

Estrutura de Dados 2

Aula 04 – Árvores: Definição e Árvores Binárias

Antonio Angelo de Souza Tartaglia

angelot@ifsp.edu.br

Árvores

▶ Árvores:

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

▶ Vértice:

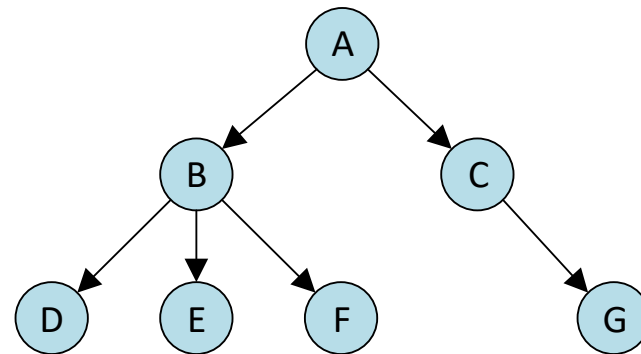
- É cada uma das entidades representadas na árvore e dependem da natureza do problema;

▶ Aresta:

- É uma conexão entre dois vértices.

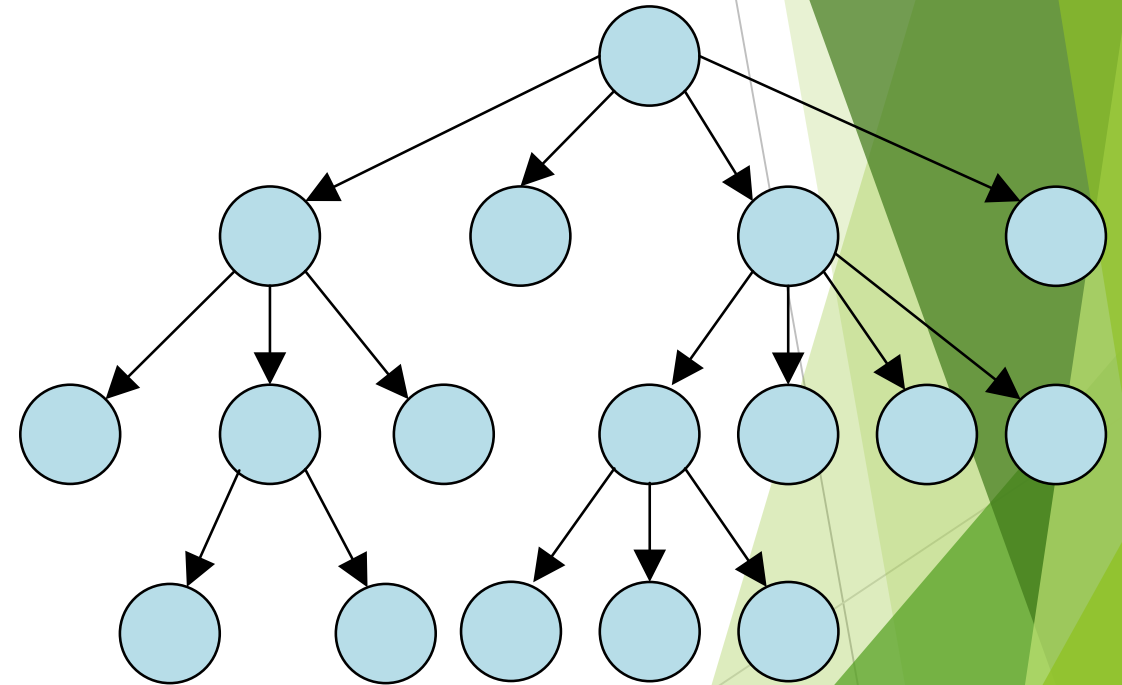
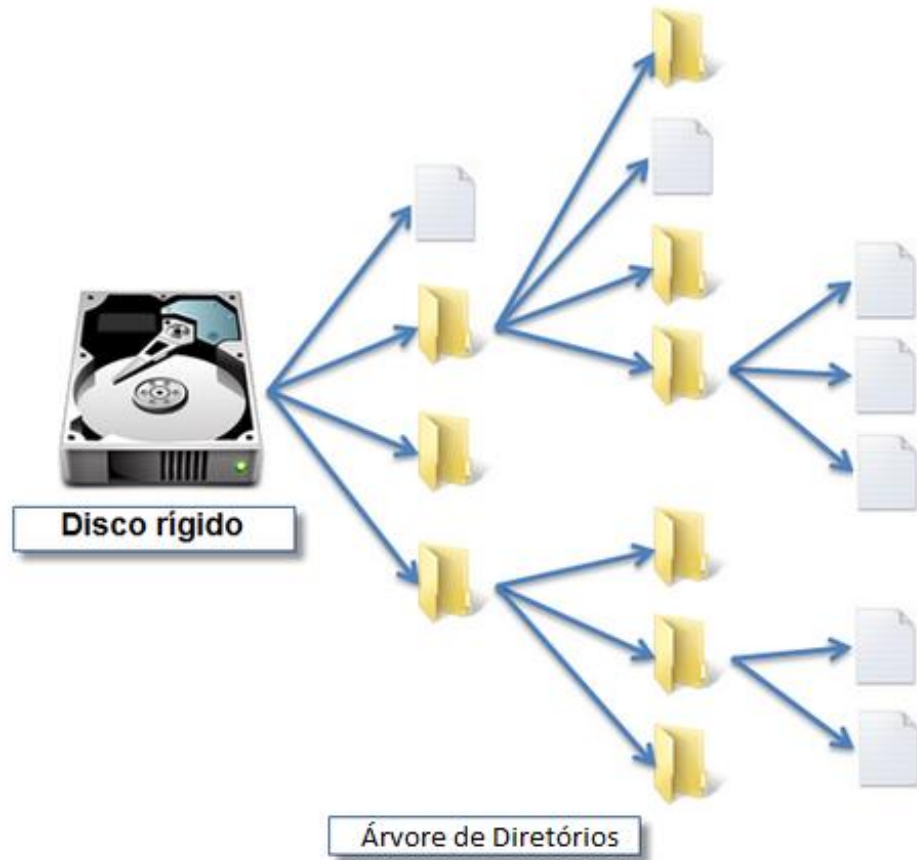
► Definição:

- São um tipo especial de Grafo;
- 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



Estrutura de Dados 2

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

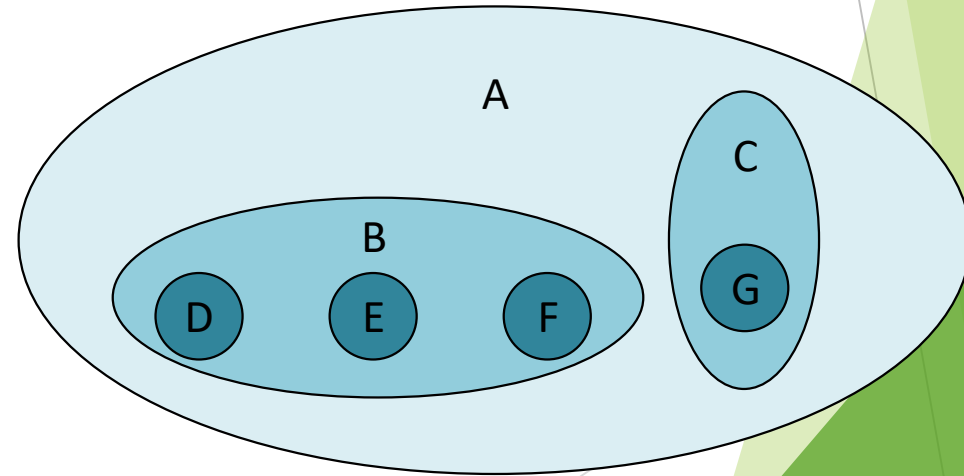
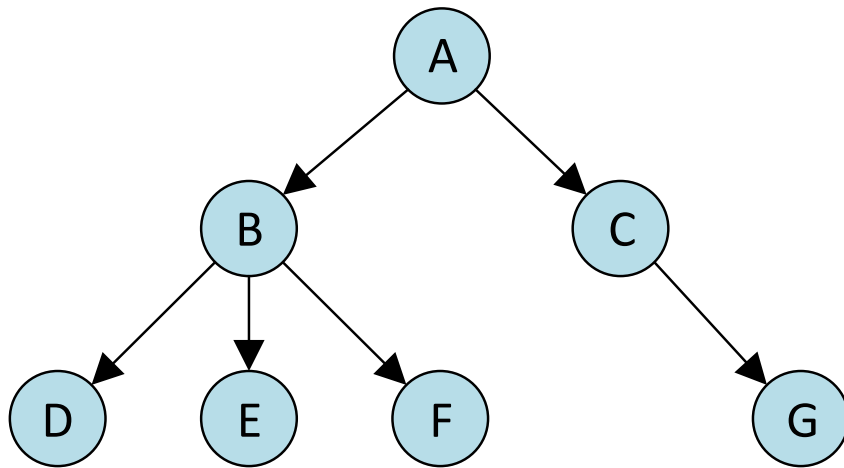
► Em computação:

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

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;
- Etc.

Árvores de Busca que são balanceadas para otimização da busca

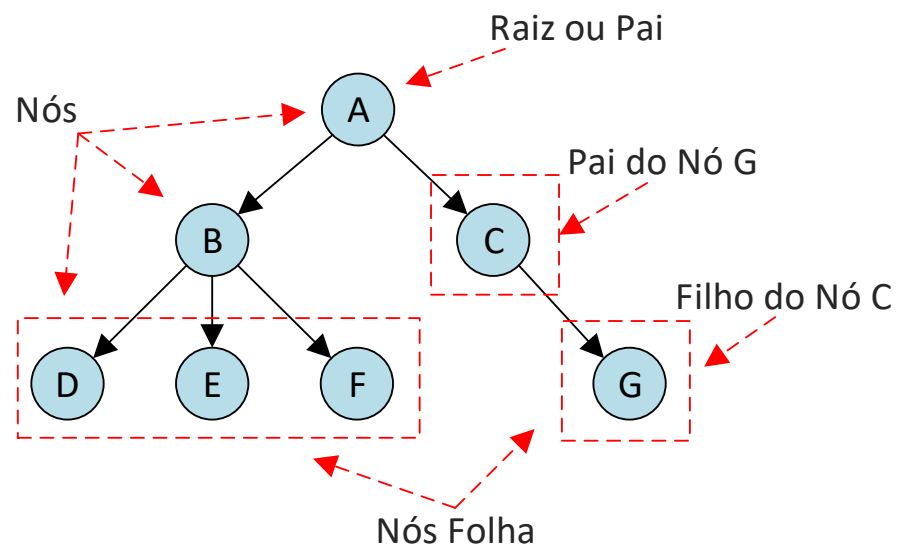
Usadas em programas de segmentação, imagem, volumes 3D, detecção de colisão, jogos, etc.

