**A data structure** is a systematic way of organizing, storing, and managing data in a computer so that it can be accessed and modified efficiently. Data structures are fundamental to computer science and software engineering because they enable the effective handling of data for various applications.

**### Key Characteristics of Data Structures**

1. Organization: Data structures determine how data is organized and structured in memory, which influences how easily it can be manipulated.

2. Operations: Different data structures provide different operations to manipulate the data, such as adding, deleting, searching, and updating elements.

3. Efficiency: The choice of data structure affects the efficiency of algorithms, impacting performance in terms of time complexity (how fast operations can be performed) and space complexity (how much memory is used).

4. Abstract Data Types: Data structures often implement abstract data types (ADTs), which define a data model and the operations that can be performed on that model, independent of how it is implemented.

**### Types of Data Structures**

Data structures can be broadly classified into two categories:

1. Primitive Data Structures: These are the basic data types that are directly supported by programming languages. Examples include:

- Integers

- Floats

- Characters

- Booleans

2. Non-Primitive Data Structures: These are more complex data structures built from primitive data types. They can be further classified into:

- Linear Data Structures: Elements are arranged sequentially. Examples include:

- Arrays: Fixed-size collections of elements of the same type.

- Linked Lists: Collections of nodes where each node contains data and a reference to the next node.

- Stacks: LIFO (Last In, First Out) structures for storing data.

- Queues: FIFO (First In, First Out) structures for storing data.

- Non-Linear Data Structures: Elements are arranged hierarchically or in a graph. Examples include:

- Trees: Hierarchical structures with nodes connected in a parent-child relationship (e.g., binary trees, AVL trees).

- Graphs: Collections of nodes connected by edges, allowing complex relationships (e.g., social networks, web page links).

**### Importance of Data Structures**

- Efficiency: Choosing the right data structure can significantly improve the efficiency of algorithms in terms of speed and memory usage.

- Organization: Data structures help in organizing data in a way that makes it easier to access and modify.

- Problem Solving: Different data structures are suited for different types of problems, enabling developers to create more effective solutions.

- Code Clarity: Using appropriate data structures can make code more understandable and maintainable.

The distinction between linear and non-linear data structures is fundamental in computer science and programming. Here’s a comparison highlighting their key differences:

**### Linear Data Structures**

In linear data structures, data elements are arranged sequentially or linearly. Each element is connected to its previous and next element, forming a single level of data organization. The relationships between the elements are one-dimensional.

**#### Characteristics:**

- Sequential Arrangement: Elements are arranged in a sequence, meaning each element has a unique predecessor and successor, except for the first and last elements.

- Single Level: The organization of data is one-dimensional.

- Memory Allocation: Memory is allocated in a contiguous manner.

- Access: Elements can be accessed in a sequential manner, typically using loops.

- Examples:

- Arrays

- Linked Lists

- Stacks

- Queues

**#### Example: Array**

```python

# Example of a linear data structure - Array

arr = [1, 2, 3, 4, 5]

# Accessing elements

print(arr[0]) # Output: 1

**### Non-Linear Data Structures**

In non-linear data structures, data elements are not arranged sequentially. Instead, they can be connected in multiple levels, forming a hierarchy or a network. The relationships between the elements can be more complex than in linear structures.

**#### Characteristics:**

- Hierarchical Arrangement: Data elements can have multiple levels and relationships, allowing for complex connections between them.

- Multiple Levels: The organization of data can be two-dimensional or multi-dimensional.

- Memory Allocation: Memory is often allocated in a non-contiguous manner.

- Access: Accessing elements can be more complex and may require traversal techniques such as depth-first or breadth-first search.

- Examples:

- Trees (e.g., binary trees, AVL trees)

- Graphs

- Heaps

#### Example: Tree

```python

# Example of a non-linear data structure - Binary Tree

class Node:

def \_\_init\_\_(self, value):

self.value = value

self.left = None

self.right = None

# Creating nodes

root = Node(1)

root.left = Node(2)

root.right = Node(3)

# Accessing nodes

print(root.value) # Output: 1 (Root node)

print(root.left.value) # Output: 2 (Left child)

print(root.right.value) # Output: 3 (Right child)

In **Python, an object is a core construct** that represents data (state) and behavior (methods). Objects are instances of classes, and almost everything in Python is an object, including integers, strings, functions, and even classes themselves. Each object has a specific type, which defines the kind of data it can hold and what operations can be performed on it.

