

---

# Árvores Binárias

---

16/11

Representação e Implementação: Encadeada  
Dinâmica  
O TAD

# ED AB, encadeada dinâmica

Para qualquer árvore, cada nó é do tipo



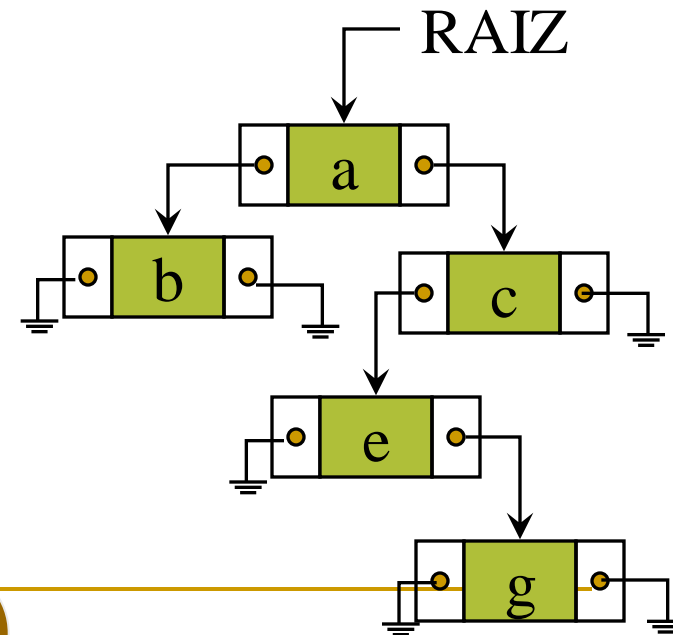
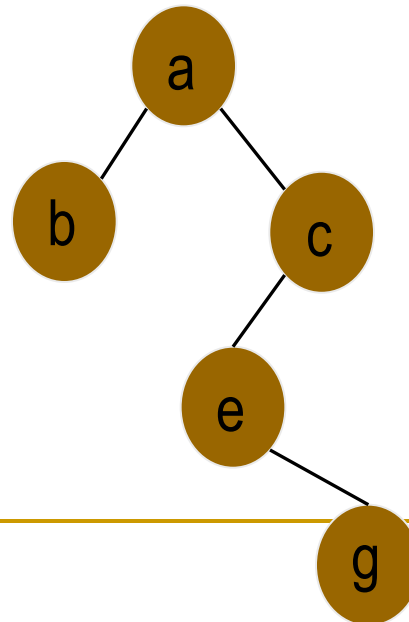
```
typedef int elem;
```

```
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.



---

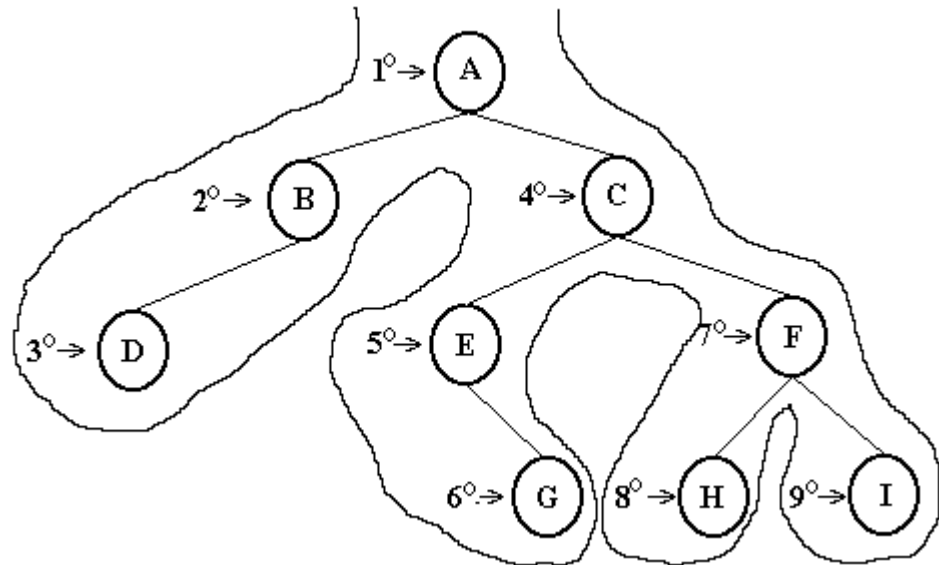
# AB – Percursos em Árvores

- 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

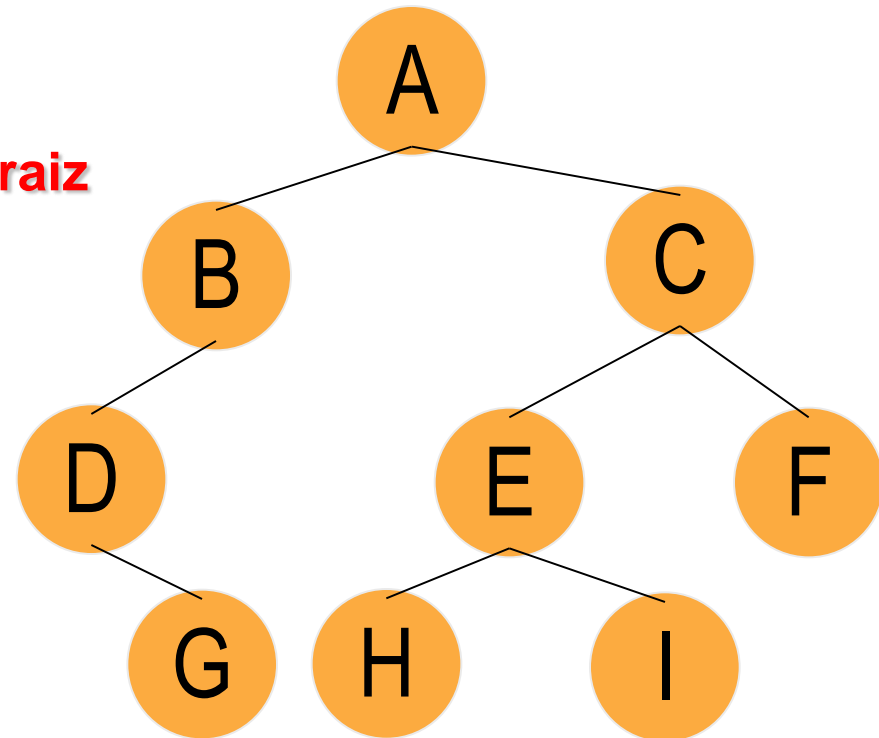
se árvore vazia; fim

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) ;
```

