

Árvore binária de busca

Prof. Paulo Henrique Pisani

março/2022

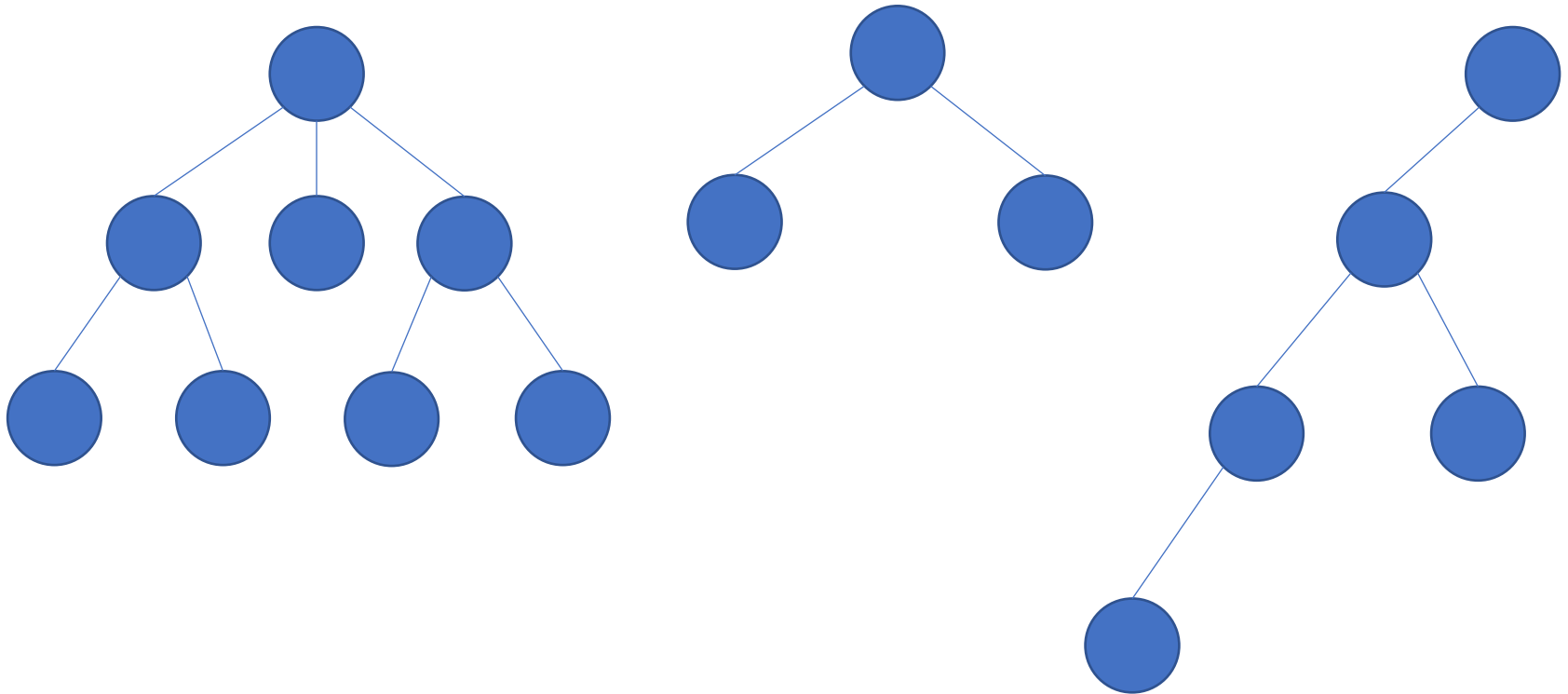
Tópicos

- Algumas definições;
- Árvore binária de busca;
- Operações;
- Complexidade das operações;
- Percurso em árvore.

Algumas definições

Árvore

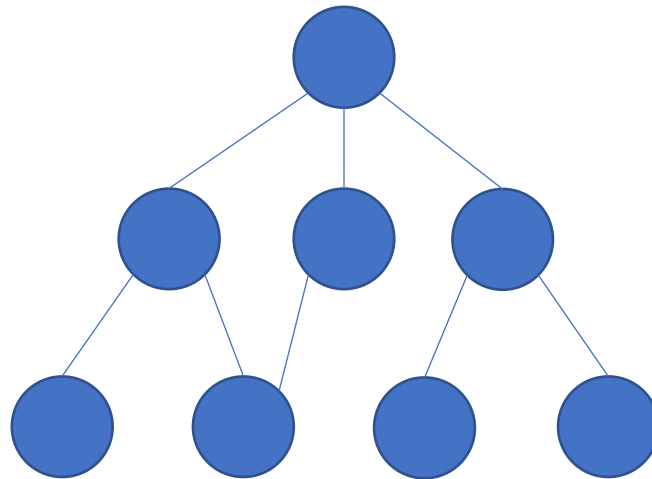
- Exemplos de árvores:



Árvore

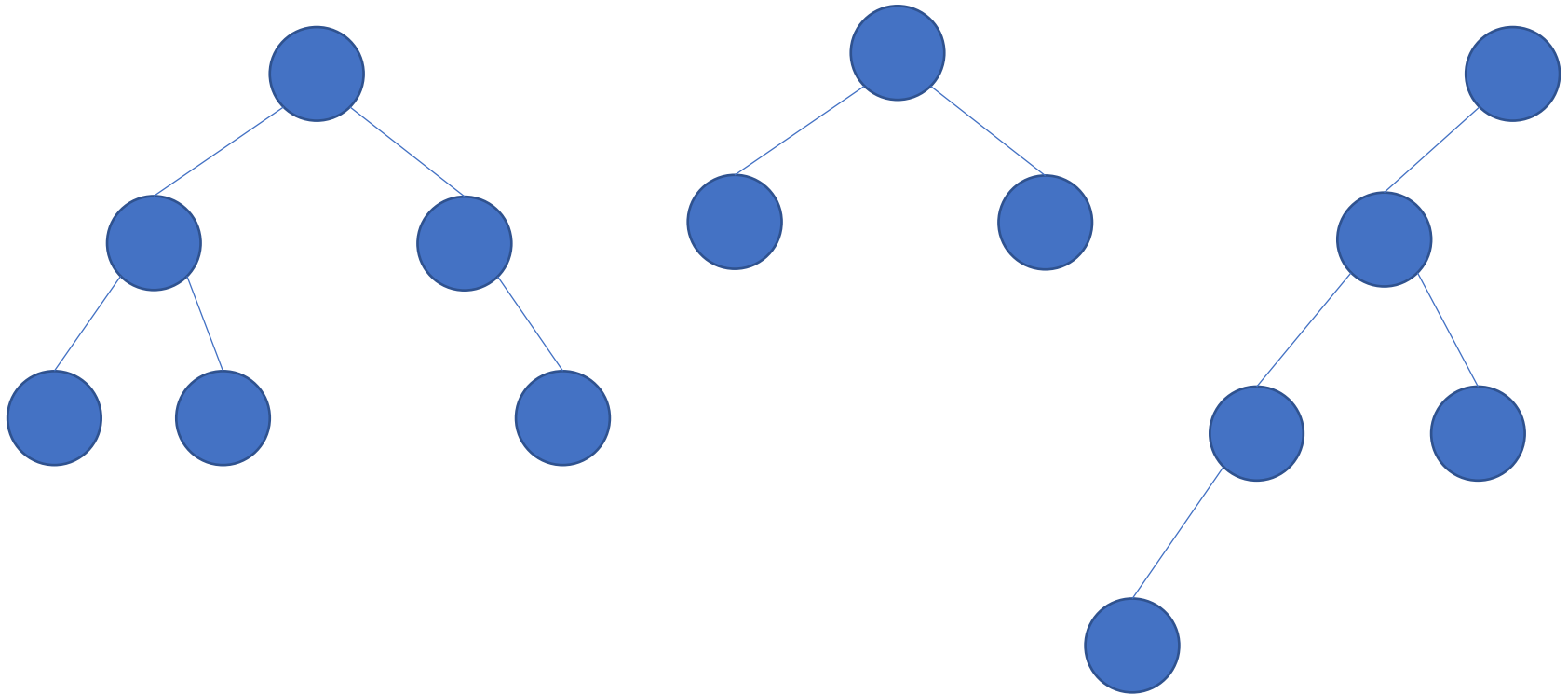
- Uma árvore é definida forma recursiva. Uma árvore T é um conjunto finito de nós em que:
 - $T = \text{vazio} \rightarrow$ é a árvore vazia;
 - Existe um nó chamado de raiz de T ; Os demais nós são um conjunto vazio ou são divididos em $m \geq 1$ conjuntos disjuntos não vazios chamados de subárvores. Cada subárvore é uma árvore.

Não é árvore



Árvore binária

- Exemplos de árvores binárias:



Árvore binária

- Uma árvore binária T é um conjunto finito de nós em que:
 - $T = \text{vazio} \rightarrow$ é a árvore vazia;
 - Existe um nó chamado de raiz de T ; Os demais nós são divididos em dois conjuntos disjuntos, chamados de subárvore esquerda e subárvore direita. Ambas são também árvores binárias.

Árvore binária

- Uma árvore binária T é um conjunto finito de nós em que:
 - $T = \text{vazio} \rightarrow$ é a árvore vazia;
 - Existe um nó chamado de raiz de T ; Os demais nós são divididos em dois conjuntos disjuntos, chamados de subárvore esquerda e subárvore direita. Ambas são também árvores binárias.

Ou seja, árvore binária é uma árvore em que todos os nós apontam para duas subárvores

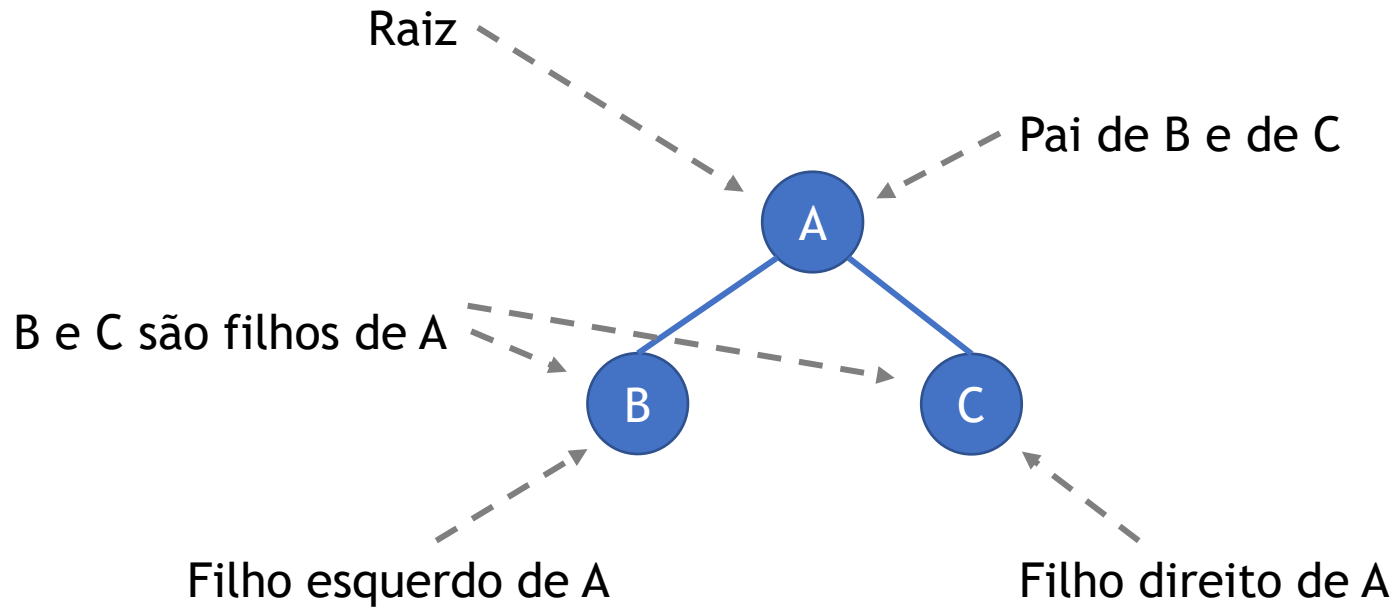
Em que aplicações
podemos usar árvores?

Árvores

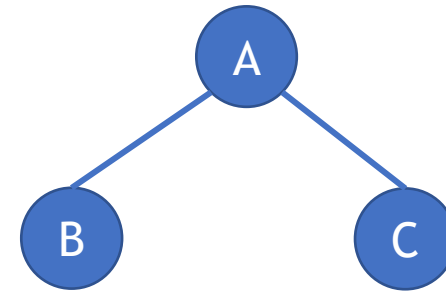
- Quando há alguma forma de ordenar os elementos, a estrutura em árvore permite:
 - **Busca, inserção e remoção** de forma eficiente;
 - Acesso sequencial eficiente.
- Com árvores balanceadas:

| Operação | Listas | Árvores |
|----------|--------|--------------|
| Busca | $O(n)$ | $O(\log(n))$ |
| Inserção | $O(n)$ | $O(\log(n))$ |
| Remoção | $O(n)$ | $O(\log(n))$ |

Algumas definições



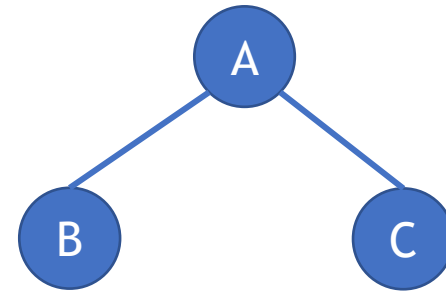
Algumas definições



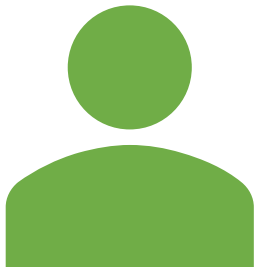
Você acabou de falar que todo nó tem dois filhos na árvore binária! Mas quais são os filhos de B e de C ???



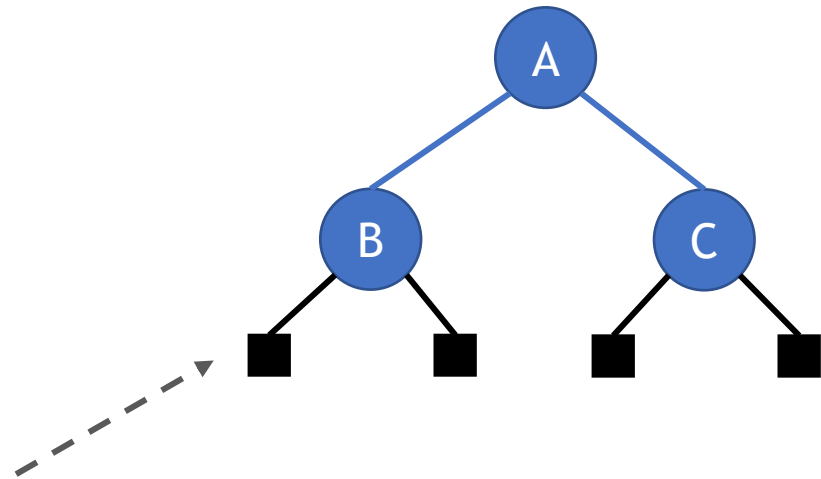
Algumas definições



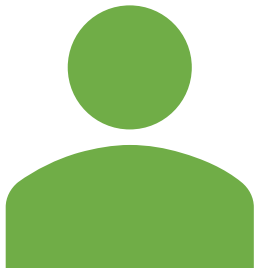
B e C tem dois filhos, mas
são dois filhos nulos!



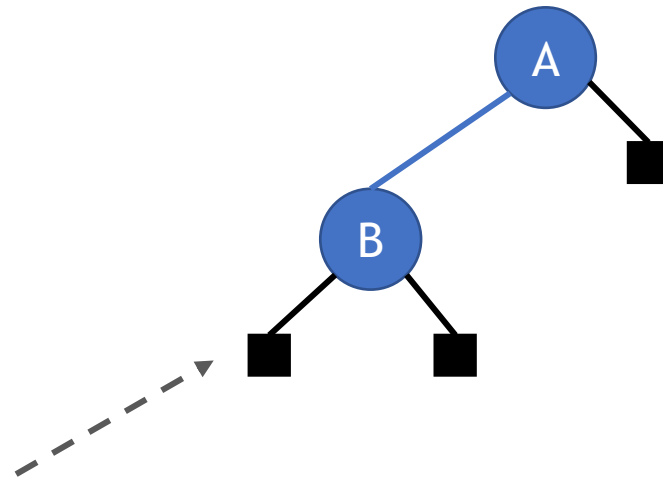
Algumas definições



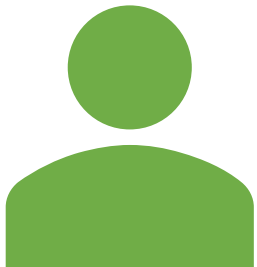
Algumas representações de árvores mostram isso mais explicitamente.



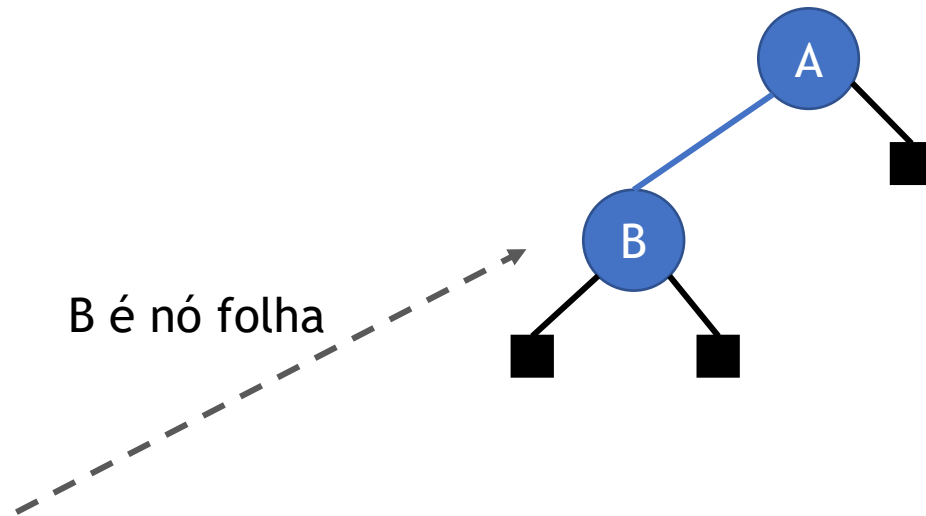
Algumas definições



Algumas representações de árvores mostram isso mais explicitamente.

