Spell Checker

João Pedro Vasconcelos Teixeira
Engenharia de Computação
Centro de Informática, UFPB
João Pessoa, Paraíba
Email: joaoteixeira@eng.ci.ufpb.br
Github: https://github.com/jpvt

João Wallace Lucena Lins
Engenharia de Computação
Centro de Informática, UFPB
João Pessoa, Paraíba
Email: jwallace.lucena@gmail.com
Github: https://github.com/void-zero

Itamar de Paiva Rocha Filho
Engenharia de Computação
Centro de Informática, UFPB
João Pessoa, Paraíba
Email: itamardprf@gmail.com
GitHub: https://github.com/ItamarRocha

***Resumo*—Neste artigo, iremos discutir e expor os resultados de nosso projeto da disciplina de Estrutura de Dados. Nosso trabalho foi implementar um spell-checker por meio do uso uma tabela hash e de outros conhecimentos adquiridos durante as aulas. Por fim, iremos comparar nossos resultados e a eficiência de nosso código em relação à funções de hash nativas de outras linguagens de programação, no caso o Python e o C++.**

***Abstract*—In this article we are going to discuss and show the results of our project of Data structures class. Our work was to implement a spell-checker using a Hash Table and other knowledges that we learned in class. Furthermore, we will compare our results and the efficiency of our code with other native hash-checking functions contained in the Python and C++ programming languages.**

I. Introdução

De acordo com Cormen [1], uma *hash table* consiste em uma estrutura de dados eficaz para a implementação de um dicionário, este que consiste em um valor atrelado à uma chave correspondente. Porém, diferentemente de um dicionário comum, em uma *hash table* cada chave é mapeada através do uso de uma função *hash*, que tem por objetivo facilitar o armazenamento e acesso da informação referente a ela nos compartimentos da *hash table*, os *buckets* (Cada chave mapeada pela função de *hash* leva a um *bucket*).

Contudo, ao se tentar armazenar múltiplos dados, há a possibilidade de ocorrer um evento conhecido como colisão, em que a função de hash endereça duas chaves diferentes para um mesmo *slot*. Existem diferentes maneiras de tratar esse evento, e elas se dão de acordo com o objetivo, as limitações do sistema e com a eficiência requerida do programa.

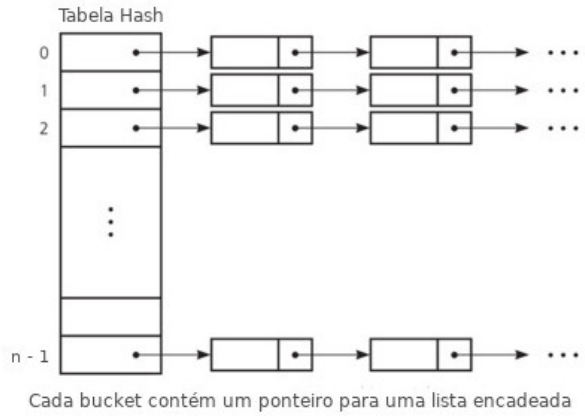

Figura 1. Representação visual do *chaining*

Dentre as formas mais comumente utilizadas para solucionar as colisões, estão: *chaining*, *linear probing* e *quadratic probing*. Assim, a primeira forma consiste no encadeamento das informações, isto é, no ”enfileiramento”delas em um *bucket*, a segunda, por sua vez, se resume a colocar cada dado em um determinado *bucket*, ou seja, cada bucket contém apenas um único elemento, e o *quadratic probing* é composto por um *linear probing* que passa por duas funções distintas de hash para gerar seu endereço.

O objetivo desse trabalho implementado em C é elaborar um corretor de texto baseado em um dicionário ASCII da língua portuguesa, através das ferramentas discutidas anteriormente.

II. Desenvolvimento

1) Tratamento de colisões e função de hash: Para alcançar nosso objetivo, fizemos uso da técnica de tratamento de colisões *chaining*, dado que é uma técnica com a qual já estamos mais familiarizados. Associado a isso, usamos da função de *hash* de nome *one_at_a_time* proposta por Bob Jenkins [2], que foi escolhida devido ao fato de ser uma função de fácil implementação, como também por sua notável velocidade e eficiência de distribuição, sendo ela optada ao invés de suas sucessoras de maior complexidade, a *lookup3* e *SpookyHash*, apresentadas pelo mesmo autor. A função apresenta técnicas de deslocamento de bits e por isso é tão eficaz, ela também é utilizada como a função de *hash* padrão da linguagem *Perl*. Segue abaixo a sua implementação em C :

```
uint32_t one_at_a_time(char * key) {
    size_t i = 0;
    uint32_t hash = 0;

    while (key[i] != '\0') {
        hash += key[i++];
        hash += hash << 10;
        hash ^= hash >> 6;
    }

    hash += hash << 3;
    hash ^= hash >> 11;
    hash += hash << 15;
    return hash % NUM_BUCKETS;
}
```

Listing 1. Código da função *one_at_a_time*

Para chegarmos à essa escolha, realizamos testes com a função *bucketScattering* para determinar o número de nós por *bucket*, e assim conseguir calcular uma média de itens em cada um deles. Dessa forma foi possível comparar diferentes funções de espalhamento, a partir de uma mesma entrada. Seguem os resultados obtidos nas funções *one_at_a_time* e uma baseado no método da multiplicação:

Como se pode ver, o *one_at_a_time jenkins* obteve uma distribuição consideravelmente mais uniforme, ao passo que a função baseada em *multiplication method* mostrou-se bem mais dispersa.

