Implementation of a client server based prefix to postfix, postfix to prefix, prefix to infix, infix to prefix, postfix to infix, and infix to postfix conversion application using TCP

Submitted by :

Name	Roll number
Prashant Kumar Mishra	2447021
Rishi Kumar	2447031
Siddharth Porwal	2447054

Submitted to :- Dr. Amit Kumar Singh

Course Code :- MC470103

Course Title :- Computer Networks

Batch :- MCA (AI & IoT)

Department of computer science & engineering

National Institute of Technology Patna

Ashok Rajpath, Mahendru, Patna, Bihar 800005

Table of contents

S.No.	Contents	Page No
1.	**Introduction**	**4**
	1.1. Transmission control protocol	4
	1.2. Client Server architecture	6
	1.3. Infix Expression	7
	1.4. Prefix Expression	7
	1.5. Postfix Expression	8
	1.6. Interconversion	9
2.	**System Requirements**	**12**
3.	**Flowchart/ Data Flow Diagram**	**13**
4.	**Code**	**15**
	4.1. Client code	15
	4.2. Server code	19
5.	**Output**	**25**
	5.1. Client output	25
	5.2. Server output	28
6.	**Limitations**	**29**
7.	**Observation**	**30**
8.	**Conclusion**	**31**
9.	**Learning outcome**	**32**
10.	**References**	**33**

List Of Figures

S.N.	Figure	Page No
1	Fig 1 - stream delivery with sending and receiving buffers in TCP	5
2	Fig 2 - TCP segment format	5
3	Fig 3 – Client Server Architecture using TCP	6
4	Fig 4 – Socket Address	6
5	Fig 5 – Server Flowchart	13
6	Fig 6 – Client Flowchart	14
7	Fig 7 – Client 1 output	25
8	Fig 8 – Client 2 output	26
9	Fig 9 – Client 3 output	27
10	Fig 10 – Server output	28

List Of Tables

S.N.	Table	Page No
1	Table 1 - well-known ports used by TCP	4

Aim Of The Experiment

implementation of a client server based prefix to postfix , postfix to prefix, prefix to infix, infix to prefix, postfix to infix, and infix to postfix conversion application using Tcp.

1. INTRODUCTION

1.1 Transmission Control Protocol (TCP)

TCP is a process-to-process (program-to-program) protocol .It works at transport layer ensuring process – to process delivery. TCP, therefore uses port numbers. TCP is a connection-oriented protocol; it creates a virtual connection between two TCPs to send data. In addition, TCP uses flow and error control mechanisms at the transport level. In brief, TCP is called a connection-oriented, reliable transport protocol. It adds connection-oriented and reliability features to the services of IP.

Services of TCP :-

- Process-to-Process Communication :-
  - TCP provides process-to-process communication using port numbers . Table-1 lists some well-known port numbers used by TCP.

Table 1 :- well-known ports used by TCP

Port	Protocol	Description
7	Echo	Echoes a received datagram back to the sender
9	Discard	Discards any datagram that is received
11	Users	Active users
13	Daytime	Returns the date and the time
17	Quote	Returns a quote of the day
19	Chargen	Returns a string of characters
20	FTP, Data	File Transfer Protocol (data connection)
21	FTP, Control	File Transfer Protocol (control connection)
23	TELNET	Terminal Network
25	SMTP	Simple Mail Transfer Protocol
53	DNS	Domain Name Server
67	BOOTP	Bootstrap Protocol
79	Finger	Finger
80	HTTP	Hypertext Transfer Protocol
111	RPC	Remote Procedure Call

- Stream Delivery Service :-
  - TCP, on the other hand, allows the sending process to deliver data as a stream of bytes and allows the receiving process to obtain data as a stream of bytes. TCP creates an environment in which the two processes seem to be connected

by an imaginary “tube” that carries their data across the Internet. This imaginary environment is depicted in figure-1 .

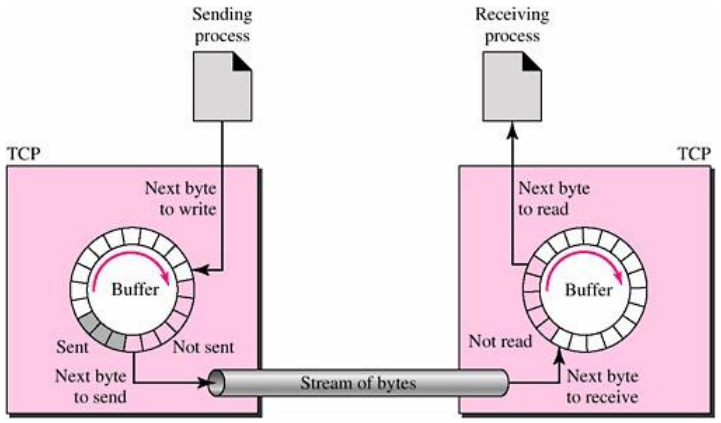

Fig 1 :- stream delivery with sending and receiving buffers in TCP

- Connection-Oriented Service :-
  - TCP is a connection-oriented protocol. When a process at site A wants to send and receive data from another process at site B, the following occurs:
    - The two TCPs establish a connection between them.
    - Data are exchanged in both directions.
    - The connection is terminated.

TCP Segment Format :- The segment consists of a 20- to 60-byte header, followed by data from the application program. The header is 20 bytes if there are no options and up to 60 bytes if it contains options.