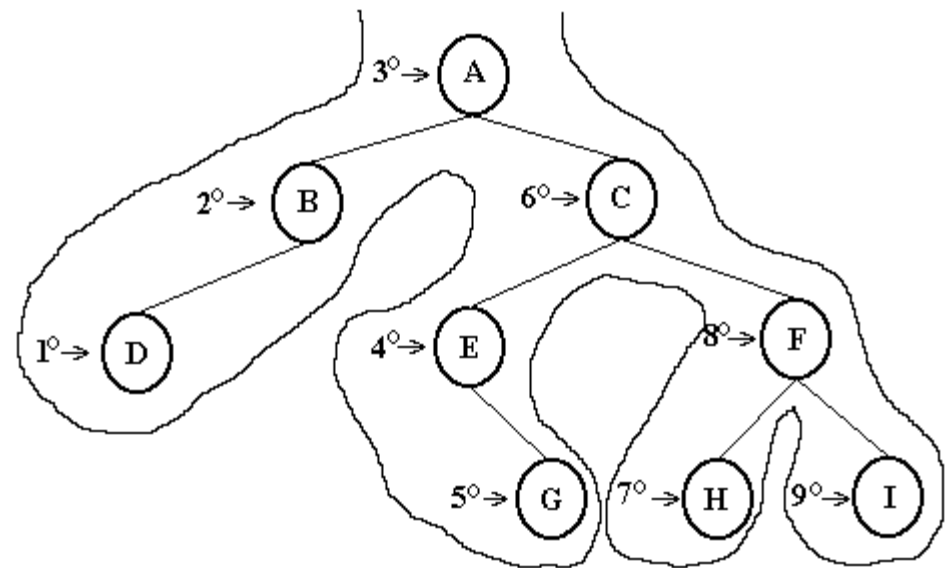
```
pre_ordem(a->esq) ;
```

Ainda é pré-ordem??

