# CPTS 122
# FINAL EXAM REVIEW

CREATED BY KYLE PARKER

DECEMBER 5TH 2024

# EXAM INFO

- Monday, Dec 9, 1030-1230 (2 hours)
- **No calculators**
- **No notes**
- **Pencils/Pens**

- **True/False**
- **Fill in the blank**
- **MC**
- **Code (define in some methods, declare some classes)**

# MS FORMS QUESTIONS

# HOW TO DEREFERENCE A DOUBLE POINTER

*Suppose we have the following code:*

```
void Requests::setValueDoublePointer(int ** ptr) {
    // What goes here?
}
```

# HOW TO DEREFERENCE A DOUBLE POINTER

*Suppose we have the following code:*

```
void Requests::setValueDoublePointer(int ** ptr) {
    (**ptr) = 100; // Does this work?
}
```

# HOW TO DEREFERENCE A DOUBLE POINTER

*Suppose we have the following code:*

```
void Requests::setValueDoublePointer(int ** ptr) {
    *(*ptr) = 100;
}
```

# HOW TO DEREFERENCE A TRIPLE POINTER

*Suppose we have the following code:*

```
void Requests::setValueTriplePointer(int *** ptr) {
    // What goes here?
}
```

# HOW TO DEREFERENCE A TRIPLE POINTER

*Suppose we have the following code:*

```
void Requests::setValueTriplePointer(int *** ptr) {
    *(*(*ptr)) = 100;
}
```

# UML DIAGRAMS

# CLASS DIAGRAMS

- What can they be used for?
- When are they used?
- What is contained within the diagrams?

# CLASS DIAGRAMS

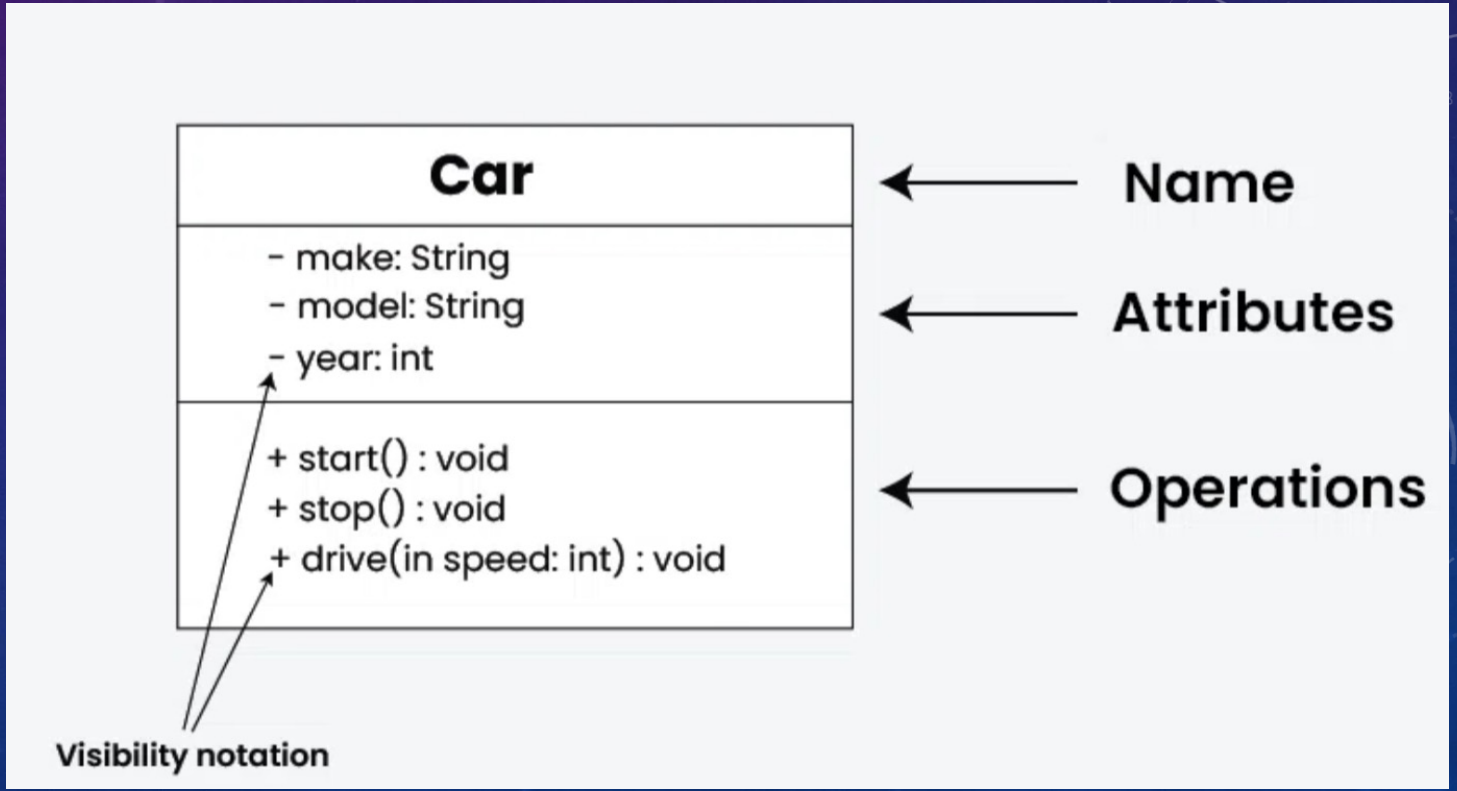
- What can they be used for?
  - Depicting relationships between classes
- When are they used?
- What is contained within the diagrams?

# CLASS DIAGRAMS

- What can they be used for?
  - Depicting relationships between classes
- When are they used?
  - During the design phase
- What is contained within the diagrams?

