UNIVERSIDADE FEDERAL DE SÃO JOÃO DEL REI

DAVI DOS REIS DE JESUS

GABRIEL DE PAULA MEIRA

HASH TEXT TABLE

Documentação completa

SÃO JOÃO DEL REI - MG

2022

DAVI DOS REIS DE JESUS

GABRIEL DE PAULA MEIRA

HASH TEXT TABLE

Documentação completa

Parte escrita do Trabalho Prático apresentado ao curso de Ciência da Computação como parte dos requisitos necessários à obtenção da aprovação na disciplina de Algoritmos e Estruturas de Dados II.

Orientador: Daniel Madeira

SÃO JOÃO DEL REI - MG

2022

SUMÁRIO

1 INTRODUÇÃO 04
2 DESENVOLVIMENTO 05
3 CONCLUSÃO 06
REFERÊNCIAS.....07

1 INTRODUÇÃO

O desafio propõe implementar uma tabela hash para contabilizar as ocorrências de determinas palavras em um um arquivo de texto, exibindo no console a quantidade de vezes que uma palavra aparece e linhas onde se encontra. O programa deve ser eficiente em uso de processamento, memória e tempo de execução.

Dadas as orientações do trabalho prático, algumas informações se fazem muito importantes para a idealização do código:

- A tabela deve usar endereçamento aberto
- Cada palavra deve ser atribuída à um único espaço
- Ignorar palavras de um caractere
- Ignorar variações de maiúsculo e minúsculo

Serão utilizados dois arquivos para tal tarefa, um contendo o texto (input.txt) e outro contendo as palavras a serem pesquisadas (pesquisa.txt). O texto possui até 256 palavras diferentes onde cada linha possui até 80 caracteres e cada palavra possui até 20 caracteres. Já a pesquisa contém uma palavra por linha.

A parte de código deve ser escrita a fim de se obter um programa com um código limpo, objetivo, bem elaborado e com a melhor performance possível. Levando em conta o tamanho da demanda que poderia ser requerida do código. Além de produzir o software, a dupla deve desenvolver uma documentação completa explicando todo o funcionamento e o progresso realizado até que se chegasse no resultado final.

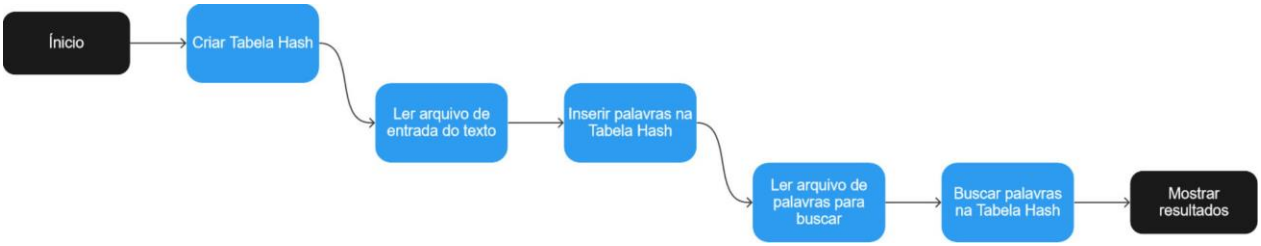
2 DESENVOLVIMENTO

Dadas as orientações sobre o trabalho, torna-se necessário, em primeiro plano, pensar em como o código deverá funcionar, definindo as etapas e tarefas a serem cumpridas para conseguir um software que não apenas resolva o problema, mas que seja de fácil manutenção e adaptação à forma a qual o projeto escalar.

2.1 ESQUEMATIZANDO O PROBLEMA

Para entender melhor o problema e como deverá ser elaborado o programa, a dupla desenvolveu um fluxograma de funcionamento do código principal, como mostrado na Figura 1.

Figura 1 - Fluxograma do funcionamento geral



Fonte: autoria própria

2.2 PROGRAMANDO A SOLUÇÃO

Após determinar o que deveria ser feito, o próximo passo foi colocar em prática as habilidades de programação na linguagem C, seguindo padrões de código para garantir a qualidade do produto final.

