No	Date	Title of the Exercises	Marks	Page no	Staff -Sign
1 (a)		MIN HEAP (Insertion ,Delete Min ,Delete Max)			
(b)		SKEW HEAP (Priority Queue Operations)			
(c)		FIBONACCI HEAP (Priority Queue Operations)			
2 (a)		AVL TREES (Insertion ,Delete and search)			
(b)		SPLAY TREES (Insertion ,Delete and search)			
(c)		B-TREES(Insertion ,Delete and search)			
(d)		RED-BLACK TREE			
3		IMPLEMENTATION OF CONVEX HULL			
4		IMPLEMENTATION OF TOPOLOGICAL SORT			
5		IMPLEMENTATION OF GRAPH SEARCH ALGORITHMS			
6		IMPLEMENTATION OF RANDOMIZED ALGORITHMS			
7		IMPLEMENTATION AND APPLICATION OF NETWORK FLOW AND LINEAR PROGRAMMING PROBLEMS			
8		IMPLEMENTATION OF ALGORITHMS USING THEHILL CLIMBING AND DYNAMIC PROGRAMMING DESIGN TECHINIQUES			
9		IMPLEMENTATION OF RECURSIVE BACKTRACKING ALGORITHMS			
10		IMPLEMENTATION OF BRANCH AND BOUND ALGORITHM			

Ex.no : 1(a)	MIN HEAP (Insertion ,Delete Min ,Delete Max)
Date :	

AIM :

To write a python program for mi heap (Insetion , Delete min , Delete min)

ALGORITHM :

Step 1 : Insertion:

- Add the new element at the end of the heap.
- Heapify Up: Swap the element with its parent until the heap property is restored.

Step 2 : Delete Min:

- Swap the root with the last element.
- Remove the last element (previously the root).

Step 3 :Heapify Down:

- Swap the root with its smallest child until the heap property is restored.

Step 4 :Delete Max:

- Swap the root with the largest child (either left or right).
- Remove the last element (previously the root).

Step 5 : Heapify Down:

- Swap the new root with its smallest child until the heap property is restored.

PROGRAM :

```
class MinHeap:
    def __init__(self):
        self.heap = []

    def insert(self, value):
        self.heap.append(value)
        self._heapify_up(len(self.heap) - 1)

    def delete_min(self):
        if len(self.heap) == 0:
            return None

        if len(self.heap) == 1:
            return self.heap.pop()

        root = self.heap[0]
```

```
        self.heap[0] = self.heap.pop()
        self._heapify_down(0)
        return root

    def delete_max(self):
        if len(self.heap) == 0:
            return None

        if len(self.heap) == 1:
            return self.heap.pop()

        max_child_idx = self._find_max_child(0)
        self.heap[0], self.heap[max_child_idx] = self.heap[max_child_idx], self.heap[0]
        deleted_max = self.heap.pop()
        self._heapify_down(max_child_idx)
        return deleted_max

    def _heapify_up(self, index):
        parent_index = (index - 1) // 2
        while index > 0 and self.heap[index] < self.heap[parent_index]:
            self.heap[index], self.heap[parent_index] = self.heap[parent_index], self.heap[index]
            index = parent_index
            parent_index = (index - 1) // 2

    def _heapify_down(self, index):
        left_child_index = 2 * index + 1
        right_child_index = 2 * index + 2
        smallest = index

        if left_child_index < len(self.heap) and self.heap[left_child_index] < self.heap[smallest]:
            smallest = left_child_index

        if right_child_index < len(self.heap) and self.heap[right_child_index] < self.heap[smallest]:
            smallest = right_child_index

        if smallest != index:
            self.heap[index], self.heap[smallest] = self.heap[smallest], self.heap[index]
            self._heapify_down(smallest)

    def _find_max_child(self, index):
        left_child_index = 2 * index + 1
        right_child_index = 2 * index + 2

        if right_child_index >= len(self.heap):
            return left_child_index

        return left_child_index if self.heap[left_child_index] > self.heap[right_child_index] else right_child_index
min_heap = MinHeap()
```

```
value1 = int(input("Enter a value: "))
min_heap.insert(value1)
value2 = int(input("Enter another value: "))
min_heap.insert(value2)
value3 = int(input("Enter one more value: "))
min_heap.insert(value3)
value4 = int(input("Enter another value: "))
min_heap.insert(value4)
value5 = int(input("Enter one more value: "))
min_heap.insert(value5)

print("Values in 'min_heap':", min_heap.heap)
print("Min Heap:", min_heap.heap)
print("Delete Min:", min_heap.delete_min())
print("Min Heap after Delete Min:", min_heap.heap)
print("Delete Max:", min_heap.delete_max())
print("Min Heap after Delete Max:", min_heap.heap)
```

OUTPUT :

```
Enter a value: 9
Enter another value: 3
Enter one more value: 6
Enter one more value: 1
Enter one more value: 5
Values in 'min_heap': [1, 3, 6, 9, 5]
Min Heap: [1, 3, 6, 9, 5]
Delete Min: 1
Min Heap after Delete Min: [3, 5, 6, 9]
Delete Max: 9
Min Heap after Delete Max: [6, 5, 3]
```

CONTENTS	MARKS ALLOTED	MARKS OBTAINED
PROGRAM AND EXECUTION	**15**	
VIVA-VOCE	**10**	
TOTAL	**25**	

RESULT :

Thus the program of the Implementation of min heap is executed and output is verified successfully.