# CLASS DIAGRAMS

- What can they be used for?
  - Depicting relationships between classes
- When are they used?
  - During the design phase
- What is contained within the diagrams?
  - See right. From GeeksforGeeks



# BIG-O NOTATION

# WHAT IS BIG-O NOTATION?

- One of three ways to express the complexity of a function

# WHAT IS BIG-O NOTATION?

- One of three ways to express the complexity of a function
- Describes runtime and space complexities

# WHAT IS BIG-O NOTATION?

- One of three ways to express the complexity of a function
- Describes runtime and space complexities
- Helps us understand how scalable a given routine is

# WHAT IS BIG-O NOTATION?

- One of three ways to express the complexity of a function
- Describes runtime and space complexities
- Helps us understand how scalable a given routine is

| Notation | Definition | Explanation |
|---|---|---|
| Big O (O) | $f(n) \leq C * g(n)$ for all $n \geq n_0$ | Describes the upper bound of the algorithm's running time in the **worst case**. |
| Ω (Omega) | $f(n) \geq C * g(n)$ for all $n \geq n_0$ | Describes the lower bound of the algorithm's running time in the **best case**. |
| θ (Theta) | $C_1 * g(n) \leq f(n) \leq C_2 * g(n)$ for $n \geq n_0$ | Describes both the upper and lower bounds of the algorithm's **running time**. |

GeeksforGeeks

# WHAT IS THE BIG-O TIME?

```
void f1(vector<int> input) {
    for (int ele : input) {
        // Process
    }
}
```

# WHAT IS THE BIG-O TIME?

```
void f1(vector<int> input) {
    for (int ele : input) {
        // Process
    }
}
```

$O(n)$ // It is proportional to the input size (linear)

# WHAT IS THE BIG-O TIME?

```
void multiply(int mat1[][N], int mat2[][N], int res[][N])
{
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            res[i][j] = 0;
            for (int k = 0; k < N; k++)
                res[i][j] += mat1[i][k] * mat2[k][j];
        }
    }
}
// From GeeksforGeeks
```

# WHAT IS THE BIG-O TIME?

```
void multiply(int mat1[][N], int mat2[][N], int res[][N])
{
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            res[i][j] = 0;
            for (int k = 0; k < N; k++)
                res[i][j] += mat1[i][k] * mat2[k][j];
        }
    }
}
```

$O(n^3)$ // It is now cubic (non-linear)

# WHAT IS THE BIG-O TIME?

```
void permute(int* a, int l, int r) {
    if (l == r) {
        for (int i = 0; i <= r; i++) {
            cout << a[i] << " ";
        }
        cout << endl;
    } else {
        for (int i = l; i <= r; i++) {
            swap(a[l], a[i]);
            permute(a, l + 1, r);
            swap(a[l], a[i]); // backtrack
        }
    }
}
// From GeeksforGeeks
```

# WHAT IS THE BIG-O TIME?

```
void permute(int* a, int l, int r) {
    if (l == r) {
        for (int i = 0; i <= r; i++) {
            cout << a[i] << " ";
        }
        cout << endl;
    } else {
        for (int i = l; i <= r; i++) {
            swap(a[l], a[i]);
            permute(a, l + 1, r);
            swap(a[l], a[i]); // backtrack
        }
    }
}
O(n!) // Factorial time, horrible!
```

# SIMPLIFY BIG-O TIME?
# NOT ON THIS EXAM, BUT USEFUL FOR 223

$$m(n) = 50n$$

$$t(n) = 50$$

$$f(n) = 3n^3 + 2n^2 + 5n + 1$$

$$g(n) = 2n^2 + 15n^2 + 200$$

# SIMPLIFY BIG-O TIME?
# NOT ON THIS EXAM, BUT USEFUL FOR 223

$$m(n) = 50n = O(n)$$

$$t(n) = 50$$

$$f(n) = 3n^3 + 2n^2 + 5n + 1$$

$$g(n) = 2n^2 + 15n^2 + 200$$

# SIMPLIFY BIG-O TIME?
# NOT ON THIS EXAM, BUT USEFUL FOR 223

$$m(n) = 50n = O(n)$$

$$t(n) = 50 = O(1)$$

$$f(n) = 3n^3 + 2n^2 + 5n + 1$$

$$g(n) = 2n^2 + 15n^2 + 200$$

# SIMPLIFY BIG-O TIME?
# NOT ON THIS EXAM, BUT USEFUL FOR 223

$$m(n) = 50n = O(n)$$

$$t(n) = 50 = O(1)$$

$$f(n) = 3n^3 + 2n^2 + 5n + 1 = O(n^3)$$

$$g(n) = 2n^2 + 15n^2 + 200$$

# SIMPLIFY BIG-O TIME?
# NOT ON THIS EXAM, BUT USEFUL FOR 223

$$m(n) = 50n = O(n)$$

$$t(n) = 50 = O(1)$$

$$f(n) = 3n^3 + 2n^2 + 5n + 1 = O(n^3)$$

$$g(n) = 2n^2 + 15n^2 + 200 = O(n^2)$$

# C++ CONCEPTS

GENERAL REVIEW

# WHAT DOES THIS CODE DO?

```
void op(struct Something *ptr) {
   while (ptr->next != NULL) {
     printf("%d  ", ptr->data);
        ptr = ptr->next;
   }
}
```

# WHAT ARE THE “BIG THREE” IN C++?

a) Setters, getters, constructors
b) Destructors, constructors, getters
c) Copy constructor, overloaded = operator, constructor
d) Destructor, copy constructor, constructor
e) Constructors, setters, destructors

