15. IMPLEMENTATION OF HASH TABLE

Preamble

Hashing is a technique that maps a large set of data to a small set of data. It uses a hash function for doing this mapping. It is an irreversible process and we cannot find the original value of the key from its hashed value because we are trying to map a large set of data into a small set of data, which may cause collisions. It is not uncommon to encounter collisions when mapping a large dataset into a smaller one. Suppose, We have three buckets and each bucket can store 1L of water in it and we have 5L of water also. We have to put all the water in these three buckets and this kind of situation is known as a collision. URL shorteners are an example of hashing as it maps large size URL to small size

Hash Table is a data structure which stores data in an associative manner. In hash table, the data is stored in an array format where each data value has its own unique index value. Access of data becomes very fast, if we know the index of the desired data.

Steps

- Define key-value pair.
- Define the functions for the desired actions.
  - insertion
  - Search
  - Hash Function
  - Delete
  - Rehashing (Optional)

Implementation in C

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>


// Linked List node

struct node

{


    // key is string

    char* key;

    // value is also string
```

```
      char* value;
      struct node* next;
};

// like constructor
void setNode(struct node* node, char* key, char* value)
{
      node->key = key;
      node->value = value;
      node->next = NULL;
      return;
};

struct hashMap
{

      // Current number of elements in hashMap
      // and capacity of hashMap
      int numOfElements, capacity;
      // hold base address array of linked list
      struct node** arr;
};

// like constructor
void initializeHashMap(struct hashMap* mp)
{
      // Default capacity in this case
      mp->capacity = 100;
      mp->numOfElements = 0;
      // array of size = 1
      mp->arr = (struct node**)malloc(sizeof(struct node*)
      * mp->capacity);
```

```
	return;
}

int hashFunction(struct hashMap* mp, char* key)
{
	int bucketIndex;
	int sum = 0, factor = 31;
	for (int i = 0; i < strlen(key); i++)
	{
		// sum = sum + (ascii value of
		// char * (primeNumber ^ x))...
		// where x = 1, 2, 3....n
		sum = ((sum % mp->capacity)
		+ (((int)key[i]) * factor) % mp->capacity)% mp->capacity;

		// factor = factor * prime
		// number....(prime
		// number) ^ x
		factor = ((factor % __INT16_MAX__) * (31 % __INT16_MAX__))
		% __INT16_MAX__;
	}
	bucketIndex = sum;
	return bucketIndex;
}

void insert(struct hashMap* mp, char* key, char* value)
{
	// Getting bucket index for the given
	// key - value pair
	int bucketIndex = hashFunction(mp, key);
	struct node* newNode = (struct node*)malloc(
            // Creating a new node
            sizeof(struct node));
      // Setting value of node
      setNode(newNode, key, value);
      // Bucket index is empty....no collision
      if (mp->arr[bucketIndex] == NULL)
      {
            mp->arr[bucketIndex] = newNode;
      }
     // Collision
      else
      {

            // Adding newNode at the head of
            // linked list which is present
            // at bucket index....insertion at
            // head in linked list
            newNode->next = mp->arr[bucketIndex];
            mp->arr[bucketIndex] = newNode;
      }
      return;
}

void delete (struct hashMap* mp, char* key)
{
      // Getting bucket index for the
      // given key
      int bucketIndex = hashFunction(mp, key);
      struct node* prevNode = NULL;
      // Points to the head of
      // linked list present at
      // bucket index
```

```
	struct node* currNode = mp->arr[bucketIndex];
	while (currNode != NULL)
	{
		// Key is matched at delete this
		// node from linked list
		if (strcmp(key, currNode->key) == 0)
		{
			// Head node
			// deletion
			if (currNode == mp->arr[bucketIndex])
			{
				mp->arr[bucketIndex] = currNode->next;
			}
			// Last node or middle node
			else
			{
			prevNode->next = currNode->next;
			}
			free(currNode);
			break;
		}
		prevNode = currNode;
		currNode = currNode->next;
	}
	return;
}

char* search(struct hashMap* mp, char* key)
{
	// Getting the bucket index
	// for the given key
	int bucketIndex = hashFunction(mp, key);
```

```
      // Head of the linked list
      // present at bucket index
      struct node* bucketHead = mp->arr[bucketIndex];
      while (bucketHead != NULL)
      {
            // Key is found in the hashMap
            if (bucketHead->key == key)
            {
                  return bucketHead->value;
            }
            bucketHead = bucketHead->next;
    }

    // If no key found in the hashMap
    // equal to the given key
    char* errorMssg = (char*)malloc(sizeof(char) * 25);
    errorMssg = "Oops! No data found.\n";
    return errorMssg;
}

// Drivers code
int main()
{
      // Initialize the value of mp
      struct hashMap* mp = (struct hashMap*)malloc(sizeof(struct hashMap));
      initializeHashMap(mp);
      insert(mp, "Yogaholic", "Anjali");
      insert(mp, "pluto14", "Vartika");
      insert(mp, "elite_Programmer", "Manish");
      insert(mp, "GFG", "BITS");
      insert(mp, "decentBoy", "Mayank");
      printf("%s\n", search(mp, "elite_Programmer"));
      printf("%s\n", search(mp, "Yogaholic"));
      printf("%s\n", search(mp, "pluto14"));
      printf("%s\n", search(mp, "decentBoy"));
      printf("%s\n", search(mp, "GFG"));
      // Key is not inserted
      printf("%s\n", search(mp, "randomKey"));
      printf("\nAfter deletion : \n");
     // Deletion of key
      delete (mp, "decentBoy");
      printf("%s\n", search(mp, "decentBoy"));
      return 0;
}
```

Sample Input and Output

```
Manish
Anjali
Vartika
Mayank
BITS
Oops! No data found.

After deletion :
Oops! No data found.
```

Explanation:

Insertion: Inserts the key-value pair at the head of a linked list which is present at the given bucket index.

*HashFunction: Gives the bucket index for the given key. Our hash function = ASCII value of character * primeNumber^x. The prime number in our case is 31 and the value of x is increasing from 1 to n for consecutive characters in a key.*

Deletion: Deletes key-value pair from the hash table for the given key. It deletes the node from the linked list which holds the key-value pair.

Search: Search for the value of the given key.