# Data Structure And Application (1333203) - Summer 2024 Solution

Milav Dabgar

June 12, 2024

## Question 1(a) [3 marks]

**Define linear data structure and give its examples.**

**Solution**

A linear data structure is a collection of elements arranged in sequential order where each element has exactly one predecessor and one successor (except first and last elements).

**Table 1.** Linear Data Structures Examples

| Data Structure | Description |
|---|---|
| **Array** | Fixed-size collection of elements accessed by index |
| **Linked List** | Chain of nodes with data and reference to next node |
| **Stack** | LIFO (Last In First Out) structure |
| **Queue** | FIFO (First In First Out) structure |

**Mnemonic**

"ALSQ are in a Line"

## Question 1(b) [4 marks]

**Define time and space complexity.**

**Solution**

Time and space complexity measure algorithm efficiency in terms of execution time and memory usage as input size grows.

**Table 2.** Complexity Comparison

| Complexity Type | Definition | Measurement | Importance |
|---|---|---|---|
| **Time Complexity** | Measures execution time required by an algorithm as a function of input size | Big O notation (O(n), O(1), O($n^2$)) | Determines how fast an algorithm runs |
| **Space Complexity** | Measures memory space required by an algorithm as a function of input size | Big O notation (O(n), O(1), O($n^2$)) | Determines how much memory an algorithm needs |

**Mnemonic**

"TS: Time-Speed and Space-Storage"

# Question 1(c) [7 marks]

**Explain the concept of class and object with example.**

**Solution**

Classes and objects are fundamental OOP concepts where classes are blueprints for creation objects with attributes and behaviors.

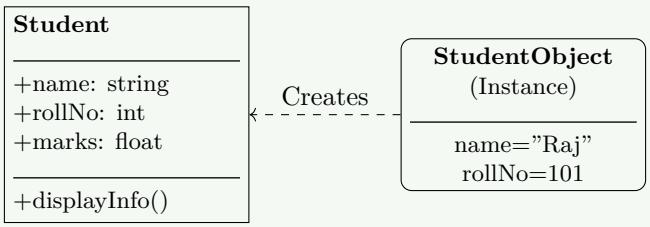



**Figure 1.** Class and Object Relationship

**Listing 1.** Class and Object Example

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

    def displayInfo(self):
        print(f"Name: {self.name}, Roll No: {self.rollNo}, Marks: {self.marks}")

# Creating object
student1 = Student("Raj", 101, 85.5)
student1.displayInfo()
```

- **Class**: Blueprint defining attributes (name, rollNo, marks) and methods (displayInfo)
- **Object**: Instance (student1) created from the class with specific values

**Mnemonic**

"CAR - Class defines Attributes and Routines"

# Question 1(c) OR [7 marks]

**Explain instance method, class method and static method with example.**

**Solution**

Python supports three method types: instance, class, and static methods, each serving different purposes.

**Table 3.** Comparison of Method Types

| Method Type | Decorator | First Parameter | Purpose | Access |
|---|---|---|---|---|
| **Instance Method** | None | self | Operate on instance data | Can access/modify instance state |
| **Class Method** | @classmethod | cls | Operate on class data | Can access/modify class state |
| **Static Method** | @staticmethod | None | Utility functions | Cannot access instance or class state |

**Listing 2.** Method Types Example

```python
class Student:
    school = "ABC School"  # class variable

    def __init__(self, name):
        self.name = name  # instance variable

    def instance_method(self):  # instance method
        return f"Hi {self.name} from {self.school}"

    @classmethod
    def class_method(cls):  # class method
        return f"School is {cls.school}"

