



# Estruturas de Dados

## Árvores

Prof. Thiago Caproni Tavares <sup>1</sup>    Paulo Muniz de Ávila <sup>2</sup>

<sup>1</sup>[thiago.tavares@ifsuldeminas.edu.br](mailto:thiago.tavares@ifsuldeminas.edu.br)

<sup>2</sup>[paulo.avila@ifsuldeminas.edu.br](mailto:paulo.avila@ifsuldeminas.edu.br)

## 1 Introdução

## 2 Árvores

### 3 Árvores Binárias

- Operações em Árvores Binárias

### 4 Aplicações de Árvores Binárias

- Árvore Binária para Remoção de Elementos duplicados
- Árvore Binária para Ordenação de Dados
- Algoritmo HUFFMAN

## 5 Árvore AVL

## 1 Introdução

## 2 Árvores

### 3 Árvores Binárias

- Operações em Árvores Binárias

### 4 Aplicações de Árvores Binárias

- Árvore Binária para Remoção de Elementos duplicados
- Árvore Binária para Ordenação de Dados
- Algoritmo HUFFMAN

## 5 Árvore AVL

*"As árvores da computação têm a curiosa tendência de crescer para baixo..."*

## 1 Introdução

## 2 Árvores

### 3 Árvores Binárias

- Operações em Árvores Binárias

### 4 Aplicações de Árvores Binárias

- Árvore Binária para Remoção de Elementos duplicados
- Árvore Binária para Ordenação de Dados
- Algoritmo HUFFMAN

### 5 Árvore AVL

- Uma **árvore** é um conjunto finito **T** de um ou mais *nós*, tais que:
  - ① Existe um nó denominado **raiz** da árvore;
  - ② Os demais nós formam conjuntos disjuntos  $S_1, S_2, \dots, S_n$ , onde cada um destes conjuntos é uma árvore. As árvores  $S_i (1 \leq i \leq n)$  recebem a denominação de **subárvores**
- As árvores são utilizadas para representação de objetos que possuem relações hierárquicas entre si, como por exemplo: *estruturas empresariais, expressões aritméticas, realização de buscas em grafos.*

## 1 Introdução

## 2 Árvores

## 3 Árvores Binárias

- Operações em Árvores Binárias

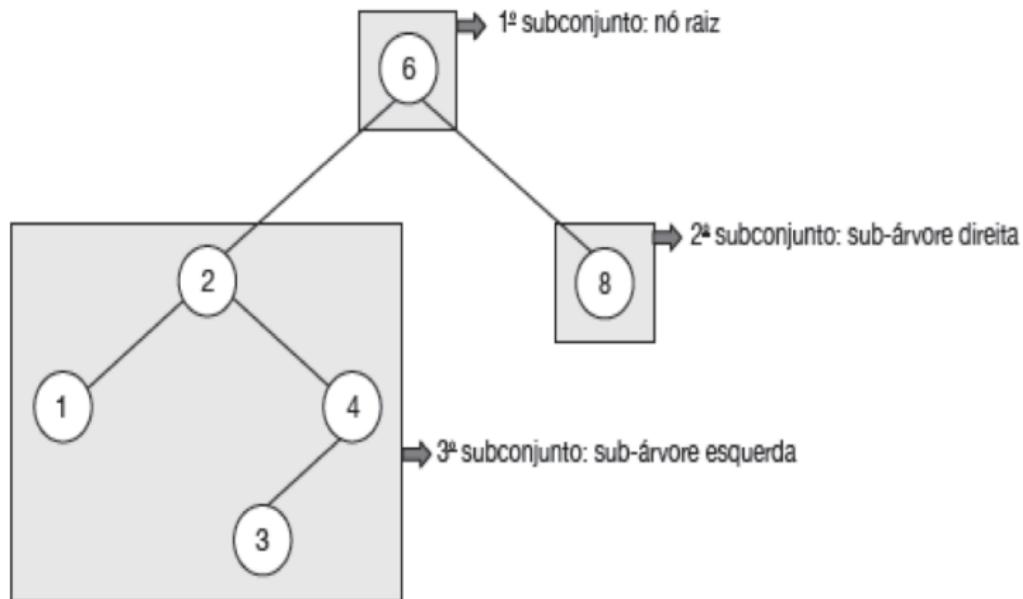
## 4 Aplicações de Árvores Binárias

- Árvore Binária para Remoção de Elementos duplicados
- Árvore Binária para Ordenação de Dados
- Algoritmo HUFFMAN

## 5 Árvore AVL

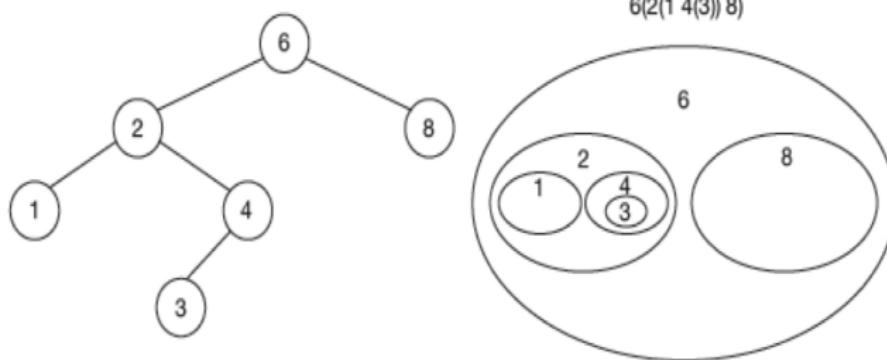
- Uma **árvore binária** é um conjunto finito de elementos, onde cada elemento é denominado **nó** e o primeiro é conhecido como **raiz** da árvore. Esse conjunto pode estar vazio ou ser particionado em três subconjuntos distintos, sendo eles:
  - ① subconjunto (nó raiz);
  - ② sub-árvore direita;
  - ③ sub-árvore esquerda.
- A figura no próximo slide ilustra essa partição.

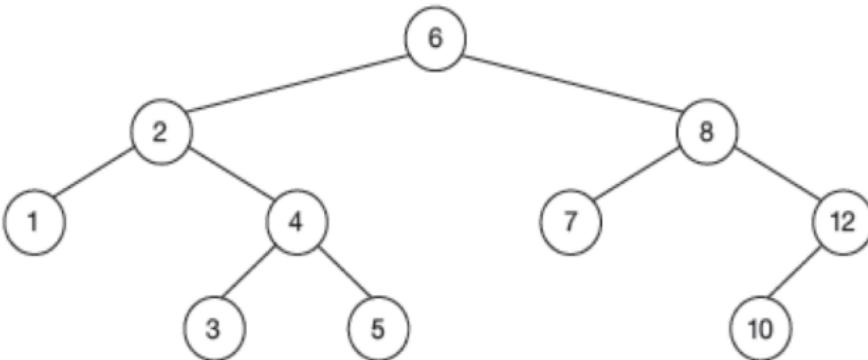
# Árvores binária



# Árvores binária

- As árvores binárias podem ser ilustradas de três formas:

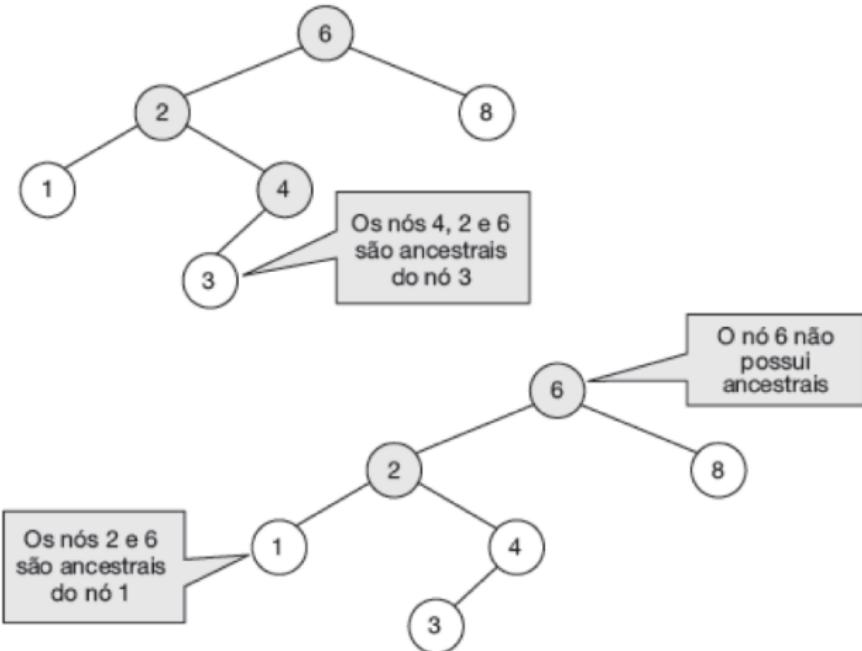




- Se o nó 6 é a raíz da árvore binária e 2 é sua subárvore esquerda, então diz-se que 6 é **pai** de 2 e que 2 é o **filho direto** de 6.
- Um nó sem filhos (como 1,3,5,7 e 10) é chamado **folha**.
- Dois nós são **irmãos** se forem filhos esquerdos ou direitos do mesmo pai. Por exemplo: os nós 3 e 5.

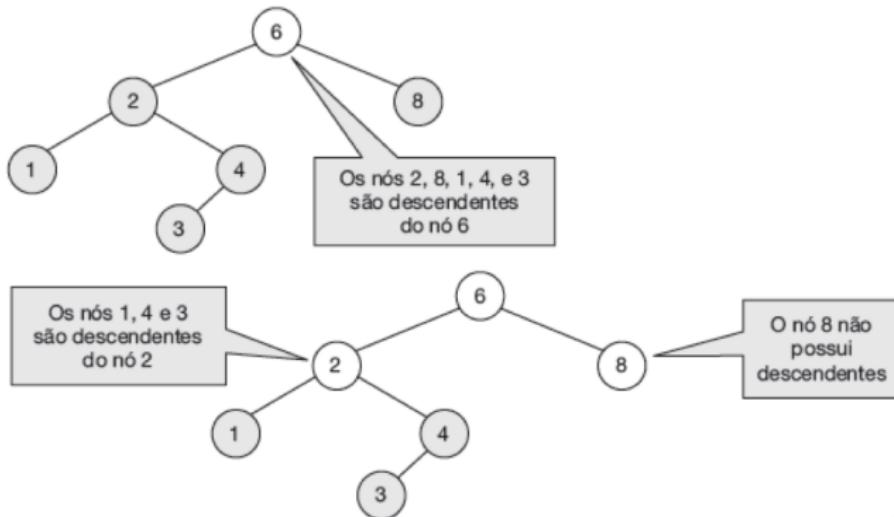
# Árvores binária

- Nós ancestrais: estão acima de um nó e têm ligação direta ou indireta.



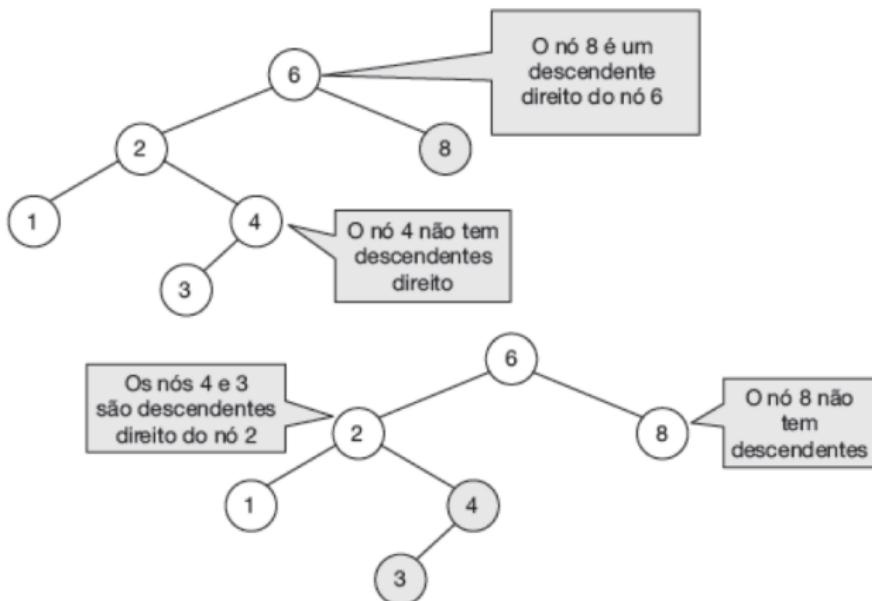
# Árvores binária

- Nós descendentes: estão abaixo de um nó e possuem ligação direta ou indireta.



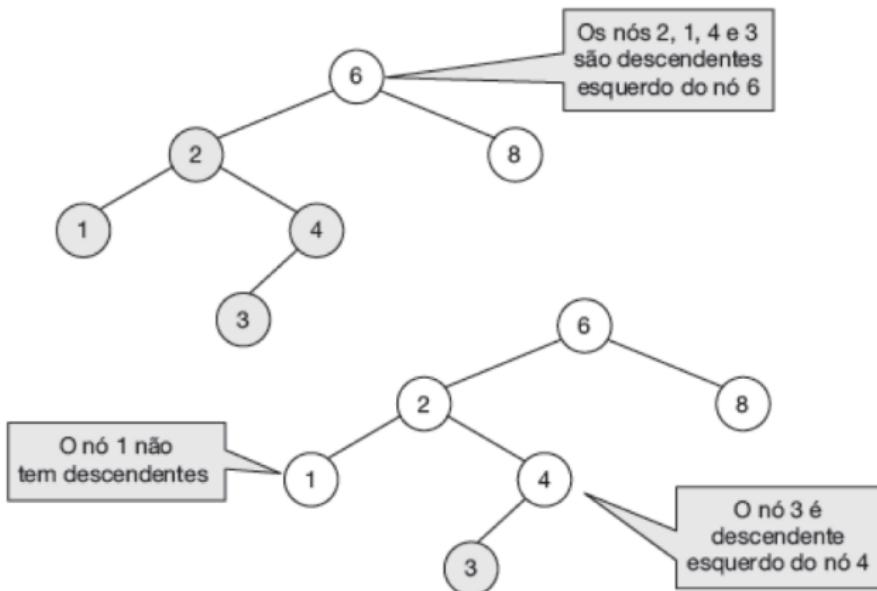
# Árvores binária

- Nós descendentes direito: estão abaixo de um nó, possuem ligação direta ou indireta e fazem parte da sub-árvore direita.



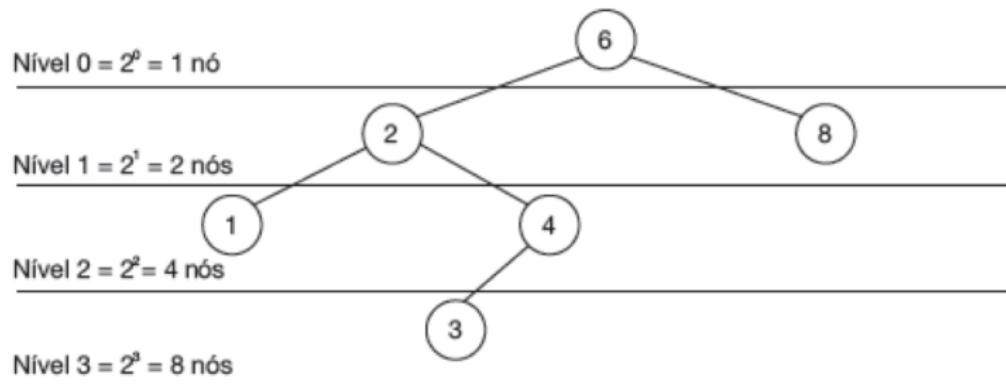
# Árvores binária

- Nós descendentes esquerdo: estão abaixo de um nó, possuem ligação direta ou indireta e fazem parte da sub-árvore esquerda.