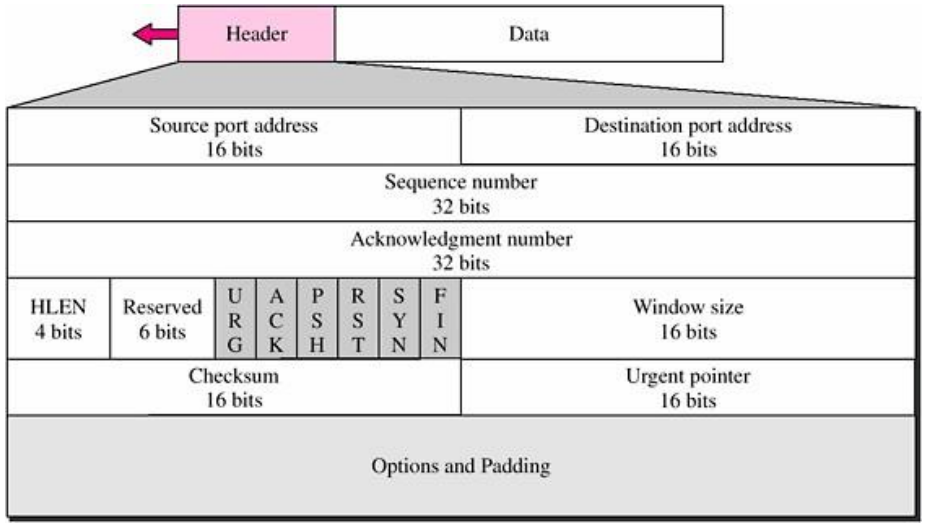

Fig 2 :- TCP segment format

1.2 Client Server architecture using TCP

In the client-server architecture, when the client computer sends a request for data to the server through the internet, the server accepts the requested process and deliver the data packets requested back to the client. CLIENT: takes the input and sends request to the servers. SERVER: receives and processes requests from clients. It is a computing model in which the server hosts, delivers and manages most of the resources and services to be consumed by the client. This type of architecture has one or more client computers connected to a central server over a network or internet connection. In fig 3 , we can se a client server architecture based on TCP.

Socket Addresses :- Process-to-process delivery needs two identifiers, IP address and the port number, at each end to make a connection. The combination of an IP address and a port number is called a socket address. The client socket address defines the client process uniquely just as the server socket address defines the server process uniquely.

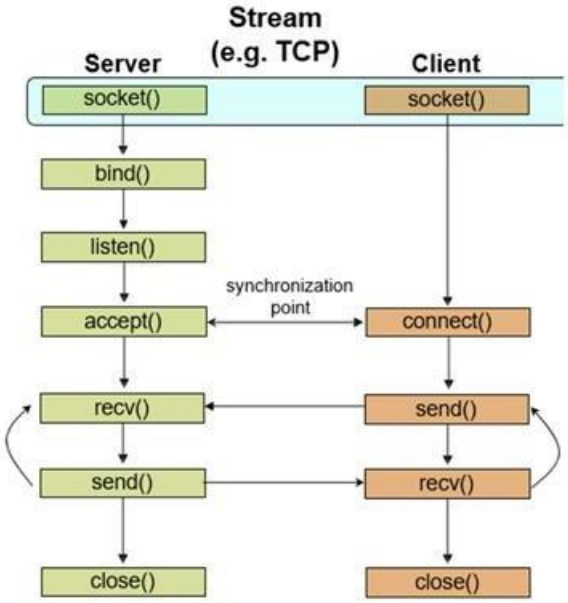

Fig 3 :- client server architecture using TCP

Fig 4 :- Socket address

1.3 Infix Expression

Infix expressions are mathematical expressions where the operator is placed between its operands. This is the most common mathematical notation used by humans. For example, the expression "2 + 3" is an infix expression, where the operator "+" is placed between the operands "2" and "3".

Infix notation is easy to read and understand for humans, but it can be difficult for computers to evaluate efficiently. This is because the order of operations must be taken into account, and parentheses can be used to override the default order of operations.

Infix expressions follow operator precedence rules, which determine the order in which operators are evaluated. For example, multiplication and division have higher precedence than addition and subtraction. This means that in the expression "2 + 3 * 4", the multiplication operation will be performed before the addition operation.

Approach to Validate Infix Expression

- Initialize Counters:
    - Use a counter for parentheses to ensure they are balanced.
    - Keep track of the last character to determine if the current character is valid.
- Iterate Through the String:
    - For each character in the string:
        - If it is an operand (e.g., a letter or number), update the last character.
        - If it is an operator (e.g., +, -, *, /):
            - Ensure the last character was an operand (or a closing parenthesis).
            - Update the last character to the operator.
        - If it is an opening parenthesis (, increment the parentheses counter.
        - If it is a closing parenthesis), decrement the parentheses counter and ensure it does not go negative.
- Final Checks:
    - After iterating, ensure the parentheses counter is zero (indicating balanced parentheses).
    - Ensure the last character is not an operator or an opening parenthesis.

1.4 Prefix Expression

Prefix expressions are also known as Polish notation, are a mathematical notation where the operator precedes its operands. This differs from the more common infix notation, where the operator is placed between its operands.

In prefix notation, the operator is written first, followed by its operands. For example, the infix expression "a + b" would be written as "+ a b" in prefix notation.

Evaluating prefix expressions can be useful in certain scenarios, such as when dealing with expressions that have a large number of nested parentheses or when using a stack-based programming language.

Rules for Valid Prefix Expressions

i. Operators must precede their operands: For example, + a b is valid, but a b + is not.
ii. Operators must have the correct number of operands: For binary operators, there should be exactly two operands for each operator.
iii. The expression should not start or end with an operand: It should start with an operator.

Approach to Validate Prefix Expression

i. Use a stack: Start from the right end of the expression and process each token.
ii. Count operands: For every operator encountered, check if there are enough operands in the stack.
iii. Push operands onto the stack: When you encounter an operand, push it onto the stack.
iv. Check the final state of the stack: At the end of the process, there should be exactly one element left in the stack, which represents the final result of the expression.

1.5 Postfix Expression

Postfix expressions are also known as Reverse Polish Notation (RPN), are a mathematical notation where the operator follows its operands. This differs from the more common infix notation, where the operator is placed between its operands.

In postfix notation, operands are written first, followed by the operator. For example, the infix expression "5 + 2" would be written as "5 2 +" in postfix notation.

Evaluating postfix expressions can be useful in certain scenarios, such as when dealing with expressions that have a large number of nested parentheses or when using a stack-based programming language.

Rules for Valid Postfix Expressions

i. Operands must precede their operators: For example, a b + is valid, but + a b is not.

ii. Operators must have the correct number of operands: For binary operators, there should be exactly two operands for each operator.

iii. The expression should not start or end with an operator: It should start with an operand.

