Estrutura de Dados

Hamilton José Brumatto

Bacharelado em Ciências da Computação - UESC

7 de fevereiro de 2024

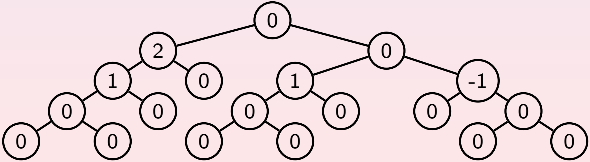
Árvores Balanceadas - AVL

Árvores Balanceadas

- As árvores Binárias de Busca são excelentes para busca rápida de elementos
- Esta eficiência somente vale se á arvore for cheia, ou quase cheia.
- Em especial se a árvore tem altura máxima de $\log n$.
- Neste caso, dizemos que a árvore é balanceada.
- Uma árvore somente será balanceada se seus filhos e descendentes também forem balanceados.
- O grau do balanceamento determinará a eficiência da busca.

Árvores Balanceadas

- O grau de balanceamento de uma árvore se dá pela máxima diferença absoluta entre as alturas de seus filhos esquerdo e direito em todos os ramos.
- Na figura abaixo, nos nós estão indicadas as diferenças entre a altura do filho esquerdo e direito.
- Embora no raiz este valor é 0, o grau de desbalanceamento da árvore é 2. Ou seja, a diferença entre a altura dos filhos está entre -2 e 2, inclusive.
- Vamos chamar a diferença entre a altura dos filhos de um nó de Fator de Desbalanceamento (FD) de um nó.

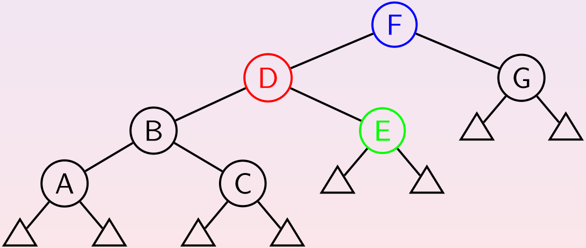


Árvores AVL

- As árvores AVL são árvores balanceadas de grau máximo 1 (poderia ser grau 0).
- A árvore AVL é então uma árvore binária de busca ótima, pois a busca ou inserção de elementos é sempre $O(\log n)$.
- As operações de inserção e remoção podem destruir o balanceamento. É preciso refazer o balanceamento, se necessário.
- Para refazer o balancemanto lançaremos mão de operações de rotações na estrutura da árvore.

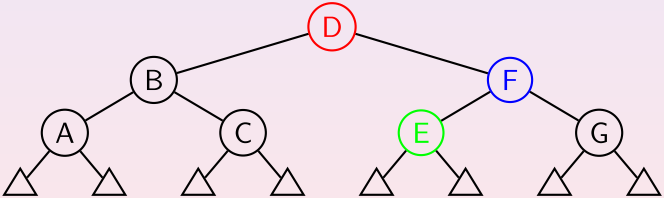
Rotação Direita

- Na rotação direita o filho esquerdo toma o lugar do raiz, e o raiz passa a ser o filho direito. Se houver filho direito de quem era filho, ele passa a ser filho esquerdo de quem era o raiz.



Rotação Direita

- Na rotação direita o filho esquerdo toma o lugar do raiz, e o raiz passa a ser o filho direito. Se houver filho direito de quem era filho, ele passa a ser filho esquerdo de quem era o raiz.



Rotação Direita - Algoritmo

Algoritmo $\textsc{RodarDir}(\mathit{raiz})$

$p \leftarrow \texttt{filhoEsq}(\mathit{raiz})$

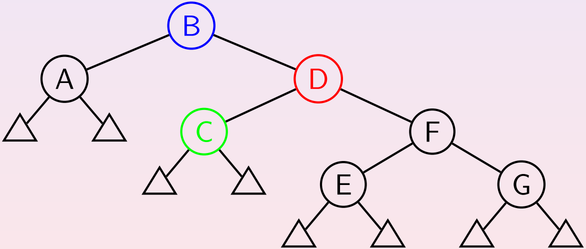
$\texttt{insFilhoEsq}(\mathit{raiz}, \texttt{filhoDir}(p))$

$\texttt{insFilhoDir}(p, \mathit{raiz})$

$\mathit{raiz} \leftarrow p$

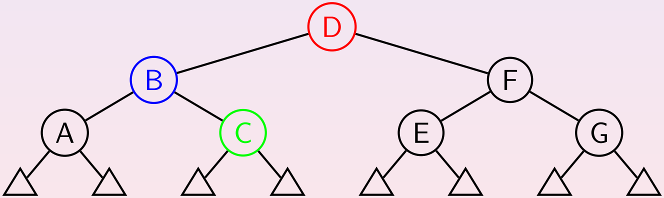
Rotação Esquerda

- Na rotação esquerda o filho direito toma o lugar do raiz, e o raiz passa a ser o filho esquerdo. Se houver filho esquerdo de quem era filho, ele passa a ser filho direito de quem era o raiz.



