Министерство науки и высшего образования Российской Федерации  
Пензенский государственный университет  
Кафедра вычислительная техника

**ОТЧЕТ**

по лабораторной работе №4

по дисциплине «Трансляторы и компиляторы»

## на тему «Построение таблиц идентификаторов на основе хэш-функций»

Выполнили студенты группы 22ВВП1

Демин М.С.

Беляев Д. И.

Приняли:

Дубинин В.Н.

Карамышева Н.С.

Пенза 2024

**Название**

Построение таблиц идентификаторов на основе хэш-функций

**Цель работы**

Ознакомление с методами построения таблиц идентификаторов. Изучение метода построения таблиц на основе хэш-функций.

**Задание**

На основе заданной хэш-функции рассчитать размер M таблицы адресов. Написать программу на языке C, создающую для данной хэш-функции таблицы идентификаторов. Входной поток идентификаторов следует получать с консоли, либо из текстового файла, используя в качестве разделителя символы перевода строки и возврата каретки. Необходимо также предусмотреть в программе функцию поиска заданного идентификатора.

Результатом выполнения лабораторной работы должен явиться исполняемый файл программы. Варианты заданий – хэш-функция n=F(r):

n = H(r), где H – сумма по модулю 23 первого символа идентификатора и длины идентификатора в байтах;

**Листинг**

**main.c**

#include <stdio.h>

#include "hashTable.h"

int main()

{

HashTable\* ht = create\_table(CAPACITY);

ht\_insert(ht, "1", "First address");

ht\_insert(ht, "2", "Second address");

ht\_insert(ht, "Hel", "Third address");

ht\_insert(ht, "Cau", "Fourth address");

print\_search(ht, "1");

print\_search(ht, "2");

print\_search(ht, "3");

print\_search(ht, "Hel");

print\_search(ht, "Cau"); // Collision!

print\_table(ht);

ht\_delete(ht, "1");

ht\_delete(ht, "Cau");

print\_table(ht);

free\_table(ht);

return 0;

}

**hashTable.c**

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include "hashTable.h"

unsigned long hash\_function(char\* str) {

int len = strlen(str);

int res = (str[0] + len) % 23;

return res % CAPACITY;

}

static LinkedList\* allocate\_list() {

LinkedList\* list = (LinkedList\*)malloc(sizeof(LinkedList));

return list;

}

static LinkedList\* linkedlist\_insert(LinkedList\* list, Ht\_item\* item) {

if (!list) {

LinkedList\* head = allocate\_list();

head->item = item;

head->next = NULL;

list = head;

return list;

}

else if (list->next == NULL) {

LinkedList\* node = allocate\_list();

node->item = item;

node->next = NULL;

list->next = node;

return list;

}

LinkedList\* temp = list;

while (temp->next->next) {

temp = temp->next;

}

LinkedList\* node = allocate\_list();

node->item = item;

node->next = NULL;

temp->next = node;

return list;

}

static Ht\_item\* linkedlist\_remove(LinkedList\* list) {

// Removes the head from the linked list

// and returns the item of the popped element

if (!list)

return NULL;

if (!list->next)

return NULL;

LinkedList\* node = list->next;

LinkedList\* temp = list;

temp->next = NULL;

list = node;

Ht\_item\* it = NULL;

memcpy(temp->item, it, sizeof(Ht\_item));

free(temp->item->key);

free(temp->item->value);

free(temp->item);

free(temp);

return it;

}

static void free\_linkedlist(LinkedList\* list) {

LinkedList\* temp = list;

while (list) {

temp = list;

list = list->next;

free(temp->item->key);

free(temp->item->value);

free(temp->item);

free(temp);

}

}

static LinkedList\*\* create\_overflow\_buckets(HashTable\* table) {

// Create the overflow buckets; an array of linkedlists

LinkedList\*\* buckets = (LinkedList\*\*)calloc(table->size, sizeof(LinkedList\*));

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

buckets[i] = NULL;

return buckets;

}

static void free\_overflow\_buckets(HashTable\* table) {

// Free all the overflow bucket lists

LinkedList\*\* buckets = table->overflow\_buckets;

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

free\_linkedlist(buckets[i]);

free(buckets);

