1.Factorial and recursion:-

```
# Factorial using recursion
def factorial(n):
    print(f"Called factorial({n})")  # Trace call
    if n < 0:
        raise ValueError("Negative numbers not allowed")
    if n == 0 or n == 1:
        print(f"Returning 1 since n = {n}")
        return 1
    result = n * factorial(n - 1)
    print(f"Computed factorial({n}) = {result}")
    return result


# Fibonacci using recursion
def fibonacci(n):
    print(f"Called fibonacci({n})")  # Trace call
    if n < 0:
        raise ValueError("Negative numbers not allowed")
    if n == 0:
        print("Returning 0 (base case)")
        return 0
    if n == 1:
        print("Returning 1 (base case)")
        return 1
    result = fibonacci(n - 1) + fibonacci(n - 2)
    print(f"Computed fibonacci({n}) = {result}")
    return result


# Main program
```

```
if __name__ == "__main__":
    print("=== Factorial using Recursion ===")
    n = int(input("Enter a number to find its factorial: "))
    print(f"\nCalculating factorial({n})...")
    fact_result = factorial(n)
    print(f"\n✅ Factorial of {n} = {fact_result}")

    print("\n=== Fibonacci using Recursion ===")
    m = int(input("Enter how many Fibonacci terms to display: "))
    print(f"\nCalculating Fibonacci sequence for 0 to {m-1}...")
    fib_sequence = []
    for i in range(m):
        fib_value = fibonacci(i)
        fib_sequence.append(fib_value)
        print(f"Fibonacci({i}) = {fib_value}")
    print(f"\n✅ Fibonacci sequence (0..{m-1}): {fib_sequence}")
```

2. **Analyze time complexity: recursive vs iterative**

```
import time


def fib_recursive(n):
    if n <= 1:
        return n
    return fib_recursive(n-1) + fib_recursive(n-2)


def fib_iterative(n):
    a, b = 0, 1
    for i in range(2, n+1):
        a, b = b, a + b
        print(f"Step {i}: {b}")  # Print after each iteration
    return b
```

```
# Main program
n = int(input("Enter n: "))

# Iterative
print("\n--- Iterative Fibonacci ---")
t0 = time.time()
iter_res = fib_iterative(n)
t1 = time.time()
print(f"Iterative result: {iter_res}, Time: {t1 - t0:.6f}s")

# Recursive
print("\n--- Recursive Fibonacci ---")
t0 = time.time()
rec_res = fib_recursive(n)
t1 = time.time()
print(f"Recursive result: {rec_res}, Time: {t1 - t0:.6f}s")
```

3. Array operations: insertion, deletion, search, traversal

```
# Array operations using Python list
def insert(arr, index, value):
    arr.insert(index, value)
    print(f"After insertion of {value} at index {index}: {arr}")

def delete(arr, index):
    if 0 <= index < len(arr):
        removed = arr.pop(index)
        print(f"Deleted {removed} from index {index}: {arr}")
    else:
        print("Index out of range")

def search(arr, value):
```

```
    for i, v in enumerate(arr):
        if v == value:
            print(f"Value {value} found at index {i}")
            return i
    print(f"Value {value} not found")
    return -1

def traverse(arr):
    print("Array elements:", arr)

# Main program
if __name__ == "__main__":
    arr = list(map(int, input("Enter initial array elements (space-separated): ").split()))
    print("Initial array:", arr)

    val = int(input("\nEnter value to insert: "))
    idx = int(input("Enter index to insert at: "))
    insert(arr, idx, val)

    val = int(input("\nEnter value to search: "))
    search(arr, val)

    idx = int(input("\nEnter index to delete: "))
    delete(arr, idx)

    print("\nFinal array traversal:")
    traverse(arr)
```

4. Singly Linked List with all operations

```
class Node:
    def __init__(self, data):
```

```python
        self.data = data
        self.next = None

class SinglyLinkedList:
    def __init__(self):
        self.head = None

    def insert_at_begin(self, data):
        node = Node(data)
        node.next = self.head
        self.head = node
        print(f"Inserted {data} at beginning.")

    def insert_at_end(self, data):
        node = Node(data)
        if not self.head:
            self.head = node
        else:
            cur = self.head
            while cur.next:
                cur = cur.next
            cur.next = node
        print(f"Inserted {data} at end.")

    def delete(self, key):
        cur, prev = self.head, None
        while cur and cur.data != key:
            prev, cur = cur, cur.next
        if not cur:
            print(f"{key} not found — deletion failed.")
            return
```

```
        if not prev:
            self.head = cur.next
        else:
            prev.next = cur.next
        print(f"Deleted {key} from list.")

    def search(self, key):
        cur, index = self.head, 0
        while cur:
            if cur.data == key:
                print(f"Found {key} at position {index}.")
                return index
            cur, index = cur.next, index + 1
        print(f"{key} not found in list.")
        return -1

    def traverse(self):
        print("Current list:", end=" ")
        cur = self.head
        while cur:
            print(cur.data, end=" ")
            cur = cur.next
        print()



if __name__ == "__main__":
    s = SinglyLinkedList()

    # initial nodes
    n = int(input("Enter number of initial elements: "))
    for _ in range(n):
```

```
        val = int(input("Enter value: "))
        s.insert_at_end(val)
    s.traverse()

    # insert at beginning
    val = int(input("\nEnter value to insert at beginning: "))
    s.insert_at_begin(val)
    s.traverse()

    # insert at end
    val = int(input("\nEnter value to insert at end: "))
    s.insert_at_end(val)
    s.traverse()

    # search element
    val = int(input("\nEnter value to search: "))
    s.search(val)

    # delete element
    val = int(input("\nEnter value to delete: "))
    s.delete(val)
    s.traverse()
```

