MAC0323
Algoritmos e Estruturas de Dados II
EP2

Nathan Luiz
Universidade de São Paulo

19/05/2022

1 Introdução

Esse Exercício Programa tem como objetivo testar diferentes implementações de uma tabela de símbolos ordenada. Cada implementação contém pelo menos as 4 funções a seguir:

- `void add (Key key, Item val)` – adiciona (key, val) na estrutura;
- `Item value (Key key)` – retorna o valor da chave key;
- `int rank (Key key)` – retorna o número de chaves estritamente menor do que key;
- `Key select (int k)` – retorna a k-ésima chave em ordem crescente, indexado do 0.

Ao todo 5 implementações diferentes foram feitas. São elas:

- vetor dinâmico ordenado;
- árvore de busca binária;
- treaps;
- árvore 2-3;
- árvore rubro-negra.

O projeto completo se encontra no github. Basta clonar o repositório e usar os comandos `make generate_tests` e `make run_tests` para criar e rodar os casos teste, respectivamente.

2 Implementação

Cada tabela de símbolos foi feita utilizando Templates. É necessário que o objeto `Key` tenha função de comparação implementada para tudo funcionar.

2.1 Vetor dinâmico ordenado

Foi utilizado `vector`, um vetor dinâmico implementado na STL, para guardas os elementos da tabela de símbolo. Também, foi criada a função `find_first_bigger`, que acha o primeiro elemento maior ou igual a uma Key com uma busca binária, já que o vetor se mantém sempre ordenado. Essa função garante que `rank` e `value` funcionem em $O(logN)$. A função `add` funciona em $O(N)$ no pior caso, e a função `select` em $O(1)$, já que basta acessar o k-ésimo elemento de um array.

2.2 Árvore de Busca Binária

Na implementação da ABB, foi criada uma subclasse que guarda o tipo de nó da árvore. Temos ponteiros que ligam para o filho esquerdo, filho direito e pai. O atributo `peso` guarda o tamanho da sub-árvore do nó. Ele é essencial para que algumas funções fiquem eficientes. Foi criada uma variável global da classe `Node *root`, que guarda a raiz do objeto todo. Para todas as outras árvores, foi criada também uma sub-classe parecida com essa. Além disso, as funções `rank`, `select` e `value` funcionam da mesma maneira em todas as árvores. Como nós temos os pesos de cada subárvore, conseguimos fazer a implementação das funções acima em $O(h)$, onde h é a altura da árvore. Em casos teste aleatórios, como a altura esperada da árvore é $logN$, então essa implementação fica boa.

2.3 Treap

Na subclasse do nó, temos um atributo a mais que é diferente da ABB. Definimos como `long long prioridade`, que é um número gerado aleatoriamente cada vez que o objeto é criado. Quando inserimos um elemento, devemos sempre manter nós com prioridade menor em cima de nós com prioridade maior. Para garantir isso, temos que poder rotacionar algum nó para esquerda ou direita. Por isso, foram criadas as funções `rotate_right` e `rotate_left`. O resto da implementação é bem parecida com a ABB.

2.4 Árvore Rubro-Negra

Nesta estrutura, a subclasse nó possui um atributo `int cor`, que define a cor do nó. Se `cor = 0`, então o nó é vermelho e, caso contrário, o nó é preto. Foi usada a convenção de que a raiz do objeto é sempre preta. Além da variável global `Node *root`, foi criada a variável global `Node *NULO` que é um nó que representa `NULL`. A diferença é que pintamos `NULO` de cor preta e evitamos ter que fazer alguns casos de borda a mais. As funções `rotate_right` e `rotate_left` foram criadas e fazem a mesma coisa que na Treap, rotacionar o nó. Elas são necessárias para manter as priopriedades da ARN.

2.5 Árvore 23

Esta estrutura foi com certeza a mais trabalhosa de fazer. A subclasse nó tem algumas diferenças. Primeiro, como cada nó pode ter 2 pares (chave, valor), então foram criados `key_l, val_l` e `key_r, val_r`. Além dos ponteiros para o filho esquerdo, direito e pai, tem mais um ponteiro apontando para o filho do meio. Caso o nó só tenha uma chave, o filho direito é sempre `NULL`. Além disso, foi criada a variável `int Node_2` indicando se o nó tem 2 ou 3 filhos.

