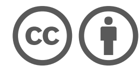
# Árvores de busca binária balanceadas

Programação II – Engenharia de Telecomunicações

Prof. Emerson Ribeiro de Mello

mello@ifsc.edu.br

# Licenciamento



Slides licenciados sob Creative Commons "Atribuição 4.0 Internacional"

# Árvores de busca binária balanceadas

Projetadas para manter sua altura em $O(\log n)$ e assim manter o desempenho das operações de busca, inserção e remoção em $O(\log n)$

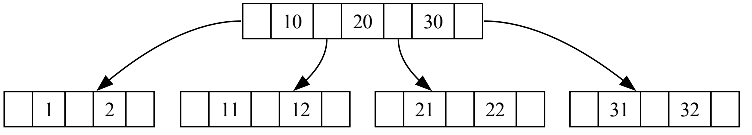
- **Árvore de busca binária balanceada**
  - Se para cada nó, a diferença entre a altura da subárvore esquerda e a subárvore direita é no máximo 1
- **Árvore de busca binária completamente balanceada**
  - Igual a balanceada e todas as folhas estão no mesmo nível
- **Árvore de busca binária perfeitamente balanceada**
  - Igual a completamente balanceada e o número de nós em cada nível é o máximo possível

# Tipos de árvores de busca binária balanceadas

- **Árvore AVL**
  - Inventada em 1962 por Adelson-Velsky e Landis
  - Inserção e remoção são mais custosas, pois impõe regras de balanceamento mais rígidas
  - Adequadas para aplicações onde as operações de busca são mais frequentes que inserção e remoção
- **Árvore vermelho-preto**
  - Inventada em 1972 por Rudolf Bayer
  - Desempenho melhor para inserção e remoção, pois regras de balanceamento não são tão rígidas quanto aquelas da AVL
  - Implementação mais complexa que a AVL
  - Utilizada em implementações de mapas e conjuntos em C++ e Java

# Tipos de árvores de busca binária balanceadas

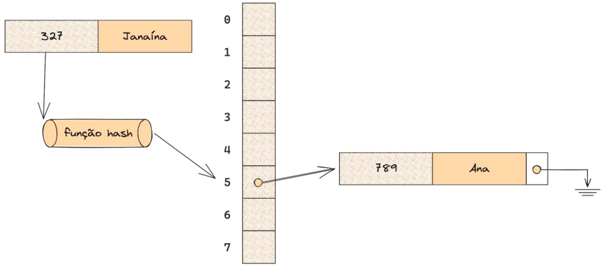
- **Árvore B**
  - Inventada em 1971 por Rudolf Bayer e Edward Meyers McCreight
    - A origem do nome é desconhecida, mas provavelmente vem da palavra *balanceada* ou de *Bayer*
  - Projetada para funcionar em memória secundária, como um disco rígido
  - Semelhante a uma árvore vermelho-preto, mas minimiza o número de acessos ao disco
  - Amplamente utilizada em bancos de dados e sistemas de arquivos



# Exemplo de uso de árvore de busca binária balanceada

## Implementação do HashMap no Java 8

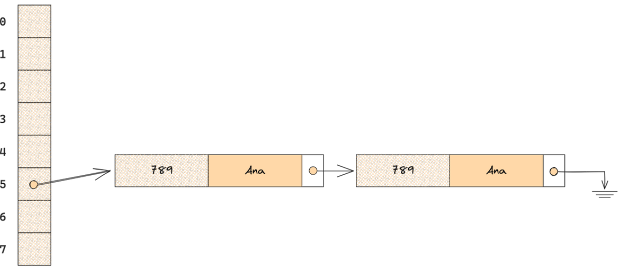
- Tabela de dispersão combinada com uma árvore vermelho-preto
- Busca em caso de colisão é feita em tempo $O(\log n)$ no pior caso, contra $O(n)$ se fosse feita em uma lista encadeada



# Exemplo de uso de árvore de busca binária balanceada

## Implementação do HashMap no Java 8

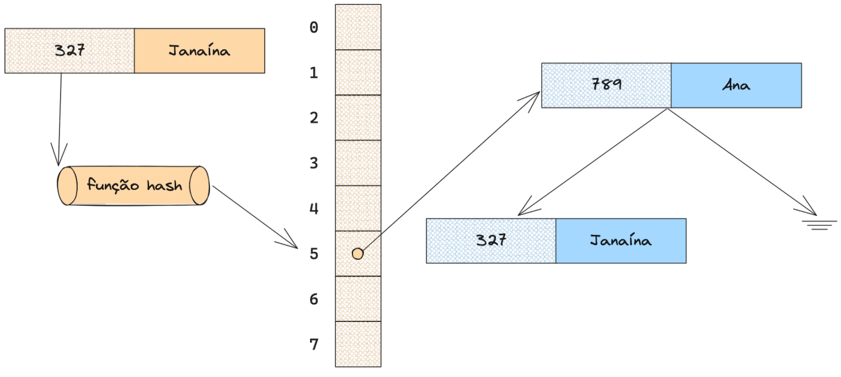
- Tabela de dispersão combinada com uma árvore vermelho-preto
- Busca em caso de colisão é feita em tempo $O(\log n)$ no pior caso, contra $O(n)$ se fosse feita em uma lista encadeada



