

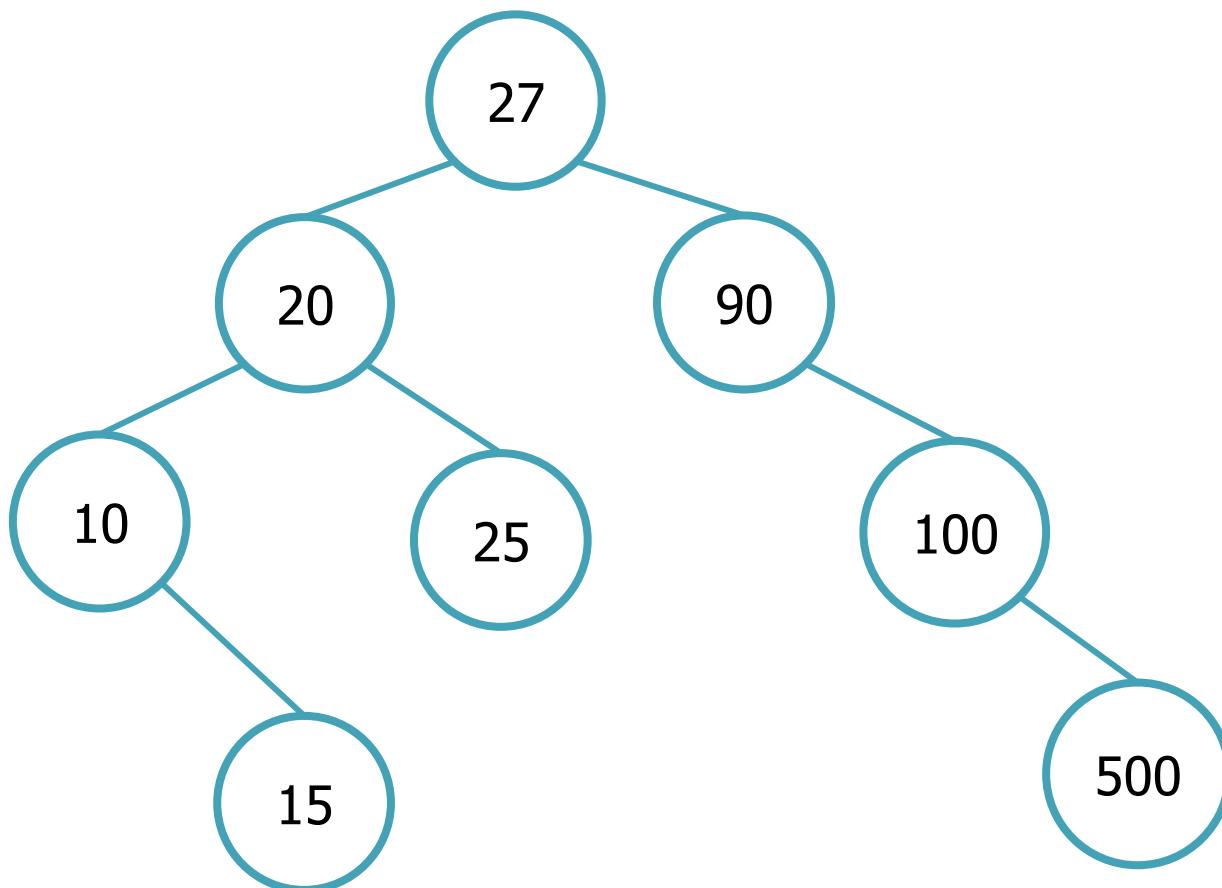


SCC-223 Estruturas de Dados I

Árvore Binária de Busca

Profa. Elaine Parros Machado de Sousa

Exemplo





Árvore Binária de Busca - ABB (*Binary Search Tree* – BST)

- Os nós pertencentes à **subárvore esquerda** possuem **valores menores** do que o valor associado ao **nó raiz**
- Os nós pertencentes à **subárvore direita** possuem **valores maiores** do que o valor associado ao **nó raiz**



Definição

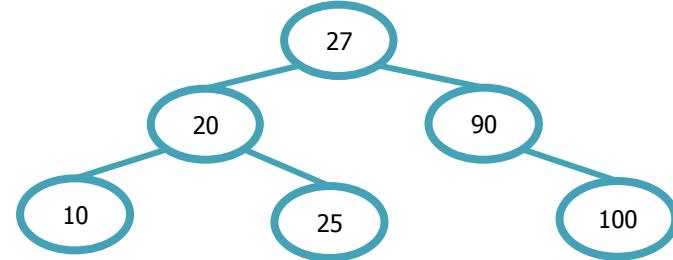
- Um **Árvore Binária de Busca (ABB)** possui as seguintes propriedades:
 - Seja $S = (s_1, s_2, \dots, s_n)$ o conjunto de chaves na árvore T , tal que:
 - $s_1 < s_2 < \dots < s_n$
 - cada nó v_j de T está associado a uma chave s_j distinta de S , que pode ser consultada por $r(v_j) = s_j$



Definição

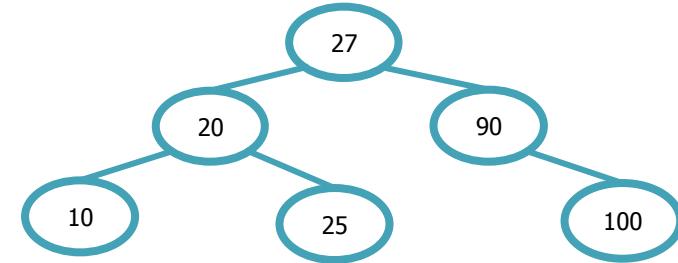
- Um **Árvore Binária de Busca (ABB)** possui as seguintes propriedades:
 - ...
 - Dado um nó v de T :
 - Se v_j pertence à subárvore esquerda de T , então $r(v_j) < r(v)$
 - se v_j pertence à subárvore direita de T , então $r(v_j) > r(v)$

Questões



- Como obter as chaves (armazenadas numa ABB) em **ordem crescente**?
 - percurso **em-ordem**
- E se invertermos as propriedades descritas na definição anterior => subárvore esquerda de um nó com valores maiores e a subárvore direita com valores menores?
 - percurso **em-ordem** resultaria nas chaves em **ordem decrescente**

Questões



- Um conjunto de chaves sempre gera a mesma ABB?
 - **NÃO!** o resultado depende da sequência de inserção dos dados
- Qual a maior utilidade de ABB?
 - **busca binária => $O(\log_2 n)$**
- E por que usar ABB e não uma lista sequencial estática ordenada?



