

SCC0202 – Algoritmos e Estruturas de Dados I

Árvores Binárias de Busca

Prof.: Dr. Rudinei Goularte

(rudinei@icmc.usp.br)

Instituto de Ciências Matemáticas e de Computação -
ICMC

Sala 4-229

Conteúdo

- Conceitos Introdutórios
- Operações
 - ▣ Inserção
 - ▣ Pesquisa
 - ▣ Remoção
- Conceitos Adicionais

Definições

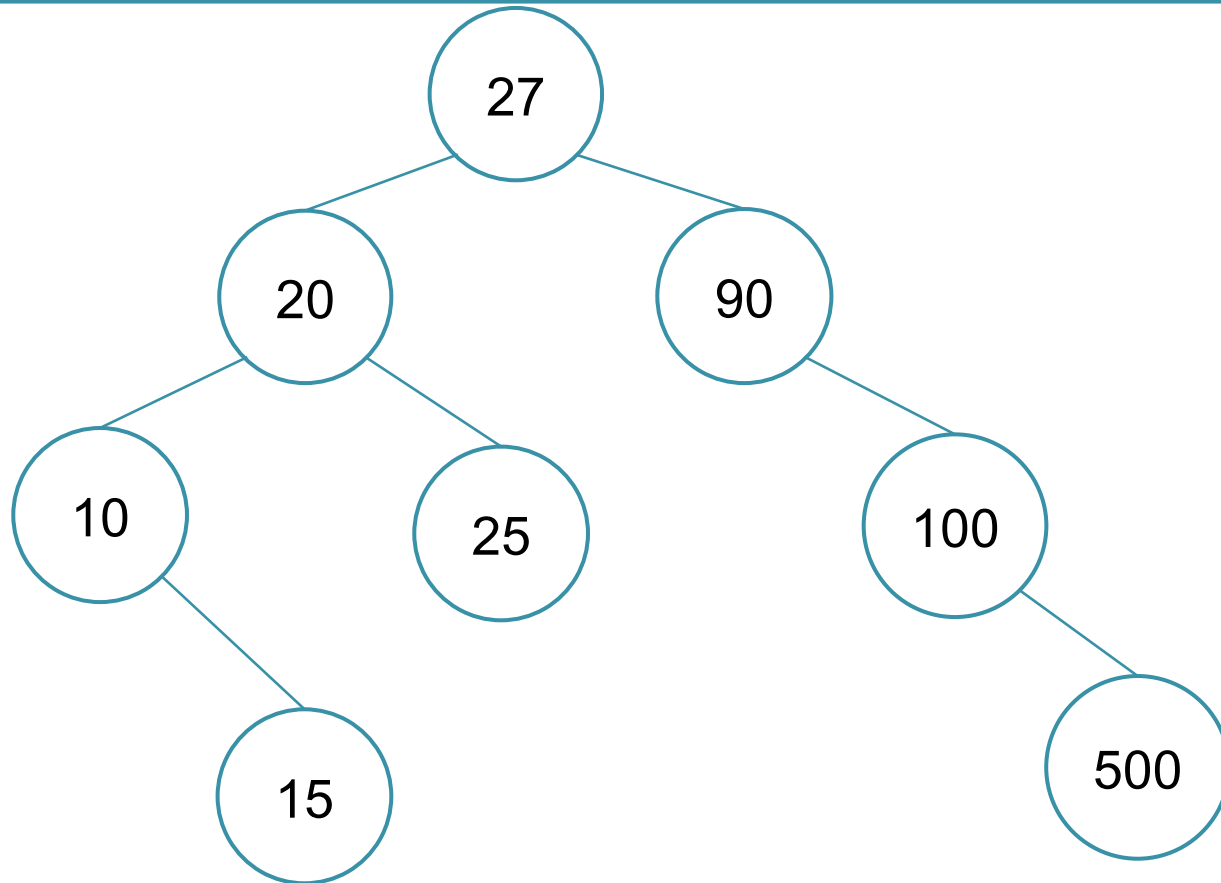
- Uma Árvore Binária de Busca (ABB) possui as seguintes propriedades
 - ▣ Seja $S = \{s_1, \dots, s_n\}$ o conjunto de chaves dos nós da árvore T
 - Esse conjunto satisfaz $s_1 < \dots < s_n$
 - A cada nó $v_j \in T$ está associada uma chave distinta $s_j \in S$, que pode ser consultada por $r(v_j) = s_j$
 - ▣ Dado um nó v de T
 - Se v_i pertence a sub-árvore esquerda de v , então $r(v_i) < r(v)$
 - Se v_i pertence a sub-árvore direita de v , então $r(v_i) > r(v)$

Definições

□ Em outras palavras

- Os nós pertencentes à sub-árvore esquerda possuem valores de chave menores que o valor associado à chave do nó raíz r
- Os nós pertencentes à sub-árvore direita possuem valores de chave maiores que o valor associado à chave do nó raíz r

Exemplo



Definições

- Um **percurso em-ordem** em uma ABB resulta na sequência de valores em **ordem crescente**
- Se **invertêssemos as propriedades** descritas na definição anterior, de maneira que a sub-árvore esquerda de um nó contivesse valores maiores e a sub-árvore direita valores menores, o percurso em-ordem resultaria nos valores em **ordem decrescente**
- **Uma ABB** criada a partir de um conjunto de valores **não é única**: o resultado depende da sequência de inserção dos dados

Definições

- A grande utilidade da árvore binária de busca é armazenar dados contra os quais outros dados são frequentemente verificados (busca!)
- Uma árvore binária de busca é dinâmica e pode sofrer alterações (inserções e remoções de nós) após ter sido criada

Listas versus ABB

- O tempo de busca é estimado pelo número de comparações entre chaves.
- Em listas de n elementos, temos:
 - ▣ Sequenciais (Array): $O(n)$ se não ordenadas; ou $O(\log_2 n)$, se ordenadas
 - ▣ Encadeadas (Dinâmicas): $O(n)$
- As ABB constituem a alternativa que combina as vantagens de ambos: são encadeadas e permitem a busca binária $O(\log_2 n)$!!!

