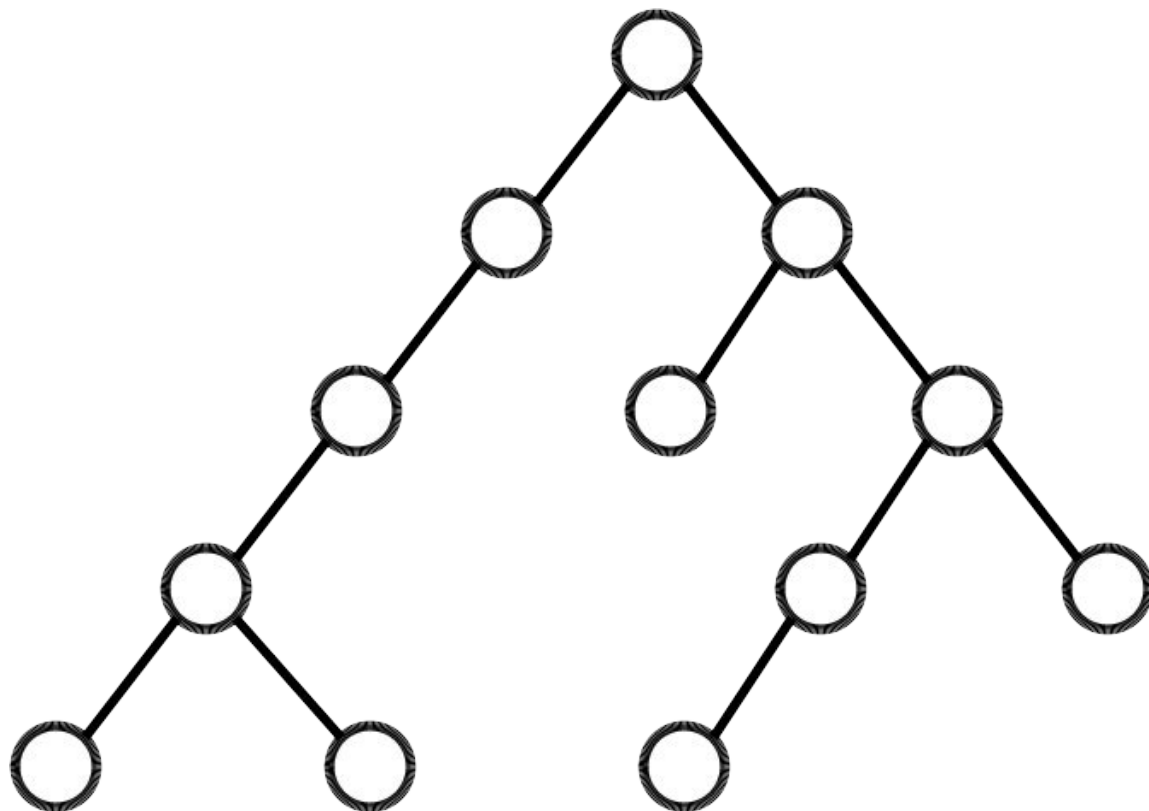


Árvores Binárias

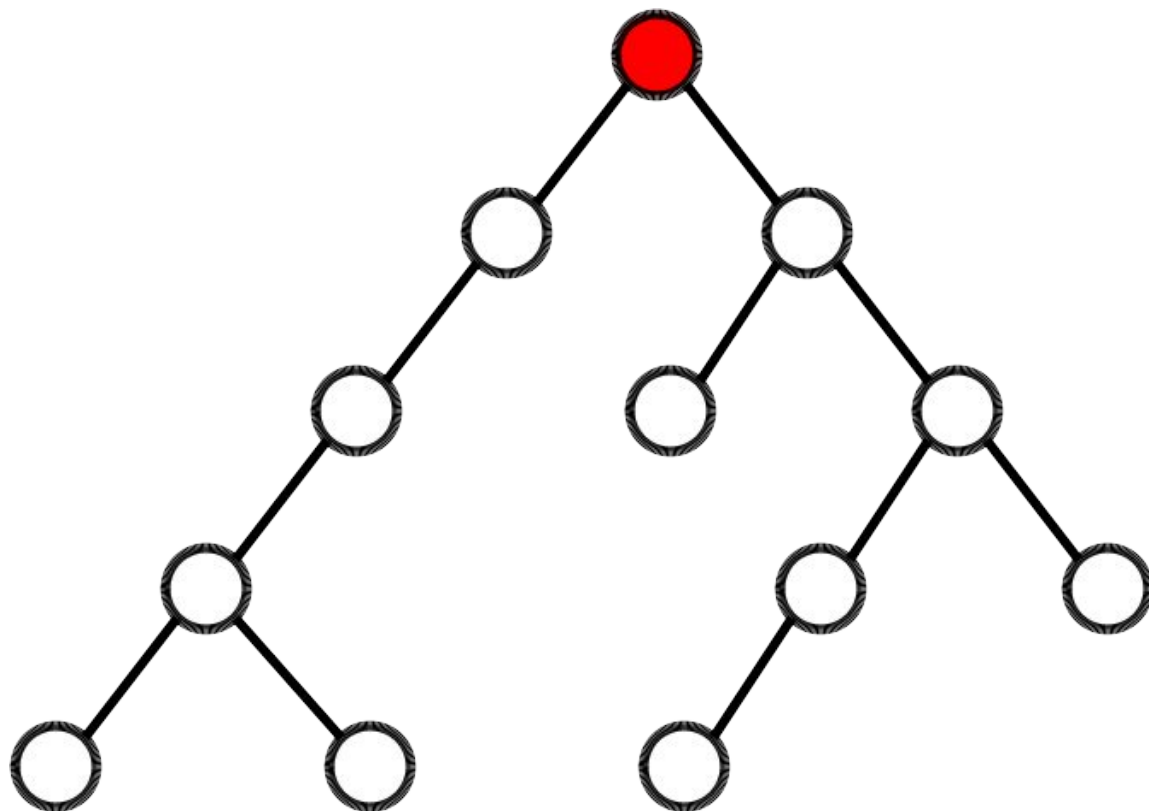
Algoritmos e Estruturas de Dados 2
2017-1

Flavio Figueiredo (<http://flaviovdf.github.io>)

