UNIVERSIDADE FEDERAL DO MARANHÃO – UFMA

DEPARTAMENTO DE INFORMÁTICA – DEINF

CURSO DE CIÊNCIA DA COMPUTAÇÃO

ESTRUTURA DE DADOS II

MARIA JUCIMARA PEREIRA FERREIRA

2018018368

RELATÓRIO

2° TRABALHO DE IMPLEMENTAÇÃO

ALGORITMOS DE PESQUISA

SÃO LUÍS – MA

2020

SUMÁRIO

1. ENUNCIADO

Nesta atividade você terá que implementar um programa para resolver o seguinte problema. Dado um conjunto P de palavras-chave e um conjuntos T de twittes que possuem essas palavras-chave, você deverá gerar uma lista de termos distintos, associados a palavra-chave, com o número de ocorrências de cada termo nos twittes.

2. INTRODUÇÃO

Algoritmos de busca verificam se uma dada informação ocorre em uma sequência ou não. Por exemplo, dado um conjunto de palavras guardada em uma lista e uma palavras x, escreva uma função/método que responda à pergunta: x ocorre encontra-se na lista?

Existe uma grande variedade de métodos de pesquisa e a escolha do método ideal para determinada aplicação depende de vários fatores, dentre esses, a primeira a ser analisada é quantidade de dados envolvidos, logo em seguida, se o arquivo estará sujeito a inserções e retiradas frequentes.

O melhor algoritmo de pesquisa, dado um conjunto de dados, será aquele com o menor tempo de processamento, levando-se em consideração não somente o tempo que leva para retornar dada informação, mais também seu tempo para inserir e/ou remover determinado dado.

3. ALGORITMOS DE PESQUISA

3.1. Hashing

A função de hashing espalha as chaves pela tabela de hash, que é usada para mapear um determinado valor com uma chave específica para acesso mais rápido de elementos. A eficiência do mapeamento depende da eficiência da função hash utilizada.

O encadeamento externo utiliza uma área extra, além da tabela hash
Todos os registros são armazenados em uma lista encadeada fora da tabela hash.

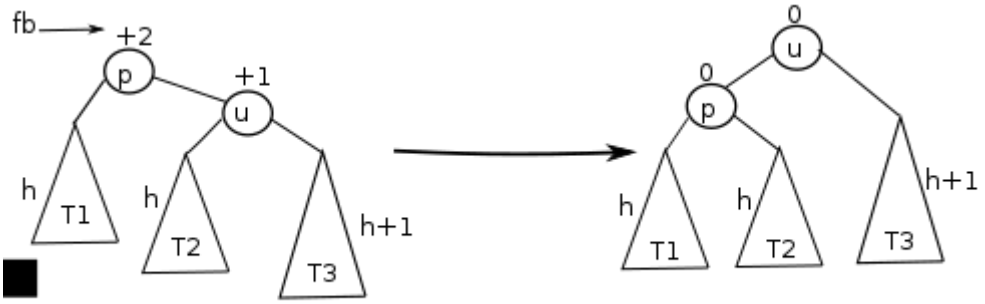
3.2. Árvore Balanceada

Árvores são estrutura de dados não lineares, formadas por um conjunto de dados(elementos) que armazenam informações de forma hierárquica denominadas como nós. Dos 3 tipos de árvores visto no curso, optei por implementar a AVL.

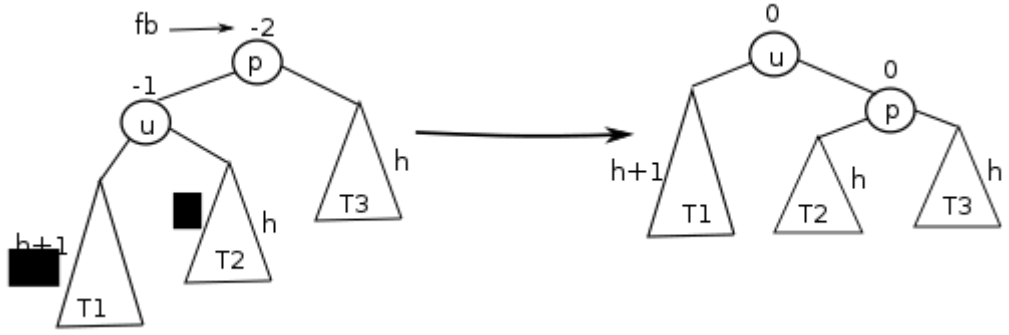
Criada em 1962 pelo soviéticos Adelson Velsky e Landis a árvore AVL é uma árvore binária de pesquisa balanceada, na qual cada elemento é chamado de nó e cada nó armazena uma chave e dois ponteiros, uma para a subárvore esquerda e outro para a subárvore direita.

Para a árvore se manger balanceada após cada inserção é preciso obedecer algumas regras de altura da árvore, Toda vez que um nó é inserido é preciso verificar o fator de balanceamento e fazer a rotação necessária.

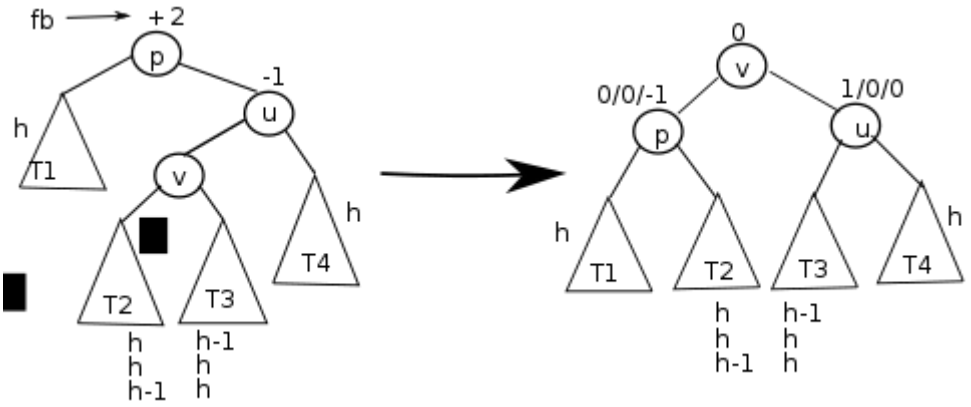
Rotação simples à esquerda



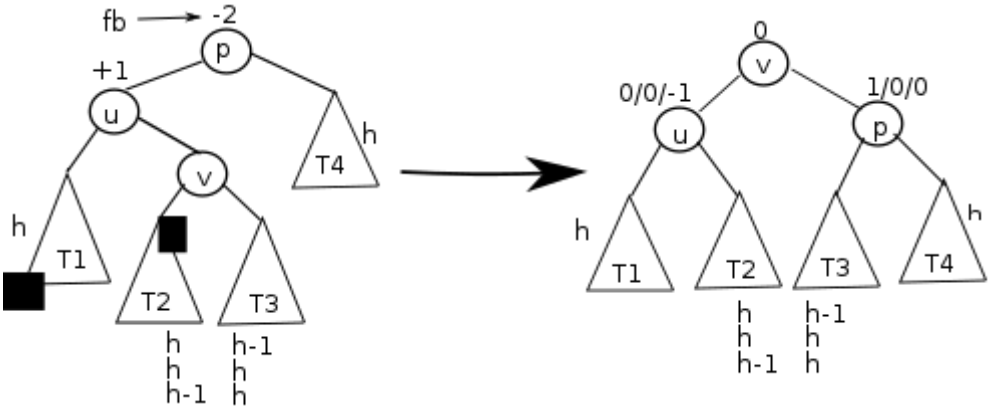
Rotação simples à direita



Rotação dupla à esquerda



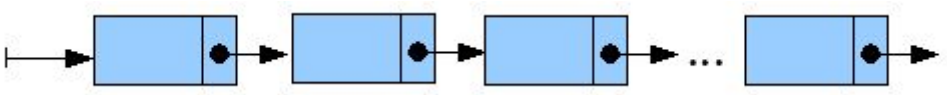
Rotação dupla à direita



3.3. Lista

Uma das formas mais simples de armazenamento de dados. As listas são estruturas muito flexíveis, permitindo a manipulação de uma grande quantidade de dados, elas podem implementar uma coleção ordenada de dados, onde o mesmo dado pode ocorrer mais de uma vez.

lista linear simplesmente encadeada



Feia por meio de um estrutura auto-referenciada, cada item da lista contém os dados necessários para alcançar o próximo item.

3.4. Análise de Complexidade

- Hashing

N representa o número de registros na tabela. M representa o tamanho da tabela.

	Melhor Caso	Pior caso
Inserção	O(1)	O(1 + N/M))
Busca	O(1))	O(1 + N/M)

- Árvore

	Média	Pior Caso
Espaço	O(n)	O(n)
Busca	O(log n)	O(log n)
Inserção	O(log n)	O(log n)
Deleção	O(log n)	O(log n)

- Lista

	Melhor Caso	Pior caso
Inserção	O(1)	O(n)
Busca	O(1)	O(n)

4. BASE DE DADOS

4.1. Twitter Developer

“O Twitter API v1.1 foi lançado em 2012 e permite que você publique, interaja e recupere dados para recursos como Tweets, Usuários, Mensagens Diretas, Listas, Tendências, Mídia e Lugares.”

Para ter acesso ao Twitter developer foi necessário o preenchimento de um questionário explicando o motivo do pedido de acesso além dos dados da instituição de ensino. A assinatura usada foi a Padrão.

Níveis de assinatura v1.1

Tweets	Padrão	Premium	Empresa
Publicar e engajar	Get started ›		
Tweets de pesquisa: 7 dias	✓		
Tweets de pesquisa: 30 dias		✓	✓
Tweets de pesquisa: arquivo completo		✓	✓
Tweets de filtro	✓		✓
Tweets de exemplo	✓		✓
Tweets em lote			✓

4.2. Tweepy

Uma biblioteca para acessar a API do Twitter. Não foi necessário nenhum cadastro, apenas instalação de alguns componentes e seguir a documentação da mesma.

Instalação

```
pip install tweepy
cd tweepy
pip install .
```

ou

```
git clone https://github.com/tweepy/tweepy.git
cd tweepy
pip install git+https://github.com/tweepy/tweepy.git
```

5. IMPLEMENTAÇÃO DO CÓDIGO

5.1 Pacotes: Java

5.1.2. Principal

O pacote principal contém 4 classes, são elas: Ler, Main, Menus e Processamento.

Ler: classe de leitura de entrada para deixar o algoritmo mais limpo

Main: class principal, responsável pelo funcionamento do programa

Menu: mensagens e menu de opções

Processamento: responsável por abrir o arquivo de leitura com os tweets coletados e saída com os algoritmos de pesquisa pedido.

5.2.3. Hashing

Foi implementado uma tabela hashing com encadeamento externo por uma lista

```
public class HashingEncadeado
```

Para definir onde cada palavra iria ser inserida, utilizei o alfabeto, sendo que cada posição faz referência a uma lista

```
private Lista[] HTable = new Lista[26];

    public HashingEncadeado() {

        for (int i = 0; i < this.HTable.length; i++) {

            this.HTable[i] = new Lista();

        }

    }
```

Mas antes de ser inserida precisei implementar uma função hash para pegar o primeiro caractere da palavra

```
    private int função Hash(String palavra) {

        return palavra.toLowerCase().charAt(0) % 26;

    }
```

Para adicionar as palavras no Hashing primeiro é preciso calcular a posição, para de acordo com a posição, inserir na tabela

```
public void add(TAD dados) {

        int posicao = funcaoHash(dados.getPalavra());

        HTable[posicao].add(dados);

    }
```

Para fazer a busca das palavras primeiro verifica em que posição ela está (letra)

```
    public TAD busca(String palavra) {

        int posicao = funcaoHash(palavra);

        return this.HTable[posicao].buscaListaH(palavra);

    }
```

5.2.4. Árvores

Dentre os 3 tipos de árvores estudados optei por utilizar a AVL. Não sendo necessário nenhuma remoção a árvore se limitou apenas a inserir os elementos.

```
public void inserir(TAD dados) {

        NoAVL n = new NoAVL(dados);

        inserirAVL(this.raiz, n);

    }

private void inserirAVL(NoAVL comparar, NoAVL inserir) {

        if (comparar == null) {

            this.raiz = inserir;
```

```
        } else {

                                                                    if
(inserir.getValor().getPalavra().compareToIgnoreCase(comparar.getValor(
).getPalavra()) < 0) {

                if (comparar.getEsquerdo() == null) {

                    comparar.setEsquerdo(inserir);

                    inserir.setPai(comparar);

                    verificarBalanceamento(comparar);

                } else {

                    inserirAVL(comparar.getEsquerdo(), inserir);

                }



                                                        }     else     if
(inserir.getValor().getPalavra().compareToIgnoreCase(comparar.getValor(
).getPalavra()) > 0) {

                if (comparar.getDireito() == null) {

                    comparar.setDireito(inserir);

                    inserir.setPai(comparar);

                    verificarBalanceamento(comparar);

                } else {

                    inserirAVL(comparar.getDireito(), inserir);

                }

            }

        }

    }
```

Deve-se sempre tomar cuidados com o balanceamento da árvore,a cada inserção o fator de balanço deve ser atualizado a partir do pai do nó inserido até a raiz da árvore.

Todos os nós da AVL devem ter fb = -1, 0 ou 1 (fator de balanceamento)

```
// verificar que tipo de rotamento sera necessario

    private void verificarBalanceamento(NoAVL atual) {

        setBalanceamento(atual);

        int balanceamento = atual.getBalanceamento();



        if (balanceamento == -2) {

                        if  (altura(atual.getEsquerdo().getEsquerdo())  >=
altura(atual.getEsquerdo().getDireito())) {

                atual = rotacaoDireita(atual);



            } else {

                atual = duplaRotacaoEsquerdaDireita(atual);

            }

        } else if (balanceamento == 2) {

                        if  (altura(atual.getDireito().getDireito())  >=
altura(atual.getDireito().getEsquerdo())) {

                atual = rotacaoEsquerda(atual);



            } else {

                atual = duplaRotacaoDireitaEsquerda(atual);
```

```
            }

        }

        if (atual.getPai() != null) {

            verificarBalanceamento(atual.getPai());

        } else {

            this.raiz = atual;

        }

    }
```

Na inserção basta encontrar o primeiro nó desregulado (fb= -2 ou fb= 2) e aplicar o operação de rotação necessária.

5.1.4. Lista

A lista é usada tanto na parte na pesquisa por lista quanto na pesquisa por hashing

```
// inserir na lista

    public void add(TAD elemento){

        // se a lista estiver fazia

        if(this.total_elementos == 0){

            // insira no inicio

            this.addInicio(elemento);

        }else{

            // crie um novo no e ensira na frente do ultimo

            SllNode nova = new SllNode(elemento);

            this.ultimo.setProximo(nova);

            this.ultimo = nova;
```

```
            this.total_elementos++;
        }
    }

    // adicionar no inicio de uma lista vazia
    public void addInicio(TAD elemento){
        SllNode nova = new SllNode(this.primeiro, elemento);
        this.primeiro = nova;

        if(this.total_elementos == 0){
            this.ultimo = nova;
        }
        this.total_elementos ++;
    }

    public TAD busca(String palavra) {
        return buscaLista(palavra);
    }

    // metodo de busca na lista
    public TAD buscaLista(String chave) {
            // comeca a procurar do primeiro
        SllNode atual = this.primeiro;
        while(atual!=null){
            Processamento.passosLista();
```

```
if(atual.getElemento().getPalavra().compareToIgnoreCase(chave)==0){
                return atual.getElemento();
            }
            atual = atual.getProximo();
        }
        return null;
    }



    // busca linear na lista para o Hashing
    public TAD buscaListaH(String key){
        // comeca a procurar do primeiro
        SllNode atual = this.primeiro;
        while(atual!=null){
            Processamento.passosHash();

if(atual.getElemento().getPalavra().compareToIgnoreCase(key)==0){
                return atual.getElemento();
            }
            atual = atual.getProximo();
        }
        return null;
    }
```

5.1.5. TAD

A class TAD é usada para guardar as palavras pesquisadas (Listas) e em quais linhas a mesma palavra se encontra.

```
// Guarda a palavra lida do arquivo e a linha de sua ocorrencia
public class TAD {
    private String palavra;
                private    ArrayList<Integer>    posicao_linha    =    new
ArrayList<Integer>();

    public TAD(){}
    public TAD(String palavra, int posicao_linha) {
        this.palavra = palavra;
        this.posicao_linha.add(posicao_linha);
    }

    public String linhaString(){
        String linha = "linha(s): ";
        Iterator<Integer> esta = posicao_linha.iterator();
        while(esta.hasNext()){
            linha = linha + esta.next().toString();
            linha = linha + "-";
        }
        return linha.substring(0,linha.length()-1);
    }
```

5.2 Extração de Dados: Python

5.2.1 Arquivos

A pasta Arquivos contém a base de entrada para o funcionamento do programa (Entrada) e o armazenamento das listas, árvores e hashings (Saida). Os arquivos são salvos no formato CSV para facilitar o processo de leitura.

5.2.2 extrair-tweets

A implementação do algoritmo de extração foi feita em Python, com base na documentação da API do Twitter Developer e de pesquisas

```
# -*- coding: utf-8 -*-
import sys
import csv
from time import time
import tweepy #https://github.com/tweepy/tweepy

# Autenticações
consumer_key = '3EnfmJo5rjIWQUycXHiKKX19s'
consumer_secret = 'FN1X0tNMXigHptrOr84WaAJXrTRk0dwIdwbb3heiym4HeaDwM4'
access_token = '1088915040483319808-1TcKZ9uPSAioC1cJqD2Rw9Bq2OKcKP'
access_token_secret = 'Yvhao8hwjuCJBqxBfMQPXY9ho2LQpDHsmOlyBA4iUbpwG'

# Autorização e inicialização da tweepy
auth = tweepy.OAuthHandler(consumer_key, consumer_secret)
auth.set_access_token(access_token, access_token_secret)
api = tweepy.API(auth, wait_on_rate_limit_notify=True,
wait_on_rate_limit=True) # Twitter's em tempo real

busca_palavra = "Futebol, Flamengo, Mengão" # Palavra para procurar
nova_busca = busca_palavra + " -filter:retweets"
resultados = tweepy.Cursor(api.search, q=nova_busca, lang="pt",
tweet_mode='extended').items(1000)

# Transforma os tweets em uma lista que vai preencher o CSV
saida_tweets = [['@'+tweet.user.screen_name, tweet.full_text]
```

```
                for tweet in resultados]

# Escreve o CSV
with open('edii_tweets.csv', 'w', encoding='utf-8') as arquivo_saida:
    escritor = csv.writer(arquivo_saida, delimiter=":",
lineterminator=':::\n') #o tweet termina com espacamento
    escritor.writerow(["autor","texto"])
    escritor.writerows(saida_tweets)
```

6. ANÁLISE DE RESULTADOS

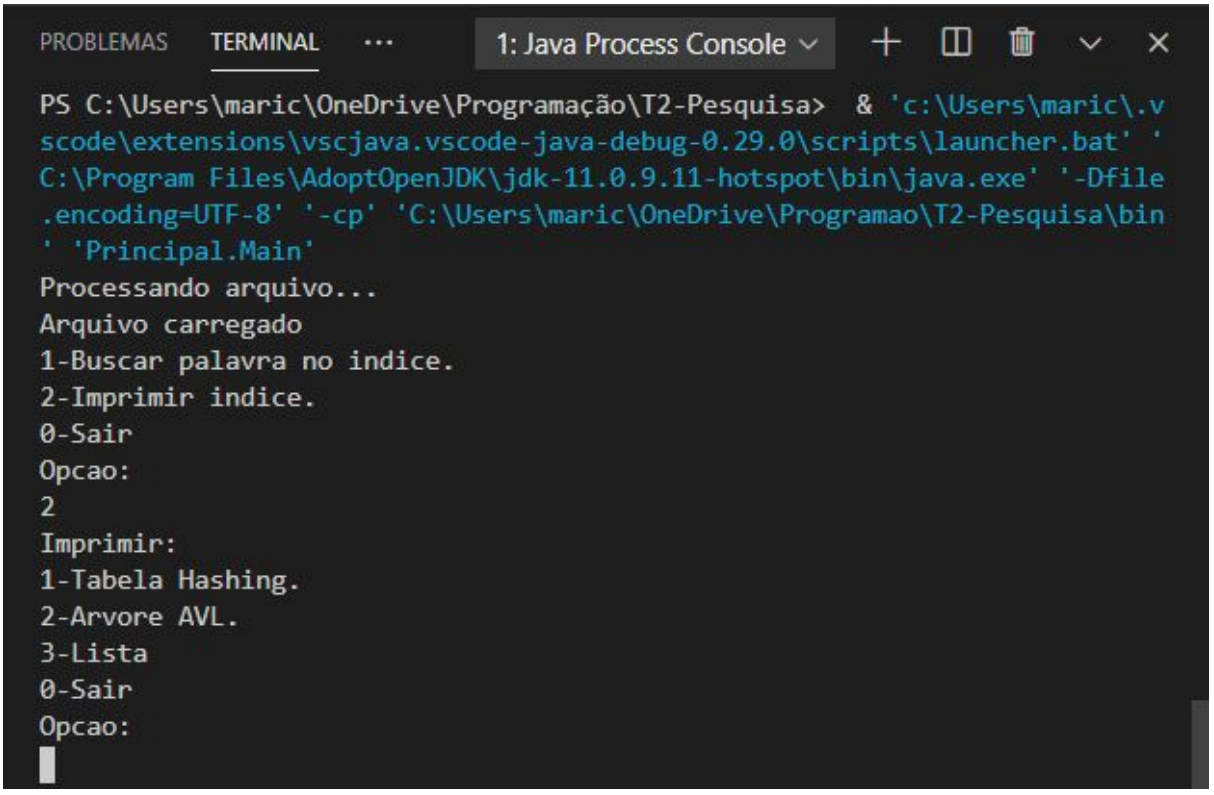
Os teste foram elaborados para analisar a quantidade de acessos para encontrar uma determinada chave (palavra). Os algoritmos de pesquisa: Árvore AVL, Hashing, Lista foram testado e comparados, para verificar qual se comporta melhor (rapidez).

Foram testados arquivos CSV, com tweets extraídos com tweepy, da API do Twitter. Após pedir autorização para usar a ferramenta tive de esperar de 3 a 5 dias para conseguir a base, porém devido algumas restrições da própria API, não consegui extrair os 10000 pedidos. Para os teste foram utilizados 10, 100, 1000 e um pouco mais de 1000 e menos de 10000 tweets, mas foi o suficiente para a análise.

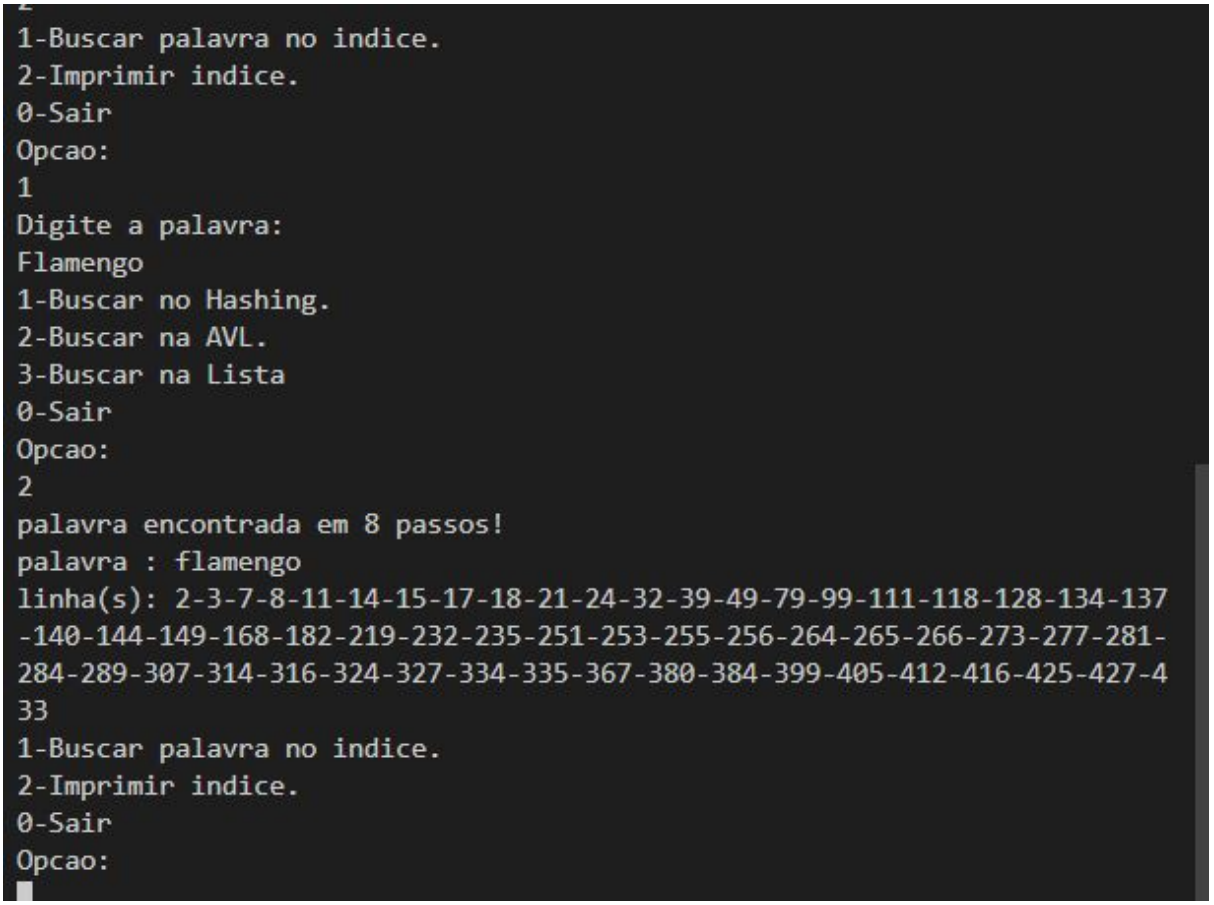
O programa foi implementado e testado em um Intel(R) Core(™) i5-8265U CPU @ 1.60GHz 1.80 GHz, segue abaixo as etapas para o funcionamento correto do mesmo:

Primeiro é verificado se o arquivo de entrada é existente.

Se o arquivo de saída já ter sido gerado a pesquisa pode ser feita, caso contrário será necessário carregar o arquivo



Um busca na árvore AVL seria feita assim:



A linha se refere a linhas no arquivo original

Para simplificar o exemplo foi utilizado uma base de 1000 tweets

O tempo de inserção na árvore AVL se deu de forma muito rápida, no hash um pouco mais, já a Lista demorou tanto para inserir, quanto para pesquisar.

7. CONCLUSÃO

Após inúmeros teste pude concluir, empiricamente, que tanto o hash quanto a árvore se mostrar eficientes, embora eu tenha implementado um hash com encadeamento externo para evitar colisões, a inserção na tabela requer uma busca e inserção dentro da lista encadeada, o que pode resultar em um alto custo de execução.

Como se trata de um grande quantidade de dados, não recomendo o uso de lista para pesquisa, por causa do tempo que leva para buscar uma palavra, embora ela tenha sido bastante eficiente no hashing.

8. REFERÊNCIAS BIBLIOGRÁFICAS

Livros:

- Nivio Ziviani. *Projeto de Algoritmos com implementações em Java e C++.* Thomson Learning, 2007
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Algoritmos: Teoria e Prática.* 3a edição. Elsevier, 2012

Sites:

- https://www.geeksforgeeks.org/avl-tree-set-1-insertion/
- Lista Simplesmente Encadeada [C/C++] (vivaolinux.com.br)
- https://www.ime.usp.br/~pf/estruturas-de-dados/aulas/st-hash.html
- https://developer.twitter.com/en
- Tweepy Documentation — tweepy 3.9.0 documentation
- https://www.geeksforgeeks.org/hashing-data-structure/