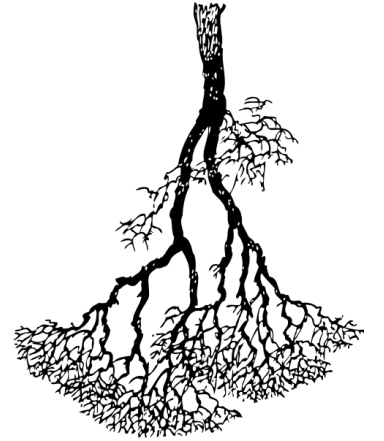


# Árvores Binárias de Busca (BST)



# Árvores Binária

- As árvores binárias são árvores que os nós possuem no máximo 2 filhos
- Devem possuir os atributos que representam o conteúdo do nó + dois ponteiros para os filhos:



- O campo **int value**, apenas ilustra um tipo de dado a ser armazenado

# Árvores Binária

- A inserção deve seguir a forma das listas:
  - Aloca memória
  - Atribui o valor para os campos de armazenamento
  - **NULL**ifica os ponteiros que apontam para esquerda e direita
  - Encontra a melhor posição na árvore para o novo nó

```
newnode=(Node *) malloc(sizeof(Node));  
newnode->value=value;  
newnode->left=NULL;  
newnode->right=NULL;
```



# Árvores de busca

- Uma árvore binária de busca é uma árvore binária que restringe a forma que os nós são inseridos:
  - Binary Search Tree (BST)
- Para todo nó de uma árvore de busca binária:
  - O elemento à esquerda deve ser menor ou igual ao pai
  - O elemento à direita deve ser maior ou igual ao pai
  - A igualdade deve ser **escolhida** ou para **a direita** ou para **a esquerda**

# Árvores de busca (Ex)

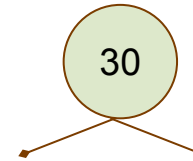
atual

30	51	14	43	82	21	9	3	27	99
0	1	2	3	4	5	6	7	8	9

# Árvores de busca (Ex)

atual

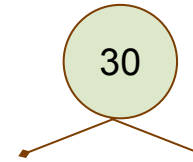
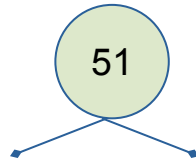
30	51	14	43	82	21	9	3	27	99
0	1	2	3	4	5	6	7	8	9



**Raiz**

# Árvores de busca (Ex)

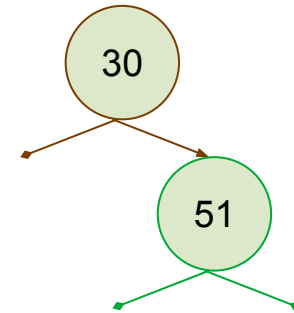
atual									
30	51	14	43	82	21	9	3	27	99
0	1	2	3	4	5	6	7	8	9



**Raiz**

# Árvores de busca (Ex)

atual									
30	51	14	43	82	21	9	3	27	99
0	1	2	3	4	5	6	7	8	9

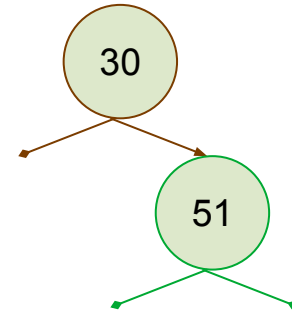
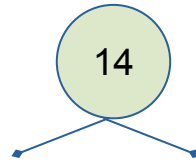


**Raiz**



# Árvores de busca (Ex)

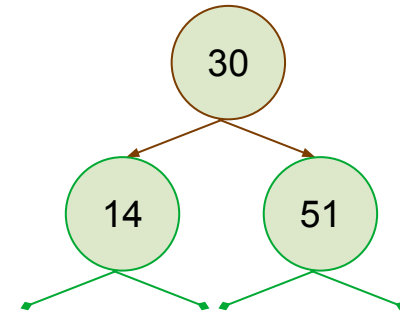
atual									
30	51	14	43	82	21	9	3	27	99
0	1	2	3	4	5	6	7	8	9



**Raiz**

# Árvores de busca (Ex)

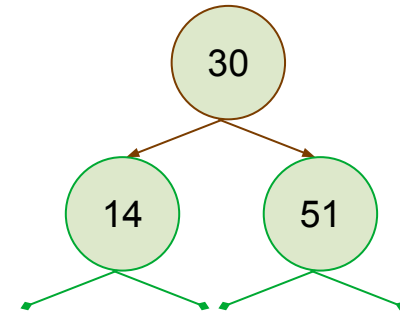
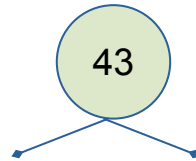
		atual							
30	51	14	43	82	21	9	3	27	99
0	1	2	3	4	5	6	7	8	9



**Raiz**

# Árvores de busca (Ex)

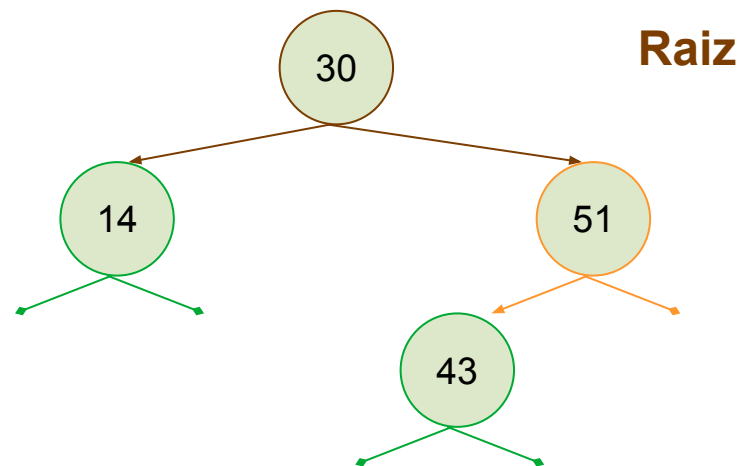
30	51	14	43	82	21	9	3	27	99
0	1	2	3	4	5	6	7	8	9



**Raiz**

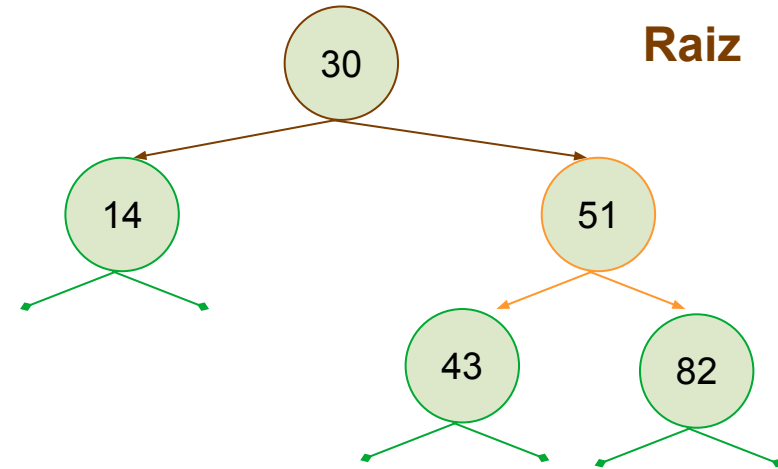
# Árvores de busca (Ex)

			atual						
30	51	14	43	82	21	9	3	27	99
0	1	2	3	4	5	6	7	8	9



# Árvores de busca (Ex)

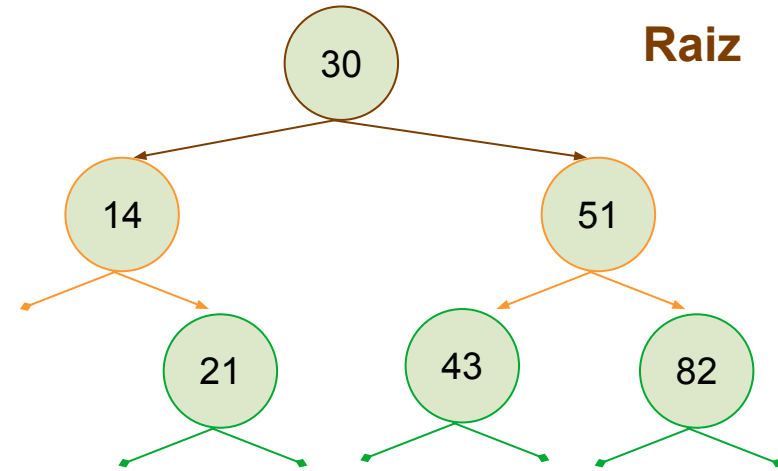
30	51	14	43	82	21	9	3	27	99
0	1	2	3	4	5	6	7	8	9



# Árvores de busca (Ex)

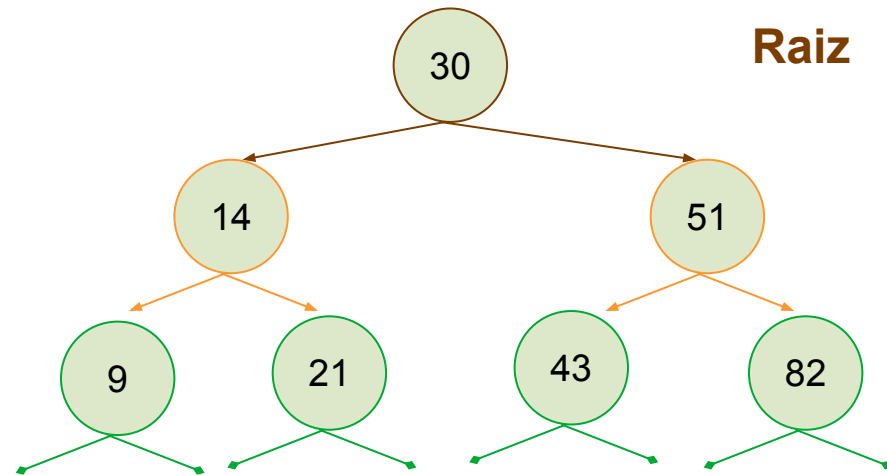
30	51	14	43	82	21	9	3	27	99
0	1	2	3	4	5	6	7	8	9

atual



# Árvores de busca (Ex)

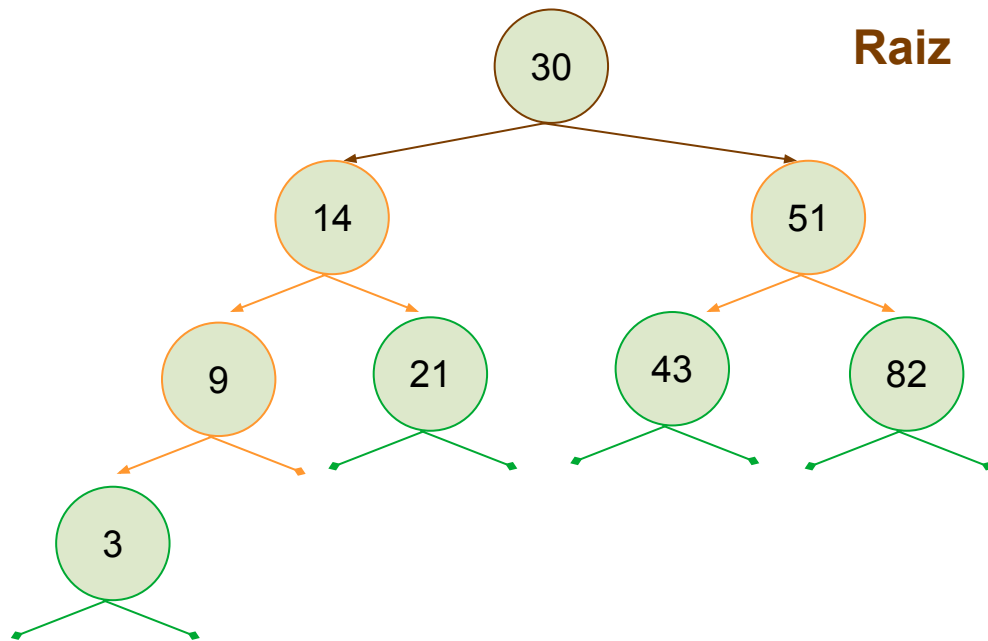
30	51	14	43	82	21	9	3	27	99
0	1	2	3	4	5	6	7	8	9



