Árvore Binária de Busca

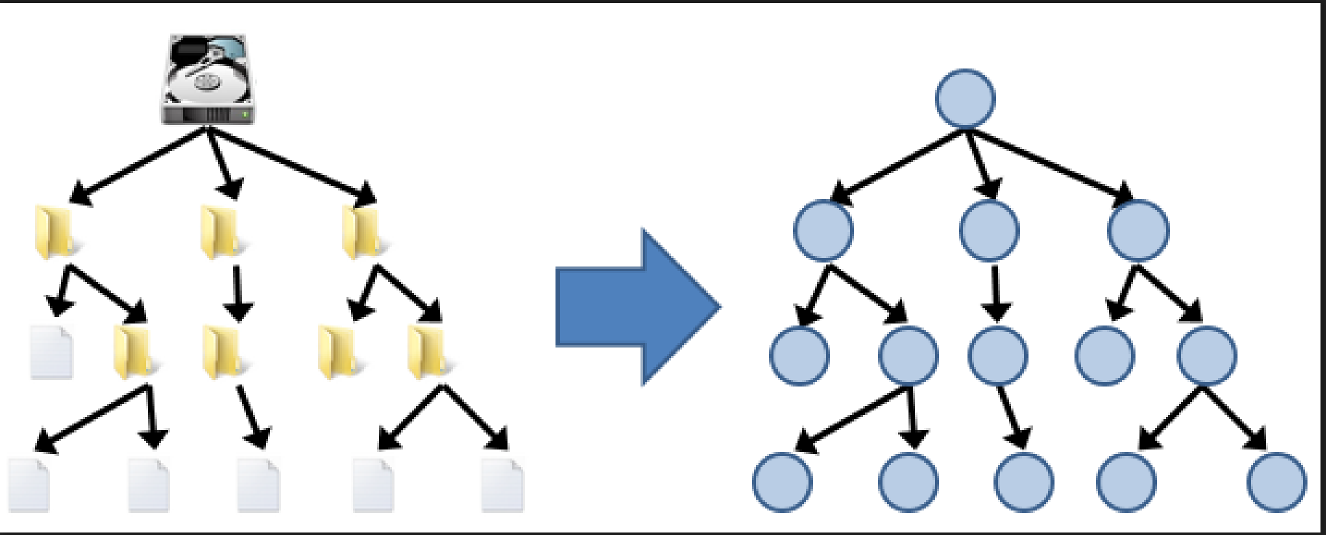
Prof. André Backes | @progdescomplicada

Definição

- Diversas aplicações necessitam que se represente um conjunto de objetos e as suas relações hierárquicas
- Uma árvore é uma abstração matemática usada para representar estruturas hierárquicas não lineares dos objetos modelados

Definição

- Exemplo
  - Estrutura de pastas do computador



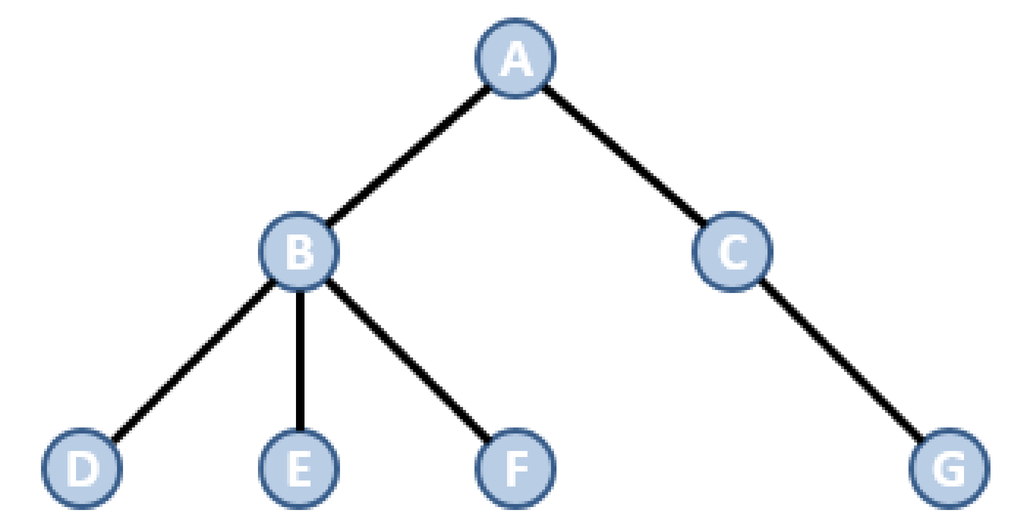
Definição

- Exemplos
  - relações de descendência (pai, filho, etc.)
  - diagrama hierárquico de uma organização;
  - campeonatos de modalidades desportivas;
  - taxonomia

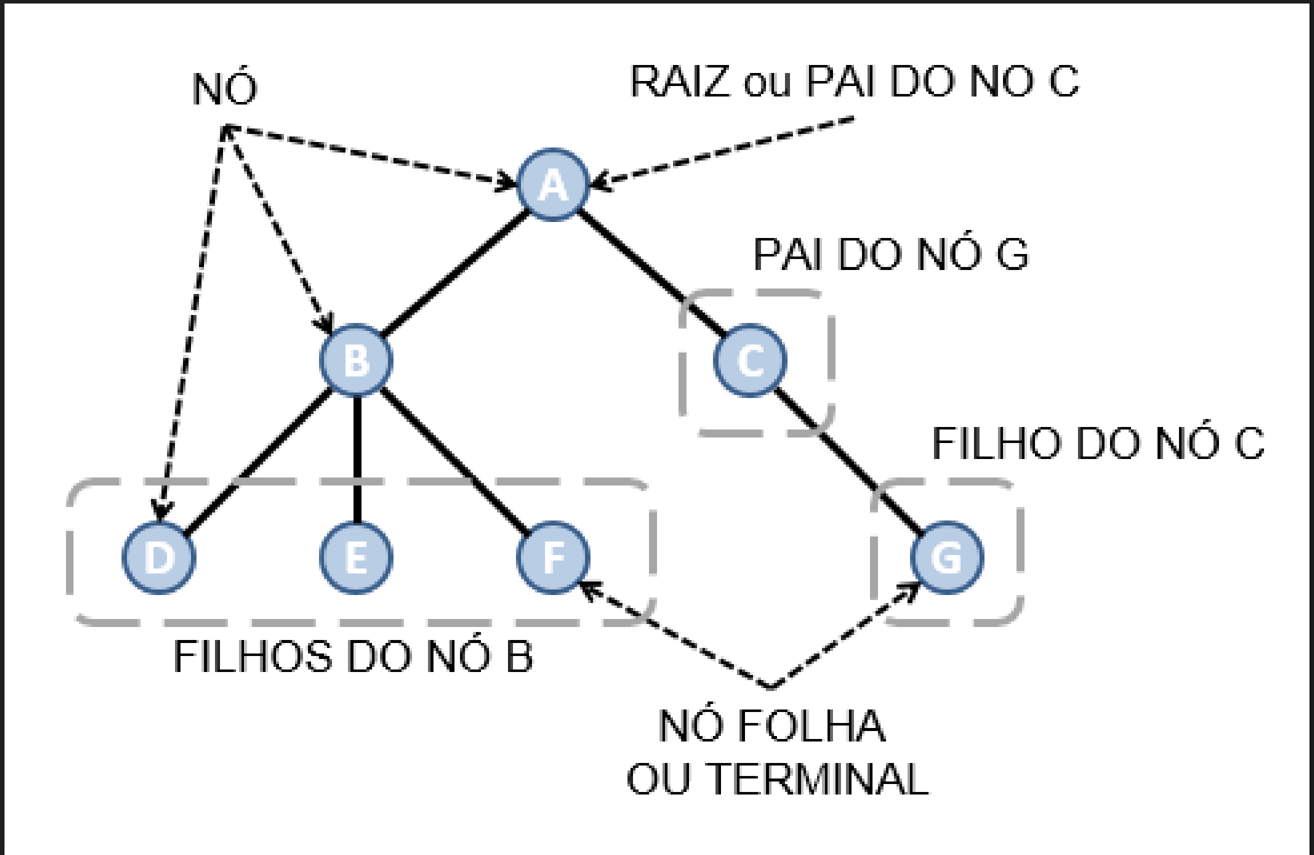
- Exemplos em computação
  - busca de dados armazenados no computador
  - representação de espaço de soluções
    - Exemplo: jogo de xadrez;
  - modelagem de algoritmos

Definição

- É um tipo especial de grafo
  - Definida usando um conjunto de nós (ou vértices) e arestas
  - Qualquer par de vértices está conectado a apenas uma aresta
    - Grafo não direcionado, conexo e acíclico (sem ciclos)



Definição


NÓ
RAIZ ou PAI DO NO C
A
PAI DO NÓ G
B
C
FILHO DO NÓ C
D
E
F
G
FILHOS DO NÓ B
NÓ FOLHA
OU TERMINAL

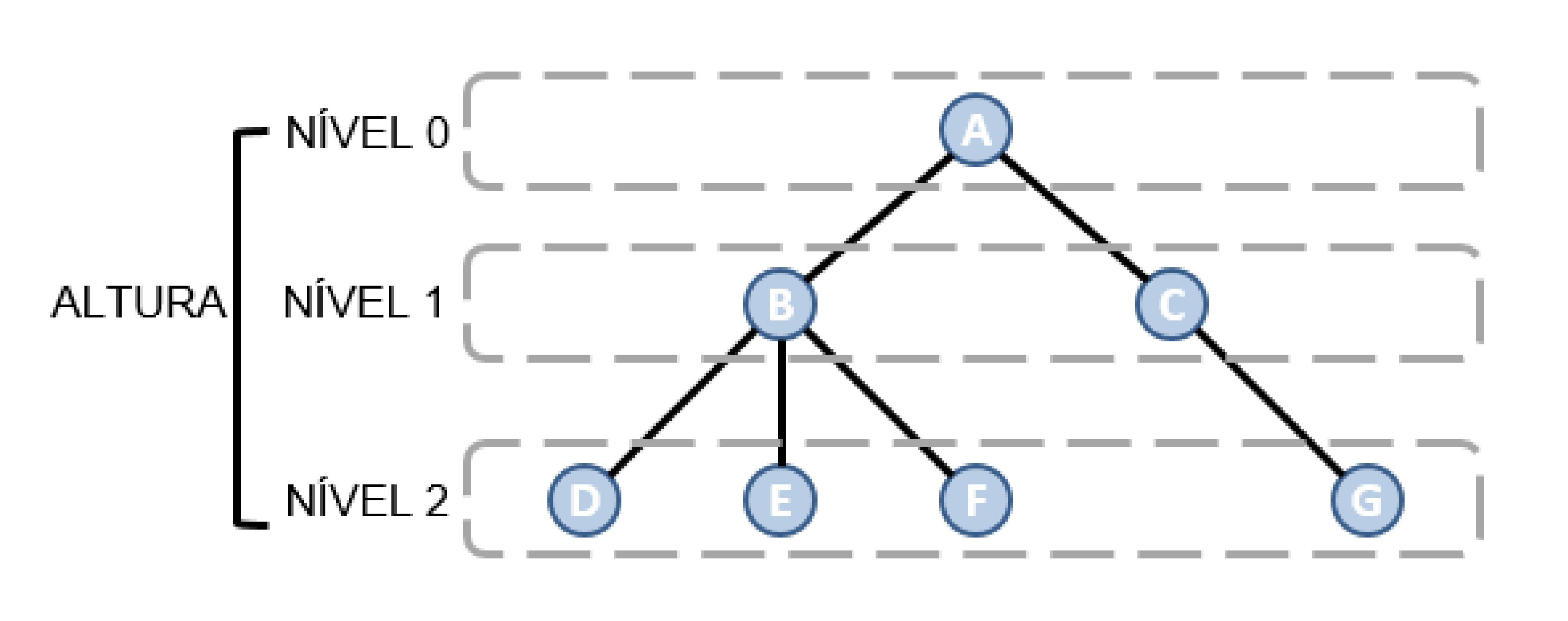
Definição

- Observação
  - Dado um determinado nó da árvore, cada filho seu é considerado a **raiz** de uma nova **sub-árvore**
    - Qualquer nó é a **raiz** de uma **sub-árvore** consistindo dele e dos nós abaixo dele
    - Conceito recursivo

Conceitos básicos

- Principais conceitos relativos as árvores
  - Nível
    - É dado pelo o número de nós que existem no caminho entre esse nó e a raiz (nível 0)
    - Nós são classificados em diferentes níveis
  - Altura
    - Também chamada de profundidade
    - Número total de níveis de uma árvore
    - Comprimento do caminho mais longo da raiz até uma das suas folhas

Conceitos básicos

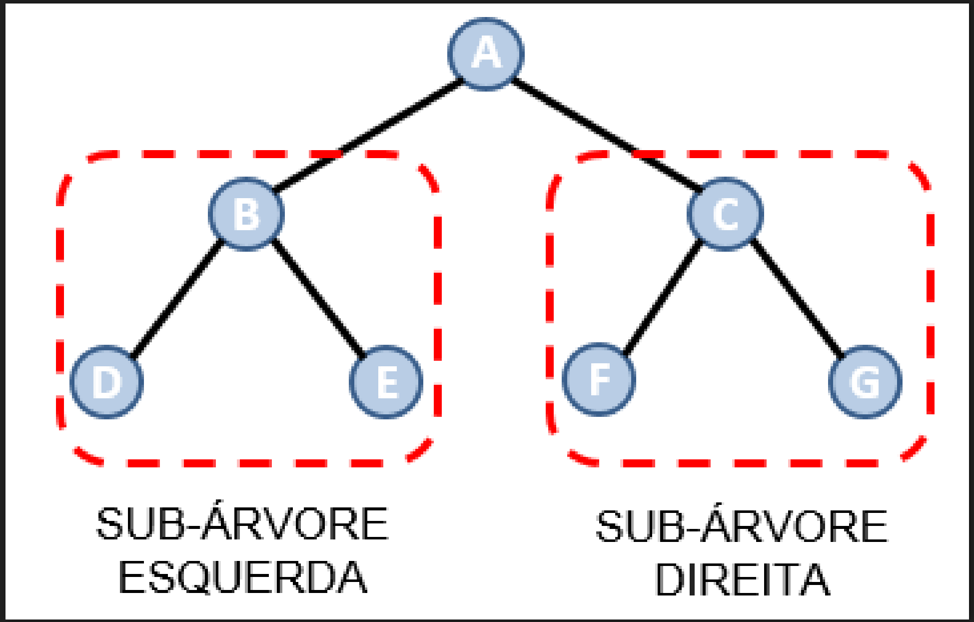


Tipos de árvores

- Na computação, assim como na natureza, existem vários tipos diferentes de árvores.
  - Cada uma delas foi desenvolvida pensando diferentes tipos de aplicações
    - árvore binária de busca
    - árvore AVL
    - árvore Rubro-Negra
    - árvore B, B+ e B*
    - árvore 2-3
    - árvore 2-3-4
    - quadtree
    - octree

