Section 5: HW6 and Midterm

Slides by Vinod Rathnam and Geoffrey Liu

(with material from Alex Mariakakis,
Kellen Donohue, David Mailhot, and Hal Perkins)

Breadth-First Search (BFS)

Often used for discovering connectivity

Calculates the shortest path *if and only if* all edges have same positive or no weight

Depth-first search (DFS) is commonly mentioned with BFS

- BFS looks "wide", DFS looks "deep"
- Can also be used for discovery, but not the shortest path

BFS Pseudocode

```
public boolean find(Node start, Node end) {
      put start node in a queue
      while (queue is not empty) {
            pop node N off queue
            if (N is goal)
                  return true;
            else {
                  for each node O that is child of N
                        push O onto queue
            }
      }
      return false;
}
```

Breadth-First Search

START:

Q: <A>

Pop: A, Q: <>

Q: <B, C>

Pop: B, Q: <C>

Q: <C>

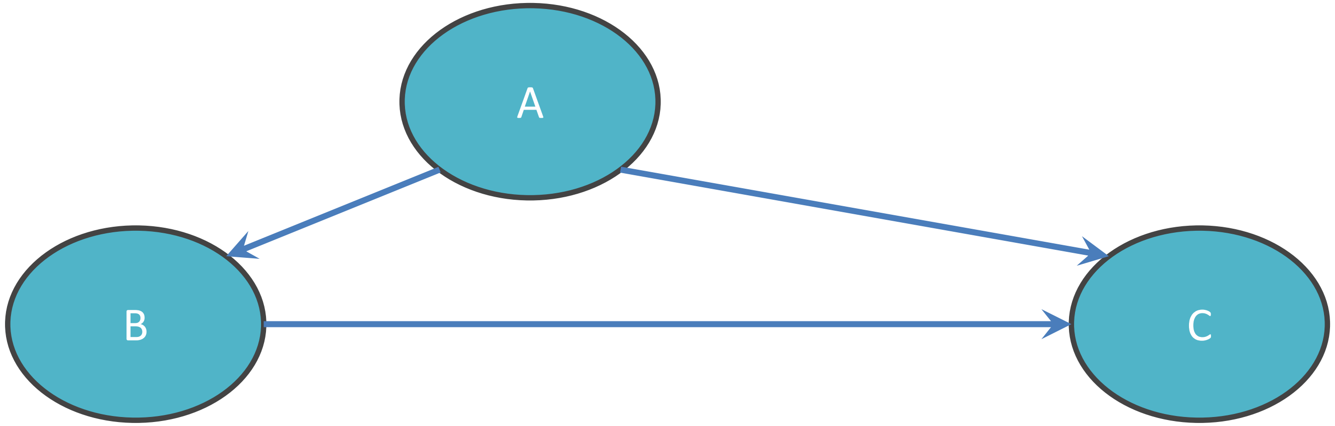
Pop: C, Q: <C>

Q: <>

DONE

Starting at A

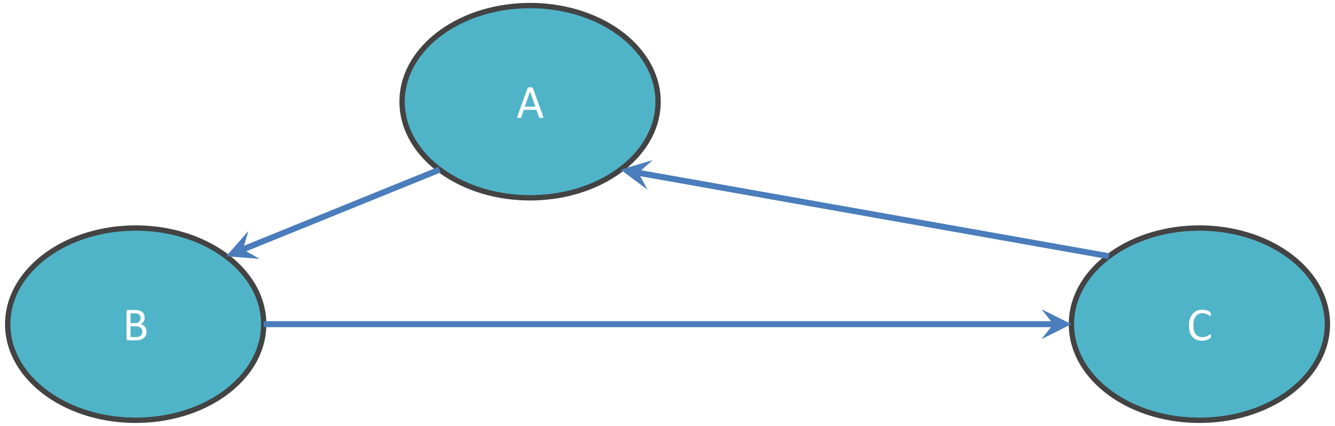
Goal: Fully explore



Breadth-First Search with Cycle

START:
Q: <A>
Pop: A, Q: <>
Q: <B>
Pop: B, Q: <>
Q: <C>
Pop: C, Q: <>
Q: <A>
NEVER DONE

Starting at A
Goal: Fully Explore



BFS Pseudocode

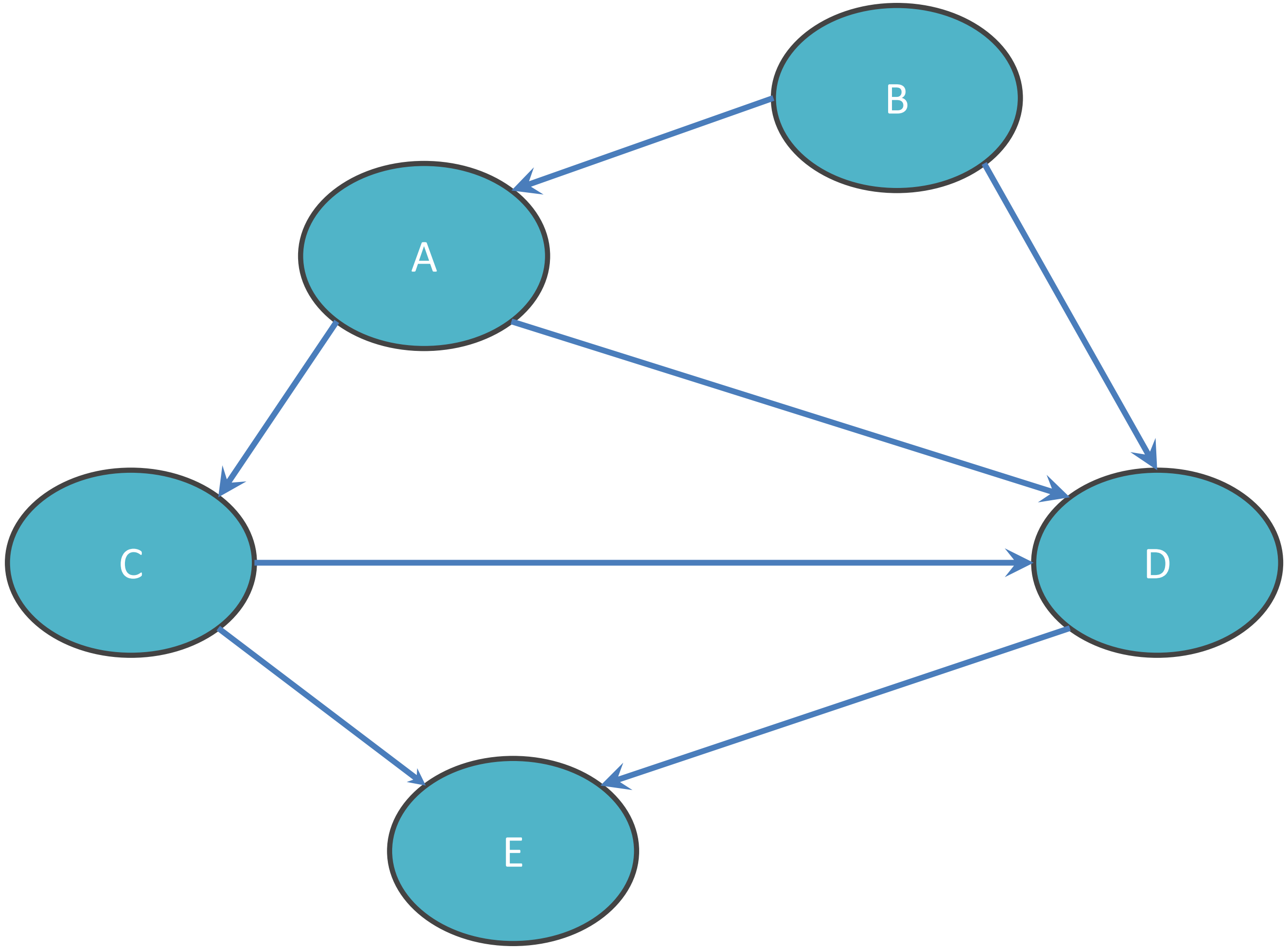
```
public boolean find(Node start, Node end) {
      put start node in a queue
      while (queue is not empty) {
            pop node N off queue
            mark node N as visited

            if (N is goal)
                  return true;
            else {
                  for each node O that is child of N
                        if O is not marked visited
                              push O onto queue
            }
      }
      return false;
}
```

Mark the node as visited!

