
Árvores Binárias

9/11 e 11/11

Conceitos

Representação e Implementação

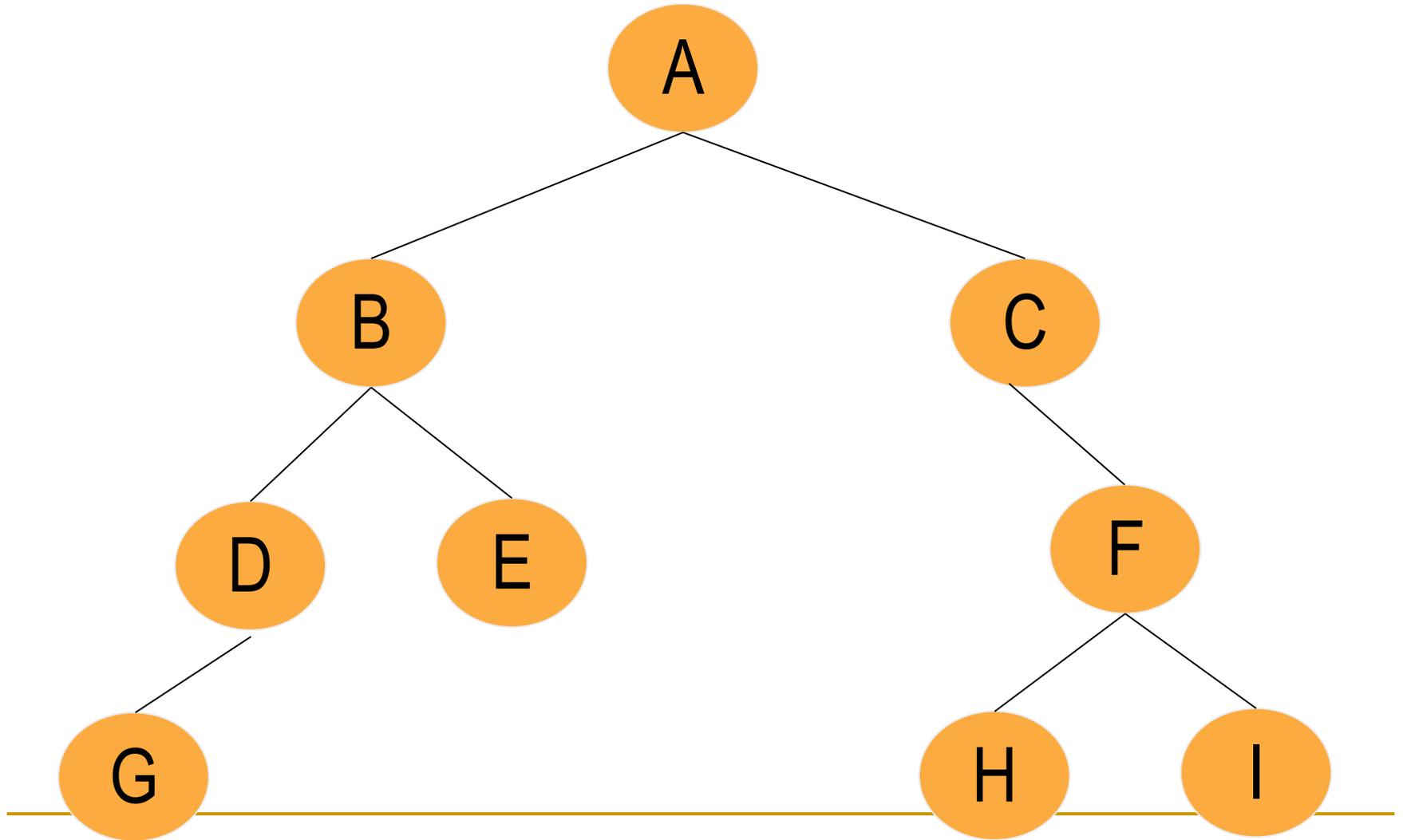
Árvore Binárias (AB)

- Uma Árvore Binária (AB) T é um conjunto finito de elementos, denominados nós ou vértices, tal que:
 - (i) Se $T = \emptyset$, a árvore é dita vazia, ou
 - (ii) T contém um nó especial, chamado raiz de T , e os demais nós podem ser subdivididos em
 - dois sub-conjuntos distintos T_E e T_D , os quais também são árvores binárias.
 - T_E e T_D são denominados sub-árvore esquerda e sub-árvore direita de T , respectivamente

Árvore Binárias (AB) (cont.)

- A raiz da sub-árvore esquerda (direita) de um nó v , se existir, é denominada filho esquerdo (direito) de v . Pela natureza da árvore binária, o filho esquerdo pode existir sem o direito, e vice-versa
- Se r é a raiz de T , diz-se que T_{Er} e T_{Dr} são as sub-árvores esquerda e direita de T , respectivamente
- **Árvore Binária é uma árvore ordenada de grau 2.**

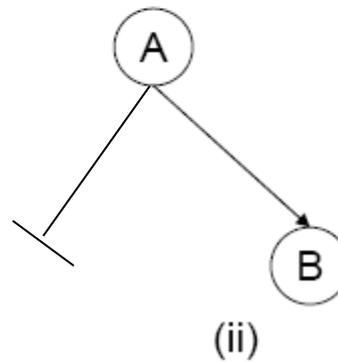
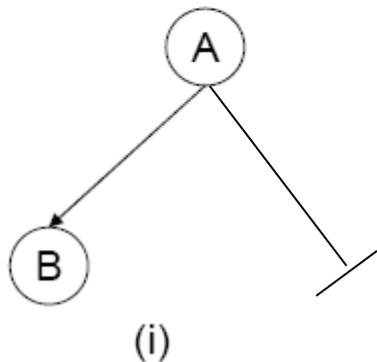
Exemplos de Árvore Binárias (AB)



Exemplos de Árvores Binárias

As duas AB seguintes são distintas

- (i) a primeira tem subárvore direita vazia
- (ii) a segunda tem subárvore esquerda vazia



Eficiência para as operações

- Árvores binárias podem ser usadas para guardar e recuperar informações, com número de operações proporcional à altura da árvore,
 - ou seja, variando, aproximadamente, entre $\log_2 n$ e n (quando se torna degenerada para uma lista).
- Mais tarde veremos como esta manipulação pode ser realizada de maneira a garantir a altura da ordem de $O(\log_2 n)$

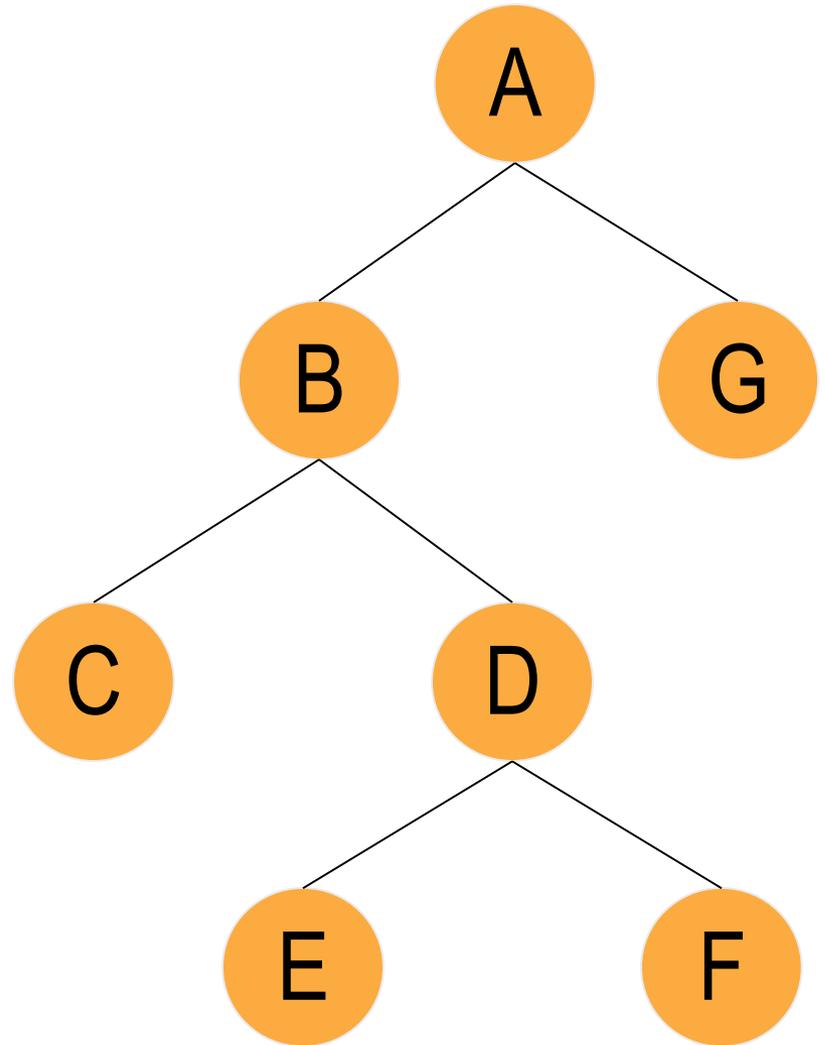
Tipos de AB

- Árvore Estritamente Binária
 - Árvore Binária Completa
 - Há duas definições: (i) completa = cheia e (ii) completa = a definição de quase completa, dada abaixo
 - Árvore Binária Quase Completa