# Árvores de busca (Ex)

30	51	14	43	82	21	9	3	27	99
0	1	2	3	4	5	6	7	8	9

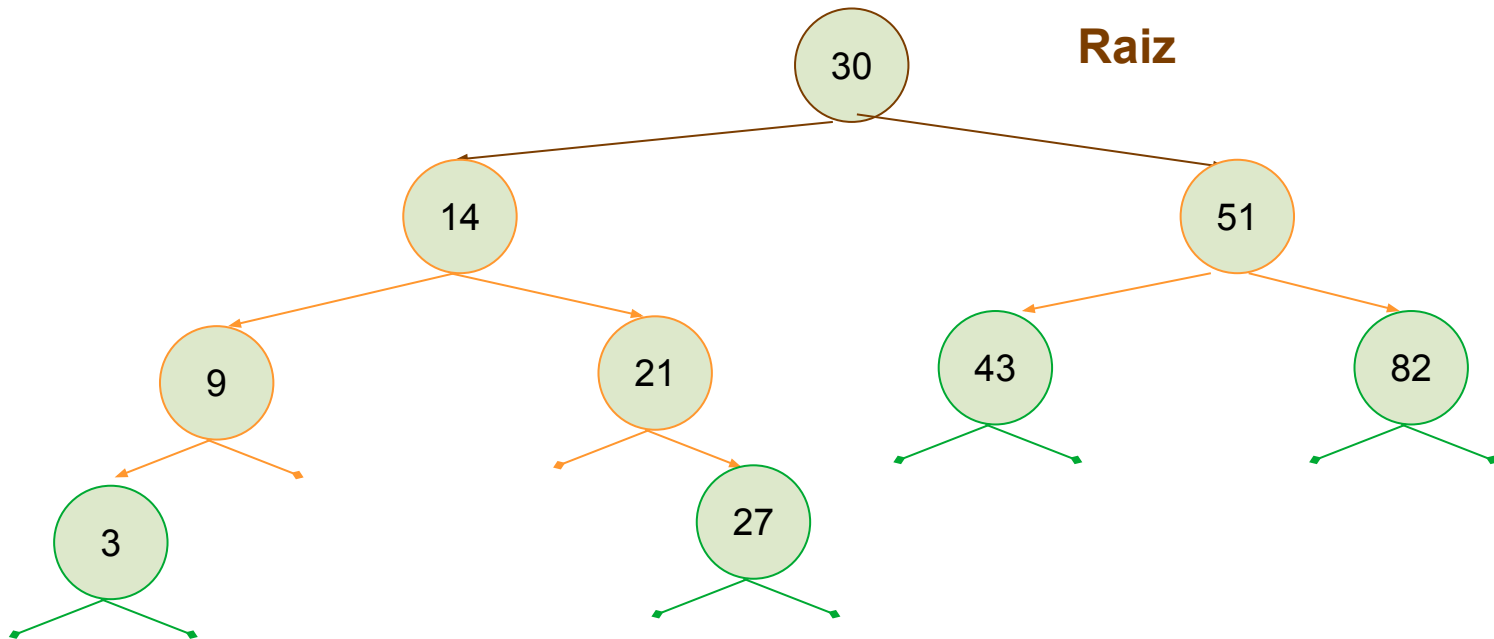
atual





# Árvores de busca (Ex)

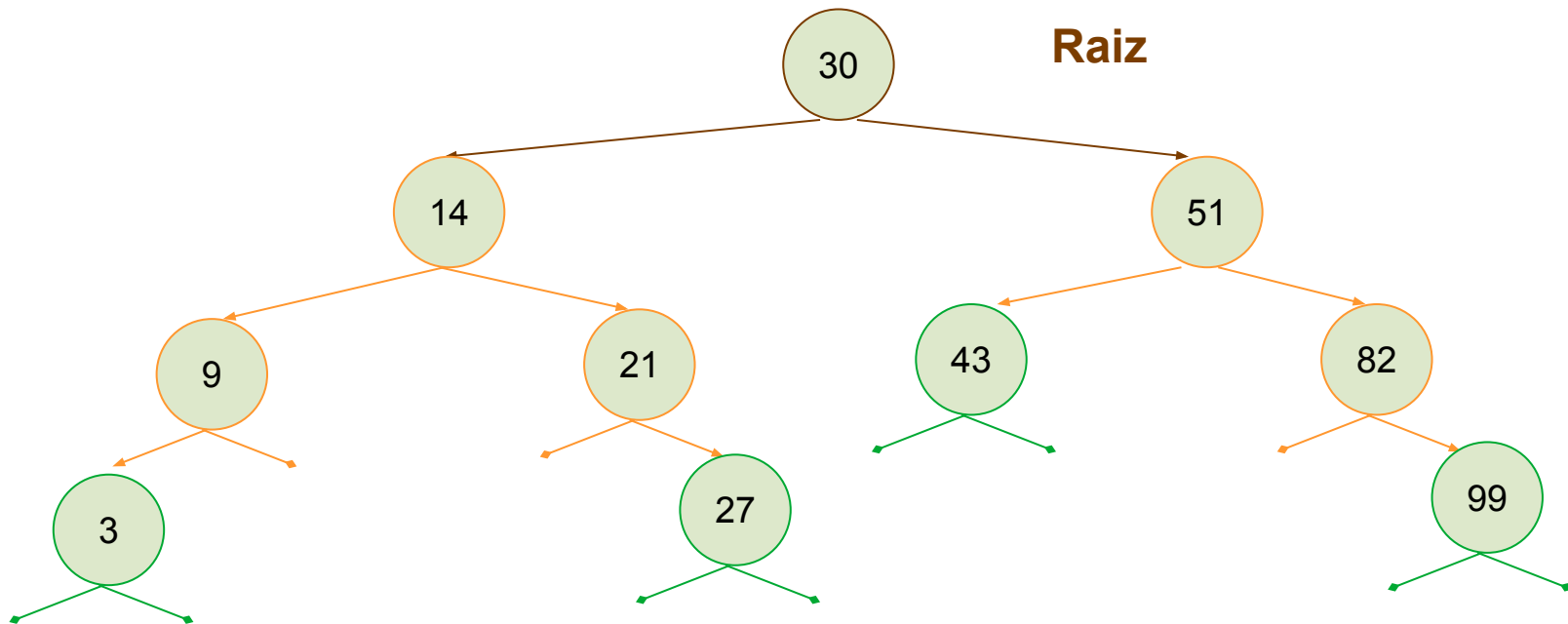
30	51	14	43	82	21	9	3	<sup>atual</sup> 27	99
0	1	2	3	4	5	6	7	8	9



# Árvores de busca (Ex)

30	51	14	43	82	21	9	3	27	99
0	1	2	3	4	5	6	7	8	9

atual



# Árvores de busca

Algoritmo adicionaNo(Node \*raiz, Node \*new)

Início

    if (raiz == NULL)

        return new;

    else

        If (raiz→valor >= new→valor)

            raiz→left = adicionaNo(raiz→left, new);

        else

            raiz→right = adicionaNo(raiz→right, new);

        fim se

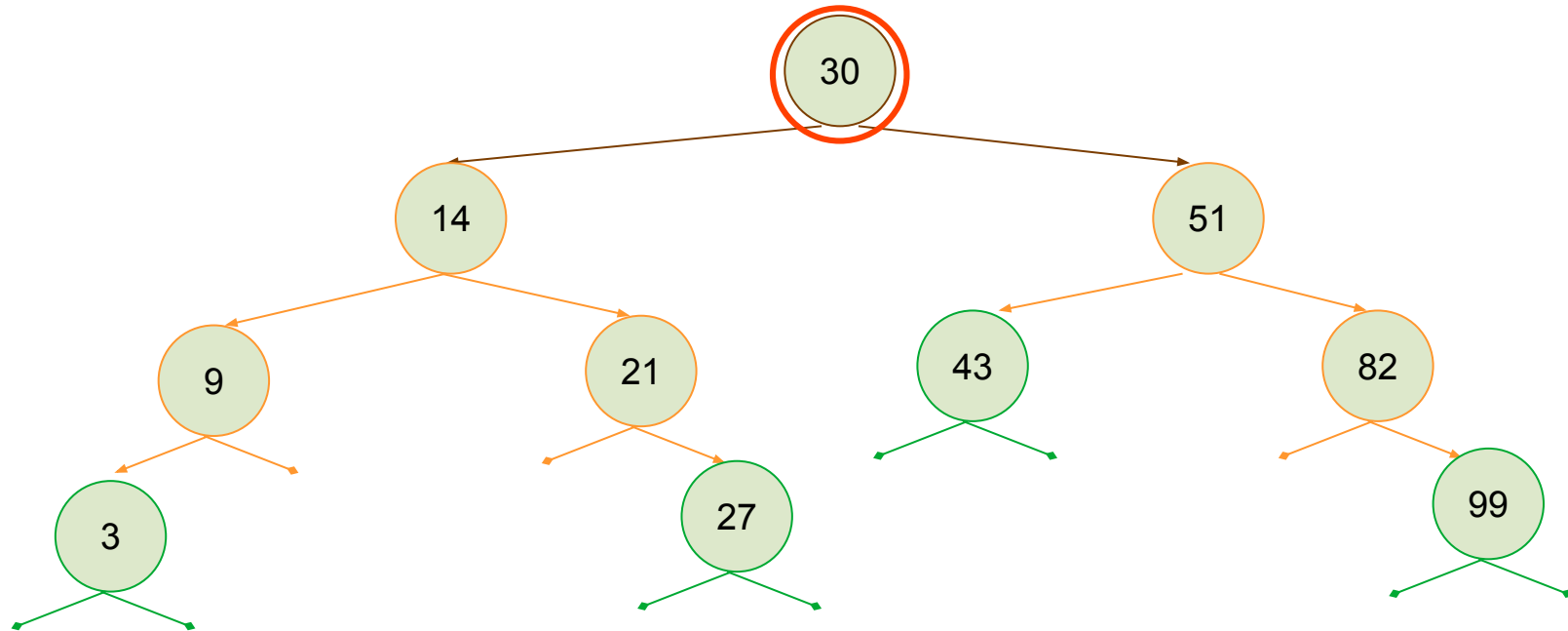
    fim se

    return raiz;

fimAlgoritmo

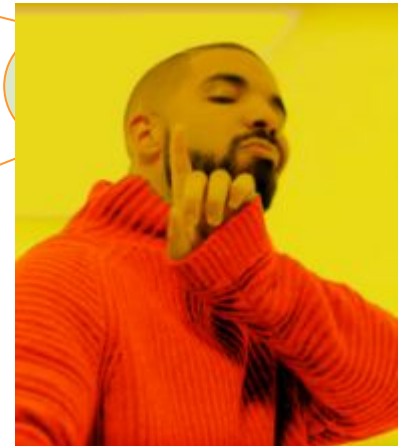
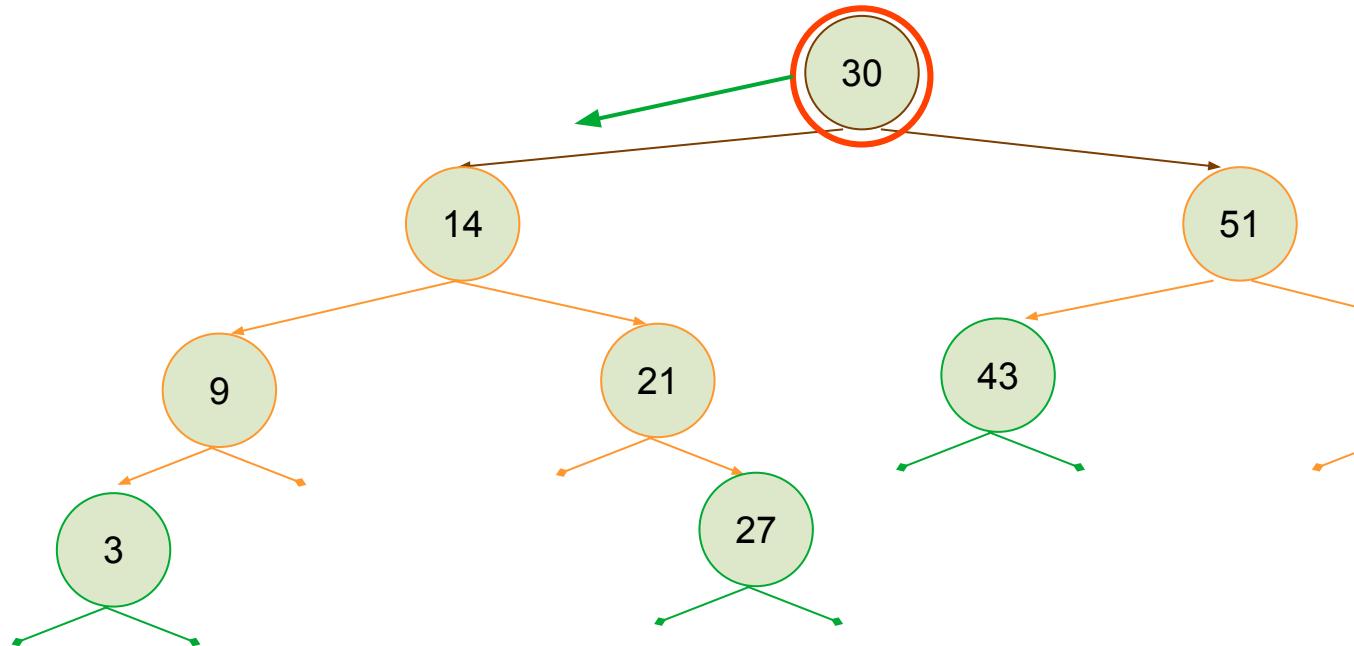
# Árvores de busca (Busca)

