UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
INSTITUTO METRÓPOLE DIGITAL

André Freitas Silveira

João Guilherme Lopes Alves da Costa

Jonas de Oliveira Freire Filho

Análise Assintótica do Contêiner Árvore Binária de Busca

NATAL, RN
2022

André Freitas Silveira

João Guilherme Lopes Alves da Costa

Jonas de Oliveira Freire Filho

Análise Assintótica do Contêiner Árvore Binária de Busca

Relatório técnico apresentado a disciplina Estrutura de Dados Básica II para obtenção de nota parcial da segunda unidade do semestre letivo 2021.2 do curso Bacharelado em Tecnologia da Informação pela Universidade Federal do Rio Grande do Norte.

Orientadora: Profª. Dra. Silvia Maria Diniz Monteiro Maia

NATAL, RN

2022

Sumário

1 INTRODUÇÃO

Este presente relatório tem como objetivo explicitar a abordagem adotada para implementar uma Árvore Binária de Busca e a análise de complexidade assintótica de seus métodos. Esse trabalho foi produzido durante o período 2021.2 para a disciplina de Estrutura de Dados Básicas II pela Universidade Federal do Rio Grande do Norte (UFRN), aplicando os conceitos aprendidos em sala para implementar a estrutura de dados.

2 METODOLOGIA

Essa seção tem como objetivo explicar todo o processo de criação do contêiner $bst :: BinarySearchTree$ que é uma implementação da estrutura de dados "Árvore Binária de Busca" capaz de armazenar chaves e dados. Contendo os seguintes métodos: *insert*, *search*, *remove*, *elementInPosition*, *findPositionOfElement*, *median*, *isFull*, *isComplete* e *toString*. A implementação ocorreu na linguagem de programação $C++$, a qual possibilitou a generalização do contêiner para suportar qualquer estrutura de dados, usando o $template$, desde que a estruturara de dados das chaves (*keyType*) seja comparável.

Primeiramente, foi criado o *struct* base do Nó, chamado de "*Node**". Esse *struct* contém os atributos "*Data*" (Informação a ser armazenada no contêiner), "*Key*" (Chave única), "*Node* Left*" (Ponteiro que aponta para a esquerda do nó) e "*Node* Right*" (Ponteiro que aponta para a direita do nó). Em sala de aula a docente a fim de explicação considerou "*Key*"igual a "Data". Além disso, para facilitar a melhorar a performance dos métodos, foram criados também outros três atributos. "*Node* raw_pointer*" (ponteiro que aponta para o nó raíz), "*height*" (um inteiro que corresponde a altura da árvore) e o "*number_of_nodes*" (outro inteiro para armazenar a quantidade de nós na árvore).

Em seguida, foi a vez de implementar os métodos modificadores da árvore. São eles: *insert*, *remove* e *search*. O método *insert* é responsável por inserir um novo elemento na árvore, sendo checado antes se o elemento já existe. Em seguida, o método *remove* foi feito para remover um determinado nó. Caso o elemento passado como parâmetro não exista, sua remoção é ignorada. Por fim, o método *search*, que busca um dado na árvore passando por todos os nós.

Logo em seguida foi a vez de implementar os métodos de acesso da árvore binária. Nessa parte, não só foram feitos os métodos requisitados, mas também métodos auxiliares, para ajudar o grupo a resolver determinado problema enfrentado durante o desenvolvimento dos métodos obrigatórios do trabalho. Os métodos de acesso são: *elementInPosition* (recebe uma posição e retorna o elemento que ocupa essa posição), *findPositionOfElement* (recebe um elemento e retorna a posição que esse elemento ocupa), *median* (retorna a mediana da árvore), *isComplete* (retorna verdadeiro se a árvore for completa, e falso caso o contrário), *isFull* (retorna verdadeiro se a árvore for cheia, e falso caso o contrário), *toString* (retorna uma string que contém a sequência de visitação por nível), *simetric* (um método adicional para percorrer a árvore em ordem simétrica) e finalmente o *simetricToElement* (Apenas uma variação do método anterior para auxiliar o método *elementInPosition*).

