1. Write a python programs that use both recursive and non-recursive functions for implementing the following searching methods:

- a) Linear search
- b) Binary search

In [1]:
```
def linear_search_iterative(arr, target):
    """
    Iterative implementation of linear search

    Args:
        arr (list): List to search in
        target: Element to find

    Returns:
        int: Index of target if found, -1 if not found
    """
    for i in range(len(arr)):
        if arr[i] == target:
            return i
    return -1

def linear_search_recursive(arr, target, index=0):
    """
    Recursive implementation of linear search

    Args:
        arr (list): List to search in
        target: Element to find
        index (int): Current index being checked

    Returns:
        int: Index of target if found, -1 if not found
    """
    # Base cases
    if index >= len(arr):
        return -1
    if arr[index] == target:
        return index

    # Recursive case
    return linear_search_recursive(arr, target, index + 1)

def binary_search_iterative(arr, target):
    """
    Iterative implementation of binary search

    Args:
        arr (list): Sorted list to search in
        target: Element to find

    Returns:
        int: Index of target if found, -1 if not found
    """
    left, right = 0, len(arr) - 1

    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
```

```
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1

def binary_search_recursive(arr, target, left=0, right=None):
    """
    Recursive implementation of binary search

    Args:
        arr (list): Sorted list to search in
        target: Element to find
        left (int): Left boundary of current search
        right (int): Right boundary of current search

    Returns:
        int: Index of target if found, -1 if not found
    """
    if right is None:
        right = len(arr) - 1

    if left > right:
        return -1

    mid = (left + right) // 2
    if arr[mid] == target:
        return mid
    elif arr[mid] < target:
        return binary_search_recursive(arr, target, mid + 1, right)
    else:
        return binary_search_recursive(arr, target, left, mid - 1)

# Test the implementations
if __name__ == "__main__":
    # Test array
    test_array = [1, 3, 5, 7, 9, 11, 13, 15, 17]
    target = 7

    print("Test Array:", test_array)
    print("Searching for:", target)
    print("\nLinear Search Results:")
    print("Iterative:", linear_search_iterative(test_array, target))
    print("Recursive:", linear_search_recursive(test_array, target))

    print("\nBinary Search Results:")
    print("Iterative:", binary_search_iterative(test_array, target))
    print("Recursive:", binary_search_recursive(test_array, target))
```

```
Test Array: [1, 3, 5, 7, 9, 11, 13, 15, 17]
Searching for: 7

Linear Search Results:
Iterative: 3
Recursive: 3

Binary Search Results:
Iterative: 3
Recursive: 3
```

1. Write Java programs to implement the following using arrays and linked lists

- a) List ADT.

In [2]:
```
class ArrayList:
    """Implementation of List ADT using array (Python list)"""

    def __init__(self):
        self.array = []

    def append(self, item):
        """Add an item to the end of the list"""
        self.array.append(item)

    def insert(self, index, item):
        """Insert an item at a specific index"""
        self.array.insert(index, item)

    def remove(self, item):
        """Remove first occurrence of item"""
        try:
            self.array.remove(item)
            return True
        except ValueError:
            return False

    def pop(self, index=-1):
        """Remove and return item at index (default last)"""
        return self.array.pop(index)

    def get(self, index):
        """Get item at index"""
        return self.array[index]

    def size(self):
        """Return number of items in list"""
        return len(self.array)

    def is_empty(self):
        """Check if list is empty"""
        return len(self.array) == 0

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

class Node:
    """Node class for LinkedList"""
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    """Implementation of List ADT using linked list"""

    def __init__(self):
        self.head = None
        self._size = 0

    def append(self, item):
        """Add an item to the end of the list"""
        new_node = Node(item)
        if not self.head:
```

```
            self.head = new_node
        else:
            current = self.head
            while current.next:
                current = current.next
            current.next = new_node
        self._size += 1

    def insert(self, index, item):
        """Insert an item at a specific index"""
        if index < 0 or index > self._size:
            raise IndexError("Index out of range")

        new_node = Node(item)
        if index == 0:
            new_node.next = self.head
            self.head = new_node
        else:
            current = self.head
            for _ in range(index - 1):
                current = current.next
            new_node.next = current.next
            current.next = new_node
        self._size += 1

    def remove(self, item):
        """Remove first occurrence of item"""
        if not self.head:
            return False

        if self.head.data == item:
            self.head = self.head.next
            self._size -= 1
            return True

        current = self.head
        while current.next:
            if current.next.data == item:
                current.next = current.next.next
                self._size -= 1
                return True
            current = current.next
        return False

    def size(self):
        """Return number of items in list"""
        return self._size

    def is_empty(self):
        """Check if list is empty"""
        return self._size == 0

    def __str__(self):
        result = []
        current = self.head
        while current:
            result.append(str(current.data))
            current = current.next
        return "[" + ", ".join(result) + "]"

# Test the implementations
if __name__ == "__main__":
    # Test ArrayList
    print("Testing ArrayList:")
```

```
    arr_list = ArrayList()
    arr_list.append(1)
    arr_list.append(2)
    arr_list.append(3)
    print("After appending 1, 2, 3:", arr_list)

    arr_list.insert(1, 4)
    print("After inserting 4 at index 1:", arr_list)

    arr_list.remove(2)
    print("After removing 2:", arr_list)

    # Test LinkedList
    print("\nTesting LinkedList:")
    linked_list = LinkedList()
    linked_list.append(1)
    linked_list.append(2)
    linked_list.append(3)
    print("After appending 1, 2, 3:", linked_list)

    linked_list.insert(1, 4)
    print("After inserting 4 at index 1:", linked_list)

    linked_list.remove(2)
    print("After removing 2:", linked_list)
```

```
Testing ArrayList:
After appending 1, 2, 3: [1, 2, 3]
After inserting 4 at index 1: [1, 4, 2, 3]
After removing 2: [1, 4, 3]

Testing LinkedList:
After appending 1, 2, 3: [1, 2, 3]
After inserting 4 at index 1: [1, 4, 2, 3]
After removing 2: [1, 4, 3]
```