ABB versus Listas

- O tempo de busca é estimado pelo **número de comparações entre chaves**.
- Em **listas** de n elementos (chaves), temos:
 - sequenciais ordenadas (Array) => $O(\log_2 n)$
 - encadeadas dinâmicas => $O(n)$
- As **ABB** são a alternativa que combina vantagens:
 - são dinâmicas
 - permitem a busca binária => $O(\log_2 n)$



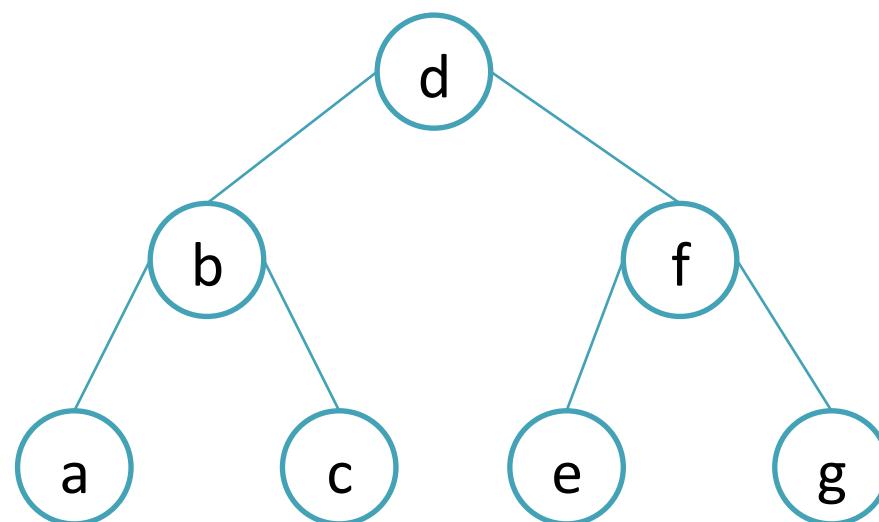
TAD - ABB

- Principais implementações
 - Sequencial estática
 - Dinâmica
- Principais operações
 - Criar/Apagar árvore
 - **Inserir chave**
 - **Remover chave**
 - **Buscar chave**
 - Percorrer/imprimir árvore
 - Verificar se árvore está vazia
 - Calcular/retornar altura
 -

TAD – ABB

Implementação Sequencial Estática

- Armazenar os nós, por nível, em um *array*

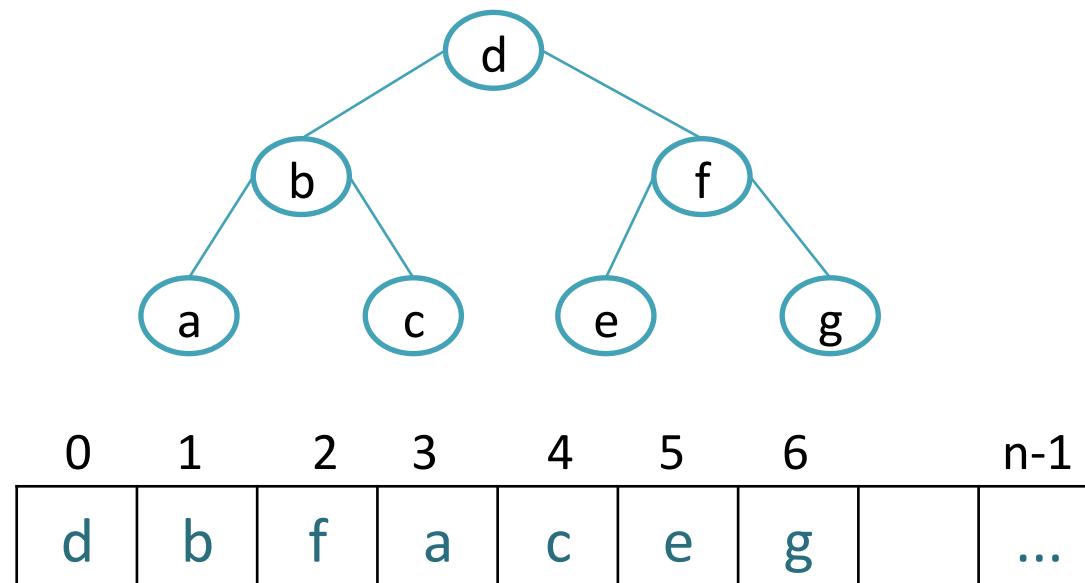


0	1	2	3	4	5	6	$n-1$
d	b	f	a	c	e	g	...

TAD – ABB

Implementação Sequencial Estática

- Para um vetor indexado a partir da posição 0 , se um nó está na posição i , seus filhos estão nas posições
 - $2i + 1$: filho da esquerda
 - $2i + 2$: filho da direita





TAD – ABB

Implementação Sequencial Estática

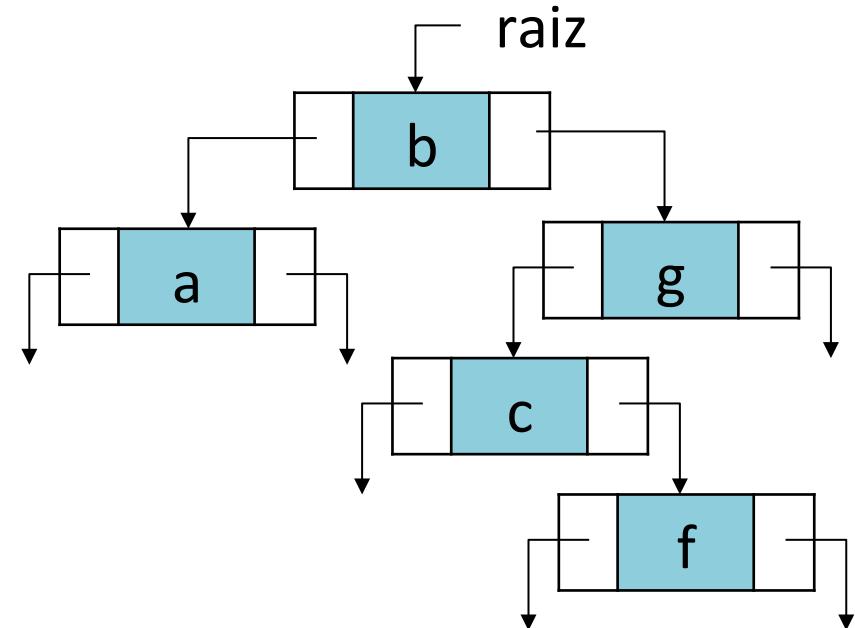
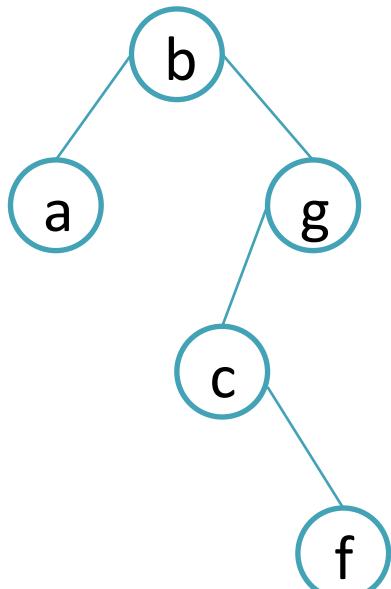
- Vantagem:
 - espaço só p/ armazenar conteúdo => ligações implícitas
- Desvantagens:
 - requer estimativa do número de nós
 - espaços vagos se árvore não está completa e cheia

