

# Subject Name Solutions

1333203 – Summer 2025

Semester 1 Study Material

*Detailed Solutions and Explanations*

## Question 1(a) [3 marks]

Define Big - O Notation, Big Omega Notation, Big Theta Notation.

### Solution

Table 1: Asymptotic Notations Comparison

Notation	Symbol	Description	Usage
Big-O	$O(f(n))$	Upper bound	Worst case
Big Omega	$\Omega(f(n))$	Lower bound	Best case
Big Theta	$\Theta(f(n))$	Tight bound	Average case

- **Big-O Notation:** Describes maximum time/space complexity
- **Big Omega:** Describes minimum time/space complexity
- **Big Theta:** Describes exact time/space complexity

### Mnemonic

“OWT - O for wOrst, Omega for Best, Theta for Tight”

## Question 1(b) [4 marks]

Define Set. Write various operations that can be performed on Set.

### Solution

**Definition:** Set is a collection of unique elements with no duplicates.

Table 2: Set Operations

Operation	Symbol	Description	Example
Union	$A \cup B$	Combines all elements	$\{1,2\} \cup \{2,3\} = \{1,2,3\}$
Intersection	$A \cap B$	Common elements	$\{1,2\} \cap \{2,3\} = \{2\}$
Difference	$A - B$	Elements in A not in B	$\{1,2\} - \{2,3\} = \{1\}$
Subset	$A \subseteq B$	All A elements in B	$\{1\} \subseteq \{1,2\} = \text{True}$

- **Add/Insert:** Adding new element
- **Remove/Delete:** Removing existing element
- **Contains:** Check if element exists

### Mnemonic

“UIDS - Union, Intersection, Difference, Subset”

## Question 1(c) [7 marks]

Write a Python class to represent a Cricketer. The class contains the name of the cricketer, team name and run as the data members. The member functions are as follows: to initialize the data members, to set run and display run.

## Solution

```
1 class Cricketer:
2     def __init__(self, name="", team="", run=0):
3         self.name = name
4         self.team = team
5         self.run = run
6
7     def set_run(self, run):
8         self.run = run
9
10    def display_run(self):
11        print(f"Player: {self.name}")
12        print(f"Team: {self.team}")
13        print(f"Runs: {self.run}")
14
15    # Example usage
16    player = Cricketer("Virat Kohli", "India", 100)
17    player.display_run()
```

- **Constructor:** Initializes name, team, and run
- **set\_run():** Updates run value
- **display\_run():** Shows player information

## Mnemonic

“CSD - Constructor, Set, Display”

## Question 1(c OR) [7 marks]

Design a student class for reading and displaying the student information, the getInfo() and displayInfo() methods will be used respectively. Where getInfo() will be a private method.

## Solution

```
1 class Student:
2     def __init__(self):
3         self.name = ""
4         self.roll_no = ""
5         self.marks = 0
6         self.__getInfo() # Private method call
7
8     def __getInfo__(self): # Private method
9         self.name = input("Enter name: ")
10        self.roll_no = input("Enter roll number: ")
11        self.marks = int(input("Enter marks: "))
12
13    def displayInfo(self):
14        print(f"Name: {self.name}")
15        print(f"Roll No: {self.roll_no}")
16        print(f"Marks: {self.marks}")
17
18    # Example usage
19    student = Student()
20    student.displayInfo()
```

- **Private method:** Uses double underscore (\_\_getInfo)
- **Constructor:** Automatically calls private method
- **Public method:** displayInfo() shows student data

## Mnemonic

“PCP - Private, Constructor, Public”

### Question 2(a) [3 marks]

Differentiate between Stack and Queue.

#### Solution

Table 3: Stack vs Queue Comparison

Feature	Stack	Queue
Order	LIFO (Last In First Out)	FIFO (First In First Out)
Operations	Push, Pop	Enqueue, Dequeue
Access Point	One end (top)	Two ends (front & rear)
Example	Plates stack	Bank queue

- **Stack:** Like book pile - last added, first removed
- **Queue:** Like waiting line - first come, first served

#### Mnemonic

“SLIF QFIF - Stack LIFO, Queue FIFO”

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

Define recursion. Explain with example.

