Estruturas de Dados 2

aula 04 - Árvores e Árvores Binárias de Busca - BSTs

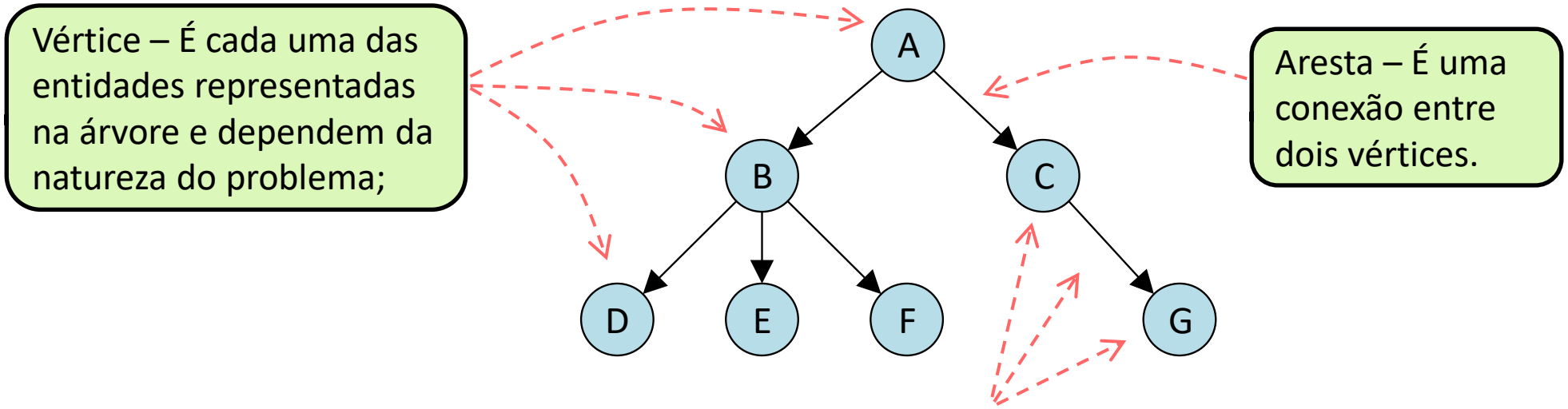
Antonio Angelo de Souza Tartaglia

angelot@ifsp.edu.br

Estrutura de Dados 2

Árvores

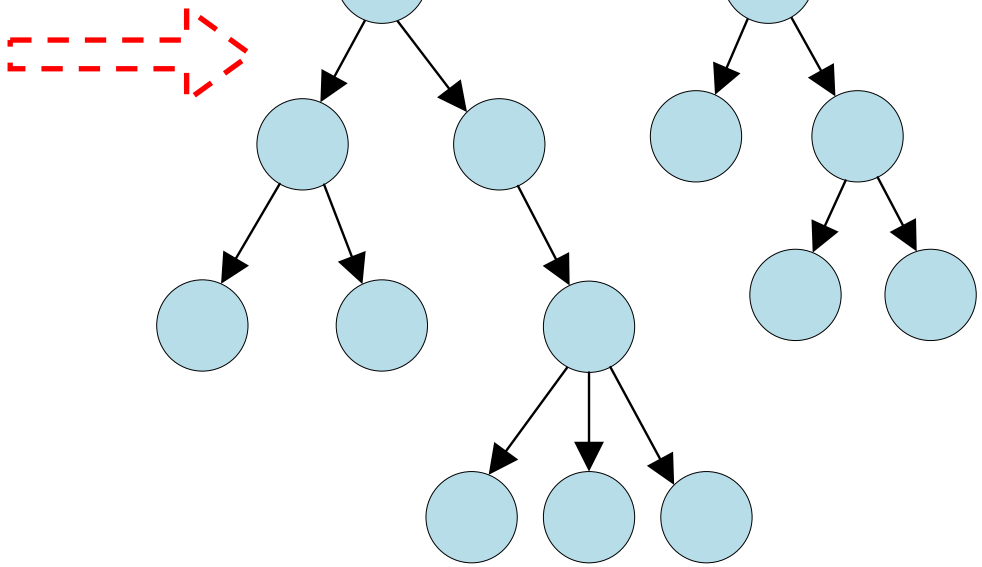
- Definição de árvores:
  - São um tipo especial de Grafo, são definidas como um conjunto não vazio de vértices (ou nós) e arestas que satisfazem certos requisitos para se conectarem;

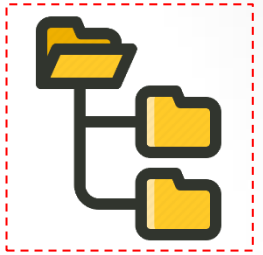


- Qualquer par de vértices está conectado a apenas uma aresta;
- Grafo Conexo: existe exatamente um caminho entre quaisquer dois de seus vértices, e é acíclico, não possui ciclos

Estrutura de Dados 2

Árvores





Árvores

- Aplicações:
  - Árvores são adequadas para representar estruturas hierárquicas não lineares;
- Exemplos:
  - Relações de descendências (pai, filho, etc.);
  - Diagrama hierárquico de uma organização;
  - Campeonatos de modalidades desportivas;
  - Taxonomia.

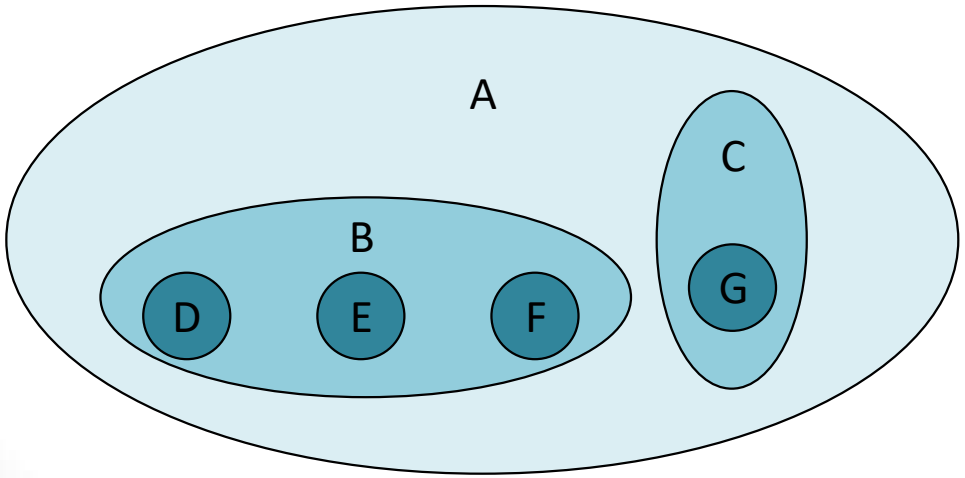
Ciência que estuda os seres vivos

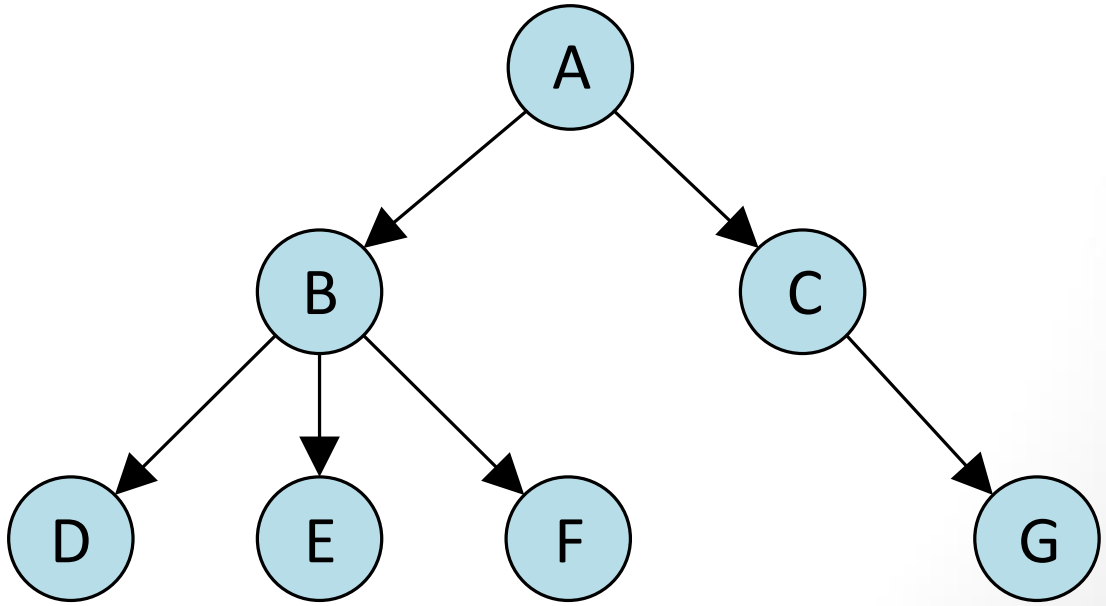
- Em computação:
  - Estrutura de Diretórios (pastas);
  - Busca de dados armazenados no computador;
  - Representação de espaço de soluções (Ex. jogo de xadrez);
  - Modelagem de algoritmos.

Estrutura de Dados 2

Árvores

- Formas de representação:
  - Grafos – é a mais comum;
  - Diagrama de Venn – conjuntos aninhados.





Árvores

- Existem vários tipos de árvores em computação desenvolvidas para diferentes tipos de aplicações:

  - Árvore Binária de Busca;
  - Árvore AVL;
  - Árvore Rubro-Negra;
  - Árvore B+;
  - Árvore 2 – 3;
  - Árvore 2 – 3 – 4;
  - Quadtree;
  - Octree;
  - Trie, P.A.T.R.I.C.I.A.;
  - Fenwick
  - Etc.

Árvores Binárias de Busca que são balanceadas para otimização da busca

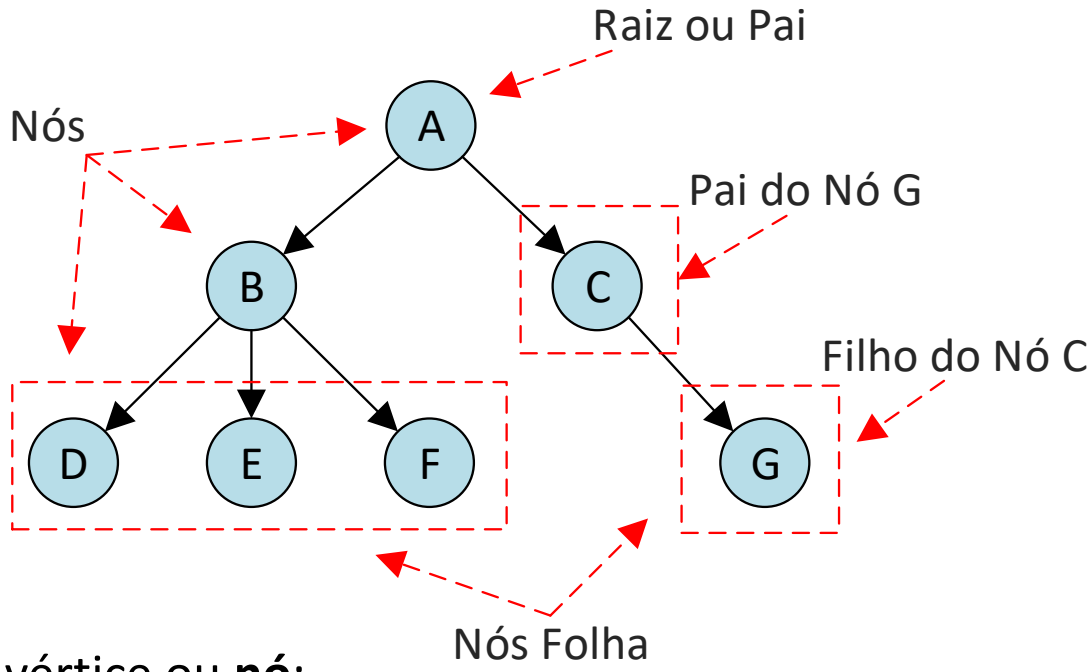
Usadas em programas de segmentação, imagem, volumes 3D, detecção de colisão, jogos, etc.

A escolha do tipo de árvore a ser utilizada, depende sempre da aplicação a ser desenvolvida.

