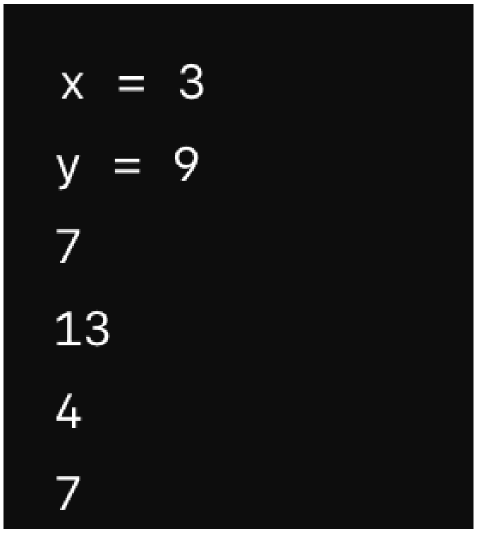
15B17CI371 – Data Structures Lab

ODD 2024
Week 2-LAB A
Practice Lab

1.



```
x = 3
y = 9
7
13
4
7
```

Dry Run

1. Initial State

- `stack` is empty.
- `x = 4`
- `y = 0`

2. Operations

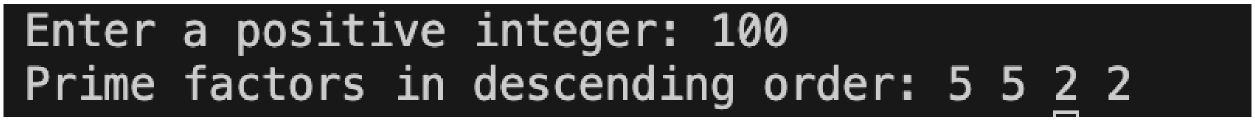
1. `stack.push(7);`
   - Stack: `[7]`
2. `stack.push(x);` (x is 4)
   - Stack: `[7, 4]`
3. `stack.push(x + 5);` (`x + 5` is `4 + 5 = 9`)
   - Stack: `[7, 4, 9]`
4. `y = stack.top();`

- y is assigned the top value of the stack, which is 9.
- Stack: `[7, 4, 9]`
- `y = 9`

5. `stack.pop();`
   - Removes the top value (9).
   - Stack: `[7, 4]`
6. `stack.push(x + y);` (`x + y` is `4 + 9 = 13`)
   - Stack: `[7, 4, 13]`
7. `stack.push(y - 2);` (`y - 2` is `9 - 2 = 7`)
   - Stack: `[7, 4, 13, 7]`
8. `stack.push(3);`
   - Stack: `[7, 4, 13, 7, 3]`
9. `x = stack.top();`
   - x is assigned the top value of the stack, which is 3.
   - Stack: `[7, 4, 13, 7, 3]`
   - `x = 3`
10. `stack.pop();`
    - Removes the top value (3).
    - Stack: `[7, 4, 13, 7]`

2.