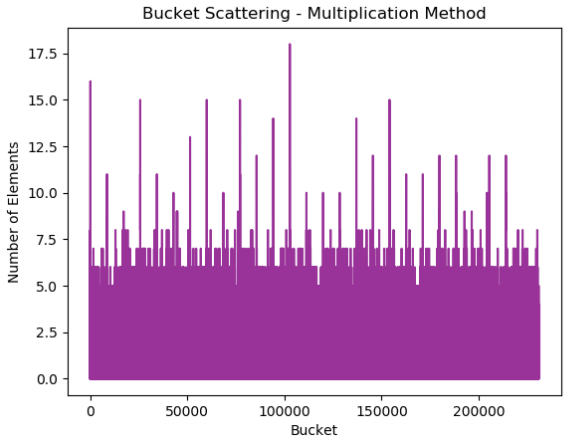

Figura 2. Gráfico de distribuição de palavras por bucket do método da multiplicação

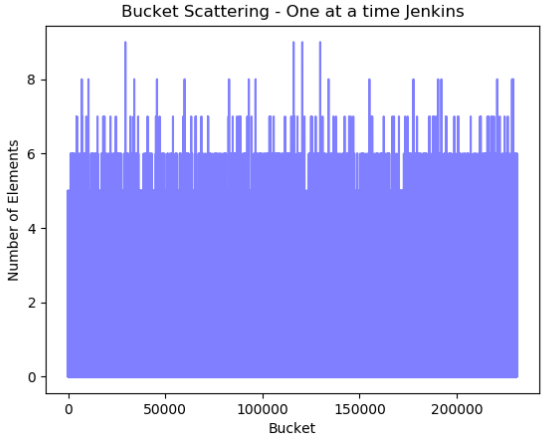

Figura 3. Gráfico de distribuição de palavras por buckets do método one at a time

2) Quantidade de buckets: Quanto à escolha do número de *buckets*, a fim de possibilitar que a pesquisa tenha uma complexidade próxima de *O(1)*, dado que foi escolhida a técnica de *chaining*, é necessário uma grande quantidade de *buckets* para que seja possível uma distribuição de aproximadamente dois elementos em cada um deles. No entanto, se aumentarmos exacerbadamente o número de *buckets*, a distribuição fica com um desvio médio alto e, portanto, o programa fica ineficiente, então para encontrar um bom número, fizemos testes comparando a média de elementos em cada *bucket*, desvio padrão, quantidade máximo e mínima de elementos nos *buckets*. Na tabela I é possível ver os resultados:

Tabela I
MEDIDAS DE CADA BUCKET

Nº de Buckets	Média	Desvio P.	Min	Max
153929	1,99	1,15	0	12
213497	1,42	0.83	0	9
215503	1,42	0.83	0	9
230891	1.33	0.79	0	9

Assim, com as informações obtidas, encontramos que 230891 buckets nos dava uma distribuição com o menor desvio padrão e a menor média entre os observados, com uma amplitude de 9 elementos por *bucket*, o que se equiparava aos outros. Portanto, optamos por escolher esse número devido ao seu desempenhos na prática quando comparado aos outros.

3) Armazenamento da informação e checagem: Com relação ao armazenamento dos dados, inserimos todas as palavras do dicionário na tabela *hash*, através da função *readDictionary* que abre o arquivo onde o dicionário está contido e lê cada linha, alocando elas em seu devido lugar através da função de *hash*. Assim, em primeiro lugar analisamos os dados para em seguida checar o arquivo fornecido.

Por conseguinte, foi usada a função *check* que tem como parâmetros um ponteiro para a tabela *hash*, um ponteiro para *char* que representa o arquivo fornecido para checagem no dicionário e um ponteiro para *tArray* que representa um array dinâmico que irá armazenar os erros, ou seja, as palavras não encontradas no dicionário.

