

Data Structure with Python (4331601) - Winter 2023 Solution

Milav Dabgar

January 11, 2024

Question 1(a) [3 marks]

Define best case, worst case and average case for time complexity.

Solution

Answer:

Table 1. Time Complexity Cases

Case Type	Definition	Example
Best Case	Minimum time needed for algorithm execution	Linear search finds element at first position
Worst Case	Maximum time needed for algorithm execution	Linear search finds element at last position
Average Case	Expected time for typical input scenarios	Linear search finds element in middle

- **Best Case:** Algorithm performs optimally with ideal input conditions
- **Worst Case:** Algorithm takes maximum possible time with unfavorable input
- **Average Case:** Mathematical expectation of execution time across all possible inputs

Mnemonic

BWA - Best, Worst, Average

Question 1(b) [4 marks]

What is Class and Object in OOP? Give suitable example.

Solution

Answer:

Table 2. Class vs Object

Aspect	Class	Object
Definition	Blueprint/template for creating objects	Instance of a class
Memory	No memory allocated	Memory allocated when created
Example	Car (template)	my_car = Car()

Listing 1. Class and Object Example

```
1 # Class definition
2 class Student:
```

```

3     def __init__(self, name, age):
4         self.name = name
5         self.age = age
6
7     def display(self):
8         print(f"Name: {self.name}, Age: {self.age}")
9
10    # Object creation
11    student1 = Student("John", 20)
12    student1.display()

```

- **Class:** Template defining attributes and methods
- **Object:** Real instance with actual values

Mnemonic

Class = Cookie Cutter, Object = Actual Cookie

Question 1(c) [7 marks]

Write a program for two matrix multiplication using simple nested loop and numpy module.

Solution

Answer:

Listing 2. Matrix Multiplication

```

1  # Method 1: Using Simple Nested Loop
2  def matrix_multiply_nested(A, B):
3      rows_A, cols_A = len(A), len(A[0])
4      rows_B, cols_B = len(B), len(B[0])
5
6      # Initialize result matrix
7      result = [[0 for _ in range(cols_B)] for _ in range(rows_A)]
8
9      # Matrix multiplication
10     for i in range(rows_A):
11         for j in range(cols_B):
12             for k in range(cols_A):
13                 result[i][j] += A[i][k] * B[k][j]
14
15     return result
16
17 # Method 2: Using NumPy
18 import numpy as np
19
20 def matrix_multiply_numpy(A, B):
21     A_np = np.array(A)
22     B_np = np.array(B)
23     return np.dot(A_np, B_np)
24
25 # Example usage
26 A = [[1, 2], [3, 4]]
27 B = [[5, 6], [7, 8]]
28
29 print("Nested Loop Result:", matrix_multiply_nested(A, B))
30 print("NumPy Result:", matrix_multiply_numpy(A, B))

```

- **Nested Loop:** Three loops for row, column, and multiplication

- **NumPy:** Built-in dot() function for efficient multiplication

Mnemonic

Row \times Column = Result

Question 1(c OR) [7 marks]

Write a program to implement basic operations on arrays.

Solution

Answer:

Listing 3. Array Operations

```

1 import array
2
3 # Create array
4 arr = array.array('i', [1, 2, 3, 4, 5])
5
6 def array_operations():
7     print("Original array:", arr)
8
9     # Insert element
10    arr.insert(2, 10)
11    print("After insert(2, 10):", arr)
12
13    # Append element
14    arr.append(6)
15    print("After append(6):", arr)
16
17    # Remove element
18    arr.remove(10)
19    print("After remove(10):", arr)
20
21    # Pop element
22    popped = arr.pop()
23    print(f"Popped element: {popped}, Array: {arr}")
24
25    # Search element
26    index = arr.index(3)
27    print(f"Index of 3: {index}")
28
29    # Count occurrences
30    count = arr.count(2)
31    print(f"Count of 2: {count}")
32
33 array_operations()

```

Table 3. Array Operations

Operation	Method	Description
Insert	insert(index, value)	Add element at specific position
Append	append(value)	Add element at end
Remove	remove(value)	Remove first occurrence
Pop	pop()	Remove and return last element

Mnemonic

IARP - Insert, Append, Remove, Pop

Question 2(a) [3 marks]

Explain Big 'O' Notation.