Estrutura de Dados 2

Árvores

- Propriedades:

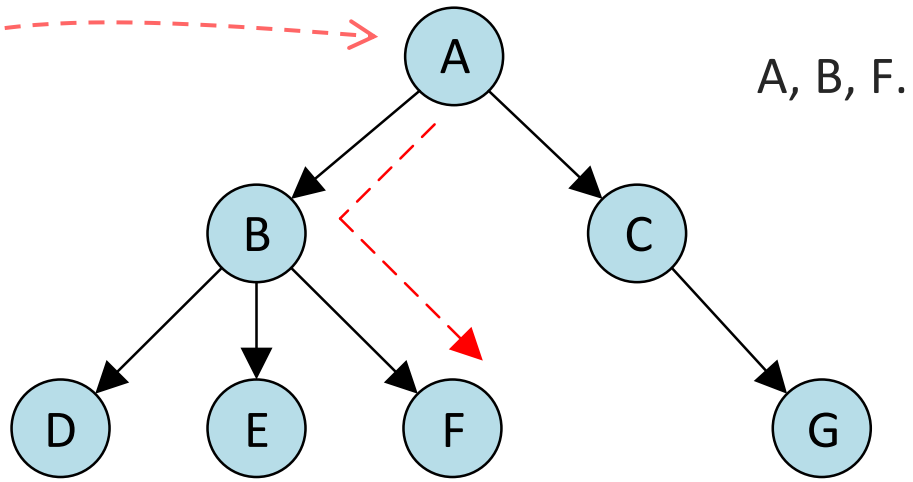


- **Pai**: é o antecessor imediato de um vértice ou **nó**;
- **Filho**: é o sucessor imediato de um vértice ou **nó**;
- **Raiz**: é o vértice que não possui Pai;
- **Nós Terminais** ou **Folhas**: qualquer vértice que não possui **filhos**;
- **Nós Não Terminais** ou **Internos**: qualquer vértice que possui ao menos 1 **filho**.

Árvores

- Caminho em uma árvore:
  - É uma sequência de vértices de modo que existe sempre uma aresta ligando o vértice anterior com o seguinte;
  - Existe exatamente um caminho entre a raiz e cada um dos nós da árvore.

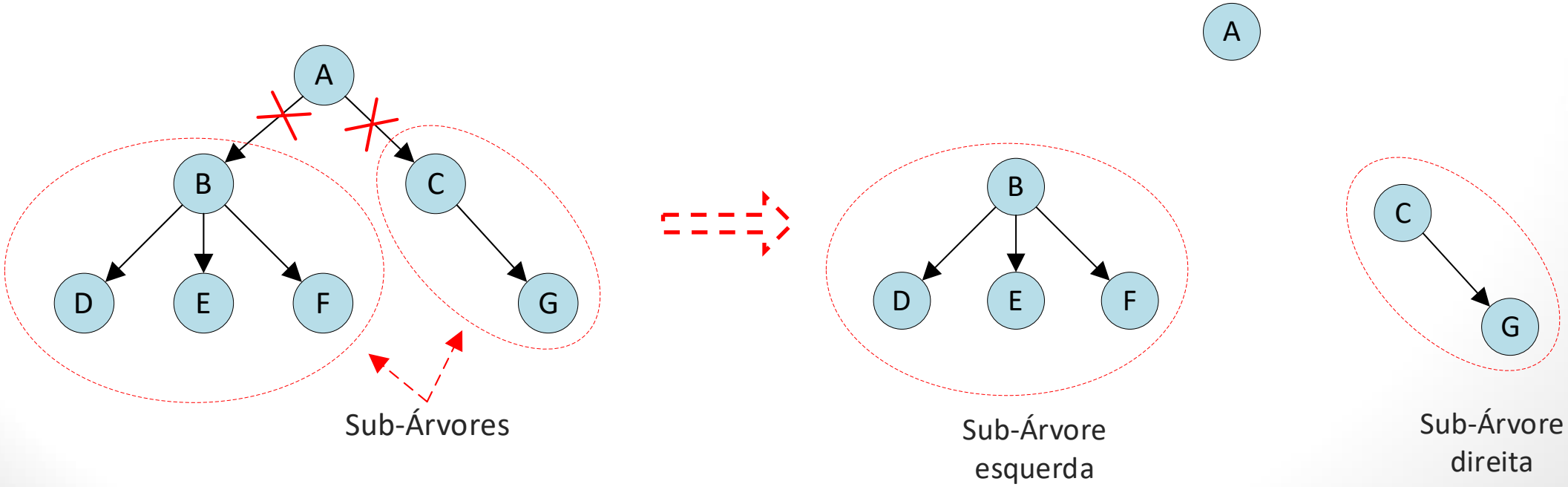
Sempre é possível sair da Raiz e chegar em qualquer vértice: É um Grafo Conexo.



A, B, F.

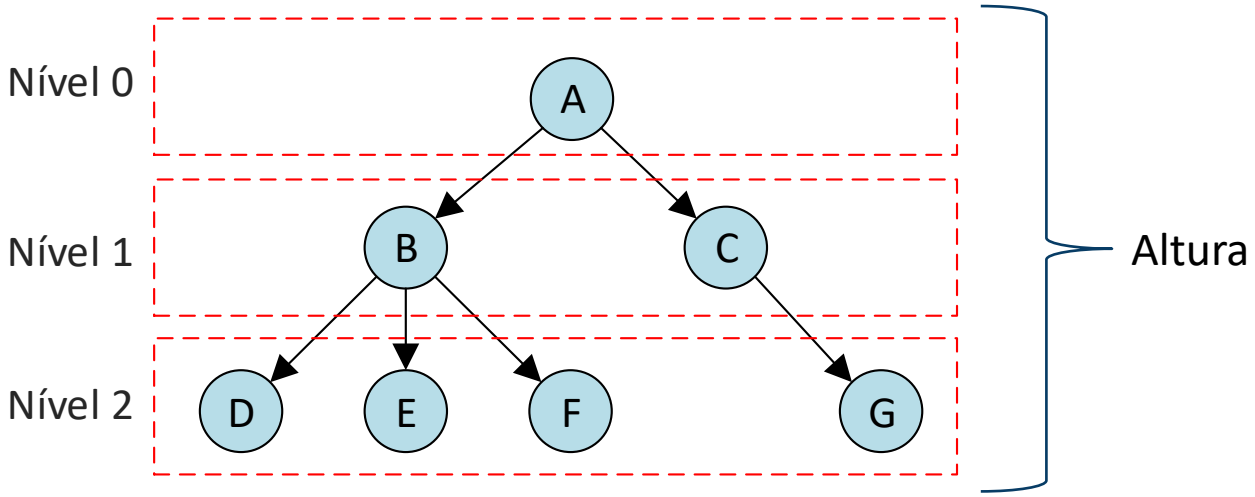
Árvores

- Sub-Árvores:
  - Dado um determinado vértice, cada filho seu é a raiz de uma nova árvore;
  - De fato, qualquer vértice é a Raiz de uma sub-árvore, consistindo dele e dos nós abaixo dele.
- Grau de um vértice:
  - É o número de sub-árvores do vértice.



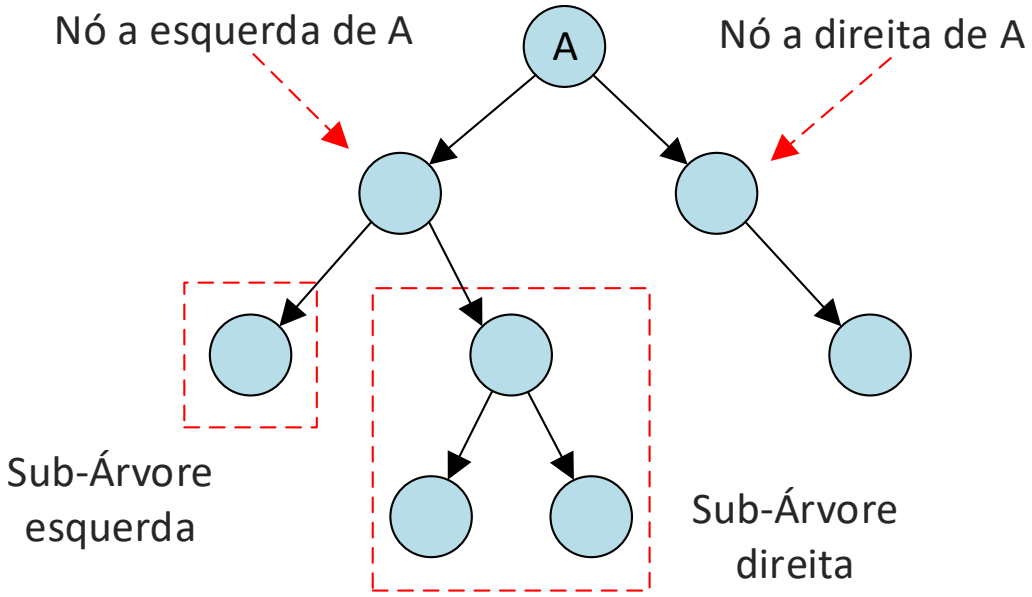
Árvores

- Altura de uma árvore:
  - Também chamada de profundidade;
  - É o comprimento do caminho mais longo da Raiz até uma das suas Folhas;
- Níveis:
  - Em uma árvore, os vértices são classificados em níveis;
  - O nível é o número de nós no caminho entre o vértice e a Raiz



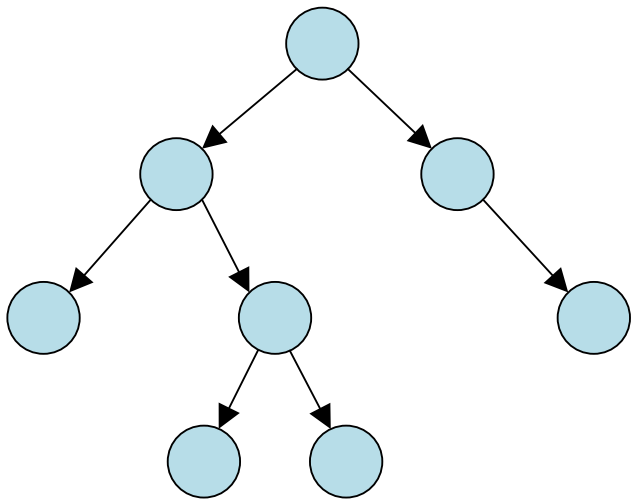
Árvore Binária

- Árvore Binária:
  - É um tipo especial de árvore;
  - Cada vértice pode possuir duas sub-árvores: **sub-árvore esquerda** e **sub-árvore Direita**
  - O Grau de cada vértice (número de Folhas), pode ser 0, 1 ou 2.

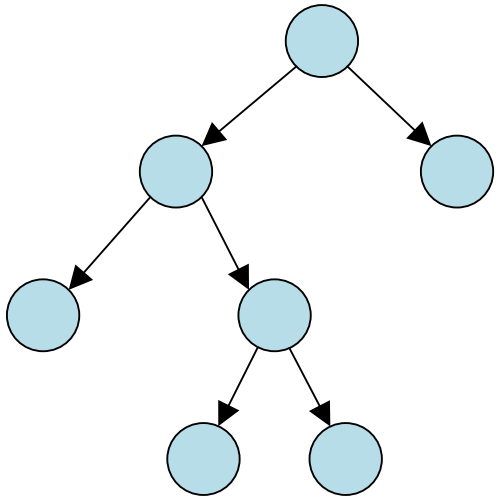


Árvore Binária

- Árvore Estritamente Binária:
  - Cada nó (vértice) possui 0 ou 2 sub-árvores;
  - Nenhum nó tem **filho único**;
  - nós internos (não Folhas), sempre têm 2 filhos;



