Data Structure and Application (1333203) - Winter 2023 Solution

Milav Dabgar

January 18, 2024

Question 1(a) [3 marks]

Define linked list. List different types of linked list.

Solution

Table 1. Linked List Definition and Types

Definition	Types of Linked List
A linked list is a linear data structure where elements are stored in nodes, and each node points to the next node in the sequence	1. Singly Linked List 2. Doubly Linked List 3. Circular Linked List 4. Circular Doubly Linked List

Singly: Data|Next ⟶ Data|Next ⟶ Data|Next ⟶ NULL

Doubly: P|D|N ⟷ P|D|N ⟷ P|D|N ⟶ NULL

Circular: Data|Next ⟶ Data|Next ⟶ Data|Next

Figure 1. Types of Linked Lists

Mnemonic

"Single, Double, Circle, Double-Circle"

Question 1(b) [4 marks]

Explain Linear and Non Linear Data structure in Python with examples.

> **Solution**
>
> **Table 2.** Linear vs Non-Linear Data Structures
>
> | Data Structure | Description | Python Examples |
> |---|---|---|
> | **Linear** | Elements arranged in sequential order where each element has exactly one predecessor and successor (except first and last) | Lists: `[1, 2, 3]`
Tuples: `(1, 2, 3)`
Strings: `"abc"`
Queue: `queue.Queue()` |
> | **Non-Linear** | Elements not arranged sequentially; an element can connect to multiple elements | Dictionary: `{"a": 1, "b": 2}`
Set: `{1, 2, 3}`
Tree: Custom implementation
Graph: Custom implementation |
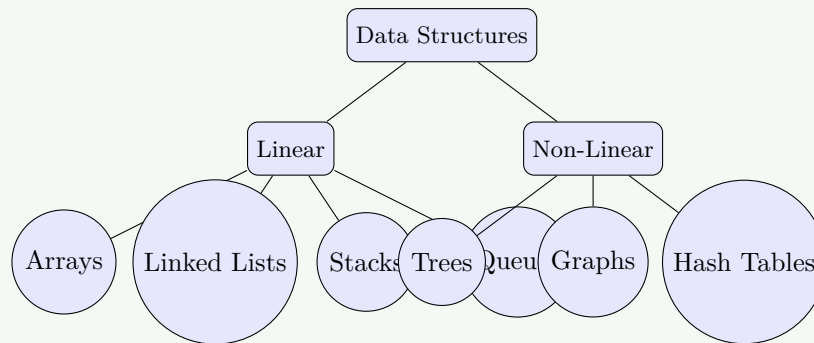>
>
>
> **Figure 2.** Classification of Data Structures

> **Mnemonic**
>
> "Linear Listens In Sequence, Non-linear Navigates Various Paths"

Question 1(c) [7 marks]

Explain class, attributes, object and class method in python with suitable example.

> **Solution**
>
>
>
> **Figure 3.** Class Diagram Example
>
> **Table 3.** OOP Concepts
>
> | Term | Description |
> |---|---|
> | **Class** | Blueprint for creating objects with shared attributes and methods |
> | **Attributes** | Variables that store data inside a class |
> | **Object** | Instance of a class with specific attribute values |
> | **Class Method** | Functions defined within a class that can access and modify class states |
>
> ```python
> class Student:
> ```

```
 2        # Class attribute
 3        school = "GTU"
 4
 5        # Constructor
 6        def __init__(self, roll_no, name):
 7            # Instance attributes
 8            self.roll_no = roll_no
 9            self.name = name
10
11        # Instance method
12        def display(self):
13            print(f"Roll No: {self.roll_no}, Name: {self.name}")
14
15        # Class method
16        @classmethod
17        def change_school(cls, new_school):
18            cls.school = new_school
19
20   # Creating object
21   student1 = Student(101, "Raj")
22   student1.display()   # Output: Roll No: 101, Name: Raj
```

Mnemonic

"Class Creates, Attributes Store, Objects Use, Methods Operate"

Question 1(c) OR [7 marks]

Define Data Encapsulation & Polymorphism. Develop a Python code to explain Polymorphism.

Solution

Table 4. Definitions

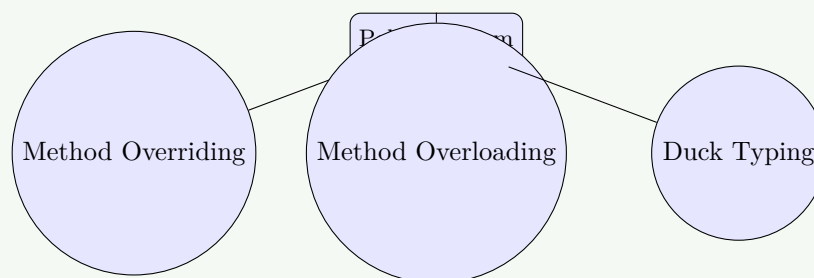| Concept	Definition
Data Encapsulation	Bundling data and methods into a single unit (class) and restricting direct access to some components
Polymorphism	Ability of different classes to provide their own implementation of methods with the same name