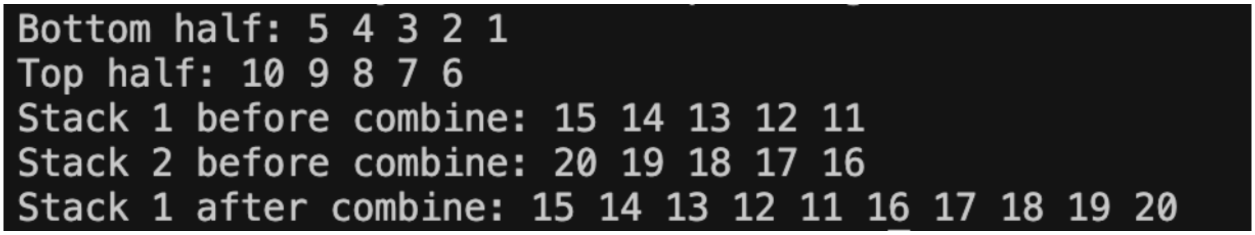
```
#include <iostream>
using namespace std;
const int MAX_SIZE = 100;
struct Stack {
    int data[MAX_SIZE];
    int top;
    Stack() : top(-1) {}
    void push(int value) {
        if (top < MAX_SIZE - 1) {
            data[++top] = value;
        } else {
            cout << "Stack overflow" << endl;
        }
    }
    void pop() {
        if (top >= 0) {
            --top;
        } else {
            cout << "Stack underflow" << endl;
        }
    }
    int peek() const {
        if (top >= 0) {
            return data[top];
        }
        return -1; // Sentinel value indicating empty stack
    }
    bool isEmpty() const {
        return top == -1;
    }
};
void printPrimeFactors(int n) {
    Stack stack;
    while (n % 2 == 0) {
        stack.push(2);
        n /= 2;
    }
    for (int i = 3; i * i <= n; i += 2) {
        while (n % i == 0) {
            stack.push(i);
            n /= i;
        }
    }
    if (n > 2) {
        stack.push(n);
    }
    while (!stack.isEmpty()) {
        cout << stack.peek() << " ";
        stack.pop();
    }
    cout << endl;
}
int main() {
    int number;
    cout << "Enter a positive integer: ";
    cin >> number;
    if (number <= 0) {
        cout << "Please enter a positive integer." << endl;
        return 1;
    }
    cout << "Prime factors in descending order: ";
    printPrimeFactors(number);
    return 0;
}
```



```
Enter a positive integer: 100
Prime factors in descending order: 5 5 2 2
```

3.

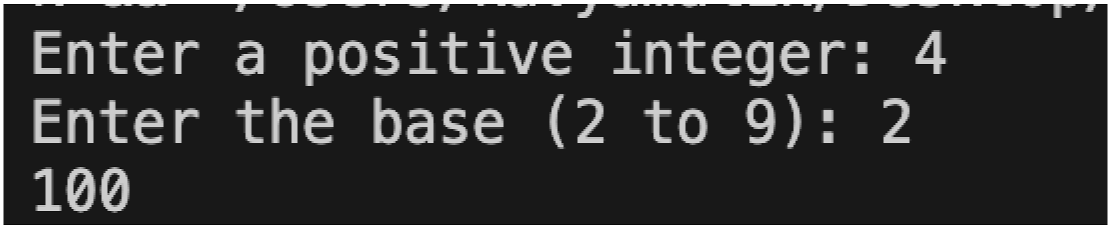
```
#include <iostream>

using namespace std;

const int MAX_SIZE = 100;

struct Stack {
    int data[MAX_SIZE];
    int top;

    Stack() : top(-1) {}

    void push(int value) {
        if (top < MAX_SIZE - 1) {
            data[++top] = value;
        } else {
            cout << "Stack overflow" << endl;
        }
    }

    void pop() {
        if (top >= 0) {
            --top;
        } else {
            cout << "Stack underflow" << endl;
        }
    }

    int peek() const {
        if (top >= 0) {
            return data[top];
        }
        return -1;
    }

    bool isEmpty() const {
        return top == -1;
    }
    int size() const {
        return top + 1;
    }
};

void splitStack(const Stack& original, Stack& bottomHalf, Stack& topHalf) {
    int totalSize = original.size();
    int halfSize = totalSize / 2;
    int count = 0;

    Stack tempStack;
    Stack reversedStack = original;

    while (!reversedStack.isEmpty()) {
        tempStack.push(reversedStack.peek());
        reversedStack.pop();
    }

    while (!tempStack.isEmpty()) {
        if (count < halfSize) {
            bottomHalf.push(tempStack.peek());
        } else {
            topHalf.push(tempStack.peek());
        }
        tempStack.pop();
        count++;
    }
}

void combineStacks(Stack& stack1, Stack& stack2) {
    Stack tempStack;

    while (!stack1.isEmpty()) {
        tempStack.push(stack1.peek());
        stack1.pop();
    }

    while (!stack2.isEmpty()) {
        stack1.push(stack2.peek());
        stack2.pop();
    }

    while (!tempStack.isEmpty()) {
        stack1.push(tempStack.peek());
        tempStack.pop();
    }
}

void printStack(const Stack& stack) {
    Stack tempStack = stack;
    while (!tempStack.isEmpty()) {
        cout << tempStack.peek() << " ";
        tempStack.pop();
    }
    cout << endl;
}

int main() {
    Stack originalStack;
    for (int i = 1; i <= 10; ++i) {
        originalStack.push(i);
    }

    Stack bottomHalf, topHalf;
    splitStack(originalStack, bottomHalf, topHalf);

    cout << "Bottom half: ";
    printStack(bottomHalf);

    cout << "Top half: ";
    printStack(topHalf);

    Stack stack1, stack2;
    for (int i = 11; i <= 15; ++i) {
        stack1.push(i);
    }
    for (int i = 16; i <= 20; ++i) {
        stack2.push(i);
    }

    cout << "Stack 1 before combine: ";
    printStack(stack1);
    cout << "Stack 2 before combine: ";
    printStack(stack2);

    combineStacks(stack1, stack2);
    cout << "Stack 1 after combine: ";
    printStack(stack1);

    return 0;
}
```



```
Bottom half: 5 4 3 2 1
Top half: 10 9 8 7 6
Stack 1 before combine: 15 14 13 12 11
Stack 2 before combine: 20 19 18 17 16
Stack 1 after combine: 15 14 13 12 11 16 17 18 19 20
```