Rotação Esquerda

- Na rotação esquerda o filho direito toma o lugar do raiz, e o raiz passa a ser o filho esquerdo. Se houver filho esquerdo de quem era filho, ele passa a ser filho direito de quem era o raiz.



Rotação Esquerda - Algoritmo

Algoritmo $\textsc{RodarEsq}(\mathit{raiz})$

$p \leftarrow \texttt{filhoDir}(\mathit{raiz})$

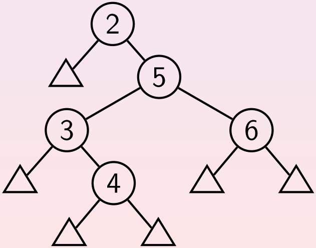
$\texttt{insFilhoDir}(\mathit{raiz}, \texttt{filhoEsq}(p))$

$\texttt{insFilhoEsq}(p, \mathit{raiz})$

$\mathit{raiz} \leftarrow p$

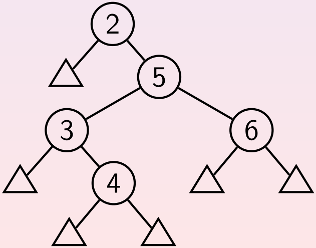
Quando Rodar?

- Verificamos nos exemplos anteriores que se um nó tem um FD positivo, a altura de seu filho esquerdo supera a altura do filho direito, quando este fator é maior que 1 realizamos uma rotação para a direita para balancear a árvore.
- Da mesma forma se o FD por menor que -1, então realizamos uma rotação para a esquerda.
- Veja o seguinte caso, que rotação faremos?



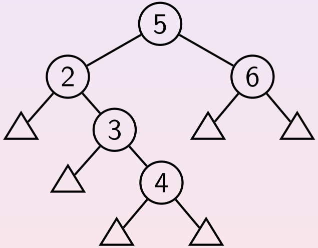
Quando Rodar?

- Verificamos nos exemplos anteriores que se um nó tem um FD positivo, a altura de seu filho esquerdo supera a altura do filho direito, quando este fator é maior que 1 realizamos uma rotação para a direita para balancear a árvore.
- Da mesma forma se o FD por menor que -1, então realizamos uma rotação para a esquerda.
- Veja o seguinte caso, que rotação faremos? Esquerda?



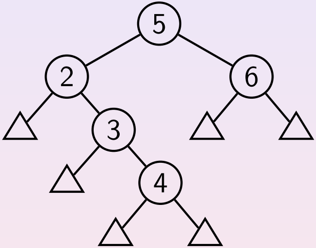
Quando Rodar?

- Rodando para a esquerda:



Quando Rodar?

- Rodando para a esquerda:



- Além de não balancear, agora o filho também ficou desbalanceado. Isto é ruim em uma chamada recursiva.
- O balanceamento da árvore deveria ser: balancear os filhos recursivamente depois balancear o raiz. Se balancear o raiz desbalanceia o filho será complicado.

Quando Rodar?

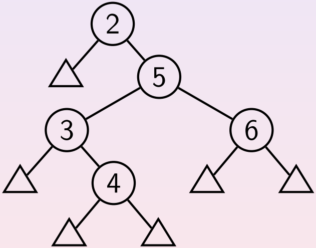
- Fizemos uma rotação Esquerda, mas antes da rotação o filho direito já tinha um fator de desbalanceamento positivo.
- Ou seja seu filho esquerdo do filho direito já tinha uma altura maior.
- Ao girar, este "neto" esquerdo é inserido como filho direito daquele que era antigo raiz.
- Isto faz com que a diferença aumente.
- Precisamos evitar que o "neto" que será deslocado seja o que tenha maior altura.
- Para isso vamos rodar no sentido contrário, primeiro, o filho que tem um desbalanceamento contrário da rotação que faremos.