Como nas outras árvores, temos uma variável global `Node *root` que é a raiz do objeto, mas também criamos a variável `Node *subiu`, que ajuda na função de inserir. Sempre que tentamos inserir em um nó, e ele está saturado, então o elemento do meio 'sobe' pela variável `subiu`.

3 Testes realizados

Foram utilizados 7 textos diferentes para analisar o desempenho de cada estrutura. Para cada texto, criamos 3 conjuntos de casos teste: o primeiro conjunto são testes aleatórios, mas sem repetição de palavras; o segundo são testes com as palavras da entrada ordenadas, também sem repetição de palavras; e por último o conjunto de casos teste aleatórios, mas podendo repetir palavras. Como são 7 textos e 5 estruturas diferentes, foram 35 testes gerados. Além disso, para cada caso teste temos $3 \cdot 10^5$ operações (entre os tipos $2, 3, 4$) definidas de forma aleatória.

3.1 Vetor dinâmico ordenado

Vamos analisar primeiro o conjunto de teste em que as palavras estão ordenadas aleatoriamente, mas sem repetição. Observe que o tempo de execução cresce aproximadamente quadrático. Isso se deve ao fato de a função `add` fazer N operações no pior caso, e $\frac{N}{2}$ no caso médio. Pelo fato de as outras funções serem rápidas, o tempo de execução não fica tão grande, mesmo com o número alto de operações.

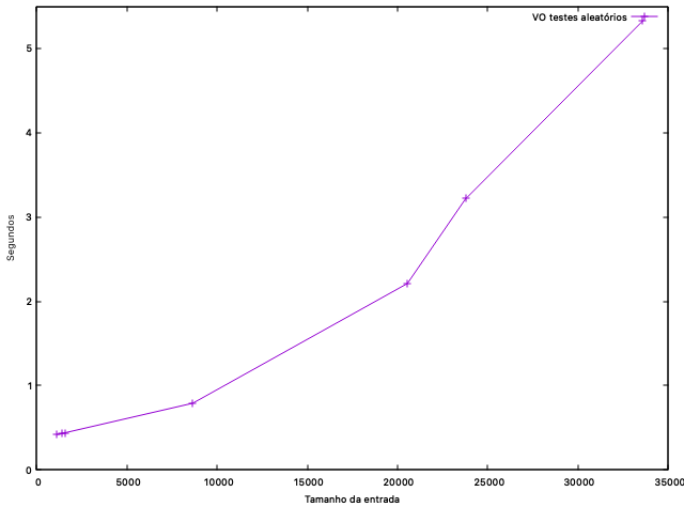

Figura 1: Conjunto de testes com palavras ordenadas aleatoriamente, sem repetição

Olhando para o conjunto de testes em que a entrada está ordenada, Caímos sempre no pior caso da função *add*. É interessante que o tempo de execução fica aproximadamente o dobro do passado, já que fazemos N operações a cada inserção. As outras operações não alteram tanto o tempo de execução.

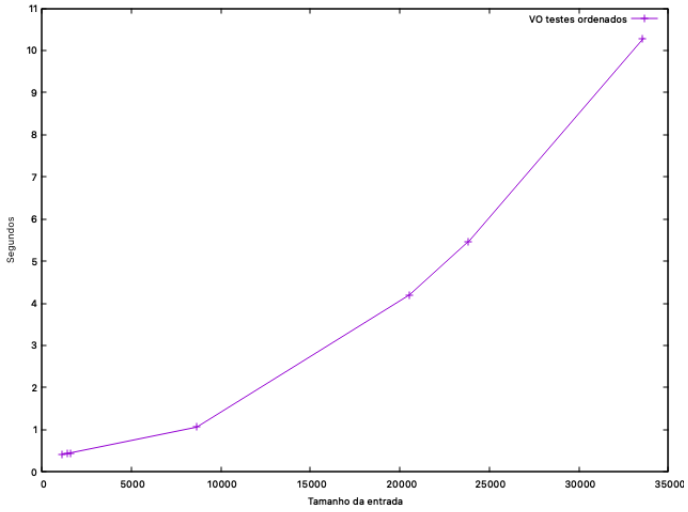

Figura 2: Conjunto de testes com palavras ordenadas, sem repetição

Já o conjunto de testes aleatórios podendo conter palavras repetidas, observamos algo interes-

