Trabalho desenvolvido por:

Vanessa Gabriele Lima Pessoa Matrícula: 20170159710

Arthur Ricardo Ribeiro Lopes Matrícula: 20170033039

Introdução

Este trabalho tem como objetivo desenvolver uma spellchecker utilizando tabela hash. 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.

Função de Hash

Uma função de hash tem como objetivo transformar o que vai ser guardado em um valor inteiro para daí determinar sua posição na tabela hash.A função de hash utilizada no desenvolvimento deste trabalho foi uma função chamada djb2 que é uma das melhores funções de hash para palavras. Seu funcionamento é simples, o valor de hash é inicializado em um número primo grande(normalmente 5381) depois disso o valor de hash será este número vezes 33 adicionado o valor da letra em ASIIC isso para cada letra da palavra.

Observe a implementação dela abaixo:

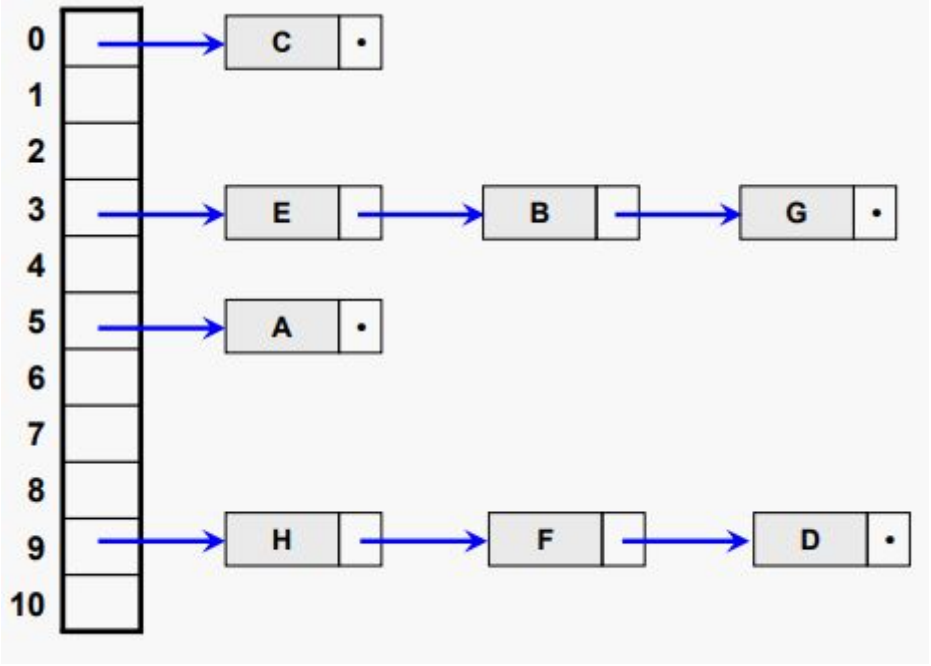
```
unsigned int HashTable::hash(char *word){


    int size = strlen(word);

    unsigned int value = 5381;//

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

    value = value * 33  + word[i];

    }

    return value;
};
```

Quando vamos inserir um novo elemento na nossa tabela depois de calcularmos o valor de hash para definir o local onde esse elemento vai ficar tiramos o mod (resto da divisão inteira)do valor de hash de acordo com a quantidade de buckets que a nossa tabela tem, por mais que nossa função de hash seja boa, por temos um número limitado de buckets dependendo da quantidade de itens que queremos inserir quase sempre vai gerar colisões ,ou seja, itens caindo no mesmo bucket, para tratarmos essas colisões temos várias maneiras.

Tratamento de colisão

Encadeamento (chaining)

Encadeamento é uma das maneiras mais simples de resolução de colisão. Com esse tratamento cada elemento da tabela de hash é uma lista encadeada. Para inserir um item na tabela basta inserir esse item na lista encadeada específica caso tenha colisão todos esses itens ficaram na mesma lista. Na hora de pesquisar basta ir na lista correspondente daquele item e olhar se ele está lá ou não.