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

Árvores

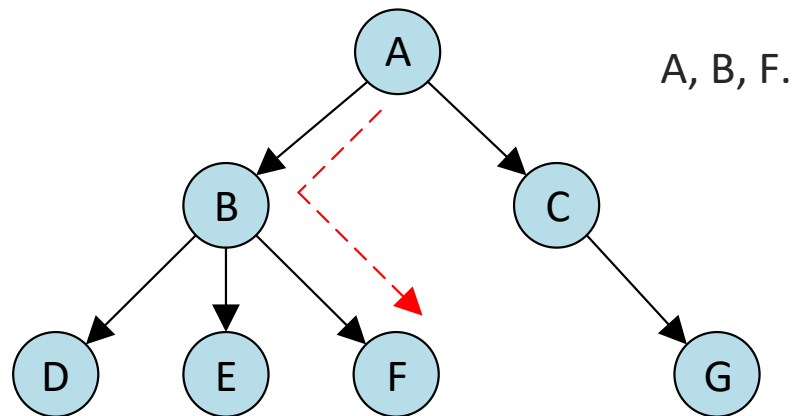
► Propriedades:

- **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**.



► 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.

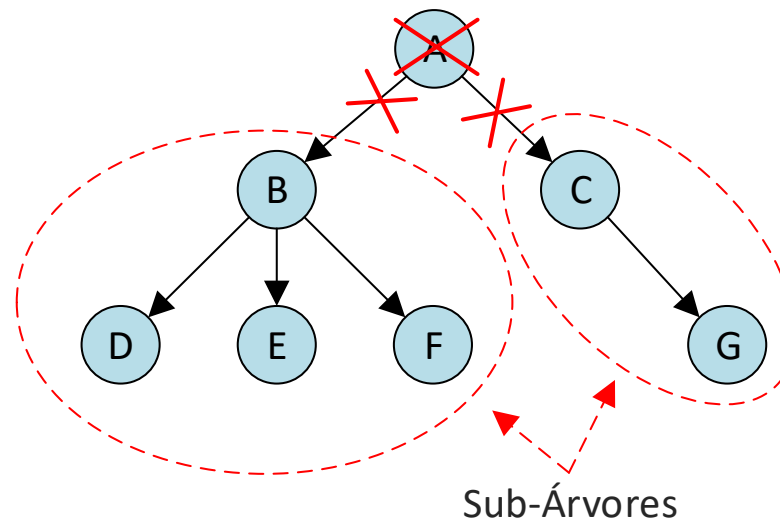
Árvores

► 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.

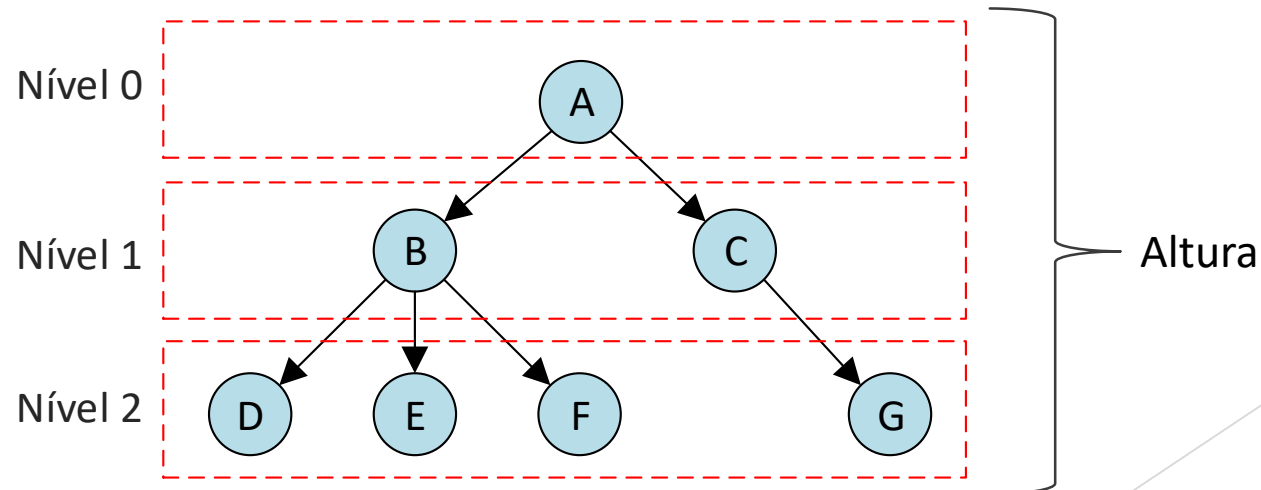
► Grau de um vértice:

- É o número de Sub-Árvores do vértice.



Árvores

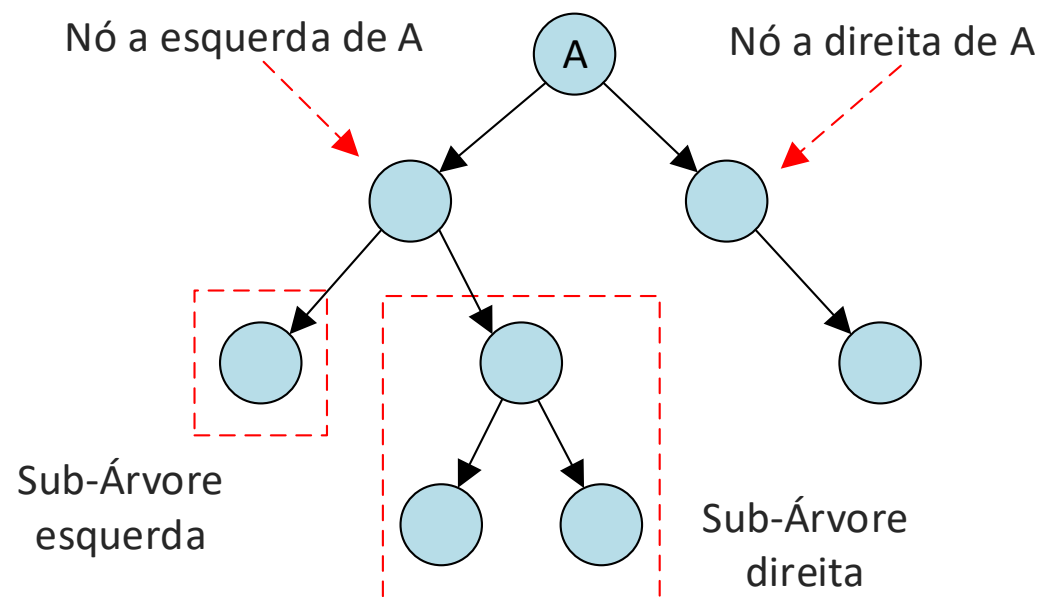
- ▶ Altura de uma Árvore:
 - Também chamada de profundidade;
 - É o comprimento do caminho mais longo da Raiz até uma das suas Folhas;
- ▶ Níveis:
 - Em uma Árvore, os vértices são classificados em níveis;
 - O nível é o número de Nós no caminho entre o vértice e a Raiz



Árvore Binária

► Árvore Binária:

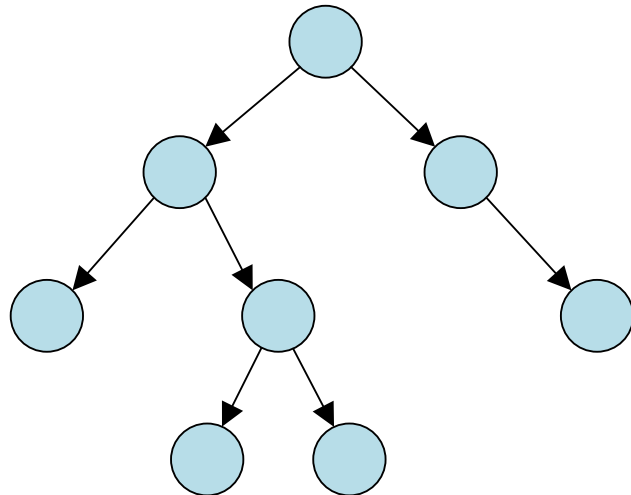
- É um tipo especial de Árvore;
- Cada vértice pode possuir duas Sub-Árvores: **Sub-Árvore esquerda** e **Sub-Árvore Direita**
- O Grau de cada vértice (número de Folhas), pode ser 0, 1 ou 2.



