Data Structure with Python (4331601) - Winter 2024 Solution

Milav Dabgar

December 03, 2024

Question 1(a) [3 marks]

Explain set data structure in python?

Solution

A **set** is an unordered collection of unique elements in Python. Sets are mutable but contain only immutable elements.

Key Properties:

Table 1. Set Properties

Property	Description
Unique Elements	No duplicate values allowed
Unordered	No indexing or slicing
Mutable	Can add/remove elements
Iterable	Can loop through elements

Basic Operations:

Listing 1. Set Operations

```
# Create set
my_set = {1, 2, 3, 4}
# Add element
my_set.add(5)
# Remove element
my_set.remove(2)

```

Mnemonic

"Sets are Unique Unordered Collections"

Question 1(b) [4 marks]

Define Tuple in python? Explain operations of tuple data structure in python.

Solution

A **tuple** is an ordered collection of items that is immutable (cannot be changed after creation).

Tuple Definition:

- **Ordered**: Elements have defined order
- **Immutable**: Cannot modify after creation

- **Allow duplicates**: Same values can appear multiple times
- **Indexed**: Access elements using index

Tuple Operations:

Table 2. Tuple Operations

Operation	Example	Description
Creation	`t = (1, 2, 3)`	Create tuple
Indexing	`t[0]`	Access first element
Slicing	`t[1:3]`	Get subset
Length	`len(t)`	Count elements
Concatenation	`t1 + t2`	Join tuples

Listing 2. Tuple Examples

```
# Example operations
tup = (10, 20, 30, 40)
print(tup[1])      # Output: 20
print(tup[1:3])    # Output: (20, 30)

```

Mnemonic

"Tuples are Immutable Ordered Collections"

Question 1(c) [7 marks]

Explain Types of constructors in python? Write a python program to multiplication of two numbers using static method.

Solution

Types of Constructors:

Table 3. Constructor Types

Constructor Type	Description	Usage
Default Constructor	No parameters	`__init__(self)`
Parameterized Constructor	Takes parameters	`__init__(self, params)`
Non-parameterized Constructor	Only self parameter	Basic initialization

Static Method Program:

Listing 3. Static Method for Multiplication

```
class Calculator:
    def __init__(self):
        pass

    @staticmethod
    def multiply(num1, num2):
        return num1 * num2

# Usage
result = Calculator.multiply(5, 3)
print(f"Multiplication: {result}")  # Output: 15

```

Key Points:
- **Static methods**: Don't need object instance
- **staticmethod decorator**: Defines static method
- **No self parameter**: Independent of class instance

Mnemonic

"Static methods Stand Separate from Self"

Question 1(c) OR [7 marks]

Define Data Encapsulation. List out different types of methods in python. Write a python program to multilevel inheritances.

Solution

Data Encapsulation: Data encapsulation is the concept of bundling data and methods within a class and restricting direct access to some components.

Types of Methods:

Table 4. Method Types

Method Type	Access Level	Example
Public	Accessible everywhere	`method()`
Protected	Class and subclass	`_method()`
Private	Only within class	`__method()`
Static	Class level	`staticmethod`
Class	Class and subclasses	`classmethod`

Multilevel Inheritance Program:

Listing 4. Multilevel Inheritance

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

    def speak(self):
        print(f"{self.name} makes sound")

class Mammal(Animal):
    def __init__(self, name, warm_blooded):
        super().__init__(name)
        self.warm_blooded = warm_blooded

class Dog(Mammal):
    def __init__(self, name, breed):
        super().__init__(name, True)
        self.breed = breed

    def bark(self):
        print(f"{self.name} barks")

# Usage
dog = Dog("Buddy", "Golden Retriever")
dog.speak()  # From Animal class
dog.bark()   # From Dog class

```

Mnemonic

"Encapsulation Hides Internal Details"

Question 2(a) [3 marks]

Differentiate between simple queue and circular queue.

Solution

Table 5. Simple vs Circular Queue

Feature	Simple Queue	Circular Queue
Structure	Linear arrangement	Circular arrangement
Memory Usage	Wasteful (empty spaces)	Efficient (reuses space)
Rear Pointer	Moves linearly	Wraps around
Front Pointer	Moves linearly	Wraps around
Space Utilization	Poor	Excellent

Key Differences:

- **Simple Queue**: Front and rear move in one direction only
- **Circular Queue**: Rear connects back to front position
- **Efficiency**: Circular queue eliminates memory waste

Mnemonic

"Circular Queues Complete the Circle"

Question 2(b) [4 marks]

Explain polymorphism in python with example.

Solution

Polymorphism means "many forms" - same method name behaves differently in different classes.

Types of Polymorphism:

Table 6. Polymorphism Types

Type	Description	Implementation
Method Overriding	Child class redefines parent method	Inheritance
Duck Typing	Same method in different classes	Interface similarity
Operator Overloading	Same operator different behavior	Magic methods

Listing 5. Polymorphism Example

