

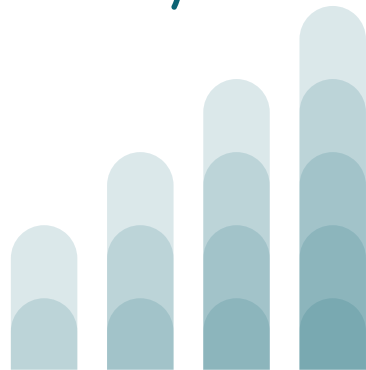
INSTITUTO FEDERAL DE  
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA  
SÃO PAULO  
Campus Campos do Jordão

# ESTRUTURA DE DADOS

## Árvore Binária de Busca

Conceitos, Inserção, Busca, Percorrer (Pesquisa Binária) e Remoção.

Professor Mestre Igor de Moraes Sampaio  
[igor.sampaio@ifsp.edu.br](mailto:igor.sampaio@ifsp.edu.br)





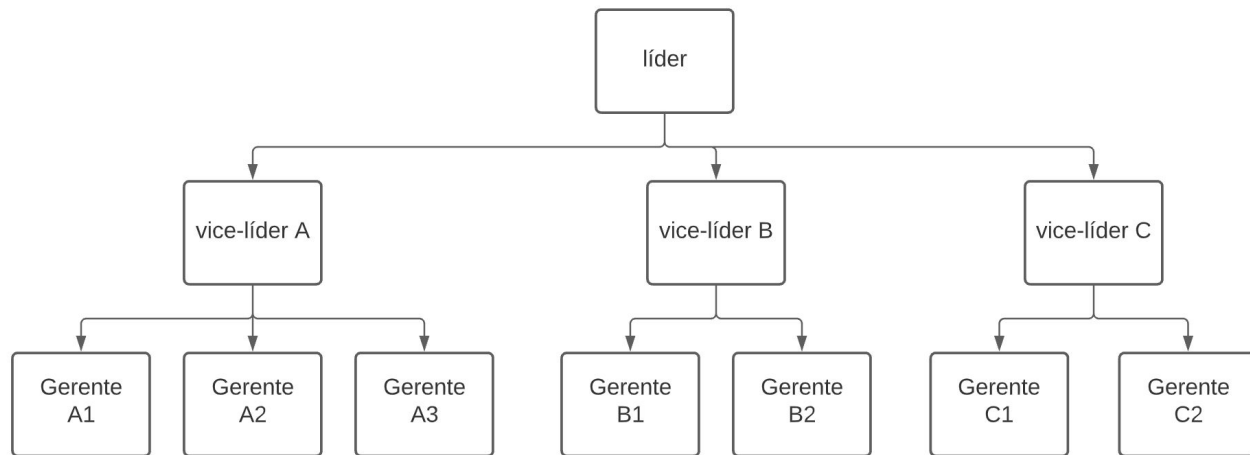
# Árvore





# O que é uma Árvore?

Uma árvore é uma estrutura de dados hierárquica que consiste em nós conectados por arestas, formando uma estrutura em forma de ramificações. É muito usada em computação para organizar dados que seguem relações de hierarquia ou para buscas e ordenações eficientes.



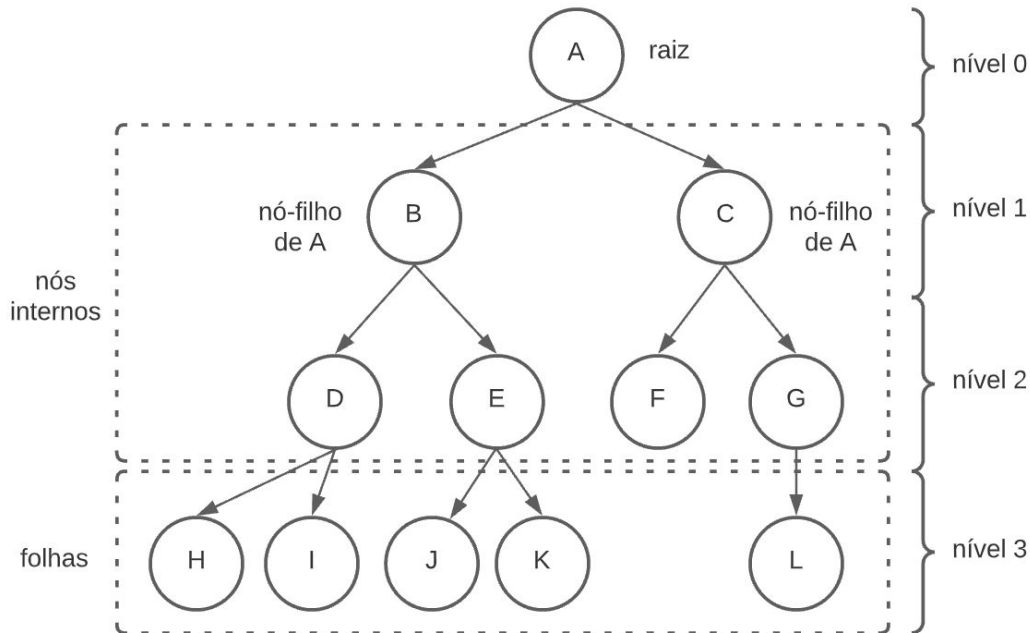



# Árvore Binária



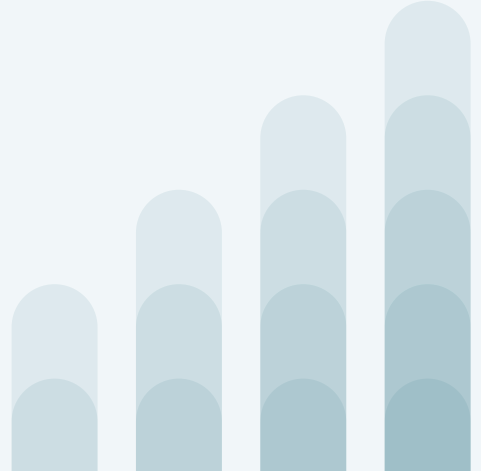
# Árvore Binária

- Cada nó pode ter no máximo dois filhos: um à esquerda e um à direita. É muito usada em algoritmos de busca e ordenação.





# Árvore Binária de Busca

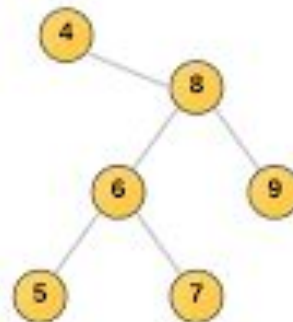
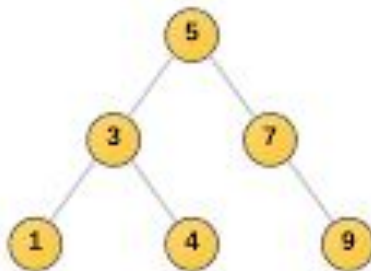
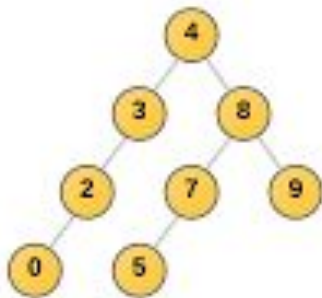




# Árvore Binária de Busca (Binary Search Tree – BST)

- Nós à esquerda < nó atual
- Nós à direita > nó atual

Permite buscas, inserções e remoções eficientes (tempo médio  $O(\log n)$ ).





# Representação do Nó da Árvore





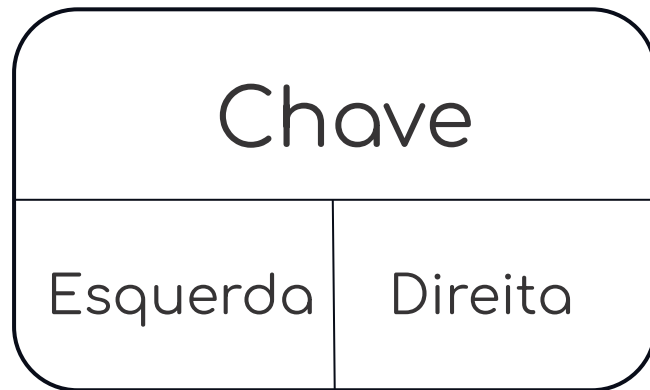


# Representação do Nó da Árvore

---

Para representar um nó em uma árvore binária de busca precisamos armazenar o valor do nó quem é seu vizinho a direita e quem é seu vizinho a esquerda. Com isso temos:

- Valor
- O nó à esquerda
- O nó à direita



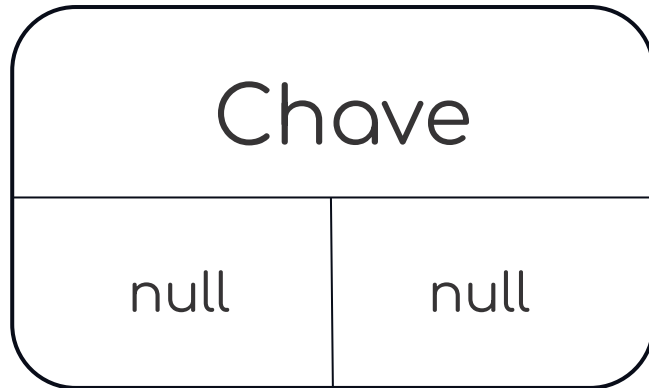


# Representação do Nó da Árvore

---

Para representar indicar os nós à esquerda e a direita usamos dois ponteiros: um para representar a subárvore da esquerda e um para representar a subárvore da direita. Além de um campo para a chave e dados.

Se não tivermos uma subárvore, usamos o valor nulo para o ponteiro.





# Representação do Nó da Árvore

---

- Estrutura mínima para um nó de uma árvore binária em c++.

```
// Estrutura de um nó da árvore binária
struct No
{
    int valor;
    No* dir;
    No* esq;
};
```



# Inicialização da Árvore





# Inicialização da Árvore

---

Para representarmos uma árvore, precisaremos apenas do endereço do nó raiz, pois a partir da raiz temos acesso a todos os nós da árvore. E para inicializarmos a árvore basta tornarmos esse endereço NULL.

```
No* inicializar()
{
    return nullptr; // Inicializa o nó da árvore binária como nulo
}

int main() {
    No* no = inicializar(); // Inicializa o nó da árvore binária
}
```



Inserir um  
Elemento





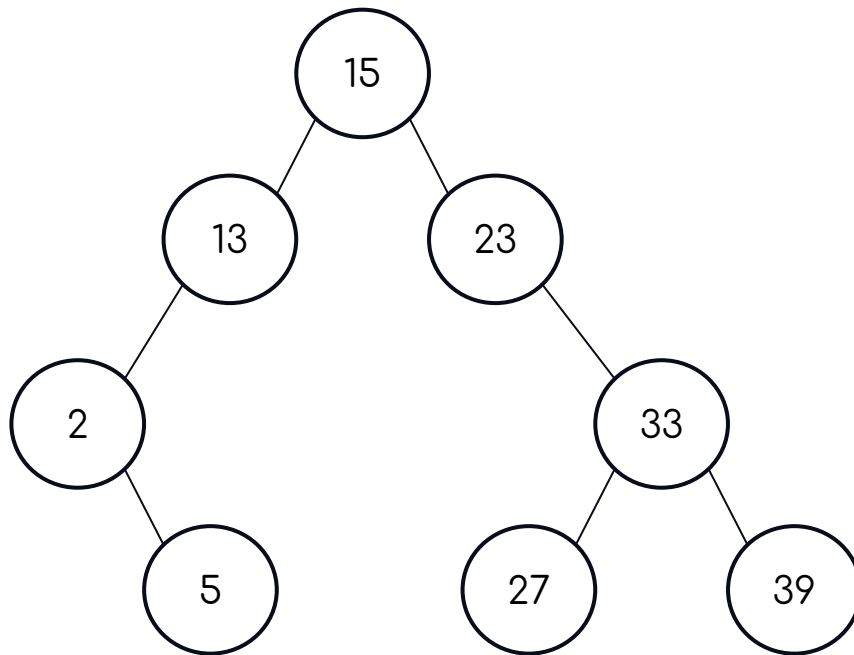
# Inserção - Chaves Duplicadas

---

- Por padrão árvores binária não permitem chaves duplicadas, porém caso queira permitir, basta definir uma política.
  - Exemplo: chaves  $\leq$  à de um determinado nó ficam na subárvore à esquerda deste.
  - Em nosso caso não vamos permitir duplicação de chaves.

# Inserção

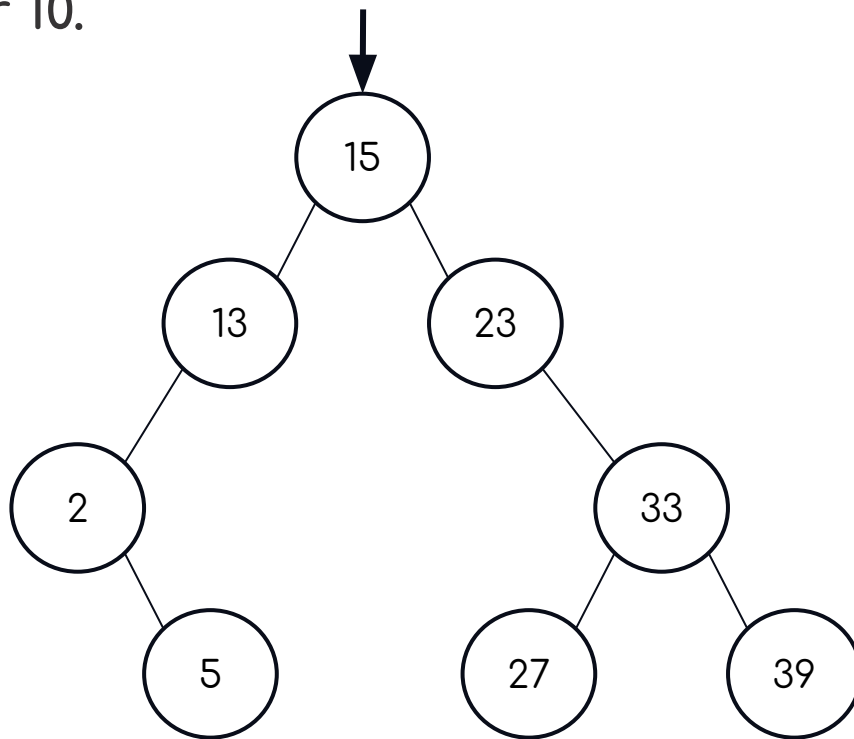
- Exemplo: Inserir 10.





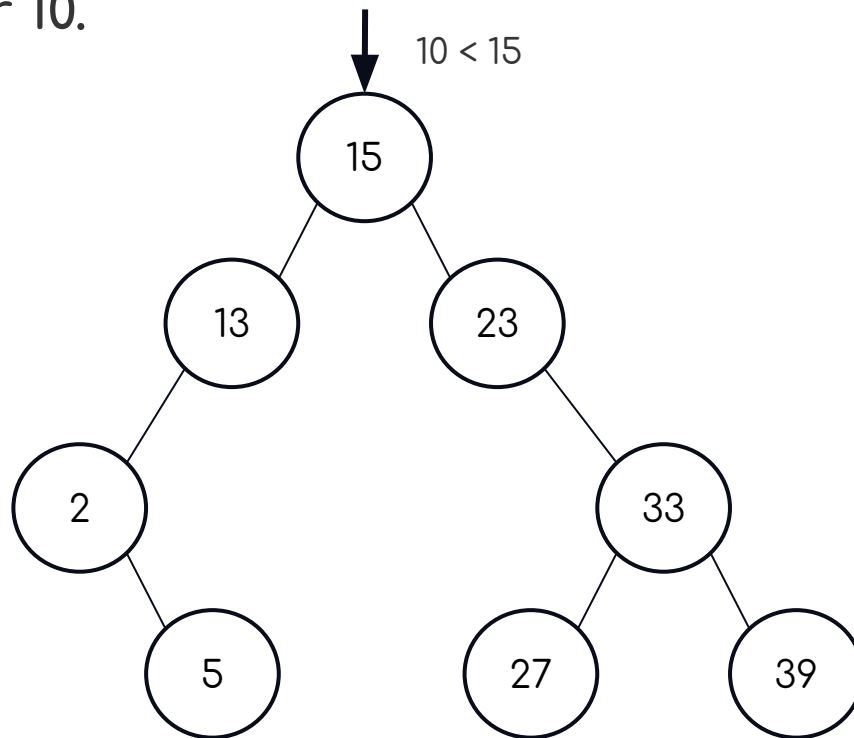
# Inserção

- Exemplo: Inserir 10.



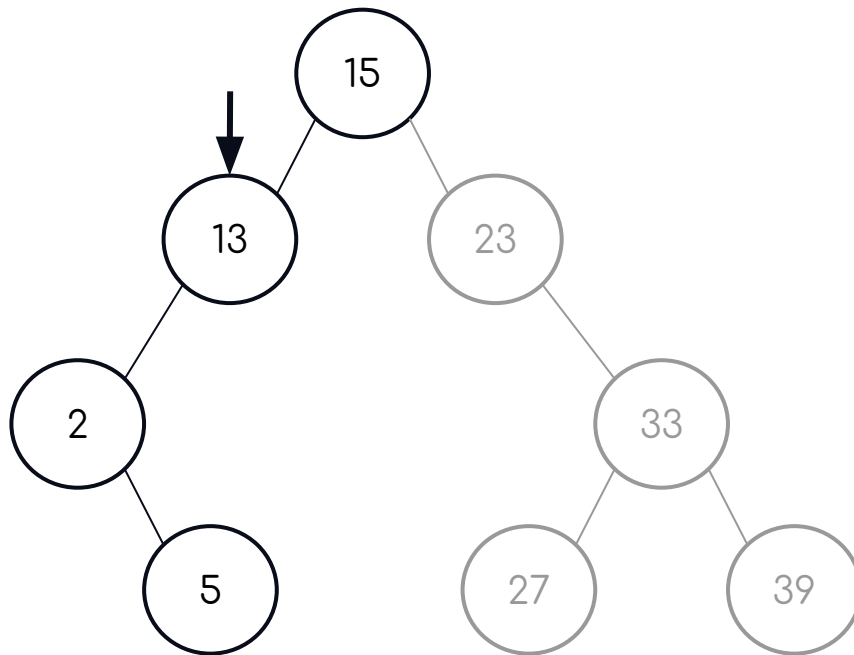
# Inserção

- Exemplo: Inserir 10.



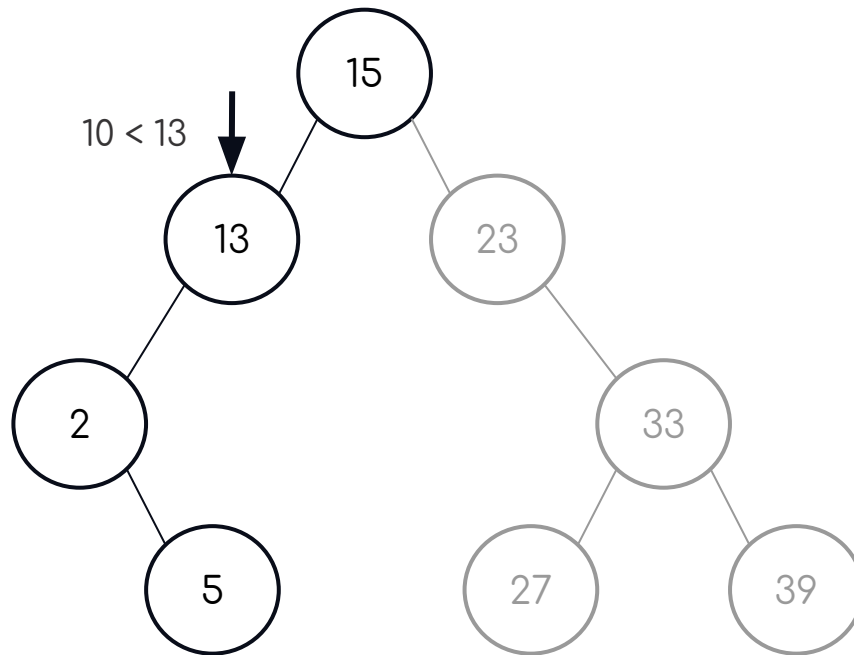
# Inserção

- Exemplo: Inserir 10.