 - Árvore Binária Balanceada (**altura** difere max 1)
 - Árvore Binária Perfeitamente Balanceada (número de **nós** difere max 1)
-

Árvore Estritamente Binária

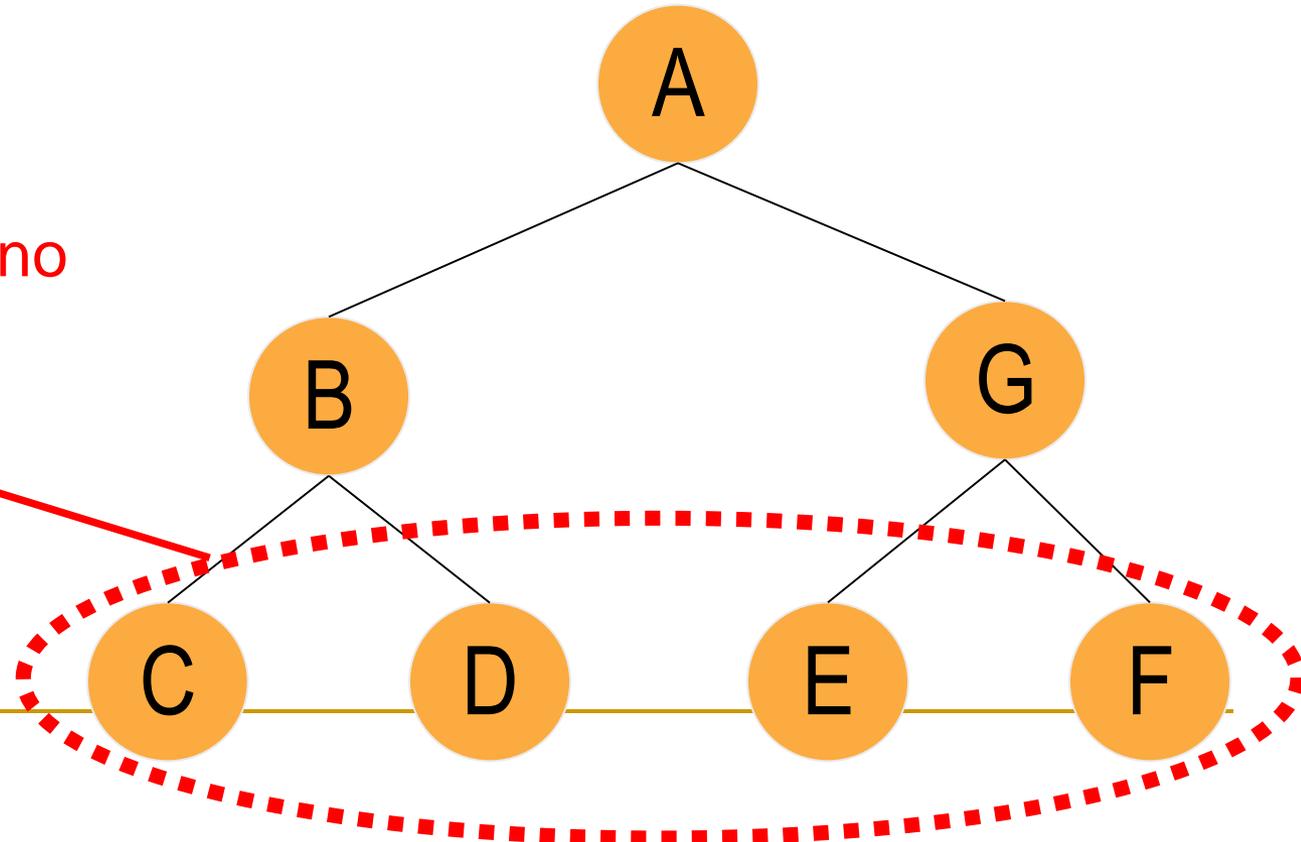
- Uma **Árvore Estritamente Binária** tem nós que têm ou 0 (nenhum) ou dois filhos
- Nós internos (não folhas) sempre têm 2 filhos



Árvore Binária Completa (Cheia)

- **Árvore Binária Completa (Cheia) (ABC)**
 - é estritamente binária, de nível d ; e
 - todos os seus nós-folha estão no mesmo nível (d)

C,D,E,F estão no nível 3
(altura = 3)



Árvore Binária Completa (Cheia)

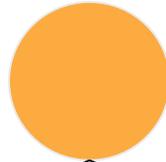
- Dada uma ABC e sua altura, pode-se calcular o número total de nós na árvore
 - p.ex., uma ABC com altura 3 tem 7 nós
 - Nível 1: => 1 nó
 - Nível 2: => 2 nós
 - Nível 3: => 4 nós
 - No. Total de nós = $1 + 2 + 4 = 7$
 - Verifique que: se uma ABC tem altura h , então o número de nós da árvore é dado por:

$$N = 2^h - 1$$

Nível

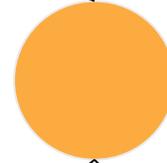
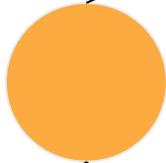
Número de nós por nível

1



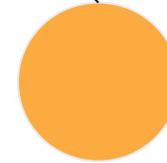
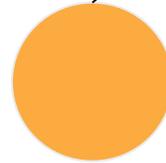
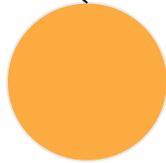
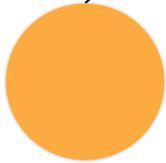
$$1 = 2^0$$

2



$$2 = 2^1$$

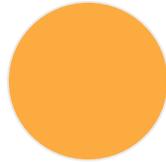
3



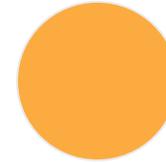
$$4 = 2^2$$

.....

h



.....



$$2^{h-1}$$

$$\therefore N = \sum_{i=0}^{h-1} 2^i = 2^h - 1$$

Inversamente:

- Se **N** é o número de nós de uma ABC, de grau **d**, qual é a altura **h** da árvore?

$$N(h) = \frac{d^h - 1}{d - 1}$$

$$h = \log_d(N \cdot d - N + 1)$$

$$\text{para } d=2: h = \log_2(N + 1)$$

Árvore Binária Quase Completa

- **Árvore Binária Quase Completa**
 - Se a altura da árvore é d , cada nó folha está no nível d ou no nível $d-1$.
 - Em outras palavras, a diferença de altura entre as sub-árvores de qualquer nó é no máximo 1.