Approach to Validate Postfix Expression

i. Use a stack: Start from the left end of the expression and process each token.

ii. Count operands: For every operator encountered, check if there are enough operands in the stack.

iii. Push operands onto the stack: When you encounter an operand, push it onto the stack.

iv. Check the final state of the stack: At the end of the process, there should be exactly one element left in the stack, which represents the final result of the expression.

1.6 Interconversion

Steps to Convert Prefix to Postfix

i. Initialize an empty stack.

ii. Read the prefix expression from right to left.

iii. If the token is an operand, push it onto the stack.

iv. If the token is an operator:

- Pop the top two elements from the stack (these will be the operands).
- Create a new postfix expression by combining the two operands and the operator in the format operand1 operand2 operator.
- Push the resulting expression back onto the stack.

v. At the end, the stack will contain one element, which is the postfix expression.

Steps to Convert Postfix to Prefix

i. Initialize an empty stack.

ii. Read the postfix expression from left to right.

iii. If the token is an operand, push it onto the stack.

iv. If the token is an operator:

- Pop the top two elements from the stack (these will be the operands).
- Create a new prefix expression by combining the operator and the two operands in the format operator operand1 operand2.
- Push the resulting expression back onto the stack.

v. At the end, the stack will contain one element, which is the prefix expression.

Steps to Convert Postfix to Infix

i. Initialize an empty stack.

ii. Read the postfix expression from left to right.

iii. If the token is an operand, push it onto the stack.

iv. If the token is an operator:

- Pop the top two elements from the stack (these will be the operands).
- Create a new infix expression by combining the two operands and the operator in the format (operand1 operator operand2).
- Push the resulting expression back onto the stack.

v. At the end, the stack will contain one element, which is the infix expression.

Steps to Convert Infix to Postfix

i. Initialize an empty stack.

ii. Read the infix expression from left to right.

iii. If the token is an operand, push it onto the stack.

iv. If the token is an operator:

- Check the precedence of the operator:
  - If the operator has higher precedence than the top operator on the stack, push it onto the stack.
  - If the operator has lower or equal precedence than the top operator on the stack, pop operators from the stack and append them to the output until an operator with lower precedence is found or the stack is empty. Then, push the current operator onto the stack.

v. If the token is a left parenthesis, push it onto the stack.

vi. If the token is a right parenthesis:

- Pop operators from the stack and append them to the output until a left parenthesis is found.
- Remove the left parenthesis from the stack.

vii. At the end, pop any remaining operators from the stack and append them to the output.

viii. The final output is the postfix expression

Steps to Convert Prefix to Infix

i. Initialize an empty stack.

ii. Read the prefix expression from right to left.

iii. If the token is an operand, push it onto the stack.

iv. If the token is an operator:

- Pop the top two elements from the stack (these will be the operands).
- Create a new infix expression by combining the operator and the two operands in the format (operand1 operator operand2).

- Push the resulting expression back onto the stack.

v. At the end, the stack will contain one element, which is the infix expression.

Steps to Convert Infix to Prefix

i. Reverse the infix expression: This helps in processing the expression from the right.

ii. Replace parentheses: Change (to) and vice versa.

iii. Convert the modified infix expression to postfix: This is done using the Shunting Yard algorithm or a similar approach.

iv. Reverse the postfix expression: The result will be the prefix expression.

2. SYSTEM REQUIREMENTS

Hardware Requirements:

- 2 GHz x86 processor
- 2 GB of system memory (RAM)
- 10 GB of hard-drive space
- Monitor to display output
- Keyboard/Mouse for data input

Software Requirements:

- Windows 10 or 11, macOS or a Linux
- python3 Ide
- MS Word(Documentation)

