# EDD2 - Arvores

## 1. Árvores Binárias de Busca

> Árvores binárias de busca são projetadas para um acesso rápido à informação. Idealmente a árvore deve ser razoavelmente equilibrada e a sua altura será dada (no caso de estar completa) por h=log2 (n+1) O tempo de pesquisa tende a O(log2N) Porém, com sucessivas inserções de dados principalmente ordenados, ela pode se degenerar para O(n).

**Árvores completas**: São aquelas que minimizam o número de comparações efetuadas no pior caso para uma busca com chaves de probabilidades de ocorrências idênticas.

### Propriedades

- A subárvore esquerda de um nó sempre contém apenas nós com chaves menores que a chave desse mesmo nó;
- A subárvore direita de um nó sempre contém apenas nós com chaves maiores que a chave desse mesmo nó;
- As subárvores esquerda e direita também devem ser árvores binárias de busca.

### Aplicações (alguns exemplos)

- Usada para implementar fila de prioridade dupla;
- Indexing de Databases;

### Travessia da árvore

- **pré-ordem** (os filhos de um nó são processados após o nó): Cima para baixo
- **pós-ordem** (os filhos são processados antes do nó): Baixo para cima
- **em-ordem:** em que se processa o filho à esquerda, o nó, e finalmente o filho à direita.

# Operações

## Inserção

- O primeiro elemento inserido assumirá o papel de raiz da árvore;
- Todo novo elemento entrará na árvore como uma folha;
- Se o elemento for menor ou igual à raiz será inserido no ramo da esquerda. Caso contrário, no ramo da direita (para árvores decrescentes inverte-se a regra).

> Novas chaves são sempre inseridas como folhas para podermos manter a propriedade da árvore binária de busca. O processo de inserção se inicia através de uma busca que começa na raiz da árvore. A partir dali, procuramos por uma chave até encontrarmos um nó folha. Quando o nó folha é encontrado, adicionamos o novo nó como seu filho.

💡 **Complexidade de Inserção através de recursão:**
**Pior caso:** O pior caso de inserção é O(h), sendo h a altura da árvore.
- Nesse caso, o que acontece é que precisamos percorrer a árvore desde sua raiz até o nó folha mais distante dela. Se a árvore for muito assimétrica, a altura dela acaba se tornando n e, consequentemente a complexidade de inserção se torna O(n).

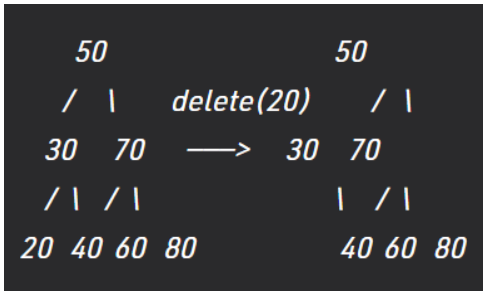
**Complexidade de Inserção através de iteração:**
A complexidade se torna O(log n), pois precisa-se iterar pela árvore de cima a baixo para que possamos inserir o novo nó.

## Remoção / Deleção

Considerando que podemos remover qualquer elemento de uma árvore, podem ocorrer as seguintes situações:

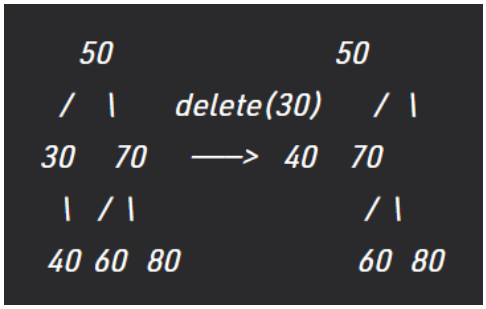
**1. O Elemento a ser removido é um nó folha (sem filhos à esquerda e à direita)**

Basta remover o nó da árvore:



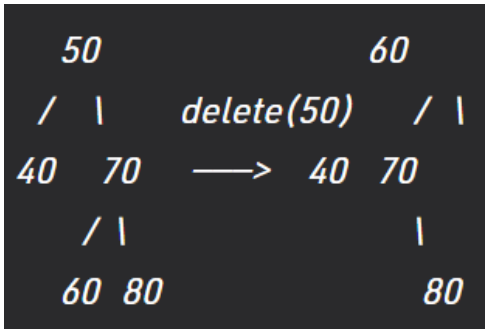

**2. O Elemento a ser removido possui apenas um filho (à direita ou à esquerda)**

Só copiar o filho do nó e remover o pai.




**3. O Elemento a ser removido possui dois filhos**

- Aqui precisamos encontrar o sucessor mais próximo do nó a ser removido;
- Depois, movemos os filhos desse nó e do nó que foi removido para o sucessor



💡 Por que árvores de busca binária são evitadas às vezes?
*Quando essa busca acontece no pior dos casos (árvore degenerada), iremos percorrer todos os nós de um lado, apenas para depois desempilhar os valores na pilha de recursão e retornar nulo quando chegar na raiz, caso não encontre o valor. Isso faz com que a complexidade do algoritmo aumente, sendo ele O(logn), consequentemente aumentando seu tempo de execução.*

*Existem outras formas de busca, como a busca iterativa, que também possui loops mas que não utilizam recursão, e sim repetição, resultando numa complexidade e tempo de execução menores.*

### Métodos de balanceamento

**Balanceamento dinâmico:** mantém a árvore balanceada toda vez que um nó é inserido ou removido. Ex.: AVL

**Balanceamento global:** permite que a árvore cresça sem limites faz o balanceamento só quando for necessário, de forma externa.

