Documentação para o programa ”BookCounter”

Pedro Edson Maciel de Araújo

[1]Curso de Ciências da Computação – Universidade Federal do Ceará (UFC)
Caixa Postal 15.064 – CEP 63900-000 – Quixadá – CE – Brazil

Abstract. *This document describes the functionalities and architecture behind the program ”bookCounter” made in C++. Here it will also contain the hurdles and problems one’s made it through during the journey to fully implement the program.*

Resumo. *Esse documento descreve as funcionalidades e arquitetura por trás do programa ”bookCounter” feito em C++. Aqui também estará contida as dificuldades e problemas que o autor passou na jornada de criar de maneira completa o programa.*

1. Introdução

Uma das curiosidades que alguém pode ter durante a leitura de um livro ou qualquer longa extensão de texto é: ”quantas vezes essa palavra é usada?” Visando isso foi criado o programa ”BookCounter”, um programa feito em C++ que permite a contagem das palavras dentro de um documento de texto.

O programa foi criado para cumprir essa necessidade, com apenas um comando você consegue ler um documento de texto, escolher o tipo de estrutura a ser usada e escolher o documento de saída, a saída é um documento padronizado que retorna as palavras organizadas em ordem de frequência.

2. Implementação

O programa ”BookCounter” utiliza da leitura de strings e de quatro estruturas de dados para armazenar cada palavra que é lida pelo programa. As estruturas de dados disponíveis na classe são: Árvore AVL, Árvore Rubro-Negra, HashMap e ChainedHashMap.

A seguir vão ter os dois comandos de como utilizar o programa, é importante ressaltar que o programa deve ser compilado com o seguinte comando dentro da pasta do programa: g++ -std=c++17 -O2 -o bookcount bookCounter.cpp

2.1. Comando de Ajuda

Caso precise de ajuda com a formatação do programa ou quais estruturas de dados estão disponíveis para uso, o usuário pode convocar o comando ”- -help” ou apenas ”- -h” (sem espaços), esse comando retorna uma mensagem no terminal da formatação para usar o programa, além das quatro estruturas de dados que o usuário pode escolher.

2.2. Comando de execução

O comando principal do programa possui a seguinte estrutura: ./bookcount [nome do arquivo a ser lido] [sigla da estrutura] [destino (opcional)]. O nome ./bookcount pode ser substituído por qualquer outro nome que o usuário resolver dar à compilação do arquivo, mas por padrão, usaremos esse nome.

É de suma importância que o usuário tenha o arquivo de leitura e de destino dentro das pastas específicadas dentro do arquivo, senão o programa não será capaz de encontrar o arquivo de texto e retornará uma mensagem de erro.

As siglas de cada estrutura de dados está a seguir:

- avlTree - avl
- red-Black-Tree - rbt
- hashTable - ht
- chainedHashTable - cht

Por fim, o local de saída do comando é opcional, caso não seja colocado, o programa automaticamente jogará a saída dentro do arquivo de texto "resultado.txt"

2.3. Entrada e saída:

O programa recebe um documento de texto enviado por parâmetro na hora da inicialização e o tipo de estrutura a ser usada. Opcionalmente, ele também pode receber o arquivo de destino, se não, ele imprime em um arquivo padronizado dentro da pasta do programa.

Ao realizar o comando, o programa vai retornar uma mensagem de conclusão e imprimirá os dados formatados no arquivo padrão ou no arquivo enviado dentro do comando.

2.4. Estruturas de dados

O programa abriga quatro estruturas de dados independentes: uma árvore AVl, uma árvore Rubro-Negra, uma Tabela de Dispersão por encadeamento exterior e outra por endereçamento aberto.

Árvore AVL: A primeira estrutura feita do zero para esse projeto, consiste em uma árvore binária de busca que procura se manter o conceito de "balanceamento", um parâmetro que usa a fórmula $[Altura(Direita) - Altura(Esquerda)]$ para cada nó, verificando se essa fórmula está dentro do intervalo -1 $\leq x \leq$ 1, caso essa condição seja violada, a árvore realiza rotações simples ou compostas para retornar à um estado balanceado.

