

# ÁRVORES



**INSTITUTO  
FEDERAL**  
Paraíba

Professor Msc Paulo de Tarso F. Júnior  
[paulodt@gmail.com](mailto:paulodt@gmail.com)

# Introdução

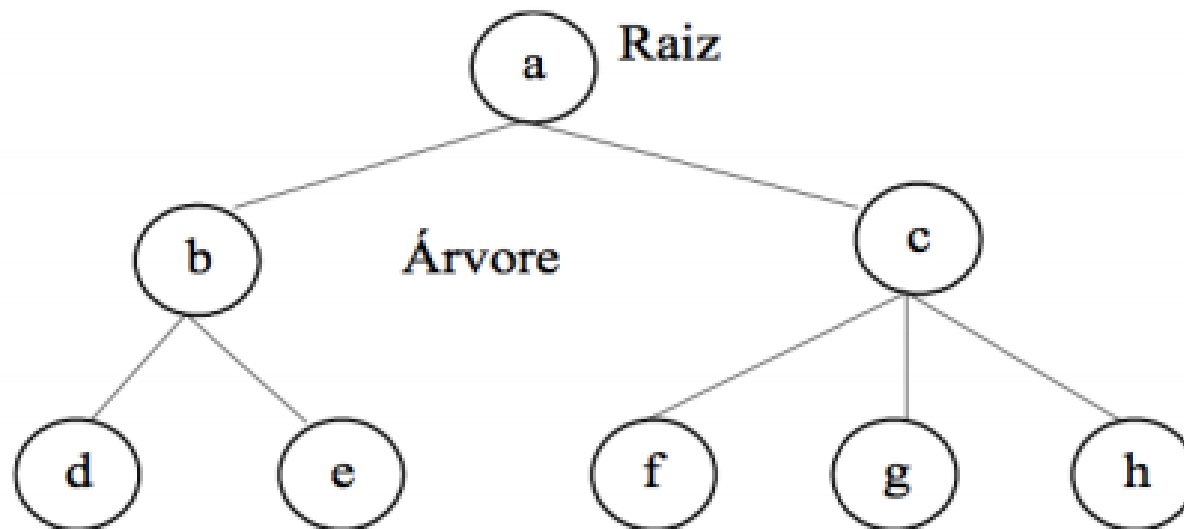
- ▶ São estruturas de dados (não-lineares) que caracterizam uma relação entre os dados
  - ▶ A relação existente entre os dados é uma relação de **hierarquia** ou de **composição** (um conjunto é subordinado a outro).

# Definição

- ▶ É um conjunto finito  $T$  de um ou mais nós, tais que:
  - ▶ Existe um nó principal chamado raiz (root);
  - ▶ Os demais nós formam  $n \geq 0$  conjuntos disjuntos  $T_1, T_2, \dots, T_n$ , onde cada um destes subconjuntos é uma árvore.
  - ▶ As árvores  $T_i$  ( $i \geq 1$  e  $i \leq n$ ) recebem a denominação de sub-árvores.

# Definição

## ► Representação Gráfica



# Definição

## ► Implementação

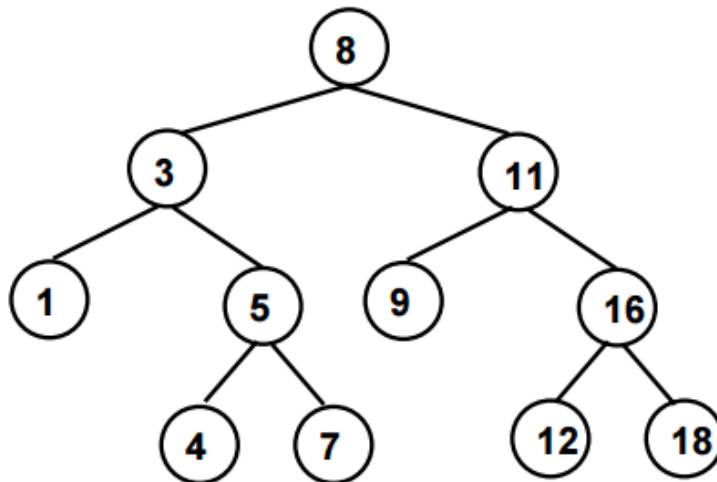
```
struct noArv {  
    int info;  
    struct noArv* esq;  
    struct noArv* dir;  
};  
  
typedef struct noArv NoArv;
```

