Estruturas de Dados

Aula 10 – Árvores AVL

Prof. Ana Carolina Sokolonski

Bacharelado em Sistemas de Informação
Instituto Federal da Bahia – Campus Feira de Santana

2026

Agenda

Motivação

Definição

Balanceamento

Inserção

Remoção

Percursos

Trabalho

Motivação

Por que precisamos de AVL?

Revisão: Árvore Binária de Busca (ABB)

Propriedade da ABB

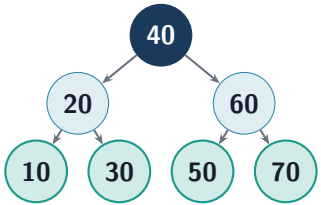
Para todo nó u com valor k:

- Todos os nós da subárvore **esquerda** $< k$
- Todos os nós da subárvore **direita** $> k$

Busca, inserção e remoção: $O(h)$, onde h é a **altura**.

O problema

A altura h depende da **ordem de inserção**.
Caso médio: $O(\log n)$ Pior caso: $O(n)$



Inserção: 40, 20, 60, 10, 30, 50, 70
Árvore equilibrada, $h = 2$

Árvore Degenerada — O Problema

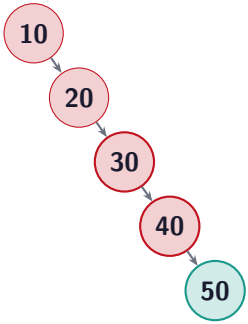
▪ Árvore Degenerada

Quando os elementos são inseridos em ordem crescente ou decrescente em uma ABB, a árvore degenera em uma **lista encadeada**. A busca passa de $O(\log N)$ para $O(N)$!

▪ Solução: Árvore AVL

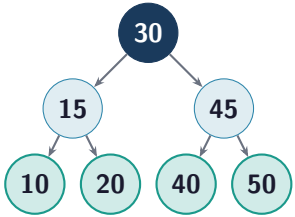
Mantém a árvore **balanceada** após cada inserção/remoção, garantindo uma busca em $O(\log N)$ sempre.

Degenerada
(inserção 10,20,30,40,50)



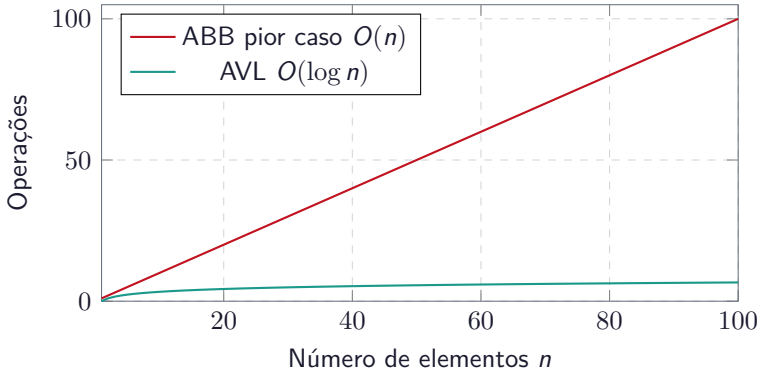
$h = 4, \quad O(N)$

Balanceada (AVL)
(mesmos elementos)



$h = 2, \quad O(\log N)$

Comparação de Desempenho



Para $n = 100$: ABB pior caso requer **100** comparações; AVL requer apenas $\approx$**7**.

Árvores AVL

Adelson-Velsky e Landis, 1962

O que é uma Árvore AVL?

Definição

ABB **balanceada**: para qualquer nó u, alturas das subárvores diferem em no máximo **1**.