O custo de inserção é sempre contínuo pois basta definir o valor de hash e dar um prepend neste item na lista, já o custo de pesquisa varia com a função de hash utilizada, se ele distribui bem ou não os itens, e com a quantidade de buckets. Se tivermos uma boa função de hash a pesquisa terá uma tempo de O(M/N) onde M é a quantidade de itens inseridos, N é a quantidade de buckets.

Implementação da inserção e pesquisa usando encadeamento utilizando C++:

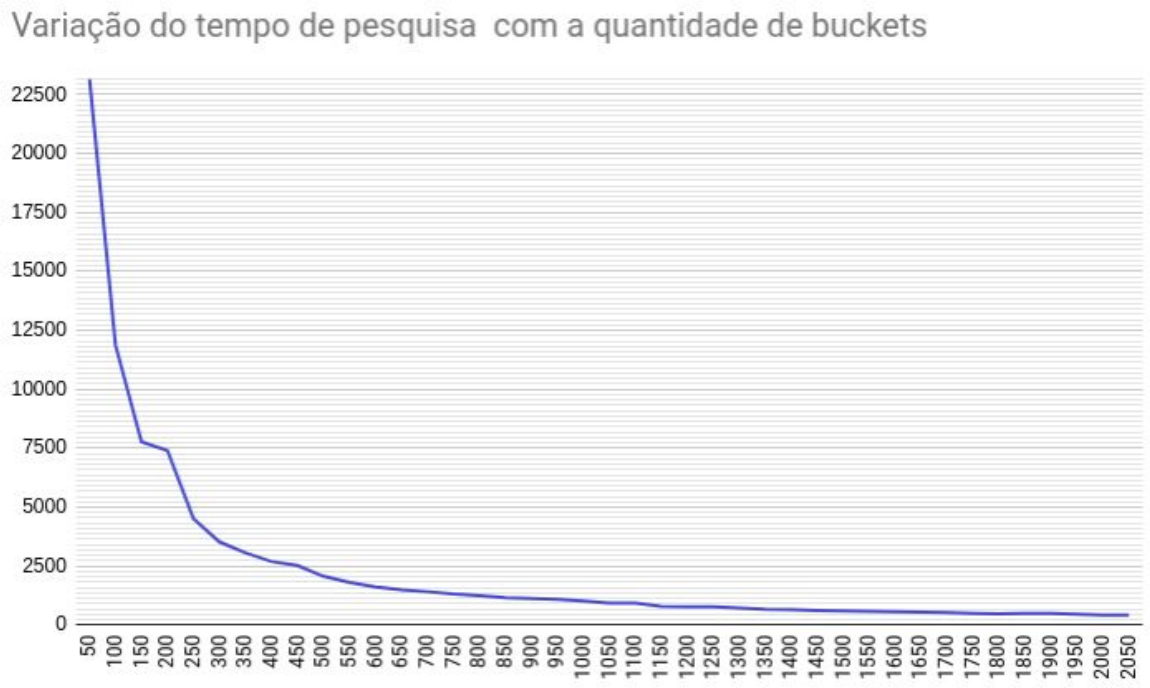
Inserção:

```
void HashTable::insert(char *word){

    LinkedList &l = this->table[(this->hash(word)) % MAX];
    l.prepend(word);
}
```

Pesquisa:

```
bool HashTable::search(char *word){

    LinkedList &L = this->table[(this->hash(word)) % MAX];

    Node *aux = L.getHead();

    while(aux != NULL){

        if(strcmp(aux->word,word) == 0){
            return true;
        }
        aux = aux->Next;
    }

    return false;
}
```

Usando o código acima para carregar o dicionário cedido pelo professor e checar um total de palavras variando a quantidade de buckets e contando apenas o tempo que levou para pesquisar as palavras os resultados foram os seguintes:



No eixo x temos a quantidade de buckets e no eixo y temos o tempo em milisegundos.