Árvore Binária

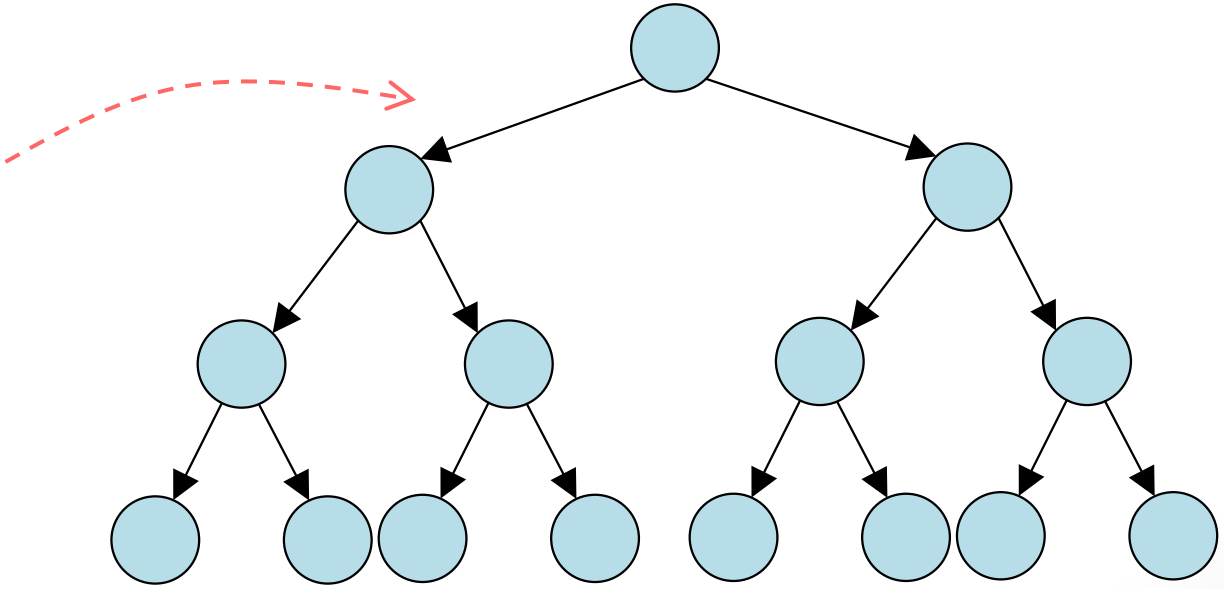


Árvore Estritamente Binária

Árvore Binária

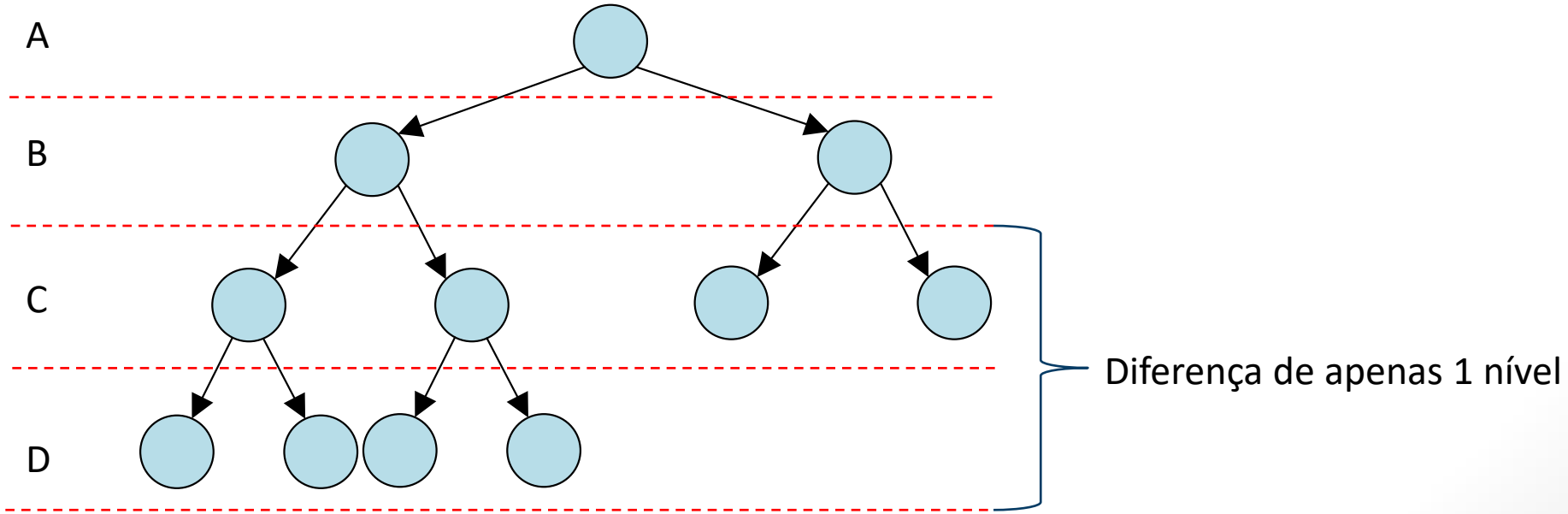
- Árvore Binária Completa
  - É Estritamente Binária e todos os seus nós-Folha estão no mesmo nível.
  - O número de nós de uma Árvore Binária Completa é 2^{h-1}, onde "h" é a altura da árvore.

Em uma Árvore Binária Completa, é possível calcular o número de nós.



Árvore Binária

- Árvore Binária Quase Completa:
  - A diferença de altura entre as sub-árvores de qualquer nó e de no máximo 1;
  - Se a altura da árvore é “D”, cada folha esta no nível “D” ou “D – 1”.



Árvore Binária

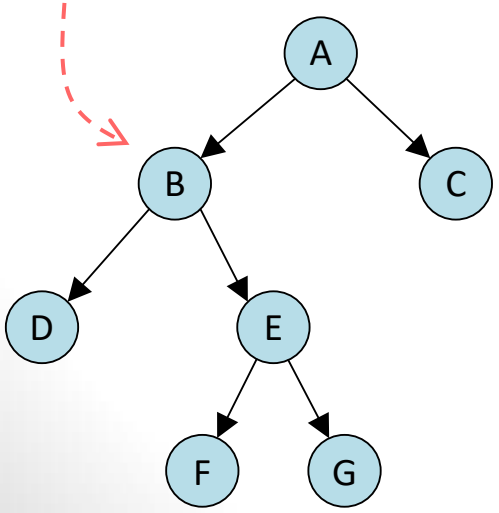
- Árvore Binária – Implementação utilizando TAD:
  - Em uma árvore Binária podemos realizar as seguintes operações
    - Criação da árvore;
    - Inserção de um elemento;
    - Remoção de um elemento;
    - Acesso à um elemento;
    - Destruição da árvore.
- Essas operações dependem do tipo de alocação de memória utilizada:
  - Estática (*heap*);
  - Dinâmica (lista encadeada).

Árvore Binária

- Alocação Estática (*Heap*):
  - Utiliza de *Array*;
  - Utiliza duas funções para retornar a posição dos filhos a esquerda e a direita de um Pai

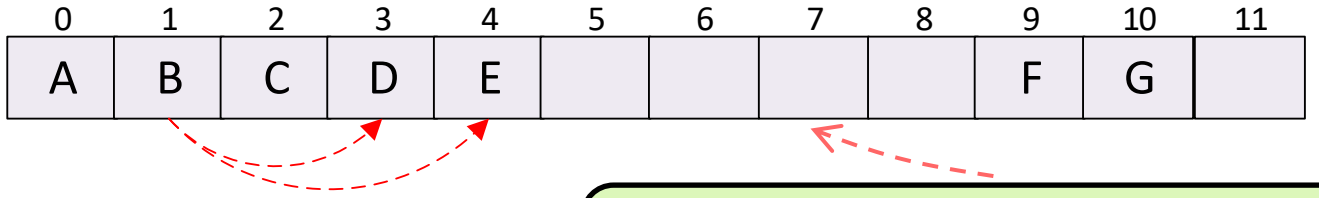
```
FILHO_ESQ(PAI) = 2 * PAI + 1;
FILHO_DIR(PAI) = 2 * PAI + 2;
```

Temos mais elementos à esquerda...



Não é boa quando temos muitos elementos alocados para um lado da árvore: teremos muito espaço alocado sem utilização. Gera muitos espaços vazios.

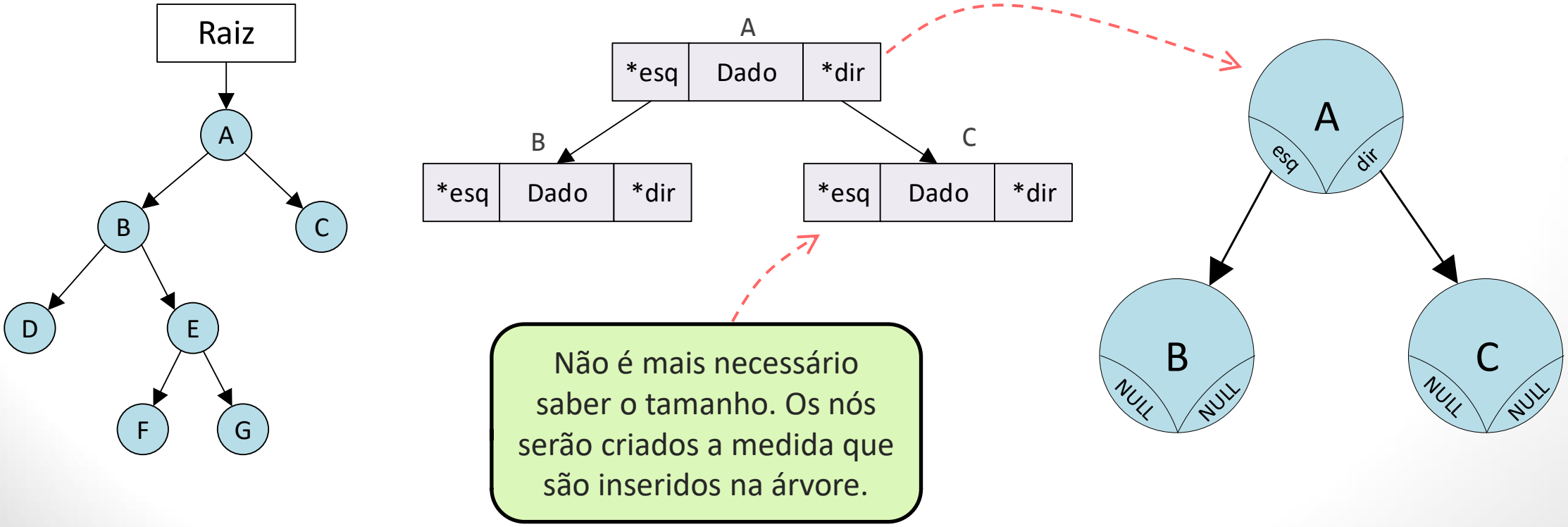
FILHO_ESQ(PAI) = 2 * PAI + 1;
FILHO_DIR(PAI) = 2 * PAI + 2;



Não é boa quando temos muitos elementos inseridos para um lado da árvore: teremos muito espaço alocado sem utilização. Gera muitos espaços vazios.

Árvore Binária

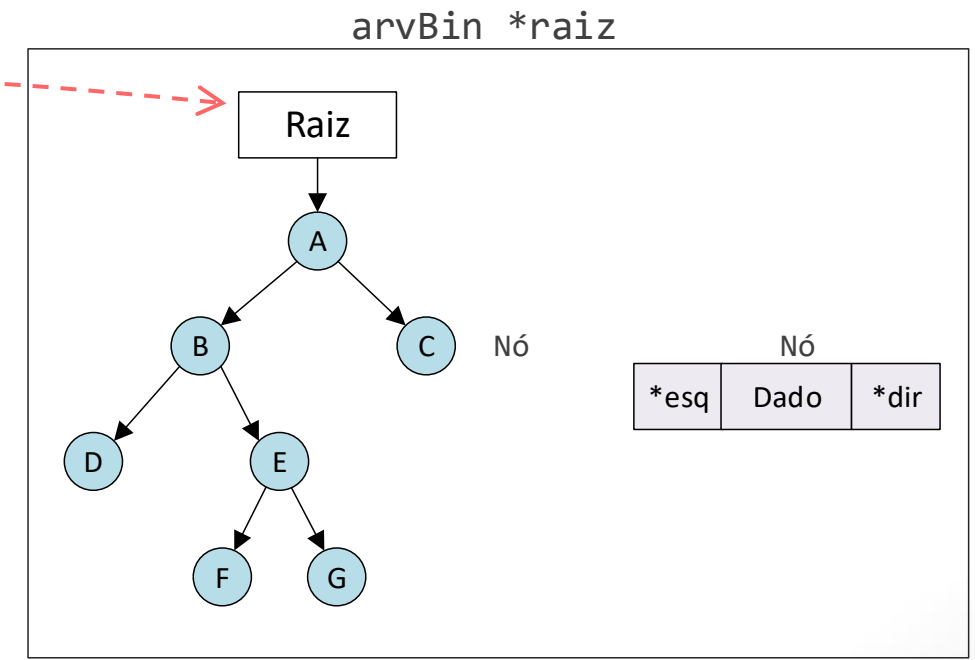
- Alocação Dinâmica – Lista Encadeada:
  - Cada nó da árvore é tratado como um ponteiro alocado dinamicamente a medida que os dados são inseridos.



Árvore Binária

- Implementando uma Árvore Binária com alocação dinâmica - Lista Encadeada:
  - Para guardar o primeiro nó da árvore (raiz), utilizaremos um ponteiro para ponteiro;
  - Um ponteiro para ponteiro pode guardar o endereço de um ponteiro;
  - Assim, fica mais fácil mudar quem é a raiz da árvore, caso seja necessário.

Em uma Árvore Binária de Busca, não é tão necessário, pois não se troca a raiz, mas em Árvores Balanceadas a raiz sempre muda, e esse projeto será utilizado por outras árvores...



Árvore Binária

- Implementando uma Árvore Binária
  - `arvoreBinaria.h` – serão definidos tudo que ficará visível ao usuário programador desta biblioteca:
    - Os protótipos das funções que manipulam o dado encapsulado;
    - O tipo de dado que será armazenado na árvore;
    - O ponteiro árvore que estará disponível para o `main()`.
  - `arvoreBinaria.c` – serão definidos:
    - O tipo de dado árvore (tipo opaco);
    - A implementação de suas funções que manipulam o tipo opaco, fechando assim o encapsulamento.