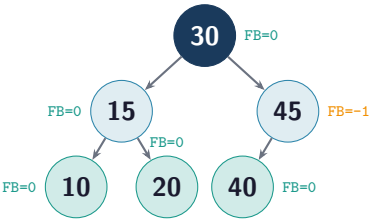
Complexidade

Busca, Inserção e Remoção: $O(\log n)$ **garantido**

Fator de Balanço (FB)

$\text{FB}(u) = h_{\text{dir}}(u) - h_{\text{esq}}(u)$

Nó **regulado**: $\text{FB}(u) \in \{-1,\ 0,\ +1\}$
$|\text{FB}| > 1 \Rightarrow$ árvore **não é AVL** $\Rightarrow$ rotacionar



- $|\text{FB}| \leq 1$: regulado (AVL)
- $|\text{FB}| > 1$: rebalancear!

Definição de altura

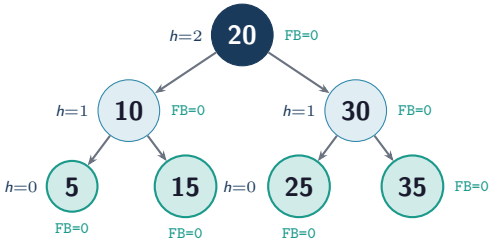
A **altura** de um nó é o comprimento do maior caminho da raiz até uma folha em sua subárvore.

$h(\text{folha}) = 0$
$h(\text{NULL}) = -1$
$h(u) = 1 + \max(h_e, h_d)$

Calculando o FB

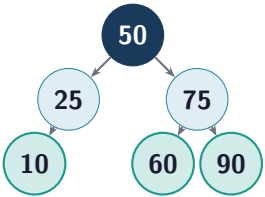
$\text{FB}(u) = h_{\text{dir}} - h_{\text{esq}}$

Se $h_{\text{dir}} = 2$ e $h_{\text{esq}} = 0$:
$\text{FB} = 2 - 0 = +2$ (desbalanceado!)



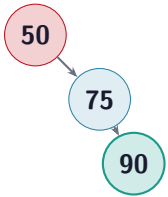
Identificando Árvores AVL e não-AVL

AVL válida



Todos $|\text{FB}| \leq 1$

Não é AVL



FB(50)=+2: rotacionar!

Regra de identificação

1. Calcule a altura de cada subárvore
2. Compute $\text{FB} = h_d - h_e$
3. Se algum $|\text{FB}| > 1$: **não é AVL**
4. Aplique a rotação adequada

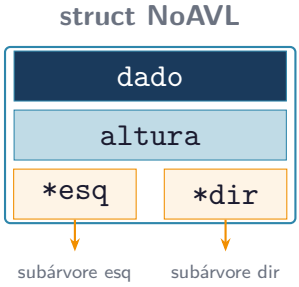
Lembre-se

$h(\text{NULL}) = -1$
Folha: $\text{FB} = (-1) - (-1) = 0$

Estrutura do Nó AVL em C

no_avl.h — Definição e funções auxiliares

```c
typedef struct NoAVL {
    int dado;
    int altura;//altura do nó
    struct NoAVL *esq;
    struct NoAVL *dir;
}NoAVL;
int altura(NoAVL *n) {
    if (n == NULL) return -1;
    return n->altura;
}//Retorna altura (-1 se NULL)
int fatorBalanco(NoAVL *n) {
    if (n == NULL) return 0;
    return altura(n->dir)-altura(n->esq
        );
}//Retorna fator de balanço
void atualizaAltura(NoAVL *n) {
    int he = altura(n->esq);
    int hd = altura(n->dir);
    n->altura = 1+(he > hd ? he : hd);
}//Atualiza altura do nó
```



- **Por quê altura e não fator?**

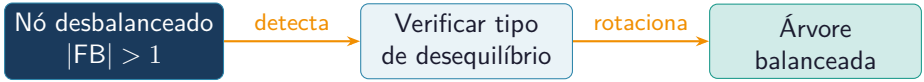
Armazenar a **altura** é mais flexível: o Fator de Balanceamento (FB), pode ser calculado a qualquer momento. Algumas implementações guardam o FB diretamente em $(-1, 0, +1)$.