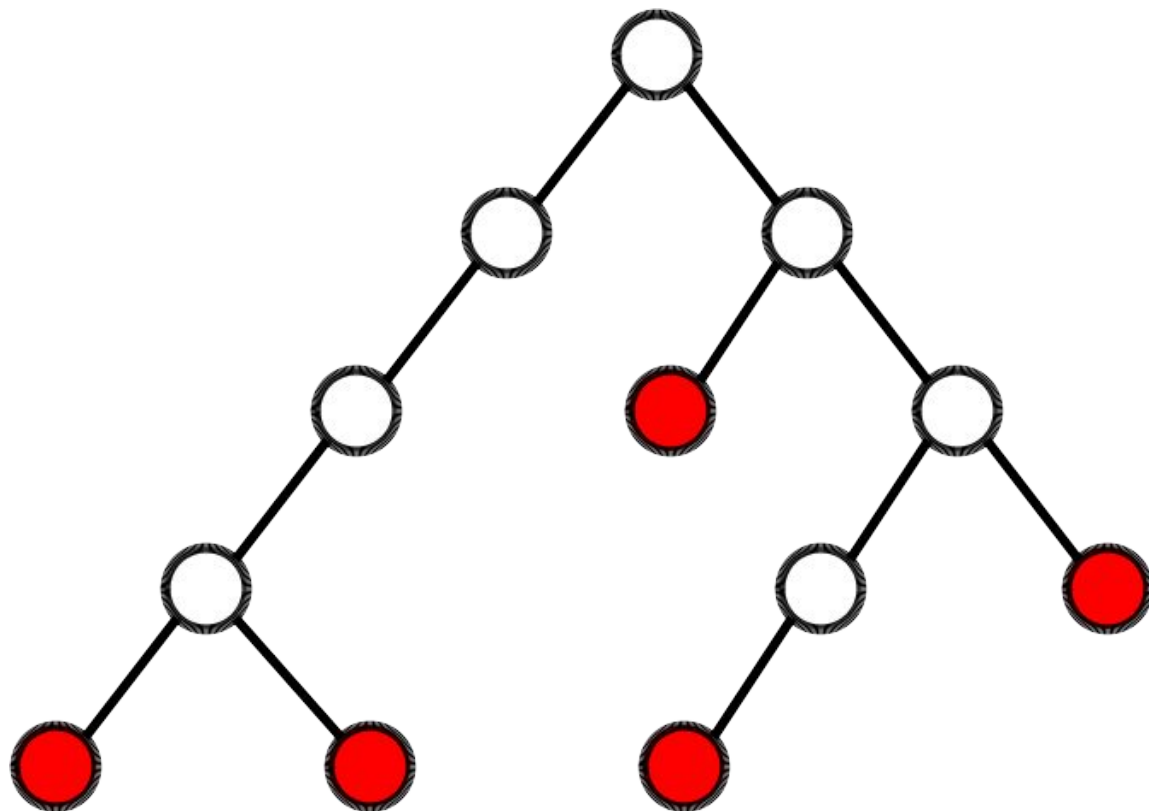
Árvores



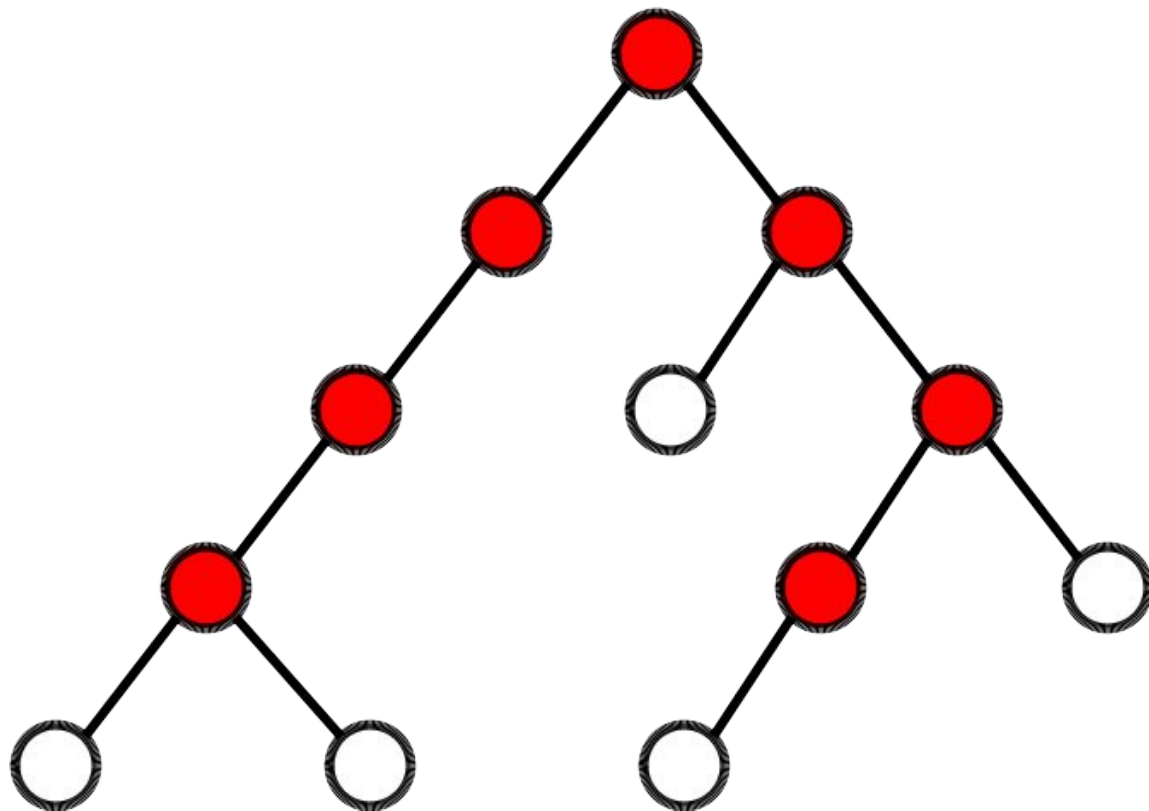
Raíz



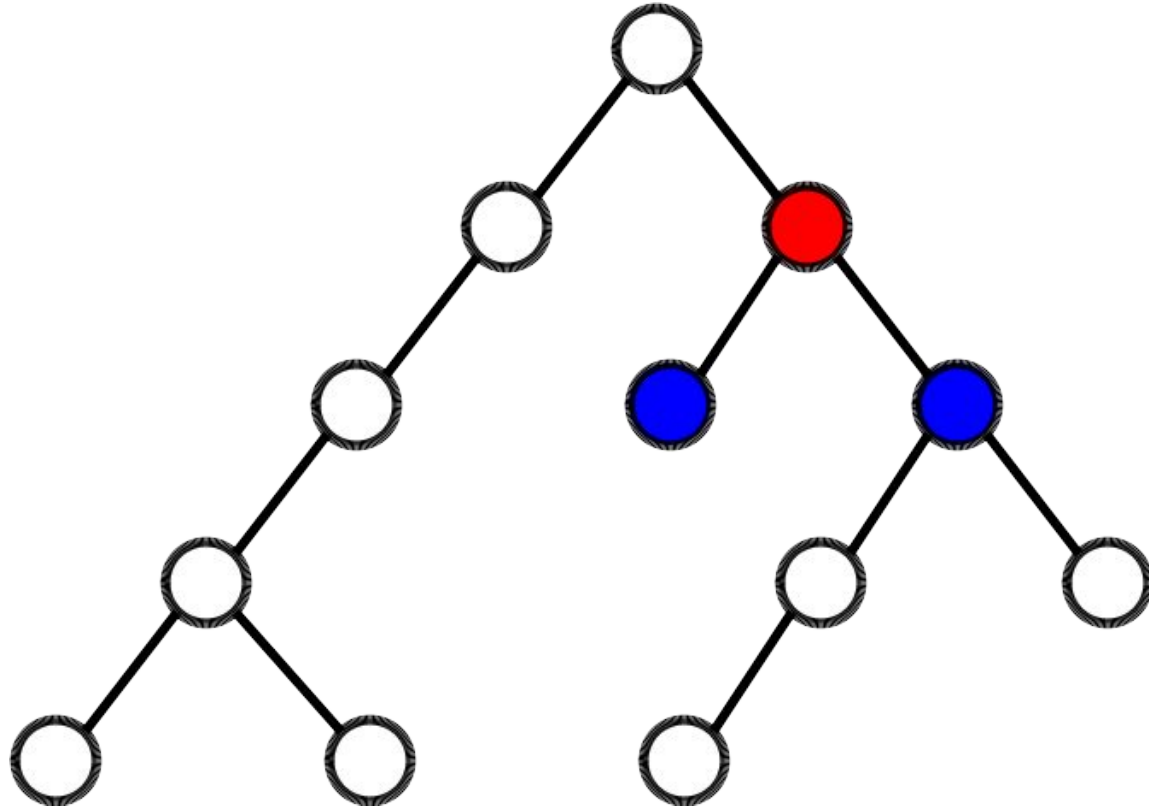
Folhas



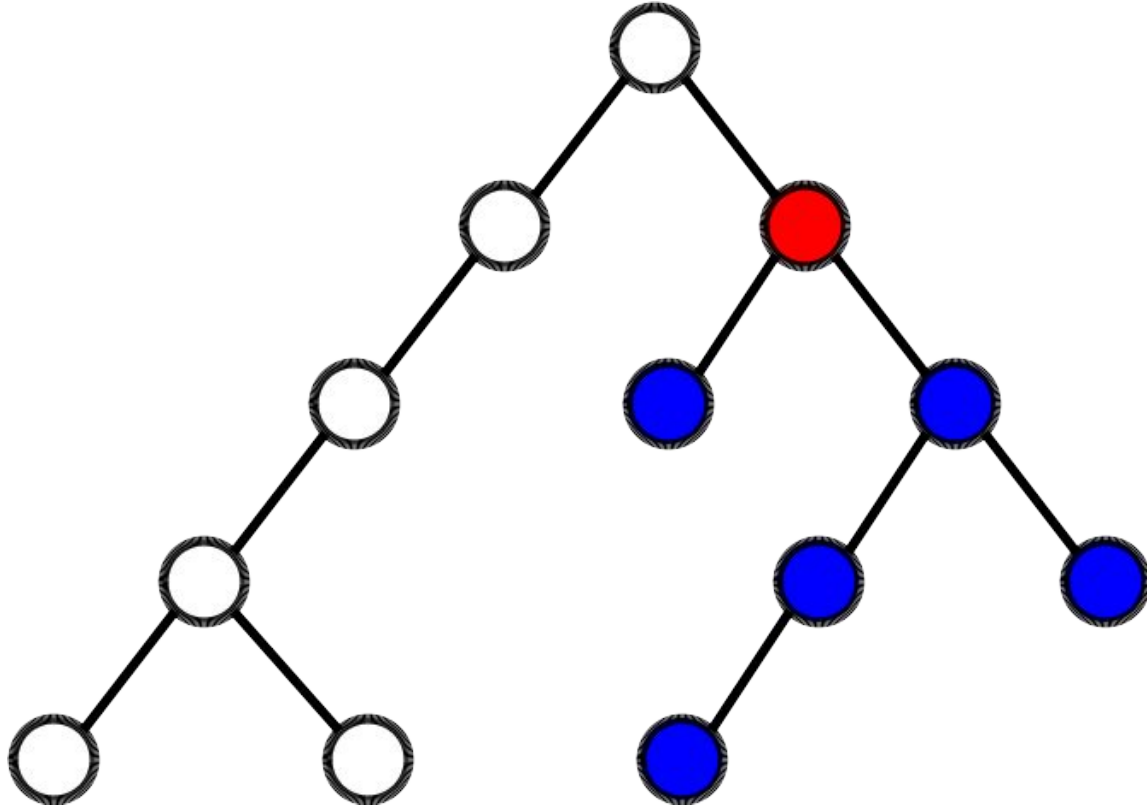
Nós Internos



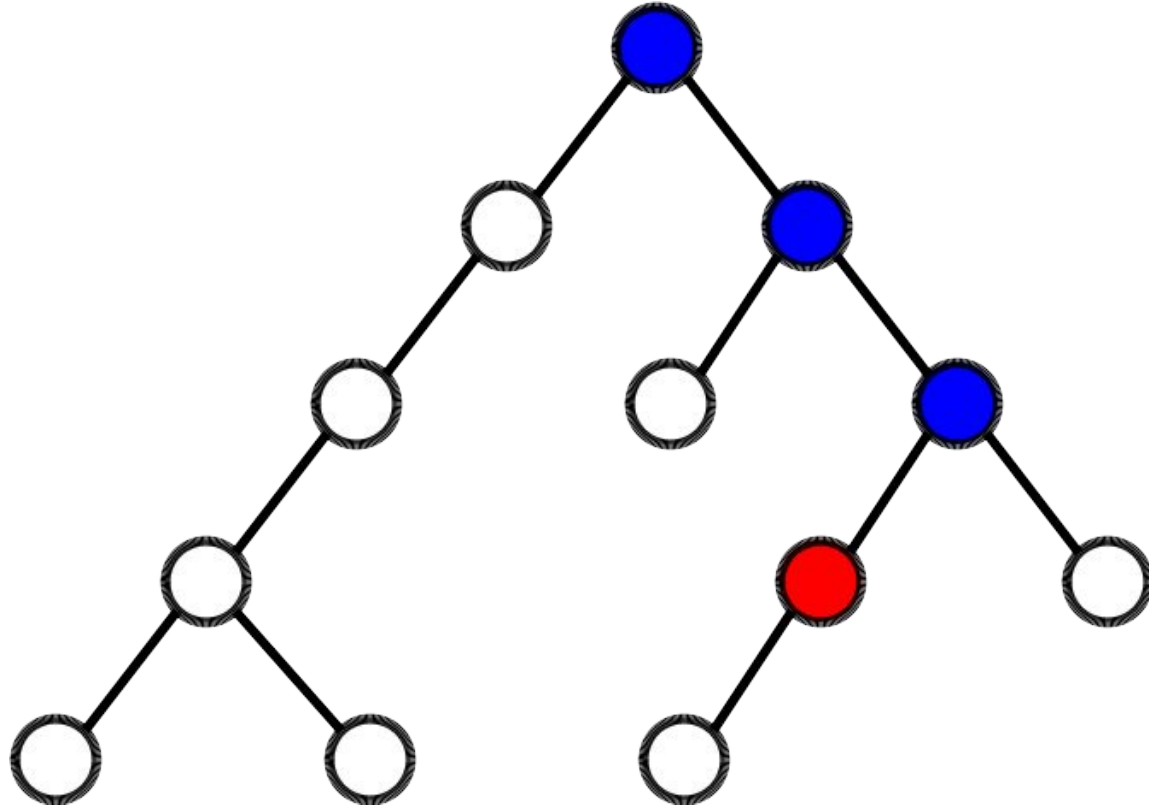
Pais e Filhos



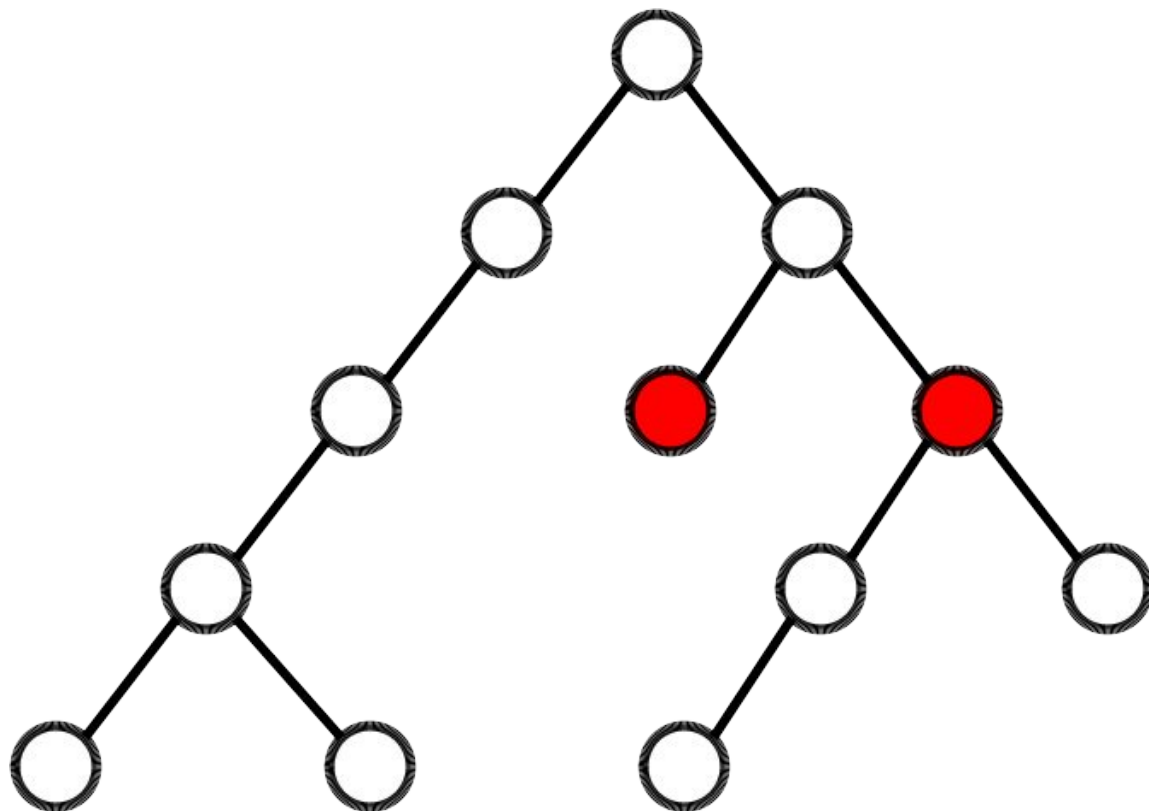
Descendentes



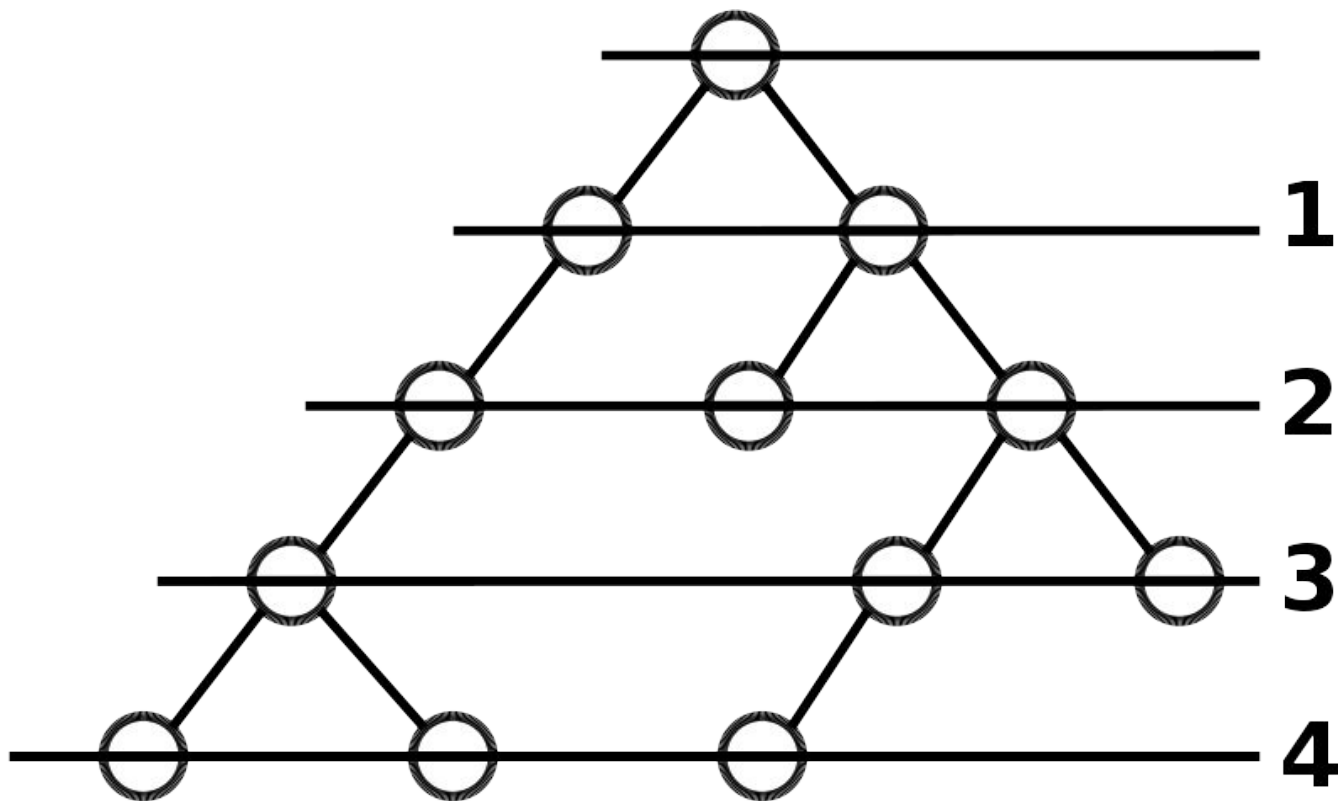
Ancestrais



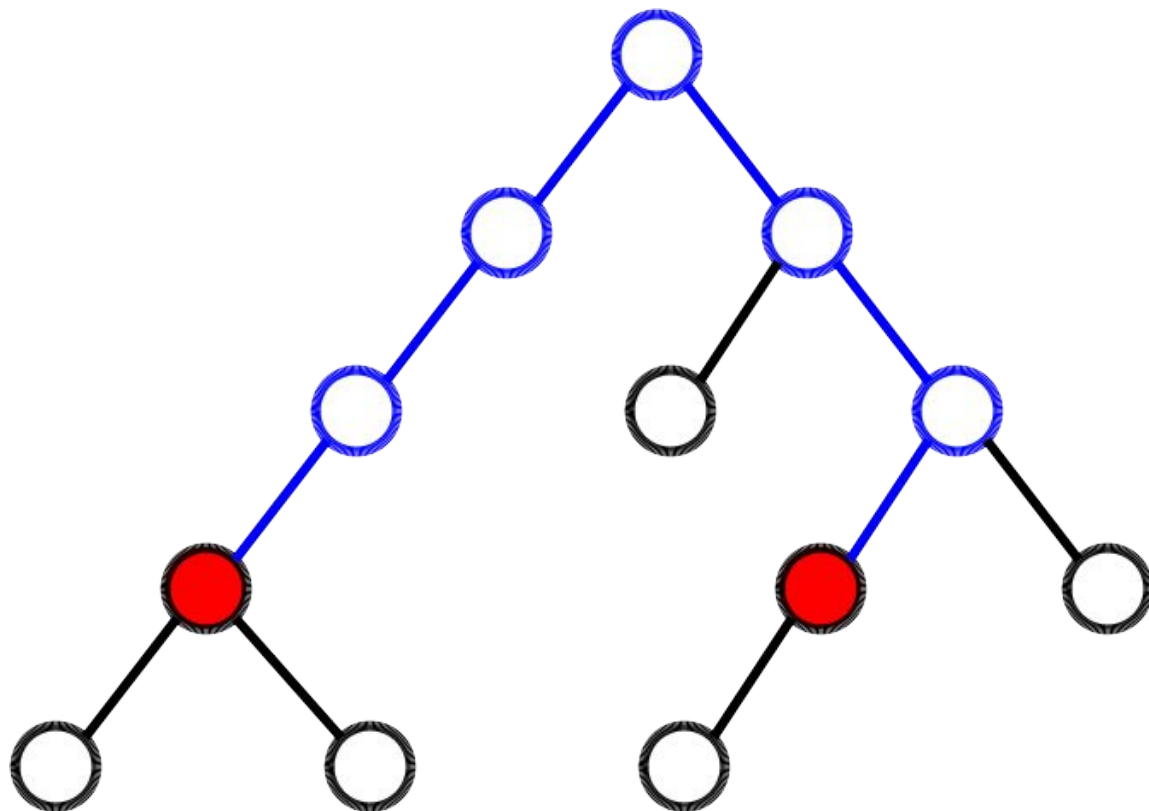
Irmãos



Níveis



Caminhos



Árvores

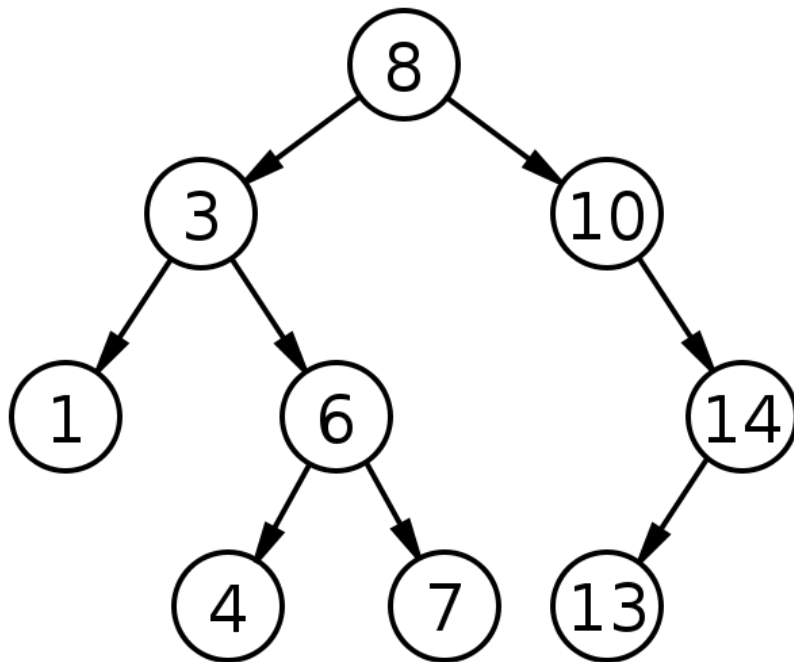
- Utilizadas para gerenciar hierarquias
- Bastante comuns na computação
- Árvore de Decisão
- Compactação
- Busca de prefixos
- Jogos

Propriedade Interessantes

- Todo nó representa uma sub-árvore
- Raiz é o nó
- Recursividade facilita nossa vida!

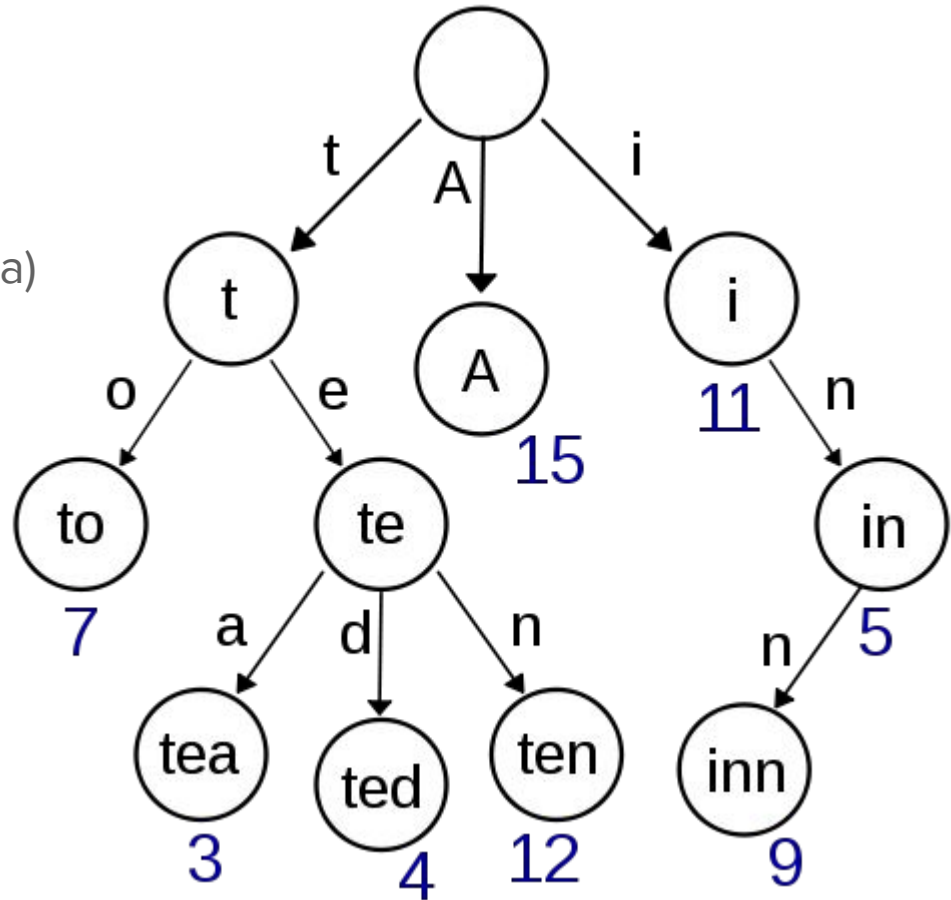
Árvore Binária

- Apenas 2 descendentes imediatos por nó



Árvore n-ária

- $n \geq 0$ descendentes imediatos
- Exemplo: Trie (não veremos agora)



Árvore Binária de Pesquisa/Busca (Binary Search Tree - BST)

- Invariantes:
 - O filho da esquerda é menor ou igual ao nó
 - O filho da direita é maior do que o nó
- Consequências:
 - Todo elemento na esquerda é menor do que o nó
 - Todo elemento na direita é maior do que o nó



TAD

- Vamos focar mais nas funções destacadas
 - Inserir
 - Remover
 - Achar Elemento
- Criar e Free similar aos TADs anteriores
- Ver código
 - <https://github.com/flaviovd/AEDS2-2017-1/tree/master/exemplos/arvores>

```
#ifndef BST_H
```

```
#define BST_H
```

```
typedef struct node {  
    int value;  
    struct node *leftChild;  
    struct node *rightChild;  
} node_t;
```

```
typedef struct {  
    node_t *root;  
} bst_t;
```

```
bst_t *createTree();  
void insertValue(bst_t *tree, int value);  
int hasValue(bst_t *tree, int value);  
int removeValue(bst_t *tree, int value);  
void bstFree(bst_t *tree);  
#endif
```

Iniciando a Árvore

```
bst_t      *createTree()      {
    bst_t *tree = (bst_t *) malloc(sizeof(bst_t));
    if (tree == NULL) {
        printf("Malloc error! Cannot create tree");
        exit(1);
    }
    tree->root = NULL;
    return tree;
}
```



Como Encontrar um Elemento x ?

Iniciando de um nó raiz (node)

1. `node->value == x?`

- Achamos o nó!

2. `node->value < x?`

- Passo para a esquerda

3. `node->value < x?`

- Passo para a direita

Como Encontrar um Elemento x ?

Iniciando de um nó raiz (node)

1. `node->value == x?`

- Achemos o nó!

2. `node->value < x?`

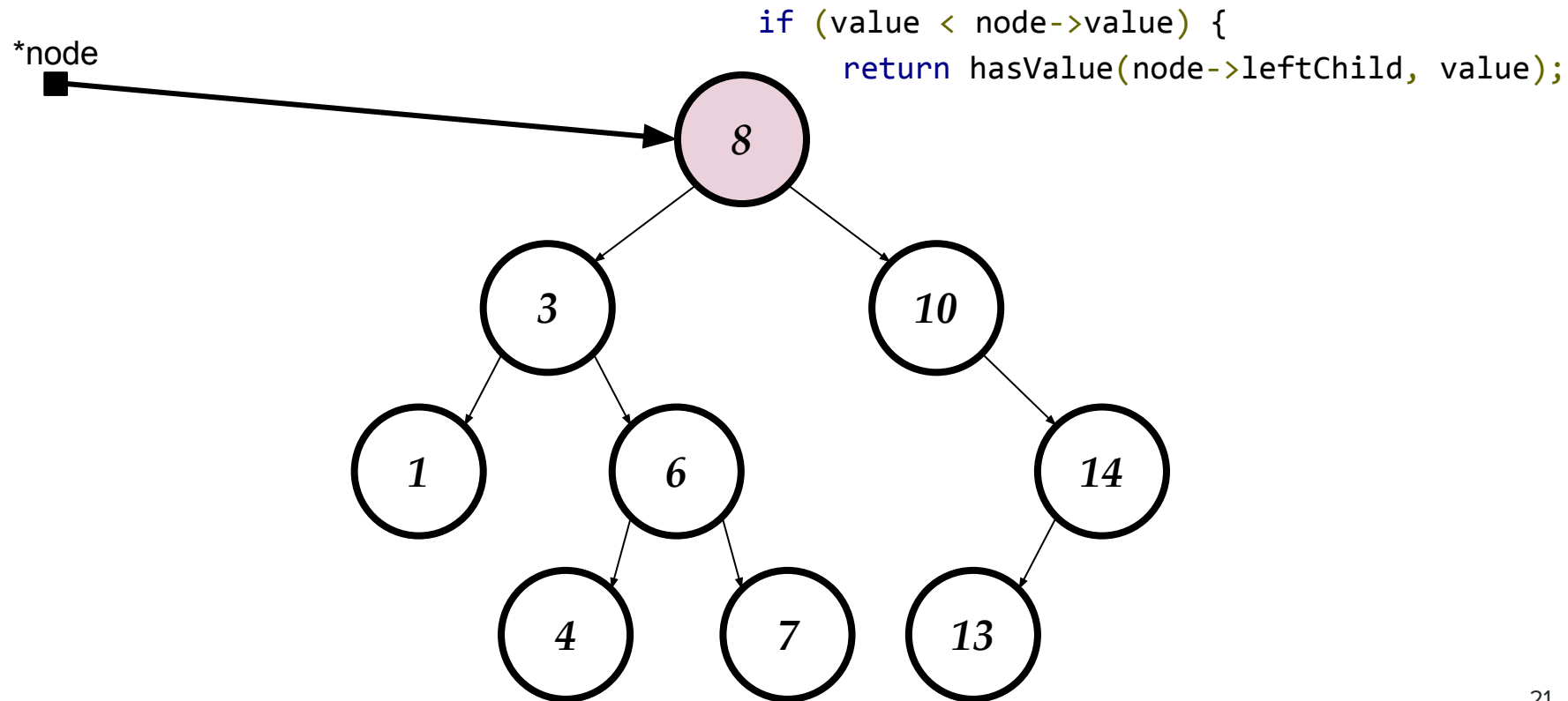
- Passo para a esquerda

3. `node->value > x?`

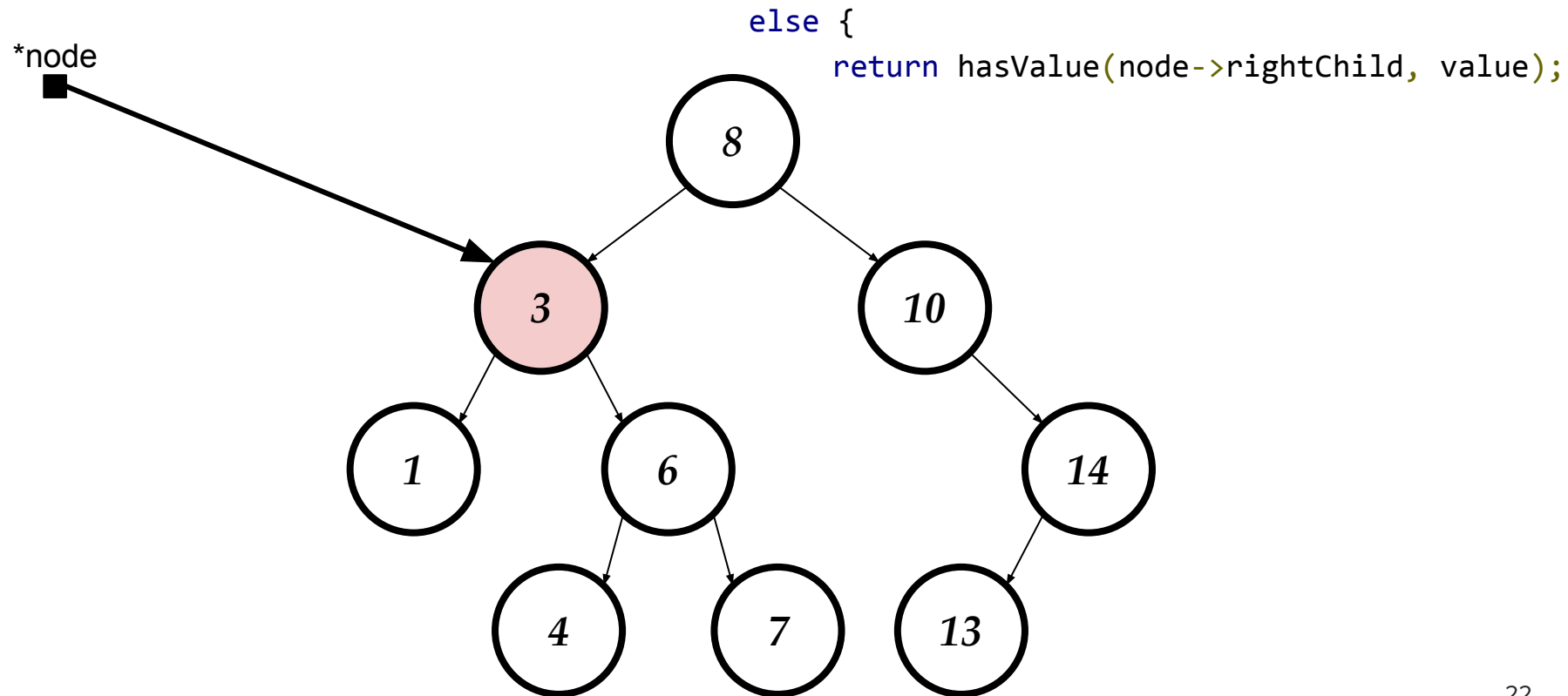
- Passo para a direita

```
int hasValue(node_t *node, int value) {  
    if (node == NULL)  
        return 0;  
    if (value == node->value)  
        return 1;  
  
    if (value < node->value) {  
        return hasValue(node->leftChild, value);  
    } else {  
        return hasValue(node->rightChild, value);  
    }  
}
```

