Árvore AVL (Adelson-Velskii e Landis)

Balanceamento e Implementação

Maria Adriana Vidigal de Lima

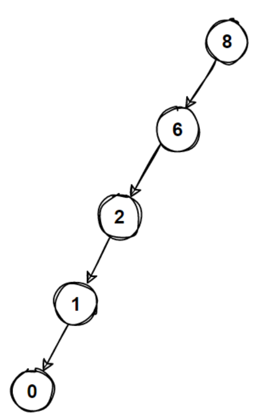
FACOM - UFU

Sumário

- Árvores Binárias Degeneradas, Cheias e Completas
- Balanceamento: Conceitos
- Árvore AVL: Definição
- Árvore AVL
    - Rotações
    - Implementação e operações básicas

Árvores Binárias Degeneradas

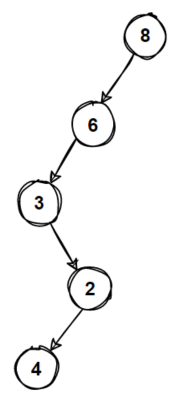
- As árvores binárias de pesquisa podem ser pouco recomendáveis para as operações básicas (inserção, remoção e busca) quando são estão balanceadas.
- Árvores desbalanceadas ao extremo são chamadas **degeneradas**.
- As árvores degeneradas tornam as operações básicas lentas em O(n).



Árvores Binárias Degeneradas

Árvore Binária Degenerada:

- Os nós internos apontam para uma única subárvore;
- Os elementos ficam numa sequência linear;
- Uma árvore de altura *h* tem *h+1* elementos.

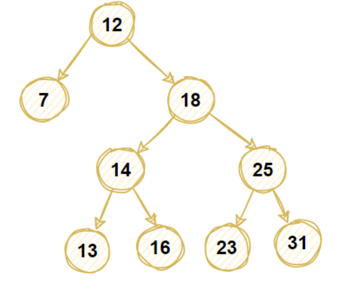


Áltura:

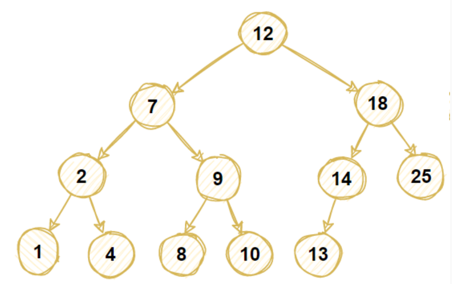
- A **altura** é uma métrica importante para a eficiência da árvore. Assim, para uma árvore binária com *n* elementos tem-se:
  - Altura proporcional a *log(n)*: árvore binária balanceada
  - Altura proporcional a *n*: árvore binária degenerada

Árvore Binária Cheia e Árvore Binária Completa

Árvore Binária Cheia: Todos os seus nós internos apontam para duas subárvores não-vazias.



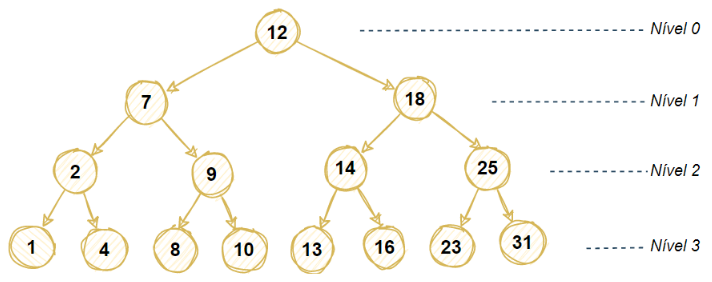
Árvore Binária Completa: Todos os seus níveis estão completamente preenchidos, com exceção do último, que tem seus elementos à esquerda.



Árvore Binária Cheia e Completa

Árvore Binária Cheia e Completa:

- Todos os seus nós internos apontam para duas subárvores não-vazias e todos os níveis estão completamente preenchidos.
- Quantidade de nós em uma árvore cheia e completa de altura h: $2^{h+1} - 1$



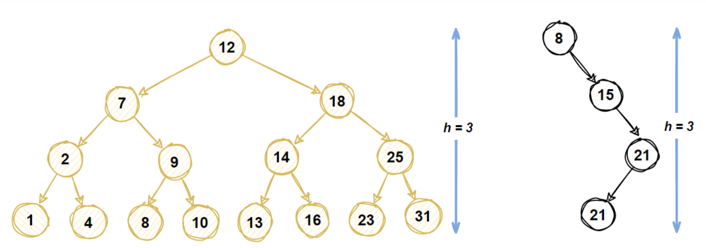
altura = 3 $\rightarrow$ *quantidade de elementos = 15*

Árvore Degenerada e Cheia/Completa: Altura

Função para calcular a **altura** de uma árvore binária:

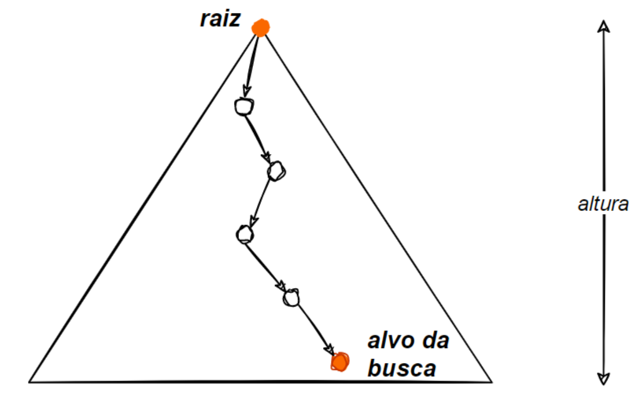
```
int max(int a, int b){
   return (a > b): a ? b;
}

int altura (Arv * a){
   if (arv_vazia(a))
      return -1; // raiz está no nível 0
   else
      return 1 + max(altura(a->esq), altura(a->dir));
}
```



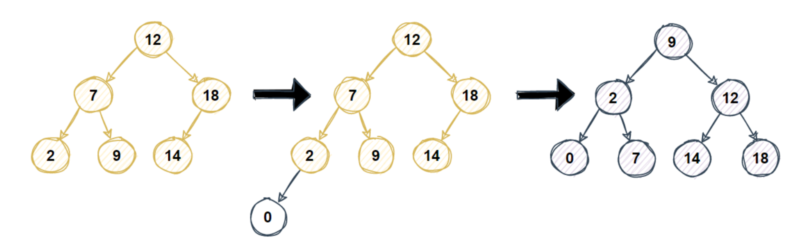
Balanceamento: Conceitos

O **balanceamento** é fundamental na manipulação de árvores de busca: o custo de acesso a um elemento depende do valor da altura da árvore.



Balanceamento: Conceitos

- Uma árvore binária completa com *n* elementos possui altura proporcional a *log(n)*, porém, após uma operação de inserção, a árvore pode perder esta característica.
- **Para tornar a árvore completa novamente**, uma solução seria aplicar um algoritmo de reorganização. Entretanto, o custo desta reorganização seria no mínimo proporcional a *n*, ou seja, *(O(n))*.



Na reorganização final, todos os nós foram movidos!!

Árvore AVL

- **Árvore AVL (Adelson-Velskii e Landis – 1962)** é uma árvore altamente balanceada, sendo que nas operações de inserção e remoção é executado um **rebalanceamento** tal que as alturas das sub-árvores esquerda e sub-árvores direita tenham alturas bem reguladas.
- É também chamada de árvore **balanceada pela altura**.
- Definição:
  - Numa árvore AVL as alturas das subárvores esquerda e direita de cada nó diferem no máximo por uma unidade.
  - Para definir o balanceamento é utilizado um cálculo específico:

    $\mathbf{FB}(\mathbf{v}) = \mathbf{h_e}(\mathbf{v}) - \mathbf{h_d}(\mathbf{v})$ onde:

    - $FB(v)$ é o fator de balanceamento para o nó v
    - $h_e(v)$ é a altura da subárvore da esquerda
    - $h_d(v)$ é a altura da subárvore da direita

Árvore AVL: Definição

- A informação armazenada em cada nó é do tipo inteiro;
- Cada nó da árvore guarda a sua **altura** em relação à raiz;
- As funções de criação e destruição são iguais àquelas da árvore binária.

```
typedef struct NO* ArvAVL;

struct NO {
    int info;
    int altura;
    struct NO *esq;
    struct NO *dir;
};
```

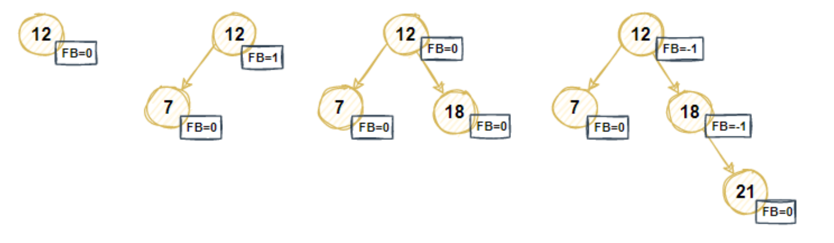
Árvore AVL

- Nós **balanceados** (ou regulados) são aqueles onde os valores do fator de balanceamento **FB** são: -1, 0 ou +1.

  -1 : sub-árvore direita mais alta que a esquerda

  0 : sub-árvore esquerda igual a da direita

  +1 : sub-árvore esquerda mais alta que a direita



Árvore AVL - Fator de Balanceamento - Função

Função que retorna o **fator de balanceamento** de um nó e função auxiliar:

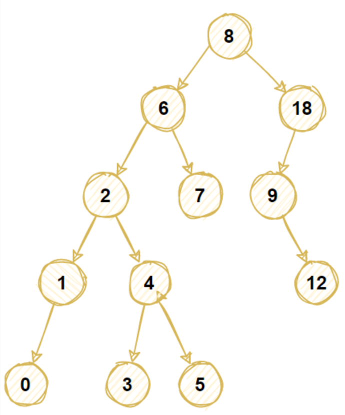
```
int altura_NO(struct NO* no){
    if(no == NULL)
        return -1;
    else
    return no->altura;
}

int fatorBalanceamento_NO(struct NO* no){
    return labs(altura_NO(no->esq) - altura_NO(no->dir));
}
```

Árvore AVL - Exercício

Complete a árvore com as informações:

1. Colocar o fator de balanceamento de cada nó.
2. Dizer se a árvore é AVL.
3. Verificar em quais posições para a inserção de elementos a árvore é AVL.