# WHAT ARE THE “BIG THREE” IN C++?

a) Setters, getters, constructors

b) Destructors, constructors, getters

c) Copy constructor, overloaded = operator, constructor

**d) Destructor, copy constructor, constructor**

e) Constructors, setters, destructors

# ANSWER THE QUESTION

```
class Test final {
    int x;
public:
    Test();
    void a();
    void c() const;
    const int b();
    const int d() const;
}
```

State whether `**x = 100;**` is valid:

a) a

b) b

c) c

d) d

e) Test()

# ANSWER THE QUESTION

```
class Test final {
    int x;
public:
    Test();
    void a();
    void c() const;
    const int b();
    const int d() const;
}
```

State whether `**x = 100;**` is valid:

a) a - VALID

b) b

c) c

d) d

e) Test()

# ANSWER THE QUESTION

```
class Test final {
    int x;
public:
    Test();
    void a();
    void c() const;
    const int b();
    const int d() const;
}
```

State whether `**x = 100;**` is valid:

a) a - VALID

b) b - VALID

c) c

d) d

e) Test()

# ANSWER THE QUESTION

```
class Test final {
    int x;
public:
    Test();
    void a();
    void c() const;
    const int b();
    const int d() const;
}
```

State whether `**x = 100;**` is valid:

a) a - VALID

b) b - VALID

c) c - INVLAID

d) d

e) Test()

# ANSWER THE QUESTION

```
class Test final {
    int x;
public:
    Test();
    void a();
    void c() const;
    const int b();
    const int d() const;
}
```

State whether `**x = 100;**` is valid:

a) a - VALID

b) b - VALID

c) c - INVLAID

d) d - INVLAID

e) Test()

# ANSWER THE QUESTION

```
class Test final {
    int x;
public:
    Test();
    void a();
    void c() const;
    const int b();
    const int d() const;
}
```

State whether `**x = 100;**` is valid:

a) a - VALID

b) b - VALID

c) c - INVLAID

d) d - INVLAID

e) Test() - VALID

# ANSWER THE QUESTION (ABOUT CONST)

```
class Test final {
    int x;
    const int y;
public:
    Test();
    void a();
    void c() const;
    const int b();
    const int d() const;
}
```

State whether `y = 100;` is valid:

a) a

b) b

c) c

d) d

e) Test()

# ANSWER THE QUESTION (ABOUT CONST)

```
class Test final {
    int x;
    const int y;
public:
    Test(int newY) : y(newY) {}
    void a();
    void c() const;
    const int b();
    const int d() const;
}
```

State whether `y = 100;` is valid:

a) a - INVLAID

b) b - INVLAID

c) c - INVLAID

d) d – INVLAID

e) Test() - INVALID

# SUPPOSE PTR IS A NODE IN A LINKED LIST

```
Line 1:
    ptr++;
Line 2:
    ptr += 1;
Line 3:
    ptr = ptr->next;
Line 4:
    ptr += ptr->next;
Assumed Link Addrs: [3, 9, 31, 42]
^^^Starting Value is 3^^^
```

| Line # | Will it compile? | Value of ptr? |
|---|---|---|
| 1 | ??? | ??? |
| 2 | ??? | ??? |
| 3 | ??? | ??? |
| 4 | ??? | ??? |

# SUPPOSE PTR IS A NODE IN A LINKED LIST

```
Line 1:
    ptr++;
Line 2:
    ptr += 1;
Line 3:
    ptr = ptr->next;
Line 4:
    ptr += ptr->next;
Assumed Link Addrs: [3, 9, 31, 42]
^^^Starting Value is 3^^^
```

| Line # | Will it compile? | Value of ptr? |
|---|---|---|
| 1 | YES | 3 |
| 2 | ??? | ??? |
| 3 | ??? | ??? |
| 4 | ??? | ??? |

# SUPPOSE PTR IS A NODE IN A LINKED LIST

```
Line 1:
    ptr++;
Line 2:
    ptr += 1;
Line 3:
    ptr = ptr->next;
Line 4:
    ptr += ptr->next;
Assumed Link Addrs: [3, 9, 31, 42]
^^^Starting Value is 3^^^
```

| Line # | Will it compile? | Value of ptr? |
|---|---|---|
| 1 | YES | 3 |
| 2 | YES | 11 |
| 3 | ??? | ??? |
| 4 | ??? | ??? |

# SUPPOSE PTR IS A NODE IN A LINKED LIST

```
Line 1:
    ptr++;
Line 2:
    ptr += 1;
Line 3:
    ptr = ptr->next;
Line 4:
    ptr += ptr->next;
Assumed Link Addrs: [3, 9, 31, 42]
^^^Starting Value is 3^^^
```

| Line # | Will it compile? | Value of ptr? |
|---|---|---|
| 1 | YES | 3 |
| 2 | YES | 11 |
| 3 | YES | 9 |
| 4 | ??? | ??? |

# SUPPOSE PTR IS A NODE IN A LINKED LIST

```
Line 1:
    ptr++;
Line 2:
    ptr += 1;
Line 3:
    ptr = ptr->next;
Line 4:
    ptr += ptr->next;
Assumed Link Addrs: [3, 9, 31, 42]
^^^Starting Value is 3^^^
```

| Line # | Will it compile? | Value of ptr? |
|---|---|---|
| 1 | YES | 3 |
| 2 | YES | 11 |
| 3 | YES | 9 |
| 4 | YES | 12 |

# INHERITANCE

# INHERITANCE CONCEPTS

- Define a “is-a” relationship
- Define a “has-a” relationship
- What is a base and derived class? Subclass? Superclass?
- Define single, multiple, hierarchal, multilevel, and hybrid inheritance
- What is the diamond problem?

# INHERITANCE CONCEPTS

- Define a “is-a” relationship (Inheritance)
  - Apple is a Fruit. This shows that Apple inherits from Fruit.
- Define a “has-a” relationship
- What is a base and derived class? Subclass? Superclass?
- Define single, multiple, hierarchal, multilevel, and hybrid inheritance
- What is the diamond problem?