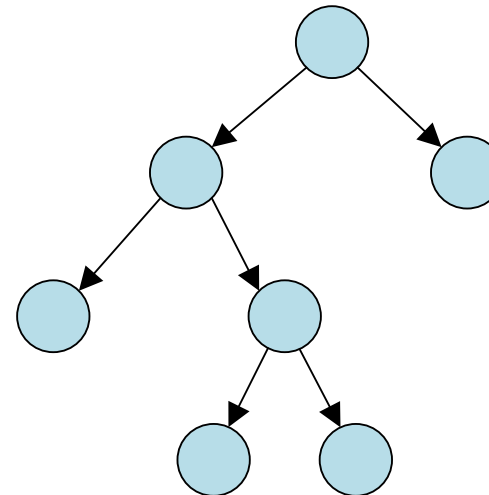
Árvore Binária

► 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

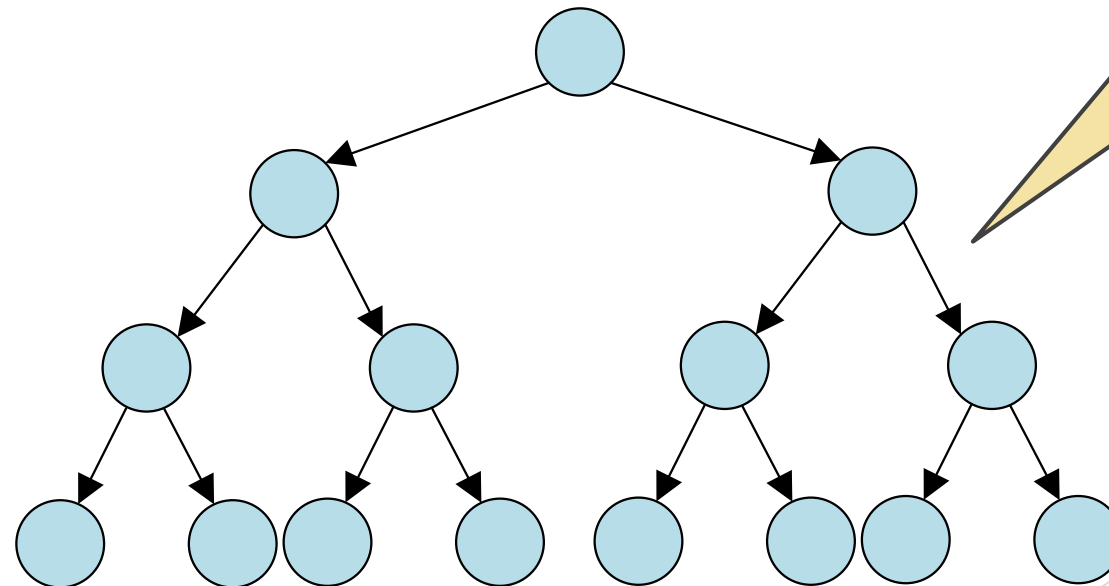


Árvore Estritamente Binária

Árvore Binária

► Árvore Binária Completa

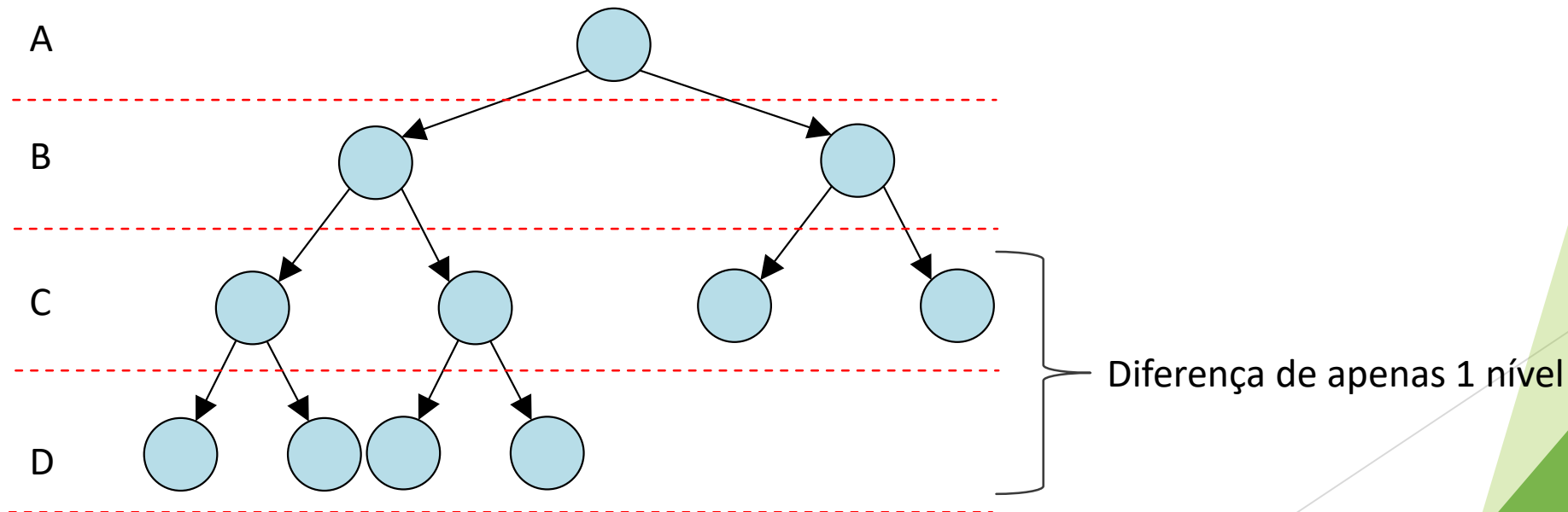
- É Estritamente Binária e todos os seus Nós-Folha estão no mesmo nível.
- O número de Nós de uma Árvore Binária Completa é $2^h - 1$, onde “ h ” é a altura da Árvore.



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:
 - 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).

Estrutura de Dados 2

Árvore Binária

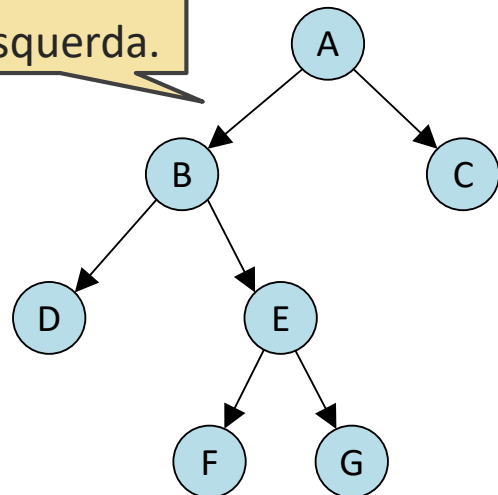
► Alocação Estática (Heap):

- Utiliza de Array;
- Utiliza duas funções para retornar a posição dos Filhos a esquerda e a direita de um Pai

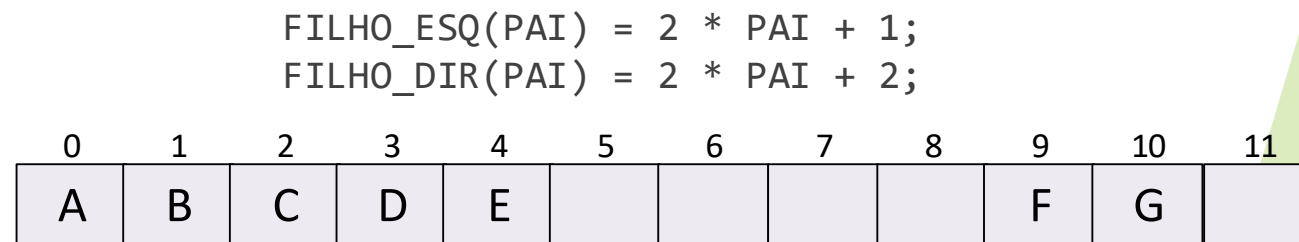
$$\text{FILHO_ESQ}(\text{PAI}) = 2 * \text{PAI} + 1;$$

$$\text{FILHO_DIR}(\text{PAI}) = 2 * \text{PAI} + 2;$$

Mais
elementos
na esquerda.



Essa representação é muito boa quando se sabe o tamanho da Árvore, como ela ficará depois de inseridos todos os elementos e quando é **Binária completa**. Ela ocupa muito bem o Vetor, sem desperdícios de espaço.

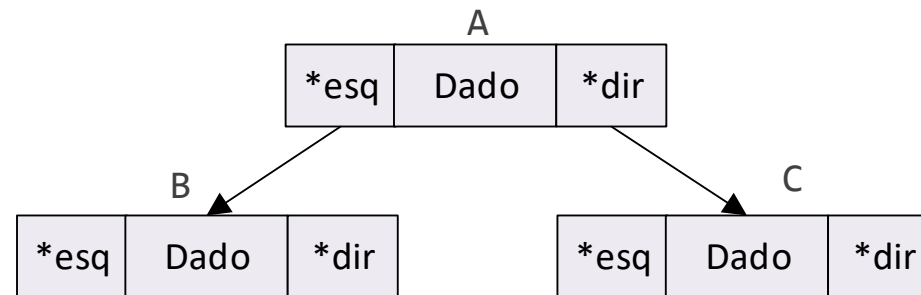
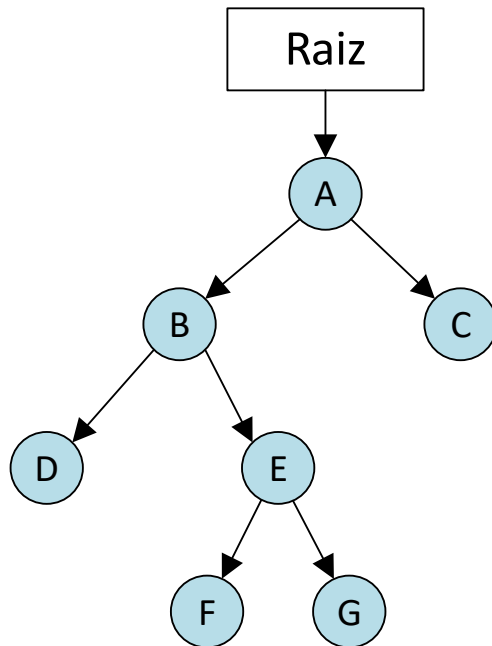


Não é boa quando temos muitos elementos alocados para um lado da Árvore: teremos muito espaço alocado sem utilização. Gera muitos espaços vazios.

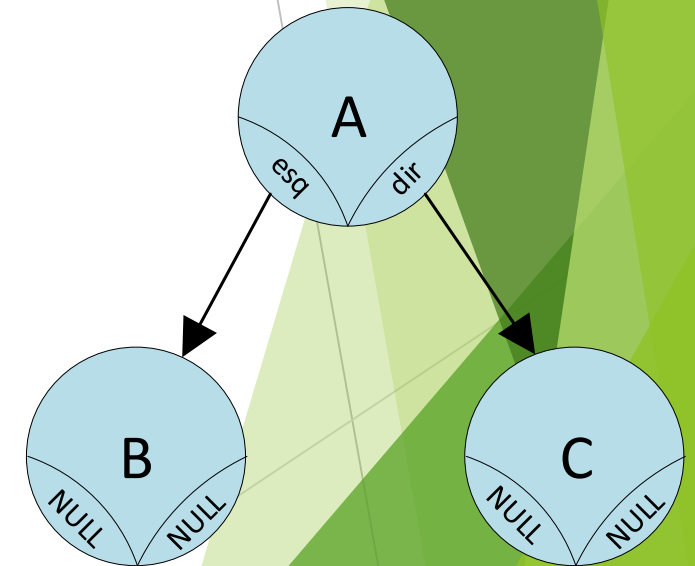
Árvore Binária

► Alocação Dinâmica – Lista Encadeada:

- Cada Nó da Árvore é tratado como um ponteiro alocado dinamicamente a medida que os dados são inseridos.



Não é mais necessário saber o tamanho. Os Nós serão criados a medida que são inseridos na Árvore.

