Universidade Federal de Viçosa - Campus Florestal

Primeiro trabalho prático da disciplina de Algoritmos e Estrutura de Dados II

Implementações a fins de comparação das árvores TRIE TST e PATRICIA

Calebe de Paula Costa 3478

Isabella Menezes Ramos 3474

Lucas Barros 3511

Vinícius Júlio 3495

Florestal - MG

26 de Outubro de 2020

Calebe de Paula Costa 3478
Isabella Menezes Ramos 3474
Lucas Barros 3511
Vinícius Júlio 3495

Primeiro trabalho prático da disciplina de Algoritmos e Estrutura de Dados II

Primeiro trabalho prático da disciplina Algoritmos e Estrutura de Dados II. O trabalho tem o intuito de comparar o desempenho sobre duas árvores, TRIE TST e PATRICIA.

Professora: Gláucia Braga

Florestal - MG
26 de Outubro de 2020

Sumário

Introdução

O trabalho visa fazer a implementação de duas árvores digitais: a TST (Ternary Search Tree), um tipo de árvore TRIE e também a PATRICIA (Practical Algorithm to Retrieve Information Coded in Alphanumeric), uma árvore digital que é uma representação compacta da TRIE e também é conhecida como Radix Binária.

Utilizaremos essas duas árvores como dicionários, com o objetivo de guardar palavras dentro delas, para fins de comparação de desempenhos de tempo, memória e números de comparações realizados dentro de cada função de inserção. Para fazer essas comparações, foram criados casos de teste nos arquivos de entrada, pensados nos piores casos ou nos casos médios de cada árvore, para observarmos os desempenho de ambas e qual árvore obtém melhor resultado em cada teste.

Para cada árvore, temos a funcionalidade de inserção por arquivo, inserção pelo menu interativo, pesquisa de palavras na árvore pelo menu interativo, exibir palavras em ordem alfabética e também a contagem de quantidade de palavras presente na árvore, lembrando que não há a ocorrência de duas palavras iguais nas árvores

Objetivo

O objetivo do trabalho é aplicar os conhecimento adquiridos em aula, comparando desempenho das duas árvore dentre os casos de teste, observando em cada caso o seu desempenho em termos de tempo de execução, memória utilizada e número de comparações realizados na inserção, e assim, analisando qual árvore obtém melhores resultados dependendo do caso de teste.

Desenvolvimento

O trabalho consiste na implementação de dois tipos abstratos de dados: TAD TRIE TST e TAD PATRICIA. As implementações básicas da Trie TST foram baseadas na implementação disponível no site do Geeks For Geeks, enquanto a implementação da PATRICIA é uma adaptação do implementação do professor Nivio Ziviani, adaptada para a inserção de strings. O link e a citação do livro estão disponíveis nas referências bibliográficas. Cada estrutura compõe suas estruturas e implementações específicas. Iremos explicar todas as implementações feitas e a estrutura de cada uma a seguir.

Ternary Search Tree (TST)

Para implementar a TST, foi criado uma estrutura *TipoNo* que contém uma variável Letra, de tipo char, com o intuito de que cada nó da árvore armazena uma letra da palavra que vai ser inserida. Temos também a variável *ehOFimDaString* que armazena um valor que diz se a letra do nó atual é a última letra de uma palavra presente na árvore, caso o valor da variável seja 1 isso quer dizer que é sim o fim da palavra, se for 0, não é a última letra da palavra. Dentro da estrutura temos também três variáveis do *TipoApontador*, que são na verdade variáveis do *TipoNo* com ponteiros, e são elas: *Esquerda*, *Igual* e *Direita* que são apontadores que apontam para outras ramificações das árvores. A estrutura implementada se encontra na Figura 01.

```
typedef unsigned char TipoChave;

typedef struct TipoNo *TipoApontador;
typedef struct TipoNo{
    TipoChave Letra;
    unsigned ehOFimDaString: 1;
    TipoApontador Esquerda, Igual, Direita;
}TipoNo;
```