# INHERITANCE CONCEPTS

- Define a “is-a” relationship (Inheritance)
  - Apple is a Fruit. This shows that Apple inherits from Fruit.
- Define a “has-a” relationship (Composition)
  - Apple has a Seed. This shows that Apple contains a Seed (that is, Seed is a member within Apple)
- What is a base and derived class? Subclass? Superclass?
- Define single, multiple, hierarchal, multilevel, and hybrid inheritance
- What is the diamond problem?

# INHERITANCE CONCEPTS

- Define a “is-a” relationship (Inheritance)
  - Apple is a Fruit. This shows that Apple inherits from Fruit.
- Define a “has-a” relationship (Composition)
  - Apple has a Seed. This shows that Apple contains a Seed (that is, Seed is a member within Apple)
- What is a base and derived class? Subclass? Superclass?
  - (Using example above) base = Fruit, derived = Apple
  - A subclass is the derived class (Apple); A superclass is the base class (Fruit). Both used in Java
- Define single, multiple, hierarchal, multilevel, and hybrid inheritance
- What is the diamond problem?

# INHERITANCE CONCEPTS

- Define a “is-a” relationship (Inheritance)
  - Apple is a Fruit. This shows that Apple inherits from Fruit.
- Define a “has-a” relationship (Composition)
  - Apple has a Seed. This shows that Apple contains a Seed (that is, Seed is a member within Apple)
- What is a base and derived class? Subclass? Superclass?
  - (Using example above) base = Fruit, derived = Apple
  - A subclass is the derived class (Apple); A superclass is the base class (Fruit). Both used in Java
- Define single, multiple, hierarchal, multilevel, and hybrid inheritance
  - Single = Inherit from 1 class | Multiple = Inherit from 2+ classes
  - Hierarchal = Multiple subclasses inherit from a single base class | Hybrid = Multiple types
- What is the diamond problem?

# INHERITANCE CONCEPTS

- Define a “is-a” relationship (Inheritance)
  - Apple is a Fruit. This shows that Apple inherits from Fruit.
- Define a “has-a” relationship (Composition)
  - Apple has a Seed. This shows that Apple contains a Seed (that is, Seed is a member within Apple)
- What is a base and derived class? Subclass? Superclass?
  - (Using example above) base = Fruit, derived = Apple
  - A subclass is the derived class (Apple); A superclass is the base class (Fruit). Both used in Java
- Define single, multiple, hierarchal, multilevel, and hybrid inheritance
  - Single = Inherit from 1 class | Multiple = Inherit from 2+ classes
  - Hierarchal = Multiple subclasses inherit from a single base class | Hybrid = Multiple types
- What is the diamond problem?
  - When you inherit from two classes which both inherit from a common base class

# DATA STRUCTURES

TREE, QUEUE, LIST, STACK

# QUEUES

- We have a front and rear pointer. Identify which pointer(s) change per enqueue/dequeue:

```
Queue q;
q.enqueue(1);
q.enqueue(2);
q.dequeue();
q.dequeue();
```

# QUEUES

- We have a front and rear pointer. Identify which pointer(s) change per enqueue/dequeue:

```
Queue q;
q.enqueue(1); // EMPTY // Front & Rear
q.enqueue(2);
q.dequeue();
q.dequeue();
```

# QUEUES

- We have a front and rear pointer. Identify which pointer(s) change per enqueue/dequeue:

```
Queue q;
q.enqueue(1); // EMPTY // Front & Rear
q.enqueue(2); // NON-EMPTY // Rear
q.dequeue();
q.dequeue();
```

# QUEUES

- We have a front and rear pointer. Identify which pointer(s) change per enqueue/dequeue:

```
Queue q;
q.enqueue(1); // EMPTY // Front & Rear
q.enqueue(2); // NON-EMPTY // Rear
q.dequeue(); // NON-EMPTY // Front
q.dequeue();
```

# QUEUES

- We have a front and rear pointer. Identify which pointer(s) change per enqueue/dequeue:

```
Queue q;
q.enqueue(1); // EMPTY // Front & Rear
q.enqueue(2); // NON-EMPTY // Rear
q.dequeue(); // NON-EMPTY // Front
q.dequeue(); // NON-EMPTY // Front & Rear (Now NULL)
```

# SUPPOSE I HAVE A QUEUE I WANT TO REVERSE. WHICH OF THE FOLLOWING DO I WANT TO USE?

```
Function reverse
    input: queue Q
    output: reversed queue Q
    stack <- {}
    While (Q is not Empty)
        Q.dequeue(item)
        stack.push(item)
    Endwhile
    While (stack is not Empty)
        stack.pop(item)
        Q.enqueue(item)
    Endwhile
    Return Q
endfunction
```

```
Function reverse
    input: queue Q
    output: reversed queue Q

    x <- 0
    item <- 0
    max <- Q.size()

    While (x < max)
        Q.dequeue(item)
        Q.enqueue(item)
    Endwhile
    Return Q
endfunction
```

# IF WE WANT TO REVERSE SOMETHING, USE A STACK

- Stacks follow FILO
- If we put something in first, we will get it last
- If we want to reverse something, we want to put the element at the beginning at the end

# WHAT DOES THE FOLLOWING CODE ACCOMPLISH? (LINE-BY-LINE ANALYSIS)

```
ptr = ptr->last;
ptr = ptr->next;
ptr = ptr->link;
ptr = ptr->prev;
```

# WHAT DOES THE FOLLOWING CODE ACCOMPLISH? (LINE-BY-LINE ANALYSIS)

```
ptr = ptr->last; // Go to last item in list
ptr = ptr->next;
ptr = ptr->link;
ptr = ptr->prev;
```