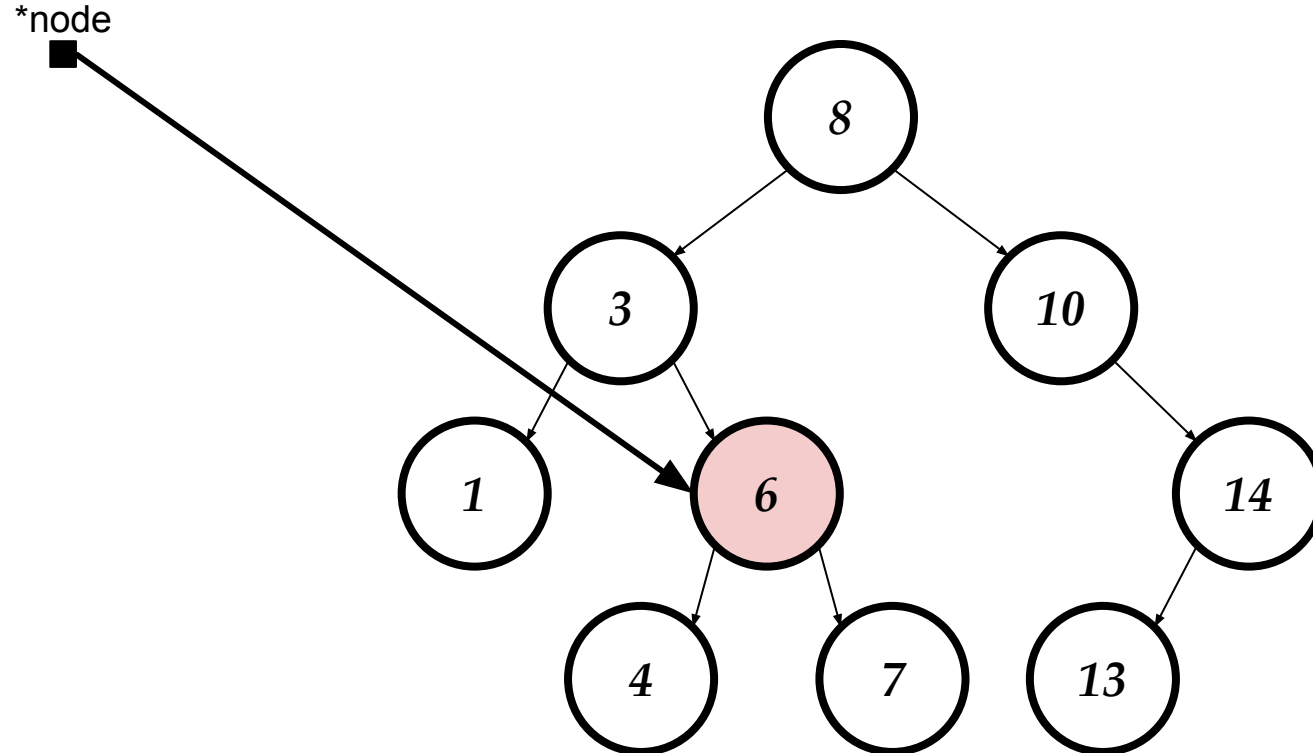
Como Encontrar um Elemento $x=4$?



Como Encontrar um Elemento $x=4$?

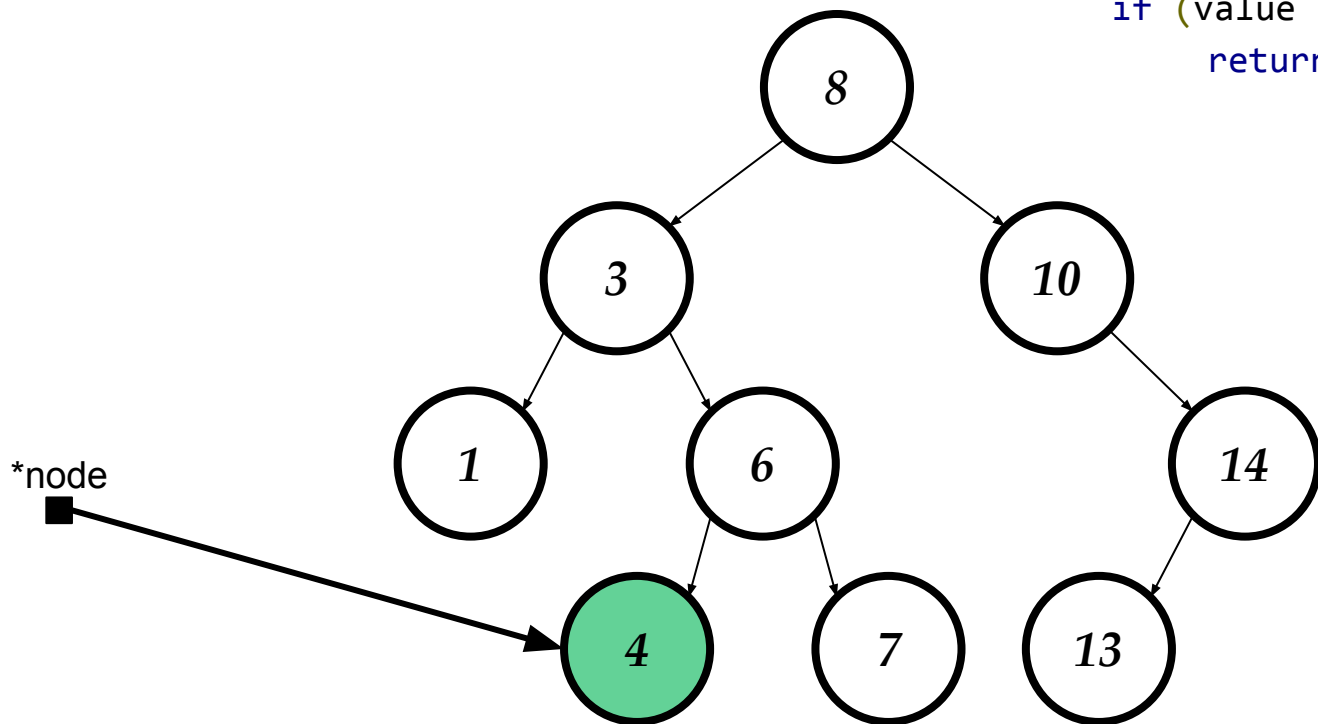


Como Encontrar um Elemento $x=4$?



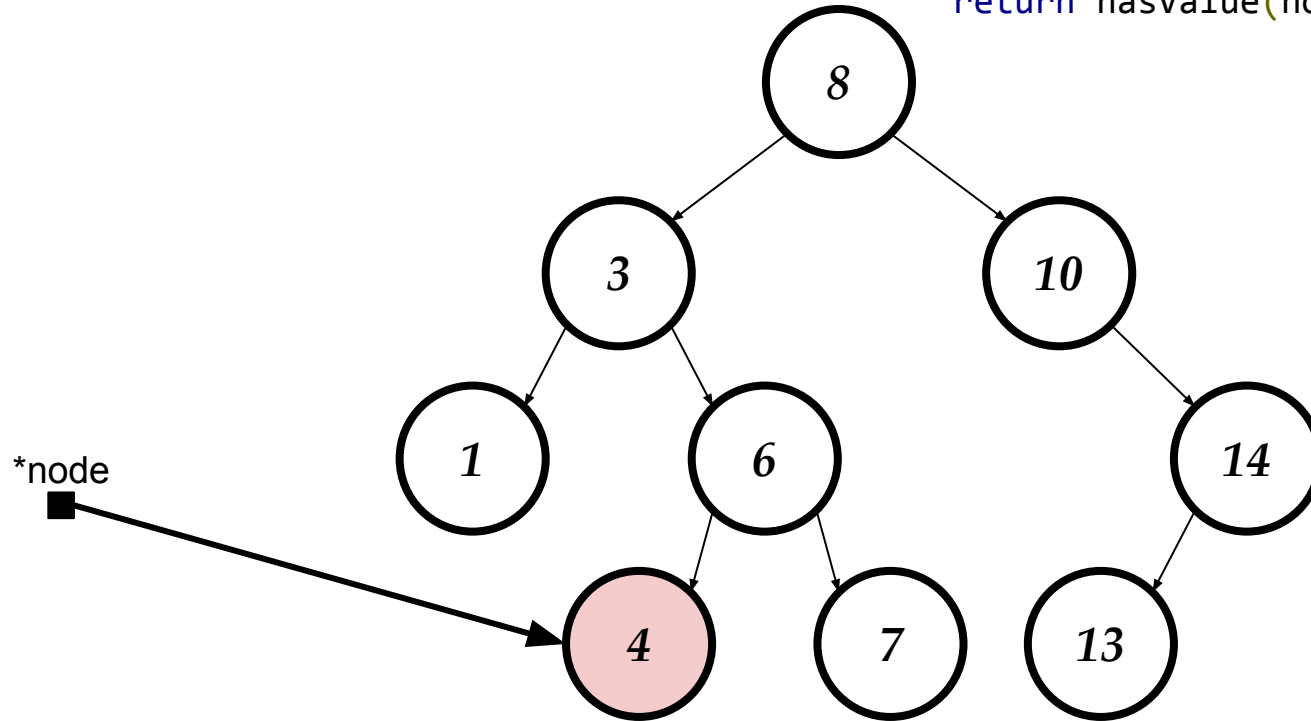
Como Encontrar um Elemento $x=4$?

```
if (value == node->value)  
    return 1;
```



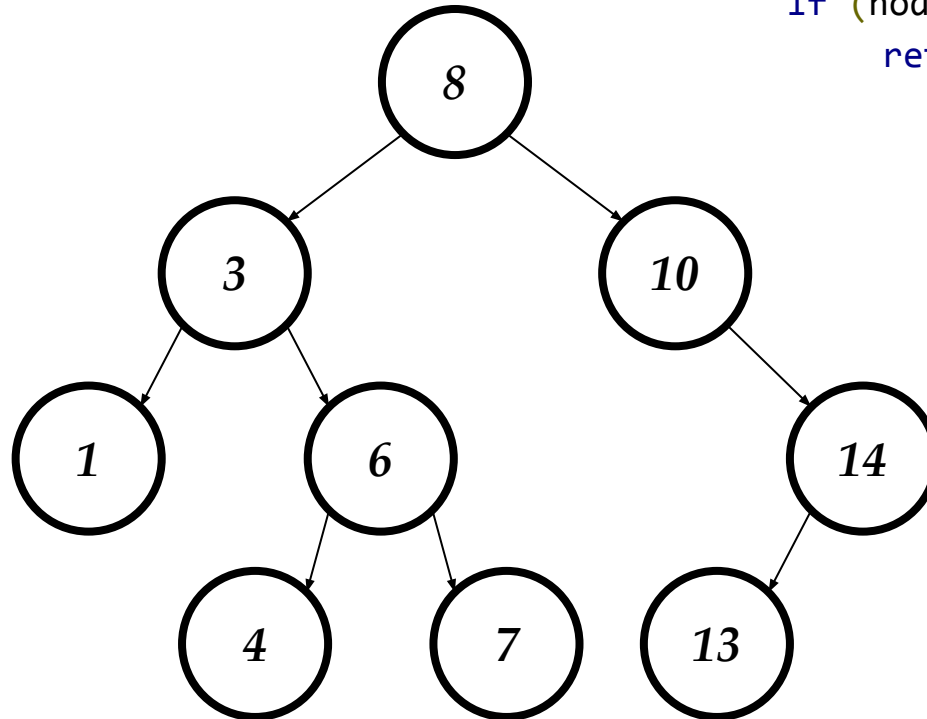
Caso não exista? $x=3$?

```
if (value < node->value) {  
    return hasValue(node->leftChild, value);  
}
```



Caso não exista? $x=3$?

*node
↓
NULL



```
if (node == NULL)  
    return 0;
```

Encontrando nó

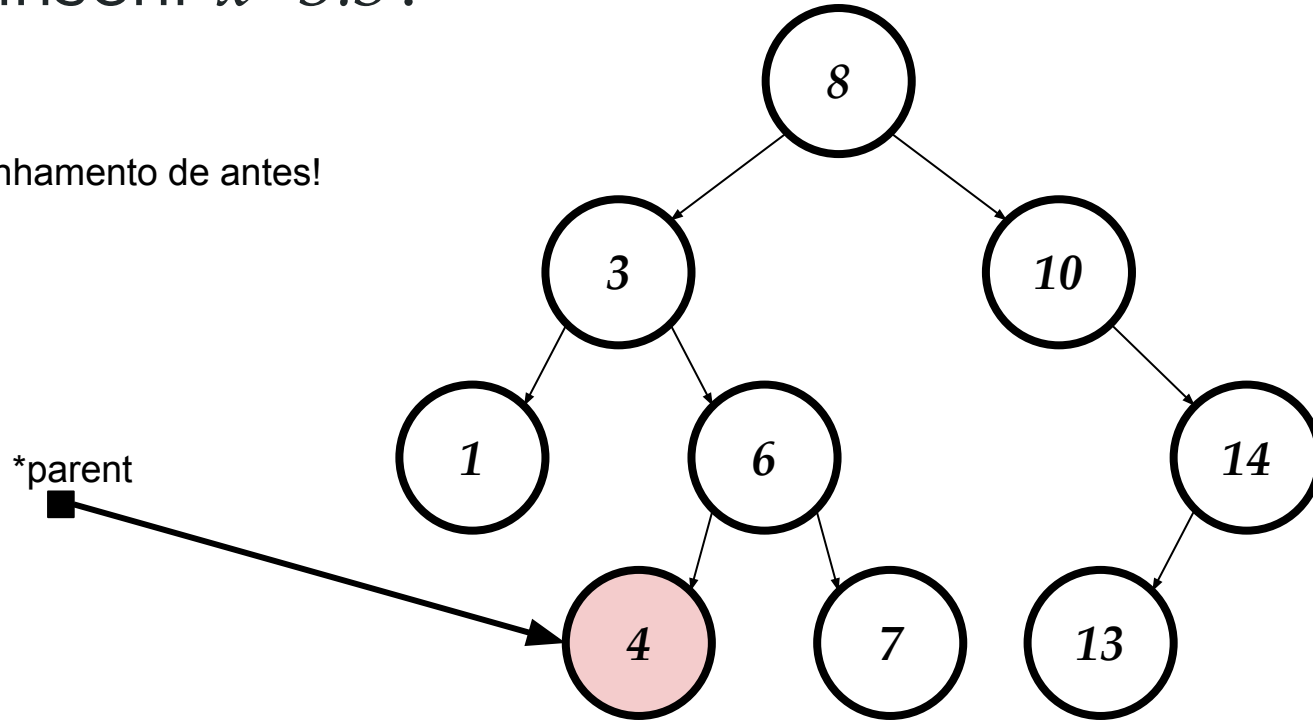
- A ideia acima é essencial para o resto das operações
- Como inserir o nó $x=3$?

Encontrando nó

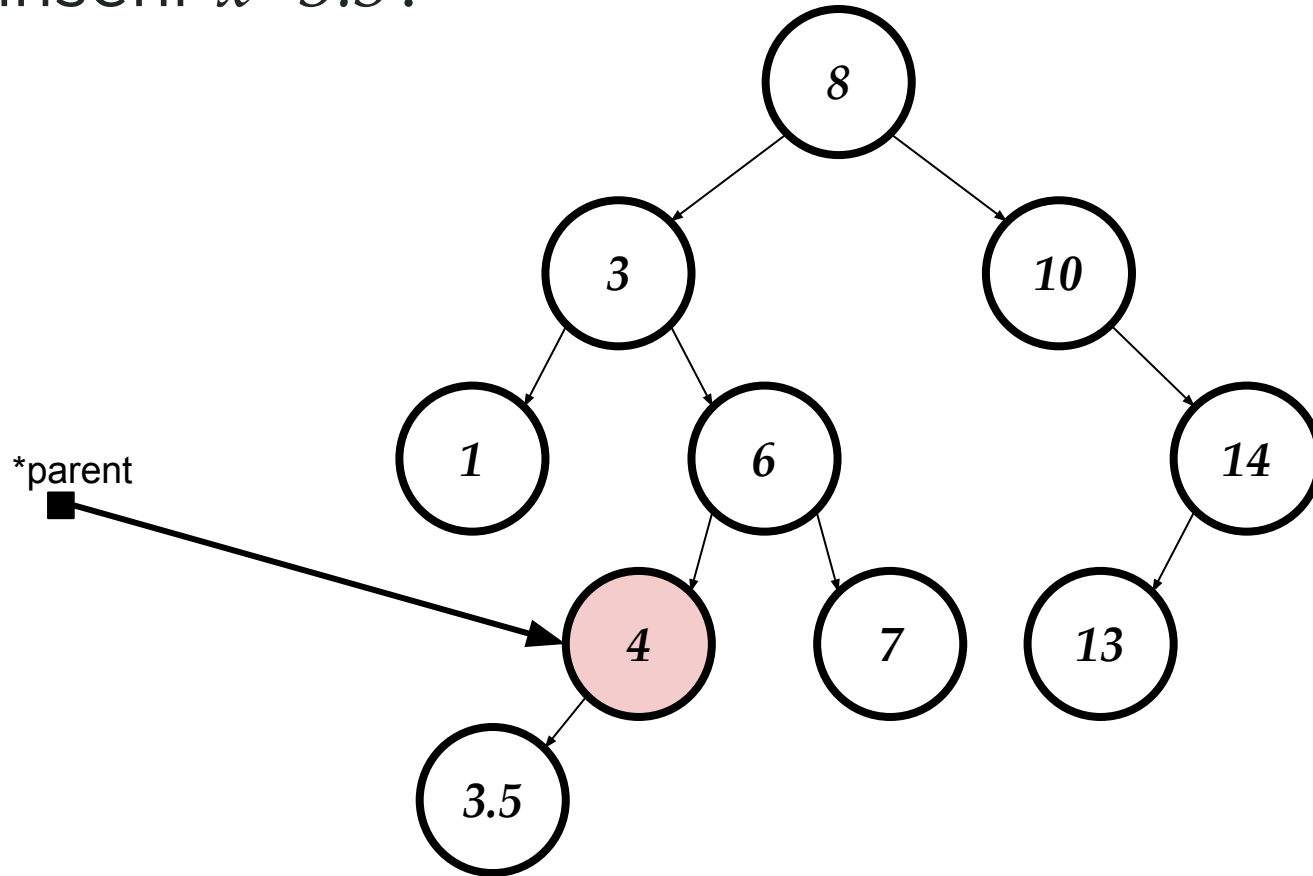
- A ideia acima é essencial para o resto das operações
- Como inserir o nó $x=3$?
- Caminhar até o ascendente imediato (pai)
- Inserir

Como inserir $x=3.5$?

Mesmo caminhamento de antes!



Como inserir $x=3.5$?



