Data Structure and Applications (DI03032031)

Complete Lecture Deck – All 30 Lectures

Milav Dabgar

Lecturer, GP Palanpur

July 22, 2026

Course Agenda I

1. Lecture 1.1: Data Structure Basic Concepts & Analysis
2. Lecture 1.2: OOP Concepts, Classes & Objects
3. Lecture 1.3: Methods & Abstract Data Types
4. Lecture 1.4: Recursion Basics
5. Lecture 2.1: 2.1 Overview of Stack, Operations on Stack
6. Lecture 2.2: 2.2 Implementation of Stack using List
7. Lecture 2.3: Stack Applications: Expressions

Course Agenda II

Course Agenda III

Course Agenda IV

Course Agenda V

Lecture 1.1: Data Structure Basic Concepts & Analysis

Complete Lecture Deck – All 30 Lectures

Milav Dabgar

Lecturer, GP Palanpur

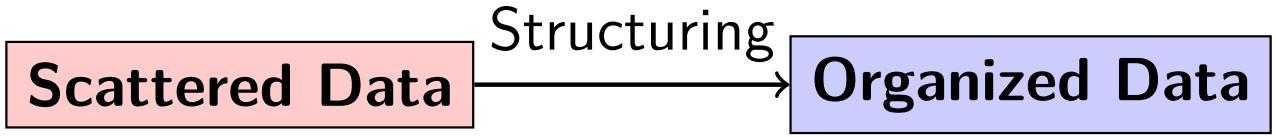
July 22, 2026

Course Overview & Today's Agenda

- Course covers core Data Structures: Arrays, Stacks, Queues, Linked Lists, Trees, and Graphs
- Today's Topics:
  1. What is a Data Structure?
  2. Types of Data Structures
  3. Introduction to Algorithm Analysis
  4. Time and Space Complexity
  5. Asymptotic Notations & Big 'O'

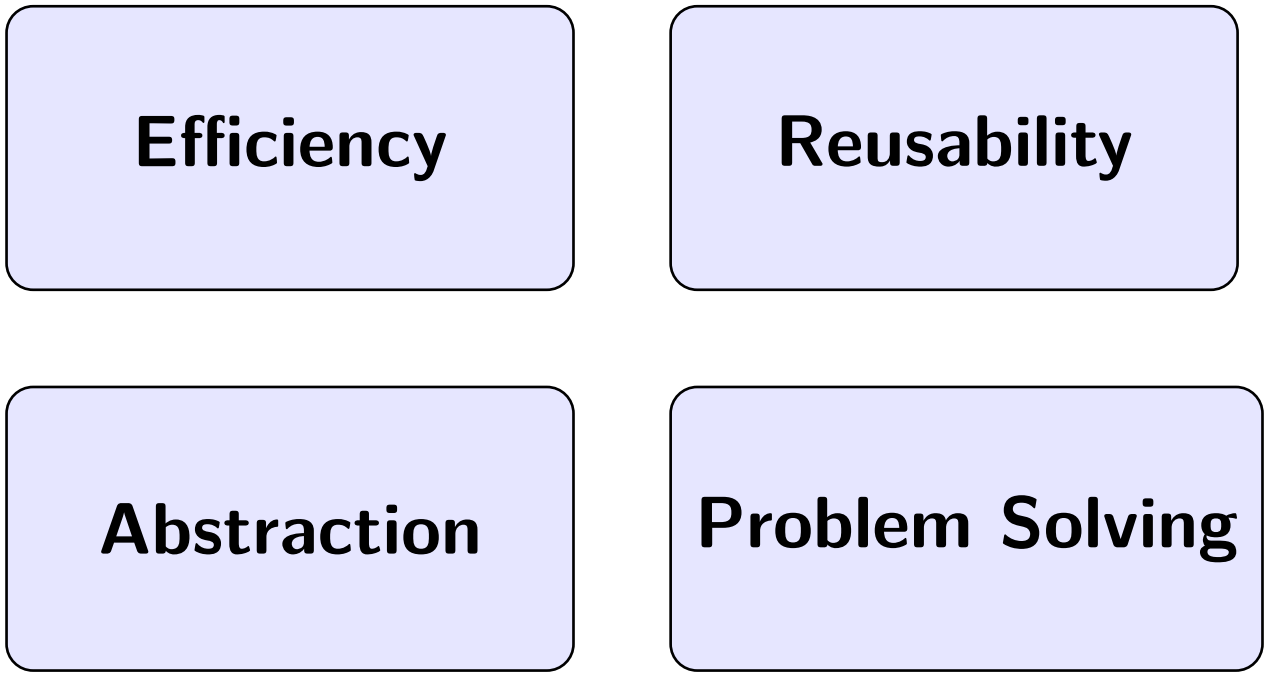
What is a Data Structure?

- A Data Structure is a specialized format for organizing, processing, retrieving and storing data.
- It provides a way to manage large amounts of data efficiently.
- While a data type defines the kind of value, a data structure models a collection of values.
- Example in Python: `my_list = [10, 20, 30]` is a list data structure.



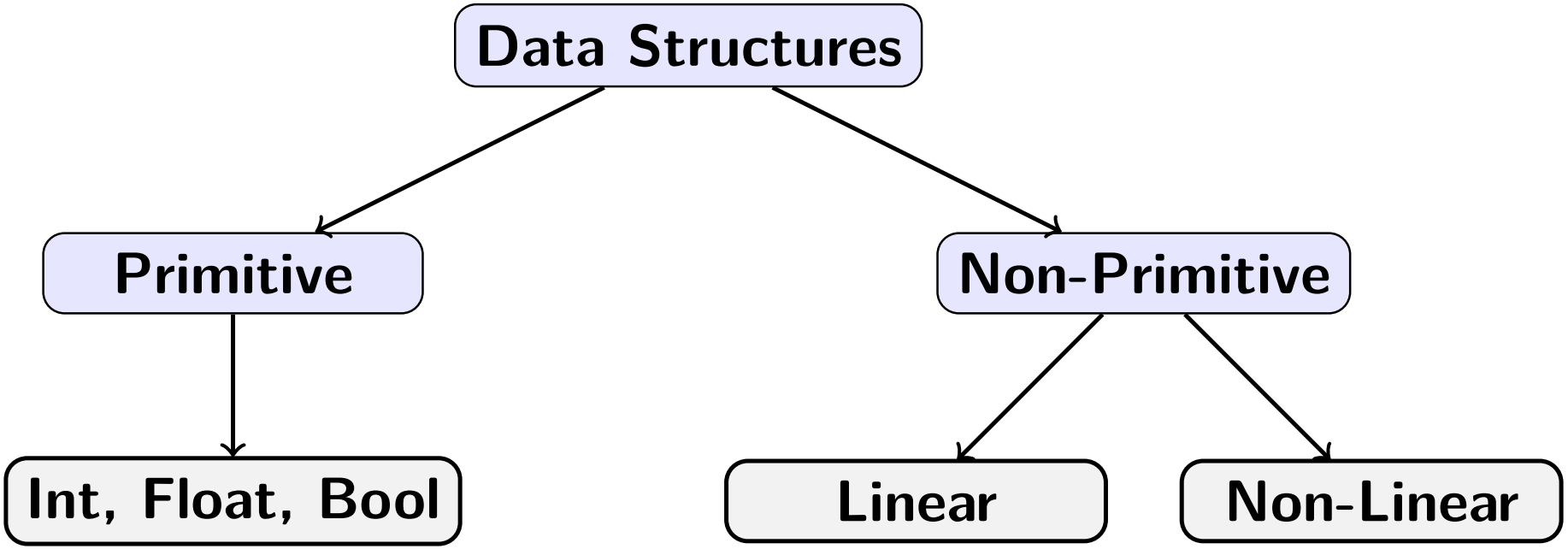
Why Do We Need Data Structures?

- **Efficiency**: Optimized storage and retrieval of data.
- **Reusability**: Standard data structures can be implemented once and used multiple times.
- **Abstraction**: Hides the complex implementation details from the user.
- **Problem Solving**: Different problems require different tools.



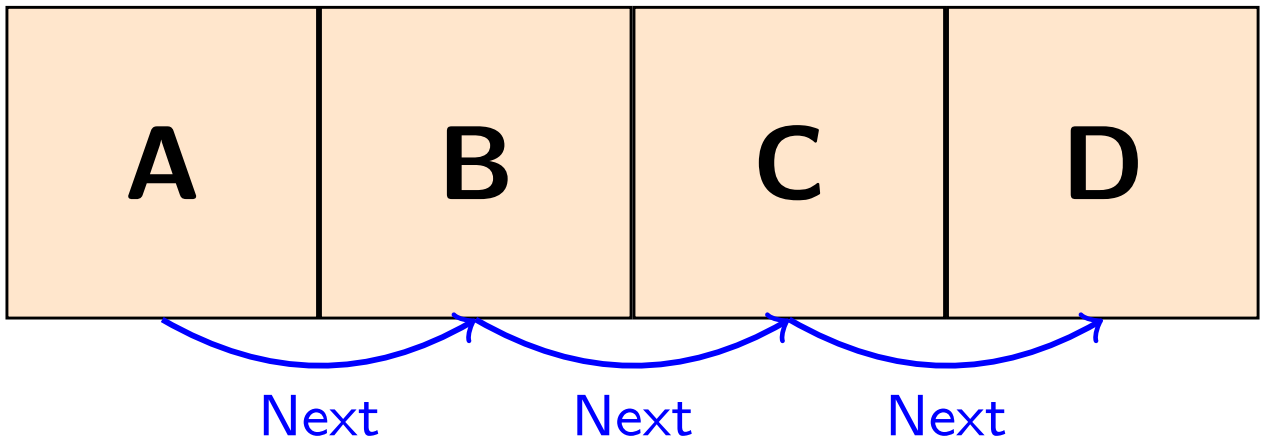
Types of Data Structures

- Data Structures can be broadly classified into:
  1. **Primitive**: Built-in basic types (Integer, Float, Character, Boolean)
  2. **Non-Primitive**: Derived from primitive types.
- Non-Primitive is further divided into Linear and Non-Linear.



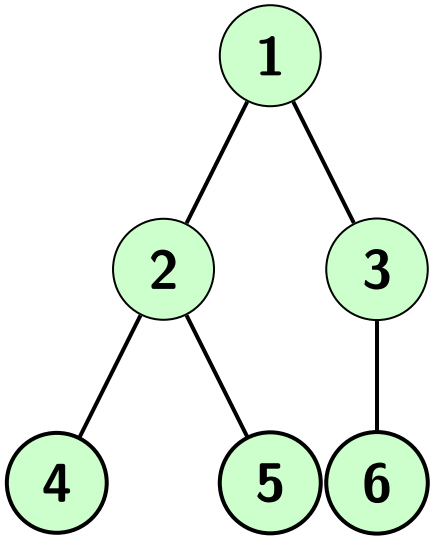
Linear Data Structures

- Elements are arranged in a sequential or linear order.
- Each element is attached to its previous and next adjacent elements.
- Examples: Arrays, Stacks, Queues, Linked Lists.

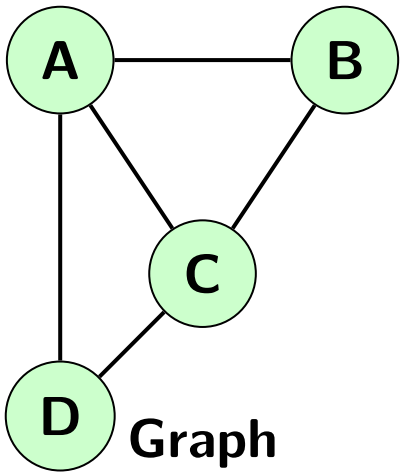


Non-Linear Data Structures

- Elements are not arranged in sequence.
- An element can be connected to multiple other elements, forming complex relationships.
- Examples: Trees, Graphs.



Tree

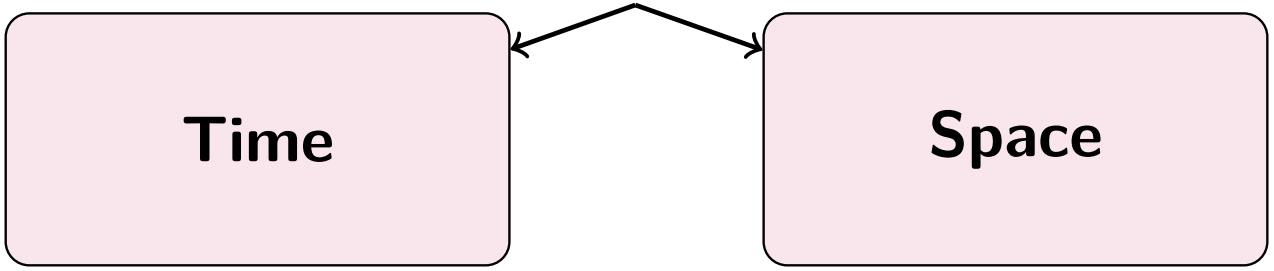


Graph

Introduction to Algorithm Analysis

- An algorithm is a step-by-step procedure to solve a problem.
- Analysis of algorithms is the process of evaluating their efficiency.
- Why analyze? To compare multiple solutions for the same problem.
- We evaluate algorithms based on two main criteria: **Time** and **Space**.

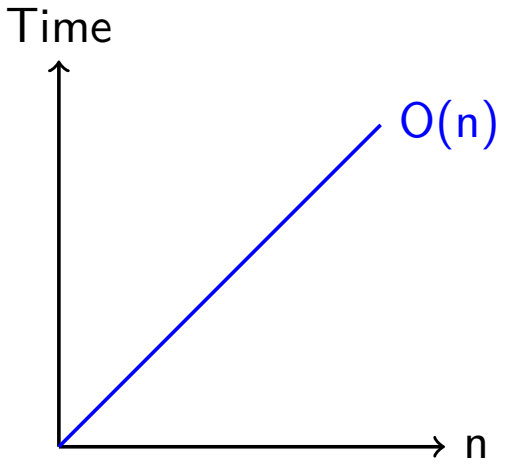
Efficiency



Time Complexity

- **Definition**: The amount of time an algorithm takes to run as a function of the length of the input (n).
- It is NOT measured in seconds, but in the **number of basic operations** performed.
- Example: A loop running 'n' times has a time complexity proportional to 'n'.

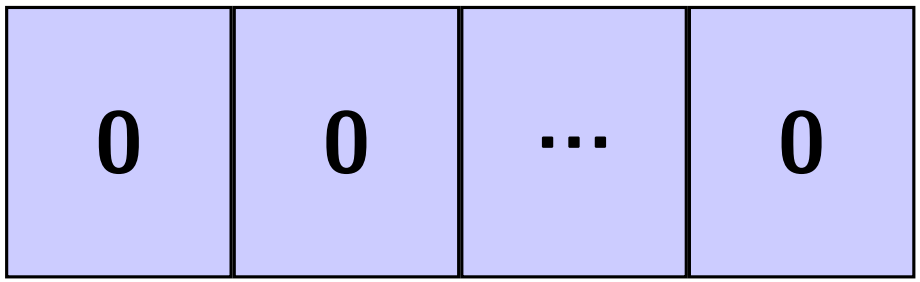
```
for i in range(n):
    print(i)  # Runs n times
```



Space Complexity

- **Definition**: The amount of memory space required by the algorithm during its execution.
- Includes both the space for input data and the extra space (auxiliary space) used by the algorithm.
- Example: Creating a new list of size 'n' to store intermediate results increases space complexity.

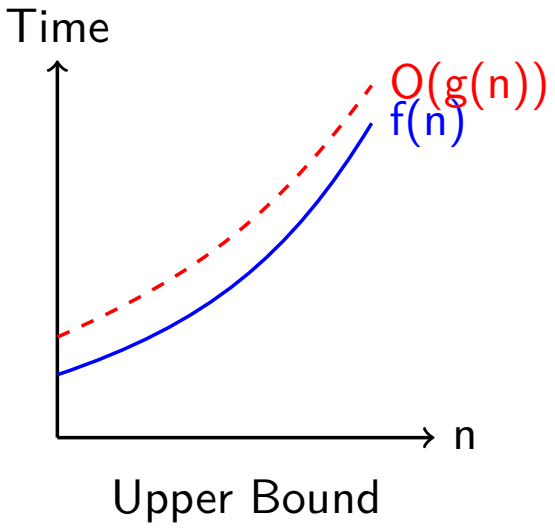
```
temp_list = [0] * n  # Requires O(n) space
```



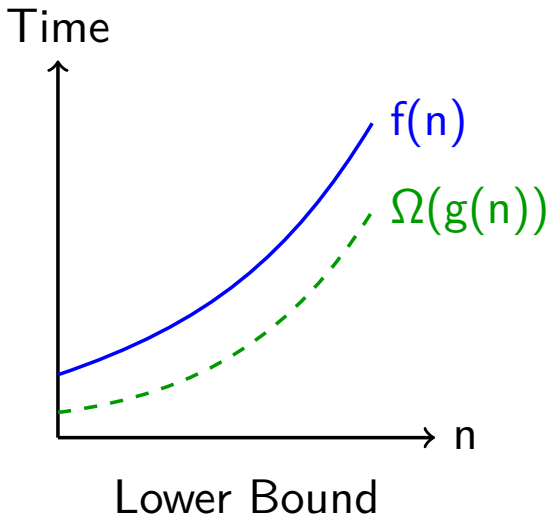
Memory blocks (size n)

Asymptotic Notations

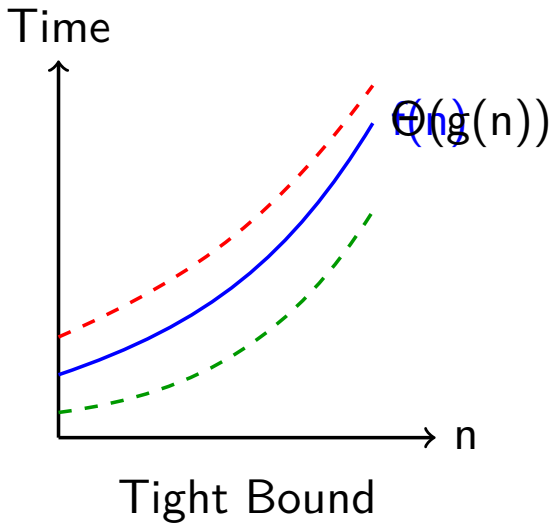
- Mathematical tools used to describe the running time of an algorithm.
- They define the algorithm's performance as the input size (n) grows towards infinity.
- Three main notations:
  1. **Big 'O' (O)** - Upper bound
  2. **Omega (Ω)** - Lower bound
  3. **Theta (Θ)** - Exact/Tight bound



Upper Bound



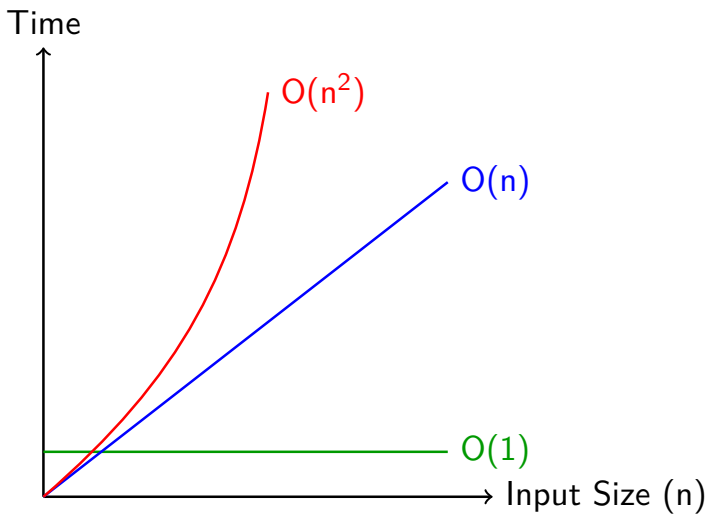
Lower Bound



Tight Bound

Big 'O' Notation (O)

- Describes the worst-case scenario or the upper bound of time complexity.
- It tells us the maximum time an algorithm will take.
- Common Complexities:
  - **O(1)**: Constant Time
  - **O(n)**: Linear Time
  - **O(n^2)**: Quadratic Time



Big 'O' Code Examples (Python)

- **O(1) - Constant:**

```
def get_first(lst):
    return lst[0]
```

- **O(n) - Linear:**

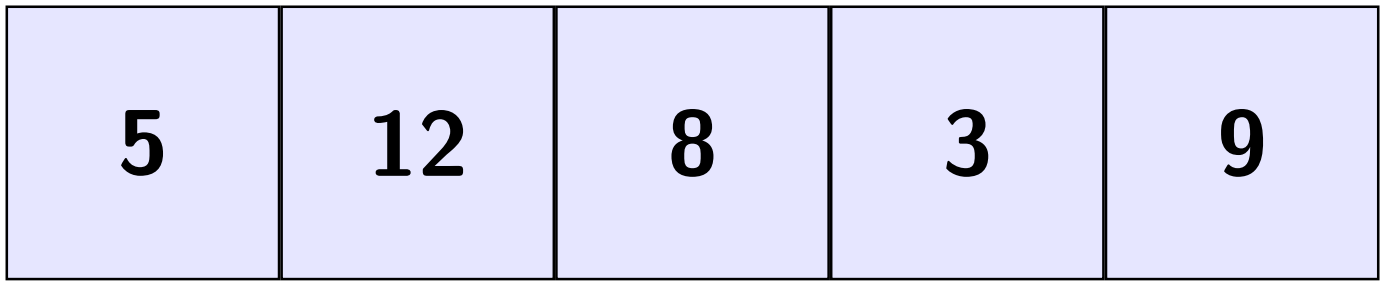
```
def print_all(lst):
    for item in lst:
        print(item)
```

- **O(n^2) - Quadratic:**

```
def print_pairs(lst):
    for i in lst:
        for j in lst:
            print(i, j)
```

Cases of Time Complexity

- When analyzing an algorithm, its performance can vary based on the specific input provided.
- We usually analyze algorithms for three cases:
  1. Best Case
  2. Average Case
  3. Worst Case
- Example: Searching for an item in a list linearly.



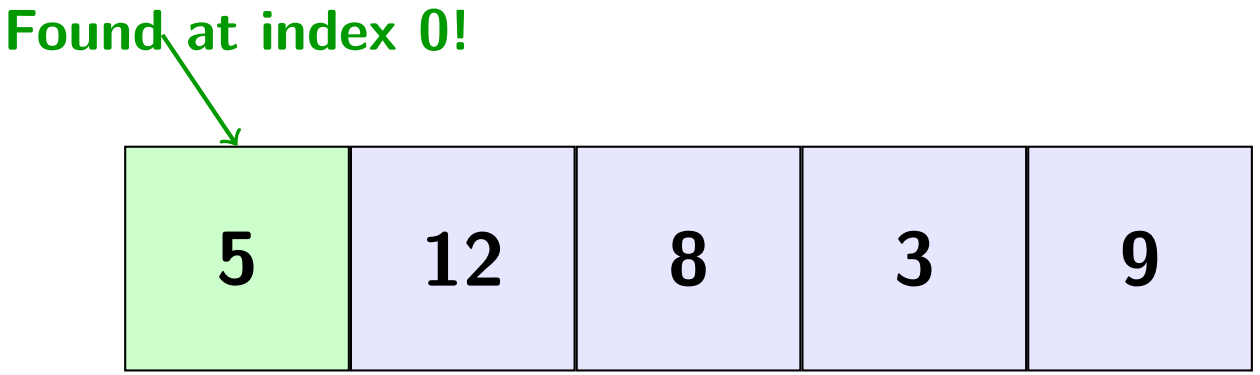
Target: 5 (Best)

Target: 8 (Average)

Target: 9 (Worst)

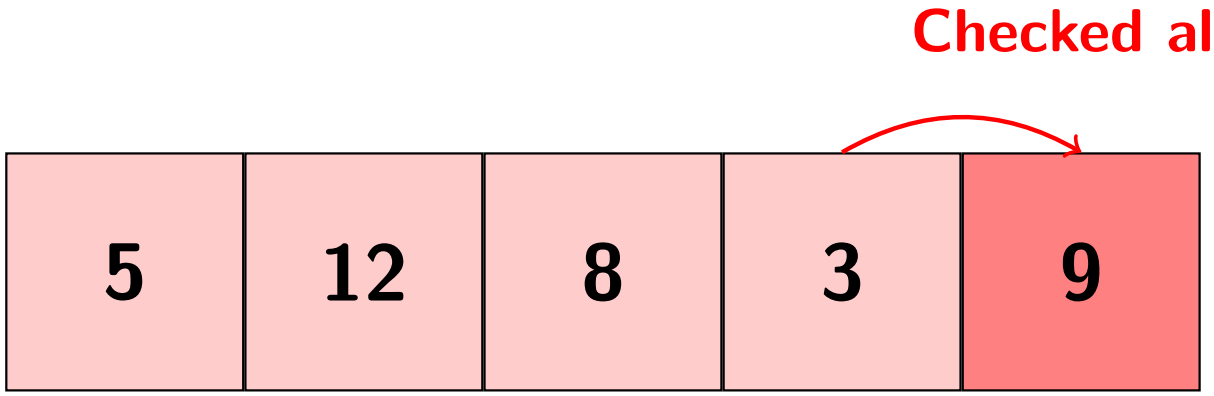
Best Case Time Complexity

- The minimum time required for program execution.
- Occurs when the input provided is the most favorable.
- Example: In Linear Search, if the element we are looking for is at the very first position.
- Complexity: O(1) for this specific scenario.



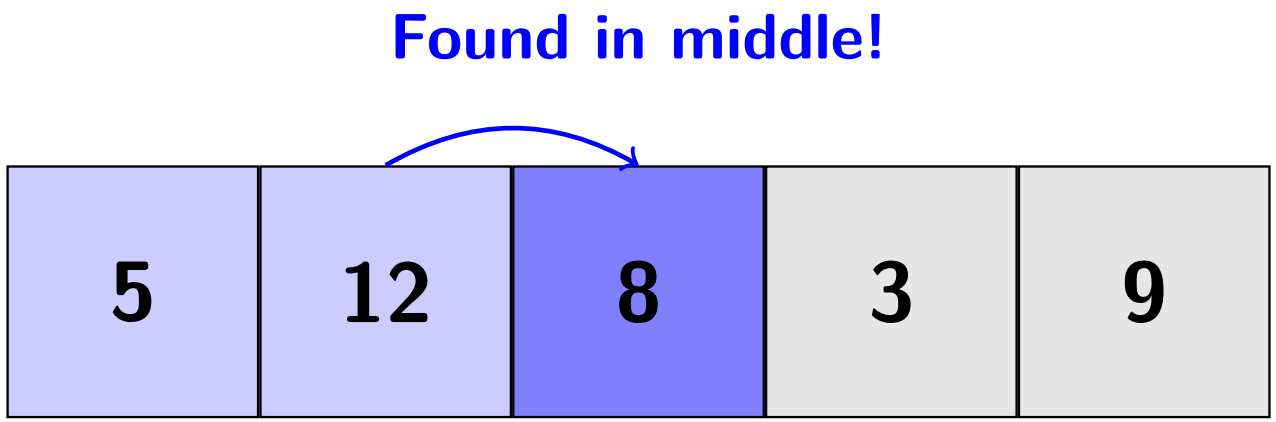
Worst Case Time Complexity

- The maximum time required for program execution.
- Occurs when the input provided is the least favorable.
- Example: In Linear Search, if the element is at the very end of the list.
- Complexity: O(n) because we must check every single element.
- We mostly focus on the Worst Case in Computer Science.



Average Case Time Complexity

- The expected time required on average across all possible inputs.
- Calculated by taking the sum of running times for all inputs and dividing by the number of inputs.
- Example: In Linear Search, on average, we might find the element halfway through the list.
- Complexity is often proportional to the worst case.



Quick Check: Test Your Understanding

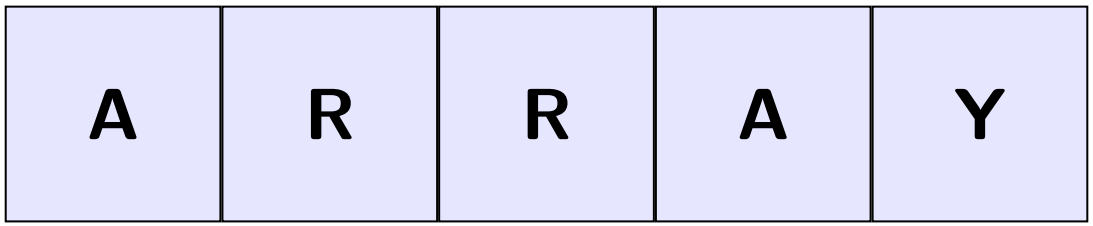
1. **Q1:** Is a Python list a linear or non-linear data structure?
2. **Q2:** Why do we measure time complexity in operations rather than seconds?
3. **Q3:** If a function contains a loop that runs 'n' times, and inside that loop is another loop running 'n' times, what is the Big O time complexity?

Key Takeaways

- Data structures are essential for efficient data organization, storage, and retrieval.
- They are classified into primitive/non-primitive and linear/non-linear types.
- Algorithm analysis evaluates efficiency via Time and Space complexity.
- Big 'O' notation is used to define the upper bound (worst-case) of an algorithm's running time.
- Algorithms are evaluated on Best, Average, and Worst cases.

Next Lecture Preview

- **Lecture 2:** We will dive into specific Linear Data Structures.
- We will explore Arrays and Strings in detail.
- Get ready to write Python code to manipulate arrays and analyze their complexity!



Questions?

Lecture 1.2: OOP Concepts, Classes & Objects

Complete Lecture Deck – All 30 Lectures

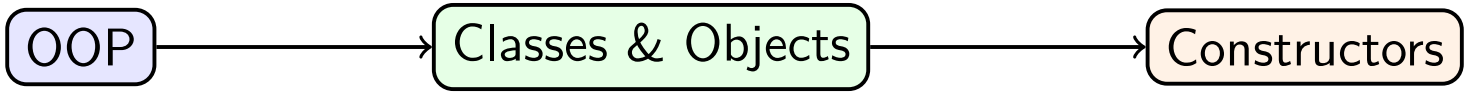
Milav Dabgar

Lecturer, GP Palanpur

July 22, 2026

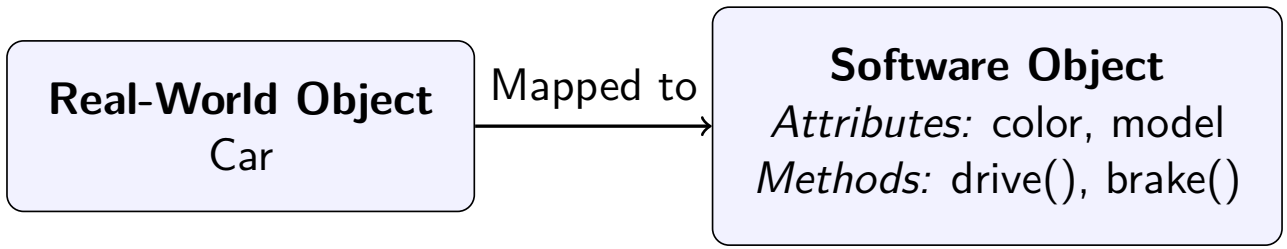
Today's Agenda

- Introduction to Object-Oriented Programming (OOP)
- The Four Pillars of OOP
- Defining Classes and Creating Objects in Python
- The Role of Constructors (`__init__`)
- Understanding the `self` parameter



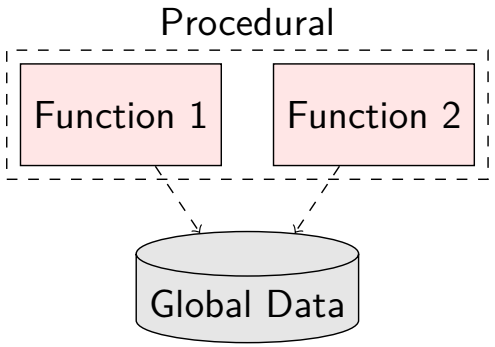
What is Object-Oriented Programming?

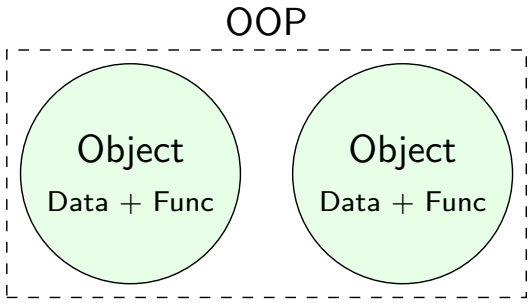
- OOP is a programming paradigm based on the concept of 'objects'
- Objects can contain data (attributes/properties) and code (methods/functions)
- It models real-world entities (e.g., a 'Student' with a name and a 'study()' action)
- Python is a multi-paradigm language, but its core is deeply object-oriented



Procedural vs Object-Oriented

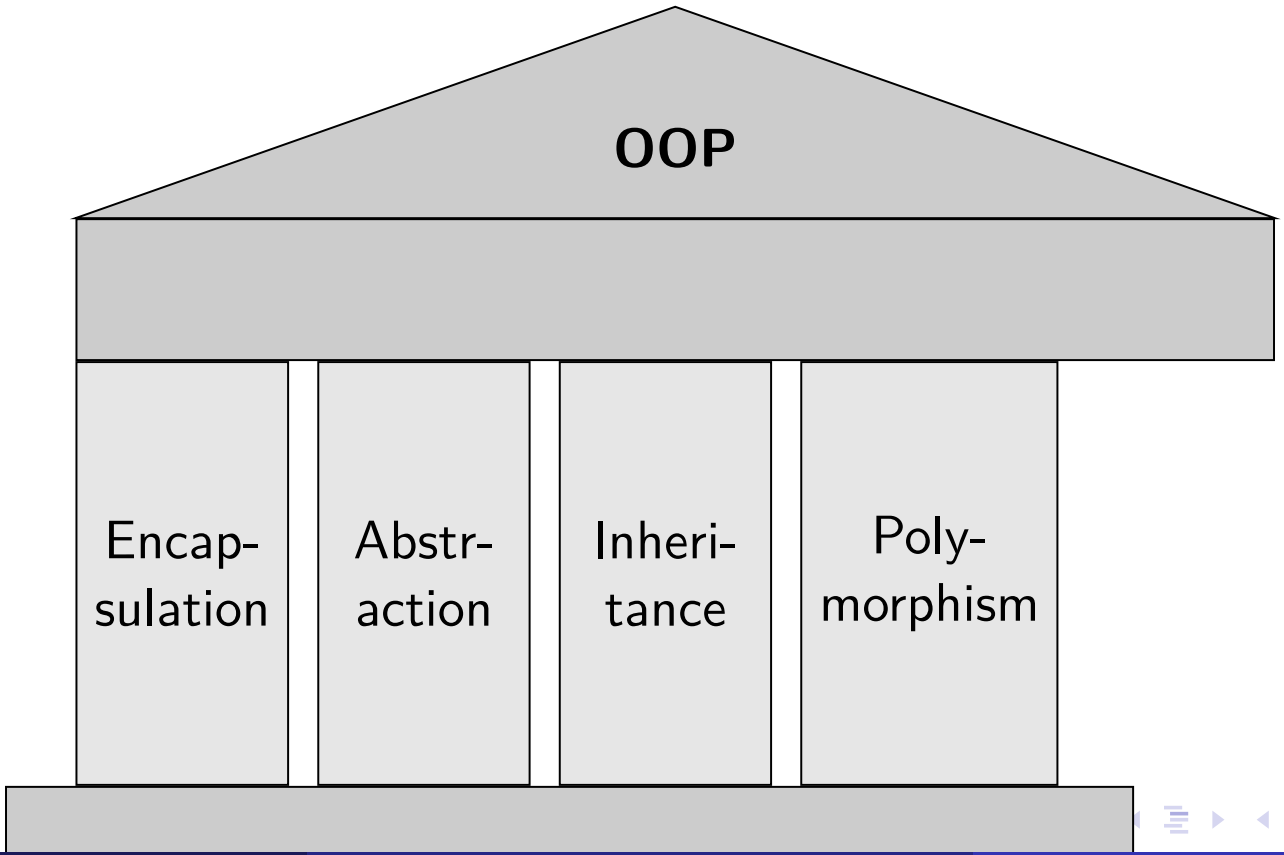
Procedural Programming	**Object-Oriented Programming**
Focuses on writing functions that operate on data	Focuses on objects that combine data and functions
Data and functions are separate	Easier to model complex systems
Harder to maintain for large projects	Promotes code reuse and modularity





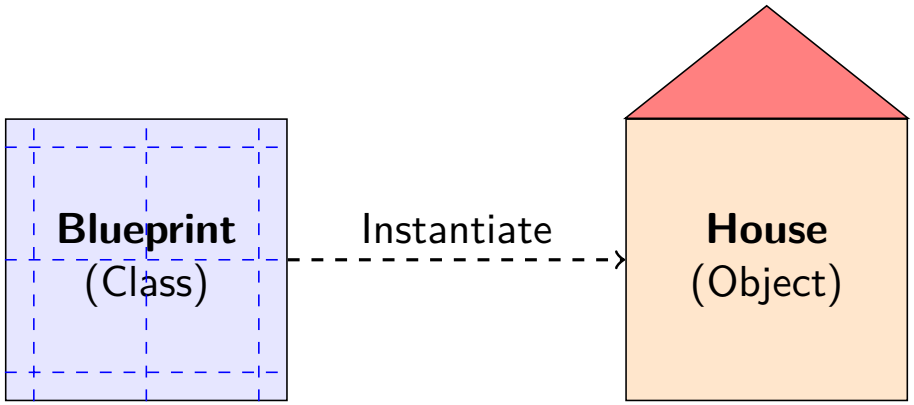
Four Pillars of OOP

1. **Encapsulation**: Bundling data and methods that work on that data within one unit
2. **Abstraction**: Hiding complex implementation details
3. **Inheritance**: Creating new classes based on existing ones
4. **Polymorphism**: Different classes treated as instances of same class



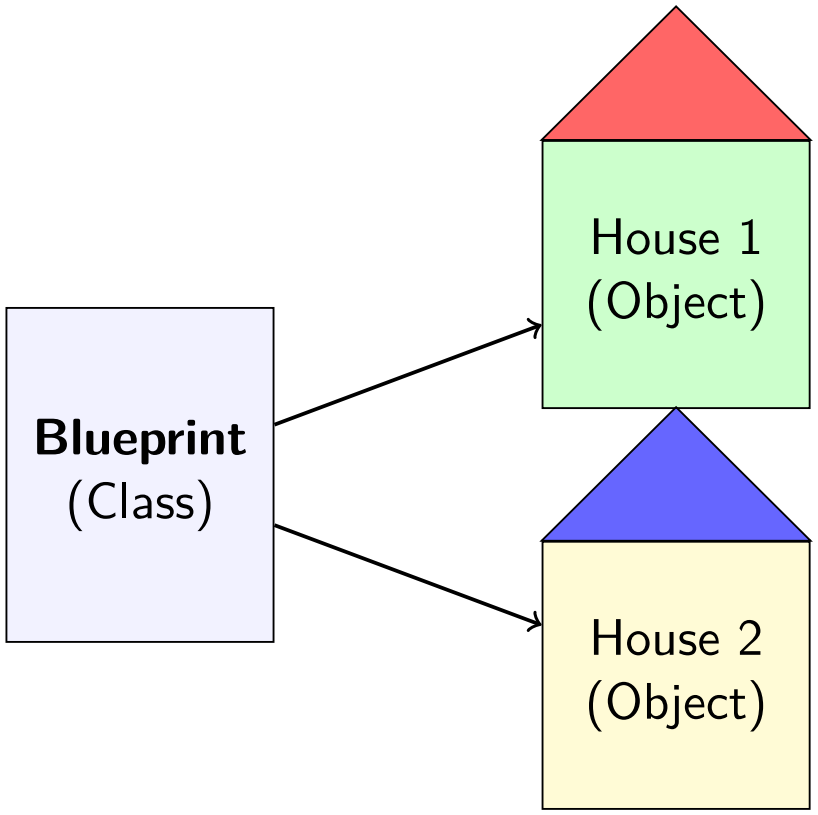
What is a Class?

- A class is a blueprint or template for creating objects
- It defines a datatype by bundling data and methods that work on the data
- A class doesn't consume memory until an object of it is created
- Analogy: A class is like the architectural blueprint of a house



What is an Object?

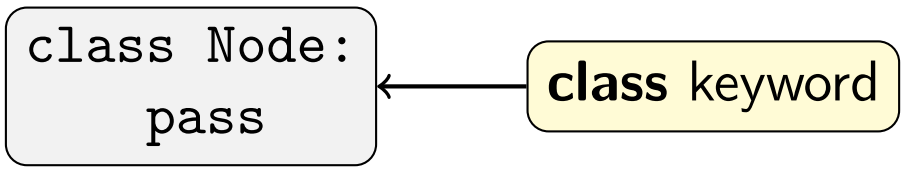
- An object is an instance of a class
- Memory is allocated only when an object is instantiated
- You can create many objects from a single class
- Analogy: The object is the actual physical house built from the blueprint



Defining a Class in Python

- Use the `class` keyword to define a class.

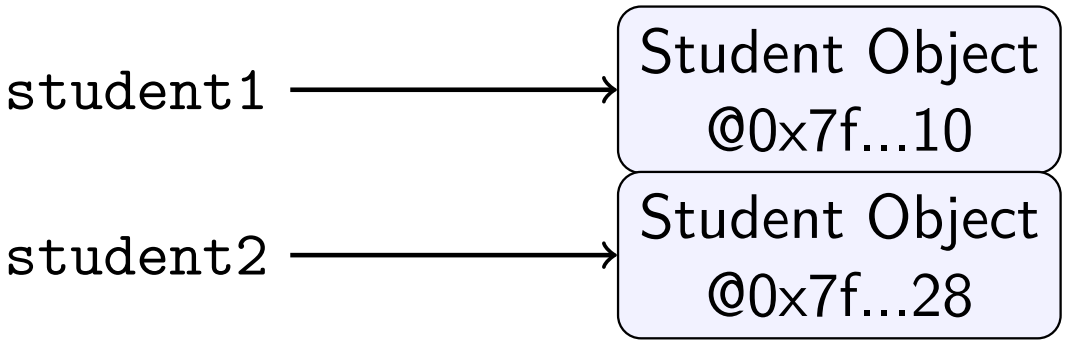
```
class Node:
    # A simple class representing a node in a Linked List
    pass

# 'pass' is used as a placeholder when the class is empty
```



Creating an Object (Instantiation)

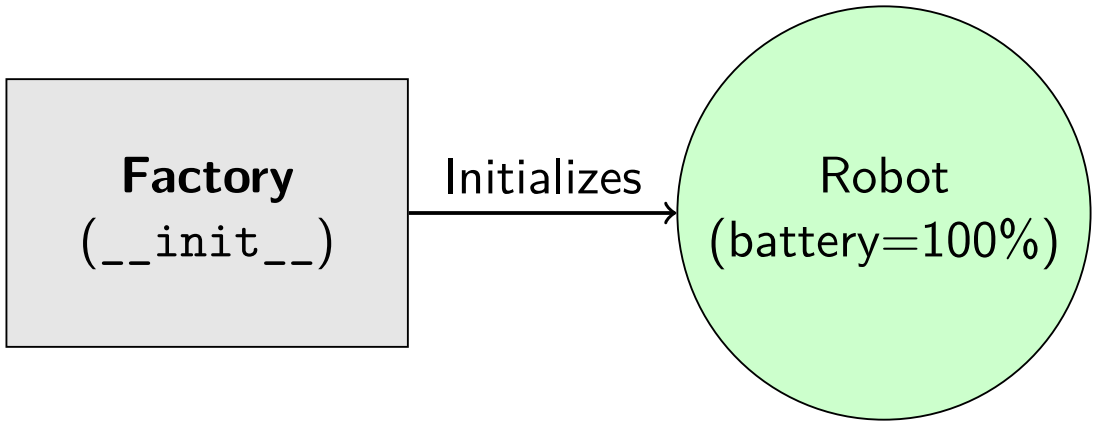
```
class Student:
    pass

# Creating objects of the Student class
student1 = Student()
student2 = Student()
```

Here, `student1` and `student2` are two distinct objects in memory.



What are Constructors?