# Árvores Binárias de Busca

- ▶ Árvore Binária de Busca  $T$  (ABB) ou Árvore Binária de Pesquisa (*Binary Search Trees - BSTs*) é tal que ou  $T = 0$  e a árvore é dita vazia ou seu nó raiz contém uma chave e:
  - ▶ Todas as chaves da sub-árvore esquerda são menores que a chave da raiz.
  - ▶ Todas as chaves da sub-árvore direita são maiores que a chave raiz.
  - ▶ As sub-árvores direita e esquerda são também Árvores Binárias de Busca.

# Árvores Binárias de Busca

- Medida de eficiência é dada pelo número de comparações necessárias para se localizar uma chave, ou descobrir que ela não existe.



# Árvores Binárias de Busca

- ▶ Numa lista linear com  $n$  chaves, temos que, no pior caso fará  $n$  comparações.
- ▶ O número de comparações cresce linearmente em função do número de chaves.
- ▶ Um percurso *em-ordem* nessa árvore resulta na sequência de valores em ordem crescente.



# Operações em Árvore Binária

- ▶ As operações básicas em uma Árvore Binária de Busca são:
  - ▶ Criação
  - ▶ Inserção
  - ▶ Busca
  - ▶ Remoção

# Criação

- Para iniciar uma árvore binária de busca, basta que o ponteiro para a raiz seja apontado para NULL

```
NoArv* abb_cria (void) {  
    return NULL;  
}
```

# Inserção

- ▶ Passos do algoritmo de inserção:
  - ▶ Procure um “local” para inserir o novo nó, começando a procura a partir do nó-raiz;
  - ▶ Para cada nó-raiz de uma sub-árvore, compare:
    - ▶ Se o novo nó possui um valor menor do que o valor nó raiz (vai para sub-árvore esquerda)
    - ▶ Se o valor é maior que o valor no nó-raiz (vai para sub-árvore direita);
  - ▶ Se uma referência (filho esquerdo/direito de um nó raiz) nula é atingida, coloque o novo nó como sendo filho do nó-raiz.


# Inserção

- ▶ Exemplo

- ▶ Para entender o algoritmo considere a inserção do conjunto de números, na sequência:

17, 99, 12, 1, 3, 100, 400

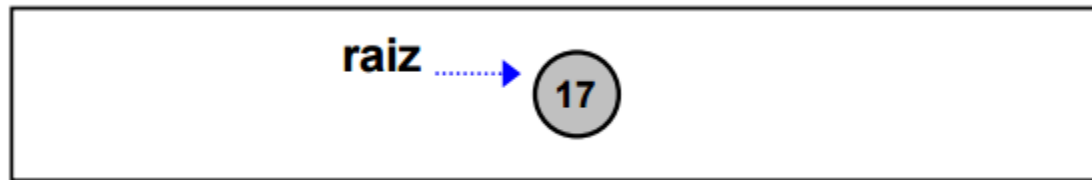
- ▶ No início a ABB está vazia!

raiz 

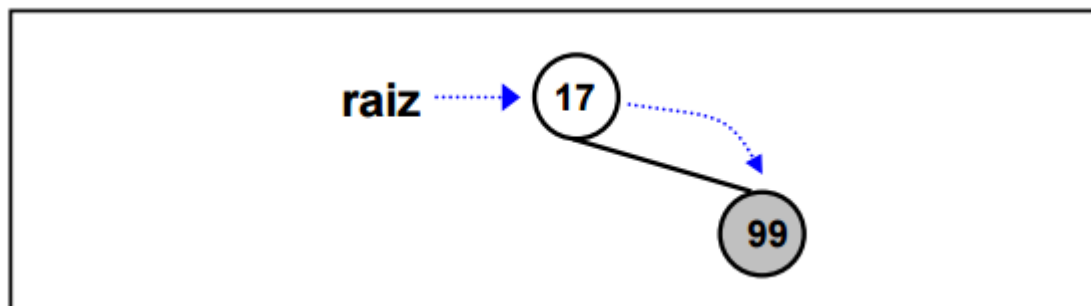
12

# Inserção

- O número 17 será inserido tornando-se o nó raiz:

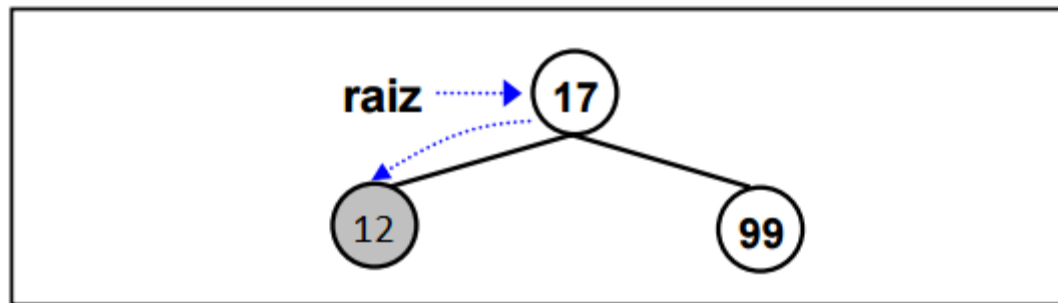


- A inserção do 99 inicia-se na raiz. Compara-se 99 com 17. Como  $99 > 17$ , 99 deve ser colocado na sub-árvore direita do nó contendo 17 (sub-árvore direita, inicialmente, nula);



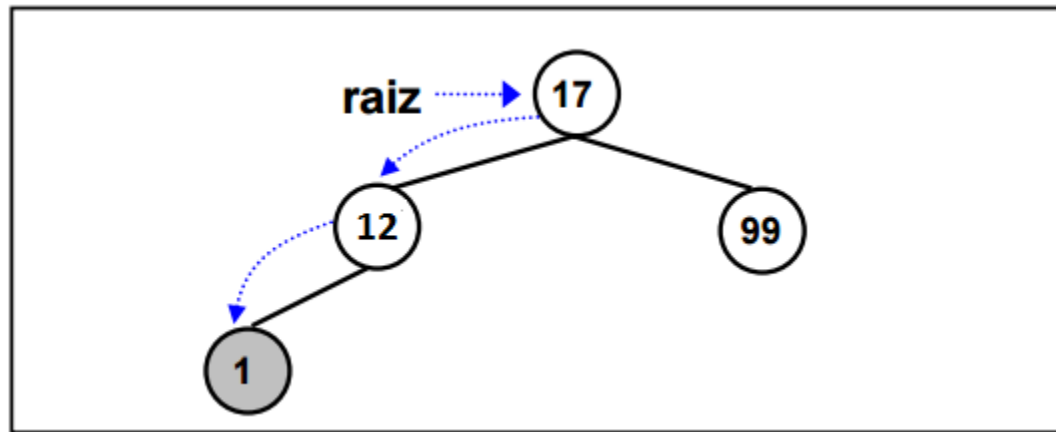
# Inserção

- ▶ A inserção do 12 inicia-se na raiz Compara-se 12 com 17:
  - ▶ Como  $12 < 17$ , 12 deve ser colocado na sub-árvore esquerda do nó contendo 17.
  - ▶ Já que o nó 17 não possui descendente esquerdo, 12 é inserido na árvore nessa posição.



# Inserção

- ▶ Para inserir o valor 1, repete-se o procedimento:
  - ▶  $1 < 17$ , então será inserido na sub-árvore esquerda.
  - ▶ Chegando nela, encontra-se o nó 12,  $1 < 12$ , então ele será inserido na sub-árvore esquerda de 12.

