A Laboratory Manual for

Advanced Data Structures (ME0100061)

M.E. Semester 1
Software Engineering
(Computer Engineering)

Institute logo

L. D. College of Engineering, Ahmedabad,

Gujarat

L. D. College of Engineering, Ahmedabad

Certificate

This is to certify that Mr./Ms. _____ _____Enrollment No. _____of M.E. Semester ___Software Engineering (Computer Engineering) of this Institute (GTU Code: ____) has satisfactorily completed the Practical / Tutorial work for the subject **Advanced Data Structure (ME0100061)** for the academic year 2025-2026.

Place: _____

Date: _____

Name and Sign of Faculty member

Head of the Department

Practical List

Sr. No.	Objective(s) of Experiment
1.	Write a program which creates Binary Search Tree. And also implement recursive and non-recursive tree traversing methods inorder, preorder and post-order for the BST.
2.	Write a program to implement any two hashing methods. Use any one of the hashing method to implement Insert, Delete and Search operations for Hash Table Management.
3.	Explain Dictionary as an Abstract Data Type. Implement Dictionary using suitable Data Structure.
4.	Write a program which creates AVLTree. Implement Insert and Delete Operations in AVL Tree. Note that each time the tree must be balanced.
5.	Implement Red-Black Tree.
6.	Implement 2-3 Tree.
7.	Implement B Tree.
8.	Implement a program for String Matching using Boyer-Moore Algorithm on a text file content.
9.	Implement a program for String Matching using Knuth-Morris-Pratt Algorithm on a text file content.
10.	Implement Huffman-Coding Method. Show the result with suitable example.
11.	Implement One Dimensional and Two Dimensional Range Searching in any language.
12.	Write a program which creates Priority Search Tree. Implement Insert and Search Operations in this Tree.
	ASSIGNMENT

Index

Sr. No.	Objective(s) of Experiment	Pg no.	Date of performance	Date of submission	Assessment Marks	Sign. Of Teacher with date	Remarks
1	Write a program which creates Binary Search Tree. And also implement recursive and non-recursive tree traversing methods inorder, preorder and post-order for the BST.						
2	Write a program to implement any two hashing methods. Use any one of the hashing method to implement Insert, Delete and Search operations for Hash Table Management.						
3	Explain Dictionary as an Abstract Data Type. Implement Dictionary using suitable Data Structure.						
4	Write a program which creates AVLTree. Implement Insert and Delete Operations in AVL Tree. Note that each time the tree must be balanced.						
5	Implement Red-Black Tree.						
6	Implement 2-3 Tree.						
7	Implement B Tree.						
8	Implement a program for String Matching using Boyer-Moore Algorithm on a text file content.						
9	Implement a program for String Matching using Knuth-Morris-Pratt Algorithm on a text file content.						
10	Implement Huffman-Coding Method. Show the result with suitable example.						

11	Implement One Dimensional and Two Dimensional Range Searching in any language.						
12	Write a program which creates Priority Search Tree. Implement Insert and Search Operations in this Tree.						
	ASSIGNMENT						
Total							

(Progressive Assessment Sheet)

PRACTICAL-1

Write a program which creates Binary Search Tree. And also implement recursive and non-recursive tree traversing methods inorder, preorder and post-order for the BST.

Code:

```
# Binary Search Tree Implementation with Traversals

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

class BST:
    def __init__(self):
        self.root = None

    # ---------- Insert ----------
    def insert(self, data):
        self.root = self._insert(self.root, data)

    def _insert(self, root, data):
        if root is None:
            return Node(data)

        if data < root.data:
            root.left = self._insert(root.left, data)
        elif data > root.data:
            root.right = self._insert(root.right, data)

        return root

    #
==========================================
==========
    # RECURSIVE TRAVERSALS
    #
==========================================
==========

    def inorder_recursive(self, root):
        if root:
            self.inorder_recursive(root.left)
            print(root.data, end=" ")
            self.inorder_recursive(root.right)
```

```
    def preorder_recursive(self, root):
        if root:
            print(root.data, end=" ")
            self.preorder_recursive(root.left)
            self.preorder_recursive(root.right)

    def postorder_recursive(self, root):
        if root:
            self.postorder_recursive(root.left)
            self.postorder_recursive(root.right)
            print(root.data, end=" ")

    #
==========================================
==========
    # NON-RECURSIVE TRAVERSALS
    #
==========================================
==========

    # ---------- Inorder ----------
    def inorder_non_recursive(self):
        stack = []
        curr = self.root

        while curr or stack:
            while curr:
                stack.append(curr)
                curr = curr.left

            curr = stack.pop()
            print(curr.data, end=" ")
            curr = curr.right

    # ---------- Preorder ----------
    def preorder_non_recursive(self):
        if not self.root:
            return

        stack = [self.root]

        while stack:
            curr = stack.pop()
            print(curr.data, end=" ")

            if curr.right:
                stack.append(curr.right)
            if curr.left:
```

```
            stack.append(curr.left)

    # ---------- Postorder ----------
    def postorder_non_recursive(self):
        if not self.root:
            return

        stack1 = [self.root]
        stack2 = []

        while stack1:
            curr = stack1.pop()
            stack2.append(curr)

            if curr.left:
                stack1.append(curr.left)
            if curr.right:
                stack1.append(curr.right)

        while stack2:
            print(stack2.pop().data, end=" ")

#
==========================================
==========
# DRIVER CODE
#
==========================================
==========

if __name__ == "__main__":
    bst = BST()

    n = int(input("Enter number of nodes: "))
    print("Enter values:")
    for _ in range(n):
        bst.insert(int(input()))

    print("\nRecursive Traversals")
    print("Inorder   :", end=" ")
    bst.inorder_recursive(bst.root)

    print("\nPreorder  :", end=" ")
    bst.preorder_recursive(bst.root)

    print("\nPostorder :", end=" ")
    bst.postorder_recursive(bst.root)
```

```
print("\n\nNon-Recursive Traversals")
print("Inorder   :", end=" ")
bst.inorder_non_recursive()

print("\nPreorder  :", end=" ")
bst.preorder_non_recursive()

print("\nPostorder :", end=" ")
bst.postorder_non_recursive()
```

Output:

PS C:\Users\BOUNA\Desktop\ADS> python practical1.py
Enter number of nodes: 3
Enter values:
1
2
3

Recursive Traversals
Inorder   : 1 2 3
Preorder : 1 2 3
Postorder : 3 2 1

Non-Recursive Traversals
Inorder   : 1 2 3
Preorder : 1 2 3
Postorder : 3 2 1

PRACTICAL-2

Write a program to implement any two hashing methods. Use any one of the hashing method to implement Insert, Delete and Search operations for Hash Table Management.