Breadth-First Search

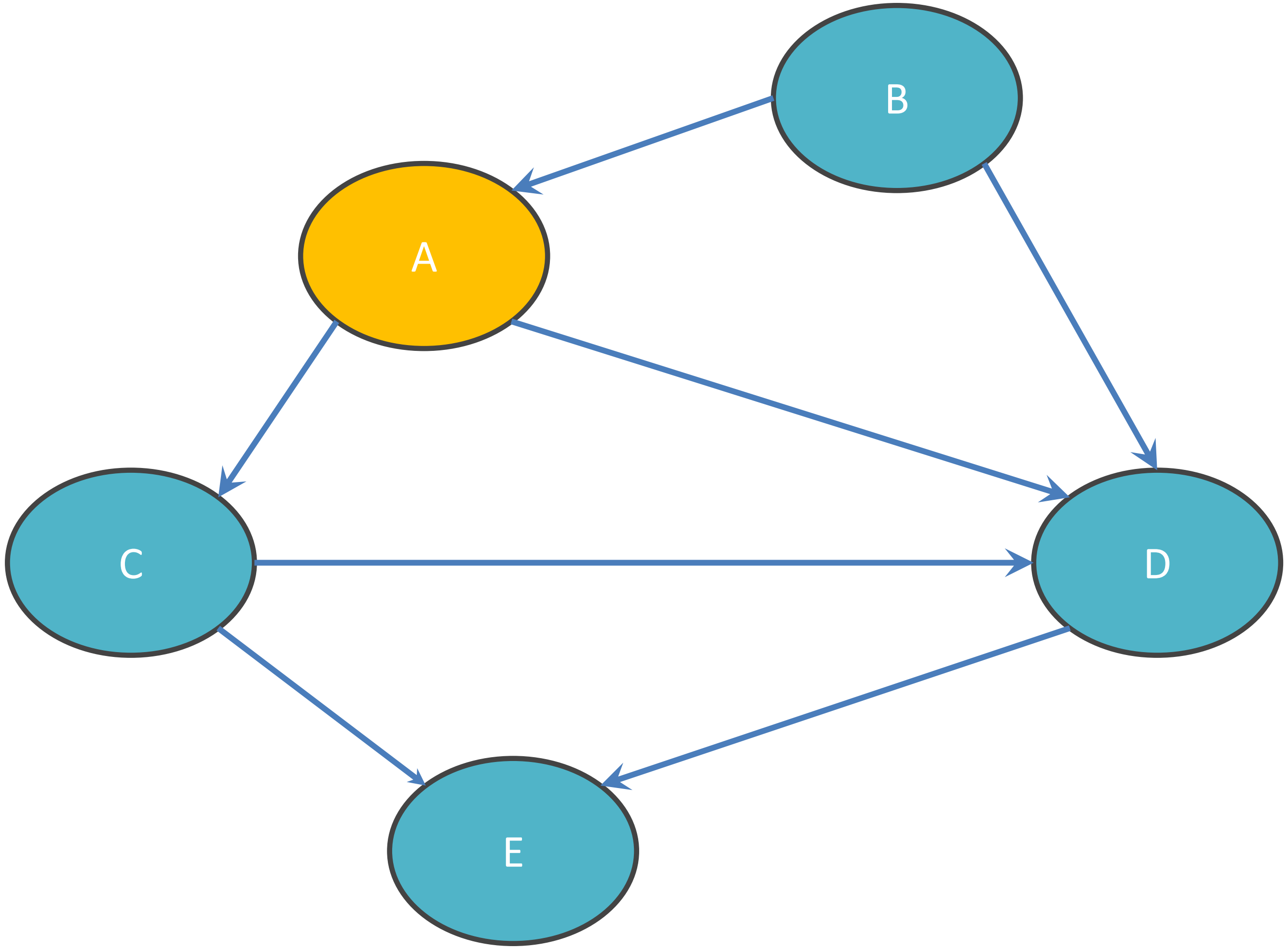
Q: <>



Breadth-First Search

Q: <>

Q: <A>

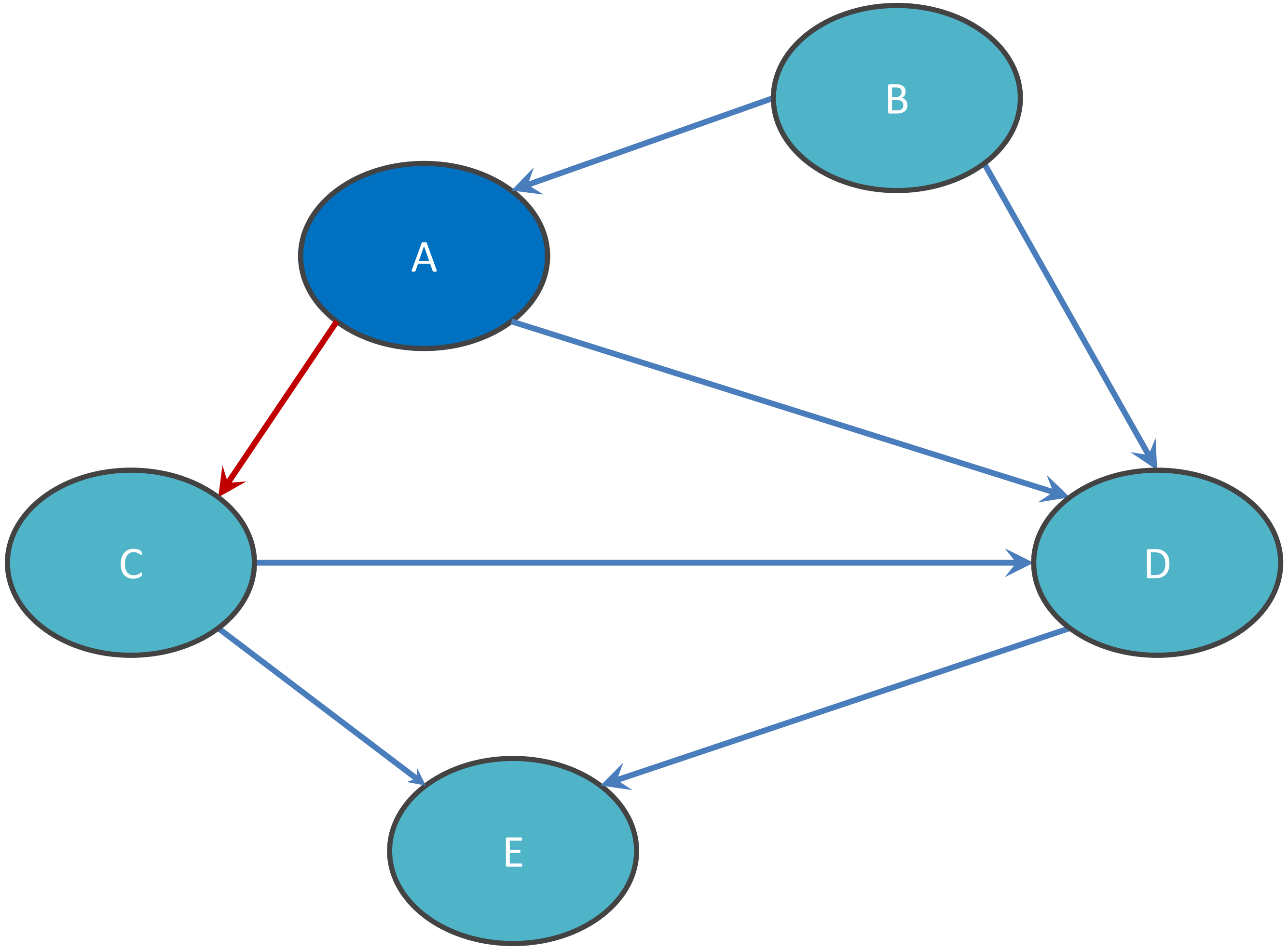


Breadth-First Search

Q: <>

Q: <A>

Q: <>



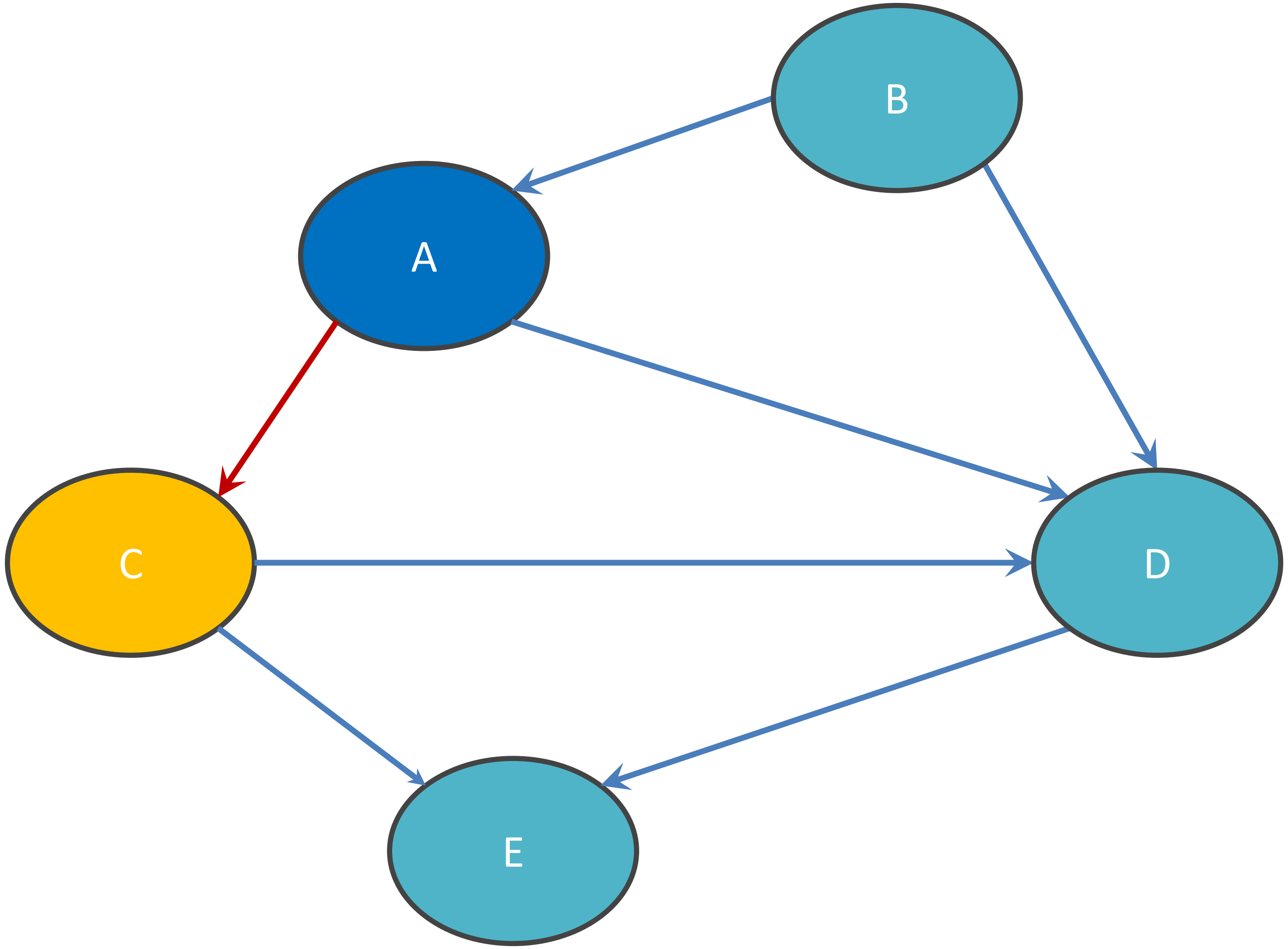
Breadth-First Search

Q: <>

Q: <A>

Q: <>

