Ex 16: Hashing

Name: Rowin.U Reg No.:231801141

OPEN ADDRESING:

#include <stdio.h>

#define max 10

int a[11] = { 10, 14, 19, 26, 27, 31, 33, 35, 42, 44, 0 };

int b[10];

void merging(int low, int mid, int high) {

int l1, l2, i;

for(l1 = low, l2 = mid + 1, i = low; l1 <= mid && l2 <= high; i++) {

if(a[l1] <= a[l2])

b[i] = a[l1++];

else

b[i] = a[l2++];

}

while(l1 <= mid)

b[i++] = a[l1++];

while(l2 <= high)

b[i++] = a[l2++];

for(i = low; i <= high; i++)

a[i] = b[i];

}

void sort(int low, int high) {

int mid;

if(low < high) {

mid = (low + high) / 2;

sort(low, mid);

sort(mid+1, high);

merging(low, mid, high);

} else {

return;

}

}

int main() {

int i;

printf("List before sorting\n");

for(i = 0; i <= max; i++)

printf("%d ", a[i]);

sort(0, max);

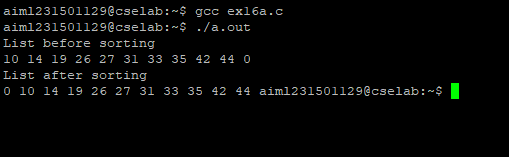
printf("\nList after sorting\n");

for(i = 0; i <= max; i++)

printf("%d ", a[i]);

}

OUTPUT:



CLOSED ADDRESING:

PROGRAM:

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

typedef struct Node {

int key;

int value;

struct Node\* next;

} Node;

typedef struct HashTable {

int size;

Node\*\* table;

} HashTable;

Node\* createNode(int key, int value) {

Node\* newNode = (Node\*)malloc(sizeof(Node));

newNode->key = key;

newNode->value = value;

newNode->next = NULL;

return newNode;

}

HashTable\* createTable(int size) {

HashTable\* newTable = (HashTable\*)malloc(sizeof(HashTable));

newTable->size = size;

newTable->table = (Node\*\*)malloc(sizeof(Node\*) \* size);

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

newTable->table[i] = NULL;

}

return newTable;

}

int hashFunction(int key, int size) {

return key % size;

}

void insert(HashTable\* hashTable, int key, int value) {

int hashIndex = hashFunction(key, hashTable->size);

Node\* newNode = createNode(key, value);

newNode->next = hashTable->table[hashIndex];

hashTable->table[hashIndex] = newNode;

}

int search(HashTable\* hashTable, int key) {

int hashIndex = hashFunction(key, hashTable->size);

Node\* current = hashTable->table[hashIndex];

while (current != NULL) {

if (current->key == key) {

return current->value;

}

current = current->next;

}

return -1;

}

void delete(HashTable\* hashTable, int key) {

int hashIndex = hashFunction(key, hashTable->size);

Node\* current = hashTable->table[hashIndex];

Node\* prev = NULL;

while (current != NULL && current->key != key) {

prev = current;

current = current->next;

}

if (current == NULL) {

return;

}

if (prev == NULL) {

hashTable->table[hashIndex] = current->next;

} else {

prev->next = current->next;

}

free(current);

}

void freeTable(HashTable\* hashTable) {

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

Node\* current = hashTable->table[i];

while (current != NULL) {

Node\* temp = current;

current = current->next;

free(temp);

}

}

free(hashTable->table);

free(hashTable);

}

int main() {

HashTable\* hashTable = createTable(10);

insert(hashTable, 1, 10);

insert(hashTable, 2, 20);

insert(hashTable, 12, 30);

printf("Value for key 1: %d\n", search(hashTable, 1));

printf("Value for key 2: %d\n", search(hashTable, 2));

printf("Value for key 12: %d\n", search(hashTable, 12));

printf("Value for key 3: %d\n", search(hashTable, 3));

delete(hashTable, 2);

