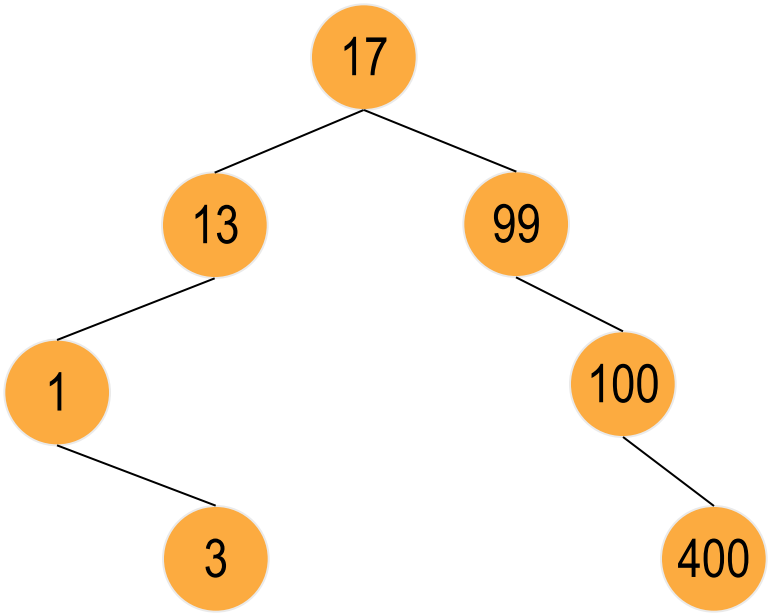
Árvores Binárias de Busca

Definição

- Uma Árvore Binária de Busca possui as mesmas propriedades de uma AB, acrescida da seguinte propriedade: Para todo nó da árvore, se seu valor é X, então:
  - Os nós pertencentes a sua sub-árvore esquerda possuem valores menores do que X;
  - Os nós pertencentes a sua sub-árvore direita possuem valores maiores do que X.
  - Um percurso em in-ordem nessa árvore resulta na seqüência de valores em ordem crescente

Exemplo



Características

- Se invertessemos 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 árvore de busca criada a partir de um conjunto de valores não é única: o resultado depende da ordem de inserção dos dados

Características

- A grande utilidade da árvore binária de busca é armazenar dados contra os quais outros dados são freqüentemente verificados (busca!)

- Uma árvore binária de busca é armazenada dinamicamente e em geral sofre 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 dinâmicas e permitem a busca binária $O(\log_2 n)$

Operações em ABB

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

Busca

- Passos do algoritmo de busca:
  - Se a árvore é vazia, fim e não achou. Se chave da raiz é igual à chave procurada, termina busca com sucesso.
  - Senão: Repita o processo para a sub-árvore esquerda, se chave da raiz é maior que a chave procurada; se for menor, 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 um ponteiro nulo.
- Faça a analogia com a busca binária feita em arrays.

Verifique em que condições a busca em ABB tem performance $O(\log_2 n)$

- Considere:
  - uma comparação por nível;
  - n nós no total
- Portanto:
  - Quanto menor a altura da árvore, menor o número de comparações;
  - Altura mínima de uma AB de n nós é $O(\log_2 n)$
  - AB perfeitamente balanceadas (PB) têm altura mínima.
- Logo: ABB ideais são PB (e deveriam ser mantidas como tal, porém são dinâmicas.....)

Busca (recursiva)

```
pno busca(tree raiz, tipo_elem valor){

    if (raiz == NULL)
        return NULL;

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

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

Busca (não recursiva)

```
pno busca(tree raiz, tipo_elem valor){
    pno p;

    p = raiz;

