TAD: Árvore Binária de Busca (ABB)

Árvore Binária de Busca (ABB)

- **Definição:** Uma Árvore Binária de Busca (ABB) é uma árvore binária T satisfazendo as seguintes propriedades:

- Para todo nó da árvore, de valor (chave) *X*, tem-se que:

  1. Os nós pertencentes a sua sub-árvore esquerda possuem **valores menores do que** *X*.

  2. Os nós pertencentes a sua sub-árvore direita possuem **valores maiores do que** *X*.

  3. Um percurso em **in-ordem** nessa árvore resulta na sequência de valores em **ordem crescente**.

Árvore Binária de Busca - exemplo

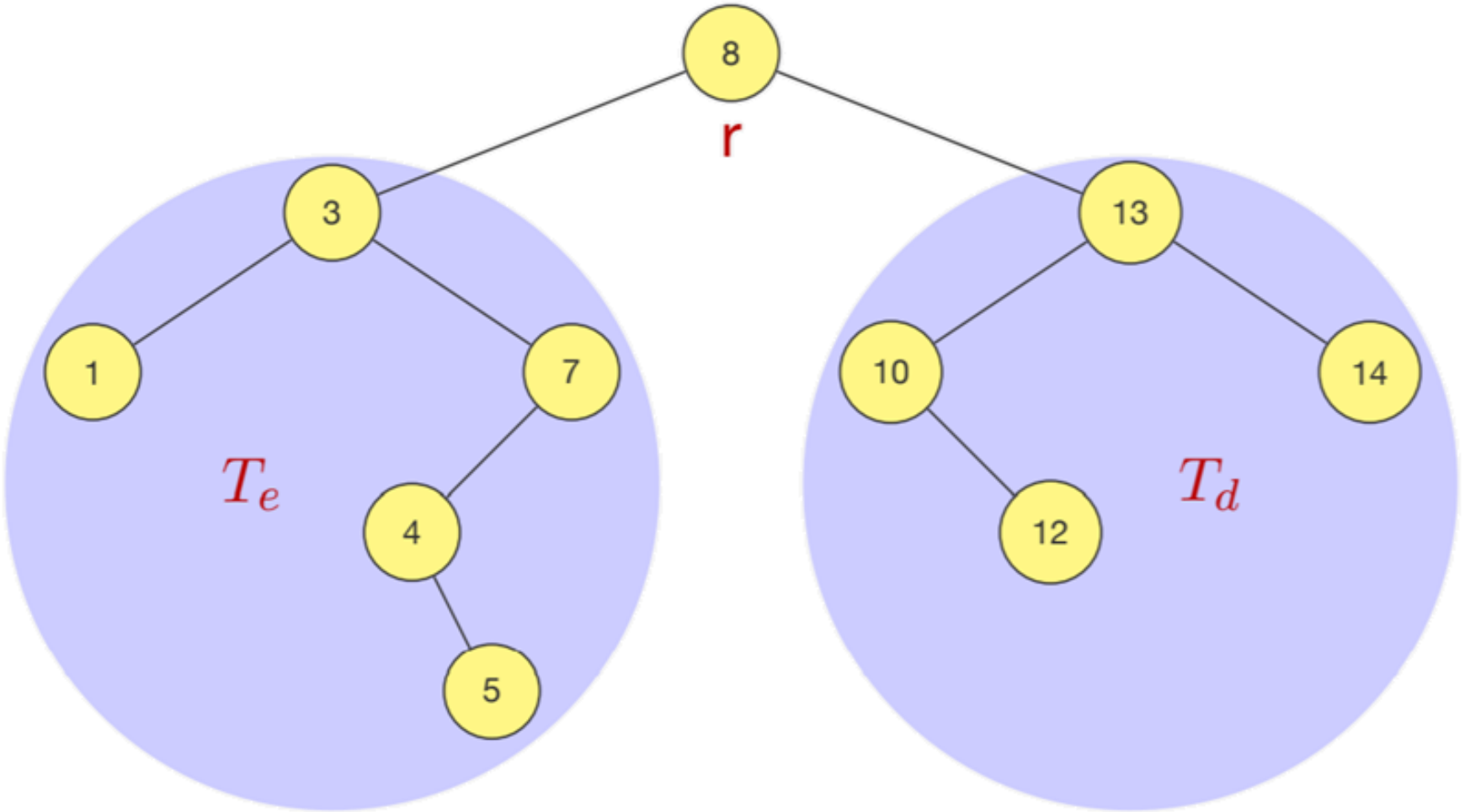
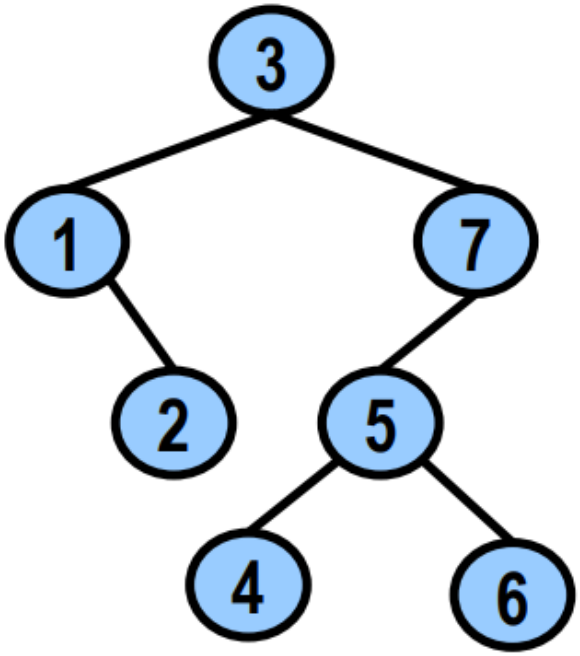


ABB – Observação

- Para um mesmo conjunto valores (atributos chave), podemos construir várias árvores binárias de busca

Exemplo:

$C = \{1, 2, 3, 4, 5, 6, 7\}$



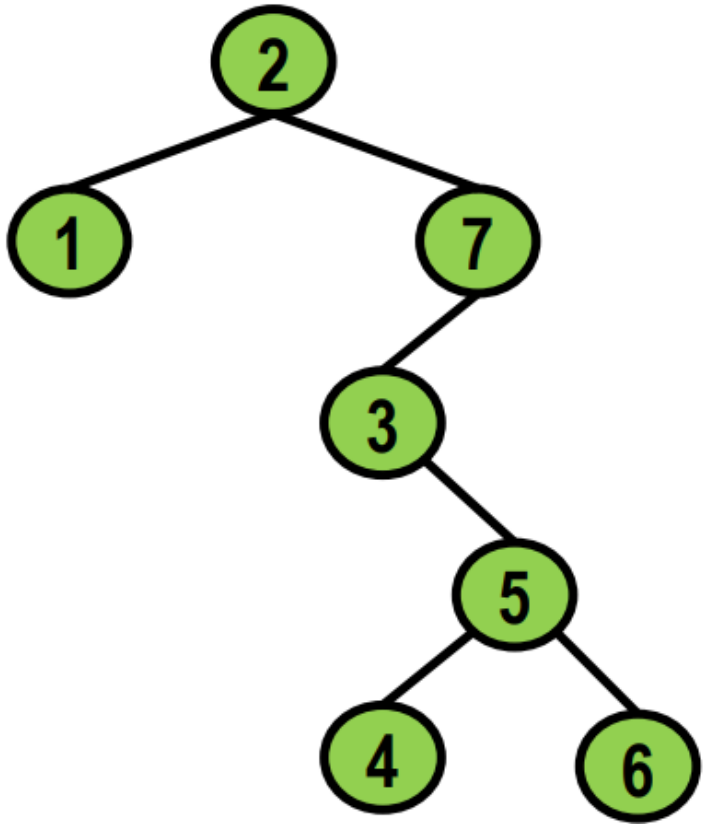


ABB – Observações e características

- Utilidade da **Árvore Binária de Busca:** armazenar dados que são frequentemente verificados, isto é, facilitar o processo de **BUSCA**.

