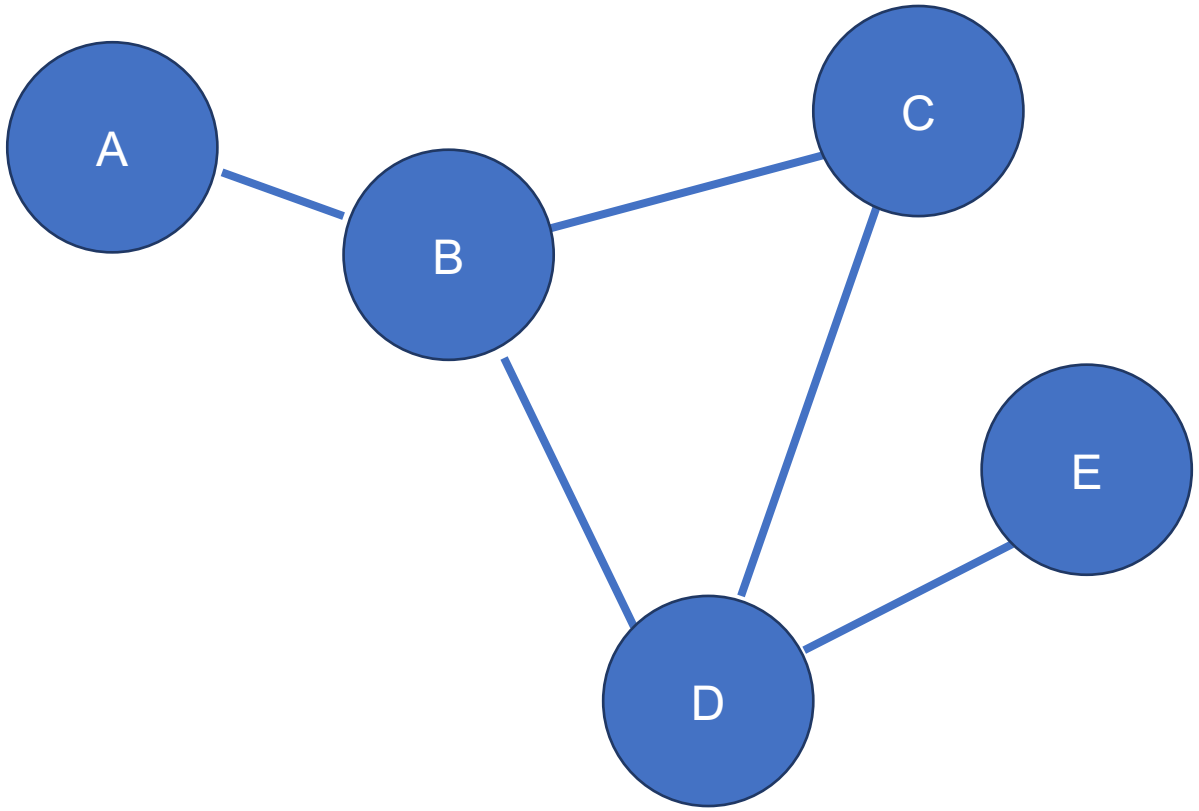
- undirected or directed
- an edge between every pair of vertices
- *Is this graph complete?*



Connected and complete graphs

Complete graph

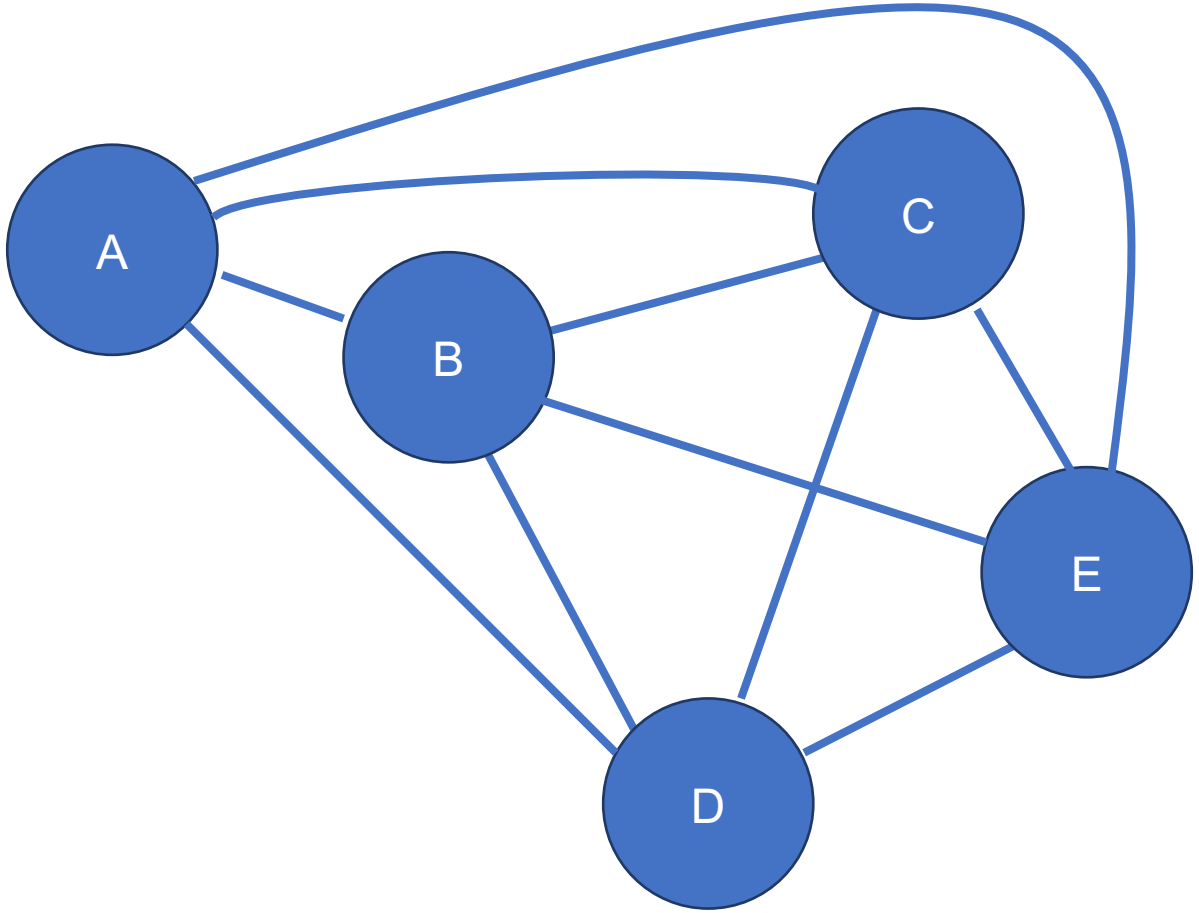
- undirected or directed
- an edge between every pair of vertices
- *Is this graph complete?*
  - *No. No direct connections between multiple pairs*