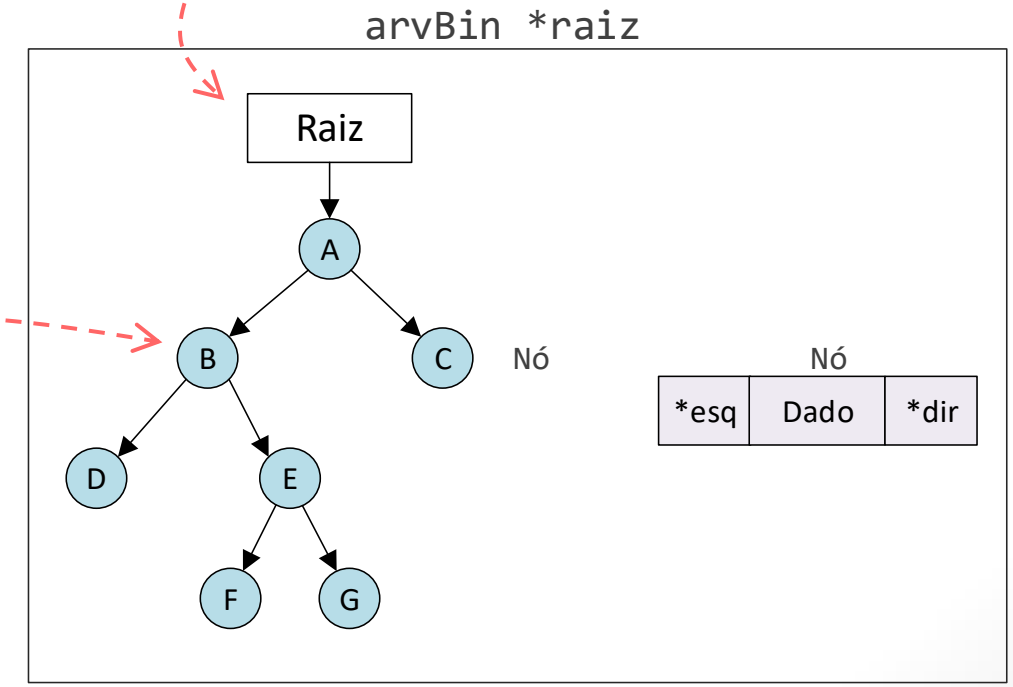
Árvore Binária

```c
//Arquivo arvoreBinaria.h
typedef struct NO *ArvBin;
```

```c
//Arquivo arvoreBinaria.c
#include <stdio.h>
#include <stdlib.h>
#include "arvoreBinaria.h"

struct NO{
    int info;
    struct NO *esq;
    struct NO *dir;
};
```

```c
//programa principal
int main(){
    int x; //será usado como retorno cod. erro
    ArvBin *raiz;
```



Árvore Binária

- Criando e destruindo a Árvore Binária:
  - Criação da árvore: ato de criar a raiz na árvore. A Raiz é um **tipo de nó especial** que aponta para o **primeiro elemento** da árvore;

Já pensando na implementação das próximas árvores, AVL e Rubro-Negra...

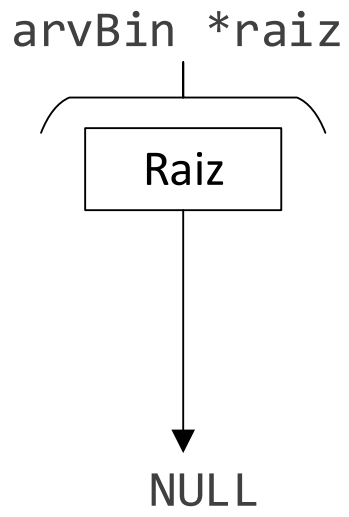
- Destruição da árvore:
  - Envolve percorrer todos os nós da árvore de modo a liberar a memória alocada para cada um deles.

Árvore Binária

```c
//Arquivo arvoreBinaria.h
ArvBin *cria_arvBin();
```

```c
//Arquivo arvoreBinaria.c
ArvBin *cria_arvBin(){
    ArvBin *raiz = (ArvBin*) malloc(sizeof(ArvBin));
    if(raiz != NULL){
        *raiz = NULL;
    }
    return raiz;
}
```

```c
//programa principal
raiz = cria_arvBin();
```



Árvore Binária

- Destruir a árvore:

```c
//Arquivo arvoreBinaria.h
void liberar_arvBin(ArvBin *raiz);
```

```c
//programa principal
liberar_arvBin(raiz);
```

```c
//Arquivo arvoreBinaria.c
void liberar_arvBin(ArvBin *raiz){
    if(raiz == NULL){
        return;
    }
    libera_NO(*raiz);
    free(raiz);
}

void libera_NO(struct NO *no){
    if(no == NULL){
        return;
    }
    libera_NO(no->esq);
    libera_NO(no->dir);
    free(no);
    no = NULL;
}
```

É chamada para liberar cada nó, individualmente.

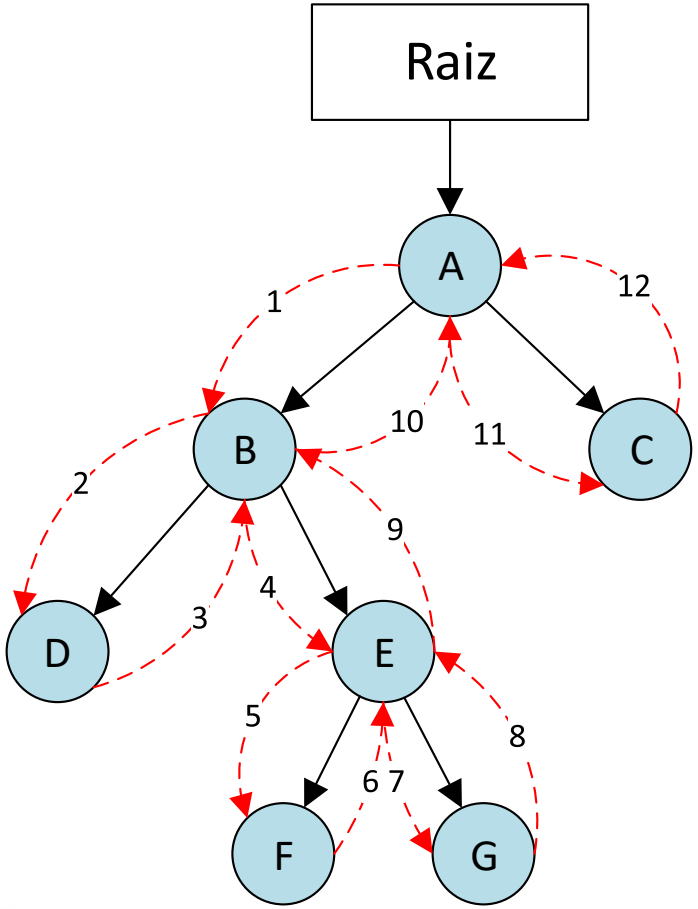
Após liberar todos os nós, libera a raiz

Percorre por recursão todos os nós, esquerdos e direitos

Armazena NULL em nó para não termos problemas.

Estrutura de Dados 2

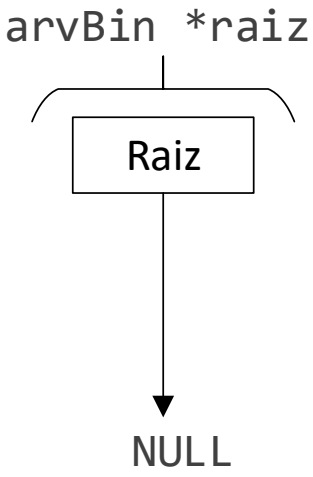
Árvore Binária



	1	Visita B
	2	Visita D
X	3	Libera D, volta para B
	4	Visita E
	5	Visita F
X	6	Libera F, volta para E
	7	Visita G
X	8	Libera G, volta para E
X	9	Libera E, volta para B
X	10	Libera B, volta para A
	11	Visita C
X	12	Libera C, volta para A
X		Libera A

Árvore Binária

- Informações Básicas:
  - Está vazia?
  - Número de nós?
  - Altura da árvore?




```c
//Arquivo arvoreBinaria.h
int vazia_arvBin(ArvBin *raiz);
```

```c
//Arquivo arvoreBinaria.c
int vazia_arvBin(ArvBin *raiz){
    if(raiz == NULL){
        return 1;
    }
    if(*raiz == NULL){
        return 1;
    }
    return 0;
}
```

```c
//programa principal
if(vazia_arvBin(raiz)){
    printf("A arvore esta vazia.");
}else{
    printf("A Arvore possui elementos.");
}
printf("\n");
```

Árvore Binária

- Altura da árvore

```c
//Arquivo arvoreBinaria.h
int altura_arvBin(ArvBin *raiz);
```

```c
//programa principal
x = altura_arvBin(raiz);
printf("Altura da arvore: %d", x);
```

```c
//Arquivo arvoreBinaria.c
int altura_arvBin(ArvBin *raiz){
    if(raiz == NULL){
        return 0;
    }
    if(*raiz == NULL){
        return 0;
    }
    int alt_esq = altura_arvBin(&((*raiz)->esq));
    int alt_dir = altura_arvBin(&((*raiz)->dir));
    if(alt_esq > alt_dir){
        return(alt_esq + 1);
    }else{
        return(alt_dir + 1);
    }
}
```

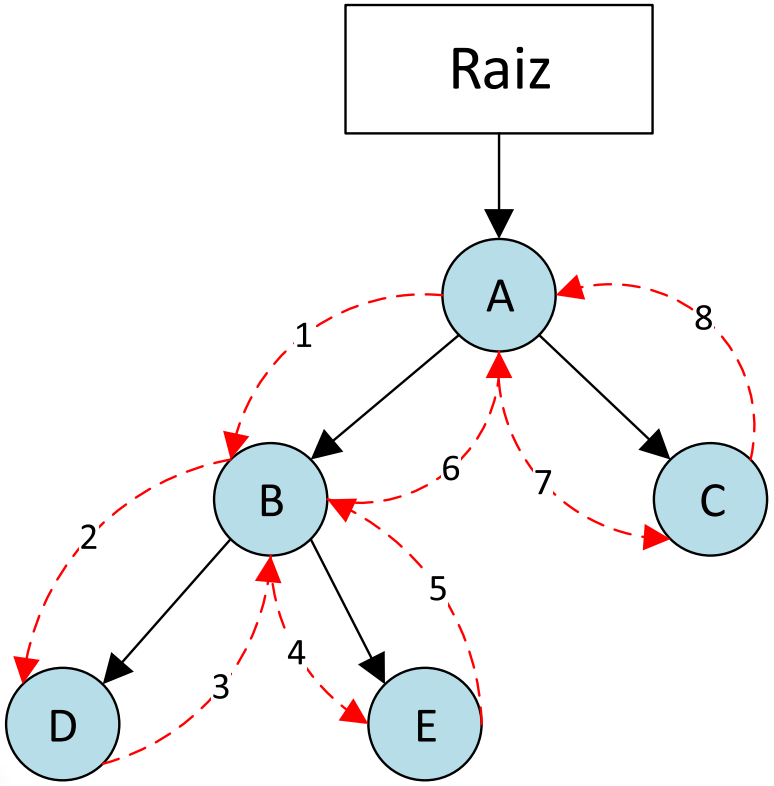
Quando a recursão descer no nó folha, ela retorna 0. Neste caso altura 0.

Passado o endereço do nó, porque a função recursiva espera um ponteiro.

A recursão então vai subindo no retorno somando 1

Para saber a altura é necessário percorrer todos os nós.

Árvore Binária



1	Visita B
2	Visita D
3	D é nó Folha: altura é 1. Volta para B
4	Visita E
5	E é nó Folha: Altura é 1. Volta para B
6	Altura de B é 2: maior altura dos filhos + 1. Volta para A
7	Visita C
8	C é nó Folha: altura é 1. Volta para A
	Altura de A é 3: maior altura dos filhos + 1.