Como podemos ver com o gráfico depois de aproximadamente **1600 buckets** o tempo de pesquisa começa a estabilizar por volta de **500 ms** ,isso porque a função de hash é muito boa e espalha bem as palavras nos buckets.

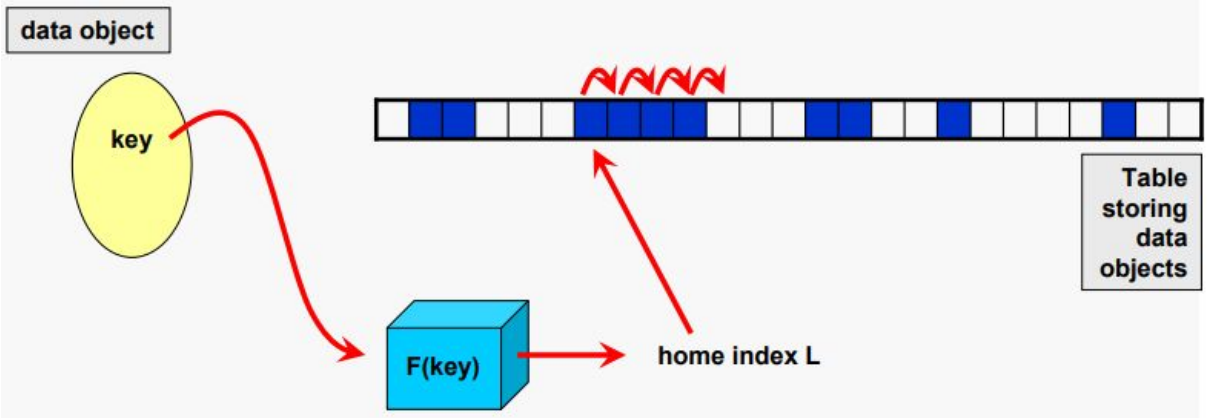
Linear probing

O endereçamento aberto linear é uma técnica de tratamento de colisões onde todos os itens são guardados em um array porém caso haja uma colisão o valor de hash desse item irá antes do mod ser incrementado de um em um até que o índice se refira a um bucket vazio, e na hora da pesquisa a mesma coisa depois de calcular o valor de hash e tirar o mod se aquela posição for uma posição ocupada e não for aquele item terá que ser feito o incremento do mesmo jeito e comparar até achar aquele item e confirmar que ele está lá ou achar um bucket vazio e assim confirmar que o item não está nessa tabela.

Matematicamente a fórmula que define próximo índice ficaria assim:

$$Next(k) = (F(key) + k)\%TableSize$$

Exemplo de inserção:



Para a utilização desse modo de tratamento de colisão o número de itens a serem inseridos nesta tabela tem que ser menor ou igual ou número de buckets

Quanto ao custo da inserção e da pesquisa não dá para atribuir um valor pois seria constante caso não tenha nenhuma colisão ,mas como é difícil não ocorrer colisões o custo passa a ser uma probabilidade de alguém tá ocupando ou não o bucket correspondente daquele item e os seus subsequentes.

Implementação da inserção e pesquisa usando linear probing usando C++:

Inserção:

```
void HashTable::linearInsert(char *word){
    int i = 0;
    unsigned int hash = this->hash(word);
    Node *N1 = new Node();
    N1->word = word;


    if( this->table[(hash + i)%MAX] == NULL)
        this->table[(hash + i)%MAX] = N1;
    else{
        while(this->table[(hash + i)%MAX] != NULL){
            i++;
        }
        this->table[(hash + i)%MAX] = N1;
    }
};
```