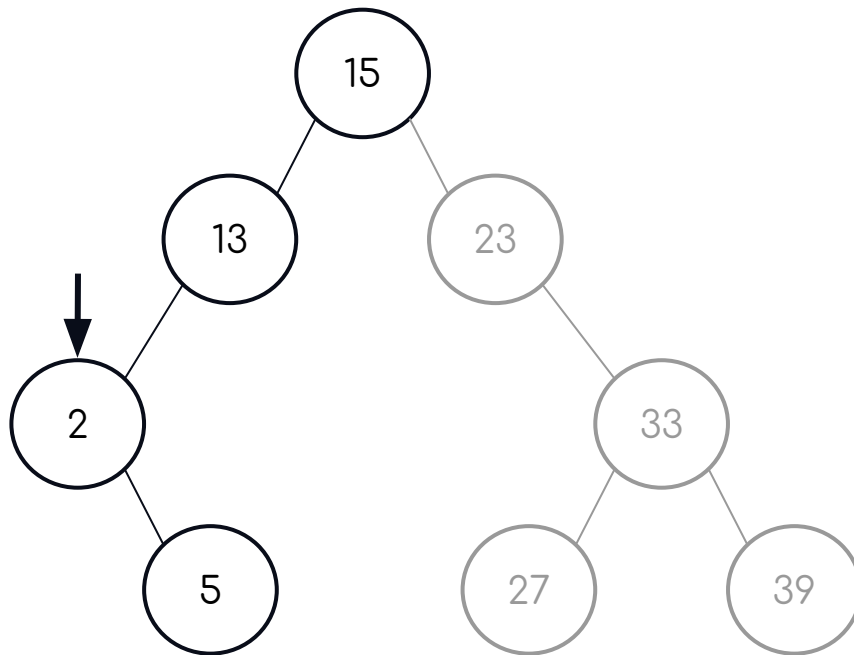
# Inserção

- Exemplo: Inserir 10.



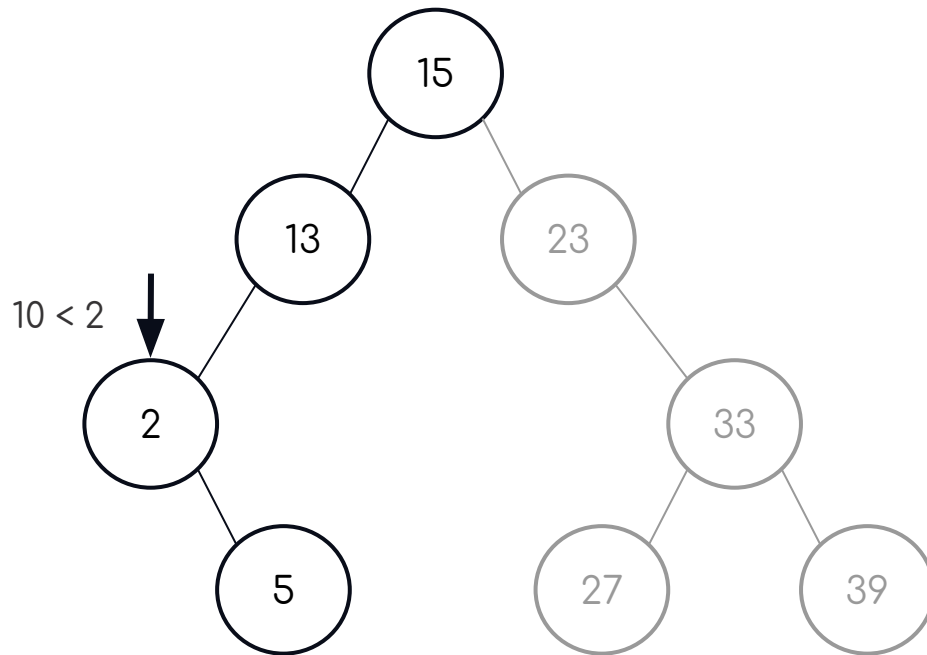
# Inserção

- Exemplo: Inserir 10.



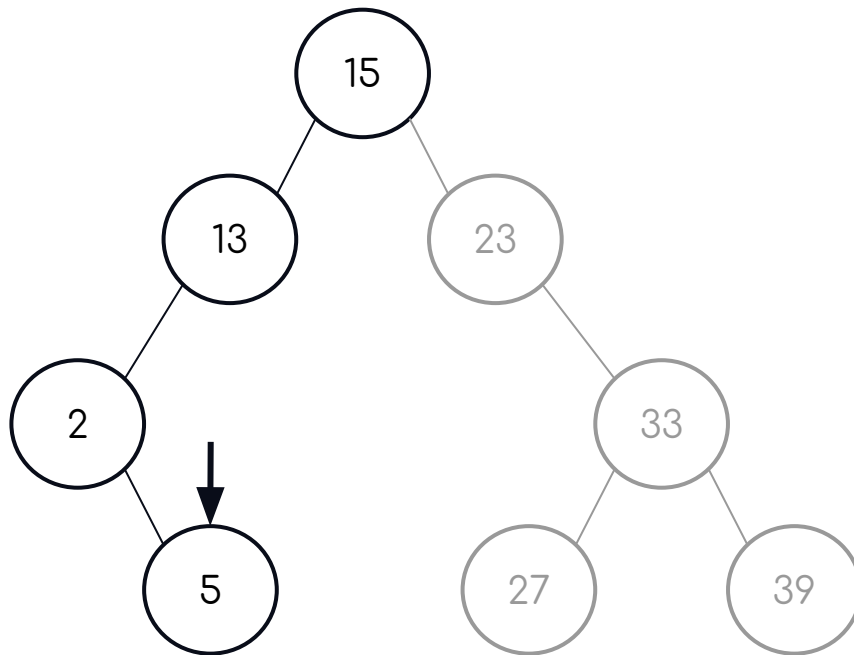
# Inserção

- Exemplo: Inserir 10.



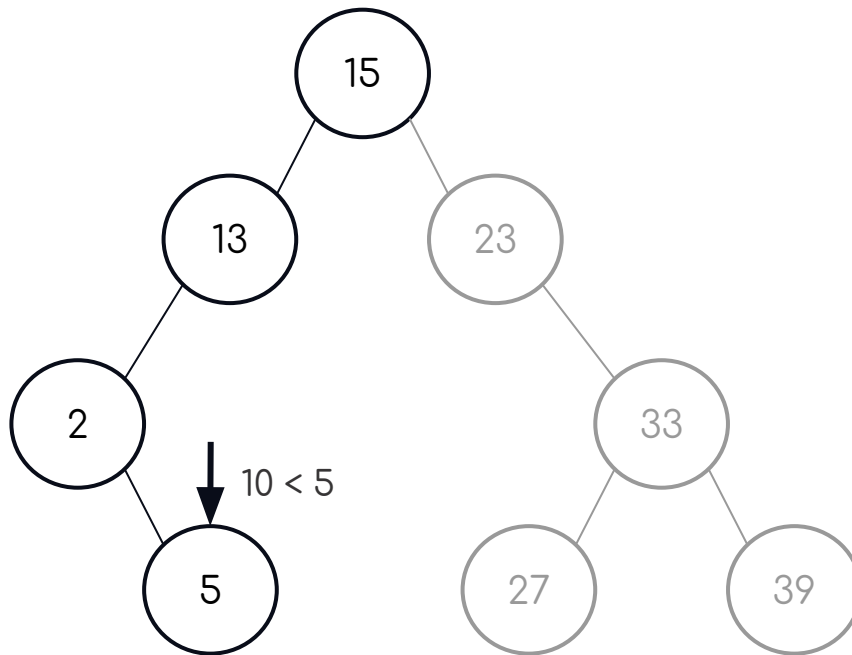
# Inserção

- Exemplo: Inserir 10.



# Inserção

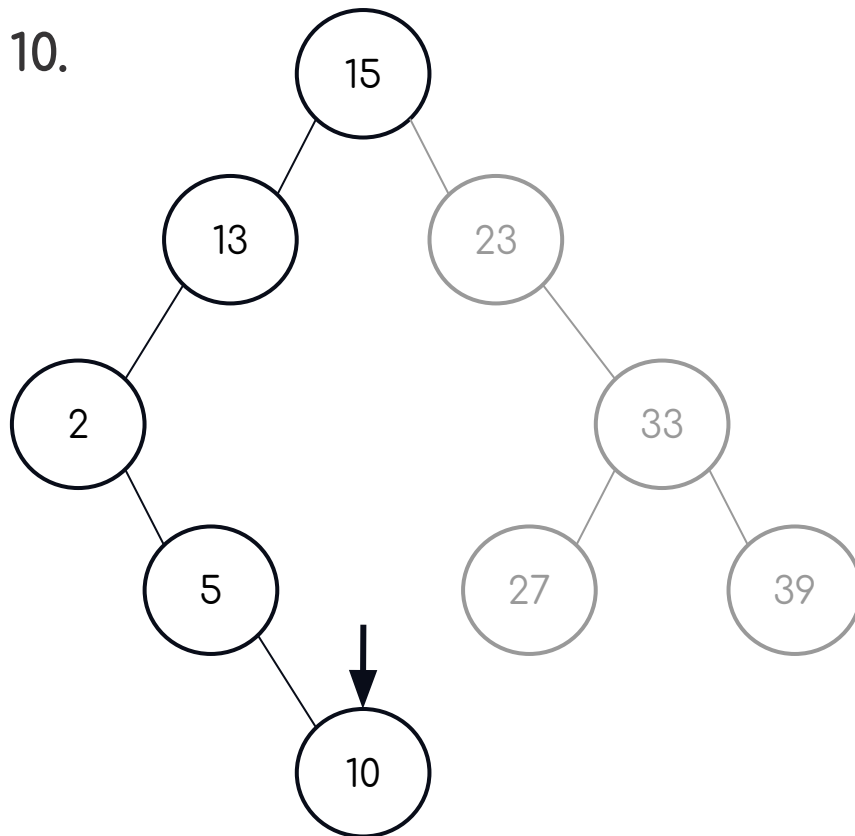
- Exemplo: Inserir 10.



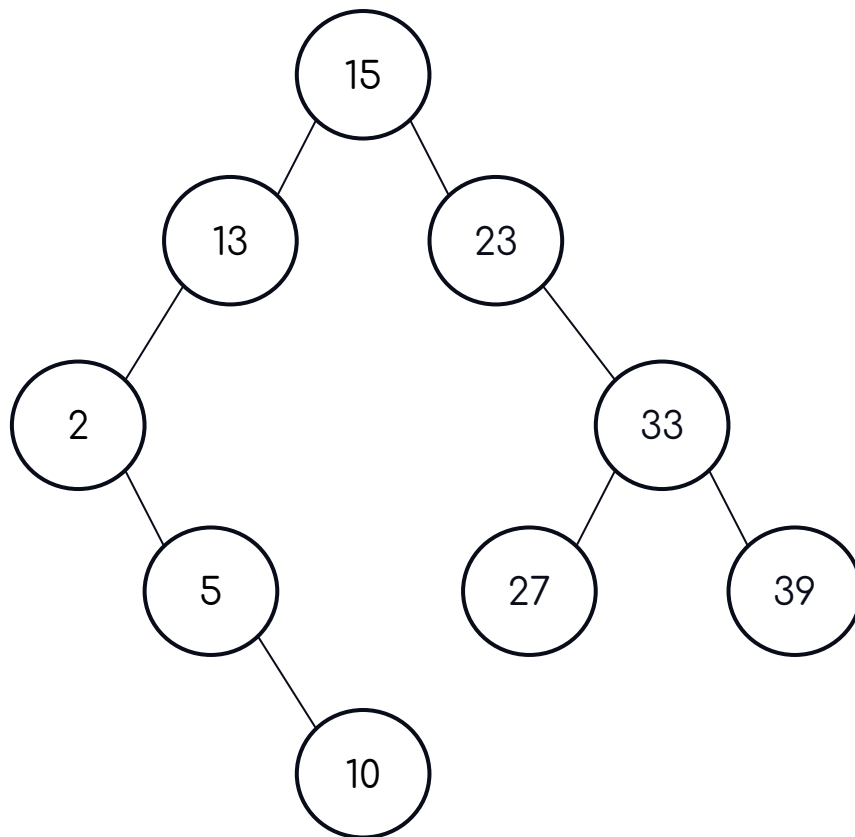


# Inserção

- Exemplo: Inserir 10.



# Inserção





# Inserção - Algoritmo

---

- Se a raiz for null, inserimos lá
- Senão:
  - Se a chave do elemento a ser inserido for menor que a raiz:
    - Insere na subárvore da esquerda
  - Senão:
    - Insere na subárvore da direita



# Inserção - Algoritmo

- Essa versão é recursiva:

```
void adicionar(No*& raiz, int valor)
{
    if (raiz == nullptr) // Se a árvore estiver vazia
    {
        raiz = new No; // Cria um novo nó
        raiz->valor = valor; // Atribui o valor ao nó
        raiz->esq = nullptr; // Inicializa o filho esquerdo como nulo
        raiz->dir = nullptr; // Inicializa o filho direito como nulo
    }
    else if (valor < raiz->valor) // Se o valor for menor que o valor do nó atual
    {
        adicionar(raiz->esq, valor); // Adiciona à subárvore esquerda
    }
    else if (valor > raiz->valor) // Se o valor for maior que o valor do nó atual
    {
        adicionar(raiz->dir, valor); // Adiciona à subárvore direita
    }
}
```



# Desafios





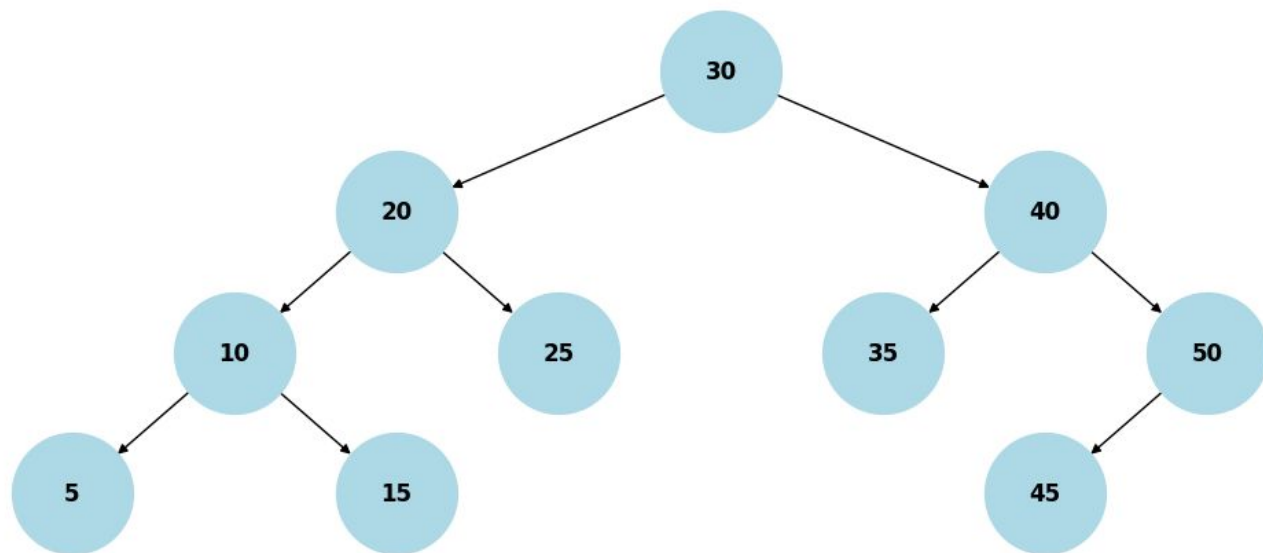
# Desafio

Considere que uma árvore binária de busca está inicialmente vazia. Insira os seguintes valores na ordem dada: 30, 20, 40, 10, 25, 35, 50, 5, 15, 45

Desenhe a árvore resultante após todas as inserções.

Depois, responda:

- Qual é a profundidade (nível máximo) da árvore?
- Quais são os nós folhas?
- Qual o percurso in-ordem da árvore?



Solução desafio 1



# Desafio

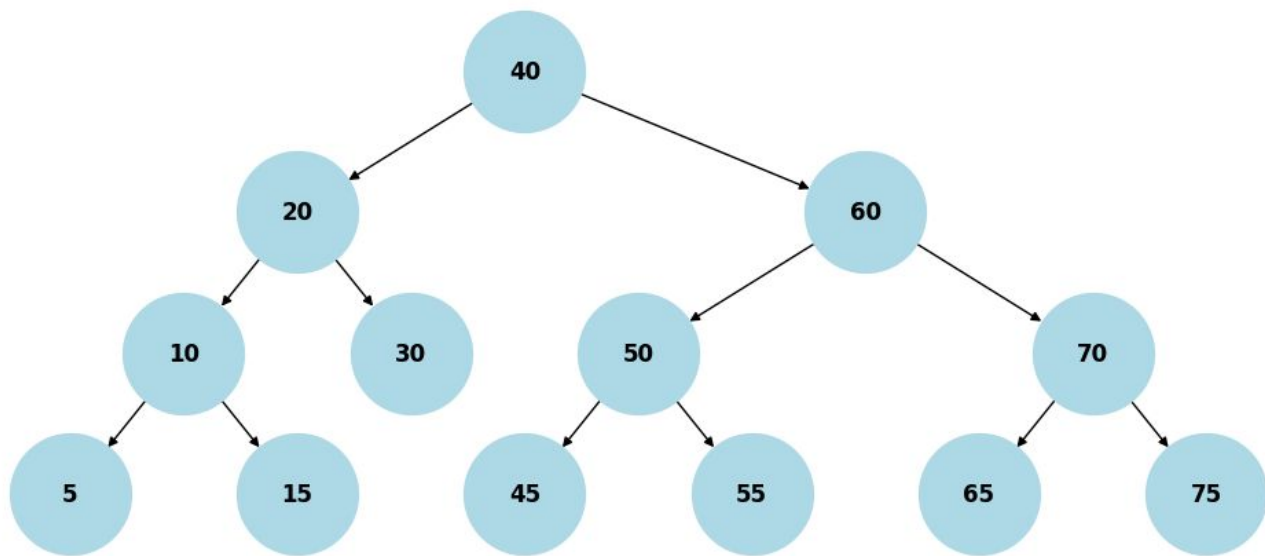
Considere uma árvore binária de busca construída com os valores: 40, 20, 60, 10, 30, 50, 70

Agora insira, nessa ordem, os seguintes valores: 5, 15, 45, 55, 65, 75.

Depois disso, responda:

- Qual o nível do nó com valor 55?
- Quais valores estão na subárvore à direita da raiz?
- Quais são os nós com exatamente um filho?