Figura 01. Estrutura do TAD TST

As funções implementadas no TAD TST se encontram na Figura 02. Iremos falar dessas funções detalhadamente mais pra frente, mas no geral, foi implementado funções necessárias para fazer uma implementação que se assemelhava a um dicionário. Temos então, uma função para iniciar a árvore, duas

funções para inserir uma palavra na árvore, duas funções para imprimir o que está inserido na árvore, uma função de pesquisar uma palavra, uma função para ler um arquivo e inserir na árvore as palavras presente nesse arquivo, também duas funções para contar a quantidade de palavras presentes nessa árvore e por último uma função para transformar as palavras em letras minúsculas, para que dessa forma haja um padrão, e que se por acaso uma mesma palavra já foi inserida com as letras minúsculas e depois inserir essa mesma palavra em letras maiúsculas, a implementação consegue identificar que a palavra já foi inserida.

```
void inicializaTST(TipoApontador *No);
void insereTST(TipoApontador *No, char *Letra, int *Comparacoes);
void insereTSTAux(TipoApontador *No, char *Letra, int *Comparacoes);
void imprimeTST(TipoApontador No);
void imprimeTSTAux(TipoApontador No, char* buffer, int depth);
int pesquisarTST(TipoApontador *No, char *Palavra, int *Comparacoes);
int abrirArquivo(TipoApontador *Arvore,char *nomeArq, int *Comparacoes);
void contarPalavrasTST(TipoApontador No);
int contarPalavrasTSTAux(TipoApontador No);
char* tranformarPalavra(char *palavra);
```

Figura 02. Funções implementadas no TAD TST

A função *inicializaTST* funciona de maneira simples, apenas fazemos o apontamento do No para nulo.

Já as funções *insereTST* e *insereTSTAux*, como já dito antes, servem para fazer a inserção de palavras na árvore. Primeiro entramos na função *insereTST*, que primeiramente faz a busca da palavra que queremos inserir na árvore, caso a palavra já esteja na árvore, é printado uma informação de que a palavra já foi inserida e não entramos na função auxiliar, que de fato faz a inserção. Caso a palavra não esteja na árvore, entramos na função auxiliar *insereTSTAux*.

A função funciona da seguinte forma: ela percorre a árvore comparando letra por letra, vendo se a letra que eu quero inserir da palavra é maior, menor ou igual a letra que está no nó em que eu me encontro. Se a letra da palavra que eu quero inserir for maior, a função é chamada recursivamente para o nó a direita, caso a letra que eu quero inserir for menor, chamo a função recursivamente para o nó à esquerda, e caso a letra que eu quero inserir for igual a letra da chave em que eu me encontro, passo para a próxima letra da palavra que eu quero inserir e chamo a função recursivamente para o nó igual, que quer dizer que essa letra também se

encontra em uma palavra já existente na árvore. Caso o nó for nulo, faço a alocação de um novo nó na árvore. E se eu chegar no fim da palavra que eu quero inserir, o nó da última letra inserida, que no caso é a última letra da palavra, tem a variável *ehOFimDaString* recebendo valor 1.

Para imprimir palavras que estão na árvore, temos duas funções, *imprimeTST* e *imprimeTSTAux*. A primeira função serve apenas para criar um vetor de char, chamado de buffer, que iremos utilizar na função *imprimeTSTAux*. Na função *imprimeTSTAux*, temos também a variável depth (profundidade), e percorremos a árvore, pela esquerda, pelo igual e pela direita, nesta ordem. Durante esse percurso, guardamos a letra do nó que estamos dentro do vetor buffer, na posição depth, que começa com zero, e vamos entrando recursivamente no nó igual e incrementando o depth, caso o nó atual seja um fim de uma string, printamos a palavra que guardamos no buffer, e continuamos recursivamente até que o nó acessado seja nulo.