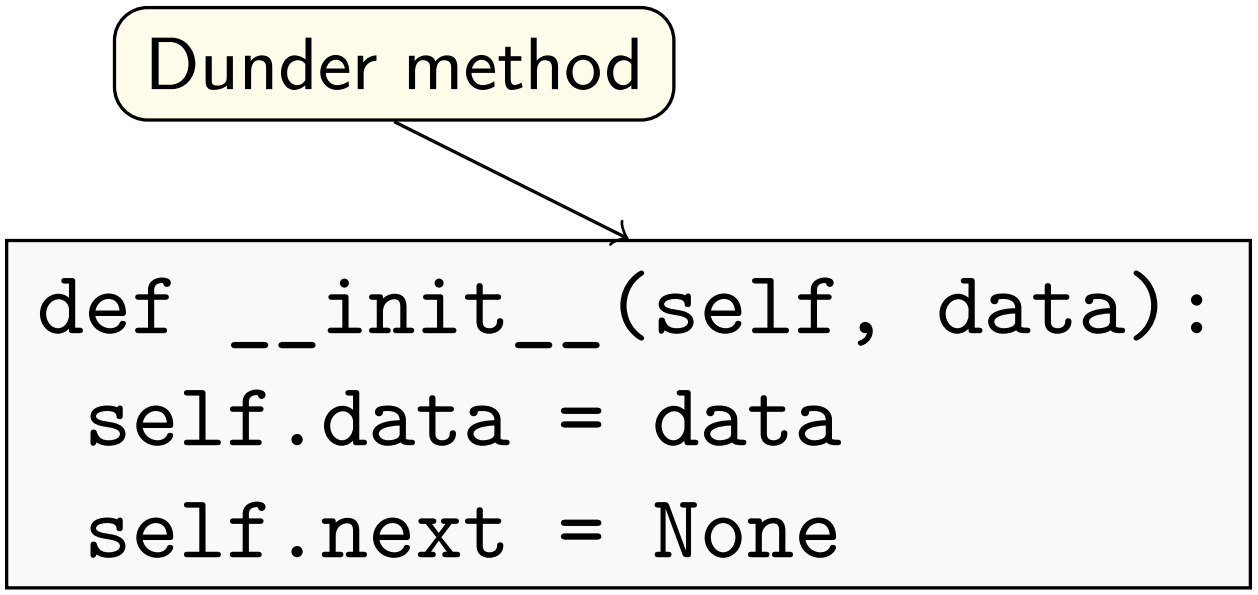
- A constructor is a special method used to initialize objects
- It is called automatically when an object is created
- Primary purpose: To assign initial values to the object's attributes
- In Python, the constructor method is always named `__init__`



The __init__ Method

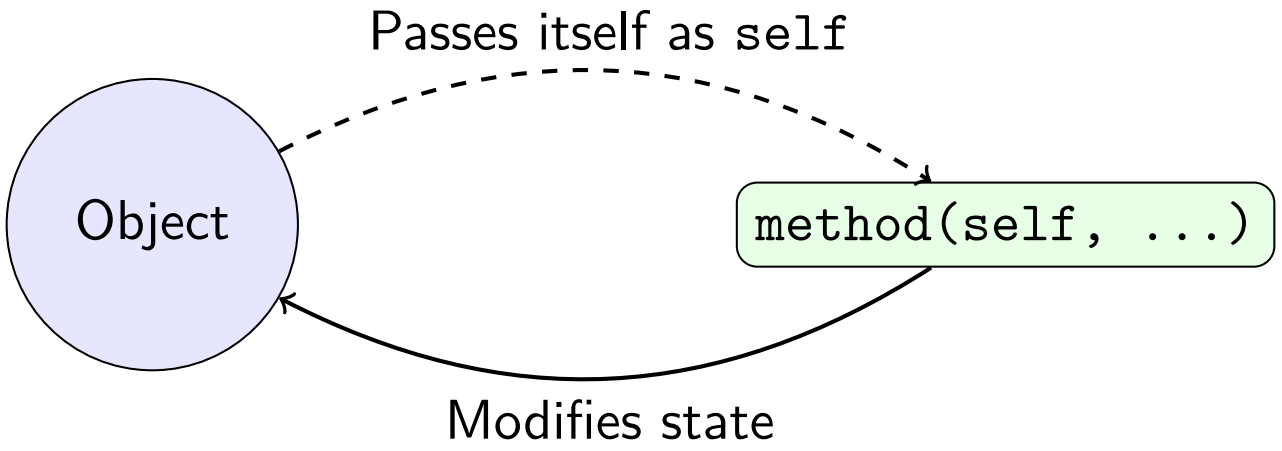
- __init__ is surrounded by double underscores (dunder method)
- It initializes the state of the new Node object

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```



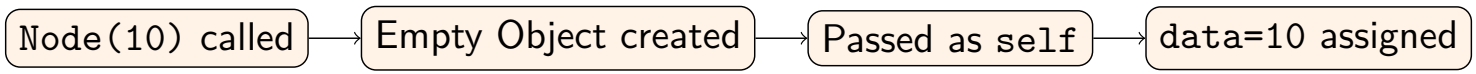
Understanding the `self` Parameter

- `self` represents the instance (object) being created or manipulated
- It is explicitly provided as the first parameter in instance methods
- Python passes the object itself to `self` automatically
- It allows access to the attributes and methods of the class



Constructor in Action

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

# Creating a new node
node1 = Node(10)
print(node1.data)  # Output: 10
```



Default vs Parameterized Constructors

Default Constructor:

- No parameters (other than `self`)
- Sets default values

```
class A:
    def __init__(self):
        self.val = 0
```

Parameterized Constructor:

- Takes arguments
- Sets specific values

```
class Node:
    def __init__(self, data):
        self.data = data
```

Simulating Multiple Constructors

Python does not support method overloading directly.
Use default arguments to simulate multiple constructors:

```
class Student:
    def __init__(self, name="Unknown"):
        self.name = name

s1 = Student()           # name is 'Unknown'
s2 = Student("Alice")    # name is 'Alice'
```

name="Unknown"
Default Argument

Quick Check: Test Your Understanding

Questions

1. What is the main difference between a class and an object?
2. What is the name of the constructor method in Python?
3. Why is the `self` parameter necessary in Python classes?

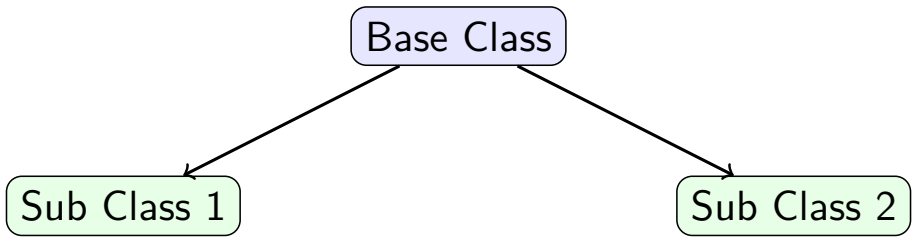
Key Takeaways

- ✓ OOP organizes code around objects combining data and behavior
- ✓ A Class is a blueprint; an Object is a physical instance of that blueprint
- ✓ Constructors (`__init__`) are special methods used to initialize newly created objects
- ✓ `self` acts as a reference to the current instance of the class

Next Lecture Preview

Preview: Lecture 1.3

- In Lecture 3, we will dive deeper into Object-Oriented principles.
- We'll cover Inheritance and Polymorphism, showing how objects can share behavior and interact flexibly.



Any Questions?

Lecture 1.3: Methods & Abstract Data Types

Complete Lecture Deck – All 30 Lectures

Milav Dabgar

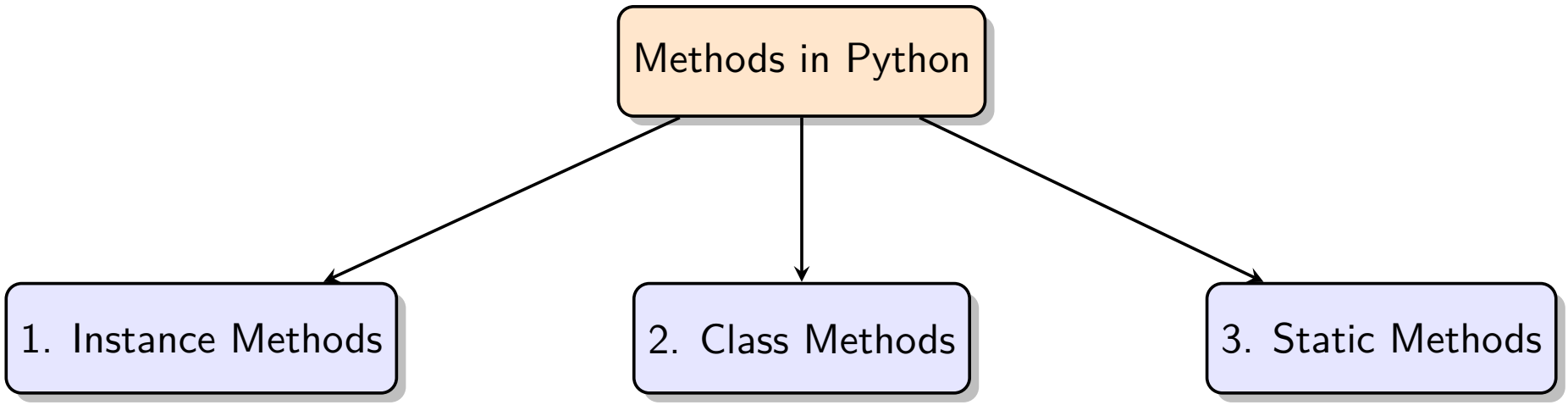
Lecturer, GP Palanpur

July 22, 2026

Course Code: DI03032031
GTU Diploma Engineering - Semester 3 (32-ICT)
Unit 1: Basic Concepts of Data Structures & OOP

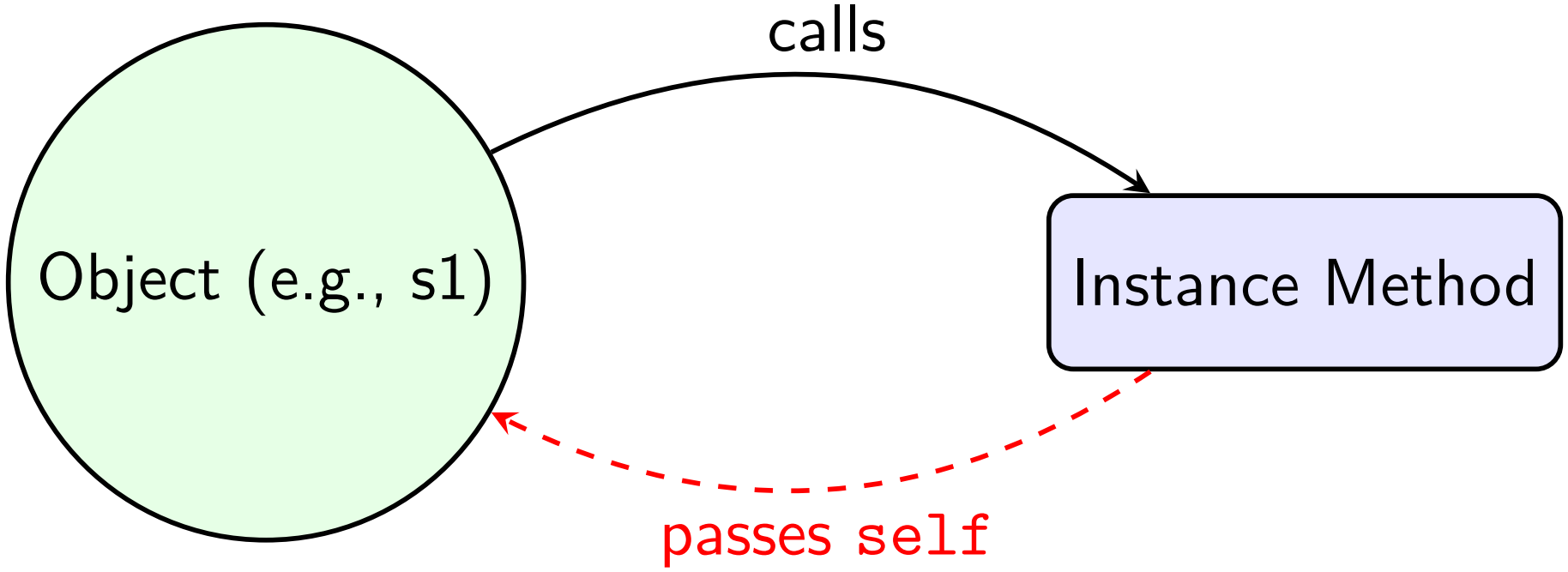
Methods in Python: An Overview

- A method is a function defined inside a class body
- Methods represent the 'behaviors' of objects or the class itself
- In Python, we categorize methods into three types:



1. Instance Methods

- The most common type of method in Python classes
- They operate on an instance of the class (an object)
- They can access and modify instance state (using `self`) and class state
- Must have `self` as the first parameter
- Called on an object: `object_name.method_name()`

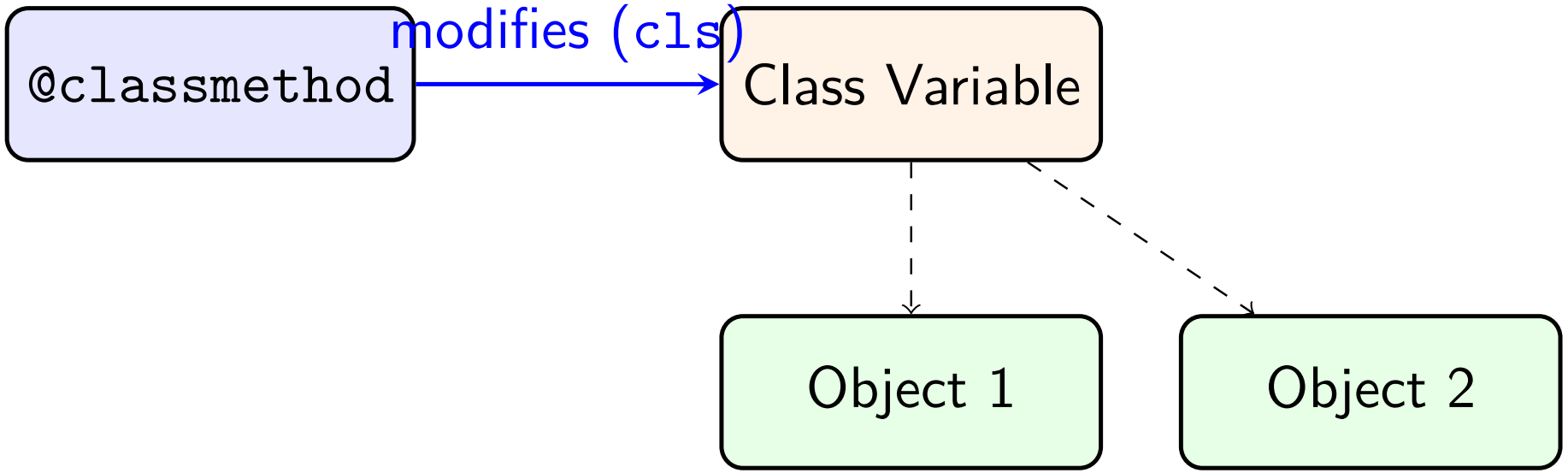


Instance Method: Example

```
class Student:
    def __init__(self, name, marks):
        self.name = name
        self.marks = marks

    def display_info(self): # Instance Method
        print(f'Student {self.name} scored {self.marks}')

s1 = Student('Alice', 85)
s1.display_info() # Output: Student Alice scored 85
```

2. Class Methods

- Bound to the class itself, not individual instances
- They can modify class state that applies across all instances
- Defined using the `@classmethod` decorator
- Must take `cls` (the class) as the first parameter instead of `self`
- Often used as alternative constructors or to manage class-level variables

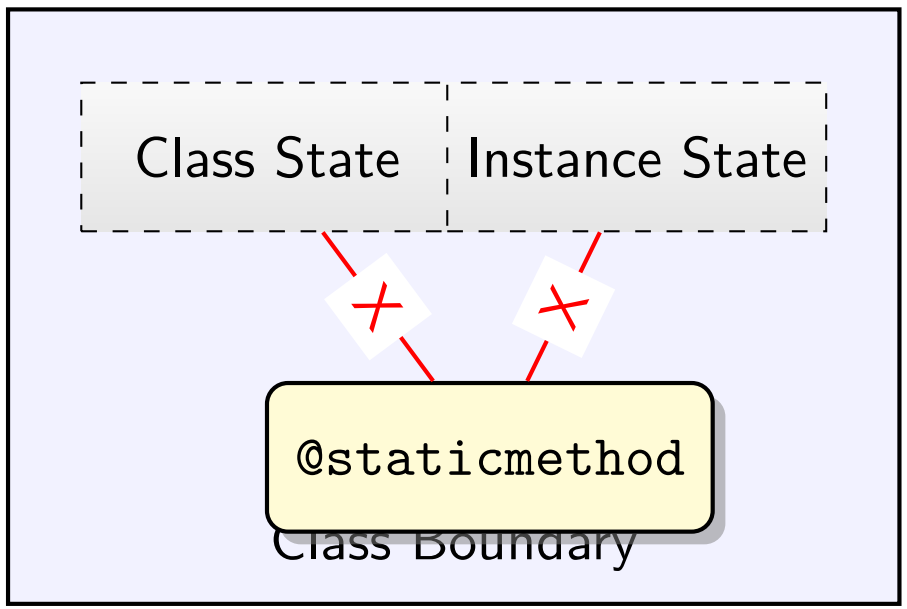


Class Method: Example

```
class Student:
    school_name = 'GTU' # Class variable

    @classmethod
    def change_school(cls, new_name):
        cls.school_name = new_name

Student.change_school('Gujarat Tech Univ')
print(Student.school_name) # Output: Gujarat Tech Univ
```

3. Static Methods

- Methods that don't need access to instance (`self`) or class (`cls`) state
- They behave like regular functions but belong to the class's namespace
- Defined using the `@staticmethod` decorator
- They do not take implicit first arguments (no self, no cls)
- Used for utility or helper functions related to the class



Static Method: Example

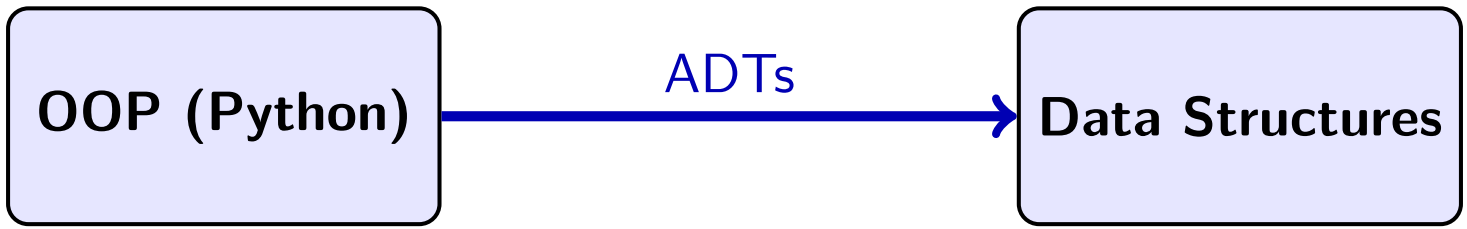
```
class MathUtils:
    @staticmethod
    def add_numbers(a, b):
        return a + b

# Called using class name
result = MathUtils.add_numbers(5, 10)
print(result) # Output: 15
```

Summary: Types of Methods

Method Type	Decorator	First Param	Accesses
Instance Method	(None)	`self`	Object & Class data
Class Method	`@classmethod`	`cls`	Class data only
Static Method	`@staticmethod`	(Standard args)	Nothing (Utility)

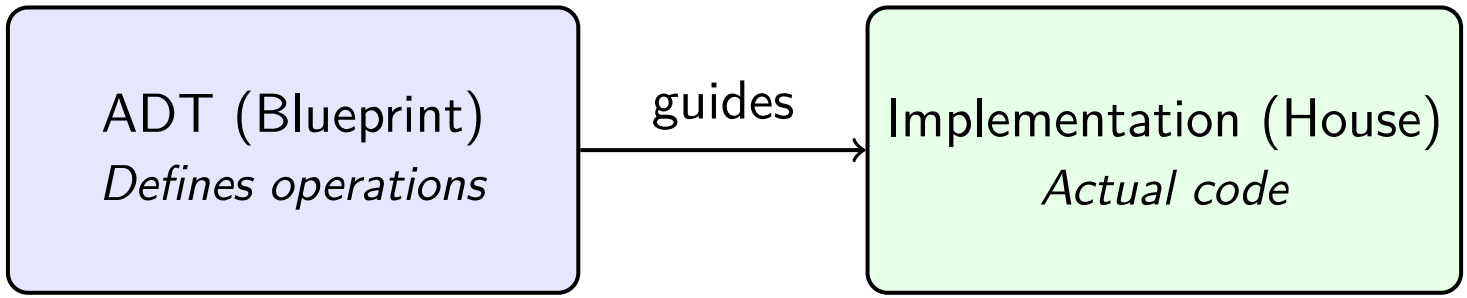
Transition to Data Structures

- We've covered the OOP tools Python gives us (Classes, Objects, Methods).
- Now, how do we use these tools to solve complex problems?
- We need ways to organize and manage data efficiently.
- Enter: **Abstract Data Types (ADTs)**.



What is an Abstract Data Type (ADT)?

- An ADT is a logical description or model of a data structure
- It specifies **WHAT** data is stored and **WHAT** operations can be performed on it
- It **DOES NOT** specify **HOW** the data is stored or **HOW** the operations are implemented
- Think of it as a blueprint or an interface in OOP



ADT vs. Data Structure

Abstract Data Type (ADT)

The theoretical concept.

- **Example:** A 'List' ADT.
- **Operations:** insert, remove, search, size.

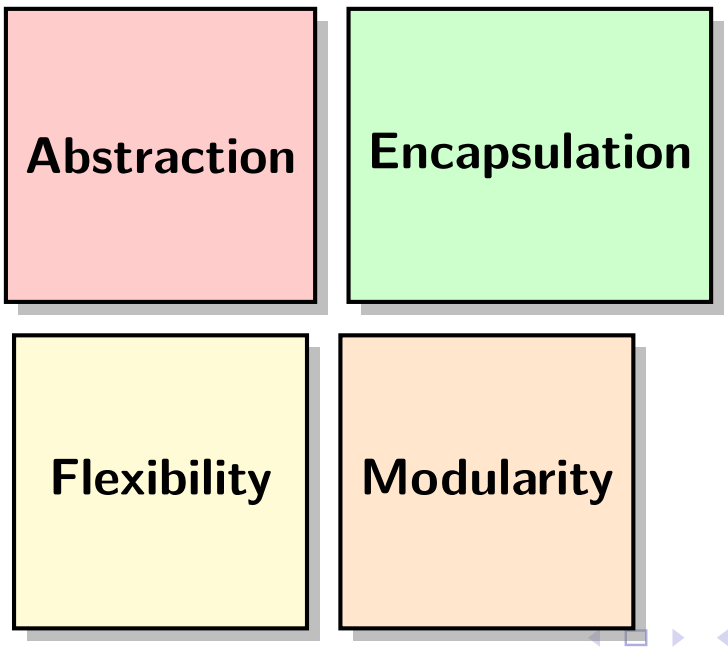
Data Structure

The physical implementation in a programming language.

- **Example:** Python's built-in 'list' (which is a dynamic array).
- Contains actual code for memory management and looping.

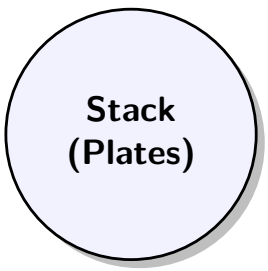
Why Use ADTs?

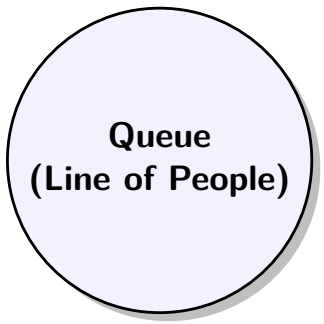
- **Abstraction:** Users don't need to know the messy internal code to use it.
- **Encapsulation:** Data is protected from outside interference.
- **Flexibility:** You can change the internal implementation without breaking the user's code.
- **Modularity:** Large programs are broken into smaller, manageable ADTs.



Common Examples of ADTs

- **Stack ADT:** LIFO (Last In, First Out). Operations: `push()`, `pop()`, `peek()`.
- **Queue ADT:** FIFO (First In, First Out). Operations: `enqueue()`, `dequeue()`.
- **List ADT:** Ordered collection. Operations: `insert()`, `delete()`, `get()`.
- We will build all of these from scratch in Python during this course!





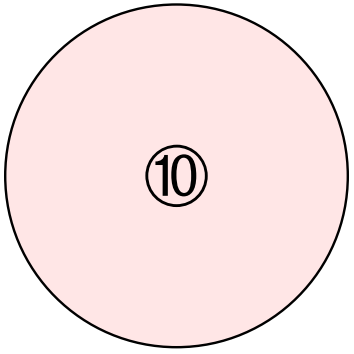
Quick Check: Test Your Understanding

1. Which method type in Python takes `cls` as its first parameter?
2. True or False: A static method can modify an object's instance variables.
3. Is 'Stack' an ADT or a specific block of Python code?
4. What is the main difference between an ADT and a Data Structure?

Key Takeaways

- ✓ Instance methods operate on objects (`self`).
- ✓ Class methods operate on classes (`cls`) using `@classmethod`.
- ✓ Static methods are independent functions inside a class using `@staticmethod`.
- ✓ An ADT defines what a data structure does, but not how it is implemented.
- ✓ ADTs rely on abstraction and encapsulation.

Next Lecture Preview



Performance Analysis

- In the next lecture, we will dive into Performance Analysis.
- We will learn about Time Complexity, Space Complexity, and Big-O notation.
- How do we know if our Data Structure is fast or slow?

Thank You!

Any Questions?

Lecture 1.4: Recursion Basics

Complete Lecture Deck – All 30 Lectures

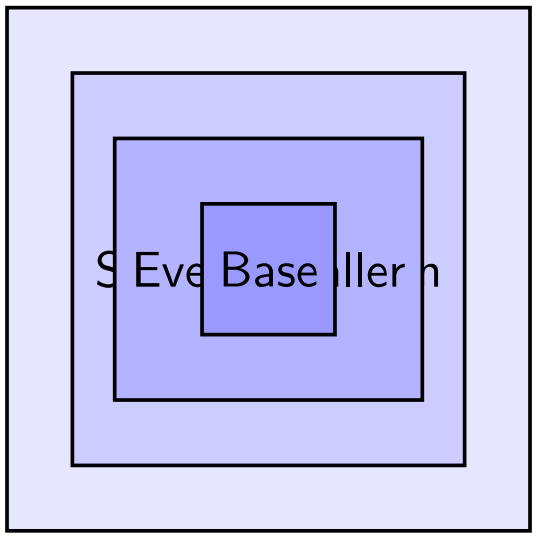
Milav Dabgar

Lecturer, GP Palanpur

July 22, 2026

What is Recursion?

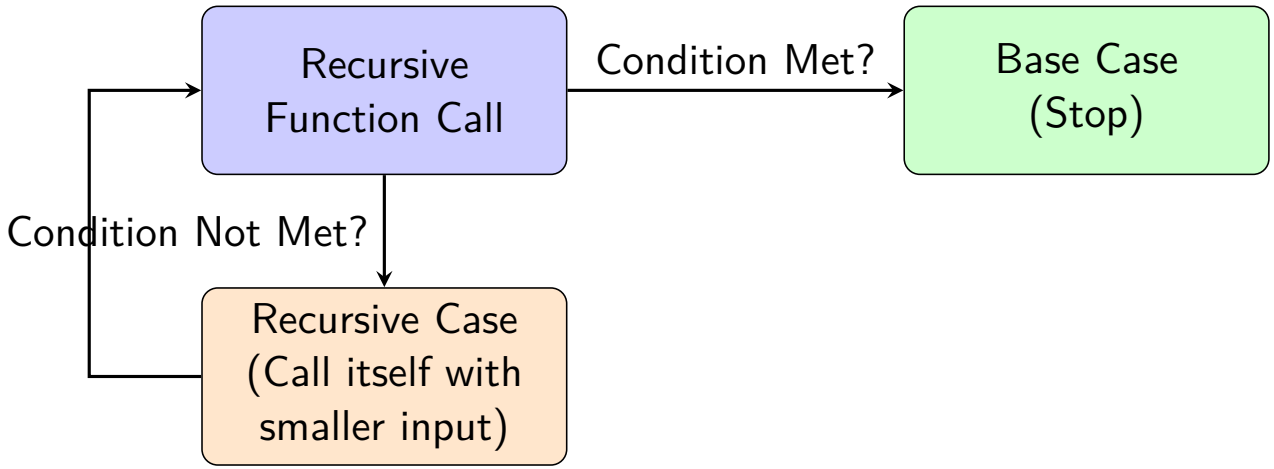
- In programming, recursion occurs when a function calls itself to solve a smaller instance of the same problem.
- It's a method of solving a problem where the solution depends on solutions to smaller instances of the same problem.
- Think of looking at two mirrors facing each other, creating an infinite tunnel of reflections.
- Or Russian Matryoshka dolls: opening one reveals a smaller identical doll inside, until you reach the smallest one.



The Anatomy of a Recursive Function

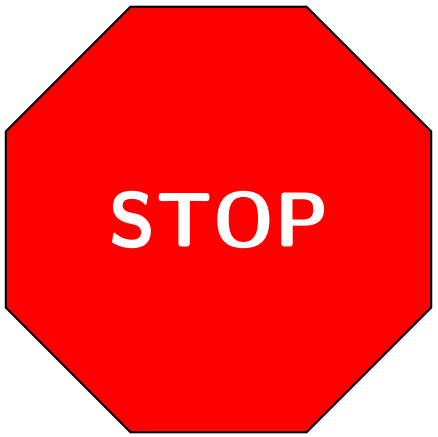
Every properly written recursive function must have two parts:

1. **The Base Case (The Stopping Condition):**
   - The condition under which the function stops calling itself.
   - Without this, the function would run forever.
2. **The Recursive Case (The Self-Call):**
   - The part where the function calls itself with a modified, smaller input.
   - It must move the function closer to the base case.



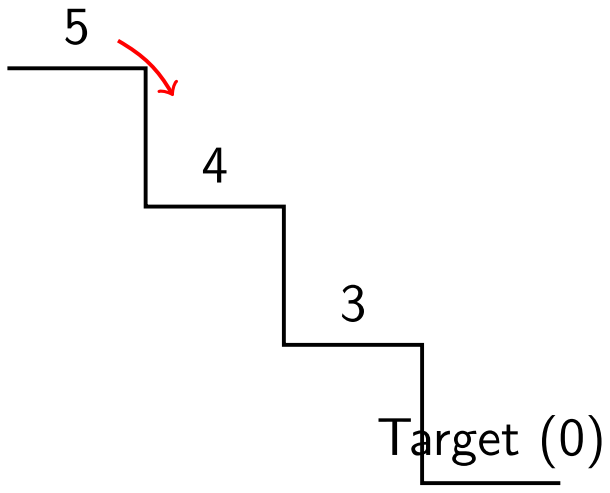
The Base Case in Detail

- The base case is the simplest possible scenario that can be answered directly without further recursion.
- It acts as the 'anchor' or 'exit strategy'.
- Example: If you are counting down to zero, zero is the base case.
- When the base case is reached, the function simply returns a result instead of making another recursive call.



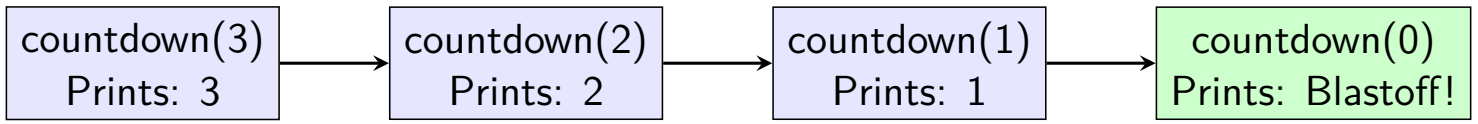
The Recursive Case in Detail

- The recursive case breaks the larger problem down into a smaller piece.
- It performs some action, and then calls the same function again with new parameters.
- Crucially, the new parameters MUST be a step towards the base case.
- If you are at 5, and the base case is 0, the next call should be for 4, not 6.



Example 1: A Simple Countdown (Concept)

- **Problem:** Write a function that counts down from a number N to 1, then prints 'Blastoff!'.
- **Base Case:** If N is 0 or less, print 'Blastoff!' and stop.
- **Recursive Case:** Print N, then call the countdown function again for (N - 1).

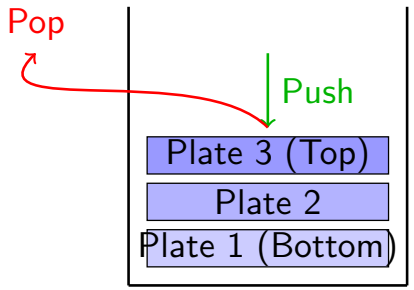


Example 1: A Simple Countdown (Python Code)

```
def countdown(n):
    # Base Case
    if n <= 0:
        print('Blastoff!')
        return

    # Recursive Case
    print(n)
    countdown(n - 1)

countdown(3)
```

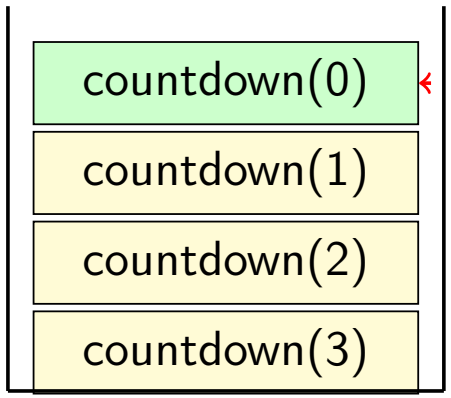
Behind the Scenes: The Call Stack

- How does the computer keep track of all these function calls?
- It uses a data structure called the 'Call Stack'.
- A stack works like a stack of plates: Last-In, First-Out (LIFO).
- Every time a function is called, it's added (pushed) to the top of the stack.
- Every time a function finishes (returns), it's removed (popped) from the stack.



Visualizing the Countdown Stack

1. `countdown(3)` starts. Calls `countdown(2)`.
2. `countdown(2)` starts. Calls `countdown(1)`.
3. `countdown(1)` starts. Calls `countdown(0)`.
4. `countdown(0)` starts. Hits base case, prints Blastoff! Returns.
5. `countdown(1)` resumes and finishes. Returns.
6. `countdown(2)` resumes and finishes. Returns.
7. `countdown(3)` resumes and finishes. Returns.



Base case reached!

Call Stack

Example 2: Calculating Factorial (Concept)

- Factorial of N (written as N!) is the product of all positive integers less than or equal to N.
- 5! = 5 * 4 * 3 * 2 * 1 = 120
- Notice the pattern: 5! = 5 * (4 * 3 * 2 * 1) = 5 * 4!
- In general: N! = N * (N - 1)!
- Base Case: 1! = 1 (and 0! = 1)

$$\mathbf{5! = 5 \times 4!}$$

$$\mathbf{4! = 4 \times 3!}$$

$$\mathbf{3! = 3 \times 2!}$$

$$\mathbf{2! = 2 \times 1!}$$

$$\mathbf{1! = 1}$$

Example 2: Factorial Code (Python)

```python
def factorial(n):
    # Base Case
    if n == 0 or n == 1:
        return 1

    # Recursive Case
    return n * factorial(n - 1)

print(factorial(5)) # Output: 120
```

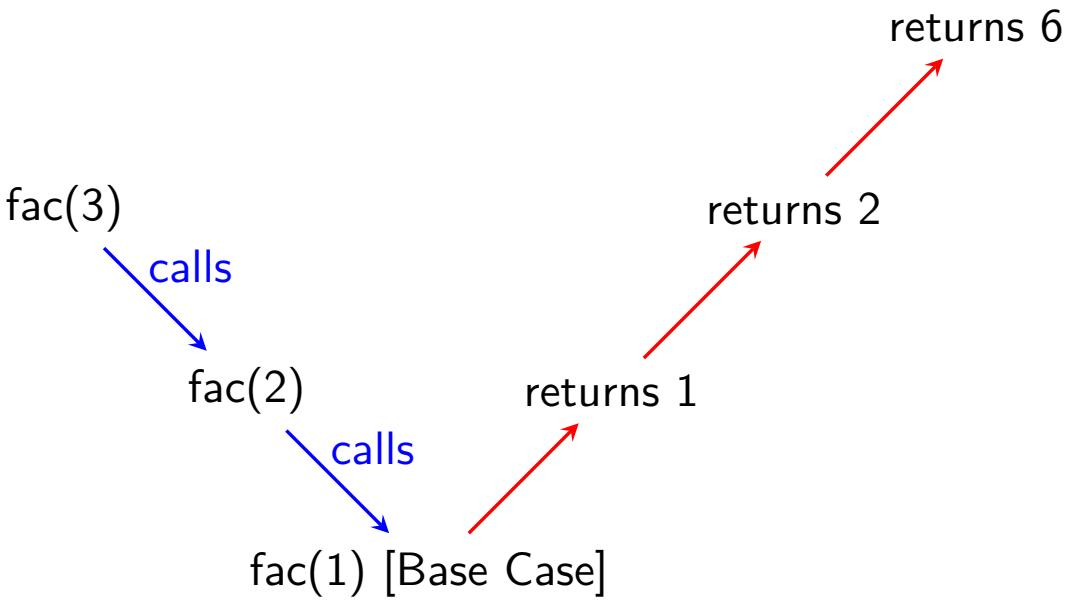
Tracing Factorial Execution

Call phase (winding up):

- `fac(3)` → waits for 3 * `fac(2)`
- `fac(2)` → waits for 2 * `fac(1)`
- `fac(1)` → hits base case, returns 1

Return phase (unwinding):

- `fac(2)` receives 1, calculates 2 * 1 = 2, returns 2
- `fac(3)` receives 2, calculates 3 * 2 = 6, returns 6

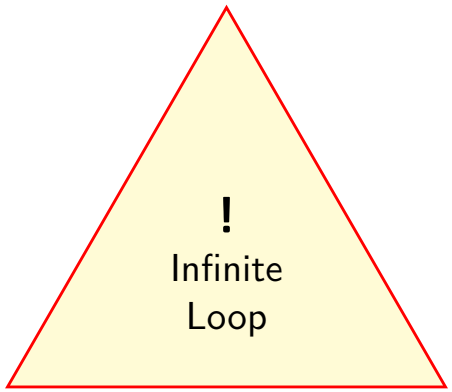


Danger: Infinite Recursion

What happens if we forget the base case, or if our recursive step never reaches it?

```
def infinite(n):
    print(n)
    infinite(n + 1) # Problem: numbers get larger
```

The function calls itself forever, adding a new frame to the call stack every time.



Stack Overflow Error

- The call stack has a limited amount of memory.
- If a recursive function calls itself too many times without returning, the stack runs out of space.
- In Python, this results in a `RecursionError: maximum recursion depth exceeded`.
- This is universally known as a 'Stack Overflow' (yes, like the website!).

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in infinite
...
  File "<stdin>", line 3, in infinite
RecursionError: maximum recursion depth exceeded
```

Recursion vs Iteration (Loops)

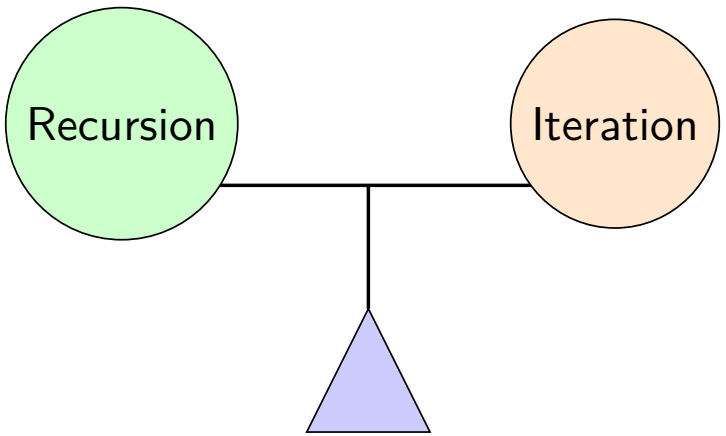
Anything you can do with recursion, you can do with a loop (iteration), and vice versa.

Why use Recursion?

- Often leads to cleaner, shorter, and more elegant code.
- Great for problems naturally defined recursively.

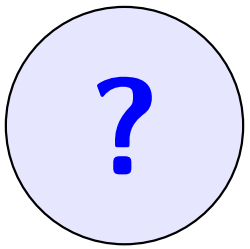
Why use Iteration?

- Recursion has a performance cost (function calls).
- Loops don't risk Stack Overflow for large inputs.



Quick Check: Test Your Understanding

1. **Q1:** What are the two essential parts of any recursive function?
2. **Q2:** What happens if a recursive function does not have a base case?
3. **Q3:** Name the data structure the computer uses to keep track of recursive function calls.



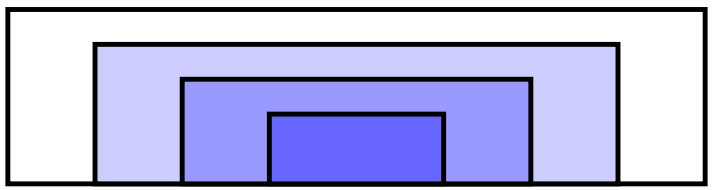
Key Takeaways

- Recursion is a technique where a function calls itself to solve a smaller sub-problem.
- A base case is required to stop the recursion and prevent stack overflow.
- The recursive step must always move the problem closer to the base case.
- The Call Stack tracks active function calls, pausing the parent function while the child executes.
- Recursion offers elegant solutions but uses more memory than iterative loops.

Next Lecture Preview

Lecture 5: Advanced Recursion.

We will explore the Fibonacci sequence, understand the concept of multiple recursive calls, and look at some puzzle-solving applications like the Towers of Hanoi.



Towers of Hanoi

Questions?

Q & A

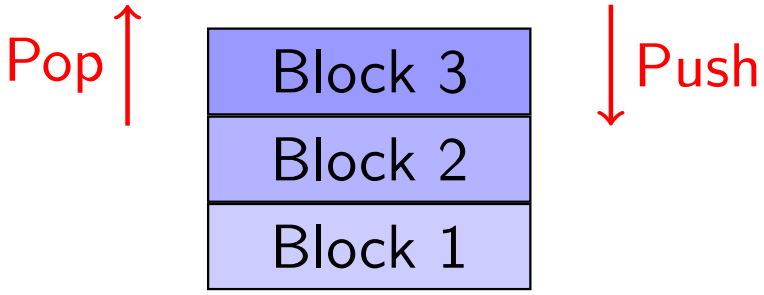
Lecture 2.1: 2.1 Overview of Stack, Operations on Stack

Complete Lecture Deck – All 30 Lectures

Milav Dabgar

Lecturer, GP Palanpur

July 22, 2026

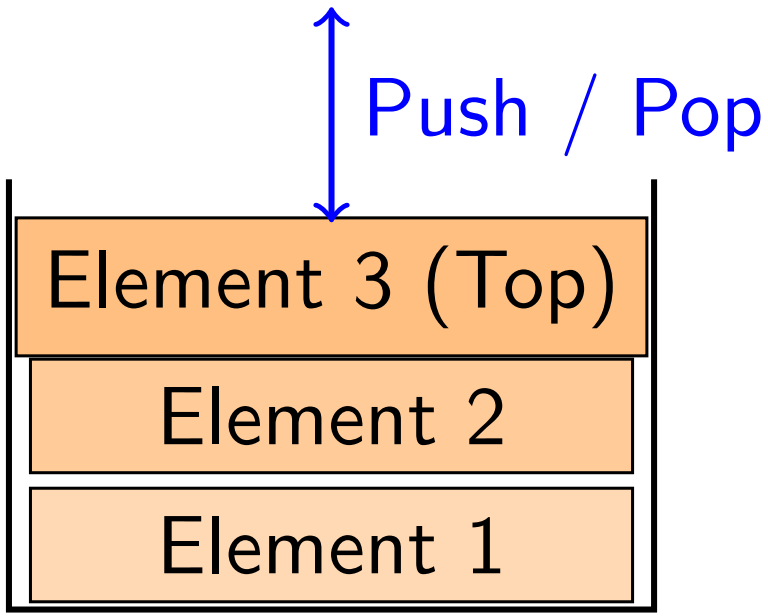


Today's Agenda

- ✓ What is a Stack?
- ✓ Real-world and Computing Examples
- ✓ The LIFO Principle
- ✓ Core Stack Operations Overview
- ✓ Deep Dive: Push Operation
- ✓ Deep Dive: Pop Operation
- ✓ Stack Overflow and Underflow

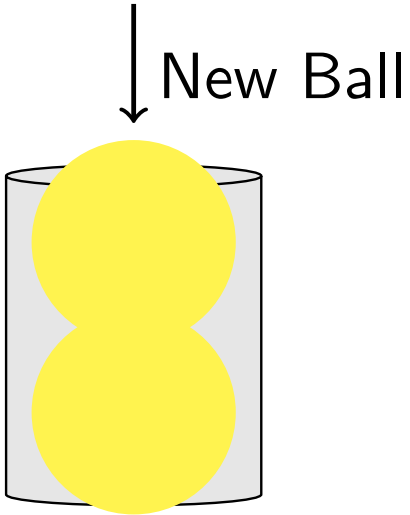
What is a Stack?

- A Stack is a linear data structure that follows a particular order in which the operations are performed.
- The order is LIFO (Last In First Out).
- This means the last element added to the stack will be the first element to be removed.
- Elements are added and removed from only one end, called the 'top' of the stack.



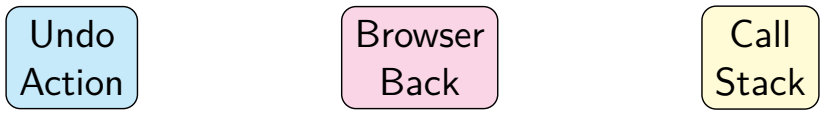
Real-world Analogies

- A stack of plates in a cafeteria: you take the top plate, and new plates are added to the top.
- A stack of books on a desk: adding a new book goes on top, reading a book starts from the top.
- Tennis balls in a cylindrical container: the last ball put in is the first one taken out.