K = 21



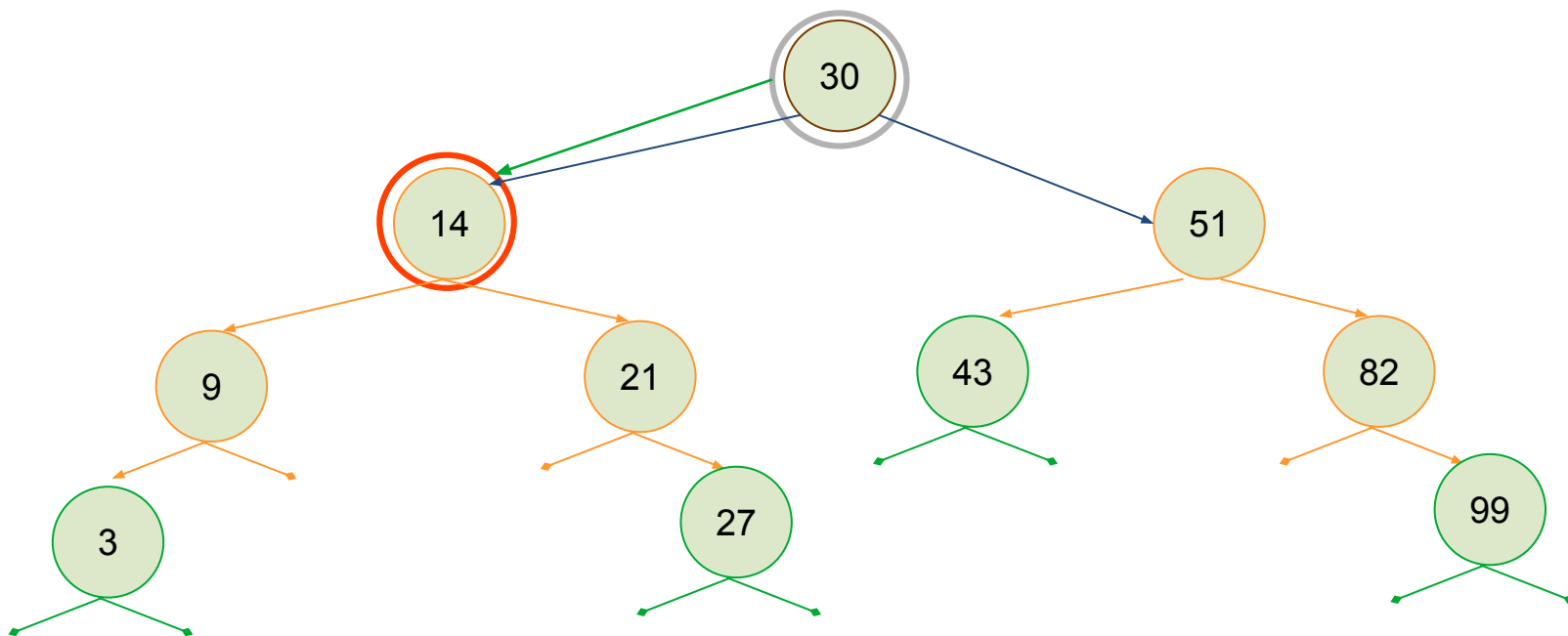
# Árvores de busca (Busca)

K = 21



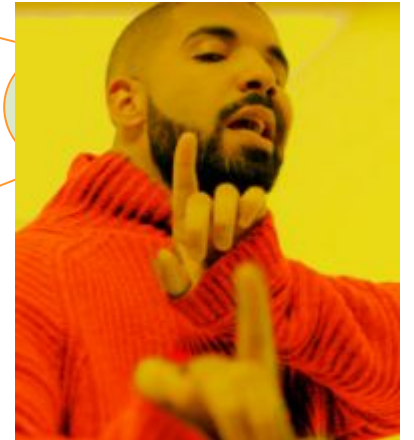
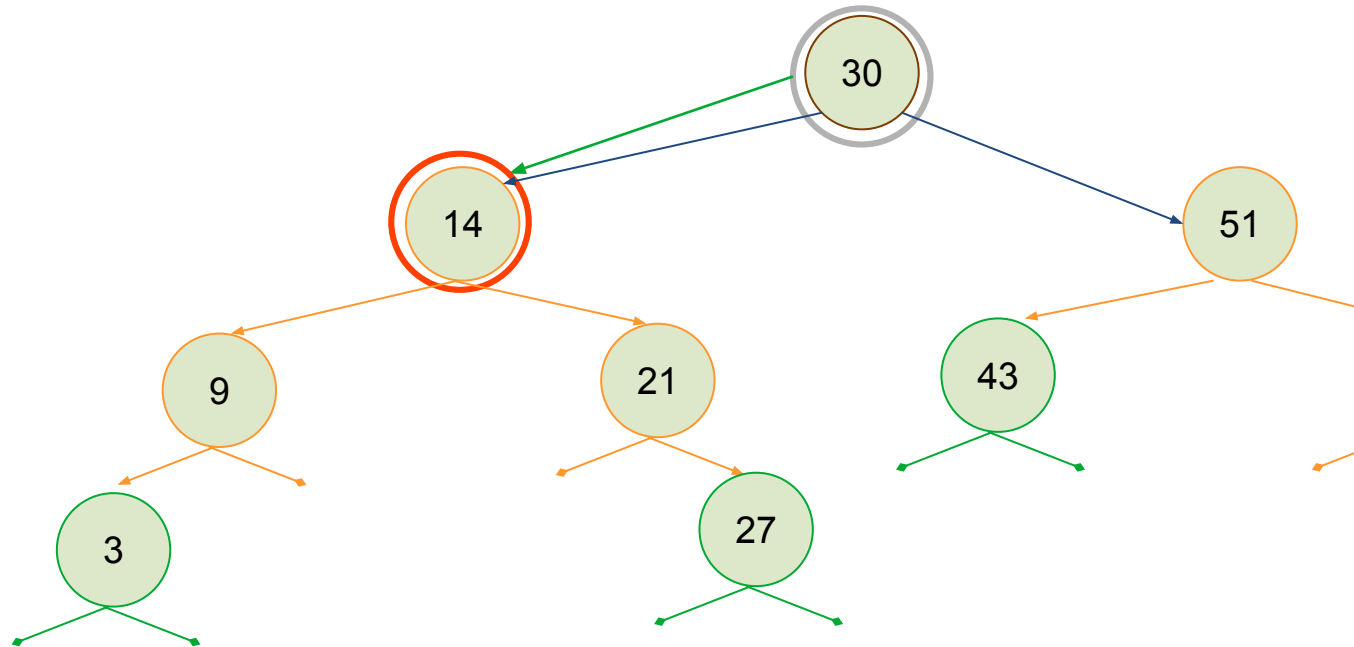
# Árvores de busca (Busca)

K = 21



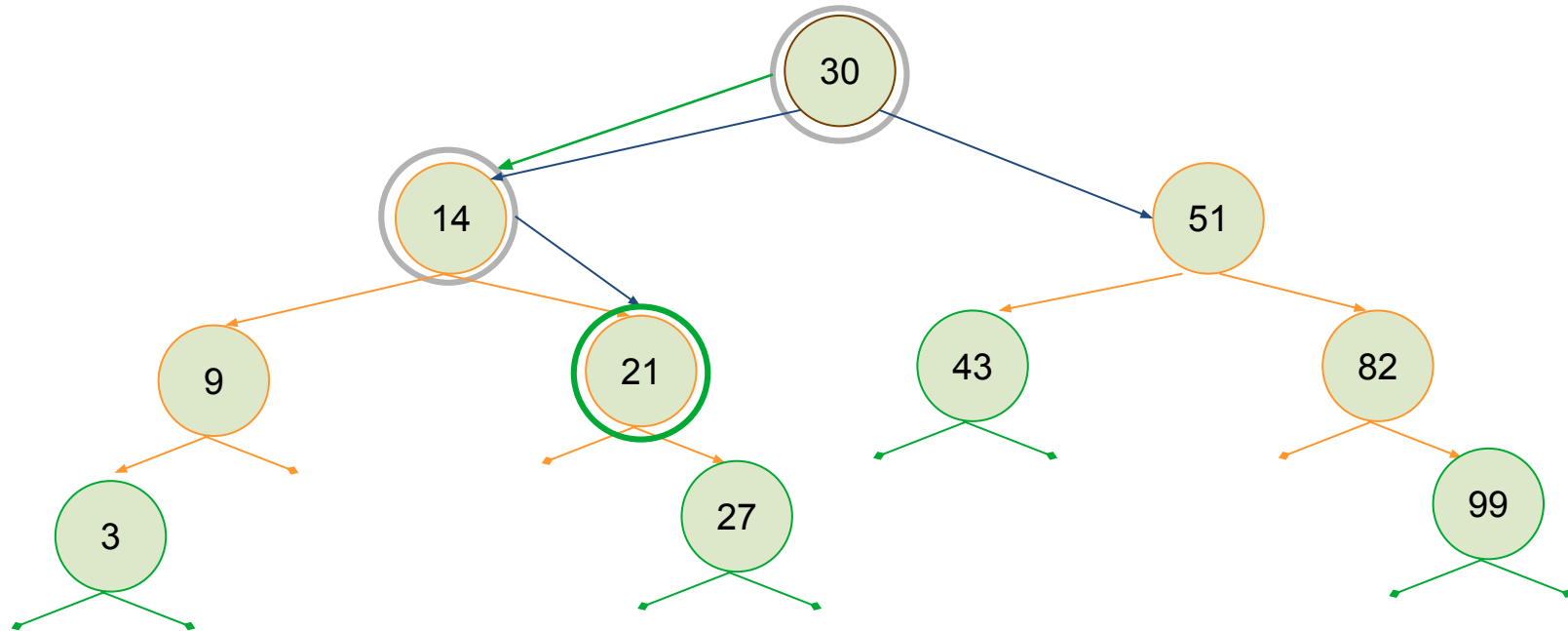
# Árvores de busca (Busca)

K = 21



# Árvores de busca (Busca)

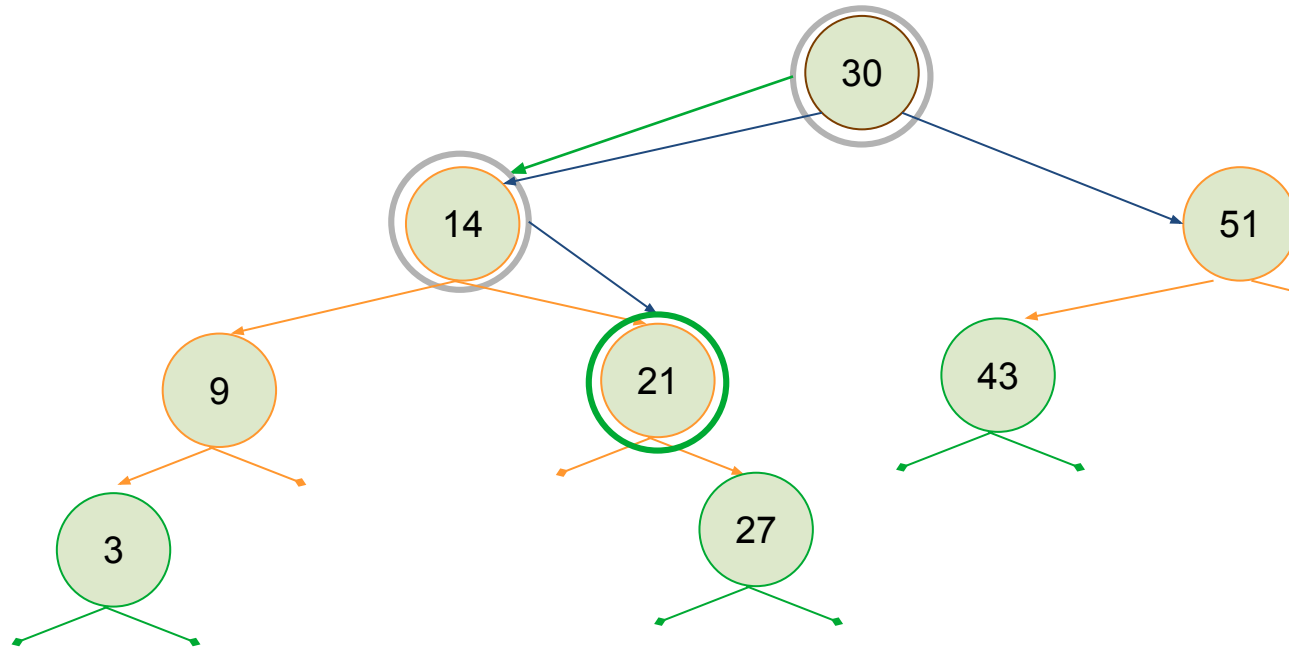
K = 21





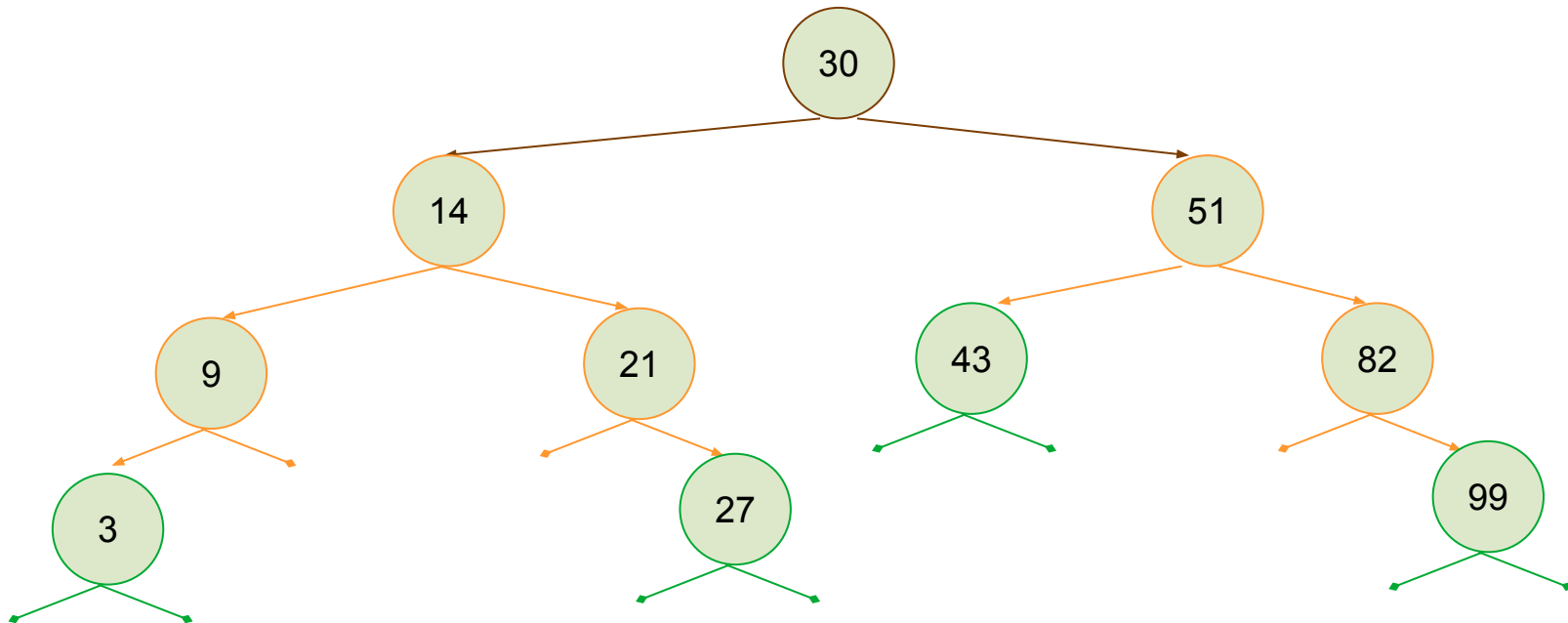
# Árvores de busca (Busca)