5. Doubly Linked List

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


class DoublyLinkedList:
```

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

def insert_end(self, data):
    node = Node(data)
    if not self.head:
        self.head = node
        print(f"Inserted {data} as the first node.")
        return
    cur = self.head
    while cur.next:
        cur = cur.next
    cur.next = node
    node.prev = cur
    print(f"Inserted {data} at the end.")

def delete(self, key):
    cur = self.head
    while cur and cur.data != key:
        cur = cur.next
    if not cur:
        print(f"{key} not found — deletion failed.")
        return
    if cur.prev:
        cur.prev.next = cur.next
    else:
        self.head = cur.next
    if cur.next:
        cur.next.prev = cur.prev
    print(f"Deleted {key} from list.")
```

```python
    def traverse(self):
        cur = self.head
        if not cur:
            print("List is empty.")
            return
        print("Current list:", end=" ")
        while cur:
            print(cur.data, end=" ")
            cur = cur.next
        print()


if __name__ == "__main__":
    d = DoublyLinkedList()

    # Insert user-defined elements
    n = int(input("Enter number of elements to insert: "))
    for _ in range(n):
        val = int(input("Enter value: "))
        d.insert_end(val)
        d.traverse()

    # Delete a value
    val = int(input("\nEnter value to delete: "))
    d.delete(val)
    d.traverse()

    # Insert one more at end
    val = int(input("\nEnter value to insert at end: "))
    d.insert_end(val)
    d.traverse()
```

6. Circular Linked List

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

class CircularLinkedList:
    def __init__(self):
        self.tail = None

    def insert(self, data):
        node = Node(data)
        if not self.tail:
            self.tail = node
            node.next = node
            print(f"Inserted {data} as the first node.")
            return
        node.next = self.tail.next
        self.tail.next = node
        self.tail = node
        print(f"Inserted {data} at the end of the circular list.")

    def traverse(self):
        if not self.tail:
            print("List is empty.")
            return
        print("Current circular list:", end=" ")
        cur = self.tail.next
        while True:
            print(cur.data, end=" ")
            cur = cur.next
```

```python
            if cur == self.tail.next:
                break
        print()

if __name__ == "__main__":
    c = CircularLinkedList()

    # Insert user-defined nodes
    n = int(input("Enter number of elements to insert: "))
    for _ in range(n):
        val = int(input("Enter value: "))
        c.insert(val)
        c.traverse()

    print("\nFinal circular linked list:")
    c.traverse()
```

7. Stack using arrays

```python
class StackArray:
    def __init__(self):
        self.stack = []

    def push(self, x):
        self.stack.append(x)
        print(f"Pushed {x}. Stack now: {self.stack}")

    def pop(self):
        if not self.stack:
            print("Stack is empty! Cannot pop.")
            return
        popped = self.stack.pop()
        print(f"Popped {popped}. Stack now: {self.stack}")
```

```
        return popped

    def peek(self):
        if not self.stack:
            print("Stack is empty! Nothing to peek.")
            return None
        print(f"Top element is {self.stack[-1]}")
        return self.stack[-1]


if __name__ == "__main__":
    s = StackArray()

    n = int(input("Enter number of elements to push: "))
    for _ in range(n):
        val = int(input("Enter value: "))
        s.push(val)

    print("\nPerforming peek operation:")
    s.peek()

    print("\nPerforming pop operation:")
    s.pop()

    print("\nFinal stack state:", s.stack)
```

8. Stack using linked list

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

```
class StackLinked:
  def __init__(self):
    self.top = None

  def push(self, data):
    node = Node(data)
    node.next = self.top
    self.top = node
    print(f"Pushed {data} onto stack.")

