Uma aplicação sobre o SpellChecker

Danilo Henrique da Silva Santana
Universidade Federal da Paraíba
Centro de Informática
João Pessoa, Paraíba, Brasil
Email: danilosantana@mat.ci.ufpb.br

Ítalo Nicácio
Universidade Federal da Paraíba
Centro de Informática
João Pessoa, Paraíba, Brasil
Email: italonicacio@mat.ci.ufpb.br

Vinicius dos Santos
Universidade Federal da Paraíba
Centro de Informática
João Pessoa, Paraíba, Brasil
Email: vncssnts2016@hotmail.com

***Resumo*—O trabalho consiste na implementação de um Spell-Checker, onde será utilizado uma tabela hash e o encadeamento foi escolhido como solução de colisão. Utilizando a linguagem C, comparamos o tempo de pesquisa com a tabela hash pronta do C++.**

1. Introdução

O Spell checker é um programa que testa as palavras de um texto contra um dicionário. Se uma determinada palavra do texto é encontrada no dicionário, presume-se que ela está escrita corretamente. Se a palavra não é encontrada no dicionário, considera-se que ela esteja escrita de forma incorreta ou que o dicionário em questão ainda não a contém.

2. Tabela Hash

A tabela hash é uma técnica de programação para se implementar um dicioniário, onde a inserção é constante pois, basta calcular o valor de hash da chave e inserir naquela posição. O tempo para pesquisar depende da função de hash escolhida, pois se deseja inserir m elementos na tabela em n quantidade de buckets o número de colisões geradas é $O(\frac{m}{n})$.

3. Função de Hash

Cada chave tem a mesma probabilidade de hash para qualquer um dos n buckets, independentemente de onde as chaves tenham sido divididas. Infelizmente, na maioria dos casos não temos como verificar essa condição, pois raramente sabemos a distribuição de probabilidade da qual as chaves são espalhadas. Além disso, as chaves não podem ser espalhadas independentemente, por isso implementamos funções de hash.

A função de Hash utilizada no trabalho foi a **djb2**, uma das melhoras funções de hash para strings pois garante um bom espalhamento das palavras na tabela hash.

Seu funcionamento se da na seguinte forma: É escolhido um número primo 5381 que, a cada iteração, é somado ao seu valor multiplicado por 33 e somado pelo valor de ascii da string.

Na figura 1 é mostrado a implementação da função de hash na linguagem C.

Figura 1. Função de Hash djb2.

```c
unsigned int Hash(char* word) {

    int size = strlen(word);
    int i;
    unsigned int value = 5381;

    for(i = 0; i < size; ++i) {

        value += (value << 5)  + word[i];

    }

    return value;
}
```

Existe um problema no momento em que o hash da palavra é calculado, duas palavras podem cair no mesmo bucket. Nós chamamos essa situação de colisão. Felizmente, existem técnicas para solucionar este problema. A técnica utilizada no trabalho foi por encadeamento.

4. Encadeamento

Por encadeamento, nos colocamos todos os elementos que tem o mesmo valor de hash em uma lista simplesmente encadeada. O procedimento de inserção é rápido em parte pois, o primeiro elemento a ser inserido não gera colisão. Para pesquisar, o pior caso é proporcional ao tamanho da lista, no entanto, dizemos que no pior caso a inserção é O(1) e pesquisa é O(n).

As vantagens dessa técnica é que podemos inserir quantos elementos quisermos na tabela e não precisa necessariamente da melhor função de hash.

As desvantagens é que ocupa uma boa quantidade de memória, já que para cada nó criado um espaço na memória é alocado e também dependendo da quantidade de buckets que escolhermos pode haver desperdício de memória.

A implementação em C da função de inserção e pesquisa por encadeamento pode ser vista nas figuras a seguir.

Figura 2. Inserção encadeada.

```
void ChainedInsert(HashTable h, char* word) {

    int index = (Hash(word) % length);

    struct Sll* list = NULL;

    if(h[index] == NULL) {

        list = CreateNode();

        h[index] = list;

        Prepend(list, word);

    }else {

        list = h[index];

        Prepend(list, word);

    }

}
```

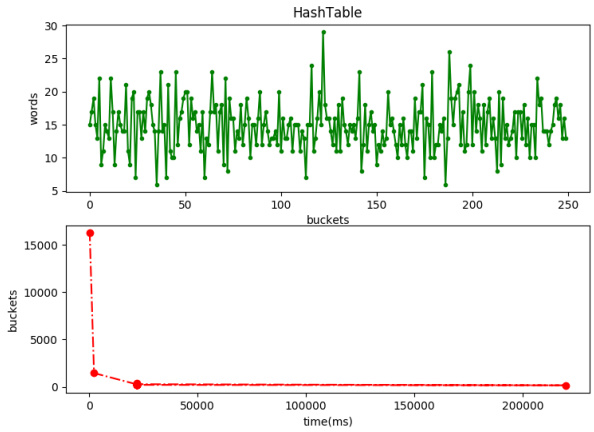
Toda implementação pode ser encontrada no repositório do GitHub. [3]

Figura 3. Pesquisa encadeada.

```
int ChainedSearch(HashTable h, char* word) {

    int index = (Hash(word) % length);

    if(h[index] != NULL) {

        const struct Sll* list = h[index];

        const struct Node* aux = list->head;

        while (aux != NULL) {

            if((strcmp(aux->word, word)) == 0) {

                return 1;
            }

            aux = aux->next;

        }

    }

    return 0;

}
```

Podemos observar na figura 4 que no primeiro gráfico o **eixo y** são as palavras e no **eixo x** são os buckets. Como estamos usando uma boa função de hash, as palavras vão ficando bem distribuídas. No segundo gráfico temos no **eixo y** a quantidade de buckets e no **eixo x** o tempo em milisegundos. A medida que a quantidade de buckets vai aumentando, o tempo de pesquisa se estabiliza depois de um tempo. Utilizando como dicionário um arquivo que contém 307855 palavras e como texto a constituição federal, com **22000 buckets** o tempo de pesquisa foi **103.92ms**.

Figura 4. Distribuição das palavras nos buckets e variação de pesquisa com a quantidade de buckets.



5. Extra

Como extra implementamos uma tabela hash em C++ utilizando a **unordered_map** que são contêineres associativos que armazenam elementos formados pela combinação de um valor de chave e um valor mapeado.

Internamente, os elementos no **unordered_map** não são classificados em nenhuma ordem específica em relação a seus valores-chave ou mapeados, mas organizados em intervalos, dependendo de seus valores de hash, para permitir acesso rápido a elementos individuais diretamente por seus valores-chave.

Utilizando como dicionário o mesmo arquivo, que contém 307855 palavras, e como texto a constituição federal, analisando o tempo de pesquisa obtivemos **37ms**.

Logo, pegando o melhor caso de encadeamento que foi **103.92ms**, a tabela hash implementada em C++ foi mais rápido em **66,92ms**, mesmo sendo uma tabela hash genérica, foi quase duas vezes mais rápido do que a implementada em C.

6. Conclusão

Portanto, implementamos o spellchecker utilizando uma tabela hash, através da técnica de colisão por encadeamento analisamos a quantidade de buckets em relação ao tempo e a importância de uma boa função de hash. Com isso, obtivemos um tempo bom de pesquisa, no entanto, a tabela hash do C++ foi muito mais eficiente.

Referências

[1] CORMEN, Thomas,. et al. Introduction to Algorithms 3ed.

[2] SKIENA, Steven. The Algorithm Design Manual.

[3] https://github.com/danhenriquex/SpellChecker

[4] http://www.cplusplus.com/reference/unordered_map/unordered_map/