Tópico 8: *Árvores* - Conceitos Gerais, Árvores Binárias e Árvores de Busca Binária

Prof. Dr. Juliano Henrique Foleis

Estude com atenção os vídeos e as leituras sugeridas abaixo. Os exercícios servem para ajudar na fixação do conteúdo e foram escolhidos para complementar o material básico apresentado nos vídeos e nas leituras. Quando o exercício pede que crie ou modifique algum algoritmo, sugiro que implemente-o em linguagem C para ver funcionando na prática. O único exercício que é necessário entregar está descrito na Seção "Atividade Para Entregar".

Vídeos

Árvores: Conceitos Gerais

Árvores Binárias: Estrutura e Percursos

Árvores de Busca Binária: Busca e Inserção

Explicação da Remoção em uma ABB

A remoção de um nó de uma árvore de busca binária deve ser realizada de forma que a árvore permaneça respeitando as propriedades de uma árvore de busca binária após a remoção. Existem 3 casos a considerar:

1. O nó a ser removido é folha;
2. O nó a ser removido tem um único filho; e
3. O nó a ser removido tem dois filhos.

Remoção de um Nó Folha

No caso que o nó a ser removido é folha, basta desalocá-lo, e fazer quem estava apontando pra ele passe a apontar para nada (NULL), como mostrado na Figura 1.

Remoção de um Nó com Apenas um Filho

Neste caso, basta fazer quem apontava para o nó a ser removido passe a apontar para o único filho do nó sendo removido. Finalmente, o nó sendo removido deve ser desalocado. Este processo está representado na Figura 2.

Remoção de um Nó com Dois Filhos

Este caso é um pouquinho mais complicado. Temos que considerar que os 2 filhos podem não ser folhas, ou seja, podem ter sub-árvores "penduradas"!

Vamos chamar o nó a ser removido de x. Como a árvore é uma árvore de busca binária, toda chave em x.esq é menor que x e toda chave x.dir é maior que x. Logo, a maior chave de x.esq também é menor que toda chave em x.dir. Portanto, se a maior chave de x.esq for colocada no lugar de x, a árvore continuará sendo uma ABB. O maior elemento de x.esq é chamado de *antecessor de x*. Da mesma forma, a menor chave de

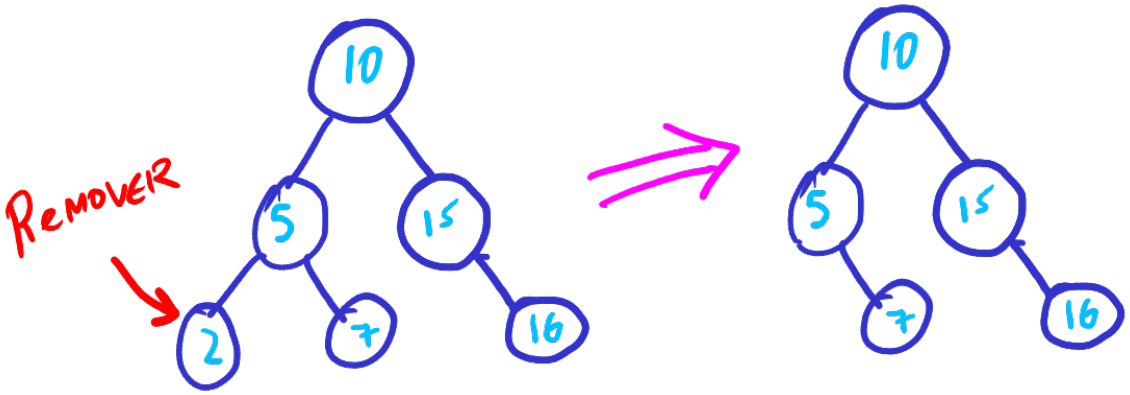

Figure 1: Remoção de um Nó Folha

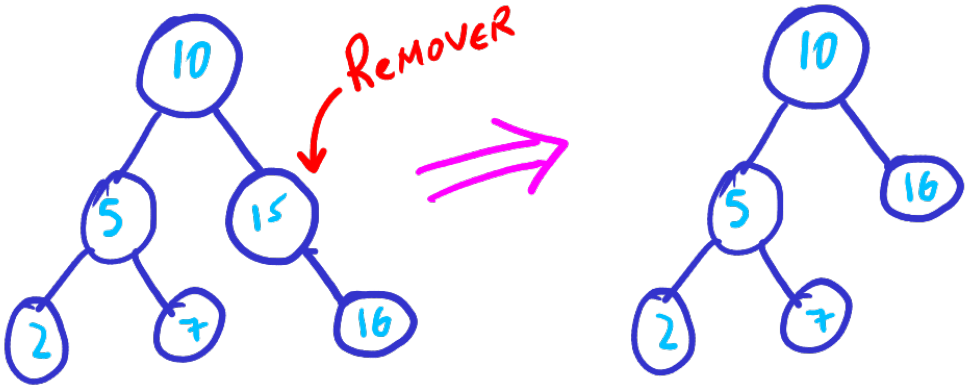

Figure 2: Remoção de um Nó com Apenas um Filho

x.dir é maior que toda chave em x.esq. Da mesma forma, se a menor chave de x.dir for colocada no lugar de x, a árvore continuará sendo uma ABB. O menor elemento de x.dir é chamado de *sucessor de x*. A Figura a 3 mostra o antecessor e o sucessor de 20 em uma árvore.

