Question:-6 (Implement Stack ADT Singly Linked List PRE, POST, IN,Fix conversion and evaluation):

```
#include <iostream>
#include <stack>
#include <cmath>
#include <algorithm>  // Include the algorithm header for reverse

using namespace std;

struct Node {
    char data;
    Node* next;
    Node(char val) : data(val), next(nullptr) {}
};

class Stack {
private:
    Node* top;

public:
    Stack() : top(nullptr) {}

    void push(char val) {
        Node* newNode = new Node(val);
        newNode->next = top;
        top = newNode;
    }

    char pop() {
        if (isEmpty()) {
```

```
            cerr << "Error: Stack is empty." << endl;
            return '\0';
        }
        char val = top->data;
        Node* temp = top;
        top = top->next;
        delete temp;
        return val;
    }

    bool isEmpty() {
        return top == nullptr;
    }

    char peek() {
        if (isEmpty()) {
            cerr << "Error: Stack is empty." << endl;
            return '\0';
        }
        return top->data;
    }
};

bool isOperand(char ch) {
    return isalnum(ch);
}

int precedence(char op) {
    return (op == '+' || op == '-') ? 1 : ((op == '*' || op == '/') ? 2 : (op == '^' ? 3 : -1));
}
```

```
string infixToPostfix(string infix) {
    string postfix = "";
    Stack stack;

    for (char ch : infix) {
        if (isOperand(ch))
            postfix += ch;
        else if (ch == '(')
            stack.push(ch);
        else if (ch == ')') {
            while (!stack.isEmpty() && stack.peek() != '(')
                postfix += stack.pop();
            stack.pop(); // Pop the '('
        } else {
            while (!stack.isEmpty() && precedence(ch) <= precedence(stack.peek()))
                postfix += stack.pop();
            stack.push(ch);
        }
    }

    while (!stack.isEmpty())
        postfix += stack.pop();

    return postfix;
}

string infixToPrefix(string infix) {
    reverse(infix.begin(), infix.end());  // Include the algorithm header for reverse
    for (char& ch : infix) {
        if (ch == '(')
            ch = ')';
```

```
        else if (ch == ')')
            ch = '(';
    }

    string postfix = infixToPostfix(infix);
    reverse(postfix.begin(), postfix.end());
    return postfix;
}

int evaluatePostfix(string postfix) {
    Stack stack;

    for (char ch : postfix) {
        if (isOperand(ch))
            stack.push(ch - '0');
        else {
            int operand2 = stack.pop();
            int operand1 = stack.pop();

            switch (ch) {
                case '+':
                    stack.push(operand1 + operand2);
                    break;
                case '-':
                    stack.push(operand1 - operand2);
                    break;
                case '*':
                    stack.push(operand1 * operand2);
                    break;
                case '/':
                    stack.push(operand1 / operand2);
```

```
                break;
            case '^':
                stack.push(pow(operand1, operand2));
                break;
            default:
                cerr << "Error: Invalid operator." << endl;
                return 0;
        }
      }
   }

   return stack.pop();
}

int evaluatePrefix(string prefix) {
   reverse(prefix.begin(), prefix.end());  // Include the algorithm header for reverse
   Stack stack;

   for (char ch : prefix) {
      if (isOperand(ch))
         stack.push(ch - '0');
      else {
         int operand1 = stack.pop();
         int operand2 = stack.pop();

         switch (ch) {
            case '+':
               stack.push(operand1 + operand2);
               break;
            case '-':
               stack.push(operand1 - operand2);
```

```
                break;
            case '*':
                stack.push(operand1 * operand2);
                break;
            case '/':
                stack.push(operand1 / operand2);
                break;
            case '^':
                stack.push(pow(operand1, operand2));
                break;
            default:
                cerr << "Error: Invalid operator." << endl;
                return 0;
        }
      }
    }

    return stack.pop();
}

int main() {