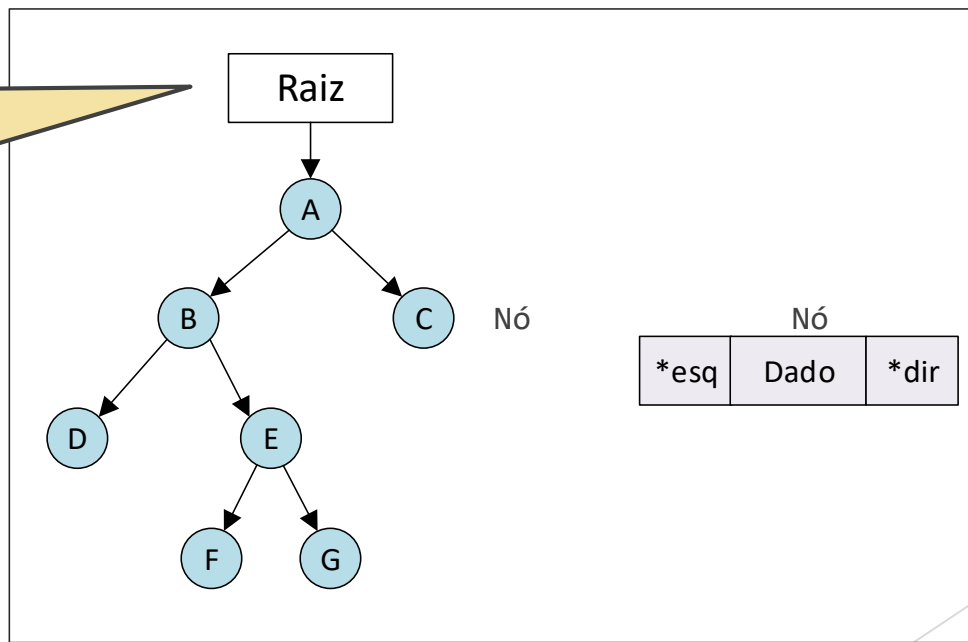


Árvore Binária

- Implementando uma Árvore Binária com alocação dinâmica - Lista Encadeada:
 - Para guardar o primeiro Nó da Árvore 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.

arvBin *raiz

Em uma Árvore Binária de Busca, não é tão necessário, mas em Árvores Balanceadas a raiz sempre muda



Estrutura de Dados 2

Árvore Binária

- ▶ Implementando uma Árvore Binária
 - ▶ `arvoreBinaria.h` – serão definidos:
 - ▶ Os protótipos das funções;
 - ▶ O tipo de dado armazenado na árvore;
 - ▶ O ponteiro árvore.
 - ▶ `arvoreBinaria.c` – serão definidos:
 - ▶ O tipo de dado árvore;
 - ▶ A implementação de suas funções.

Estrutura de Dados 2

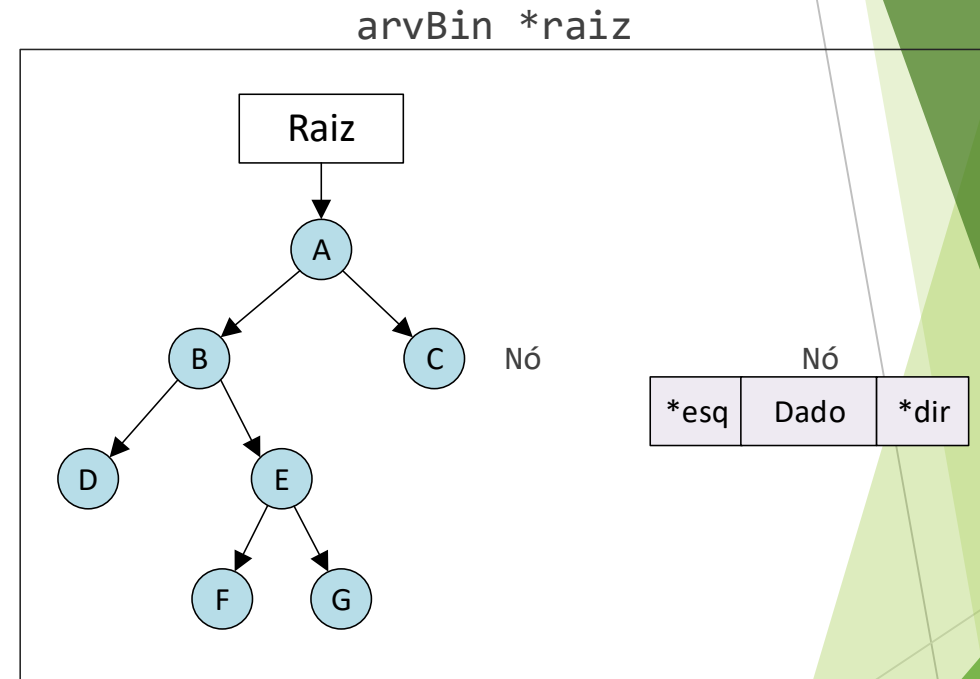
Árvore Binária

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

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

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

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



Estrutura de Dados 2

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 em uma
Árvore AVL...

► Destruição da Árvore:

- Envolve percorrer todos os Nós da Árvore de modo a liberar a memória alocada para cada um deles.

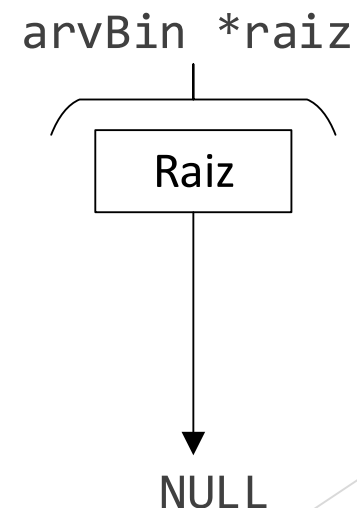
Estrutura de Dados 2

Árvore Binária

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

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

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



Estrutura de Dados 2

Árvore Binária

► Destruir a Árvore:

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

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

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

Libera cada Nó

Libera a Raiz

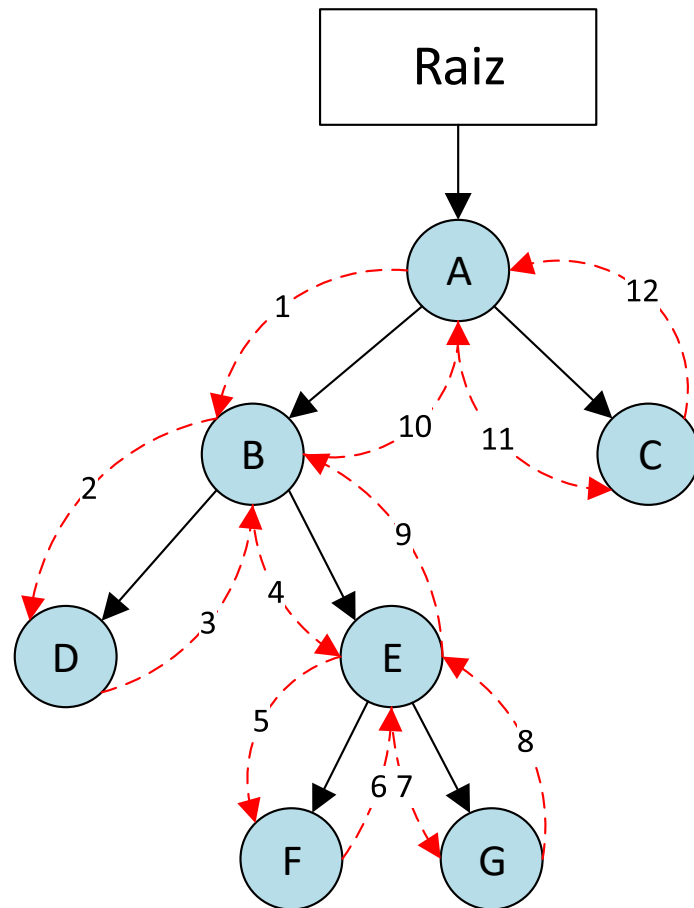
```
void libera_NO(struct NO *no) {  
    if (no == NULL) {  
        return;  
    }  
    libera_NO(no->esq);  
    libera_NO(no->dir);  
    free(no);  
    no = NULL;  
}
```

Percorre por recursão
todos os Nós,
esquerdos e direitos

Armazena NULL em Nó
para não ter problemas.

Estrutura de Dados 2

Árvore Binária



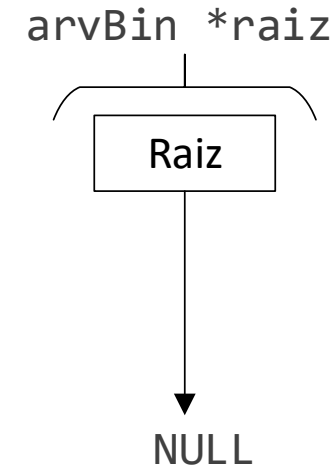
	1	Visita B
	2	Visita D
×	3	Libera D, volta para B
	4	Visita E
	5	Visita F
×	6	Libera F, volta para E
	7	Visita G
×	8	Libera G, volta para E
×	9	Libera E, volta para B
×	10	Libera B, volta para A
	11	Visita C
×	12	Libera C, volta para A
×		Libera A

Estrutura de Dados 2

Árvore Binária

► Informações Básicas:

- Está vazia?
- Número de Nós?
- Altura da Árvore?



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

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

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

Árvore Binária

► Altura da Árvore

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

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

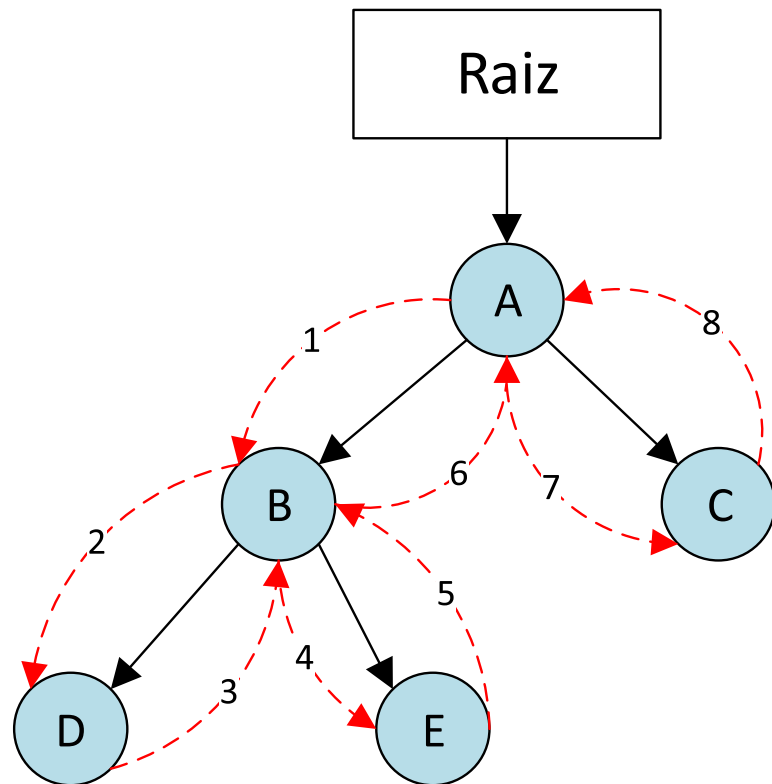
```
//Arquivo arvoreBinaria.c  
int altura_arvBin(ArvBin *raiz){  
    if(raiz == NULL){  
        return 0;  
    }  
    if(*raiz == NULL){  
        return 0;  
    }  
    int alt_esq = altura_arvBin(&((*raiz)->esq));  
    int alt_dir = altura_arvBin(&((*raiz)->dir));  
    if(alt_esq > alt_dir){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

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

A recursão então vai subindo no retorno somando 1

Endereço do Nó, para ficar igual a chamada da função recursiva.

Para saber a altura é necessário percorrer todos os Nós.



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.

Estrutura de Dados 2

Árvore Binária

► Número total de Nós

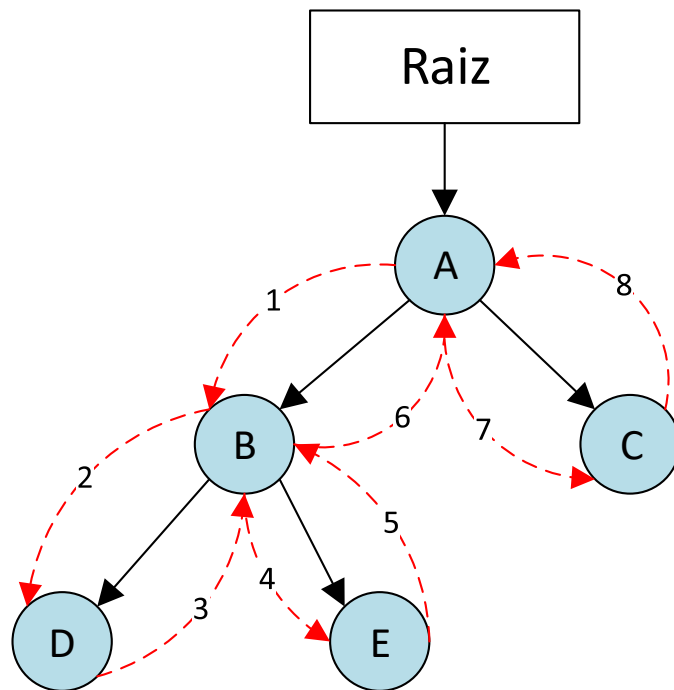
```
//Arquivo arvoreBinaria.h
int totalNO_arvBin(ArvBin *raiz);
```

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