3. FLOWCHART / DATA FLOW DIAGRAM

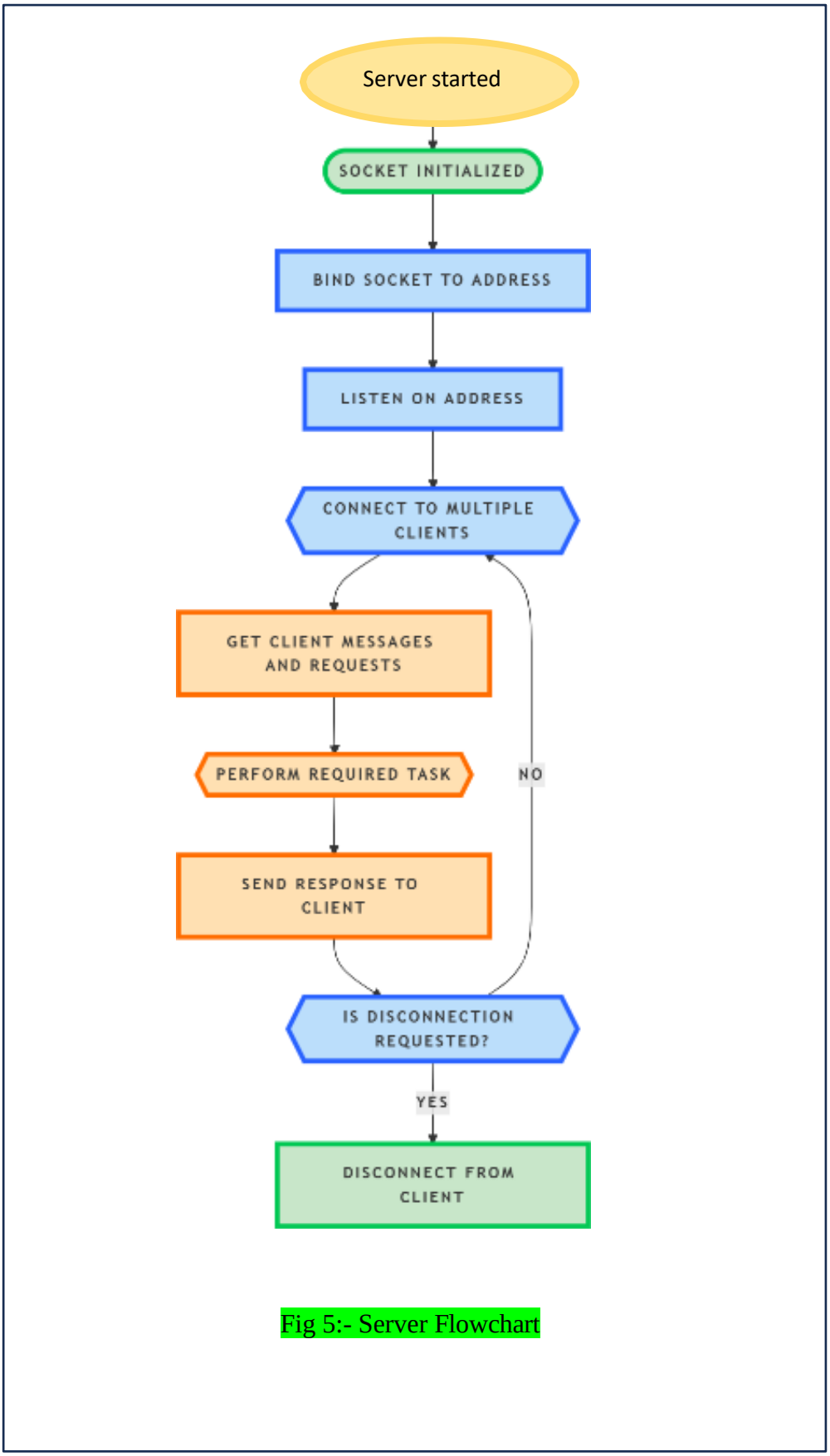

Fig 5:- Server Flowchart

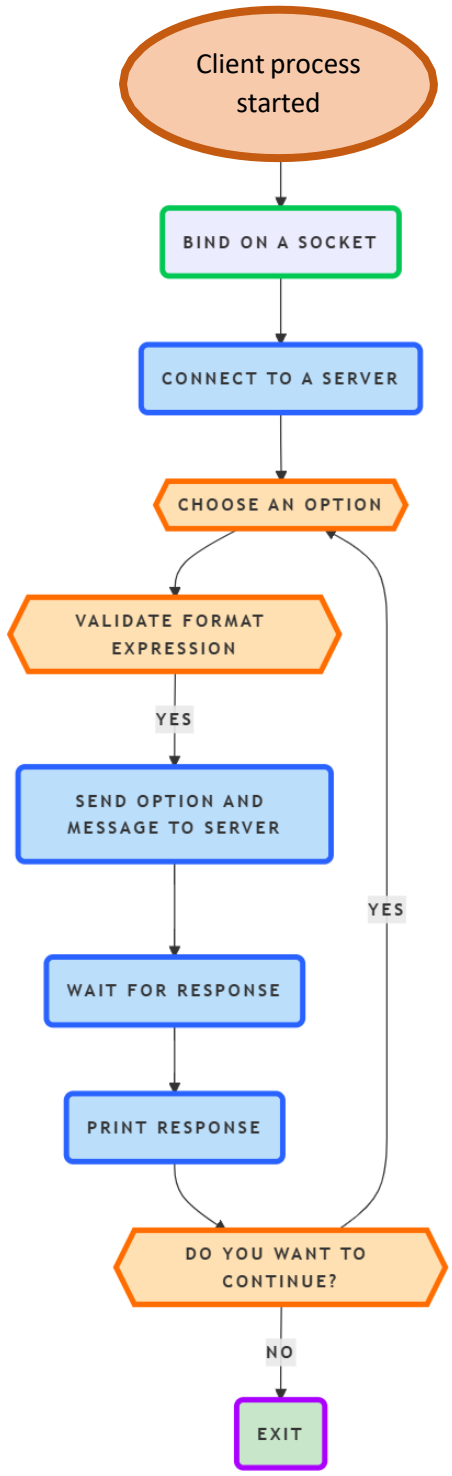

Fig 6 :- Client Flowchart

4. CODE

4.1 Client Code

Code to connect to server

```
import socket

HEADER = 2048
FORMAT = 'utf-8'
DISCONNECT_MESSAGE = "!DISCONNECT"

PORT = 8000
SERVER = "172.20.10.3"  # The IP address of the server
ADDR = (SERVER, PORT)

client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect(ADDR)
```

Code to validate entered infix string at client side

```
# Functions to validate infix expression
def is_valid_infix(expression):
    operators = set('+-*/')
    parentheses_count = 0
    last_char = ''

    for char in expression.split(' '):
        if char.isalnum():  # Operand
            last_char = char
        elif char in operators:  # Operator
            if last_char in operators or last_char == '':
                return False
            last_char = char
        elif char == '(':  # Opening parenthesis
            parentheses_count += 1
            last_char = char
        elif char == ')':  # Closing parenthesis
            parentheses_count -= 1
            if parentheses_count < 0 or last_char in operators:
                return False
            last_char = char
        else:
            return False

    return parentheses_count == 0 and last_char not in operators and last_char != '('
```

code to validate entered postfix string at client side

```
# Functions to validate postfix expression

def is_valid_postfix(expression):
    tokens = expression.split(' ')
    stack = []

    for token in tokens:
        if token.isalnum():  # Operand
            stack.append(token)
        elif token in '+-*/':  # Operator
            if len(stack) < 2:
                return False
            operand2 = stack.pop()
            operand1 = stack.pop()
            stack.append('result')  # Placeholder for result
        else:
            return False

    return len(stack) == 1
```

code to validate entered prefix string at client side

```
# Functions to validate prefix expression
def is_valid_prefix(expression):
    tokens = expression.split(' ')
    stack = []

    for token in reversed(tokens):
        if token.isalnum():  # Operand
            stack.append(token)
        elif token in '+-*/':  # Operator
            if len(stack) < 2:
                return False
            operand1 = stack.pop()
            operand2 = stack.pop()
            stack.append('result')  # Placeholder for result
        else:
            return False

    return len(stack) == 1
```