Q: <C>



Breadth-First Search

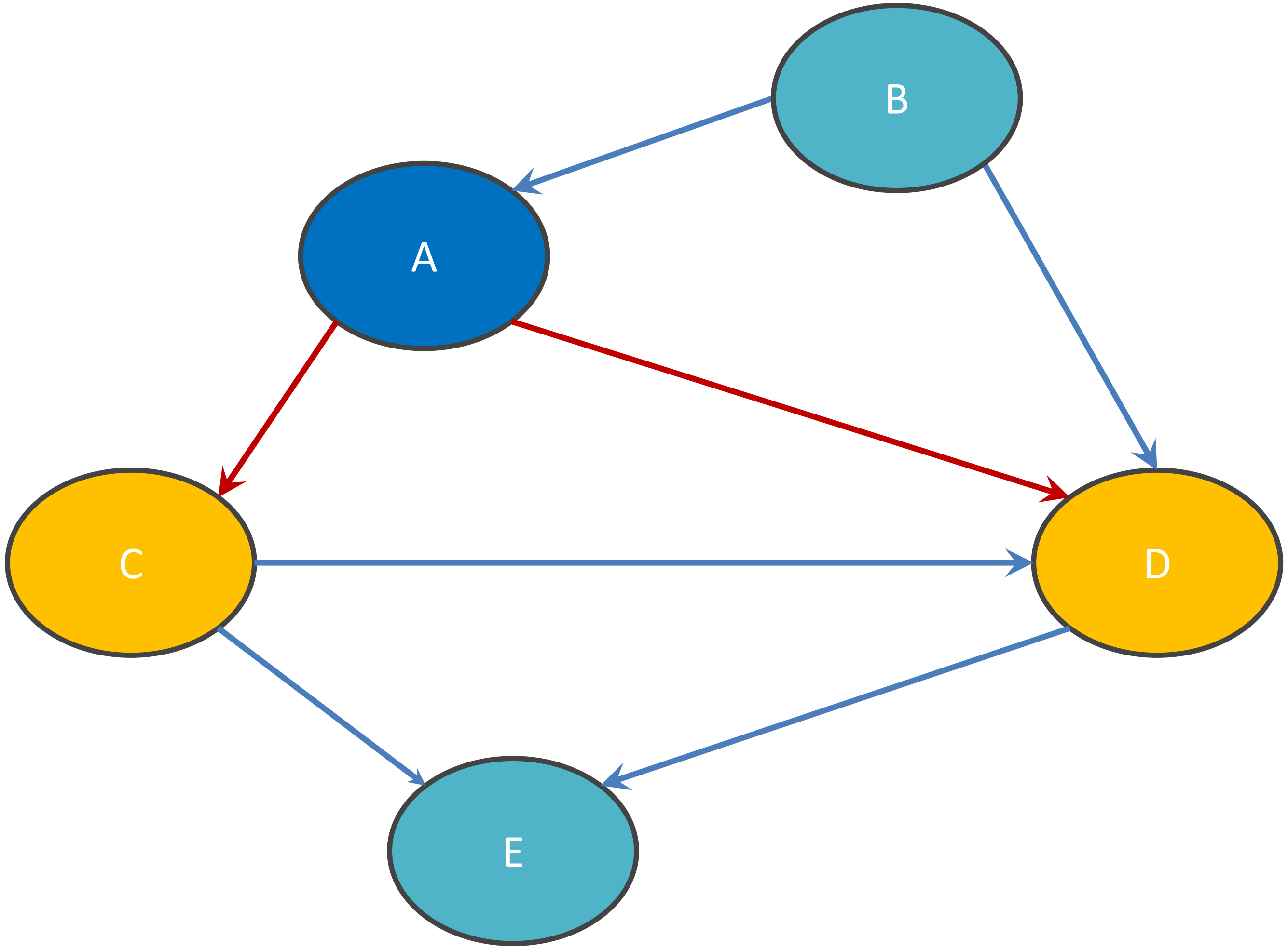
Q: <>

Q: <A>

Q: <>

Q: <C>

Q: <C ,D>



Breadth-First Search

Q: <>

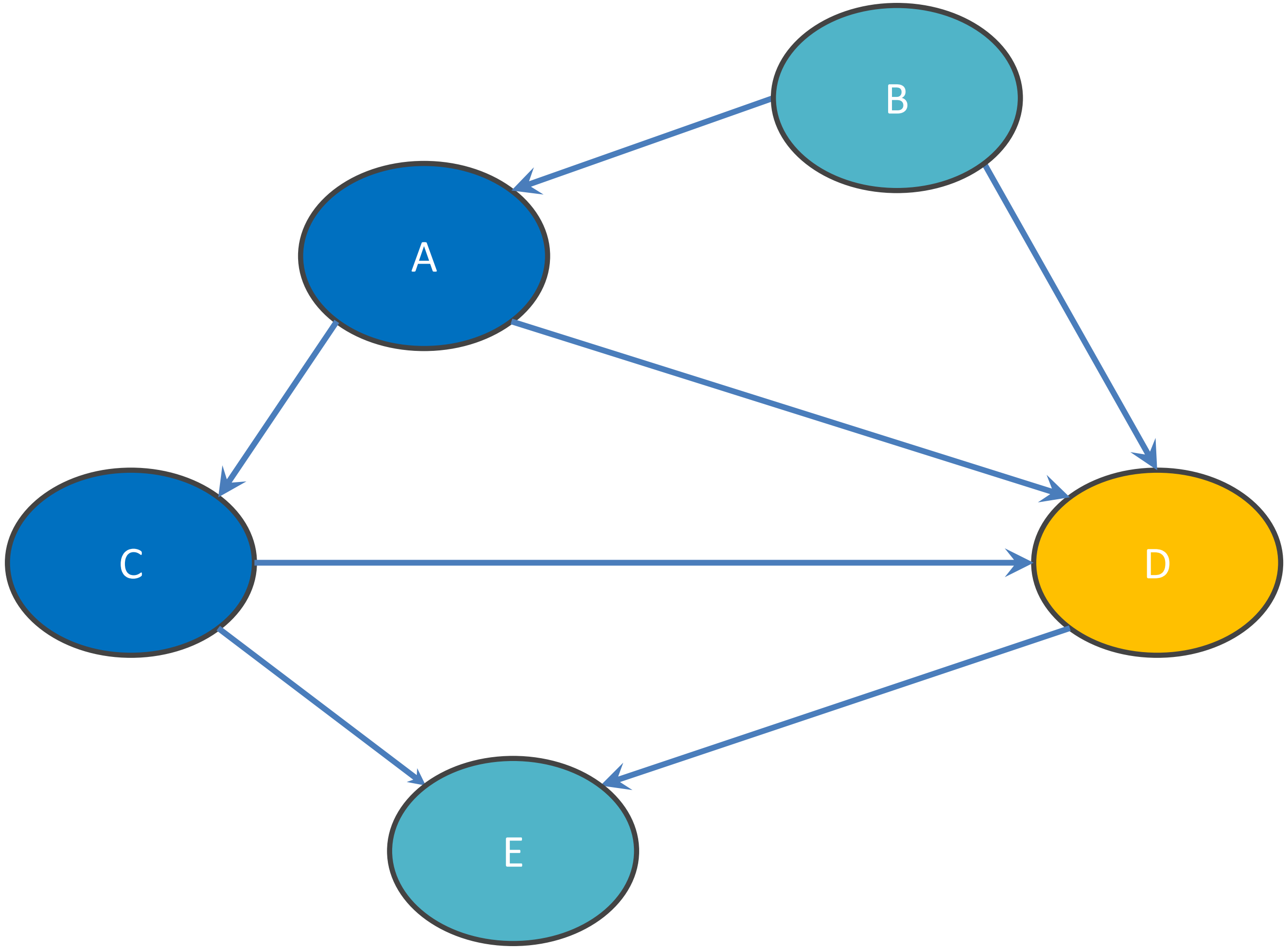
Q: <A>

Q: <>

Q: <C>

Q: <C ,D>

Q: <D>



Breadth-First Search

Q: <>

Q: <A>