Code:

```
#
====================
====================
==
# TWO HASHING
METHODS
#
====================
====================
==

def division_hash(key,
table_size):
    return key % table_size


def mid_square_hash(key,
table_size):
    square = key * key
    mid = (square // 10) %
table_size
    return mid


#
====================
====================
==
# HASH TABLE USING
CHAINING (Division
Method)
#
====================
====================
==

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

```
range(size)]

    # ---------- Insert ----------
    def insert(self, key):
        index =
division_hash(key, self.size)
        if key not in
self.table[index]:

self.table[index].append(key)
            print(f"{key} inserted
at index {index}")
        else:
            print("Key already
exists")

    # ---------- Search ----------
    def search(self, key):
        index =
division_hash(key, self.size)
        if key in
self.table[index]:
            print(f"{key} found at
index {index}")
        else:
            print(f"{key} not
found")

    # ---------- Delete ----------
    def delete(self, key):
        index =
division_hash(key, self.size)
        if key in
self.table[index]:

self.table[index].remove(key)
            print(f"{key} deleted
from index {index}")
        else:
            print(f"{key} not
found")

    # ---------- Display ----------
    def display(self):
        print("\nHash Table:")
        for i in range(self.size):
            print(f"{i} :
{self.table[i]}")
```

```
#
==========================================
==
# MENU-DRIVEN DRIVER
CODE
#
==========================================
==

if __name__ == "__main__":
    size = int(input("Enter hash
table size: "))
    ht = HashTable(size)

    while True:
        print("\n1.Insert
2.Search  3.Delete  4.Display
5.Exit")
        choice = int(input("Enter
choice: "))

        if choice == 1:
            key = int(input("Enter
key to insert: "))
            ht.insert(key)

        elif choice == 2:
            key = int(input("Enter
key to search: "))
            ht.search(key)

        elif choice == 3:
            key = int(input("Enter
key to delete: "))
            ht.delete(key)

        elif choice == 4:
            ht.display()

        elif choice == 5:
            print("Exiting
program...")
            break
```

```
        else:
            print("Invalid choice")
```

Output:

```
PS C:\Users\BOUNA\Desktop\ADS> python practical2.py
Enter hash table size: 3

1.Insert  2.Search  3.Delete  4.Display  5.Exit
Enter choice: 1
Enter key to insert: 2
2 inserted at index 2

1.Insert  2.Search  3.Delete  4.Display  5.Exit
Enter choice: 2
Enter key to search: 1
1 not found

1.Insert  2.Search  3.Delete  4.Display  5.Exit
Enter choice: 3
Enter key to delete: 3
3 not found

1.Insert  2.Search  3.Delete  4.Display  5.Exit
Enter choice: 4

Hash Table:
0 : []
1 : []
2 : [2]

1.Insert  2.Search  3.Delete  4.Display  5.Exit
Enter choice:
```

PRACTICAL-3

Explain Dictionary as an Abstract Data Type. Implement Dictionary using suitable Data Structure.

Code:

```
# ============================================
# Dictionary ADT using Hash Table (Chaining)
# ============================================

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

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

    def insert(self, key, value):
        index = self.hash_function(key)
        for pair in self.table[index]:
            if pair[0] == key:
                pair[1] = value
                return
        self.table[index].append([key, value])

    def search(self, key):
        index = self.hash_function(key)
        for pair in self.table[index]:
            if pair[0] == key:
                return pair[1]
        return None

    def delete(self, key):
        index = self.hash_function(key)
        for pair in self.table[index]:
            if pair[0] == key:
                self.table[index].remove(pair)
                return

    def display(self):
        print("\nDictionary Contents:")
        for bucket in self.table:
            for pair in bucket:
                print(f"{pair[0]} : {pair[1]}")

# ============================================
# DRIVER CODE WITH RANDOM SAMPLE DATA
# ============================================

if __name__ == "__main__":
    d = DictionaryADT(5)
```

```
# Random example data
d.insert("name", "Alice")
d.insert("age", 21)
d.insert("city", "Delhi")
d.insert("course", "BTech")
d.insert("roll_no", 105)

d.display()

print("\nSearch Example:")
print("city →", d.search("city"))

print("\nAfter Update:")
d.insert("age", 22)
d.display()

print("\nAfter Delete:")
d.delete("course")
d.display()
```

Output:

PS C:\Users\BOUNA\Desktop\ADS> python practical3.py

Dictionary Contents:
roll_no : 105
course : BTech
age : 21
city : Delhi
name : Alice

Search Example:
city → Delhi

After Update:

Dictionary Contents:
roll_no : 105
course : BTech
age : 22
city : Delhi
name : Alice

After Delete:

Dictionary Contents:
roll_no : 105
age : 22
city : Delhi
name : Alice

PRACTICAL-4

Write a program which creates AVLTree. Implement Insert and Delete Operations in AVL Tree. Note that each time the tree must be balanced.

Code:

```
# ============================================
# AVL TREE IMPLEMENTATION
# ============================================

class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
        self.height = 1

class AVLTree:

    # ---------- Get Height ----------
    def get_height(self, root):
        return root.height if root else 0

    # ---------- Get Balance Factor ----------
    def get_balance(self, root):
        if not root:
            return 0
        return self.get_height(root.left) - self.get_height(root.right)

    # ---------- Right Rotation ----------
    def right_rotate(self, y):
        x = y.left
        T2 = x.right

        x.right = y
        y.left = T2

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

        return x

    # ---------- Left Rotation ----------
    def left_rotate(self, x):
        y = x.right
        T2 = y.left

        y.left = x
```

```
    x.right = T2

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

    return y

# ================================
# INSERT OPERATION
# ================================

def insert(self, root, key):

    if not root:
        return Node(key)

    if key < root.key:
        root.left = self.insert(root.left, key)
    elif key > root.key:
        root.right = self.insert(root.right, key)
    else:
        return root