code to send messages and options to the server

```
# Function to send messages and options to the server
def send(msg, op):
    # Send the operation code (option)
    option = op.encode(FORMAT)
    op_length = len(option)
    send_oplen = str(op_length).encode(FORMAT)
    send_oplen += b' ' * (HEADER - len(send_oplen))  # Padding for consistent length
    client.send(send_oplen)  # Send the length of the option
    client.send(option)  # Send the actual option

    # Send the message (expression)
    message = msg.encode(FORMAT)
    msg_length = len(message)
    send_length = str(msg_length).encode(FORMAT)
    send_length += b' ' * (HEADER - len(send_length))  # Padding for consistent length
    client.send(send_length)  # Send the length of the message
    client.send(message)  # Send the actual message

    # Receive the output from the server
    output_length = int(client.recv(HEADER).decode(FORMAT))  # Receive the output
length
    output = client.recv(output_length).decode(FORMAT)  # Receive the actual output
    print(output)
```

code of Main loop for user input

```
# Main loop for user input
ch = True
while ch:
    print("\n1: Prefix to Postfix conversion\n2: Postfix to Prefix conversion")
    print("3: Prefix to Infix conversion\n4: Infix to Prefix conversion")
    print("5: Postfix to Infix conversion\n6: Infix to Postfix conversion")
    print("7: Exit")

    choice = int(input("\nEnter your choice: "))

    if choice == 1:  # Prefix to Postfix conversion
```

```
        msg = input("\nEnter prefix expression: ")
        if not is_valid_prefix(msg):
            print("\nInvalid Prefix Expression! Please try again.")
            continue
        send(msg, str(choice))

    elif choice == 2:  # Postfix to Prefix conversion
        msg = input("\nEnter postfix expression: ")
        if not is_valid_postfix(msg):
            print("\nInvalid Postfix Expression! Please try again.")
            continue
        send(msg, str(choice))

    elif choice == 3:  # Prefix to Infix conversion
        msg = input("\nEnter prefix expression: ")
        if not is_valid_prefix(msg):
            print("\nInvalid Prefix Expression! Please try again.")
            continue
        send(msg, str(choice))

    elif choice == 4:  # Infix to Prefix conversion
        msg = input("\nEnter infix expression: ")
        if not is_valid_infix(msg):
            print("\nInvalid Infix Expression! Please try again.")
            continue
        send(msg, str(choice))

    elif choice == 5:  # Postfix to Infix conversion
        msg = input("\nEnter postfix expression: ")
        if not is_valid_postfix(msg):
            print("\nInvalid Postfix Expression! Please try again.")
            continue
        send(msg, str(choice))

    elif choice == 6:  # Infix to Postfix conversion
        msg = input("\nEnter infix expression: ")
        if not is_valid_infix(msg):
            print("\nInvalid Infix Expression! Please try again.")
            continue
        send(msg, str(choice))

    elif choice == 7:  # Exit
```

```
        ch = False
        send(DISCONNECT_MESSAGE, str(choice))  # Send disconnect message
        break

    else:
        print("\nWrong choice entered. Please try again.")
```

4.2 Server Code

code to start server

```
import socket
import threading

HEADER = 2048
FORMAT = 'utf-8'
DISCONNECT_MESSAGE = "!DISCONNECT"

SERVER = socket.gethostbyname(socket.gethostname())  # Use the server's local IP
PORT = 8000
ADDR = (SERVER, PORT)

server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind(ADDR)
```

code to find precedence of operators

```
def precedence(op):
    if op in ('+', '-'):
        return 1
    if op in ('*', '/'):
        return 2
    return 0
```

code for conversion from infix to postfix

```
def infix_to_postfix(expression):
    stack = []  # Stack to hold operators
    postfix = []  # List for the output (postfix expression)

    for char in expression.split(' '):  # Use split(' ') explicitly
        if char.isalnum():  # Operand
            postfix.append(char)
        elif char == '(':  # Opening parenthesis
            stack.append(char)
        elif char == ')':  # Closing parenthesis
            while stack and stack[-1] != '(':
                postfix.append(stack.pop())  # Pop from stack to output
            stack.pop()  # Pop the '(' from the stack
        else:  # Operator
            while stack and precedence(stack[-1]) >= precedence(char):
                postfix.append(stack.pop())  # Pop from stack to output
            stack.append(char)

    # Pop all remaining operators
    while stack:
        postfix.append(stack.pop())

    return ' '.join(postfix)  # Join into string
```

code for conversion from infix to prefix

```
def infix_to_prefix(expression):
    # Step 1: Reverse the expression
    expression = expression[::-1]

    # Step 2: Replace parentheses
    expression = ''.join([')' if char == '(' else '(' if char == ')' else char for char in expression])

    # Step 3: Convert to postfix
    postfix = infix_to_postfix(expression)

    # Step 4: Reverse the postfix to get prefix
    return postfix[::-1]
```

code for conversion from prefix to infix

```
def prefix_to_infix(prefix):
    stack = []

    for token in reversed(prefix.split(' ')):  # Use split(' ') explicitly
        if token.isalnum():  # Operand
            stack.append(token)
        else:  # Operator
            operand1 = stack.pop()
            operand2 = stack.pop()
            new_expr = f"({operand1} {token} {operand2})"
            stack.append(new_expr)

    return stack.pop()
```

code for conversion from postfix to infix

```
def postfix_to_infix(postfix):
    stack = []

    for token in postfix.split(' '):  # Use split(' ') explicitly
        if token.isalnum():  # Operand
            stack.append(token)
        else:  # Operator
            operand2 = stack.pop()
            operand1 = stack.pop()
            new_expr = f"({operand1} {token} {operand2})"
            stack.append(new_expr)

    return stack.pop()
```

code for conversion from postfix to prefix

```
def postfix_to_prefix(postfix):
    stack = []

    for token in postfix.split(' '):  # Use split(' ') explicitly
        if token.isalnum():  # Operand
            stack.append(token)
        else:  # Operator
            operand2 = stack.pop()
            operand1 = stack.pop()
```

```
            new_expr = f"{token} {operand1} {operand2}"
            stack.append(new_expr)

    return stack.pop()
```