A função *pesquisarTST* é do tipo inteiro, e ela retorna 0 caso a palavra não esteja na árvore e 1 caso encontre a palavra na árvore. A função é semelhante a função *insereTSTAux,* comparando letra de cada nó como a função insere, e se a letra for igual, chama recursivamente para o apontador de nó igual, comparando letra por letra até chegar no final da palavra que eu quero procurar na árvore, nesse caso retornando 1, ou até chegar em um nó nulo, que nesse caso retorna 0, o que significa que a palavra não está na árvore.

Para contar as palavras presentes na árvore, foi feito duas funções, *contarPalavrasTST* e *contarPalavrasTSTAux.* A função *contarPalavrasTST*, tem o intuito apenas de inicializar uma variável chamada count para receber o resultado da função auxiliar, e chamar a função *contarPalavrasTSTAux*. A função auxiliar, retorna um inteiro, e funciona da seguinte forma, ela utiliza da variável *ehOFimDaString*, para saber se é igual a 1 (caso for, quer dizer que ali tem um término de uma palavra), para fazer incremento, e soma com essa mesma comparação só que para chamadas recursivas à direita e à esquerda, no final da execução dessa função, teremos o número de palavras presentes na árvore.

A função *abrirArquivo*, como o nome já sugere, é utilizada para abrir e ler os arquivos de texto, e também fazer a inserção das palavras que estão dentro do arquivo. Ela utiliza a função *strtok*, incluída no pacote *string.h*, que consiste em “cortar” uma frase/string por um delimitador que você definir, no nosso caso, esta

função foi utilizada em cada linha do arquivo, caso insira mais de uma palavra por linha. E durante esse processo, utilizando o strtok, é inserido na árvore as palavras presente no arquivo.

Por último, no TAD TST, temos a função *tranformarPalavra*, que utilizamos tanto no menu interativo, quanto na função *abrirArquivo*. Basicamente utilizamos a função para que haja um padrão na hora de inserção, pois, caso uma palavra seja inserida em letra maiúscula, e logo depois a mesma palavra seja inserida em letra minúscula, sem essa função, o programa reconhece que temos palavras diferentes, por isso da importância da implementação dessa função, pois não podemos ter a ocorrência de duas palavras iguais no dicionário.

Practical Algorithm To Retrieve Information Coded In Alphanumeric (PATRICIA)

Um nó da PATRICIA é representado pela struct *TipoPatNo*, que contém uma union *PatNo* e uma enumeração *TipoDoNo*, que pode receber os valores *interno* ou *externo* e é usada justamente para rotular um nó como sendo interno ou externo.

A union *PatNo* contém a struct *NoInterno* e um ponteiro para char *chave*, que aponta para a string armazenada pelo nó, se ele for externo. Por se tratar de uma union, apenas um deles pode estar alocados ao mesmo tempo, *NoInterno* ou *chave*.

Por fim, a struct *NoInterno* é composta por um unsigned char *indice*, um char *compara* e dois ponteiros para *TipoPatNo*, *esquerda* e *direita*. *indice* e *char* são usados para guardar a posição e a letra que aquele nó interno está diferenciando, respectivamente. A estrutura implementada, juntamente com a declaração da enumeração *TipoDoNo* pode ser vista na Figura 03.

```
typedef enum{
    interno, externo
}TipoDoNo;

typedef struct TipoPatNo{
    TipoDoNo notipo;
    union{
        struct{
            unsigned char indice;
            char compara;
            struct TipoPatNo *esquerda, *direita;
        }NoInterno;
        char *chave;
    }PatNo;
}TipoPatNo;
```

Figura 03. Estrutura do TAD PATRICIA

Em nossa implementação, convencionamos que toda string armazenada à direita de um nó interno terá o caractere *compara* na posição *indice*, e toda string armazenada à esquerda de um nó interno terá uma letra “menor” (ou seja, que vem antes no alfabeto) que *compara* na posição *indice*. Lembrando também que devido a natureza da PATRICIA, um nó externo nunca terá filhos, enquanto um nó interno sempre terá exatamente dois filhos que nunca serão nulos.