4.

```
#include <iostream>

using namespace std;

const int MAX_SIZE = 100;

struct Stack {
    int data[MAX_SIZE];
    int top;

    Stack() : top(-1) {}

    void push(int value) {
        if (top < MAX_SIZE - 1) {
            data[++top] = value;
        } else {
            cout << "Stack overflow" << endl;
        }
    }

    void pop() {
        if (top >= 0) {
            --top;
        } else {
            cout << "Stack underflow" << endl;
        }
    }
    int peek() const {
        if (top >= 0) {
            return data[top];
        }
        return -1; // Sentinel value indicating empty stack
    }

    bool isEmpty() const {
        return top == -1;
    }
};

void convertToBase(int number, int base) {
    if (base < 2 || base > 9) {
        cout << "Base must be between 2 and 9." << endl;
        return;
    }

    Stack stack;

    // Special case for zero
    if (number == 0) {
        cout << "0" << endl;
        return;
    }

    // Convert number to the specified base
    while (number > 0) {
        stack.push(number % base);
        number /= base;
    }

    // Print the result
    while (!stack.isEmpty()) {
        cout << stack.peek();
        stack.pop();
    }
    cout << endl;
}

int main() {
    int number, base;

    cout << "Enter a positive integer: ";
    cin >> number;

    cout << "Enter the base (2 to 9): ";
    cin >> base;

    if (number < 0) {
        cout << "Please enter a positive integer." << endl;
        return 1;
    }

    convertToBase(number, base);

    return 0;
}
```



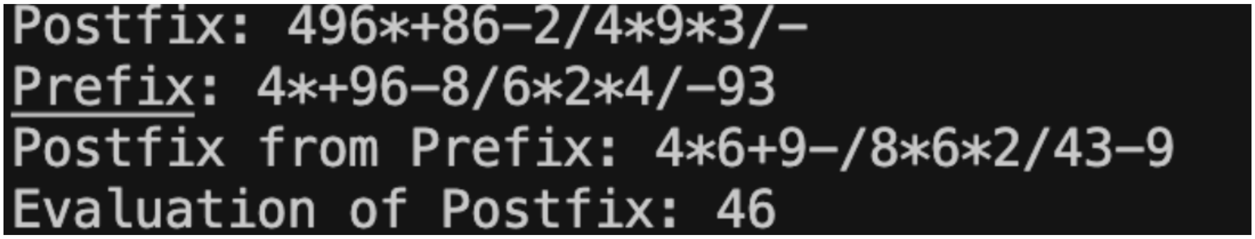
```
Enter a positive integer: 4
Enter the base (2 to 9): 2
100
```

