|  |
| --- |
| **SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES** |
| **COMPUTER SCIENCE AND ENGINEERING PROGRAMME** |

**SUB CODE: CSA0392 SUB NAME: Data Structures for Hashing Techniques**

**LIST OF PROGRAMS**

**DATE : 21.08.2024**

**Lab Questions to be practiced with test cases**

1. Write a C program to implement hashing using Separate chaining method.

Answer:

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define TABLE\_SIZE 10

// Define a node for the linked list

typedef struct Node {

int key;

int value;

struct Node\* next;

} Node;

// Define the hash table

typedef struct HashTable {

Node\* table[TABLE\_SIZE];

} HashTable;

// Create a new hash table

HashTable\* createHashTable() {

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

for (int i = 0; i < TABLE\_SIZE; i++) {

hashTable->table[i] = NULL;

}

return hashTable;

}

// Hash function

unsigned int hash(int key) {

return key % TABLE\_SIZE;

}

// Create a new node

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

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

newNode->key = key;

newNode->value = value;

newNode->next = NULL;

return newNode;

}

// Insert a key-value pair into the hash table

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

unsigned int index = hash(key);

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

if (hashTable->table[index] == NULL) {

hashTable->table[index] = newNode;

} else {

Node\* temp = hashTable->table[index];

while (temp->next != NULL && temp->key != key) {

temp = temp->next;

}

if (temp->key == key) {

temp->value = value;

free(newNode); // Key already exists, update the value

} else {

temp->next = newNode;

}

}

}

// Search for a value by key

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

unsigned int index = hash(key);

Node\* temp = hashTable->table[index];

while (temp != NULL) {

if (temp->key == key) {

return temp->value;

}

temp = temp->next;

}

return -1; // Key not found

}

// Delete a key-value pair from the hash table

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

unsigned int index = hash(key);

Node\* temp = hashTable->table[index];

Node\* prev = NULL;

while (temp != NULL) {

if (temp->key == key) {

if (prev == NULL) {

hashTable->table[index] = temp->next;

} else {

prev->next = temp->next;

}

free(temp);

return;

}

prev = temp;

temp = temp->next;

}

}

// Display the hash table

void display(HashTable\* hashTable) {

for (int i = 0; i < TABLE\_SIZE; i++) {

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

printf("Index %d: ", i);

while (temp != NULL) {

printf("(%d, %d) -> ", temp->key, temp->value);

temp = temp->next;

}

printf("NULL\n");

}

}

// Free the memory allocated for the hash table

void freeHashTable(HashTable\* hashTable) {

for (int i = 0; i < TABLE\_SIZE; i++) {

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

while (temp != NULL) {

Node\* next = temp->next;

free(temp);

temp = next;

}

}

free(hashTable);

}

// Main function to demonstrate the hash table operations

int main() {

HashTable\* hashTable = createHashTable();

// Insert some key-value pairs

insert(hashTable, 10, 100);

insert(hashTable, 20, 200);

insert(hashTable, 30, 300);

insert(hashTable, 40, 400);

insert(hashTable, 50, 500);

// Display the hash table

printf("Hash Table:\n");

display(hashTable);

// Search for a value

int key = 20;

int value = search(hashTable, key);

if (value != -1) {

printf("Value for key %d: %d\n", key, value);

} else {

printf("Key %d not found\n", key);

}

// Delete a key-value pair

delete(hashTable, 30);

printf("Hash Table after deleting key 30:\n");

display(hashTable);

// Free memory

freeHashTable(hashTable);

return 0;

}

1. Write a C program to implement hashing using Linear Probing method.

Answer:

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#define TABLE\_SIZE 10

// Define a structure for hash table entries

typedef struct {

int key;

int value;

bool isOccupied;

} HashEntry;

// Define the hash table

typedef struct {

HashEntry\* table[TABLE\_SIZE];

} HashTable;

// Create a new hash table

HashTable\* createHashTable() {

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

for (int i = 0; i < TABLE\_SIZE; i++) {

hashTable->table[i] = NULL;

}

return hashTable;

}

// Hash function

unsigned int hash(int key) {

return key % TABLE\_SIZE;

}

// Create a new hash entry

HashEntry\* createEntry(int key, int value) {

HashEntry\* entry = (HashEntry\*)malloc(sizeof(HashEntry));

entry->key = key;

entry->value = value;

entry->isOccupied = true;

return entry;

}

// Insert a key-value pair into the hash table

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

unsigned int index = hash(key);

unsigned int start = index;

do {

if (hashTable->table[index] == NULL || !hashTable->table[index]->isOccupied) {

if (hashTable->table[index] != NULL) {

free(hashTable->table[index]); // Free previously occupied slot

}

hashTable->table[index] = createEntry(key, value);

return;

} else if (hashTable->table[index]->key == key) {

hashTable->table[index]->value = value; // Update existing key

return;

}

index = (index + 1) % TABLE\_SIZE;

} while (index != start);

}

// Search for a value by key

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

unsigned int index = hash(key);

unsigned int start = index;

while (hashTable->table[index] != NULL) {

if (hashTable->table[index]->isOccupied && hashTable->table[index]->key == key) {

return hashTable->table[index]->value;