# In-Ordem

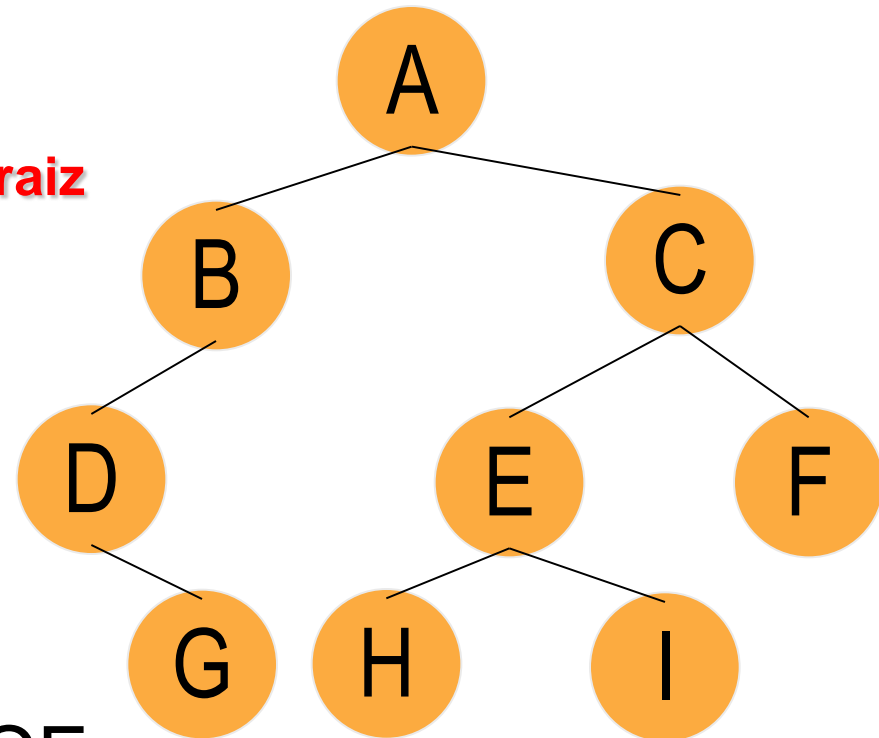
se árvore vazia, fim

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

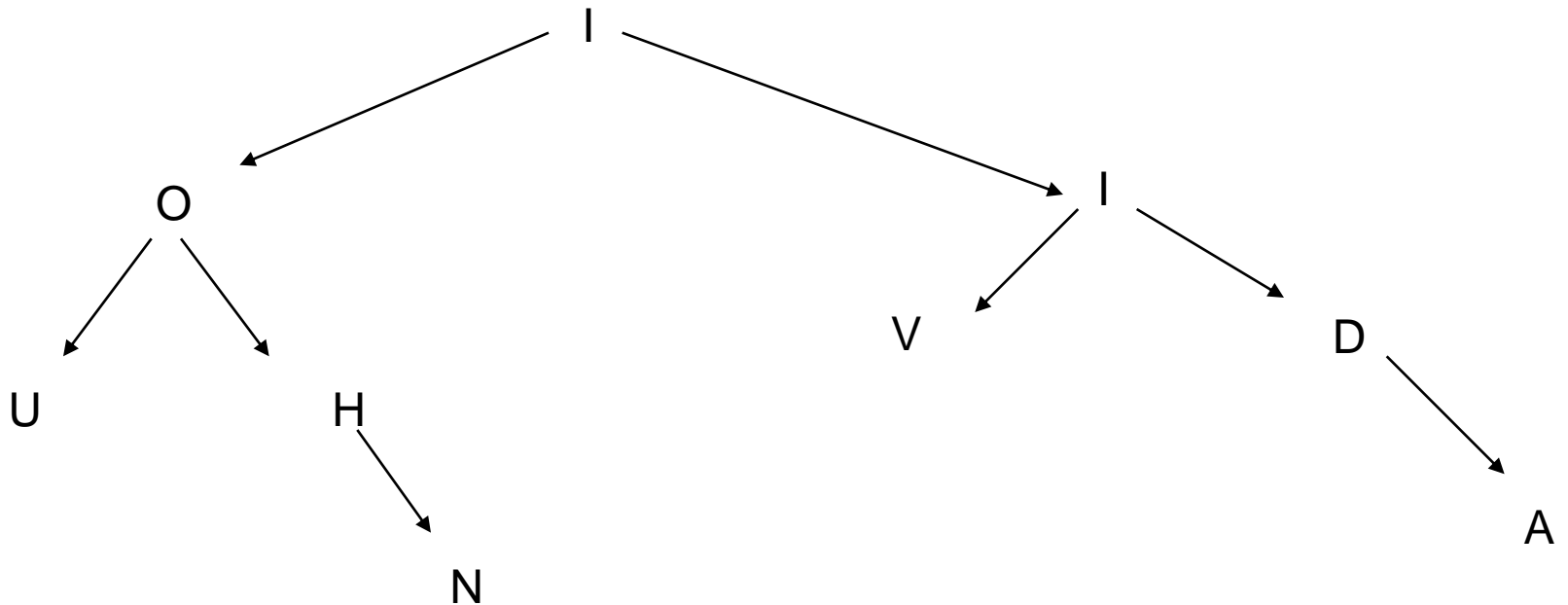
```
void arv_percurso_em_ordem(Arv a) {  
    if (!arv_vazia(a)) {  
        arv_percurso_em_ordem(a->esq);  