code for conversion from prefix to postfix

```
def prefix_to_postfix(prefix):
    stack = []

    for token in reversed(prefix.split(' ')):  # Use split(' ') explicitly
        if token.isalnum():  # Operand
            stack.append(token)
        else:  # Operator
            operand1 = stack.pop()
            operand2 = stack.pop()
            new_expr = f"{operand1} {operand2} {token}"
            stack.append(new_expr)

    return stack.pop()
```

code for handling a client

```
def handle_client(conn, addr):
    print(f"\n[ NEW CONNECTION ] {addr} is connected.\n")
    connected = True

    while connected:
        # Receive the option length and option
        option_length = int(conn.recv(HEADER).decode(FORMAT))  # Receive the option
length
        option = int(conn.recv(option_length).decode(FORMAT))  # Receive the actual option

        # Receive the message length and the message
        msg_length = int(conn.recv(HEADER).decode(FORMAT))  # Receive the message
length
        message = str(conn.recv(msg_length).decode(FORMAT))  # Receive the actual message

        if message == DISCONNECT_MESSAGE:
            print(f"\n[ REQUEST ] {addr} sent a request to disconnect\n")
            connected = False
            response= "\ndisconnected from server\n".encode(FORMAT)
```

```
            print(f"\n[ DISCONNECT ] {addr} disconnected from server \n")

        elif option == 1:  # Prefix to Postfix
            print(f"\n[ REQUEST ] {addr} sent a prefix expression to get postfix expression\n")
            output = prefix_to_postfix(message)
            response = f"\nthe postfix expression for given prefix string is:
{output}\n".encode(FORMAT)

        elif option == 2:  # Postfix to Prefix
            print(f"\n[ REQUEST ] {addr} sent a postfix expression to get prefix expression\n")
            output = postfix_to_prefix(message)
            response = f"\nthe prefix expression for given postfix string is:
{output}\n".encode(FORMAT)

        elif option == 3:  # Prefix to Infix
            print(f"\n[ REQUEST ] {addr} sent a prefix expression to get infix expression\n")
            output = prefix_to_infix(message)
            response = f"\nthe infix expression for given prefix string is:
{output}\n".encode(FORMAT)

        elif option == 4:  # Infix to Prefix
            print(f"\n[ REQUEST ] {addr} sent an infix expression to get prefix
expression\n{message}\n")
            output = infix_to_prefix(message)
            response = f"\nthe prefix expression for given infix string is:
{output}\n".encode(FORMAT)

        elif option == 5:  # Postfix to Infix
            print(f"\n[ REQUEST ] {addr} sent a postfix expression to get infix expression\n")
            output = postfix_to_infix(message)
            response = f"\nthe infix expression for given postfix string is:
{output}\n".encode(FORMAT)

        elif option == 6:  # Infix to Postfix
            print(f"\n[ REQUEST ] {addr} sent an infix expression to get postfix
expression\n{message}\n")
            output = infix_to_postfix(message)
            response = f"\nthe postfix expression for given infix string is:
{output}\n".encode(FORMAT)

        # Send the response back to the client
        response_length = len(response)
```

```
    send_length = str(response_length).encode(FORMAT)
    send_length += b' ' * (HEADER - len(send_length))  # Padding for consistent length
    conn.send(send_length)  # Send the length of the response
    conn.send(response)  # Send the actual response
```

code for starting and connecting to a client

```
# Start server and listen for connections
def start():
    server.listen()
    print(f"[LISTENING] Server is listening on {SERVER}:{PORT}\n")

    while True:
        conn, addr = server.accept()
        thread = threading.Thread(target=handle_client, args=(conn, addr))
        thread.start()


print("[STARTING] Server is starting...")
start()
```

5. OUTPUT

5.1 Client output

```
1: Prefix to Postfix conversion
2: Postfix to Prefix conversion
3: Prefix to Infix conversion
4: Infix to Prefix conversion
5: Postfix to Infix conversion
6: Infix to Postfix conversion
7: Exit

Enter your choice: 1

Enter prefix expression: + 12 32

the postfix expression for given prefix string is: 12 32 +


1: Prefix to Postfix conversion
2: Postfix to Prefix conversion
3: Prefix to Infix conversion
4: Infix to Prefix conversion
5: Postfix to Infix conversion
6: Infix to Postfix conversion
7: Exit

Enter your choice: 2

Enter postfix expression: 56 75 +

the prefix expression for given postfix string is: + 56 75


1: Prefix to Postfix conversion
2: Postfix to Prefix conversion
3: Prefix to Infix conversion
4: Infix to Prefix conversion
5: Postfix to Infix conversion
6: Infix to Postfix conversion
7: Exit

Enter your choice: 7

disconnected from server
```

Fig 7 :- Client 1 output

```
1: Prefix to Postfix conversion
2: Postfix to Prefix conversion
3: Prefix to Infix conversion
4: Infix to Prefix conversion
5: Postfix to Infix conversion
6: Infix to Postfix conversion
7: Exit

Enter your choice: 3

Enter prefix expression: + 12 21

the infix expression for given prefix string is: (12 + 21)


1: Prefix to Postfix conversion
2: Postfix to Prefix conversion
3: Prefix to Infix conversion
4: Infix to Prefix conversion
5: Postfix to Infix conversion
6: Infix to Postfix conversion
7: Exit

Enter your choice: 4

Enter infix expression: ( 56 + 96 ) + ( 12 + 91 )

the prefix expression for given infix string is: + + 56 96 + 12 91


1: Prefix to Postfix conversion
2: Postfix to Prefix conversion
3: Prefix to Infix conversion
4: Infix to Prefix conversion
5: Postfix to Infix conversion
6: Infix to Postfix conversion
7: Exit

Enter your choice: 7

disconnected from server
```