K = 21



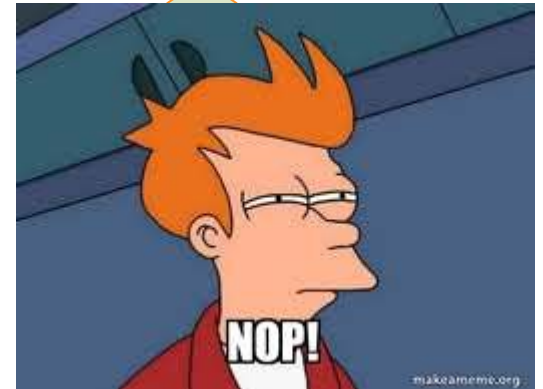
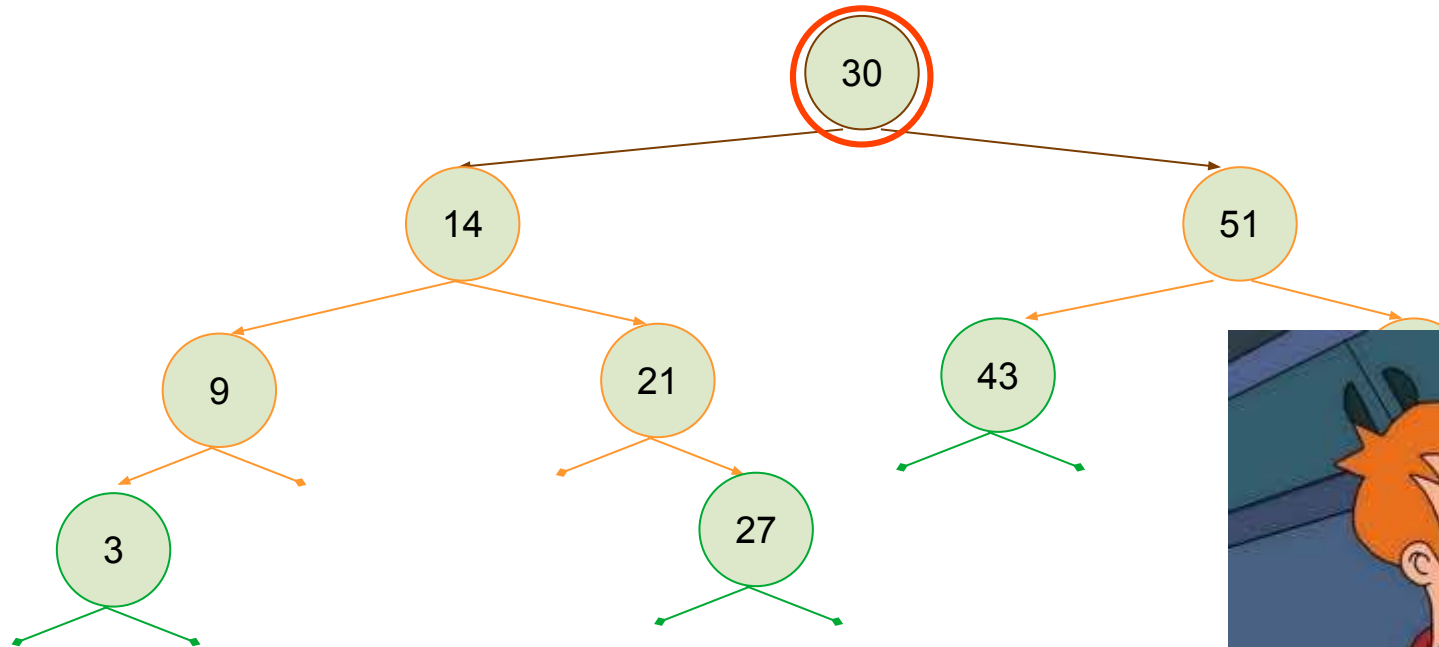
# Árvores de busca (Busca 2)

K = 96



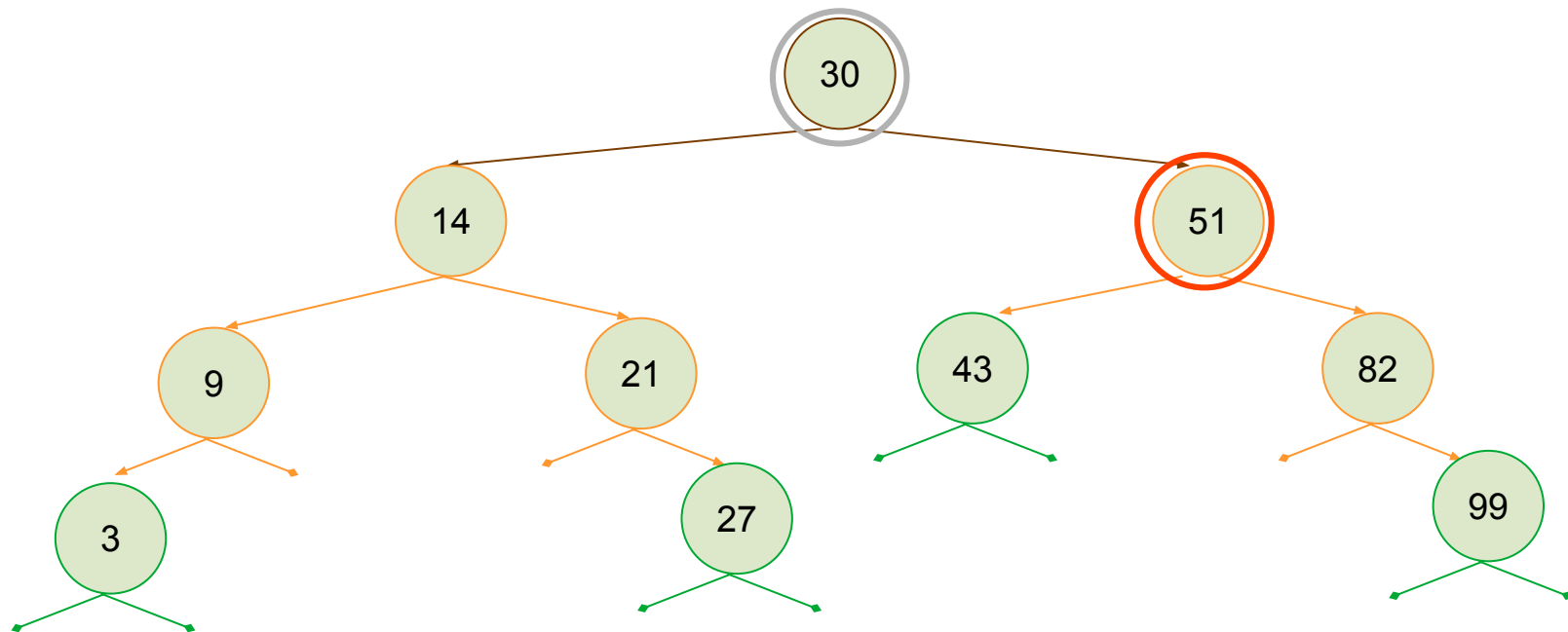
# Árvores de busca (Busca 2)

K = 96



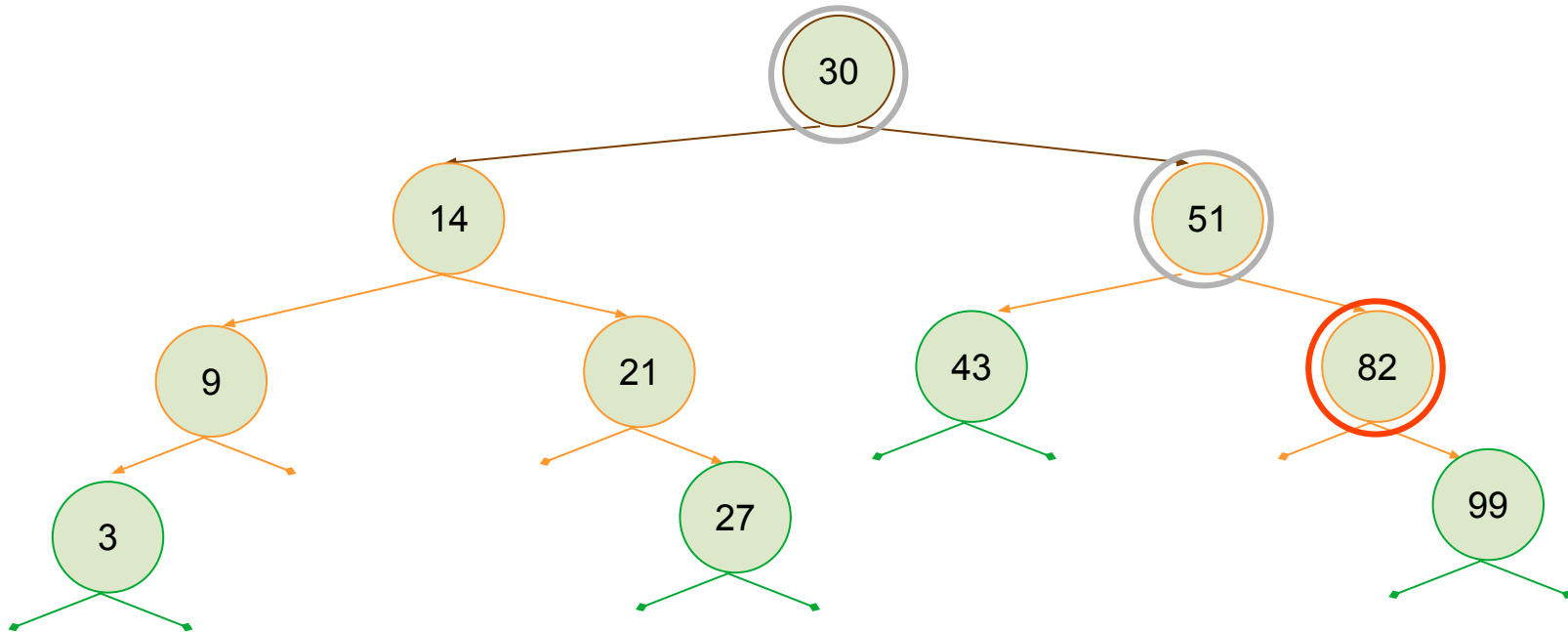
# Árvores de busca (Busca 2)

K = 96



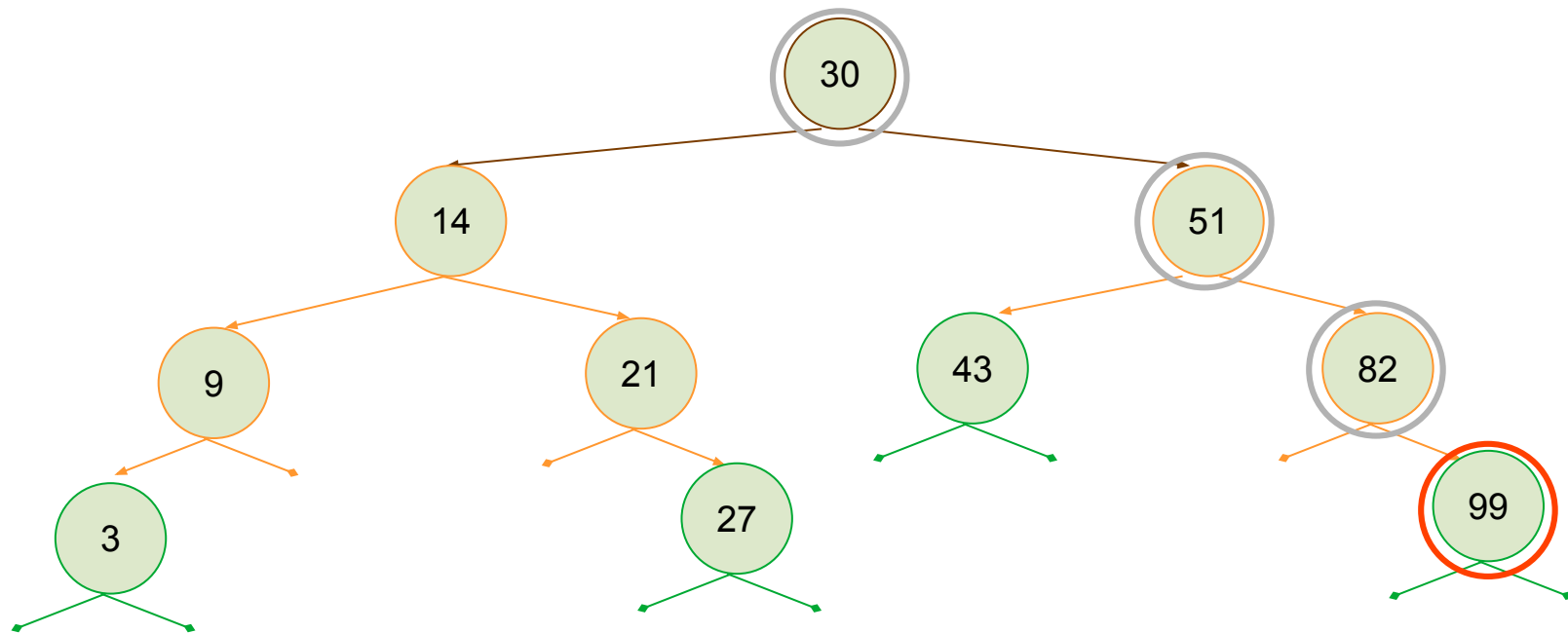
# Árvores de busca (Busca 2)

