1. Implement a singly linked list with the following operations:

- Insert at beginning, end, and given position

- Delete from beginning, end, and given position

- Display the list

Code:

```
#include <iostream>
using namespace std;

// Node structure
struct Node {
   int data;
   Node* next;
};

// Head of the list
Node* head = nullptr;

// Insert at the beginning
void insertAtBeginning(int value) {
   Node* newNode = new Node();
   newNode->data = value;
   newNode->next = head;
   head = newNode;
}

// Insert at the end
void insertAtEnd(int value) {
   Node* newNode = new Node();
   newNode->data = value;
   newNode->next = nullptr;

   if (head == nullptr) {
      head = newNode;
      return;
   }

   Node* temp = head;
   while (temp->next != nullptr) {
      temp = temp->next;
   }

   temp->next = newNode;
}

// Insert at given position (1-based index)
```

```
    if (head->next == nullptr) {
     delete head;
     head = nullptr;
      return;
 }

   Node* temp = head;
   while (temp->next->next != nullptr) {
      temp = temp->next;
   }

   delete temp->next;
   temp->next = nullptr;
}
// Delete from given position (1-based index)
void deleteFromPosition(int position) {
   if (head == nullptr) {
      cout << "List is empty!" << endl;
      return;
   }

   if (position == 1) {
      deleteFromBeginning();
      return;
   }

   Node* temp = head;
   for (int i = 1; i < position - 1 && temp != nullptr;
         i++) {
      temp = temp->next;
   }

   if (temp == nullptr || temp->next == nullptr) {
      cout << "Invalid position!" << endl;
      return;
   }

   Node* toDelete = temp->next;
   temp->next = toDelete->next;
   delete toDelete;
}
```

```
        void insertAtPosition(int value, int
        position) {
  if (position == 1) {
    insertAtBeginning(value);
    return;
  }

  Node* newNode = new Node();
  newNode->data = value;

  Node* temp = head;
  for (int i = 1; i < position - 1 && temp !=
        nullptr; i++) {
    temp = temp->next;
  }

  if (temp == nullptr) {
    cout << "Invalid position!" << endl;
    return;
  }

  newNode->next = temp->next;
  temp->next = newNode;
}

// Delete from beginning
void deleteFromBeginning() {
  if (head == nullptr) {
    cout << "List is empty!" << endl;
    return;
  }

  Node* temp = head;
  head = head->next;
  delete temp;
}

// Delete from end
void deleteFromEnd() {
  if (head == nullptr) {
    cout << "List is empty!" << endl;
    return;
  }
```

```

// Display the list
void display() {
  if (head == nullptr) {
    cout << "List is empty!" << endl;
    return;
  }

  Node* temp = head;
  cout << "List: ";
  while (temp != nullptr) {
    cout << temp->data << " -> ";
    temp = temp->next;
  }
  cout << "NULL" << endl;
}

// Main function to test
int main() {
  insertAtBeginning(10);  // 10
  insertAtEnd(20);      // 10 -> 20
  insertAtPosition(15, 2);// 10 -> 15 -> 20
  display();

  deleteFromBeginning();  // 15 -> 20
  display();

  deleteFromEnd();      // 15
  display();

  insertAtEnd(25);      // 15 -> 25
  insertAtEnd(30);      // 15 -> 25 -> 30
  deleteFromPosition(2);  // 15 -> 30
  display();

  return 0;
```

Output:

```
List: 10 -> 15 -> 20 -> NULL
List: 15 -> 20 -> NULL
List: 15 -> NULL
List: 15 -> 30 -> NULL

Process returned 0 (0x0)   execution time : 0.125 s
Press any key to continue.
```

2.Implement a stack using an array with the following operations:
Push,Pop,Peek,Display