Na Figura 04, por exemplo, podemos ver que temos como raiz o nó interno “0,f” que tem como *indice* o número 0 e como *compara* a letra ‘f’. As chaves “fada” e “fogão” estão à direita desse nó, pois elas têm a letra ‘f’ no índice 0 da string. “batata” está à esquerda de “0,f” pois a letra que ela possui no índice 0 (no caso, a letra ‘a’) vem antes no alfabeto que a letra ‘f’.

De forma semelhante, à direita de “0,f” temos “1,o” que tem como *indice* o número 1 e como *compara* a letra ‘o’. “fogão” está à sua direita pois possui ‘o’ no índice 1, enquanto “fada” está à esquerda pois ‘a’ vem antes de ‘o’ no alfabeto.

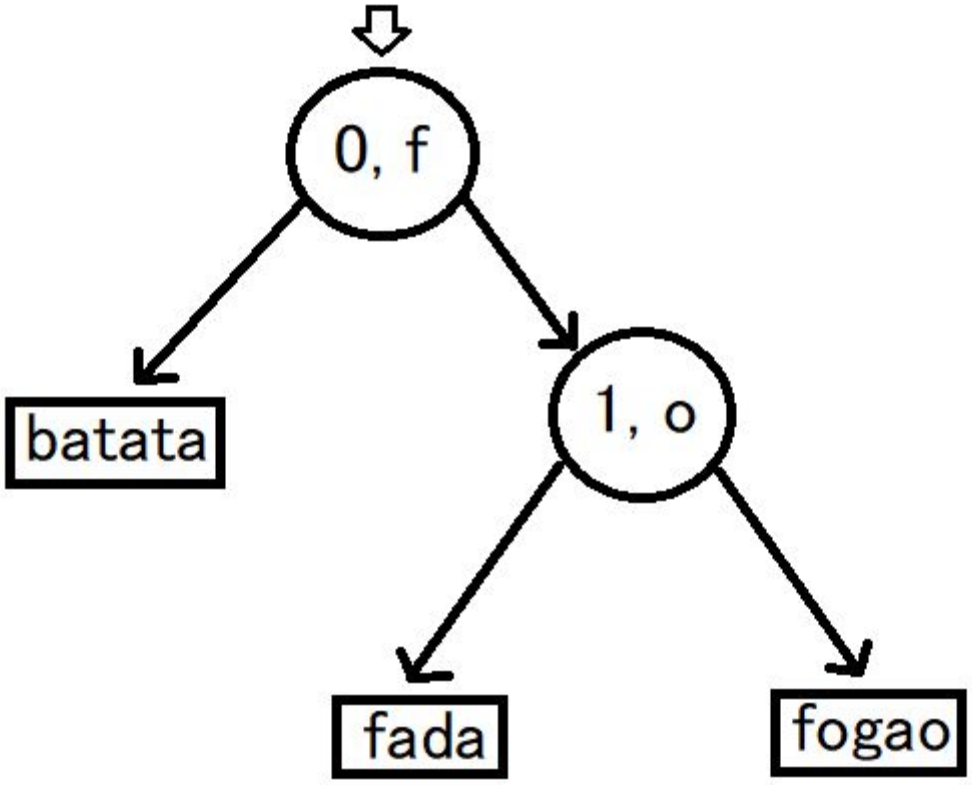

Figura 04. Exemplo de PATRICIA

A implementação da PATRICIA conta com quatro funções auxiliares, uma função de busca, duas de inserção, uma função que retorna a quantidade de palavras guardadas na árvore e uma função para imprimir todas as palavras inseridas. A declaração destas funções pode ser encontrada na Figura 05.

```
char Bit(unsigned char i, char k[]);
short NoEExterno(TipoPatNo *no);
TipoPatNo *CriaNoInt(int i, char c, TipoPatNo **esq, TipoPatNo **dir);
TipoPatNo *CriaNoExt(char *ch);
void BuscaPat (char k[], TipoPatNo *no);
TipoPatNo *InsereEntre (char k[], char compara, TipoPatNo **no, int i);
TipoPatNo *Insere(char k[], TipoPatNo **no);
int ContaPalavras(TipoPatNo *no);
void ImprimePalavras(TipoPatNo *no);
```

Figura 05. Funções do TAD PATRICIA

Começando pelas funções auxiliares, primeiramente temos a *Bit*, que recebe um vetor de char *k* e um indice *i*, retornando então o caractere na posição *i* de *k*.