K = 96



# Árvores de busca (Busca 2)

K = 96



# Árvores de busca (Busca 2)

K = 96



# Árvores de busca

Algoritmo buscaChave(Node \*raiz, key)

Inicio



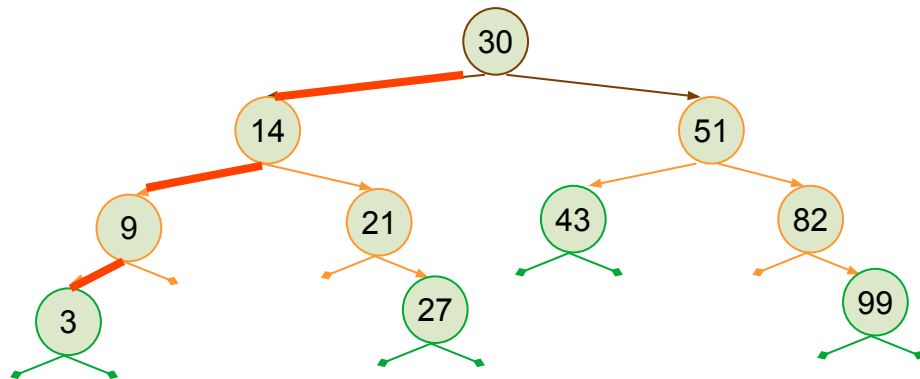
fimAlgoritmo



# Árvores de busca

- A **altura** de um nó  $n$  em uma árvore binária é a distância entre  $n$  e o seu descendente mais afastado (**maior caminho**).
  - Mais precisamente, a altura de  $n$  é o número de passos no mais longo caminho que leva de  $n$  até uma folha.
  - Os caminhos a que essa definição se refere são os obtidos pela iteração das instruções  $n = n \rightarrow \text{left}$  e  $n = n \rightarrow \text{right}$ , em qualquer ordem.
  - A altura da árvore é dada pela distância da raiz até as folhas

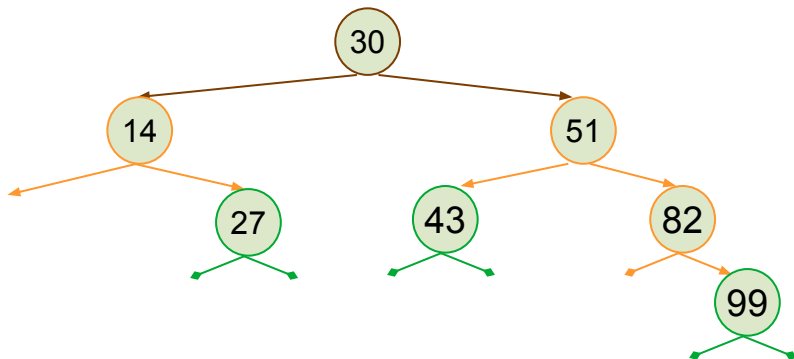
Altura = 4



# Árvores de busca

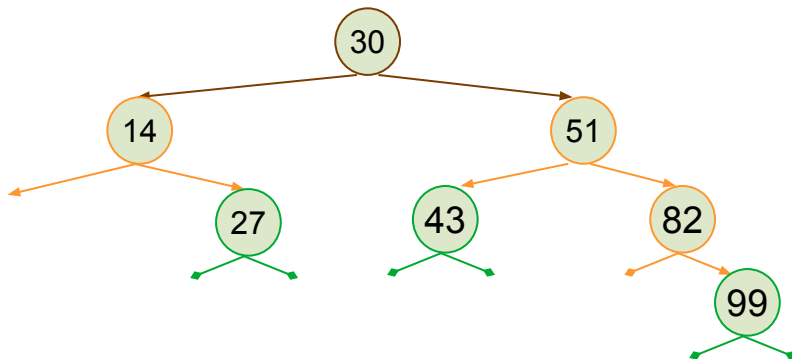
- A **altura** de um nó  $n$  em uma árvore binária é a distância entre  $n$  e o seu descendente mais afastado (**maior caminho**).
  - Mais precisamente, a altura de  $n$  é o número de passos no mais longo caminho que leva de  $n$  até uma folha.
  - Os caminhos a que essa definição se refere são os obtidos pela iteração das instruções  $n = n \rightarrow \text{left}$  e  $n = n \rightarrow \text{right}$ , em qualquer ordem.
  - A altura da árvore é dada pela distância da raiz até as folhas

Altura = ?



# Árvores de busca

- A **profundidade** de um nó é a distância deste até a raiz
  - É um forma semelhante ao cálculo da altura, mas agora não consideramos a folha mais afastada e sim o número de passos para chegar do elemento raiz até o elemento procurado.
- Ex. Qual a profundidade do elemento 82?



# Árvores de busca

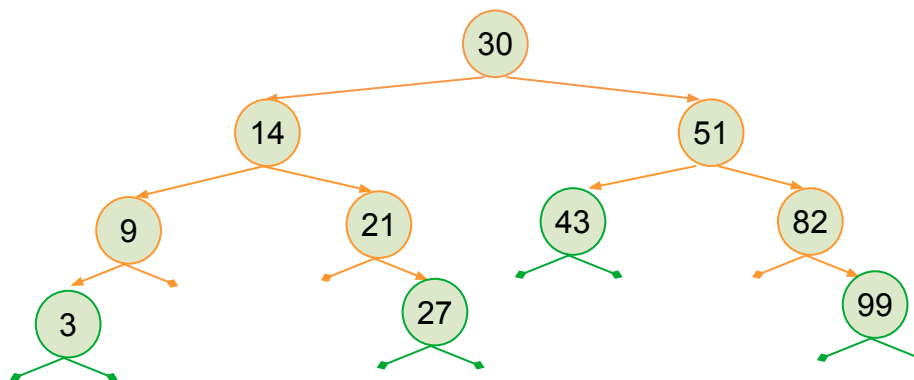
- Existem diversas aplicações onde devemos percorrer uma árvore de maneira sistemática, visitando todos os nós da árvore, um a um.
- Existem três formas de percorrer uma árvore binária:
  - Pré-order
  - In-Order
  - Pós-Order

# Árvores de busca

- Existem diversas aplicações onde devemos percorrer uma árvore de maneira sistemática, visitando todos os nós da árvore, um a um.
- Existem três formas de percorrer uma árvore binária:
  - **Pré-ordem**
  - In-ordem
  - Pós-ordem

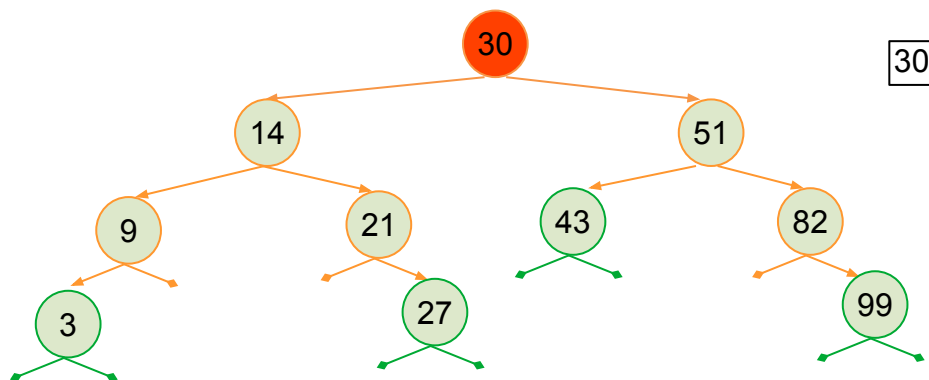
# Árvores de busca

- Pré-ordem
  - Visita a raiz
  - Percorre a subárvore esquerda
  - Percorre a subárvore direita



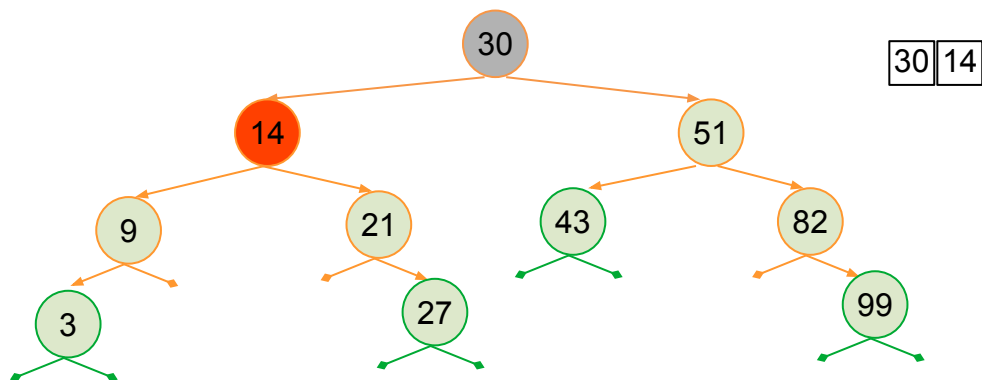
# Árvores de busca

- Pré-ordem
  - Visita a raiz
  - Percorre a subárvore esquerda
  - Percorre a subárvore direita



# Árvores de busca

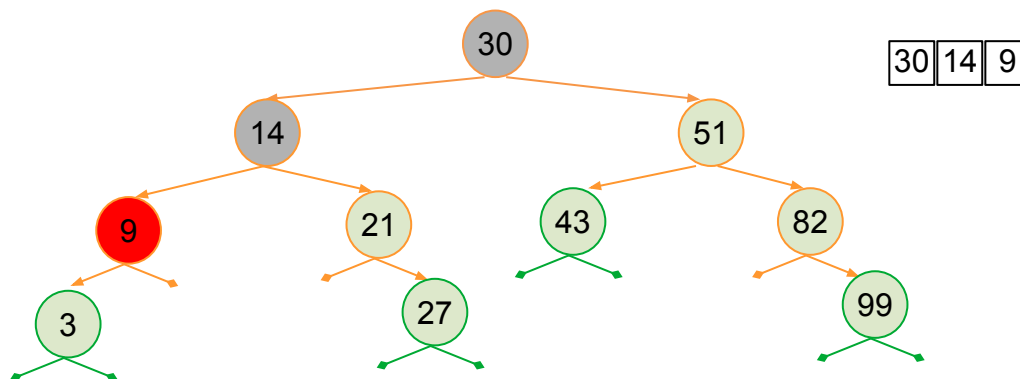
- Pré-ordem
  - Visita a raiz
  - Percorre a subárvore esquerda
  - Percorre a subárvore direita





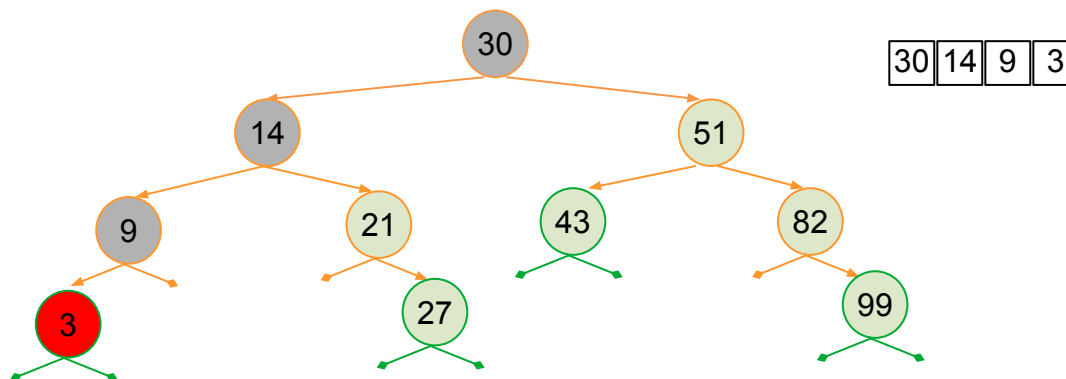
# Árvores de busca

- Pré-ordem
  - Visita a raiz
  - Percorre a subárvore esquerda
  - Percorre a subárvore direita



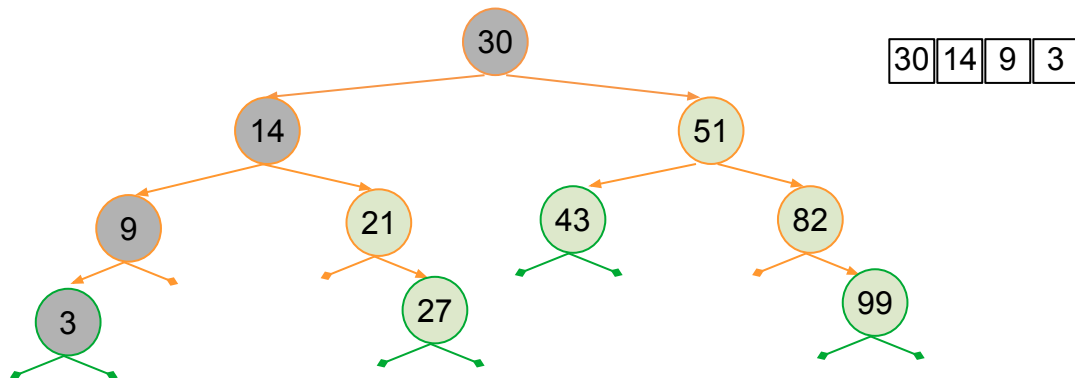
# Árvores de busca

- Pré-ordem
  - Visita a raiz
  - Percorre a subárvore esquerda
  - Percorre a subárvore direita