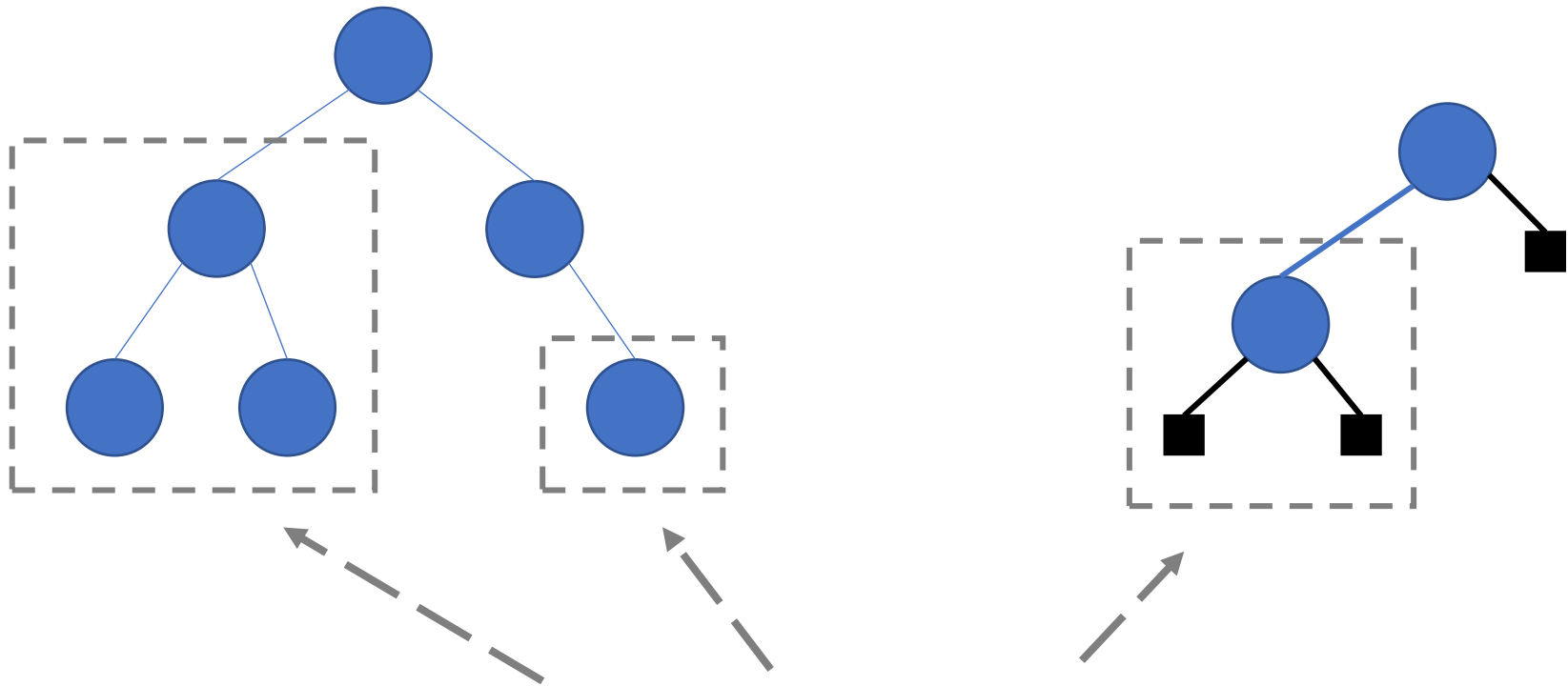


Algumas definições



Nó folha: possui dois filhos nulos (as duas subárvores são nulas)

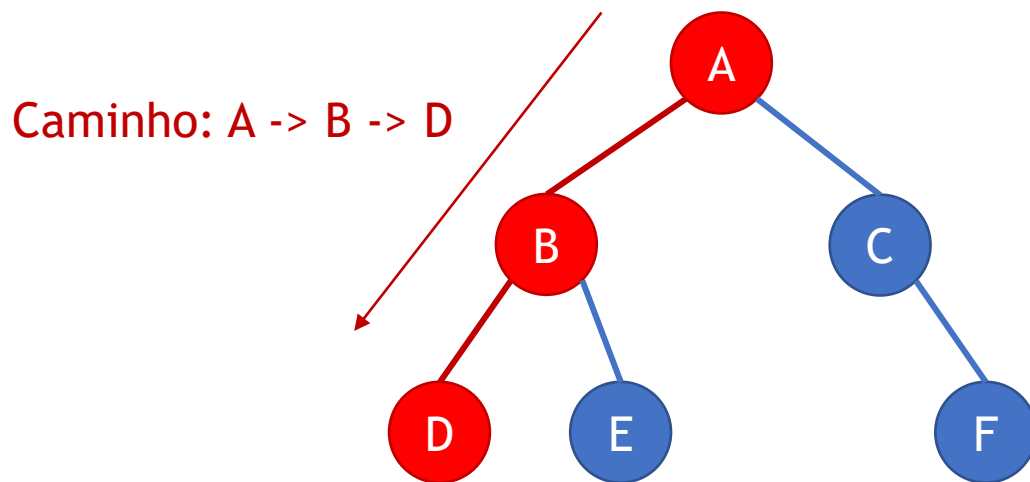
Subárvores



Ejemplos de subárvores

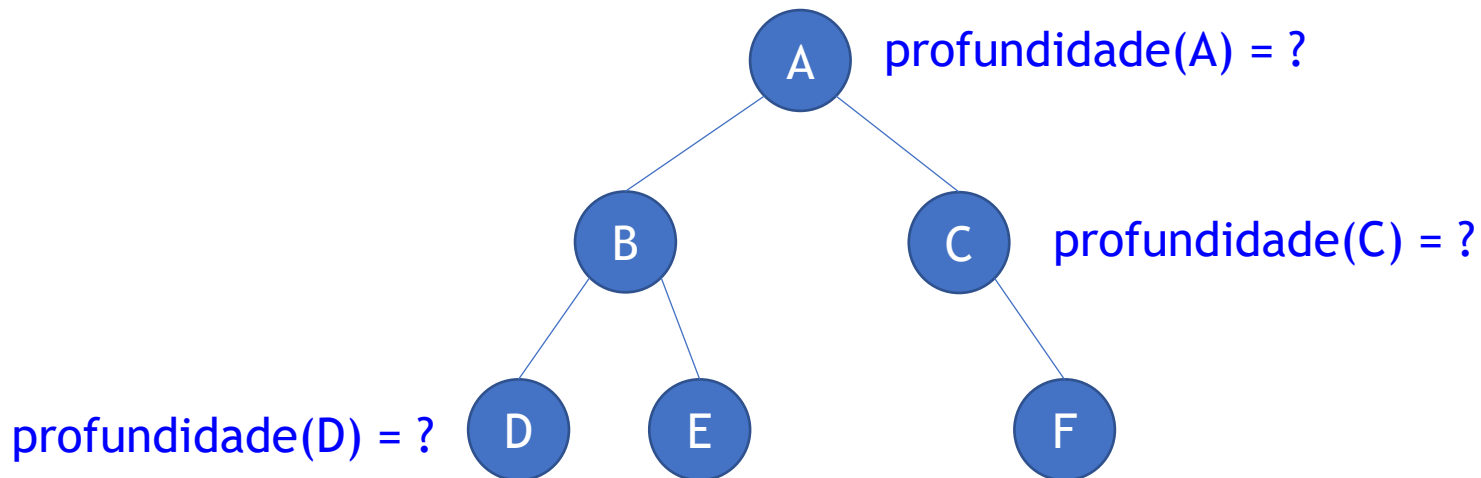
Caminho

- Uma sequência de nós distintos v_1, v_2, \dots, v_k em que há a relação “é pai de” entre nós consecutivos é denominada **caminho** na árvore.



Profundidade de um nó

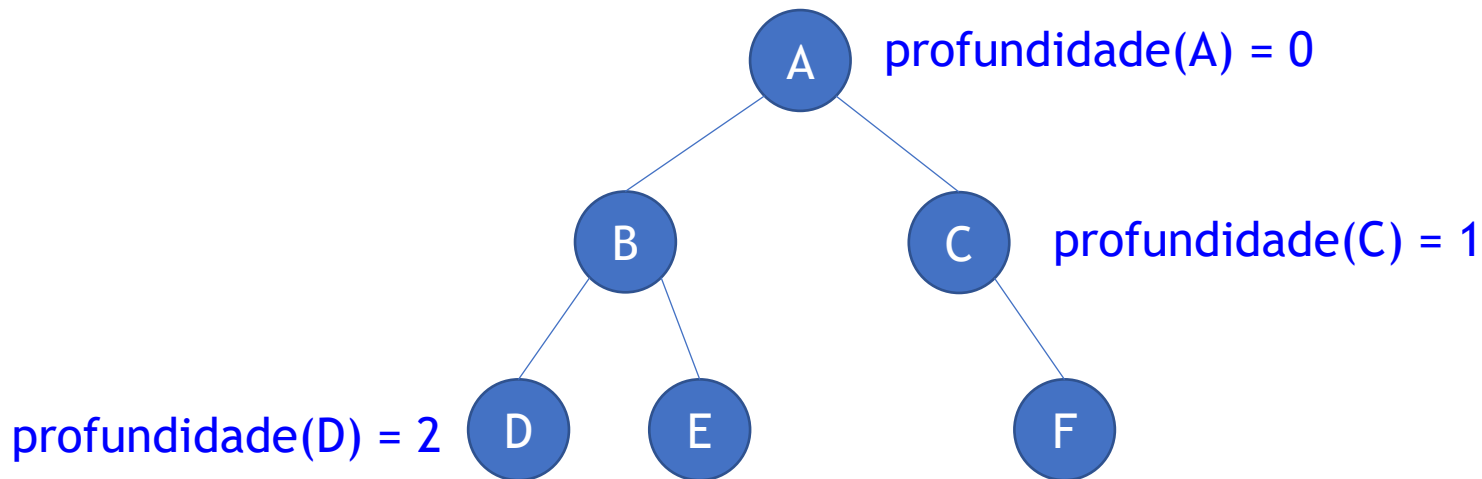
- A **profundidade de um nó** é definida como o número de arestas entre o caminho da raiz até o nó.



Qual a profundidade dos nós A, C e D?

Profundidade de um nó

- A **profundidade de um nó** é definida como o número de arestas entre o caminho da raiz até o nó.

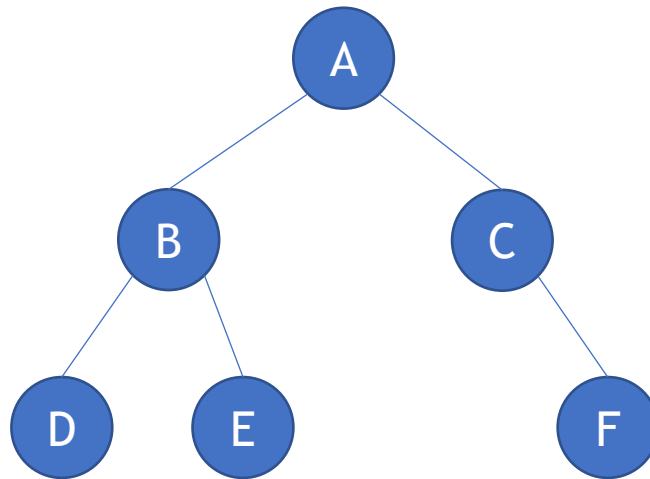


Nível de uma árvore

- Um nível de uma árvore são todos os nós que estão na mesma profundidade.

Altura de uma árvore

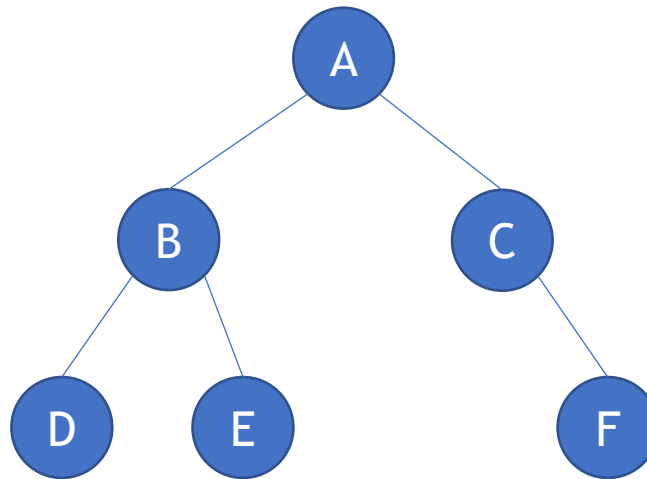
- A **altura de uma árvore** é a profundidade máxima que um nó pode atingir na árvore.



Qual a altura desta árvore?

Altura de uma árvore

- A **altura de uma árvore** é a profundidade máxima que um nó pode atingir na árvore.

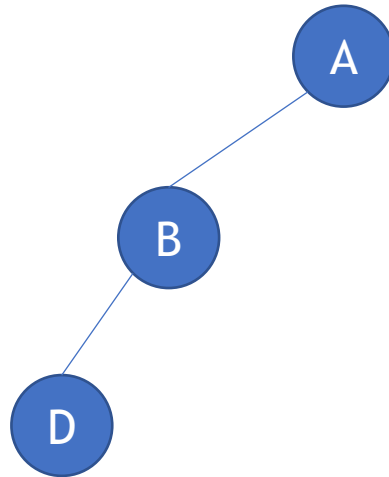


Qual a altura desta árvore?

h = 2

Altura de uma árvore

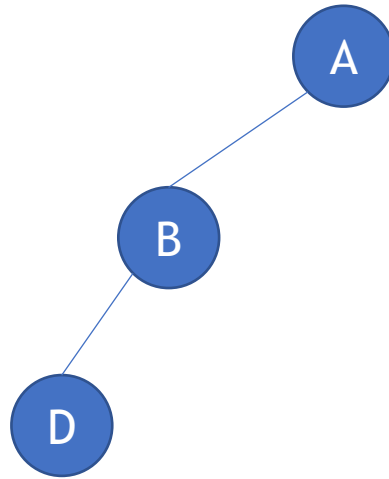
- A **altura de uma árvore** é a profundidade máxima que um nó pode atingir na árvore.



Qual a altura desta árvore?

Altura de uma árvore

- A **altura de uma árvore** é a profundidade máxima que um nó pode atingir na árvore.



Qual a altura desta árvore?

$h = 2$

Altura de uma árvore

- A **altura de uma árvore** é a profundidade máxima que um nó pode atingir na árvore.



Qual a altura desta árvore?

Altura de uma árvore

- A **altura de uma árvore** é a profundidade máxima que um nó pode atingir na árvore.



Qual a altura desta árvore?

$h = 0$

Classificações

- **Árvore estritamente binária:** todos os nós tem 0 ou 2 filhos;
- **Árvore binária completa:** todos os nós folha estão no último ou no penúltimo nível;
- **Árvore cheia:** todos os nós folha estão no último nível.

Árvore binária de busca

Árvore binária de **busca**

- Uma **árvore binária de busca (ABB)** é uma árvore binária que possui a seguinte **propriedade**:

Para todo nó v da árvore:

- Se x é um nó na subárvore esquerda de v , então:
 $x.chave < v.chave$
- Se x é um nó na subárvore direita de v , então:
 $v.chave < x.chave$

Árvore binária de **busca**

- Uma **árvore binária de busca (ABB)** é uma árvore binária que possui a seguinte **propriedade**:

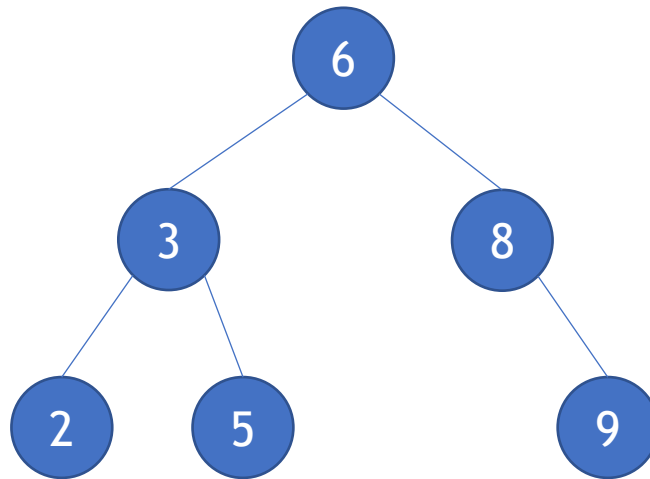
Para todo nó v da árvore:

- Se x é um nó na subárvore esquerda de v , então:
 $x.chave \leq v.chave$
- Se x é um nó na subárvore direita de v , então:
 $v.chave \leq x.chave$

Definição na página 209: CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. Algoritmos: teoria e prática. 3ª edição. Rio de Janeiro, RJ: Elsevier, 2012.

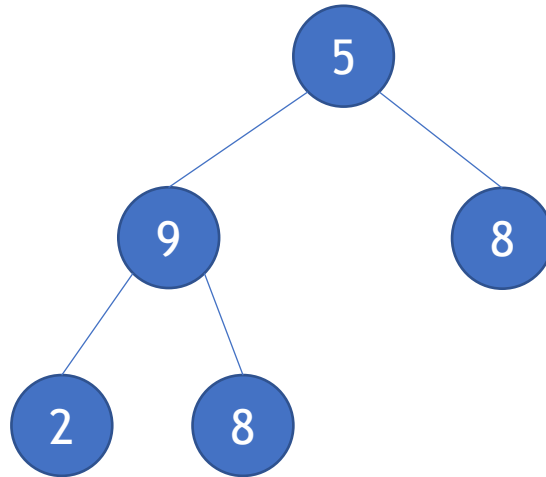
Árvore binária de **busca**

- É árvore binária de busca?



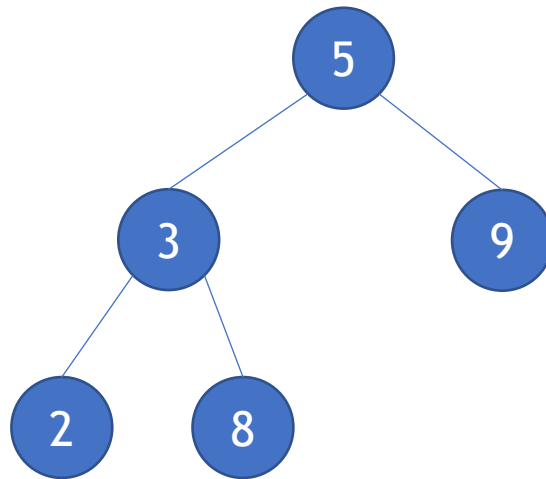
Árvore binária de **busca**

- É árvore binária de busca?



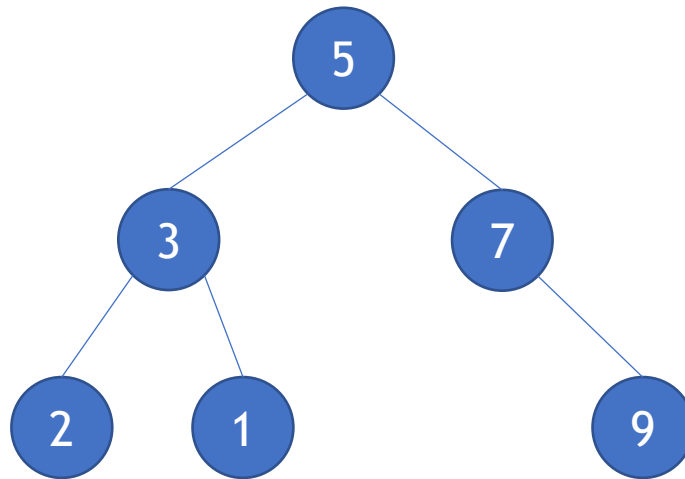
Árvore binária de **busca**

- É árvore binária de busca?



Árvore binária de **busca**

- É árvore binária de busca?