Quando rodar?

- Se o fator de desbalanceamento for positivo, maior que um.
  - Se o fator de desbalanceamento do filho direito for negativo, rotação esquerda no filho esquerdo e rotação direita no raiz (RotaçãoEsquerdaDireita: ED ou LR). Senão
  - Simples rotação direita no raiz (RotaçãoDireita: D ou R).
- Se o fator de desbalanceamento for negativo, menor que um negativo.
  - Se o fator de desbalanceamento do filho direito for positivo, rotação direita no filho direito e rotação esquerda no raiz (RotaçãoDireitaEsquerda: DE ou RL). Senão
  - Simples rotação esquerda no raiz (RotaçãoEsquerda: E ou L).

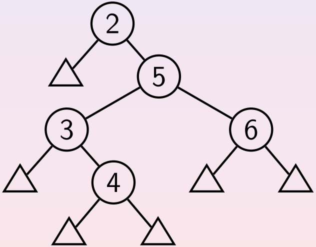
Quando Rodar?

- Veja o seguinte caso, que rotação faremos?



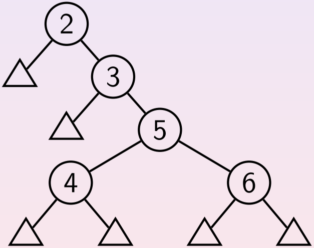
Quando Rodar?

- Veja o seguinte caso, que rotação faremos? Primeiro uma rotação Direita no filho direito

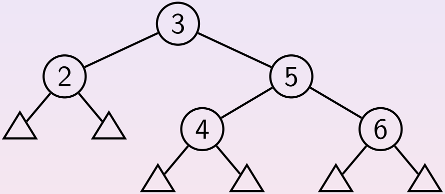


Quando Rodar?

- Agora uma rotação Esquerda no raiz



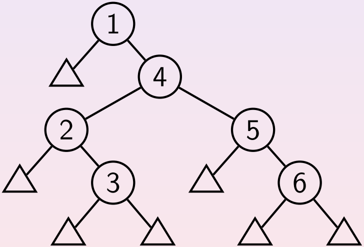
Quando Rodar?



- Isto resolve o balanceamento nos casos em que inserimos novos elementos na árvore.
- Mas isto não resolve um problema geral de desbalanceamento.

Regras não são suficientes para o caso geral

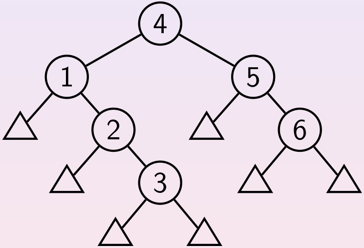
- Veja o exemplo abaixo:
- Não conseguiremos este exemplo inserindo e balanceando ao inserir.



- Pelas nossas regras, devemos aplicar uma rotação Esquerda no raiz, simplesmente.

Regras não são suficientes para o caso geral

- Veja o resultado:



- Neste caso não tem jeito, temos de colocar em loop o balanceamento a partir do raiz até que tudo esteja balanceado.

Algoritmos de Balanceamento

Algoritmo FATOR(*raiz*)

$he = \texttt{altura}(\texttt{filhoEsq}(raiz))$

$hd = \texttt{altura}(\texttt{filhoDir}(raiz))$

retorne $he - hd$

Algoritmos de Balanceamento

Algoritmo EHBALANCEADA(*raiz*)
  boolean $bal = verdade$
  se não `ehVazia`(*raiz*) **então**
    $bal = (abs(\texttt{fator}(raiz)) <= 1)$
    se *bal* **então**
      $bal = bal$ **e ehBalanceada**(`filhoEsq`(*raiz*))
      $bal = bal$ **e ehBalanceada**(`filhoDir`(*raiz*))
  retorne *bal*

Algoritmos de Balanceamento

```
Algoritmo BALANCEAR(raiz)
  se não ehVazia(raiz) então
    enquanto não ehBalanceada(raiz) faça
      Balancear(filhoEsq(raiz))
      Balancear(filhoDir(raiz))
      fd ← fator(raiz)
      se fd > 1 então
        se fator(filhoEsq(raiz)) < 0 então
          rotacaoEsquerdaDireita(raiz)
        senão
          rotacaoDireita(raiz)
      se fd < −1 então
        se fator(filhoDir(raiz)) > 0 então
          rotacaoDireitaEsquerda(raiz)
        senão
          rotacaoEsquerda(raiz)
```

Árvores AVL - Inserção e Remoção de elementos

- Para inserir e remover um elemento, usamos o algoritmo já visto.
- Em seguida, realizamos um balanceamento.