# WHAT DOES THE FOLLOWING CODE ACCOMPLISH? (LINE-BY-LINE ANALYSIS)

```
ptr = ptr->last; // Go to last item in list
ptr = ptr->next; // Go to next item in list
ptr = ptr->link;
ptr = ptr->prev;
```

# WHAT DOES THE FOLLOWING CODE ACCOMPLISH? (LINE-BY-LINE ANALYSIS)

```
ptr = ptr->last; // Go to last item in list
ptr = ptr->next; // Go to next item in list
ptr = ptr->link; // Go to next item in list
ptr = ptr->prev;
```

# WHAT DOES THE FOLLOWING CODE ACCOMPLISH? (LINE-BY-LINE ANALYSIS)

```
ptr = ptr->last; // Go to last item in list
ptr = ptr->next; // Go to next item in list
ptr = ptr->link; // Go to next item in list
ptr = ptr->prev; // Go to last item in list
```

# WHICH OF THE FOLLOWING DO WE APPLY TO A TREE?

- Inheritance
- Encapsulation
- Abstraction
- Polymorphism

# WHICH OF THE FOLLOWING DO WE APPLY TO A TREE?

- Inheritance
  - Sometimes, but not necessarily with every situation. (Example: PA 8)
- Encapsulation
- Abstraction
- Polymorphism

# WHICH OF THE FOLLOWING DO WE APPLY TO A TREE?

- Inheritance
  - Sometimes, but not necessarily with every situation. (Example: PA 8)
- Encapsulation
  - Can show up in many ways
- Abstraction
- Polymorphism

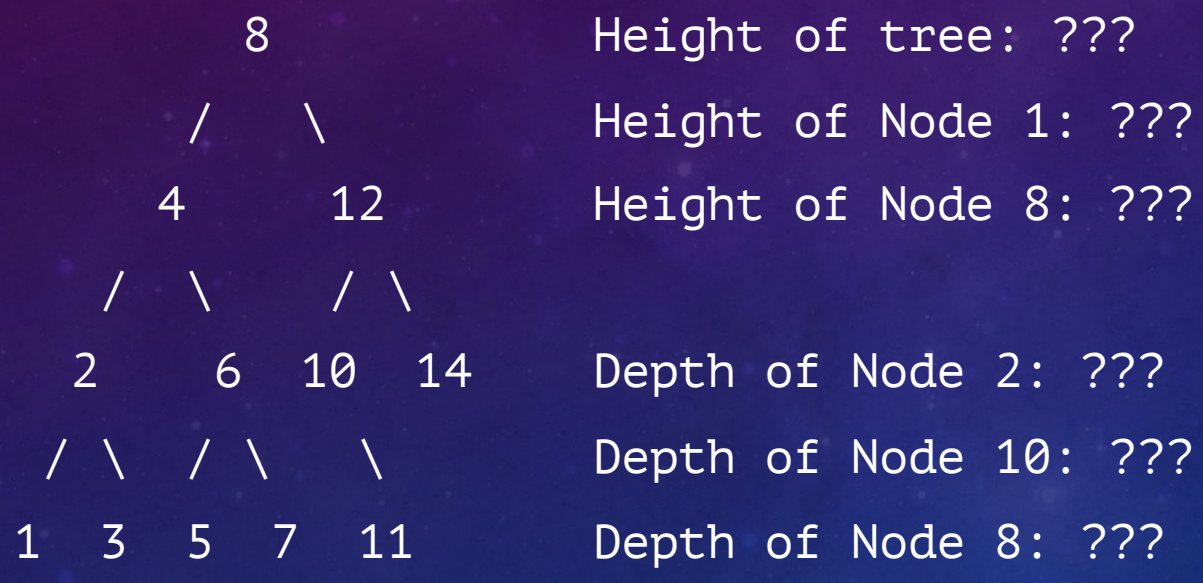
# WHICH OF THE FOLLOWING DO WE APPLY TO A TREE?

- Inheritance
  - Sometimes, but not necessarily with every situation. (Example: PA 8)
- Encapsulation
  - Can show up in many ways
- Abstraction
  - When we have a public interface which accepts data, and the private version has more parameters (i.e., root)
- Polymorphism

# WHICH OF THE FOLLOWING DO WE APPLY TO A TREE?

- Inheritance
  - Sometimes, but not necessarily with every situation (i.e., PA 8)
- Encapsulation
  - Can show up in many ways
- Abstraction
  - When we have a public interface which accepts data, and the private version has more parameters (i.e., root)
- Polymorphism
  - Probably not, but if you are using inheritance, this should be a given

# ANALYZE THE TREE




Height of tree: ???

Height of Node 1: ???

