Algoritmos e Estrutura de Dados II

Prof. Fellipe Guilherme Rey de Souza

Aula 10 –Árvore Binária (Implementação)

Agenda

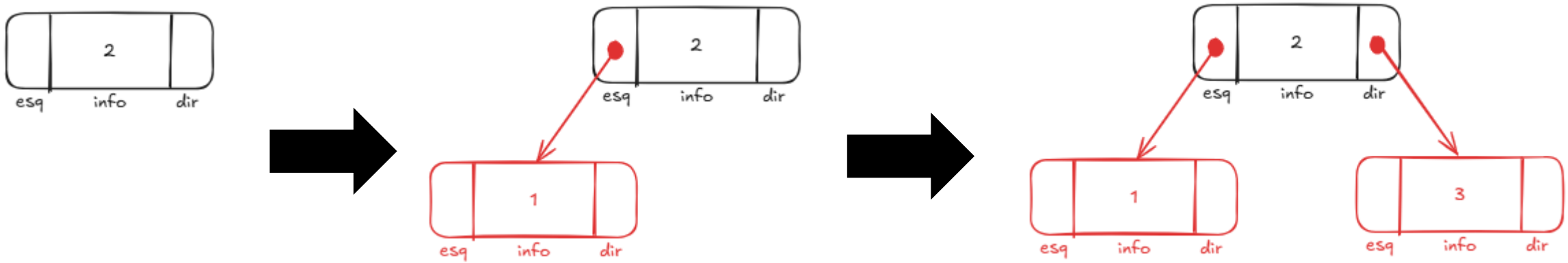
- Definição
- Inserção
- Remoção
- Busca
- Percurso
- Tamanho
- Altura

Operações em Árvore – Definição

- Semelhante à forma que vimos nas aulas anteriores (Pilha, Fila, Lista e Árvore), a estrutura para árvore usando alocação dinâmica de memória se utilizará de três campos:
  - **info:** representando o valor do nó da árvore
  - **esq:** representando um ponteiro para a subárvore à esquerda
  - **dir:** representando um ponteiro para a subárvore à direita

- **PS**: Não iremos tratar da implementação usando vetores (ruim e complexa).

Operações em Árvore – Definição



- Para conectar novos nós à árvore, precisamos analisar o conteúdo de info para sabermos se vamos adicionar um ponteiro para a esquerda ou para a direita.

Operações em Árvore – Definição

```c
#define TRUE 1
#define FALSE 0

typedef struct no {
    int info;
    struct no *esq;
    struct no *dir;
} Node;
```

- Primeiramente definimos os valores de booleano em C: TRUE(1) E FALSE(0).

- Nossa struct no possui um info (do tipo inteiro) e dois ponteiros para a esquerda e para a direita, que apontarão para as subárvores esquerda e direita (respectivamente).

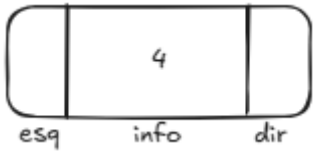
Operações em Árvore – Inserção

- Utilizando a estrutura anterior (struct no), vamos realizar um exemplo de inserção na nossa árvore binária de busca.

- Para isso, vamos inserior os seguintes elementos (em ordem): 4, 6, 7, 2, 3, 1 e 5.

Operações em Árvore – Inserção

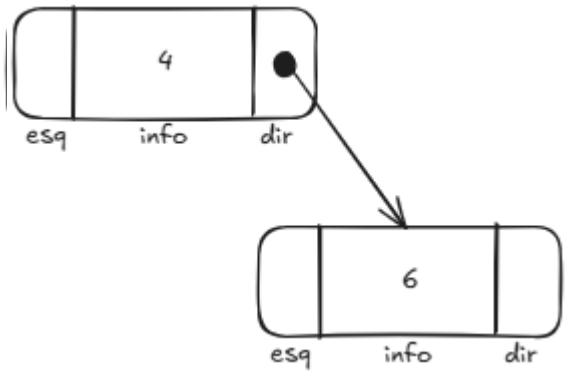
- Inserir (4, 6, 7, 2, 3, 1 e 5):
  - Ação: Inserir o número 4



Subárvore esquerda: Valores menores que o pai
Subárvore direita: Valores maiores que o pai

Operações em Árvore – Inserção

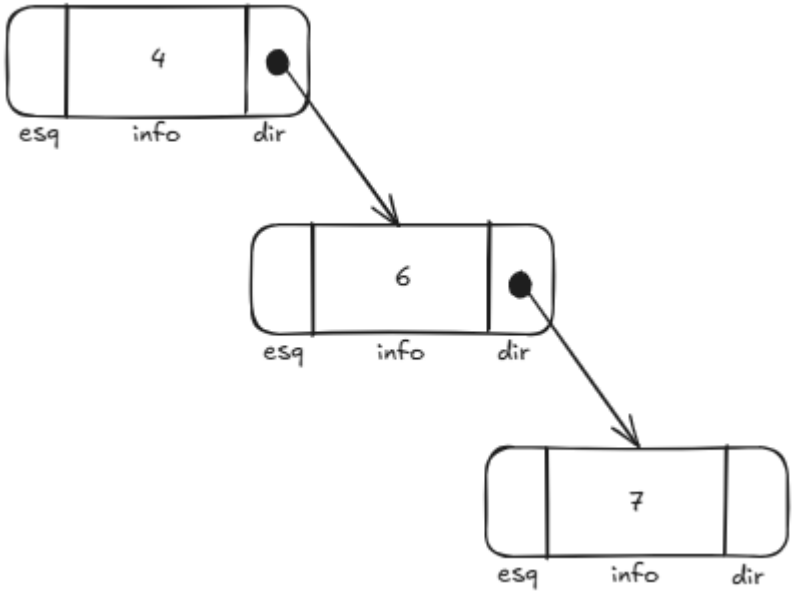
- Inserir (4, 6, 7, 2, 3, 1 e 5):
  - ~~Ação: Inserir o número 4~~
  - Ação: Inserir o número 6



Subárvore esquerda: Valores menores que o pai
Subárvore direita: Valores maiores que o pai

