Relatório EP2 MAC0323

Matheus Sanches Jurgensen (12542199)

26/05/2022

1. RESUMO

Relatório do 2º Exercício-Programa da disciplina MAC0323: Algoritmos e Estruturas de Dados II. Deve-se implementar uma tabela de símbolos ordenada, que armazene a frequência das palavras de um texto. Para isso, foram solicitadas cinco diferentes implementações para o problema, utilizando cinco diferentes estruturas de dados: Vetor Ordenado, Árvore de Busca Binária, Treap, Árvore 2-3 e Árvore Rubro-Negra. Em relação aos métodos, a tabela de símbolos deve contar, pelo menos, com os quatro a seguir:

(1) void add (Key key, Item val);
(2) Item value (Key key);
(3) int rank (Key key);
(4) Key select (int k);

A entrega deste projeto será feita por meio de um arquivo de tipo zip, contendo os seguintes arquivos: EP2_MAC0323.java, ST.java, ST_Array.java, ST_BST.java, ST_Treap.java, ST_23.java, ST_RB.java, Node.java, NodeBST.java, NodeTreap.java, Node23.java, NodeRB.java, 10k.txt, 100k.txt e 1M.txt. Com exceção do primeiro arquivo, todos os outros com extensão .java implementam diferentes classes necessárias para este projeto, e que serão melhor explicadas adiante, neste relatório. O arquivo EP2_MAC0323.java é o arquivo principal, e ele também será explicado mais a fundo nas páginas a seguir. Por fim, os três últimos arquivos, de extensão .txt, são arquivos de teste, contendo textos de 10 mil, 100 mil e (quase) 1 milhão de palavras, respectivamente. Todos esses arquivos de texto estão - sem nenhum motivo em específico - fazendo chamadas aos métodos de uma tabela de símbolos implementada como árvore rubro-negra. Isso pode ser facilmente alterado, de uma forma que também será explicada mais adiante (basta alterar a primeira linha do arquivo).

É importante destacar que o único arquivo que deve ser compilado e executado é o arquivo EP2_MAC0323.java. Por consequência, todos os outros serão automaticamente compilados e executados.

2. A IMPLEMENTAÇÃO

Em primeiro lugar, é importante destacar alguns aspectos gerais sobre a implementação do exercício. Em primeiro lugar, a linguagem utilizada foi Java. Dessa forma, criou-se um arquivo para cada classe implementada.

Para as classes, por sua vez, fez-se uso, em primeiro lugar, do conceito de herança. Como, com exceção do vetor ordenado, todas as outras implementações

são estruturadas através do conceito de nó, criou-se uma classe genérica Node, à qual deu-se um conjunto de sub classes (que implementam a ideia de nó para cada um dos tipos de árvore). Node recebeu apenas atributos comuns a todos os quatro diferentes tipos de nós necessários: uma chave, um valor, e um inteiro que guarda a quantidade de nós na sub-árvore enraizada nele (e, portanto, contando com ele).

Ademais, para as classes que implementam a tabela de símbolos, criou-se uma interface, armazenada no arquivo ST.java, indicando os quatro métodos a serem implementados, além de um extra - Item search (Key key) -, o qual, recebendo uma possível chave da tabela de símbolos, faz a busca por ela e retorna o seu valor (ou *null*, quando a tabela de símbolos não a contiver).

2.1 VETOR ORDENADO

Para a implementação de uma tabela de símbolos a partir de um vetor ordenado, criou-se a classe chamada ST_Array. Ela faz uso de um vetor de capacidade dinâmica, o qual é inicializado com tamanho unitário e realiza a sua expansão conforme a necessidade (sempre que a sua capacidade atinge o limite, ela é dobrada). Essa classe tem, como atributos, primeiramente, dois arrays, sendo que o primeiro deles (keys) armazena as chaves contidas na tabela de símbolos, enquanto o segundo (items) armazena os valores dessas chaves (em posições equivalentes). Ademais, existem dois inteiros, sendo que o primeiro armazena o tamanho do array (size), e o segundo, o índice do maior elemento armazenado no array (top).

O construtor de um objeto dessa classe inicia os dois arrays vazios e de tamanho unitário. Além disso, top é inicializado como -1, afinal, como a tabela ainda não contém nenhuma chave, ainda não existe um elemento de maior chave. Portanto, inicializar esse atributo como -1 (que seria, em Java, uma posição inválida do array) é mostrar que esse maior elemento ainda não existe.

Destaca-se que, como o array é ordenado, quando uma chave é inserida nele, todas aquelas que são maiores devem ser deslocadas, no array, uma posição para a direita. Isso ocorre, também, no array de valores, de maneira idêntica ao de chaves.

2.2 ÁRVORE DE BUSCA BINÁRIA

Agora, visando à implementação de uma tabela de símbolos utilizando uma árvore de busca binária, foi criada a classe ST_BST. Toda a implementação dessa classe é estruturada a partir de sua unidade de armazenamento básica, NodeBST. Essa é umas das subclasses de Node e, além de armazenar uma chave, o seu valor e o número de nós em sua sub-árvore, ele faz referência a seu pai e seus filhos da esquerda e da direita.

