Enzo Oliveira Fernandes
NUSP: 12563752

RELATÓRIO EP2- AED2

INTRODUÇÃO:

Este EP implementa uma Tabela de Símbolos em diferentes estruturas de dados além das seguintes operações sobre a tabela: encontrar palavra mais frequente, encontrar a ocorrência de uma dada palavra, encontrar as palavras mais longas, encontrar as maiores palavras que não repetem letras e as menores palavras que mais possuem vogais. A linguagem de programação utilizada foi **JAVA.**

COMO RODAR

Durante o desenvolvimento deste EP foi utilizada a IDE Eclipse. Nessa IDE é necessário criar um “projeto java”, que contém classes, pacotes e outros elementos importantes no desenvolvimento. Um dos arquivos enviados se chama **ep2AED2.tar.gz**, neste arquivo está a pasta do “projeto java” utilizado no Eclipse. Para abrir a pasta e o projeto no Eclipse é necessário ir em File > Import > General > Existing Projects into Workspace > Select Root Directory -> selecionar a pasta do projeto. Após isso o projeto será aberto no eclipse e para rodá-lo basta apertar F11.

Também foi enviado um outro arquivo chamado **somenteClasses.tar.gz**, neste arquivo há somente os arquivos .java utilizados, caso você prefira abrir em outra IDE.

INPUTS ESPERADOS:

A ordem dos inputs estão de acordo com o enunciado do EP na seguinte ordem:
estrutura a ser usada -> quantidade de palavras -> palavras -> quantidade de operações -> operações. O output foi ligeiramente modificado para facilitar o entendimento dos testes, foram apenas inseridos mais detalhes das operações. Por exemplo, ao invés de só printar a maior palavra, é printada a seguinte frase “As maiores palavras do texto contém x letras, são elas:”

SOBRE AS CLASSES

Foram escritas 5 classes, são elas:

Main.java -> Classe que possui o método main. Além do main, há 4 métodos que são invocados dentro do main de acordo com a estrutura de dados selecionada (rodaVO, rodaBST, rodaARN e rodaTreap). Há também 2 métodos auxiliares para saber se as palavras que serão inseridas na tabela de símbolos repetem letras e a quantidade de vogais.
SymbolTable_VetorOrdenado -> Classe que implementa a tabela de símbolos com vetor ordenado.
SymbolTable_BST -> Classe que implementa a tabela de símbolos com uma árvore binária de busca.
SymbolTable_RubroNegra -> Classe que implementa a tabela de símbolos com uma árvore rubro negra.
SymbolTable_Treap -> Classe que implementa a tabela de símbolos com uma Treap.

EXPLICAÇÃO DE ALGUNS PONTOS DE CADA IMPLEMENTAÇÃO:

Vetor Ordenado-> Foram criados 5 vetores para guardar as seguintes informações: nome da palavra, quantidade de ocorrências da palavra, tamanho da palavra, número de vogais sem repetição da palavra e um vetor de booleanos que guarda se a palavra repete ou não letras. O índice de cada um desses vetores diz respeito a um mesmo item. O vetor é ordenado de acordo com a ordem alfabética das palavras. Segue um exemplo:

TEXTO INSERIDO: eu gosto de estrutura de dados
VETORES:
nome das palavras -> [eu , dados, de, estrutura, gosto]
qtd de ocorrencias -> [1, 1, 2, 1, 1]
tamanhoPalavra -> [2, 5, 2, 9, 5]
vogaisSemRepetir -> [2, 2, 1, 3, 1]
repeteLetras -> [false,true ,false ,true ,true]

A palavra é inserida na posição correta e todos os vetores são atualizados dinamicamente, após cada inserção. Sempre que o vetor fica cheio, o método (resize) aumenta o tamanho de cada um dos vetores em 100 unidades.