Operações em Árvore – Inserção

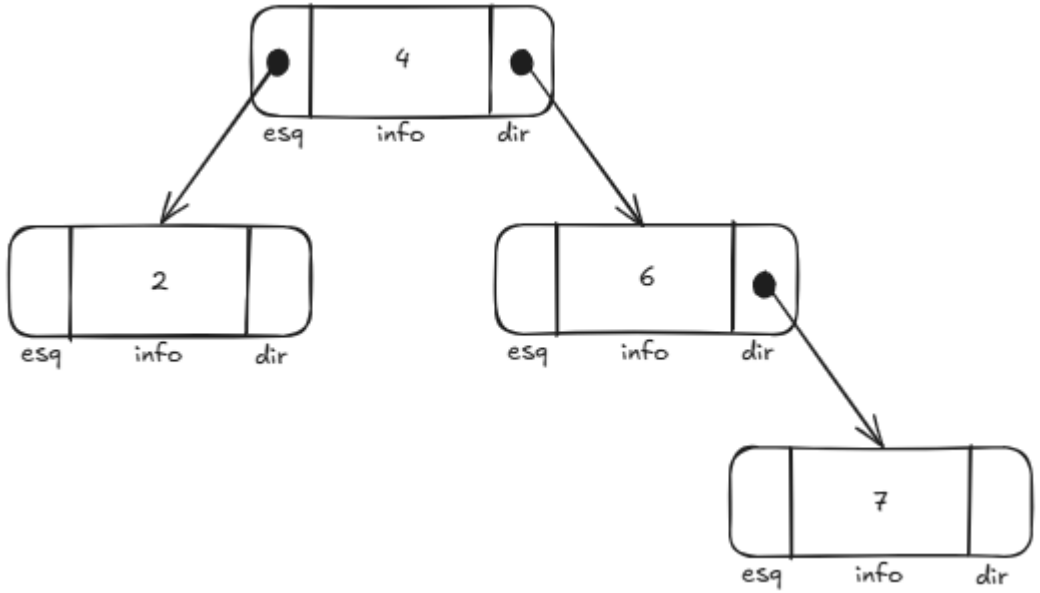
- Inserir (4, 6, 7, 2, 3, 1 e 5):
  - ~~Ação: Inserir o número 4~~
  - ~~Ação: Inserir o número 6~~
  - Ação: Inserir o número 7



Subárvore esquerda: Valores menores que o pai
Subárvore direita: Valores maiores que o pai

Operações em Árvore – Inserção

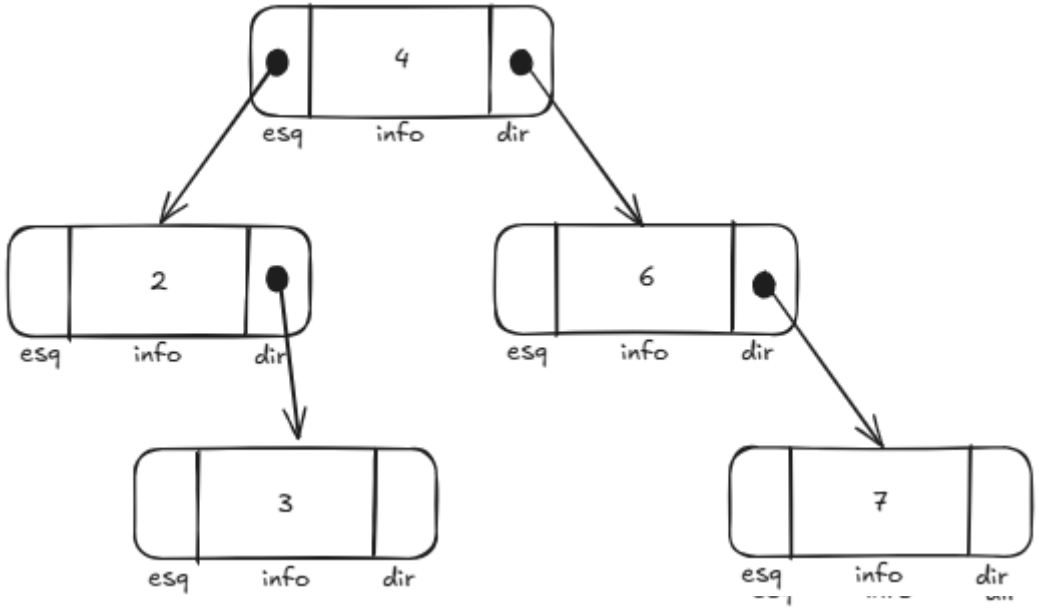
- Inserir (4, 6, 7, 2, 3, 1 e 5):
  - ~~Ação: Inserir o número 4~~
  - ~~Ação: Inserir o número 6~~
  - ~~Ação: Inserir o número 7~~
  - Ação: Inserir o número 2



Subárvore esquerda: Valores menores que o pai
Subárvore direita: Valores maiores que o pai