Pesquisa:

```
bool HashTable::linearSearch(char *word){
    int i = 0;
    unsigned int hash = this->hash(word);

    if( this->table[(hash + i)%MAX] == NULL){
        return false;

    }else{
        while(this->table[(hash + i)%MAX] != NULL){

            if(strcmp(word,this->table[(hash + i)%MAX]->word) == 0)
                return true;
            else
                i++;

        }
        return false;
    }
};
```

utilizando o dicionário cedido pelo professor que tem aproximadamente **308 mil palavras** no código apresentado ,comparando a constituição brasileira o resultado foi o seguinte:

Utilizando o dicionário cedido pelo professor que tem aproximadamente **308 mil palavras** no código apresentado ,comparando a constituição brasileira(**83.763 palavras**) o resultado foi o seguinte:

Utilizando **310 mil buckets** é necessário **34000 ms** e o tempo estabiliza com **370 mil buckets** com um tempo de **29 ms.**

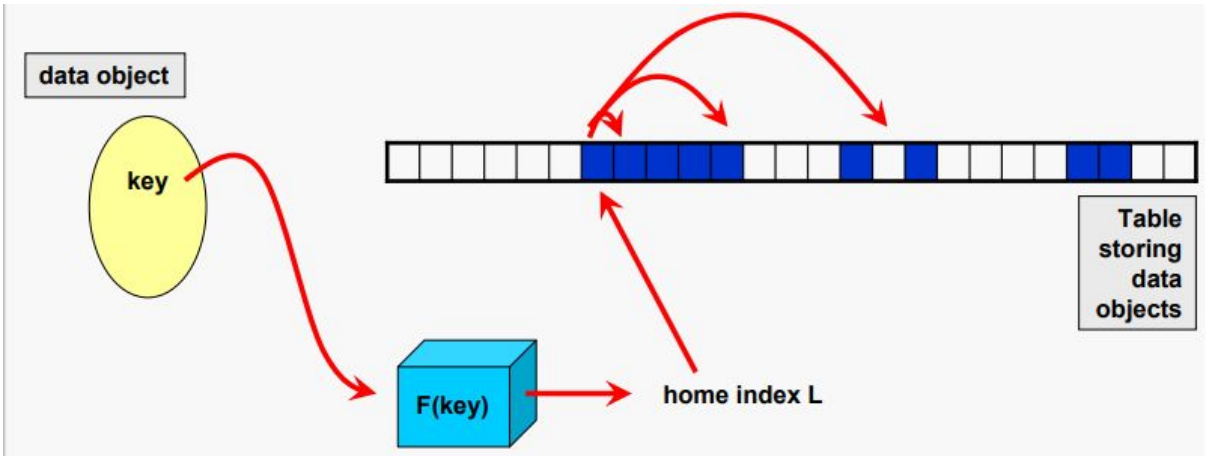
Quadratic probing

O quadratic probing tem basicamente o mesmo funcionamento do Linear probing, mas agora o incremento é feito não mais de um em um e sim com o quadrado do próximo valor de incremento.

Neste caso a fórmula que define o valor da próxima índice fica assim:

$$Next(k) = (F(key) + k^2)\%TableSize$$

Exemplo de inserção usando **quadratic probing:**



Quanto ao custo ,é mesma coisa da linear probing também não existe um custo estabelecido e sim uma probabilidade do próximo índice está ocupado ou não.

Implementação da inserção e pesquisa usando quadratic probing usando **C++:**

Inserção:

```
void HashTable::quadraticInsert(char *word){
    int i = 0;
    unsigned int hash = this->hash(word);
    Node *N1 = new Node();
    N1->word = word;


    if( this->table[(hash + (i*i))%MAX] == NULL)
        this->table[(hash + (i*i))%MAX] = N1;
    else{

        while(this->table[(hash + (i*i))%MAX] != NULL){
            i++;
        }
        this->table[((hash + (i*i))%MAX)] = N1;
    }
};
```