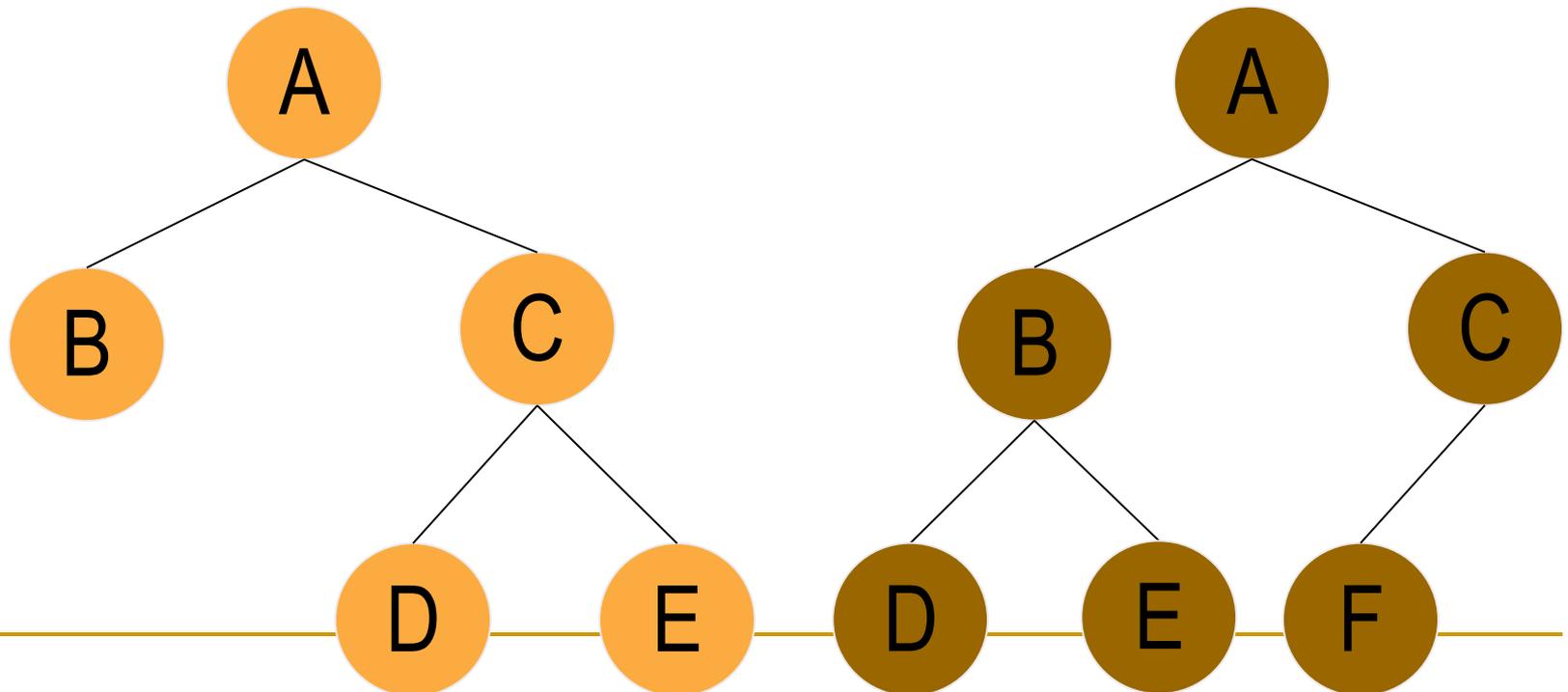
Implicações dos conceitos completa (cheia) e quase completa

Importante para:

- sua alocação em vetores, pois quando é cheia não desperdiça espaço, e
- na definição do método de ordenação heapsort que trabalha com o conceito de **árvore completa** como se fosse nossa definição de quase completa.
 - Neste método a árvore (heap) é preenchida da esquerda para direita.
- Uma **árvore completa** é aquela em que se n é um nó com algumas de subárvores vazias, então n se localiza no penúltimo ou no último nível.

Árvore Binária Balanceada

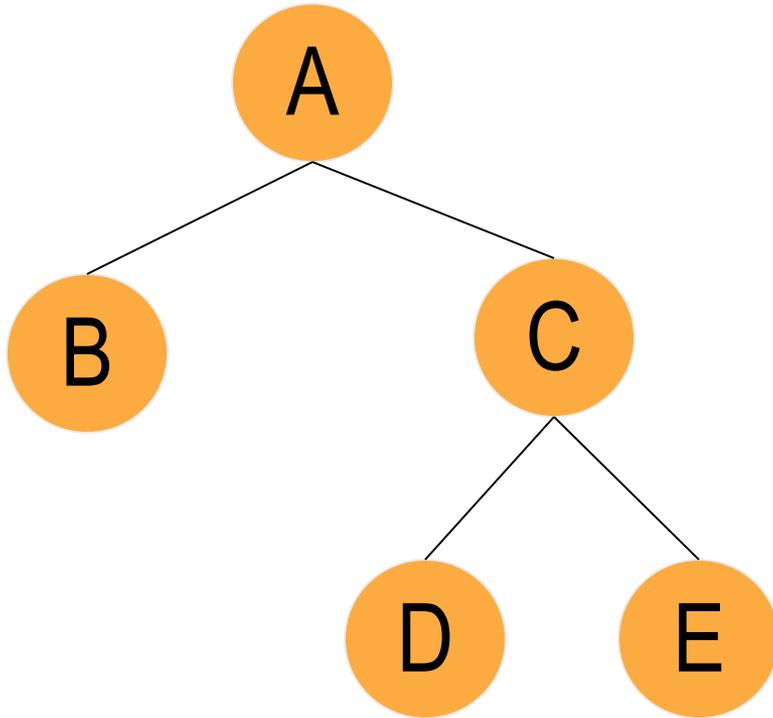
- **Árvore Binária Balanceada**
 - para cada nó, **as alturas** de suas duas sub-árvores **diferem de, no máximo, 1**



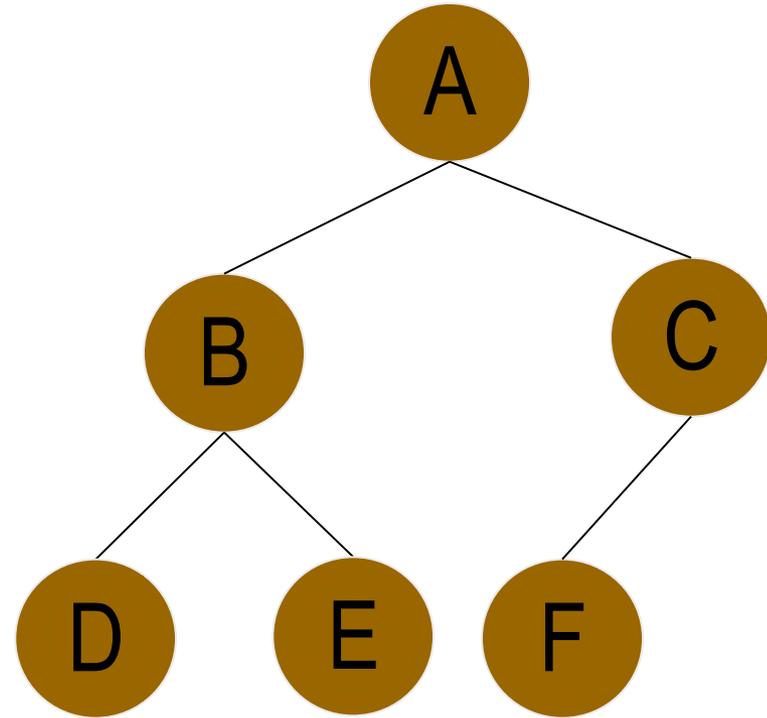
Árvore Binária Perfeitamente Balanceada

- **Árvore Binária Perfeitamente Balanceada:** para cada nó, o número de nós de suas sub-árvores esquerda e direita difere em, no máximo, 1
- Toda AB Perfeitamente Balanceada é Balanceada, mas o inverso não é necessariamente verdade.
- Uma AB com N nós tem altura mínima se e só se for Perfeitamente Balanceada.