ABB – Características

- Utilidade da **Árvore Binária de Busca:**
  - Armazenar dados que devem ser frequentemente verificados, isto é, facilitar **o processo de BUSCA**.
  - Permite armazenar elementos de forma **organizada** e **eficiente**, otimizando assim as operações de **busca**, **inserção** e **remoção**.
  - **Ordenação**: ABBs estabelecem uma relação de ordem entre seus elementos, podendo ser **percorridas (em ordem)** para gerar **listas ordenadas dos elementos**.
    - Estende o **conceito de busca binária** vista em EDs lineares para **EDs não-lineares**.

ABB – Aplicações

- **Aplicações com ABB em bancos de dados**:
  - Usadas em bancos de dados para **indexar registros** e realizar consultas eficientes.
- **Aplicações com ABB em algoritmos de grafos**:
  - Usadas em algoritmos de manipulação de grafos para percorrer um grafo em ordem específica, como a busca em profundidade-primeiro (DFS) ou busca em largura-primeiro (BFS).
- **Compressão de dados.**
- **Classificação de dados.**
- **Jogos** (Jogadas de xadrez)**.**
- **Sistemas de arquivos** (Hierarquia de pastas e arquivos).

ABB – Operações básicas

- Busca;
- Inserção – *mantendo as propriedades de ABB*;
- Remoção – *mantendo as propriedades de ABB*.

ABB – Operação de busca

ABB – Operação de busca

- **Passos do algoritmo de busca:**
  - Se chave da raíz igual a chave procurada, finaliza-se a busca.
  - **Do contrário:** Repita o processo para a sub-árvore esquerda, **caso chave procurada é menor que a chave da raíz;** se for maior, repita o processo para a sub-árvore direita.
  - Caso o nó contendo o valor pesquisado seja encontrado, retorne um ponteiro para o nó; caso contrário, retorne ponteiro nulo.
- **Observação:** analogia com a busca binária em *arrays* !!!
  - Complexidade da busca: O(log n)

ABB – Exemplo

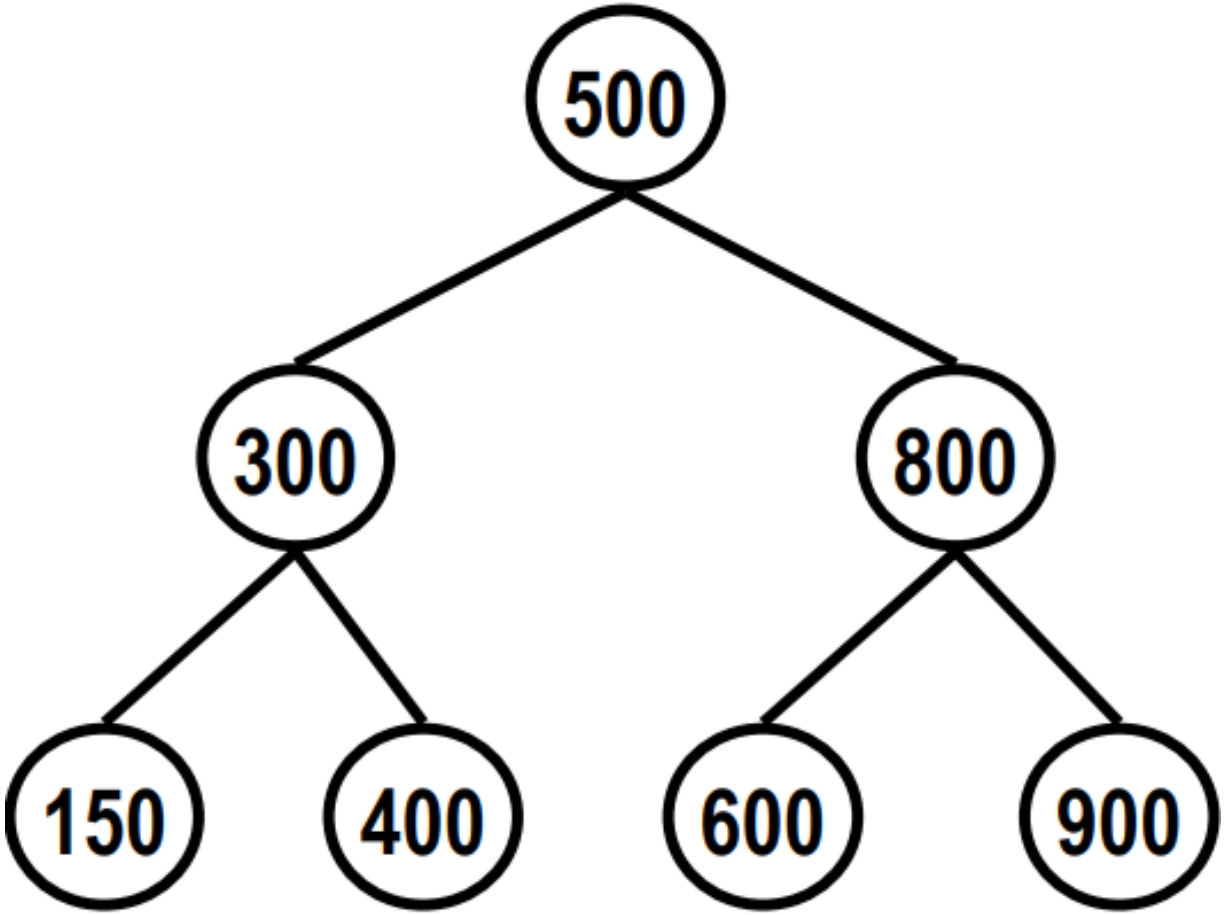


ABB – Exemplo

- E se buscarmos um valor inexistente?

200 < 500

Ir para esquerda

200 < 300

Ir para esquerda

200 > 150

Ir para a direita

NULL: chave não encontrada

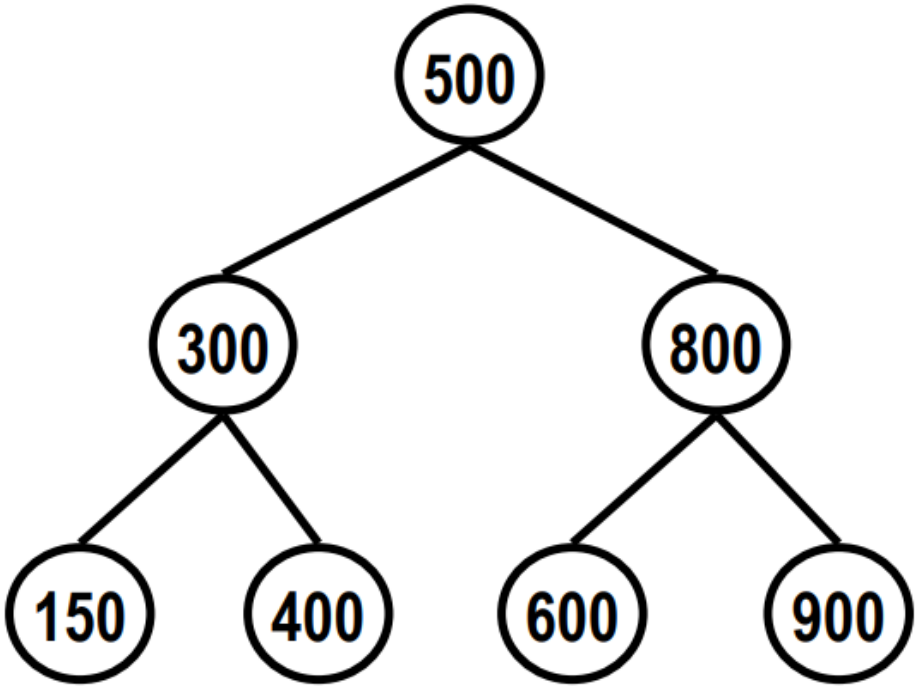


ABB – Implementação busca recursiva

```
//Função (recursiva) de busca para ABB
tree Busca_r (tree raiz, tipo_dado elem)
{
    if (raiz == NULL)
        return NULL;

    if (elem.valor == raiz->info.valor)
        return raiz;

    if (elem.valor < raiz->info.valor)
        return Busca_r(raiz->esq, elem);
    else
        return Busca_r(raiz->dir, elem);
}
```