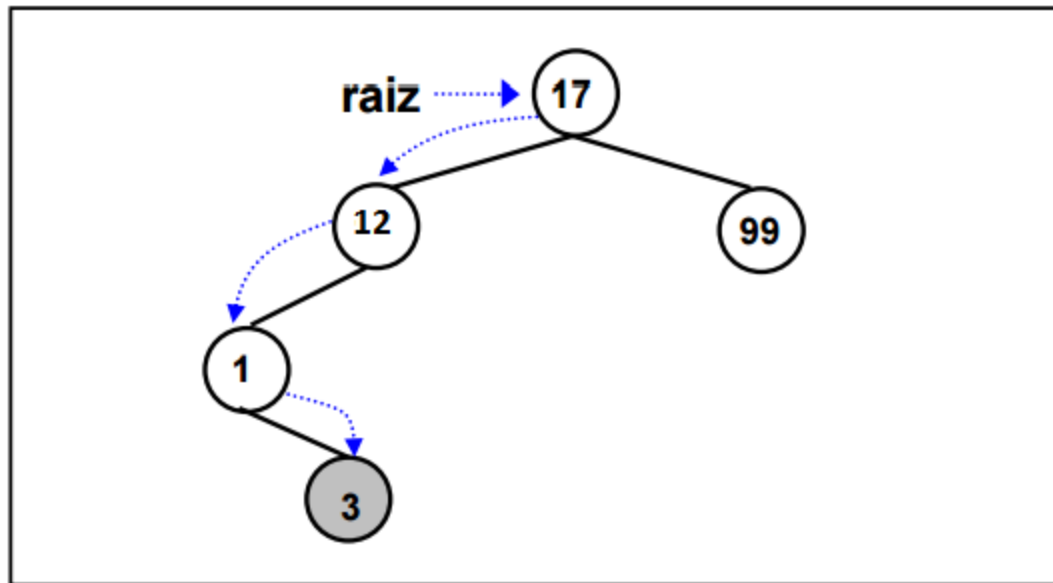


# Inserção

- ▶ Para inserir o valor 3, repete-se o procedimento:
  - ▶  $3 < 17$ , então ele será inserido na sub-árvore esquerda.
  - ▶ Chegando nela, encontra-se o nó 12,  $3 < 12$ .
  - ▶ Chegando à sub-árvore esquerda encontra-se o nó 1,  $3 > 1$ , então ele será inserido na sub-árvore esquerda de 12.

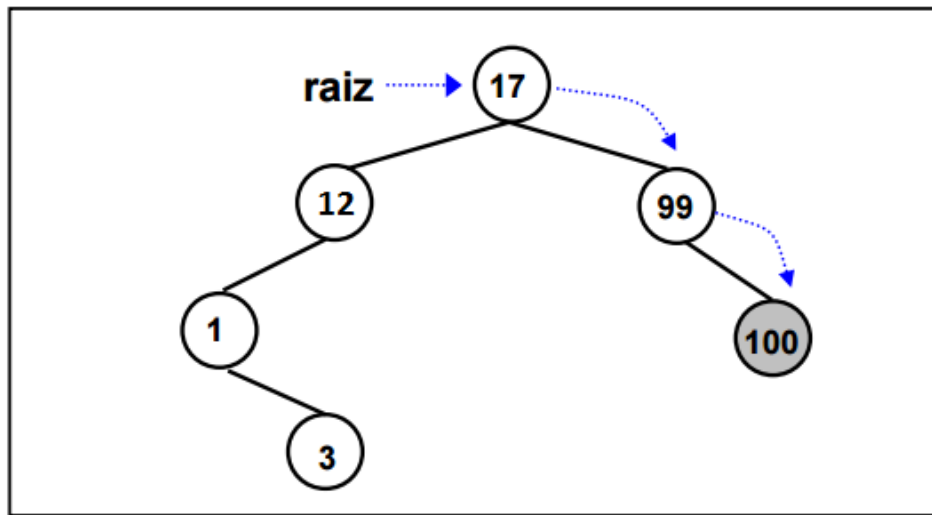


# Inserção



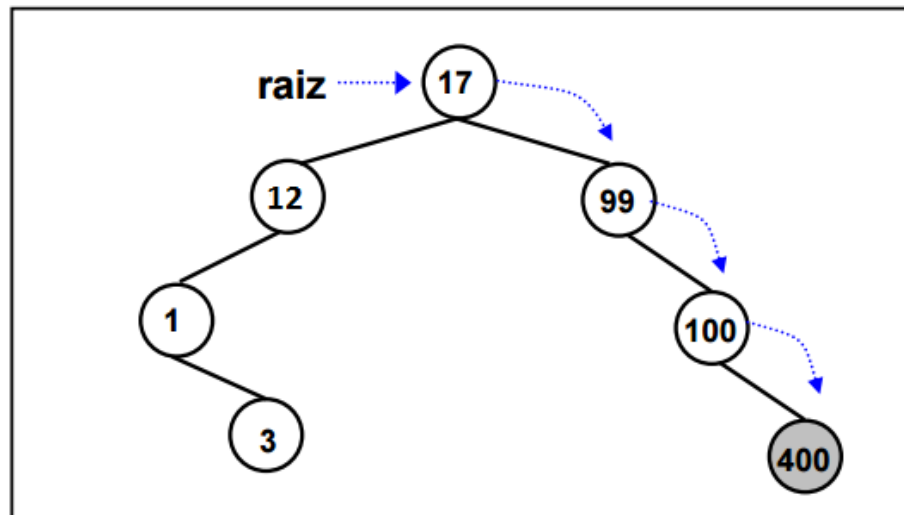
# Inserção

- ▶ Repete-se o procedimento para inserir o elemento 100:
  - ▶  $100 > 17$  (vai para a direita)
  - ▶  $100 > 99$  (vai para a direita);