    @staticmethod
    def static_method():  # static method
        return "This is a utility function"
```

**Mnemonic**

"ICS: Instance-Self, Class-Cls, Static-Solo"

# Question 2(a) [3 marks]

**Explain concept of recursive function.**

**Solution**

A recursive function is a function that calls itself during its execution to solve smaller instances of the same problem.

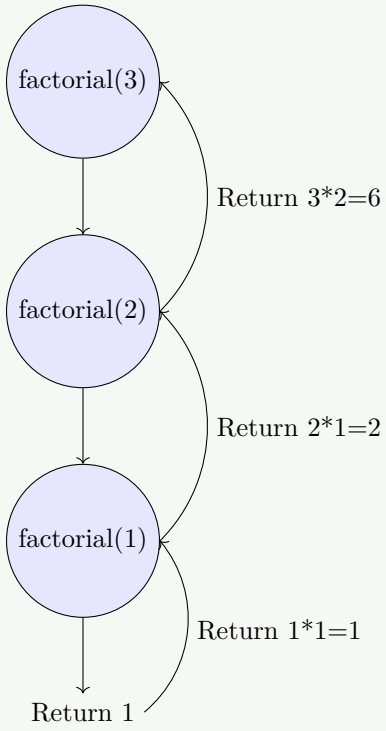



**Figure 2.** Recursive Function Execution

**Mnemonic**

"BASE and RECURSE - Base case stops, Recursion repeats"

## Question 2(b) [4 marks]

**Define stack and queue.**

**Solution**

Stack and queue are linear data structures with different access patterns for data insertion and removal.

**Table 4.** Stack vs Queue

| Feature | Stack | Queue |
|---|---|---|
| **Access Pattern** | LIFO (Last In First Out) | FIFO (First In First Out) |
| **Operations** | Push (insert), Pop (remove) | Enqueue (insert), Dequeue (remove) |
| **Access Points** | Single end (top) | Two ends (front, rear) |
| **Visualization** | Like plates stacked vertically | Like people in a line |
| **Applications** | Function calls, undo operations | Print jobs, process scheduling |

**Mnemonic**

"SLIFF vs QFIFF - Stack-LIFO vs Queue-FIFO"

## Question 2(c) [7 marks]

**Explain basic operations on stack.**

**Solution**

Stack operations follow LIFO (Last In First Out) principle.

**Table 5.** Stack Operations

| Operation | Description | Time Complexity |
|---|---|---|
| **Push** | Insert element at the top | O(1) |
| **Pop** | Remove element from the top | O(1) |
| **Peek/Top** | View top element without removing | O(1) |
| **isEmpty** | Check if stack is empty | O(1) |
| **isFull** | Check if stack is full | O(1) |

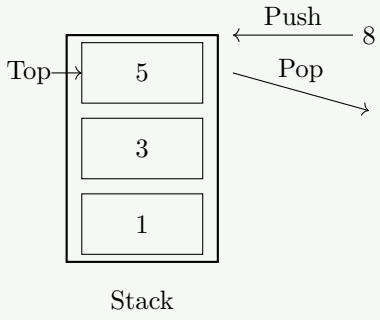



**Figure 3.** Stack Push and Pop

**Listing 3.** Stack Implementation

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

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

    def pop(self):
        if not self.isEmpty():
            return self.items.pop()

    def peek(self):
        if not self.isEmpty():
            return self.items[-1]

    def isEmpty(self):
        return len(self.items) == 0
```

**Mnemonic**

"PIPES - Push In, Pop Exit, See top"

# Question 2(a) OR [3 marks]

**Define singly linked list.**

**Solution**

A singly linked list is a linear data structure with a collection of nodes where each node contains data and a reference to the next node.

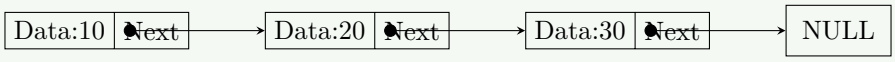



**Figure 4.** Singly Linked List

**Mnemonic**

"DNL - Data and Next Link"

## Question 2(b) OR [4 marks]

**Explain Enqueue and Dequeue operations on Queue.**

**Solution**

Enqueue and Dequeue are the primary operations for adding and removing elements in a queue data structure.