Solução desafio 1

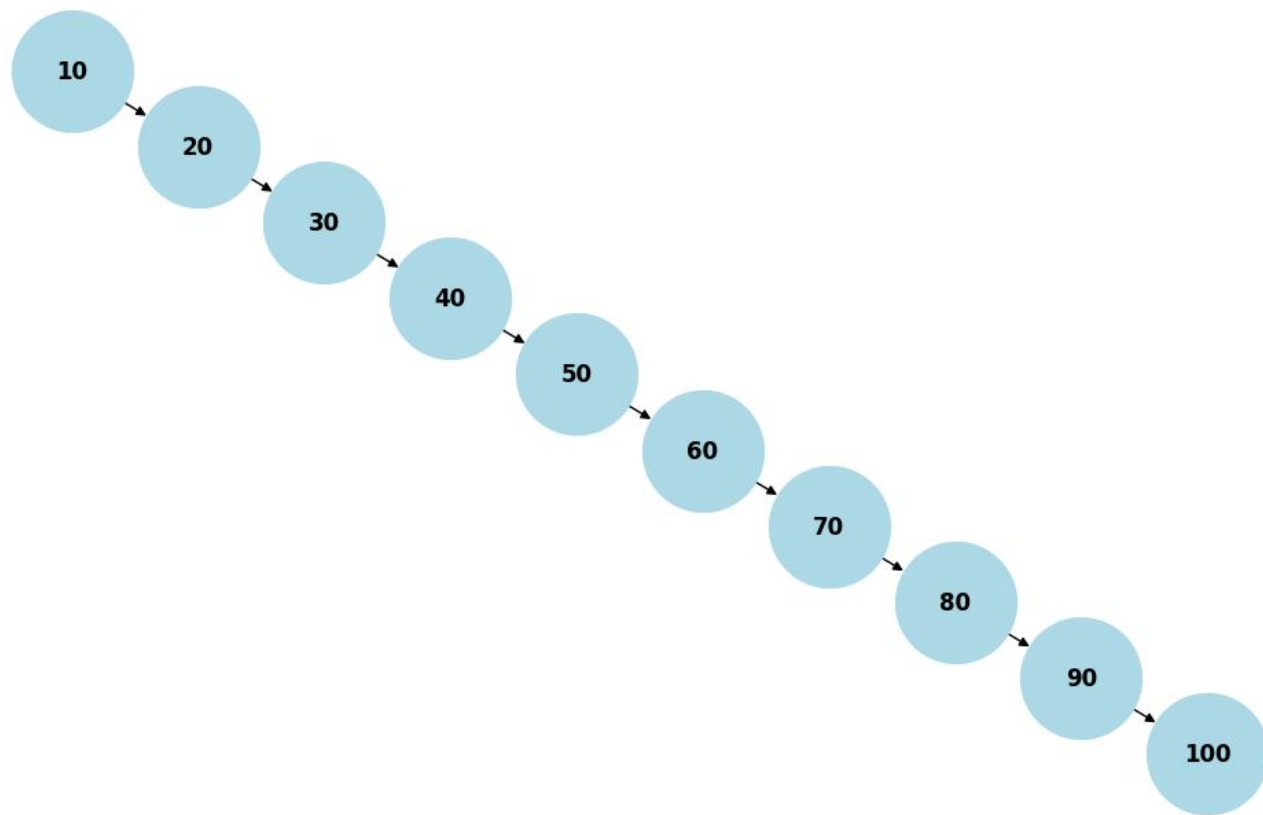


# Desafio


Crie uma sequência de inserção de 10 valores inteiros distintos tal que a árvore resultante:

- Tenha altura máxima (ou seja, vire uma lista encadeada degenerada).
- Seja uma árvore binária de busca válida.
- Tenha todos os valores na subárvore da direita da raiz.


Explique sua escolha e desenhe a árvore.



Solução desafio 1



# Buscar um Elemento





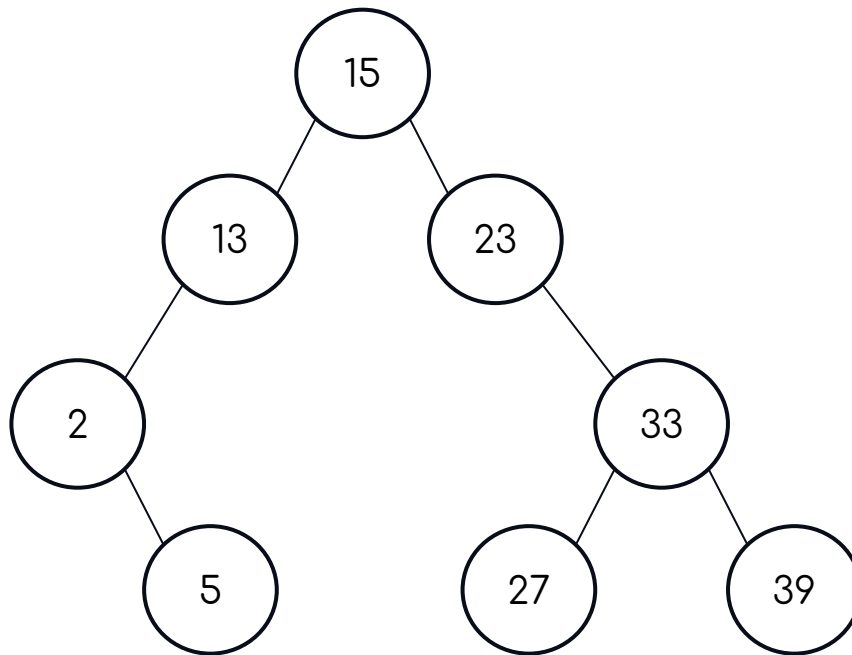
## Desafio

Você entendeu realmente a estrutura da árvore?

Para confirmar isso pense no algoritmo para buscar um elemento na árvore binária de busca, depois:

- Descreva o algoritmo
- Implemente

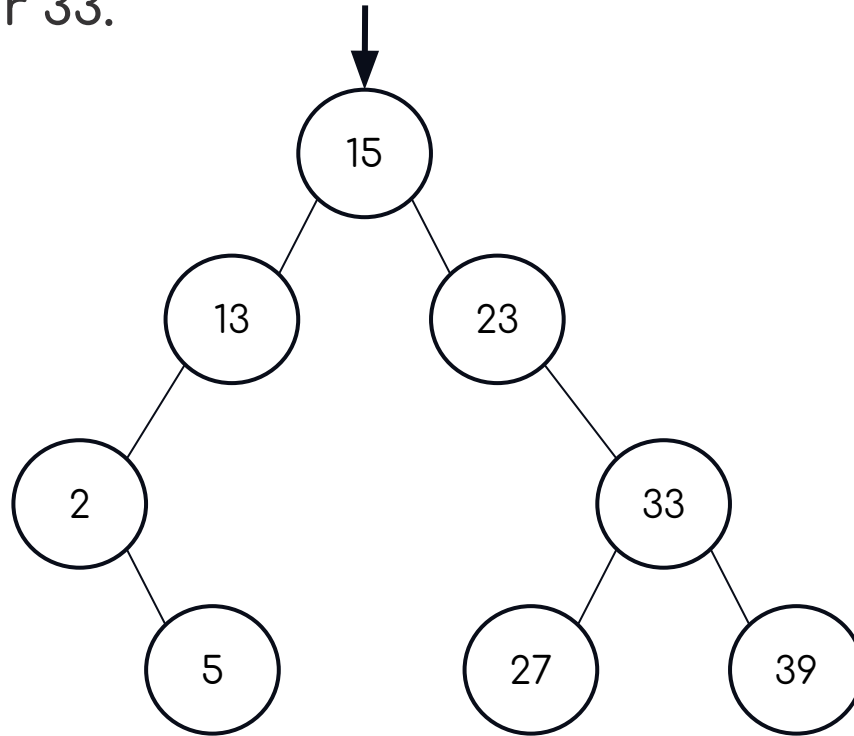
- Exemplo: Buscar 33.





# Busca

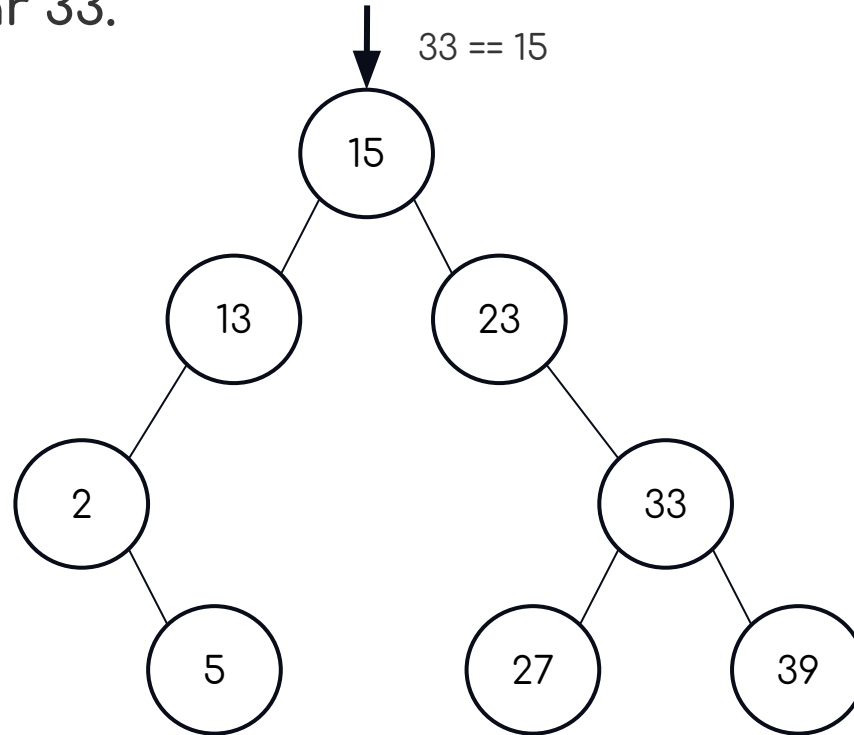
- Exemplo: Buscar 33.





# Busca

- Exemplo: Buscar 33.

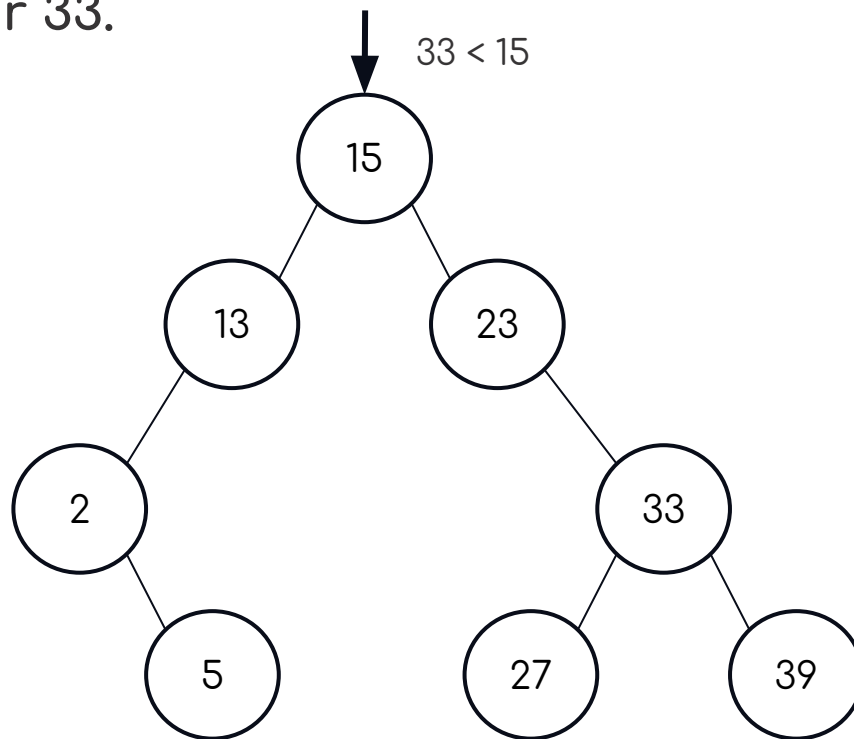






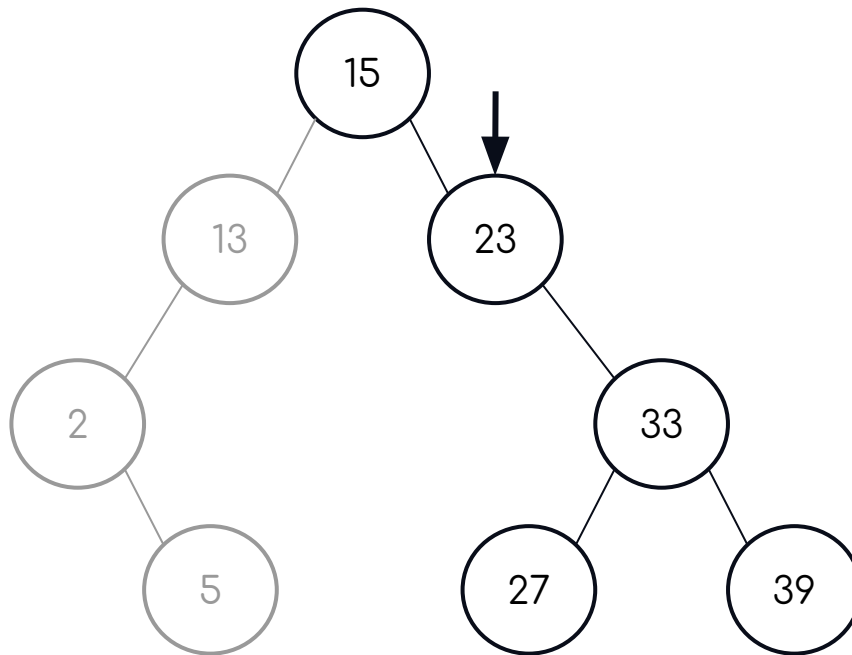
# Busca

- Exemplo: Buscar 33.

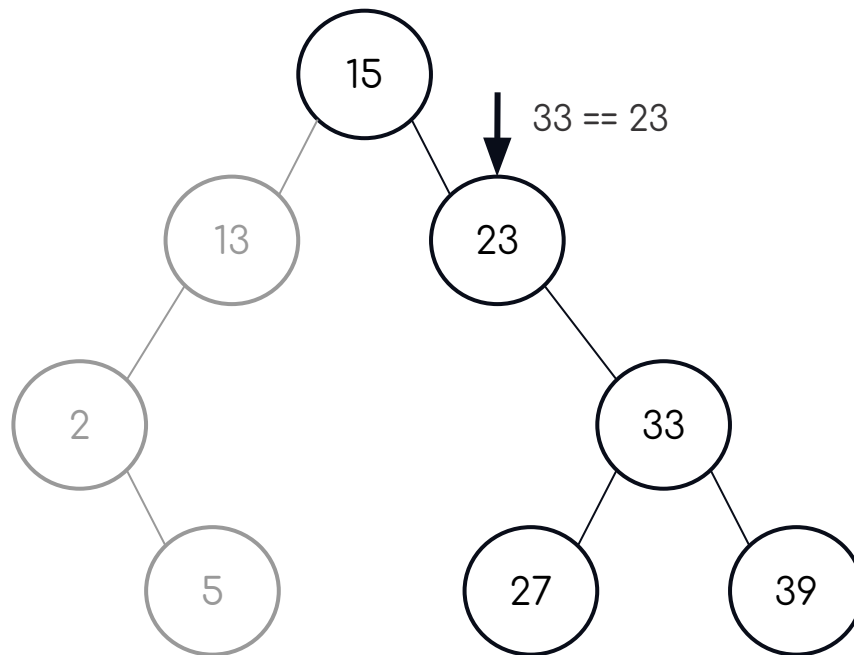


# Inserção

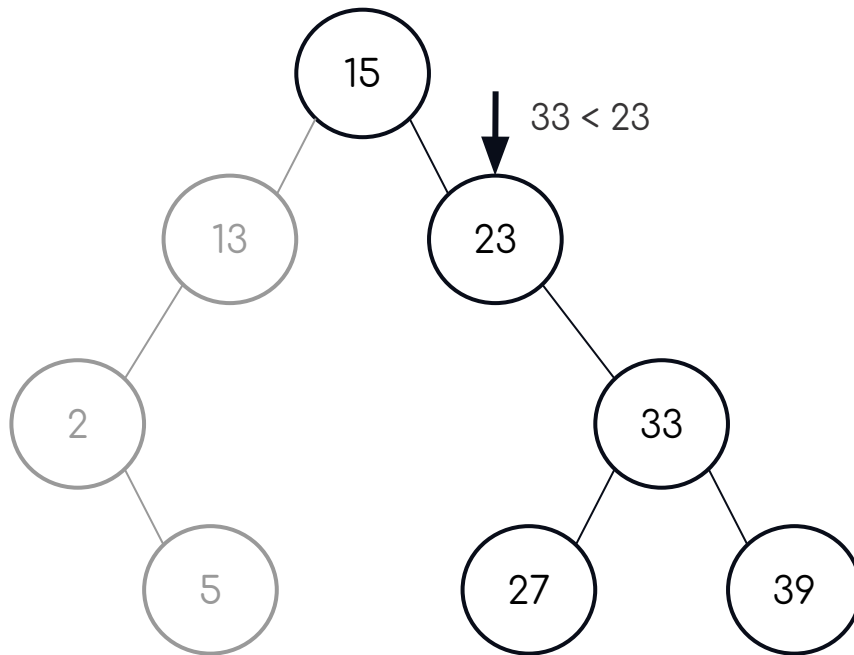
- Exemplo: Buscar 33.



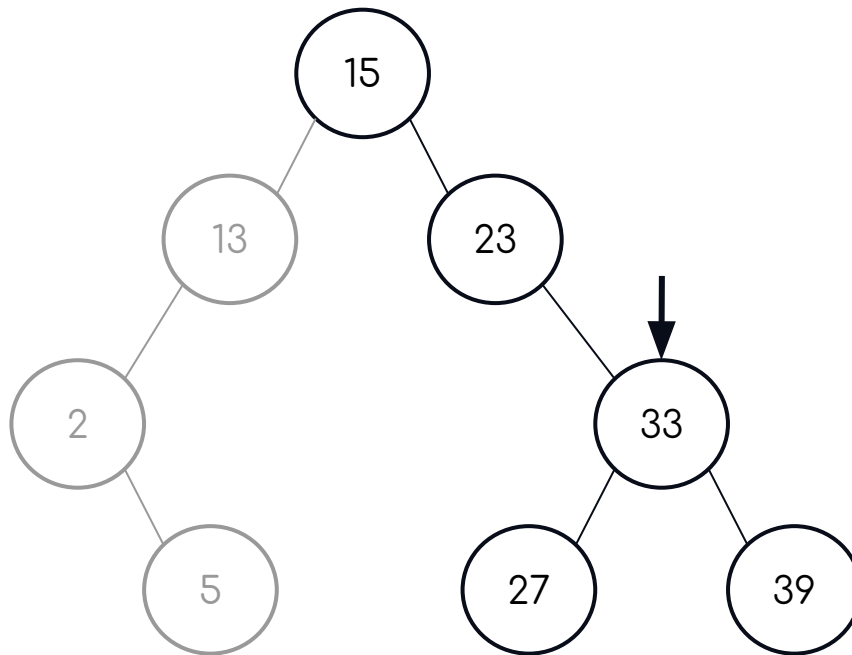
- Exemplo: Buscar 33.



- Exemplo: Buscar 33.



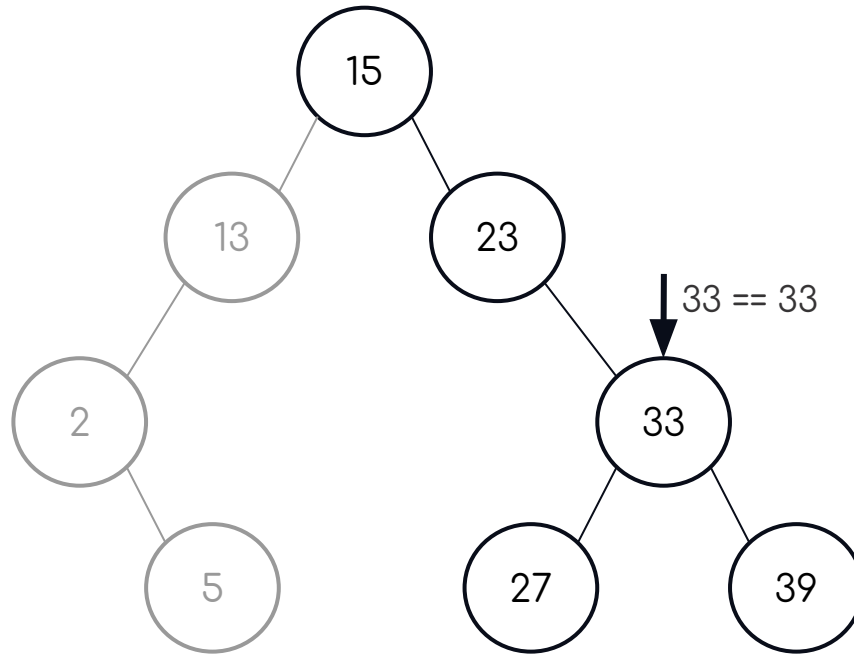
- Exemplo: Buscar 33.