Connected and complete graphs

Complete graph

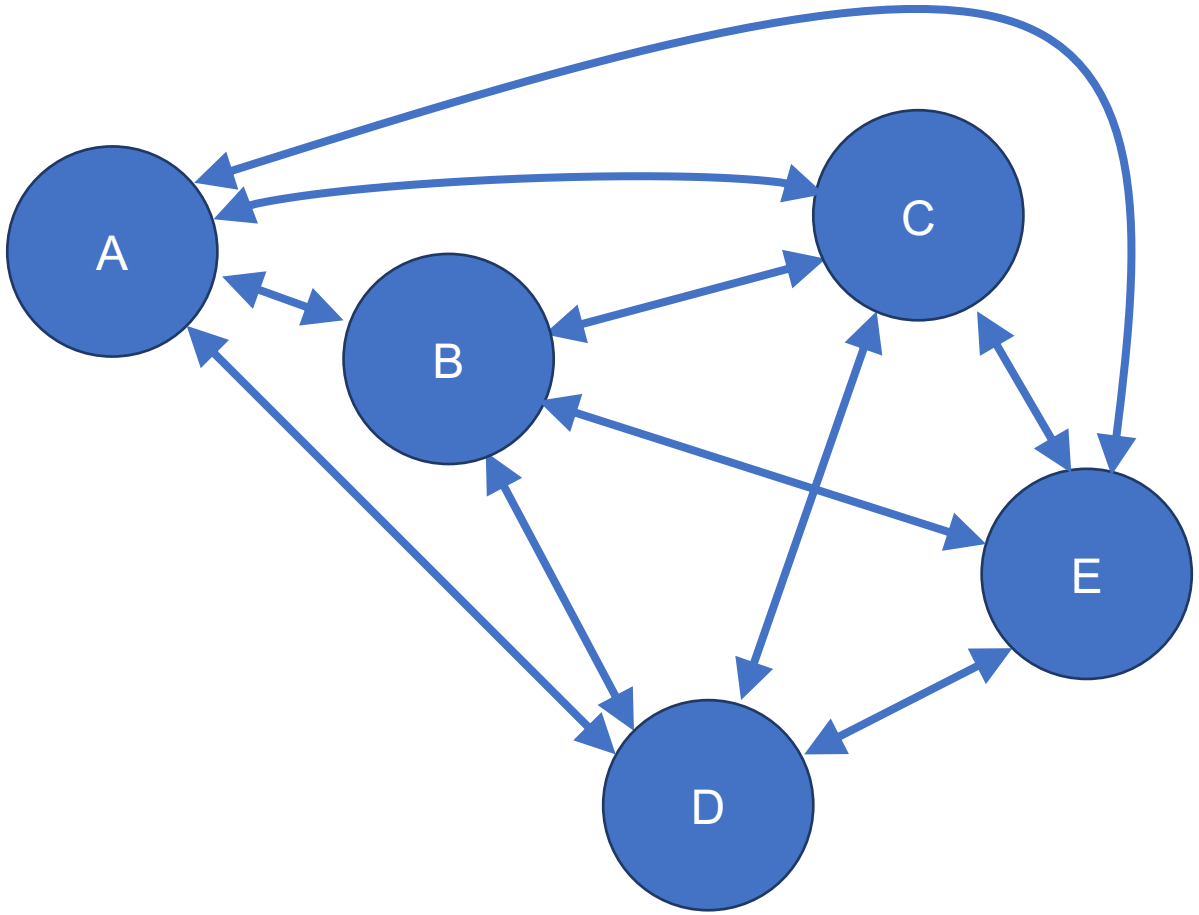
- …now it's complete



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);`

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?

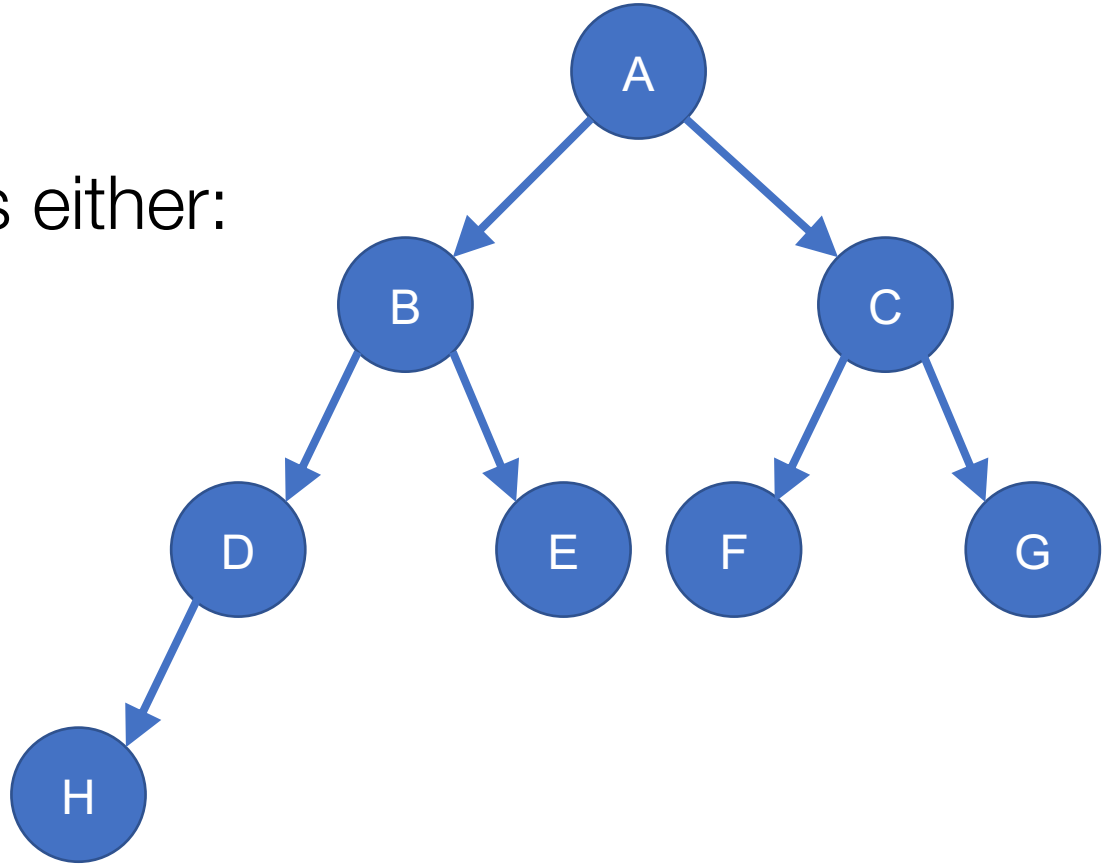
A **directed**, **acyclic** structure of linked nodes

- 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:

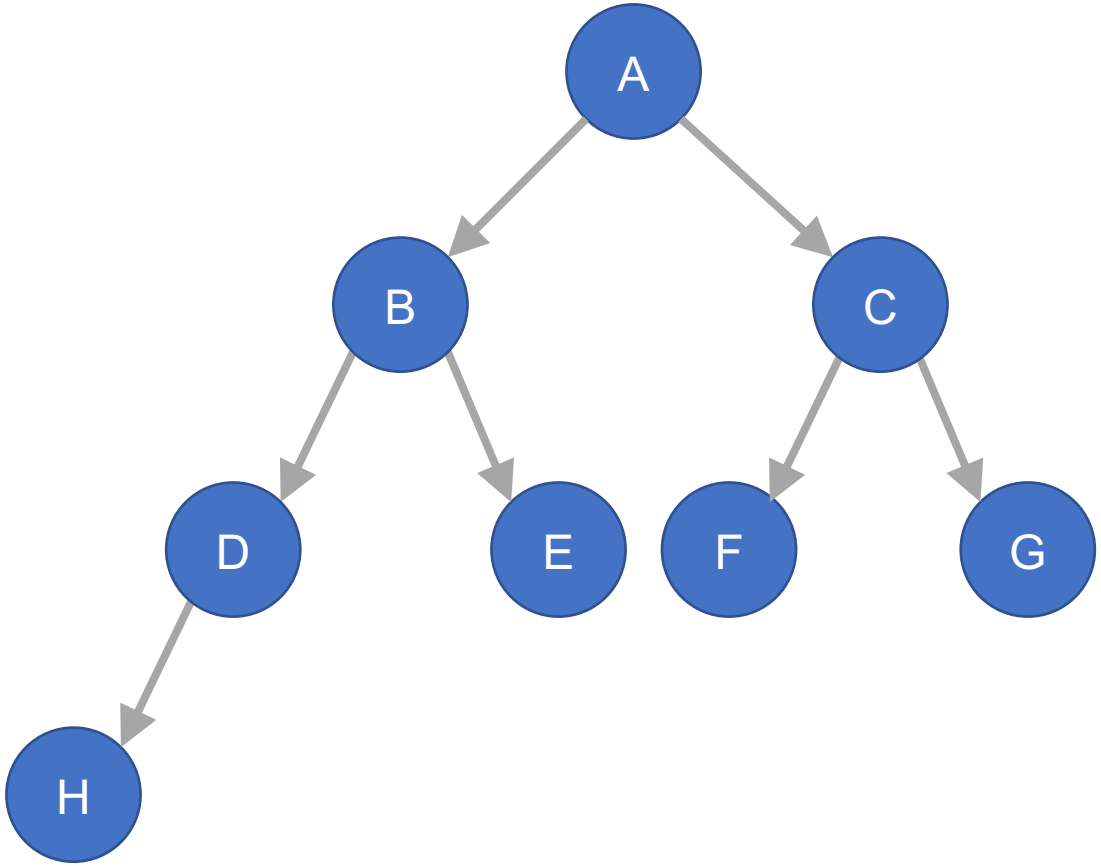
- Empty, or
- A root node that contains:
  - Data
  - A left subtree
  - A right subtree



Tree terminology

Nodes

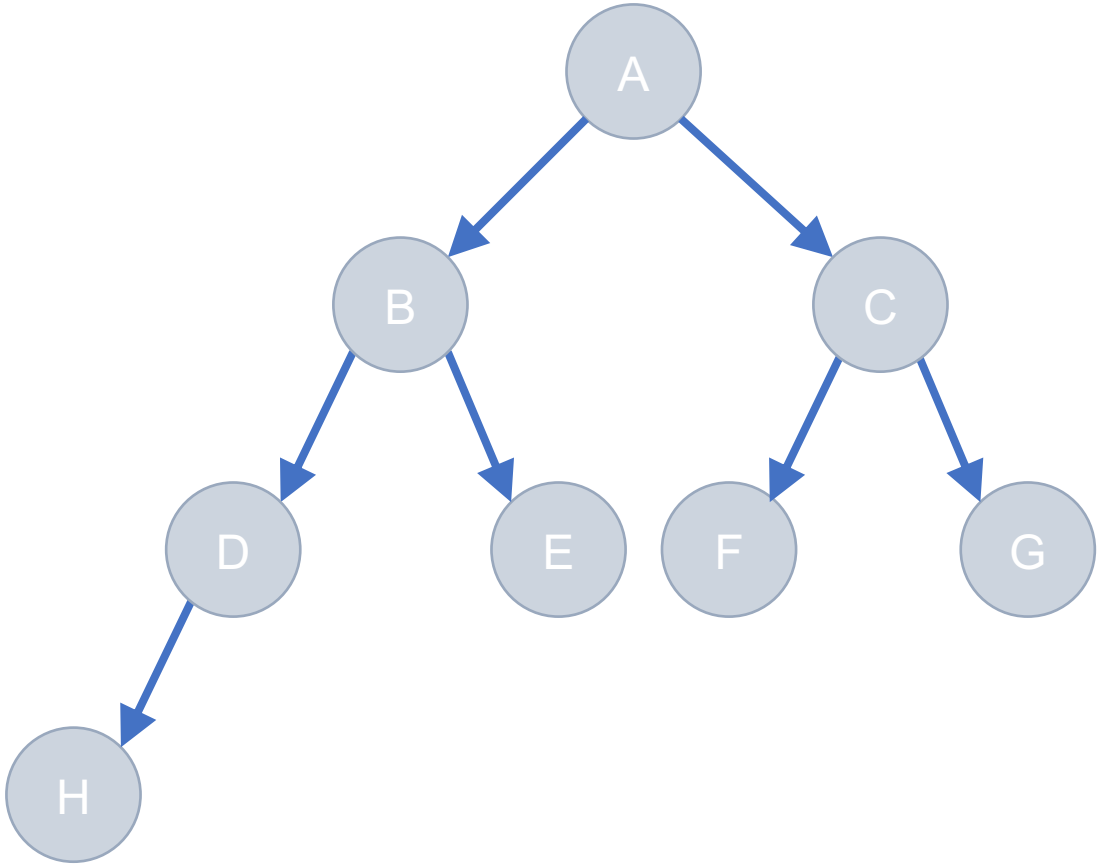
- Contain data
- Links to left/right children