Balanceamento

Rotações Simples e Duplas

Os 4 Tipos de Rotação

Rotação	Quando usar	FB nó	FB filho
Simples Esquerda (LL→R)	Subárvore **dir** alta	$+2$	≥ 0
Simples Direita (RR→L)	Subárvore **esq** alta	-2	≤ 0
Dupla Esq-Dir (RL)	Filho dir, subárv **esq** alta	$+2$	< 0
Dupla Dir-Esq (LR)	Filho esq, subárv **dir** alta	-2	> 0



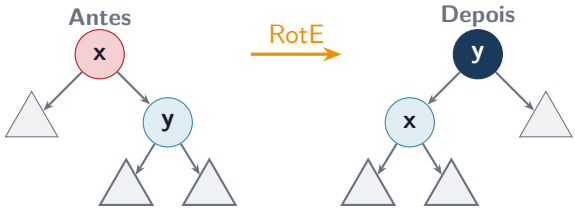
Rotação Simples à Esquerda (RotE)

Quando?

FB do nó $= +2$ **e** FB do filho direito ≥ 0.
A subárvore direita é mais alta.

Algoritmo

1. Filho direito y sobe para a posição de x
2. x desce para filho esquerdo de y
3. Subárvore esq. de y passa para dir. de x
4. Atualizar alturas (x primeiro, depois y)



A, B, C são subárvores genéricas (triângulos)

Rotação Simples à Esquerda — Código C

rotacao_esq.c

```c
NoAVL* rotacaoEsquerda(NoAVL *x) {
    NoAVL *y  = x->dir;  /* y eh o filho direito de x */
    NoAVL *T2 = y->esq;  /* subarvore esquerda de y (B) */
    /* Rotacao: y sobe, x desce */
    y->esq = x;
    x->dir = T2;
    /* Atualizar alturas (x primeiro, pois agora eh filho de y) */
    atualizaAltura(x);
    atualizaAltura(y);
    return y;   /* nova raiz do subconjunto */
}
```

▪ Exemplo concreto

Inserção de 10, 20, 30 em uma ABB:
Após inserir 30: FB(10)=+2, FB(20)=+1
$\Rightarrow$ RotE em 10 $\Rightarrow$ raiz vira 20

▪ Complexidade

A rotação é $O(1)$ — apenas redirecionamento de ponteiros. A altura é atualizada em $O(1)$ porque está armazenada no próprio nó.

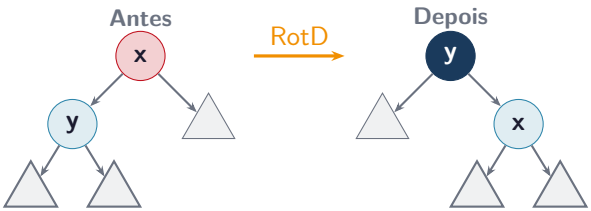
Rotação Simples à Direita (RotD)

Quando?

FB do nó $= -2$ **e** FB do filho esquerdo ≤ 0.
A subárvore esquerda é mais alta.

Algoritmo

1. Filho esquerdo y sobe para a posição de x
2. x desce para filho direito de y
3. Subárvore dir. de y passa para esq. de x
4. Atualizar alturas (x primeiro, depois y)



Simétrica à rotação à esquerda

Rotação Dupla Esquerda-Direita (RotED)

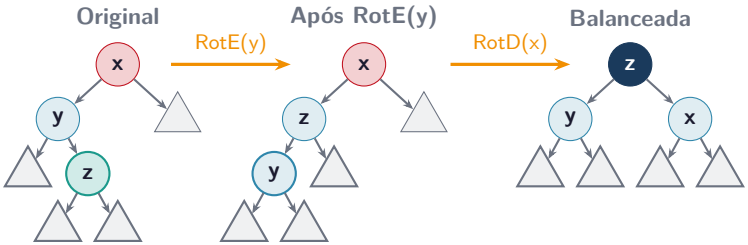
Quando?

FB do nó $= -2$ **e** FB do filho esquerdo > 0.
O "joelho" está dobrado para a direita.