Figure 4. Types of Polymorphism

```
1   # Polymorphism example
2   class Animal:
3       def speak(self):
```

```
4          pass
5
6   class Dog(Animal):
7       def speak(self):
8           return "Woof!"
9
10  class Cat(Animal):
11      def speak(self):
12          return "Meow!"
13
14  class Duck(Animal):
15      def speak(self):
16          return "Quack!"
17
18  # Function demonstrating polymorphism
19  def animal_sound(animal):
20      return animal.speak()
21
22  # Creating objects
23  dog = Dog()
24  cat = Cat()
25  duck = Duck()
26
27  # Same function works for different animal objects
28  print(animal_sound(dog))   # Output: Woof!
29  print(animal_sound(cat))   # Output: Meow!
30  print(animal_sound(duck))  # Output: Quack!
```

Mnemonic

"Encapsulate to Protect, Polymorphism for Flexibility"

Question 2(a) [3 marks]

Differentiate between Stack and Queue.

Solution

Table 5. Stack vs Queue

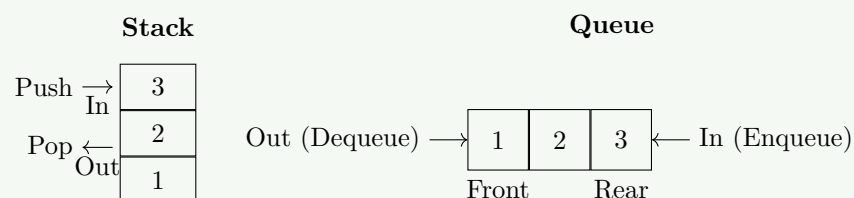| Feature	Stack	Queue
Principle	LIFO (Last In First Out)	FIFO (First In First Out)
Operations	Push, Pop	Enqueue, Dequeue
Access	Elements can only be added/removed from one end (top)	Elements are added at rear end and removed from front end

Figure 5. Stack vs Queue Visual

Question 2(b) [4 marks]

Write an algorithm for PUSH and POP operation of stack in python.

Solution

PUSH Algorithm:

```
Start
  1. Check if stack is full
  2. If not full, increment top by 1
  3. Add element at position 'top'
End
```

POP Algorithm:

```
Start
  1. Check if stack is empty
  2. If not empty, retrieve element at 'top'
  3. Decrement top by 1
  4. Return retrieved element
End
```

```python
class Stack:
    def __init__(self, size):
        self.stack = []
        self.size = size
        self.top = -1

    def push(self, element):
        if self.top >= self.size - 1:
            return "Stack Overflow"
        else:
            self.top += 1
            self.stack.append(element)
            return "Pushed " + str(element)

    def pop(self):
        if self.top < 0:
            return "Stack Underflow"
        else:
            element = self.stack.pop()
            self.top -= 1
            return element
```

Question 2(c) [7 marks]

Convert following equation from infix to postfix using Stack.
A * (B + C) - D / (E + F)

Solution

Infix	$A * (B + C) - D/(E + F)$
Postfix	$ABC + *DEF + /-$

Table 6. Infix to Postfix Conversion Trace

Step	Symbol	Stack	Output
1	A		A
2	*	*	A
3	(	* (	A
4	B	* (	A B
5	+	* (+	A B
6	C	* (+	A B C
7	)	*	A B C +
8	-	-	A B C + *
9	D	-	A B C + * D
10	/	- /	A B C + * D
11	(	- / (	A B C + * D
12	E	- / (	A B C + * D E
13	+	- / (+	A B C + * D E
14	F	- / (+	A B C + * D E F
15	)	- /	A B C + * D E F +
16	end		A B C + * D E F + / -

Answer: `A B C + * D E F + / -`

Mnemonic

"Operators Stack, Operands Print"

Question 2(a) OR [3 marks]

Differentiate between simple Queue and circular Queue.

Solution

Table 7. Simple vs Circular Queue

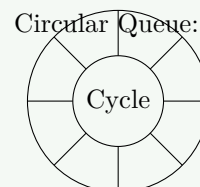| Feature	Simple Queue	Circular Queue
Structure	Linear data structure	Linear data structure with connected ends
Memory	Inefficient memory usage due to unused space after dequeue	Efficient memory usage by reusing empty spaces
Implementation	Front always at index 0, rear increases	Front and rear move in circular fashion using modulo

Simple Queue:
Front   Rear
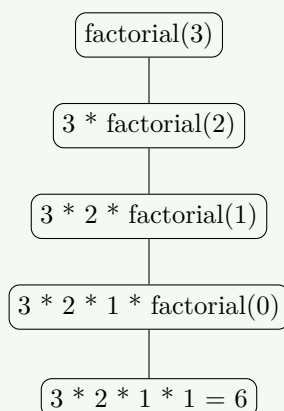
Circular Queue:

Cycle

Figure 6. Simple vs Circular Queue Structure

Question 2(b) OR [4 marks]

Explain concept of recursive function with suitable example.