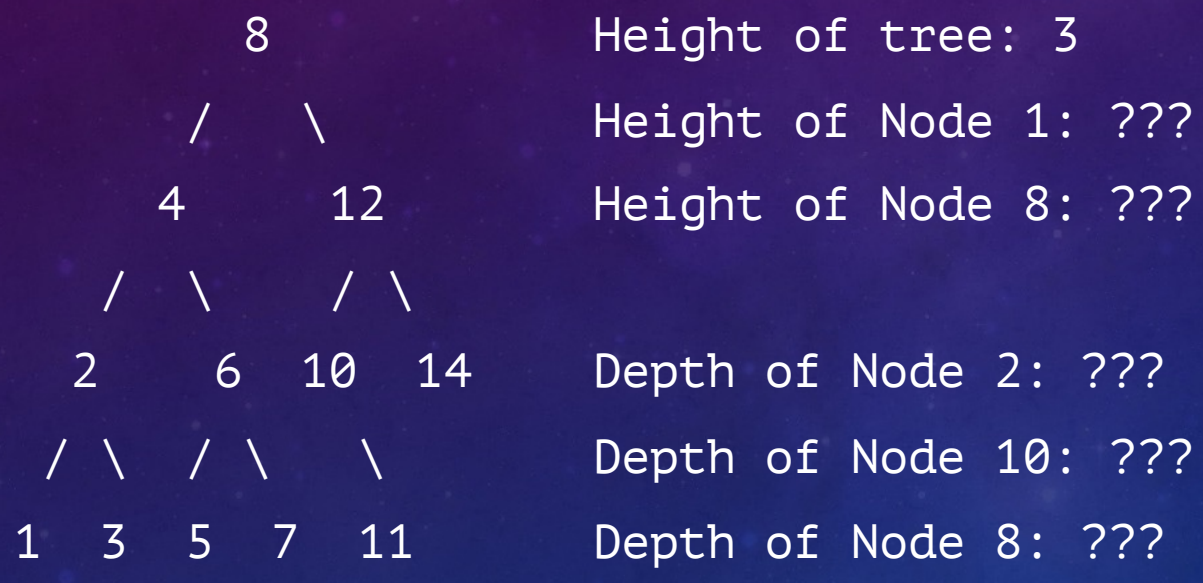
Height of Node 8: ???

Depth of Node 2: ???

Depth of Node 10: ???

Depth of Node 8: ???

# ANALYZE THE TREE




Height of tree: 3

Height of Node 1: ???

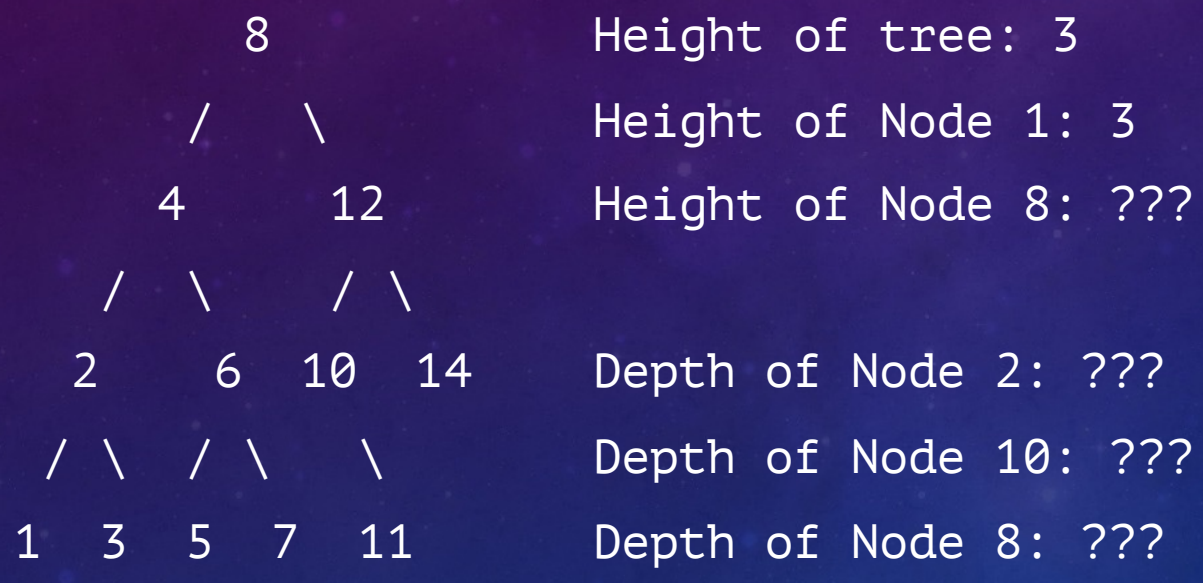
Height of Node 8: ???

Depth of Node 2: ???

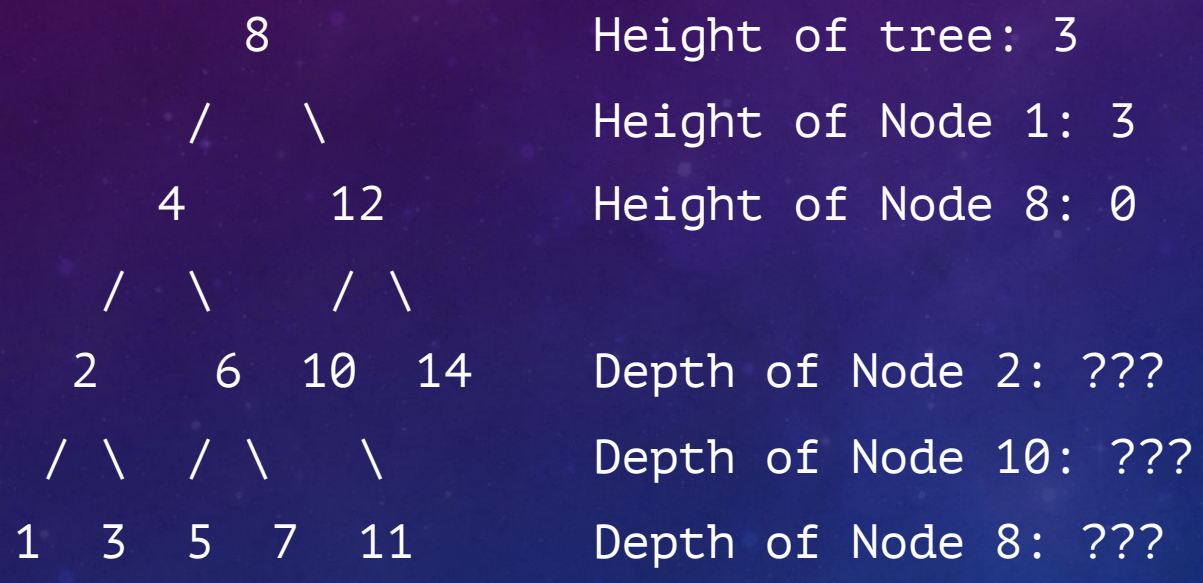
Depth of Node 10: ???