Solution

Answer:

Table 4. Big O Complexity

Notation	Name	Example
$O(1)$	Constant	Array access
$O(n)$	Linear	Linear search
$O(n^2)$	Quadratic	Bubble sort
$O(\log n)$	Logarithmic	Binary search

- **Big O:** Describes upper bound of algorithm's time complexity
- **Purpose:** Compare efficiency of different algorithms
- **Focus:** Worst-case scenario analysis

Mnemonic

Big O = Big Order of growth

Question 2(b) [4 marks]

Differentiate between class method and static method.

Solution

Answer:

Table 5. Method Types Comparison

Aspect	Class Method	Static Method
Decorator	@classmethod	@staticmethod
First Parameter	cls (class reference)	No special parameter
Access	Can access class variables	Cannot access class/instance variables
Usage	Alternative constructors	Utility functions

Listing 4. Class vs Static Method

```

1 class MyClass:
2     class_var = "I am class variable"
3
4     @classmethod
5     def class_method(cls):
6         return f"Class method accessing: {cls.class_var}"
7
8     @staticmethod
9     def static_method():

```

```

10         return "Static method - no class access"
11
12 # Usage
13 print(MyClass.class_method())
14 print(MyClass.static_method())

```

Mnemonic

Class method has CLS, Static method is STandalone

Question 2(c) [7 marks]

Implement a class for single level inheritance using public and private type derivation.

Solution

Answer:

Listing 5. Single Level Inheritance

```

1 # Base class
2 class Vehicle:
3     def __init__(self, brand, model):
4         self.brand = brand          # Public attribute
5         self._model = model         # Protected attribute
6         self.__year = 2023         # Private attribute
7
8     def start_engine(self):
9         return f"{self.brand} engine started"
10
11     def _display_model(self):        # Protected method
12         return f"Model: {self._model}"
13
14     def __private_method(self):      # Private method
15         return f"Year: {self.__year}"
16
17 # Derived class (Single level inheritance)
18 class Car(Vehicle):
19     def __init__(self, brand, model, doors):
20         super().__init__(brand, model)
21         self.doors = doors
22
23     def car_info(self):
24         # Can access public and protected members
25         return f"Car: {self.brand}, {self._display_model()}, Doors: {self.doors}"
26
27     def demonstrate_access(self):
28         print("Public access:", self.brand)
29         print("Protected access:", self._model)
30         # print("Private access:", self.__year) # This would cause error
31
32 # Usage
33 my_car = Car("Toyota", "Camry", 4)
34 print(my_car.car_info())
35 print(my_car.start_engine())
36 my_car.demonstrate_access()

```

- **Public:** Accessible everywhere (brand)
- **Protected:** Accessible in class and subclasses (_model)

- **Private:** Only accessible within same class (___year)

Mnemonic

Public = Everyone, Protected = Family, Private = Personal

Question 2(a OR) [3 marks]

Explain constructor with example.

Solution

Answer:

Table 6. Constructor Types

Type	Method	Purpose
Default	<code>__init__(self)</code>	Initialize with default values
Parameterized	<code>__init__(self, params)</code>	Initialize with custom values

Listing 6. Constructor Example

```

1 class Student:
2     def __init__(self, name="Unknown", age=18): # Constructor
3         self.name = name
4         self.age = age
5         print(f"Student {name} created")
6
7     def display(self):
8         print(f"Name: {self.name}, Age: {self.age}")
9
10 # Object creation calls constructor automatically
11 s1 = Student("Alice", 20)
12 s2 = Student() # Uses default values

```

- **Constructor:** Special method called when object is created
- **Purpose:** Initialize object attributes
- **Automatic:** Called automatically during object creation

Mnemonic

Constructor = Object's Birth Certificate

Question 2(b OR) [4 marks]

Write a program to demonstrate Polymorphism.

Solution

Answer:

Listing 7. Polymorphism Example

```

1 # Base class
2 class Animal:
3     def make_sound(self):

```

```

4         pass
5
6     # Derived classes
7     class Dog(Animal):
8         def make_sound(self):
9             return "Woof!"
10
11    class Cat(Animal):
12        def make_sound(self):
13            return "Meow!"
14
15    class Cow(Animal):
16        def make_sound(self):
17            return "Moo!"
18
19    # Polymorphism demonstration
20    def animal_sound(animal):
21        return animal.make_sound()
22
23    # Creating objects
24    animals = [Dog(), Cat(), Cow()]
25
26    # Same method call, different behavior
27    for animal in animals:
28        print(f"{animal.__class__.__name__}: {animal_sound(animal)}")

```

Table 7. Polymorphism Benefits

Benefit	Description
Flexibility	Same interface, different implementations
Maintainability	Easy to add new types
Code Reuse	Common interface for different objects

Mnemonic

Poly = Many, Morph = Forms

Question 2(c OR) [7 marks]

Write a Python to implement multiple and hierarchical inheritance.