Solution

Table 8. Recursive Function Concepts

Key Aspects	Description
Definition	A function that calls itself to solve a smaller instance of the same problem
Base Case	The condition where the function stops calling itself
Recursive Case	The condition where the function calls itself with a simpler version of the problem

factorial(3)

3 * factorial(2)

3 * 2 * factorial(1)

3 * 2 * 1 * factorial(0)

$3 * 2 * 1 * 1 = 6$

Figure 7. Recursive Calls trace for factorial(3)

```python
def factorial(n):
    # Base case
    if n == 0:
        return 1
    # Recursive case
    else:
        return n * factorial(n-1)

# Example
result = factorial(5)  # 5! = 120
```
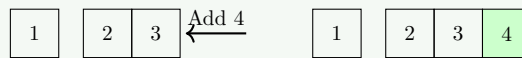
Question 2(c) OR [7 marks]

Develop a python code to implement Enqueue and Dequeue operation in Queue.

Solution

Enqueue Operation:

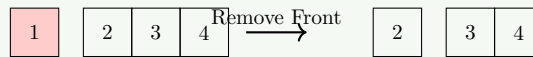**Dequeue Operation:**

Figure 8. Enqueue and Dequeue Visualization

```python
class Queue:
    def __init__(self, size):
        self.queue = []
        self.size = size
        self.front = 0
        self.rear = -1
        self.count = 0

    def enqueue(self, item):
        if self.count >= self.size:
            return "Queue is full"
        else:
            self.rear += 1
            self.queue.append(item)
            self.count += 1
            return "Enqueued " + str(item)

    def dequeue(self):
        if self.count <= 0:
            return "Queue is empty"
        else:
            item = self.queue.pop(0)
            self.count -= 1
            return item

    def display(self):
        return self.queue

# Test
q = Queue(5)
q.enqueue(10)
q.enqueue(20)
q.enqueue(30)
print(q.display())  # [10, 20, 30]
print(q.dequeue())  # 10
print(q.display())  # [20, 30]
```
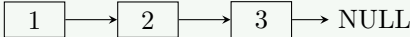
Mnemonic

"Enqueue at End, Dequeue from Start"

Question 3(a) [3 marks]

Give Difference between Singly linked list and Circular linked list.

Solution

Table 9. Singly vs Circular Linked List

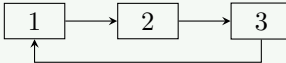| Feature	Singly Linked List	Circular Linked List
Last Node	Points to NULL	Points back to the first node
Traversal	Has a definite end	Can be traversed continuously
Memory	Each node needs one pointer	Each node needs one pointer

Singly: 1 → 2 → 3 → NULL

Circular: 1 → 2 → 3

Figure 9. Singly vs Circular Structure

Mnemonic

"Singly Stops, Circular Cycles"

Question 3(b) [4 marks]
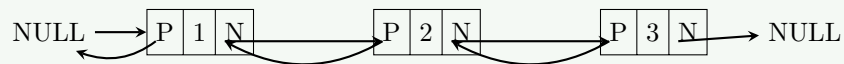
Explain concept of Doubly linked list.

Solution

NULL → P 1 N → P 2 N → P 3 N → NULL

Figure 10. Doubly Linked List Structure

Table 10. Doubly Linked List Features

Feature	Description
Node Structure	Each node contains data and two pointers (previous and next)
Navigation	Can traverse in both forward and backward directions
Operations	Insertion and deletion can be performed from both ends
Memory Usage	Requires more memory than singly linked list due to extra pointer

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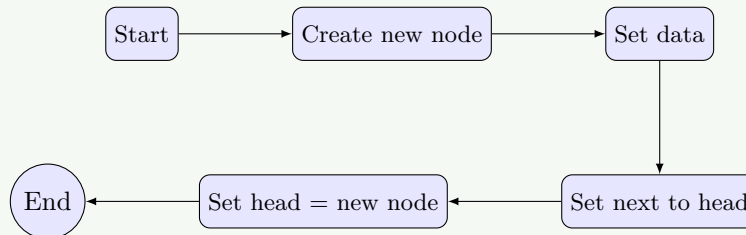
Mnemonic

"Double Pointers, Double Directions"

Question 3(c) [7 marks]

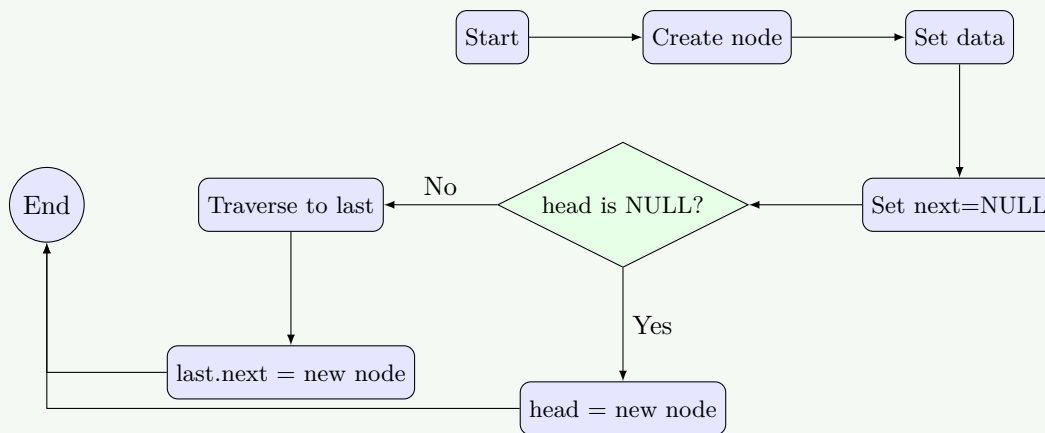**Write an algorithm for following operation on singly linked list:**
1. To insert a node at the beginning of the list.
2. To insert the node at the end of the list.