Árvore Binária de Busca -> Foi feita uma pequena “personalização” nesta árvore. Cada nó dela possui os seguintes atributos: nome da palavra, quantidade de ocorrencias da palavra, tamanho da palavra, numero de vogais sem repetição, filho esquerdo e filho direito. Segue um exemplo

TEXTO INSERIDO: Amazonas Bahia Cruzeiro Amazonas;

ÁRVORE:

Nó 1:	Nó 2:	Nó 3
key = “Amazonas”	key = Bahia	key = Cruzeiro
ocorrencias = 2	ocorrencias = 1	ocorrencias = 1
tamanho = 8	tamanho = 5	tamanho = 8
vogaisSemRep = 2	vogaisSemRep = 2	vogaisSemRep = 4
repeteLetras = true	repeteLetras = true	repeteLetras = true
filhoEsquerdo = null	filhoEsquerdo = null	filhoEsquerdo = null
filhoDireito = Nó 2	filhoDireito = Nó 3	filhoDireito = null

O critério de comparação para saber se um elemento é filho esquerdo ou direito é a ordem alfabética da palavra.

Árvore Rubro Negra - > Segue a mesma lógica da ABB, porém os nós também possuem os atributos cor e pai.

Treap -> Segue a mesma lógica da ABB, porém os nós também possuem um atributo de prioridade.

Algumas pequenas mudanças no fluxo dos algoritmos originais de cada estrutura foram realizadas, elas serão explicadas no próximo tópico.

PONTO IMPORTANTE SOBRE AS OPERAÇÕES

Durante o desenvolvimento do EP pensei em duas possibilidades para realizar as operações:.

Possibilidade 1: Durante a inserção do elemento, já fazemos uma checagem para saber se ele é o maior elemento já inserido/ maior sem repetição/ menor sem vogais/ mais frequente.
Prós: mais eficiente, pois quando as inserções são terminadas, já possuímos a resposta para as operações, exceto a operação “O”,
Contra: Código mais difícil de entender, método de inserção fica muito extenso.

Possibilidade 2: Realizamos as instruções e depois percorremos a estrutura inteira para encontrar as respostas das operações.
Prós: Código mais legível, há funções especificamente para realizar as operações.
Contra: Mais lento, é necessário percorrer toda a estrutura para realizar uma operação.

Fiquei curioso para ver o quão mais lento seria a primeira possibilidade e decidi mesclar as implementações de acordo com as diferentes estruturas de dados:
Possibilidade 1 -> Treap, Rubro Negra
Possibilidade 2 -> Vetor ordenado, ABB

TESTES:

Para testar a performance dos algoritmos, utilizei um texto de 700 palavras, presente no arquivo **teste.txt** e obtive o tempo gasto entre o momento em que o último input é recebido até o output final, obtendo os seguintes resultados:

VETOR ORDENADO -> 29ms
ÁRVORE BINÁRIA DE BUSCA -> 25ms
TREAP -> 17ms
ÁRVORE RUBRO NEGRA -> 2ms

Dois pontos principais me chamaram a atenção nos testes. O primeiro foi a esmagadora diferença de performance da rubro negra comparada às demais estruturas. Esperava sim obter uma execução mais rápida na rubro-negra porém não dessa maneira. Outro ponto que surpreendeu foi a proximidade entre o Vetor Ordenado e a Árvore Binária de Busca, pois eu esperava um gap menor entre estas duas estruturas. Um dos possíveis motivos dessa pequena diferença é que, apesar de ambas estruturas usarem a possibilidade mais lenta de realizar as operações, a ABB é mais prejudicada pois ela utiliza recursão para percorrer seus elementos, o que acaba sendo mais lento pois a recursão adiciona um *overhead* devido às numerosas chamadas de função, o que não ocorre no vetor ordenado já que ele é 100% iterativo.

SOBRE A ÁRVORE 2-3

A árvore 2-3 não está no EP pois tive dificuldades para implementá-la e considerando a carga das outras disciplinas optei por não finalizá-la pois o tempo que gastaria poderia faltar para trabalhos importantes porém mais simples de outras disciplinas. Uma pena, pois gostaria de ter implementado a 2-3 com a possibilidade mais lenta de realização das operações e comparar com a Rubro-Negra, que obteve um ótimo desempenho utilizando a possibilidade mais rápida de realizar as operações.