Data Structure with Python (4331601) - Summer 2025 Solution

Milav Dabgar

May 09, 2025

Question 1(a) [3 marks]

Differentiate between Linear and Non Linear Data Structure.

Solution

Answer:

Table 1. Linear vs Non-Linear Data Structure

Linear Data Structure	Non-Linear Data Structure
Elements stored sequentially	Elements stored hierarchically
Single level arrangement	Multi-level arrangement
Easy traversal	Complex traversal
Examples: Array, Stack, Queue	Examples: Tree, Graph

Mnemonic

Linear flows Like water, Non-linear Navigates Networks

Question 1(b) [4 marks]

Explain different concepts of Object Oriented programming.

Solution

Answer:
Table of OOP Concepts:

Table 2. OOP Concepts

Concept	Description
Encapsulation	Binding data and methods together
Inheritance	Acquiring properties from parent class
Polymorphism	One name, multiple forms
Abstraction	Hiding implementation details

- **Encapsulation**: Data hiding and bundling
- **Inheritance**: Code reusability through parent-child relationship
- **Polymorphism**: Method overriding and overloading
- **Abstraction**: Interface without implementation

Mnemonic

Every Intelligent Programmer Abstracts

Question 1(c) [7 marks]

Define Polymorphism. Write a python program for polymorphism through inheritance.

Solution

Answer:
Polymorphism means ”many forms” - same method name behaving differently in different classes.
Code:

Listing 1. Polymorphism Example

```python
class Animal:
    def sound(self):
        pass

class Dog(Animal):
    def sound(self):
        return "Bark"

class Cat(Animal):
    def sound(self):
        return "Meow"

# Polymorphism in action
animals = [Dog(), Cat()]
for animal in animals:
    print(animal.sound())
```

- **Polymorphism**: Same interface, different implementation
- **Runtime binding**: Method called based on object type
- **Code flexibility**: Easy to extend with new classes

Mnemonic

Polymorphism Provides Perfect Programming

Question 1(c OR) [7 marks]

Define Abstraction. Write a python program to understand the concept of abstract class.

Solution

Answer:
Abstraction hides implementation details and shows only essential features.
Code:

Listing 2. Abstraction Example

```python
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
```

```python
        pass

class Rectangle(Shape):
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width

# Usage
rect = Rectangle(5, 3)
print(f"Area: {rect.area()}")
```

- **Abstract class**: Cannot be instantiated directly
- **Abstract method**: Must be implemented by child classes
- **Interface definition**: Provides template for subclasses

Mnemonic

Abstraction Avoids Actual implementation

Question 2(a) [3 marks]

Define Following terms: I. Best case II. Worst case III. Average case

Solution

Answer:

Table 3. Time Complexity Cases

Case	Definition
Best case	Minimum time required for algorithm
Worst case	Maximum time required for algorithm
Average case	Expected time for random input

Mnemonic

Best-Worst-Average = Performance Analysis

Question 2(b) [4 marks]

Explain infix, postfix & prefix expressions.

Solution

Answer:

Table 4. Expression Types

Expression	Operator Position	Example
Infix	Between operands	A + B
Prefix	Before operands	+ A B
Postfix	After operands	A B +

- **Infix**: Natural mathematical notation
- **Prefix**: Polish notation
- **Postfix**: Reverse Polish notation
- **Stack usage**: Postfix eliminates parentheses

Mnemonic

In-Pre-Post = Position of operator

Question 2(c) [7 marks]

Define circular queue. Explain INSERT and DELETE operations of circular queue with diagrams.

Solution

Answer:
Circular Queue: Linear data structure where last position connects to first position.
Diagram:

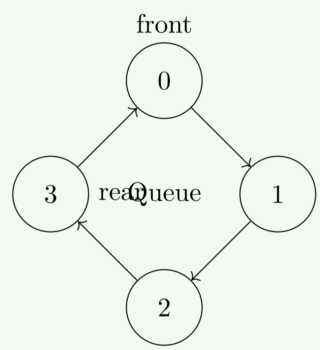

Figure 1. Circular Queue

INSERT Operation:
1. Check if queue is full
2. If not full, increment rear
3. If rear exceeds size, set rear = 0
4. Insert element at rear position

DELETE Operation:
1. Check if queue is empty
2. If not empty, remove element from front
3. Increment front
4. If front exceeds size, set front = 0

- **Circular nature**: Efficient memory utilization
- **No shifting**: Elements remain in place
- **Front-rear pointers**: Track queue boundaries

Mnemonic

Circular Saves Space

Question 2(a OR) [3 marks]

List out different Data Structure with examples.

