Parul University

FACULTY OF ENGINEERING & TECHNOLOGY

BACHELOR OF TECHNOLOGY

COMPETITIVE CODING

(303105259)

4th SEMESTER

COMPUTER SCIENCE & ENGINEERING DEPARTMENT

Laboratory Mannual

COMPETITIVE CODING

PRACTICAL BOOK

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT

PREFACE

It gives us immense pleasure to present the first edition of the **Competitive Coding** for the **B.Tech. 2nd year students** at **PARUL UNIVERSITY**.

The **Competitive Coding** theory and laboratory courses at **PARUL UNIVERSITY, WAGHODIA, VADODARA** are meticulously designed to help students build problem-solving skills through exposure to algorithmic challenges. In the theory sessions, students learn the key concepts of algorithms, data structures, and various programming paradigms. These theoretical foundations are then applied through hands-on exercises and coding challenges, which allow students to practice and enhance their problem-solving abilities in competitive coding environments.

This laboratory book is not just about learning programming syntax; it focuses on sharpening the logical and analytical skills that make an excellent competitive programmer. A deep understanding of algorithms, optimization techniques, and efficient coding practices is essential in solving complex problems, especially within tight time limits. Competitive coding emphasizes critical thinking and efficient problem-solving, which are valuable skills in both academic and professional settings.

We acknowledge the authors and publishers of all the books and resources we have consulted while developing this Practical book. We hope that the **Competitive Coding Laboratory** will effectively serve its purpose, helping students develop the skills needed to excel in coding competitions and foster a passion for algorithmic problem-solving.

Instructions to students

1. Every student should obtain a copy of laboratory Manual.
2. Dress Code: Students must come to the laboratory wearing.
   i. Trousers,
   ii. half-sleeve tops and
   iii. Leather shoes. Half pants, loosely hanging garments and slippers are not allowed.
3. To avoid injury, the student must take the permission of the laboratory staff before handling any machine.
4. Students must ensure that their work areas are clean and dry to avoid slipping.
5. Do not eat or drink in the laboratory.
6. Do not remove anything from the computer laboratory without permission.
7. Do not touch, connect or disconnect any plug or cable without your lecturer/laboratory technician's permission.
8. All students need to perform the practical/program.

Parul® University NAAC A++ ACCREDITED UNIVERSITY

Faculty of Engineering & Technology
303105259 - Competitive Coding

CERTIFICATE

This is to certify that

Mr./Ms.....with enrolment no. .....has successfully completed his/her laboratory experiments in the ***COMPETITVE CODING (303105259)*** *from the department of* ***COMPUTER SCIENCE AND ENGINEERING*** *during the academic year* ***2024-25***

योगः कर्मसु कौशलम्

PARUL UNIVERSITY

Date of Submission:.....

Staff In charge:.....

INDEX

Class: 4th Sem **Subject: - Computative Coding**
A.Y. 2024-2025 **Subject Code: 303105256**

Sr. No.	Experiment Title	Page No.		Date of Performance	Date of Assessment	Marks out 0f 10	Sign
		To	From				
1	MinStack with Standard Operations Stack.						
2	Postfix Expression Evaluation with Error Handling.						
3	Next Greater Element (NGE) in an Array.						
4	Circular Queue Implementation.						
5	Infix to Postfix Conversion Using Queue.						
6	Product of Three Largest Distinct Elements.						
7	Merging Two Sorted Linked Lists.						
8	Merge Point of Two Sorted Linked Lists.						
9	Pairwise Node Swapping in a Linked List.						
10	Validate a Binary Search Tree (BST).						
11	Build a Binary Search Tree (BST).						
12	Determine the Depth of a Binary Tree						
13	Implement Tree Traversals (Pre-order, In-order, Post-order)						

14	Boundary Traversal of a BST.						
15	Lowest Common Ancestor in a BST.						
16	Verify if Two Trees are Mirrored.						
17	Implementing a Basic Hash Function.						
18	Hash Table with Separate Chaining.						
19	Two Sum Problem using HashMap.						
20	Trie Operations: Search, Insert, Remove.						
21	Huffman Coding Implementation.						
22	Finding Distinct Substrings in a String.						
23	Counting Words in a Trie.						
24	Count Total Words in a Trie.						
25	Level Order Traversal of a Tree.						

PRACTICAL – 1

AIM: Write a program for implementing a MINSTACK which should support operations like push, pop, overflow, underflow, display 1. Construct a stack of N-capacity 2. Push elements 3. Pop elements 4. Top element 5. Retrieve the min element from the stack

Problem Statement:

Design a MinStack data structure that supports the following operations:

1. **Push**: Add an element to the top of the stack.
2. **Pop**: Remove the element from the top of the stack.
3. **Top**: Retrieve the top element of the stack without removing it.
4. **GetMin**: Retrieve the minimum element in the stack.
5. **Display**: Display all elements in the stack.
6. **Overflow**: Check if the stack is full (if a capacity is defined).
7. **Underflow**: Check if the stack is empty.

The stack should have a fixed capacity N, and it should handle overflow and underflow conditions appropriately.

Algorithms:

1. **Initialize the Stack**:
    - Create two stacks: one for storing the actual elements (stack) and another for storing the minimum elements (min_stack).
    - Define the capacity N of the stack.
2. **Push Operation**:
    - Check if the stack is full (overflow condition). If full, display an error message.
    - Push the element onto the stack.
    - If the min_stack is empty or the new element is less than or equal to the top of the min_stack, push the element onto the min_stack.

3. **Pop Operation**:
    - Check if the stack is empty (underflow condition). If empty, display an error message.
    - Pop the top element from the stack.
    - If the popped element is equal to the top of the min_stack, pop the top element from the min_stack.
4. **Top Operation**:
    - Return the top element of the stack without removing it.
5. **GetMin Operation**:
    - Return the top element of the min_stack, which is the minimum element in the stack.
6. **Display Operation**:
    - Print all elements in the stack.
7. **Overflow Check**:
    - Return True if the stack is full, otherwise False.
8. **Underflow Check**:
    - Return True if the stack is empty, otherwise False.

CODE:

```
class MinStack:
    def __init__(self, capacity):
        self.capacity = capacity
        self.stack = []
        self.min_stack = []

    def is_overflow(self):
        return len(self.stack) == self.capacity

    def is_underflow(self):
```

```
    return len(self.stack) == 0

def push(self, value):
    if self.is_overflow():
        print("Stack Overflow: Cannot push element. Stack is full.")
        return
    self.stack.append(value)
    if not self.min_stack or value <= self.min_stack[-1]:
        self.min_stack.append(value)
    print(f"Pushed {value} to the stack.")

def pop(self):
    if self.is_underflow():
        print("Stack Underflow: Cannot pop element. Stack is empty.")
        return None
    popped_value = self.stack.pop()
    if popped_value == self.min_stack[-1]:
        self.min_stack.pop()
    print(f"Popped {popped_value} from the stack.")
    return popped_value

def top(self):
    if self.is_underflow():
        print("Stack is empty.")
        return None
    return self.stack[-1]

def get_min(self):
    if self.is_underflow():
```

```
            print("Stack is empty.")
            return None
        return self.min_stack[-1]

    def display(self):
        if self.is_underflow():
            print("Stack is empty.")
        else:
            print("Stack elements:", self.stack)

# Example usage
if __name__ == "__main__":
    capacity = 5
    min_stack = MinStack(capacity)

    min_stack.push(3)
    min_stack.push(5)
    min_stack.push(2)
    min_stack.push(1)
    min_stack.push(4)

    min_stack.display()  # Output: Stack elements: [3, 5, 2, 1, 4]
    print("Top element:", min_stack.top())  # Output: Top element: 4
    print("Minimum element:", min_stack.get_min())  # Output: Minimum element: 1

    min_stack.pop()
    min_stack.pop()

    min_stack.display()  # Output: Stack elements: [3, 5, 2]
```

```
print("Minimum element:", min_stack.get_min())  # Output: Minimum element: 2


min_stack.push(0)
min_stack.display()  # Output: Stack elements: [3, 5, 2, 0]
print("Minimum element:", min_stack.get_min())  # Output: Minimum element: 0
```