```
//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.

► 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 sequencia linear de Nós, cuja ordem **sempre dependerá** de como a Árvore foi percorrida.

- ▶ 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.

Foi utilizada para liberar Nós da Árvore, na contagem dos Nós e na verificação da altura da Árvore.

Árvore Binária

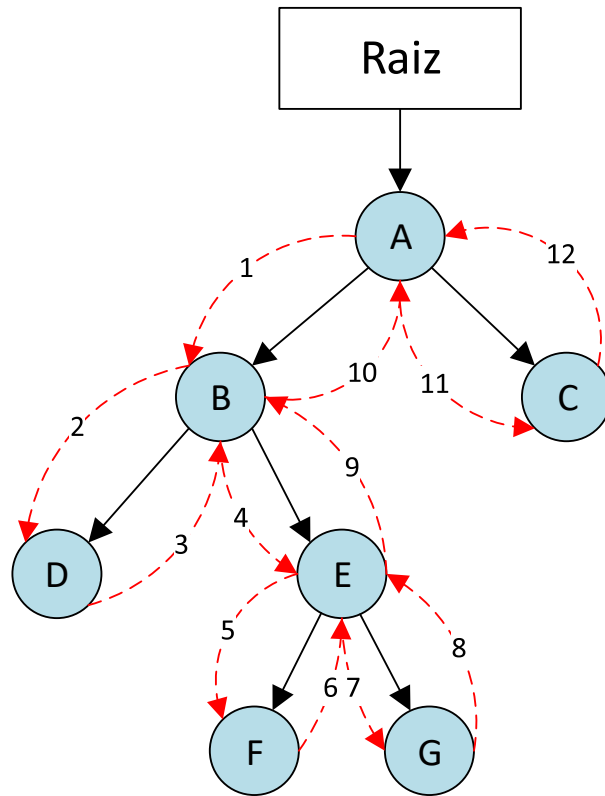
► Pré-Ordem

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

//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));
    }
}

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

Árvore Binária



Resultado: ABDEFGC

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

Estrutura de Dados 2

Árvore Binária

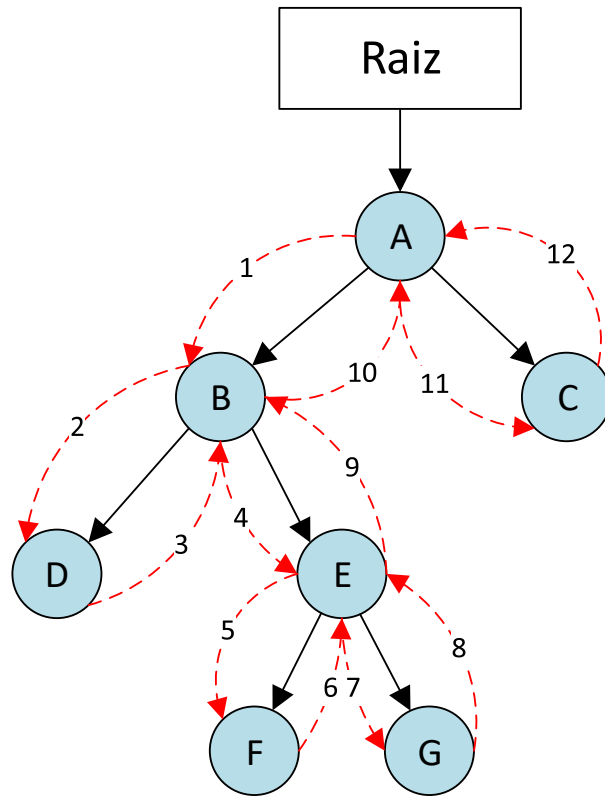
► Em-Ordem

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

//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));
    }
}

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

Árvore Binária



Resultado: DBFEGAC

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

Estrutura de Dados 2

Árvore Binária

► Pós-Ordem

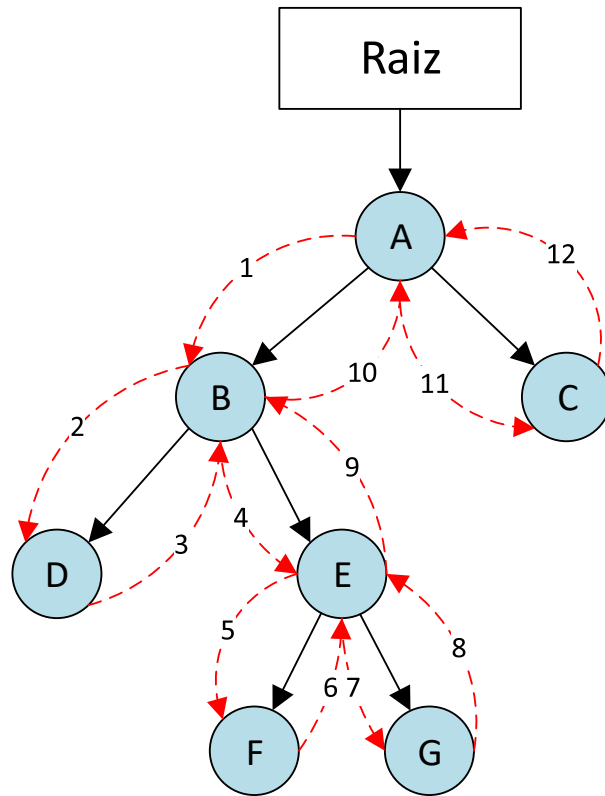
```
//Arquivo arvoreBinaria.h  
void posOrdem_arvBin(ArvBin *raiz);
```

```
//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);  
    }  
}
```

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

Este método garante que todos os Filhos serão visitados antes de se executar qualquer coisa com o Pai, como por exemplo, a sua exclusão.

Árvore Binária



Resultado: DFGEBCA

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

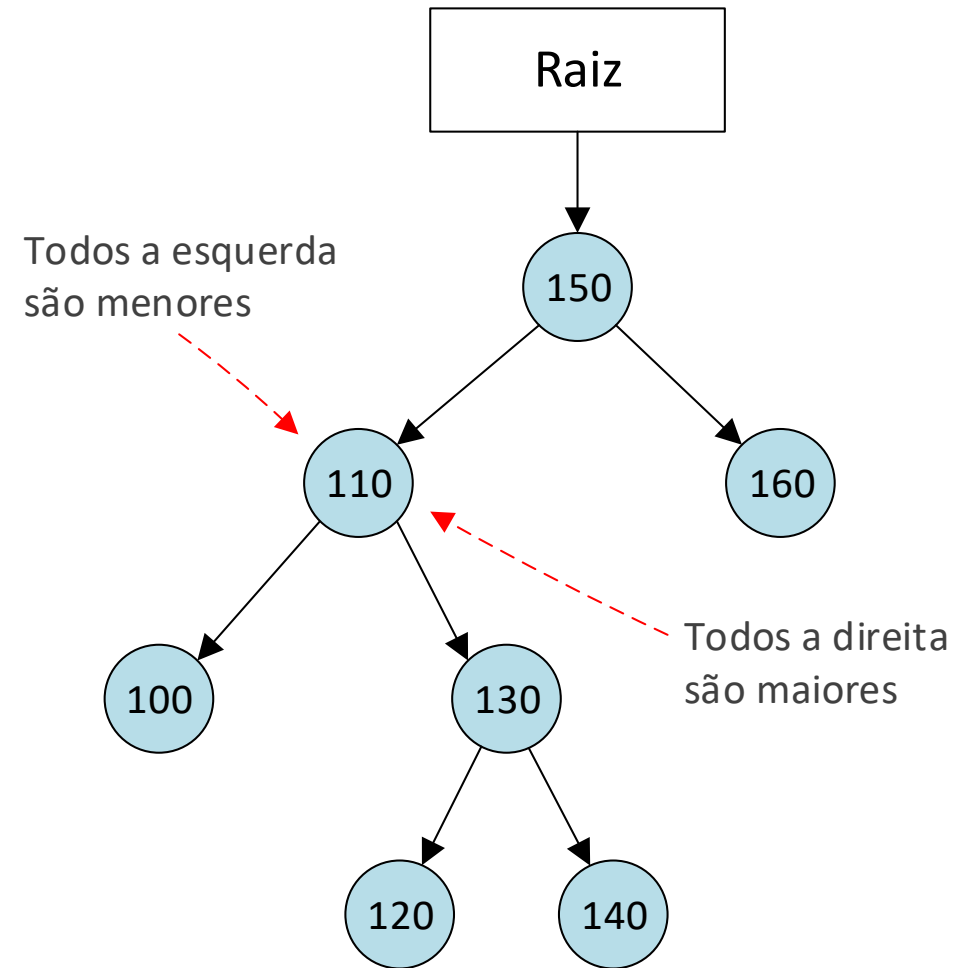
Árvore Binária

- ▶ **Árvore Binária de Busca:**
 - É um tipo de **Árvore Binária** onde cada Nó possui um **valor** (chave), associado a ele , e esse valor determina a posição do Nó na Árvore;
 - Assumiremos que **não existem valores repetidos** (não trabalharemos com eles).