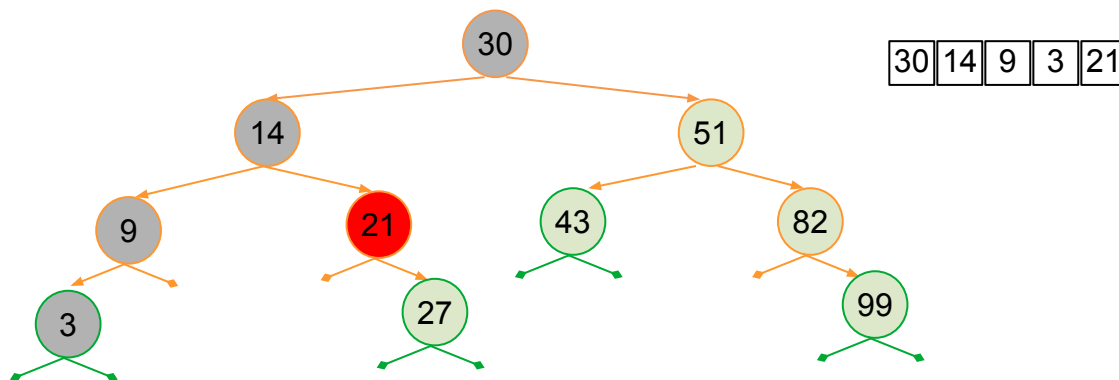
# Árvores de busca

- Pré-ordem
  - Visita a raiz
  - Percorre a subárvore esquerda
  - Percorre a subárvore direita



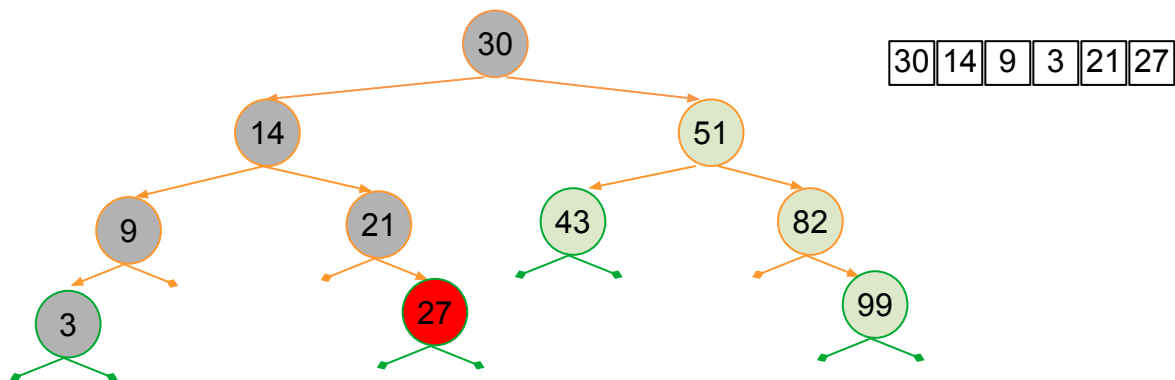
# Árvores de busca

- Pré-ordem
  - Visita a raiz
  - Percorre a subárvore esquerda
  - Percorre a subárvore direita



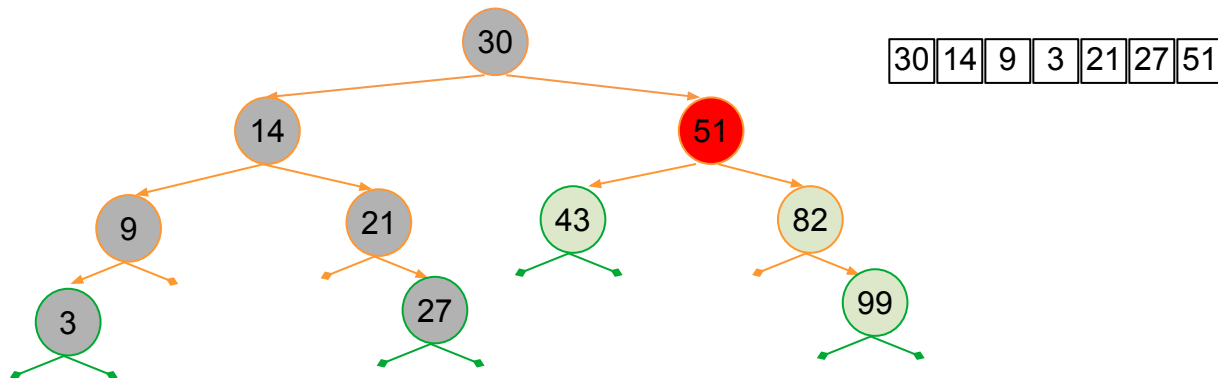
# Árvores de busca

- Pré-ordem
  - Visita a raiz
  - Percorre a subárvore esquerda
  - Percorre a subárvore direita



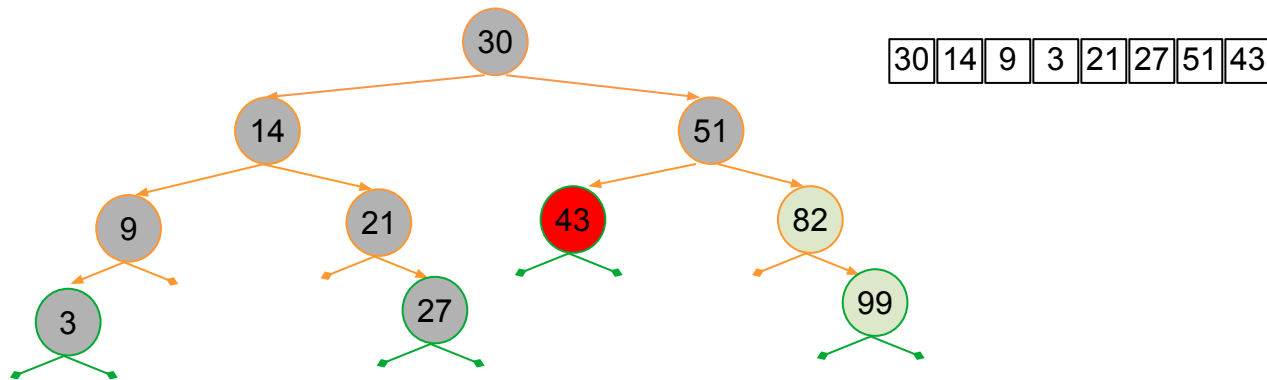
# Árvores de busca

- Pré-ordem
  - Visita a raiz
  - Percorre a subárvore esquerda
  - Percorre a subárvore direita



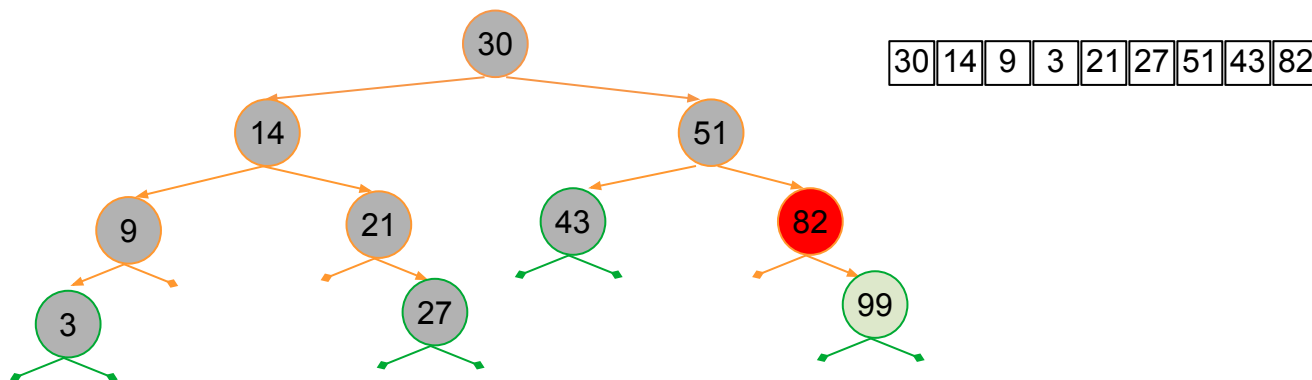
# Árvores de busca

- Pré-ordem
  - Visita a raiz
  - Percorre a subárvore esquerda
  - Percorre a subárvore direita



# Árvores de busca

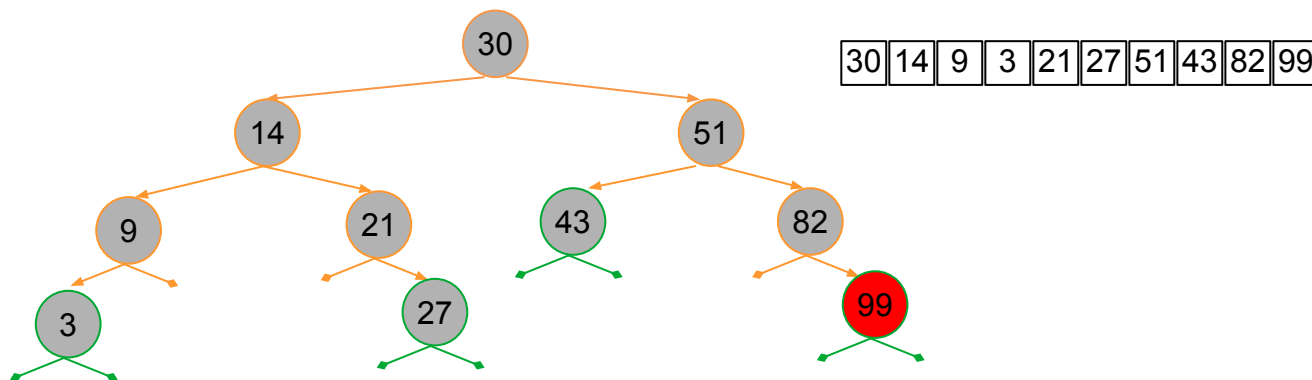
- Pré-ordem
  - Visita a raiz
  - Percorre a subárvore esquerda
  - Percorre a subárvore direita





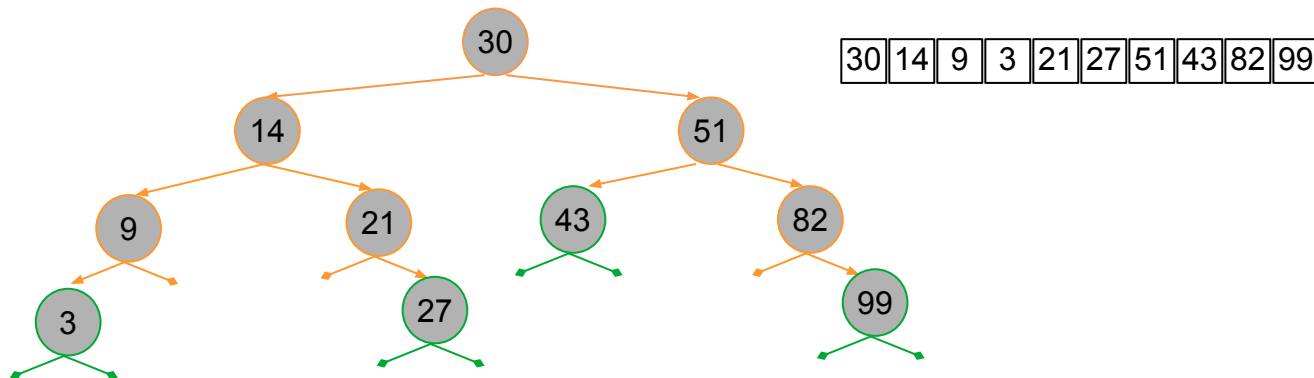
# Árvores de busca

- Pré-ordem
  - Visita a raiz
  - Percorre a subárvore esquerda
  - Percorre a subárvore direita



# Árvores de busca

- Pré-ordem
  - Visita a raiz
  - Percorre a subárvore esquerda
  - Percorre a subárvore direita



# Árvores de busca

- Pré-ordem
  - Visita a raiz
  - Percorre a subárvore esquerda
  - Percorre a subárvore direita

```
void showPreOrder (Tree *root)
{
    if (root==NULL) return;
    printf("%d ",root->value);
    showPreOrder(root->left);
    showPreOrder(root->right);
}
```

# Árvores de busca

- Pré-ordem
  - Visita a raiz
  - Percorre a subárvore esquerda
  - Percorre a subárvore direita

```
void showPreOrder (Tree *root)
{
    if (root==NULL) return;
    printf("%d ",root->value);
    showPreOrder(root->left);
    showPreOrder(root->right);
}
```

# Árvores de busca

- Pré-ordem
  - Visita a raiz
  - Percorre a subárvore esquerda
  - Percorre a subárvore direita

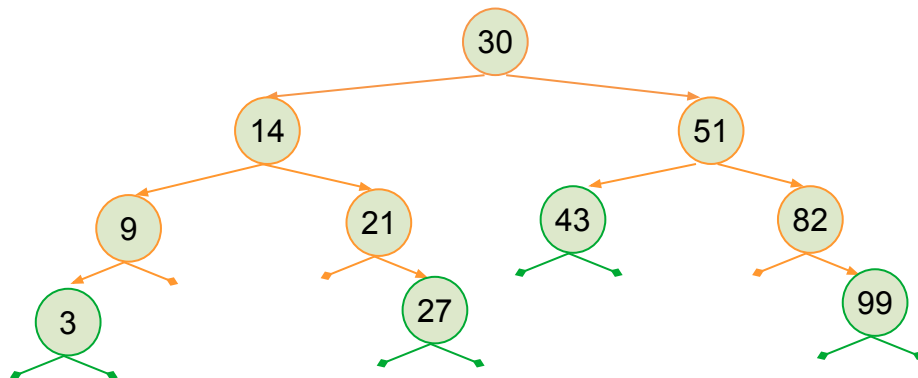
```
void showPreOrder (Tree *root)
{
    if (root==NULL) return;
    printf("%d ",root->value);
    showPreOrder(root->left);
    showPreOrder(root->right);
}
```