OUTPUT:

Output:

```
Pushed 3 to the stack.
Pushed 5 to the stack.
Pushed 2 to the stack.
Pushed 1 to the stack.
Pushed 4 to the stack.
Stack elements: [3, 5, 2, 1, 4]
Top element: 4
Minimum element: 1
Popped 4 from the stack.
Popped 1 from the stack.
Stack elements: [3, 5, 2]
Minimum element: 2
Pushed 0 to the stack.
Stack elements: [3, 5, 2, 0]
Minimum element: 0
```

PRACTICAL – 2

AIM: Write a program to deal with real-world situations where Stack data structure is widely used Evaluation of expression: Stacks are used to evaluate expressions, especially in languages that use postfix or prefix notation. Operators and operands are pushed onto the stack, and operations are performed based on the LIFO principle.

Problem Statement:

In many real-world applications, such as calculators or compilers, expressions need to be evaluated. One common way to evaluate expressions is by using the stack data structure, especially for expressions written in postfix (Reverse Polish Notation) or prefix notation. The stack's Last-In-First-Out (LIFO) property makes it ideal for this purpose.

Your task is to write a Python program that evaluates a postfix expression using a stack. The program should handle basic arithmetic operations like addition (+), subtraction (-), multiplication (*), and division (/).

Algorithm:

1. **Initialize an empty stack** to store operands.
2. **Iterate through each character** in the postfix expression:
    - If the character is an **operand** (number), push it onto the stack.
    - If the character is an **operator** (+, -, *, /):
        - Pop the top two elements from the stack.
        - Perform the operation on the two popped elements.
        - Push the result back onto the stack.
3. After processing all characters, the final result will be the only element left in the stack.
4. **Return the final result**.

CODE:

```
def evaluate_postfix(expression):
    stack = []
    for char in expression.split(): #275+- split read 2 , 7 , 5 not 275
        if char.isdigit():
            stack.append(int(char))
        else:
            b = stack.pop()
            a = stack.pop()
            if char == '+':
                stack.append(a + b)
            elif char == '-':
                stack.append(a - b)
            elif char == '*':
                stack.append(a * b)
            elif char == '/':
                stack.append(a / b)
    return stack.pop()

expression = "3 4 + 2 * 7 /"
print(evaluate_postfix(expression))

def evaluate_prefix(expression):
    stack = []
    for char in reversed(expression.split()):
        if char.isdigit():
            stack.append(int(char))
        else:
```

```
            a = stack.pop()
            b = stack.pop()
            if char == '+':
                stack.append(a + b)
            elif char == '-':
                stack.append(a - b)
            elif char == '*':
                stack.append(a * b)
            elif char == '/':
                stack.append(a / b)
    return stack.pop()

expression = "/ * + 3 4 2 7"
print(evaluate_prefix(expression))
```

OUTPUT:

Output:

```
2
2
```

PRACTICAL – 3

AIM: Write a program for finding NGE NEXT GREATER ELEMENT from an array.

Problem Statement:

Given an array of integers, the Next Greater Element (NGE) for each element is the first greater element to its right. If no such element exists, the NGE is considered to be -1. Write a program to find the NGE for each element in the array.

Algorithm;

1. **Initialize** a stack and a result array.
2. **Iterate** through the array from the end to the start.
3. For each element:
    - **Pop** elements from the stack until the top of the stack is greater than the current element or the stack is empty.
    - If the stack is empty, the NGE for the current element is -1.
    - Otherwise, the NGE is the top element of the stack.
    - **Push** the current element onto the stack.
4. **Reverse** the result array to match the original order of the input array.
5. **Return** the result array.

CODE:

```
def next_greater_element(arr):
    result = []
    for i in range(len(arr)):
        next_greater = -1

        for j in range(i+1, len(arr)):
            if arr[j] > arr[i] :
                next_greater = arr[j]
                break
        result.append(next_greater)
    return result

arr = [4, 5, 2, 10, 8]
print("Next greater elements:" , next_greater_element(arr))
```

OUTPUT:

```
Output:

('Next greater elements:', [5, 10, 10, -1, -1])
```

PRACTICAL – 4

AIM: Write a program to design a circular queue(k) which Should implement the below functions a. Enqueue b. Dequeue c. Front d. Rear

Problem Statement:

Design a circular queue data structure that supports the following operations:

1. **Enqueue**: Insert an element into the queue. If the queue is full, return an error or indicate that the operation cannot be performed.
2. **Dequeue**: Remove an element from the queue. If the queue is empty, return an error or indicate that the operation cannot be performed.
3. **Front**: Get the front item from the queue. If the queue is empty, return an error or indicate that the operation cannot be performed.
4. **Rear**: Get the last item from the queue. If the queue is empty, return an error or indicate that the operation cannot be performed.

The circular queue should efficiently utilize the space and handle wrap-around conditions.

Algorithm:

1. **Initialization**:
    - Use a list of size k to represent the circular queue.
    - Initialize two pointers, front and rear, to -1 to indicate an empty queue.
    - Track the current size of the queue using a variable size.
2. **Enqueue**:
    - Check if the queue is full (size == k). If full, return an error.
    - If the queue is empty (front == -1), set front and rear to 0.
    - Otherwise, increment rear circularly using (rear + 1) % k.
    - Insert the element at the rear position.
    - Increment the size.
3. **Dequeue**:

- Check if the queue is empty (size == 0). If empty, return an error.
- Remove the element at the front position.
- If the queue becomes empty (size == 1), reset front and rear to -1.
- Otherwise, increment front circularly using (front + 1) % k.
- Decrement the size.

4. **Front**:
   - Check if the queue is empty. If empty, return an error.
   - Return the element at the front position.
5. **Rear**:
   - Check if the queue is empty. If empty, return an error.
   - Return the element at the rear position.

CODE:

```
from collections import deque


class CircularQueue :
  def __init__(self, k) :
     self.queue = deque(maxlen = k)
     self.size = k


  def enqueue(self, value) :
     if len(self.queue) == self.size :
        return "Queue is Full!"
     self.queue.append(value)
     return f"Enqueued {value}"


  def dequeue(self) :
     if not self.queue:
        return "Queue is empty!"
```

```
        value = self.queue.popleft()
        return f"Dequeued {value}"

    def display(self):
        return list(self.queue)

k = 3

cq = CircularQueue(k)

print(cq.enqueue(10))
print(cq.enqueue(20))
print(cq.enqueue(30))
print(cq.enqueue(40))
print(cq.display())
print(cq.dequeue())
print(cq.display())
print(cq.enqueue(40))
print(cq.display())
```

OUTPUT:

Output:

```
Enqueued 10
Enqueued 20
Enqueued 30
Queue is Full!
[10, 20, 30]
Dequeued 10
[20, 30]
Enqueued 40
[20, 30, 40]
```

PRACTICAL – 5

AIM: Write a Program for an infix expression and convert it to postfix notation. Use a queue to implement the Shunting Yard Algorithm for expression conversion.

