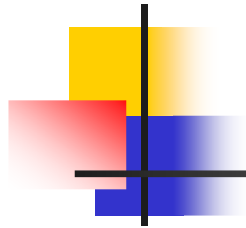




Algoritmos e Estruturas de Dados

Prof. Marcelo Zorzan
Profa. Melissa Zanatta



Situação-Problema

- Em alguns jogos de adivinhação a pessoa que detém a resposta escolhe um tema (ex: animal, cidade, etc) que é apresentado aos participantes do jogo. Em seguida essa pessoa só poderá responder sim ou não para perguntas formuladas pelos participantes. Ganha o jogo o participante que encontrar a resposta mais rápido.

Se você tivesse que representar esse jogo usando um tipo abstrato de dados, que estrutura você indicaria? Dê um exemplo.



Árvore - Aplicação

- Uma aplicação importante de árvores é na tomada de decisões.

→ Árvore de Decisão:

- *Definição*: é um instrumento de apoio à tomada de decisão que consiste numa representação gráfica das alternativas disponíveis geradas a partir de uma decisão inicial.
- *Vantagem*: possibilita que um problema complexo seja decomposto em diversos sub-problemas mais simples.

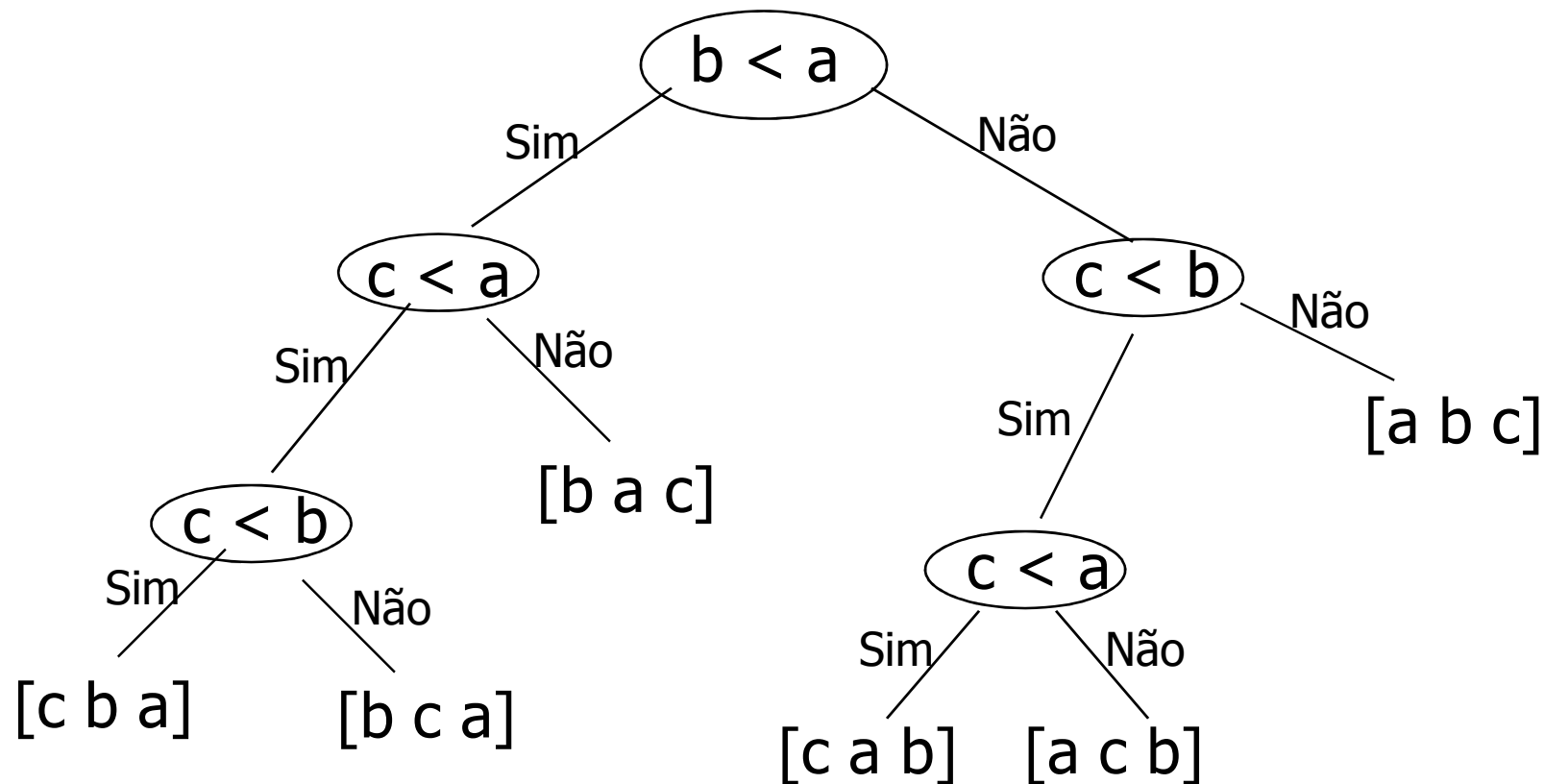


Árvore - Aplicação

- A representação gráfica da árvore de decisão utiliza linhas para identificar a decisão (por ex. "sim" ou "não") e nós para identificar as questões sobre as quais se deve decidir.
- Cada um dos ramos formador por linhas e nós termina numa espécie de folha que identifica a consequência mais provável da sequência de decisões tomadas.