```
class Animal:
    def make_sound(self):
        pass

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

```
class Cat(Animal):
    def make_sound(self):
        return "Meow!"

# Polymorphic behavior
animals = [Dog(), Cat()]
for animal in animals:
    print(animal.make_sound())
```

Mnemonic

"Polymorphism Provides Multiple Personalities"

Question 2(c) [7 marks]

Define a).Infix b).postfix. Given equation to conversion from infix to postfix using stack. A+(B*C/D)

Solution

Definitions:

Table 7. Expression Types

Expression Type	Description	Example
Infix	Operator between operands	`A + B`
Postfix	Operator after operands	`A B +`

Conversion Algorithm:

1. Scan infix expression left to right.
2. If operand, add to output.
3. If operator, compare precedence with stack top.
4. Higher precedence → push to stack.
5. Lower/equal precedence → pop and add to output.

Step-by-step Conversion: A+(B*C/D)

Table 8. Infix to Postfix Trace

Step	Symbol	Stack	Output
1	A	[ ]	A
2	+	[+]	A
3	(	[+, (]	A
4	B	[+, (]	AB
5	*	[+, (, *]	AB
6	C	[+, (, *]	ABC
7	/	[+, (, /]	ABC*
8	D	[+, (, /]	ABC*D
9	)	[+]	ABC*D/
10	End	[ ]	ABC*D/+

Final Answer: `ABC*D/+`

Mnemonic

"Stack Stores Operators Strategically"

Question 2(a) OR [3 marks]

Explain disadvantages of Queue.

Solution

Table 9. Queue Disadvantages

Disadvantage	Description	Impact
Memory Waste	Empty spaces not reused	Poor space utilization
Fixed Size	Limited capacity	Overflow issues
No Random Access	Only front/rear access	Limited flexibility

Key Issues:

- **Linear Queue**: Front spaces become unusable
- **Insertion/Deletion**: Only at specific ends
- **Search Operations**: Not efficient for searching

Mnemonic

"Queues Quietly Queue with Quirks"

Question 2(b) OR [4 marks]

Define Abstract class in python? Explain the declaration of abstract method in python?

Solution

Abstract Class: A class that cannot be instantiated and contains one or more abstract methods that must be implemented by subclasses.

Abstract Method Declaration:

Table 10. Abstract Methods

Component	Purpose	Syntax
ABC Module	Provides abstract base class	`from abc import ABC`
@abstractmethod	Decorator for abstract methods	`@abstractmethod`
Implementation	Must override in subclass	Required

Listing 6. Abstract Class Example

```
from abc import ABC, abstractmethod

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

    @abstractmethod
    def perimeter(self):
```

```
        pass

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

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

    def perimeter(self):
        return 2 * (self.length + self.width)

```

Mnemonic

"Abstract classes Are Blueprints Only"

Question 2(c) OR [7 marks]

Write an algorithm for Infix to postfix expression. Evaluate Postfix expression as: 5 6 2 + * 12 4 / -

Solution

Infix to Postfix Algorithm:

1. Initialize empty stack and output string.
2. Scan infix expression from left to right.
3. If operand → add to output.
4. If '(' → push to stack.
5. If ')' → pop until '('.
6. If operator → pop higher/equal precedence operators.
7. Push current operator to stack.
8. Pop remaining operators.

Postfix Evaluation: 5 6 2 + * 12 4 / -

Table 11. Postfix Evaluation Trace

Step	Token	Stack	Operation
1	5	[5]	Push operand
2	6	[5, 6]	Push operand
3	2	[5, 6, 2]	Push operand
4	+	[5, 8]	Pop 2, 6 → $6+2=8$
5	*	[40]	Pop 8, 5 → $5*8=40$
6	12	[40, 12]	Push operand
7	4	[40, 12, 4]	Push operand
8	/	[40, 3]	Pop 4, 12 → $12/4=3$
9	-	[37]	Pop 3, 40 → $40-3=37$

Final Result: 37

Mnemonic

"Postfix Processing Pops Pairs Precisely"

Question 3(a) [3 marks]

Write an algorithm to traverse node in single linked list.