}

Ht\_item\* create\_item(char\* key, char\* value) {

// Creates a pointer to a new hash table item

Ht\_item\* item = (Ht\_item\*)malloc(sizeof(Ht\_item));

item->key = (char\*)malloc(strlen(key) + 1);

item->value = (char\*)malloc(strlen(value) + 1);

strcpy(item->key, key);

strcpy(item->value, value);

return item;

}

HashTable\* create\_table(int size) {

// Creates a new HashTable

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

table->size = size;

table->count = 0;

table->items = (Ht\_item\*\*)calloc(table->size, sizeof(Ht\_item\*));

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

table->items[i] = NULL;

table->overflow\_buckets = create\_overflow\_buckets(table);

return table;

}

void free\_item(Ht\_item\* item) {

// Frees an item

free(item->key);

free(item->value);

free(item);

}

void free\_table(HashTable\* table) {

// Frees the table

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

Ht\_item\* item = table->items[i];

if (item != NULL)

free\_item(item);

}

free\_overflow\_buckets(table);

free(table->items);

free(table);

}

void handle\_collision(HashTable\* table, unsigned long index, Ht\_item\* item) {

LinkedList\* head = table->overflow\_buckets[index];

if (head == NULL) {

// We need to create the list

head = allocate\_list();

head->item = item;

head->next = NULL;

table->overflow\_buckets[index] = head;

return;

}

else {

// Insert to the list

table->overflow\_buckets[index] = linkedlist\_insert(head, item);

return;

}

}

void ht\_insert(HashTable\* table, char\* key, char\* value) {

// Create the item

Ht\_item\* item = create\_item(key, value);

// Compute the index

unsigned long index = hash\_function(key);

Ht\_item\* current\_item = table->items[index];

if (current\_item == NULL) {

// Key does not exist.

if (table->count == table->size) {

// Hash Table Full

printf("Insert Error: Hash Table is full\n");

// Remove the create item

free\_item(item);

return;

}

// Insert directly

table->items[index] = item;

table->count++;

} else {

// Scenario 1: We only need to update value

if (strcmp(current\_item->key, key) == 0) {

strcpy(table->items[index]->value, value);

return;

} else {

// Scenario 2: Collision

handle\_collision(table, index, item);

return;

}

}

}

char\* ht\_search(HashTable\* table, char\* key) {

// Searches the key in the hashtable

// and returns NULL if it doesn't exist

int index = hash\_function(key);

Ht\_item\* item = table->items[index];

LinkedList\* head = table->overflow\_buckets[index];

// Ensure that we move to items which are not NULL

while (item != NULL) {

if (strcmp(item->key, key) == 0)

return item->value;

if (head == NULL)

return NULL;

item = head->item;

head = head->next;

}

return NULL;

}

void ht\_delete(HashTable\* table, char\* key) {

// Deletes an item from the table

int index = hash\_function(key);

Ht\_item\* item = table->items[index];

LinkedList\* head = table->overflow\_buckets[index];

if (item == NULL) {

// Does not exist. Return

return;

} else {

if (head == NULL && strcmp(item->key, key) == 0) {

// No collision chain. Remove the item

// and set table index to NULL

table->items[index] = NULL;

free\_item(item);

table->count--;

return;

} else if (head != NULL) {

// Collision Chain exists

if (strcmp(item->key, key) == 0) {

// Remove this item and set the head of the list

// as the new item

free\_item(item);

LinkedList\* node = head;

head = head->next;

node->next = NULL;

table->items[index] = create\_item(node->item->key, node->item->value);

free\_linkedlist(node);

table->overflow\_buckets[index] = head;

return;

}

LinkedList\* curr = head;

LinkedList\* prev = NULL;

while (curr) {

if (strcmp(curr->item->key, key) == 0) {

if (prev == NULL) {

// First element of the chain. Remove the chain

free\_linkedlist(head);

table->overflow\_buckets[index] = NULL;

return;

}

else {

// This is somewhere in the chain

prev->next = curr->next;

curr->next = NULL;

free\_linkedlist(curr);

table->overflow\_buckets[index] = head;

return;

}

}

curr = curr->next;

prev = curr;

}

}

}

}

void print\_search(HashTable\* table, char\* key) {

char\* val;

if ((val = ht\_search(table, key)) == NULL) {

printf("%s does not exist\n", key);

return;