Operações em Árvore – Inserção

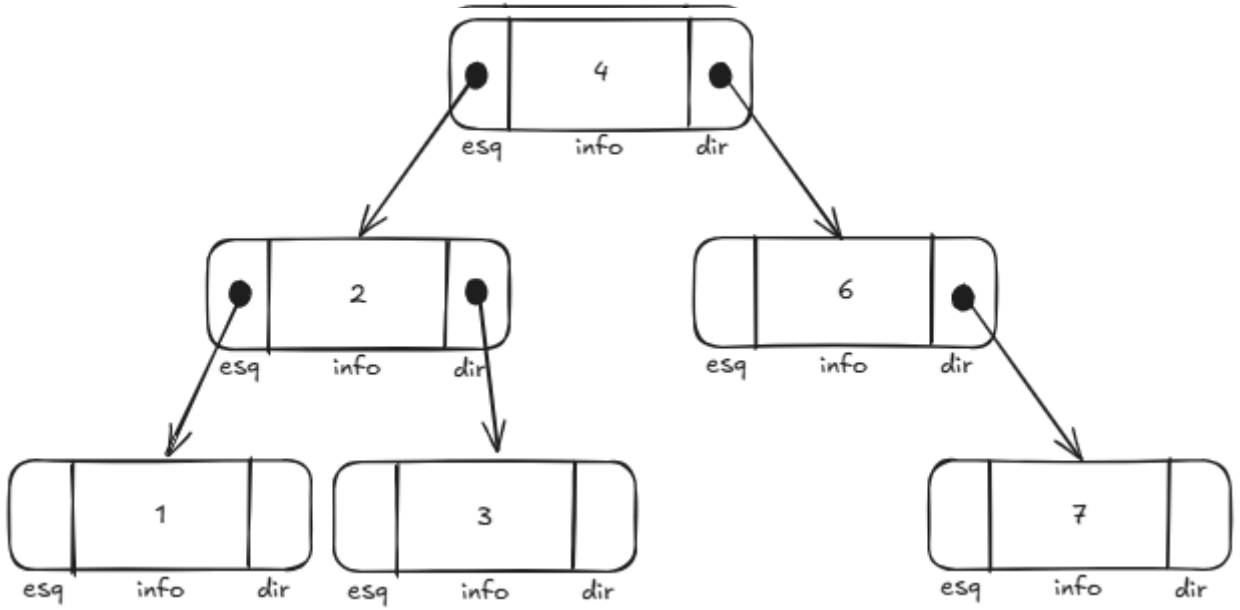
- Inserir (4, 6, 7, 2, 3, 1 e 5):
  - ~~Ação: Inserir o número 4~~
  - ~~Ação: Inserir o número 6~~
  - ~~Ação: Inserir o número 7~~
  - ~~Ação: Inserir o número 2~~
  - Ação: Inserir o número 3



Subárvore esquerda: Valores menores que o pai
Subárvore direita: Valores maiores que o pai

Operações em Árvore – Inserção

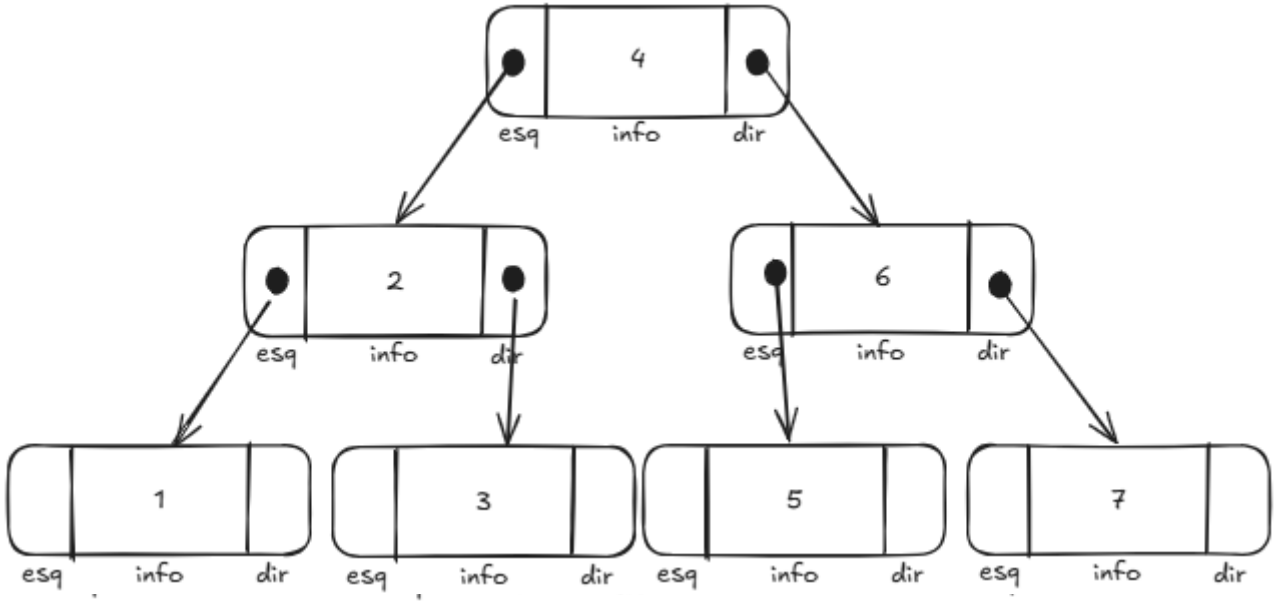
- Inserir (4, 6, 7, 2, 3, 1 e 5):
  - ~~Ação: Inserir o número 4~~
  - ~~Ação: Inserir o número 6~~
  - ~~Ação: Inserir o número 7~~
  - ~~Ação: Inserir o número 2~~
  - ~~Ação: Inserir o número 3~~
  - Ação: Inserir o número 1



Subárvore esquerda: Valores menores que o pai
Subárvore direita: Valores maiores que o pai

Operações em Árvore – Inserção

- Inserir (4, 6, 7, 2, 3, 1 e 5):
  - ~~Ação: Inserir o número 4~~
  - ~~Ação: Inserir o número 6~~
  - ~~Ação: Inserir o número 7~~
  - ~~Ação: Inserir o número 2~~
  - ~~Ação: Inserir o número 3~~
  - ~~Ação: Inserir o número 1~~
  - Ação: Inserir o número 5



Subárvore esquerda: Valores menores que o pai
Subárvore direita: Valores maiores que o pai

Operações em Árvore – Inserção

- Se a árvore (ou subárvore) estiver vazia (`raiz == NULL`), um novo nó é criado com o valor dado, e seus ponteiros `esq` e `dir` são definidos como `NULL`.

- Caso contrário, a função compara o valor a ser inserido com o valor do nó atual.

```c
Node* insere(Node* raiz, int valor) {
    if (raiz == NULL) {
        Node* node = (Node*) malloc(sizeof(Node));
        node->info = valor;
        node->esq = NULL;
        node->dir = NULL;
        return node;
    }
    if (valor < raiz->info) {
        raiz->esq = insere(raiz->esq, valor);
    }
    else if (valor > raiz->info) {
        raiz->dir = insere(raiz->dir, valor);
    }
    return raiz;
}
```

Operações em Árvore – Inserção

- Se o valor for menor, a função é chamada recursivamente para a subárvore esquerda; se for maior, a função é chamada recursivamente para a subárvore direita.