Solution**Answer:**

Listing 8. Inheritance Types

```

1     # Multiple Inheritance
2     class Teacher:
3         def __init__(self, subject):
4             self.subject = subject
5
6         def teach(self):
7             return f"Teaching {self.subject}"
8
9     class Researcher:
10        def __init__(self, field):

```

```

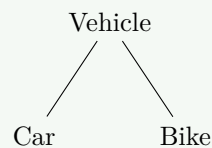
11     self.field = field
12
13     def research(self):
14         return f"Researching in {self.field}"
15
16 # Multiple inheritance
17 class Professor(Teacher, Researcher):
18     def __init__(self, name, subject, field):
19         self.name = name
20         Teacher.__init__(self, subject)
21         Researcher.__init__(self, field)
22
23     def profile(self):
24         return f"Prof. {self.name}: {self.teach()} and {self.research()}"
25
26 # Hierarchical Inheritance
27 class Vehicle:
28     def __init__(self, brand):
29         self.brand = brand
30
31     def start(self):
32         return f"{self.brand} started"
33
34 class Car(Vehicle):
35     def drive(self):
36         return f"{self.brand} car driving"
37
38 class Bike(Vehicle):
39     def ride(self):
40         return f"{self.brand} bike riding"
41
42 # Usage
43 prof = Professor("Smith", "Python", "AI")
44 print(prof.profile())
45
46 car = Car("Honda")
47 bike = Bike("Yamaha")
48 print(car.drive())
49 print(bike.ride())

```

Multiple Inheritance



Hierarchical Inheritance

**Figure 1.** Inheritance Types**Mnemonic**

Multiple = Many Parents, Hierarchical = Tree Structure

Question 3(a) [3 marks]**Explain Push and Pop operations on Stack.**

Solution**Answer:****Table 8.** Stack Operations

Operation	Description	Time Complexity
Push	Add element to top	O(1)
Pop	Remove element from top	O(1)
Peek/Top	View top element	O(1)
isEmpty	Check if stack is empty	O(1)

Listing 9. Stack Push-Pop

```

1 stack = []
2
3 # Push operation
4 stack.append(10) # Push 10
5 stack.append(20) # Push 20
6 print("After push:", stack) # [10, 20]
7
8 # Pop operation
9 item = stack.pop() # Pop 20
10 print(f"Popped: {item}, Stack: {stack}") # [10]

```

- **LIFO:** Last In, First Out principle
- **Top:** Only accessible element for operations

Mnemonic

Stack = Plate Stack - Last plate In, First plate Out

Question 3(b) [4 marks]**Explain Enqueue and Dequeue operations on Queue.****Solution****Answer:****Table 9.** Queue Operations

Operation	Description	Position	Time Complexity
Enqueue	Add element	Rear	O(1)
Dequeue	Remove element	Front	O(1)
Front	View front element	Front	O(1)
Rear	View rear element	Rear	O(1)

Listing 10. Queue Enqueue-Dequeue

```

1 from collections import deque
2
3 queue = deque()
4
5 # Enqueue operation
6 queue.append(10) # Enqueue 10
7 queue.append(20) # Enqueue 20

```

```

8 print("After enqueue:", list(queue)) # [10, 20]
9
10 # Dequeue operation
11 item = queue.popleft() # Dequeue 10
12 print(f"Dequeued: {item}, Queue: {list(queue)}") # [20]

```

- **FIFO:** First In, First Out principle
- **Two ends:** Front for removal, Rear for insertion

Mnemonic

Queue = Line at Store - First person In, First person Out

Question 3(c) [7 marks]

Explain various applications of Stack.