Exemplo



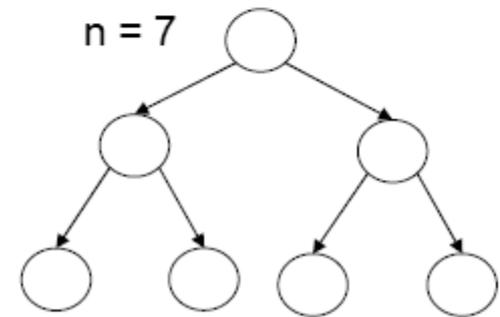
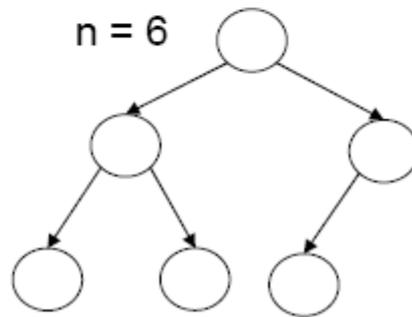
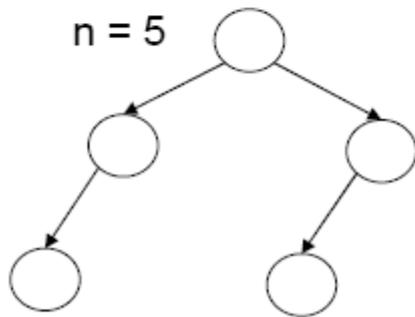
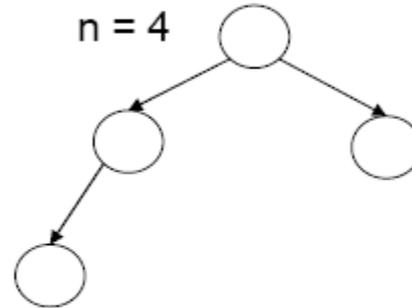
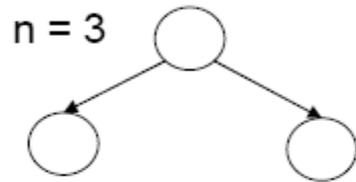
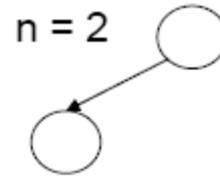
Árvore Balanceada



Árvore Perfeitamente Balanceada

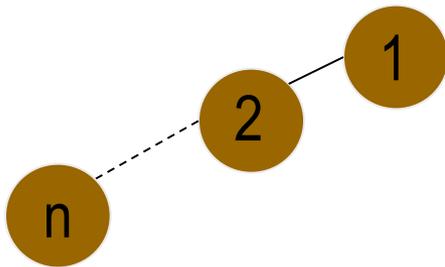
6 nós: $h_{\min} = 3$

Árvores Perfeitamente Balanceadas de Grau 2



Questões

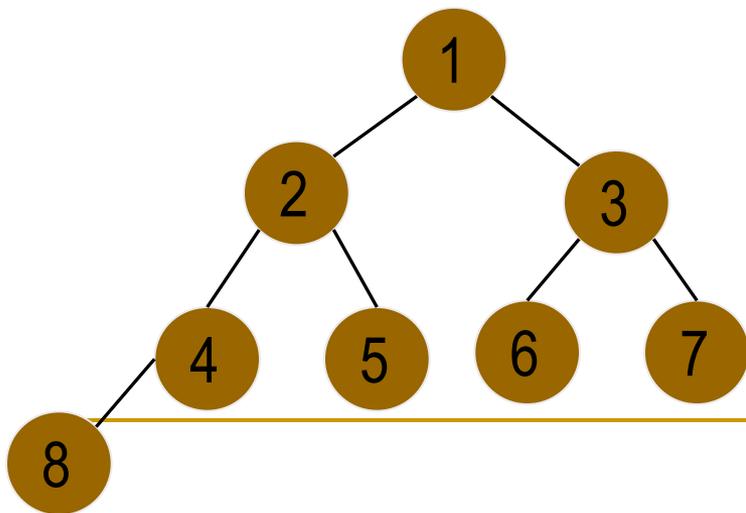
- Qual a **altura máxima** de uma AB com **n** nós?
 - Resposta: n
 - Árvore degenerada \equiv Lista



Há outra forma
para esta
árvore?

Questões

- Qual a **altura mínima** de uma AB c/ **n** nós?
 - Resposta: a mesma de uma AB Perfeitamente Balanceada com **N** nós



N=1; h=1

N=2,3; h=2

N=4..7; h=3

N=8..15; h=4

$h_{\min} = \lfloor \log_2 N \rfloor + 1$
(maior inteiro $\leq \log_2 N$) + 1

ou

$h_{\min} = \lceil \log_2 (N+1) \rceil$
menor inteiro $\geq \log_2 (N+1)$

Árvores de Altura Mínima

h	n
0	1
1	3
2	7
3	15
4	31
5	63
6	127
7	255
8	511
9	1023
10	2047

n	h
1	0
2	1
3	1
4	2
5	2
6	2
7	2
8	3
9	3
10	3
11	3
12	3
13	3
14	3
15	3
16	4

Questões

- Uma árvore binária completa é uma árvore estritamente binária?
- Uma árvore estritamente binária é uma árvore binária completa?

Implicações dos conceitos de balanceada e perfeitamente balanceada

- Estas árvores permitem que a recuperação de informação possa ser realizada de maneira a garantir a **altura da ordem de $O(\log_2 n)$** e
 - assim a eficiência em termos de tempo.

Representação/Implementação

■ Estática Seqüencial

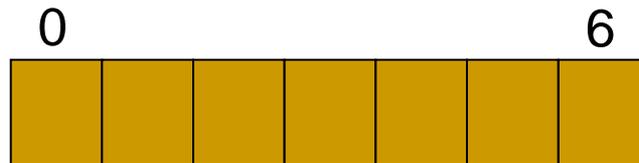
- Simples,
- Melhor solução quando é completa (cheia). Se não for completa (cheia), um campo “usado” é necessário para indicar qual posição tem ou não elemento.

■ Encadeada Dinâmica

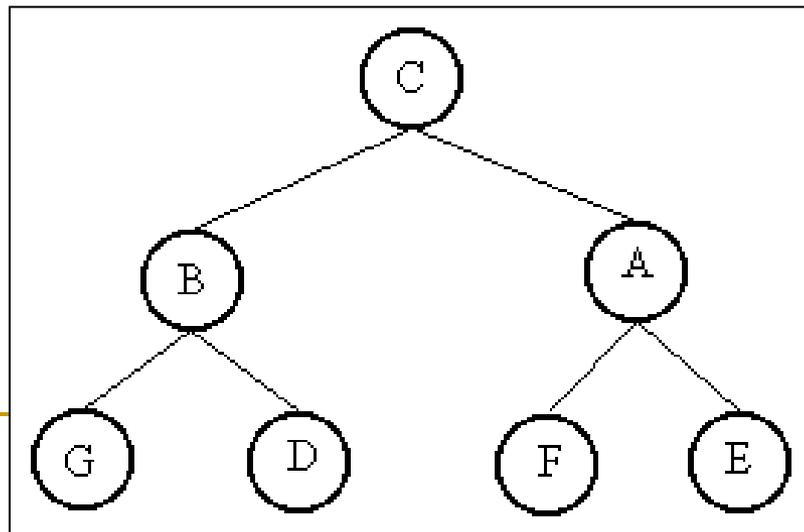
- O limite para o número de nós é a memória
- Preferida no caso geral

Árvores binárias

- Representação estática e seqüencial de árvores binárias
 - Vetor

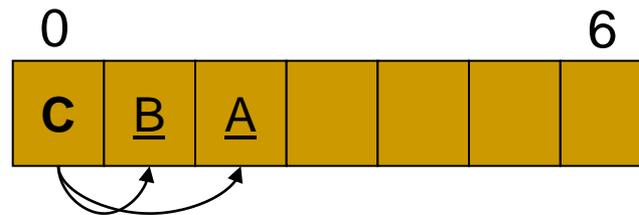


- Como colocar a árvore abaixo nesse vetor?

