```
/---- 1----/
def bubble(b):

    n=len(b)

    for i in range (n):
        for j in range (0,n-1-1):
            if b[j]>b[j+1]:
                b[j],b[j+1]=b[j+1],b[j]
    print("Sorted Salaries By Bubble Sort Is:",b)

def selection(b):

    n=len(b)

    for i in range (n):

        min_i=i

        for j in range (i+1,n):
            if b[j]<b[min_i]:
                min_i=j
        b[i],b[min_i]=b[min_i],b[i]
    print("Sorted Salaeies By Selection Sort Is:",b)

a=list(map(float, input("Enter Salary:").split()))
c=int(input("Enter 1 For Bubble Sort Or 2 For Selection Sort :"))

if c==1:

    bubble(a)

else:

    selection(a)

top=a[-5:]

top1=top[::-1]
print("to[ salaries are :",top1)


/----2---/
document = ""
undo_stack = []
redo_stack = []

def make_change():
    global document, undo_stack, redo_stack
    new_text = "Hello World"
```

```
    undo_stack.append(document)
    document += new_text
    redo_stack.clear()

def undo():
    global document, undo_stack, redo_stack
    redo_stack.append(document)
    document = undo_stack.pop()

def redo():
    global document, undo_stack, redo_stack
    undo_stack.append(document)
    document = redo_stack.pop()

def display():
    print(document)


make_change()
display()

undo()
display()

redo()
display()

/------3-----/

class HashTable:
    def __init__(self, size):
        self.size = size
        # create size empty lists for chaining
        self.table = [[] for _ in range(size)]

    # Division method hash function (works for integer keys)
    def hash_function(self, key):
        return key % self.size

    # Insert key (avoid duplicate)
    def insert(self, key):
        index = self.hash_function(key)
        chain = self.table[index]
        if key in chain:
            print(f"{key} already present at index {index}")
            return
        chain.append(key)
        print(f"Inserted {key} at index {index}")
    def search(self, key):
        index = self.hash_function(key)
```

```
        if key in self.table[index]:
            print(f"{key} found at index {index}")
            return True
        else:
            print(f"{key} not found")
            return False

    # Display entire hash table
    def display(self):
        print("\nHash Table:")
        for i, chain in enumerate(self.table):
            print(f"{i}: {chain}")


# Example usage (matches the images)
if __name__ == "__main__":
    ht = HashTable(7)  # table size 7
    keys = [10, 20, 15, 7, 5, 32]

    for key in keys:
        ht.insert(key)

    ht.display()
    ht.search(15)
    ht.search(99)

/-----4--------/

class HashTable:
    def __init__(self, size):
        self.size = size
        self.table = [None] * size  # Initialize table with None

    # Division method hash function
    def hash_function(self, key):
        return key % self.size

    # Insert key using linear probing
    def insert(self, key):
        index = self.hash_function(key)
        start_index = index

        while self.table[index] is not None:
            index = (index + 1) % self.size
            if index == start_index:
                print("Hash table is full, cannot insert", key)
                return

        self.table[index] = key
        print(f"Inserted {key} at index {index}")
```

```
    # Search key
    def search(self, key):
        index = self.hash_function(key)
        start_index = index

        while self.table[index] is not None:
            if self.table[index] == key:
                print(f"{key} found at index {index}")
                return True
            index = (index + 1) % self.size
            if index == start_index:
                break

        print(f"{key} not found")
        return False

    # Delete key
    def delete(self, key):
        index = self.hash_function(key)
        start_index = index

        while self.table[index] is not None:
            if self.table[index] == key:
                self.table[index] = None
                print(f"{key} deleted from index {index}")
                return
            index = (index + 1) % self.size
            if index == start_index:
                break

        print(f"{key} not found, cannot delete")

    # Display hash table
    def display(self):
        print("\nHash Table:")
        for i, val in enumerate(self.table):
            print(f"{i}: {val}")


# Example usage
ht = HashTable(7)  # Table size 7
keys = [10, 20, 15, 7, 5, 32]

for key in keys:
    ht.insert(key)

ht.display()

ht.search(15)
```

```
ht.delete(20)
ht.display()
ht.search(99)


/--------5-----------/

# -------------------------------
# Graph Traversal: DFS and BFS
# -------------------------------

from collections import deque

# Sample locations (nodes)
locations = ['A', 'B', 'C', 'D', 'E']

# Adjacency matrix for DFS
adj_matrix = [
    [0, 1, 1, 0, 0],  # A
    [1, 0, 1, 1, 0],  # B
    [1, 1, 0, 0, 1],  # C
    [0, 1, 0, 0, 1],  # D
    [0, 0, 1, 1, 0]   # E
]

# Adjacency list for BFS
adj_list = {
    'A': ['B', 'C'],
    'B': ['A', 'C', 'D'],
    'C': ['A', 'B', 'E'],
    'D': ['B', 'E'],
    'E': ['C', 'D']
}


# -------------------------------
# DFS using adjacency matrix
# -------------------------------
def dfs(start):
    visited = [False] * len(locations)
    result = []
    stack = [locations.index(start)]

    while stack:
        node = stack.pop()
        if not visited[node]:
            visited[node] = True
            result.append(locations[node])

            # Add neighbors in reverse order for correct sequence
```

```
            for i in reversed(range(len(locations))):
                if adj_matrix[node][i] == 1 and not visited[i]:
                    stack.append(i)
    return result


# --------------------------------
# BFS using adjacency list
# --------------------------------
def bfs(start):
    visited = {loc: False for loc in locations}
    result = []
    queue = deque([start])
    visited[start] = True