1. Write a python programs to implement the following using an array.

- a) Stack ADT
- b)Queue ADT

In [3]:

```
class ArrayStack:
    """Implementation of Stack ADT using array (Python list)"""

    def __init__(self):
        self.items = []

    def push(self, item):
        """Push an item onto the stack"""
        self.items.append(item)

    def pop(self):
        """Remove and return the top item"""
        if not self.is_empty():
            return self.items.pop()
        raise IndexError("Stack is empty")

    def peek(self):
        """Return the top item without removing it"""
        if not self.is_empty():
            return self.items[-1]
        raise IndexError("Stack is empty")
```

```
    def is_empty(self):
        """Check if stack is empty"""
        return len(self.items) == 0

    def size(self):
        """Return number of items in stack"""
        return len(self.items)

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

# Test implementation
if __name__ == "__main__":
    stack = ArrayStack()
    print("Pushing items: 1, 2, 3")
    stack.push(1)
    stack.push(2)
    stack.push(3)
    print("Stack:", stack)
    print("Pop:", stack.pop())
    print("Peek:", stack.peek())
    print("Stack after operations:", stack)
```

```
Pushing items: 1, 2, 3
Stack: [1, 2, 3]
Pop: 3
Peek: 2
Stack after operations: [1, 2]
```

In [4]:

```
class ArrayQueue:
    """Implementation of Queue ADT using array (Python list)"""

    def __init__(self):
        self.items = []

    def enqueue(self, item):
        """Add an item to the rear of the queue"""
        self.items.append(item)

    def dequeue(self):
        """Remove and return the front item"""
        if not self.is_empty():
            return self.items.pop(0)
        raise IndexError("Queue is empty")

    def front(self):
        """Return the front item without removing it"""
        if not self.is_empty():
            return self.items[0]
        raise IndexError("Queue is empty")

    def is_empty(self):
        """Check if queue is empty"""
        return len(self.items) == 0

    def size(self):
        """Return number of items in queue"""
        return len(self.items)

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

# Test implementation
```

```
if __name__ == "__main__":
    queue = ArrayQueue()
    print("Enqueuing items: 1, 2, 3")
    queue.enqueue(1)
    queue.enqueue(2)
    queue.enqueue(3)
    print("Queue:", queue)
    print("Dequeue:", queue.dequeue())
    print("Front:", queue.front())
    print("Queue after operations:", queue)
```

```
Enqueuing items: 1, 2, 3
Queue: [1, 2, 3]
Dequeue: 1
Front: 2
Queue after operations: [2, 3]
```

1. Write a python program that reads an infix expression and converts the expression to postfix form. (Use stack ADT).

In [5]:

```
class InfixToPostfix:
    def __init__(self):
        self.precedence = {'+': 1, '-': 1, '*': 2, '/': 2, '^': 3}
        self.stack = []

    def convert(self, expression):
        """Convert infix expression to postfix"""
        output = []

        for char in expression:
            if char.isalnum():  # Operand
                output.append(char)
            elif char == '(':  # Left parenthesis
                self.stack.append(char)
            elif char == ')':  # Right parenthesis
                while self.stack and self.stack[-1] != '(':
                    output.append(self.stack.pop())
                if self.stack and self.stack[-1] == '(':
                    self.stack.pop()
            else:  # Operator
                while (self.stack and self.stack[-1] != '(' and
                       self.precedence.get(char, 0) <= self.precedence.get(self.sta
                    output.append(self.stack.pop())
                self.stack.append(char)

        # Pop remaining operators
        while self.stack:
            output.append(self.stack.pop())

        return ''.join(output)

# Test implementation
if __name__ == "__main__":
    converter = InfixToPostfix()
    expressions = [
        "a+b*c",
        "(a+b)*c",
        "a+b*c+d",
        "a*(b+c)"
    ]

    for expr in expressions:
```

```
        print(f"Infix: {expr}")
        print(f"Postfix: {converter.convert(expr)}\n")
```

```
Infix: a+b*c
Postfix: abc*+

Infix: (a+b)*c
Postfix: ab+c*

Infix: a+b*c+d
Postfix: abc*+d+

Infix: a*(b+c)
Postfix: abc+*
```