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

    return p;
}
```

Recursiva versus Não-Recursiva

• Em relação a tempo (número de comparações): equivalentes

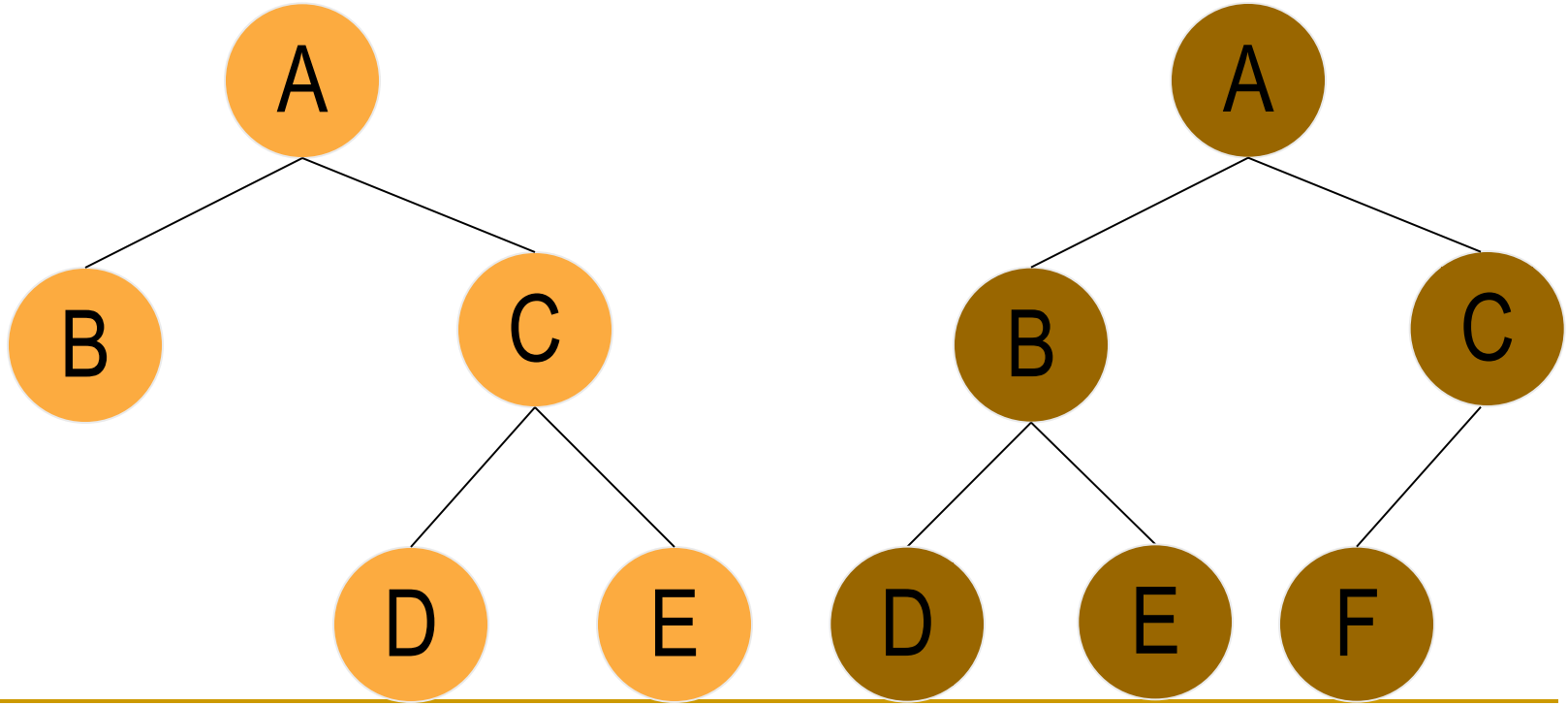
•Em relação a espaço (memória): a recursiva requer espaço extra para as chamadas recursivas. Esse espaço extra é diretamente proporcional (linear) à altura da árvore. (VERIFIQUE!)

Custo da busca em ABB

- Pior caso: como o número de passos é determinado pela altura da árvore, o pior caso é a árvore degenerada (altura = n).
- Altura da ABB depende da sequência de inserção das chaves...
  - Considere, p.ex., o que acontece se uma sequência ordenada de chaves é inserida...
  - Seria possível gerar uma árvore balanceada com essa mesma sequência, se ela fosse conhecida a priori. Como?
- Busca ótima: árvore de altura mínima (perfeitamente balanceada)
- Busca eficiente: árvore razoavelmente balanceada...(árvore balanceada)

Árvore Binária Balanceada

- Para cada nó, as alturas de suas duas sub-árvores diferem de, no máximo, 1



Árvore Binária Perfeitamente Balanceada

- O número de nós de suas sub-árvores esquerda e direita difere em, no máximo, 1