Solution

Answer:

Table 5. Data Structure Types

Type	Data Structure	Example
Linear	Array	[1,2,3,4]
Linear	Stack	Function calls
Linear	Queue	Printer queue
Non-Linear	Tree	File system
Non-Linear	Graph	Social network

Mnemonic

Arrays-Stacks-Queues = Linear, Trees-Graphs = Non-linear

Question 2(b OR) [4 marks]

Discuss how the concept of circular queue is different from simple queue.

Solution

Answer:

Table 6. Simple vs Circular Queue

Simple Queue	Circular Queue
Linear arrangement	Circular arrangement
Memory wastage	Efficient memory use
Fixed front and rear	Wraparound pointers
False overflow	True overflow detection

- **Memory efficiency**: Circular reuses deleted spaces
- **Pointer management**: Modulo arithmetic for wraparound
- **Performance**: Better space utilization

Mnemonic

Circular Conquers memory problems

Question 2(c OR) [7 marks]

Define stack. Explain PUSH & POP operation with example. Write an algorithm for

PUSH and POP operations of stack.

Solution

Answer:
Stack: LIFO (Last In First Out) data structure.
PUSH Algorithm:

1. Check if stack is full
2. If not full, increment top
3. Insert element at top position
4. Update top pointer

POP Algorithm:

1. Check if stack is empty
2. If not empty, store top element
3. Decrement top pointer
4. Return stored element

Example:

Listing 3. Stack Example

```
Stack: [10, 20, 30] <- top
PUSH 40: [10, 20, 30, 40] <- top
POP: returns 40, stack: [10, 20, 30] <- top
```

- **LIFO principle**: Last element added is first removed
- **Top pointer**: Tracks current stack position
- **Overflow/Underflow**: Check before operations

Mnemonic

Stack Stores in Last-in-first-out

Question 3(a) [3 marks]

Convert following infix expression to postfix: (((A - B) * C) + ((D - E) / F))

Solution

Answer:
Step-by-step conversion:

Table 7. Infix to Postfix Conversion

Step	Scanned	Stack	Postfix
1	(	(	
2	(	((	
3	(	(((	
4	A	(((	A
5	-	(((-	A
6	B	(((-	AB
7	)	((	AB-
8	*	((*	AB-
9	C	((*	AB-C
10	)	(	AB-C*
11	+	(+	AB-C*
12	(	(+(	AB-C*
13	(	(+((	AB-C*
14	D	(+((	AB-C*D
15	-	(+((-	AB-C*D
16	E	(+((-	AB-C*DE
17	)	(+	AB-C*DE-
18	/	(+(/	AB-C*DE-
19	F	(+(/	AB-C*DE-F
20	)	(+	AB-C*DE-F/
21	)		AB-C*DE-F/+

Final Answer: AB-C*DE-F/+

Mnemonic

Postfix Places operators after operands

Question 3(b) [4 marks]

Write a short note on doubly linked list.

Solution

Answer:
Doubly Linked List: Linear data structure with bidirectional links.
Structure:

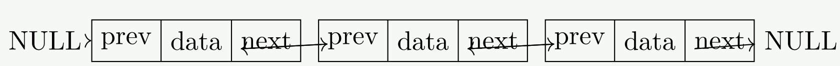

Figure 2. Doubly Linked List

Advantages:
- **Bidirectional traversal**: Forward and backward navigation
- **Efficient deletion**: No need for previous node reference
- **Better insertion**: Can insert before given node easily

Disadvantages:
- **Extra memory**: Additional pointer storage
- **Complex operations**: More pointer manipulations

Mnemonic

Doubly Delivers Bidirectional Benefits

Question 3(c) [7 marks]

Write a Python Program to delete first and last node from singly linked list.