Tree terminology

Edges

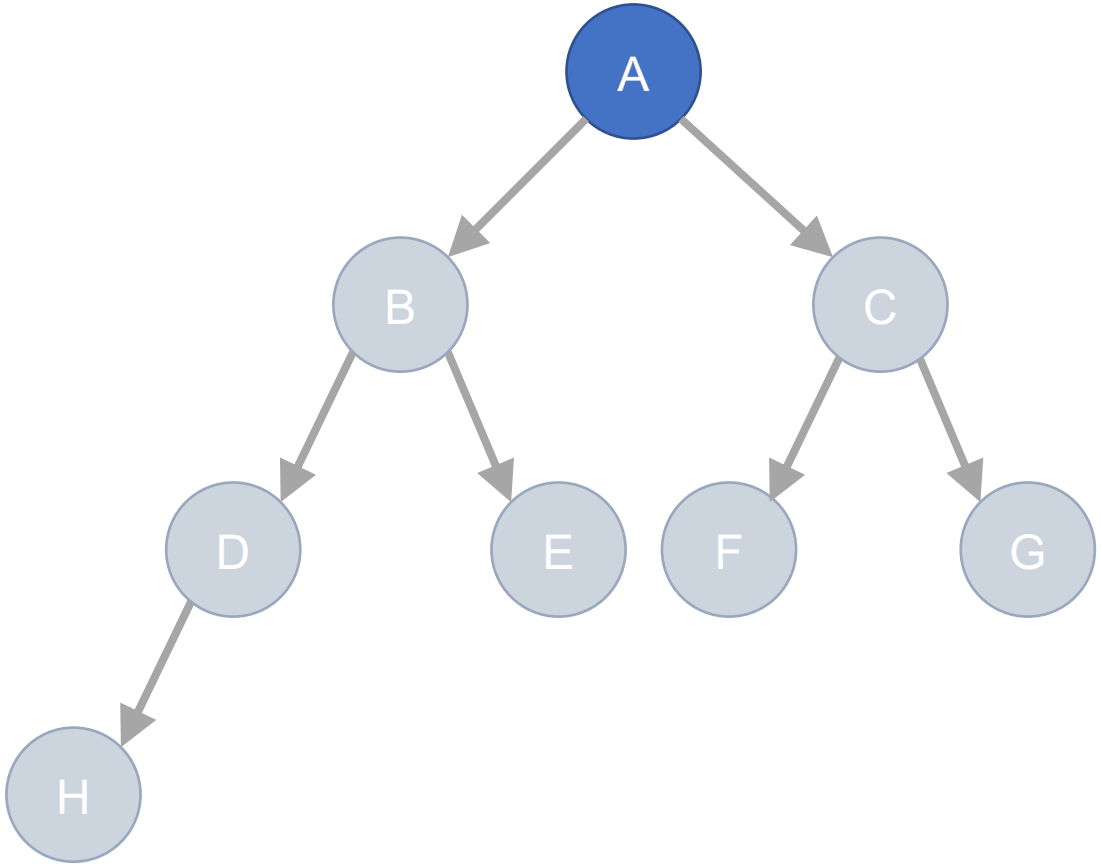
- Connect nodes



Tree terminology

Root

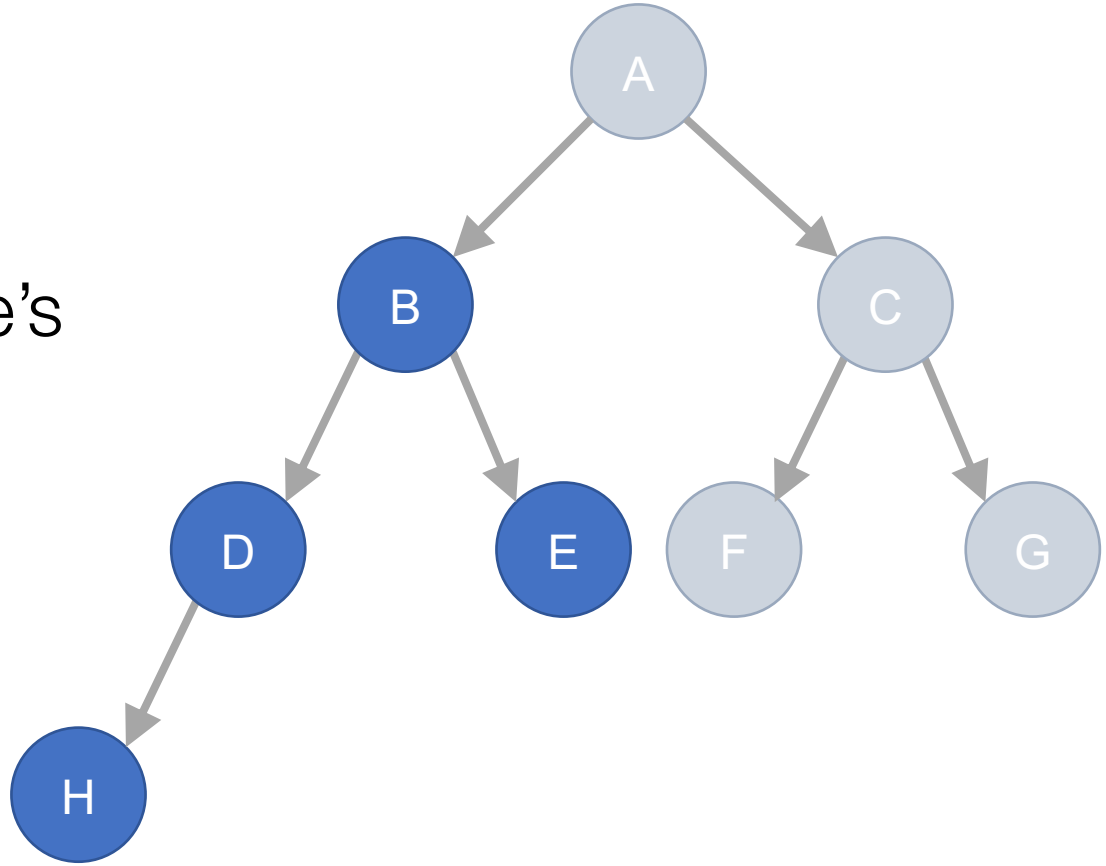
- The start of the tree



Tree terminology

Subtree

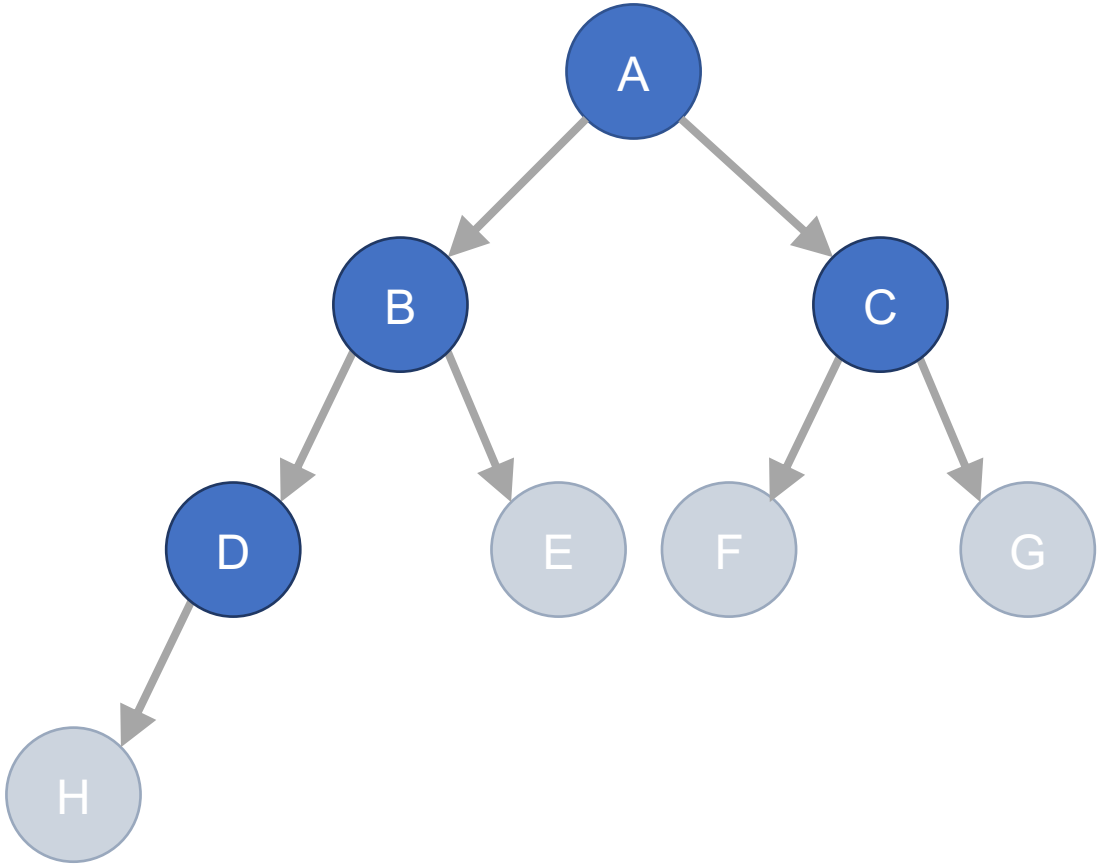
- The tree starting from one of a node's children
  - E.g. A's left subtree