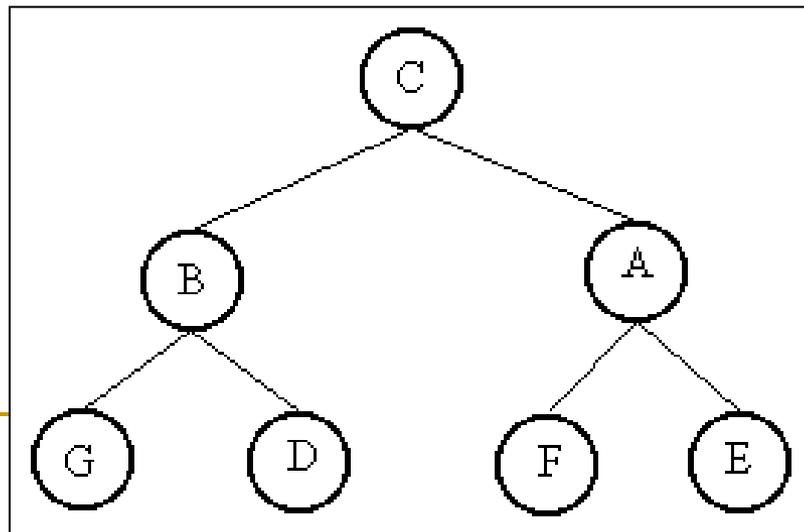


Árvores binárias

- Representação estática e seqüencial de árvores binárias
 - Vetor

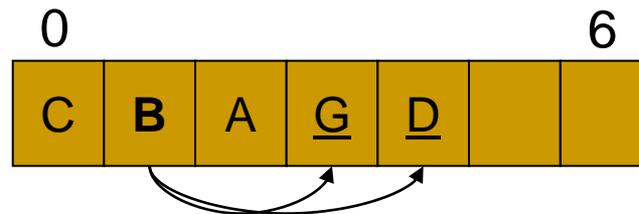


- Como colocar a árvore abaixo nesse vetor?

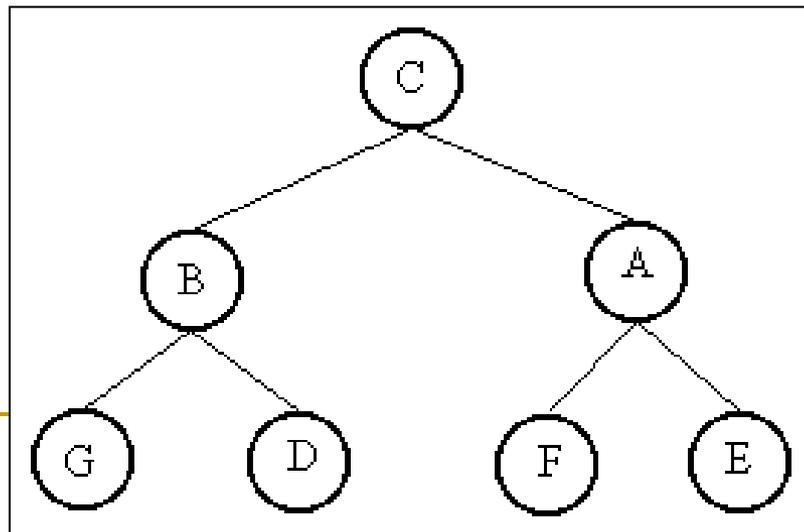


Árvores binárias

- Representação estática e seqüencial de árvores binárias
 - Vetor

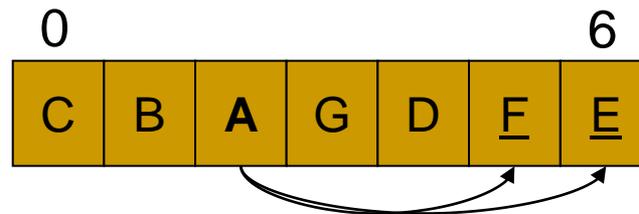


- Como colocar a árvore abaixo nesse vetor?

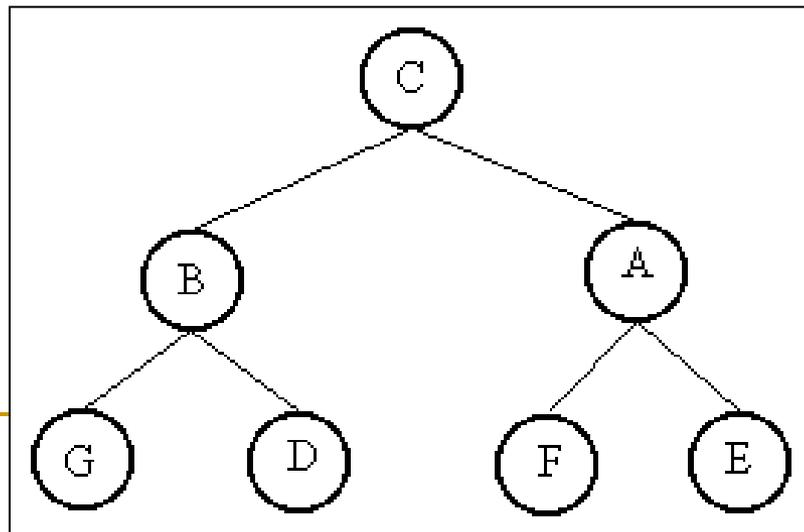


Árvores binárias

- Representação estática e seqüencial de árvores binárias
 - Vetor

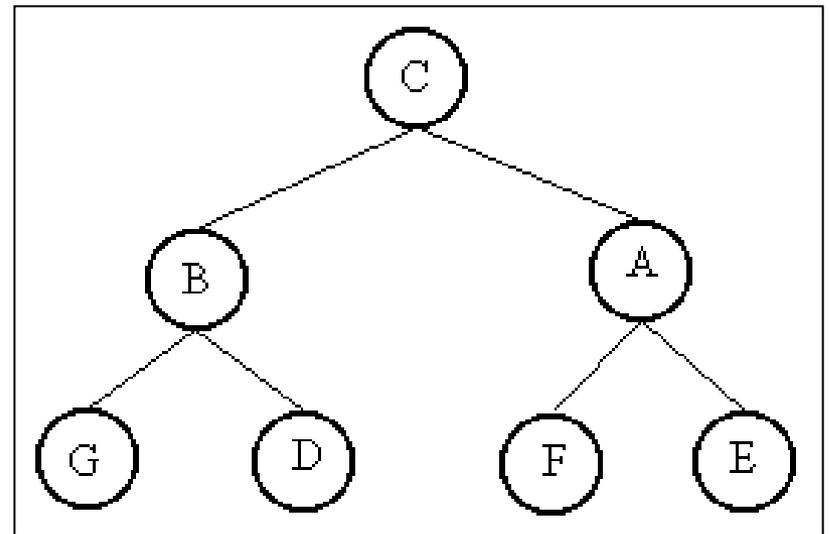
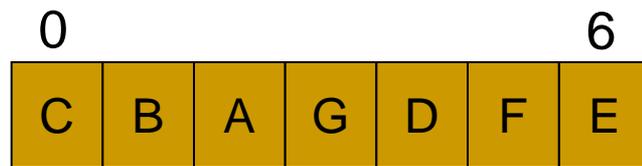


- Como colocar a árvore abaixo nesse vetor?



Árvores binárias

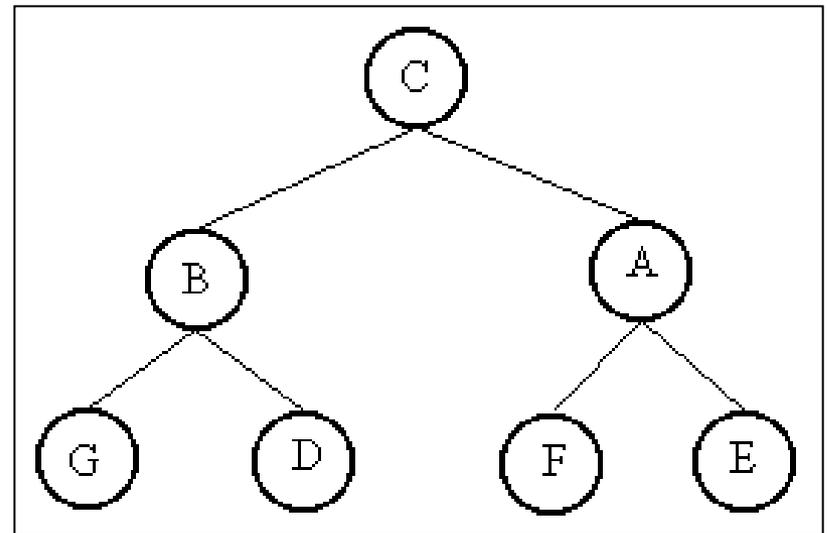
- Representação estática e seqüencial de árvores binárias



- Como saber quem é filho de quem?