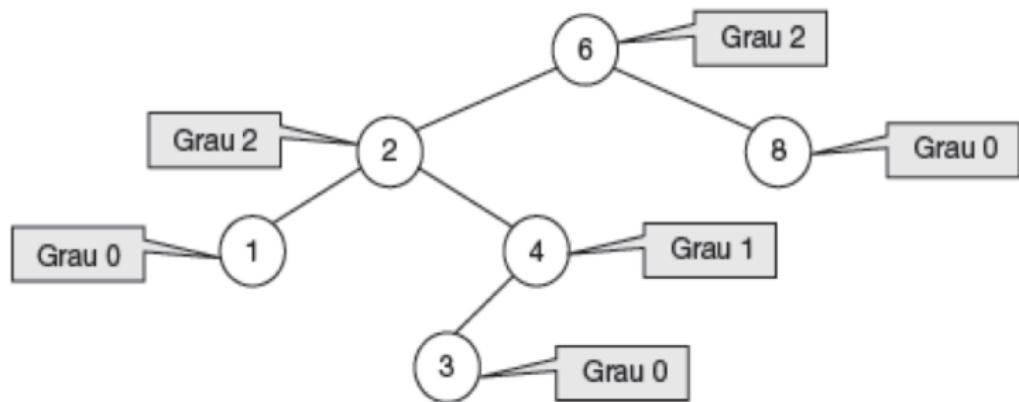
# Árvores binária

- O **nível** de um nó em uma árvore binária é definido como segue: a raiz da árvore tem nível 0, e o nível de qualquer outro nó da árvore é um nível a mais que o nível de seu pai.
- A **profundidade** ou **altura** de uma árvore binária significa o nível máximo de qualquer folha na árvore. Isso equivale ao tamanho do percurso mais distante da raiz até qualquer folha.



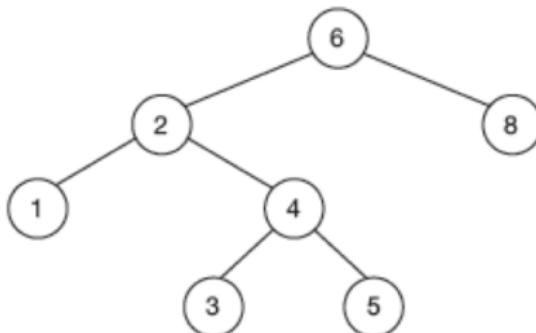
# Árvores binária - Grau

- O grau de um nó representa o seu número de sub-árvores.



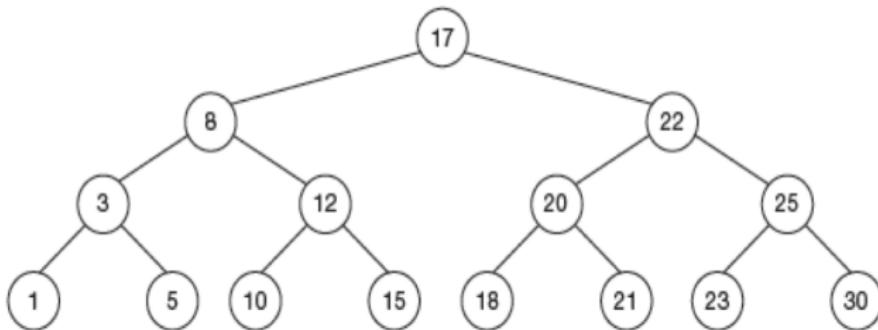
- Todos os nós de uma sub-árvore direita são maiores que o nó raiz.
- Todos os nós de uma sub-árvore esquerda são menores que o nó raiz.
- Cada sub-árvore é também uma árvore binária.
- Na árvore binária, o grau máximo de um nó é 2.
- O grau de uma árvore é igual ao máximo dos graus de todos os seus nós.
- Uma árvore binária tem grau máximo igual a 2.

- Se todo nó que não é folha em uma árvore binária tiver subárvores esquerda e direita **não-vazias**, a árvore será considerada uma **árvore estritamente binária**.
- Uma árvore estritamente binária com  $n$  folhas contém sempre  $2n - 1$  nós.



Quantidade de nós  
folha = 4.  
Os nós folha são:  
1, 3, 5 e 8.  
Número de nós desta  
árvore estritamente  
binária =  $2.n - 1$ , onde  
 $n$  é o número de folhas  
 $2.4 - 1 = 7$  nós

- Uma **árvore cheia** de profundidade  $d$  é a árvore estritamente binária em que todas as folhas estejam no nível  $d$ .



- Se uma árvore cheia contiver  $m$  nós no nível  $l$ , ela conterá no máximo  $2m$  nós no nível  $l + 1$ .
- Como uma árvore binária pode conter no máximo um nó no nível 0 (raiz), ela poderá conter no máximo  $2^l$  nós no nível  $l$ .
- O número total de nós  $tn$  em uma árvore cheia de profundidade  $d$ , é igual à soma do número de nós em cada nível entre 0 e  $d$ . Sendo assim:

$$tn = 2^0 + 2^1 + 2^2 + \dots + 2^d = \sum_{j=0}^d 2^j$$

- Por indução, pode-se demonstrar que essa soma equivale a  $tn = 2^{d+1} - 1$ .
- Podemos calcular a profundidade,  $d$ , de uma árvore cheia utilizando:

$$d = \log_2(tn + 1) - 1$$

## 1 Introdução

## 2 Árvores

### 3 Árvores Binárias

- Operações em Árvores Binárias

### 4 Aplicações de Árvores Binárias

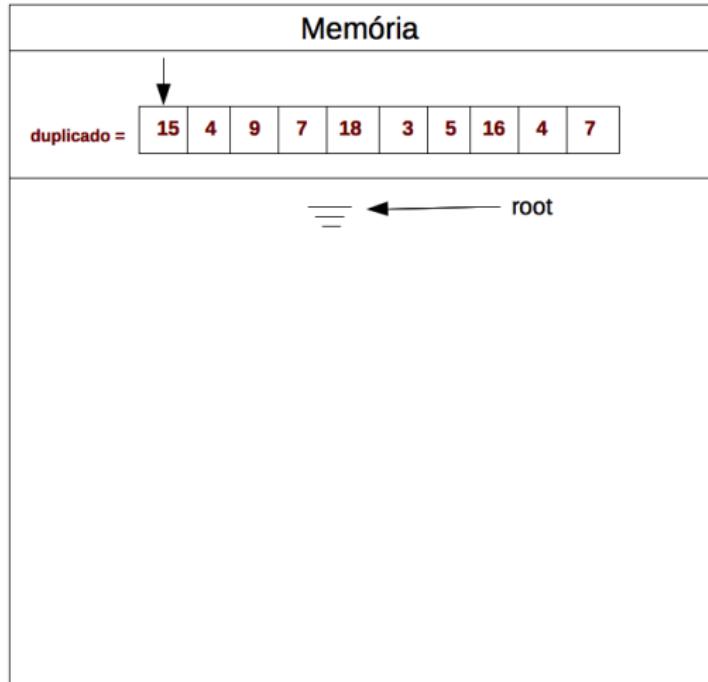
- Árvore Binária para Remoção de Elementos duplicados
- Árvore Binária para Ordenação de Dados
- Algoritmo HUFFMAN

### 5 Árvore AVL

- A inserção de um nó em uma **Árvore Binária** segue os seguintes passos:
  - ① Inicia-se a procura pela raiz da árvore binária. Se o elemento a ser inserido tiver o mesmo valor do elemento da raiz, ele existe na árvore e está localizado na raiz. Se o elemento a ser inserido tiver valor menor que o elemento da raiz , continua a procura recursivamente pela subárvore da esquerda, se o elemento a ser inserido tiver um valor maior que o valor da raiz, continua a procura recursivamente pelas subárvores da direita.
  - ② Se o elemento a ser inserido já existir na árvore, nessa implementação, nenhum novo nó será alocado e nenhuma alteração ocorrerá na árvore binária.
- Nos próximos slides iremos simular a inserção de elementos inteiros de um vetor em uma árvore binária. Observe o que ocorre quando tentamos inserir um elemento que já existe na árvore.
- Pense em como poderíamos alterar o algoritmo para permitir a inserção de elementos duplicados !

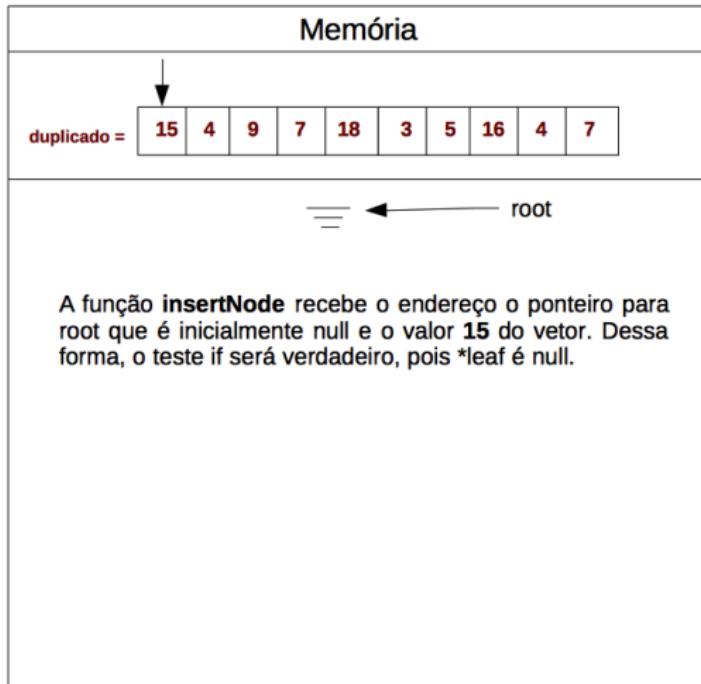
# Inserir Nós na Árvore Binária

```
void makeTree(node **root){  
    *root = NULL;  
}  
  
***** inserir *****/  
void insertNode(node **leaf, int key_value){  
    if(*leaf == NULL){  
        *leaf = (node *) malloc(sizeof(node));  
        (*leaf)->esq = NULL;  
        (*leaf)->dir = NULL;  
        (*leaf)->key = key_value;  
    }else{  
        if(key_value < (*leaf)->key)  
            insertNode(&(*leaf)->esq, key_value);  
        if(key_value > (*leaf)->key)  
            insertNode(&(*leaf)->dir, key_value);  
    }  
}  
  
typedef struct No{  
    int key;  
    struct No *esq;  
    struct No *dir;  
} node;
```



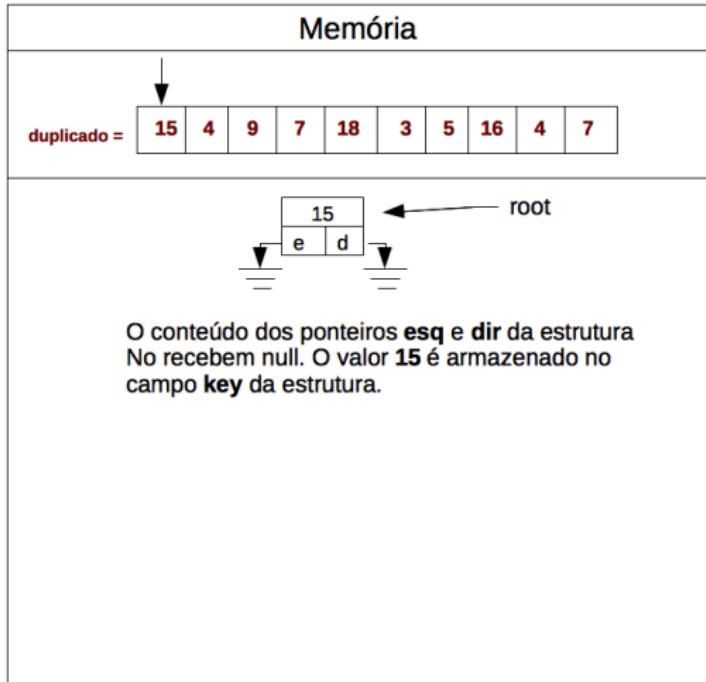
# Inserir Nós na Árvore Binária

```
void makeTree(node **root){  
    *root = NULL;  
}  
  
***** inserir *****  
void insertNode(node **leaf, int key_value){  
    if(*leaf == NULL){  
        *leaf = (node *) malloc(sizeof(node));  
        (*leaf)->esq = NULL;  
        (*leaf)->dir = NULL;  
        (*leaf)->key = key_value;  
    }else{  
        if(key_value < (*leaf)->key)  
            insertNode(&(*leaf)->esq, key_value);  
        if(key_value > (*leaf)->key)  
            insertNode(&(*leaf)->dir, key_value);  
    }  
  
typedef struct No{  
    int key;  
    struct No *esq;  
    struct No *dir;  
} node;
```



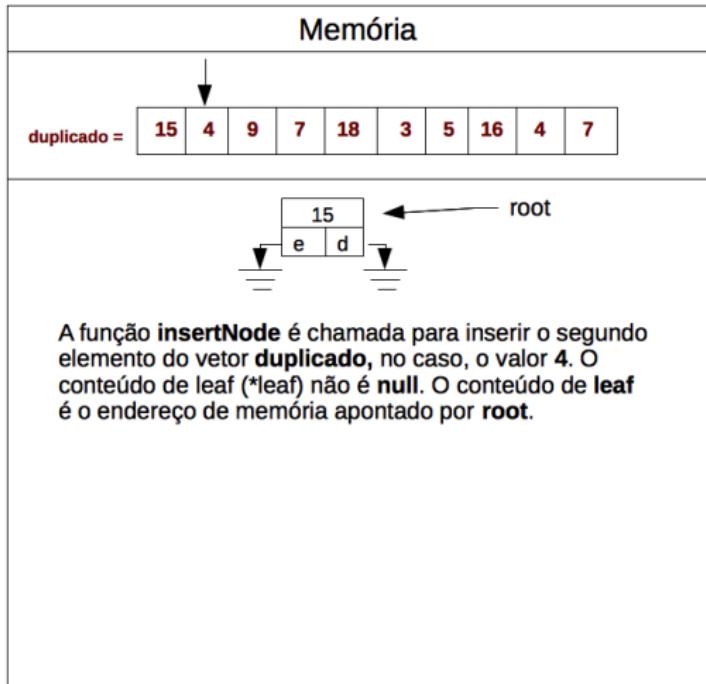
# Inserir Nós na Árvore Binária

```
void makeTree(node **root){  
    *root = NULL;  
}  
  
***** inserir *****/  
void insertNode(node **leaf, int key_value){  
    if(*leaf == NULL){  
        *leaf = (node *) malloc(sizeof(node));  
        (*leaf)->esq = NULL;  
        (*leaf)->dir = NULL;  
        (*leaf)->key = key_value;  
    }  
    else{  
        if(key_value < (*leaf)->key)  
            insertNode(&(*leaf)->esq, key_value);  
        if(key_value > (*leaf)->key)  
            insertNode(&(*leaf)->dir, key_value);  
    }  
  
typedef struct No{  
    int key;  
    struct No *esq;  
    struct No *dir;  
} node;
```