NoEExterno consulta um nó e retorna verdadeiro caso ele seja externo.

CriaNoInt recebe um caractere *c*, um inteiro *i*, e o endereço de dois ponteiros para *TipoPatNo, esq* e *dir*. Em seguida, ele cria um nó externo com *compara* igual a *c*, *indice* igual a *i* e filhos à direita e esquerda *esq* e *dir*, respectivamente. Por fim, a função retorna o enderço de memória do nó interno criado.

Semelhantemente, *CriaNoExt* recebe um vetor de char *ch* e retorna um nó externo com *ch* como *chave*.

BuscaPat recebe um vetor de char *k* e o endereço de memória do nó raiz de uma árvore PATRICIA, *no*. Inicialmente, a função irá tentar verificar se o nó atual é externo. Caso ele seja, será feita uma comparação entre a chave do nó e o vetor de char *k*, para ver se são iguais. Se forem iguais, a função escreve que a chave foi encontrada, se não forem iguais, escreve que a chave não está na árvore.

No entanto, se o nó atual não for externo, a função irá fazer comparações para descobrir se deve chamar a si mesma recursivamente para o filho à esquerda ou à direita do nó atual.

A função então consulta o *indice* e o *compara* daquele nó. Se o caractere na posição *indice* do vetor de char *k* for menor que *compara*, a função chama a si mesma recursivamente, passando como parâmetros *k* e o filho à esquerda daquele nó. Caso contrário, a função também chama a si mesma recursivamente, porém passando como parâmetros *k* e o filho à direita daquele nó.

As funções *Insere* e *InsereEntre* trabalham em conjunto para realizar a inserção, ambas retornam um ponteiro para *TipoPatNo*. *Insere* é a função que será chamada no main, e recebe a string a ser inserida e o endereço de um ponteiro para o no raiz da árvore. Na maioria dos casos, a função *InsereEntre* que será

responsável por realizar as alocações e comparações para inserção (exceto na inserção da primeira chave da árvore, onde *InsereEntre* sequer será chamada).

É importante ressaltar que ao criar um ponteiro para *TipoPatNo* no main, ele deve inicialmente apontar para NULL, e ao utilizar a função *Insere*, além de passar tal ponteiro como parâmetro da função, também é necessário que ele receba o que essa função retorna. Por exemplo, supondo que foi criado um ponteiro para TipoPatNo *pat* no main, ao realizar uma inserção será necessário dizer que *pat* recebe Insere (k, &pat), e não apenas “Insere (k, &pat)”.

Começando pela função *Insere*, ela inicialmente verifica se o que está armazenado em *no* é nulo, e se for, ela entende que a árvore está vazia, cria um nó externo com chave *k* e retorna ele como raiz.

Caso a função não entre nessa condição, ela irá procurar um nó externo na árvore para comparar com a chave *k*. Para isso, a função usa um ponteiro auxiliar *Pno* para percorrer a árvore. Cada vez que *Pno* encontrar um nó interno, ele irá consultar seu *indice* e seu *compara*. Se o caractere na posição *indice* da chave *k* for igual a *compara*, a função percorre a árvore pela direita do nó, e se não, percorre a árvore pela esquerda. Essa sequência de instruções se repete até que *Pno* encontre um nó externo.

Ao encontrar um nó externo, a função *Insere* irá comparar a chave desse nó com a chave *k*, para encontrar em que posição elas se diferem, e irá armazenar este índice na variável *i*. Antes de prosseguir para chamar *InsereEntre*, é feito mais uma comparação entre as duas strings para certificar de que não são idênticas, pois não estamos trabalhando com a inserção de palavras iguais. Ao certificar que as strings são diferentes, a função retorna uma chamada de *InsereEntre*, passando como parâmetros a chave *k*, o i-ésimo caractere da chave do nó externo encontrado por *Pno*, a variável *no*, e *i*.