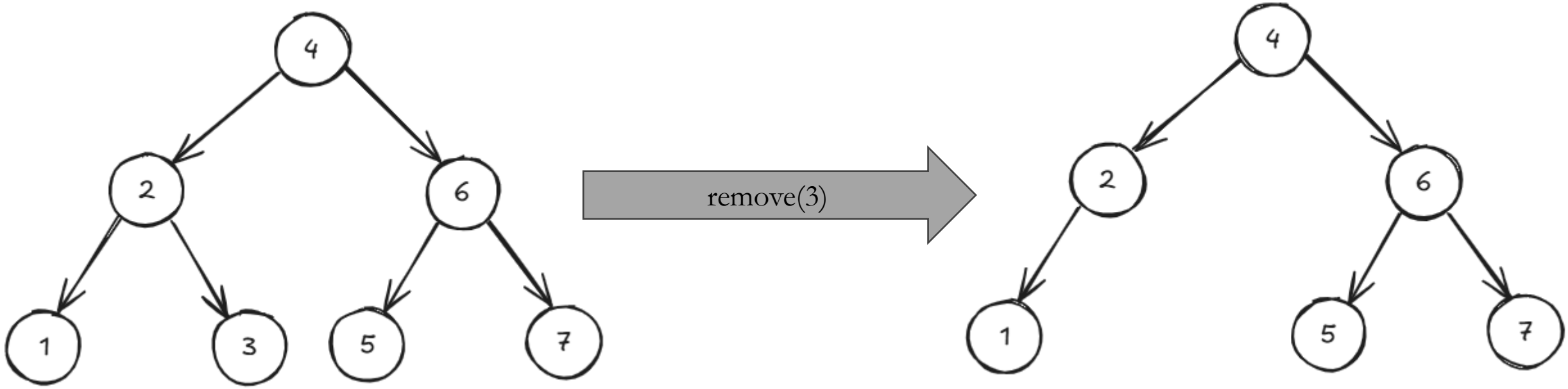
- Após a inserção, o nó atual (raiz ou subárvore) é retornado, garantindo que a árvore seja corretamente atualizada.

```c
Node* insere(Node* raiz, int valor) {
    if (raiz == NULL) {
        Node* node = (Node*) malloc(sizeof(Node));
        node->info = valor;
        node->esq = NULL;
        node->dir = NULL;
        return node;
    }
    if (valor < raiz->info) {
        raiz->esq = insere(raiz->esq, valor);
    }
    else if (valor > raiz->info) {
        raiz->dir = insere(raiz->dir, valor);
    }
    return raiz;
}
```

Operações em Árvore – Remoção

i. Remoção de um nó sem filhos

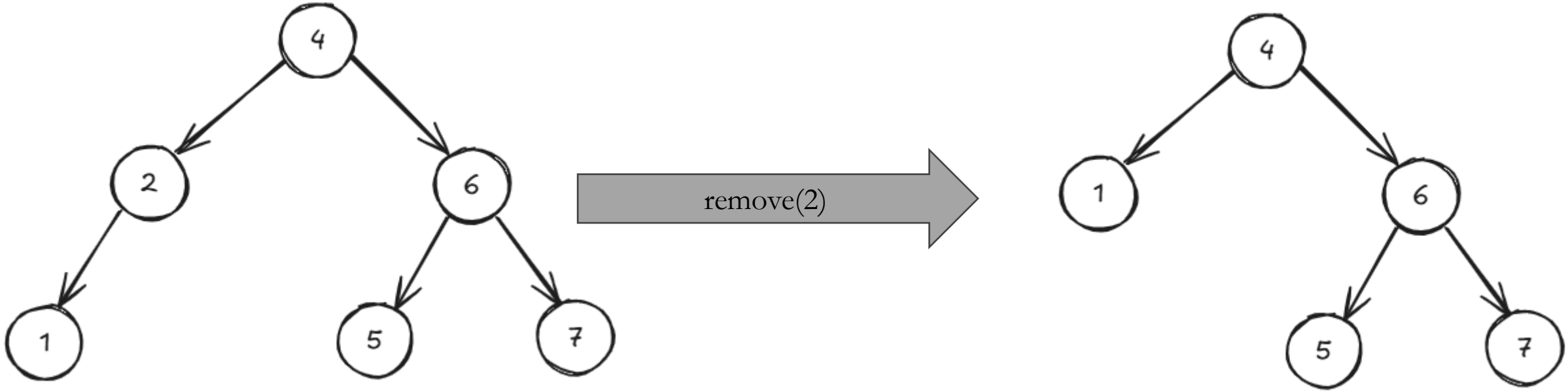
- Se o nó que você quer remover não tem filhos (é uma folha), você simplesmente pode remover o nó sem mais alterações, pois ele não afeta a estrutura da árvore.



Operações em Árvore – Remoção

ii. Remoção de um nó com um filho

- Se o nó a ser removido tem apenas um filho, basta substituir o nó pelo seu único filho. Qualquer referência ao nó removido é redirecionada para o filho.



Operações em Árvore – Remoção

iii. Remoção de um nó com dois filhos

- Este é o caso mais complexo, e a remoção exige uma estratégia especial para garantir que a árvore de busca binária continue válida.

- Quando um nó tem dois filhos, você precisa encontrar um valor que possa substituir o nó removido sem violar as propriedades da árvore de busca binária.