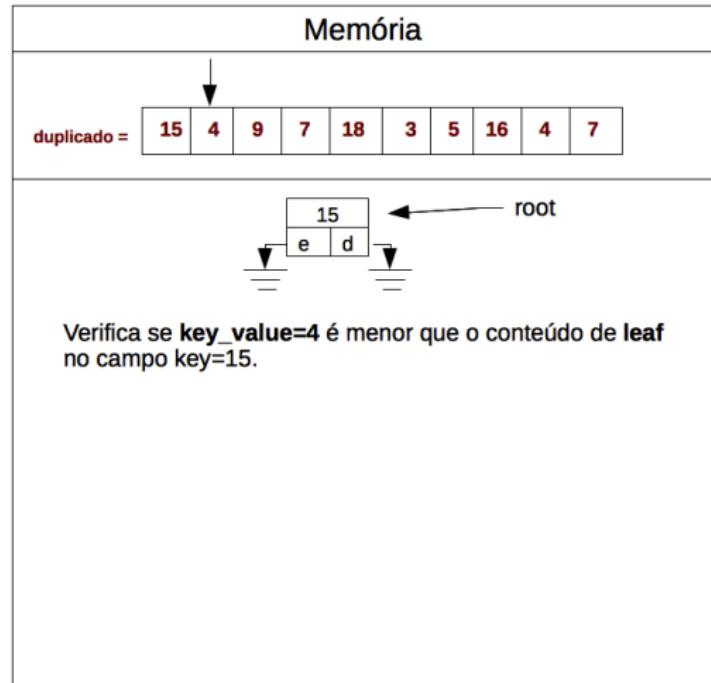
# Inserir Nós na Árvore Binária

```
void makeTree(node **root){  
    *root = NULL;  
}  
  
***** inserir *****/  
void insertNode(node **leaf, int key_value){  
    if(*leaf == NULL){  
        *leaf = (node *) malloc(sizeof(node));  
        (*leaf)->esq = NULL;  
        (*leaf)->dir = NULL;  
        (*leaf)->key = key_value;  
    }else{  
        if(key_value < (*leaf)->key)  
            insertNode(&(*leaf)->esq, key_value);  
        if(key_value > (*leaf)->key)  
            insertNode(&(*leaf)->dir, key_value);  
    }  
  
typedef struct No{  
    int key;  
    struct No *esq;  
    struct No *dir;  
} node;
```



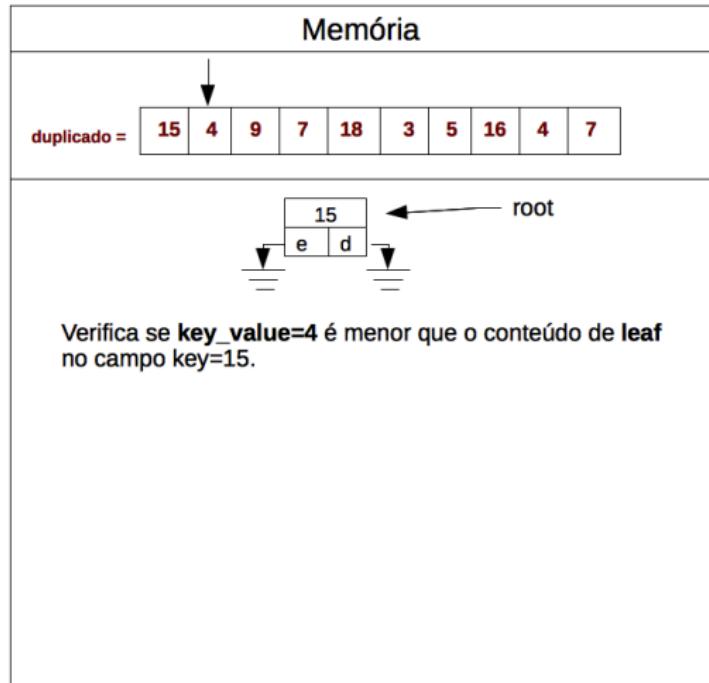
# Inserir Nós na Árvore Binária

```
void makeTree(node **root){  
    *root = NULL;  
}  
  
***** inserir *****/  
void insertNode(node **leaf, int key_value){  
    if(*leaf == NULL){  
        *leaf = (node *) malloc(sizeof(node));  
        (*leaf)->esq = NULL;  
        (*leaf)->dir = NULL;  
        (*leaf)->key = key_value;  
    }else{  
        if(key_value < (*leaf)->key)  
            insertNode(&(*leaf)->esq, key_value);  
        if(key_value > (*leaf)->key)  
            insertNode(&(*leaf)->dir, key_value);  
    }  
  
typedef struct No{  
    int key;  
    struct No *esq;  
    struct No *dir;  
} node;
```



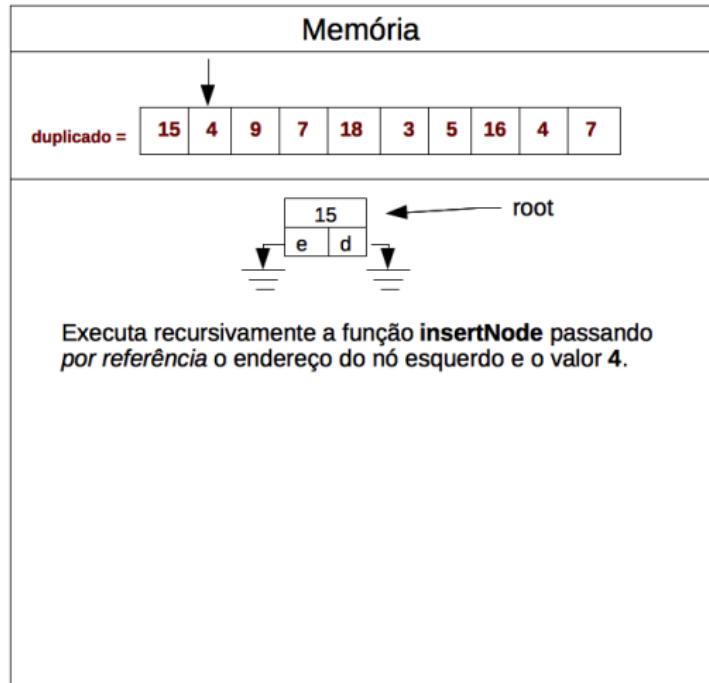
# Inserir Nós na Árvore Binária

```
void makeTree(node **root){  
    *root = NULL;  
}  
  
***** inserir *****/  
void insertNode(node **leaf, int key_value){  
    if(*leaf == NULL){  
        *leaf = (node *) malloc(sizeof(node));  
        (*leaf)->esq = NULL;  
        (*leaf)->dir = NULL;  
        (*leaf)->key = key_value;  
    }else{  
        if(key_value < (*leaf)->key)  
            insertNode(&(*leaf)->esq, key_value);  
        if(key_value > (*leaf)->key)  
            insertNode(&(*leaf)->dir, key_value);  
    }  
  
typedef struct No{  
    int key;  
    struct No *esq;  
    struct No *dir;  
} node;
```



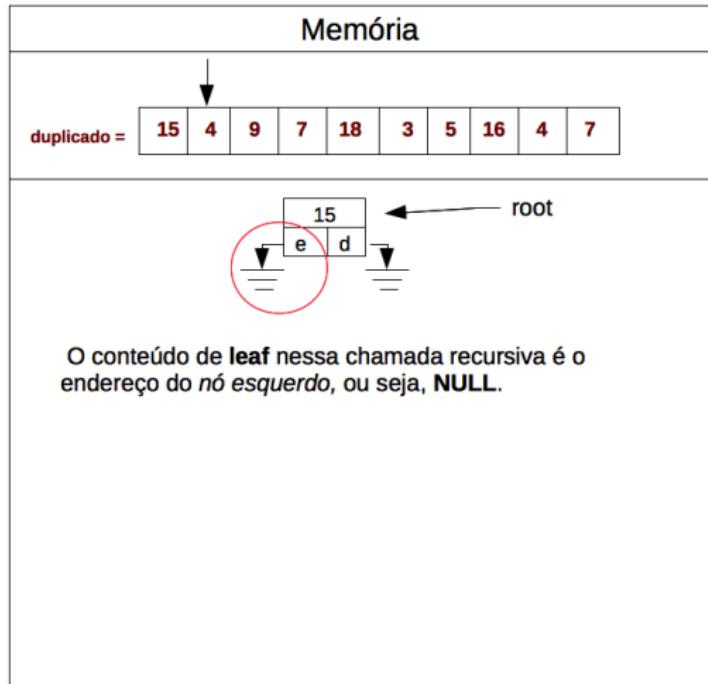
# Inserir Nós na Árvore Binária

```
void makeTree(node **root){  
    *root = NULL;  
}  
  
***** inserir *****/  
void insertNode(node **leaf, int key_value){  
    if(*leaf == NULL){  
        *leaf = (node *) malloc(sizeof(node));  
        (*leaf)->esq = NULL;  
        (*leaf)->dir = NULL;  
        (*leaf)->key = key_value;  
    }else{  
        if(key_value < (*leaf)->key)  
            insertNode(&(*leaf)->esq, key_value);  
        if(key_value > (*leaf)->key)  
            insertNode(&(*leaf)->dir, key_value);  
    }  
  
typedef struct No{  
    int key;  
    struct No *esq;  
    struct No *dir;  
} node;
```



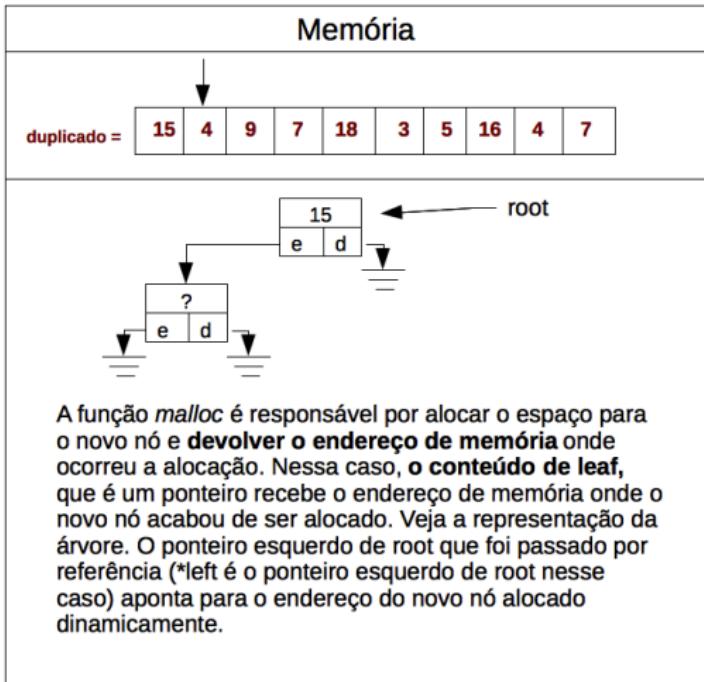
# Inserir Nós na Árvore Binária

```
void makeTree(node **root){  
    *root = NULL;  
}  
  
***** inserir *****/  
void insertNode(node **leaf, int key_value){  
    if(*leaf == NULL){  
        *leaf = (node *) malloc(sizeof(node));  
        (*leaf)->esq = NULL;  
        (*leaf)->dir = NULL;  
        (*leaf)->key = key_value;  
    }else{  
        if(key_value < (*leaf)->key)  
            insertNode(&(*leaf)->esq, key_value);  
        if(key_value > (*leaf)->key)  
            insertNode(&(*leaf)->dir, key_value);  
    }  
  
typedef struct No{  
    int key;  
    struct No *esq;  
    struct No *dir;  
} node;
```



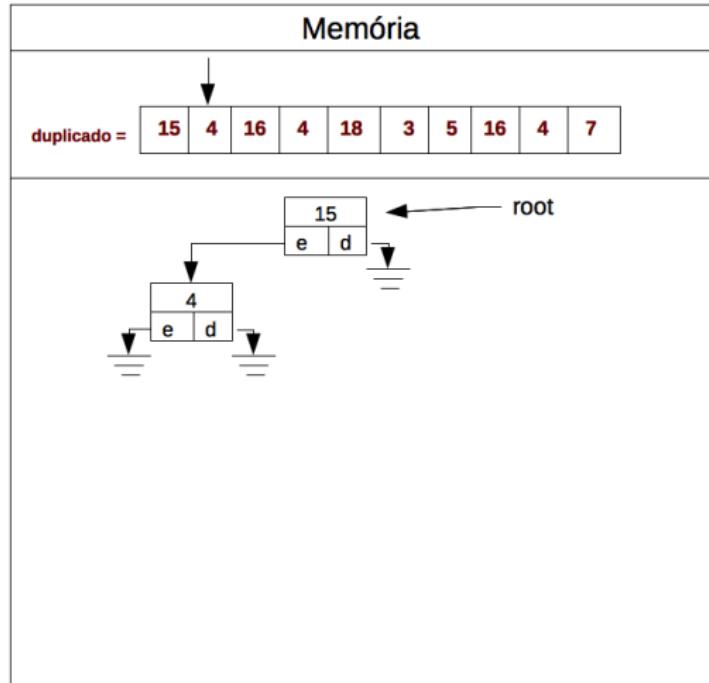
# Inserir Nós na Árvore Binária

```
void makeTree(node **root){  
    *root = NULL;  
}  
  
***** inserir *****/  
void insertNode(node **leaf, int key_value){  
    if(*leaf == NULL){  
        *leaf = (node *) malloc(sizeof(node));  
        (*leaf)->esq = NULL;  
        (*leaf)->dir = NULL;  
        (*leaf)->key = key_value;  
    }else{  
        if(key_value < (*leaf)->key)  
            insertNode(&(*leaf)->esq, key_value);  
        if(key_value > (*leaf)->key)  
            insertNode(&(*leaf)->dir, key_value);  
    }  
  
typedef struct No{  
    int key;  
    struct No *esq;  
    struct No *dir;  
} node;
```



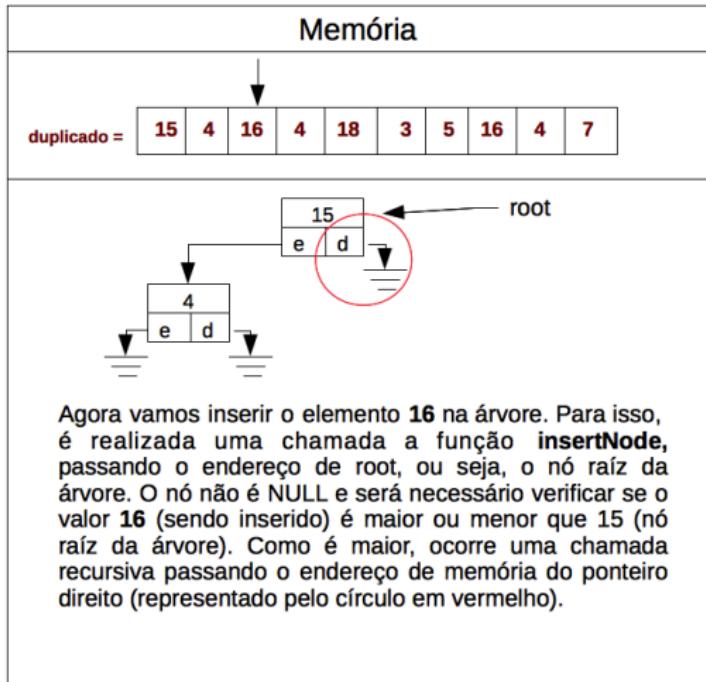
# Inserir Nós na Árvore Binária

```
void makeTree(node **root){  
    *root = NULL;  
}  
  
***** inserir *****/  
void insertNode(node **leaf, int key_value){  
    if(*leaf == NULL){  
        *leaf = (node *) malloc(sizeof(node));  
        (*leaf)->esq = NULL;  
        (*leaf)->dir = NULL;  
        (*leaf)->key = key_value;  
    }else{  
        if(key_value < (*leaf)->key)  
            insertNode(&(*leaf)->esq, key_value);  
        if(key_value > (*leaf)->key)  
            insertNode(&(*leaf)->dir, key_value);  
    }  
  
typedef struct No{  
    int key;  
    struct No *esq;  
    struct No *dir;  
} node;
```



