COMPILER DESIGN LAB
WEEK 2

SYMBOL TABLE REPORT

Name: C. Lokesh Swarup
Reg No.: AP21110011012
Section: CSE P

TITLE:

Symbol Table Implementation

STATEMENT:

Symbol table is an important data structure created and maintained by compilers in order to store information about the occurrence of various identifiers such as variable names, function names, objects, classes, interfaces, etc. Symbol table is used by both the analysis and the synthesis parts of a compiler. Symbol table can be implemented in one of the following ways:

- Linear (sorted or unsorted) list
- Binary Search Tree
- Hash table
- And other ways.

In this lab session, we are required to analyse the various implementations. We need to write code for at least two ways of implementation. We tested our code with different test cases. Submitted a report of our analysis and executable code.

CODE 1:

```
//Binary Search Tree implementation:
 class TreeNode {
    String key;
    int value;
    TreeNode left;
    TreeNode right;

    public TreeNode(String key, int value) {
       this.key = key;
       this.value = value;
       this.left = null;
       this.right = null;
    }
 }
```

```
class BST {
    TreeNode root;

    public void insert(String key, int value) {
        root = insertRecursive(root, key, value);
    }

    private TreeNode insertRecursive(TreeNode current, String key, int value) {
        if (current == null) {
            return new TreeNode(key, value);
        }
        if (key.compareTo(current.key) < 0) {
            current.left = insertRecursive(current.left, key, value);
        } else if (key.compareTo(current.key) > 0) {
            current.right = insertRecursive(current.right, key, value);
        }
        return current;
    }

    public Integer search(String key) {
        return searchRecursive(root, key);
    }

    private Integer searchRecursive(TreeNode current, String key) {
        if (current == null || current.key.equals(key)) {
            return current != null ? current.value : null;
        }
        if (key.compareTo(current.key) < 0) {
            return searchRecursive(current.left, key);
        }
        return searchRecursive(current.right, key);
    }
}

public class BinarySearchTree {
    public static void main(String[] args) {
        BST symbolTable = new BST();
        symbolTable.insert("variable1", 42);
        symbolTable.insert("variable2", 78);
        symbolTable.insert("variable3", 123);

        System.out.println(symbolTable.search("variable1")); // Output: 42
        System.out.println(symbolTable.search("variable2")); // Output: 78
        System.out.println(symbolTable.search("variable3")); // Output: 123
        System.out.println(symbolTable.search("nonexistent")); // Output: null
    }
}
```

OUTPUT:

```
42
78
123
Null
```

CONCLUSION:

The provided Java code demonstrates the implementation of a Symbol Table using a Binary Search Tree, allowing efficient storage and retrieval of key-value pairs for quick data lookup.

CODE 2:

```
//Hash Table Implementation
import java.util.Scanner;

public class HashTable {
   public static void main(String[] args) {
      int n, i = 0, j = 0;
      char[] id_Array2 = new char[20];
      char ch;

      System.out.println("Input the expression ending with ; sign:");
      Scanner sc = new Scanner(System.in);
      String input = sc.next();

      while (i < input.length() && input.charAt(i) != ';') {
         id_Array2[i] = input.charAt(i);
         i++;
      }
      n = i - 1;
      System.out.println("\nSymbol Table display");
      System.out.println("Symbol \t addr \t\t\t type");

      while (j <= n) {
         ch = id_Array2[j];

         if (Character.isLetter(ch)) {
            System.out.println("\n " + ch + " \t " + System.identityHashCode(ch) + " \t identifier");
            j++;
         } else {
            if (ch == '+' || ch == '-' || ch == '*' || ch == '/' || ch == '%' || ch == '=' || ch == '<' || ch == '>')
{
               System.out.println("\n " + ch + " \t " + System.identityHashCode(ch) + " \t operator");
               j++;
            }
         }
      }
   }
}
```

INPUT: s=a+b;

OUTPUT: Input the expression ending with s=a+b;

Symbol Table display

Symbol	addr	type
s	1989780873	identifier
=	284720968	operator
a	189568618	identifier
+	793589513	operator
b	1313922862	identifier

CONCLUSION:

The provided Java code is a basic implementation of a symbol table using a Hash Table for identifying and categorizing identifiers and operators within an input expression. It demonstrates a simple approach to symbol table functionality.