DESIGN A TURING MACHINE USING JAVA TO IMPLEMENT BASIC OPERATIONS OF TM

OBJECTIVE:

The primary goal of this project is to create a Java-based Turing Machine capable of effectively modeling and executing fundamental Turing Machine operations. By focusing on the core functionality, this project aims to contribute to a better understanding of computation theory and the essential workings of Turing Machines.

ABSTRACT:

A Turing machine is a mathematical model of computation and implementing its basic operations in Java allows for the exploration of fundamental concepts in computer science. This project involves creating a Java program that simulates a Turing machine, which can read, write, and move on an infinite tape. It serves as an educational tool to understand the principles of computation, such as state transitions and tape manipulation. This implementation will enable users to experiment with various Turing machine configurations and gain insights into the underlying computational theory.

INTRODUCTION:

Turing machines are a foundational concept in the field of computer science, introduced by the pioneering mathematician and computer scientist Alan Turing in the 1930s. They provide a theoretical framework for understanding computation and have proven to be an essential building block in the development of modern computers and algorithms. This report delves into the implementation of basic Turing machine operations using the Java programming language. By simulating the behavior of a Turing machine, we can explore the core principles of computation, such as state transitions and tape manipulation.

SOFTWARE REQUIREMENTS:

- Operating System: Windows 11
- Technology: Java
- Code Editor: Visual Studio Code

CONCEPTS/WORKING PRINCIPLE

A Turing Machine (TM) is a mathematical model which consists of an infinite length tape divided into cells on which input is given. It consists of a head which reads the input tape. A state register stores the state of the Turing machine. After reading an input symbol, it is replaced with another symbol, its internal state is changed, and it moves from one cell to the right or left. If the TM reaches the final state, the input string is accepted, otherwise rejected.

A TM can be formally described as a 7-tuple (Q, X, ∑, δ, q0, B, F) where −

- Q is a finite set of states
- X is the tape alphabet
- ∑ is the input alphabet
- δ is a transition function; δ : Q × X → Q × X × {Left_shift, Right_shift}.
- q0 is the initial state
- B is the blank symbol
- F is the set of final states

PROGRAMS:

MAIN.JAVA:

```
package main;
import tm.*; public
class Main
{

public static void main(String[] args)
{
TuringMachine TM1 = MachinesLibrary.EqualBinaryWords();

boolean done = TM1.Run("010000110101#010000110101", false); if
(done==true)
{
System.out.println("The input was accepted.");
} else
{
System.out.println("The input was rejected."); }
}

}
```

MACHINESLIBRARY.JAVA:

```
package tm;
public final class MachinesLibrary
{
        private MachinesLibrary() {}
        public static TuringMachine EqualBinaryWords()
        {
                TuringMachine newTM = new TuringMachine();
newTM.addState("q1");
```

```
                newTM.addState("q2");
newTM.addState("q3"); newTM.addState("q4");
newTM.addState("q5");                   newTM.addState("q6");
        newTM.addState("q7");
newTM.addState("q8");                   newTM.addState("qa");
        newTM.addState("qr");
newTM.setStartState("q1");
newTM.setAcceptState("qa");
                newTM.setRejectState("qr");
                newTM.addTransition("q1", '1', "q3", 'x', true);
        newTM.addTransition("q1", '0', "q2", 'x', true);
newTM.addTransition("q1", '#', "q8", '#', true);
newTM.addTransition("q2", '0', "q2", '0', true);
newTM.addTransition("q2", '1', "q2", '1', true);
newTM.addTransition("q2", '#', "q4", '#', true);
newTM.addTransition("q3", '0', "q3", '0', true);
newTM.addTransition("q3", '1', "q3", '1', true);
newTM.addTransition("q3", '#', "q5", '#', true);
newTM.addTransition("q4", 'x', "q4", 'x', true);
newTM.addTransition("q4", '0', "q6", 'x', false);
newTM.addTransition("q5", 'x', "q5", 'x', true);
newTM.addTransition("q5", '1', "q6", 'x', false);
newTM.addTransition("q6", '0', "q6", '0', false);
newTM.addTransition("q6", '1', "q6", '1', false);
newTM.addTransition("q6", 'x', "q6", 'x', false);
newTM.addTransition("q6", '#', "q7", '#', false);
newTM.addTransition("q7", '0', "q7", '0', false);
newTM.addTransition("q7", '1', "q7", '1', false);
newTM.addTransition("q7", 'x', "q1", 'x', true);
newTM.addTransition("q8", 'x', "q8", 'x', true);
newTM.addTransition("q8", '_', "qa", '_', true);
return newTM;
        }

}
```

