**Unit 3: **

3.1 Abstract Data Types (ADTs)

An Abstract Data Type (ADT) is a data structure that is defined by its behavior (the operations that can be performed on it) rather than its implementation. ADTs allow programmers to focus on what operations can be performed, rather than how they are performed.

Key Characteristics:

Encapsulation: ADTs encapsulate the data and operations, hiding the implementation details from the user.
Interface: The ADT defines a clear interface, specifying the operations that can be performed without exposing the underlying implementation.
Examples of ADTs:

List ADT: Supports operations like adding, removing, and accessing elements.
Stack ADT: Follows a Last-In-First-Out (LIFO) principle with operations like push, pop, and peek.
Queue ADT: Follows a First-In-First-Out (FIFO) principle with operations like enqueue, dequeue, and front.
2. Implementing ADTs Using Classes
In Python, ADTs are often implemented using classes. A class provides the blueprint for creating objects (instances), encapsulating both data (attributes) and operations (methods).

Basic Structure of a Class:

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

    def some_method(self):
        # Method implementation
        Pass
```

Attributes: These are variables that hold data specific to each instance of the class.
Methods: These are functions defined within the class that operate on the data.

3.2 Inheritance

Inheritance is a core concept in Object-Oriented Programming (OOP) that allows a new class, known as a child class or derived class, to inherit properties and behaviors (attributes and methods) from an existing class, known as a parent class or base class.

Purpose and Benefits:

Code Reusability: Inheritance enables the reuse of existing code, reducing redundancy. The child class can use the methods and attributes of the parent class, avoiding the need to rewrite common functionality.
Extensibility: Inheritance allows the child class to extend or modify the behavior of the parent class by adding new methods or overriding existing ones.
Hierarchical Organization: It provides a way to create a hierarchical relationship between classes, promoting a logical organization of code.
Example:

python
Copy code

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

    def speak(self):
        return "Some sound"

class Dog(Animal):
    def speak(self):
        return "Woof!"

dog = Dog("Buddy")
print(dog.speak())  # Output: Woof!
```

In this example, Dog inherits from Animal and overrides the speak method to provide a specific implementation.

3.3 Encapsulation

Encapsulation is an Object-Oriented Programming (OOP) concept that involves bundling the data (attributes) and the methods (functions) that operate on the data into a single unit or class. This helps to organize and structure code logically.
Purpose and Benefits:

Data Protection: Encapsulation restricts direct access to some of an object's components, which can protect the object’s internal state from unintended interference and misuse.

Modularity: By encapsulating data and methods, encapsulation allows for modular code where each class or module can be developed and tested independently.
Maintainability: Changes to the internal implementation of a class do not affect the external code that interacts with the class, making the system easier to maintain.

Example:

```
class Car:
    def __init__(self, model, speed):
        self.model = model
        self.__speed = speed  # Private attribute

    def set_speed(self, speed):
        self.__speed = speed

    def get_speed(self):
        return self.__speed
```

In this example, __speed is a private attribute, and it is accessed and modified through the get_speed and set_speed methods, demonstrating encapsulation.

3.4 Information Hiding

Information Hiding is a principle closely related to encapsulation that involves hiding the internal details and implementation of a class or module from the outside world. Only the necessary information or operations are exposed through a public interface.

Purpose and Benefits:

Reduces Complexity: By hiding the internal workings of a class, information hiding reduces the complexity for users of the class, who only need to understand the public interface.

Security and Integrity: It helps to protect the integrity of the data by preventing external code from directly accessing or modifying it in unintended ways.

Flexibility: The internal implementation of a class can be changed without affecting external code, as long as the public interface remains consistent.

Example:

```
class BankAccount:

    def __init__(self, balance):

        self.__balance = balance  # Private attribute
```

```
def deposit(self, amount):
    self.__balance += amount

def withdraw(self, amount):
    if amount <= self.__balance:
        self.__balance -= amount
    else:
        raise ValueError("Insufficient funds")

def get_balance(self):
    return self.__balance
```

In this example, the `__balance` attribute is hidden from direct access, and can only be accessed or modified through the provided methods (deposit, withdraw, get_balance), showcasing information hiding.

3.5 Sorting Algorithms and Hash Tables

Sorting algorithms are methods used to arrange elements in a list or array into a specific order, typically ascending or descending. Sorting is a fundamental operation in computer science that is widely used in various applications, such as searching, data analysis, and optimization.

Common Sorting Algorithms:

- Bubble Sort:

How it works: Repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process is repeated until the list is sorted.

Efficiency: Simple but inefficient for large datasets; has a time complexity of $O(n^2)$.

- Selection Sort:

How it works: Divides the list into a sorted and an unsorted part, repeatedly finding the minimum element from the unsorted part and moving it to the sorted part.

Efficiency: Also has a time complexity of $O(n^2)$, but performs fewer swaps than bubble sort.

- Merge Sort:

How it works: Uses a divide-and-conquer approach, recursively dividing the list into smaller sublists until each sublist contains a single element, and then merges the sublists to produce the sorted list.

Efficiency: More efficient with a time complexity of O(n log n), making it suitable for larger datasets.

- Quick Sort:

How it works: Another divide-and-conquer algorithm that selects a "pivot" element and partitions the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then sorted recursively.

Efficiency: Generally very fast with an average-case time complexity of O(n log n), but can degrade to O(n^2) in the worst case.

Applications:

Sorting is essential for optimizing the performance of other algorithms, like binary search, and is also used in data processing, analytics, and database indexing.

3.6 Hash Tables

A hash table is a data structure that provides a way to store and retrieve data using a key-value pair. It uses a hash function to compute an index (or hash code) into an array of buckets or slots, from which the desired value can be found.

How Hash Tables Work:

Hash Function: Converts a key into a hash code, which is an index in the array where the corresponding value is stored.

Collision Handling: Since multiple keys can produce the same hash code (a collision), hash tables use techniques like chaining (linking elements with the same hash code in a list) or open addressing (finding another empty slot) to handle collisions.

Efficiency:

Average Case: The average time complexity for search, insert, and delete operations in a hash table is O(1), making it extremely efficient for these operations.

Worst Case: In the worst case, when many collisions occur, the time complexity can degrade to O(n).

Applications:

Fast Lookup: Hash tables are used in various applications requiring fast data retrieval, such as implementing dictionaries, databases, caches, and sets.

Symbol Tables: In compilers and interpreters to store variable names and their associated values.

SILVER OAK UNIVERSITY
EDUCATION TO INNOVATION