

Universidade Federal da Paraíba

Centro de Informática

Departamento de Informática

Estrutura de Dados

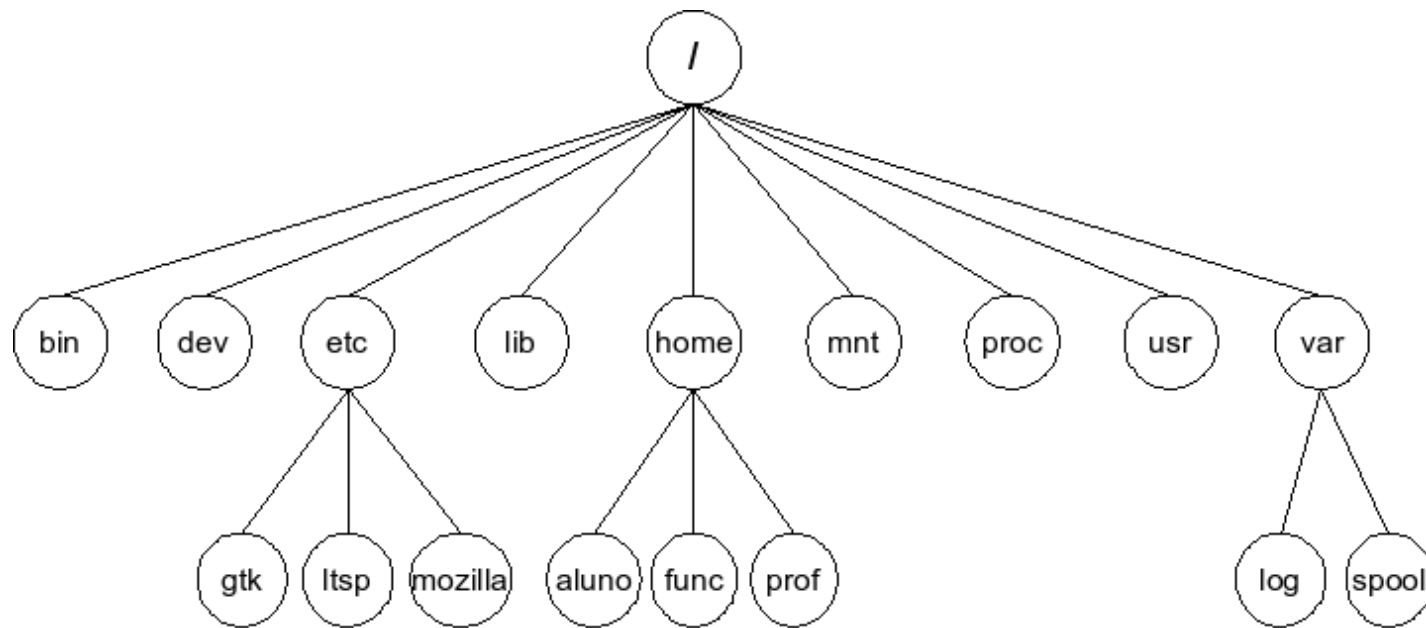
Árvores

▶ Tiago Maritan

▶ tiago@ci.ufpb.br

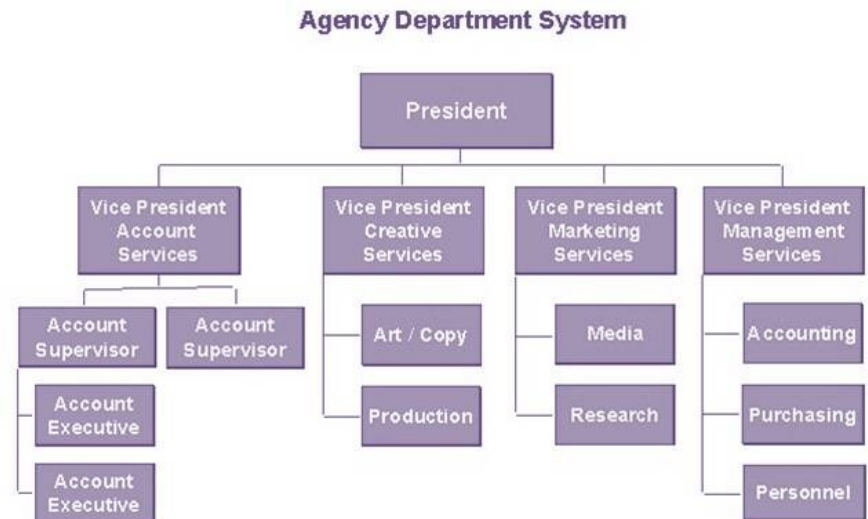
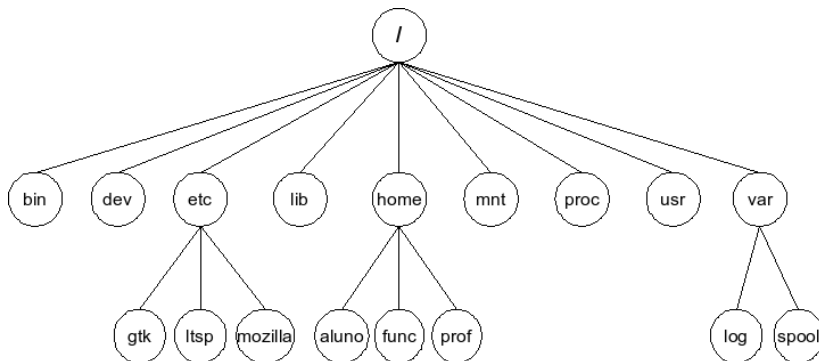
O que é uma Árvore?

- ▶ Em Ciência da Computação, uma árvore é um **modelo abstrato** de uma **estrutura hierárquica**



Árvores

- ▶ Uma árvore consiste de um **conjunto de nós** com uma **relação pai-filho** (hierárquica).
- ▶ Aplicações:
 - ▶ Sistemas de arquivos
 - ▶ Organogramas
 - ▶ Ambientes de Programação



Terminologia de Árvore

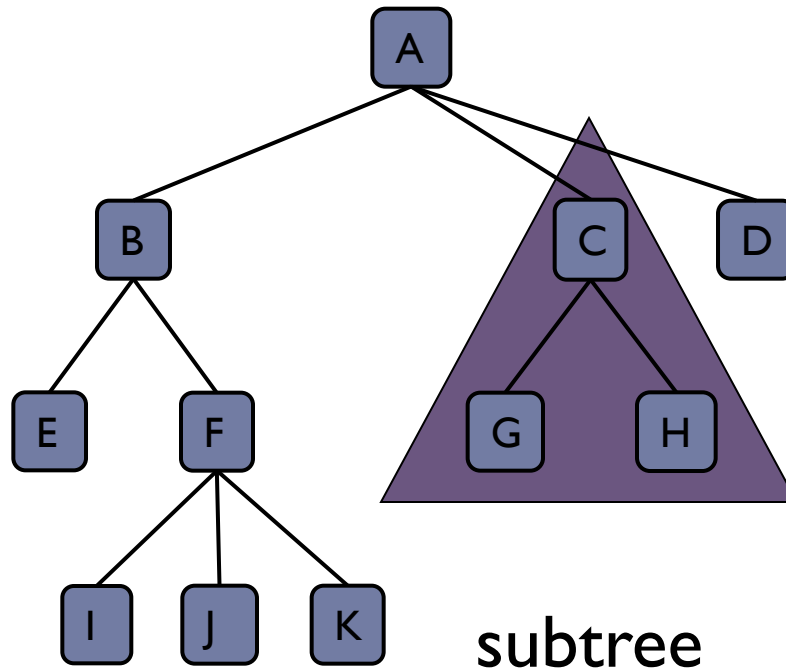
- ▶ **Raiz:** Primeiro nó da árvore... Não contém um pai (A)
- ▶ **Nó não-terminal:** nó com pelo menos um filho (A, B, C, F)
- ▶ **Nó folha (ou nó terminal) :** nó sem filhos (E, I, J, K, G, H, D)

- ▶ **Grau de um nó:** número máximo de filhos de um nó.
- ▶ **Ascendentes de um nó:** pai, avô, bisavô etc.
- ▶ **Profundidade de um nó:** número de ascendentes
- ▶ **Altura de uma árvore:** maior profundidade de qualquer nó
- ▶ **Descendente de um nó:** filho, neto, bisneto, etc.