A implementação dos métodos dessa classe é feita da maneira que é convencionalmente atrelada a uma árvore de busca binária. Em geral, faz-se a busca de um elemento a partir de sua comparação com a raiz, descendo os níveis da árvore de acordo com o valor da chave a ser encontrada comparada com aquelas percorridas. Quando a chave sendo buscada (e potencialmente inserida) é maior do que a atual, segue-se para o filho da direita. Caso seja menor, para a esquerda. Essa metodologia é em grande parte mantida pelos outros tipos de árvore, que, em sua grande maioria, reaproveitam métodos como rank() e select(), ou, até, fazem busca e inserção de maneira similar.

Falando-se mais especificamente sobre os métodos de rank() e select() para árvores de busca binária, mas sem grande detalhamento sobre a maneira como eles são implementados, utilizam-se de estratégias recursivas, iniciadas na raiz e seguindo recursivamente para níveis inferiores, dependendo enormemente do número de nós das subárvores contidas na árvore principal.

2.3 TREAP

As treaps são estruturalmente muito similares às árvores de busca binária convencionais, com a diferença de que seus nós, implementados na classe NodeTreap, além de conter referências a seus filhos e pai, contém um valor inteiro de prioridade. Esse valor, gerado aleatoriamente, interfere em como será posicionada uma chave na árvore. Além de se preocupar com a grandeza da chave em si, a sua inserção deve respeitar a seguinte regra: nós com número de prioridade maior devem estar em níveis superiores (mais próximos da raiz) em comparação com nós de menor prioridade.

Para que seja respeitada essa regra, após ser feito um processo de inserção convencional de uma árvore de busca binária na treap, compara-se o número de prioridade do nó inserido com o de seu pai. Caso ele seja maior, é necessário que o pai seja rotacionado para que essa situação seja solucionada. Quando o nó está à esquerda de seu pai, este deve ser rotacionado para a direita. Agora, quando o nó está à direita, a rotação deve ocorrer para a esquerda. Isso deve ser feito em direção ascendente na árvore, até que se encontre um pai com prioridade menor que seu filho.

Um objeto da classe ST_Treap, além de ter, como atributo, a raiz (da mesma forma que todas as outras árvores implementadas), contém um inteiro que indica aproximadamente o número de palavras máximo que essa tabela pode armazenar. Isso ocorre porque, como cada nó deve receber um inteiro que identifica a sua prioridade, é necessário saber, de maneira aproximada, qual o tamanho do intervalo de onde serão retirados os números aleatórios. É evidente que nós diferentes podem receber, mesmo que aleatoriamente, números de prioridades iguais. Isso, se não muito recorrente, prejudica pouco a estruturação da treap, e armazenar o número máximo esperado de nós busca justamente evitar isso.

O número de prioridade é gerado a partir da chamada do método Random().nextInt(n_palavras). Esse método, da biblioteca Random, retorna, de maneira aleatória, um inteiro entre 0 e n_palavras. Por isso, como essa chamada ocorre no momento da construção de um nó da classe NodeTreap, o construtor dela também deve receber, como argumento, o número máximo de nós esperado.

O restante dos métodos (value, rank e select) é implementado de maneira idêntica àquele das árvores de busca binária não-aleatorizadas.

2.4 ÁRVORE 2-3

Dessa vez, para a implementação de uma tabela de símbolos a partir de uma árvore 2-3, é necessária uma estrutura básica de armazenamento (contida na classe Node23) mais complexa. Essa estrutura é um nó que, além de conter um primeiro par chave-valor e o número de nós da subárvore enraizada nele, contém um segundo par chave-valor, referências a seu pai e seus filhos à esquerda, direita e centro, e, por último, um booleano que indica se aquele é um 2-nó (*true*) ou um 3-nó (*false*).

Caso o nó for um 2-nó, estado indicado por um booleano de valor *true*, o segundo par de chave e valor tem seu conteúdo ignorado. Além disso, o filho central recebe valor *null*. Agora, quando o nó é um 3-nó, armazena-se no centro a referência para um filho cujas chaves estão entre as duas chaves do nó (maior do que a primeira e menor do que a segunda). Portanto, é evidente que o segundo par chave-valor, para um 3-nó (caso em que o booleano é *false*), não tem seu conteúdo desconsiderado.

É importante destacar que foram implementados três diferentes construtores para um nó de uma árvore 2-3. Em primeiro lugar, tem-se o construtor de um 2-nó, que recebe apenas o primeiro par chave-valor. Em segundo, tem-se um primeiro construtor de um 3-nó, recebendo os dois pares. Por fim, tem-se um segundo construtor para um nó com dois pares de chave e valor, mas, agora, recebendo, como parâmetro, valores para todos os seus atributos. Assim, nesse último construtor, as referências ao pai e aos filhos não são inicializadas como *null*.

Nesse momento, é importante destacar que um tipo extra de nó é implementado no código da classe ST_23 (que é quem, de fato, implementa uma tabela de símbolos a partir de uma árvore 2-3). Esse nó, chamado de FourNode, é uma subclasse de Node23, com a diferença de que contém um par chave-valor e um filho extras. Esse nó não está contido na definição de uma árvore 2-3. Por isso, ele é apenas um estado intermediário de um 3-nó que recebe um novo conjunto de chave e valor e deve, em seguida, sofrer um split (a chave de valor intermediário deve ser separada das outras duas e ascender um nível na árvore, para que tente ser adicionada no nó pai).