sante. Tem um teste com mais palavras, mas roda bem mais rápido que um teste com menos palavras. Isso acontece pois, quando a chave se encontra na tabela de símbolos, a função `add` faz apenas $logN$ operações. Ou seja, esse caso possui várias palavras repetidas. Em específico, ele foi gerado no estilo *lore ipsum*. Observe que o tempo de execução aumenta pouco em relação ao primeiro conjunto de testes.

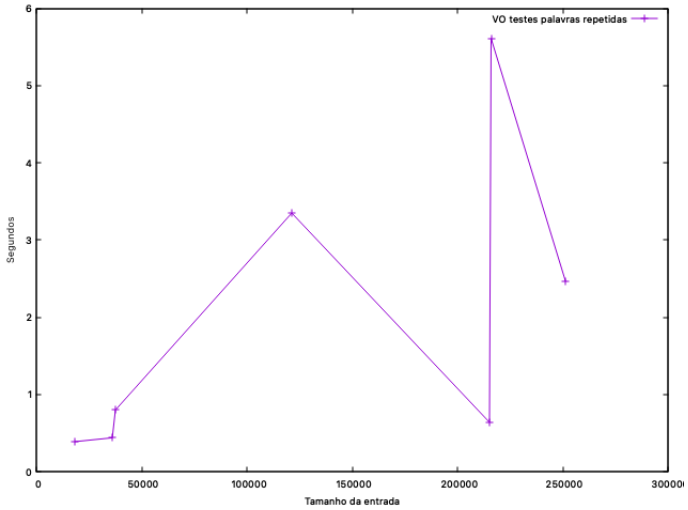

Figura 3: Conjunto de testes com palavras ordenadas aleatoriamente, com repetição

3.2 Árvore de Busca Binária

Para esta estrutura, o que mais afeta o desempenho é quando a entrada é dada de maneira que a altura da árvore fique linear, como vamos observar nos casos teste. Todas as operações tem complexidade $O(h)$, em que h é a altura da árvore. Portanto, pode ficar bem lento. No primeiro conjunto de casos teste, como a altura esperada da árvore é $O(logN)$, então tudo é executado rapidamente.

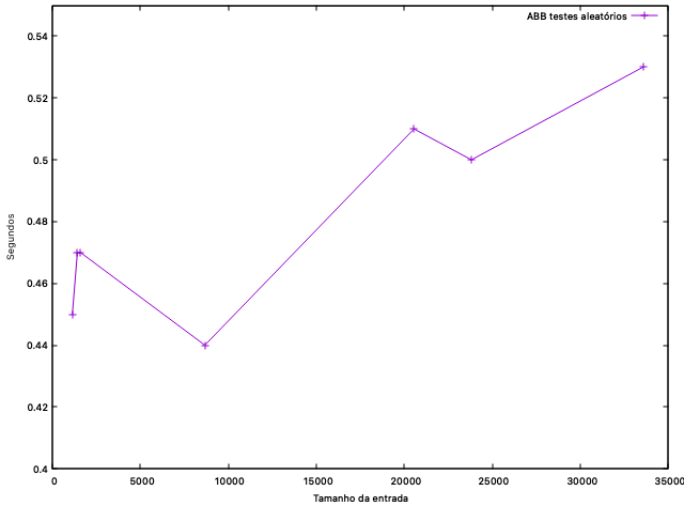

Figura 4: Conjunto de testes com palavras ordenadas aleatoriamente, sem repetição

É nesse conjunto de testes que podemos ver o quão ruim pode ficar o desempenho desta estrutura. O gráfico parece linear. A questão é que cada operação é feita em $O(N)$. Como o número de operações é bem maior que N, então a complexidade fica $O(N \cdot Q + N^2) = O(N \cdot Q)$, onde Q é o número de operações. O maior caso levou mais de 100 segundos para ser executado.