Exemplo

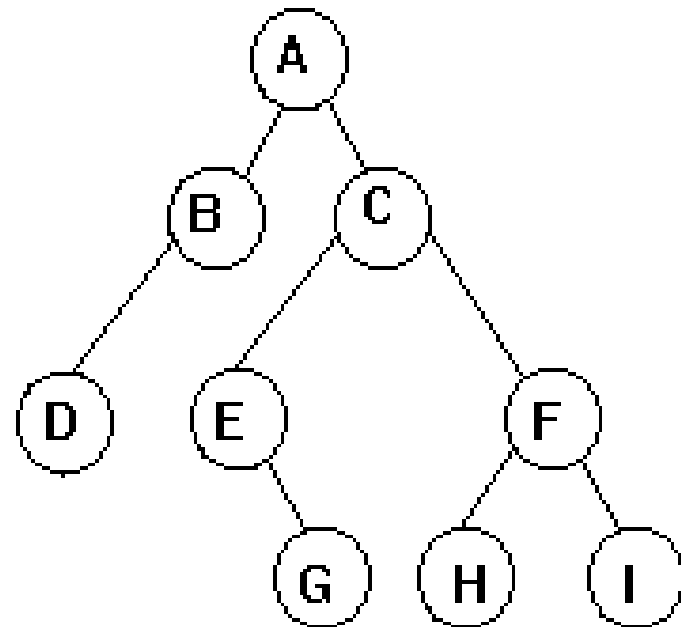
- **Sub-ávore:** árvore consistindo de um nó e seus descendentes



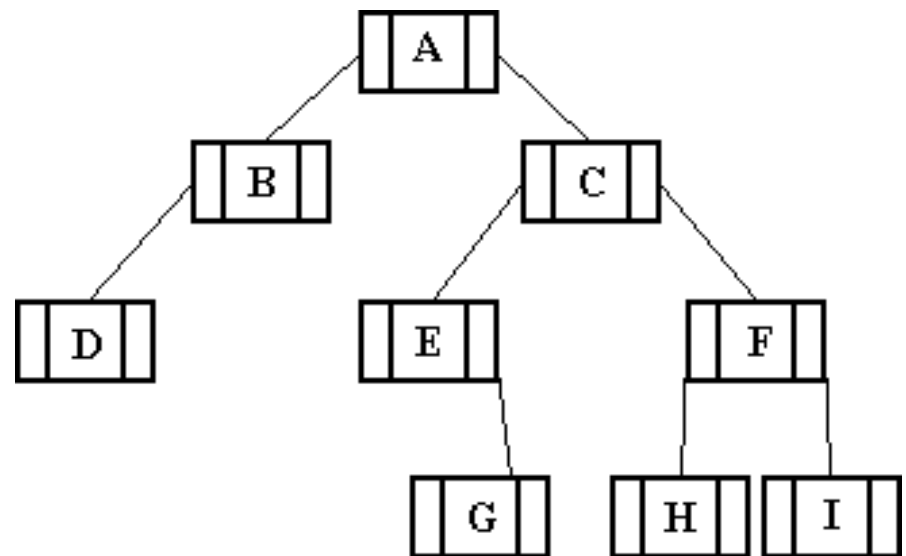
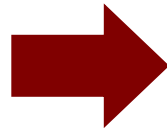
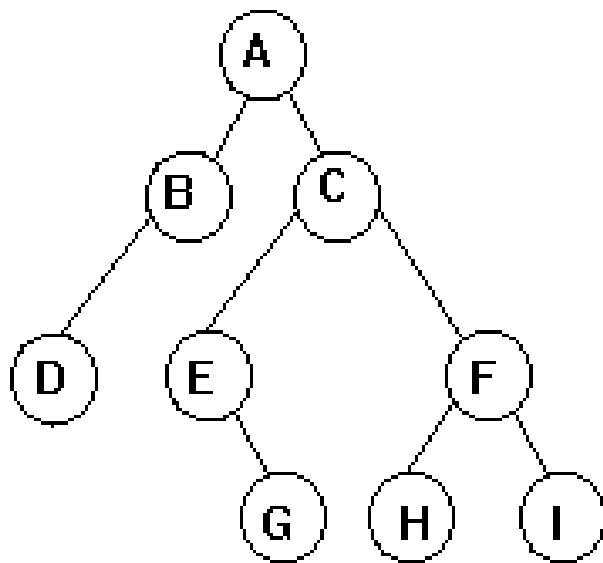
Árvores Binárias

Árvores Binárias

- ▶ É um tipo de Árvore, onde o **grau de cada nó** é no máximo **dois**.
- ▶ Cada nó tem no máximo 2 filhos.
- ▶ **Árvore Binária:** Conjunto de nós que ou é vazio ou consiste de:
 - ▶ Um nó raiz
 - ▶ Duas árvores binárias disjuntas:
 - ▶ **Sub-árvores direita e**
 - ▶ **Sub-árvore esquerda.**



Implementação de Árvores Binárias



Implementação de Árvores Binárias

▶ Interface

- ▶ Criar uma árvore vazia;
- ▶ Verificar se a árvore está vazia ou não;
- ▶ Buscar um elemento na árvore;
- ▶ Inserir um nó raiz;
- ▶ Inserir um filho à direita de um nó;
- ▶ Inserir um filho à esquerda de um nó;
- ▶ Exibir a árvore.