Code:

```
#include <iostream>
using namespace std;

#define SIZE 100  // Max size of the stack

int stack[SIZE];
int top = -1; // Stack is empty initially

// Push an element onto the stack
void push(int value) {
  if (top >= SIZE - 1) {
    cout << "Stack Overflow! Cannot push " << value << endl;
    return;
  }
  top++;
  stack[top] = value;
  cout << value << " pushed to stack." << endl;
}

// Pop an element from the stack
void pop() {
  if (top < 0) {
    cout << "Stack Underflow! Cannot pop." << endl;
    return;
  }
  cout << "Popped element: " << stack[top] << endl;
  top--;
}
```

```
// Display all elements in the stack
void display() {
  if (top < 0) {
    cout << "Stack is empty!" << endl;
    return;
  }

  cout << "Stack elements: ";
  for (int i = top; i >= 0; i--) {
    cout << stack[i] << " ";
  }
  cout << endl;
}

// Main function to test
int main() {
  push(10);  // Stack: 10
  push(20);  // Stack: 20, 10
  push(30);  // Stack: 30, 20, 10
  display(); // Show stack

  peek();    // Show top (30)

  pop();     // Remove 30
  display(); // Show stack

  return 0;
}
```

```
// Peek the top element
void peek() {
  if (top < 0) {
    cout << "Stack is empty!" << endl;
    return;
  }
  cout << "Top element: " << stack[top] << endl;
```

Ouput:

```
10 pushed to stack.
20 pushed to stack.
30 pushed to stack.
Stack elements: 30 20 10
Top element: 30
Popped element: 30
Stack elements: 20 10

Process returned 0 (0x0)   execution time : 0.143 s
Press any key to continue.
```

3.Implement a queue using an array with the following operations:

Enqueue, Dequeue , Peek, Display

Code:

```
#include <iostream>
using namespace std;

#define SIZE 100

int queue[SIZE];
int front = -1, rear = -1;

// Enqueue (add item)
void enqueue(int value) {
  if (rear == SIZE - 1) {
    cout << "Queue Overflow! Cannot add " <<
value << endl;
    return;
  }

  if (front == -1) front = 0; // First element

  rear++;
  queue[rear] = value;
```

```
  if (front == -1 || front > rear) {
    cout << "Queue is empty!" << endl;
    return;
  }

  cout << "Front element: " << queue[front] <<
endl;
}
// Display all items
void display() {
  if (front == -1 || front > rear) {
    cout << "Queue is empty!" << endl;
    return;
  }

  cout << "Queue elements: ";
  for (int i = front; i <= rear; i++) {
    cout << queue[i] << " ";
  }
  cout << endl;
```

```
    cout << value << " enqueued." << endl;
}

// Dequeue (remove item)
void dequeue() {
    if (front == -1 || front > rear) {
        cout << "Queue Underflow! Nothing to
dequeue." << endl;
        return;
    }

    cout << "Dequeued: " << queue[front] << end
    front++;
}
// Peek (view front)
void peek() {
```

```
}

// Main function to test
int main() {
    enqueue(10);  // Queue: 10
    enqueue(20);  // Queue: 10, 20
    enqueue(30);  // Queue: 10, 20, 30
    display();    // Show queue

    peek();       // Show front (10)

    dequeue();    // Remove 10
    display();    // Queue: 20, 30

    return 0;
}
```

Output:

```
10 enqueued.
20 enqueued.
30 enqueued.
Queue elements: 10 20 30
Front element: 10
Dequeued: 10
Queue elements: 20 30

Process returned 0 (0x0)   execution time : 0.140 s
Press any key to continue.
```