# Exemplo de uso de árvore de busca binária balanceada

## Implementação do HashMap no Java 8

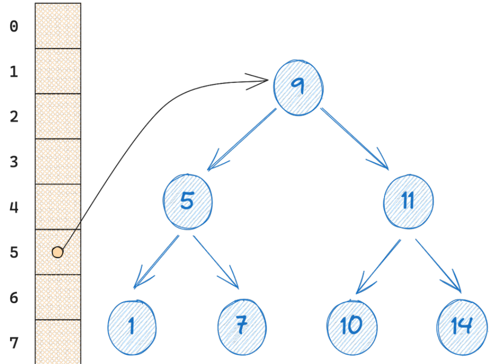
- Tabela de dispersão combinada com uma árvore vermelho-preto
- Busca em caso de colisão é feita em tempo $O(\log n)$ no pior caso, contra $O(n)$ se fosse feita em uma lista encadeada



# Exemplo de uso de árvore de busca binária balanceada

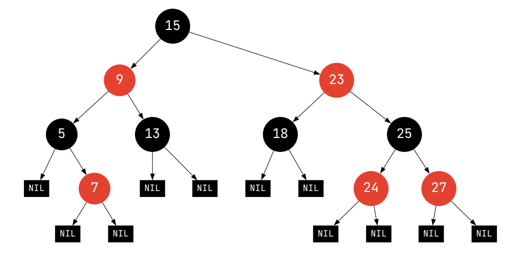
## Implementação do HashMap no Java 8

- Tabela de dispersão combinada com uma árvore vermelho-preto
- Busca em caso de colisão é feita em tempo $O(\log n)$ no pior caso, contra $O(n)$ se fosse feita em uma lista encadeada



# Árvore Vermelho-Preto[1]

1. Nó é colorido de vermelho ou preto
   - A cor é armazenada em um *bit*
2. Todas as folhas são pretas e não carregam informação
   - Representada por um nó nulo
3. Se um nó é vermelho, então seus filhos são pretos
4. Todos os caminhos da raiz até as folhas descendentes contém o mesmo número de nós pretos




Propriedades garantem que a altura da árvore é $O(\log n)$

[1] Esse nome foi cunhado por Guibas & Sedgewick em um artigo em 1978, por 2 motivos: quadro branco e impressora colorida experimental que usaram. https://sedgewick.io/research

# Árvore AVL

- Uma árvore AVL é uma árvore binária de busca balanceada
- Uma árvore binária $T$ é denominada AVL quando, para qualquer nó $v$ de $T$, as alturas de suas duas subárvores, esquerda e direita, diferem em módulo de até uma unidade

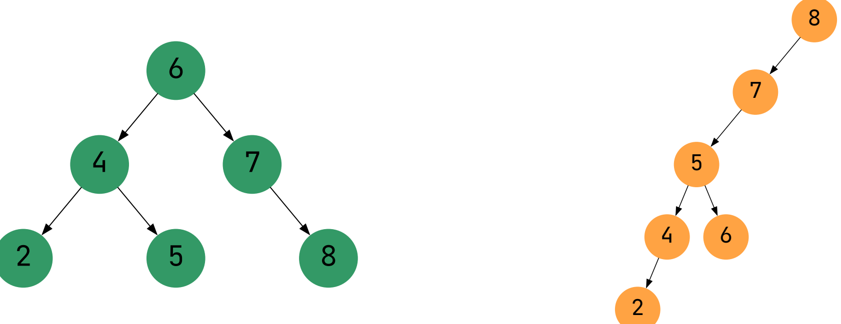
$$|h_{esq}(v) - h_{dir}(v)| \leq 1, \forall v \in T \quad (1)$$

- Um nó que satisfaça essa condição é chamado de **nó regulado**, caso contrário é chamado de **nó desregulado**

```
typedef struct no_avl {
    int valor;
    int altura; // altura da subárvore
    struct no *esquerda;
    struct no *direita;
} no_avl_t;
```

# Árvores de busca binária balanceadas

Uma árvore pode ficar **desbalanceada após operações de inserção ou remoção**




- Árvores desbalanceadas podem ter altura $O(n)$ e operações de busca, inserção e remoção podem ter complexidade $O(n)$

# Inclusão em árvores AVL

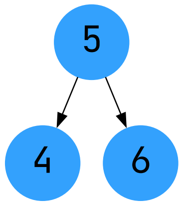
- Após cada inclusão é necessário verificar se algum nó foi desregulado
  - $-1 < (h_{esq}(v) - h_{dir}(v)) > 1$
- Se algum nó foi desregulado, é necessário realizar uma operação de balanceamento (ancestral mais próximo do nó inserido)
- A **operação de balanceamento** é realizada em um nó desregulado e seus ancestrais até a raiz
- As operações de balanceamento são chamadas de **rotações**

```
// retorna a altura da subárvore
int altura(no_avl_t *v) {
  if (v == NULL) {
    return 0;
  } else {
    return v->altura;
  }
}

// negativo se a subárvore direita for
    maior
int fator_balanceamento(no_avl_t *v) {
  if (v == NULL) {
    return 0;
  } else {
    return altura(v->esquerda) - altura(v->
    direita);
  }
}
```

