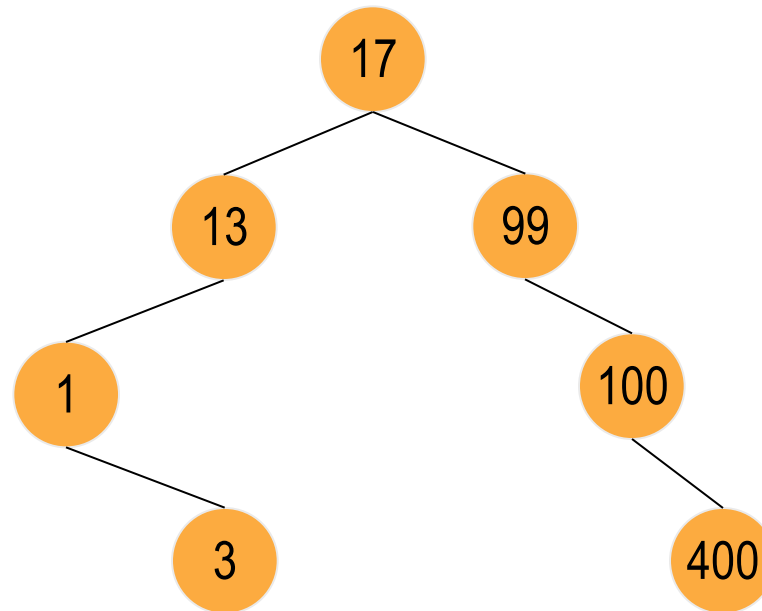

Árvores Binárias de Busca (ABB)

18/11

Definição

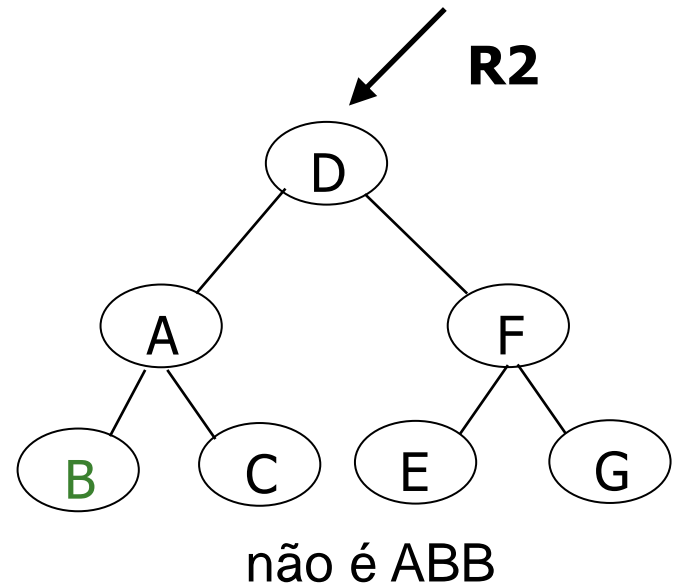
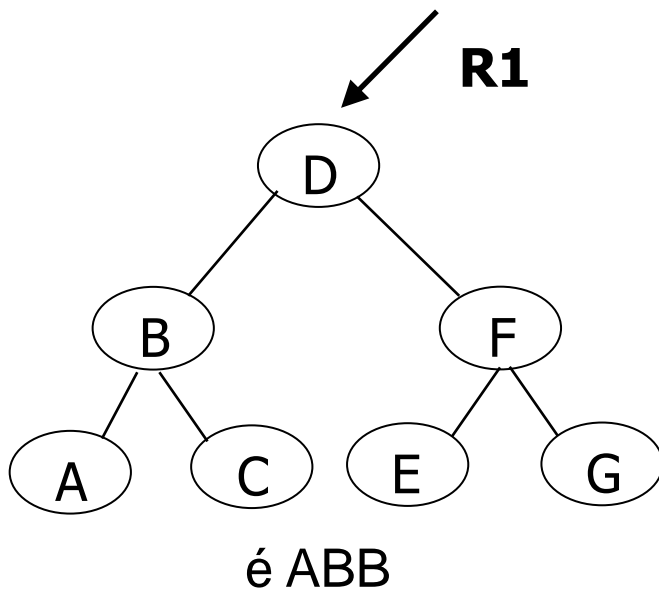
- Uma Árvore Binária de Busca possui as mesmas propriedades de uma AB, acrescida da seguintes propriedade:
 - Para todo nó da árvore, se seu valor é X , então:
 - Os nós pertencentes a sua sub-árvore esquerda possuem valores menores do que X ;
 - Os nós pertencentes a sua sub-árvore direita possuem valores maiores do que X .
 - Não há elementos duplicados
 - Um percurso **in-ordem** nessa árvore resulta na seqüência de valores em ordem crescente
- Também chamadas de “árvores de pesquisa” ou “árvores ordenadas”

Exemplo: ABB com chave integer



- In-Ordem: 1, 3, 13, 17, 99, 100, 400

Exemplos: ABB com chave char



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 **in-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 **seqüência de inserção** dos dados