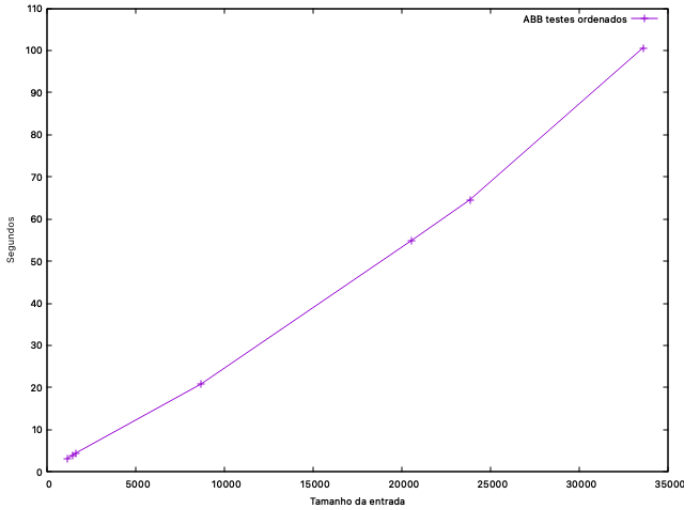

Figura 5: Conjunto de testes com palavras ordenadas, sem repetição

O terceiro conjunto de casos teste não diz muita coisa. Basicamente a altura da árvore tende a ser $logN$, portanto tudo roda rápido.

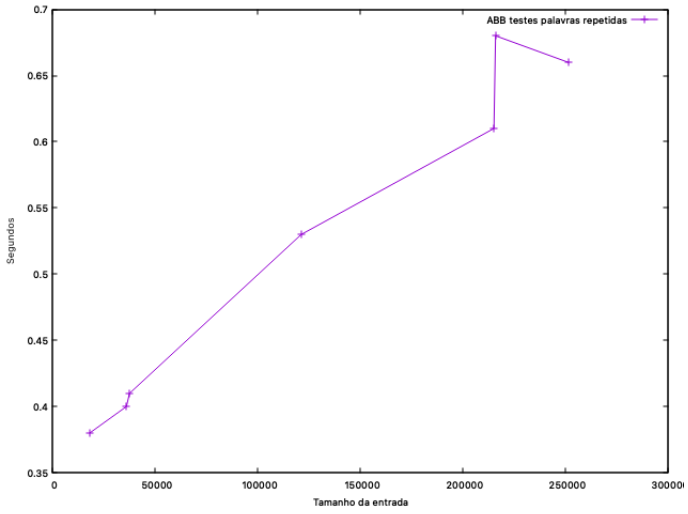

Figura 6: Conjunto de testes com palavras ordenadas aleatoriamente, com repetição

3.3 Treap

A partir de agora, todas as estruturas são boas tabelas de símbolo. Na nossa Treap, o valor da prioridade é um número aleatório até 10^9, portanto para entradas não tão grandes a complexidade

esperada de cada operação é $O(logN)$. Observe que nesse conjunto de casos, o tempo de execução é bem semelhante com o da ABB.

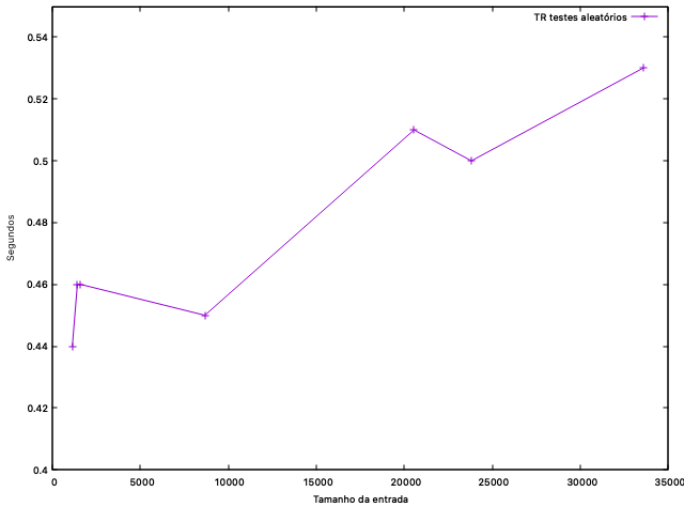

Figura 7: Conjunto de testes com palavras ordenadas aleatoriamente, sem repetição

Já no segundo conjunto de casos, vemos que a Treap é bem melhor que as estruturas apresentadas anteriormente. Como a altura esperada é logarítmica, então todas as operações funcionam em $O(logN)$, independente do tipo de entrada.