5.

```
#include <iostream>
#include <cstring>
#include <cctype>

using namespace std;

const int MAX_SIZE = 100;

struct Stack {
    char data[MAX_SIZE];
    int top;
    Stack() : top(-1) {}

    void push(char value) {
        if (top < MAX_SIZE - 1) {
            data[++top] = value;
        } else {
            cout << "Stack overflow" << endl;
        }
    }

    char pop() {
        if (top >= 0) {
            return data[top--];
        }
        cout << "Stack underflow" << endl;
        return '\0'; // Return null character to indicate error
    }

    char peek() const {
        if (top >= 0) {
            return data[top];
        }
        return '\0'; // Return null character if stack is empty
    }

    bool isEmpty() const {
        return top == -1;
    }
};

int precedence(char op) {
    switch (op) {
        case '+':
        case '-':
            return 1;
        case '*':
        case '/':
            return 2;
        case '^':
            return 3;
        default:
            return 0;
    }
}
```

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

void infixToPostfix(const char* infix, char* postfix) {
    Stack operators;
    int index = 0;

    for (int i = 0; infix[i] != '\0'; ++i) {
        char c = infix[i];
        if (isdigit(c)) {
            postfix[index++] = c;
        } else if (c == '(') {
            operators.push(c);
        } else if (c == ')') {
            while (!operators.isEmpty() && operators.peek() != '(') {
                postfix[index++] = operators.pop();
            }
            operators.pop(); // Remove '(' from the stack
        } else if (isOperator(c)) {
            while (!operators.isEmpty() && precedence(operators.peek()) >= precedence(c)) {
                postfix[index++] = operators.pop();
            }
            operators.push(c);
        }
    }

    while (!operators.isEmpty()) {
        postfix[index++] = operators.pop();
    }

    postfix[index] = '\0'; // Null-terminate the postfix expression
}

void reverseString(char* str) {
    int length = strlen(str);
    for (int i = 0; i < length / 2; ++i) {
        char temp = str[i];
        str[i] = str[length - i - 1];
        str[length - i - 1] = temp;
    }
}
```

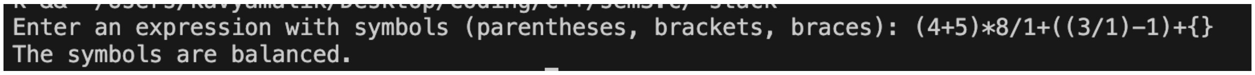
```
void postfixToPrefix(const char* postfix, char* prefix) {
    Stack st;
    int length = strlen(postfix);

    for (int i = 0; i < length; ++i) {
        char c = postfix[i];
        if (isdigit(c)) {
            st.push(c);
        } else if (isOperator(c)) {
            char op1 = st.pop();
            char op2 = st.pop();
            prefix[0] = c;
            prefix[1] = op2;
            prefix[2] = op1;
            prefix[3] = '\0'; // Null-terminate the prefix expression
            for (int j = 0; prefix[j] != '\0'; ++j) {
                st.push(prefix[j]);
            }
        }
    }

    int prefixIndex = 0;
    while (!st.isEmpty()) {
        prefix[prefixIndex++] = st.pop();
    }
    prefix[prefixIndex] = '\0';
    reverseString(prefix); // Reverse the prefix to get correct order
}

void prefixToPostfix(const char* prefix, char* postfix) {
    Stack st;
    int length = strlen(prefix);

    for (int i = length - 1; i >= 0; --i) {
        char c = prefix[i];
        if (isdigit(c)) {
            st.push(c);
        } else if (isOperator(c)) {
            char op1 = st.pop();
            char op2 = st.pop();
            postfix[0] = op1;
            postfix[1] = op2;
            postfix[2] = c;
            postfix[3] = '\0'; // Null-terminate the postfix expression
            for (int j = 0; postfix[j] != '\0'; ++j) {
                st.push(postfix[j]);
            }
        }
    }