#### Solution

**Definition:** Function calling itself with smaller problem until base condition.

```
1 def factorial(n):
2     # Base case
3     if n <= 1:
4         return 1
5     # Recursive case
6     return n * factorial(n-1)
7
8 # Example: factorial(3)
9 # 3 * factorial(2)
10 # 3 * 2 * factorial(1)
11 # 3 * 2 * 1 = 6
```

- **Base case:** Stopping condition
- **Recursive case:** Function calls itself
- **Problem reduction:** Each call handles smaller problem

#### Mnemonic

“BRP - Base, Recursive, Problem-reduction”

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

Consider the size of the stack as 5. Apply the following operation on stack and show status and top pointer after each operation. Push a,b,c pop

#### Solution

**Stack Operations Trace:**

```
1 Initial State:
2 Stack: [ _ _ _ _ ] Top: -1
3       0 1 2 3 4
4
5 After Push 'a':
6 Stack: [ a _ _ _ ] Top: 0
```

```

7      0 1 2 3 4
8
9  After Push 'b':
10 Stack: [ a b _ _ _ ] Top: 1
11      0 1 2 3 4
12
13 After Push 'c':
14 Stack: [ a b c _ _ ] Top: 2
15      0 1 2 3 4
16
17 After Pop:
18 Stack: [ a b _ _ _ ] Top: 1
19      0 1 2 3 4
20 Popped element: c

```

- **Push operations:** Add elements from index 0 onwards
- **Top pointer:** Points to last inserted element
- **Pop operation:** Removes top element, decrements top pointer

#### Mnemonic

“PTD - Push Top Decrement”

### Question 2(a OR) [3 marks]

List applications of Stack and Queue.

#### Solution

Table 4: Applications of Stack and Queue

Data Structure	Applications
<b>Stack</b>	Function calls, Undo operations, Expression evaluation, Browser history
<b>Queue</b>	Process scheduling, Printer queue, BFS traversal, Handling requests

- **Stack applications:** Undo-redo, recursion, parsing
- **Queue applications:** Task scheduling, buffering, breadth-first search

#### Mnemonic

“Stack FUBE, Queue SPBH”

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

\*\*Convert following algebraic expression into postfix notation using Stack: i)  $(ab)(c^d(d+e)-f)$  ii)  $a-b/(c*d/e)**$

#### Solution

i)  $(ab)(c^d(d+e)-f)$

Symbol	Stack	Output
(	(	
a	(	a
*	(*	a
b	(*	ab
)		ab*
*	*	ab*
(	*(	ab*

c	*	(	ab*c
^	*	(^	ab*c
d	*	(^	ab*cd
(	*	((^	ab*cd
d	*	((^	ab*cdd
+	*	(^(+	ab*cdd
e	*	(^(+	ab*cdde
)	*	(^	ab*cdde+
)	*		ab*cdde+^
-	*	-	ab*cdde+^
f	*	-	ab*cdde+^f
			abcdde+^f-

**Result:**  $abcdde+^f-$

**\*\*ii)**  $a-b/(c*d/e)$

**\*\*Result:**  $abcd*e/-**$

### Mnemonic

“PEMDAS reversed for postfix”

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

Develop a program to implement a queue using a list that performs following operations: enqueue, dequeue.

### Solution

```

1 class Queue:
2     def __init__(self):
3         self.queue = []
4         self.front = 0
5         self.rear = -1
6
7     def enqueue(self, item):
8         self.queue.append(item)
9         self.rear += 1
10        print(f"Enqueued: {item}")
11
12    def dequeue(self):
13        if self.front <= self.rear:
14            item = self.queue[self.front]
15            self.front += 1
16            print(f"Dequeued: {item}")
17            return item
18        else:
19            print("Queue is empty")
20            return None
21
22    def display(self):
23        if self.front <= self.rear:
24            print("Queue:", self.queue[self.front:self.rear+1])
25        else:
26            print("Queue is empty")
27
28 # Example usage
29 q = Queue()
30 q.enqueue('A')
31 q.enqueue('B')
32 q.dequeue()
33 q.display()

```

- **Enqueue:** Add element at rear

- **Dequeue:** Remove element from front
- **FIFO principle:** First In, First Out

### Mnemonic

“ERF - Enqueue Rear, Front”

## Question 3(a) [3 marks]

List types of linked lists. Give graphical representation of each type.