Árvore Binária

- Número total de nós

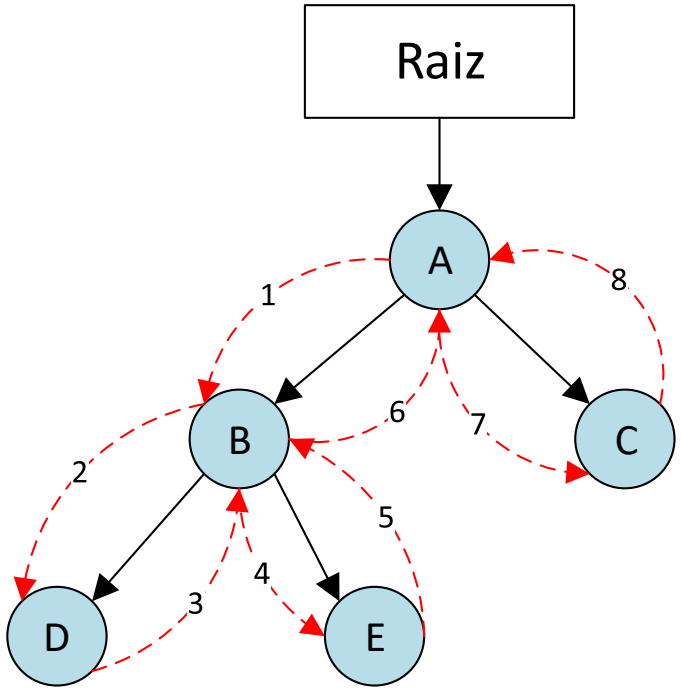
```c
//Arquivo arvoreBinaria.h
int totalNO_arvBin(ArvBin *raiz);
```

```c
//programa principal
x = totalNO_arvBin(raiz);
printf("Total de nos na arvore: %d", x);
```

```c
//Arquivo arvoreBinaria.c
int totalNO_arvBin(ArvBin *raiz){
    if(raiz == NULL){
        return 0;
    }
    if(*raiz == NULL){
        return 0;
    }
    int alt_esq = totalNO_arvBin(&((*raiz)->esq));
    int alt_dir = totalNO_arvBin(&((*raiz)->dir));
    return(alt_esq + alt_dir + 1);
}
```

Árvore Binária

- Número total de nós



1	Visita B
2	Visita D
3	D é nó Folha: conta como 1 nó. Volta para B
4	Visita E
5	E é nó Folha: conta como 1 nó. Volta para B
6	Número de nós em B é 3: total de nós à direita (1) + Total de nós à esquerda (1) + 1. Volta para A
7	Visita C
8	C é nó Folha: conta como 1 nó. Volta para A
	Número de nós em A é 5: Total de nós a direita (1) + Total de nós à esquerda (3) + 1.

Árvore Binária

- Percorrendo uma Árvore Binária:
  - Muitas operações em Árvores Binárias necessitam que se percorra todos os nós de suas sub-árvores, executando alguma ação ou tratamento em cada nó;
  - Temos que garantir que cada nó será **visitado** uma única vez;
  - Isso gera uma sequência linear de nós, cuja ordem **sempre dependerá** de como a árvore foi percorrida.

Árvore Binária

- Veremos 3 maneiras de percorrer uma árvore, existem outras, mas estas são as mais utilizadas:
  - Pré-Ordem
    - Visita a Raiz, o filho da esquerda e o filho da direita;
  - Em-Ordem
    - Visita o filho da esquerda, a Raiz e o filho da direita;
  - Pós-Ordem
    - Visita o filho da esquerda, o filho da direita e a Raiz.

Este foi o modo utilizado para percorrer e para liberar nós da árvore, na contagem dos nós e na verificação da altura da árvore.

Árvore Binária

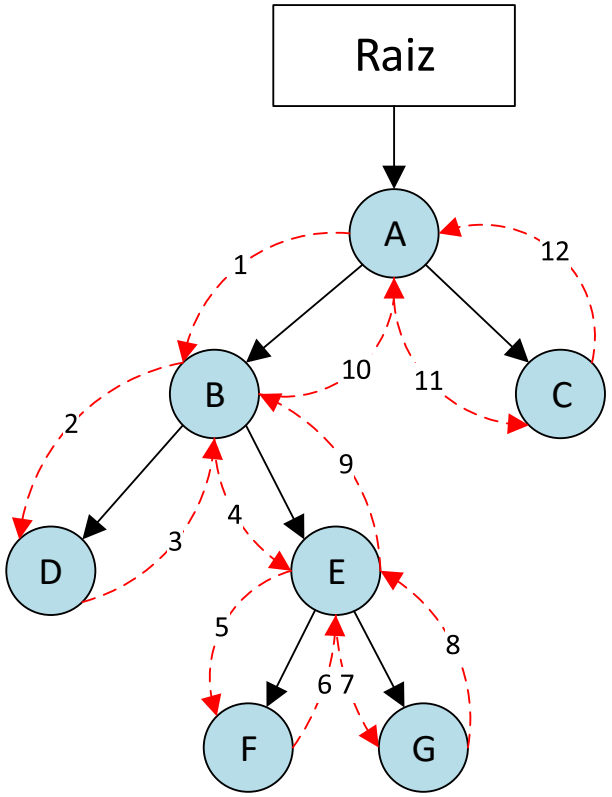
- Pré-Ordem

```c
//Arquivo arvoreBinaria.h
void preOrdem_arvBin(ArvBin *raiz);
```

```c
//programa principal
preOrdem_arvBin(raiz);
```

```c
//Arquivo arvoreBinaria.c
void preOrdem_arvBin(ArvBin *raiz){
    if(raiz == NULL){
        return;
    }
    if(*raiz != NULL){
        printf("%d\n", (*raiz)->info);
        preOrdem_arvBin(&((*raiz)->esq));
        preOrdem_arvBin(&((*raiz)->dir));
    }
}
```

Árvore Binária



1	Imprime A, visita B
2	Imprime B, visita D
3	Imprime D, volta para B
4	Visita E
5	Imprime E, visita F
6	Imprime F, volta para E
7	Visita G
8	Imprime G, volta para E
9	Volta para B
10	Volta para A
11	Visita C
12	Imprime C, volta para a

Resultado: ABDEFGC

Árvore Binária

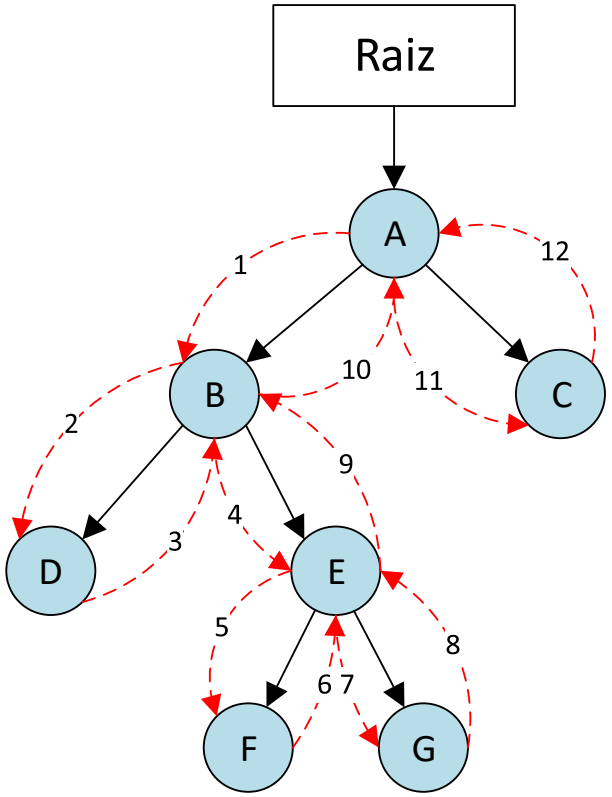
- Em-Ordem

```c
//Arquivo arvoreBinaria.h
void emOrdem_arvBin(ArvBin *raiz);
```

```c
//programa principal
emOrdem_arvBin(raiz);
```

```c
//Arquivo arvoreBinaria.c
void emOrdem_arvBin(ArvBin *raiz){
    if(raiz == NULL){
        return;
    }
    if(*raiz != NULL){
        emOrdem_arvBin(&((*raiz)->esq));
        printf("%d\n", (*raiz)->info);
        emOrdem_arvBin(&((*raiz)->dir));
    }
}
```

Árvore Binária



1	Visita B
2	Visita D
3	Imprime D, volta para B
4	Imprime B, visita E
5	Visita F
6	Imprime F, volta para E
7	Imprime E, visita G
8	Imprime G, volta para E
9	Volta para B
10	Volta para A
11	Imprime A, visita C
12	Imprime C

Resultado: DBFEGAC

Árvore Binária

- Pós-Ordem

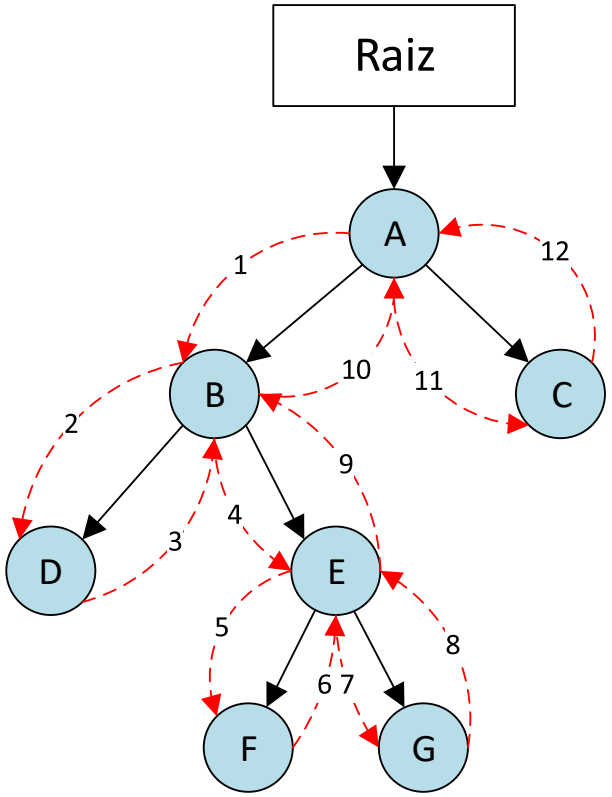
```c
//Arquivo arvoreBinaria.h
void posOrdem_arvBin(ArvBin *raiz);
```

```c
//programa principal
posOrdem_arvBin(raiz);
```

```c
//Arquivo arvoreBinaria.c
void posOrdem_arvBin(ArvBin *raiz){
    if(raiz == NULL){
        return;
    }
    if(*raiz != NULL){
        posOrdem_arvBin(&((*raiz)->esq));
        posOrdem_arvBin(&((*raiz)->dir));
        printf("%d\n", (*raiz)->info);
    }
}
```

Este método garante que todos os filhos serão visitados antes de se executar qualquer ação no nó Pai, como por exemplo, a sua exclusão.

Árvore Binária



1	Visita b
2	Visita D
3	Imprime D, volta para B
4	Visita E
5	Visita F
6	Imprime F, volta para E
7	Visita G
8	Imprime G, volta para E
9	Imprime E, volta para B
10	Imprime B, volta para A
11	Visita C
12	Imprime C, volta para A
	Imprime A

Resultado: DFGEBCA

Árvore Binária de Busca

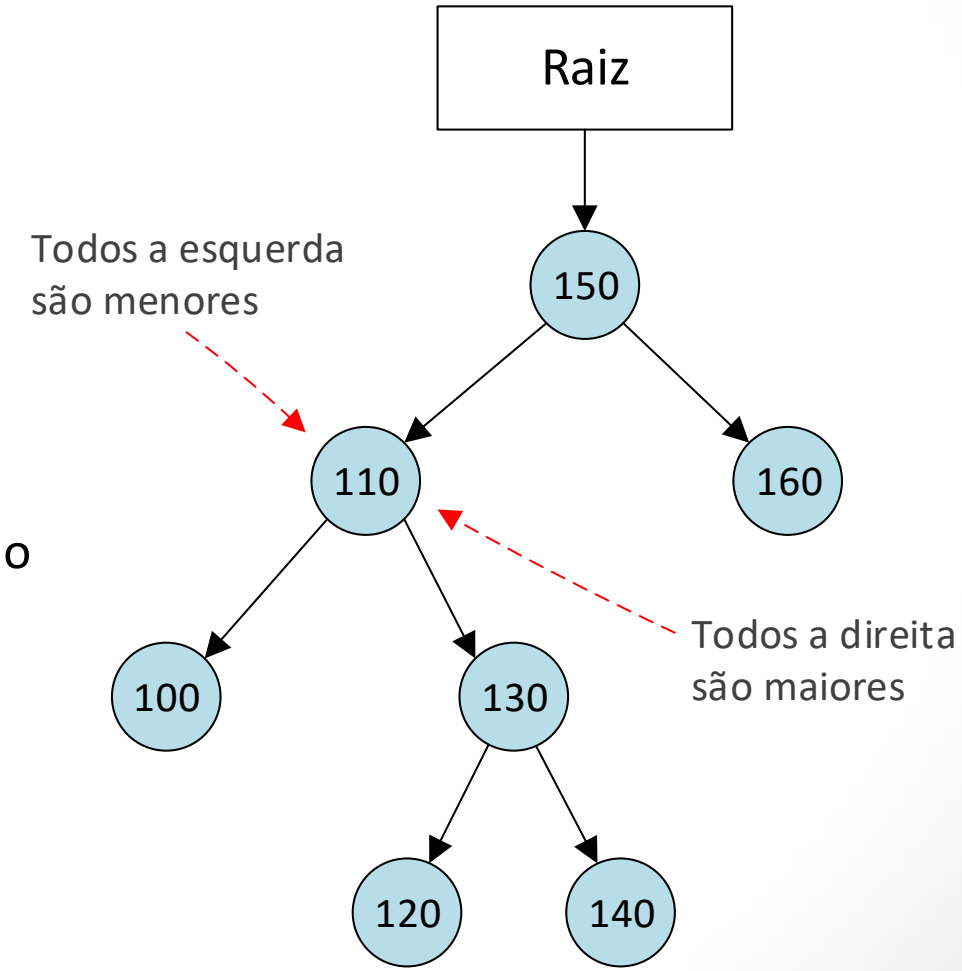
- Árvore Binária de Busca:
  - A BST é um tipo de **Árvore Binária**, específica para buscas, onde cada nó possui um **valor** (chave), associado a ele , e esse valor determina a posição que o nó ocupará na árvore;
  - São um tipo especial de Árvore Binária porque, quando estão ordenadas, elas se aplicam à buscas, inserções e deleções, com acesso aos seus itens de forma extremamente veloz e em qualquer ordem. Além disso, a operação de recuperação é não-destrutiva. Embora fáceis de visualizar, as árvores apresentam difíceis problemas de programação.

