SYMBOL TABLE LAB REPORT

Hemanth challa

ROLL.NO-AP21110010982

SEC:CSE-P

TITLE:

Symbol Table Implementation

STATEMENT:

The symbol table is a fundamental construct in compiler design, serving as a repository for essential information about identifiers like variable names, functions, classes, and more. Integral to both the analysis and synthesis phases of compilation, it enables efficient storage and retrieval of identifier-related data. Symbol tables can be realized using diverse techniques, including hash tables. Hash tables efficiently allocate and access data by employing hash functions, ensuring rapid insertion and retrieval operations. This approach's constant-time average-case access performance is pivotal in enhancing the overall efficiency and accuracy of compiler processes, such as lexical and semantic analysis, as well as code generation.

In this lab session, you are required to analyse the various implementations. You need to write

code for at least two ways of implementation. Test your code with different test cases. Submit

a report of your analysis and executable code by the end of the session.

PROCEDURE:

Absolutely, here's a concise breakdown:

1. **Setup**: Begin by defining a "tree" structure for storing data with keys and values.
2. **Insert**: Create a way to add data to the tree. If the tree is empty, make a new node with a key and value. If not, compare the key and put the new node on the left or right.
3. **Search**: Develop a method to find data. If there's no match or no node, say "not found" or share the value found.
4. **Example**: Create a tree and add a node with key "variable1" and value 93. Search for "variable1" and show its value.
5. **Finish**: The program completes its tasks.

In short, this code sets up a tree to store and find data using keys and values, highlighting adding and searching operations.

CODE 1:

```
// AP21110010982
#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 2:

```
//AP21110010982
#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
*/
```