Algoritmo

É uma composição de duas rotações simples:

1. **RotE** no filho esquerdo de x
2. **RotD** no próprio nó x



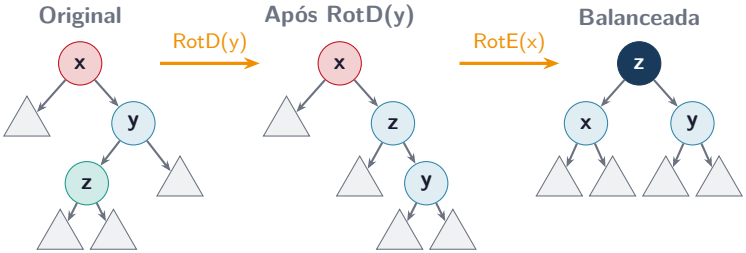
Rotação Dupla Direita-Esquerda (RotDE)

Quando?

FB do nó $= +2$ **e** FB do filho direito < 0.
O "joelho" está dobrado para a esquerda.

Algoritmo

Composição simétrica:
1. **RotD** no filho direito de x
2. **RotE** no próprio nó x



Inserção na AVL

Inserir e rebalancear

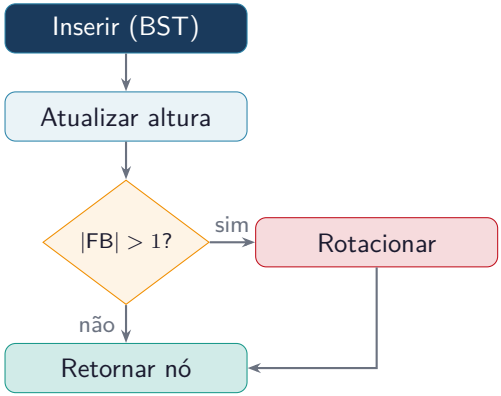
Algoritmo de Inserção

Passos

1. Inserir como em uma **ABB** normal
2. Na volta da recursão, **atualizar altura**
3. Calcular o **fator de balanço**
4. Se $|FB| > 1$: aplicar a **rotação adequada**
5. Retornar a nova raiz do subconjunto

Importante

O rebalanceamento sobe pela árvore por recursão. Apenas **uma** rotação (simples ou dupla) é necessária por inserção em uma AVL.



Inserção AVL — Código C — Parte 1

inserir_avl.c — Parte 1

```c
NoAVL* inserir(NoAVL *raiz, int valor){
    if (raiz == NULL){
        NoAVL *novo = (NoAVL*) malloc(sizeof(NoAVL));
        novo->dado = valor;
        novo->altura = 0;
        novo->esq = novo->dir = NULL;
        return novo;
    }
    if (valor < raiz->dado)
        raiz->esq = inserir(raiz->esq, valor);
    else if (valor > raiz->dado)
        raiz->dir = inserir(raiz->dir, valor);
    else return raiz;  /* duplicatas ignoradas */
    atualizaAltura(raiz);
    int fb = fatorBalanco(raiz);
/* continua no proximo slide... */
```

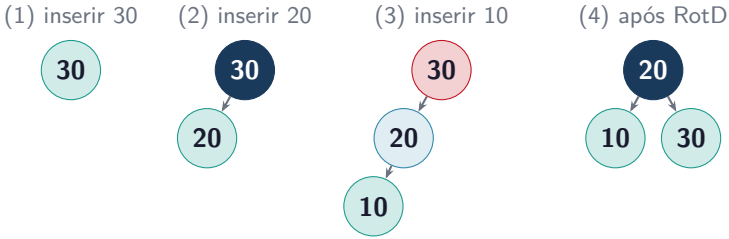
Inserção AVL — Código C — Parte 2

inserir_avl.c — Parte 2

```c
/* continuacao da funcao inserir na AVL */
    if (fb > 1 && valor > raiz->dir->dado)
        return rotacaoEsquerda(raiz);
    if (fb < -1 && valor < raiz->esq->dado)
        return rotacaoDireita(raiz);
    if (fb > 1 && valor < raiz->dir->dado) {
        raiz->dir = rotacaoDireita(raiz->dir);
        return rotacaoEsquerda(raiz);
    }
    if (fb < -1 && valor > raiz->esq->dado) {
        raiz->esq = rotacaoEsquerda(raiz->esq);
        return rotacaoDireita(raiz);
    }
    return raiz;
}
```