1. Write a python program to implement circular queue ADT using an array.

In [6]:
```
class CircularQueue:
    """Implementation of Circular Queue ADT using array"""

    def __init__(self, capacity):
        self.capacity = capacity
        self.queue = [None] * capacity
        self.front = self.rear = -1

    def is_full(self):
        """Check if queue is full"""
        return ((self.rear + 1) % self.capacity == self.front or
                (self.front == 0 and self.rear == self.capacity - 1))

    def is_empty(self):
        """Check if queue is empty"""
        return self.front == -1

    def enqueue(self, item):
        """Add an item to the queue"""
        if self.is_full():
            raise IndexError("Queue is full")

        if self.front == -1:  # First element
            self.front = 0
            self.rear = 0
        else:
            self.rear = (self.rear + 1) % self.capacity

        self.queue[self.rear] = item

    def dequeue(self):
        """Remove and return the front item"""
        if self.is_empty():
            raise IndexError("Queue is empty")

        item = self.queue[self.front]

        if self.front == self.rear:  # Last element
            self.front = -1
            self.rear = -1
        else:
            self.front = (self.front + 1) % self.capacity

        return item
```

```
    def __str__(self):
        if self.is_empty():
            return "[]"

        items = []
        index = self.front
        while True:
            items.append(str(self.queue[index]))
            if index == self.rear:
                break
            index = (index + 1) % self.capacity
        return "[" + ", ".join(items) + "]"

# Test implementation
if __name__ == "__main__":
    cq = CircularQueue(5)
    print("Enqueuing: 1, 2, 3, 4")
    cq.enqueue(1)
    cq.enqueue(2)
    cq.enqueue(3)
    cq.enqueue(4)
    print("Queue:", cq)

    print("\nDequeuing two items")
    print("Dequeued:", cq.dequeue())
    print("Dequeued:", cq.dequeue())
    print("Queue:", cq)

    print("\nEnqueuing: 5, 6")
    cq.enqueue(5)
    cq.enqueue(6)
    print("Queue:", cq)
```

```
Enqueuing: 1, 2, 3, 4
Queue: [1, 2, 3, 4]

Dequeuing two items
Dequeued: 1
Dequeued: 2
Queue: [3, 4]

Enqueuing: 5, 6
Queue: [3, 4, 5, 6]
```

1. Write a python program that uses both a stack and a queue to test whether the given string is a palindrome or not.

In [7]:

```
from collections import deque

def is_palindrome(s):
    """
    Check if the given string is a palindrome using a stack and a queue.

    Args:
        s (str): The input string.

    Returns:
        bool: True if the string is a palindrome, False otherwise.
    """
    # Normalize the string: remove non-alphanumeric characters and convert to lower
    s = ''.join(char.lower() for char in s if char.isalnum())

    # Initialize a stack and a queue
```

```
    stack = []
    queue = deque()

    # Populate the stack and the queue with characters from the string
    for char in s:
        stack.append(char)
        queue.append(char)

    # Compare characters popped from the stack and dequeued from the queue
    while stack:
        if stack.pop() != queue.popleft():
            return False

    return True

# Test the function
if __name__ == "__main__":
    test_string = input("Enter a string to check if it is a palindrome: ")
    if is_palindrome(test_string):
        print(f"\"{test_string}\" is a palindrome.")
    else:
        print(f"\"{test_string}\" is not a palindrome.")
```

```
Enter a string to check if it is a palindrome: 121
"121" is a palindrome.
```

1. Write a python programs to implement the following using a singly linked list.

- a) Stack ADT
- b) Queue ADT

In [8]:

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

class LinkedStack:
    """Implementation of Stack ADT using linked list"""

    def __init__(self):
        self.top = None
        self._size = 0

    def push(self, item):
        """Push an item onto the stack"""
        new_node = Node(item)
        new_node.next = self.top
        self.top = new_node
        self._size += 1

    def pop(self):
        """Remove and return the top item"""
        if self.is_empty():
            raise IndexError("Stack is empty")

        item = self.top.data
        self.top = self.top.next
        self._size -= 1
        return item

    def peek(self):
        """Return the top item without removing it"""
```

```
        if self.is_empty():
            raise IndexError("Stack is empty")
        return self.top.data

    def is_empty(self):
        """Check if stack is empty"""
        return self.top is None

    def size(self):
        """Return number of items in stack"""
        return self._size

    def __str__(self):
        items = []
        current = self.top
        while current:
            items.append(str(current.data))
            current = current.next
        return "[" + ", ".join(items) + "]"

# Test implementation
if __name__ == "__main__":
    stack = LinkedStack()
    print("Pushing items: 1, 2, 3")
    stack.push(1)
    stack.push(2)
    stack.push(3)
    print("Stack:", stack)
    print("Pop:", stack.pop())
    print("Peek:", stack.peek())
    print("Stack after operations:", stack)
```