Tree terminology

Parent/ancestor

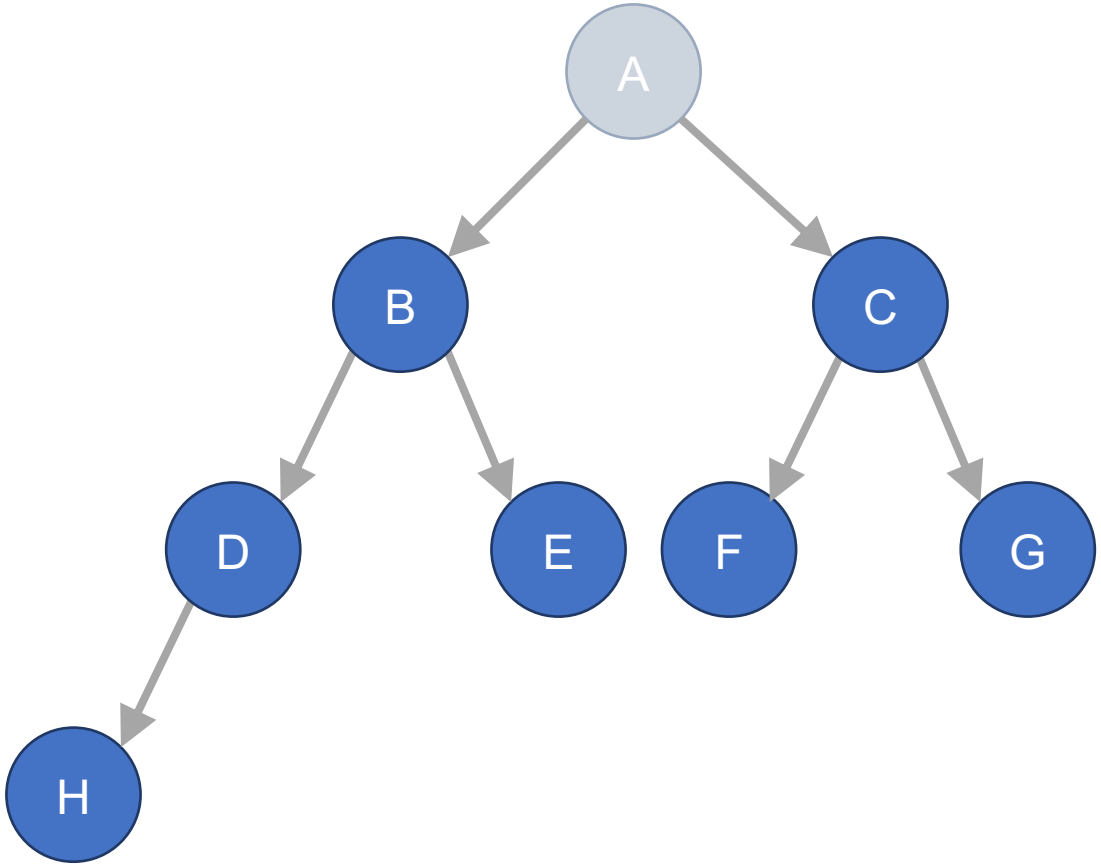
- Any node with at least one child
  - E.g. B is the parent of D and E



Tree terminology

Child

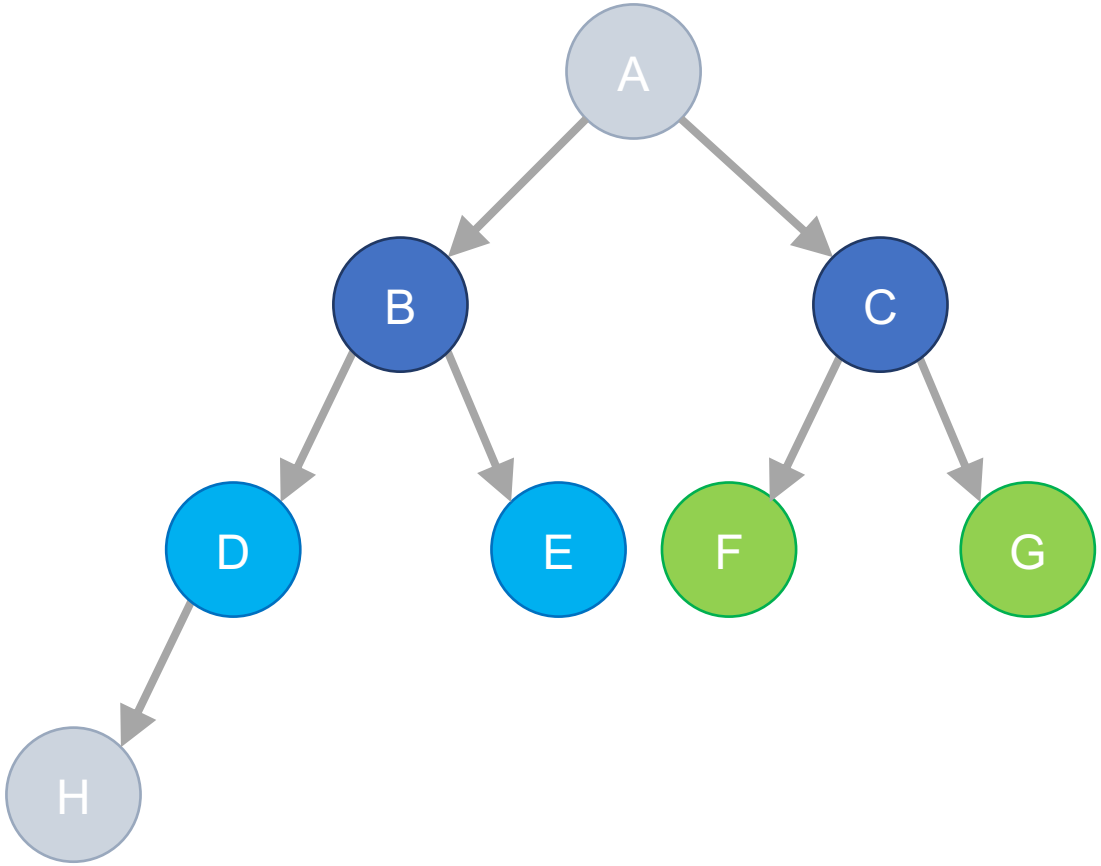
- Any node with a parent
  - E.g. B is the child of A