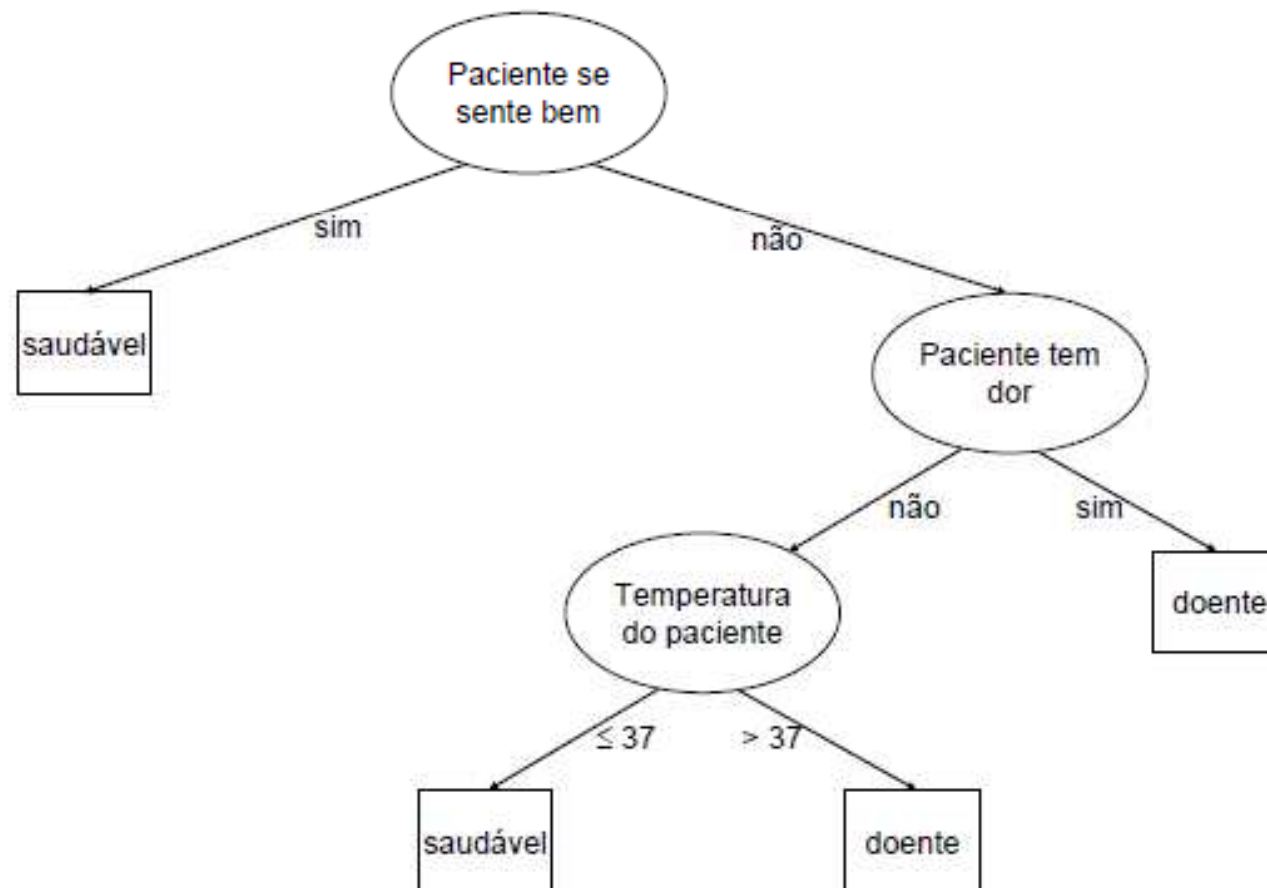
Árvore - Aplicação

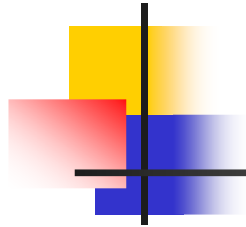
- Exemplo 1: Ordenação



Árvore - Aplicação

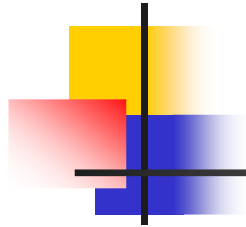
- Exemplo 2:





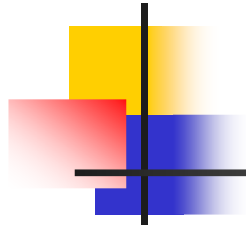
Aula de Hoje

- Árvore Binária de Pesquisa



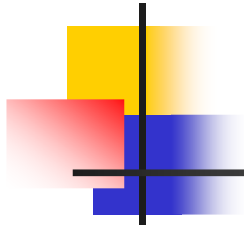
Definição

- Uma Árvore Binária de Pesquisa possui as mesmas propriedades de uma Árvore Binária, acrescida das seguintes propriedades:
 - Os nós pertencentes à sub-árvore esquerda possuem valores menores do que o valor associado ao nó-raiz
 - Os nós pertencentes à sub-árvore direita possuem valores maiores do que o valor associado ao nó-raiz
 - Um percurso *em-ordem* nessa árvore resulta na sequência de valores em ordem crescente



