Estruturas de Dados Avançadas (EDA) – INF1010
Departamento de Informática – PUC-Rio - 2023.1
Tarefa 3 – Tabelas de Dispersão (Hash Tables)

Lívia Lutz dos Santos - 2211055

Luiza Marcondes Paes Leme - 2210275

Introdução:

O trabalho consiste em implementar uma tabela hash com vetor de 1031 posições e com o TDA hashtable, para armazenar structs que guardam valores de placas de carro geradas aleatoriamente. As placas devem ser geradas no formato “CCCNNNN” e guardadas em um arquivo, para depois esse ser percorrido colocando as informações na hashtable. A inserção deve ter tratamento de colisão com encadeamento interno, enquanto conta o número de colisões que ocorreram. Para inserir, é necessário que cada struct tenha uma chave única para que a função de dispersão funcione corretamente, por isso a placa não pode ser usada como chave (caso aleatoriamente seja gerada mais de uma placa igual). Além disso, é preciso desenvolver uma função de busca por chave e uma função de exclusão de informações. Por fim, no momento de rodar o programa, é preciso calcular o tempo de inserção e busca de todos os elementos para quantidades de placas iguais a 128, 256 e 512, para assim gerar um gráfico contemplando a complexidade prática do algoritmo criado. Com isso, procura-se um menor tempo de execução e menor número de colisões.

Estrutura do programa:

Usamos apenas 1 header (hashtable.h) para definir as funções usadas, assim como as structs.

No módulo hashtable.c importamos as bibliotecas <stdio.h> , <stdlib.h> , <string.h> e o header hashtable.h, para lidar com alocações dinâmicas, strings e funções da hashtable.

No struct “ttabpos”, criamos um campo que corresponde ao array com informações das placas de carros, contendo um inteiro chave, que serve para buscar o elemento na hash table, a string dados para armazenar os caracteres da placa e um inteiro "próx" que indica a posição do próximo elemento na hashtable para lidar com colisões.

No struct “smapa”, criamos um campo correspondente à hash table , contendo um inteiro “tam”, que corresponde ao seu tamanho (quantidade de itens que consegue armazenar) e um ponteiro para o array “ttabpos”.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "hashtable.h"

#define TAMINICIAL 1031 //Primo após 1024

//Usamos encadeamento interior

//Struct do array de informações da placa

typedef struct {
        int chave; // Chave da placa no array
        char dados[8]; //Placa
        int prox; //Número da próxima posição livre no array
} ttabpos;

//Struct da hastable

struct smapa {
        int tam; //Tamanho da hashtable
        ttabpos* tabpos; //Ponteiro para o array de placas
};
```

Na função hash, passamos a chave do elemento e recebemos como retorno o índice que essa chave irá ocupar na hash table, para isso usamos o método da multiplicação (ou “meio do quadrado”) e multiplicamos a chave por um número aleatório antes para gerar índices não sequenciais. (O 14 foi escolhido porque dos números aleatórios que testamos ele estava dando o menor número de colisões).

```c
int hash(int key) {
    key *= key;
    key *= 14;
    key = key & 0x3FFF;
    key = key >> 4;
    return key % TAMINICIAL;
}
```

Usamos o método de encadeamento interior para fazer o mapeamento da hashtable.

Na função cria, passamos como parâmetro os valores presentes na struct ttabpos e ela retorna um ponteiro para a hash table criada (Mapa*). Nela fazemos uma alocação dinâmica para Mapa e uma alocação dinâmica no ponteiro para ttabpos (alocamos um espaço equivalente a 1031 elementos no array) e definimos o tamanho do mapa como sendo 1031. Depois, fazemos um loop para definir cada chave e próximo no mapa como -1 (equivale a não ter valor), retornando o ponteiro para Mapa (Mapa*).