# Inclusão em árvores AVL

**Nó desregulado:** $-1 < (h_{esq}(v) - h_{dir}(v)) > 1$

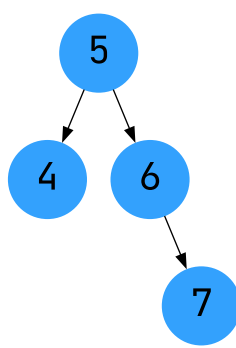
- $h_{esq}(5) - h_{dir}(5) = 0$
- $h_{esq}(4) - h_{dir}(4) = 0$
- $h_{esq}(6) - h_{dir}(6) = 0$



# Inclusão em árvores AVL

**Nó desregulado:** $-1 < (h_{esq}(v) - h_{dir}(v)) > 1$

- $h_{esq}(5) - h_{dir}(5) = -1$
- $h_{esq}(4) - h_{dir}(4) = 0$
- $h_{esq}(6) - h_{dir}(6) = -1$
- $h_{esq}(7) - h_{dir}(7) = 0$

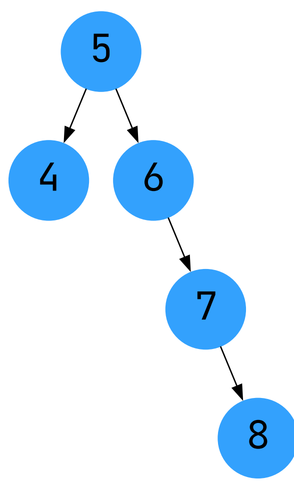


# Inclusão em árvores AVL

**Nó desregulado:** $-1 < (h_{esq}(v) - h_{dir}(v)) > 1$

- $h_{esq}(5) - h_{dir}(5) = -2$
- $h_{esq}(4) - h_{dir}(4) = 0$
- $h_{esq}(6) - h_{dir}(6) = -2$
- $h_{esq}(7) - h_{dir}(7) = -1$
- $h_{esq}(8) - h_{dir}(8) = 0$

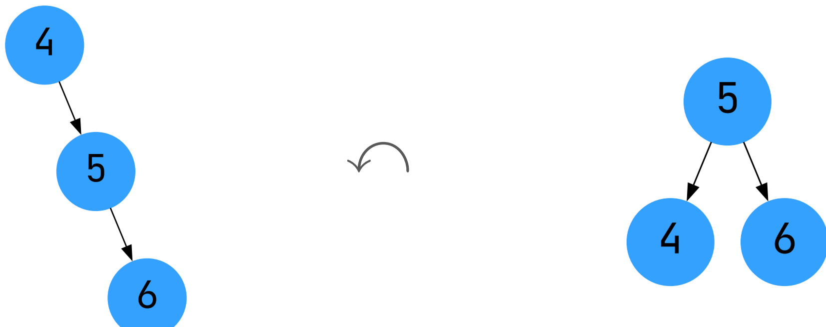
Nó desregulado: 6



# Rotação simples à esquerda

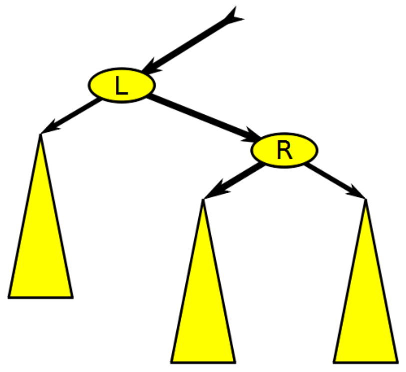
## Caso direita-direita

- O nó 4 está desregulado e assim é feita uma rotação simples à esquerda, onde o nó 4 se torna filho esquerdo do nó 5
- Todos os nós são movidos uma posição para a esquerda