# Busca

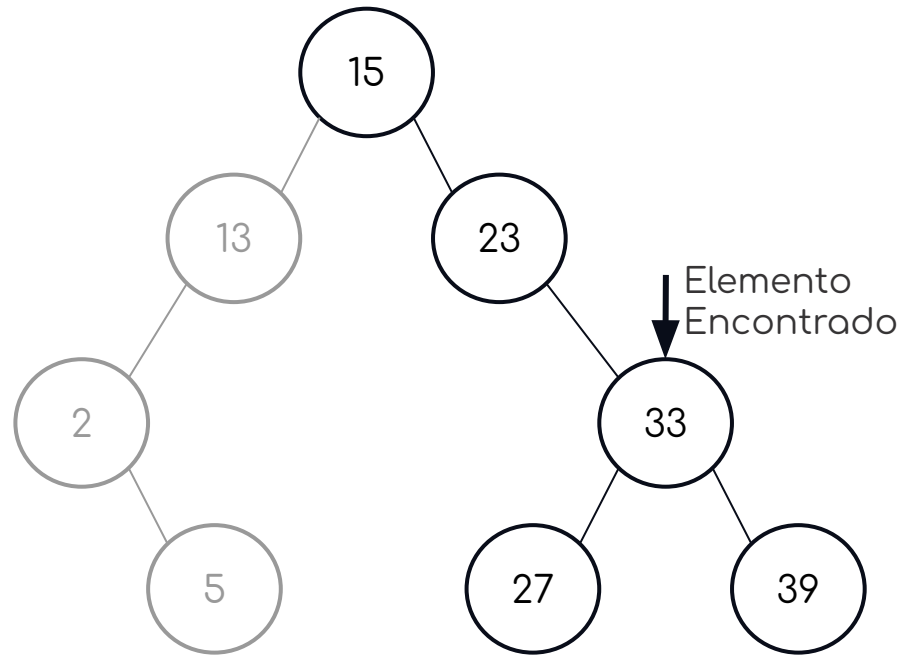
- Exemplo: Buscar 33.





# Busca

- Exemplo: Buscar 33.





# Busca - Algoritmo

---

- Se a raiz for null
  - O elemento não foi encontrado
- Senão:
  - Se a chave do elemento a ser buscado é igual a raiz:
    - O elemento foi encontrado
  - Senão se a chave do elemento a ser buscado for menor que a raiz:
    - Buscar na subárvore da esquerda
  - Senão:
    - Buscar na subárvore da direita





# Busca - Algoritmo

- Essa versão é recursiva:

```
void buscar(No* raiz, int valor)
{
    if (raiz == nullptr) // Se a árvore estiver vazia
    {
        cout << "Valor não encontrado." << endl; // Valor não encontrado
        return;
    }
    if (raiz->valor == valor) // Se o valor do nó atual for igual ao valor buscado
    {
        cout << "Valor encontrado: " << raiz->valor << endl; // Valor encontrado
        return;
    }
    else if (valor < raiz->valor) // Se o valor buscado for menor que o valor do nó atual
    {
        buscar(raiz->esq, valor); // Busca na subárvore esquerda
    }
    else // Se o valor buscado for maior que o valor do nó atual
    {
        buscar(raiz->dir, valor); // Busca na subárvore direita
    }
}
```



# Percorrer os Elementos





# Percorrer os Elementos

---

- Percurso (traversal): visitar os nós de forma sistemática:
  - Pré-ordem (pre-order): raiz  $\rightarrow$  esquerda  $\rightarrow$  direita
  - Em ordem (in-order): esquerda  $\rightarrow$  raiz  $\rightarrow$  direita
  - Pós-ordem (post-order): esquerda  $\rightarrow$  direita  $\rightarrow$  raiz
  - Largura (BFS): nível por nível



# Percorrer os Elementos

---

- Todo o percurso envolve 3 elementos:
  - O nó raiz
  - A subárvore à esquerda de cada nó
  - A subárvore à direita de cada nó



# Percorrer os Elementos

---

- Algumas possibilidades:
  - Subárvore esquerda - raiz - subárvore direita
  - Subárvore direita - raiz - subárvore esquerda
  - Raiz - subárvore esquerda - subárvore direita



# Percorrer os Elementos

---

- Em relação a árvores binárias de busca, uma ordem bastante útil é subárvore esquerda - raiz - subárvore direita também chamada de em ordem, in-order transversal, varredura infixa, ou varredura central. Nessa varredura, os nós são visitados na ordem crescente das chaves de busca.



# Contagem - Algoritmo

---

- Se a raiz for null
  - Contagem = 0
- Senão:
  - Contém subárvore à esquerda:
    - Contar subárvore da esquerda
  - Conta a raiz
    - Contagem += 1
  - Contém subárvore à direita:
    - Contar subárvore da direita



# Contagem - Algoritmo

---

- Essa versão é recursiva:

```
int contagem(No* raiz)
{
    if (raiz == nullptr) // Se a árvore estiver vazia
    {
        return 0; // Retorna 0
    }
    else // Se a árvore não estiver vazia
    {
        // Retorna 1 + contagem da subárvore esquerda + contagem da subárvore direita
        return contagem(raiz->esq) + 1 + contagem(raiz->dir);
    }
}
```





# Desafio

- Converta o código de contagem para um código de exibição dos elementos em ordem.



# Percorrer os Elementos

---

## In-order reversa

- Em árvores binárias de busca, a varredura subárvore direita - raiz - subárvore esquerda é conhecida como in-order reversa, varredura infixa reversa ou varredura central invertida. Nessa varredura, os nós são visitados na ordem decrescente das chaves de busca. Esse tipo de percurso é útil quando se deseja processar os elementos do maior para o menor valor, preservando a lógica da estrutura da árvore.



# Desafio

- Converta o código de exibição dos elementos em ordem para um código de exibição dos elementos em ordem reversa.



# Percorrer os Elementos

---

## Pré-ordem

- Em relação a árvores binárias de busca, uma ordem de visitação importante é raiz - subárvore esquerda - subárvore direita, também chamada de pré-ordem, pre-order traversal ou varredura prefixa. Nessa varredura, a raiz de cada subárvore é visitada antes de seus filhos, o que a torna útil para gerar uma cópia da árvore ou para expressar sua estrutura de forma direta.



## Desafio

- Converta o código de exibição dos elementos em ordem para um código de exibição dos elementos em pré-ordem.



# Percorrer os Elementos

---

## Pós-ordem

- Em árvores binárias de busca, outra forma de percorrer os nós é subárvore esquerda - subárvore direita - raiz, conhecida como pós-ordem, post-order traversal ou varredura posfixa. Esse tipo de varredura é útil para operações que requerem o processamento dos filhos antes do nó pai, como a exclusão de todos os nós da árvore.



## Desafio

- Converta o código de exibição dos elementos em ordem para um código de exibição dos elementos em pós-ordem.



# Percorrer os Elementos

---

## Largura

- A varredura por largura, ou percurso em nível (level-order traversal), percorre os nós da árvore por níveis, da raiz para as folhas e da esquerda para a direita em cada nível. Esse tipo de percurso é útil para algoritmos que precisam analisar a árvore como um todo, nível a nível, como em algoritmos de busca em largura ou em aplicações que envolvem filas.





## Desafio

- Crie um código de exibição dos elementos em largura (dica use Fila).



# Exibição - Algoritmo

---

- Para exibir por largura, foi necessário usar uma fila:

```
// Função de varredura por largura
void percorrerPorLargura(No* raiz) {
    if (raiz == nullptr) return;

    Fila fila;
    fila.enqueue(raiz);

    while (!fila.vazia()) {
        No* atual = fila.dequeue();
        cout << atual->valor << " ";

        if (atual->esquerda != nullptr)
            fila.enqueue(atual->esquerda);
        if (atual->direita != nullptr)
            fila.enqueue(atual->direita);
    }
}
```



# Remover um Elemento





# Remover um Elemento

---

- Ao remover um nó surgem alguns problemas:
  - O que fazer com as subárvores desse nó?
  - Como tratar isso mantendo a árvore como uma Árvore Binária de Busca?



# Remover um Elemento

---

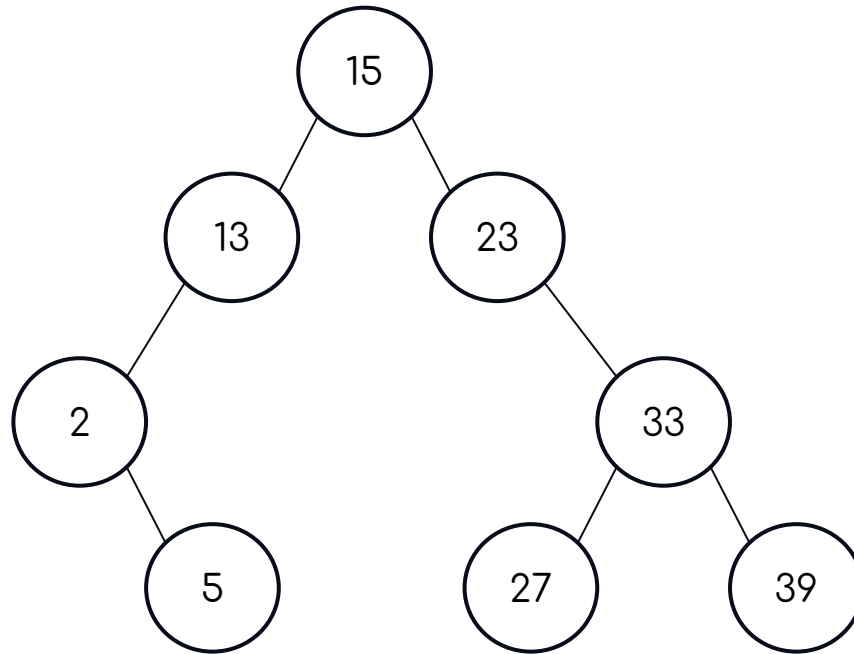
- Como remover:
  - Se o nó possui somente um filho, basta substituí-lo.
  - Se o nó possui dois filhos:
    - Substituímos o nó a ser retirado pelo nó mais à direita da subárvore da esquerda
    - OU
    - Substituímos o nó a ser retirado pelo nó mais à esquerda da subárvore da direita



# Remover

---

- Exemplo 1: Remover 15.

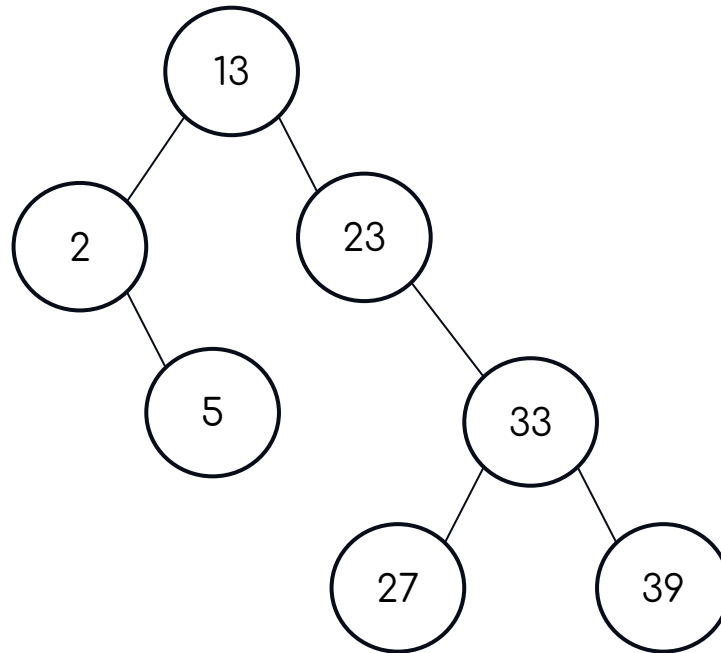




# Remover

---

- Exemplo 1: Remover 15.





# Remover

---

OU

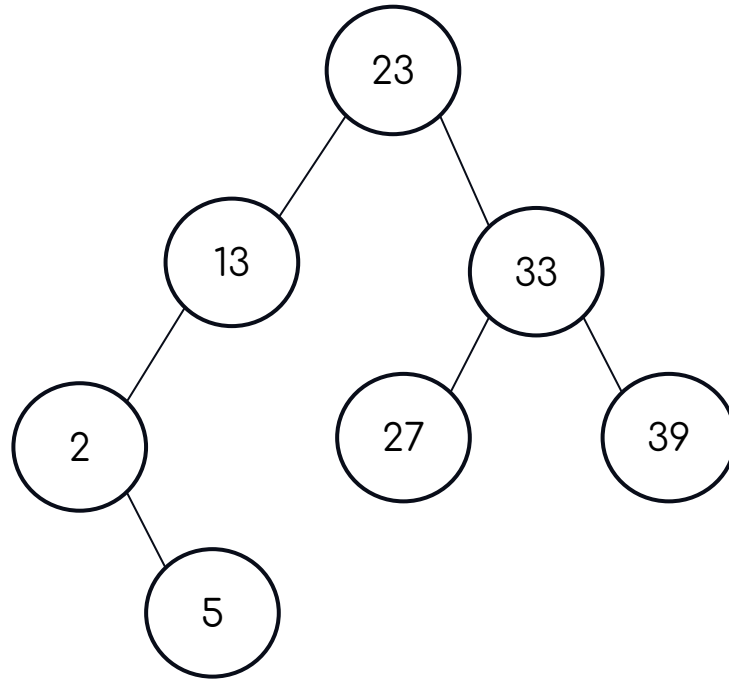




# Remover

---

- Exemplo 1: Remover 15.

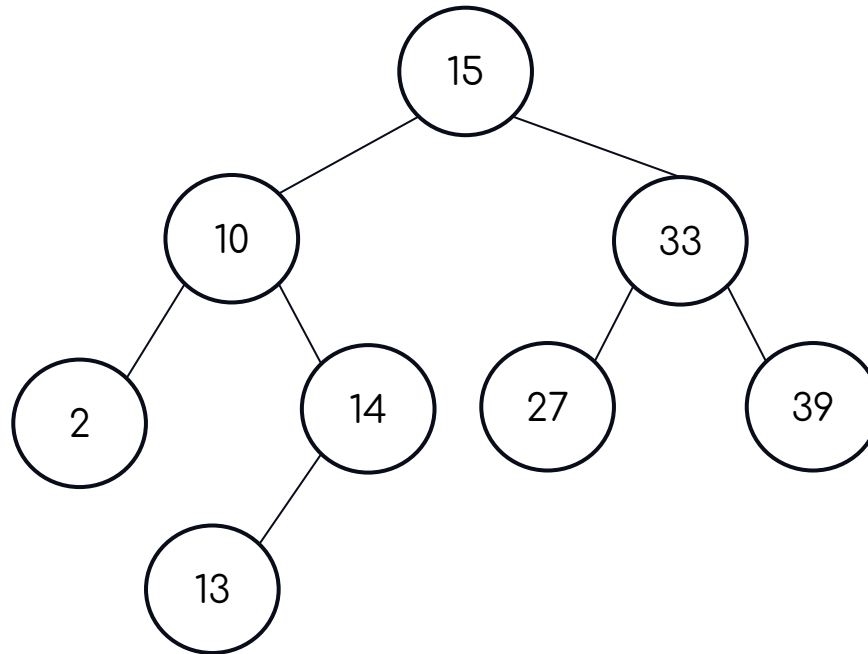




# Remover

---

- Exemplo 2: Remover 15.

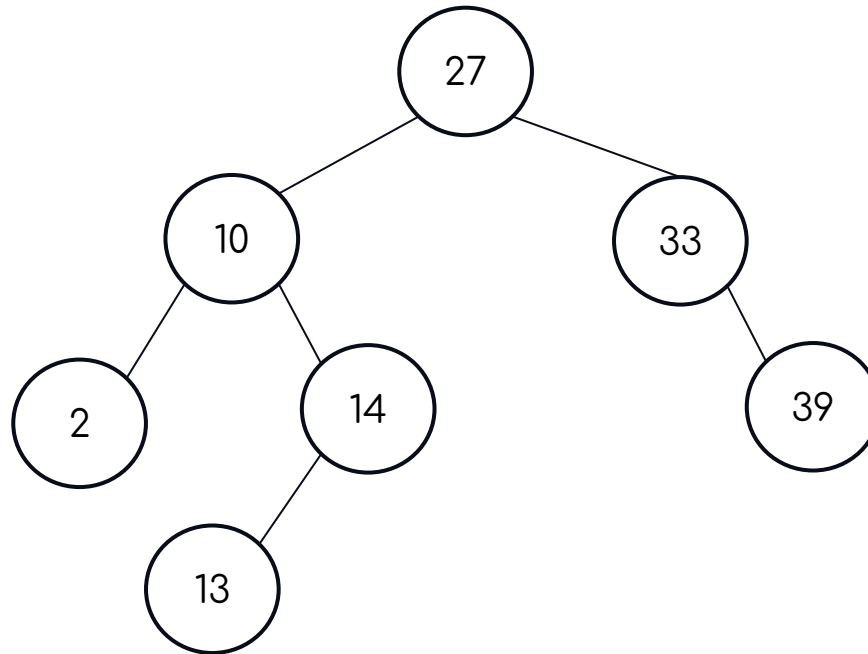




# Remover

---

- Exemplo 2: Remover 15.





# Remover

---

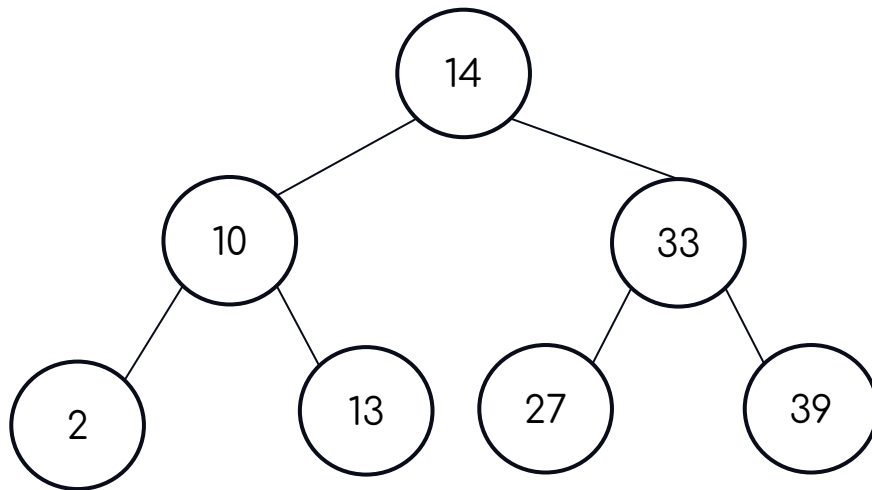
OU



# Remover

---

- Exemplo 2: Remover 15.



- Note que o 13 vira filho do pai do 14, ou seja, vira filho do 10.



# Remover um Elemento

---

- Quais informações precisamos saber para remover um nó:
  - O nó a ser removido.
  - O pai do nó a ser removido.
  - O nó substituto.
  - O pai do nó substituto.
- Para essa aula vamos substituir o nó a ser retirado pelo nó mais à direita da subárvore da esquerda.