ABB – Implementação busca não-recursiva

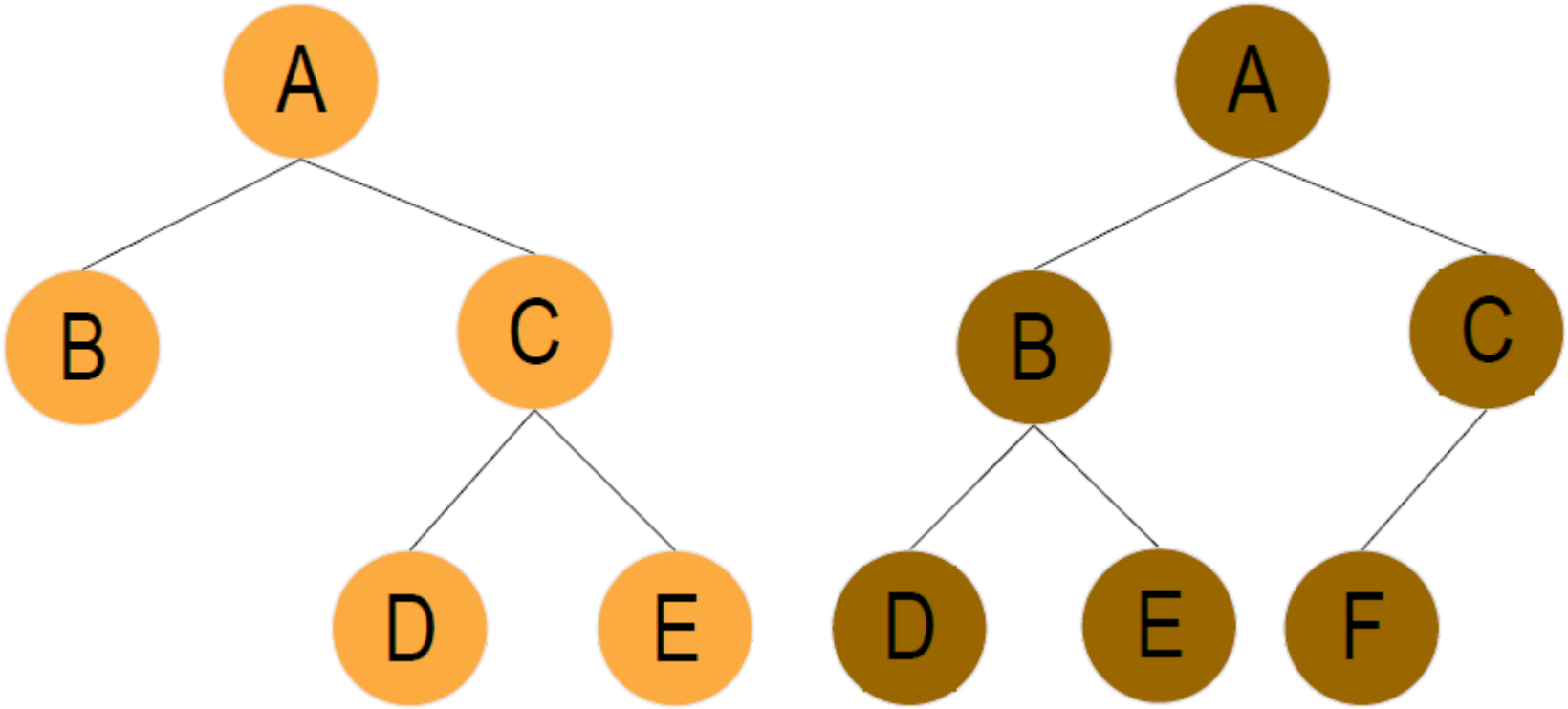
```
//Função (não-recursiva) de busca para ABB
tree Busca_nr (tree raiz, tipo_dado elem)
{
    tree p = raiz;

    while (p != NULL)
    {
        if (p->info.valor == elem.valor)
            return p;
        else if (elem.valor > p->info.valor)
            p = p->dir;
        else
            p = p->esq;
    }

    return p;
}
```

Relembrando: árvore binária balanceada

- **Árvore binária balanceada**:
  - Para cada nó, **as alturas** de suas duas sub-árvores diferem de, **no máximo, uma unidade**.



Árvore binária perfeitamente balanceada

- **Árvore binária perfeitamente balanceada (relembrando)**:
  - É uma AB balanceada com a propriedade de que, para cada nó da árvore, os **números de nós** de suas sub-árvores esquerda e direita **diferem em, no máximo, 1.**

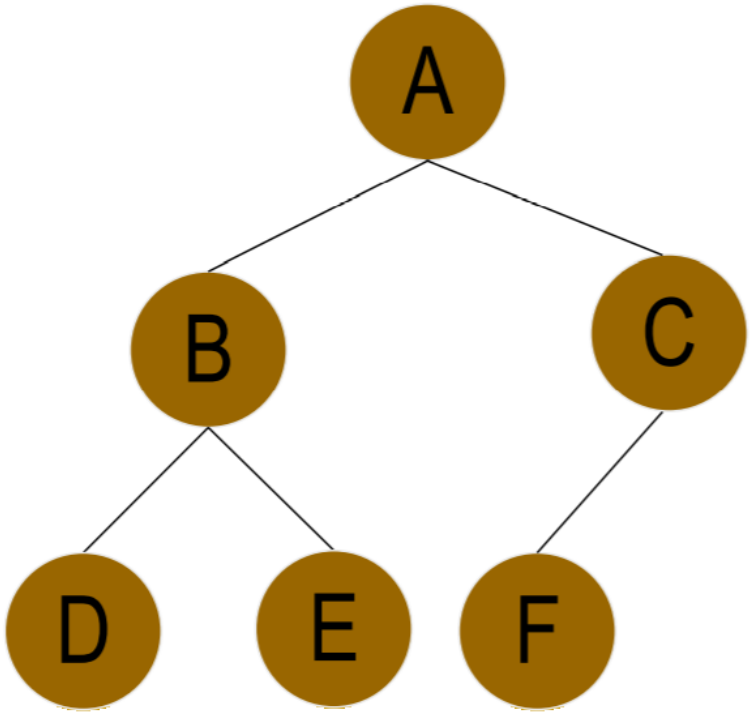


ABB – Busca x complexidade

- Em cada chamada da função busca, é efetuado uma comparação.
  - A **complexidade** da busca: determinada em relação ao **número de chamadas da função**.
  - Quanto **menor a altura da árvore**, **menor o número de comparações** (já que é feira uma comparação por nível).

- **Pior caso** (quando chave buscada está na folha): complexidade está relacionada com a **altura da árvore** (Árvore degenerada!).

- **Busca ótima**: árvore de altura mínima (perfeitamente balanceada).

- **Busca eficiente**: árvore razoavelmente balanceada (árvore balanceada).

ABB – Operação de inserção

ABB – Operação de inserção

- Para cada nó do tipo raiz, comparar:
  1. Se a nova chave for **menor** do que a chave do nó-raiz, repetir o processo para a **sub-árvore esquerda**; ou
  2. Se a nova chave for **maior** que a chave do nó-raiz, repetir o processo para a **sub-árvore direita**.
  - Se um ponteiro nulo (filho esquerdo/direito de um nó) é atingido, coloque o novo nó como sendo a raiz dessa sub-árvore vazia.

- **Obs. 1:** O novo nó a ser inserido será sempre **uma folha** ➔ não exige deslocamentos.
- **Obs. 2:** O algoritmo **não garante** que a árvore resultante seja perfeitamente balanceada ou mesmo balanceada.