Ex.no : 1(b)	SKEW HEAP (Priority Queue Operations)
Date :	

AIM :

To write a python program for skew heap (Priority Queue Operations).

ALGORITHM :

Step 1 : Insertion:

- Create a new node with the desired value.
- Merge the new node with the existing skew heap.

Step 2 : Delete Min (or Delete Max):

- Merge the left and right children of the root, effectively removing the root.
- The merged result becomes the new root.

PROGRAM :

```
class SkewNode:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None
class SkewHeap:
    def __init__(self):
        self.root = None
    def merge(self, h1, h2):
        if not h1:
            return h2
        if not h2:
            return h1

        if h1.value > h2.value:
            h1, h2 = h2, h1  # Swap h1 and h2

        h1.right, h1.left = h1.left, self.merge(h1.right, h2)
        return h1

    def insert(self, value):
        new_node = SkewNode(value)
        self.root = self.merge(self.root, new_node)

    def delete_min(self):
        if not self.root:
            return None
```

```
        min_value = self.root.value
        self.root = self.merge(self.root.left, self.root.right)
        return min_value

skew_heap = SkewHeap()
skew_heap.insert(4)
skew_heap.insert(2)
skew_heap.insert(7)
skew_heap.insert(6)

print("Skew Heap after insertion:", skew_heap.root.value)

min_val = skew_heap.delete_min()
print("Deleted Min:", min_val)
print("Skew Heap after deletion:", skew_heap.root.value)
min_val = skew_heap.delete_min()
print("Deleted Min:", min_val)
```

OUTPUT :

Skew Heap after insertion: 2
Deleted Min: 2
Skew Heap after deletion: 4
Deleted Min: 4

CONTENTS	MARKS ALLOTED	MARKS OBTAINED
PROGRAM AND EXECUTION	**15**	
VIVA-VOCE	**10**	
TOTAL	**25**	

RESULT :

Thus ,the program of the Implementation of skew heap is executed and output is verified.

Ex.no : 1(c)	FIBONACCI HEAP (Priority Queue Operations)
Date :	

AIM :

To write a python program for Fibonacci heap (Priority Queue Operations).

ALGORITHM :

Step 1 :Insertion:

- Create a new tree with the given value.
- Insert the new tree into the root list.

Step 2 :Delete Min:

- Identify the tree with the minimum root.
- Remove the root of that tree and merge its children with the root list.
- Consolidate the heap by combining trees with the same degree.

Step 3 :Delete Max:

- Same as Delete Min, but identify the tree with the maximum root.

PROGRAM :

```
# Fibonacci Heap in python
import math
# Creating fibonacci tree
class FibonacciTree:
    def __init__(self, value):
        self.value = value
        self.child = []
        self.order = 0

    # Adding tree at the end of the tree
    def add_at_end(self, t):
        self.child.append(t)
        self.order = self.order + 1

# Creating Fibonacci heap
class FibonacciHeap:
    def __init__(self):
        self.trees = []
        self.least = None
        self.count = 0
    # Insert a node
    def insert_node(self, value):
        new_tree = FibonacciTree(value)
```

```
        self.trees.append(new_tree)
        if (self.least is None or value < self.least.value):
            self.least = new_tree
        self.count = self.count + 1

    # Get minimum value
    def get_min(self):
        if self.least is None:
            return None
        return self.least.value

    # Extract the minimum value
    def extract_min(self):
        smallest = self.least
        if smallest is not None:
            for child in smallest.child:
                self.trees.append(child)
            self.trees.remove(smallest)
            if self.trees == []:
                self.least = None
            else:
                self.least = self.trees[0]
                self.consolidate()
            self.count = self.count - 1
            return smallest.value

    # Consolidate the tree
    def consolidate(self):
        aux = (floor_log(self.count) + 1) * [None]
        while self.trees != []:
            x = self.trees[0]
            order = x.order
            self.trees.remove(x)
            while aux[order] is not None:
                y = aux[order]
                if x.value > y.value:
                    x, y = y, x
                x.add_at_end(y)
                aux[order] = None
                order = order + 1
            aux[order] = x

        self.least = None
        for k in aux:
            if k is not None:
                self.trees.append(k)
                if (self.least is None
                        or k.value < self.least.value):
                    self.least = k
```

```
def floor_log(x):
    return math.frexp(x)[1] - 1

fibonacci_heap = FibonacciHeap()
fibonacci_heap.insert_node(7)
fibonacci_heap.insert_node(3)
fibonacci_heap.insert_node(17)
fibonacci_heap.insert_node(24)

print('the minimum value of the fibonacci heap: {}'.format(fibonacci_heap.get_min()))
print('the minimum value removed: {}'.format(fibonacci_heap.extract_min()))
```

OUTPUT :

The minimum value of the fibonacci heap: 3
The minimum value removed: 3

CONTENTS	MARKS ALLOTED	MARKS OBTAINED
PROGRAM AND EXECUTION	**15**	
VIVA-VOCE	**10**	
TOTAL	**25**	

RESULT :

Thus the program of the Implementation of fibonacci heap is executed and output is verified.