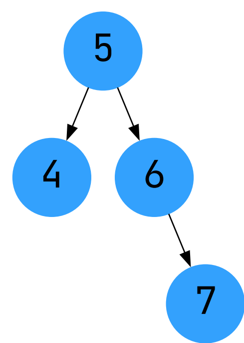


# Rotação simples à esquerda

## Inserção de um novo nó na subárvore direita



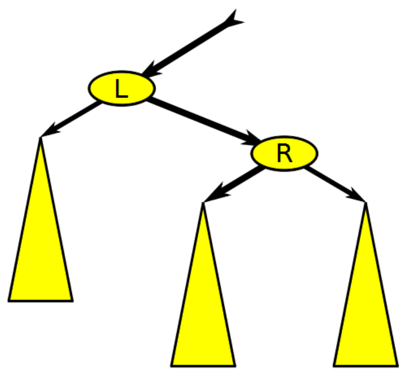

Fonte: Wikipedia



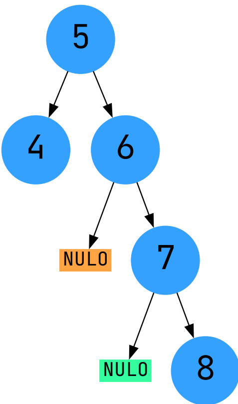

Será adicionado novo nó com valor 8

# Rotação simples à esquerda

## Inserção de um novo nó na subárvore direita



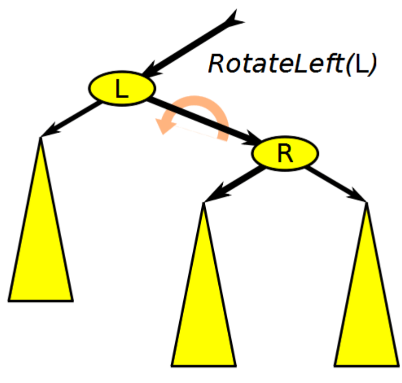

Fonte: Wikipedia



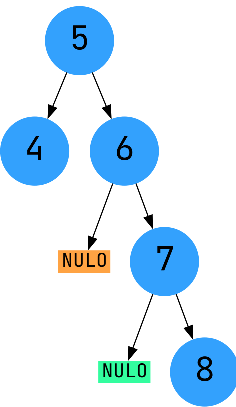

Nó desregulado: 6

# Rotação simples à esquerda

## Inserção de um novo nó na subárvore direita



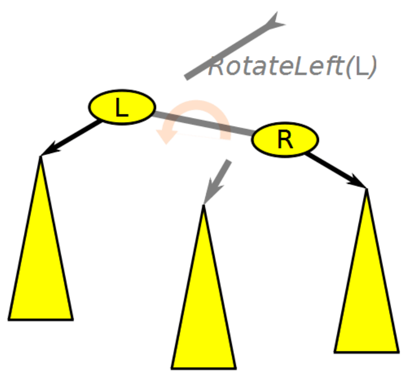

Fonte: Wikipedia



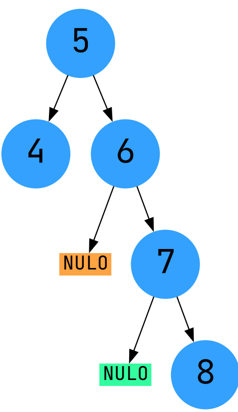

Nó desregulado: 6

# Rotação simples à esquerda

## Inserção de um novo nó na subárvore direita



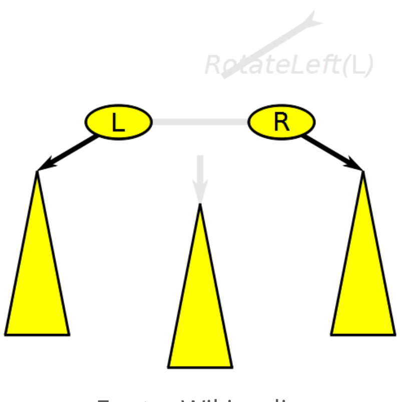

Fonte: Wikipedia



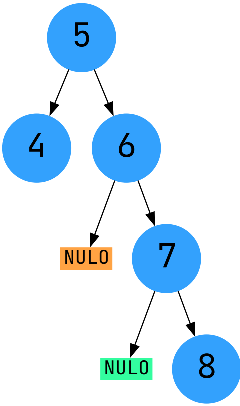

Nó desregulado: 6

# Rotação simples à esquerda

## Inserção de um novo nó na subárvore direita



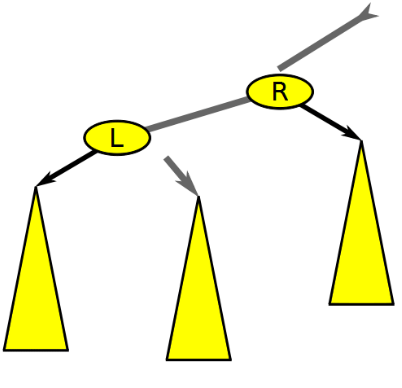

Fonte: Wikipedia



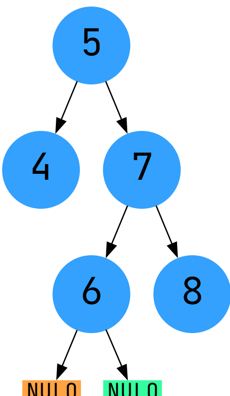

Nó desregulado: 6

# Rotação simples à esquerda

## Inserção de um novo nó na subárvore direita

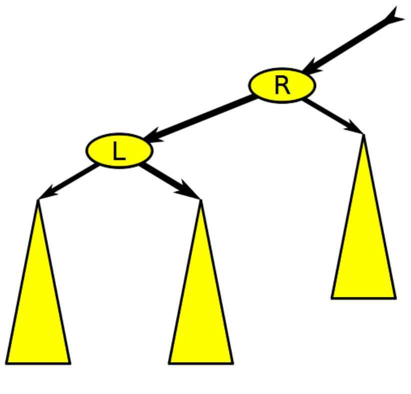



Fonte: Wikipedia

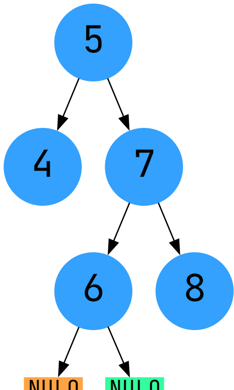


# Rotação simples à esquerda

## Inserção de um novo nó na subárvore direita

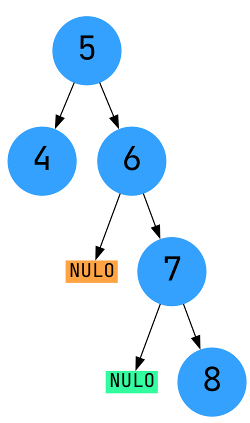



Fonte: Wikipedia



