UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE

BACHARELADO EM TECNOLOGIA DA INFORMAÇÃO

Deyvid William Silva de Medeiros

Guilherme Euller Dantas Silva

Nathãn Vitor de Lima

Árvore de Busca Binária

NATAL

2022

Buscar

Função recursiva que sai verificando os valores dos nós para saber se é igual ao valor informado e retorna o nó caso o encontre. Verificação simples, caso o valor seja menor do que o valor do nó atual, ele chama a própria função passando o filho à esquerda e o valor que deseja ser buscado. Caso o valor seja maior, ele verifica para fazer a recursão com o filho à direita. Depois disso tudo, retorna o nó.

Complexidade: Θ(log n), pois a cada nó que é visitado, metade dos nós filhos não são visitados.

Inserir

Chama a função `inserirSemAtualizar()`, e depois chama a função `atualizaNumeroNos()`, e por fim retorna o valor obtido na primeira função.

Complexidade: Θ(n), pois a cada nó que é visitado, metade dos nós filhos não são visitados.

Remover

Chama a função `removerSemAtualizar()`, e depois chama a função `atualizaNumeroNos()`, e por fim retorna o valor obtido na primeira função.

Complexidade: Θ(n), pois a cada nó que é visitado, metade dos nós filhos não são visitados.

Enésimo Elemento

O método `enesimoElemento()` retorna o valor do índice informado, e para isso os campos que informam o número de nós à esquerda e à direita do nó são utilizados.

Se o nó informado for nulo, significa que por meio da recursão anterior foi informado uma subárvore vazia, e portanto o nó com índice informado não existe.

Caso contrário, se o índice for igual ao número de nós à esquerda mais um, significa que o nó atual possui o índice buscado, e o nó é retornado.

Caso contrário, se for menor que o número de nós à esquerda mais um, significa que o nó desejado está à esquerda do nó atual, e é retornado o enésimo

elemento de forma recursiva usando como parâmetros o nó à esquerda do nó atual e o mesmo índice.

Caso contrário, se o nó buscado não é e nem está à esquerda do nó atual, então possivelmente está à direita, e é retornado o enésimo elemento de forma recursiva usando como parâmetros o nó à direita do nó atual e removendo do índice desejado o número de nós à esquerda do nó atual mais um.

Complexidade: Θ(log n), pois a cada nó que é visitado, metade dos nós filhos não são visitados.

Posição

O método `posicao()` retorna a posição do elemento informado. Para isto é verificado se o valor informado é menor ou maior que o da raiz, caso seja menor ele vai chamar recursivamente a função para o lado esquerdo de forma que seja obtido o índice desejado, caso seja maior ele chama recursivamente a função para o lado direito para, assim como na condição anterior, conseguir o índice do valor, contudo há o acréscimo da soma de todos os índices do lado esquerdo mais um, referente à raiz, caso nenhuma das condições sejam atendidas, significa que o valor é a própria raiz, por tanto ele só retorna a soma de todos os valores da esquerda mais um.

Complexidade: Θ(log n), pois a cada nó que é visitado, metade dos nós filhos não são visitados.

Mediana

O método `mediana()` retorna a mediana de todos os nós da árvore.

É obtido o tamanho da árvore usando os campos do número de nós à esquerda e à direita da árvore e adicionado um. Em seguida, é verificado se esse número é par ou ímpar para obter o valor exatamente do meio ou o menor dos dois no meio se for par através do método `enesimoElemento()`.

Complexidade: Θ(log n), pois a complexidade utiliza em sua maior parte o método `enesimoElemento()`.

Soma

O método `soma()` executa de forma recursiva, somando o valor do nó raiz mais a soma dos nós à direita mais a soma dos nós à esquerda, caso o nó raiz seja diferente de nulo.

Complexidade: Θ(n), pois visita todos os nós da árvore para somá-los.

Média

O método `media()` inicia-se com uma busca recursiva para o nó com o valor informado. Após isso é utilizado o método de soma e esse valor obtido é dividido pelo tamanho da árvore (com os campos que informam o número de nós à esquerda e à direita do nó atual mais um).

