Problem Statement: Linked List Manipulation Using OOP in C++

Objective: Implement a singly linked list in C++ using Object-Oriented Programming (OOP) principles. The implementation should support the following operations:

1. **Insertion**:
   - At the beginning of the list.
   - At the end of the list.
   - At a specific position in the list.
2. **Deletion**:
   - Delete a node at a specific position.
3. **Traversal**:
   - Display the elements of the linked list.

Requirements:

1. **Class Design**:
   - Design a `Node` class to represent the nodes in the linked list.
   - Design a `LinkedList` class to encapsulate the operations on the linked list.
2. **Node Class**:
   - Data Members:
     - `int data`: To store the data of the node.
     - `Node* next`: Pointer to the next node in the list.
   - Methods:
     - Constructor to initialize the data and the next pointer.
3. **LinkedList Class**:
   - Data Members:
     - `Node* head`: Pointer to the head of the linked list.
   - Methods:
     - `void insertAtBeginning(int data)`: Insert a node at the beginning of the list.
     - `void insertAtEnd(int data)`: Insert a node at the end of the list.
     - `void insertAtPosition(int data, int position)`: Insert a node at a specific position in the list.
     - `void deleteAtPosition(int position)`: Delete a node at a specific position.
     - `void traverse()`: Display the elements of the linked list.

Implementation Details:

1. **Node Class**:
   - The `Node` class should have a constructor to initialize the `data` and `next` pointer.
2. **LinkedList Class**:
   - The `LinkedList` class should manage the head of the list and provide methods for insertion, deletion, and traversal.
   - Insertion at the beginning should update the head of the list.

- Insertion at the end should traverse to the last node and update its next pointer.
- Insertion at a specific position should handle edge cases such as inserting at the beginning or end of the list.
- Deletion should handle the removal of the head node and nodes at specific positions.
- Traversal should iterate through the list and print each node's data.

Example Usage

Implement the classes and provide a `main` function to demonstrate the following operations:

1. Create a linked list.
2. Insert elements at the beginning, end, and specific positions.
3. Delete elements at specific positions.
4. Traverse and print the elements of the linked list.

Problem Statement: Stack Implementation Using OOP in C++

Objective: Implement a stack data structure in C++ using Object-Oriented Programming (OOP) principles. The stack should use an array as the underlying storage mechanism and support the following operations:

1. **Push**: Add an element to the top of the stack.
2. **Pop**: Remove and return the top element from the stack.
3. **Peek**: Return the top element without removing it from the stack.
4. **IsEmpty**: Check if the stack is empty.
5. **IsFull**: Check if the stack is full.

Requirements:

1. **Class Design**:
   - Design a `Stack` class to encapsulate the operations and data of the stack.
2. **Stack Class**:
   - Data Members:
     - `int* arr`: Dynamic array to store stack elements.
     - `int top`: Index of the top element in the stack.
     - `int capacity`: Maximum capacity of the stack.
   - Methods:
     - Constructor to initialize the stack with a specified capacity.
     - Destructor to clean up the allocated memory.
     - `void push(int data)`: Add an element to the top of the stack.

        - `int pop()`: Remove and return the top element from the stack.
        - `int peek()`: Return the top element without removing it from the stack.
        - `bool isEmpty()`: Check if the stack is empty.
        - `bool isFull()`: Check if the stack is full.

Implementation Details:

1. **Stack Class**:
    - The `Stack` class should manage an array to store the stack elements and provide methods to perform push, pop, peek, and utility operations.
    - Ensure that the stack does not overflow or underflow. Handle edge cases appropriately.
    - Implement a destructor to free the allocated memory for the stack array.

Example Usage

Implement the `Stack` class and provide a `main` function to demonstrate the following operations:

1. Create a stack with a specified capacity.
2. Push elements onto the stack.
3. Pop elements from the stack.
4. Peek at the top element of the stack.
5. Check if the stack is empty or full.

Problem Statement: Library Management System Using OOP in C++

Objective: Design and implement a Library Management System using Object-Oriented Programming (OOP) principles in C++. The system should manage books, members, and borrowing/returning of books.

Requirements:

1. **Class Design**:
    - **Design a Book class to represent books in the library.**
    - **Design a Member class to represent members of the library.**
    - **Design a Library class to manage the collection of books and members, and handle borrowing and returning of books.**
2. **Book Class**:
    - **Data Members:**
        - **int bookID: Unique identifier for the book.**
        - **std::string title: Title of the book.**
        - **std::string author: Author of the book.**
        - **bool isAvailable: Availability status of the book.**
    - **Methods:**
        - **Constructor to initialize the book details.**
        - **Getter methods for book details.**
        - **Method to check availability and set availability status.**
3. **Member Class**:

   - **Data Members:**
     - **int memberID: Unique identifier for the member.**
     - **std::string name: Name of the member.**
     - **std::vector<int> borrowedBooks: List of book IDs borrowed by the member.**
   - **Methods:**
     - **Constructor to initialize the member details.**
     - **Method to borrow a book.**
     - **Method to return a book.**
     - **Getter methods for member details.**

4. **Library Class**:
   - **Data Members:**
     - **std::vector<Book> books: Collection of books in the library.**
     - **std::vector<Member> members: Collection of members in the library.**
   - **Methods:**
     - **Method to add a new book.**
     - **Method to add a new member.**
     - **Method to borrow a book.**
     - **Method to return a book.**
     - **Method to display all books.**
     - **Method to display all members.**

Example Usage

Implement the classes and provide a main function to demonstrate the following operations:

1. Add books to the library.
2. Add members to the library.
3. Borrow books.
4. Return books.
5. Display the list of books.
6. Display the list of members.

Problem Statement: Binary Search Tree (BST) Operations Using OOP in C++

Objective: Implement a Binary Search Tree (BST) in C++ using Object-Oriented Programming (OOP) principles. The BST should support the following operations:

1. **Insertion**: Insert a node with a given value into the BST.
2. **Deletion**: Delete a node with a given value from the BST.
3. **Search**: Search for a node with a given value in the BST.
4. **Traversal**: Perform in-order, pre-order, and post-order traversals of the BST.

Requirements:

1. **Class Design**:
    - **Design a TreeNode class to represent the nodes in the BST.**
    - **Design a BinarySearchTree class to encapsulate the operations on the BST.**
2. **TreeNode Class**:
    - **Data Members:**
        - **int data: To store the value of the node.**
        - **TreeNode* left: Pointer to the left child node.**
        - **TreeNode* right: Pointer to the right child node.**
    - **Methods:**
        - **Constructor to initialize the node with a given value.**
3. **BinarySearchTree Class**:
    - **Data Members:**
        - **TreeNode* root: Pointer to the root node of the BST.**
    - **Methods:**
        - **Constructor to initialize an empty BST.**
        - **Destructor to clean up the allocated memory.**
        - **void insert(int value): Insert a node with the given value into the BST.**
        - **void deleteValue(int value): Delete a node with the given value from the BST.**
        - **TreeNode* search(int value): Search for a node with the given value in the BST.**
        - **void inorderTraversal(): Perform in-order traversal of the BST.**
        - **void preorderTraversal(): Perform pre-order traversal of the BST.**
        - **void postorderTraversal(): Perform post-order traversal of the BST.**

Example Usage

Implement the classes and provide a main function to demonstrate the following operations:

1. Insert nodes into the BST.
2. Delete nodes from the BST.
3. Search for nodes in the BST.
4. Perform in-order, pre-order, and post-order traversals of the BST.