Árvore AVL - Modificações

Operações de inserção e remoção alteram o balanceamento da árvore, e portanto, é necessário executar a reorganização dos nós, de forma a manter as propriedades da árvore AVL:

- O percurso em ordem fica inalterado em relação a árvore desbalanceada, ou seja, a árvore continua a ser uma árvore de busca binária.
- A árvore modificada passa de desbalanceada para um estado de balanceamento.
- A reorganização dos elementos é baseada em operações de **rotação**.

Árvore AVL - Rotações

Uma operação de rotação altera o balanceamento de uma árvore binária, garantindo a propriedade AVL e a sequência de percurso em ordem.

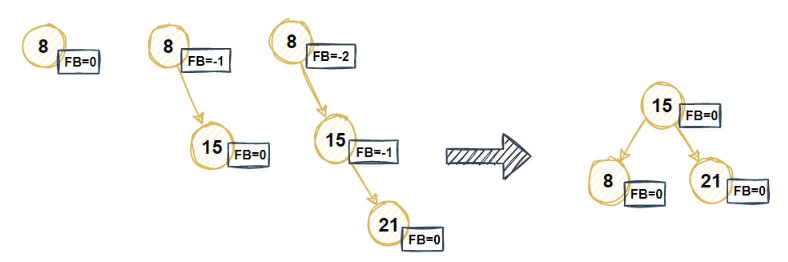
Podem-se definir 4 tipos diferentes de rotação:

- Rotação à esquerda
- Rotação à direita
- Rotação dupla à esquerda
- Rotação dupla à direita

Árvore AVL - Rotação à Esquerda

Rotação à esquerda: Exemplo

- Inserir 8
- Inserir 15
- Inserir 21

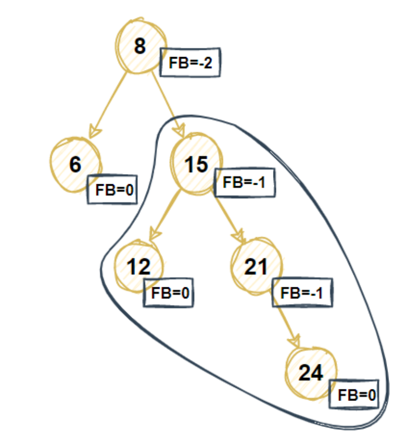


Após a rotação à esquerda, a árvore ficou balanceada e o percurso *em-ordem* permanece o mesmo: 8, 15, 21

Árvore AVL - Rotação à Esquerda

Passos da rotação à esquerda:

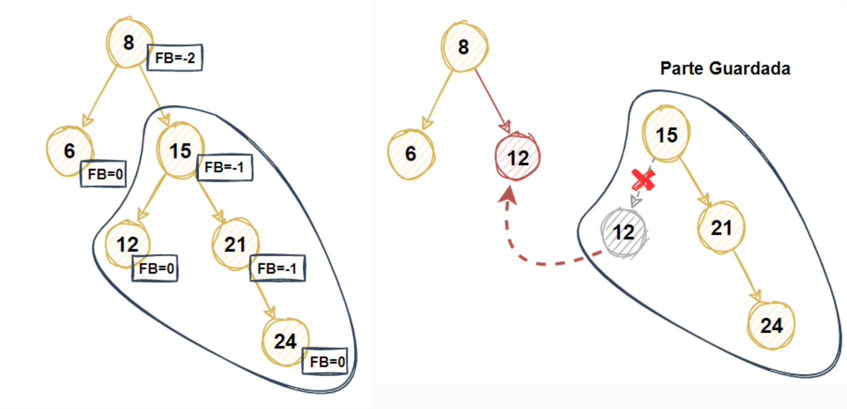
1. Separar e guardar a subárvore da direita.



Árvore AVL - Rotação à Esquerda

Passos da rotação à esquerda:

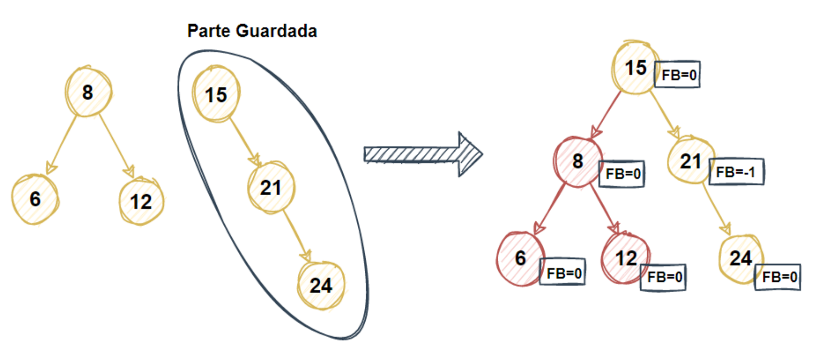
2. Trocar a subárvore da direita pela subárvore esquerda da parte guardada.



