Fundação Getulio Vargas, EMAp
Matemática Aplicada

Edgard Junio da Silva Viana,
Gabriel Pavanato.

Árvore Rubro-Negra

Relatório da Avaliação Substitutiva
Estrutura de Dados

Rio de Janeiro
julho de 2024

Construção da Árvore

Uma árvore rubro-negra é uma árvore binária *balanceada*, ou seja, ela tende a ficar com uma quantidade *simétrica* de nós em relação à raiz. Uma árvore rubro-negra é definida com as seguintes propriedades:

1. Todo nó é vermelho ou preto;
2. Caso um nó tenha um filho *NIL* então esse filho conta como um nó preto;
3. Um nó vermelho não tem filho vermelho;
4. Todo caminho partindo de um nó **N** até um *NIL* passa pela mesma quantidade de nós pretos;
5. (Opcional) A raiz é preta.

A imagem abaixo representa uma árvore rubro-negra.

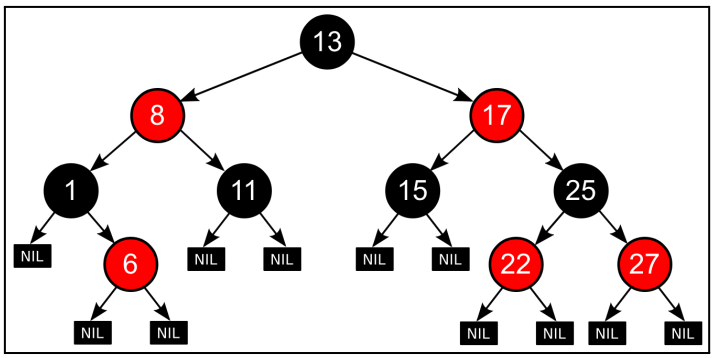

Figura 1: Exemplo de árvore rubro-negra

Implementação

Ao aplicar funções de inserção, remoção, etc, para uma árvore rubro-negra, é necessário criar casos particulares para manter a árvore balanceada, ou seja, dado um nó inicial, todos caminhos até um *nullpointer* deve conter a mesma quantidade de nós pretos, esta condição será muito útil para otimizações.

Inserção de Nó

A inserção em uma árvore rubro-negra envolve duas etapas principais: a inserção do nó como em uma árvore binária de busca (BST) comum e o ajuste das propriedades da árvore rubro-negra para manter seu balanceamento. Vamos explorar essas etapas em detalhes.

1. Inserção como uma BST comum

Primeiro, inserimos o novo nó na árvore como se fosse uma árvore binária de busca comum. Isso envolve encontrar a posição correta para o novo nó com base em seu valor.

1. **Criação do Novo Nó:** Um novo nó é criado com a função newNode. Este nó é inicialmente vermelho.
2. **Percorrendo a Árvore:** Percorremos a árvore a partir da raiz, procurando a posição correta para o novo nó.
3. **Inserção do Nó:** O novo nó é inserido na posição encontrada, sendo filho de um nó existente ou a nova raiz se a árvore estiver vazia.

2. Ajuste das Propriedades da Árvore Rubro-Negra

Depois de inserir o nó como em uma BST comum, ajustamos a árvore para garantir que ela satisfaça as propriedades da árvore rubro-negra. Isso é feito na função fixInsert.

1. **Verificação de Casos:** Se o pai do novo nó também é vermelho, precisamos corrigir a árvore. Existem três casos principais a serem considerados:
    - **Caso 1 (Tio é vermelho):** Se o tio do novo nó é vermelho, mudamos a cor do pai e do tio para preto, e do avô para vermelho, e subimos o ponteiro para o avô.
    - **Caso 2 (Tio é preto e o novo nó é filho direito):** Se o tio é preto e o novo nó é filho direito, rotacionamos o pai para a esquerda.
    - **Caso 3 (Tio é preto e o novo nó é filho esquerdo):** Se o tio é preto e o novo nó é filho esquerdo, rotacionamos o avô para a direita.
2. **Recolorindo:** Após as rotações e ajustes, recolorimos o nó conforme necessário.

3. **Garantia da Propriedade da Raiz Preta:** Asseguramos que a raiz da árvore seja sempre preta ao final do processo.

Resumo dos Passos

1. **Inserção:** Inserir o nó na posição correta como em uma BST.
2. **Ajuste:** Verificar e ajustar as cores e realizar rotações conforme necessário para manter as propriedades da árvore rubro-negra.

A inserção e os ajustes subsequentes garantem que essas propriedades sejam mantidas, assegurando o balanceamento da árvore e a eficiência das operações de busca, inserção e remoção.

Remoção de Nó

Esta função é responsável por remover um nó de uma árvore rubro-negra. A remoção de um nó pode desbalancear a árvore, então é necessário aplicar correções para manter as propriedades da árvore rubro-negra.

