16(a). Implementation of Hashing (Open Addressing)

Program :

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

#define TABLE_SIZE 10

int hash(int key) {
    return key % TABLE_SIZE;
}

void insert(int table[], int key) {
    int index = hash(key);
    int i = 0;
    while (table[(index + i) % TABLE_SIZE] != -1) {
        i++;
    }
    table[(index + i) % TABLE_SIZE] = key;
}

void display(int table[]) {
    for (int i = 0; i < TABLE_SIZE; i++) {
        if (table[i] != -1)
            printf("%d -> %d\n", i, table[i]);
        else
            printf("%d -> \n", i);
    }
}

int main() {
    int table[TABLE_SIZE];
    for (int i = 0; i < TABLE_SIZE; i++)
        table[i] = -1;

    insert(table, 10);
    insert(table, 20);
    insert(table, 30);
    insert(table, 25);

    printf("Open Addressing Hash Table:\n");
    display(table);
    return 0;
}
```

Output :

```
Open Addressing Hash Table:
0 -> 10
1 -> 20
2 -> 30
3 ->
4 ->
5 -> 25
6 ->
7 ->
8 ->
9 ->
```

16(b). Implementation of Hashing (Closed Addressing)

Program :

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

#define TABLE_SIZE 10

typedef struct Node {
    int key;
    struct Node* next;
} Node;

Node* createNode(int key) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->key = key;
    newNode->next = NULL;
    return newNode;
}

int hash(int key) {
    return key % TABLE_SIZE;
}

void insert(Node* table[], int key) {
    int index = hash(key);
    Node* newNode = createNode(key);
    newNode->next = table[index];
    table[index] = newNode;
}

void display(Node* table[]) {
    for (int i = 0; i < TABLE_SIZE; i++) {
        printf("%d -> ", i);
        Node* temp = table[i];
        while (temp) {
            printf("%d -> ", temp->key);
            temp = temp->next;
        }
        printf("NULL\n");
    }
}

int main() {
    Node* table[TABLE_SIZE] = { NULL };

    insert(table, 10);
    insert(table, 20);
    insert(table, 30);
    insert(table, 25);

    printf("Closed Addressing (Chaining) Hash Table:\n");
    display(table);
    return 0;
}
```

Output :

```
Closed Addressing (Chaining) Hash Table:
0 -> 30 -> 20 -> 10 -> NULL
1 -> NULL
2 -> NULL
3 -> NULL
4 -> NULL
5 -> 25 -> NULL
6 -> NULL
7 -> NULL
8 -> NULL
9 -> NULL
```

16(c). Implementation of Hashing (Rehashing)

Program :

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

#define INITIAL_TABLE_SIZE 10
#define LOAD_FACTOR_THRESHOLD 0.7

typedef struct {
    int* table;
    int size;
    int count;
} HashTable;

void insert(HashTable*, int);

int hash(int key, int size) {
    return key % size;
}

HashTable* createHashTable(int size) {
    HashTable* ht = (HashTable*)malloc(sizeof(HashTable));
    ht->size = size;
    ht->count = 0;
    ht->table = (int*)malloc(size * sizeof(int));
    for (int i = 0; i < size; i++)
        ht->table[i] = -1;
    return ht;
}

void rehash(HashTable* ht) {
    int oldSize = ht->size;
    int* oldTable = ht->table;

    ht->size *= 2;
    ht->count = 0;
    ht->table = (int*)malloc(ht->size * sizeof(int));
    for (int i = 0; i < ht->size; i++)
        ht->table[i] = -1;

    for (int i = 0; i < oldSize; i++) {
        if (oldTable[i] != -1)
            insert(ht, oldTable[i]);
    }
    free(oldTable);
}

void insert(HashTable* ht, int key) {
    if ((float)ht->count / ht->size > LOAD_FACTOR_THRESHOLD)
        rehash(ht);

    int index = hash(key, ht->size);
    int i = 0;
    while (ht->table[(index + i) % ht->size] != -1) {
        i++;
    }
```

```c
    ht->table[(index + i) % ht->size] = key;
    ht->count++;
}

void display(HashTable* ht) {
    for (int i = 0; i < ht->size; i++) {
        if (ht->table[i] != -1)
            printf("%d -> %d\n", i, ht->table[i]);
        else
            printf("%d -> \n", i);
    }
}

int main() {
    HashTable* ht = createHashTable(INITIAL_TABLE_SIZE);

    insert(ht, 10);
    insert(ht, 20);
    insert(ht, 30);
    insert(ht, 25);
    insert(ht, 15);
    insert(ht, 35);
    insert(ht, 40);

    printf("Rehashing Hash Table:\n");
    display(ht);

    free(ht->table);
    free(ht);
    return 0;
}
```

Output :

```
Rehashing Hash Table:
0 -> 10
1 -> 20
2 -> 30
3 -> 40
4 ->
5 -> 25
6 -> 15
7 -> 35
8 ->
9 ->
```