Ex.no : 2(a)	AVL TREES (Insertion ,Delete and search)
Date :	

AIM :

To Write a Python Program to implement a insert, delete , search a element by using the AVL tree properly.

ALGORITHM :

Step 1.Insertion:

- Perform standard BST insertion.
- Update height of each node from the newly inserted node to the root.
- Check balance factor to see if rotation is required.
- Right Rotation: If balance factor > 1 and new node is inserted into the left subtree of the left child.
- Left Rotation: If balance factor < -1 and new node is inserted into the right subtree of the right child.
- Left-Right Rotation: If balance factor > 1 and new node is inserted into the right subtree of the left child.
- Right-Left Rotation: If balance factor < -1 and new node is inserted into the left subtree of the right child.

Step 2.Deletion:

- Perform standard BST deletion.
- Update height of each node from the deleted node to the root.
- Check balance factor and perform necessary rotations if the tree becomes unbalanced.

Step 3.Search:

- Perform standard BST search.

PROGRAM :

```
class AVLNode:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
        self.height = 1
class AVLTree:
    def __init__(self):
        self.root = None
    def height(self, node):
        if not node:
            return 0
```

```
    return node.height

def balance(self, node):
    if not node:
        return 0
    return self.height(node.left) - self.height(node.right)

def right_rotate(self, y):
    x = y.left
    T2 = x.right

    x.right = y
    y.left = T2

    y.height = 1 + max(self.height(y.left), self.height(y.right))
    x.height = 1 + max(self.height(x.left), self.height(x.right))
    return x

def left_rotate(self, x):
    y = x.right
    T2 = y.left

    y.left = x
    x.right = T2

    x.height = 1 + max(self.height(x.left), self.height(x.right))
    y.height = 1 + max(self.height(y.left), self.height(y.right))
    return y

def insert(self, node, key):
    if not node:
        return AVLNode(key)
    if key < node.key:
        node.left = self.insert(node.left, key)
    else:
        node.right = self.insert(node.right, key)

    node.height = 1 + max(self.height(node.left), self.height(node.right))

    balance = self.balance(node)

    if balance > 1 and key < node.left.key:
        return self.right_rotate(node)
```

```
        if balance < -1 and key > node.right.key:
            return self.left_rotate(node)

        if balance > 1 and key > node.left.key:
            node.left = self.left_rotate(node.left)
            return self.right_rotate(node)

        if balance < -1 and key < node.right.key:
            node.right = self.right_rotate(node.right)
            return self.left_rotate(node)
        return node

    def delete(self, root, key):
        if not root:
            return root

        if key < root.key:
            root.left = self.delete(root.left, key)
        elif key > root.key:
            root.right = self.delete(root.right, key)
        else:
            if not root.left or not root.right:
                temp = root.left if root.left else root.right
                root = None
                return temp
            temp = self.min_value_node(root.right)
            root.key = temp.key
            root.right = self.delete(root.right, temp.key)

  if not root:
            return root

        root.height = 1 + max(self.height(root.left), self.height(root.right))

        balance = self.balance(root)

        if balance > 1 and self.balance(root.left) >= 0:
            return self.right_rotate(root)

        if balance < -1 and self.balance(root.right) <= 0:
            return self.left_rotate(root)

        if balance > 1 and self.balance(root.left) < 0:
```

```
            root.left = self.left_rotate(root.left)
            return self.right_rotate(root)

        if balance < -1 and self.balance(root.right) > 0:
            root.right = self.right_rotate(root.right)
            return self.left_rotate(root)

        return root

    def min_value_node(self, node):
        current = node
        while current.left:
            current = current.left
        return current

    def search(self, root, key):
        if not root or root.key == key:
            return root
        if root.key < key:
            return self.search(root.right, key)
        return self.search(root.left, key)

    def inorder(self, root):
        if root:
            self.inorder(root.left)
            print(root.key, end=' ')
            self.inorder(root.right)

avl = AVLTree()
avl.root = avl.insert(avl.root, 10)
avl.root = avl.insert(avl.root, 20)
avl.root = avl.insert(avl.root, 30)
avl.root = avl.insert(avl.root, 40)
avl.root = avl.insert(avl.root, 50)
avl.root = avl.insert(avl.root, 25)

print("Inorder traversal of AVL tree:")
avl.inorder(avl.root)
print()

avl.root = avl.delete(avl.root, 20)
print("Inorder traversal of AVL tree after deletion of 20:")
avl.inorder(avl.root)
```

```
print()

search_key = 30
if avl.search(avl.root, search_key):
    print(f"{search_key} found in AVL tree.")
else:
    print(f"{search_key} not found in AVL tree.")
```

