Nome: João Pedro de Abreu Martins Pereira
RA: 21026515
Disciplina: Algoritmos e Estruturas de Dados II
Professor: Carlos da Silva dos Santos
Link do Repositório: https://github.com/jpampereira/AEDII_funcoes_de_espalhamento

---

Projeto 1 – Funções de Espalhamento (hashing)

Arquivo hashing.h

```c
#ifndef HASHING_H
#define HASHING_H

int chaveDivisao(int key, int m, int* r);
int chaveMultiplicacao(int key, int m, float a, int* r);

#endif // HASHING_H
```

Arquivo hashing.c

```c
#include "hashing.h"

int chaveDivisao(int key, int m, int* r) {
    if(key < 0 || m < 0) return 0;

    *r = key % m;

    return 1;
}

int chaveMultiplicacao(int key, int m, float a, int* r) {
    if(key < 0 || m < 0 || a < 0) return 0;

    float value = key * a;
    value = value - (int) value;
    *r = (int) (m * value);

    return 1;
}
```

Dentre as linguagens de programação permitidas para a implementação das funções foi escolhida a linguagem C. A implementação encontra-se no repositório informado no início do documento.

Seguindo os pré-requisitos, as funções de espalhamento pelo método da divisão e da multiplicação foram implementadas em um módulo, nos arquivos hashing.h, onde as funções foram declaradas e no arquivo hashing.c, onde foram implementadas de fato. Isso permite a reutilização dessas funções.

Outro pré-requisito atendido foi procurar tratar situações em que os parâmetros passados para as funções estivessem em intervalos inválidos (entende-se, valores negativos). A linguagem C, por se tratar de linguagem mais antiga, não possui o tratamento de exceções, como linguagens mais modernas como C++, Java, PHP, etc. A alternativa encontrada foi utilizar de um modelo de implementação bastante utilizado na linguagem aonde as funções retornam valores para indicar seu correto funcionamento. Caso a função tenha sido executada normalmente, o valor retornado é 1, caso contrário, retorna 0. O valor gerado a partir da chave passada como parâmetro é armazenado em um endereço de memória indicado pelo usuário através dos parâmetros passados para a função (r). Portanto, caso um dos valores entre a chave (key), o

tamanho da tabela hash (m) e a variável ‘a’ (exclusiva do método da multiplicação) sejam menores do que zero, a função retorna erro antes mesmo de ser executada.

Arquivo main.c

```c
#include <stdio.h>
#include "hashing.h"


void ex1ab(int tableLen) {
    int value;
    int result;

    for(int key = 0; key <= 100; key++) {
        result = chaveDivisao(key, tableLen, &value);
        if(result == 0)
            exit (1);

        if(value == 3)
            printf("Chave = %d => Valor = %d\n", key, value);
    }
}

void ex1c(int tableLen, char* filename) {
    FILE *file = createFile(filename);

    int* counter = createArray(tableLen);

    int value;
    int result;
    for(int key = 1; key <= 10000; key++) {
            result = chaveDivisao(key, tableLen, &value);
            if(result == 0)
                exit(1);

            counter[value]++;
    }

    printFile(file, counter, tableLen);
    fclose(file);
}

void ex2ab(int tableLen, float a, char* filename) {
    FILE *file = createFile(filename);

    int* counter = createArray(tableLen);

    int value;
    int result;
    for(int key = 1; key <= 500000; key++) {
            result = chaveMultiplicacao(key, tableLen, a, &value);
            if(result == 0)
                exit(1);

            counter[value]++;
    }

    printFile(file, counter, tableLen);
    fclose(file);
}
```

Nas imagens acima, podemos ver as funções que realizam os experimentos solicitados no roteiro do projeto. Na primeira imagem, vemos o uso da diretiva *#include* para a importação do módulo com as funções de hashing implementadas.

A função *ex1ab* realiza os experimento ‘a’ e ‘b’ do método de divisão, em que é passado para ela o tamanho da tabela hash e então são realizados testes para chaves de 0 a 100 e caso o valor gerado a partir da chave seja igual a 3, ele é impresso.

Na função *ex1c*, são testadas chaves de 1 a 10.000 também para o método da divisão. A diferença para o primeiro exercício é que nesse caso todas as tuplas (chave, valor) são impressas em um arquivo de texto chamado de *saida_experimento1c*.

Na função *ex2ab* é realizado o mesmo procedimento que na função ex1c, dessa vez para o método da multiplicação. As saídas nesses casos vão para os arquivos saida_experimento2a e *saida_experimento2b*.

A função fica responsável por chamar essas funções para realizar os experimentos, passando os valores pré-determinados nos enunciados.