Operações em ABB

- ☐ Inserção
- ☐ Pesquisa/Busca
- ☐ Remoção

Inserção em ABB

□ Passos do algoritmo de inserção

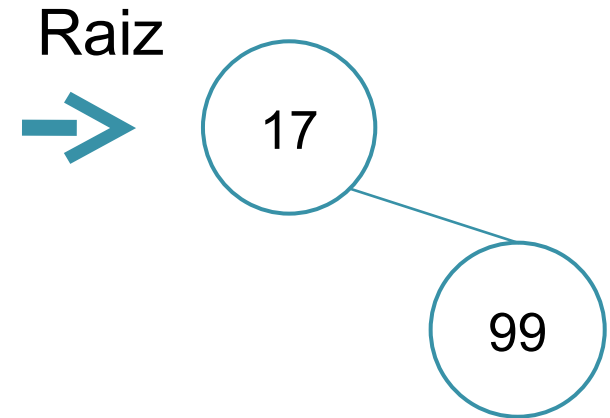
- Procure um “local” para inserir o novo nó, começando a procura a partir do nó-raiz.
- Se um ponteiro (filho esquerdo/direito de um nó-raiz) nulo é atingido, coloque o novo nó como sendo filho do nó-raiz
- Para cada nó-raiz de uma sub-árvore, compare:
 - se o novo nó possui chave menor do que a chave do nó-raiz (vai para sub-árvore esquerda), ou
 - se a chave é maior que a chave do nó-raiz (vai para sub-árvore direita)

Inserção em ABB

- Para entender o algoritmo considere a inserção do conjunto de números, na sequência
 - ▣ 17, 99, 13, 1, 3, 100, 400
- No início a ABB está vazia!

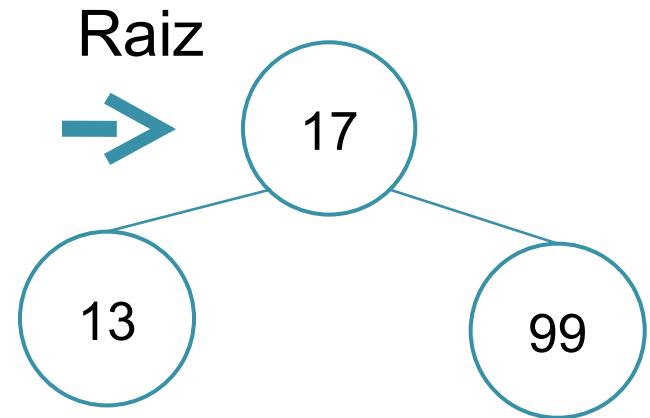
Inserção em ABB

- ❑ 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 (subárvore direita, inicialmente, nula)



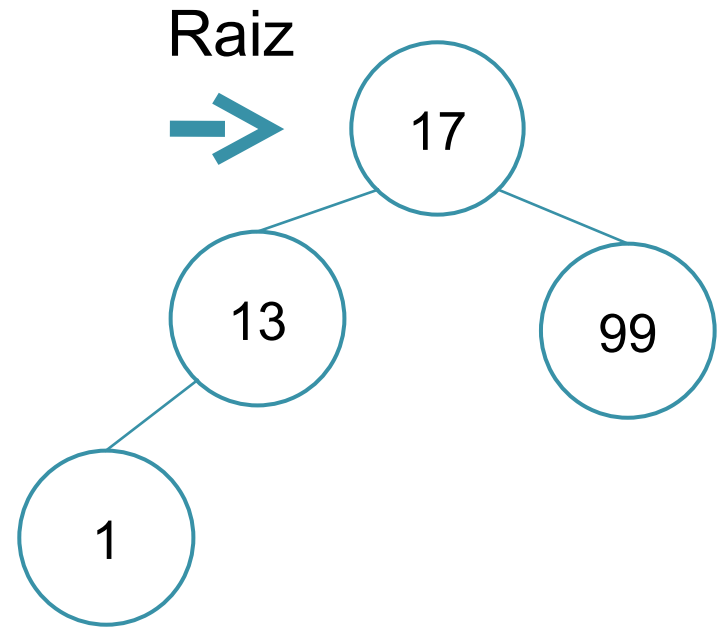
Inserção em ABB

- A inserção do 13 inicia-se na raiz
- Compara-se 13 com 17. Como $13 < 17$, 13 deve ser colocado na sub-árvore esquerda do nó contendo 17
- Já que o nó 17 não possui descendente esquerdo, 13 é inserido na árvore nessa posição



Inserção em ABB

- Repete-se o procedimento para inserir o valor 1
- $1 < 17$, então será inserido na sub-árvore esquerda
- Chegando nela, encontra-se o nó 13, $1 < 13$ então ele será inserido na sub-árvore esquerda de 13