Inserções

1. Assumindo que o nó já foi alocado corretamente
 - a. Vocês já devem se acostumar com os mallocs

Inserções

1. Chamada já com novo nó alocado

```
void insertValueRecursive(node_t *parent, node_t *newNode) {  
    //Caminha para esquerda  
    if ((newNode->value <= parent->value)) {  
        if (parent->leftChild == NULL) {  
            parent->leftChild = newNode;  
        } else {  
            insertValueRecursive(parent->leftChild, newNode);  
        }  
    }  
  
    //Caminha para direita  
    if ((newNode->value > parent->value)) {  
        if (parent->rightChild == NULL) {  
            parent->rightChild = newNode;  
        } else {  
            insertValueRecursive(parent->rightChild, newNode);  
        }  
    }  
}
```


Inserções

1. Chamada já com novo nó alocado
2. Insere nó caso encontramos NULL

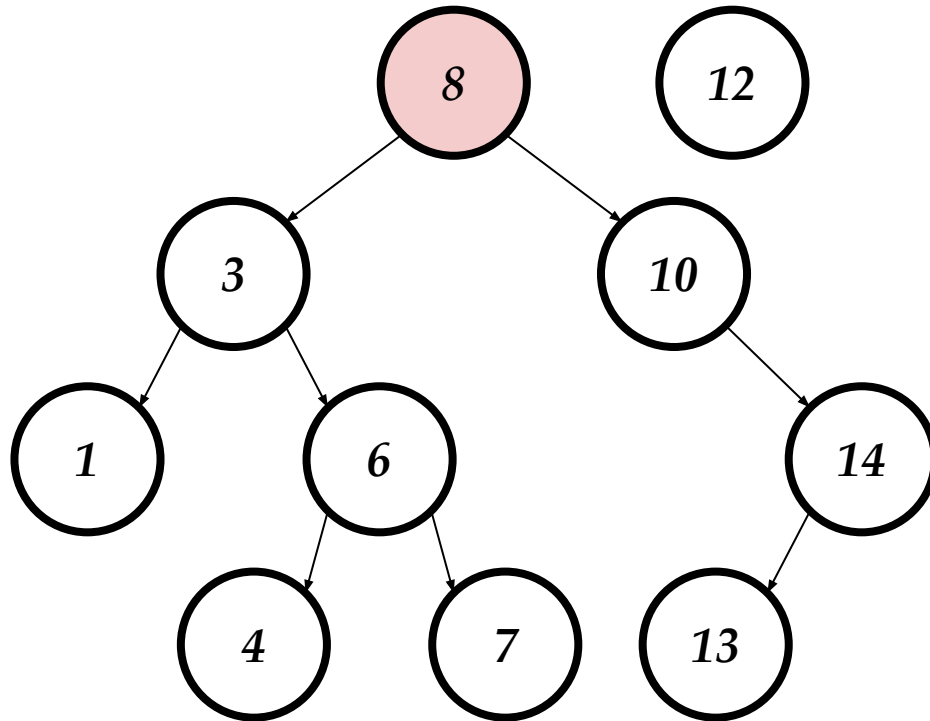
```
void insertValueRecursive(node_t *parent, node_t *newNode) {  
    //Caminha para esquerda  
    if ((newNode->value <= parent->value)) {  
        if (parent->leftChild == NULL) {  
            parent->leftChild = newNode;  
        } else {  
            insertValueRecursive(parent->leftChild, newNode);  
        }  
    }  
  
    //Caminha para direita  
    if ((newNode->value > parent->value)) {  
        if (parent->rightChild == NULL) {  
            parent->rightChild = newNode;  
        } else {  
            insertValueRecursive(parent->rightChild, newNode);  
        }  
    }  
}
```

Inserções

1. Chamada já com novo nó alocado
2. Insere nó caso encontramos NULL
3. Caminhamento

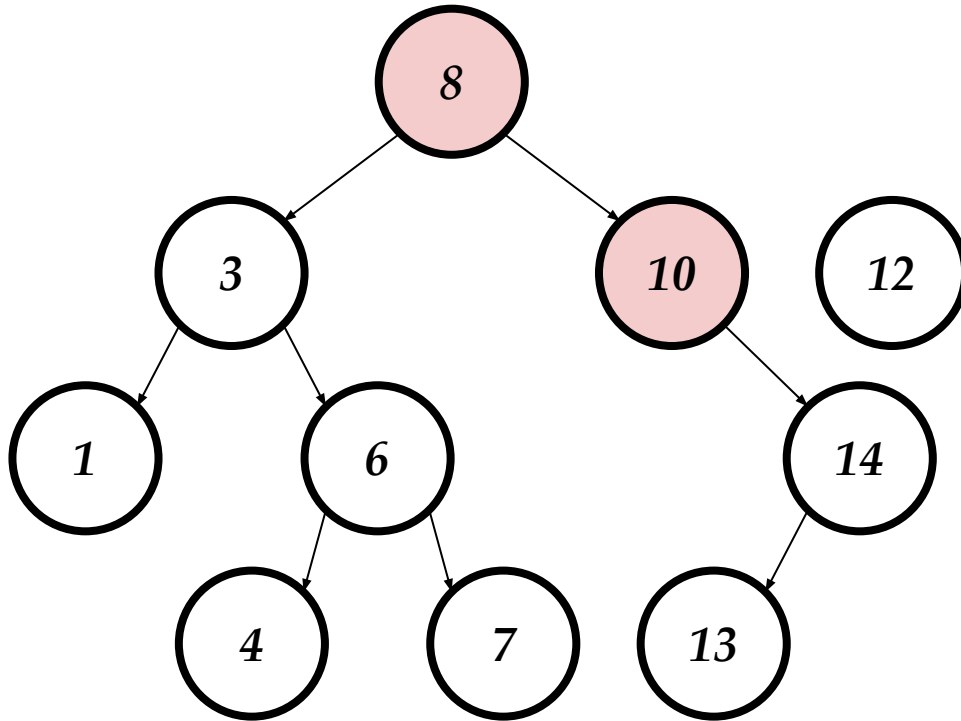
```
void insertValueRecursive(node_t *parent, node_t *newNode) {  
    //Caminha para esquerda  
    if ((newNode->value <= parent->value)) {  
        if (parent->leftChild == NULL) {  
            parent->leftChild = newNode;  
        } else {  
            insertValueRecursive(parent->leftChild, newNode);  
        }  
    }  
  
    //Caminha para direita  
    if ((newNode->value > parent->value)) {  
        if (parent->rightChild == NULL) {  
            parent->rightChild = newNode;  
        } else {  
            insertValueRecursive(parent->rightChild, newNode);  
        }  
    }  
}
```

Maior -> Direita



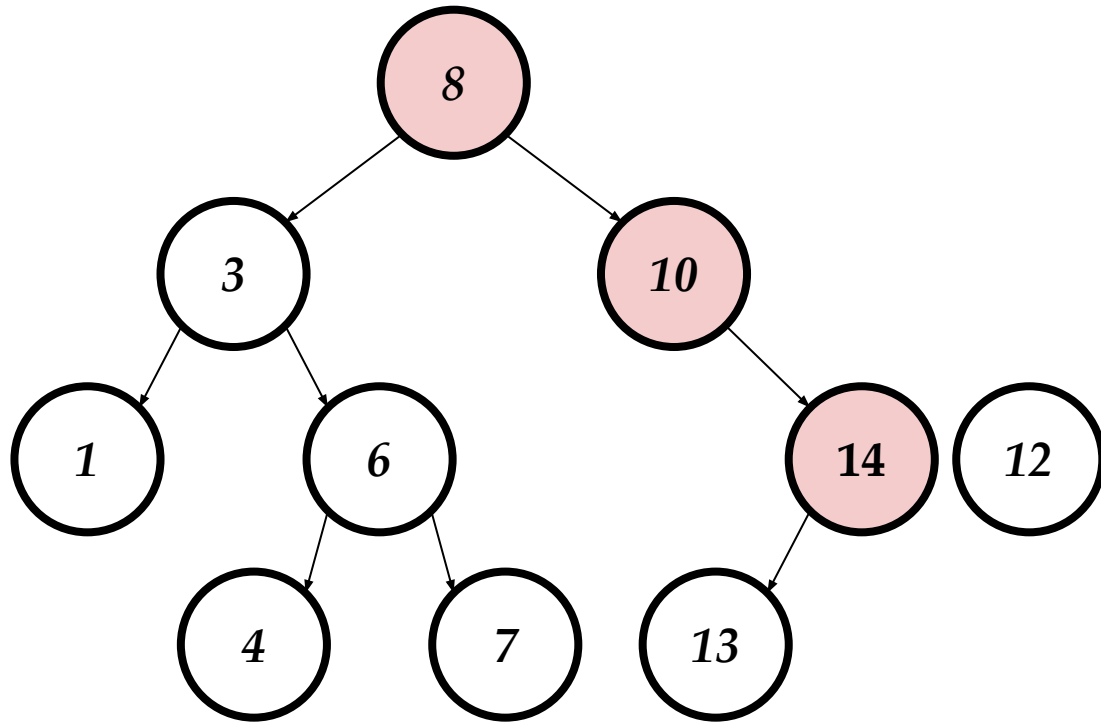
`insert({3,NULL,NULL},{8,3,10})`

Maior -> Direita



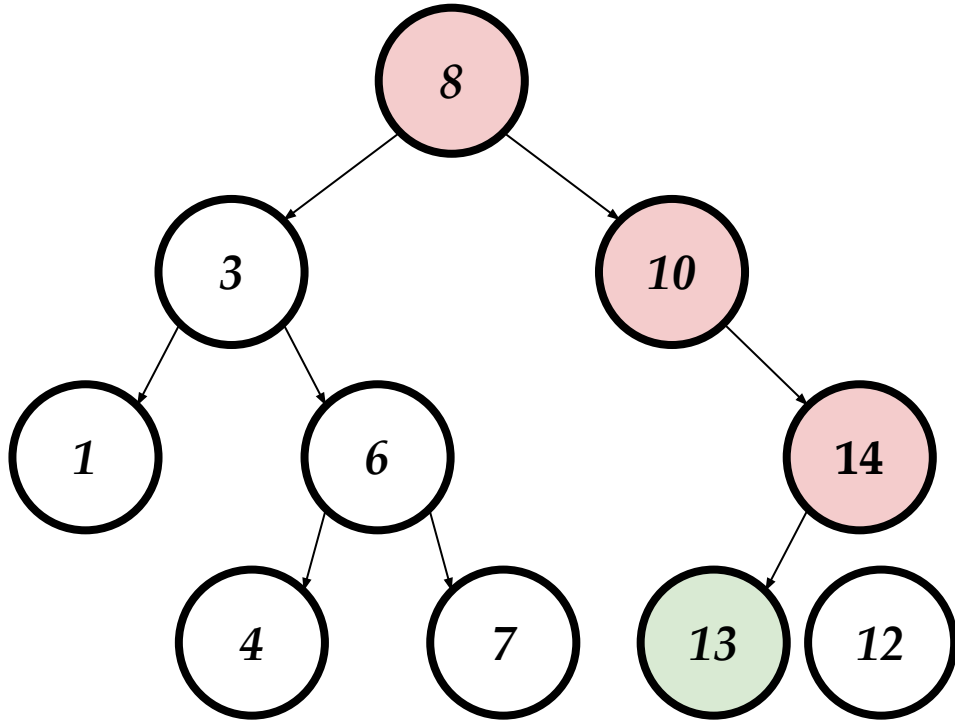
<code>insert({12,NULL,NULL},{10,NULL,14})</code>
<code>insert({12,NULL,NULL},{8,3,10})</code>
<code>insert(Node,Parent)</code>

Menor <- Esquerda



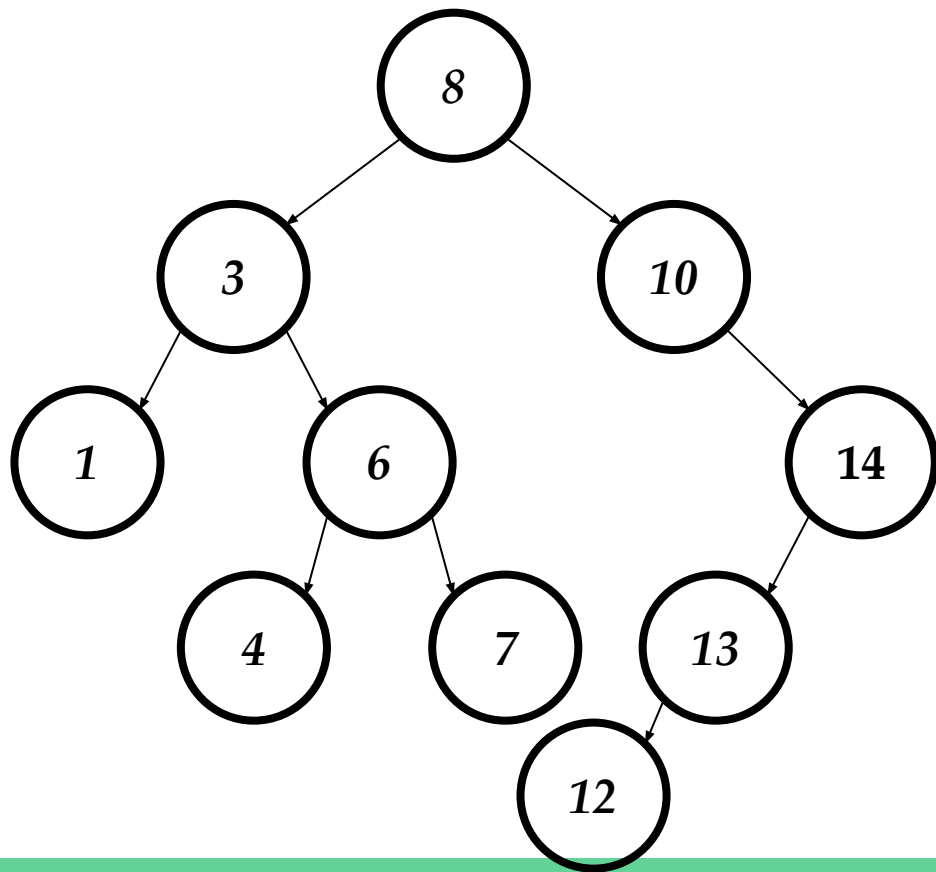
<code>insert({12,NULL,NULL},{14,13,NULL})</code>
<code>insert({12,NULL,NULL},{10,NULL,14})</code>
<code>insert({12,NULL,NULL},{8,3,10})</code>
<code>insert(Node,Parent)</code>

Menor <- Esquerda (NULL! Achamos o local)



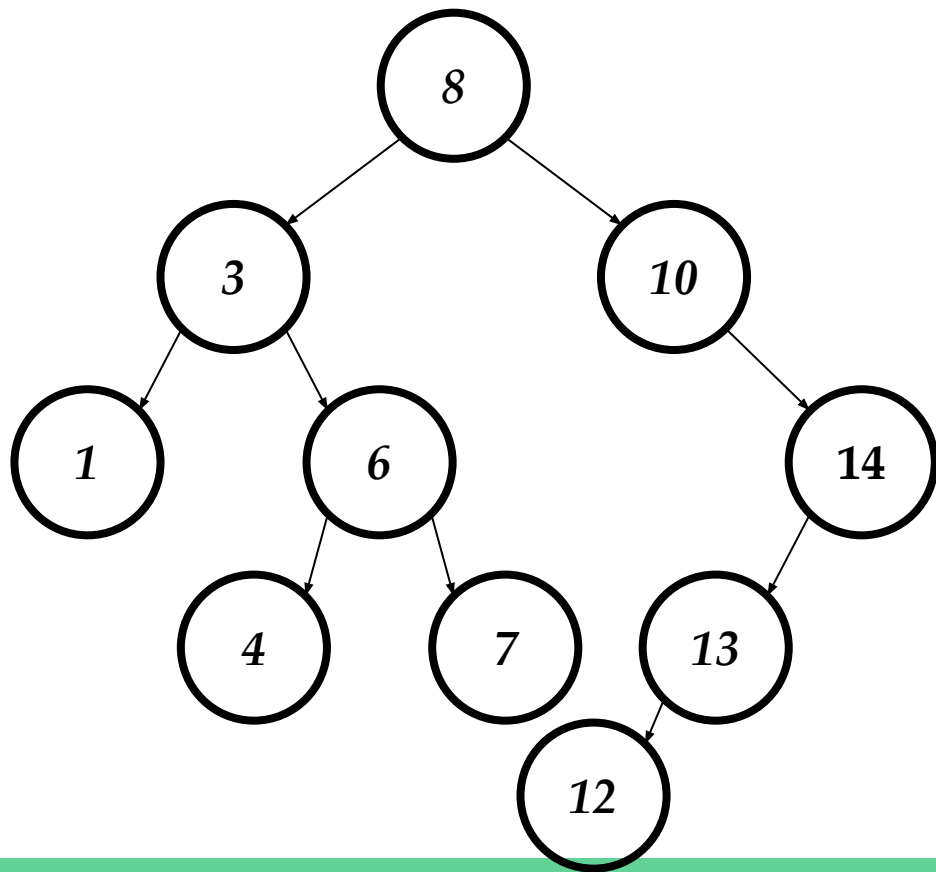
<code>insert({12,NULL,NULL},{13,NULL,NULL})</code>
<code>insert({12,NULL,NULL},{14,13,NULL})</code>
<code>insert({12,NULL,NULL},{10,NULL,14})</code>
<code>insert({12,NULL,NULL},{8,3,10})</code>
<code>insert(Node,Parent)</code>

Desempilha



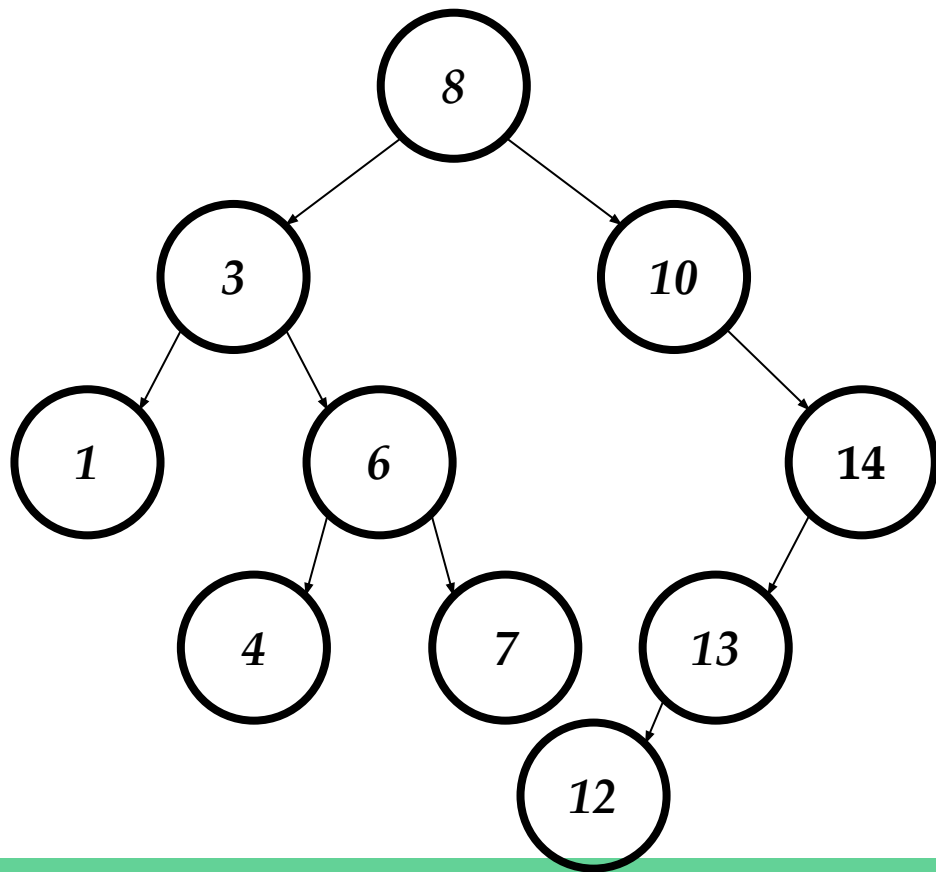
<code>insert({12,NULL,NULL},{13,NULL,NULL})</code>
<code>insert({12,NULL,NULL},{14,13,NULL})</code>
<code>insert({12,NULL,NULL},{10,NULL,14})</code>
<code>insert({12,NULL,NULL},{8,3,10})</code>
<code>insert(Node,Parent)</code>

Desempilha



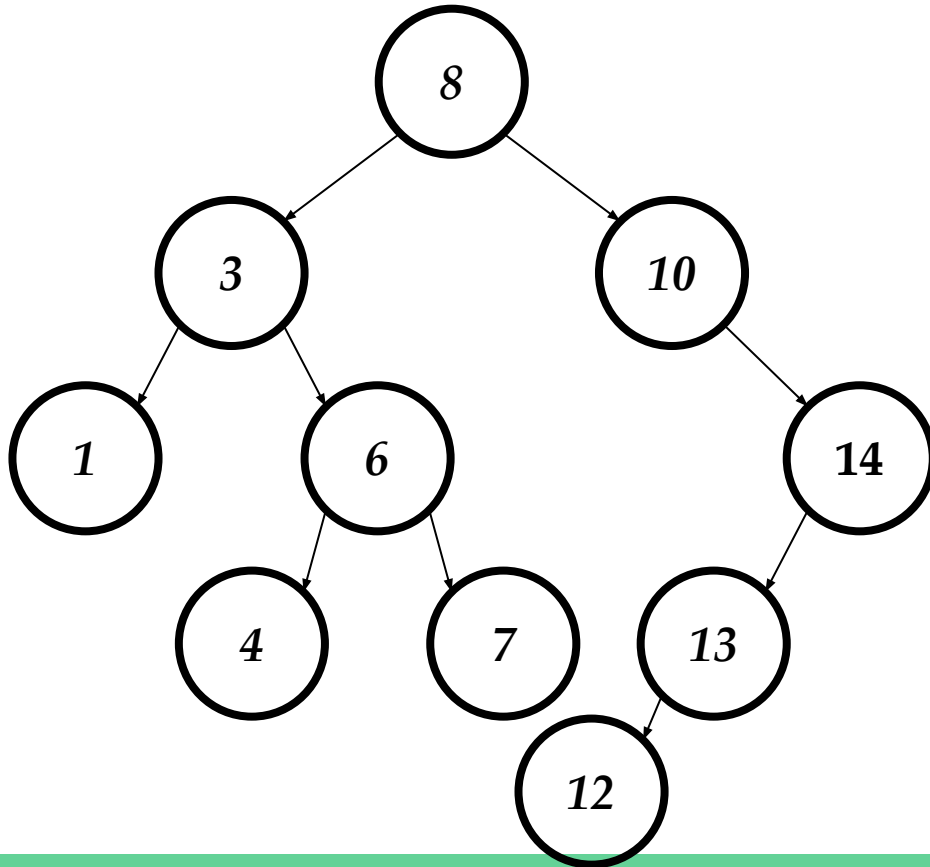
<code>insert({12,NULL,NULL},{14,13,NULL})</code>
<code>insert({12,NULL,NULL},{10,NULL,14})</code>
<code>insert({12,NULL,NULL},{8,3,10})</code>
<code>insert(Node,Parent)</code>

Desempilha



<code>insert({12,NULL,NULL},{10,NULL,14})</code>
<code>insert({12,NULL,NULL},{8,3,10})</code>
<code>insert(Node,Parent)</code>

