Rajalakshmi Engineering College

Name: Alwin Abishek
Email: 241501017@rajalakshmi.edu.in
Roll no: 241501017
Phone: 9444177993
Branch: REC
Department: I AI & ML FA
Batch: 2028
Degree: B.E - AI & ML

Scan to verify results

NeoColab_REC_CS23231_DATA STRUCTURES

REC_DS using C_Week 7_COD_Question 4

Attempt : 1
Total Mark : 10
Marks Obtained : 10

Section 1 : Coding

1. Problem Statement

Develop a program using hashing to manage a fruit contest where each fruit is assigned a unique name and a corresponding score. The program should allow the organizer to input the number of fruits and their names with scores.

Then, it should enable them to check if a specific fruit, identified by its name, is part of the contest. If the fruit is registered, the program should display its score; otherwise, it should indicate that it is not included in the contest.

Input Format

The first line consists of an integer N, representing the number of fruits in the contest.

The following N lines contain a string K and an integer V, separated by a space, representing the name and score of each fruit in the contest.

The last line consists of a string T, representing the name of the fruit to search for.

Output Format

If T exists in the dictionary, print "Key "T" exists in the dictionary.".

If T does not exist in the dictionary, print "Key "T" does not exist in the dictionary.".

Refer to the sample outputs for the formatting specifications.

Sample Test Case

Input: 2
banana 2
apple 1
Banana
Output: Key "Banana" does not exist in the dictionary.