4. Implement a binary tree and insert nodes into the binary tree recursively with the following traversals:

In-order, Pre-order, Post-order

Code:

```
#include <iostream>
using namespace std;

// Define a node of the binary tree
struct Node {
    int data;
    Node* left;
    Node* right;
```

```
// Pre-order traversal (Root → Left → Right)
void preorder(Node* root) {
    if (root != nullptr) {
        cout << root->data << " ";
        preorder(root->left);
        preorder(root->right);
    }
}
```

```
};

// Create a new node with given value
Node* createNode(int value) {
  Node* newNode = new Node(); // Allocate
memory
  newNode->data = value;
  newNode->left = nullptr;
  newNode->right = nullptr;
  return newNode;
}

// Insert node into binary tree recursively (user
decides placement)
Node* insertNode(Node* root, int value) {
  if (root == nullptr) {
    return createNode(value);
  }

  if (value < root->data) {
    root->left = insertNode(root->left, value);  /
Insert left
  } else {
    root->right = insertNode(root->right, value)
// Insert right
  }

  return root;
}

// In-order traversal (Left → Root → Right)
void inorder(Node* root) {
  if (root != nullptr) {
    inorder(root->left);
    cout << root->data << " ";
    inorder(root->right);
  }
}
```

```
// Post-order traversal (Left → Right → Root)
void postorder(Node* root) {
  if (root != nullptr) {
    postorder(root->left);
    postorder(root->right);
    cout << root->data << " ";
  }
}

// Main function to test
int main() {
  Node* root = nullptr;

  // Inserting nodes
  root = insertNode(root, 50);
  root = insertNode(root, 30);
  root = insertNode(root, 70);
  root = insertNode(root, 20);
  root = insertNode(root, 40);
  root = insertNode(root, 60);
  root = insertNode(root, 80);

  cout << "In-order traversal: ";
  inorder(root);
  cout << endl;

  cout << "Pre-order traversal: ";
  preorder(root);
  cout << endl;

  cout << "Post-order traversal: ";
  postorder(root);
  cout << endl;

  return 0;
}
```

Output:

```
In-order traversal: 20 30 40 50 60 70 80
Pre-order traversal: 50 30 20 40 70 60 80
Post-order traversal: 20 40 30 60 80 70 50

Process returned 0 (0x0)   execution time : 0.128 s
Press any key to continue.
```

5. Count the total number of nodes and leaf nodes in a binary tree.

Code:

```
#include <iostream>
using namespace std;

// Define the structure for a tree node
struct Node {
    int data;
    Node* left;
    Node* right;
};

// Function to create a new node
Node* createNode(int value) {
    Node* newNode = new Node();
    newNode->data = value;
    newNode->left = nullptr;
    newNode->right = nullptr;
    return newNode;
}

// Insert node into binary tree (recursive)
Node* insertNode(Node* root, int value) {
    if (root == nullptr)
        return createNode(value);

    if (value < root->data)
        root->left = insertNode(root->left, value);
    else
        root->right = insertNode(root->right, value);

    return root;
}

// Count total number of nodes
int countTotalNodes(Node* root) {
    if (root == nullptr)
        return 0;
    return 1 + countTotalNodes(root->left) + countTotalNodes(root->right);
}

// Count only leaf nodes (no children)
int countLeafNodes(Node* root) {
    if (root == nullptr)
        return 0;

    if (root->left == nullptr && root->right == nullptr)
        return 1;

    return countLeafNodes(root->left) + countLeafNodes(root->right);
}

// Main function
int main() {
    Node* root = nullptr;

    // Insert nodes into the tree
    root = insertNode(root, 50);
    root = insertNode(root, 30);
    root = insertNode(root, 70);
    root = insertNode(root, 20);
    root = insertNode(root, 40);
    root = insertNode(root, 60);
    root = insertNode(root, 80);

    // Count and display
    cout << "Total number of nodes: " << countTotalNodes(root) << endl;
    cout << "Total number of leaf nodes: " << countLeafNodes(root) << endl;

    return 0;
}
```

Output:

```
Total number of nodes: 7
Total number of leaf nodes: 4

Process returned 0 (0x0)   execution time : 0.155 s
Press any key to continue.
```

6. Implement a Binary Search Tree (BST) with In-order Traversal.

Write recursive functions for the following:

- Factorial of a number
- Fibonacci series
- Binary search
- Tower of Hanoi

Code:

```
#include <iostream>
using namespace std;

// ========== BST ========== //
struct Node {
    int data;
    Node* left;
    Node* right;
};

// Create new node
Node* createNode(int value) {
    Node* newNode = new Node();
    newNode->data = value;
    newNode->left = nullptr;
    newNode->right = nullptr;
    return newNode;
}

// Insert node in BST
Node* insertBST(Node* root, int value) {
    if (root == nullptr)
        return createNode(value);

    if (value < root->data)
        root->left = insertBST(root->left, value);
    else
        root->right = insertBST(root->right, value);
```

```
        else
            return binarySearch(arr, mid + 1, right,
key);
    }

    return -1;
}
// === Tower of Hanoi (Recursive) ==== //
void towerOfHanoi(int n, char from, char
temp, char to) {
    if (n == 1) {
        cout << "Move disk 1 from " << from << "
to " << to << endl;
        return;
    }
    towerOfHanoi(n - 1, from, to, temp);
    cout << "Move disk " << n << " from " <<
from << " to " << to << endl;
    towerOfHanoi(n - 1, temp, from, to);
}

// ========== MAIN FUNCTION ==========
//
int main() {
    // ----- BST with In-order Traversal -----
    Node* root = nullptr;
    root = insertBST(root, 50);
    insertBST(root, 30);
    insertBST(root, 70);
```

```
    return root;
}

// In-order traversal (Left -> Root -> Right)
void inorder(Node* root) {
    if (root != nullptr) {
        inorder(root->left);
        cout << root->data << " ";
        inorder(root->right);
    }
}

// ========== Factorial (Recursive)
========== //
int factorial(int n) {
    if (n <= 1)
        return 1;
    return n * factorial(n - 1);
}

// ========== Fibonacci (Recursive)
========== //
int fibonacci(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    return fibonacci(n - 1) + fibonacci(n - 2);
}

// ========== Binary Search (Recursive)
========== //
int binarySearch(int arr[], int left, int right, int
key) {
    if (left <= right) {
        int mid = (left + right) / 2;

        if (arr[mid] == key)
            return mid;

        if (key < arr[mid])
            return binarySearch(arr, left, mid - 1,
key);
```

```
    insertBST(root, 20);
    insertBST(root, 40);
    insertBST(root, 60);
    insertBST(root, 80);

    cout << "In-order Traversal (BST): ";
    inorder(root);
    cout << endl;

    // ----- Factorial -----
    int num = 5;
    cout << "Factorial of " << num << " = " <<
factorial(num) << endl;

    // ----- Fibonacci -----
    cout << "Fibonacci Series (first 6 numbers):
";
    for (int i = 0; i < 6; i++) {
        cout << fibonacci(i) << " ";
    }
    cout << endl;

    // ----- Binary Search -----
    int arr[] = {10, 20, 30, 40, 50};
    int key = 30;
    int index = binarySearch(arr, 0, 4, key);
    if (index != -1)
        cout << "Binary Search: " << key << "
found at index " << index << endl;
    else
        cout << "Binary Search: " << key << " not
found" << endl;

    // ----- Tower of Hanoi -----
    int disks = 3;
    cout << "Tower of Hanoi (" << disks << "
disks):" << endl;
    towerOfHanoi(disks, 'A', 'B', 'C');

    return 0;
}
```

Output:

```
In-order Traversal (BST): 20 30 40 50 60 70 80
Factorial of 5 = 120
Fibonacci Series (first 6 numbers): 0 1 1 2 3 5
Binary Search: 30 found at index 2
Tower of Hanoi (3 disks):
Move disk 1 from A to C
Move disk 2 from A to B
Move disk 1 from C to B
Move disk 3 from A to C
Move disk 1 from B to A
Move disk 2 from B to C
Move disk 1 from A to C

Process returned 0 (0x0)   execution time : 0.151 s
Press any key to continue.
```