**Table 6.** Queue Operations

| Operation | Description | Implementation | Time Complexity |
|---|---|---|---|
| **Enqueue** | Add element at the rear end | queue.append(element) | O(1) |
| **Dequeue** | Remove element from the front end | element = queue.pop(0) | O(1) linked list, O(n) array |

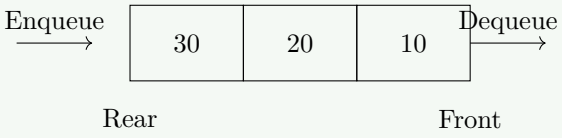



**Figure 5.** Queue Operations

**Mnemonic**

"ERfDFr - Enqueue at Rear, Dequeue from Front"

## Question 2(c) OR [7 marks]

**Convert expression A+B/C+D to postfix and evaluate postfix expression using stack assuming some values for A, B, C and D.**

**Solution**

Converting and evaluating the expression "A+B/C+D" using stack:
**Step 1: Convert to Postfix**

**Table 7.** Infix to Postfix Conversion

| Symbol | Stack | Output | Action |
|---|---|---|---|
| A | | A | Add to output |
| + | + | A | Push to stack |
| B | + | A B | Add to output |
| / | + / | A B | Push to stack (higher prec) |
| C | + / | A B C | Add to output |
| + | + | A B C / | Pop /, push + |
| D | + | A B C / + D | Add to output |
| End | | A B C / + D + | Pop remaining |

**Final Postfix:** $ABC/+D+$
**Step 2: Evaluate with values A=5, B=10, C=2, D=3**

**Table 8.** Postfix Evaluation

| Symbol | Stack | Calculation |
|---|---|---|
| 5 (A) | 5 | Push value |
| 10 (B) | 5, 10 | Push value |
| 2 (C) | 5, 10, 2 | Push value |
| / | 5, 5 | $10/2 = 5$ |
| + | 10 | $5 + 5 = 10$ |
| 3 (D) | 10, 3 | Push value |
| + | 13 | $10 + 3 = 13$ |

**Result:** 13

**Mnemonic**

"PC-SE - Push operands, Calculate when operators, Stack holds Everything"

## Question 3(a) [3 marks]

**Enlist applications of Linked List.**

**Solution**

**Table 9.** Applications of Linked List

| Application | Why Linked List is Used |
|---|---|
| **Dynamic Memory Allocation** | Efficient insertion/deletion without reallocation |
| **Implementing Stacks & Queues** | Can grow and shrink as needed |
| **Undo Functionality** | Easy to add/remove operations from history |
| **Hash Tables** | For handling collisions via chaining |
| **Music Playlists** | Easy navigation between songs (next/previous) |

**Mnemonic**

"DSUHM - Dynamic allocation, Stacks & queues, Undo, Hash tables, Music players"

# Question 3(b) [4 marks]

**Explain creation of singly linked list in python.**

**Solution**

Creating a singly linked list in Python involves defining a Node class and implementing basic operations.

**Listing 4.** Creating Linked List

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

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

    def append(self, data):
        new_node = Node(data)
        # If empty list, set new node as head
        if self.head is None:
            self.head = new_node
            return
        # Traverse to the end and add node
        last = self.head
        while last.next:
            last = last.next
        last.next = new_node
```

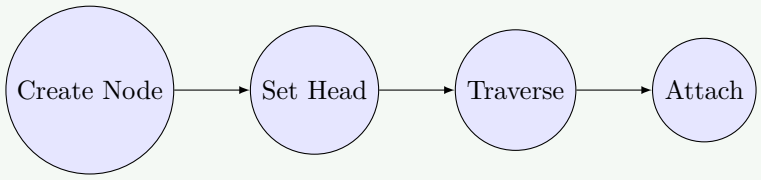



**Figure 6.** Creation Process

**Mnemonic**

"CHEN - Create nodes, Head first, End attachment, Next pointers"

# Question 3(c) [7 marks]

**Write a code to insert a new node at the beginning and end of singly linked list.**

**Solution**

**Listing 5.** Insertion Operations

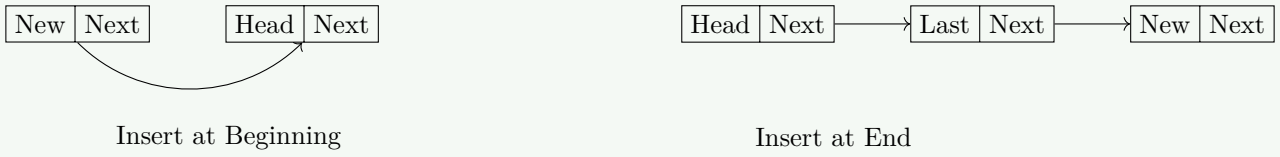
```python
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
```