Solution

1. Insert at Beginning:

2. Insert at End:


```python
def insert_at_beginning(head, data):
    new_node = Node(data)
    new_node.next = head
    return new_node  # New head

def insert_at_end(head, data):
    new_node = Node(data)
    new_node.next = None

    # If linked list is empty
    if head is None:
        return new_node

    # Traverse to the last node
    temp = head
    while temp.next:
        temp = temp.next

    # Link the last node to new node
    temp.next = new_node
    return head
```

> **Mnemonic**
>
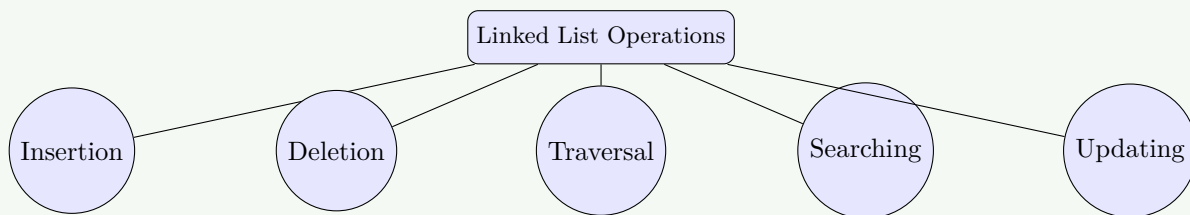> "Begin: New Leads Old, End: Old Leads New"

Question 3(a) OR [3 marks]

List different operations performed on singly linked list.

> **Solution**
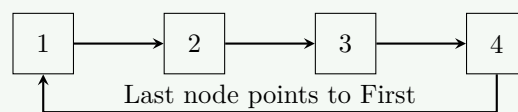>
> **Table 11.** Operations on Singly Linked List
>
> | Operations |
> | --- |
> | 1. Insertion (at beginning, middle, end) |
> | 2. Deletion (from beginning, middle, end) |
> | 3. Traversal (visiting each node) |
> | 4. Searching (finding a specific node) |
> | 5. Updating (modifying node data) |
>
> **Figure 11.** LL Operations

> **Mnemonic**
>
> "Insert Delete Traverse Search Update"

Question 3(b) OR [4 marks]

Explain concept of Circular linked list.

> **Solution**
>
> **Figure 12.** Circular Linked List Visualization
>
> **Table 12.** Circular LL Features
>
> | Feature | Description |
> | --- | --- |
> | **Structure** | Last node points to the first node instead of NULL |
> | **Advantage** | Allows continuous traversal through all nodes |
> | **Applications** | Round robin scheduling, circular buffer implementation |
> | **Operations** | Insertion and deletion similar to singly linked list with special handling for the last node |

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

# Creating a circular linked list with 3 nodes
head = Node(1)
node2 = Node(2)
node3 = Node(3)

head.next = node2
node2.next = node3
node3.next = head  # Makes it circular
```

Mnemonic

"Last Links to First"

Question 3(c) OR [7 marks]

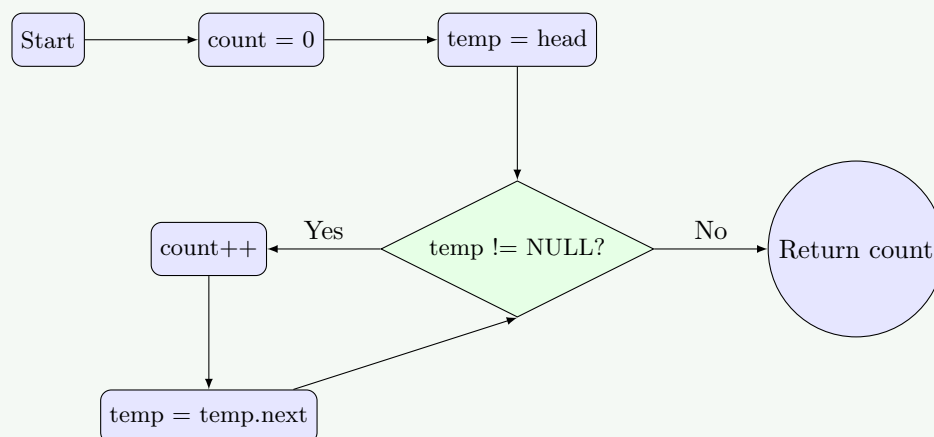**List application of linked list. Write an algorithm to count the number of nodes in singly linked list.**

Solution

Table 13. Applications

Applications of Linked List
1. Implementation of stacks and queues
2. Dynamic memory allocation
3. Undo functionality in applications
4. Hash tables (chaining)
5. Adjacency lists for graphs

Algorithm to Count Nodes:


```python
def count_nodes(head):
    count = 0
    temp = head

```

```
5       while temp:
6           count += 1
7           temp = temp.next
8
9       return count
10
11  # Example usage
12  # Assuming head points to the first node of a linked list
13  total_nodes = count_nodes(head)
14  print(f"Total nodes: {total_nodes}")
```

Mnemonic

"Count While Moving"

Question 4(a) [3 marks]

Compare Linear search with Binary search.

Solution

Table 14. Linear vs Binary Search

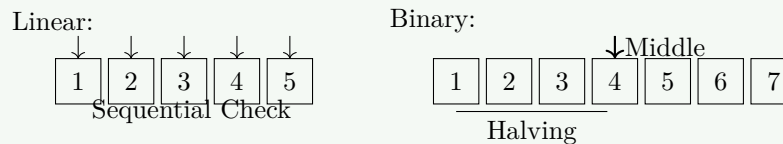| Feature	Linear Search	Binary Search
Data Arrangement	Works on both sorted and unsorted data	Works only on sorted data
Time Complexity	O(n)	O(log n)
Implementation	Simpler	More complex
Best For	Small datasets or unsorted data	Large sorted datasets

Figure 13. Search Comparison
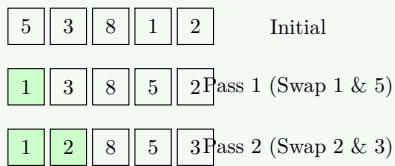
Mnemonic
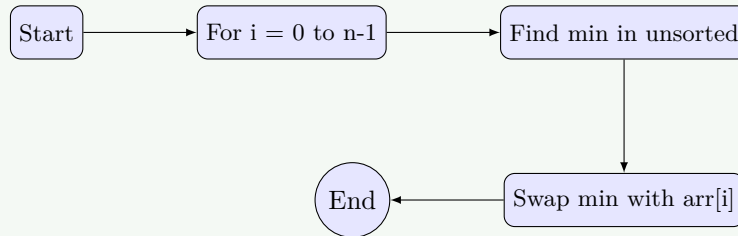
"Linear Looks at All, Binary Breaks in Half"

Question 4(b) [4 marks]

Write an algorithm for selection sort method.

Solution

Visualization:

| 5 | 3 | 8 | 1 | 2 | | Initial |

| 1 | 3 | 8 | 5 | 2 | Pass 1 (Swap 1 & 5) |

| 1 | 2 | 8 | 5 | 3 | Pass 2 (Swap 2 & 3) |

Algorithm:

Start → For i = 0 to n-1 → Find min in unsorted

End ← Swap min with arr[i]

```python
def selection_sort(arr):
    n = len(arr)

    for i in range(n):
        min_idx = i

        # Find the minimum element in unsorted array
        for j in range(i+1, n):
            if arr[j] < arr[min_idx]:
                min_idx = j

        # Swap the found minimum element with the first element
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
```

Mnemonic

"Find Minimum, Swap Position"

Question 4(c) [7 marks]

Develop a python code to sort following list in ascending order using Bubble sort method. list1=[5,4,3,2,1,0]
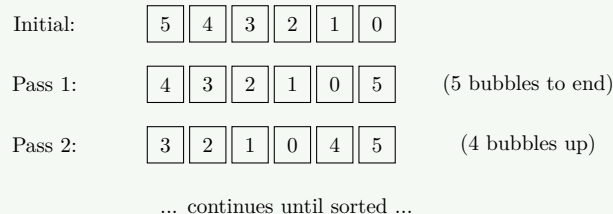
Solution

Initial: | 5 | 4 | 3 | 2 | 1 | 0 |

Pass 1: | 4 | 3 | 2 | 1 | 0 | 5 | (5 bubbles to end)

Pass 2: | 3 | 2 | 1 | 0 | 4 | 5 | (4 bubbles up)

... continues until sorted ...

Figure 14. Bubble Sort Trace

```python
def bubble_sort(arr):
    n = len(arr)