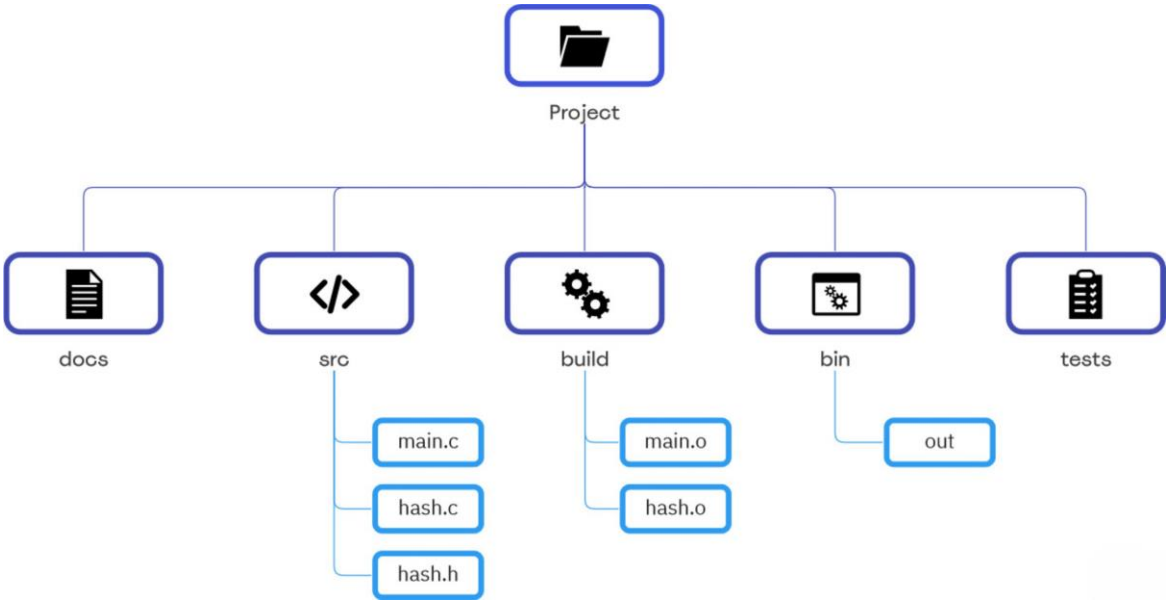
2.2.1 Estrutura de Arquivos

Antes de iniciar o processo de escrita dos códigos é ideal que se planeje a forma como tudo será estruturado, a começar pelo diretório dos arquivos. Não é preciso conhecimento técnico para saber que arquivos “jogados” em uma pasta acabam dificultando o gerenciamento e comprometendo todo o fluxo de trabalho.

Sabendo disso, a equipe buscou por um estrutura que padronizasse de maneira simples o diretório principal, chegando ao resultado mostrado na Figura 2:

- src – Código-fonte do programa (arquivos .c e .h)
- build – Arquivos gerados durante a compilação (arquivos .o)
- bin – Diretório de saída (arquivos executáveis)
- docs – Documentação do projeto
- tests – Arquivos usados nos testes do programa

Figura 2 - Estrutura dos arquivos



Fonte: autoria própria

Além da estrutura das pastas principais, outros arquivos serviram para auxiliar no desenvolvimento do programa, tais como:

- Makefile – compilar os múltiplos arquivos.
- .gitignore – auxiliar no controle de versionamento do projeto.
- README.md – fornecer uma descrição sobre o projeto quando publicado.

2.2.2 Funções

Para garantir uma melhor organização do programa, este foi modularizado em três arquivos:

- main.c - contendo a função main() e as funções de leitura dos arquivos.
- hash.c - contendo as funções usadas para manipular a tabela hash.
- hash.h - contendo os registros e assinaturas das funções.