Answer

```
#include <stdio.h>   // Required for standard input/output functions like printf, scanf
#include <stdlib.h>  // Required for dynamic memory allocation functions like malloc, free, exit
#include <string.h>  // Required for string manipulation functions like strcpy, strcmp, strncpy

// Define constants for maximum fruit name length and hash table size.
// TABLE_SIZE is chosen as a prime number (17) slightly larger than the maximum N (15)
// to help with better distribution of keys and reduce collisions.
#define MAX_FRUIT_NAME_LENGTH 50
#define TABLE_SIZE 17

// Structure to represent a single key-value pair (fruit name and its score).
// It also includes a 'next' pointer for separate chaining to handle hash collisions.
```

```
typedef struct KeyValuePair {
    char key[MAX_FRUIT_NAME_LENGTH]; // Stores the fruit name (key)
    int value;                       // Stores the fruit's score (value)
    struct KeyValuePair *next;       // Pointer to the next KeyValuePair in case of a
collision
} KeyValuePair;

// Structure to represent the hash table itself.
// It contains an array of pointers to KeyValuePair, where each pointer is the head
// of a linked list (bucket) for separate chaining.
typedef struct {
    KeyValuePair *buckets[TABLE_SIZE]; // Array of pointers, each pointing to a
linked list of KeyValuePairs
} HashTable;

// --- Hash Function ---
// This function calculates a hash index for a given string (fruit name).
// It uses a simple polynomial rolling hash algorithm to convert the string
// into an integer index within the bounds of TABLE_SIZE.
unsigned int hashFunction(const char *key) {
    unsigned int hash = 0;
    int i = 0;
    // Iterate through each character of the key string
    while (key[i] != '\0') {
        // Multiply the current hash by a prime number (31 is common) and add the
ASCII value of the character.
        // This helps in distributing the hash values more evenly.
        hash = (hash * 31) + key[i];
        i++;
    }
    // Return the hash modulo TABLE_SIZE to ensure the index is within the array
bounds.
    return hash % TABLE_SIZE;
}

// --- Dictionary Operations ---

// Initializes the hash table by setting all bucket pointers to NULL.
// This ensures that all linked lists in the hash table are initially empty.
void initHashTable(HashTable *ht) {
    for (int i = 0; i < TABLE_SIZE; i++) {
        ht->buckets[i] = NULL; // Set each bucket (head of linked list) to NULL
```

```
    }
}

// Inserts a new key-value pair (fruit name and score) into the hash table.
// It calculates the hash index and adds the new pair to the corresponding
bucket's linked list.
void insert(HashTable *ht, const char *key, int value) {
    unsigned int index = hashFunction(key); // Get the hash index for the given key

    // Allocate memory for a new KeyValuePair node.
    KeyValuePair *newNode = (KeyValuePair *)malloc(sizeof(KeyValuePair));
    if (newNode == NULL) {
        // If memory allocation fails, print an error and exit the program.
        fprintf(stderr, "Memory allocation failed for new node during insert!\n");
        exit(EXIT_FAILURE);
    }

    // Copy the key (fruit name) and value (score) into the new node.
    // strncpy is used for safety to prevent buffer overflows, ensuring null-
termination.
    strncpy(newNode->key, key, MAX_FRUIT_NAME_LENGTH - 1);
    newNode->key[MAX_FRUIT_NAME_LENGTH - 1] = '\0'; // Explicitly ensure null-
termination
    newNode->value = value;

    // Add the new node to the beginning of the linked list at the calculated hash
index.
    // This is the separate chaining method for collision resolution.
    newNode->next = ht->buckets[index]; // New node points to the current head of
the list
    ht->buckets[index] = newNode;       // The new node becomes the new head of
the list
}

// Searches for a specific key (fruit name) in the hash table.
// It returns 1 if the key is found, and 0 otherwise.
int search(HashTable *ht, const char *key) {
    unsigned int index = hashFunction(key); // Get the hash index for the key to
search
    KeyValuePair *current = ht->buckets[index]; // Start traversing the linked list at
this bucket
```

```
    // Traverse the linked list at the calculated bucket index.
    while (current != NULL) {
        // Compare the current node's key with the key being searched for.
        // strcmp returns 0 if the strings are identical (case-sensitive).
        if (strcmp(current->key, key) == 0) {
            return 1; // Key found in the dictionary
        }
        current = current->next; // Move to the next node in the linked list
    }
    return 0; // Key not found after traversing the entire linked list
}

// Frees all dynamically allocated memory used by the hash table.
// This prevents memory leaks by iterating through each bucket and freeing all
nodes
// in their respective linked lists.
void freeHashTable(HashTable *ht) {
    for (int i = 0; i < TABLE_SIZE; i++) {
        KeyValuePair *current = ht->buckets[i];
        while (current != NULL) {
            KeyValuePair *temp = current; // Store the current node
            current = current->next;     // Move to the next node before freeing current
            free(temp);                  // Free the memory of the stored node
        }
        ht->buckets[i] = NULL; // After freeing all nodes, set the bucket pointer to
NULL
    }
}

// --- Main Function ---
// This is the entry point of the program. It orchestrates the entire process:
// 1. Initializes the hash table.
// 2. Reads the number of fruits (N).
// 3. Reads N fruit names and their scores, inserting them into the hash table.
// 4. Reads the name of the fruit to search for.
// 5. Calls the search function and prints the appropriate output based on the
result.
// 6. Frees all dynamically allocated memory before exiting.
int main() {
    HashTable fruitContest;      // Declare a HashTable variable
    initHashTable(&fruitContest); // Initialize the hash table
```

```
    int N;
    scanf("%d", &N); // Read the number of fruits (N) from input

    // Loop N times to read each fruit's name and score and insert it into the hash
table.
    for (int i = 0; i < N; i++) {
        char fruitName[MAX_FRUIT_NAME_LENGTH]; // Buffer to store the fruit
name
        int score;                        // Variable to store the fruit's score
        scanf("%s %d", fruitName, &score);     // Read fruit name (string) and score
(integer)
        insert(&fruitContest, fruitName, score); // Insert the key-value pair into the
hash table
    }

    // Read the name of the fruit that the user wants to search for.
    char searchFruitName[MAX_FRUIT_NAME_LENGTH];
    scanf("%s", searchFruitName);

    // Call the search function to check if the target fruit exists in the hash table.
    if (search(&fruitContest, searchFruitName)) {
        // If the search function returns 1 (true), the key exists.
        printf("Key \"%s\" exists in the dictionary.\n", searchFruitName);
    } else {
        // If the search function returns 0 (false), the key does not exist.
        printf("Key \"%s\" does not exist in the dictionary.\n", searchFruitName);
    }

    // Free all dynamically allocated memory to prevent memory leaks.
    freeHashTable(&fruitContest);

    return 0; // Indicate successful program execution
}
```

Status : Correct ***Marks : 10/10***