Universidade Federal do Rio Grande do Norte
Instituto Metrópole Digital

Estruturas de Dados Clássicas: Árvores Binárias

Bianca Medeiros, Gabriel Carvalho, Marina Medeiros, Vinicius de Lima

Estrutura de Dados Básica II

Trabalho da Segunda Unidade

9 de janeiro de 2025

Sumário

Capítulo 1

Árvore Heap

Introdução

A Heap (ou Binary Heap) é uma estrutura de dados que representa uma árvore binária completa ou quase-completa. Nesse tipo de árvore, todos os níveis, exceto possivelmente o último, estão preenchidos, da esquerda para a direita.

Existem duas variações de Binary Heap: Max Heap e Min Heap. No Max Heap, os pais têm valores maiores que ou iguais aos dos filhos, enquanto no Min Heap, os pais têm valores menores que ou iguais aos dos filhos.

Nesse projeto, a Heap foi implementada em Rust e foram utilizados testes para verificar as funções de Alteração de Prioridade, Inserção, Remoção (da raiz) e Construção das Heaps. Tais testes podem ser vistos funcionando no vídeo enviado pelo grupo.

HeapSort

HeapSort é um algoritmo de ordenação que usa uma Árvore Heap para classificar os elementos. Ele opera construindo o heap, extraindo repetidamente a raiz e reconstituindo a heap.

Para isso, algumas funções auxiliares são necessarias como: construir(tranforma uma lista em Heap levando em conta suas propriedades e características), subir(move um elemento para níveis superiores) e descer (move um elemento para níveis inferiores).

Dessa forma, estabelecemos o pseudocódigo da HeapSort como:

Algorithm 1 HeapSort

Input: Lista $A = A_1, A_2, \ldots, A_n$
Output: Lista A ordenada

```
1: function HeapSort(A)
2:     construir(A)
3:     for j ← A.tamanho to 1 do
4:         troque A[1] por A[i]
5:         A.tamanho = A.tamanho-1;
6:         descer (H, 1);
7:     end for
8: end function
```

Capítulo 2

Árvore Binária

Para criar uma árvore binária pouco desequilibrada a partir de uma lista de dados de entrada, podemos usar a estratégia de **divisão e conquista**. A ideia é organizar os dados de entrada, selecionar o elemento do meio como raiz e repetir esse processo para as sub-árvores da esqueda e direita.

Estratégia: Construção Balanceada com Lista Ordenada

1. **Ordenar os dados de entrada:**
   - Se os dados não estiverem ordenados, ordene-os. Isso garante que a árvore seja uma *árvore binária de busca*.
2. **Escolher o elemento do meio como raiz:**
   - Divida a lista ao meio. O elemento central vai ser a raiz da árvore ou subárvore.
3. **Repetir de forma recursiva:**
   - Para os elementos à esquerda do meio, crie a subárvore esquerda.
   - Para os elementos à direita do meio, crie a subárvore direita.
4. **Continuar até que a lista esteja vazia:**
   - A recursão terminará quando não houverem mais elementos na lista.

Exemplo: Construção da Árvore

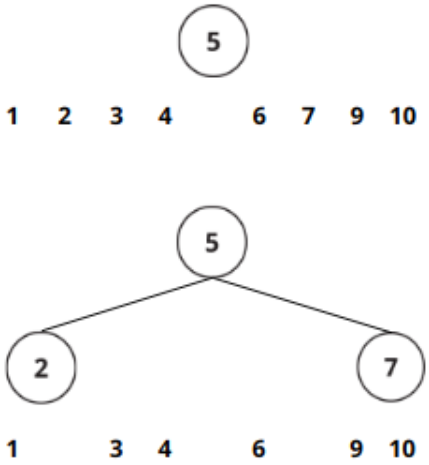
Considere os seguintes dados iniciais:

$$10,\ 2,\ 5,\ 3,\ 9,\ 7,\ 1,\ 4,\ 6$$

Primeiro, ordenaremos os valores.

$$1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 9,\ 10$$

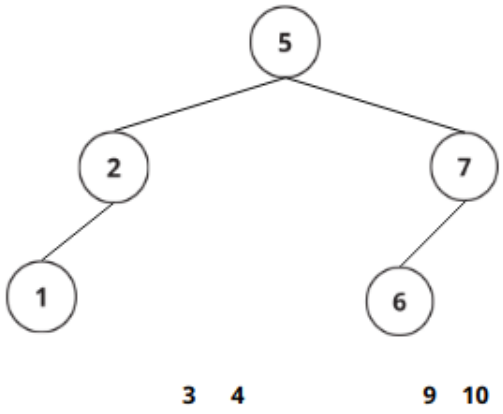
1. O nó **5** é escolhido como raiz da árvore, pois é o elemento central da lista.
2. A subárvore esquerda contém os números menores que 5 (**1, 2, 3, 4**) e a da direita contém os números maiores que 5 (**6, 7, 9, 10**)



- O elemento central (**2**) é escolhido como a raiz da subárvore esquerda.
- O elemento central (**7**) é escolhido como a raiz da subárvore direita.

3. Repetimos o processo a partir dos nós 2 e 7

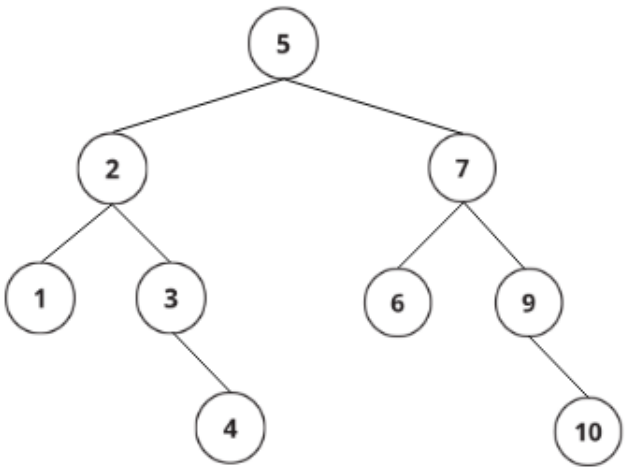
- A subárvore esquerda de 2 contém os números menores que 2 (**1**) e a da direita contém os números maiores que 2 (**3, 4**)
- (**1**) é escolhido como raíz da árvore esquerda de 2
- A subárvore esquerda de 7 contém os números menores que 7 (**6**) e a da direita contém os números maiores que 7 (**9, 10**)
- (**6**) é escolhido como raíz da árvore esquerda de 7



4. Finalizamos a árvore

- O elemento central (**3**) da árvore direita de 2 é escolhido como a raiz da subárvore direita.
- Como (**4**) é maior que 3, ele será a raiz da subárvore direita de 3.
- O elemento central (**9**) da árvore direita de 7 é escolhido como a raiz da subárvore direita.

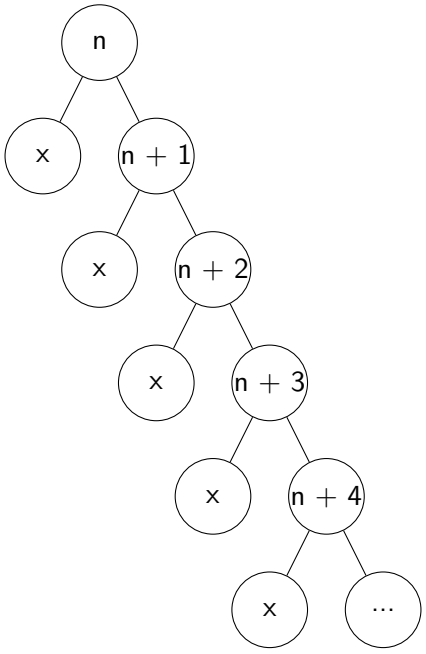
- Como (**10**) é maior que 9, ele será a raiz da subárvore direita de 9.