TAD – ABB

Implementação Dinâmica

- Cada nó é do tipo:

esq ← ─ ┌ ┘ item ┘ └ ─ dir



TAD – ABB

Implementação Dinâmica

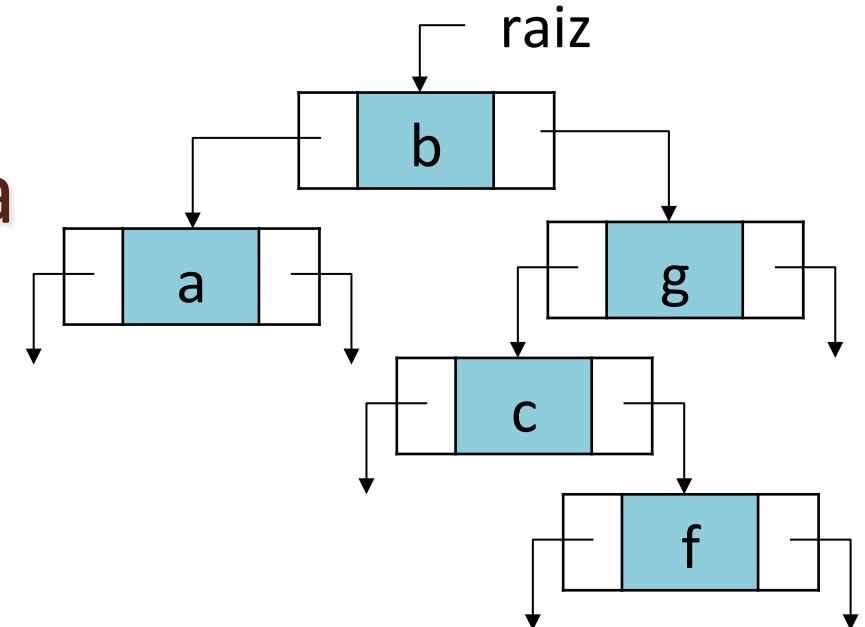
```
//abb.c

typedef struct no NO;

struct no {
    ITEM item;
    NO *esq;
    NO *dir;
};

struct arb {
    NO *raiz;
    // int altura;
    ...
};

...
```



```
//abb.h

typedef struct arb ARV;
...
ARV* abb_criar();
...
```

```
//main.c

ARV *T;
T= abb_criar();
...
```

Operações do TAD ABB

```
// função com protótipo no .h
ARV *abb_criar(void) {
    ARV *r = (ARV *) malloc(sizeof(ARV));
    if (r != NULL) {
        r->raiz = NULL;
    }
    return (r);
}
```

Operações do TAD ABB

```
void abb_apagar(ARV **T) { // função com protótipo no .h
    if (*T != NULL) {
        abb_apagar_nos(&(*T)->raiz);
        free(*T);
        *T = NULL;
    }
}

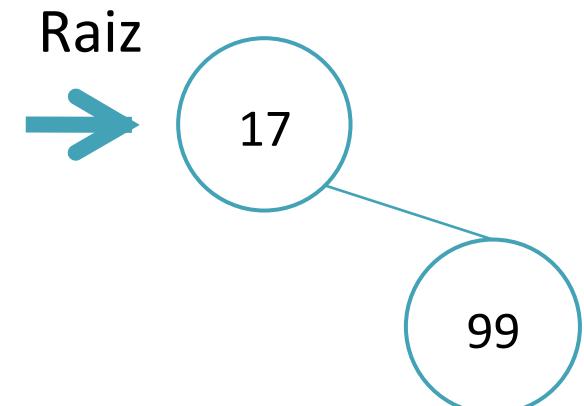
void abb_apagar_nos(NO **raiz) {// função interna ao TAD
    if (*raiz != NULL) {
        abb_apagar_nos(&(*raiz)->esq);
        abb_apagar_nos(&(*raiz)->dir);
        free(*raiz);
        *raiz = NULL;
    }
}
```

Inserção em ABB

- Passos do algoritmo de inserção da chave k
 - comece a pesquisa a partir do nó raiz de T
 - para cada nó raiz de uma (sub)árvore, compare:
 - se k é menor que a chave do nó raiz => vai para subárvore esquerda
 - Se k é maior que a chave do nó raiz => vai para subárvore direita
 - quando um ponteiro (filho esquerdo/direito de um nó raiz) nulo é atingido, coloque um novo nó (com a chave k) como filho (esquerdo/direito) do nó raiz da subárvore

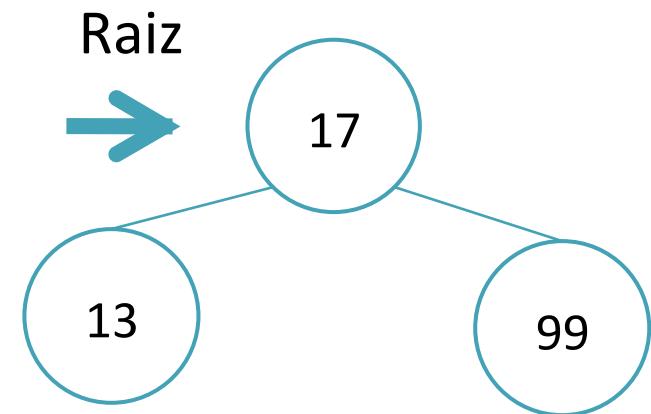
Inserção em ABB - Exemplo

- Considere a inserção do conjunto de números, na sequência
17, 99, 13, 1, 3, 100, 400
- No início a ABB está vazia!
- Inserção do **17** => nó-raiz
- Inserção do **99**
 - inicia-se a busca na raiz
 - compara-se 99 com 17
 - **99 > 17** => 99 deve ser inserido na **subárvore direita** do nó 17
 - subárvore direita, inicialmente, nula