# 2. AVL

💡 Árvore binária de busca auto balanceada em que a diferença de alturas das subárvores esquerda e direita de cada nó não pode ser maior que um.

**Complexidade da árvore AVL em notação O**

| | Média | Pior caso |
|---|---|---|
| **Espaço** | $O(n)$ | $O(n)$ |
| **Busca** | $O(\log n)$ | $O(\log n)$ |
| **Inserção** | $O(\log n)$ | $O(\log n)$ |
| **Deleção** | $O(\log n)$ | $O(\log n)$ |

### Aplicações (alguns exemplos)

- Utilizada para indexar e buscar quantidades grandes de registros em bases de dados;
- Softwares que necessitam de busca otimizada;
- Jogos com story-line;

### Vantagens

- Árvores AVL se auto balanceiam;
- Não são assimétricas;
- Pesquisas mais rápidas que as Rubro Negras;
- Melhor complexidade de busca em relação aos outros tipos de árvores binárias;
- Sua altura não consegue passar de log(n), sendo n o número total de nós na árvore.

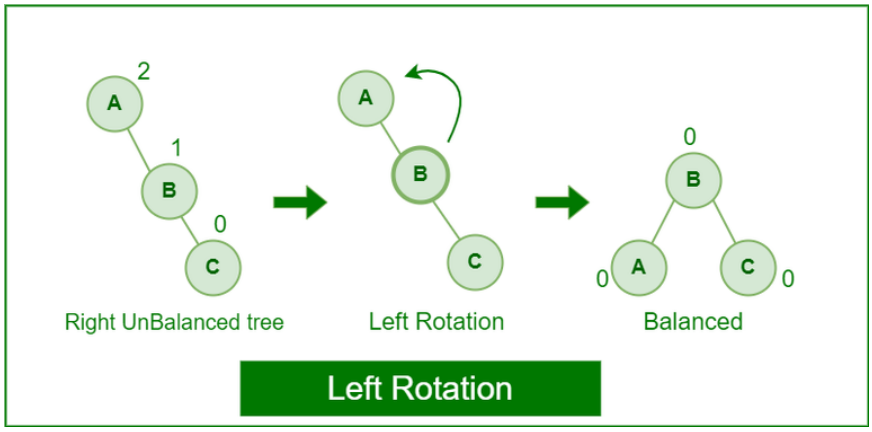
### Desvantagens

- Difícil de implementar;
- Possui fatores constantes bem altos para algumas operações;
- São menos utilizadas que as Rubro negras;
- Como é estritamente balanceada, sua inserção e remoção são bem complicadas, já que necessitam de várias rotações;
- Precisam de mais processamento para realizar os balanceamentos.

## Rotações em subárvores de uma AVL

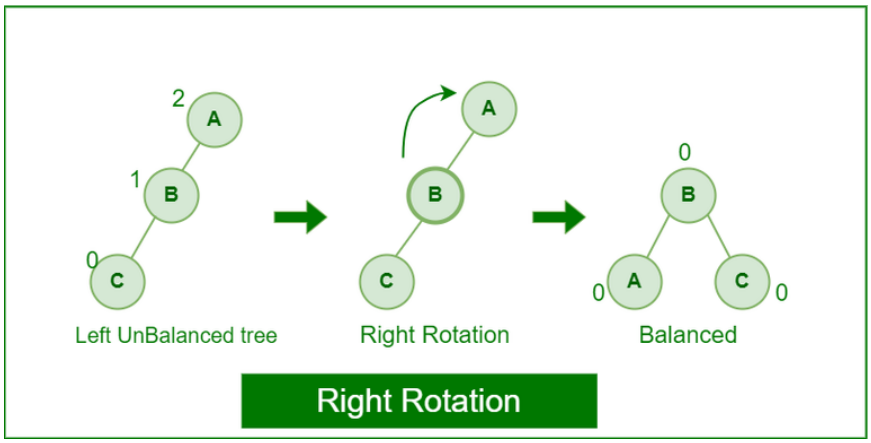
### Rotação esquerda

- Acontece quando um nó é inserido em uma subárvore direita de uma subárvore direita e precisa-se de um balanceamento



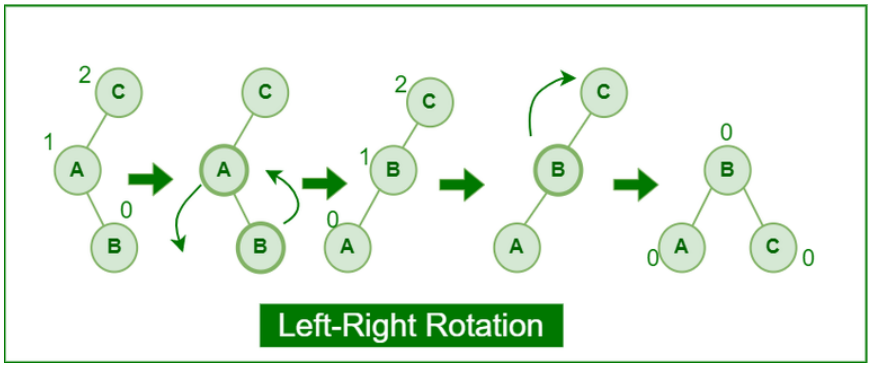

## Rotação direita

- Acontece quando um nó é inserido em uma subárvore esquerda de uma subárvore esquerda e precisa-se de um balanceamento