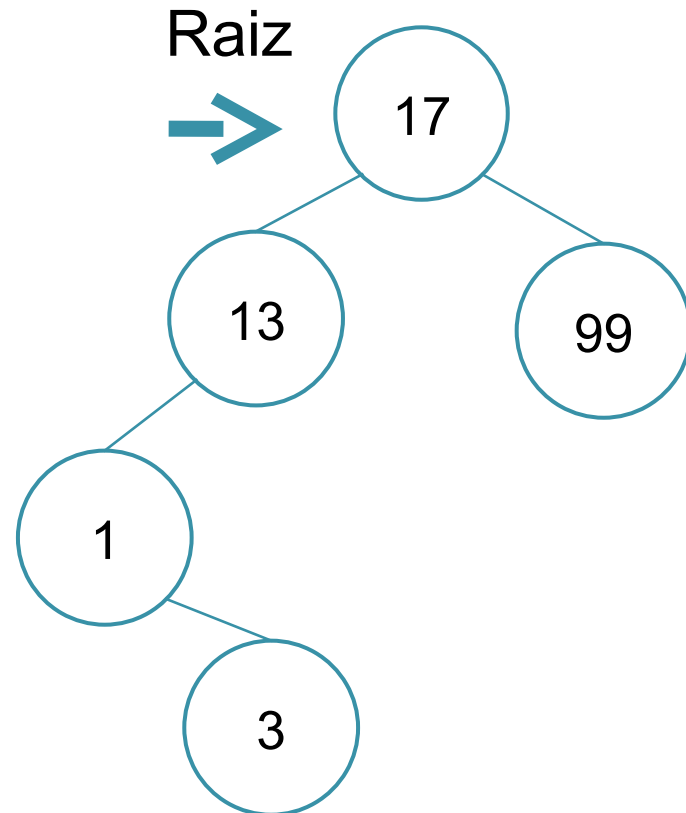
Inserção em ABB

□ Repete-se o procedimento para inserir o elemento 3

▣ $3 < 17$

▣ $3 < 13$

▣ $3 > 1$

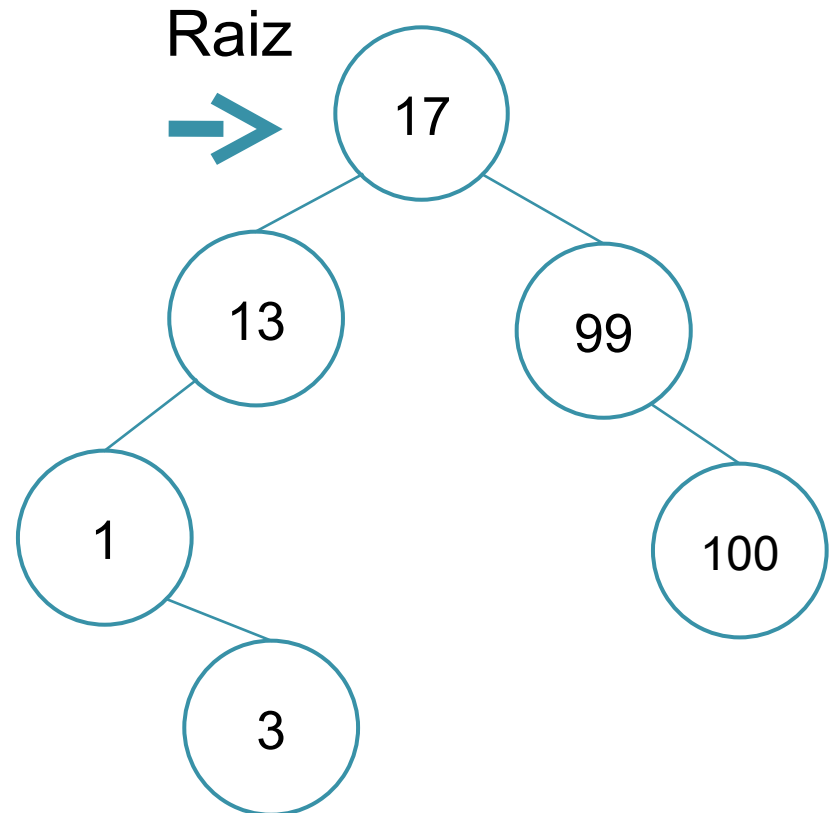


Inserção em ABB

□ Repete-se o procedimento para inserir o elemento 100

▣ $100 > 17$

▣ $100 > 99$



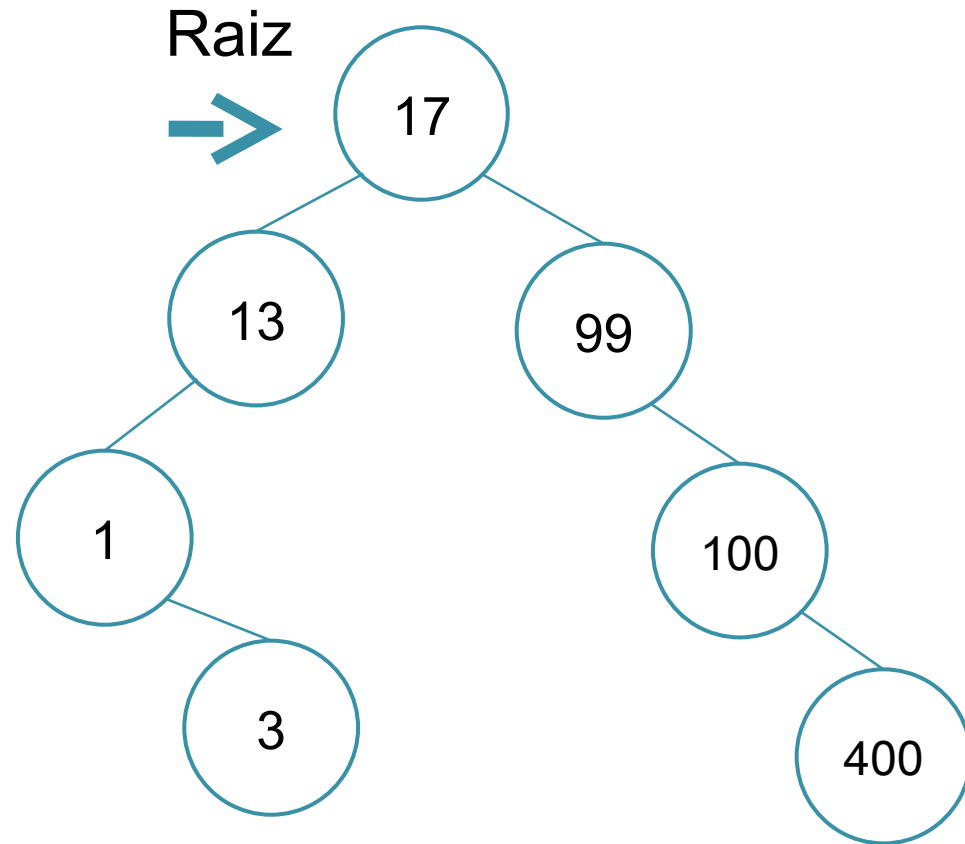
Inserção em ABB


□ Repete-se o procedimento para inserir o elemento 400

▣ $400 > 17$

▣ $400 > 99$

▣ $400 > 100$



- 
- Código para Inserção em ABBs
 - Lógica a partir do programa cliente

Código para ABB

```
1. NO *abb_inserir_no(NO *raiz, NO *novo_no){
2.     if (raiz == NULL)
3.         raiz = novo_no;
4.     else if(item_get_chave(novo_no->item) < item_get_chave(raiz->item))
5.         raiz->esq = abb_inserir_no(raiz->esq,novo_no);
6.     else if(item_get_chave(novo_no->item) > item_get_chave(raiz->item))
7.         raiz->dir = abb_inserir_no(raiz->dir,item);
8.     return(raiz);
9. }
```

```
1. bool abb_inserir (ABB *T, ITEM *item){
2.     NO *novo_no;
3.     if (T == NULL)
4.         return(false);
5.     novo_no = abb_cria_no(item);