OUTPUT :

```
Inorder traversal of AVL tree:
10 20 25 30 40 50
Inorder traversal of AVL tree after deletion of 20:
10 25 30 40 50
30 found in AVL tree
```

CONTENTS	MARKS ALLOTED	MARKS OBTAINED
PROGRAM AND EXECUTION	**15**	
VIVA-VOCE	**10**	
TOTAL	**25**	

RESULT :

Thus ,the program of the Implementation of AVL TREE is executed and output is verified.

Ex.no : 2(c)	B-TREES(Insertion ,Delete and search)
Date :	

AIM :

To Write a Python Program to insert, delete and search the given element elements by using the B- trees.

ALGORITHM :

Step 1. Insertion:

- Perform standard BST insertion.
- Update height of each node from the newly inserted node to the root.
- Check balance factor to see if rotation is required.
- Right Rotation: If balance factor > 1 and new node is inserted into the left subtree of the left child.
- Left Rotation: If balance factor < -1 and new node is inserted into the right subtree of the right child.
- Left-Right Rotation: If balance factor > 1 and new node is inserted into the right subtree of the left child.
- Right-Left Rotation: If balance factor < -1 and new node is inserted into the left subtree of the right child.

Step 2. Deletion:

- Perform standard BST deletion.
- Update height of each node from the deleted node to the root.
- Check balance factor and perform necessary rotations if the tree becomes unbalanced.

Step 3. Search:

- Perform standard BST search.

PROGRAM :

```
class BTreeNode:
    def __init__(self, keys=[], children=[], is_leaf=True, max_keys=4):
        self.keys = keys
        self.children = children
        self.is_leaf = is_leaf
        if not max_keys:
            # Default value for max_keys
            self.max_keys = 4
        else:
            self.max_keys = max_keys
class BTree:
    def __init__(self, max_keys=4):
        self.root = BTreeNode(max_keys=max_keys)
```

```
def search(self, node, key):
    i = 0
    while i < len(node.keys) and key > node.keys[i]:
        i += 1
    if i < len(node.keys) and key == node.keys[i]:
        return node, i
    if node.is_leaf:
        return None, None
    return self.search(node.children[i], key)

def insert(self, key):
    root = self.root
    if len(root.keys) == root.max_keys:
        new_root = BTreeNode(keys=[root.keys.pop(len(root.keys)//2)], children=[root])
        self.split_child(new_root, 0)
        self.root = new_root
        root = new_root
    self.insert_non_full(root, key)

def insert_non_full(self, node, key):
    i = len(node.keys) - 1
    if node.is_leaf:
        node.keys.append(None)
        while i >= 0 and key < node.keys[i]:
            node.keys[i+1] = node.keys[i]
            i -= 1
        node.keys[i+1] = key
    else:
        while i >= 0 and key < node.keys[i]:
            i -= 1
        i += 1
        if len(node.children[i].keys) == node.children[i].max_keys:
            self.split_child(node, i)
            if key > node.keys[i]:
                i += 1
        self.insert_non_full(node.children[i], key)

def split_child(self, parent, i):
    node_to_split = parent.children[i]
    new_node = BTreeNode(max_keys=node_to_split.max_keys, is_leaf=node_to_split.is_leaf)

    parent.keys.insert(i, node_to_split.keys[len(node_to_split.keys)//2])
    parent.children.insert(i+1, new_node)

    new_node.keys = node_to_split.keys[len(node_to_split.keys)//2+1:]
    node_to_split.keys = node_to_split.keys[:len(node_to_split.keys)//2]

    if not node_to_split.is_leaf:
```

```
            new_node.children = node_to_split.children[len(node_to_split.keys)+1:]
            node_to_split.children = node_to_split.children[:len(node_to_split.keys)+1]

    def delete(self, key):
        self.delete_recursive(self.root, key)

    def delete_recursive(self, node, key):
        i = 0
        while i < len(node.keys) and key > node.keys[i]:
            i += 1

        if i < len(node.keys) and key == node.keys[i]:
            if node.is_leaf:
                del node.keys[i]
            else:
                # Replace with predecessor and delete predecessor from child
                if len(node.children[i]) >= node.max_keys/2:
                    pred = self.get_predecessor(node, i)
                    node.keys[i] = pred.keys.pop()
                # Replace with successor and delete successor from child
                elif len(node.children[i+1]) >= node.max_keys/2:
                    succ = self.get_successor(node, i)
                    node.keys[i] = succ.keys.pop(0)
                # Merge nodes
                else:
                    self.merge(node, i)
                    self.delete_recursive(node.children[i], key)
        else:
            if node.is_leaf:
                return
            elif len(node.children[i]) == node.max_keys/2:
                self.fix_borrow_or_merge(node, i)
            self.delete_recursive(node.children[i], key)

    def get_predecessor(self, node, i):
        curr = node.children[i]
        while not curr.is_leaf:
            curr = curr.children[-1]
        return curr

    def get_successor(self, node, i):
        curr = node.children[i+1]
        while not curr.is_leaf:
            curr = curr.children[0]
        return curr

    def merge(self, node, i):
        child = node.children[i]
        sibling = node.children[i+1]
```

```
        child.keys.append(node.keys[i])
        child.keys += sibling.keys

        if not child.is_leaf:
            child.children += sibling.children

        del node.keys[i]
        del node.children[i+1]

    def fix_borrow_or_merge(self, node, i):
        if i > 0 and len(node.children[i-1]) > node.max_keys/2:
            # Borrow from left sibling
            left_sibling = node.children[i-1]
            child = node.children[i]
            if child.is_leaf:
                child.keys.insert(0, node.keys[i-1])
                node.keys[i-1] = left_sibling.keys.pop()
            else:
                child.keys.insert(0, node.keys[i-1])
                node.keys[i-1] = left_sibling.keys.pop(-1)
                child.children.insert(0, left_sibling.children.pop(-1))
        elif i < len(node.children) - 1 and len(node.children[i+1]) > node.max_keys/2:
            # Borrow from right sibling
            right_sibling = node.children[i+1]
            child = node.children[i]
            if child.is_leaf:
                child.keys.append(node.keys[i])
                node.keys[i] = right_sibling.keys.pop(0)
            else:
                child.keys.append(node.keys[i])
                node.keys[i] = right_sibling.keys.pop(0)
                child.children.append(right_sibling.children.pop(0))
        else:
            # Merge with sibling
            if i > 0:
                self.merge(node, i-1)
                del node.keys[i-1]
            else:
                self.merge(node, i)
                del node.keys[i]

    def inorder_traversal(self, node):
        if node:
            i = 0
            while i < len(node.keys):
                self.inorder_traversal(node.children[i])
                print(node.keys[i], end=' ')
                i += 1
            self.inorder_traversal(node.children[i])
```

```
btree = BTree()
btree.insert(10)

btree.insert(20)
btree.insert(5)
btree.insert(6)
btree.insert(12)
btree.insert(30)
btree.insert(7)
btree.insert(17)

print("Inorder traversal of B-Tree:")
btree.inorder_traversal(btree.root)
print()
btree.delete(12)
print("Inorder traversal of B-Tree after deletion of 12:")
btree.inorder_traversal(btree.root)
print()
search_key = 17
if btree.search(btree.root, search_key):
    print(f"{search_key} found in B-Tree.")
else:
    print(f"{search_key} not found in B-Tree.")
```