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

    balance = self.get_balance(root)

    # LL Case
    if balance > 1 and key < root.left.key:
        return self.right_rotate(root)

    # RR Case
    if balance < -1 and key > root.right.key:
        return self.left_rotate(root)

    # LR Case
    if balance > 1 and key > root.left.key:
        root.left = self.left_rotate(root.left)
        return self.right_rotate(root)

    # RL Case
    if balance < -1 and key < root.right.key:
        root.right = self.right_rotate(root.right)
        return self.left_rotate(root)

    return root

# ---------- Minimum Value Node ----------
def get_min_value_node(self, root):
```

```
    if root is None or root.left is None:
        return root
    return self.get_min_value_node(root.left)

# ===================================
# DELETE OPERATION
# ===================================

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 root.left is None:
            return root.right
        elif root.right is None:
            return root.left

        temp = self.get_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.get_height(root.left),
                          self.get_height(root.right))

    balance = self.get_balance(root)

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

    # LR Case
    if balance > 1 and self.get_balance(root.left) < 0:
        root.left = self.left_rotate(root.left)
        return self.right_rotate(root)

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

    # RL Case
    if balance < -1 and self.get_balance(root.right) > 0:
        root.right = self.right_rotate(root.right)
        return self.left_rotate(root)
```

```
        return root

    # ---------- Inorder Traversal ----------
    def inorder(self, root):
        if root:
            self.inorder(root.left)
            print(root.key, end=" ")
            self.inorder(root.right)

# ============================================
# DRIVER CODE
# ============================================

if __name__ == "__main__":
    avl = AVLTree()
    root = None

    values = [30, 20, 40, 10, 25]

    for val in values:
        root = avl.insert(root, val)

    print("Inorder traversal after insertion:")
    avl.inorder(root)

    root = avl.delete(root, 20)

    print("\n\nInorder traversal after deletion of 20:")
    avl.inorder(root)
```

Output:

```
PS C:\Users\BOUNA\Desktop\ADS> python practical4.py
Inorder traversal after insertion:
10 20 25 30 40

Inorder traversal after deletion of 20:
10 25 30 40
```

PRACTICAL-5

Implement Red-Black Tree.

Code:

```
# ============================================
# RED-BLACK TREE IMPLEMENTATION
# ============================================

RED = True
BLACK = False


class Node:
    def __init__(self, data):
        self.data = data
        self.color = RED
        self.left = None
        self.right = None
        self.parent = None


class RedBlackTree:
    def __init__(self):
        self.NIL = Node(0)
        self.NIL.color = BLACK
        self.root = self.NIL

    # ---------- Left Rotate ----------
    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 is None:
            self.root = y
        elif x == x.parent.left:
            x.parent.left = y
        else:
            x.parent.right = y

        y.left = x
        x.parent = y

    # ---------- Right Rotate ----------
    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 is None:
            self.root = x
        elif y == y.parent.right:
            y.parent.right = x
        else:
            y.parent.left = x

        x.right = y
        y.parent = x

    # ---------- Insert ----------
    def insert(self, key):
        node = Node(key)
        node.left = self.NIL
        node.right = self.NIL

        parent = None
        curr = self.root

        while curr != self.NIL:
            parent = curr
            if node.data < curr.data:
                curr = curr.left
            else:
                curr = curr.right

        node.parent = parent
        if parent is None:
            self.root = node
        elif node.data < parent.data:
            parent.left = node
        else:
            parent.right = node

        node.color = RED
        self.fix_insert(node)

    # ---------- Fix Violations ----------
    def fix_insert(self, k):
        while k.parent and k.parent.color == RED:
            if k.parent == k.parent.parent.left:
                u = k.parent.parent.right  # Uncle
                if u.color == RED:
                    k.parent.color = BLACK
                    u.color = BLACK
                    k.parent.parent.color = RED
                    k = k.parent.parent
                else:
                    if k == k.parent.right:
                        k = k.parent
```

```
                    self.left_rotate(k)
                k.parent.color = BLACK
                k.parent.parent.color = RED
                self.right_rotate(k.parent.parent)
            else:
                u = k.parent.parent.left
                if u.color == RED:
                    k.parent.color = BLACK
                    u.color = BLACK
                    k.parent.parent.color = RED
                    k = k.parent.parent
                else:
                    if k == k.parent.left:
                        k = k.parent
                        self.right_rotate(k)
                    k.parent.color = BLACK
                    k.parent.parent.color = RED
                    self.left_rotate(k.parent.parent)

        self.root.color = BLACK

    # ---------- Inorder Traversal ----------
    def inorder(self, node):
        if node != self.NIL:
            self.inorder(node.left)
            color = "R" if node.color else "B"
            print(f"{node.data}({color})", end=" ")
            self.inorder(node.right)


# ==============================================
# DRIVER CODE
# ==============================================

if __name__ == "__main__":
    rbt = RedBlackTree()
    values = [10, 20, 30, 15, 25, 5]

    for v in values:
        rbt.insert(v)

    print("Inorder traversal of Red-Black Tree:")
    rbt.inorder(rbt.root)
```

Output:

```
PS C:\Users\BOUNA\Desktop\ADS> python practical5.py
Inorder traversal of Red-Black Tree:
5(R) 10(B) 15(R) 20(B) 25(R) 30(B)
```

PRACTICAL-6

Implement 2-3 Tree.

Code:

```
# ============================================
# 2–3 TREE IMPLEMENTATION
# ============================================

class TwoThreeNode:
    def __init__(self, keys=None, children=None):
        self.keys = keys or []
        self.children = children or []

    def is_leaf(self):
        return len(self.children) == 0


class TwoThreeTree:
    def __init__(self):
        self.root = None

    # ---------- Insert ----------
    def insert(self, key):
        if not self.root:
            self.root = TwoThreeNode([key])
        else:
            new_root = self._insert(self.root, key)
            if isinstance(new_root, tuple):
                self.root = TwoThreeNode(
                    [new_root[1]],
                    [new_root[0], new_root[2]]
                )
            else:
                self.root = new_root

    def _insert(self, node, key):
        if node.is_leaf():
            node.keys.append(key)
            node.keys.sort()
        else:
            if key < node.keys[0]:
```