```
        self.head = None

    # Insert at beginning (prepend)
    def insert_at_beginning(self, data):
        new_node = Node(data)
        new_node.next = self.head
        self.head = new_node

    # Insert at end (append)
    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
```




Insert at Beginning

Insert at End

**Figure 7.** Insertion Visualization

**Mnemonic**

"BEN - Beginning is Easy and Next-based, End Needs traversal"

## Question 3(a) OR [3 marks]

**Write a code to count the number of nodes in singly linked list.**

**Solution**

**Listing 6.** Count Nodes

```
def count_nodes(self):
    count = 0
    current = self.head

    # Traverse the list and count nodes
    while current:
        count += 1
        current = current.next

    return count
```

**Mnemonic**

"CIT - Count Incrementally while Traversing"

# Question 3(b) OR [4 marks]

**Match appropriate options from column A and B**

**Solution**

**Table 10.** Match Columns

| Column A | Column B | Match |
|---|---|---|
| 1. Singly Linked List | c. Nodes contain data and reference to next | 1-c |
| 2. Doubly Linked List | d. Nodes contain data, next and prev | 2-d |
| 3. Circular Linked List | b. Last node points to first node | 3-b |
| 4. Node | a. Basic unit containing data and references | 4-a |

Singly: $A \rightarrow B \rightarrow NULL$

Doubly: $A \leftrightarrow B \leftrightarrow NULL$

Circular: $A \rightarrow B \rightarrow A$

**Figure 8.** Linked List Types

**Mnemonic**

"SDCN - Single-Direction, Double-Direction, Circular-Connection, Node-Component"

# Question 3(c) OR [7 marks]

**Explain deletion of first and last node in singly linked list.**

**Solution**

Deletion complexity depends on position.

**Table 11.** Deletion Comparison

| Position | Approach | Complexity |
|---|---|---|
| First Node | Change head pointer | O(1) |
| Last Node | Traverse to second-last node | O(n) |

**Listing 7.** Deletion Code

```
def delete_first(self):
    if self.head is None: return
    self.head = self.head.next

def delete_last(self):
    if self.head is None: return
    if self.head.next is None:
        self.head = None
        return
    current = self.head
    while current.next.next:
        current = current.next
    current.next = None
```

**Mnemonic**

"FELO - First is Easy, Last needs One-before-last"

## Question 4(a) [3 marks]

**Explain concept of doubly linked list.**

**Solution**

A doubly linked list is a bidirectional linear data structure.

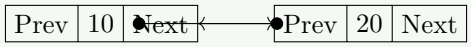



**Figure 9.** Doubly Linked List

**Mnemonic**

"PDN - Previous, Data, Next"

## Question 4(b) [4 marks]

**Explain concept of linear search.**

**Solution**

Linear search sequentially checks each element.

**Table 12.** Linear Search

| Aspect | Description |
|---|---|
| **Working** | Check each element start to end |
| **Time Complexity** | O(n) worst/average |
| **Best Case** | O(1) |

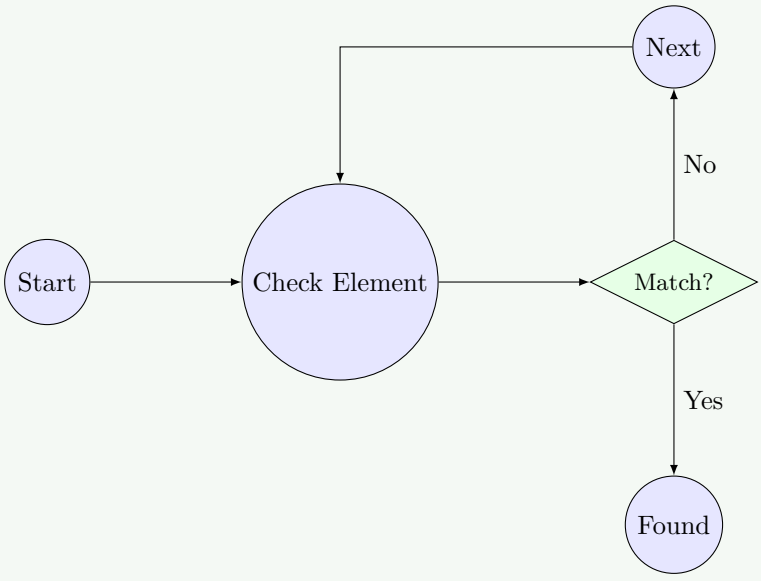



**Figure 10.** Linear Search Flow

**Mnemonic**

"SCENT - Search Consecutively Each element until Target"

## Question 4(c) [7 marks]

**Write a code to implement binary search algorithm.**

**Solution**

Binary search divides the search interval in half.

**Listing 8.** Binary Search