```
Pushing items: 1, 2, 3
Stack: [3, 2, 1]
Pop: 3
Peek: 2
Stack after operations: [2, 1]
```

In [9]:

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

class LinkedQueue:
    """Implementation of Queue ADT using linked list"""

    def __init__(self):
        self.front = None
        self.rear = None
        self._size = 0

    def enqueue(self, item):
        """Add an item to the rear of the queue"""
        new_node = Node(item)
        if self.is_empty():
            self.front = new_node
        else:
            self.rear.next = new_node
        self.rear = new_node
        self._size += 1

    def dequeue(self):
        """Remove and return the front item"""
        if self.is_empty():
```

```
            raise IndexError("Queue is empty")

        item = self.front.data
        self.front = self.front.next
        if self.front is None:
            self.rear = None
        self._size -= 1
        return item

    def front_item(self):
        """Return the front item without removing it"""
        if self.is_empty():
            raise IndexError("Queue is empty")
        return self.front.data

    def is_empty(self):
        """Check if queue is empty"""
        return self.front is None

    def size(self):
        """Return number of items in queue"""
        return self._size

    def __str__(self):
        items = []
        current = self.front
        while current:
            items.append(str(current.data))
            current = current.next
        return "[" + ", ".join(items) + "]"

# Test implementation
if __name__ == "__main__":
    queue = LinkedQueue()
    print("Enqueuing items: 1, 2, 3")
    queue.enqueue(1)
    queue.enqueue(2)
    queue.enqueue(3)
    print("Queue:", queue)
    print("Dequeue:", queue.dequeue())
    print("Front:", queue.front_item())
    print("Queue after operations:", queue)
```

```
Enqueuing items: 1, 2, 3
Queue: [1, 2, 3]
Dequeue: 1
Front: 2
Queue after operations: [2, 3]
```

1. Write a python programs to implement the deque (double ended queue) ADT using

- a) Array
- b) Singly linked list
- c) Doubly linked list.

In [10]:

```
class ArrayDeque:
    """Implementation of Deque ADT using array"""

    def __init__(self):
        self.items = []

    def add_front(self, item):
```

```
        """Add an item to the front"""
        self.items.insert(0, item)

    def add_rear(self, item):
        """Add an item to the rear"""
        self.items.append(item)

    def remove_front(self):
        """Remove and return the front item"""
        if not self.is_empty():
            return self.items.pop(0)
        raise IndexError("Deque is empty")

    def remove_rear(self):
        """Remove and return the rear item"""
        if not self.is_empty():
            return self.items.pop()
        raise IndexError("Deque is empty")

    def front(self):
        """Return the front item"""
        if not self.is_empty():
            return self.items[0]
        raise IndexError("Deque is empty")

    def rear(self):
        """Return the rear item"""
        if not self.is_empty():
            return self.items[-1]
        raise IndexError("Deque is empty")

    def is_empty(self):
        """Check if deque is empty"""
        return len(self.items) == 0

    def size(self):
        """Return number of items in deque"""
        return len(self.items)

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

# Test implementation
if __name__ == "__main__":
    deque = ArrayDeque()
    print("Adding items to front: 1, 2")
    deque.add_front(1)
    deque.add_front(2)
    print("Adding items to rear: 3, 4")
    deque.add_rear(3)
    deque.add_rear(4)
    print("Deque:", deque)

    print("\nRemoving from front:", deque.remove_front())
    print("Removing from rear:", deque.remove_rear())
    print("Deque after operations:", deque)
```

```
Adding items to front: 1, 2
Adding items to rear: 3, 4
Deque: [2, 1, 3, 4]

Removing from front: 2
Removing from rear: 4
Deque after operations: [1, 3]
```

1. Write a python program to implement priority queue ADT.

In [11]:
```
class PriorityQueue:
    """Implementation of Priority Queue ADT using a binary heap"""

    def __init__(self):
        self.heap = []

    def parent(self, i):
        """Get parent index"""
        return (i - 1) // 2

    def left_child(self, i):
        """Get left child index"""
        return 2 * i + 1

    def right_child(self, i):
        """Get right child index"""
        return 2 * i + 2

    def swap(self, i, j):
        """Swap elements at indices i and j"""
        self.heap[i], self.heap[j] = self.heap[j], self.heap[i]

    def insert(self, key):
        """Insert a new key into the priority queue"""
        self.heap.append(key)
        self._sift_up(len(self.heap) - 1)

    def _sift_up(self, i):
        """Move a node up to its proper position"""
        parent = self.parent(i)
        if i > 0 and self.heap[i] < self.heap[parent]:
            self.swap(i, parent)
            self._sift_up(parent)

    def extract_min(self):
        """Remove and return the minimum element"""
        if len(self.heap) == 0:
            raise IndexError("Priority queue is empty")

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

        min_val = self.heap[0]
        self.heap[0] = self.heap.pop()
        self._sift_down(0)

        return min_val

    def _sift_down(self, i):
        """Move a node down to its proper position"""
        min_index = i
        left = self.left_child(i)
        right = self.right_child(i)

        if left < len(self.heap) and self.heap[left] < self.heap[min_index]:
            min_index = left

        if right < len(self.heap) and self.heap[right] < self.heap[min_index]:
            min_index = right
```

```
        if i != min_index:
            self.swap(i, min_index)
            self._sift_down(min_index)

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

# Test implementation
if __name__ == "__main__":
    pq = PriorityQueue()
    print("Inserting: 5, 2, 8, 1, 9")
    pq.insert(5)
    pq.insert(2)
    pq.insert(8)
    pq.insert(1)
    pq.insert(9)
    print("Priority Queue:", pq)

    print("\nExtracting minimum elements:")
    for _ in range(3):
        print("Extracted:", pq.extract_min())
        print("Priority Queue:", pq)
```

```
Inserting: 5, 2, 8, 1, 9
Priority Queue: [1, 2, 8, 5, 9]

Extracting minimum elements:
Extracted: 1
Priority Queue: [2, 5, 8, 9]
Extracted: 2
Priority Queue: [5, 9, 8]
Extracted: 5
Priority Queue: [8, 9]
```