### Solution

Table 5: Types of Linked Lists

Type	Description	Diagram
Singly	One direction pointer	A
Doubly	Two direction pointers	NULL B C
Circular	Last points to first	A

Singly Linked List:

[Data|Next] -> [Data|Next] -> [Data|NULL]

Doubly Linked List:

[Prev|Data|Next] <-> [Prev|Data|Next] <-> [Prev|Data|Next]

Circular Linked List:

[Data|Next] -> [Data|Next] -> [Data|Next]

```

  ^
  |
  |-----|
  
```

### Mnemonic

“SDC - Singly, Doubly, Circular”

## Question 3(b) [4 marks]

Write an algorithm to search a given node in a singly link list.

### Solution

```

1 def search_node(head, key):
2     current = head
3     position = 0
4
5     while current is not None:
6         if current.data == key:
7             return position
8         current = current.next
9         position += 1
10
11     return -1 # Not found
12
13 # Algorithm steps:
14 # 1. Start from head
15 # 2. Compare current data with key
16 # 3. If found, return position
17 # 4. Move to next node
18 # 5. Repeat until end
  
```

- **Linear search:** Traverse from head to tail

- **Time complexity:**  $O(n)$
- **Return:** Position if found, -1 if not found

#### Mnemonic

“SCMR - Start, Compare, Move, Return”

### Question 3(c) [7 marks]

Implement program to perform following operation on singly linked list: 1) Insert a node at the beginning of a singly linked list. 2) Delete a node from the beginning of a singly linked list.

#### Solution

```

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        new_node = Node(data)
12        new_node.next = self.head
13        self.head = new_node
14        print(f"Inserted {data} at beginning")
15
16    def delete_from_beginning(self):
17        if self.head is None:
18            print("List is empty")
19            return None
20
21        deleted_data = self.head.data
22        self.head = self.head.next
23        print(f"Deleted {deleted_data} from beginning")
24        return deleted_data
25
26    def display(self):
27        current = self.head
28        while current:
29            print(current.data, end=" -> ")
30            current = current.next
31        print("NULL")
32
33 # Example usage
34 ll = SinglyLinkedList()
35 ll.insert_at_beginning(10)
36 ll.insert_at_beginning(20)
37 ll.delete_from_beginning()
38 ll.display()

```

- **Insert:** Create node, link to head, update head
- **Delete:** Store data, move head to next, return data

#### Mnemonic

“CLU - Create, Link, Update”

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

Differentiate between circular linked list and singly linked list.

### Solution

Table 6: Circular vs Singly Linked List

Feature	Singly Linked List	Circular Linked List
Last node points to	NULL	First node (head)
Traversal	Linear (one direction)	Circular (continuous)
End detection	next == NULL	next == head
Memory	Less (no extra pointer)	Same structure

- **Circular advantage:** No NULL pointers, continuous traversal
- **Singly advantage:** Simple implementation, clear end point

### Mnemonic

“CNTE - Circular No Termination End”

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

Explain three applications of linked list in brief.

### Solution

Table 7: Linked List Applications

Application	Description	Advantage
<b>Dynamic memory allocation</b>	Manage memory blocks	Efficient memory usage
<b>Implementation of stacks/queues</b>	Using linked structure	Dynamic size
<b>Polynomial representation</b>	Store coefficients and powers	Easy arithmetic operations

- **Music playlist:** Add/remove songs dynamically
- **Browser history:** Navigate back/forward
- **Image viewer:** Previous/next image navigation

### Mnemonic

“DIP - Dynamic, Implementation, Polynomial”

### Question 3(c OR) [7 marks]

Implement a program to create and display circular linked lists.

### Solution

```
1 class Node:
2     def __init__(self, data):
3         self.data = data
4         self.next = None
5
6 class CircularLinkedList:
7     def __init__(self):
8         self.head = None
9
10    def insert(self, data):
11        new_node = Node(data)
12
13        if self.head is None:
14            self.head = new_node
15            new_node.next = self.head
16        else:
```



```

7         current = self.head
8         while current.next != self.head:
9             current = current.next
10        current.next = new_node
11        new_node.next = self.head
12
13    def display(self):
14        if self.head is None:
15            print("List is empty")
16            return
17
18        current = self.head
19        print("Circular List:")
20        while True:
21            print(current.data, end=" -> ")
22            current = current.next
23            if current == self.head:
24                break
25        print(f"{self.head.data} (back to head)")
26
27    # Example usage
28    cll = CircularLinkedList()
29    cll.insert(10)
30    cll.insert(20)
31    cll.insert(30)
32    cll.display()

```

- **Creation:** Link last node to head
- **Display:** Stop when reaching head again

### Mnemonic

“CLH - Create, Link, Head”

## Question 4(a) [3 marks]

Write a program for Selection Sort Method.