}

else {

printf("Key:%s, Value:%s\n", key, val);

}

}

void print\_table(HashTable\* table) {

printf("\n-------------------\n");

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

if (table->items[i]) {

printf("Index:%d, Key:%s, Value:%s", i, table->items[i]->key, table->items[i]->value);

if (table->overflow\_buckets[i]) {

printf(" => Overflow Bucket => ");

LinkedList\* head = table->overflow\_buckets[i];

while (head) {

printf("Key:%s, Value:%s ", head->item->key, head->item->value);

head = head->next;

}

}

printf("\n");

}

}

printf("-------------------\n");

}

**hashTable.h**

#ifndef HASH\_TABLE

#define HASH\_TABLE

#define CAPACITY 50000

typedef struct Ht\_item Ht\_item;

// Define the Hash Table Item here

struct Ht\_item {

char\* key;

char\* value;

};

typedef struct LinkedList LinkedList;

// Define the Linkedlist here

struct LinkedList {

Ht\_item\* item;

LinkedList\* next;

};

typedef struct HashTable HashTable;

// Define the Hash Table here

struct HashTable {

// Contains an array of pointers

// to items

Ht\_item\*\* items;

LinkedList\*\* overflow\_buckets;

int size;

int count;

};

unsigned long hash\_function(char\* str);

// Linked list

static LinkedList\* allocate\_list();

static LinkedList\* linkedlist\_insert(LinkedList\* list, Ht\_item\* item);

static Ht\_item\* linkedlist\_remove(LinkedList\* list);

static void free\_linkedlist(LinkedList\* list);

// HashTable

// -- Oveflow

static LinkedList\*\* create\_overflow\_buckets(HashTable\* table);

static void free\_overflow\_buckets(HashTable\* table);

// -- HashTable func

Ht\_item\* create\_item(char\* key, char\* value);

HashTable\* create\_table(int size);

void free\_item(Ht\_item\* item);

void free\_table(HashTable\* table);

void handle\_collision(HashTable\* table, unsigned long index, Ht\_item\* item);

void ht\_insert(HashTable\* table, char\* key, char\* value);

char\* ht\_search(HashTable\* table, char\* key);

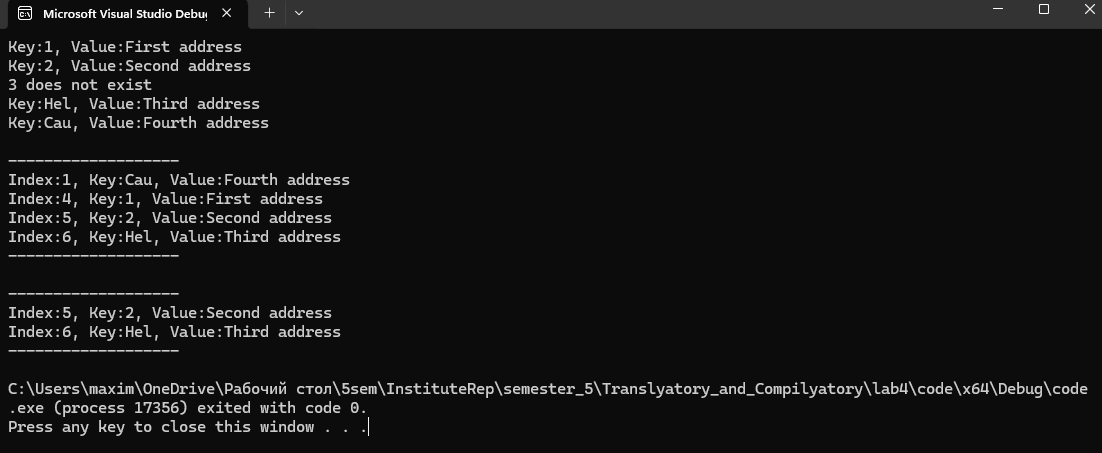
void ht\_delete(HashTable\* table, char\* key);

void print\_search(HashTable\* table, char\* key);

void print\_table(HashTable\* table);

#endif

**Результат работы**



**Вывод**

Ознакомились с методами построения таблиц идентификаторов. Изучили методы построения таблиц на основе хэш-функций.