1. Write a python program to perform the following operations:

- a) Construct a binary search tree of elements.
- b) Search for a key element in the above binary search tree. CSE 2014-2015 SR Engineering College 21
- c) Delete an element from the above binary search tree.

In [12]:

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

class BinarySearchTree:
    """Implementation of Binary Search Tree"""

    def __init__(self):
        self.root = None

    def insert(self, key):
        """Insert a key into the BST"""
        self.root = self._insert_recursive(self.root, key)

    def _insert_recursive(self, root, key):
        """Helper method for insert"""
        if root is None:
            return Node(key)
```

```
        if key < root.key:
            root.left = self._insert_recursive(root.left, key)
        else:
            root.right = self._insert_recursive(root.right, key)

        return root

    def search(self, key):
        """Search for a key in the BST"""
        return self._search_recursive(self.root, key)

    def _search_recursive(self, root, key):
        """Helper method for search"""
        if root is None or root.key == key:
            return root

        if key < root.key:
            return self._search_recursive(root.left, key)
        return self._search_recursive(root.right, key)

    def delete(self, key):
        """Delete a key from the BST"""
        self.root = self._delete_recursive(self.root, key)

    def _delete_recursive(self, root, key):
        """Helper method for delete"""
        if root is None:
            return root

        if key < root.key:
            root.left = self._delete_recursive(root.left, key)
        elif key > root.key:
            root.right = self._delete_recursive(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 inorder successor (smallest in right subtree)
            temp = self._min_value_node(root.right)
            root.key = temp.key
            root.right = self._delete_recursive(root.right, temp.key)

        return root

    def _min_value_node(self, node):
        """Find the node with minimum key value in BST"""
        current = node
        while current.left:
            current = current.left
        return current

    def inorder(self):
        """Inorder traversal of BST"""
        result = []
        self._inorder_recursive(self.root, result)
        return result

    def _inorder_recursive(self, root, result):
        """Helper method for inorder traversal"""
        if root:
```

```
            self._inorder_recursive(root.left, result)
            result.append(root.key)
            self._inorder_recursive(root.right, result)

# Test implementation
if __name__ == "__main__":
    bst = BinarySearchTree()
    keys = [50, 30, 70, 20, 40, 60, 80]

    print("Inserting keys:", keys)
    for key in keys:
        bst.insert(key)

    print("Inorder traversal:", bst.inorder())

    search_key = 40
    print(f"\nSearching for {search_key}:",
          "Found" if bst.search(search_key) else "Not found")

    delete_key = 30
    print(f"\nDeleting {delete_key}")
    bst.delete(delete_key)
    print("Inorder traversal after deletion:", bst.inorder())
```

```
Inserting keys: [50, 30, 70, 20, 40, 60, 80]
Inorder traversal: [20, 30, 40, 50, 60, 70, 80]

Searching for 40: Found

Deleting 30
Inorder traversal after deletion: [20, 40, 50, 60, 70, 80]
```

1. Write a python program to implement all the functions of a dictionary (ADT) using Hashing.

In [13]:
```
class HashTable:
    """Implementation of Hash Table using linear probing"""

    def __init__(self, size=10):
        self.size = size
        self.table = [None] * size
        self.count = 0

    def _hash(self, key):
        """Hash function"""
        if isinstance(key, str):
            # For strings, use sum of ASCII values
            return sum(ord(c) for c in key) % self.size
        return key % self.size

    def _probe(self, index):
        """Linear probing"""
        return (index + 1) % self.size

    def insert(self, key, value):
        """Insert a key-value pair"""
        if self.count >= self.size:
            raise IndexError("Hash table is full")

        index = self._hash(key)
        while self.table[index] is not None:
            # If key already exists, update value
            if self.table[index][0] == key:
```

```
                self.table[index] = (key, value)
                return
            index = self._probe(index)

        self.table[index] = (key, value)
        self.count += 1

    def get(self, key):
        """Get value for a key"""
        index = self._hash(key)
        original_index = index

        while self.table[index] is not None:
            if self.table[index][0] == key:
                return self.table[index][1]
            index = self._probe(index)
            if index == original_index:
                break

        raise KeyError(f"Key '{key}' not found")

    def remove(self, key):
        """Remove a key-value pair"""
        index = self._hash(key)
        original_index = index

        while self.table[index] is not None:
            if self.table[index][0] == key:
                self.table[index] = None
                self.count -= 1
                return
            index = self._probe(index)
            if index == original_index:
                break

        raise KeyError(f"Key '{key}' not found")

    def __str__(self):
        return str([item for item in self.table if item is not None])