Dessa forma, a inserção na árvore 2-3 ocorre de maneira bem clara: em primeiro lugar, busca-se em qual folha a chave deve tentar se inserir (em um processo bem similar à busca de uma árvore de busca binária convencional, mas, agora, levando-se em consideração que podem existir dois pares de chave e valor e três filhos em um nó). Encontrada essa chave, tenta-se a inserção em seu nó, a qual é facilmente concluída caso ele seja um 2-nó. Agora, em caso de uma tentativa de inserção em um 3-nó, constrói-se, momentaneamente, um 4-nó, apenas para que a chave de valor intermediário ascenda e tente ser inserida em seu pai. A partir daí, o processo de inserção - ou tentativa dela - se repete da mesma forma.

Pode-se chegar a um caso em que, tentando-se inserir uma chave, todos os seus antepassados foram percorridos e nenhuma posição para inserção foi encontrada. Nesse caso, um novo nó é elevado à posição de raiz, levando ao crescimento da árvore.

Por fim, pensando nos outros três métodos da tabela de símbolos a serem implementados, eles repetem em boa parte aquilo que é feito para uma árvore de busca binária, mas, agora, tomando precauções quanto à existência de possíveis dois pares de chave e valor e de um filho central adicional.

2.5 ÁRVORE RUBRO-NEGRA

Finalmente, tem-se a última implementação para a tabela de símbolos: a árvore rubro-negra. Ela é muito similar a uma árvore de busca binária, com a diferença de que cada um de seus nós - definidos na classe NodeRB - armazenam um booleano que indica a sua cor (*true*, em caso de vermelho, e *false*, caso contrário). Como convenção, nesta implementação utiliza-se a raiz da árvore sempre como preta.

Para esse tipo de árvore, os seus métodos são implementados de maneira muito similar às árvores de busca binária - value(), rank() e select() chegam a ser praticamente iguais. Quanto à inserção, ela inicialmente também ocorre de maneira igual, apenas com o detalhe de que um nó é sempre inserido como vermelho. O que muda é que, para que as regras das árvores rubro-negras sejam respeitadas, algumas rotações e mudanças de cor se fazem necessárias. Para isso, algumas regras são estabelecidas, com um conjunto de ações que devem ser tomadas:

(1) Se o nó inserido for raiz, ele deve ser colorido de preto.
(2) Se o pai do nó inserido for vermelho, existem 3 casos:
    (a) Se o tio desse nó for vermelho, alteram-se as cores de seu pai, tio e avô.
    (b) Se o tio for preto (ou *null*) e o nó, seu pai e avô estiverem dispostos de maneira triangular, o pai deve ser rotacionado para o sentido contrário do nó (se ele está à esquerda, a rotação deve ocorrer para a direita).

(c) Se o tio for preto (ou *null*) e o nó, seu pai e avô estiverem dispostos em linha, o avô deve ser rotacionado para o lado oposto do pai do nó, e ambos (pai e avô) devem ter suas cores alteradas.

Essas regras são analisadas, a partir de onde ocorrer a inserção, até níveis acima onde não se encontre mais nenhuma infração de regra da árvore rubro-negra.

2.6 ARQUIVO PRINCIPAL

Por fim, detalha-se o arquivo principal, nomeado como EP2_MAC0323.java. Em primeiro lugar, é importante destacar o papel da função createInstance(). Essa função, recebendo, em um String, qual a implementação de tabela de símbolos que deve ser utilizada, além do número de palavras a ser lido e potencialmente inserido na tabela (número este que, como já dito, é importante para a Treap), cria, inicialmente, uma tabela de símbolos de tipo genérico. Depois disso, seguindo a implementação específica indicada no String recebido como argumento, essa tabela genérica é convertida para a específica.

Agora, é importante destacar o processo de interação com o usuário, para que o programa possa receber todos os dados que são necessários para o seu funcionamento. De maneira geral, esses dados são fornecidos por meio de um arquivo de texto. Dessa forma, primeiramente, é solicitado ao usuário o endereço desse arquivo, o qual deve seguir uma estruturação específica.

Falando sobre essa estruturação, o arquivo deve conter, na primeira linha, um indicativo para qual a implementação de tabela de símbolos que deverá ser usada. Esses indicativos podem ser os seguintes: VO (vetor ordenado), ABB (árvore de busca binária), TR (treap), A23 (árvore 2-3) e ARN (árvore rubro-negra).

Na linha seguinte, deve ser dado o número de palavras no texto que será lido. O texto, por sua vez, deverá vir nas linhas subsequentes, com número de palavras igual ao dado anteriormente. Finalizado o texto, na próxima linha, deve estar indicado o número de operações que serão feitas na tabela de símbolos. Essas operações serão, então, incluídas nas linhas em sequência, cada linha contendo apenas uma operação.

Voltando, agora, a falar sobre o código do arquivo principal, ele recebe todos esses dados do usuário a partir de objetos da classe Scanner, que possibilita o recebimento de input. Conforme o texto dado é lido, ele é armazenado em um array de Strings, cujo tamanho equivale ao número de palavras do texto. Em seguida, é criada a tabela de símbolos do tipo indicado na primeira linha do arquivo, fazendo a chamada do método createInstance(). Por fim, cada uma das operações no arquivo é lida e executada, e o tempo de execução de cada uma delas é indicado.