### Key Characteristics of Python Objects

1. Identity: A unique identifier for each object, which can be obtained using `id()`.

2. Type: The type of the object determines its properties and behaviors, which can be checked using `type()`.

3. Value: The actual data held by the object, which can vary depending on the object’s type.

**### Types of Objects in Python**

Python supports various types of objects, which can be broadly categorized as follows:

**#### 1. Basic (Primitive) Data Types**

- Integers (`int`): Whole numbers, e.g., `42`, `-5`.

- Floating Point Numbers (`float`): Decimal numbers, e.g., `3.14`, `-0.001`.

- Complex Numbers (`complex`): Numbers with real and imaginary parts, e.g., `2 + 3j`.

- Strings (`str`): Sequence of characters, e.g., `"hello"`, `"Python"`.

- Booleans (`bool`): Represents truth values, `True` and `False`.

**#### 2. Collection Types**

- Lists (`list`): Ordered, mutable sequence of elements, e.g., `[1, 2, 3]`.

- Tuples (`tuple`): Ordered, immutable sequence of elements, e.g., `(1, 2, 3)`.

- Sets (`set`): Unordered, mutable collection of unique elements, e.g., `{1, 2, 3}`.

- Dictionaries (`dict`): Key-value pairs, mutable and unordered, e.g., `{"a": 1, "b": 2}`.

**#### 3. User-Defined Types**

- Classes and Instances: When you define a class using the `class` keyword, you create a new data type. Instances of this class are objects.

```python

class MyClass:

def \_\_init\_\_(self, data):

self.data = data

obj = MyClass(10) # obj is an object of MyClass

**#### 4. Callable Objects**

- Functions: Defined using `def` or `lambda`, they can be called with arguments, e.g., `def my\_function():`.

- Methods: Functions that are defined within a class and are associated with class instances.

- Lambdas: Anonymous functions defined with the `lambda` keyword, e.g., `lambda x: x + 1`.

**#### 5. Special Types**

- NoneType (`None`): Represents the absence of a value, with a single instance `None`.

- Ellipsis (`...`): A built-in object represented by `...`, often used as a placeholder.

**#### 6. File Objects**

- File handling is done using `open()` which returns a file object to read, write, or manipulate files.

```python

f = open("example.txt", "r") # f is a file object

**#### 7. Modules and Packages**

- Module Objects: Modules in Python (imported using `import` statements) are objects themselves.

- Packages: A collection of modules, structured as directories, which allows for organizing code.

**#### 8. Iterators and Generators**

- Iterators: Objects that implement the iterator protocol (`\_\_iter\_\_` and `\_\_next\_\_` methods), allowing for iteration over collections.

- Generators: Special iterators created with `yield` statements, used for generating sequences lazily.

**#### 9. Built-in Function and Method Objects**

- Functions and methods in Python’s standard library (e.g., `len`, `sum`, etc.) are objects that can be passed around and used like other objects.

**#### 10. Type Objects**

- The `type` itself is an object in Python, and classes are instances of `type`. Every class created with `class` is a `type` object.

**### Example of Object Creation and Types**

```python

**# Integer object**

num = 10

print(type(num)) # Output: <class 'int'>

**# List object**

my\_list = [1, 2, 3]

print(type(my\_list)) # Output: <class 'list'>

**# Function object**

def my\_func():

return "Hello"

print(type(my\_func)) # Output: <class 'function'>

**# User-defined class object**

class Person:

def \_\_init\_\_(self, name):

self.name = name

person = Person("Alice")

print(type(person)) # Output: <class '\_\_main\_\_.Person'>

**### Special Note: Everything in Python is an Object**

In Python, every entity, from numbers to functions, is treated as an object. This makes Python highly flexible, as objects can be passed as arguments, assigned to variables, and used in various ways that enhance modularity and reusability.

**In Python, class objects and instance objects are two key concepts that help organize data and behavior in object-oriented programming.**

### 1. Class Object

A class object is an object created when a class itself is defined. It serves as a blueprint for creating instances of that class. The class object contains all the class-level attributes and methods, which are shared among all instances created from this class.

In Python, classes are first-class objects, which means that they can be assigned to variables, passed to functions, and manipulated just like any other object.

**### Example of a Class Object**