```
            child = self._insert(node.children[0], key)
        elif len(node.keys) == 1 or key < node.keys[1]:
            child = self._insert(node.children[1], key)
        else:
            child = self._insert(node.children[2], key)

        if isinstance(child, tuple):
            left, median, right = child
            self._add_child(node, median, left, right)

    if len(node.keys) == 3:
        return self._split(node)

    return node

# ---------- Add Child ----------
def _add_child(self, node, key, left, right):
    node.keys.append(key)
    node.keys.sort()
    pos = node.keys.index(key)
    node.children.pop(pos)
    node.children.insert(pos, left)
    node.children.insert(pos + 1, right)

# ---------- Split Node ----------
def _split(self, node):
    mid_key = node.keys[1]

    left = TwoThreeNode(
        [node.keys[0]],
        node.children[:2]
    )
    right = TwoThreeNode(
        [node.keys[2]],
        node.children[2:]
    )

    return (left, mid_key, right)

# ---------- Inorder Traversal ----------
def inorder(self, node):
    if node:
```

```
        if node.is_leaf():
            for k in node.keys:
                print(k, end=" ")
        else:
            for i, k in enumerate(node.keys):
                self.inorder(node.children[i])
                print(k, end=" ")
            self.inorder(node.children[-1])


# ============================================
# DRIVER CODE
# ============================================

if __name__ == "__main__":
    tree = TwoThreeTree()
    values = [10, 20, 5, 15, 25, 30]

    for v in values:
        tree.insert(v)

    print("Inorder Traversal of 2–3 Tree:")
    tree.inorder(tree.root)
```

Output:

```
PS C:\Users\BOUNA\Desktop\ADS> python practical6.py
Inorder Traversal of 2–3 Tree:
5 10 15 20 25 30
PS C:\Users\BOUNA\Desktop\ADS>
```

PRACTICAL-7

Implement B Tree.

Code:

```
# ============================================
# B TREE IMPLEMENTATION (MIN DEGREE t = 2)
# ============================================


class BTreeNode:
    def __init__(self, t, leaf):
        self.t = t
        self.leaf = leaf
        self.keys = []
        self.children = []


class BTree:
    def __init__(self, t):
        self.t = t
        self.root = BTreeNode(t, True)

    # ---------- Traverse ----------
    def traverse(self, node):
        i = 0
        for i in range(len(node.keys)):
            if not node.leaf:
                self.traverse(node.children[i])
            print(node.keys[i], end=" ")
        if not node.leaf:
            self.traverse(node.children[i + 1])

    # ---------- Search ----------
    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 True

        if node.leaf:
            return False

        return self.search(node.children[i], key)

    # ---------- Insert ----------
    def insert(self, key):
        root = self.root
        if len(root.keys) == (2 * self.t - 1):
            new_root = BTreeNode(self.t, False)
            new_root.children.append(root)
            self.split_child(new_root, 0)
            self.root = new_root
            self.insert_non_full(new_root, key)
        else:
            self.insert_non_full(root, key)

    def insert_non_full(self, node, key):
        i = len(node.keys) - 1

        if node.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) == (2 * self.t - 1):
            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):
    t = self.t
    y = parent.children[i]
    z = BTreeNode(t, y.leaf)

    parent.keys.insert(i, y.keys[t - 1])
    parent.children.insert(i + 1, z)

    z.keys = y.keys[t:(2 * t - 1)]
    y.keys = y.keys[0:(t - 1)]

    if not y.leaf:
        z.children = y.children[t:(2 * t)]
        y.children = y.children[0:t]

# =============================================
# DRIVER CODE
# =============================================
```

```
if __name__ == "__main__":
    btree = BTree(2)
    values = [10, 20, 5, 6, 12, 30, 7, 17]

    for v in values:
        btree.insert(v)

    print("Traversal of B-Tree:")
    btree.traverse(btree.root)

    print("\n\nSearch 12:", btree.search(btree.root, 12))
    print("Search 50:", btree.search(btree.root, 50))
```

Output:

```
PS C:\Users\BOUNA\Desktop\ADS> python practical7.py
Traversal of B-Tree:
5 6 7 10 12 17 20 30

Search 12: True
Search 50: False
PS C:\Users\BOUNA\Desktop\ADS>
```

PRACTICAL-8

Implement a program for String Matching using Boyer-Moore Algorithm on a text file content.

Code:

```
# ============================================
# BOYER-MOORE STRING MATCHING (NO FILE INPUT)
# ============================================

NO_OF_CHARS = 256

def bad_character_heuristic(pattern):
    bad_char = [-1] * NO_OF_CHARS
    for i in range(len(pattern)):
        bad_char[ord(pattern[i])] = i
    return bad_char


def boyer_moore_search(text, pattern):
    m = len(pattern)
    n = len(text)

    bad_char = bad_character_heuristic(pattern)
    positions = []

    shift = 0
    while shift <= n - m:
        j = m - 1

        while j >= 0 and pattern[j] == text[shift + j]:
            j -= 1

        if j < 0:
            positions.append(shift)
            shift += (m - bad_char[ord(text[shift + m])] if shift + m < n else 1)
        else:
            shift += max(1, j - bad_char[ord(text[shift + j])])

    return positions
```

```
# ============================================
# MAIN DRIVER
# ============================================

if __name__ == "__main__":
    text = (
        "Boyer Moore algorithm is efficient. "
        "This algorithm is faster than naive string matching."
    )

    print("Text:")
    print(text)

    pattern = input("\nEnter pattern to search: ")

    result = boyer_moore_search(text, pattern)

    if result:
        print("Pattern found at positions:", result)
    else:
        print("Pattern not found")
```

Output:
PS C:\Users\BOUNA\Desktop\ADS> python practical8.py
Text:
Boyer Moore algorithm is efficient. This algorithm is faster than naive string matching.

PRACTICAL-9

Implement a program for String Matching using Knuth-Morris-Pratt Algorithm on a text file content.

Code:

```
# ============================================
# KMP STRING MATCHING (PRE-DEFINED TEXT)
# ============================================

def compute_lps(pattern):
    lps = [0] * len(pattern)
    length = 0
    i = 1

    while i < len(pattern):
        if pattern[i] == pattern[length]:
            length += 1
            lps[i] = length
            i += 1
        else:
            if length != 0:
                length = lps[length - 1]
            else:
                lps[i] = 0
                i += 1
    return lps

def kmp_search(text, pattern):
    lps = compute_lps(pattern)
    i = j = 0
    positions = []

    while i < len(text):
        if text[i] == pattern[j]:
            i += 1
            j += 1

        if j == len(pattern):
            positions.append(i - j)
            j = lps[j - 1]

        elif i < len(text) and text[i] != pattern[j]:
            if j != 0:
                j = lps[j - 1]
            else:
                i += 1

    return positions

# ============================================
```

```
# MAIN PROGRAM
# ============================================

if __name__ == "__main__":
    text = (
        "Knuth Morris Pratt algorithm is efficient. "
        "The KMP algorithm avoids unnecessary comparisons "
        "in string matching."
    )

    print("Text:")
    print(text)

    pattern = input("\nEnter pattern to search: ")

    result = kmp_search(text, pattern)

    if result:
        print("Pattern found at positions:", result)
    else:
        print("Pattern not found")
```

Output:

PS C:\Users\BOUNA\Desktop\ADS> python practical9.py

Text:

Knuth Morris Pratt algorithm is efficient. The KMP algorithm avoids unnecessary comparisons in string matching.

Enter pattern to search: algorithm

Pattern found at positions: [19, 51]

PRACTICAL-10

Implement Huffman-Coding Method. Show the result with suitable example.

Code:

```
# ============================================
# HUFFMAN CODING IMPLEMENTATION
# ============================================

import heapq
from collections import Counter

class HuffmanNode:
    def __init__(self, char, freq):
        self.char = char
        self.freq = freq
        self.left = None
        self.right = None

    # For priority queue comparison
    def __lt__(self, other):
        return self.freq < other.freq


def build_huffman_tree(text):
    frequency = Counter(text)

    heap = []
    for char, freq in frequency.items():
        heapq.heappush(heap, HuffmanNode(char, freq))

    while len(heap) > 1:
        left = heapq.heappop(heap)
        right = heapq.heappop(heap)

        merged = HuffmanNode(None, left.freq + right.freq)
        merged.left = left
        merged.right = right

        heapq.heappush(heap, merged)

    return heap[0]
```

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

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

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


def huffman_encoding(text):
    root = build_huffman_tree(text)
    codes = {}
    generate_codes(root, "", codes)

    encoded_text = "".join(codes[char] for char in text)
    return codes, encoded_text


# ============================================
# MAIN PROGRAM WITH EXAMPLE
# ============================================

if __name__ == "__main__":
    text = "huffman coding example"

    print("Original Text:")
    print(text)

    codes, encoded_text = huffman_encoding(text)

    print("\nHuffman Codes:")
    for char, code in codes.items():
        print(f"'{char}' : {code}")

    print("\nEncoded Text:")
    print(encoded_text)
```

Output:

```
PS C:\Users\BOUNA\Desktop\ADS> python practical10.py
Original Text:
huffman coding example

Huffman Codes:
'a' : 000
'f' : 001
'n' : 010
'u' : 0110
'd' : 0111
'e' : 100
' ' : 1010
'm' : 1011
'h' : 11000
'g' : 11001
'x' : 11010
'p' : 11011
'i' : 11100
'c' : 11101
'o' : 11110
'l' : 11111

Encoded Text:
1100001100010011011000010101011101111100111111000101100110101001101000010111
101111111100
```

PRACTICAL-11

Implement One Dimensional and Two Dimensional Range Searching in any language.

One-Dimensional Range Searching:

```
# ============================================
# ONE-DIMENSIONAL RANGE SEARCHING
# ============================================

def one_d_range_search(arr, low, high):
    result = []
    for x in arr:
        if low <= x <= high:
            result.append(x)
    return result



# ============================================
# MAIN
# ============================================

if __name__ == "__main__":
    data = [5, 3, 9, 1, 12, 7, 4, 10]

    print("Data:", data)

    low = int(input("Enter lower bound: "))
    high = int(input("Enter upper bound: "))

    result = one_d_range_search(data, low, high)

    print("Elements in range:", result)
```

Output:

```
PS C:\Users\BOUNA\Desktop\ADS> python practical11_1D.py
Data: [5, 3, 9, 1, 12, 7, 4, 10]
Enter lower bound: 3
Enter upper bound: 4
Elements in range: [3, 4]
```

Two-Dimensional Range Searching:

```
# ============================================
# TWO-DIMENSIONAL RANGE SEARCHING
# ============================================

def two_d_range_search(points, x_min, x_max, y_min, y_max):
    result = []
    for (x, y) in points:
        if x_min <= x <= x_max and y_min <= y <= y_max:
            result.append((x, y))
    return result


# ============================================
# MAIN
# ============================================

if __name__ == "__main__":
    points = [(2, 3), (5, 4), (9, 6), (4, 7), (8, 1), (7, 2)]

    print("Points:", points)

    x_min = int(input("Enter x_min: "))
    x_max = int(input("Enter x_max: "))
    y_min = int(input("Enter y_min: "))
    y_max = int(input("Enter y_max: "))

    result = two_d_range_search(points, x_min, x_max, y_min, y_max)

    print("Points in range:", result)
```

Output:

```
PS C:\Users\BOUNA\Desktop\ADS> python practical11_2D.py
Points: [(2, 3), (5, 4), (9, 6), (4, 7), (8, 1), (7, 2)]
Enter x_min: 2
Enter x_max: 7
Enter y_min: 5
Enter y_max: 8
Points in range: [(4, 7)]
```

PRACTICAL-12

Write a program which creates Priority Search Tree. Implement Insert and Search Operations in this Tree.