Tree terminology

Siblings

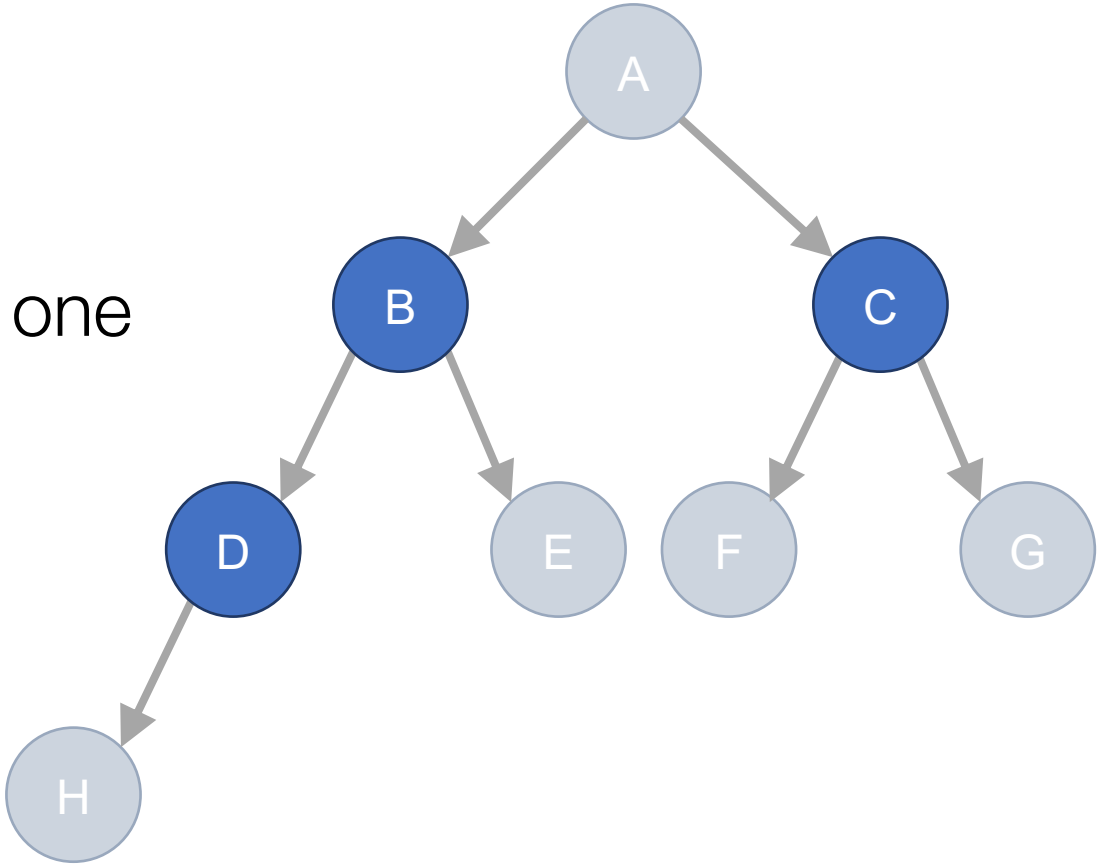
- Node with a common parent
  - B & C are siblings
  - D & E are siblings
  - F & G are siblings



Tree terminology

Branch / internal node

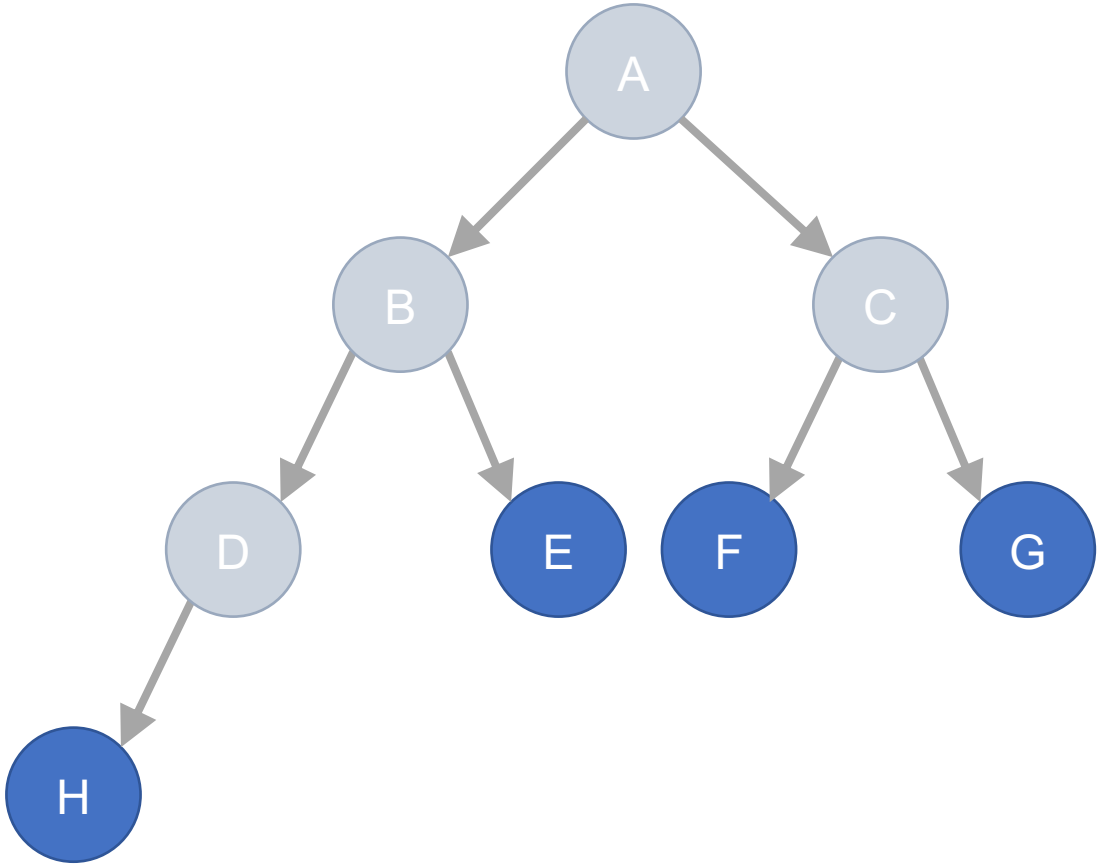
- Any node with a parent and at least one child



