AVL Tree - Respostas (1 a 4)

Antonio Deivid Santos Costa

April 2023

1 Dê exemplo de uma família de árvores AVL cuja exclusão de nós implica a realização de *O(lgn)* operações de rotação para o rebalanceamento.

Dê exemplo de uma família de árvores AVL cuja exclusão de nós implica a realização de *O(lgn)* operações de rotação para o rebalanceamento.

Um ótimo exemplo a ser dado é a fámilia de árvores fibonacci. Essa família é uma sequência de árvores fibonacci, em que cada árvore é obtida ao adicionar um novo nó à árvore anterior, mantendo a propriedade de árvore de busca binária e balanceada.

1. A primeira árvore na família de árvores fibonacci possui apenas um único nó, a raíz.
2. A segunda possui dois nós, raíz e filho.
3. A terceira possui três nós, raíz e dois filhos.
4. A quarta possui cinco nós, raíz e filhos.

E assim segue, obedecendo a sequência de fibonacci para o número de nós em cada árvore.

Com base no que foi dito, é certo dizer que a estrutura é mantida ao ato de inclusão de novos nós. Logo, ao remover um nó, não são necessárias operações de rotação para o rebalanceamento, visto que sua estrutura é preservada. Assim sendo, a remoção de um nó em uma árvore fibonacci implica em *O(1)* operações de rotação, uma complexidade de tempo constante, garantindo uma altura máxima de *O(lgn)* para a árvore.

Por conseguinte, como possui altura máxima igual a *O(lgn)*, onde n é o número de nós, uma família de árvores fibonacci é um exemplo de estrutura que utiliza no máximo *O(lgn)* para remoção de um nó, a tornando eficiente em termos de complexidade de tempo.

2 Detalhar o algoritmo de exclusão de nós em árvores AVL.

A exclusão de nós em uma árvore AVL pode, de certa forma, se assemelhar à inclusão. Ela também pode ser feita em *O(lgn)*; sempre que um nó é excluído, se faz necessária a verificação da árvore, no sentido de detectar se ela se tornou desregulada; os nós examinados pertencem a um caminho que vai da raíz até uma de suas folhas descendentes.

Dito isso, antes de qualquer coisa é necessário realizar uma busca pelo nó que irá ser removido, para então verificar se este nó é nulo e, caso seja, ou não esteja na árvore, não há o que fazer. Não há o que remover. Em contrapartida, se for encontrado o nó com a chave desejada, antes de removê-lo existem dois cenários a considerar: o nó em questão possuir, ou não, filho direito. Caso o nó x não tenha filho direito, o filho esquerdo assume a posição do pai e então x é liberado. Nesse cenário, é preciso que a altura de todos os nós ancestrais devem ser atualizadas e estes nós precisam ser regulados pela rotação apropriada, caso se faça necessário. Caso x possua filho direito, sua chave é trocada com a de seu sucessor e, então, este nó sucessor é liberado. Ainda, assim como no caso anterior, a altura de todos os nós no caminho do antigo pai do sucessor de x até a raíz precisam ser atualizadas, e os nós regulados pela rotação apropriada, caso se faça necessário.

Como é perceptível, apenas os nós ancestrais do nó fisicamente removido tem a possibilidade de se tornarem desregulados, logo, temos: Suponha um nó X

1. quando um nó Y é removido do lado esquerdo de X e as subárvores de X têm alturas iguais, a altura de X não é alterada, não há e nem haverá regulagem neste caso.

2. quando um nó Y é removido do lado esquerdo de X e as subárvores de X têm alturas diferentes, caso a subárvore removida seja a mais alta, a altura de X diminui. Não há regulagem em X, mas algum ancestrar pode precisar.

3. quando um nó Y é removido do lado esquerdo de X e as subárvores de X têm alturas diferentes, caso a subárvore removida seja a mais baixa, tudo irá depender do fator de balanceamento do filho direito de X. Se o fator for 0, realiza-se rotação a esquerda e a altura de X não se altera e nem é preciso regulagem. Se for igual a +1, acontece uma rotação a esquerda e altura de X diminui, podendo deixar algum ancestral desregulado. Por fim, se for igual a -1, ocorre uma rotação dupla a esquerda e a altura de X diminui, também podendo deixar algum ancestral desregulado.

3 Explicação da complexidade da questão 3

A função *add()* utiliza um loop while para percorrer a árvore AVL a partir do nó p até encontrar o local correto para inserir o novo nó com a chave key. Durante esse percurso, a função realiza comparações de chaves para determinar se deve descer para o filho esquerdo ou direito do nó current. Como a árvore AVL é balanceada, a altura máxima da árvore é *O(n + m)*, onde n é o número de nós na árvore. Logo, o tempo necessário para percorrer a árvore é proporcional à altura da árvore, ou seja, *O(n + m)*.

Após encontrar o local correto para inserção do novo nó, a função cria um novo nó com a chave key e o insere no local correto, ajustando os ponteiros dos nós e realizando rotações de balanceamento, se necessário. Essas operações têm um tempo constante, então não afetam a complexidade.

Após isso, a função retorna o nó raiz da árvore AVL atualizada, que pode ter sido alterada durante a inserção do novo nó.

Logo, a complexidade da função add é *O(n + m)*, pois o tempo de percorrer a árvore é o principal fator que determina a complexidade na maioria dos casos.

4 Explicação da complexidade da questão 4

A função mergeTrees leva *O(n + m)* porque seu desempenho é diretamente proporcional ao número de nós nas duas árvores de entrada, ou seja, n e m, respectivamente.

A função começa percorrendo ambas as árvores, av1 e av2, em ordem simétrica para coletar as chaves dos nós em dois vetores keys1 e keys2, respectivamente. O tempo necessário para percorrer uma árvore em ordem simétrica é proporcional ao número de nós na árvore. Portanto, o tempo de execução para essa parte é *O(n + m)*, onde n é o número de nós em av1 e m é o número de nós em av2.

Em seguida, a função faz um merge ordenado desses dois vetores, keys1 e keys2, para criar um novo vetor keys que contém todas as chaves em ordem crescente. Esse processo de merge ordenado leva *O(n + m)* de tempo, pois é necessário comparar as chaves em ambos os vetores e combiná-las em ordem crescente em um novo vetor.

Por fim, a função chama a função construir passando o vetor keys como entrada para construir uma nova árvore AVL balanceada. A função construir tem uma complexidade *O(n)*, pois é uma construção de árvore AVL balanceada a partir de um vetor ordenado com n elementos.

Portanto, somando todas as partes da função mergeTrees, a complexidade total é *O(n + m) + O(n) = O(n + m)*, uma vez que o termo de maior magnitude é *O(n + m)*.