    int postfixIndex = 0;
    while (!st.isEmpty()) {
        postfix[postfixIndex++] = st.pop();
    }
    postfix[postfixIndex] = '\0';
}

int evaluatePostfix(const char* postfix) {
    Stack st;
    int length = strlen(postfix);

    for (int i = 0; i < length; ++i) {
        char c = postfix[i];
        if (isdigit(c)) {
            st.push(c - '0');
        } else if (isOperator(c)) {
            int val2 = st.pop();
            int val1 = st.pop();
            switch (c) {
                case '+': st.push(val1 + val2); break;
                case '-': st.push(val1 - val2); break;
                case '*': st.push(val1 * val2); break;
                case '/': st.push(val1 / val2); break;
            }
        }
    }

    return st.pop();
}

int main() {
    char infix[] = "(4+9*6)-((8-6)/2*4)*9/3";
    char postfix[MAX_SIZE];
    char prefix[MAX_SIZE];

    infixToPostfix(infix, postfix);
    cout << "Postfix: " << postfix << endl;
    postfixToPrefix(postfix, prefix);
    cout << "Prefix: " << prefix << endl;

    char postfixFromPrefix[MAX_SIZE];
    prefixToPostfix(prefix, postfixFromPrefix);
    cout << "Postfix from Prefix: " << postfixFromPrefix << endl;

    int result = evaluatePostfix(postfix);
    cout << "Evaluation of Postfix: " << result << endl;

    return 0;
}
```



```
Postfix: 496*+86-2/4*9*3/-
Prefix: 4*+96-8/6*2*4/-93
Postfix from Prefix: 4*6+9-/8*6*2/43-9
Evaluation of Postfix: 46
```

6.

```
#include <iostream>
#include <cstring>

using namespace std;

const int MAX_SIZE = 100;

struct Stack {
    char data[MAX_SIZE];
    int top;

    Stack() : top(-1) {}

    void push(char value) {
        if (top < MAX_SIZE - 1) {
            data[++top] = value;
        } else {
            cout << "Stack overflow" << endl;
        }
    }

    char pop() {
        if (top >= 0) {
            return data[top--];
        }
        cout << "Stack underflow" << endl;
        return '\0'; // Return null character to indicate error
    }

    char peek() const {
        if (top >= 0) {
            return data[top];
        }
        return '\0'; // Return null character if stack is empty
    }

    bool isEmpty() const {
        return top == -1;
    }
};

bool isOpeningSymbol(char c) {
    return (c == '(' || c == '[' || c == '{');
}

bool isClosingSymbol(char c) {
    return (c == ')' || c == ']' || c == '}');
}

bool isMatchingPair(char opening, char closing) {
    return (opening == '(' && closing == ')') ||
           (opening == '[' && closing == ']') ||
           (opening == '{' && closing == '}');
}

bool areSymbolsBalanced(const char* expression) {
    Stack stack;

    for (int i = 0; expression[i] != '\0'; ++i) {
        char c = expression[i];
        if (isOpeningSymbol(c)) {
            stack.push(c);
        } else if (isClosingSymbol(c)) {
            if (stack.isEmpty() || !isMatchingPair(stack.pop(), c)) {
                return false; // Mismatch or unbalanced closing symbol
            }
        }
    }

    return stack.isEmpty(); // If stack is empty, symbols are balanced
}

int main() {
    const int MAX_LENGTH = 100;
    char expression[MAX_LENGTH];

    cout << "Enter an expression with symbols (parentheses, brackets, braces): ";
    cin.getline(expression, MAX_LENGTH);

    if (areSymbolsBalanced(expression)) {
        cout << "The symbols are balanced." << endl;
    } else {
        cout << "The symbols are not balanced." << endl;
    }

    return 0;
}
```

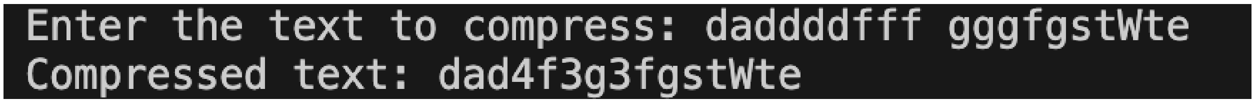


```
Enter an expression with symbols (parentheses, brackets, braces): (4+5)*8/1+((3/1)-1)+{}
The symbols are balanced.
```

7.

