HASHING

Write a C program to create a hash table and perform collision resolution using the following techniques.

(i) Open addressing .

(ii) Closed Addressing .

(iii) Rehashing.

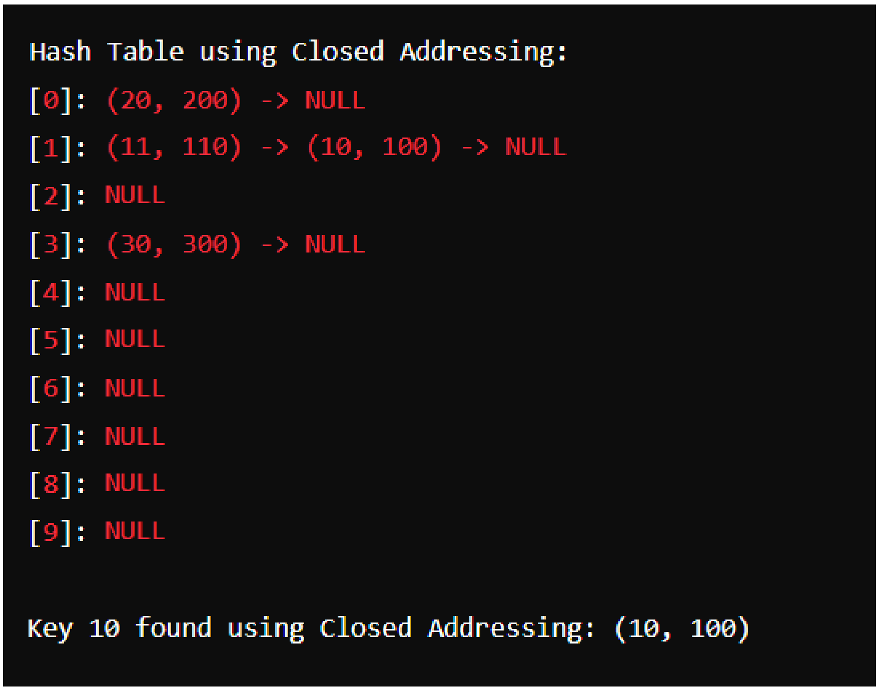
PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#define SIZE 10  // Size of the hash table

// Structure for a node in the hash table (used for chaining)
struct Node {
   int key;
   int data;
   struct Node* next;
};

// Structure for the hash table
struct HashTable {
   struct Node** array;  // Array of linked list (for chaining)
   int size;            // Size of the hash table
};

// Function to create a new node
struct Node* createNode(int key, int data) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   if (newNode == NULL) {
      printf("Memory allocation failed\n");
      exit(EXIT_FAILURE);
   }
   newNode->key = key;
```

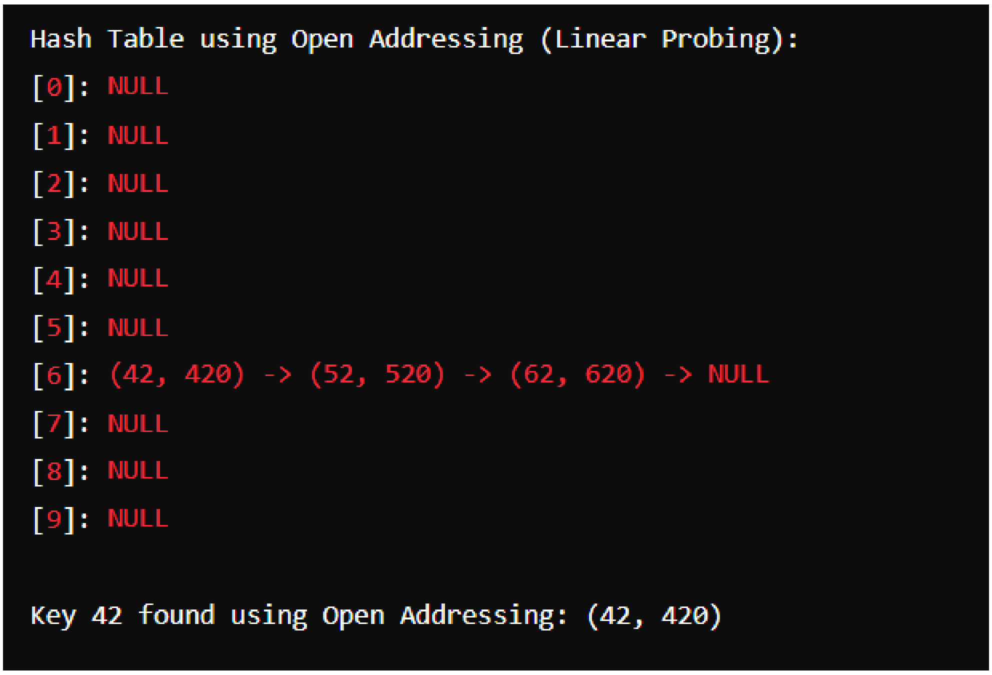
```
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