Code:

```
# ============================================
# PRIORITY SEARCH TREE IMPLEMENTATION
# ============================================

class PSTNode:
    def __init__(self, point):
        self.point = point      # (x, y)
        self.left = None
        self.right = None


class PrioritySearchTree:
    def __init__(self):
        self.root = None

    # ----------------------------------------
    # INSERT OPERATION
    # ----------------------------------------
    def insert(self, root, point):
        if root is None:
            return PSTNode(point)

        # Priority based on y (max heap)
        if point[1] > root.point[1]:
            point, root.point = root.point, point

        # BST property based on x
        if point[0] < root.point[0]:
            root.left = self.insert(root.left, point)
        else:
            root.right = self.insert(root.right, point)

        return root

    def insert_point(self, point):
```

```
        self.root = self.insert(self.root, point)

    # ----------------------------------------
    # RANGE SEARCH OPERATION
    # Search for points where:
    # x_min ≤ x ≤ x_max and y ≥ y_min
    # ----------------------------------------
    def range_search(self, root, x_min, x_max, y_min, result):
        if root is None:
            return

        x, y = root.point

        if x_min <= x <= x_max and y >= y_min:
            result.append((x, y))

        if root.left and x_min <= x:
            self.range_search(root.left, x_min, x_max, y_min, result)

        if root.right and x <= x_max:
            self.range_search(root.right, x_min, x_max, y_min, result)

    def search(self, x_min, x_max, y_min):
        result = []
        self.range_search(self.root, x_min, x_max, y_min, result)
        return result


# ================================================
# MAIN PROGRAM
# ================================================

if __name__ == "__main__":
    pst = PrioritySearchTree()

    points = [(5, 20), (3, 15), (8, 25), (2, 10), (6, 18), (9, 5)]

    print("Inserting points:")
    for p in points:
        print(p)
```

```
    pst.insert_point(p)

x_min = 3
x_max = 8
y_min = 15

print("\nSearching points in range:")
print(f"x in [{x_min}, {x_max}] and y ≥ {y_min}")

result = pst.search(x_min, x_max, y_min)
print("Result:", result)
```

Output:

```
PS C:\Users\BOUNA\Desktop\ADS> python practical12.py
Inserting points:
(5, 20)
(3, 15)
(8, 25)
(2, 10)
(6, 18)
(9, 5)

Searching points in range:
x in [3, 8] and y ≥ 15
Result: [(8, 25), (5, 20), (3, 15), (6, 18)]
```

Assignment

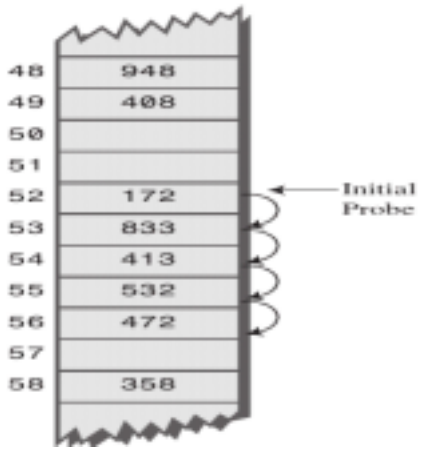
1. Compare types of hashing and explain clustering in hashing.

Ans. There are many different types of hash algorithms such as RipeMD, Tiger, xxhash and more, but the most common type of hashing used for file integrity checks are MD5, SHA1 and CRC32.
MD5 - An MD5 hash function encodes a string of information and encodes it into a 128-bit fingerprint. MD5 is often used as a checksum to verify data integrity. However, due to its age, MD5 is also known to suffer from extensive hash collision vulnerabilities, but it's still one of the most widely used algorithms in the world.
SHA1 – SHA1, developed by the National Security Agency (NSA), is a cryptographic hash function. Results from SHA1 are expressed as a 160-bit hexadecimal number. This hash function is widely considered the successor to MD5.
CRC32 – A cyclic redundancy check (CRC) is an error-detecting code often used for detection of accidental changes to data. Encoding the same data string using CRC32 will always result in the same hash output, thus CRC32 is sometimes used as a hash algorithm for file integrity checks.

Two type of clustering

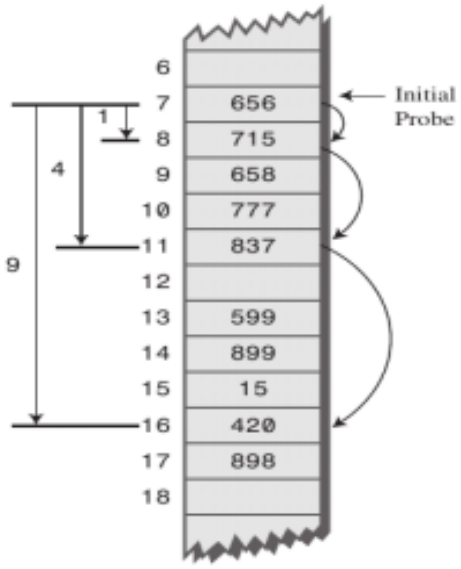
1) Primary clustering

- Primary Clustering is the tendency for a collision resolution scheme such as linear probing to create long runs of filled slots near the hash position of keys. • If the primary hash index is x, subsequent probes go to x+1,x+2,x+3 and so on, this results in Primary Clustering.
- Once the primary cluster forms, the bigger the cluster gets, the faster it grows. And it reduces the performance.



2) Secondary clustering

- Secondary Clustering is the tendency for a collision resolution scheme such as quadratic probing to create long runs of filled slots away from the hash position of keys.
- If the primary hash index is x, probes go to x+1,x+4,x+9,x+16,x+25 and so on, this results in Secondary Clustering.
- Secondary clustering is less severe in terms of performance hit than primary clustering, and is an attempt to keep clusters from forming by using Quadratic Probing. The idea is to probe more widely separated cells, instead of those adjacent to the primary hash site.



2. Compare hashing and dictionary.

Dictionary	Hashing
Dictionary is generic type data structure to hold key-value pairs.	hash table is not a generic type data structure to hold key value pairs.
Doesn't need any mathematical function.	Uses functions to generate key values called hash functions.