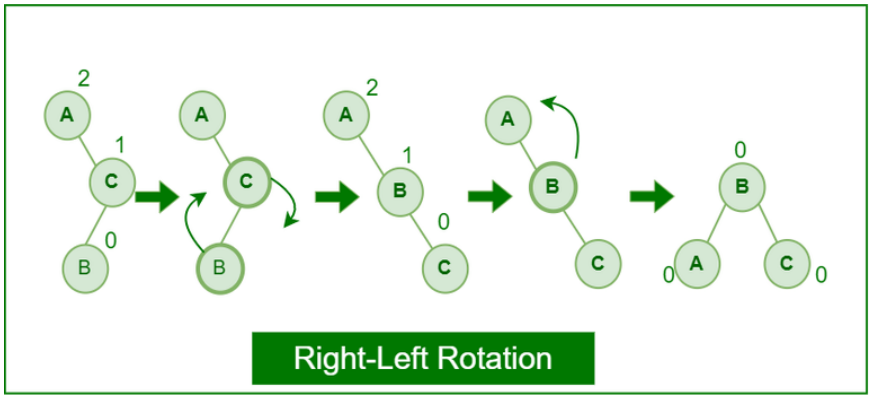

## Rotação LR

- Primeiro acontece uma rotação esquerda e depois uma rotação direita

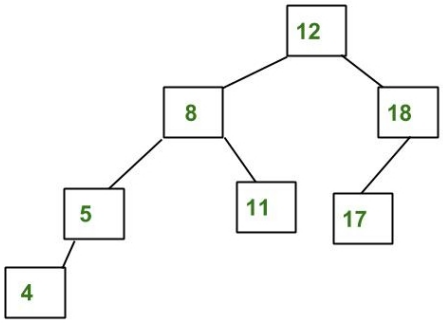



## Rotação RL

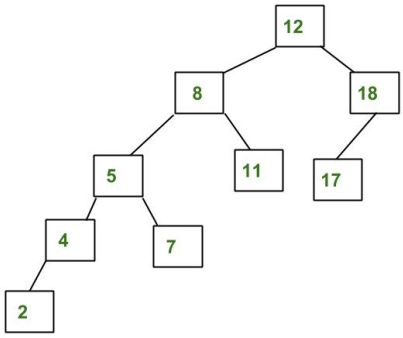
- Primeiro acontece uma rotação direita e depois uma rotação esquerda




## Exemplo de AVL

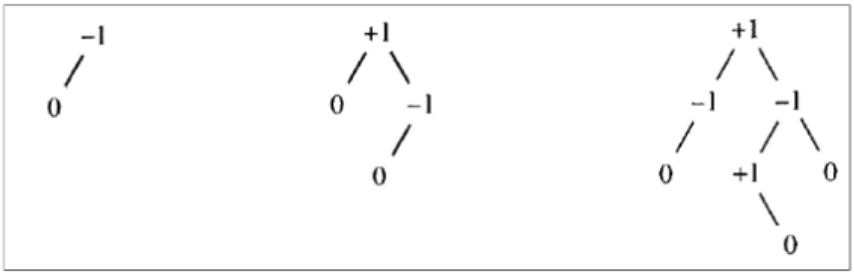



## Exemplo de árvore que não é AVL




## Balanceamento

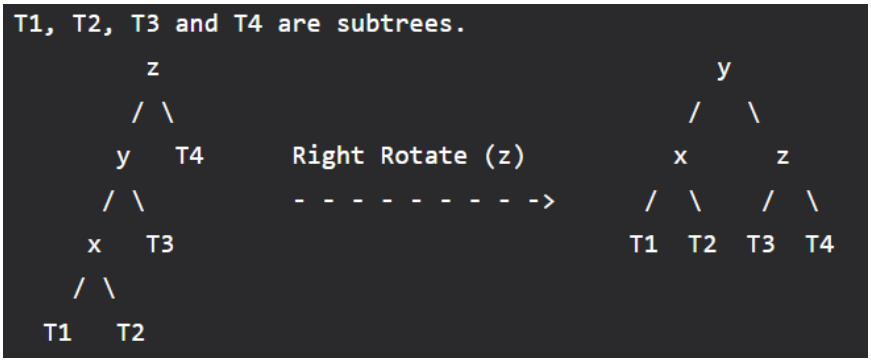
- **hd – he ϵ {0, 1, -1}**
- Se o **fator de balanceamento** de qualquer nó ficar menor do que -1 ou maior do que 1, então a árvore tem que ser balanceada.