6.     if (novo_no != NULL){
7.         T->raiz = abb_inserir_no(T->raiz, novo_no);
8.         return(true);
9.     }
10.    return(false);
11. }
```

Exercício

- Criar um método iterativo para inserção em ABB

Busca em ABB

- Passos do algoritmo de busca
 - ▣ Comece a busca a partir do nó-raiz
 - ▣ Caso o nó pesquisado seja nulo, retorne nulo; caso o nó contendo a chave pesquisada seja encontrado, retorne o “item” do nó pesquisado.
 - ▣ 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), ou se o valor é maior que o valor no nó-raiz (sub-árvore direita)

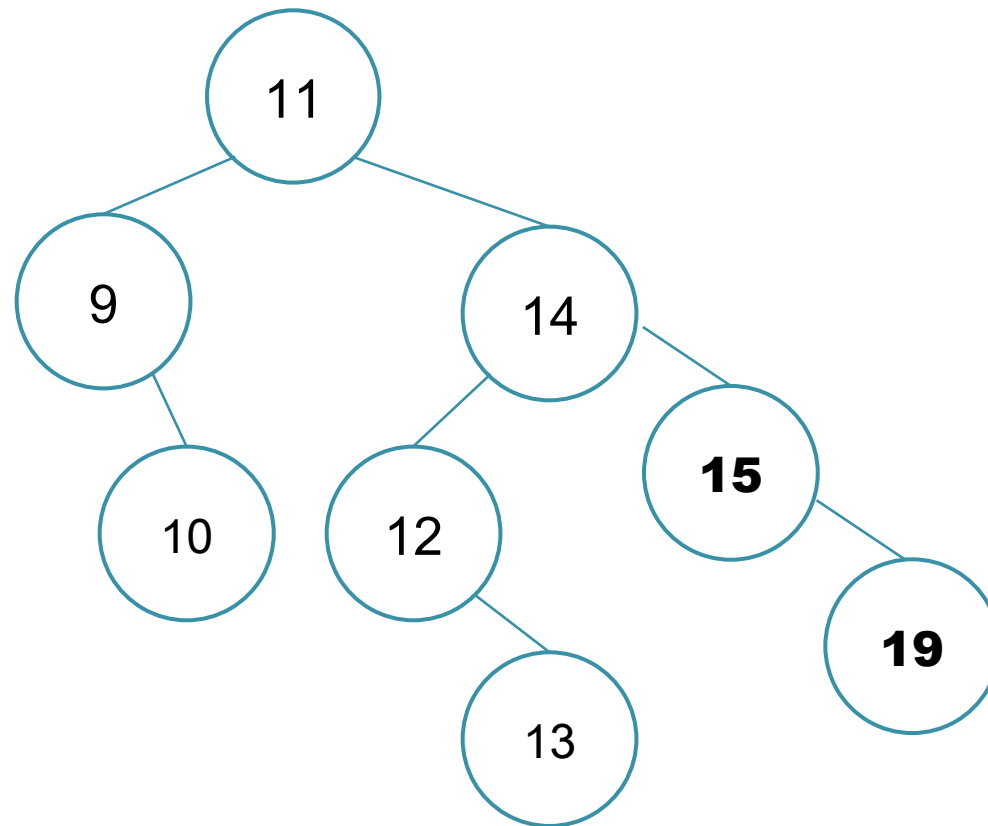
Busca em ABB

```
1  ITEM *abb_busca2(N0 *raiz, int chave){
2      if(raiz == NULL)
3          return NULL;
4      if(chave == item_get_chave(raiz->item))
5          return (raiz->item);
6
7
8      if(chave < item_get_chave(raiz->item))
9          return (abb_busca2(raiz->esq, chave));
10     else
11         return (abb_busca2(raiz->dir, chave));
12 }
13
14 ITEM *abb_busca(ABB *T, int chave){
15     return(abb_busca2(T->raiz, chave));
16 }
```