# Test implementation
if __name__ == "__main__":
    ht = HashTable()

    print("Inserting key-value pairs:")
    pairs = [
        ("apple", 5),
        ("banana", 8),
        ("orange", 3),
        ("grape", 2)
    ]

    for key, value in pairs:
        ht.insert(key, value)
        print(f"Inserted: {key} -> {value}")

    print("\nHash Table:", ht)

    print("\nGetting values:")
    for key, _ in pairs:
        print(f"{key} -> {ht.get(key)}")

    remove_key = "banana"
    print(f"\nRemoving '{remove_key}'")
```

```
    ht.remove(remove_key)
    print("Hash Table after removal:", ht)
```

```
Inserting key-value pairs:
Inserted: apple -> 5
Inserted: banana -> 8
Inserted: orange -> 3
Inserted: grape -> 2

Hash Table: [('apple', 5), ('orange', 3), ('grape', 2), ('banana', 8)]

Getting values:
apple -> 5
banana -> 8
orange -> 3
grape -> 2

Removing 'banana'
Hash Table after removal: [('apple', 5), ('orange', 3), ('grape', 2)]
```

1. Write a python program to implement Dijkstra‟s algorithm for Single source shortest path problem.

In [14]:
```
import heapq

def dijkstra(graph, start):
    # Dictionary to store the shortest distance from the start node to each node
    shortest_distances = {node: float('inf') for node in graph}
    shortest_distances[start] = 0

    # Priority queue to store (distance, node) tuples
    priority_queue = [(0, start)]

    # Dictionary to store the shortest path to each node
    previous_nodes = {node: None for node in graph}

    while priority_queue:
        current_distance, current_node = heapq.heappop(priority_queue)

        # Skip processing if a shorter path to the node has already been found
        if current_distance > shortest_distances[current_node]:
            continue

        # Explore neighbors of the current node
        for neighbor, weight in graph[current_node].items():
            distance = current_distance + weight

            # If a shorter path to the neighbor is found
            if distance < shortest_distances[neighbor]:
                shortest_distances[neighbor] = distance
                previous_nodes[neighbor] = current_node
                heapq.heappush(priority_queue, (distance, neighbor))

    return shortest_distances, previous_nodes

def reconstruct_path(previous_nodes, start, target):
    path = []
    current = target
    while current is not None:
        path.append(current)
        current = previous_nodes[current]
    path.reverse()
    return path if path[0] == start else []
```

```
# Example usage
graph = {
    'A': {'B': 1, 'C': 4},
    'B': {'A': 1, 'C': 2, 'D': 6},
    'C': {'A': 4, 'B': 2, 'D': 3},
    'D': {'B': 6, 'C': 3}
}

start_node = 'A'
target_node = 'D'

shortest_distances, previous_nodes = dijkstra(graph, start_node)
shortest_path = reconstruct_path(previous_nodes, start_node, target_node)

print("Shortest distances from node", start_node, ":", shortest_distances)
print("Shortest path from", start_node, "to", target_node, ":", shortest_path)
```

```
Shortest distances from node A : {'A': 0, 'B': 1, 'C': 3, 'D': 6}
Shortest path from A to D : ['A', 'B', 'C', 'D']
```

1. Write a python programs that use recursive and non-recursive functions to traverse the given binary tree in

- a) Preorder
- b) Inorder
- c) Postorder.

In [15]:
```
class TreeNode:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

class BinaryTree:
    """Binary Tree with different traversal methods"""

    def __init__(self):
        self.root = None

    def preorder_recursive(self, root, result=None):
        """Recursive preorder traversal"""
        if result is None:
            result = []

        if root:
            result.append(root.key)
            self.preorder_recursive(root.left, result)
            self.preorder_recursive(root.right, result)

        return result

    def preorder_iterative(self, root):
        """Iterative preorder traversal"""
        if not root:
            return []

        result = []
        stack = [root]

        while stack:
            node = stack.pop()
```

```
            result.append(node.key)

            if node.right:
                stack.append(node.right)
            if node.left:
                stack.append(node.left)

        return result

    def inorder_recursive(self, root, result=None):
        """Recursive inorder traversal"""
        if result is None:
            result = []

        if root:
            self.inorder_recursive(root.left, result)
            result.append(root.key)
            self.inorder_recursive(root.right, result)

        return result

    def inorder_iterative(self, root):
        """Iterative inorder traversal"""
        result = []
        stack = []
        current = root

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

            current = stack.pop()
            result.append(current.key)
            current = current.right

        return result

    def postorder_recursive(self, root, result=None):
        """Recursive postorder traversal"""
        if result is None:
            result = []

        if root:
            self.postorder_recursive(root.left, result)
            self.postorder_recursive(root.right, result)
            result.append(root.key)

        return result

    def postorder_iterative(self, root):
        """Iterative postorder traversal"""
        if not root:
            return []

        result = []
        stack1 = [root]
        stack2 = []