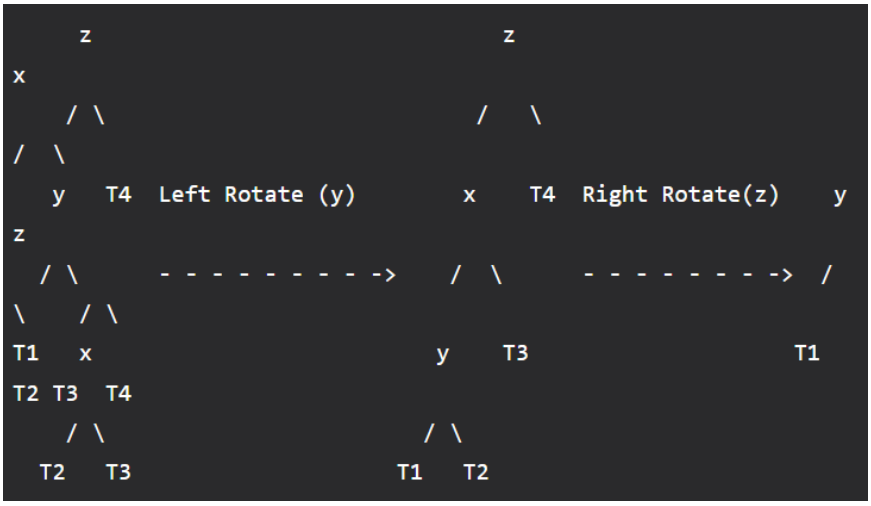

## Inserção em AVL

Se quisermos inserir um nó k na árvore,

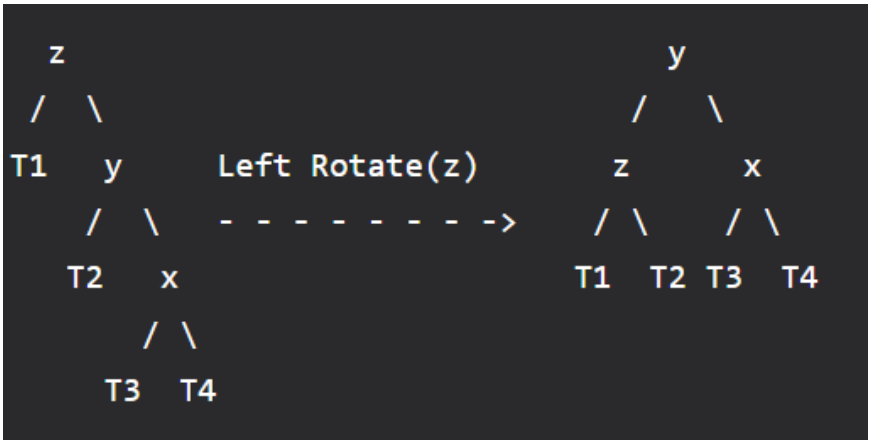
- deveremos inserir k do jeito que inserimos em uma árvore binária de busca;
- então, a partir de k, precisamos encontrar o primeiro nó que está desbalanceado. para isso, calculamos a altura de cada nó e vemos qual deles é o primeiro a aparecer desbalanceado (ou seja, possui altura diferente de -1, 0 e 1)
- ao encontrar o nó desbalanceado, realizamos o balanceamento utilizando as rotações apropriadas para o caso. Existem 4 casos possíveis:
  - **LL - Left Left**



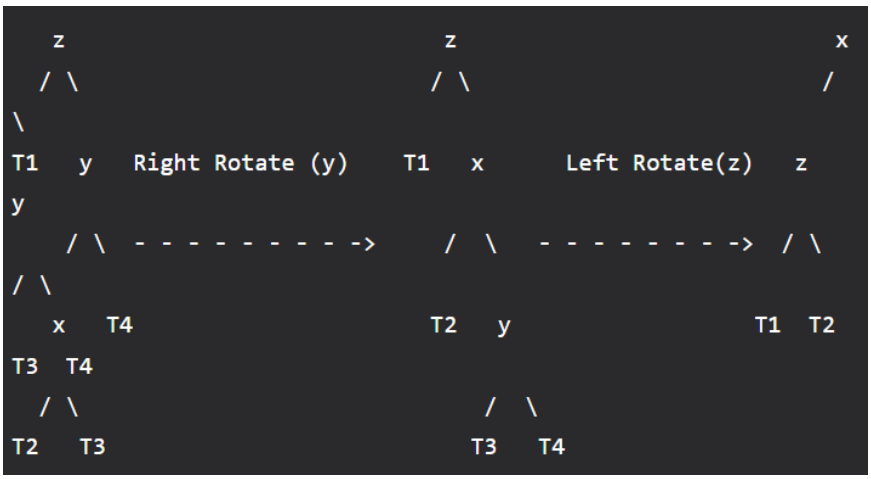

  - **LR - Left Right**




- **RR - Right Right**




- **RL - Right Left**




## Remoção em AVL

- O primeiro passo para remover é escolher o nó que se quer retirar da árvore e realizar a remoção padrão de uma árvore binária de busca;
- Se quisermos remover um nó k, por exemplo, após retirá-lo, precisamos calcular o seu balanceamento atual e encontrar o primeiro nó desbalanceado. Assim, como na inserção, realizamos o balanceamento utilizando as rotações apropriadas para o caso, sendo elas as mesmas 4 apresentadas acima.

# 3. Árvore Rubro Negra

> Sabe-se que a performance de uma árvore binária de busca depende muito do seu formato, já que ela pode ficar muito assimétrica e, no seu pior caso, acabar se tornando uma estrutura linear de complexidade O(n).

💡 Árvores rubro negras são um tipo de árvores binárias de busca balanceadas que fazem uso de um conjunto de regras que garantem que elas estejam sempre balanceadas.
Esse balanceamento garante que a complexidade de suas operações de inserção, remoção e busca sejam sempre O(log n), independente do formato inicial da árvore.

### Atributos de uma árvore rubro negra

- Cada nó possui uma cor - vermelho ou preto;
- A raiz da árvore é sempre preta;
- Um nó vermelho não pode ter um pai ou filho também vermelho - nós vermelhos não podem ser adjacentes;
- Todo caminho de um nó para algum de seus descendentes deve possuir o mesmo número de nós pretos (isso inclui a raiz);
- Toda folha (nó nil) deve ser preto

💡 A altura de uma árvore rubro negra é sempre O(log n), em que n é o número de nós existentes nela.

| Algorithm | Time Complexity |
|---|---|
| Search | O(log n) |
| Insert | O(log n) |
| Delete | O(log n) |

## Rubro Negra vs AVL

- AVL requer mais rotações para ser balanceada;
- Rubro negra: máximo de 2 rotações por balanceamento;
- Busca é mais rápida na AVL, já que ela é estritamente mais balanceada;
- Inserção e deleção são mais rápidas na rubro negra, pois requer menos rotações.