# Inserção

- ▶ Repete-se o procedimento para inserir o elemento 400:
  - ▶  $400 > 17$  (vai para a direita)
  - ▶  $400 > 99$  (vai para a direita)
  - ▶  $400 > 100$  (vai para a direita)



# Inserção - Implementação

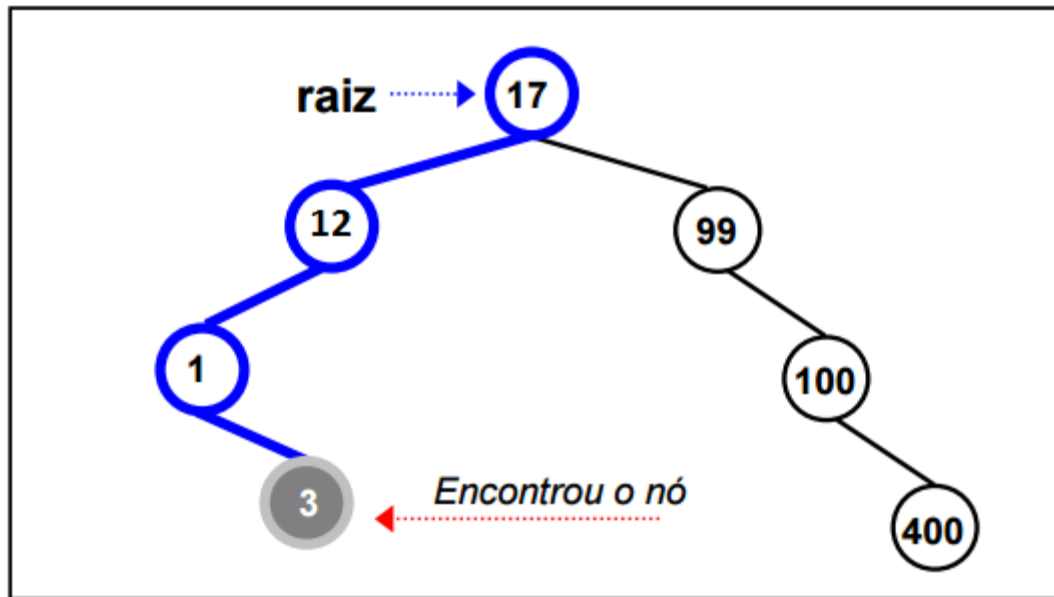
```
NoArv* abb_inserere (NoArv* a, int v) {  
    if (a==NULL) {  
        a = (NoArv*)malloc(sizeof(NoArv));  
        a->info = v;  
        a->esq = a->dir = NULL;  
    } else if (v < a->info){  
        a->esq = abb_inserere(a->esq,v);  
    } else { /* v >= a->info */  
        a->dir = abb_inserere(a->dir,v);  
    }  
    return a;  
}
```

# Busca

- ▶ Comece a busca a partir do nó-raiz;
- ▶ Para cada nó-raiz de uma sub-árvore compare:
  - ▶ Se o valor procurado é menor que o valor no nó-raiz (continua pela sub-árvore esquerda)
  - ▶ Se o valor é maior que o valor no nó-raiz (sub-árvore direita);
  - ▶ Caso o nó contendo o valor pesquisado seja encontrado, retorne o nó;
  - ▶ Caso contrário retorne nulo.