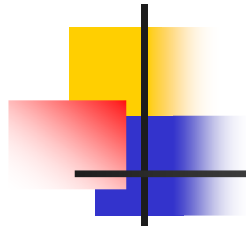
Definição

- Se invertermos 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 sequência de inserção dos dados



Definição

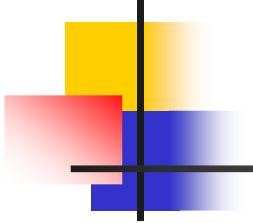
- Uma árvore binária de pesquisa é dinâmica e pode sofrer alterações (inserções e remoções de nós) após ter sido criada
- Operações em árvores binárias de pesquisa:
 - Definição da estrutura de dados árvore
 - Inicializar
 - Inserir
 - Pesquisar
 - Remover



Árvore Binária de Pesquisa

```
typedef struct sNo
{
    int info;
    struct sNo* esq;
    struct sNo* dir;
}NO;
```

```
typedef struct sArvBinPesq
{
    NO* ptRaiz;
}ArvBinPesq;
```



Operação – inicializar

```
void inicializar(NO** raiz) {  
    *raiz = NULL;  
}
```



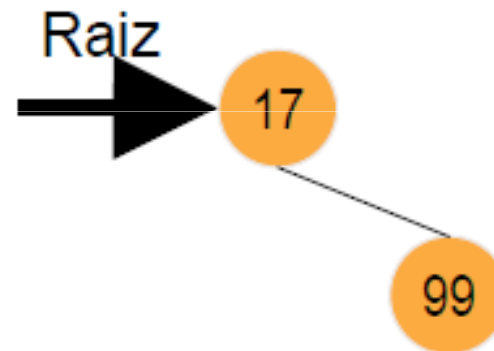
Operação – inserir

- Para entender o algoritmo considere a inserção do conjunto de números, na sequência
 $\{17, 99, 13, 1, 3, 100, 400\}$
- No início a árvore binária de pesquisa está vazia!



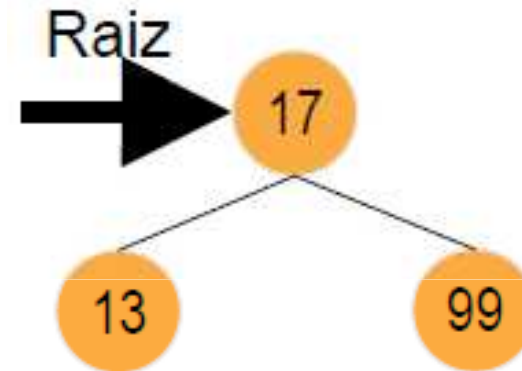
Operação – inserir

- O número 17 será inserido tornando-se o nó raiz
- A inserção do 99 inicia-se na raiz. Compara-se 99 com 17.
- Como $99 > 17$, ele deve ser colocado na sub-árvore direita do nó contendo 17



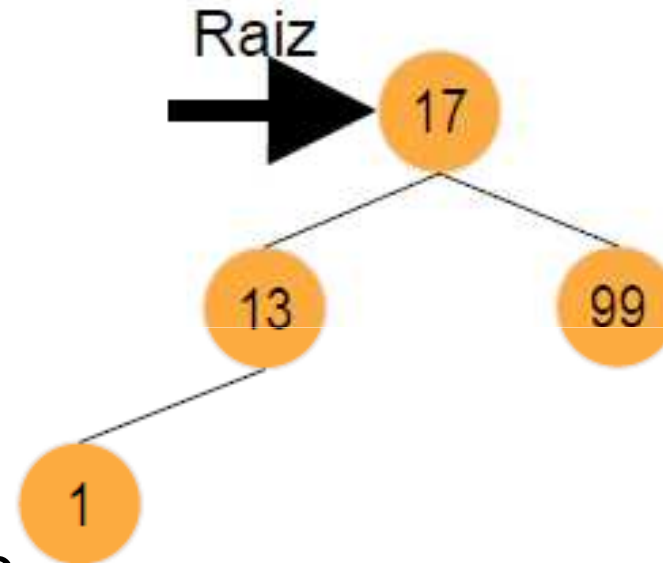
Operação – inserir

- A inserção do 13 inicia-se na raiz
- Compara-se 13 com 17. Como $13 < 17$, ele 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