- É a árvore de altura mínima para o conjunto de chaves
- Toda AB Perfeitamente Balanceada é Balanceada, sendo que o inverso não é necessariamente verdade

Inserção (operações em ABB's)

- Passos do algoritmo de inserção:
  - Procure um “local” para inserir a nova chave, começando a procura a partir do nó-raiz:
  - Para cada nó-raiz, compare:
    - se a nova chave for menor do que o valor no nó-raiz, repita o processo para sub-árvore esquerda; ou
    - se a nova chave for maior que o valor no nó-raiz, repita o processo para sub-árvore direita.
  - Se um ponteiro (filho esquerdo/direito de um nó-raiz) nulo é atingido, coloque o novo nó como sendo raiz dessa sub-árvore vazia.
  - A inserção sempre se dá como nó folha: não exige deslocamentos!

Inserção

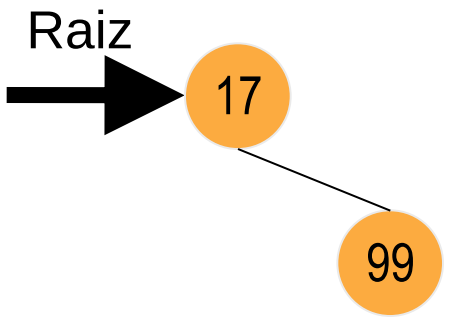
- Para entender o algoritmo, considere a inserção do conjunto de números, na seqüência

  {17, 99, 13, 1, 3, 100, 400}

- No início, a ABB está vazia!

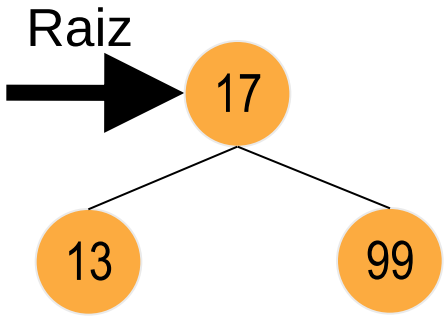
Inserção

- O número 17 será inserido tornando-se o nó raiz
- A inserção do 99 inicia-se na raiz. Compara-se 99 c/ 17.