# Rotação simples à esquerda

## Implementação

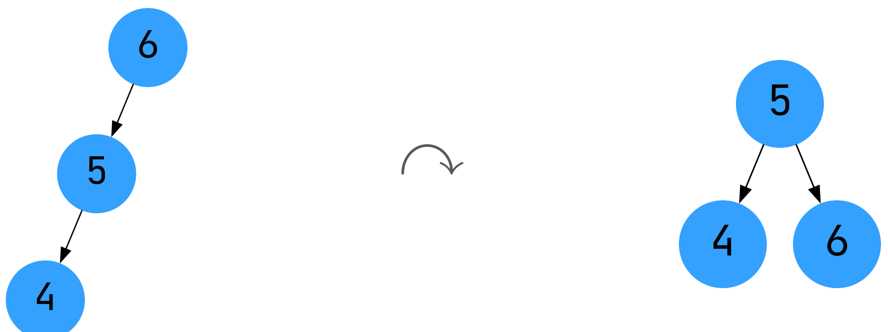



```c
#define max(a,b) (((a) > (b)) ? (a) : (b))

// v é o nó desregulado (6) na figura ao lado
no_avl_t *rotacao_esquerda(no_avl_t *v) {
    // u é (7) na figura ao lado
    no_avl_t *u = v->direita;
    // filho direito de v será o filho esquerdo de u (8)
    v->direita = u->esquerda;
    // filho esquerdo de u será v (6)
    u->esquerda = v;

    // atualiza a altura de v
    v->altura = max(altura(v->esquerda), altura(v->direita)) + 1;
    // atualiza a altura de u
    u->altura = max(altura(u->esquerda), altura(u->direita)) + 1;

    // retorna o novo nó raiz da subárvore
    return u;
}
```

# Rotação simples à direita

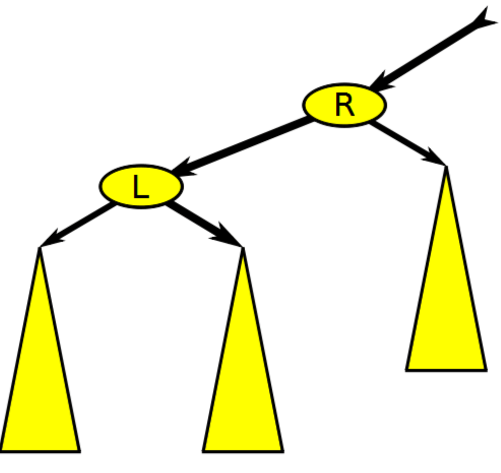
## Caso esquerda-esquerda

- O nó 6 está desregulado e assim é feita uma rotação simples à direita, onde o nó 6 se torna filho direito do nó 5
- Todos os nós são movidos uma posição para a direita



# Rotação simples à direita

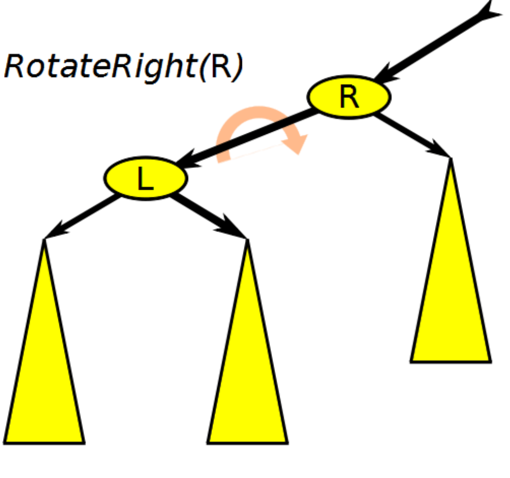
## Quando um nó desregulado tem um filho desregulado à esquerda




Fonte: Wikipedia

# Rotação simples à direita

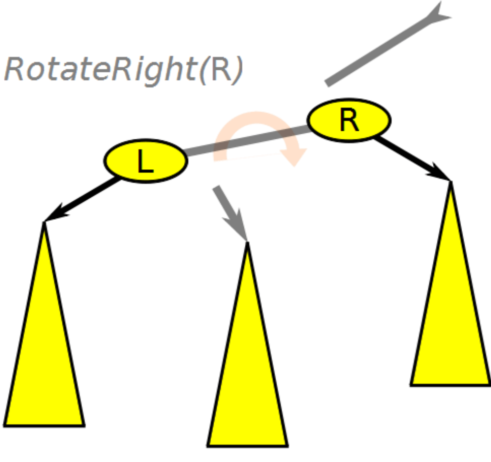
## Quando um nó desregulado tem um filho desregulado à esquerda




Fonte: Wikipedia

# Rotação simples à direita

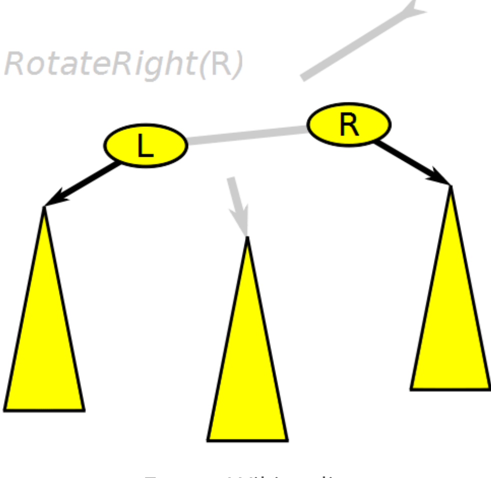
## Quando um nó desregulado tem um filho desregulado à esquerda