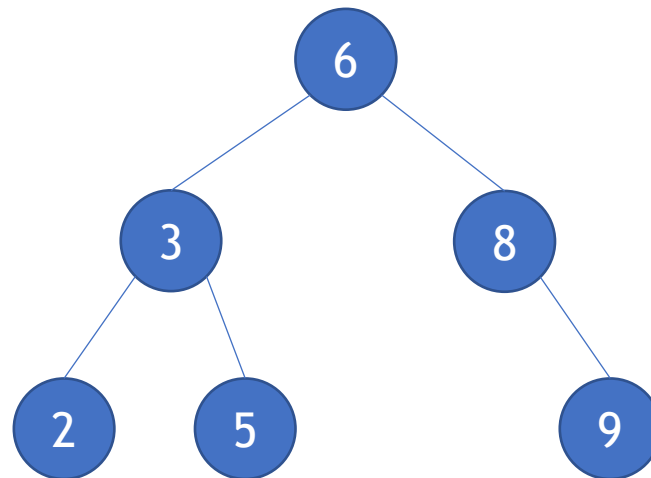


Árvore binária de **busca**

- Operações:
 - Busca
 - Inserção
 - Remoção
- Operações adicionais:
 - Sucessor/Antecessor
 - Primeiro/Último

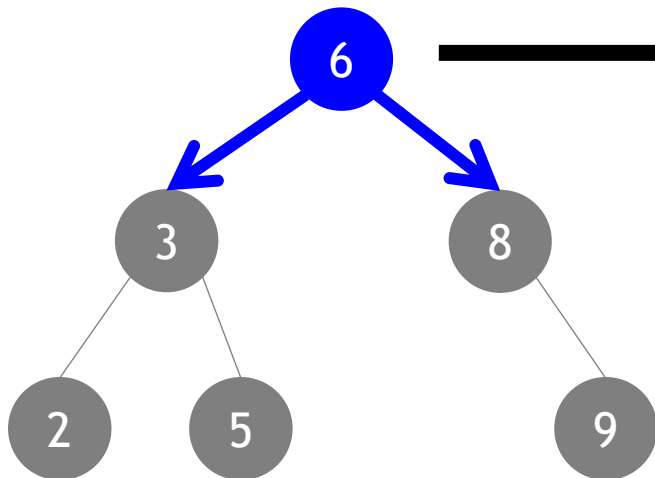
Árvore binária de **busca**

Ok, e como seria a implementação em C?



Árvore binária de **busca**

Ok, e como seria a implementação em C?



→

```
typedef struct NoArvore NoArvore;  
struct NoArvore {  
    int chave;  
    NoArvore *esq, *dir;  
};
```

Operações

Busca, inserção, remoção

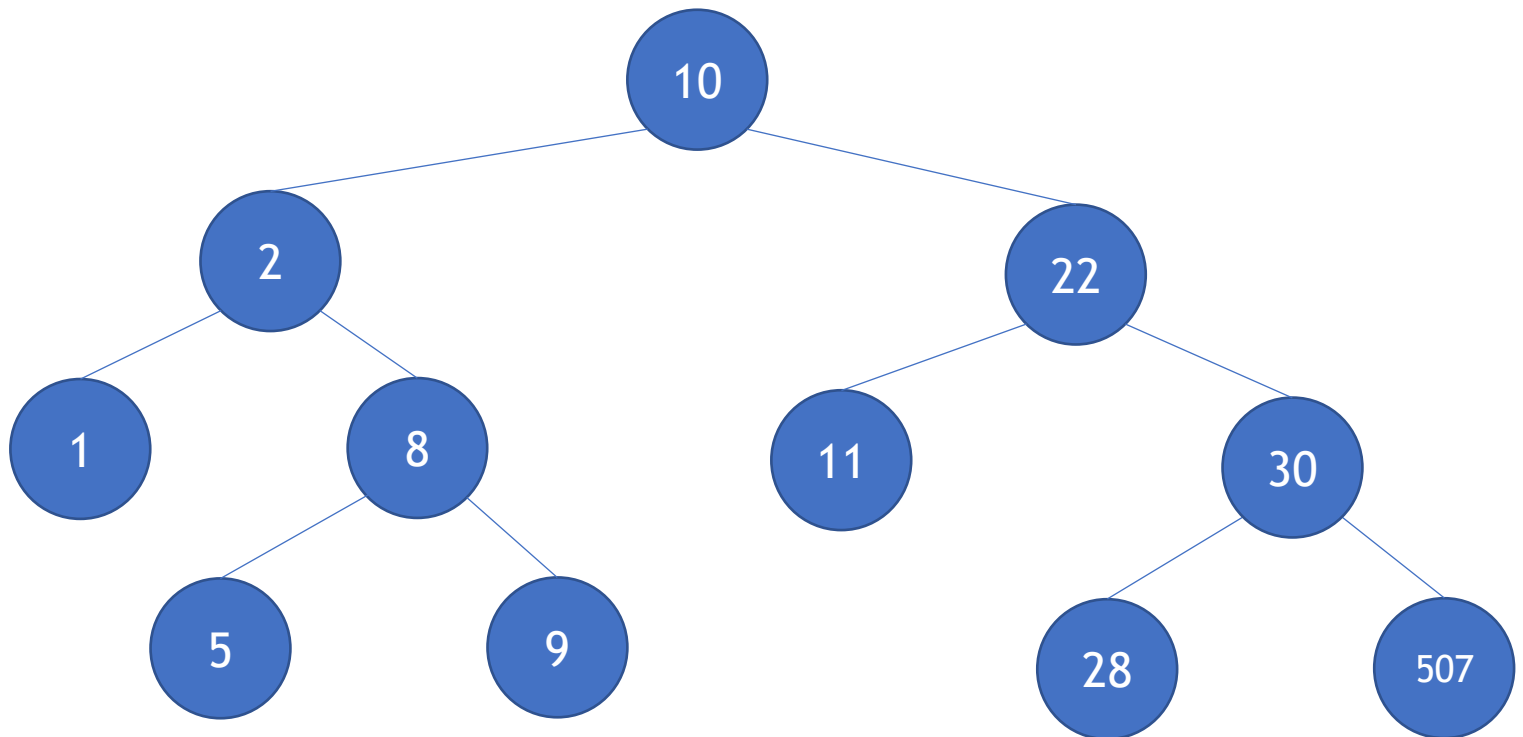
Busca em ABB

- Valor a ser encontrado: x
 - Começa a busca na raiz
1. Se a chave do nó atual é igual a x , encerra a busca (chave encontrada);
 2. Senão verifica se x é menor que a chave do nó atual. Neste caso, continua a busca na subárvore à esquerda; Caso contrário, continua a busca na subárvore à direita; Volta ao passo 1.

O processo segue até que x seja encontrado, ou se chegar a um nó vazio (caso em que a busca encerra por não achar um nó com chave x).

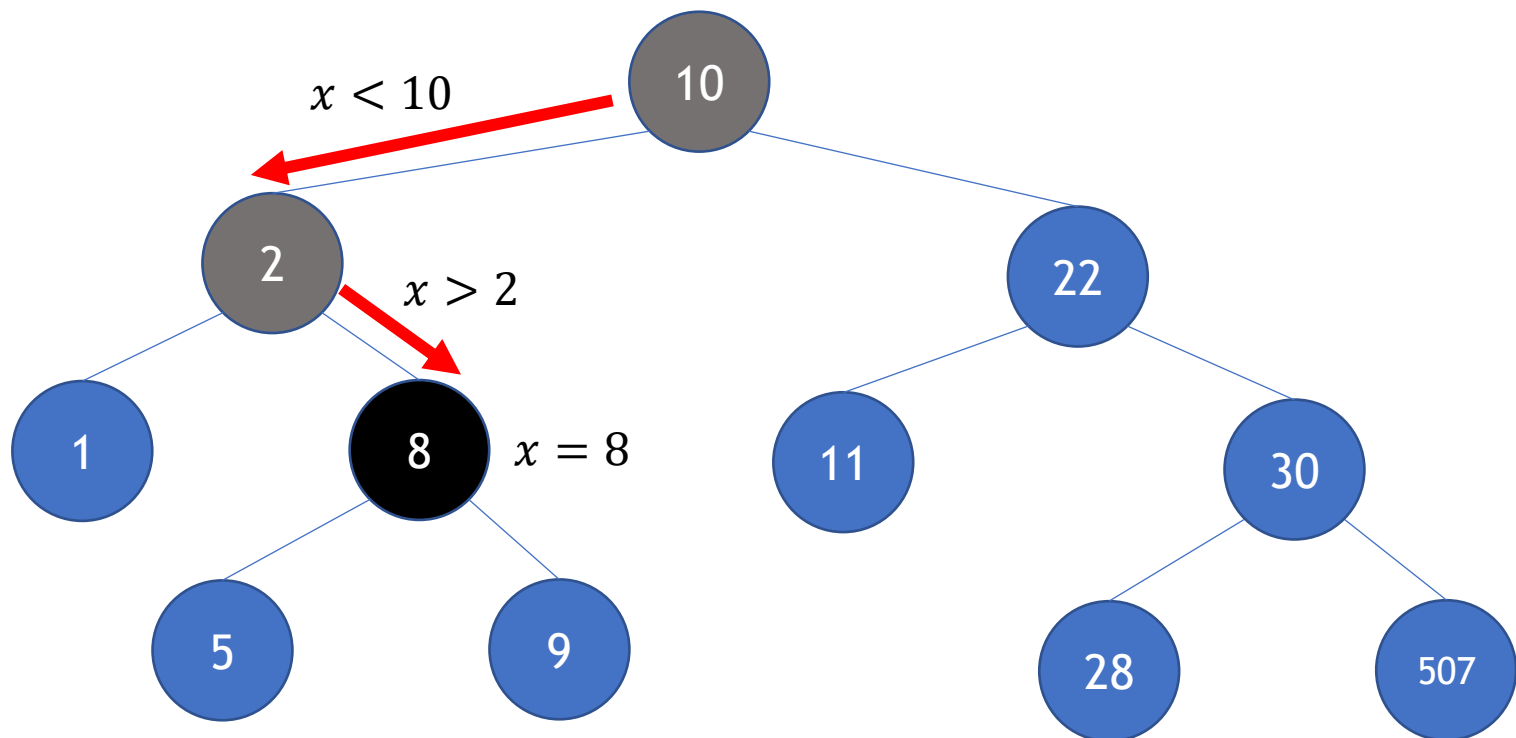
Busca em ABB

- Exemplo: encontrar chave $x = 8$



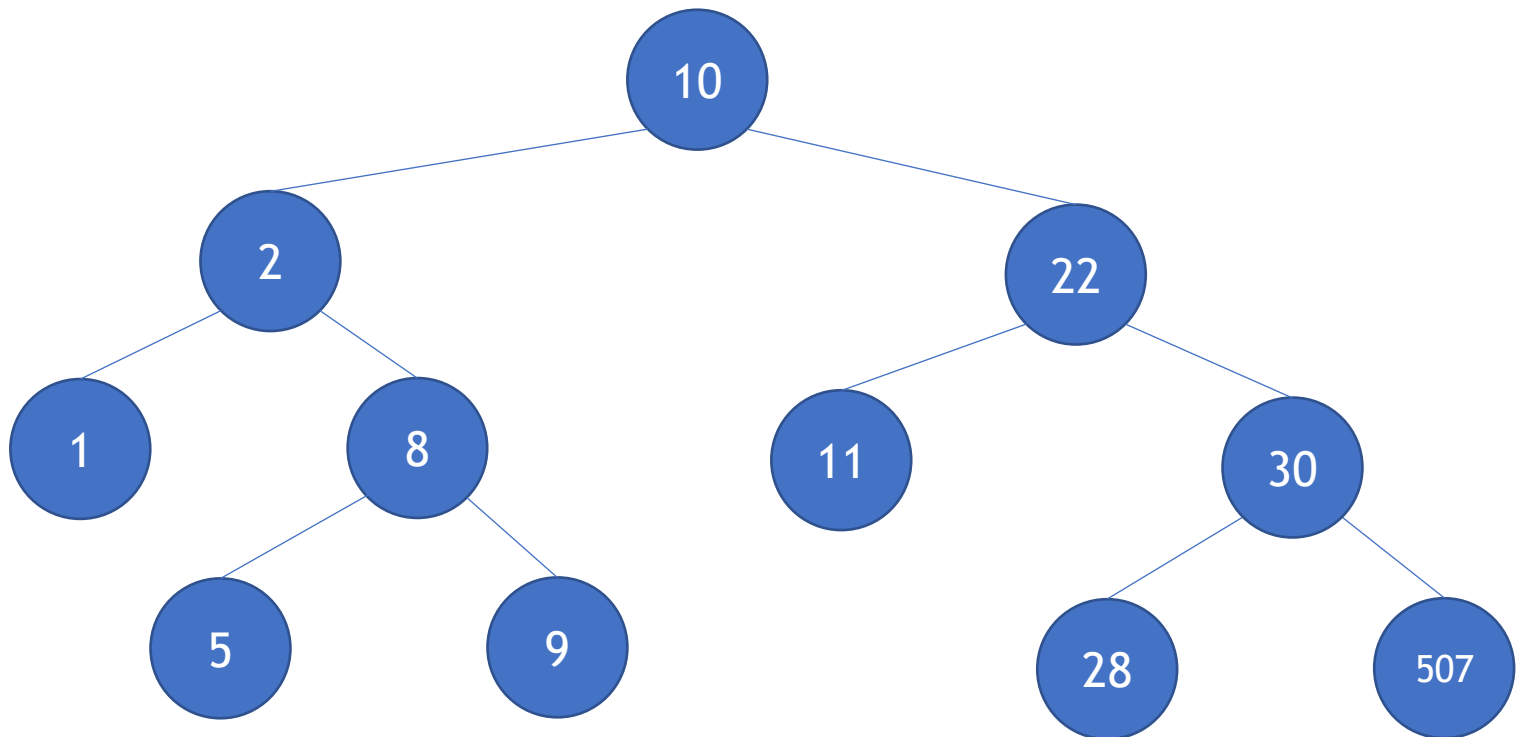
Busca em ABB

- Exemplo: encontrar chave $x = 8$



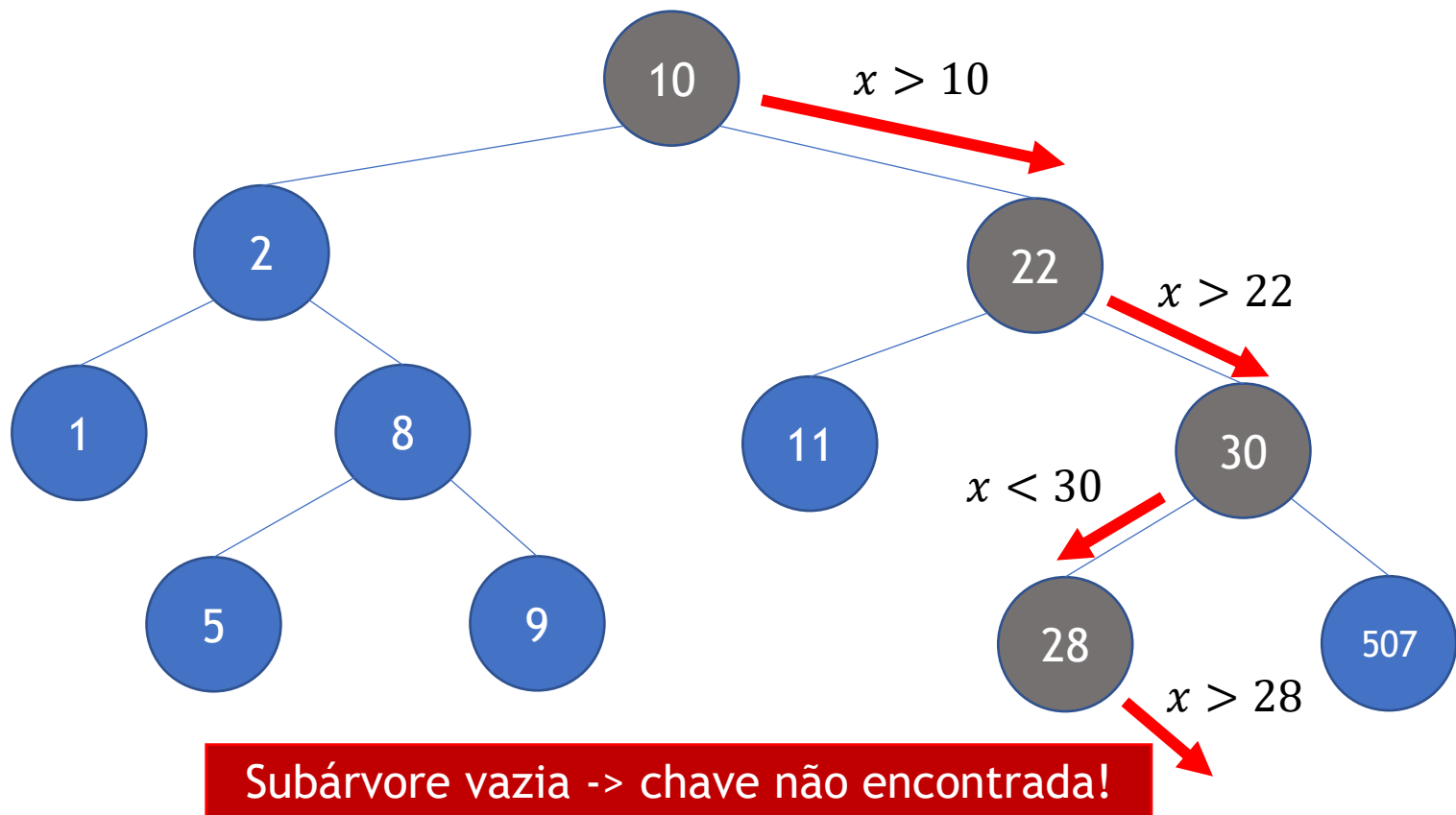
Busca em ABB

- Exemplo: encontrar chave $x = 29$



Busca em ABB

- Exemplo: encontrar chave $x = 29$



Busca em ABB

- Implementação em C

Busca em ABB

- Implementação em C

```
NoArvore *buscaNo(NoArvore *raiz, int chave_busca) {  
    if (raiz == NULL) return NULL;  
    if (chave_busca == raiz->chave)  
        return raiz;  
    else if (chave_busca < raiz->chave)  
        return buscaNo(raiz->esq, chave_busca);  
    else  
        return buscaNo(raiz->dir, chave_busca);  
}
```

Busca em ABB

- Implementação em C

```
NoArvore *buscaNo(NoArvore *raiz, int chave_busca) {  
    if (raiz == NULL) return NULL;  
    if (chave_busca == raiz->chave)  
        return raiz;  
    else if (chave_busca < raiz->chave)  
        return buscaNo(raiz->esq, chave_busca);  
    else  
        return buscaNo(raiz->dir, chave_busca);  
}
```

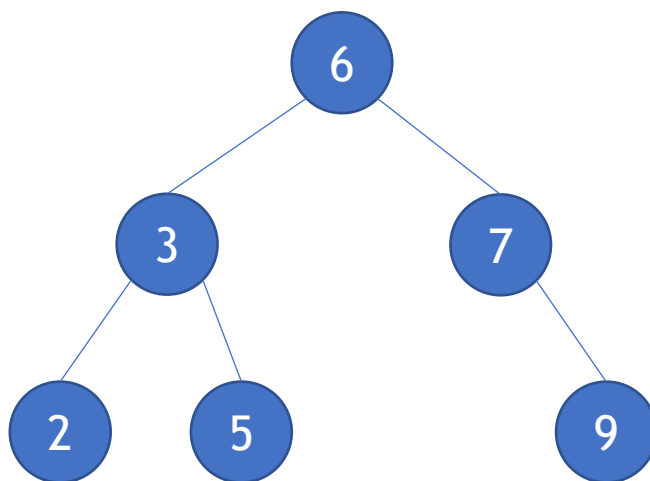
E como seria a implementação iterativa (sem recursão)?