- ▶ **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

Estrutura de Dados 2

Árvore Binária



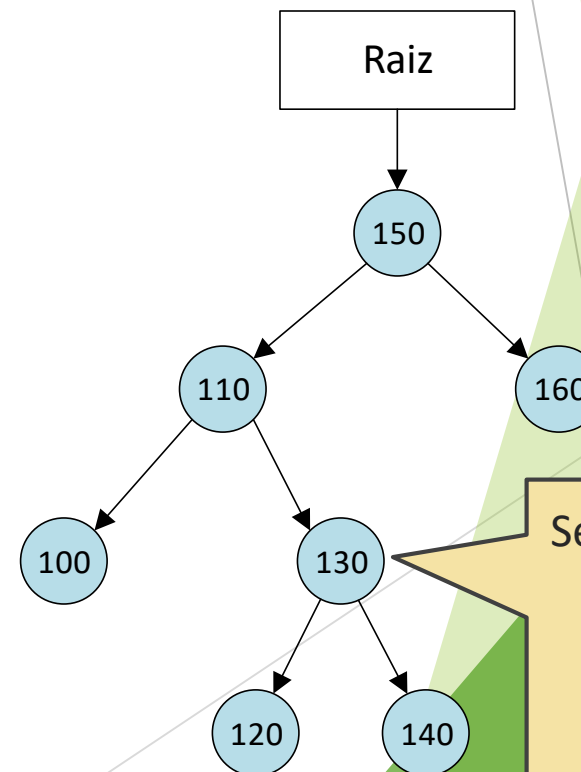
Estrutura de Dados 2

Árvore Binária

- ▶ 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 remover o 130, tem que haver tratamento para definir qual Nó vai ocupar seu lugar.

Estrutura de Dados 2

Árvore Binária

► Principais operações:

► Inserção

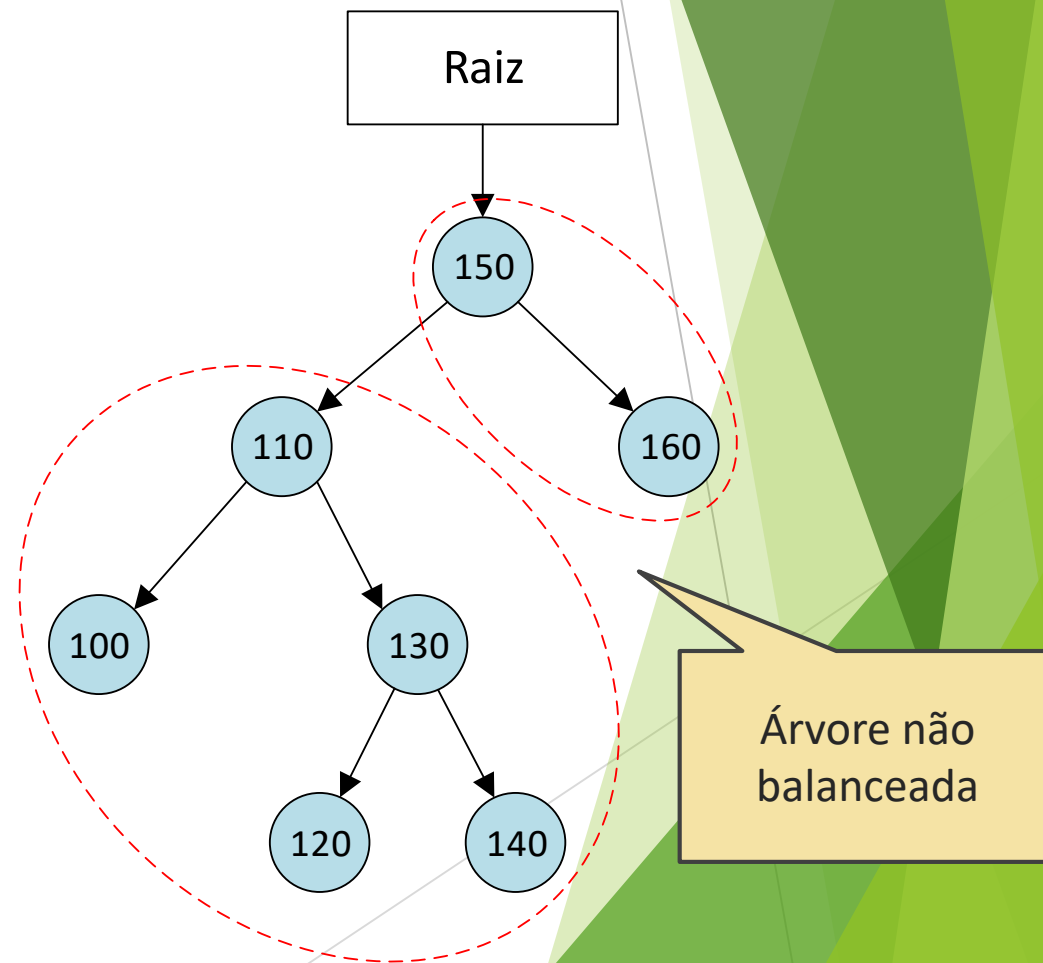
- Caso médio $O(\log n)$;
- Pior caso $O(n)$. (Árvore não balanceada)

► Remoção

- Caso médio $O(\log n)$;
- Pior caso $O(n)$. (Árvore não balanceada)

► Consulta

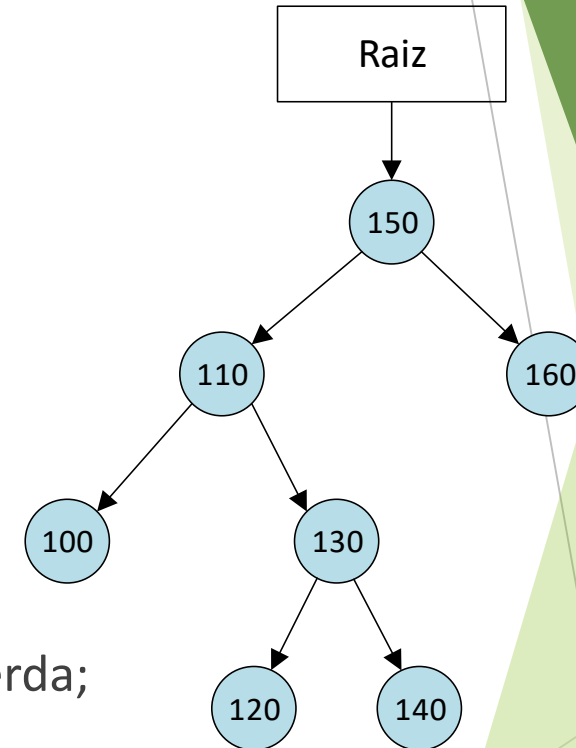
- Caso médio $O(\log n)$;
- Pior caso $O(n)$. (Árvore não balanceada)



Inserção na Árvore Binária de Busca:

► Para inserir um elemento (V), na Árvore Binária de Busca :

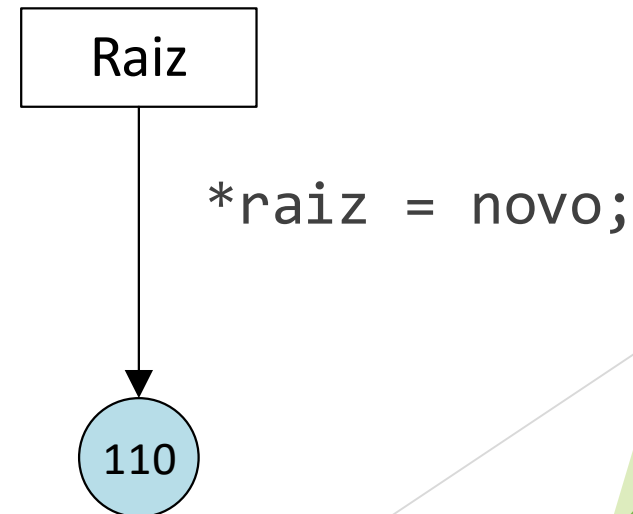
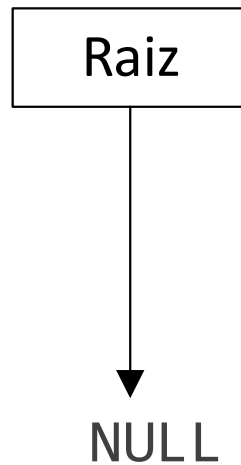
- Primeiro compare com a Raiz, e se:
 - V é menor do que a raiz, vá para a Sub-Árvore da esquerda;
 - V é maior do que a raiz, vá para a Sub-Árvore da direita;
- Aplique o método recursivamente (também pode ser aplicado sem recursão).



Estrutura de Dados 2

Inserção na Árvore Binária de Busca:

- Também existe o caso onde a inserção é feita em uma Árvore que está vazia:



Estrutura de Dados 2

Inserção na Árvore Binária de Busca:

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

```
//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){
```

Sempre um novo
NÓ, será uma folha.

Quando
atual
receber
NULL, ele
chegou na
folha.

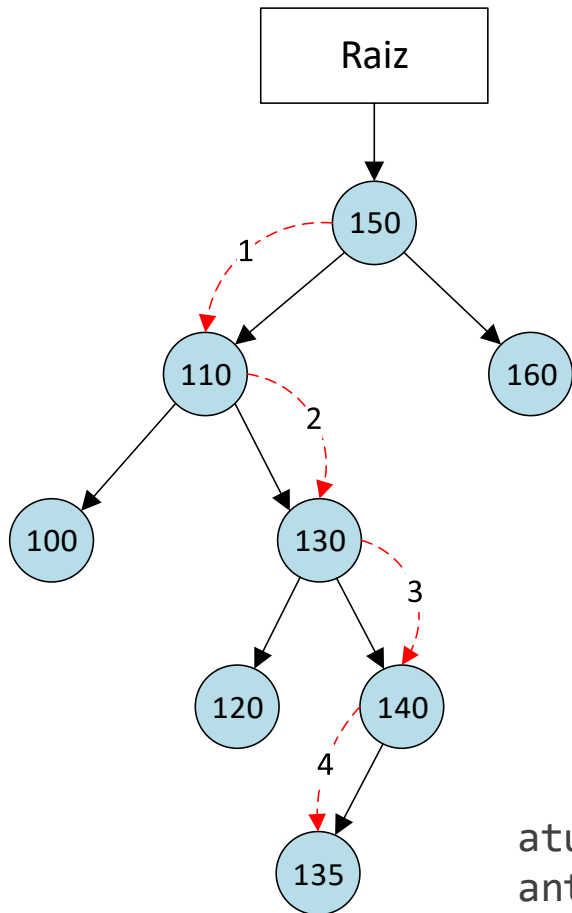
Inserir como
filho desse
NÓ Folha.

```
    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;  
}
```

Guarda o atual
antes dele
mudar. Quando
atual passar a
apontar p/ NULL,
ant ainda aponta
p/ a última 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.