Exercício

- Criar um método iterativo para busca em ABB

Remoção em ABB

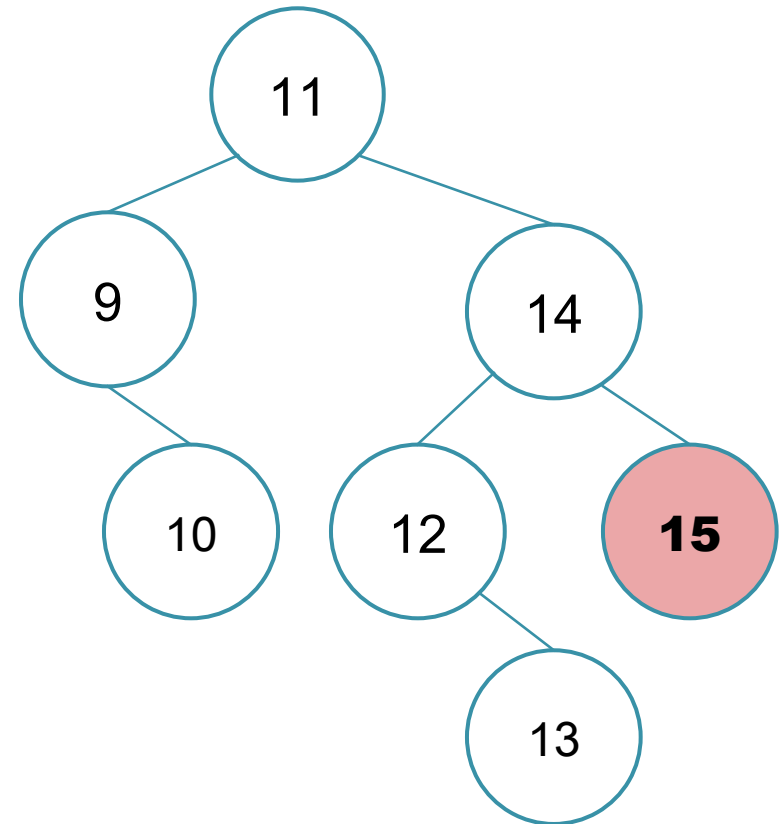


Remoção em ABB

- Casos a serem considerados no algoritmo de remoção de nós de uma ABB
 - ▣ 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

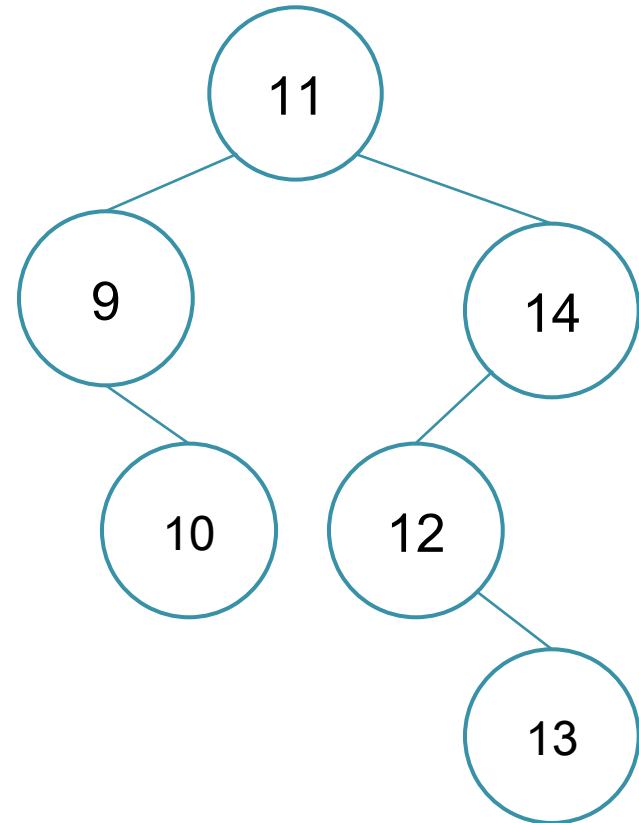
Remoção – Caso 1

- Caso o valor a ser removido seja o 15
- Pode ser removido sem problema, não requer ajustes posteriores