    while queue:
        node = queue.popleft()
        result.append(node)

        for neighbor in adj_list[node]:
            if not visited[neighbor]:
                visited[neighbor] = True
                queue.append(neighbor)
    return result


# --------------------------------
# Perform Traversals
# --------------------------------
print("DFS Sequence:", dfs('A'))
print("BFS Sequence:", bfs('A'))



/---------6---------/

# --------------------------------
# Assignment 8 - Binary Search Tree
# --------------------------------

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


# Binary Search Tree class
class BST:
```

```
def __init__(self):
    self.root = None

# Insert key into BST
def insert(self, root, key):
    if root is None:
        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)
    return root

# Search for a key
def search(self, root, key):
    if root is None or root.key == key:
        return root
    if key < root.key:
        return self.search(root.left, key)
    return self.search(root.right, key)

# Find the node with minimum key
def minValueNode(self, node):
    current = node
    while current.left:
        current = current.left
    return current

# Delete a key from BST
def delete(self, root, key):
    if root is None:
        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:
        # Node with only one child or no child
        if root.left is None:
            return root.right
        elif root.right is None:
            return root.left

        # Node with two children:
        # Get the inorder successor (smallest in the right subtree)
        temp = self.minValueNode(root.right)
        root.key = temp.key
        root.right = self.delete(root.right, temp.key)
```

```
        return root

    # Inorder traversal (Left → Root → Right)
    def inorder(self, root):
        return self.inorder(root.left) + [root.key] + self.inorder(root.right) if
root else []


# ------------------------------
# Example Usage
# ------------------------------

bst = BST()
keys = [50, 30, 70, 20, 40, 60, 80]

# Insert keys into BST
for key in keys:
    bst.root = bst.insert(bst.root, key)

# Display inorder traversal
print("Inorder Traversal:", bst.inorder(bst.root))

# Search for a key
key = 40
print(f"Search {key}:", "Found" if bst.search(bst.root, key) else "Not Found")

# Delete a key
bst.root = bst.delete(bst.root, 30)
print("Inorder after deleting 30:", bst.inorder(bst.root))


/-----7-------/


def linear(y, n):
    if n in y:
        print("ID Found In Linear Search")
    else:
        print("ID Not Found In Linear Search")

def binary(y, n):
    y.sort()
    print("Sorted List Is :", y)
    left = 0
    right = len(y) - 1
    while left <= right:
        mid = (left + right) // 2
        if y[mid] == n:
            return mid
        if y[mid] < n:
```

```
            left = mid + 1
        else:
            right = mid - 1
    return -1

# --- Main Program ---
cs = list(map(int, input("Enter List : ").split()))
print("Entered List Is :", cs)
x = int(input("Enter ID To Search : "))
a = int(input("Enter 1 For Linear Search Or 2 For Binary Search : "))

if a == 1:
    linear(cs, x)
else:
    b = binary(cs, x)
    if b != -1:
        print("ID Found In Binary Search")
    else:
        print("ID Not Found In Binary Search")

/--------8--------/

# Define Node class for each student
class StudentNode:
    def __init__(self, name, roll_no, marks):
        self.name = name
        self.roll_no = roll_no
        self.marks = marks
        self.next = None

# Define Linked List class
class StudentLinkedList:
    def __init__(self):
        self.head = None

    # Add student at end
    def push_back(self, name, roll_no, marks):
        newNode = StudentNode(name, roll_no, marks)
        if self.head is None:
            self.head = newNode
        else:
            temp = self.head
            while temp.next:
                temp = temp.next
            temp.next = newNode
        print("New student data is added.")

    # Display all student data
    def display(self):
        if self.head is None:
```

```
            print("No student data available.")
            return
        else:
            temp = self.head
            while temp:
                print("name=", temp.name, " roll_no=", temp.roll_no, " marks=",
temp.marks)
                temp = temp.next

    # Search student by name or roll number
    def search(self):
        print("Enter what do you want to search:")
        print("1. Name \n2. Roll_no")
        choice = int(input())

        if choice == 1:
            sname = input("Enter name you want to search (case-sensitive): ")
            temp = self.head
            i = 0
            found = False
            while temp:
                if temp.name == sname:
                    print(f"Student record found at index {i}: name={temp.name},
roll_no={temp.roll_no}, marks={temp.marks}")
                    found = True
                i += 1
                temp = temp.next
            if not found:
                print("Student not found.")

        elif choice == 2:
            sroll = int(input("Enter roll number you want to search: "))
            temp = self.head
            i = 0
            found = False
            while temp:
                if temp.roll_no == sroll:
                    print(f"Student record found at index {i}: name={temp.name},
roll_no={temp.roll_no}, marks={temp.marks}")
                    found = True
                i += 1
                temp = temp.next
            if not found:
                print("Student not found.")

        else:
            print("Invalid choice!")


# --- Main Program ---
```

```
ll = StudentLinkedList()
ll.push_back("Yash", 36, 100)
ll.push_back("Aditya", 35, 100)
ll.push_back("Mohit", 32, 100)

print("\n--- Student Data ---")
ll.display()

print("\n--- Search Operation ---")
ll.search()
```