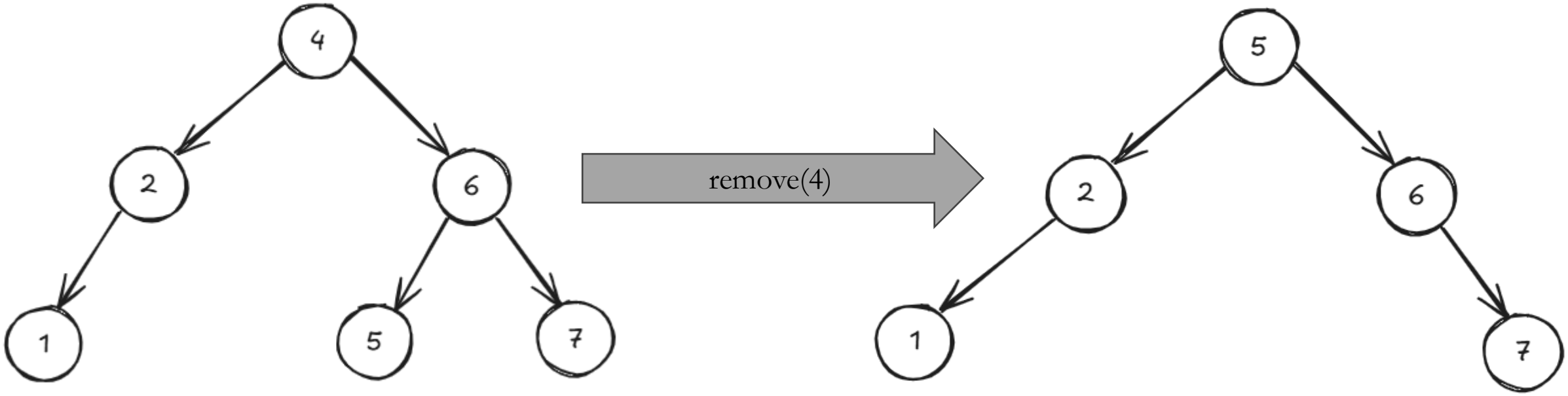
Operações em Árvore – Remoção

iii. Remoção de um nó com dois filhos (*cont.*)

- O valor substituto pode ser:
  - O maior valor da subárvore esquerda (o nó mais à direita da subárvore esquerda), ou;
  - O menor valor da subárvore direita (o nó mais à esquerda da subárvore direita).

- A escolha mais comum é o sucessor **em-ordem** (o menor valor da subárvore direita).

Operações em Árvore – Remoção



Em-ordem: 1, 2, 4, 5, 6, 7

Operações em Árvore – Remoção

```c
Node* minimo(Node* raiz) {
    while (raiz != NULL && raiz->esq != NULL) {
        raiz = raiz->esq;
    }
    return raiz;
}
```

- A função **minimo** tem como objetivo encontrar o nó com o menor valor em uma árvore binária de busca.
- Ela começa com o nó raiz e, em seguida, percorre recursivamente a subárvore esquerda, já que, em uma árvore binária de busca, o menor valor está sempre na extremidade esquerda.

Operações em Árvore – Remoção

```c
Node* minimo(Node* raiz) {
    while (raiz != NULL && raiz->esq != NULL) {
        raiz = raiz->esq;
    }
    return raiz;
}
```

- A função continua movendo-se para a esquerda enquanto houver um nó filho esquerdo, até chegar a um nó que não tem mais filhos à esquerda, ou seja, o nó com o valor mínimo.
- Quando esse nó é encontrado, ele é retornado como o nó mínimo da árvore ou da subárvore à qual a função foi aplicada.

Operações em Árvore – Remoção

- Começamos verificando se a árvore está vazia, retornando `NULL` nesse caso.
- Em seguida, ela recursivamente desce pela árvore para encontrar o nó a ser removido, comparando o valor a ser removido com o valor do nó atual, indo para a subárvore esquerda ou direita conforme necessário.

```
Node* removeNode(Node* raiz, int valor) {
    if (raiz == NULL) {
        return raiz;
    }
    if (valor < raiz->info) {
        raiz->esq = removeNode(raiz->esq, valor);
    }
    else if (valor > raiz->info) {
        raiz->dir = removeNode(raiz->dir, valor);
    }
    else {
        if (raiz->esq == NULL) {
            Node* temp = raiz->dir;
            free(raiz);
            return temp;
        }
        else if (raiz->dir == NULL) {
            Node* temp = raiz->esq;
            free(raiz);
            return temp;
        }
        Node* temp = minimo(raiz->dir);
        raiz->info = temp->info;
        raiz->dir = removeNode(raiz->dir, temp->info);
    }
    return raiz;
}
```

Operações em Árvore – Remoção

- Quando encontra o nó a ser removido, o comportamento depende da quantidade de filhos do nó:
  - Se o nó não tem filhos, ele é simplesmente removido;
  - Se tem um filho, o nó é substituído pelo seu único filho.

```c
Node* removeNode(Node* raiz, int valor) {
    if (raiz == NULL) {
        return raiz;
    }
    if (valor < raiz->info) {
        raiz->esq = removeNode(raiz->esq, valor);
    }
    else if (valor > raiz->info) {
        raiz->dir = removeNode(raiz->dir, valor);
    }
    else {
        if (raiz->esq == NULL) {
            Node* temp = raiz->dir;
            free(raiz);
            return temp;
        }
        else if (raiz->dir == NULL) {
            Node* temp = raiz->esq;
            free(raiz);
            return temp;
        }
        Node* temp = minimo(raiz->dir);
        raiz->info = temp->info;
        raiz->dir = removeNode(raiz->dir, temp->info);
    }
    return raiz;
}
```