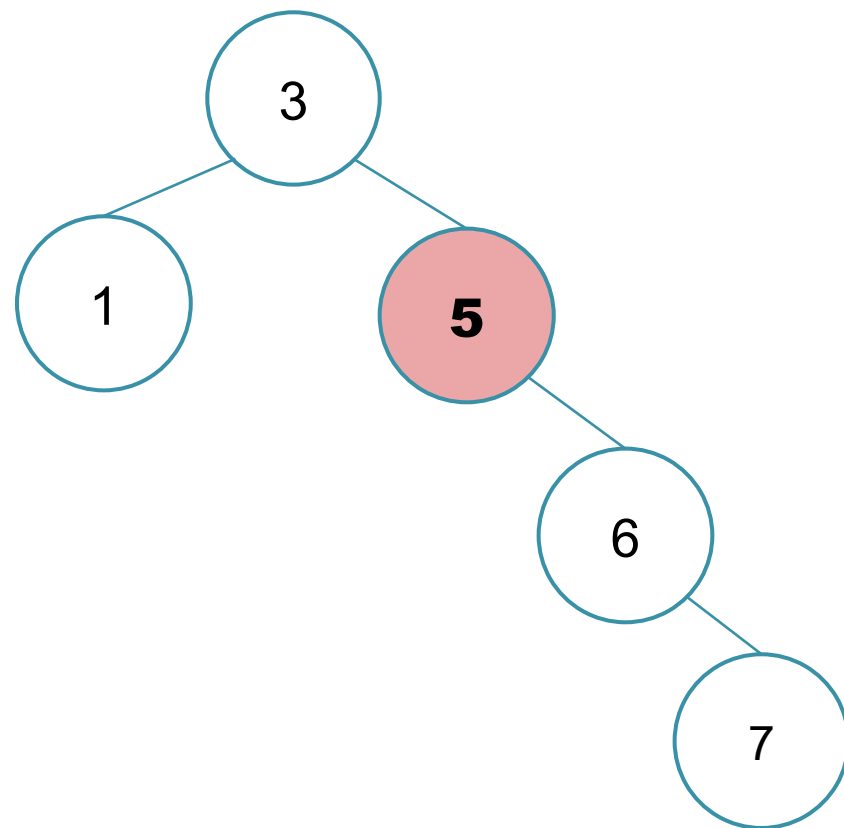
Remoção – Caso 1

- Os nós com os valores 10 e 13 também podem ser removidos



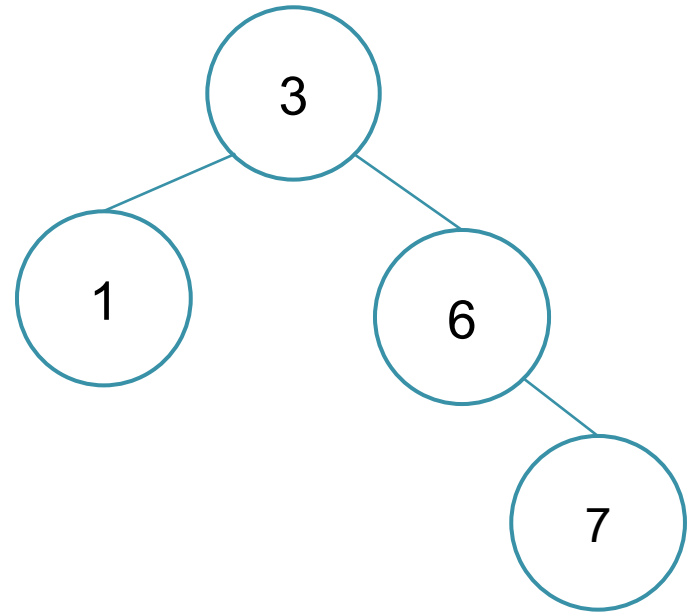
Remoção – Caso 2

- Removendo-se o nó com o valor 5
- Como ele possui somente a sub-árvore direita, o nó contendo o valor 6 pode “ocupar” o lugar do nó removido