ABB – Implementação da inserção

```
//Função (recursiva) para inserir um elemento com um
//valor x em uma ABB, caso ele ainda não esteja lá.
//Retorna o ponteiro para o nó que contém o elem. c/ valor x
tree Busca_insere (tree raiz, tipo_dado elem) {
   //Inserir elem. a partir de um nó desalocado
   if (raiz == NULL) {
      raiz = malloc(sizeof(no));
      raiz->info = elem;
      raiz->esq = NULL;
      raiz->dir = NULL;

      return raiz;
   }

   if (elem.valor < raiz->info.valor)
      raiz->esq = Busca_insere(raiz->esq, elem);

   if (elem.valor > raiz->info.valor)
      raiz->dir = Busca_insere(raiz->dir, elem);

   return raiz;
}
```

ABB – Operação de inserção

- **Exemplo:**

  Considere a inserção na ABB do conjunto X a seguir:

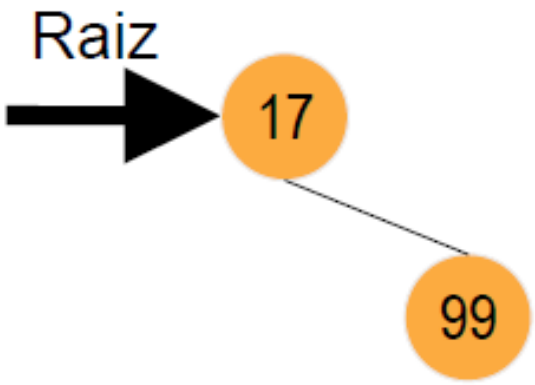
$$X = \{17,\ 99,\ 13,\ 1,\ 3,\ 100,\ 400\}$$

- Obviamente, no início, a ABB encontra-se vazia.

ABB – Operação de inserção

$X = \{17,\ 99,\ 13,\ 1,\ 3,\ 100,\ 400\}$

1. O número 17 será o primeiro a ser inserido, tornando-se nosso nó-raiz.



2. A inserção do 99 inicia-se a partir da raiz. Compara-se então 99 com 17.
   - Como $99 > 17$, o 99 deve então ser colocado na **sub-árvore direita** do nó contendo 17.

ABB – Operação de inserção

$X = \{17,\ 99,\ 13,\ 1,\ 3,\ 100,\ 400\}$

3. 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 (é nulo), 13 é inserido como raiz dessa nova sub-árvore, à esquerda.

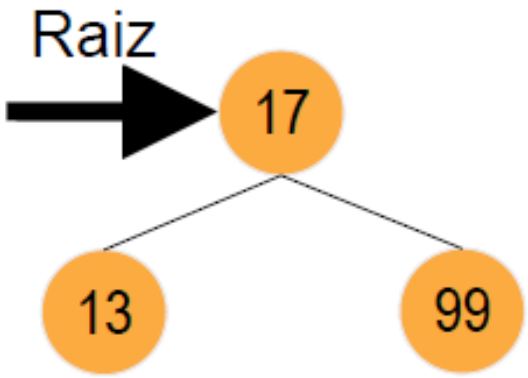


ABB – Operação de inserção

$X = \{17,\ 99,\ 13,\ 1,\ 3,\ 100,\ 400\}$

4. Repete-se o mesmo procedimento para inserir o valor 1.
   - $1 < 17$, então deverá ser remanejado para a **sub-árvore esquerda**.
   - Chegando nela, compara-se com o nó 13. Como $1 < 13$, então 1 deverá ser inserido na **sub-árvore esquerda** de 13.

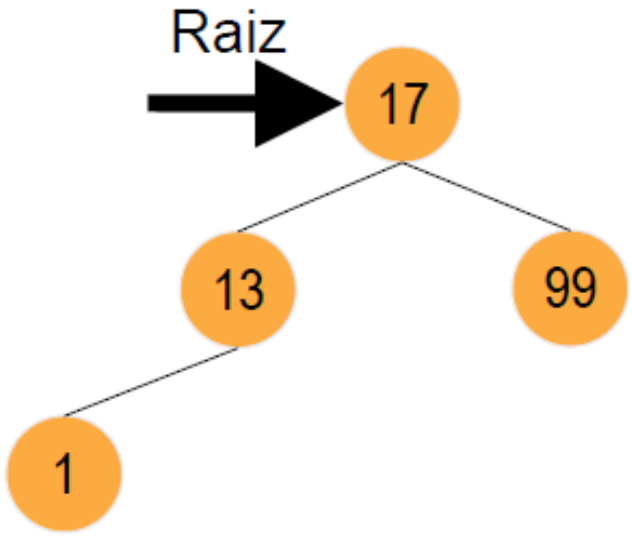


ABB – Operação de inserção

$X = \{17,\ 99,\ 13,\ 1,\ 3,\ 100,\ 400\}$

5. Repete-se o procedimento para inserir o valor 3.
   - $3 < 17$
   - $3 < 13$
   - $3 > 1$

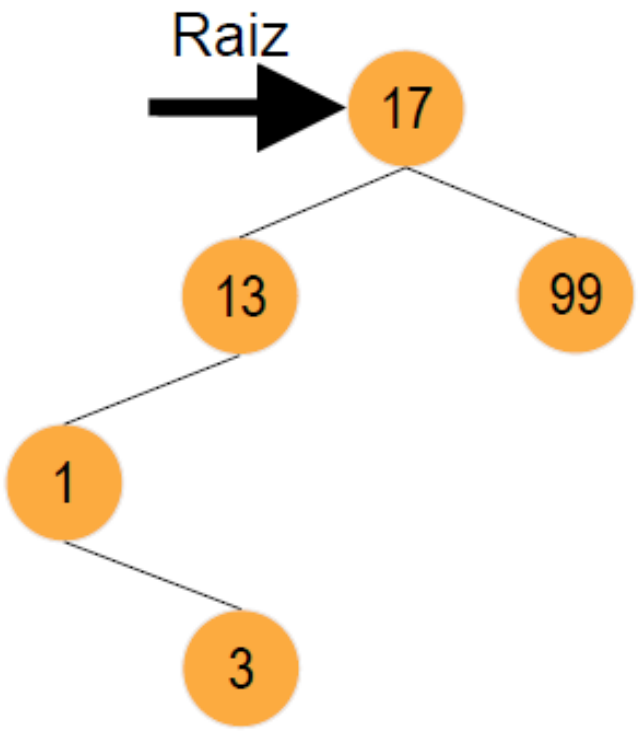


ABB – Operação de inserção

$X = \{17,\ 99,\ 13,\ 1,\ 3,\ 100,\ 400\}$

5. Repete-se o procedimento para inserir o valor 100.
   - $100 > 17$
   - $100 > 99$

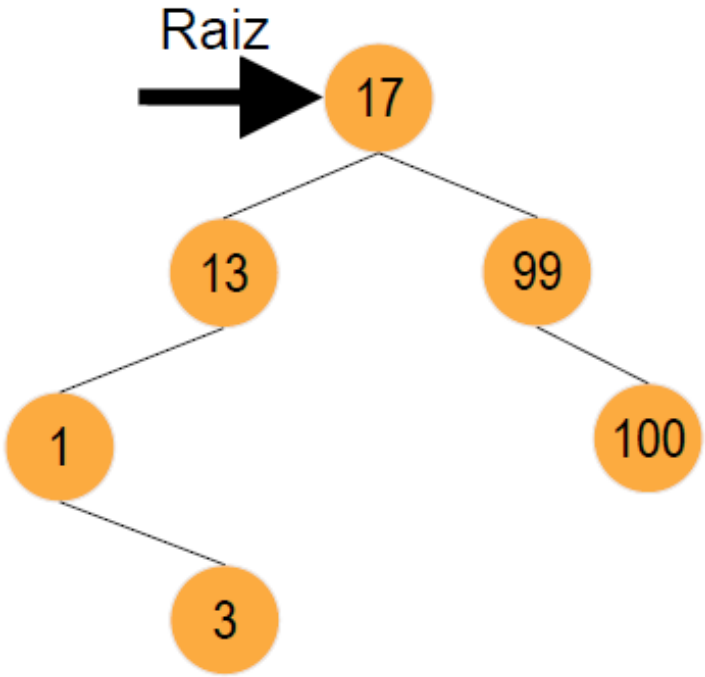
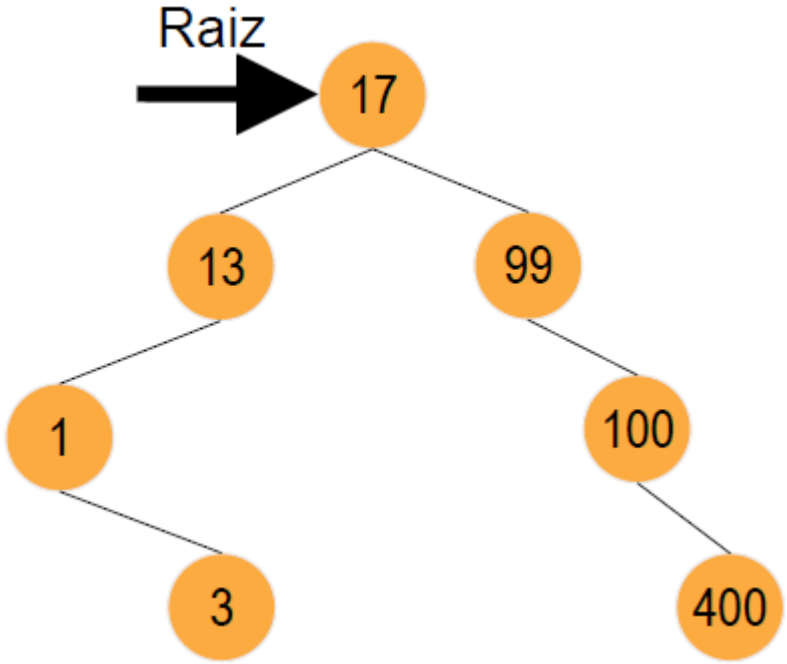