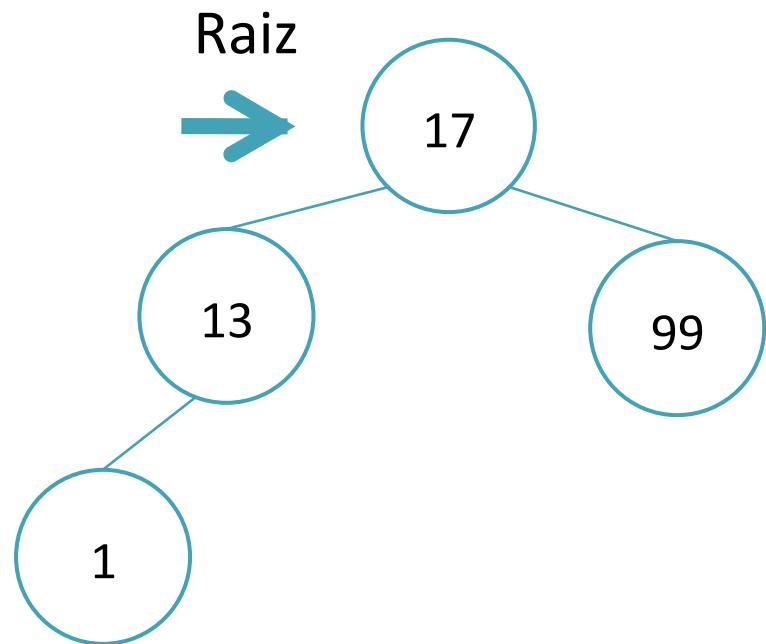
Inserção em ABB - Exemplo

- Inserção do **13**
 - inicia-se busca na raiz
 - compara-se 13 com 17
 - **13 < 17 =>** 13 deve ser inserido na **subárvore esquerda** do nó 17
 - subárvore esquerda, inicialmente, nula



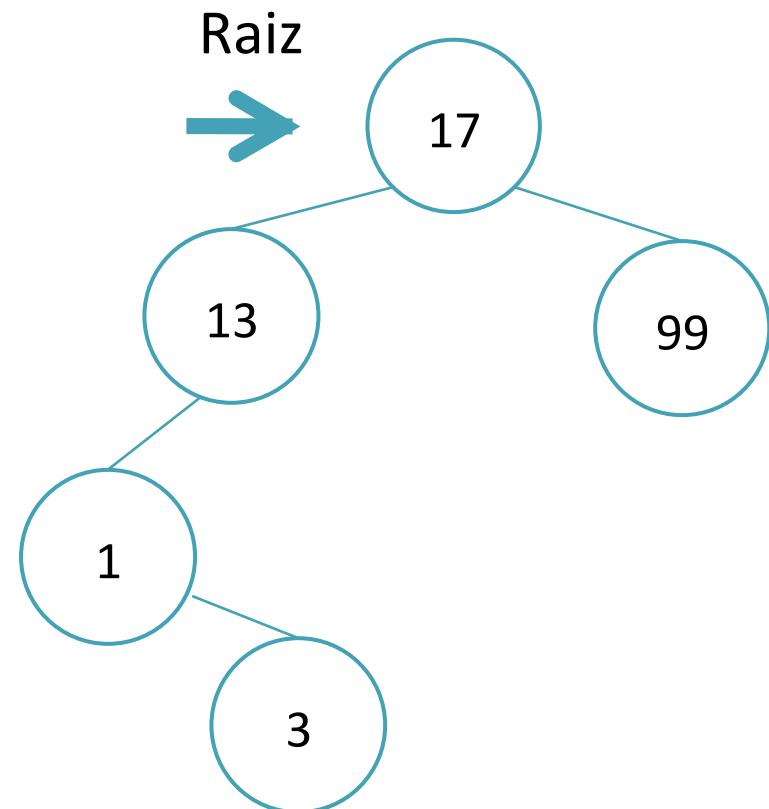
Inserção em ABB - Exemplo

- Inserção do **1**
 - inicia-se busca na raiz
 - compara-se 1 com 17
 - $1 < 17 \Rightarrow 1$ deve ser inserido na **subárvore esquerda**
 - na subárvore esquerda, compara-se 1 com 13
 - $1 < 13 \Rightarrow 1$ deve ser inserido na **subárvore esquerda**



Inserção em ABB - Exemplo

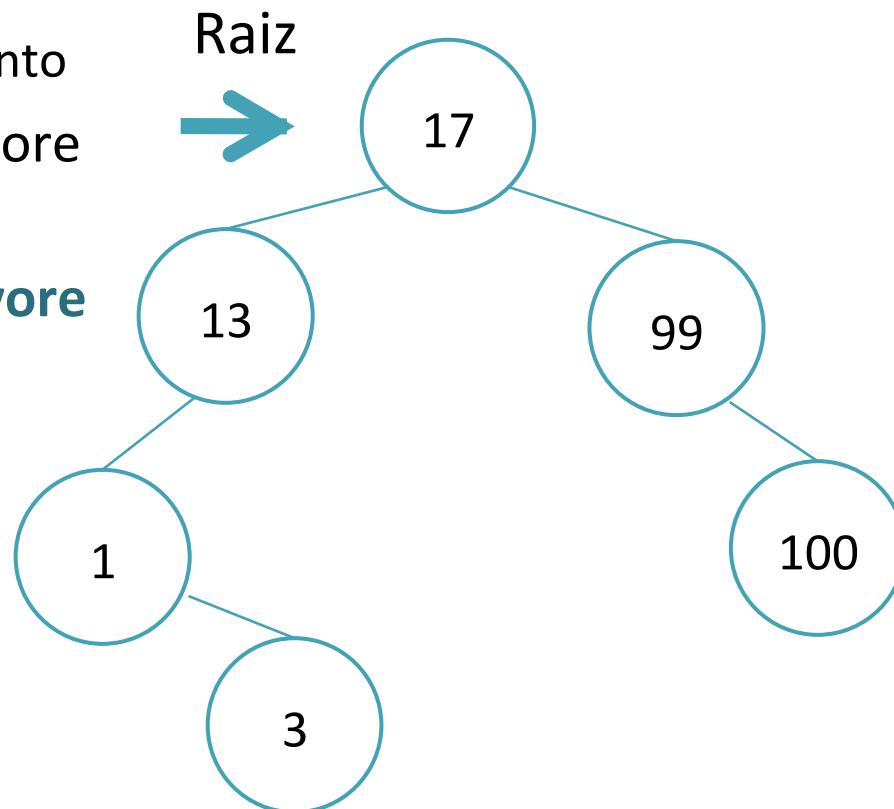
- Inserção do 3
 - repete-se o procedimento
 - $3 < 17 \Rightarrow$ subárvore esquerda
 - $3 < 13 \Rightarrow$ subárvore esquerda
 - $3 > 1 \Rightarrow$ subárvore direita