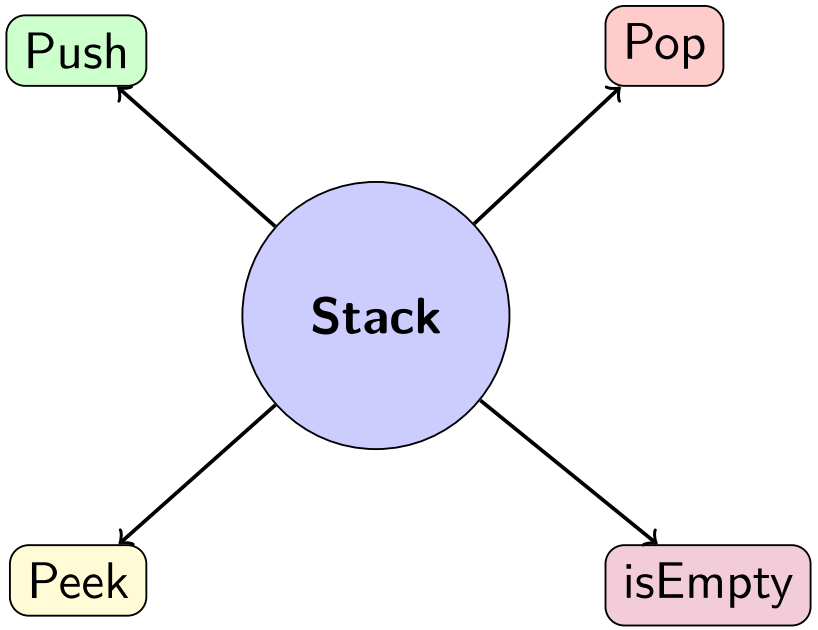
Stacks in Computing

- Undo mechanism in text editors: your most recent action is undone first.
- Browser history (Back button): clicking back takes you to the most recently visited page.
- Function calls in programming: tracking active subroutines (the Call Stack).



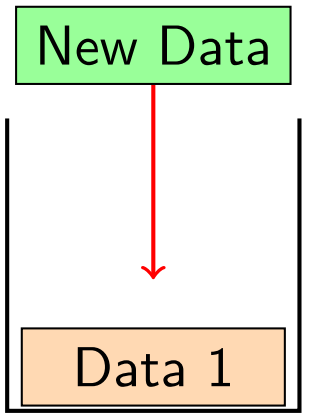
Core Stack Operations

- **Push**: Adds an item to the stack. If the stack is full, it results in an *Overflow* condition.
- **Pop**: Removes an item from the stack. If the stack is empty, it results in an *Underflow* condition.
- **Peek** (or Top): Returns the top element of the stack without removing it.
- **isEmpty**: Returns true if the stack is empty.



The Push Operation - Concept

- Pushing means inserting a new element at the top of the stack.
- Step 1: Check if the stack is full.
- Step 2: If the stack is full, produce an error and quit.
- Step 3: If not full, increment the top pointer to point to the next empty space.
- Step 4: Add the data element to the stack location where top is pointing.



Top moves up after insertion

Push Operation - Algorithm

Algorithm: push(stack, max_size, item)

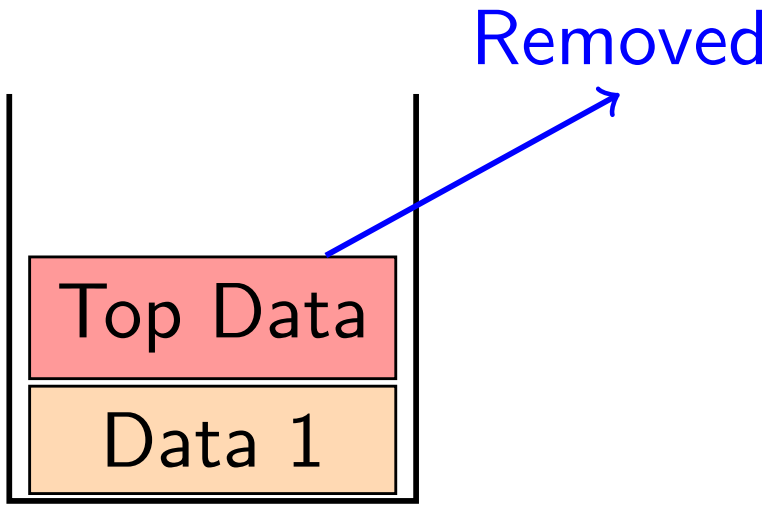
1. If `top == max_size - 1`
   - Print 'Stack Overflow'
   - Return
2. Else
   - `top = top + 1`
   - `stack[top] = item`
   - Print 'Pushed', item

Push Operation - Python Code

```
def push(stack, item, max_size):
    if len(stack) >= max_size:
        print('Stack Overflow!')
    else:
        stack.append(item)
        print(f'Pushed {item} to stack')

my_stack = []
push(my_stack, 10, 3)
```

The Pop Operation - Concept

- Popping means removing the top element from the stack.
- Step 1: Check if the stack is empty.
- Step 2: If the stack is empty, produce an error and quit.
- Step 3: If not empty, access the data element at which top is pointing.
- Step 4: Decrease the value of top by 1.
- Step 5: Return the extracted data.



Pop Operation - Algorithm

Algorithm: pop(stack)

1. If `top == -1`
   - Print 'Stack Underflow'
   - Return null
2. Else
   - `item = stack[top]`
   - `top = top - 1`
   - Return item

Pop Operation - Python Code

```
def pop(stack):
    if len(stack) == 0:
        print('Stack Underflow!')
        return None
    else:
        item = stack.pop()
        print(f'Popped {item} from stack')
        return item

popped_item = pop(my_stack)
```

Edge Cases: Overflow and Underflow

Stack Limits

- **Stack Overflow**: Occurs when trying to push an element onto a stack that is already full. Memory limit is reached.
- **Stack Underflow**: Occurs when trying to pop an element from an empty stack. No elements exist to be removed.

Robust code must always check for these conditions before performing stack operations to prevent program crashes.

WARNING: Overflow

WARNING: Underflow

Quick Check: Test Your Understanding

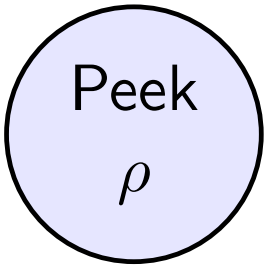
- **Q1**: What does LIFO stand for?
- **Q2**: If you push elements 1, 2, and 3 onto a stack in that order, what is the first element popped?
- **Q3**: What error do you get if you try to pop from an empty stack?
- **Q4**: In Python, which list method is used to simulate a stack push?

Key Takeaways

- A Stack is a LIFO (Last-In-First-Out) linear data structure.
- Push operation adds an element to the top of the stack.
- Pop operation removes an element from the top of the stack.
- Always check for Stack Overflow before pushing, and Stack Underflow before popping.

Next Lecture Preview

- In Lecture 6, we will explore the remaining stack operations: Peek and isEmpty.
- We will also look at practical applications of stacks, such as reversing a string and expression evaluation.



$$(A + B) * C$$

Questions?

Lecture 2.2: 2.2 Implementation of Stack using List

Complete Lecture Deck – All 30 Lectures

Milav Dabgar

Lecturer, GP Palanpur

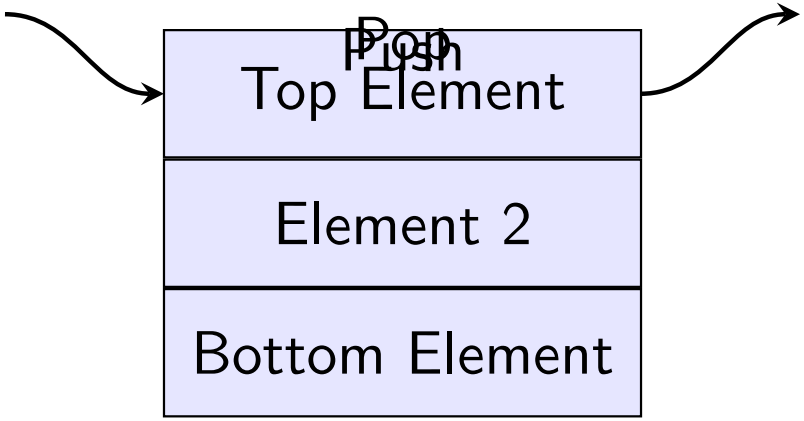
July 22, 2026

Today's Agenda

- Recap: What is a Stack?
- Why use Python Lists for Stacks?
- Mapping Stack operations to List methods
- Implementing Push and Pop
- Implementing Peek and isEmpty
- Full Stack Class Implementation
- Time Complexity Analysis

Recap: What is a Stack?

- A linear data structure following the LIFO principle
- LIFO stands for Last In, First Out
- Elements are added and removed from only one end, called the 'top'
- Real-world analogy: A stack of cafeteria plates, or the 'Undo' feature in a text editor



Why use Python Lists for Stacks?

- Python lists are dynamic arrays, meaning they can grow and shrink in size automatically
- They provide built-in methods that perfectly map to stack operations
- No need to manually manage memory or array resizing (unlike C or C++)
- It's the most straightforward and pythonic way to start building a stack

Python List vs Stack Operations

- Stack `push(item)` → List `append(item)`
- Stack `pop()` → List `pop()`
- Stack `peek()` or `top()` → List `list[-1]`
- Stack `isEmpty()` → Check if length is 0: `len(list) == 0` or `not list`

Stack Operation	Python List Equivalent
Push	`append(item)`
Pop	`pop()`
Peek/Top	`list[-1]`
isEmpty	`len(list) == 0`

Stack Initialization in Python

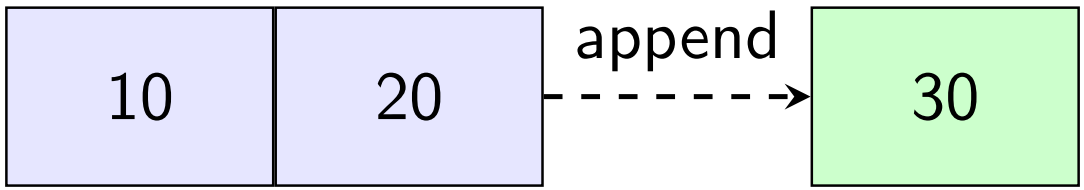
To create a stack, we simply initialize an empty list.

```
# Initialize an empty stack
my_stack = []

print("Initial stack:", my_stack)
```

Output: `Initial stack: []`

Implementing Push Operation

- The 'push' operation adds an element to the top of the stack.
- In our list implementation, the 'top' is the end of the list (the rightmost side).
- We use the list's `append(item)` method.
- This adds the item to the end of the list efficiently.



Code Example: Push Operation

```
stack = []

# Push elements onto the stack
stack.append(10)
stack.append(20)
stack.append(30)

print("Stack after pushes:", stack)
```

Output: Stack after pushes: [10, 20, 30]

Implementing Pop Operation

- The 'pop' operation removes and returns the element at the top of the stack.
- We use the list's built-in `pop()` method without any arguments.
- By default, `pop()` removes and returns the last element in the list.
- Crucial check: We must ensure the stack is not empty before calling `pop()` to avoid an IndexError.

Code Example: Pop Operation

```
stack = [10, 20, 30]

# Pop the top element
top_element = stack.pop()

print("Popped:", top_element)
print("Stack after pop:", stack)
```

Output:

```
Popped: 30
Stack after pop: [10, 20]
```

Handling Stack Underflow

```
stack = []

# Checking if empty before popping
if len(stack) > 0:
    top_element = stack.pop()
    print("Popped:", top_element)
else:
    print("Stack Underflow: The stack is empty!")
```

Output: Stack Underflow: The stack is empty!

Implementing Peek (or Top)

- The 'peek' operation returns the top element without removing it.
- We can access the top element using Python's negative indexing: `stack[-1]`.
- Index `-1` refers to the last element in the list.
- Like pop, we must check if the stack is empty first to avoid an IndexError.

Code Example: Peek Operation

```
stack = [10, 20, 30]

# Peek at the top element
if stack:  # Pythonic way to check if not empty
    top_element = stack[-1]
    print("Top element is:", top_element)
```

Output: `Top element is: 30`
(Notice the stack itself remains `[10, 20, 30]`)

Building a Stack Class (Object-Oriented)

- While we can just use a list directly, wrapping it in a Class is better practice.
- A class encapsulates the list and exposes ONLY the stack operations (push, pop, peek, isEmpty).
- This prevents users of the stack from accidentally doing invalid operations like inserting in the middle (`list.insert(0, item)`).
- It provides abstraction.

Code Example: Complete Stack Class

```
class Stack:
    def __init__(self):
        self.items = []

    def is_empty(self):
        return len(self.items) == 0

    def push(self, item):
        self.items.append(item)
```

Complete Stack Class (continued)

```
def pop(self):
    if not self.is_empty():
        return self.items.pop()
    raise IndexError("pop from empty stack")

def peek(self):
    if not self.is_empty():
        return self.items[-1]
    raise IndexError("peek from empty stack")
```

Time Complexity Analysis

- Push (`append`): O(1) amortized time. Adding to the end of a list is very fast.
- Pop (`pop()`): O(1) time. Removing from the end requires no shifting of other elements.
- Peek (`list[-1]`): O(1) time. Direct index access.
- isEmpty (`len()`): O(1) time.
- Conclusion: Python list is highly efficient for basic stack operations.

Operation	**Time Complexity**
Push	O(1) amortized
Pop	O(1)
Peek	O(1)
isEmpty	O(1)

Quick Check: Test Your Understanding

- Q: Which list method is used to implement a stack's push operation?
- Q: Why don't we use `insert(0, item)` for push and `pop(0)` for pop?
- Q: What index do we use to peek at the top element of the stack?

Key Takeaways

- Python lists provide an easy and efficient way to implement stacks.
- Stack operations map nicely: push $\rightarrow$ append(), pop $\rightarrow$ pop(), top $\rightarrow$ list[-1].
- All primary stack operations using this method have an O(1) time complexity.
- Encapsulating the list inside a custom Stack class is best practice to enforce LIFO rules.

Next Lecture Preview

Next time, we will explore practical applications of Stacks, including checking for balanced parentheses and evaluating expressions.

Questions?

Any Questions?

Lecture 2.3: Stack Applications: Expressions

Complete Lecture Deck – All 30 Lectures

Milav Dabgar

Lecturer, GP Palanpur

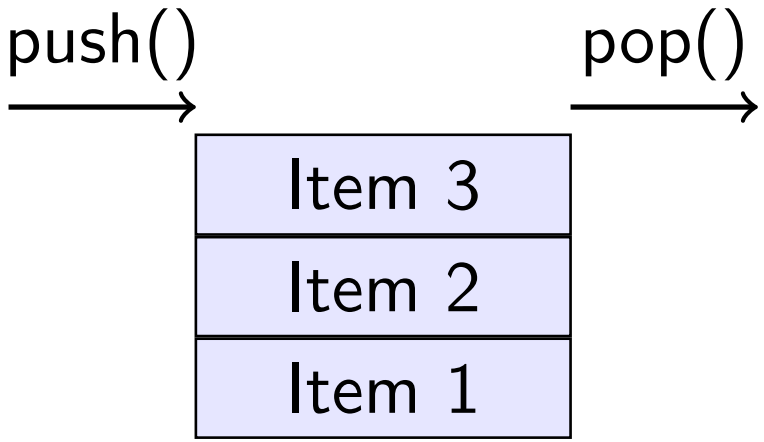
July 22, 2026

Today's Agenda

- ✓ What is an Expression?
- ✓ Infix, Prefix, and Postfix notations
- ✓ Why we need Postfix and Prefix
- ✓ Manual conversion of Infix to Postfix
- ✓ How Stacks are used in these conversions

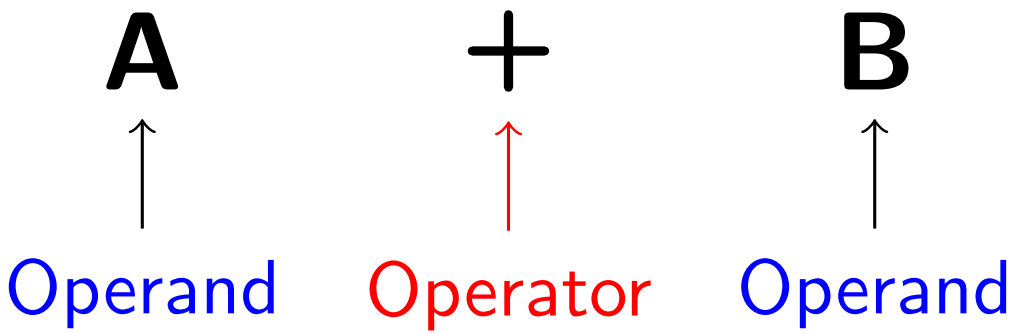
Recap: The Stack Data Structure

- Stack follows LIFO (Last In, First Out)
- Operations: push() adds an item, pop() removes the top item
- peek() views the top item without removing it
- We implemented it using Python lists: append() for push, pop() for pop



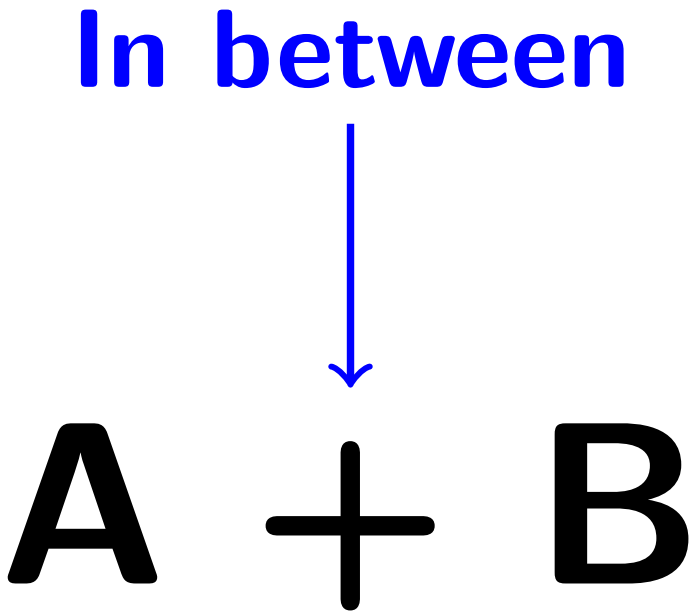
What is an Expression?

- An expression is a combination of operators and operands
- **Operands**: The data or variables (e.g., A, B, 5, 10)
- **Operators**: Symbols indicating operations (e.g., +, -, *, /, ^)
- Example: A + B



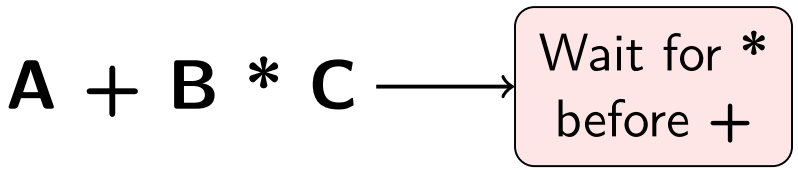
Types of Expressions: Infix

- Infix expression: Operator is placed between operands
- Format: Operand1 Operator Operand2
- Example: A + B
- This is how we write math naturally



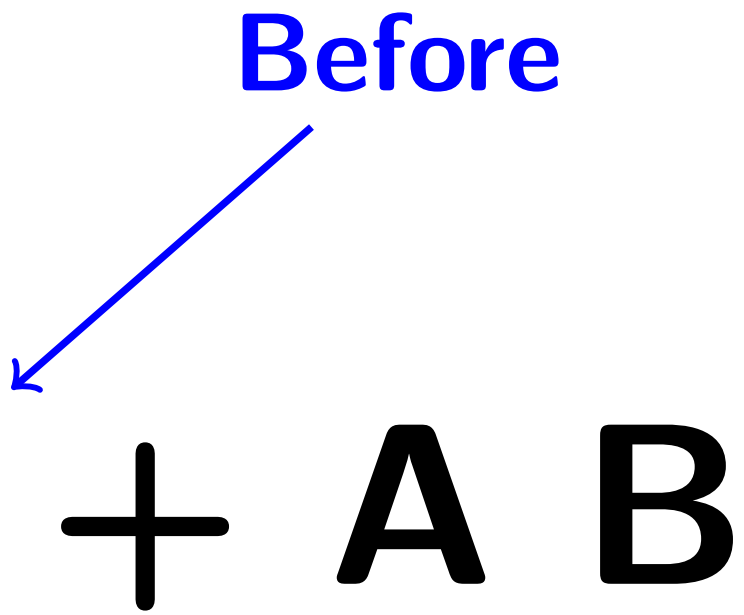
Limitations of Infix Expressions

- Requires knowledge of operator precedence (BODMAS / PEMDAS)
- Requires handling of parentheses to override precedence
- Computers find it difficult to evaluate left-to-right efficiently
- Example: A + B * C requires parsing the whole string before calculating B * C



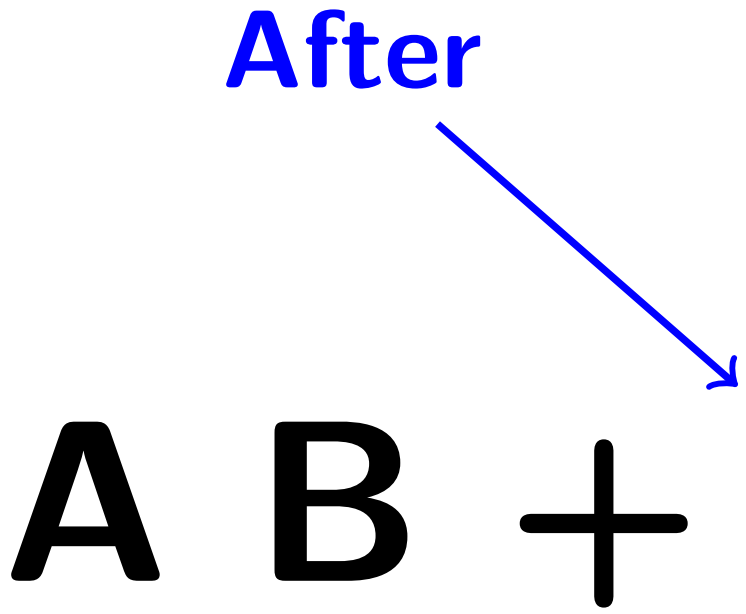
Types of Expressions: Prefix

- Prefix expression: Operator is placed before operands
- Format: Operator Operand1 Operand2
- Example: + A B
- Also known as Polish Notation (invented by Jan Łukasiewicz)



Types of Expressions: Postfix

- Postfix expression: Operator is placed after operands
- Format: Operand1 Operand2 Operator
- Example: A B +
- Also known as Reverse Polish Notation (RPN)



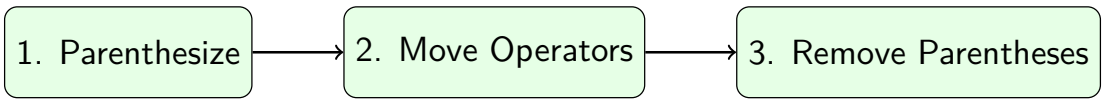
Why Prefix and Postfix?

- ✓ No need for parentheses!
- ✓ No need to remember operator precedence
- ✓ Unambiguous order of evaluation
- ✓ Can be easily evaluated by computers using a Stack in a single left-to-right pass

Infix (Complex)	**Postfix (Simple)**
((A + B) * (C - D)) / E	A B + C D - * E /

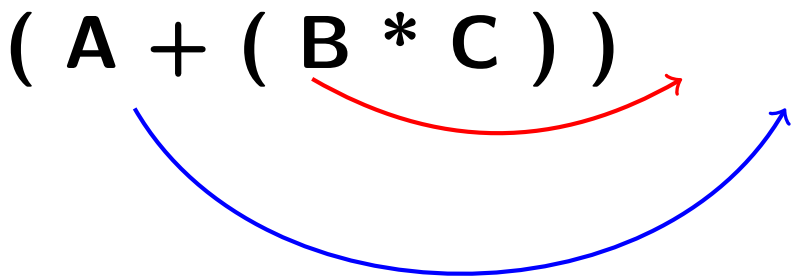
Manual Conversion: Infix to Postfix

- **Step 1:** Fully parenthesize the expression according to precedence
- **Step 2:** Move each operator to its corresponding right parenthesis
- **Step 3:** Remove all parentheses



Example: Infix to Postfix (Manual)

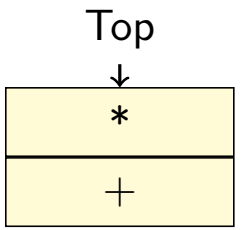
- **Infix:** A + B * C
- **Step 1:** Parenthesize → (A + (B * C))
- **Step 2:** Move operators → (A (B C)*)+
- **Step 3:** Remove parentheses → A B C * +



A B C * +

Role of Stack in Conversion

- While humans can look ahead, computers process left to right
- We use a Stack to hold operators until their right operand is processed
- Higher precedence operators are pushed on top of lower precedence ones
- If a lower precedence operator arrives, we pop the higher ones to the output



* has higher precedence

Algorithm: Infix to Postfix (Using Stack)

1. Scan expression left to right.
2. If operand, add to output.
3. If '(', push to stack.
4. If ')', pop from stack to output until '(' is found.
5. If operator, pop operators with $\geq$ precedence from stack to output, then push current operator.

Scan L to R

Operand $\rightarrow$ Output

Operator $\rightarrow$ Check Stack & Push/Pop

Quick Check: Test Your Understanding

- **Q:** Convert 'X - Y' to Postfix.
- **Q:** Why don't Postfix expressions need parentheses?
- **Q:** In a stack converting A + B * C, what happens when we read '*'?

Key Takeaways

- ✓ Expressions consist of operators and operands
- ✓ Infix is for humans; Postfix and Prefix are for machines
- ✓ Postfix doesn't require parentheses or precedence rules
- ✓ Stacks are heavily used to convert and process these expressions

Next Lecture Preview

- In Lecture 8, we will learn how to evaluate these postfix expressions to get a final result, and dive into Recursion!

Coming up: Postfix Evaluation & Recursion

Thank You!

Any Questions?

Lecture 8: Stack Applications: Evaluation & Recursion

Complete Lecture Deck – All 30 Lectures

Milav Dabgar

Lecturer, GP Palanpur

July 22, 2026

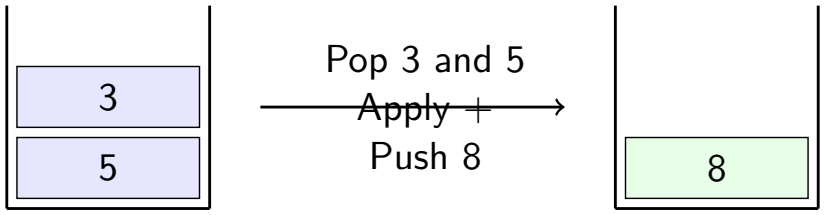
GTU Diploma Engineering - Semester 3 (32-ICT)
Unit 2: Stack and Queues

Today's Agenda

- Algorithm for Postfix Evaluation
- Python Code: Evaluating Postfix
- Introduction to Recursion
- The Role of Call Stack in Recursion
- Python Code: Factorial and Fibonacci

Evaluating Postfix Expressions

- Once an expression is in postfix format (e.g., '5 3 +'), how does a computer calculate it?
- We use a Stack to hold the operands (numbers).
- When an operator is encountered, we pop the required operands, apply the operator, and push the result back.



Algorithm: Postfix Evaluation

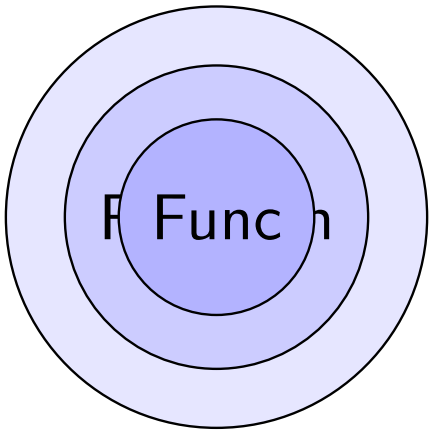
1. Scan the postfix string from left to right.
2. If character is an operand (number), push it to stack.
3. If character is an operator:
    1. Pop top element (operand2)
    2. Pop next element (operand1)
    3. Evaluate operand1 [operator] operand2
    4. Push the result back to stack.
4. When string ends, the final result is the only item left in the stack.

Python Code: Postfix Evaluation

```
def evaluate_postfix(expression):
    stack = []
    for char in expression:
        if char.isdigit():
            stack.append(int(char))
        else:
            val2 = stack.pop()
            val1 = stack.pop()
            if char == '+': stack.append(val1 + val2)
            elif char == '-': stack.append(val1 - val2)
            elif char == '*': stack.append(val1 * val2)
            elif char == '/': stack.append(val1 / val2)
    return stack.pop()
```

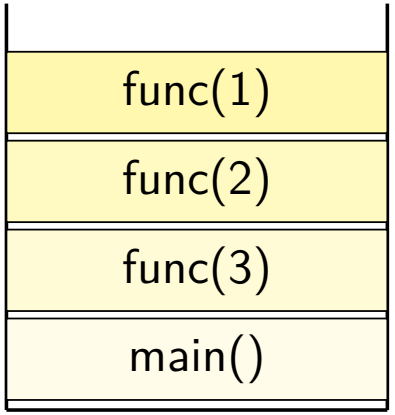
What is Recursion?

- Recursion is a programming technique where a function calls itself.
- It breaks down a large complex problem into smaller, similar sub-problems.
- Must have two parts:
  1. **Base Case**: The condition where recursion stops.
  2. **Recursive Case**: The part where the function calls itself with modified arguments.



Recursion and the Call Stack

- Why is recursion an application of Stacks?
- Behind the scenes, the OS/Python interpreter uses a 'Call Stack' to manage function calls.
- When a function calls itself, its current state (variables, return address) is PUSHED onto the call stack.
- When the base case is reached, functions return, and their states are POPPED from the stack.



Stack frames pushed and popped

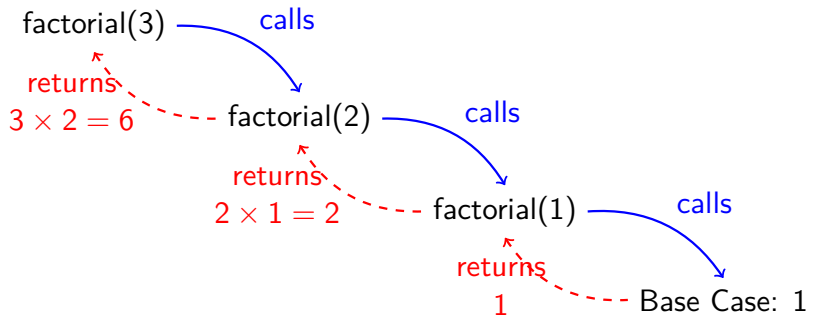
Example 1: Factorial Concept

- Factorial of N (written as N!) is the product of all positive integers less than or equal to N.
- $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$
- Notice the pattern: $5! = 5 \times 4!$
- Generally: $N! = N \times (N-1)!$
- Base Case: $0! = 1$ or $1! = 1$

Python Code: Recursive Factorial

```python
def factorial(n):
    # Base Case
    if n == 0 or n == 1:
        return 1
    # Recursive Case
    else:
        return n * factorial(n - 1)

print(factorial(5))  # Output: 120
```

Tracing the Factorial Call Stack



- factorial(3) $\rightarrow$ returns 3 * factorial(2)
- $\rightarrow$ returns 2 * factorial(1)
- $\rightarrow$ returns 1 (Base Case)
- $\leftarrow 2 \times 1 = 2$
- $\leftarrow 3 \times 2 = 6$

Example 2: Fibonacci Series

- A series where each number is the sum of the two preceding ones.
- Sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21...
- Formula: Fib(n) = Fib(n-1) + Fib(n-2)
- Base Cases:
  - Fib(0) = 0
  - Fib(1) = 1

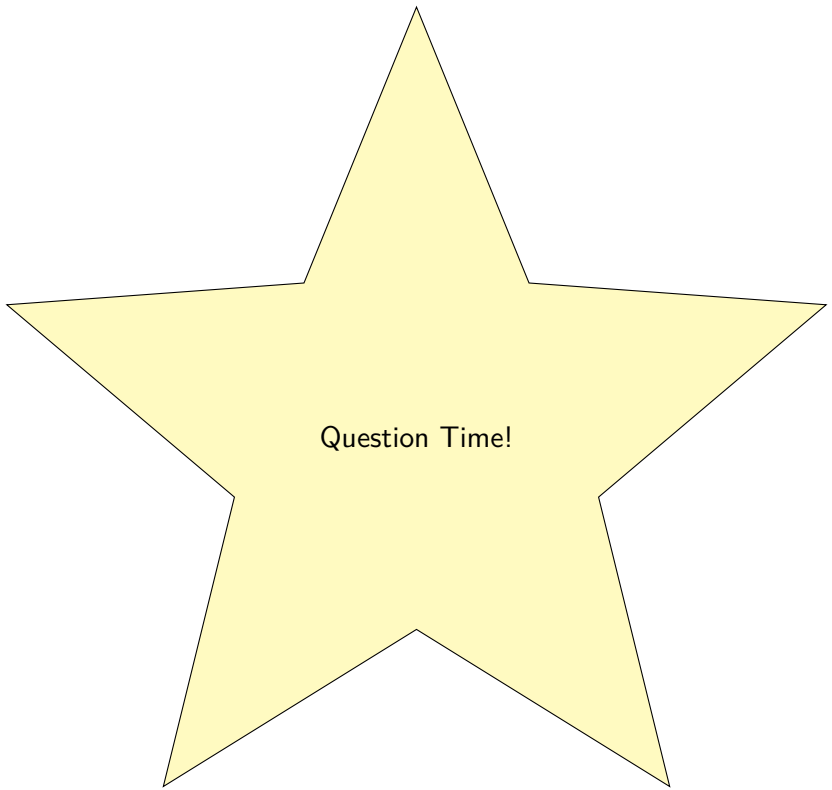
Index	0	1	2	3	4	5	6	7	8
Fib(n)	0	1	1	2	3	5	8	13	21

Python Code: Recursive Fibonacci

```
def fibonacci(n):
    # Base Cases
    if n == 0:
        return 0
    elif n == 1:
        return 1
    # Recursive Case
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)

print(fibonacci(6))  # Output: 8
```

Quick Check: Test Your Understanding

- Q: In postfix evaluation, which popped operand goes on the right side of the operator?
- Q: What happens if a recursive function has no base case?
- Q: What underlying data structure manages recursive function calls?

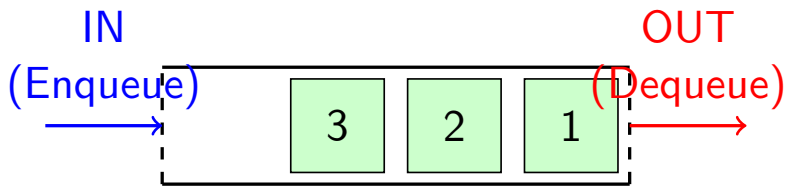


Key Takeaways

- Postfix evaluation uses a stack to hold operands and evaluate left-to-right
- Recursion is a function calling itself with a base case to terminate
- The system call stack manages memory and state for recursive calls
- Factorial and Fibonacci are classic problems elegantly solved with recursion

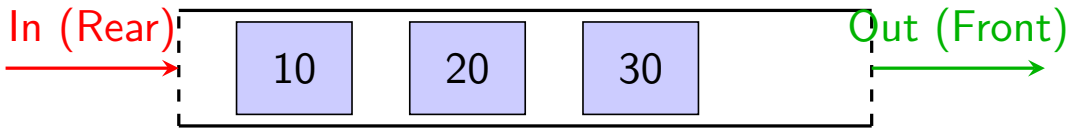
Next Lecture Preview

- In Lecture 9, we introduce a new Data Structure: the Queue.
- Instead of Last-In-First-Out, we will learn about First-In-First-Out (FIFO)!



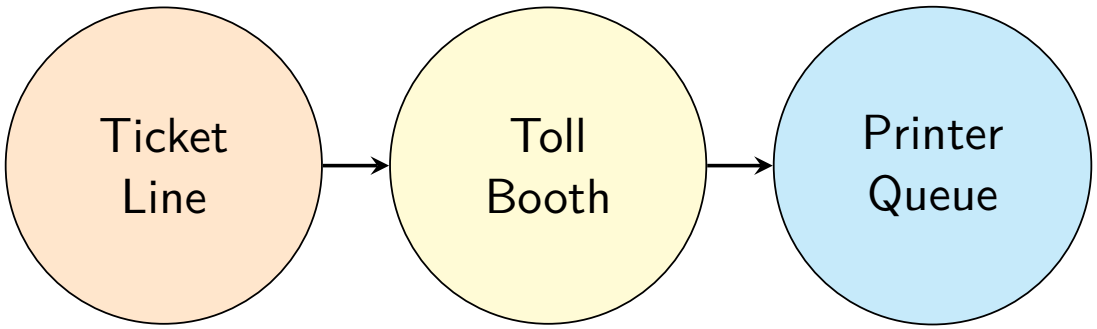
What is a Queue?

- A Queue is a linear data structure.
- It follows the First-In-First-Out (FIFO) principle.
- The element inserted first is the first one to be removed.
- Elements are added at one end and removed from the other.



Real-World Examples of Queues

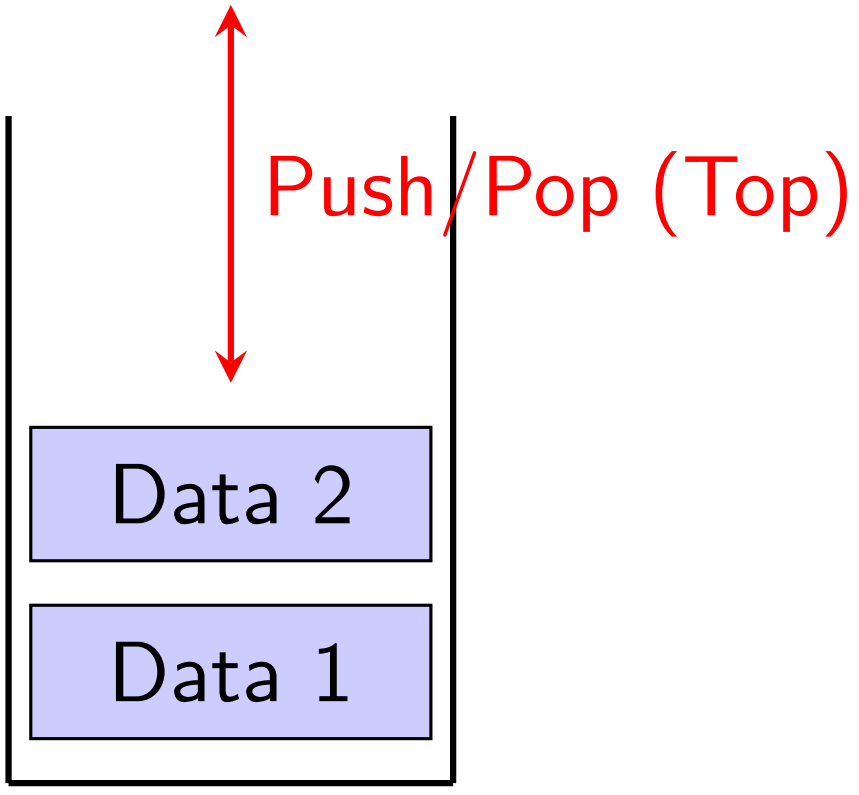
- People waiting in line for a movie ticket
- Cars waiting at a toll booth
- Printing documents: the first document sent is printed first
- Customer service calls waiting for an agent



Stack vs. Queue

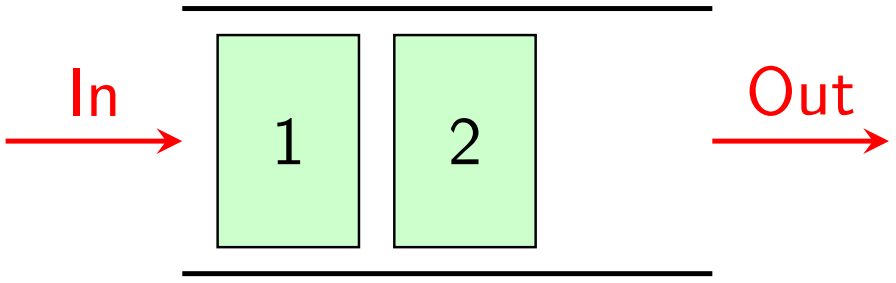
Stack:

- LIFO (Last In, First Out)
- Insertion and deletion happen at the SAME end (Top).



Queue:

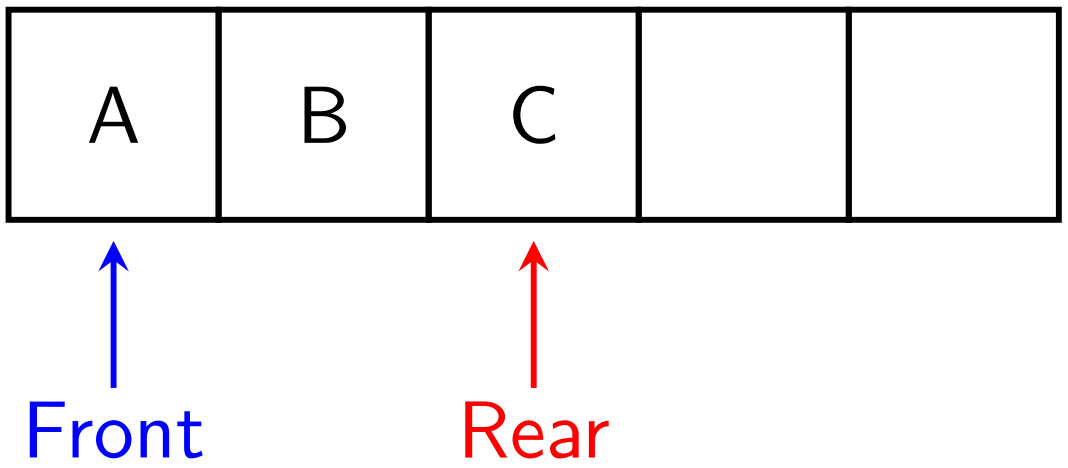
- FIFO (First In, First Out)
- Insertion happens at one end, deletion at the OTHER end.



Queue Terminology: Front and Rear

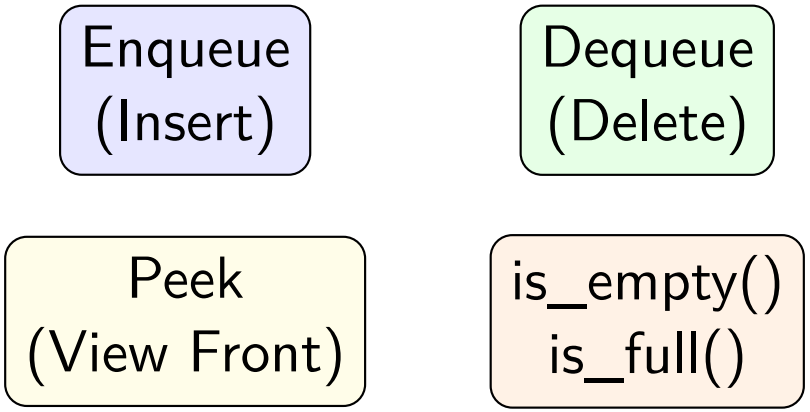
To manage a queue, we use two pointers/indices:

- **Front (or Head):** Points to the first element in the queue. This is where Deletion happens.
- **Rear (or Tail):** Points to the last element inserted. This is where Insertion happens.



Primary Operations

- **1. Enqueue:** Adding an element to the rear of the queue.
- **2. Dequeue:** Removing an element from the front of the queue.
- **3. Peek (or Front):** Viewing the front element without removing it.
- **4. is_empty():** Checking if the queue has no elements.
- **5. is_full():** Checking if the queue has reached its capacity.

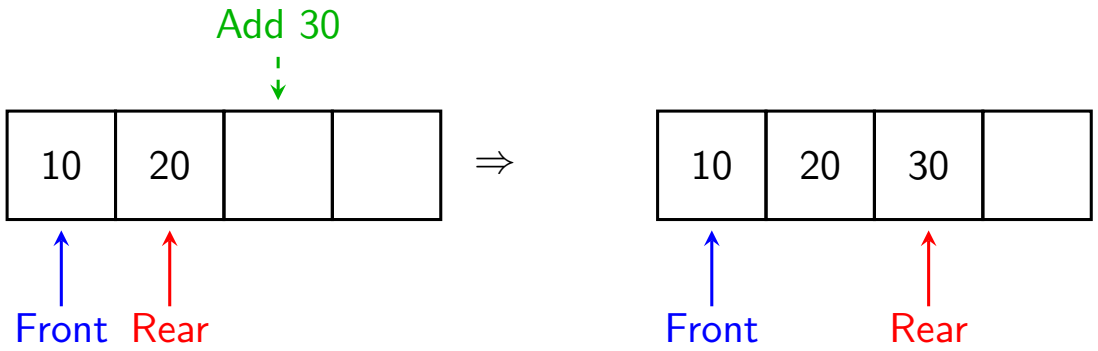


The Enqueue Operation

Enqueue means adding a new item.

Steps:

1. Check if the queue is full (Overflow condition).
2. If not full, increment the Rear pointer.
3. Add the new element at the position indicated by Rear.

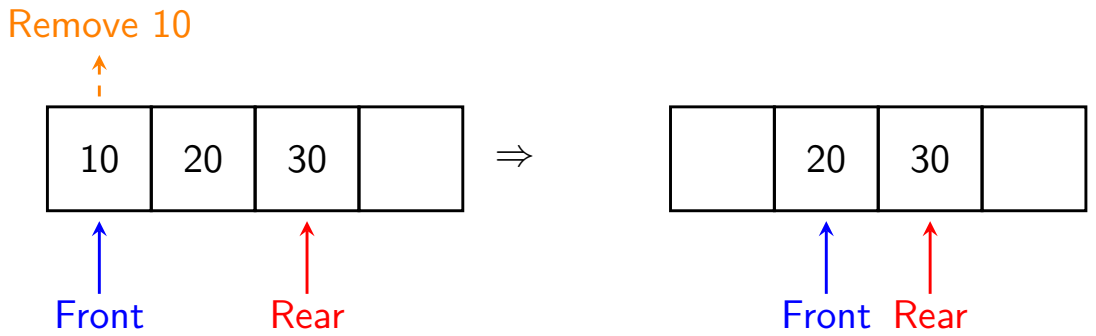


The Dequeue Operation

Dequeue means removing an item.