Solution

Answer:

Table 10. Stack Applications

Application	Description	Example
Expression Evaluation	Convert infix to postfix	$(a+b)*c \rightarrow ab+c*$
Function Calls	Manage function call sequence	Recursion handling
Undo Operations	Reverse recent actions	Text editor undo
Browser History	Navigate back through pages	Back button
Parentheses Matching	Check balanced brackets	$\{[(())]\}$ validation

Listing 11. Parentheses Matching

```

1 # Example: Parentheses matching
2 def is_balanced(expression):
3     stack = []
4     pairs = {'(': ')', '[': ']', '{': '}'}
5
6     for char in expression:
7         if char in pairs: # Opening bracket
8             stack.append(char)
9         elif char in pairs.values(): # Closing bracket
10            if not stack:
11                return False
12            if pairs[stack.pop()] != char:
13                return False
14
15     return len(stack) == 0
16
17 # Test
18 print(is_balanced("({[]})")) # True
19 print(is_balanced("({[})")) # False

```

- **Memory Management:** Function call stack in programming
- **Backtracking:** Maze solving, game algorithms
- **Compiler Design:** Syntax analysis and parsing

Mnemonic

Stack Applications = UFPB (Undo, Function, Parentheses, Browser)

Question 3(a OR) [3 marks]

List out limitations of Single Queue.

Solution

Answer:

Table 11. Single Queue Limitations

Limitation	Description	Problem
Memory Wastage	Front space becomes unusable	Inefficient memory usage
Fixed Size	Cannot resize dynamically	Space constraints
False Overflow	Queue appears full when front space empty	Premature capacity limit
No Reuse	Dequeued positions not reusable	Linear space utilization

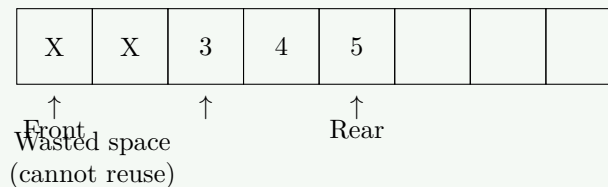


Figure 2. Single Queue Problem

- **Linear Implementation:** Cannot utilize dequeued space
- **Static Array:** Fixed size allocation

Mnemonic

Single Queue = One-Way Street (No U-Turn)

Question 3(b OR) [4 marks]

Differentiate circular and simple queues.

Solution

Answer:

Table 12. Queue Types Comparison

Aspect	Simple Queue	Circular Queue
Memory Usage	Linear, wasteful	Circular, efficient
Space Reuse	No reuse of dequeued space	Reuses all positions
Overflow	False overflow possible	True overflow only
Implementation	Front and rear pointers	Front and rear with modulo

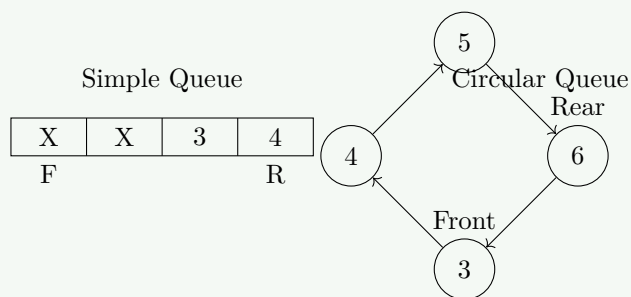


Figure 3. Simple vs Circular Queue

Listing 12. Circular Queue Implementation

```

1 class CircularQueue:
2     def __init__(self, size):
3         self.size = size
4         self.queue = [None] * size
5         self.front = -1
6         self.rear = -1
7
8     def enqueue(self, item):
9         if (self.rear + 1) % self.size == self.front:
10             print("Queue Full")
11             return
12         if self.front == -1:
13             self.front = 0
14         self.rear = (self.rear + 1) % self.size
15         self.queue[self.rear] = item
16
17     def dequeue(self):
18         if self.front == -1:
19             print("Queue Empty")
20             return None
21         item = self.queue[self.front]
22         if self.front == self.rear:
23             self.front = self.rear = -1
24         else:
25             self.front = (self.front + 1) % self.size
26         return item

```

Mnemonic

Circular = Ring Road (Continuous), Simple = Dead End Street

Question 3(c OR) [7 marks]Convert the following infix expression into postfix: $(a * b) * (c ^ (d + e) - f)$ **Solution****Answer:**

Table 13. Operator Precedence

Operator	Precedence	Associativity
\wedge	3	Right to Left
$*, /$	2	Left to Right
$+, -$	1	Left to Right

Step-by-step conversion:

1. $(a * b) \rightarrow ab*$
2. $(d + e) \rightarrow de+$
3. $c \wedge (de+) \rightarrow c de+ \wedge$
4. $(c de+ \wedge) - f \rightarrow c de+ \wedge f -$
5. $(ab*) * (c de+ \wedge f -) \rightarrow ab* c de+ \wedge f - *$

Final Answer: $ab*cde+\wedge f-*$

Listing 13. Infix to Postfix

```

1 def infix_to_postfix(expression):
2     precedence = {'+': 1, '-': 1, '*': 2, '/': 2, '^': 3}
3     stack = []
4     output = []
5
6     for char in expression:
7         if char.isalnum():
8             output.append(char)
9         elif char == '(':
10            stack.append(char)
11        elif char == ')':
12            while stack and stack[-1] != '(':
13                output.append(stack.pop())
14            stack.pop() # Remove '('
15        elif char in precedence:
16            while (stack and stack[-1] != '(' and
17                   stack[-1] in precedence and
18                   precedence[stack[-1]] >= precedence[char]):
19                output.append(stack.pop())
20            stack.append(char)
21
22    while stack:
23        output.append(stack.pop())
24
25    return ''.join(output)
26
27 # Test
28 result = infix_to_postfix("(a*b)*(c^(d+e)-f)")
29 print("Postfix:", result) # ab*cde+\wedge f-*

```

Mnemonic

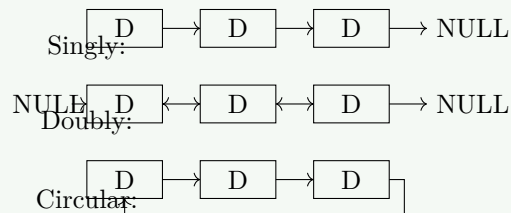
PEMDAS for precedence, Stack for operators

Question 4(a) [3 marks]

List types of Linked List.

Solution**Answer:****Table 14.** Linked List Types

Type	Description	Key Feature
Singly Linked	One pointer to next node	Forward traversal only
Doubly Linked	Pointers to next and previous	Bidirectional traversal
Circular Linked	Last node points to first	No NULL pointer
Doubly Circular	Doubly + Circular features	Both directions + circular

**Figure 4.** Linked List Types

- **Memory:** Each node contains data and pointer(s)
- **Dynamic:** Size can change during runtime

Mnemonic

SDCD - Singly, Doubly, Circular, Doubly-Circular

Question 4(b) [4 marks]**Differentiate between circular linked list and singly linked list.****Solution****Answer:****Table 15.** Singly vs Circular Linked List

Aspect	Singly Linked List	Circular Linked List
Last Node	Points to NULL	Points to first node
Traversal	Ends at NULL	Continuous loop
Memory	Last node stores NULL	No NULL pointer
Detection	Check for NULL	Check for starting node

Listing 14. Singly vs Circular Traversal

```

1  # Singly Linked List Node
2  class SinglyNode:
3      def __init__(self, data):
4          self.data = data
5          self.next = None
6
7  # Circular Linked List Node
8  class CircularNode:
9      def __init__(self, data):
10         self.data = data

```

```

11     self.next = None
12
13 def traverse_singly(head):
14     current = head
15     while current: # Stops at NULL
16         print(current.data)
17         current = current.next
18
19 def traverse_circular(head):
20     if not head:
21         return
22     current = head
23     while True:
24         print(current.data)
25         current = current.next
26         if current == head: # Back to start
27             break

```

Mnemonic

Singly = Dead End, Circular = Race Track

Question 4(c) [7 marks]

Implement a program to perform following operation on singly linked list: a. Insert a node at the beginning of a singly linked list. b. Insert a node at the end of a singly linked list.

Solution

Answer:

Listing 15. Singly Linked List Insertion

```

1 class Node:
2     def __init__(self, data):
3         self.data = data
4         self.next = None
5
6 class SinglyLinkedList:
7     def __init__(self):
8         self.head = None
9
10    def insert_at_beginning(self, data):
11        """Insert node at the beginning"""
12        new_node = Node(data)
13        new_node.next = self.head
14        self.head = new_node
15        print(f"Inserted {data} at beginning")
16
17    def insert_at_end(self, data):
18        """Insert node at the end"""
19        new_node = Node(data)
20
21        if not self.head: # Empty list
22            self.head = new_node
23            print(f"Inserted {data} at end (first node)")
24            return
25