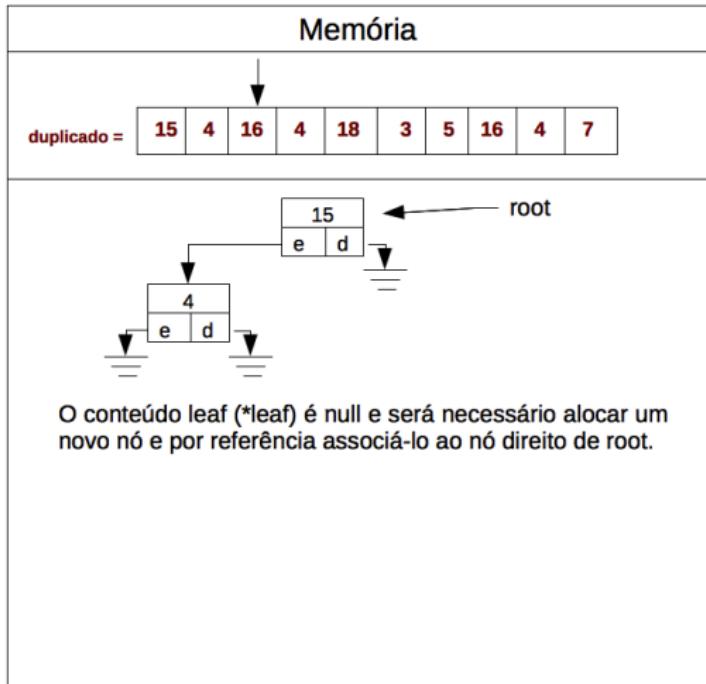
# Inserir Nós na Árvore Binária

```
void makeTree(node **root){  
    *root = NULL;  
}  
  
***** inserir *****/  
void insertNode(node **leaf, int key_value){  
    if(*leaf == NULL){  
        *leaf = (node *) malloc(sizeof(node));  
        (*leaf)->esq = NULL;  
        (*leaf)->dir = NULL;  
        (*leaf)->key = key_value;  
    }else{  
        if(key_value < (*leaf)->key)  
            insertNode(&(*leaf)->esq, key_value);  
        if(key_value > (*leaf)->key)  
            insertNode(&(*leaf)->dir, key_value);  
    }  
  
    typedef struct No{  
        int key;  
        struct No *esq;  
        struct No *dir;  
    } node;
```



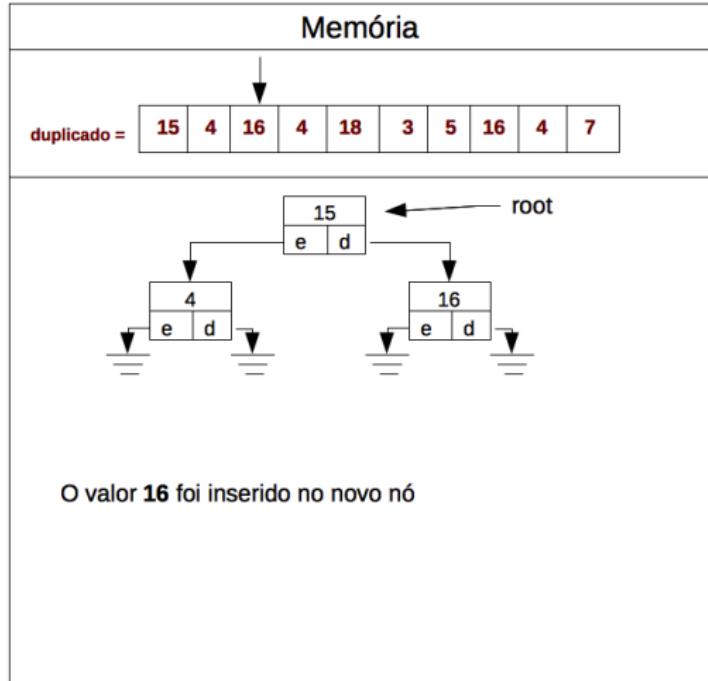
# Inserir Nós na Árvore Binária

```
void makeTree(node **root){  
    *root = NULL;  
}  
  
***** inserir *****/  
void insertNode(node **leaf, int key_value){  
    if(*leaf == NULL){  
        *leaf = (node *) malloc(sizeof(node));  
        (*leaf)->esq = NULL;  
        (*leaf)->dir = NULL;  
        (*leaf)->key = key_value;  
    }else{  
        if(key_value < (*leaf)->key)  
            insertNode(&(*leaf)->esq, key_value);  
        if(key_value > (*leaf)->key)  
            insertNode(&(*leaf)->dir, key_value);  
    }  
  
typedef struct No{  
    int key;  
    struct No *esq;  
    struct No *dir;  
} node;
```



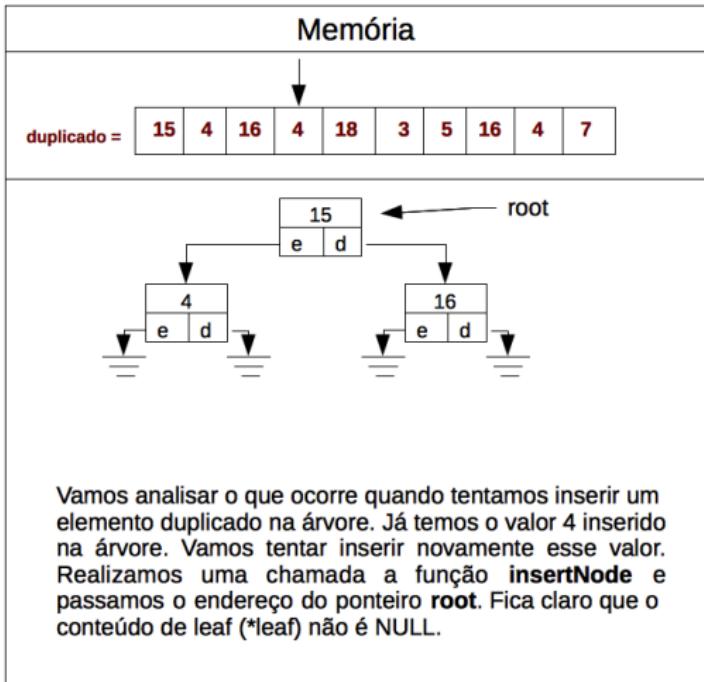
# Inserir Nós na Árvore Binária

```
void makeTree(node **root){  
    *root = NULL;  
}  
  
***** inserir *****/  
void insertNode(node **leaf, int key_value){  
    if(*leaf == NULL){  
        *leaf = (node *) malloc(sizeof(node));  
        (*leaf)->esq = NULL;  
        (*leaf)->dir = NULL;  
        (*leaf)->key = key_value;  
    }  
    else{  
        if(key_value < (*leaf)->key)  
            insertNode(&(*leaf)->esq, key_value);  
        if(key_value > (*leaf)->key)  
            insertNode(&(*leaf)->dir, key_value);  
    }  
  
typedef struct No{  
    int key;  
    struct No *esq;  
    struct No *dir;  
} node;
```



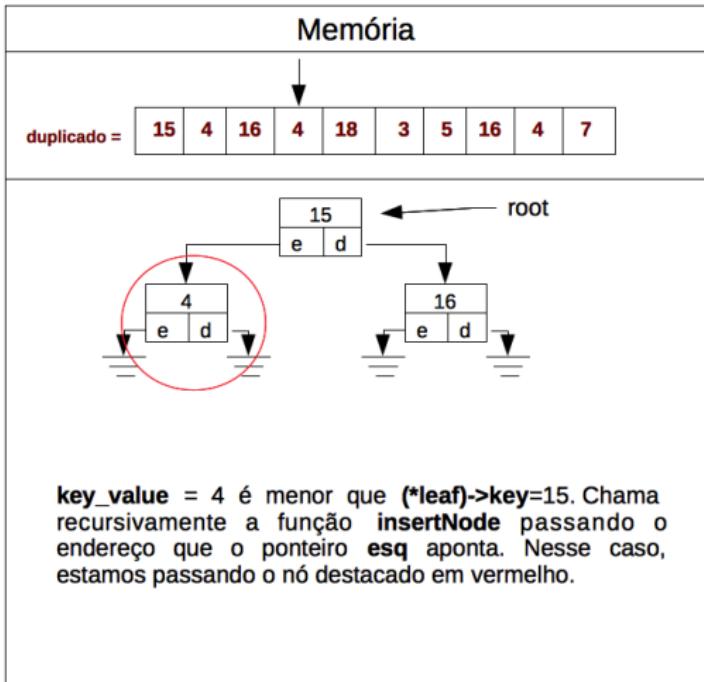
# Inserir Nós na Árvore Binária

```
void makeTree(node **root){  
    *root = NULL;  
}  
  
***** inserir *****/  
void insertNode(node **leaf, int key_value){  
    if(*leaf == NULL){  
        *leaf = (node *) malloc(sizeof(node));  
        (*leaf)->esq = NULL;  
        (*leaf)->dir = NULL;  
        (*leaf)->key = key_value;  
    }else{  
        if(key_value < (*leaf)->key)  
            insertNode(&(*leaf)->esq, key_value);  
        if(key_value > (*leaf)->key)  
            insertNode(&(*leaf)->dir, key_value);  
    }  
  
    typedef struct No{  
        int key;  
        struct No *esq;  
        struct No *dir;  
    } node;
```



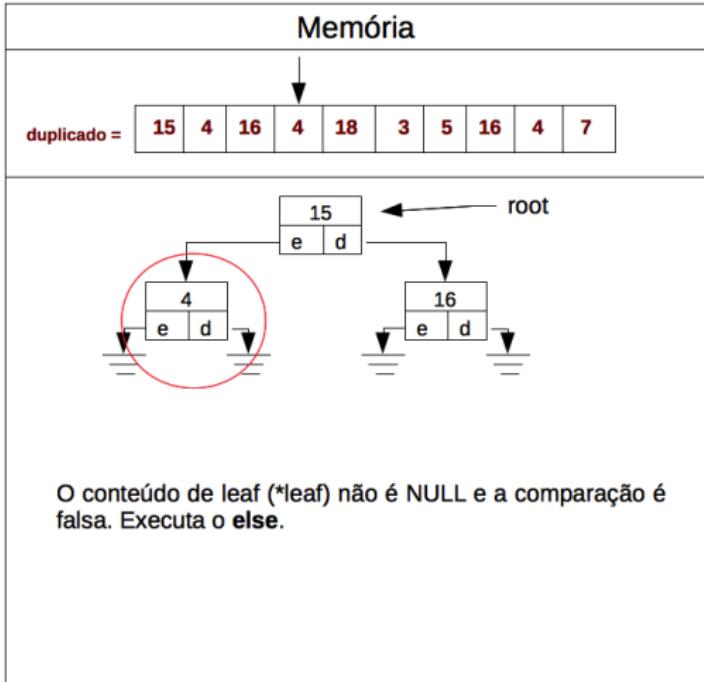
# Inserir Nós na Árvore Binária

```
void makeTree(node **root){  
    *root = NULL;  
}  
  
***** inserir *****/  
void insertNode(node **leaf, int key_value){  
    if(*leaf == NULL){  
        *leaf = (node *) malloc(sizeof(node));  
        (*leaf)->esq = NULL;  
        (*leaf)->dir = NULL;  
        (*leaf)->key = key_value;  
    }else{  
        if(key_value < (*leaf)->key)  
            insertNode(&(*leaf)->esq, key_value);  
        if(key_value > (*leaf)->key)  
            insertNode(&(*leaf)->dir, key_value);  
    }  
  
    typedef struct No{  
        int key;  
        struct No *esq;  
        struct No *dir;  
    } node;
```



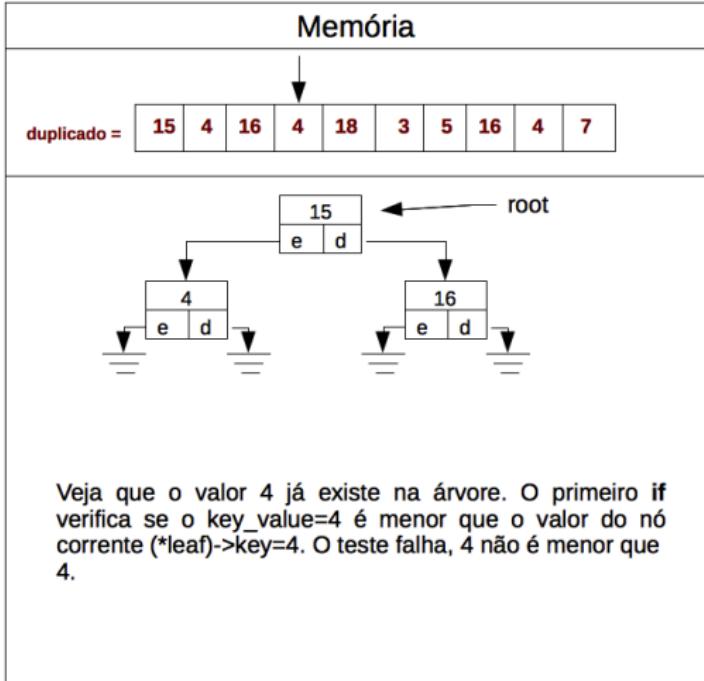
# Inserir Nós na Árvore Binária

```
void makeTree(node **root){  
    *root = NULL;  
}  
  
***** inserir *****/  
void insertNode(node **leaf, int key_value){  
    if(*leaf == NULL){  
        *leaf = (node *) malloc(sizeof(node));  
        (*leaf)->esq = NULL;  
        (*leaf)->dir = NULL;  
        (*leaf)->key = key_value;  
    }else{  
        if(key_value < (*leaf)->key)  
            insertNode(&(*leaf)->esq, key_value);  
        if(key_value > (*leaf)->key)  
            insertNode(&(*leaf)->dir, key_value);  
    }  
  
    typedef struct No{  
        int key;  
        struct No *esq;  
        struct No *dir;  
    } node;
```



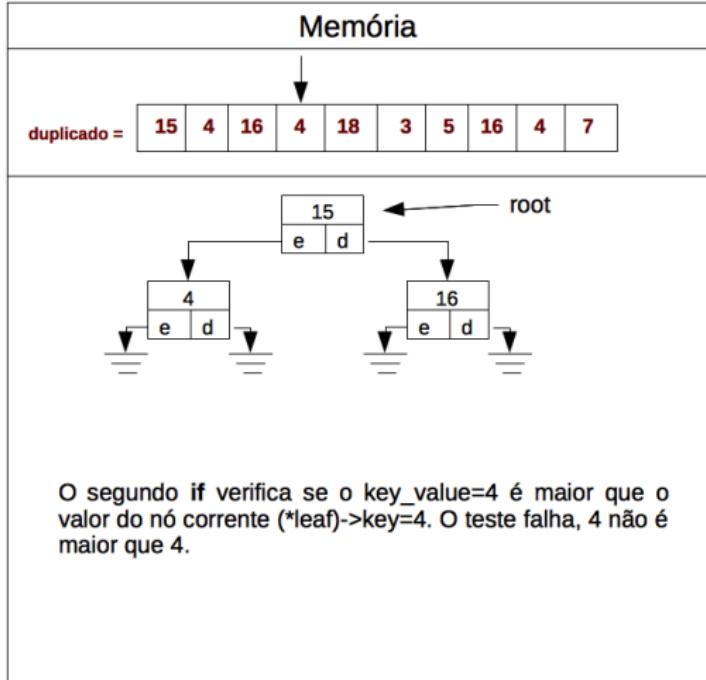
# Inserir Nós na Árvore Binária

```
void makeTree(node **root){  
    *root = NULL;  
}  
  
***** inserir *****/  
void insertNode(node **leaf, int key_value){  
    if(*leaf == NULL){  
        *leaf = (node *) malloc(sizeof(node));  
        (*leaf)->esq = NULL;  
        (*leaf)->dir = NULL;  
        (*leaf)->key = key_value;  
    }else{  
        if(key_value < (*leaf)->key)  
            insertNode(&(*leaf)->esq, key_value);  
        if(key_value > (*leaf)->key)  
            insertNode(&(*leaf)->dir, key_value);  
    }  
  
    typedef struct No{  
        int key;  
        struct No *esq;  
        struct No *dir;  
    } node;
```