- Como 99 > 17, 99 deve ser colocado na sub-árvore direita do nó contendo 17 (subárvore direita, inicialmente, nula)



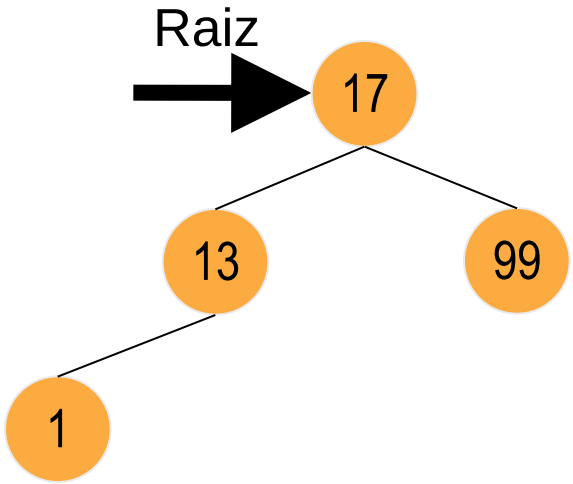
Inserção

- A inserção do 13 inicia-se na raiz
- Compara-se 13 c/ 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 como raiz dessa sub-árvore



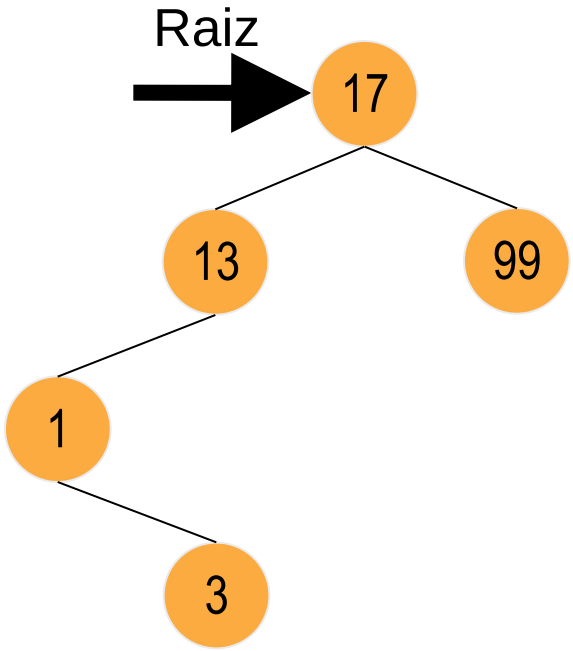
Inserção

- 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



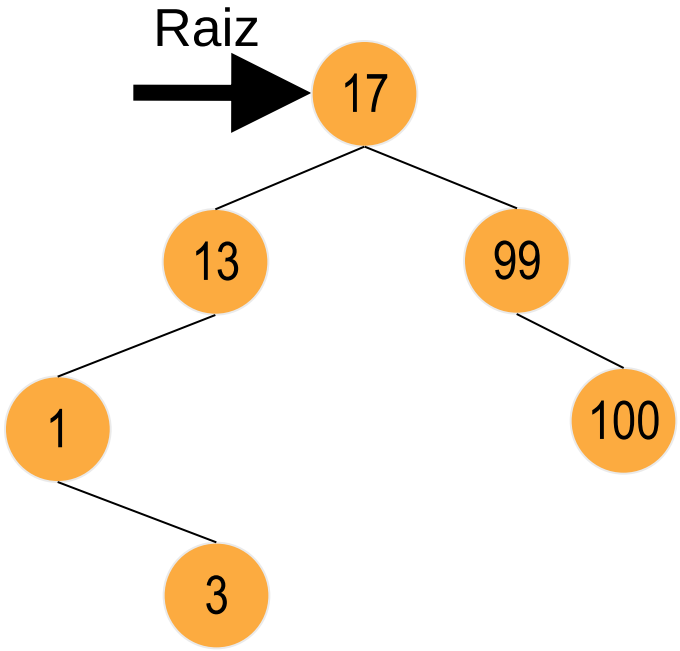
Inserção

- Repete-se o procedimento para inserir o elemento 3:
  - 3 < 17;
  - 3 < 13
  - 3 > 1



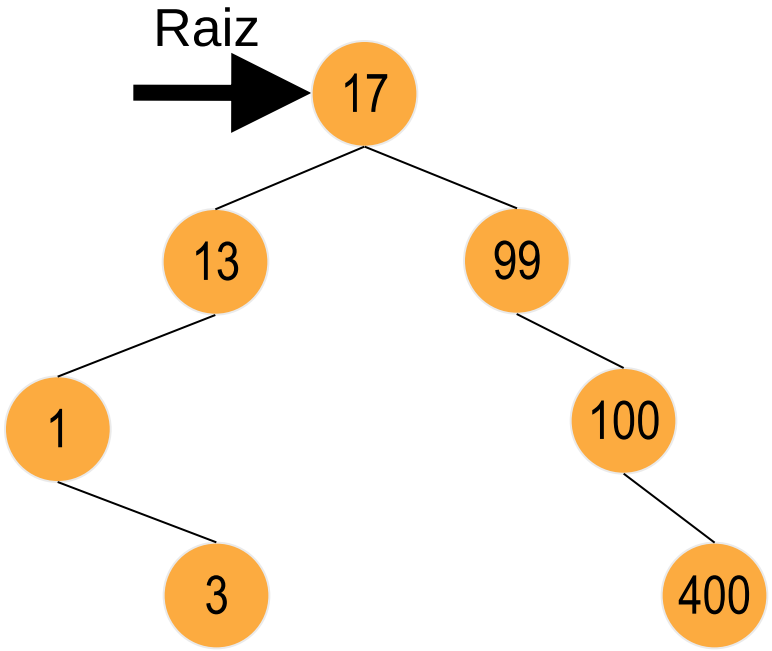
Inserção

- Repete-se o procedimento para inserir o elemento 100:
  - 100 > 17
  - 100 > 99



Inserção

- Repete-se o procedimento para inserir o elemento 400:
  - 400 > 17
  - 400 > 99
  - 400 > 100



Inserção

- O algoritmo de inserção não garante que a árvore resultante seja perfeitamente balanceada ou mesmo apenas balanceada

Função Recursiva de Inserção

- Seja uma função recursiva que insere uma chave x numa ABB, se ela já não estiver lá. Retorna o ponteiro para o nó que contém x.

- Essa função é usada para a construção de uma ABB.