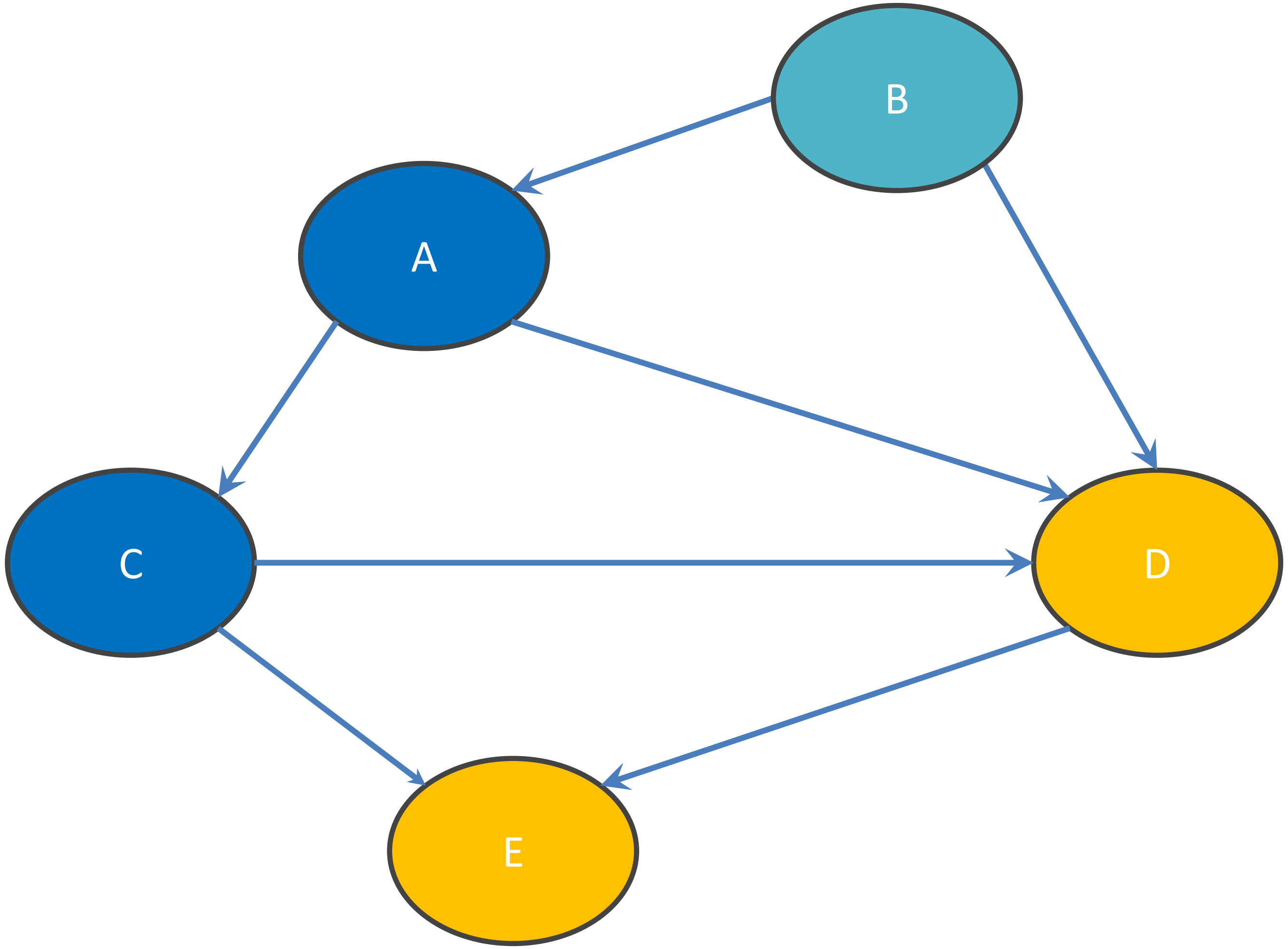
Q: <>

Q: <C>

Q: <C ,D>

Q: <D>

Q: <D, E>



Breadth-First Search

Q: <>

Q: <A>

Q: <>

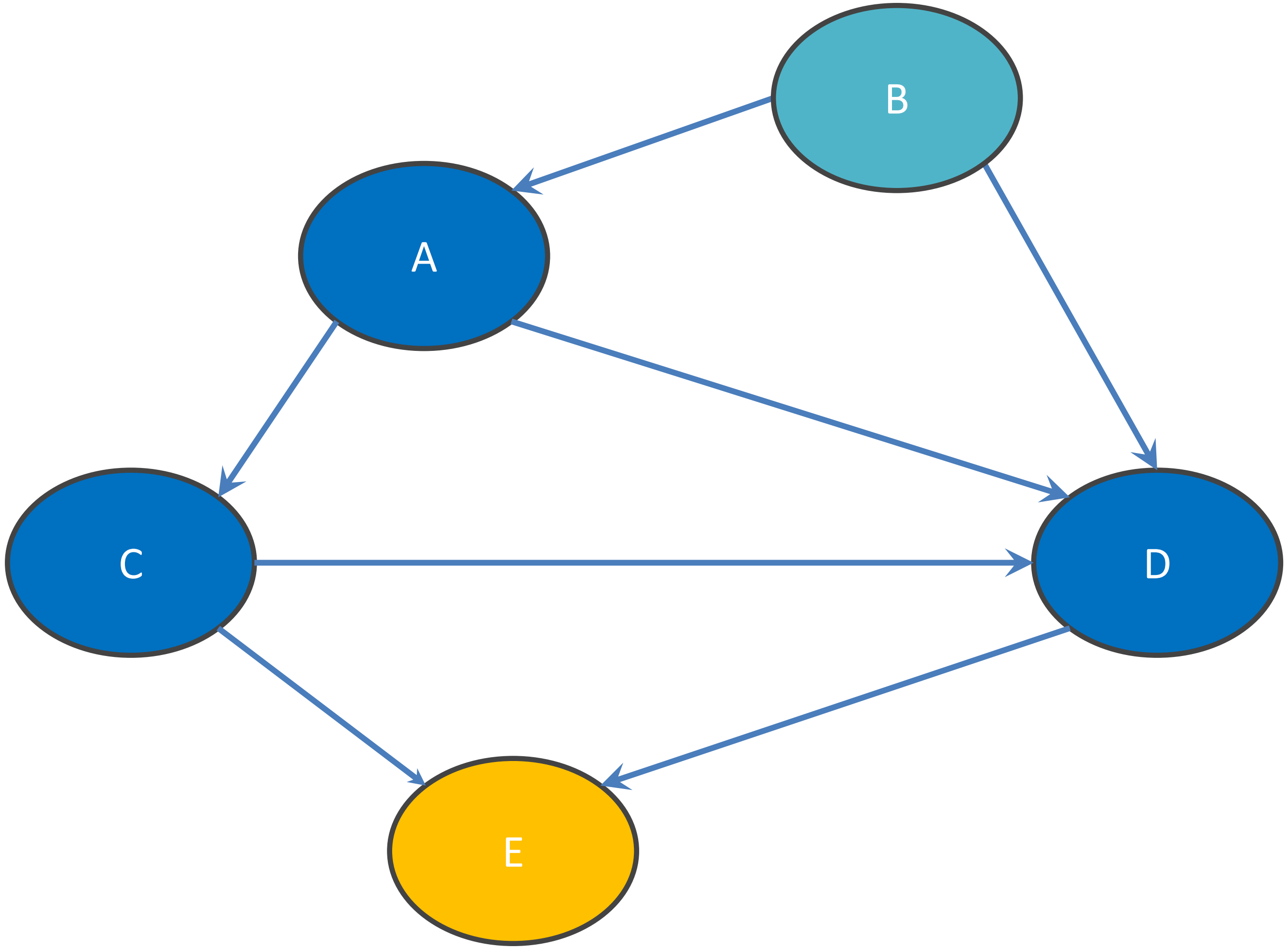
Q: <C>

Q: <C ,D>

Q: <D>

Q: <D, E>

Q: <E>



Breadth-First Search

Q: <>

Q: <A>

Q: <>

Q: <C>

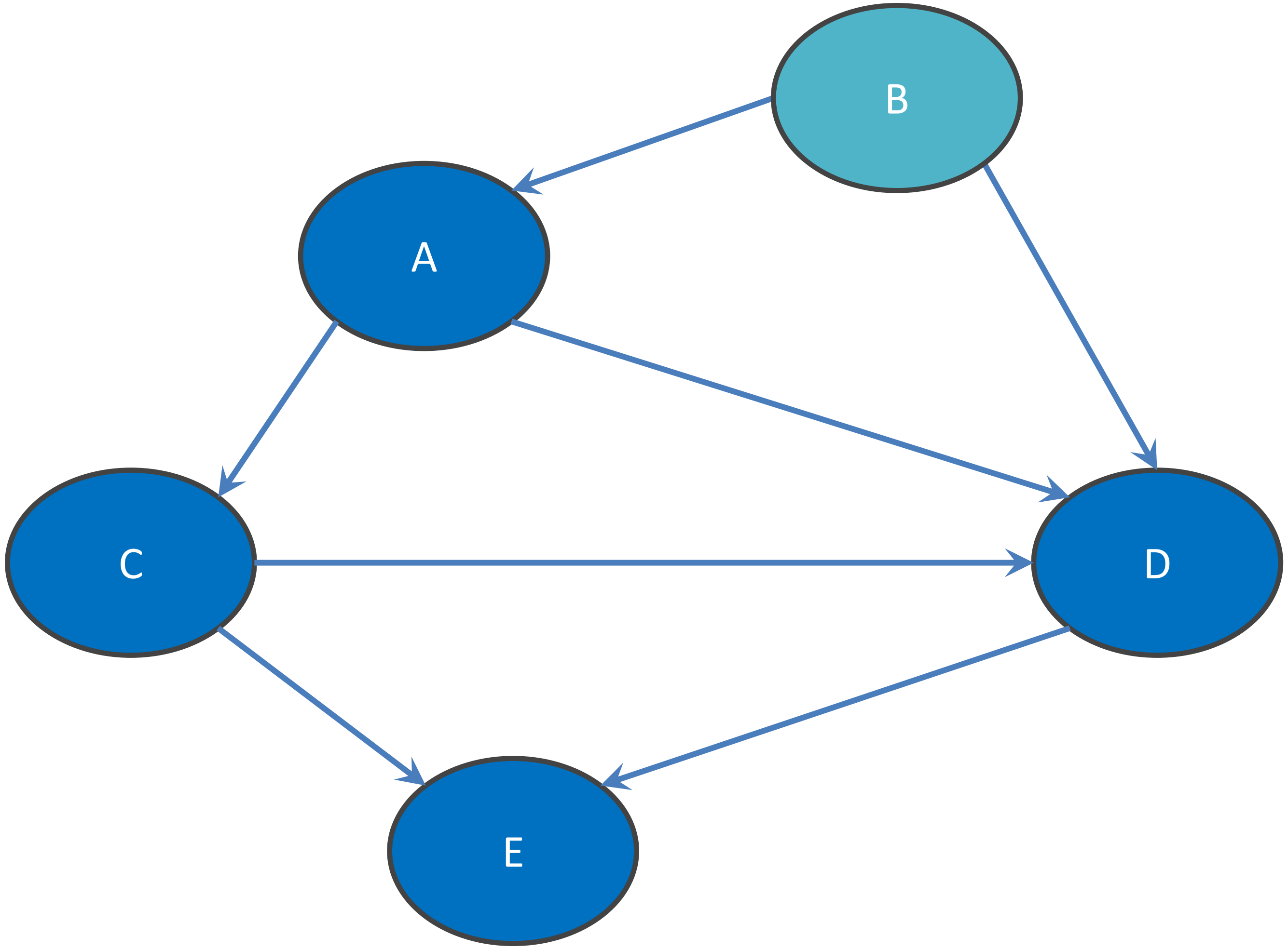