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

            if node.left:
```

```
                stack1.append(node.left)
            if node.right:
                stack1.append(node.right)

        while stack2:
            node = stack2.pop()
            result.append(node.key)

        return result

# Test implementation
if __name__ == "__main__":
    # Create a binary tree
    tree = BinaryTree()
    tree.root = TreeNode(1)
    tree.root.left = TreeNode(2)
    tree.root.right = TreeNode(3)
    tree.root.left.left = TreeNode(4)
    tree.root.left.right = TreeNode(5)

    print("Tree Traversals:")
    print("\nPreorder:")
    print("Recursive:", tree.preorder_recursive(tree.root))
    print("Iterative:", tree.preorder_iterative(tree.root))

    print("\nInorder:")
    print("Recursive:", tree.inorder_recursive(tree.root))
    print("Iterative:", tree.inorder_iterative(tree.root))

    print("\nPostorder:")
    print("Recursive:", tree.postorder_recursive(tree.root))
    print("Iterative:", tree.postorder_iterative(tree.root))
```

```
Tree Traversals:

Preorder:
Recursive: [1, 2, 4, 5, 3]
Iterative: [1, 2, 4, 5, 3]

Inorder:
Recursive: [4, 2, 5, 1, 3]
Iterative: [4, 2, 5, 1, 3]

Postorder:
Recursive: [4, 5, 2, 3, 1]
Iterative: [4, 5, 2, 3, 1]
```

1. Write a python programs for the implementation of bfs and dfs for a given graph.

In [16]:

```
from collections import defaultdict, deque

class Graph:
    """Graph implementation with BFS and DFS traversals"""

    def __init__(self):
        self.graph = defaultdict(list)

    def add_edge(self, src, dest):
        """Add an edge to the graph"""
        self.graph[src].append(dest)

    def bfs(self, start):
        """
```

```
        Breadth First Search traversal

        Args:
            start: Starting vertex

        Returns:
            list: BFS traversal order
        """
        visited = set()
        queue = deque([start])
        result = []

        while queue:
            vertex = queue.popleft()
            if vertex not in visited:
                visited.add(vertex)
                result.append(vertex)

                # Add unvisited neighbors to queue
                queue.extend(v for v in self.graph[vertex] if v not in visited)

        return result

    def dfs_recursive(self, vertex, visited=None, result=None):
        """
        Recursive Depth First Search traversal

        Args:
            vertex: Current vertex
            visited: Set of visited vertices
            result: List to store traversal order

        Returns:
            list: DFS traversal order
        """
        if visited is None:
            visited = set()
        if result is None:
            result = []

        visited.add(vertex)
        result.append(vertex)

        for neighbor in self.graph[vertex]:
            if neighbor not in visited:
                self.dfs_recursive(neighbor, visited, result)

        return result

    def dfs_iterative(self, start):
        """
        Iterative Depth First Search traversal

        Args:
            start: Starting vertex

        Returns:
            list: DFS traversal order
        """
        visited = set()
        stack = [start]
        result = []

        while stack:
```

```
            vertex = stack.pop()
            if vertex not in visited:
                visited.add(vertex)
                result.append(vertex)

                # Add unvisited neighbors to stack
                stack.extend(v for v in reversed(self.graph[vertex])
                             if v not in visited)

        return result

# Test implementation
if __name__ == "__main__":
    # Create a graph
    g = Graph()
    g.add_edge(0, 1)
    g.add_edge(0, 2)
    g.add_edge(1, 2)
    g.add_edge(2, 0)
    g.add_edge(2, 3)
    g.add_edge(3, 3)

    start_vertex = 2
    print(f"Graph traversals starting from vertex {start_vertex}:")

    print("\nBFS traversal:")
    print(g.bfs(start_vertex))

    print("\nDFS traversal (recursive):")
    print(g.dfs_recursive(start_vertex))

    print("\nDFS traversal (iterative):")
    print(g.dfs_iterative(start_vertex))
```

```
Graph traversals starting from vertex 2:

BFS traversal:
[2, 0, 3, 1]

DFS traversal (recursive):
[2, 0, 1, 3]

DFS traversal (iterative):
[2, 0, 1, 3]
```

1. Write a python programs for implementing the following sorting methods:
   - a) Bubble sort
   - b) Insertion sort
   - c) Quick sort
   - d) Merge sort
   - g) Binary tree sort
   - e) Heap sort
   - f) Radix sort

In [17]:

```
class SortingAlgorithms:
    """Implementation of various sorting algorithms"""

    @staticmethod
    def bubble_sort(arr):
        """
```

```
        Bubble Sort implementation
        Time Complexity: O(n^2)
        """
        n = len(arr)
        for i in range(n):
            swapped = False
            for j in range(0, n - i - 1):
                if arr[j] > arr[j + 1]:
                    arr[j], arr[j + 1] = arr[j + 1], arr[j]
                    swapped = True
            if not swapped:
                break
        return arr

    @staticmethod
    def insertion_sort(arr):
        """
        Insertion Sort implementation
        Time Complexity: O(n^2)
        """
        for i in range(1, len(arr)):
            key = arr[i]
            j = i - 1
            while j >= 0 and arr[j] > key:
                arr[j + 1] = arr[j]
                j -= 1
            arr[j + 1] = key
        return arr

    @staticmethod
    def quick_sort(arr):
        """
        Quick Sort implementation
        Time Complexity: O(n log n) average case
        """
        if len(arr) <= 1:
            return arr
        else:
            pivot = arr[len(arr) // 2]
            left = [x for x in arr if x < pivot]
            middle = [x for x in arr if x == pivot]
            right = [x for x in arr if x > pivot]
            return SortingAlgorithms.quick_sort(left) + middle + SortingAlgorithms.