Problem Statement:

Write a Python program that takes an infix expression as input and converts it to postfix notation using the Shunting Yard Algorithm. The Shunting Yard Algorithm should be implemented using a queue to manage the output and a stack to manage the operators.

Algorithm:

1. **Initialize**:
    - A stack to hold operators.
    - A queue to hold the output (postfix expression).
    - A dictionary to define operator precedence.
2. **Tokenize the Input**:
    - Split the input string into tokens (numbers, operators, parentheses).
3. **Process Each Token**:
    - If the token is a number, add it to the output queue.

- If the token is an operator:
  - While there is an operator on the top of the stack with higher or equal precedence, pop it and add it to the output queue.
  - Push the current operator onto the stack.
- If the token is a left parenthesis (, push it onto the stack.
- If the token is a right parenthesis):
  - Pop from the stack and add to the output queue until a left parenthesis is encountered.
  - Pop the left parenthesis from the stack and discard it.

4. **Empty the Stack**:
   - After all tokens are processed, pop any remaining operators from the stack and add them to the output queue.
5. **Output the Postfix Expression**:
   - Convert the queue to a string to represent the postfix expression.

CODE:

```
from collections import deque


class ShuntingYard:
    def __init__(self):
        self.precedence = {'+': 1, '-': 1, '*': 2, '/': 2, '^': 3}
        self.operators = set(['+', '-', '*', '/', '^'])
        self.parenthesis = set(['(', ')'])


    def is_higher_precedence(self, op1, op2):
        return self.precedence.get(op1, 0) >= self.precedence.get(op2, 0)


    def infix_to_postfix(self, expression):
        stack = deque()
```

```
        queue = deque()
        i = 0
        while i < len(expression):
            if expression[i].isdigit():
                num = ''
                while i < len(expression) and expression[i].isdigit():
                    num += expression[i]
                    i += 1
                queue.append(num)
                continue
            elif expression[i] in self.operators:
                while (stack and stack[-1] in self.operators and
                       self.is_higher_precedence(stack[-1], expression[i])):
                    queue.append(stack.pop())
                stack.append(expression[i])
            elif expression[i] == '(':
                stack.append(expression[i])
            elif expression[i] == ')':
                while stack and stack[-1] != '(':
                    queue.append(stack.pop())
                stack.pop()  # Remove '(' from stack
            i += 1

        while stack:
            queue.append(stack.pop())
        return ' '.join(queue)

# Example usage
if __name__ == "__main__":
```

```
shunting_yard = ShuntingYard()
infix_expression = "3 + 4 * 2 / ( 1 - 5 ) ^ 2 ^ 3"
postfix_expression = shunting_yard.infix_to_postfix(infix_expression)
print("Infix Expression:", infix_expression)
print("Postfix Expression:", postfix_expression)
```

OUTPUT:

```
Output:

Infix Expression: 3 + 4 * 2 / ( 1 - 5 ) ^ 2 ^ 3
Postfix Expression: 3 4 2 * 1 5 - 2 ^ 3 ^ / +
```

PRACTICAL – 6

AIM: Write a Program for finding the Product of the three largest Distinct Elements. Use a Priority Queue to efficiently find and remove the largest elements.

Problem Statement:

Given an array of integers, find the product of the three largest distinct elements in the array. If there are fewer than three distinct elements, return -1. Use a priority queue (heap) to efficiently find and remove the largest elements.

Algorithm:

1. **Input**: An array of integers.
2. **Output**: The product of the three largest distinct elements or -1 if there are fewer than three distinct elements.
3. **Steps**:
    - Use a max-heap (priority queue) to efficiently find and remove the largest elements.
    - Insert all elements into the heap.
    - Extract the top three distinct elements from the heap.

- If fewer than three distinct elements are found, return -1.
- Otherwise, return the product of the three largest distinct elements.

CODE:

```
class PriorityQueue :
    def __init__(self) :
        self.array = []
    def is_empty(self) :
        return not self.array

    def enqueue(self, data, priority) :
        self.array.append((data, priority))
        self.array.sort(key = lambda x : x[1])
    def dequeue(self) :
        if self.is_empty() :
            print("Queue is underflowing")
            return None
        return self.array.pop(0)[0]
pq = PriorityQueue()
pq.enqueue(10,5)
pq.enqueue(12,2)
pq.enqueue(13,1)
pq.enqueue(7,3)
pq.enqueue(54,4)
elements = []
while not pq.is_empty():
    elements.append(pq.dequeue())
if len(elements) < 3 :
    print("Not enough element")
```

```
else :
    product = elements[-1] * elements[-2] * elements[-3]
    print(f"Product of the largest number is {product}")
```

OUTPUT:

```
Output:

Product of the largest number is 3780
```

PRACTICAL – 7

AIM: Write a Program to Merge two linked lists(sorted).

Problem Statement:

Given two sorted linked lists, merge them into a single sorted linked list. The new linked list should be created by splicing together the nodes of the first two lists.

Algorithm:

1. **Initialize Pointers**: Start with two pointers, one for each linked list, and a dummy node to build the merged list.
2. **Compare Nodes**: Compare the values of the nodes pointed to by the two pointers.
3. **Append Smaller Node**: Append the smaller node to the merged list and move the corresponding pointer to the next node.
4. **Repeat**: Continue the process until one of the lists is exhausted.
5. **Append Remaining Nodes**: Append the remaining nodes from the non-exhausted list to the merged list.
6. **Return Merged List**: Return the merged list, starting from the node after the dummy node.

CODE:

```
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def mergeTwoLists(l1: ListNode, l2: ListNode) -> ListNode:
    dummy = ListNode()
    current = dummy

    while l1 and l2:
        if l1.val < l2.val:
            current.next = l1
            l1 = l1.next
        else:
            current.next = l2
            l2 = l2.next
        current = current.next

    if l1:
        current.next = l1
    else:
        current.next = l2

    return dummy.next

def createLinkedList(values):
    dummy = ListNode()
    current = dummy
```

```
    for val in values:
        current.next = ListNode(val)
        current = current.next
    return dummy.next

def printLinkedList(head):
    current = head
    while current:
        print(current.val, end=" -> ")
        current = current.next
    print("None")

# Example usage
if __name__ == "__main__":
    # Create two sorted linked lists
    l1 = createLinkedList([1, 3, 5])
    l2 = createLinkedList([2, 4, 6])

    # Merge the two linked lists
    merged_list = mergeTwoLists(l1, l2)

    # Print the merged linked list
    printLinkedList(merged_list)
```

OUTPUT:

Output:

```
1 -> 2 -> 3 -> 4 -> 5 -> 6 -> None
```

PRACTICAL – 8

AIM: Write a Program to find the Merge point of two linked lists(sorted).