Capítulo 3

Árvores autobalanceáveis

Conforme mostrado anteriormente, são muitas as vantagens do uso de uma árvore binária, entretanto, imagine um seguinte cenário: Após definida uma raiz de valor **n**, são adicionados valores maiores que **n** e maiores entre si. A situação poderia ser representada conforme abaixo.



Perceba que esse caso é o pior cenário, e que ele nada difere de uma lista ligada. Tendo como complexidade de busca e inserção **O(n)**, perdendo assim, todas as vantagens em relação a estruturas mais simples.

Também vale ressaltar que apesar de se tratar do pior caso, não é tão remoto quanto parece, basta que um galho da árvore tenha um pouco mais elementos do que o seu irmão de nó que é possível observar semelhança com a situação descrita anteriormente. Dada essa situação, árvores com elementos mal distribuídos entre os galhos são ditas **desbalanceadas**, podendo formalizar a definição da seguinte maneira:

$$(\forall t \in \text{BinTree } \alpha) \quad [\ |\ \text{height(t.left)} - \text{height(t.right)}\ | \leq 1 \implies \text{t é balanceada}]$$

Por tanto, uma vez apresentado o conceito que fundamenta as **BST**'s(Árvores Binárias de Busca), a seguir veremos sua implementação através da Árvore AVL e a Árvore Rubro-Negra.

3.1 Árvore AVL

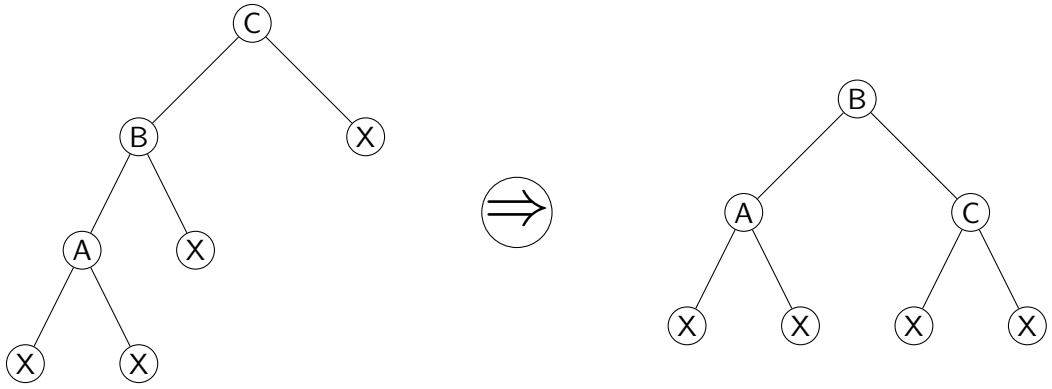
Esta se trata de uma implementação balanceada de uma BST. Sua primeira menção é datada no ano de 1962, no artigo **An algorithm for organization of information**, dos autores Georgy Maximovich **A**delson-**V**elsky e Evgenii Mikhailovich **L**andis, cujas iniciais foram eternizadas na nomeação do conceito. Sendo assim, sua intenção minimizar a complexidade da operação de busca, enquanto aumenta um pouco o custo da inserção e remoção, e isso acontece através do conceito de rotações.

E sabendo do conceito de balanceamento apresentado anteriormente, o critério usado para aplicá-lo é o chamado **Fato de Balanceamento**, que pode ser formalizado da seguinte forma

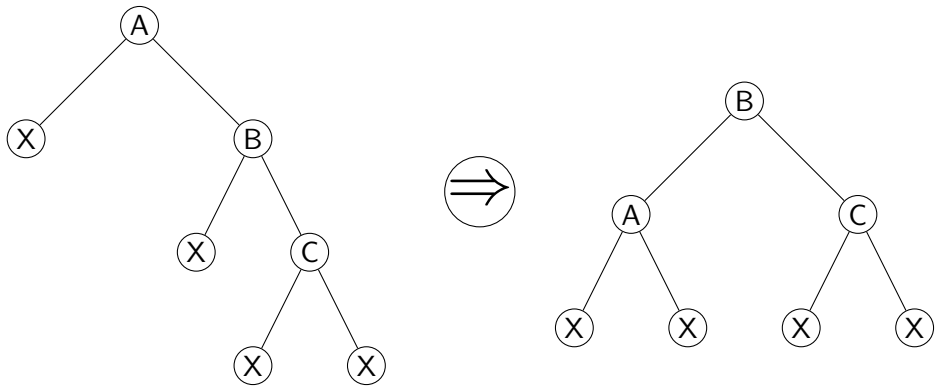
$$(\forall t \in \text{AVLTree } \alpha) \quad [\,\text{BF} = \text{height(t.left)} - \text{height(t.right)}\,]$$

Sendo assim, seja **n** um nó de uma Árvore AVL, os quatro casos em que o **Fator de balanceamento** é usado como critério para balancear a árvore são os seguintes:

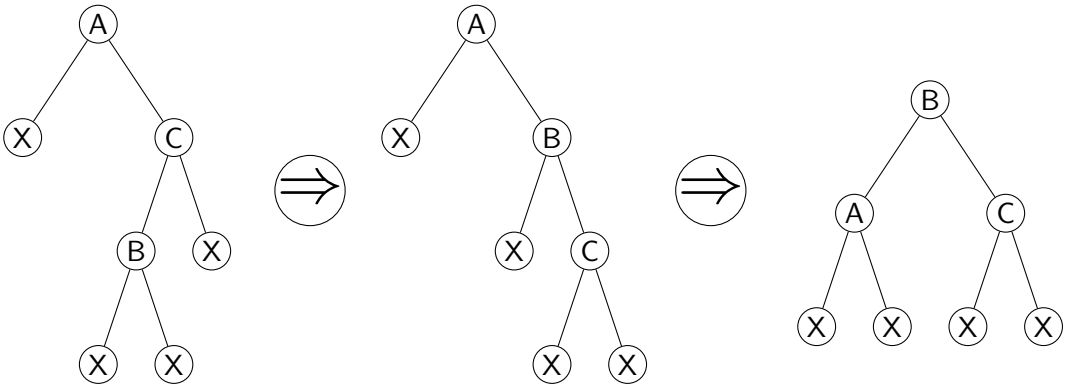
- **Rotação direita**
  Quando temos que **BF(n) $<$ -1 e height(n.right.left) $<$ height(n.right.right)**, o nó está muito pesado na esquerda, sendo assim é necessária uma que o nó onde o desbalanceamento inicia seja rotacionado à direita. Ou seja, visualmente algo parecido com o exemplo abaixo



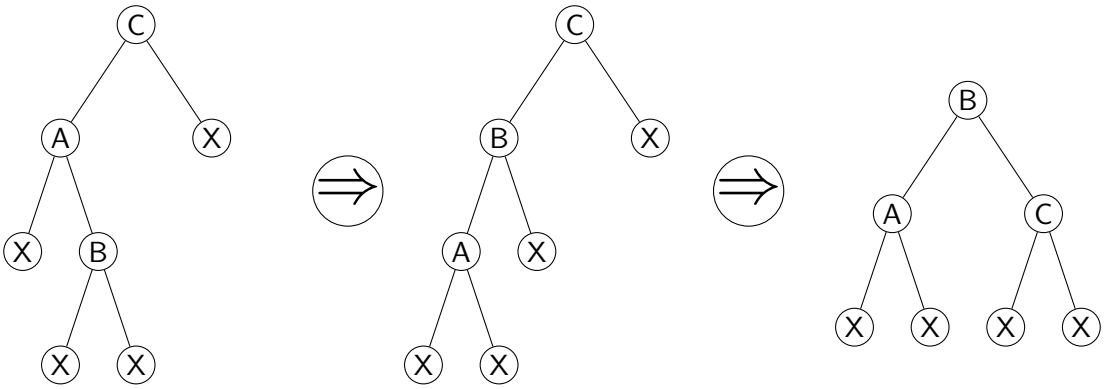
- **Rotação esquerda**
  Quando temos que **BF(n) $>$ 1 e height(n.left.right) $<$ height(n.left.left)**, há uma sobrecarga na direita do nó, portanto se faz necessária uma rotação à esquerda.