Árvore Binária de Busca

- Árvore Binária de Busca:
  - A maioria das funções que as árvores usam, são recursivas, porque a própria árvore é uma estrutura de dados recursiva, isto é, cada sub-árvore, é uma árvore em si mesma. Assim, as rotinas desenvolvidas, também são naturalmente recursivas.
  - Versões não recursivas dessas funções existem, mas o seu código é muito mais difícil de ser compreendido e implementado.
  - Para esta implementação, assumiremos que **não existem valores repetidos** (não trabalharemos com eles).

Árvore Binária de Busca

- Posicionamento dos valores:
  - Para cada nó Pai:
    - Todos os valores da sub-árvore esquerda são menores do que os do nó Pai;
    - Todos os valores da sub-árvore direita são maiores do que os do nó Pai

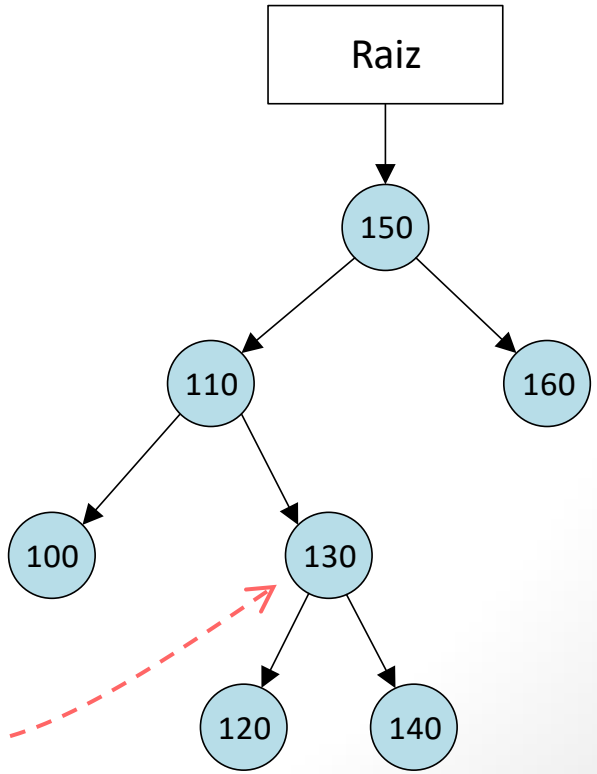


Árvore Binária de Busca

- A inserção e a remoção de nós da árvore devem ser realizadas respeitando esta propriedade da árvore.

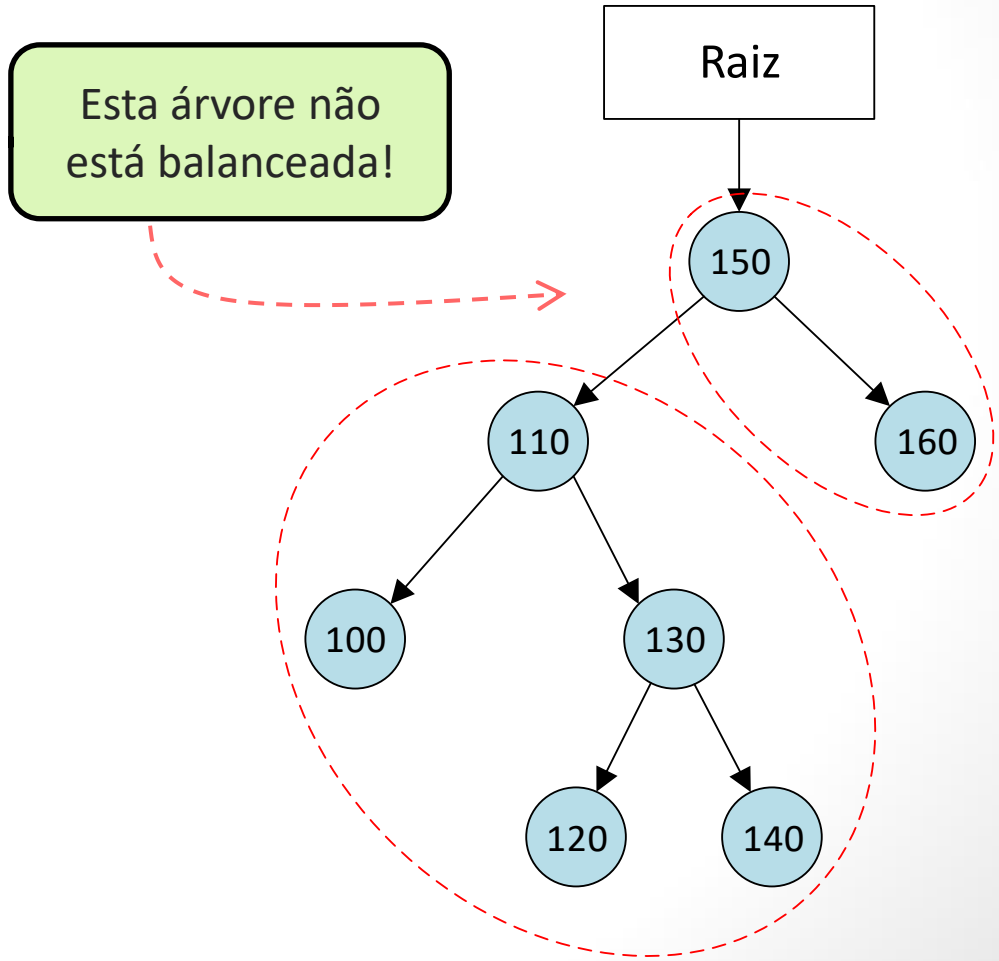
- Aplicações:
  - Busca Binária;
  - Análise de expressões algébricas: prefixa, infixa e pós-fixa.

Se for removido o elemento 130, é necessário existir um tratamento para definir qual nó ocupará o seu lugar, mantendo as propriedades da árvore.



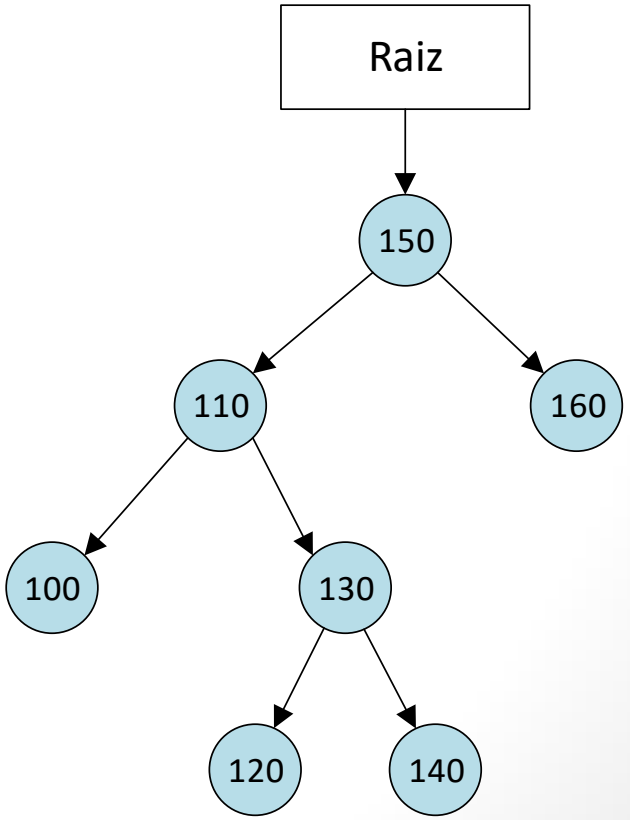
Árvore Binária de Busca

- Principais operações:
  - Inserção
    - Caso médio $O(log\ n)$;
    - Pior caso $O(n)$. (árvore não balanceada)
  - Remoção
    - Caso médio $O(log\ n)$;
    - Pior caso $O(n)$. (árvore não balanceada)
  - Consulta
    - Caso médio $O(log\ n)$;
    - Pior caso $O(n)$. (árvore não balanceada)



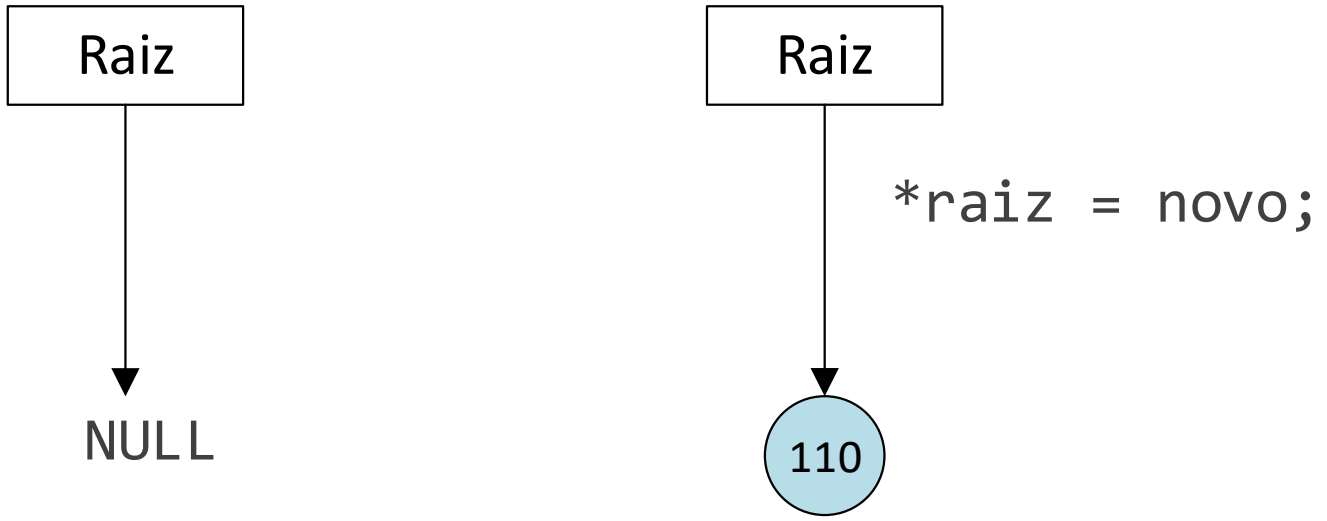
Inserção na Árvore Binária de Busca:

- Para inserir um elemento (V), na Árvore Binária de Busca :
  - Primeiro compare com a Raiz, e se:
    - V é menor do que a raiz, vá para a sub-árvore da esquerda;
    - V é maior do que a raiz, vá para a sub-árvore da direita;
  - Aplique o método recursivamente (também pode ser aplicado sem recursão).



Inserção na Árvore Binária de Busca:

- Também existe o caso onde a inserção é feita em uma árvore que está vazia, portanto, deve existir um tratamento específico para este caso:



Inserção na Árvore Binária de Busca:

```c
//Arquivo arvoreBinaria.h
int insere_arvBin(ArvBin *raiz, int valor);
```

```c
//Arquivo arvoreBinaria.c
int insere_arvBin(ArvBin *raiz, int valor){
    if(raiz == NULL){
        return 0;
    }
    struct NO *novo;
    novo = (struct NO*) malloc(sizeof(struct NO));
    if(novo == NULL){
        return 0;
    }
    novo->info = valor;
    novo->dir = NULL;
    novo->esq = NULL;
    if(*raiz == NULL){
        *raiz = novo;
    }else{
        struct NO *atual = *raiz;
        struct NO *ant = NULL;
        while(atual != NULL){
            ant = atual;
            if(valor == atual->info){
                free(novo);//elemento já existe!
                return 0;
            }
            if(valor > atual->info){
                atual = atual->dir;
            }else{
                atual = atual->esq;
            }
        }
        if(valor > ant->info){
            ant->dir = novo;
        }else{
            ant->esq = novo;
        }
    }
    return 1;
```

Armazena o nó antes dele mudar. Quando atual passar a apontar p/ NULL, ant ainda apontará p/ o último nó, a folha.