ABB

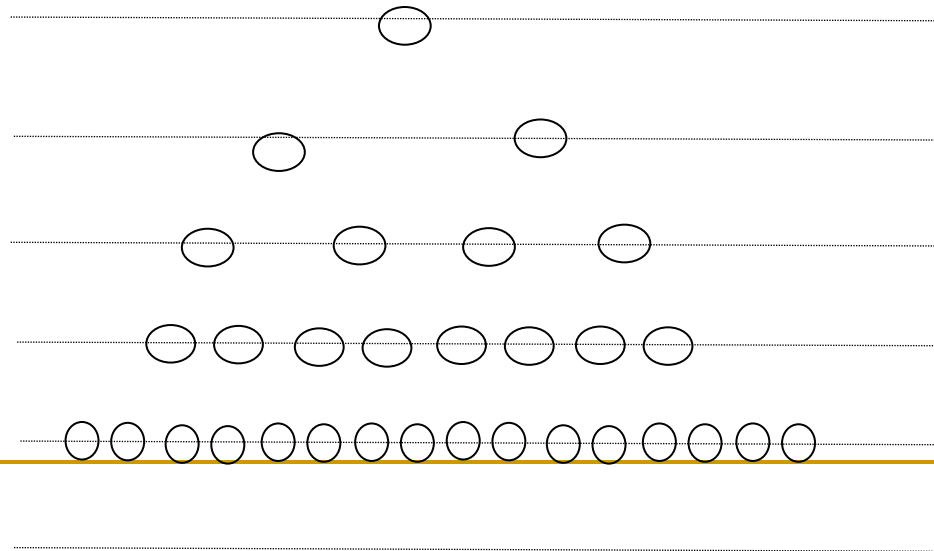
- Por que uma ABB é boa?
 - Imagine a situação
 - Sistema de votação por telefone (exemplo do programa antigo “Você decide” ou Big Brother)
 - Cada número só pode votar uma vez
 - Um sistema deve armazenar todos os números que já ligaram
 - A cada nova ligação, deve-se consultar o sistema para verificar se aquele número já votou; o voto é computado apenas se o número ainda não votou
 - A votação deve ter resultado on-line
-

ABB

- Por que uma ABB é boa?
 - Solução com ABBs
 - Cada número de telefone é armazenado em uma ABB
 - Suponha que em um determinado momento, a ABB tenha 1 milhão de telefones armazenados
 - Surge nova ligação e preciso saber se o número está ou não na árvore (se já votou ou não)
-

ABB

- Por que uma ABB é boa?
- Considere uma ABB com chaves uniformemente distribuídas (árvore cheia)

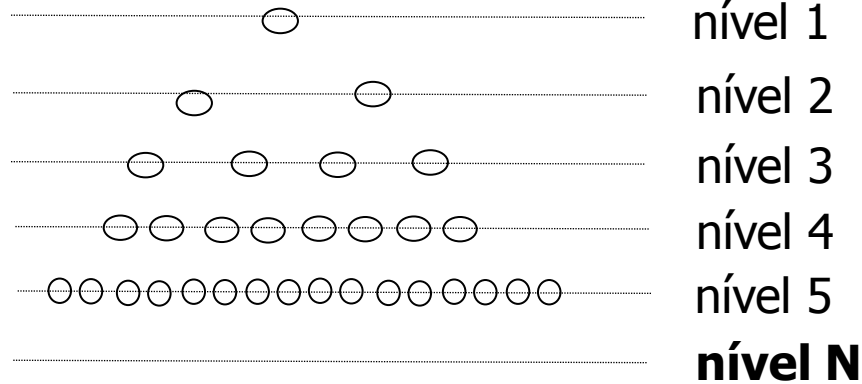


ABB

- Por que uma ABB é boa?
 - Resposta
 - Quantos elementos cabem em uma árvore de N níveis, como a anterior?
 - Como achar um elemento em uma árvore assim a partir da raiz?
 - Quantos nós se tem que visitar, no máximo, para achar o telefone na árvore, ou ter certeza de que ele não está na árvore?
-

ABB

■ Por que uma ABB é boa?



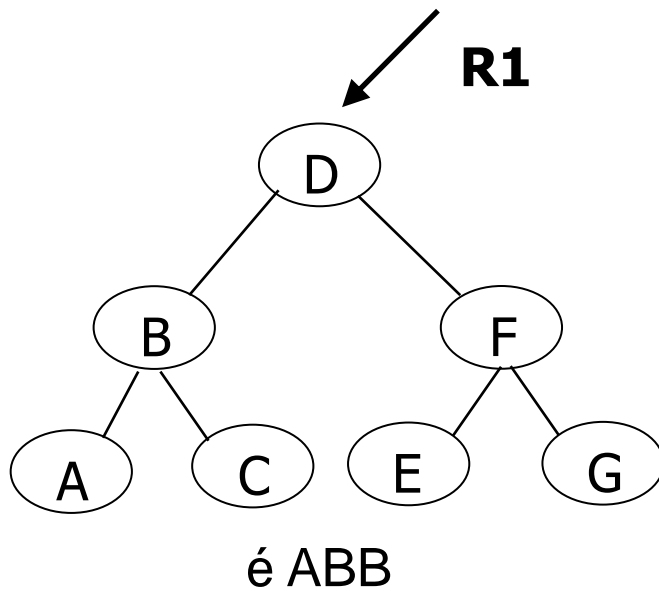
Nível	Quantos cabem
1	1
2	3
3	7
4	15
...	...
N	$2^N - 1$
10	1.024
13	8.192
16	65.536
18	262.144
20	1 milhão
30	1 bilhão
	...