Depth of Node 8: ???

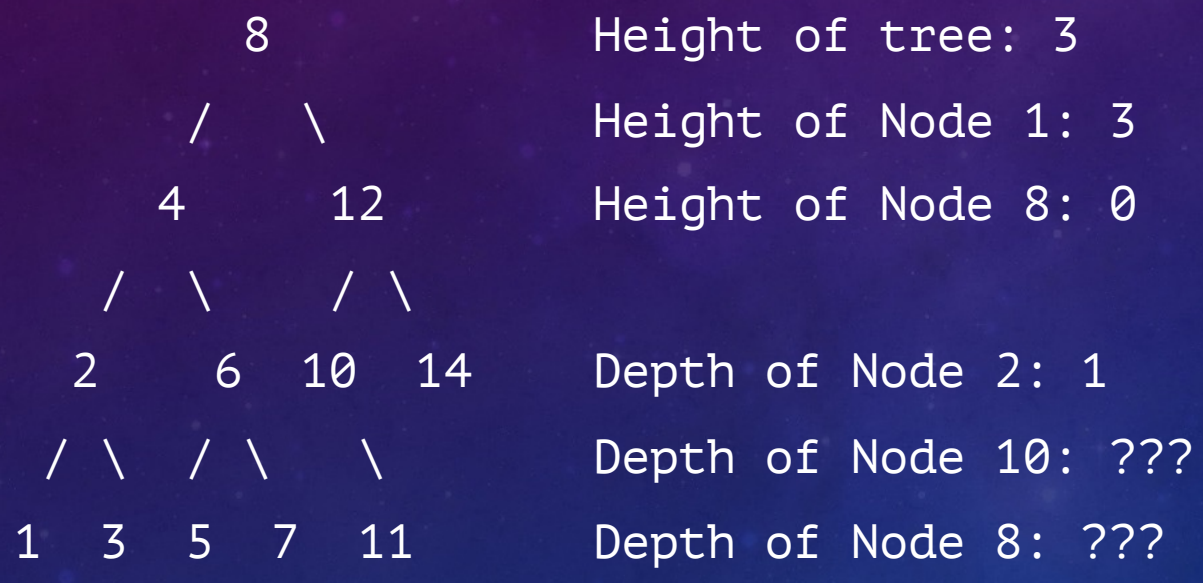
# ANALYZE THE TREE



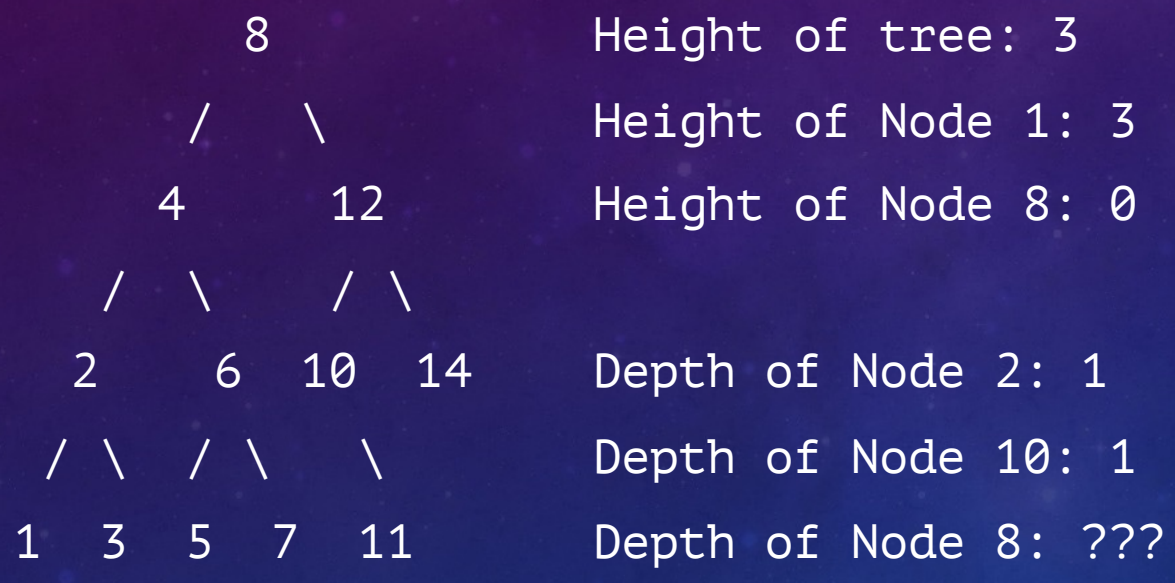
# ANALYZE THE TREE



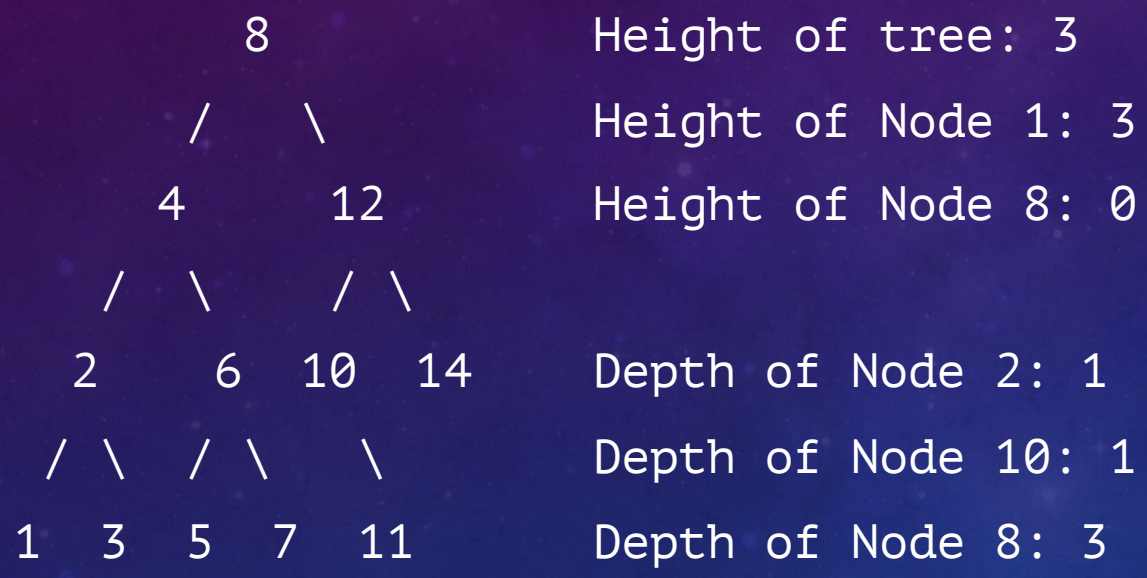
# ANALYZE THE TREE



# ANALYZE THE TREE



# ANALYZE THE TREE




Height of tree: 3

Height of Node 1: 3

Height of Node 8: 0

Depth of Node 2: 1

Depth of Node 10: 1

Depth of Node 8: 3

# ANALYZE THE TREE


Pre-order: ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??

In-order: ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??

Post-order: ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??

# ANALYZE THE TREE


Pre-order: 08 04 02 01 03 06 05 07 12 10 11 14

In-order: ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??

Post-order: ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??

# ANALYZE THE TREE


Pre-order: 08 04 02 01 03 06 05 07 12 10 11 14

In-order: 01 02 03 04 05 06 07 08 10 11 12 14

Post-order: ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??

# ANALYZE THE TREE


Pre-order: 08 04 02 01 03 06 05 07 12 10 11 14

In-order: 01 02 03 04 05 06 07 08 10 11 12 14

Post-order: 01 03 02 05 07 06 04 11 10 14 12 08

# ANALYZE A TREE

Suppose Node N has height 20. Node F also has height 20.

What do we know about N and F?

Suppose the height of some tree is 10.

How many nodes can have height 0?

Suppose the height of some (binary) tree is 2.

What is the max number of nodes this tree can have?

# ANALYZE A TREE

Suppose Node N has height 20. Node F also has height 20.

What do we know about N and F?

**We know N and F must be on the same level.**

Suppose the height of some tree is 10.

How many nodes can have height 0?

Suppose the height of some (binary) tree is 2.

What is the max number of nodes this tree can have?

# ANALYZE A TREE

Suppose Node N has height 20. Node F also has height 20.

What do we know about N and F?

**We know N and F must be on the same level.**

Suppose the height of some tree is 10.

How many nodes can have height 0?

**1 - only the root node can be of height 0.**

Suppose the height of some (binary) tree is 2.

What is the max number of nodes this tree can have?

# ANALYZE A TREE

Suppose Node N has height 20. Node F also has height 20.

What do we know about N and F?

**We know N and F must be on the same level.**

Suppose the height of some tree is 10.

How many nodes can have height 0?