Problem Statement:

Given two sorted linked lists, find the merge point where the two lists merge into a single linked list. The merge point is the first node where the two lists share the same node. If the two lists do not merge, return None.

Algorithm:

1. **Initialize Pointers**: Start with two pointers, one for each linked list (let's call them p1 and p2).
2. **Traverse the Lists**: Traverse both linked lists simultaneously:
    - If the current node in p1 is smaller than the current node in p2, move p1 to the next node.
    - If the current node in p2 is smaller than the current node in p1, move p2 to the next node.
    - If the current nodes in p1 and p2 are equal, that node is the merge point.
3. **Handle Unequal Lengths**: If one list is longer than the other, continue traversing the longer list until the pointers meet.

4. **Return the Merge Point**: If a merge point is found, return the node. If the pointers reach the end of both lists without finding a merge point, return None.

CODE:

```
class ListNode:
    def __init__(self, value=0, next=None):
        self.value = value
        self.next = next


def find_merge_point(head1, head2):
    p1, p2 = head1, head2

    while p1 != p2:
        p1 = p1.next if p1 else head2
        p2 = p2.next if p2 else head1

    return p1


def create_linked_list(values):
    if not values:
        return None
    head = ListNode(values[0])
    current = head
    for value in values[1:]:
        current.next = ListNode(value)
        current = current.next
    return head


# Example usage:
# Create two linked lists that merge at some point
```

```
# List 1: 1 -> 3 -> 5 -> 7 -> 9
# List 2: 2 -> 4 -> 7 -> 9
# The merge point is the node with value 7

head1 = create_linked_list([1, 3, 5, 7, 9])
head2 = create_linked_list([2, 4])
# Merge the two lists at node with value 7
head2.next.next = head1.next.next.next

merge_point = find_merge_point(head1, head2)
if merge_point:
    print(f"Merge point value: {merge_point.value}")
else:
    print("No merge point found.")
```

OUTPUT:

```
Output:

Merge point value: 7
```

PRACTICAL – 9

AIM: Write a Program to Swap Nodes pairwise.

Problem Statement:

Given a singly linked list, swap every two adjacent nodes and return the head of the modified list. You must solve the problem without modifying the values in the list's nodes (i.e., only nodes themselves may be changed.)

Algorithm:

1. **Initialize Pointers:**
    - Use a dummy node to simplify edge cases (e.g., swapping the first two nodes).
    - Use prev to keep track of the node before the pair to be swapped.
    - Use curr to point to the first node of the pair.
2. **Swap Nodes:**
    - While curr and curr.next are not None:
        - Identify the two nodes to be swapped: first and second.
        - Update prev.next to point to second.
        - Update first.next to point to second.next.
        - Update second.next to point to first.
        - Move prev to first and curr to first.next.
3. **Return the Modified List:**
    - The new head of the list is dummy.next.

CODE:

```
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
```

```
        self.next = next

def swapPairs(head: ListNode) -> ListNode:
    # Create a dummy node to simplify edge cases
    dummy = ListNode(0)
    dummy.next = head
    prev = dummy

    while prev.next and prev.next.next:
        # Nodes to be swapped
        first = prev.next
        second = prev.next.next

        # Swapping
        prev.next = second
        first.next = second.next
        second.next = first

        # Move prev to the node before the next pair
        prev = first

    return dummy.next

# Helper function to print the linked list
def printList(head: ListNode):
    while head:
        print(head.val, end=" -> " if head.next else "\n")
        head = head.next
```

```
# Example usage
if __name__ == "__main__":
    # Create a linked list: 1 -> 2 -> 3 -> 4
    head = ListNode(1)
    head.next = ListNode(2)
    head.next.next = ListNode(3)
    head.next.next.next = ListNode(4)

    print("Original List:")
    printList(head)

    # Swap nodes pairwise
    new_head = swapPairs(head)

    print("List after swapping pairwise:")
    printList(new_head)
```

OUTPUT:

Output:

```
Original List:
1 -> 2 -> 3 -> 4
List after swapping pairwise:
2 -> 1 -> 4 -> 3
```

PRACTICAL – 10

AIM: Write a Program for Building a Function ISVALID to VALIDATE BST.

Problem Statement:

Given a binary tree, write a function isValidBST to determine if it is a valid Binary Search Tree (BST). A BST is valid if:

1. The left subtree of a node contains only nodes with values less than the node's value.
2. The right subtree of a node contains only nodes with values greater than the node's value.
3. Both the left and right subtrees must also be valid BSTs.

Algorithm:

1. **Define a helper function**:
    - The helper function will recursively check each node in the tree.
    - It will take three parameters: the current node, the minimum allowed value (min_val), and the maximum allowed value (max_val).
2. **Base Case**:
    - If the current node is None, return True (an empty tree is a valid BST).
3. **Check Current Node**:
    - If the current node's value is less than or equal to min_val or greater than or equal to max_val, return False (it violates BST rules).
4. **Recursive Case**:
    - Recursively check the left subtree, updating the max_val to the current node's value.
    - Recursively check the right subtree, updating the min_val to the current node's value.
5. **Return Result**:
    - If both subtrees are valid, return True.

CODE:

```
class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None
def is_valid_bst(root):
    def validate(node, low, high):
        if not node:
            return True
        if node.value <= low or node.value >= high:
            return False
        return (validate(node.left, low, node.value) and validate(node.right, node.value, high))
    return validate(root, float('-inf'), float('inf'))
root = Node(10)
root.left = Node(5)
root.right = Node(15)
print(is_valid_bst(root))
root = Node(10)
root.left = Node(15)
root.rigth = Node(5)
print(is_valid_bst(root))
```

OUTPUT:

```
Output:

True
False
```

PRECTICAL – 11

AIM: Write a Program to Build BST.

Problem Statement:

Build a Binary Search Tree (BST):

Given a list of integers, write a program to construct a Binary Search Tree (BST) from the list. The BST should satisfy the following properties:

1. The left subtree of a node contains only nodes with keys less than the node's key.
2. The right subtree of a node contains only nodes with keys greater than the node's key.
3. Both the left and right subtrees must also be binary search trees.

Algorithm:

1. **Define the Node Structure:**
    - Each node in the BST will have a value, a left child, and a right child.
2. **Insertion into BST:**
    - Start with the root node. If the tree is empty, the first element becomes the root.
    - For each subsequent element, compare it with the current node:
        - If the element is less than the current node's value, move to the left subtree.
        - If the element is greater than the current node's value, move to the right subtree.
    - Repeat the process until you find an empty spot (where the new node can be inserted).
3. **In-order Traversal (Optional):**
    - To verify the BST, perform an in-order traversal. The output should be a sorted list of elements.

CODE:

```
# Define the structure of a Node
class Node:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key

# Function to insert a new node with the given key
def insert(root, key):
    if root is None:
        return Node(key)
    else:
        if root.val < key:
            root.right = insert(root.right, key)
        else:
            root.left = insert(root.left, key)
    return root

# Function to do in-order traversal of BST
def inorder_traversal(root):
    if root:
        inorder_traversal(root.left)
        print(root.val, end=" ")
        inorder_traversal(root.right)

# Function to build BST from a list of integers
def build_bst(elements):
    root = None
```

```
    for element in elements:
        root = insert(root, element)
    return root

# Example usage
if __name__ == "__main__":
    elements = [50, 30, 20, 40, 70, 60, 80]
    root = build_bst(elements)

    print("In-order traversal of the constructed BST:")
    inorder_traversal(root)
```

OUTPUT:

Output:

```
In-order traversal of the constructed BST:
20 30 40 50 60 70 80
```

PRACTICAL – 12

AIM: Write a Program to determine the depth of a given Tree by Implementing MAXDEPTH.