TURINGMACHINE.JAVA:

```
package tm;  import
java.util.*; public class
TuringMachine
{ private Set<String> StateSpace; private
        Set<Transition> TransitionSpace;
        private String StartState;
        private String AcceptState;
        private String RejectState;
```

```
        private String Tape; private
        String CurrentState;
        private int CurrentSymbol;

        class Transition
        {
                String readState;
char readSymbol;                String
writeState;
                char writeSymbol;
                boolean moveDirection;        //true is right, false is left

                boolean isConflicting(String state, char symbol)
                {
                        if (state.equals(readState) && symbol == readSymbol)
                        {
                                return true;
                        }
                        else
                        {
                                return false;
                        }
                }
        }


        public TuringMachine()
        {
                StateSpace = new HashSet<String>();
                TransitionSpace = new HashSet<Transition>();
                StartState = new String("");
                AcceptState = new String("");
                RejectState = new String("");
                Tape = new String("");
                CurrentState = new String("");
                CurrentSymbol = 0;

        }

        public boolean Run(String input, boolean silentmode)
        {
                CurrentState = StartState;
Tape = input;
                while(!CurrentState.equals(AcceptState) &&
!CurrentState.equals(RejectState))
                {
                        boolean foundTransition = false;
```

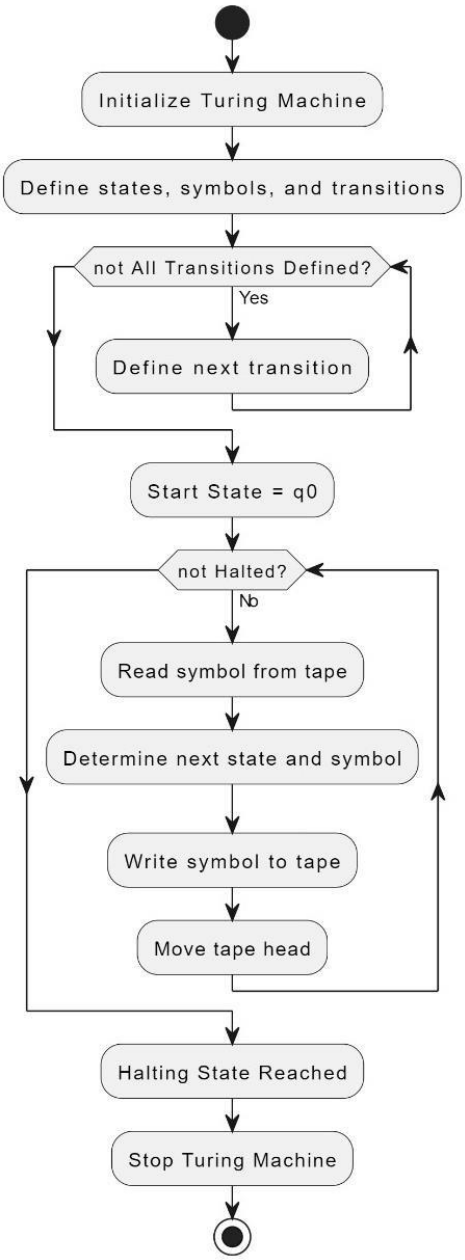
```
                    Transition CurrentTransition = null;
                    if (silentmode == false)
                    {
                            if (CurrentSymbol>0)
                            {
                                    System.out.println(Tape.substring(0, CurrentSymbol)
+ " " + CurrentState + " " + Tape.substring(CurrentSymbol));
                            }
                            else
                            {
                                    System.out.println(" " + CurrentState + " " +
Tape.substring(CurrentSymbol));
                            }
                    }

                    Iterator<Transition> TransitionsIterator = TransitionSpace.iterator();
                while ( TransitionsIterator.hasNext() && foundTransition == false)
                   {
                            Transition nextTransition = TransitionsIterator.next();
                            if (nextTransition.readState.equals(CurrentState) &&
nextTransition.readSymbol == Tape.charAt(CurrentSymbol))
                            {
                                    foundTransition = true;
                                    CurrentTransition = nextTransition;
                            }
                   }

                    if (foundTransition == false)
                    {