# Remover - Algoritmo Recursivo

---

- Se a árvore estiver vazia:
  - Valor não encontrado.
- Se o valor buscado for menor que o valor do nó atual:
  - Remove na subárvore esquerda.
- Se o valor buscado for maior que o valor do nó atual:
  - Remove na subárvore direita.
- Senão o valor buscado for igual ao valor do nó atual:
  - Se não houver filho esquerdo:
    - Substitui pelo filho direito.
  - Senão se não houver filho direito:
    - Substitui pelo filho esquerdo.
  - Senão:
    - Encontra o sucessor (maior na subárvore esquerda)
    - Substitui o valor do nó atual pelo sucessor
    - Remove o sucessor da subárvore direita

# Remover - Algoritmo Recursivo

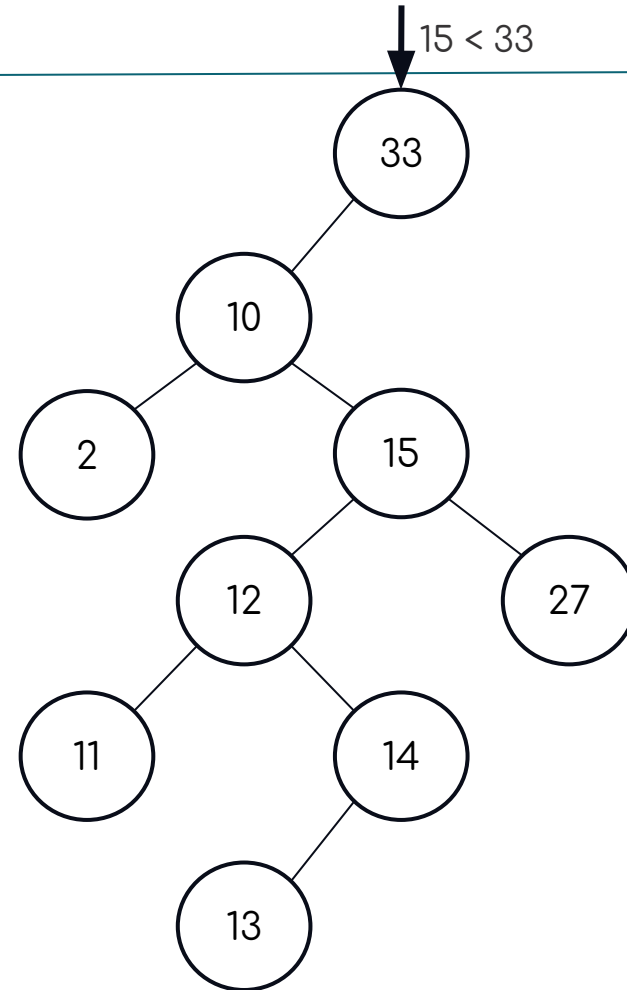
```
void remover(No*& raiz, int valor)
{
    if (raiz == nullptr) // Se a árvore estiver vazia
    {
        cout << "Valor não encontrado." << endl; // Valor não encontrado
        return;
    }
    if (valor < raiz->valor) // Se o valor buscado for menor que o valor do nó atual
    {
        remover(raiz->esq, valor); // Busca na subárvore esquerda
    }
    else if (valor > raiz->valor) // Se o valor buscado for maior que o valor do nó atual
    {
        remover(raiz->dir, valor); // Busca na subárvore direita
    }
    else // Se o valor buscado for igual ao valor do nó atual
    {
        No* temp = raiz; // Armazena o nó a ser removido
        if (raiz->esq == nullptr) // Se não houver filho esquerdo
        {
            raiz = raiz->dir; // Substitui pelo filho direito
        }
        else if (raiz->dir == nullptr) // Se não houver filho direito
        {
            raiz = raiz->esq; // Substitui pelo filho esquerdo
        }
        else // Se houver ambos os filhos
        {
            No* sucessor = raiz->dir; // Encontra o sucessor (menor na subárvore direita)
            while (sucessor->esq != nullptr)
            {
                sucessor = sucessor->esq;
            }
            raiz->valor = sucessor->valor; // Substitui o valor do nó atual pelo sucessor
            remover(raiz->dir, sucessor->valor); // Remove o sucessor da subárvore direita
        }
        delete temp; // Libera a memória do nó removido
    }
}
```





# Remover

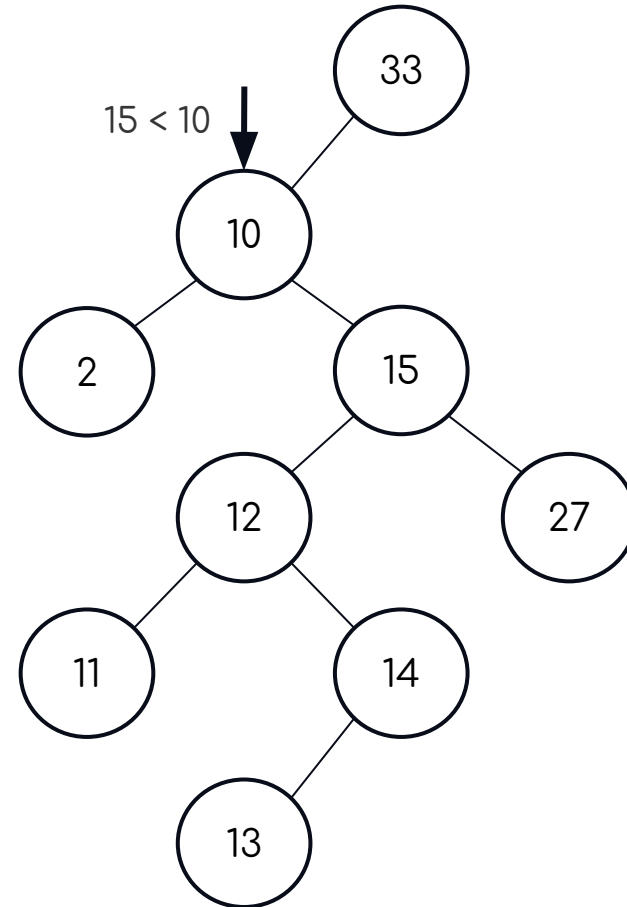
- Exemplo 1: Remover 15.





# Remover

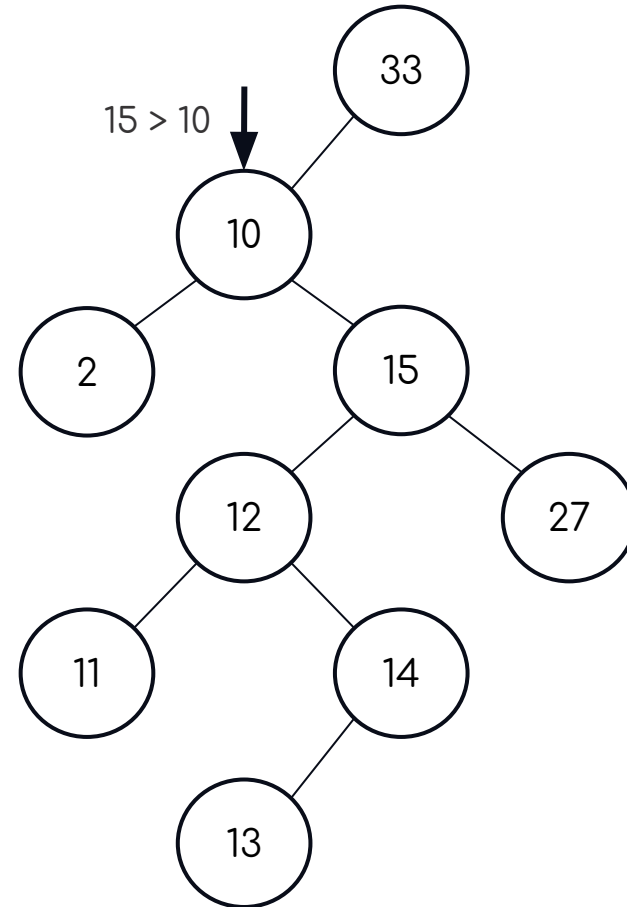
- Exemplo 1: Remover 15.





# Remover

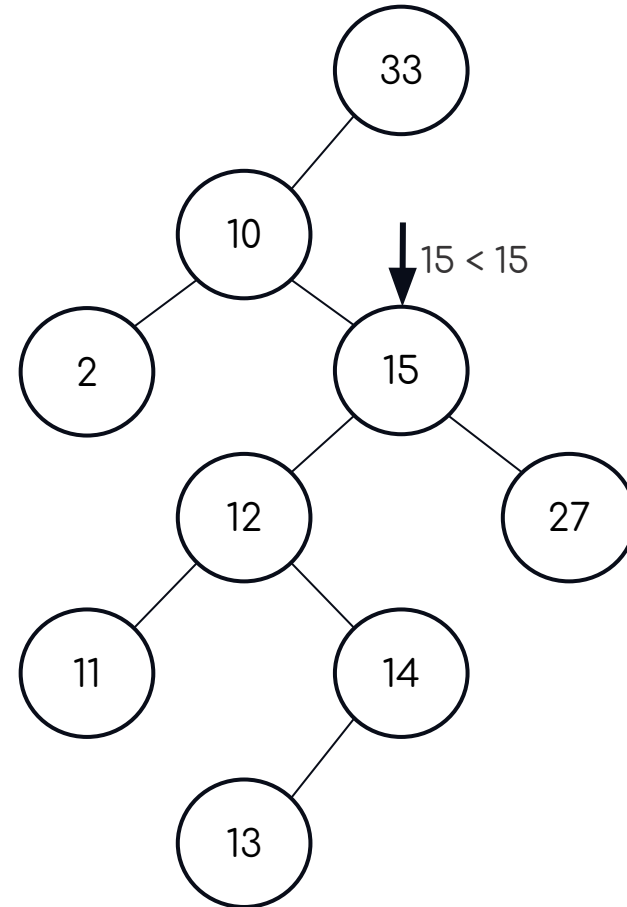
- Exemplo 1: Remover 15.





# Remover

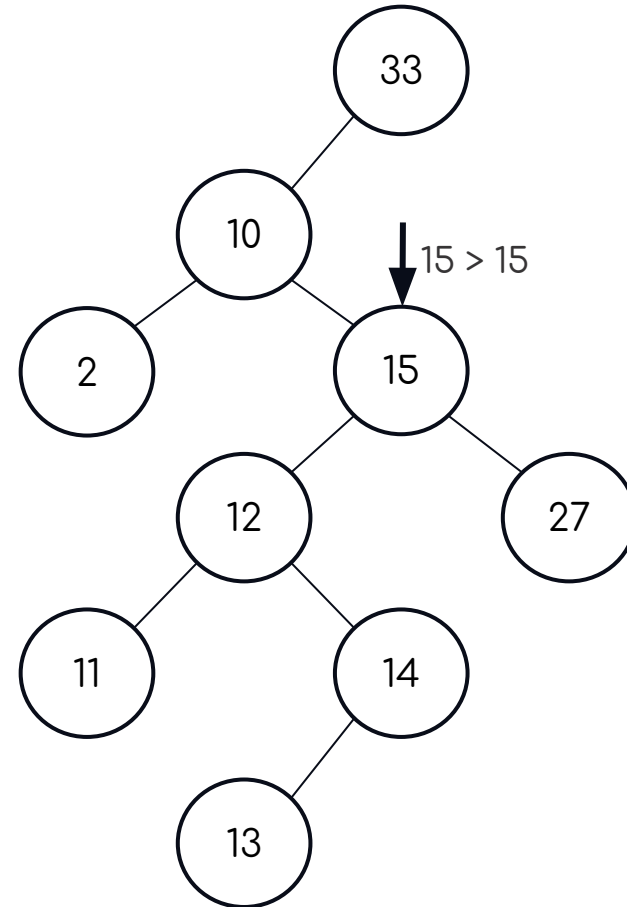
- Exemplo 1: Remover 15.





# Remover

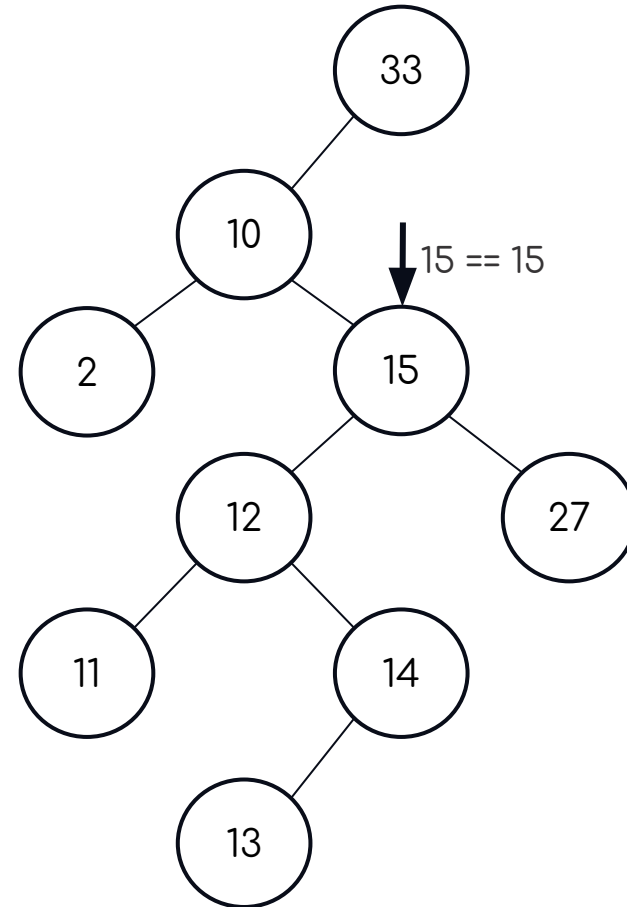
- Exemplo 1: Remover 15.





# Remover

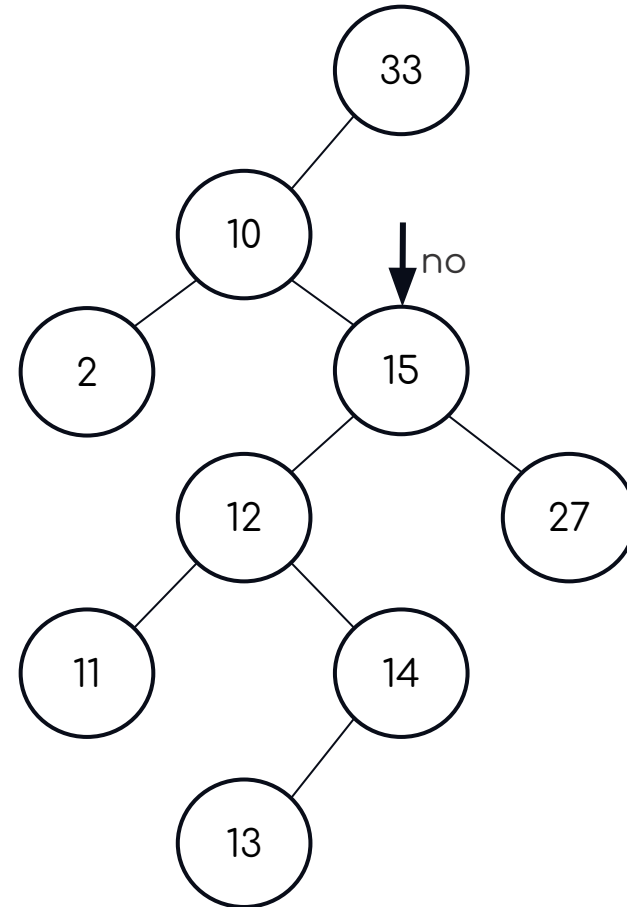
- Exemplo 1: Remover 15.





# Remover

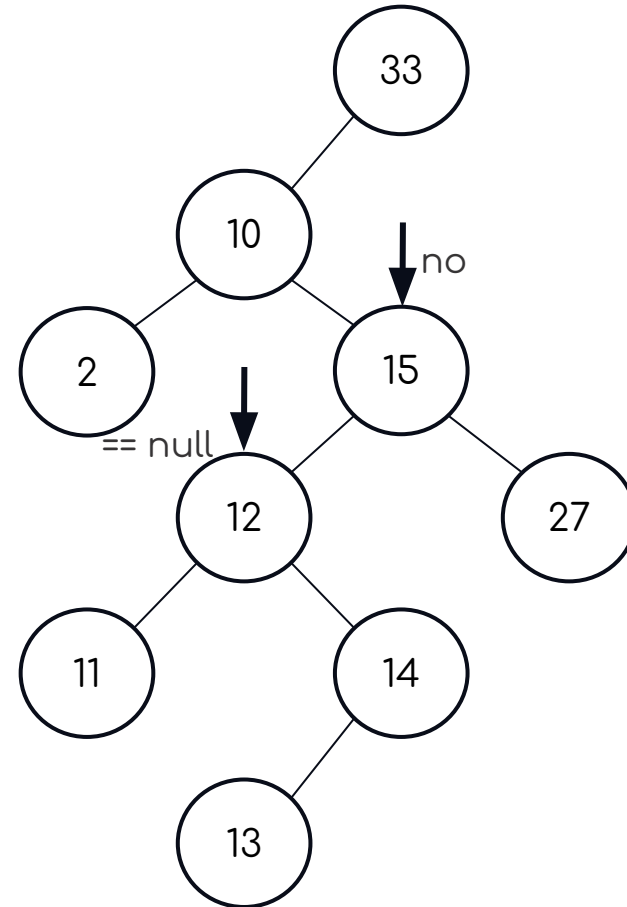
- Exemplo 1: Remover 15.





# Remover

- Exemplo 1: Remover 15.

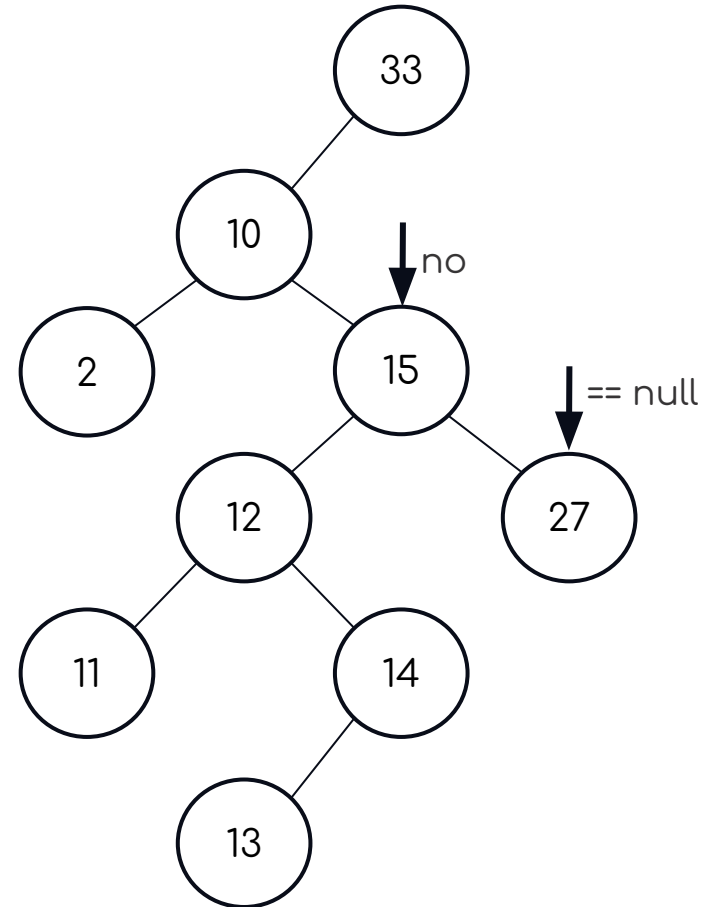






# Remover

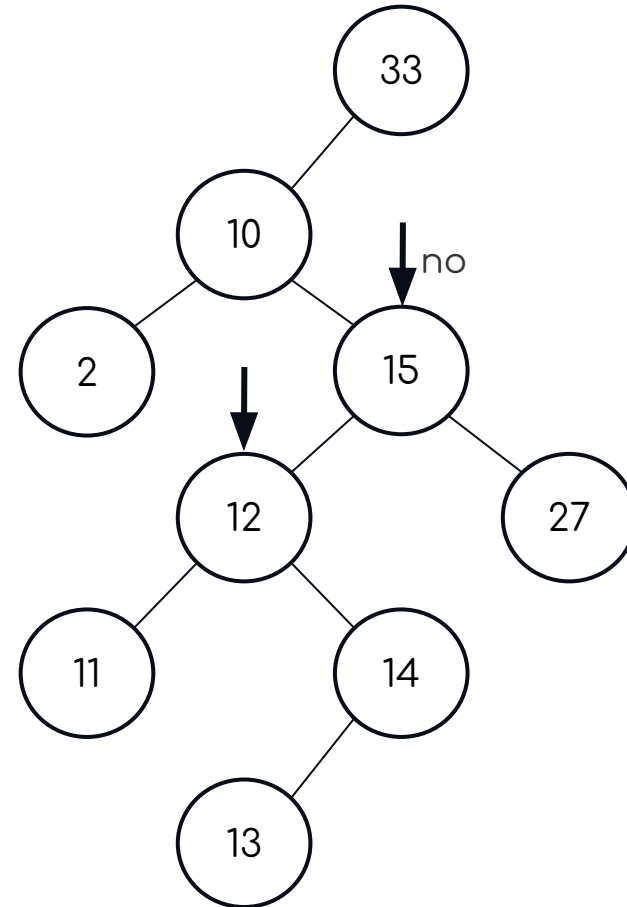
- Exemplo 1: Remover 15.