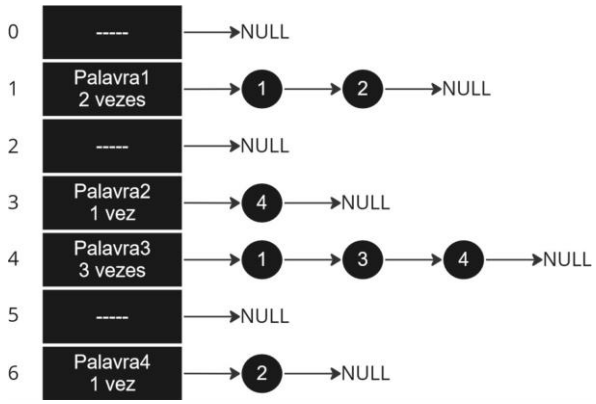
2.2.2.1 Criar Tabela Hash

Como definido nas orientações do trabalho, a estrutura da tabela foi projetada para utilizar hash de endereçamento aberto, ou seja, é composta por um vetor de N posições, onde cada uma representa uma palavra.

No programa elaborado, cada hash é um registro que contém a palavra (char*), a quantidade total de ocorrências dessa palavra (int) e um ponteiro para a primeira linha em que ela aparece (struct occurence), este que por sua vez é um registro contendo o número da linha do texto e um novo ponteiro que direciona para a próxima linha de ocorrência (struct occurence), formando um ciclo até chegar na última, que aponta para NULL, representando o fim das linhas de ocorrência.

```c
typedef struct occurrence {
    int line;
    struct occurrence* p_prox;
} occurrence_t;

typedef struct {
    char word[MAX_CHAR_PER_WORD + 1];
    int quantity;
    occurrence_t* first;
} hash;
```

Figura 3 - Diagrama da Tabela Hash



Fonte: autoria própria

No começo do programa, a tabela é criada com todos os elementos vazios, representados pela string “-----”.

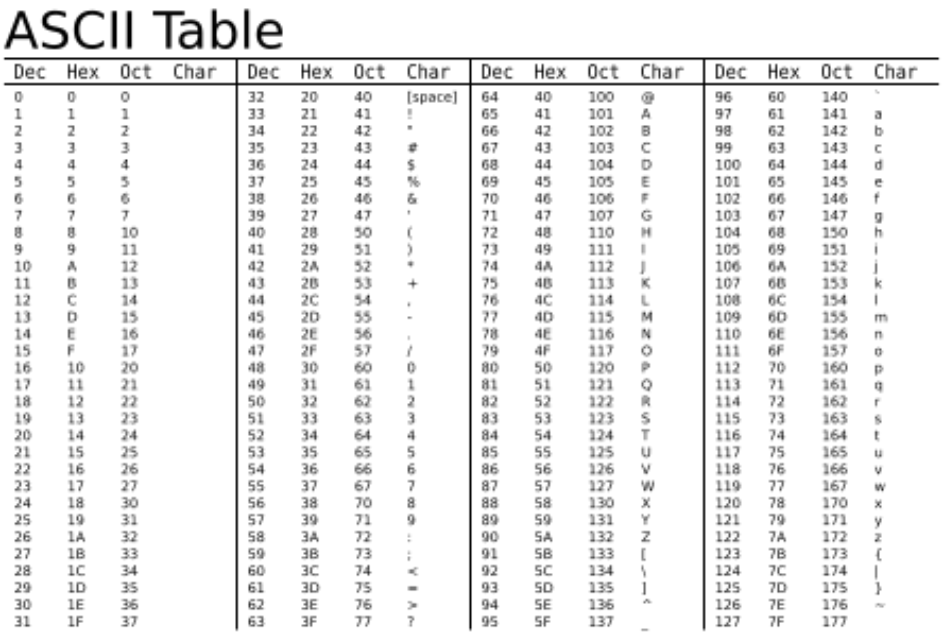
```c
#define EMPTY "-----"

// Create hashTable and assign EMPTY for all words
hash* createHashTable() {
    hash* newHashTable = (hash*)(malloc(HASH_SLOTS * sizeof(hash)));
    for (int i = 0; i < HASH_SLOTS; i++) {
        strcpy(newHashTable[i].word, EMPTY);
        newHashTable[i].first = NULL;
    }
    return newHashTable;
}
```

2.2.2.2 Gerar hash por palavra

A proposta da Tabela Hash é organizar os dados de maneira que a busca seja simples e efetiva, já que cada dado tem seu endereço definido por uma função de espalhamento, ou função de hash.