Solution

Traversal Algorithm:

Listing 7. Linked List Traversal

```
def traverse_linked_list(head):
    current = head
    while current is not None:
        print(current.data)
        current = current.next

```

Algorithm Steps:

1. Start from head node.
2. Check if current $\neq$ NULL.
3. Process current node.
4. Move to next node.
5. Repeat until end.

Mnemonic

"Traverse Through Till The Tail"

Question 3(b) [4 marks]

Write an algorithm for Dequeue operation in queue using List.

Solution

Dequeue Algorithm:

Listing 8. Dequeue Operation

```
def dequeue(queue):
    if len(queue) == 0:
        print("Queue is empty")
        return None
    else:
        element = queue.pop(0)
        return element

```

Algorithm Steps:

1. Check empty: If queue is empty.
2. Handle underflow: Display error message.
3. Remove element: Delete front element.
4. Return element: Return removed value.
5. Update structure: Adjust queue pointers.

Time Complexity: $O(n)$ due to list shifting

Mnemonic

"Dequeue Deletes from Front Door"

Question 3(c) [7 marks]

Define double linked list. Enlist major operation of Linked List. Write an algorithm to insert a node at beginning in singly linked list.

Solution

Double Linked List: A linear data structure where each node contains data and two pointers - one pointing to the next node and another to the previous node.

Major Linked List Operations:

Table 12. Linked List Operations

Operation	Description	Time Complexity
Insertion	Add new node	$O(1)$ at beginning
Deletion	Remove node	$O(1)$ if node known
Traversal	Visit all nodes	$O(n)$
Search	Find specific node	$O(n)$
Update	Modify node data	$O(1)$ if node known

Insert at Beginning Algorithm:

Listing 9. Insert at Beginning

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

def insert_at_beginning(head, data):
    new_node = Node(data)
    new_node.next = head
    head = new_node
    return head

```

Algorithm Steps:

1. Create new node with given data.
2. Set new node's next to current head.
3. Update head to point to new node.
4. Return new head.

Mnemonic

"Insert at Beginning Builds Better Lists"

Question 3(a) OR [3 marks]

Explain the applications of single linked list.

Solution

Single Linked List Applications:

Table 13. Applications

Application	Use Case	Advantage
Dynamic Memory	Variable size data	Memory efficient
Stack Implementation	LIFO operations	Easy push/pop
Queue Implementation	FIFO operations	Dynamic sizing
Music Playlist	Sequential playback	Easy navigation
Browser History	Page navigation	Forward traversal
Polynomial Representation	Mathematical operations	Coefficient storage

Key Benefits:
- **Dynamic Size**: Grows/shrinks during runtime
- **Memory Efficiency**: Allocates as needed
- **Insertion/Deletion**: Efficient at any position

Mnemonic

"Linked Lists Link Many Applications"

Question 3(b) OR [4 marks]

Write an algorithm for PUSH operation of stack using List.

Solution

PUSH Algorithm:

Listing 10. Stack Push

```
def push(stack, element):
    stack.append(element)
    print(f"Pushed {element} to stack")

```

Algorithm Steps:
1. **Check capacity**: Verify stack not full (for fixed size).
2. **Add element**: Append to end of list.
3. **Update top**: Top points to last element.
4. **Confirm**: Display success message.

Time Complexity: O(1)

Mnemonic

"PUSH Puts on Stack Summit"

Question 3(c) OR [7 marks]

Explain advantages of a linked list. Write an algorithm to delete node at last from single linked list.

Solution

Linked List Advantages:

Table 14. Advantages

Advantage	Description	Benefit
Dynamic Size	Size changes at runtime	Memory flexible
Memory Efficient	Allocates as needed	No waste
Easy Insertion	Add anywhere efficiently	O(1) operation
Easy Deletion	Remove efficiently	O(1) operation
No Memory Shift	Elements don't move	Fast operations

Delete Last Node Algorithm:

Listing 11. Delete Last Node

```
def delete_last_node(head):
    # Empty list
    if head is None:
        return None

    # Single node
    if head.next is None:
        return None

    # Multiple nodes
    current = head
    while current.next.next is not None:
        current = current.next

    current.next = None
    return head

```

Mnemonic

"Linked Lists Lead to Logical Advantages"

Question 4(a) [3 marks]

Write an algorithm of Bubble sort.

Solution

Bubble Sort Algorithm:

Listing 12. Bubble Sort

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
    return arr

```

Time Complexity: $O(n^2)$

Mnemonic

"Bubbles Rise to Surface Slowly"

Question 4(b) [4 marks]

Explain circular linked list with its advantages.