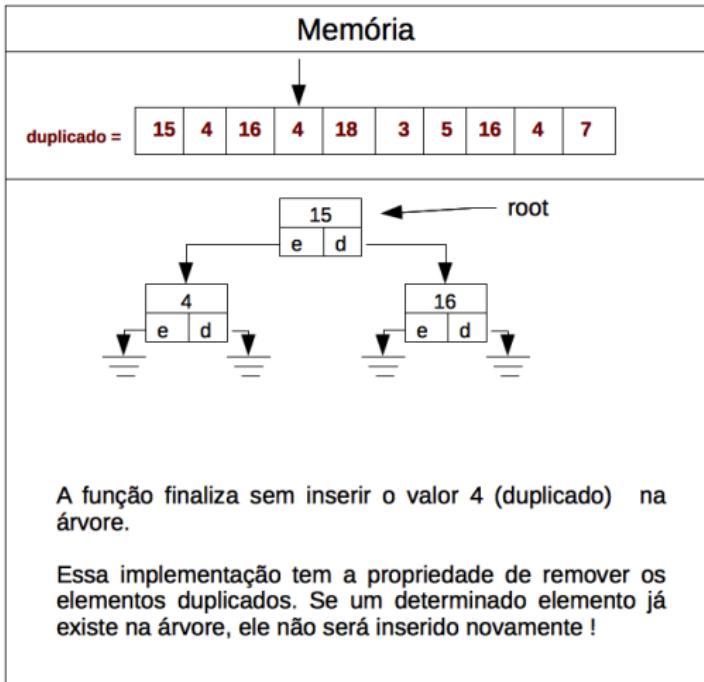
# Inserir Nós na Árvore Binária

```
void makeTree(node **root){  
    *root = NULL;  
}  
  
***** inserir *****/  
void insertNode(node **leaf, int key_value){  
    if(*leaf == NULL){  
        *leaf = (node *) malloc(sizeof(node));  
        (*leaf)->esq = NULL;  
        (*leaf)->dir = NULL;  
        (*leaf)->key = key_value;  
    }else{  
        if(key_value < (*leaf)->key)  
            insertNode(&(*leaf)->esq, key_value);  
        if(key_value > (*leaf)->key)  
            insertNode(&(*leaf)->dir, key_value);  
    }  
  
typedef struct No{  
    int key;  
    struct No *esq;  
    struct No *dir;  
} node;
```

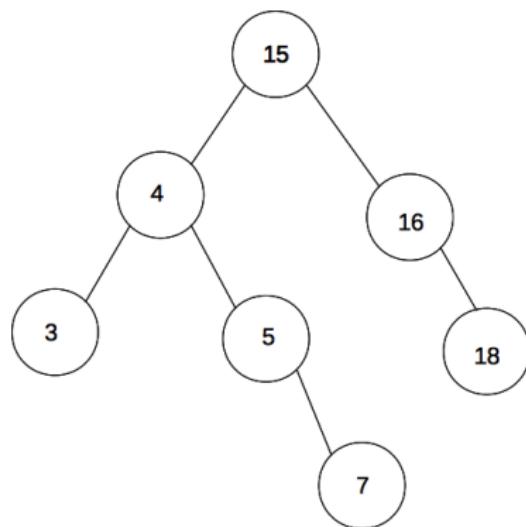


# Inserir Nós na Árvore Binária

```
void makeTree(node **root){  
    *root = NULL;  
}  
  
***** inserir *****/  
void insertNode(node **leaf, int key_value){  
    if(*leaf == NULL){  
        *leaf = (node *) malloc(sizeof(node));  
        (*leaf)->esq = NULL;  
        (*leaf)->dir = NULL;  
        (*leaf)->key = key_value;  
    }else{  
        if(key_value < (*leaf)->key)  
            insertNode(&(*leaf)->esq, key_value);  
        if(key_value > (*leaf)->key)  
            insertNode(&(*leaf)->dir, key_value);  
    }  
}  
  
typedef struct No{  
    int key;  
    struct No *esq;  
    struct No *dir;  
} node;
```



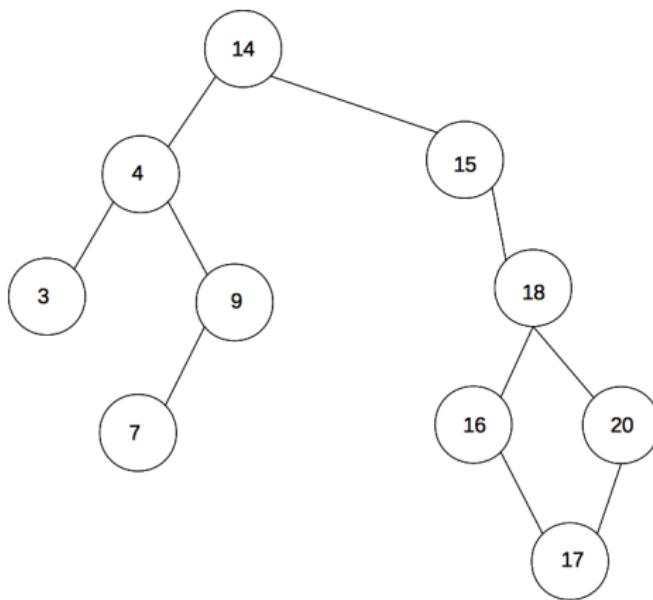
## Representação da árvore binária



- Para remover um determinado nó de uma árvore binária, é preciso considerar três casos:
  - ① Não existe o nó com o valor que está sendo solicitada a remoção.
  - ② O nó contendo o valor a ser removido tem 0 ou 1 filho.
  - ③ o nó contendo o valor a ser removido tem 2 filhos.
- Para o primeiro caso não temos o que fazer. O elemento solicitado não existe na árvore para ser removido.
- Vamos simular os outros dois casos quando não existe filho, existe um filho ou existem 2 filhos.

# Remover Nós

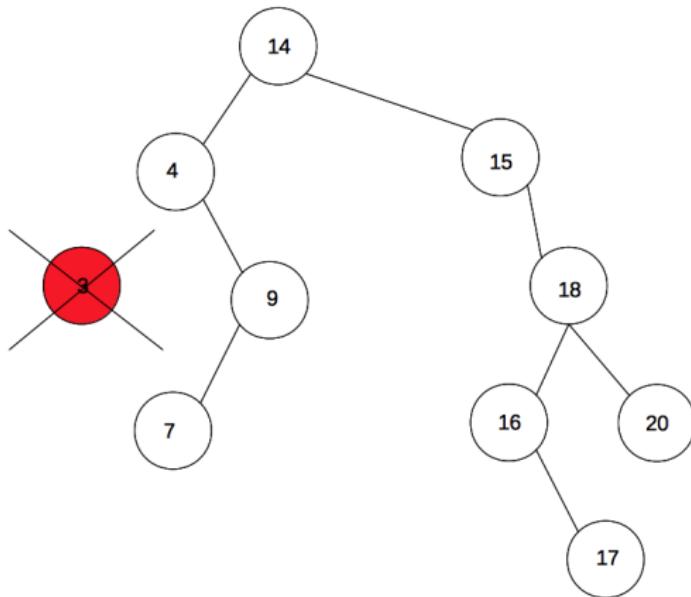
Considerando a árvore binária apresentada na figura abaixo, vamos simular o caso 2, na situação em que o nó a ser removido não possui **filhos**. Dessa forma, vamos remover o nó com o valor **3** dessa árvore binária.



# Remover Nós

- O nó não tem filhos e dessa forma atribuímos o valor **NULL** ao seu conteúdo.

```
//fragmento de código para remover o nó
node *pAux = *root;
if (((*root)->esq == NULL) && ((*root)->dir == NULL)){
    // se não houver filhos...
    free(pAux);
    (*root) = NULL;
}
```

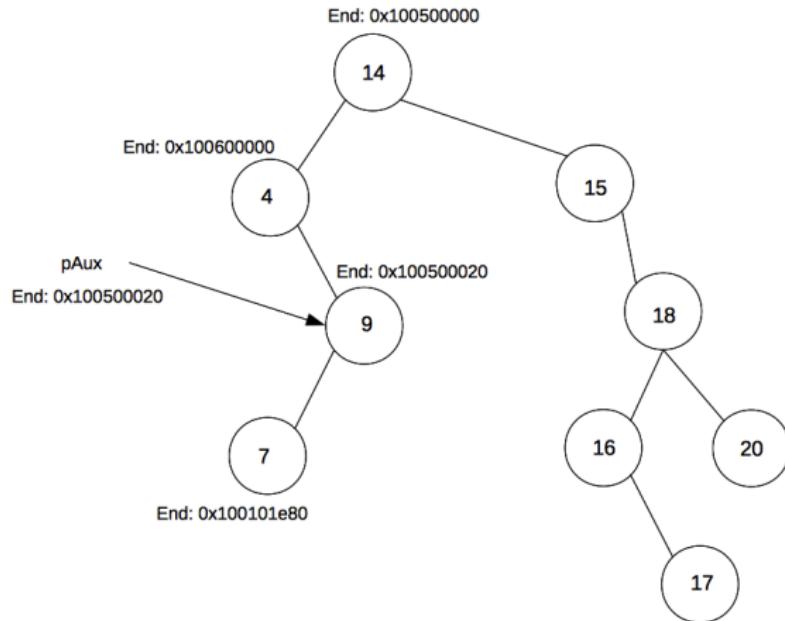


# Remover Nós - Nô pai com um filho

- Vamos remover o nó com o valor **9**.
- Inicialmente associamos um ponteiro auxiliar ao nó que iremos remover. *pAux* aponta para o nó que será removido.

```
node *pAux = *root;
```

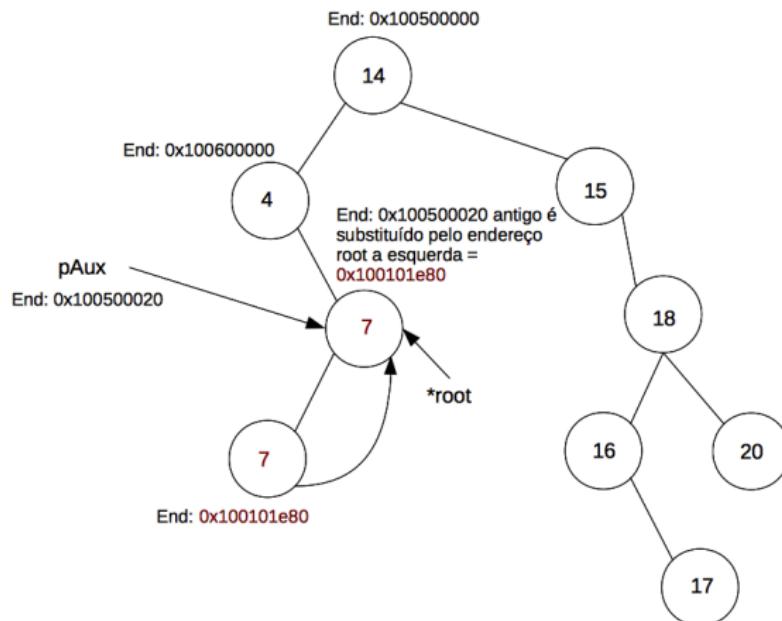
```
//so tem filho da esquerda
if ((*root)->dir == NULL){
    (*root) = (*root)->esq;
    pAux->esq = NULL;
    free(pAux);
    pAux = NULL;
}
```



# Remover Nós - Nô pai com um filho

- $*\text{root} = (*\text{root})-\text{esq}$ .
- Fazemos o conteúdo do nó a ser removido apontar para o nó filho a esquerda.

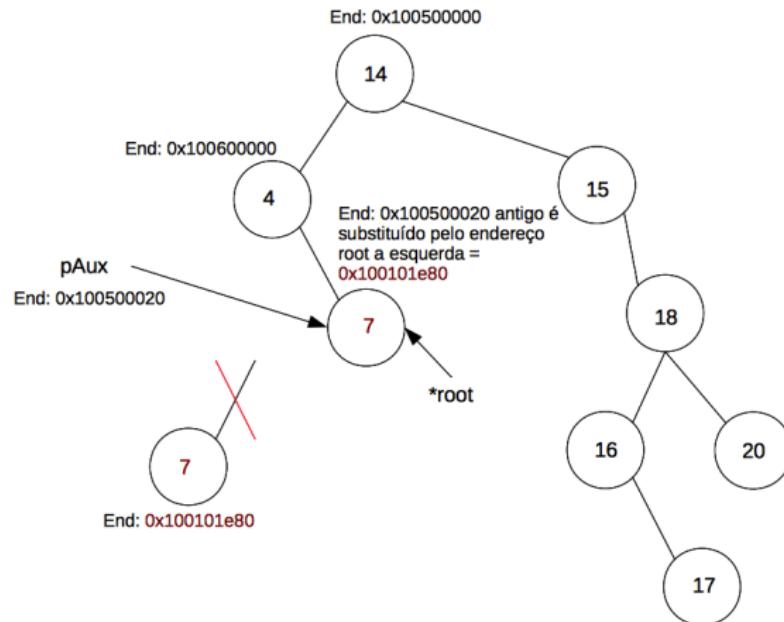
```
node *pAux = *\text{root};  
  
//so tem filho da esquerda  
if ((*root)->dir == NULL){  
    (*root) = (*root)->esq;  
    pAux->esq = NULL;  
    free(pAux); pAux = NULL;  
}
```



# Remover Nós - Nô pai com um filho

- $pAux->esq = NULL$ .
- Após efetuar a troca do nó filho com o pai, remove a ligação que existia entre eles.

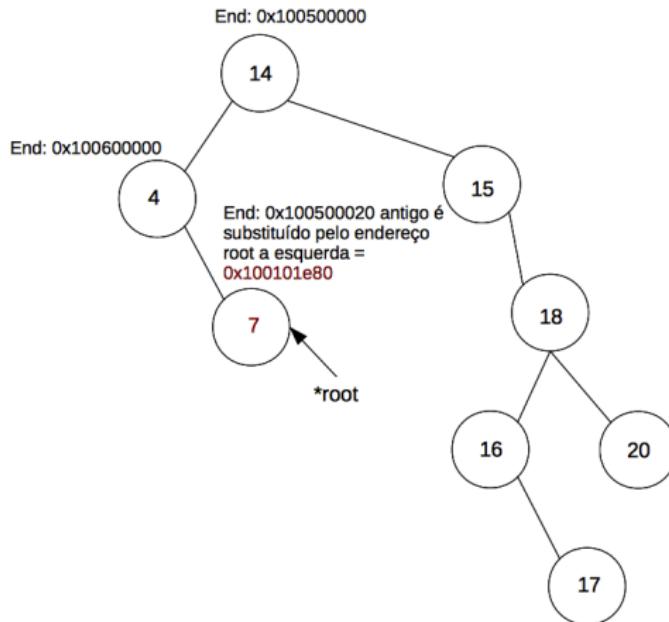
```
node *pAux = *root;  
  
//so tem filho da esquerda  
if ((*root)->dir == NULL){  
    (*root) = (*root)->esq;  
    pAux->esq = NULL;  
    free(pAux); pAux = NULL;  
}
```



# Remover Nós - Nô pai com um filho

- free(pAux) e pAux=NULL.
- Remoção efetuada. Libera a memória !

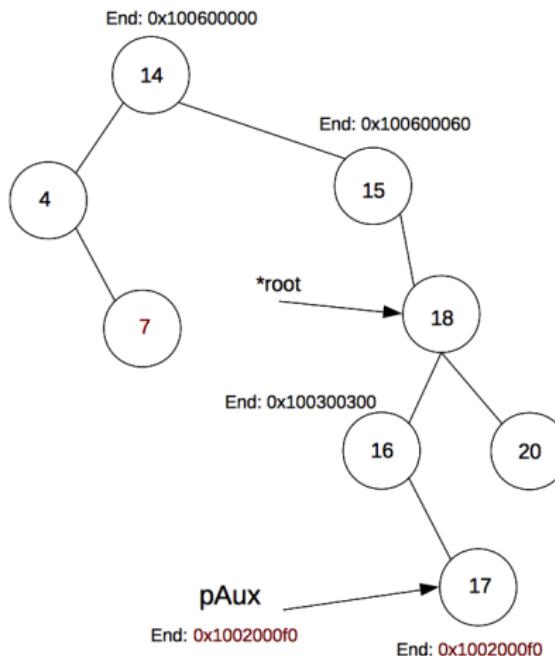
```
node *pAux = *root;  
  
//so tem filho da esquerda  
if ((*root)->dir == NULL){  
    (*root) = (*root)->esq;  
    pAux->esq = NULL;  
    free(pAux); pAux = NULL;  
}
```



# Remover Nós - Nô pai com dois filhos

- Vamos remover o nó 18 que tem filhos a esquerda e direita.
- A função `find_max_right(&(*root)->esq)` procura o maior filho da subárvore à esquerda e retorna um ponteiro para esse nó.