# Busca

- Por exemplo, para encontrar a chave 3, o caminho de busca é representado a seguir:



# Busca - Implementação

```
NoArv* abb_busca (NoArv* r, int v) {  
    if (r == NULL){  
        return NULL;  
    } else if (r->info > v) {  
        return abb_busca (r->esq, v);  
    } else if (r->info < v) {  
        return abb_busca (r->dir, v);  
    } else {  
        return r;  
    }  
}
```

# Remoção

Para a remoção de um nó em uma árvore binária, devem ser considerados três casos:

► ***Caso 1: o nó é folha***

- O nó pode ser retirado sem problema;

► ***Caso 2: o nó possui uma sub-árvore (esq./dir.)***

- O nó-raiz da sub-árvore (esq./dir.) “ocupa” o lugar do nó retirado;

► ***Caso 3: o nó possui duas sub-árvores***

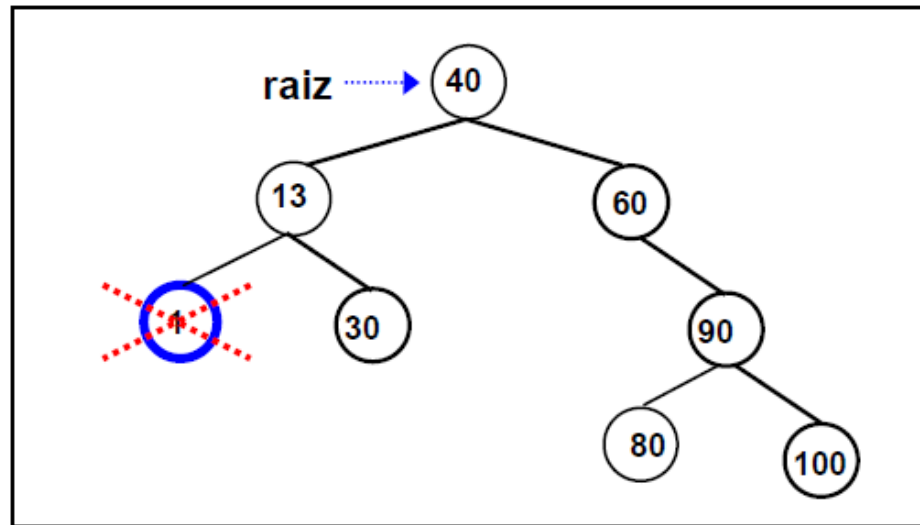
- O nó contendo o menor valor da sub-árvore direita pode “ocupar” o lugar; ou o maior valor da sub-árvore esquerda pode “ocupar” o lugar



# Remoção

## ► *Caso 1: Remoção do nó 1*

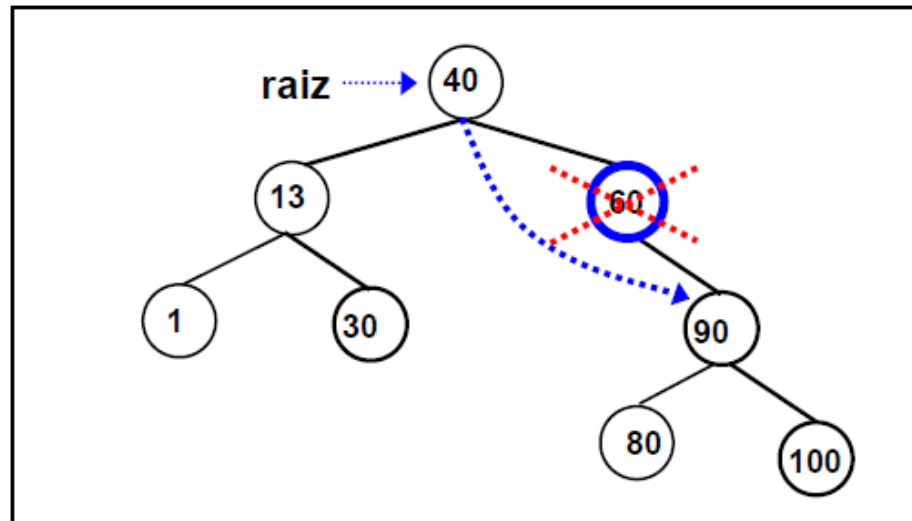
- Ele pode ser removido sem problema, pois não requer ajustes posteriores.
- Os nós **30**, **80** e **100** também podem ser removidos sem problemas!



# Remoção

## ► *Caso 2: Remoção do nó 60*

- Como ele possui apenas a sub-árvore direita, o nó contendo o valor **90** pode “ocupar” o lugar do nó removido.