# Remover

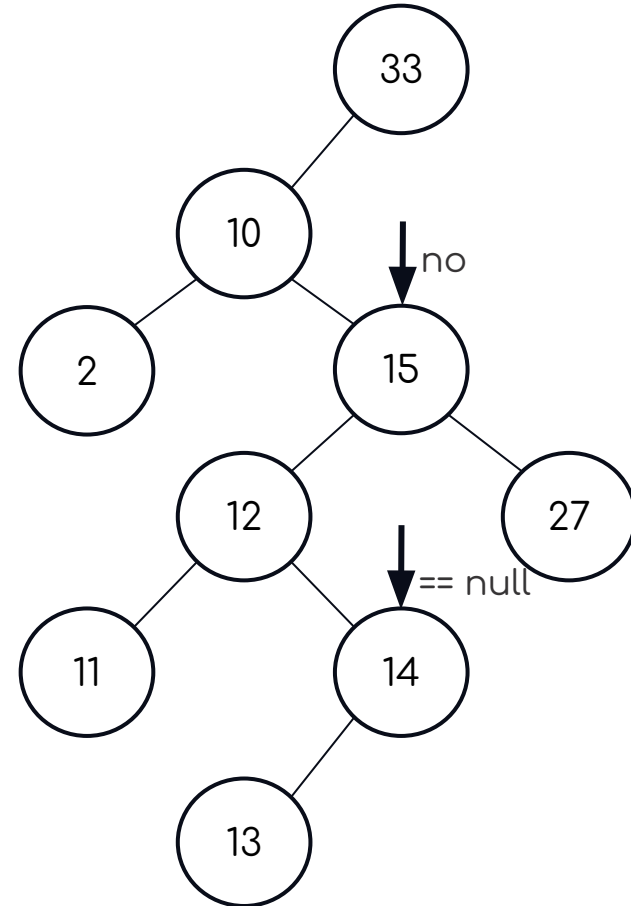
- Exemplo 1: Remover 15.





# Remover

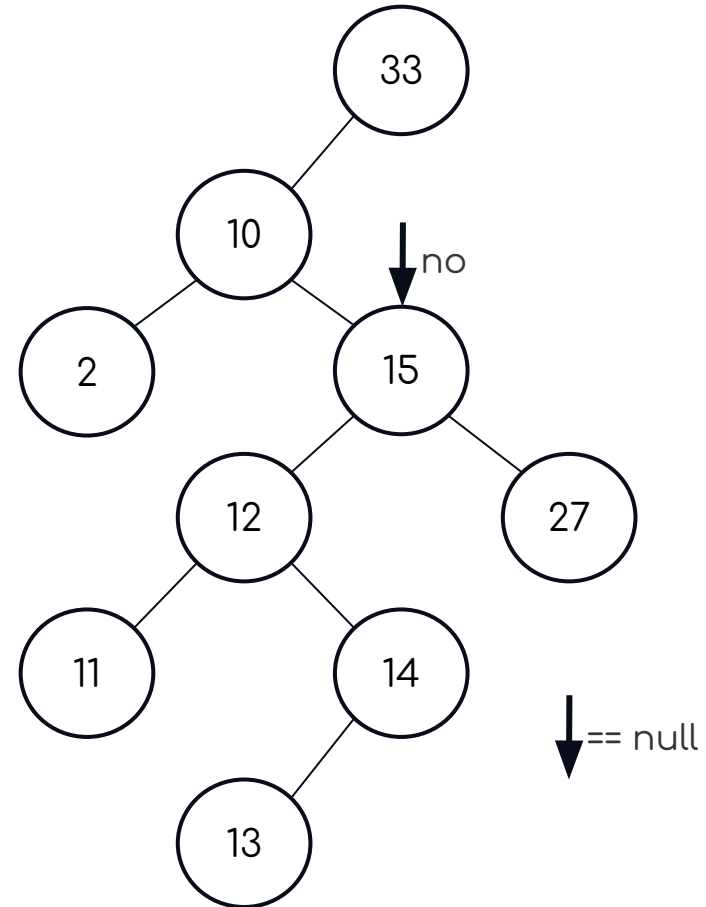
- Exemplo 1: Remover 15.





# Remover

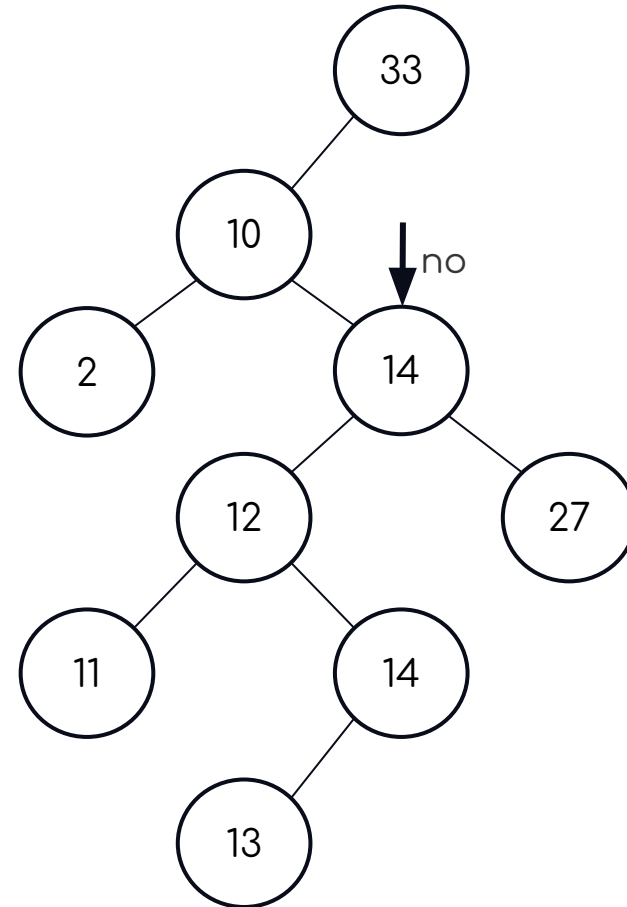
- Exemplo 1: Remover 15.





# Remover

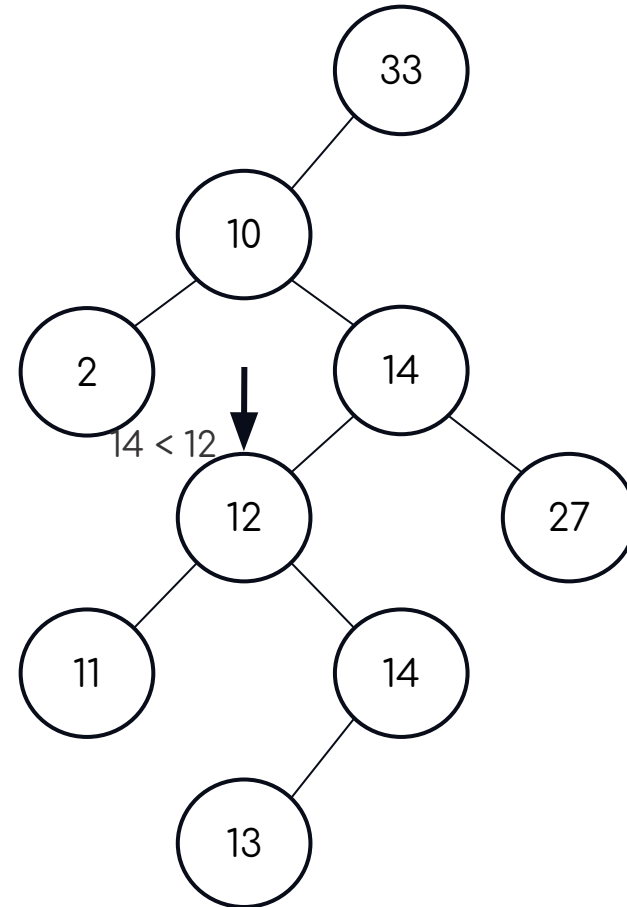
- Exemplo 1: Remover 15.





# Remover

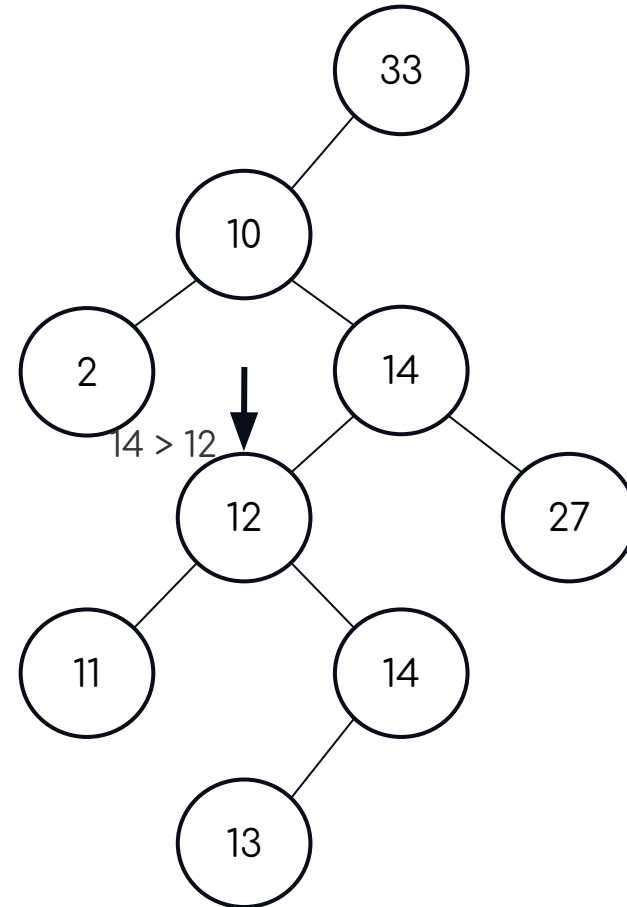
- Exemplo 1: Remover 14.





# Remover

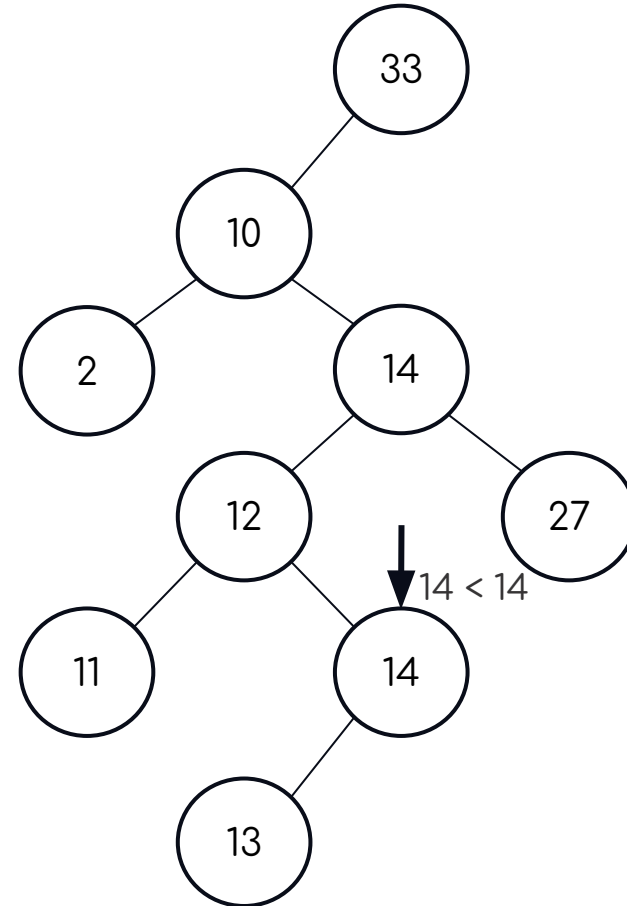
- Exemplo 1: Remover 14.





# Remover

- Exemplo 1: Remover 14.

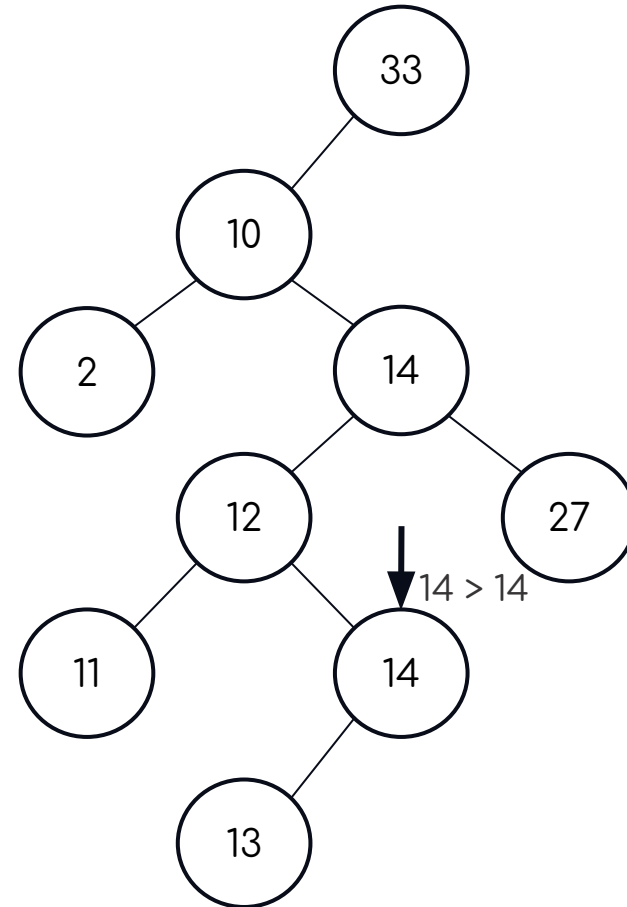






# Remover

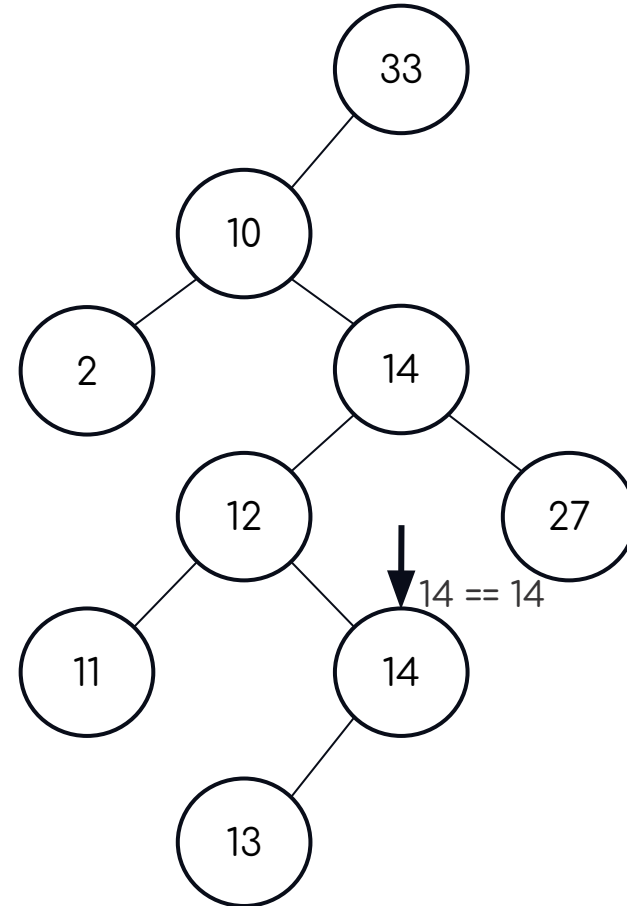
- Exemplo 1: Remover 14.





# Remover

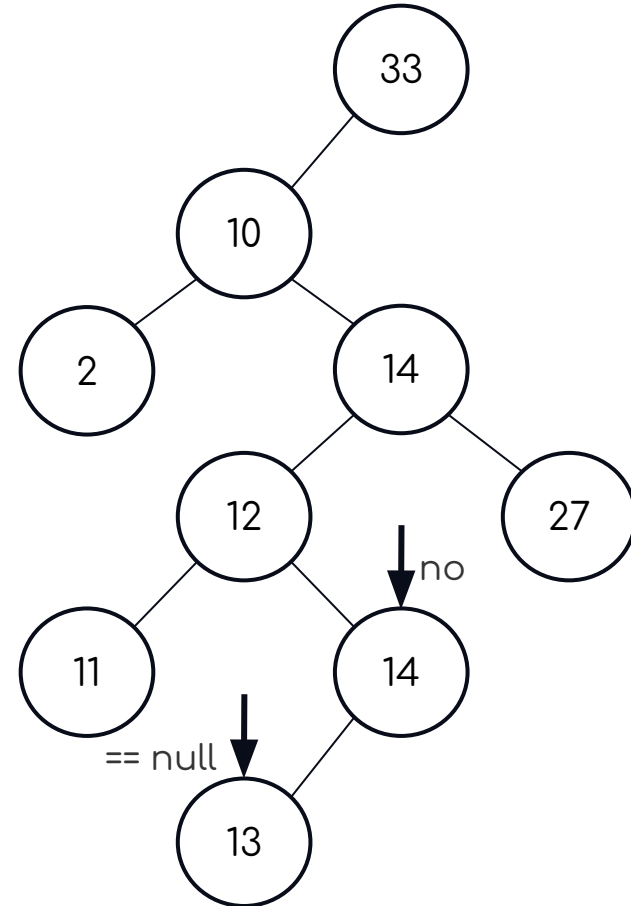
- Exemplo 1: Remover 14.