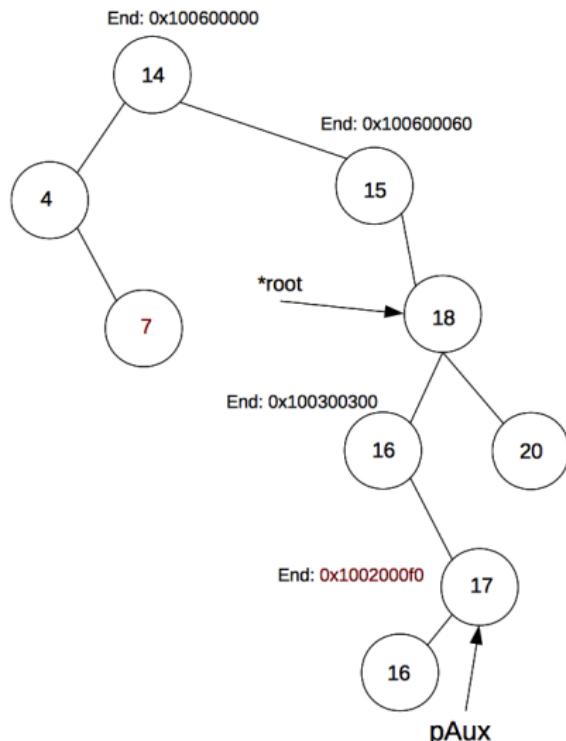
```
node *pAux;  
  
//Escolhi fazer o maior filho direito da subárvore esquerda.  
pAux = find_max_right(&(*root)->esq);  
pAux->esq = (*root)->esq;  
pAux->dir = (*root)->dir;  
(*root)->esq = (*root)->dir = NULL;  
free((*root)); *root = pAux; pAux = NULL;  
}
```



# Remover Nós - Nô pai com dois filhos

- O conteúdo do ponteiro esquerdo de *pAux* recebe o endereço do ponteiro esquerdo do nó que será removido.

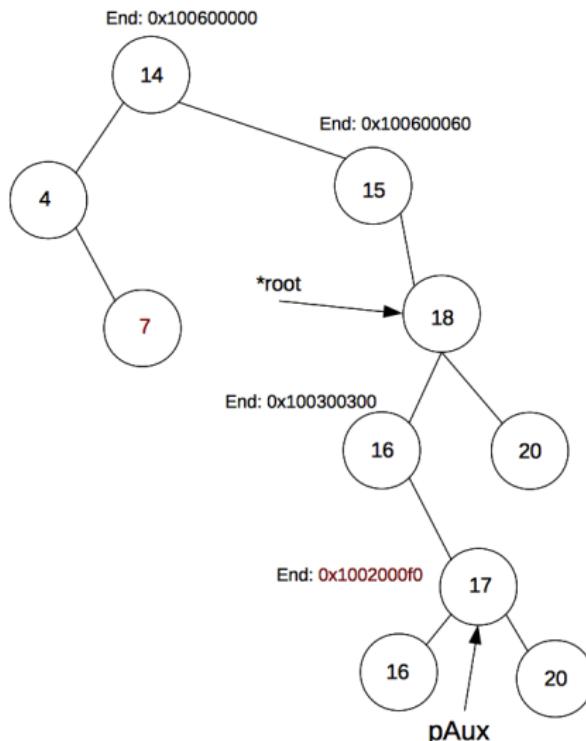
```
node *pAux;  
  
//Escolhi fazer o maior filho direito da subarvore esquerda.  
pAux = find_max_right(&(*root)->esq);  
pAux->esq = (*root)->esq;  
pAux->dir = (*root)->dir;  
(*root)->esq = (*root)->dir = NULL;  
free((*root)); *root = pAux; pAux = NULL;  
}
```



# Remover Nós - Nô pai com dois filhos

- O conteúdo do ponteiro direito de *pAux* recebe o endereço do ponteiro direito do nó que será removido.

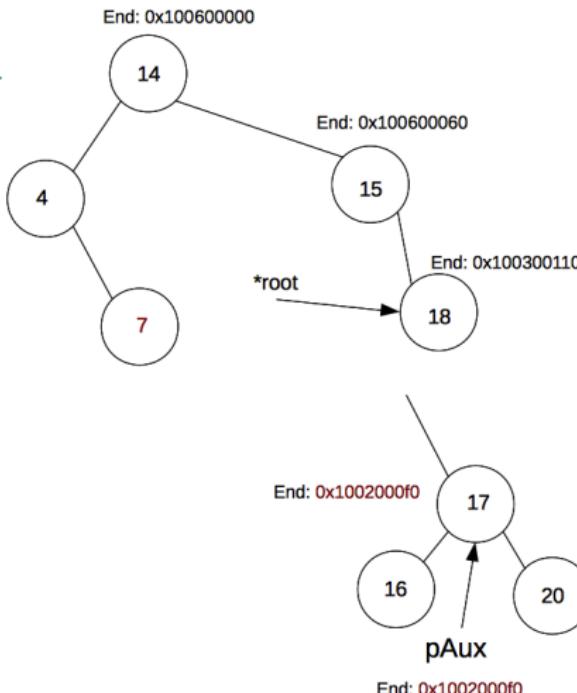
```
node *pAux;  
  
//Escolhi fazer o maior filho direito da subarvore esquerda.  
pAux = find_max_right(&(*root)->esq);  
pAux->esq = (*root)->esq;  
pAux->dir = (*root)->dir;  
(*root)->esq = (*root)->dir = NULL;  
free((*root)); *root = pAux; pAux = NULL;  
}
```



# Remover Nós - Nó pai com dois filhos

- Os ponteiros *esquerdo* e *direito* do nó a ser removido são apontados para **NULL**.
- Na prática, o nó a ser removido perde a referência aos seus nós filho.

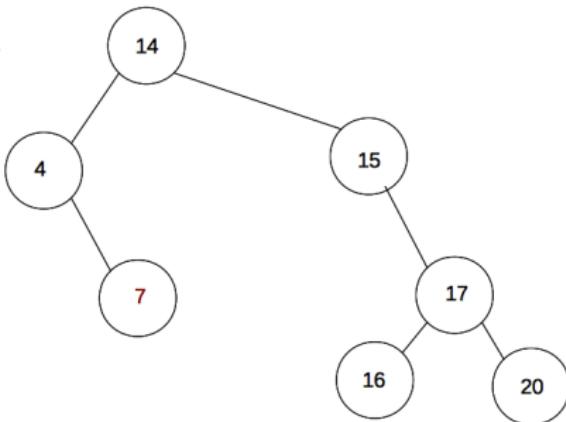
```
node *pAux;  
  
//Escolhi fazer o maior filho direito da subarvore esquerda.  
pAux = find_max_right(&(*root)->esq);  
pAux->esq = (*root)->esq;  
pAux->dir = (*root)->dir;  
(*root)->esq = (*root)->dir = NULL;  
free((*root)); *root = pAux; pAux = NULL;  
}
```



## Remover Nós - Nô pai com dois filhos

- O nó apontado por `*root` que é o nó a ser removido é liberado (`free()`).
- O ponteiro `*root` recebe o endereço de `*pAux`.
- O nó foi removido.

```
node *pAux;  
  
//Escolhi fazer o maior filho direito da subarvore esquerda.  
pAux = find_max_right(&(*root)->esq);  
pAux->esq = (*root)->esq;  
pAux->dir = (*root)->dir;  
(*root)->esq = (*root)->dir = NULL;  
free((*root)); *root = pAux; pAux = NULL;  
}
```



- Uma operação comum é **percorrer** uma árvore binária, ou seja, percorrer a árvore enumerando cada um de seus nós uma vez. É possível apenas imprimir o conteúdo de cada nó ao enumerá-lo, ou pode-se realizar algum processamento. Se qual for o caso, iremos utilizar o termo **visitar** cada nó à medida que é enumerado.
- Diferente das listas e vetores que são lineares, não existe uma ordem *natural* para visitar os nós de uma árvore.
- Vamos definir três métodos de percurso: **pré-ordem, em-ordem e pós-ordem**.
- A única diferença entre os métodos é a ordem na qual essas três operações são efetuadas.

# Percorso em Pré-ordem

- Para percorrer uma árvore binária não-vazia em **pré-ordem** (conhecida também como **percurso em profundidade**), efetuamos as três seguintes operações:
  - Visitamos a raiz.
  - Percorremos a subárvore esquerda em ordem prévia.
  - Percorremos a subárvore direita em ordem prévia.

```
void preOrder(node *root){  
    if(!root)  
        return;  
    if(root->key){  
        printf("%d ", root->key);  
    }  
  
    preOrder(root->esq);  
    preOrder(root->dir);  
}
```

- Para percorrer uma árvore binária não-vazia em **em-ordem** (conhecida também como **ordem simétrica**), efetuamos as três seguintes operações:
  - Percorremos a subárvore esquerda em ordem simétrica.
  - Visitamos a raiz.
  - Percorremos a subárvore direita em ordem simétrica.

```
void inOrder(node *root){  
    if(!root)  
        return;  
  
    inOrder(root->esq);  
    if(root->key){  
        printf("%d ", root->key);  
    }  
    inOrder(root->dir);  
}
```

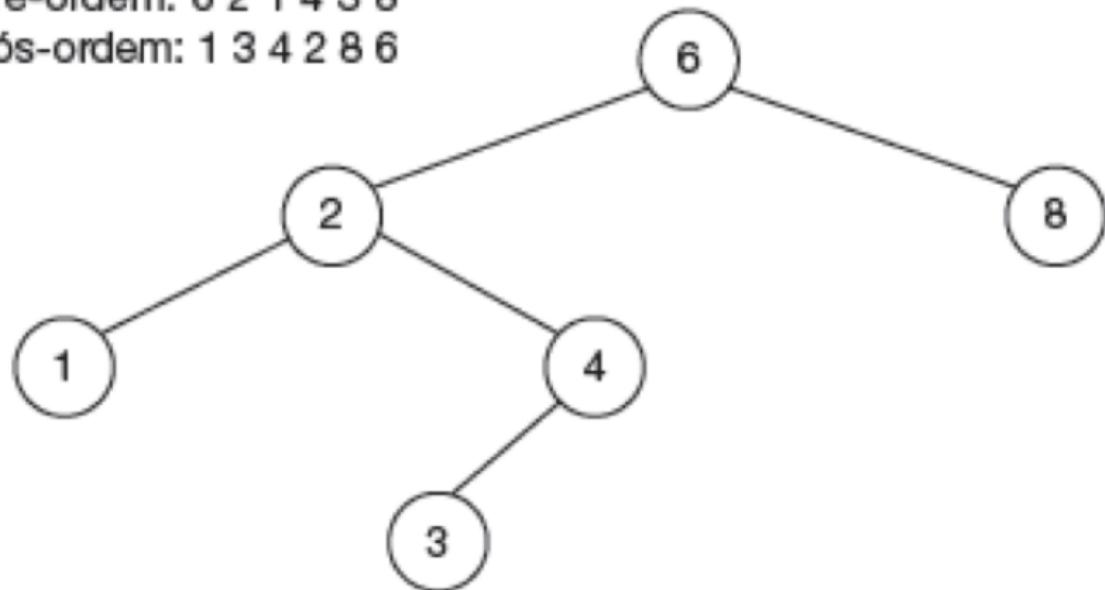
- Para percorrer uma árvore binária não-vazia em **pós-ordem**, efetuamos as três seguintes operações:
  - Percorremos a subárvore esquerda em ordem posterior.
  - Percorremos a subárvore direita em ordem posterior.
  - Visitamos a raiz.

```
void postOrder(node *root){  
    if(!root)  
        return;  
  
    postOrder(root->esq);  
    postOrder(root->dir);  
  
    if(root->key){  
        printf("%d ", root->key);  
    }  
}
```

Em ordem: 1 2 3 4 6 8

Pré-ordem: 6 2 1 4 3 8

Pós-ordem: 1 3 4 2 8 6



## 1 Introdução

## 2 Árvores

### 3 Árvores Binárias

- Operações em Árvores Binárias

### 4 Aplicações de Árvores Binárias

- Árvore Binária para Remoção de Elementos duplicados
- Árvore Binária para Ordenação de Dados
- Algoritmo HUFFMAN

## 5 Árvore AVL

## 1 Introdução

## 2 Árvores

### 3 Árvores Binárias

- Operações em Árvores Binárias

### 4 Aplicações de Árvores Binárias

- Árvore Binária para Remoção de Elementos duplicados
- Árvore Binária para Ordenação de Dados
- Algoritmo HUFFMAN

## 5 Árvore AVL

- Uma **árvore binária** é uma estrutura de dados útil quando precisam ser tomadas decisões bidirecionais em cada ponto de um processo.
- Por exemplo, suponha que precisemos encontrar todas as repetições em uma lista de números. Uma maneira de resolver isso é comparar cada número com todos os que o precedem. Entretanto, isso envolve um grande número de comparações.
- O número de comparações pode ser reduzido utilizando uma árvore binária. Vejamos:
  - ① O primeiro número da lista é inserido na raiz da árvore.
  - ② Cada número sucessivo na lista é, então, comparado ao número na raiz. Se coincidirem teremos uma repetição.
  - ③ Se for menor, examinaremos as subárvores à esquerda.
  - ④ Se for maior, examinaremos as subárvores à direita.
  - ⑤ Se ao finalizar o percurso nas subárvores e a posição do número estiver vazia, o número não estará repetido e será inserido um novo nó nesta posição da árvore.

## 1 Introdução

## 2 Árvores

### 3 Árvores Binárias

- Operações em Árvores Binárias

### 4 Aplicações de Árvores Binárias

- Árvore Binária para Remoção de Elementos duplicados
- Árvore Binária para Ordenação de Dados
- Algoritmo HUFFMAN

## 5 Árvore AVL

- Uma **árvore binária** pode ser utilizada para **ordenação** de dados.
- Dada uma lista de números, por exemplo, queremos imprimi-los em ordem ascendente. Ao lermos os números, eles podem ser inseridos em uma árvore binária e impressos em ordem ascendente posteriormente.
- Entretanto, ao contrário do algoritmo de remoção de duplicatas, essa abordagem exige que os valores repetidos sejam mantidos.
- Para isso, quando um determinado elemento é comparado ao conteúdo de um nó da árvore, uma ramificação esquerda é usada se o número for **menor** que o conteúdo do nó, e uma ramificação direita se ele for **maior** ou **igual** ao conteúdo do nó.