3. TESTES

3.1 CORRETUDE

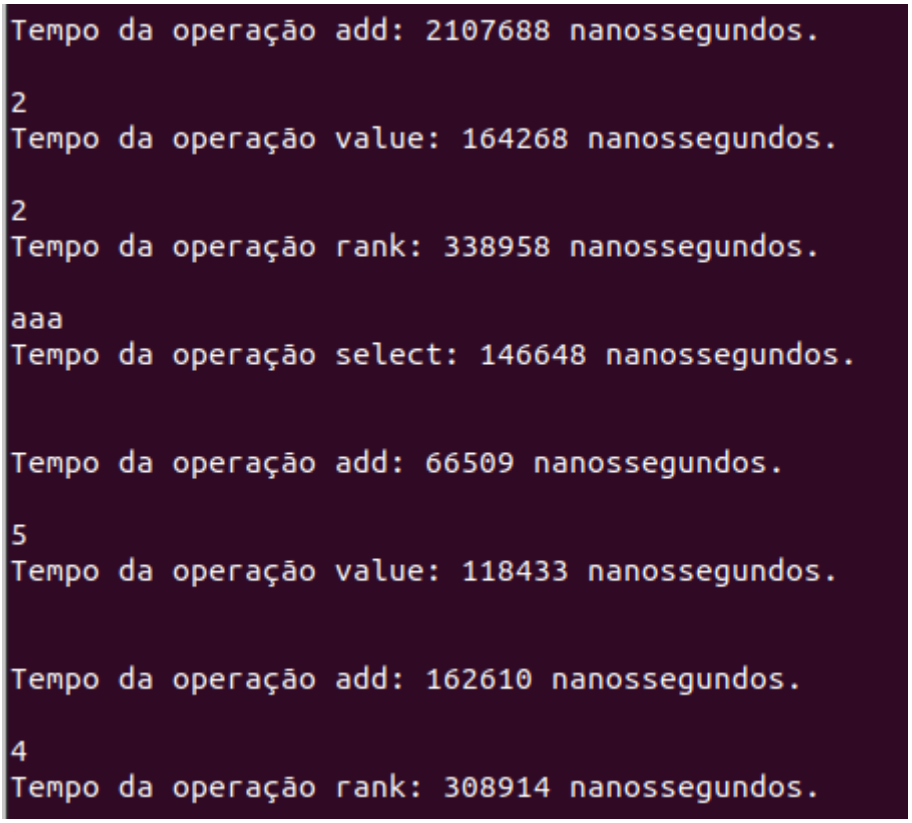
Em primeiro lugar, é importante testar as classes e seus métodos quanto à corretude das saídas que geram. Para isso, pode-se, primeiramente, realizar testes iguais aos do enunciado e comparar as saídas geradas. O primeiro deles, um caso teste de vetor ordenado com um texto de Machado de Assis, obteve, utilizando a classe ST_Array, a saída a seguir, que é a mesma dada no enunciado, com a diferença de que são exibidos os tempos de execução.

```
Tempo da operação add: 89441 nanossegundos.

ao
Tempo da operação select: 34643 nanossegundos.

1
Tempo da operação rank: 52167 nanossegundos.
```

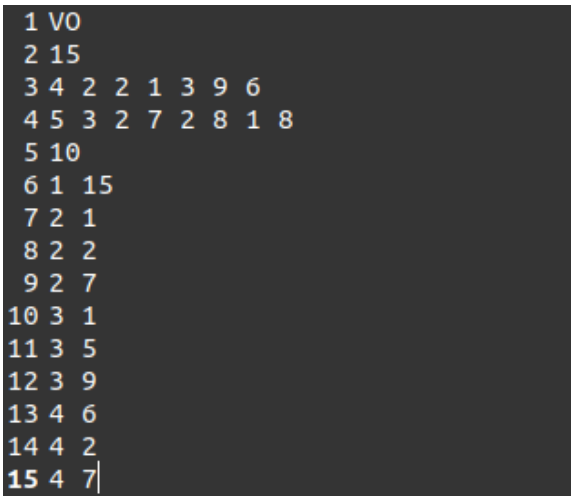
Agora, para o segundo caso concreto de teste, utilizando uma árvore 2-3, obteve-se a seguinte saída, a qual, mais uma vez, é a mesma do enunciado, porém incluídos os tempos de execução.

```
Tempo da operação add: 72926 nanossegundos.

1
Tempo da operação value: 49836 nanossegundos.


Tempo da operação add: 163164 nanossegundos.

2
Tempo da operação value: 27392 nanossegundos.
```