```c
//Função de criar a hashtable

Mapa* cria(void) {

	//Cria hashtable

	int i;
	Mapa* m = (Mapa*)malloc(sizeof(Mapa));

	if (m == NULL) {
		printf("erro na alocação! \n");
		exit(1);
	}

	//Cria array de placas

	m->tabpos = (ttabpos*)malloc(TAMINICIAL * sizeof(ttabpos));
	if (m->tabpos == NULL) {
		printf("erro na alocação! \n");
		exit(1);
	}

	m->tam = TAMINICIAL;

	//inicializa as chaves e os próximos com -1

	for (i = 0; i < TAMINICIAL; i++) {
		m->tabpos[i].chave = -1;
		m->tabpos[i].prox = -1;
	}

	//retorna a hashtable

	return m;
}
```

Para a função de inserir, passamos como parâmetro a hash table na qual queremos inserir cada placa, a chave de cada uma e seus caracteres, retornando o número de colisões geradas. Primeiro calculamos seu índice usando a função hash guardando em uma variável e inicializamos 2 variáveis contadoras para colisão e quantidade de elementos. Se a chave no índice calculado for -1, inserimos os dados do parâmetro no índice e a função retorna um número de colisões de 0, caso contrário, procuramos dentro do índice por uma posição vazia para inserir, aumentando o número de colisões a cada passada do loop. Caso chegue na última posição do array, a função exibe uma mensagem de erro e retorna o número de colisões até o momento, se a posição livre for encontrada, fazemos a inserção dos dados e retornamos o número de colisões.

```c
int insere(Mapa* m, int c, char* d) {

	//calcula posição da chave no array usando hash()

	int pos = hash(c);

	//contador de colisões e posições

	int col = 0, i = 1;

	//se houver espaço para alocar os dados no índice calculado, insere normalmente retornando 0 colisões

	if (m->tabpos[pos].chave == -1) {
		m->tabpos[pos].chave = c;
		strcpy(m->tabpos[pos].dados, d);
		m->tabpos[pos] = m->tabpos[pos];
		return col;
	}

	//senão buscamos a próxima posição livre no índice e adicionamos as informações dos parâmetros lá
	//adicionamos 1 na colisão para cada posição nao livre no array

	for (; m->tabpos[pos + i].chave != -1; i++) {
		col++;

		//se chegarmos ao fim dos índices da hashtable, exibimos uma mensagem de erro e retornamos o número de colisões

		if ((pos + i) == 1030) {
			printf("Erro ao inserir.\n");
			return col;
		}
	}

	//Fazemos a inserção das informações dos parâmetros na posição livre encontrada e retornamos o número de colisões

	m->tabpos[pos + i].chave = c;
	strcpy(m->tabpos[pos + i].dados, d);
	m->tabpos[pos + i].prox = p.prox;
	m->tabpos[pos].prox = pos + i;
	return col;
}
```

Na função de excluir, passamos como parâmetro a hash table que queremos fazer a remoção e a chave do elemento a ser removido e a função retorna a posição que o elemento removido estava. Usando a função busca, obtemos sua posição no array, se a posição for -1, o elemento não se encontra no índice, por isso não há remoção e a função retorna -1, caso contrário, redefinimos a chave como -1 e os dados são removidos usando um "\0" no lugar da string presente, retornando a posição do elemento.

```c
int exclui(Mapa*m,int c){
	//buscamos a posição na qual o elemento a ser removido está
	int pos = busca(m, c);

	//caso não seja encontrado, a função retorna -1
	if (pos == -1)
		return -1;

	//senão definimos seu valor de chave como -1 e como não há placas, a string é alterada para \0
	m->tabpos[pos].chave = -1;
	strcpy(m->tabpos[pos].dados, "\0");

	//retorna a posição do elemento removido
	return pos;
}
```

Por fim, na função de busca, passamos como parâmetro a hash table e a chave do elemento que estamos procurando e a função retorna a posição que ele se encontra. Começamos obtendo o índice que a chave se encontra usando a função hash e inicializamos o contador de posições como 1. Se a chave na posição calculada for igual à chave do parâmetro, retornamos a posição, senão buscamos nas posições do índice por essa mesma condição, retornando a posição. Caso a posição seja a última, retornamos seu valor, senão a chave não foi encontrada e a função retorna -1.