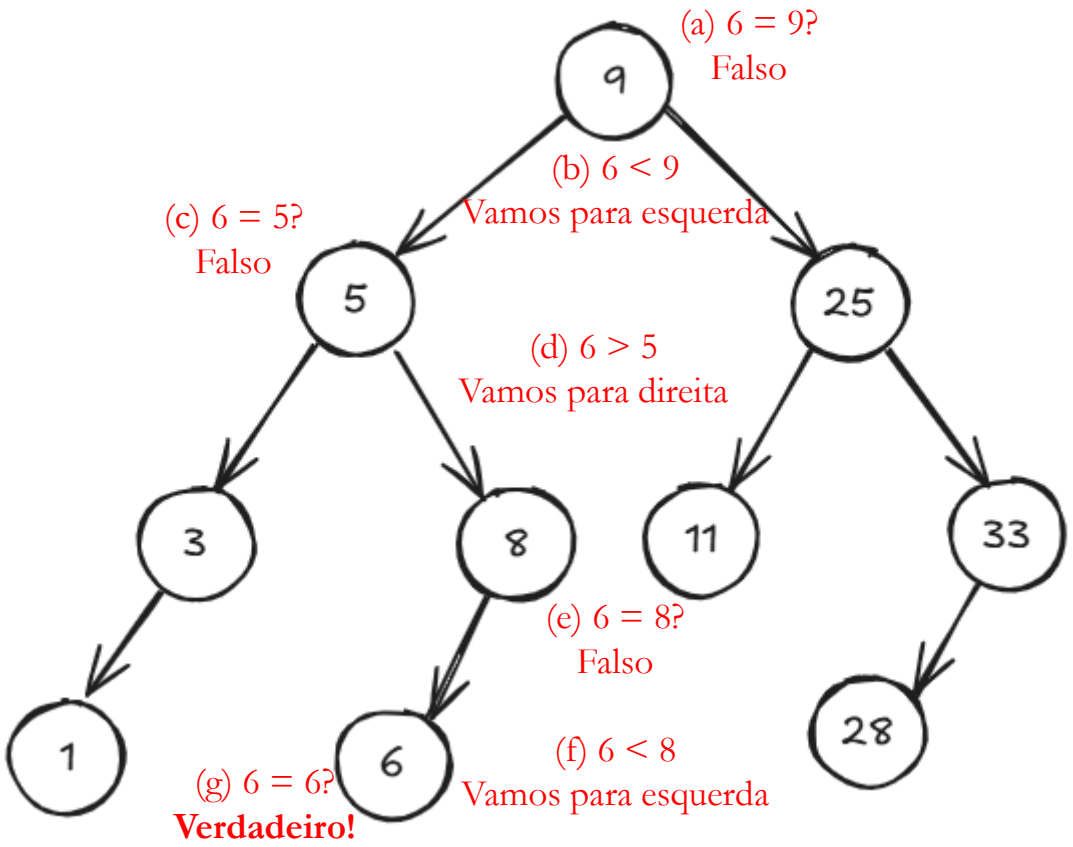
Operações em Árvore – Remoção

- Se tem dois filhos, o nó é substituído pelo menor valor da subárvore direita (encontrado pela função `minimo`), e a função é chamada novamente para remover esse valor mínimo da subárvore direita.
- Por fim, a árvore modificada é retornada.

```
Node* removeNode(Node* raiz, int valor) {
    if (raiz == NULL) {
        return raiz;
    }
    if (valor < raiz->info) {
        raiz->esq = removeNode(raiz->esq, valor);
    }
    else if (valor > raiz->info) {
        raiz->dir = removeNode(raiz->dir, valor);
    }
    else {
        if (raiz->esq == NULL) {
            Node* temp = raiz->dir;
            free(raiz);
            return temp;
        }
        else if (raiz->dir == NULL) {
            Node* temp = raiz->esq;
            free(raiz);
            return temp;
        }
        Node* temp = minimo(raiz->dir);
        raiz->info = temp->info;
        raiz->dir = removeNode(raiz->dir, temp->info);
    }
    return raiz;
}
```