Um dos segredos da eficiência da tabela hash é a forma pela qual a função escolhe o índice para que o dado seja alocado. No caso das strings recebidas do texto, uma excelente forma de gerar a “chave” é convertendo os caracteres para números usando a tabela ASCII, representada na Figura 4.

Figura 4 - Tabela ASCII



ASCII Table

Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char
0	0	0		32	20	40	[space]	64	40	100	@	96	60	140	`
1	1	1		33	21	41	!	65	41	101	A	97	61	141	a
2	2	2		34	22	42	"	66	42	102	B	98	62	142	b
3	3	3		35	23	43	#	67	43	103	C	99	63	143	c
4	4	4		36	24	44	$	68	44	104	D	100	64	144	d
5	5	5		37	25	45	%	69	45	105	E	101	65	145	e
6	6	6		38	26	46	&	70	46	106	F	102	66	146	f
7	7	7		39	27	47	'	71	47	107	G	103	67	147	g
8	8	10		40	28	50	(	72	48	110	H	104	68	150	h
9	9	11		41	29	51	)	73	49	111	I	105	69	151	i
10	A	12		42	2A	52	*	74	4A	112	J	106	6A	152	j
11	B	13		43	2B	53	+	75	4B	113	K	107	6B	153	k
12	C	14		44	2C	54	,	76	4C	114	L	108	6C	154	l
13	D	15		45	2D	55	-	77	4D	115	M	109	6D	155	m
14	E	16		46	2E	56	.	78	4E	116	N	110	6E	156	n
15	F	17		47	2F	57	/	79	4F	117	O	111	6F	157	o
16	10	20		48	30	60	0	80	50	120	P	112	70	160	p
17	11	21		49	31	61	1	81	51	121	Q	113	71	161	q
18	12	22		50	32	62	2	82	52	122	R	114	72	162	r
19	13	23		51	33	63	3	83	53	123	S	115	73	163	s
20	14	24		52	34	64	4	84	54	124	T	116	74	164	t
21	15	25		53	35	65	5	85	55	125	U	117	75	165	u
22	16	26		54	36	66	6	86	56	126	V	118	76	166	v
23	17	27		55	37	67	7	87	57	127	W	119	77	167	w
24	18	30		56	38	70	8	88	58	130	X	120	78	170	x
25	19	31		57	39	71	9	89	59	131	Y	121	79	171	y
26	1A	32		58	3A	72	:	90	5A	132	Z	122	7A	172	z
27	1B	33		59	3B	73	;	91	5B	133	[	123	7B	173	{
28	1C	34		60	3C	74	<	92	5C	134	\	124	7C	174	|
29	1D	35		61	3D	75	=	93	5D	135	]	125	7D	175	}
30	1E	36		62	3E	76	>	94	5E	136	^	126	7E	176	~
31	1F	37		63	3F	77	?	95	5F	137	_	127	7F	177	

Fonte: Programação GLOBAL®

A chave é definida pela soma dos códigos ASCII dos caracteres multiplicados por suas posições na string.

```c
// Generate hash based on each char value (ASCII) and position
int hashGenerator(char* string) {
    int key = 0;
    for (int i = 0; i < strlen(string); i++) {
        key += ((i + 1) * string[i]);
    }
    int index = key % HASH_SLOTS;
    return index;
}
```

O endereço da palavra não pode ser diretamente a chave resultante da fórmula, visto que a quantidade de algarismos escala exponencialmente, o que seria um gasto exorbitante de memória sem a devida utilidade. Dessa forma, é utilizado o operador de “resto da divisão” que fará com que o maior índice esteja limitado à quantidade de hashs definida previamente pelo programador.