// Function to create a new hash table
struct HashTable* createHashTable(int size) {
    struct HashTable* hashTable = (struct HashTable*)malloc(sizeof(struct HashTable));
    if (hashTable == NULL) {
        printf("Memory allocation failed\n");
        exit(EXIT_FAILURE);
    }
    hashTable->size = size;
    hashTable->array = (struct Node**)malloc(size * sizeof(struct Node*));
    if (hashTable->array == NULL) {
        printf("Memory allocation failed\n");
        exit(EXIT_FAILURE);
    }
    // Initialize each slot to NULL (no collision yet)
    for (int i = 0; i < size; ++i)
        hashTable->array[i] = NULL;
    return hashTable;
}

// Hash function (simple modulo hashing)
int hashFunction(int key, int size) {
    return key % size;
}

// Function to insert key-value pair into hash table using chaining (closed addressing)
void insertClosedAddressing(struct HashTable* hashTable, int key, int data) {
    int index = hashFunction(key, hashTable->size);

    // Create a new node
    struct Node* newNode = createNode(key, data);

    // Insert the node into the linked list at index
    if (hashTable->array[index] == NULL) {
        hashTable->array[index] = newNode;
    } else {
        // Handle collision by chaining (insert at the beginning of the linked list)
        newNode->next = hashTable->array[index];
        hashTable->array[index] = newNode;
    }
}

// Function to search for a key in the hash table using chaining (closed addressing)
struct Node* searchClosedAddressing(struct HashTable* hashTable, int key) {
    int index = hashFunction(key, hashTable->size);

    // Traverse the linked list at index to find the key
    struct Node* current = hashTable->array[index];
    while (current != NULL) {
        if (current->key == key)
            return current;  // Found the key
        current = current->next;
    }
    return NULL;  // Key not found
}

// Function to print the hash table (chaining)
void displayClosedAddressing(struct HashTable* hashTable) {
    printf("Hash Table using Closed Addressing:\n");
    for (int i = 0; i < hashTable->size; ++i) {
        printf("[%d]: ", i);
        struct Node* current = hashTable->array[i];
        while (current != NULL) {
            printf("(%d, %d) -> ", current->key, current->data);
            current = current->next;
        }
        printf("NULL\n");
    }
    printf("\n");
}

// Function to insert key-value pair into hash table using open addressing (linear probing)
void insertOpenAddressing(struct HashTable* hashTable, int key, int data) {
    int index = hashFunction(key, hashTable->size);

    // Find the next free slot using linear probing
    while (hashTable->array[index] != NULL) {
        index = (index + 1) % hashTable->size;  // Wrap around if needed
    }

    // Insert the node into the found free slot
    hashTable->array[index] = createNode(key, data);
}