Problem Statement:

Given a binary tree, determine its maximum depth. The maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node. A leaf node is a node with no children.

Algorithm:

1. **Base Case**: If the tree is empty (i.e., the root is None), return 0.
2. **Recursive Case**:
    - Calculate the depth of the left subtree.
    - Calculate the depth of the right subtree.
    - The depth of the current node is the maximum of the depths of the left and right subtrees, plus 1 (for the current node).
3. **Return**: The depth of the tree.

CODE:

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def maxDepth(root: TreeNode) -> int:
    # Base case: if the tree is empty, return 0
    if not root:
        return 0
```

```
    # Recursively calculate the depth of the left and right subtrees
    left_depth = maxDepth(root.left)
    right_depth = maxDepth(root.right)

    # The depth of the current node is the maximum of the left and right depths, plus 1
    return max(left_depth, right_depth) + 1

# Example usage:
# Constructing a binary tree:
#     3
#    / \
#   9  20
#     /  \
#    15   7

root = TreeNode(3)
root.left = TreeNode(9)
root.right = TreeNode(20)
root.right.left = TreeNode(15)
root.right.right = TreeNode(7)

print("The maximum depth of the tree is:", maxDepth(root))
```

OUTPUT:

```
Output:

The maximum depth of the tree is: 3
```

PRACTICAL-13

AIM: Write a Program to Understand and implement Tree traversals i.e. Pre-Order Post-Order, In-Order.

Problem Statement:

Implement a binary tree and perform the following tree traversals:

1. **Pre-Order Traversal**: Visit the root node, then traverse the left subtree, and finally traverse the right subtree.
2. **In-Order Traversal**: Traverse the left subtree, visit the root node, and then traverse the right subtree.
3. **Post-Order Traversal**: Traverse the left subtree, traverse the right subtree, and finally visit the root node.

Algorithms:

1. **Pre-Order Traversal**:
    - Visit the root node.
    - Recursively perform a pre-order traversal of the left subtree.
    - Recursively perform a pre-order traversal of the right subtree.
2. **In-Order Traversal**:
    - Recursively perform an in-order traversal of the left subtree.
    - Visit the root node.
    - Recursively perform an in-order traversal of the right subtree.
3. **Post-Order Traversal**:
    - Recursively perform a post-order traversal of the left subtree.
    - Recursively perform a post-order traversal of the right subtree.
    - Visit the root node.

CODE:

```
# Define the structure of a Node
class Node:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key

# Pre-Order Traversal
def pre_order_traversal(root):
    if root:
        print(root.val, end=" ")  # Visit the root
        pre_order_traversal(root.left)  # Traverse the left subtree
        pre_order_traversal(root.right)  # Traverse the right subtree

# In-Order Traversal
def in_order_traversal(root):
    if root:
        in_order_traversal(root.left)  # Traverse the left subtree
        print(root.val, end=" ")  # Visit the root
        in_order_traversal(root.right)  # Traverse the right subtree

# Post-Order Traversal
def post_order_traversal(root):
    if root:
        post_order_traversal(root.left)  # Traverse the left subtree
        post_order_traversal(root.right)  # Traverse the right subtree
        print(root.val, end=" ")  # Visit the root
```

```
# Driver code
if __name__ == "__main__":
    # Create a binary tree
    root = Node(1)
    root.left = Node(2)
    root.right = Node(3)
    root.left.left = Node(4)
    root.left.right = Node(5)
    root.right.left = Node(6)
    root.right.right = Node(7)

    print("Pre-Order Traversal:")
    pre_order_traversal(root)
    print("\n")
    print("In-Order Traversal:")
    in_order_traversal(root)
    print("\n")
    print("Post-Order Traversal:")
    post_order_traversal(root)
    print("\n")
```

OUTPUT:

Output:

```
Pre-Order Traversal:
1 2 4 5 3 6 7

In-Order Traversal:
4 2 5 1 6 3 7

Post-Order Traversal:
4 5 2 6 7 3 1
```

PRACTICAL-14

AIM: Write a Program to perform Boundary Traversal on BST.

Problem Statement:

Given a Binary Search Tree (BST), perform a **Boundary Traversal** of the tree. Boundary Traversal involves traversing the boundary nodes of the tree in the following order:

1. **Left Boundary**: Traverse the left boundary from the root to the leftmost leaf (excluding the leaf if it has a right subtree).
2. **Leaf Nodes**: Traverse all the leaf nodes from left to right.
3. **Right Boundary**: Traverse the right boundary from the rightmost leaf to the root (excluding the leaf if it has a left subtree).

The traversal should output the nodes in the correct order.

Algorithm:

1. **Left Boundary Traversal**:
    - Start from the root and move to the left child until a leaf node is reached.
    - If a node has no left child, move to the right child.
    - Exclude the leaf node if it has a right subtree.
2. **Leaf Nodes Traversal**:
    - Perform an inorder traversal and collect all leaf nodes.
3. **Right Boundary Traversal**:
    - Start from the root and move to the right child until a leaf node is reached.
    - If a node has no right child, move to the left child.
    - Exclude the leaf node if it has a left subtree.
    - Reverse the collected nodes to get the correct order.
4. **Combine Results**:
    - Combine the left boundary, leaf nodes, and right boundary to get the final result.

CODE:

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right


def is_leaf(node):
    """Check if a node is a leaf node."""
    return node.left is None and node.right is None


def left_boundary(root, result):
    """Traverse the left boundary."""
    if root is None or is_leaf(root):
        return
    result.append(root.val)
    if root.left:
        left_boundary(root.left, result)
    else:
        left_boundary(root.right, result)


def right_boundary(root, result):
    """Traverse the right boundary."""
    if root is None or is_leaf(root):
        return
    if root.right:
        right_boundary(root.right, result)
    else:
        right_boundary(root.left, result)
```

```
    result.append(root.val)  # Append after traversal to reverse the order

def leaf_nodes(root, result):
    """Traverse and collect all leaf nodes."""
    if root is None:
        return
    if is_leaf(root):
        result.append(root.val)
        return
    leaf_nodes(root.left, result)
    leaf_nodes(root.right, result)

def boundary_traversal(root):
    """Perform boundary traversal of the BST."""
    if root is None:
        return []

    result = []

    # Add root if it's not a leaf
    if not is_leaf(root):
        result.append(root.val)

    # Traverse left boundary
    left_boundary(root.left, result)

    # Traverse leaf nodes
    leaf_nodes(root, result)
```

```
    # Traverse right boundary
    right_boundary(root.right, result)

    return result

# Example usage
if __name__ == "__main__":
    # Construct a BST
    root = TreeNode(20)
    root.left = TreeNode(8)
    root.left.left = TreeNode(4)
    root.left.right = TreeNode(12)
    root.left.right.left = TreeNode(10)
    root.left.right.right = TreeNode(14)
    root.right = TreeNode(22)
    root.right.right = TreeNode(25)

    # Perform boundary traversal
    print("Boundary Traversal:", boundary_traversal(root))
```

OUTPUT:

```
Output:

Boundary Traversal: [20, 8, 4, 10, 14, 25, 22]
```

PRACTICAL-15

AIM: Write a program for Lowest Common Ancestors

Problem Statement:

Given a binary tree and two nodes p and q, find the lowest common ancestor (LCA) of the two nodes. The lowest common ancestor is the lowest node in the tree that has both p and q as descendants (where we allow a node to be a descendant of itself).