ABB – Operação de inserção

$X = \{17,\ 99,\ 13,\ 1,\ 3,\ 100,\ 400\}$

6. Finalmente, repete-se o mesmo procedimento para inserir o valor final 400.
   - $400 > 17$
   - $400 > 99$
   - $400 > 100$



Perguntas?

ABB – Operação de remoção

ABB – Operação de remoção

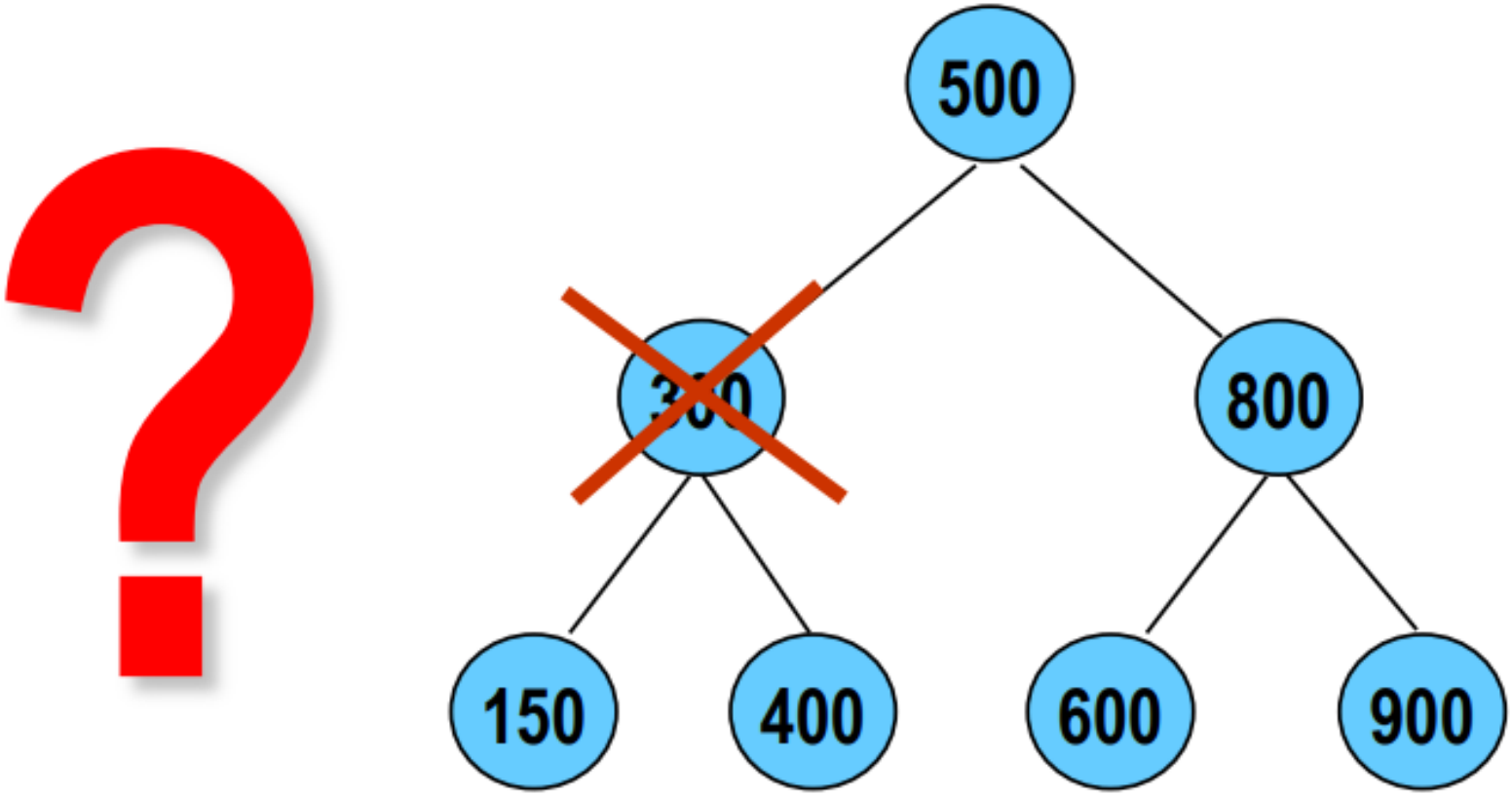


ABB – Operação de remoção

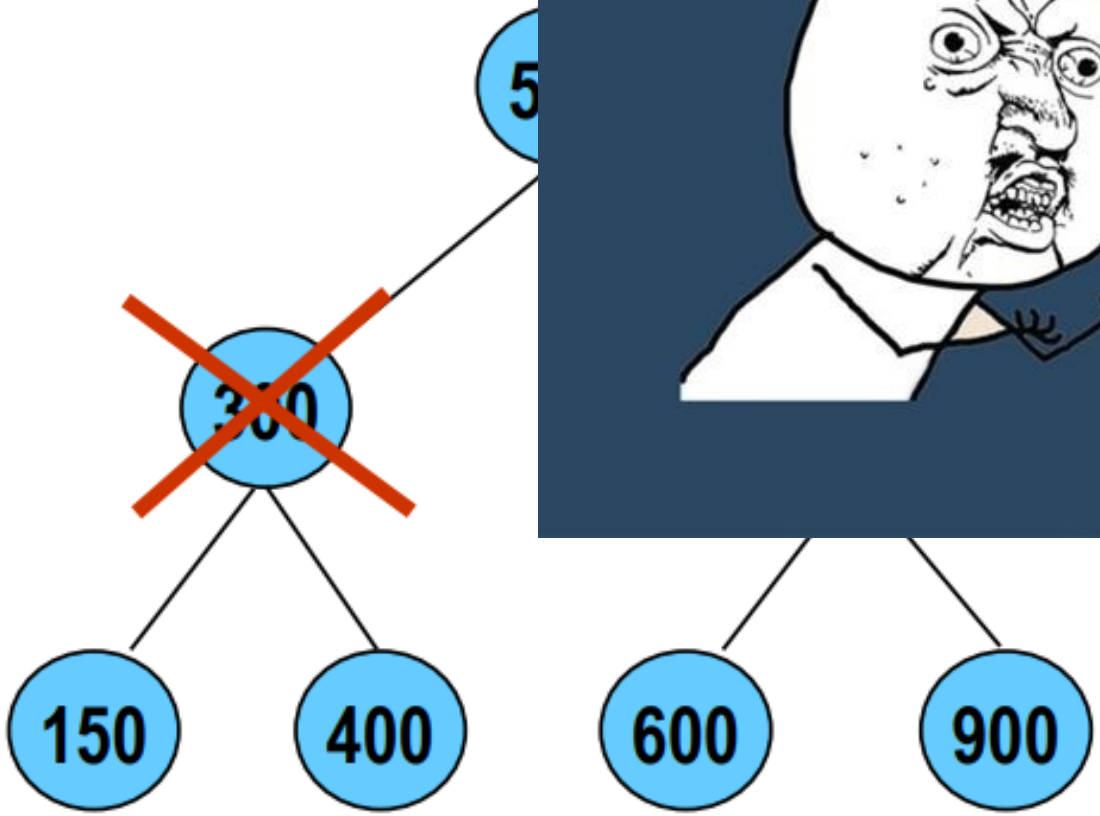


ABB – Operação de remoção

- Casos a serem considerados na remoção de um nó em uma ABB:
  1. Nó é do tipo folha.
     - Nó pode ser removido sem problemas!
  2. Nó possui apenas uma sub-árvore (esquerda/direita)
     - O nó-raiz da sub-árvore (esq./dir.) pode substituir o nó eliminado.
  3. Nó possui duas sub-árvores
     - O nó de **menor valor da sub-árvore direita** pode substituir o nó eliminado ou, alternativamente, o de **maior valor da sub-árvore esquerda** pode substituí-lo. Nota: a partir dessas regras, a árvore irá preservar as propriedades de ser uma ABB.