- **Rotação direita-esquerda**
  Quando temos que **BF(n)** $<$ **-1**, o nó à esquerda está muito pesado, entretanto não basta uma uma rotação direita, mas sim uma rotação direita seguida por outra à esquerda.



- **Rotação esquerda-direita**
  Quando temos que **BF(n)** $>$ **1**, há uma sobrecarga na direita do nó, entretanto uma rotação esquerda não é suficiente, mas sim uma rotação esquerda seguida por outra à direita.



3.1.1 Implementação

No que diz respeito à implementação, esta foi feita utilizando Haskell, uma linguagem de programação funcional. A estrutura de dado foi implementada da conforme o código abaixo.

```
data AVLTree a
  = Node
      { key :: a
      , left :: AVLTree a
      , right :: AVLTree a
      }
  | Nil
```

Sendo assim, é possível criar instâncias de `AVLTree a`, onde a é um tipo genérico. Podendo ser através do construtor `Nil` que representa uma folha vazia, indicando o fim daquele determinado galho. Bem como também pode ser usando o `Node` que se trata de um nó, que deve receber 3 argumentos: O valor de tipo a que ele vai armazenar e suas sub-árvores esquerda e direita.

A seguir será elucidado como foram implementadas as funções de Inserção, Remoção e Busca.

3.1.2 Busca

Esta se trata da operação que o funcionamento da Árvore AVL acaba privilegiando, seu comportamento é identico à de uma implementação básica de uma BST. Entretanto o balanceamento é um fator importantíssimo para que a função opere com o máximo desempenho, apesar de ser idêntica à uma BST padrão. E isso acontece pois **apenas** as operações de inserção e remoção podem balancear a árvore, sendo assim, esta já está em seu melhor cenário de distribuição de elementos quando uma busca é iniciada. A operação foi implementada conforme a seguir.

```
search :: (Ord a) => AVLTree a -> a -> Maybe [Direction]
search Nil _ = Nothing
search (Node k l r) v
    | v == k = Just []
    | v <= k = fmap (L :) (search l v)
    | otherwise = fmap (R :) (search r v)
```

Esta implementação tem algumas particularidades, a primeira delas sendo que ao invés de retornar apenas um booleano correspondente à existência de determinado elemento naquela instância de de `AVLTree a`, esta retorna a lista de direções que leva até ele.

Também note que ela utiliza o `Maybe a`, que é um tipo genérico que tem dois construtores: O `Nothing` e o `Just x`. Em que o `Nothing` é uma forma segura de dizer que não há correspondente ao que se procura, semelhante à uma exceção. Já o `Just x` é uma mera forma de encapsular um retorno bem sucedido da operação.

Por fim, também é usado o conceito de `Typeclass` através do `Ord a`, sendo esse um conceito dentro da Haskell para tipos que atendem a determinadas especificações. Em especial a `Ord a` se trata de uma "família"de tipos cujos valores são ordenáveis, o que torna possível a operação $>$, que orienta as direções da busca. Em síntese, já que a Árvore AVL é extremamente dependente de comparações ordenáveis, não faz sentido que a seja um tipo como Bool, por exemplo.

3.1.3 Inserção

Se tratando a Árvore AVL, a inserção está longe de ser uma das suas operações mais eficientes, e isso acontece pois ela é responsável por balancear a árvore pós-inserção. Ou seja, podemos dividir a inserção em duas partes: Inserir de forma bruta e balancear a árvore uma vez feita a inserção. Sendo assim, o primeiro passo é muito bem retratado pela função `insert'` abaixo.

```
insert' :: (Ord a) => a -> AVLTree a -> AVLTree a
insert' v Nil = Node v Nil Nil
insert' v (Node v' l r)
    | v' <= v = Node v' l (insert' v r)
    | otherwise = Node v' (insert' v l) r
```

Em síntese, ela se comporta praticamente igual à busca, com a ressalva de que ela só para quando chega em uma folha vazia, e faz a sobrescrita do `Nil` para que este vire um `Node (valor Nil Nil)`.

Entretanto, após isso é necessário se certificar que se um balanceamento se faz necessário. Nesse caso podemos dividir o ato de balancear em duas partes: Verificar qual o nó que causa o desbalanceamento(caso ele exista) e rotacioná-lo. Essas funções são respectivamente atribuídas às funções `findUnbalancedNode` e `rotateAt`

```haskell
insert :: (Ord a) => a -> AVLTree a -> AVLTree a
insert v t =
    let treeAfterInsertion = insert' v t
     in rotateAt treeAfterInsertion (findUnbalancedNode treeAfterInsertion
    )

findUnbalancedNode :: (Ord a) => AVLTree a -> Maybe [Direction]
findUnbalancedNode Nil = Nothing
findUnbalancedNode t@(Node v l r)
    | balanced t = Nothing
    | not (balanced t) && balanced l && balanced r = Just []
    | balanced l = fmap (R :) (findUnbalancedNode r)
    | otherwise = fmap (L :) (findUnbalancedNode l)

rotateAt :: (Ord a) => AVLTree a -> Maybe [Direction] -> AVLTree a
rotateAt t Nothing = t
rotateAt (Node v l r) (Just (d : ds)) = case d of
    R -> Node v l (rotateAt r (Just ds))
    L -> Node v (rotateAt l (Just ds)) r
rotateAt t (Just []) = rotateNode t
```

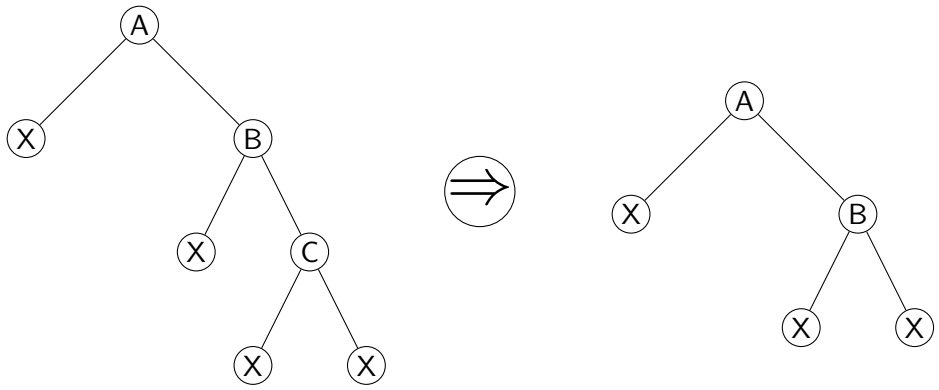
Ademais, também são usadas as funções auxiliares abaixo. Sendo a `rotateNode` responsável por fazer a rotação cabível, uma vez que há a garantia que ela está no nó que causa do desbalanceamento. Bem como as demais funções são as rotações simples e duplas.

```haskell
rotateNode :: (Ord a) => AVLTree a -> AVLTree a
rotateNode Nil = Nil
rotateNode t@(Node _ l r)
    | balanceFactor < -1 && height (left r) < height (right r)
        =
        trace "Rotation R" rotateR t
    | balanceFactor > 1 && height (right l) < height (left l)
        =
        trace "Rotation L" rotateL t
    | balanceFactor < -1 && height (left r) > height (right r)
        =
        trace "Rotation RL" rotateRL t
    | balanceFactor > 1 && height (right l) > height (left l)
        =
        trace "Rotation LR" rotateLR t
  where
    balanceFactor = height l - height r

rotateR :: (Ord a) => AVLTree a -> AVLTree a
rotateR (Node e l (Node er lr rr)) = Node er (Node e l lr) rr

rotateL :: (Ord a) => AVLTree a -> AVLTree a
rotateL (Node e (Node el ll rl) r) = Node el ll (Node e rl r)

rotateLR :: (Ord a) => AVLTree a -> AVLTree a
rotateLR (Node e (Node el ll (Node elr lrl lrr)) r) = Node elr (Node el ll
    lrl) (Node e lrr r)

rotateRL :: (Ord a) => AVLTree a -> AVLTree a
rotateRL (Node e l (Node er (Node erl rll rlr) rr)) = Node erl (Node e l
    rll) (Node er rlr rr)
```