Solution

Circular Linked List: A linked list where the last node points to the first node, forming a circular structure.
Advantages:

- **Memory Efficient**: No NULL pointers
- **Circular Traversal**: Can loop continuously
- **Queue Implementation**: Efficient enqueue/dequeue
- **Round Robin Scheduling**: CPU time sharing

Figure 1. Circular Linked List

Mnemonic

"Circular Lists Create Continuous Connections"

Question 4(c) [7 marks]

Explain merge sort with suitable example.

Solution

Merge Sort is a divide-and-conquer algorithm that divides array into halves, sorts them separately, and merges back.
Algorithm Phases:

Table 15. Merge Sort Phases

Phase	Action	Description
Divide	Split array	Divide into two halves
Conquer	Sort subarrays	Recursively sort halves
Combine	Merge results	Merge sorted halves

Example: [38, 27, 43, 3, 9, 82, 10]

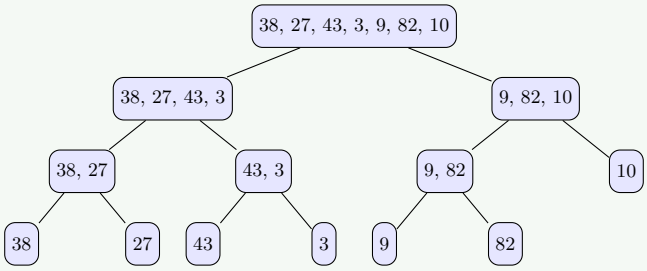

Figure 2. Merge Sort Division

Merged Result: `[3, 9, 10, 27, 38, 43, 82]`
Time Complexity: O(n log n)

Mnemonic

"Merge Sort Methodically Merges Segments"

Question 4(a) OR [3 marks]

Write an algorithm for selection sort.

Solution

Selection Sort Algorithm:

Listing 13. Selection Sort

```
def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_idx = i
        for j in range(i+1, n):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
    return arr

```

Time Complexity: $O(n^2)$

Mnemonic

"Selection Sort Selects Smallest Successfully"

Question 4(b) OR [4 marks]

Explain double linked list with its advantages.

Solution

Double Linked List: A linked list where each node contains data and two pointers - next and previous.
Advantages:

- **Bidirectional Traversal**: Forward and backward movement
- **Easy Deletion**: Can delete without knowing previous node
- **Efficient Insertion**: Insert at any position easily

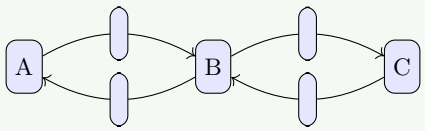

Figure 3. Double Linked List

Mnemonic

"Double Links provide Dual Direction"

Question 4(c) OR [7 marks]

Explain insertion sort. Give trace of following numbers using insertion sort : 25, 15,30,9,99,20,26

Solution

Insertion Sort: Builds sorted array one element at a time by inserting each element into its correct position.
Trace Evaluation:

Table 16. Insertion Sort Trace

Pass	Current	Array State	Action
Initial	-	[25, 15, 30, 9, 99, 20, 26]	Start
1	15	[15, 25, 30, 9, 99, 20, 26]	Insert 15
2	30	[15, 25, 30, 9, 99, 20, 26]	No change
3	9	[9, 15, 25, 30, 99, 20, 26]	Insert 9
4	99	[9, 15, 25, 30, 99, 20, 26]	No change
5	20	[9, 15, 20, 25, 30, 99, 26]	Insert 20
6	26	[9, 15, 20, 25, 26, 30, 99]	Insert 26

Final Sorted Array: `[9, 15, 20, 25, 26, 30, 99]`

Mnemonic

"Insertion Inserts Into Increasing Order"

Question 5(a) [3 marks]

Explain application of binary tree.

