UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
INSTITUTO METRÓPOLE DIGITAL (IMD)
BACHARELADO EM TECNOLOGIA DA INFORMAÇÃO
DISCIPLINA: ESTRUTURAS DE DADOS BÁSICAS II

Relatório de implementação de uma árvore binária

Bianca Antonelly Medeiros Dos Santos
Vanessa Maria De Oliveira Silva

Natal/RN - 14/11/2023

Sumário:

1 Introdução

A árvore binária surge no campo de estrutura de dados como uma alternativa para solucionar os problemas vigentes nas demais estruturas, e tem como proposta quebrar um problema maior em pequenas partes, com a finalidade de manter um código eficaz e com boa complexidade. A principal característica dessa estrutura é o Nó raiz e os seus nos filhos, no esquerdo e no direito. Ademais, o no também guarda o valor armazenado, assim como seus filhos.

O presente relatório tem como objetivo analisar a implementação de uma árvore binária de busca. Cujo tem os seguintes métodos:

1.1 int enesimoElemento (int n):

Retorna o n-ésimo elemento (contando a partir de 1) do percurso em ordem (ordem simétrica) da ABB.

1.2 int posicao (int x):

Retorna a posição ocupada pelo elemento x em um percurso em ordem simétrica na ABB (contando a partir de 1).

1.3 int mediana ():

Retorna o elemento que contém a mediana da ABB. Se a ABB possuir um número par de elementos, retorne o menor dentre os dois elementos medianos.

1.4 double média (int x):

Retorna a média aritmética dos nós da árvore que x é a raiz.

1.5 boolean ehCheia ():

Retorna verdadeiro se a ABB for uma árvore binária cheia e falso, caso contrário.

1.6 boolean ehCompleta ():

Retorna verdadeiro se a ABB for uma árvore binária completa.

1.7 String pre_ordem ():

Retorna uma String que contém a sequência de visitação (percorrimento) da ABB em pré-ordem.

1.8 void imprimeArvore (int s):

Se “s” igual a 1, o método imprime a árvore no formato 1, “s” igual a 2, imprime no formato 2.

Além disso, é implementado os métodos inserir, buscar e remover. Dessa forma, vamos analisar a lógica pensada para cada método, suas especificações e sua complexidade.

2 Metodologia

2.1 Classe No:

A classe No é a encarnação de uma árvore binária, apresentando os atributos essenciais: um inteiro valor, além dos nós à esquerda (No noEsquerdo) e à direita (No noDireito). Com isso, ela se torna um bloco fundamental para a construção e manipulação de estruturas de árvores binárias.

2.2 Classe ArvoreABB:

A classe ArvoreABB abriga a implementação das funcionalidades cruciais para operações em árvores binárias. O ponto focal é o objeto No raiz, sobre o qual todas as operações são executadas.

As operações incluem inserção, remoção, busca e outros procedimentos característicos de árvores binárias de busca. A classe se destaca pela eficácia na manipulação e organização de dados, aproveitando as propriedades estruturais das árvores binárias.

2.3 Classe Main:

A classe Main desempenha um papel central na execução do programa, assumindo a responsabilidade de ler arquivos e transmitir as informações pertinentes para o nó raiz da classe ArvoreBB. Esse processo é crucial para habilitar a execução eficiente das funcionalidades disponibilizadas pela classe ArvoreBB.

Ao coordenar a comunicação entre a entrada de dados e a manipulação das estruturas de árvore, a classe Main desempenha um papel vital na orquestração fluida do programa, proporcionando uma interface facilitadora para o usuário interagir com as funcionalidades disponíveis na árvore binária.

3 Funções Implementadas:

3.1 Função enesimoElemento(int n):

Esta função retorna o enésimo elemento de uma Árvore Binária de Busca (ABB). No melhor caso a complexidade de melhor caso é O(1) e no pior caso a complexidade é a O(h), onde h é o tamanho da altura.

A lógica por trás disso é a seguinte: Se a quantidade de nós à esquerda do nó atual mais um for igual à posição "n", retornamos o valor desse nó. Se a quantidade for maior do que a posição "n", chamamos o filho à esquerda do nó. Se for menor, chamamos o filho à direita e subtraímos a quantidade mais um, deixando apenas a quantidade de elementos restantes até a posição desejada.

3.2 Função mediana():

Esta função retorna o elemento mediano de uma Árvore Binária de Busca (ABB). Sua complexidade é determinada principalmente pela chamada à função enesimoElemento, na qual é passado como parâmetro o índice da mediana e que retorna o elemento mediano. Com isso, a complexidade no melhor caso é O(1) e no pior caso é O(h) com h sendo a altura da árvore.