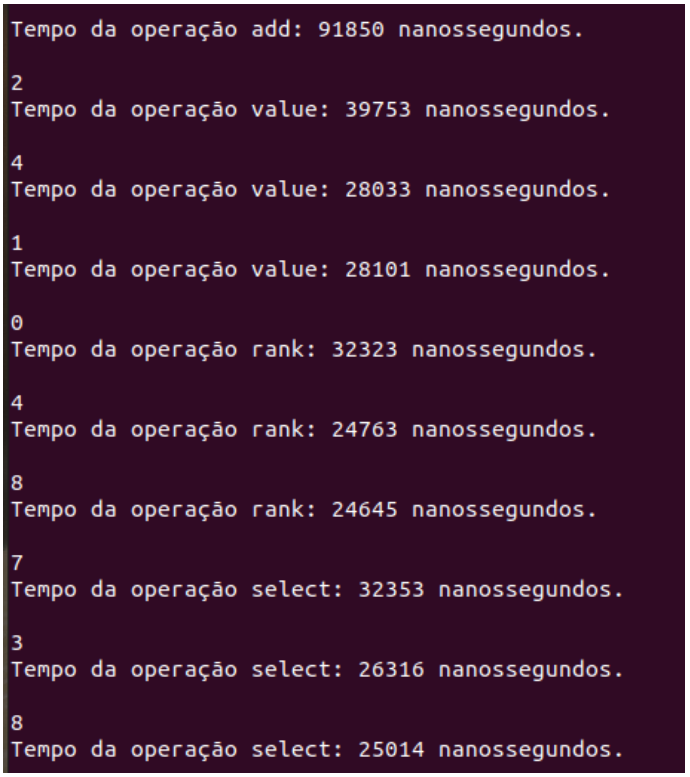
Por fim, tem-se o último caso de teste, inserindo palavras e caracteres aleatórios em uma tabela de símbolos implementada como árvore rubro-negra. Novamente, a saída do programa é a mesma do enunciado, adicionados os tempos.



Além disso, pode-se averiguar a corretude das classes e seus métodos a partir de um teste simples, adicionando-se números entre 1 e 9 na tabela de símbolos (apesar de a tabela ter a intenção de contabilizar palavras em um texto, e não inteiros, pode-se, por praticidade, fazer esse teste com inteiros apenas para analisar se o programa retorna os valores corretos). Adicionam-se quatro vezes a chave 2 e duas vezes as chaves 1, 3 e 8. Todas as outras (4, 5, 6, 7 e 9) são adicionadas apenas uma vez, conforme se pode ver na imagem abaixo, do arquivo dado como entrada do programa (nele, a implementação da tabela de símbolos escolhida é a para vetores ordenados, mas esses mesmos testes serão feitos, em seguida, para todas as implementações).



É fácil garantir que as operações rank e select estão retornando os valores corretos, pois, considerando a tabela que foi construída, o rank de um inteiro N entre 1 e 9 será sempre N - 1. Agora, quanto à operação select, dado a ela um valor k entre 0 e 8, ela sempre retornará k + 1. Abaixo, seguem as saídas do programa para, respectivamente, vetores ordenados, árvore de busca binária, treaps, árvore 2-3 e árvore rubro-negra.



Saída do programa para vetor ordenado

```
Tempo da operação add: 327184 nanossegundos.

2
Tempo da operação value: 32785 nanossegundos.

4
Tempo da operação value: 22654 nanossegundos.

1
Tempo da operação value: 24835 nanossegundos.

0
Tempo da operação rank: 34235 nanossegundos.

4
Tempo da operação rank: 25574 nanossegundos.

8
Tempo da operação rank: 23485 nanossegundos.

7
Tempo da operação select: 43043 nanossegundos.

3
Tempo da operação select: 24346 nanossegundos.

8
Tempo da operação select: 22525 nanossegundos.
```

Saída do programa para árvore de busca binária

```
Tempo da operação add: 481997 nanossegundos.

2
Tempo da operação value: 39952 nanossegundos.

4
Tempo da operação value: 31354 nanossegundos.

1
Tempo da operação value: 24186 nanossegundos.

0
Tempo da operação rank: 33900 nanossegundos.

4
Tempo da operação rank: 25944 nanossegundos.

8
Tempo da operação rank: 23514 nanossegundos.

7
Tempo da operação select: 32474 nanossegundos.

3
Tempo da operação select: 23282 nanossegundos.

8
Tempo da operação select: 21972 nanossegundos.
```

Saída do programa para treap

```
Tempo da operação add: 125630 nanossegundos.

2
Tempo da operação value: 32580 nanossegundos.

4
Tempo da operação value: 24302 nanossegundos.

1
Tempo da operação value: 24814 nanossegundos.

0
Tempo da operação rank: 31886 nanossegundos.

4
Tempo da operação rank: 28355 nanossegundos.

8
Tempo da operação rank: 26109 nanossegundos.

7
Tempo da operação select: 31244 nanossegundos.

3
Tempo da operação select: 24846 nanossegundos.

8
Tempo da operação select: 23764 nanossegundos.
```

Saída do programa para árvore 2-3

```
Tempo da operação add: 391036 nanossegundos.

2
Tempo da operação value: 35191 nanossegundos.

4
Tempo da operação value: 25540 nanossegundos.

1
Tempo da operação value: 26582 nanossegundos.

0
Tempo da operação rank: 35917 nanossegundos.

4
Tempo da operação rank: 24789 nanossegundos.

8
Tempo da operação rank: 27693 nanossegundos.

7
Tempo da operação select: 45415 nanossegundos.

3
Tempo da operação select: 23036 nanossegundos.

8
Tempo da operação select: 21521 nanossegundos.
```