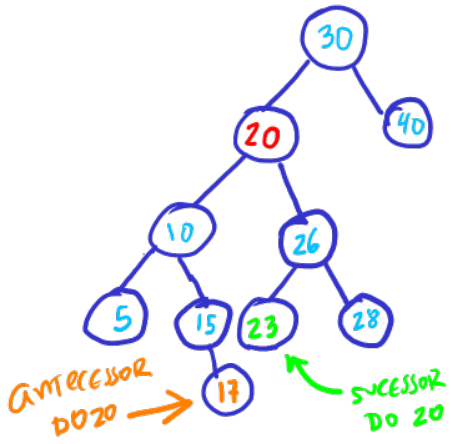

Figure 3: Antecessor e Sucessor de um Nó

Como visto na Figura, o antecessor é o maior valor da sub-árvore enraizada em x.esq. Como os itens em sub-árvores à direita são sempre maiores que sua raíz, o maior item de uma sub-árvore é sempre o ultimo elemento em um percurso que segue os ponteiros à direita. Dessa forma, o antecessor de x é encontrado seguindo o percurso dos ponteiros à direita de x.esq. Por exemplo, o antecessor de 20 na Figura 4 é encontrado seguindo o caminho $esq \rightarrow dir \rightarrow dir$ a partir do nó com chave 20.

Da mesma forma, o sucessor é o menor valor da sub-árvore enraizada em x.dir. Ele pode ser encontrado seguindo o percurso dos ponteiros à esquerda de x.dir. Por exemplo, o sucessor de 20 na Figura a seguir é encontrado seguindo o caminho $dir \rightarrow esq$ a partir de do nó com chave 20.

Portanto, para remover x, podemos colocar o antecessor ou o sucessor de x no lugar de x. Para deixar a simplificação mais enxuta, a explicação a seguir considera que x está sendo substituído por seu sucessor. A Figura 4 mostra o processo de remoção. Primeiro, o sucessor $s(x)$ é encontrado. Os dados de $s(x)$ substituem os dados de x no nó x. Neste momento, os dados de $s(x)$ estão replicados, como mostra a Figura 4. Agora basta remover o nó $s(x)$ original. A remoção de $s(x)$ pode ser feita usando a mesma rotina de remoção, e, por definição, $s(x)$ tem no máximo um filho. Portanto, sua remoção é trivial, conforme abordado acima.

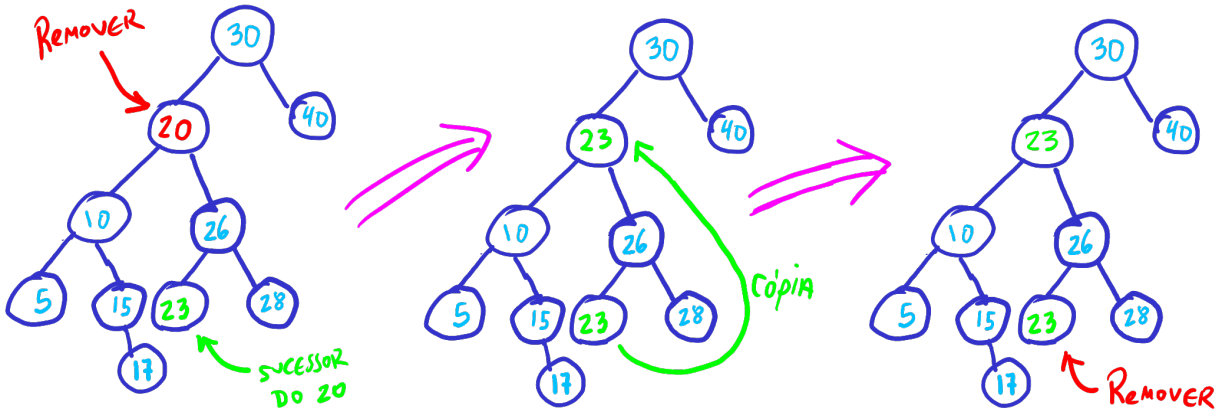

Figure 4: Remoção de um Nós com Dois Filhos

Leitura Sugerida

FEOFILOFF, Paulo. Estruturas de Dados. *Árvores binárias de busca (BSTs)* (Link)