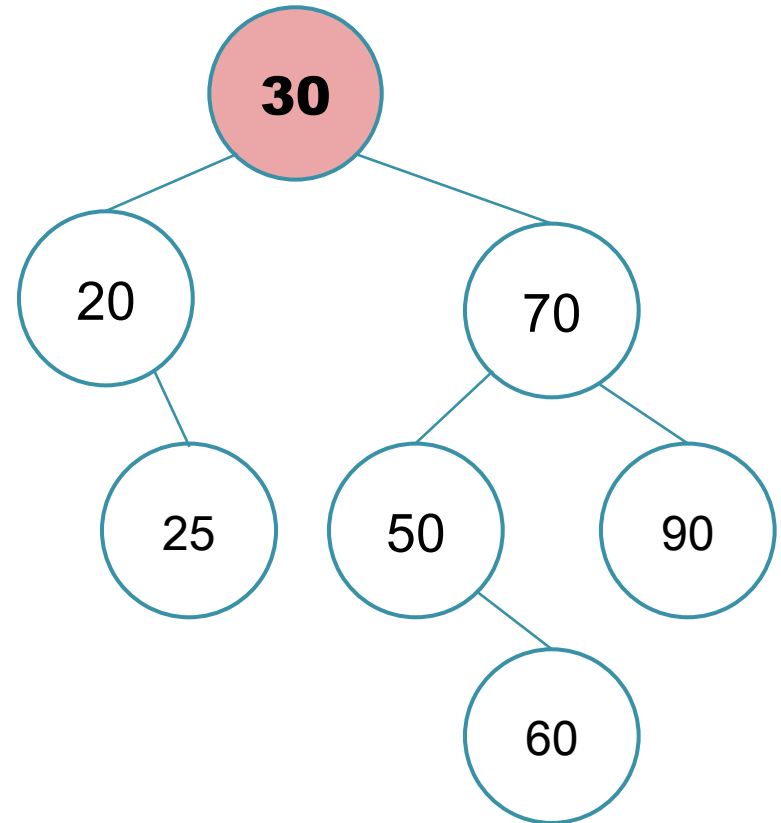
Remoção – Caso 2

- Esse segundo caso é análogo caso existir um nó com sub-árvore esquerda apenas



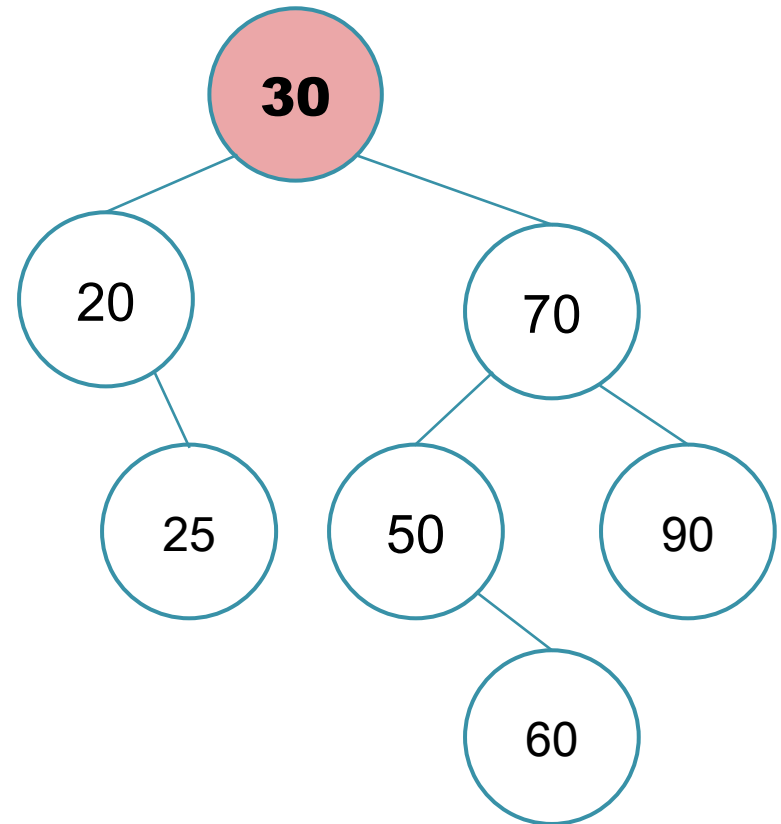
Remoção – Caso 3

- Eliminando-se o nó de chave 30



Remoção – Caso 3

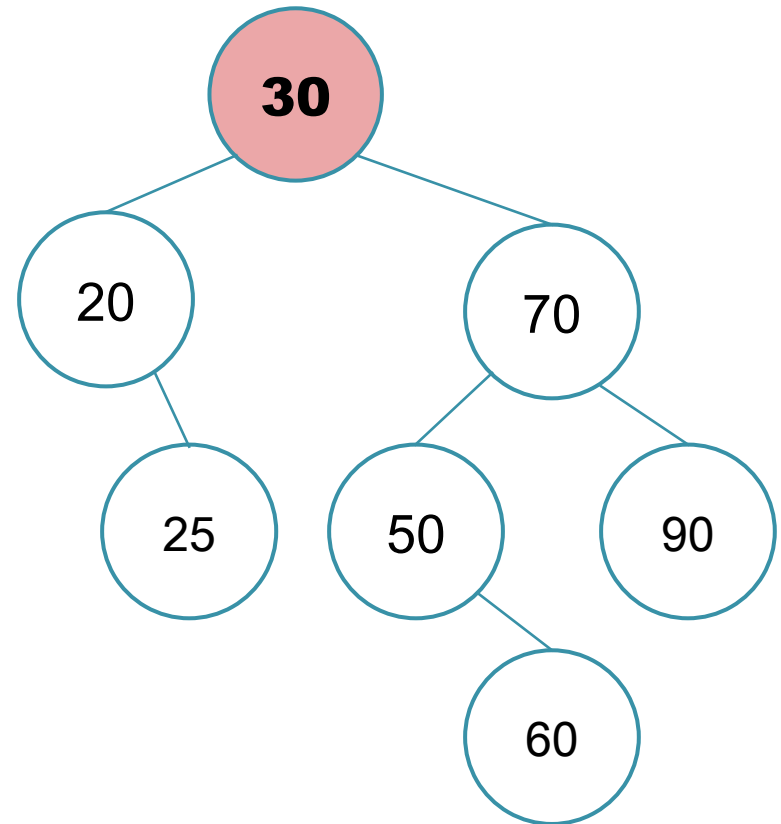
- Eliminando-se o nó de chave 30
- Neste caso, existem 2 opções
 - ▣ O nó com chave 25 pode “ocupar” o lugar do nó-raiz – maior da sub-árvore esquerda, ou
 - ▣ O nó com chave 50 pode “ocupar” o lugar do nó-raiz – menor da sub-árvore direita.



Remoção – Caso 3

□ Ex.:

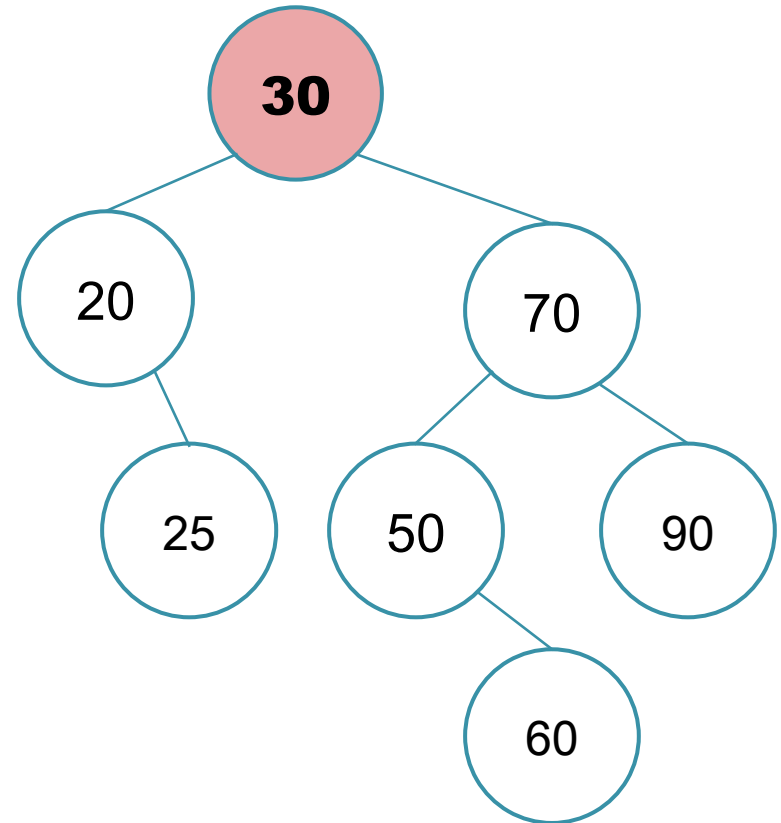
- ▣ O nó com chave 25 pode “ocupar” o lugar do nó-raiz