```

```

26         # Traverse to last node
27         current = self.head
28         while current.next:
29             current = current.next
30
31         current.next = new_node
32         print(f"Inserted {data} at end")
33
34     def display(self):
35         """Display the linked list"""
36         if not self.head:
37             print("List is empty")
38             return
39
40         current = self.head
41         elements = []
42         while current:
43             elements.append(str(current.data))
44             current = current.next
45
46         print(" -> ".join(elements) + " -> NULL")
47
48     # Usage example
49     sll = SinglyLinkedList()
50
51     # Insert at beginning
52     sll.insert_at_beginning(10)
53     sll.insert_at_beginning(20)
54     sll.display()  # 20 -> 10 -> NULL
55
56     # Insert at end
57     sll.insert_at_end(30)
58     sll.insert_at_end(40)
59     sll.display()  # 20 -> 10 -> 30 -> 40 -> NULL

```

Table 16. Insertion Operations

Operation	Time Complexity	Steps
Beginning	O(1)	1. Create node 2. Point to head 3. Update head
End	O(n)	1. Create node 2. Traverse to end 3. Link last node

Mnemonic

Beginning = Quick (O(1)), End = Journey (O(n))

Question 4(a OR) [3 marks]

Explain doubly linked list.

Solution

Answer:

Table 17. Doubly Linked List Features

Feature	Description
Two Pointers	prev and next in each node
Bidirectional	Can traverse forward and backward
Memory	Extra space for prev pointer
Flexibility	Easy insertion/deletion anywhere

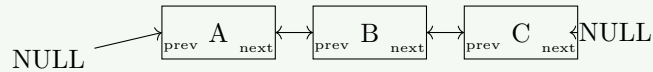


Figure 5. Doubly Linked List Structure

- **Advantages:** Bidirectional traversal, easier deletion
- **Disadvantages:** Extra memory for prev pointer

Mnemonic

Doubly = Two-Way Street

Question 4(b OR) [4 marks]

Describe applications of Linked List.

Solution

Answer:

Table 18. Linked List Applications

Application	Use Case	Benefit
Dynamic Arrays	When size varies	Efficient memory usage
Stack/Queue	LIFO/FIFO operations	Dynamic size
Graphs	Adjacency list representation	Space efficient
Music Playlist	Previous/Next songs	Easy navigation
Browser History	Back/Forward navigation	Dynamic history
Undo Operations	Text editors	Efficient undo/redo

Mnemonic

Linked Lists = Dynamic, Flexible, Connected

Question 4(c OR) [7 marks]

Implement Merge Sort algorithm.

Solution

Answer:

Listing 16. Merge Sort Implementation

```

1 def merge_sort(arr):
2     """Merge Sort implementation"""

```

```

3     if len(arr) <= 1:
4         return arr
5
6     # Divide the array into two halves
7     mid = len(arr) // 2
8     left_half = arr[:mid]
9     right_half = arr[mid:]
10
11    # Recursively sort both halves
12    left_sorted = merge_sort(left_half)
13    right_sorted = merge_sort(right_half)
14
15    # Merge the sorted halves
16    return merge(left_sorted, right_sorted)
17
18    def merge(left, right):
19        """Merge two sorted arrays"""
20        result = []
21        i = j = 0
22
23        # Compare elements and merge
24        while i < len(left) and j < len(right):
25            if left[i] <= right[j]:
26                result.append(left[i])
27                i += 1
28            else:
29                result.append(right[j])
30                j += 1
31
32        # Add remaining elements
33        result.extend(left[i:])
34        result.extend(right[j:])
35
36        return result
37
38    # Example usage
39    def demonstrate_merge_sort():
40        arr = [64, 34, 25, 12, 22, 11, 90]
41        print("Original array:", arr)
42
43        sorted_arr = merge_sort(arr)
44        print("Sorted array:", sorted_arr)
45
46    demonstrate_merge_sort()

```

Table 19. Merge Sort Analysis

Aspect	Value
Time Complexity	$O(n \log n)$
Space Complexity	$O(n)$
Stability	Stable
Type	Divide and Conquer

- **Divide:** Split array into two halves
- **Conquer:** Recursively sort both halves
- **Combine:** Merge sorted halves

Mnemonic

Merge Sort = Divide, Sort, Merge

Question 5(a) [3 marks]

Describe applications of binary tree.