Árvores binárias

- Representação estática e seqüencial de árvores binárias



- Como saber quem é filho de quem?
 - Filhos de i são $2i+1$ e $2i+2$

AB Completa (Cheia) (estática, seqüencial)

- Vantagem: espaço só p/ armazenar conteúdo; ligações implícitas
- Desvantagem: espaços vagos se árvore não é completa por níveis, ou se sofrer eliminação

ED AB, seqüencial estática

```
#define NUMNODES 500  
typedef int elem;
```

```
struct arv {  
    elem info;  
    int used;  
}
```

```
typedef struct arv Arv[NUMNODES];
```

Criar uma árvore com nó raiz

```
void criar(elem x, Arv t[ ])
{
  Int p;
  t[0].info = x;
  t[0].used = 1;
  For (p=1; p<NUMNODES;p++)
    t[p].used = 0;
}
// para criar a árvore vazia é só setar FALSE no
// campo used de 0.. NUMNODES
```

ED AB, encadeada dinâmica

Para qualquer árvore, cada nó é do tipo



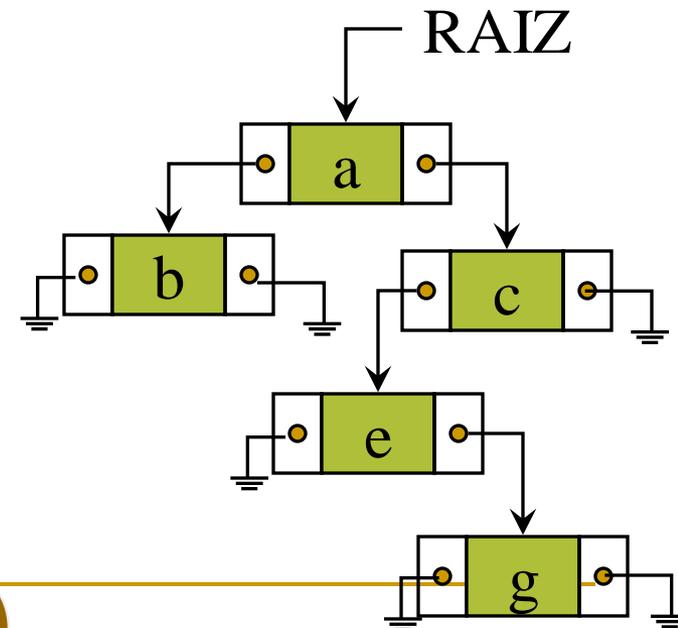
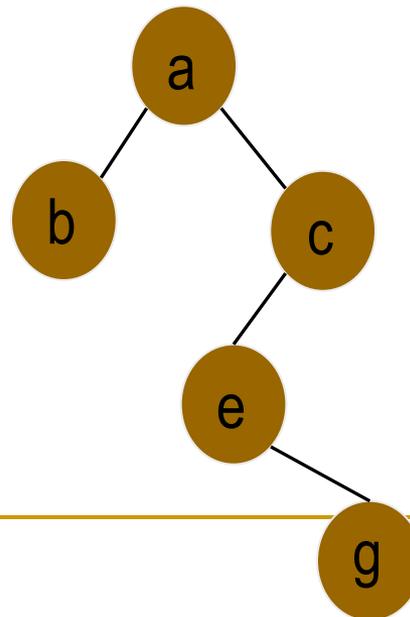
```
typedef int elem;
```

```
typedef struct arv *Arv;
```

```
struct arv {  
    elem info;  
    struct arv* esq;  
    struct arv* dir;  
};
```

AB.h

AB.c



Árvore binária

■ Operações do TAD

1. Criação de uma árvore

Dois casos: seguindo a definição recursiva

1. Criar uma árvore vazia
2. Criar uma árvore com 2 subárvores (e,d): **faz o papel de inserir nó**

2. Verificar se árvore está vazia, mostrar o conteúdo do nó

3. Buscar o pai de um nó, buscar um nó

4. Retornar: nro de nós, nro de folhas, altura

5. Destruir árvore

6. Remover um nó

Dois casos:

1. O nó não tem filhos
2. O nó tem um filho a esq, ou a dir ou os dois

7. Verificar nível de um nó

8. Verificar se é balanceada, se é perfeitamente balanceada

□ Outras?

Operações com resolução similar: usam percursos

■ Operações do TAD

1. Criação de uma árvore

Dois casos: seguindo a definição recursiva

1. Criar uma árvore vazia

2. Criar uma árvore com 2 subárvores (e,d): **faz o papel de inserir nó**

2. Verificar se árvore está vazia, mostrar o conteúdo do nó

3. **Buscar o pai de um nó, buscar um nó**

4. Retornar: nro de nós, nro de folhas, altura

5. **Destruir árvore**

6. Remover um nó

Dois casos:

1. O nó não tem filhos

2. O nó tem um filho a esq, ou a dir ou os dois

7. Verificar nível de um nó

8. Verificar se é balanceada, se é perfeitamente balanceada

□ **Imprime**

Interface para as funções do TAD

- `typedef int elem;`
`// Tipo exportado`
- `typedef struct arv *Arv;`
- `// Cria uma árvore vazia`
`Arv arv_criavazia (void);`
- `// Cria uma árvore com uma raiz e duas subárvores`
`Arv arv_cria (elem c, Arv e, Arv d);`
- `//Libera toda a memória usada na árvore a`
`Arv arv_destroi (Arv a);`
- `// Imprime a lista a`
`void arv_imprime (Arv a);`

- // Verifica se a árvore a é vazia
int arv_vazia (Arv a);
- // Retorna o conteúdo de um nó x na árvore a
elem arv_mostra_no(Arv a, int *erro);
- //Retorna (busca) o endereço de c na árvore a. Se c não está,
retorna NULL
Arv arv_busca (Arv a, elem c);
- // Retorna (busca) o endereço do pai de x na árvore a
Arv arv_busca_pai(Arv a, elem x);
- //Remove um elemento da árvore, caso ele esteja nela. Retorna
erro = 0 se sucesso e erro = 1 se não encontra
void arv_remove(Arv a, elem x, int *erro);

- //Retorna a altura de uma árvore binária a. Considera o nível da raiz = 1
int arv_altura(Arv a);
- // Retorna o número de nós na árvore a
int arv_numero_nos(Arv a);
- //Retorna o número de folhas da árvore a
int arv_nro_folhas(Arv a);
- //função de percurso de pre-ordem na árvore = busca em profundidade
void arv_percurso_pre_ordem(Arv a);
- //função de percurso de em-ordem na árvore
void arv_percurso_em_ordem(Arv a);
- //função de percurso de pos-ordem na árvore
void arv_percurso_pos_ordem(Arv a);

AB - Percursos

- **Objetivo:** Percorrer uma AB 'visitando' cada nó uma única vez.
- Um percurso gera uma seqüência linear de nós, e podemos então falar de nó **predecessor** ou **sucessor** de um nó, segundo um dado percurso.
 - Não existe um percurso único para árvores (binárias ou não): diferentes percursos podem ser realizados, **dependendo da aplicação.**
- **Utilização:** imprimir uma árvore, atualizar um campo de cada nó, buscar um item, destruir uma árvore, etc.

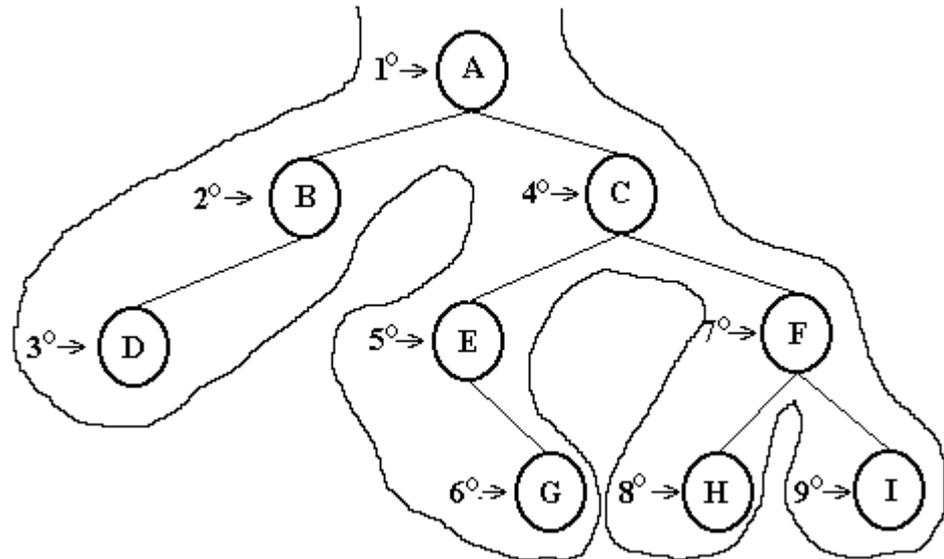
AB – Percursos em Árvores

- 3 percursos básicos para AB's:
 - pré-ordem (Pre-order)
 - in-ordem (In-order)
 - pós-ordem (Post-order)
- A diferença entre eles está, basicamente, na ordem em que cada nó é alcançado pelo percurso
 - “Visitar” um nó pode ser:
 - Mostrar (imprimir) o seu valor;
 - Modificar o valor do nó;
 - Verificar se o valor pertence a árvore (membro)