A função *InsereEntre* tem quatro cenários possíveis e receber um vetor de char *k*, um char *compara*, o endereço de memória de um ponteiro para *TipoPatNo* *no*, e um inteiro *i*.

A primeira condição é quando **no* aponta para um nó externo. Nesse caso, a função cria um ponteiro auxiliar *Pno*, que recebe um nó externo com chave *k*. Em seguida, será criado um nó interno com *indice* igual a *i*. Com a ajuda da função auxiliar *Bit*, a função *InsereEntre* verifica se i-ésimo caractere de *k* é maior que o i-ésimo caractere da chave armazenada pelo nó externo **no*. Se for, o nó interno

criado terá como *compara* o i-ésimo caractere de *k*, *Pno* como filho à direita e **no* como filho à esquerda. Caso contrário, *compara* será o i-ésimo caractere da chave de **no*, *Pno* será filho à esquerda e **no* será filho à direita. A função por fim retorna o no interno criado.

Nas demais três condições de *InsereEntre*, **no* irá apontar para um nó interno, e o que as difere é se *i* será maior, menor, ou igual ao *indice* do nó interno.

Se *i* for igual ao *indice*, entramos na segunda condição. O ponteiro auxiliar *Pno* novamente é criado e recebe um nó externo com chave *k*. Usamos a função *Bit* para verificar se o i-ésimo caractere de *k* é maior que o *compara* do nó interno **no*, e caso ele seja, retornamos um nó interno com *indice* igual a *i*, *compara* igual ao i-ésimo caractere de *k*, filho à esquerda **no* e filho à direita *Pno*. Caso contrário, o filho à esquerda do nó interno **no* recebe uma chamada recursiva de *InsereEntre* com parâmetros *k*, *compara* (OBS: esse *compara* é o da função *InsereEntre*, não do nó interno), o endereço do ponteiro para o filho à esquerda e *i*. Nesse caso a função retorna o próprio **no*.

Se *i* for menor que o *indice* do nó interno, entramos na terceira condição. Mais uma vez criamos o ponteiro auxiliar *Pno* e ele recebe um nó externo com chave *k*. De forma semelhante à primeira condição, aqui nós iremos criar e retornar um nó interno com *indice* igual a *i*. Usamos a função *Bit* para verificar se o i-ésimo caractere de *k* é maior que o char *compara* passado para a função *InsereEntre*. Caso seja verdadeiro o nó interno terá como *compara* o i-ésimo caractere de *k*, filho à direita *Pno* e filho à esquerda **no*. Caso contrário, o *compara* do nó interno criado será igual ao *compara* passado para a função *InsereEntre*, o filho à direita será **no* e filho à esquerda *Pno*. O nó interno criado é retornado pela função.

O único caso restante é quando *i* for maior que o *indice* do nó interno **no*, entrando na quarta condição da função *InsereEntre*. Quando entramos nessa condição, não vamos criar nenhum nó externo ou retornar um nó interno. A função irá consultar o *indice* e o *compara* do nó interno **no* e verificar se o caractere na posição *indice* da chave *k* é igual ao caractere *compara*. Se ele for, o filho à direita de **no* recebe uma chamada recursiva de *InsereEntre*, com parâmetros *k*, *compara*, o endereço do ponteiro para o filho à direita, e *i*. Se não, o filho à esquerda é quem recebe uma chamada recursiva de *InsereEntre* com parâmetros *k*, *compara*, o endereço do ponteiro para o filho à esquerda, e *i*. (OBS: novamente, nesse caso

compara se refere à variável da função *InsereEntre*, e não à variável *compara* do nó interno). Por fim, a função retorna **no*.