Complexidade: Θ(n), utilização da função soma e acesso a todos os nós abaixo da raiz (nó informado).

É cheia

O método `ehCheia()` retorna de forma booleana se a árvore é cheia ou não. Primeiramente é verificado se o nó é uma folha, e se este está no último nível da árvore, pois caso contrário já é retornado false. Caso passe nessa validação verificamos se possui filho à esquerda e não possui filho à direita, ou o caso inverso, pois desta forma também não seria uma árvore cheia. Ainda assim, caso não tenha entrado em nenhuma dessas condições, a função verifica se o filho à esquerda é nulo e de forma recursiva chama a função novamente passando o filho à esquerda para verificar se não encontra nenhuma condição que invalide ele prossegue para a mesma verificação só que para o lado direito, e caso ele consiga passar por todas essas validações ele retorna true, ou seja, a árvore será cheia.

Complexidade: Θ(n).

É completa

O método `ehCompleta()` vai retornar se a árvore é completa ou não por meio de um booleano. Para isso, a função recebe o nó raiz para usar como base e começa as verificações. Primeiro é verificado se o nó tem uma subárvore vazia, caso

isso ocorra, ele pega o último nível da árvore, pega o nível atual desse nó e verifica se o nível do nó é diferente do nível máximo ou do nível máximo menos um, de forma que se um desses casos acontece é retornado false. Depois disso a função realiza mais duas verificações, caso o filho da esquerda do nó seja diferente de nulo a função é chamada recursivamente para este; caso o filho da direita seja diferente de nulo a função é chamada recursivamente para este. Se nenhuma das validações retornar falso, então significa que a árvore é completa e é retornado verdadeiro.

Complexidade: Θ(n), já que passa por todos os nós da árvore;

Pré-ordem

O método `pre_ordem()` retorna uma string com a pré-ordem da árvore. Utilizando o que vimos em aula, basicamente é concatenado à string de resposta o valor do nó e depois a função é chamada recursivamente para o filho da esquerda, caso diferente de nulo, e depois é chamada recursivamente para o filho da direita, caso este seja diferente de nulo, também, dessa forma conseguimos obter uma string contendo a pré-ordem da árvore então é retornada a string.

Complexidade: Θ(n) pois visita todos os nós

Imprime Árvore

O método `imprimeArvore()` é um seletor de como será retornada a árvore. É passado um parâmetro contendo o valor referente à qual árvore deve ser exibida, depois disso ele verifica se o valor é igual à 1 ou 2, se for 1, retorna o formato 1 senão retorna o formato 2 (ambos explicados abaixo).

Complexidade: Θ(n), pois visita todos os nós.

Formato 1

O método pega o nível do nó, coloca as tabs (“\t”) necessárias para cada nível, imprime o valor e depois faz um looping para colocar os “-----” depois chama a função de forma recursiva para colocar os seus filhos à esquerda e à direita caso eles sejam diferentes de nulo.

Complexidade: Θ(n), pois visita todos os nós.

Formato 2

O método cria uma string inicial com “(“ mais o valor do nó. Se houver um nó à esquerda, adiciona à string um espaço e a função de formato 2 recursivamente com o nó à esquerda do nó atual, e de forma análoga com o nó à direita do nó atual, e ao final adiciona um “)” à string e retorna a string.

Complexidade: Θ(n), pois visita todos os nós.

Inserir Sem Atualizar

O método `inserirSemAtualizar()` insere na árvore o valor informado sem atualizar as informações de quantidade de nós à direita e ou esquerda (essas informações são atualizadas na função `atualizaNumeroNos()`), explicada mais abaixo. O processo de inserção é muito simples, onde o valor é verificado se é menor ou maior que o nó atual, e caso seja ele vai chamando a função recursivamente passando os filhos da direita ou esquerda até encontrar a sua posição correta para ser adicionado, caso não seja possível adicionar é retornado false.

Complexidade: Θ(log n) pois ao inserir o nó, metade dos nós são removidos ao descer um nível da árvore.