Q: <C ,D>

Q: <D>

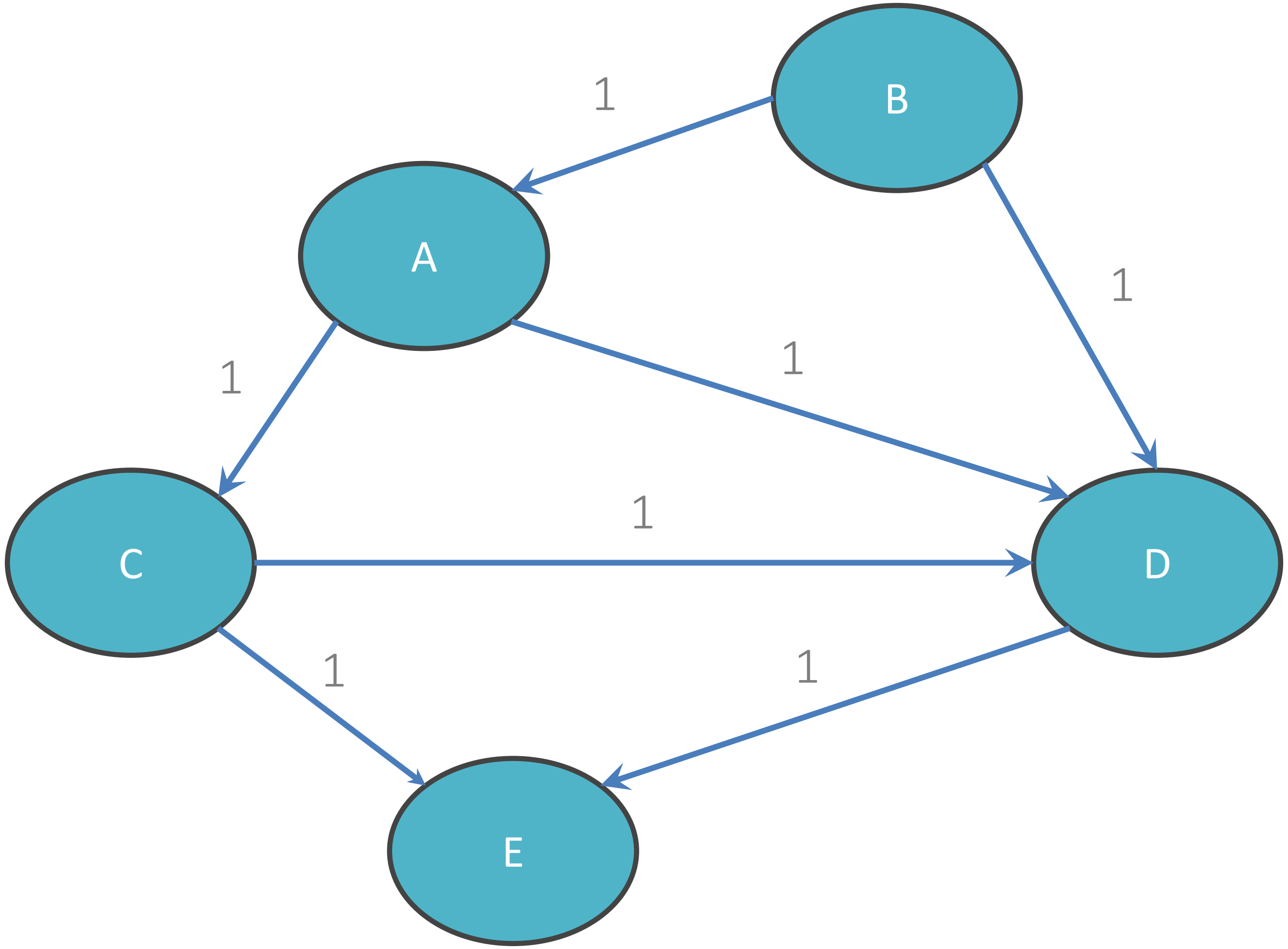
Q: <D, E>

Q: <E>

DONE



Shortest Paths with BFS

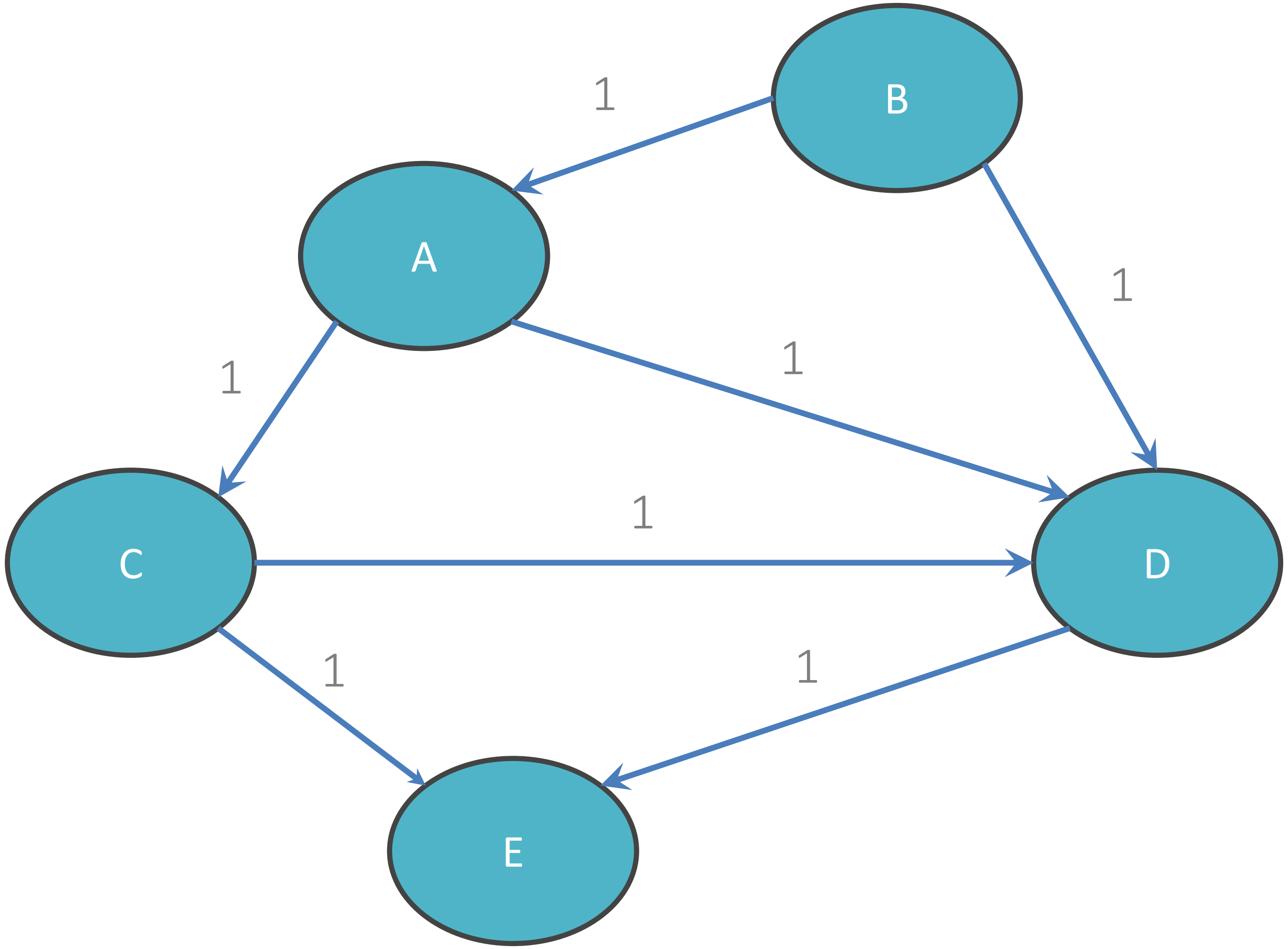


From Node B

Destination	Path	Cost
A	<B,A>	1
B	<B>	0
C	<B,A,C>	2
D		
E		

Shortest path to D? to E?
What are the costs?

