WEEK 2

Symbol Table:

A symbol table is a fundamental data structure used in computer science, especially in programming language compilers, interpreters, and assemblers.

It is responsible for storing and managing information about symbols (identifiers) used in a program, such as variable names, function names, and labels.

The primary purpose of a symbol table is to facilitate the efficient retrieval and management of information associated with these symbols during various phases of the compilation or execution process.

Implementation of Symbol table using C language:

1. *Using linear List*

```c
#include <stdio.h>
#include <string.h>

#define MAX_SYMBOLS 100

struct SymbolEntry {
    char name[50];
    int value;
};

struct SymbolTable {
    struct SymbolEntry entries[MAX_SYMBOLS];
    int size;
};

void initSymbolTable(struct SymbolTable* table) {
    table->size = 0;
}

int insertSymbol(struct SymbolTable* table, const char* name, int value) {
    if (table->size >= MAX_SYMBOLS) {
        return 0; // Table is full
    }

    for (int i = 0; i < table->size; i++) {
        if (strcmp(name, table->entries[i].name) == 0) {
```

```
            table->entries[i].value = value;
            return 1;
        }
    }
    strncpy(table->entries[table->size].name, name, sizeof(table->entries[table->size].name));
    table->entries[table->size].value = value;
    table->size++;
    return 1;
}

int lookupSymbol(struct SymbolTable* table, const char* name) {
    for (int i = 0; i < table->size; i++) {
        if (strcmp(name, table->entries[i].name) == 0) {
            return table->entries[i].value;
        }
    }
    return -1;
}

int main() {
    struct SymbolTable symbolTable;
    initSymbolTable(&symbolTable);

    insertSymbol(&symbolTable, "variable1", 42);
    insertSymbol(&symbolTable, "variable2", 56);

    int value1 = lookupSymbol(&symbolTable, "variable1");
    int value2 = lookupSymbol(&symbolTable, "variable2");
    int value3 = lookupSymbol(&symbolTable, "variable3");

    printf("Value of variable1: %d\n", value1); // Output: 42
    printf("Value of variable2: %d\n", value2); // Output: 56
    printf("Value of variable3: %d\n", value3); // Output: -1 (not found)

    return 0;
}

```

2. ***Using Binary Search Tree***

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct TreeNode {
    char key[50];
    int value;
    struct TreeNode* left;
    struct TreeNode* right;
};

struct TreeNode* createTreeNode(const char* key, int value) {
    struct TreeNode* newNode = (struct TreeNode*)malloc(sizeof(struct TreeNode));
    if (newNode != NULL) {
        strncpy(newNode->key, key, sizeof(newNode->key));
        newNode->value = value;
        newNode->left = NULL;
        newNode->right = NULL;
    }
    return newNode;
}

struct TreeNode* insert(struct TreeNode* current, const char* key, int value) {
    if (current == NULL) {
        return createTreeNode(key, value);
    }
    int compareResult = strcmp(key, current->key);
    if (compareResult < 0) {
        current->left = insert(current->left, key, value);
    } else if (compareResult > 0) {
        current->right = insert(current->right, key, value);
    }
    return current;
}

int search(struct TreeNode* current, const char* key) {
    if (current == NULL || strcmp(key, current->key) == 0) {
        return (current != NULL) ? current->value : -1;
    }
    int compareResult = strcmp(key, current->key);
    if (compareResult < 0) {
```

```
        return search(current->left, key);
    }
    return search(current->right, key);
}

int main() {
    struct TreeNode* root = NULL;
    root = insert(root, "variable1", 42);

    int result = search(root, "variable1");
    if (result != -1) {
        printf("Value found: %d\n", result);
    } else {
        printf("Key not found\n");
    }
    return 0;
}
```