Pré-ordem

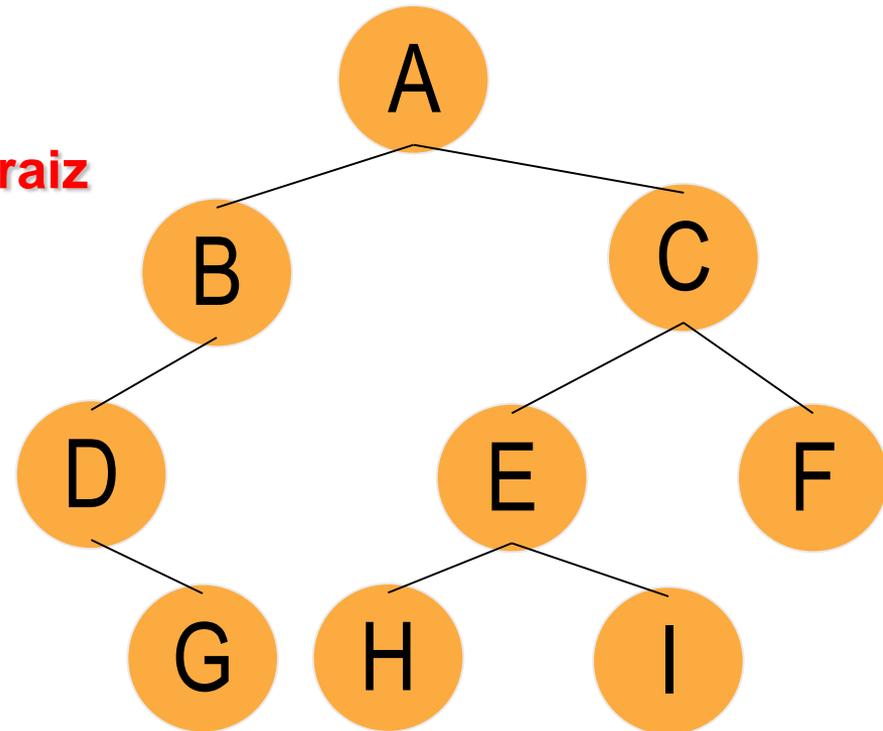
se árvore vazia; fim

1. visitar o nó raiz
2. percorrer em pré-ordem a subárvore esquerda
3. percorrer em pré-ordem a subárvore direita



AB - Percurso Pré-Ordem

```
void arv_percurso_pre_ordem(Arv a) {  
    if (!arv_vazia(a)) {  
        printf("%d\n",a->info); // processa raiz  
        arv_percurso_pre_ordem(a->esq);  
        arv_percurso_pre_ordem(a->dir);  
    }  
}
```



Resultado: ABDGCEHIF

E se invertermos as chamadas recursivas?

Trocar

```
pre_ordem(a->esq) ;
```

```
pre_ordem(a->dir) ;
```

POR:

```
pre_ordem(a->dir) ;
```

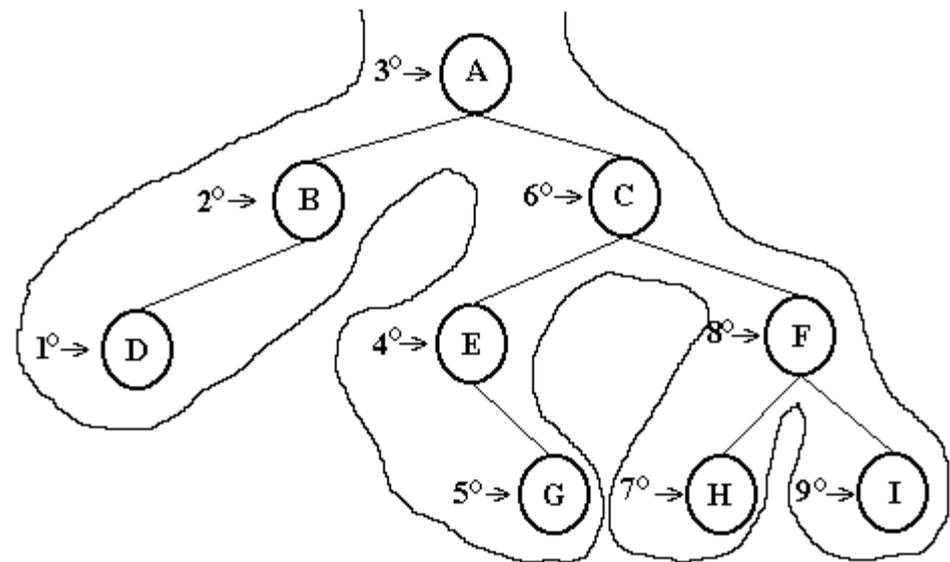
```
pre_ordem(a->esq) ;
```

Ainda é pré-ordem??

In-Ordem

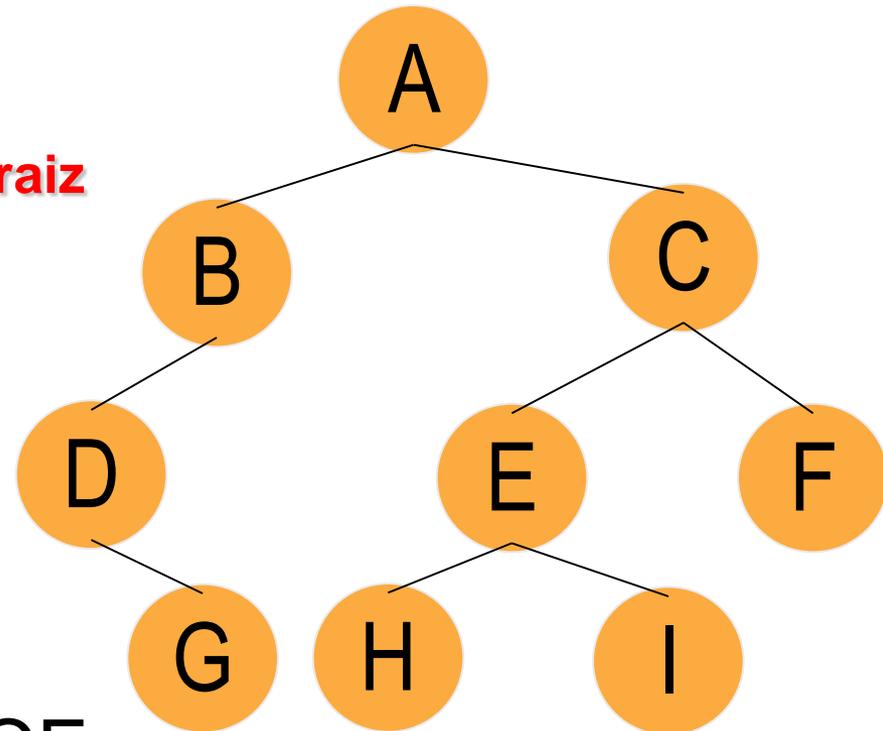
se árvore vazia, fim

1. percorrer em in-ordem a subárvore esquerda
2. visitar o nó raiz
3. percorrer em in-ordem a subárvore direita



