**AP21110010976**

**Implementation of Symbol Table :**

# 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

## Linked List

* + This implementation is using a linked list. A link field is added to each record.
  + Searching of names is done in order pointed by the link of the link field.
  + A pointer "First" is maintained to point to the first record of the symbol table.
  + Insertion is fast O(1), but lookup is slow for large tables - O(n) on average

## Hash Table

* + A hash table is an array with an index range: 0 to table size - 1. These entries are pointers pointing to the names of the symbol table.
  + To search for a name we use a hash function that will result in an integer between 0 to table size - 1.
  + Insertion and lookup can be made very fast - O(1).
  + The advantage is quick to search is possible and the disadvantage is that hashing is complicated to implement.

## Binary Search Tree

* + Another approach to implementing a symbol table is to use a binary search tree i.e. we add two link
  + fields i.e. left and right child.
  + All names are created as child of the root node that always follows the property of the binary search tree.
  + Insertion and lookup are O(log2 n) on average.

**Code**

#include <stdio.h>

#include <ctype.h> #include <stdlib.h> int main()

{ int x = 0, n, i = 0, j = 0, p = 0;

void \*ptr,

\*id\_address[5]; char ch, id\_Array2[25],

id\_Array3[25], c; printf("Input the expression that ends with ; sign:"); char s[30];



**Code Using (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", 59); 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: 59