// Function to search for a key in the hash table using open addressing (linear probing)
struct Node* searchOpenAddressing(struct HashTable* hashTable, int key) {
    int index = hashFunction(key, hashTable->size);
    int originalIndex = index;

    // Traverse the array to find the key or an empty slot
    while (hashTable->array[index] != NULL) {
        if (hashTable->array[index]->key == key)
            return hashTable->array[index];  // Found the key
        index = (index + 1) % hashTable->size;  // Move to the next slot
        if (index == originalIndex)
            break;  // Reached back to the starting point, table is full
    }
    return NULL;  // Key not found
```

```
}

// Function to print the hash table (open addressing)
void displayOpenAddressing(struct HashTable* hashTable) {
    printf("Hash Table using Open Addressing (Linear Probing):\n");
    for (int i = 0; i < hashTable->size; ++i) {
        if (hashTable->array[i] != NULL)
            printf("[%d]: (%d, %d)\n", i, hashTable->array[i]->key, hashTable->array[i]->data);
        else
            printf("[%d]: NULL\n", i);
    }
    printf("\n");
}

// Function to rehash the hash table (increase size and reinsert all elements)
struct HashTable* rehash(struct HashTable* hashTable) {
    int newSize = hashTable->size * 2;  // Double the size for rehashing
    struct HashTable* newHashTable = createHashTable(newSize);

    // Reinsert all elements from the old hash table into the new hash table
    for (int i = 0; i < hashTable->size; ++i) {
        struct Node* current = hashTable->array[i];
        while (current != NULL) {
            insertClosedAddressing(newHashTable, current->key, current->data);
            current = current->next;
        }
    }

    // Free the memory allocated for the old hash table
    for (int i = 0; i < hashTable->size; ++i) {
        struct Node* current = hashTable->array[i];
        while (current != NULL) {
            struct Node* temp = current;
            current = current->next;
            free(temp);
        }
    }
    free(hashTable->array);
    free(hashTable);

    return newHashTable;
}

// Function to print the hash table after rehashing
void displayRehashing(struct HashTable* hashTable) {
    printf("Hash Table after Rehashing:\n");
    for (int i = 0; i < hashTable->size; ++i) {
        printf("[%d]: ", i);
        struct Node* current = hashTable->array[i];
        while (current != NULL) {
            printf("(%d, %d) -> ", current->key, current->data);
            current = current->next;
        }
        printf("NULL\n");
    }
    printf("\n");
}

// Main function to test the hash table operations
int main() {
    struct HashTable* hashTable = createHashTable(SIZE);

    // Insert using Closed Addressing (Chaining)
    insertClosedAddressing(hashTable, 10, 100);
    insertClosedAddressing(hashTable, 20, 200);
    insertClosedAddressing(hashTable, 30, 300);
    insertClosedAddressing(hashTable, 11, 110);  // Collision with key 10
    // Display the hash table using Closed Addressing
    displayClosedAddressing(hashTable);

    // Search using Closed Addressing
    int keyToSearch = 10;
    struct Node* resultClosed = searchClosedAddressing(hashTable, keyToSearch);
    if (resultClosed != NULL)
        printf("Key %d found using Closed Addressing: (%d, %d)\n\n", keyToSearch, resultClosed->key, resultClosed->data);
    else
        printf("Key %d not found using Closed Addressing\n\n", keyToSearch);

    // Insert using Open Addressing (Linear Probing)
    insertOpenAddressing(hashTable, 42, 420);
    insertOpenAddressing(hashTable, 52, 520);
    insertOpenAddressing(hashTable, 62, 620);
    insertOpenAddressing(hashTable, 42, 420);  // Collision with key 42

    // Display the hash table using Open Addressing
    displayOpenAddressing(hashTable);