### Solution

```

1  def selection_sort(arr):
2      n = len(arr)
3
4      for i in range(n):
5          min_idx = i
6          for j in range(i+1, n):
7              if arr[j] < arr[min_idx]:
8                  min_idx = j
9
10         arr[i], arr[min_idx] = arr[min_idx], arr[i]
11
12     return arr
13
14 # Example usage
15 data = [64, 34, 25, 12, 22]
16 sorted_data = selection_sort(data)
17 print("Sorted array:", sorted_data)

```

- **Find minimum:** In unsorted portion
- **Swap:** With first unsorted element
- **Time complexity:**  $O(n^2)$

### Mnemonic

“FMS - Find, Minimum, Swap”

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

Apply Insertion sort to following data to arrange them in ascending order. 25 15 35 20 30 5 10

#### Solution

##### Insertion Sort Steps:

```
1 Initial: [25, 15, 35, 20, 30, 5, 10]
2
3 Pass 1: [15, 25, 35, 20, 30, 5, 10] (Insert 15)
4 Pass 2: [15, 25, 35, 20, 30, 5, 10] (35 in place)
5 Pass 3: [15, 20, 25, 35, 30, 5, 10] (Insert 20)
6 Pass 4: [15, 20, 25, 30, 35, 5, 10] (Insert 30)
7 Pass 5: [5, 15, 20, 25, 30, 35, 10] (Insert 5)
8 Pass 6: [5, 10, 15, 20, 25, 30, 35] (Insert 10)
9
10 Final: [5, 10, 15, 20, 25, 30, 35]
```

- **Method:** Take element, find position in sorted part
- **Comparisons:** 15 total comparisons
- **Shifts:** Elements moved to make space

### Mnemonic

“TFI - Take, Find, Insert”

### Question 4(c) [7 marks]

Implement a python program to search a particular element from a list using Linear Search.

#### Solution

```
1 def linear_search(arr, target):
2     comparisons = 0
3
4     for i in range(len(arr)):
5         comparisons += 1
6         if arr[i] == target:
7             print(f"Element {target} found at index {i}")
8             print(f"Number of comparisons: {comparisons}")
9             return i
10
11     print(f"Element {target} not found")
12     print(f"Number of comparisons: {comparisons}")
13     return -1
14
15 def linear_search_all_positions(arr, target):
16     positions = []
17     for i in range(len(arr)):
18         if arr[i] == target:
19             positions.append(i)
20     return positions
21
22 # Example usage
23 data = [10, 25, 30, 15, 20, 30, 35]
24 target = 30
25
26 result = linear_search(data, target)
27 all_positions = linear_search_all_positions(data, target)
28 print(f"All positions of {target}: {all_positions}")
```

- **Sequential search:** Check each element one by one
- **Time complexity:**  $O(n)$  worst case
- **Best case:**  $O(1)$  if found at first position

#### Mnemonic

“CEO - Check Each One”

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

Write a program of Insertion Sort Method.

#### Solution

```

1 def insertion_sort(arr):
2     for i in range(1, len(arr)):
3         key = arr[i]
4         j = i - 1
5
6         while j >= 0 and arr[j] > key:
7             arr[j + 1] = arr[j]
8             j -= 1
9
10        arr[j + 1] = key
11
12    return arr
13
14 # Example usage
15 data = [12, 11, 13, 5, 6]
16 print("Original:", data)
17 sorted_data = insertion_sort(data.copy())
18 print("Sorted:", sorted_data)

```

- **Key element:** Current element to be inserted
- **Shift right:** Larger elements move right
- **Insert:** Key at correct position

#### Mnemonic

“KSI - Key, Shift, Insert”

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

Apply Quick Sort to the following data and arrange them in the proper manner. 5 6 1 8 2 9 10 15 7 13