Inserção em ABB

- Na ABB, toda chave nova é inserida em um novo nó, e esse nó será um nó folha.
- Antes de inserir o nó, é necessário localizar quem será o nó pai deste novo nó para manter a propriedade da ABB.

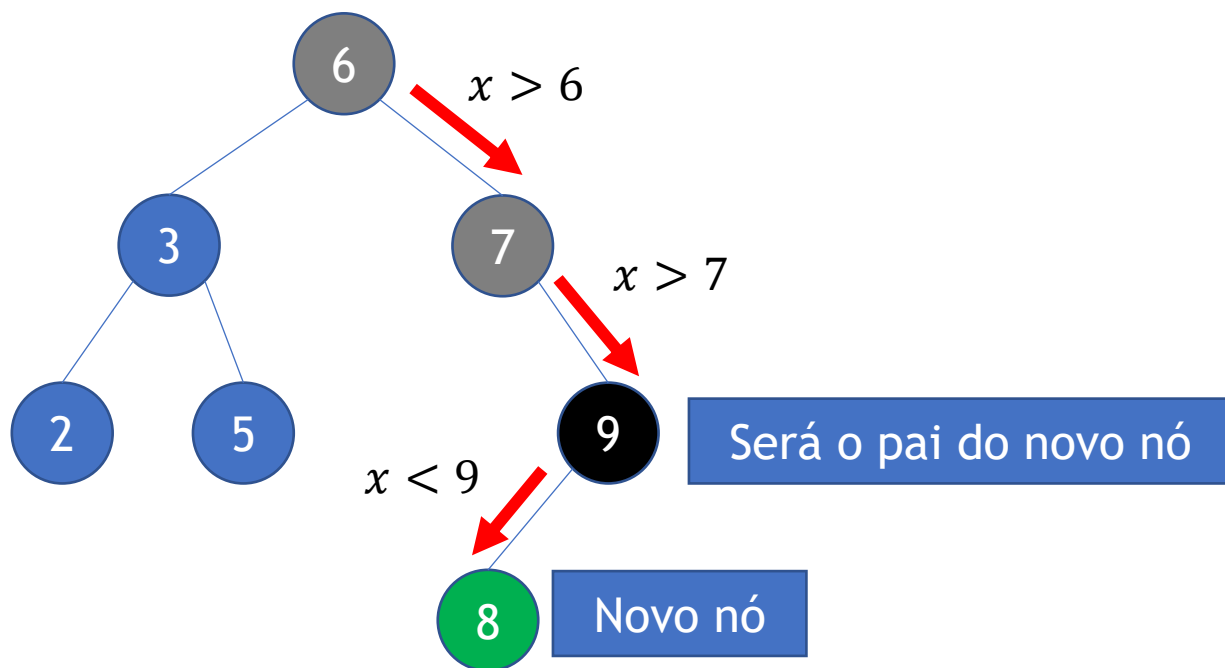
Inserção em ABB

- Exemplo: inserir chave = 8



Inserção em ABB

- Exemplo: inserir chave = 8



Inserção em ABB

- Implementação em C

Chamada:

```
tree = inserir(tree, 507);
```

Chamada:
`tree = inserir(tree, 507);`

Inserção em ABB

- Implementação em C

```
NoArvore *inserir(NoArvore *raiz, int chave_nova) {  
    if (raiz == NULL) {  
        NoArvore *novo_no = malloc(sizeof(NoArvore));  
        novo_no->chave = chave_nova;  
        novo_no->esq = NULL;  
        novo_no->dir = NULL;  
        return novo_no;  
    }  
  
    if (chave_nova < raiz->chave)  
        raiz->esq = inserir(raiz->esq, chave_nova);  
    else if (chave_nova > raiz->chave)  
        raiz->dir = inserir(raiz->dir, chave_nova);  
  
    return raiz;  
}
```

Chamada:
`tree = inserir(tree, 507);`

Inserção em ABB

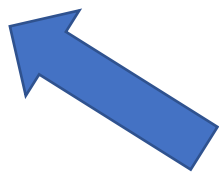
- Implementação em C

```
NoArvore *inserir(NoArvore *raiz, int chave_nova) {  
    if (raiz == NULL) {  
        NoArvore *novo_no = malloc(sizeof(NoArvore));  
        novo_no->chave = chave_nova;  
        novo_no->esq = NULL;  
        novo_no->dir = NULL;  
        return novo_no;  
    }  
  
    if (chave_nova < raiz->chave)  
        raiz->esq = inserir(raiz->esq, chave_nova);  
    else if (chave_nova > raiz->chave)  
        raiz->dir = inserir(raiz->dir, chave_nova);  
  
    return raiz;  
}
```

E como seria a implementação iterativa
(sem recursão)?

Remoção em ABB

- Na remoção, primeiro buscamos o nó e então o removemos;
- Nesta operação, o nó a ser removido por ser:
 1. Nó folha;
 2. Nó com 1 filho;
 3. Nó com 2 filhos.



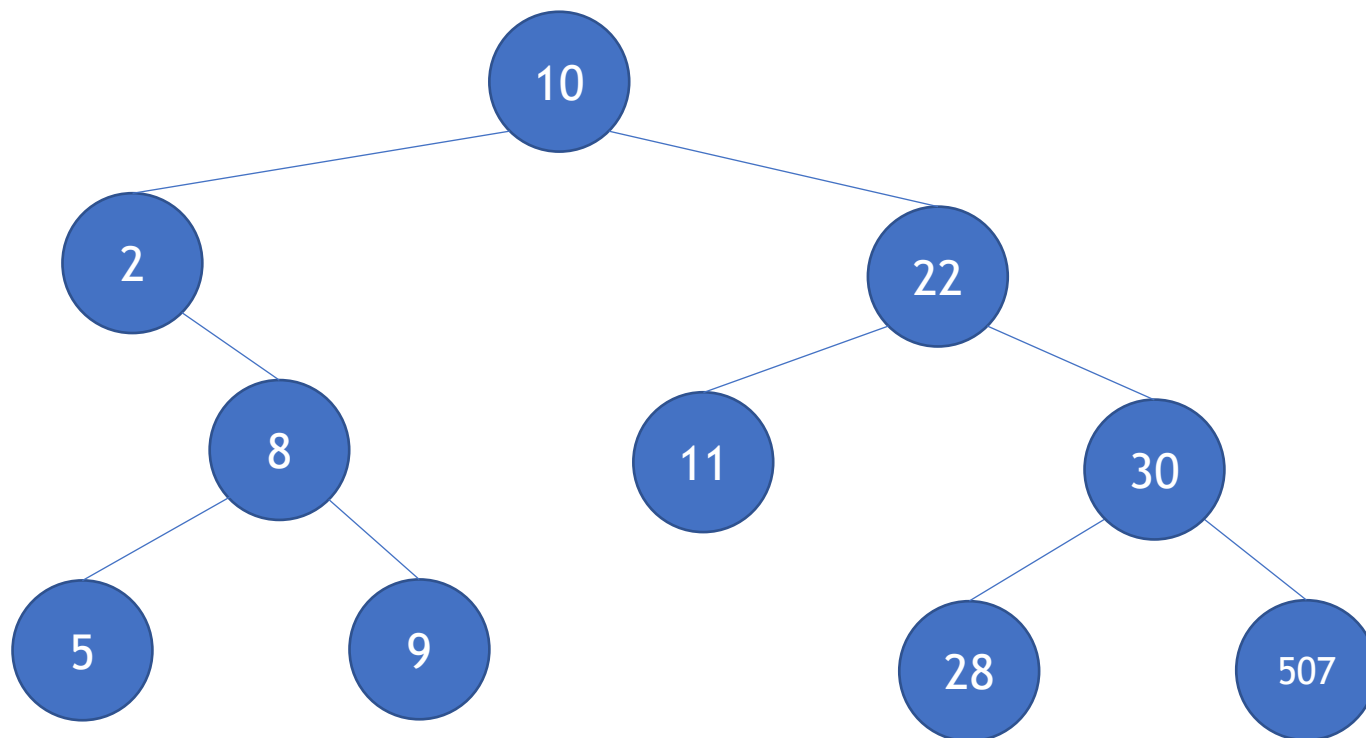
Cada caso requer um tratamento específico!