3 ANÁLISE ASSINTÓTICA

Esse tópico é responsável por demonstrar brevemente as análises assintóticas de cada método implementado.

Os métodos inserção e remoção foram implementados de forma independente do algoritmo de busca, como requisitado pela docente, de modo que os algoritmos sejam o mais eficiente possível.

3.1 Inserção

Como pode ser observado no repositório do trabalho, o método de inserção tem complexidade $O(h)$, sendo h a altura da árvore. Uma vez que o método *insert* percorre toda a árvore em nível, até encontrar a posição do elemento a ser inserido, e esse percurso teria no máximo h passos.

Note que no método *insert*, é chamado o método privado *get_height*, responsável por calcular a altura atual da árvore. Como esse método tem complexidade $O(h)$, teríamos para a inserção $O(h + h)$ que é $O(h)$.

3.2 Remoção

Assim como no método de inserção, o método de remoção possui complexidade $O(h)$, sendo h a altura da árvore. Isso ocorre pois o *remove* percorre toda a árvore em nível, até encontrar o elemento a ser removido.

Ademais, o método de remoção recalcula a altura da árvore. E assim como na inserção, a complexidade assintótica não é afetada.

3.3 Busca

Analogamente ao método de inserção e de remoção, o método de busca tem complexidade $O(h)$ pelas mesmas justificativas, com exceção que neste método não verifica-se a altura.

3.4 Enésimo Elemento

O método Enésimo Elemento usufrui de outro algoritmo recursivo chamado *simetricToElement*, que percorre a árvore binária de busca em ordem simétrica. Dessa forma, como *simetricToElement* percorre a árvore por nível até encontrar a posição desejada. Por conta disso, a complexidade do *elementInPosition* é $O(n)$.

3.5 Posição

De maneira análoga ao método "Enésimo Elemento", o método "Posição" necessita de um algoritmo auxiliar chamado de *simetric*. Este algoritmo é similar ao *simetricToElement*, e, portanto tem complexidade: $O(n)$. Por causa disso, o método *findPositionOfElement* também tem complexidade $O(n)$.

3.6 Mediana

O método mediana é de complexidade $O(n)$, sendo n o número de elementos da árvore. Já que para buscar o elemento que se encontra na mediana da árvore foi necessário percorrer todos os elementos da árvore por ordem simétrica e armazená-los em um vetor.

3.7 É Cheia

Como pode ser observado no repositório do trabalho, o método $isFull$ tem complexidade $O(1)$.

3.8 É Completa

Nesse método teremos um for que vai iterar no máximo $h-1$ vezes, e em cada iteração teremos a chamada do método *nodeOnLevel*, que fará um percorrimento por nível, e no último nível teremos o percorrimento de todos os níveis, sendo realizado n passos.

Dessa forma, a complexidade do método será $O\Big(n * (h-1)\Big)$. Logo, $O(nh)$.

3.9 To String

Diferente dos outros métodos, o "To String" possui duas formas de ser chamado: por nível e simétrico. O método de impressão simétrico percorre, assim como em outros métodos, toda a árvore recursivamente, chamando todos os filhos, e, por isso, tem complexidade $O(n)$.

Quanto ao método de impressão por nível, foi implementado de forma não recursiva. Entretanto, este também possui complexidade de $O(n)$, pois sua implementação consiste em adicionar os elementos em uma fila e ir consumindo esses elementos até que todos os elementos sejam impressos. Desse modo, ambas implementações têm complexidade $O(n)$.

4 CONCLUSÃO

A implementação deste contêiner pela equipe possibilitou um melhor entendimento das estrutura dados estudadas em sala de aula e um maior refinamento quanto as complexidades que cada método. Este trabalho foi um grande desafio, foi necessário estudar intensamente os algoritmos e conceitos vistos em sala de aula, o que contribuiu demasiadamente para compreender melhor a estrutura e as operações da árvore binária de busca.