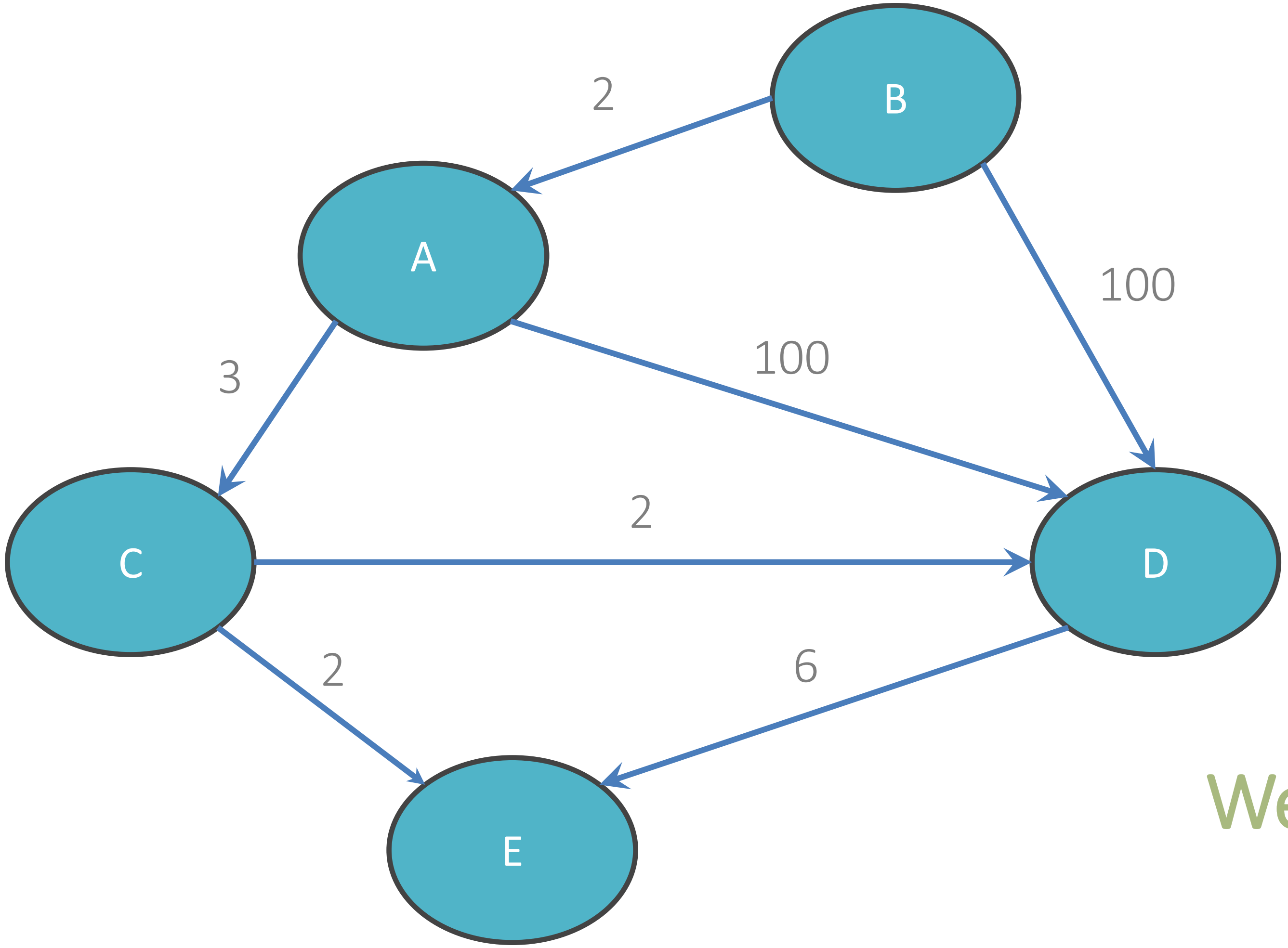
Shortest Paths with BFS



From Node B

Destination	Path	Cost
A	<B,A>	1
B	<B>	0
C	<B,A,C>	2
D	<B,D>	1
E	<B,D,E>	2

Shortest Paths with Weights



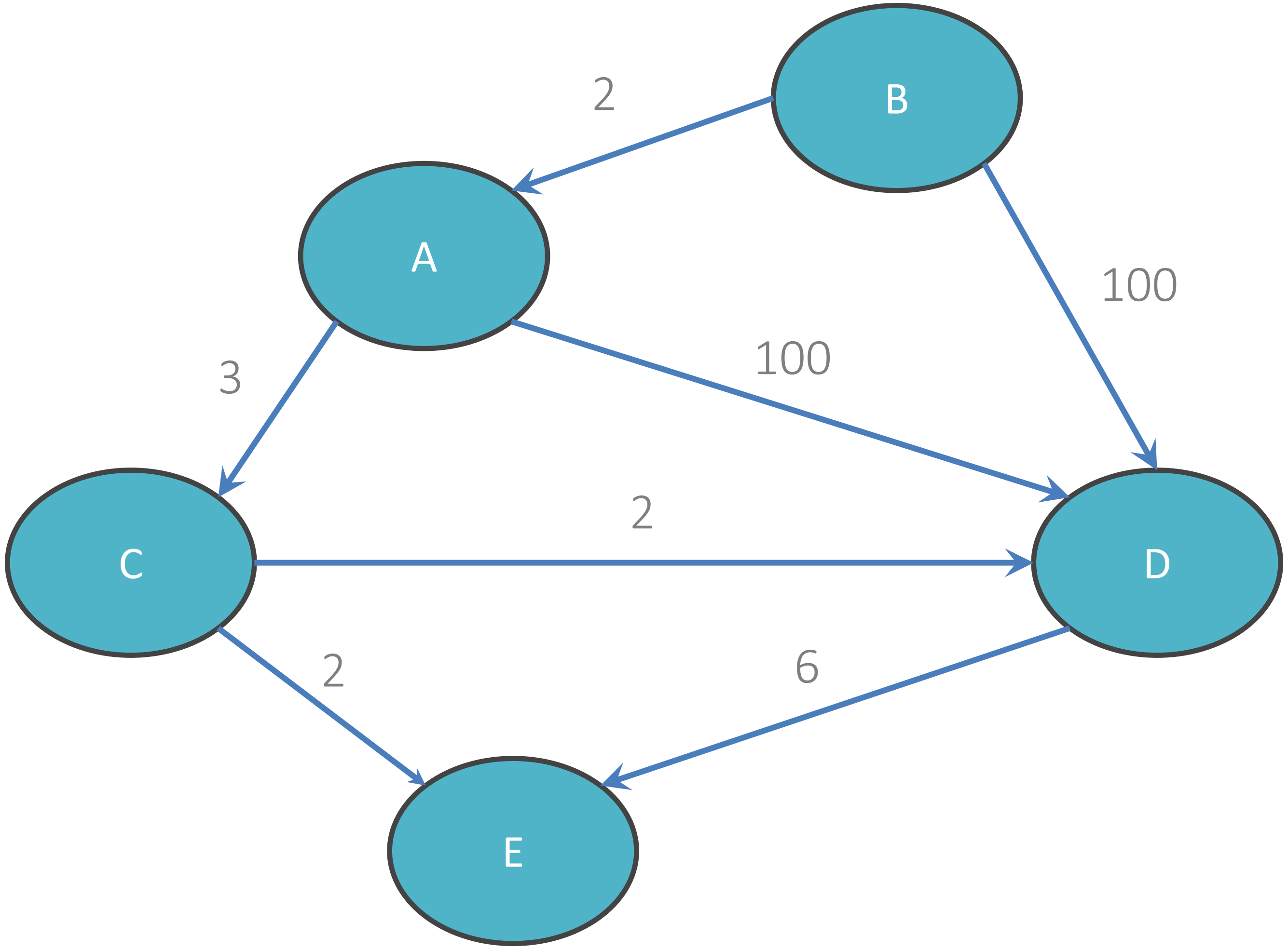
From Node B

Destination	Path	Cost
A	<B,A>	2
B	<B>	0
C	<B,A,C>	5
D		
E		

Weights are not the same!

Are the paths?

Shortest Paths with Weights



From Node B

Destination	Path	Cost
A	<B,A>	2
B	<B>	0
C	<B,A,C>	5
D	<B,A,C,D>	7
E	<B,A,C,E>	7

Midterm review

Midterm topics

Reasoning about code

Specification vs. Implementation

Abstract Data Types (ADTs)

Identity & equality

Testing

Reasoning about code 1

Using backwards reasoning, find the weakest precondition for each sequence of statements and postcondition below. Insert appropriate assertions in each blank line. You should simplify your answers if possible.

```
{_______________}
z = x + y;
{_______________}
y = z – 3;
{x > y}
```

Reasoning about code 1

Using backwards reasoning, find the weakest precondition for each sequence of statements and postcondition below. Insert appropriate assertions in each blank line. You should simplify your answers if possible.

```
{_______________}
z = x + y;
{x > z - 3}
y = z - 3;
{x > y}
```

Reasoning about code 1

Using backwards reasoning, find the weakest precondition for each sequence of statements and postcondition below. Insert appropriate assertions in each blank line. You should simplify your answers if possible.

```
{x > x + y - 3 => y < 3}
z = x + y;
{x > z - 3}
y = z - 3;
{x > y}
```

Reasoning about code 1

Using backwards reasoning, find the weakest precondition for each sequence of statements and postcondition below. Insert appropriate assertions in each blank line. You should simplify your answers if possible.

```
{_______________}
p = a + b;
{_______________}
q = a - b;
{p + q = 42}
```

Reasoning about code 1

Using backwards reasoning, find the weakest precondition for each sequence of statements and postcondition below. Insert appropriate assertions in each blank line. You should simplify your answers if possible.

```
{________________}
p = a + b;
{p + a - b = 42}
q = a - b;
{p + q = 42}
```

Reasoning about code 1

Using backwards reasoning, find the weakest precondition for each sequence of statements and postcondition below. Insert appropriate assertions in each blank line. You should simplify your answers if possible.

```
{a + b + a – b = 42 ⇒ a = 21}
p = a + b;
{p + a - b = 42}
q = a - b;
{p + q = 42}
```

Specification vs. Implementation

Suppose we have a `BankAccount` *class with instance variable* `balance`. *Consider the following specifications:*

```
A.  @effects decreases balance by amount

B.  @requires amount >= 0 and amount <= balance
    @effects decreases balance by amount

C.  @throws InsufficientFundsException
        if balance < amount
    @effects decreases balance by amount
```

Which specifications does this implementation meet?

```
I.  void withdraw(int amount) {
        balance -= amount;
    }
```

Another way to ask the question:

If the client does not know the implementation, will the method do what the client expects it to do based on the specification?

Specification vs. Implementation

Suppose we have a `BankAccount` *class with instance variable* `balance`*. Consider the following specifications:*

```
A.  @effects decreases balance by amount
```

✔ does exactly what the spec says

```
B.  @requires amount >= 0 and amount <= balance
    @effects decreases balance by amount

C.  @throws InsufficientFundsException
        if balance < amount
    @effects decreases balance by amount
```

Which specifications does this implementation meet?

```
I.   void withdraw(int amount) {
        balance -= amount;
     }
```

Specification vs. Implementation

Suppose we have a `BankAccount` *class with instance variable* `balance`*. Consider the following specifications:*

```
A.  @effects decreases balance by amount
```

✔ does exactly what the spec says

```
B.  @requires amount >= 0 and amount <= balance
    @effects decreases balance by amount
```

✔ If the client follows the `@requires` precondition, the code will execute as expected

```
C.  @throws InsufficientFundsException
        if balance < amount
    @effects decreases balance by amount
```

Which specifications does this implementation meet?

```
I.  void withdraw(int amount) {
        balance -= amount;
    }
```

Specification vs. Implementation

Suppose we have a `BankAccount` *class with instance variable* `balance`*. Consider the following specifications:*

```
A.  @effects decreases balance by amount
```

✔ does exactly what the spec says

```
B.  @requires amount >= 0 and amount <= balance
    @effects decreases balance by amount
```

✔ If the client follows the `@requires` precondition, the code will execute as expected

```
C.  @throws InsufficientFundsException
        if balance < amount
    @effects decreases balance by amount
```

✘ Method never throws an exception

Which specifications does this implementation meet?

```
I.   void withdraw(int amount) {
         balance -= amount;
     }
```

Specification vs. Implementation

Suppose we have a `BankAccount` *class with instance variable* `balance`. *Consider the following specifications:*

```
A.  @effects decreases balance by amount

B.  @requires amount >= 0 and amount <= balance
    @effects decreases balance by amount

C.  @throws InsufficientFundsException
        if balance < amount
    @effects decreases balance by amount
```