#### Solution

##### Quick Sort Steps:

```

1 Initial: [5, 6, 1, 8, 2, 9, 10, 15, 7, 13]
2 Pivot: 5 (first element)
3
4 Partition 1: [1, 2] 5 [6, 8, 9, 10, 15, 7, 13]
5
6 Left subarray [1, 2]:
7 Pivot: 1 \rightarrow [] 1 [2]
8 Result: [1, 2]
9
10 Right subarray [6, 8, 9, 10, 15, 7, 13]:
11 Pivot: 6 \rightarrow [] 6 [8, 9, 10, 15, 7, 13]
12

```

```
3 Continue partitioning...
4
5 Final: [1, 2, 5, 6, 7, 8, 9, 10, 13, 15]
```

- **Divide:** Choose pivot, partition around it
- **Conquer:** Recursively sort subarrays
- **Average time:**  $O(n \log n)$

#### Mnemonic

“DCC - Divide, Conquer, Combine”

### Question 4(c OR) [7 marks]

Implement Merge sort algorithm.

#### Solution

```
1 def merge_sort(arr):
2     if len(arr) <= 1:
3         return arr
4
5     mid = len(arr) // 2
6     left = merge_sort(arr[:mid])
7     right = merge_sort(arr[mid:])
8
9     return merge(left, right)
10
11 def merge(left, right):
12     result = []
13     i = j = 0
14
15     while i < len(left) and j < len(right):
16         if left[i] <= right[j]:
17             result.append(left[i])
18             i += 1
19         else:
20             result.append(right[j])
21             j += 1
22
23     result.extend(left[i:])
24     result.extend(right[j:])
25
26     return result
27
28 # Example usage
29 data = [38, 27, 43, 3, 9, 82, 10]
30 sorted_data = merge_sort(data)
31 print("Sorted array:", sorted_data)
```

- **Divide:** Split array into halves
- **Merge:** Combine sorted subarrays
- **Time complexity:**  $O(n \log n)$  always

#### Mnemonic

“DSM - Divide, Sort, Merge”

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

Write Short note on: Applications of Tree.

### Solution

Table 8: Tree Applications

Application	Description	Example
<b>File systems</b>	Directory structure	Folders and files
<b>Expression parsing</b>	Mathematical expressions	$(a+b)*c$
<b>Database indexing</b>	Fast data retrieval	B-trees in databases

- **Decision trees:** AI and machine learning
- **Huffman coding:** Data compression
- **Game trees:** Chess, tic-tac-toe

### Mnemonic

“FED - File, Expression, Database”

## Question 5(b) [4 marks]

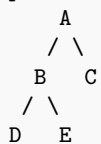
Explain different Tree Traversal Methods.

### Solution

Table 9: Tree Traversal Methods

Method	Order	Process
<b>Inorder</b>	Left-Root-Right	LNR
<b>Preorder</b>	Root-Left-Right	NLR
<b>Postorder</b>	Left-Right-Root	LRN

Example Tree:



Inorder: D B E A C

Preorder: A B D E C

Postorder: D E B C A

- **Inorder:** Gives sorted sequence for BST
- **Preorder:** Used for copying tree
- **Postorder:** Used for deleting tree

### Mnemonic

“LNR PNL LRN for In-Pre-Post”

## Question 5(c) [7 marks]

Write a menu driven program to perform the following operation on Binary Search Tree: Create a BST.

### Solution

```
1 class TreeNode:
2     def __init__(self, data):
3         self.data = data
4         self.left = None
```

```

5         self.right = None
6
7     class BST:
8         def __init__(self):
9             self.root = None
10
11         def insert(self, data):
12             self.root = self._insert_recursive(self.root, data)
13
14         def _insert_recursive(self, node, data):
15             if node is None:
16                 return TreeNode(data)
17
18             if data < node.data:
19                 node.left = self._insert_recursive(node.left, data)
20             elif data > node.data:
21                 node.right = self._insert_recursive(node.right, data)
22
23             return node
24
25         def inorder(self, node):
26             if node:
27                 self.inorder(node.left)
28                 print(node.data, end=" ")
29                 self.inorder(node.right)
30
31     def main():
32         bst = BST()
33
34         while True:
35             print("\n1. Insert")
36             print("2. Display (Inorder)")
37             print("3. Exit")
38
39             choice = int(input("Enter choice: "))
40
41             if choice == 1:
42                 data = int(input("Enter data: "))
43                 bst.insert(data)
44             elif choice == 2:
45                 print("BST (Inorder):", end=" ")
46                 bst.inorder(bst.root)
47                 print()
48             elif choice == 3:
49                 break
50
51     if __name__ == "__main__":
52         main()

