Implementation of Symbol

The symbol table can be implemented in the unordered list if the compiler is used to handle the small amount of data.
A Symbol table can be implemented in one of the following techniques:

• Linear (sorted or unsorted) list
• Binary Search Tree
• Hash table

Symbol tables are mostly implemented as Hash Table

The operations provided by symbol table are
• Insert()
• lookup()

1. Insertion:
    - When a new symbol (like a variable or function) is encountered in the source code for the first time, it's inserted into the symbol table.
    - Along with the symbol's name, additional information might be stored. This can include data type, scope level, memory address, size, and more, depending on the application's needs.
2. Lookup:
    - Whenever a symbol is referenced in the source code, the compiler or interpreter looks up the symbol in the table to get its associated details.
    - This can be to check if a variable has been declared before use, to get the data type of a variable during type-checking, or to fetch its memory address during code generation.

Two common ways to implement a symbol table are using hash tables and balanced binary search trees. Each has its advantages and trade-offs. Hash tables

offer fast average-case lookup times, while balanced binary search trees maintain items in sorted order and provide efficient range queries.

Binary Search Tree (BST):

- Symbols are stored in a binary tree structure.
- Each node has a symbol; left children are symbols less than the node, and right children are symbols greater than the node.
- Lookup, insertion, and deletion are O(log n) in a balanced tree, but can degrade to O(n) in the worst-case scenario (unbalanced tree).

Hash Table:

- One of the most common and efficient implementations for symbol tables.
- Symbols are hashed to an index in an array (the hash table).
- Collision resolution strategies, like chaining (linked lists at each index) or open addressing, are employed.
- On average, operations like lookup, insertion, and deletion are O(1), but can degrade based on the load factor and the quality of the hash function.

Procedure for hash table for a symbol table:

1.Setup: A HashTable class is defined, which contains a vector of linked lists. Each linked list will hold pairs of key-value entries.

2.Insert: The insert function takes a key and a value as parameters. It calculates the hash of the key, determines the appropriate index in the vector, and inserts a new key-value pair into the linked list at that index.

3.Search: The search function takes a key as a parameter. It calculates the hash of the key, determines the index in the vector, and searches through the linked list at that index for the specified key. If found, it returns the associated value; otherwise, it returns -1 to indicate that the key is not present.

4.Remove: The remove function takes a key as a parameter. It calculates the hash of the key, determines the index in the vector, and searches through the linked list at that index to find and remove the specified key-value pair.

5.Main Program: In the main function, a HashTable object named symbolTable is created. Key-value pairs are inserted into the hash table using the insert method. The search method is then used to retrieve and display the values associated with certain keys. After that, the remove method is used to remove a specific key-value pair from the hash table.

6.Finish: The program concludes execution.

Procedure for binary search tree:

1.Structure Definition: The code defines a struct Tree structure with members key, value, left, and right. This structure represents a node in the binary search tree. Each node has a key (string), a value (integer), and pointers to its left and right children.

2.Insertion (Recursive): The insertRecursive function is used to insert nodes into the binary search tree. It takes a current node pointer (current), a key (key), and a value (value) as parameters. If the current node is NULL, a new node is dynamically allocated, and its key, value, left, and right pointers are set accordingly. If the key is less than the current node's key, the function is called recursively on the left subtree; otherwise, it's called on the right subtree.

4.Search (Recursive): The searchRecursive function is used to search for a given key in the binary search tree. It takes a current node pointer (current) and a key (key) as parameters. If the current node is NULL or the keys match, it returns the value associated with the key (or -1 if not found). If the key is less than the current node's key, the function is called recursively on the left subtree; otherwise, it's called on the right subtree.

5.Main Function: In the main function, a pointer to the root of the binary search tree is initialized as NULL. Then, the insertRecursive function is used to insert a node with key "variable1" and value 93 into the tree. Finally, the searchRecursive function is called to search for the key "variable1," and the associated value is printed.

Code(binary search tree):

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

struct Tree {
    char key[100];
    int value;
    struct Tree* left;
    struct Tree* right;
};

struct Tree* insertRecursive(struct Tree* current, char* key, int value) {
    if (current == NULL) {
        struct Tree* newNode = (struct Tree*)malloc(sizeof(struct Tree));
        strcpy(newNode->key, key);
        newNode->value = value;
        newNode->left = NULL;
        newNode->right = NULL;
        return newNode;
    }
    if (strcmp(key, current->key) < 0) {
        current->left = insertRecursive(current->left, key, value);
    } else if (strcmp(key, current->key) > 0) {
        current->right = insertRecursive(current->right, key, value);
    }
    return current;
}

int searchRecursive(struct Tree* current, char* key) {
    if (current == NULL || strcmp(current->key, key) == 0) {
```

```
        return current != NULL ? current->value : -1;
    }
    if (strcmp(key, current->key) < 0) {
        return searchRecursive(current->left, key);
    }
    return searchRecursive(current->right, key);
}

int main() {
    struct Tree* root = NULL;
    root = insertRecursive(root, "variable1", 93);
    printf("%d\n", searchRecursive(root, "variable1"));
    return 0;
}
```

INPUT:

No direct input is required in this code. The input is embedded in the code itself.

OUTPUT:

93

Symbol Table display :

Symbol	type
Variable1	identifier

CONCLUSION:

Symbol table is a data structure used by the compiler, where each identifier in program's source code is stored along with information associated with it relating to its declaration**.**

Code(hash table):

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define HASH_TABLE_SIZE 100
```

```
struct SymbolEntry
{
    char *name;
    int value;
    struct SymbolEntry *next;
};

struct SymbolTable
{
    struct SymbolEntry *hash_table[HASH_TABLE_SIZE];
};

unsigned int hash(const char *str)
{
    unsigned int hash = 0;
    while(*str)
    {
        hash = (hash << 5) + *str++;
    }
    return hash % HASH_TABLE_SIZE;
}

void insert(struct SymbolTable *table, const char *name, int value)
{
    unsigned int index = hash(name);
    struct SymbolEntry *entry = (struct SymbolEntry *)malloc(sizeof(struct
SymbolEntry));
    if (!entry)
    {
        perror("Memory allocation failed");
        exit(EXIT_FAILURE);
    }
    entry->name = strdup(name);
    entry->value = value;
    entry->next = table->hash_table[index];
    table->hash_table[index] = entry;
```

```
}

struct SymbolEntry *search(struct SymbolTable *table, const char
*name)
{
    unsigned int index = hash(name);
    struct SymbolEntry *entry = table->hash_table[index];
    while (entry != NULL)
    {
        if (strcmp(entry->name, name) == 0)
        {
            return entry;
        }
        entry = entry->next;
    }
    return NULL;
}

int main()
{
    struct SymbolTable symbol_table;
    for (int i = 0; i < HASH_TABLE_SIZE; i++)
    {
        symbol_table.hash_table[i] = NULL;
    }

    insert(&symbol_table, "x", 93);
    insert(&symbol_table, "y", 27);

    struct SymbolEntry *entry_x = search(&symbol_table, "x");
    if (entry_x)
    {
        printf("Symbol: %s, Value: %d\n", entry_x->name, entry_x->value);
    } else
    {
        printf("Symbol not found.\n");
    }
```

```
    for (int i = 0; i < HASH_TABLE_SIZE; i++)
    {
        struct SymbolEntry *entry = symbol_table.hash_table[i];
        while (entry)
        {
            struct SymbolEntry *next = entry->next;
            free(entry->name);
            free(entry);
            entry = next;
        }
    }
    return 0;
```

OUTPUT :

```
Symbol: x, Value: 93
```