Fonte: Wikipedia

# Rotação simples à direita

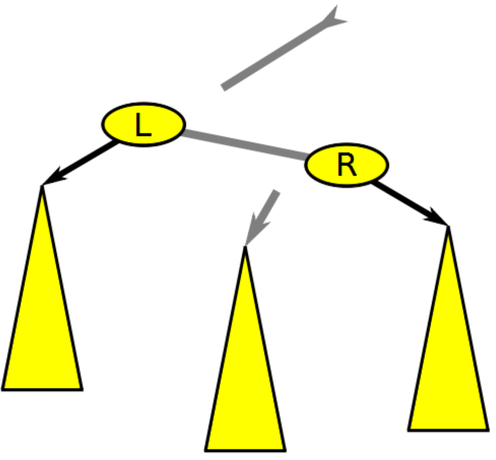
## Quando um nó desregulado tem um filho desregulado à esquerda




Fonte: Wikipedia

# Rotação simples à direita

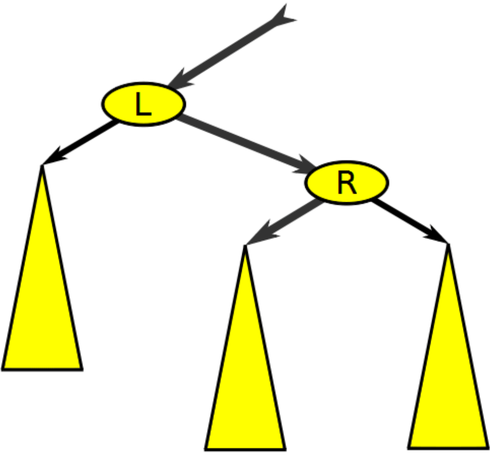
## Quando um nó desregulado tem um filho desregulado à esquerda




Fonte: Wikipedia

# Rotação simples à direita

## Quando um nó desregulado tem um filho desregulado à esquerda

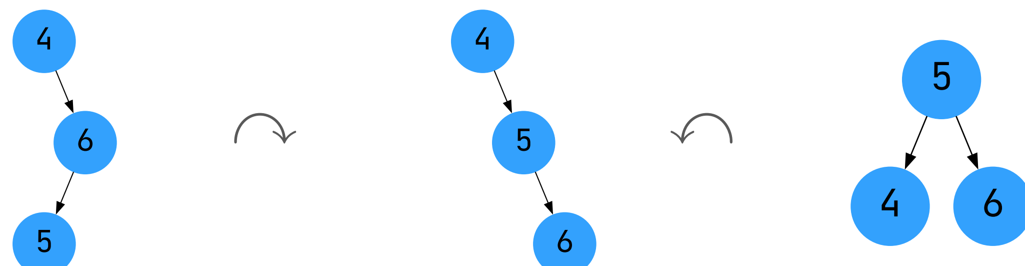



Fonte: Wikipedia

# Rotação dupla à esquerda ou rotação direita-esquerda

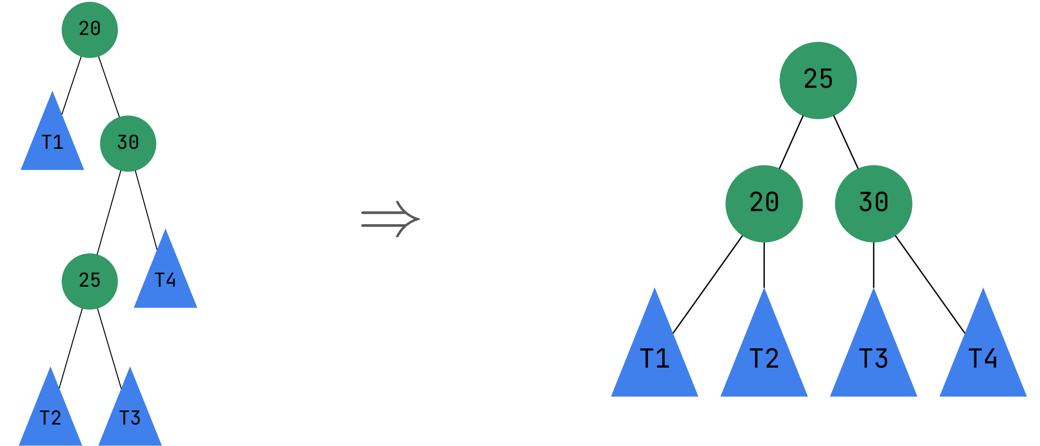
## Caso direita-esquerda

- O nó 4 está desregulado e assim é feita uma rotação dupla à esquerda
- Na primeira rotação simples à direita, o nó 6 se torna filho do nó 5
- Na segunda rotação simples à esquerda, o nó 4 se torna filho do nó 5