```
pno busca_insere(tipo_elem x, tree raiz);
```

x: chave para inserir; raiz: raiz da árvore onde deve inserir;
busca_insere: retorna endereço onde está x

```
pno busca_insere(tipo_elem x, tree raiz){

    if (raiz == NULL) { /*inserir x como raiz da árvore*/
        raiz = malloc(sizeof(tree));
           raiz->info = x;
           raiz->esq = NULL;
           raiz->dir = NULL;
           return raiz;
    }

    if (x < raiz->info)
        return busca_insere(x, raiz->esq);

    if (x > raiz->info)
        return busca_insere(x, raiz->dir);

    return raiz;

}
```

Acrescente um parâmetro booleano para indicar se houve a inserção ou não.

Custo da Operação de Inserção

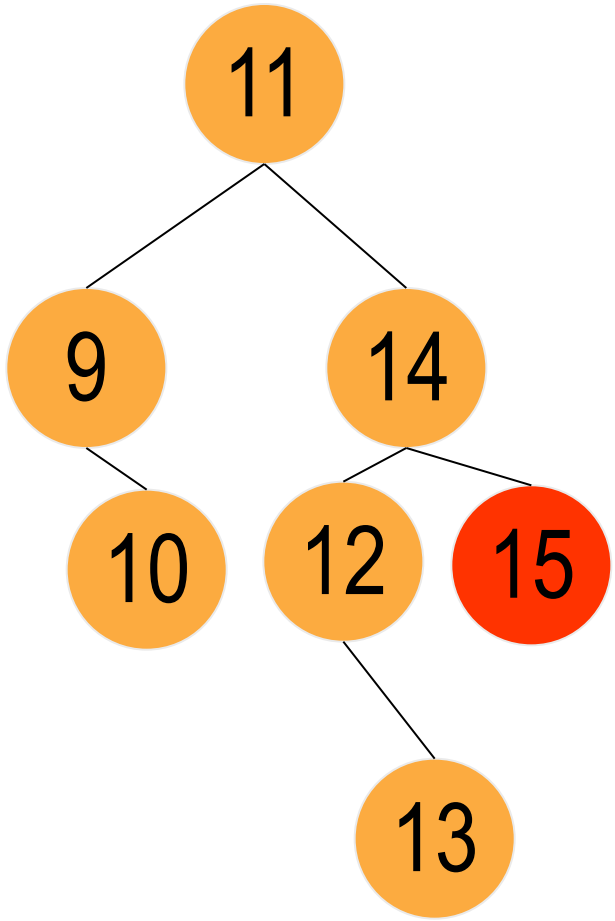
- A inserção requer uma busca pelo lugar da chave, portanto, com custo de uma busca qualquer (tempo proporcional à altura da árvore).