# Árvores de busca

- Existem diversas aplicações onde devemos percorrer uma árvore de maneira sistemática, visitando todos os nós da árvore, um a um.
- Existem três formas de percorrer uma árvore binária:
  - Pré-ordem
  - **In-ordem**
  - Pós-ordem

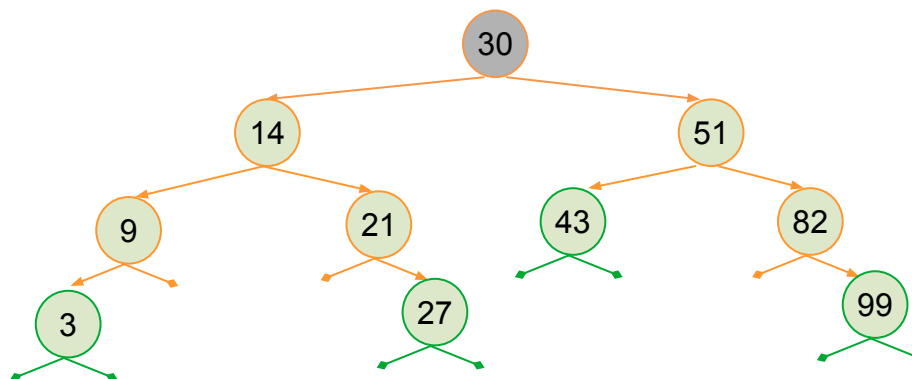
# Árvores de busca

- In-ordem
  - Percorre a subárvore esquerda
  - Visita a raiz
  - Percorre a subárvore direita



# Árvores de busca

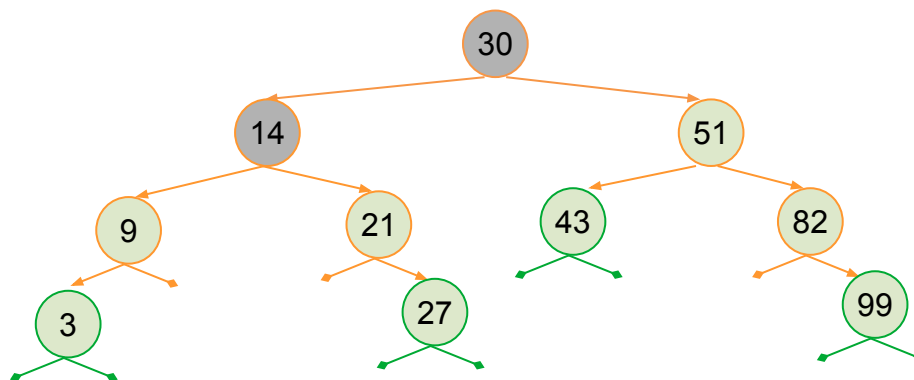
- In-ordem
  - Percorre a subárvore esquerda
  - Visita a raiz
  - Percorre a subárvore direita





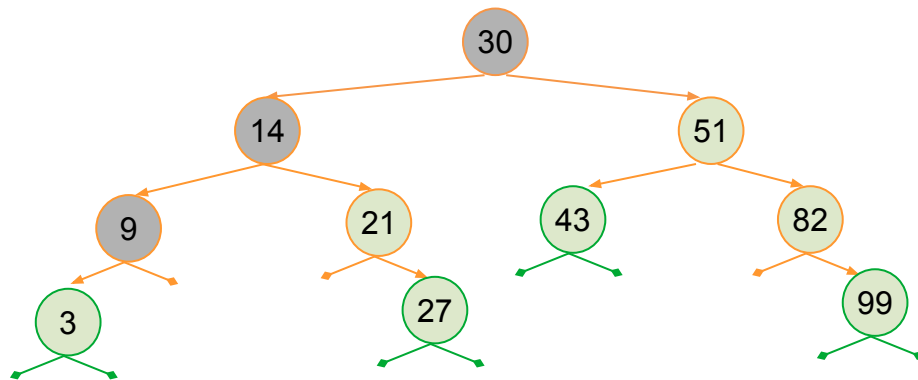
# Árvores de busca

- In-ordem
  - Percorre a subárvore esquerda
  - Visita a raiz
  - Percorre a subárvore direita



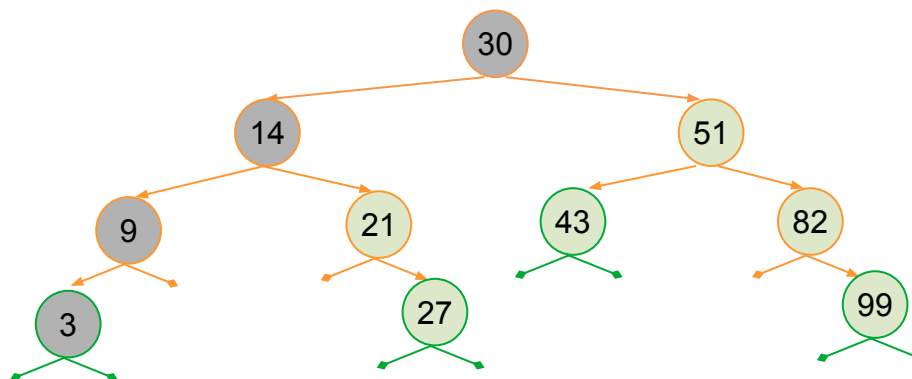
# Árvores de busca

- In-ordem
  - Percorre a subárvore esquerda
  - Visita a raiz
  - Percorre a subárvore direita



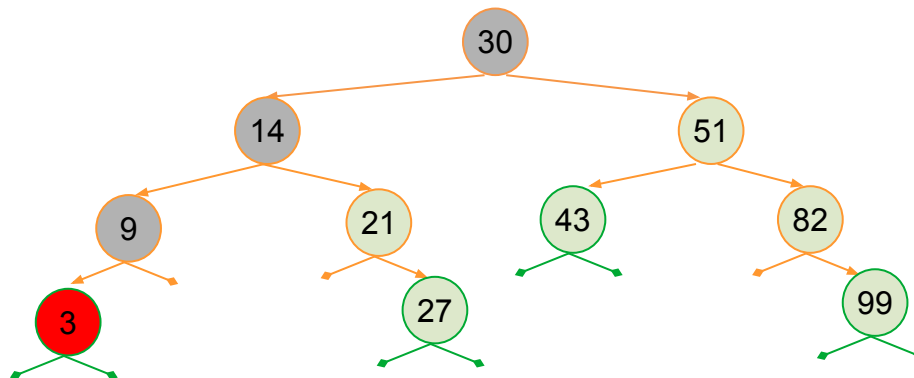
# Árvores de busca

- In-ordem
  - Percorre a subárvore esquerda
  - Visita a raiz
  - Percorre a subárvore direita



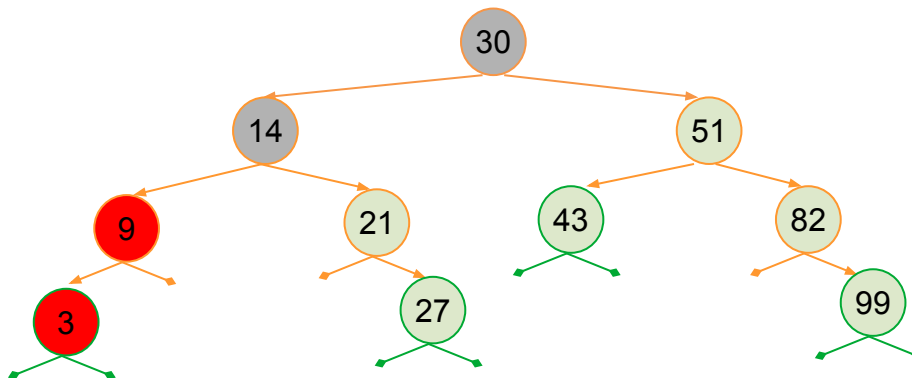
# Árvores de busca

- In-ordem
  - Percorre a subárvore esquerda
  - Visita a raiz
  - Percorre a subárvore direita



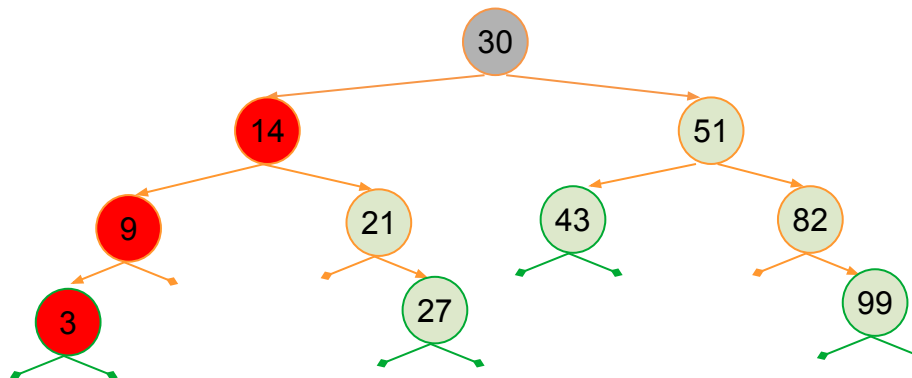
# Árvores de busca

- In-ordem
  - Percorre a subárvore esquerda
  - Visita a raiz
  - Percorre a subárvore direita



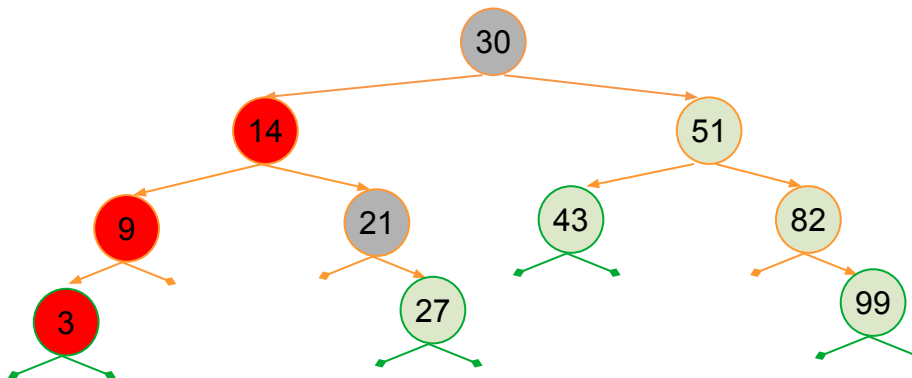
# Árvores de busca

- In-ordem
  - Percorre a subárvore esquerda
  - Visita a raiz
  - Percorre a subárvore direita



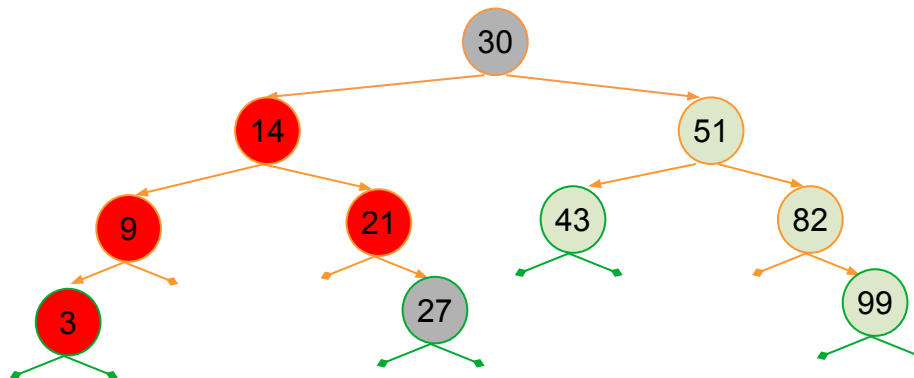
# Árvores de busca

- In-ordem
  - Percorre a subárvore esquerda
  - Visita a raiz
  - Percorre a subárvore direita



# Árvores de busca

- In-ordem
  - Percorre a subárvore esquerda
  - Visita a raiz
  - Percorre a subárvore direita

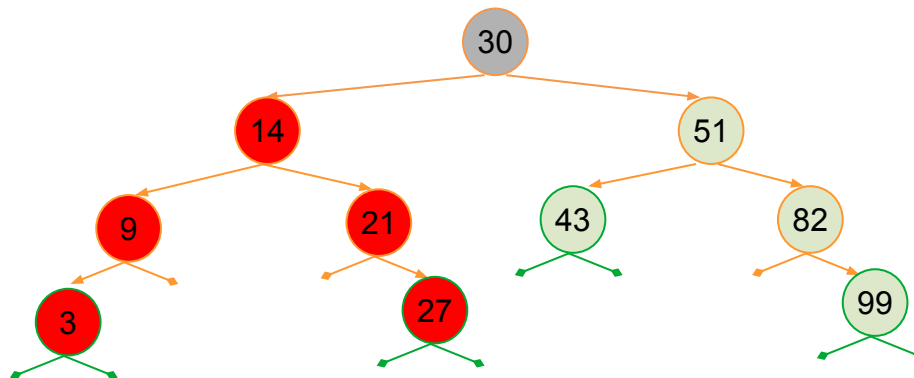




# Árvores de busca

- In-ordem
  - Percorre a subárvore esquerda
  - Visita a raiz
  - Percorre a subárvore direita