Implementação de Árvores Binárias

```
/* Definição da Estrutura de Dados */

typedef struct no {
    int conteudo;      /* conteudo */
    struct no *esq;    /* ref. para filho da esquerda */
    struct no *dir;    /* ref. para filho da direita */
} tNo;                /* tipo do nó */

typedef tNo *tArvBin; /* tipo árvore binária */
```

Implementação de Árvores Binárias

```
/* Definição das Operações */

//Cria uma árvore vazia
void cria (tArvBin *T) {
    *T = NULL;
}

//Verifica se a árvore está vazia
int vazia (tArvBin T) {
    return (T == NULL);
}
```

Implementação de Árvores Binárias

```
//Buscar um elemento na árvore
//Retorna o endereço se o elemento for
// encontrado, caso contrário retorna NULL
tArvBin busca(tArvBin T, int dado) {
    tArvBin achou;
    if (T == NULL)
        return NULL; // Arvore Vazia

    if(T->conteudo == dado)
        return T; //Elem. encontrado na raiz

    achou = busca(T->esq, dado);
    if (achou == NULL)
        achou = busca(T->dir, dado);

    return achou;
}
```

Implementação de Árvores Binárias

```
// Insere um nó raiz em uma árvore vazia
// Retorna 1 se a inserção for com sucesso.
// Caso contrário 0
int insereRaiz(tArvBin *T, int dado ) {
    tNo *novoNo;
    if (*T != NULL)
        return (0);    //Erro: árvore não está vazia

    novoNo = malloc(sizeof(tNo));
    if (novoNo == NULL) return 0;    //Err: mem.  Insuf.

    novoNo->conteudo = dado;
    novoNo->esq = NULL;
    novoNo->dir = NULL;
    *T = novoNo;
    return 1;
}
```

Implementação de Árvores Binárias

```
// Insere um filho à direita de um dado nó
// Retorna 1 se a inserção for com sucesso,
// Caso contrário 0
int insereDireita(tArvBin T, int vPai, int vFilho ) {
    tNo *f, *p, *novoNo;
    //Verifica se o elemento já não existe
    f = busca(T, vFilho);
    if (f != NULL) return 0;    //Err: dado já existe

    //Busca o pai e verifica se possui filho direito
    p = busca(T, vPai);
    if (p == NULL)
        return 0; //Err: pai não encontrado
    if (p->dir != NULL)
        return 0; //Err: filho dir já existe
    //continua...
```

Implementação de Árvores Binárias

```

novoNo = malloc(sizeof(tNo));
if (novoNo == NULL)
    return (0);    //Err: mem. insuf.

novoNo->conteudo = vFilho;
novoNo->esq = NULL;
novoNo->dir = NULL;
p->dir = novoNo;
return 1;
}
```

Implementação de Árvores Binárias

```
//Insere um filho à esquerda de um dado nó
//Retorna 1 se a inserção for com sucesso,
// caso contrário 0
int insereEsq(tArvBin T, int vPai, int vFilho ) {
    tNo *f, *p, *novoNo;
    //Verifica se o elemento já não existe
    f = busca(T, vFilho);
    if (f != NULL) return 0;    //Err: dado já existe

    //Busca o pai e verifica se possui filho direito
    p = busca(T, vPai);
    if (p == NULL)
        return 0; //Err: pai não encontrado
    if (p->esq != NULL)
        return 0; //Err: filho esq já existe
    //continua...
```



Implementação de Árvores Binárias

```
    novoNo = malloc(sizeof(tNo));  
    if (novoNo == NULL)  
        return (0);    //Err: mem. insuf.  
  
    novoNo->conteudo = vFilho;  
    novoNo->esq = NULL;  
    novoNo->dir = NULL;  
    p->esq = novoNo;  
    return 1;  
}
```

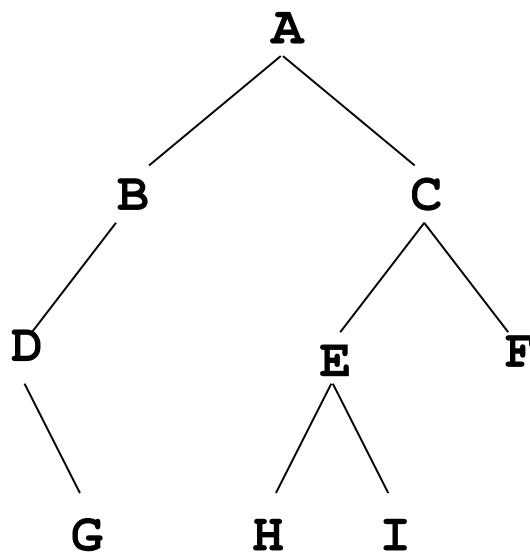
Caminhamento em Árvores Binárias

Caminhamento em Árvores Binárias

- ▶ Ação de percorrer (**visitar**) todos os nós de uma árvore
 - ▶ Cada nó é visitado uma única vez.
- ▶ Em geral, objetivo é executar alguma operação nestes nós
 - ▶ Ex: imprimir, consultar, alterar, etc.
- ▶ Contudo, em **árvores** não há nenhuma **ordem linear natural** para se visitar os nós
 - ▶ Podem-se definir várias ordens de caminhamento.
 - ▶ Listas, no entanto, são estruturas lineares porque existe uma **ordem linear natural** para se visitar os nós.
Ex: do início ao fim da lista.

Caminhamento em Árvores Binárias

- ▶ Três tipos de caminhamento em AVBin mais comuns são:
 1. Caminhamento em **ordem prefixa (pré-ordem)**;
 2. Caminhamento em **ordem infixa (in-ordem)**;
 3. Caminhamento em **ordem sufixa (pós-ordem)**.



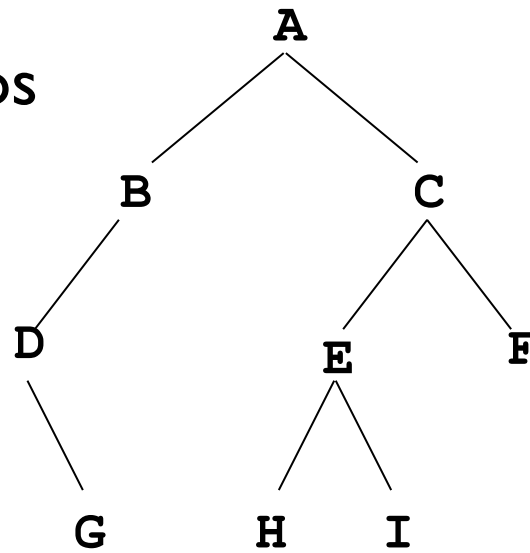
Caminhamento em Árvores Binárias

- ▶ Esses métodos de caminhamento em AVBin são geralmente definidos de **forma recursiva**.
- ▶ Idéia geral:
 - ▶ **Visitar a raiz, e**
 - ▶ **Caminhar em suas sub-árvores esquerda e direita.**

Caminhamento em Pré-ordem

- ▶ 1. Visite a raiz;
- ▶ 2. Caminhe na sub-árvore esquerda em pré-ordem;
- ▶ 3. Caminhe na sub-árvore direita em pré-ordem.