Solution

Answer:
Code:

Listing 4. Singly Linked List Deletion

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

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

    def delete_first(self):
        if self.head is None:
            return "List is empty"
        self.head = self.head.next
        return "First node deleted"

    def delete_last(self):
        if self.head is None:
            return "List is empty"
        if self.head.next is None:
            self.head = None
            return "Last node deleted"

        current = self.head
        while current.next.next:
            current = current.next
        current.next = None
        return "Last node deleted"

    def display(self):
        elements = []
        current = self.head
        while current:
            elements.append(current.data)
            current = current.next
        return elements

# Usage
ll = LinkedList()
# Add nodes and test deletion
```

- **Delete first**: Update head pointer
- **Delete last**: Traverse to second last node
- **Edge cases**: Empty list and single node

Mnemonic

Delete Delivers by pointer updates

Question 3(a OR) [3 marks]

List different applications of Queue.

Solution

Answer:
Queue Applications:

Table 8. Queue Applications

Application	Usage
CPU Scheduling	Process management
Print Queue	Document printing
BFS Algorithm	Graph traversal
Buffer	Data streaming

- **FIFO nature**: First come first served
- **Real-time systems**: Handle requests in order
- **Resource sharing**: Fair allocation

Mnemonic

Queues Quietly handle ordered operations

Question 3(b OR) [4 marks]

Explain different operations which we can perform on singly linked list.

Solution

Answer:
Singly Linked List Operations:

Table 9. Singly Linked List Operations

Operation	Description
Insertion	Add node at beginning/end/middle
Deletion	Remove node from any position
Traversal	Visit all nodes sequentially
Search	Find specific data in list
Count	Count total number of nodes

- **Dynamic size**: Grow/shrink during runtime
- **Memory efficiency**: Allocate as needed
- **Sequential access**: No random access

Mnemonic

Insert-Delete-Traverse-Search-Count

Question 3(c OR) [7 marks]

Write an algorithm to insert a new node at the end of doubly linked list.

Solution

Answer:
Algorithm for insertion at end:

1. Create new node with given data
2. Set new node's next = NULL
3. If list is empty:
   - Set head = new node
   - Set new node's prev = NULL
4. Else:
   - Traverse to last node
   - Set last node's next = new node
   - Set new node's prev = last node
5. Return success

Code:

Listing 5. Insert at End Doubly Linked List

```python
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
```

- **Two-way linking**: Update both next and prev pointers
- **End insertion**: Traverse to find last node
- **Bidirectional connection**: Maintain list integrity

Mnemonic

Insert Intelligently with bidirectional links

Question 4(a) [3 marks]

Write a python program for linear search.

Solution

Answer:
Code:

Listing 6. Linear Search

```python
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i
    return -1

# Example usage
data = [10, 20, 30, 40, 50]
result = linear_search(data, 30)
print(f"Element found at index: {result}")
```

- **Sequential search**: Check each element one by one
- **Time complexity**: O(n)
- **Simple implementation**: Easy to understand

Mnemonic

Linear Looks through every element

Question 4(b) [4 marks]

Write a short note on Circular linked list.

Solution

Answer:
Circular Linked List: Last node points back to first node forming a circle.
Diagram:

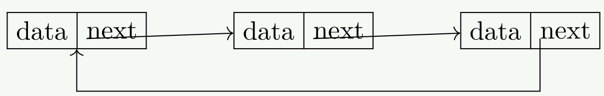

Figure 3. Circular Linked List

Characteristics:

- **No NULL pointers**: Last node connects to first
- **Continuous traversal**: Can traverse infinitely
- **Memory efficiency**: Better cache performance
- **Applications**: Round-robin scheduling, multiplayer games

Advantages:

- **Efficient insertion**: At any position
- **No wasted pointers**: All nodes connected

Mnemonic

Circular Connects everything in a loop

Question 4(c) [7 marks]

Explain Quick sort algorithm with an example.

Solution

Answer:
Quick Sort: Divide and conquer sorting algorithm using pivot element.
Algorithm:

1. Choose pivot element
2. Partition array around pivot
3. Recursively sort left subarray
4. Recursively sort right subarray

Example: Sort [64, 34, 25, 12, 22, 11, 90]
Step 1: Pivot = 64

```
[34, 25, 12, 22, 11] 64 [90]
```

Step 2: Sort left partition [34, 25, 12, 22, 11] (Pivot = 34)