Inserção em ABB - Exemplo

- Inserção do **100**

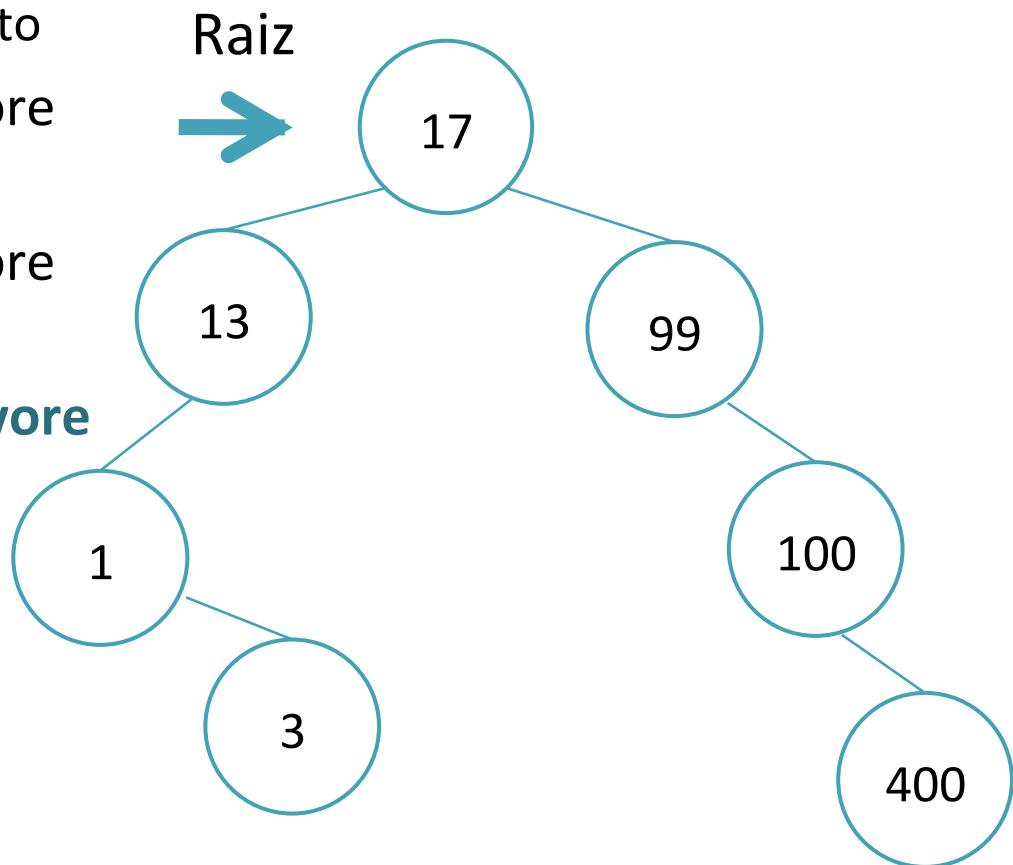
- repete-se o procedimento
 - $100 > 17 \Rightarrow$ subárvore direita
 - **$100 > 99 \Rightarrow$ subárvore direita**



Inserção em ABB - Exemplo

- Inserção do **400**

- repete-se o procedimento
 - $400 > 17 \Rightarrow$ subárvore direita
 - $400 > 99 \Rightarrow$ subárvore direita
 - **$400 > 100 \Rightarrow$ subárvore direita**



```
// função com protótipo no .h
boolean abb_inserir(ARV *T, ITEM item) {
    if(T == NULL) return(ERRO);
    if(abb_vazia(T)){
        T->raiz = (NO *) malloc(sizeof(NO));
        if (T->raiz != NULL) {
            T->raiz->item = item;
            T->raiz->esq = NULL;
            T->raiz->dir = NULL;
            return (TRUE);
        }
    }
    else
        return (abb_inserir_no(T->raiz, item));
}
```

```
// função de apoio - interna no .c do TAD
boolean abb_inserir_no(N0 *raiz, ITEM item) {
    if (item < raiz->item) {      // no .h: #define ITEM int
        if(raiz->esq != NULL)
            return (abb_inserir_no(raiz->esq, item));
        else
            return (abb_inserir_filho(FILHO_ESQ, raiz, item)!=NULL);
    }
    else {
        if(item > raiz->item) {
            if(raiz->dir != NULL)
                return abb_inserir_no(raiz->dir, item);
            else
                return (abb_inserir_filho(FILHO_DIR, raiz, item)!=NULL);
        }
        else return (FALSE);
    }
}
```

```
// no .h: #define FILHO_ESQ 0
//           #define FILHO_DIR 1
// função de apoio - interna ao .c do TAD
NO *abb_inserir_filho(int filho, NO *no, ITEM item) {
    NO *pnovo = (NO *) malloc(sizeof(NO));
    if (pnovo != NULL) {
        pnovo->dir = NULL;
        pnovo->esq = NULL;
        pnovo->item = item;
        if (filho == FILHO_ESQ)
            no->esq = pnovo;
        else
            no->dir = pnovo;
    }
    return (pnovo);
}
```



Exercício

- Desenvolva uma função iterativa para inserção em ABB



Busca (Pesquisa) em ABB

- Passos do algoritmo de busca
 - comece a busca a partir do nó raiz
 - para cada nó raiz de uma subárvore compare:
 - se a chave procurada é menor que a chave no nó raiz => continua pela subárvore esquerda
 - se a chave procurada é maior que a chave no nó raiz => subárvore direita
 - Caso o nó contendo a chave pesquisada seja encontrado, retorne o “item” do nó pesquisado, caso contrário retorne nulo

Busca (Pesquisa) em ABB

```
// função com protótipo no .h
ITEM* abb_buscar(ARV *T, int chave) {
    /*considerando chave como int */
    if(T == NULL) return NULL;
    if(abb_vazia(T))
        return NULL;
    else {
        NO* no = abb_buscar_no(T->raiz, chave);
        return (&(no->item));
    }
}
```

```
// função de apoio - interna no .c do TAD

NO* abb_buscar_no (NO* raiz, int chave) {
    if(raiz == NULL){
        return NULL;
    }
    else {
        if (chave == raiz->item){
            return (raiz);
        }
        else {
            if (chave < raiz->item){
                return (abb_buscar_no(raiz->esq, chave));
            }
            else{
                return (abb_buscar_no(raiz->dir, chave));
            }
        }
    }
}
```