Sequência de visitação dos
nós: ABDGCEHIF



Caminhamento em Pré-ordem

```
//Exibe o conteúdo de uma árvore em pré-ordem
void exibePreOrdem(tArvbin T) {
    if (T == NULL)
        return ;

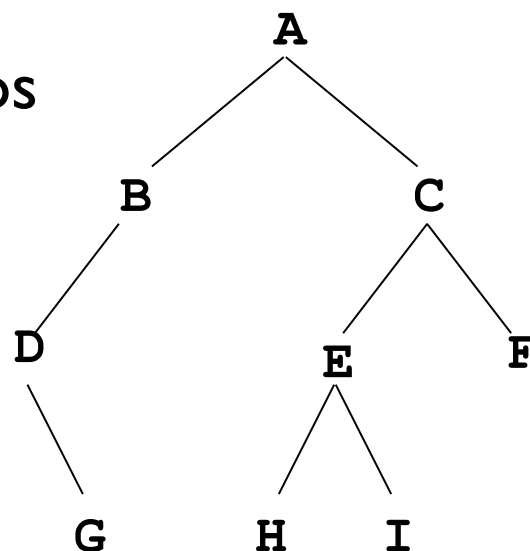
    printf("%d    ", T->conteudo) ;
    if (T->esq != NULL)
        exibePreOrdem(T->esq) ;

    if (T->dir != NULL)
        exibePreOrdem(T->dir) ;
}
```

Caminhamento em In-ordem

- ▶ 1. Caminhe na sub-árvore esquerda em in-ordem;
- ▶ 2. Visite a raiz.
- ▶ 3. Caminhe na sub-árvore direita em in-ordem;

Sequência de visitação dos
nós: **DGBAHEICF**



Caminhamento em In-ordem

```
//Exibe o conteúdo de uma árvore em pré-ordem
void exibeInOrdem(tArvbin T) {
    if (T == NULL)
        return ;

    if (T->esq != NULL)
        exibeInOrdem(T->esq) ;

    printf("%d    ", T->conteudo) ;

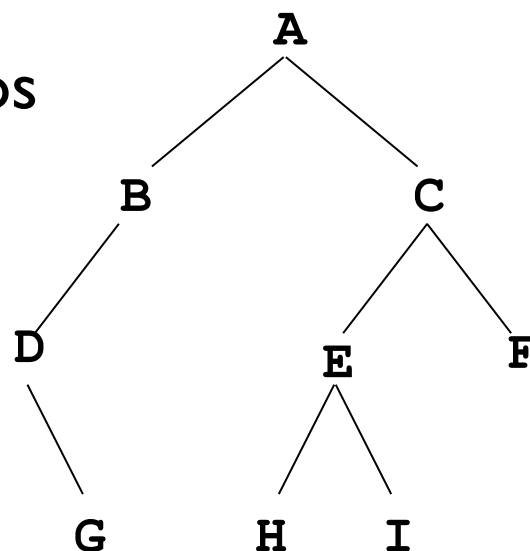
    if (T->dir != NULL)
        exibeInOrdem(T->dir) ;

}
```

Caminhamento em Pós-ordem

- ▶ 1. Caminhe na sub-árvore esquerda em pós-ordem;
- ▶ 2. Caminhe na sub-árvore direita em pós-ordem;
- ▶ 3. Visite a raiz.

Sequência de visitação dos
nós: **GDBHIEFCA**



Caminhamento em Pós-ordem

```
//Exibe o conteúdo de uma árvore em pré-ordem
void exibePosOrdem(tArvbin T) {
    if (T == NULL)
        return ;

    if (T->esq != NULL)
        exibePosOrdem(T->esq);
    if (T->dir != NULL)
        exibePosOrdem(T->dir);

    printf("%d  ", T->conteudo);

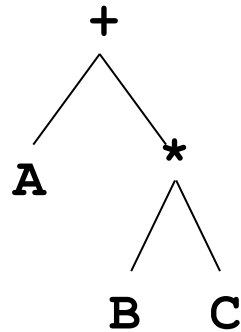
}
```

Caminhamento em Árvores Binárias

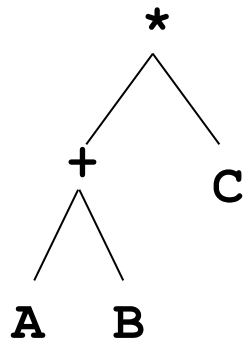
- ▶ Árvores binárias podem ser utilizadas para representar **expressões aritméticas**.
- ▶ Neste método de representação:
 - ▶ Raiz da árvore contém o operador que deve ser aplicado;
 - ▶ aos resultados das avaliações das expressões aritméticas representadas pelas sub-árvores esquerda e direita.
- ▶ Um nó que representa um operador (binário) possui duas sub-árvores não-vazias, enquanto que um nó que representa um operando não possui sub-árvores (i.e., é um nó-folha).

Caminhamento em Árvores Binárias

- Assim, a expressão $A + B * C$ seria representada pela árvore:

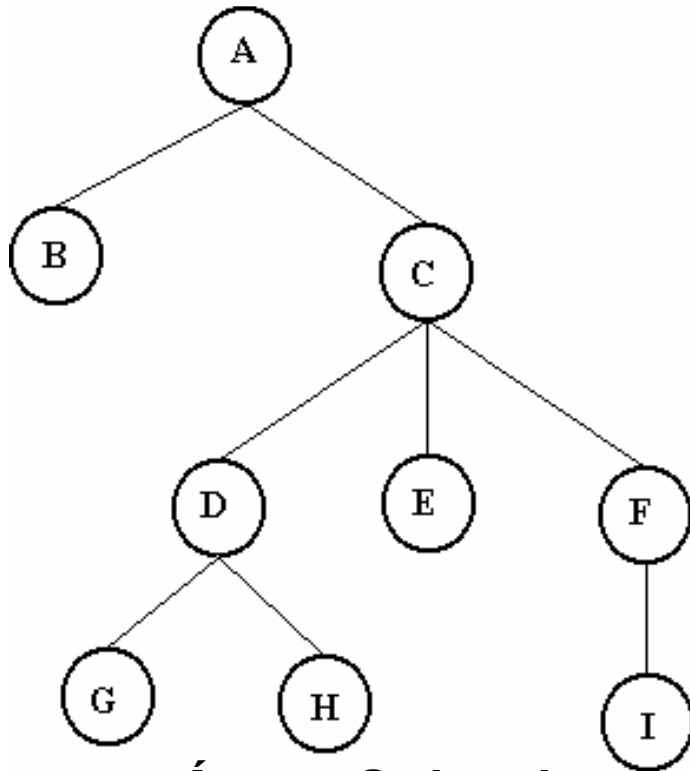


- Enquanto que a expressão $(A + B) * C$ seria representada como:

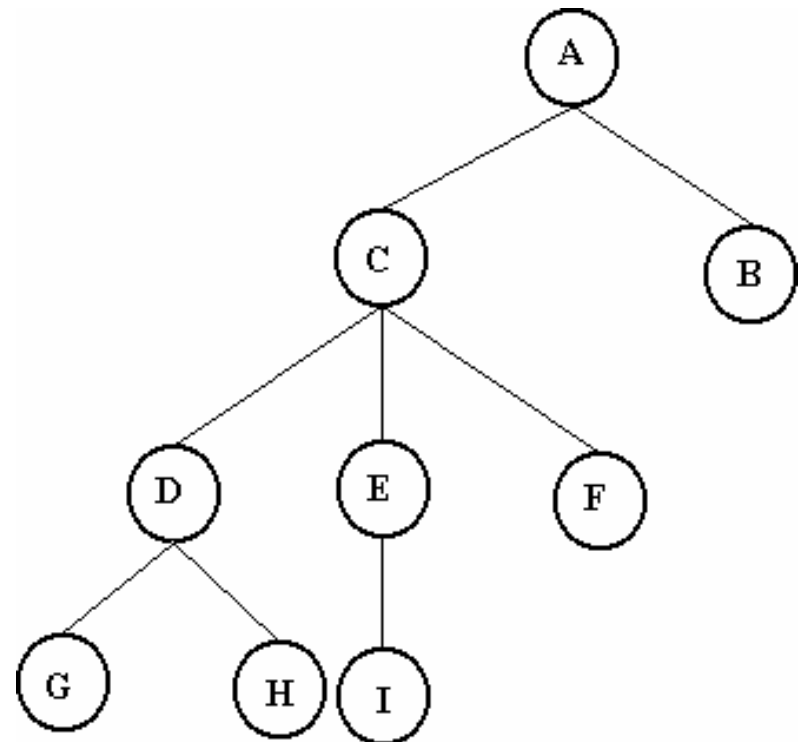


Árvores Ordenadas

- ▶ **Árvore ordenada:** aquela na qual os filhos de cada nó estão ordenados.
- ▶ Assume-se ordenação da esquerda para a direita.