Quando a árvore contém um número par de elementos, a função seleciona o menor valor dos dois elementos centrais como a mediana.

3.3 Função media(int x):

Essa função retorna a média aritmética dos elementos da árvore que têm um valor "x" como raiz. Primeiramente, realiza-se uma busca pelo nó com o valor "x". Esta função possui complexidade O(1) no melhor caso e O(h) no pior caso. Esse algoritmo é utilizado para percorrer os elementos da árvore e somar seus valores.

3.4 Função ehCheia()

A função ehCheia() é responsável por verificar se uma árvore binária é "cheia" ou não. Uma árvore é considerada "cheia" quando todos os nós da árvore têm zero ou dois filhos não nulos. A implementação da função percorre todos os nós da árvore de forma recursiva para fazer essas verificações.

A função inicia sua verificação pelo nó raiz da árvore. Se o nó raiz for nulo (ou seja, a árvore estiver vazia), ela imediatamente considera a árvore como "cheia" e retorna esse resultado.

Em seguida, a função verifica se o nó raiz não tem filhos (nenhum filho esquerdo e nenhum filho direito). Se esta condição for atendida, a árvore é novamente considerada "cheia," e a função retorna esse resultado. A complexidade do método é constante em todos os casos, ou seja O(1).

3.5 Função imprimirBarras(No raiz, int tracos, int blank)

A função imprimirBarras tem como objetivo apresentar visualmente uma Árvore Binária de Busca (ABB) no formato "formato 1", conforme especificado nos requisitos do projeto. Aqui está uma descrição detalhada da função:

A função desempenha um papel fundamental na representação gráfica de uma árvore binária no console, usando barras (-) para indicar as conexões entre os nós e espaços em branco para a formatação adequada, possui complexidade O(n).

3.6 Função imprimirParenteses(No raiz)

A função imprimirParenteses desempenha um papel crucial na apresentação estruturada de uma Árvore Binária de Busca (ABB) no formato "formato 2", conforme especificado nos requisitos do projeto. Aqui está uma descrição detalhada da função:

A função tem como objetivo criar uma representação textual da árvore no formato de parênteses aninhados, facilitando a visualização hierárquica da estrutura da árvore, possui complexidade O(n).

3.7 Função posicao(int x)

A função posição desempenha um papel crucial ao determinar a posição de um elemento na Árvore Binária de Busca (ABB) a partir de um valor recebido como parâmetro. Sua complexidade é O(n) no pior caso, quando a árvore assume uma forma linear, e O (logn) em cenários ideais, onde a árvore não é linear.

A principal finalidade desta função é fornecer a posição de um elemento na árvore, baseando-se no valor x fornecido como argumento. Ao realizar esse mapeamento, a função destaca-se por sua eficiência em cenários de árvores não lineares, onde a busca pela posição do elemento é otimizada.

3.8 Função ehCompleta(No no):

Essa função tem como responsabilidade verificar se a árvore binária é completa. Por definição, uma árvore binária é dita completa “Se v é um nó com uma subárvore vazia, então v está no último ou no penúltimo nível de T”. A ideia principal dessa função é descer a ávore recursivamente e comparar a altura do no quando for folha e verificar se a sua altura é menor que a altura da árvore (raiz) - 1. Para verificar isso, é utilizado uma função auxiliar que tem como parâmetro um nó, e possui dois ifs que verificam respectivamente se no.getNoEsquerdo() != null e se no.getNoDireito() != null, se for diferente de null, continua chamando a função auxiliar passando o elemento filho de acordo com o respectivo lado do nó analisado no condicional. Enquanto isso, é necessário verificar se a altura do noAtual é menor que a altura da raiz (árvore) - 1. A complexidade da função principal é O(n) e da função auxiliar também é O(n), logo, a complexidade é O(n^2).

3.9 Função pre_ordem(No no):

A função pré ordem tem como finalidade percorrer a árvore binária de busca na seguinte ordem: visita raiz, depois percorre a subárvore à esquerda, por último, percorre subárvore à direita. O algoritmo vai passar por cada nó. A complexidade dessa função é O(n), onde n é o número de nós que possui a árvore.

3.10 Função contNos(No no):

Método responsável por atualizar a quantidade de nós a esquerda e direita do nó passado como parâmetro. A complexidade desse algoritmo é O(n) no pior caso e O(n.log (n)) no melhor caso.