ABB – Operação de remoção

- **Exemplo (Caso 1):**
  - Caso o valor a ser removido seja o 15.
  - Ele pode ser removido sem problemas, visto que não é necessário ajustes.

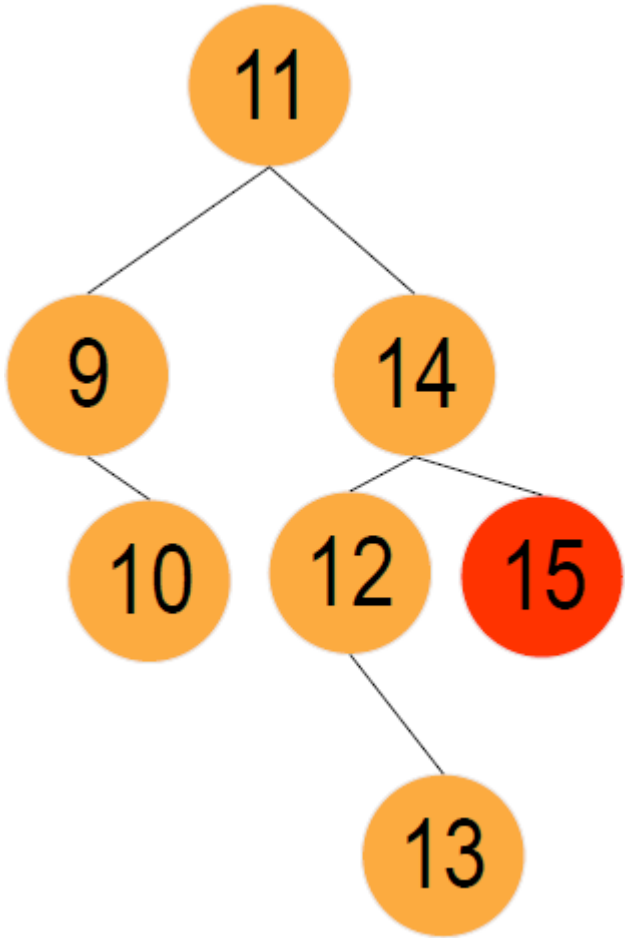


ABB – Operação de remoção

- Casos a serem considerados na remoção de um nó em uma ABB:
  1. Nó é do tipo folha.
     - Nó pode ser removido sem problemas!
  2. Nó possui apenas uma sub-árvore (esquerda/direita)
     - O nó-raiz da sub-árvore (esq./dir.) pode substituir o nó eliminado.
  3. Nó possui duas sub-árvores
     - O nó de **menor valor da sub-árvore direita** pode substituir o nó eliminado ou, alternativamente, o de **maior valor da sub-árvore esquerda** pode substituí-lo. Nota: a partir dessas regras, a árvore irá preservar as propriedades de ser uma ABB.

ABB – Operação de remoção

- **Exemplo (Caso 2):**
  - Remoção do nó com o valor 5.
  - Como ele (nó de valor 5) possui uma sub-árvore direita, o nó contendo o valor 6 pode "ocupar" o lugar desse nó removido.

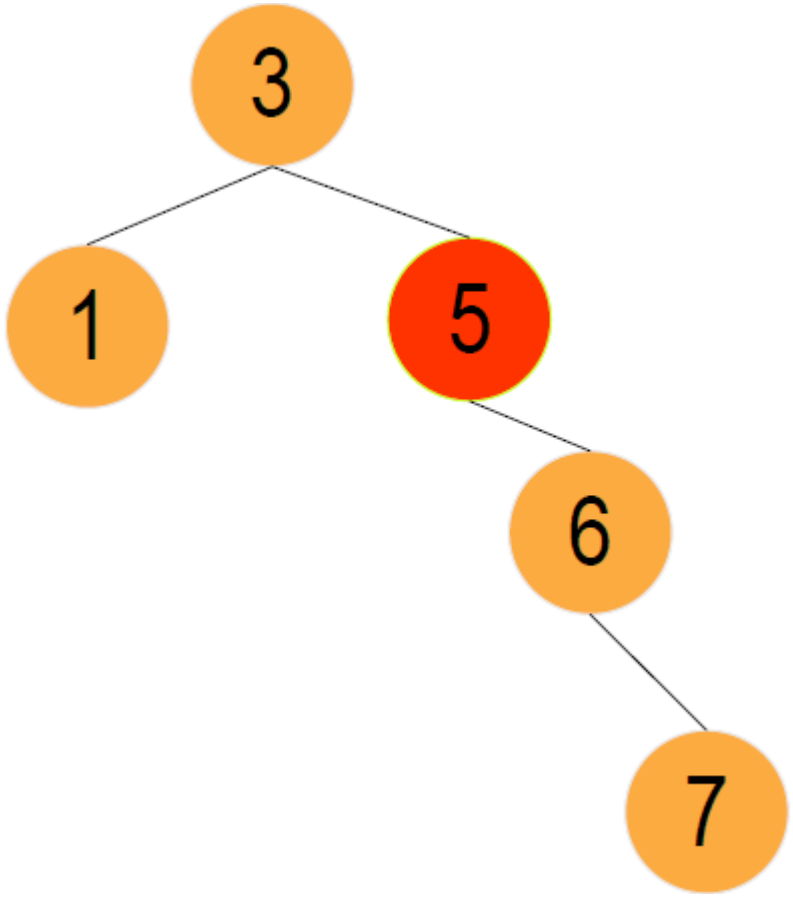


ABB – Operação de remoção

- **Exemplo (Caso 2):**
  - Remoção do nó com o valor 5.
  - Como ele (nó de valor 5) possui uma sub-árvore direita, o nó contendo o valor 6 pode "ocupar" o lugar desse nó removido.

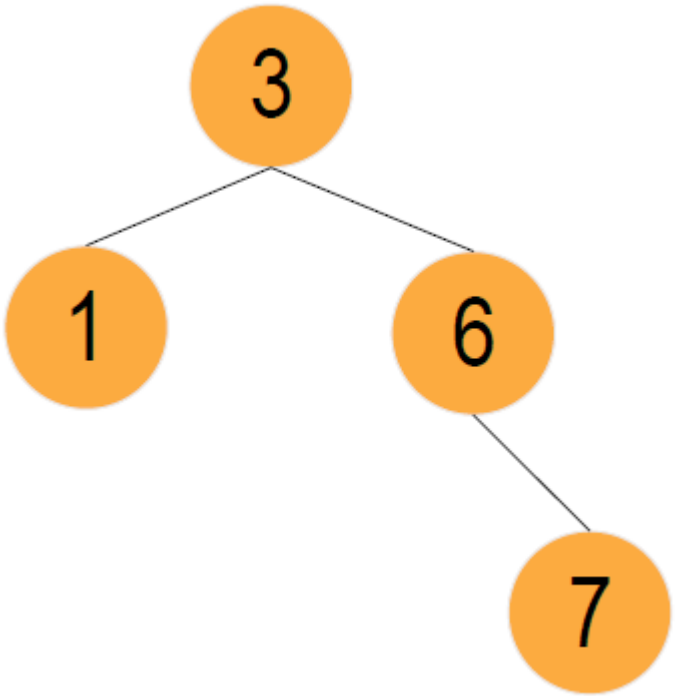


ABB – Operação de remoção

- Casos a serem considerados na remoção de um nó em uma ABB:
  1. Nó é do tipo folha.
     - Nó pode ser removido sem problemas!
  2. Nó possui apenas uma sub-árvore (esquerda/direita)
     - O nó-raiz da sub-árvore (esq./dir.) pode substituir o nó eliminado.
  3. Nó possui duas sub-árvores
     - O nó de menor valor da sub-árvore direita pode substituir o nó eliminado ou, alternativamente, o de maior valor da sub-árvore esquerda pode substituí-lo. Nota: a partir dessas regras, a árvore irá preservar as propriedades de ser uma ABB.