}

index = (index + 1) % TABLE\_SIZE;

if (index == start) break;

}

return -1; // Key not found

}

// Delete a key-value pair from the hash table

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

unsigned int index = hash(key);

unsigned int start = index;

while (hashTable->table[index] != NULL) {

if (hashTable->table[index]->isOccupied && hashTable->table[index]->key == key) {

hashTable->table[index]->isOccupied = false;

return;

}

index = (index + 1) % TABLE\_SIZE;

if (index == start) break;

}

}

// Display the hash table

void display(HashTable\* hashTable) {

for (int i = 0; i < TABLE\_SIZE; i++) {

if (hashTable->table[i] != NULL && hashTable->table[i]->isOccupied) {

printf("Index %d: (%d, %d)\n", i, hashTable->table[i]->key, hashTable->table[i]->value);

} else {

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

}

}

}

// Free the memory allocated for the hash table

void freeHashTable(HashTable\* hashTable) {

for (int i = 0; i < TABLE\_SIZE; i++) {

if (hashTable->table[i] != NULL) {

free(hashTable->table[i]);

}

}

free(hashTable);

}

// Main function to demonstrate hash table operations

int main() {

HashTable\* hashTable = createHashTable();

// Insert some key-value pairs

insert(hashTable, 10, 100);

insert(hashTable, 20, 200);

insert(hashTable, 30, 300);

insert(hashTable, 40, 400);

insert(hashTable, 50, 500);

// Display the hash table

printf("Hash Table:\n");

display(hashTable);

// Search for values

int key = 20;

int value = search(hashTable, key);

if (value != -1) {

printf("Value for key %d: %d\n", key, value);

} else {

printf("Key %d not found\n", key);

}

// Delete a key-value pair

delete(hashTable, 30);

printf("Hash Table after deleting key 30:\n");

display(hashTable);

// Free memory

freeHashTable(hashTable);

return 0;

}

1. Write a C program to implement hashing using Quadratic Probing method.

Answer:

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#define TABLE\_SIZE 10

// Define a structure for hash table entries

typedef struct {

int key;

int value;

bool isOccupied;

bool isDeleted;

} HashEntry;

// Define the hash table

typedef struct {

HashEntry\* table[TABLE\_SIZE];

} HashTable;

// Create a new hash table

HashTable\* createHashTable() {

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

for (int i = 0; i < TABLE\_SIZE; i++) {

hashTable->table[i] = NULL;

}

return hashTable;

}

// Hash function

unsigned int hash(int key) {

return key % TABLE\_SIZE;

}

// Create a new hash entry

HashEntry\* createEntry(int key, int value) {

HashEntry\* entry = (HashEntry\*)malloc(sizeof(HashEntry));

entry->key = key;

entry->value = value;

entry->isOccupied = true;

entry->isDeleted = false;

return entry;

}

// Insert a key-value pair into the hash table

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

unsigned int index = hash(key);

unsigned int i = 0;

while (i < TABLE\_SIZE) {

unsigned int newIndex = (index + i \* i) % TABLE\_SIZE;

if (hashTable->table[newIndex] == NULL || hashTable->table[newIndex]->isDeleted) {

if (hashTable->table[newIndex] != NULL) {

free(hashTable->table[newIndex]); // Free previously occupied slot

}

hashTable->table[newIndex] = createEntry(key, value);

return;

} else if (hashTable->table[newIndex]->key == key) {

hashTable->table[newIndex]->value = value; // Update existing key

return;

}

i++;

}

}

// Search for a value by key

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

unsigned int index = hash(key);

unsigned int i = 0;

while (i < TABLE\_SIZE) {

unsigned int newIndex = (index + i \* i) % TABLE\_SIZE;

if (hashTable->table[newIndex] == NULL) {

return -1; // Key not found

}

if (hashTable->table[newIndex]->isOccupied && hashTable->table[newIndex]->key == key) {

return hashTable->table[newIndex]->value;

}

i++;

}

return -1; // Key not found

}

// Delete a key-value pair from the hash table

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

unsigned int index = hash(key);

unsigned int i = 0;

while (i < TABLE\_SIZE) {

unsigned int newIndex = (index + i \* i) % TABLE\_SIZE;

if (hashTable->table[newIndex] == NULL) {

return; // Key not found

}

if (hashTable->table[newIndex]->isOccupied && hashTable->table[newIndex]->key == key) {

hashTable->table[newIndex]->isDeleted = true;

return;

}

i++;

}

}

// Display the hash table

void display(HashTable\* hashTable) {

for (int i = 0; i < TABLE\_SIZE; i++) {

if (hashTable->table[i] != NULL && hashTable->table[i]->isOccupied && !hashTable->table[i]->isDeleted) {

printf("Index %d: (%d, %d)\n", i, hashTable->table[i]->key, hashTable->table[i]->value);

} else {

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

}

}

}

// Free the memory allocated for the hash table

void freeHashTable(HashTable\* hashTable) {

for (int i = 0; i < TABLE\_SIZE; i++) {

if (hashTable->table[i] != NULL) {

free(hashTable->table[i]);