## Outras características sobre Rubro Negras:

- A altura preta de uma árvore rubro negra é o número de nós pretos em um caminho da raiz até uma das folhas da árvore - uma árvore de altura h possui altura preta equivalente a h/2;
- altura da árvore com n nós = 2 log2(n+1);
- todos os nós são pretos;

## Aplicações (alguns exemplos)

- usada para implementar CPU Scheduling no Linux;
- usadas para implementar administrador de memória virtual em alguns sistemas operacionais;

- utilizadas em algoritmos como o caminho mais curto de Dijkstra ou o algoritmo de Prim;
- utilizadas na implementação de engines de jogos;
- utilizada no algoritmo de K-mean clustering em machine learning para diminuir o tempo de complexidade

### Vantagens

- sua complexidade é garantida como O(log n) nas operações básicas de inserção, deleção e busca;
- são auto balanceadas;
- performance eficiente e versatilidade;
- mecanismo simples e fácil de ser compreendido

### Desvantagens

- árvores rubro negras requerem um bit a mais de armazenamento para cada nó, já que precisam armazenar a cor deles;
- complexidade de implementação;
- sua performance eficiente nas operações básicas ainda não a torna a melhor escolha para alguns tipos de dados ou determinados casos de uso.

## Inserção na Árvore Rubro Negra

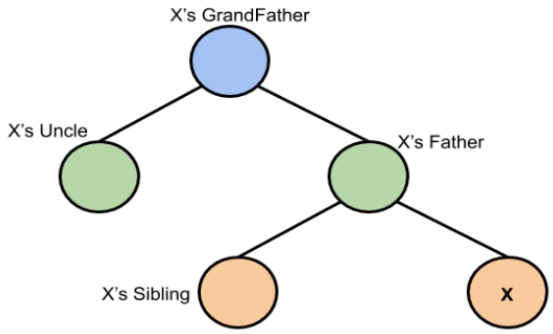
Na árvore rubro negra, utilizamos duas ferramentas para realizar o balanceamento, sendo elas:

- **recolorir** os nós: trocar a cor dos nós. Se for vermelho, trocamos para preto, vice-versa.
  - nó null é sempre preto;
  - recoloração é sempre tentada primeiro - se ela não funcionar, partimos para a rotação.
- **rotação**;

**→ Algoritmo**

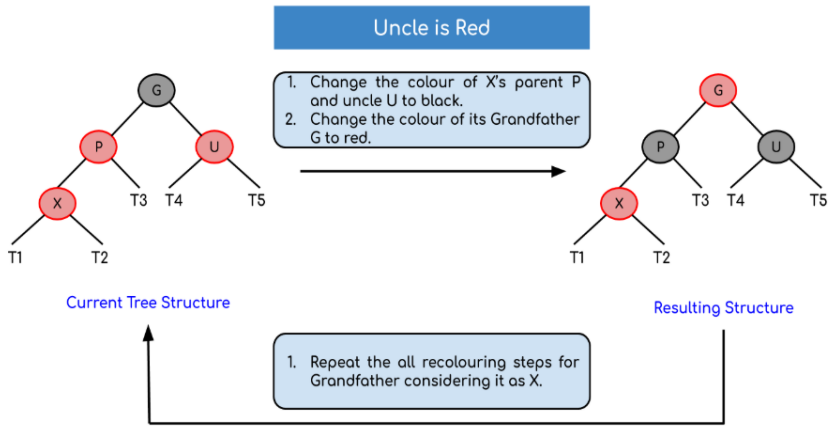
1. Se a árvore estiver vazia, cria-se um novo nó que será a raiz com a cor **preta**;

2. Se a árvore não estiver vazia, cria-se um novo nó que será uma folha com a cor **vermelha**;
3. Se o pai do novo nó for **preto**, então termina-se o processo de inserção;
4. Se o pai do novo nó for **vermelho**, então checamos a cor do pai do irmão do novo nó;
    a. Se a cor dele for **preta** ou *null*, então realizamos as rotações adequadas e recolorimos os nós;
    b. Se a cor for **vermelha**, então recolorimos e também checamos se o pai do pai do novo nó não é o nó raiz e recolorimos ele e por fim, checamos novamente se há algum conflito de cor.