Steps:

1. Check if the queue is empty (Underflow condition).
2. If not empty, access the data at the Front pointer.
3. Increment the Front pointer to point to the next element.

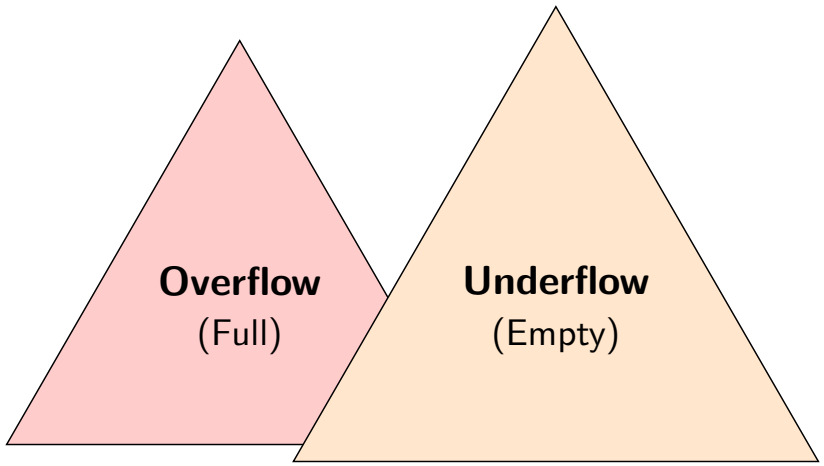


Visualizing Operations

Operation	Queue State	Front	Rear
Empty Queue	`[ ]`	-1	-1
Enqueue 'A'	`[A]`	0	0
Enqueue 'B'	`[A, B]`	0	1
Dequeue	`[_, B]` (A removed)	1	1

Edge Cases

- **Queue Overflow:**
  - Occurs when we try to Enqueue to a full queue.
  - In fixed-size arrays, Rear reaches the maximum size.
- **Queue Underflow:**
  - Occurs when we try to Dequeue from an empty queue.
  - Checked when Front $>$ Rear or queue size is 0.



Time Complexity (Ideal)

- **Enqueue: O(1)** - Adding to the end takes constant time.
- **Dequeue: O(1)** - Removing from the front takes constant time (if implemented correctly with pointers).
- **Peek: O(1)** - Just looking at the front item.

Operation	**Time Complexity**
Enqueue	$\mathcal{O}(1)$
Dequeue	$\mathcal{O}(1)$
Peek / Front	$\mathcal{O}(1)$
is_empty / is_full	$\mathcal{O}(1)$

Quick Check: Test Your Understanding

- **Q:** What principle does a Queue follow?
- **Q:** Where are items added, and where are they removed?
- **Q:** What error occurs if we dequeue an empty queue?

Key Takeaways

- Queue is a FIFO (First-In-First-Out) data structure
- Front points to the deletion end; Rear points to the insertion end
- Enqueue adds items to the Rear; Dequeue removes from the Front
- Overflow and Underflow are critical edge cases to handle

Next Lecture Preview

In Lecture 10, we will get our hands dirty and write Python code to implement this Queue, and discover a hidden flaw in single linear queues.

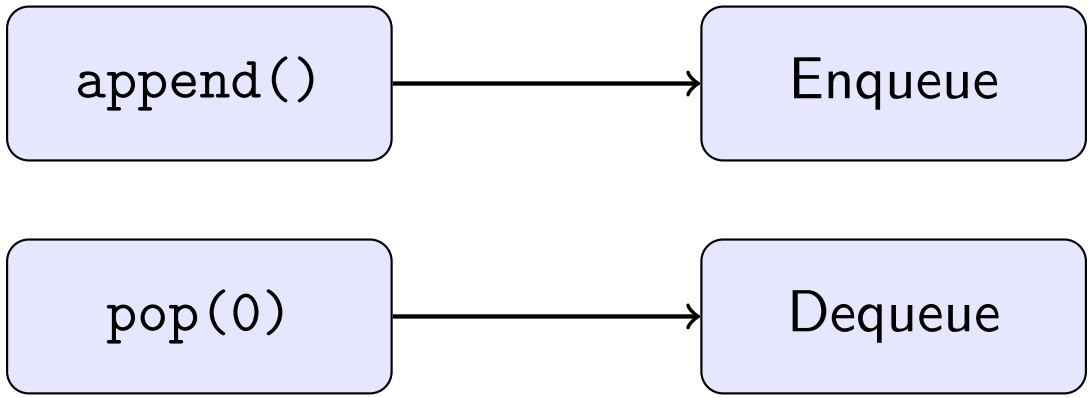
Python Implementation & Circular Queue Ahead!

Questions?

Any Questions?

Queue via Python Lists

- Python lists are dynamic arrays, making them easy to use for data structures.
- To Enqueue: We can use `list.append(item)` to add to the end (Rear).
- To Dequeue: We can use `list.pop(0)` to remove from the start (Front).
- Let's wrap this in a Class for clean Object-Oriented design.



Python Code: Queue Class Setup

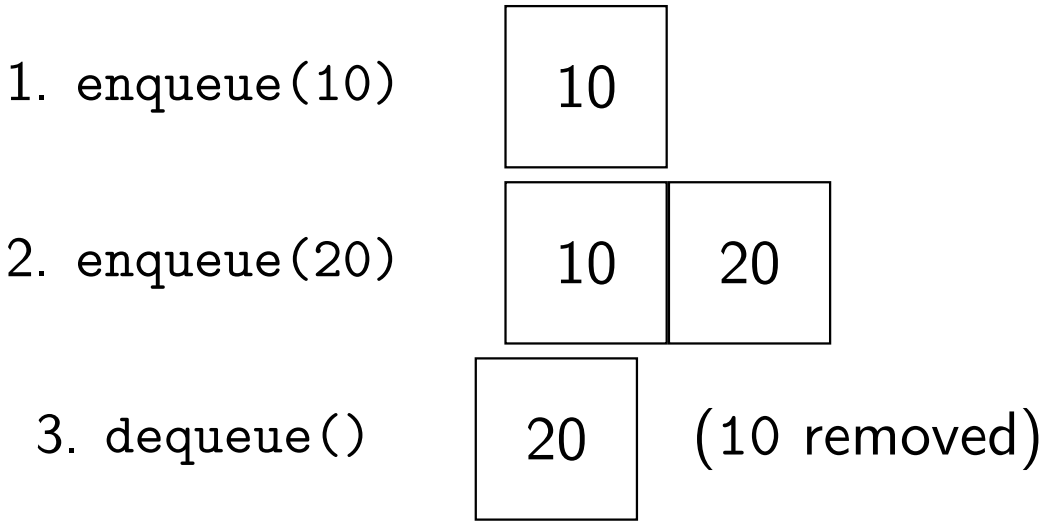
```
class Queue:
    def __init__(self):
        self.queue = []

    def is_empty(self):
        return len(self.queue) == 0

    def peek(self):
        if not self.is_empty():
            return self.queue[0]
        return 'Queue is empty'
```

Python Code: Enqueue & Dequeue

```
def enqueue(self, item):
    self.queue.append(item)
    print(f'Enqueued: {item}')

def dequeue(self):
    if self.is_empty():
        return 'Queue Underflow'
    return self.queue.pop(0)
```

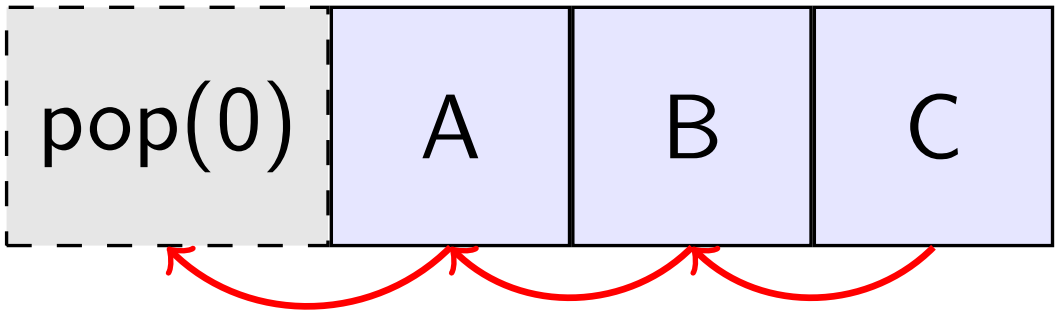
Tracing the Python Queue

- `q = Queue()`
- `q.enqueue(10)` → `[10]`
- `q.enqueue(20)` → `[10, 20]`
- `q.dequeue()` → Returns 10. List becomes `[20]`
- `q.enqueue(30)` → `[20, 30]`



The Hidden Cost: Time Complexity

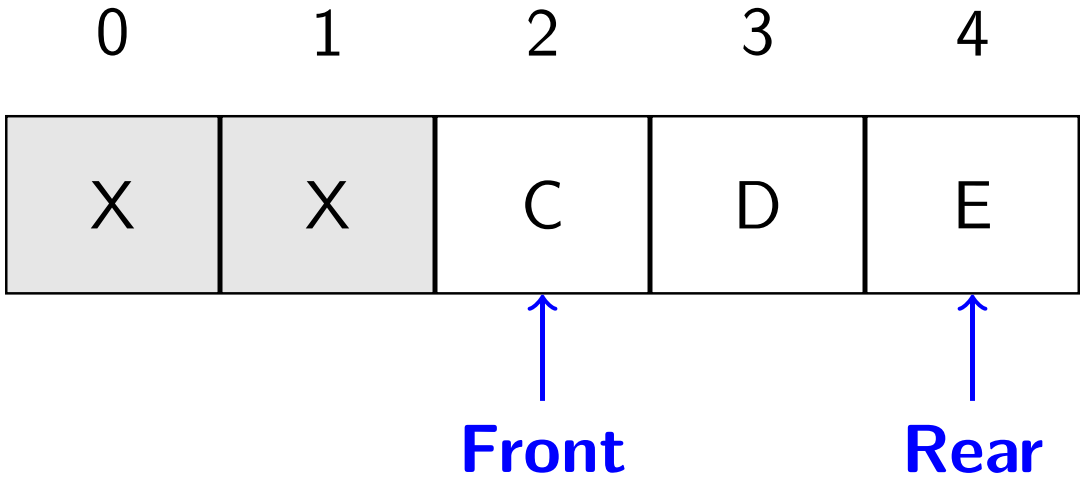
- Enqueue (`append`): O(1) Time. It just adds to the end.
- Dequeue (`pop(0)`): **O(N) Time!** Why?
- When index 0 is removed, Python must shift all remaining elements one position to the left.
- For a queue of 1,000,000 items, a single dequeue takes 1,000,000 shift operations!



Elements shift left $O(N)$

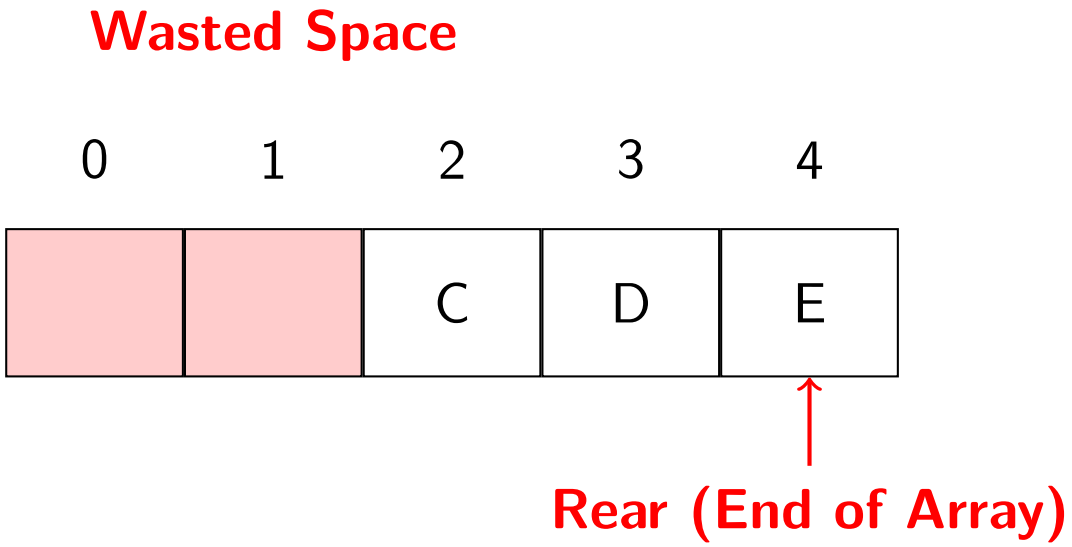
Limitation of Fixed-Size Single Queue

- In lower-level languages (like C) or fixed-size arrays, we manage Front and Rear pointers explicitly.
- Imagine an Array of Size 5.
- If we enqueue 5 items: Rear reaches index 4 (Full).
- If we dequeue 2 items: Front moves to index 2.



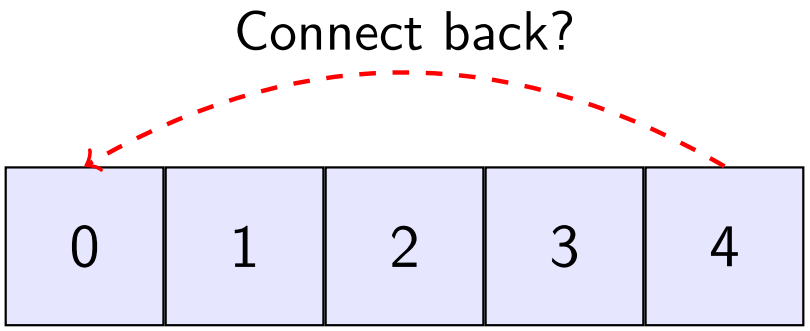
The Wasted Space Problem

- Array = `[_, _, C, D, E]` (Indices 0, 1 are empty; 2, 3, 4 are full)
- `Front = 2`, `Rear = 4`
- What happens if we try to Enqueue 'F'?
- Rear is at Max Size! It says **Queue Overflow**.
- BUT we have two empty spaces at the beginning!



Why not just shift elements?

- We could shift elements to the left (like Python does).
- But shifting takes O(N) time, making the queue very slow.
- We need a way to reuse the empty spaces at the front WITHOUT shifting elements.
- How can we connect the end of the array back to the beginning?



Summary of Limitations

Key Limitations

1. **Dynamic lists (Python):** Using `pop(0)` is slow (O(N) shifting).
2. **Fixed arrays (C/C++):** Managing with pointers leads to wasted space (**False Overflow**).

Single Linear Queues are inefficient for memory reuse.

Quick Check: Test Your Understanding

- **Q1:** Why is `pop(0)` slow in Python?
- **Q2:** In a fixed single queue, when does False Overflow occur?

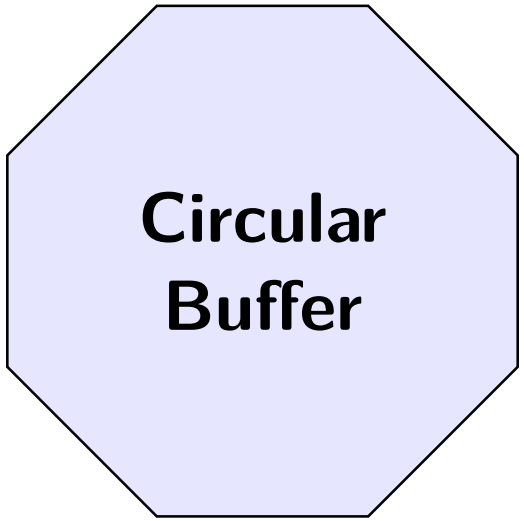
Key Takeaways

- Queues can be implemented in Python using a Class and a list.
- Enqueue is O(1) via `append`.
- Dequeue via `pop(0)` is O(N) due to memory shifting.
- Fixed-size simple queues suffer from memory wastage (False Overflow).

Next Lecture Preview

Lecture 11

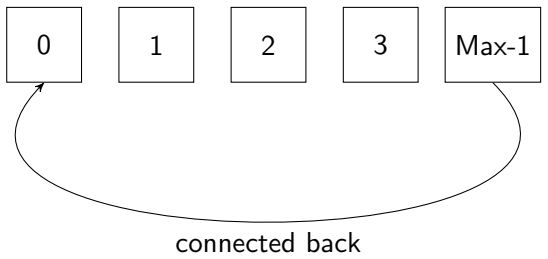
In Lecture 11, we will solve the 'wasted space' problem by bending our queue into a circle: **The Circular Queue!**



Questions?

What is a Circular Queue?

- A Circular Queue is a linear data structure that follows FIFO, but the last position is connected back to the first position.
- Also known as a Ring Buffer.
- It overcomes the 'False Overflow' problem of a simple queue by reusing empty spaces at the front.



The Magic of Modulo Arithmetic

- How do we make Rear jump from index (Size - 1) back to 0?
- We use the modulo operator (%)!
- Next Position = (Current Position + 1) % Size
- Example (Size = 5):
- If Current = 3, Next = (3 + 1) % 5 = 4 % 5 = 4
- If Current = 4, Next = (4 + 1) % 5 = 5 % 5 = 0 (Wraps around!)

Circular Enqueue Logic

1. Check if full: `(Rear + 1) % Size == Front`
2. If empty (Front = -1), set Front = 0, Rear = 0.
3. Else, move Rear: `Rear = (Rear + 1) % Size`
4. Insert item at `Queue[Rear]`

Circular Dequeue Logic

1. Check if empty: `Front == -1`
2. Get data from `Queue[Front]`
3. If Front == Rear (only one element was left), set Front = -1, Rear = -1 (Reset).
4. Else, move Front: `Front = (Front + 1) % Size`

Python Implementation Snippet

```
class CircularQueue:
    def __init__(self, size):
        self.size = size
        self.queue = [None] * size
        self.front = self.rear = -1

    def is_full(self):
        return (self.rear + 1) % self.size == self.front

    def is_empty(self):
        return self.front == -1
```

Python Snippet: Enqueue

```
def enqueue(self, item):
    if self.is_full():
        print('Queue is Full')
        return
    if self.front == -1:
        self.front = self.rear = 0
    else:
        self.rear = (self.rear + 1) % self.size
    self.queue[self.rear] = item
```

Tracing Circular Queue

Action	Queue State	Front	Rear
Initial	Size=3	-1	-1
Enq A	[A, _, _]	0	0
Enq B	[A, B, _]	0	1
Deq	[_, B, _]	1	1
Enq C	[_, B, C]	1	2
Enq D	[D, B, C]	1	0

Simple Queue vs Circular Queue

- Let's compare them directly to understand the exam question:
- 'Differentiate circular queue and simple queue'

Difference: Memory Utilization

Simple Queue	**Circular Queue**
Memory is wasted. Dequeued spaces at the front cannot be reused.	Memory is highly utilized. Dequeued spaces can be reused as Rear wraps around.

Difference: Full Condition

Simple Queue	**Circular Queue**
Queue is full when `Rear == Size - 1` (even if Front > 0).	Queue is full when `(Rear + 1) % Size == Front`.

Difference: Complexity of Implementation

Simple Queue		**Circular Queue**
Easier to implement. pointer increments.	Linear	Slightly complex implementation involving modulo arithmetic.

Quick Check: Test Your Understanding

- Q: In a Circular Queue of size 5, if Rear is 4, what is the next Rear value after enqueue?
- Q: True or False: Circular Queues solve the Overflow problem entirely.
- Q: What happens when Front == Rear in a Circular Queue Dequeue?

Key Takeaways

- Circular Queues reuse empty spaces by wrapping the Rear pointer to the front
- Modulo arithmetic `(index + 1) % size` is used for pointer movement
- It solves the 'wasted space / false overflow' issue of simple fixed queues
- Simple queues are easier to build, but circular queues are memory efficient

Next Lecture Preview

- In Lecture 12, we will look at real-world computing applications of Queues and wrap up Unit 2!

Any Questions?

Lecture 2.6: Application of queue

Complete Lecture Deck – All 30 Lectures

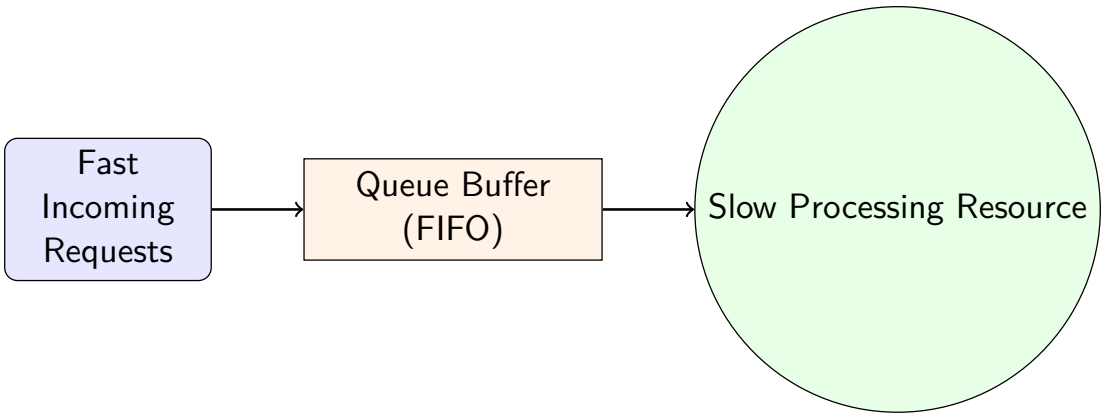
Milav Dabgar

Lecturer, GP Palanpur

July 22, 2026

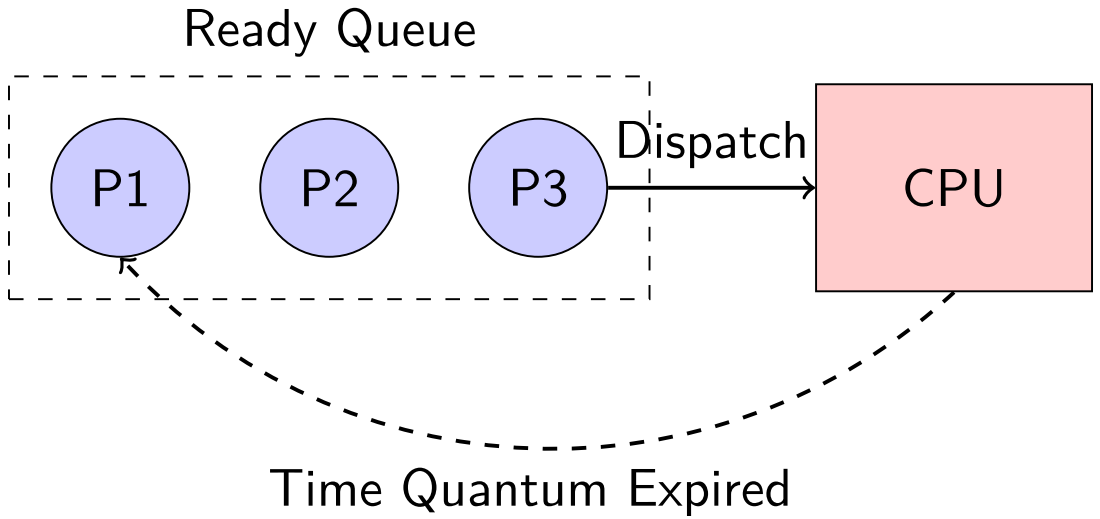
Why Systems Need Queues

- Any scenario involving shared resources needs Queues.
- When requests come in faster than a resource can process them, they must wait.
- FIFO ensures fairness: the oldest request is served first.
- Queues act as 'Buffers' to hold data smoothly.



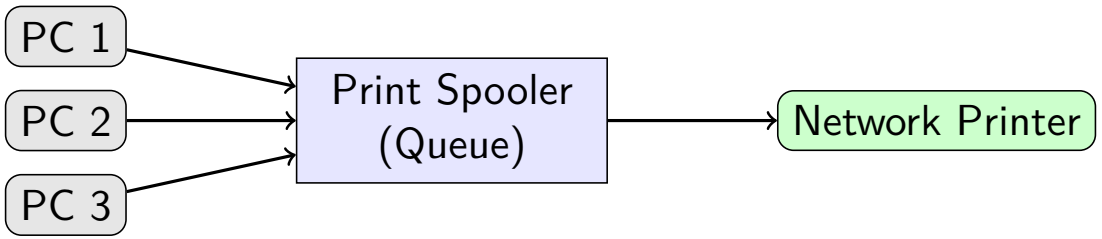
Application 1: CPU Task Scheduling

- Your OS runs multiple programs (browser, music player, IDE).
- The CPU can only execute one task at a time (per core).
- Tasks are placed in a 'Ready Queue'.
- The OS scheduler picks the task at the Front, runs it, and may push it to the Rear if not finished (Round Robin).



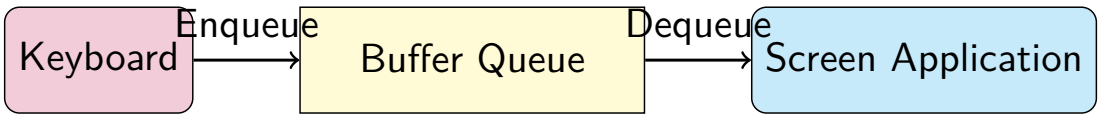
Application 2: Print Spooling

- Printers are very slow compared to CPUs.
- If 5 users send documents to a network printer, the OS creates a Print Spooler (a Queue).
- Documents are Enqueued as they arrive.
- The printer Dequeues and prints them one by one in order.



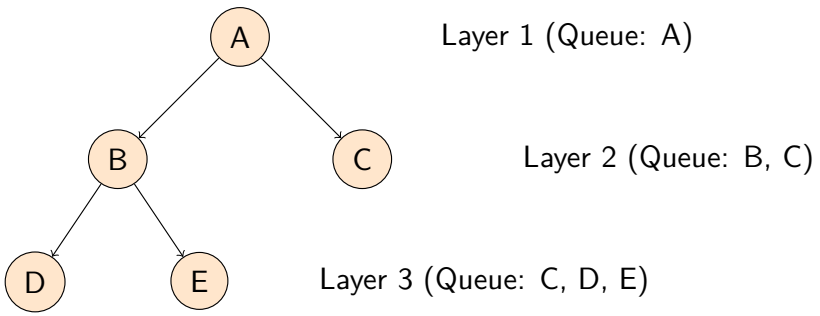
Application 3: I/O Buffers

- Keyboard input: When you type fast, characters go into a keyboard buffer (Queue).
- The word processor dequeues them to display on screen.
- Streaming video: Video frames are downloaded into a buffer (Queue) and dequeued to the screen for smooth playback.



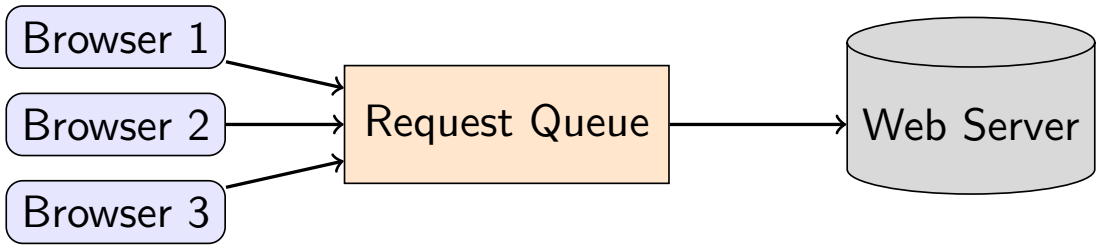
Application 4: Breadth-First Search (BFS)

- A core algorithm used in AI, pathfinding, and networking.
- To explore a graph/maze layer by layer, we use a Queue.
- Add a node, explore its neighbors, add them to the rear.
- Keeps track of nodes to visit in a FIFO manner.



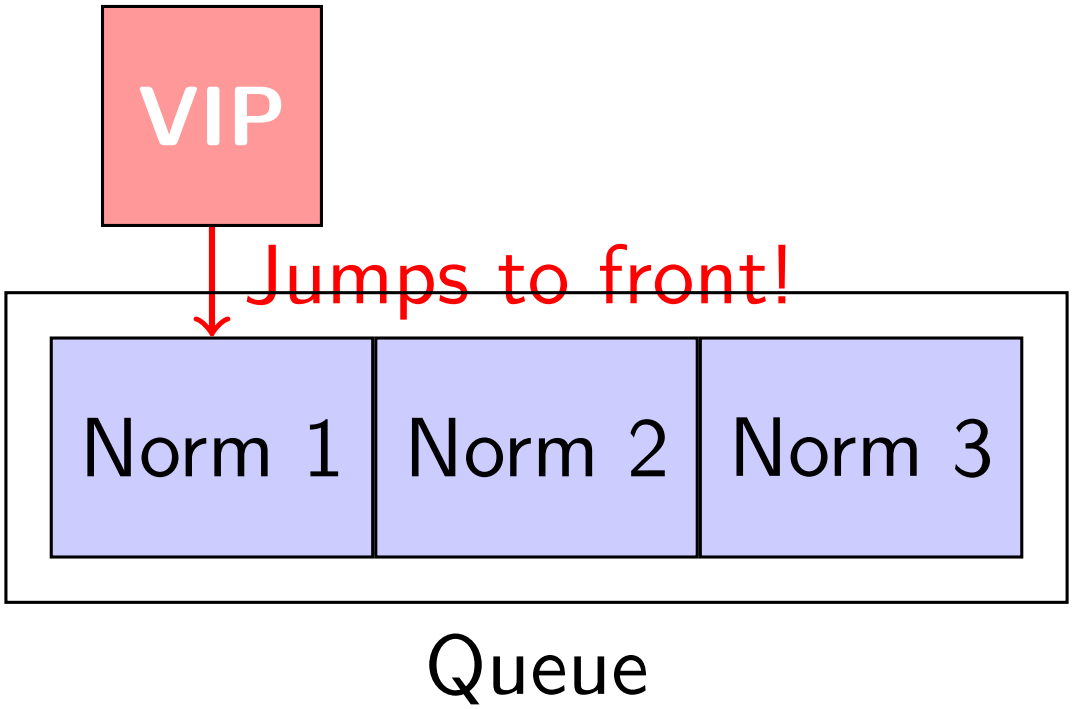
Application 5: Web Server Requests

- When thousands of users click a website at once, the server cannot serve them all instantly.
- Incoming HTTP requests are placed in a Queue.
- Worker threads dequeue requests and serve HTML pages.
- If the queue overflows, you get a 'Server Too Busy' error!



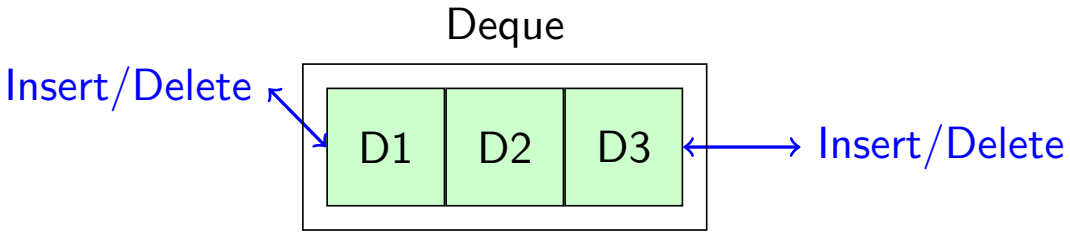
Advanced Queue: Priority Queue

- Sometimes FIFO isn't enough. VIPs need to jump the line.
- Priority Queue: Elements are dequeued based on priority, not just arrival time.
- Example: VIP hospital emergency cases, or High-Priority system interrupts in OS.



Advanced Queue: Deque (Double Ended Queue)

- Deque: Insertion and Deletion can happen at BOTH Front and Rear ends.
- It can act as both a Stack AND a Queue depending on how you use it.
- Used in browser history, undo operations, and advanced algorithms.



Python's Built-in Queue: collections.deque

```python
from collections import deque

# Fast O(1) queue operations in Python
q = deque()
q.append('A')       # Enqueue Rear
q.append('B')
item = q.popleft() # Dequeue Front (O(1) time!)
print(item)         # Prints 'A'
```

Efficiency

Using `list` for queues in Python is slow ($O(N)$ for `pop(0)`). Always use `collections.deque` for $O(1)$ operations!

Unit 2 Recap: Stacks & Queues

- **Stacks:** LIFO. Push/Pop at one end. Used for Expressions, Recursion, Undo/Redo.
- **Queues:** FIFO. Front/Rear pointers. Used for Scheduling, Buffers, Printers.
- Circular Queues optimize memory space.
- Both are fundamental building blocks of computing systems.

Key Takeaways

- Queues manage shared resources and asynchronous data transfers
- CPU scheduling and Print Spooling are classic Queue applications
- Priority Queues and Deques offer advanced behaviors beyond strict FIFO
- Python provides `collections.deque` for efficient O(1) queue operations

Next Lecture Preview

Unit 2 complete! Next, we begin Unit 3 and dive into Linked Lists - a truly dynamic data structure.

Questions?

Any Questions?

Lecture 3.1: Introduction to Linked Lists

Complete Lecture Deck – All 30 Lectures

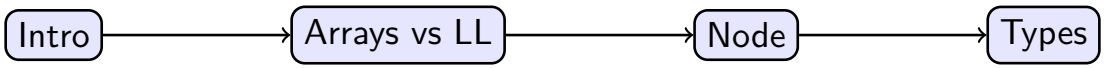
Milav Dabgar

Lecturer, GP Palanpur

July 22, 2026

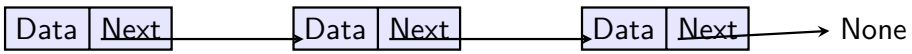
Today's Agenda

- Introduction: What is a Linked List?
- Arrays vs. Linked Lists: Why do we need them?
- Anatomy of a Node: Data and Reference
- Python Code: Implementing a Node
- Types of Linked Lists (Singly, Doubly, Circular)



What is a Linked List?

- A linked list is a linear data structure.
- Unlike arrays, elements are NOT stored in contiguous memory locations.
- Elements are linked using pointers (references in Python).
- The list is formed by a sequence of connected 'Nodes'.



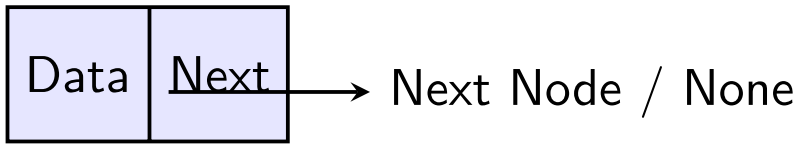
Why Linked Lists? (Arrays vs. Linked Lists)

- **Arrays:** Fixed size (usually), contiguous memory, expensive insertions/deletions.
- **Linked Lists:** Dynamic size, non-contiguous memory, fast insertions/deletions (if location is known).
- **Drawback of Linked Lists:** Random access is not allowed (no index-based access like arr[5]).

Feature	**Array**	**Linked List**
Size	Fixed	Dynamic
Memory	Contiguous	Non-contiguous
Insertion	Slow	Fast
Random Access	Yes $O(1)$	No $O(n)$

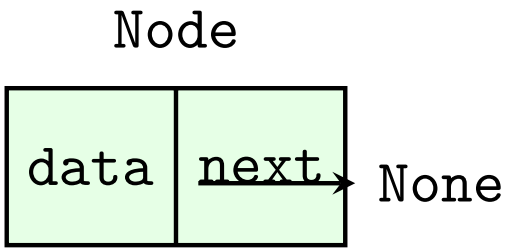
Anatomy of a Node

- A node is the fundamental building block of a linked list.
- It contains two parts:
  1. **Data:** The actual value or information stored.
  2. **Reference (Next):** A pointer/reference to the next node in the sequence.
- The last node points to 'None' (indicating the end of the list).



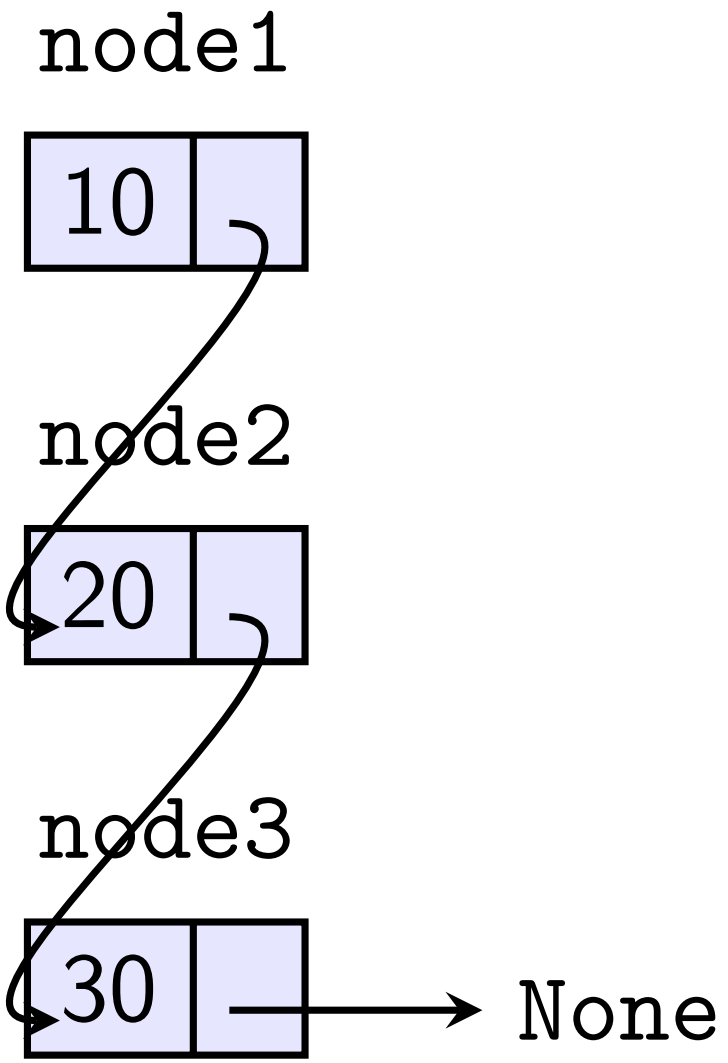
Code Example: Implementing a Node in Python

```
class Node:
    def __init__(self, data):
        self.data = data        # Assign data
        self.next = None        # Initialize next as null
```



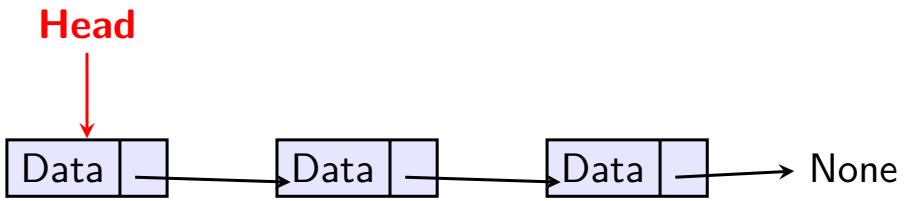
Creating and Linking Nodes

```
# Create nodes
node1 = Node(10)
node2 = Node(20)
node3 = Node(30)

# Link nodes
node1.next = node2
node2.next = node3
```



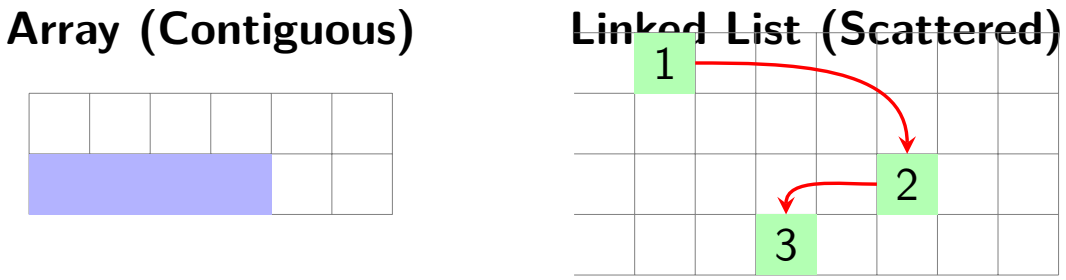
The 'Head' of the List

- To access a linked list, we only need to keep track of the first node.
- This reference is conventionally called the 'Head'.
- If Head is None, the list is empty.
- We can traverse the entire list starting from the Head and following the 'next' references.



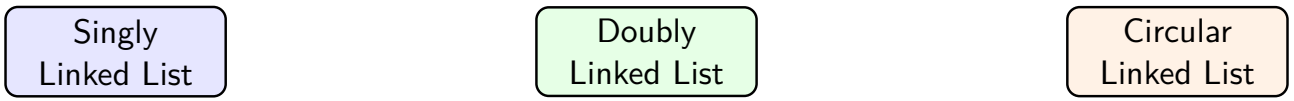
Memory Allocation Visualized

- **Array:** Stored in one continuous block of RAM.
- **Linked List:** Nodes are scattered across RAM wherever space is available.
- The OS manages this seamlessly, and the 'next' references stitch them together logically.



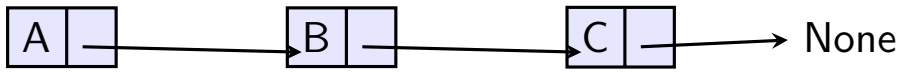
Types of Linked Lists

- There are three main variations of linked lists:
  1. Singly Linked List
  2. Doubly Linked List
  3. Circular Linked List
- Each type serves different use cases and has different memory requirements.



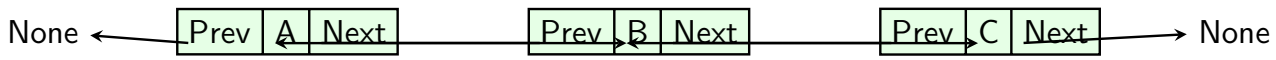
1. Singly Linked List

- The simplest type of linked list.
- Navigation is forward only.
- Each node has data and one reference (next).
- The last node points to None.



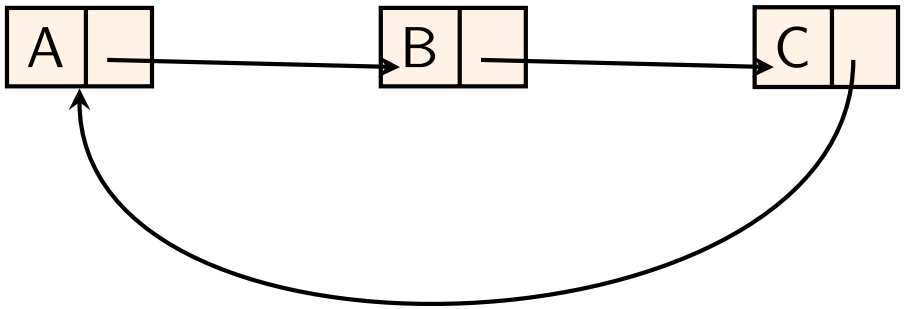
2. Doubly Linked List

- Navigation is possible in both directions (forward and backward).
- Each node has data and TWO references: 'next' and 'prev'.
- Requires more memory per node.
- Easier to delete a node if you only have a pointer to it.



3. Circular Linked List

- The last node points back to the first node (Head) instead of None.
- Forms a circle.
- Can be implemented as Singly Circular or Doubly Circular.
- Useful for round-robin scheduling algorithms or multiplayer games.



Quick Check

- **Q1:** What are the two main components of a standard Node?
- **Q2:** Why can't we access the 5th element of a linked list in O(1) time like an array?
- **Q3:** Which type of linked list allows backwards traversal?

Key Takeaways

- Linked Lists are dynamic data structures using non-contiguous memory.
- They are made of Nodes (Data + Next reference).
- They are better for frequent insertions/deletions, but worse for random access compared to arrays.
- Three main types: Singly, Doubly, and Circular.

Next Lecture Preview

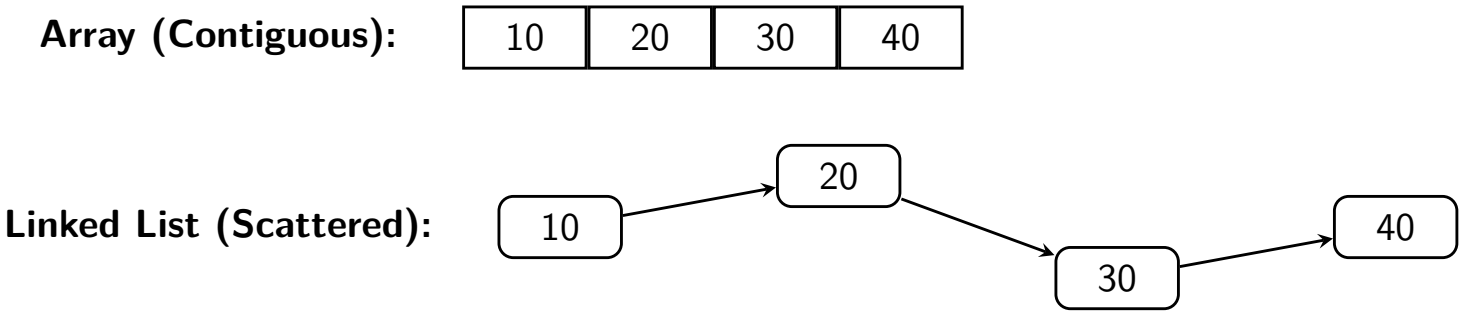
- In the next lecture, we will dive deeper into Singly Linked Lists.
- We will learn how to implement a full LinkedList class in Python.
- We'll write code to insert nodes at the beginning, end, and middle!