                            System.err.println ("There is no valid transition for this
phase! (state=" + CurrentState + ", symbol=" + Tape.charAt(CurrentSymbol) + ")");
                            return false;
                    }
                    else
                    {
                            CurrentState = CurrentTransition.writeState;
                            char[] tempTape = Tape.toCharArray();

                            tempTape[CurrentSymbol] =
CurrentTransition.writeSymbol;
                            Tape =  new String(tempTape);
                            if(CurrentTransition.moveDirection==true)
                            {
                                    CurrentSymbol++;
                            }
```

```
                        else
                        {
                                CurrentSymbol--;
                        }

                        if (CurrentSymbol < 0)
                                CurrentSymbol = 0;

                        while (Tape.length() <= CurrentSymbol)
                        {
                                Tape = Tape.concat("_");

                }

        }

        if (CurrentState.equals(AcceptState))
        {
                return true;
        }
        else
        {
                return false;
        }
}

public boolean addState(String newState)
{
        if (StateSpace.contains(newState))
        {
                return false;
        }
        else
        {
                StateSpace.add(newState);
                return true;
        }
}

public boolean setStartState(String newStartState)
{
        if (StateSpace.contains(newStartState))
        {
                StartState = newStartState;
return true;
        }
        else
        {
```

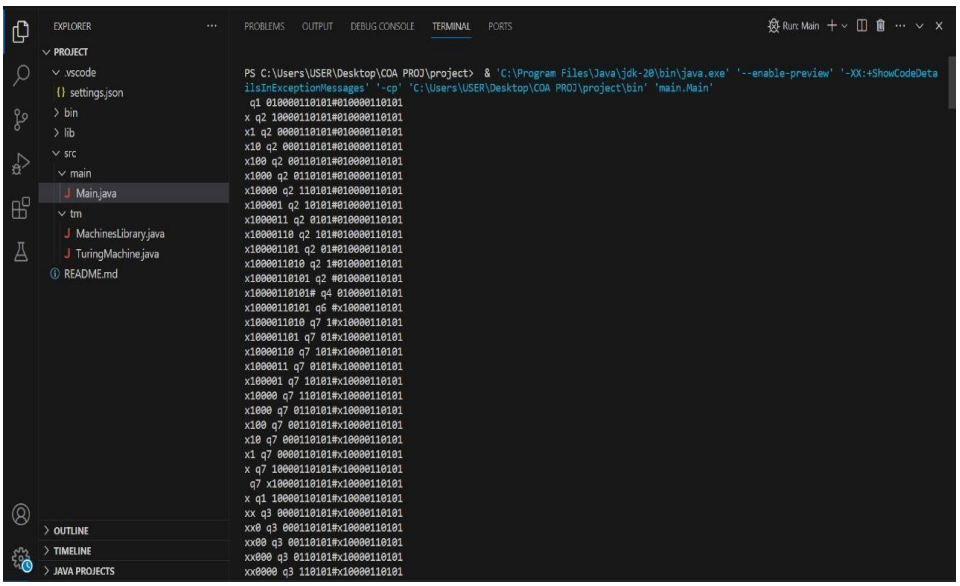
```
                return false;
            }
        }

        public boolean setAcceptState(String newAcceptState)
        {
            if (StateSpace.contains(newAcceptState) &&
!RejectState.equals(newAcceptState))
            {
                AcceptState = newAcceptState;
                return true;
            }
            else
            {
                return false;
            }

        }
        public boolean setRejectState(String newRejectState)
        {
            if (StateSpace.contains(newRejectState) &&
!AcceptState.equals(newRejectState))
            {
                RejectState = newRejectState;
            return true;
            }
            else
            {
                return false;
            }
        }

        public boolean addTransition(String rState, char rSymbol, String wState, char
wSymbol, boolean mDirection)
        {
            if(!StateSpace.contains(rState) || !StateSpace.contains(wState))
            {
                return false;
            }

            boolean conflict = false;
            Iterator<Transition> TransitionsIterator = TransitionSpace.iterator();
while ( TransitionsIterator.hasNext() && conflict == false)
          {
                 Transition nextTransition = TransitionsIterator.next();
```