    # Traverse through all array elements
```

```
 5        for i in range(n):
 6            # Last i elements are already in place
 7            for j in range(0, n-i-1):
 8                # Swap if current element is greater than next element
 9                if arr[j] > arr[j+1]:
10                    arr[j], arr[j+1] = arr[j+1], arr[j]
11
12        return arr
13
14   # Input list
15   list1 = [5, 4, 3, 2, 1, 0]
16
17   # Sorting the list
18   sorted_list = bubble_sort(list1)
19
20   # Displaying the result
21   print("Sorted list:", sorted_list)
22   # Output: Sorted list: [0, 1, 2, 3, 4, 5]
```

Mnemonic

"Bubble Biggest Upward"

Question 4(a) OR [3 marks]

Define sorting. List different sorting methods.

Solution

Table 15. Sorting Definition

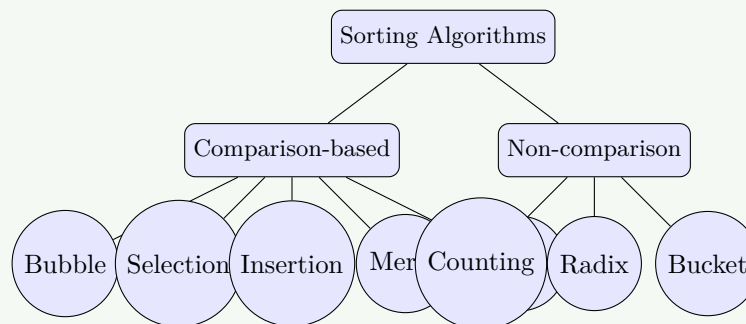| Definition	Sorting Methods
Sorting is the process of arranging data in a specified order (ascending or descending)	1. Bubble Sort 2. Selection Sort 3. Insertion Sort 4. Merge Sort 5. Quick Sort 6. Heap Sort 7. Radix Sort

Figure 15. Hierarchy of Sorting Algorithms

Mnemonic

"Better Sort Improves Many Query Results"

Question 4(b) OR [4 marks]

Write an algorithm for Insertion sort method.

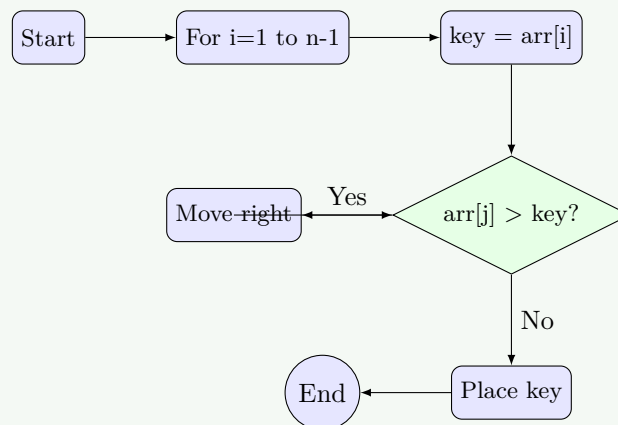**Solution**

Visualization:

Initial:	5	2	4	
Pass 1:	2	5	4	Insert 2
Pass 2:	2	4	5	Insert 4

Algorithm Flow:

Start → For i=1 to n-1 → key = arr[i]

Move right ←— Yes ——— arr[j] > key?

No

End ← Place key

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

        # Move elements that are greater than key
        # to one position ahead of their current position
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1

        arr[j + 1] = key
```

Mnemonic

"Take Card, Insert In Order"

Question 4(c) OR [7 marks]

Develop a python code to sort following list in ascending order using selection sort method. list1=[6,3,25,8,-1,55,0]

Solution

Initial:	6	3	25	8	-1	55	0					

Pass 1: | -1 | 3 | 25 | 8 | 6 | 55 | 0 | Swap -1 & 6

Pass 2: | -1 | 0 | 25 | 8 | 6 | 55 | 3 | Swap 0 & 3

... continues ...

Figure 16. Selection Sort Trace

```python
def selection_sort(arr):
    n = len(arr)

    for i in range(n):
        # Find the minimum element in remaining unsorted array
        min_idx = i
        for j in range(i+1, n):
            if arr[j] < arr[min_idx]:
                min_idx = j

        # Swap the found minimum element with the first element
        arr[i], arr[min_idx] = arr[min_idx], arr[i]

    return arr

# Input list
list1 = [6, 3, 25, 8, -1, 55, 0]

# Sorting the list
sorted_list = selection_sort(list1)

# Displaying the result
print("Sorted list:", sorted_list)
# Output: Sorted list: [-1, 0, 3, 6, 8, 25, 55]
```

Mnemonic

"Select Smallest, Shift to Start"

Question 5(a) [3 marks]

Define following terms regarding Tree data structure:
1. Forest
2. Root node
3. Leaf node

Solution

Table 16. Tree Terminology

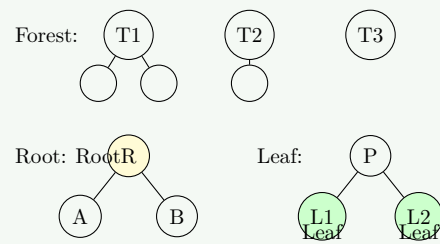| Term	Definition
Forest	Collection of disjoint trees (multiple trees without connections between them)
Root Node	Topmost node of a tree with no parent, from which all other nodes are descended
Leaf Node	Node with no children (terminal node at the bottom of the tree)

Figure 17. Tree Terms Visualization

Question 5(b) [4 marks]