Remoção em ABB - Caso 1

Remover nó folha

- Exemplo: remover 28

- remove o nó
- ponteiro do pai recebe NULL

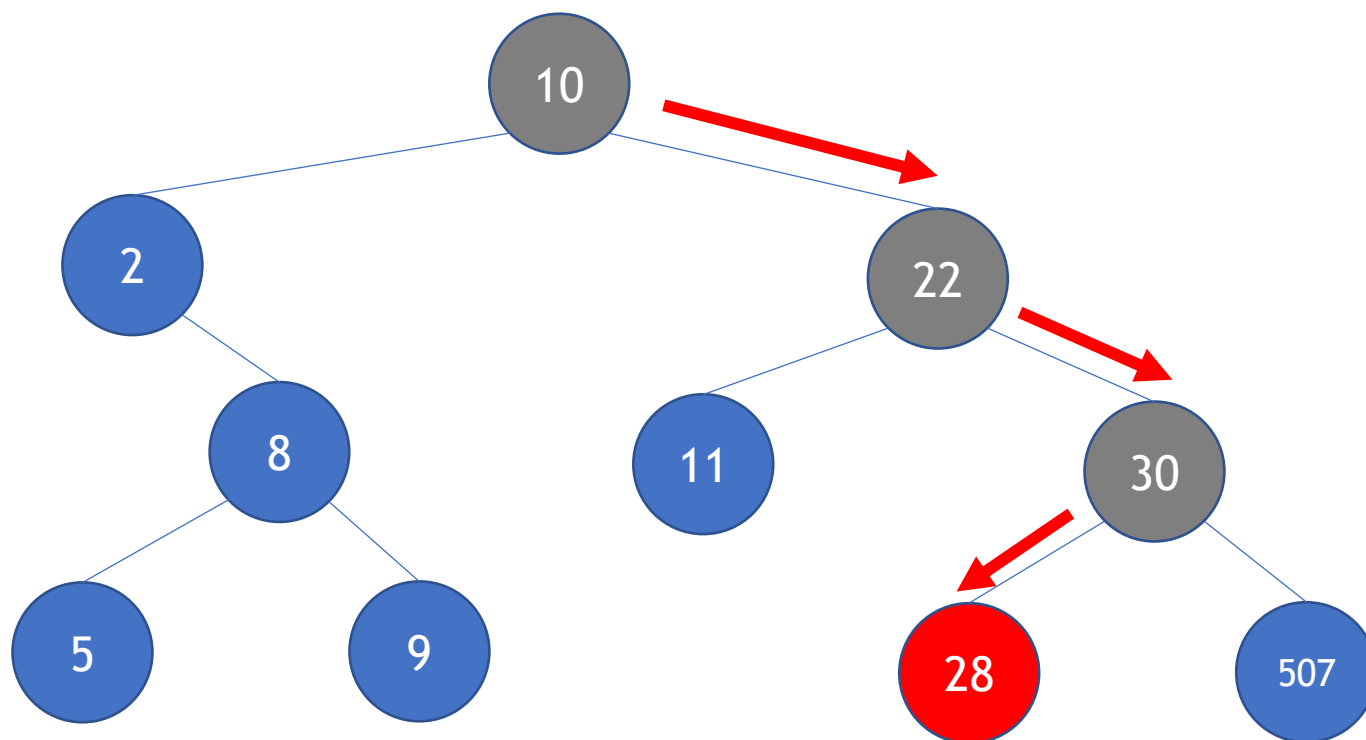


Remoção em ABB - Caso 1

Remover nó folha

- Exemplo: remover 28

- remove o nó
- ponteiro do pai recebe NULL

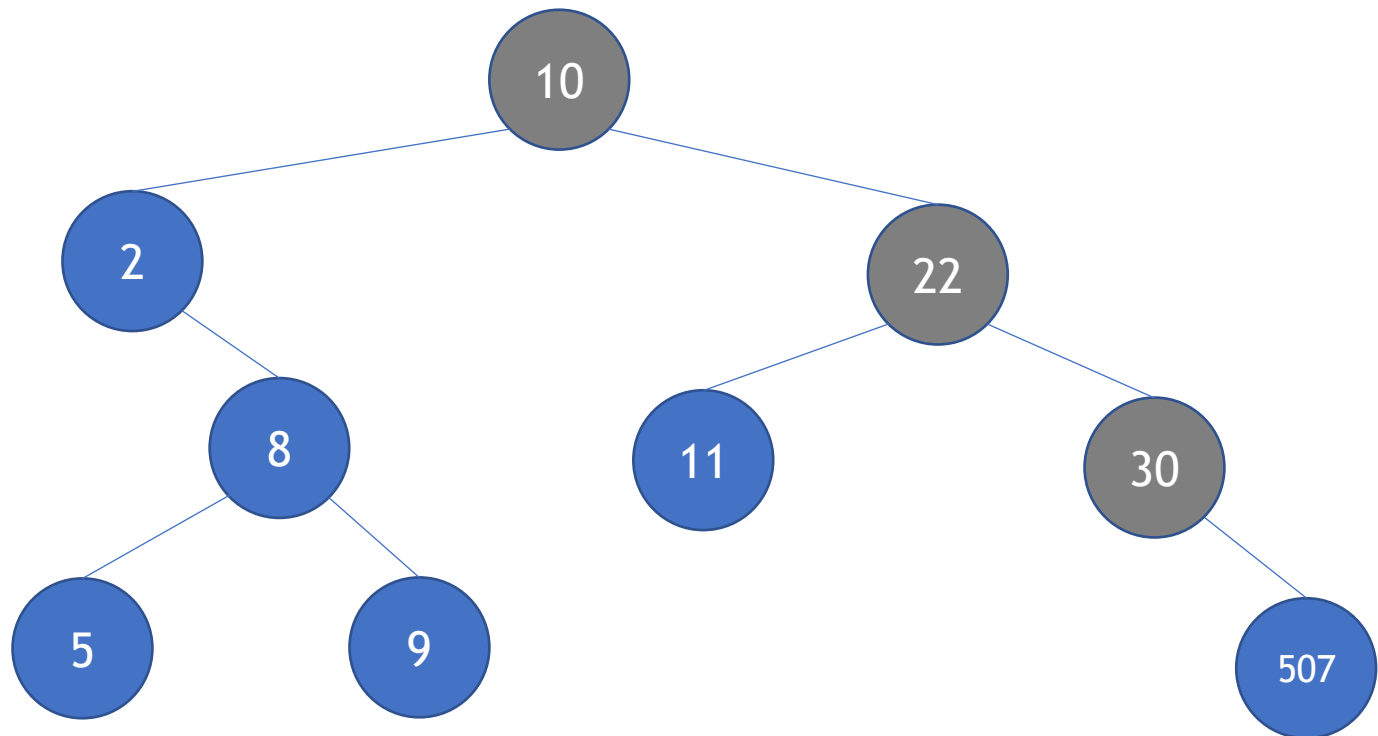


Remoção em ABB - Caso 1

Remover nó folha

- Exemplo: remover 28

- remove o nó
- ponteiro do pai recebe NULL

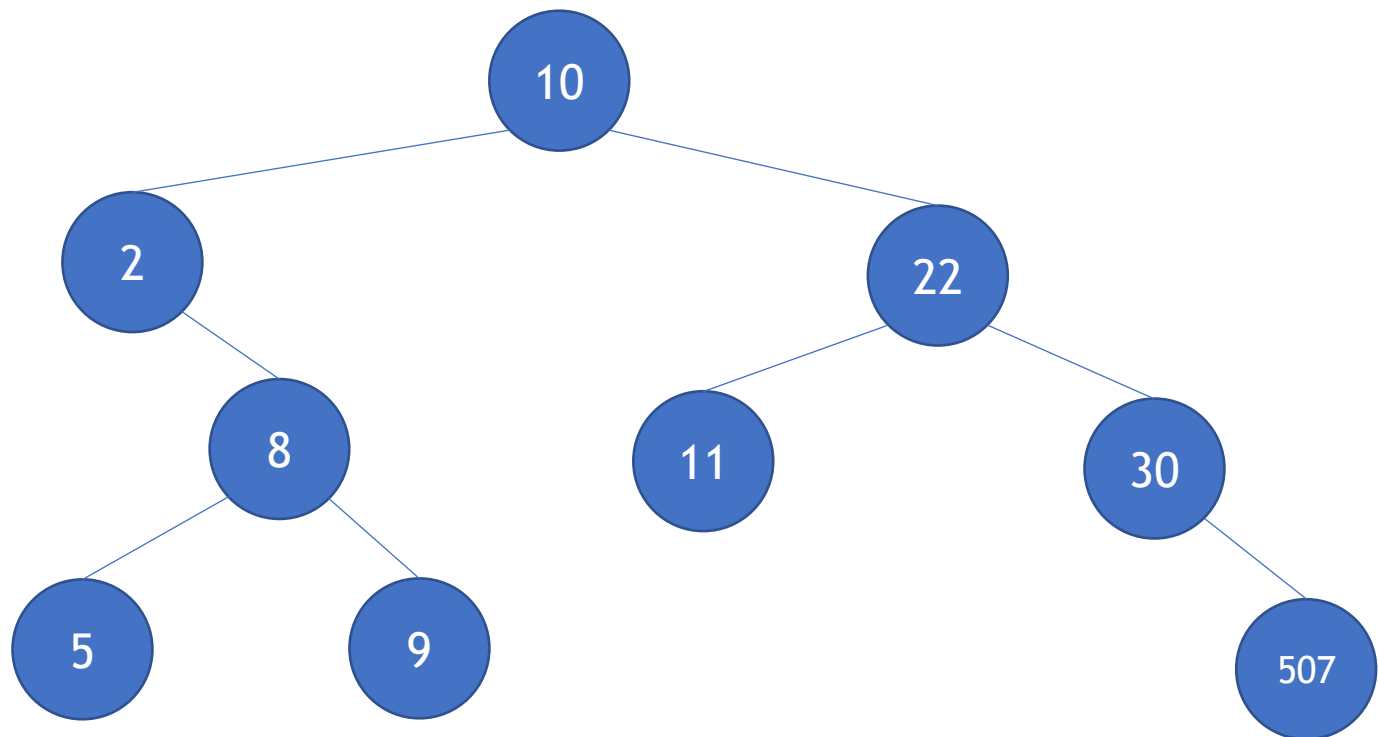


Remoção em ABB - Caso 2

- Exemplo: remover 2

Remover nó com 1 filho

- nó pai aponta para filho do nó a ser removido
- remove nó

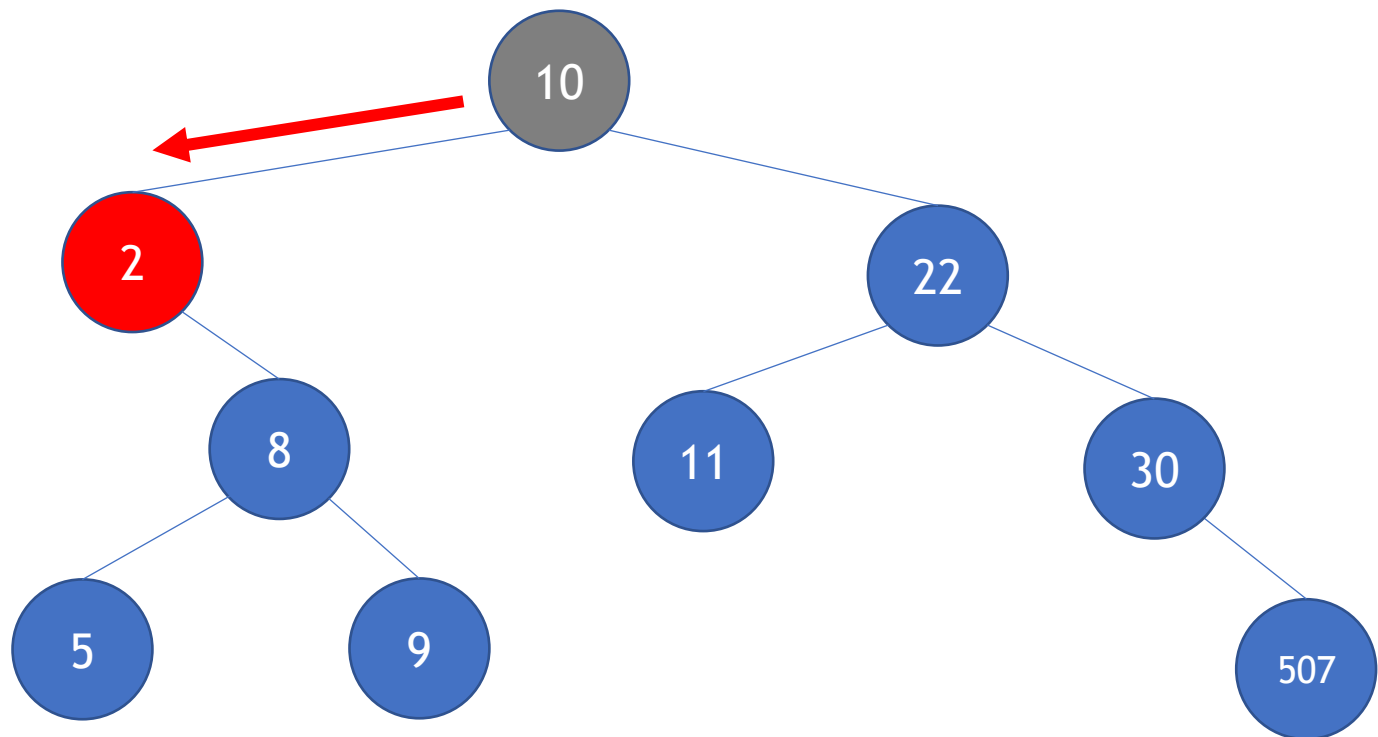


Remoção em ABB - Caso 2

- Exemplo: remover 2

Remover nó com 1 filho

- nó pai aponta para filho do nó a ser removido
- remove nó

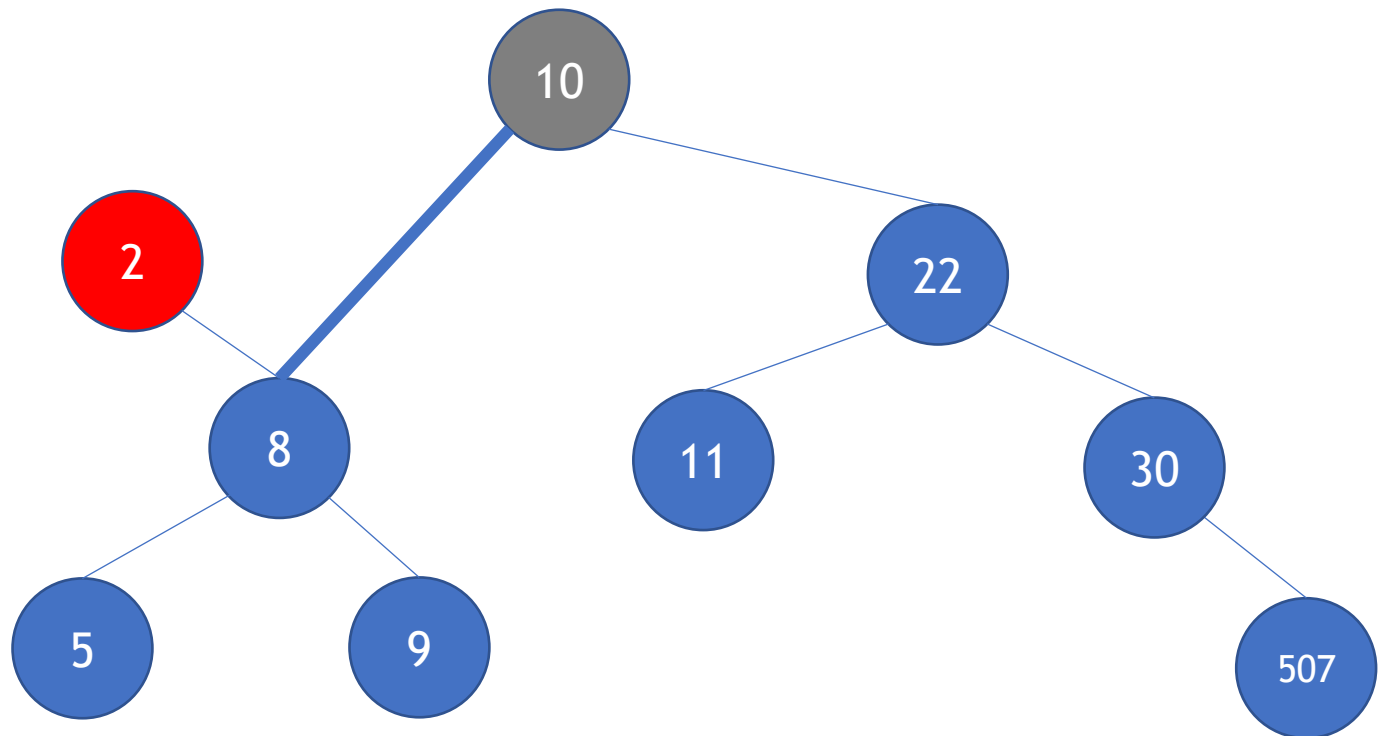


Remoção em ABB - Caso 2

- Exemplo: remover 2

Remover nó com 1 filho

- nó pai aponta para filho do nó a ser removido
- remove nó

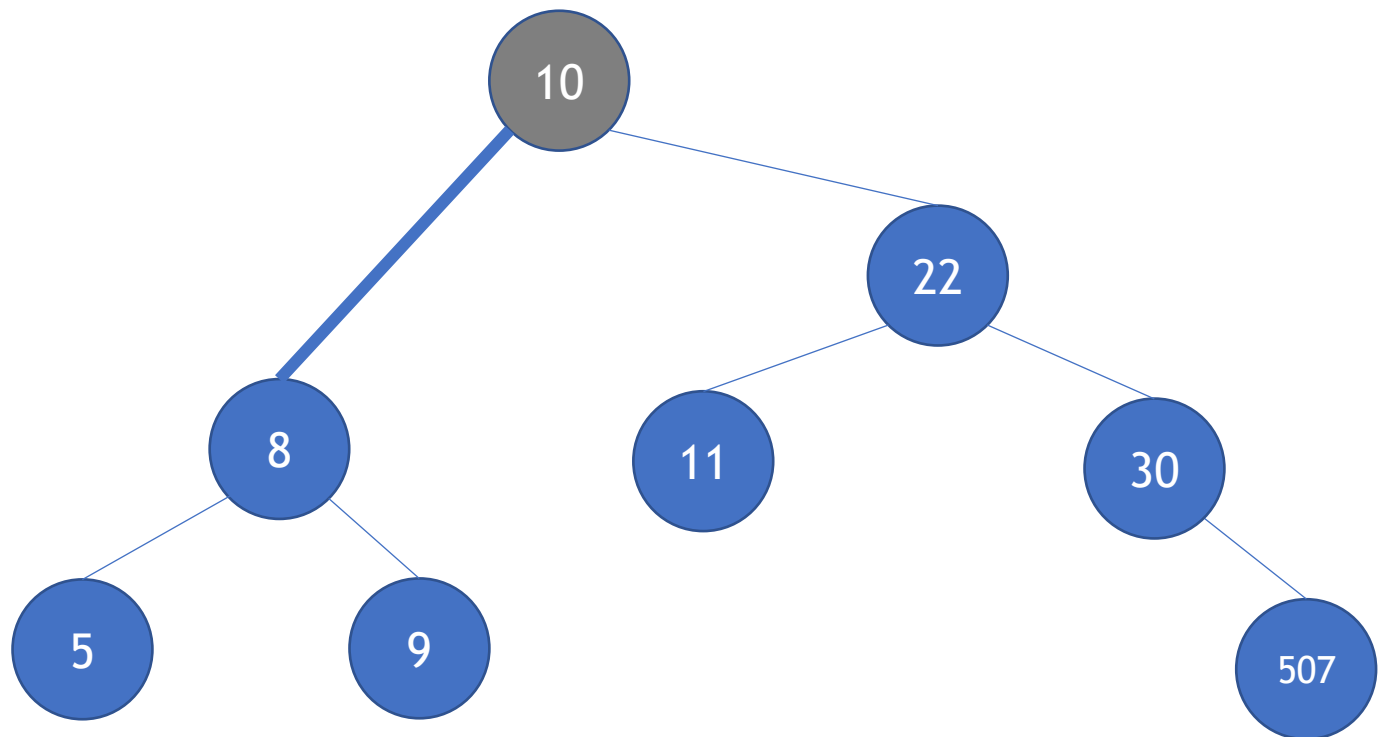


Remoção em ABB - Caso 2

- Exemplo: remover 2

Remover nó com 1 filho

- nó pai aponta para filho do nó a ser removido
- remove nó

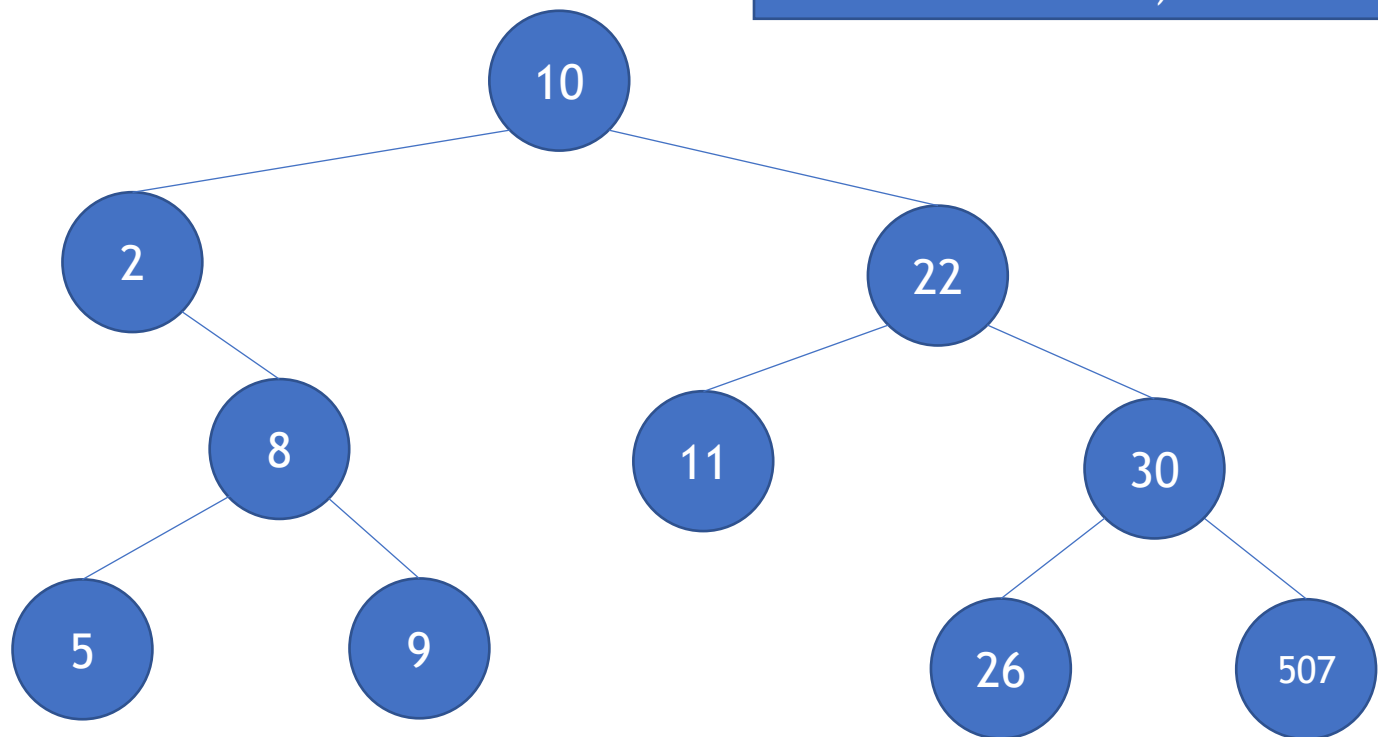


Remoção em ABB - Caso 3

- Exemplo: remover 22

Remover nó com 2 filhos

- copia valor do sucessor no nó
- remove cópia na subárvore direita (o sucessor terá no máximo 1 filho)

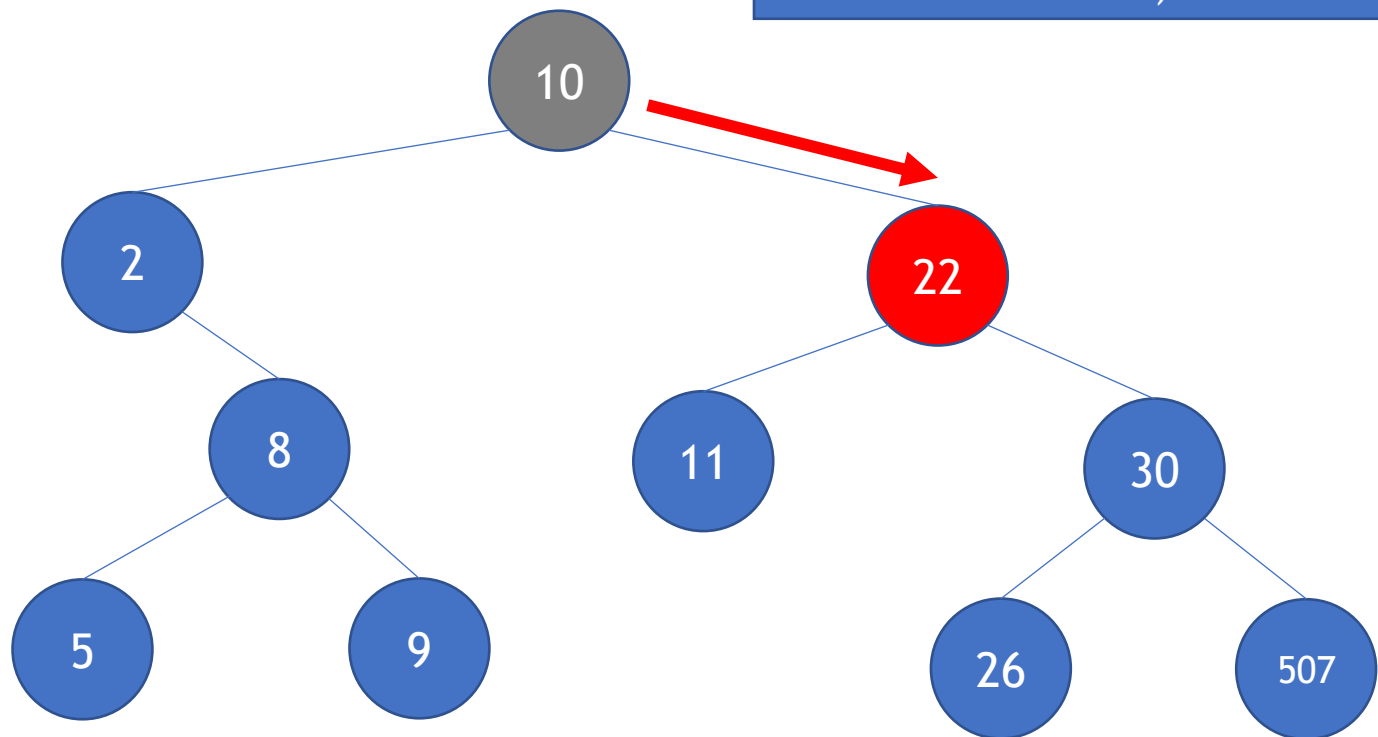


Remoção em ABB - Caso 3

- Exemplo: remover 22

Remover nó com 2 filhos

- copia valor do sucessor no nó
- remove cópia na subárvore direita (o sucessor terá no máximo 1 filho)

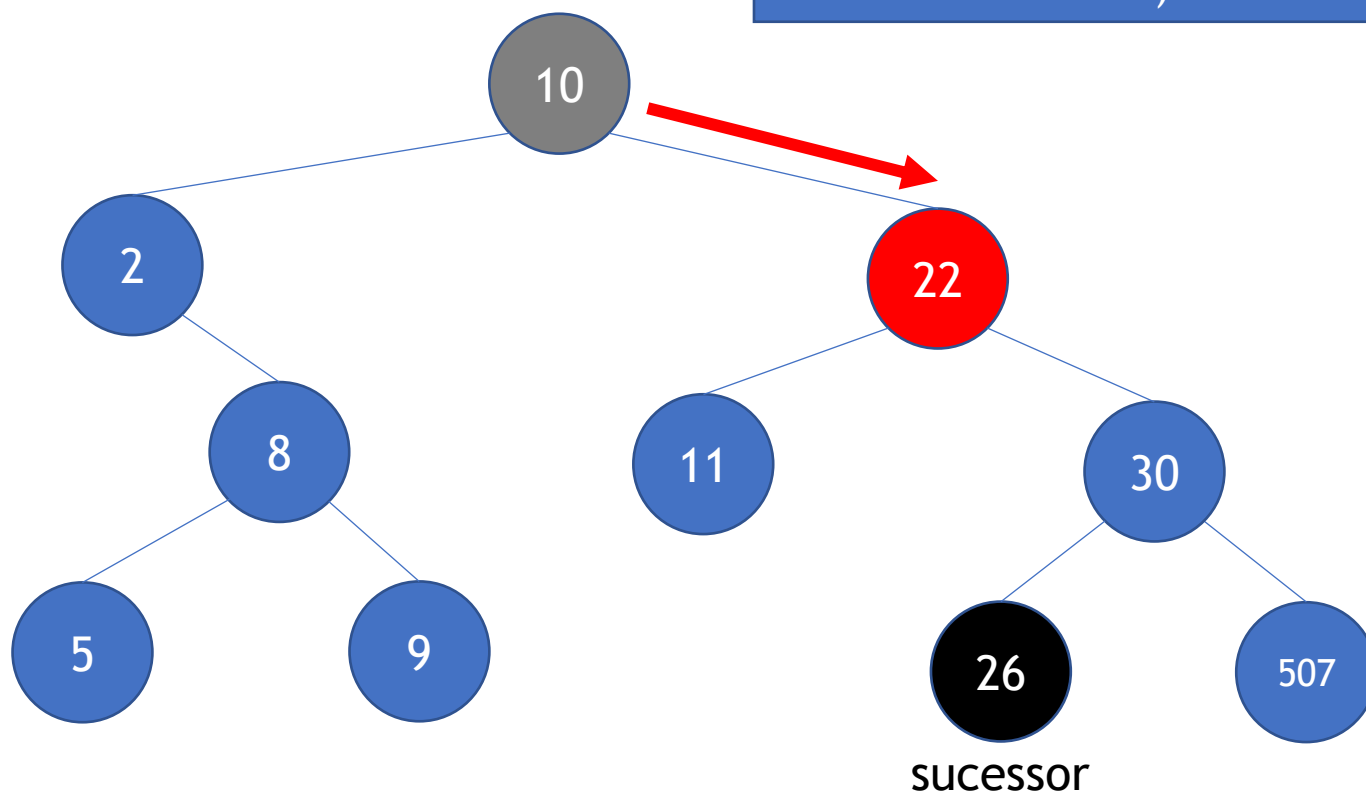


Remoção em ABB - Caso 3

- Exemplo: remover 22

Remover nó com 2 filhos

- copia valor do sucessor no nó
- remove cópia na subárvore direita (o sucessor terá no máximo 1 filho)