Algoritmo $\textsc{InsAVL}(\textit{raiz}, \textit{valor})$
    `InsValor`$(\textit{raiz}, \textit{valor})$
    `Balancear`$(\textit{raiz})$

Algoritmo $\textsc{RemAVL}(\textit{raiz}, \textit{valor})$
    boolean $ret = RemValor(raiz, valor)$
    se *ret* **então**
        `Balancear`$(\textit{raiz})$
    retorne *ret*

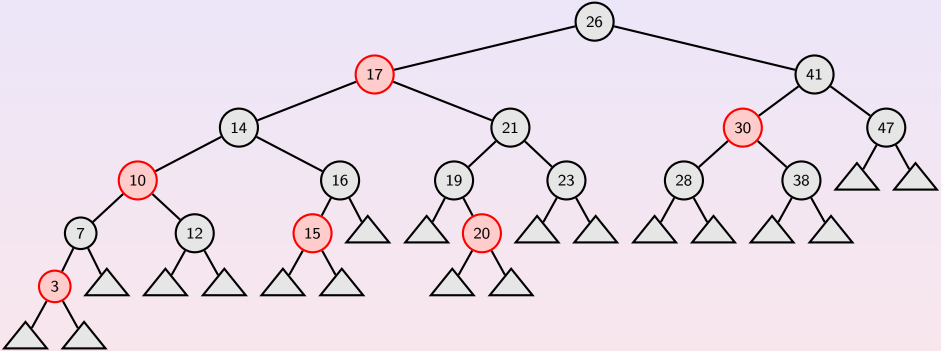
Árvores Rubro-Negras

- As árvores Rubro-Negras também são conhecidas como Vermelho-Preto ou *Red-Black Trees*.
- Estas árvores são árvores binárias de busca, e são aproximadamente balanceadas.
- O objetivo é que qualquer operação seja $O(\log n)$ (busca, inserção, remoção, ...)
- Por ser "aproximadamente" balanceada, as operações de balanceamento são menos custosas.
- Foram inventadas por Bayer com o nome de "Árvores Binárias Simétricas" em 1972, quase 10 anos depois das árvores AVL.
- As árvores Rubro-Negra tem altura no máximo $2\log(n+1)$

Árvores Rubro-Negras

- Os campos de uma árvore Rubro-Negra são: 1 bit de cor, o valor que armazena (*info*) os filhos esquerdo e direito e o pai, estes últimos são ponteiros.
- As árvores rubro-negras devem seguir as seguintes regras:
  1. Todo nó da árvore é vermelho ou preto.
  2. A raiz é preta.
  3. Toda folha (árvore) vazia é preta.
  4. Se um nó é vermelho, então ambos os filhos são pretos.
  5. Para todo nó, todos os caminhos do nó até as folhas descendentes contêm o mesmo número de nós pretos.
- A regra 5 define a altura negra da árvore rubro-negra.

Exemplo de uma árvore rubro-negra



- Para facilitar a visualização não incluiremos mais as árvores vazias, filhas das folhas. Mas todas são sempre pretas.

Operações em uma árvore rubro-negra

- As operações na árvore são de inserção ou remoção. Vamos começar com a operação de inserção.
- Quando inserimos um novo nó ele será inserido na cor vermelha (com suas árvores vazias pretas). A seu info será atribuído o valor fornecido.
- A operação de inserção segue as regras da inserção em uma árvore binária.
- Como na árvore AVL, onde a inserção pode alterar o balanceamento da árvore, aqui a inserção pode violar alguma das regras definidas para a árvore Rubro-Negra. Neste caso pode-se lançar mão das rotações, como na AVL.
- Inicialmente ao criar uma árvore a inserir, seus filhos e pai são NULL.