ABB – Operação de remoção

- **Exemplo (Caso 3):**
  - Remoção do nó de chave 11.
  - Neste caso, existem 2 opções:
    - O nó com chave 12, que é o **menor valor** da sub-árvore direita pode "ocupar" o lugar do nó-raiz.
    - O nó com chave 10, que é o **maior valor** da sub-árvore esquerda, pode "ocupar" o lugar do nó-raiz.

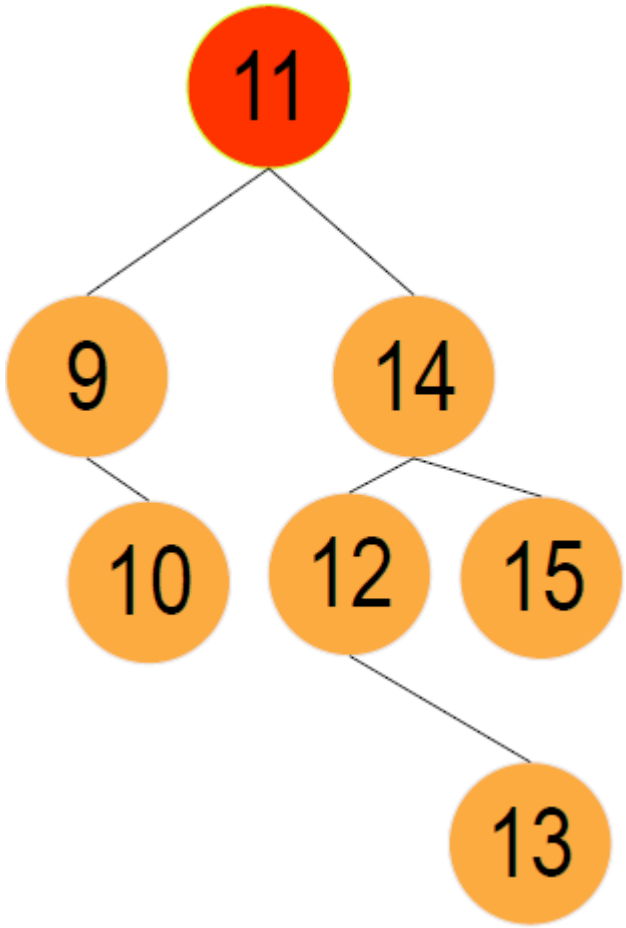


ABB – Operação de remoção

- **Exemplo (Caso 3):**
  - Remoção do nó de chave 11.
  - Neste caso, existem 2 opções:
    - O nó com chave 12, que é o **menor valor** da sub-árvore direita pode "ocupar" o lugar do nó-raiz.
    - O nó com chave 10, que é o **maior valor** da sub-árvore esquerda, pode "ocupar" o lugar do nó-raiz.

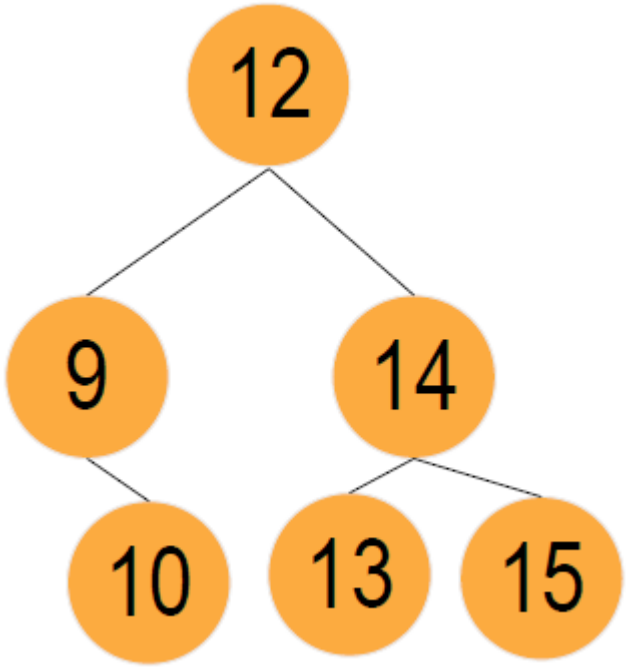


ABB – Operação de remoção

- **Exemplo (Caso 3):**
  - Esse terceiro caso também se aplica ao nó com chave 14, caso seja removido.
    - Nessa configuração, os nós com chave 13, (ou ainda, 15 de acordo com uma das regras adotadas) poderiam ocupar seu lugar.

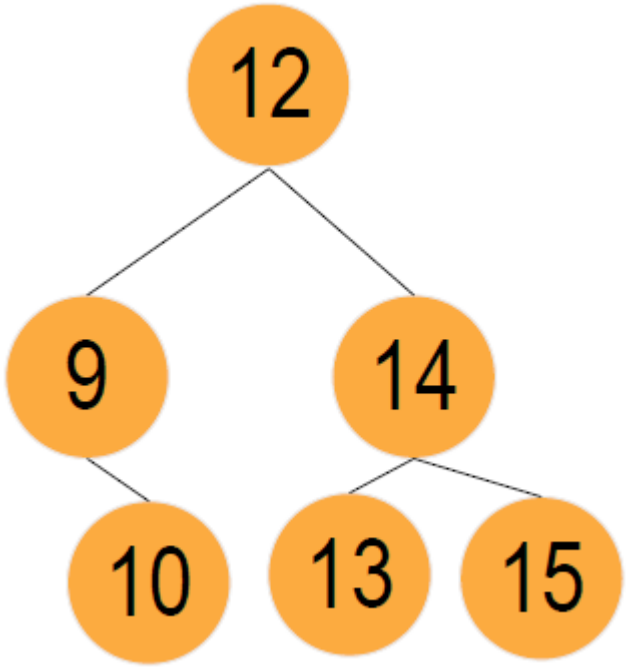
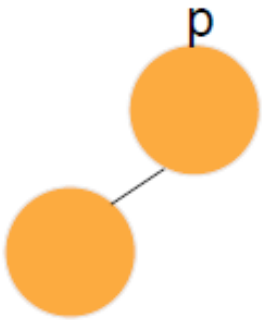


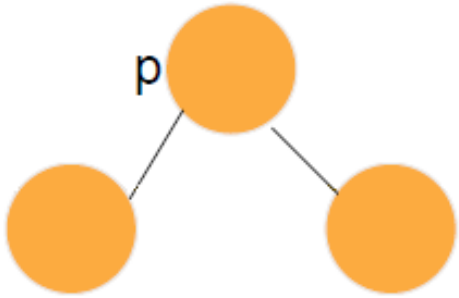
ABB – Implementação da remoção

AQUI TEMOS UM EXEMPLO DE COMO PROGRAMAR SEM PRESSÃO

```c
void Remove_no (tree *p)
{
    tree q;

    //Caso filho único à direita ou é nó folha
    if ((*p)->esq == NULL)
    {
        //Substitui pelo filho à direita
        q = *p;
        *p = (*p)->dir;
        free(q);
    }
    else if ((*p)->dir == NULL) //Caso filho único à esquerda
    {
        //Substitui pelo filho à esquerda
        q = *p;
        *p = (*p)->esq;
        free(q);
    }
    else //Caso tenha dois filhos
        Substitui_menor_a_direita(p, &(*p)->dir);
        //Alternativamente: Substituir_maior_a_esquerda(p, p->esq)
}
```