Árvore AVL - Rotação à Esquerda

Passos da rotação à esquerda:

3. Colocar na subárvore esquerda da parte guardada a árvore inicial
4. Verificar o balanceamento

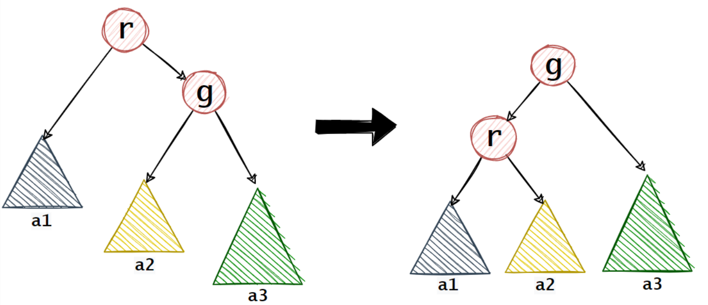


Árvore AVL - Rotação à Esquerda - Quando aplicar?

Cenário para aplicação da rotação à esquerda (ou Rotação RR):

$h_d(r) > h_e(r)$

$h_d(g) > h_e(g)$



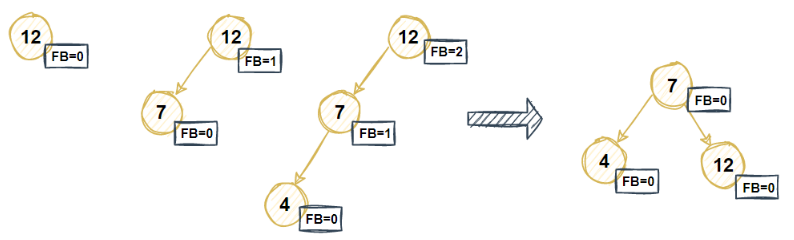
Implementação:

```
int maior(int x, int y){
    if(x > y) return x;
    else  return y;
}
void RotacaoEsquerda(ArvAVL *raiz){
    struct NO *G;
    G = (*raiz)->dir;
    (*raiz)->dir = G->esq;
    G->esq = (*raiz);
    (*raiz)->altura = maior(altura_NO((*raiz)->esq),
                                  altura_NO((*raiz)->dir)) + 1;
    G->altura = maior(altura_NO(G->dir),(*raiz)->altura) + 1;
    (*raiz) = G;
}
```

Árvore AVL - Rotação à Direita

Rotação à direita: Exemplo

- Inserir 12
- Inserir 7
- Inserir 4

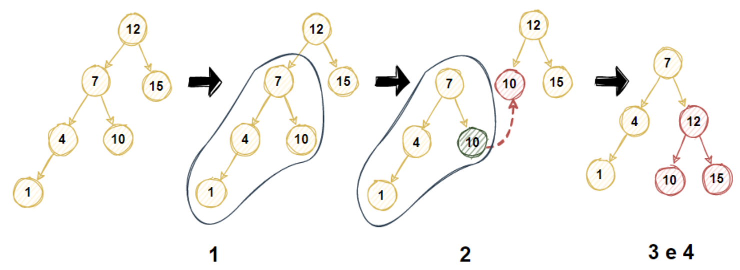


Após a rotação à direita, a árvore ficou balanceada e o percurso *em-ordem* permanece o mesmo: 4, 7, 12

Árvore AVL - Rotação à Direita

A **rotação à direita** segue os mesmos passos da rotação à esquerda, só que em direção oposta:

1. Separar e guardar a subárvore da esquerda.
2. Trocar a subárvore da esquerda pela subárvore direita da parte guardada.
3. Colocar na subárvore direita da parte guardada a árvore inicial.
4. Verificar o balanceamento.

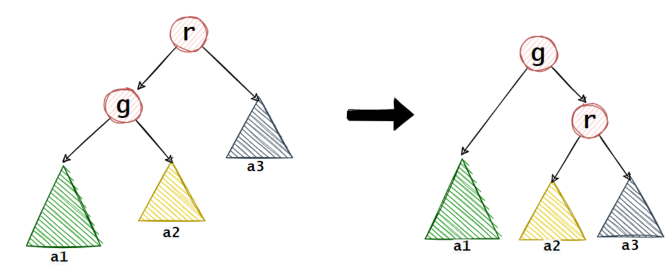


Árvore AVL - Rotação à Direita - Quando aplicar?