```python

class Car:

wheels = 4 # Class attribute, shared among all instances

def start\_engine(self):

print("Engine started")

In the above example:

- `Car` is a class object. You can refer to it directly to access class-level attributes like `wheels`.

- You can also call methods on the class object itself, though usually, they're called through instances.

```python

# Accessing a class attribute directly through the class object

print(Car.wheels) # Output: 4

### 2. Instance Object

An instance object is created by calling the class object, which allocates memory and sets up the instance with initial values as defined in the `\_\_init\_\_` method (if provided). Each instance object represents a unique, individual object of that class, with its own separate state.

**### Example of an Instance Object**

```python

# Creating an instance (object) of the Car class

my\_car = Car() # my\_car is an instance object

# Accessing an instance method

my\_car.start\_engine() # Output: Engine started

# Accessing a class attribute through an instance (common practice)

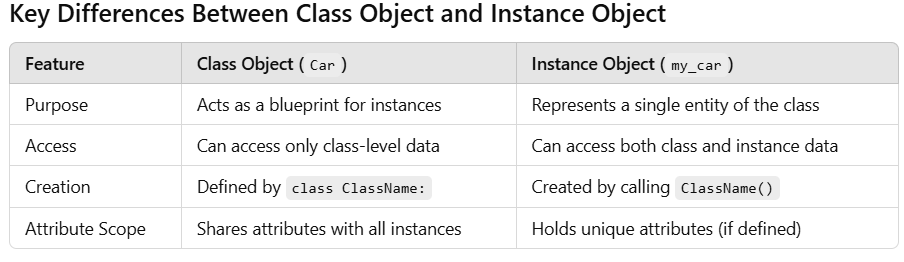
print(my\_car.wheels) # Output: 4

In this example:

- `my\_car` is an instance object of the `Car` class. It has access to the methods and attributes defined in `Car`.

- Calling `my\_car.start\_engine()` triggers the method in the context of this specific instance, although in this example it doesn't depend on instance-specific data.

### Key Differences Between Class Object and Instance Object



### Additional Example: Class vs. Instance Attributes

In this example, each instance can have its own unique `color`, but the `wheels` attribute is shared:

```python

class Car:

wheels = 4 # Class attribute

def \_\_init\_\_(self, color):

self.color = color # Instance attribute

# Creating instances of Car

car1 = Car("Red")

car2 = Car("Blue")

# Accessing attributes

print(f"Car1: Color={car1.color}, Wheels={car1.wheels}") # Car1: Color=Red, Wheels=4

print(f"Car2: Color={car2.color}, Wheels={car2.wheels}") # Car2: Color=Blue, Wheels=4

Here:

- `wheels` is a class attribute shared by all instances.

- `color` is an instance attribute, unique to each instance (`car1` has color "Red" and `car2` has color "Blue").

**singly linked list**

**A singly linked list is a type of data structure that consists of a sequence of nodes, where each node contains two main parts:**

1. Data: The value or information the node holds.

2. Next Pointer: A reference (or link) to the next node in the sequence.

In a singly linked list, each node points to the next node in the list, creating a chain-like structure. However, nodes only link forward, from one node to the next, making it "singly" linked. The list has a starting node (commonly referred to as `head` or `start`), which is the entry point for accessing all elements in the list. The last node points to `None`, indicating the end of the list.

### Characteristics of a Singly Linked List

- Unidirectional: Traversal is only possible in one direction (from the start node to the end).

- Dynamic Size: Unlike arrays, linked lists don’t need a predefined size and can grow or shrink as elements are added or removed.

- Efficient Insertions/Deletions: Adding or removing elements can be efficient, especially at the beginning or end of the list, as it involves adjusting pointers rather than shifting elements (as with arrays).

### Visual Representation

Here’s a simple illustration of a singly linked list with four nodes:

```plaintext

head

|

↓

+-------+ +-------+ +-------+ +-------+

| 1 | ---> | 2 | ---> | 3 | ---> | 4 | ---> None

+-------+ +-------+ +-------+ +-------+

**### Basic Operations on a Singly Linked List**

1. Insertion: Adding a node at the beginning, end, or after a specific node.

2. Deletion: Removing a node from the beginning, end, or a specific location.

3. Traversal: Visiting each node to access or print its data.

4. Searching: Finding if a node with a particular value exists in the list.

**### Advantages and Disadvantages**

#### Advantages

- Dynamic memory allocation: They do not require a fixed size, allowing the list to grow or shrink in size.

- Efficient insertions/deletions: Especially at the start or end, since there's no need for element shifting.

#### Disadvantages

- Sequential access: Accessing an element by index requires traversing the list, which is slower than accessing an element in an array.

- Extra memory: Each node requires additional memory to store the pointer to the next node.

A singly linked list is useful in scenarios where you need a flexible and efficient way to manage a collection of elements, especially when frequent insertions and deletions are required.

**In a linked list, memory allocation is handled differently than in arrays due to the dynamic nature of linked list structures. Here’s how it works:**

**### 1. Dynamic Memory Allocation**

- In linked lists, each node is created individually in memory as needed, rather than allocating a continuous block as with arrays.

- When a new node is added to a linked list, memory is allocated dynamically, meaning memory space is reserved for that node at runtime (often through languages with dynamic memory allocation like C/C++ using `malloc()` or `new`).

- The address of this newly allocated memory block is stored in the previous node's `next` pointer, linking the nodes together.

**### 2. Structure of a Node in Memory**

Each node in a linked list consists of:

- Data: The value or information held by the node (e.g., an integer, string, or complex object).

- Pointer/Reference to Next Node: A pointer to the memory address of the next node, forming a chain-like connection.

Each node is stored independently in memory, and nodes don’t have to be stored contiguously. They can be scattered across different memory locations.

**### 3. Allocation Process in a Singly Linked List**

- Creating a Node: When you create a new node (e.g., in Python `node = Node(data)`), memory is allocated for that node. This memory includes space for both the data and the pointer to the next node.

- Linking Nodes: The address of the newly created node is stored in the `next` pointer of the previous node, linking them sequentially. This allows traversal through the nodes in the list.

- End of List: The last node’s `next` pointer is set to `None` (or `NULL` in languages like C/C++), indicating the end of the list.

**### 4. Example of Node Creation in Memory (in Python)**

Here's how memory allocation would look in Python:

```python

class Node:

def \_\_init\_\_(self, data):

self.data = data # memory allocated for data

self.next = None # memory allocated for the pointer

When nodes are created, they each occupy their own space in memory, and Python (or the language being used) handles the memory address assignment. The linked list's `start` pointer or `head` points to the first node’s memory address, allowing traversal from node to node via the `next` pointers.

### 5. Memory Deallocation

- Automatic (in languages with garbage collection like Python or Java): When nodes are deleted, and there are no more references to them, the garbage collector deallocates their memory automatically.

- Manual (in languages without garbage collection like C): You need to free the memory explicitly (e.g., `free()` in C) to avoid memory leaks.

### Illustration of Memory Allocation in a Linked List

Suppose we have three nodes with values `5`, `10`, and `15`.

```plaintext

start (or head)

|

↓

+-----------+ +-----------+ +-----------+

| Data: 5 | ---> | Data: 10 | ---> | Data: 15 | ---> None

| Next: 100 | | Next: 200 | | Next: None|

+-----------+ +-----------+ +-----------+

Memory Address Memory Address Memory Address

100 200 300

1. Each node has a `data` field and a `next` field pointing to the address of the next node.

2. Nodes are linked through their memory addresses rather than being stored sequentially, allowing for efficient insertion and deletion without shifting elements, as you would need in an array.