  def pop(self):
    if not self.top:
      print("Stack is empty! Cannot pop.")
      return None
    val = self.top.data
    self.top = self.top.next
    print(f"Popped {val} from stack.")
    return val

  def peek(self):
    if not self.top:
      print("Stack is empty! Nothing to peek.")
      return None
    print(f"Top element is {self.top.data}")
    return self.top.data

  def display(self):
    if not self.top:
      print("Stack is empty.")
      return
```

```
        print("Current stack (top to bottom):", end=" ")
        cur = self.top
        while cur:
            print(cur.data, end=" ")
            cur = cur.next
        print()


if __name__ == "__main__":
    s = StackLinked()

    n = int(input("Enter number of elements to push: "))
    for _ in range(n):
        val = int(input("Enter value: "))
        s.push(val)
        s.display()

    print("\nPerforming peek operation:")
    s.peek()

    print("\nPerforming pop operation:")
    s.pop()
    s.display()
```

11. Queue using arrays

```
class QueueArray:
    def __init__(self):
        self.q = []

    def enqueue(self, x):
        self.q.append(x)
        print(f"Enqueued {x}. Queue now: {self.q}")
```

```python
    def dequeue(self):
        if not self.q:
            print("Queue is empty! Cannot dequeue.")
            return None
        val = self.q.pop(0)
        print(f"Dequeued {val}. Queue now: {self.q}")
        return val

    def peek(self):
        if not self.q:
            print("Queue is empty! Nothing to peek.")
            return None
        print(f"Front element is {self.q[0]}")
        return self.q[0]

    def display(self):
        print("Current queue:", self.q)


if __name__ == "__main__":
    q = QueueArray()

    n = int(input("Enter number of elements to enqueue: "))
    for _ in range(n):
        val = int(input("Enter value: "))
        q.enqueue(val)

    print("\nPerforming peek operation:")
    q.peek()
```

```
    print("\nPerforming dequeue operation:")
    q.dequeue()

    print("\nFinal queue state:")
    q.display()
```

12. Circular Queue

```
class CircularQueue:
    def __init__(self, k):
        self.size = k
        self.q = [None]*k
        self.head = -1
        self.tail = -1

    def enqueue(self, val):
        if (self.tail + 1) % self.size == self.head:
            print("Queue is full! Cannot enqueue.")
            return
        if self.head == -1:
            self.head = 0
        self.tail = (self.tail + 1) % self.size
        self.q[self.tail] = val
        print(f"Enqueued {val}. Queue: {self.q}, head={self.head}, tail={self.tail}")

    def dequeue(self):
        if self.head == -1:
            print("Queue is empty! Cannot dequeue.")
            return None
        val = self.q[self.head]
        if self.head == self.tail:
            self.head = self.tail = -1
        else:
```

```python
            self.head = (self.head + 1) % self.size
        print(f"Dequeued {val}. Queue: {self.q}, head={self.head}, tail={self.tail}")
        return val

    def display(self):
        if self.head == -1:
            print("Queue is empty.")
            return
        print("Queue elements:", end=" ")
        i = self.head
        while True:
            print(self.q[i], end=" ")
            if i == self.tail:
                break
            i = (i + 1) % self.size
        print()


if __name__ == "__main__":
    k = int(input("Enter size of circular queue: "))
    cq = CircularQueue(k)

    n = int(input("Enter number of elements to enqueue: "))
    for _ in range(n):
        val = int(input("Enter value: "))
        cq.enqueue(val)
        cq.display()

    print("\nPerforming dequeue operation:")
    cq.dequeue()
    cq.display()
```

13. Queue using linked list

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


class QueueLinked:
    def __init__(self):
        self.front = None
        self.rear = None

    def enqueue(self, data):
        node = Node(data)
        if not self.rear:
            self.front = self.rear = node
            print(f"Enqueued {data} (first element).")
            return
        self.rear.next = node
        self.rear = node
        print(f"Enqueued {data}.")

    def dequeue(self):
        if not self.front:
            print("Queue is empty! Cannot dequeue.")
            return None
        val = self.front.data
        self.front = self.front.next
        if not self.front:
            self.rear = None
        print(f"Dequeued {val}.")
        return val
```

```
    def display(self):
        if not self.front:
            print("Queue is empty.")
            return
        cur = self.front
        print("Current queue:", end=" ")
        while cur:
            print(cur.data, end=" ")
            cur = cur.next
        print()


if __name__ == "__main__":
    ql = QueueLinked()

    n = int(input("Enter number of elements to enqueue: "))
    for _ in range(n):
        val = int(input("Enter value: "))
        ql.enqueue(val)
        ql.display()

    print("\nPerforming dequeue operation:")
    ql.dequeue()
    ql.display()
```