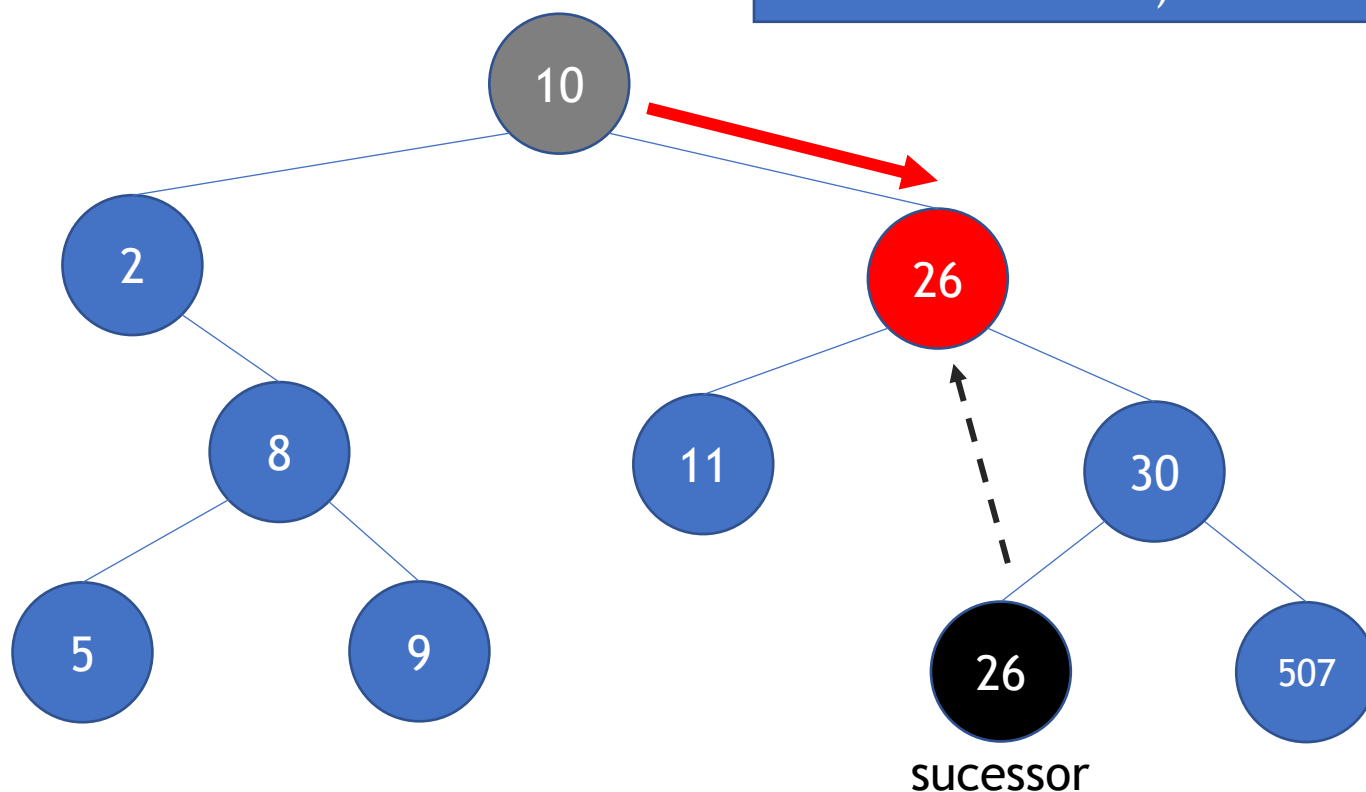


Remoção em ABB - Caso 3

- Exemplo: remover 22

Remover nó com 2 filhos

- copia valor do sucessor no nó
- remove cópia na subárvore direita (o sucessor terá no máximo 1 filho)

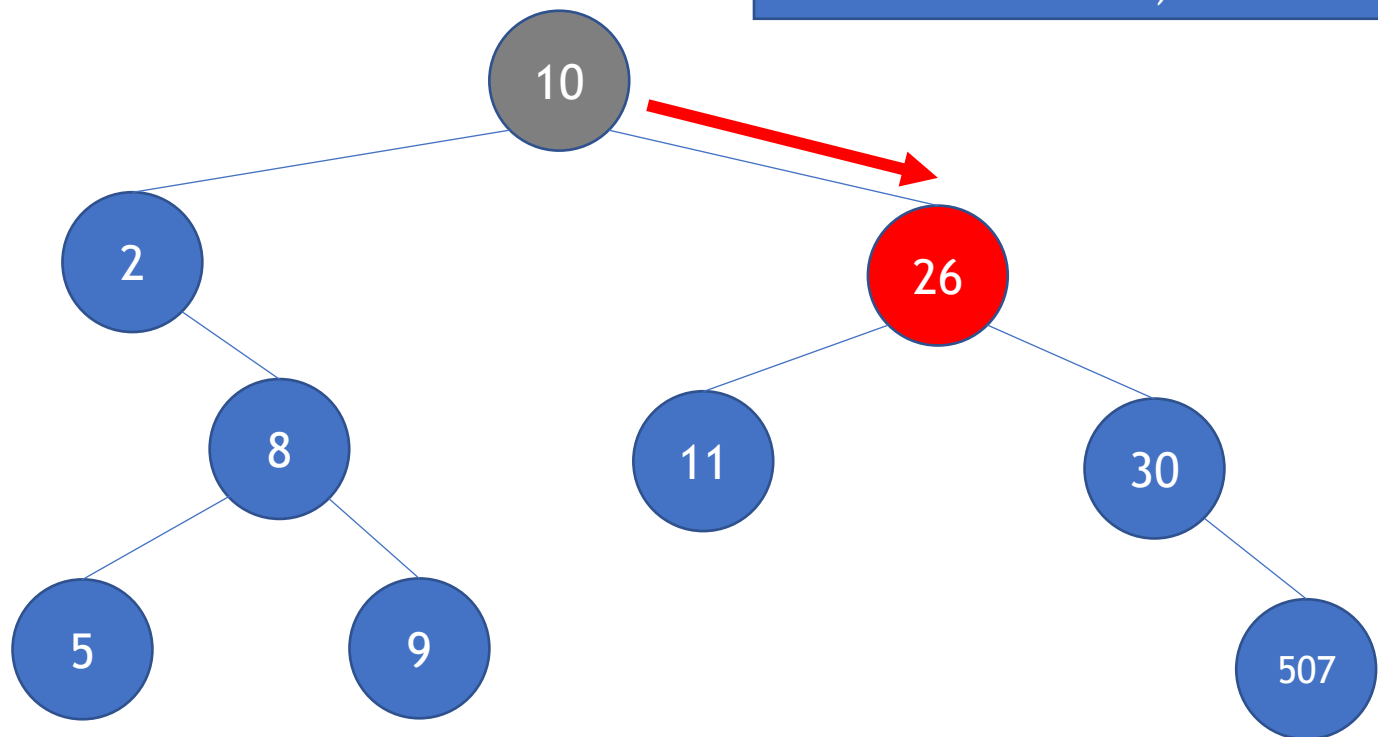


Remoção em ABB - Caso 3

- Exemplo: remover 22

Remover nó com 2 filhos

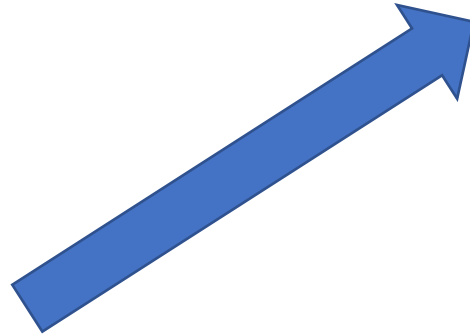
- copia valor do sucessor no nó
- remove cópia na subárvore direita (o sucessor terá no máximo 1 filho)



Remoção em ABB - Caso 3

Remover nó com 2 filhos

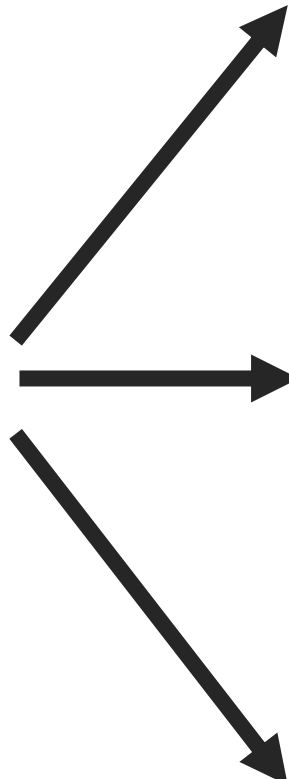
- copia valor do sucessor no nó
- remove cópia na subárvore direita (o sucessor terá no máximo 1 filho)



Podemos também copiar o antecessor
(nesse caso, a remoção da cópia será
na subárvore esquerda)

Remoção em ABB

Remoção em árvore
binária de busca



Remover nó folha

- remove o nó
- ponteiro do pai recebe NULL

Remover nó com 1 filho

- nó pai aponta para filho do nó a ser removido
- remove nó

Remover nó com 2 filhos

- copia valor do sucessor no nó
- remove cópia na subárvore direita (o sucessor terá no máximo 1 filho)

Remover em ABB

- Implementação em C

Chamada:

```
tree = remover(tree, 507);
```

Remover em ABB

Chamada:
tree = remover(tree, 507);

- Implementação em C

```
NoArvore *remover(NoArvore *raiz, int chave) {
    if (raiz == NULL) return NULL;
    if (chave < raiz->chave)
        raiz->esq = remover(raiz->esq, chave);
    else if (chave > raiz->chave)
        raiz->dir = remover(raiz->dir, chave);
    else {
        if (raiz->esq == NULL && raiz->dir == NULL) {
            free(raiz);
            return NULL;
        }
        if (raiz->esq == NULL) {
            NoArvore *filhoDir = raiz->dir;
            free(raiz);
            return filhoDir;
        }
        if (raiz->dir == NULL) {
            NoArvore *filhoEsq = raiz->esq;
            free(raiz);
            return filhoEsq;
        }
        int sucessor = get_min_iter(raiz->dir);
        raiz->chave = sucessor;
        raiz->dir = remover(raiz->dir, sucessor);
    }
    return raiz;
}
```

Explicação a seguir



```

NoArvore *remover(NoArvore *raiz, int chave) {
    if (raiz == NULL) return NULL;
    if (chave < raiz->chave)
        raiz->esq = remover(raiz->esq, chave);
    else if (chave > raiz->chave)
        raiz->dir = remover(raiz->dir, chave);
    else {
        if (raiz->esq == NULL && raiz->dir == NULL) {
            free(raiz);
            return NULL;
        }
        if (raiz->esq == NULL) {
            NoArvore *filhoDir = raiz->dir;
            free(raiz);
            return filhoDir;
        }
        if (raiz->dir == NULL) {
            NoArvore *filhoEsq = raiz->esq;
            free(raiz);
            return filhoEsq;
        }
        int sucessor = get_min_iter(raiz->dir);
        raiz->chave = sucessor;
        raiz->dir = remover(raiz->dir, sucessor);
    }
    return raiz;
}

```

Busca nó a ser removido

Caso 1

Caso 2

Caso 3


```
NoArvore *remover(NoArvore *raiz, int chave) {  
    if (raiz == NULL) return NULL;
```

```
    if (chave < raiz->chave)  
        raiz->esq = remover(raiz->esq, chave);  
    else if (chave > raiz->chave)  
        raiz->dir = remover(raiz->dir, chave);  
    else {
```

Busca nó a ser removido

```
        if (raiz->esq == NULL) {  
            NoArvore *filhoDir = raiz->dir;  
            free(raiz);  
            return filhoDir;
```

```
        }  
        if (raiz->dir == NULL) {  
            NoArvore *filhoEsq = raiz->esq;  
            free(raiz);  
            return filhoEsq;
```

**Casos 1 e 2
(simplificação)**

```
        int sucessor = get_min_iter(raiz->dir);  
        raiz->chave = sucessor;  
        raiz->dir = remover(raiz->dir, sucessor);
```

Caso 3

```
    }  
    return raiz;
```

```
}
```

Complexidade das operações

Complexidade das operações

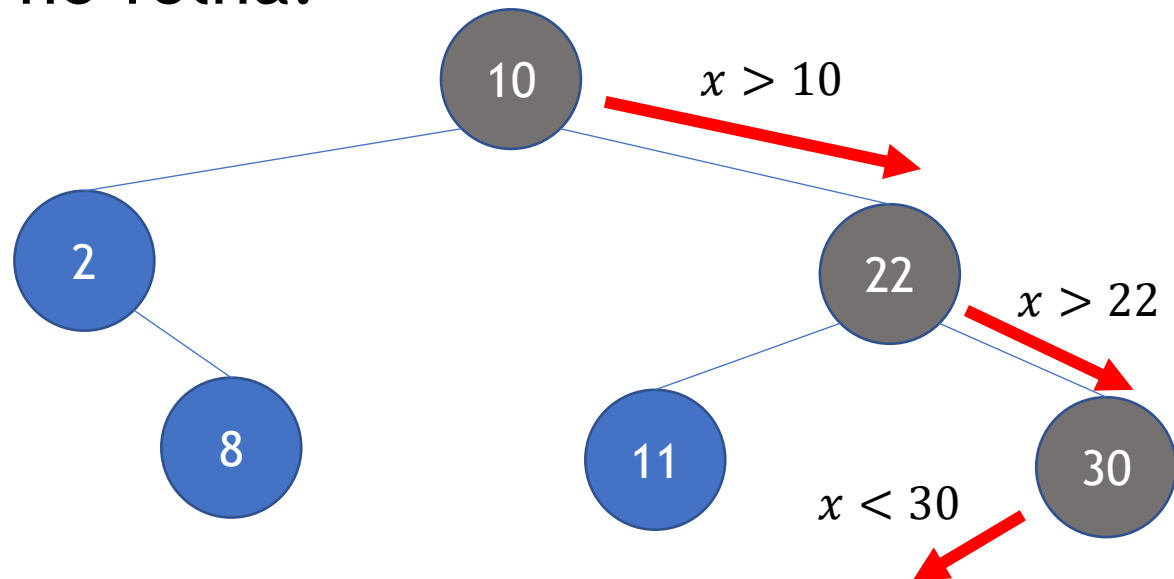
- No início da aula, vimos que a complexidade das operações básicas em árvores pode ser:

| Operação | Árvores |
|----------|--------------|
| Busca | $O(\log(n))$ |
| Inserção | $O(\log(n))$ |
| Remoção | $O(\log(n))$ |

- Será que isso vale para qualquer árvore binária?

Complexidade das operações

- **Busca:** o pior caso ocorre quando a chave buscada não é encontrada, pois o algoritmo vai até um nó folha.



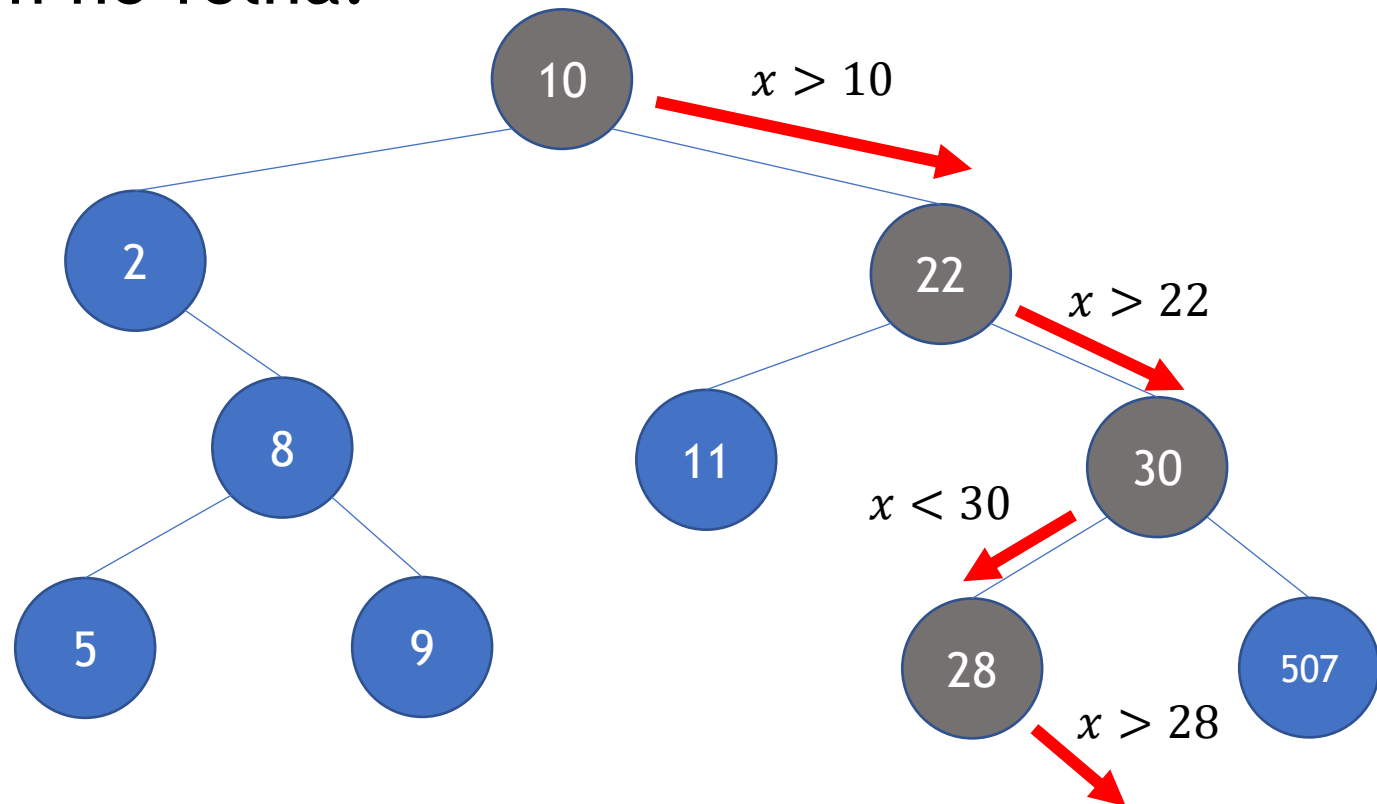
Buscar
 $x = 29$

6 nós na árvore
Altura(h) = 2

3 comparações

Complexidade das operações

- **Busca:** o pior caso ocorre quando a chave buscada não é encontrada, pois o algoritmo vai até um nó folha.