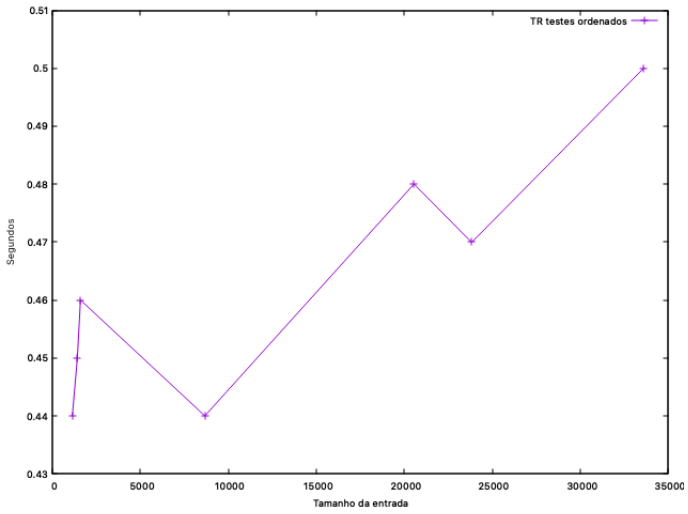

Figura 8: Conjunto de testes com palavras ordenadas, sem repetição

No terceiro conjunto de casos teste, não vemos nada de novo. O tempo de execução total aumenta apenas pelo fato de ter mais palavras na entrada.

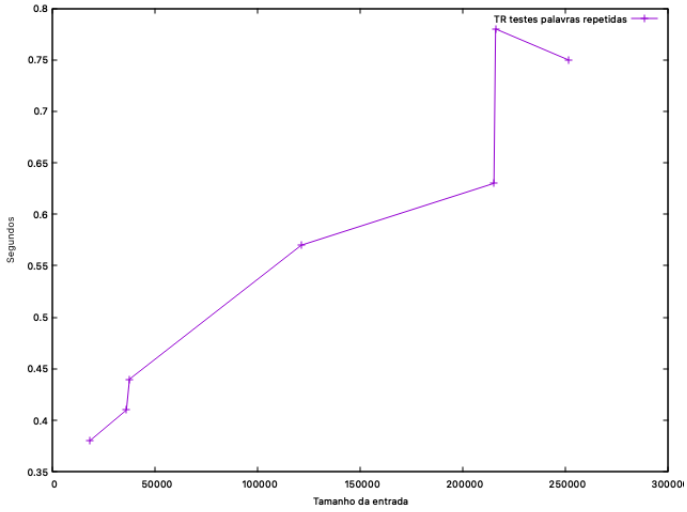

Figura 9: Conjunto de testes com palavras ordenadas aleatoriamente, com repetição

3.4 Árvore Rubro Negra

A árvore rubro negra tem uma complexidade garantida boa. Como a altura máxima não passa de $log2N$, todas as operações são executadas em $O(logN)$. Podemos observar que o tempo de execução do primeiro conjunto de testes é bem parecido com o das outras árvores de busca.

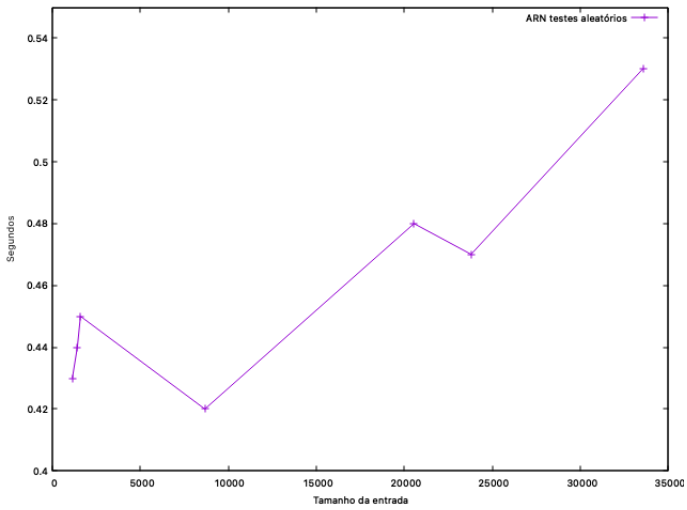

Figura 10: Conjunto de testes com palavras ordenadas aleatoriamente, sem repetição