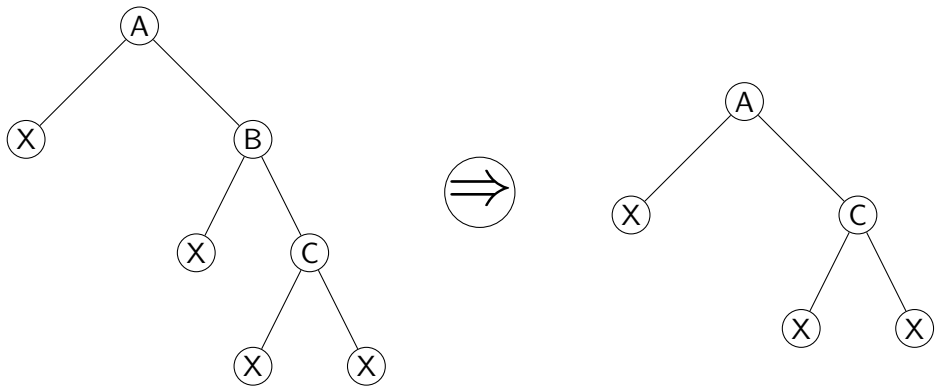
3.1.4 Remoção

Por fim, a remoção talvez se trate da complexa entre as operações. Isso pois seu balanceamento envolve mais elementos do que meramente rotacionar a árvore. Entretanto, podemos separá-la nos três seguintes casos

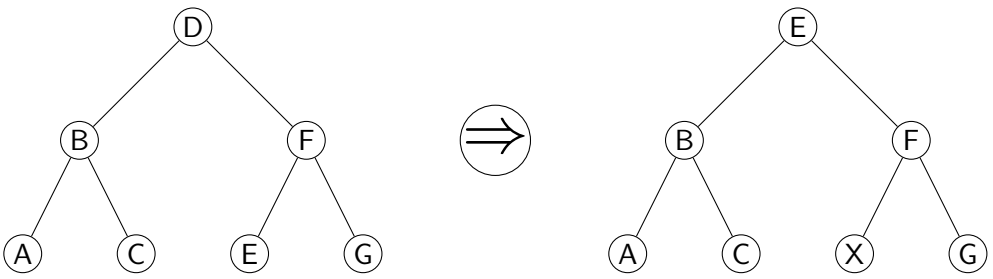
- **Remoção de um nó sem filhos**
  Esse caso é trivial, basta substituir um `Node` que contenha o valor a ser removido por um `Nil`. Suponha que o **C** é o elemento a ser removido.



- **Remoção de um nó com um filho**
  Nesse caso, independente se o nó tem uma sub-árvore esquerda ou direita, é feita uma ligação direta entre seu filho e seu avô. Suponha que **B** é o elemento a ser removido.



- **Remoção da um nó com dois filhos**
  Nesse caso precisaremos substituir o valor do nó pelo valor de uma folha, podendo ser tanto o menor valor da sub-árvore direita ou o maior da sub-árvore esquerda, geralmente usa-se o primeiro método. Suponha que o **D** deve ser removido.



Apresentados os casos mais triviais, vale ressaltar que as implementações reais também envolvem o balanceamento pós-remoção, conforme feito no código abaixo. Nele fora usadas, além das funções já apresentadas anteriormente, as funções `remove`, `remove'` e `removeMin`. Que respectivamente fazem o balanceamento pós-remoção, a remoção de fato e a busca e remoção do menor elemento da sub-árvore direita.

```
remove :: (Ord a) => a -> AVLTree a -> AVLTree a
remove v t =
    let treeAfterRemotion = remove' v t
     in rotateAt treeAfterRemotion (findUnbalancedNode treeAfterRemotion)

remove' :: (Ord a) => a -> AVLTree a -> AVLTree a
remove' _ Nil = Nil
remove' n t@(Node v Nil Nil)
    | n == v = Nil
    | otherwise = trace "Element does not exist" t
remove' n (Node v l Nil)
    | n == v = l
    | n < v = Node v (remove' n l) Nil
    | otherwise = trace "Element does not exist" (Node v l Nil)
remove' n (Node v Nil r)
    | n == v = r
    | n > v = Node v Nil (remove' n r)
    | otherwise = trace "Element does not exist" (Node v Nil r)
remove' n (Node v l r)
    | n < v = Node v (remove' n l) r
    | n > v = Node v l (remove' n r)
    | n == v =
        let (successor, newRight) = removeMin r
         in Node successor l newRight

removeMin :: (Ord a) => AVLTree a -> (a, AVLTree a)
removeMin (Node v Nil r) = (v, r)
removeMin (Node v l r) =
    let (minValue, newLeft) = removeMin l
     in (minValue, Node v newLeft r)
```

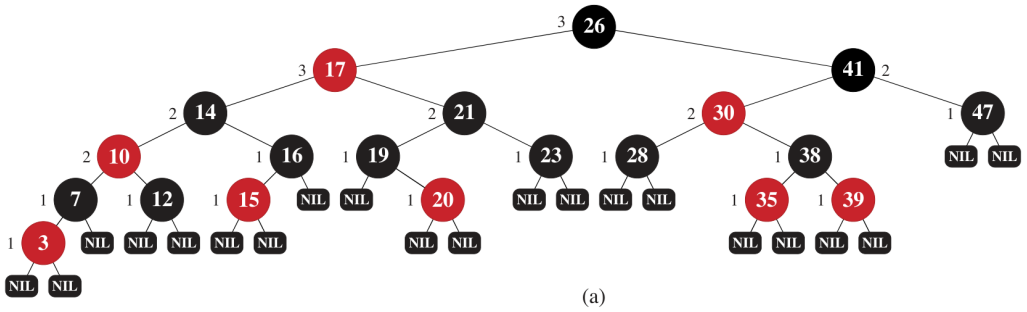
Em súmula, observamos que a Árvore AVL é uma implementação que privilegia a operação de busca pois terceiriza o balanceamento para as operações de inserção e remoção. Isso implica em um maior custo computacional para executar as últimas duas, entretanto nada tão significativo, pois, no fim, ambas as três tem complexidade **O(log n)**.

3.2 Árvore Rubro Negra

Como uma alternativa mais relaxada à Arvore AVL, a Arvore Rubro Negra surgiu do trabalho "A Dichromatic Framework for Balanced Trees" de Leonidas J. e Robert Sedgewick, e a escolha das cores rubro e negra foram influenciados pela disponibilidade de canetas dessa cor na época e qualidade de impressão[1].

Para atender a necessidade de se auto balancear, a Rubro-Negra assume algumas regras e características:

1. Todo nó é vermelho (rubro) ou preto (negro).
2. O nó raiz é preto.
3. Toda folha (`Nil`) é preta.
4. Se um nó é vermelho, seus dois filhos são pretos.
5. Cada caminho da raiz até um nó `Nil` tem a mesma quantidade de nós pretos.



(a)

Figure 3.1: Imagem retirada de **?**

Devida sua natureza, uma árvore rubro negra de n elementos tem *profundidade*[2] de ao máximo $2\lfloor \log(n+1) \rfloor$, garantindo que as operações de INSERÇÃO, BUSCA, REMOÇÃO tenham complexidade $O(\log n)$ no pior caso.

3.2.1 Datatype

Para sua implementação, usaremos a linguagem de programação Haskell, assumindo a seguinte estrutura.

```haskell
data RBTree a
  = Nil -- black leaf
  | Node Color (RBTree a) a (RBTree a)
  deriving (Show)
```

Listing 3.1: Árvore Rubro-Negra

[1] Contribuidores da Wikipedia (2024)
[2] A distância entre um nó qualquer e a raiz

Um elemento do tipo `RedBlackTree a`, onde 'a' é genérico, tem dois construtores: construtor `Nil`, que implicitamente tem cor preta, ou a função `Node` aplicada em quatro argumentos, a cor (`Color`), a árvore esquerda (`RBTree a`), o valor dentro do nó (`a`) e a árvore direita (`RBTree a`),

Neste documento implementaremos a INSERÇÃO[3] e REMOÇÃO[4], visto que as implementações da BUSCA e ROTAÇÃO são análogas a árvore AVL e binária.