Saída do programa para árvore rubro-negra

3.2 EFICIÊNCIA

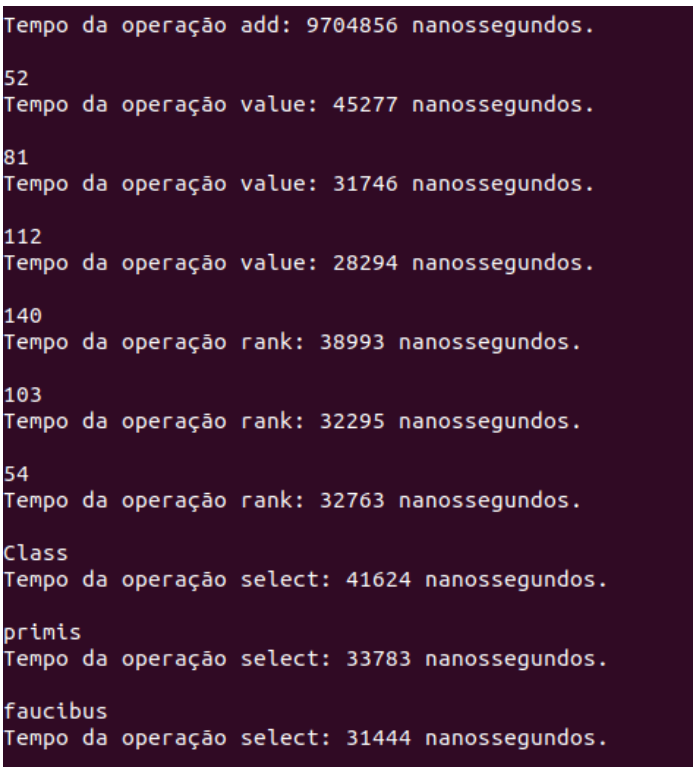
Por fim, sabendo que as classes e seus métodos funcionam de maneira correta, resta testar essas implementações para textos maiores, visando à análise da eficiência do algoritmo. Em primeiro lugar, realizam-se testes para um texto de 10 mil palavras geradas aleatoriamente, do tipo *lorem ipsum* (10k.txt). O arquivo de entrada do programa, indicando quais devem ser as execuções no código, contém os seguintes comandos:

```
10
1 10000
2 lorem
2 posuere
2 ipsum
3 pellentesque
3 libero
3 cubilia
4 2
4 151
4 79
```

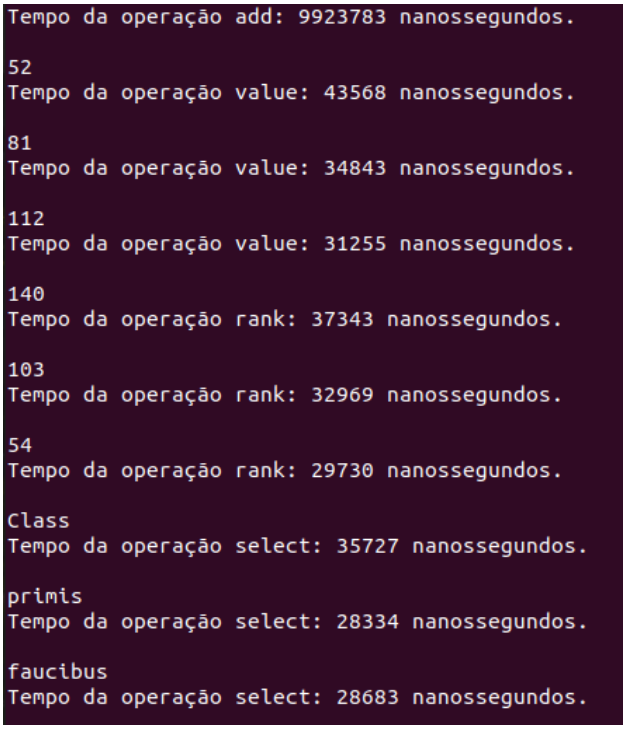
As saídas obtidas constam a seguir, para todas as cinco implementações.

```
Tempo da operação add: 9780950 nanossegundos.

52
Tempo da operação value: 48465 nanossegundos.

81
Tempo da operação value: 35538 nanossegundos.

112
Tempo da operação value: 32522 nanossegundos.

140
Tempo da operação rank: 36249 nanossegundos.

103
Tempo da operação rank: 37785 nanossegundos.

54
Tempo da operação rank: 31810 nanossegundos.

Class
Tempo da operação select: 37023 nanossegundos.

primis
Tempo da operação select: 31070 nanossegundos.

faucibus
Tempo da operação select: 31706 nanossegundos.
```