Questions?

Any Questions?

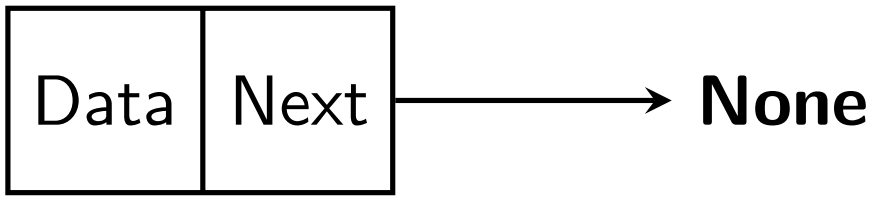
Recap: What is a Singly Linked List?

- A linear data structure made of 'nodes'.
- Unlike arrays, elements are not stored in contiguous memory locations.
- Each node points to the next node in the sequence.
- The list is tracked by a 'head' reference pointing to the first node.



Anatomy of a Node

- A node in a singly linked list contains two parts:
- 1. Data: The actual value or information stored (integer, string, object, etc.).
- 2. Next pointer (or reference): A link to the next node in the list.
- The last node points to 'None', indicating the end of the list.

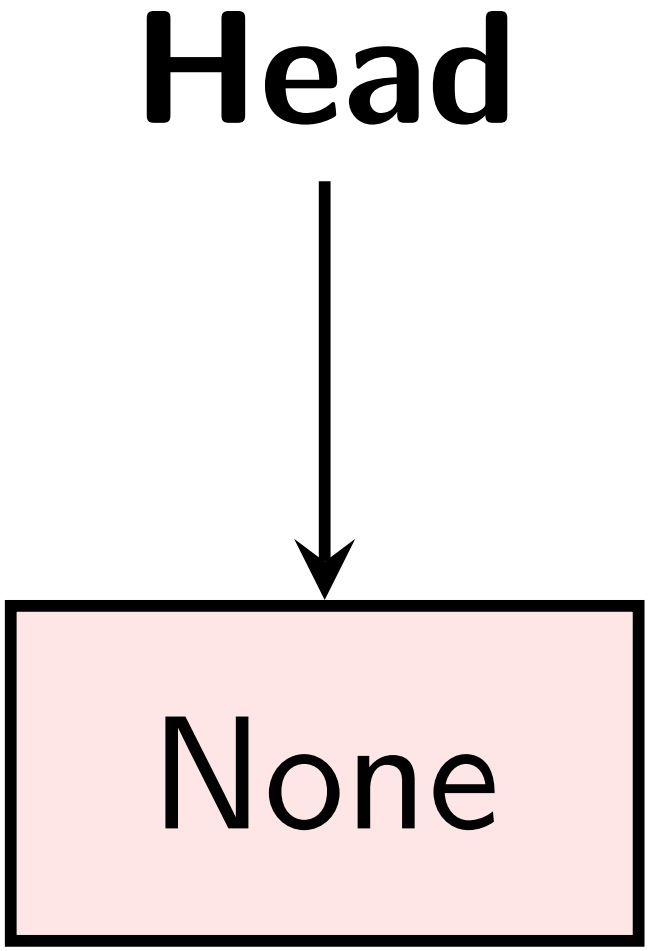


Creating a Node in Python

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

# Creating an instance
first_node = Node(10)
print(first_node.data) # Output: 10
print(first_node.next) # Output: None
```

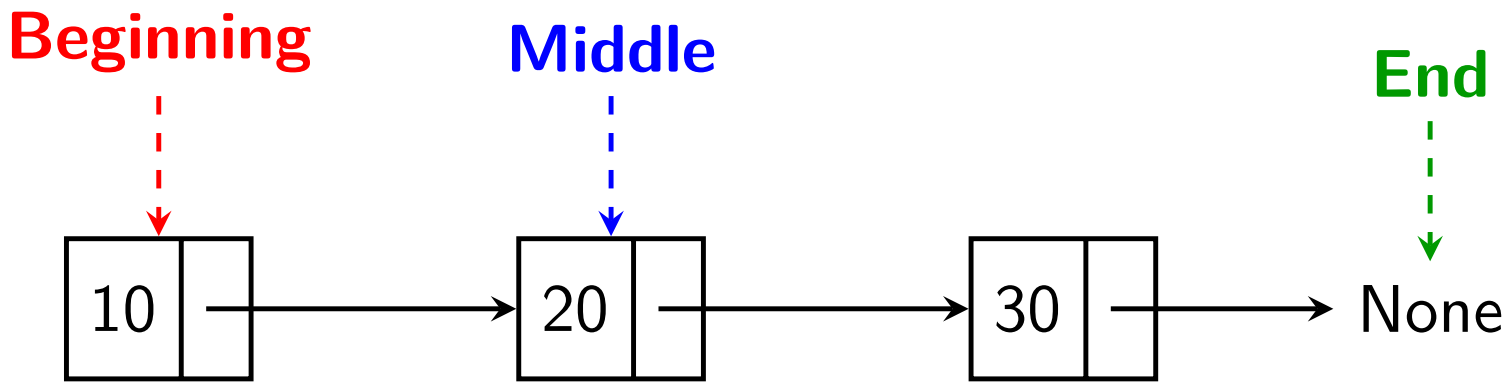
The LinkedList Class Structure

```
class LinkedList:
    def __init__(self):
        self.head = None

# Instantiating an empty list
my_list = LinkedList()
print(my_list.head) # Output: None
```



Insertion Operation

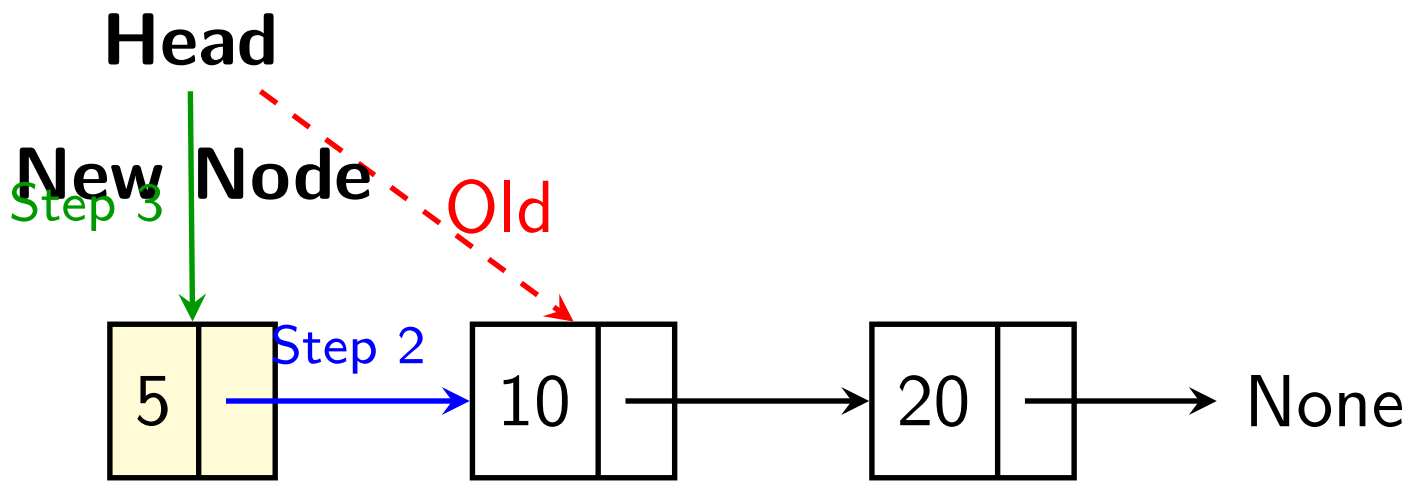
- Insertion means adding a new node to the list.
- There are three common places to insert:
- 1. At the beginning (Head)
- 2. At the end (Tail)
- 3. At a specific position (Middle)
- Today, we will focus on inserting at the beginning and at the end.



Inserting at the Beginning: Concept

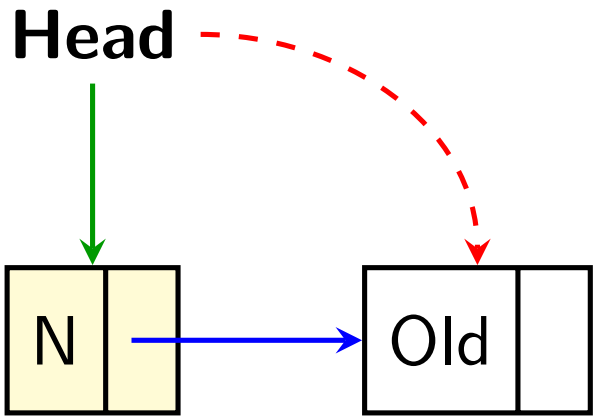
To insert a new node at the front (head) of the list:

- Step 1: Create the new node.
- Step 2: Point the new node's 'next' to the current 'head'.
- Step 3: Update the 'head' to point to the new node.



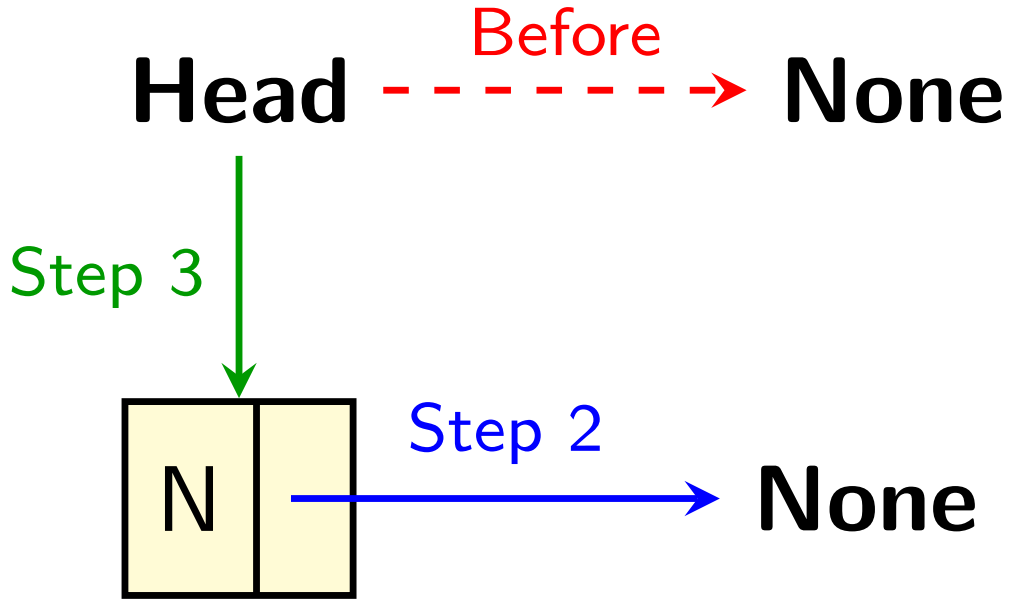
Inserting at the Beginning: Python Code

```
def insert_at_beginning(self, new_data):
    new_node = Node(new_data) # Step 1
    new_node.next = self.head # Step 2
    self.head = new_node      # Step 3
```



Inserting at the Beginning: Edge Case

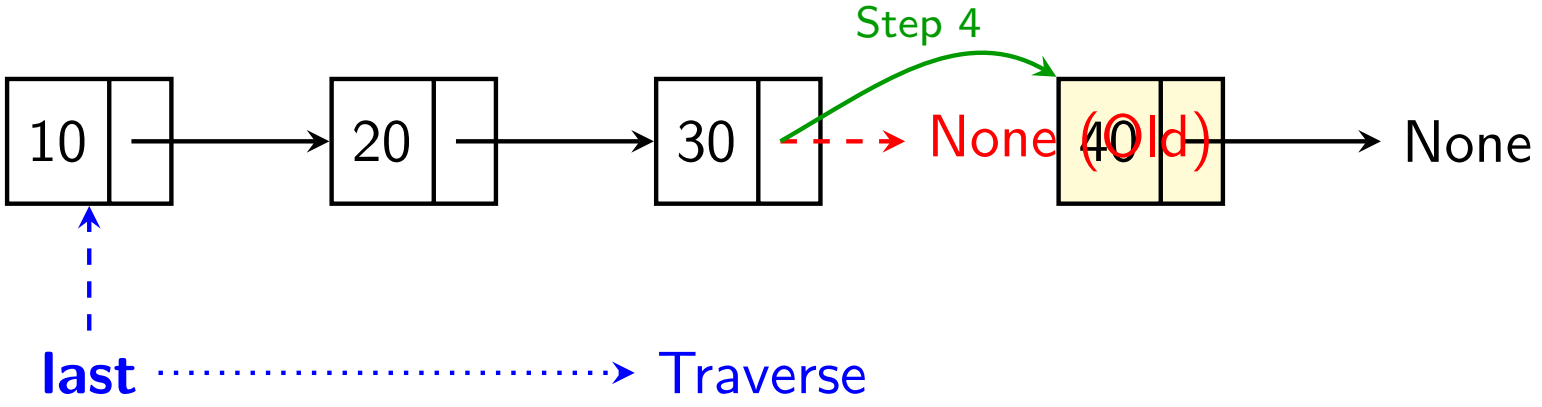
- What if the list is completely empty (head is None)?
- The code still works perfectly!
- `new_node.next` becomes `None` (which is correct, it's the only node).
- `self.head` becomes `new_node` (which is correct, it's now the first node).



Inserting at the End: Concept

To insert at the tail of the list:

- Step 1: Create the new node.
- Step 2: If the list is empty, make it the head.
- Step 3: Otherwise, traverse the list starting from head to find the last node.
- Step 4: Update the last node's 'next' to point to the new node.



Inserting at the End: Python Code

```
def insert_at_end(self, new_data):
    new_node = Node(new_data)
    if self.head is None: # Empty list
        self.head = new_node
        return

    last = self.head
    while last.next:        # Traverse to the end
        last = last.next
    last.next = new_node   # Link it
```

Time Complexity Analysis

- Insertion at Beginning:
  - Time Complexity: O(1) (Constant Time)
  - Only pointer assignments are needed.
- Insertion at End:
  - Time Complexity: O(N) (Linear Time)
  - We must traverse N nodes to find the last one.
  - (Note: If we maintain a 'tail' pointer, this can be optimized to O(1))

Operation	**Time Complexity**	**Requirement**
Insert Beginning	O(1)	Update Head
Insert End	O(N)	Traverse to End

Quick Check: Test Your Knowledge

- Q1: When inserting at the beginning, why must we link the new node to 'head' BEFORE updating 'head'?
- Q2: What is the time complexity of inserting at the beginning of a singly linked list?
- Q3: In the 'insert at end' function, what is the condition in our while loop to find the last node?

Key Takeaways

- ✓ A Node in Python is typically a class with 'data' and 'next' attributes.
- ✓ The LinkedList class manages the nodes by maintaining a 'head' reference.
- ✓ Inserting at the beginning takes O(1) time and requires careful pointer re-assignment.
- ✓ Inserting at the end takes O(N) time because it requires traversing to the last node.

Next Lecture Preview

- Next time, we will continue our journey with Singly Linked Lists.
- Topic: Inserting a node at a specific position (middle) and Deletion operations.
- Prepare by reviewing today's code and practicing the traversals!

Coming Up Next

Insertion at a specific position & Deletion operations!

Questions?

Any Questions?

Lecture 3.3: 3.3 Insertion in Singly Linked List

Complete Lecture Deck – All 30 Lectures

Milav Dabgar

Lecturer, GP Palanpur

July 22, 2026

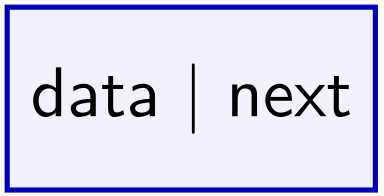
Today's Agenda

- Recap: Basic Node structure in Python
- Concept: Inserting after a given node
- Algorithm & Python Implementation: Insert After
- Concept: Inserting before a given node
- Algorithm & Python Implementation: Insert Before
- Complexity Analysis
- Practice & Quick Check

Recap: Node and Linked List Basics

- A linked list is a sequence of nodes where each node contains data and a reference to the next node.

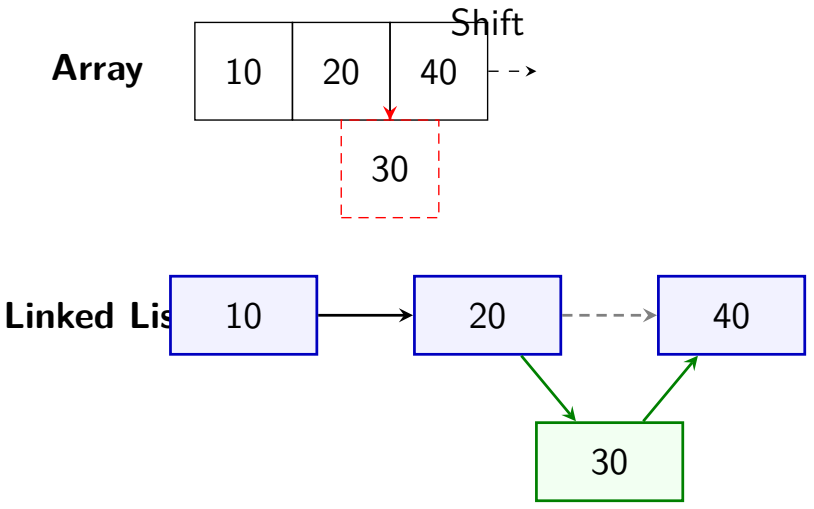
```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```



- The first node is the `head`.
- We previously learned to insert at the very beginning (updating head) and the very end.

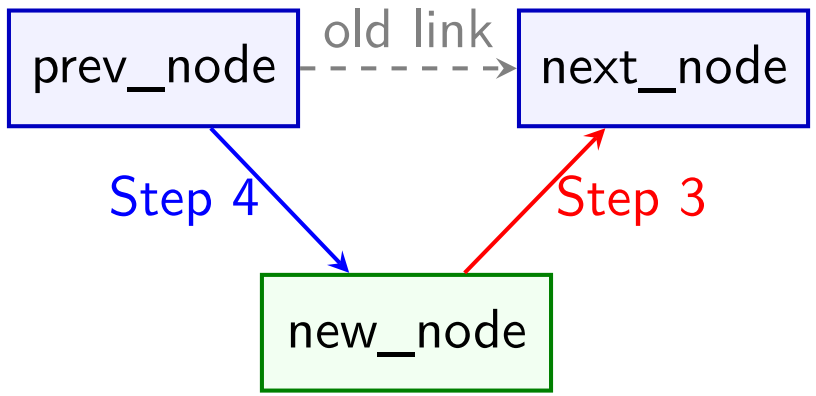
Why Insert in the Middle?

- Often, data needs to be sorted or ordered.
- We might need to insert a student record right after their peer, or right before a specific ID.
- Unlike arrays, inserting in the middle of a linked list doesn't require shifting elements.
- We only need to update a few 'next' pointers!



Concept: Inserting AFTER a Given Node

- Scenario: We are given a reference to a node (let's call it `prev_node`) and new data to insert.
- Goal: Create a new node and link it immediately following `prev_node`.
- Steps:
  1. Check if `prev_node` is None.
  2. Create the new node.
  3. Set new node's next to `prev_node`'s next.
  4. Set `prev_node`'s next to the new node.



Algorithm: Insert AFTER

- Let the node after which insertion happens be `prev_node`.
- New data is `new_data`.

Algorithm Steps

1. If `prev_node` is NULL: Error, return
2. Allocate `new_node` with `new_data`
3. `new_node.next = prev_node.next`
4. `prev_node.next = new_node`

Python Implementation: Insert AFTER

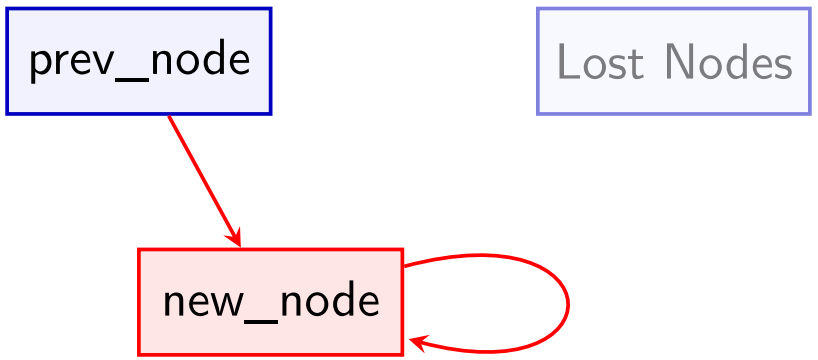
```
def insert_after(prev_node, new_data):
    # 1. Check if the given prev_node exists
    if prev_node is None:
        print("The given previous node must be in LinkedList.")
        return

    # 2. Create new node & 3. Make next of new Node as next of
        prev_node
    new_node = Node(new_data)
    new_node.next = prev_node.next

    # 4. make next of prev_node as new_node
    prev_node.next = new_node
```

Common Pitfall: Order of Assignments

- What happens if we swap the pointer assignments?

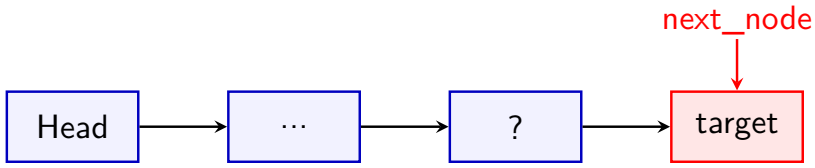
```
# WRONG ORDER
prev_node.next = new_node
new_node.next = prev_node.next
```

- Result: `new_node.next` points to itself!
- The rest of the linked list is lost (memory leak in languages like C++, garbage collected in Python).



Concept: Inserting BEFORE a Given Node

- Scenario: Insert a new node right *before* a specific node (`next_node`).
- Challenge: In a Singly Linked List, we only have forward pointers!
- We cannot traverse backward from `next_node`.
- Solution: We must traverse the list from the `head` to find the node that points to `next_node`.



Steps to Insert BEFORE

1. Check if the list is empty (head is None).
2. Special case: If the node to insert before is the head, update the head.
3. Traverse from head: `while current.next != next_node:`
4. If we reach the end and don't find it, the node isn't in the list.
5. Once found, `current` is the node *before* `next_node`.
6. Proceed as 'insert after' `current`!

Algorithm: Insert BEFORE

- Given `head`, `next_node`, `new_data`

Algorithm Steps

1. If `head` is NULL: return
2. If `head == next_node`: Insert at beginning
3. Set `curr = head`
4. Loop while `curr.next != NULL` AND `curr.next != next_node`:
   - `curr = curr.next`
5. If `curr.next == NULL`: node not found
6. Create `new_node`, `new_node.next = curr.next`, `curr.next = new_node`

Python Implementation: Insert BEFORE

```
def insert_before(head, next_node, new_data):
    if head is None or next_node is None:
        return head
    if head == next_node:
        new_node = Node(new_data)
        new_node.next = head
        return new_node
    curr = head
    while curr.next and curr.next != next_node:
        curr = curr.next
    if not curr.next:
        return head # next_node not found
    new_node = Node(new_data)
    new_node.next = curr.next
    curr.next = new_node
    return head
```

Complexity Analysis

Operation	Time Complexity	Space Complexity
Insert After	$O(1)$	$O(1)$
Insert Before	$O(n)$	$O(1)$

- **Insert After**: $O(1)$ time due to constant time pointer rewiring. $O(1)$ space for one new node.
- **Insert Before**: $O(n)$ time since in the worst case, we traverse the entire list to find the preceding node. $O(1)$ space.

Quick Check: Test Your Understanding

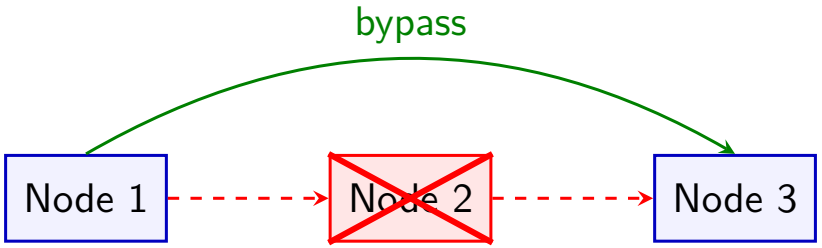
- **Q1:** Why is inserting BEFORE a node slower than inserting AFTER a node in a Singly Linked List?
- **Q2:** In `insert_after`, what is the correct order of pointer updates?
- **Q3:** If we want to easily insert before *or* after any node in $O(1)$ time, what type of linked list would we need?

Key Takeaways

- Inserting nodes requires careful pointer manipulation. Order matters!
- `new_node.next` must be assigned before `prev_node.next` is updated.
- Inserting AFTER a given reference is an $O(1)$ operation.
- Inserting BEFORE a node in a singly linked list requires a traversal from the head, making it an $O(n)$ operation.

Next Lecture Preview

- Lecture 16 will cover Deletion in a Singly Linked List, including deleting the head, the tail, and intermediate nodes.



Questions?

Lecture 3.4: Singly Linked List: Deletion & Counting

Complete Lecture Deck – All 30 Lectures

Milav Dabgar

Lecturer, GP Palanpur

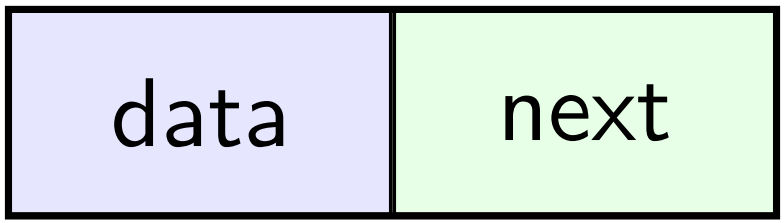
July 22, 2026

Recap: Node Structure in Python

Recall that a Linked List is made of Nodes.
Each Node contains 'data' and a 'next' pointer.

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Node Object



Deletion: Important Edge Cases

Before writing deletion logic, we must consider:

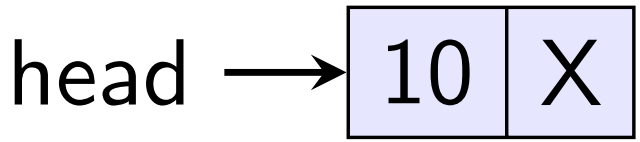
1. Is the list empty? (head is None)
2. Does the list have only ONE node?
3. Does the list have MULTIPLE nodes?

Handling these cases prevents runtime errors like `AttributeError` in Python.

1. Empty List

head ⟶ None

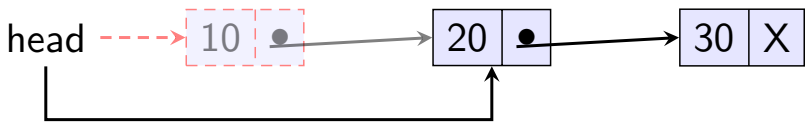
2. One Node



Deleting the First Node: Concept

- **Goal:** Remove the node currently at the 'head' of the list.
- **Step 1:** Check if the list is empty. If yes, nothing to delete.
- **Step 2:** Move the 'head' pointer to point to the second node.

By reassigning 'head', the first node becomes disconnected and is eventually garbage collected by Python.



Deleting the First Node: Python Code

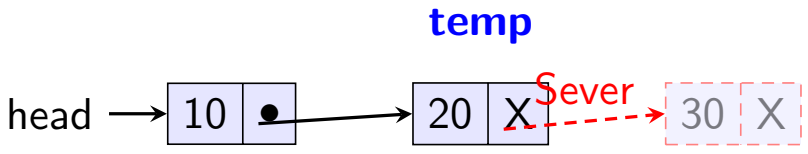
```
class LinkedList:
    def delete_first(self):
        if self.head is None:
            print('List is empty')
            return
        # Move head to the next node
        self.head = self.head.next
```

What if there is only 1 node?

If there is 1 node, `self.head.next` is `None`.
So, `self.head` becomes `None`, making the list empty!

Deleting the Last Node: Concept

- **Goal:** Remove the very last node (where next is None).
- **Challenge:** To delete the last node, we need to update the 'next' pointer of the **SECOND-TO-LAST** node to None.
- **Step 1 & 2:** Check if empty or if only 1 node.
- **Step 3:** Traverse the list to find the second-to-last node (`temp.next.next is not None`).
- **Step 4:** Set `temp.next = None`.



Deleting the Last Node: Python Code

```
def delete_last(self):
    if self.head is None: # Empty
        return
    if self.head.next is None: # 1 Node
        self.head = None
        return

    temp = self.head
    while temp.next.next is not None:
        temp = temp.next

    temp.next = None # Disconnect last
```

Time Complexity of Deletion

- **Deleting the First Node: O(1) time.**
  - We only change the head pointer. No traversal needed.
- **Deleting the Last Node: O(n) time.**
  - We must traverse the entire list to find the second-to-last node.
  - 'n' is the total number of nodes in the list.

Operation	**Time Complexity**
Delete First Node	$O(1)$
Delete Last Node	$O(n)$

Counting Nodes: Concept

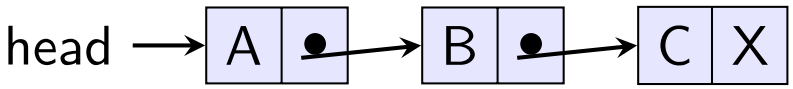
Unlike arrays, linked lists don't have a built-in 'length' property.
To find the length, we must traverse the list from 'head' to the end.

Algorithm:

1. Initialize a counter to 0.
2. Start a pointer (`temp`) at 'head'.
3. While the pointer is not None:
   - Increment counter.
   - Move pointer to 'next'.
4. Return counter.

Counting Nodes: Python Code

```
def count_nodes(self):
    count = 0
    temp = self.head
    while temp is not None:
        count += 1
        temp = temp.next
    return count
```



count: 3

Application & Best Practices

When is Counting useful?

- Validating indices before inserting or deleting at a specific position.

Best Practices in Python:

- Avoid memory leaks by ensuring references to deleted nodes are properly cleared.
- Can we make counting $O(1)$? Yes, by maintaining a `self.size` variable in the `LinkedList` class and updating it on every insert/delete.

✓ **Pro Tip:** Keep a `size` attribute for $O(1)$ counting!

Quick Check: Test Your Understanding

Questions

- **Q1:** Why do we need to check if `self.head.next is None` when deleting the last node?
- **Q2:** What happens to a node in Python after we remove all pointers to it?
- **Q3:** What is the time complexity of the `count_nodes()` function?

Key Takeaways

- Deleting the first node is fast ($O(1)$) and only requires moving the head pointer.
- Deleting the last node requires $O(n)$ traversal to find the second-to-last node.
- Always account for edge cases: empty lists and single-node lists.
- Counting nodes requires traversing the entire list, resulting in $O(n)$ time complexity.

Next Lecture Preview

Coming Up Next...

In the next lecture, we will complete Singly Linked Lists by looking at **Deletion at a Specific Position** and **Searching** for an element.

Questions?

Any Questions?

Lecture 3.5: Circular Linked List

Complete Lecture Deck – All 30 Lectures

Milav Dabgar

Lecturer, GP Palanpur

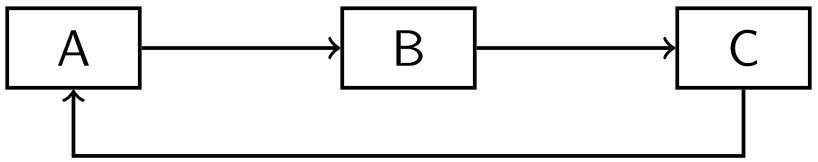
July 22, 2026

Agenda

- What is a Circular Linked List?
- Structure of a Circular Linked List
- Real-world Examples
- Singly vs. Circular Linked Lists
- Python Implementation: Creating a Node

What is a Circular Linked List?

- A Circular Linked List is a variation of a linked list.
- In this structure, the last node points back to the first node instead of pointing to None (or Null).
- This forms a continuous loop or circle.
- Any node can be chosen as the starting point to traverse the entire list.



Types of Circular Linked Lists

- **Singly Circular Linked List:** Each node has data and one pointer to the next node. The last node points to the first.
- **Doubly Circular Linked List:** Each node has data, a next pointer, and a previous pointer. The last node's next points to the first, and the first node's previous points to the last.

Real-world Analogies

- **Multiplayer Board Games:** Turns pass from player 1 to 2 to 3, and back to 1.
- **Operating System Task Scheduling (Round Robin):** The CPU allocates a time slice to each process in a circular order.
- **Music Player Playlists:** When 'Repeat All' is selected, the playlist goes back to the first song after finishing the last.

Advantages of Circular Linked Lists

- ✓ Any node can be a starting point for traversal.
- ✓ Useful for implementing queue data structures (especially circular queues).
- ✓ Ideal for applications that require repetitive looping through elements.

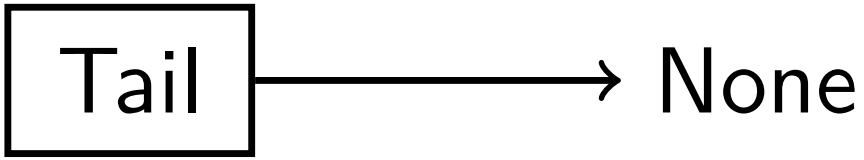
Disadvantages of Circular Linked Lists

- ✗ More complex to implement compared to simple singly linked lists.
- ✗ Potential for infinite loops if traversal conditions are not carefully managed.
- ✗ No natural 'end' to the list, so we must track the head or a specific node to know when to stop.

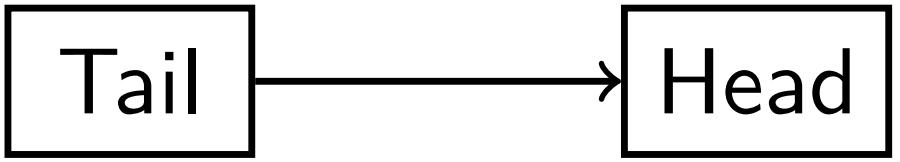
Singly Linked List vs. Circular Linked List

- **Singly Linked List:** Last node's next pointer is None.
- **Circular Linked List:** Last node's next pointer points to the head node.

Singly:



Circular:



Comparison: Traversal

- **Singly Linked List:** Traversal stops when a node's next pointer is None.
- **Circular Linked List:** Traversal stops when we reach the node we started from.

Comparison: Starting Point

- **Singly Linked List:** Must always start from the head node to traverse the entire list.
- **Circular Linked List:** Can start from any node and traverse the entire list.

Python Code: Node Class

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Python Code: Creating a Circular Link

```
# Create nodes
node1 = Node(10)
node2 = Node(20)
node3 = Node(30)

# Link them
node1.next = node2
node2.next = node3
node3.next = node1 # This creates the circle!
```

Python Code: Traversal (Snippet)

```
current = head
if current is not None:
    while True:
        print(current.data)
        current = current.next
        if current == head:
            break
```

Quick Check: Test Your Understanding

- Q: What does the last node point to in a Circular Linked List?
- Q: How do we prevent an infinite loop when traversing?
- Q: Name one real-world application of a Circular Linked List.

Key Takeaways

- A Circular Linked List has no 'None' at the end; the last node points back to the first.
- It's useful for cyclical data like round-robin scheduling.
- Traversal requires checking if we've returned to the starting node, not checking for None.

Next Lecture Preview

In the next lecture, we will dive deeper into Operations on Circular Linked Lists: Insertion, Deletion, and Searching.

Questions?

Lecture 18: Doubly Linked List Overview

Complete Lecture Deck – All 30 Lectures

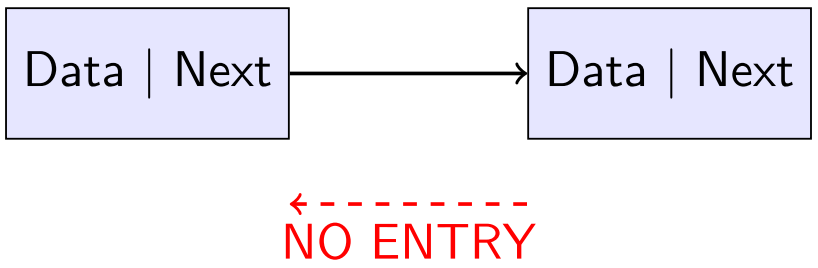
Milav Dabgar

Lecturer, GP Palanpur

July 22, 2026

Recap: Limitations of Singly Linked List

- In a singly linked list, each node has data and a single pointer ('next') to the next node
- Traversal is strictly unidirectional: from head to tail
- Cannot easily access the previous node without traversing again from the head
- Operations like deleting the last node are O(N) because we need to find the second-to-last node



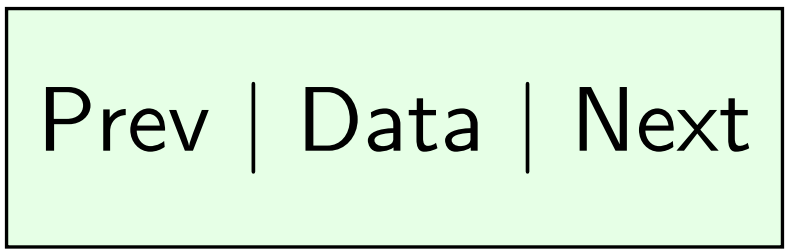
What is a Doubly Linked List (DLL)?

A Doubly Linked List is a linear data structure where each node contains three parts:

1. Data: The actual value stored
2. Next Pointer: Reference to the next node
3. Previous Pointer: Reference to the previous node

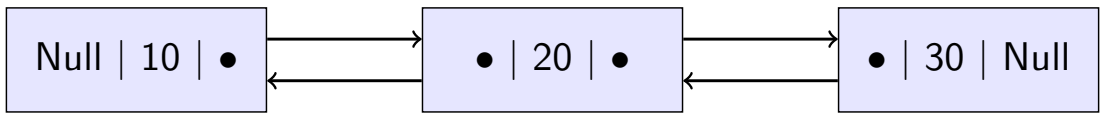
Key Feature

Allows bidirectional traversal: we can move both forward and backward



DLL Node Structure Visually

- `[ Prev | Data | Next ]`
- The 'prev' pointer of the first node (Head) is always Null / None
- The 'next' pointer of the last node (Tail) is always Null / None
- Intermediate nodes link to their neighbors on both sides



Implementing a DLL Node in Python

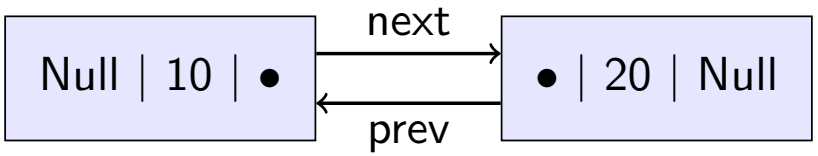
```
class Node:
    def __init__(self, data):
        self.data = data
        self.prev = None   # Pointer to previous node
        self.next = None   # Pointer to next node
```

Comparison with SLL

The only difference is the addition of the `self.prev` pointer.

Connecting DLL Nodes in Python

```
head = Node(10)
node2 = Node(20)

head.next = node2       # Forward link
node2.prev = head       # Backward link
```

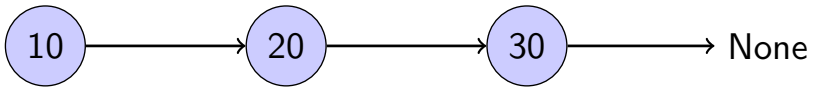


Singly vs Doubly Linked List

Singly Linked List	**Doubly Linked List**
1 pointer per node (next)	2 pointers per node (prev, next)
Less memory	More memory
One-way traversal	Two-way traversal
Deletion at tail: O(N) time complexity	Deletion at tail: O(1) time complexity (if tail pointer maintained)

Forward Traversal Concept

- Start at the Head node
- Process the current node's data
- Move to the next node using the 'next' pointer (`current = current.next`)
- Repeat until the current node becomes None
- Exactly the same logic as traversing a Singly Linked List!

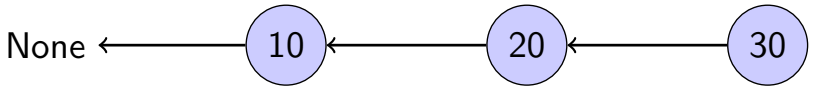


Forward Traversal in Python

```
def traverse_forward(head):
    current = head
    while current is not None:
        print(current.data, end=' <-> ')
        current = current.next
    print('None')
```

Backward Traversal Concept

- Start at the Tail node (or traverse forward to find the last node first)
- Process the current node's data
- Move to the previous node using the 'prev' pointer (`current = current.prev`)
- Repeat until the current node becomes None (which happens after the head node)
- This is the unique power of the Doubly Linked List

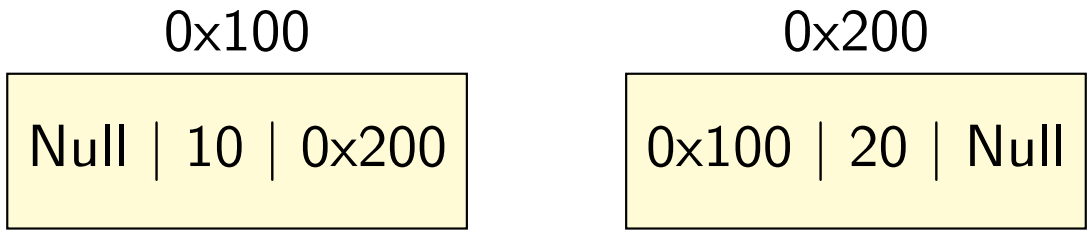


Backward Traversal in Python

```
def traverse_backward(tail):
    current = tail
    while current is not None:
        print(current.data, end=' <-> ')
        current = current.prev
    print('None')
```

Memory Representation of DLL

- Each node exists in dynamically allocated memory (Heap)
- Nodes are NOT stored in contiguous memory locations (unlike arrays)
- A Node takes up: Size(Data) + Size(Pointer) $\times$ 2
- Pointers hold the actual memory addresses of the adjacent nodes



Advantages of Doubly Linked List

- ✓ Can be traversed in both forward and backward directions easily
- ✓ Deleting a specific given node is easier because we have direct access to its predecessor
- ✓ Inserting a new node before a given node is simpler than in an SLL
- ✓ Excellent for problems requiring frequent bidirectional movement

Disadvantages of Doubly Linked List

- ✗ Extra memory required for the 'prev' pointer in every node
- ✗ More complex implementation: every insertion/deletion requires modifying 4 pointers instead of 2
- ✗ Increased risk of bugs (e.g., accidentally breaking the backwards chain while fixing the forward chain)
- ✗ Slightly slower operations due to the overhead of updating extra pointers

Real-World Applications of DLL

- Web Browser History (Forward and Back buttons)
- Music/Video Player Playlists (Next track, Previous track)
- Undo/Redo functionality in text editors (Word, VS Code)
- Recent Applications list in mobile operating systems
- Complex data structures like Fibonacci Heaps or LRU Caches

Quick Check: Test Your Understanding

1. What is the value of 'prev' in the Head node of a DLL?
2. Why does a DLL node consume more memory than an SLL node?
3. Can a DLL be traversed backwards if you only have a pointer to the Head? How?
4. In Python, how do you move to the previous node?

Key Takeaways

- A Doubly Linked List node contains data, a 'next' pointer, and a 'prev' pointer
- Bidirectional traversal is the primary advantage over Singly Linked Lists
- Implementation in Python requires careful management of both forward and backward references
- Trade-off: We gain flexibility and speed for some operations at the cost of extra memory and code complexity

Next Lecture Preview

Lecture 19: Insertion Operations in a Doubly Linked List

- We will write the Python code to insert nodes at the beginning, end, and middle of a DLL.
- Be ready to track and update 4 pointers simultaneously!

Questions?

Any Questions?

Today's Agenda

- What is a Doubly Linked List?
- Structure of a DLL Node in Python
- Singly vs. Doubly Linked List
- Traversing Forward and Backward
- Insertion Operations (Beginning & End)
- Pros, Cons, and Real-world Applications

What is a Doubly Linked List?

- A Doubly Linked List (DLL) is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence.
- Each node consists of three parts: Data, Next Pointer, and Previous Pointer.
- The 'Previous Pointer' of the first node points to None.
- The 'Next Pointer' of the last node points to None.