Função de espalhamento pelo método da divisão

(a) Usando m = 12, faça um programa que teste sua função com valores de chave variando de 0 até 100. Quando o resultado h(x) for igual a 3, imprima o valor de chave correspondente. Você consegue notar um padrão para esses valores de chave?

```
Experimento a:

Chave = 3 => Valor = 3
Chave = 15 => Valor = 3
Chave = 27 => Valor = 3
Chave = 39 => Valor = 3
Chave = 51 => Valor = 3
Chave = 63 => Valor = 3
Chave = 75 => Valor = 3
Chave = 87 => Valor = 3
Chave = 99 => Valor = 3
```

Podemos perceber que existe um padrão na distribuição das chaves, onde a cada m chaves de valores sequenciais processadas pela função de espalhamento pelo método da divisão, o resultado h(x) correspondente é o mesmo. No caso, como m = 12, a cada 12 chaves sequenciais, o valor obtido será o mesmo. Com isso podemos perceber que o espalhamento é uniforme.

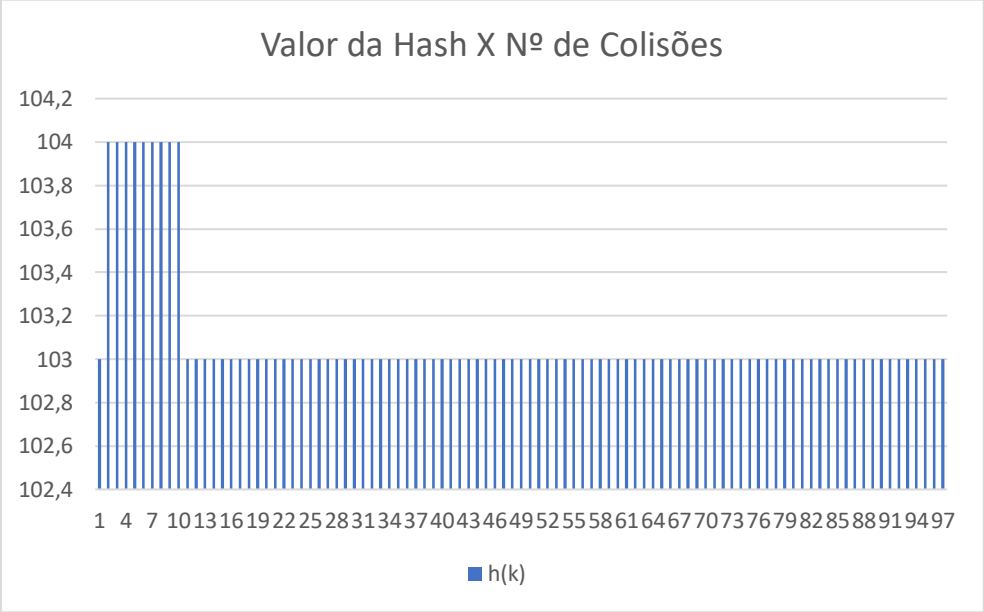
(b) Repita o item anterior com m = 11 e imprimindo as chaves que resultam em h(x) = 3. Você consegue notar um padrão para esses valores de chave?

```
Experimento b:

Chave = 3 => Valor = 3
Chave = 14 => Valor = 3
Chave = 25 => Valor = 3
Chave = 36 => Valor = 3
Chave = 47 => Valor = 3
Chave = 58 => Valor = 3
Chave = 69 => Valor = 3
Chave = 80 => Valor = 3
Chave = 91 => Valor = 3
```

Podemos perceber que acontece a mesma coisa do experimento anterior. Com a diferença de que para esse caso, como m = 11, a cada 11 chaves sequenciais, o valor resultante obtido será o mesmo.

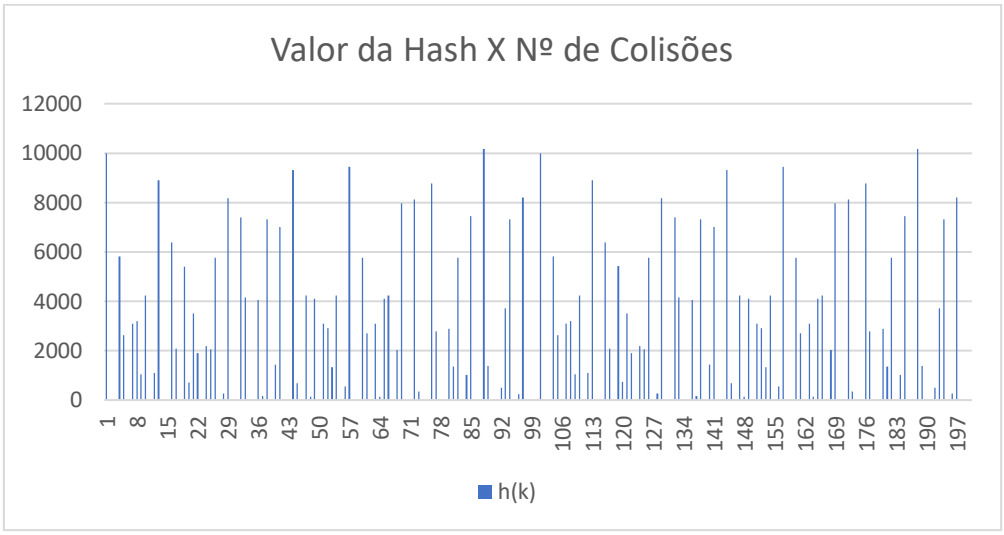
(c) Usando m = 97 (um número primo) conte o número de colisões para cada valor diferente de h(k), usando chaves no intervalo {1, 2, 3, ... , 10000}. Dica: você pode acumular as contagens em um vetor de m posições, inicialmente preenchido com zeros. A cada vez que você calcular um novo valor h(k), incremente a posição correspondente no vetor de contagens. Salve os resultados dessas contagens em um arquivo e faça um gráfico de número de colisões em função do valor do hash. Você pode salvar as contagens em um arquivo de valores separados por vírgula, em que cada linha tem o formato chave, contagem. O gráfico pode ser construído em um programa qualquer de planilhas, por exemplo.