Cenário para aplicação da rotação à direita (ou Rotação LL):

$$h_e(r) > h_d(r)$$

$$h_e(g) > h_d(g)$$



Árvore AVL - Rotação à Direita - Implementação

Implementação:

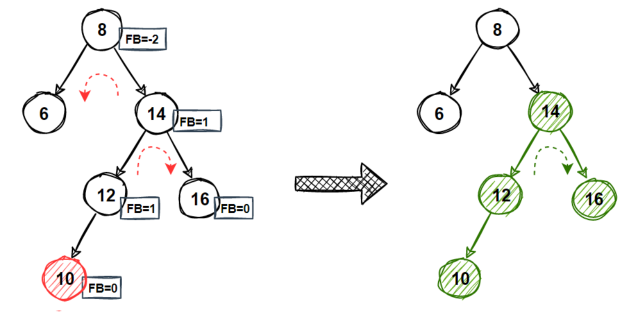
```
void RotacaoDireita(ArvAVL *raiz){
    struct NO *G;
    G = (*raiz)->esq;
    (*raiz)->esq = G->dir;
    G->dir = *raiz;
    (*raiz)->altura = maior(altura_NO((*raiz)->esq), altura_NO((*raiz)->dir)) + 1;
    G->altura = maior(altura_NO(G->esq), (*raiz)->altura) + 1;
    *raiz = G;
}
```

Árvore AVL - Rotação Dupla à Esquerda

Rotação Dupla à Esquerda: Exemplo

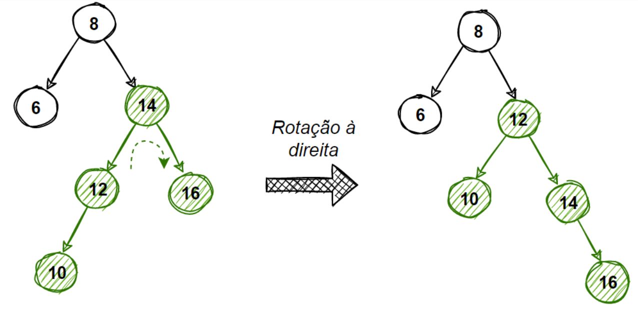
- Inserir 10
- A inserção do 10 afeta o balanceamento do elemento 8 (FB=-2)
- Serão necessárias **duas rotações**: uma à direita e outra à esquerda

A primeira operação a ser feita é a **rotação à direita** da subárvore direita ao elemento 8:



Árvore AVL - Rotação Dupla à Esquerda

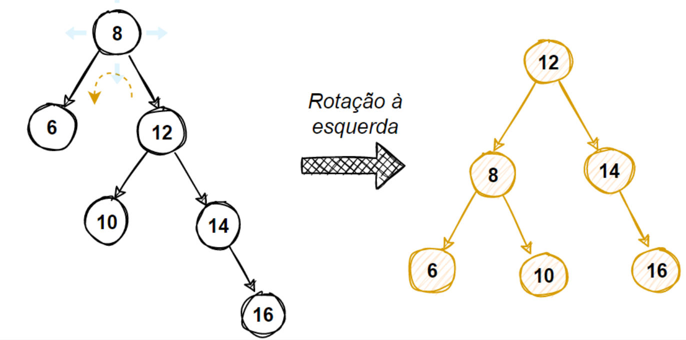
Resultado da operação de **rotação à direita** da subárvore direita ao elemento 8:



Em seguida, deve-se efetuar a **rotação à esquerda** na subárvore do elemento 8.

Árvore AVL - Rotação Dupla à Esquerda

Resultado da operação de **rotação à esquerda** da subárvore do elemento 8:

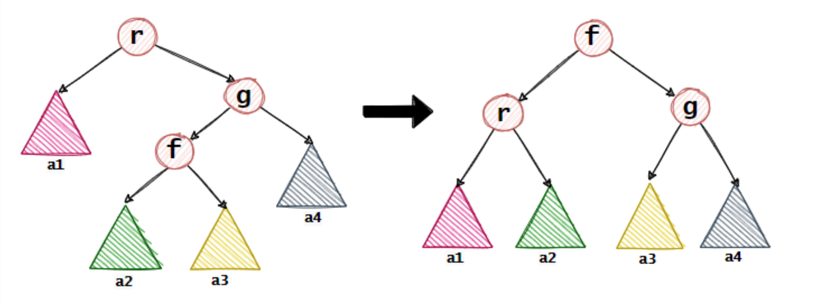


Como resutado, tem-se que a árvore resultante é uma AVL.

Cenário para aplicação da rotação dupla à esquerda (ou Rotação RL):