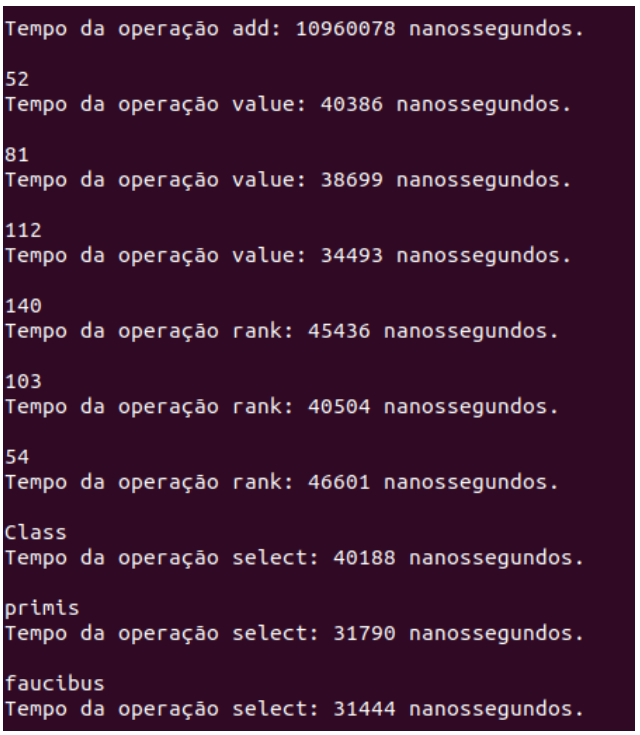
Saída do programa para vetor ordenado



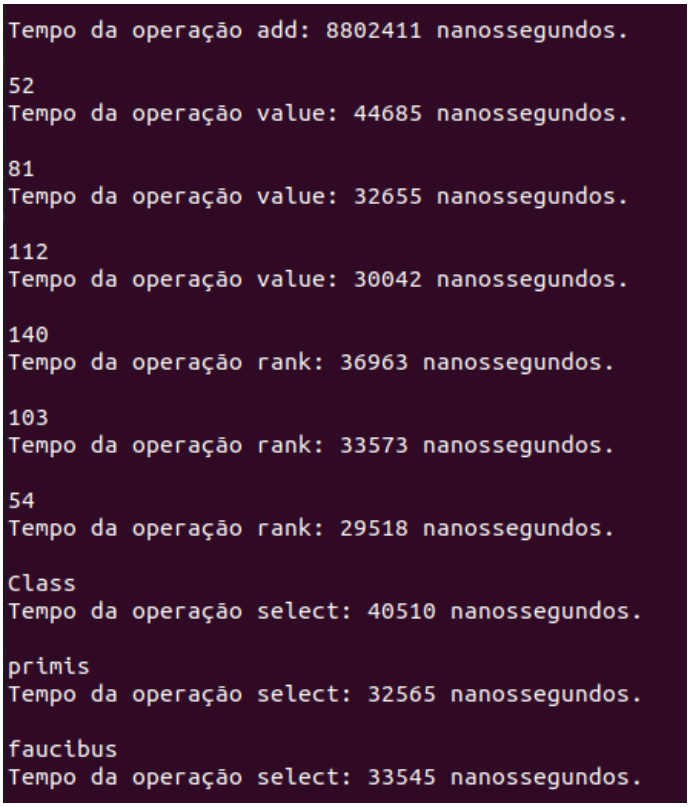
Saída do programa para árvore de busca binária



Saída do programa para treap



Saída do programa para árvore 2-3



Saída do programa para árvore rubro-negra

Agora, serão realizados testes para um texto de 100 mil palavras, pertencentes às mais diferentes línguas e culturas (100k.txt). Os comandos a serem executados, dados pelo arquivo de entrada, constam a seguir.

```
10
1 100000
2 lights
2 the
2 apple
3 irregular
3 noise
3 at
4 7
4 89
4 500
```

A seguir, apresentam-se as saídas do programa, fazendo a execução para as cinco diferentes implementações.

```
Tempo da operação add: 36292140 nanossegundos.

53
Tempo da operação value: 66661 nanossegundos.

3307
Tempo da operação value: 53857 nanossegundos.

0
Tempo da operação value: 59885 nanossegundos.

831
Tempo da operação rank: 23609 nanossegundos.

1004
Tempo da operação rank: 38778 nanossegundos.

360
Tempo da operação rank: 53499 nanossegundos.

Alex
Tempo da operação select: 37046 nanossegundos.

Fourth
Tempo da operação select: 34347 nanossegundos.

curse
Tempo da operação select: 19233 nanossegundos.
```

Saída do programa para vetor ordenado

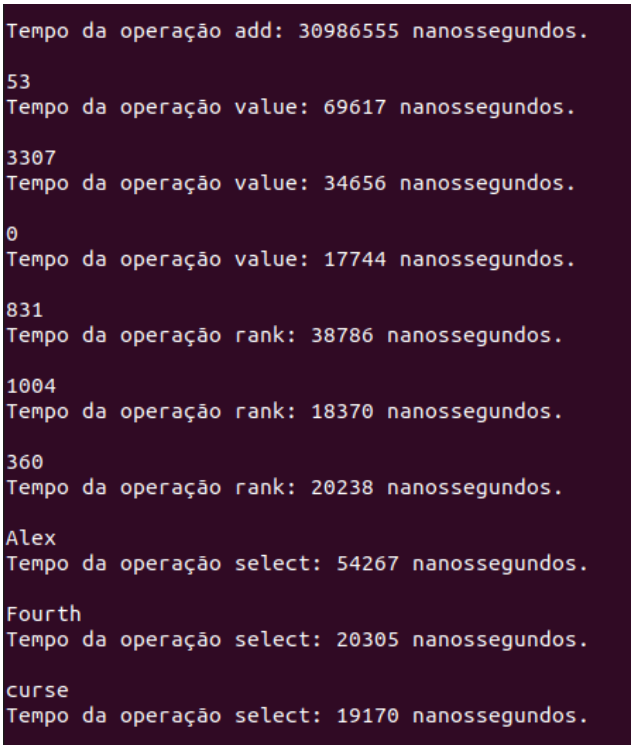
```
Tempo da operação add: 32437073 nanossegundos.

53
Tempo da operação value: 33133 nanossegundos.

3307
Tempo da operação value: 26158 nanossegundos.

0
Tempo da operação value: 32330 nanossegundos.

831
Tempo da operação rank: 60462 nanossegundos.

1004
Tempo da operação rank: 34948 nanossegundos.

360
Tempo da operação rank: 20914 nanossegundos.

Alex
Tempo da operação select: 25327 nanossegundos.

Fourth
Tempo da operação select: 19895 nanossegundos.

curse
Tempo da operação select: 20875 nanossegundos.
```