Structure of a DLL Node in Python

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None
```

Singly vs. Doubly Linked List

Singly Linked List	**Doubly Linked List**
- Nodes have one pointer (next).	- Nodes have two pointers (prev, next).
- Forward traversal only.	- Forward and backward traversal.
- Uses less memory per node.	- Uses more memory (extra pointer per node).

Basic Operations in DLL

Just like Singly Linked Lists, we can perform standard operations:

1. **Traversal (Forward & Backward)**
2. **Insertion (At front, end, or middle)**
3. Deletion (From front, end, or middle)
4. Searching for an element

Today, we focus on **Traversal and Basic Insertions**.

Traversing Forward

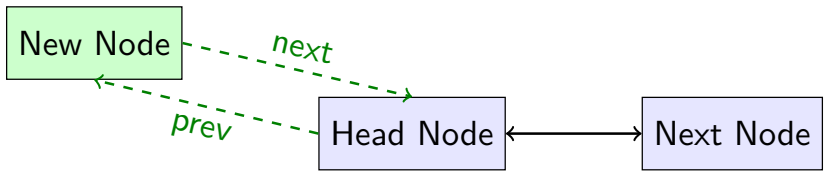
```
def display_forward(self):
    if self.head is None:
        print('List is empty')
        return
    current = self.head
    while current:
        print(current.data, end=' <-> ')
        current = current.next
    print('None')
```

Traversing Backward

```
def display_backward(self, tail):
    if tail is None:
        print('List is empty')
        return
    current = tail
    while current:
        print(current.data, end=' <-> ')
        current = current.prev
    print('None')
```

Insertion at the Beginning (Concept)

- Step 1: Create the new node.
- Step 2: Set new node's next to current head.
- Step 3: If list is not empty, set current head's prev to new node.
- Step 4: Update head to point to the new node.



Insertion at the Beginning (Code)

```
def insert_at_beginning(self, data):
    new_node = Node(data)
    if self.head is None:
        self.head = new_node
        return
    new_node.next = self.head
    self.head.prev = new_node
    self.head = new_node
```

Insertion at the End (Concept)

- Step 1: Create the new node.
- Step 2: If list is empty, make it the head.
- Step 3: Otherwise, traverse to the last node.
- Step 4: Set last node's next to the new node.
- Step 5: Set new node's prev to the last node.

Insertion at the End (Code)

```
def insert_at_end(self, data):
    new_node = Node(data)
    if self.head is None:
        self.head = new_node
        return
    current = self.head
    while current.next:
        current = current.next
    current.next = new_node
    new_node.prev = current
```

Time Complexity of Operations

- Insertion at Beginning: O(1)
  - Only updating a few pointers at the head.
- Insertion at End: O(n) without tail pointer, O(1) with tail pointer.
  - Need to traverse the entire list if we only know the head.
- Traversal: O(n)
  - Must visit every node.

Advantages of DLL

- Can be traversed in both forward and backward directions.
- Deletion of a given node is easier if a pointer to the node is given (don't need to traverse to find previous node).
- Can quickly insert a new node before a given node.

Disadvantages of DLL

- Extra memory space for the 'prev' pointer in each node.
- All operations require an extra pointer to be maintained (more complex code).
- Higher risk of pointer mismanagement bugs.

Real World Applications

- **Browser History:** Forward and Back buttons navigate a DLL of visited pages.
- **Music Playlists:** Next Track and Previous Track functionalities.
- **Undo/Redo Operations:** Text editors use DLL to step forward and backward through states.
- **LRU Cache:** Least Recently Used caching algorithms use DLLs for fast O(1) updates.

Quick Check

- Q1: What are the three fields inside a Doubly Linked List Node?
- Q2: How many pointers need to be updated when inserting a node at the beginning of a non-empty DLL?
- Q3: Give one real-world application of a Doubly Linked List.

Key Takeaways

- A DLL node contains data, a next pointer, and a prev pointer.
- The prev pointer allows bidirectional traversal, enabling more efficient backward operations.
- Insertion requires careful updating of both next and prev pointers to maintain chain integrity.
- DLLs trade memory efficiency (extra pointer) for functional flexibility (two-way movement).

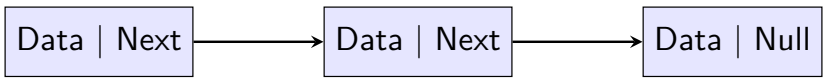
Next Lecture Preview

In Lecture 20, we will continue with Doubly Linked Lists, covering Deletion operations (from beginning, end, and middle) and Circular Linked Lists.

Questions?

Data Structure and Applications (Python)

- Course: Data Structure and Applications (Python)
- Course Code: DI03032031
- GTU Diploma Engineering - Semester 3
- Unit 3: Linked List
- Lecture 20: Applications of Linked List



Today's Agenda

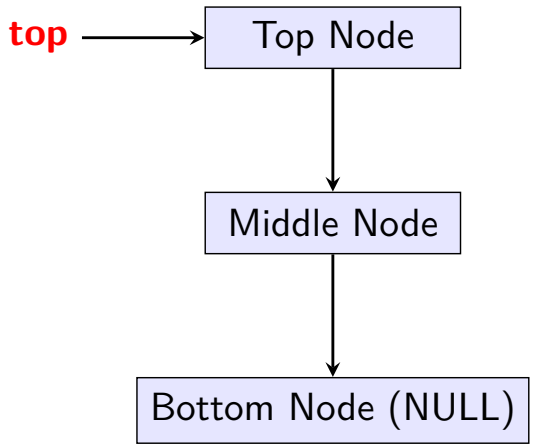
- ➢ Why use Linked Lists for Applications?
- ➢ Implementing Stacks using Linked Lists
- ➢ Implementing Queues using Linked Lists
- ➢ Polynomial Representation & Addition
- ➢ Sparse Matrix Representation

Why Use Linked Lists for Stacks & Queues?

- **Dynamic Sizing:** Arrays have a fixed size or require expensive resizing (O(n)). Linked lists grow and shrink dynamically at O(1) cost.
- **No Memory Wastage:** Memory is allocated only when a new element is pushed or enqueued.
- **Constant Time Operations:** If we keep track of the head (and tail for queues), insertions and deletions are O(1).
- **Trade-off:** Extra memory overhead for storing the 'next' pointer in each node.

Stack using Linked List: Concept

- A Stack follows LIFO (Last In, First Out).
- Using a Linked List, the 'top' of the stack is simply the 'head' of the linked list.
- **Push Operation:** Insert a new node at the head (beginning) of the list.
- **Pop Operation:** Remove the node from the head (beginning) of the list.



Stack using LL: Push Operation (Python)

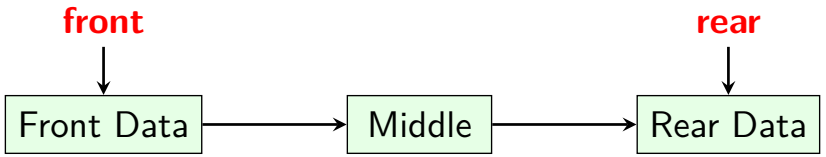
```python
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class Stack:
    def __init__(self):
        self.top = None

    def push(self, data):
        new_node = Node(data)
        new_node.next = self.top
        self.top = new_node
```

Stack using LL: Pop Operation (Python)

```
def pop(self):
    if self.is_empty():
        return 'Stack Underflow'
    popped_node = self.top
    self.top = self.top.next
    popped_node.next = None  # Optional cleanup
    return popped_node.data

def is_empty(self):
    return self.top is None
```

Queue using Linked List: Concept

- A Queue follows FIFO (First In, First Out).
- We need two pointers: 'front' (for dequeuing) and 'rear' (for enqueuing).
- **Enqueue:** Insert at the 'rear' (end) of the linked list.
- **Dequeue:** Remove from the 'front' (beginning) of the linked list.
- Maintaining both pointers allows O(1) operations for both enqueue and dequeue.



Queue using LL: Enqueue (Python)

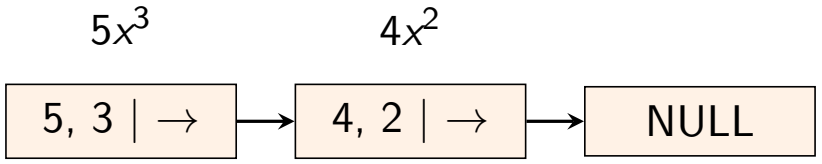
```
class Queue:
    def __init__(self):
        self.front = self.rear = None

    def enqueue(self, data):
        new_node = Node(data)
        if self.rear is None:  # Queue is empty
            self.front = self.rear = new_node
            return
        self.rear.next = new_node
        self.rear = new_node
```

Queue using LL: Dequeue (Python)

```
def dequeue(self):
    if self.front is None:
        return 'Queue is Empty'
    temp = self.front
    self.front = temp.next

    # If front becomes None, rear must also become None
    if self.front is None:
        self.rear = None

    return temp.data
```

Polynomial Representation using Linked List

- Mathematical polynomials like $5x^3 + 4x^2 - 2x + 1$ can be represented as a Linked List.
- Each term in the polynomial becomes a Node in the list.
- A Node stores three things:
  1. Coefficient (e.g., 5)
  2. Exponent (e.g., 3)
  3. Pointer to the next term
- List is typically maintained in descending order of exponents.

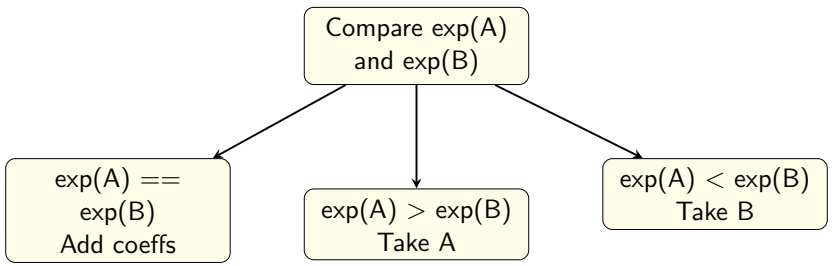


Node Structure for Polynomials (Python)

```
class PolyNode:
    def __init__(self, coeff, exp):
        self.coeff = coeff
        self.exp = exp
        self.next = None

# Example: 5x^3
term1 = PolyNode(5, 3)
```

Polynomial Addition: Logic

- Traverse both linked lists (Poly A and Poly B) simultaneously.
- Compare the exponents of the current nodes:
  - If $\exp(A) == \exp(B)$: Add coefficients, create new node, advance both A and B.
  - If $\exp(A) > \exp(B)$: Create new node from A, advance A.
  - If $\exp(A) < \exp(B)$: Create new node from B, advance B.
- Append any remaining terms from A or B to the result.



Polynomial Addition Example

Algebraic Addition:

$$\begin{array}{rr} \text{Poly 1:} & 5x^2 + 4x^1 + 2x^0 \\ +\text{Poly 2:} & 5x^1 + 5x^0 \\ \hline \text{Result:} & 5x^2 + 9x^1 + 7x^0 \end{array}$$

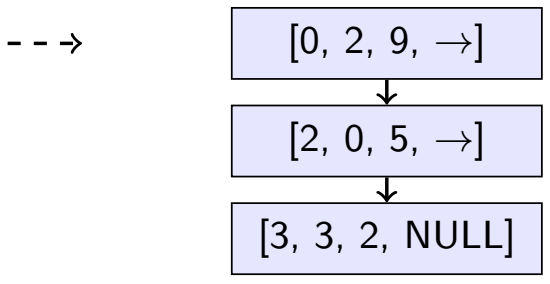
Linked List Traversal:

```
LL1: [5,2] -> [4,1] -> [2,0]
LL2:          [5,1] -> [5,0]
Res: [5,2] -> [9,1] -> [7,0]
```

Sparse Matrix Representation

- A **Sparse Matrix** is a matrix where the vast majority of elements are zero.
- Storing it as a 2D array wastes huge amounts of memory.
- **Linked List approach:** Only store the non-zero elements.
- Each Node stores:
  1. Row Index
  2. Column Index
  3. Value
  4. Pointer to next non-zero element

0	0	9	0
0	0	0	0
5	0	0	0
0	0	0	2



Array vs. Linked List Applications

Use Arrays When:	**Use Linked Lists When:**
Size is known in advance and fixed.	Size fluctuates dynamically (Stacks, Queues).
Frequent random access is needed (e.g., arr[50]).	Data is sparse (Sparse Matrices, Polynomials).
Memory space is extremely tight.	Insertions/deletions at the ends are frequent.

Quick Check: Test Your Understanding

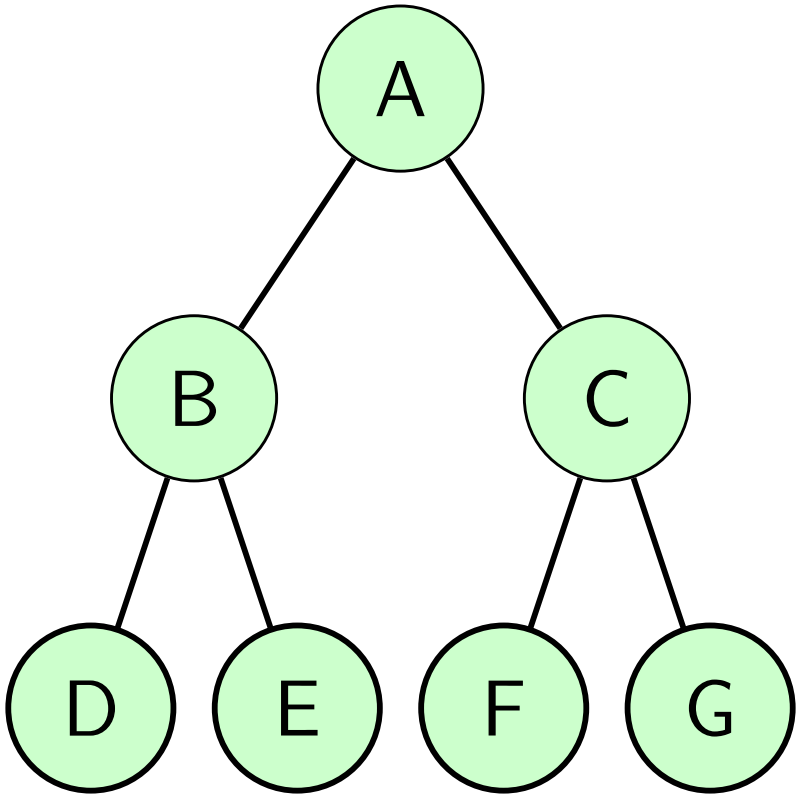
- **Q1:** For a Linked List based Queue, why do we maintain a 'rear' pointer?
- **Q2:** In polynomial addition, what do we do if Poly A's exponent is greater than Poly B's exponent?
- **Q3:** How does a linked list save memory in a Sparse Matrix representation?

Key Takeaways

- ✓ Linked Lists provide O(1) dynamic insertion/deletion, making them ideal for Stacks and Queues.
- ✓ A LL Stack uses the 'head' as the 'top' for O(1) push and pop.
- ✓ A LL Queue uses 'front' and 'rear' pointers for O(1) dequeue and enqueue respectively.
- ✓ Polynomials and Sparse Matrices use modified LL nodes to store multi-dimensional data, drastically saving memory when zero-values are common.

Next Lecture Preview

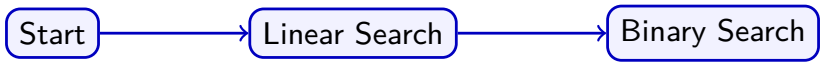
- In Lecture 21, we will begin Unit 4: Non-Linear Data Structures, starting with the fundamentals of Trees and Tree Terminology.



Any Questions?

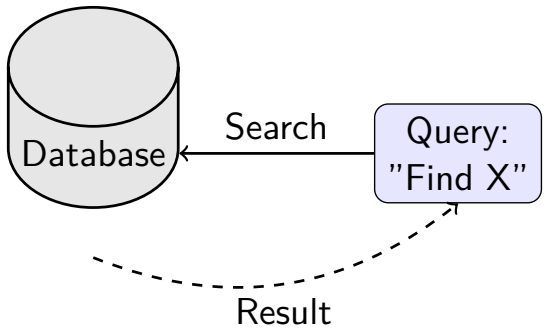
Today's Agenda

- Introduction to Searching Algorithms
- Linear Search: Concept & Python Implementation
- Linear Search: Complexity Analysis
- Binary Search: The Divide & Conquer Approach
- Binary Search: Iterative & Recursive Implementation
- Binary Search: Complexity Analysis
- Comparing Linear vs. Binary Search



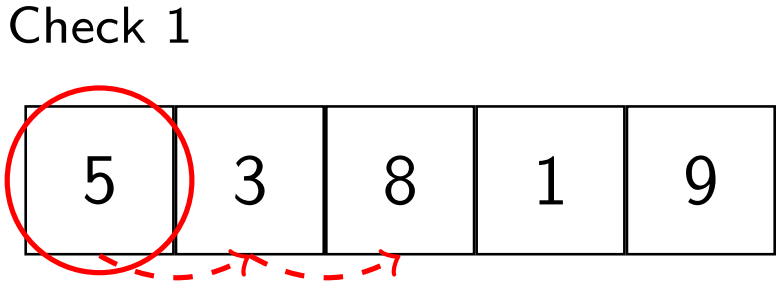
What is Searching?

- Searching is the process of finding the location of a specific element (target or key) within a collection of data
- It is one of the most common operations in computer science (e.g., finding a contact, searching a database)
- The outcome of a search is typically the index/position of the item if found, or a failure indicator (like -1 or None) if not found
- The efficiency of a search algorithm depends heavily on the underlying data structure and whether the data is sorted



Linear Search: The Concept

- Linear Search (or Sequential Search) is the simplest searching algorithm
- It checks every element in a list one by one from the beginning until the target element is found or the end of the list is reached
- Does not require the list to be sorted - works on any arbitrary list
- Intuitive but can be very slow for large datasets

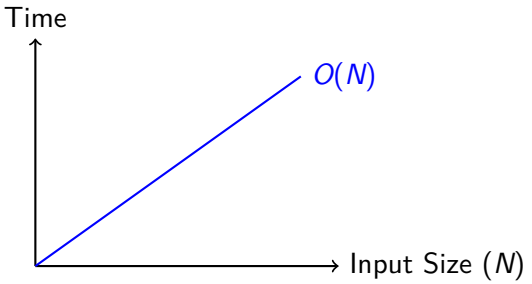


Linear Search: Python Implementation

```python
def linear_search(arr, target):
    # Iterate through the list with index and value
    for i in range(len(arr)):
        if arr[i] == target:
            return i  # Return index if found
    return -1  # Return -1 if target is not in the list

# Example usage
my_list = [64, 34, 25, 12, 22, 11, 90]
result = linear_search(my_list, 25) # Returns 2
```

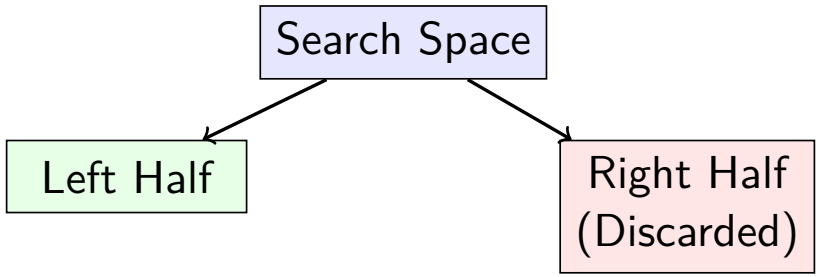
Linear Search: Complexity Analysis

- Best Case Time Complexity: $O(1)$ - The target is the very first element
- Worst Case Time Complexity: $O(N)$ - The target is the last element or not present at all
- Average Case Time Complexity: $O(N)$ - On average, we might check half the elements ($N/2$), which simplifies to $O(N)$
- Space Complexity: $O(1)$ - Requires no extra space, just a few variables for iteration



Binary Search: The Concept

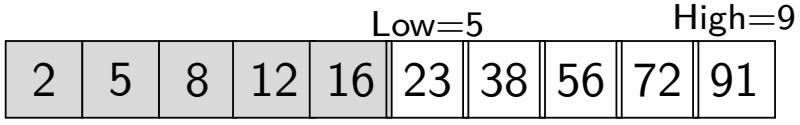
- Binary Search is a highly efficient searching algorithm that follows the Divide and Conquer strategy
- **Crucial Requirement**: The input list MUST be sorted beforehand
- Instead of checking every element, it repeatedly divides the search interval in half
- If the target value is less than the middle element, narrow the interval to the lower half. Otherwise, narrow it to the upper half



Binary Search: Step-by-Step

Target: 23. Array: [2, 5, 8, 12, 16, 23, 38, 56, 72, 91]

- **Step 1**: Low=0, High=9. Mid=4 (Val 16). $23 > 16$. Discard left half.
- **Step 2**: Low=5, High=9. Mid=7 (Val 56). $23 < 56$. Discard right half.
- **Step 3**: Low=5, High=6. Mid=5 (Val 23). Found target at index 5!



Binary Search: Iterative Implementation

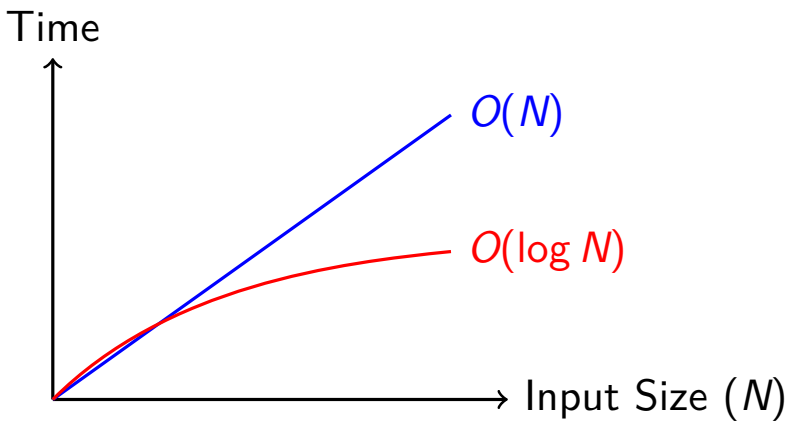
```
def binary_search(arr, target):
    low = 0
    high = len(arr) - 1

    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
            return mid       # Found it!
        elif arr[mid] < target:
            low = mid + 1    # Discard left half
        else:
            high = mid - 1   # Discard right half

    return -1                # Not found
```

Binary Search: Recursive Implementation

```
def binary_search_recursive(arr, low, high, target):
    if low > high:  # Base case: not found
        return -1

    mid = (low + high) // 2

    if arr[mid] == target:  # Base case: found
        return mid
    elif arr[mid] < target:
        return binary_search_recursive(arr, mid + 1, high,
            target)
    else:
        return binary_search_recursive(arr, low, mid - 1, target
            )
```

Binary Search: Complexity Analysis

- Best Case Time Complexity: $O(1)$ - Target is exactly at the middle on the first try
- Worst & Average Case Time Complexity: $O(\log N)$ - The search space is halved at every step
- Space Complexity (Iterative): $O(1)$ - Only a few variables for pointers
- Space Complexity (Recursive): $O(\log N)$ - Call stack memory used for recursive function calls



Comparison: Linear vs Binary Search

Feature	**Linear Search**	**Binary Search**
Pre-requisite	None	Requires sorted list
Approach	Sequential	Divide and Conquer
Time Complexity	$O(N)$	$O(\log N)$
Best Use Case	Small/unsorted lists	Large, sorted datasets

Built-in Search in Python

```python
# Python uses Linear Search under the hood for the 'in' operator
    on lists
my_list = [10, 20, 30, 40]
print(30 in my_list)  # True, O(N)

# For Binary Search, Python provides the 'bisect' module
import bisect
sorted_list = [10, 20, 30, 40]
# bisect.bisect_left finds the insertion point to maintain order
index = bisect.bisect_left(sorted_list, 30)
if index != len(sorted_list) and sorted_list[index] == 30:
    print('Found at', index)  # O(log N)
```

Quick Check: Test Your Understanding

Questions

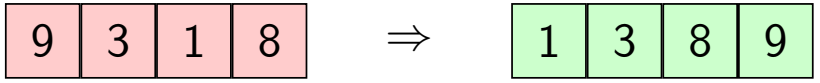
1. What happens if you run Binary Search on an unsorted array?
2. If an array has 1024 elements, what is the maximum number of comparisons Binary Search will make?
3. In what scenario would Linear Search perform better than Binary Search?

Key Takeaways

- ✓ Linear Search checks elements sequentially, is $O(N)$, and works on unsorted lists
- ✓ Binary Search halves the search space, is highly efficient at $O(\log N)$, but requires a sorted list
- ✓ Binary Search utilizes the Divide and Conquer approach
- ✓ Binary Search can be implemented both iteratively ($O(1)$ space) and recursively ($O(\log N)$ space)

Next Lecture Preview

In Lecture 22, we will introduce Sorting Algorithms, starting with Bubble Sort and Selection Sort to understand how to prepare data for efficient searching.

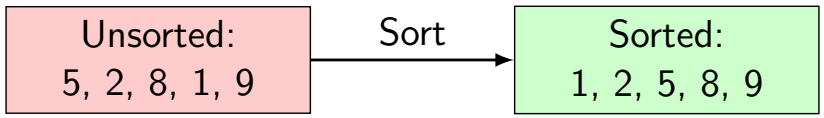


Questions?

Any questions?

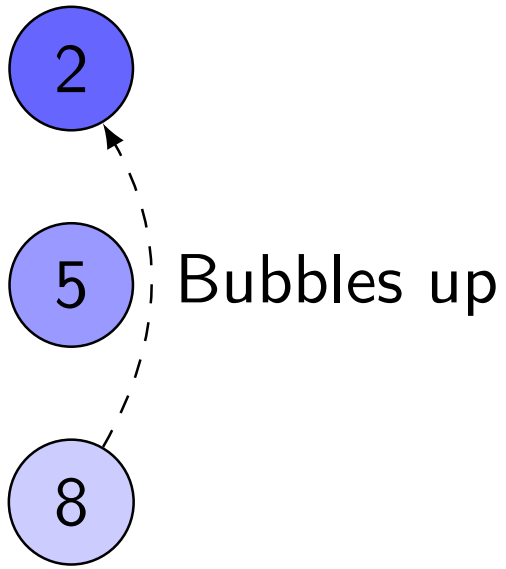
Why Sorting?

- Sorting is the process of arranging data in a particular format (ascending or descending).
- It significantly optimizes data searching (e.g., Binary Search requires sorted data).
- Makes data more readable and easier to analyze (e.g., top 10 scores, highest salaries).
- Foundation for many complex algorithms like merge, greedy algorithms, etc.



Bubble Sort: The Concept

- Simplest sorting algorithm.
- Repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.
- The pass through the list is repeated until the list is sorted.
- Largest elements 'bubble' to the top (end of the array) with each pass.



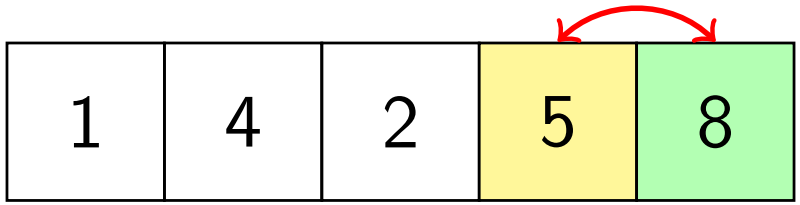
Bubble Sort: Step-by-Step Walkthrough

Initial Array: `[5, 1, 4, 2, 8]`

Pass 1:

- Compare 5 & 1 → Swap → `[1, 5, 4, 2, 8]`
- Compare 5 & 4 → Swap → `[1, 4, 5, 2, 8]`
- Compare 5 & 2 → Swap → `[1, 4, 2, 5, 8]`
- Compare 5 & 8 → No Swap → `[1, 4, 2, 5, 8]`

(8 is in its final place)



Bubble Sort: Python Implementation

```python
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        # Last i elements are already in place
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                # Swap elements
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr
```

Optimized Bubble Sort

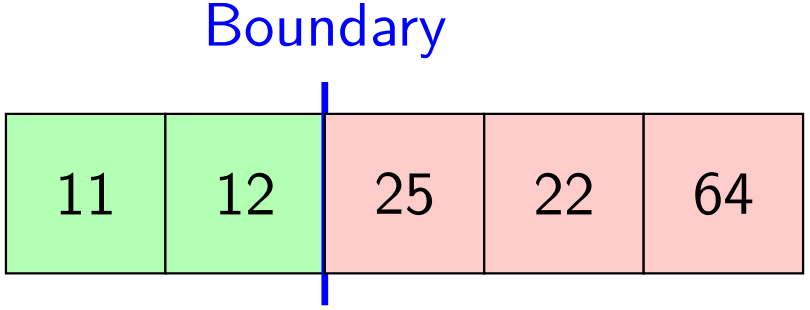
```
def optimized_bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        swapped = False
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True
        if not swapped:
            break # Array is sorted
    return arr
```

Bubble Sort: Complexity Analysis

- **Best Case Time Complexity:** $O(n)$ (Using optimized version on an already sorted array)
- **Worst & Average Case Time Complexity:** $O(n^2)$ (Reverse sorted or random array)
- **Space Complexity:** $O(1)$ (In-place sorting)
- **Stability:** Stable (Equal elements retain their relative order)

Selection Sort: The Concept

- Divides the array into two parts: sorted and unsorted.
- Repeatedly finds the minimum element from the unsorted part.
- Swaps that minimum element with the first element of the unsorted part.
- The sorted boundary grows by one element after each pass.



Selection Sort: Step-by-Step Walkthrough

Initial Array: `[64, 25, 12, 22, 11]`

- **Pass 1:** Min is 11. Swap with 64. $\rightarrow$ `[11, 25, 12, 22, 64]`
- **Pass 2:** Min in [25..64] is 12. Swap with 25. $\rightarrow$ `[11, 12, 25, 22, 64]`
- **Pass 3:** Min in [25..64] is 22. Swap with 25. $\rightarrow$ `[11, 12, 22, 25, 64]`
- **Pass 4:** Min in [25..64] is 25. Swap with 25 (itself). $\rightarrow$ `[11, 12, 22, 25, 64]`

Selection Sort: Python Implementation

```
def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_idx = i
        for j in range(i + 1, n):
            if arr[j] < arr[min_idx]:
                min_idx = j
        # Swap the found minimum with the first element
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
    return arr
```

Selection Sort: Complexity Analysis

- **Best Case Time Complexity:** $O(n^2)$ (Must always scan unsorted part)
- **Worst & Average Case Time Complexity:** $O(n^2)$
- **Space Complexity:** $O(1)$ (In-place sorting)
- **Stability:** Generally Unstable (Swaps can jump over identical elements)

Comparison: Bubble vs Selection Sort

Feature	Bubble Sort	Selection Sort
Swaps (Worst-case)	$O(n^2)$	$O(n)$
Best Case Time	$O(n)$ (optimized)	$O(n^2)$
Stability	Stable	Unstable
Ideal Use Case	Almost sorted array	When memory writes are costly

Quick Check: Test Your Understanding

- **Q:** Which sorting algorithm performs fewer swaps overall?
- **Q:** Can Bubble Sort detect if an array is already sorted?
- **Q:** In Python, how do you easily swap two elements without a temporary variable?
- **Q:** Why is Selection Sort considered unstable?

Key Takeaways

- **Bubble Sort** repeatedly compares adjacent elements and swaps them, bubbling the largest to the end.
- **Selection Sort** finds the minimum element in the unsorted portion and places it at the correct position.
- Both have an average and worst-case time complexity of $O(n^2)$.
- Bubble Sort is stable and can be optimized to $O(n)$ best-case. Selection Sort makes fewer swaps ($O(n)$).
- Both algorithms are in-place, meaning they have a space complexity of $O(1)$.

Next Lecture Preview

- In Lecture 23, we will explore more advanced and efficient sorting techniques: Insertion Sort, Merge Sort, and Quick Sort.
- We'll see how Divide and Conquer strategies can drastically reduce sorting time from $O(n^2)$ to $O(n \log n)$.

Merge Sort
$O(n \log n)$

Quick Sort
$O(n \log n)$

Questions?

Lecture 4.3: Sorting Methods: Insertion Sort and Merge Sort

Complete Lecture Deck – All 30 Lectures

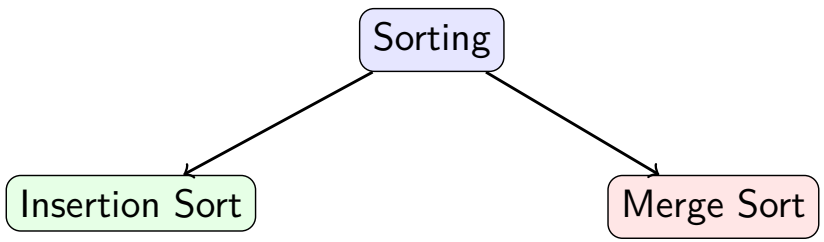
Milav Dabgar

Lecturer, GP Palanpur

July 22, 2026

Course Overview & Today's Agenda

- Quick recap of Bubble and Selection Sort
- Insertion Sort: Concept and Intuition
- Insertion Sort: Python Implementation and Complexity
- Introduction to Divide & Conquer Strategy
- Merge Sort: Splitting and Merging
- Merge Sort: Python Implementation and Complexity
- Comparison of Sorting Algorithms



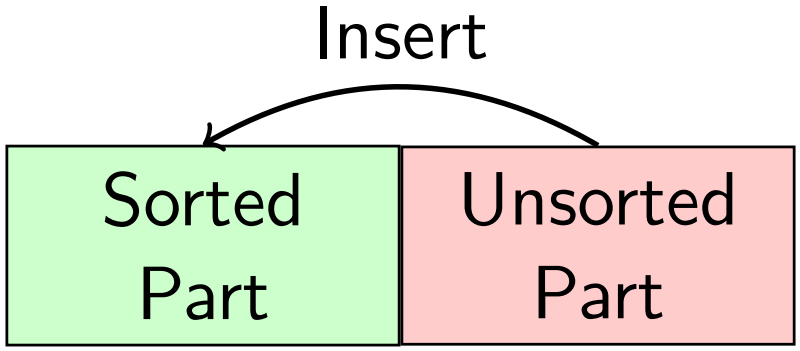
Recap: Bubble & Selection Sort

- Bubble Sort repeatedly swaps adjacent out-of-order elements until no swaps are needed.
- Selection Sort repeatedly finds the minimum element from the unsorted part and puts it at the beginning.
- Both algorithms have $O(N^2)$ time complexity, making them inefficient for large datasets.
- Today, we will learn Insertion Sort (also $O(N^2)$ but often faster in practice) and Merge Sort ($O(N \log N)$).

Algorithm	**Time Complexity**	**In-place**
Bubble Sort	$O(N^2)$	Yes
Selection Sort	$O(N^2)$	Yes

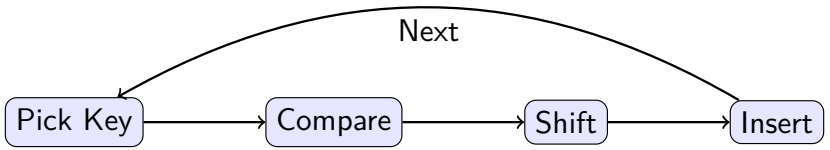
Introduction to Insertion Sort

- Think of how you sort playing cards in your hands: you take one card at a time and insert it into its correct position among the already sorted cards.
- Insertion Sort works the same way: it builds the final sorted array one item at a time.
- The array is virtually split into a 'sorted' part and an 'unsorted' part.
- Values from the unsorted part are picked and placed at the correct position in the sorted part.



Insertion Sort Algorithm Step-by-Step

1. Assume the first element (index 0) is already sorted.
2. Take the next element (index 1) and store it in a variable 'key'.
3. Compare 'key' with elements in the sorted sub-array (from right to left).
4. Shift all elements that are greater than 'key' one position to the right.
5. Insert 'key' into the correct, newly-opened position.
6. Repeat for all remaining elements.



Insertion Sort Trace Example

- Array: [4, 3, 2, 10, 12, 1, 5, 6]
- i=1, key=3: Compare with 4. Shift 4 right. Insert 3. Array: [3, 4, 2, 10, 12, 1, 5, 6]
- i=2, key=2: Compare with 4, 3. Shift both right. Insert 2. Array: [2, 3, 4, 10, 12, 1, 5, 6]
- i=3, key=10: $10 > 4$. No shifts needed. Array: [2, 3, 4, 10, 12, 1, 5, 6]
- i=4, key=12: $12 > 10$. No shifts needed. Array: [2, 3, 4, 10, 12, 1, 5, 6]

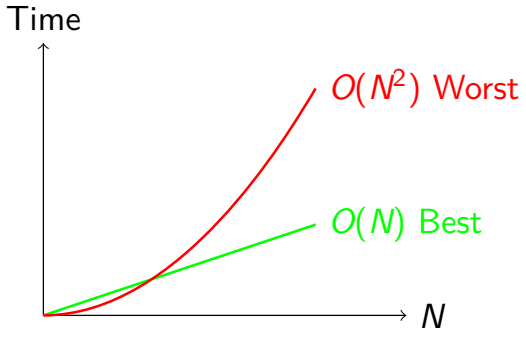
4	3	2	10	12	1	5	6
3	4	2	10	12	1	5	6
2	3	4	10	12	1	5	6

Insertion Sort in Python

```
def insertion_sort(arr):
    # Traverse from 1 to len(arr)
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1

        # Move elements of arr[0..i-1], that are
        # greater than key, to one position ahead
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1

        arr[j + 1] = key
    return arr
```

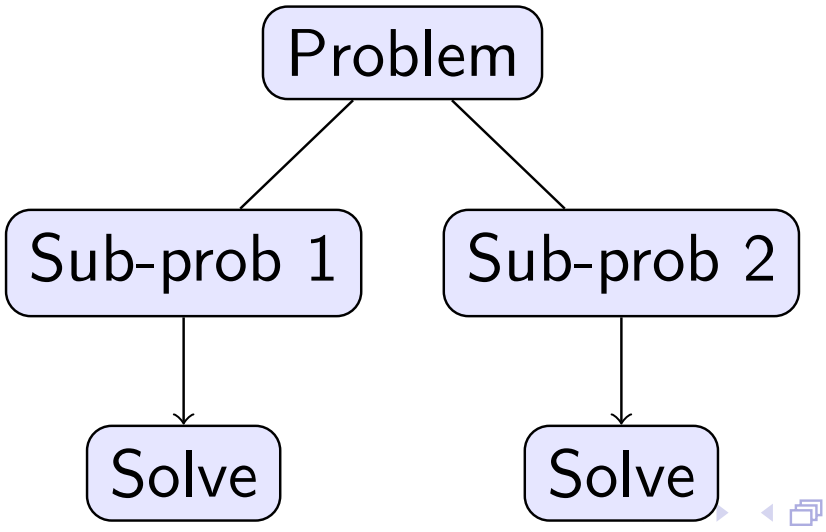
Insertion Sort Time Complexity

- Best Case: $O(N)$ - Occurs when the array is already sorted. The inner while-loop condition is false immediately.
- Worst Case: $O(N^2)$ - Occurs when the array is reverse sorted. Every element must be shifted to the very beginning.
- Average Case: $O(N^2)$ - On average, we scan half of the sorted sub-array.
- Space Complexity: $O(1)$ - It is an 'in-place' sorting algorithm (no extra arrays needed).
- Stability: Stable - Does not change the relative order of elements with equal keys.



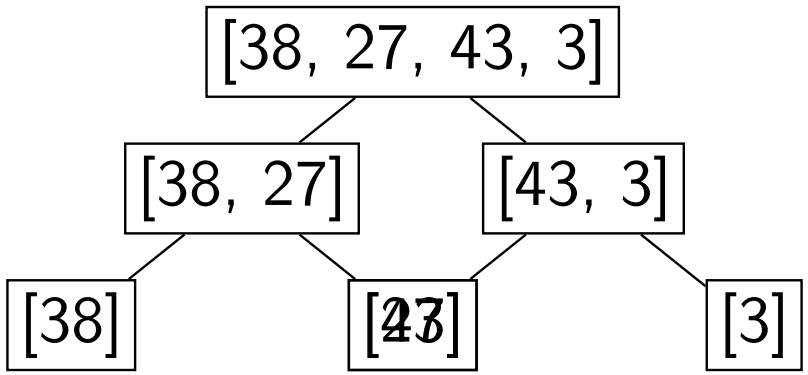
Introduction to Divide and Conquer

- We are now moving to a new class of algorithms: Divide and Conquer.
- **Divide**: Break the problem into smaller sub-problems that are similar to the original problem.
- **Conquer**: Solve the sub-problems recursively. If they are small enough, solve them directly (base case).
- **Combine**: Merge the solutions of the sub-problems to create the solution to the original problem.
- Merge Sort is a classic example of this paradigm.



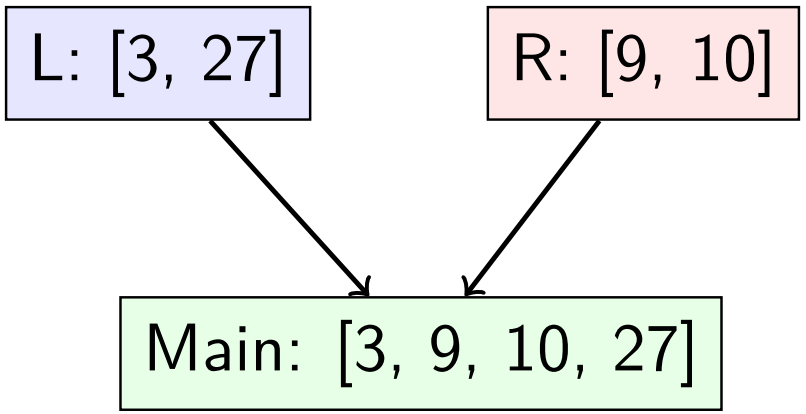
Merge Sort Concept

- Merge Sort continuously divides the array in half until each sub-array contains only a single element.
- An array of one element is considered sorted (Base Case).
- Then, it repeatedly merges the sub-arrays to produce new sorted sub-arrays until there is only one sorted array remaining.
- The heavy lifting happens in the 'Merge' step, where two sorted arrays are combined into a single sorted array.



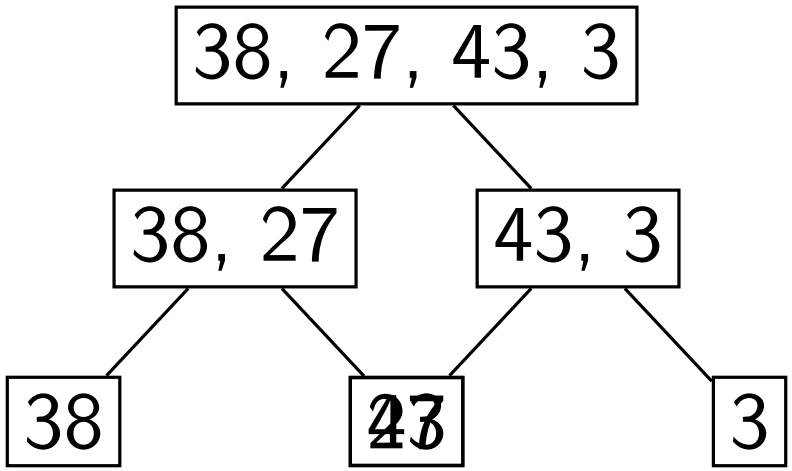
The Merge Step Logic

- How do we merge two already sorted arrays (Left and Right) efficiently?
- 1. Create pointers for Left (i), Right (j), and the Main array (k).
- 2. Compare Left[i] and Right[j].
- 3. Place the smaller element into Main[k] and increment the corresponding pointer.
- 4. Repeat until one array is exhausted.
- 5. Copy any remaining elements from the non-exhausted array.



Merge Sort Trace Example

- Initial: [38, 27, 43, 3, 9, 82, 10]
- Split: [38, 27, 43, 3] and [9, 82, 10]
- Split down to singletons: [38], [27], [43], [3], [9], [82], [10]
- Merge: [27, 38], [3, 43], [9, 82], [10]
- Merge: [3, 27, 38, 43] and [9, 10, 82]
- Final Merge: [3, 9, 10, 27, 38, 43, 82]



Merge Sort in Python: The Split

```
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        L = arr[:mid]
        R = arr[mid:]

        # Recursive calls to sort both halves
        merge_sort(L)
        merge_sort(R)

        # Next slide handles the merge step...
        # merge(arr, L, R)
```

Merge Sort in Python: The Merge

```
i = j = k = 0
# Copy data to temp arrays L[] and R[]
while i < len(L) and j < len(R):
    if L[i] < R[j]:
        arr[k] = L[i]
        i += 1
    else:
        arr[k] = R[j]
        j += 1
    k += 1