$h_d(r) > h_e(r)$

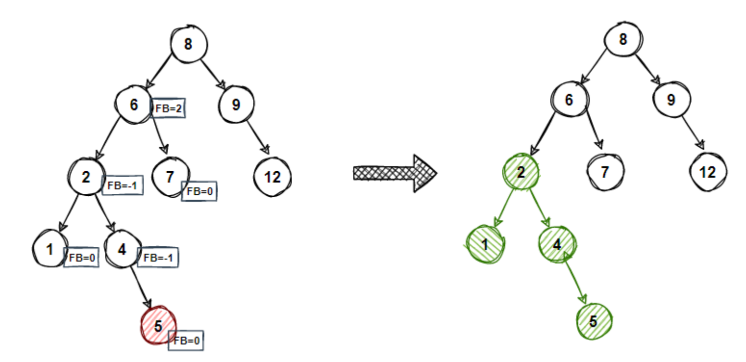
$h_e(g) > h_d(g)$



Rotação Dupla à Direita: Exemplo

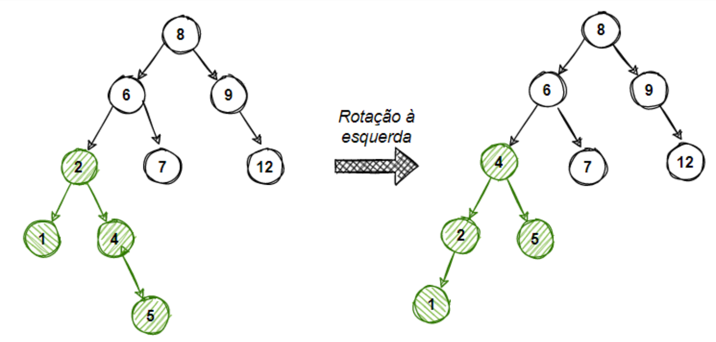
- Inserir 5
- A inserção do 5 afeta o balanceamento do elemento 6 (FB=2)
- Serão necessárias **duas rotações**: uma à esquerda e outra à direita

A primeira operação a ser feita é a **rotação à esquerda** da subárvore esquerda do elemento 6:



Árvore AVL - Rotação Dupla à Direita

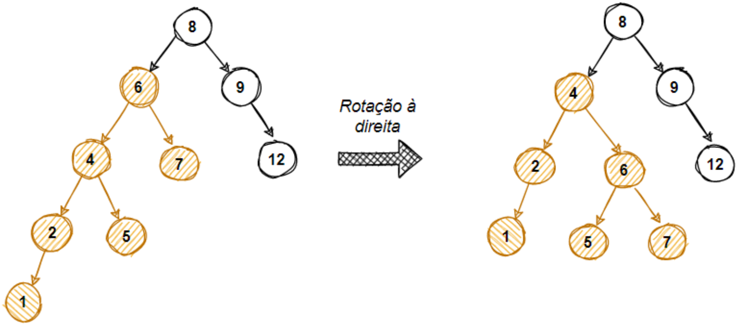
Resultado da operação de **rotação à esquerda** da subárvore à esquerda do elemento 6:



Em seguida, deve-se efetuar a **rotação à direita** na subárvore do elemento 6.

Árvore AVL - Rotação Dupla à Direita

Resultado da operação de **rotação à direita** da subárvore do elemento 6:



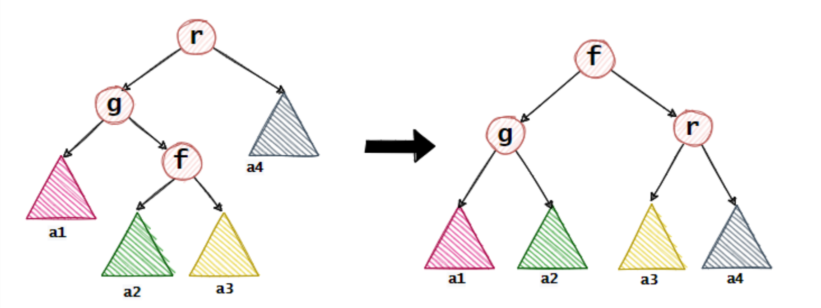
Como resutado, tem-se que a árvore resultante é uma AVL.

Árvore AVL - Rotação Dupla à Direita - Quando aplicar?

Cenário para aplicação da rotação dupla à direita (ou Rotação LR):

$$\mathbf{h_e(r)} > \mathbf{h_d(r)}$$

$$\mathbf{h_d(g)} > \mathbf{h_e(g)}$$



Árvore AVL - Rotações Duplas - Implementação

Implementação das **rotações duplas à esquerda e à direita**:

```
void RotacaoDuplaEsquerda(ArvAVL *A){//RL
    RotacaoDireita(&(*A)->dir);
    RotacaoEsquerda(A);
}

void RotacaoDuplaDireita(ArvAVL *A){//LR
    RotacaoEsquerda(&(*A)->esq);
    RotacaoDireita(A);
}
```

Árvore AVL - Inserção de um novo elemento

Para **inserir** um novo valor *v* na árvore, devem ser considerados os casos/ações:

- Se a raiz é igual a NULL: criar o nó e inserir *v*.
- Se *v* é menor do que a raiz: caminhar para a subárvore esquerda e **reiniciar** a inserção.
- Se *v* é maior que a raiz: caminhar para a subárvore direita e **reiniciar** a inserção.

A aplicação recursiva do método permite alcançar uma posição nula na árvore, local em que o novo valor *v* será inserido.

Árvore AVL - Balanceamento após a inserção de um novo elemento

Após a inserção de um novo valor na árvore AVL, deve-se:

- Retroceder no caminho da inserção, verificando o **fator de balanceamento** de cada um dos nós visitados;
- Executar, a depender do fator de balanceamento obtido para um nó (+2 ou -2), a **rotação adequada** para a reorganização da sua subárvore.