ABB

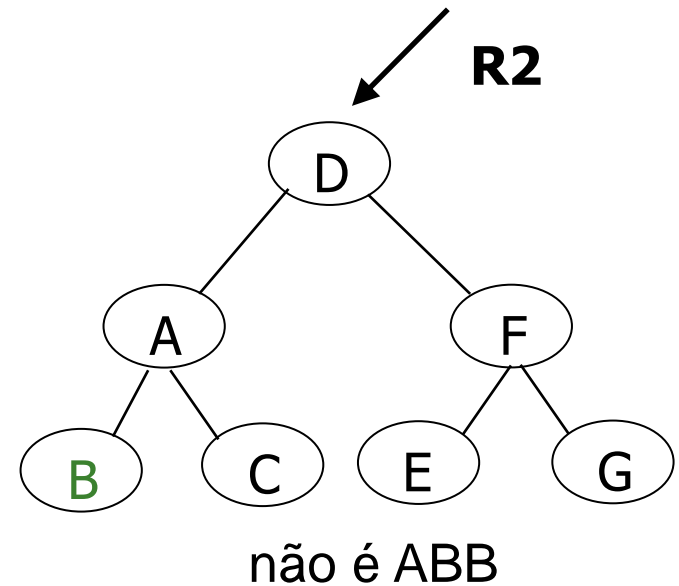
- Por que uma ABB é boa?
- Para se buscar em uma ABB
 - Em cada nó, compara-se o elemento buscado com o elemento presente
 - Se menor, percorre-se a subárvore esquerda
 - Se maior, percorre-se subárvore direita
 - Desce-se verticalmente até as folhas, no pior caso, sem passar por mais de um nó em um mesmo nível
 - Portanto, no pior caso, a busca passa por tantos nós quanto for a altura da árvore

ABB

- Exemplo: busca pelo elemento **E** nas árvores abaixo



3 consultas



6 consultas

ABB

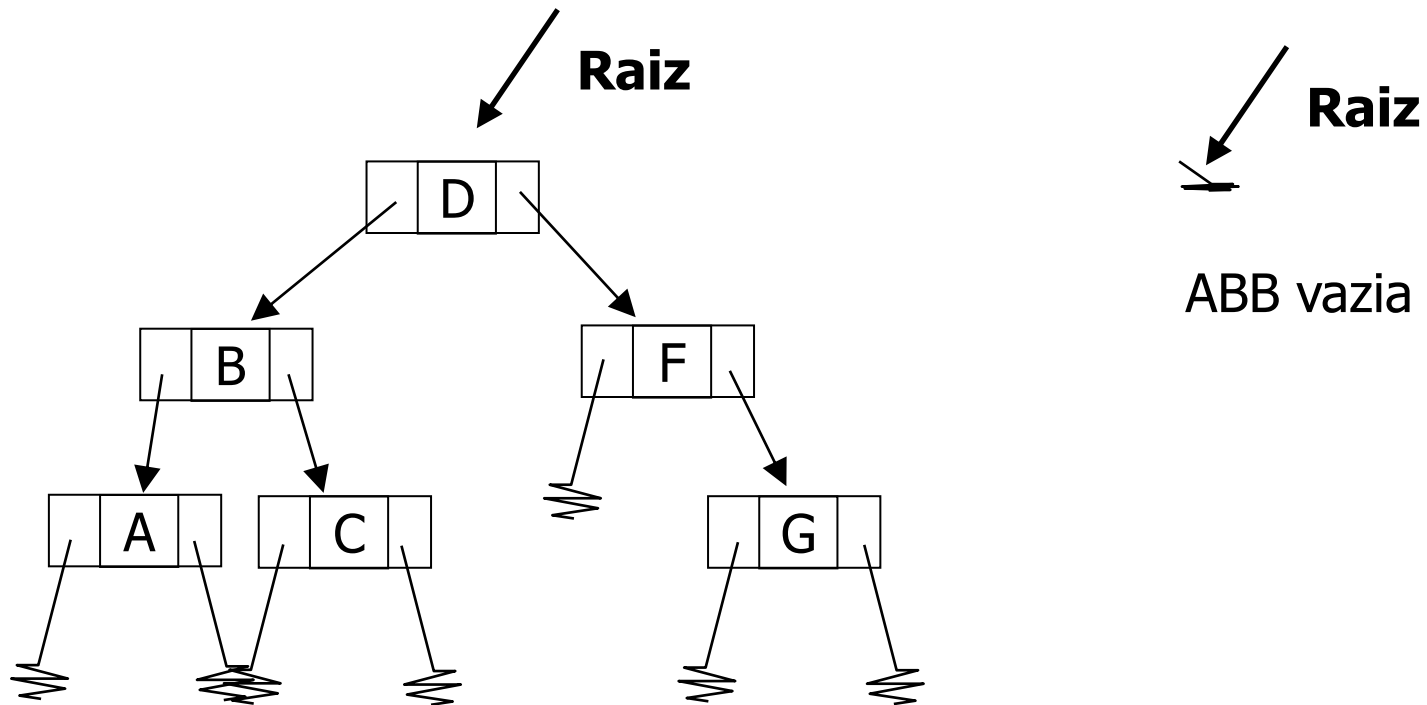
- Por que uma ABB é boa?
 - Buscas muito rápidas!!!