# Aplicações de árvores binárias

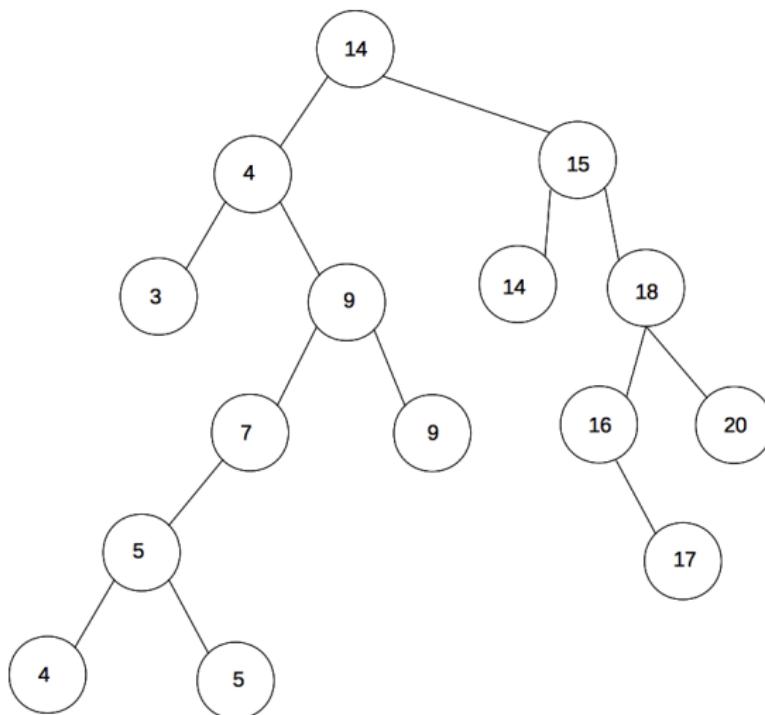
- A única alteração necessária no código para inserir um nó na árvore é o acréscimo do sinal de = na segunda chamada recursiva da função **insertNode**.

```
void insertNode(node **leaf, int key_value){  
    if(*leaf == NULL){  
        *leaf = (node *) malloc(sizeof(node));  
        (*leaf)->esq = NULL;  
        (*leaf)->dir = NULL;  
        (*leaf)->key = key_value;  
    }else{  
        if(key_value < (*leaf)->key)  
            insertNode(&(*leaf)->esq, key_value);  
        if(key_value >= (*leaf)->key)  
            insertNode(&(*leaf)->dir, key_value);  
    }  
}
```

# Aplicações de árvores binárias

Considere o vetor com os elementos: 14,15,4,9,7,18,3,5,16,4,20,17,9,14,5.

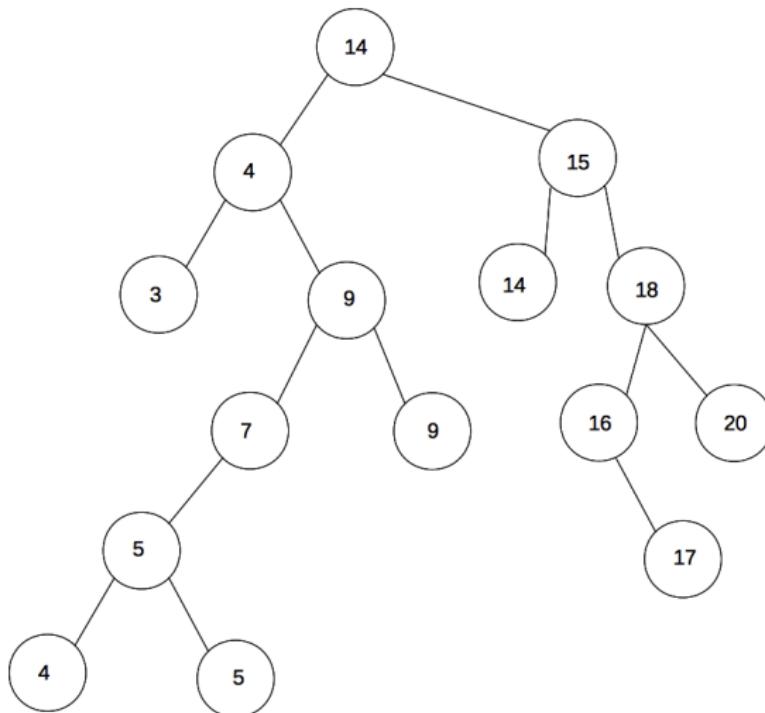
A árvore binária para classificação é apresentada:



- Uma árvore binária construída com o propósito de classificar tem a propriedade de todos os elementos na subárvore esquerda de um nó **n** serem menores que o conteúdo de **n**. E todos os elementos na subárvore a direita de **n** serem maiores ou iguais ao conteúdo de **n**. Essa árvore é chamada de **árvore de busca binária**.
- Se uma árvore de busca binária for percorrida em **ordem simétrica** ou **percurso em-ordem**, os elementos serão impressos em ordem ascendente.

# Aplicações de árvores binárias

Como exercício percorra a árvore binária de busca em ordem simétrica.



## 1 Introdução

## 2 Árvores

### 3 Árvores Binárias

- Operações em Árvores Binárias

### 4 Aplicações de Árvores Binárias

- Árvore Binária para Remoção de Elementos duplicados
- Árvore Binária para Ordenação de Dados
- **Algoritmo HUFFMAN**

## 5 Árvore AVL

## Um exemplo: O algoritmo de HUFFMAN

- A **codificação de Huffman** é um método de compressão que usa as probabilidades de ocorrência dos símbolos no conjunto de dados a ser comprimido para determinar códigos de tamanho variável para cada símbolo. Ele foi desenvolvido em 1952 por *David A. Huffman*.
- Suponha que tenhamos um alfabeto de **n** símbolos e uma extensa mensagem consistindo em símbolos desse alfabeto. O objetivo é codificar a mensagem como uma extensa cadeia de *bits* atribuindo uma string de bits como código a cada símbolo do alfabeto.
- Vamos supor que nosso alfabeto consista nos quatro símbolos: **A**, **B**, **C** e **D** e que são atribuídos códigos a esses símbolos, como segue:

Símbolo	Código
A	010
B	100
C	000
D	111

## Um exemplo: O algoritmo de HUFFMAN

- A mensagem **ABACCD**A seria, então, codificada como *010100010000000111010*. Essa codificação é ineficiente porque são usados três bits para cada símbolo, de modo que 21 bits são necessários para codificar a mensagem inteira.
- Vamos supor que um código de dois bits seja atribuído a cada símbolo, como segue:

Símbolo	Código
A	00
B	01
C	10
D	11

- Dessa forma, o código para a mensagem seria *00010010101100* o que exige somente 14 bits.
- **Melhorou, mas nosso objetivo é descobrir um código que diminua ainda mais o tamanho da mensagem codificada.**

## Um exemplo: O algoritmo de HUFFMAN

- Considerando a mensagem **ABACCDA** podemos verificar que os símbolos **B** e **D** aparecem apenas uma vez na mensagem, enquanto o símbolo **A** aparece três vezes.
- Se escolhermos um código para representar **A** mais curto que o código dos símbolos **B** e **D**, o tamanho da mensagem codificada será menor.
- Isso ocorre porque o código pequeno que representa o símbolo **A**, nesse exemplo, aparecia com mais frequência que o código extenso, representando os símbolos **B** e **D**.
- Vamos atribuir os códigos como segue:

Símbolo	Código
A	0
B	110
C	10
D	111

- Utilizando esse código a mensagem seria *0110010101110* o que exige apenas 13 bits.

## Um exemplo: O algoritmo de HUFFMAN

- A construção dos códigos que representam os símbolos devem seguir regras para que seja possível o processo de **decodificação** da mensagem.
- O código para um símbolo **não** pode ser prefixo do código para o outro. Se isso acontecer, no momento da decodificação da mensagem, não será possível determinar se o código representa um símbolo ou se ele é primeira parte de outro símbolo. Vejamos um exemplo considerando a tabela de símbolos abaixo:

Símbolo	Código
A	0
B	110
C	10
D	111

- e a mensagem *0110010101110*.

## Um exemplo: O algoritmo de HUFFMAN

- Vejamos um exemplo considerando a Mensagem: 0110010101110.

Símbolo	Código
A	0
B	110
C	10
D	111

- A decodificação inicia da esquerda para direita lendo o código e procurando o símbolo equivalente na tabela. O primeiro código localizado é um **0**, o símbolo referente a esse código é o **A**. O processo continua e lê o próximo código, nesse caso, o valor **1**; Temos três opções que começam com **1**. O próximo código lido é o **1** e temos agora duas possibilidades: O símbolo é o **B** ou **C**. O próximo código lido é o **0**, o símbolo será portanto o **B** (110).
- O processo será repetido até que o fim da mensagem seja alcançado.

# Um exemplo: O algoritmo de HUFFMAN

- O objetivo do **Algoritmo de HUFFMAN** é fornecer mecanismos para criar a tabela de símbolos, em função da frequência de cada símbolo em uma mensagem.
- Para implementar a tabela, encontre os dois símbolos que aparecem com menos frequência. Em nosso exemplo, são os símbolos **B** e **C**.

Alfabeto: A,B,C,D  
Mensagem: ABACCDAA  
Codificada : ?

Símbolo	Frequência
A	3
C	2
B	1
D	1

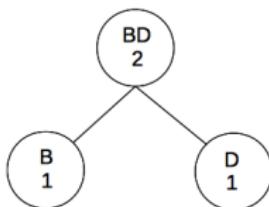


# Um exemplo: O algoritmo de HUFFMAN

- Combine esses dois símbolos no símbolo isolado **BD**, cujo código representa a noção de que um símbolo é um **B** ou um **D**. A frequência desse novo símbolo é a soma das frequências de seus dois símbolos constituintes. Sendo assim, a frequência de **BD** é 2.

Alfabeto: A,B,C,D  
Mensagem: ABACCDA  
Codificada : ?

Símbolo	Frequência
A	3
C	2
B	1
D	1

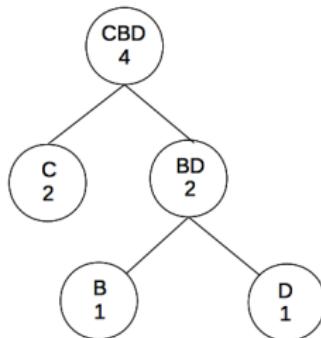


# Um exemplo: O algoritmo de HUFFMAN

- Agora existem três símbolos: **A (frequência 3)**, **C (frequência 2)** e **BD (frequência 2)**.
- Selecione novamente os dois símbolos com a menor frequência: **C** e **BD**. Os dois símbolos são, em seguida, combinados no símbolo isolado **CBD**.

Alfabeto: A,B,C,D  
Mensagem: ABACCDAA  
Codificada : ?

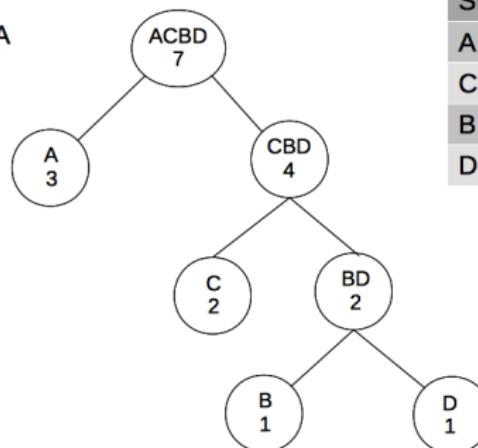
Símbolo	Frequência
A	3
C	2
B	1
D	1



# Um exemplo: O algoritmo de HUFFMAN

- Agora existem dois símbolos: **A** e **CDB**.
- Esses símbolos são combinados no símbolo isolado **ACBD**.

Alfabeto: A,B,C,D  
Mensagem: ABACCDA  
Codificada : ?

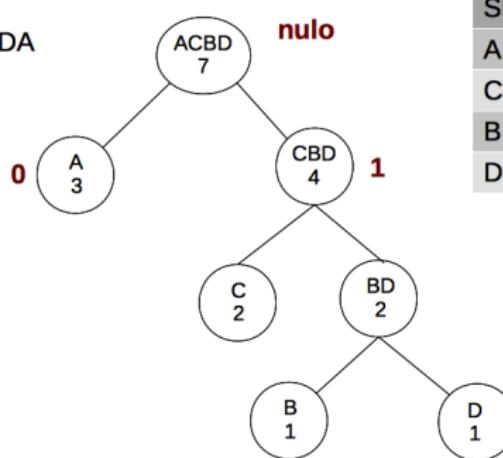


Símbolo	Frequência
A	3
C	2
B	1
D	1

# Um exemplo: O algoritmo de HUFFMAN

- A árvore está construída. Para atribuir os valores e montar a tabela proceda da seguinte forma:
  - O nó raiz (ACBD) recebe bit nulo, como seu código.
  - Os dois símbolos que formam **ACBD**, no caso o nó esquerdo **A** e o nó direito **CBD** recebem respectivamente os código **0** e **1**. Isso significa que se o símbolo **A** for encontrado o mesmo será substituído pelo código **0**. Se o código **1** for encontrado, será necessário analisar mais código para determinar qual o símbolo: **B**, **C** ou **D**.

Alfabeto: A,B,C,D  
Mensagem: ABACCDA  
Codificada : ?

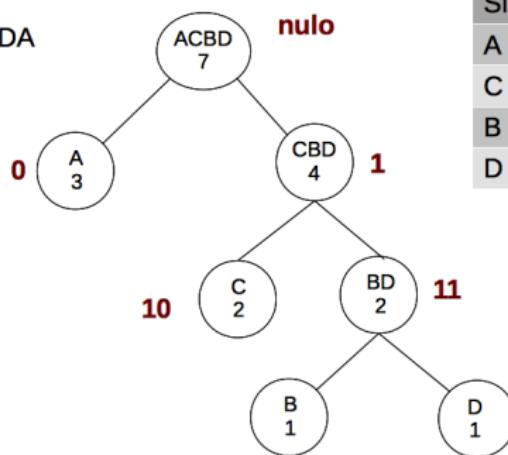


Símbolo	Frequência
A	3
C	2
B	1
D	1

# Um exemplo: O algoritmo de HUFFMAN

- O nó **C** que está a esquerda de **CBD** recebe o último bit **0**, concatenado com os bits que compõe o nó pai e o nó **BD** que está a direita recebe o último bit **1**, também concatenado ao bits que compõe o nó pai.

Alfabeto: A,B,C,D  
Mensagem: ABACCDA  
Codificada : ?

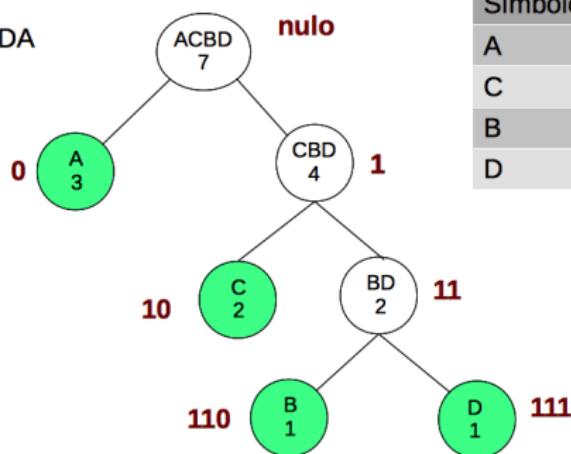


Símbolo	Frequência
A	3
C	2
B	1
D	1

# Um exemplo: O algoritmo de HUFFMAN

- Observe que todos os nós *folhas*, destacados na imagem, possuem os códigos que representam cada símbolo do alfabeto.

Alfabeto: A,B,C,D  
Mensagem: ABACCDA  
Codificada : ?



Símbolo	Frequência
A	3
C	2
B	1
D	1

- Uma vez que a árvore de huffman esteja construída, é possível iniciar a codificação, o algoritmo é simples. Leia um símbolo da string de entrada e escreva o código em bits que representa esse símbolo.
- A codificação **Huffman** é frequentemente utilizado no algoritmo de compactação do PKZIP e alguns codecs, tais como JPEG e MP3.