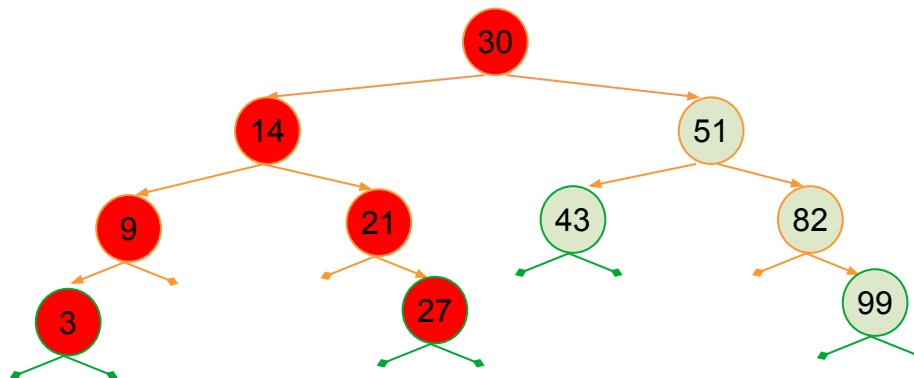
3	9	14	21	27				
---	---	----	----	----	--	--	--	--



# Árvores de busca

- In-ordem
  - Percorre a subárvore esquerda
  - Visita a raiz
  - Percorre a subárvore direita

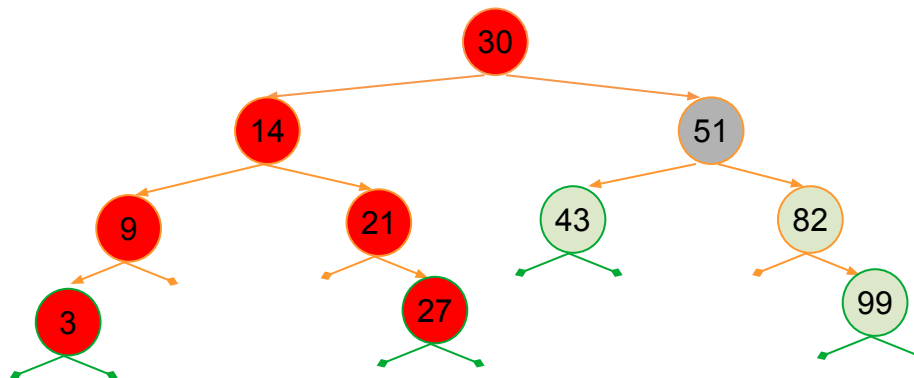
3	9	14	21	27	30			
---	---	----	----	----	----	--	--	--



# Árvores de busca

- In-ordem
  - Percorre a subárvore esquerda
  - Visita a raiz
  - Percorre a subárvore direita

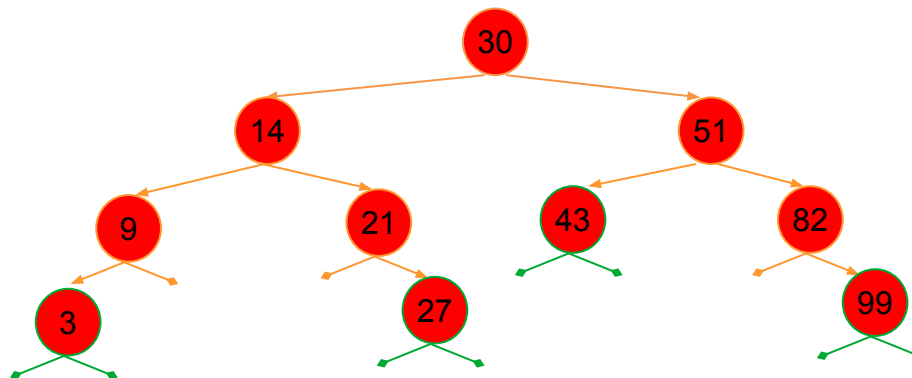
3	9	14	21	27	30			
---	---	----	----	----	----	--	--	--



# Árvores de busca

- In-ordem
  - Percorre a subárvore esquerda
  - Visita a raiz
  - Percorre a subárvore direita

3	9	14	21	27	30	43	51	82	99
---	---	----	----	----	----	----	----	----	----



# Árvores de busca

- In-ordem
  - Percorre a subárvore esquerda
  - Visita a raiz
  - Percorre a subárvore direita

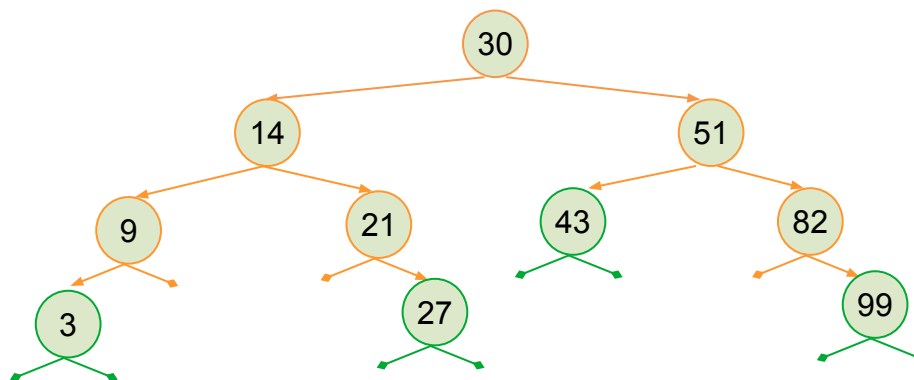
```
void showInOrder (Tree *root)
{
    /// Como seria???
}
```

# Árvores de busca

- Existem diversas aplicações onde devemos percorrer uma árvore de maneira sistemática, visitando todos os nós da árvore, um a um.
- Existem três formas de percorrer uma árvore binária:
  - Pré-ordem
  - In-ordem
  - **Pós-ordem**

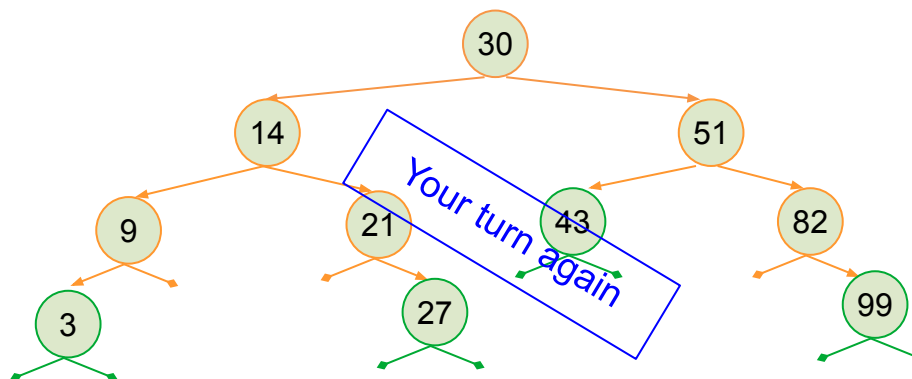
# Árvores de busca

- Pós-ordem
  - Percorre a subárvore esquerda
  - Percorre a subárvore direita
  - Visita a raiz



# Árvores de busca

- Pós-ordem
  - Percorre a subárvore esquerda
  - Percorre a subárvore direita
  - Visita a raiz



Saída?



# Árvores de busca

- Pós-ordem
  - Percorre a subárvore esquerda
  - Percorre a subárvore direita
  - Visita a raiz

```
void showPosOrder (Tree *root)
{
    /// Como seria???

}
```

# Árvores de busca

- Árvores representam um ótima otimização de performance para o acesso a informações
  - A complexidade é da **ordem logarítmica** já que temos uma árvore
    - **$\log_2$**  pois a árvore é binária

# Árvores de busca

- Árvores representam um ótima otimização de performance para o acesso a informações
  - A complexidade é da **ordem logarítmica** já que temos uma árvore
    - **$\log_2$**  pois a árvore é binária

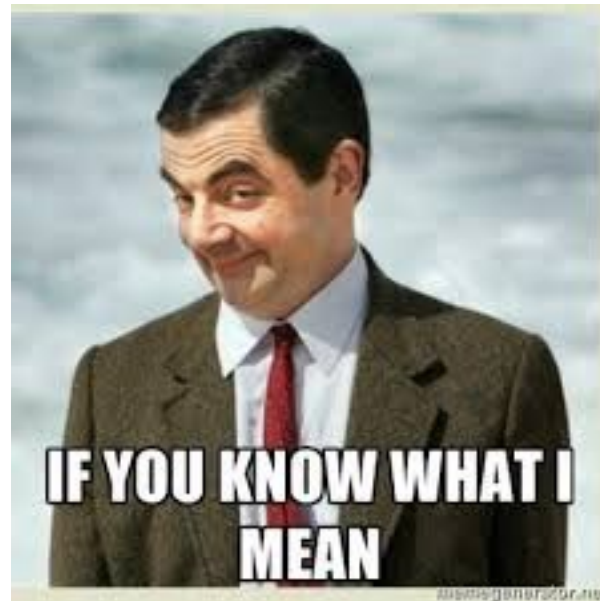


# Árvores de busca

- Árvores representam um ótima otimização de performance para o acesso a informações
  - A complexidade é difícil e depende da organização da árvore
    - Árvores balanceadas mantêm uma complexidade de acesso na ordem de  $\log_b n$  (b é a ordem da árvore), pois mantém seu crescimento controlado
  - A construção da árvore pode gerar dois tipos de árvores
    - Balanceadas: as folhas estão quase na mesma altura
    - Desbalanceadas (ou degeneradas): as alturas das folhas divergem muito. Neste caso a complexidade pode chegar a ser linear.

# Árvores de busca

- Árvores degeneradas

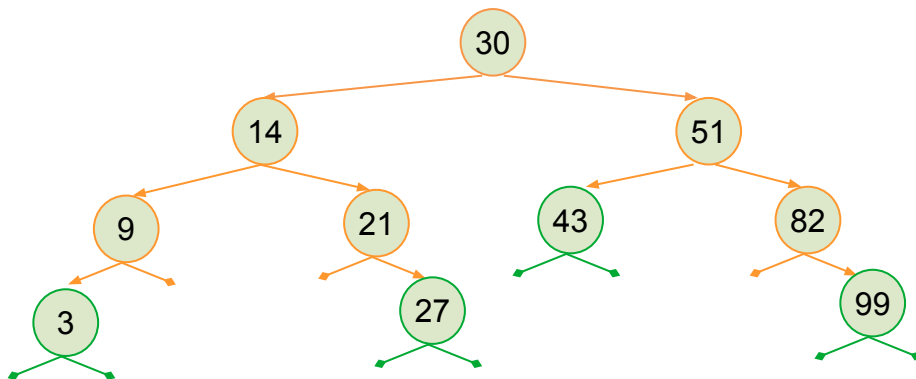


# Árvores de busca

- Dada a entrada

30	51	14	43	82	21	9	3	27	99
0	1	2	3	4	5	6	7	8	9

- Utilizando o nosso algoritmo (menor ou igual à esquerda, maior à direita):



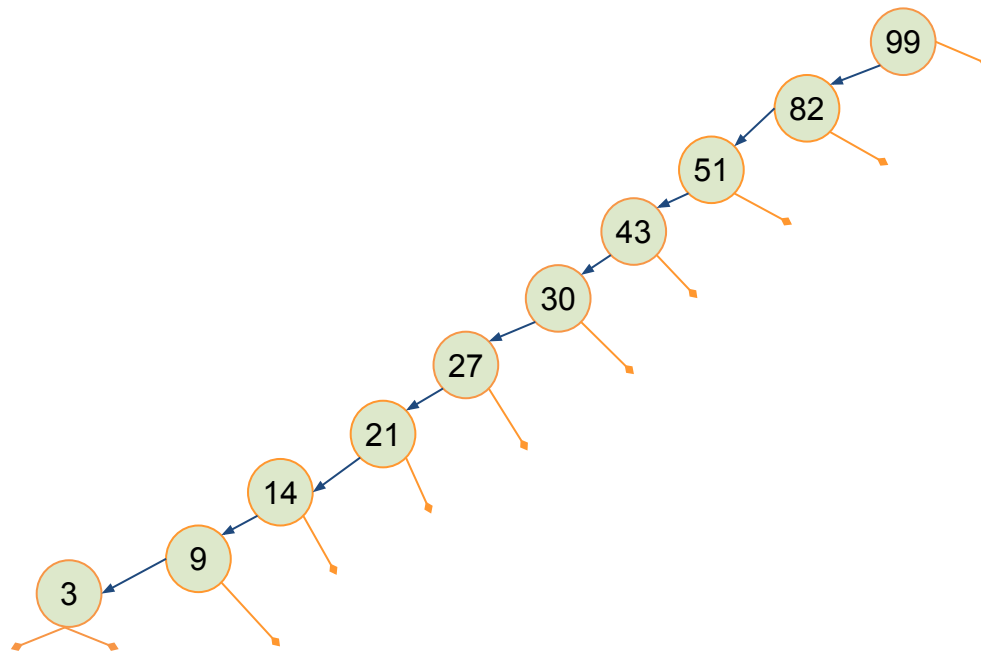
- Resultado: árvore balanceada, folhas quase na mesma altura

# Árvores de busca

- Dada a entrada:

99	82	51	43	30	27	21	14	9	3
0	1	2	3	4	5	6	7	8	9

- Resultado: árvore degenerada



# Árvores de busca

- Dada a entrada:

99	82	51	43	30	27	21	14	9	3
0	1	2	3	4	5	6	7	8	9

- Resultado: árvore degenerada
- Tem solução?
  - Algoritmos de balanceamento: AVL e Red-Black

<https://www.cs.usfca.edu/~galles/visualization/AVLtree.html>