Solution

Binary Tree Applications:

Table 17. Applications

Application	Use Case	Benefit
Expression Trees	Mathematical expressions	Easy evaluation
Binary Search Trees	Searching/Sorting	O(log n) operations
Heap Trees	Priority queues	Efficient min/max
File Systems	Directory structure	Hierarchical organization

Mnemonic

"Binary Trees Branch into Many Applications"

Question 5(b) [4 marks]

Write an algorithm for binary search using list.

Solution

Binary Search Algorithm:

Listing 14. Binary Search

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

```

Prerequisite: Array must be sorted. **Time Complexity: O(log n)**

Mnemonic

"Binary Search Bisects to Find Faster"

Question 5(c) [7 marks]

Define Tree. Enlist Types of Tree. Write an algorithm to insert node in binary search tree using python.

Solution

Tree: A hierarchical data structure consisting of nodes connected by edges, with one root node and no cycles.
Types of Trees:

- **Binary Tree**: Max 2 children per node.
- **Binary Search Tree**: Ordered binary tree (Left < Root < Right).
- **Complete Binary Tree**: All levels filled except last.
- **Full Binary Tree**: All nodes have 0 or 2 children.
- **AVL Tree**: Self-balancing BST.

BST Insertion Algorithm:

Listing 15. BST Insertion

```
class TreeNode:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

def insert_bst(root, data):
    if root is None:
        return TreeNode(data)
```

```
    if data < root.data:
        root.left = insert_bst(root.left, data)
    elif data > root.data:
        root.right = insert_bst(root.right, data)

    return root

```

Mnemonic

"Trees Grow with Structured Organization"

Question 5(a) OR [3 marks]

Write an algorithm for in-order traversal of tree.

Solution

In-order Traversal Algorithm:

Listing 16. In-order Traversal

```
def inorder_traversal(root):
    if root is not None:
        inorder_traversal(root.left)    # Left
        print(root.data)                # Root
        inorder_traversal(root.right)   # Right

```

Traversal Order: Left → Root → Right

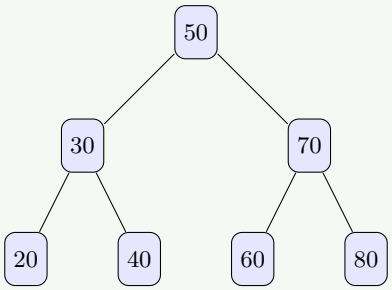

Figure 4. Example Tree

In-order: 20, 30, 40, 50, 60, 70, 80

Mnemonic

"In-order: Left, Root, Right"

Question 5(b) OR [4 marks]

Define search? Write an algorithm for Linear search using list.

Solution

Search: The process of finding a specific element or checking if an element exists in a data structure.
Linear Search Algorithm:

Listing 17. Linear Search

```
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i  # Return index if found
    return -1  # Return -1 if not found

```

Time Complexity: O(n)

Mnemonic

"Linear Search Looks through Lists Linearly"

Question 5(c) OR [7 marks]

Define: a) Path b). Leaf Node. Construct a binary search tree for following data items. 60, 40, 37,31,59,21,65,30

Solution

Definitions:
- **Path**: Sequence of nodes from one node to another connected by edges.
- **Leaf Node**: Node with no children (no left or right child).

BST Construction for: 60, 40, 37, 31, 59, 21, 65, 30

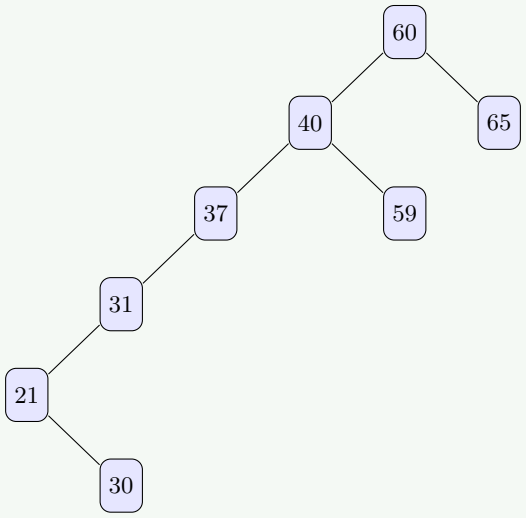

Figure 5. Final Binary Search Tree

Leaf Nodes: 30, 59, 65

Mnemonic

"BST Building follows Binary Search Tree rules"