Árvore Ordenada

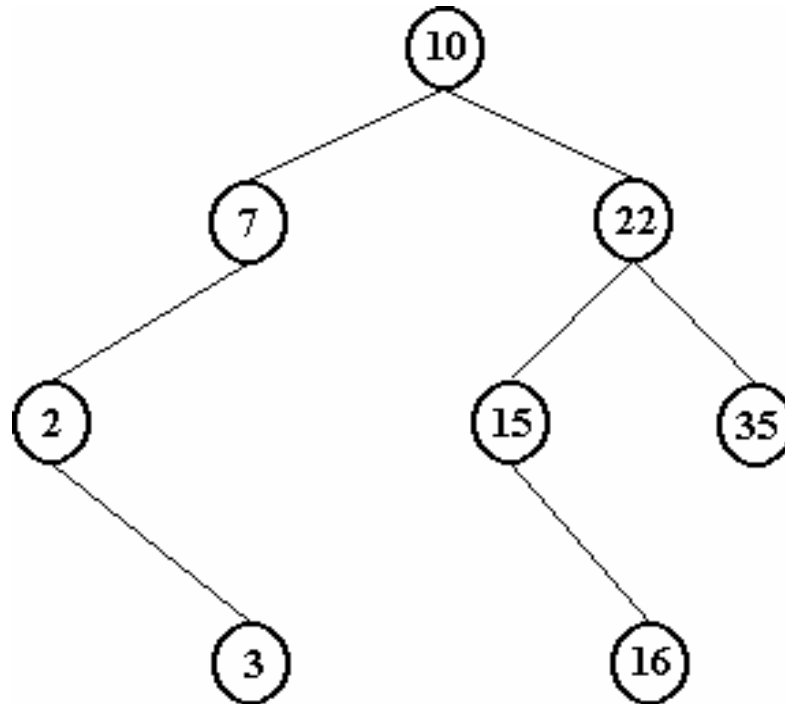


Árvore Não-Ordenada

Árvore Binária de Pesquisa

Árvore Binária de Pesquisa (ABP ou Arvore Binária de Busca)

- ▶ Árvores que são vazias ou que o nó raiz contém uma chave e:
 - ▶ Chaves da subárvore esquerda $<$ chave da raiz.
 - ▶ Chaves da subárvore direita $>$ chave da raiz.
 - ▶ Subárvores direita e esquerda são também ABP.



Implementação de ABPs

▶ Interface

- ▶ Criar uma árvore vazia;
- ▶ Verificar se a árvore está vazia ou não;
- ▶ Buscar um elemento na árvore;
- ▶ Inserir um nó na árvore;
- ▶ Exibir a árvore;

Implementação de ABPs

```
/* Definição da Estrutura de Dados */  
  
typedef struct no {  
    int info;  
    struct no *esq; /* nó esquerdo */  
    struct no *dir; /* nó direito */  
} tNo;              /* nó da árvore */  
  
typedef tNo *tAbp; /* ponteiro para a raiz da ABP */
```

Implementação de ABPs

```
/* Definição das Operações */

//Cria uma árvore vazia
void cria (tAbp *T) {
    *T = NULL;
}

//Verifica se a árvore está vazia
int vazia (tAbp T) {
    return (T == NULL);
}
```

Implementação de ABPs

```
//Buscar um elemento na árvore
//Retorna o endereço se o elemento for encontrado,
//Caso contrário retorna NULL
tAbp busca(tAbp T, int dado) {
    tAbp achou;
    if (T == NULL) return NULL;    //Err: arv. Vazia;

    if (T->info == dado)
        return T;

    if (T->info > dado)
        return busca(T->esq, dado);
    else
        return busca(T->dir, dado);
}
```

Implementação de ABPs

```
//Exibe o conteúdo de uma árvore no formato in-ordem
//(preserva a ordenação)
void exibe (tAbp T) {
    if (T != NULL) {
        exibe(T->esq);
        printf("%d    ", T->info);
        exibe(T->dir);
    }
}
```

Implementação de ABPs

```
//Insere um nó em uma árvore ABP
//Retorna 1 se a inserção for com sucesso.
//Caso contrário retorna 0
int insere (tAbp *T, int item) {
    tNo *novoNo, *atual, *p;
    novoNo = malloc(sizeof(tno));
    if (novoNo == NULL) return 0;

    novoNo->info = item;
    novoNo->esq = NULL;
    novoNo->dir = NULL;

    if ((*T) == NULL) { // Arvore vazia
        (*T) = novoNo;
        return 1;
    }
    // continua...
```

Implementação de ABPs

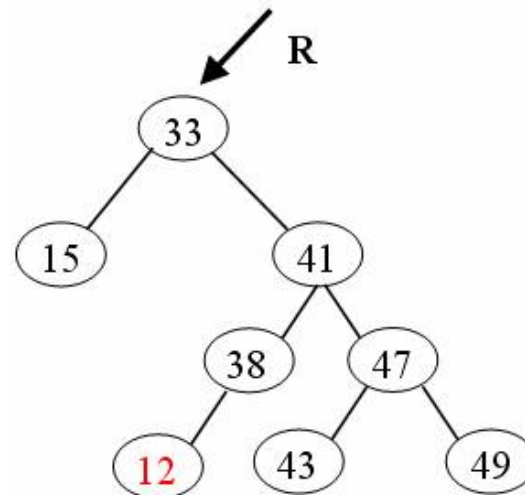
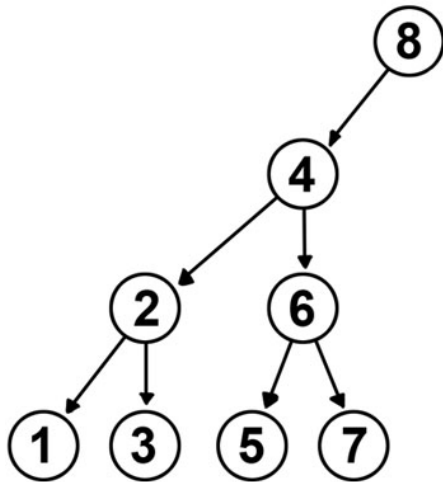
```
atual = (*T);
while (atual != NULL) {
    p = atual;
    if (atual->info > item)
        atual = atual->esq;
    else
        atual = atual->dir;
}
// Encontrou um nó folha para inserir
if (p->info > item)
    p->esq = novoNo;
else
    p->dir = novoNo;

return 1;
}
```

Árvore Binária de Pesquisa Balanceada (AVL)

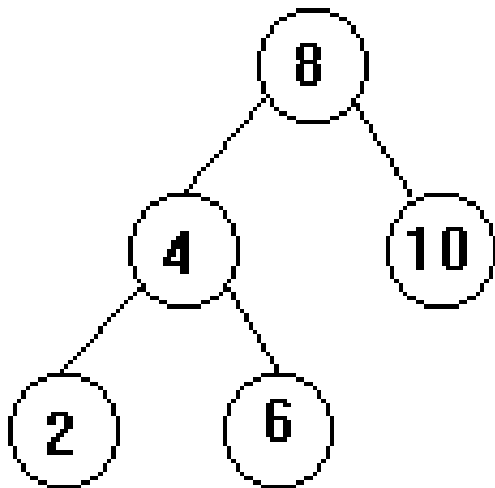
ABPs Desbalanceadas

- ▶ Inserções e remoções sucessivas em uma ABP podem torná-las **desequilibradas** ou **desbalanceadas**.
 - ▶ Sub-árvores da esquerda e direita tem alturas muito diferentes;
- ▶ Desbalanceamento pode resultar em diferenças de desempenho para acessar aos nós da ABP.