# Checking if any element was left
while i < len(L):
    arr[k] = L[i]; i += 1; k += 1
while j < len(R):
    arr[k] = R[j]; j += 1; k += 1
```

Merge Sort Complexity Analysis

- Time Complexity (Best, Worst, Average): $O(N \log N)$
- Why $O(N \log N)$? The array is halved $\log N$ times. At each level, merging takes N operations.
- Space Complexity: $O(N)$ - It is NOT an in-place sort. We need auxiliary space for the temporary sub-arrays (L and R).
- Stability: Stable - Equal elements retain their original relative order (important when sorting objects).

Time Complexity Formula

Total Time = (Number of Levels) $\times$ (Work per level) = $(\log N) \times (N) = O(N \log N)$

Comparison: Insertion vs Merge Sort

Insertion Sort:

- Best for small datasets or mostly sorted arrays ($O(N)$ best case).
- $O(1)$ space complexity (In-place).

Merge Sort:

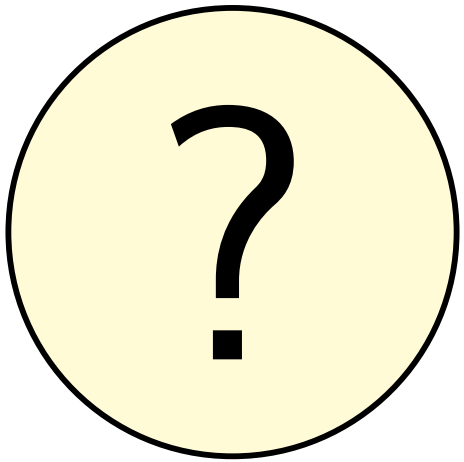
- Best for large datasets, guarantees $O(N \log N)$.
- $O(N)$ space complexity (Requires extra memory).

Fun Fact

Python's built-in Timsort algorithm is actually a hybrid of Merge Sort and Insertion Sort!

Quick Check: Test Your Understanding

- **Q:** Why is Insertion Sort highly efficient for an already sorted array?
- **Q:** What algorithmic paradigm does Merge Sort utilize?
- **Q:** If memory space is strictly limited, would you choose Insertion Sort or Merge Sort? Why?

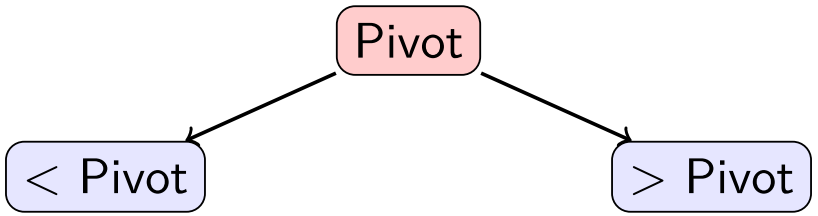


Key Takeaways

- Insertion Sort builds a sorted section one element at a time and is extremely fast ($O(N)$) for nearly sorted data.
- Divide and Conquer involves breaking a problem down, solving base cases, and merging results.
- Merge Sort splits arrays into singletons and merges them back in sorted order.
- Merge Sort guarantees $O(N \log N)$ time complexity but requires $O(N)$ auxiliary space.

Next Lecture Preview

- Lecture 24 completes our sorting unit by diving into Quick Sort, another divide-and-conquer algorithm that is often faster in practice and sorts in-place!



Questions?

Any Questions?

Lecture 4.4: Sorting: Quick Sort

Complete Lecture Deck – All 30 Lectures

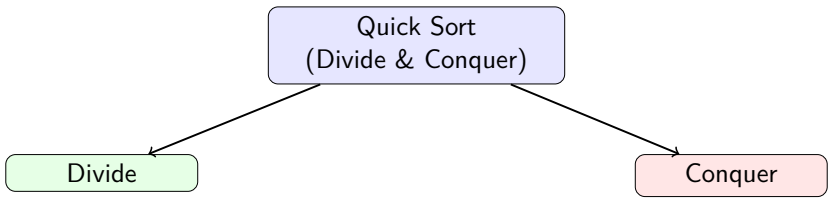
Milav Dabgar

Lecturer, GP Palanpur

July 22, 2026

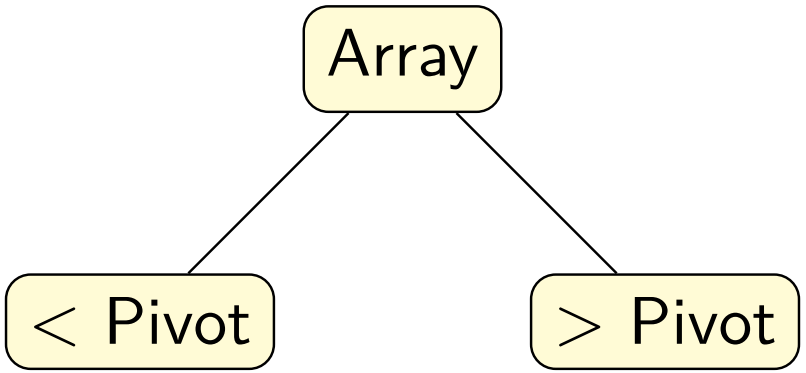
What is Quick Sort?

- Quick Sort is a highly efficient, general-purpose sorting algorithm.
- Developed by British computer scientist Tony Hoare in 1959.
- It uses a divide-and-conquer strategy to sort elements.
- In practice, it is often faster than Merge Sort and Heap Sort for in-memory sorting.



The Divide-and-Conquer Strategy

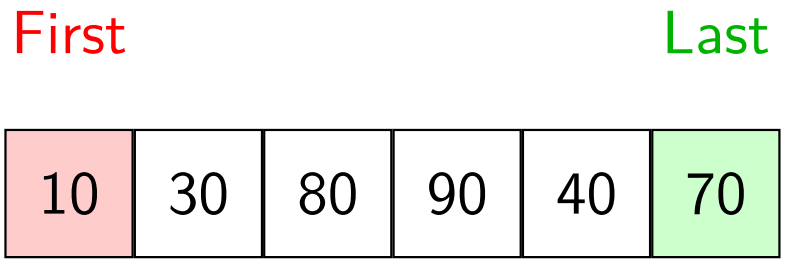
- **Divide:** Pick an element called a 'pivot' and partition the array into two sub-arrays.
- **Conquer:** Recursively sort the left sub-array (elements $<$ pivot) and right sub-array (elements $>$ pivot).
- **Combine:** Since the sub-arrays are sorted in place, no extra work is needed to combine them.



Choosing the Pivot Element

The choice of pivot heavily influences the performance of Quick Sort.
Common choices for the pivot:

1. First element of the array
2. Last element of the array (we'll use this for simplicity)
3. Random element
4. Median-of-three (first, middle, last)



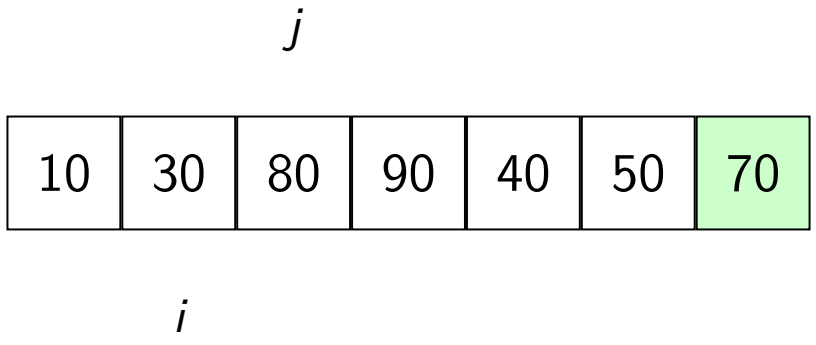
The Partitioning Process

- **Goal:** Place the pivot in its correct sorted position.
- **Rule 1:** All elements to the left of the pivot must be smaller than or equal to the pivot.
- **Rule 2:** All elements to the right must be greater than the pivot.
- This is achieved by scanning the array and swapping elements as needed.

Partitioning Example

Array: [10, 80, 30, 90, 40, 50, 70], Pivot = 70
i tracks the last smaller element. We iterate j from start to end-1.

- $10 \leq 70$ (swap 10, 10), $i = 0$
- $80 > 70$ (no swap)
- $30 \leq 70$ (swap 80, 30), $i = 1$, Array: [10, 30, 80, 90...]
- Finally, swap pivot with element at $i + 1$ to place it.



Python Implementation: Partition Function

```
def partition(arr, low, high):
    pivot = arr[high]
    i = low - 1
    for j in range(low, high):
        if arr[j] <= pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1
```

The Recursive Quick Sort Logic

- Once we partition the array, we get the index of the pivot, say `pi`.
- We then recursively call `quick_sort` on the left sub-array: from `low` to `pi - 1`.
- We also recursively call `quick_sort` on the right sub-array: from `pi + 1` to `high`.
- **Base case:** Recursion stops when `low` is no longer less than `high`.

Python Implementation: Quick Sort

```
def quick_sort(arr, low, high):
    if low < high:
        # Find pivot index
        pi = partition(arr, low, high)
        # Recursively sort left side
        quick_sort(arr, low, pi - 1)
        # Recursively sort right side
        quick_sort(arr, pi + 1, high)
```

Using the Quick Sort Function

```
# Example usage
data = [10, 80, 30, 90, 40, 50, 70]
n = len(data)

quick_sort(data, 0, n - 1)

print('Sorted array:', data)
```

Output

```
Sorted array: [10, 30, 40, 50, 70, 80, 90]
```

Time Complexity of Quick Sort

- **Best Case:** $O(n \log n)$ - Pivot always divides the array into two equal halves.
- **Average Case:** $O(n \log n)$ - Typical scenario for random data.
- **Worst Case:** $O(n^2)$ - Occurs when the array is already sorted (or reverse sorted) and we pick the first/last element as pivot.
- The worst case can be avoided using a randomized pivot.

Case	**Time Complexity**
Best	$O(n \log n)$
Average	$O(n \log n)$
Worst	$O(n^2)$

Space Complexity and Stability

- **Space Complexity:** $O(\log n)$ average, $O(n)$ worst case.
- Even though Quick Sort is an in-place sort, it requires extra space for the recursive call stack.
- **Stability:** Quick Sort is **NOT stable**.
- It can swap identical elements across the pivot, altering their relative initial order.

In-Place ✓

Stable ✗

Quick Sort vs. Merge Sort

- **Quick Sort:** In-place, $O(n^2)$ worst case, unstable.
- **Merge Sort:** Requires $O(n)$ extra space, $O(n \log n)$ worst case, stable.
- **Why prefer Quick Sort?** For arrays, Quick Sort often has smaller constant factors and better cache locality.
- Python's built-in `sort()` (Timsort) is actually a hybrid of Merge Sort and Insertion Sort!

Feature	**Quick Sort**	**Merge Sort**
Space	In-place ($O(\log n)$)	$O(n)$ extra space
Worst Time	$O(n^2)$	$O(n \log n)$
Stability	Unstable	Stable

Quick Check: Test Your Understanding

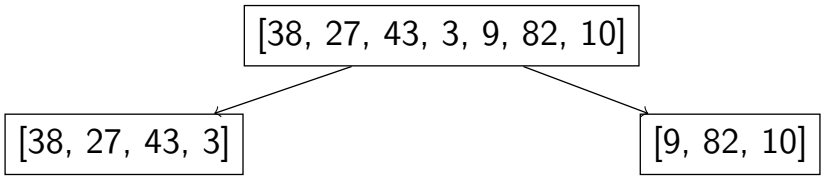
1. What is the main role of the 'pivot' in Quick Sort?
2. Does Quick Sort require creating a new list to hold sorted elements?
3. In what scenario does Quick Sort degrade to $O(n^2)$ time complexity?
4. Is Quick Sort a stable sorting algorithm?

Key Takeaways

- Quick Sort uses a Divide-and-Conquer strategy to achieve $O(n \log n)$ average performance.
- Partitioning places a chosen pivot in its correct final sorted position.
- It operates in-place, modifying the original Python list directly.
- While fast on average, poor pivot selection can degrade it to $O(n^2)$.

Next Lecture Preview

- In the next lecture, we will explore Merge Sort, another divide-and-conquer algorithm.
- We'll see how it guarantees $O(n \log n)$ performance even in the worst case!



Questions?

Any Questions?

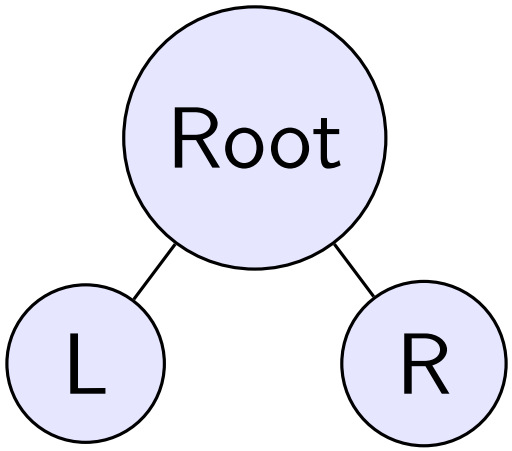
Lecture 5.1: Introduction to Binary Trees

Complete Lecture Deck – All 30 Lectures

Milav Dabgar

Lecturer, GP Palanpur

July 22, 2026



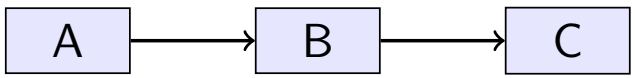
Agenda

- Transitioning from Linear to Non-Linear Data Structures
- What is a Tree?
- Basic Tree Terminology (Root, Leaf, Edge)
- Node Properties: Degree, In-degree, Out-degree
- Tree Structure: Level Number & Height
- Introduction to Binary Trees
- Complete Binary Trees
- Python Representation of a Binary Tree Node

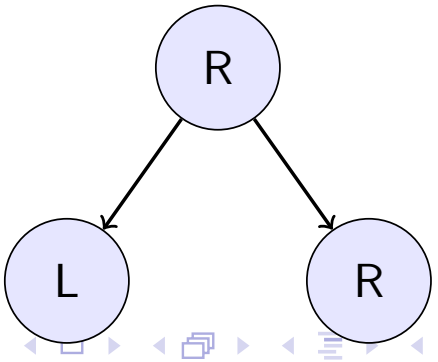
Linear vs. Non-Linear Data Structures

- **Linear Structures**: Data is organized sequentially (e.g., Arrays, Linked Lists). Each element has a unique predecessor and successor.
- **Non-Linear Structures**: Data is organized hierarchically. Elements can have multiple connections.
- **Why non-linear?** To represent hierarchical relationships like file systems, organizational charts, or family trees.
- Trees allow for faster search and retrieval compared to simple linear structures.

Linear (Linked List)

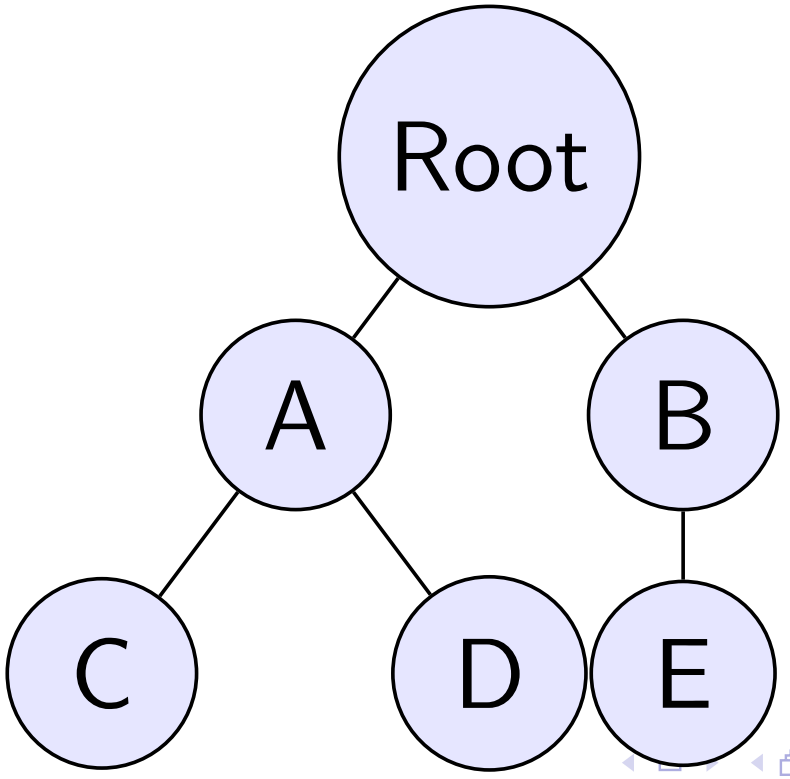


Non-Linear (Tree)



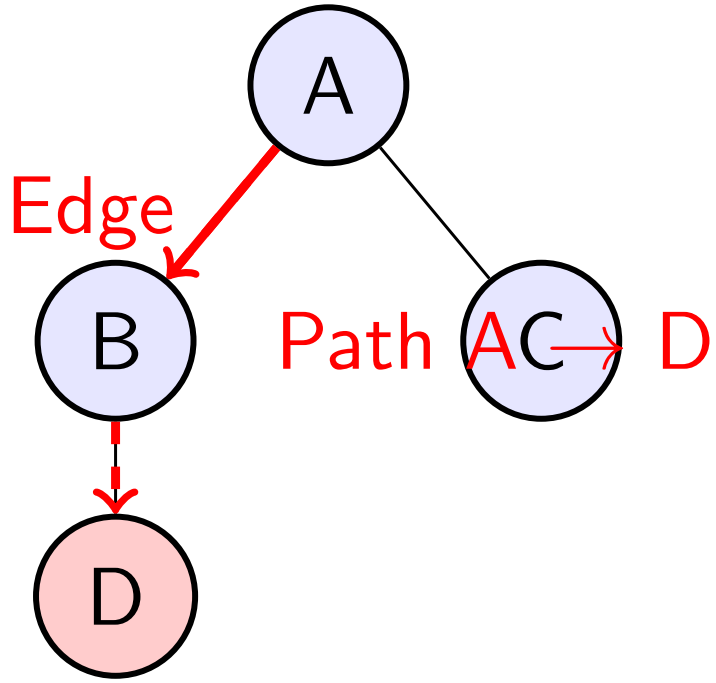
What is a Tree?

- A Tree is a collection of entities called nodes linked together to simulate a hierarchy.
- It consists of a distinguished node called the 'root' and zero or more subtrees.
- Nodes are connected by edges.
- Crucially, a tree contains no cycles (no closed loops).



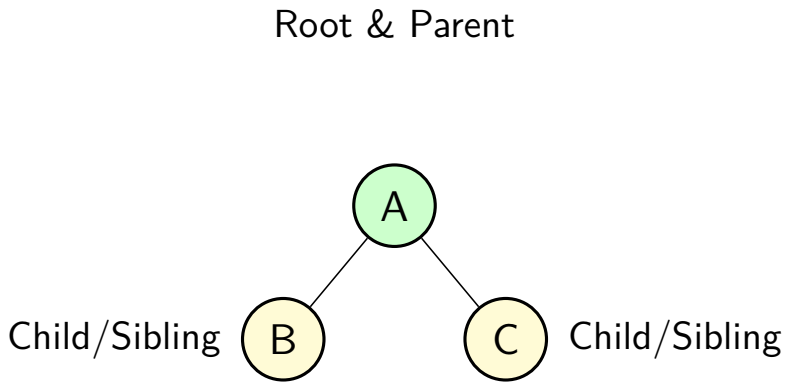
Basic Terms: Nodes and Edges

- **Node**: A fundamental part of a tree, containing data and links to other nodes.
- **Edge**: The connecting link between two nodes.
- **Path**: A sequence of nodes and edges connecting a node with a descendant.
- **A tree with 'N' nodes always has exactly 'N-1' edges.**



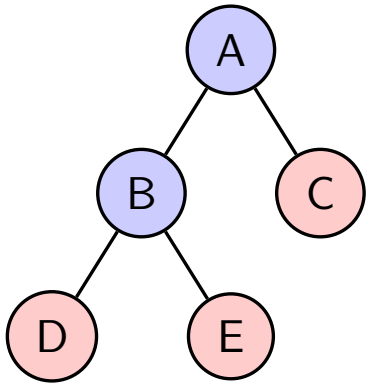
Basic Terms: Root, Parent, and Child

- **Root Node**: The topmost node in the tree. It has no parent.
- **Parent Node**: A node that has branches extending to other nodes (its children).
- **Child Node**: A node that is a descendant of another node.
- **Siblings**: Nodes that share the same parent.



Basic Terms: Leaf and Internal Nodes

- **Leaf Node** (Terminal Node): A node with no children (out-degree is 0).
- **Internal Node** (Non-Terminal Node): A node with at least one child.
- The root can be an internal node, unless the tree has only one node (the root itself).

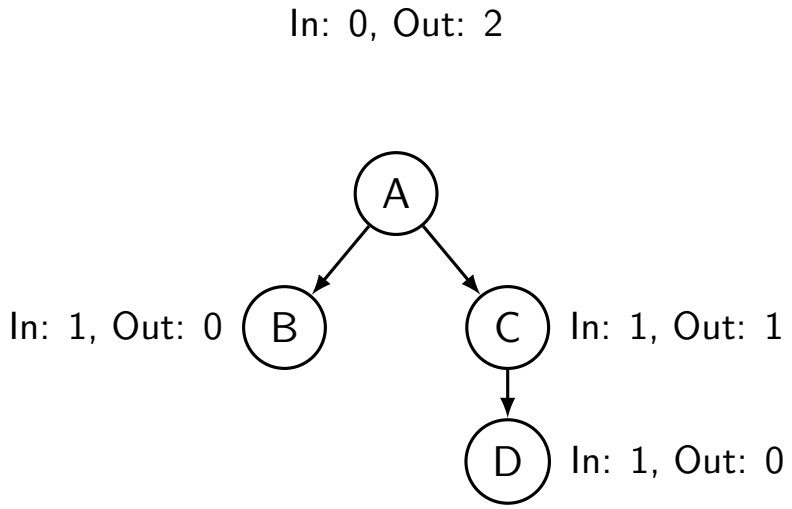


Internal Nodes

Leaf Nodes

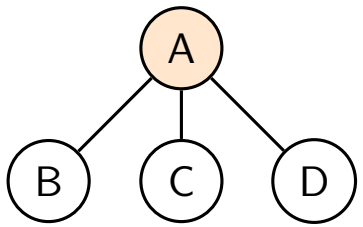
Node Degree, In-degree, and Out-degree

- **In-degree**: The number of edges arriving at a node.
- **Out-degree**: The number of edges leaving a node (number of children).
- **Degree of a node**: In trees, this usually refers to the out-degree.
- The in-degree of the root is always 0. The in-degree of every other node is exactly 1.



Tree Degree

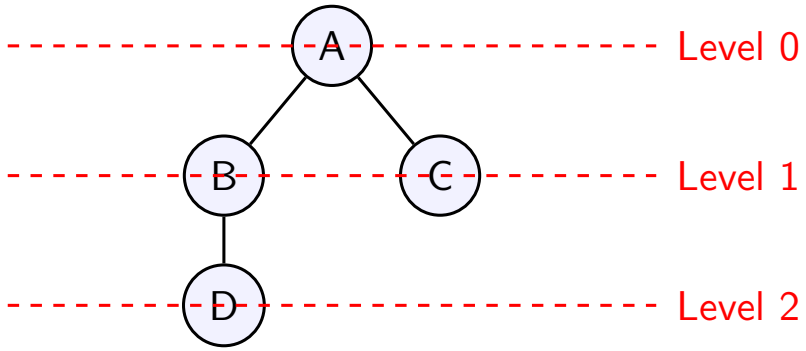
- **Degree of a Tree**: The maximum degree of any node in the tree.
- If a tree has nodes with out-degrees of 0, 1, 2, and 3, the degree of the tree is 3.
- A binary tree has a maximum degree of 2.



Node A has 3 children
$\Rightarrow$ Max Degree = 3
$\Rightarrow$ Tree Degree = 3

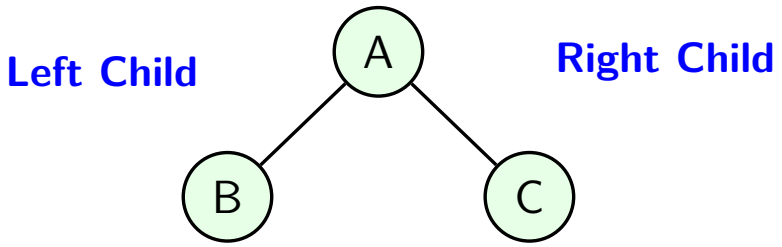
Level Number and Depth

- **Level**: The rank of the hierarchy. The root is at Level 0.
- Its children are at Level 1, grandchildren at Level 2, etc.
- **Depth of a node**: The length of the path from the root to the node.
- **Height of a node**: The length of the longest path from the node to a leaf.



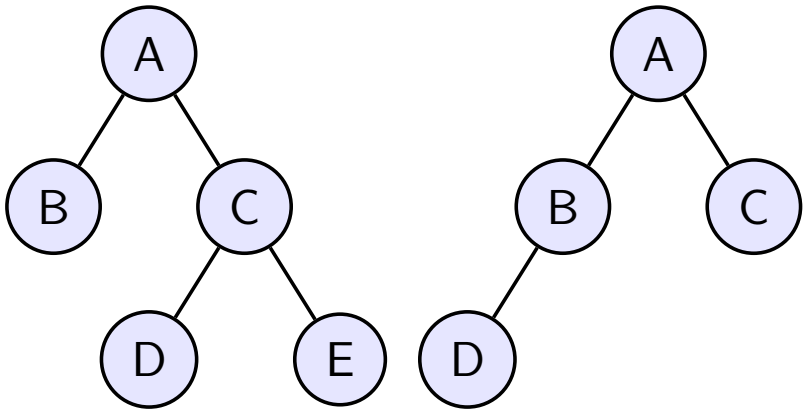
Introduction to Binary Trees

- **Binary Tree**: A tree in which every node has at most two children.
- The children are specifically referred to as the 'left child' and the 'right child'.
- A binary tree can be empty (no nodes).
- Because it has at most 2 children, the degree of a binary tree is at most 2.



Types of Binary Trees (Strict/Full)

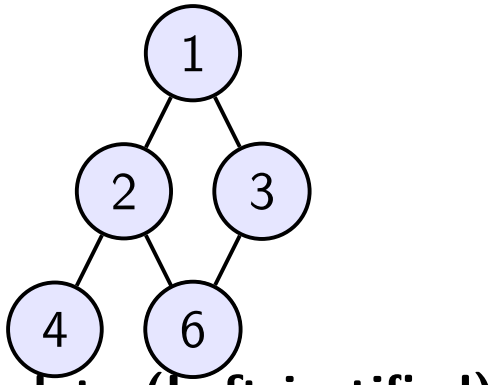
- **Strictly Binary Tree (Full Binary Tree)**: Every node has either 0 or 2 children.
- No node has exactly 1 child.
- Also known as a proper binary tree.
- Useful in representing mathematical expressions.



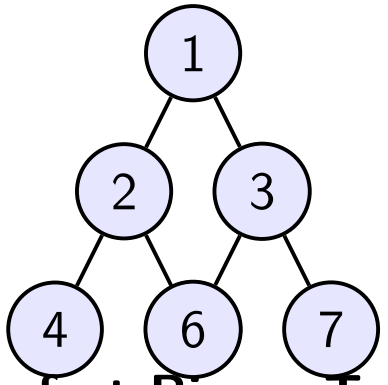
Full (Strict) **Not Full**

Complete Binary Tree

- **Complete Binary Tree**: All levels are fully filled except possibly the last level.
- In the last level, all nodes are filled as far left as possible.
- **Perfect Binary Tree**: A complete binary tree where the last level is also fully filled.
- Complete binary trees are highly efficient for array-based representation (Heaps).



Complete (Left-justified)



Perfect Binary Tree

Python: Defining a Binary Tree Node

- In Python, we use a class to define a node.
- It needs three parts: the data, a reference to the left child, and a reference to the right child.

Binary Tree Node Structure

Code Example: Node Class

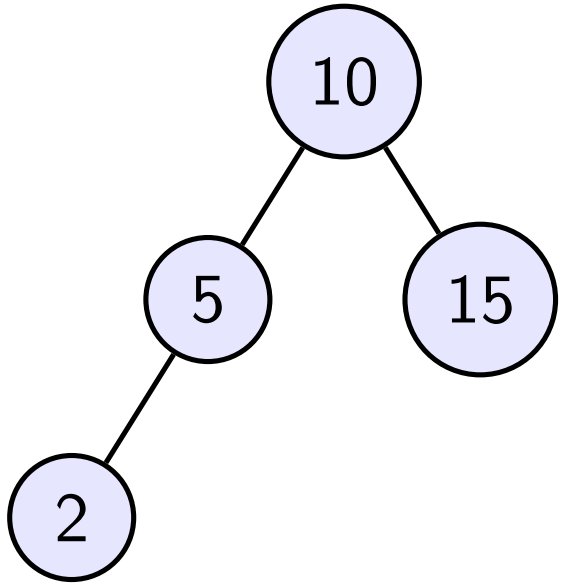
- Below is the Python class representation for a Binary Tree Node.

```
class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

# Creating the root node
root = Node(10)
```

Code Example: Building a Simple Tree

```
# Root node
root = Node(10)

# Adding children
root.left = Node(5)
root.right = Node(15)

# Adding a grandchild
root.left.left = Node(2)
```

This creates a small hierarchical structure manually.



Test Your Understanding

Quiz Time

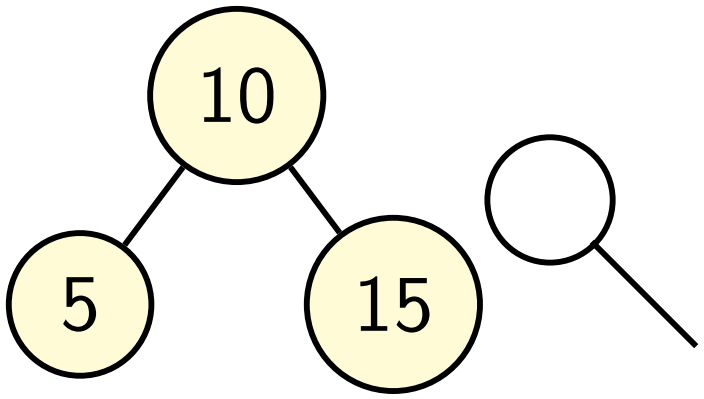
1. If a tree has 15 nodes, how many edges does it have?
2. What is the in-degree of the root node?
3. In a binary tree, what is the maximum number of children a node can have?
4. Is a tree with one node a complete binary tree?

Key Takeaways

- **Trees** are non-linear data structures used to represent hierarchical relationships.
- Key terms include **Root, Leaf, Node, Edge, Level, and Degree**.
- A **binary tree** restricts nodes to a maximum of two children (left and right).
- **Complete binary trees** have all levels filled, with the last level strictly left-aligned.
- We implement tree nodes in Python using a class with `data`, `left`, and `right` attributes.

Next Lecture Preview

- In Lecture 26, we will introduce the **Binary Search Tree (BST)**, a special type of binary tree that makes searching incredibly fast, and explore how to search for and insert data into it.



Questions?

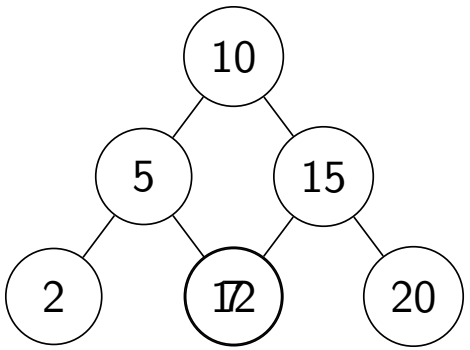
Any Questions?

Agenda

- What is a Binary Search Tree (BST)?
- The BST Property/Rule
- Valid vs. Invalid BSTs
- Why use a BST? (Search efficiency)
- The Search Algorithm in BST
- Tracing the Search operation
- Python Implementation of BST Search
- Time Complexity Analysis

Introduction to Binary Search Tree

- A Binary Search Tree (BST) is a binary tree with an additional structural property.
- It maintains data in an ordered fashion, allowing for fast lookup, addition, and removal of items.
- Think of it as a dynamic version of a sorted array combined with the structural flexibility of a linked list.
- The fundamental rule applies to every single node in the tree.

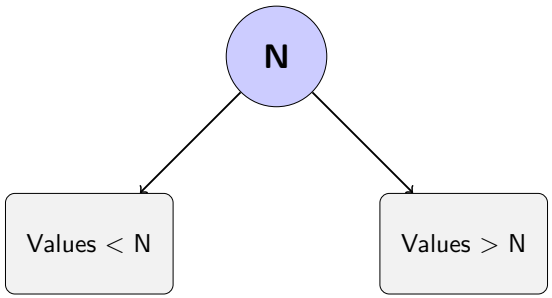


The BST Property (The Golden Rule)

For ANY given node N in the tree:

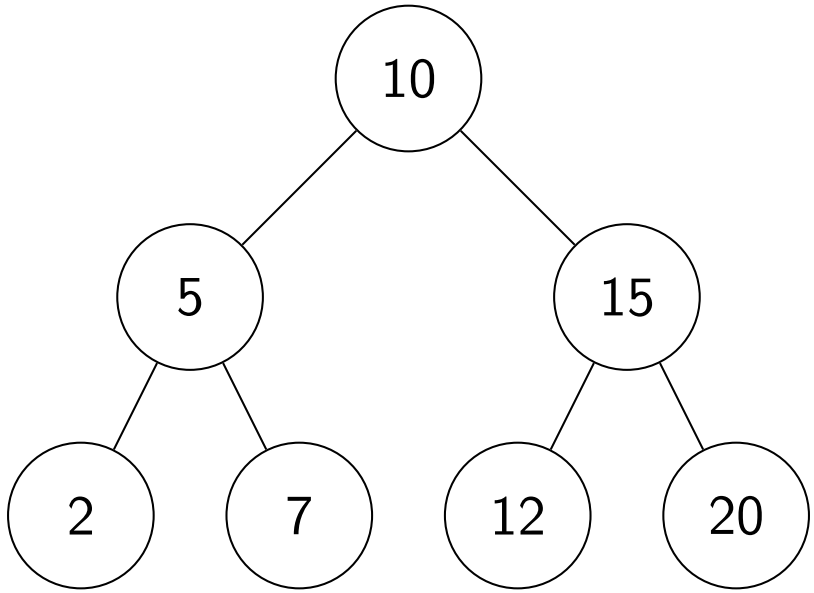
1. All values in the LEFT subtree of N are STRICTLY LESS THAN the value of N.
2. All values in the RIGHT subtree of N are STRICTLY GREATER THAN the value of N.
3. Both the left and right subtrees must also be valid binary search trees.

Note: Duplicate values are typically not allowed, or handled via specific rules.



Validating a BST: Is this a BST?

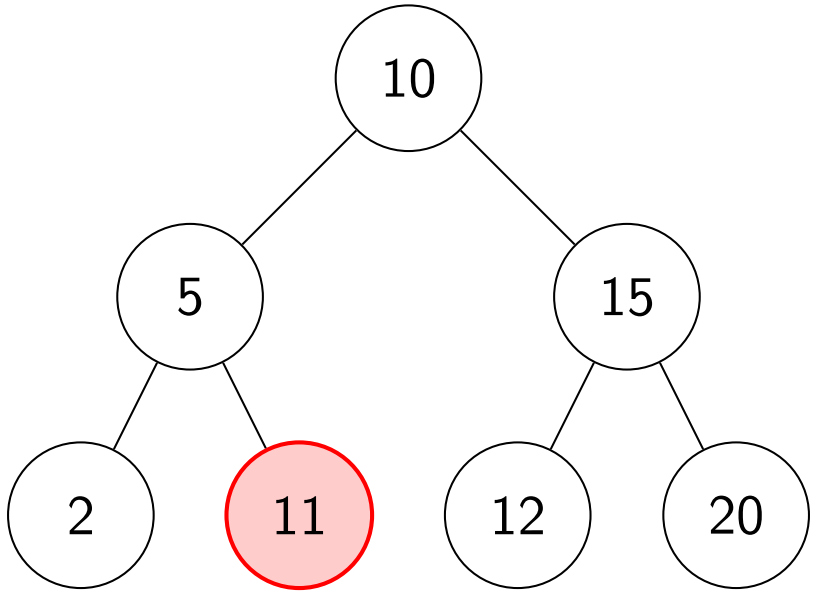
Let's look at an example tree:



Yes! It follows the BST property for all nodes.

Validating a BST: What about this?

Example 2:



No! 11 is in the left subtree of 10, but 11 is not less than 10.

Why Use a BST?

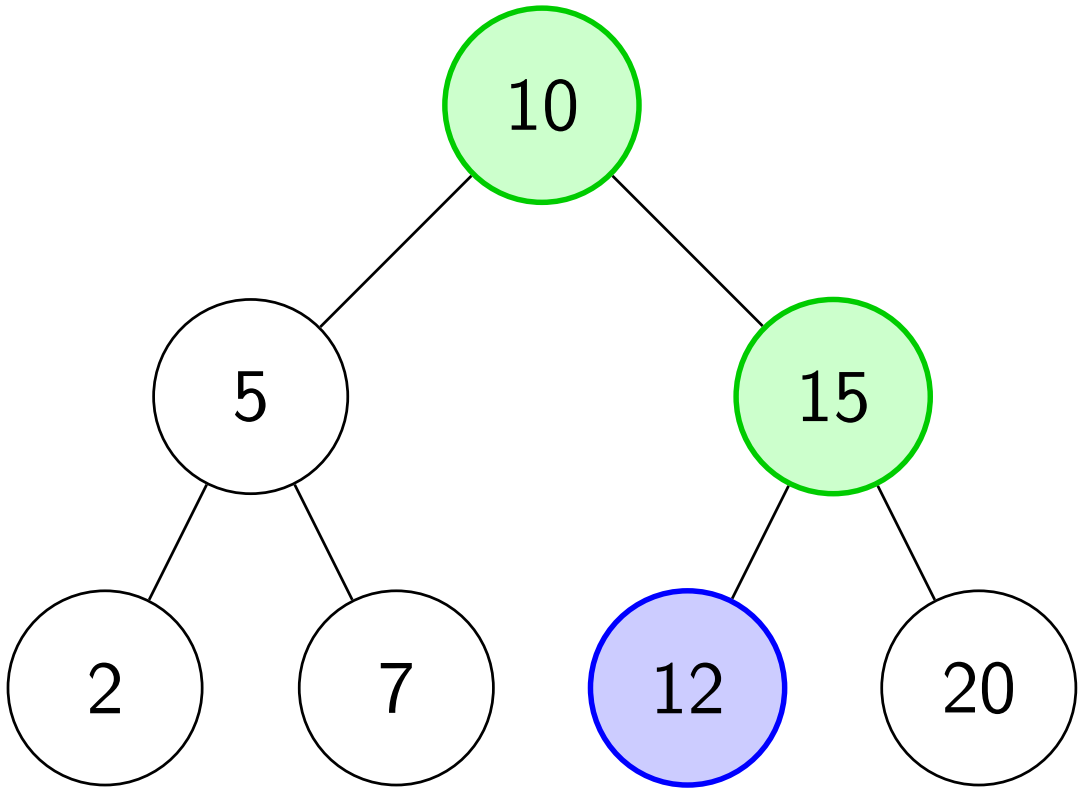
- In a regular binary tree, searching for an element requires checking potentially every node ($O(N)$ time).
- In a BST, the structural property allows us to eliminate half the remaining tree at each step.
- If the target is less than the current node, we only search the left subtree.
- If the target is greater, we only search the right subtree.
- This is identical in logic to Binary Search in a sorted array!

The Search Algorithm (Intuition)

1. Start at the root node.
2. Compare the target value with the current node's value.
3. If they match, you found it! Return the node.
4. If target $<$ current node, move to the left child and repeat step 2.
5. If target $>$ current node, move to the right child and repeat step 2.
6. If you reach a null (empty) node, the target is not in the tree.

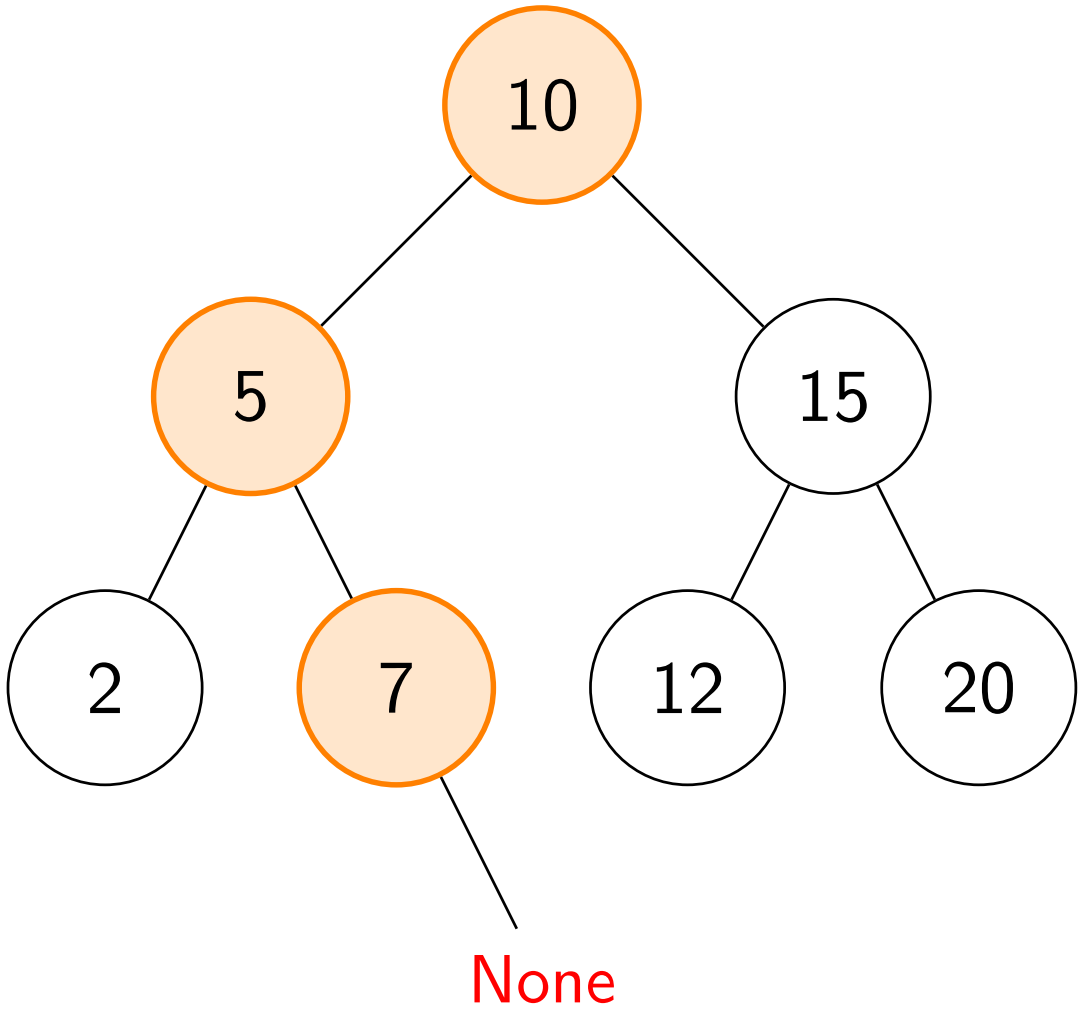
Tracing Search: Find 12

- **Target:** 12
- **Step 1:** Current = 10. $12 > 10$. Go Right.
- **Step 2:** Current = 15. $12 < 15$. Go Left.
- **Step 3:** Current = 12. $12 == 12$. Found!



Tracing Search: Find 8

- **Target:** 8
- **Step 1:** Current = 10. $8 < 10$. Go Left.
- **Step 2:** Current = 5. $8 > 5$. Go Right.
- **Step 3:** Current = 7. $8 > 7$. Go Right.
- **Step 4:** Current = None. Target 8 not found.



Python Implementation: Iterative Search

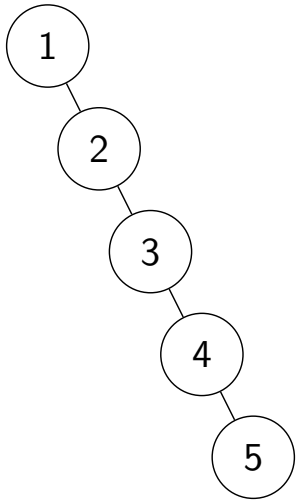
```
def search_iterative(root, target):
    current = root
    while current is not None:
        if current.data == target:
            return current  # Found
        elif target < current.data:
            current = current.left
        else:
            current = current.right
    return None  # Not found
```

Python Implementation: Recursive Search

```
def search_recursive(node, target):
    # Base cases: root is null or key is present at root
    if node is None or node.data == target:
        return node