# Remover

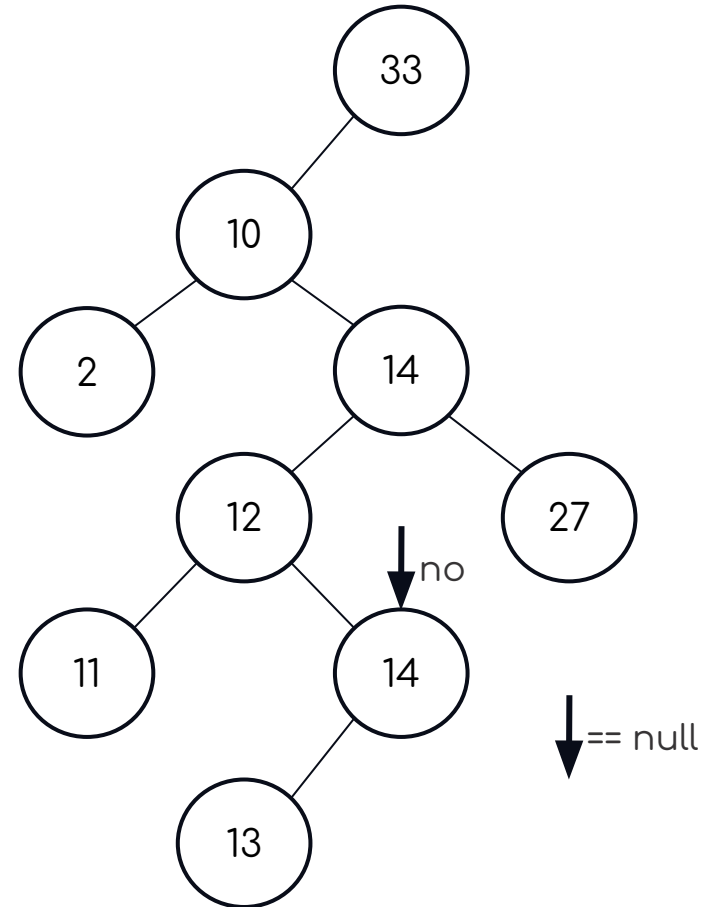
- Exemplo 1: Remover 14.





# Remover

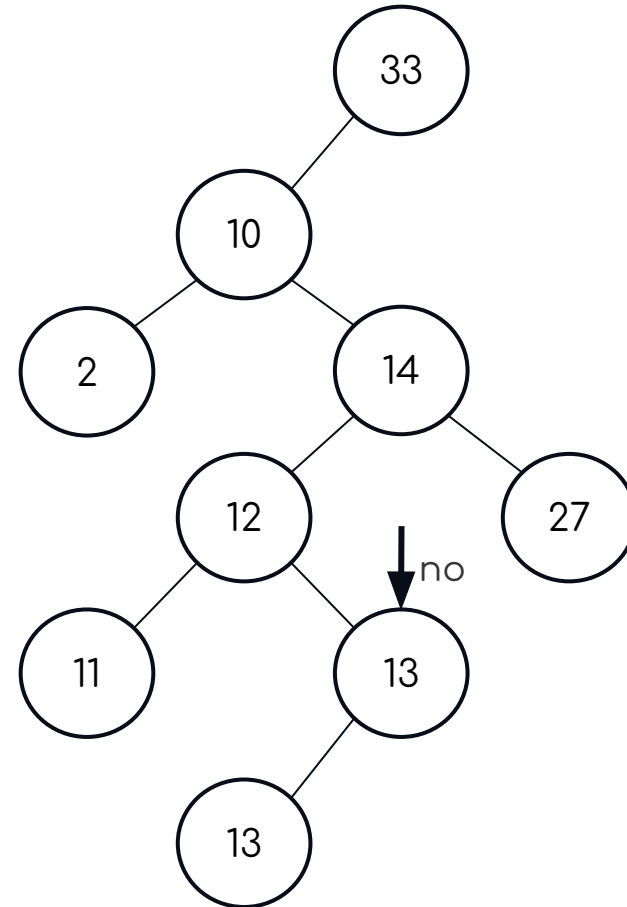
- Exemplo 1: Remover 14.





# Remover

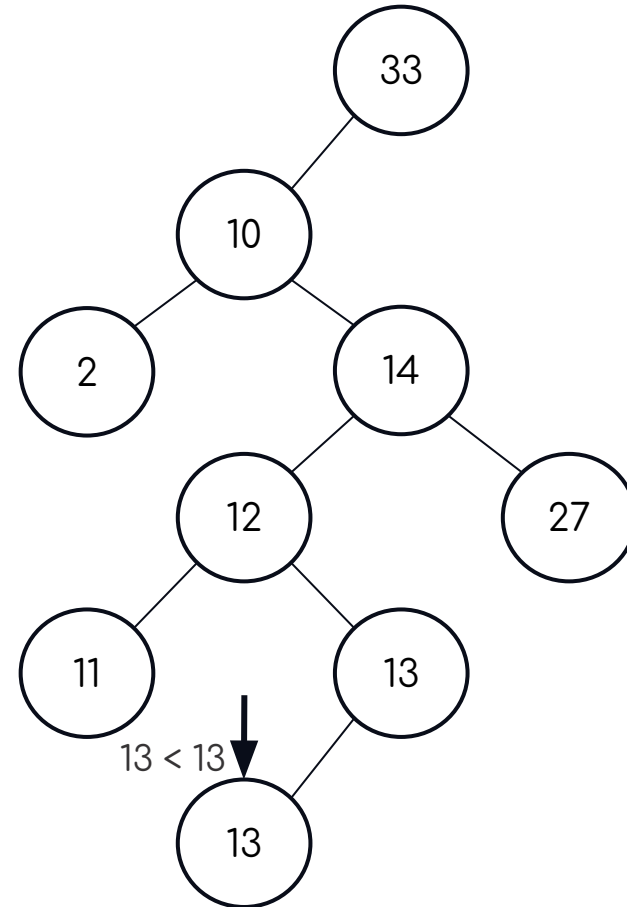
- Exemplo 1: Remover 14.





# Remover

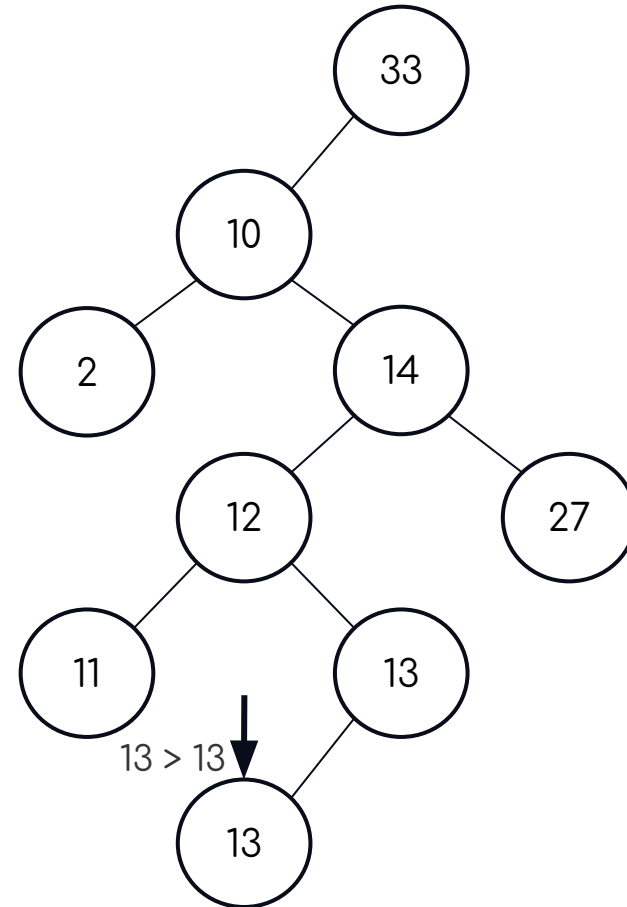
- Exemplo 1: Remover 13.





# Remover

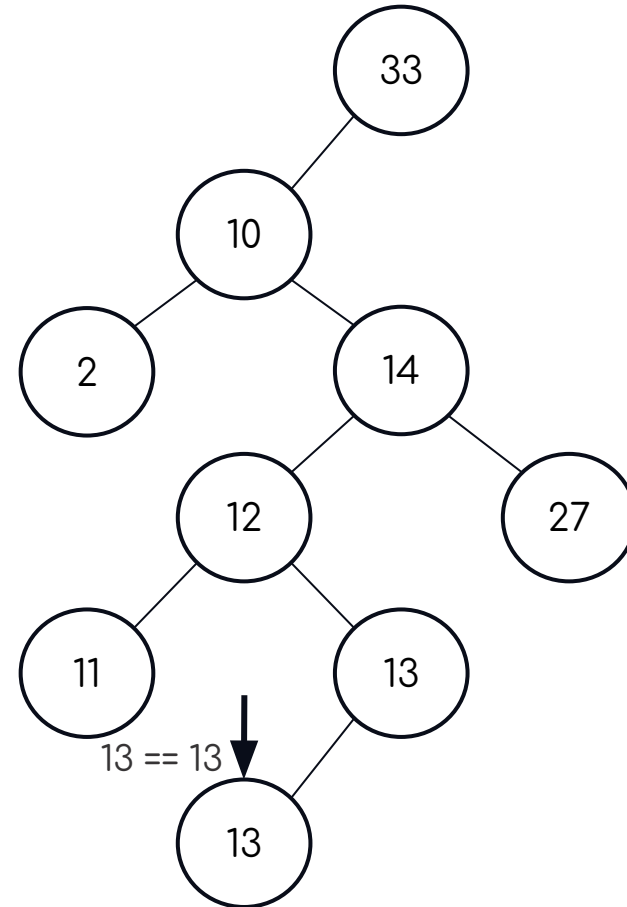
- Exemplo 1: Remover 13.





# Remover

- Exemplo 1: Remover 13.

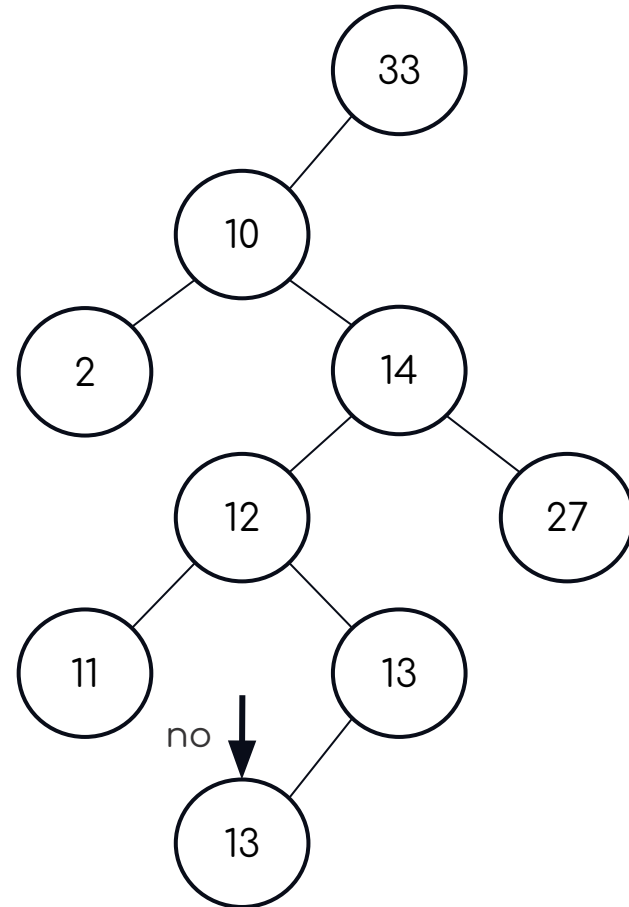






# Remover

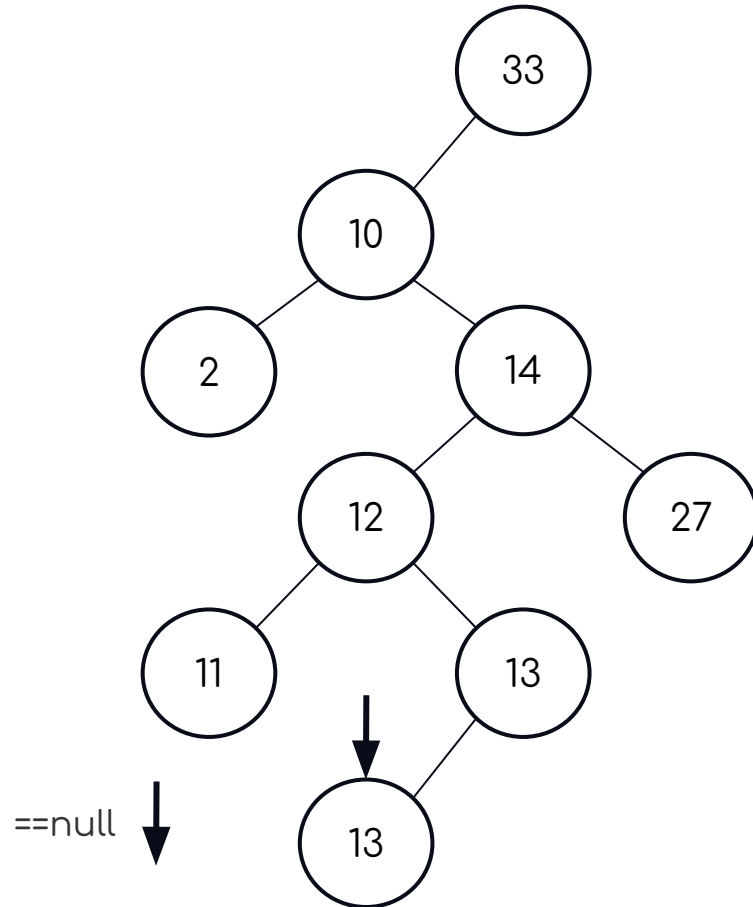
- Exemplo 1: Remover 13.





# Remover

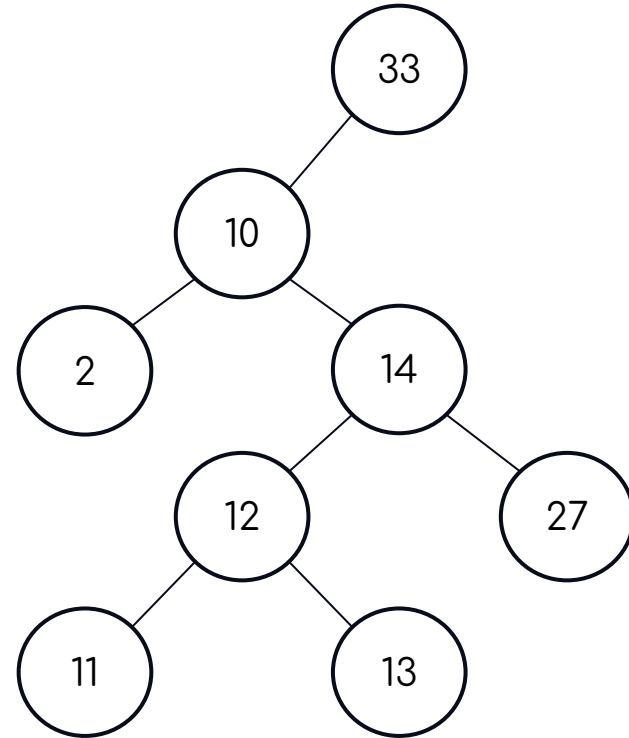
- Exemplo 1: Remover 13.





# Remover

- Exemplo 1: Remover 14.





# Remover - Algoritmo Iterativo

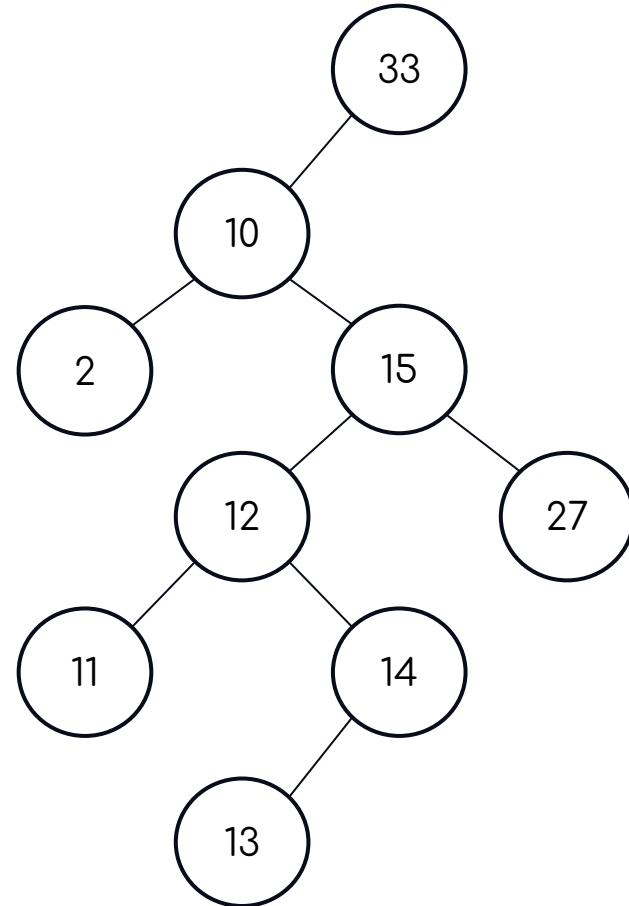
---

- Algoritmo auxiliar
  - Busco o nó na árvore e armazeno o nó e seu pai, caso não encontrar armazeno nulo.
- Se o nó buscado for nulo:
  - Remover nada.
- Senão:
  - Se ele só tiver um filho:
    - Substituo o filho do pai pelo filho do nó a ser removido.
  - Senão:
    - Busco o nó mais à direita da subárvore à esquerda do nó a ser removido e também armazenar o pai.
    - Se o pai do nó que eu vou substituir não é o nó a ser removido.
      - substituto para a o nó à esquerda.
    - Senão:
      - Substitui o filho do pai do nó substituto.
  - Se o nó a ser removido é a raiz:
    - Liberar a raiz e retornar o nó promovido.
  - Senão:
    - O pai do nó a ser removido aponta para o nó substituto.



# Remover

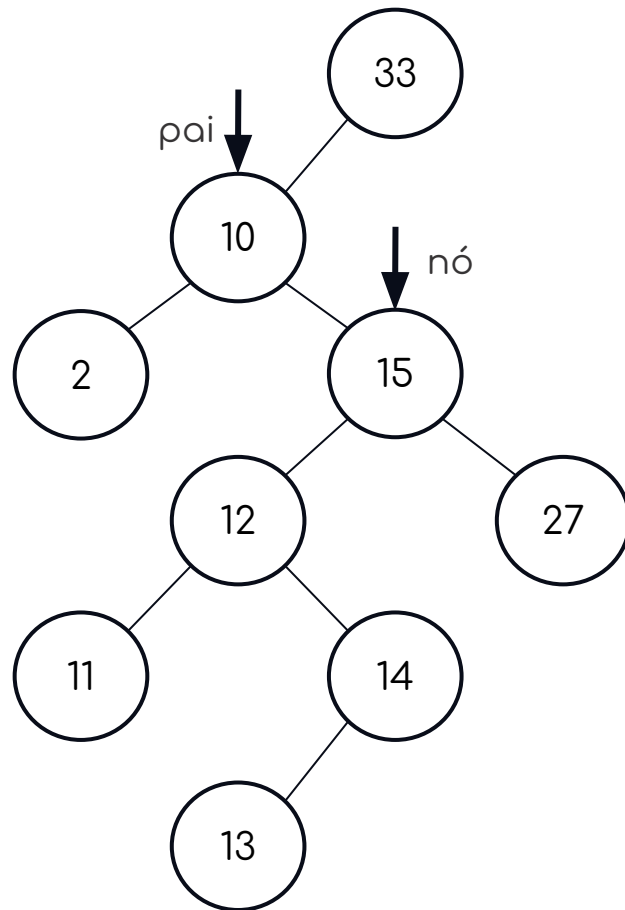
- Exemplo 1: Remover 15.