Tree terminology

Leaf / external node

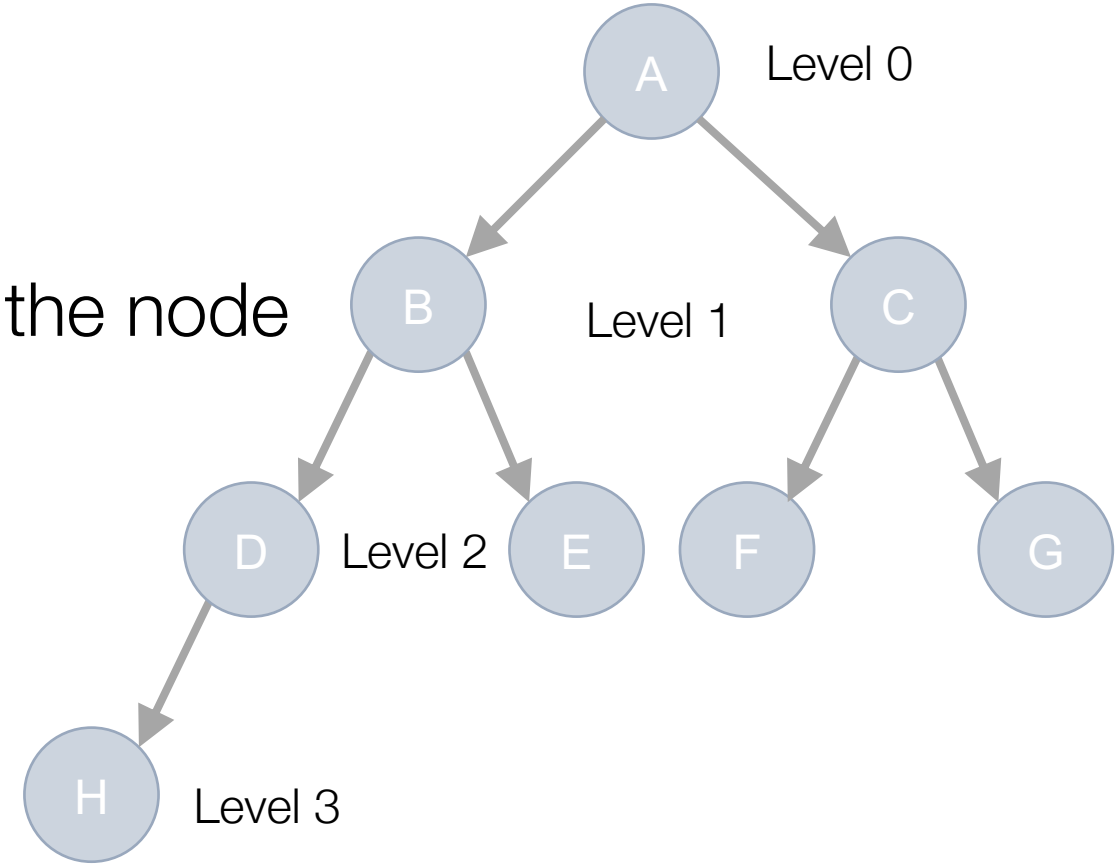
- A node with no children



Tree terminology

Level or **depth** of a node

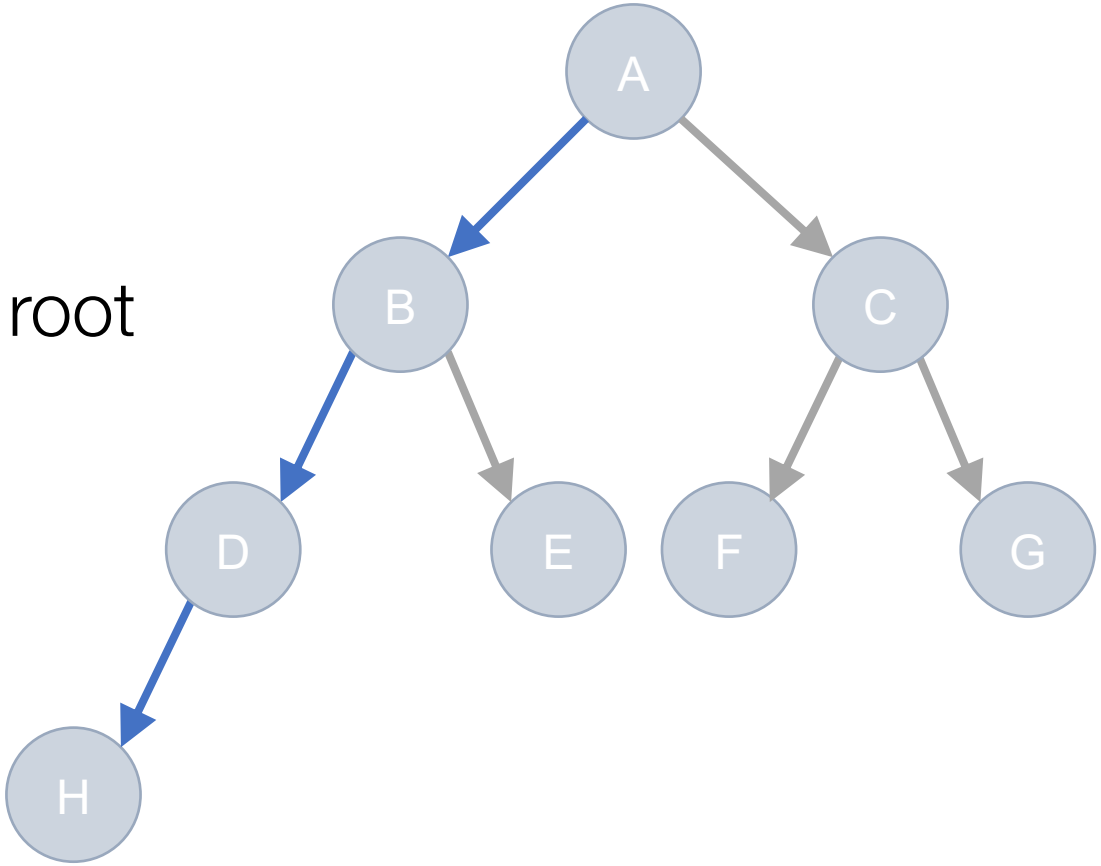
- Length of the path from the root to the node
- +1 if counting from 1 instead of 0



Tree terminology

Height of the tree

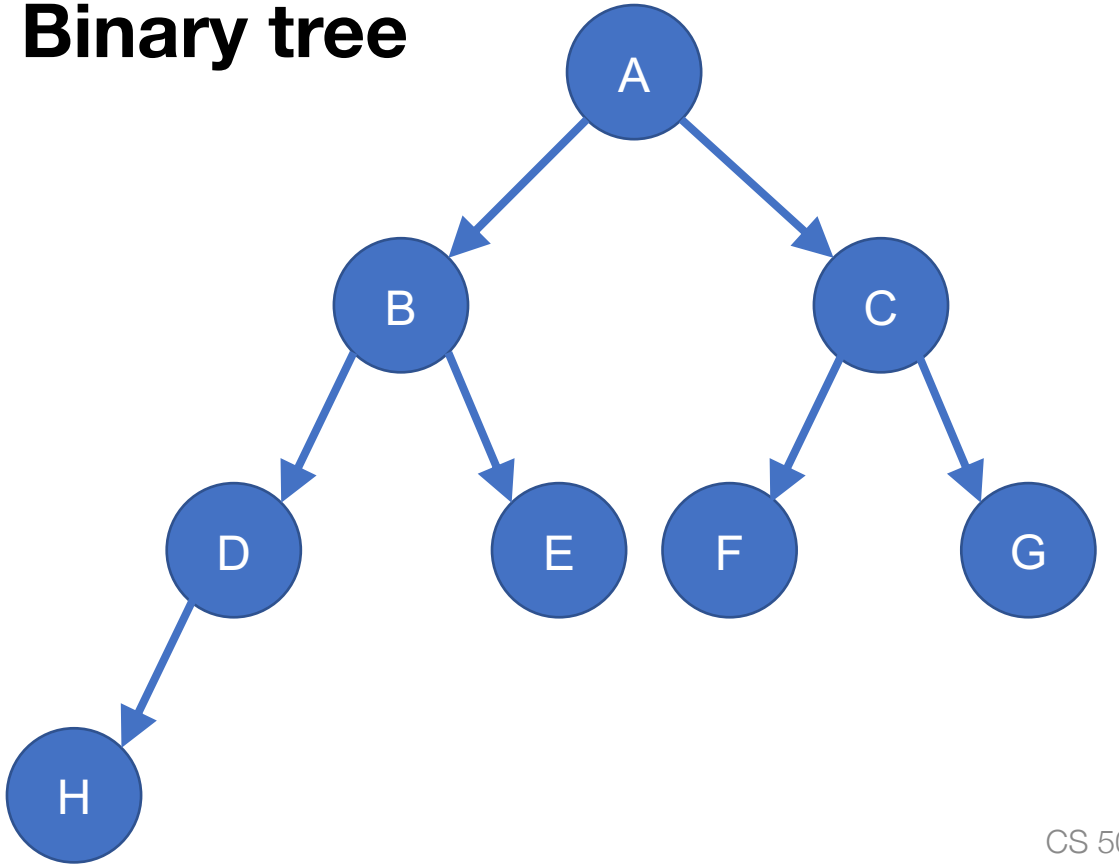
- Length of the longest path from the root to a leaf