```
int busca(Mapa*m,int c){

	//Pegamos o índice da chave procurada no array e inicializamos o contador de posições

	int pos = hash(c);
	int i = 1;

	//Se a primeira posição do índice for a buscada, retornamos o valor da posição

	if (m->tabpos[pos].chave == c) {
		return pos;
	}

	//senão buscamos dentro da posição até encontrar a chave procurada

	else {
		for (; m->tabpos[pos+i].prox != -1; i++) {

			//se for encontrada, retornamos sua posição

			if (m->tabpos[pos + i].chave == c) {
				return pos + i;
			}
		}
	}

	//caso chegue na última posição (-1) retornamos pos + i

	if (m->tabpos[pos + i].chave == c) {
		return pos + i;
	}

	//se não encontrou a chave, retornamos -1

	return -1;

}
```

Solução:

Na main, para calcular as colisões e os tempos, resolvemos fazer um arquivo com 512 placas (o total que precisaríamos) e depois testar separadamente a inserção e busca com as três quantidades pedidas. Nós estranhamos que alguns tempos estavam dando valores iguais, por isso criamos uma variável para guardar cada um para ter certeza de que não era algum problema de não estar sendo possível alterar os valores dentro de cada variável.

```c
/* Lívia Lutz dos Santos - 2211055 */
/* Luiza Marcondes Paes Leme - 2210275 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include "hashtable.h"

int main(void) {

    //caracteres de numeros nas placas

    char numerais[10] = { '0','1','2','3','4','5','6','7','8','9'};

    //placa do carro

    char placa[8];

    //caracteres a serem inseridos em placas

    char c1,c2,c3,n,ch;

    //variaveis contadoras

    int c = 0,j,d = 0,e = 0;

    //variaveis para calcular o tempo total

    double elapsed,elapsed1,elapsed2,elapsed3,elapsed4,elapsed5,elapsed6,elapsed7,elapsed8;

    //arquivo

    FILE* f;

    //variaveis de tempo

    clock_t t0,t1,t2,t3,t4,t5,t6,t7,t8,t9,t10;
```

```c
//abrindo o arquivo para escrita

f = fopen("Placas.txt", "w");

//tornando a funcao rand() completamente aleatoria

srand(time(NULL));

//inserindo as placas no arquivo

for (int i = 0; i < 512; i++) {

    //gera 3 caracteres ASCII e inclui em placas

    c1 = rand() % 26 + 65;
    c2 = rand() % 26 + 65;
    c3 = rand() % 26 + 65;
    placa[0] = '\0';
    strncat(placa, &c1, 1);
    strncat(placa, &c2, 1);
    strncat(placa, &c3, 1);

    //gera 4 caracteres numericos e insere em placas

    for (int i = 0; i < 4; i++) {
        n = numerais[rand() % 10];
        strncat(placa, &n,1);
    }

    //escreve a placa no arquivo

    fwrite(&placa,sizeof(placa)-1, 1, f);
    ch = '\n';
    fwrite(&ch, sizeof(ch), 1, f);

}

//fecha o arquivo
```

```c
//fecha o arquivo

fclose(f);

//cria os mapas para cada teste

Mapa* m = cria();
Mapa* m1 = cria();
Mapa* m2 = cria();

// ------------------------ 128 placas ----------------------//

//abre o arquivo para leitura


f = fopen("Placas.txt", "r");

//comeca a contar o tempo de insercao

t0 = clock();

//para 128 placas

for (j = 0; j < 128; j++) {

    //le a placa do arquivo

    fread(placa,sizeof(placa), 1, f);

    placa[7] = '\0';

    //insere a placa no mapa

    c += insere(m,j,placa);

}

//termina a contagem
```

```c
t1 = clock();

printf("%d colisoes para 128 placas\n", c);

//fecha o arquivo

fclose(f);

//calcula o tempo total de insercao

elapsed = 1000 * ((double)t1 - (double)t0 / CLOCKS_PER_SEC);
printf("Tempo usado pra inserir 128 placas em milisegundos: %.2f\n",elapsed);