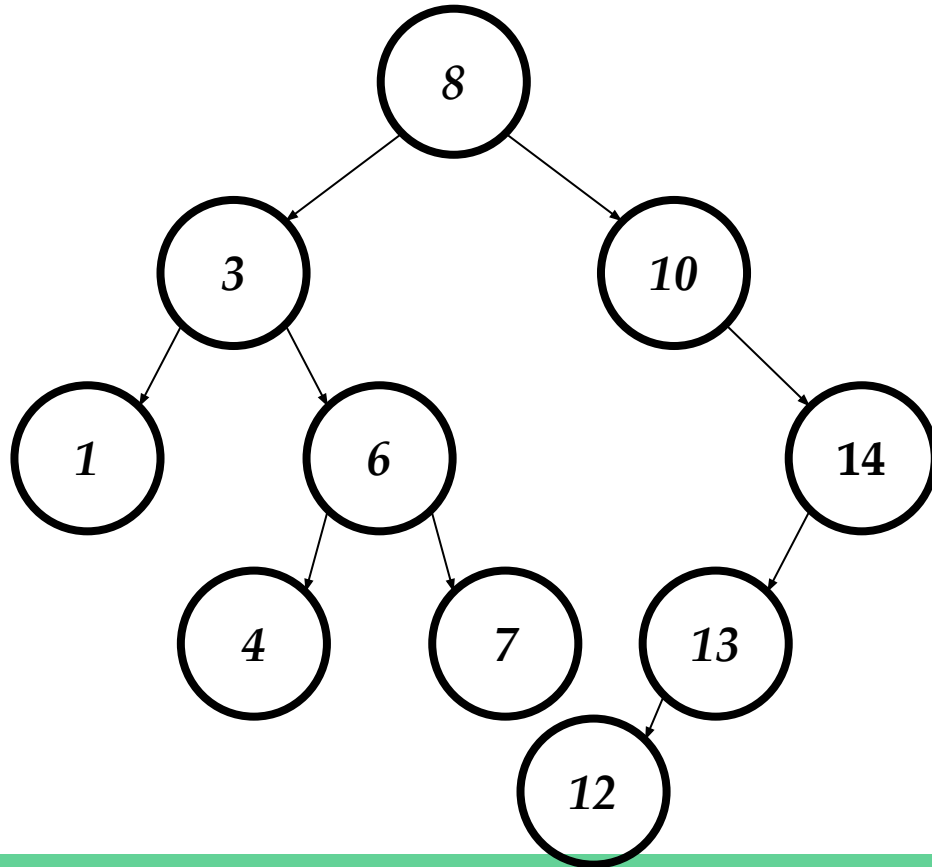
Desempilha



`insert({12,NULL,NULL},{8,3,10})`

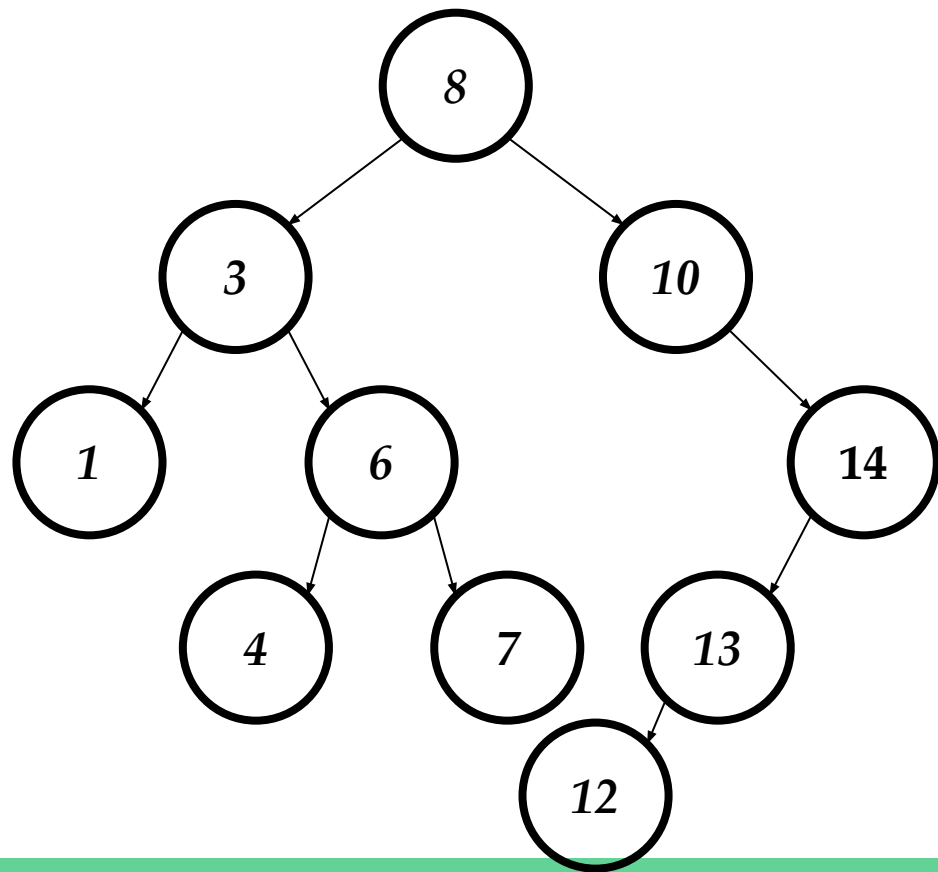
`insert(Node,Parent)`

Desempilha



`insert(Node, Parent)`

Desempilha



Recursão

- Como mencionar, árvores são “pratos cheio” para recursão
- Podemos fazer inserções sem recursão
 - Todo código recursivo pode ser feito iterativamente
 - Como?
- Ver exemplo no site
 - <https://github.com/flaviovdv/AEDS2-2017-1/tree/master/exemplos/arvores>

Remover Elementos

- Remoção na BST é mais complicado
- Precisamos considerar 3 casos:
 - a. Quando o nó é folha
 - b. Quando o nó tem apenas 1 descendente imediato
 - c. Quando o nó tem 2 descendentes imediatos

Remover Elementos

- Remoção na BST é mais complicado
- Precisamos considerar 3 casos:
 - a. Quando o nó é folha
 - Simples, free e atualizar ponteiros
 - b. Quando o nó tem apenas 1 descendente imediato
 - Vamos deixar 1 sub-árvore órfã
 - c. Quando o nó tem 2 descendentes imediatos
 - Vamos deixar 2 sub-árvore órfãs

Remover Elementos

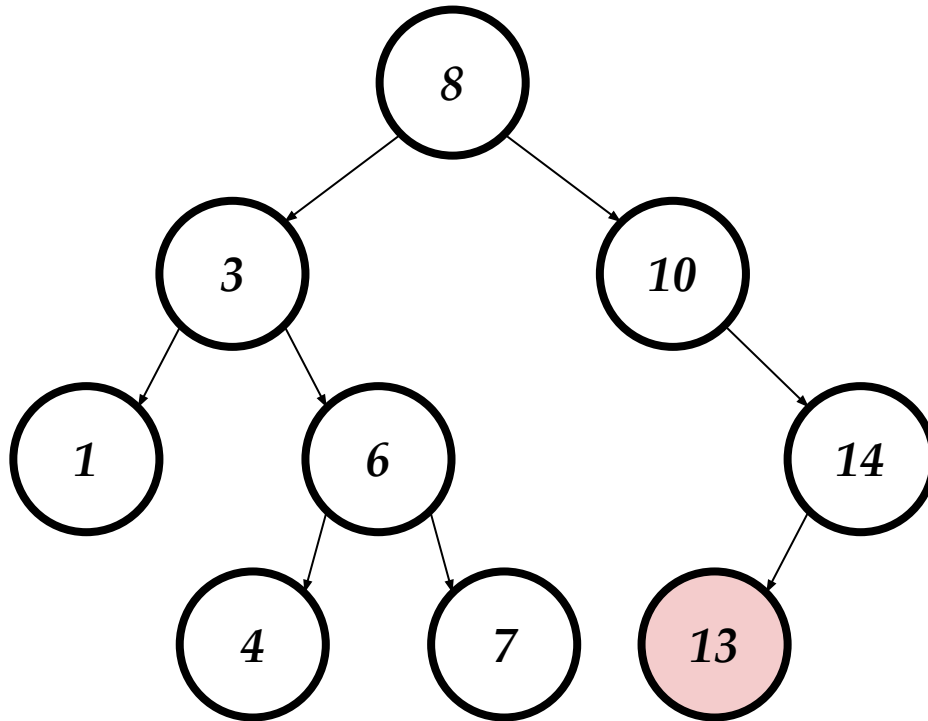
- Remoção na BST é mais complicado
- Precisamos considerar 3 casos:
 - a. Quando o nó é folha
 - Simples, free e atualizar ponteiros
 - b. Quando o nó tem apenas 1 descendente imediato
 - Vamos deixar 1 sub-árvore órfã
 - Parente imediato pode herdar
 - c. Quando o nó tem 2 descendentes imediatos
 - Vamos deixar 2 sub-árvore órfãs

Remover Elementos

- Remoção na BST é mais complicado
- Precisamos considerar 3 casos:
 - a. Quando o nó é folha
 - Simples, free e atualizar ponteiros
 - b. Quando o nó tem apenas 1 descendente imediato
 - Vamos deixar 1 sub-árvore órfã
 - Parente imediato pode herdar
 - c. Quando o nó tem 2 descendentes imediatos
 - Vamos deixar 2 sub-árvores órfãs]
 - Precisamos de cuidado para manter a invariante



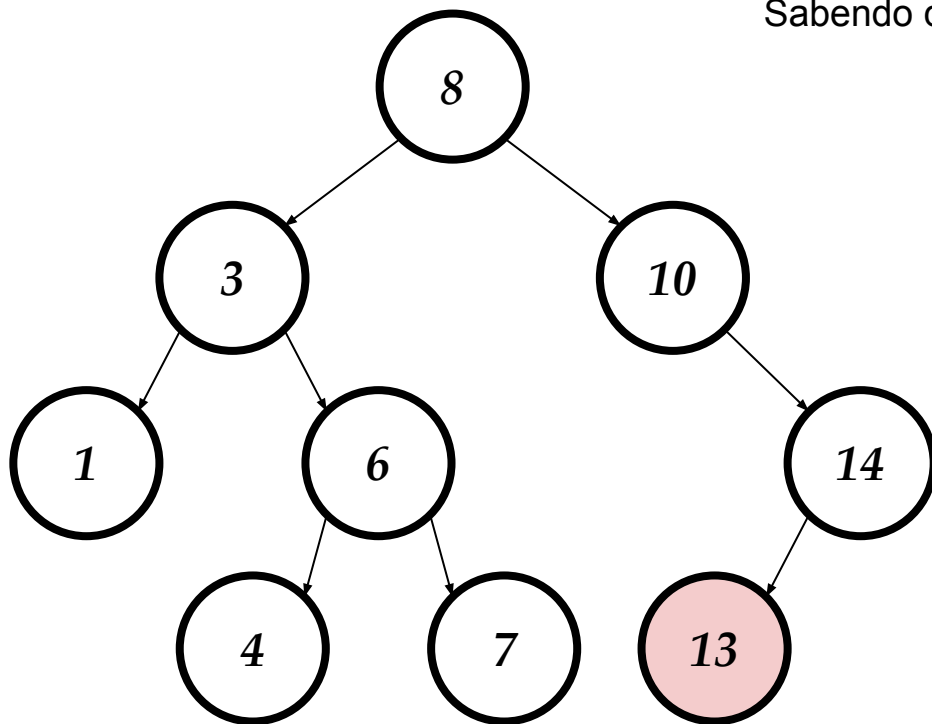
Removendo Folha



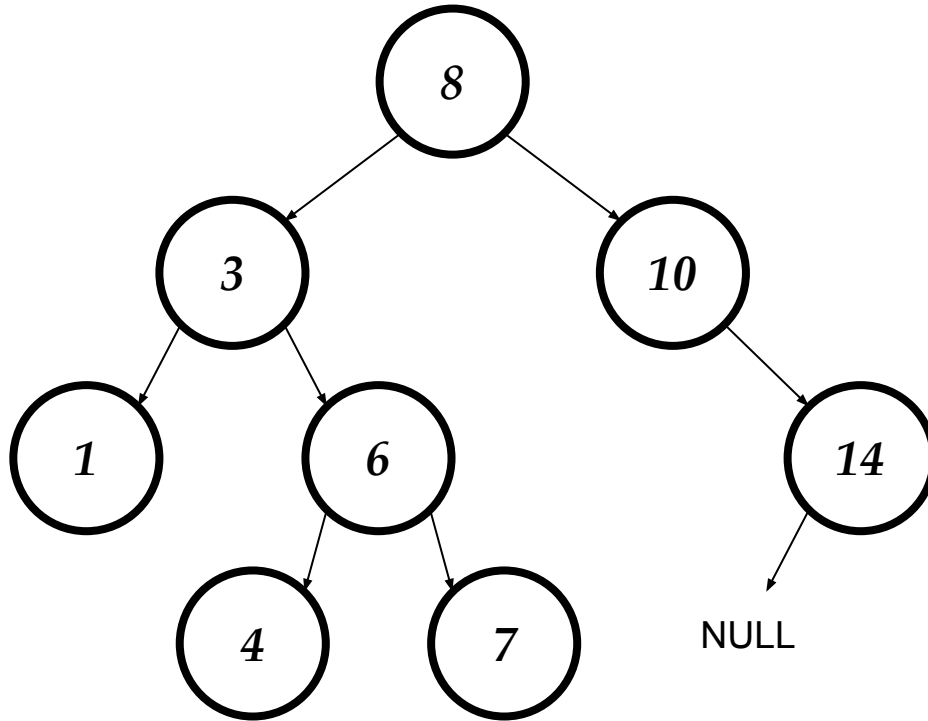
Removendo Folha

Free the node!

Sabendo quem é o pai, faz apontar para NULL



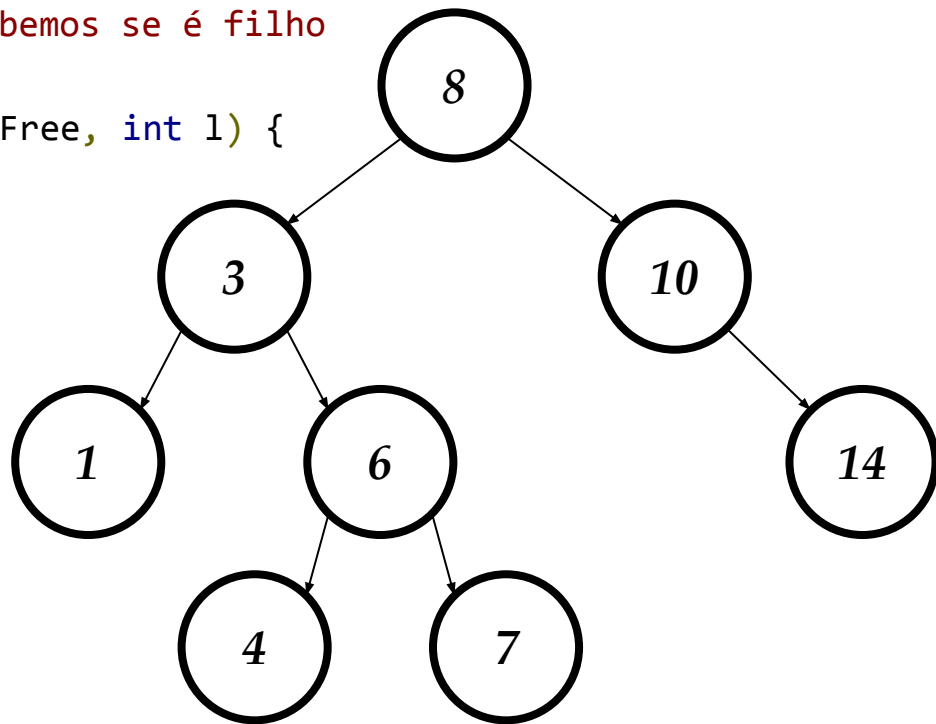
Removendo Folha



Removendo Folha

//Assumindo que já achamos o nó, o pai e sabemos se é filho
//da esquerda/direita (l).

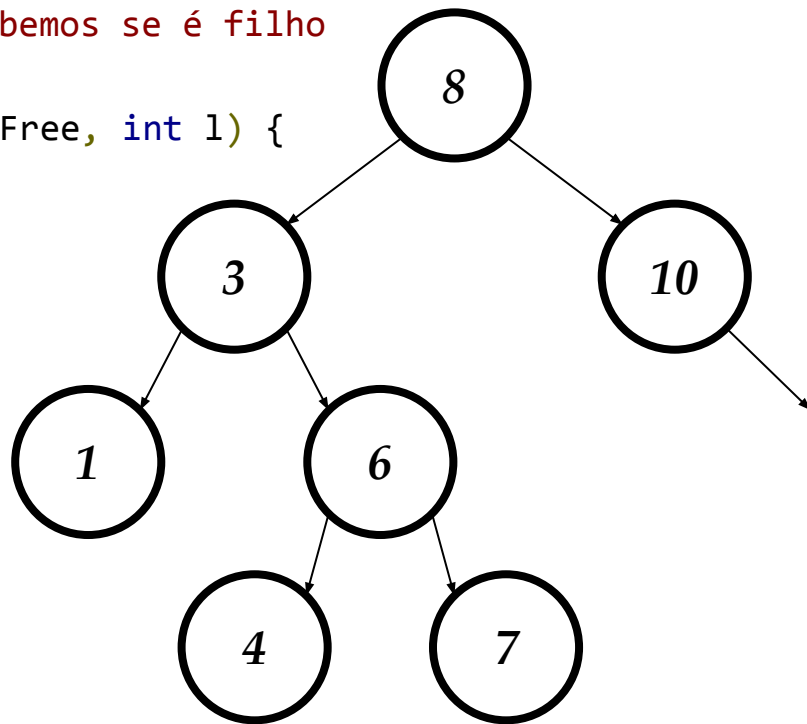
```
void removeCaso1(node_t *parent, node_t *toFree, int l) {  
    //Caso 1: Apenas Free!  
    if (toFree->leftChild == NULL \  
        && toFree->rightChild == NULL) {  
        free(toFree);  
        if (l == 1)  
            parent->leftChild = NULL;  
        else  
            parent->rightChild = NULL;  
    }  
}
```



Removendo Folha

//Assumindo que já achamos o nó, o pai e sabemos se é filho
//da esquerda/direita (l).

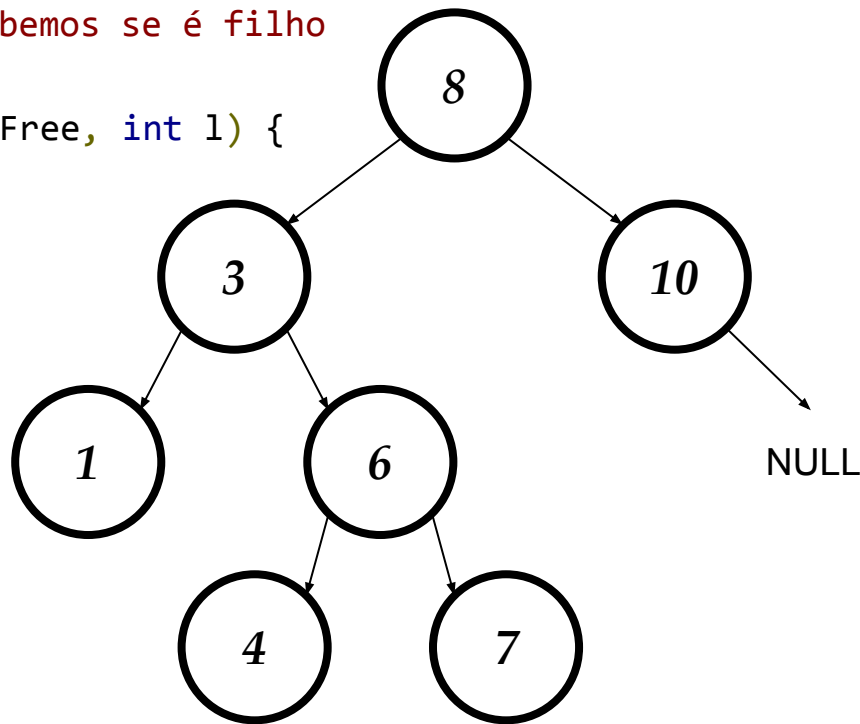
```
void removeCaso1(node_t *parent, node_t *toFree, int l) {  
    //Caso 1: Apenas Free!  
    if (toFree->leftChild == NULL \  
        && toFree->rightChild == NULL) {  
        free(toFree);  
        if (l == 1)  
            parent->leftChild = NULL;  
        else  
            parent->rightChild = NULL;  
    }  
}
```



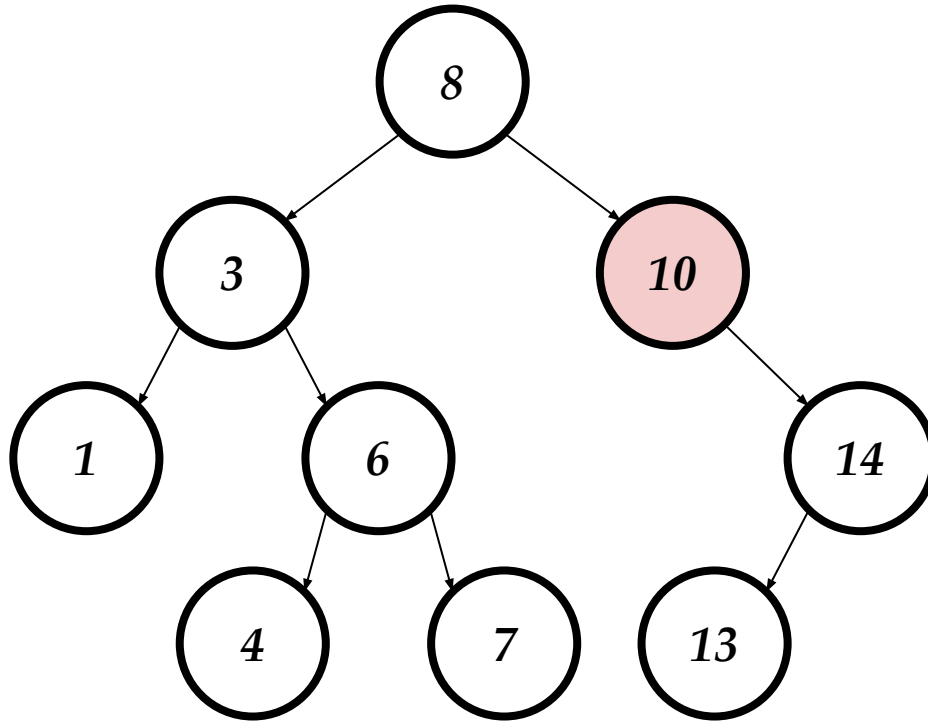
Removendo Folha

//Assumindo que já achamos o nó, o pai e sabemos se é filho
//da esquerda/direita (l).

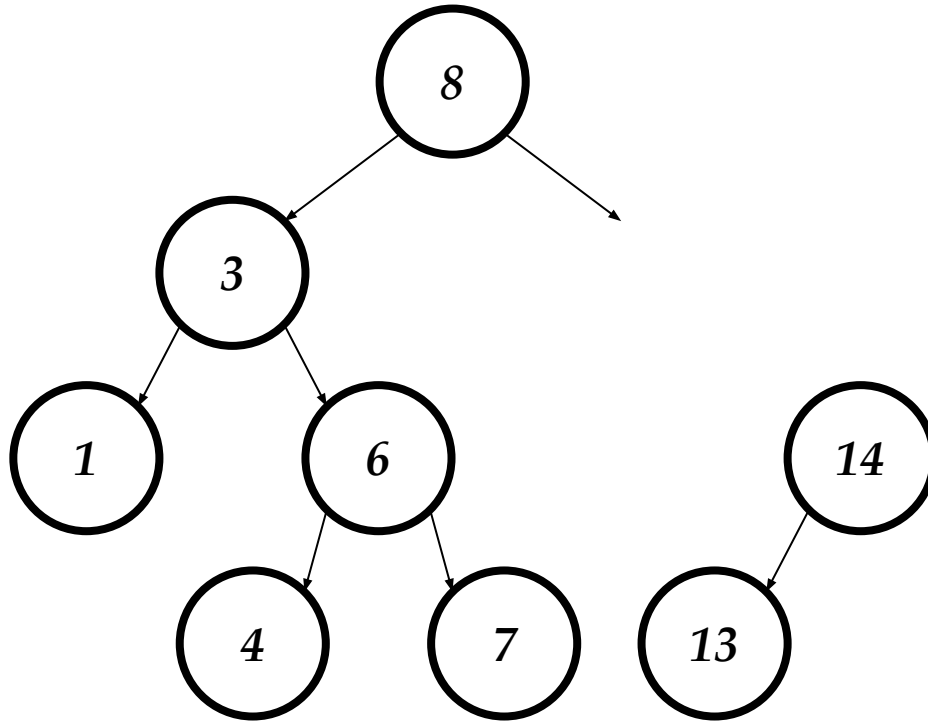
```
void removeCaso1(node_t *parent, node_t *toFree, int l) {  
    //Caso 1: Apenas Free!  
    if (toFree->leftChild == NULL \  
        && toFree->rightChild == NULL) {  
        free(toFree);  
        if (l == 1)  
            parent->leftChild = NULL;  
        else  
            parent->rightChild = NULL;  
    }  
}
```



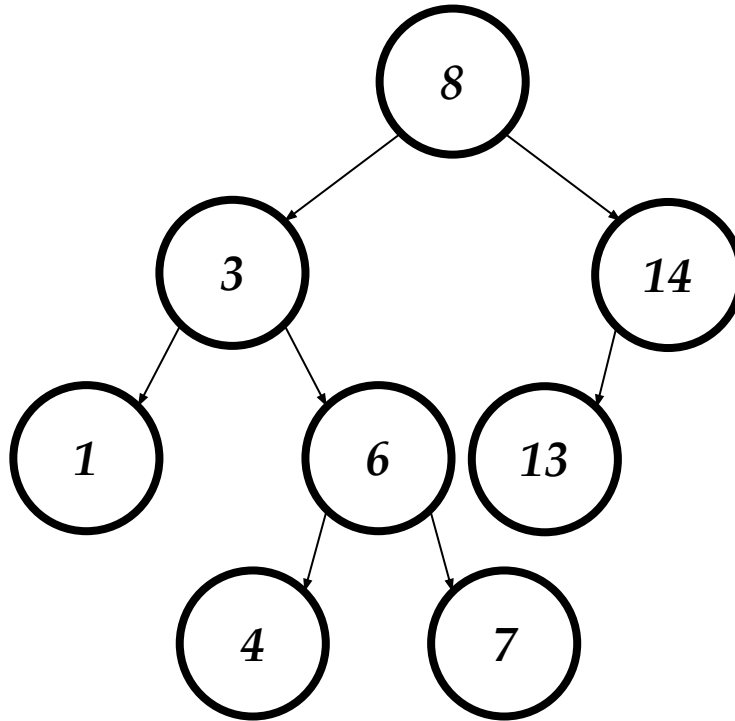
Removendo Com 1 Descendente



Removendo Com 1 Descendente

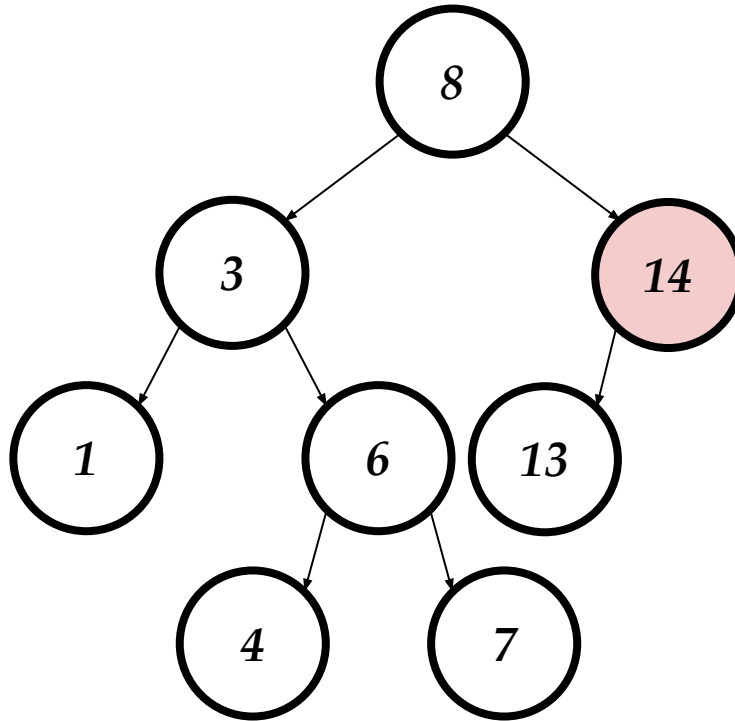


Removendo Com 1 Descendente

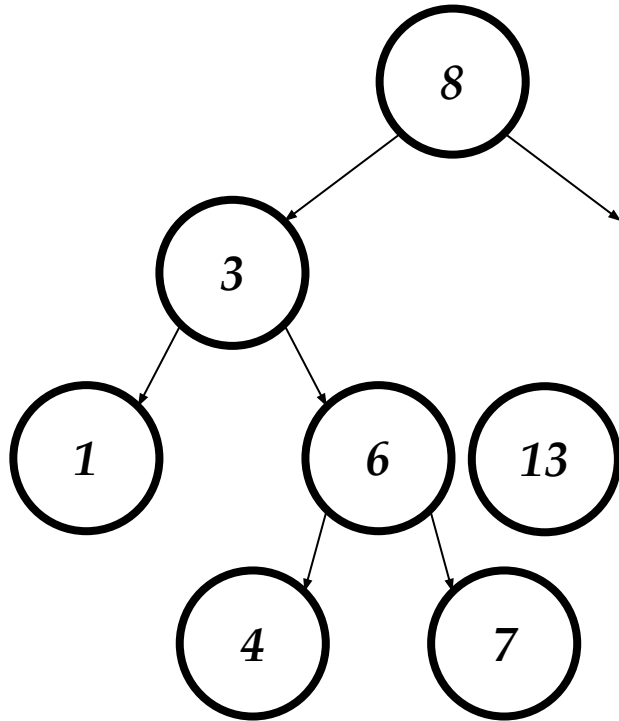


Free!
Herda

Removendo Com 1 Descendente

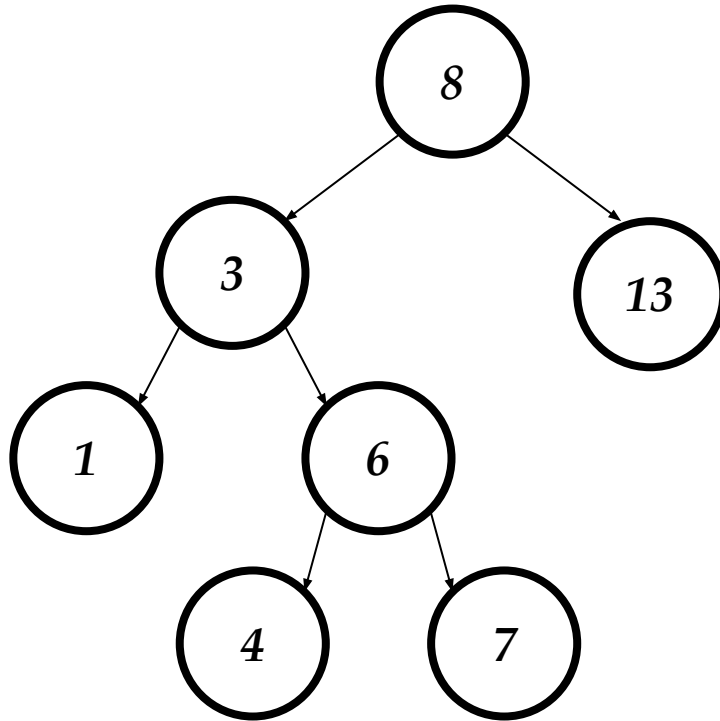


Removendo Com 1 Descendente



Free!
Herda

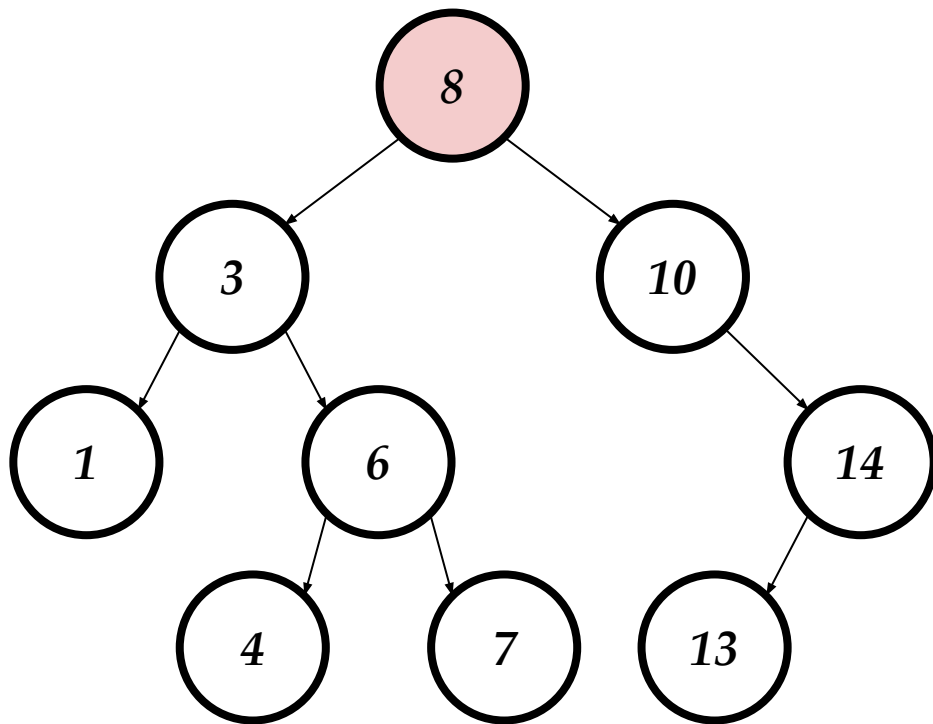
Removendo Com 1 Descendente



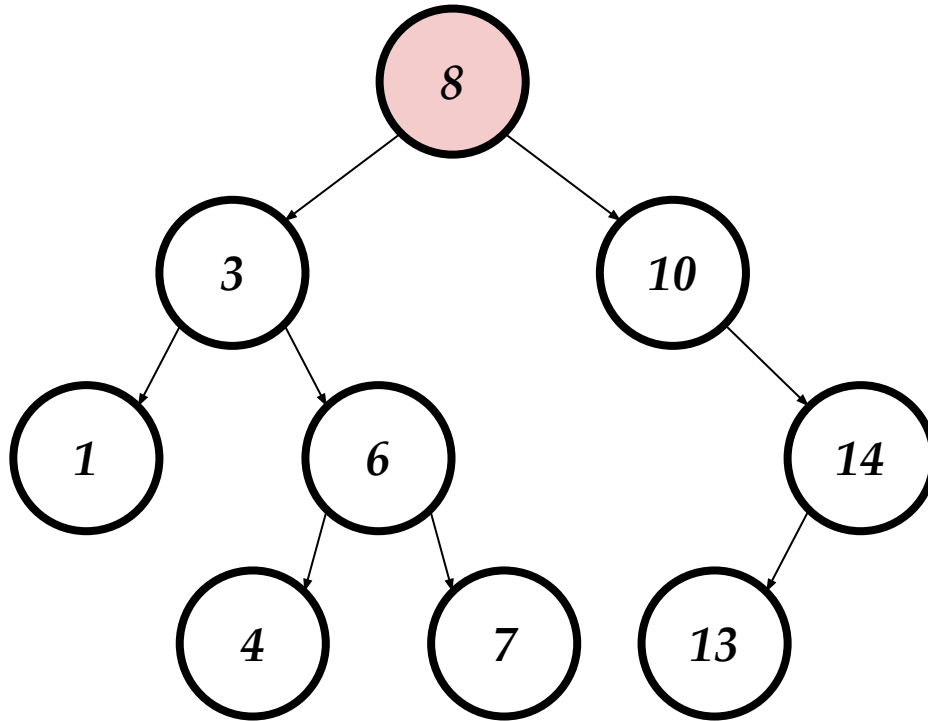
Free!
Herda

```
//Assumindo que já achamos o nó, o pai e sabemos se é filho
//da esquerda/direita (l).
void removeCaso2(node_t *parent, node_t *toFree, int l) {
    //Caso 2: Quando esquerda é NULL
    if (toFree->leftChild == NULL) {
        if (l == 1)
            parent->leftChild = toFree->rightChild; //Herda nó D
        else
            parent->rightChild = toFree->rightChild;
        free(toFree);
    }
    //Caso 2: Quando direita é NULL
    if (toFree->rightChild == NULL) {
        if (l == 1)
            parent->leftChild = toFree->leftChild; //Herda nó L
        else
            parent->rightChild = toFree->leftChild;
        free(toFree);
    }
}
```

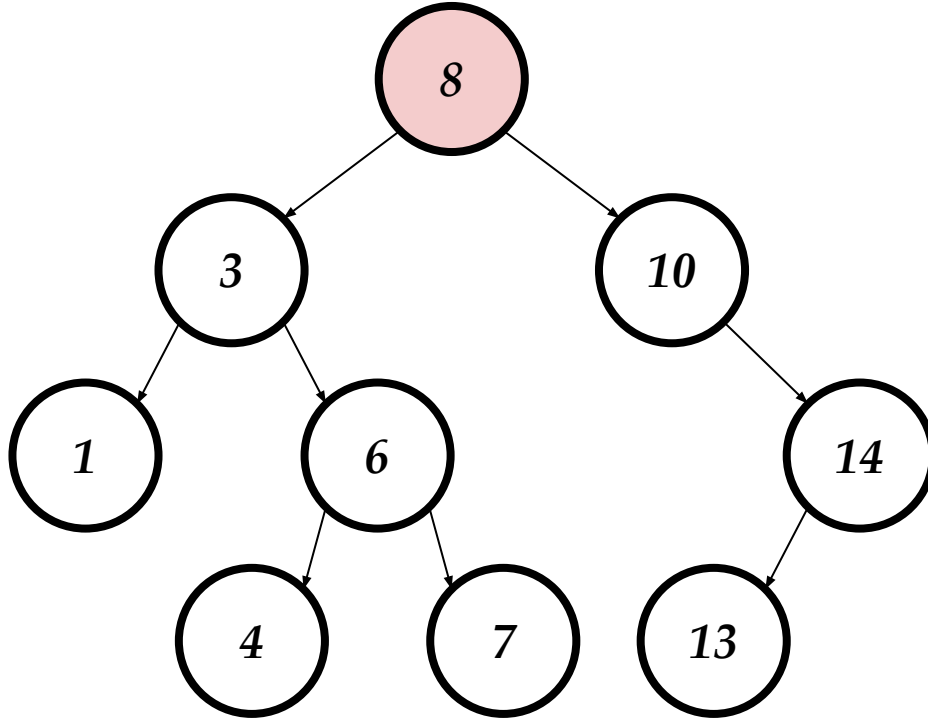
Terceiro Caso



Terceiro Caso: Qual valor pode substituir 8?

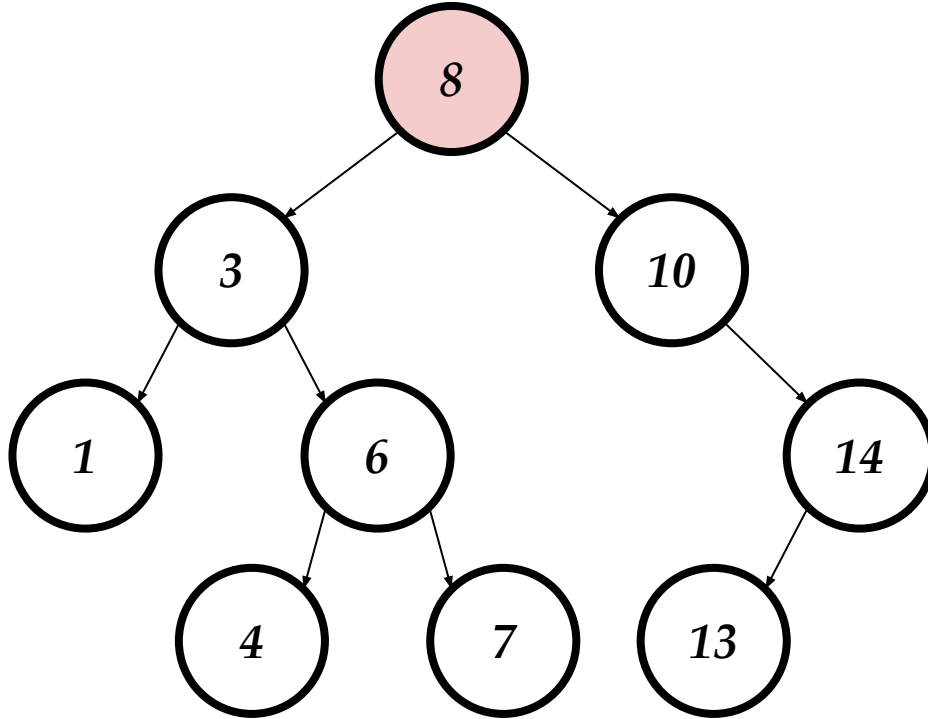


Terceiro Caso: Qual valor pode substituir 8?



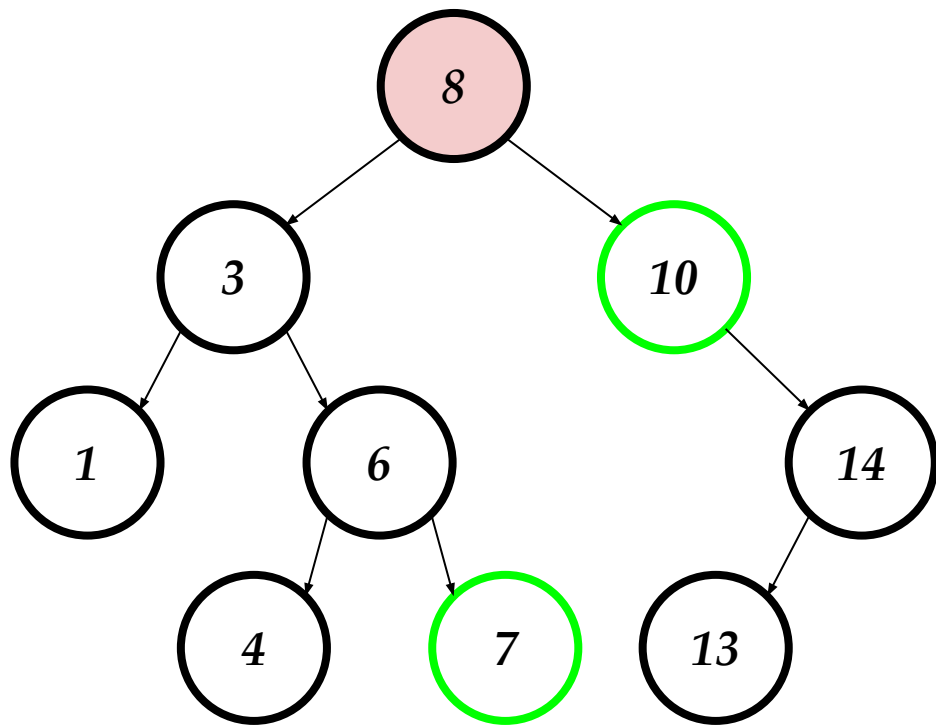
Olhe para os nós de forma ordenadas
1, 3, 4, 6, 7, 8, 10, 13, 14

Terceiro Caso: 7 ou 10



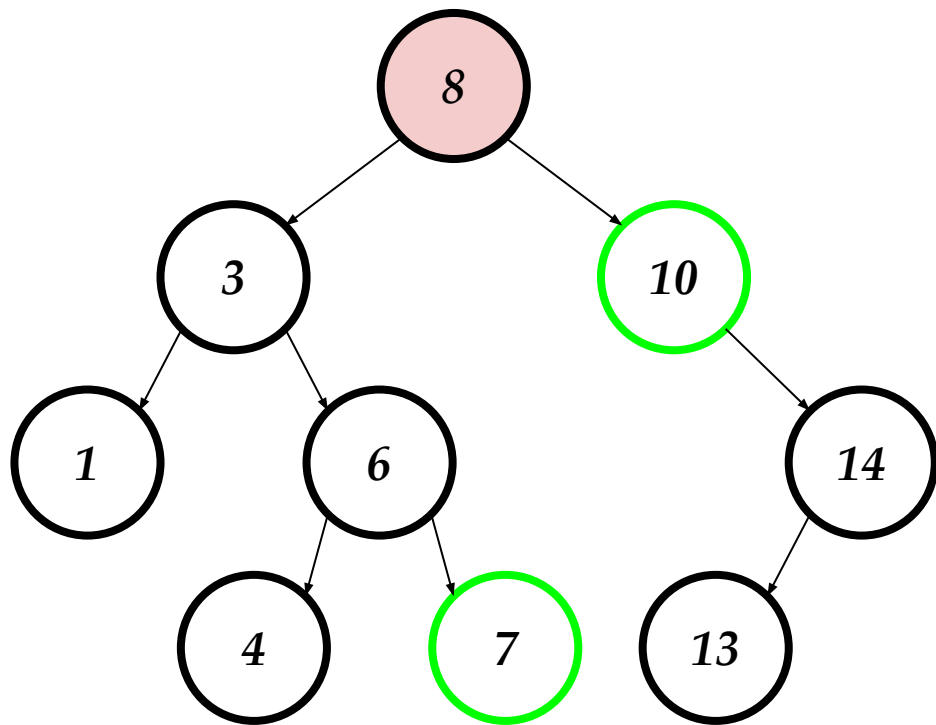
Olhe para os nós de forma ordenadas
1, 3, 4, 6, 7, 8, 10, 13, 14

Terceiro Caso: 7 ou 10



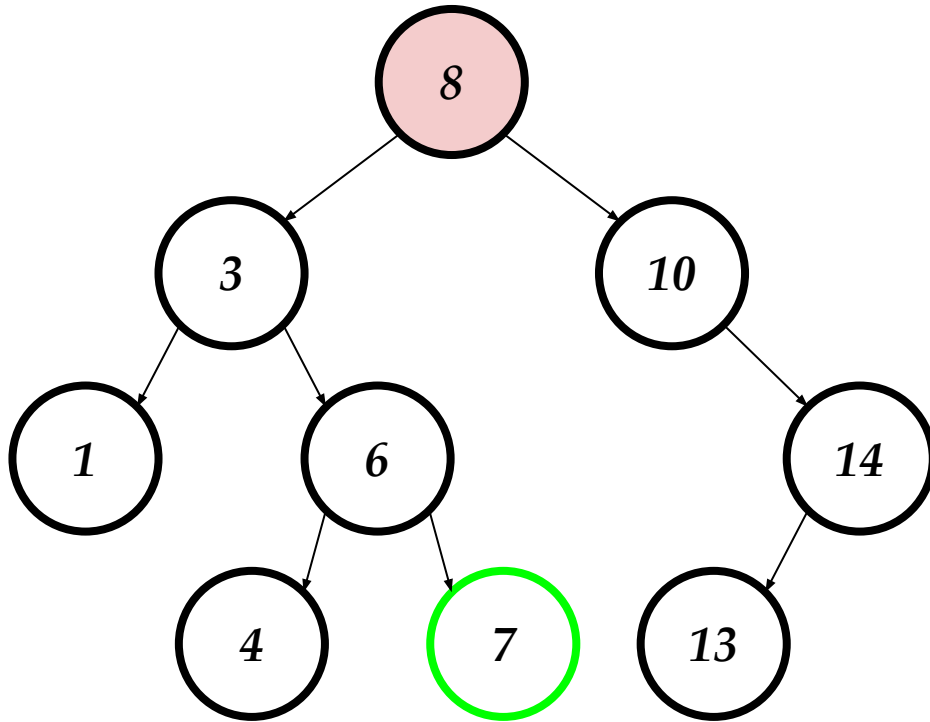
- Os elementos vizinhos quando pensamos em ordem
- Não quebram a invariante
 - 7 é maior do que 3 e menor do que 10
 - 10 é maior do que 3 e menor do que 14
- Como procedemos?