Solution

Answer:

Table 20. Binary Tree Applications

Application	Description	Example
Expression Trees	Mathematical expression representation	$(a+b)*c$
Decision Trees	Decision making in AI/ML	Classification algorithms
File Systems	Directory structure organization	Folder hierarchy
Database Indexing	B-trees for efficient searching	Database indices
Huffman Coding	Data compression technique	File compression
Heap Operations	Priority queues implementation	Task scheduling

- **Hierarchical Data:** Naturally represents tree-like structures
- **Efficient Search:** Binary search trees provide $O(\log n)$ operations
- **Memory Management:** Used in compiler design for syntax trees

Mnemonic

Binary Trees = EDFDHH (Expression, Decision, File, Database, Huffman, Heap)

Question 5(b) [4 marks]

Explain Indegree and Outdegree of Binary Tree with example.

Solution

Answer:

Table 21. Degree Definitions

Term	Definition	Binary Tree Value
Indegree	Number of edges coming into a node	0 (root) or 1 (others)
Outdegree	Number of edges going out of a node	0, 1, or 2
Degree	Total edges connected to node	Indegree + Outdegree

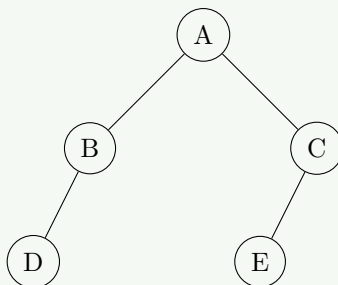


Figure 6. Binary Tree Example**Table 22.** Example Analysis

Node	Indegree	Outdegree	Node Type
A	0	2	Root
B	1	1	Internal
C	1	1	Internal
D	1	0	Leaf
E	1	0	Leaf

Mnemonic

In = Coming In, Out = Going Out

Question 5(c) [7 marks]

Write a program to implement construction of binary search trees.

Solution**Answer:****Listing 17.** Binary Search Tree Construction

```

1 class TreeNode:
2     def __init__(self, data):
3         self.data = data
4         self.left = None
5         self.right = None
6
7 class BinarySearchTree:
8     def __init__(self):
9         self.root = None
10
11     def insert(self, data):
12         """Insert a node in BST"""
13         if self.root is None:
14             self.root = TreeNode(data)
15         else:
16             self._insert_recursive(self.root, data)
17
18     def _insert_recursive(self, node, data):
19         if data < node.data:
20             if node.left is None:
21                 node.left = TreeNode(data)
22             else:
23                 self._insert_recursive(node.left, data)
24         elif data > node.data:
25             if node.right is None:
26                 node.right = TreeNode(data)
27             else:
28                 self._insert_recursive(node.right, data)
29
30     def search(self, data):
31         """Search for a node in BST"""
32         return self._search_recursive(self.root, data)
33

```

```

34 def _search_recursive(self, node, data):
35     if node is None or node.data == data:
36         return node
37
38     if data < node.data:
39         return self._search_recursive(node.left, data)
40     else:
41         return self._search_recursive(node.right, data)
42
43 def inorder_traversal(self):
44     """Inorder traversal (Left, Root, Right)"""
45     result = []
46     self._inorder_recursive(self.root, result)
47     return result
48
49 def _inorder_recursive(self, node, result):
50     if node:
51         self._inorder_recursive(node.left, result)
52         result.append(node.data)
53         self._inorder_recursive(node.right, result)
54
55 # Example usage
56 bst = BinarySearchTree()
57 values = [50, 30, 70, 20, 40, 60, 80]
58
59 print("Inserting values:", values)
60 for value in values:
61     bst.insert(value)
62
63 print("\nInorder traversal:", bst.inorder_traversal())

```

Table 23. BST Operations

Operation	Time Complexity	Description
Insert	$O(\log n)$ average, $O(n)$ worst	Add new node
Search	$O(\log n)$ average, $O(n)$ worst	Find specific node
Delete	$O(\log n)$ average, $O(n)$ worst	Remove node
Traversal	$O(n)$	Visit all nodes

Mnemonic

BST Rule = Left < Root < Right

Question 5(a OR) [3 marks]

Define level, degree and leaf node in binary tree.