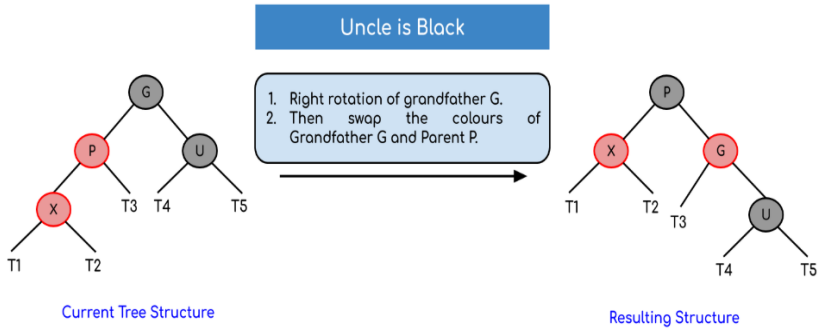



## Exemplo de inserção

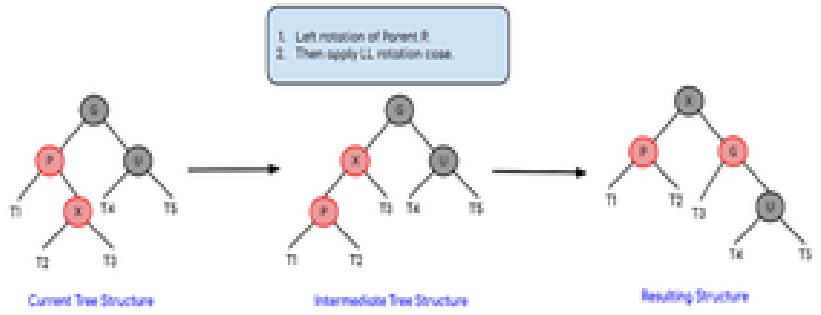
- Inserir um nó e atribuir a ele a cor vermelha, já que já existem outros nós na árvore;
- Se o nó for raiz, trocamos a cor dele para preto, mas se não for então nós checamos a cor do pai dele;
    - Se for preto, não trocamos a cor;
    - Se for vermelho, checamos a cor do tio desse nó;
        - Se o nó tio for vermelho, trocamos a cor do nó pai e do nó tio para preto. Trocamos também a cor do nó avô para vermelho e repetimos o processo para ele.
            - Se o nó avô for raiz, não trocamos a cor dele para vermelho.
            -



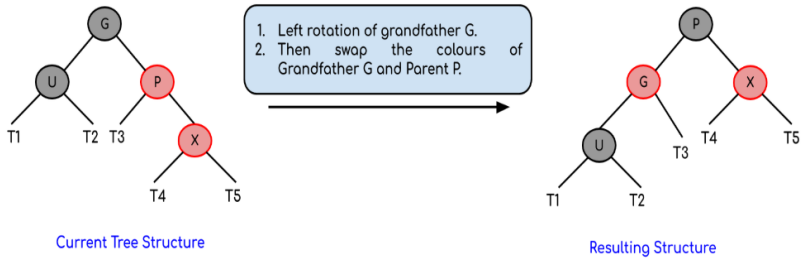

- Se o nó tio for preto, existem 4 possibilidades:
  - **Rotação LL (left left)**



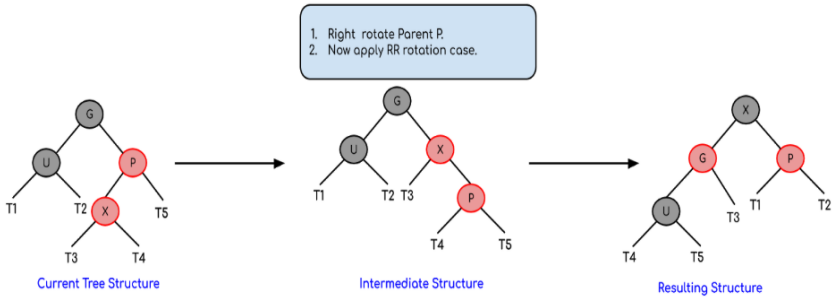

  - **Rotação LR (left right)**



  - **Rotação RR (right right)**




- **Rotação RL (right left)**

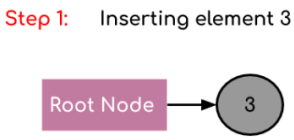



Recoloração é realizada depois dessas rotações, de acordo com cada rotação.

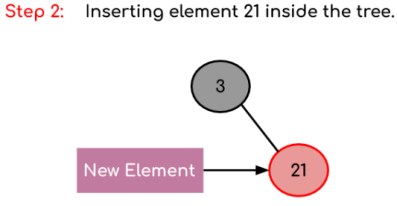
## Exemplo de inserção em uma árvore vazia

Criar uma árvore rubro negra com elementos 3, 21, 32 e 15, começando de uma estrutura vazia:

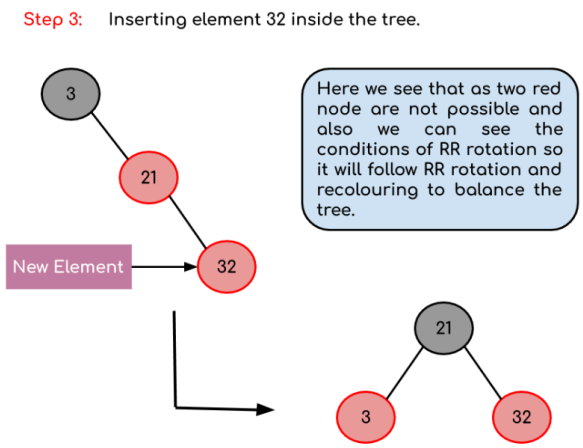
1. O primeiro elemento inserido é a raiz e possui a cor preta;



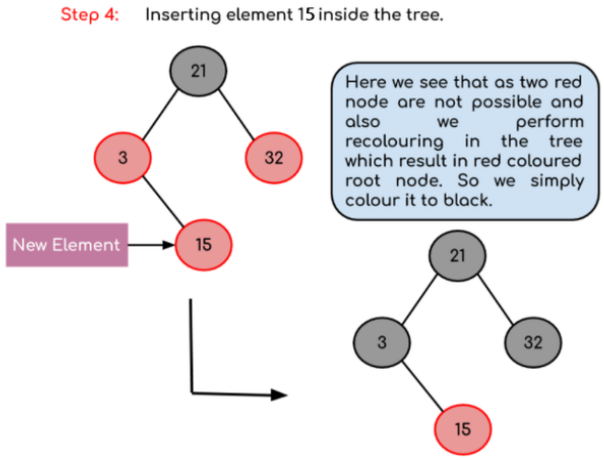

2. O próximo elemento é uma folha e é inserido com a cor vermelho do lado direito já que é maior que 3;



3. Inserir o 32 do lado direito do 21, já que é maior que ele. Como um nó vermelho não pode ter um filho vermelho, precisamos rotacionar a árvore e depois recolorir.