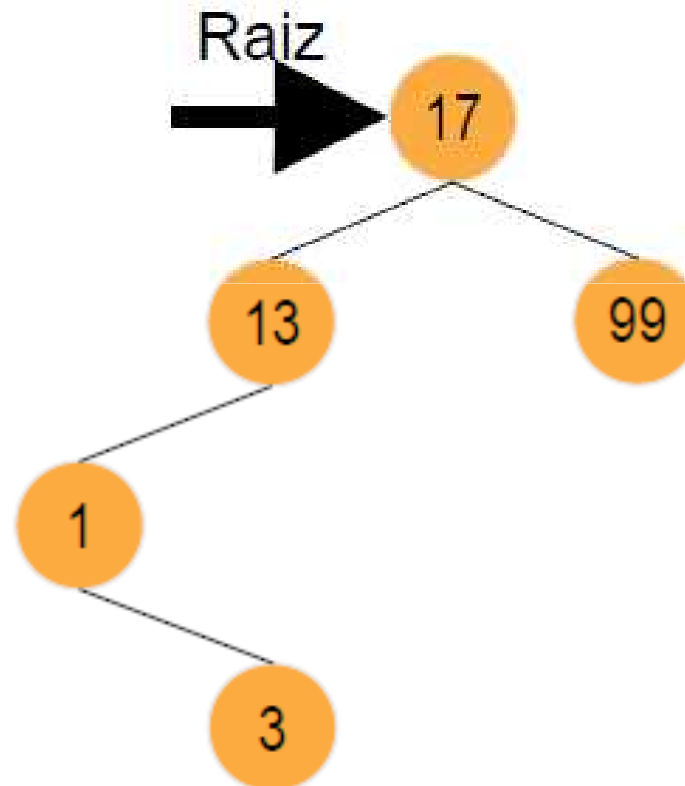
Operação – inserir

- 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



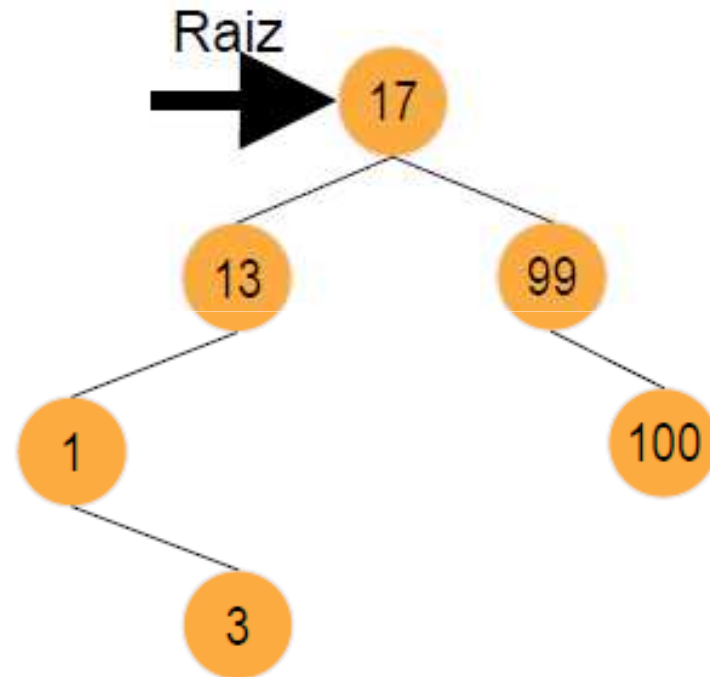
Operação – inserir

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



Operação – inserir

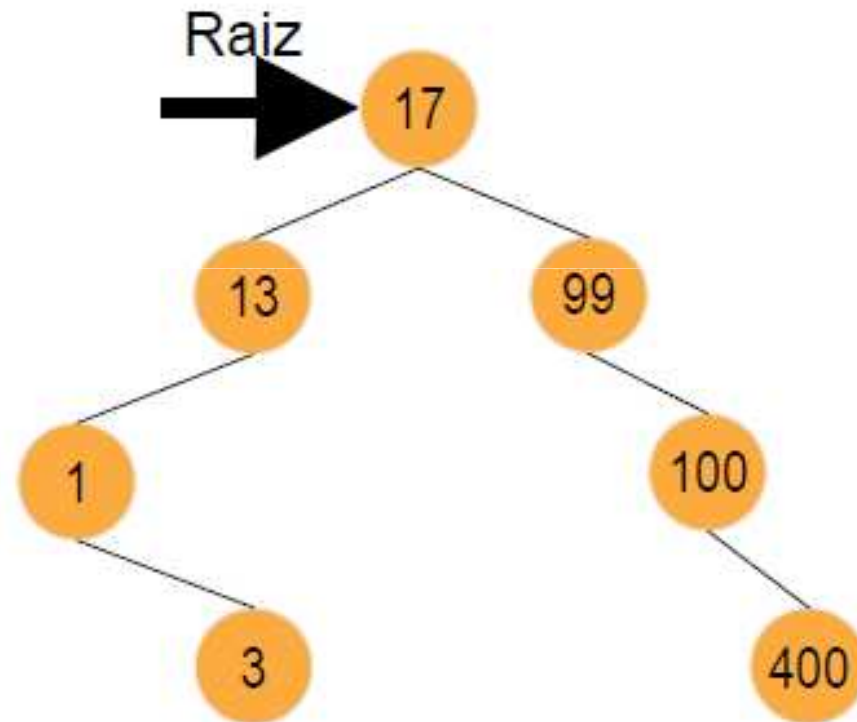
- Repete-se o procedimento para inserir o elemento 100:
 - ❑ $100 > 17$
 - ❑ $100 > 99$



Operação – inserir

- Repete-se o procedimento para inserir o elemento 400:

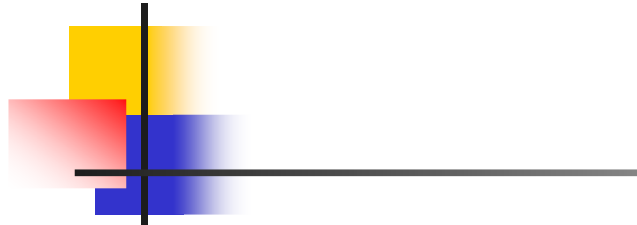
- ❑ $400 > 17$
- ❑ $400 > 99$
- ❑ $400 > 100$





Operação – inserir (implementação)

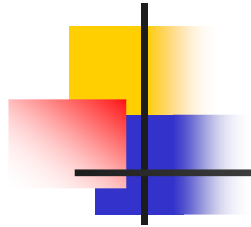
```
void inserir(NO** raiz, int x){  
    NO* aux = *raiz;  
    NO* aux2 = NULL;  
    NO* novo;  
  
    novo = (NO*) malloc(sizeof(NO));  
    if(novo == NULL)  
    {  
        printf("\nErro. Memoria nao alocada");  
        return;  
    }  
  
    novo ->info = x;  
    novo ->esq = NULL;  
    novo ->dir = NULL;
```



inserir (cont.)