OUTPUT:

Inorder traversal of B-Tree:

5 6 7 10 12 17 20 30

Inorder traversal of B-Tree after deletion of 12:

CONTENTS	MARKS ALLOTED	MARKS OBTAINED
PROGRAM AND EXECUTION	**15**	
VIVA-VOCE	**10**	
TOTAL	**25**	

RESULT :

Thus ,the program of the Implementation of B-TREE is executed and output is verified.

Ex.no : 2(d)	RED-BLACK TREE
Date :	

AIM :

To Write a Python Program to insert, delete and search the given set of elements by using the Red- Black tree.

ALGORITHM :

Step 1. Insertion Operation:

- Perform standard BST insertion.
- After insertion, fix any violations of the Red-Black tree properties by rotating and recoloring nodes if necessary

There are four cases for fixing violations:

- Case 1: The uncle of the newly inserted node is red.
- Case 2: The uncle of the newly inserted node is black and the newly inserted node is a right child of a left child or vice versa.
- Case 3: The uncle of the newly inserted node is black and the newly inserted node is a left child of a left child or a right child of a right child.
- Case 4: The uncle of the newly inserted node is black and the newly inserted node is a left child of a right child or a right child of a left child.

Step 2 . Deletion Operation:

- Perform standard BST deletion.
- After deletion, fix any violations of the Red-Black tree properties by rotating and recoloring nodes if necessary.

There are three cases for fixing violations after deletion:

- Case 1: Sibling is red.
- Case 2: Sibling is black and both of its children are black.
- Case 3: Sibling is black, its left child is red, and its right child is black.
- Case 4: Sibling is black, its right child is red.
- Case 5: Sibling is black, its left child is black, and its right child is red.

Step 3 . Search Operation:

- Perform standard BST search.