Remoção – Caso 3

□ Ex.:

- Eliminando o nó 30
- O nó 50 ocupa seu lugar do nó 30



Remoção em ABB

```
1. boolean abb_remove_aux (NO **raiz, int chave){
2.     NO *p;
3.     if(*raiz == NULL)
4.         return (FALSE);
5.     if(chave == item_get_chave((*raiz)->item))
6.     {
7.         if ((*raiz)->esq == NULL || (*raiz)->dir == NULL)
8.             /*Caso 1 se resume ao caso 2: há um filho ou nenhum*/
9.             p = *raiz;
10.            if((*raiz)->esq == NULL)
11.                *raiz = (*raiz)->dir;
12.            else
13.                *raiz = (*raiz)->esq;
14.            free(p);
15.            p = NULL;
16.        }
17.        else /*Caso 3: há ambos os filhos*/
18.            troca_max_esq((*raiz)->esq, (*raiz), (*raiz));
19.        return(TRUE);
20.    }
21.    else
22.        if(chave < item_get_chave((*raiz)->item))
23.            return abb_remove_aux (&(*raiz)->esq, chave);
24.        else
25.            return abb_remove_aux (&(*raiz)->dir, chave);
26.}
```

```
1. boolean abb_remove(ABB *T, int chave){
2.     if (T != NULL)
3.         return (abb_remove_aux(&T->raiz, chave));
4.     return (FALSE);
5. }
```

Atenção: Para deixar o slide mais limpo e focar na remoção do nó, este algoritmo não está removendo o item!

Remoção em ABB

- Troca com o máximo elemento da sub-árvore esquerda

```
1 void troca_max_esq(NO *troca, NO *raiz, NO *ant)
2 {
3     if(troca->dir != NULL)
4     {
5         troca_max_esq(troca->dir, raiz, troca);
6         return;
7     }
8     if(raiz == ant)
9         ant->esq = troca->esq;
10    else
11        ant->dir = troca->esq;
12
13    raiz->item = troca->item;
14    free(troca); troca = NULL;
15 }
```

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.
- Logo, tem complexidade análoga 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.
 - ▣ de sua posição na árvore, caso tenha 2 sub-árvores (quanto mais próximo do último nível, menor esse custo)
- Repare que um maior custo na busca implica num menor custo na remoção pp. dita; e vice-versa.
- Logo, tem complexidade dependente da altura da árvore.
 - ▣ Chamadas à “Troca_max_esq” requerem localizar o maior elemento da sub-árvore esquerda. Mas o número de operações é sempre menor que a altura da árvore.

Árvores Binárias de Busca

□ ABB “aleatória”

- Nós externos: descendentes dos nós folha (não estão, de fato, na árvore)
- Uma árvore A com n nós possui $n + 1$ nós externos
- Uma inserção em A é considerada “aleatória” se ela tem probabilidade igual de acontecer em qualquer um dos $n + 1$ nós externos
- Uma ABB aleatória com n nós é uma árvore resultante de n inserções aleatórias sucessivas em uma árvore inicialmente vazia

Árvores Binárias de Busca

- É possível demonstrar que para uma ABB “aleatória” o número esperado de comparações para recuperar um registro qualquer é cerca de $1,39 * \log_2(n)$
 - ▣ 39% pior do que o custo do acesso em uma árvore balanceada
- Pode ser necessário garantir um melhor balanceamento da ABB para melhor desempenho na busca

Consequências das operações de inserção e eliminação

- Uma ABB balanceada ou perfeitamente balanceada tem a organização ideal para buscas.
- Inserções e eliminações podem desbalancear uma ABB, tornando futuras buscas 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 para criar uma 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 sub-árvore esquerda;
4. Idem para a metade direita e sub-árvore direita;
5. Repita o processo até não poder dividir mais.

Algoritmo de Rebalanceamento

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 PB.

ABB: Resumo

- 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 em ordem aleatória de chaves, para evitar linearização (se ordenadas)
- Para manter o balanceamento, 2 opções:
 - ▣ como descrito anteriormente
 - ▣ **Árvores Binárias Balanceadas (AVL, p.e)**

Exercícios

- ❑ Quais sequências de inserções criam uma ABB degenerada? Quais sequências criam uma ABB balanceada?
- ❑ Implemente um TAD para árvores binárias de busca com as operações discutidas em aula
- ❑ Implemente uma versão iterativa do algoritmo de remoção em ABBs

Exercícios

- Escreva uma função que verifique se uma árvore binária está perfeitamente balanceada
 - ▣ O número de nós de suas sub-árvores esquerda e direita difere em, no máximo, 1