Se árvore vazia, insere o primeiro nó.

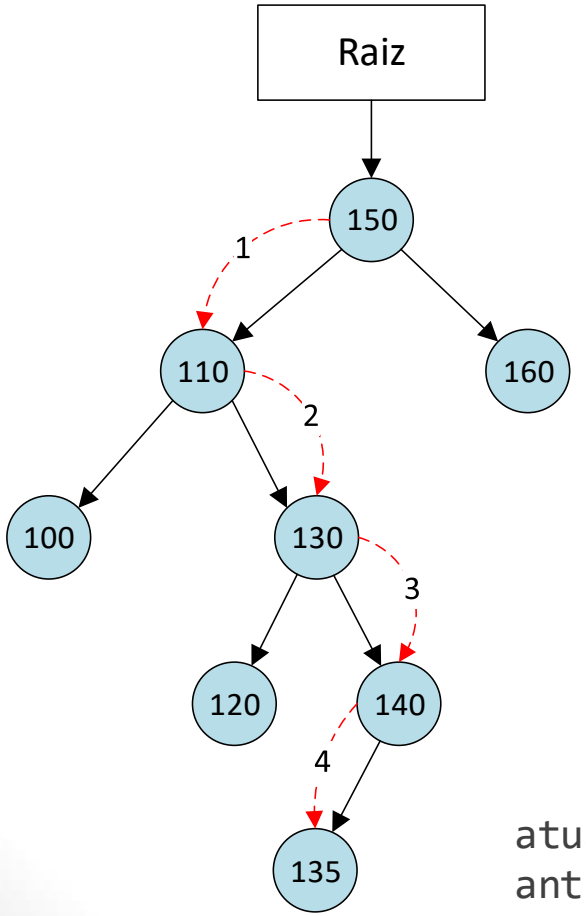
Sempre um novo nó será inserido como folha

Insere o novo nó como filho desse nó folha encontrado.

Quando atual receber NULL, ele chegou na folha.

Inserção na Árvore Binária de Busca:

- Insere como um NÓ Folha: Valor 135




```
atual == NULL
ant == 140
```

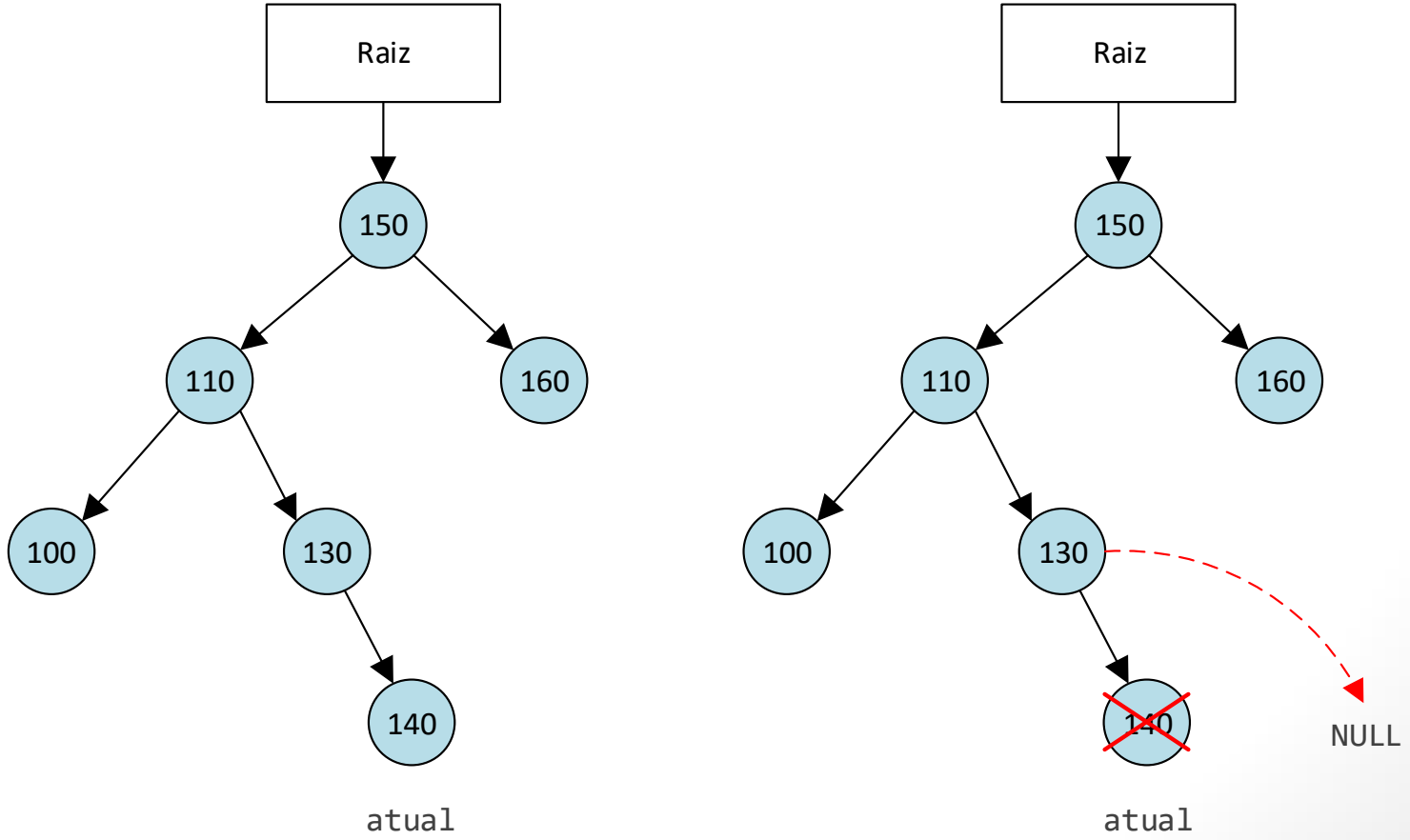
	Valor a inserir: 135.
1	Valor é menor do que 150, visita filho de esquerda;
2	Valor é maior do que 110, visita filho da direita;
3	Valor é maior do que 130, visita filho da direita;
4	Valor é menor do que 140, visita filho da esquerda;
	Não existe filho da esquerda, então “Valor” passa a ser filho da esquerda de 140.

Inserção na Árvore Binária de Busca:

```
//programa principal
x = insere_arvBin(raiz, 150);
if(x){
    printf("Elemento inserido na arvore com sucesso.\n");
}else{
    printf("Erro: elemento nao inserido na arvore.\n");
}
x = insere_arvBin(raiz, 110);
if(x){
    printf("Elemento inserido na arvore com sucesso.\n");
}else{
    printf("Erro: elemento nao inserido na arvore.\n");
}
x = insere_arvBin(raiz, 100);
x = insere_arvBin(raiz, 130);
x = insere_arvBin(raiz, 120);
x = insere_arvBin(raiz, 140);
x = insere_arvBin(raiz, 160);
```

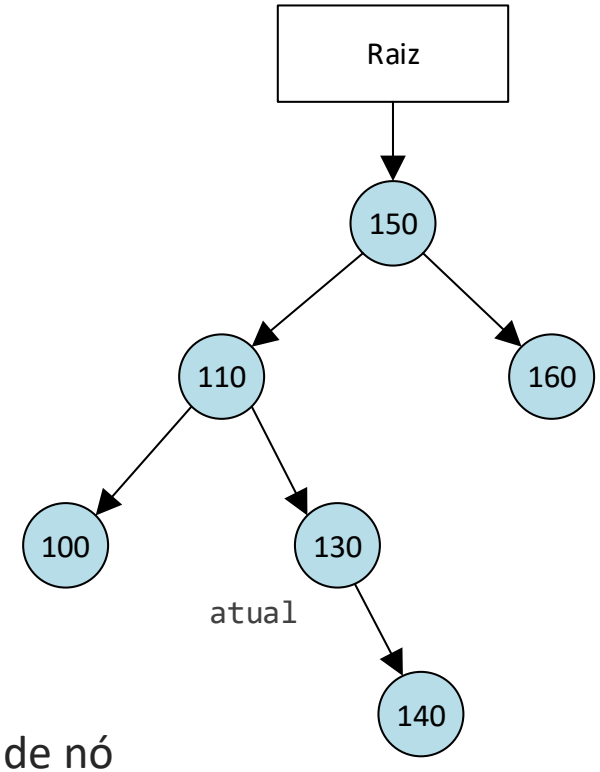
Remoção na Árvore Binária de Busca:

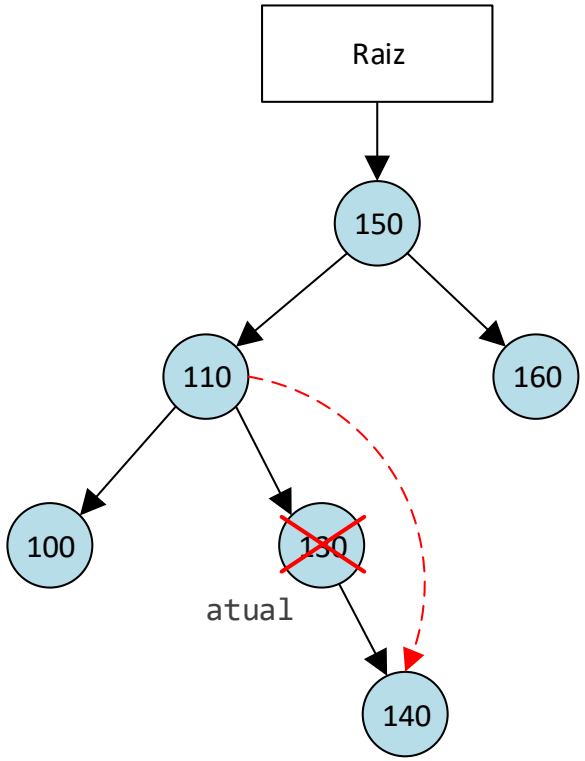
- Existem 3 tipos e remoção em Árvores Binárias de Busca:
  - Nó Folha (sem filhos);
  - Nó com 1 filho;
  - Nó com 2 filhos.



Remoção de nó
Folha sem filhos

Remoção na Árvore Binária de Busca:

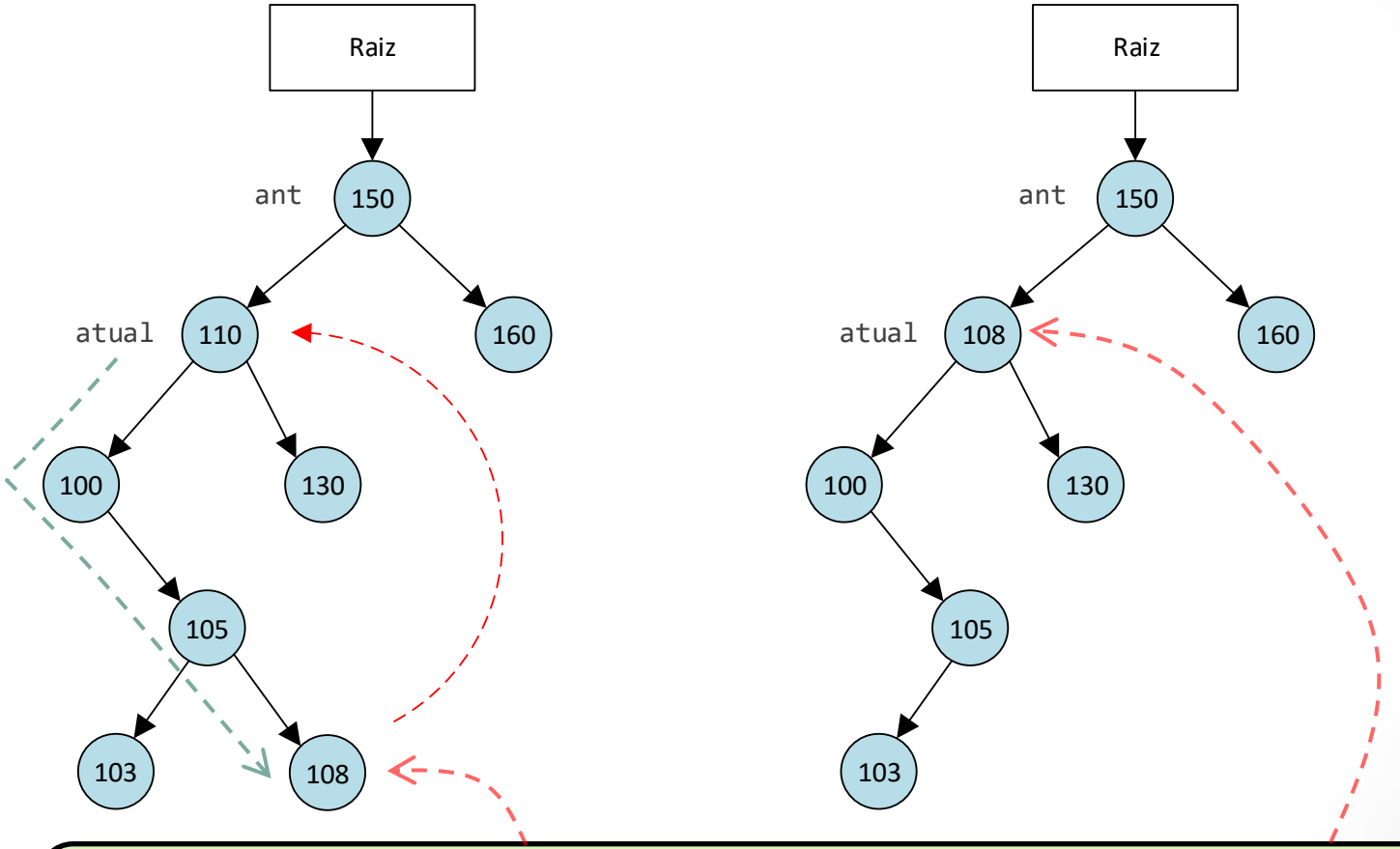




Remoção de nó
Folha com 1 filho

Remoção na Árvore Binária de Busca:

Para garantir que a árvore continue sendo uma árvore Binária de Busca, é necessário reposicionar os elementos, movimentando o filho da sub-árvore da esquerda, que está **mais a direita**, para substituir o nó que será removido.