```
def binary_search(arr, target):
    left = 0
    right = len(arr) - 1

    while left <= right:
        mid = (left + right) // 2

        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1

    return -1
```

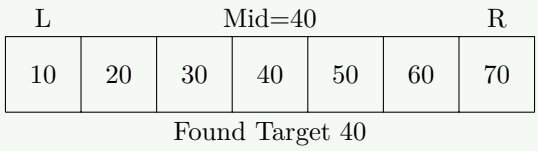



**Figure 11.** Binary Search Execution

**Mnemonic**

"MCLR - Middle Compare, Left or Right adjust"

## Question 4(a) OR [3 marks]

**Explain concept of selection sort algorithm.**

**Solution**

Selection sort finds the minimum element and places it at the beginning.

- **Time Complexity**: $O(n^2)$
- **Space Complexity**: $O(1)$
- **Method**: Find smallest, swap with current position.

**Mnemonic**

"FSMR - Find Smallest, Move to Right position, Repeat"

# Question 4(b) OR [4 marks]

**Explain bubble sort method.**

**Solution**

Bubble sort repeatedly swaps adjacent elements if they are in the wrong order.

**Table 13.** Bubble Sort

| | |
|---|---|
| **Complexity** | $O(n^2)$ |
| **Passes** | $n-1$ passes |
| **Type** | In-place, Stable |

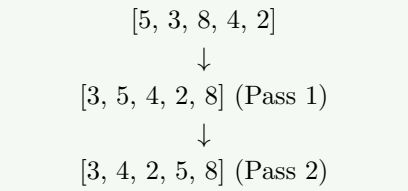



**Figure 12.** Bubble Sort Passes

**Mnemonic**

"CABS - Compare Adjacent, Bubble-up Swapping"

# Question 4(c) OR [7 marks]

**Explain the working of quick sort method with example.**

**Solution**

Quick sort selects a pivot and partitions the array.

**Table 14.** Quick Sort Steps

| | |
|---|---|
| 1 | Choose Pivot |
| 2 | Partition: Small go Left, Large go Right |
| 3 | Recursively sort Left and Right subarrays |

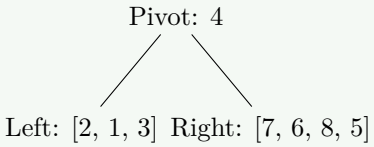



**Figure 13.** Quick Sort Partition

**Mnemonic**

"PPR - Pivot, Partition, Recursive divide"

## Question 5(a) [3 marks]

**Explain binary tree.**

**Solution**

A binary tree is a hierarchical data structure where each node has at most two children.

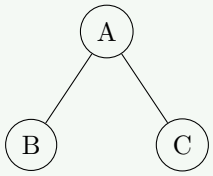