```
#include <iostream>
#include <cstring>
using namespace std;
const int MAX_SIZE = 100;
struct Queue {
    char data[MAX_SIZE];
    int front, rear, size;
    Queue() : front(0), rear(0), size(0) {}
    void enqueue(char value) {
        if (size < MAX_SIZE) {
            data[rear] = value;
            rear = (rear + 1) % MAX_SIZE;
            size++;
        } else {
            cout << "Queue overflow" << endl;
        }
    }
    char dequeue() {
        if (size > 0) {
            char value = data[front];
            front = (front + 1) % MAX_SIZE;
            size--;
            return value;
        }
        cout << "Queue underflow" << endl;
        return '\0';
    }
    bool isEmpty() const {
        return size == 0;
    }
    char peek() const {
        if (size > 0) {
            return data[front];
        }
        return '\0';
    }
};
void compressText(const char* input, char* output) {
    Queue queue;
    int index = 0;

    for (int i = 0; input[i] != '\0'; ++i) {
        if (input[i] != ' ') {
            queue.enqueue(input[i]);
        }
    }
    while (!queue.isEmpty()) {
        char currentChar = queue.dequeue();
        int count = 1;
        while (!queue.isEmpty() && queue.peek() == currentChar) {
            queue.dequeue();
            count++;
        }

        output[index++] = currentChar;
        if (count > 1) {
            output[index++] = count + '0';
        }
    }
    output[index] = '\0';
```

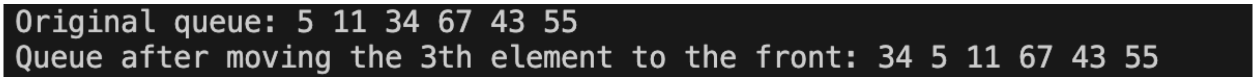
```
}
int main() {
    const int MAX_LENGTH = 100;
    char input[MAX_LENGTH];
    char output[MAX_LENGTH];
    cout << "Enter the text to compress: ";
    cin.getline(input, MAX_LENGTH);
    compressText(input, output);
    cout << "Compressed text: " << output << endl;
    return 0;
}
```



```
Enter the text to compress: daddddfff gggfgstWte
Compressed text: dad4f3g3fgstWte
```

8.

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

void moveNthFront(queue<int>& q, int n) {
    if (q.size() < n || n <= 0) {
        cout << "Invalid value of n." << endl;
        return;
    }

    queue<int> tempQueue;

    for (int i = 1; i < n; ++i) {
        tempQueue.push(q.front());
        q.pop();
    }

    int nthElement = q.front();
    q.pop();

    while (!q.empty()) {
        tempQueue.push(q.front());
        q.pop();
    }

    q.push(nthElement);
    while (!tempQueue.empty()) {
        q.push(tempQueue.front());
        tempQueue.pop();
    }
}

int main() {
    queue<int> q;
    q.push(5);
    q.push(11);
    q.push(34);
    q.push(67);
    q.push(43);
    q.push(55);

    int n = 3;

    cout << "Original queue: ";
    queue<int> temp = q;
    while (!temp.empty()) {
        cout << temp.front() << " ";
        temp.pop();
    }
    cout << endl;

    moveNthFront(q, n);

    cout << "Queue after moving the " << n << "th element to the front: ";
    while (!q.empty()) {
        cout << q.front() << " ";
        q.pop();
    }
    cout << endl;

    return 0;
}
```