PROGRAM :

```
RED = True
BLACK = False
class Node:
    def __init__(self, key, color=RED):
        self.key = key
        self.left = None
```

```
        self.right = None
        self.parent = None
        self.color = color
class RedBlackTree:
    def __init__(self):
        self.nil = Node(None, color=BLACK)
        self.root = self.nil
    def left_rotate(self, x):
        y = x.right
        x.right = y.left
        if y.left != self.nil:
            y.left.parent = x
        y.parent = x.parent
        if x.parent == self.nil:
            self.root = y
        elif x == x.parent.left:
            x.parent.left = y
        else:
            x.parent.right = y
        y.left = x
        x.parent = y
    def right_rotate(self, y):
        x = y.left
        y.left = x.right
        if x.right != self.nil:
            x.right.parent = y
        x.parent = y.parent
        if y.parent == self.nil:
            self.root = x
        elif y == y.parent.right:
            y.parent.right = x
        else:
            y.parent.left = x
        x.right = y
        y.parent = x
    def insert(self, key):
        new_node = Node(key)
        y = self.nil
        x = self.root
        while x != self.nil:
            y = x
            if new_node.key < x.key:
                x = x.left
            else:
                x = x.right
        new_node.parent = y
        if y == self.nil:
         self.root = new_node
        elif new_node.key < y.key:
```

```
            y.left = new_node
        else:
            y.right = new_node
        new_node.left = self.nil
        new_node.right = self.nil
        new_node.color = RED
        self.insert_fixup(new_node)

    def insert_fixup(self, z):
        while z.parent.color == RED:
            if z.parent == z.parent.parent.left:
                y = z.parent.parent.right
                if y.color == RED:
                    z.parent.color = BLACK
                    y.color = BLACK
                    z.parent.parent.color = RED
                    z = z.parent.parent
                else:
                    if z == z.parent.right:
                        z = z.parent
                        self.left_rotate(z)
                    z.parent.color = BLACK
                    z.parent.parent.color = RED
                    self.right_rotate(z.parent.parent)
            else:
                y = z.parent.parent.left
                if y.color == RED:
                    z.parent.color = BLACK
                    y.color = BLACK
                    z.parent.parent.color = RED
                    z = z.parent.parent
                else:
                    if z == z.parent.left:
                        z = z.parent
                        self.right_rotate(z)
                    z.parent.color = BLACK
                    z.parent.parent.color = RED
                    self.left_rotate(z.parent.parent)
        self.root.color = BLACK

    def transplant(self, u, v):
        if u.parent == self.nil:
            self.root = v
        elif u == u.parent.left:
            u.parent.left = v
        else:
            u.parent.right = v
        v.parent = u.parent
    def minimum(self, x):
```

```
        while x.left != self.nil:
            x = x.left
        return x

    def delete_node(self, z):
        y = z
        y_original_color = y.color
        if z.left == self.nil:
            x = z.right
            self.transplant(z, z.right)
        elif z.right == self.nil:
            x = z.left
            self.transplant(z, z.left)
        else:
            y = self.minimum(z.right)
            y_original_color = y.color
            x = y.right
            if y.parent == z:
                x.parent = y
            else:
                self.transplant(y, y.right)
                y.right = z.right
                y.right.parent = y
            self.transplant(z, y)
            y.left = z.left
            y.left.parent = y
            y.color = z.color
        if y_original_color == BLACK:
            self.delete_fixup(x)

    def delete_fixup(self, x):
        while x != self.root and x.color == BLACK:
            if x == x.parent.left:
                w = x.parent.right

                if w.color == RED:
                    w.color = BLACK
                    x.parent.color = RED
                    self.left_rotate(x.parent)
                    w = x.parent.right
                if w.left.color == BLACK and w.right.color == BLACK:
                    w.color = RED
                    x = x.parent
                else:

                    if w.right.color == BLACK:
                        w.left.color = BLACK
                        w.color = RED
                        self.right_rotate(w)
```

```
                w = x.parent.right
            w.color = x.parent.color
            x.parent.color = BLACK
            w.right.color = BLACK
            self.left_rotate(x.parent)
            x = self.root
        else:
            w = x.parent.left
            if w.color == RED:
                w.color = BLACK
                x.parent.color = RED
                self.right_rotate(x.parent)
                w = x.parent.left
            if w.right.color == BLACK and w.left.color == BLACK:

                w.color = RED
                x = x.parent
            else:
                if w.left.color == BLACK:
                    w.right.color = BLACK
                    w.color = RED
                    self.left_rotate(w)
                    w = x.parent.left
                w.color = x.parent.color
                x.parent.color = BLACK
                w.left.color = BLACK
                self.right_rotate(x.parent)
                x = self.root
        x.color = BLACK

    def search(self, key):
        return self.search_recursive(self.root, key)

    def search_recursive(self, node, key):
        if node == self.nil or key == node.key:
            return node

        if key < node.key:
              return self.search_recursive(node.left, key)
             return self.search_recursive(node.right, key)

    def inorder_traversal(self, node):
        if node != self.nil:

            self.inorder_traversal(node.left)
            print(node.key, end=' ')
            self.inorder_traversal(node.right)
rb_tree = RedBlackTree()
rb_tree.insert(10)
```

```
rb_tree.insert(20)
rb_tree.insert(30)
rb_tree.insert(40)
rb_tree.insert(50)
rb_tree.insert(15)
rb_tree.insert(25)

print("Inorder traversal of Red-Black Tree:")
rb_tree.inorder_traversal(rb_tree.root)
print()

rb_tree.delete_node(rb_tree.search(20))
print("Inorder traversal of Red-Black Tree after deletion of 20:")
rb_tree.inorder_traversal(rb_tree.root)
print()

search_key = 30
if rb_tree.search(search_key):
    print(f"{search_key} found in Red-Black Tree.")
else:
    print(f"{search_key} not found in Red-Black Tree.")
```