```
[25, 12, 22, 11] 34 []
```

Final sorted: [11, 12, 22, 25, 34, 64, 90]

- **Divide and conquer**: Break problem into smaller parts
- **In-place sorting**: Minimal extra memory
- **Average complexity**: O(n log n)

Mnemonic

Quick Partitions then conquers

Question 4(a OR) [3 marks]

Explain Binary search algorithm with an example.

Solution

Answer:
Binary Search: Search algorithm for sorted arrays using divide and conquer.
Algorithm:

1. Set left = 0, right = array length - 1
2. While left <= right:
   - Calculate mid = (left + right) / 2
   - If target = array[mid], return mid
   - If target < array[mid], right = mid - 1
   - If target > array[mid], left = mid + 1
3. Return -1 if not found

Example: Search 22 in [11, 12, 22, 25, 34, 64, 90]

Table 10. Binary Search Trace

Step	Left	Right	Mid	Value	Action
1	0	6	3	25	22 < 25, right = 2
2	0	2	1	12	22 > 12, left = 2
3	2	2	2	22	Found!

Mnemonic

Binary Bisects to find quickly

Question 4(b OR) [4 marks]

Discuss different applications of linked list.

Solution

Answer:
Linked List Applications:

Table 11. Linked List Applications

Application	Usage
Dynamic Arrays	Resizable data storage
Stack/Queue Implementation	LIFO/FIFO structures
Graph Representation	Adjacency lists
Memory Management	Free memory blocks
Music Playlist	Next/previous song navigation

- **Dynamic memory**: Allocate as needed
- **Efficient insertion/deletion**: No shifting required
- **Flexible structure**: Adapt to changing requirements

Mnemonic

Linked Lists Live in dynamic applications

Question 4(c OR) [7 marks]

Write a python program for insertion sort with an example.

Solution

Answer:
Code:

Listing 7. Insertion Sort

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

        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1

        arr[j + 1] = key

    return arr

# Example
data = [64, 34, 25, 12, 22, 11, 90]
sorted_data = insertion_sort(data)
print(f"Sorted array: {sorted_data}")
```

Step-by-step example:
- Initial: [64, 34, 25, 12, 22, 11, 90]
- Pass 1: [34, 64, 25, 12, 22, 11, 90]

- Pass 2: [25, 34, 64, 12, 22, 11, 90]
- Pass 3: [12, 25, 34, 64, 22, 11, 90]
- Pass 4: [12, 22, 25, 34, 64, 11, 90]
- Pass 5: [11, 12, 22, 25, 34, 64, 90]
- Pass 6: [11, 12, 22, 25, 34, 64, 90]
- **Card sorting analogy**: Like arranging playing cards
- **Stable sort**: Maintains relative order of equal elements
- **Online algorithm**: Can sort list as it receives data

Mnemonic

Insertion Inserts in right position

Question 5(a) [3 marks]

Define following terms: I. Complete Binary tree II. In-degree III. Out-degree.

Solution

Answer:

Table 12. Binary Tree Terms

Term	Definition
Complete Binary Tree	All levels filled except possibly last level from left
In-degree	Number of edges coming into a node
Out-degree	Number of edges going out from a node

Mnemonic

Complete-In-Out = Tree terminology

Question 5(b) [4 marks]

Explain bubble sort algorithm with an example.

Solution

Answer:
Bubble Sort: Compare adjacent elements and swap if in wrong order.
Algorithm:

1. For each pass (0 to n-1):
2. For each element (0 to n-pass-1):
3. If arr[j] > arr[j+1]:
4. Swap arr[j] and arr[j+1]

Example: [64, 34, 25, 12]

Table 13. Bubble Sort Trace

Pass	Comparisons	Result
1	64>34(swap), 64>25(swap), 64>12(swap)	[34,25,12,64]
2	34>25(swap), 34>12(swap)	[25,12,34,64]
3	25>12(swap)	[12,25,34,64]

- **Bubble up**: Largest element bubbles to end
- **Multiple passes**: Each pass places one element correctly
- **Simple implementation**: Easy to understand

Mnemonic

Bubble Brings biggest to back

Question 5(c) [7 marks]

Create a Binary Search Tree for the keys 15, 35, 12, 48, 5, 25, 58, 8 and write the Preorder, Inorder and Postorder traversal sequences.