- O custo da inserção, após a localização do lugar, é constante; não depende do número de nós.
- Logo, tem complexidade análoga à da busca.

Remoção (operações em ABB's)

- 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.) pode substituir o nó eliminado;
  - Caso 3: o nó possui duas sub-árvores
    - O nó cuja chave seja a menor da sub-árvore direita pode substituir o nó eliminado; ou, alternativamente, o de maior valor da sub-árvore esquerda pode substituí-lo.

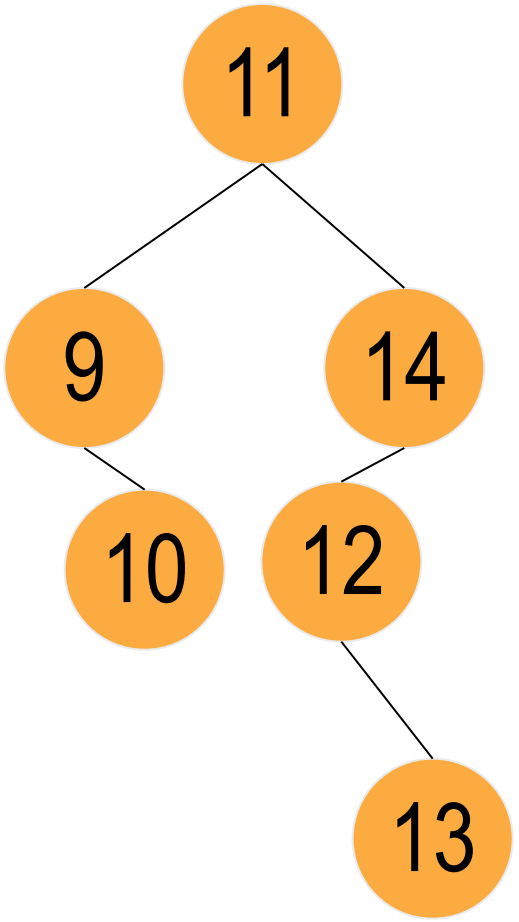
Remoção – Caso 1

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



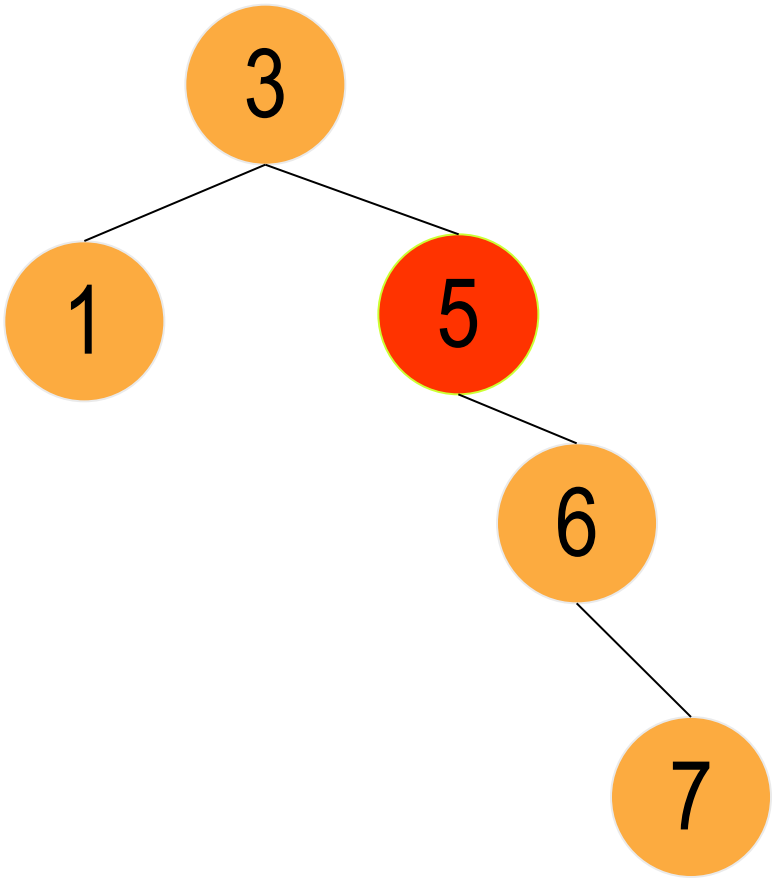
Remoção – Caso 1

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



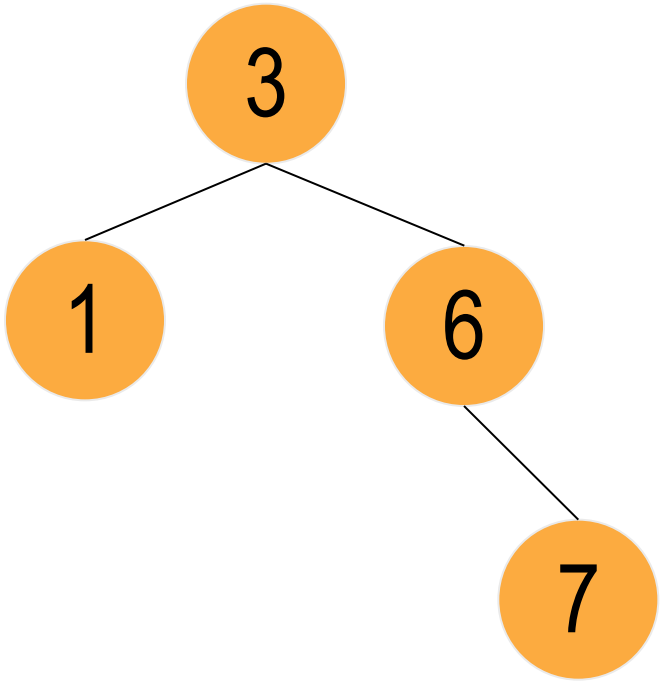
Remoção – Caso 2

- Removendo-se o nó com o valor 5
- Como ele possui uma sub-árvore direita, o nó contendo o valor 6 pode "ocupar" o lugar do nó removido