Exemplo de Inserção Passo a Passo

Inserindo: 30, 20, 10



$\Rightarrow$ Rotação à Direita!

Remoção na AVL

Remover e rebalancear

Algoritmo de Remoção

Passos

1. Remover como em uma **ABB** normal
2. Na volta da recursão: atualizar altura
3. Calcular o fator de balanço
4. Se $|FB| > 1$: aplicar rotação

Diferença da inserção

Na remoção podem ser necessárias **múltiplas** rotações ao longo do caminho de volta à raiz, diferente da inserção onde uma única rotação basta sempre.

Caso especial: 2 filhos

Substituir o nó pelo seu **sucessor em-ordem** (menor da subárvore direita) ou pelo **antecessor em-ordem** (maior da subárvore esquerda), depois remover o substituto.

Complexidade

Remoção: $O(\log n)$
Número de rotações: $O(\log n)$

Remoção AVL — Código C — Parte 1

remover_avl.c — Parte 1

```c
NoAVL* menorNo(NoAVL *n) {
    while (n->esq != NULL) n = n->esq;
    return n;
}
NoAVL* remover(NoAVL *raiz, int valor) {
    if (raiz == NULL) return NULL;
    if (valor < raiz->dado)
        raiz->esq = remover(raiz->esq, valor);
    else if (valor > raiz->dado)
        raiz->dir = remover(raiz->dir, valor);
    else {
        if (raiz->esq == NULL || raiz->dir == NULL) {
            NoAVL *tmp = raiz->esq ? raiz->esq : raiz->dir;
            free(raiz);  return tmp;
        }
        NoAVL *suc = menorNo(raiz->dir);
        raiz->dado = suc->dado;
        raiz->dir  = remover(raiz->dir, suc->dado);
    }
/* continua no proximo slide... */
```

Remoção AVL — Código C — Parte 2

remover_avl.c — Parte 2

```c
/* continuacao da funcao remover na AVL */
    atualizaAltura(raiz);
    int fb = fatorBalanco(raiz);
    if (fb > 1 && fatorBalanco(raiz->dir) >= 0)
        return rotacaoEsquerda(raiz);
    if (fb < -1 && fatorBalanco(raiz->esq) <= 0)
        return rotacaoDireita(raiz);
    if (fb > 1 && fatorBalanco(raiz->dir) < 0) {
        raiz->dir = rotacaoDireita(raiz->dir);
        return rotacaoEsquerda(raiz);
    }
    if (fb < -1 && fatorBalanco(raiz->esq) > 0) {
        raiz->esq = rotacaoEsquerda(raiz->esq);
        return rotacaoDireita(raiz);
    }
    return raiz;
}
```

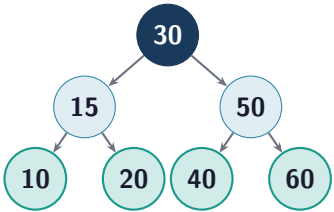
Percursos na AVL

Pré-ordem, Em-ordem e Pós-ordem

Percursos em Árvores AVL

percursos.c

```c
/* Pré-ordem: Raiz -> Esq -> Dir */
void preOrdem(NoAVL *r) {
    if (r == NULL) return;
    printf("%d ", r->dado);
    preOrdem(r->esq);
    preOrdem(r->dir);
}
/* Em-ordem: Esq -> Raiz -> Dir */
void emOrdem(NoAVL *r) {
    if (r == NULL) return;
    emOrdem(r->esq);
    printf("%d ", r->dado);
    emOrdem(r->dir);
}
/* Pós-ordem: Esq -> Dir -> Raiz */
void posOrdem(NoAVL *r) {
    if (r == NULL) return;
    posOrdem(r->esq);
    posOrdem(r->dir);
    printf("%d ", r->dado);
}
```



Saídas

Pré: 30 15 10 20 50 40 60
Em: 10 15 20 30 40 50 60
Pós: 10 20 15 40 60 50 30

Em-ordem sempre produz a sequência **ordenada**!