# Rotação dupla à esquerda ou rotação direita-esquerda

## Um novo nó é inserido na subárvore esquerda do filho direito do nó desregulado (20)

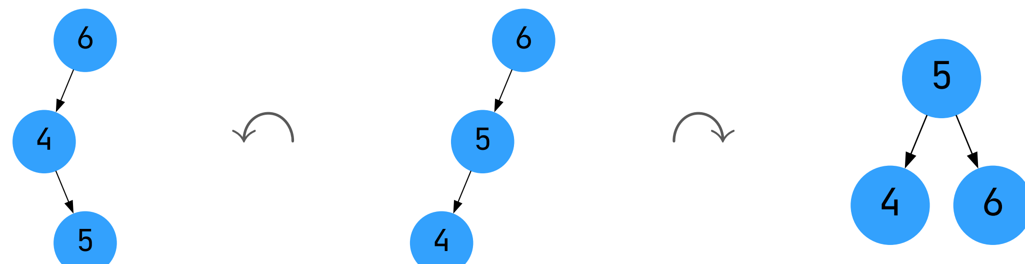



Fonte: Exemplo adaptado de Szwarcfiter e Markenzon (2010)

# Rotação dupla à direita ou rotação esquerda-direita

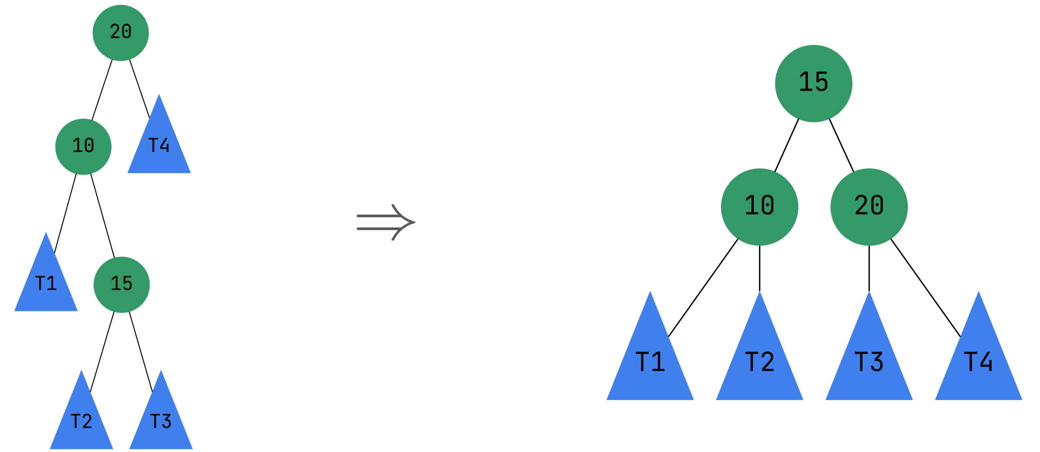
## Caso esquerda-direita

- O nó 6 está desregulado e assim é feita uma rotação dupla à direita
- Na primeira rotação simples à esquerda, o nó 4 se torna filho do nó 5
- Na segunda rotação simples à direita, o nó 6 se torna filho do nó 5



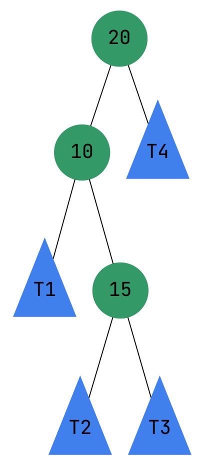
# Rotação dupla à direita

## Um novo nó é inserido na subárvore direita do filho esquerdo do nó desregulado (20)



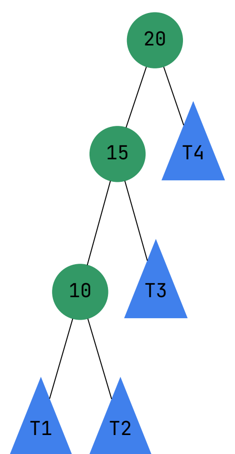

Fonte: Exemplo adaptado de Szwarcfiter e Markenzon (2010)

# Rotação dupla à direita




```
// v é o nó desregulado (20) na figura ao lado
no_avl_t *rotacao_dupla_direita(no_avl_t *v) {

    // rotação simples à esquerda no filho esquerdo de v
    v->esquerda = rotacao_esquerda(v->esquerda);

    // rotação simples à direita em v
    return rotacao_direita(v);
}
```

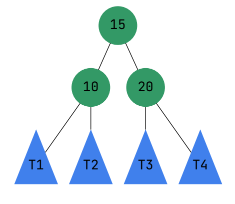
# Rotação dupla à direita




1. Rotação simples à esquerda no filho esquerdo de v

```
// v é o nó desregulado (20) na figura ao lado
no_avl_t *rotacao_dupla_direita(no_avl_t *v) {

    // rotação simples à esquerda no filho esquerdo de v
    v->esquerda = rotacao_esquerda(v->esquerda);

    // rotação simples à direita em v
    return rotacao_direita(v);
}
```

# Rotação dupla à direita




2 Rotação simples à direita em v

```
// v é o nó desregulado (20) na figura ao lado
no_avl_t *rotacao_dupla_direita(no_avl_t *v) {

    // rotação simples à esquerda no filho esquerdo de v
    v->esquerda = rotacao_esquerda(v->esquerda);

    // rotação simples à direita em v
    return rotacao_direita(v);
}
```