Draw Binary search tree for 78, 58, 82, 15, 66, 80, 99 and write In-order traversal for the tree.
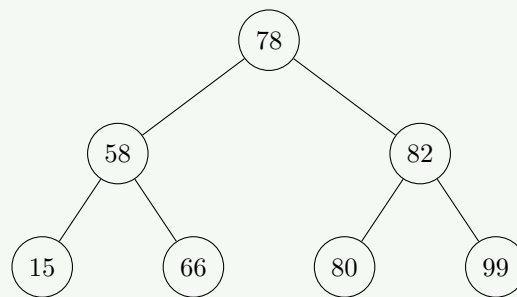
Solution

Figure 18. Binary Search Tree for Given Data

In-order Traversal:

Step	Visit Order
1	Visit left subtree of 78
2	Visit left subtree of 58
3	Visit 15
4	Visit 58
5	Visit 66
6	Visit 78
7	Visit left subtree of 82
8	Visit 80
9	Visit 82
10	Visit 99

In-order Traversal Result: 15, 58, 66, 78, 80, 82, 99
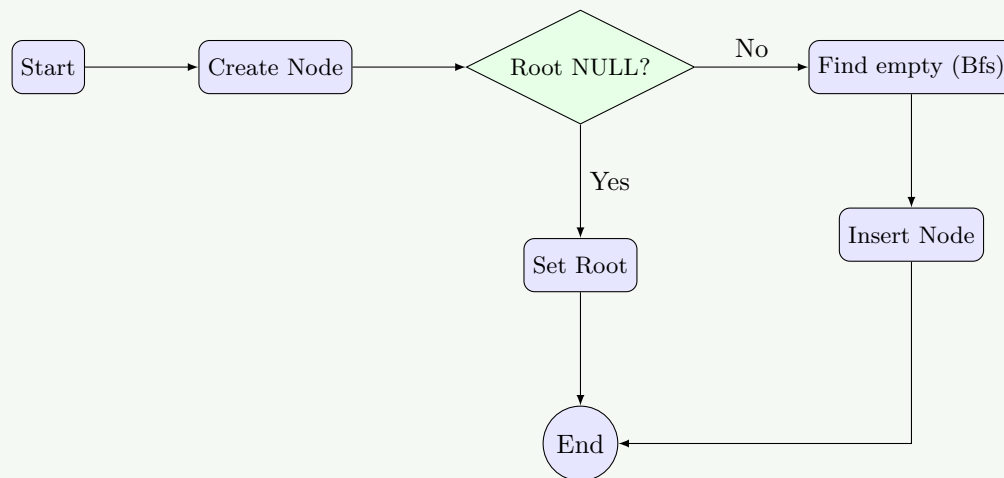
Mnemonic

"Left, Root, Right"

Question 5(c) [7 marks]
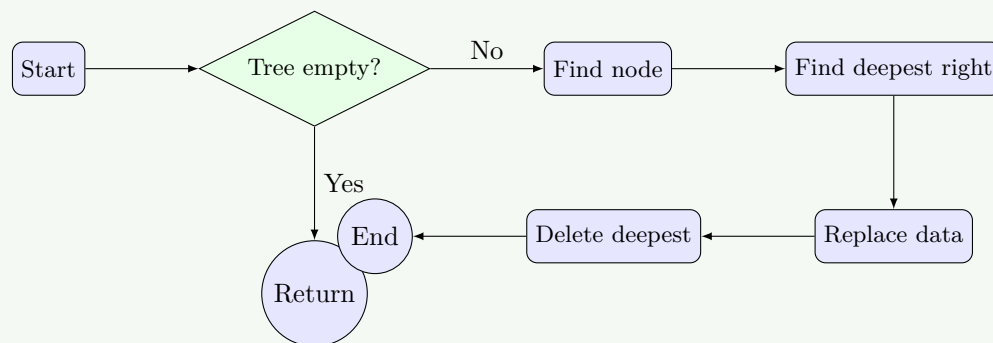
Write an algorithm for following operation:
1. Insertion of Node in Binary Tree
2. Deletion of Node in Binary Tree

Solution

Insertion Algorithm:

Deletion Algorithm:


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

# Insertion in Binary Tree
def insert(root, data):
    if root is None:
        return Node(data)

    # Level order traversal to find vacant position
    queue = []
    queue.append(root)

    while queue:
        temp = queue.pop(0)