Terceiro Caso: 7 ou 10



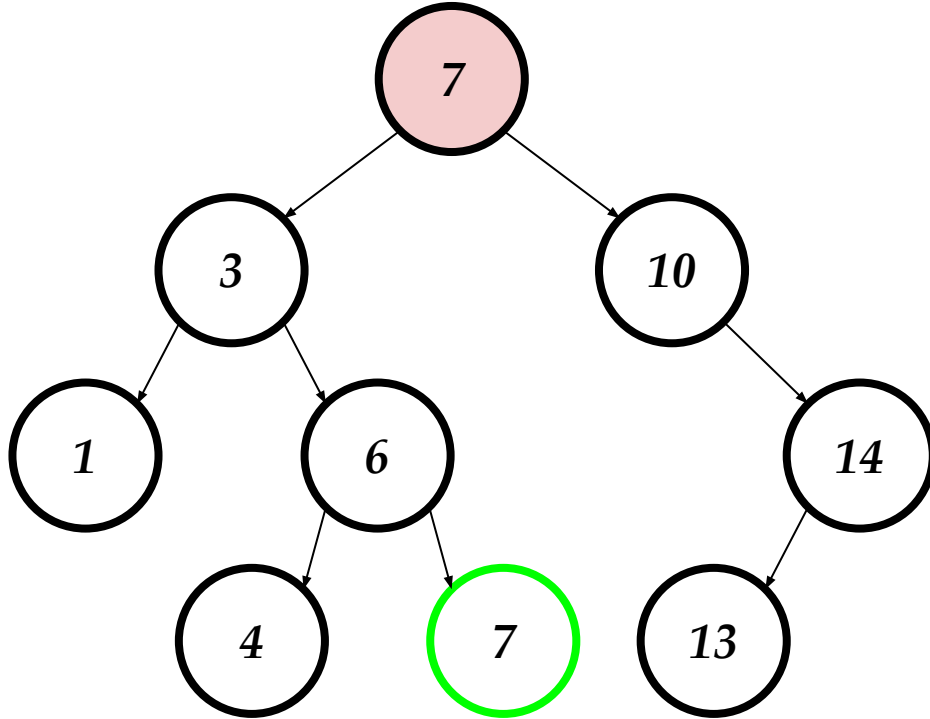
- Os elementos vizinhos quando pensamos em ordem
- Não quebram a invariante
 - 7 é maior do que 3 e menor do que 10
 - 10 é maior do que 3 e menor do que 14
- Como procedemos?
 - Trocar 8 por um deles

Terceiro Caso: Olhando para 7



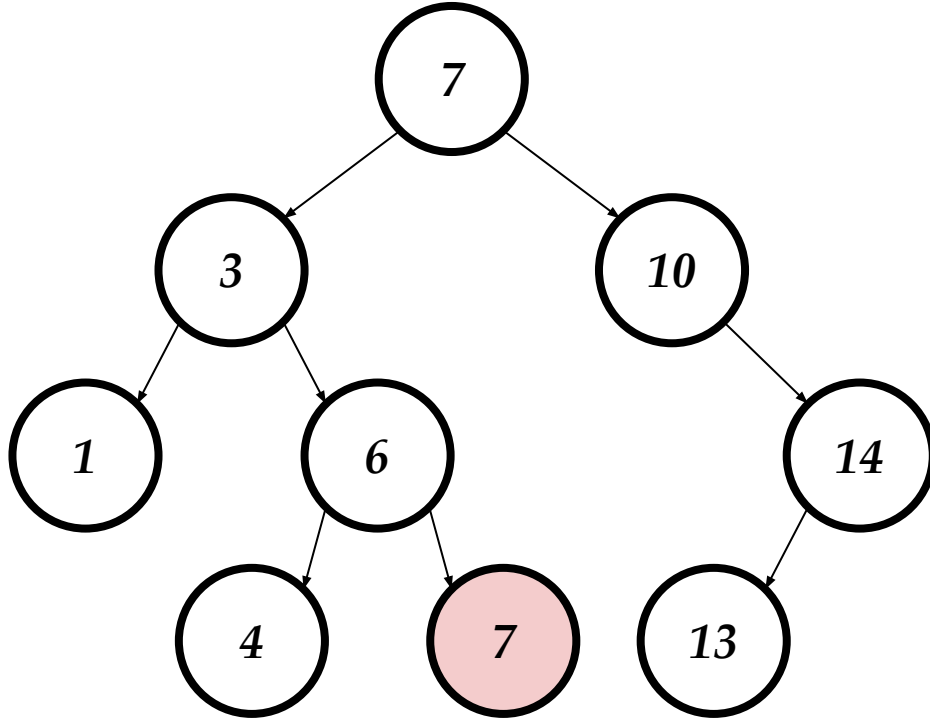
- Trocando 8 com 7
- Vamos trocar o **valor**!

Terceiro Caso: Olhando para 7



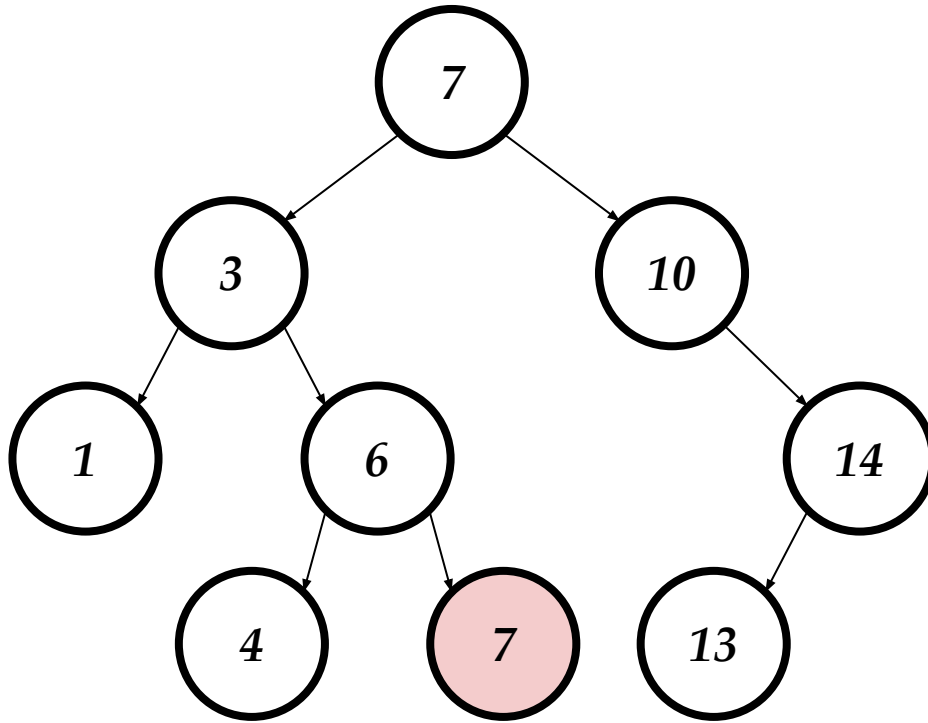
- Trocando 8 com 7
- Vamos trocar o **valor!**

Terceiro Caso: Olhando para 7



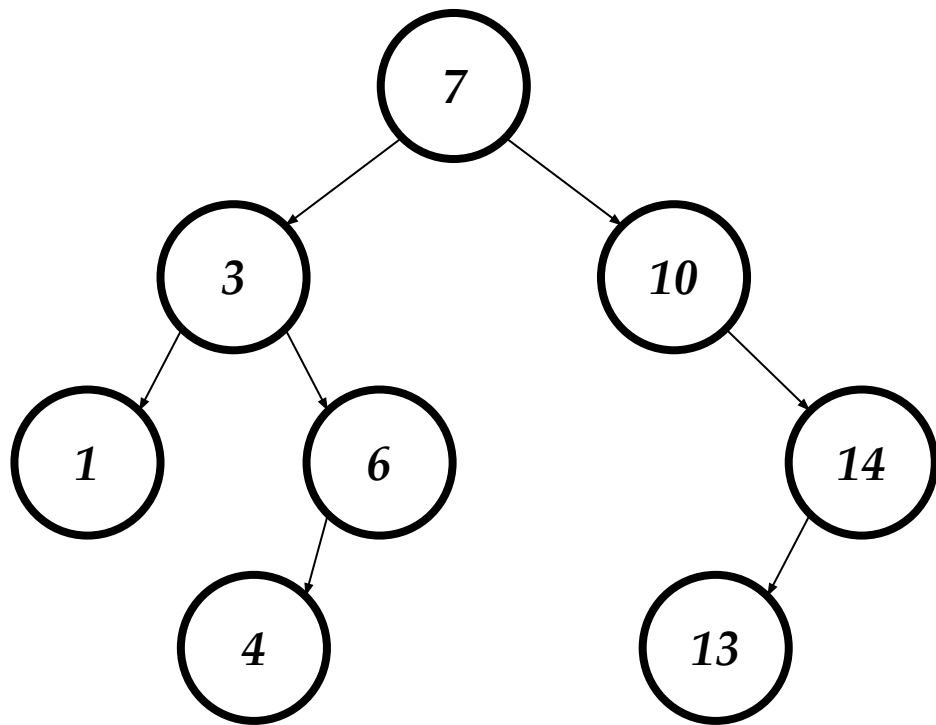
- Trocando 8 com 7
- Vamos trocar o **valor**!
- Remover o antigo 7

Terceiro Caso: Olhando para 7



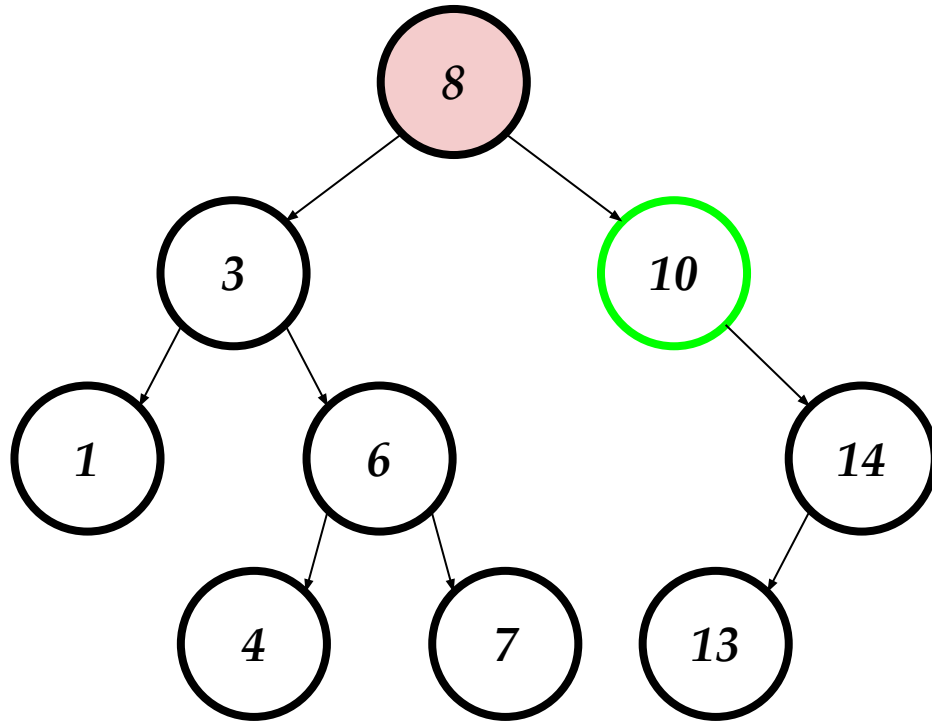
- Trocando 8 com 7
- Vamos trocar o **valor**!
- Remover o antigo 7
 - Caso 1 sem descendentes

Terceiro Caso: Olhando para 7



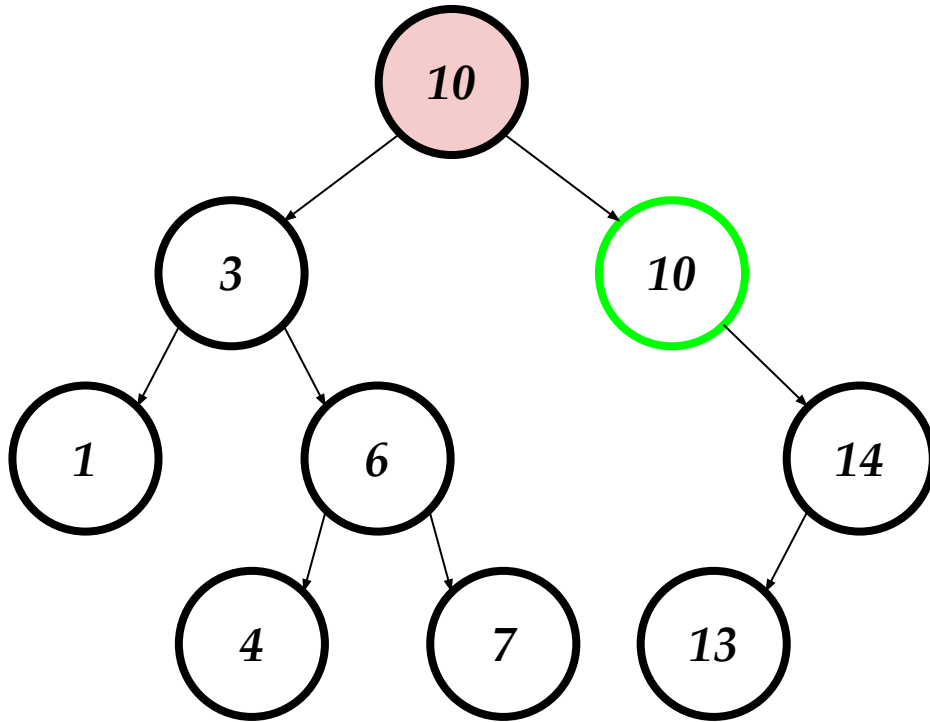
- Trocando 8 com 7
- Vamos trocar o **valor**!
- Remover o antigo 7
 - Caso 1 sem descendentes
- Fazemos uso da função de remove

Terceiro Caso: Olhando para 10



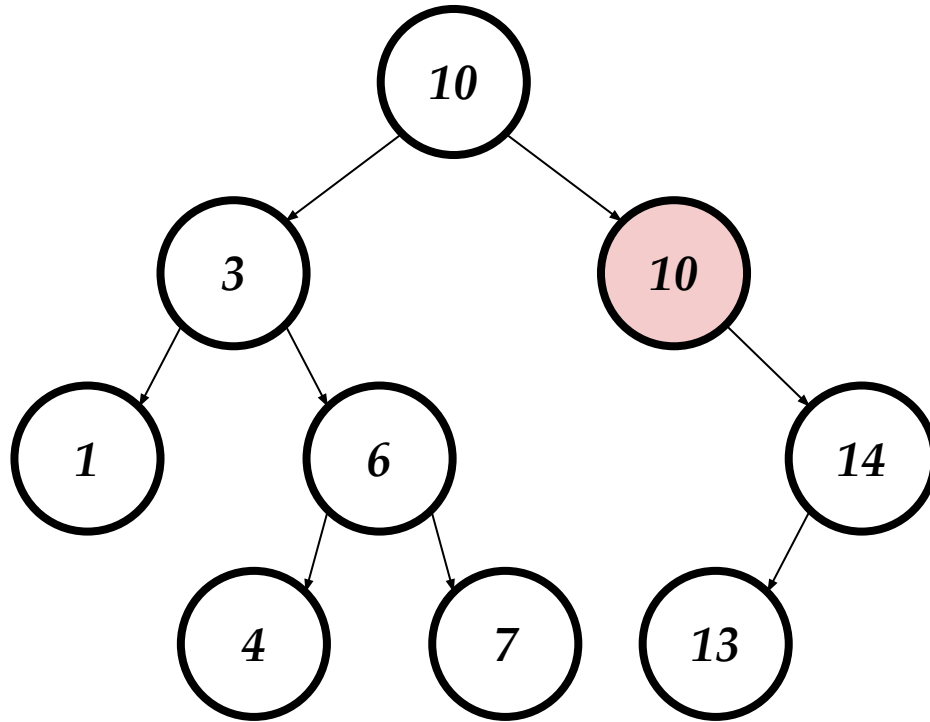
- Trocando 8 com 7
- Vamos trocar o **valor**!

Terceiro Caso: Olhando para 10



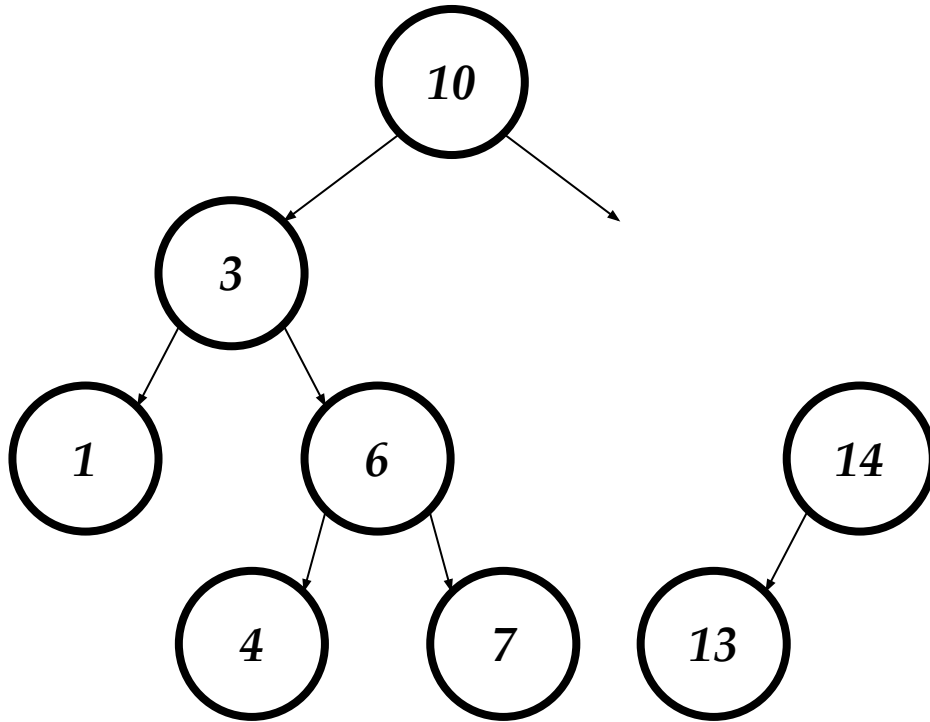
- Trocando 8 com 7
- Vamos trocar o **valor**!
- Remover 10

Terceiro Caso: Olhando para 10



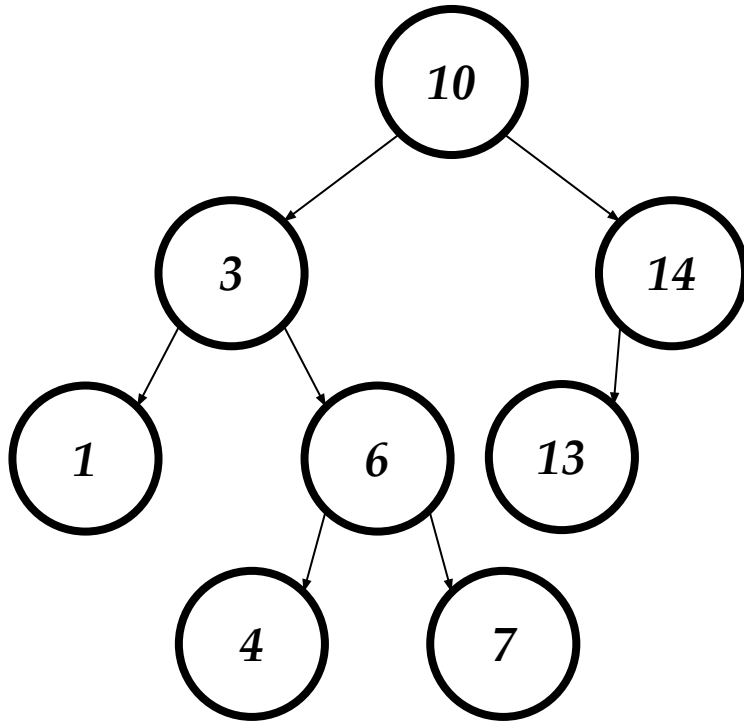
- Trocando 8 com 7
- Vamos trocar o **valor**!
- Remover 10
 - Caso 2
 - 1 descendente

Terceiro Caso: Olhando para 10



- Trocando 8 com 7
- Vamos trocar o **valor**!
- Remover 10
 - Caso 2
 - 1 descendente

Terceiro Caso: Olhando para 10



- Trocando 8 com 7
- Vamos trocar o **valor**!
- Remover 10
 - Caso 2
 - 1 descendente

Como achamos o 7 ou 10?