AB - Percurso In-Ordem

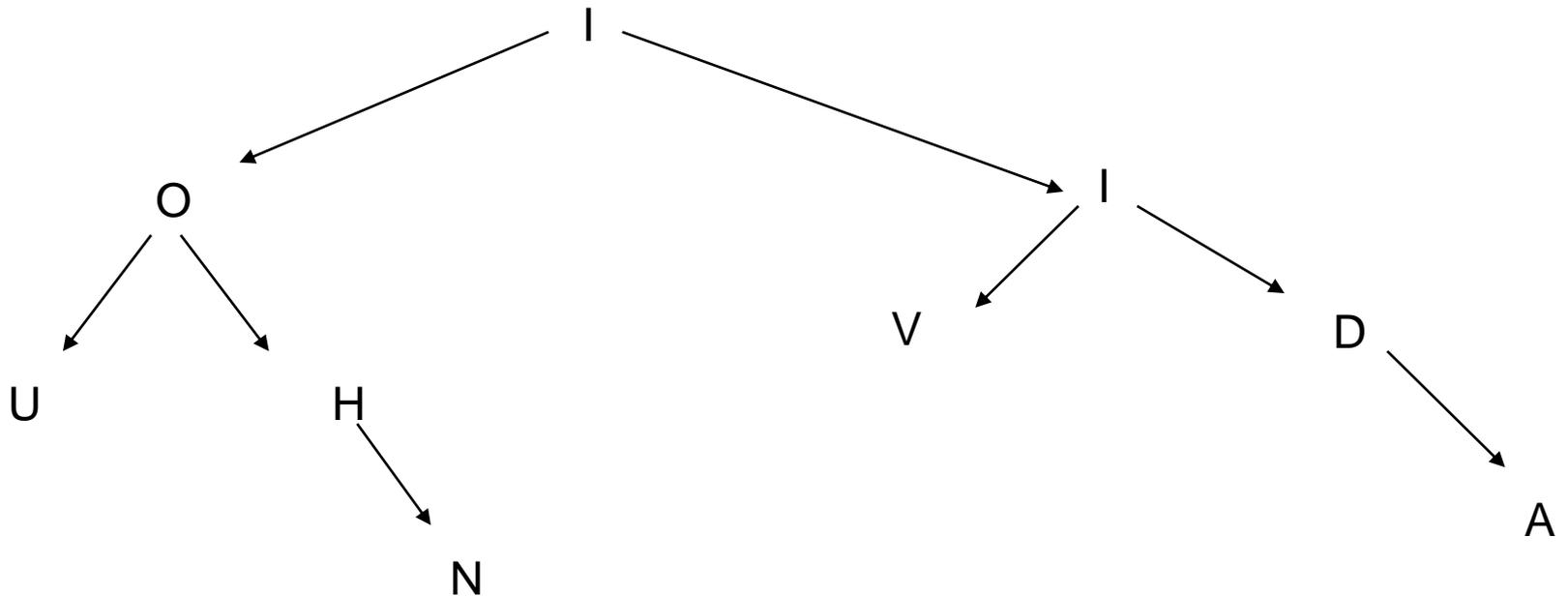
```
void arv_percurso_em_ordem(Arv a) {  
    if (!arv_vazia(a)) {  
        arv_percurso_em_ordem(a->esq);  
        printf("%d\n",a->info); // processa raiz  
        arv_percurso_em_ordem(a->dir);  
    }  
}
```



Resultado: DGBAHEICF

E se invertermos as chamadas recursivas?

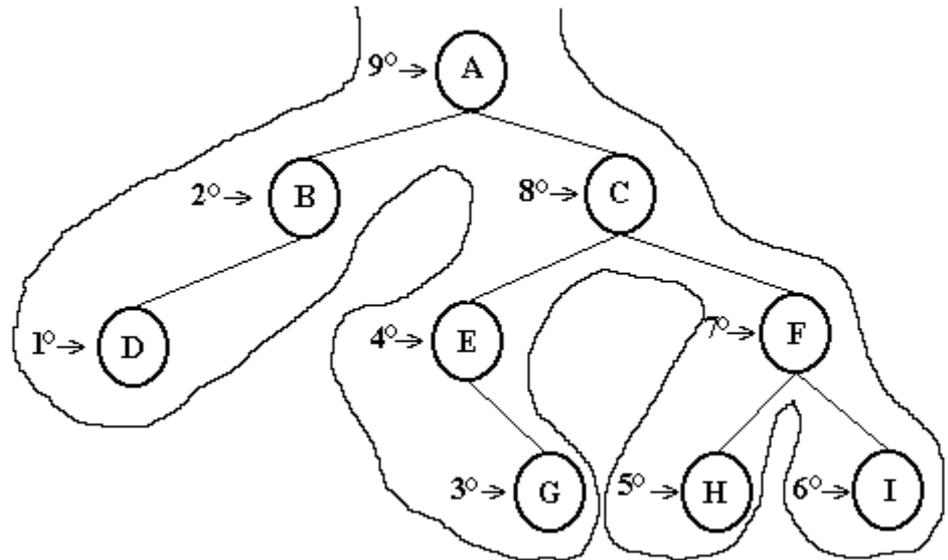
- Ainda é in-ordem? Qual o resultado para a árvore abaixo?



Pós-ordem

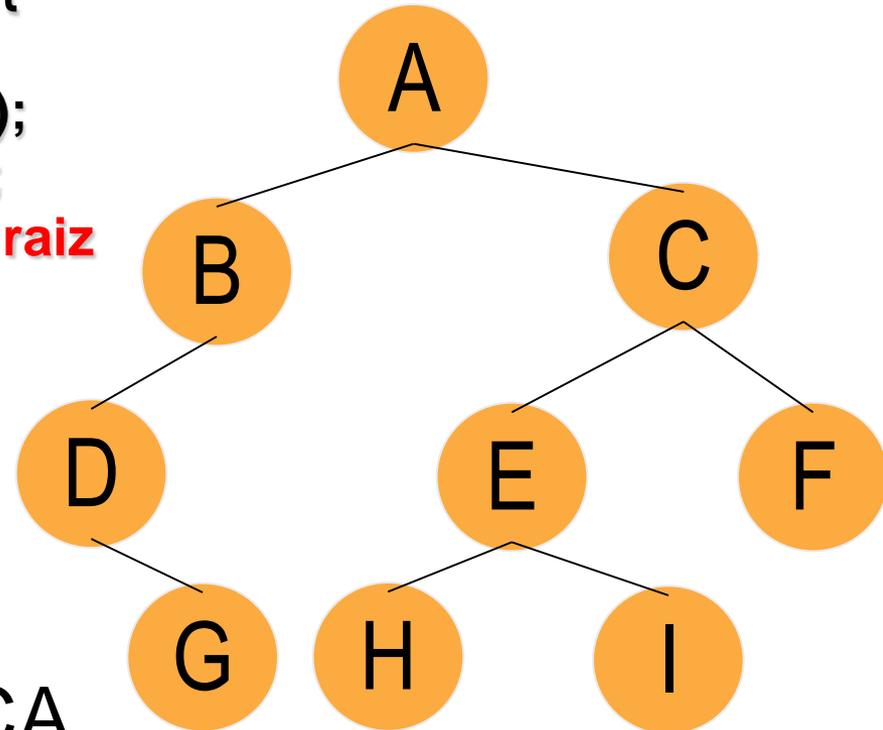
se árvore vazia, fim

1. percorrer em Pós-Ordem a subárvore esquerda
2. percorrer em Pós-Ordem a subárvore direita
3. visitar o nó raiz



AB - Percurso Pós-Ordem

```
void arv_percurso_pos_ordem(Arv a) {  
    if (!arv_vazia(a)) {  
        arv_percurso_pos_ordem(a->esq);  
        arv_percurso_pos_ordem(a->dir);  
        printf("%d\n",a->info); //processa raiz  
    }  
}
```



Resultado: GDBHIEFCA

E se invertermos as chamadas recursivas?

- Ainda é pós-ordem?

-
- Procedimento recursivo p/ destruir árvore, liberando o espaço alocado
 - Usar percurso em pós-ordem

 - Arv arv_destroi (Arv a)
-

Procedimento recursivo p/ destruir árvore, liberando o espaço alocado (percurso em pós-ordem)

Porque não usamos pré ou in-ordem para esta tarefa???

```
Arv arv_destroi (Arv a) {  
  if (!arv_vazia(a)) {  
    arv_destroi(a->esq); /* libera sae */  
    arv_destroi(a->dir); /* libera sad */  
    free(a);           /* libera raiz */  
  }  
  return NULL;  
}
```