4. Inserir o 15 como nó folha do nó 3. Como o 3 é vermelho, ele não pode ter filhos vermelhos, então checamos a condição e pecebemos que o irmão do 3 também é vermelho. Assim, recolorimos os dois e mantemos o 15 vermelho, já que ele é uma folha e precisa manter-se nesse estado.




# Remoção na Árvore Rubro Negra

- Na remoção, precisamos checar a cor do irmão do nó a ser removido para realizar a ação mais adequada;
- A principal propriedade violada na remoção é a mudança da altura preta nas subárvores, já que a remoção de um nó preto acaba reduzindo a altura preta do caminho da raiz até um determinado nó.
- Processo mais complexo - entra o conceito de preto duplo (double black)

- Double black: quando um nó preto é removido e substituído por um nó filho preto, o filho é marcado como preto duplo e então precisamos tornar ele em um preto normal.

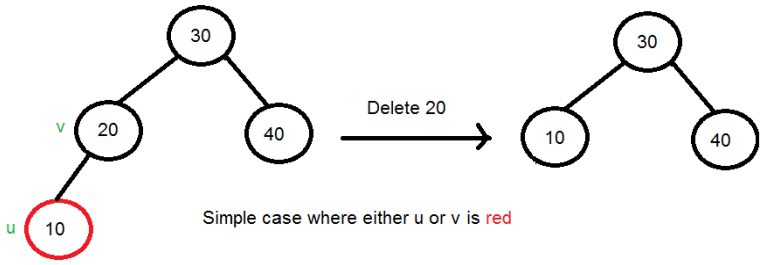
→ **Algoritmo**

1. Realizar a remoção padrão da árvore binária de busca.
   a. Sempre acabamos removendo um nó que é uma folha ou possui só um filho - esses são os casos em que precisamos ter cuidado.
   b. Remover um nó k que possui um filho u. O nó u vai ficar no lugar do nó a ser removido.

Existem dois casos possíveis:

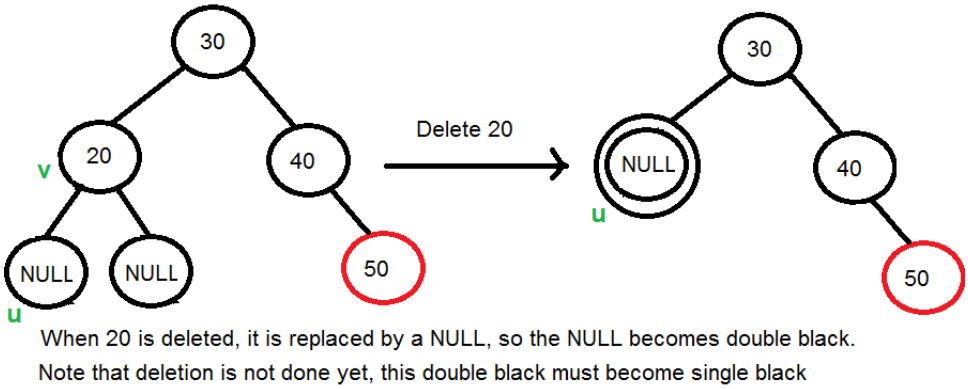
2. **Se u ou v forem vermelhos**, colorimos o filho que foi substituiu o nó removido como preto para não haver mudança na altura preta.

   *Obs: u e v não podem ser vermelhos, pois v é pai de u.*




3. **Se u e v forem pretos**, colorimos u como preto duplo. O objetivo então é transformar esse nó preto duplo em um preto comum.

   Obs: Se v é um nó folha, então u é null e a cor de um nó null é considerada preta - então, a remoção de um nó folha preto também gera um preto duplo.



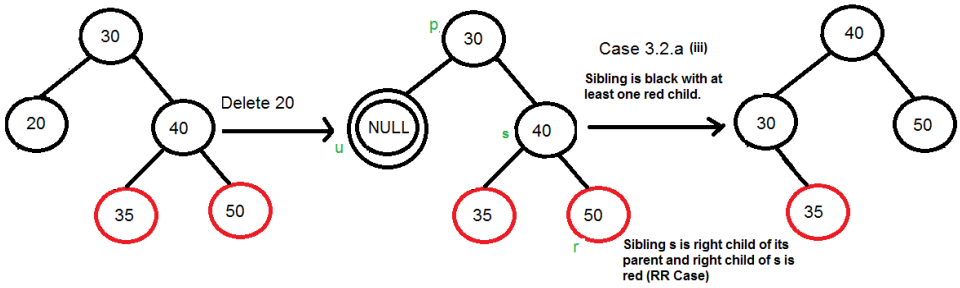
3.2. Enquanto o nó u atual for preto duplo e não for a raiz - considerar o irmão do nó como s:

a. se o irmão s for preto e pelo menos um dos filhos do irmão for vermelho: realizamos rotações - isso nos leva a 4 subcasos que dependem das posições de s e de seu filho vermelho (r).