3.2.2 Implementação da inserção

Inicialmente, o algoritmo será similar a inserção na Árvore binária convencional, mas uma vez inserido o valor em algum nó, precisamos decidir qual será sua cor e garantir que as regras da Rubro-negra sejam atendidas.

Em relação a cor, insertamos o nó como a cor vermelha, pois minimiza a quantidade de violações. Para garantir que nenhum nó vermelho seja raiz, simplesmente pintamos o último nó de preto. Teremos, então:

```
1 insert :: (Ord a) => a -> RBTree a -> RBTree a
2 insert x tree = blacken (ins tree)
```

Listing 3.2: Função principal

Naturalmente, os valores que uma Rubro-negra precisa armazenar devem ser ordenáveis (restrição imposta por `Ord a`), ademais, no primeiro momento, usaremos duas funções auxiliares, a `blacken` e `ins`. A `blacken` será responsável por corrigir a violação anteriormente citada e a `ins` realizará de fato a inserção, em conjunto com as outras correções.

Começando pela mais simples:

```
1 blacken Nil = Nil
2 blacken (Node _ left r right) = Node Black left r right
```

Na `ins` prosseguimos com uma inserção tradicional em árvore binária, apenas com algumas ressalvas.

```
1 ins Nil = Node Red Nil x Nil
2 ins tree@(Node color left r right)
3   | x < r = balance color (ins left) r right
4   | x > r = balance color left r (ins right)
5   | otherwise = tree
```

Usaremos mais uma função auxiliar chamada `balance`, que terá a mesma tipagem do construtor `Node` e será responsável por corrigir as possíveis violações locais causadas pela inserção do novo nó vermelho, as quais:

- **Violação Esquerda-Esquerda:** Dois vermelhos na sub-árvore esquerda.
  Uma vez chegando no caso base e inserindo o valor `x`, essa violação ocorrerá da seguinte forma:

[3] Versão de **?**
[4] Versão de **?**

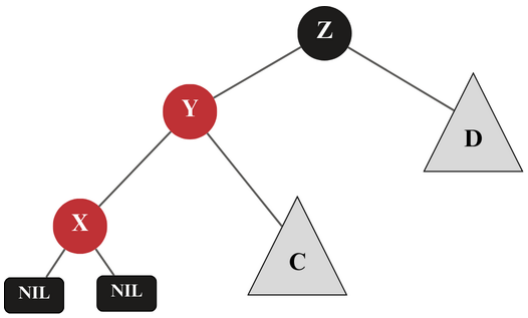


Figure 3.2

E para resolver localmente essa violação, rotacionamos e alteramos as cores da árvore:

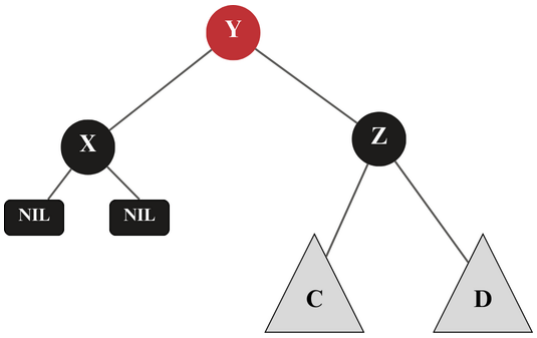


Figure 3.3

Entretanto, depois de balancear essa sub-árvore, a raiz vermelha pode ser filho de algum outro nó vermelho na árvore maior. Mas, como a `balance` é chamada em cada etapa do `ins`, logo, ela vai continuar corrigindo as violações nas sub-árvores até a topo da árvore. Então generalizamos esse caso:

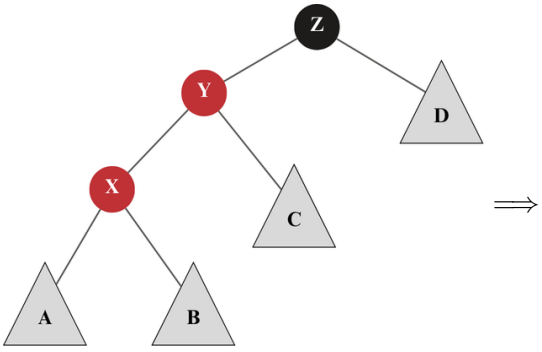


Figure 3.4

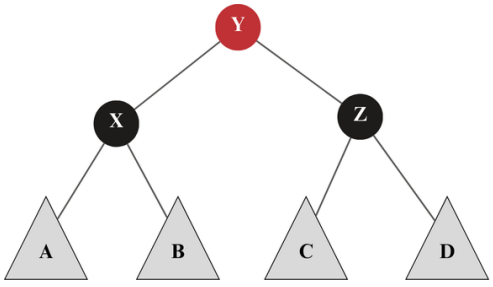


Figure 3.5

- **Violação Esquerda-Direita:** Dois vermelhos, um no filho esquerdo e outro no filho direito do filho esquerdo.

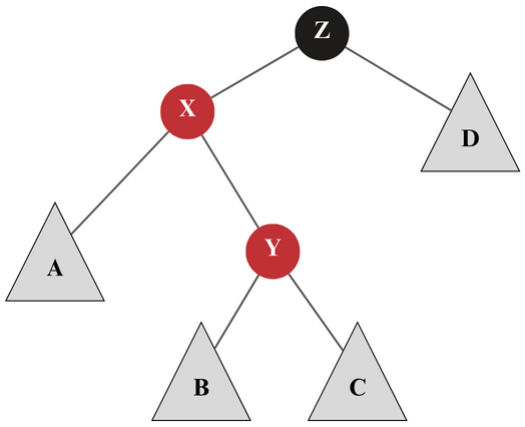


Figure 3.6

Para resolver essa violação, também vamos reconfigurar como foi feito na Figura 3.5, assim como todas as restantes.

- **Violação Direita-Direita:** Dois vermelhos, na sub-árvore direita.

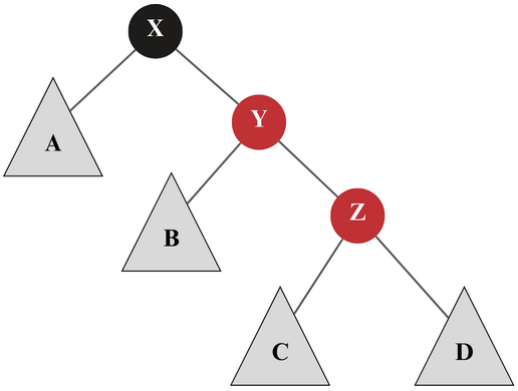


Figure 3.7

- **Violação Direita-Esquerda** Dois vermelhos, um no filho direito e outro no filho esquerdo do filho direito.

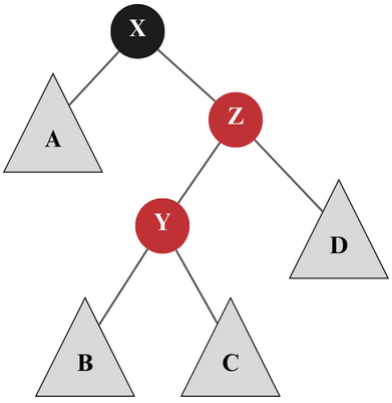


Figure 3.8

Uma vez estabelecidos os casos a, implementação desse algoritmo se torna bem mais simples do que versões iterativas, visto que todos os casos produzem o mesmo resultado.

```
-- Esquerda-Esquerda
balance Black (Node Red (Node Red at x bt) y ct) z dt =
  Node Red (Node Black at x bt) y (Node Black ct z dt)
-- Esquerda-Direita
balance Black (Node Red at x (Node Red bt y ct)) z dt =
  Node Red (Node Black at x bt) y (Node Black ct z dt)
-- Direita-Direita
balance Black at x (Node Red (Node Red bt y ct) z dt) =
  Node Red (Node Black at x bt) y (Node Black ct z dt)
-- Direita-Esquerda
balance Black at x (Node Red bt y (Node Red ct z dt)) =
  Node Red (Node Black at x bt) y (Node Black ct z dt)
-- Padrao
balance color left key right = Node color left key right
```

Teste da implementação:

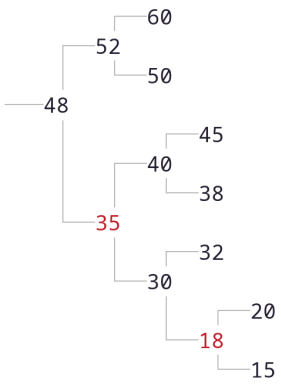
Para assertar o funcionamento da Inserção na Árvore Rubro-negra, criaremos uma árvore do zero com o uso desse algoritmo:

```
fromList :: (Ord a) => RedBlackTree a -> [a] -> RedBlackTree a
fromList = foldr insert
```

Usando os elementos dessa lista:

```
[15, 18, 20, 35, 32, 38, 30, 40, 32, 45, 48, 52, 60, 50]
```

```
λ> prettyPrint $ fromList Nil [15, 18, 20, 35, 32, 38, 30, 40, 32, 45, 48, 52, 60, 50]
```

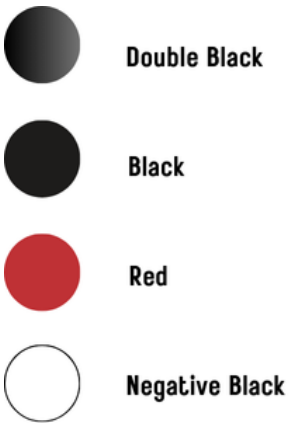


3.2.3 Implementação da remoção

Assim como na inserção, a remoção na Árvore Rubro-negra tem diversos casos, muitos dos quais são simples de resolver e até idênticos aos da inserção. Entretanto, a remoção de um nó preto sem filho requer uma atenção especial, pois altera a altura da árvore. O truque para lidar com esse caso é adicionar duas cores temporárias 'BBlack' e 'NBlack', e quebrar-lo em três partes: **Remoção**, **Borbulhamento** e **Balanceamento**.

1. Por adicionar a cor 'BBlack', o caso se reduz a mudar o nó para uma folha 'NNil'. Um nó com cor 'BBlack', conta duas vezes a altura de um nó de cor preta, o que permite que a regra seja preservada.
2. O borbulhamento visa eliminar os 'BBlack's criados pela remoção. As vezes, é possível eliminá-los recolorindo seu pai e irmão. Se não for possível, o 'BBlack' é "borbulhado" para seu pai. Para fazer isso, talvez seja necessário recolorir o irmão do nó para 'NBlack', caso seja vermelho.
3. Por fim, o balanceamento elimina os 'BBlack's e 'NBlack's ao mesmo tempo. Generalizamos e usamos a mesma função balance da inserção com dois novos casos.

Representaremos as cores da árvore graficamente como:



E no código como:

```haskell
data Color
  = Red
  | Black
  | BBlack -- double black
  | NBlack -- negative black
  deriving (Show)
```

Para acomodar adição de novas cores, alteramos o datatype da rubro-negra adicionando o construtor NNil, que representa uma folha nula 'BBlack'.

```haskell
data RBTree a
  = Nil -- black leaf
  | NNil -- double black leaf
  | Node Color (RBTree a) a (RBTree a)
  deriving (Show)
```

Além disso, para simplificar o algoritmo de remoção, usaremos "aritmética" de cores, isto é escurecer e "emvermelhar"uma cor:

```haskell
blacker Red = Black
blacker Black = BBlack
blacker NBlack = Red
blacker BBlack = error "blacker: too black, received Double Black"

redder NBlack = error "redder: not black enough, received Negative Black"
redder Red = NBlack
redder Black = Red
redder BBlack = Black
```

Primeira parte: Remoção

Inicialmente começaremos como uma remoção tradicional em um árvore binária, isto é:

```
remove :: (Ord a, Show a) => a -> RBTree a -> RBTree a
remove x tree = blacken (rem tree)
 where
  rem Nil = Nil
  rem tree@(Node color left r right)
    | x < r = bubble color (rem left) r right
    | x > r = bubble color left r (rem right)
    | otherwise = remove' tree
```

Aqui a função `rem` cumpre papel idêntico ao `ins` do algoritmo de inserção, entretanto, ao invés de usar diretamente o `balance`, invocaremos duas partes do nosso algoritmo de remoção, o "Borbulhamento" (`bubble`) e a remoção em si (`remove'`).

Começando pela remoção, seus casos se agrupam de acordo com quantos filhos o nó alvo tem. Se o nó alvo tem duas sub-árvores, a remoção vai reduzir para o caso em que tem no máximo uma sub-árvore. Para que tal aconteça, encontramos o máximo da sua sub-árvore esquerda, removemos esse nó, e colocamos seu valor no nó a ser removido.

Por exemplo, tome o nó azul como alvo, e o verde como o mais a direita da sub-árvore esquerda do azul.

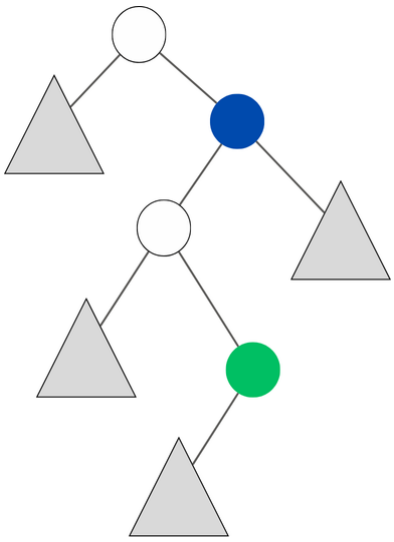


Figure 3.9

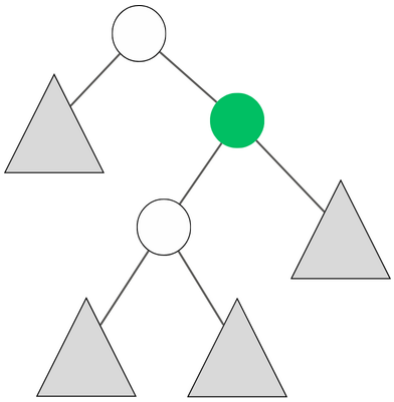


Figure 3.10

Se o nó alvo for uma folha, ou seja, ter filhos `Nil`, a remoção é imediata.