Árvore AVL - Função de Inserção

Inserção: Caso em que a raiz é nula

```
int insere_ArvAVL(ArvAVL *raiz, int valor){
    int res;
    if(*raiz == NULL){//árvore vazia ou nó folha
        struct NO *novo;
        novo = (struct NO*) malloc (sizeof(struct NO));
        if(novo == NULL) return 0;

        novo->info = valor;
        novo->altura = 0;
        novo->esq = NULL;
        novo->dir = NULL;
        *raiz = novo;
        return 1;
    }
    //... continua
```

Árvore AVL - Função de Inserção

Inserção: Caso em que o valor é menor que o contido na raiz

```
struct NO *atual = *raiz;
if(valor < atual->info){
    res = insere_ArvAVL(&(atual->esq), valor);
    if(res == 1){
        if(fatorBalanceamento_NO(atual) >= 2){
            if(valor < (*raiz)->esq->info )
                RotacaoDireita(raiz);
            else
                RotacaoDuplaDireita(raiz);
        }
    }
}
//... continua
```

Árvore AVL - Função de Inserção

Inserção: Caso em que o valor é maior que o contido na raiz

```
 else{
  if(valor > atual->info){
       res = insere_ArvAVL(&(atual->dir), valor)
       if(res == 1){
            if(fatorBalanceamento_NO(atual) >= 2){
                 if((*raiz)->dir->info < valor)
                      RotacaoEsquerda(raiz);
                 else
                      RotacaoDuplaEsquerda(raiz);
            }
       }
  }
  else{
       printf("Valor duplicado!!");
       return 0;
  }
 }
 atual->altura = maior(altura_NO(atual->esq),altura_NO(atual->dir)) + 1;
 return res;
}
```

Árvore AVL - Remoção

A **remoção** em árvores AVL é similar à remoção em uma árvore binária de busca. Entretanto, pode ser necessário o rebalanceamento da parte afetada pela remoção.

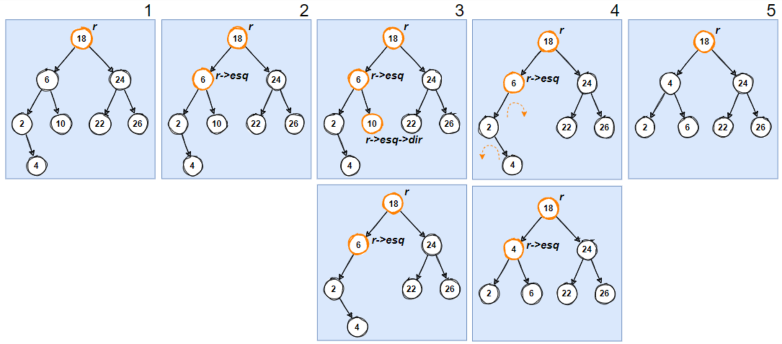
Na **remoção** de um valor *v*, encontrado no nó *p* na árvore, devem ser considerados os casos/ações:

- Se *p* possui **nenhum filho**: remover *p*, rebalancear nós antecessores afetados, se necessário.
- Se *p* possui **um filho**: subir este filho, remover *p*, rebalancear antecessores, se necessário.
- Se *p* possui **dois filhos**:
  - Encontrar o menor na subárvore à direita e substituir seu valor em *p*;
  - Executar a remoção do menor na subárvore à direita de *p* (este nó possuirá um ou nenhum filho);
  - Após a remoção do menor à direita, rebalancear *p*, caso necessário.

Árvore AVL - Remoção - Exemplo

Remoção de um nó sem nenhum filho:

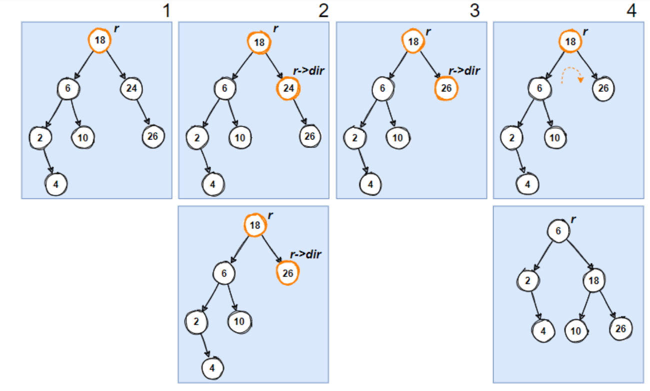
1. Remover 10 em r
2. Remover 10 em $r \rightarrow esq$
3. Remover 10 em $r \rightarrow esq \rightarrow dir$ ▶ Destruir nó 10 e retornar NULL
4. Rebalancear $r \rightarrow esq$ ▶ Rotação dupla à direita
5. Não será necessário rebalancear r



Árvore AVL - Remoção - Exemplo

Remoção de um nó com um filho:

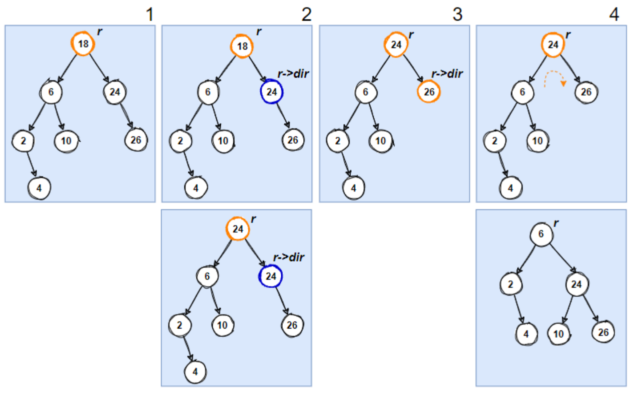
1. Remover 24 em *r*
2. Remover 24 em $r \to dir$ ▶ Subir filho à direita
3. Não será necessário rebalancear $r \to dir$
4. Rebalancear *r* ▶ Rotação simples à direita