Solution

Answer:

Table 24. Binary Tree Terms

Term	Definition	Example
Level	Distance from root (root = level 0)	Root=0, Children=1, etc.
Degree	Number of children a node has	0, 1, or 2
Leaf Node	Node with no children (degree = 0)	Terminal nodes

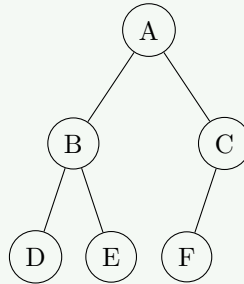


Figure 7. Binary Tree with Levels

- **Height:** Maximum level in tree
- **Depth:** Same as level for a node

Mnemonic

Level = Floor number, Degree = Children count, Leaf = No children

Question 5(b OR) [4 marks]

Explain complete binary tree with example.

Solution

Answer:

Table 25. Binary Tree Types

Type	Description	Property
Complete	All levels filled except last, left-filled	Efficient array representation
Full	Every node has 0 or 2 children	No single-child nodes
Perfect	All levels completely filled	$2^h - 1$ nodes

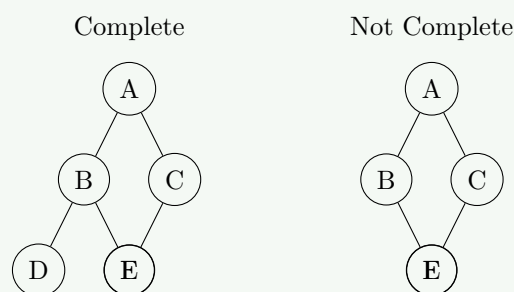


Figure 8. Complete vs Non-Complete Binary Tree

Listing 18. Complete Binary Tree Structure

```

1 class CompleteBinaryTree:
2     def __init__(self):

```

```

3     self.tree = []
4
5     def insert(self, data):
6         """Insert in complete binary tree manner"""
7         self.tree.append(data)
8
9     def get_parent_index(self, i):
10        return (i - 1) // 2
11
12    def get_left_child_index(self, i):
13        return 2 * i + 1
14
15    def get_right_child_index(self, i):
16        return 2 * i + 2

```

Mnemonic

Complete = All floors full except last, filled left to right

Question 5(c OR) [7 marks]

Construct a Binary Search Tree (BST) for the following sequence of numbers: 50, 70, 60, 20, 90, 10, 40, 100

Solution

Answer:

Step-by-step BST Construction:

1. Insert 50: Root
2. Insert 70: $70 > 50 \rightarrow$ Right
3. Insert 60: $60 > 50 \rightarrow$ Right, $60 < 70 \rightarrow$ Left
4. Insert 20: $20 < 50 \rightarrow$ Left
5. Insert 90: $90 > 50 \rightarrow$ Right, $90 > 70 \rightarrow$ Right
6. Insert 10: $10 < 50 \rightarrow$ Left, $10 < 20 \rightarrow$ Left
7. Insert 40: $40 < 50 \rightarrow$ Left, $40 > 20 \rightarrow$ Right
8. Insert 100: $100 > 50 \rightarrow$ Right... $100 > 90 \rightarrow$ Right

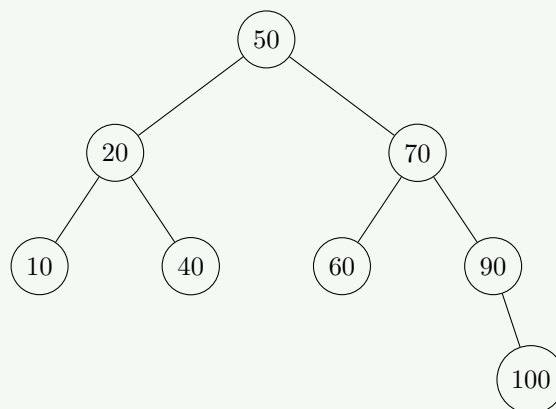


Figure 9. Final BST Structure

Listing 19. BST Construction

```

1 # Construct the BST
2 bst = BST()

```

```
3 sequence = [50, 70, 60, 20, 90, 10, 40, 100]
4
5 for num in sequence:
6     bst.insert(num)
```

Table 26. Traversal Results

Traversal	Result
Inorder	10, 20, 40, 50, 60, 70, 90, 100
Preorder	50, 20, 10, 40, 70, 60, 90, 100
Postorder	10, 40, 20, 60, 100, 90, 70, 50

Mnemonic

BST Construction = Compare, Choose direction, Insert