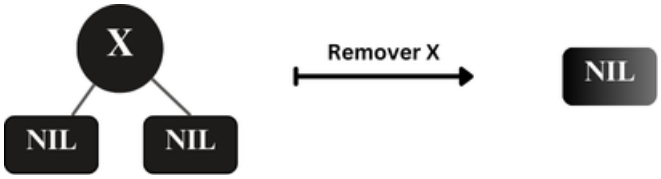


Figure 3.11: Remoção de folha preta

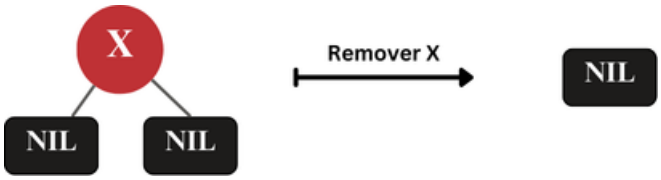


Figure 3.12: Remoção de folha vermelha

Se o nó alvo possuir apenas um filho, existe uma única possibilidade para a esquerda e direita:

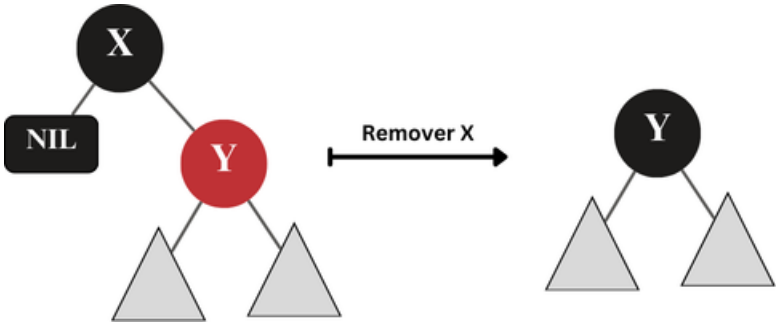


Figure 3.13: Remoção de folha vermelha

Em código, definiremos a função `remove'`, que lidará com esses casos da seguinte forma (em ordem contrária da que eu apresentei):

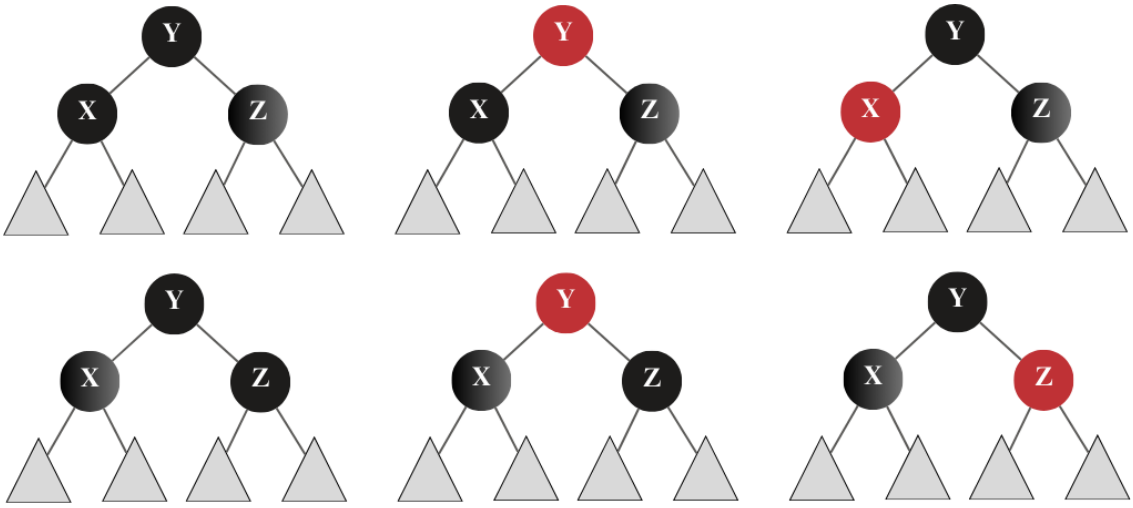
```
remove' Nil = Nil
remove' (Node Red Nil _ Nil) = Nil
remove' (Node Black Nil _ Nil) = NNil
remove' (Node Black Nil _ (Node Red at x bt)) = Node Black at x bt
remove' (Node Black (Node Red at x bt) _ Nil) = Node Black at x bt
remove' (Node color left x right) =
  bubble color (removeMax left) (max left) right
```

A função `removeMax` remove o elemento mais a direita e a função `max` vira esse elemento.

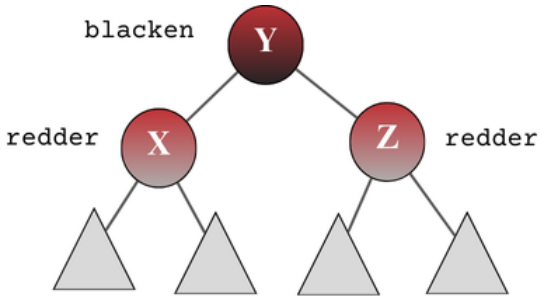
```
max Nil = error "max: Nil tree, there's no maximum"
max (Node _ _ x Nil) = x
max (Node _ _ x right) = max right

removeMax Nil = error "removeMax: Nil tree, there's no maximum"
removeMax tree@(Node _ _ _ Nil) = remove' tree
removeMax tree@(Node color left x right) =
  bubble color left x (removeMax right)
```

Segunda parte: Borbulhamento

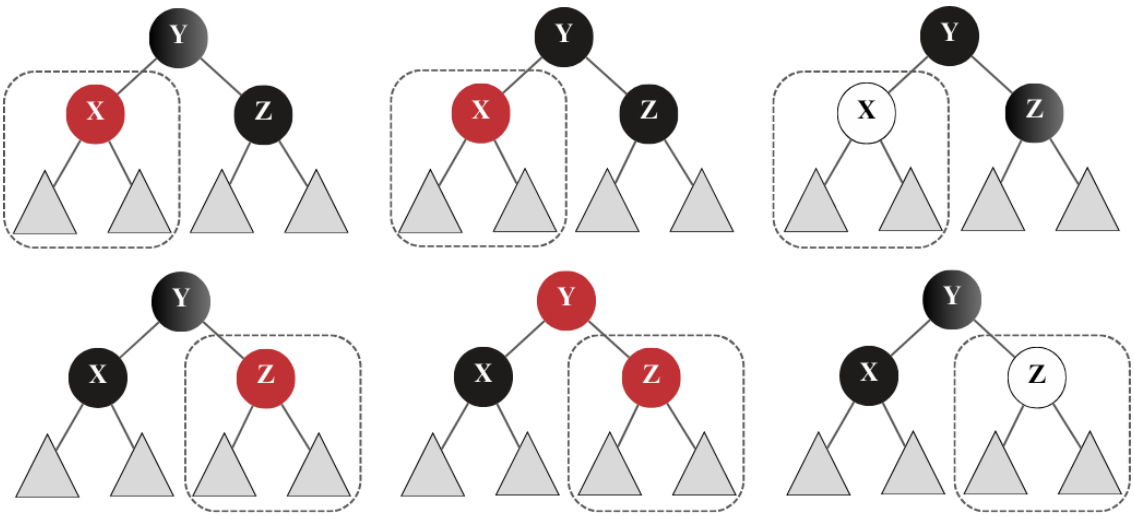
O borbulhamento move `BBlack`'s para os pais, ou elimina-os completamente se possível, no total, existem 6 casos possíveis onde `BBlack`'s aparecem:



Em todos os casos, a ação necessária para mover o `BBlack` para cima é o mesmo: "Em-vermelhar" a cor dos filhos e "escurecer" a cor dos pais:



De forma que:



O pontilhado indica a necessidade de rebalanceamento, a causa é a introdução de uma relacionamento Vermelho-Vermelho ou NBlack/Vermelho entre pai e filho, cujo viola as regras

da rubro-negra.

Como toda ação é a mesma, o código da `bubble` também é relativamente simples:

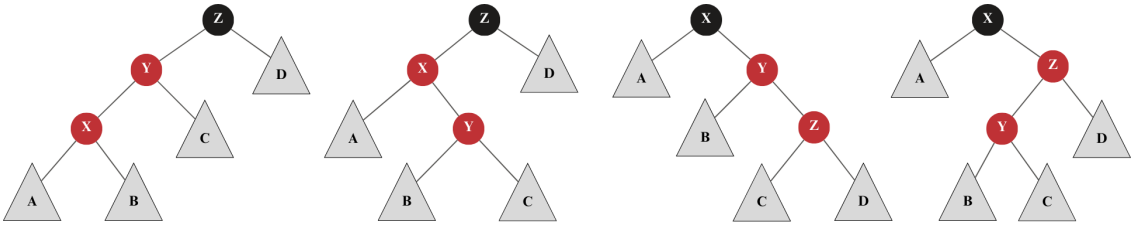
```
bubble :: Color -> RBTree a -> a -> RBTree a -> RBTree a
bubble color left x right
  | isBBlack left || isBBlack right = balance (blacker color) (redder' left) x (redder' right)
  | otherwise = balance color left x right
```