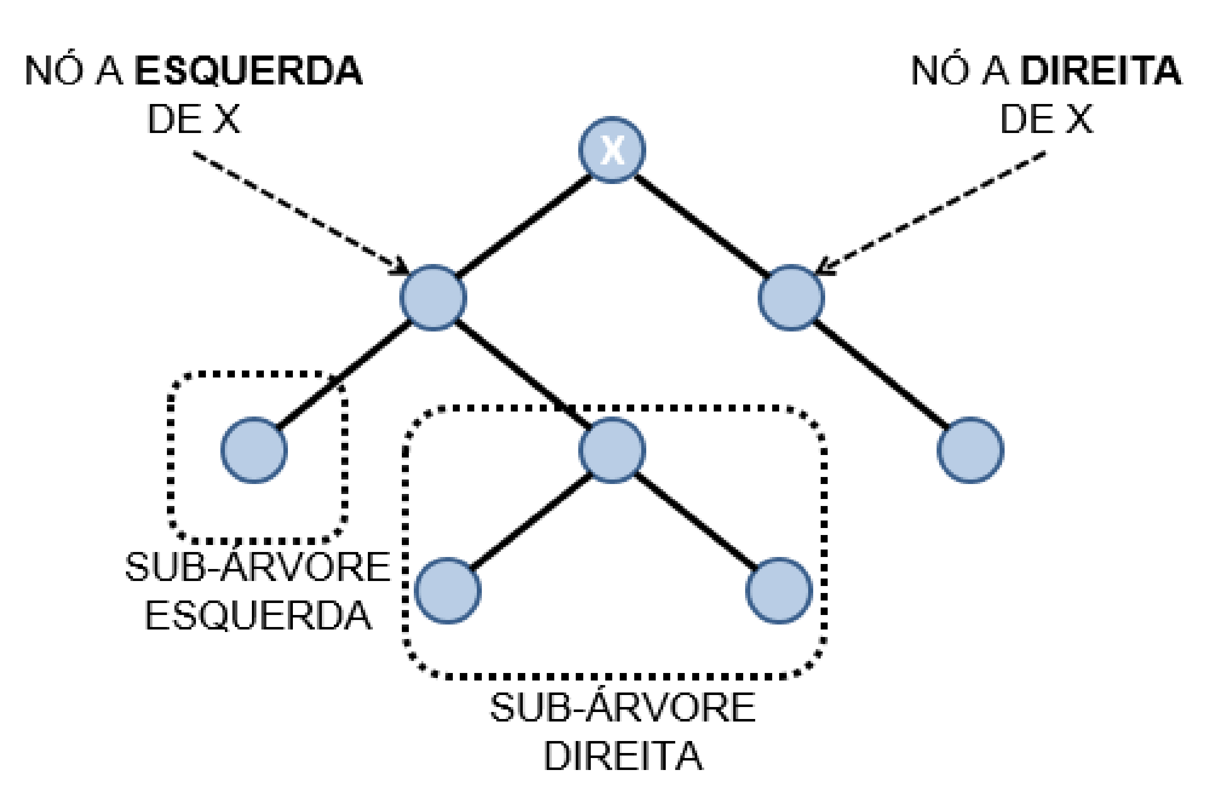
Árvore Binária

- É um tipo especial de árvore
  - Cada nó pode possuir nenhuma, uma ou no máximo duas **sub-árvores**
    - Sub-árvore da **esquerda** e a da **direita**
  - Usadas em situações onde, a cada passo, é preciso tomar uma decisão entre duas direções



Árvore Binária

- Exemplo de árvore binária

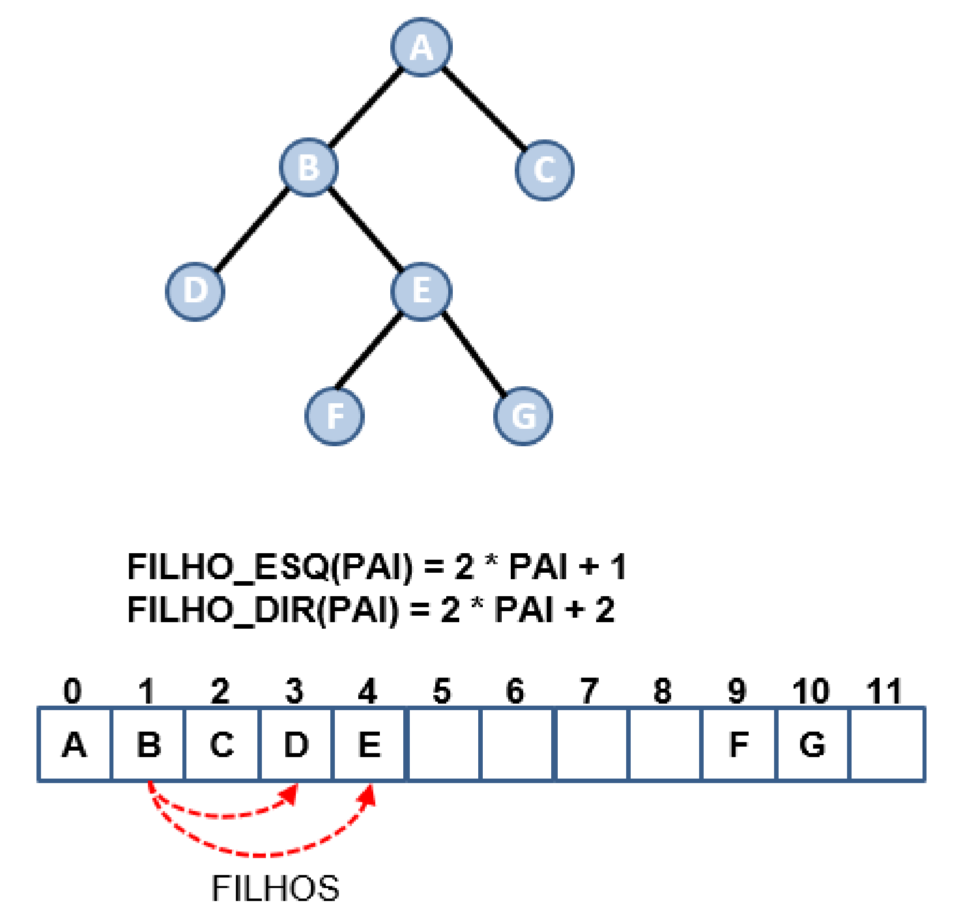


Tipo de representação

- Como implementar uma árvore no computador?
- Existem duas abordagens muito utilizadas
  - Usando um array (alocação estática)
  - Usando uma lista encadeada (alocação dinâmica)

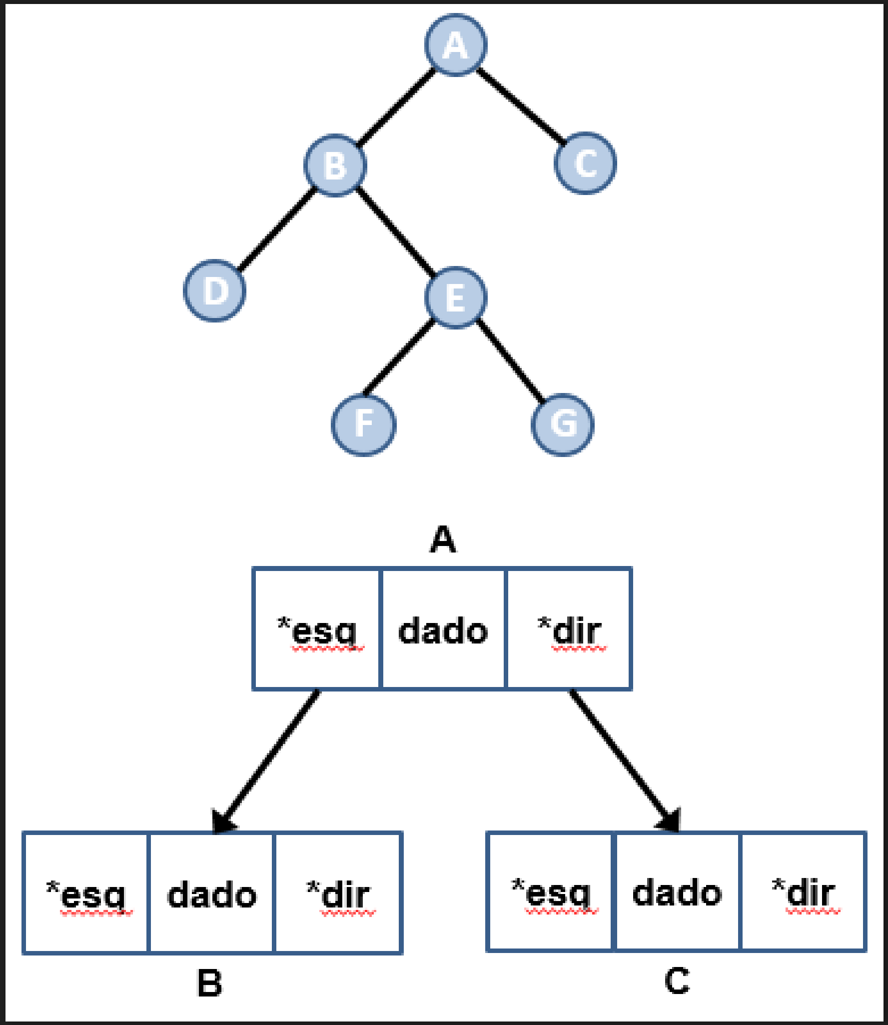
Tipo de representação

- Usando um array (alocação estática)
  - Necessário definir o número máximo de nós
    - Tamanho do array
  - Usa 2 funções para retornar a posição dos filhos à esquerda e à direita de um pai



Tipo de representação

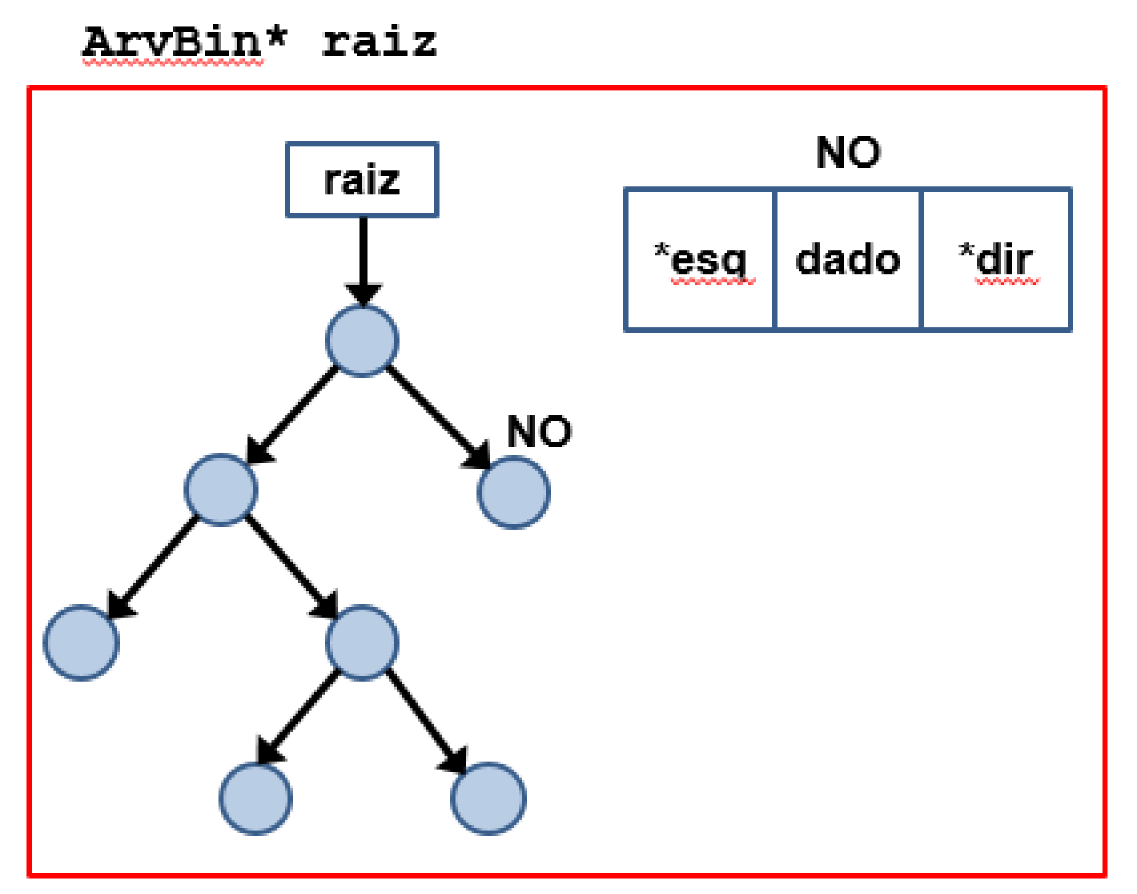
- Lista encadeada (alocação dinâmica)
  - Espaço de memória alocado em tempo de execução
    - A árvore cresce à medida que novos elementos são armazenados, e diminui à medida que elementos são removidos



