PES UNIVERSITY

PROJECT REPORT ON

IMPLEMENTATION OF MAP IN C USING GENERICS

BY

Sumedh Arani 01FB14ECS258
Vandana V Kulkarni 01FB14ECS276
Srilakshmi M Bharadwaj 01FB14ECS251

Program : BTECH – CSE
Semester : IV
Section : E

INTRODUCTION

In computing, a **hash map** is a data structure used to implement an associative array, a structure that can map keys to values. A hash table uses a hash function to compute an *index* into an array of *buckets* or *slots*, from which the desired value can be found.

Ideally, the hash function will assign each key to a unique bucket, but it is possible that two keys will generate an identical hash causing both keys to point to the same bucket. Instead, most hash table designs assume that hash *collisions* —different keys that are assigned by the hash function to the same bucket—will occur and must be accommodated in some way.

In a well-dimensioned hash table, the average cost (number of instructions) for each lookup is independent of the number of elements stored in the table. Many hash table designs also allow arbitrary insertions and deletions of key-value pairs, at constant average cost per operation.

In many situations, hash tables turn out to be more efficient than search trees or any other table lookup structure. For this reason, they are widely used in many kinds of computer software, particularly for associative arrays, database indexing, caches, and sets.

PROJECT AIM

The aim of this project is to implement hash map in C using generics. In order to achieve generic programming in C the #preprocessor macro is being used. A macro is a piece of code that was labeled with a name. Whenever the preprocessor encounters it, the label is replaced by the associated code. Basically two kinds of macros are being used : object-like macros (resemble data objects when used) and function-like macros (resemble function calls).

IMPLEMENTATION

Hash map is implemented using open hashing. In open hashing, keys are stored in linked lists attached to cells of a hash table. Every cell of hash table points to the head of a linked list. The nodes of the linked list has three parts : pointer to key, poinetr to value(value in turn can be a list), pointer to next node. In case two keys correspond to the same hash index, a new node is created, which is inseted at the end of the linked list at that particular hash cell.

Structure :

0	1	2	3	4	5	6

Key ptr	Value ptr	

Key ptr	Value ptr	

Use of macros

The data types of key and value is passed as an argument to map_define() and map_declare() which serves as an interface for defining the map. These data types are interpreted accordingly in the functions being accessed. #preprocessor can be used to concatenate a particulay data type with an identifier. All the structures use object-like macros whereas the function definitions use function-like macros.

Examples

- Object-like macro :

```
#define list_declare(type) \

typedef struct type##_lnode \

{ \
        type *data; \

        struct type##_lnode *next; \

}type##_node; \
```

- Function-like macro:

```
void list_##type##_insert(type##_node** head, type *data);
```

Hash function

The hash function for integer is as follows:

```
int hashcode_int(int* key)

{
        int code = 0;
        code = ((*key) * 9) % 16;
        return code;
}
```

The size of the hash table is 16. The hash index is calculated by multiplying the integer with a number that is of the order of hash table size (2^4 in this case). The multiplier should not be a factor of 16. The hash index thus obtained will be the remainder of division of the result of multiplication with the hash table size. In this way, enough variation is produced in the value of hash index obtained, thus minimizing collisions.

The hash function for strings/character is as follows:

```
int hashcode_char(char* key)

{
        int i;
        int code = 0;
        int len = 0;
        len = strlen(key)-1;
        for(i = 0; i <= len; ++i)
        {
                if(key[i] >= 65 && key[i] <= 90)
                        code += (key[i] - 65 + 1);
                else if(key[i] >= 97 && key[i] <= 122)
                        code += (key[i] - 97 + 1);
        }
        return code % 16;
}
```

In this case the hash index is computed by taking the summation of the position of the alphabets in the alphabet set, the result of which is divided by the hash table size. The remainder of the division provides the hash index.

The insert function

Prototype:

```
void
map_##keytype##_##valuetype##_insert(keytype##_##valuetype##_node**
head, keytype *k, valuetype *v);
```

The required key value pairs to be inseted in the map is passed as an argument to map_insert(). The macros interpret these data types accordingly. The argument to the function is a string of key-value pairs separated by colon (:) and different key-value pairs are separated by (;). This string is processed in hash_sfind() and the keys and values are separated and inseted into the map. Each time the hash index corresponding to the key is calculated and the value corresponding to the key is inserted at the hash index. In case of collisions or first node a new node is created and appended at the end of the list at the desired hash index.

Delete and search function

Prototype:
Delete : void map_keytype##_valuetype##_delete(keytype* k, int index);
Search : void map_keytype##_valuetype##_display(keytype* k, int index);

The key to be searched or deleted is passed as an argument to search or delete function. The hash index corresponding to the key is calculated . If the desired key is found at the hash index computed the value corresponding to the key is displayed in case of search or the key-value pair is deleted in case of delete respectively. If the key is not found an error message "Key not found" is displayed.

In additional to these functions there are some additional intermediate functions to compute the hash index for serach ,delete and insert respectively.

RESULTS

Works for large files.(Tested for 1000)
Handles duplicate keys.
Works for composite key type.
Provision for user defined hash function