        if temp.left is None:
            temp.left = Node(data)
```

```
21                  break
22              else:
23                  queue.append(temp.left)
24
25              if temp.right is None:
26                  temp.right = Node(data)
27                  break
28              else:
29                  queue.append(temp.right)
30
31      return root
32
33  # Deletion in Binary Tree
34  def delete_node(root, key):
35      if root is None:
36          return None
37
38      if root.left is None and root.right is None:
39          if root.data == key:
40              return None
41          else:
42              return root
43
44      # Find the node to delete and deepest node
45      key_node = None
46      last = None
47      parent = None
48      queue = []
49      queue.append(root)
50
51      while queue:
52          temp = queue.pop(0)
53          if temp.data == key:
54              key_node = temp
55          if temp.left:
56              parent = temp
57              queue.append(temp.left)
58              last = temp.left
59          if temp.right:
60              parent = temp
61              queue.append(temp.right)
62              last = temp.right
63
64      if key_node:
65          key_node.data = last.data
66          if parent.right == last:
67              parent.right = None
68          else:
69              parent.left = None
70
71      return root
```

Mnemonic

"Insert at Empty, Delete by Swap and Remove"

Question 5(a) OR [3 marks]

Define following terms regarding Tree data structure:

1. **In-degree**
2. **Out-degree**
3. **Depth**

Solution

Table 17. Definitions

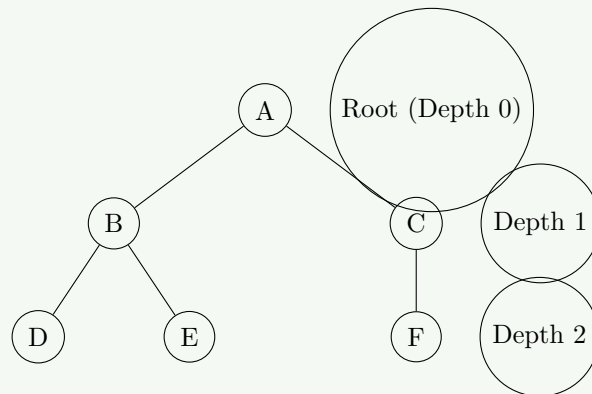| Term	Definition
In-degree	Number of edges coming into a node (always 1 for each node except root node in a tree)
Out-degree	Number of edges going out from a node (number of children)
Depth	Length of the path from root to the node (number of edges in path)

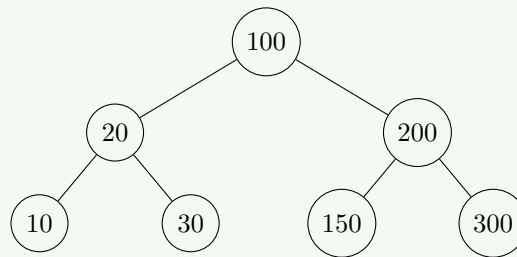**Figure 19.** Tree Depth and Degrees

Table 18. Degree Analysis

Node	In-degree	Out-degree
A	0	2
B	1	2
C	1	1
D	1	0
E	1	0
F	1	0

Mnemonic

"In Counts Parents, Out Counts Children, Depth Counts Edges from Root"

Question 5(b) OR [4 marks]

Write Preorder and postorder traversal of following Binary tree.
100 -> (20 -> (10, 30), 200 -> (150, 300))

> **Solution**
>
>
>
> **Figure 20.** Given Binary Tree
>
> **Table 19.** Traversals
>
> | Traversal | Order | Result |
> |---|---|---|
> | **Preorder** | Root, Left, Right | 100, 20, 10, 30, 200, 150, 300 |
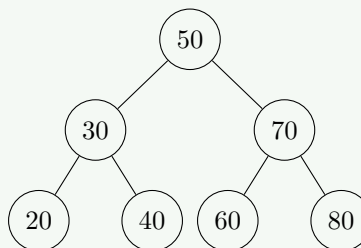> | **Postorder** | Left, Right, Root | 10, 30, 20, 150, 300, 200, 100 |

> **Mnemonic**
>
> "Preorder: Root First, Postorder: Children First"

Question 5(c) OR [7 marks]

Develop a program to implement construction of Binary Search Tree.

> **Solution**
>
> **Construction Visualization:**
>
>
>
> **Figure 21.** BST Constructed from [50, 30, 20, 40, 70, 60, 80]

```python
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

def insert(root, key):
    # If the tree is empty, return a new node
    if root is None:
        return Node(key)

    # Otherwise, recur down the tree
    if key < root.key:
        root.left = insert(root.left, key)
    else:
        root.right = insert(root.right, key)

```

```
18        # Return the unchanged node pointer
19        return root
20
21  def inorder(root):
22      if root:
23          inorder(root.left)
24          print(root.key, end=" ")
25          inorder(root.right)
26
27  def preorder(root):
28      if root:
29          print(root.key, end=" ")
30          preorder(root.left)
31          preorder(root.right)
32
33  def postorder(root):
34      if root:
35          postorder(root.left)
36          postorder(root.right)
37          print(root.key, end=" ")
38
39  # Driver program to test the above functions
40  def main():
41      # Create BST with these elements: 50, 30, 20, 40, 70, 60, 80
42      root = None
43      elements = [50, 30, 20, 40, 70, 60, 80]
44
45      for element in elements:
46          root = insert(root, element)
47
48      # Print traversals
49      print("Inorder traversal: ", end="")
50      inorder(root)
51      print("\nPreorder traversal: ", end="")
52      preorder(root)
53      print("\nPostorder traversal: ", end="")
54      postorder(root)
55
56  # Run the program
57  main()
```

Example Output:

```
1  Inorder traversal: 20 30 40 50 60 70 80
2  Preorder traversal: 50 30 20 40 70 60 80
3  Postorder traversal: 20 40 30 60 80 70 50
```

Mnemonic

"Insert Smaller Left, Larger Right"