**Figure 14.** Binary Tree

**Mnemonic**

"RLTM - Root, Left, Two, Maximum"

## Question 5(b) [4 marks]

**Define the terms root, path, parent and children with reference to tree.**

**Solution**

**Table 15.** Tree Terminology

| Term | Definition |
|---|---|
| **Root** | Topmost node with no parent |
| **Path** | Sequence of connected nodes |
| **Parent** | Node with one or more children |
| **Children** | Nodes connected to a parent |

**Mnemonic**

"RPPC - Root at Top, Path connects, Parent above, Children below"

## Question 5(c) [7 marks]

**Apply preorder and postorder traversal for given below tree.**

**Solution**

Tree: 40(Root), Left:30, Right:50...

**Table 16.** Traversals

| Traversal | Order | Result |
|---|---|---|
| **Preorder** | Root, Left, Right | 40, 30, 25, 15, 28, 35, 50, 45, 60, 55, 70 |
| **Postorder** | Left, Right, Root | 15, 28, 25, 35, 30, 45, 55, 70, 60, 50, 40 |

**Mnemonic**

"PRE-NLR, POST-LRN"

## Question 5(a) OR [3 marks]

**Enlist applications of binary tree.**

**Solution**

- Binary Search Trees (Efficient searching)
- Expression Trees (Compilers)
- Huffman Coding (Compression)
- Priority Queues (Heaps)
- Decision Trees (ML)

**Mnemonic**

"BEHPD"

## Question 5(b) OR [4 marks]

**Explain insertion of a node in binary search tree.**

**Solution**

BST Property: Left < Node < Right.

1. Start at root.
2. If New < Current, go Left.
3. If New > Current, go Right.
4. Insert at empty spot.

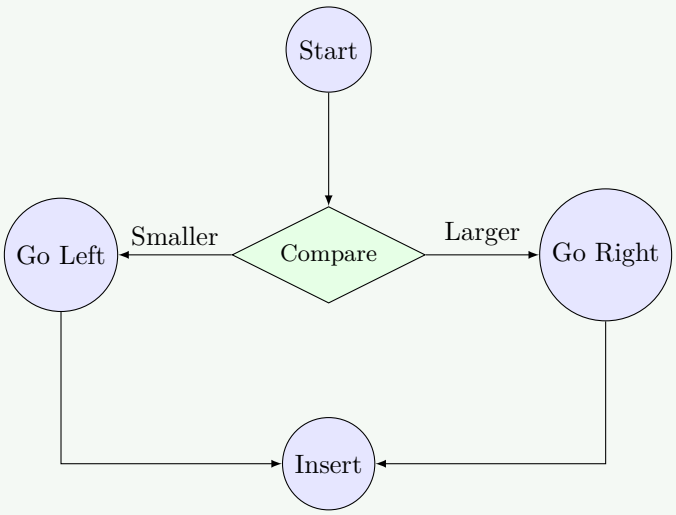



**Figure 15.** BST Insertion

**Mnemonic**

"LSRG"

# Question 5(c) OR [7 marks]

**Draw Binary search tree for 8, 4, 12, 2, 6, 10, 14, 1, 3, 5 and write In-order traversal for the tree.**

**Solution**

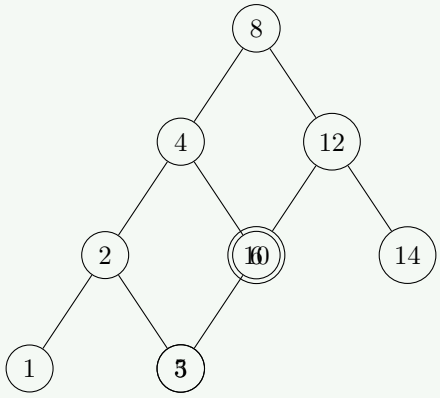



**Figure 16.** Constructed BST

**In-order Traversal (Left, Root, Right):**
1, 2, 3, 4, 5, 6, 8, 10, 12, 14

**Mnemonic**

"LNR - Left, Node, Right"