    # Key is greater than root's key
    if node.data < target:
        return search_recursive(node.right, target)

    # Key is smaller than root's key
    return search_recursive(node.left, target)
```

Time Complexity Analysis

- **Best Case:** $O(1)$ - The target is the root node.
- **Average Case:** $O(\log N)$ - The tree is somewhat balanced. We eliminate half the nodes each step.
- **Worst Case:** $O(N)$ - The tree is completely skewed (like a linked list).
- **Space Complexity:** $O(1)$ for iterative, $O(h)$ for recursive (where h is height of tree).



Addressing the Worst Case

- How do we prevent a BST from degrading to $O(N)$?
- We use **self-balancing binary search trees**.
- Examples include AVL Trees and Red-Black Trees.
- These trees automatically adjust their structure during insertion/deletion to keep the height around $O(\log N)$.
- We will briefly discuss AVL trees later in the course.

Test Your Understanding

1. **Q1:** Is this a valid BST? Root=20, L=10, R=30, L.L=5, L.R=25.
2. **Q2:** In a balanced BST of 1,000,000 nodes, roughly how many comparisons does it take to find an element?
3. **Q3:** Why might we choose iterative search over recursive in Python?

Key Takeaways

- A BST imposes a strict ordering: Left subtrees $<$ Node $<$ Right subtrees.
- This property allows us to search for elements efficiently by halving the search space.
- Average time complexity for search is $O(\log N)$.
- Worst-case time complexity is $O(N)$ if the tree becomes skewed.
- Search can be implemented both iteratively and recursively in Python.

Next Lecture Preview

In Lecture 27, we will learn how to build a BST from scratch by exploring the Insertion algorithm, ensuring the BST property is maintained every time a new node is added.

Any Questions?

Lecture 5.3: BST Insertion

Complete Lecture Deck – All 30 Lectures

Milav Dabgar

Lecturer, GP Palanpur

July 22, 2026

The Goal of BST Insertion

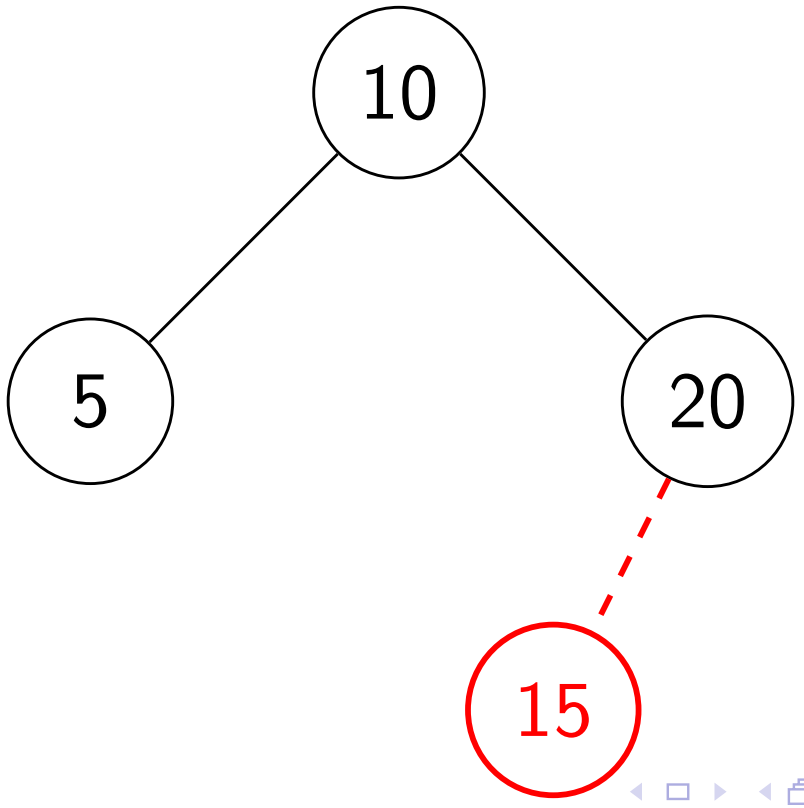
- When we insert a new value into a BST, we must find the ONE correct spot for it.
- Crucially, the tree MUST remain a valid BST after the insertion.
- **New nodes are ALWAYS inserted as LEAF nodes.**
- We never insert a node in the middle of the tree and push existing nodes down (unless we are rebalancing, which we aren't doing here).

The Insertion Algorithm Logic

- 1. Start at the root.
- 2. If the tree is empty, the new node becomes the root. Done.
- 3. Compare the new value to the current node's value.
- 4. If new value $<$ current node, go LEFT. If left child is empty, attach new node here. Done. Otherwise, repeat step 3.
- 5. If new value $>$ current node, go RIGHT. If right child is empty, attach new node here. Done. Otherwise, repeat step 3.
- 6. (Optional) If value $==$ current node, handle duplicate (ignore, or put on right).

Tracing Insertion: Insert 15

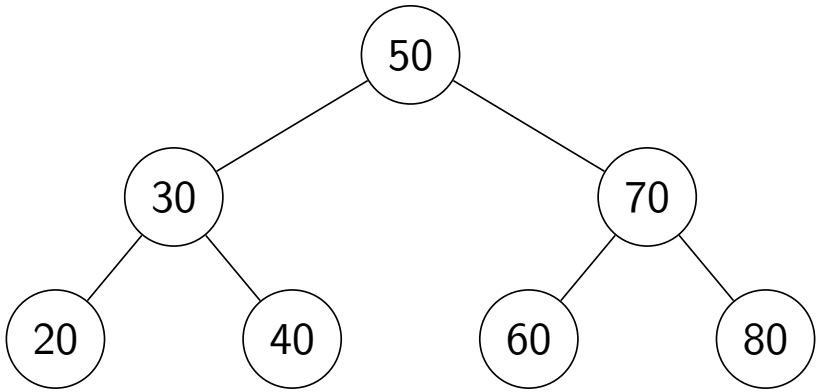
- Current Tree: Root=10. Left=5, Right=20.
- Step 1: Start at root (10).
- Step 2: $15 > 10$. Go Right.
- Step 3: Current node is 20. $15 < 20$. Go Left.
- Step 4: 20's Left is empty. Attach 15 as Left child of 20.



Building a Tree from Scratch

Let's build a BST by inserting these numbers in order: [50, 30, 70, 20, 40, 60, 80]

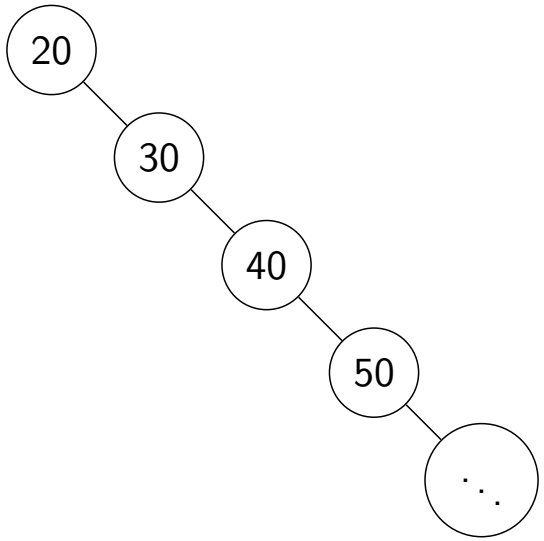
- Step 1: 50 (Root)
- Step 2: 30 (<50, left of 50)
- Step 3: 70 (>50, right of 50)
- Step 4: 20 (<50, <30, left of 30)



The Impact of Insertion Order

Now let's insert the SAME numbers in a different order: [20, 30, 40, 50, 60, 70, 80]

- Step 1: 20 (Root)
- Step 2: 30 ($>$20, right of 20)
- Step 3: 40 ($>$20, $>$30, right of 30)
- Result: A 'skewed' tree that looks exactly like a Linked List!



Python Implementation: Iterative Setup

```
class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

def insert_iterative(root, data):
    new_node = Node(data)
    if root is None:
        return new_node
```

Python Implementation: Iterative Loop

```
current = root
while True:
    if data < current.data:
        if current.left is None:
            current.left = new_node
            break
        current = current.left
    else: # data >= current.data
        if current.right is None:
            current.right = new_node
            break
        current = current.right
return root
```

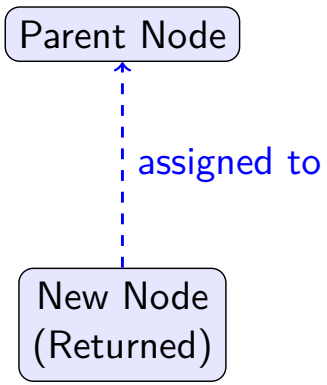
Python Implementation: Recursive

```
def insert_recursive(root, data):
    if root is None:
        return Node(data)

    if data < root.data:
        root.left = insert_recursive(root.left, data)
    else:
        root.right = insert_recursive(root.right, data)

    return root
```

Understanding the Recursive Return

Why do we say `root.left = insert_recursive(...)`?

- When `insert_recursive` hits a None, it creates a new Node and returns it.
- The caller (the parent node) receives that new Node and assigns it to its `left` or `right` pointer.
- For nodes that already exist, it just returns the node itself, effectively saying `root.left = root.left`, keeping the structure intact.



Time Complexity Analysis (Insertion)

Because insertion follows the exact same path as a search operation...

Case	**Complexity**
Best Case	$O(1)$ - Inserting into an empty tree.
Average Case	$O(\log N)$ - Assuming the tree is reasonably balanced.
Worst Case	$O(N)$ - Inserting into a skewed tree (essentially traversing a linked list).
Space (Iterative)	$O(1)$
Space (Recursive)	$O(h)$ - where h is the height of the tree.

Test Your Understanding

- Draw the BST resulting from inserting these numbers in order: 10, 5, 15, 2, 7, 12, 20.
- What is the height of the resulting tree?
- If you insert the sequence 1, 2, 3, 4, 5, what is the height?

Key Takeaways

- New nodes are always inserted as leaf nodes in a standard BST.
- The logic mirrors searching: go left if smaller, go right if larger, until an empty spot is found.
- The shape and performance of a BST depend heavily on the order in which data is inserted.
- Insertion can be implemented iteratively (using a while loop) or recursively.

Next Lecture Preview

In Lecture 28, we will tackle the most complex BST operation: Deletion. Removing a leaf is easy, but what happens when you need to delete a node that has two children?

Questions?

Lecture 28: BST Deletion

Complete Lecture Deck – All 30 Lectures

Milav Dabgar

Lecturer, GP Palanpur

July 22, 2026

The Challenge of Deletion

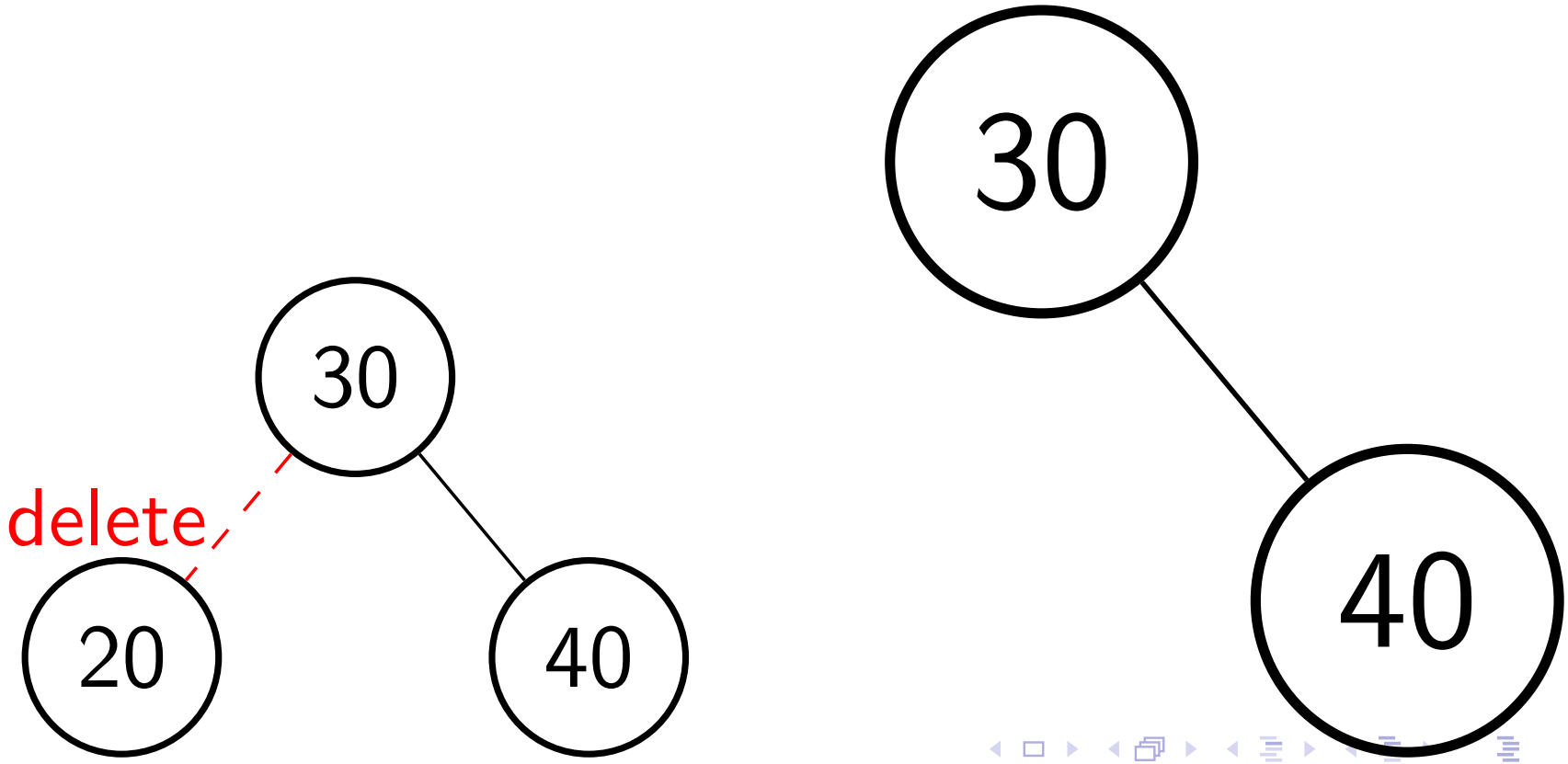
- When you search or insert, you just traverse down the tree.
- When you delete, you might remove a node that is holding other nodes together.
- Rule: After deletion, the remaining tree MUST still be a valid Binary Search Tree.
- We must 'patch' the hole left by the deleted node without violating the left $<$ node $<$ right property.

The Three Cases of Deletion

- Depending on the node we want to delete, we face three different scenarios:
- Case 1: The node is a leaf (0 children).
- Case 2: The node has exactly 1 child (either left or right).
- Case 3: The node has exactly 2 children (both left and right).
- We handle each case differently.

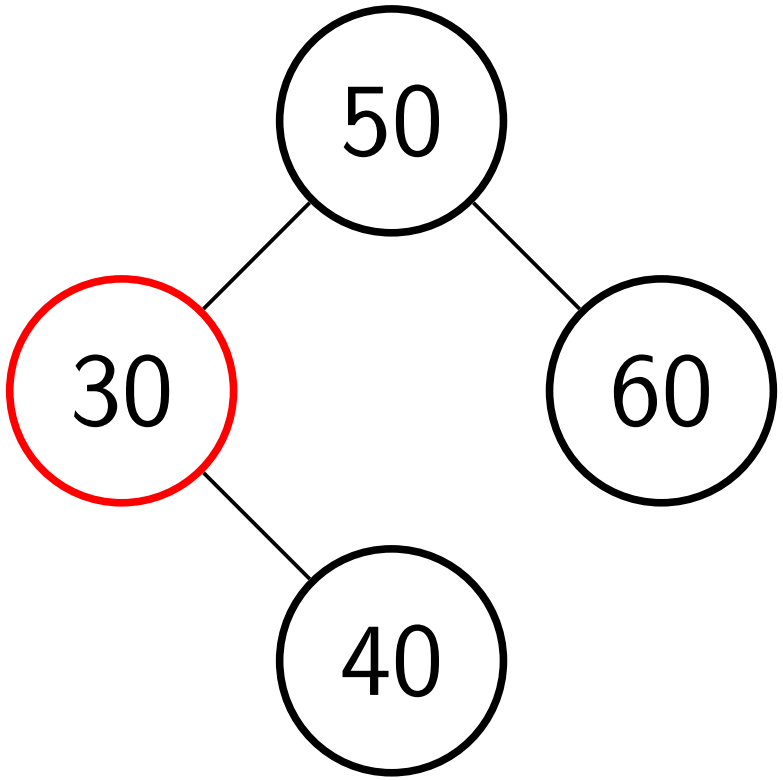
Case 1: Deleting a Leaf Node

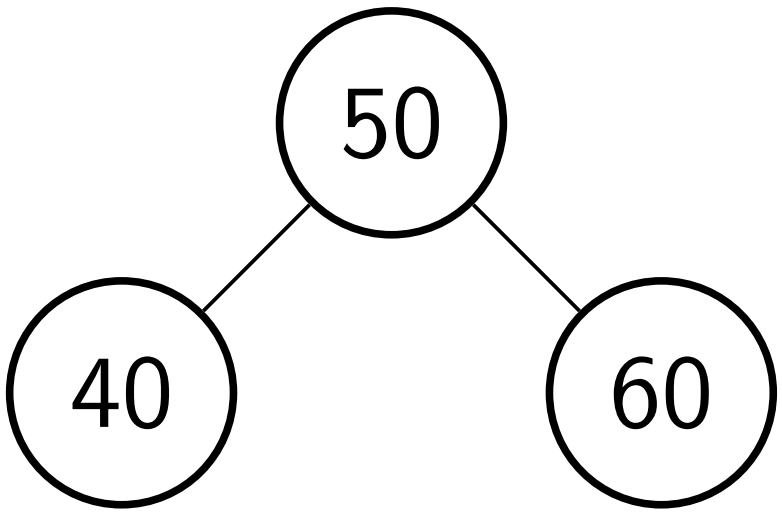
- Scenario: The node to be deleted has no children.
- Action: Simply remove the node from the tree.
- How? Update the parent's pointer (left or right) to point to None.
- Example: Delete 20 from a tree where 20 is a leaf.



Case 2: Deleting a Node with One Child

- Scenario: The node to be deleted has exactly one child.
- Action: Remove the node and replace it with its child.
- How? Connect the parent of the deleted node directly to the child of the deleted node.
- Think of it as 'bypassing' the deleted node.



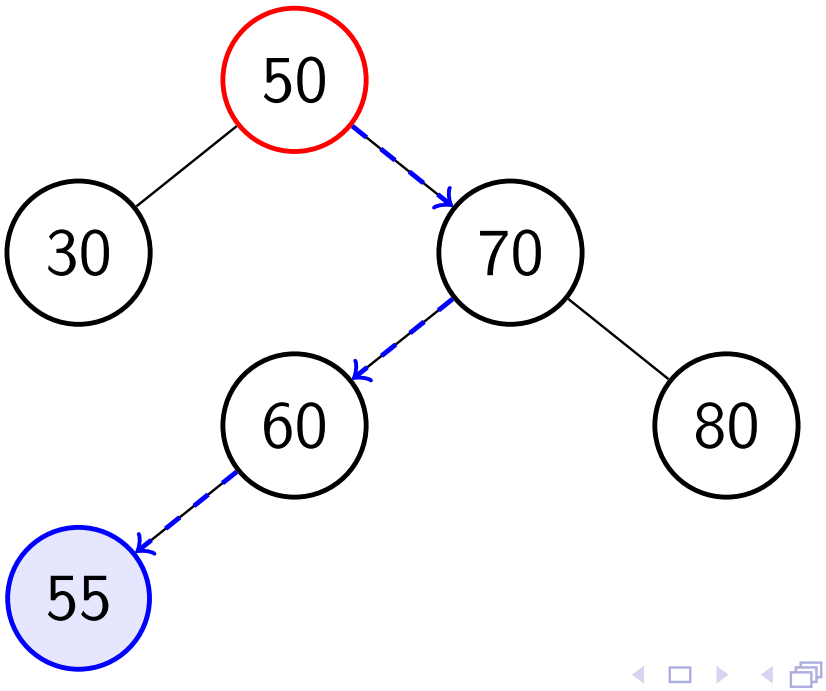


Case 3: Deleting a Node with Two Children

- Scenario: The node to be deleted has BOTH a left and right child.
- Action: We cannot simply bypass it. We have two subtrees and only one parent pointer!
- Solution: Find a suitable 'replacement' node from further down the tree.
- The replacement must be chosen carefully to maintain the BST property.
- We use either the In-order Predecessor or In-order Successor.

Finding the Replacement (In-order Successor)

- In-order Successor: The node with the smallest value that is strictly greater than the node being deleted.
- How to find it: Go to the RIGHT child, then go as far LEFT as possible.
- Why it works: Because it's the smallest value on the right, it is greater than everything on the left, but smaller than everything else on the right.



Case 3: Step-by-Step Logic

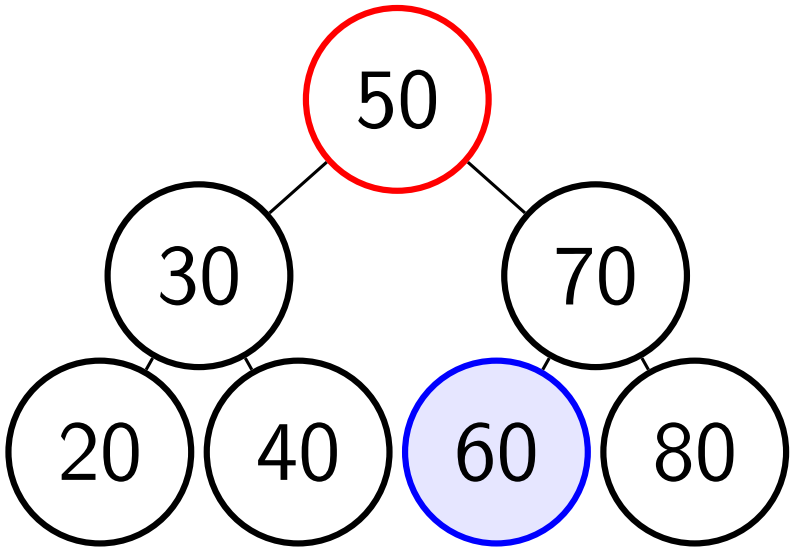
1. Find the In-order Successor of the node to be deleted.
2. Copy the data from the Successor into the node to be deleted (overwriting its value).
3. The node is now 'replaced'. But we have a duplicate value further down!
4. Recursively delete the old Successor node from the right subtree.

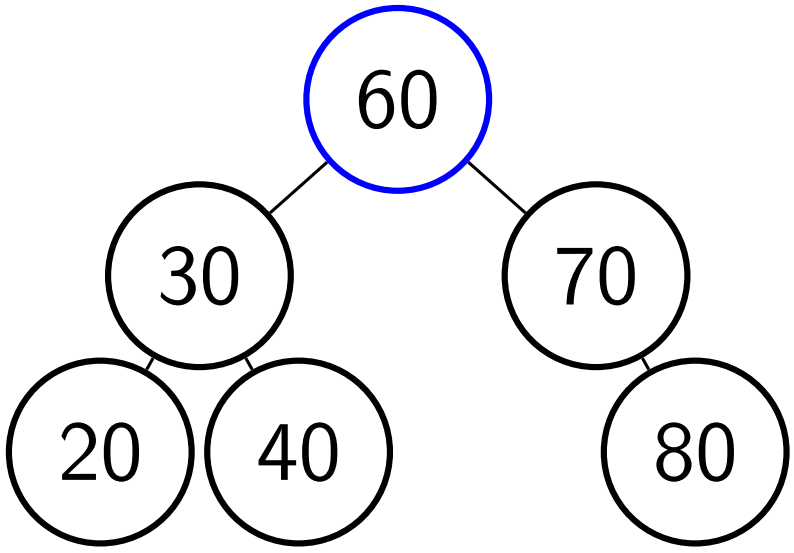
Important Note

The old Successor will ALWAYS be a Case 1 or Case 2 deletion (it cannot have a left child by definition)!

Trace Case 3: Delete 50 (Root)

- Tree: Root(50). L(30), R(70). L's children: (20, 40). R's children: (60, 80).
- Step 1: Delete 50. It has 2 children.
- Step 2: Find successor $\rightarrow$ Right child (70), then all the way left (60). Successor is 60.
- Step 3: Replace 50's value with 60. Root is now 60.
- Step 4: Delete the node 60 from the right subtree (Case 1).





Python: Helper Function for Successor

- First, we need a small helper function to find the minimum value node in a given subtree.

```
def min_value_node(node):
    current = node
    # loop down to find the leftmost leaf
    while current.left is not None:
        current = current.left
    return current
```

Python Implementation: Deletion (Part 1)

```
def delete_node(root, key):
    if root is None: return root

    # Searching for the node to delete
    if key < root.data:
        root.left = delete_node(root.left, key)
    elif key > root.data:
        root.right = delete_node(root.right, key)
    else:
        # Node found! Handle cases below...
```

Python Implementation: Deletion (Part 2)

```
# Case 1 & 2: Node with only one child or no child
if root.left is None:
    temp = root.right
    root = None
    return temp
elif root.right is None:
    temp = root.left
    root = None
    return temp
```

Python Implementation: Deletion (Part 3)

```
        # Case 3: Node with two children
        # Get the inorder successor (smallest in the right
            subtree)
        temp = min_value_node(root.right)

        # Copy the inorder successor's content to this node
        root.data = temp.data

        # Delete the inorder successor
        root.right = delete_node(root.right, temp.data)

    return root
```

Test Your Understanding

- Q1: When deleting a leaf node, what is returned to the parent pointer?
- Q2: If you want to delete a node with two children, could you use the largest value in the left subtree instead of the smallest in the right?
- Q3: Does deleting a node ever increase the height of a BST?

Key Takeaways

- Deletion requires restructuring the tree to maintain the BST property.
- Case 1 (Leaf): Simply remove.
- Case 2 (One Child): Bypass the node.
- Case 3 (Two Children): Replace with the In-order Successor, then delete the original Successor node.
- The recursive implementation beautifully handles pointer reassignment.

Next Lecture Preview

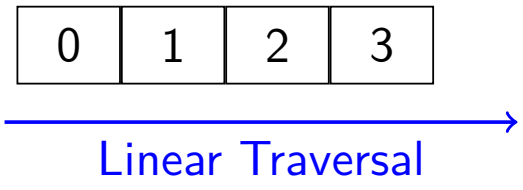
- In Lecture 29, we step away from modifying the tree and look at how to read data out of it systematically using Tree Traversals (Inorder, Preorder, Postorder).

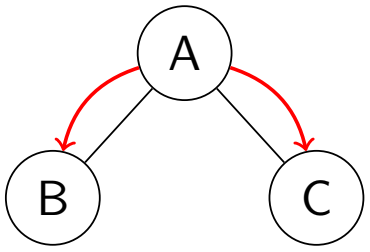
Questions?

Any Questions?

What is Tree Traversal?

- Traversal is the process of visiting (e.g., printing or processing) every node in a tree exactly once.
- In a linear structure (like an array), traversal is obvious: start at index 0 and go to the end.
- In a non-linear tree, there are many different paths we can take to visit all the nodes.
- The two main categories are Depth-First Search (DFS) and Breadth-First Search (BFS).

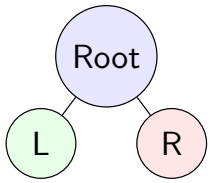




Non-linear Traversal

DFS: The Three Methods

- Depth-First Search goes as deep as possible down one path before backing up.
- There are three standard DFS traversals. Their names indicate when we visit the ROOT node relative to its subtrees:
  1. **Pre-order**: Visit Root before children.
  2. **In-order**: Visit Root in-between children.
  3. **Post-order**: Visit Root after children.

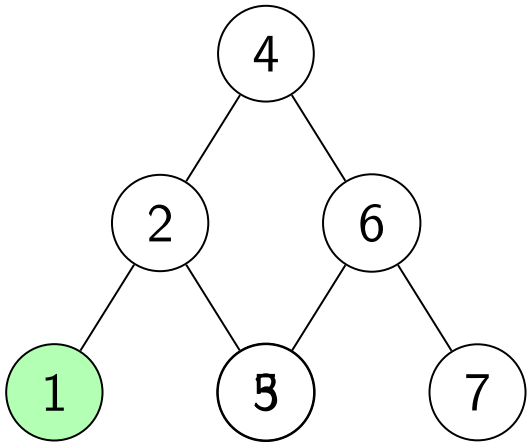


Pre: Root → L → R (NLR)
In: L → Root → R (LNR)
Post: L → R → Root (LRN)

Inorder Traversal (Left, Root, Right)

Algorithm (Recursive):

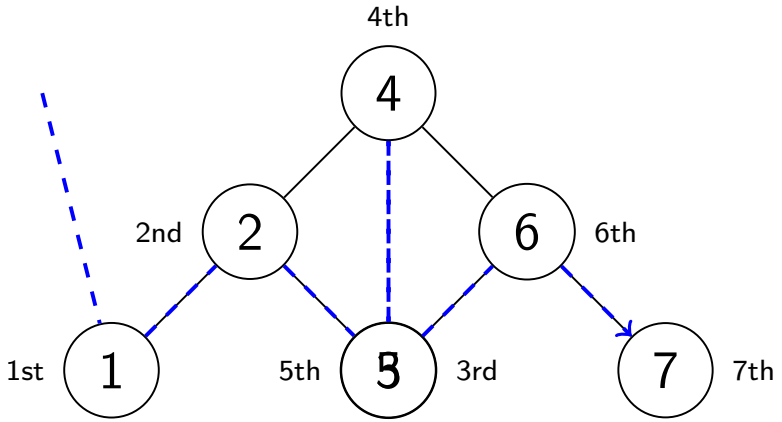
1. Recursively traverse the left subtree.
2. Visit the root node.
3. Recursively traverse the right subtree.



1st

Tracing Inorder Traversal

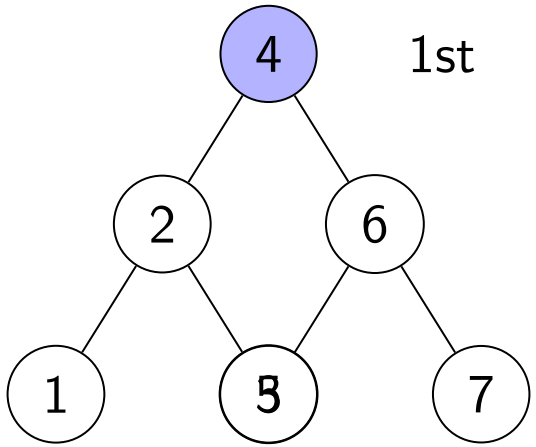
- Tree: Root(4), L(2), R(6). 2's children(1,3). 6's children(5,7).
- **Trace**:
  - Go left from 4 to 2, left from 2 to 1. **Print 1**.
  - Back to 2. **Print 2**. Go right to 3. **Print 3**.
  - Back to 4. **Print 4**. Go right to 6...
- **Output**: 1, 2, 3, 4, 5, 6, 7



Preorder Traversal (Root, Left, Right)

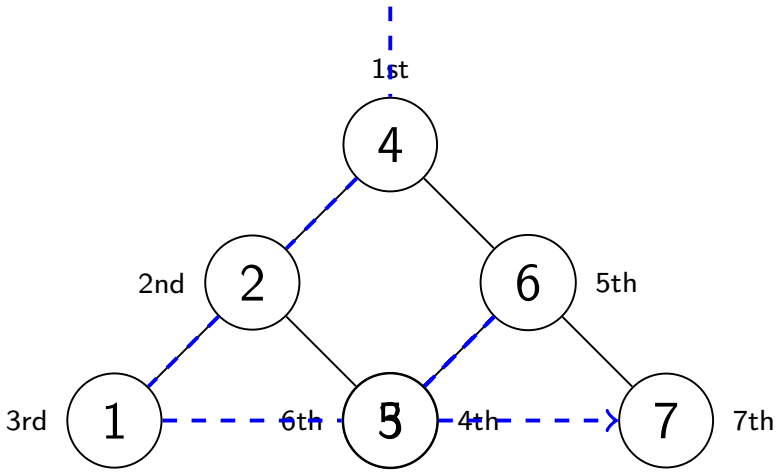
Algorithm (Recursive):

1. Visit the root node.
2. Recursively traverse the left subtree.
3. Recursively traverse the right subtree.



Tracing Preorder Traversal

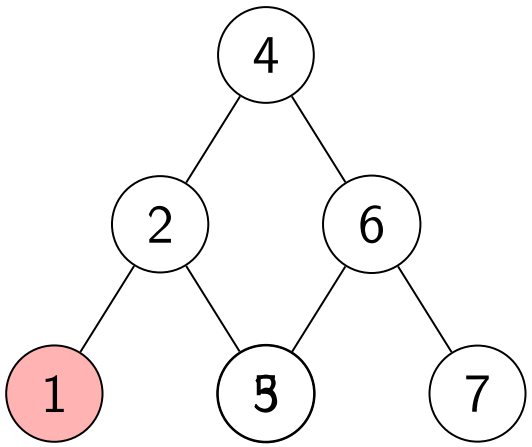
- Tree: Root(4), L(2), R(6). 2's children(1,3). 6's children(5,7).
- **Trace**:
  - **Print root 4**. Go left to 2.
  - **Print root 2**. Go left to 1. **Print root 1**.
  - Go right from 2 to 3. **Print 3**...
- **Output**: 4, 2, 1, 3, 6, 5, 7



Postorder Traversal (Left, Right, Root)

Algorithm (Recursive):

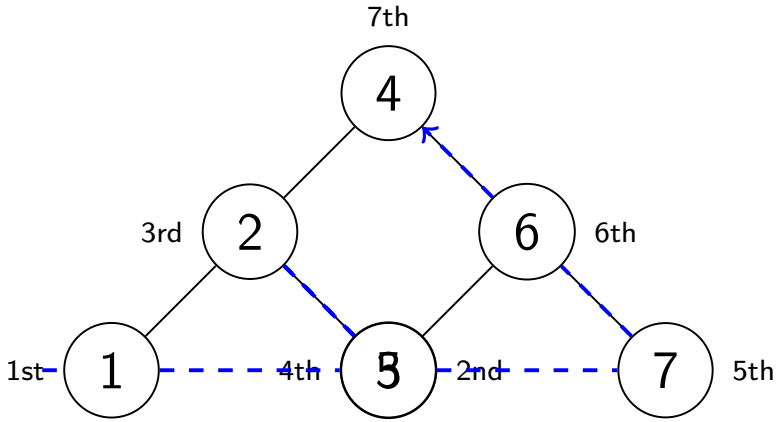
1. Recursively traverse the left subtree.
2. Recursively traverse the right subtree.
3. Visit the root node.



1st

Tracing Postorder Traversal

- Tree: Root(4), L(2), R(6). 2's children(1,3). 6's children(5,7).
- **Trace**:
  - Go down to left leaf 1. **Print 1**.
  - Go to right sibling 3. **Print 3**.
  - Go to parent 2. **Print 2**...
- **Output**: 1, 3, 2, 5, 7, 6, 4



Python Implementation (Inorder)

```
def inorder(root):
    if root is not None:
        inorder(root.left)
        print(root.data, end=" ")
        inorder(root.right)
```

Python Implementation (Preorder)

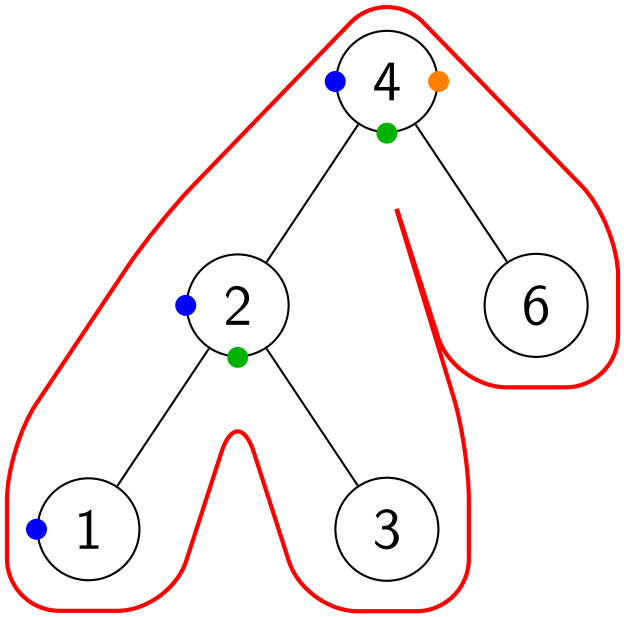
```
def preorder(root):
    if root is not None:
        print(root.data, end=" ") # Visit Root first
        preorder(root.left)
        preorder(root.right)
```

Python Implementation (Postorder)

```
def postorder(root):
    if root is not None:
        postorder(root.left)
        postorder(root.right)
        print(root.data, end=" ") # Visit Root last
```

Trick: The 'Outline' Method

- To quickly find the traversal by hand, draw an outline around the tree.
- **Preorder**: Note a node when you pass to its LEFT.
- **Inorder**: Note a node when you pass underneath it (BOTTOM).
- **Postorder**: Note a node when you pass its RIGHT side moving up.



Time and Space Complexity

- **Time Complexity**: $O(N)$ for all three traversals.
- **Why?** We must visit every single node exactly once to print it.
- **Space Complexity**: $O(h)$ where h is the height of the tree, due to the recursion call stack.
- Worst case space (skewed tree) is $O(N)$. Best case space (balanced tree) is $O(\log N)$.

Traversal	Time	Space (Worst Case)
Preorder	$O(N)$	$O(N)$
Inorder	$O(N)$	$O(N)$
Postorder	$O(N)$	$O(N)$

Test Your Understanding

Consider a tree with Root A, Left child B, Right child C.

1. **Q1**: What is the Preorder traversal?
2. **Q2**: What is the Inorder traversal?
3. **Q3**: What is the Postorder traversal?
4. **Q4**: Which traversal is best to flatten a BST into a sorted list?

Answers (Think before checking!)

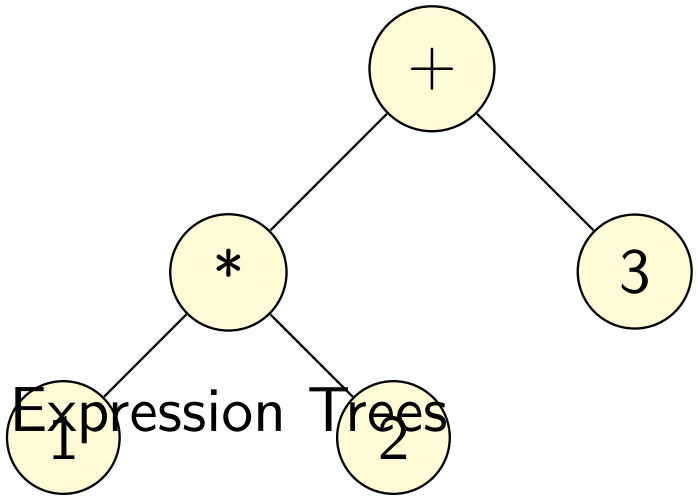
1. A, B, C
2. B, A, C
3. B, C, A
4. Inorder

Key Takeaways

- Traversals visit every node in a tree methodically.
- The names Pre, In, and Post refer to when the ROOT is visited relative to its children.
- Inorder traversal of a BST yields a sorted sequence.
- All DFS traversals run in $O(N)$ time and are elegantly implemented using recursion.

Next Lecture Preview

- In our final lecture on Trees, Lecture 30, we will look at real-world applications of binary trees and do a full recap of Unit 5 to prepare for the exam.



Questions?

Lecture 5.6: Tree Applications & Course Recap

Complete Lecture Deck – All 30 Lectures

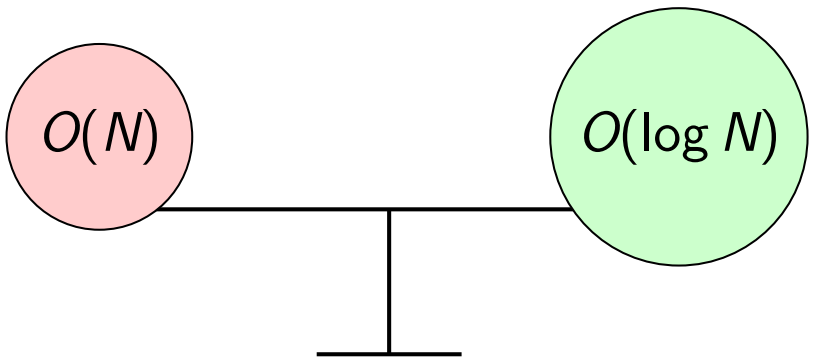
Milav Dabgar

Lecturer, GP Palanpur

July 22, 2026

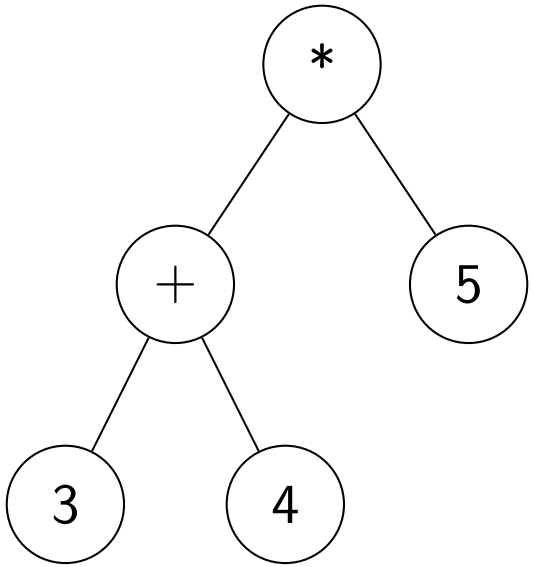
Why Learn Trees?

- Arrays and Linked Lists are linear. They force a sequential scan for searching (unless sorted + array).
- Trees provide a natural way to represent hierarchical data.
- They offer highly efficient ($O(\log N)$) search, insertion, and deletion capabilities.
- They are the foundational structure behind many complex algorithms and systems.



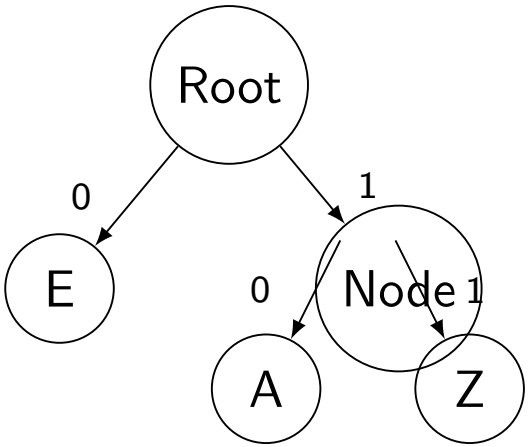
Application 1: Expression Trees

- Compilers use Binary Trees to parse and evaluate mathematical expressions.
- Leaves are operands (numbers or variables).
- Internal nodes are operators (+, -, *, /).
- Example: (3 + 4) * 5



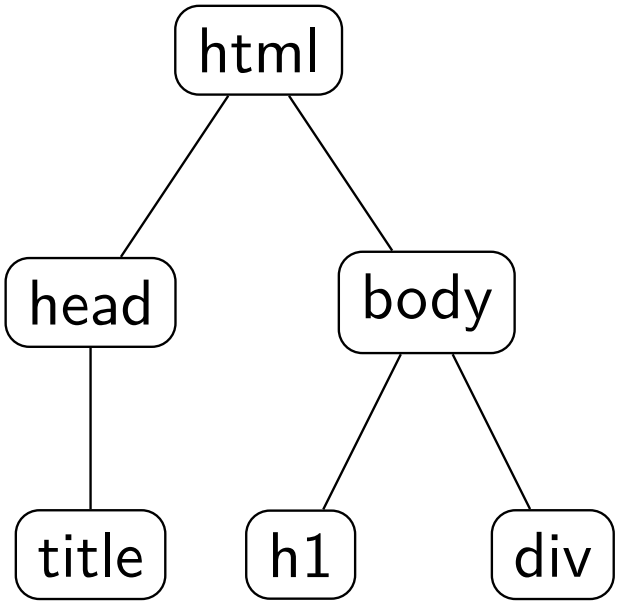
Application 2: Huffman Coding Tree

- Used for data compression (like in ZIP files or JPEGs).
- It builds a binary tree based on the frequency of characters in a file.
- More frequent characters are placed closer to the root.
- Less frequent characters are placed deeper as leaves.
- Left branch = 0, Right branch = 1.



Application 3: File Systems & DOM

- File Systems: Your OS uses a tree to manage directories and files.
- HTML/DOM: Web browsers parse HTML into a Document Object Model (DOM) tree. The root is `<html>`, branches are `<body>`, `<div>`, etc.
- Network Routing: Routers use tree-like structures (spanning trees) to find paths and prevent loops.

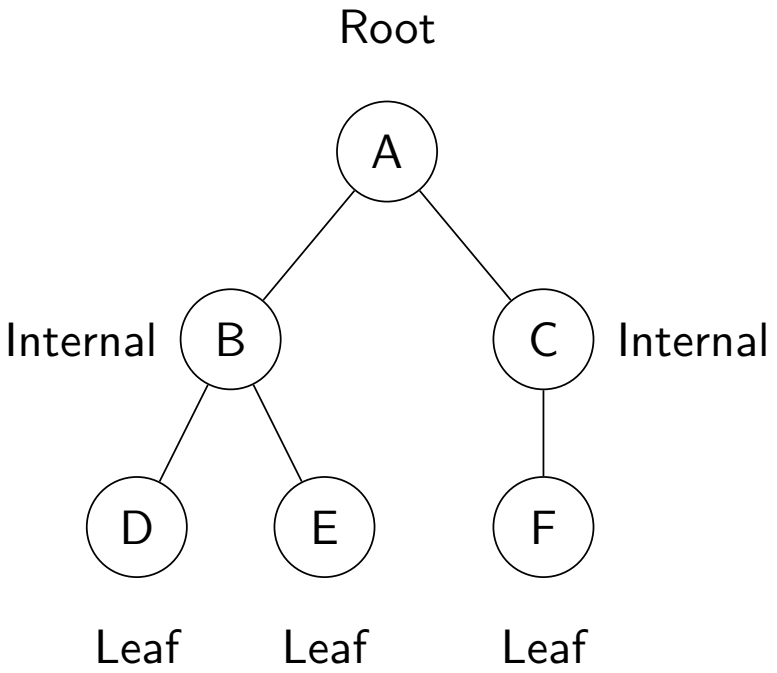


Application 4: Databases and Searching

- Binary Search Trees are the foundation, but real databases use advanced variants.
- B-Trees and B+ Trees: Used in SQL databases to index data on hard drives efficiently.
- Trie (Prefix Tree): Used in search engines for auto-complete and spell-checking.
- Heaps (Complete Binary Trees): Used in priority queues and the HeapSort algorithm.

Unit 5 Recap: Terminology

- Root: Top node.
- Leaf: Node with no children.
- Edge: Connection between nodes.
- Depth/Level: Distance from root.
- Binary Tree: Max 2 children per node.
- Complete vs Full Binary Trees.



Unit 5 Recap: BST Property

- The Golden Rule of BST:
- Left Subtree values $<$ Root value.
- Right Subtree values $>$ Root value.
- Search/Insert/Delete rely on this to achieve $O(\log N)$ average time.
- Worst case is $O(N)$ if the tree is skewed.

Unit 5 Recap: Operations

- Insertion: Always adds a new node as a leaf.
- Deletion Case 1: Delete leaf (easy).
- Deletion Case 2: Delete node with 1 child (bypass).
- Deletion Case 3: Delete node with 2 children (replace with inorder successor, then delete successor).

Unit 5 Recap: Traversals

- Inorder: Left $\rightarrow$ Root $\rightarrow$ Right (Yields sorted data).
- Preorder: Root $\rightarrow$ Left $\rightarrow$ Right (Useful for copying trees).
- Postorder: Left $\rightarrow$ Right $\rightarrow$ Root (Useful for deleting trees).
- Outline Method trick: Left dot (Pre), Bottom dot (In), Right dot (Post).

Exam Preparation Tips

- Practice drawing trees! Don't just read the code.
- Given a list of numbers, practice inserting them into an empty BST.
- Given a tree, practice writing out the Inorder, Preorder, and Postorder sequences.
- Understand the difference between a general binary tree and a Binary Search Tree.

Q & A

- Any questions on Unit 5: Trees?
- - Terminology?
- - BST Properties?
- - Insertion / Deletion?
- - Traversals?

Key Takeaways

- Trees are versatile structures used in compilers, file systems, compression, and databases.
- A strong grasp of BST properties is essential for understanding how fast lookups work.
- Traversals allow us to extract data from trees in specific, useful sequences.
- You are now equipped with the knowledge of hierarchical data structures!

Next Lecture Preview

End of Course!