Fig 8 :- Client 2 output

```
1: Prefix to Postfix conversion
2: Postfix to Prefix conversion
3: Prefix to Infix conversion
4: Infix to Prefix conversion
5: Postfix to Infix conversion
6: Infix to Postfix conversion
7: Exit

Enter your choice: 5

Enter postfix expression: a b + c d - *

the infix expression for given postfix string is: ((a + b) * (c - d))


1: Prefix to Postfix conversion
2: Postfix to Prefix conversion
3: Prefix to Infix conversion
4: Infix to Prefix conversion
5: Postfix to Infix conversion
6: Infix to Postfix conversion
7: Exit

Enter your choice: 6

Enter infix expression: ( ( a / b ) * ( c / d ) )

the postfix expression for given infix string is: a b / c d / *


1: Prefix to Postfix conversion
2: Postfix to Prefix conversion
3: Prefix to Infix conversion
4: Infix to Prefix conversion
5: Postfix to Infix conversion
6: Infix to Postfix conversion
7: Exit

Enter your choice: 7

disconnected from server
```

Fig 9 :- Client 3 output

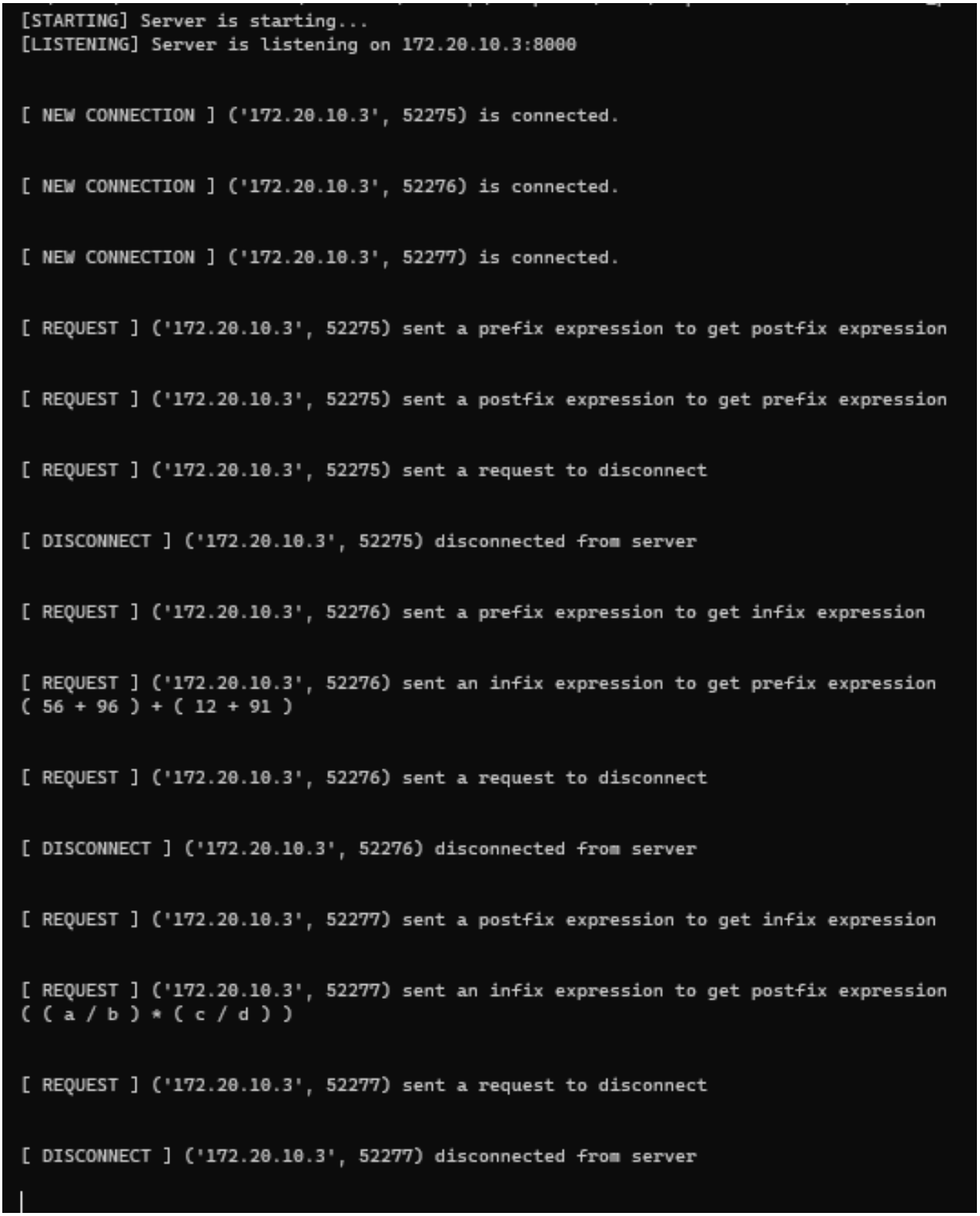

Fig 10 :- Server output

6. Limitations

Limitations of the Implementation

While the client-server application for expression conversion demonstrates the fundamental concepts of TCP communication and expression handling, several limitations must be acknowledged:

1. **Lack of Error Handling**: The implementation lacks comprehensive error handling for socket operations, which could result in crashes or unhandled exceptions during network communication.
2. **No Support for Complex Expressions**: The application may not support complex expressions involving multi-digit numbers, variables, or functions, limiting its usability for advanced mathematical tasks. For example the implementation supports only four operators " **+ - * /** " and only one Parentheses " **( )** " , any other operator or Parentheses will give error.
3. **No User Authentication**: The absence of user authentication or access control exposes the server to unauthorized access or misuse.
4. **Limited User Interface**: The text-based client interface may not be user-friendly for all users, and a graphical user interface (GUI) could enhance usability.
5. **No Logging or Monitoring**: The lack of logging mechanisms makes it difficult to track user interactions, errors, or performance metrics, complicating issue diagnosis.
6. **Hardcoded Server Address**: The client code contains a hardcoded server address, limiting flexibility and requiring code modification for different server connections.
7. **No Input Sanitization**: The implementation does not sanitize user input, which could lead to security vulnerabilities such as injection attacks.
8. **Performance Limitations**: The conversion algorithms may not be optimized for performance, especially for very large expressions, leading to slower response times.