Buscar
 $x = 29$

10 nós na árvore
Altura(h) = 3

4 comparações

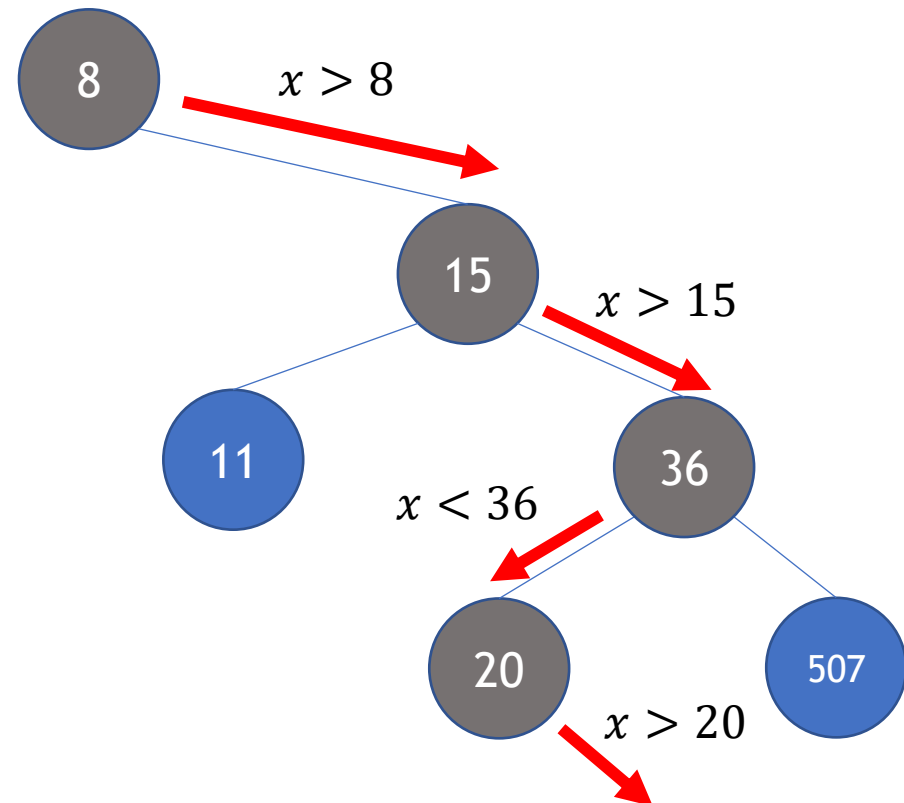
Complexidade das operações

- **Busca:** o pior caso ocorre quando a chave buscada não é encontrada, pois o algoritmo vai até um nó folha.

Buscar
 $x = 29$

6 nós na árvore
Altura(h) = 3

4 comparações



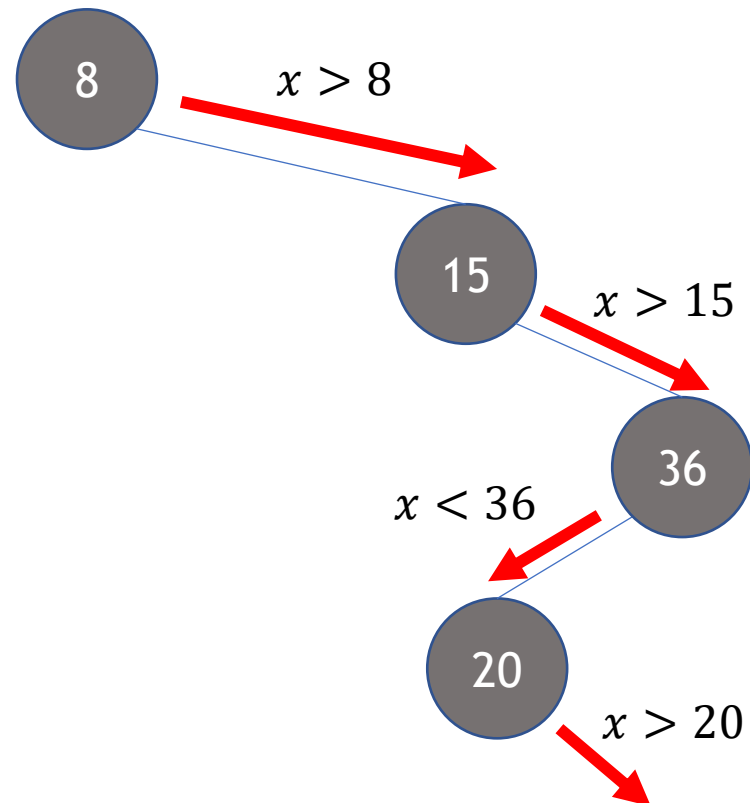
Complexidade das operações

- **Busca:** o pior caso ocorre quando a chave buscada não é encontrada, pois o algoritmo vai até um nó folha.

Buscar
 $x = 29$

4 nós na árvore
Altura(h) = 3

4 comparações



Complexidade das operações

- **Busca:** o pior caso ocorre quando a chave buscada não é encontrada, pois o algoritmo vai até um nó folha.
- Podemos ver que a quantidade de comparações é dependente da **altura (h) da árvore**. Portanto, a complexidade da busca é:

$$O(h)$$

Complexidade das operações

- As operações de inserção e remoção envolvem busca também. Portanto, chegamos que:

| Operação | Árvores |
|----------|---------|
| Busca | $O(h)$ |
| Inserção | $O(h)$ |
| Remoção | $O(h)$ |

Complexidade das operações

- As operações de inserção e remoção envolvem busca também. Portanto, chegamos que:

| Operação | Árvores |
|----------|---------|
| Busca | $O(h)$ |
| Inserção | $O(h)$ |
| Remoção | $O(h)$ |

No pior caso, a altura é $(n-1)$. O que levaria as três operações acima para $O(n)$.

Percurso em árvore

Percursos

- Existem alguns percursos sistemáticos em árvores binárias:
 - Percurso em ordem;
 - Percurso pré-ordem;
 - Percurso pós-ordem.
- Nesses percursos, cada nó é visitado apenas uma vez.

Percursos

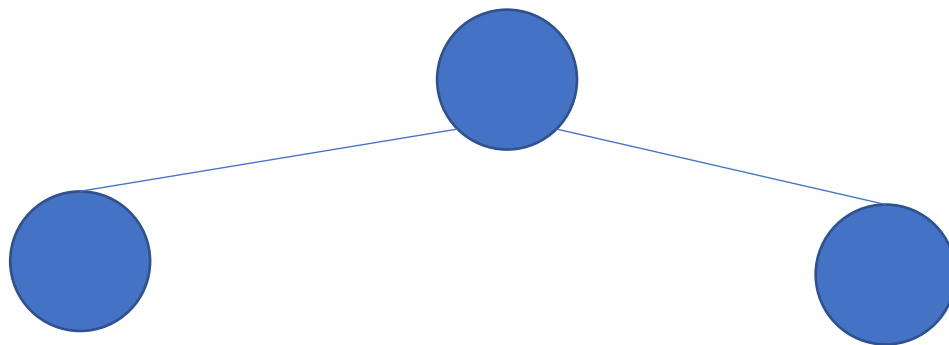
- Existem alguns percursos sistemáticos em árvores binárias:
 - Percurso em ordem;
 - Percurso pré-ordem;
 - Percurso pós-ordem.
- Nesses percursos, cada nó é visitado apenas uma vez.

Os algoritmos em árvores podem ser escritos de forma iterativa ou recursiva, mas frequentemente a implementação recursiva é mais simples para ser escrita.

Percurso em ordem

- Percorre a árvore esquerda *em ordem*;
- Visita a raiz;
- Percorre a árvore direita *em ordem*.

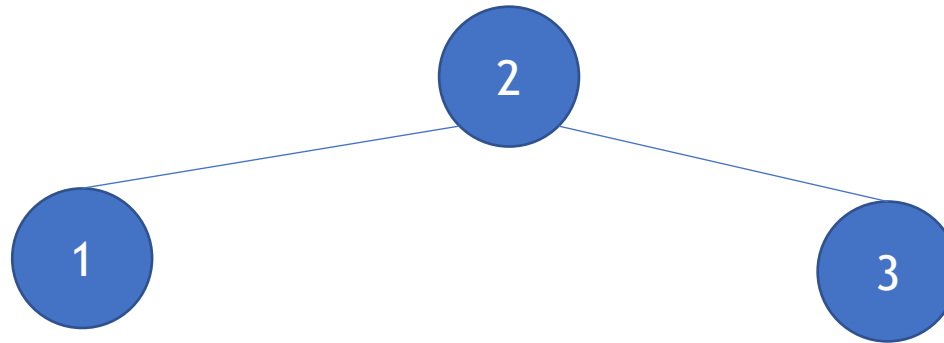
Percurso em ordem



Como seria o percurso em ordem aqui?

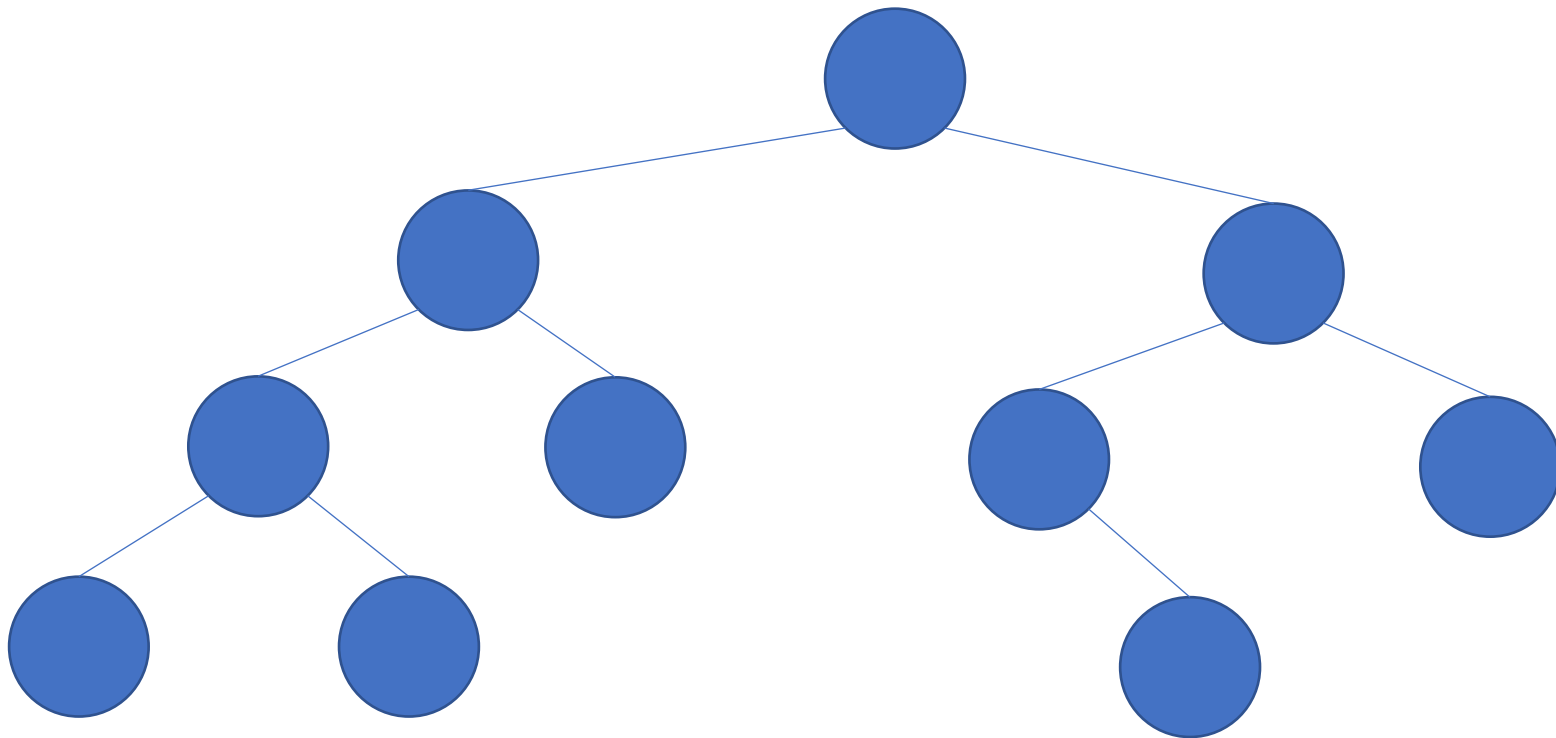
Os números representam a ordem em que cada nó é visitado

Percurso em ordem



Como seria o percurso em ordem aqui?

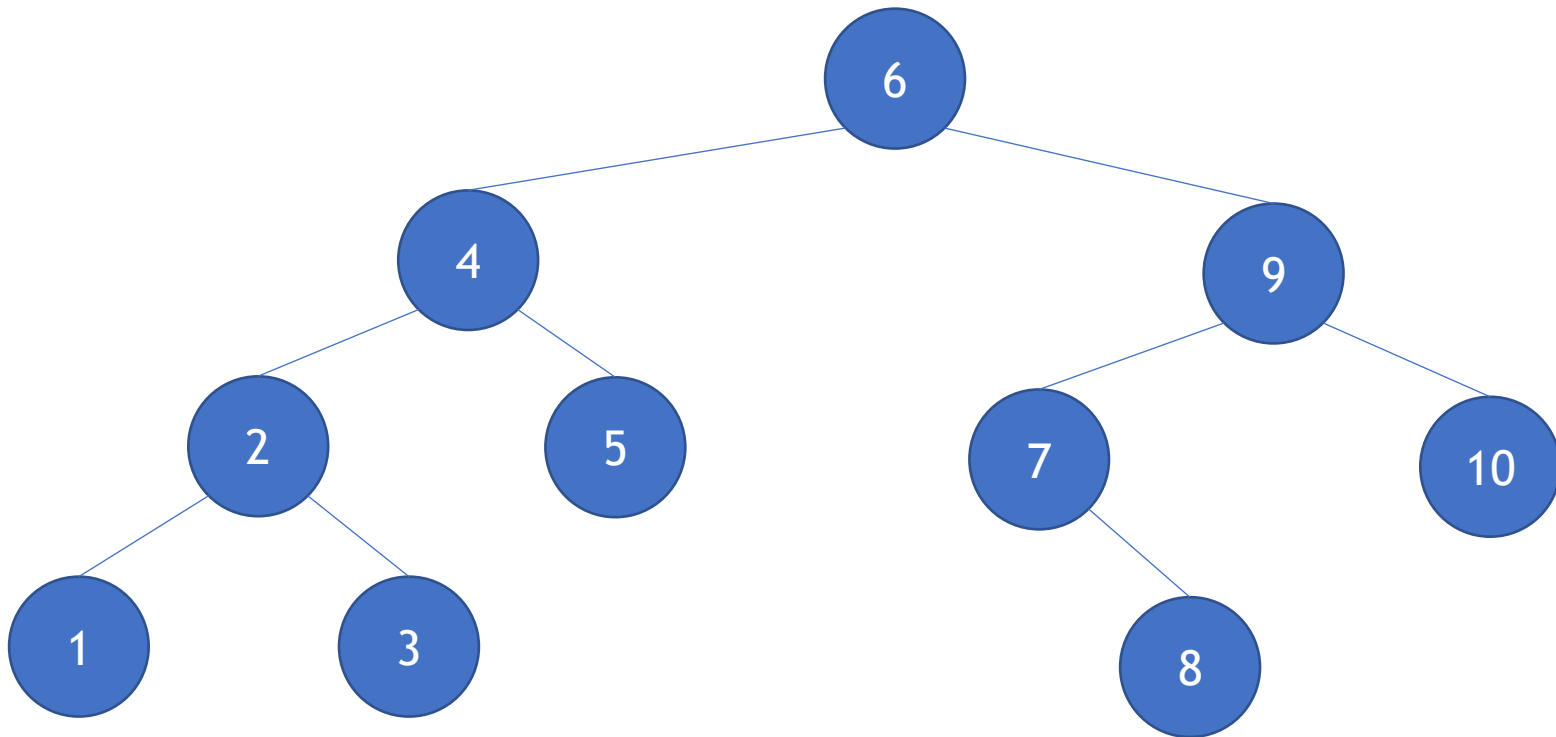
Percurso em ordem



Como seria o percurso em ordem aqui?

Os números representam a ordem em que cada nó é visitado

Percurso em ordem

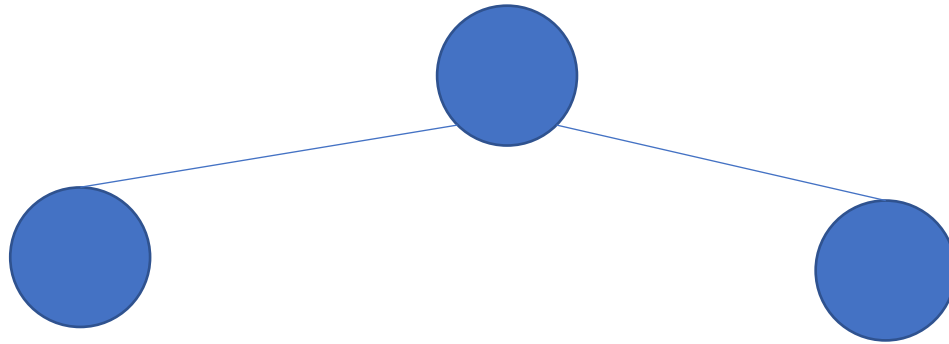


O percurso em ordem percorre os nós em ordem crescente. Pode ser usado, por exemplo, para imprimir todos os nós em ordem crescente.

Percurso pré-ordem

- Visita a raiz;
- Percorre a árvore esquerda *pré-ordem*;
- Percorre a árvore direita *pré-ordem*;

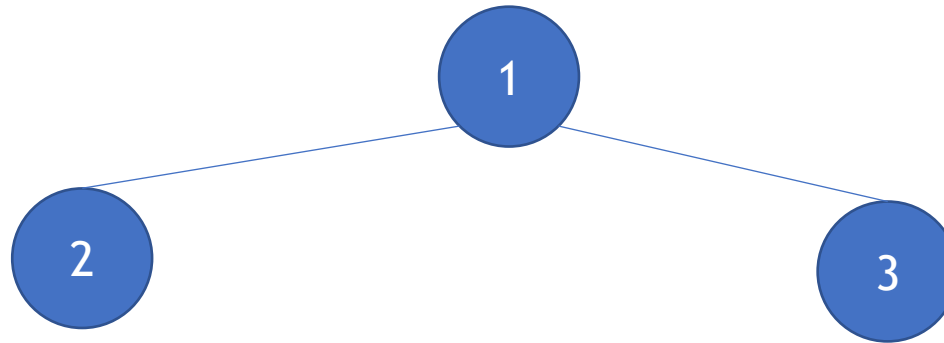
Percurso pré-ordem



Como seria o percurso em pré-ordem aqui?