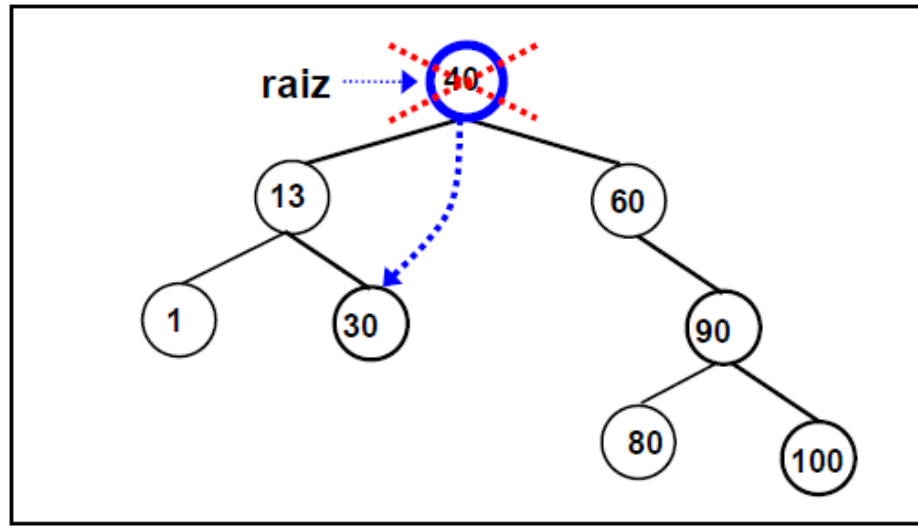


# Remoção

## ► *Caso 3: Remoção do nó 40*

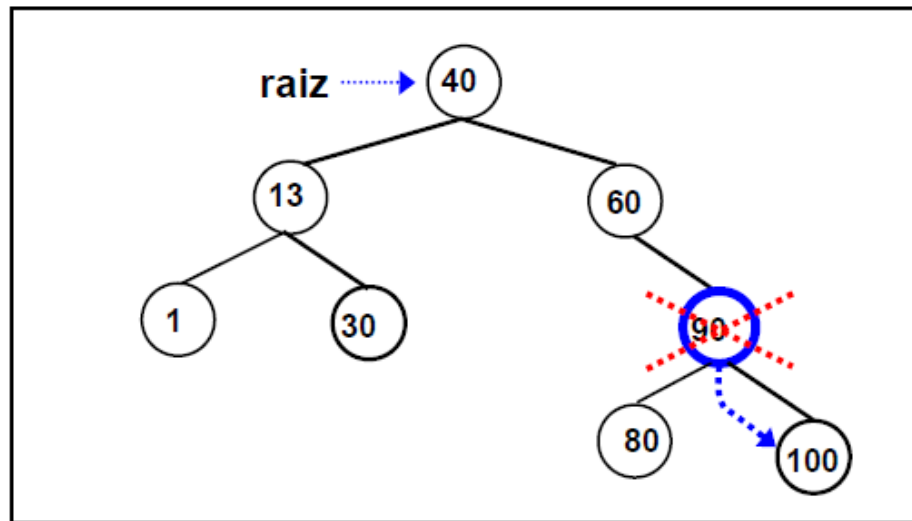
### ► Neste caso, existem *2 opções*:

- O nó com valor **30** pode “ocupar” o lugar do nó-raiz, ou
- O nó com valor **60** pode “ocupar” o lugar do nó-raiz.



# Remoção

- ▶ Este caso também se aplica ao nó **90**:
  - ▶ O nó com valor **80** pode “ocupar” o lugar do nó-raiz, ou
  - ▶ O nó com valor **100** pode “ocupar” o lugar do nó-raiz.