Which specifications does this implementation meet?

```
II. void withdraw(int amount) {
        if (balance >= amount) balance -= amount;
    }
```

Specification vs. Implementation

Suppose we have a `BankAccount` *class with instance variable* `balance`*. Consider the following specifications:*

```
A.  @effects decreases balance by amount
```

✘ balance does not always decrease

```
B.  @requires amount >= 0 and amount <= balance
    @effects decreases balance by amount

C.  @throws InsufficientFundsException
        if balance < amount
    @effects decreases balance by amount
```

Which specifications does this implementation meet?

```
II. void withdraw(int amount) {
        if (balance >= amount) balance -= amount;
    }
```

Specification vs. Implementation

Suppose we have a `BankAccount` *class with instance variable* `balance`*. Consider the following specifications:*

```
A.  @effects decreases balance by amount
```

✘ balance does not always decrease

```
B.  @requires amount >= 0 and amount <= balance
    @effects decreases balance by amount
```

✔ If the client follows the `@requires` precondition, the code will execute as expected

```
C.  @throws InsufficientFundsException
        if balance < amount
    @effects decreases balance by amount
```

Which specifications does this implementation meet?

```
II. void withdraw(int amount) {
        if (balance >= amount) balance -= amount;
    }
```

Specification vs. Implementation

Suppose we have a `BankAccount` *class with instance variable* `balance`*. Consider the following specifications:*

```
A.  @effects decreases balance by amount
```
✘ balance does not always decrease

```
B.  @requires amount >= 0 and amount <= balance
    @effects decreases balance by amount
```
✔ If the client follows the `@requires` precondition, the code will execute as expected

```
C.  @throws InsufficientFundsException
        if balance < amount
    @effects decreases balance by amount
```
✘ Method never throws an exception

Which specifications does this implementation meet?

```
II. void withdraw(int amount) {
        if (balance >= amount) balance -= amount;
    }
```

Specification vs. Implementation

Suppose we have a `BankAccount` *class with instance variable* `balance`*. Consider the following specifications:*

```
A.  @effects decreases balance by amount

B.  @requires amount >= 0 and amount <= balance
    @effects decreases balance by amount

C.  @throws InsufficientFundsException
        if balance < amount
    @effects decreases balance by amount
```

Which specifications does this implementation meet?

```
III.void withdraw(int amount) {
        if (amount < 0) throw new IllegalArgumentException();
        balance -= amount;
     }
```

Specification vs. Implementation

Suppose we have a `BankAccount` *class with instance variable* `balance`. *Consider the following specifications:*

```
A.  @effects decreases balance by amount
```

✘ balance does not always decrease

```
B.  @requires amount >= 0 and amount <= balance
    @effects decreases balance by amount

C.  @throws InsufficientFundsException
        if balance < amount
    @effects decreases balance by amount
```

Which specifications does this implementation meet?

```
III.void withdraw(int amount) {
        if (amount < 0) throw new IllegalArgumentException();
        balance -= amount;
    }
```

Specification vs. Implementation

Suppose we have a `BankAccount` *class with instance variable* `balance`*. Consider the following specifications:*

```
A.  @effects decreases balance by amount
```

✘ balance does not always decrease

```
B.  @requires amount >= 0 and amount <= balance
    @effects decreases balance by amount
```

✔ If the client follows the `@requires` precondition, the code will execute as expected

```
C.  @throws InsufficientFundsException
        if balance < amount
    @effects decreases balance by amount
```

Which specifications does this implementation meet?

```
III.void withdraw(int amount) {
        if (amount < 0) throw new IllegalArgumentException();
        balance -= amount;
    }
```

Specification vs. Implementation

Suppose we have a `BankAccount` *class with instance variable* `balance`*. Consider the following specifications:*

```
A.  @effects decreases balance by amount
```
✘ balance does not always decrease

```
B.  @requires amount >= 0 and amount <= balance
    @effects decreases balance by amount
```
✔ If the client follows the `@requires` precondition, the code will execute as expected

```
C.  @throws InsufficientFundsException
        if balance < amount
    @effects decreases balance by amount
```
✘ Method throws wrong exception for wrong reason

Which specifications does this implementation meet?

```
III.void withdraw(int amount) {
        if (amount < 0) throw new IllegalArgumentException();
        balance -= amount;
    }
```

Specification vs. Implementation

Suppose we have a `BankAccount` *class with instance variable* `balance`*. Consider the following specifications:*

```
A.  @effects decreases balance by amount

B.  @requires amount >= 0 and amount <= balance
    @effects decreases balance by amount

C.  @throws InsufficientFundsException
        if balance < amount
    @effects decreases balance by amount
```

Which specifications does this implementation meet?

```
IV. void withdraw(int amount) throws InsufficientFundsException {
        if (balance < amount) throw new InsufficientFundsException();
        balance -= amount;
    }
```

Specification vs. Implementation

Suppose we have a `BankAccount` *class with instance variable* `balance`*. Consider the following specifications:*

```
A.  @effects decreases balance by amount
```

✘ balance does not always decrease

```
B.  @requires amount >= 0 and amount <= balance
    @effects decreases balance by amount

C.  @throws InsufficientFundsException
        if balance < amount
    @effects decreases balance by amount
```

Which specifications does this implementation meet?

```
IV. void withdraw(int amount) throws InsufficientFundsException {
        if (balance < amount) throw new InsufficientFundsException();
        balance -= amount;
    }
```

Specification vs. Implementation

Suppose we have a `BankAccount` *class with instance variable* `balance`*. Consider the following specifications:*

```
A.  @effects decreases balance by amount
```

✘ balance does not always decrease

```
B.  @requires amount >= 0 and amount <= balance
    @effects decreases balance by amount
```

✔ If the client follows the `@requires` precondition, the code will execute as expected

```
C.  @throws InsufficientFundsException
        if balance < amount
    @effects decreases balance by amount
```

Which specifications does this implementation meet?

```
IV. void withdraw(int amount) throws InsufficientFundsException {
        if (balance < amount) throw new InsufficientFundsException();
        balance -= amount;
    }
```

Specification vs. Implementation

Suppose we have a `BankAccount` *class with instance variable* `balance`*. Consider the following specifications:*

```
A.  @effects decreases balance by amount
```

✘ balance does not always decrease

```
B.  @requires amount >= 0 and amount <= balance
    @effects decreases balance by amount
```

✔ If the client follows the `@requires` precondition, the code will execute as expected

```
C.  @throws InsufficientFundsException
        if balance < amount
    @effects decreases balance by amount
```

✔ Method does what the spec says

Which specifications does this implementation meet?

```
IV. void withdraw(int amount) throws InsufficientFundsException {
        if (balance < amount) throw new InsufficientFundsException();
        balance -= amount;
    }
```

Specifications 2

```
/**
 * An IntPoly is an immutable, integer-valued polynomial
 * with integer coefficients. A typical IntPoly value
 * is a_0 + a_1*x + a_2*x^2 + ... + a_n*x_n. An IntPoly
 * with degree n has coefficent a_n != 0, except that the
 * zero polynomial is represented as a polynomial of
 * degree 0 and a_0 = 0 in that case.
 */

public class IntPoly {
    int a[];
    // AF(this) = a has n+1 entries, and for each entry,
    // a[i] = coefficient a_i of the polynomial.
}
```

Specifications 2

```
/**
 * Return a new IntPoly that is the sum of this and other
 * @requires
 * @modifies
 * @effects
 * @return
 * @throws
 */
public IntPoly add(IntPoly other)
```

Specifications 2

```
/**
 * Return a new IntPoly that is the sum of this and other
 * @requires other != null
 * @modifies none
 * @effects none
 * @return a new IntPoly representing the sum of this and other
 * @throws none
 */
public IntPoly add(IntPoly other)
```

Representation invariants

One of your colleagues is worried that this creates a potential representation exposure problem. Another colleague says there's no problem since an `IntPoly` *is immutable. Is there a problem? Give a brief justification for your answer.*

```
public class IntPoly {
    int a[];
    // AF(this) = a has n+1 entries, and for each entry,
    // a[i] = coefficient a_i of the polynomial.

    // Return the coefficients of this IntPoly
    public int[] getCoeffs() {
        return a;
    }
}
```

Representation invariants

One of your colleagues is worried that this creates a potential representation exposure problem. Another colleague says there's no problem since an `IntPoly` *is immutable. Is there a problem? Give a brief justification for your answer.*

```
public class IntPoly {
    int a[];
    // AF(this) = a has n+1 entries, and for each entry,
    // a[i] = coefficient a_i of the polynomial.

    // Return the coefficients of this IntPoly
    public int[] getCoeffs() {
        return a;
    }
}
```

The return value is a reference to the same coefficient array stored in the **`IntPoly`** and the client code could alter those coefficients.

Representation invariants

If there is a representation exposure problem, give a new or repaired implementation of `getCoeffs` *that fixes the problem but still returns the coefficients of the* `IntPoly` *to the client. If it saves time you can give a precise description of the changes needed instead of writing the detailed Java code.*

```
public class IntPoly {
    int a[];
    // AF(this) = a has n+1 entries, and for each entry,
    // a[i] = coefficient a_i of the polynomial.

    // Return the coefficients of this IntPoly
    public int[] getCoeffs() {
        return a;
    }
}
```

Representation invariants

If there is a representation exposure problem, give a new or repaired implementation of `getCoeffs` *that fixes the problem but still returns the coefficients of the* `IntPoly` *to the client. If it saves time you can give a precise description of the changes needed instead of writing the detailed Java code.*

```
public int[] getCoeffs() {
    int[] copyA = new int[a.length];
    for (int i = 0; i < copyA.length; i++) {
        copyA[i] = a[i]
    }
    return copyA
}
```

Representation invariants

If there is a representation exposure problem, give a new or repaired implementation of `getCoeffs` *that fixes the problem but still returns the coefficients of the* `IntPoly` *to the client. If it saves time you can give a precise description of the changes needed instead of writing the detailed Java code.*

```
public int[] getCoeffs() {
    int[] copyA = new int[a.length];
    for (int i = 0; i < copyA.length; i++) {
        copyA[i] = a[i]
    }
    return copyA
}
```

1. Make a copy
2. Return the copy

Representation invariants

If there is a representation exposure problem, give a new or repaired implementation of `getCoeffs` *that fixes the problem but still returns the coefficients of the* `IntPoly` *to the client. If it saves time you can give a precise description of the changes needed instead of writing the detailed Java code.*

```
public int[] getCoeffs() {
    int[] copyA = new int[a.length];
    for (int i = 0; i < copyA.length; i++) {
        copyA[i] = a[i]
    }
    return copyA
}
```

1. Make a copy
2. Return the copy

Alternatively, we can just use…

`Arrays.copyOf(a, a.length)`

Reasoning about code 2

We would like to add a method to this class that evaluates the `IntPoly` *at a particular value x. In other words, given a value x, the method* `valueAt(x)` *should return* $a_0 + a_1x + a_2x^2 + \ldots + a_nx^n$*, where* a_0 *through an are the coefficients of this* `IntPoly`*.*

For this problem, develop an implementation of this method and prove that your implementation is correct.