TAD Árvore Binária

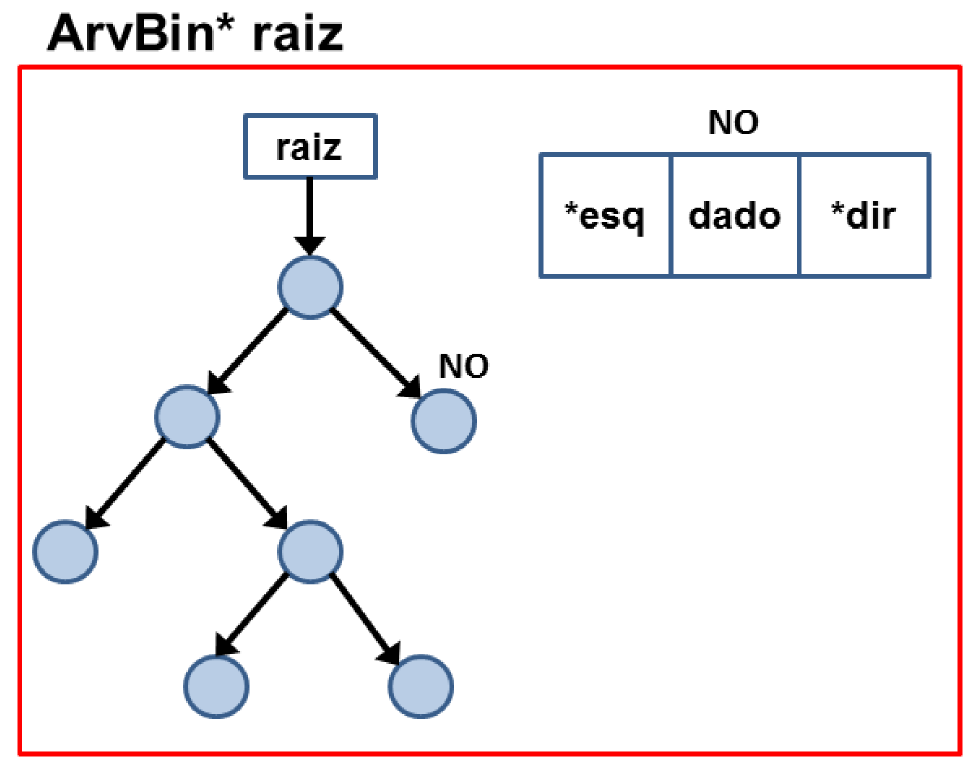
- Definição
  - Uso de alocação dinâmica
    - Para guardar o primeiro nó da árvore utilizamos um **ponteiro para ponteiro**
    - Um **ponteiro para ponteiro** pode guardar o endereço de um **ponteiro**
    - Assim, fica fácil mudar quem é a **raiz** da árvore (se necessário)

TAD Árvore Binária



TAD Árvore Binária

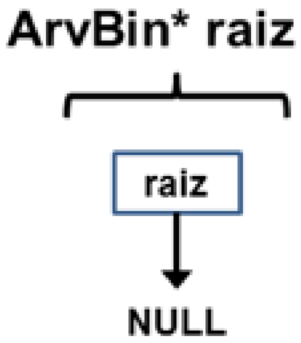
```c
//Arquivo ArvoreBinaria.h
typedef struct NO* ArvBin;

//Arquivo ArvoreBinaria.c
#include <stdio.h>
#include <stdlib.h>
#include "ArvoreBinaria.h" //inclui os Protótipos
struct NO{
    int info;
    struct NO *esq;
    struct NO *dir;
};
//programa principal
ArvBin* raiz; //ponteiro para ponteiro
```



TAD Árvore Binária

- Criando a árvore

```c
//arquivo ArvoreBinaria.h
ArvBin* cria_ArvBin();

//arquivo ArvoreBinaria.c
ArvBin* cria_ArvBin(){
    ArvBin* raiz = (ArvBin*) malloc(sizeof(ArvBin));
    if(raiz != NULL)
        *raiz = NULL;
    return raiz;
}

//programa principal
ArvBin* raiz = cria_ArvBin();
```



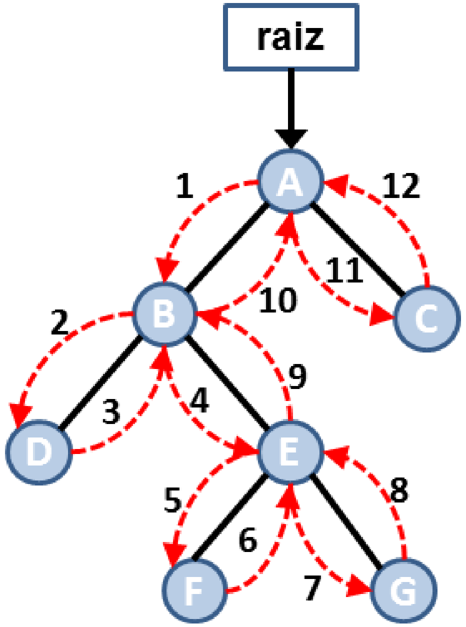
TAD Árvore Binária

- Liberando a árvore
  - Uso de 2 funções: uma percorre e libera os nós, outra trata a raiz

```
void libera_NO(struct NO* no){
    if(no == NULL)
        return;
    libera_NO(no->esq);
    libera_NO(no->dir);
    free(no);
    no = NULL;
}
void libera_ArvBin(ArvBin* raiz){
    if(raiz == NULL)
        return;
    libera_NO(*raiz);//libera cada nó
    free(raiz);//libera a raiz
}
```

TAD Árvore Binária

- Remoção: passo a passo



	1	visita B
	2	visita D
✖	3	libera D, volta para B
	4	visita E
	5	visita F
✖	6	libera F, volta para E
	7	visita G
✖	8	libera G, volta para E
✖	9	libera E, volta para B
✖	10	libera B, volta para A
	11	visita C
✖	12	libera C, volta para A e
✖		libera A
✖		libera raiz

Árvore Binária de Busca - ABB

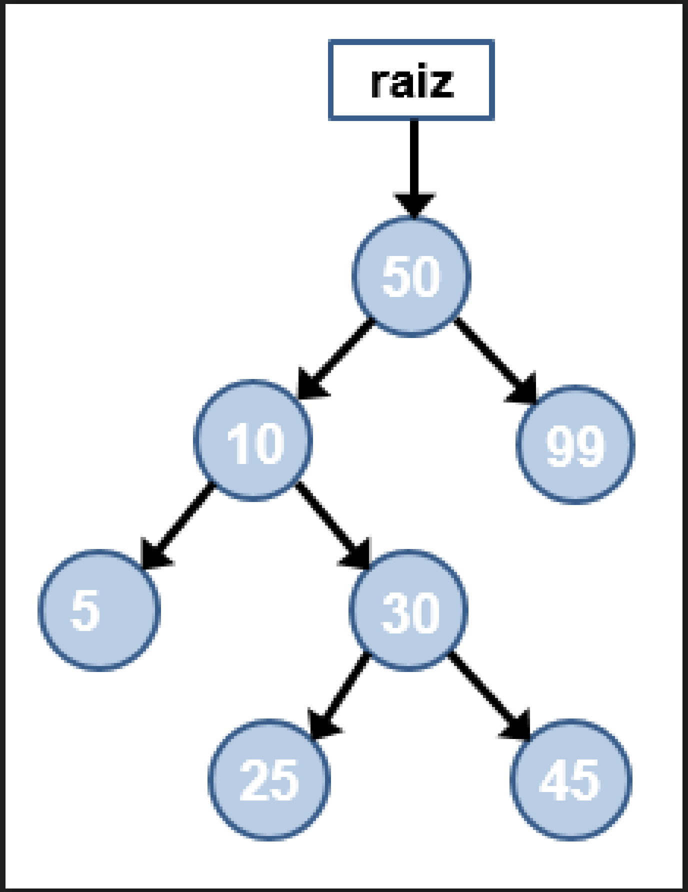
Árvore Binária de Busca

- Definição
  - É uma árvore binária
    - Cada nó pode ter 0, 1 ou 2 filhos
  - Cada nó possui da árvore possui um valor (chave) associado a ele
    - Não existem valores repetidos
  - Esse valor determina a posição do nó na árvore

Árvore Binária de Busca

- Regra para posicionamento dos valores na árvore
  - Para cada nó pai
    - todos os valores da sub-árvore **esquerda são menores** do que o nó pai
    - todos os valores da sub-árvore **direita são maiores** do que o nó pai;
  - Inserção e remoção devem ser realizadas respeitando essa regra de posicionamento dos nós.

Árvore Binária de Busca



Árvore Binária de Busca

- Ótima alternativa para operações de busca binária
  - Possui a vantagem de ser uma estrutura dinâmica em comparação ao array
  - É mais fácil inserir valores na árvore do que em um array ordenado
    - Array: envolve deslocamento de elementos

Árvore Binária de Busca

- Custo para as principais operações em uma **árvore binária de busca** contendo **N** nós.
  - O pior caso ocorre quando a árvore não está balanceada

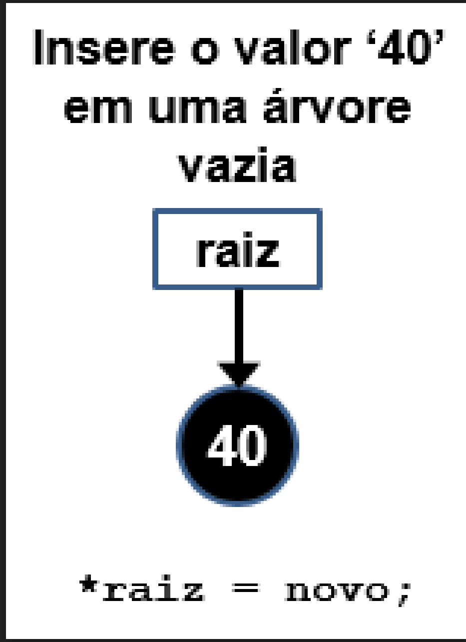
	Melhor Caso	Pior Caso
Inserção	*O(log N)*	*O(N)*
Remoção	*O(log N)*	*O(N)*
Busca	*O(log N)*	*O(N)*

Árvore Binária de Busca - Inserção

- Para inserir um valor **V** na árvore
  - Se a raiz é igual a **NULL**, insira o nó
  - Se **V** é menor do que a raiz: vá para a **sub-árvore esquerda**
  - Se **V** é maior do que a raiz: vá para a **sub-árvore direita**
  - Aplique o método **recursivamente**
    - pode ser feito sem recursão
- Dessa forma, percorremos um conjunto de nós da árvore até chegar ao nó folha que irá se tornar o pai do novo nó

Árvore Binária de Busca - Inserção

- Devemos também considerar a inserção em uma árvore que está vazia



Árvore Binária de Busca - Inserção

```c
int insere_ArvBin(ArvBin* raiz, int valor){
    if(raiz == NULL)
        return 0;
    struct NO* novo;
    novo = (struct NO*) malloc(sizeof(struct NO));
    if(novo == NULL)
        return 0;
    novo->infor = valor;
    novo->dir = NULL;
    novo->esq = NULL;
    //procurar onde inserir!
```

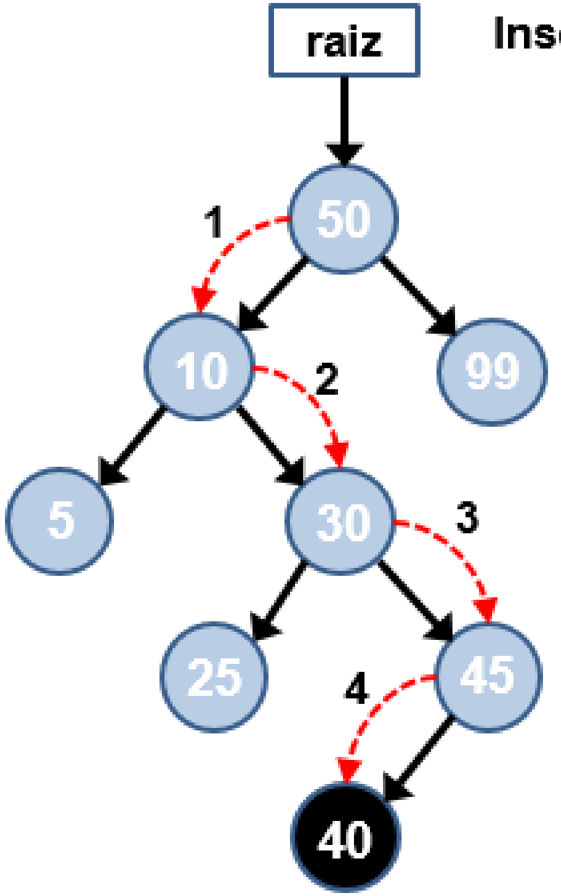
Árvore Binária de Busca - Inserção

Navega nos nós da árvore até chegar em um nó folha

Insere como filho desse nó folha

```
if(*raiz == NULL)
    *raiz = novo;
else{
    struct NO* atual = *raiz;
    struct NO* ant = NULL;
    while(atual != NULL){
        ant = atual;
        if(valor == atual->info){
            free(novo);
            return 0;//elemento já existe
        }
        if(valor > atual->info)
            atual = atual->dir;
        else
            atual = atual->esq;
    }
    if(valor > ant->info)
        ant->dir = novo;
    else
        ant->esq = novo;
}
return 1;
```

Árvore Binária de Busca - Inserção



Insere o valor '40' como um nó folha

1	valor é menor do que 50: visita filho da esquerda
2	valor é maior do que 10: visita filho da direita
3	valor é maior do que 30: visita filho da direita
4	valor é menor do que 45: visita filho da esquerda
Fim	Não existe filho da esquerda. Valor passa a ser o filho da esquerda de 45

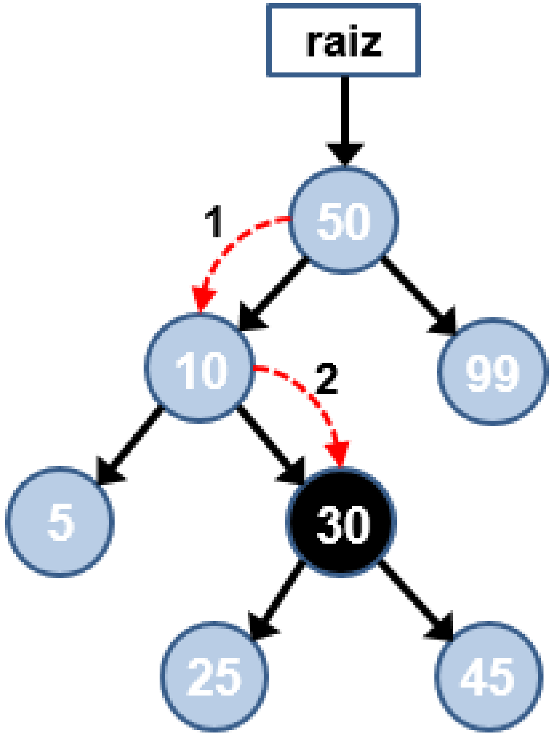
Árvore Binária de Busca: Busca

- Consultar se um determinado nó **V** existe em uma árvore é similar a operação de inserção
  - primeiro compare o valor buscado com a **raiz**;
  - se **V** é menor do que a raiz: vá para a **sub-árvore da esquerda**;
  - se **V** é maior do que a raiz: vá para a **sub-árvore da direita**;
  - aplique o método recursivamente até que a raiz seja igual ao valor buscado
    - pode ser feito sem recursão