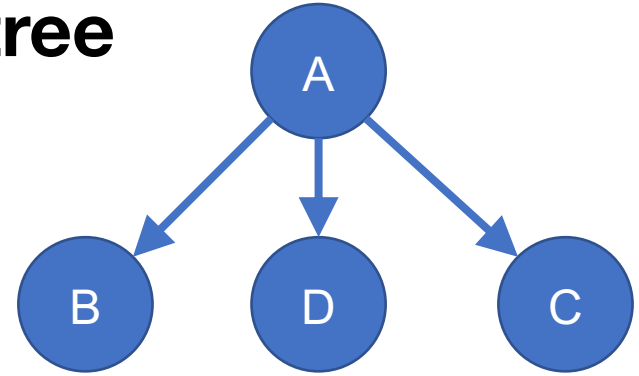
Binary Tree implementation

Binary tree – no node can have more than two children

Binary tree



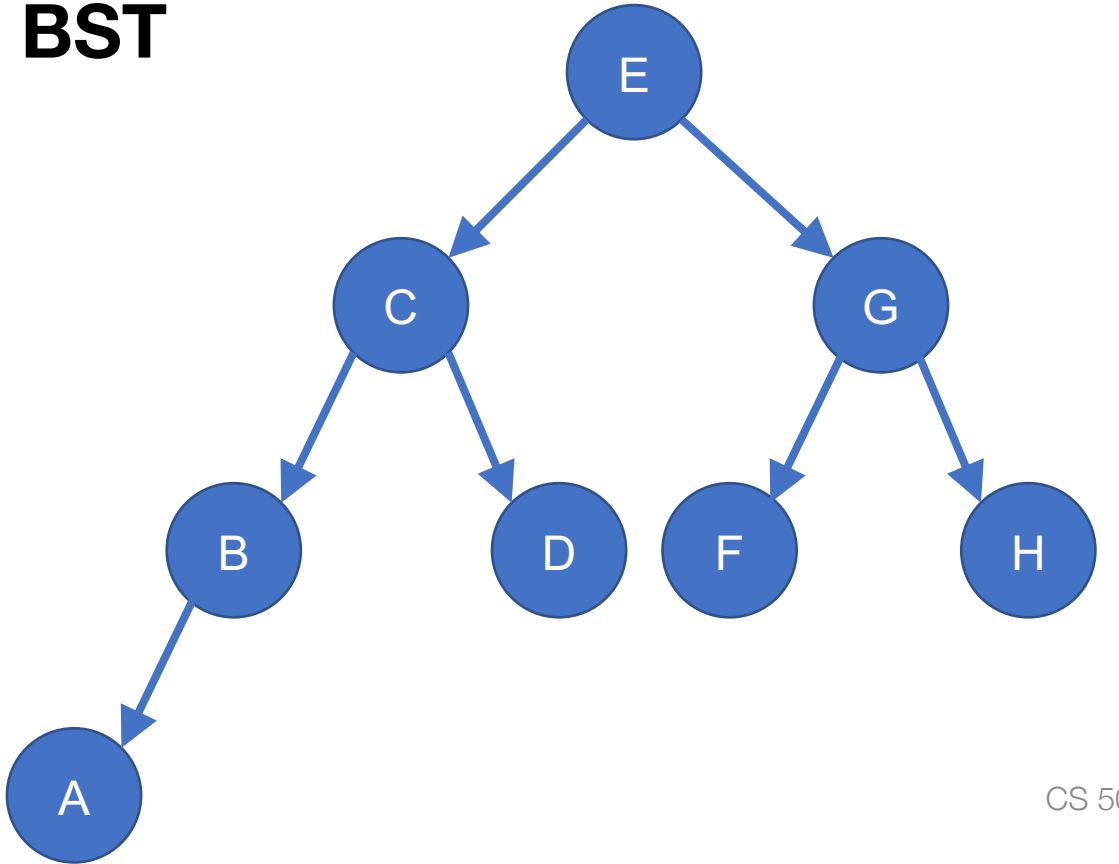
Not a binary tree



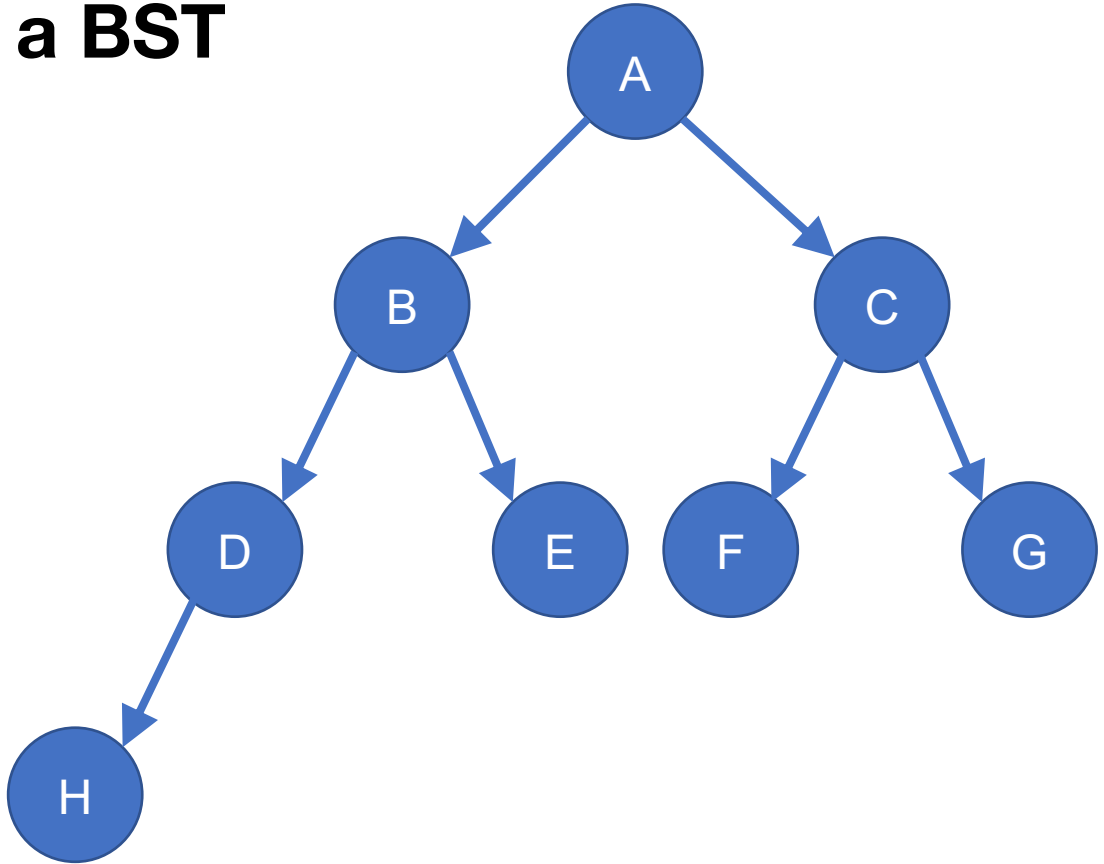
Implementing a binary search tree (BST)

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);
}
```

Exercise: BST with nested node class

- Create a private static nested BinaryNode class
- Keeps the nodes encapsulated in the tree