- Menor elemento da árvore
 - Sempre caminhar para esquerda
- Maior elemento da árvore
 - Sempre caminhar para direita
- 7 é o maior elemento entre os menores do que 8
 - 1 passo para direita
 - Acha menor indo para esquerda
- 10 é o menor elemento entre os maiores do que 8
 - 1 passo para esquerda
 - Acha menor indo para direita

Substituímos e fazemos uso do Caso 1 ou 2

```
void removeCaso3(node_t *toFree) {  
    node_t *replacement = findMin(toFree->rightChild);  
    toFree->value = replacement->value;  
    if (toFree->leftChild == NULL && toFree->rightChild == NULL)  
        removeCaso1(replacement);  
    else  
        removeCaso2(replacement);  
}
```


Inserção e Remoção

- Pode ser feito recursivamente ou iterativamente
- Vários pequenos detalhes
 - Cuidar dos NULLs
 - Frees
 - Herança
- Slides mostram a ideia
- Olhar o código exemplo
 - Veja como tem vários ifs, cuidar de tudo
 - Foque no essencial!

Liberando a Árvore Toda

- Podemos reutilizar todo aquele remove que fizemos

- Free no bst_tree

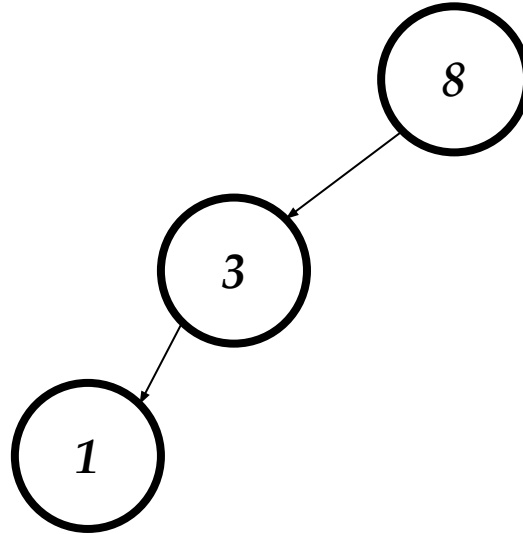
- Simples!

```
void bstFree(bst_t *tree) {  
    while(tree->root != NULL)  
        //Caso 1, 2 e 3 dentro de 1 remove  
        removeValue(tree, tree->root->value);  
    free(tree);  
}
```

- Podemos usar caminhamentos também
 - Slides futuros

Pior Caso (inserimos na ordem)

Vira uma lista



Custos

- Inserir?
- Remover?
- Achar valor?

Custos: Pior Caso

- Inserir?
 - $O(n)$
- Remover?
 - $O(n)$
- Achar valor?
 - $O(n)$

Custos: Assumindo que a árvore é balanceada

Elementos inseridos de forma uniforme causam balanceamento na média

- Inserir?
 - $O(\log n)$
- Remover?
 - $O(\log n)$
- Achar valor?
 - $O(\log n)$
- Melhor do que a lista para achar valores
- Podemos garantir que a árvore sempre vai ser balanceada
 - Aulas futuras

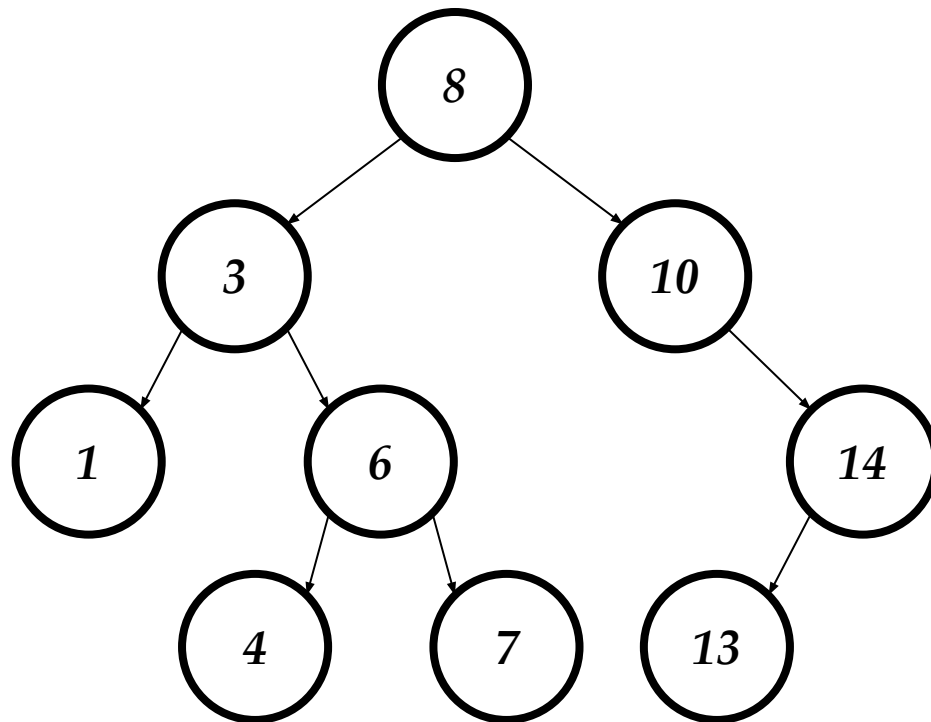
Caminhamento: Como imprimir os nós?

- Diferentes formas de visitar todos os nós

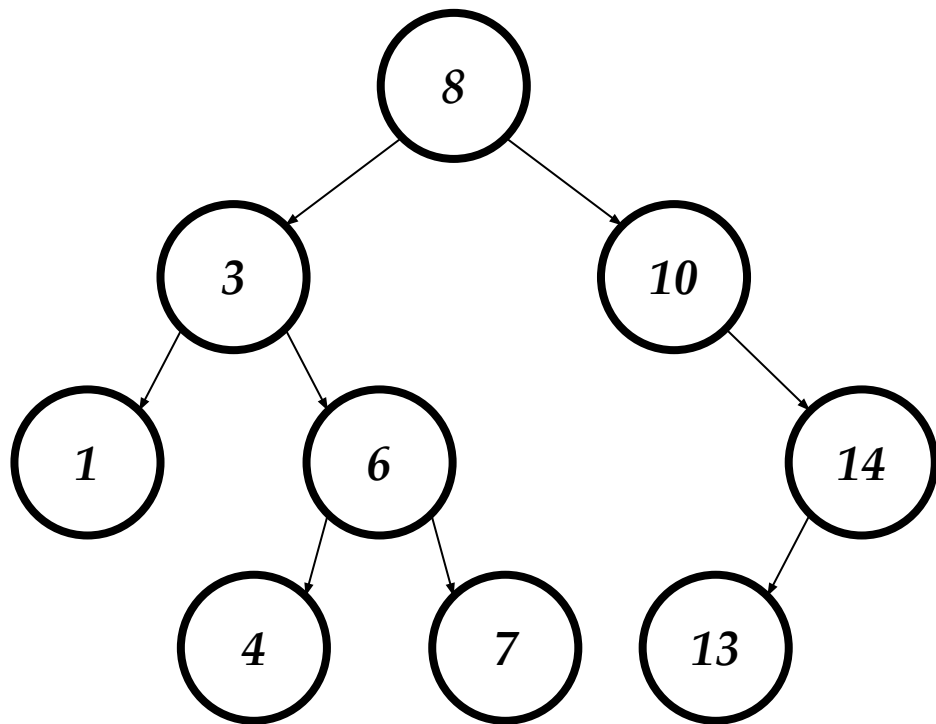
Caminhamento: Como imprimir os nós?

- Diferentes formas de visitar todos os nós
- Central (em ordem, in order)
 - Visitar esquerda
 - Imprimir nó
 - Visitar direita
- Pré ordem (pre order)
 - Imprimir nó
 - Visitar esquerda
 - Visitar direita
- Pós ordem (pre order)
 - Visitar esquerda
 - Visitar direita
 - Imprimir

Em Ordem



Em Ordem: 1, 3, 4, 6, 7, 8, 10, 13, 14

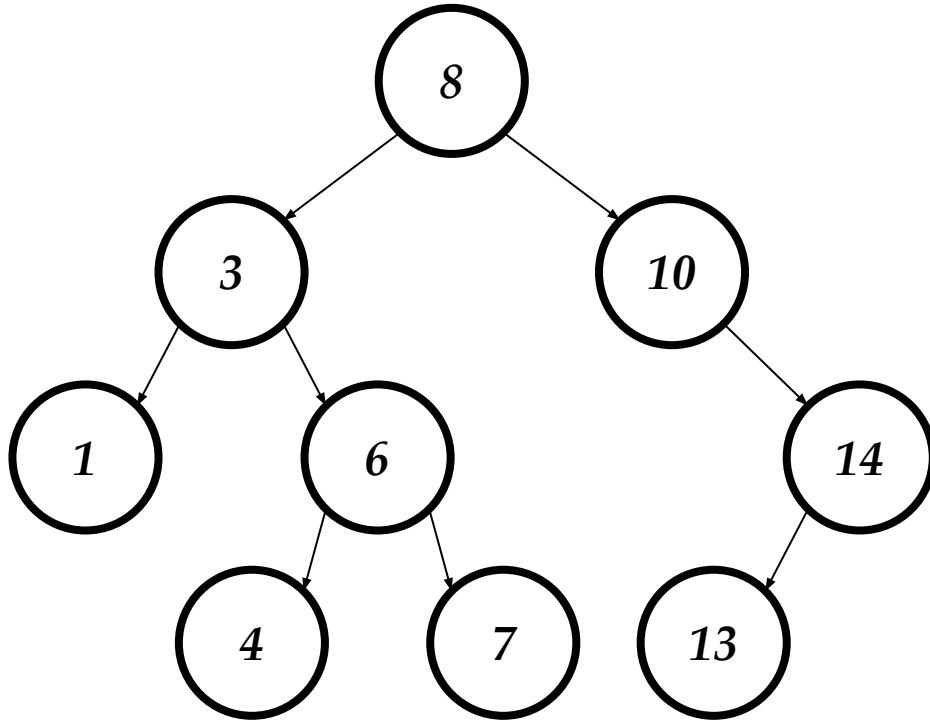


```
void printInOrder(node_t *node) {  
    if (node == NULL) {  
        return;  
    }  
    printInOrder(node->leftChild);  
    printf("%d ", node->value);  
    printInOrder(node->rightChild);  
}
```

Em Ordem

- Essencialmente imprime os nós ordenados
- Como caminhamos para a esquerda até o fim
 - Imprime menor
 - Imprime seguinte
 -

Pre Ordem: 8, 3, 1, 6, 4, 7, 10, 14, 13



```
void printPreOrder(node_t *node) {  
    if (node == NULL) {  
        return;  
    }  
    printf("%d ", node->value);  
    printInOrder(node->leftChild);  
    printInOrder(node->rightChild);  
}
```

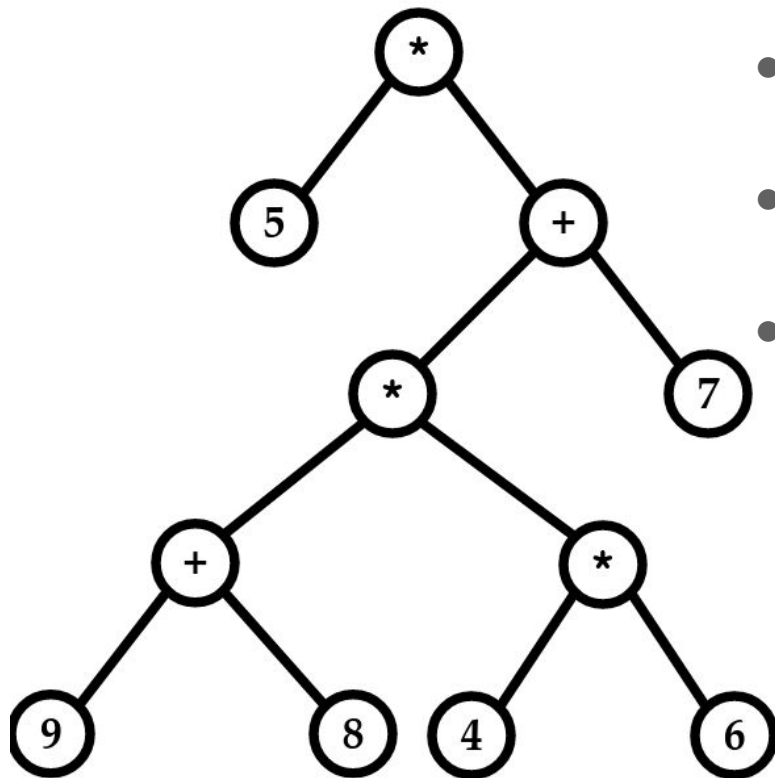
Caminhamento Pré Ordem

- Pode ser utilizado para duplicar uma árvore
- Como fazer?
 - Cria uma nova árvore
 - Caminha pré ordem
 - Insere os nós (ao invés de print) na novas
- Também pode usado para expressões resolver expressões

```
void duplicate(bst_t *tree, node_t *node) {  
    if (node == NULL) {  
        return;  
    }  
    insertValue(tree, node->value);  
    duplicate(node->leftChild);  
    duplicate(node->rightChild);  
}
```

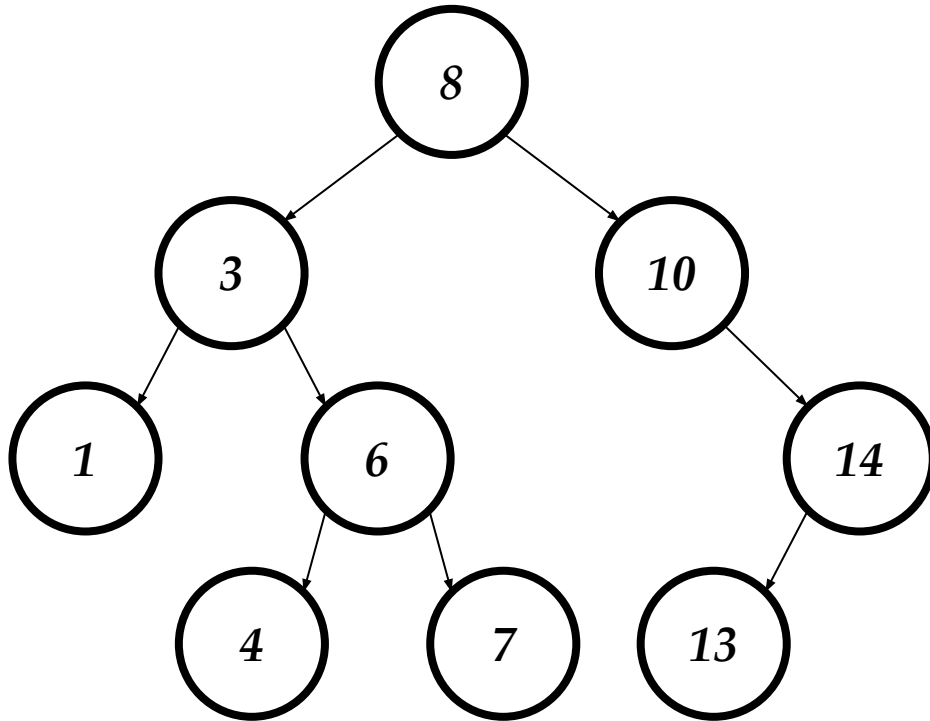
Resolvendo Expressões Pré Ordem

$5 * ((9+8)*4*6 + 7)$



- Leia o valor do nó
- Coloque em uma pilha
- Utilize o algoritmo de notação polonesa na pilha

Pos Ordem: 1, 4, 7, 6, 3, 13, 14, 10, 8



```
void printPosOrder(node_t *node) {  
    if (node == NULL) {  
        return;  
    }  
    printInOrder(node->leftChild);  
    printInOrder(node->rightChild);  
    printf("%d ", node->value);  
}
```

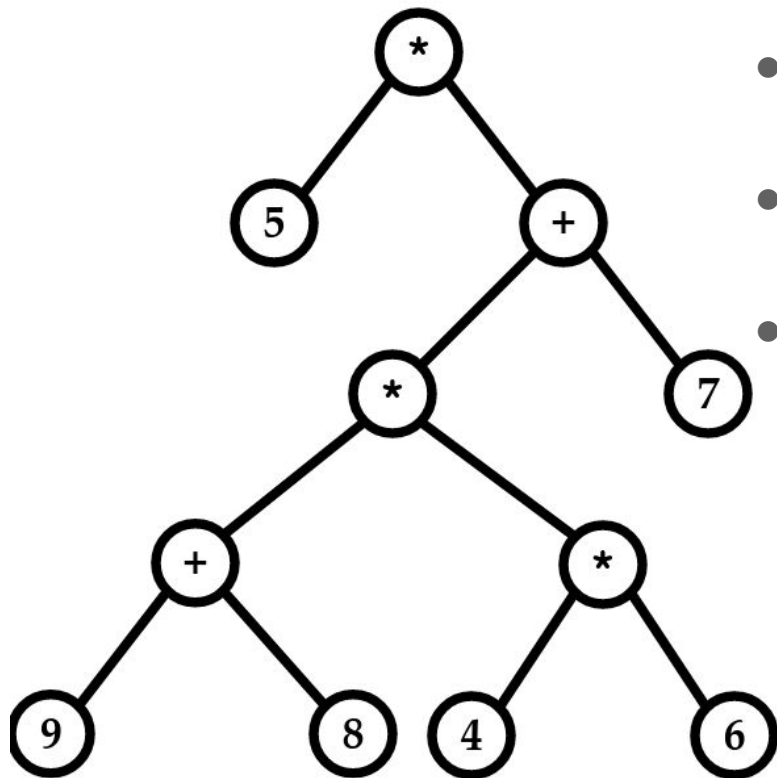
Caminhamento Pós Ordem

- Pode ser utilizado para liberar a árvore
- Sempre estamos com um nó sem descendentes
- Melhor do que o código anterior

```
void bstFree(node_t *node) {  
    if (node == NULL) {  
        return;  
    }  
    bstFree(node->leftChild);  
    bstFree(node->rightChild);  
    free(node);  
}
```


Resolvendo Expressões Pós Ordem

$5 * ((9+8)*4*6 + 7)$



- Leia o valor do nó
- Coloque em uma pilha
- Utilize o algoritmo de notação polonesa **reversa** na pilha

Caminhamento

- Qual o custo?

Caminhamento

- Qual o custo?
 - $O(n)$
 - Visitamos todos os nós

Caminhamento

- Qual o custo?
 - $O(n)$
 - Visitamos todos os nós
 - Podemos aplicar o teorema mestre também

Teorema Mestre

- $T(n) = a * T(n/b) + f(n)$
- Se $f(n) = O(n^{\log_b a - \varepsilon})$ para uma constante $\varepsilon > 0$
 - temos que $T(n) = \Theta(n^{\log_b a})$
- Se $f(n) = \Theta(n^{\log_b a})$, ou seja, $\varepsilon = 0$
 - temos que $T(n) = \Theta(n^{\log_b a} \log n)$
- Se $f(n) = \Omega(n^{\log_b a - \varepsilon})$, com $\varepsilon > 0$ e se $a * f(n/b) \leq c * f(n)$, com $c < 1$
 - temos que $T(n) = \Theta(f(n))$

Teorema Mestre

- $T(n) = 2 * T(n/2) + O(1)$
 $c = 0, \varepsilon = 1$
- *Se $f(n) = O(n^{\log_b a - \varepsilon})$ para uma constante $\varepsilon > 0$*
 - *temos que $T(n) = \Theta(n^{\log_b a})$*
- *Se $f(n) = \Theta(n^{\log_b a})$, ou seja, $\varepsilon = 0$*
 - *temos que $T(n) = \Theta(n^{\log_b a} \log n)$*
- *Se $f(n) = \Omega(n^{\log_b a - \varepsilon})$, com $\varepsilon > 0$ e se $a * f(n/b) \leq c * f(n/b)$, com $c < 1$*
 - *temos que $T(n) = \Theta(f(n))$*

Notas Finais

- Código aqui são pequenos trechos de um código completo de árvore
- Tentem implementar do 0