Este filho que estava mais a direita na sub-árvore da esquerda, é menor do que qualquer elemento que está na sub-árvore da direita, e maior do que qualquer elemento que está na sub-árvore da esquerda.

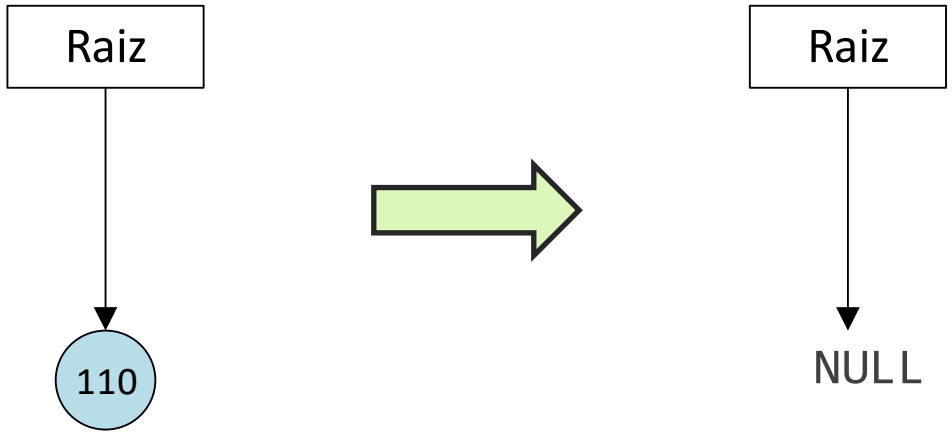
Remoção de nó Folha com 2 filhos

Remoção na árvore Binária de Busca:

- Os três tipos de remoção trabalham juntos. A remoção sempre remove um elemento específico da árvore, o qual pode ser um nó folha, ter um ou dois filhos.

- Cuidado:

  - Não se pode remover de uma árvore vazia;
  - Removendo o último nó, a árvore fica vazia.



Remoção na árvore Binária de Busca:

```c
//programa principal
x = remove_arvBin(raiz, 100);
```

```c
//Arquivo arvoreBinaria.h
int remove_arvBin(ArvBin *raiz, int valor);
```

```c
//Arquivo arvoreBinaria.c

//função responsável pela busca do Nó a ser removido
int remove_arvBin(ArvBin *raiz, int valor){

//função responsável por tratar os 3 tipos de remoção
struct NO *remove_atual(struct NO *atual){
```

Estrutura de Dados 2

```c
//Arquivo arvoreBinaria.c
//função responsável pela busca do Nó a ser removido
int remove_arvBin(ArvBin *raiz, int valor){
    if(raiz == NULL){
        return 0;
    }
    struct NO *ant = NULL;
    struct NO *atual = *raiz;
    while(atual != NULL){
        if(valor == atual->info){
            if(atual == *raiz){
                *raiz = remove_atual(atual);
            }else{
                if(ant->dir == atual){
                    ant->dir = remove_atual(atual);
                }else{
                    ant->esq = remove_atual(atual);
                }

            }
            return 1;
        }
        ant = atual;
        if(valor > atual->info){
            atual = atual->dir;
        }else{
            atual = atual->esq;
        }
    }
}
```

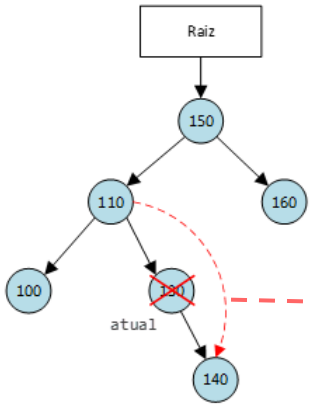
Caso Base: desceu pela árvore e encontrou o nó a ser removido.

Enquanto não chegar na folha, continua procurando...

Continua descendo pela árvore a procura do nó a ser removido.

se for a raiz, a remove, senão, trata de que lado se dará a remoção.

Estrutura de Dados 2




```c
//Arquivo arvoreBinaria.c
//função responsável por tratar os 3 tipos de remoção:
struct NO *remove_atual(struct NO *atual){
    struct NO *no1, *no2;
    if(atual->esq == NULL){
        no2 = atual->dir;
        free(atual);
        return no2;
    }
    no1 = atual;
    no2 = atual->esq;
    while(no2->dir != NULL){
        no1 = no2;
        no2 = no2->dir;
    }
    if(no1 != atual){
        no1->dir = no2->esq;
        no2->esq = atual->esq;
    }
    no2->dir = atual->dir;
    free(atual);
    return no2;
}
```

Trata nó sem filho da esquerda: filho da direita é devolvido em seu lugar (no2), mesmo que seja NULL, ou seja, não o tenha. Este bloco trata nó folha e nó com 1 filho.

Busca filho mais a direita na sub-árvore da esquerda.

Refaz as ligações dos ponteiros do antigo nó que será removido, para o nó que foi encontrado. Devolve esse nó mais a direita na sub-árvore da esquerda para a chamada desta função, porém, antes apagando o nó a ser removido.

Implementar esta função no arquivo `arvoreBinaria.c` antes da função `remove_arvBin()` do slide anterior

Consulta na Árvore Binária de Busca

- Para pesquisar um nó “V” na Árvore Binária de Busca
  - Primeiro compare com a Raiz.
    - V é menor do que a Raiz:
      - Vá para a sub-árvore da esquerda.
    - V é maior do que a Raiz:
      - Vá para a sub-árvore da direita.
- Aplique o método recursivamente (pode ser feito sem recursão).

Consulta na Árvore Binária de Busca

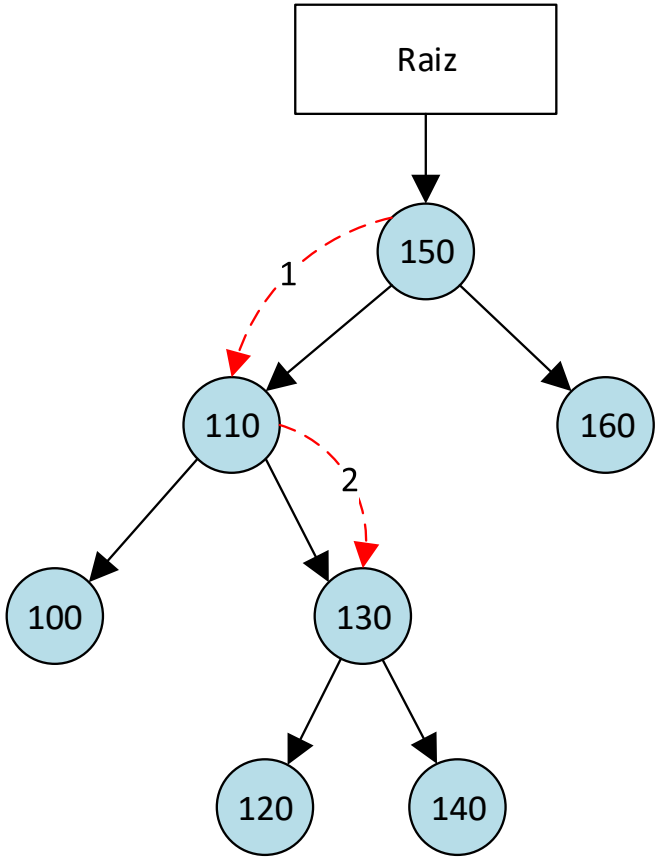
```c
//Arquivo arvoreBinaria.h
int consulta_arvBin(ArvBin *raiz, int valor);
```

```c
//Arquivo arvoreBinaria.c
int consulta_arvBin(ArvBin *raiz, int valor){
    if(raiz == NULL){
        return 0;
    }
    struct NO *atual = *raiz;
    while(atual != NULL){
        if(valor == atual->info){
            return 1;
        }
        if(valor > atual->info){
            atual = atual->dir;
        }else{
            atual = atual->esq;
        }
    }
    return 0;
}
```

```c
//programa principal
printf("\nBusca na Arvore Binaria:\n");
if(consulta_arvBin(raiz, 140)){
    printf("\nConsulta realizada com sucesso!");
}else{
    printf("\nElemento nao encontrado...");
}
```

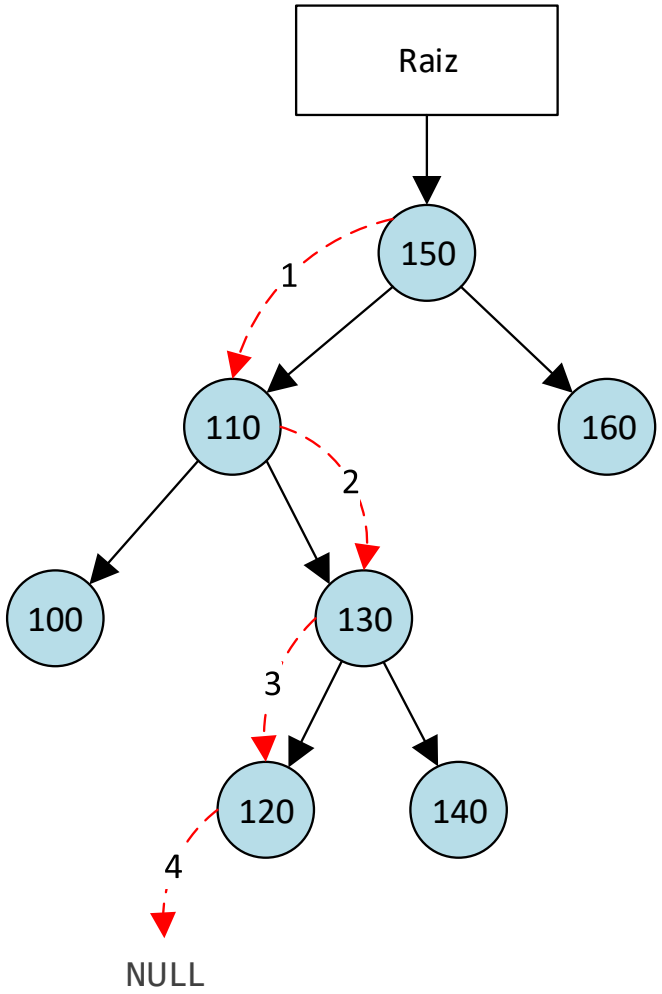
Se chegar ao final: `atual == NULL`, significa que toda a árvore foi percorrida e o elemento não foi encontrado.

Consulta na Árvore Binária de Busca



	Valor procurado: 130
1	Valor procurado é menor do que 150, visita filho da esquerda;
2	Valor procurado é maior do que 110, visita filho da direita;
	Valor procurado é igual ao do nó: retornar dados do nó;

Consulta na Árvore Binária de Busca



	Valor procurado: 115
1	Valor procurado é menor do que 150, visita filho da esquerda;
2	Valor procurado é maior do que 110, visita filho da direita;
3	Valor procurado é menor do que 130, visita filho da esquerda;
4	Valor procurado é menor do que 120, visita filho da esquerda;
	filho da esquerda de 120 não existe, é NULL: Busca Falhou, elemento não existe.

Estrutura de Dados 2

Atividade 1

- Entregue no Moodle o projeto Árvore Binária de Busca completo como Atividade Árvore Binária de Busca

- **Atenção! Guarde este projeto, e tenha-o sempre à mão, pois ele será utilizado como base para os próximos tipos de árvores que veremos.**