(see starter code on next slide)

Reasoning about code 2

```
/** Return the value of this IntPoly at point x */
public int valueAt(int x) {
    int val = a[0];
    int xk = 1;
    int k = 0;
    int n = a.length-1; // degree of this, n >=0
    {_____}
    while (k != n) {
        {_____}
        xk = xk * x;
        {_____}
        val = val + a[k+1]*xk;
        {_____}
        k = k + 1;
        {_____}
    }
    {_____}
    return val;
}
```

Reasoning about code 2

```
/** Return the value of this IntPoly at point x */
public int valueAt(int x) {
    int val = a[0];
    int xk = 1;
    int k = 0;
    int n = a.length-1; // degree of this, n >=0
    {inv: xk = x^k && val = a[0] + a[1]*x + ... + a[k]*x^k}
    while (k != n) {
        {_____}
        xk = xk * x;
        {_____}
        val = val + a[k+1]*xk;
        {_____}
        k = k + 1;
        {_____}
    }
    {_____}
    return val;
}
```

This should come with the code...

Reasoning about code 2

```
/** Return the value of this IntPoly at point x */
public int valueAt(int x) {
    int val = a[0];
    int xk = 1;
    int k = 0;
    int n = a.length-1; // degree of this, n >=0
    {inv: xk = x^k && val = a[0] + a[1]*x + ... + a[k]*x^k}
    while (k != n) {
        {inv && k != n}
        xk = xk * x;
        {_____}
        val = val + a[k+1]*xk;
        {_____}
        k = k + 1;
        {_____}
    }
    {_____}
    return val;
}
```

Reasoning about code 2

```
/** Return the value of this IntPoly at point x */
public int valueAt(int x) {
    int val = a[0];
    int xk = 1;
    int k = 0;
    int n = a.length-1; // degree of this, n >=0
    {inv: xk = x^k && val = a[0] + a[1]*x + ... + a[k]*x^k}
    while (k != n) {
        {inv && k != n}
        xk = xk * x;
        {xk = x^(k+1) && val = a[0] + a[1]*x + ... + a[k]*x^k}
        val = val + a[k+1]*xk;
        {_____}
        k = k + 1;
        {_____}
    }
    {_____}
    return val;
}
```

Reasoning about code 2

```
/** Return the value of this IntPoly at point x */
public int valueAt(int x) {
    int val = a[0];
    int xk = 1;
    int k = 0;
    int n = a.length-1; // degree of this, n >=0
    {inv: xk = x^k && val = a[0] + a[1]*x + ... + a[k]*x^k}
    while (k != n) {
        {inv && k != n}
        xk = xk * x;
        {xk = x^(k+1) && val = a[0] + a[1]*x + ... + a[k]*x^k}
        val = val + a[k+1]*xk;
        {xk = x^(k+1) && val = a[0] + a[1]*x + ... + a[k+1]*x^(k+1)}
        k = k + 1;
        {_____}
    }
    {_____}
    return val;
}
```

Reasoning about code 2

```
/** Return the value of this IntPoly at point x */
public int valueAt(int x) {
    int val = a[0];
    int xk = 1;
    int k = 0;
    int n = a.length-1; // degree of this, n >=0
    {inv: xk = x^k && val = a[0] + a[1]*x + ... + a[k]*x^k}
    while (k != n) {
        {inv && k != n}
        xk = xk * x;
        {xk = x^(k+1) && val = a[0] + a[1]*x + ... + a[k]*x^k}
        val = val + a[k+1]*xk;
        {xk = x^(k+1) && val = a[0] + a[1]*x + ... + a[k+1]*x^(k+1)}
        k = k + 1;
        {inv}
    }
    {_____}
    return val;
}
```

Reasoning about code 2

```
/** Return the value of this IntPoly at point x */
public int valueAt(int x) {
    int val = a[0];
    int xk = 1;
    int k = 0;
    int n = a.length-1; // degree of this, n >=0
    {inv: xk = x^k && val = a[0] + a[1]*x + ... + a[k]*x^k}
    while (k != n) {
        {inv && k != n}
        xk = xk * x;
        {xk = x^(k+1) && val = a[0] + a[1]*x + ... + a[k]*x^k}
        val = val + a[k+1]*xk;
        {xk = x^(k+1) && val = a[0] + a[1]*x + ... + a[k+1]*x^(k+1)}
        k = k + 1;
        {inv}
    }
    {inv && k = n ⇒ val = a[0] + a[1]*x + ... + a[n]*x^n}
    return val;
}
```

Equality

*Suppose we are defining a class **`StockItem`** to represent items stocked by an online grocery store. Here is the start of the class definition, including the class name and instance variables:*

```
public class StockItem {
    String name;
    String size;
    String description;
    int quantity;

    /* Construct a new StockItem */
    public StockItem(...);
}
```

Equality

A summer intern was asked to implement an `equals` *function for this class that treats two* `StockItem` *objects as equal if their* `name` *and* `size` *fields match. Here's the result:*

```
/** return true if the name and size fields match */
public boolean equals(StockItem other) {
    return name.equals(other.name) && size.equals(other.size);
}
```

This `equals` *method seems to work sometimes but not always. Give an example showing a situation when it fails.*

Equality

A summer intern was asked to implement an `equals` *function for this class that treats two* `StockItem` *objects as equal if their* `name` *and* `size` *fields match. Here's the result:*

```
/** return true if the name and size fields match */
public boolean equals(StockItem other) {
    return name.equals(other.name) && size.equals(other.size);
}
```

This `equals` *method seems to work sometimes but not always. Give an example showing a situation when it fails.*

```
Object s1 = new StockItem("thing", 1, "stuff", 1);
Object s2 = new StockItem("thing", 1, "stuff", 1);
System.out.println(s1.equals(s2));
```

Equality

A summer intern was asked to implement an `equals` *function for this class that treats two* `StockItem` *objects as equal if their* `name` *and* `size` *fields match. Here's the result:*

```
/** return true if the name and size fields match */
public boolean equals(StockItem other) {  // equals is overloaded, not overridden
    return name.equals(other.name) && size.equals(other.size);
}
```

This `equals` *method seems to work sometimes but not always. Give an example showing a situation when it fails.*

```
Object s1 = new StockItem("thing", 1, "stuff", 1);
Object s2 = new StockItem("thing", 1, "stuff", 1);
System.out.println(s1.equals(s2));
```

Equality

Show how you would fix the `equals` *method so it works properly (*`StockItems` *are equal if their* `names` *and* `sizes` *are equal)*

```
/** return true if the name and size fields match */
```

Equality

Show how you would fix the `equals` *method so it works properly (*`StockItems` *are equal if their* `names` *and* `sizes` *are equal)*

```
/** return true if the name and size fields match */
@Override
public boolean equals(Object o) {
    if (!(o instanceof StockItem)) {
        return false;
    }
    StockItem other = (StockItem) o;
    return name.equals(other.name) && size.equals(other.size);
}
```

hashCode

Which of the following implementations of `hashCode()` *for the* ***`StockItem`*** *class are legal:*

```
1.  return name.hashCode();
2.  return name.hashCode() * 17 + size.hashCode();
3.  return name.hashCode() * 17 + quantity;
4.  return quantity;
```

hashCode

Which of the following implementations of `hashCode()` *for the* ***`StockItem`*** *class are legal:*

1. `return name.hashCode();` ✔ legal
2. `return name.hashCode() * 17 + size.hashCode();`
3. `return name.hashCode() * 17 + quantity;`
4. `return quantity;`

hashCode

Which of the following implementations of `hashCode()` *for the* ***`StockItem`*** *class are legal:*

1. `return name.hashCode();` ✔ legal
2. `return name.hashCode() * 17 + size.hashCode();` ✔ legal
3. `return name.hashCode() * 17 + quantity;`
4. `return quantity;`

hashCode

Which of the following implementations of `hashCode()` *for the* ***`StockItem`*** *class are legal:*

1. `return name.hashCode();` ✔ legal
2. `return name.hashCode() * 17 + size.hashCode();` ✔ legal
3. `return name.hashCode() * 17 + quantity;` ✘ illegal!
4. `return quantity;`

hashCode

Which of the following implementations of `hashCode()` *for the* ***`StockItem`*** *class are legal:*

1. `return name.hashCode();` ✔ legal
2. `return name.hashCode() * 17 + size.hashCode();` ✔ legal
3. `return name.hashCode() * 17 + quantity;` ✘ illegal!
4. `return quantity;` ✘ illegal!

hashCode

Which of the following implementations of `hashCode()` *for the* ***`StockItem`*** *class are legal:*

1. `return name.hashCode();` ✔ legal
2. `return name.hashCode() * 17 + size.hashCode();` ✔ legal
3. `return name.hashCode() * 17 + quantity;` ✘ illegal!
4. `return quantity;` ✘ illegal!

   The **`equals`** method does
   not care about **`quantity`**

hashCode

Which implementation do you prefer?

```
public int hashCode() {
    return name.hashCode();
}
```

```
public int hashCode() {
    return name.hashCode()*17 + size.hashCode();
}
```

hashCode

Which implementation do you prefer?

```
public int hashCode() {
    return name.hashCode();
}
```

(ii) will likely do the best job since it takes into account both the size and name fields. (i) is also legal but it gives the same **`hashCode`** for **`StockItems`** that have different sizes as long as they have the same name, so it doesn't differentiate between different **`StockItems`** as well as (ii).

```
public int hashCode() {
    return name.hashCode()*17 + size.hashCode();
}
```