Estrutura de Dados 2

Inserção na Árvore Binária de Busca:

```
//programa principal  
raiz = cria_arvBin();  
  
x = insere_arvBin(raiz, 150);  
x = insere_arvBin(raiz, 110);  
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);
```

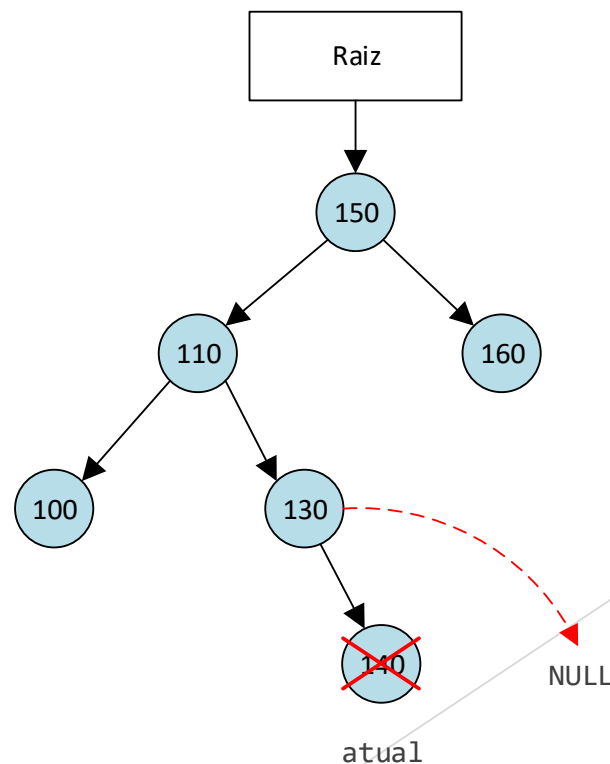
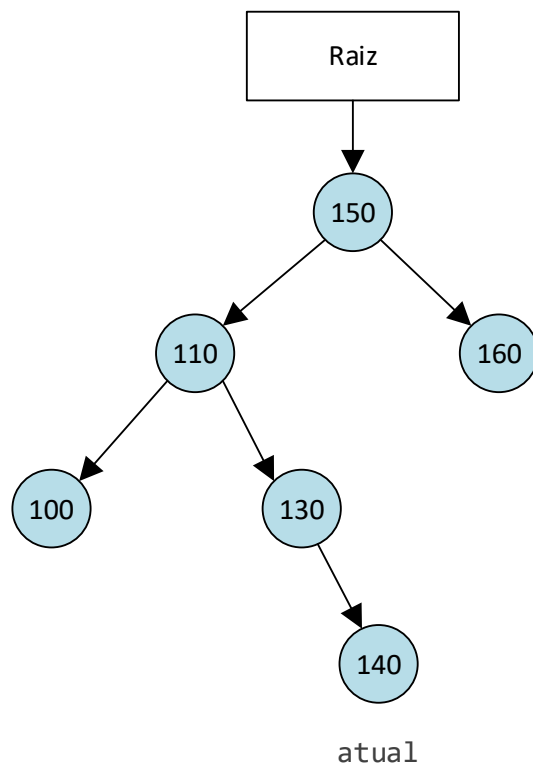
Estrutura de Dados 2

Remoção na Árvore Binária de Busca:

► Existem 3 tipos e remoção em Árvores Binárias de Busca:

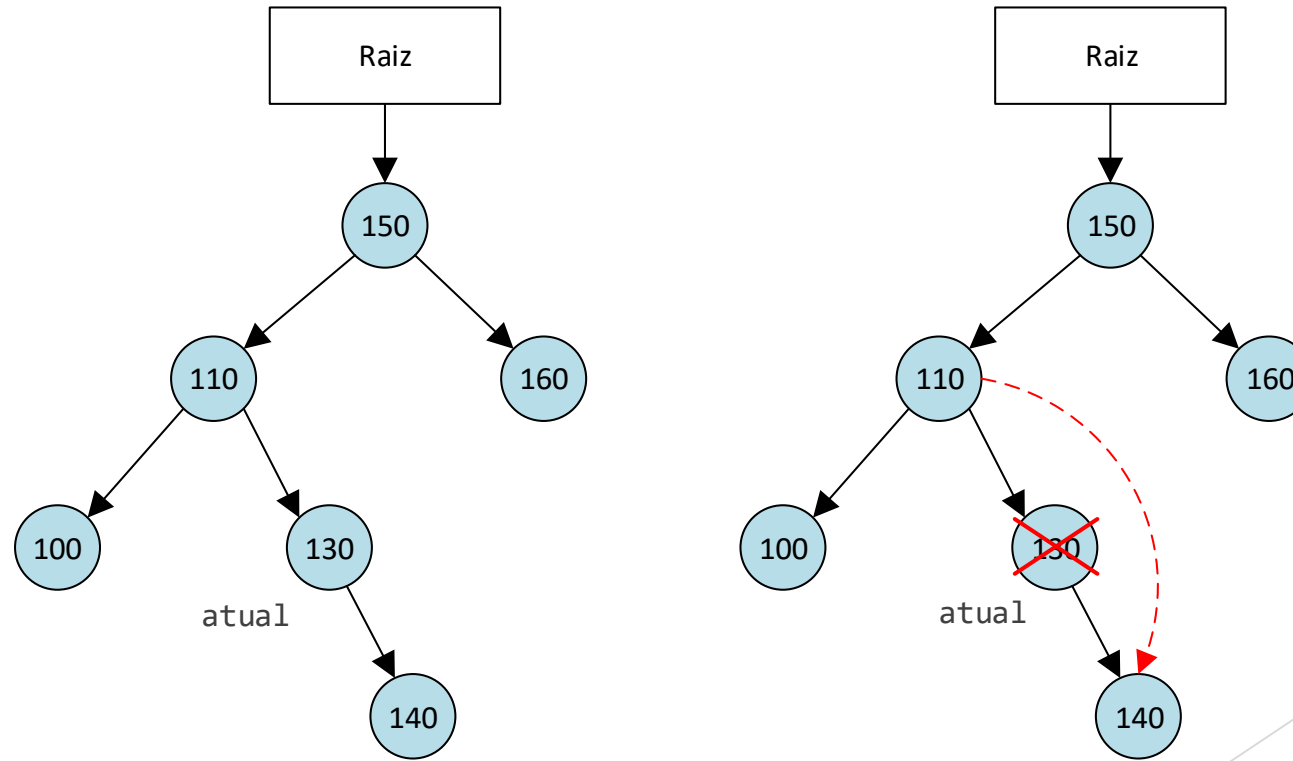
- Nó Folha (sem Filhos);
- Nó com 1 Filho;
- Nó com 2 Filhos.

Remoção de Nó
Folha sem Filhos



Estrutura de Dados 2

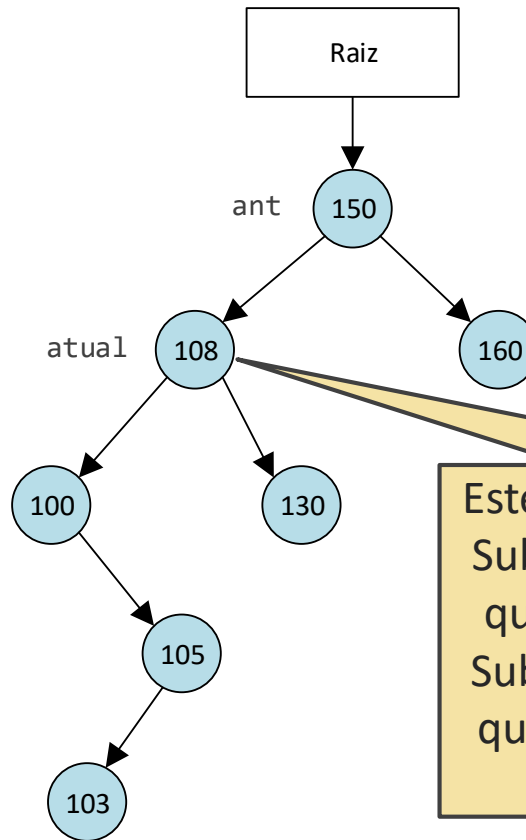
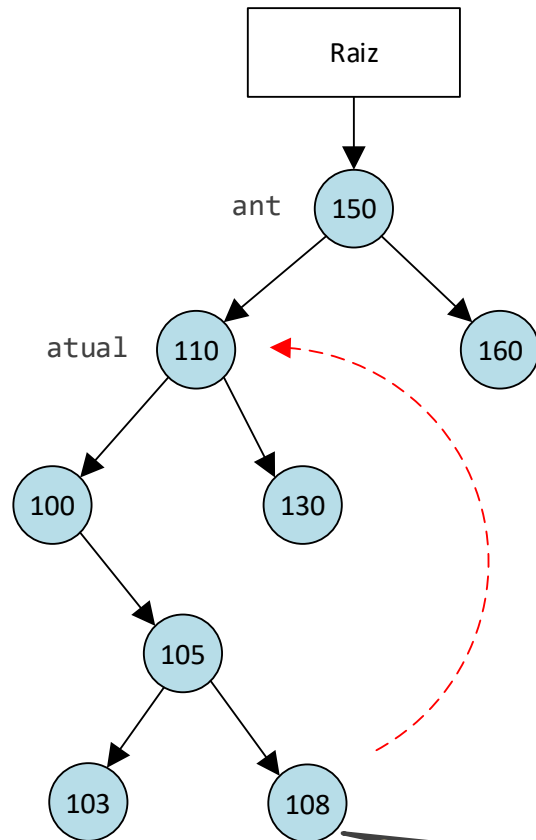
Remoção na Árvore Binária de Busca:



Remoção de Nó
Folha com 1 Filho

Estrutura de Dados 2

Remoção na Árvore Binária de Busca:



Remoção de Nó
Folha com 2 Filhos

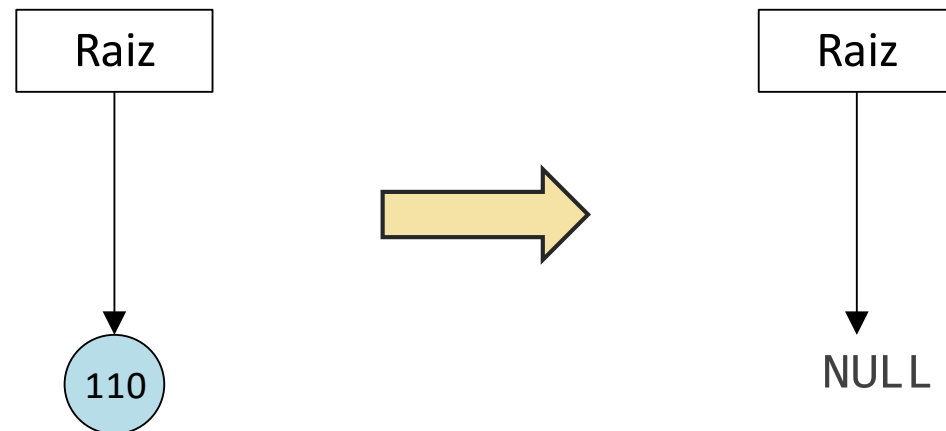
Este Filho que estava mais a direita na Sub-Árvore da esquerda, é menor do que qualquer elemento que está na Sub-Árvore da direita e maior do que qualquer elemento que está na Sub-Árvore da esquerda.