# Função para balancear nó em uma árvore AVL

```
no_avl_t *balancear(no_avl_t *v) {
    int fb = fator_balanceamento(v);

    if (fb > 1){// nó desregulado tem filho desregulado à esquerda
    if (fator_balanceamento(v->esquerda) > 0) {
        // caso esquerda-esquerda
        return rotacao_direita(v);
    } else {
        // caso esquerda-direita
        return rotacao_dupla_direita(v);
    }
    } else if (fb < -1) { // nó desregulado tem filho desregulado à direita
    if (fator_balanceamento(v->direita) < 0) {
        // caso direita-direita
        return rotacao_esquerda(v);
    } else {
        // caso direita-esquerda
        return rotacao_dupla_esquerda(v);
    }
    }
    return v;
}
```

# Inserção em árvores AVL

```c
no_avl_t *inserir(no_avl_t *v, int valor) {
  if (v == NULL) {
    v = criar_no(valor);
  } else if (valor < v->valor) {
    v->esquerda = inserir(v->esquerda, valor);
  } else if (valor > v->valor) {
    v->direita = inserir(v->direita, valor);
  }
  v->altura= 1 + max(altura(v->esquerda), altura(v->direita));
  v = balancear(v); // é necessário balancear após cada inserção
  return v;
}
```

# Remoção em árvores AVL

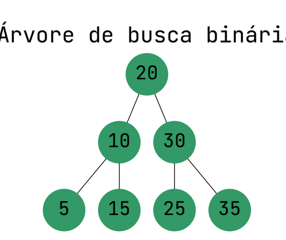
```
no_avl_t *remover(no_avl_t *v, int valor) {
  if (v == NULL) { return NULL;
  } else if (valor < v->valor) {
    v->esquerda = remover(v->esquerda, valor);
  } else if (valor > v->valor) {
    v->direita = remover(v->direita, valor);
  } else { // valor == v->valor
    if (raiz->esquerda == NULL || raiz->direita == NULL) { // nó folha ou nó com um filho
        // ...
    } else{ // nó com dois filhos
        no_t *aux = v->esquerda;
        while (aux->direita != NULL) {
          aux = aux->direita;
        }
        v->valor = aux->valor;
        v->esquerda = remover(v->esquerda, aux->valor);
    }
  }
  if (v != NULL) {
    v->altura = 1 + max(altura(v->esquerda), altura(v->direita));
    v = balancear(v); // é necessário balancear após cada remoção
  }
  return v;
}
```

# Exercícios

# Exercício 1

- Imprimir a árvore como um grafo usando a *dot language*[2]
  - **Dica:** Lógica parecida com a usada para imprimir a árvore em pré-ordem
- Para visualizar, use o site https://dreampuf.github.io/GraphvizOnline/ ou a extensão *Graphviz Preview* do VSCode (é necessário instalar o Graphviz)

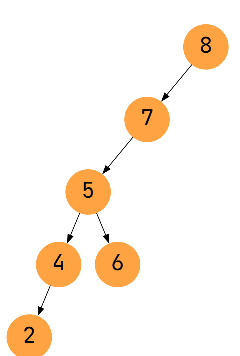
```
strict graph{
  label="Árvore de busca binária";
  node [shape="circle", color="#339966", style="filled",
     fixedsize=true];

  20 -- 10;
  20 -- 30;
  10 -- 5;
  10 -- 15;
  30 -- 25;
  30 -- 35;
}
```




[2] https://graphviz.org/doc/info/lang.html

# Exercício 2

1. Implementar um programa que insira $n$ números inteiros em uma árvore sem balanceamento e imprimir a árvore como um grafo
2. Monte manualmente, por meio de instruções de inserção, uma árvore sem balanceamento como a da figura ao lado
3. Faça o balanceamento da árvore e imprima a árvore como um grafo



# Exercício 3

1. Implementar na biblioteca `libprg` as funções para manipulação de árvores AVL
2. Demonstrar que uma árvore AVL está balanceada após a inserção de um novo nó
3. Demonstrar que uma árvore AVL está balanceada após a remoção de um nó
4. Implementar um programa que insira $n$ números inteiros em uma árvore AVL e imprimir o total de vezes que uma função de rotação foi chamada
5. Implementar um programa que insira $n$ números inteiros em uma árvore AVL, depois remova $z$ números inteiros (sorteados e que estejam na árvore) e imprimir o total de vezes que uma função de rotação foi chamada

# Referências

## Aula baseada em


- CORMEN, Thomas H. et al. **Algoritmos: teoria e prática**. LTC, 2012. Disponível em: <https://app.minhabiblioteca.com.br/reader/books/9788595158092>.
- LAGO PEREIRA, Silvio do. **Estruturas de Dados em C - Uma Abordagem Didática**. Saraiva, 2016. ISBN 9788536517254. Disponível em: <https://app.minhabiblioteca.com.br/#/books/9788536517254>. Acesso em: 1 nov. 2023.
- SZWARCFITER, Jayme Luiz; MARKENZON, Lilian. **Estruturas de dados e seus algoritmos**. LTC, 2010. Disponível em: <https://app.minhabiblioteca.com.br/reader/books/978-85-216-2995-5>.