    string infixExpression = "2+3*4";

    cout << "Infix Expression: " << infixExpression << endl;

    string postfixExpression = infixToPostfix(infixExpression);
    cout << "Postfix Expression: " << postfixExpression << endl;

    string prefixExpression = infixToPrefix(infixExpression);
    cout << "Prefix Expression: " << prefixExpression << endl;
```

```
    int postfixResult = evaluatePostfix(postfixExpression);
    cout << "Postfix Evaluation Result: " << postfixResult << endl;

    int prefixResult = evaluatePrefix(prefixExpression);
    cout << "Prefix Evaluation Result: " << prefixResult << endl;

    return 0;
}
```

Question 8:- (Construct an Expression Tree from postfix and prefix expression. Perform recursive and non- recursive In-order traversals.):

```
#include <iostream>
#include <stack>

using namespace std;

struct Node {
    char data;
    Node* left;
    Node* right;
    Node(char val) : data(val), left(nullptr), right(nullptr) {}
};

bool isOperator(char ch) {
    return (ch == '+' || ch == '-' || ch == '*' || ch == '/' || ch == '^');
}

Node* constructExpressionTreePostfix(const string& postfix) {
    stack<Node*> st;
    for (char ch : postfix) {
        Node* newNode = new Node(ch);
```

```
        if (isOperator(ch)) {
            newNode->right = st.top();
            st.pop();
            newNode->left = st.top();
            st.pop();
        }
        st.push(newNode);
    }
    return st.top();
}

Node* constructExpressionTreePrefix(const string& prefix) {
    stack<Node*> st;
    for (int i = prefix.size() - 1; i >= 0; --i) {
        Node* newNode = new Node(prefix[i]);
        if (isOperator(prefix[i])) {
            newNode->left = st.top();
            st.pop();
            newNode->right = st.top();
            st.pop();
        }
        st.push(newNode);
    }
    return st.top();
}

void recursiveInOrderTraversal(Node* root) {
    if (root) {
        recursiveInOrderTraversal(root->left);
        cout << root->data << " ";
        recursiveInOrderTraversal(root->right);
```

```
    }
}

void nonRecursiveInOrderTraversal(Node* root) {
    stack<Node*> st;
    Node* current = root;
    while (current || !st.empty()) {
        while (current) {
            st.push(current);
            current = current->left;
        }
        current = st.top();
        st.pop();
        cout << current->data << " ";
        current = current->right;
    }
}

int main() {
    string postfixExpression = "23*5+";
    string prefixExpression = "+*235";

    Node* postfixRoot = constructExpressionTreePostfix(postfixExpression);
    Node* prefixRoot = constructExpressionTreePrefix(prefixExpression);

    cout << "Recursive In-order Traversal (Postfix): ";
    recursiveInOrderTraversal(postfixRoot);
    cout << endl;

    cout << "Non-recursive In-order Traversal (Postfix): ";
    nonRecursiveInOrderTraversal(postfixRoot);
```

```
    cout << endl;

    cout << "Recursive In-order Traversal (Prefix): ";
    recursiveInOrderTraversal(prefixRoot);
    cout << endl;

    cout << "Non-recursive In-order Traversal (Prefix): ";
    nonRecursiveInOrderTraversal(prefixRoot);
    cout << endl;

    return 0;
}
```

Question 9:-(Construct an Expression Tree from postfix and prefix expression. Perform recursive and non- recursive pre-order traversals.):

```
#include <iostream>
#include <stack>

using namespace std;

struct Node {
    char data;
    Node* left;
    Node* right;
    Node(char val) : data(val), left(nullptr), right(nullptr) {}
};

bool isOperator(char ch) {
    return (ch == '+' || ch == '-' || ch == '*' || ch == '/' || ch == '^');
}
```

```
Node* constructExpressionTreePostfix(const string& postfix) {
    stack<Node*> st;
    for (char ch : postfix) {
        Node* newNode = new Node(ch);
        if (isOperator(ch)) {
            newNode->right = st.top();
            st.pop();
            newNode->left = st.top();
            st.pop();
        }
        st.push(newNode);
    }
    return st.top();
}

Node* constructExpressionTreePrefix(const string& prefix) {
    stack<Node*> st;
    for (int i = prefix.size() - 1; i >= 0; --i) {
        Node* newNode = new Node(prefix[i]);
        if (isOperator(prefix[i])) {