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 $O(\log_2 n)$ – no caso de árvore balanceada**

ABB

- Representação Encadeada Dinâmica



ABB

■ Declaração

```
typedef int elem;
```

```
// deve haver em problemas reais
```

```
typedef char *tipo_chave;
```

```
typedef struct arv *Arv;
```

ABB.h

Por simplicidade, nos algoritmos a seguir usaremos o campo info, como sendo a chave de busca.

```
struct arv {
```

```
// este campo serve para as comparações em programas reais
```

```
tipo_chave chave;
```

```
// dados associados aa chave
```

```
elem info;
```

```
struct arv* esq;
```

```
struct arv* dir;
```

ABB.c

```
};
```


Operações Básicas em ABB's

- **TAD ABB:**
- Definir (igual a AB)
- Remover
- Inserir
- Imprimir elementos em seqüência (igual a AB)
- Busca
- Destruir (igual a AB)

ABB

- **Operações** sobre a ABB
 - Devem considerar a ordenação dos elementos da árvore
 - Por exemplo, na **inserção**, deve-se **procurar pelo local certo** na árvore para se inserir um elemento
- **Exercício**
 - **Prática com o TAD (exemplos)**
 - Construa a partir do início uma ABB com os elementos K, E, C, P, G, F, A, T, M, U, V, X, Z

TAD ABB

■ Busca

- Comparando o parâmetro “chave” com a informação no nó “raiz”, 4 casos podem ocorrer:
 - A árvore é vazia \Rightarrow a chave não está na árvore \Rightarrow fim do algoritmo
 - Elemento da raiz = chave \Rightarrow achou o elemento (está no nó raiz) \Rightarrow fim do algoritmo
 - Chave < elemento da raiz \Rightarrow chave pode estar na subárvore esquerda
 - Chave > elemento da raiz \Rightarrow chave pode estar na subárvore direita
- Pergunta: quais os casos que podem ocorrer para a subárvore esquerda? E para a subárvore direita?

Os mesmos!

TAD ABB

■ Exercício

- Implementação da sub-rotina de busca de um elemento valor na árvore
- Retorna NULL se não achou (insucesso) ou o ponteiro do elemento valor (sucesso) no caso de desejarmos acessar os campos associados ao campo chave
- Arv busca(Arv p, elem valor);

```
Arv busca(Arv p, elem valor){  
  
    if (p == NULL)  
        return NULL;  
    if (valor == p->info)  
        return p;  
    if (valor < p->info)  
        return busca(p->esq, valor);  
    else  
        return busca(p->dir, valor);  
}
```

Inserção

- Passos do algoritmo de inserção
 - Procure um “local” para inserir o novo nó, começando a procura a partir do nó-raiz;
 - Para cada nó-raiz de uma sub-árvore, compare; se o novo nó possui um valor menor do que o valor no nó-raiz (vai para sub-árvore esquerda), ou se o valor é maior que o valor no nó-raiz (vai para sub-árvore direita);
 - Se um ponteiro (filho esquerdo/direito de um nó-raiz) nulo é atingido, coloque o novo nó como sendo filho do nó-raiz.

Inserção

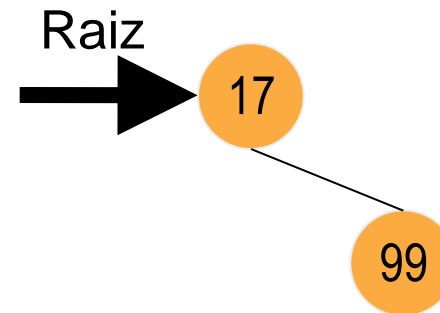
- Para entender o algoritmo considere a inserção do conjunto de números, na seqüência

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

- No início a ABB está vazia!

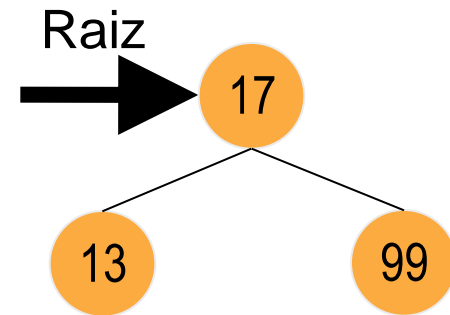
Inserção

- 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 (sub-árvore direita, inicialmente, nula)