2.2.2.3 Inserir palavras na tabela

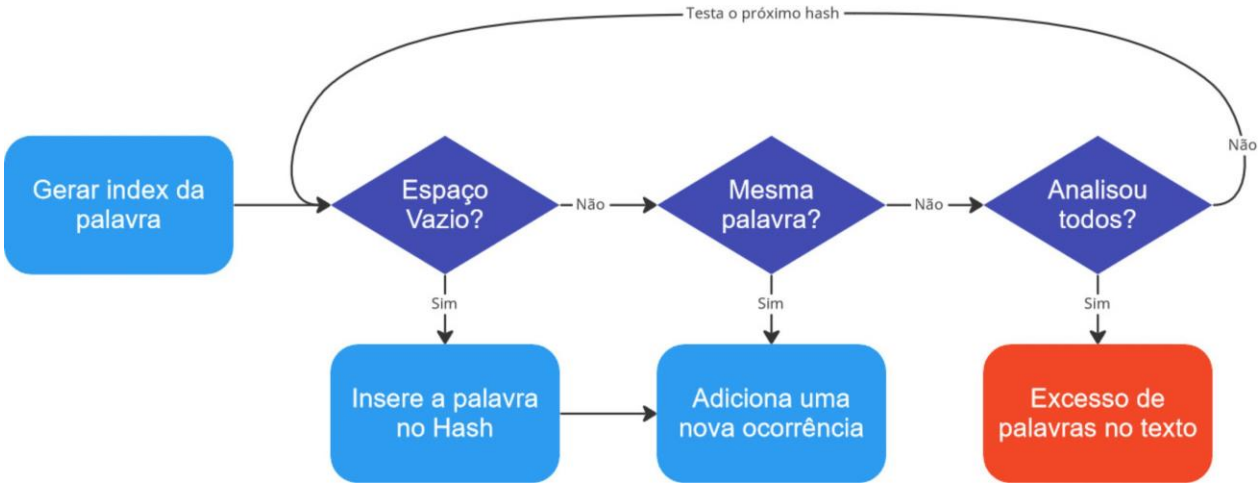
Para inserir uma palavra na tabela é necessário saber onde ela será inserida, para isso a função descrita em 2.2.2.2 (Gerar hash por palavra) é usada.

No caso de uma tabela vazia, qualquer que seja a palavra, ela será alocada no espaço definido por seu hash. Esse processo se repete bem até que o espaço que deveria estar vazio já esteja ocupado por outra palavra diferente, nessa situação é necessário procurar o próximo espaço vazio para conseguir inserir na sequência.

Caso durante a procura por um espaço vazio seja encontrada a mesma palavra que está tentando ser inserida, significa que deve ser acrescentada mais uma ocorrência da mesma, avaliando se é a primeira vez que ocorre na mesma linha ou não.

Caso a tabela seja percorrida por completo e não há nenhum espaço vazio para acrescentar a palavra, significa que a tabela foi completamente preenchida, o que resulta em um erro, já que a quantidade máxima de palavras diferentes deveria ter sido informada previamente para o usuário.

