Double Hashing in C

Introduction

This report provides an in-depth analysis of a C program that implements a double hashing technique for handling collisions in a hash table. The program uses a primary hash function and a secondary hash function to insert keys into a hash table, ensuring efficient data retrieval and minimal collision rates.

Problem Statement

In hashing, collisions occur when two keys hash to the same index in a hash table. To handle collisions, various methods can be employed, one of which is double hashing. This technique uses two hash functions: one to determine the initial position and another to determine the step size for probing. The goal is to minimize collisions and distribute keys uniformly across the table.

Solution Overview

The solution presented in the C code utilizes double hashing to resolve collisions. The primary hash function (`h1`) and a secondary hash function (using the reverse of the key) work together to place each key in an appropriate slot in the hash table. If a collision occurs, the secondary hash function determines the next slot to check, ensuring that all slots can be probed in a cyclic manner.

Code Structure

The main components of the C code are as follows:

1. `h1`: A primary hash function that computes the initial hash index.

2. `reverse`: A function that calculates the reverse of a key, used in the secondary hash function.

3. `doubleHashing`: The main function that inserts keys into the hash table using double hashing.

Detailed Code Analysis

1. `h1` Function

The `h1` function computes the initial index in the hash table for a given key using the formula:

```
int x = (key + 7) * (key + 7);
x = x / 16;
x = x + key;
x = x % M;
```

This function ensures that the index is within the bounds of the hash table size (M).

2. `reverse` Function

The `reverse` function calculates the reverse of the digits of the key. This reversed number is used as a secondary hash value to determine the step size for probing. This function helps in generating a unique step size for each key, thereby reducing the chances of clustering.

3. `doubleHashing` Function

The `doubleHashing` function is responsible for inserting keys into the hash table. It first calculates the initial index using `h1` and then determines the step size using the `reverse` function. If a collision occurs (i.e., the slot is already occupied), the function probes the next slot by adding the step size to the current index (modulo M). This process continues until an empty slot is found.

Testing and Results

The provided code includes a test case with a set of keys {43, 23, 1, 0, 15, 31, 4, 7, 11, 3}. The resulting hash table, after applying the double hashing technique, shows an efficient distribution of keys with minimal collisions. The table is printed at the end of the program to verify the correct placement of each key.

Conclusion

The double hashing technique implemented in this C code demonstrates an effective method for handling collisions in a hash table. By using two hash functions, the solution ensures a more uniform distribution of keys and reduces the likelihood of clustering. This report highlights the critical components and functionality of the code, providing a comprehensive understanding of its operation.

Appendix: Full Source Code

```c
#include <stdio.h>
#define M 11


int h1(int key) {
    int x = (key + 7) * (key + 7);

    x = x / 16;

    x = x + key;

    x = x % M;

    return x;

}



int reverse(int key) {

    int rev = 0;

    while (key > 0) {

        rev = rev * 10 + key % 10;

        key = key / 10;

    }

    return rev;

}



void doubleHashing(int keys[], int size) {

    int hashTable[M];

    for (int i = 0; i < M; i++) hashTable[i] = -1;



    for (int i = 0; i < size; i++) {

        int key = keys[i];

        int home = h1(key);

        int step = reverse(key) % M;
        int slot = home;

        while (hashTable[slot] != -1) {

            slot = (slot + step) % M;

        }

        hashTable[slot] = key;

    }


    printf("Final Hash Table (Double Hashing):\n");

    for (int i = 0; i < M; i++) {

        if (hashTable[i] != -1)

            printf("Slot %d: %d\n", i, hashTable[i]);

        else

            printf("Slot %d: \n", i);

    }

}


int main() {

    int keys[] = {43, 23, 1, 0, 15, 31, 4, 7, 11, 3};

    int size = sizeof(keys) / sizeof(keys[0]);

    doubleHashing(keys, size);

    return 0;

}
```