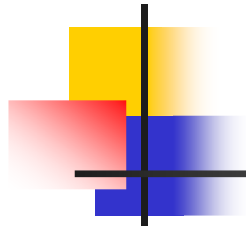
```
while(aux != NULL)
{
    aux2 = aux;
    if(x < aux->info)
        aux = aux -> esq;
    else
        aux = aux -> dir;
}

if(aux2 == NULL)
    *raiz = novo;
else
{
    if(x < aux2->info)
        (aux2)-> esq = novo;
    else
        (aux2)-> dir = novo;
}
}
```



Operação – inserir

Desenvolva uma função recursiva para a operação de inserção.



Operação – pesquisar

- 1) Comece a pesquisar a partir do nó-raiz;
- 2) Para cada nó-raiz de uma sub-árvore compare:
 - se o valor procurado é menor que o valor no nó-raiz
(continue pela sub-árvore esquerda)
 - se o valor é maior que o valor no nó-raiz
(continue pela sub-árvore direita);
- 3) Caso o nó contendo o valor pesquisado seja encontrado, retorne um ponteiro para o nó; caso contrário, retorne um ponteiro nulo.



Operação – pesquisar (implementação)

```
NO* pesquisar(NO* raiz, int k) {  
    NO* aux;  
    aux = raiz;  
    if (aux == NULL)  
        return NULL;  
    else if (aux->info > k)  
        return pesquisar(aux->esq, k);  
    else if (aux->info < k)  
        return pesquisar(aux->dir, k);  
    else  
        return aux;  
}
```



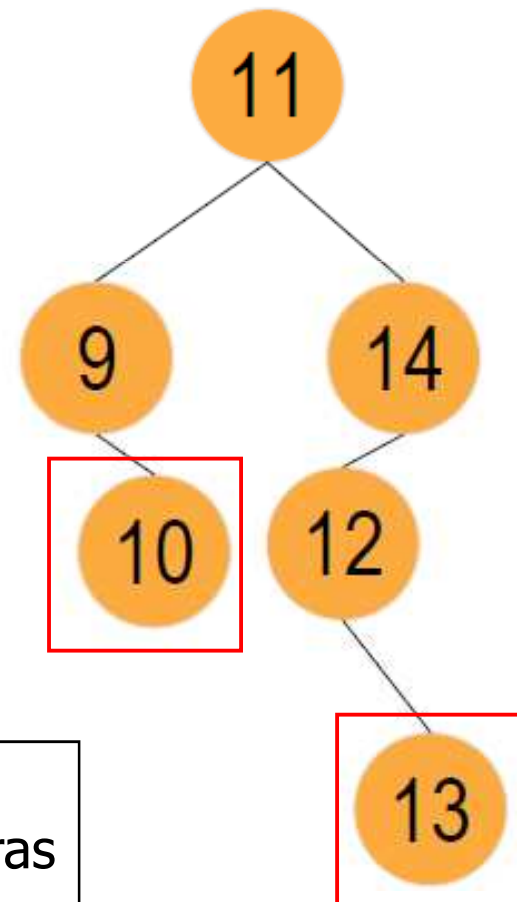

Operação – remover

- Casos a serem considerados :
 - Caso 1: o nó é folha
 - O nó pode ser removido sem problema
 - Caso 2: o nó possui uma sub-árvore (esq ou dir)
 - O nó raiz da sub-árvore (esq ou dir.) “ocupa” o lugar do nó removido
 - Caso 3: o nó possui duas sub-árvores (esq e dir)
 - O nó contendo o menor valor da sub-árvore direita pode “ocupar” o lugar; ou o maior valor da sub-árvore esquerda pode “ocupar”

Operação – remover

- Remover – Caso 1

Os nós com os valores 10 e 13 podem ser removidos sem necessidade de reajuste.