In summary, while the implementation serves as a foundational application for expression conversion, addressing these limitations would enhance its robustness, usability, and security.

7. Observation

7.1 Expression Conversion Techniques

The application successfully implements various expression conversion techniques, including:

- Infix to Postfix: The conversion is performed using the Shunting Yard algorithm, which effectively handles operator precedence and associativity.
- Postfix to Infix: The algorithm correctly reconstructs infix expressions from postfix notation, ensuring that the order of operations is preserved.
- Prefix to Infix: The conversion from prefix to infix is handled efficiently, allowing for accurate expression reconstruction.
- Infix to Prefix: The application successfully converts infix expressions to prefix notation, demonstrating the versatility of the implemented algorithms.
- Postfix to Prefix: The conversion from postfix to prefix is implemented, allowing users to switch between these two notations seamlessly.
- Prefix to Postfix: The application also supports conversion from prefix to postfix, providing a complete set of interconversion functionalities.

7.2 Validation Functions

The application includes validation functions that check the correctness of the input expressions before conversion. These functions ensure that:

- The expressions are well-formed and contain valid characters.
- The operators and operands are used correctly according to the rules of arithmetic.
- Parentheses are balanced in infix expressions, preventing errors during conversion.

While these validation functions are effective for most cases, further enhancements could be made to handle more complex scenarios and edge cases.

7.3 User Experience

The client interface, while functional, is text-based and may not be user-friendly for all users. Users must be familiar with command-line operations to effectively interact with the application. A graphical user interface (GUI) could significantly enhance usability and accessibility.

7.4 Concurrency and Scalability

The server's ability to handle multiple clients simultaneously through threading is a significant advantage. However, the single-threaded processing of expressions could lead to delays in response times when multiple clients send complex requests. Future enhancements could include optimizing the processing logic to improve scalability and responsiveness.

Overall, the project successfully demonstrates the core concepts of TCP communication and expression conversion. This implementation serves as a foundational application.

8. Conclusion

This mini project highlights the significant potential of expression conversion algorithms and the effectiveness of client-server architecture in facilitating communication between users and the server. Throughout the project, I successfully implemented various algorithms for converting between infix, prefix, and postfix expressions, demonstrating a solid understanding of the underlying principles of expression parsing and evaluation.

The client-server model employed in this project effectively demonstrates how TCP can be utilized for reliable communication between a client and a server. This architecture not only supports multiple client connections but also allows for efficient processing of requests, although there are areas for improvement in terms of scalability and performance under heavy loads.

Through this study, We gained valuable insights into the advantages and limitations of different expression conversion techniques. While the implemented algorithms perform well for standard cases, challenges remain in handling complex expressions and ensuring robust error handling. Future enhancements could focus on optimizing performance, improving user experience through a graphical interface, and implementing security measures to protect against potential vulnerabilities.

In conclusion, this project has provided us with a deeper understanding of expression conversion algorithms and the practical application of client-server architecture. The knowledge and skills acquired during this project will be beneficial for future endeavours in network programming.

9. LEARNING OUTCOME

Following are the learning outcomes We achieved through this mini project on "Client-Server-Based Application for Expression Conversion":

- **Familiarity with Expression Conversion Techniques**: We gained a comprehensive understanding of various expression conversion techniques, including infix, prefix, and postfix notations.
- **Programming Skills**: We developed the ability to write efficient algorithms for converting between different expression formats, enhancing our programming skills in the process.
- **Problem-Solving Abilities**: We improved our problem-solving skills by tackling numerical and logical challenges related to expression parsing and evaluation.
- **Classification of Algorithms**: We are now able to classify and differentiate between various expression conversion algorithms based on their functionality and use cases.
- **Identification of Errors**: We learned to identify common errors and pitfalls in expression conversion, which is crucial for debugging and improving algorithm performance.
- **Application of Knowledge**: We can apply my knowledge of expression conversion algorithms in future projects, particularly in areas such as mathematical expression evaluation.
- **Understanding of Client-Server Architecture**: We developed a solid understanding of client-server architecture, which will aid me in designing and implementing networked applications.
- **Security Awareness**: We recognized the importance of implementing security measures in network applications, which will help us build secure and robust software solutions in the future.

10. REFERENCES

[1] Tanenbaum S. Andrew and David J. Wetherall, Computer Networks (Vol. 5), Pearson Education Inc., 2011.

[2] Forouzan A. Behrouz, Data Communications And Networking (Vol.4), McGraw HilI Forouzan networking series, 2007.

[3] Thomas H. Cormen , Charles E. Leiserson , Ronald L. Rivest and Clifford Stein , Introduction to Algorithms(Vol. 4) , The MIT Press , 2022.

[4] Michael T. Goodrich , Roberto Tamassia , David M. Mount , Data Structures and Algorithms in C++ (Vol. 2) , John Wiley & Sons, Inc. , 2011.

[5] Lab 8 manual – Introduction to socket programming – computer networks - **MC470103**

[6] https://www.calcont.in/Conversion/infix_to_postfix

[7] https://www.geeksforgeeks.org/convert-infix-expression-to-postfix-expression/

[8] https://stackoverflow.com/questions/39823568/how-can-i-check-whether-the-given-expression-is-an-infix-expression-postfix-exp

[9] https://www.geeksforgeeks.org/expression-evaluation/

[10] https://www.geeksforgeeks.org/infix-postfix-prefix-notation/

[11] https://medium.com/@serene_mulberry_tiger_125/understanding-expressions-infix-prefix-and-postfix-notations-in-computer-science-and-mathematics-c5390cee01be

[12] https://cs.berea.edu/cppds/BasicDS/InfixPrefixandPostfixExpressions.html

[13] http://www.cs.man.ac.uk/~pjj/cs212/fix.html

[14] https://www.geeksforgeeks.org/what-are-the-minimum-hardware-requirements-for-python-programming/#hardware-specifications

[15] https://nitp.knimbus.com/openFullText.html?DP=https://www-accessengineeringlibrary-com-nitp.knimbus.com:443/browse/data-communications-and-networking-fourth-edition