EP2 – Algoritmos e Estruturas de Dados II

Adriano Elias Andrade

21/04/2023

1.1 Informações gerais:

Compilação: Para compilar o ep, basta usar o comando “make”.

Entrada: A entrada do programa deve seguir a seguinte estrutura:

```
<Estrutura a ser utilizada (VO, ABB, TR, A23, ARN)>
<Número de palavras do texto>
<Texto(não pode conter palavras com acentos)>
<Número de consultas a serem feitas>
<Consultas a serem feitas (F, O palavra, L, SR, VD)>
```

1.1Entrada de teste geral:

Um exemplo de entrada de teste é:

```
VO
42
Esta frase tem 42 palavras. A palavra mais frequente eh
"palavra". A palavra mais longa eh "repeticoes". A maior
palavra que nao repete letra eh "vogais".A menor palavra
com mais vogais sem repeticoes eh "maior". A palavra
"palavra" ocorre 7 vezes.
5
F
L
SR
VD
O palavra
```

Este exemplo, para qualquer estrutura utilizada, deve gerar a saída:

```
palavra
repeticoes
vogais
maior
7
```

2. Estruturas utilizadas:

Para cada estrutura, foi feito um breve resumo da implementação, bem como um teste de eficiência para um texto (Lorem ipsum) com 1 milhão de palavras. Foram cronometrados os tempos de inserção e de verificação das palavras desse texto para cada estrutura.

2.1 Palavra.h:

Como todas as estruturas utilizam chave e valor das palavras, existe o arquivo palavra.h, comum a todas as classes.

Uma Key é apenas uma string. Já um item contém, além das três variáveis pedidas, as funções:

- Item() : construtor, inicializa as palavras com 0 ocorrências, letras e vogais.
- Item Inicializa(Key key): a partir da chave, retorna um item com o número correto de letras, vogais, e 1 ocorrência (é útil para inserir novas palavras nas estruturas).
- bool verificaSR(Key palavra, int tamanho): retorna se uma palavra não tem letras repetidas (utilizado na consulta “SR”).
- Int nVD(Key palavra, int tamanho): retorna o número de vogais diferentes numa palavra (utilizado na consulta “VD”).

2.2 Vetor ordenado:

Implementação: Para o vetor ordenado, cada espaço do vetor é um nó de vetor ordenado “noVO”, struct com chave e valor.

Um vetor ordenado guarda um ponteiro para o próprio vetor, o seu tamanho “size” (inicialmente 2), e o número de palavras nele, “nPalavras. Isso é útil para fazer buscas binárias nos valores, e também para dar resize.

Para adicionar as palavras no vetor, é feita uma busca binária. Se ela estiver no vetor, aumentamos suas ocorrências. Se não estiver, todas as casas à frente da sua posição são empurradas para frente, liberando um espaço para a nova.

Para a busca do valor de uma chave, é feita uma busca binária recursiva. Caso a chave não esteja no texto, é retornado um item “nulo”, com 0 ocorrências, 0 letras e 0 vogais (para todas as outras estruturas, é feito assim também).

Eficiência: Para inserir um texto de 1 milhão de palavras no vetor, demoram **4,1s**. Para verificar todas as palavras desse texto, demoram **3,3s** (verificação mais rápida).

2.3 Árvore de busca binária:

Implementação: Para a ABB, cada nó “noABB”, é uma struct com chave e valor, e ponteiros para os filhos esquerdo e direito.

Uma ABB possui um ponteiro para sua raíz, para que seja possível utilizar a função add como void (todas as outras árvores são feitas assim também).

Para adicionar as palavras na árvore, é feita uma busca binária iterativa, e a palavra é inserida como folha, ou tem suas ocorrências incrementadas.

Para a busca do valor de uma chave, é feita uma busca binária recursiva.

Eficiência: Para inserir um texto de 1 milhão de palavras na ABB, demoram **4,1s**. Para verificar todas as palavras desse texto, demoram **5,4s**.

2.4 Treap:

Implementação: Para a treap, cada nó “noTreap”, é uma struct com chave e valor, ponteiros para os filhos esquerdo e direito, e seu valor de heap. O valor de heap é sorteado na função construtora do nó, e está entre 0 e 10000.

Para adicionar as palavras na treap, é feita uma busca binária recursiva, e a palavra é inserida como folha, ou tem suas ocorrências incrementadas. Em seguida, o nó inserido é consertado recursivamente de acordo com o heap, com rotações.

Para a busca do valor de uma chave, é feita uma busca binária recursiva.

Eficiência: Para inserir um texto de 1 milhão de palavras na treap, demoram **20,9s**. Para verificar todas as palavras desse texto, demoram **3,8s**.

2.5 Árvore 2-3:

Implementação: Para a 2-3, cada nó “no23”, é uma struct com 2 chaves e 2 valores, ponteiros para os filhos esquerdo, meio e direito, e uma flag para indicar se é 3-nó (um 2-nó possui apenas filhos esquerdo e direito, um 3-nó possui o filho do meio).

Para adicionar as palavras na 2-3, é feita uma busca binária recursiva, e a palavra é inserida como folha, ou tem suas ocorrências incrementadas. Durante a inserção, são guardadas as direções em que a palavra está sendo inserida, em cada nó (na esquerda, no meio, ou na direita), o que é útil na hora de corrigir a árvore. Em seguida, o nó inserido é consertado recursivamente de modo a respeitar as regras da rubro-negra. Quando um 3-nó cresce, essa informação é passada para o nó de cima (que já sabe qual dos seus filhos cresceu, durante a inserção), até que chegue num 2-nó ou na raíz.

Para a busca do valor de uma chave, é feita uma busca trinaria recursiva.

Eficiência: Para inserir um texto de 1 milhão de palavras na 2-3, demoram **3,9s** (inserção mais rápida). Para verificar todas as palavras desse texto, demoram **4,4s**.

2.6 Árvore rubro-negra:

Implementação: Para a rubro-negra, cada nó "noRN", é uma struct com chave e valor, ponteiros para os filhos esquerdo e direito, e uma flag para indicar se é um nó preto.

Para adicionar as palavras na 2-3, é feita uma busca binária recursiva, e a palavra é inserida como folha, ou tem suas ocorrências incrementadas. Em seguida, o nó inserido é consertado recursivamente de modo a respeitar as regras da rubro-negra. Essa correção é feita a partir de um avô, que consegue enxergar algum problema com os netos (2 vermelhos seguidos), bem como ter acesso aos tios deles.

Para a busca do valor de uma chave, é feita uma busca binária recursiva.

Eficiência: Para inserir um texto de 1 milhão de palavras na rubro-negra, demoram **6,0s**. Para verificar todas as palavras desse texto, demoram **4,9s**.

3. consultas extras:

Para as consultas "F", "L", "SR", "VD", foram utilizados vectors, já que é necessário fazer várias operações de push_back e clear para as palavras de cada categoria. Inicialmente era utilizado um vetor normal, mas como isso trazia problemas para alocação de memória, foi utilizada essa classe da std.

Além dos vectors, são guardadas em variáveis, informações de cada categoria (para "F" o número de repetições da palavra mais frequente, para "L" o número de letras na palavra mais longa, ...).