A função *ContaPalavras* retorna um inteiro e tem um funcionamento bastante intuitivo. A função começa verificando se o nó atual é externo, e se ele for, ela retorna 1. Caso o nó não for externo, a função retorna um chamada recursiva de *ContaPalavras* para o filho à esquerda mais uma chamada de *ContaPalavras* para o filho à direita.

Por fim, a função *ImprimePalavras*, como o nome sugere, imprime todas as palavras armazenadas na árvore, em ordem alfabética. Ela também começa verificando se o nó atual é externo, e caso ele for, ela imprime a chave armazenada. Caso não, ela faz uma chamada recursiva de *ImprimePalavras* para o filho à esquerda e em seguida faz uma chamada de *ImprimePalavras* para o filho à direita.

Testes

Foram realizados em cada árvore 4 testes diferentes considerando os casos de dicionários com palavras em ordem alfabética, fora de ordem, grupos de palavras com mesmo prefixo em ordem alfabética, e grupos de palavras com mesmo prefixo fora de ordem.

Os dados sobre a máquina utilizada e resultados obtidos estão apresentados abaixo:

Configurações do Computador Testado:
1 Processador: Intel(R) Core(™) i5-6200U CPU @ 2.39GHz
2 Memória RAM: 1 x 8GB @ 2.7GHz; DDR4 DRAM
3 Sistema Operacional: Microsoft Windows 10 Home (64-bit)

Dicionário de 1009 palavras em ordem alfabética (dicionario.txt):

Inserir dicionario.txt			
	Comparações	**Tempo**	**Memória Alocada**
TST	51193	2 ms	10732 bytes
PATRICIA	13031	1 ms	49216 bytes

Figura 06. Inserção do dicionario.txt

inserir Palavra Não Existente:	cavalo		
	Comparações	**Tempo**	**Memória alocada**
TST	38	<1 ms	16 bytes
PATRICIA	100	<1 ms	48 bytes

Figura 07. Inserção de uma palavra não existente na árvore.

Inserir Palavra Existente:	horse		
	Comparações	**Tempo**	**Memória alocada**
TST	21	3 ms	0 bytes
PATRICIA	27	1 ms	0 bytes

Figura 08. Tentativa de inserção de uma palavra existente na árvore.

Procura Palavra Existente :		cavalo
	Comparações	Tempo
TST	17	1 ms
PATRICIA	33	3 ms

Figura 09. Pesquisa de uma palavra existente na árvore.

Procura Palavra Não Existente:		pato
	Comparações	Tempo
TST	27	2 ms
PATRICIA	21	2 ms

Figura 10. Pesquisa de uma palavra não existente na árvore.

Dicionário de 1009 palavras fora de ordem (dicionario1.txt):

Inserir dicionario1.txt			
	Comparações	Tempo	Memória Alocada
TST	30424	3 ms	10732 bytes
PATRICIA	63197	2 ms	84512 bytes

Figura 11. Inserção do dicionario1.txt

inserir Palavra Não Existente:		cavalo	
	Comparações	Tempo	Memória alocada
TST	42	1 ms	16 bytes
PATRICIA	100	<1 ms	48 bytes

Figura 12. Inserção de uma palavra não existente na árvore.

Inserir Palavra Existente:	horse		
	Comparações	Tempo	Memória alocada
TST	18	1 ms	0 bytes
PATRICIA	27	2 ms	0 bytes

Figura 13. Tentativa de inserção de uma palavra existente na árvore.

Procura Palavra Existente :	cavalo	
	Comparações	Tempo
TST	19	2 ms
PATRICIA	33	3 ms

Figura 14. Pesquisa de uma palavra existente na árvore.

Procura Palavra Não Existente:	pato	
	Comparações	Tempo
TST	11	1 ms
PATRICIA	21	2 ms

Figura 15. Pesquisa de uma palavra não existente na árvore.