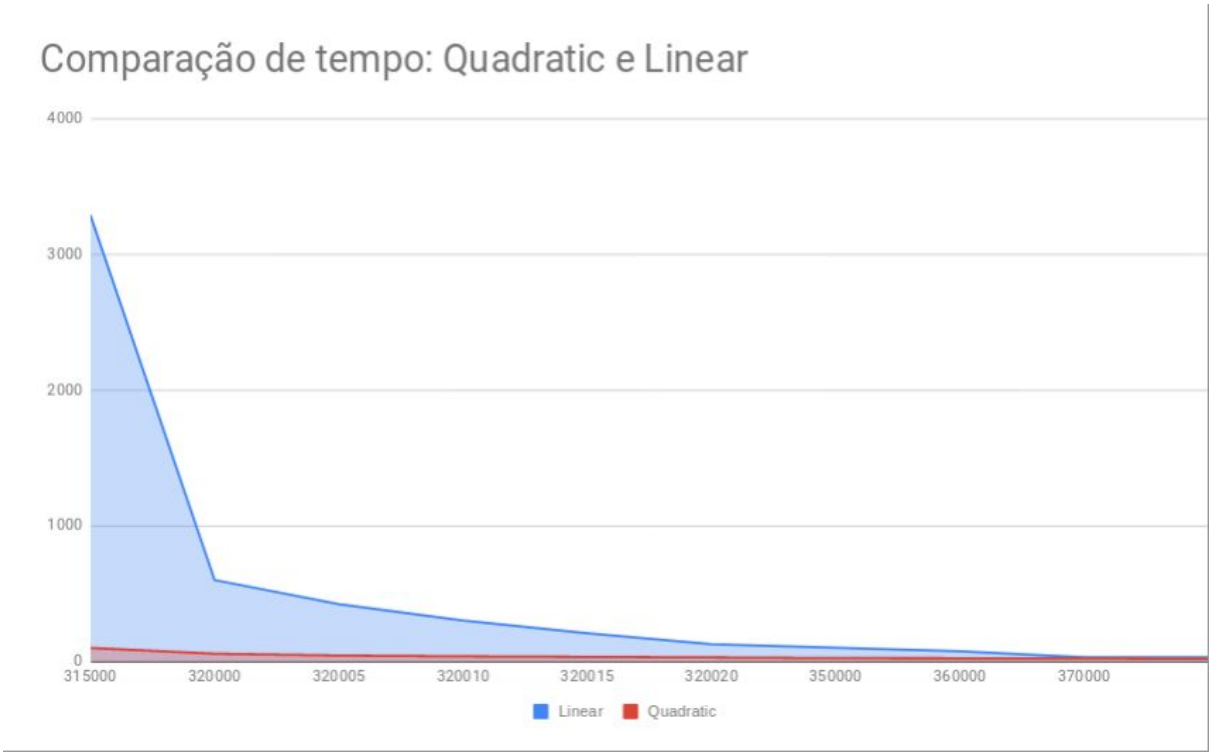
Pesquisa:

```
bool HashTable::quadraticSearch(char *word){
    int i = 0;
    unsigned int hash = this->hash(word);

    if( this->table[(hash + (i*i))%MAX] == NULL){
        return false;
    }else{
        while(this->table[(hash + i*i)%MAX] != NULL){

            if(strcmp(word,this->table[(hash + i*i)%MAX]->word) == 0)
                return true;
            else{
                i++;
            }
        }
        return false;
    }
};
```

Utilizando o mesmo dicionário do linear para comparar o mesmo texto com o quadratic resultado foi o seguinte:

Utilizando **310 mil buckets** é necessário **312 ms** , para atingir o resultado onde o linear é estabilizado que é de **29 ms** são necessários apenas **340 buckets**,já o quadratic estabiliza por volta do de **370 mil buckets** com um tempo de **18 ms**.

Comparando a quadratic com a linear em relação ao tempo de pesquisa em função do número de buckets temos o seguinte resultado:



como podemos ver pelo e pelos números apresentados a quadratic é bem superior a linear começando mais rápida e permanecendo assim com o aumento do número de buckets.

Extra

Para a atividade extra do trabalho foi escolhida a linguagem **java** para ser implementado um spellchecker utilizando a **tabela hash nativa** da linguagem.