//comeca a contar o tempo de busca

t2 = clock();

//busca cada placa no mapa

for (j = 0; j < 128; j++) {
    c = busca(m, j);
}

//termina a contagem do tempo de busca

t3 = clock();

//calcula o tempo total de busca

elapsed1 = 100 * ((double)t3 - (double)t2 / CLOCKS_PER_SEC);
printf("Tempo usado para buscar 128 placas em mililsegundos:%.2f\n", elapsed1);

//calcula o tempo total de busca + insercao

elapsed2 = elapsed + elapsed1;

printf("Tempo de busca e insercao de 128 placas em ms: %.2f\n", elapsed2);
```

```c
// ---------------------- 256 placas ----------------------//

f = fopen("Placas.txt", "r");

//comeca a contar o tempo de insercao

t3 = clock();

//para 128 placas

for (j = 0; j < 256; j++) {

    //le a placa do arquivo

    fread(placa, sizeof(placa), 1, f);

    placa[7] = '\0';

    //insere a placa no mapa

    d += insere(m1, j, placa);

}

//termina a contagem

t4 = clock();

printf("%d colisoes para 256 placas\n", d);

//fecha o arquivo

fclose(f);

//calcula o tempo total de insercao

elapsed3 = 1000 * ((double)t4 - (double)t3 / CLOCKS_PER_SEC);
printf("Tempo usado pra inserir 256 placas em milisegundos: %.2f\n",elapsed3);
```

```c
//comeca a contar o tempo de busca

t5 = clock();

//busca cada placa no mapa

for (j = 0; j < 256; j++) {
    d = busca(m1, j);
}

//termina a contagem do tempo de busca

t6 = clock();

//calcula o tempo total de busca

elapsed4 = 100 * ((double)t6 - (double)t5 / CLOCKS_PER_SEC);
printf("Tempo usado para buscar 256 placas em mililsegundos:%.2f\n", elapsed4);

//calcula o tempo total de busca + insercao

elapsed5 = elapsed3 + elapsed4;

printf("Tempo de busca e insercao de 256 placas em ms: %.2f\n", elapsed5);


// ----------------------------- 512 placas --------------------//


f = fopen("Placas.txt", "r");

//comeca a contar o tempo de insercao

t7 = clock();

//para 512 placas

for (j = 0; j < 512; j++) {
```

```
    //le a placa do arquivo

    fread(placa, sizeof(placa), 1, f);

    placa[7] = '\0';

    //insere a placa no mapa

    e += insere(m2, j, placa);

}

//termina a contagem

t8 = clock();

printf("%d colisoes para 512 placas\n", e);

//fecha o arquivo

fclose(f);

//calcula o tempo total de insercao

elapsed6 = 1000 * ((double)t8 - (double)t7 / CLOCKS_PER_SEC);
printf("Tempo usado pra inserir 512 placas em milisegundos: %.2f\n",elapsed6);

//comeca a contar o tempo de busca

t9 = clock();

//busca cada placa no mapa

for (j = 0; j < 512; j++) {
    e = busca(m2, j);
```

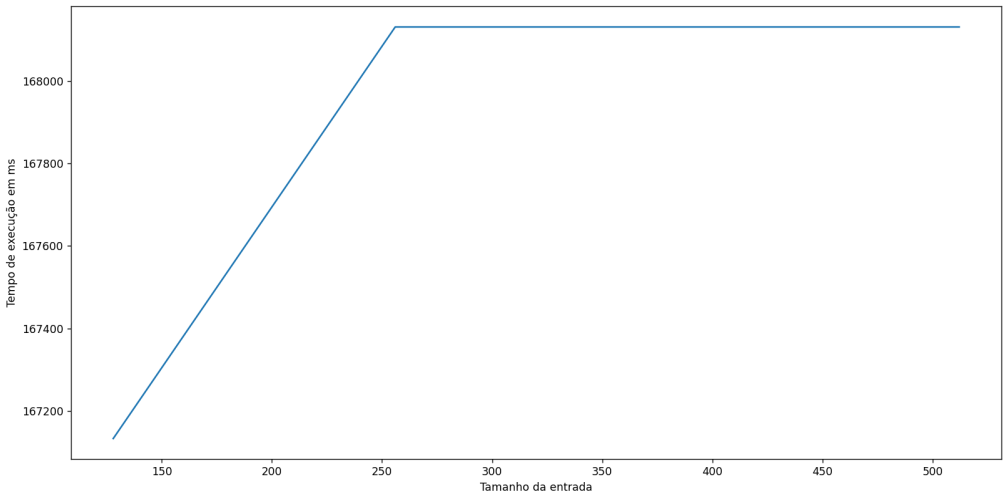
```
}