ABPs Balanceada

- ▶ Uma ABP é dita **balanceada** quando:
 - ▶ Para qualquer nó, as sub-árvores à esquerda e à direita possuem a mesma altura (ou alturas similares).

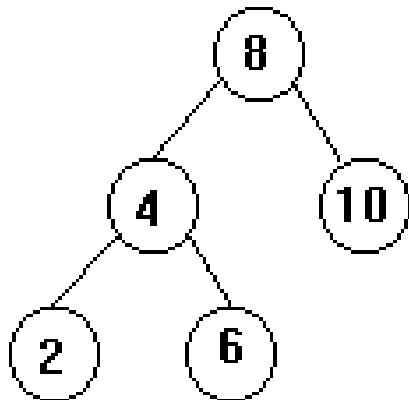


Árvore Binária de Pesquisa Balanceada

- ▶ O balanceamento de uma árvore binária pode ser:
 - ▶ Estático
 - ▶ Dinâmico (AVL ou Árvores Rubro-negras)
- ▶ **Balanceamento estático:** consiste em construir uma nova versão da árvore, reorganizando-a.
- ▶ **Balanceamento dinâmico:** a cada nova operação realizada na ABP, são realizadas rotações, para torná-la, novamente, balanceada.
 - ▶ AVL ou Árvores Rubro-negras

Árvores AVL

- ▶ O termo AVL foi definido em homenagem aos matemáticos russos **Adelson-Velskii** e **Landis**.
- ▶ Árvores AVL são ABPs onde a diferença em altura entre as sub-árvores esquerda e direita é no máximo 1.



Árvores AVL

- ▶ Uma árvore AVL é mais eficiente nas suas operações de busca do que uma ABP degenerada (ou desbalanceada),
 - ▶ O nº comparações diminui sensivelmente.
- ▶ Ex: Árvore com 10.000 nós.
 - ▶ Árvore AVL: média de 14 comparações em uma busca.
 - ▶ Árvore degenerada: média de 5.000 comparações, numa busca;

Inserção em Árvores AVL

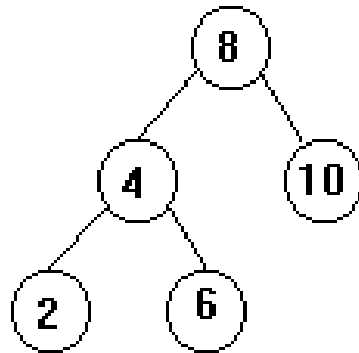
- ▶ O que pode acontecer quando um novo nó é inserido numa árvore balanceada ?
- ▶ Seja uma árvore com subárvores E (esq) e D (dir), e supondo que a inserção deve ser feita na sub-árvore da esquerda.

Podemos distinguir 3 casos:

1. **$hE = hD$** : E e D ficam com alturas diferentes mas continuam balanceadas.
2. **$hE < hD$** : E e D ficam com alturas iguais e balanceamento foi melhorado.
3. **$hE > hD$** : E fica ainda maior e o balanceamento foi violado.

Inserção AVL

- ▶ Ex: Suponha a árvore abaixo:



- ▶ Inserção de 9 ou 11 não desbalanceia a árvore.
 - ▶ Subárvore com raiz 10 passa a ter uma subárvore e
 - ▶ Subárvore com raiz 8 vai ficar melhor balanceada !
- ▶ Inserção dos nós 3, 5 ou 7 deixam a árvore desbalanceada
 - ▶ Requerem que a árvore seja rebalanceada!

Inserção AVL

- ▶ **Fator de Balanceamento (FB) de um nó:** corresponde a altura da subárvore direita do nó menos a altura da subárvore esquerda do nó.

$$\text{FB} = hD(n) - hE(n)$$

- ▶ Seja **n** um nó qualquer da árvore:
 - ▶ se $FB(n) = 0$, as duas sub-árvores têm a mesma altura;
 - ▶ se $FB(n) = -1$, a sub-árvore esquerda é mais alta em 1 nível;
 - ▶ se $FB(n) = +1$, a sub-árvore direita é mais alta em 1 nível;

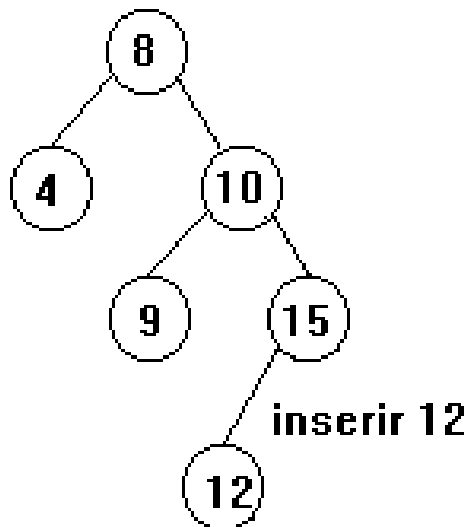
Inserção AVL

- ▶ Como manter então uma árvore AVL após as inserções?
- ▶ Solução: a cada inserção, rebalanceia a árvore.
 - ▶ Fazer as correções necessárias para manter sempre a árvore como uma AVL
 - ▶ Qualquer nó n deve ter $|\text{FB}(n)| \leq 1$.

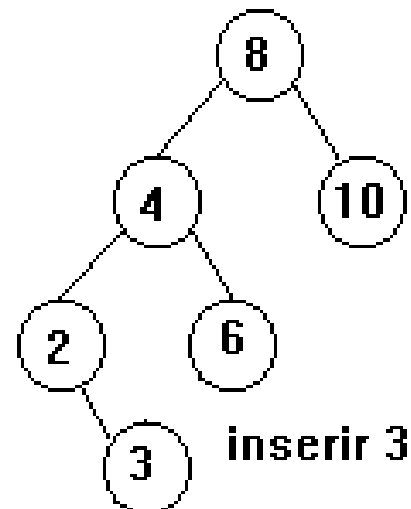
Inserção AVL - Rebalanceamento

- ▶ **Caso I:** Raiz de uma subárvore tem $FB = 2$ (ou -2) e um filho com $FB = 1$ (-1), e os dois tem o mesmo sinal

Exemplo 1:



Exemplo 2:



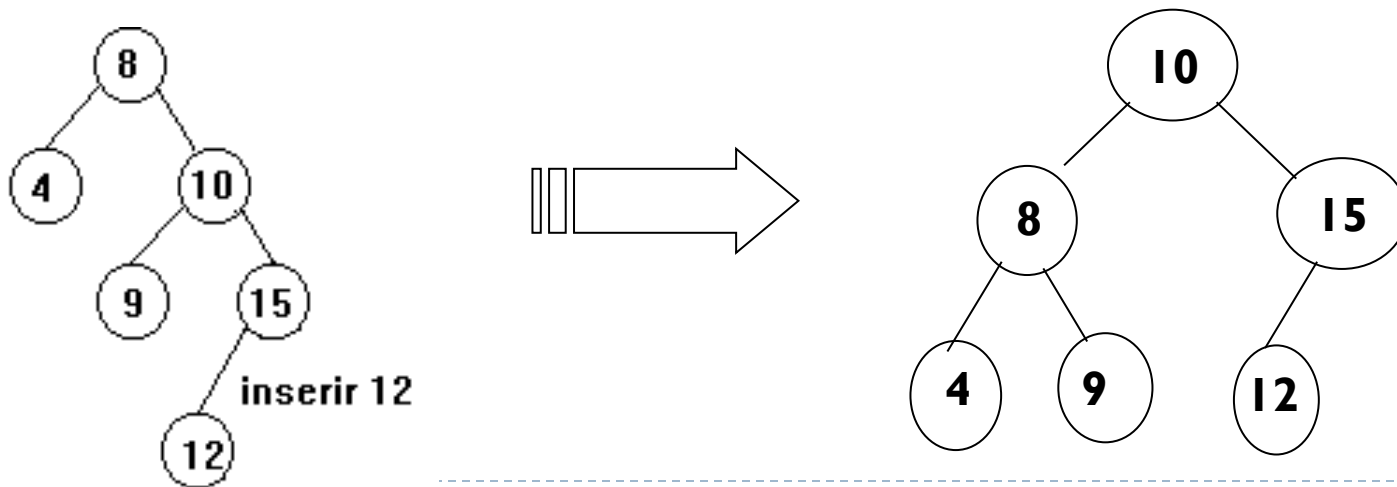
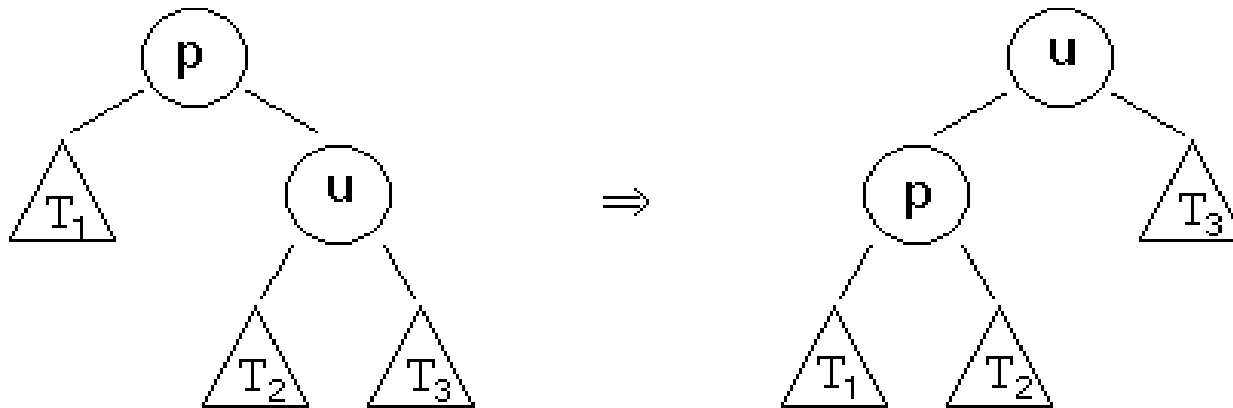
Solução: Rotação simples sobre o nó de $FB=2$ (-2).

Rotações são feitas à esquerda quando FB positivo e à direita quando FB negativo.

Inserção AVL - Rebalanceamento

► Rotação à esquerda

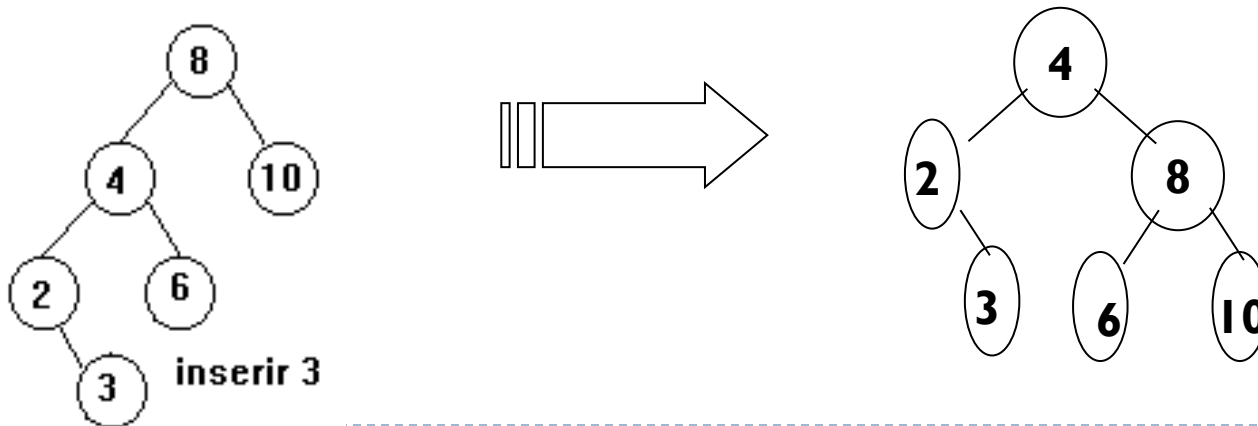
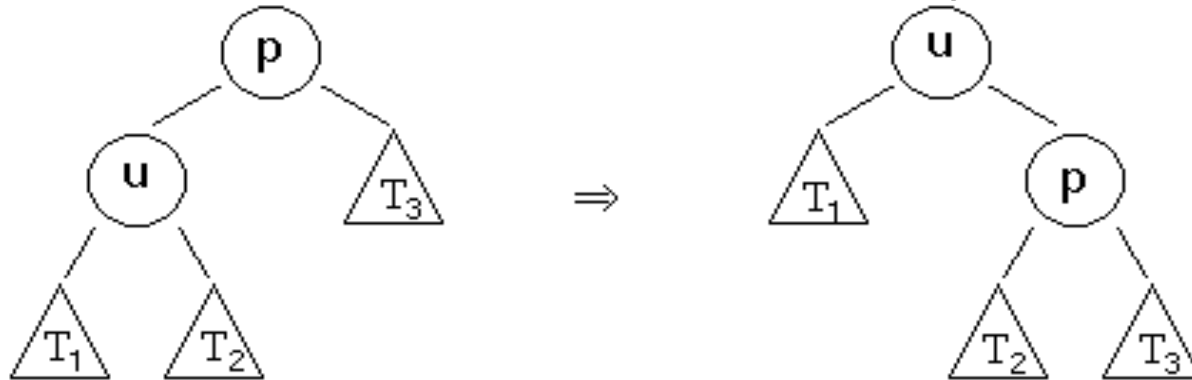
As subárvores T_1, T_2, T_3 podem ser vazias ou não.