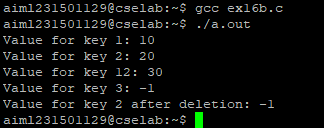
printf("Value for key 2 after deletion: %d\n", search(hashTable, 2));

freeTable(hashTable);

return 0;

}

OUTPUT:



REHASHING:

PROGRAM:

#include <stdio.h>

#include <stdlib.h>

typedef struct Node {

int key;

int value;

struct Node\* next;

} Node;

typedef struct HashTable {

int size;

int count;

Node\*\* table;

} HashTable;

Node\* createNode(int key, int value) {

Node\* newNode = (Node\*)malloc(sizeof(Node));

newNode->key = key;

newNode->value = value;

newNode->next = NULL;

return newNode;

}

HashTable\* createTable(int size) {

HashTable\* newTable = (HashTable\*)malloc(sizeof(HashTable));

newTable->size = size;

newTable->count = 0;

newTable->table = (Node\*\*)malloc(sizeof(Node\*) \* size);

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

newTable->table[i] = NULL;

}

return newTable;

}

int hashFunction(int key, int size) {

return key % size;

}

void insert(HashTable\* hashTable, int key, int value);

void rehash(HashTable\* hashTable) {

int oldSize = hashTable->size;

Node\*\* oldTable = hashTable->table;

int newSize = oldSize \* 2;

hashTable->table = (Node\*\*)malloc(sizeof(Node\*) \* newSize);

hashTable->size = newSize;

hashTable->count = 0;

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

hashTable->table[i] = NULL;

}

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

Node\* current = oldTable[i];

while (current != NULL) {

insert(hashTable, current->key, current->value);

Node\* temp = current;

current = current->next;

free(temp);

}

}

free(oldTable);

}

void insert(HashTable\* hashTable, int key, int value) {

if ((float)hashTable->count / hashTable->size >= 0.75) {

rehash(hashTable);

}

int hashIndex = hashFunction(key, hashTable->size);

Node\* newNode = createNode(key, value);

newNode->next = hashTable->table[hashIndex];

hashTable->table[hashIndex] = newNode;

hashTable->count++;

}

int search(HashTable\* hashTable, int key) {

int hashIndex = hashFunction(key, hashTable->size);

Node\* current = hashTable->table[hashIndex];

while (current != NULL) {

if (current->key == key) {

return current->value;

}

current = current->next;

}

return -1;

}

void delete(HashTable\* hashTable, int key) {

int hashIndex = hashFunction(key, hashTable->size);

Node\* current = hashTable->table[hashIndex];

Node\* prev = NULL;

while (current != NULL && current->key != key) {

prev = current;

current = current->next;

}

if (current == NULL) {

return;

}

if (prev == NULL) {

hashTable->table[hashIndex] = current->next;

} else {

prev->next = current->next;

}

free(current);

hashTable->count--;

}

void freeTable(HashTable\* hashTable) {

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

Node\* current = hashTable->table[i];

while (current != NULL) {

Node\* temp = current;

current = current->next;

free(temp);

}

}

free(hashTable->table);

free(hashTable);

}

int main() {

HashTable\* hashTable = createTable(5);

insert(hashTable, 1, 10);

insert(hashTable, 2, 20);

insert(hashTable, 3, 30);

insert(hashTable, 4, 40);

insert(hashTable, 5, 50);

insert(hashTable, 6, 60);

printf("Value for key 1: %d\n", search(hashTable, 1));

printf("Value for key 2: %d\n", search(hashTable, 2));

printf("Value for key 3: %d\n", search(hashTable, 3));

printf("Value for key 4: %d\n", search(hashTable, 4));

printf("Value for key 5: %d\n", search(hashTable, 5));

printf("Value for key 6: %d\n", search(hashTable, 6));

delete(hashTable, 3);

printf("Value for key 3 after deletion: %d\n", search(hashTable, 3));

freeTable(hashTable);

return 0;

}

OUTPUT:

