SERVIÇO PÚBLICO FEDERAL · MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE VIÇOSA · UFV
CAMPUS FLORESTAL

Trabalho Prático 1 - AEDS 2

Aplicação com Árvores digitais e tabelas Hash

João Victor Graciano Belfort de Andrade
Mateus Henrique Vieira Figueiredo
Vitor Ribeiro Lacerda

Florestal - MG

Sumário

1. Introdução

Esta documentação se refere ao projeto feito no Trabalho Prático I, da disciplina ALGORITMOS E ESTRUTURA DE DADOS II, de código CCF 212. No trabalho, foi sugerido a implementação de um código para a construção de índice invertido para máquinas de busca, a análise do tempo de execução, a relevância dos termos para a consulta, etc. a fim de compreender melhor como funciona a busca de palavras-chave. O projeto realizado, descrito neste documento, está disponível na página do GitHub: Basic-Search-Engine.

Para a realização do projeto, foram utilizadas referências e conteúdos de sites externos, de slides e conteúdos disponibilizados, tais como as explicações do professor Nivio Ziviani, contidas no livro Projeto de Algoritmos com Implementações em Pascal e C.

Algumas bibliotecas externas foram utilizadas para o programa, elas são “time.h”, “stdbool.h”, ”stddef.h”, “limits.h”, “math.h”, “ctype.h” e “string.h”. “ctype.h” foi utilizada para realizar operações com caracteres, “time.h” foi utilizada para pegar o horário da máquina e calcular o tempo de execução do código e gerar uma seed para garantir a aleatoriedade, a biblioteca “stdbool.h” foi utilizada para inserção do tipo bool e para lógica booleana, ”stddef.h” foi utilizada para a inserção do macro “NULL”, a biblioteca “math.h” foi utilizada para ter acesso a funções matemáticas, tais como “sqrt”, “pow”, entre outras, que foram utilizadas em partes distintas dos códigos. “string.h” foi utilizada para o tratamento de strings e “limits.h” foi utilizada para a definição de “CHAR MAX".

O programa foi dividido em 2 pastas, sendo uma para armazenamento dos arquivos de teste utilizados para a montagem dos índices e outra, contendo 6 subpastas, para o algoritmo solicitado no trabalho. Estas subpastas funcionam como subdivisões voltadas para a maior organização das TAD 's.

2. Metodologia

Para elaboração do projeto, o grupo tomou a decisão de dividir tarefas e adotar métodos voltados para a organização. Assim como anteriormente citado, para o versionamento do código, foi tomada a decisão de utilizar o GitHub, o que garantiu também uma maior organização. Outra estratégia fundamental foi a utilização da ferramenta “Code

with me", disponibilizada pela IDE Clion, o que facilitou o contato mais imediato dos integrantes com a lógica e a implementação dos demais, garantindo um melhor e mais rápido compartilhamento de ideias.

Para os testes, o grupo optou por utilizar os livros da saga Harry Potter, assim como os testes disponibilizados. Ao submeter o código a testes tão robustos, os conhecimentos sobre utilização de memória tiveram de ser expandidos, assim como os conhecimentos sobre complexidade de algoritmos, o que acabou por se tornar uma boa oportunidade para os integrantes. Tais testes também foram muito importantes para uma comparação mais clara entre as estruturas Patricia e Hash.

Ao finalizar o trabalho, o grupo pode chegar a conclusão de que a divisão de tarefas acabou por ser uma importante estratégia. Esta tomada de decisão fez com que o trabalho pudesse ser executado em um tempo menor, além do mais, cada aluno pode contribuir em sua área de maior conhecimento, assim como desenvolver ainda mais suas habilidades, realizando pesquisas sobre o assunto.

3. Organização

Na pasta principal do arquivo podem ser encontrados os arquivos, cada um destes com os seguintes nomes: files (Pasta de arquivos), src (Pasta de arquivos), .gitignore.txt. Os arquivos mencionados do tipo "Pasta de arquivos" ainda contém outros, em **"files"**: "arquivo1.txt", "arquivo2.txt", "arquivo3.txt", "arquivo4".txt, "arquivo5.txt", "arquivo6.txt", "arquivo7".txt, "arquivo8.txt", "arquivo9.txt" e "entrada.txt", é importante relembrar que caso outros arquivos sejam criados, os arquivos **devem estar na pasta "files"**. Para garantir uma maior portabilidade do código, a pasta **"build"** contém o código compilado, juntamente com o arquivo Cmake. Já em **"src"** temos: "main.c", e outras 6 subpastas, **"file"**, **"hashtable"**, **"linkedlist"**, **"pairlinkedlist"**, **"patricia"** e **"tfidf"**. Ainda pode-se encontrar arquivos dentro das subpastas, em **"file"** temos: "file.c" e "file.h". Em **"hashtable"**, temos: "hashtable.c" e "hashtable.h". Em **"linkedlist"** temos: "linkedlist.c", "linkedlist.h", "node.c" e "node.h". Em **"pairlinkedlist"** temos: "pairlinkedlist.c", "pairlinkedlis.h", "pairnode.c" e "pairnode.h". Em **"patricia"** temos: "patricia.c", "patricia.h", "treenode.c" e "treenode.h". Em **"tfidf"** temos: "tfidf.c" e "tfidf.h".

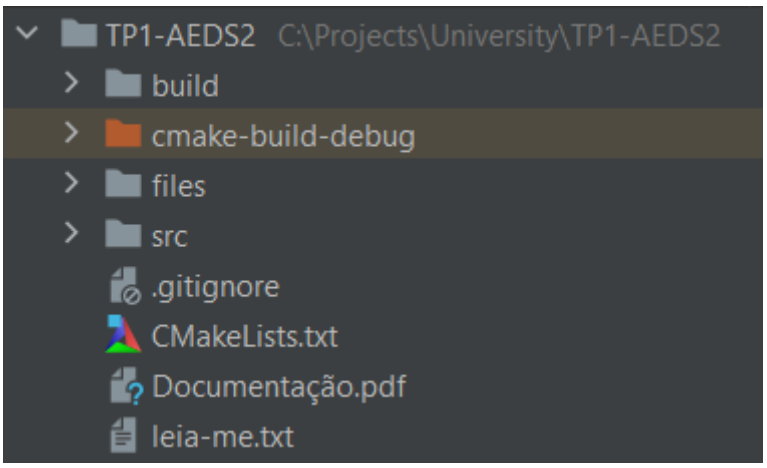

Figura 1- Repositório do projeto

Sendo assim, a pasta principal do projeto contém arquivos referentes ao repositório do github, e as pastas contendo os códigos e os arquivos, além de arquivos referentes ao projeto e arquivos criados pela ide CLion na compilação do código. Em "files" têm-se os arquivos criados pelo grupo, variando de "arquivo1.txt" até "arquivo.txt", os arquivo "arquivo.txt", do 3 até o 9 contém a saga inteira de Harry Potter, já os arquivos "arquivo1.txt" e "arquivo2.txt" contém os testes disponibilizados. O arquivo "entrada.txt", contém em sua primeira linha, o número de arquivos que serão lidos, e nas outras linhas, o nome dos arquivos a serem lidos pelo programa. A pasta "src" contém as TAD's necessárias para a implementação do trabalho, separadas em sub pastas, e o arquivo "main.c".

4. Desenvolvimento

Como supracitado, as divisões foram feitas para uma maior organização. Podemos encontrar TAD's voltadas para diferentes partes do código, como por exemplo na pasta **"file"**, onde temos:

4.1 "file.c" e "file.h"

Nestes arquivos foi descrita a "TAD file", essa estrutura contém funções utilizadas no código, funções estas que são voltadas para a leitura dos arquivos. Foram criadas as seguintes funções:

“reformatString”: Função utilizada para formatar strings, removendo pontuações e espaços e convertendo toda a string para “lowecase”. Estas strings formatadas, serão utilizadas posteriormente em partes distintas do código.

“readFilenamesHashtable”: Função que utiliza os dados fornecidos para ler o nome dos arquivos que contêm os textos a serem inseridos na Hashtable.

“readFilenamesPatricia”: Função que utiliza os dados fornecidos para ler o nome dos arquivos que contêm os textos a serem inseridos na Patricia.

“readFileIntoHashtable”: Função utilizada para ler o texto a ser inserido do arquivo para a tabela Hash. Esta função é fundamental para o código, visto que se apresentar algum erro, pode fazer com que a leitura e as inserções sejam feitas de forma equivocada.

“readFileIntoPatricia”: Função utilizada para ler o texto a ser inserido do arquivo para a árvore Patricia. Esta função, assim como a função “readFileIntoHashtable”, é fundamental para o código, uma vez que se apresentar algum erro, pode fazer com que a leitura e as inserções sejam feitas de forma equivocada.

“freeFilenames”: Função que "desloca" da memória os arrays de string que contém os nomes dos arquivos. Esta função possui uma grande importância para certificar que algum nome errado não seja utilizado, assim como para liberar espaço na memória.

Já em **“tfidf”**, temos:

4.2 “tfidf.c” e “tfidf.h”

Nestes arquivos foi descrita a “TAD tfidf” , estrutura que é voltada para o cálculo da frequência dos termos da consulta em cada documento da coleção bem como na frequência inversa dos documentos.

Em **“tfidf.h”**, podemos encontrar as structs:

“TFIDF” e “Relevance”: Ambas structs são voltadas para o cálculo da frequência dos termos da consulta em cada documento, guardando tudo o que for necessário para a realização.

Há também nestes arquivos, funções podemos encontrar, são elas:

“initialiseTFIDF”: Função que inicializa a estrutura “TFIDF” com os valores fornecidos.

“initialiseRelevance”: Essa função é usada para inicializar a estrutura “Relevance” com os valores fornecidos.

“freeTFIDF”: Função usada para desalocar a estrutura “TFIDF” da memória. Esta função possui uma grande importância pois libera espaço na memória

“freeRelevance”: Função usada para desalocar a estrutura “Relevance” da memória. Esta função possui uma grande importância para liberar espaço na memória

Para o desenvolvimento e implementação das TAD’s da árvore Patricia e da tabela Hash, o grupo desenvolveu outras duas estruturas, linkedlist e pairlinkedlist. Em **“pairlinkedlist”**, encontra-se funções e structs que serão utilizadas em partes do código. Dentro dessa subpasta podemos encontrar:

4.3 “pairnode.c” e “pairnode.h”

Nestes arquivos foi descrita a “TAD pairnode” , estrutura que é voltada para o estabelecimento dos “pairnodes” que serão utilizados no código.

Em **“pairnode.h”**, podemos encontrar a struct:

“PairNode” : Esta estrutura armazena os dados necessários para a utilização, tais como o número de ocorrência dos termos, o ID do documento e o ponteiro para o próximo node.

Podemos, nestes arquivos, encontrar a função:

“initialisePairNode”: Função que inicializa a estrutura “PairNode” com os valores fornecidos. A função retorna o ponteiro para o “externalNode” inicializado.

Ainda na subpasta, podemos encontrar outros arquivos, tais como:

4.4 “pairlinkedlist.c” e “pairlinkedlist.h”

Nestes arquivos foi descrita a “TAD pairlinkedlist” , estrutura que implementa a estrutura de uma lista duplamente encadeada. Podemos encontrar nestes arquivos a estrutura da lista, contendo as informações necessárias, a estrutura tem o nome de “PairLinkedList”.

Podemos também encontrar as funções:

“initialisePairLinkedList” : Função utilizada para inicializar a lista duplamente encadeada com os valores anteriormente definidos.

“searchPairNode” : Esta função é utilizada para buscar o “pairNode” na lista duplamente encadeada, retorna o “pairNode” e caso a lista esteja vazia, retorna NULL.

“pushPair” : Assim como já “definido” em seu nome, a função introduz um novo par (PairNode) na lista. Por ser do tipo bool, a função retorna se a operação foi concluída com sucesso ou não.

“getSizeOfPairLinkedList” : Esta função é utilizada para pegar qual o “sizeof”da lista duplamente encadeada, retornando o seu tamanho.

“isPairLinkedListEmpty” : Função utilizada para checar se a lista está vazia, a função retorna se a lista está ou não vazia.

“getTFIDFPairLinkedList” : Função que calcula o TF-IDF e coloca o resultado na estrutura “TFIDF”. Esta função tem grande importância no cálculo e depende de partes supracitadas do código.

“printPairLinkedList” : Como parâmetro, a função necessita do ponteiro para a lista duplamente encadeada, após receber tal parâmetro, a função imprime cada “externalNode” da lista.

“freePairLinkedList” : A fim de liberar espaço na memória, a função “desaloca” a lista duplamente encadeada (alocada dinamicamente) da memória.

Na subpasta **“linkedlist”**, encontram-se outras structs e funções a serem utilizadas no código. Dentro dessa subpasta podemos encontrar:

4.5 “node.c” e “node.h”

No header, podemos encontrar uma struct do node utilizado na linked list. Este node armazena informações necessárias, tais como o próximo node, um ponteiro para a palavra, entre outras. Podemos encontrar nestes arquivos as funções:

“initialiseNode”: Função que inicializa a estrutura “Node” com os valores previamente estabelecidos, a função retorna um ponteiro para o “externalNode” inicializado.

“initialiseNodeWithExistentData”: A função inicializa um novo node com os conteúdos existentes do “externalNode”. Como dito, a função retorna um novo ponteiro.

“compareAlphabetically”: Esta função é utilizada para comprar duas palavras, caractere por caractere.

“hashCode”: Função utilizada para retornar o hashcode de uma palavra gerada com base nos pesos obtidos no array de pesos.

“searchNode”: Função utilizada para buscar por um “externalNode” na estrutura linkedlist. A função retorna se o node foi encontrado ou não.

Ainda na subpasta “linkedlist”, podemos encontrar os arquivos:

4.6 “linkedlist.c” e “linkedlist.h”

Em “linkedlist.h” pode ser encontrada a estrutura “LinkedList”, que armazena valores e ponteiros importantes para o funcionamento da estrutura. Nos arquivos podem ser encontradas funções, tais como:

“initialiseLinkedList”: Esta função é utilizada para inicializar a estrutura da linkedlist com os valores anteriormente definidos. A função retorna um ponteiro para a estrutura dinamicamente alocada.

“push”: Função que introduz uma nova palavra na estrutura. A função retorna se a operação obteve sucesso ou não.

“pushSorted”: Esta função é utilizada para inserir a palavra na estrutura, de forma ordenada, é importante salientar que este método é internamente utilizado para pegar informações ordenadas.

“getSizeOfLinkedList”: Função utilizada para retornar o “sizeof” da linkedlist.

“isLinkedListEmpty”: Esta função recebe como parâmetro um ponteiro para a estrutura e retorna se a lista está ou não vazia.

“getTFIDFLinkedList”: Esta função utiliza de diferentes partes do código para calcular o TF-IDF, a função insere o resultado obtido na estrutura TFIDF.

“printLinkedList”: Função que recebe como parâmetro o ponteiro para a estrutura e que é utilizada para printar a linkedlist.

“freeLinkedList”: Visando uma melhor utilização e otimização o uso da memória, esta função foi criada para desalocar a linkedlist (dinamicamente alocada) da memória.

“freeSortedLinkedList”: Esta função desaloca a estrutura que foi construída para as informações ordenadas, a função apenas desaloca os nodes da estrutura “Node”. Deve-se notar que o método é usado internamente apenas para a linkedlist que foi alocada pelo método “sortAndPrintHashtable”.

Partindo para as estruturas principais do trabalho prático, tais estruturas podem ser encontradas nas subpastas **“hashtable”** e **"patricia"**. Na implementação destas estruturas foram tomadas decisões chaves, como por exemplo na implementação da **Tabela Hash**, em que o grupo optou por utilizar alguns conceitos contidos nos slides disponibilizados, tais como o método de Sedgewick e o cálculo de pesos. Conceitos mais aprofundados serão explicados nas próximas seções.

Na subpasta **“hashtable”**, podem ser encontrados diferentes arquivos, são estes:

4.7 “hashtable.c” e “hashtable.h”

Nos arquivos supracitados podem ser encontradas uma estrutura “Hashtable”, que contém os dados e informações necessárias para a implementação de tal estrutura, tais como o ponteiro para o array de pesos, o tamanho máximo da tabela hash, entre outras. O código leva também em consideração o tamanho da maior palavra da língua inglesa, que contém 45 caracteres. Podem ser encontradas também nos arquivos as funções:

“initialiseHashtable”: Função utilizada para inicializar a tabela hash. Devem ser passados como parâmetro o ponteiro para a estrutura Hashtable e um tamanho para a tabela.

“initialiseWeightsArray”: Utilizada para inicializar o array de pesos, utilizado na geração dos hash codes para as palavras. Retorna um ponteiro para o array.

“hash”: Função utilizada para pegar um endereço (index) dentro do array da tabela hash baseado em um código hash. A função retorna um index para o array da tabela hash.

“getTwoPowerValueGreaterOrEqual”: Esta função é utilizada para achar as duas potências maiores ou iguais ao número estabelecido, ou seja, acha o valor “M” de acordo com o método Sedgewick.

“getPreviousPrime”: Esta função é utilizada para encontrar o número primo mais próximo ao número previamente estabelecido. Esta função também acha o valor “M” de acordo com o método Sedgewick.

“checkForPrimality”: Função que retorna “true” ou “false” para caso o número seja primo. Ou seja, a função é utilizada para checar se o número é ou não primo.

“insertionSort”: Função que performa um “insertion sort” no array.

“insertIntoHashtable”: Função utilizada para inserir uma nova palavra na tabela Hash, a função retorna se a operação foi concluída com sucesso.

“getSizeOfHashtable”: Esta função recebe como parâmetro o ponteiro para a struct da tabela Hash e retorna o “sizeof” da tabela.

“isHashtableEmpty”: Checa se a estrutura da tabela hash está vazia. A função retorna se a tabela se encontra vazia ou não.

“getTFIDFHashtable”: Esta função utiliza de diferentes partes do código para calcular o TF-IDF, a função insere o resultado obtido na estrutura TFIDF.

“sortAndPrintHashtable”: Função que ordena e “printa” a hash table. A função recebe como parâmetro o ponteiro para a estrutura da tabela. Para “printar” a HashTable em ordem alfabética, foi usado uma lista de ponteiros, de forma a inserir as listas encadeadas da tabela de forma ordenada, assim, ao realizar o output destas informações, será garantida a ordem alfabética.

“freeHashtable”: Assim como outras funções de “free”, o grupo optou por uma maior otimização do uso de memória e definiu funções que libera as estruturas. A função “desaloca” a estrutura (que foi dinamicamente alocada) da memória.

“calculateWeightHashtable”: Esta função é utilizada para calcular o TF-IDF de uma palavra previamente estabelecida.

“relevanceHashtable”: Função utilizada para calcular a relevância de um termo baseada no cálculo do TF-IDF. Esta função está diretamente relacionada ao resultado exigido no trabalho.

Na subpasta **“patricia”**, podemos encontrar 4 arquivos, sendo eles: “treenode.c” e “treenode.h”, que juntamente, definem a “TAD treenode”, que será utilizada na “TAD patricia”, constituída pelos arquivos “patricia.c” e “patricia.h”.

Existem alguns apontamentos importantes a serem considerados a cerca de escolhas e adaptações ao código fornecido e ensinado nas aulas, como por exemplo o caminho dos valores. Os valores maiores ou iguais ao valor do node vão para o node à direita na árvore, assim garantindo que os valores (principalmente os prefixos) estejam em ordem. Outra adaptação é que os prefixos da árvore, ao serem inseridas novas palavras, seguem para a esquerda e a palavra (que contém o prefixo) é inserida no nó à direita, seguindo o conceito dos maiores valores para a direita.

Estas foram as principais diferenças na implementação do código, e estas foram fundamentais para que o código funcionasse adequadamente, como o esperado. Como dito anteriormente, nesta subpasta podemos encontrar diferentes TADS, como:

4.8 “treenode.c” e “treenode.h”

Nestes arquivos, podem ser encontradas diversas “structs” que são necessárias para a construção da árvore, tais como as definições dos tipos de node, apontadores, entre outros. Todas estas structs têm uma grande importância na criação da árvore, visto que definem o funcionamento dos nodes. Assim como as structs, as funções contidas nos arquivos também tem grande importância, são elas:

“createInternalNode”: Função utilizada para inicialização de um “externalNode” interno com os valores estabelecidos (node aponta para NULL, ou seja). A função retorna o ponteiro para o no inicializado.

“createExternalNode”: Função utilizada para inicialização de um “externalNode” externo com os valores previamente estabelecidos. A função retorna o ponteiro para o no inicializado.

“insertBetween”: Esta função é utilizada para encontrar um lugar para a palavra a ser inserida. Cria um node interno para inserir a palavra.

“insertTreeNode”: Esta função deve ser utilizada para inserir palavras na árvore. A função retorna sucesso ou falha ao tentar inserir a palavra e leva em consideração diversos casos que podem ocorrer ao tentar inserir a palavra.

“getDifferChar”: Função criada com o intuito de achar a letra que difere entre duas palavras. Para essa função, existem alguns casos especiais, tais como:

- Prefixo: Retorna a primeira letra da maior palavra que está sendo comparada.
- Igual: Retorna a última letra da palavra.

A função retorna o caractere que se diferencia ou os casos especiais supracitados.

“isWordGreaterOrEqualThanChar”: Checa se o index da palavra especificada e retorna se o char da posição especificada pelo index é maior ou igual ao caractere.

“isExternalNode”: Esta função serve para checar se o node especificado é um node do tipo externo. A função recebe o ponteiro para a estrutura do node e por ser do tipo “bool” retorna o resultado se é externo ou não.

“printTreeNode”: Função utilizada para printar o node, um após o outro. Esta função recebe como parâmetro o ponteiro para a estrutura do node.

“getTFIDFTreeNodes”: Esta função utiliza de diferentes partes do código para calcular o TF-IDF e insere o resultado obtido na estrutura TFIDF.

“freeTreeNodes”: Função utilizada para “desalocar” o node da árvore da memória. A função recebe o ponteiro para a raiz como parâmetro e percorre a árvore desalocando os nodes.

“getSizeTreeNodes”: Função utilizada para descobrir o tamanho de um “TreeNode”. Esta função recebe como parâmetro um ponteiro para a estrutura “TreeNodeType” e um ponteiro para o tamanho dos nodes em bites.

Ainda na sub pasta, encontram-se os arquivos:

4.8 “patricia.c” e “patricia.h”

Nestes arquivos está definida a TAD Patricia, em "patrícia.h" pode ser encontrada a estrutura “PATRICIA”, tal como diversas funções, como:

“initialisePATRICIA”: Função utilizada para inicializar a estrutura PATRICIA utilizando os valores previamente definidos.

“insertIntoTree”: Função definida e utilizada com o intuito de inserir um node na árvore Patricia. A função retorna se a operação obteve ou não sucesso.

“getTFIDFPATRICIA”: Esta função recebe como parâmetro um ponteiro para a struct PATRICIA e um ponteiro para a struct TFIDF. A função utiliza diferentes partes do código para calcular o TF-IDF e insere o resultado obtido na estrutura TFIDF.

“calculateWeightPATRICIA”: Calcula o peso do TD-IDF de determinada palavra. A função utiliza diferentes partes do código para calcular o TD-IDF e insere o resultado obtido na estrutura TDIDF.

“relevancePATRICIA”: Função utilizada para calcular a relevância de um termo, baseado no cálculo TF-IDF, obtido em outras etapas do código.

“getSizeOfPATRICIA”: Função que recebe como parâmetro um ponteiro para a estrutura PATRICIA e retorna o tamanho da árvore.

“freeTree”: Visando uma melhor otimização de memória na implementação, o grupo optou por "deslocar" as structs. Para isto, essa função foi criada, com o intuito de "deslocar" as estruturas que haviam sido dinamicamente alocadas.

4.9 “main.c”

Neste arquivo foi implementado o código principal do programa que utiliza dos demais executáveis e headers, para criar uma interface onde o usuário poderá escolher qual das estruturas para gerar os índices invertidos. Dentro deste arquivo, foram definidas funções, são elas:

“getUserOperationOption”: Função que recebe o input do usuário, inputs esses que serão utilizados nas operações. A função retorna se o valor inserido foi ou não válido.

“getNumberTerms”: Função utilizada para receber do usuário o número de termos a serem analisados. A função retorna se o valor inserido foi ou não válido.

“calculateTotalTime”: Função utilizada para calcular o tempo decorrido, utilizando o tempo inicial (momento em que o código começou a ser executado). A função retorna o tempo decorrido.

“freeWords”: Função utilizada para desalocar o array de strings, que contém as palavras, da memória.

“cleanStdin”: Função utilizada para limpar o “stdin” a fim de evitar problemas com a função “scanf()”.

Para o código da implementação do menu, foi escolhido uma abordagem o mais interativa o possível com o usuário, tentando prever até mesmo os inputs feitos de forma incorreta e retornando mensagens de erro, outra interação importante é que código apresenta um tamanho default para a hash table, que pode ser alterado pelo usuário. Para conseguir observar os resultados, o usuário terá de colocar qual das estruturas gostaria de executar, tal como o nome do arquivo que deve ser lido pelo código. Quando escolhida a estrutura e caso o arquivo especificado esteja na formatação correta, o código irá executar e o usuário poderá escolher printar os índices no terminal e poderá analisar o tempo decorrido.

5. Resultados

Para gerar os resultados, os arquivos usados para os testes foram os arquivos “arquivo.txt” do 3º ao 9º, contendo os livros da saga Harry Potter, totalizando juntos 1 milhão, 133 mil e 124 palavras.

No gráfico de tempo abaixo, foi observado uma HashTable de tamanho 100 em comparação com uma árvore Patricia.

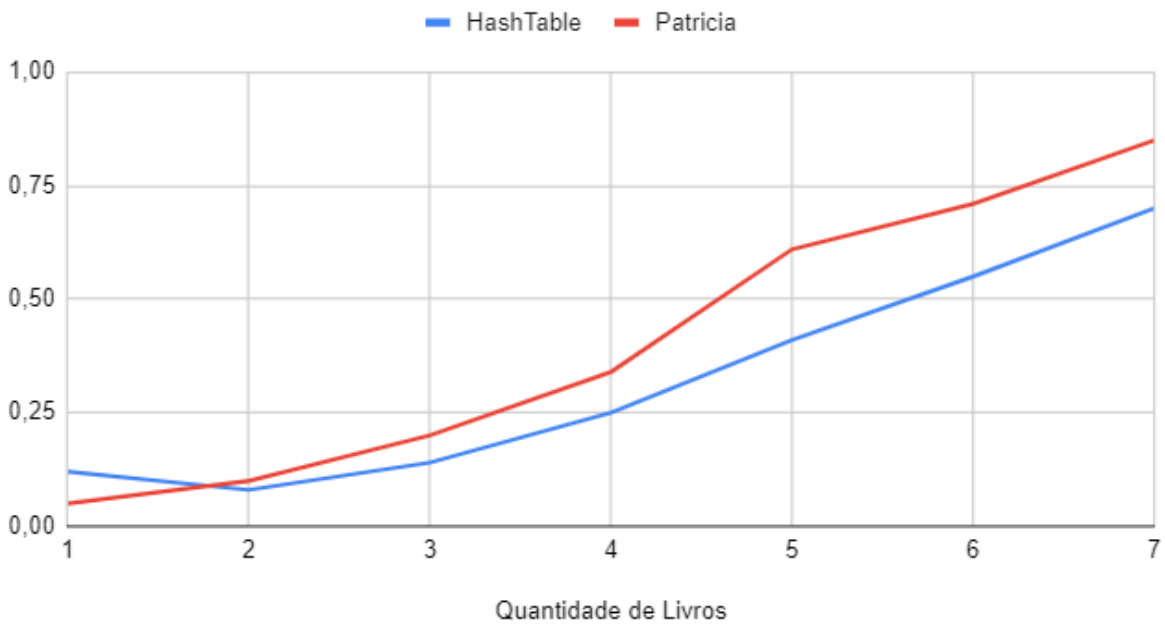

Figura 2 - Gráfico de tempo para as estruturas

O gráfico relaciona o tempo decorrido para inserção dos livros em cada uma das estruturas, levando em consideração a quantidade de livros utilizados. Podemos notar que a partir de dois livros, a Hashtable passa a apresentar melhores resultados, porém a estrutura Patricia obtém certa superioridade quando apenas um livro é utilizado. O que muda quando levamos em consideração o uso de memória, como podemos notar no gráfico a seguir.

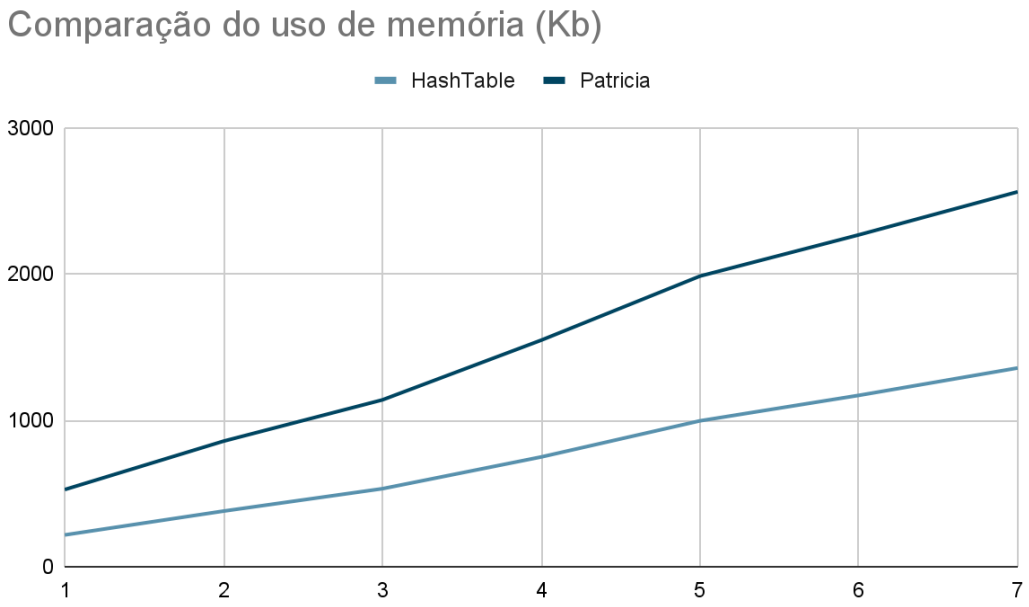

Figura 3 - Gráfico de uso de memória para as estruturas

Quando levamos em consideração os KiloBytes (Kb) necessários, pode ser observado que a HashTable representa uma melhor escolha caso a otimização do uso de memória seja a prioridade, pois desde a leitura do primeiro livro, a estrutura Patrícia demonstra um uso maior de memória.

Em conclusão, pode-se afirmar que a Hashtable, para a implementação em questão e para os parâmetros analisados, representa uma melhor escolha devido ao menor tempo decorrido para a inserção das palavras. É importante ressaltar que o tamanho da Hashtable pode alterar completamente os resultados apresentados nos gráficos.

6. Conclusão

De forma geral, foram implementados TADs e códigos, de forma organizada, comentada e de fácil entendimento para o problema solicitado pelo trabalho, que acabou por ser uma grande fonte de conhecimento sobre uso de memória e interação com o usuário, já

que foi decidida uma abordagem mais interativa com o mesmo (assim como pedido no trabalho), que ao fim foi uma ótima opção abstraindo o código, e o deixando mais fácil de ser visualizado e compreendido pelo usuário. Os resultados saíram assim como o esperado, sendo que algumas implementações feitas, apesar de deixarem o código mais extenso, facilitam seu uso e entendimento.

Em geral, a maior dificuldade encontrada pelo grupo, durante a realização do trabalho prático, foi o entendimento da teoria acerca das estruturas de dados, um problema que foi contornado, buscando por diversas vezes conhecimentos de fontes externas. Sendo assim o trabalho, de forma geral atende as exigências passadas pela professora do curso de AEDESII-2022/1.

7. Referências

Para versionar o projeto foi utilizado o Github [1].

Explicação de transversalização nas estruturas de árvore [2].

Código utilizado para se basear na montagem da árvore Patrícia [3].

Explicação de como separar caracteres alfabético dos demais tipos [4].

Códigos e explicação das Implementações da Hashtable e Árvore Patricia [5].

Slides disponibilizados via PVA Moodle [6]

Arquivos ".txt" dos livros da saga Harry Potter [7]

[1] Github. Disponível em: <https://github.com/Mateus-Henr/Basic-Search-Engine> Último acesso em: 26 de Junho de 2022.

[2] Geeks for Geeks, **Tree Traversals (Inorder, Preorder and Postorder)**, 15 de Junho de 2022. Disponivel em: <https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/> Último acesso em: 22 de Junho de 2022.

[3] DCC UFMG, **Código da árvore patrícia em C**, Disponivel em : <http://www2.dcc.ufmg.br/livros/algoritmos/cap5/codigo/c/5.16a5.21-patricia.c> Último acesso em: 21 de Junho de 2022.

[4] Programiz, **C isalpha()**, Disponível em : <https://www.programiz.com/c-programming/library-function/ctype.h/isalpha> Último acesso em: 21 de Junho de 2022.

[5] Nivio Ziviani, **Projeto de Algoritmos com Implementações em Pascal e C**, 23 junho 2010.

[6] Slides Profª Gláucia Braga e Silva, disponibilizados pela própria. Último acesso em 26 de Junho de 2022.

[7] Github, **formcept/whiteboard**, Disponível em: <https://github.com/formcept/whiteboard/tree/master/nbviewer/notebooks/data/harrypotter> Último acesso em: 26 de Junho de 2022.