Figura 5 - Fluxograma de inserção na tabela



Fonte: autoria própria

Para adicionar uma nova ocorrência da palavra é necessário verificar se é a primeira vez que a palavra ocorre no texto, para isso basta checar se o ponteiro da primeira ocorrência aponta para NULL. Caso contrário, é preciso navegar até a última ocorrência e verificar se é ou não a primeira vez que a palavra aparece naquela linha. Em ambos os casos a quantidade total é aumentada em uma unidade.

```c
// Add new occurrence to hashCell (same line or another)
void addOccurrence(hash* hashCell, int line) {
    occurrence_t** aux = &hashCell->first;

    if (*aux == NULL) {  // First occurrence is NULL
        *aux = createOccurrence(line);
        hashCell->quantity = 1;
    } else {
        while ((*aux)->p_prox != NULL) {
            aux = &((*aux)->p_prox);  // Browse the list until last line
        }
        if (line != (*aux)->line) (*aux)->p_prox = createOccurrence(line);
        hashCell->quantity++;
    }
}

// Insert the string in hashTable by hash
void insertElement(hash* hashTable, char* string, int line) {
    int index = hashGenerator(string);

    int counter = 0;
    while (counter < HASH_SLOTS) {
        // Slot is EMPTY, can assign the string and add new occurrence
        if (strcmp(hashTable[index].word, EMPTY) == 0) {
            strcpy(hashTable[index].word, string);
            addOccurrence(&hashTable[index], line);
            return;
        }

        // Slot has the same string, add line occurrence
        if (strcmp(hashTable[index].word, string) == 0) {
            addOccurrence(&hashTable[index], line);
            return;
        }

        counter++;
        index++;
        index = index % HASH_SLOTS;
    }

    printf("\nERROR - HashTable full (%d+ different words)\n\n", HASH_SLOTS);
    exit(1);
}
```

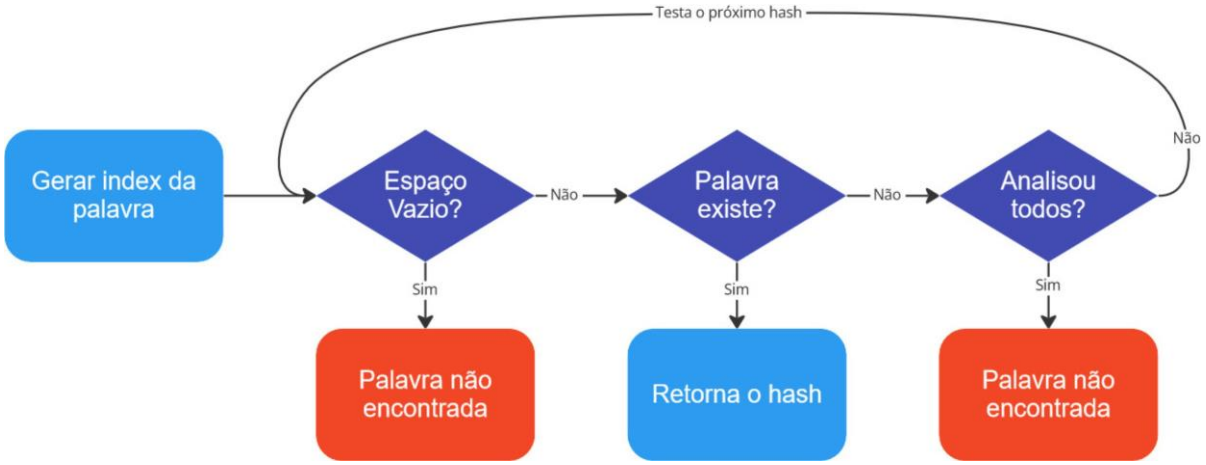
2.2.2.4 Buscar palavras na tabela

A forma de buscar as palavras é semelhante à de inserção, já que também é necessário gerar o hash e percorrer a tabela.

Durante a busca, se for encontrado um espaço vazio no endereço determinado pelo hash, significa que a palavra não existe. Da mesma forma, se todos os espaços forem percorridos e nenhum corresponder à palavra buscada, significa que a tabela está completa, mas mesmo assim a palavra não existe.

Caso encontre a palavra, a função deve retornar o verdadeiro índice onde ela se encontra, para que as ocorrências sejam mostradas ao usuário em outra função.

Figura 6 - Fluxograma de busca na tabela



Fonte: autoria própria

Foi padronizado por meio da diretiva #define que o retorno da função caso não encontre a palavra seja -1, já que não existem posições negativas em vetores.

```c
#define INDEX_NOT_FOUND -1

int searchElementIndex(hash* hashTable, char* string) {
    int index = hashGenerator(string);
    int counter = 0;
    while (counter < HASH_SLOTS) {
        // Slot is EMPTY, string was not found
        if (strcmp(hashTable[index].word, EMPTY) == 0) {
            return INDEX_NOT_FOUND;
        }

        // Slot has the string, return index
        if (strcmp(hashTable[index].word, string) == 0) {
            return index;
        }
```

```
            counter++;
            index++;
            index = index % HASH_SLOTS;
        }

        // Seached in all HASH_SLOTS but did not found
        return INDEX_NOT_FOUND;
    }
```

2.3 TESTES DE SOFTWARE

Desenvolver um software vai muito além de apenas produzir uma solução para um problema, é necessário que sejam realizados vários testes para melhorar ao máximo a experiência tanto do usuário como do programador, gerando uma aplicação performática e de fácil manutenção.