**### Summary**

- Memory allocation for each node in a linked list is non-contiguous and handled dynamically.

- Pointers are used to connect nodes in memory, so the linked list can grow or shrink as needed.

- In languages with garbage collection, memory is automatically reclaimed, but in low-level languages, it must be manually managed.

**QUESTIONS**

1. Define a class Node to describe a node of a singly linked list. 2. Define a class SLL to implement Singly Linked List with \_init() method to create and initialise start reference variable. 3. Define a method is\_empty() to check if the linked list is empty in SLL class. 4. In class SLL, define a method insert\_at\_start() to insert an element at the starting of the list. 5. In class SLL, define a method insert\_at\_last() to insert an element at the end of the list. 6. In class SLL, define a method search() to find the node with specified element value. 7. In class SLL, define a method insert\_after() to insert a new node after a given node of the list. 8. In class SLL, define a method to print all the elements of the list. 9. In class SLL, implement iterator for SLL to access all the elements of the list in a sequence. 10. In class SLL, define a method delete\_first() to delete first element from the list. 11. In class SLL, define a method delete last() to delete last element from the list.

Here's a Python implementation of a singly linked list with the required functionalities:

```python

class Node:

"""Represents a node in a singly linked list."""

def \_\_init\_\_(self, data):

self.data = data

self.next = None

class SLL:

"""Represents a singly linked list."""

def \_\_init\_\_(self):

self.start = None

def is\_empty(self):

"""Check if the linked list is empty."""

return self.start is None

def insert\_at\_start(self, data):

"""Insert an element at the beginning of the list."""

new\_node = Node(data)

new\_node.next = self.start

self.start = new\_node

def insert\_at\_last(self, data):

"""Insert an element at the end of the list."""

new\_node = Node(data)

if self.is\_empty():

self.start = new\_node

else:

current = self.start

while current.next:

current = current.next

current.next = new\_node

def search(self, data):

"""Search for a node with the specified data."""

current = self.start

while current:

if current.data == data:

return current

current = current.next

return None

def insert\_after(self, prev\_data, data):

"""Insert a new node after the node with the given previous data."""

current = self.search(prev\_data)

if current is None:

print(f"Node with data {prev\_data} not found.")

return

new\_node = Node(data)

new\_node.next = current.next

current.next = new\_node

def print\_list(self):

"""Print all the elements of the list."""

current = self.start

while current:

print(current.data, end=" -> ")

current = current.next

print("None")

def \_\_iter\_\_(self):

"""Iterator to access elements of the list in a sequence."""

current = self.start

while current:

yield current.data

current = current.next

def delete\_first(self):

"""Delete the first element from the list."""

if self.is\_empty():

print("List is empty. No element to delete.")

return

self.start = self.start.next

def delete\_last(self):

"""Delete the last element from the list."""

if self.is\_empty():

print("List is empty. No element to delete.")

return

if self.start.next is None:

self.start = None

else:

current = self.start

while current.next.next:

current = current.next

current.next = None

# Example usage:

sll = SLL()

sll.insert\_at\_start(10)

sll.insert\_at\_start(5)

sll.insert\_at\_last(20)

sll.insert\_at\_last(25)

sll.insert\_after(10, 15)

print("Linked List Elements:")

sll.print\_list()

print("After deleting the first element:")

sll.delete\_first()

sll.print\_list()

print("After deleting the last element:")

sll.delete\_last()

sll.print\_list()

print("Elements in the list using iterator:")

for data in sll:

print(data, end=" ")

**### Explanation:**

1. Node class: Defines a node in the singly linked list with `data` and `next` attributes.

2. SLL class: Contains the singly linked list methods, including initialization, insertion, deletion, and traversal.

3. is\_empty: Checks if the list is empty.

4. insert\_at\_start: Adds a node to the beginning of the list.

5. insert\_at\_last: Adds a node to the end of the list.

6. search: Finds a node with specified data.

7. insert\_after: Inserts a node after a specific node.

8. print\_list: Prints all elements in the list.

9. \_\_iter\_\_: Makes the list iterable.

10. delete\_first: Removes the first element.

11. delete\_last: Removes the last element.

**### Singly Linked List Diagram**

```plaintext

SLL (start)

|

↓

+-------+ +-------+ +-------+ +-------+ +-------+

| 5 | ---> | 10 | ---> | 15 | ---> | 20 | ---> | 25 | ---> None

+-------+ +-------+ +-------+ +-------+ +-------+

**### Explanation of the Diagram**

1. `start`: This is a reference in the `SLL` class that points to the first node of the list.