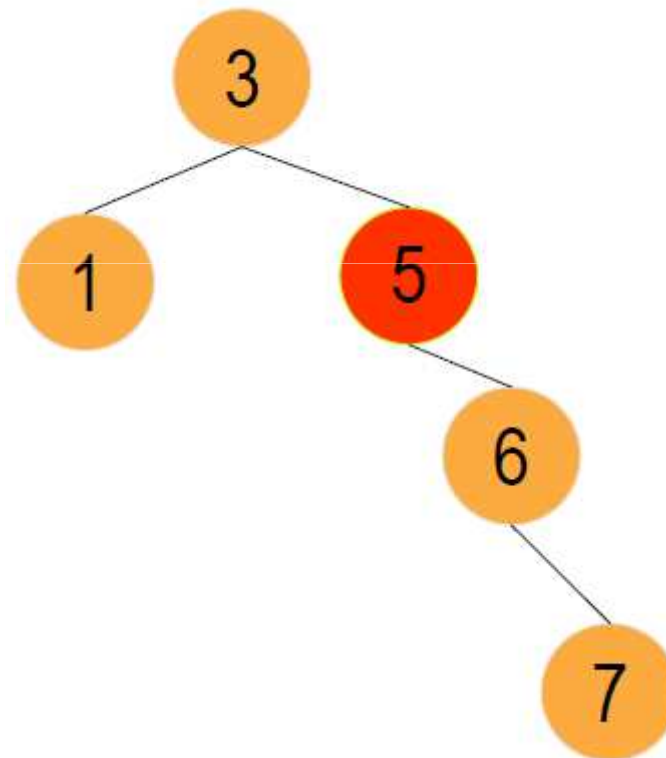


“Quando o nó a ser removido é folha, o mesmo pode ser retirado da árvore sem outras alterações”

Operação – remover

- Remover – Caso 2

Como o elemento com valor 5 possui uma sub-árvore direita, o nó contendo o valor 6 irá “ocupar” o lugar do nó removido



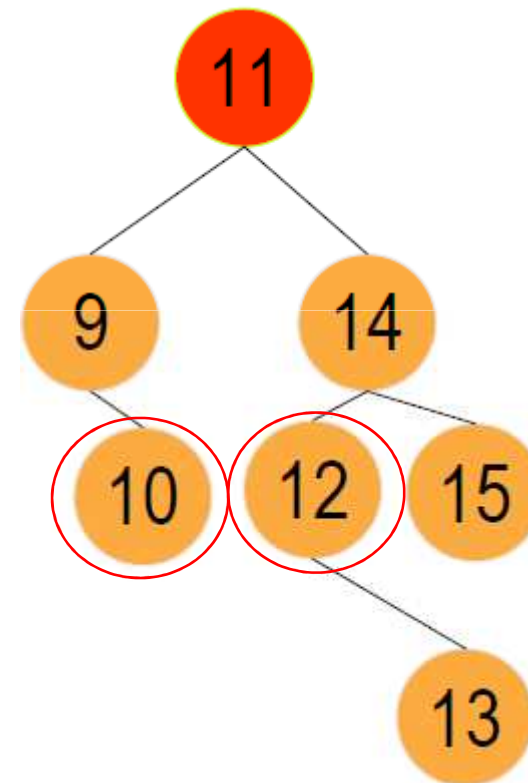
“O nó raiz da sub-árvore (esq ou dir.) “ocupa” o lugar do nó removido”

Operação – remover

- Remover – Caso 3

Neste caso, existem 2 opções:

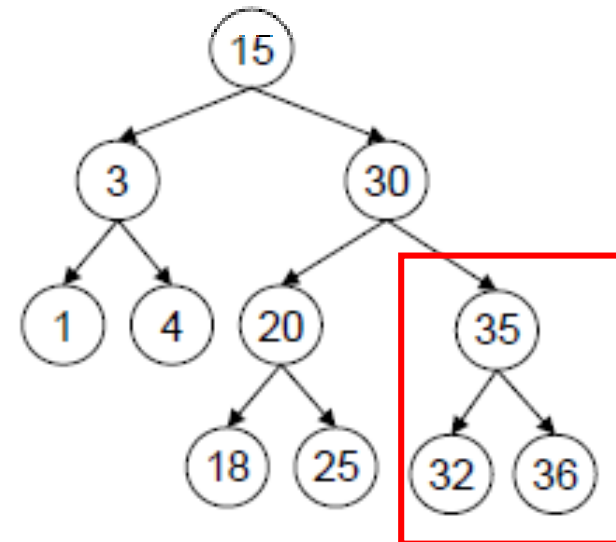
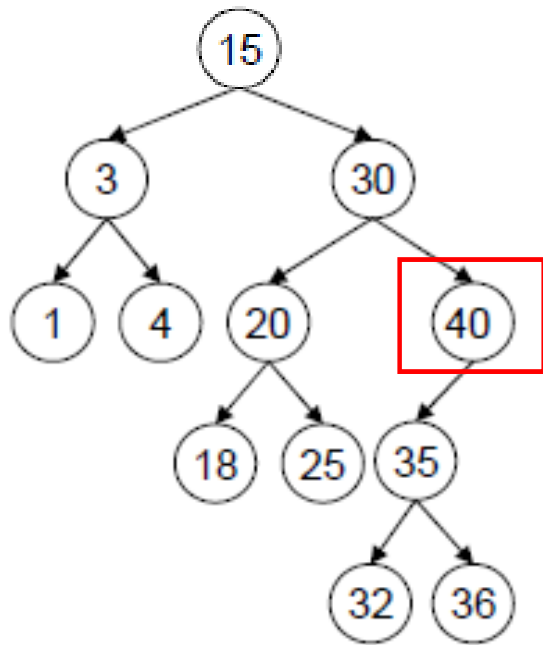
- 1) O nó com chave 10 pode “ocupar” o lugar do nó raiz,
- 2) O nó com chave 12 pode “ocupar” o lugar do nó-raiz



“O nó contendo o menor valor da sub-árvore direita pode “ocupar” o lugar; ou o maior valor da sub-árvore esquerda pode “ocupar” “