Além disso, optamos pelo uso da função *strtok* da biblioteca *string.h* para ignorar caracteres como números, acentos e outros identificadores que não são considerados letras, de forma a viabilizar a comparação. Essa que é realizada através da aplicação da função de *hash* sobre a chave, referente à cada palavra lida no arquivo, e, logo após, é feita sua comparação com todas as palavras contidas dentro do bucket endereçado. Dessa maneira, quando não encontrada a palavra é armazenada no *array* de erros para posteriormente ser escrita no arquivo de relatório. Segue abaixo o código para uma melhor visualização.

```
int check(tHashTable* t, char* directory, tArray*
    errors){
    FILE *fp = fopen(directory, "r");

    if(fp == NULL){
        printf("Erro ao abrir o arquivo\n");
        return -1;
    }

    tNode* cursor;
    char *string, recebido[500], found,
        *token = " \n\r\t!\"#$%&()
    *+,./0123456789:;<=>?@[\\]^_`{|}~";
    int numberOfWords = 0;

    while(!feof(fp)){
        fgets(recebido,500,fp);
        recebido[strlen(recebido) - 1] = '\0';
        string = strtok(recebido, token);
        while(string != NULL){
            numberOfWords++;
            cursor = t->buckets[h(string)];
            found = 0;
            while(cursor != NULL){
                if(!strcmp(cursor->value, string)){
                    found = 1;
                    break;
                }
                cursor = cursor->next;
            }
            if(!found)
                push(errors, string);
            string = strtok(NULL, token);
        }
    }

    fclose(fp);
    return numberOfWords;
}
```

Listing 2. Código da função *check*

4) Verificação do tempo: Quanto ao quesito de cronometragem, o método usado para aferir tal medida foi feito através da criação de uma versão de *bechmark* dos programas, isto é, uma versão que só mantém as funcionalidades relacionadas aos processos de *hashing* e de *checking*. Para cada uma das linguagens, temos:

- **C:** Fizemos uso da função *clock()*, que, de acordo com *C++ Reference* [3], tem como valor de retorno o tempo de processamento consumido pelo programa, presente na biblioteca *time.h*;
- **C++:** Utilizamos de variáveis *high resolution clock* e da função *duration_cast*, contidos na biblioteca *chrono*;
- **Python:**Empregamos a biblioteca nativa *time* e utilizamos a função *clock()* que tem como retorno o tempo de processamento consumido pelo programa

Com isso, foi medido a diferença da quantidade de *clock ticks* antes e após da chamada da função *check* e, a partir daí, dividimos o valor obtido pela quantidade de ciclos de *clock* por segundo, e por fim conseguimos determinar o tempo em milissegundos, de forma a garantir uma precisão satisfatória.

5) Armazenamento das informações em um arquivo .txt: Como especificado na descrição do projeto, o armazenamento das informações deveria seguir certo padrão. Portanto, criamos uma função *archiveRelatory*, que recebe como parâmetros os dados exigidos pelo *layout* proposto e tem por objetivo escrever em um arquivo os dados passados, através da função *fprintf()*.

III. Conclusão

Após terminarmos a implementação do *spell-checker* em C, fizemos uma versão em C++ e uma versão em Python usando as funções de *hash* padrões das respectivas linguagens. O nosso algoritmo, quando aplicado à Constituição brasileira de 1988, em formato ASCII (usado como padrão de comparação), ao ser rodado em um computador com o processador Intel© Core™ i7-8550U, apresentou um tempo aproximado de:

$$Tempo : 10.4(ms)$$

para checar todas as palavras. Enquanto isso, o códido em python, na mesma máquina, apresentou um tempo de:

$$Tempo : 335(ms)$$

e o em C++, por sua vez, apresentou:

$$Tempo : 18(ms)$$

Como foi observado anteriormente, nossa função *check()* aliada a função *hash* mostraram um desempenho superior quando comparados ao padrão da linguagem C++ e do Python, obtendo uma margem extra de aproximadamente 8 ms e 325 ms de diferença entre os dois códigos, respectivamente. Como confirmação, percebe-se que nossa função de espalhamento conseguiu atingir uma complexidade de checagem próximo ao constante para o dicionário usado, o que é de assaz importância na implementação de um corretor. Além disso, nossa distribuição apresentou um desvio padrão de 0.79 unidades e uma média de 1.33 palavras por bucket, o que comprova que a distribuição ocorreu de forma equilibrada.

Assim, Após analisar uma versão em ASCII da Constituição Federal que possui 51781 palavras - tomando como base o dicionário fornecido - imprimimos o seguinte relatório :

Numero total de palavras do texto: 57181
Tempo total de verificação: 16.9850 ms
Numero de palavras que falharam no spell check 23967
Lista de palavras que falharam no spell check

Num.Ocorrencia - Palavra

---

1 - CAMARA
2 - DOS
3 - DEPUTADOS
...

Vale salientar que o tempo exibido no relatório difere do tempo utilizado na comparação entre os códigos, pois como citado anteriormente na subsessão 4 do Desenvolvimento, foi-se utilizado uma versão de *benchmark* dos programas.

Referências

[1] T. H. Cormen *et al.*, *Introduction to Algorithms*, 3rd ed. The MIT Press, 2009.
[2] R. J. J. Junior. (2006) A hash function for hash table lookup. [Online]. Available: http://www.burtleburtle.net/bob/hash/doobs.html
[3] [Online]. Available: http://www.cplusplus.com/reference/ctime/clock