Árvore Binária de Busca: Busca

```c
int consulta_ArvBin(ArvBin *raiz, int valor){
    if(raiz == NULL)
        return 0;
    struct NO* atual = *raiz;
    while(atual != NULL){
        if(valor == atual->info){
            return 1;
        }
        if(valor > atual->info)
            atual = atual->dir;
        else
            atual = atual->esq;
    }
    return 0;
}
```

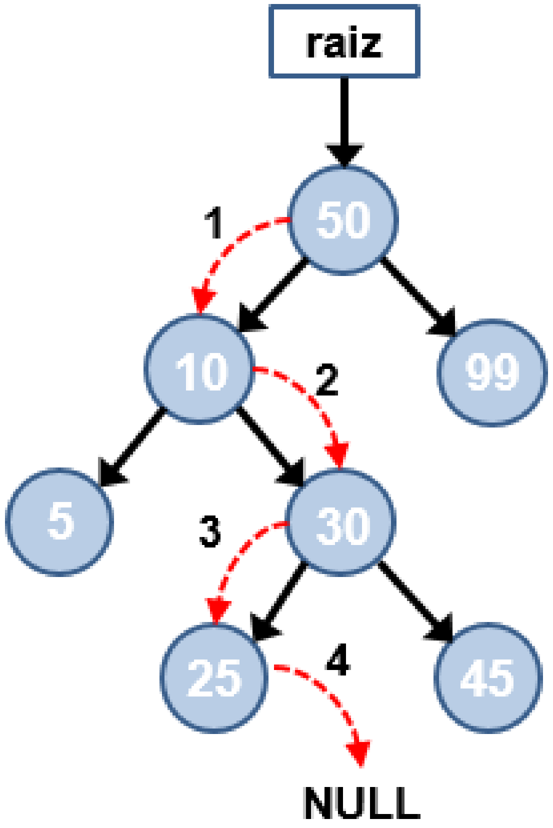
Árvore Binária de Busca: Busca



Valor procurado: 30

1	valor procurado é menor do que 50: visita filho da esquerda
2	valor procurado é maior do que 10: visita filho da direita
Fim	valor procurado é igual ao do nó: retornar dados do nó

Árvore Binária de Busca: Busca



Valor procurado: 28

1	valor procurado é menor do que 50: visita filho da esquerda
2	valor procurado é maior do que 10: visita filho da direita
3	valor procurado é menor do que 30: visita filho da esquerda
4	valor procurado é maior do que 25: visita filho da direita
Fim	Filho da direita de 25 não existe: a busca falhou

Árvore Binária de Busca: Remoção

- Remover um nó de uma árvore binária de busca não é uma tarefa tão simples quanto a inserção.
  - Isso ocorre porque precisamos procurar o nó a ser removido da árvore o qual pode ser um
    - nó folha
    - nó interno (que pode ser a raiz), com um ou dois filhos.
  - Se for um nó interno
    - Reorganizar a árvore para que ela continue sendo uma árvore binária de busca

Árvore Binária de Busca: Remoção

- Remoção em uma Árvore Binária de Busca
  - Trabalha com 2 funções
    - Busca pelo nó
    - Tratar os 3 tipos de remoção: com 0, 1 ou 2 filhos

```c
int remove_ArvBin(ArvBin *raiz, int valor){
    /*
    FUNÇÃO RESPONSÁVEL PELA BUSCA
    DO NÓ A SER REMOVIDO
    */
}
struct NO* remove_atual(struct NO* atual){
    /*
    FUNÇÃO RESPONSÁVEL POR TRATAR OS 3
    TIPOS DE REMOÇÃO
    */
}
```

Árvore Binária de Busca: Remoção

Achou o nó a ser removido. Tratar o tipo de remoção

Continua andando na árvore a procura do nó a ser removido

```c
int remove_ArvBin(ArvBin *raiz, int valor){
    if(raiz == NULL) return 0;
    struct NO* ant = NULL;
    struct NO* atual = *raiz;
    while(atual != NULL){
        if(valor == atual->info){
            if(atual == *raiz)
                *raiz = remove_atual(atual);
            else{
                if(ant->dir == atual)
                    ant->dir = remove_atual(atual);
                else
                    ant->esq = remove_atual(atual);
            }
            return 1;
        }
        ant = atual;
        if(valor > atual->info)
            atual = atual->dir;
        else
            atual = atual->esq;
    }
    return 0;
```

Árvore Binária de Busca: Remoção

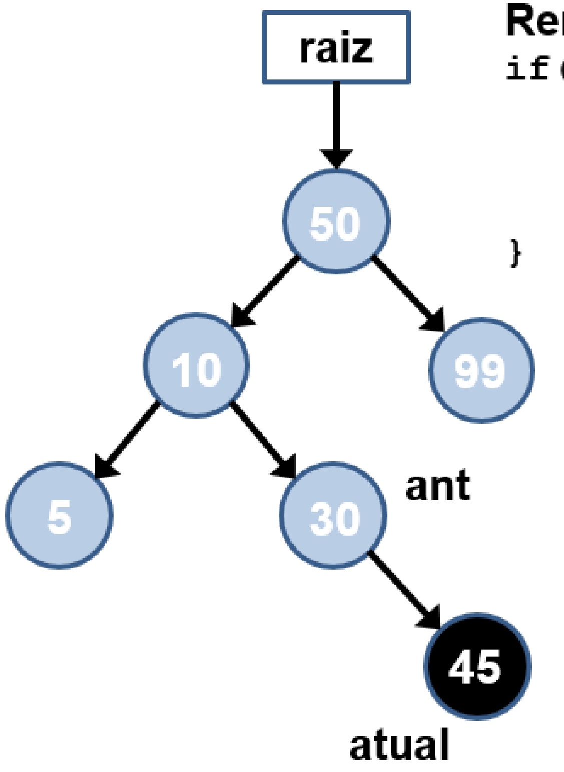
```c
struct NO* remove_atual(struct NO* atual) {
    struct NO *no1, *no2;
    if(atual->esq == NULL){
        no2 = atual->dir;
        free(atual);
        return no2;
    }
    no1 = atual;
    no2 = atual->esq;
    while(no2->dir != NULL){
        no1 = no2;
        no2 = no2->dir;
    }

    if(no1 != atual){
        no1->dir = no2->esq;
        no2->esq = atual->esq;
    }
    no2->dir = atual->dir;
    free(atual);
    return no2;
}
```

Sem filho da esquerda. Apontar para o filho da direita (trata nó folha e nó com 1 filho)

Procura filho mais a direita na sub-árvore da esquerda.

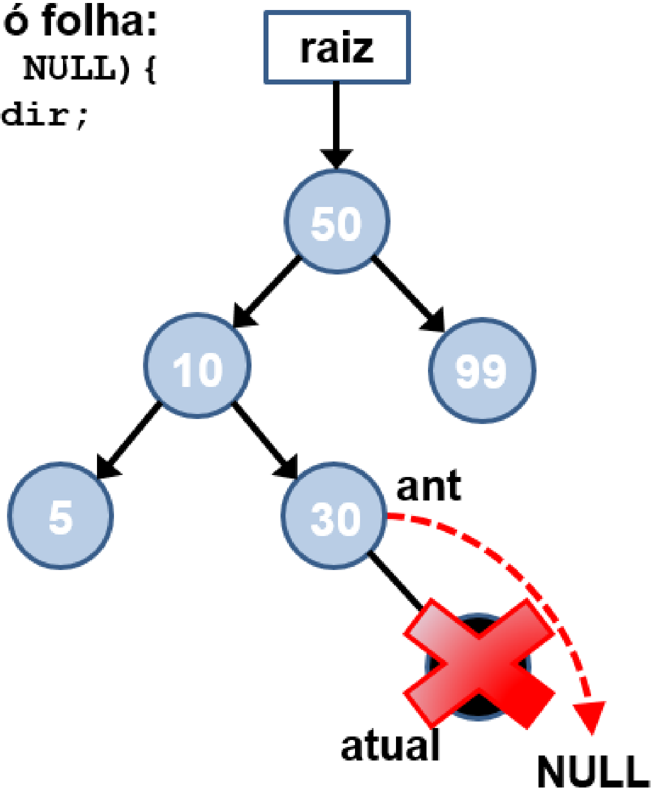
Copia o filho mais a direita na sub-árvore da esquerda para o lugar do nó removido.

Árvore Binária de Busca: Remoção

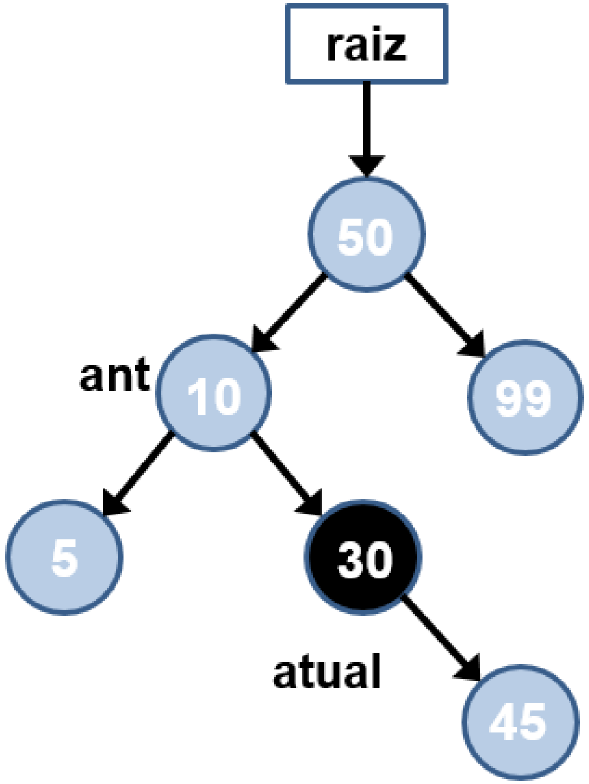


Remoção de um nó folha:

```
if(atual->esq == NULL){
    no2 = atual->dir;
    free(atual);
    return no2;
}
```

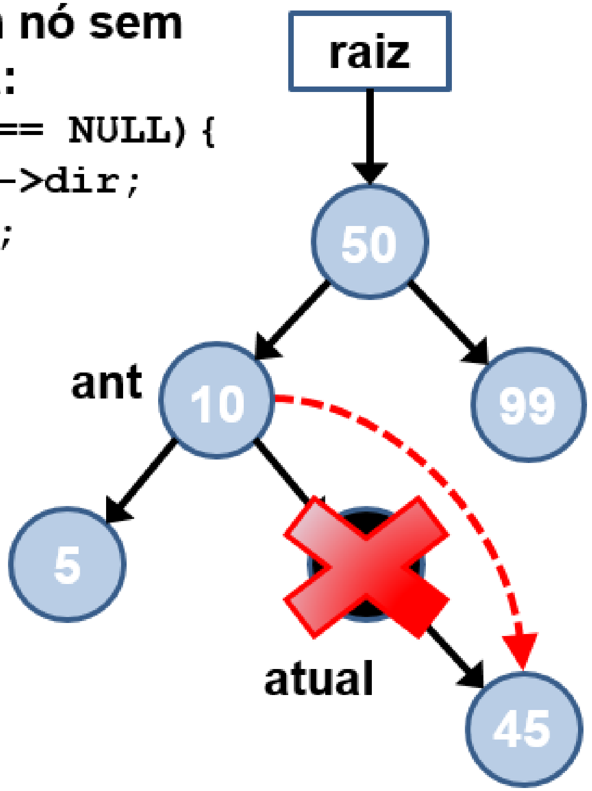


Árvore Binária de Busca: Remoção

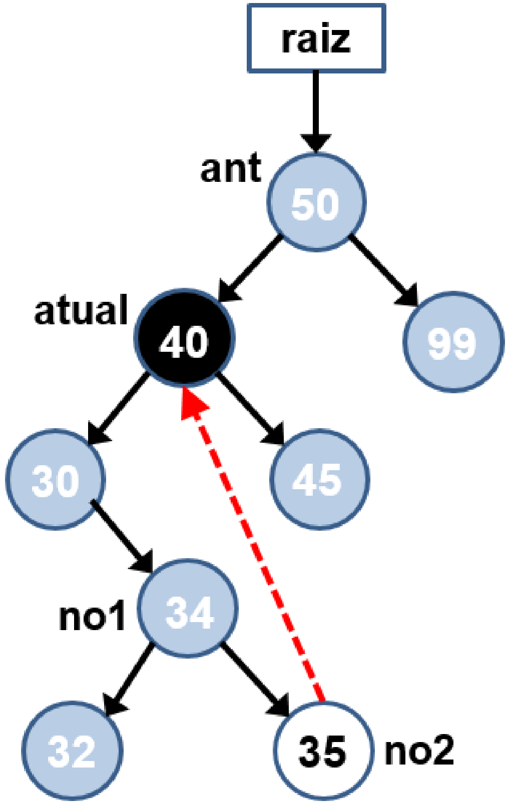


Remoção de um nó sem filho a esquerda:

```c
if(atual->esq == NULL){
    no2 = atual->dir;
    free(atual);
    return no2;
}
```

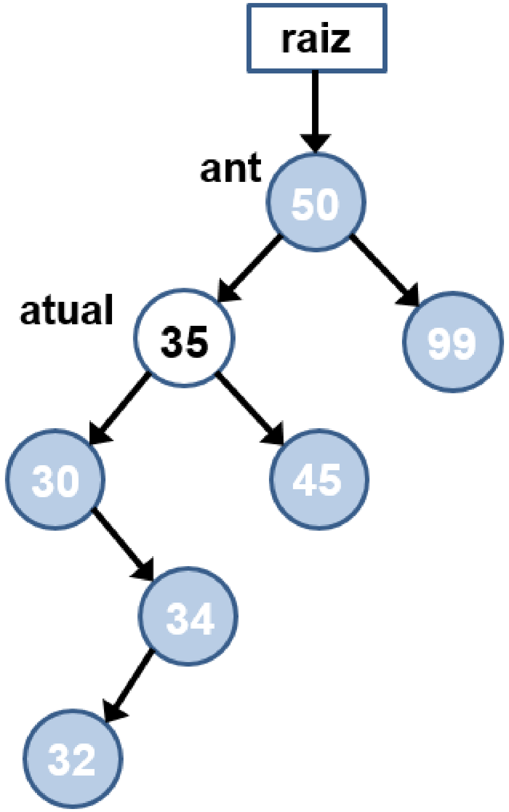


Árvore Binária de Busca: Remoção



Remoção de um nó com ambos os filhos:

Remover o nó "atual" e substituí-lo pelo nó "mais a direita" da sub-árvore da esquerda do nó "atual"



Árvore Binária de Busca Balanceada

Problema do balanceamento

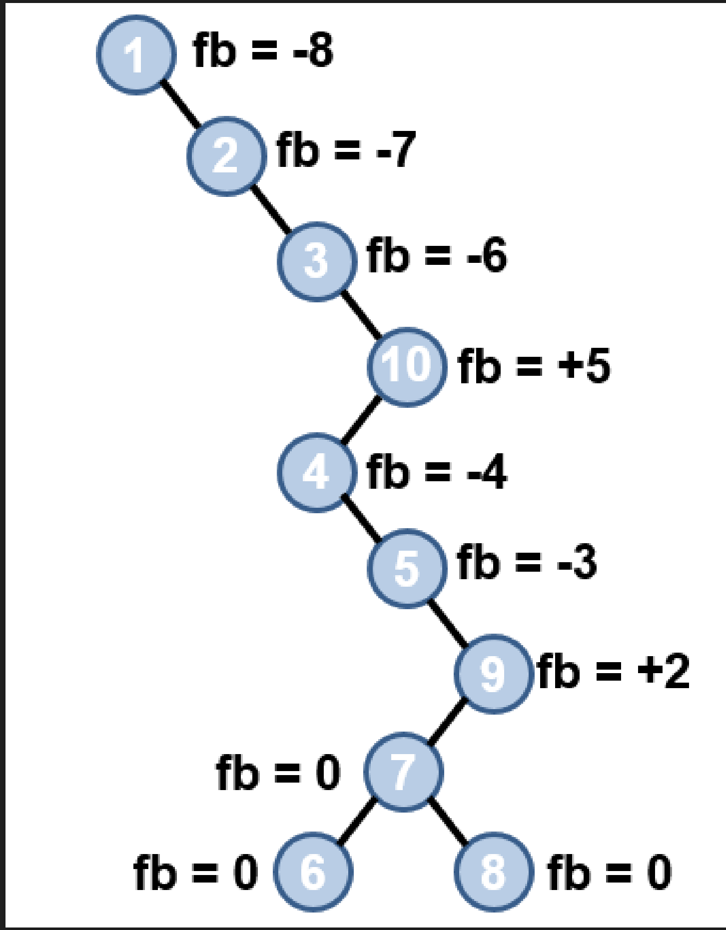
- A eficiência da busca em uma árvore binária depende do seu balanceamento.
  - ***O(log N)***, se a árvore está balanceada
  - ***O(N)***, se a árvore não está balanceada
    - **N** corresponde ao número de nós na árvore

Problema do balanceamento

- Infelizmente, os algoritmos de inserção e remoção em árvores binárias não garantem que a árvore gerada a cada passo esteja balanceada.
- Dependendo da ordem em que os dados são inseridos na árvore, podemos criar uma árvore na forma de uma escada

Problema do balanceamento

- Inserção dos valores
- {1,2,3,10,4,5,9,7,8,6}



Problema do balanceamento

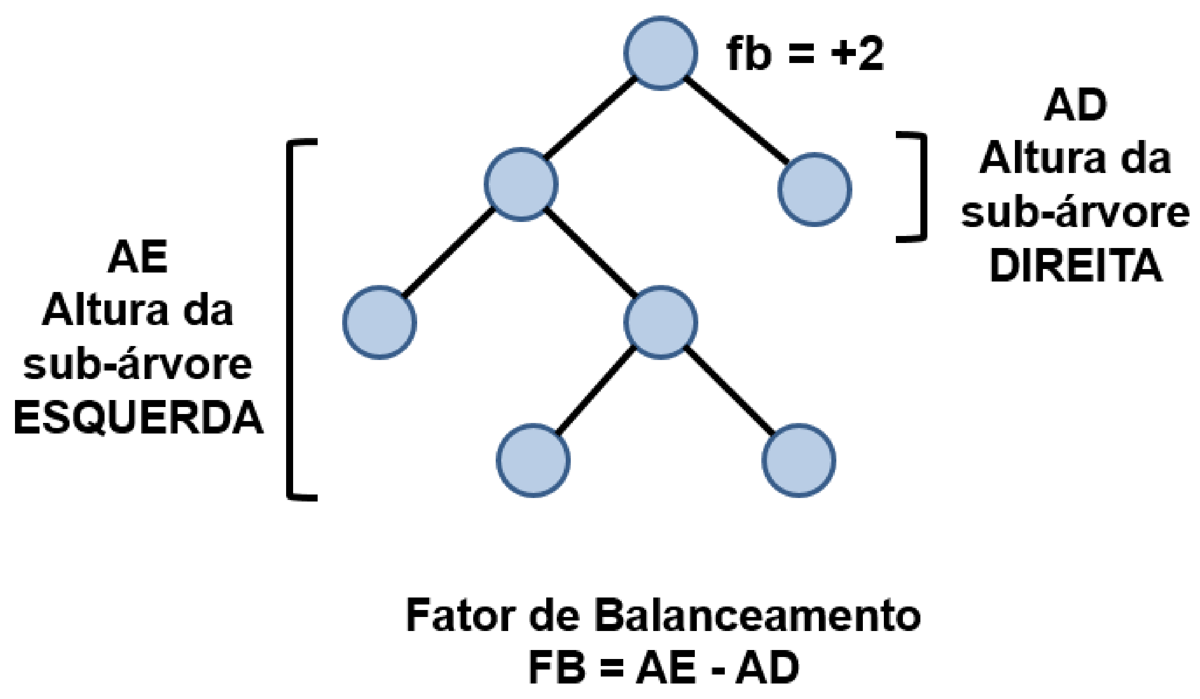
- Solução para o problema de balanceamento
  - Modificar as operações de inserção e remoção de modo a balancear a árvore a cada nova inserção ou remoção.
    - Garantir que a diferença de alturas das sub-árvores esquerda e direita de cada nó seja de no máximo uma unidade
  - Exemplos de árvores balanceadas
    - Árvore AVL
    - Árvore 2-3-4
    - Árvore Rubro-Negra

Árvore AVL

- Definição
  - Criada por **A**delson-**V**elskii e **L**andis, de onde recebeu a sua nomenclatura, em 1962
  - Permite o rebalanceamento local da árvore
    - Apenas a parte afetada pela inserção ou remoção é rebalanceada
  - Usa **rotações simples** ou **duplas** na etapa de rebalanceamento
    - Executadas a cada inserção ou remoção
    - Custo máximo de qualquer algoritmo é ***O(log N)***

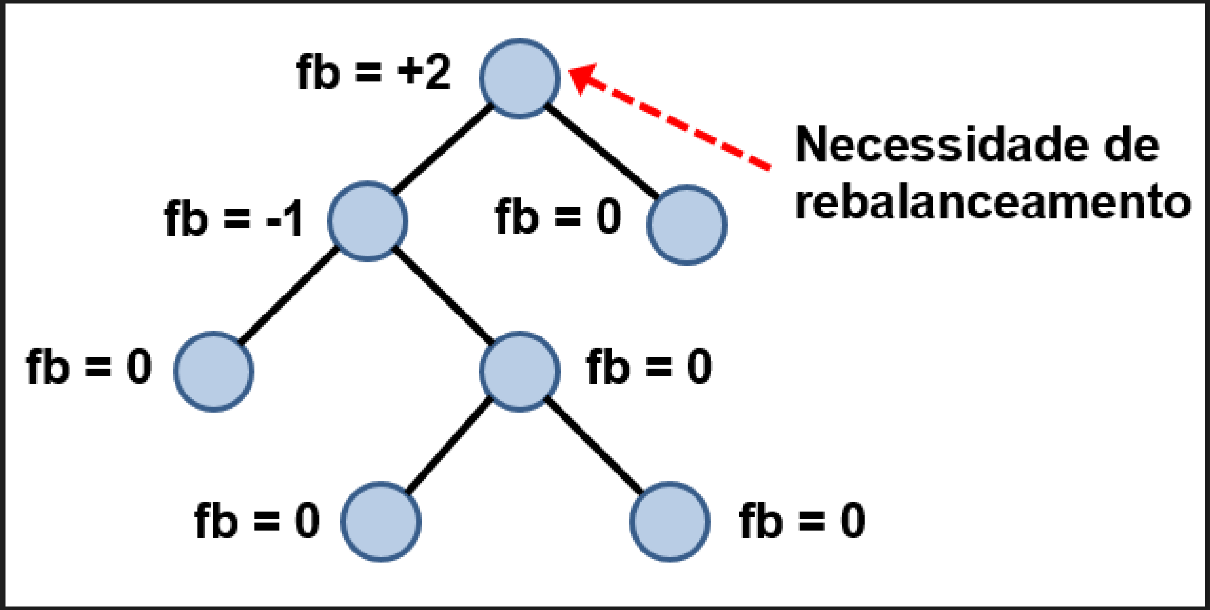
Árvore AVL

- Objetivo das rotações:
  - Corrigir o **fator de balanceamento** (ou **fb**)
    - Diferença entre as alturas das sub-árvore de um nó
  - Caso uma das sub-árvores de um nó não existir, então a altura dessa sub-arvore será igual a -1.



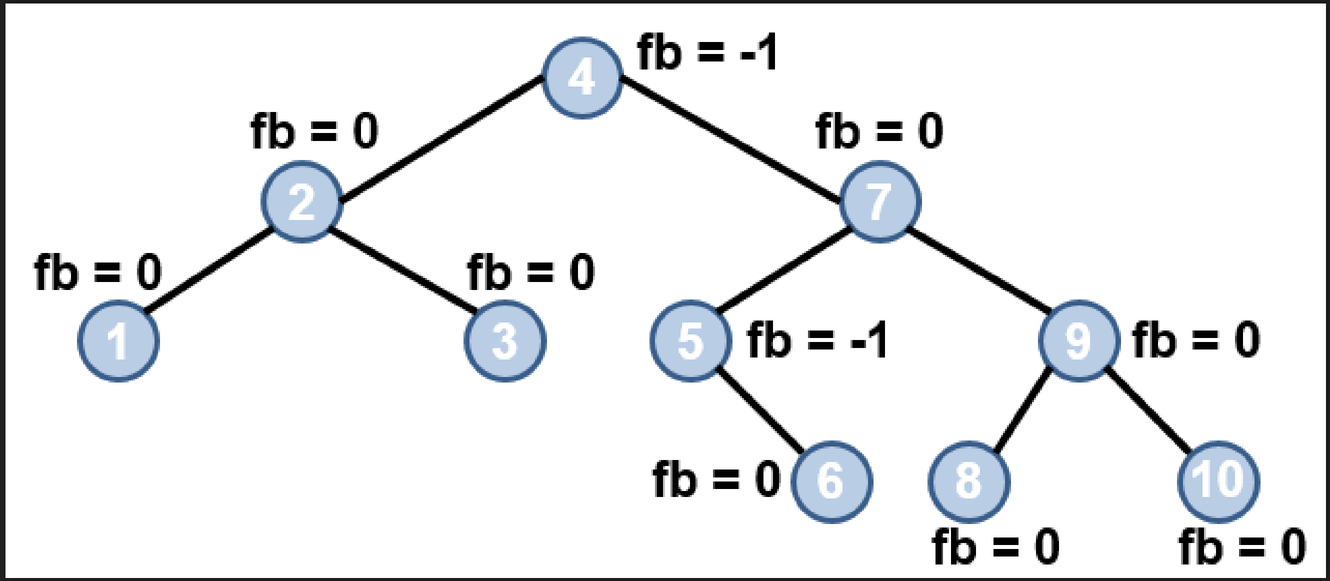
Árvore AVL

- As alturas das sub-árvores de cada nó diferem de no máximo uma unidade
  - O fator de balanceamento deve ser +1, 0 ou -1
  - Se **fb > +1** ou **fb < -1**: a árvore deve ser balanceada naquele nó



Árvore AVL

- Voltando ao problema anterior
- Inserção dos valores
- {1,2,3,10,4,5,9,7,8,6}



TAD Árvore AVL

- Definindo a árvore
  - Criação e destruição: igual a da árvore binária

```c
//Arquivo ArvoreAVL.h
typedef struct NO* ArvAVL;

//Arquivo ArvoreAVL.c
#include <stdio.h>
#include <stdlib.h>
#include "ArvoreAVL.h" //inclui os Protótipos
struct NO{
    int info;
    int alt;//altura daquela sub-árvore
    struct NO *esq;
    struct NO *dir;
};
//programa principal
ArvAVL* raiz; //ponteiro para ponteiro
```

TAD Árvore AVL

- Calculando o fator de balanceamento

```c
//Funções auxiliares
//Calcula a altura de um nó
int alt_NO(struct NO* no){
    if(no == NULL)
        return -1;
    else
    return no->alt;
}
//Calcula o fator de balanceamento de um nó
int fatorBalanceamento_NO(struct NO* no){
    return labs(alt_NO(no->esq) - alt_NO(no->dir));
}
```

Rotações

- Objetivo: corrigir o **fator de balanceamento** (ou **fb**) de cada nó
  - Operação básica para balancear uma árvore AVL
- Ao todo, existem dois tipos de rotação
  - Rotação simples
  - Rotação dupla

Rotações

- As rotações diferem entre si pelo sentido da inclinação entre o nó pai e filho
  - Rotação simples
    - O nó desbalanceado (pai), seu filho e o seu neto estão todos no mesmo sentido de inclinação
  - Rotação dupla
    - O nó desbalanceado (pai) e seu filho estão inclinados no sentido inverso ao neto
    - **Equivale a duas rotações simples.**

Rotações

- Ao todo, existem duas rotações simples e duas duplas:
  - Rotação **simples a direita** ou **Rotação LL**
  - Rotação **simples a esquerda** ou **Rotação RR**
  - Rotação **dupla a direita** ou **Rotação LR**
  - Rotação **dupla a esquerda** ou **Rotação RL**