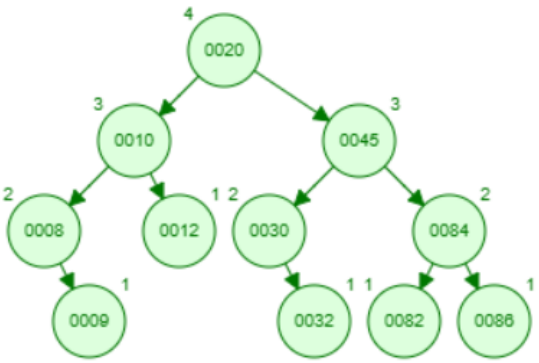

Figure 1. Exemplo de Árvore AVL

Árvore Rubro-Negra: Outra variação muito popular da árvore binária de busca é a Rubro-Negra, uma estrutura de dados que também procura manter o balanceamento da árvore para evitar degeneração, a resposta da árvore Rubro-Negra está no próprio nome, utilizando duas cores, preto e vermelho, para manter seu balanceamento. Durante as inserções e remoções, as cores dos nós são checadas para verificar se está ideal, sempre tentando minimizar a quantidade de nós da cor preta, pois esse é a referência usada para o balanceamento da árvore, chamada *altura negra* para essa tabela, eu decidi utilizar uma expressão constante para representar as cores da árvore, sendo chamadas de RED e BLACK.

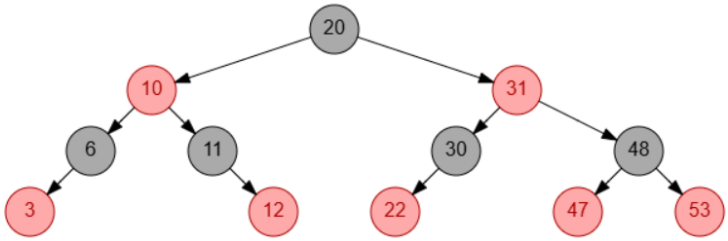

Figure 2. Exemplo de Árvore Rubro-Negra

Hash Table (Com endereçamento aberto): Hash Tables são estruturas que armazenam dados em o que pode ser visualizado como uma ”tabela”, são consideradas rápidas por conta de uma característica muito importante: elas possuem uma função determinística que codifica a chave do item e aloca em algum lugar baseado nesse endereço. Para manter suas operações eficientes, essa tabela de dispersão utiliza de ENUMS para representar estados e uma função de *probing* para navegar pela estrutura de forma que evite a navegação linear. Para esse trabalho, utilizei o *probing linear* que consiste em codificar a chave e incrementá-la com o índice para navegar os endereços da tabela em caso de colisão, caso o limite da tabela seja atingido, um *rehashing* acontece, dobrando o tamanho da tabela para o número primo mais próximo.

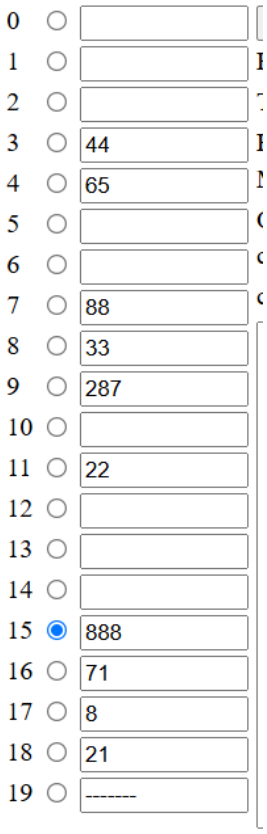

Figure 3. Tabela de Dispersão com endereçamento Aberto

ChainedHashTable (Encadeamento Exterior): Similar à HashTable, a ChainedHashTable ao invés de verificar o próximo endereço caso encontre um lugar ocupado, ela usa uma lista encadeada para conectar os itens, fazendo o vetor primário se tornar um ”vetor de endereços”, que direciona o item para o final de uma lista encadeada sempre que é inserido. Essas tabelas tentam manter um *fator de carga* α que é calculado utilizando a fórmula: $\frac{N}{M}$, N sendo a quantidade de itens dentro da tabela e M sendo o tamanho máximo da tabela. Idealmente, a tabela tenta manter um balanceamento de $\alpha \leq \frac{N}{M}$, isso evita que colisões desnecessárias aconteçam, a função de rehashing dessa tabela compartilha a mesma lógica da hashTable, dobrando o tamanho e encontrando o número primo mais próximo.