2.3.1 Fator de Carga

Ao definir o tamanho fixo de uma tabela é necessário “prever” quantos espaços serão utilizados a fim de reservar o suficiente para englobar a quantidade máxima de palavras esperadas. As orientações do trabalho definem que o texto de teste terá no máximo 256 palavras diferentes.

A principal exigência do trabalho é que a tabela hash gere o mínimo de colisões possível. Uma colisão ocorre ao tentar inserir um elemento no hash gerado pela função de espalhamento quando esse espaço já está em uso, ou seja, será necessário procurar outro local para inserir esse elemento. Quanto mais colisões ocorrem durante o preenchimento da tabela, maior será o consumo de CPU (processamento) e mais tempo levará para finalizar o programa.

Por esse motivo, não é indicado que o tamanho da tabela seja o mesmo da quantidade de palavras, pois considerando um caso extremo, no qual o texto é composto por exatamente 256 palavras diferentes, para armazenar as últimas palavras do texto, a probabilidade de gerar muitas colisões é demasiadamente alta.

Dessa forma, é necessário ter atenção com o fator de carga, definido pela razão entre o número de elementos pelo tamanho da tabela.

$$Fator\ de\ Carga = \frac{número\ de\ elementos}{tamanho\ da\ tabela}$$

Por se tratar de uma razão onde o numerador sempre será menor ou igual ao denominador, o fator de carga está definido entre 0 e 1. Quanto mais próximo de 0, mais o programa tende a ser O(1), tendo em vista que em quase todos os casos de inserção haverá um espaço disponível, também haverá um grande uso de memória. Já quanto mais próximo de 1, mais próximo de O(n) será, pois maior é a tendência de gerar colisões, por preencher a tabela mais facilmente. Portanto, um meio-termo desses dois valores é tornar o tamanho da tabela duas vezes maior que a quantidade de palavras.

Para realizar os testes aleatórios foi utilizado um gerador aleatório de strings, que possibilitou a criação de 256 strings aleatórias de 20 caracteres cada de maneira simples e rápida.

Em um teste aleatório foi possível analisar os diferentes totais de colisões variando de acordo ao tamanho da tabela. Com exatamente 256 espaços houveram 2743 colisões, mais de 10 vezes a quantidade de palavras, número bem expressivo, já colocando o dobro (512) esse número cai para 128, menos de 5% do total anterior.

Porém, testando com um tamanho de 509 espaços, a quantidade caiu para 102, a princípio esse resultado não faz sentido, porém analisando mais cuidadosamente, esse número é primo, o que faz com que a operação de resto da divisão resulte em menos valores iguais. A explicação desse fenômeno está relacionada à “Natureza da Matemática”, que exige grande conhecimento técnico para ser explicada, porém não se aplica à finalidade desta documentação.

2.3.2 Tempo de execução

...

3 CONCLUSÃO

Após a estruturação do funcionamento do programa, deve-se compilar os arquivos por meio do makefile e, posteriormente, passar os arquivos de texto pelo Terminal (a função main receberá os arquivos de texto como parâmetro). Dessa forma, se os arquivos forem passados corretamente, os parâmetros *argv* e *argc* da função principal contabilizarão o que foi passado e o programa será executado da forma esperada.

No fim da execução, o output do programa será, respectivamente, a quantidade de vezes que a palavra foi encontrada, a palavra que foi pesquisada e as linhas em que ela apareceu. Exemplo a seguir:

REFERÊNCIAS