Observe que no segundo conjunto de casos teste, com as palavras ordenadas, o tempo de execução não aumenta. Ou seja, não é atingido o pior caso desta estrutura com estes testes.

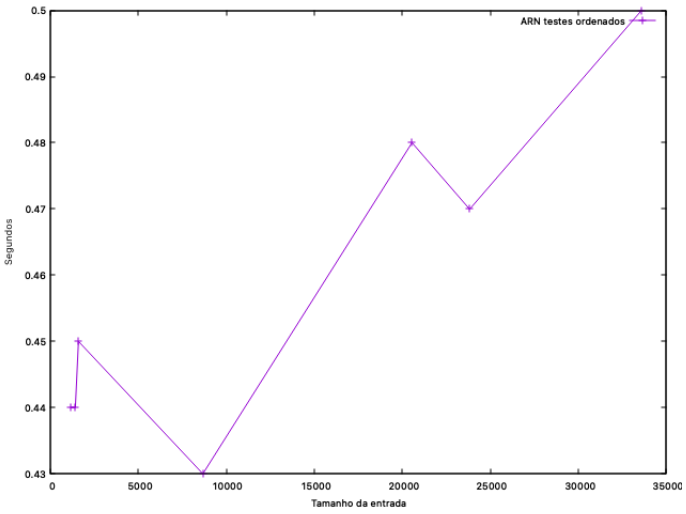

Figura 11: Conjunto de testes com palavras ordenadas, sem repetição

Já no conjunto de casos com entrada grande, vemos que a *ARN* funciona mais rápido que a Treap. Ou seja, operações de inserir nesta estrutura são um pouco mais rápidas na constante, provavelmente por causa da aleatoridade da Treap, já que a implementação é bem parecida.

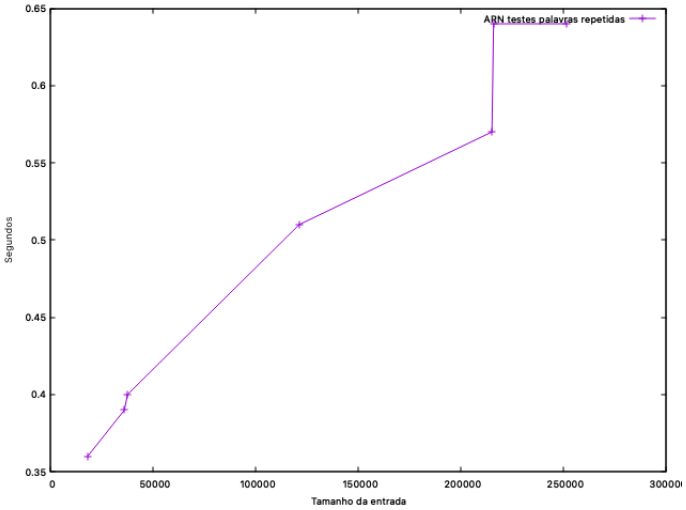

Figura 12: Conjunto de testes com palavras ordenadas aleatoriamente, com repetição

3.5 Árvore 2-3

A maior altura desta árvore é no máximo $logN$. Ou seja, o pior caso desta estrutura é aproximadamente o melhor caso da *ARN*. Como a implementação não tem uma constante tão ruim, conseguimos observar um desempenho um pouco melhor da $A23$.