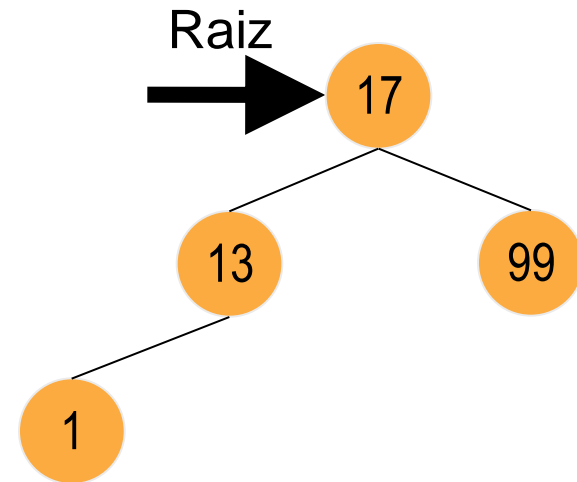
Inserção

- 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 na árvore nessa posição



Inserção

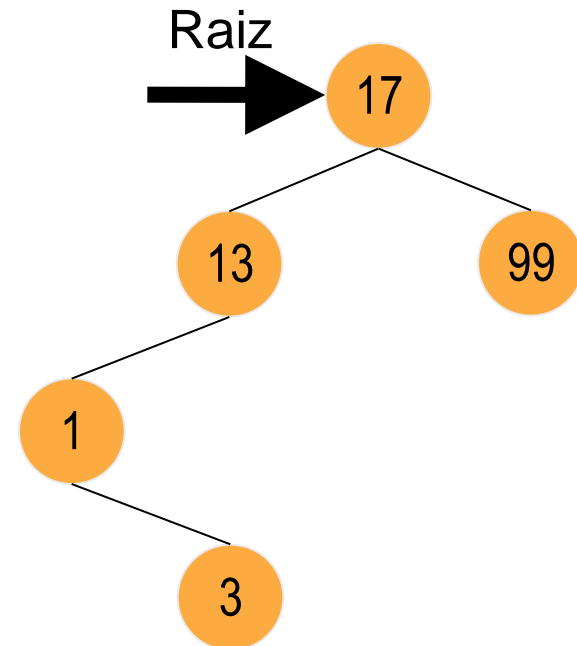
- Repete-se o procedimento para inserir o valor 1
- $1 < 17$, então será inserido na sub-árvore esquerda
- Chegando nela, encontra-se o nó 13, $1 < 13$ então ele será inserido na sub-árvore esquerda de 13



Inserção

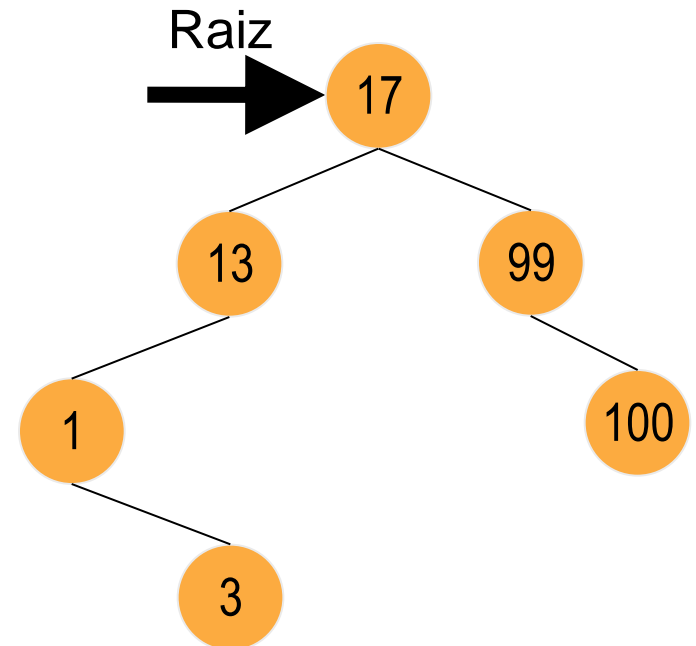
- Repete-se o procedimento para inserir o elemento 3:

- $3 < 17$;
- $3 < 13$
- $3 > 1$



Inserção

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

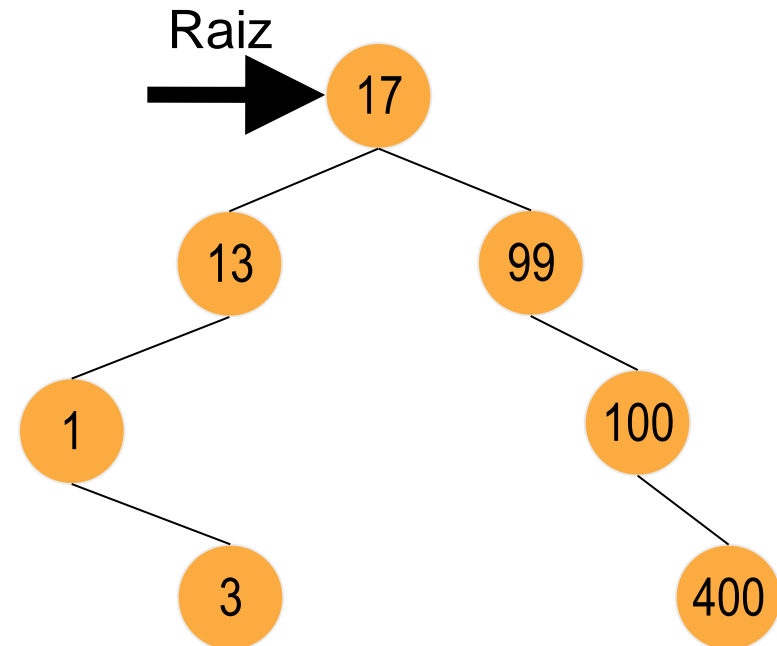


Inserção

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

400:

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



TAD ABB

■ Inserção

- Estratégia geral
 - Inserir elementos como nós folha (sem filhos)
 - Procurar o lugar certo e então inserir

 - Comparando o parâmetro “chave” com a informação no nó “raiz”, 4 casos podem ocorrer
 - A árvore é vazia => insere o elemento, que passará a ser a raiz (se houver espaço); fim do algoritmo
 - Chave < elemento da raiz => insere na subárvore esquerda
 - Chave > elemento da raiz => insere na subárvore direita
 - Elemento da raiz = chave => o elemento já está na árvore; fim do algoritmo
-

TAD ABB

■ Exercício

- Implementação da sub-rotina de inserção de um elemento na árvore

 - O que o insere deve retornar??
 - Sucesso ou insucesso,
 - além do ponteiro para o nó inserido, para o caso de desejarmos complementar as informações do nó chave, como usaremos no nosso **Projeto 3**.

 - `int insere(Arv *p, elem v);`
-

```

int insere(Arv *p, elem v){
if (*p==NULL) {
    *p = (Arv) malloc(sizeof(struct arv));
    if (*p == NULL)
        return 0; // falha na inserção: não há espaço
    else {
        (*p)->info = v;
        (*p)->esq = NULL;
        (*p)->dir = NULL;
        return 1; // inserção com sucesso
    }
}
if (v < (*p)->info)
    return insere(&(*p)->esq,v); // insere na sae
else if (v > (*p)->info)
    return insere(&(*p)->dir,v); // insere na sad
else return 0; // elemento duplicado; não insere
}

```

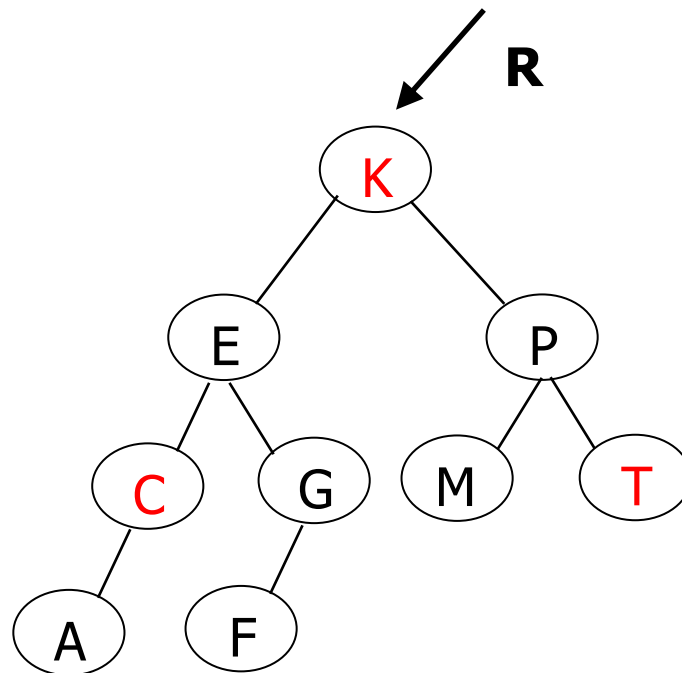
Custo da Operação de Inserção

- 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.**

TAD ABB

■ Remoção

- Prática com o TAD (exemplos)
- Para a árvore abaixo, remova os elementos T, C e K, nesta ordem (cada um destes é um caso diferente)