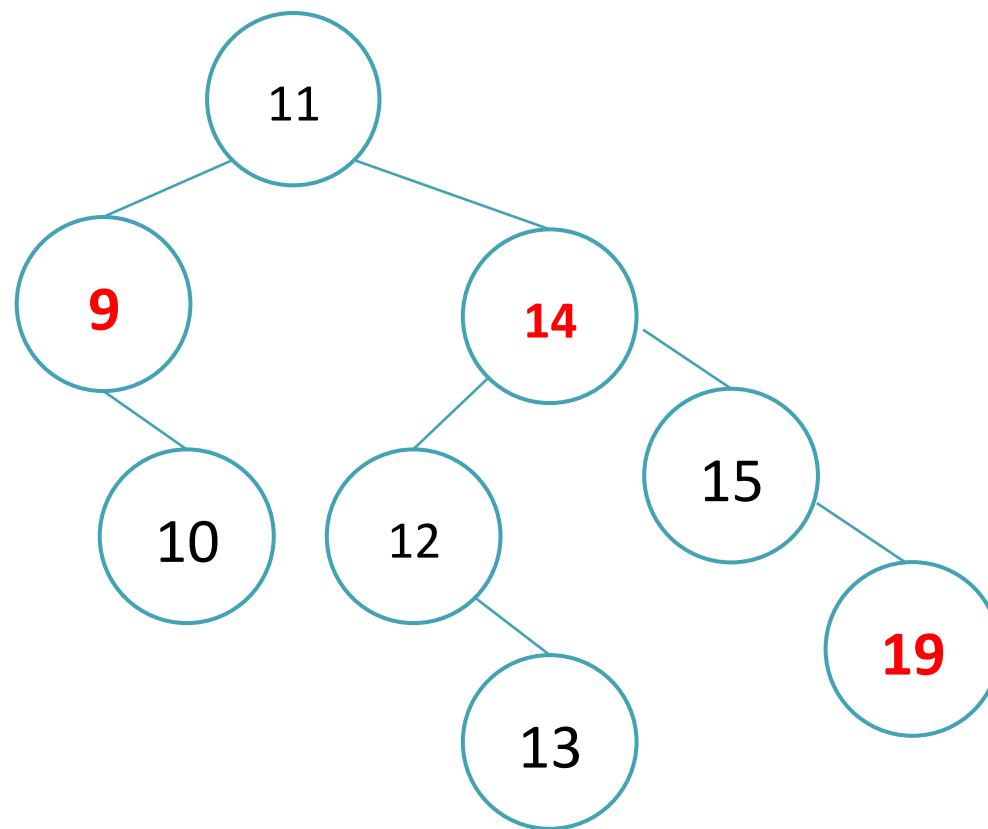


Exercício

- Desenvolva uma função iterativa para busca em ABB

Remoção em ABB

3 Casos...





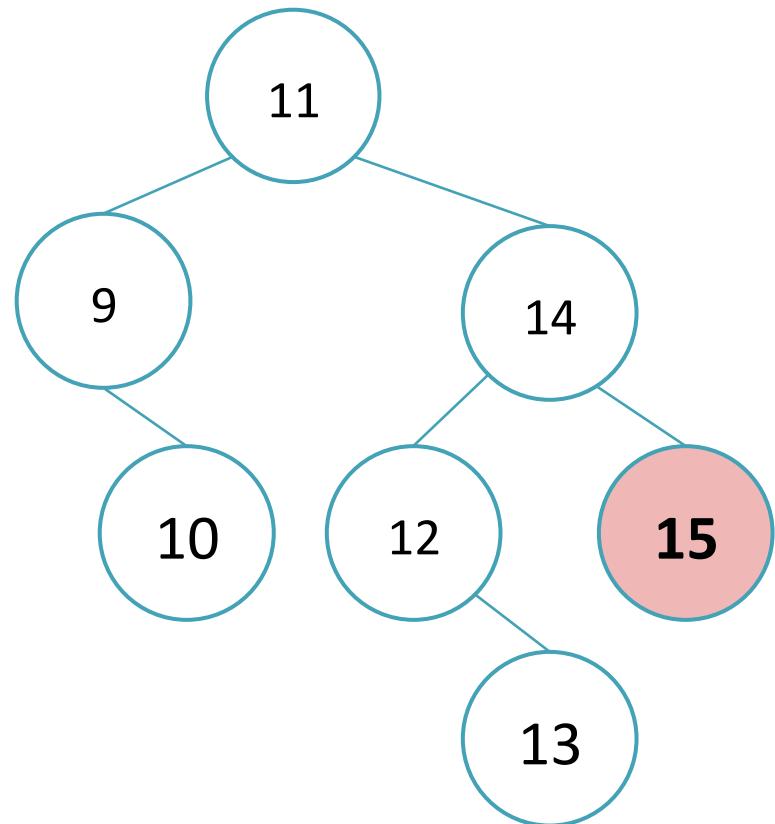
Remoção em ABB

- Casos a serem considerados no algoritmo de remoção de nó de uma ABB
- **Caso 1:** o nó é **folha**
 - nó pode ser simplesmente removido
- **Caso 2:** o nó possui **uma subárvore** (esq/dir)
 - nó raiz da subárvore (esq/dir) “ocupa” o lugar do nó removido
- **Caso 3:** o nó possui **duas subárvores**
 - nó contendo o menor valor da subárvore direita pode “ocupar” o lugar do nó removido, OU
 - o maior valor da sub-árvore esquerda pode “ocupar” o lugar do nó removido.

Remoção em ABB

- **CASO 1**

- remoção do **15**
- nó pode ser removido diretamente => não requer ajustes posteriores

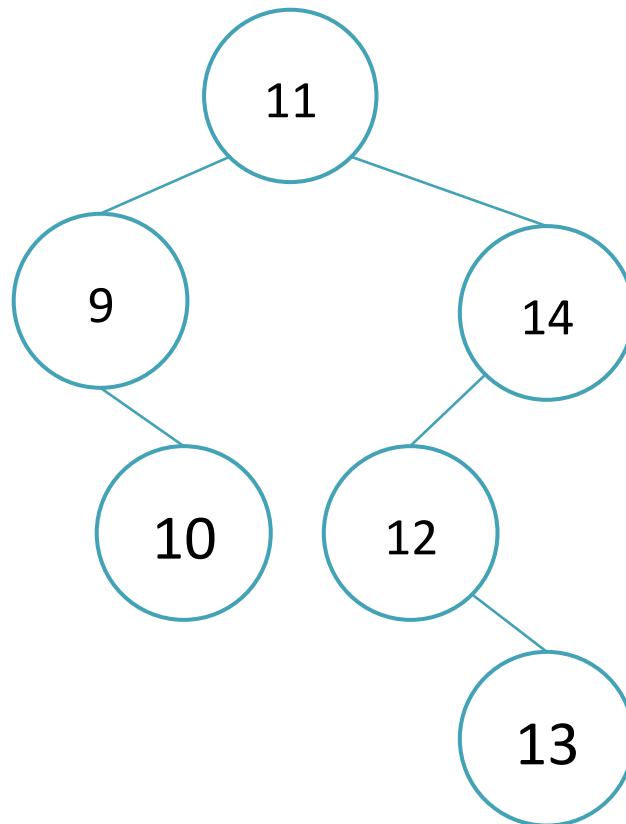


Remoção em ABB

- **CASO 1**

- remoção do **15**
- nó pode ser removido sem problema => não requer ajustes posteriores