Dicionário de palavras com prefixos semelhantes em ordem alfabética (prefix.txt):

Inserir prefix.txt			
	Comparações	Tempo	Memória Alocada
TST	5547	7 ms	3124 bytes
PATRICIA	1350	2 ms	6896 bytes

Figura 16. Inserção do prefix.txt

inserir Palavra Não Existente:	international		
	Comparações	Tempo	Memória alocada
TST	57	<1 ms	32 bytes
PATRICIA	40	<1 ms	112 bytes

Figura 17. Inserção de uma palavra não existente na árvore.

Inserir Palavra Existente:	extract		
	Comparações	Tempo	Memória alocada
TST	13	1 ms	0 bytes
PATRICIA	17	<1 ms	0 bytes

Figura 18. Tentativa de inserção de uma palavra existente na árvore.

Procura Palavra Existente :	international	
	Comparações	Tempo
TST	15	<1 ms
PATRICIA	15	<1 ms

Figura 19. Pesquisa de uma palavra existente na árvore.

Procura Palavra Não Existente:	pato	
	Comparações	Tempo
TST	13	2 ms
PATRICIA	10	1 ms

Figura 20. Pesquisa de uma palavra não existente na árvore

Dicionário de palavras com prefixos semelhantes fora de ordem (prefix1.txt):

Inserir prefix1.txt			
	Comparações	Tempo	Memória Alocada
TST	4746	2 ms	3160 bytes
PATRICIA	3525	2 ms	8080 bytes

Figura 21. Inserção do prefix1.txt

inserir Palavra Não Existente:	international		
	Comparações	Tempo	Memória alocada
TST	55	<1 ms	32 bytes
PATRICIA	40	<1 ms	112 bytes

Figura 22. Inserção de uma palavra não existente na árvore.

Inserir Palavra Existente:	extract		
	Comparações	Tempo	Memória alocada
TST	16	2 ms	0 bytes
PATRICIA	17	<1 ms	0 bytes

Figura 23. Tentativa de inserção de uma palavra existente na árvore.

Procura Palavra Existente :	international	
	Comparações	Tempo
TST	22	2 ms
PATRICIA	15	2 ms

Figura 24. Pesquisa de uma palavra existente na árvore.

Procura Palavra Não Existente:	pato	
	Comparações	Tempo
TST	8	<1 ms
PATRICIA	10	<1 ms

Figura 25. Pesquisa de uma palavra não existente na árvore

Conclusão

Com a realização desse trabalho foi possível colocar em prática os conceitos aprendidos em aula. Ao implementar as árvores TST e PATRICIA fomos capazes de ter um melhor entendimento de seu funcionamento e suas diferenças. Analisando os dados dos testes nota-se que, nos quesitos observados, a árvore PATRICIA sempre utiliza uma quantidade maior de memória do que a árvore TST. Na inserção do dicionário a PATRICIA tem o menor número de comparações exceto no caso do dicionário fora de ordem. A árvore TST se mostra mais eficaz do que a PATRICIA em inserções de somente uma palavra, efetuando menos comparações nesses casos. Assim, entende-se que os pontos fortes da TST são a baixa utilização de memória e menor número de comparações na inserção de uma única palavra, já a PATRICIA lida melhor com várias palavras de uma vez, no geral é mais rápida, mas em contrapartida consome mais memória quase dobrando o seu valor quando se trata de um dicionário fora de ordem em relação ao ordenado.

Referências bibliográficas

TERNARY Search Tree. [S. l.], 22 mar. 2017. Disponível em: https://www.geeksforgeeks.org/ternary-search-tree/. Acesso em: 11 out. 2020.

ZIVIANI, Nivio. Projeto de algoritmos com implementações Pascal • C. 4. ad. São Paulo: Pioneira, 1999. p234-236

1000 Most Common Words in English. EF. Disponível em: https://www.ef.com/wwen/english-resources/english-vocabulary/top-1000-words/ Acesso em: 17 out. 2020.

35 Common Prefixes in English. NORDQUIST, Richard. 04 mai. 2019. Disponível em: https://www.thoughtco.com/common-prefixes-in-english-1692724 Acesso em 20 out. 2020