    @staticmethod
    def merge_sort(arr):
        """
        Merge Sort implementation
        Time Complexity: O(n log n)
        """
        if len(arr) <= 1:
            return arr

        mid = len(arr) // 2
        left = SortingAlgorithms.merge_sort(arr[:mid])
        right = SortingAlgorithms.merge_sort(arr[mid:])

        return SortingAlgorithms._merge(left, right)

    @staticmethod
    def _merge(left, right):
        """Helper method for merge sort"""
        result = []
        i = j = 0
```

```
        while i < len(left) and j < len(right):
            if left[i] <= right[j]:
                result.append(left[i])
                i += 1
            else:
                result.append(right[j])
                j += 1

        result.extend(left[i:])
        result.extend(right[j:])
        return result

    @staticmethod
    def heap_sort(arr):
        """
        Heap Sort implementation
        Time Complexity: O(n log n)
        """
        def heapify(arr, n, i):
            largest = i
            left = 2 * i + 1
            right = 2 * i + 2

            if left < n and arr[left] > arr[largest]:
                largest = left

            if right < n and arr[right] > arr[largest]:
                largest = right

            if largest != i:
                arr[i], arr[largest] = arr[largest], arr[i]
                heapify(arr, n, largest)

        n = len(arr)

        # Build max heap
        for i in range(n // 2 - 1, -1, -1):
            heapify(arr, n, i)

        # Extract elements from heap
        for i in range(n - 1, 0, -1):
            arr[0], arr[i] = arr[i], arr[0]
            heapify(arr, i, 0)

        return arr

    @staticmethod
    def radix_sort(arr):
        """
        Radix Sort implementation
        Time Complexity: O(d * (n + k)) where d is number of digits
        """
        def counting_sort(arr, exp):
            n = len(arr)
            output = [0] * n
            count = [0] * 10

            for i in range(n):
                index = arr[i] // exp
                count[index % 10] += 1

            for i in range(1, 10):
                count[i] += count[i - 1]
```

```
            i = n - 1
            while i >= 0:
                index = arr[i] // exp
                output[count[index % 10] - 1] = arr[i]
                count[index % 10] -= 1
                i -= 1

            for i in range(n):
                arr[i] = output[i]

        if not arr:
            return arr

        max_num = max(arr)
        exp = 1
        while max_num // exp > 0:
            counting_sort(arr, exp)
            exp *= 10

        return arr

# Test implementation
if __name__ == "__main__":
    # Test arrays
    test_arrays = [
        [64, 34, 25, 12, 22, 11, 90],
        [38, 27, 43, 3, 9, 82, 10],
        [170, 45, 75, 90, 802, 24, 2, 66]
    ]

    sorter = SortingAlgorithms()

    for i, arr in enumerate(test_arrays, 1):
        print(f"\nTest Array {i}: {arr}")

        # Test each sorting algorithm
        print("Bubble Sort:", sorter.bubble_sort(arr.copy()))
        print("Insertion Sort:", sorter.insertion_sort(arr.copy()))
        print("Quick Sort:", sorter.quick_sort(arr.copy()))
        print("Merge Sort:", sorter.merge_sort(arr.copy()))
        print("Heap Sort:", sorter.heap_sort(arr.copy()))
        print("Radix Sort:", sorter.radix_sort(arr.copy()))
```

```
Test Array 1: [64, 34, 25, 12, 22, 11, 90]
Bubble Sort: [11, 12, 22, 25, 34, 64, 90]
Insertion Sort: [11, 12, 22, 25, 34, 64, 90]
Quick Sort: [11, 12, 22, 25, 34, 64, 90]
Merge Sort: [11, 12, 22, 25, 34, 64, 90]
Heap Sort: [11, 12, 22, 25, 34, 64, 90]
Radix Sort: [11, 12, 22, 25, 34, 64, 90]

Test Array 2: [38, 27, 43, 3, 9, 82, 10]
Bubble Sort: [3, 9, 10, 27, 38, 43, 82]
Insertion Sort: [3, 9, 10, 27, 38, 43, 82]
Quick Sort: [3, 9, 10, 27, 38, 43, 82]
Merge Sort: [3, 9, 10, 27, 38, 43, 82]
Heap Sort: [3, 9, 10, 27, 38, 43, 82]
Radix Sort: [3, 9, 10, 27, 38, 43, 82]

Test Array 3: [170, 45, 75, 90, 802, 24, 2, 66]
Bubble Sort: [2, 24, 45, 66, 75, 90, 170, 802]
Insertion Sort: [2, 24, 45, 66, 75, 90, 170, 802]
Quick Sort: [2, 24, 45, 66, 75, 90, 170, 802]
Merge Sort: [2, 24, 45, 66, 75, 90, 170, 802]
Heap Sort: [2, 24, 45, 66, 75, 90, 170, 802]
Radix Sort: [2, 24, 45, 66, 75, 90, 170, 802]
```