A função `isBBlack` apenas verifica se a árvore tem cor `BBlack`, e a `redder'` "opera" o vermelho com a árvore, isto é:

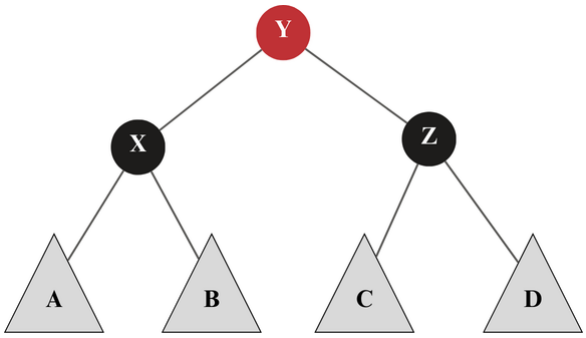
```
redder' NNil = Nil
redder' (Node color left x right) = Node (redder color) left x right
```

Terceira parte: Balanceamento

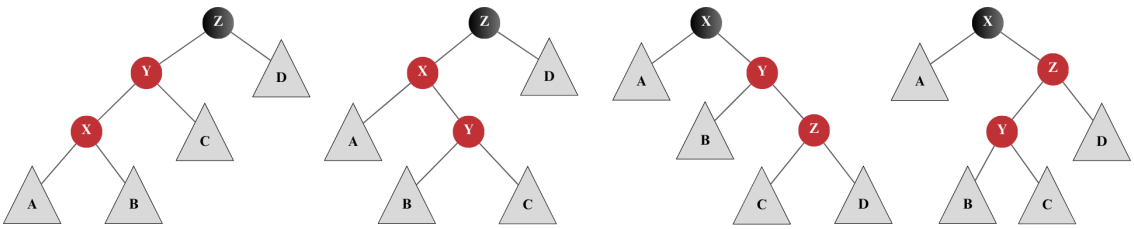
Aqui aproveitamos a `balance` implementada para a inserção, que consertava essas violações onde a raiz é preta:



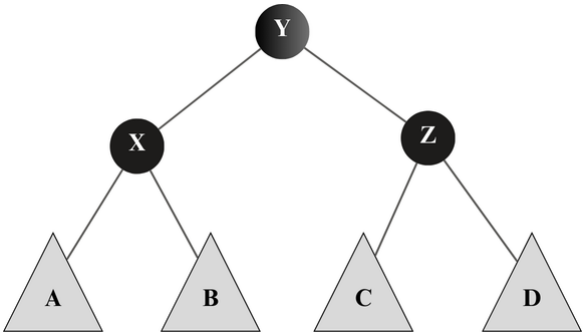
Tornando-as nessa árvore:



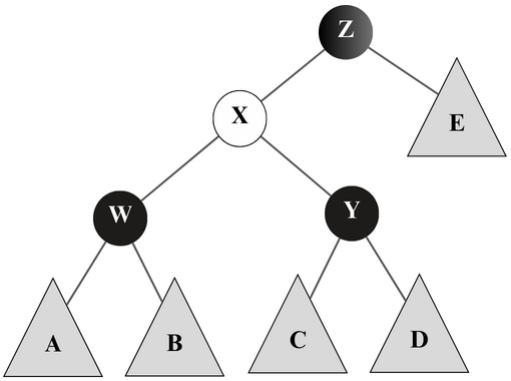
Os novos casos, vão lidar agora com as mesmas violações, mas quando a raiz é `BBlack`:



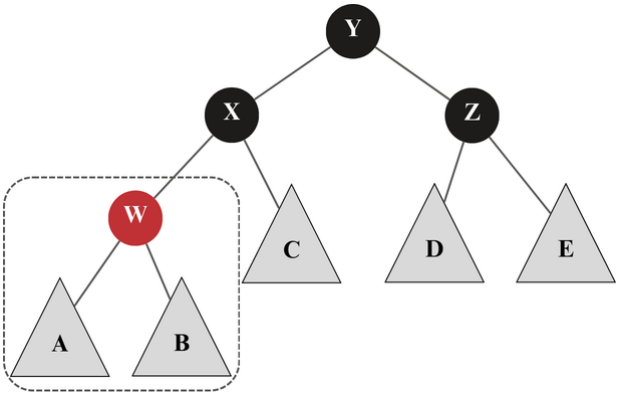
Transformando todos nessa árvore:



Entretanto, se um NBlack aparecer como resultado do Borbulhamento, como em:



Uma outra transformação é necessária:



Novamente, a linha pontilhada indica uma possível violação Vermelho-Vermelho, o que pode precisar de outro rebalanceamento e faz com que a `balance` precise ser recursiva. Mas, ela não vai se chamar mais de uma vez. Além disso, existe a versão espelhada dessa última operação. No código, adicionamos os seguintes casos à `balance`:

```
-- Violacoes Vermelho-Vermelho com raiz double black
balance BBlack (Node Red (Node Red at x bt) y ct) z dt =
  Node Black (Node Black at x bt) y (Node Black ct z dt)
balance BBlack (Node Red at x (Node Red bt y ct)) z dt =
  Node Black (Node Black at x bt) y (Node Black ct z dt)
balance BBlack at x (Node Red (Node Red bt y ct) z dt) =
  Node Black (Node Black at x bt) y (Node Black ct z dt)
balance BBlack at x (Node Red bt y (Node Red ct z dt)) =
  Node Black (Node Black at x bt) y (Node Black ct z dt)
-- Casos extra com negative black
```

```
11 balance BBlack at x (Node NBlack (Node Black bt y ct) z dt@(Node Black _ _
       _)) =
12   Node Black (Node Black at x bt) y (balance Black ct z (redden dt))
13 balance BBlack (Node NBlack at@(Node Black _ _ _) x (Node Black bt y ct))
       z dt =
14   Node Black (balance Black (redden at) x bt) y (Node Black ct z dt)
15 balance color at x bt = Node color at x bt
```

Assim, garantimos que todos `NBlacks`'s sejam removidos e a árvore se mantenha balanceada depois de ter um elemento removido.

Capítulo 4

Ambiente computacional

Todos os testes de performance nesse trabalho foram realizados no seguinte sistema:

Software	Sistema Operacional	Arch Linux x86_64
	Kernel	Linux 6.12.8-arch1-1
	Gerenciador de Janelas	Hyprland (Wayland)
	Terminal	Ghostty 1.0.1
	Compilador e REPL de Haskell	ghc/ghci 9.4.8
	Compilador de C++	clang 18.1.8
	Compilador de Rust	rustc 1.82.0
	Versão do Cargo	cargo 1.82.0
Hardware	CPU	AMD Ryzen 5 5500
	GPU	GeForce RTX 4060 Ti
	Driver da GPU	nvidia (proprietário) 565.77
	Memória RAM	31.24 GiB
	Armazenamento	SSD NVMe 2TB

Referências Bibliográficas

Adel'son-Vel'skii, G. M. and Landis, E. M. (1962), 'An algorithm for organization of information', *Doklady Akademii Nauk SSSR* **146**(2), 263–266.

Contribuidores da Wikipedia (2024), 'Red-black tree'.
URL: *https://en.wikipedia.org/wiki/Red-black*$_t$*ree*

Contribuidores da Wikipedia (2025), 'Avl tree'.
URL: *https://en.wikipedia.org/wiki/AVL*$_t$*ree*

Cormen, T. H., Leiserson, C. E., Rivest, R. L. and Stein, C. (2022), *Introduction to Algorithms*, 4th edn, The MIT Press, Cambridge, Massachusetts.

Germane, K. and Might, M. (2014), 'Deletion: The curse of the red-black tree', *Journal of Functional Programming* **24**(4), 423–433.

Might, M. (2022), 'The missing method: Deleting from okasaki's red-black trees'.
URL: *https://matt.might.net/articles/red-black-trees-deletion/*

Okasaki, C. (1999), *Purely Functional Data Structures*, Cambridge University Press, Cambridge, UK.