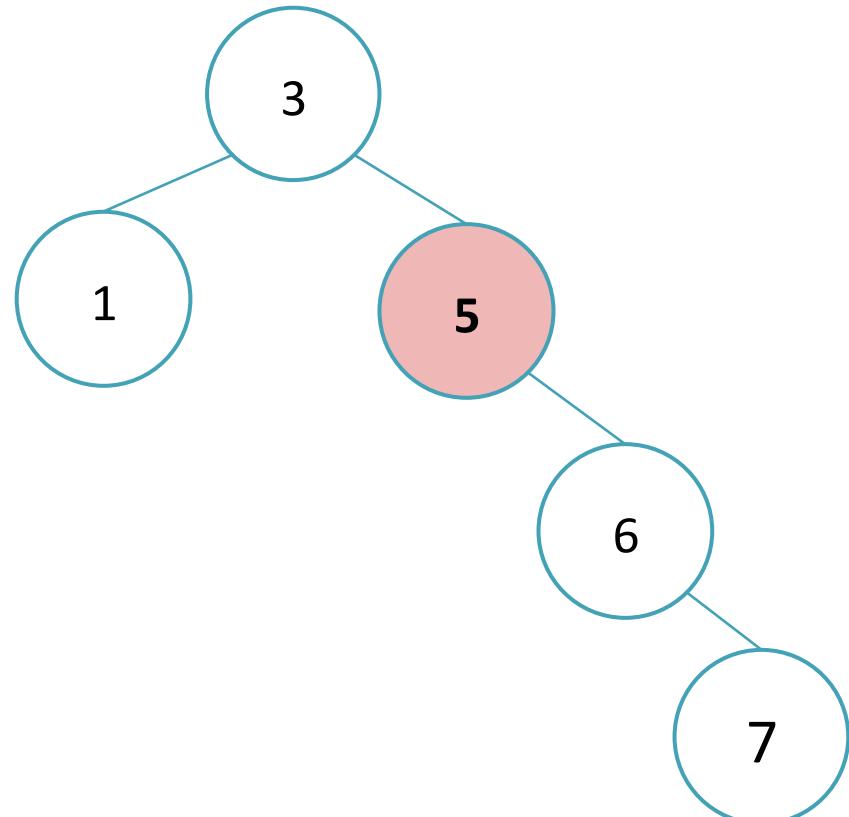
➤ idem para os nós **10** e **13**



Remoção em ABB

- **CASO 2**

- remoção do **5**
- como ele possui somente a subárvore direita, o nó contendo a chave 6 pode “ocupar” o lugar do nó removido

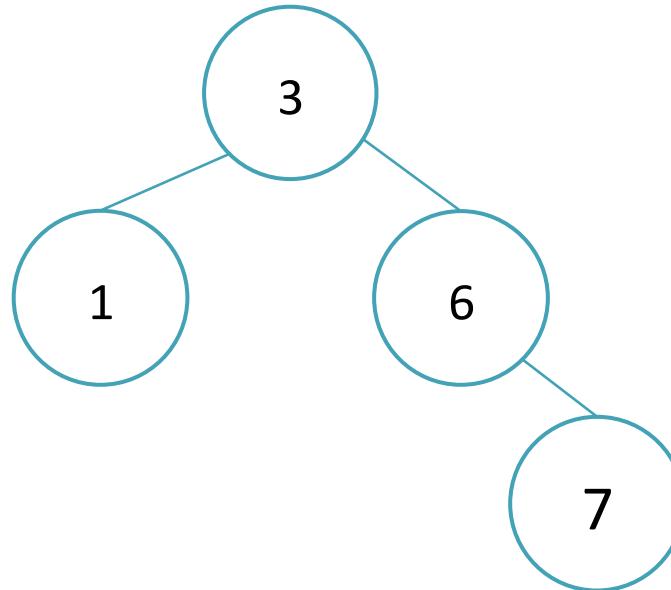


Remoção em ABB

- **CASO 2**

- remoção do **5**
- como ele possui somente a subárvore direita, o nó contendo a chave 6 pode “ocupar” o lugar do nó removido

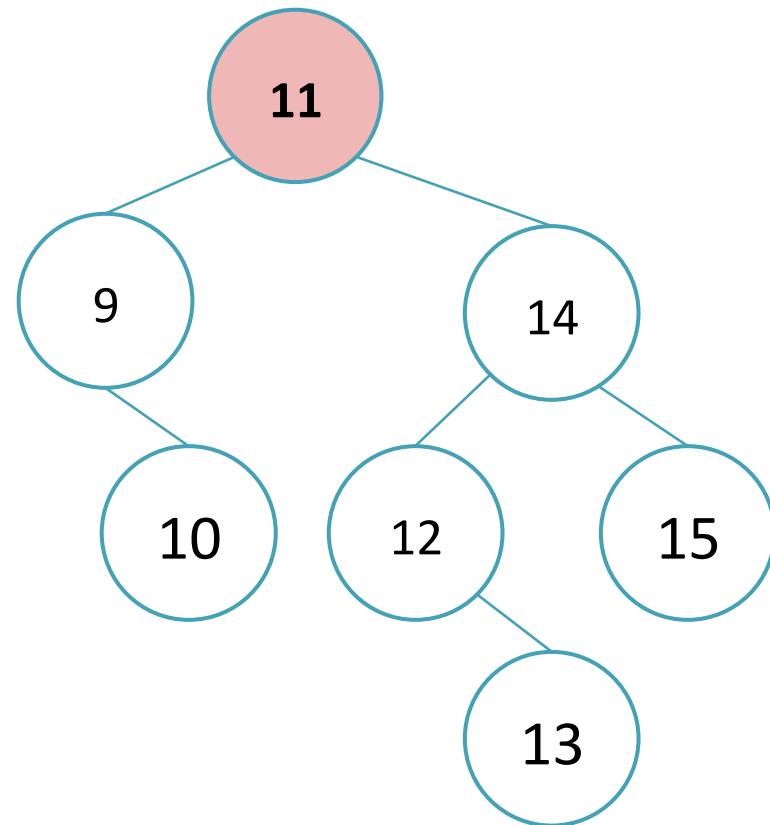
➤ procedimento análogo para nós com apenas a subárvore esquerda