Algorithm:

1. **Base Case:** If the current node is None, return None.
2. **Recursive Case:**
    - If the current node is either p or q, return the current node.
    - Recursively find the LCA in the left subtree.
    - Recursively find the LCA in the right subtree.
3. **Decision:**
    - If both the left and right subtrees return a non-None value, it means p and q are found in different subtrees, so the current node is the LCA.
    - If only one subtree returns a non-None value, return that value (either p or q is found in that subtree).
    - If both subtrees return None, return None.

CODE:

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def lowestCommonAncestor(root: TreeNode, p: TreeNode, q: TreeNode) -> TreeNode:
```

```
    # Base case: if root is None, return None
    if not root:
        return None

    # If root is one of p or q, return root
    if root == p or root == q:
        return root

    # Recursively find LCA in left and right subtrees
    left_lca = lowestCommonAncestor(root.left, p, q)
    right_lca = lowestCommonAncestor(root.right, p, q)

    # If both left and right return non-None, root is the LCA
    if left_lca and right_lca:
        return root

    # Otherwise, return the non-None value (either left_lca or right_lca)
    return left_lca if left_lca else right_lca

# Example usage:
# Constructing the binary tree
#       3
#      / \
#     5   1
#    / \ / \
#   6  2 0  8
#     / \
#    7   4
```

```
root = TreeNode(3)
root.left = TreeNode(5)
root.right = TreeNode(1)
root.left.left = TreeNode(6)
root.left.right = TreeNode(2)
root.right.left = TreeNode(0)
root.right.right = TreeNode(8)
root.left.right.left = TreeNode(7)
root.left.right.right = TreeNode(4)

# Nodes to find LCA for
p = root.left  # Node with value 5
q = root.left.right.right  # Node with value 4

# Find LCA
lca = lowestCommonAncestor(root, p, q)
print(f"LCA of {p.val} and {q.val} is {lca.val}")  # Output: LCA of 5 and 4 is 5
```

OUTPUT:

```
Output:

LCA of 5 and 4 is 5
```

PRACTICAL-16

AIM: Write a Program to verify and validate mirrored trees or not.

Problem Statement:

Given two binary trees, write a program to determine if they are mirrors of each other. Two binary trees are considered mirrors if they are symmetric in structure, and the corresponding nodes have the same values but are mirrored in position.

Algorithm:

1. **Base Case**: If both trees are empty (None), they are mirrors of each other.
2. **Recursive Case**:
    - Check if the current nodes of both trees have the same value.
    - Recursively check if the left subtree of the first tree is a mirror of the right subtree of the second tree.
    - Recursively check if the right subtree of the first tree is a mirror of the left subtree of the second tree.
3. If any of the above conditions fail, the trees are not mirrors.

CODE:

```
class Node:
    def _init_(self,value):
        self.value = value
        self.left = self.right = None


def is_mirrored(root1 , root2):
    if not root1 and not root2:
        return True
```

```
    if not root1 or not root2:
        return False

    return(root1.value == root2.value) and is_mirrored(root1.left, root2.right) and
is_mirrored(root2.left, root1.right)

tree1 = Node(1)
tree1.left = Node(2)
tree1.right = Node(3)

tree2 = Node(1)
tree2.left = Node(3)
tree2.right = Node(2)

if is_mirrored(tree1, tree2):
    print("Mirror Tree")

else:
    print("Not mirrored tree")
```

OUTPUT:

```
Output:

Mirror Tree
```

PRACTICAL-17

AIM: Write a Program for a basic hash function in a programming language of your choice. Demonstrate its usage to store and retrieve key-value pairs.

Problem Statement:

Design and implement a basic hash function in Python that can be used to store and retrieve key-value pairs. The hash function should map keys to indices in a fixed-size array (hash table). Demonstrate the usage of this hash function by storing and retrieving key-value pairs.

Algorithm:

1. **Define a Hash Function**:
    - The hash function will take a key as input and return an integer representing the index in the hash table.
    - A simple hash function can be implemented by summing the ASCII values of the characters in the key and then taking the modulo with the size of the hash table.
2. **Create a Hash Table**:
    - The hash table will be a list of a fixed size, where each index can store a key-value pair.
3. **Insert Key-Value Pair**:
    - Use the hash function to determine the index for the key.
    - Store the key-value pair at the computed index in the hash table.
4. **Retrieve Value by Key**:
    - Use the hash function to determine the index for the key.
    - Retrieve the value stored at the computed index in the hash table.
5. **Handle Collisions**:
    - For simplicity, this implementation will not handle collisions. In a real-world scenario, techniques like chaining or open addressing would be used.

CODE:

```
class HashTable:
  def __init__(self):
    self.table = {}

  def insert(self, key, value):
    self.table[key] = value

  def retrieve(self, key):
    return self.table.get(key, "Not Found")

hash_table = HashTable()
hash_table.insert("name", "Kirtan")
hash_table.insert("age", 20)

print("Name:", hash_table.retrieve("name"))
print("Age:", hash_table.retrieve("age"))
```

OUTPUT:

Output:

```
Name: Kirtan
Age: 20
```

PRACTICAL-18

AIM: Implement a hash table using separate chaining for collision handling. Perform operations like insertion, deletion, and search on the hash table.

Problem Statement:

Implement a hash table using separate chaining for collision handling. The hash table should support the following operations:

1. **Insertion**: Insert a key-value pair into the hash table. If the key already exists, update the value.
2. **Deletion**: Remove a key-value pair from the hash table based on the key.
3. **Search**: Retrieve the value associated with a given key from the hash table.

The hash table should handle collisions using separate chaining, where each bucket in the hash table is a linked list of key-value pairs.

Algorithm:

1. **Hash Function**:
    - Use a simple hash function, such as hash(key) % size, where size is the number of buckets in the hash table.
2. **Insertion**:
    - Compute the hash value for the key.
    - Locate the bucket corresponding to the hash value.
    - If the bucket is empty, create a new linked list node with the key-value pair and add it to the bucket.
    - If the bucket is not empty, traverse the linked list to check if the key already exists. If it does, update the value. If not, append the new key-value pair to the end of the linked list.
3. **Deletion**:
    - Compute the hash value for the key.
    - Locate the bucket corresponding to the hash value.
    - Traverse the linked list to find the key. If found, remove the node from the linked list.
4. **Search**:

- Compute the hash value for the key.
- Locate the bucket corresponding to the hash value.
- Traverse the linked list to find the key. If found, return the associated value. If not found, return None.

CODE:

```
class HashTable:
    def __init__(self, size):
        self.size = size
        self.table = [[] for _ in range(size)]

    def _hash(self, key):
        return hash(key) % self.size

    def insert(self, key, value):
        index = self._hash(key)
        bucket = self.table[index]