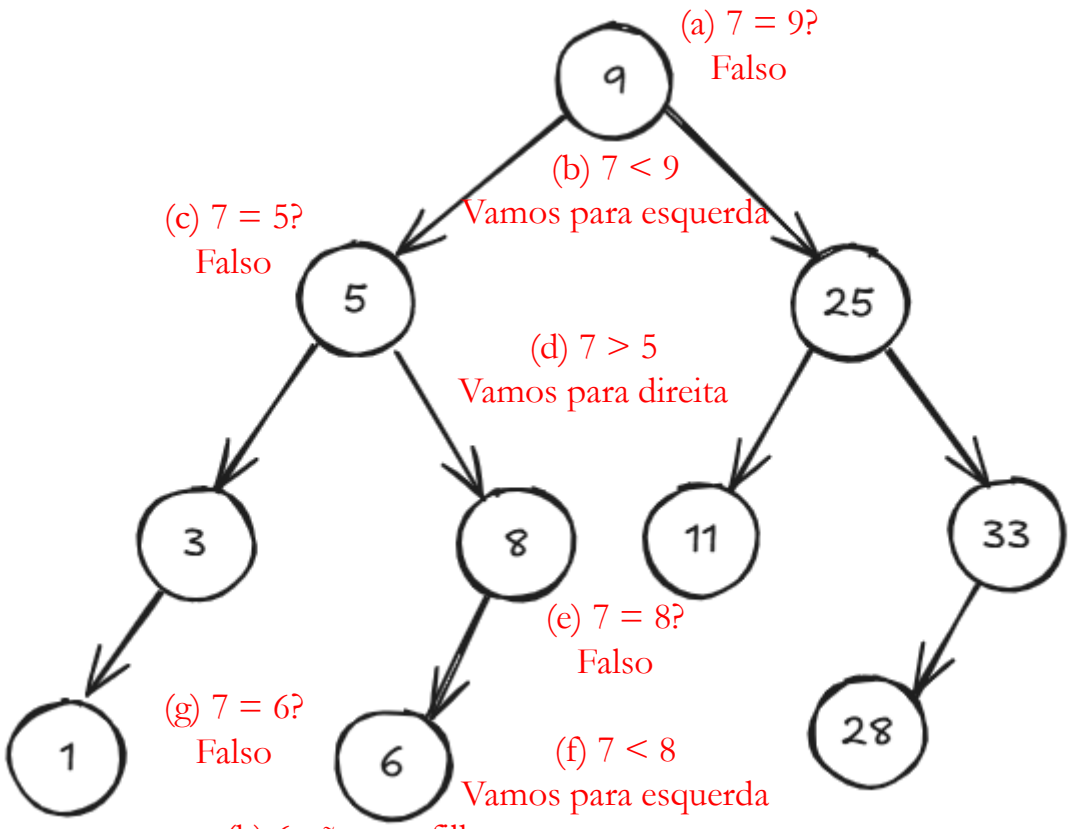
Operações em Árvore – Busca

- Exemplo 01: Buscar o elemento 6 na árvore



Operações em Árvore – Busca

- Exemplo 02: Buscar o elemento 7 na árvore

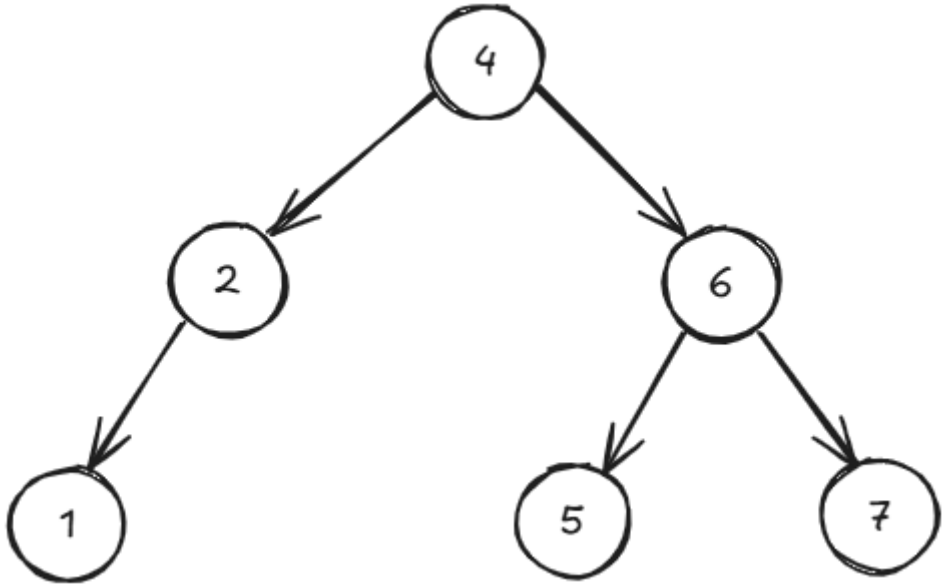


Operações em Árvore – Busca

- Caso a raiz seja nula, quer dizer que o item não existe em nossa árvore.
- Caso o valor procurado seja igual ao conteúdo da raiz, então retornaremos TRUE (1).
- Caso o valor procurado seja menor que o valor da raiz, procuraremos na subárvore esquerda.
- Caso contrário (o valor procurado seja maior que o valor da raiz), procuraremos na subárvore direita.

```
int busca(Node* raiz, int valor) {
    if (raiz == NULL) {
        return FALSE;
    }
    if (raiz->info == valor) {
        return TRUE;
    }
    if (valor < raiz->info) {
        return busca(raiz->esq, valor);
    }
    return busca(raiz->dir, valor);
}
```

Operações em Árvore – Percurso



Pré-ordem: 4, 2, 1, 6, 5, 7

Em-ordem: 1, 2, 4, 5, 6, 7

Pós-ordem: 1, 2, 5, 7, 6, 4

Operações em Árvore – Percurso

- Para a função **preOrdem**, caso a raiz não seja nula, printaremos primeiramente o valor do conteúdo e depois percorreremos de forma recursiva em pré ordem a raiz na esquerda e posteriormente percorreremos de forma recursiva a raiz na direita.

```
void preOrdem(Node* raiz) {
    if (raiz != NULL) {
        printf("%d ", raiz->info);
        preOrdem(raiz->esq);
        preOrdem(raiz->dir);
    }
}

void emOrdem(Node* raiz) {
    if (raiz != NULL) {
        emOrdem(raiz->esq);
        printf("%d ", raiz->info);
        emOrdem(raiz->dir);
    }
}

void posOrdem(Node* raiz) {
    if (raiz != NULL) {
        posOrdem(raiz->esq);
        posOrdem(raiz->dir);
        printf("%d ", raiz->info);
    }
}
```

Operações em Árvore – Percurso

- Para a função **emOrdem**, caso a raiz não seja nula, primeiramente percorreremos de forma recursiva em em ordem a raiz na esquerda e depois printaremos o valor do conteúdo e posteriormente percorreremos de forma recursiva a raiz na direita.

```c
void preOrdem(Node* raiz) {
    if (raiz != NULL) {
        printf("%d ", raiz->info);
        preOrdem(raiz->esq);
        preOrdem(raiz->dir);
    }
}

void emOrdem(Node* raiz) {
    if (raiz != NULL) {
        emOrdem(raiz->esq);
        printf("%d ", raiz->info);
        emOrdem(raiz->dir);
    }
}

void posOrdem(Node* raiz) {
    if (raiz != NULL) {
        posOrdem(raiz->esq);
        posOrdem(raiz->dir);
        printf("%d ", raiz->info);
    }
}
```

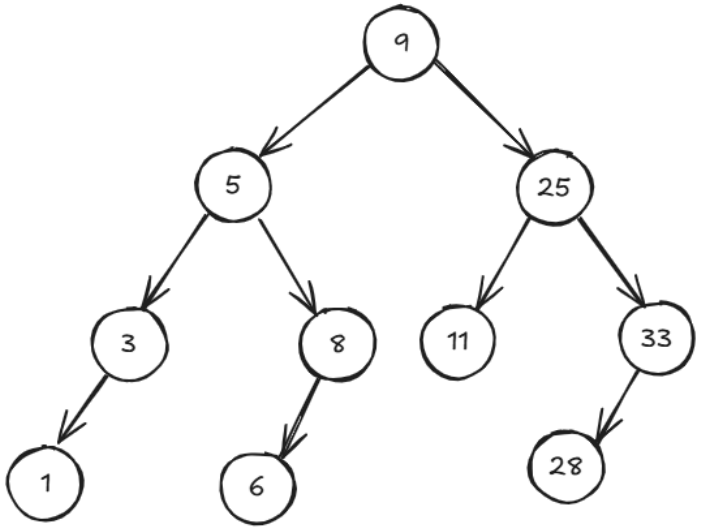
Operações em Árvore – Percurso

- Para a função **posOrdem**, caso a raiz não seja nula, primeiramente percorreremos de forma recursiva em pós ordem a raiz na esquerda e depois percorreremos de forma recursiva a raiz na direita e posteriormente printaremos o valor do conteúdo.