Solution

Answer:
BST Construction:

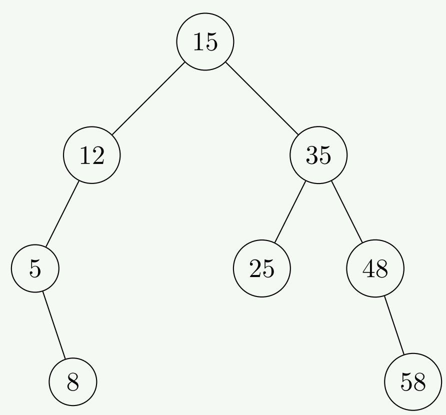

Figure 4. Binary Search Tree

Traversal Sequences:

Table 14. Traversals

Traversal	**Sequence**
Preorder	15, 12, 5, 8, 35, 25, 48, 58
Inorder	5, 8, 12, 15, 25, 35, 48, 58
Postorder	8, 5, 12, 25, 58, 48, 35, 15

Traversal Rules:

- **Preorder**: Root → Left → Right
- **Inorder**: Left → Root → Right (gives sorted order)
- **Postorder**: Left → Right → Root

Mnemonic

Pre-In-Post = Root position

Question 5(a OR) [3 marks]

Define binary tree. Explain searching a node in binary tree.

Solution

Answer:
Binary Tree: Hierarchical data structure where each node has at most two children.
Search Algorithm:

1. Start from root
2. If target = current node, return found
3. If target < current node, go left
4. If target > current node, go right
5. Repeat until found or reach NULL

- **Hierarchical structure**: Parent-child relationship
- **Binary property**: Maximum two children per node
- **Search efficiency**: O(log n) for balanced trees

Mnemonic

Binary Branches into two paths

Question 5(b OR) [4 marks]

Give the trace to sort the given data using bubble sort method. Data are: 44, 72, 94, 28, 18, 442, 41

Solution

Answer:
Bubble Sort Trace:

Table 15. Bubble Sort Trace

Pass	Array State	Swaps
Initial	[44, 72, 94, 28, 18, 442, 41]	-
Pass 1	[44, 72, 28, 18, 94, 41, 442]	94>28, 94>18, 442>41
Pass 2	[44, 28, 18, 72, 41, 94, 442]	72>28, 72>18, 94>41
Pass 3	[28, 18, 44, 41, 72, 94, 442]	44>28, 44>18, 72>41
Pass 4	[18, 28, 41, 44, 72, 94, 442]	28>18, 44>41
Pass 5	[18, 28, 41, 44, 72, 94, 442]	No swaps

Final sorted array: [18, 28, 41, 44, 72, 94, 442]

Mnemonic

Bubble sort Bubbles largest to end each pass

Question 5(c OR) [7 marks]

List applications of trees. Explain the technique for converting general tree into a Binary Search Tree with example.

Solution

Answer:
Tree Applications:

Table 16. Tree Applications

Application	Usage
File System	Directory hierarchy
Expression Trees	Mathematical expressions
Decision Trees	AI and machine learning
Heap	Priority queues

General Tree to BST Conversion:
Technique: First Child - Next Sibling Representation
Original General Tree:

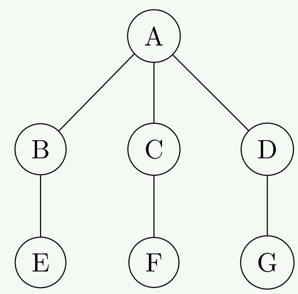

Figure 5. General Tree

Converted to Binary Tree:

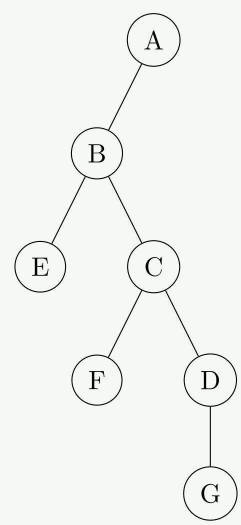

Figure 6. Converted Binary Tree

Steps:

1. **First child**: Becomes left child in binary tree
2. **Next sibling**: Becomes right child in binary tree
3. **Recursive application**: Apply to all nodes

- **Systematic conversion**: Preserves tree structure
- **Binary representation**: Uses only two pointers per node
- **Space efficiency**: Standard binary tree operations apply

Mnemonic

First-child Left, Next-sibling Right