        # Check if the key already exists in the bucket
        for i, (k, v) in enumerate(bucket):
            if k == key:
                bucket[i] = (key, value)  # Update the value if key exists
                return

        # If key does not exist, append the new key-value pair
        bucket.append((key, value))

    def delete(self, key):
        index = self._hash(key)
        bucket = self.table[index]
```

```
        # Traverse the bucket to find the key
        for i, (k, v) in enumerate(bucket):
            if k == key:
                del bucket[i]  # Remove the key-value pair
                return

    def search(self, key):
        index = self._hash(key)
        bucket = self.table[index]

        # Traverse the bucket to find the key
        for k, v in bucket:
            if k == key:
                return v  # Return the value if key is found

        return None  # Return None if key is not found

    def __str__(self):
        return str(self.table)

# Example usage
if __name__ == "__main__":
    ht = HashTable(10)

    ht.insert("apple", 5)
    ht.insert("banana", 10)
    ht.insert("orange", 15)
```

```
print("Hash Table after insertions:")
print(ht)

print("Search for 'apple':", ht.search("apple"))
print("Search for 'grape':", ht.search("grape"))

ht.delete("banana")
print("Hash Table after deleting 'banana':")
print(ht)

ht.insert("apple", 20)
print("Hash Table after updating 'apple':")
print(ht)
```

OUTPUT:

Output:

```
Hash Table after insertions:
[[], [], [], [('banana', 10), ('orange', 15)], [], [('apple', 5)], [], [], [], []]
Search for 'apple': 5
Search for 'grape': None
Hash Table after deleting 'banana':
[[], [], [], [('orange', 15)], [], [('apple', 5)], [], [], [], []]
Hash Table after updating 'apple':
[[], [], [], [('orange', 15)], [], [('apple', 20)], [], [], [], []]
```

PRACTICAL-19

AIM: Write a Program to Implement Two sums using HASHMAP.

Problem Statement:

Given an array of integers nums and an integer target, return the indices of the two numbers such that they add up to the target. You may assume that each input would have exactly one solution, and you may not use the same element twice. You can return the answer in any order.

Algorithm:

1. **Initialize a Hash Map**: Create an empty hash map (dictionary in Python) to store the numbers and their indices.
2. **Iterate through the array**: Loop through the array nums using an index i.
3. **Calculate the complement**: For each number nums[i], calculate the complement as complement = target - nums[i].
4. **Check if the complement exists in the hash map**:
    - If it exists, return the current index i and the index stored in the hash map for the complement.
    - If it does not exist, store the current number nums[i] and its index i in the hash map.
5. **Return the result**: If no solution is found, return an empty list or handle it as per requirements.

CODE:

```
def two_sum(nums, target):
  hashmap = {}

  for i, num in enumerate(nums):
    diff = target - num
    if diff in hashmap:
      return [hashmap[diff], i]
    hashmap[num] = i
```

```
return[]


nums = [2, 7, 11, 15]
target = 18
print(two_sum(nums, target))
```

OUTPUT:

```
Output:

[1, 2]
```

PRACTICAL-20

AIM: Write a Program to Implement Search, insert, and Remove in Trie.

Problem Statement:

A **Trie** (pronounced "try") is a tree-like data structure used to store a dynamic set of strings. It is particularly useful for operations like searching, inserting, and removing strings efficiently. Each node in a Trie represents a character of a string, and the path from the root to a node represents a prefix of one or more strings.

Your task is to implement a Trie data structure in Python with the following operations:

1. **Insert**: Insert a string into the Trie.
2. **Search**: Check if a string exists in the Trie.
3. **Remove**: Remove a string from the Trie.

Algorithm:

1. **Trie Node Structure**:
    - Each node in the Trie will have:
        - A dictionary to store child nodes (key: character, value: child node).
        - A boolean flag is_end_of_word to indicate if the node represents the end of a word.
2. **Insert Operation**:
    - Start from the root node.
    - For each character in the string, check if it exists in the current node's children.
    - If it doesn't exist, create a new node and add it to the children.
    - Move to the child node and repeat the process for the next character.
    - After processing all characters, mark the last node as the end of the word.
3. **Search Operation**:
    - Start from the root node.
    - For each character in the string, check if it exists in the current node's children.
    - If it doesn't exist, return False.
    - Move to the child node and repeat the process for the next character.

- After processing all characters, check if the last node is marked as the end of a word.

4. **Remove Operation**:
   - Start from the root node.
   - For each character in the string, check if it exists in the current node's children.
   - If it doesn't exist, return False (string not found).
   - Move to the child node and repeat the process for the next character.
   - After processing all characters, mark the last node as not being the end of a word.
   - If the node has no children, remove it from its parent's children.

CODE:

```
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end_of_word = False

class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word):
        current_node = self.root
        for char in word:
            if char not in current_node.children:
                current_node.children[char] = TrieNode()
            current_node = current_node.children[char]
        current_node.is_end_of_word = True

    def search(self, word):
```

```
    current_node = self.root
    for char in word:
        if char not in current_node.children:
            return False
        current_node = current_node.children[char]
    return current_node.is_end_of_word

def remove(self, word):
    def _remove(node, word, depth):
        if depth == len(word):
            if node.is_end_of_word:
                node.is_end_of_word = False
                return len(node.children) == 0
            return False

        char = word[depth]
        if char not in node.children:
            return False

        should_delete_child = _remove(node.children[char], word, depth + 1)

        if should_delete_child:
            del node.children[char]
            return len(node.children) == 0

        return False

    _remove(self.root, word, 0)
```

```
# Example Usage
trie = Trie()
trie.insert("apple")
trie.insert("app")
trie.insert("banana")

print(trie.search("apple"))  # Output: True
print(trie.search("app"))    # Output: True
print(trie.search("appl"))   # Output: False

trie.remove("app")
print(trie.search("app"))    # Output: False
print(trie.search("apple"))  # Output: True
```

OUTPUT:

```
Output:

True
True
False
False
True
```

PRACTICAL-21

AIM: Write a Program to Implement Huffman coding.

Problem Statement:

Huffman coding is a lossless data compression algorithm. The idea is to assign variable-length codes to input characters, with shorter codes assigned to more frequent characters. The algorithm builds a binary tree (Huffman tree) based on the frequency of characters and generates codes by traversing the tree. The goal is to implement Huffman coding to compress and decompress a given input string.

Algorithm:

1. **Frequency Calculation**: Calculate the frequency of each character in the input string.
2. **Priority Queue**: Create a priority queue (min-heap) where the node with the lowest frequency has the highest priority.
3. **Build Huffman Tree**:
    - Extract two nodes with the lowest frequency from the priority queue.
    - Create a new internal node with a frequency equal to the sum of the two nodes' frequencies.
    - Add the new node back to the priority queue.
    - Repeat until there is only one node left in the queue, which becomes the root of the Huffman tree.
4. **Generate Codes**: Traverse the Huffman tree to assign binary codes to each character.
5. **Encode**: Use the generated codes to encode the input string.
6. **Decode**: Use the Huffman tree to decode the encoded string back to the original string.

CODE:

```
import heapq

from collections import defaultdict, Counter


class HuffmanNode:

    def __init__(self, char, freq):
```

```
        self.char = char
        self.freq = freq
        self.left = None
        self.right = None

    def __lt__(self, other):
        return self.freq < other.freq

def build_huffman_tree(freq_map):
    priority_queue = [HuffmanNode(char, freq) for char, freq in freq_map.items()]
    heapq.heapify(priority_queue)

    while len(priority_queue) > 1:
        left = heapq.heappop(priority_queue)
        right = heapq.heappop(priority_queue)
        merged = HuffmanNode(None, left.freq + right.freq)
        merged.left = left
        merged.right = right
        heapq.heappush(priority_queue, merged)

    return heapq.heappop(priority_queue)

def generate_codes(root, current_code, codes):
    if root is None:
        return

    if root.char is not None:
        codes[root.char] = current_code
        return
```

```
    generate_codes(root.left, current_code + "0", codes)
    generate_codes(root.right, current_code + "1", codes)

def huffman_encode(data):
    freq_map = Counter(data)
    root = build_huffman_tree(freq_map)
    codes = {}
    generate_codes(root, "", codes)

    encoded_data = "".join([codes[char] for char in data])
    return encoded_data, root

def huffman_decode(encoded_data, root):
    decoded_data = []
    current_node = root

    for bit in encoded_data:
        if bit == '0':
            current_node = current_node.left
        else:
            current_node = current_node.right

        if current_node.char is not None:
            decoded_data.append(current_node.char)
            current_node = root

    return "".join(decoded_data)
```

```
# Example Usage
if __name__ == "__main__":
    input_data = "Huffman coding is a data compression algorithm."

    # Encoding
    encoded_data, huffman_tree = huffman_encode(input_data)
    print(f"Encoded Data: {encoded_data}")

    # Decoding
    decoded_data = huffman_decode(encoded_data, huffman_tree)
    print(f"Decoded Data: {decoded_data}")

    # Verify
    assert input_data == decoded_data, "Decoded data does not match the original!"
```

OUTPUT:

```
Encoded Data: 1111110000111100111000111010011010100011100111100010110110111
    1001100110101010111110010100010101010001110001111111101101010000100110
    0011100011010101011101111011110011010001100011110100111000000
Decoded Data: Huffman coding is a data compression algorithm.
```

PRACTICAL-22

AIM: Write a Program to find Distinct substrings in a string.

Problem Statement:

Given a string, find the number of distinct substrings in the string. A substring is a contiguous sequence of characters within a string. For example, the string "abc" has the following substrings: "a", "b", "c", "ab", "bc", and "abc". The total number of distinct substrings is 6.

Algorithm:

1. **Initialize a set to store unique substrings**: A set is used because it automatically handles duplicates.
2. **Iterate over the string**: Use nested loops to generate all possible substrings.
    - The outer loop runs from the start of the string to the end.
    - The inner loop runs from the current position of the outer loop to the end of the string.
3. **Generate substrings**: For each pair of indices (i, j), extract the substring from index i to j+1.
4. **Store substrings in the set**: Add each generated substring to the set.
5. **Count the number of unique substrings**: The size of the set at the end of the iteration will be the number of distinct substrings.
6. **Return the count**: Output the number of distinct substrings.

CODE:

```
def dist_substrings(s):
  substrings = set()
  for i in range(len(s)):
    for j in range(i +1, len(s) + 1):
      substrings.add(s[i:j])
  return substrings
s = "abc"
print(dist_substrings(s))
```

OUTPUT:

Output:

```
{'c', 'b', 'a', 'ab', 'abc', 'bc'}
```

PRACTICAL-23

AIM: Write a Program to find The No of Words in a Trie.

Problem Statement

Given a tree data structure, where each node contains a word, write a program to find the total number of words in the tree. The tree can be represented as a collection of nodes, where each node has a value (the word) and a list of children nodes.

Algorithm

1. Define the Tree Node Structure:
    - Each node should have a value (the word) and a list of children nodes.
2. Traverse the Tree:
    - Use a Depth-First Search (DFS) or Breadth-First Search (BFS) approach to traverse the tree.
    - For each node visited, increment a counter to keep track of the number of words.
3. Return the Count:
    - After traversing the entire tree, return the total count of words.

CODE:

```
class TreeNode:
    def __init__(self, value):
        self.value = value
        self.children = []

def count_words_in_tree(root):
    if not root:
        return 0
```

```
    # Initialize the count with the current node's word
    count = 1

    # Recursively count words in all children
    for child in root.children:
        count += count_words_in_tree(child)

    return count

# Example Usage
if __name__ == "__main__":
    # Create a sample tree
    root = TreeNode("Hello")
    child1 = TreeNode("World")
    child2 = TreeNode("Python")
    child3 = TreeNode("Programming")

    root.children.append(child1)
    root.children.append(child2)
    child2.children.append(child3)

    # Count the words in the tree
    total_words = count_words_in_tree(root)
    print(f"Total number of words in the tree: {total_words}")
```

OUTPUT:

```
Output:
Total number of words in the tree: 4
```

PRACTICAL-24

AIM: Write a Program to view a tree from left View.

Problem Statement:

Given a binary tree, print the left view of the tree. The left view of a binary tree contains all nodes that are visible when the tree is viewed from the left side. The leftmost node at each level is included in the left view.

Algorithm:

1. If the tree is empty, return an empty list.
2. Use a queue to perform a level-order traversal of the tree.
3. For each level, store the first node encountered in the left view list.
4. Continue traversing the tree level by level until all nodes are visited.
5. Print or return the left view list.

CODE:

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


def left_view(root, level = 0):
  if not root:
    return


  if level == len(view):
    print(root.value, end=" ")
    view.append(root.value)
```

```
    left_view(root.left, level + 1)
    left_view(root.right, level + 1)

view = []

root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
root.left.right.right = Node(6)
left_view(root)
```

OUTPUT:

```
Output:

1 2 4 6
```

PRACTICAL-25

AIM: Write a Program to Traverse a Tree using Level Order Traversal.

Problem Statement:

Given a binary tree, write a program to perform a **Level Order Traversal** (also known as **Breadth-First Traversal**). This traversal visits all nodes at each depth level before moving on to the next level.

Algorithm:

1. If the tree is empty, return.
2. Initialize a queue and enqueue the root node.
3. While the queue is not empty:
    - Dequeue a node from the front.
    - Process the dequeued node (print or store its value).
    - If the dequeued node has a left child, enqueue it.
    - If the dequeued node has a right child, enqueue it.
4. Repeat until all nodes have been processed.

CODE:

```
from collections import deque
class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None
def level_order_traversal(root):
    if not root:
        return
```

```
    queue = deque([root])

    while queue:
        current = queue.popleft()
        print(current.value, end=" ")  # Process the current node

        if current.left:
            queue.append(current.left)
        if current.right:
            queue.append(current.right)

if __name__ == "__main__":
    root = Node(1)
    root.left = Node(2)
    root.right = Node(3)
    root.left.left = Node(4)
    root.left.right = Node(5)
    root.right.left = Node(6)
    root.right.right = Node(7)

    print("Level Order Traversal of the Tree:")
    level_order_traversal(root)
```

OUTPUT:

```
Output:

Level Order Traversal of the Tree:
1 2 3 4 5 6 7
```