**1 - only the root node can be of height 0.**

Suppose the height of some (binary) tree is 2.

What is the max number of nodes this tree can have?

**Level 0**: $2^0 = 1$; **Level 1**: $2^1 = 2$; **Level 2**: $2^2 = 4$

=> $1 + 2 + 4 = 7$ Nodes

# IMPORTANT NOTE ON USING AI (WRT TREES)

- If you use AI and deal with trees, AI is rather bad at answering questions
- Avoid using for helping you learn; more than likely you will learn wrong or become more confused
- For example, I told it to generate a tree of height 4 for these slides and it produced height 3

- AI can be a good companion, but you need to be able to fix any errors it makes

# EXCEPTIONS

# NOTE

- When I say ”specialize” I mean to become more specific in the inheritance hierarchy (go down).
- When I say “generalize” I mean we become more generic in the inheritance hierarchy (go up).
- Example: If B inherits from A, A is specialized from B.

# WHAT ARE THE TWO CLASSES WHICH SPECIALIZE STD::EXCEPTION?

# WHAT ARE THE TWO CLASSES WHICH SPECIALIZE STD::EXCEPTION?

logic_error

# WHAT ARE THE TWO CLASSES WHICH SPECIALIZE STD::EXCEPTION?

- logic_error
- runtime_error

# WHAT ARE THE TWO CLASSES WHICH SPECIALIZE STD::EXCEPTION?

- logic_error
  - domain_error – suppose f(x) only accepts positive values, passing in a negative value should throw domain_error
  - invlaid_argument – attempt to pass in an argument of the wrong type (which passes complication)
  - length_error – attempt to set myVector to a size greater than .maxSize()
  - out_of_range – ex: myVector.at(10) when it only has 5 elements
  - future_error – look into std::future if you are interested
- runtime_error
  - range_error – report range errors in internal computations. [cplusplus.com]
  - overflow_error – a value exceeds the maximal value (never thrown by the STL)
  - underflow_error – a value passes the minimal value (never thrown by the STL)
  - system_error – low-level or OS-related issue

# CEASAR CIPHER

- Very simple shift encoder
- Shift in the alphabet by a given amount, the default is 3 (shown below)
- To do this by hand, simply use two lines:
  1. Write the alphabet: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
  2. Write the alphabet offset: X Y Z A B C D E F G H I J K L M N O P Q R S T U V W
- To encode, go down
- To decode, go up (to brute force this, try all shifts (25))