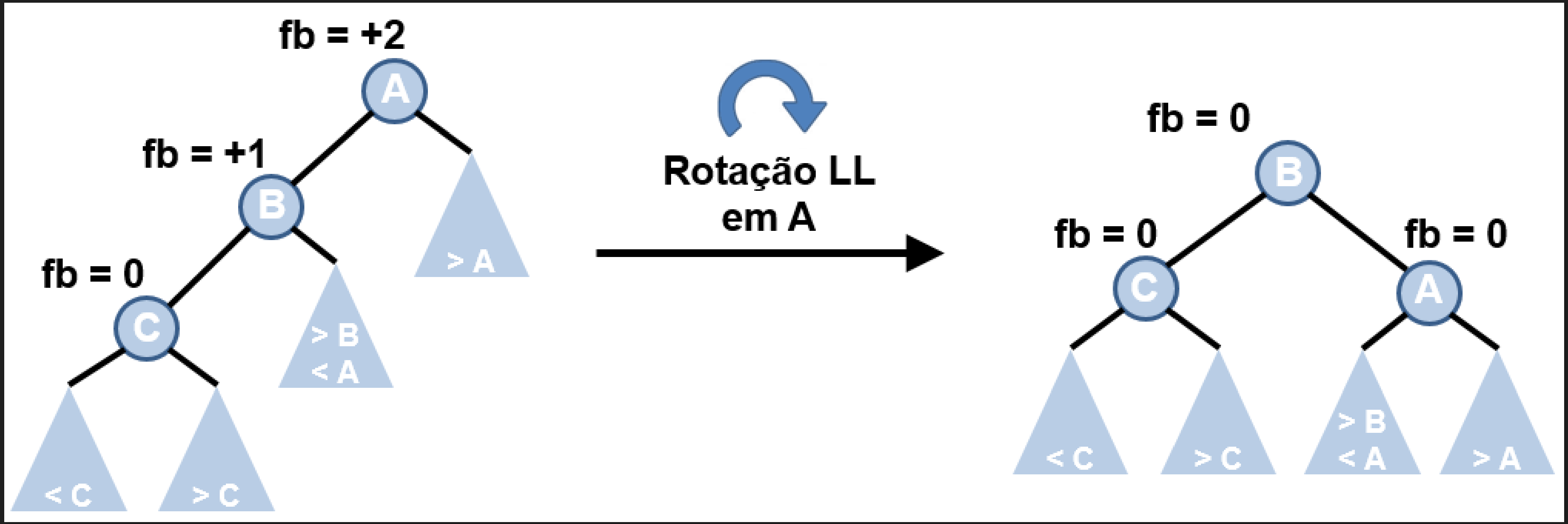
Rotações

- Rotações são aplicadas no ancestral mais próximo do nó inserido cujo fator de balanceamento passa a ser +2 ou -2
  - Após uma inserção ou remoção, devemos voltar pelo mesmo caminho da árvore e recalcular o fator de balanceamento, **fb**, de cada nó
  - Se o **fb** desse nó for +2 ou -2, uma rotação deverá ser aplicada

Rotação LL

- Rotação LL ou rotação simples à direita
  - Um novo nó é inserido na **sub-árvore da esquerda do filho esquerdo** de **A**
    - **A** é o nó desbalanceado
    - Dois movimentos para a esquerda: **LEFT LEFT**
  - É necessário fazer uma rotação à direita, de modo que o nó intermediário **B** ocupe o lugar de **A**, e **A** se torne a sub-árvore direita de **B**

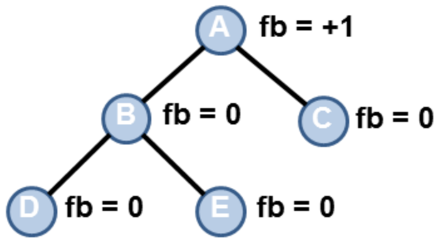
Rotação LL



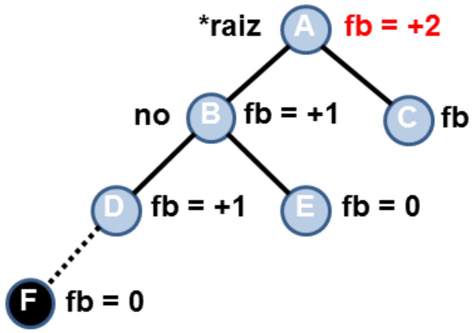
Rotação LL

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

Rotação LL



Árvore AVL e fator de balanceamento de cada nó

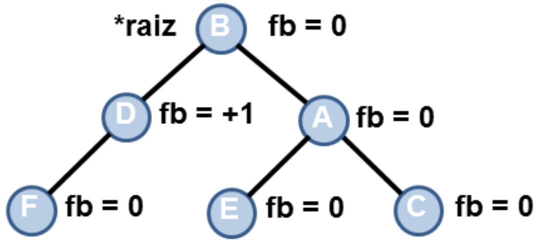


Inserção do nó F na árvore

Árvore fica desbalanceada no nó A.

Aplicar Rotação LL no nó A

```
no = (*raiz)->esq;
(*raiz)->esq = no->dir;
no->dir = *raiz;
*raiz = no;
```

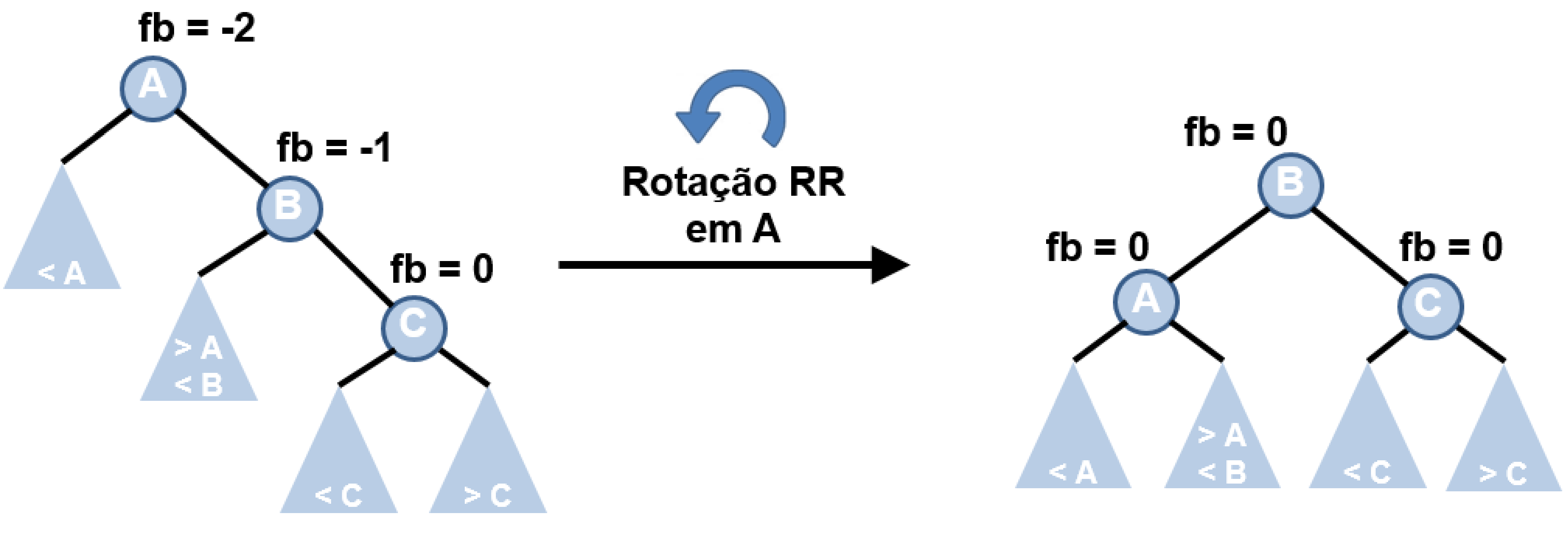


Árvore Balanceada

Rotação RR

- Rotação RR ou rotação simples à esquerda
  - Um novo nó é inserido na **sub-árvore da direita do filho direito** de **A**
    - **A** é o nó desbalanceado
    - Dois movimentos para a direita: **RIGHT RIGHT**
  - É necessário fazer uma rotação à esquerda, de modo que o nó intermediário **B** ocupe o lugar de **A**, e **A** se torne a sub-árvore esquerda de **B**

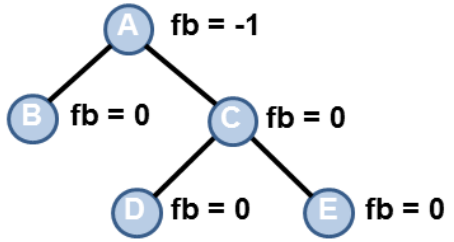
Rotação RR



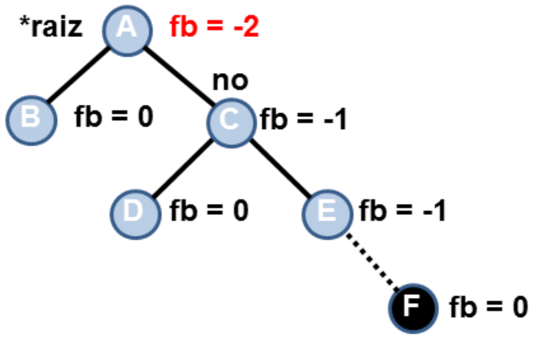
Rotação RR

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

Rotação RR



Árvore AVL e fator de balanceamento de cada nó

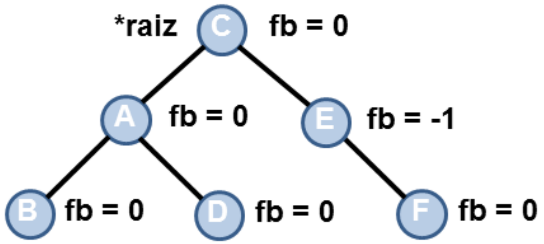


Inserção do nó F na árvore

Árvore fica desbalanceada no nó A.

Aplicar Rotação RR no nó A

```
no = (*raiz)->dir;
(*raiz)->dir = no->esq;
no->esq = (*raiz);
(*raiz) = no;
```



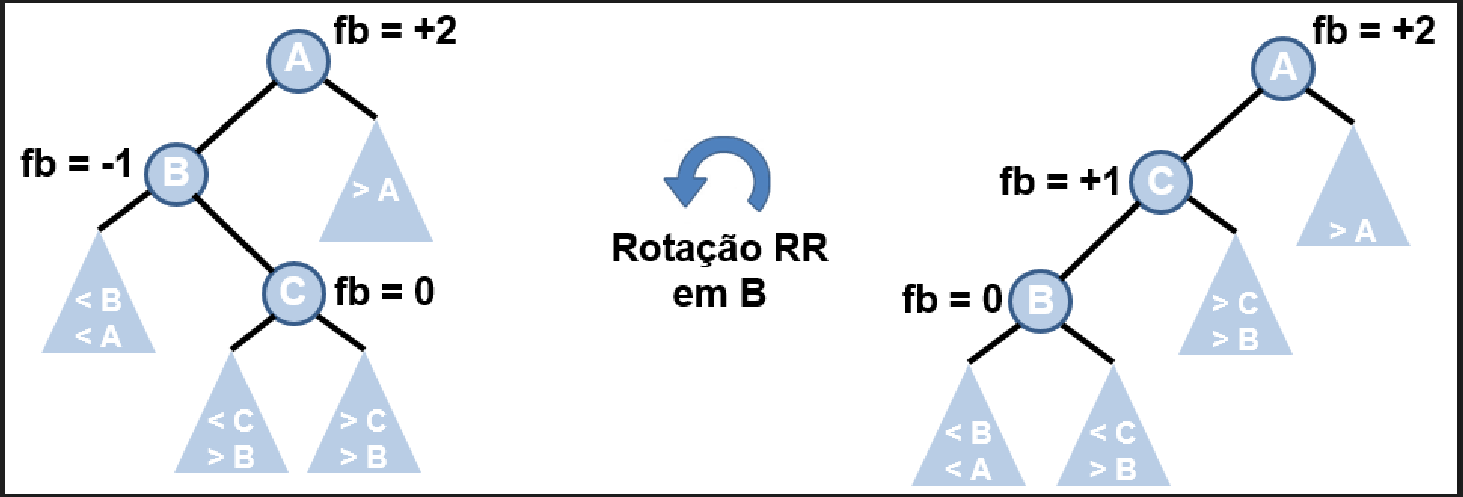
Árvore Balanceada

Rotação LR

- Rotação LR ou rotação dupla à direita
  - Um novo nó é inserido na **sub-árvore da direita do filho esquerdo** de **A**
    - **A** é o nó desbalanceado
    - Um movimento para a esquerda e outro para a direita: **LEFT RIGHT**
  - É necessário fazer uma rotação dupla, de modo que o nó **C** se torne o pai dos nós **A** (filho da direita) e **B** (filho da esquerda)
    - Rotação RR em **B**
    - Rotação LL em **A**

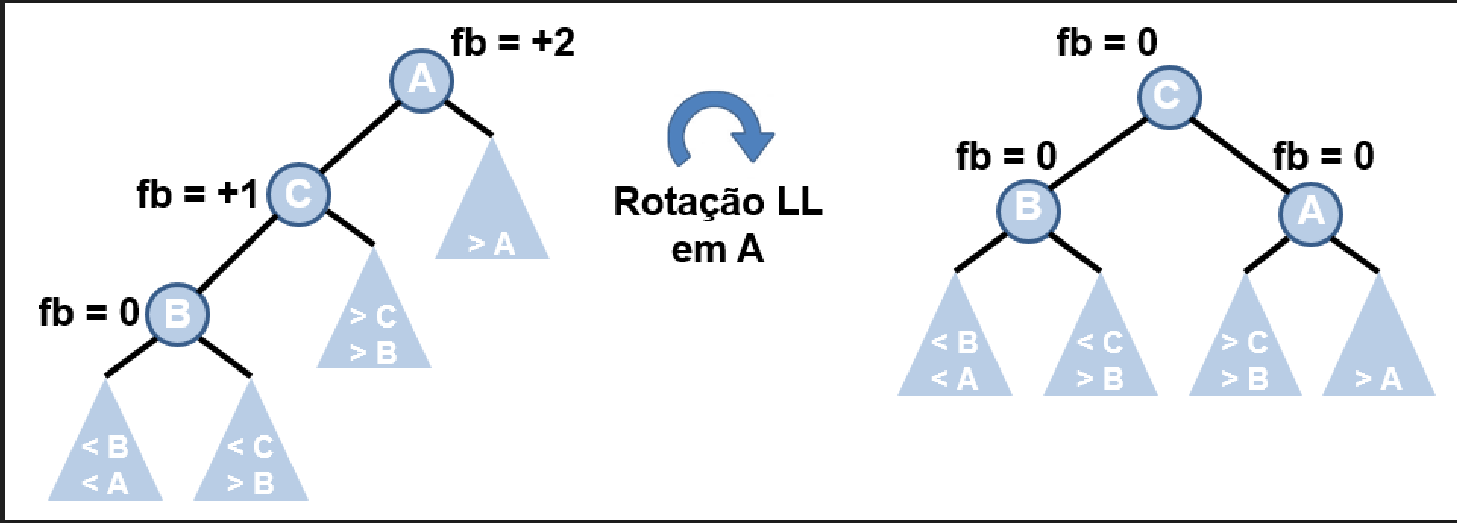
Rotação LR

```c
void RotacaoLR(ArvAVL *raiz){
    RotacaoRR(&(*raiz)->esq);
    RotacaoLL(raiz);
}
```