Buscar nó Pai

O método `buscarNoPai()` irá buscar o nó pai do nó cujo valor é informado. Basicamente a função busca a partir do nó raiz, por meio de recursividade, qual nó possui um filho cujo o valor é igual ao informado, quando encontrado ele retorna o nó pai. Funciona como um método de busca normal, o seu diferencial é o fato do ponto de parada ser o filho do nó atual ser igual ao valor informado no parâmetro da função.

Complexidade: Θ(log n) pois ao buscar o nó pai, metade dos nós são removidos ao descer um nível da árvore.

Remover sem atualizar

O método `removerSemAtualizar()` vai remover o elemento, sem atualizar o valor da propriedade de nós à direita e nós à esquerda (novamente, atualizados pela função abaixo - `atualizaNumeroNos()`). O processo de remoção se dá da seguinte maneira: primeiro pegamos o nó pai do nó que queremos remover, caso o valor seja nulo nada acontece, caso contrário, **verificamos se o filho à esquerda desse pai é diferente de nulo e se o valor desse filho é igual ao valor que queremos remover da árvore (I)**, caso seja verdadeiro fazemos outra validação que é para saber qual filho deverá "subir", caso haja, portanto, verificamos se o filho à esquerda do filho do pai atual é nulo, caso não seja, "subimos" o valor adicionando ele no lugar do filho do pai, caso seja nulo fazemos essa validação para o filho da direita. Caso **(I)** seja verificado como falso então verificamos se o filho à esquerda do filho da direita é nulo, pois caso seja, o filho da direita vai receber o seu filho, caso ainda seja inválido, fazemos a mesma validação para o filho da direita do filho da direita (seria algo como *pai.filho.filho*), de forma que o filho da direita recebe o filho da esquerda do filho da direita. Caso todos esses casos sejam falsos, então colocamos para o filho da direita atualizar seu valor recebendo o valor retornado pela função `remover(noMaisDireita(pai.direita.esquerda))`. Por fim é retornado o valor.

Complexidade: O(n), já que chama a função de noMaisDireita() e recursivamente faz uma chamada linear da própria função.

Atualiza Números Nós

O método `atualizaNumeroNos()` atualiza os números referentes à quantidade de nós à esquerda e à direita da raiz. De forma recursiva chama a própria função passando os filhos à direita e à esquerda realizando um somatório.

Complexidade: Θ(n), pois são visitado todos os nós para o número de nós à esquerda e à direita serem atualizados.

No Mais a Direita

O método `noMaisDireita()` como o nome já diz, retorna o nó mais à direita. O processo é bem simples, por meio de um looping é pego sempre filho à direita enquanto ele não for nulo, depois disso é retornado o valor do filho mais à direita.

Complexidade: Θ(log n) pois ao buscar o nó mais à direita, metade dos nós são removidos ao descer um nível da árvore.

Get Nível

O método `getNivel()` retorna o nível do nó. Para isso ela verifica o seu valor é maior que o valor da raiz, caso sim, retorna a chamada da função, por meio de recursão, da própria função, com a própria raiz e o filho da direita, mais um; depois verifica se é menor que o valor da raiz, pois caso seja, ele retorna a própria função passando o nó e o seu filho da esquerda, somando isso a um: (`funcao(no, filho à esquerda) + 1`). Por fim, ele retorna 1, nisso, obtemos o esperado do nível, já que é executado recursivamente.

Complexidade: Θ(log n), pois checa se o valor é menor ou maior a cada chamada da função, dividindo a árvore em 2 a cada chamada.

Max Nível

O método `maxNivel()` verifica se o filho da esquerda do nó é diferente de nulo, caso verdadeiro, ele chama a função recursivamente passando o filho da esquerda, com isso ele verifica se a altura retornada é maior do que a que já temos, caso verdadeiro ele troca as alturas, se não, apenas prossegue, repetindo o mesmo processo, agora, para o filho da direita caso ele seja diferente de nulo. Após isso tudo ainda temos a validação de se o nó é uma folha, pois caso seja, o nível máximo vai receber a função de forma recursiva passando o nó e a raiz, por fim retorna o valor final que é o que queríamos.

Complexidade: Θ(n), pois todos passa por todos os nós.