//termina a contagem do tempo de busca

t10 = clock();

//calcula o tempo total de busca

elapsed7 = 100 * ((double)t10 - (double)t9 / CLOCKS_PER_SEC);
printf("Tempo usado para buscar 512 placas em mililsegundos:%.2f\n", elapsed7);

//calcula o tempo total de busca + insercao

elapsed8 = elapsed7 + elapsed6;

printf("Tempo de busca e insercao de 512 placas em ms: %.2f\n", elapsed8);


return 0;
```

A saída desse programa foi a seguinte:

```
1 colisoes para 128 placas
Tempo usado pra inserir 128 placas em milisegundos: 151848.00
Tempo usado para buscar 128 placas em mililsegundos:15284.70
Tempo de busca e insercao de 128 placas em ms: 167132.70
14 colisoes para 256 placas
Tempo usado pra inserir 256 placas em milisegundos: 152847.00
Tempo usado para buscar 256 placas em mililsegundos:15284.70
Tempo de busca e insercao de 256 placas em ms: 168131.70
133 colisoes para 512 placas
Tempo usado pra inserir 512 placas em milisegundos: 152847.00
Tempo usado para buscar 512 placas em mililsegundos:15284.70
Tempo de busca e insercao de 512 placas em ms: 168131.70
```

Com os valores dessa saída utilizamos um programa em Python para exibir o gráfico da complexidade prática.

```python
# Lívia Lutz dos Santos - 2211055
# Luiza Marcondes Paes Leme - 2210275

# importando a biblioteca para desenhar gráficos

import matplotlib.pyplot as plt

# valores de entrada
x = [128,256,512]

# tempo de execução de cada valor em ms
y = [167132.70,168131.70,168131.70]

# desenhando os pontos
plt.plot(x, y)

# nomeando o eixo x para o tamanho da entrada

plt.xlabel('Tamanho da entrada')

# nomeando o eixo y para o tamanho da entrada

plt.ylabel('Tempo de execução em ms')

# desenha o gráfico na tela
plt.show()
```



Conclusão:

Alguns aspectos da tarefa foram mais difíceis do que nós imaginávamos e outros foram mais fáceis do que imaginávamos também. A começar pela função de dispersão, ficamos um bom tempo testando números aleatórios para ver se algum tinha um resultado significativamente melhor quanto às colisões. Além disso, demoramos para conseguir desenvolver corretamente o algoritmo de inserção quando havia colisões. Por outro lado, quando esse estava resolvido, a função de busca foi fácil pois a lógica já estava implementada na de inserção. A função de excluir parecia complicada, mas no final conseguimos fazer rapidamente sem problemas.

Quanto à main, a princípio parecia complicada a parte de contar o tempo, mas lembramos de uma tarefa de Prog II em que tivemos que fazer isso e consultamos para essa tarefa. Ademais, acredito que a maior dificuldade nesse trabalho foi fazer o gráfico com alguma biblioteca da linguagem C, uma vez que a “graphics.h” estava dando problema e não soubemos resolver. Com isso, resolvemos usar os valores de tempo que o programa em C calcula para gerar os gráficos com um programa em Python. Por fim, quanto à execução, estranhamos o fato de alguns tempos darem os mesmos valores, ainda que com uma quantidade diferente de placas como entrada.