1. **Busca do Nó a Ser Removido**: A função searchNode é chamada para encontrar o nó que contém o valor data. Se o nó não for encontrado, a função retorna a raiz original, pois não há nada a remover.
2. **Armazenamento de Informações Temporárias**: As variáveis ptrTemp, ptrSubstitution e originalColor são usadas para armazenar informações temporárias sobre o nó a ser removido, seu substituto e a cor original do nó a ser removido, respectivamente.
3. **Remoção de Nó com Zero ou Um Filho**: Se o nó a ser removido não tem filho à esquerda, ele é substituído por seu filho à direita. Se não tem filho à direita, ele é substituído por seu filho à esquerda.
4. **Remoção de Nó com Dois Filhos**: Se o nó a ser removido tem dois filhos, o menor nó da subárvore direita (ou seja, o nó com o menor valor que é maior que o nó a ser removido) é encontrado e utilizado como substituto. Dependendo da posição do substituto, diferentes ajustes são feitos para manter a estrutura da árvore.
5. **Correção da Árvore**: Se a cor original do nó removido era preta, a função fixDelete é chamada para corrigir a árvore e garantir que ela continue satisfazendo as propriedades de uma árvore rubro-negra.

Função auxiliar Transplant:

Esta função substitui uma subárvore por outra, ajustando os ponteiros dos pais adequadamente.

1. **Substituição na Raiz**: Se o nó a ser substituído é a raiz, o nó de substituição torna-se a nova raiz.
2. **Substituição de um Filho Esquerdo ou Direito**: Se o nó a ser substituído não é a raiz, ele é substituído pelo nó de substituição, ajustando os ponteiros do pai para apontar para o novo nó.
3. **Atualização do Pai do Nó de Substituição**: Se o nó de substituição não é nulo, seu ponteiro de pai é atualizado para apontar para o pai do nó substituído.

Função auxiliar FixDelete:

Esta função corrige a árvore após a remoção de um nó preto, para garantir que as propriedades da árvore rubro-negra sejam mantidas.

1. **Laço de Correção**: O laço continua até que o nó a ser deletado seja a raiz ou até que ele seja vermelho.
2. **Correção para Filho Esquerdo**: Se o nó a ser deletado é o filho esquerdo, o irmão do nó é analisado e correções são feitas dependendo da cor do irmão e dos seus filhos.
3. **Correção para Filho Direito**: Se o nó a ser deletado é o filho direito, uma lógica similar à do filho esquerdo é aplicada, ajustando as cores e realizando rotações conforme necessário.
4. **Ajuste Final da Cor**: Finalmente, a cor do nó a ser deletado é ajustada para preto, se necessário.

Busca Nó por Valor

Um ponto forte das *BST's* é a busca de nó por valor, onde isto se vê presente até no nome da estrutura (*Binary Search Trees*). Como a árvore rubro-negro é uma *BST*, podemos fazer uso desse algoritmo, onde não há espaço para otimizações. O algoritmo functiona da seguinte forma.

1. Compara o valor buscado (data) com o valor do nó atual (root->data).
2. Se a data for menor que o valor do nó atual, a função se move para a subárvore esquerda.
3. Se a data for maior ou igual, a função se move para a subárvore direita.
4. O processo se repete até encontrar o nó com o valor desejado ou até alcançar um nó nulo (indicando que o valor não está na árvore).

A função retorna o ponteiro para o nó encontrado ou nullptr se o valor não estiver presente na árvore.

Percurso InOrder

A função inorderTraversal realiza a travessia em ordem (in-order traversal) de uma árvore binária. A travessia em ordem segue três etapas principais para cada nó:

1. **Visita a subárvore esquerda.**
2. **Processa o nó raiz (imprime seu valor).**
3. **Visita a subárvore direita.**

Essa abordagem garante que os nós da árvore binária de busca sejam visitados em ordem crescente. Se o nó atual é nulo, a função retorna imediatamente, servindo como condição de parada.

Verificação da validez de uma árvore

Essa função tem como objetivo verificar se todas as propriedades de uma árvore Rubro-Negra são válidas:

1. **Verificação Inicial**:
    - Primeiro, verificamos se o nó raiz é nulo. Se for, a árvore é considerada uma árvore Rubro-Negra válida, pois uma árvore vazia cumpre todas as propriedades necessárias.
2. **Verificação da Cor da Raiz**:
    - Em seguida, verificamos se a raiz da árvore é preta. Esta é uma das propriedades fundamentais das árvores Rubro-Negras. Se a raiz não for preta, a função retorna false imediatamente.
3. **Contagem de Nós Pretos**:
    - Se as verificações iniciais forem satisfeitas, iniciamos um loop que conta a quantidade de nós pretos no caminho da raiz até as folhas mais à esquerda e à direita. Utilizamos duas variáveis para essa contagem:
        - iAmountBlack para contar os nós pretos no caminho à esquerda.
        - iAmountright para contar os nós pretos no caminho à direita.
    - Durante esse percurso, também verificamos se há dois nós vermelhos consecutivos, o que violaria as propriedades da árvore Rubro-Negra.
4. **Verificação de Altura Negra e Nós Vermelhos Consecutivos**:
    - Após contar os nós pretos, iniciamos um segundo loop para percorrer a árvore e garantir que todos os caminhos da raiz até as folhas têm a mesma quantidade de nós pretos (altura negra).
    - Utilizamos uma pilha para realizar uma travessia in-order da árvore.
    - Durante essa travessia, verificamos:

        - Se todos os caminhos têm a mesma altura negra.
        - Se há nós vermelhos com filhos vermelhos.
5. **Resultado Final**:
    - Se todas as verificações forem passadas com sucesso, a função retorna true, indicando que a árvore é uma árvore Rubro-Negra válida. Caso contrário, retorna-se false.

Este processo garante que todas as propriedades de uma árvore Rubro-Negra sejam rigorosamente verificadas, assegurando a integridade da estrutura.

Encontrar Nó Máximo

A função maxNode tem como objetivo encontrar o valor máximo em uma árvore binária rubro-negra. Ela realiza isso ao navegar recursivamente até o elemento mais à direita da árvore, pois em uma árvore binária de busca, o maior valor sempre se encontra no nó mais à direita. A função continua a chamar a si mesma recursivamente até alcançar o nó que não possui um filho direito, retornando assim o nó com o maior valor na árvore.

Encontrar Nó Mínimo

A função minNode tem como objetivo encontrar o valor mínimo em uma árvore binária rubro-negra. Ela realiza isso ao navegar recursivamente até o elemento mais à esquerda da árvore, pois em uma árvore binária de busca, o menor valor sempre se encontra no nó mais à esquerda. A função continua a chamar a si mesma recursivamente até alcançar o nó que não possui um filho esquerdo, retornando assim o nó com o menor valor na árvore.

Calcular a altura

Esta função tem uma otimização muito interessante por conta da estrutura da árvore rubro-negra. Caso um nó tenha dois filhos, um vermelho e um preto, não é necessário caminhar pela sub-árvore que o filho preto gera, pois é garantido que na sub-árvore do filho vermelho haverá pelo menos mais um nó preto, para satisfazer a condição dos caminhos com mesmo número de pretos, onde o único caso em que o nó vermelho não garante outro filho é caso ele seja o próprio nó que determina a altura da árvore.

O algoritmo checa se a partir da raiz existem nós de cores diferentes, se sim, nos interessa seguir apenas o que for de cor vermelha, onde isso ocorre chamando recursivamente a função para esse nó. Caso contrário, a função chama recursivamente os dois lados e pega o máximo.

Testes

Os testes abaixo são resultado do próprio código no GitHub. Primeiro iniciamos com a criação da seguinte árvore:

```
Árvore após inserção de 5, 3, 7 ,2 ,1 ,13 e 11:
        5B
   2B       11B
1R  3R   7R   13R

Busca por 3: Encontrado
Busca por 4: Não encontrado
Mínimo: 1
Máximo: 13
InOrderTraversal: 1 2 3 5 7 11 13
Essa árvore é válida
```

Depois fazemos a seguinte inserção:

```
Árvore após inserção do 0:
          5B
     2R        11B
  1B   3B   7R   13R
0R

Busca por 3: Encontrado
Busca por 4: Não encontrado
Mínimo: 0
Máximo: 13
InOrderTraversal: 0 1 2 3 5 7 11 13
Essa árvore é válida
```

Após a inserção do zero, a principal mudança foi a alteração nas cores da maioria dos nós. Isso ocorreu porque o zero devia ser inserido à esquerda da raiz, mas tanto o nó 1 quanto o nó 3 eram vermelhos. Não poderíamos simplesmente adicionar um nó vermelho sem violar as propriedades da árvore rubro-negra. Portanto, foi necessário modificar as cores para manter as propriedades da árvore rubro-negra, garantindo que ela continue balanceada e respeitando as regras de cores.

```
Árvore após inserção do -1:
              5B
        2R          11B
    0B    3B    7R    13R
-1R   1R

Busca por 3: Encontrado
Busca por 4: Não encontrado
Mínimo: -1
Máximo: 13
InOrderTraversal: -1 0 1 2 3 5 7 11 13
Essa árvore é válida
```

Quando inserimos o valor -1, foi necessária uma rotação para manter as propriedades da árvore rubro-negra. A rotação fez com que o nó 1, que anteriormente era pai do nó 0, se tornasse o filho direito do nó 0. Como resultado, o nó 0 passa a ser pai do -1 e do 1.