## 1 Introdução

## 2 Árvores

### 3 Árvores Binárias

- Operações em Árvores Binárias

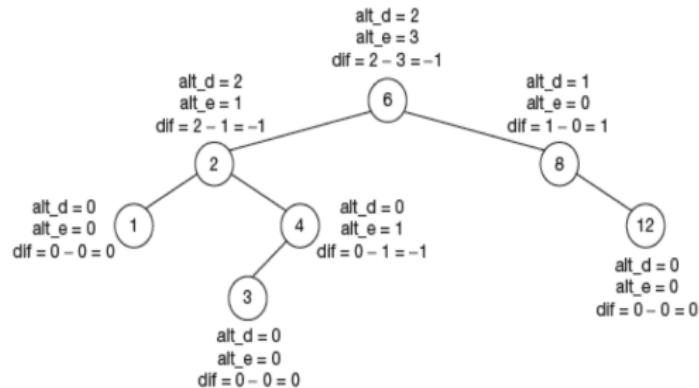
### 4 Aplicações de Árvores Binárias

- Árvore Binária para Remoção de Elementos duplicados
- Árvore Binária para Ordenação de Dados
- Algoritmo HUFFMAN

## 5 Árvore AVL

# Árvore AVL

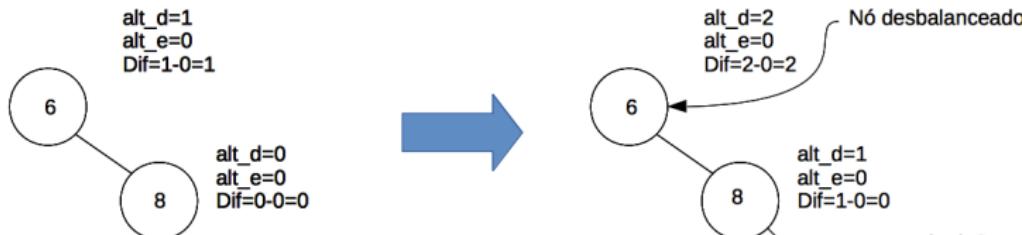
- Criada em 1962 por Adelson-Velsky e Landis, é uma árvore binária balanceada que obedece a todas as propriedades da árvore binária e em que cada nó apresenta diferença de altura entre as sub-árvore direita e esquerda de 1, 0 ou -1.



- Se a diferença de altura entre as subárvore de um nó é maior que **1** ou menor que **-1**, a árvore está *desbalanceada* e haverá uma **rotação**.
- A **rotação** é a operação básica para *balanceamento* de uma árvore AVL.
- Vamos analisar os tipos de rotação nos próximos slides.

# Árvore AVL

- Rotação simples para à esquerda ou Rotação RR. O nó 12 foi inserido na subárvore direita (8) da subárvore direita de (6).

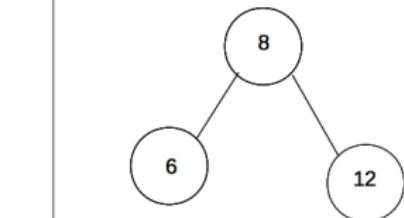
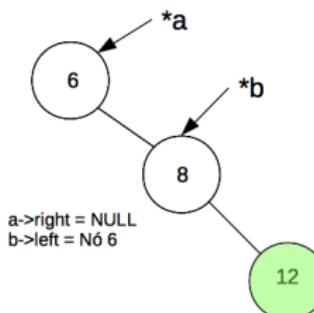


Árvore Balanceada

```
*avl_rotate_rightright( avl_node
_t *node ) {
    avl_node_t *a = node;
    avl_node_t *b = a->right;

    a->right = b->left;
    b->left = a;

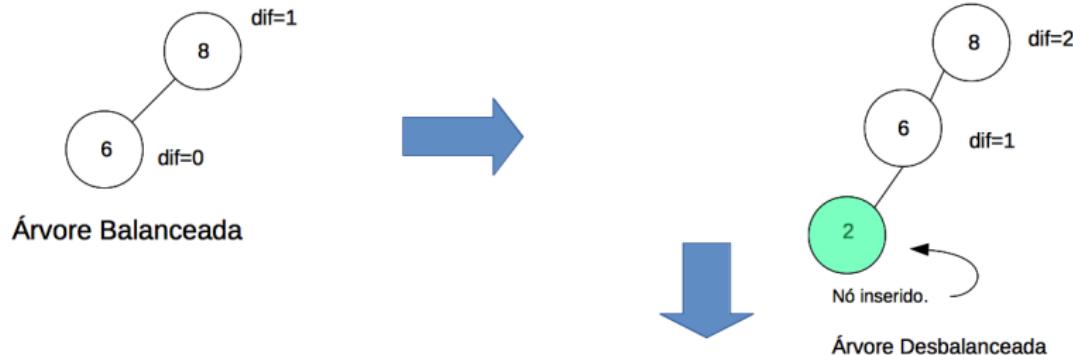
    return( b );
}
```



Árvore Balanceada

# Árvore AVL

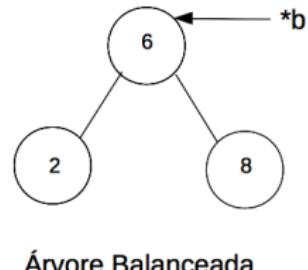
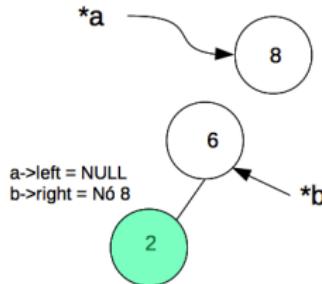
- Rotação simples para à direita ou Rotação LL. O nó 2 foi inserido na subárvore esquerda (6) da subárvore esquerda de (8).



```
*avl_rotate_leftleft( avl_node_t *node ) {
    avl_node_t *a = node;
    avl_node_t *b = a->left;

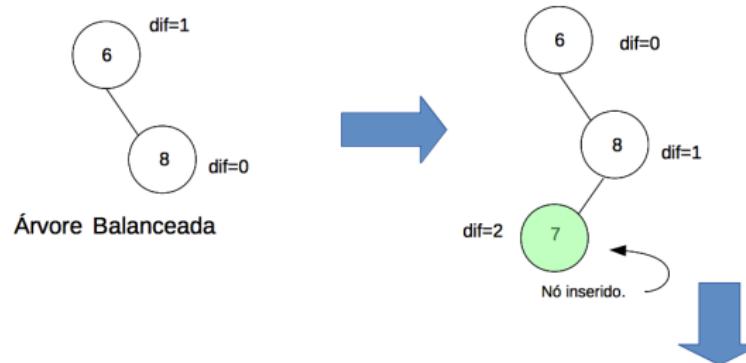
    a->left = b->right;
    b->right = a;

    return( b );
}
```



# Árvore AVL

- Rotação dupla para à esquerda ou Rotação RL. O nó 7 foi inserido na subárvore esquerda (8) da subárvore direita de (6). O nó 7 deve ser selecionado para ser o *raiz* da árvore.



```
*avl_rotate_rightleft( avl_node_t *node ) {
    avl_node_t *a = node;
    avl_node_t *b = a->right;
    avl_node_t *c = b->left;

    a->right = c->left;
    b->left = c->right;
    c->right = b;
    c->left = a;

    return( c );
}
```

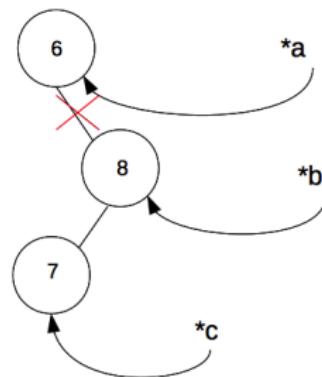
# Árvore AVL

- Ponteiro direito de **a** recebe o ponteiro esquerdo de **c**.

```
*avl_rotate_rightleft(avl_node_t*node) {
    avl_node_t *a = node;
    avl_node_t *b = a->right;
    avl_node_t *c = b->left;

    a->right = c->left;
    b->left = c->right;
    c->right = b;
    c->left = a;

    return(c);
}
```



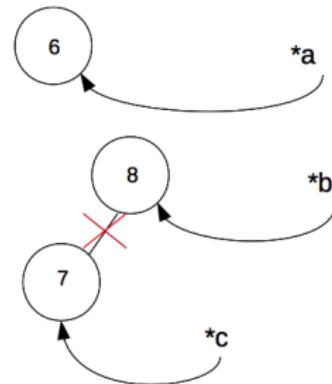
# Árvore AVL

- Ponteiro esquerdo de **b** recebe o ponteiro direito de **c**.

```
*avl_rotate_rightleft(avl_node_t*node) {
    avl_node_t *a = node;
    avl_node_t *b = a->right;
    avl_node_t *c = b->left;

    a->right = c->left;
    b->left = c->right;
    c->right = b;
    c->left = a;

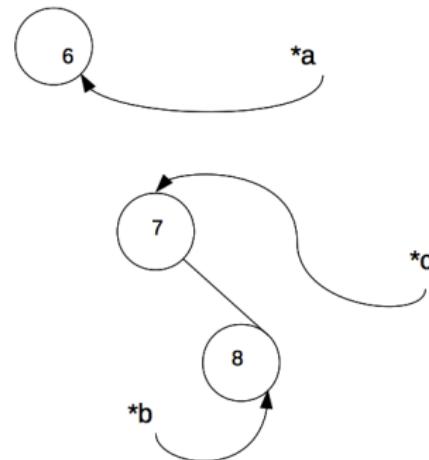
    return(c);
}
```



# Árvore AVL

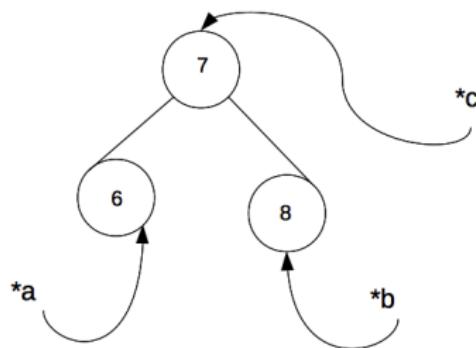
- Ponteiro direito de **c** recebe o **b**.

```
*avl_rotate_rightleft( avl_node_t *node ) {  
    avl_node_t *a = node;  
    avl_node_t *b = a->right;  
    avl_node_t *c = b->left;  
  
    a->right = c->left;  
    b->left = c->right;  
    c->right = b;  
    c->left = a;  
  
    return( c );  
}
```



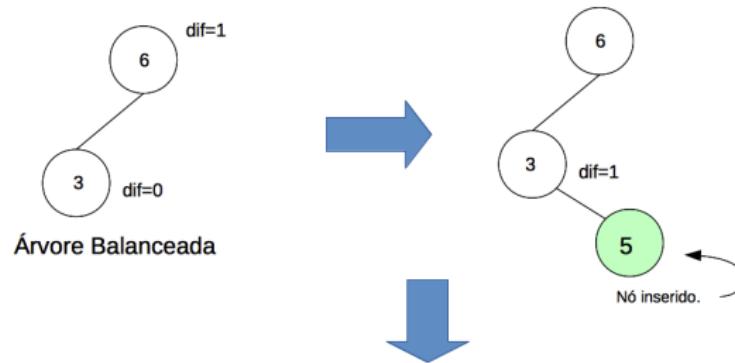
- Ponteiro esquerdo de **c** recebe **a**.

```
*avl_rotate_rightleft( avl_node_t *node ) {  
    avl_node_t *a = node;  
    avl_node_t *b = a->right;  
    avl_node_t *c = b->left;  
  
    a->right = c->left;  
    b->left = c->right;  
    c->right = b;  
    c->left = a;  
  
    return( c );  
}
```

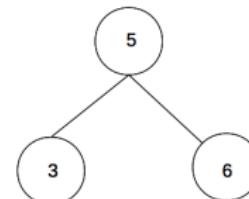


# Árvore AVL

- Rotação dupla para à direita ou Rotação LR. O nó 5 foi inserido na subárvore direita (3) da subárvore esquerda de (6). O nó 5 deve ser selecionado para ser o *raiz* da árvore.



```
*avl_rotate_leftright( avl_node_t *node ) {  
    avl_node_t *a = node;  
    avl_node_t *b = a->left;  
    avl_node_t *c = b->right;  
  
    a->left = c->right;  
    b->right = c->left;  
    c->left = b;  
    c->right = a;  
  
    return( c );  
}
```



- A inserção nas árvores AVL ocorre de maneira análoga as árvores binárias.
- A cada inserção realizada, um fator de balanceamento é calculado para decidir se deve ou não ocorrer a rotação da árvore AVL. Vamos analisar o código que calcula o **fator de balanceamento**.

```
int avl_balance_factor( avl_node_t *node ) {  
    int bf = 0;  
  
    if( node->left)  
        bf = bf + avl_node_height( node->left );  
    if( node->right)  
        bf = bf - avl_node_height( node->right );  
  
    return bf ;  
}
```

# Árvore AVL Inserção

- A função `avl_node_height` verifica a altura dos nós em relação a raiz.
- Essa função retorna a **maior** altura encontrada.
- Uma árvore AVL estará desbalanceada quando a diferença entre a *altura esquerda* e a *altura direita* for maior ou igual a 2.

```
int avl_node_height( avl_node_t *node ) {
    int height_left = 0;
    int height_right = 0;

    if( node->left )
        height_left = avl_node_height( node->left );
    if( node->right )
        height_right = avl_node_height( node->right );

    return height_right > height_left ?
        ++height_right :
        ++height_left;
}
```

- A função `avl_balance_factor` retorna valores **positivos** conforme a subárvore à esquerda recebe um novo nó e retorna valores **negativos** conforme a subárvore à direita é incrementada.
- Um valor de  $bf \geq 2$  rotaciona a árvore AVL para direita / rotação dupla à direita, dependendo da situação.
- Um valor de  $bf \leq -2$  rotaciona a árvore AVL para esquera / rotação dupla à esquera, dependendo da situação.
- Analise a função `*avl_balance_node` no código de exemplo.

```
int avl_balance_factor( avl_node_t *node ){
    int bf = 0;
    if( node->left)
        bf = bf + avl_node_height(node->left);
    if( node->right)
        bf = bf - avl_node_height(node->right);
    return bf ;
}
```

- A informação existente entre a altura da árvore ( $h$ ) e o número de nós ( $n$ ) de uma **árvore binária** ou **árvore AVL** é fundamental para análise da complexidade.
- Uma árvore binária completa com  $n > 0$  nós possui altura mínima  $h = 1 + \lfloor \log n \rfloor$ .
- A operação de busca em um árvore binária é igual ao número de nós existente no caminho desde a raiz até o nó procurado. Em uma árvore binária, no **pior caso**, esse nó encontra-se a um distância  $O(n)$  da raiz da árvore (árvore desbalanceada à direita ou à esquerda).
- Conclui-se que a complexidade de busca corresponde à altura da árvore. No **melhor caso**, uma árvore binária completa, o tempo de busca é  $O(\log n)$ .

Obrigado pela atenção!!!  
[thiago.tavares@ifsuldeminas.edu.br](mailto:thiago.tavares@ifsuldeminas.edu.br)



-  ASCENCIO, A.; CAMPOS, E. de. *Fundamentos da programação de computadores: algoritmos, Pascal, C/C++ e Java*. Pearson Prentice Hall, 2008. ISBN 9788576051480. Disponível em: <<https://books.google.com.br/books?id=p-mTPgAACAAJ>>.
-  C: A Reference Manual. Pearson Education, 2007. ISBN 9788131714409. Disponível em: <<https://books.google.com.br/books?id=Wt2NEypdGNIC>>.
-  DAMAS, L. *LINGUAGEM C*. LTC. ISBN 9788521615194. Disponível em: <<https://books.google.com.br/books?id=22-vPgAACAAJ>>.
-  FEOFILOFF, P. *Algoritmos Em Linguagem C*. CAMPUS - RJ, 2009. ISBN 9788535232493. Disponível em: <<http://books.google.com.br/books?id=LfUQai78VQgC>>.
-  KERNIGHAN, B.; RITCHIE, D. *C: a linguagem de programação padrão ANSI*. Campus, 1989. ISBN 9788570015860. Disponível em: <<https://books.google.com.br/books?id=aVWrQwAACAAJ>>.

-  LOPES, A.; GARCIA, G. *Introdução à programação: 500 algoritmos resolvidos*. Campus, 2002. ISBN 9788535210194. Disponível em: <<https://books.google.com.br/books?id=Rd-LPgAACAAJ>>.
-  MIZRAHI, V. *Treinamento em linguagem C*. Pearson Prentice Hall, 2008. ISBN 9788576051916. Disponível em: <<https://books.google.com.br/books?id=7xt7PgAACAAJ>>.
-  SCHILDT, H.; MAYER, R. *C completo e total*. Makron, 1997. ISBN 9788534605953. Disponível em: <<https://books.google.com.br/books?id=PbI0AAAACAAJ>>.