Remoção em ABB

- **CASO 3**

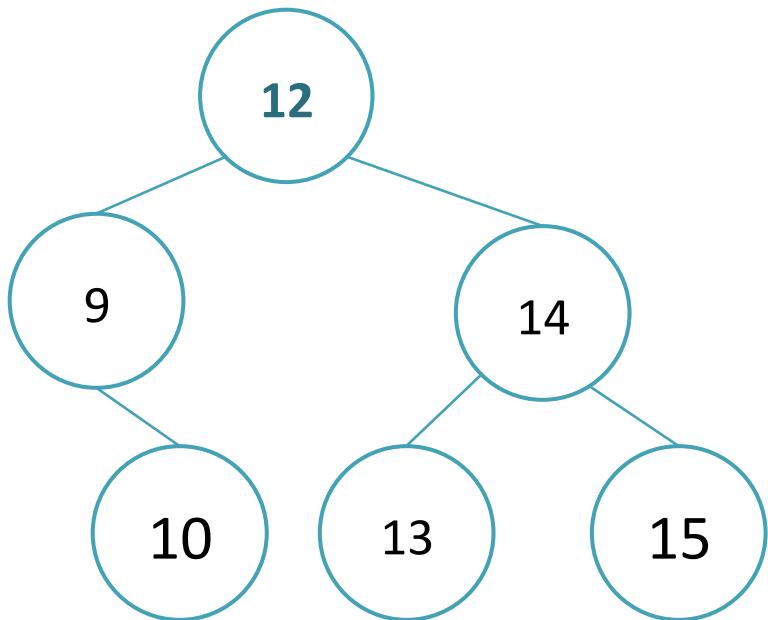
- remoção do **11**
- neste caso, há 2 opções:
 - ① nó com chave **10** pode “ocupar” o lugar do nó-raiz, ou
 - ② nó com chave **12** pode “ocupar” o lugar do nó-raiz



Remoção em ABB

- **CASO 3**

- remoção do **11**
- exemplo:
 - nó com chave **12** ocupa o lugar do 11



Remoção em ABB

- Função de remoção com protótipo no .h do TAD

```
boolean abb_remover(ARV *T, int chave){  
    if (T != NULL)  
        return (abb_remover_no(&T->raiz, chave));  
    return(FALSE);}
```

```
boolean abb_remover_no(NO **raiz, int chave){ // função interna no .c

    if ((*raiz) == NULL) return (FALSE); // chave não existe

    if ((*raiz)->item == chave){ //chave encontrada
        NO *aux = (*raiz);
        // CASOS 1 e 2
        if ((*raiz)->esq == NULL) {
            (*raiz) = (*raiz)->dir;
            free(aux);
            aux = NULL;
        }
        else if ((*raiz)->dir == NULL) {
            (*raiz) = (*raiz)->esq;
            free(aux);
            aux = NULL;
        }
        else // CASO 3 - troca {.....}
    }
    else if ((*raiz)->item > chave)
        abb_remover_no (&(*raiz)->esq, chave);
    else
        abb_remover_no (&(*raiz)->dir, chave);
return (TRUE);
}
```



Exercício

- Implemente a funcionalidade completa para remoção em ABB



Custo Computacional da ABB

- Busca
- Inserção
- Remoção



Custo da Busca em ABB

- Pior caso
 - Número de passos é determinado pela altura da árvore
 - **Árvore degenerada** possui altura igual a n
- Altura da árvore depende da sequência de inserção das chaves
 - o quê acontece se uma sequência ordenada de chaves é inserida?
- Busca é eficiente se árvore está “razoavelmente balanceada”
 - $O(\log_2 n)$



Custo da Inserção em ABB

- A inserção requer uma busca pelo lugar da chave, portanto, com custo de uma busca qualquer => **tempo proporcional à altura da árvore.**
- O custo da inserção, após a localização do lugar, é constante (não depende do número de nós).
=> **complexidade equivalente à da busca.**



Custo da Remoção em ABB

- A remoção requer uma busca pela chave do nó a ser removido, portanto, com custo de uma busca qualquer => **tempo proporcional à altura da árvore.**
- O custo da remoção, após a localização do nó dependerá de 2 fatores:
 - do caso em que se enquadra a remoção: se o nó tem 0, 1 ou 2 sub-árvores;
 - se 0 ou 1 filho => custo é constante.
 - da posição do nó na árvore, caso tenha 2 sub-árvores
 - quanto mais próximo do último nível, menor esse custo



Custo da Remoção em ABB

- Observe que um maior custo na busca implica num menor custo na remoção pp. dita; e vice-versa.

=> remoção tem complexidade dependente da altura da árvore.

- Operações de Troca (Caso 3) requerem localizar o maior (menor) elemento da subárvore esquerda (direita) => mas o número de operações é geralmente menor que a altura da árvore.



Impacto das operações de inserção e remoção em ABB

- ABB balanceada ou perfeitamente balanceada => organização ideal para buscas.
- Inserções e eliminações podem desbalancear uma ABB => buscas futuras ineficientes.
- Possível solução:
 - construir uma ABB inicialmente perfeitamente balanceada (algoritmo a seguir)
 - após várias inserções/eliminações, aplicamos um processo de rebalanceamento (algoritmo a seguir)



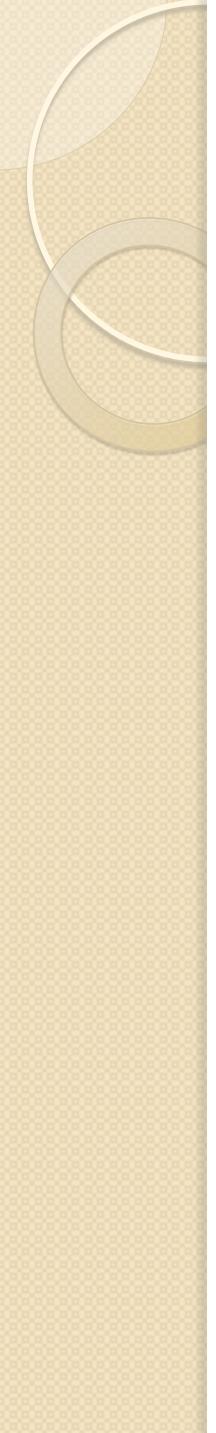
Algoritmo – ABB perfeitamente balanceada

- 1) Ordenar num *array* os registros em ordem crescente das chaves;
- 2) O registro do meio é inserido na ABB vazia (como raiz);
- 3) Tome a metade esquerda do *array* e repita o passo 2 para a subárvore esquerda;
- 4) Idem para a metade direita e subárvore direita;
- 5) Repita o processo até não poder dividir mais.



Algoritmo – rebalanceamento em ABB

- 1) Percorra em **em-ordem** a árvore para obter uma sequência ordenada em *array*.
- 2) Repita os passos 2 a 5 do algoritmo de criação de ABB perfeitamente balanceada.



Considerações Finais

- ABB => boa opção como ED para aplicações de pesquisa (busca) de chaves
 - se árvore balanceada: $O(\log_2 n)$
- Inserções (como folhas) e eliminações (mais complexas) causam desbalanceamento.
 - inserções: melhor se for em ordem aleatória de chaves, para evitar linearização (se ordenadas)
- Para manter o balanceamento, 2 opções:
 - algoritmo descrito anteriormente
 - **Árvores AVL** (próxima aula)