Implementação em Java

Foi utilizado um HashMap para armazenar os dados do dicionário. HashMap trabalha com o conceito de Key- value, ou seja, cada elemento de sua lista possui uma chave e valor associado. Assim podemos realizar uma busca rápida através da chave, sem precisar percorrer toda a lista. A implementação de HashMap fornece desempenho em tempo constante para operações básicas get() e put(), assumindo que a função hash dispersa os elementos corretamente entre os buckets. A classe HashMap é equivalente a Hashtable, exceto que ela não está sincronizada e permite valores nulos. Observe a implementação do nosso código abaixo:

```java
private static HashMap<String, String> dicionario = new HashMap<>(5381);

/**
 *
 * @throws FileNotFoundException
 * @throws IOException
 */
public static void criarDicionario() throws FileNotFoundException, IOException {

    Scanner dic = new Scanner(new FileReader("dicionario.txt"));
    while (dic.hasNext()) {
        String palavra = dic.nextLine();
        dicionario.put(palavra, palavra);
    }
    System.out.println("Tamanho do dicionario: " + dicionario.size() + " palavaras.");
}
```

Adicionamos o arquivo dentro de um ArrayList de String, que implementa todas as operações de lista opcionais e permite todos os elementos, além de implementar a interface List, a classe ArrayList fornece métodos para manipular o tamanho da matriz que é usada internamente para armazenar a lista diferentemente de um array.

```java
//adicionar o texto no ArrayList
ArrayList<String[]> text = new ArrayList<>();
BufferedReader br = null;
try {
    FileReader fr = new FileReader("texto.txt");
    br = new BufferedReader(fr);

    String str;
    while ((str = br.readLine()) != null) {
        text.add(str.split(" "));
    }
} catch (IOException e) {
    System.out.println("Arquivo não encontrado!");
} finally {
    br.close();
}
System.out.println(text.size());
```

Então percorremos todo o ArrayList e verificamos se cada elemento está no dicionário. Utilizamos o próprio valor de cada elemento como chave.

```java
//pesquisar as palavras do texto no dicionario
long tempoInicial = System.currentTimeMillis();
text.stream().forEach((s) -> {
    for (String item : s) {
        if (dicionario.get(item) == null) {
            System.out.println(item);
        }
    }
});
long tempoFinal = System.currentTimeMillis();
System.out.println( tempoFinal - tempoInicial + "ms" );
```

Resultados:

O dicionário utilizado foi o mesmo dado pelo professor, contendo **307855 palavras**. O texto usado para comparar com o dicionário contem **83.763 palavras**. O tempo que levou para percorrer o HashMap sem exibir mensagem das palavras que falharam, analisando apenas o tempo que leva para percorrer todo o dicionário:

```
run:
Tamanho do dicionario: 307855 palavaras.
Tempo total de verificação:94ms
CONSTRUÍDO COM SUCESSO (tempo total: 1 segundo)
```

O tempo que levou para percorrer o HashMap exibindo mensagem das palavras que não foram encontradas no dicionário:

```
-
VirgÍlio
TÁvora.
Tempo total de verificação:  2496ms
CONSTRUÍDO COM SUCESSO (tempo total: 3 segundos)
```

Pegando o melhor caso da implementação em **c++** e comparando com a implementação em java , a do **c++** é mais rápida com um tempo de **18 ms** contra um tempo de **94 ms** do **java** 5 vezes mais rápida, mas isso se deve ao fato que na implementação em **c++** foi utilizada uma função de hash ótima para string , já no **HashMap()** do **java** é utilizado um função de hash mais geral, o'que pode deixá-la mais lenta em alguns casos.

Referências:

Slides do Professor CHRISTIAN

https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html

https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html

https://en.wikipedia.org/wiki/Linear_probing

https://en.wikipedia.org/wiki/Quadratic_probing

https://www.geeksforgeeks.org/hashing-set-1-introduction/