14. Binary Tree (linked representation)

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

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

    def insert_level_order(self, data):
        node = Node(data)
        if not self.root:
            self.root = node
            print(f"Inserted {data} as root node.")
            return
        q = [self.root]
        while q:
            cur = q.pop(0)
            if not cur.left:
                cur.left = node
                print(f"Inserted {data} to the left of {cur.data}")
                return
            else:
                q.append(cur.left)
            if not cur.right:
                cur.right = node
                print(f"Inserted {data} to the right of {cur.data}")
                return
            else:
                q.append(cur.right)

    def level_order_traversal(self):
        if not self.root:
            print("Tree is empty.")
            return
```

```
        q = [self.root]
        print("Level-order traversal:", end=" ")
        while q:
            cur = q.pop(0)
            print(cur.data, end=" ")
            if cur.left:
                q.append(cur.left)
            if cur.right:
                q.append(cur.right)
        print()


if __name__ == "__main__":
    bt = BinaryTree()
    n = int(input("Enter number of nodes to insert: "))
    for _ in range(n):
        val = int(input("Enter node value: "))
        bt.insert_level_order(val)
        bt.level_order_traversal()
```

15. Inorder, Preorder, Postorder traversals

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


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


    def insert_level_order(self, data):
        node = Node(data)
```

```
        if not self.root:
            self.root = node
            return
        q = [self.root]
        while q:
            cur = q.pop(0)
            if not cur.left:
                cur.left = node
                return
            else:
                q.append(cur.left)
            if not cur.right:
                cur.right = node
                return
            else:
                q.append(cur.right)

    def inorder(self, node):
        if node:
            self.inorder(node.left)
            print(node.data, end=" ")
            self.inorder(node.right)

    def preorder(self, node):
        if node:
            print(node.data, end=" ")
            self.preorder(node.left)
            self.preorder(node.right)

    def postorder(self, node):
        if node:
```

```
        self.postorder(node.left)
        self.postorder(node.right)
        print(node.data, end=" ")

# Main
bt = BinaryTree()
for _ in range(int(input("Number of nodes: "))):
    bt.insert_level_order(int(input("Node value: ")))

print("Inorder:", end=" "); bt.inorder(bt.root); print()
print("Preorder:", end=" "); bt.preorder(bt.root); print()
print("Postorder:", end=" "); bt.postorder(bt.root); print()
```

16. BST with insert and search

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

def bst_insert(root, key):
    if not root:
        print(f"Inserted {key}.")
        return BSTNode(key)
    if key < root.key:
        root.left = bst_insert(root.left, key)
    else:
        root.right = bst_insert(root.right, key)
    return root

def bst_search(root, key):
```

```
    if not root:
        return None
    if root.key == key:
        return root
    if key < root.key:
        return bst_search(root.left, key)
    return bst_search(root.right, key)

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

# Main program
root = None
n = int(input("Enter number of nodes to insert: "))
for _ in range(n):
    val = int(input("Enter node value: "))
    root = bst_insert(root, val)

print("\nInorder traversal of BST:", end=" ")
inorder(root)
print()

search_val = int(input("\nEnter value to search: "))
res = bst_search(root, search_val)
if res:
    print(f"Value {search_val} found in BST.")
else:
    print(f"Value {search_val} not found in BST.")
```

17. Delete node from BST

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

def bst_insert(root, key):
    if not root:
        return BSTNode(key)
    if key < root.key:
        root.left = bst_insert(root.left, key)
    else:
        root.right = bst_insert(root.right, key)
    return root

def bst_delete(root, key):
    if not root:
        return root
    if key < root.key:
        root.left = bst_delete(root.left, key)
    elif key > root.key:
        root.right = bst_delete(root.right, key)
    else:
        if not root.left:
            return root.right
        if not root.right:
            return root.left
        succ = root.right
        while succ.left:
            succ = succ.left
```

```
        root.key = succ.key
        root.right = bst_delete(root.right, succ.key)
    return root

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

# Main
root = None
n = int(input("Enter number of nodes to insert: "))
for _ in range(n):
    val = int(input("Enter node value: "))
    root = bst_insert(root, val)

print("\nInorder before deletion:", end=" ")
inorder(root)
print()

del_val = int(input("\nEnter value to delete: "))
root = bst_delete(root, del_val)

print("\nInorder after deletion:", end=" ")
inorder(root)
print()
```

18. Priority Queue using heap (heapq)

```
import heapq

# Initialize empty heap
```

```
heap = []

# Input number of tasks
n = int(input("Number of tasks: "))
for _ in range(n):
    task = input("Task name: ")
    priority = int(input("Priority (smaller = higher): "))
    heapq.heappush(heap, (priority, task))  # Push as (priority, task)
    print(f"Queue: {[t for p, t in heap]}")

# Pop highest-priority task
if heap:
    priority, task = heapq.heappop(heap)
    print(f"Popped task: {task} with priority {priority}")
    print(f"Queue now: {[t for p, t in heap]}")
else:
    print("Queue is empty!")
```