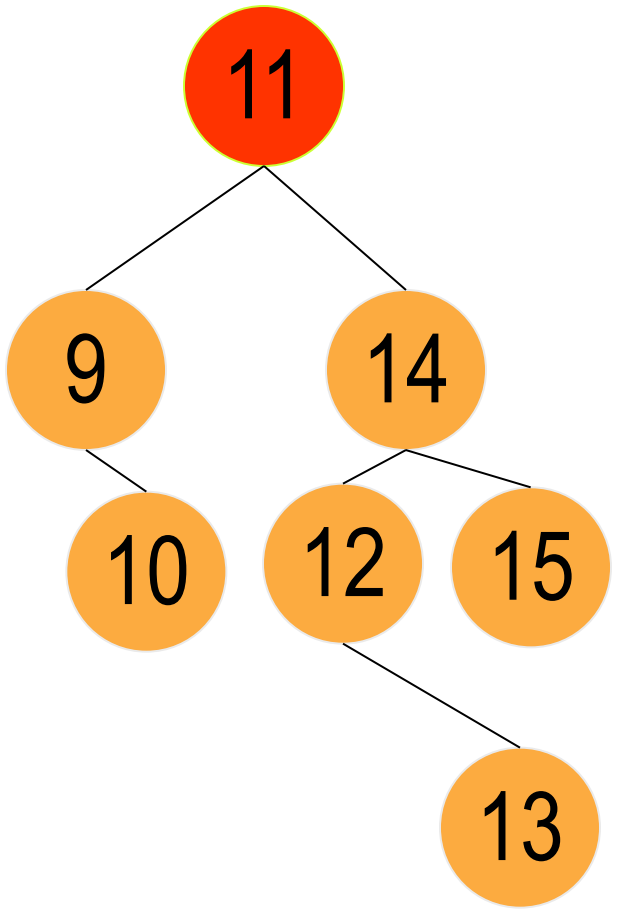
Remoção – Caso 2

- Caso existisse um nó com somente uma sub-árvore esquerda, seria análogo.



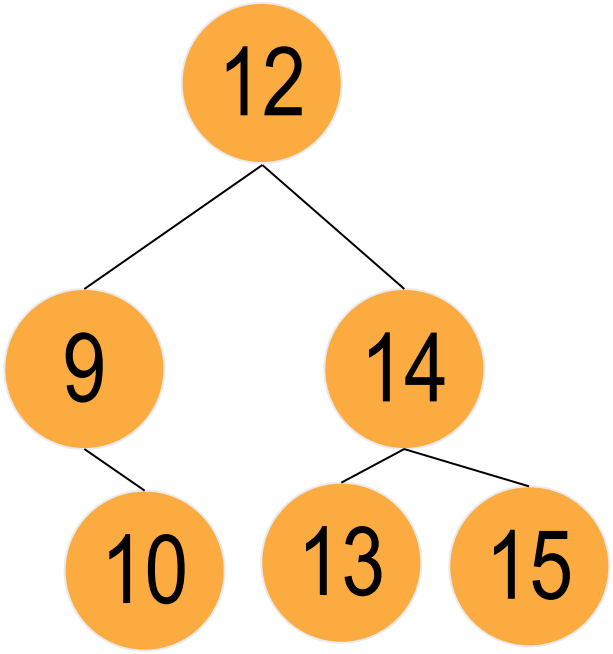
Remoção – Caso 3

- Eliminando-se o nó de chave 11
- Neste caso, existem 2 opções:
  - O nó com chave 10 pode "ocupar" o lugar do nó-raiz, ou
  - O nó com chave 12 pode "ocupar" o lugar do nó-raiz



Remoção – Caso 3

- Esse terceiro caso, também se aplica ao nó com chave 14, caso seja retirado.
  - Nessa configuração, os nós com chave 13 ou 15 poderiam ocupar seu lugar.



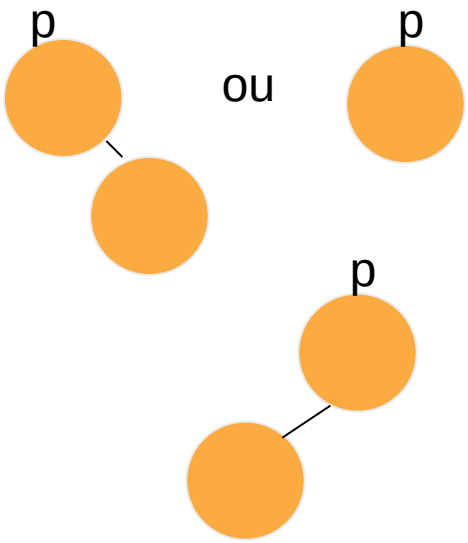
Considere o contexto no programa principal, para a remoção da chave x:

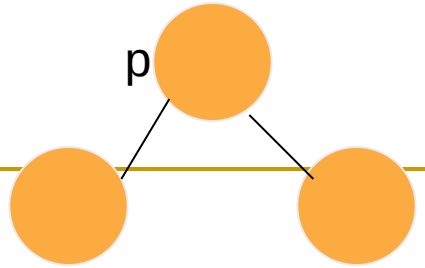
```
...
if (busca_remove(raiz, x))  printf("houve a remoção");
else printf("x não pertence à árvore");
...
```