OUTPUT :

Inorder traversal of Red-Black Tree:

10 15 20 25 30 40 50

Inorder traversal of Red-Black Tree after deletion of 20:

10 15 25 30 40 50
30 found in Red-Black Tree

CONTENTS	MARKS ALLOTED	MARKS OBTAINED
PROGRAM AND EXECUTION	**15**	
VIVA-VOCE	**10**	
TOTAL	**25**	

RESULT :

Thus the program of the Implementation of RED-BLACK TREE is executed and output is verified.

Ex.no : 3	CONVEX HULL
Date :	

AIM :

To implement the convex hull algorithm in Python, you can use the Graham Scan algorithm.

ALGORITHM :

STEP 1: Find the point with the lowest y-coordinate (and leftmost point if there's a tie).
STEP 2: Sort the remaining points by polar angle with respect to the starting point.
STEP 3: Initialize an empty stack and push the first two points onto it.
STEP 4:. Process the remaining points, adding them to the stack if they form a left turn with the last two points.
STEP 5: Return the points on the stack as the convex hull.

PROGRAM :

```
def orientation(p, q, r):
    """Find orientation of triplet (p, q, r)."""
    val = (q[1] - p[1]) * (r[0] - q[0]) - (q[0] - p[0]) * (r[1] - q[1])
    if val == 0:
        return 0  # Collinear
    return 1 if val > 0 else 2  # Clockwise or Counterclockwise

def convex_hull(points):
    """Compute the convex hull of a set of points."""
    n = len(points)
    if n < 3:
        return []

    # Find the leftmost point
    l = min(range(n), key=lambda x: points[x][0])

    hull = []
    p = l
    q = 0
    while True:
        hull.append(p)
        q = (p + 1) % n
        for i in range(n):
            if orientation(points[p], points[i], points[q]) == 2:
                q = i
        p = q
        if p == l:
            break
    return [points[i] for i in hull]
# Example usage:
```

```
points = [(0, 3), (1, 1), (2, 2), (4, 4), (0, 0), (1, 2), (3, 1), (3, 3)]
print(convex_hull(points))
```

OUTPUT :

[(0,3), (0,0), (3,1), (4,4)]

CONTENTS	MARKS ALLOTED	MARKS OBTAINED
PROGRAM AND EXECUTION	**15**	
VIVA-VOCE	**10**	
TOTAL	**25**	

RESULT :

Thus ,the program of the Implementation of convex hull executed and output is verified.

Ex.no : 4	TOPOLOGICAL SORT
Date :	

AIM :

To perform topological sorting in Python, you can implement a depth-first search (DFS) based algorithm

ALGORITHM :

STEP 1: Start with a graph represented as an adjacency list.
STEP 2: Perform a depth-first search (DFS) on the graph.
STEP 3: During the DFS traversal, mark nodes as visited and recursively explore adjacent nodes.
STEP 4: When a node has no unvisited neighbors, add it to a stack or list.
STEP 5: After completing the DFS traversal, reverse the order of the nodes obtained from the stack or list to get the topological sorting.

PROGRAM :

```
from collections import defaultdict
class Graph:
    def _init_(self, vertices):
        self.graph = defaultdict(list)
        self.V = vertices

    def add_edge(self, u, v):
        self.graph[u].append(v)

    def topological_sort_util(self, v, visited, stack):
        visited[v] = True
        for i in self.graph[v]:
            if not visited[i]:
                self.topological_sort_util(i, visited, stack)
        stack.append(v)

    def topological_sort(self):
        visited = [False] * self.V
        stack = []

        for i in range(self.V):
            if not visited[i]:
                self.topological_sort_util(i, visited, stack)

        return stack[::-1]
# Example usage:
g = Graph(6)
g.add_edge(5, 2)
g.add_edge(5, 0)
```

```
g.add_edge(4, 0)
g.add_edge(4, 1)
g.add_edge(2, 3)
g.add_edge(3, 1)

print("Topological Sort:")
print(g.topological_sort())
```

OUTPUT :

Topological Sort: ['A', 'C', 'E', 'B', 'D', 'F']

CONTENTS	MARKS ALLOTED	MARKS OBTAINED
PROGRAM AND EXECUTION	**15**	
VIVA-VOCE	**10**	
TOTAL	**25**	

RESULT :

Thus the program of the Implementation of topological sort is executed and output isverified.