2. Nodes: Each box represents a node in the list. The boxes contain:

- Data: The value held in the node (e.g., 5, 10, 15, 20, 25).

- Next pointer: An arrow (`->`) to the next node. If it's the last node, it points to `None`, indicating the end of the list.

**### Example of Operations on the List**

- Insert at Start: Adding a new node with data `5` at the beginning makes `5` the first node, with `start` pointing to it.

- Insert at Last: Adding a new node with data `25` at the end attaches it to the last node (`20`), updating the `next` pointer of `20` to `25`.

- Insert After: Inserting a node with data `15` after the node with data `10` adjusts the pointers so that `10 -> 15 -> 20`.

- Delete First: Removes the node at the start by making `start` point to the next node (e.g., if `5` is deleted, `start` will point to `10`).

- Delete Last: Removes the last node by updating the `next` pointer of the second-last node to `None`.

This is how the linked list operations affect the structure visually. Each pointer adjustment changes the flow of data connections, allowing insertion, deletion, and traversal through the list.

**A singly linked list is a type of data structure that consists of a sequence of nodes, where each node contains two main parts:**

1. Data: The value or information the node holds.

2. Next Pointer: A reference (or link) to the next node in the sequence.

In a singly linked list, each node points to the next node in the list, creating a chain-like structure. However, nodes only link forward, from one node to the next, making it "singly" linked. The list has a starting node (commonly referred to as `head` or `start`), which is the entry point for accessing all elements in the list. The last node points to `None`, indicating the end of the list.

**### Characteristics of a Singly Linked List**

- Unidirectional: Traversal is only possible in one direction (from the start node to the end).

- Dynamic Size: Unlike arrays, linked lists don’t need a predefined size and can grow or shrink as elements are added or removed.

- Efficient Insertions/Deletions: Adding or removing elements can be efficient, especially at the beginning or end of the list, as it involves adjusting pointers rather than shifting elements (as with arrays).

**### Visual Representation**

Here’s a simple illustration of a singly linked list with four nodes:

```plaintext

head

|

↓

+-------+ +-------+ +-------+ +-------+

| 1 | ---> | 2 | ---> | 3 | ---> | 4 | ---> None

+-------+ +-------+ +-------+ +-------+

**### Basic Operations on a Singly Linked List**

1. Insertion: Adding a node at the beginning, end, or after a specific node.

2. Deletion: Removing a node from the beginning, end, or a specific location.

3. Traversal: Visiting each node to access or print its data.

4. Searching: Finding if a node with a particular value exists in the list.

**### Advantages and Disadvantages**

#### Advantages

- Dynamic memory allocation: They do not require a fixed size, allowing the list to grow or shrink in size.

- Efficient insertions/deletions: Especially at the start or end, since there's no need for element shifting.

#### Disadvantages

- Sequential access: Accessing an element by index requires traversing the list, which is slower than accessing an element in an array.

- Extra memory: Each node requires additional memory to store the pointer to the next node.

A singly linked list is useful in scenarios where you need a flexible and efficient way to manage a collection of elements, especially when frequent insertions and deletions are required.

**Traversing in a linked list means visiting each node in the list, one by one, in sequence. This is a fundamental operation, allowing access to each node's data, checking if specific data exists in the list, or performing an action at each node.**

**### How Traversal Works in a Singly Linked List**

In a singly linked list, traversal typically begins at the head node (or `start` node). By following the `next` pointer of each node, you can move from one node to the next until you reach the end of the list, where the `next` pointer is `None`.

**### Steps for Traversal**

1. Start at the head: Begin with a pointer/reference to the head node.

2. Visit each node: For each node, you can access the `data` stored in it and perform any necessary actions.

3. Move to the next node: Update the pointer/reference to the current node’s `next` pointer, moving to the next node in the sequence.

4. End at None: Stop when you reach a node where `next` is `None`, which indicates the end of the list.

**### Example Code for Traversal in Python**

Here’s a Python example that demonstrates traversing a singly linked list to print each node’s data.