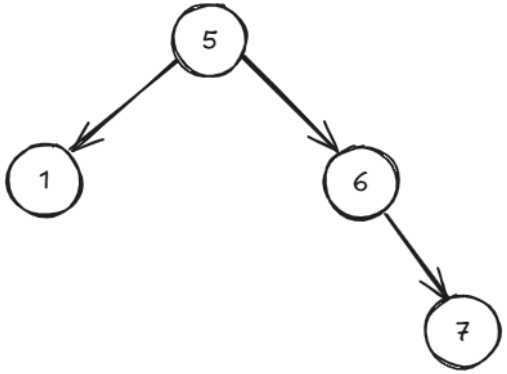
```c
void preOrdem(Node* raiz) {
    if (raiz != NULL) {
        printf("%d ", raiz->info);
        preOrdem(raiz->esq);
        preOrdem(raiz->dir);
    }
}

void emOrdem(Node* raiz) {
    if (raiz != NULL) {
        emOrdem(raiz->esq);
        printf("%d ", raiz->info);
        emOrdem(raiz->dir);
    }
}

void posOrdem(Node* raiz) {
    if (raiz != NULL) {
        posOrdem(raiz->esq);
        posOrdem(raiz->dir);
        printf("%d ", raiz->info);
    }
}
```

Operações em Árvore – Tamanho

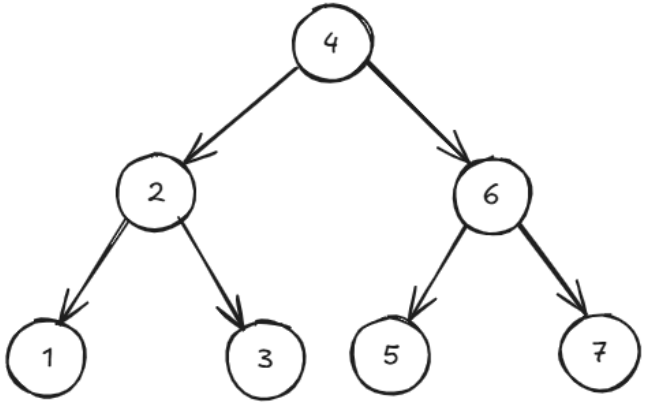
- O tamanho de uma árvore binária de busca é definido pelo número total de nós que uma árvore possui.



Tamanho: 10



Tamanho: 4



Tamanho: 7

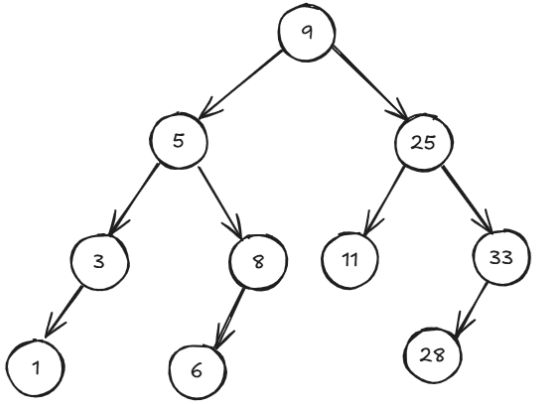
Operações em Árvore – Tamanho

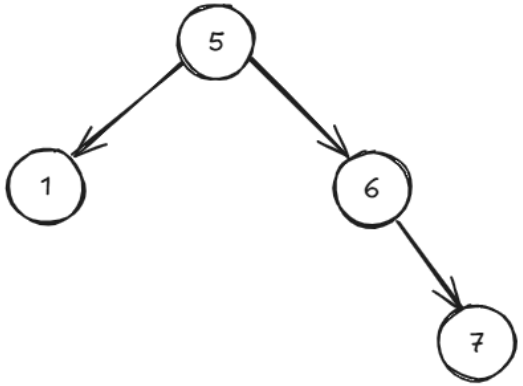

```c
int tamanho(Node* raiz) {
    if (raiz == NULL) {
        return 0;
    }
    return 1 + tamanho(raiz->esq) + tamanho(raiz->dir);
}
```

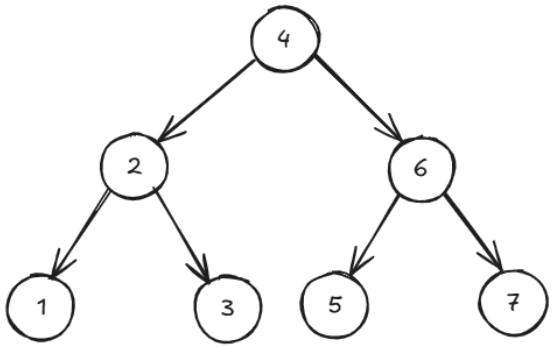
- Para calcular o tamanho, primeiro verificamos se a raiz é nula. Se for, retornamos 0.
- Caso contrário, retornamos 1 somado à recursão da chamada da função **tamanho** (primeiro ao lado esquerdo) somando à recursão da chamada da função **tamanho** (lado direito).

Operações em Árvore – Altura

NULL







Altura: 0

Altura: 4

Altura: 3

Altura: 3

Operações em Árvore – Altura

```c
int altura(Node* raiz) {
    if (raiz == NULL) {
        return 0;
    }
    int altura_esq = altura(raiz->esq);
    int altura_dir = altura(raiz->dir);

    return 1 + (altura_esq > altura_dir ? altura_esq : altura_dir);
}
```

- Para calcular a altura, vemos inicialmente se a raiz é nula. Se for, retornamos 0.
- Caso a raiz não seja nula, calculamos a altura da esquerda, da direita e retornamos 1 + maior valor entre alturas esquerda e direita.

Algoritmos e Estrutura de Dados II

Prof. Fellipe Guilherme Rey de Souza

Aula 10 –Árvore Binária (Implementação)