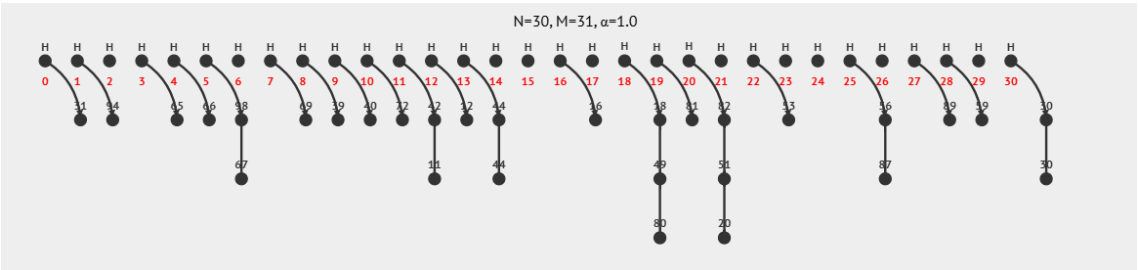

Figure 4. Tabela de Dispersão com encadeamento exterior

3. Testes e Resultados:

Pela falta de tempo, foram feitos dez testes robustos utilizando tipos de livros e textos de diferentes tipos e tamanhos.

Para os testes, utilizei um programa em Python para analisar os resultados e gerar um gráfico de acordo com os dados gerados pela execução em cada texto, os resultados podem ser visualizados a seguir:

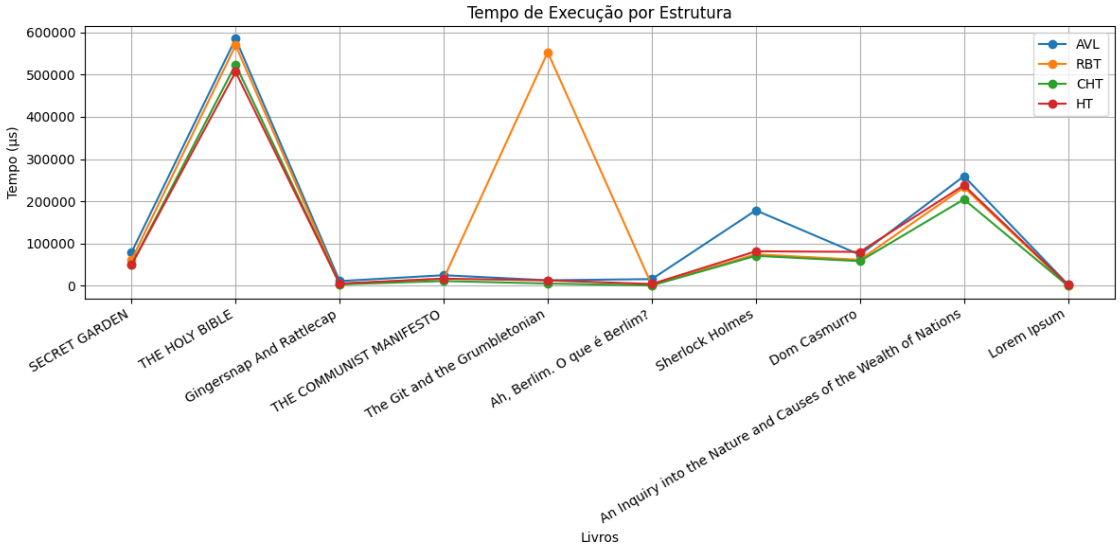

Figure 5. Velocidade de Execução de cada estrutura

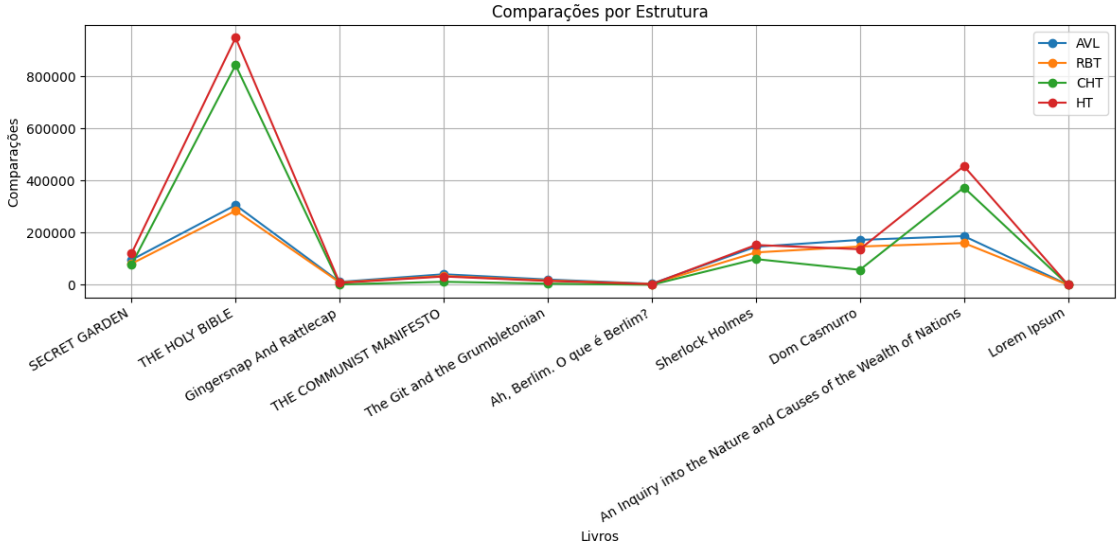

Figure 6. Quantidade de Comparações feitas por cada estrutura

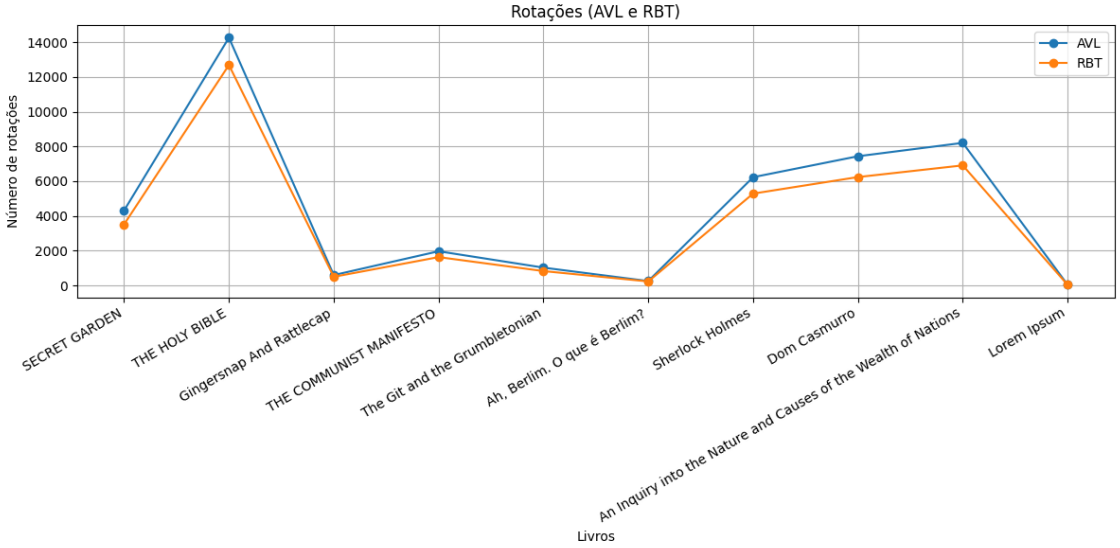

Figure 7. Quantidade de rotações realizadas pelas árvores

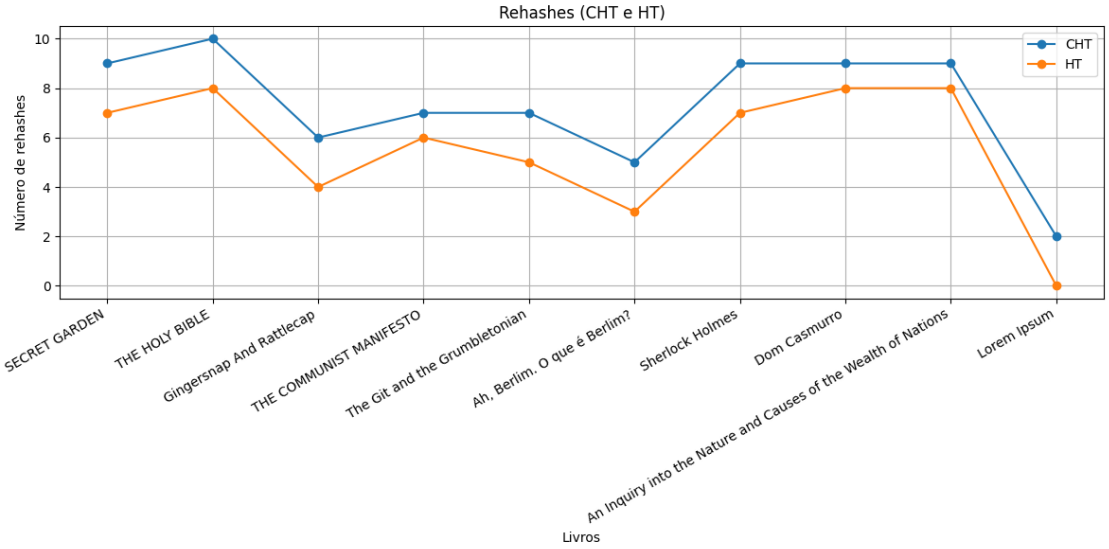

Figure 8. Quantidade de rehashes das tabelas

Em análise, é possível notar que a árvore AVL é a estrutura mais lenta entre todas, apesar de ser uma das que menos faz comparações, baseado também no gráfico de rotações, é possível presumir que isso acontece pela quantidade de rotações que a árvore costuma fazer, muito mais que uma rubro-negra, que faz apenas em casos específicos.

Após vermos a estrutura mais lenta, podemos ver que a mais rápida é a ChainedHashTable, com certeza isso se deve pela sua característica O(1) na inserção, precisando apenas converter a chave e jogar o item solicitado ali, sem se preocupar com colisões. Por outro lado, é possível notar que a ChainedHashTable faz muito mais rehashings do que a HashTable, apesar de isso possivelmente acontecer por uma característica de

implementação. É possível deduzir que a tabela de encadeamento exterior naturalmente consome mais memória por usar duas estruturas de dados na sua formação (Um vetor e uma lista encadeada).

Outro comportamento observado é o fato da HashTable ter ser a tabela que mais possui comparações, indicando que há muito mais colisões nela do que na sua semelhante, apesar de isso ser de se esperar.

4. Conclusão:

Esse trabalho, apesar de longo e extenuante, foi bastante divertido e me ensinou bastante na prática de como Estruturas de Dados funcionam. A maior dificuldade que passei foi fazendo a árvore Rubro-Negra, por possuir tantos casos de inserçao e deleção, sendo necessário que eu fizesse cada caso tanto para o lado esquerdo quanto para o lado direito, tudo de maneira iterativa (pois era o jeito que o Cormen fazia e eu preferi seguir os algoritimos do livro à risca). Tirando isso, eu já possuía uma árvore AVL implementada, então tive apenas que adaptar para que ela utilizasse dicionários.

Além disso, a implementação mais divertida que tive foi implementar a HashTable, por conta de ser uma estrutura razoavelmente simples e pude usar Enums para ajustar o código, assim como aprender sobre hashing linear, hashing quadrático, entre outros.

Minhas conclusões finais sobre esse projeto é que ele foi bastante divertido e super útil. Pretendo postar ele em meu gitHub após o fim do semestre para que outras pessoas possam usar.

5. Bibliografia:

Visualização da Hashtable:
https://visualgo.net/en/hashtable

Visualização da ChainedHashTable:
https://www.ime.usp.br/ pf/estruturas-de-dados/aulas/st-hash.html

Site explicando sobre HashTables:
https://www.cs.tufts.edu/ ablumer/openhashing.html

Site oficial do C++ explicando sobre a biblioteca Chrono:
https://cplusplus.com/reference/chrono

Visualização da Árvore AVL:
https://www.cs.usfca.edu/ galles/visualization/AVLtree.html

Visualização da Árvore Rubro-Negra:
https://ds2-iiith.vlabs.ac.in/exp/red-black-tree/red-black-tree-oprations/simulation/redblack.html

Livro ”Introduction to Algorithms — Thomas H. Cormen” 3rd Edition.