```python

class Node:

def \_\_init\_\_(self, data):

self.data = data

self.next = None

class SinglyLinkedList:

def \_\_init\_\_(self):

self.head = None

def traverse(self):

# Start with the head node

current = self.head

while current is not None:

print(current.data, end=" -> ")

# Move to the next node

current = current.next

print("None") # End of list

# Creating and linking nodes in a linked list

linked\_list = SinglyLinkedList()

linked\_list.head = Node(5)

second\_node = Node(10)

third\_node = Node(15)

# Linking nodes

linked\_list.head.next = second\_node

second\_node.next = third\_node

# Traversing and printing the linked list

linked\_list.traverse()

```

### Output

```plaintext

5 -> 10 -> 15 -> None

**### Explanation of the Code**

1. Node Class: Each node contains `data` and a `next` pointer.

2. SinglyLinkedList Class: This manages the linked list and has a `traverse` method to visit each node.

3. Traverse Method:

- Starts with the `head` node.

- Prints the `data` of each node.

- Moves to the next node using the `next` pointer.

- Ends when it reaches `None`, printing `None` to indicate the end.

**### Applications of Traversing in Linked Lists**

- Search Operation: To find if a specific value exists in the list.

- Counting Nodes: To determine the number of elements.

- Updating Data: To change data in specific nodes based on certain conditions.

- Deleting or Inserting Nodes: Traversal helps locate the correct position for insertion or deletion.

**### Complexity of Traversing**

- Time Complexity: \(O(n)\), where \(n\) is the number of nodes in the list, because you need to visit each node once.

- Space Complexity: \(O(1)\), as only one extra pointer/reference is used, regardless of list size.

Traversal is an essential operation in managing and utilizing the elements in a linked list.

**QUESTIONS:**

**create a class employee with attributes,empid,name,salary also define methods to accesss propertiees if employee**

**```python**

**class Employee:**

**def \_\_init\_\_(self, empid, name, salary):**

**self.\_empid = empid # Private attribute**

**self.\_name = name # Private attribute**

**self.\_salary = salary # Private attribute**

**# Getter method for empid**

**def get\_empid(self):**

**return self.\_empid**

**# Setter method for empid**

**def set\_empid(self, empid):**

**self.\_empid = empid**

**# Getter method for name**

**def get\_name(self):**

**return self.\_name**

**# Setter method for name**

**def set\_name(self, name):**

**self.\_name = name**

**# Getter method for salary**

**def get\_salary(self):**

**return self.\_salary**

**# Setter method for salary**

**def set\_salary(self, salary):**

**if salary < 0:**

**print("Salary can't be negative")**

**else:**

**self.\_salary = salary**

**# Example usage**

**emp1 = Employee(101, "Alice", 50000)**

**# Accessing properties using getter methods**

**print("Employee ID:", emp1.get\_empid()) # Output: Employee ID: 101**

**print("Employee Name:", emp1.get\_name()) # Output: Employee Name: Alice**

**print("Employee Salary:", emp1.get\_salary()) # Output: Employee Salary: 50000**

**# Modifying properties using setter methods**

**emp1.set\_name("Alice Brown")**

**emp1.set\_salary(55000)**

**print("Updated Name:", emp1.get\_name()) # Output: Updated Name: Alice Brown**

**print("Updated Salary:", emp1.get\_salary()) # Output: Updated Salary: 55000**

**### Explanation**

**- Attributes: `\_empid`, `\_name`, and `\_salary` are defined as private attributes by convention (using a leading underscore).**

**- Getter and Setter Methods: Defined for each attribute to allow controlled access and modification of the properties.**

**- The `set\_salary` method checks if the salary is non-negative before updating it.**