}

}

free(hashTable);

}

// Main function to demonstrate hash table operations

int main() {

HashTable\* hashTable = createHashTable();

// Insert some key-value pairs

insert(hashTable, 10, 100);

insert(hashTable, 20, 200);

insert(hashTable, 30, 300);

insert(hashTable, 40, 400);

insert(hashTable, 50, 500);

// Display the hash table

printf("Hash Table:\n");

display(hashTable);

// Search for values

int key = 20;

int value = search(hashTable, key);

if (value != -1) {

printf("Value for key %d: %d\n", key, value);

} else {

printf("Key %d not found\n", key);

}

// Delete a key-value pair

delete(hashTable, 30);

printf("Hash Table after deleting key 30:\n");

display(hashTable);

// Free memory

freeHashTable(hashTable);

return 0;

}

1. Write a C program to implement hashing using Double hashing method.

Answer:

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#define TABLE\_SIZE 10

// Define a structure for hash table entries

typedef struct {

int key;

int value;

bool isOccupied;

bool isDeleted;

} HashEntry;

// Define the hash table

typedef struct {

HashEntry\* table[TABLE\_SIZE];

} HashTable;

// Create a new hash table

HashTable\* createHashTable() {

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

for (int i = 0; i < TABLE\_SIZE; i++) {

hashTable->table[i] = NULL;

}

return hashTable;

}

// Hash function 1

unsigned int hash1(int key) {

return key % TABLE\_SIZE;

}

// Hash function 2

unsigned int hash2(int key) {

return 1 + (key % (TABLE\_SIZE - 1)); // Must be non-zero

}

// Create a new hash entry

HashEntry\* createEntry(int key, int value) {

HashEntry\* entry = (HashEntry\*)malloc(sizeof(HashEntry));

entry->key = key;

entry->value = value;

entry->isOccupied = true;

entry->isDeleted = false;

return entry;

}

// Insert a key-value pair into the hash table

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

unsigned int index = hash1(key);

unsigned int stepSize = hash2(key);

unsigned int i = 0;

while (i < TABLE\_SIZE) {

unsigned int newIndex = (index + i \* stepSize) % TABLE\_SIZE;

if (hashTable->table[newIndex] == NULL || hashTable->table[newIndex]->isDeleted) {

if (hashTable->table[newIndex] != NULL) {

free(hashTable->table[newIndex]); // Free previously occupied slot

}

hashTable->table[newIndex] = createEntry(key, value);

return;

} else if (hashTable->table[newIndex]->key == key) {

hashTable->table[newIndex]->value = value; // Update existing key

return;

}

i++;

}

}

// Search for a value by key

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

unsigned int index = hash1(key);

unsigned int stepSize = hash2(key);

unsigned int i = 0;

while (i < TABLE\_SIZE) {

unsigned int newIndex = (index + i \* stepSize) % TABLE\_SIZE;

if (hashTable->table[newIndex] == NULL) {

return -1; // Key not found

}

if (hashTable->table[newIndex]->isOccupied && hashTable->table[newIndex]->key == key) {

return hashTable->table[newIndex]->value;

}

i++;

}

return -1; // Key not found

}

// Delete a key-value pair from the hash table

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

unsigned int index = hash1(key);

unsigned int stepSize = hash2(key);

unsigned int i = 0;

while (i < TABLE\_SIZE) {

unsigned int newIndex = (index + i \* stepSize) % TABLE\_SIZE;

if (hashTable->table[newIndex] == NULL) {

return; // Key not found

}

if (hashTable->table[newIndex]->isOccupied && hashTable->table[newIndex]->key == key) {

hashTable->table[newIndex]->isDeleted = true;

return;

}

i++;

}

}

// Display the hash table

void display(HashTable\* hashTable) {

for (int i = 0; i < TABLE\_SIZE; i++) {

if (hashTable->table[i] != NULL && hashTable->table[i]->isOccupied && !hashTable->table[i]->isDeleted) {

printf("Index %d: (%d, %d)\n", i, hashTable->table[i]->key, hashTable->table[i]->value);

} else {

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

}

}

}

// Free the memory allocated for the hash table

void freeHashTable(HashTable\* hashTable) {

for (int i = 0; i < TABLE\_SIZE; i++) {

if (hashTable->table[i] != NULL) {

free(hashTable->table[i]);

}

}

free(hashTable);

}

// Main function to demonstrate hash table operations

int main() {

HashTable\* hashTable = createHashTable();

// Insert some key-value pairs

insert(hashTable, 10, 100);

insert(hashTable, 20, 200);

insert(hashTable, 30, 300);

insert(hashTable, 40, 400);

insert(hashTable, 50, 500);

// Display the hash table

printf("Hash Table:\n");

display(hashTable);

// Search for values

int key = 20;

int value = search(hashTable, key);

if (value != -1) {

printf("Value for key %d: %d\n", key, value);

} else {

printf("Key %d not found\n", key);

}

// Delete a key-value pair

delete(hashTable, 30);

printf("Hash Table after deleting key 30:\n");

display(hashTable);

// Free memory

freeHashTable(hashTable);

return 0;

}