```

- **BST property:** Left < Root < Right
- **Insertion:** Compare and go left/right
- **Menu driven:** User-friendly interface

### Mnemonic

“CIM - Compare, Insert, Menu”

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

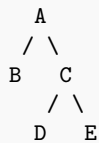
Define and give examples : Strict Binary Tree and Complete Binary Tree.

### Solution

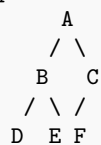
Table 10: Binary Tree Types

Type	Definition	Example
<b>Strict Binary Tree</b>	Every node has 0 or 2 children	Each internal node has exactly 2 children
<b>Complete Binary Tree</b>	All levels filled except possibly last, filled left to right	Perfect structure till second last level

Strict Binary Tree:



Complete Binary Tree:



- **Strict:** No node with single child
- **Complete:** Optimal space utilization

### Mnemonic

“SC - Strict Complete”

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

Explain basic terminology of Binary Tree : Level number, Degree, Indegree , Out-degree , Leaf Node.

### Solution

Binary Tree Example:

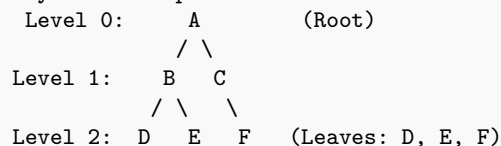


Table 11: Binary Tree Terminology

Term	Definition	Example
<b>Level number</b>	Distance from root (root = 0)	A=0, B=1, D=2
<b>Degree</b>	Number of children	A=2, B=2, C=1
<b>Indegree</b>	Number of incoming edges	All nodes = 1 (except root = 0)
<b>Out-degree</b>	Number of outgoing edges	Same as degree
<b>Leaf Node</b>	Node with no children	D, E, F

### Mnemonic

“LDIOL - Level, Degree, In-Out, Leaf”

## Question 5(c OR) [7 marks]

Write a menu driven program to perform the following operation on Binary Search Tree: Insert an element in BST.

## Solution

```
1 class TreeNode:
2     def __init__(self, data):
3         self.data = data
4         self.left = None
5         self.right = None
6
7 class BST:
8     def __init__(self):
9         self.root = None
10
11     def insert(self, data):
12         if self.root is None:
13             self.root = TreeNode(data)
14             print(f"Root node {data} created")
15         else:
16             self._insert_helper(self.root, data)
17
18     def _insert_helper(self, node, data):
19         if data < node.data:
20             if node.left is None:
21                 node.left = TreeNode(data)
22                 print(f"Inserted {data} to left of {node.data}")
23             else:
24                 self._insert_helper(node.left, data)
25         elif data > node.data:
26             if node.right is None:
27                 node.right = TreeNode(data)
28                 print(f"Inserted {data} to right of {node.data}")
29             else:
30                 self._insert_helper(node.right, data)
31         else:
32             print(f"Data {data} already exists")
33
34     def display_inorder(self, node, result):
35         if node:
36             self.display_inorder(node.left, result)
37             result.append(node.data)
38             self.display_inorder(node.right, result)
39
40 def main():
41     bst = BST()
42
43     while True:
44         print("\n--- BST Operations ---")
45         print("1. Insert Element")
46         print("2. Display BST (Inorder)")
47         print("3. Exit")
48
49         choice = int(input("Enter your choice: "))
50
51         if choice == 1:
52             data = int(input("Enter element to insert: "))
53             bst.insert(data)
54         elif choice == 2:
55             result = []
56             bst.display_inorder(bst.root, result)
57             print("BST Elements (sorted):", result)
58         elif choice == 3:
59             print("Exiting...")
60             break
61         else:
62             print("Invalid choice!")
63
64 if __name__ == "__main__":
65     main()
```



#6

```
main()
```

- **Insert logic:** Compare with current node, go left/right
- **Recursive approach:** Clean and efficient implementation
- **Menu system:** Interactive user interface

### Mnemonic

“CRL - Compare, Recursive, Left/right”