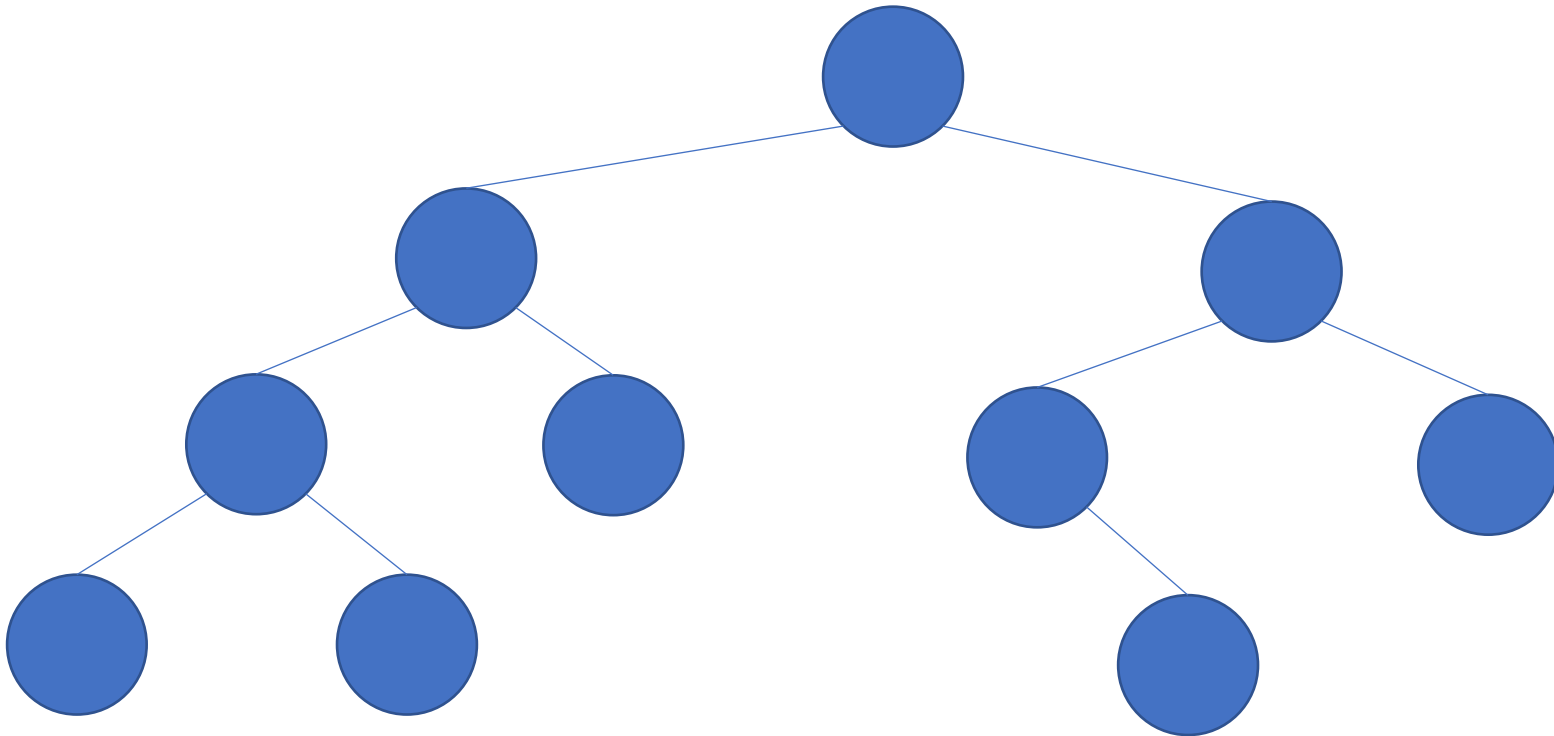
Os números representam a ordem em que cada nó é visitado

Percurso pré-ordem



Como seria o percurso em pré-ordem aqui?

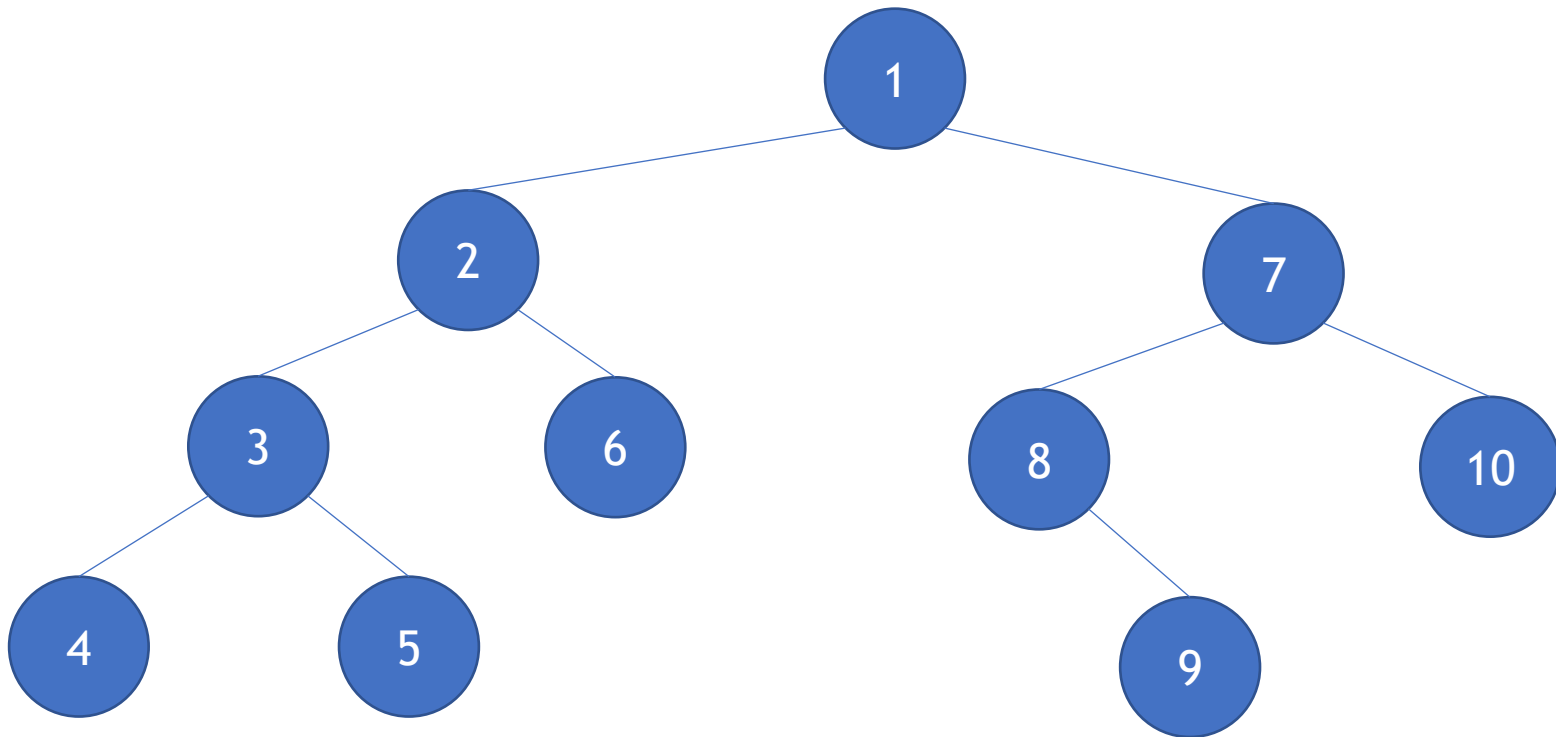
Percurso pré-ordem



Como seria o percurso em pré-ordem aqui?

Os números representam a ordem em que cada nó é visitado

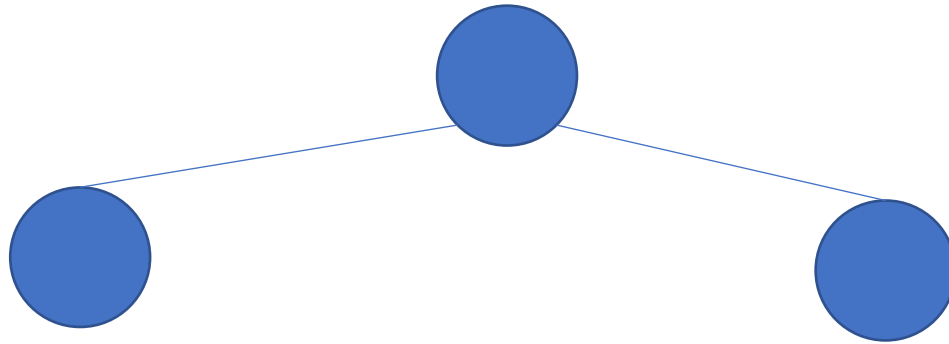
Percurso pré-ordem



Percurso pós-ordem

- Percorre a árvore esquerda *pós-ordem*;
- Percorre a árvore direita *pós-ordem*;
- Visita a raiz.

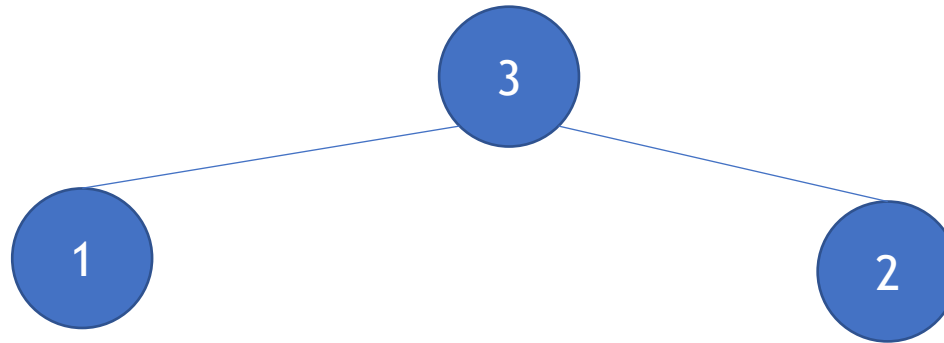
Percurso pós-ordem



Como seria o percurso em pós-ordem aqui?

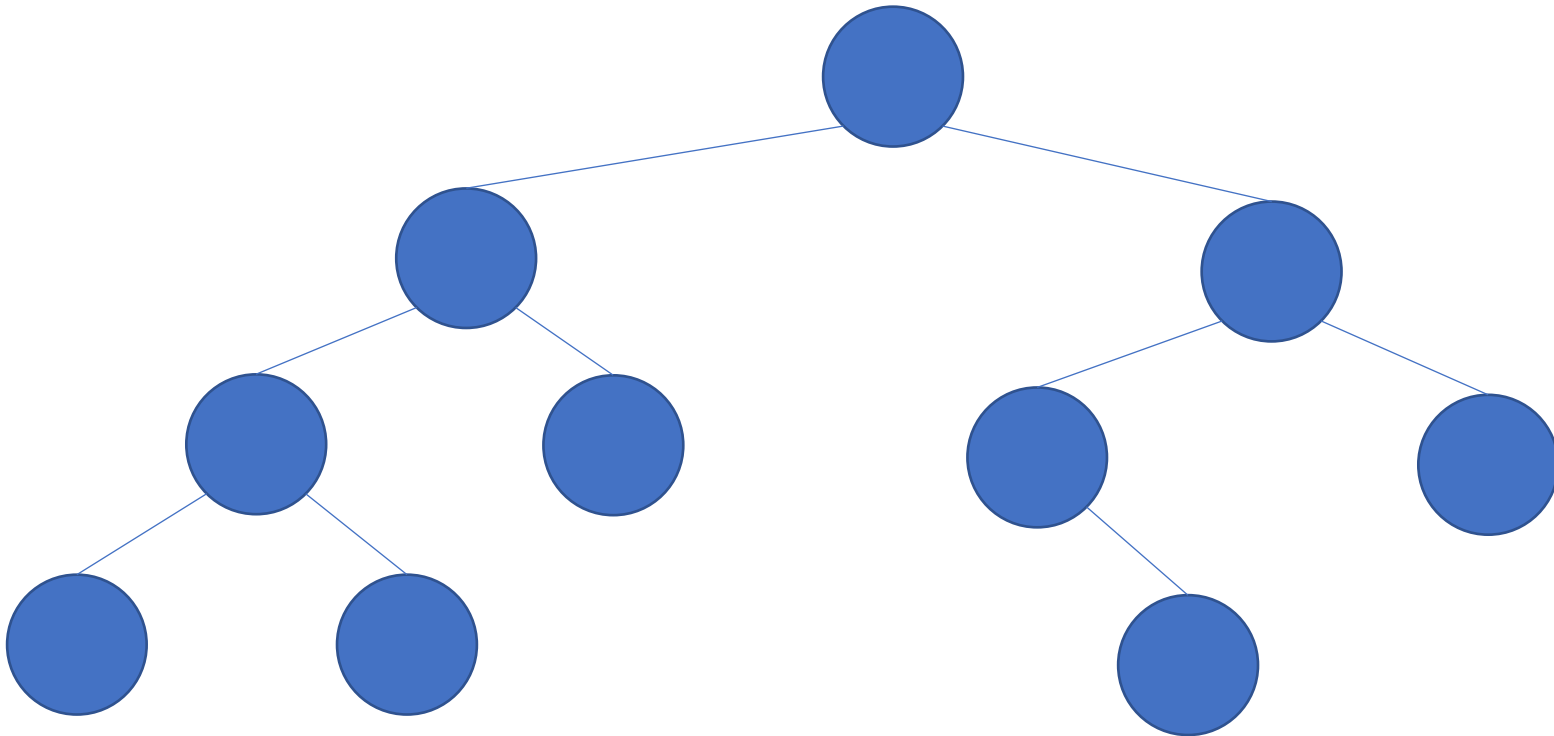
Os números representam a
ordem em que cada nó é visitado

Percurso pós-ordem



Como seria o percurso em pós-ordem aqui?

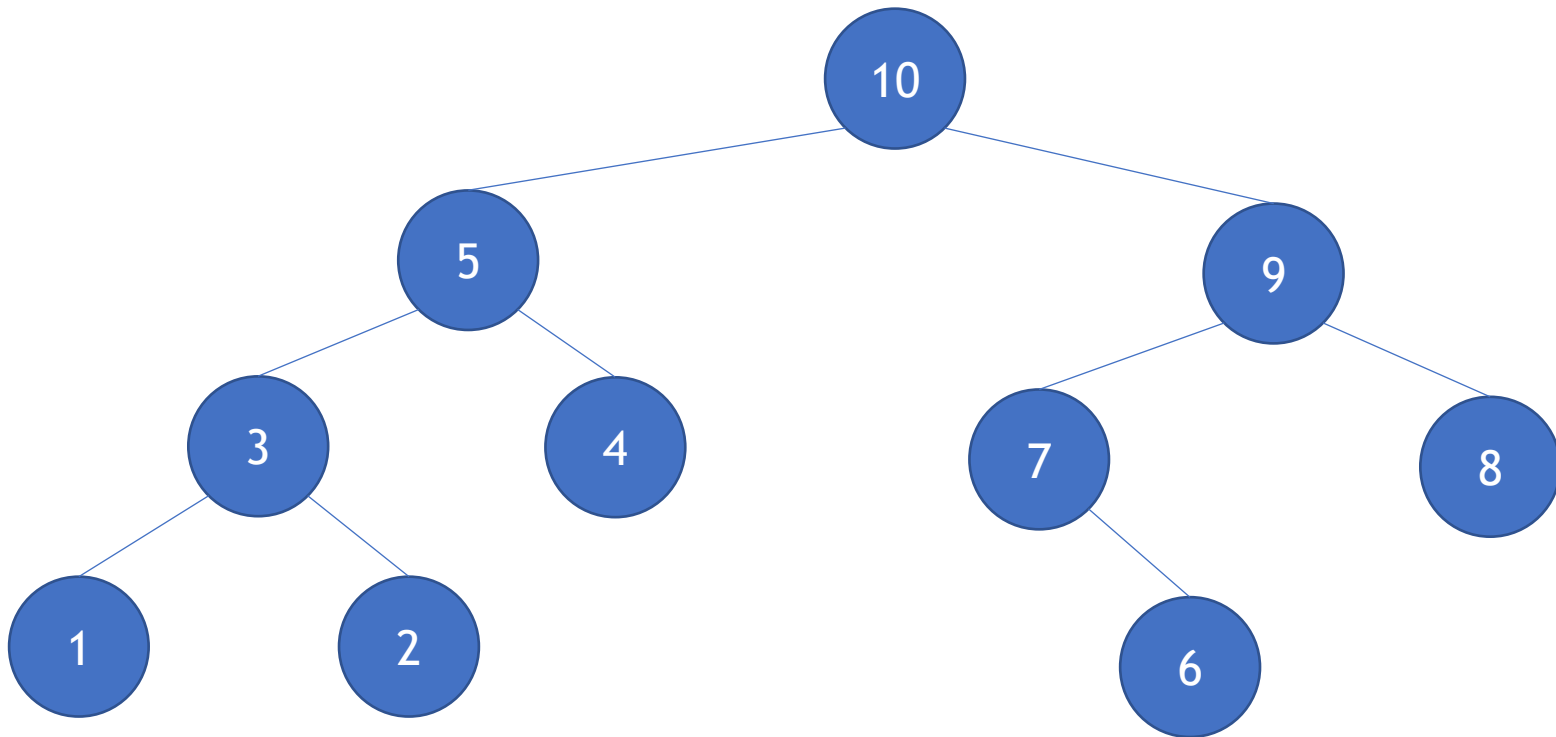
Percurso pós-ordem



Como seria o percurso em pós-ordem aqui?

Os números representam a ordem em que cada nó é visitado

Percurso pós-ordem



Liberar árvore da memória

- Para liberar uma árvore, é necessário **percorrer toda a árvore** e liberar cada um dos nós.
- Qual dos três percursos que vimos seria o melhor para liberar a árvore?

Liberar árvore da memória

- Percurso em pós-ordem para liberar árvore da memória:

```
void liberar_arvore(NoArvore *raiz) {  
    if (raiz == NULL) return;  
    liberar_arvore(raiz->esq);  
    liberar_arvore(raiz->dir);  
    free(raiz);  
}
```

Referências

- Slides do Prof. Monael Pinheiro Riberio:
 - <https://sites.google.com/site/aed2019q1/>
- Slides da Profa. Mirtha Lina Fernández Venero
 - Algoritmos e Estruturas de Dados I - 2019
- Slides do Prof. Jesús P. Mena-Chalco:
 - <http://professor.ufabc.edu.br/~jesus.mena/courses/aed1-1q-2019/>

Bibliografia básica

- PINHEIRO, F. A. C. Elementos de programação em C. Porto Alegre, RS: Bookman, 2012.
- FORBELLONE, A. L. V.; EBERSPACHER, H. F. Lógica de programação: a construção de algoritmos e estruturas de dados. 3ª edição. São Paulo, SP: Prentice Hall, 2005.
- CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. Algoritmos: teoria e prática. 2ª edição. Rio de Janeiro, RJ: Campus, 2002.

Bibliografia complementar

- AGUILAR, L. J. Programação em C++: algoritmos, estruturas de dados e objetos. São Paulo, SP: McGraw-Hill, 2008.
- DROZDEK, A. Estrutura de dados e algoritmos em C++. São Paulo, SP: Cengage Learning, 2009.
- KNUTH D. E. The art of computer programming. Upper Saddle River, USA: Addison- Wesley, 2005.
- SEDGEWICK, R. Algorithms in C++: parts 1-4: fundamentals, data structures, sorting, searching. Reading, USA: Addison-Wesley, 1998.
- SZWARCFITER, J. L.; MARKENZON, L. Estruturas de dados e seus algoritmos. 3a edição. Rio de Janeiro, RJ: LTC, 1994.
- TEWNENBAUM, A. M.; LANGSAM, Y.; AUGENSTEIN, M. J. Estruturas de dados usando C. São Paulo, SP: Pearson Makron Books, 1995.