The Dictionary class is a strongly types < T Key, T Value > and you must specify the data types for both the key and value.	The Hashtable is a weakly typed data structure, so you can add keys and values of any Object Type to the Hashtable.(in programming)
A dictionary is kind of "an interface", it can be implemented using a sorted array, an unsorted array, an hash table, etc.	Hash table is a possible implementation of Dictionary interface.
Keys can be sorted or unsorted, depending on need.	Keys are always nonsorted, as it's generated randomly from a function.
Keys can be of any data type.	keys are mostly integers or they are converted into iintegers.

3. Explain Insertion, Search & Delete operation of dictionary with example.

Ans. Python Program to Perform Dictionary Operations (Hash Table Implementation)
In this implementation, a Dictionary is created using a Hash Table with Separate Chaining for collision resolution.

- Hash Function: It uses key % 10 to find the index.
- Collision Resolution: If multiple keys map to the same index, they are stored in a linked list at that index.

Code:

```
import sys

# Node class for the linked list (chaining)
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class Dictionary:
    def __init__(self):
        self.MAX = 10
        # Initialize the hash table with None
        self.root = [None] * self.MAX

    # 1. Insert Operation
    def insert(self, key):
        index = int(key % self.MAX)
        new_node = Node(key)

        if self.root[index] is None:
            self.root[index] = new_node
        else:
            # Collision: Add to the end of the chain
            temp = self.root[index]
```

```
            while temp.next is not None:
                temp = temp.next
            temp.next = new_node
        print(f"Inserted {key} at Index {index}")

    # 2. Search Operation
    def search(self, key):
        index = int(key % self.MAX)
        temp = self.root[index]
        found = False

        while temp is not None:
            if temp.data == key:
                print(f"\n[Success] Key {key} found at Index {index}!")
                found = True
                break
            temp = temp.next

        if not found:
            print(f"\n[Not Found] Key {key} does not exist in the dictionary.")

    # 3. Delete Operation
    def delete_ele(self, key):
        index = int(key % self.MAX)
        temp = self.root[index]
        prev = None

        while temp is not None and temp.data != key:
            prev = temp
            temp = temp.next

        if temp is not None and temp.data == key:
            if prev is None:
                # Deleting the head of the chain
                self.root[index] = temp.next
            else:
                # Deleting from middle or end
                prev.next = temp.next
            print(f"\n[Deleted] Key {key} removed from Index {index}.")
            del temp
        else:
            print(f"\n[Error] Key {key} not found, cannot delete.")

    # 4. Display Operation (New)
    def display(self):
        print("\n" + "="*30)
        print("   CURRENT DICTIONARY STATE   ")
        print("="*30)
        for i in range(self.MAX):
            print(f"Index [{i}]:", end=" ")
            temp = self.root[i]
            if temp is None:
```

```
            print("Empty")
        else:
            # Traverse the chain
            while temp is not None:
                print(f"{temp.data} ->", end=" ")
                temp = temp.next
            print("NULL")
    print("="*30)

# 5. Count Operation (New)
def get_count(self):
    count = 0
    for i in range(self.MAX):
        temp = self.root[i]
        while temp is not None:
            count += 1
            temp = temp.next
    print(f"\nTotal elements in Dictionary: {count}")

# Main Driver Code
def main():
    d = Dictionary()

    while True:
        print("\n--- DICTIONARY MENU ---")
        print("1. Insert Element")
        print("2. Search Element")
        print("3. Delete Element")
        print("4. Display All (Show Table)")
        print("5. Count Total Elements")
        print("6. Exit")

        try:
            ch = int(input("\nEnter your choice: "))
        except ValueError:
            print("Invalid input! Please enter a number.")
            continue

        if ch == 1:
            try:
                line = input("Enter element(s) to insert (space separated): ")
                elements = list(map(int, line.split()))
                for num in elements:
                    d.insert(num)
            except ValueError:
                print("Please enter valid integers.")

        elif ch == 2:
            try:
                n = int(input("Enter element to search: "))
                d.search(n)
            except ValueError:
```

```
            print("Invalid input.")

        elif ch == 3:
            try:
                n = int(input("Enter element to delete: "))
                d.delete_ele(n)
            except ValueError:
                print("Invalid input.")

        elif ch == 4:
            d.display()

        elif ch == 5:
            d.get_count()

        elif ch == 6:
            print("Exiting program...")
            break

        else:
            print("Invalid choice. Please try again.")

if __name__ == "__main__":
    main()
```

Output:

```
--- DICTIONARY MENU ---
1. Insert Element
2. Search Element
3. Delete Element
4. Display All (Show Table)
5. Count Total Elements
6. Exit

Enter your choice: 1
Enter element(s) to insert (space separated): 234 4563 0 2345 45 10 20
Inserted 234 at Index 4
Inserted 4563 at Index 3
Inserted 0 at Index 0
Inserted 2345 at Index 5
Inserted 45 at Index 5
Inserted 10 at Index 0
Inserted 20 at Index 0

--- DICTIONARY MENU ---
...
Enter your choice: 4

==============================
  CURRENT DICTIONARY STATE
==============================
Index [0]: 0 -> 10 -> 20 -> NULL
```

```
Index [1]: Empty
Index [2]: Empty
Index [3]: 4563 -> NULL
Index [4]: 234 -> NULL
Index [5]: 2345 -> 45 -> NULL
Index [6]: Empty
Index [7]: Empty
Index [8]: Empty
Index [9]: Empty
================================

--- DICTIONARY MENU ---
...
Enter your choice: 5

Total elements in Dictionary: 7

--- DICTIONARY MENU ---
...
Enter your choice: 3
Enter element to delete: 45

[Deleted] Key 45 removed from Index 5.

--- DICTIONARY MENU ---
...
Enter your choice: 4

================================
  CURRENT DICTIONARY STATE
================================
Index [0]: 0 -> 10 -> 20 -> NULL
...
Index [5]: 2345 -> NULL
...
================================
```

4. How height and breadth do important roles in tree operations? Explain technically. How rotation is useful in balancing a tree?