Árvore AVL - Remoção - Exemplo

Remoção de um nó com dois filhos:

1. Remover 18 em r
2. Buscar menor elemento em $r \rightarrow dir$ ▶ Substituir menor em r
3. Remover menor elemento de $r \rightarrow dir$ ▶ Remoção de nó com um filho
4. Rebalancear r ▶ Rotação simples à direita

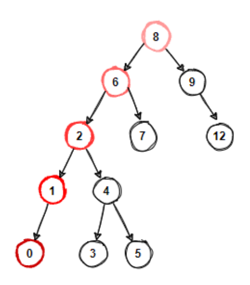


Árvore AVL - Função para Buscar Menor Elemento

Na remoção de um nó contendo duas subárvores (esquerda e direita), utiliza-se uma função para **buscar o menor elemento** na subárvore à direita.

Função **Busca Menor**: Percurso sempre à esquerda

```
struct NO* buscaMenor(struct NO* raiz){
    struct NO *atual, *prox;
    if (estaVazia_ArvAVL(&raiz)) return NULL;
    atual = raiz;
    prox = raiz->esq;
    while(prox != NULL){
        atual = prox;
        prox = prox->esq;
    }
    return atual;
}
```



Árvore AVL - Função para Remover um Elemento

```
int remove_ArvAVL(ArvAVL *raiz, int valor){
    int r;
    if(*raiz == NULL){          // valor não existe
        printf("valor não encontrado!!");
        return 0;
    }
    if(valor < (*raiz)->info){     // buscar elemento à esquerda
        r = remove_ArvAVL(&(*raiz)->esq,valor);
        if(r == 1){
            if(fatorBalanceamento_NO(*raiz) >= 2){
                if(altura_NO((*raiz)->dir->esq) <= altura_NO((*raiz)->dir->dir))
                    RotacaoEsquerda(raiz);
                else
                    RotacaoDuplaEsquerda(raiz);
            }
        }
    }
```

Árvore AVL - Função para Remover um Elemento

```
 if((*raiz)->info < valor){      // buscar elemento à direita
    r = remove_ArvAVL(&(*raiz)->dir, valor);
    if(r == 1){
        if(fatorBalanceamento_NO(*raiz) >= 2){
            if(altura_NO((*raiz)->esq->dir) <= altura_NO((*raiz)->esq->esq) )
                RotacaoDireita(raiz);
            else
                RotacaoDuplaDireita(raiz);
        }
    }
}
 continua .... // Elemento encontrado

(*raiz)->altura = maior(altura_NO((*raiz)->esq),
                            altura_NO((*raiz)->dir)) + 1;
return r;
```

Árvore AVL - Função para Remover um Elemento

```
if((*raiz)->info == valor){     // elemento encontrado
    if(((*raiz)->esq == NULL || (*raiz)->dir == NULL)){ // 1 filho ou nenhum
        struct NO *raizAntiga = (*raiz);
        if((*raiz)->esq != NULL)
            *raiz = (*raiz)->esq;
        else
            *raiz = (*raiz)->dir;
        free(raizAntiga);
    }
    else {                          // nó tem 2 filhos
        struct NO* temp = buscaMenor((*raiz)->dir);
        (*raiz)->info = temp->info;
        remove_ArvAVL(&(*raiz)->dir, (*raiz)->info);
        if (fatorBalanceamento_NO(*raiz) >= 2)
            if(altura_NO((*raiz)->esq->dir) <= altura_NO((*raiz)->esq->esq))
                RotacaoDireita(raiz);
            else
                RotacaoDuplaDireita(raiz);
    }
    if (*raiz != NULL)
        (*raiz)->altura = maior(altura_NO((*raiz)->esq),
                                altura_NO((*raiz)->dir)) + 1;
    return 1;
}
```

Exercícios

1. Sejam as sequências de números inteiros abaixo. Para cada uma, crie uma árvore AVL fazendo a inserção de cada elemento e as rotações necessárias. Faça no caderno, sem utilizar computador.

   (p1): 10, 9, 8, 7, 6, 5, 4, 3, 2, 1

   (p2): 5, 12, 7, 2, 15, 8, 28, 29, 45, 1, 56

2. Dado um valor *v*, faça uma função para eliminar todos os valores menores que *v* em uma árvore AVL.
3. Dadas duas árvores AVL, faça uma função que verifica se elas são idênticas.

Referência e Material Extra

- Celes, W.; Cerqueira, R.; Rangel, J.L. *Introdução a Estruturas de Dados com Técnicas de Programação em C*. 2a. ed. Elsevier, 2016.

* Backes, A. *Programação Descomplicada - Estruturas de Dados*.
    - Vídeo-aulas 79 a 84: *https://programacaodescomplicada.wordpress.com/indice/estrutura-de-dados/*