Program:

```
import java.util.*;
public class PDA {
    private Stack<Character> stack;
    private String inputString;
    private char startSymbol;
     private Map<Character, List<String>> productionRules;
public PDA(char startSymbol, Map<Character, List<String>> productionRules) {
      this.stack = new Stack<>();
      this.inputString = "";
      this.startSymbol = startSymbol;
      this.productionRules = productionRules;
  }
public boolean accepts(String inputString) {
      this.inputString = inputString;
      this.stack.push(startSymbol); // Initialize stack with start symbol
      System.out.println("Initial Stack: " + stack);
      return simulate();
  }
  private boolean simulate() {
        int i = 0;
     while (!stack.isEmpty()) {
          char topOfStack = stack.pop();
    Character currentInput = (i < inputString.length()) ? inputString.charAt(i) : null;
        System.out.println("Current Stack: " + stack);
        System.out.println("Current Input: " + currentInput + ", Top of Stack: " + topOfStack);
        if (Character.isUpperCase(topOfStack)) {
            if (productionRules.containsKey(topOfStack)) {
                boolean ruleApplied = false;
             for (String production : productionRules.get(topOfStack)) {
                 if (production.startsWith("" + currentInput) || production.equals("ε")) {
                   System.out.println("Applying Production: " + topOfStack + " -> " + production);
                   for (int j = production.length() - 1; j >= 0; j--) {
                      if (production.charAt(j) != 'ε') {
```

```
                    stack.push(production.charAt(j));
                }   }
                ruleApplied = true;
                break;
            } }
        if (!ruleApplied) {
            System.out.println("No rule applicable for " + topOfStack);
            return false;
        }
    } else {
        System.out.println("No production for non-terminal " + topOfStack);
        return false;
    }   }
    else if (topOfStack == currentInput) {
        System.out.println("Matched terminal " + topOfStack + " with input " + currentInput);
        i++; // Move to the next input character
    } else {
        System.out.println("Terminal mismatch: stack " + topOfStack + " vs input " + currentInput);
        return false;
    }  }
  return i == inputString.length();
}
public static void main(String[] args) {
  // Example CFG
  // Grammar: S -> a | aS | bSS | SSb | SbS
  Map<Character, List<String>> cfgRules = new HashMap<>();
  cfgRules.put('S', Arrays.asList("a", "aS", "bSS", "SSb", "SbS"));
  char startSymbol = 'S';
  PDA pda = new PDA(startSymbol, cfgRules);
  String inputString = "baa";
  if (pda.accepts(inputString)) {
    System.out.println("The input string '" + inputString + "' is accepted by the PDA.");
  } else {
    System.out.println("The input string '" + inputString + "' is rejected by the PDA.");  }}}
```

OUTPUT:

```
Initial Stack: [S]
Current Stack: []
Current Input: b, Top of Stack: S
Applying Production: S -> bSS
Current Stack: [S, S]
Current Input: b, Top of Stack: b
Matched terminal b with input b
Current Stack: [S]
Current Input: a, Top of Stack: S
Applying Production: S -> a
Current Stack: [S]
Current Input: a, Top of Stack: a
Matched terminal a with input a
Current Stack: []
Current Input: a, Top of Stack: S
Applying Production: S -> a
Current Stack: []
Current Input: a, Top of Stack: a
Matched terminal a with input a
The input string 'baa' is accepted by the PDA.
```

OBSERVATION:

In this experiment, I learned about how to convert context free grammar into push down automata using top down approach by using stack – based simulation. Here PDA behavior is simulated by ensuring that each production rule is applied in a leftmost derivation, similar to how a recursive-descent parser works

CONCLUSION:

Through this experiment, I gained a deeper understanding of how push down automata using top down approach works . It applies stack – based simulation. Acceptance condition in push down automata is that the input string is accepted if it is fully consumed, and the stack is empty.