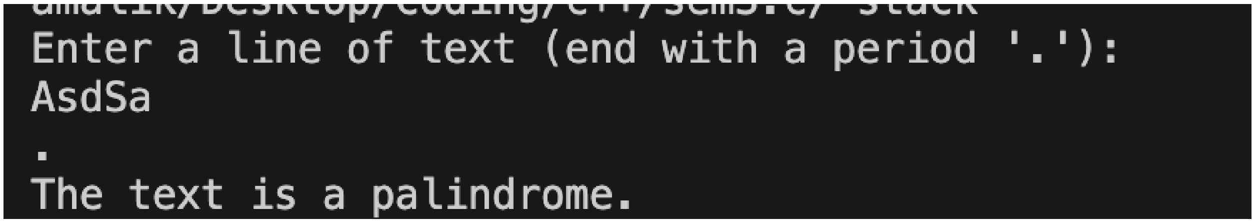


```
Original queue: 5 11 34 67 43 55
Queue after moving the 3th element to the front: 34 5 11 67 43 55
```

9.

```
#include <iostream>
```

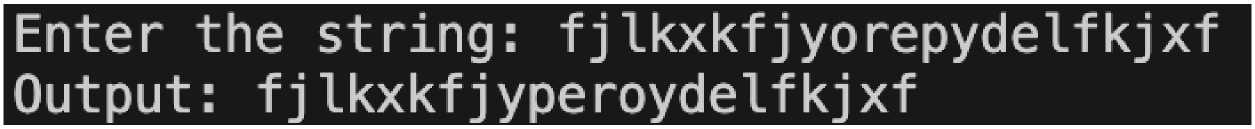
```
#include <queue>
#include <stack>
#include <cctype>

using namespace std;

bool isPalindrome(queue<char>& q, stack<char>& s) {
    while (!q.empty() && !s.empty()) {
        if (q.front() != s.top()) {
            return false;
        }
        q.pop();
        s.pop();
    }
    return q.empty() && s.empty();
}

int main() {
    queue<char> q;
    stack<char> s;

    cout << "Enter a line of text (end with a period '.'): " << endl;

    char c;
    while (cin.get(c)) {
        if (c == '.') {
            break;  // End input when a period is encountered
        }
        if (isalpha(c)) {
            char lowerChar = tolower(c);
            q.push(lowerChar);
            s.push(lowerChar);
        }
    }

    if (isPalindrome(q, s)) {
        cout << "The text is a palindrome." << endl;
    } else {
        cout << "The text is not a palindrome." << endl;
    }

    return 0;
}
```



```
amalik/Desktop/coding/c++/sem3.c/"stack
Enter a line of text (end with a period '.'):
AsdSa
.
The text is a palindrome.
```

10.

```
#include <iostream>
#include <string>
#include <algorithm>
std::string reverse_between_substrings(const std::string& s) {
    size_t start_S1 = s.find('x');
    if (start_S1 == std::string::npos) return "Invalid Input";
    size_t end_S1 = s.find('y', start_S1);
    if (end_S1 == std::string::npos) return "Invalid Input";
    size_t start_S2 = s.find('y', end_S1 + 1);
    if (start_S2 == std::string::npos) return "Invalid Input";
    size_t end_S2 = s.find('x', start_S2 + 1);
    if (end_S2 == std::string::npos) return "Invalid Input";
    if (end_S1 >= start_S2) return "Invalid Input";
    std::string content_between = s.substr(end_S1 + 1, start_S2 - end_S1 - 1);
    std::reverse(content_between.begin(), content_between.end());
    std::string result = s.substr(0, end_S1 + 1) + content_between + s.substr(start_S2);
    return result;
}
int main() {
    std::string input_string;
    std::cout << "Enter the string: ";
    std::getline(std::cin, input_string);
    std::string output_string = reverse_between_substrings(input_string);
    std::cout << "Output: " << output_string << std::endl;
    return 0;
}
```



```
Enter the string: fjlkxkfjyorepydelfkjxf
Output: fjlkxkfjyperoydelfkjxf
```