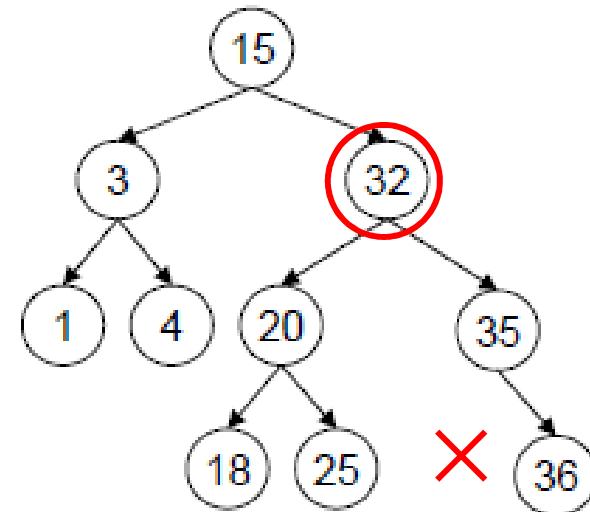
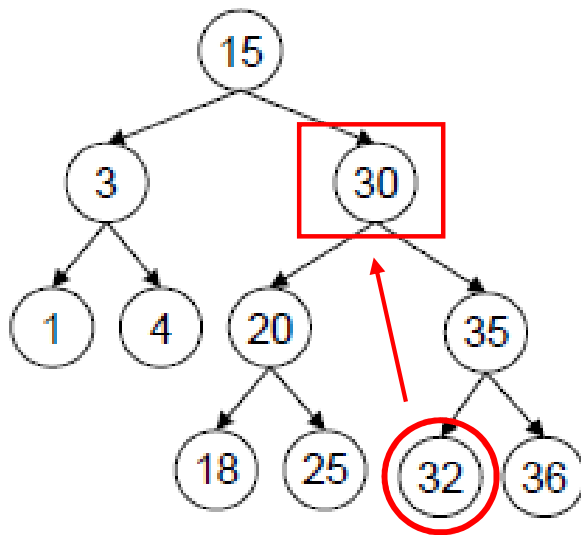
Operação – remover

Remover o elemento 40



Operação – remove

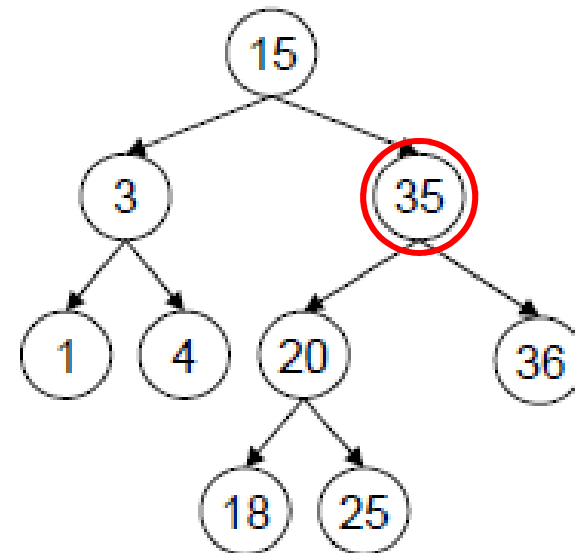
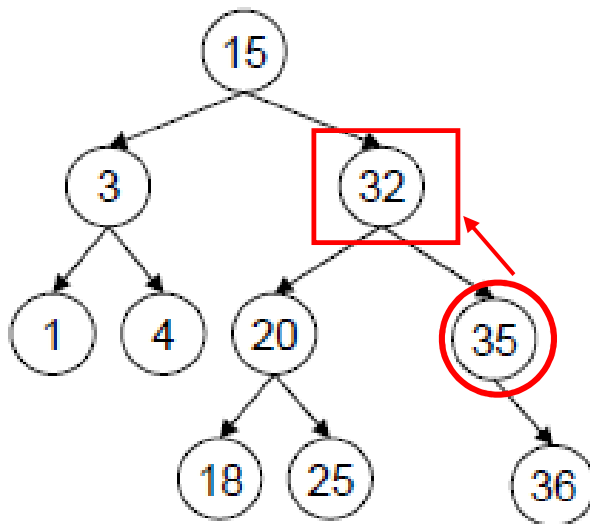
Remover o elemento 30



Qual seria a outra opção?