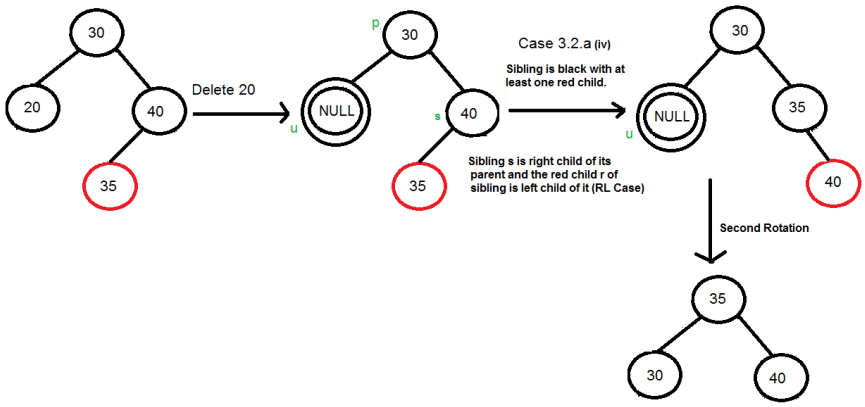
   a. (i) left left: s é filho esquerdo do seu pai e r é filho esquerdo de s ou os dois filhos de s são vermelhos;

   b. (ii) left right:  é filho esquerdo do seu pai e r é filho direito.

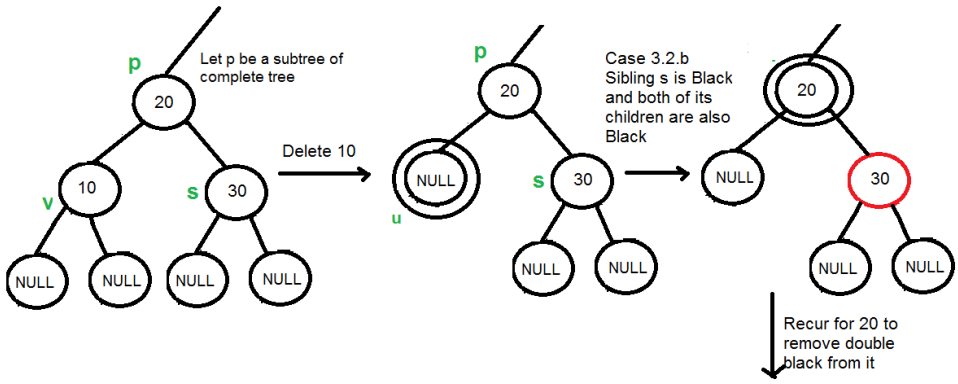
   c. (iii) right right: s é filho direito do seu pai e r é filho direito de s ou seus dois filhos são vermelhos.




d. (iv) right left: s é filho direito do seu pai e r é filho esquerdo de s



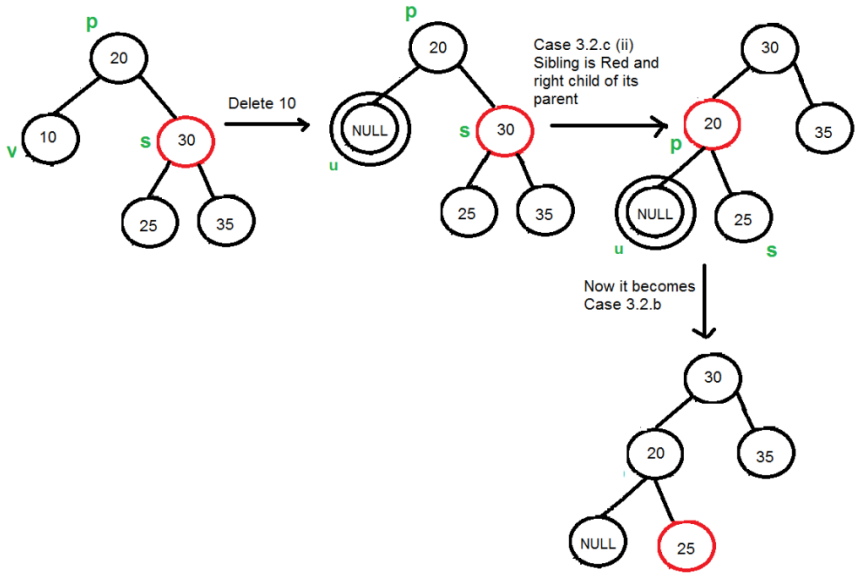

b. se o irmão é preto e seus dois filhos são pretos: colorimos eles e voltamos atrás do pai se o mesmo for preto.




- no caso acima, se o pai fosse vermelho, então precisaríamos voltar atrás do pai - bastaria colorir ele para preto
    - vermelho + preto duplo = preto simples

c. se o irmão for vermelho, fazemos uma rotação para mover o irmão para cima e depois colorimos ele e o pai.

- o novo irmão é sempre preto;




# Bibliografia


- https://www.geeksforgeeks.org/introduction-to-red-black-tree/
- https://www.geeksforgeeks.org/insertion-in-red-black-tree/
- https://www.geeksforgeeks.org/deletion-in-red-black-tree/
- https://www.geeksforgeeks.org/introduction-to-avl-tree/

- https://www.geeksforgeeks.org/insertion-in-an-avl-tree/
- https://www.geeksforgeeks.org/deletion-in-an-avl-tree/
- https://www.geeksforgeeks.org/avl-trees-containing-a-parent-node-pointer/
- https://www.youtube.com/watch?v=jDM6_TnYIqE&pp=ugMICgJwdBABGAE%3D
- https://www.youtube.com/watch?v=3RQtq7PDHog&pp=ugMICgJwdBABGAE%3D
- https://www.youtube.com/watch?v=qA02XWRTBdw
- https://www.youtube.com/watch?v=aZjYr87r1b8&pp=ugMICgJwdBABGAE%3D