Subject Name Solutions

4331601 – Winter 2024

Semester 1 Study Material

Detailed Solutions and Explanations

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:

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:

```
\# 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:

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

```
\# 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:

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:

```
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:

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:

```
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

Queue Comparison:

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:

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

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:

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 → *pushtostack*
4. Lower/equal precedence → *popandaddtooutput*

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

```
Input: A+(B*C/D)

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

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:

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

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 → *addtooutput*
3. If '(' → *pushtostack*
3. If ')' → *popuntil*'('
3. If operator → *pophigher/equalprecedenceoperators*
3. Push current operator to stack

4. Pop remaining operators

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

```
Expression: 5 6 2 + * 12 4 / {-}

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:

```
def traverse\_linked\_list(head):
    current = head
    while current is not None:
        print(current.data)
        current = current.next
```

Algorithm Steps:

Step	Action	Purpose
1	Start from head node	Initialize traversal
2	Check if current $\neq NULL$	Continue condition
3	Process current node	Perform operation
4	Move to next node	Advance pointer
5	Repeat until end	Complete traversal

Mnemonic

"Traverse Through Till The Tail"

Question 3(b) [4 marks]

Write an algorithm for Dequeue operation in queue using List.

Solution

Dequeue Algorithm:

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

Algorithm Steps:

Step	Condition	Action
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:

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:

```
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:

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:

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

Algorithm Steps:

Step	Action	Description
1	Check capacity	Verify stack not full
2	Add element	Append to end of list
3	Update top	Top points to last element
4	Confirm operation	Display success message

Detailed Algorithm:

1. Accept stack and element to push
2. Check if stack has capacity (for fixed size)
3. Add element to end of list using append()
4. List automatically handles memory allocation
5. Return success status

Time Complexity: O(1) - Constant time operation

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:

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:

```
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
```

Algorithm Steps:

1. Handle empty list case
2. Handle single node case
3. Traverse to second last node
4. Set second last node's next to NULL
5. Return updated head

Mnemonic

"Linked Lists Lead to Logical Advantages"

Question 4(a) [3 marks]

Write an algorithm of Bubble sort.

Solution

Bubble Sort Algorithm:

```
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
```

Algorithm Steps:

Step	Action	Purpose
1	Outer loop i=0 to n-1	Number of passes
2	Inner loop j=0 to n-i-2	Compare adjacent elements
3	Compare arr[j] and arr[j+1]	Check ordering
4	Swap if out of order	Correct positioning
5	Repeat until sorted	Complete sorting

Time Complexity: $\mathbf{O(n^2)}$

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.
Characteristics:

Feature	Description	Benefit
Circular Structure	Last node → *Firstnode*	Continuous traversal
No NULL Pointers	No end marker	Always connected
Efficient Traversal	Can start from any node	Flexible access

Advantages:

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

```
Circular Linked List Structure:

[A]  [B]  [C]  [D]
 ↑                ↓
```

Mnemonic

"Circular Lists Create Continuous Connections"

Question 4(c) [7 marks]

Explain merge sort with suitable example.

Solution

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

```
Merge Sort Process:

Level 0:  [38, 27, 43, 3, 9, 82, 10]
             ↓
Level 1:  [38, 27, 43, 3] | [9, 82, 10]
             ↓                   ↓
Level 2:  [38, 27] [43, 3] | [9, 82] [10]
             ↓       ↓         ↓      ↓
Level 3:  [38][27] [43][3] | [9][82] [10]
             ↓       ↓         ↓      ↓
Merge:    [27, 38] [3, 43] | [9, 82] [10]
             ↓                   ↓
          [3, 27, 38, 43] | [9, 10, 82]
             ↓
          [3, 9, 10, 27, 38, 43, 82]
```

Time Complexity: O(n log n) Space Complexity: O(n)

Mnemonic

"Merge Sort Methodically Merges Segments"

Question 4(a) OR [3 marks]

Write an algorithm for selection sort.

Solution

Selection Sort Algorithm:

```
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
```

Algorithm Steps:

Step	Action	Purpose
1	Find minimum element	Locate smallest
2	Swap with first position	Place in sorted position
3	Move to next position	Advance boundary
4	Repeat for remaining	Continue sorting
5	Complete when done	Finish algorithm

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.
Structure:

Component	Purpose	Direction
Data	Store information	-
Next Pointer	Points to next node	Forward
Previous Pointer	Points to previous node	Backward

Advantages:

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

```
Double Linked List Structure:

NULL  [prev|data|next]  [prev|data|next]  [prev|data|next]  NULL
```

Applications:

- **Browser navigation** (back/forward buttons)
- **Music player** (previous/next song)
- **Undo/Redo operations**

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.
Algorithm Concept:

- **Sorted Portion**: Left side of current element
- **Unsorted Portion**: Right side of current element
- **Insert Strategy**: Place current element in correct position in sorted portion

Trace of [25, 15, 30, 9, 99, 20, 26]:

Pass	Current	Array State	Comparisons	Action
Initial	-	[25, 15, 30, 9, 99, 20, 26]	-	Start
1	15	[15, 25, 30, 9, 99, 20, 26]	$15 < 25$	Insert 15 before 25
2	30	[15, 25, 30, 9, 99, 20, 26]	$30 > 25$	Keep 30 in place
3	9	[9, 15, 25, 30, 99, 20, 26]	$9 <$ all	Insert at beginning
4	99	[9, 15, 25, 30, 99, 20, 26]	$99 > 30$	Keep 99 in place
5	20	[9, 15, 20, 25, 30, 99, 26]	Insert between 15,25	Shift and insert
6	26	[9, 15, 20, 25, 26, 30, 99]	Insert between 25,30	Final position

Final Sorted Array: [9, 15, 20, 25, 26, 30, 99]
Time Complexity: $\mathbf{O(n^2)}worstcase, \mathbf{O(n)}bestcase$

Mnemonic

"Insertion Inserts Into Increasing Order"

Question 5(a) [3 marks]

Explain application of binary tree.

Solution

Binary Tree 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
Decision Trees	AI/ML algorithms	Classification
Huffman Coding	Data compression	Optimal encoding

Key Benefits:

- **Hierarchical Structure**: Natural tree organization
- **Efficient Operations**: Search, insert, delete
- **Recursive Processing**: Easy to implement

Mnemonic

"Binary Trees Branch into Many Applications"

Question 5(b) [4 marks]

Write an algorithm for binary search using list.

Solution

Binary Search Algorithm:

```
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  \# Element not found
```

Algorithm Steps:

Step	Action	Purpose
1	Set left=0, right=n-1	Initialize boundaries

2	Calculate mid	Find middle element
3	Compare target with mid	Determine direction
4	Update boundaries	Narrow search space
5	Repeat until found	Continue searching

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

Types of Trees:

Tree Type	Description	Special Property
Binary Tree	Max 2 children per node	Left and right child
Binary Search Tree	Ordered binary tree	Left < Root < Right
Complete Binary Tree	All levels filled except last	Efficient heap
Full Binary Tree	All nodes have 0 or 2 children	No single child
AVL Tree	Self-balancing BST	Height balanced
Red-Black Tree	Self-balancing BST	Color properties

BST Insertion Algorithm:

```
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
```

Algorithm Steps:

1. If tree empty, create root node
2. If data < root.data, insert in left subtree
3. If data > root.data, insert in right subtree
4. If data = root.data, ignore (no duplicates)
5. Return updated 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:

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

Algorithm Steps:

Step	Action	Order
1	Traverse left subtree	Recursive call
2	Visit root node	Process data
3	Traverse right subtree	Recursive call

Traversal Order: Left $\rightarrow$ *Root* $\rightarrow$ *Right*
Properties:

- **BST Property: In-order gives sorted sequence**
- **Time Complexity: O(n)**
- **Space Complexity: O(h) where h is height**

Example Tree Result:

```
    Tree:      50
              /  {}
            30    70
           / {    / }
         20  40 60  80

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

```
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}
```

Algorithm Characteristics:

Feature	Description	Value
Method	Sequential checking	Element by element
Time Complexity	O(n)	Linear time
Space Complexity	O(1)	Constant space
Data Requirement	Any order	Unsorted data OK

Algorithm Steps:

1. Start from first element
2. Compare each element with target
3. If match found, return index
4. If end reached, return -1

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:

Term	Definition	Characteristics
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
Step-by-step Construction:

```
Step 1: Insert 60 (Root)
        60

Step 2: Insert 40 (40 { 60, go left)}
        60
       /
      40

Step 3: Insert 37 (37 { 60, go left; 37  40, go left)}
        60
       /
      40
     /
    37

Step 4: Insert 31 (31 { 60, left; 31  40, left; 31  37, left)}
        60
       /
      40
     /
    37
   /
  31

Step 5: Insert 59 (59 { 60, left; 59  40, right)}
        60
       /
      40
     / {}
    37  59
   /
  31

Step 6: Insert 21 (21 { 60, left; 21  40, left; 21  37, left; 21  31, left)}
        60
       /
      40
     / {}
    37  59
   /
  31
 /
21

Step 7: Insert 65 (65 { 60, go right)}
        60
       / {}
      40  65
     / {}
    37  59
   /
  31
 /
21

Step 8: Insert 30 (30 { 60, left; 30  40, left; 30  37, left; 30  31, left; 30  21, right)}
        60
       / {}
      40  65
```

```
     / {}
    37  59
   /
  31
 / {}
21  30
```

Final BST Structure:

Level	Nodes	Type
0	60	Root
1	40, 65	Internal
2	37, 59	Internal, Internal
3	31	Internal
4	21	Internal
5	30	Leaf

Leaf Nodes: 30, 59, 65

Mnemonic

"BST Building follows Binary Search Tree rules"