# Remover

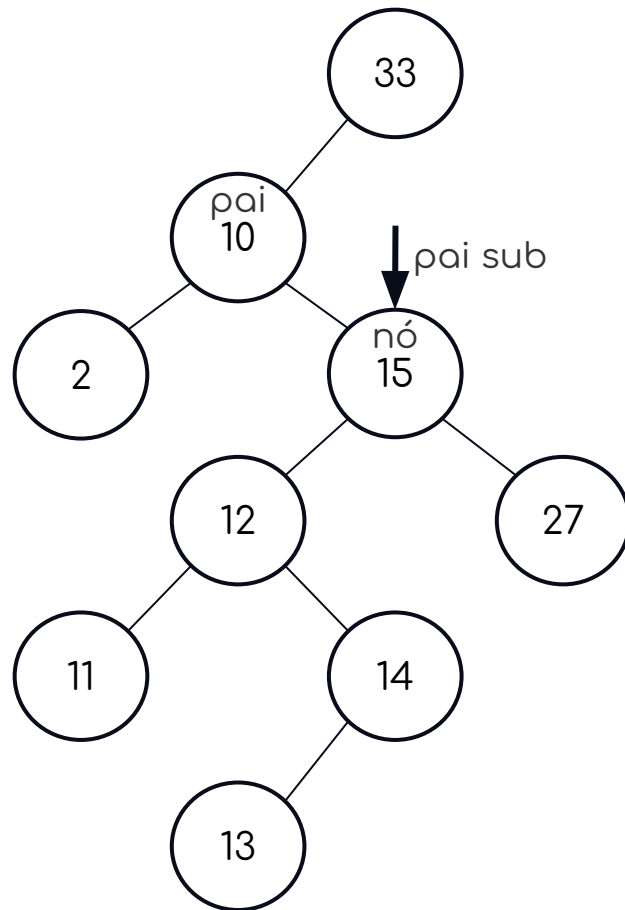
- Exemplo 1: Remover 15.





# Remover

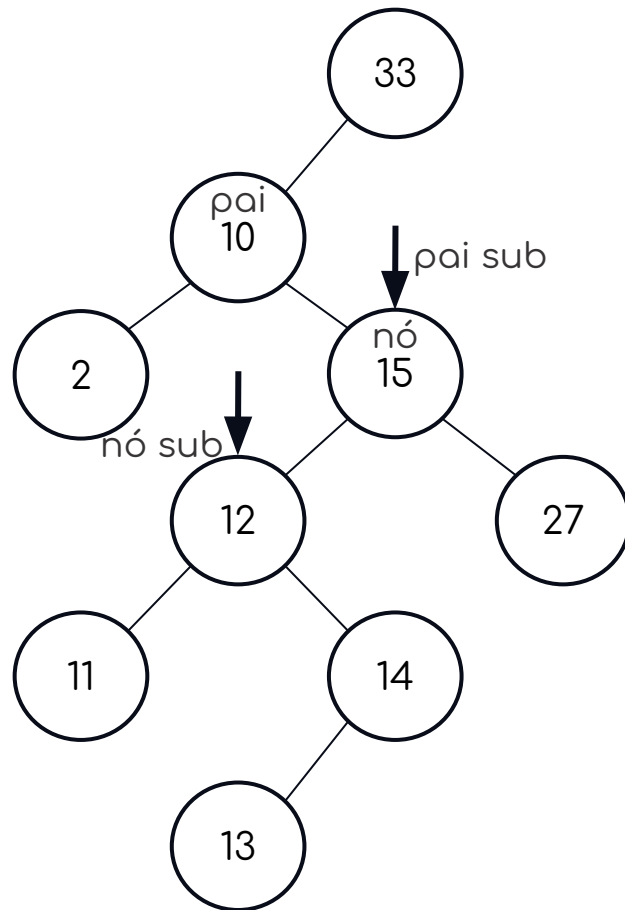
- Exemplo 1: Remover 15.





# Remover

- Exemplo 1: Remover 15.

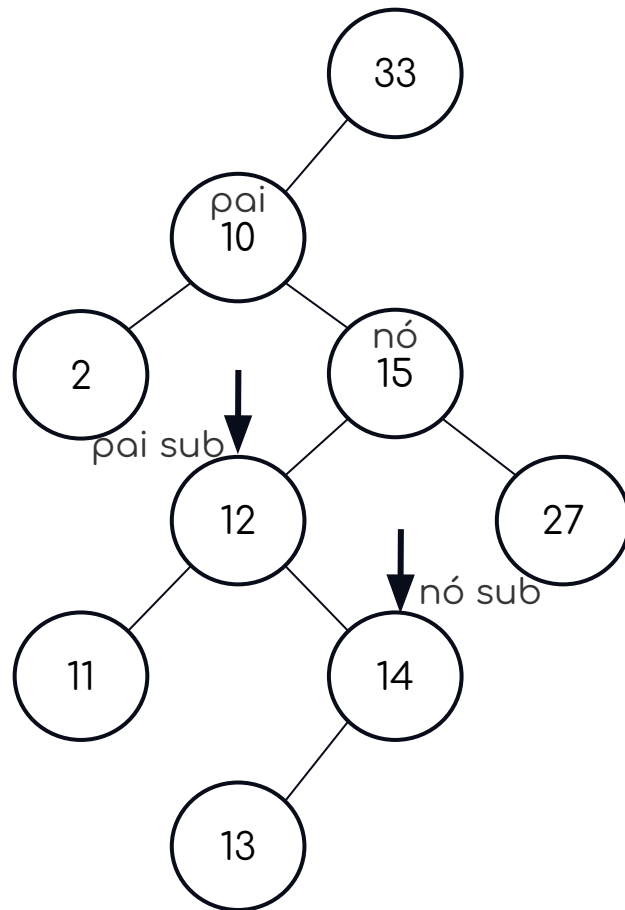






# Remover

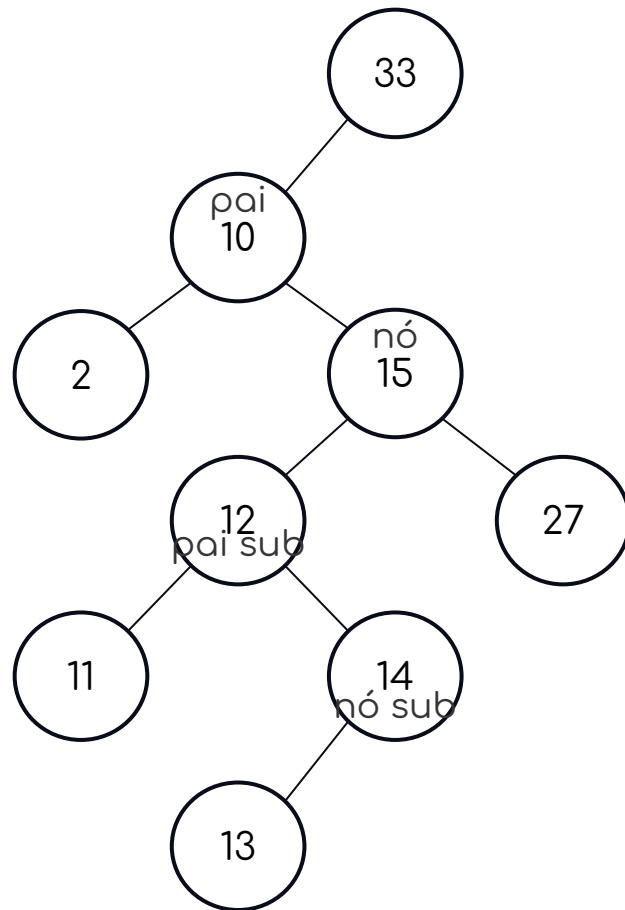
- Exemplo 1: Remover 15.





# Remover

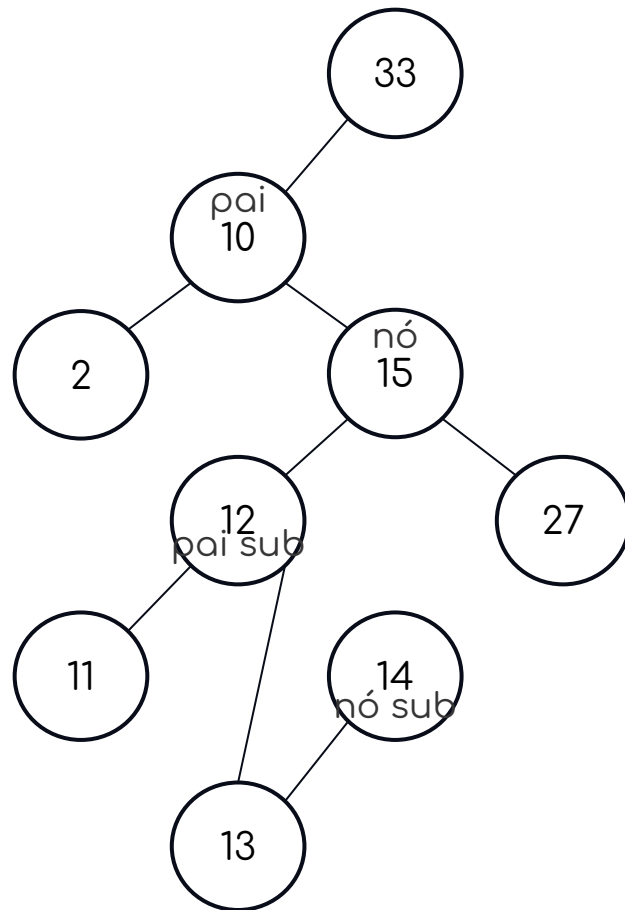
- Exemplo 1: Remover 15.





# Remover

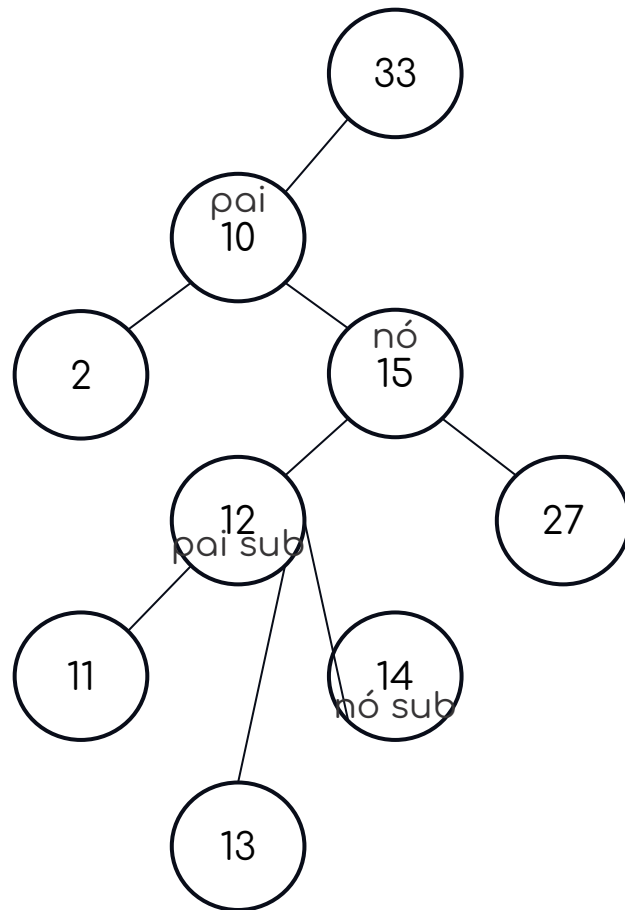
- Exemplo 1: Remover 15.





# Remover

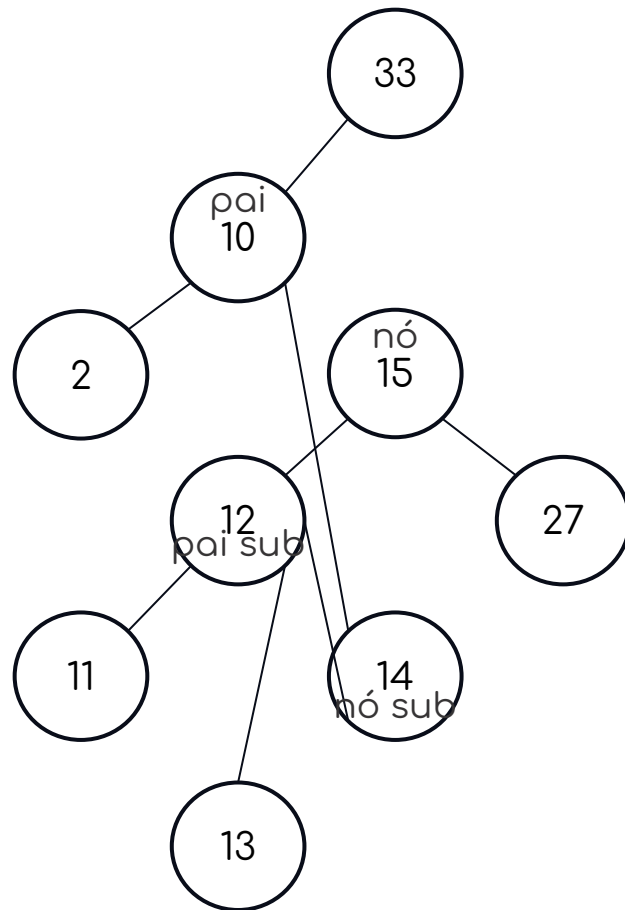
- Exemplo 1: Remover 15.





# Remover

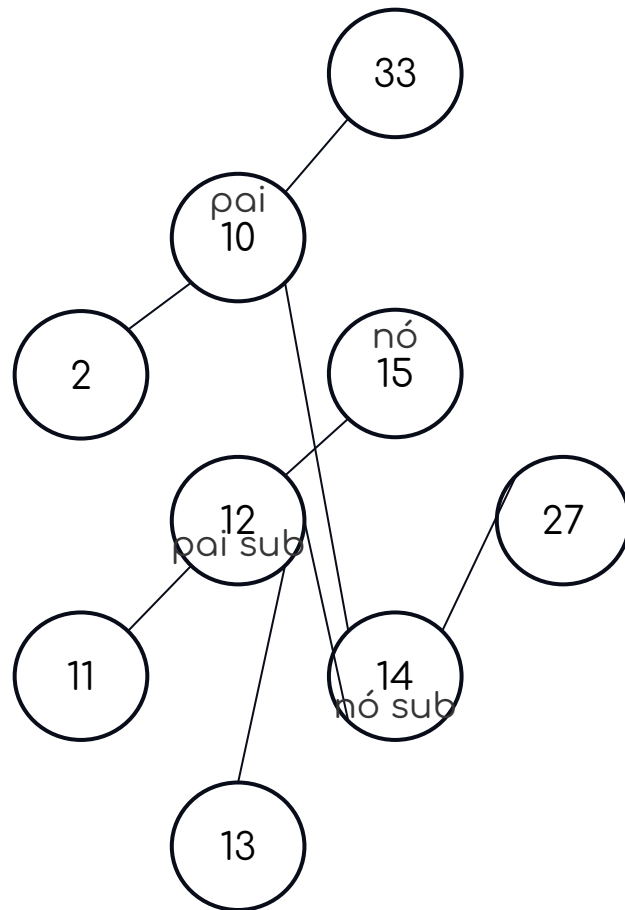
- Exemplo 1: Remover 15.





# Remover

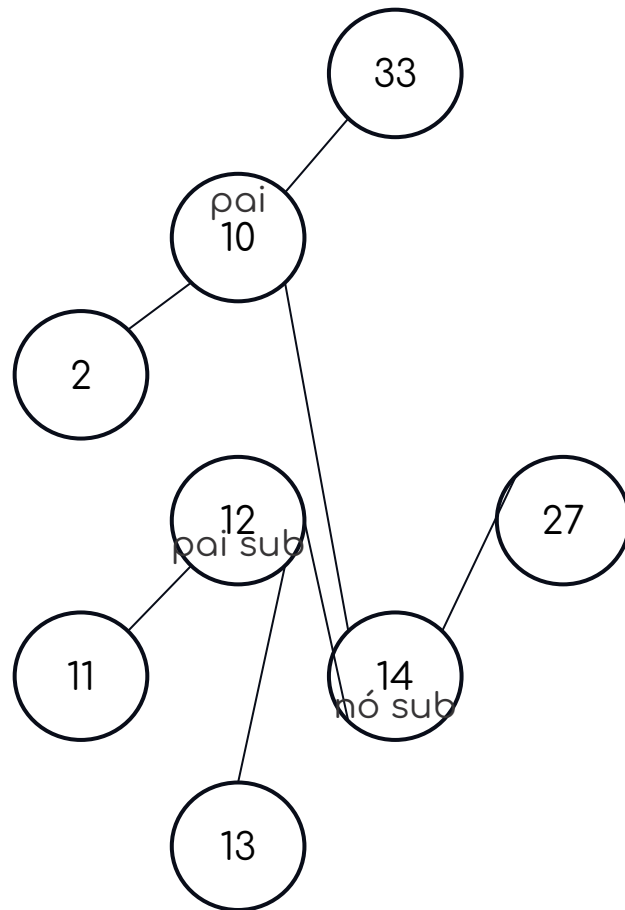
- Exemplo 1: Remover 15.





# Remover

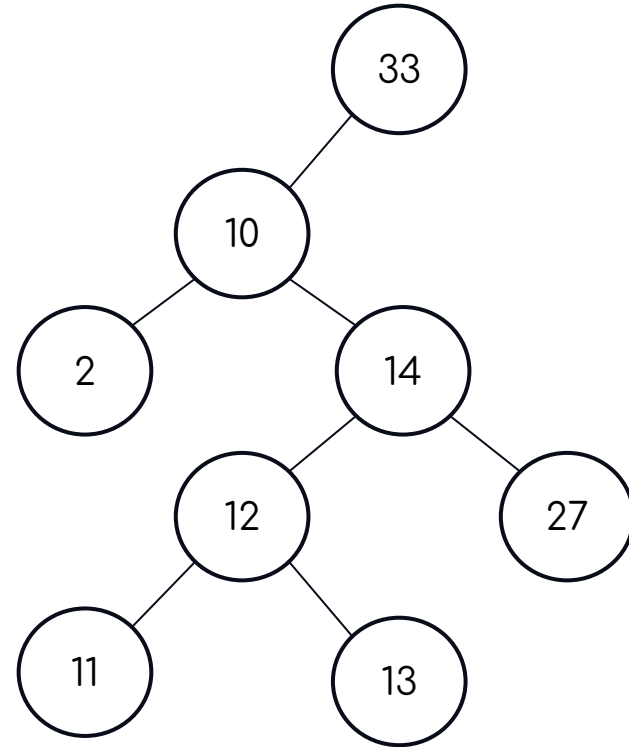
- Exemplo 1: Remover 15.





# Remover

- Exemplo 1: Remover 15.







## Desafio

- Implemente o algoritmo para remoção de um elemento da árvore, de forma não recursiva.



# Desafios





## Desafio

Considere uma árvore binária de busca construída com os valores: 40, 20, 60, 10, 30, 50, 70

Agora remova, nessa ordem, os seguintes valores: 40, 30, 70.

- Desenhe como a árvore ficou ao final.



## Desafio

Construa uma árvore inserindo os seguintes valores na ordem dada: 50, 30, 70, 20, 40, 60, 80.

- Represente a árvore graficamente.
- Faça os percursos pré-ordem, em-ordem, e pós-ordem.



## Desafio

Com base na árvore do anterior, remova os seguintes nós (todos são folhas):

- 20
- 40
- 80
- Após cada remoção, desenhe a nova árvore.



## Desafio

Construa uma árvore com os valores: 50, 30, 70, 60, 80, 65.

- Remova o nó 60.
- Remova o nó 70.
- Mostre a árvore após cada operação.



## Desafio

Usando a árvore construída anteriormente, remova:

- o nó 30
- o nó 50 (a raiz)
- Mostre a árvore resultante após cada remoção.



## Desafio

Dada a sequência de inserção: 15, 10, 20, 8, 12, 17, 25, 6, 11, 13

- Construa a árvore.
- Remova os nós, na ordem: 10, 15, 20, usando a árvore da direita do nó a ser removido.
- Após cada remoção, desenhe a árvore.





## Desafio

Dada a sequência de inserção: 15, 10, 20, 8, 12, 17, 25, 6, 11, 13

- Construa a árvore.
- Remova os nós, na ordem: 10, 15, 20, usando a árvore da direita do nó a ser removido.
- Após cada remoção, desenhe a árvore.