# Remoção

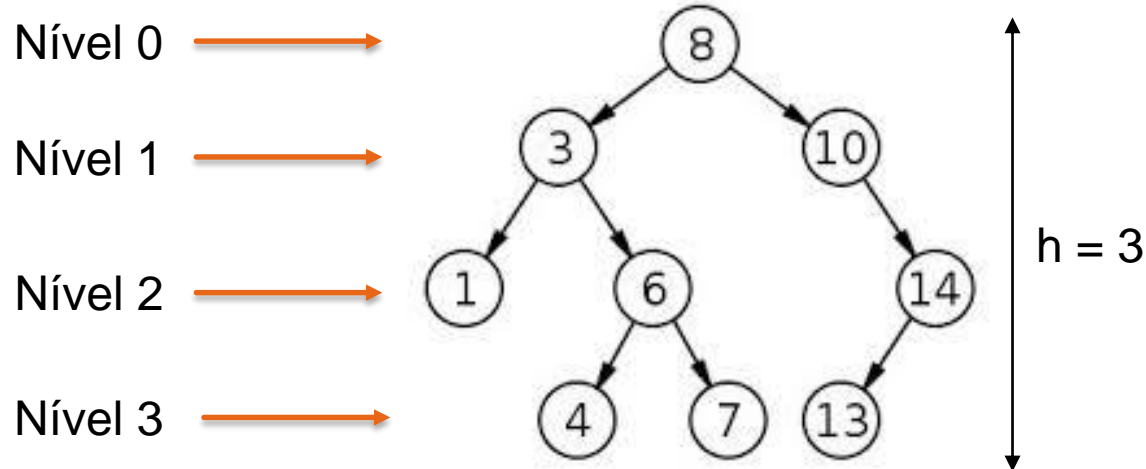
- **Importante:** Uma vez definida a regra de escolha do nó substituto, ela deve ser a mesma para todas as operações de remoção!

# Altura

- ▶ A altura de um nó  $X$  em uma árvore binária é a distância entre  $X$  e um descendente mais afastado.
- ▶ Mais precisamente, a altura de  $X$  é o número de passos do mais longo caminho que leva de  $X$  até uma folha.

# Altura

- ▶ A altura de uma árvore é a altura da raiz da árvore.
- ▶ Uma árvore com um único nó tem altura 0.
- ▶ A árvore da figura tem altura 3.



# Profundidade

- ▶ A *profundidade* (=depth) de um nó em uma árvore binária com raiz  $r$  é a distância de  $r$  a  $s$ .
- ▶ Mais precisamente, a profundidade de  $s$  é o comprimento do (único) caminho que vai de  $r$  até  $s$ .
- ▶ Por exemplo, a profundidade de  $r$  é 0 e a profundidade de  $r \rightarrow \text{esq}$  é 1.



# Custo de Busca em ABB

- ▶ **Pior caso:** número de passos é determinado pela altura da árvore.
  - ▶ A altura da Árvore de Busca Binária depende da sequência de inserção das chaves.
  - ▶ Considere, por exemplo, o que acontece se uma sequência ordenada de chaves é inserida.
  - ▶ Seria possível gerar uma árvore balanceada com essa mesma sequência, se ela fosse conhecida *a priori*.
  - ▶ A busca pode ser considerada eficiente se a árvore estiver razoavelmente balanceada.

# Custo de Busca em ABB

- ▶ Muitas operações em árvores binárias envolvem o percurso de todas as suas sub-árvores.
  - ▶ Alguma ação de tratamento é executada em cada nó.
- ▶ É comum percorrer uma árvore em uma das seguintes ordens:
  - ▶ **Pré-Ordem:** tratar **RAIZ**, percorrer a árvore da **ESQ**, percorrer a árvore da **DIR**.
  - ▶ **Em-Ordem:** Percorrer a árvore da **ESQ**, tratar a **RAIZ**, percorrer a árvore da **DIR**.
  - ▶ **Pós-Ordem:** Percorrer a árvore da **ESQ**, percorrer a árvore da **DIR** e tratar a **RAIZ**.

# Dúvidas



# Referências

- ▶ Aaron M. Tenenbaum, Yedidyah Langsam, Moshe J. Augenstein ***Estruturas de Dados Usando C***. Pearson (26 de junho de 1995)
- ▶ Backes A. ***Estrutura de Dados Descomplicada em Linguagem C***. Elsevier; Edição: 1ª (9 de agosto de 2016)
- ▶ Programar em C/Árvores Binárias. Disponível em: [https://pt.wikibooks.org/wiki/Programar\\_em\\_C/%C3%81rvores\\_bin%C3%A1rias#Arvore\\_bin.C3.A1ria](https://pt.wikibooks.org/wiki/Programar_em_C/%C3%81rvores_bin%C3%A1rias#Arvore_bin.C3.A1ria). Acesso em: 03/09/2017.
- ▶ Árvores Binárias. Disponível em: <https://www.ime.usp.br/~pf/algoritmos/aulas/bint.html>. Acesso em: 05/09/2017.