Exercício

Implementação da Árvore AVL

Implementação — Menu de Operações

Implemente a Árvore AVL com o seguinte menu

1. Inserir elemento na Árvore AVL
2. Mostrar em **Pré-ordem**
3. Mostrar em **Pós-ordem**
4. Mostrar **Em-Ordem**
5. **Desenhar** a Árvore AVL da raiz às folhas
6. **Remover** elemento da Árvore AVL
7. **Buscar** elemento na Árvore AVL
8. Fechar

Atenção

Toda inserção/remoção deve verificar o **fator de balanço** e aplicar a rotação adequada quando necessário.

Dica: Desenhando a Árvore no Terminal

desenhar.c

```c
/* Imprime arvore de lado (rotac. 90) */
void desenhar(NoAVL *raiz, int nivel)
    {
    if (raiz == NULL) return;
    desenhar(raiz->dir, nivel + 1);
    int k;
    for (k = 0; k < nivel; k++)
        printf("    ");
    printf("[%d] FB=%d\n",
           raiz->dado,
           fatorBalanco(raiz));
    desenhar(raiz->esq, nivel + 1);
}
/* Chamada inicial: desenhar(raiz, 0); */
```

Exemplo de saída

```
    [60] FB=0
  [50] FB=0
    [40] FB=0
 [30] FB=0
  [15] FB=0
    [20] FB=0
    [10] FB=0
```

Como ler

A árvore aparece **deitada à esquerda**. O nó mais à esquerda na tela é a **raiz**. Filhos direitos aparecem acima, filhos esquerdos abaixo.

Resumo — Conceitos e Rotações

Conceitos

- ABB pode degenerar em $O(n)$ sem balanceamento
- AVL garante $O(\log n)$ para busca, inserção, remoção
- Fator de Balanço: $h_d - h_e \in \{-1, 0, +1\}$
- Altura armazenada no nó para cálculo $O(1)$

Rotações

- Simples Esq. (FB=+2, filho FB≥0)
- Simples Dir. (FB=-2, filho FB≤0)
- Dupla ED (FB=-2, filho FB>0)
- Dupla DE (FB=+2, filho FB<0)

Operações

- Inserção: BST + rotação na volta da recursão
- Remoção: BST + possível sequência de rotações
- Percursos: pré, em, pós-ordem (iguais à ABB)
- Em-ordem sempre devolve a sequência ordenada

Complexidade

Op.	ABB médio	AVL
Busca	$O(\log n)$	$O(\log n)$
Pior caso	$O(n)$	$O(\log n)$
Rotação	–	$O(1)$

Fim da Aula 10

Dúvidas e próximos passos

Próxima Aula e Referências

Próxima aula — Árvores B e B+

- Nós com **múltiplas chaves** e múltiplos filhos
- Motivação: estruturas otimizadas para **disco**
- **Árvore B+**: todas as chaves nas folhas
- Aplicações em **bancos de dados** e sistemas de arquivo

Referências

- CORMEN et al. *Introduction to Algorithms*, 4ª ed. Cap. 13.
- TENENBAUM et al. *Estruturas de Dados usando C*. Cap. 5.

Resumo da Aula 10

- AVL garante $O(\log N)$ em todas as operações
- FB $= h_d - h_e \in \{-1, 0, +1\}$
- 4 tipos de rotação: simples e duplas
- Em-ordem = sequência crescente

Dúvidas?

carolsoko@ifba.edu.br

IFBA – Campus Feira de Santana