Como foi comentado no experimento (a) e podemos perceber através do gráfico, o espalhamento dos valores obtidos a partir do método da divisão é uniforme, ou seja, todos os valores possuem a mesma probabilidade de serem direcionados para uma entrada da tabela hash, mesmo utilizando um valor de m primo e longe de uma potência de 2. Podemos levar em consideração também que o valor de posições na tabela é muito menor do que a quantidade de chaves testadas, tendo um fator de carga de $\alpha = 10.000/97 = 103,09$, ou seja, para cada posição da tabela hash, existem 103 elementos a serem armazenados, que é o valor que podemos observar olhando o gráfico, sendo outro indício para comprovar que para o método da divisão, o espalhamento é uniforme

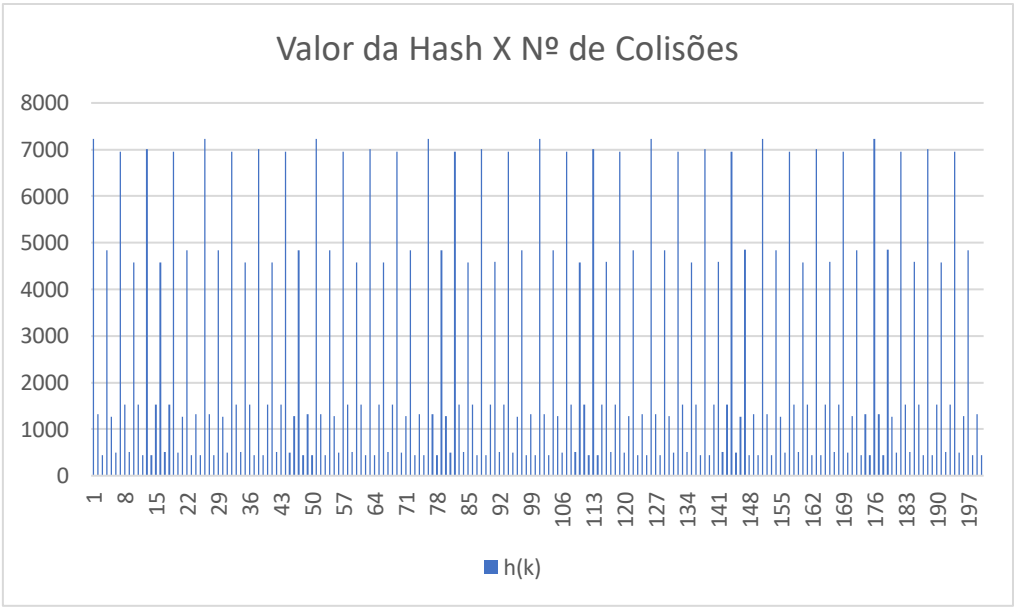
Função de espalhamento pelo método da multiplicação

(a) Usando m = 200 e A = 0.62, faça um programa que teste sua função com valores de chave variando de 1 até 500 mil. Conte o número de colisões para cada valor diferente de h(k). Salve os resultados dessas contagens em um arquivo e faça um gráfico de número de colisões em função do valor do hash.



Podemos perceber que diferentemente da função de espalhamento pelo método da divisão, os valores resultantes h(k) correspondentes as chaves k que variam de 1 até 500 mil não são distribuídos de forma uniforme quando calculados através do método da multiplicação, visto que existem alguns valores de h(k) que foram muito mais calculados do que outros, como por exemplo, h(k) = 87 foi calculado 10.158 vezes, enquanto h(k) = 86 não foi calculado nenhuma vez.

(b) Usando m = 200 e A = 0,61803398875 (número derivado da razão áurea), repita o item anterior. Compare os resultados de distribuição das colisões.



Comparando ao experimento anterior, é possível perceber que a distribuição das colisões aumentou, não havendo mais casos em que um único valor h(k) foi calculado mais de dez mil vezes através do método de divisão, sendo o valor mais calculado h(k) = 100 7229 vezes e tendo agora todos os valores entre 0 e m-1 tendo sido calculados ao menos uma vez. Isso mostra que um bom valor para o coeficiente a é importante para o bom funcionamento da função de hashing, mesmo para casos onde os coeficientes se diferem em décimos.