```
boolean busca_remove(tree raiz, tipo_elem x){
   /*retorna true se removeu x; false, se x não estava na árvore*/

     if (raiz == NULL)
          /*árvore vazia; x não está na árvore*/
          return FALSE;

     if (raiz->info == x) {  /*achou x: eliminar*/
          removerNo(raiz);
          /*se alterar raiz no procedimento, altera aqui*/
          return TRUE;
     }

     if (raiz->info < x)
          /*buscar e remover na sub-árvore direira*/
          return busca_remove(raiz->dir, x);
     else
          /*buscar e remover na sub-árvore esquerda*/
          return busca_remove(raiz->esq, x);
}
```

Repare no efeito da passagem por endereço da variável raiz nas funções `removerNo()` e `busca_remove()`

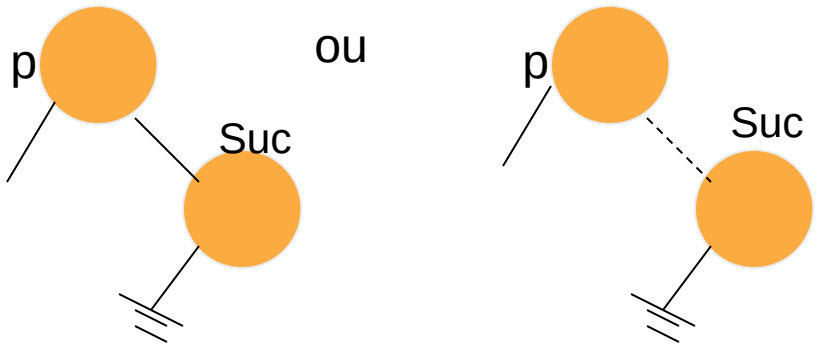
```
void removerNo(pno p){
    /* p é o ponteiro para o nó a ser removido; retorna em p
o ponteiro para o nó que o substituiu */

    pno q;

    if (p->esq == NULL){
        /* substitui por filho à direita */
        q = p;  p = p->dir;  free(q);

    } else if (p->dir == NULL) {
        /* substitui por filho à esquerda */
        q = p;  p = p->esq;  free(q);

    } else

        substituimenoradireita(p, p->dir);
      /*alternativamente:substituirmaioraesquerda(p,p->esq)*/
}
```

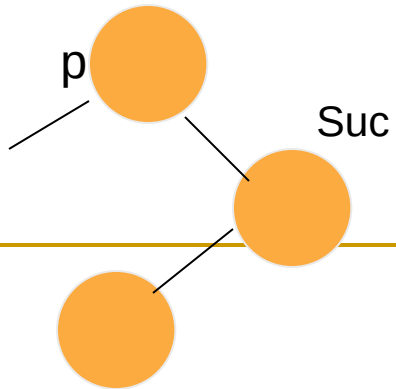




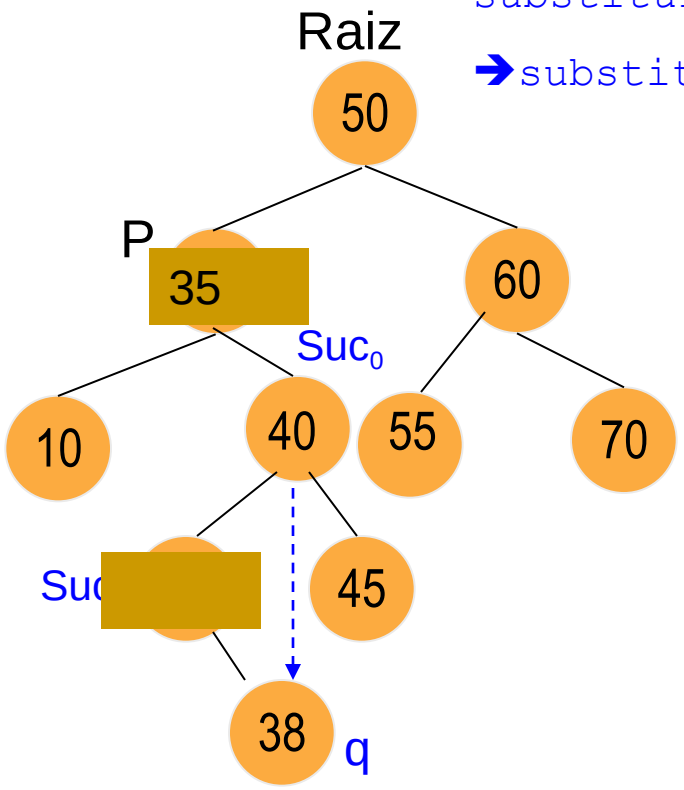
Repare que ao alterar p, alteramos o parâmetro atual (veja quem é ele na função busca_remove)

```
void substituimenoradireita(pno p, pno suc){
    /*encontra o sucessor de p, ou seja, o descendente
mais a esquerda da sub-arvore à direita de p. É um nó
terminal - seu conteúdo é copiado em q e ele será
removido*/

    pno q;

    if (suc->esq == NULL){
        p->info = suc->info;
        /*remover suc*/
        q = suc;
        suc = suc->dir; /*altera p->dir*/
        free(q);
    } else
        substituimenoradireita(p, suc->esq);
}
```





Exemplo



Suc$_0$

`substituimenoradireita(p, p->dir)`

➔`substituimenoradireita(p,`Suc$_0$`->esq)`

Suc$_1$

Custo da Operação de Remoção

- 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.

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 In-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 AVL