    // Search using Open Addressing
    keyToSearch = 42;
    struct Node* resultOpen = searchOpenAddressing(hashTable, keyToSearch);
    if (resultOpen != NULL)
        printf("Key %d found using Open Addressing: (%d, %d)\n\n", keyToSearch, resultOpen->key, resultOpen->data);
    else
        printf("Key %d not found using Open Addressing\n\n", keyToSearch);

    // Perform Rehashing
    printf("Performing Rehashing...\n");
    struct HashTable* newHashTable = rehash(hashTable);

    // Display the hash table after Rehashing
    displayRehashing(newHashTable);

    // Free memory
    for (int i = 0; i < newHashTable->size; ++i) {
        struct Node* current = newHashTable->array[i];
        while (current != NULL) {
            struct Node* temp = current;
            current = current->next;
            free(temp);
        }
    }
    free(newHashTable->array);
    free(newHashTable);

    return 0;
}
```

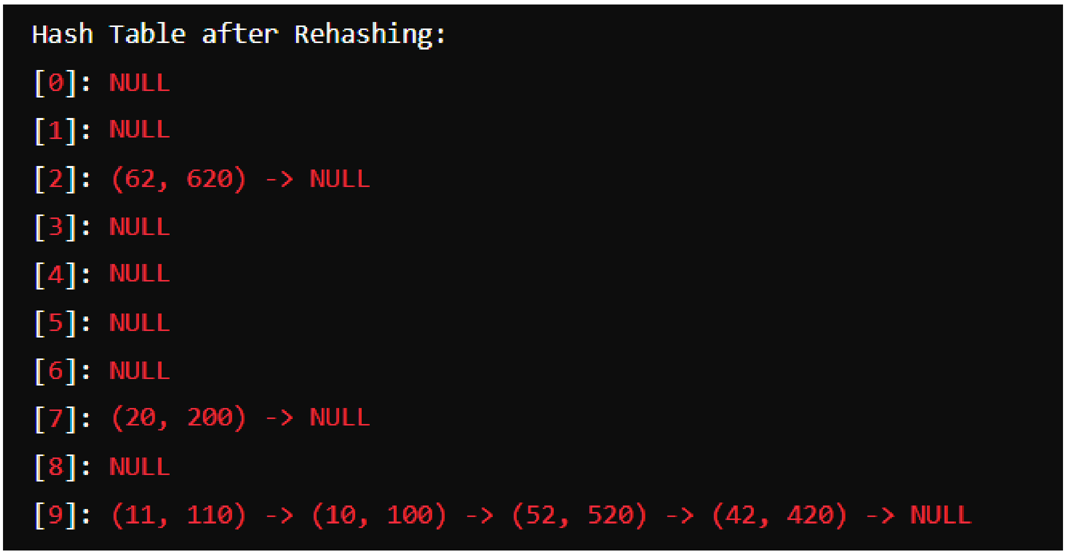
OUTPUT:



```
Hash Table using Closed Addressing:
[0]: (20, 200) -> NULL
[1]: (11, 110) -> (10, 100) -> NULL
[2]: NULL
[3]: (30, 300) -> NULL
[4]: NULL
[5]: NULL
[6]: NULL
[7]: NULL
[8]: NULL
[9]: NULL

Key 10 found using Closed Addressing: (10, 100)
```



```
Hash Table using Open Addressing (Linear Probing):
[0]: NULL
[1]: NULL
[2]: NULL
[3]: NULL
[4]: NULL
[5]: NULL
[6]: (42, 420) -> (52, 520) -> (62, 620) -> NULL
[7]: NULL
[8]: NULL
[9]: NULL

Key 42 found using Open Addressing: (42, 420)
```



```
Hash Table after Rehashing:
[0]: NULL
[1]: NULL
[2]: (62, 620) -> NULL
[3]: NULL
[4]: NULL
[5]: NULL
[6]: NULL
[7]: (20, 200) -> NULL
[8]: NULL
[9]: (11, 110) -> (10, 100) -> (52, 520) -> (42, 420) -> NULL
```