Saída do programa para árvore de busca binária

```
Tempo da operação add: 33293584 nanossegundos.

53
Tempo da operação value: 92686 nanossegundos.

3307
Tempo da operação value: 57770 nanossegundos.

0
Tempo da operação value: 33950 nanossegundos.

831
Tempo da operação rank: 54995 nanossegundos.

1004
Tempo da operação rank: 20850 nanossegundos.

360
Tempo da operação rank: 19220 nanossegundos.

Alex
Tempo da operação select: 38702 nanossegundos.

Fourth
Tempo da operação select: 19340 nanossegundos.

curse
Tempo da operação select: 18812 nanossegundos.
```

Saída do programa para treap

Saída do programa para árvore 2-3



Saída do programa para árvore rubro-negra

Por fim, realizam-se testes para um arquivo com 924.701 palavras da língua portuguesa (1M.txt). O arquivo de entrada contém os seguintes comandos:

```
10
1 924701
2 em
2 Bernadino
2 vigiados
3 desejar
3 promulgada
3 classe
4 147
4 53500
4 60000
```

Os resultados obtidos constam a seguir.

```
Nome do arquivo de comandos: test8.txt

Tempo da operação add: 39099164546 nanossegundos.

10222
Tempo da operação value: 34674 nanossegundos.

1
Tempo da operação value: 21868 nanossegundos.

2
Tempo da operação value: 67102 nanossegundos.

85294
Tempo da operação rank: 55129 nanossegundos.

104022
Tempo da operação rank: 20531 nanossegundos.

80901
Tempo da operação rank: 20738 nanossegundos.

0510
Tempo da operação select: 21759 nanossegundos.

Bang
Tempo da operação select: 27918 nanossegundos.

Fortim
Tempo da operação select: 19169 nanossegundos.
```

Saída do programa para vetor ordenado

```
Tempo da operação add: 344284493 nanossegundos.

10222
Tempo da operação value: 34046 nanossegundos.

1
Tempo da operação value: 54299 nanossegundos.

2
Tempo da operação value: 34066 nanossegundos.

85294
Tempo da operação rank: 29588 nanossegundos.

104022
Tempo da operação rank: 50858 nanossegundos.

80901
Tempo da operação rank: 22619 nanossegundos.

0510
Tempo da operação select: 28156 nanossegundos.

Bang
Tempo da operação select: 25231 nanossegundos.

Fortim
Tempo da operação select: 25371 nanossegundos.
```

Saída do programa para árvore de busca binária

```
Tempo da operação add: 639034732 nanossegundos.

10222
Tempo da operação value: 47647 nanossegundos.

1
Tempo da operação value: 52188 nanossegundos.

2
Tempo da operação value: 37052 nanossegundos.

85294
Tempo da operação rank: 29637 nanossegundos.

104022
Tempo da operação rank: 22399 nanossegundos.

80901
Tempo da operação rank: 23318 nanossegundos.

0510
Tempo da operação select: 24996 nanossegundos.

Bang
Tempo da operação select: 23843 nanossegundos.

Fortim
Tempo da operação select: 22697 nanossegundos.
```

Saída do programa para treap

```
Tempo da operação add: 419733450 nanossegundos.

10222
Tempo da operação value: 69774 nanossegundos.

1
Tempo da operação value: 33110 nanossegundos.

2
Tempo da operação value: 18430 nanossegundos.

85294
Tempo da operação rank: 25965 nanossegundos.

104022
Tempo da operação rank: 36894 nanossegundos.

80901
Tempo da operação rank: 20593 nanossegundos.

0510
Tempo da operação select: 24872 nanossegundos.

Bang
Tempo da operação select: 34228 nanossegundos.

Fortim
Tempo da operação select: 34460 nanossegundos.
```

Saída do programa para árvore 2-3

```
Tempo da operação add: 601471321 nanossegundos.

10222
Tempo da operação value: 48877 nanossegundos.

1
Tempo da operação value: 19986 nanossegundos.

2
Tempo da operação value: 19660 nanossegundos.

85294
Tempo da operação rank: 26773 nanossegundos.

104022
Tempo da operação rank: 29652 nanossegundos.

80901
Tempo da operação rank: 26405 nanossegundos.

0510
Tempo da operação select: 38112 nanossegundos.

Bang
Tempo da operação select: 46697 nanossegundos.

Fortim
Tempo da operação select: 28566 nanossegundos.
```

Saída do programa para rubro-negra