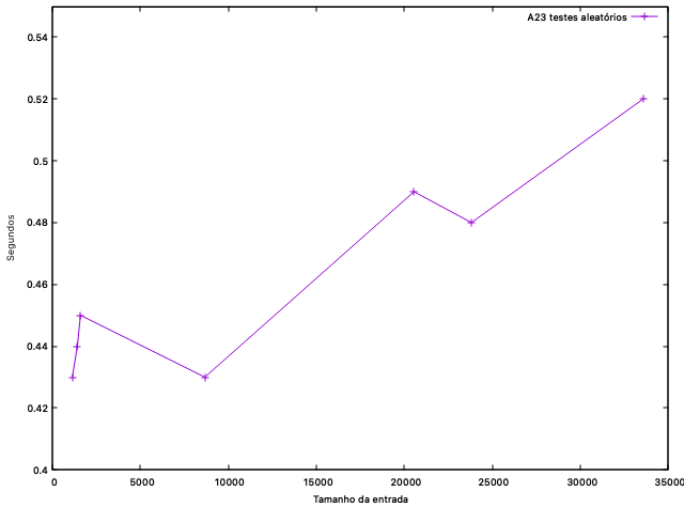

Figura 13: Conjunto de testes com palavras ordenadas aleatoriamente, sem repetição

No conjunto de testes com entrada ordenada, observamos que o tempo de execução quase não muda, o que é esperado.

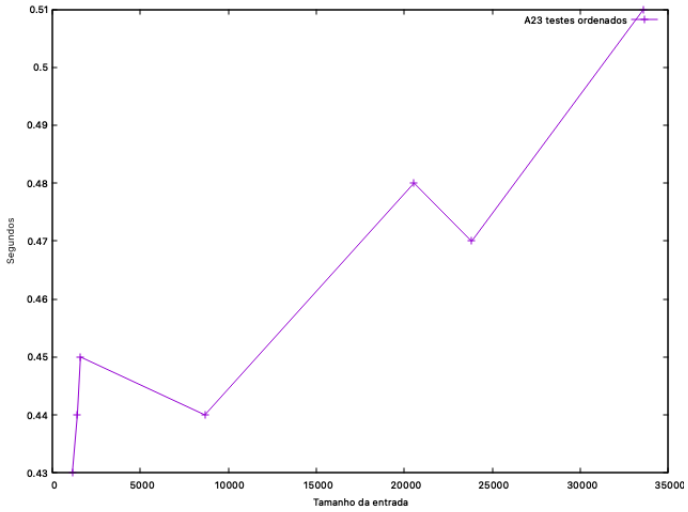

Figura 14: Conjunto de testes com palavras ordenadas, sem repetição

No último conjunto de testes, vemos que esta estrutura ficou um pouco pior do que a ARN, porém é uma diferença desconsiderável.

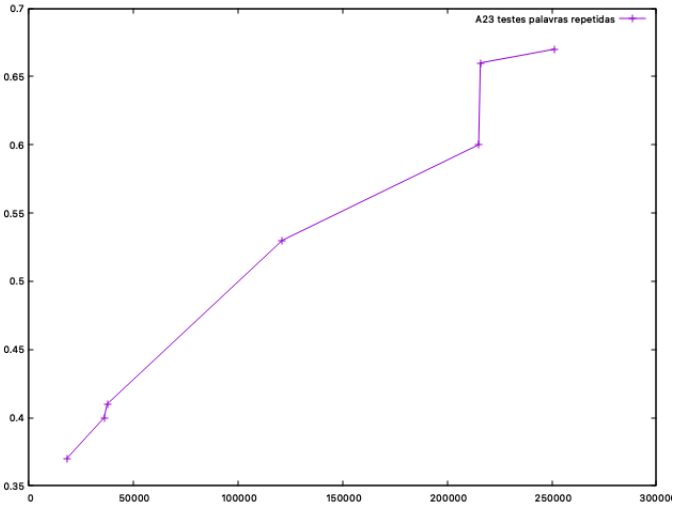

Figura 15: Conjunto de testes com palavras ordenadas, sem repetição

4 Conclusão e Feedback

Este EP, apesar de bastante trabalhoso, foi muito interessante. Foi desafiador implementar as árvores (principalmente $A23$ e ARN), e também deixar tudo organizado (fazer os casos, gráficos, etc). Apesar de ter achado particularmente divertido, a carga horária requerida é bem grande em relação a outros projetos. Talvez, se fosse dividido em etapas, seria melhor.