Operação – remover

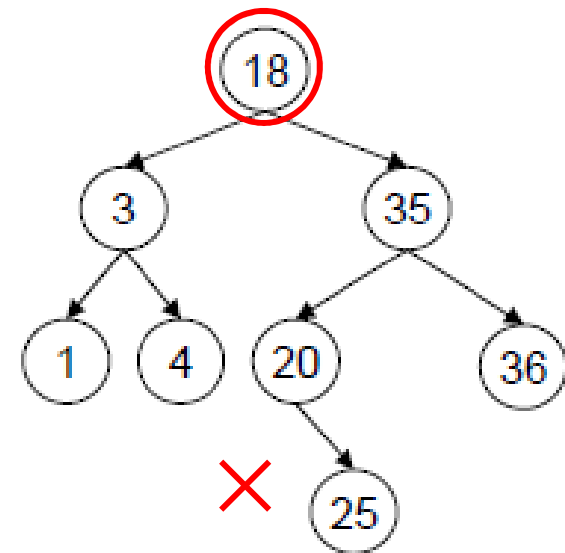
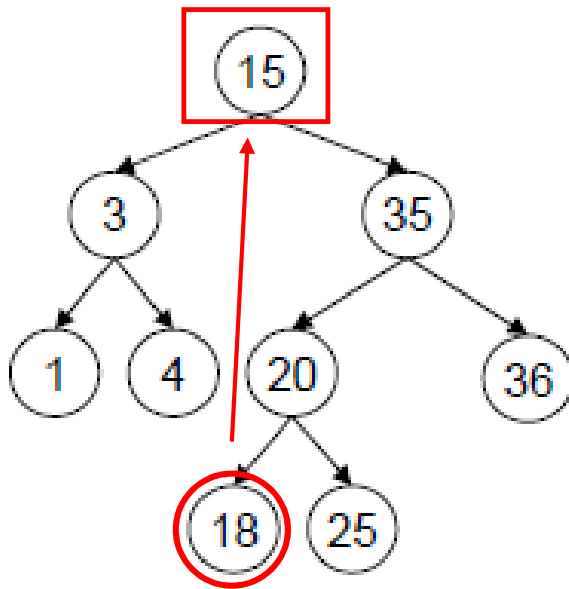
Remover o elemento 32



Qual seria a outra opção?

Operação – remover

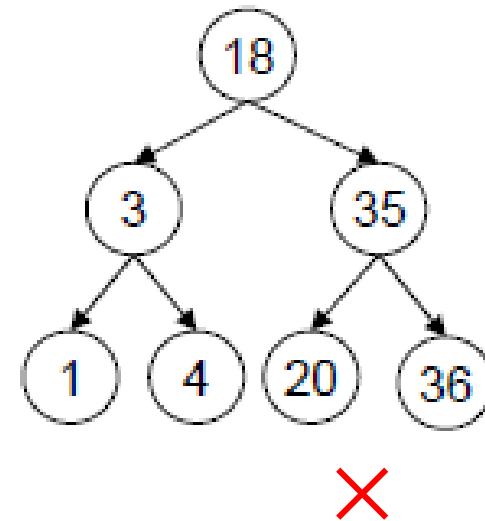
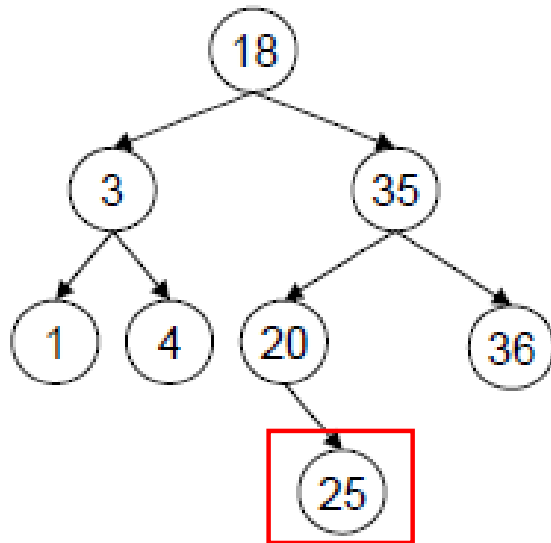
Remover o elemento 15



Qual seria a outra opção?

Operação – remover

Remover o elemento 25





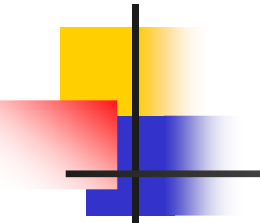
Operação – remover

→ Revisão: quando o elemento **x** possui as duas subárvores não vazias, há duas estratégias possíveis para substituir o valor de **x**, preservando as propriedades de árvore binária de pesquisa:

❑ Encontrar o elemento de **menor** valor **y** na subárvore **direita** de **x** e transferi-lo para o nó ocupado por **x**

❑ Encontrar o elemento de **maior** valor **y** na subárvore **esquerda** de **x** e transferi-lo para o nó ocupado por **x**

Operação – remover (implementação)

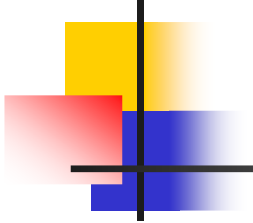


```
NO* remover(NO *raiz, int k) {  
    if (raiz == NULL)  
        return NULL;  
    else if (raiz->info > k)  
        raiz->esq = remover(raiz->esq, k);  
    else if (raiz->info < k)  
        raiz->dir = remover(raiz->dir, k);  
}
```

else

```
{ /* achou o elemento */  
  if (raiz->esq == NULL && raiz->dir == NULL) /* elemento sem filhos */  
  {  
    free (raiz);  
    raiz = NULL;  
  }  
  else if (raiz->esq == NULL) /* só tem filho à direita */  
  {  
    NO* temp = raiz;  
    raiz = raiz->dir;  
    free(temp);  
  }  
  else if (raiz->dir == NULL) /* só tem filho à esquerda */  
  {  
    NO* temp = raiz;  
    raiz = raiz->esq;  
    free(temp);  
  }  
}
```

Operação – remover (implementação)



```
else
{ /* tem os dois filhos */
    NO* Pai = raiz;
    NO* F = raiz->esq;
    while (F->dir != NULL) {
        Pai = F;
        F = F->dir;
    }
    raiz->info = F->info; /* troca as informações */
    F->info = k;
    raiz->esq = remove(raiz->esq,k) ;
}
}
return raiz;
}
```