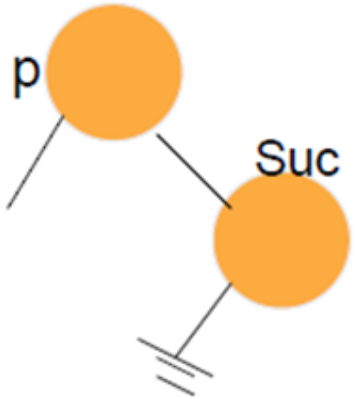


```
//Encontra o sucessor de p, isto é, o descendente mais
//à esquerda da sub-arvore à direita de p.
void Substitui_menor_a_direita(tree *p, tree *suc)
{
    tree q;

    if ((*suc)->esq == NULL)
    {
        (*p)->info = (*suc)->info;

        //Remover sucessor
        q = *suc;
        *suc = (*suc)->dir;
        free(q);

        //Atualizar o ponteiro p na árvore (nao necessario p essa proposta)
        //*p = *suc;
    }
    else
        Substitui_menor_a_direita(p, &(*suc)->esq);
}
```



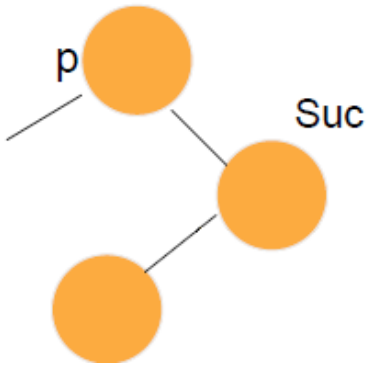
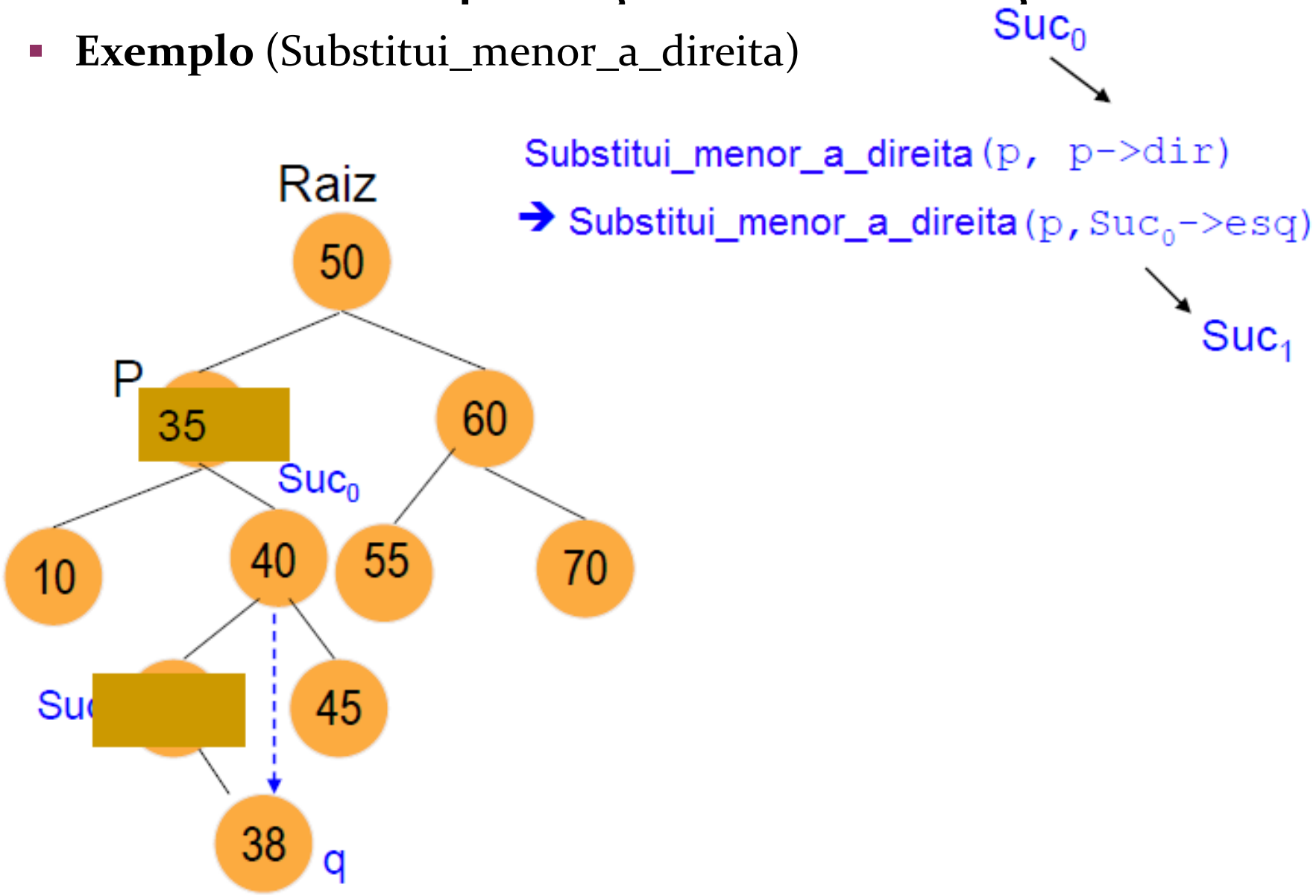


ABB – Operação de remoção

- **Exemplo** (Substitui_menor_a_direita)



```
//Retorna true, se removeu elemento; false se x não está na árvore
bool Busca_remove(tree *p_raiz, tipo_dado elem)
{
   tree raiz = *p_raiz;
   //Árvore vazia; x não está na árvore
   if (raiz == NULL)
      return false;

   //Encontrou exatamente x: eliminar
   if (raiz->info.valor == elem.valor) {
      Remove_no(p_raiz);
      //Caso altere a raiz no procedimento, altera aqui
      return true;
   }

   if (elem.valor < raiz->info.valor) {
      //Buscar e remover na sub-árvore esquerda
      return Busca_remove(&(raiz->esq), elem);
   }
   else {
      //Buscar e remover na sub-árvore direita
      return Busca_remove((&raiz->dir), elem);
   }
}
```

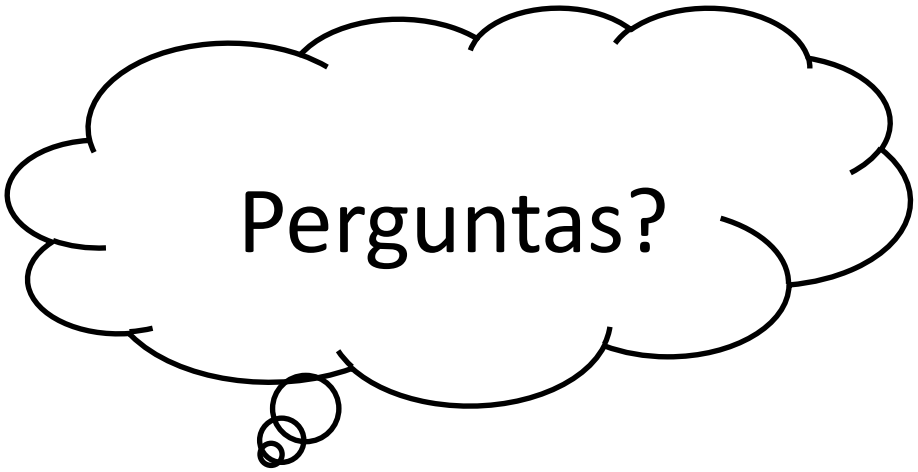

Perguntas?

ABB – Conclusão

- Boa opção como TAD para aplicações de pesquisa (**Busca**) de chaves (valores). Se a árvore é balanceada ➔ $O(\log_2 n)$.
- **Inserções** (como folhas) e **Eliminações** (mais complexas) podem causar o desbalanceamento da árvore.
- **Inserções:** melhor se em ordem aleatória de chaves, para evitar linearização (se ordenadas).
- Para manter o balanceamento da árvore, tem-se duas opções:
  - Algoritmos de rebalanceamento.
  - Árvores AVL.