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.

Estrutura de Dados 2

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:

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

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

```
//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

```
//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;
        }
    }
}
```

Achou o Nó a ser removido: tratar o lado da remoção

Continua andando na Árvore a procura do Nó a ser removido

Estrutura de Dados 2

```
//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;
}
```

Sem Filho da esquerda,
apontar para filho da direita.
Este bloco trata Nó Folha e
Nó com 1 Filho.

Procura filho mais a direita
na Sub-Árvore da esquerda.

Copia o Filho mais a direita na
Sub-Árvore da esquerda para
o lugar do Nó removido.

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

Estrutura de Dados 2

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).

Estrutura de Dados 2

Consulta na Árvore Binária de Busca

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

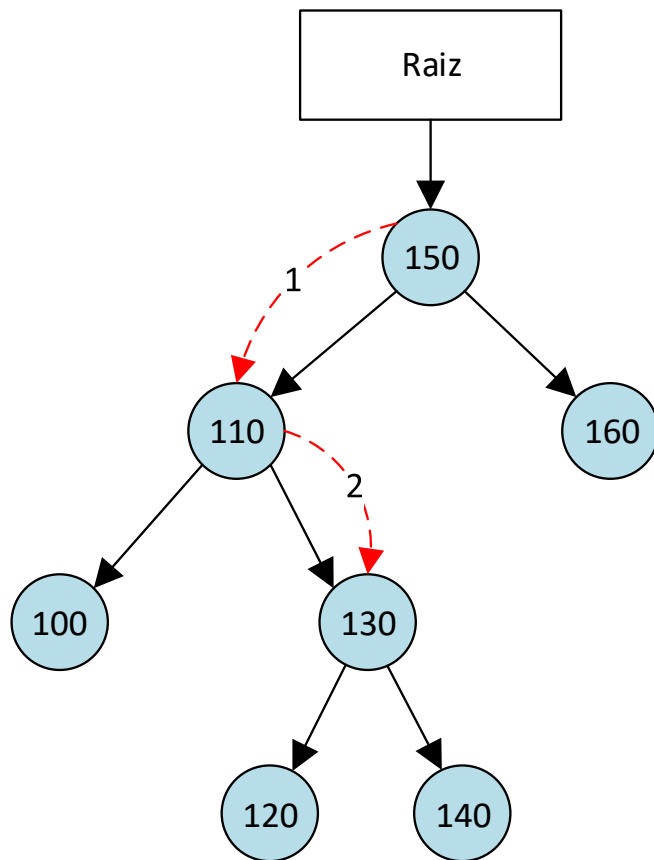
```
//Arquivo arvoreBinaria.c
int consulta_arvBin(ArvBin *raiz, int valor){
    if(raiz == NULL){
        return 0;
    }
    struct NO *atual = *raiz;
    while(atual != NULL){
        if(valor == atual->info){
            return 1;
        }
        if(valor > atual->info){
            atual = atual->dir;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

```
//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 valor
não foi encontrado.

Estrutura de Dados 2

Consulta na Árvore Binária de Busca

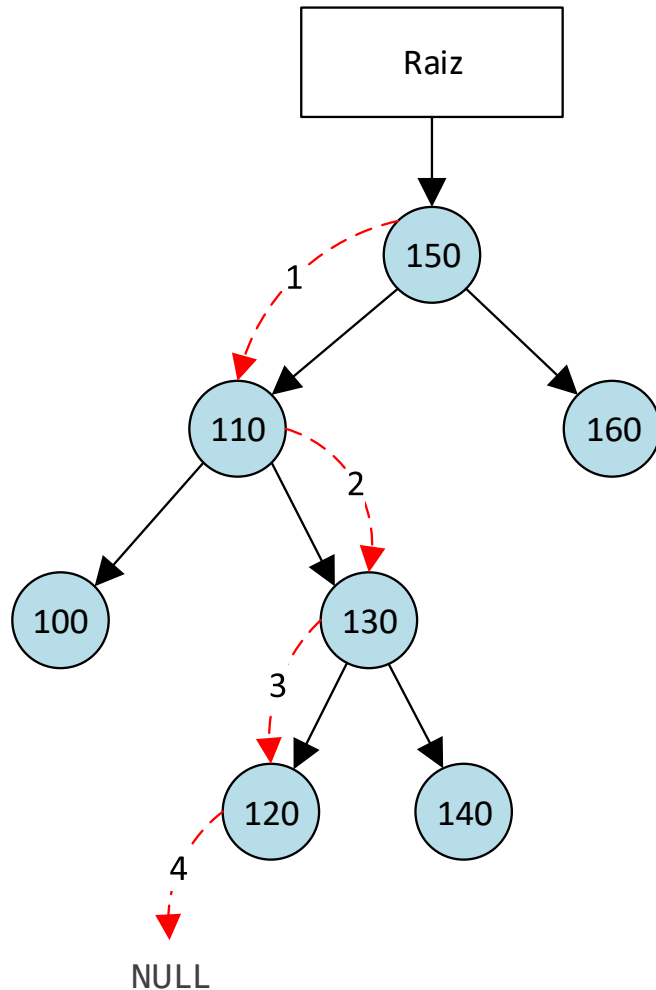


Valor procurado: 130

- 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;

Estrutura de Dados 2

Consulta na Árvore Binária de Busca



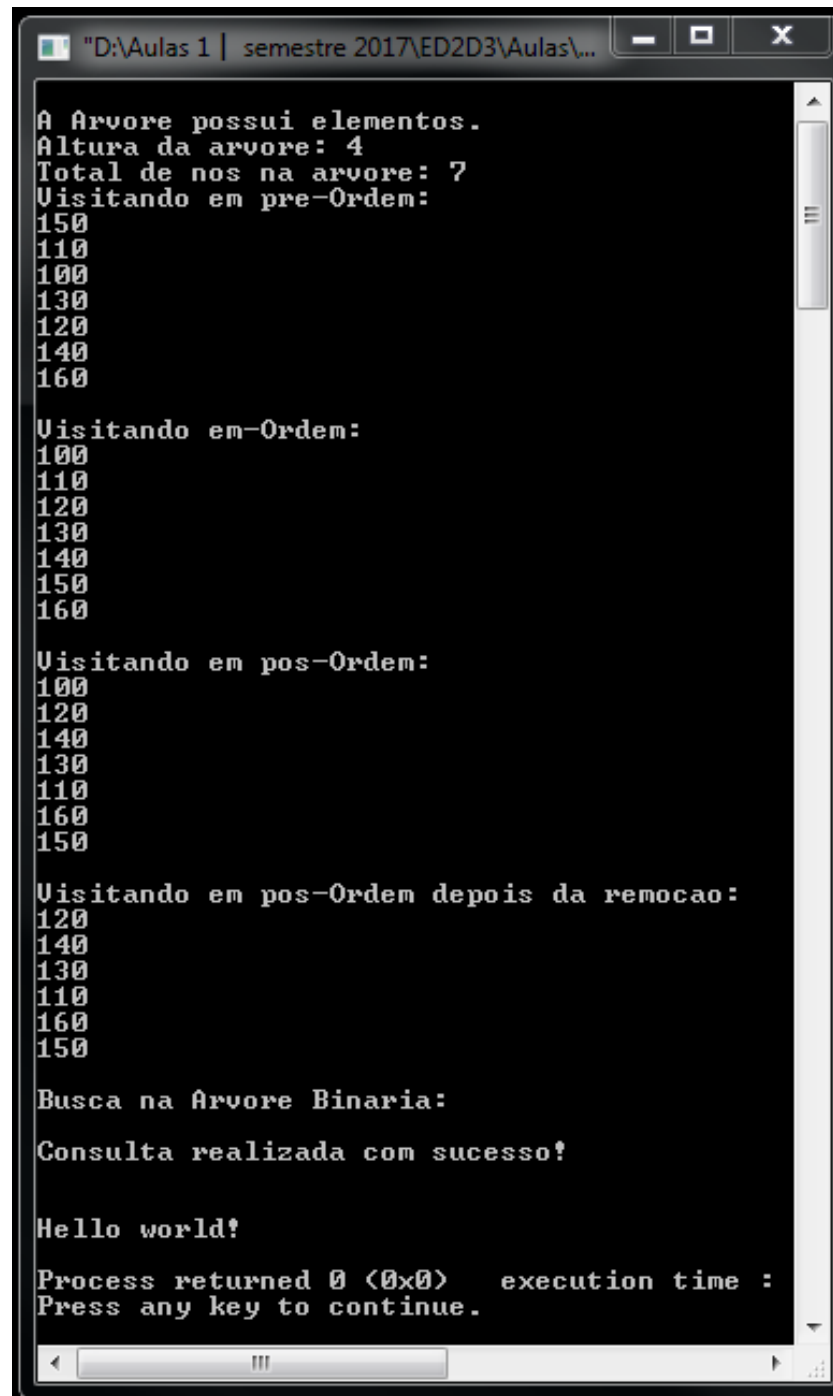
Valor procurado: 115

- 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.



```
"D:\Aulas 1 | semestre 2017\ED2D3\Aulas\..."
A Arvore possui elementos.
Altura da arvore: 4
Total de nos na arvore: 7
Visitando em pre-Ordem:
150
110
100
130
120
140
160

Visitando em-Ordem:
100
110
120
130
140
150
160

Visitando em pos-Ordem:
100
120
140
130
110
160
150

Visitando em pos-Ordem depois da remocao:
120
140
130
110
160
150

Busca na Arvore Binaria:
Consulta realizada com sucesso!

Hello world!

Process returned 0 (0x0)   execution time :
Press any key to continue.
```