            newNode->left = st.top();
            st.pop();
            newNode->right = st.top();
            st.pop();
        }
        st.push(newNode);
    }
    return st.top();
}

void recursivePreOrderTraversal(Node* root) {
```

```
    if (root) {
        cout << root->data << " ";
        recursivePreOrderTraversal(root->left);
        recursivePreOrderTraversal(root->right);
    }
}

void nonRecursivePreOrderTraversal(Node* root) {
    stack<Node*> st;
    if (root)
        st.push(root);
    while (!st.empty()) {
        Node* current = st.top();
        st.pop();
        cout << current->data << " ";
        if (current->right)
            st.push(current->right);
        if (current->left)
            st.push(current->left);
    }
}

int main() {
    string postfixExpression = "23*5+";
    string prefixExpression = "+*235";

    Node* postfixRoot = constructExpressionTreePostfix(postfixExpression);
    Node* prefixRoot = constructExpressionTreePrefix(prefixExpression);

    cout << "Recursive Pre-order Traversal (Postfix): ";
    recursivePreOrderTraversal(postfixRoot);
```

```
    cout << endl;

    cout << "Non-recursive Pre-order Traversal (Postfix): ";
    nonRecursivePreOrderTraversal(postfixRoot);
    cout << endl;

    cout << "Recursive Pre-order Traversal (Prefix): ";
    recursivePreOrderTraversal(prefixRoot);
    cout << endl;

    cout << "Non-recursive Pre-order Traversal (Prefix): ";
    nonRecursivePreOrderTraversal(prefixRoot);
    cout << endl;

    return 0;
}
```

Question 10:-(Construct an Expression Tree from postfix and prefix expression. Perform recursive and non- recursive post-order traversals.):

```
#include <iostream>
#include <stack>

using namespace std;

struct Node {
    char data;
    Node* left;
    Node* right;
    Node(char val) : data(val), left(nullptr), right(nullptr) {}
};
```

```
bool isOperator(char ch) {
    return (ch == '+' || ch == '-' || ch == '*' || ch == '/' || ch == '^');
}

Node* constructExpressionTreePostfix(const string& postfix) {
    stack<Node*> st;
    for (char ch : postfix) {
        Node* newNode = new Node(ch);
        if (isOperator(ch)) {
            newNode->right = st.top();
            st.pop();
            newNode->left = st.top();
            st.pop();
        }
        st.push(newNode);
    }
    return st.top();
}

Node* constructExpressionTreePrefix(const string& prefix) {
    stack<Node*> st;
    for (int i = prefix.size() - 1; i >= 0; --i) {
        Node* newNode = new Node(prefix[i]);
        if (isOperator(prefix[i])) {
            newNode->left = st.top();
            st.pop();
            newNode->right = st.top();
            st.pop();
        }
        st.push(newNode);
    }
```

```
    return st.top();
}

void recursivePostOrderTraversal(Node* root) {
    if (root) {
        recursivePostOrderTraversal(root->left);
        recursivePostOrderTraversal(root->right);
        cout << root->data << " ";
    }
}

void nonRecursivePostOrderTraversal(Node* root) {
    stack<Node*> st;
    stack<char> output;
    st.push(root);

    while (!st.empty()) {
        Node* current = st.top();
        st.pop();
        output.push(current->data);

        if (current->left)
            st.push(current->left);
        if (current->right)
            st.push(current->right);
    }

    while (!output.empty()) {
        cout << output.top() << " ";
        output.pop();
    }
```

```
}

int main() {
    string postfixExpression = "23*5+";
    string prefixExpression = "+*235";

    Node* postfixRoot = constructExpressionTreePostfix(postfixExpression);
    Node* prefixRoot = constructExpressionTreePrefix(prefixExpression);

    cout << "Recursive Post-order Traversal (Postfix): ";
    recursivePostOrderTraversal(postfixRoot);
    cout << endl;

    cout << "Non-recursive Post-order Traversal (Postfix): ";
    nonRecursivePostOrderTraversal(postfixRoot);
    cout << endl;

    cout << "Recursive Post-order Traversal (Prefix): ";
    recursivePostOrderTraversal(prefixRoot);
    cout << endl;

    cout << "Non-recursive Post-order Traversal (Prefix): ";
    nonRecursivePostOrderTraversal(prefixRoot);
    cout << endl;

    return 0;
}
```