3.11 Função inserirNo(No noATual, No no):

Esse método permite inserir nos no nó raiz, ou seja, inserir nos na árvore binária. Para isso, analisamos algumas verificações:

- Se o nó já existe na árvore, se sim, não podemos inserir
- Se o valor do nó que desejo inserir é menor que o valor do nó raiz
- Se o valor do nó que desejo inserir é maior que o valor do nó raiz

Por fim, atualizamos a quantidade de nós do noAtual, a cada finalização do método. A complexidade desse algoritmo é O(h), onde h representa a altura da árvore.

3.12 Função buscar(No no, int valor):

A função buscar tem como finalidade buscar um nó dentro da árvore binária e retornar se esse elemento foi encontrado. O racicíonio aplicado ao buscar é verificar se o valor passado como parâmetro é igual a no.getValor(), se não verificar se esse valor é menor que valor da raiz, ou se o valor é maior que a raiz. A partir disso, é chamado recursivamente a função até que o valor seja encontrado, caso isso não aconteça, novas condições devem ser seguidas. A complexidade dessa funcionalidade no seu pior caso é O(h), onde h corresponde a altura da árvore.

3.13 Função removerNo(No noATual, No no):

O método remover tem a finalidade de remover um nó da árvore binária. Para isso, é preciso analisar alguns casos como:

- Se o nó é uma folha
- Se o nó possui um filho
- Se o nó possui dois filhos

A ideia geral do algortimo é realizar comparações na subárvore à esquerda e direita, à procura do elemento a ser removido, quando encontrar realizar a operação de remover o nó e atualizar a árvore caso seja necessário. Por fim, é utilizado um método auxiliar para verificar se foi possível remover o nó. A complexidade desse algoritmo é O(h), onde h representa a altura da árvore.

3.14 Função noMin(No no):

Essa função retorna o nó mínimo da subarvore a esquerda. Enquanto, o nó à esquerda for diferente de vazio o while vai ser executado e o no passado como parâmetro recebe o no.getNoEsquerdo(). Quando a condição for falsa vai sair do while e vai ser retornado o nó mínimo da árvore binária. A complexidade é O(log n) no melhor caso e de O(n) no pior caso.

3.15 Função altura(No no):

O método altura retorna a altura do nó passado como parâmetro. A ideia geral do algoritmo é realizar chamadas recursivas e acumalar a altura a direita e altura a esquerda, depois realizar comparações a fim de verificar qual a maior altura e retornar. A complexidade é O(n), onde n corresponde ao número de nós da árvore.

4 Resultados

A inserção eficiente em uma árvore binária de busca é crucial para manter a estrutura da árvore organizada e otimizada. O código fornecido apresenta um método inserirNo que realiza essa tarefa de maneira recursiva, garantindo que o novo nó seja inserido no local apropriado da árvore.

A função inserirNo recebe dois parâmetros: noNovo, representando o nó a ser inserido, e noAtual, representando o nó atual na recursão. O algoritmo compara os valores dos nós para determinar se o nó deve ser inserido à esquerda ou à direita do nó atual.

Ao final da inserção, a função contNos é chamada para atualizar a contagem de nós em cada nó, levando em consideração os nós à esquerda e à direita. Isso é crucial para otimizar operações futuras na árvore, como a busca por determinado valor. Além, da remoção de um elemento que também é realizada de forma eficiente.

Concluímos que a partir da implementação da árvore binária de busca conseguimos averiguar a eficiência e notar sua importância no campo de estruturas de dados.

Na captura de tela do console do nosso projeto, observamos a impressão dos resultados conforme esperado. Os dados exibidos coincidem perfeitamente com o conteúdo do arquivo de saída base, refletindo a correta execução do nosso projeto.

```
A árvore não é cheia
A árvore é completa!
20
36 adicionado
A árvore é cheia
 32 13 5 20 41 36 60
32-------------------------
    13--------------------
         5-----------------
        20-----------------
    41--------------------
        36-----------------
        60-----------------
 (32 (13 (5) (20)) (41 (36) (60)))
50 não está na árvore, não pode ser removido
15 adicionado
39 adicionado
32 removido
3
39 já está na árvore, não pode ser inserido
36
20
28.625
Chave encontrada
25 adicionado
25
```

5 Conclusão

Ao concluir este projeto de implementação da árvore binária aumentada, observamos que a abordagem adotada proporcionou resultados sólidos em termos de eficiência e funcionalidade. Enfrentamos desafios intrigantes, como lidar com casos especiais durante inserções e remoções, e a implementação de estratégias para otimizar o desempenho em cenários específicos.

Este projeto não apenas consolidou nosso conhecimento em estruturas de dados, mas também proporcionou insights valiosos sobre a importância de escolher abordagens eficientes na resolução de problemas computacionais. Agradeço a oportunidade de realizar este trabalho e consolidar meu aprendizado nesta disciplina.