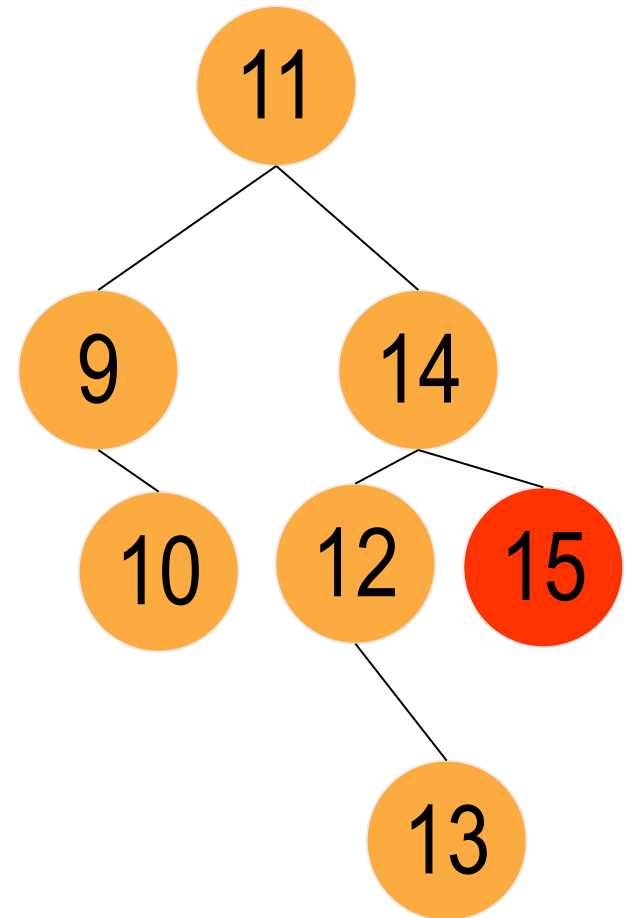
TAD ABB

■ Remoção

- Caso 1 (remover **T**): o nó a ser removido (R) não tem filhos
 - Remove-se o nó
 - R aponta para NULL (importante no caso de ser a raiz)
- Caso 2 (remover **C**): o nó a ser removido tem 1 único filho
 - “Puxa-se” o filho para o lugar do pai
 - Remove-se o nó
- Caso 3 (remover **K**): o nó a ser removido tem 2 filhos
 - Acha-se a maior chave da subárvore esquerda (ou o menor da direita)
 - R recebe o valor dessa chave
 - Remove-se a maior chave da subárvore esquerda (ou a menor da direita)