Ans. The depth of a node is defined as: the number of edges between the node and the root. The height of a node is the number of edges on the longest path between that node and a leaf. The height of a tree is the height of its root node. A forest is a set of $n \geq 0$ disjoint trees.

Generally, depth of a tree is referred as height of a tree.
Height and breadth are useful in traversal and searching operations in tree data structure.
A Tree is typically traversed in two ways:

- Breadth First Traversal (Or Level Order Traversal)
- Depth First Traversals
    - Inorder Traversal (Left-Root-Right)
    - Preorder Traversal (Root-Left-Right)

- Postorder Traversal (Left-Right-Root)

BFS and DFSs of above Tree

```
    1
   / \
  2   3
 / \
4   5
```

- Breadth First Traversal: 1 2 3 4 5
- Depth First Traversals:
  - Preorder Traversal: 1 2 4 5 3
  - Inorder Traversal: 4 2 5 1 3
  - Postorder Traversal: 4 5 2 3 1

Two kind of searching operations can be done in tree data structure:
1. BFS
2. DFS

BFS (FIFO): BFS starts traversal from the root node and then explore the search in the level by level manner i.e. as close as possible from the root node. BFS is useful in finding shortest path. BFS can be used to find the shortest distance between some starting node and the remaining nodes of the graph.

DFS (LIFO): DFS starts the traversal from the root node and explore the search as far as possible from the root node i.e. depth wise. DFS in not so useful in finding shortest path. It is used to perform a traversal of a general graph and the idea of DFS is to make a path as long as possible, and then go back (backtrack) to add branches also as long as possible.

Tree Rotation: Tree rotation is an operation on a binary tree that changes the structure without interfering with the order of the elements. To balance itself, an AVL tree may perform the following four kinds of rotations:
- Left rotation
- Right rotation
- Left-Right rotation
- Right-Left rotation

Left Rotation: If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation.

```
A              B
 \            / \
  B    =>    A   C
   \
    C
```

Right Rotation: AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.

```
    C          B
   /          / \
  B    =>    A   C
 /
```

A

Left-Right Rotation: A left-right rotation is a combination of left rotation followed by right rotation.

Right-Left Rotation: The second type of double rotation is Right-Left Rotation. It is a combination of right rotation followed by left rotation.

5. Explain insertion and deletion operation in BST by suitable example. Explain search, min, max operation of that example.

Ans.

BST OPERATIONS

Inserting a node A naïve algorithm for inserting a node into a BST is that, we start from the root node, if the node to insert is less than the root, we go to left child, and otherwise we go to the right child of the root. We continue this process (each node is a root for some sub tree) until we find a null pointer (or leaf node) where we cannot go any further. We then insert the node as a left or right child of the leaf node based on node is less or greater than the leaf node. We note that a new node is always inserted as a leaf node.

A recursive algorithm for inserting a node into a BST is as follows. Assume we insert a node N to tree T. if the tree is empty, the we return new node N as the tree. Otherwise, the problem of inserting is reduced to inserting the node N to left of right sub trees of T, depending on N is less or greater than T. A definition is as follows.

Insert(N, T) = N if T is empty = insert(N, T.left) if N < T = insert(N, T.right) if N > T

Searching for a node Searching for a node is similar to inserting a node. We start from root, and then go left or right until we find (or not find the node). A recursive definition of search is as follows. If the node is equal to root, then we return true. If the root is null, then we return false. otherwise we recursively solve the problem for T.left or T.right, depending on N < T or N > T. A recursive definition is as follows. Search should return a true or false, depending on the node is found or not.

Search(N, T) = false if T is empty = true if T = N = search(N, T.left) if N < T = search(N, T.right) if N > T

Deleting a node A BST is a connected structure. That is, all nodes in a tree are connected to some ther node. For example, each node has a parent, unless node is the root. Therefore deleting node could affect all sub trees of that node. For example, deleting node 5 from the tree could result in losing sub trees that are rooted at 1 and 9. Hence we need to be careful about deleting nodes from a tree. The best way to deal with deletion seems to be considering special cases. What if the node to delete is a leaf node?. What if the node is a node with just one child?. What if the node is an internal node (with two children). The latter case is the hardest to resolve. But we will find a way to handle this situation as well.

Case 1 : The node to delete is a leaf node This is a very easy case. Just delete the node. We are done.

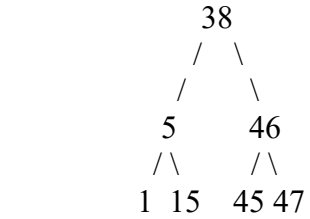
Case 2 : The node to delete is a node with one child. This is also not too bad. If the node to be deleted is a left child of the parent, then we connect the left pointer of the parent (of

the deleted node) to the single child. Otherwise if the node to be deleted is a right child of the parent, then we connect the right pointer of the parent (of the deleted node) to single child.

Case 3: The node to delete is a node with two children This is a difficult case as we need to deal with two sub trees. But we find an easy way to handle it. First we find a replacement node (from leaf node or nodes with one child) for the node to be deleted. We need to do this while maintaining the BST order property. Then we swap leaf node or node with one child with the node to be deleted (swap the data) and delete the leaf node or node with one child (case 1 or case 2).

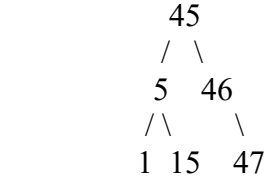
Next problem is finding a replacement leaf node for the node to be deleted. We can easily find this as follows. If the node to be deleted is N, the find the largest node in the left sub tree of N or the smallest node in the right sub tree of N. These are two candidates that can replace the node to be deleted without losing the order property.

For example, consider the following tree and suppose we need to delete the root 38.



Then we find the largest node in the left sub tree (15) or smallest node in the right sub tree (45) and replace the root with that node and then delete that node. The following set of images demonstrates this process.

Using Inorder Successor (Smallest in Right Subtree - 45):



Min and Max Operation For the minimum, go to the left child of the root; then go to the left child of that child and so on, until you come to a node that has no left child. This node is the minimum.

For the maximum value in the tree follow the same procedure, but go from right child to right child until you find a node with no right child. This node is the maximum.

The value as min will be 1 and 47 as max in the above tree example.