Ex.no : 5	GRAPH SEARCH ALGORITHM
Date :	

AIM :

To perform topological sorting in Python, you can implement a depth-first search (DFS) based algorithm.

ALGORITHM :

STEP 1: Start with an empty data structure to track visited nodes.
STEP 2: Start with an empty data structure for processing nodes (e.g., a stack for DFS or a queue for BFS).
STEP 3: Add the starting node to the processing data structure and mark it as visited.
STEP 4:. While there are nodes in the processing data structure:

- Pop or dequeue a node.
- Process the node (e.g., print it or perform an operation).
- Mark the node as visited.

STEP 5: Terminate when there are no more nodes to process.

PROGRAM :

BFS:

```
from collections import defaultdict, deque
def bfs(graph, start):
    visited = set()
    queue = deque([start])
    visited.add(start)

    while queue:
        node = queue.popleft()
        print(node, end=" ")

        for neighbor in graph[node]:
            if neighbor not in visited:
                queue.append(neighbor)
                visited.add(neighbor)
# Example usage:
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],

    'F': ['C', 'E']
}

print("BFS traversal:")
```

```
bfs(graph, 'A')
```

DFS:

```
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    print(start, end=" ")
    visited.add(start)

    for neighbor in graph[start]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)

# Example usage:
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}

print("DFS traversal:")

dfs(graph, 'A')
```

OUTPUT :

BFS traversal:
A B C D E F

CONTENTS	MARKS ALLOTED	MARKS OBTAINED
PROGRAM AND EXECUTION	15	
VIVA-VOCE	10	
TOTAL	25	

RESULT :

Thus the program of the Implementation of graph search algorithms is executed and output is verified.

Ex.no : 7	IMPLEMENTATION OF RANDOMIZED ALGORIT IMPLEMENTATION OF RANDOMIZED ALGORITHMS HMS
Date :	

AIM :

The aim of implementing network flow and linear programming problems is to model and solve optimization challenges. In network flow, the goal is to find the maximumflow in a network, while linear programming involves optimizing a linear objective functionsubject to linear equality and inequality constraints.

ALGORITHM :

Model the Network Flow Problem:

STEP 1: Define the graph representing the network.

STEP 2: Assign capacities to edges representing flow constraints.

STEP 3: Define source and sink nodes.

STEP 4: Solve the Network Flow Problem.

STEP 5: Use algorithms like Ford-Fulkerson or Edmonds-Karp to find the maximum flow.

Model the Linear Programming Problem:

STEP 1: Define decision variables.

STEP 2: Formulate the objective function.

STEP 3: Add constraints.

STEP 4: Solve the Linear Programming Problem.

STEP 5: Use libraries like scipy.optimize.linprog to find the optimal solution.

PROGRAM :

NETWORK FLOW PROBLEM:

import networkx as nx

Step 1: Model the Network Flow Problem G = nx.DiGraph()

G.add_edge('source', 'A', capacity=10) G.add_edge('source', 'B', capacity=5) G.add_edge('A', 'C', capacity=9) G.add_edge('A', 'B', capacity=3) G.add_edge('B', 'C', capacity=7) G.add_edge('B', 'sink', capacity=8) G.add_edge('C', 'sink', capacity=12)

Step 2: Solve the Network Flow Problem

max_flow_value, flow_dict = nx.maximum_flow(G, 'source', 'sink') print("Maximum Flow Value:", max_flow_value)

print("Flow Dict:", flow_dict)

LINEAR PROGRAMMING PROBLEM:

from scipy.optimize import linprog

# Step 3: Model the Linear Programming Problem
c = [-3, -2] # Coefficients of the objective function (to minimize)
A = [[1, 1], # Coefficients of inequality constraints [1, 0], [0, 1]]

b = [10, 8, 5] # Right-hand side of inequality constraints x0_bounds = (0, None) # Bounds for decision variables x1_bounds = (0, None)

# Step 4: Solve the Linear Programming Problem
res = linprog(c, A_ub=A, b_ub=b, bounds=[x0_bounds, x1_bounds], method='highs') print("Optimal Solution:", res.x)
print("Optimal Objective Value:", res.fun)

OUTPUT :

NETWORK FLOW PROBLEM:

Maximum Flow Value: 15
Flow Dict: {'source': {'A': 10, 'B': 5}, 'A': {'C': 9, 'B': 1}, 'B': {'C': 5, 'sink': 5}, 'C': {'sink': 12}, 'sink': {}}

LINEAR PROGRAMMING PROBLEM:

Optimal Solution: [4. 1.]
Optimal Objective Value: -14.0

CONTENTS	MARKS ALLOTED	MARKS OBTAINED
PROGRAM AND EXECUTION	15	
VIVA-VOCE	10	
TOTAL	25	

RESULT :

Thus , the output represent the maximum flow value in the network and the optimal solution to the linear programming problem are executed and output is verified.