Exercícios dos materiais de leitura sugerida

Exercícios 1.1, 1.2, 1.3, 1.4, 1.5, 2.1, 2.2, 2.3, 5.1, 5.3 da página do Prof. Feofiloff (Árvores binárias de busca (BSTs)): (Link)

Exercícios Complementares

1. Clone (ou atualize!) o repositório da disciplina no github. A implementação da árvore de busca binária está nos arquivos *bin_trees/abb.c* e *bin_trees/abb.h*.

a. Implemente as funções *ABB_Criar*, *ABB_Buscar* e *ABB_Inserir* conforme mostrado no vídeo.

b. Implemente a função *ABB_Imprimir* de forma que produza a saída mostrada no vídeo.

c. Escreva uma função recursiva *ABB_Tamanho* que devolva o número de nós de uma árvore binária.

d. Escreva uma função recursiva *ABB_Altura* que calcule a altura de uma árvore binária. Sua implementação deve ser preguiçosa (*lazy*), ou seja, não é necessário calcular as profundidades antes.

e. Acrescente um campo *profundidade* a estrutura ABB para armazenar a profundidade do nó. Escreva uma função *ABB_CalcularProfundidades* que atribua as profundidades de todos os nós.

f. O comprimento interno de uma árvore binária é a soma das profundidades dos seus nós, ou seja, a soma de todos os caminhos que levam da raíz até um nó. Escreva um método *ABB_ComprimentoInterno* que retorne o comprimento interno de uma árvore binária.

g. Escreva uma função *ABB_ABB* que receba uma árvore binária e verifique se ela é ou não uma árvore de busca binária. Retorne 1 caso seja uma ABB, ou 0, caso contrário.

h. Método Tamanho ansioso. No exercício **c** você provavelmente implementou a função *ABB_Tamanho* de forma preguiçosa, que examina toda a árvore e assim consome tempo proporcional ao número de nós na árvore. Escreva uma implementação mais eficiente usando a seguinte idéia (conhecida como implementação ansiosa, *eager*): acrescente a estrutura ABB um campo N, que guarde o número de nós na subárvore enraizada naquele nó. Dessa forma, para saber o tamanho da árvore, basta retornar N da raíz, que tem complexidade constante. N é atualizado durante as operações que alteram a estrutura da árvore, como a inserção. Altere também o método ABB_Inserir para atualizar o campo N conforme necessário, apenas dos nós no caminho da inserção. Você pode alterar a assinatura da função, se necessário.

i. Seguindo o raciocínio do exercício **h**, acrescente um campo inteiro *h* na estrutura ABB, e escreva uma versão ansiosa da função que retorne a altura da árvore binária (*ABB_Altura*). Altere as funções necessárias.

j. Seguindo o raciocínio do exercício **h**, acrescente um campo inteiro *ci* na estrutura ABB, e escreva uma versão ansiosa da função *ABB_ComprimentoInterno* que retorne o comprimento interno de uma árvore binária.

k. Implemente a função *ABB_CustoMedioBemSucedida*, que compute o custo médio de uma busca bem-sucedida, supondo que cada chave tem a mesma probabilidade de ser buscada. Considere que o custo de uma busca é o número de comparações de chaves.

l. Implemente a função *ABB_CustoMedioMalSucedida*, que compute o custo médio de uma busca malsucedida, supondo que cada chave tem a mesma probabilidade de ser buscada. Considere que o custo de uma busca é o número de comparações de chaves.

m. Implemente versões iterativas das funções *ABB_Buscar* e *ABB_Inserir*.

n. Implemente versões iterativas das funções *ABB_Tamanho*, *ABB_ABB*. **DICA:** use alguma estrutura de dados auxiliar para armazenar os nós a serem processados.

o. Implemente uma função *void ABB_Destruir(ABB** A)*, que desaloca todos os recursos usados pela árvore A.

2. Um percurso em-ordem de uma árvore de busca binária visita os nós da árvore em ordem crescente. Isto pode ser explorado para implementar um algoritmo de ordenação, conforme segue:

```
ENTRADA: vetor V com N inteiros
1. Crie uma ABB A
2. Insira todos os elementos de V em A
3. Faça um percurso em-ordem de A, inserindo os elementos de volta em V
4. Destrua a arvore A
```

a. Implemente a função *void ABBSort(int* v, int n)* conforme o pseudocódigo acima.

b. Qual é o custo do algoritmo acima no pior caso? Não é necessário fazer uma prova formal, apenas discutir qual seria esse custo.

c. No vídeo eu discuto que se as chaves forem uniformemente distribuídas, o custo de uma busca ou inserção é aproximadamente $1.4 \lg n$ se n for grande. Como você pode aproveitar essa idéia para fugir do custo no pior caso discutido na resposta do exercício anterior? Implemente a modificação e compare o resultado das duas implementações no pior caso.

BONS ESTUDOS!