Algoritmo de inserção:

Algoritmo INSERIR_RN(T,val)
  $a \leftarrow Arvore\_RN(val)$
  $f \leftarrow NULL$
  $p \leftarrow T$
  enquanto $p \neq NULL$ **faça**
    $f \leftarrow p$
    se $val < p->info$ **então**
      $p \leftarrow p->filhoEsq$
    senão
      $p \leftarrow p->filhoDir$
  $a->pai \leftarrow f$
  se $f = NULL$ **então**
    $T \leftarrow a$
  senão
    se $val < f->info$ **então**
      $f->filhoEsq \leftarrow a$
    senão
      $f->filhoDir \leftarrow a$
  corrigir_ins(T, a)

Operação de inserção

- Podemos observar no algoritmo a chamada da função: `corrigir`
- Isto acontece pois como ressaltamos a inserção pode violar alguma regra.
- Vamos analisar quais regras poderiam ser violadas por esta operação:
    - As regras 1, 3 e 5 se mantém: Todos nós são vermelho ou preto, as árvores vazias são todas pretas. E todo caminho de um nó para seus descendentes continuam tendo o mesmo número de nós pretos.
    - A regra 2 pode ser violada, se for o primeiro nó a ser inserido. Aí basta mudar a cor deste nó.
    - A regra 4 pode ser violada, se o nó for inserido como filho de um nó vermelho.

Corrigir Inserção

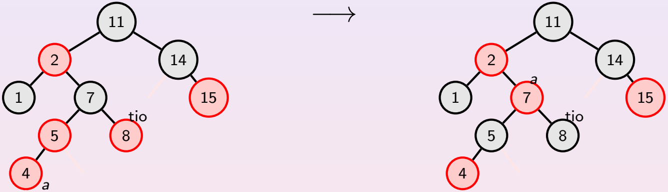
- Vamos por partes para entender o algoritmo.

Algoritmo CORRIGIR_INS(T, a) ▷ T é o raiz e a o elemento inserido
  enquanto $T \neq a$ **e** $a->pai->cor = V$ **faça**
    ...
  $T->cor = P$

- Se a é o raiz, no final ele será preto. Se esta era a violação, ela está corrigida.
- Se a não é o raiz, e $a->pai$ é vermelho, então é garantido que exista $a->pai->pai$ pois o raiz é preto.
- A única violação, então é a regra 4.
- Vamos corrigir agora, considerando que a esteja do lado esquerdo do pai, (no **se**) no **senão** basta trocar esquerdo por direito.

Corrigir Inserção

- No primeiro caso, tanto o pai, quanto o "tio" de *a* são vermelhos, então pintamos os dois de preto e o avô de vermelho, e tomamos *a* o avô.



- As regras 1, 2, 3 e 5 continuam válidas, em especial a regra 5, pois, o avô era preto, e o pai e tio vermelhos, agora inverteu, então todos caminhos até os descendentes continuam com o mesmo número de nós pretos.
- Mas a regra 4 pode continuar violada, neste caso o avô é o novo nó vermelho cujo pai é vermelho.

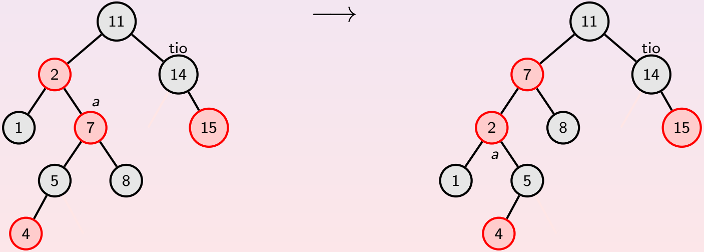
Corrigir Inserção

- Tanto o pai, quanto o tio são vermelhos.

Algoritmo CORRIGIR_INS(T, a) ▷ T é o raiz e a o elemento inserido
  enquanto $T \neq a$ **e** $a->pai->cor = V$ **faça**
    se $a->pai = a->pai->pai->filhoEsq$ **então** ▷ O pai é filho esquerdo
      $tio \leftarrow a->pai->pai->filhoDir$
      se $tio->cor = V$ **então**
        $a->pai->cor \leftarrow P$
        $a->pai->pai->cor \leftarrow V$
        $tio->cor \leftarrow P$
        $a \leftarrow a->pai->pai$
      senão
        ... ▷ Correção por rotações
    senão
      ... ▷ O pai é filho direito, então temos de pegar o tio esquerdo
  $T->cor = P$

Corrigir Inserção

- Ainda falamos do nó cujo pai é filho esquerdo do avô.
- Se o nó for filho direito do pai, é necessário uma rotação dupla esquerda-direita, senão, uma rotação simples direita.
- Na rotação dupla, a primeira rotação esquerda não há troca de cores, o novo nó *a* passa a ser quem era seu pai:

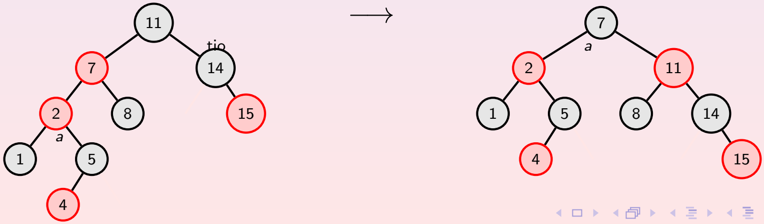


Corrigir Inserção

- A rotação dupla aconteceu quando o ó era filho direito do pai.
- Feita esta rotação, podemos observar que as regras 1, 2, 3 e 5 também se mantiveram.
- A regra 5, pois, do pai de a, do lado esquerdo tinha *n* descendentes pretos, e *a* também tinha *n* descendentes pretos.
- Após a rotação o filho esquerdo de *a* passou a ser filho direito de quem era pai de *a*, ou seja quem era pai de *a* continua com *n* descendentes pretos, da mesma forma que *a*.
- Após a rotação, o novo *a* continua com seu pai vermelho, porém agora *a* é filho esquerdo, então teremos de fazer uma rotação direita do pai de *a*.
- Nesta rotação, o pai de *a* e o seu avo mudam as cores. O pai passa a ser preto e quem era avô vermelho.

Corrigir Inserção

- Esta rotação mantém as regras 1,2,3 e 5 corretas, e corrige a regra 2.
- A regra 2, pois o novo raiz desta subárvore passa a ser preto (que pode ser o raiz geral)
- O filho direito de quem era avô era preto, então mudar a cor do avô para vermelho não afeta as regras
- A regra 5, pois no caminho direito do novo raiz, não aumentou a quantidade de pretos, o "avô" que passou a fazer parte do caminho, passou a ser vermelho.



Corrigir Inserção: Código

Algoritmo CORRIGIR_INS(T, a) ▷ T é o raiz e a o elemento inserido
  enquanto $T \neq a$ **e** $a->pai->cor = V$ **faça**
    se $a->pai = a->pai->pai->filhoEsq$ **então** ▷ O pai é filho esquerdo
      $tio \leftarrow a->pai->pai->filhoDir$
      se $tio->cor = V$ **então**
        $a->pai->cor \leftarrow P$
        $a->pai->pai->cor \leftarrow V$
        $tio->cor \leftarrow P$
        $a \leftarrow a->pai->pai$
      senão
        se $a = a->pai->filhoDir$ **então** ▷ Rotação esquerda-direita
          $a \leftarrow a->pai$ ▷ Rotação esquerda
          rodar_esq(T, a)
        $a \leftarrow a->pai$ ▷ Rotação direita com troca de cores
        $a->cor \leftarrow P$
        $a->pai->cor \leftarrow V$
        rodar_dir(T, a)
    senão
      ... ▷ O pai é filho direito, então temos de pegar o tio esquerdo
  $T->cor = P$

Operação de Remoção

- A operação de elementos da árvore Rubro-Negra segue a remoção de elementos em uma árvore binária de busca.
- Ou seja, precisamos ver se o nó é folha, se não possui nó irmão e por fim, escolher um sucessor ou predecessor para substituí-lo.
- A remoção pode destruir algumas regras da formação da árvore RN.
- É necessário, então chamar um algoritmo de correção da árvore.
- A correção também acontece realizando trocas de cores e/ou rotações.
- A de se considerar mais casos possíveis na remoção, o que torna o algoritmo de correção mais complexo.
- Este algoritmo não será abordado nesta disciplina, mas pode ser encontrado no livro Algoritmos de Cormen et al.

Comparando as árvores AVL e Rubro-Negra

- Operação de busca, tanto AVL quanto RN são $O(\log(n))$. Embora como a AVL é melhor balanceada, pode ser uma pouco mais rápida na busca.
- Operação de Inserção, AVL necessita somente de uma rotação (simples ou dupla), o mesmo a RN, porém a RN pode precisar de $\log(n)$ trocas de cores.
- Operação de Remoção, AVL pode fazer $\log(n)$ rotações, enquanto a RN faz no máximo 1 rotação (simples ou dupla)
- A árvore RN tem melhor pior caso que a árvore AVL, o que pode ser importante para estruturas que precisam de garantias no pior caso.