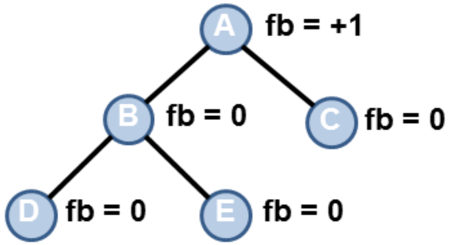


Rotação LR

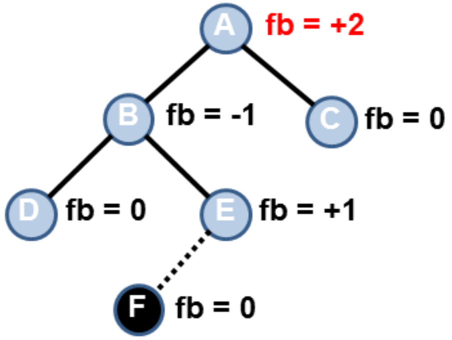
```
void RotacaoLR(ArvAVL *raiz){
    RotacaoRR(&(*raiz)->esq);
    RotacaoLL(raiz);
}
```



Rotação LR



Árvore AVL e fator de balanceamento de cada nó

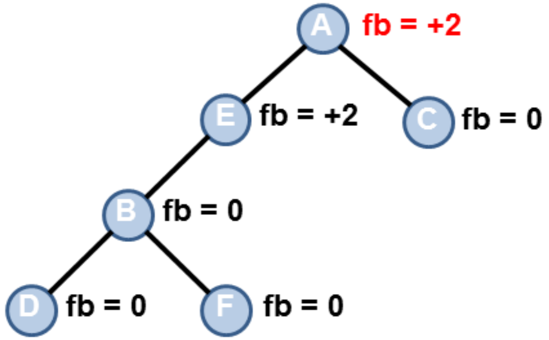


Inserção do nó F na árvore

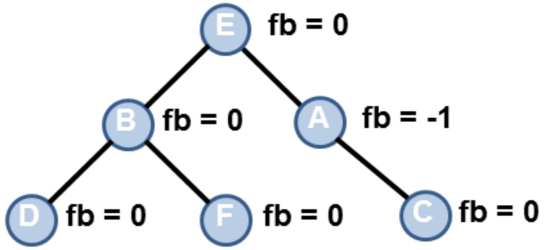
Árvore fica desbalanceada no nó A.

Aplicar Rotação LR no nó A.
Isso equivale a:

- Aplicar a Rotação RR no nó B
- Aplicar a Rotação LL no nó A



Árvore após aplicar a Rotação RR no nó B



Árvore após aplicar a Rotação LL no nó A

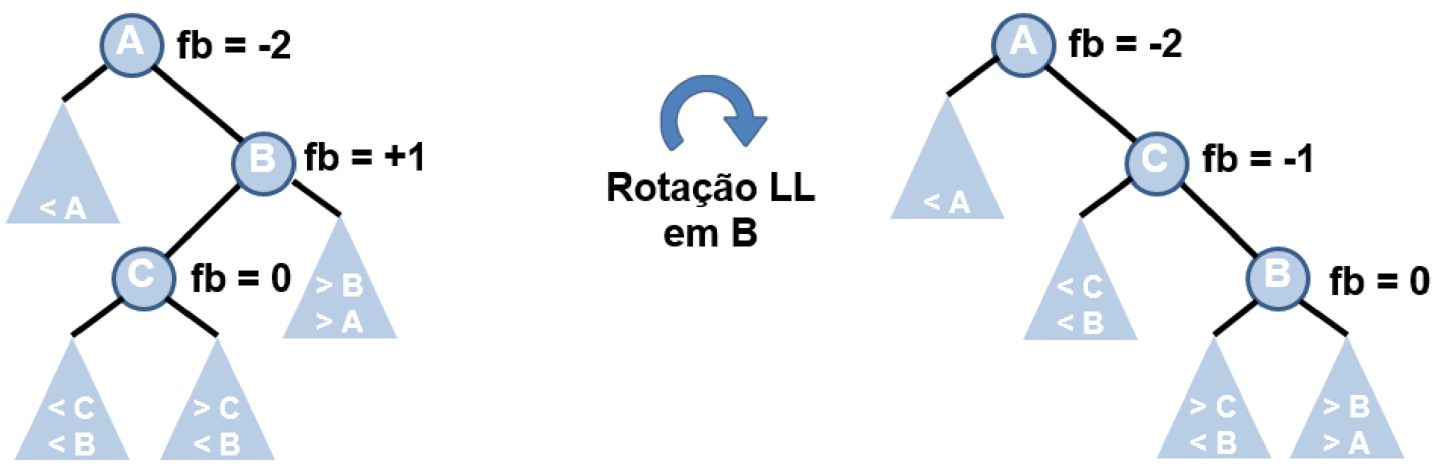
Árvore Balanceada

Rotação RL

- Rotação RL ou rotação dupla à esquerda
  - um novo nó é inserido na **sub-árvore da esquerda do filho direito** de **A**
    - **A** é o nó desbalanceado
    - Um movimento para a direita e outro para a esquerda: **RIGHT LEFT**
  - É necessário fazer uma rotação dupla, de modo que o nó **C** se torne o pai dos nós **A** (filho da esquerda) e **B** (filho da direita)
    - Rotação LL em **B**
    - Rotação RR em **A**

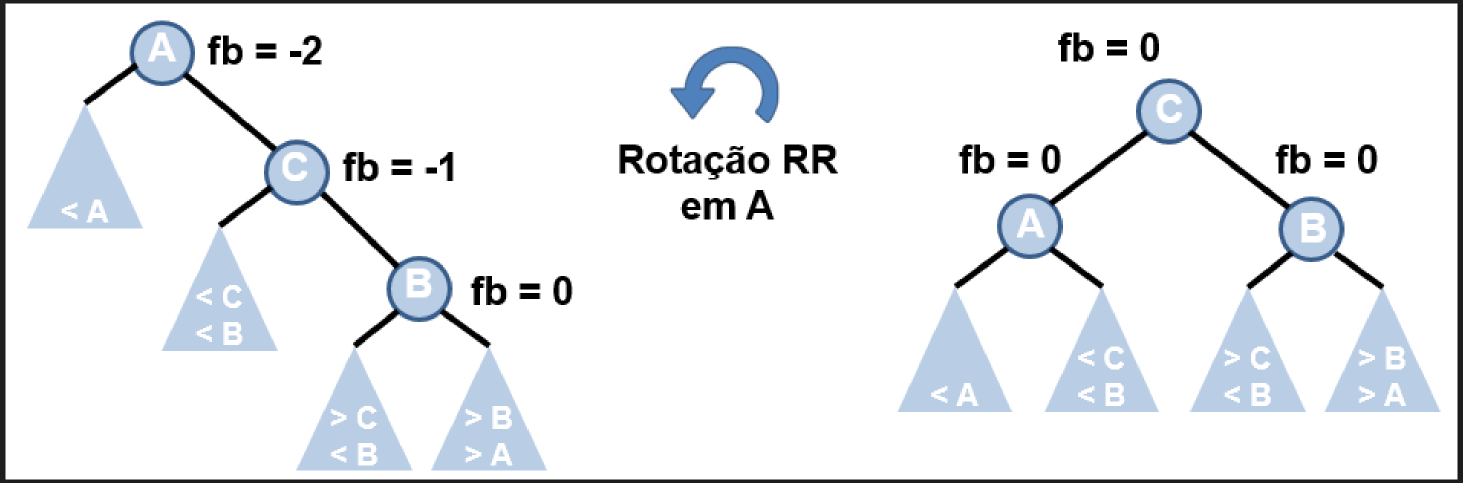
Rotação RL

```
void RotacaoRL(ArvAVL *raiz){
    RotacaoLL(&(*raiz)->dir);
    RotacaoRR(raiz);
}
```

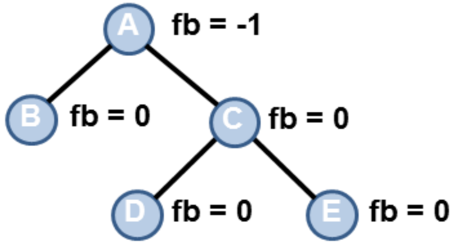


Rotação RL

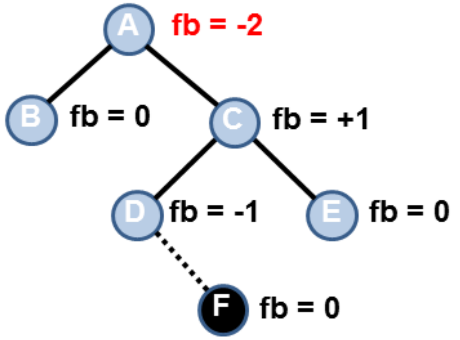
```
void RotacaoRL(ArvAVL *raiz){
    RotacaoLL(&(*raiz)->dir);
    RotacaoRR(raiz);
}
```



Rotação RL



Árvore AVL e fator de balanceamento de cada nó

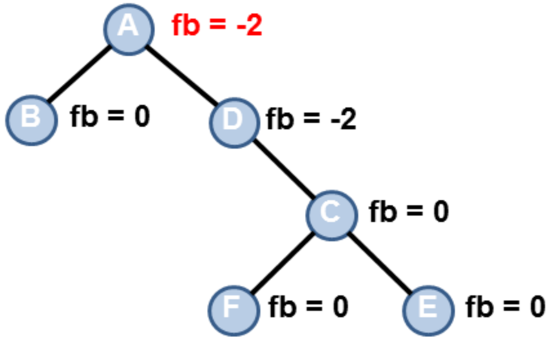


Inserção do nó F na árvore

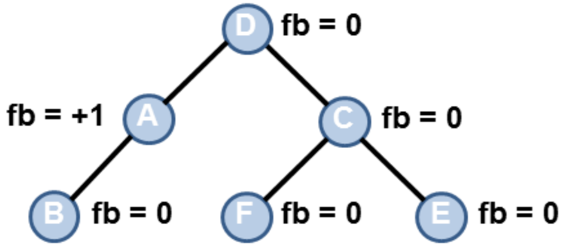
Árvore fica desbalanceada no nó A.

Aplicar Rotação RL no nó A. Isso equivale a:

- Aplicar a Rotação LL no nó C
- Aplicar a Rotação RR no nó A



Árvore após aplicar a Rotação LL no nó C



Árvore após aplicar a Rotação RR no nó A

Árvore Balanceada

Árvore AVL: Inserção

- Para inserir um valor **V** na árvore
  - Se a raiz é igual a **NULL**, insira o nó
  - Se **V** é menor do que a raiz: vá para a **sub-árvore esquerda**
  - Se **V** é maior do que a raiz: vá para a **sub-árvore direita**
  - Aplique o método **recursivamente**
- Dessa forma, percorremos um conjunto de nós da árvore até chegar ao nó folha que irá se tornar o pai do novo nó

Árvore AVL: Inserção

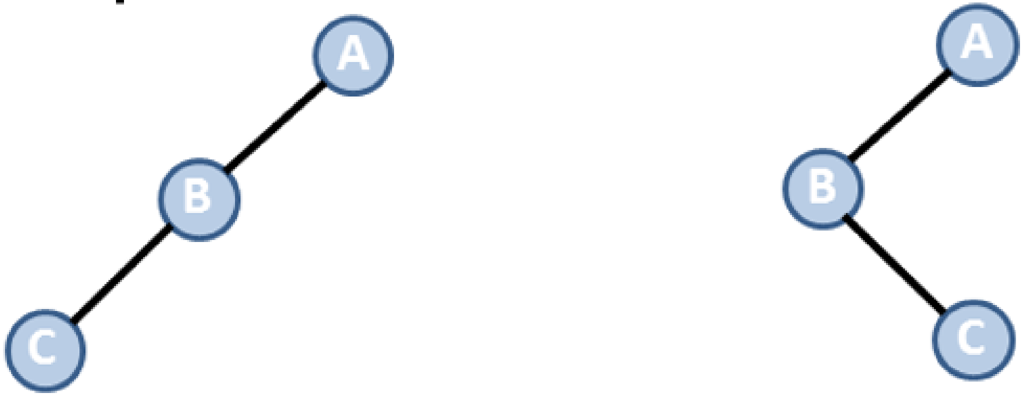
- Uma vez inserido o novo nó
  - Devemos voltar pelo caminho percorrido e calcular o fator de balanceamento de cada um dos nós visitados
  - Aplicar a rotação necessária para restabelecer o balanceamento da árvore se o fator de balanceamento for **+2** ou **-2**