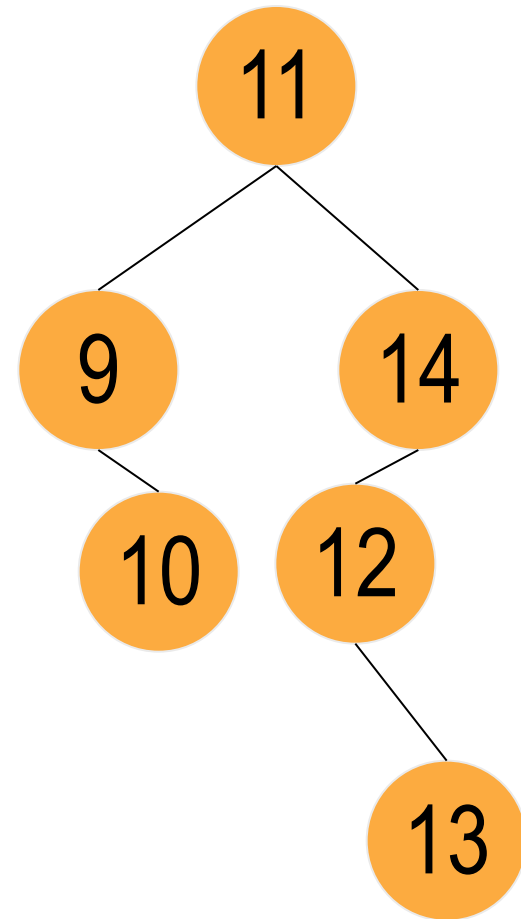
Remoção – Caso 1

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



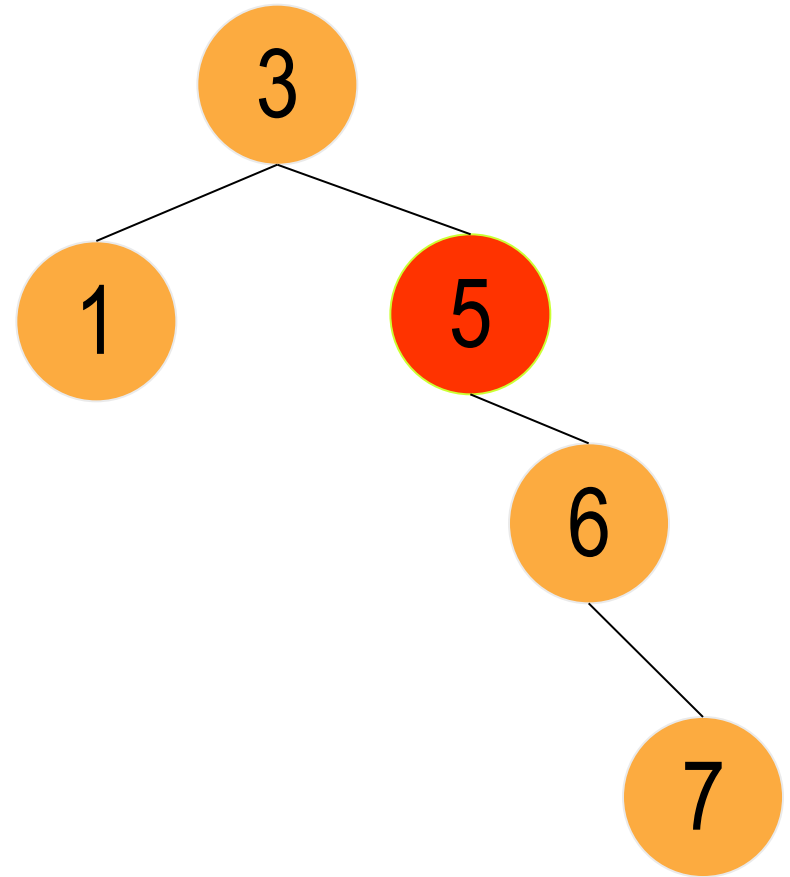
Remoção – Caso 1

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



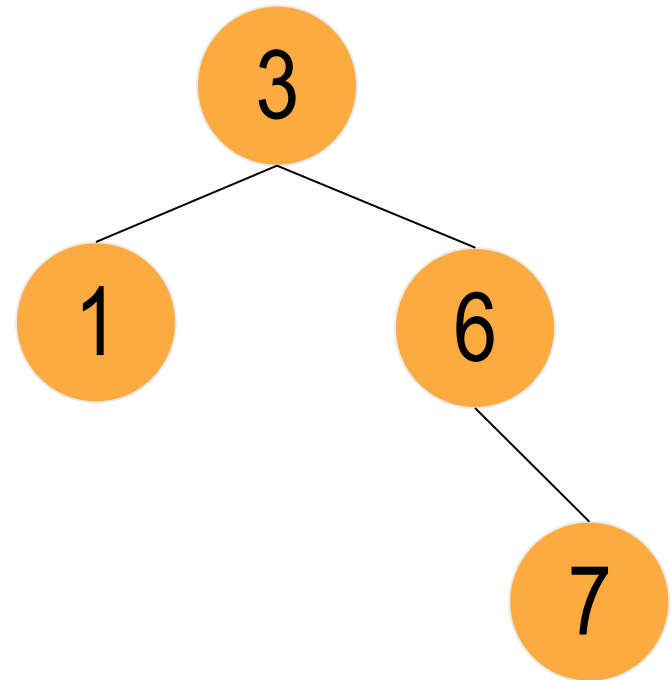
Remoção – Caso 2

- 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



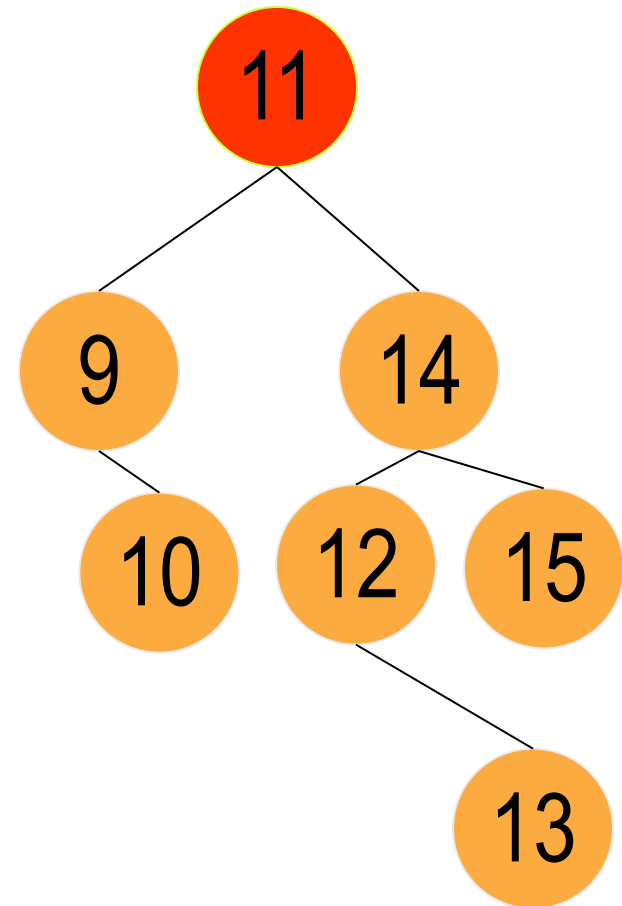
Remoção – Caso 2

- Esse segundo caso é análogo caso existisse um nó com somente uma sub-árvore esquerda



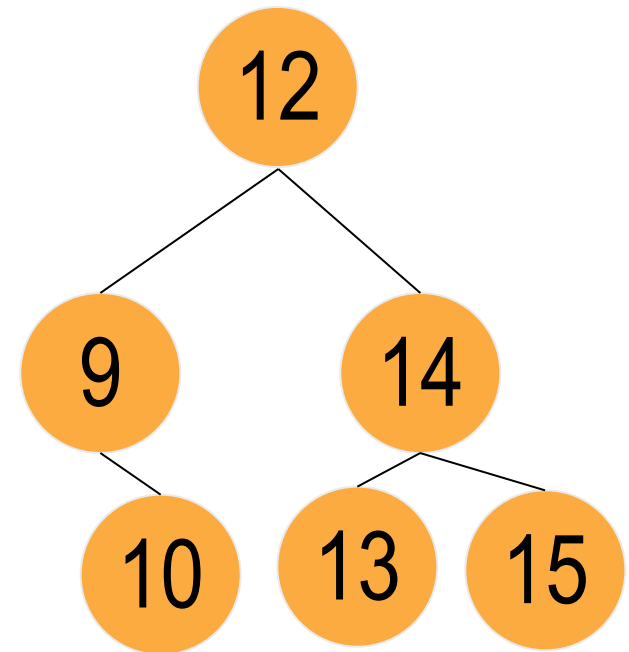
Remoção – Caso 3

- 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, o nó com chave 15 poderia “ocupar” o lugar.



TAD ABB

■ Exercício

- Implementação da sub-rotina de remoção de um elemento da árvore
- O que o remove deve retornar??
 - Sucesso ou insucesso,
 - além da árvore atualizada, pois a remoção pode afetar o endereço da raiz.
- `int remover(Arv *p, elem x)`