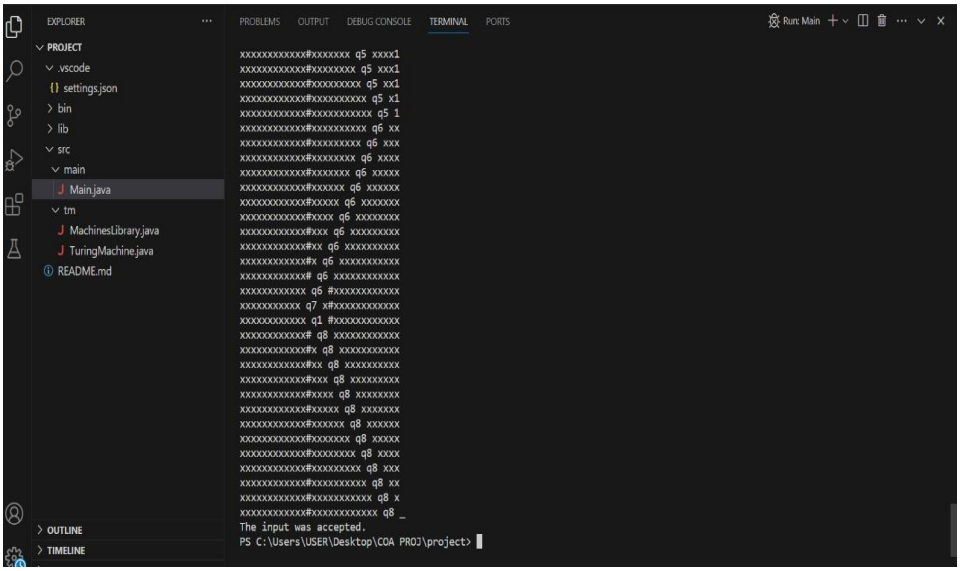
```
                    if (nextTransition.isConflicting(rState, rSymbol))
                    {
                        conflict = true;
                    }

            }
            if (conflict == true)
            {
                return false;
            }
            else
            {
                    Transition newTransition = new Transition();
            newTransition.readState = rState; newTransition.readSymbol =
            rSymbol;
                newTransition.writeState = wState;
                newTransition.writeSymbol = wSymbol;
                newTransition.moveDirection = mDirection;
                TransitionSpace.add(newTransition);
                return true;
            }
        }
}
```

FLOWCHART:


Initialize Turing Machine
Define states, symbols, and transitions
not All Transitions Defined?
Yes
Define next transition
Start State = q0
not Halted?
No
Read symbol from tape
Determine next state and symbol
Write symbol to tape
Move tape head
Halting State Reached
Stop Turing Machine

OUTPUT:





CONCLUSIONS:

Implementing a Turing machine in Java for basic operations involves creating classes to represent the machine's components and simulating its behavior, providing a hands-on understanding of computational theory.

REFERENCES:

- https://www.javatpoint.com/automata-turing-machine