Árvore AVL: Inserção

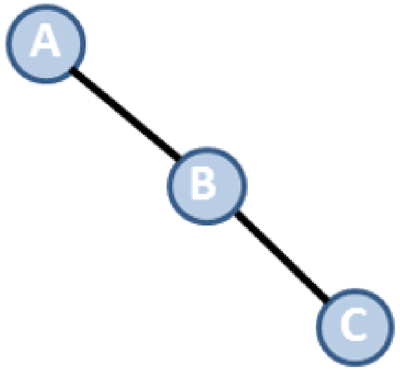
```c
int insere_ArvAVL(ArvAVL *raiz, int valor){
    int res;
    if(*raiz == NULL){//árvore vazia ou nó folha
        struct NO *novo;
        novo = (struct NO*)malloc(sizeof(struct NO));
        if(novo == NULL)
            return 0;

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

Árvore AVL: Inserção

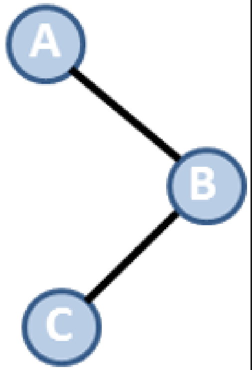
```c
//continuação
struct NO *atual = *raiz;
if(valor < atual->info){
    if((res=insere_ArvAVL(&(atual->esq), valor))==1){
        if(fatorBalanceamento_NO(atual) >= 2){
            if(valor < (*raiz)->esq->info ){
                RotacaoLL(raiz);
            }else{
                RotacaoLR(raiz);
            }
        }
    }
}
```



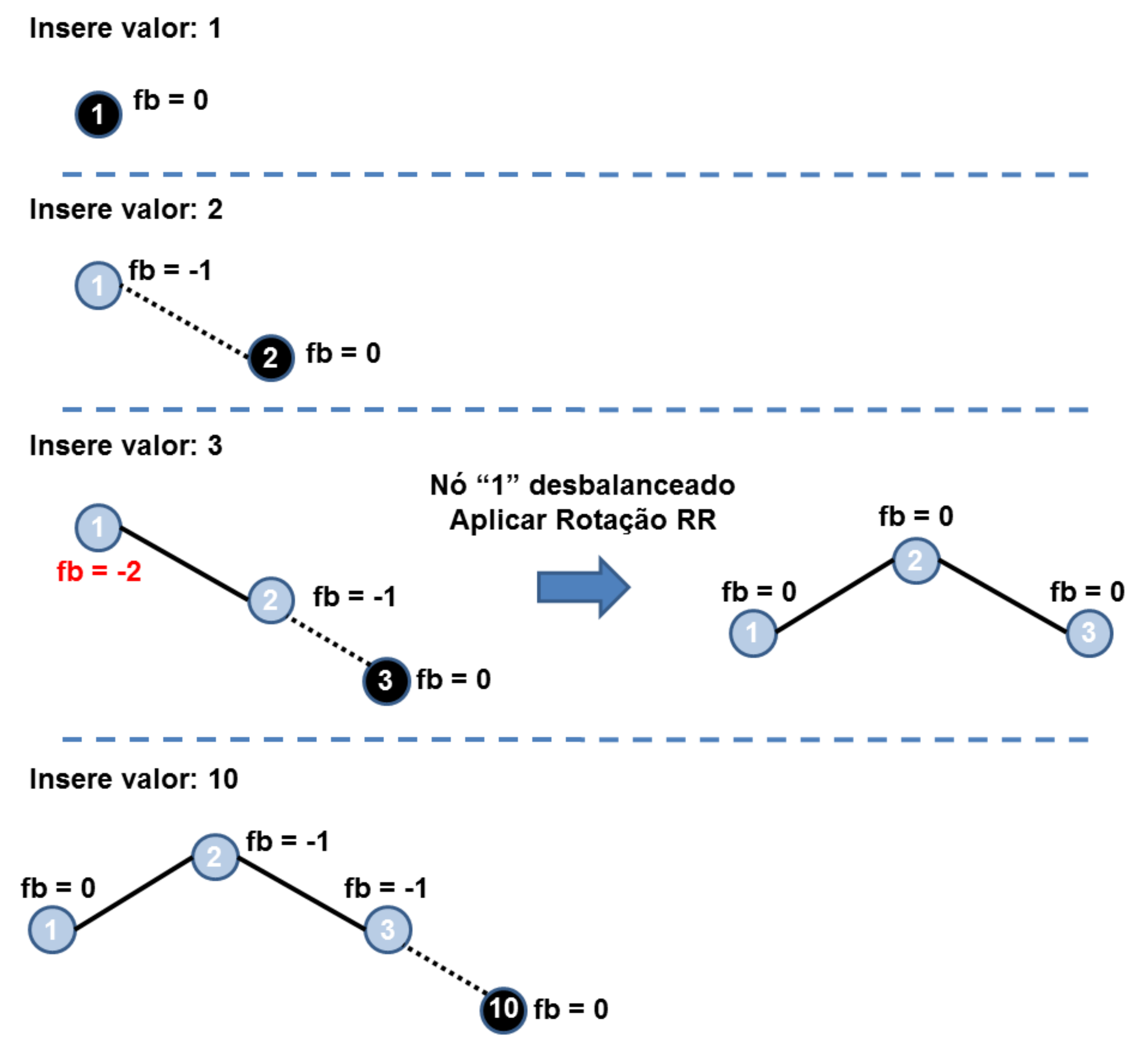
Árvore AVL: Inserção



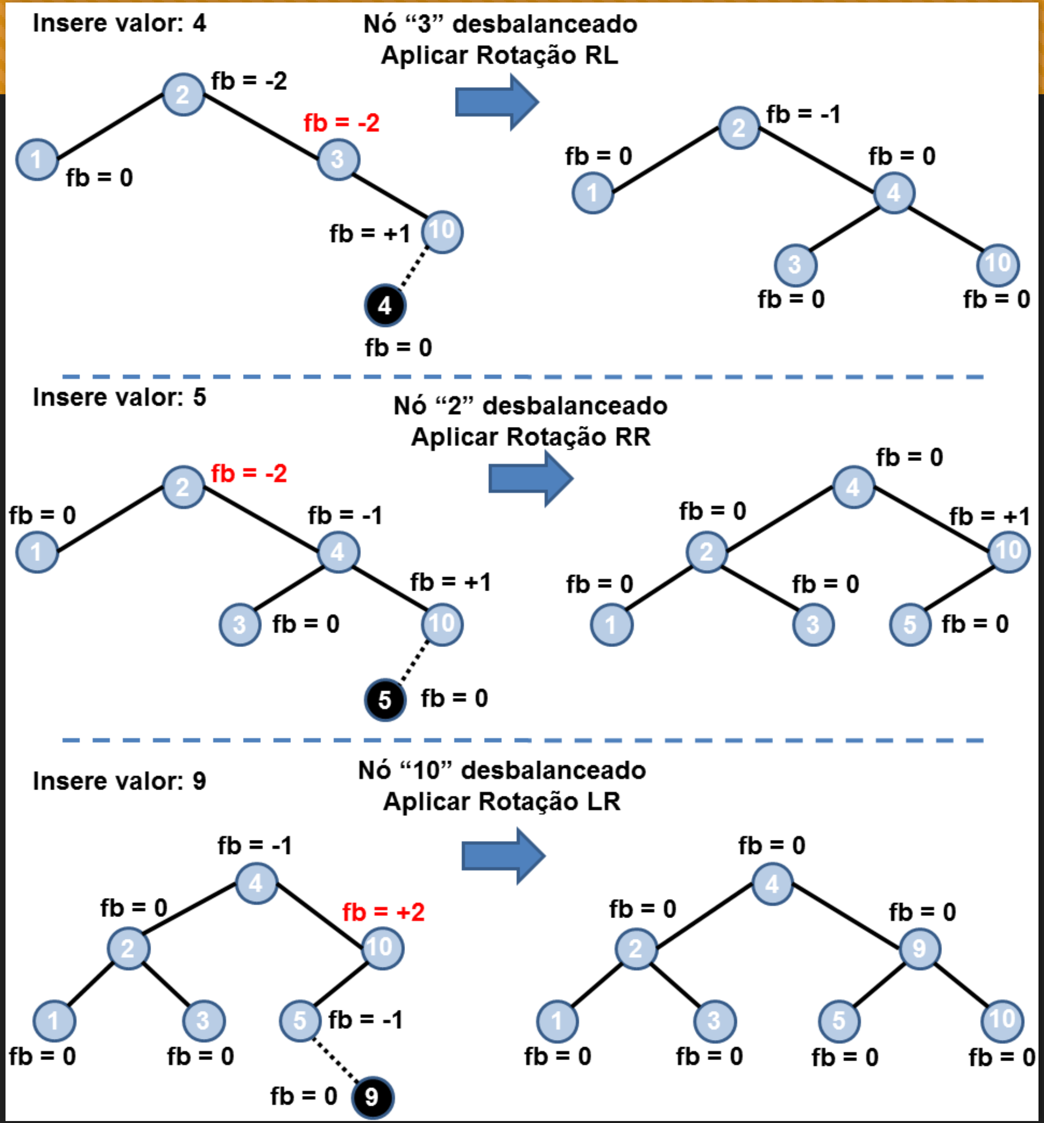

```c
//continuação
else{
    if(valor > atual->info){
      if((res=insere_ArvAVL(&(atual->dir), valor))==1){
        if(fatorBalanceamento_NO(atual) >= 2){
          if((*raiz)->dir->info < valor){
            RotacaoRR(raiz);
          }else{
            RotacaoRL(raiz);
          }
        }
    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



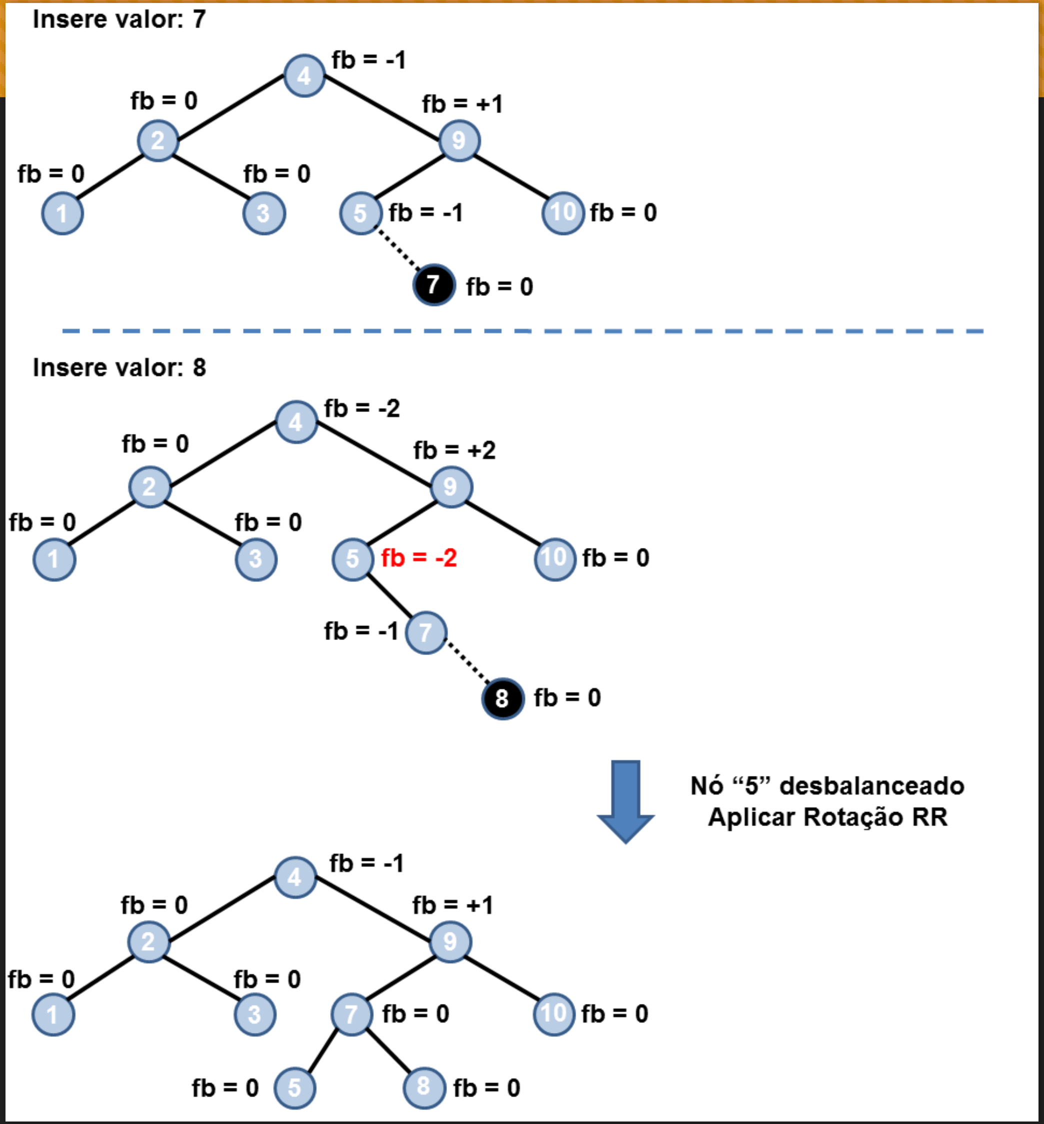
Árvore AVL: Inserção



Árvore AVL: Inserção



Árvore AVL: Inserção



Material Complementar

- Aula 67: Árvores: youtu.be/iLvpaqAoVD8
- Aula 68: Árvores: propriedades: youtu.be/U7IiLJlMfnU
- Aula 69: Árvore Binária: Definição: youtu.be/9WxCeWX9qDs
- Aula 70: Árvore Binária: Implementação: youtu.be/TR8ZLUKmcPc
- Aula 71: Criando e destruindo uma árvore binária: youtu.be/QAJkoJW8bEc
- Aula 72: Árvore Binária: informações básicas: youtu.be/qVnNdmx4fOA
- Aula 73: Percorrendo uma Árvore Binária: youtu.be/z7XwVVYQRAA
- Aula 74: Árvore Binária de Busca: youtu.be/M7cb4HjePJk
- Aula 75: Inserção em Árvore Binária de Busca: youtu.be/8cdbmsPaR-k

Material Complementar

- Aula 76: Remoção em Árvore Binária de Busca: youtu.be/_0Yu9BSYXGY
- Aula 77: Consulta em Árvore Binária de Busca: youtu.be/mw_wqqB48yY
- Aula 78 – Árvores Balanceadas: youtu.be/Au-6c55J90c
- Aula 79: Árvore AVL: Definição: youtu.be/4eO3UbTiRyo
- Aula 80: Árvore AVL: Implementação: youtu.be/I5cl39jdnow
- Aula 81: Árvore AVL: Tipos de Rotação: youtu.be/1HkWqH7L2rU
- Aula 82: Árvore AVL: Implementando as Rotações: youtu.be/6OJ8stXwdq0
- Aula 83: Árvore AVL: Inserção: youtu.be/lQsVUxa3Auk
- Aula 84: Árvore AVL: Remoção: youtu.be/F7_Daymw-WM