Inserção AVL - Rebalanceamento

► Rotação à direita

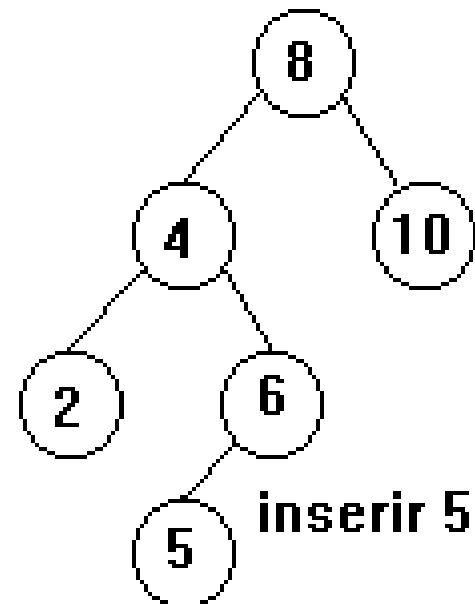
As subárvores T_1, T_2, T_3 podem ser vazias ou não.



Inserção AVL - Rebalanceamento

- ▶ **Caso 2:** Raiz de uma subárvore tem $FB = 2$ (ou -2) e um filho com $FB = 1$ (-1), e os dois tem sinais opostos.

- ▶ Exemplo: $FB(8) = -2$, $FB(4) = 1$

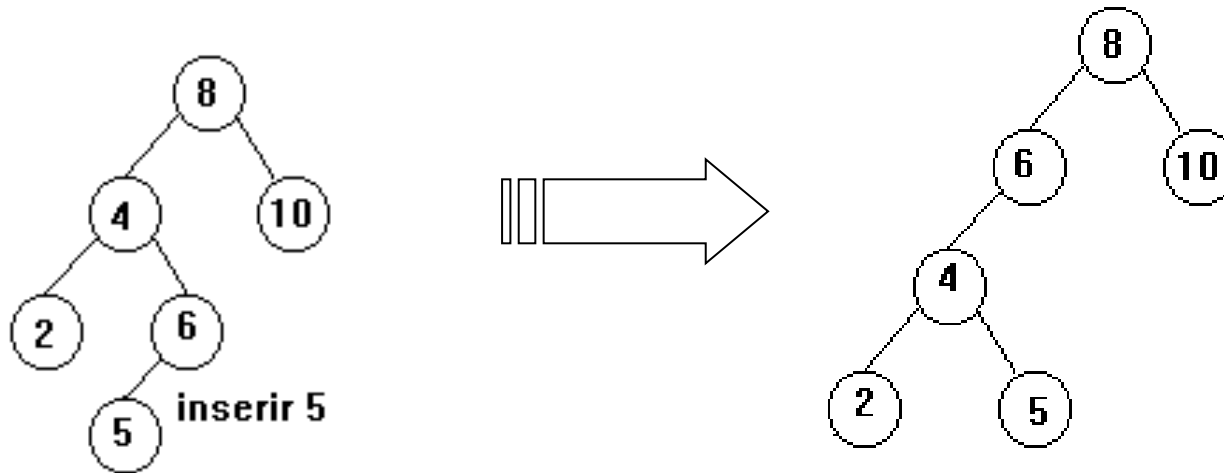


Solução: Duas rotações:

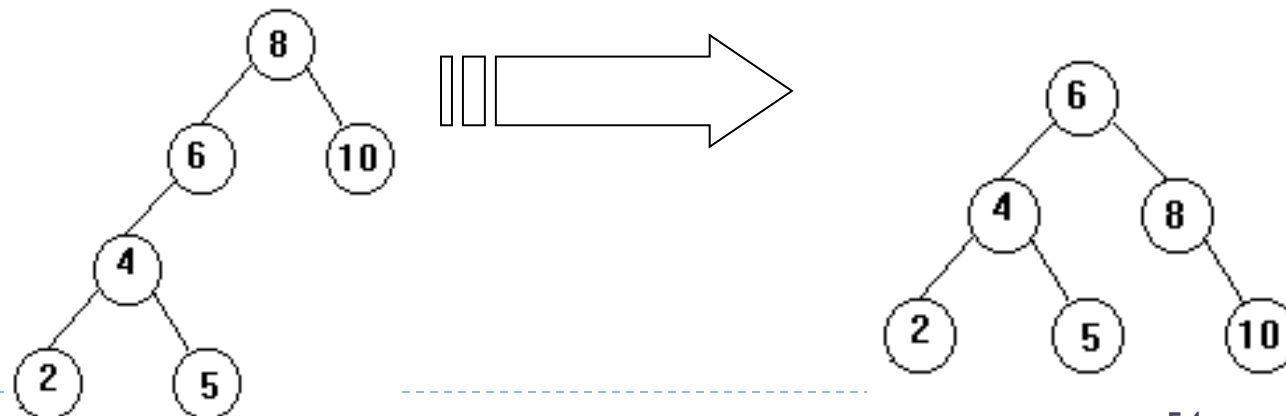
1. Roda-se o nó com $FB=1$ (-1) na direção apropriada
2. Depois roda-se o nó que tinha $FB=-2$ (2) na direção oposta

Inserção AVL - Rebalanceamento

- ▶ Exemplo: 1ª Rotação - Nó 4 rotacionado à esquerda



- ▶ Exemplo: 2ª Rotação - Nó 8 rotacionado à direita



Inserção AVL - Rebalanceamento

▶ Dicas:

- ▶ FB positivo (+) a rotação para à esquerda
- ▶ FB negativo (-) a rotação para à direita

- ▶ Para identificar quando uma rotação é simples ou dupla deve-se observar os sinais do FB:
 - ▶ Sinais iguais para pai e filho, a rotação é simples
 - ▶ Sinal diferentes para pai e filho, a rotação é dupla

Conclusões

- ▶ Balanceamento **busca minimizar** o número **médio de comparações** necessárias para localizar qualquer dado.
- ▶ **Operações de inserção e remoção** tendem a tornar as árvores **desbalanceadas**.
- ▶ Há um **custo extra de processamento**.
- ▶ Esse custo é compensado quando os dados armazenados precisam ser recuperados **muitas vezes**.

Implementação de AVLs

```
/* Definição da Estrutura de Dados */  
  
typedef struct no {  
    int info;  
    int bal; /* fator de balanceamento */  
    struct no *esq;  
    struct no *dir;  
} tNo;  
  
typedef tNo *tAvl; /* ponteiro para raiz da AVL */
```

Implementação de AVLs

```
//Rotina responsável pela rotação para a direita
void rot_dir (tAvl *p) {
    tAvl u;
    u = (*p)->esq;
    (*p )->esq = u->dir;
    u->dir = *p;
    *p  = u;
    /* precisa recalcular ainda o u->bal e p->bal*/
}
```

Implementação de AVLs

```
//Rotina responsável pela rotação para a esquerda
void rot_esq (tAvl *T) {
    tAvl u;
    u = (*T) -> dir;
    (*T) -> dir = u -> esq;
    u -> esq = *T;
    *T = u;
    /* precisa recalcular ainda o u->bal e p->bal. */
}
```

Universidade Federal da Paraíba

Centro de Informática

Departamento de Informática

Estrutura de Dados

Árvores

- ▶ Tiago Maritan
- ▶ tiago@ci.ufpb.br