        printf("%d\n",a->info); // processa raiz  
        arv_percurso_em_ordem(a->dir);  
    }  
}
```



Resultado: DGBAHEICF

# E se invertermos as chamadas recursivas?

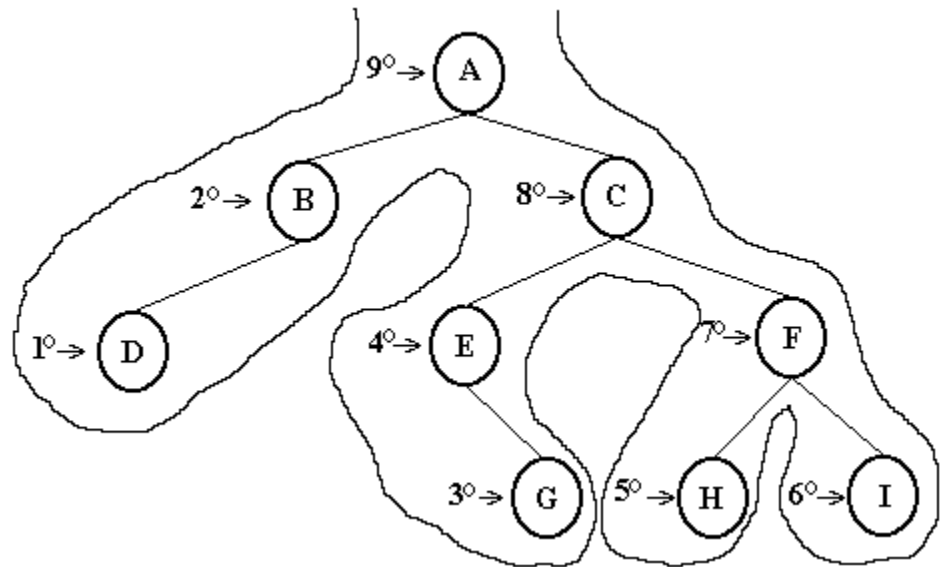
- Ainda é in-ordem? Qual o resultado para a árvore abaixo?



# Pós-ordem

se árvore vazia, fim

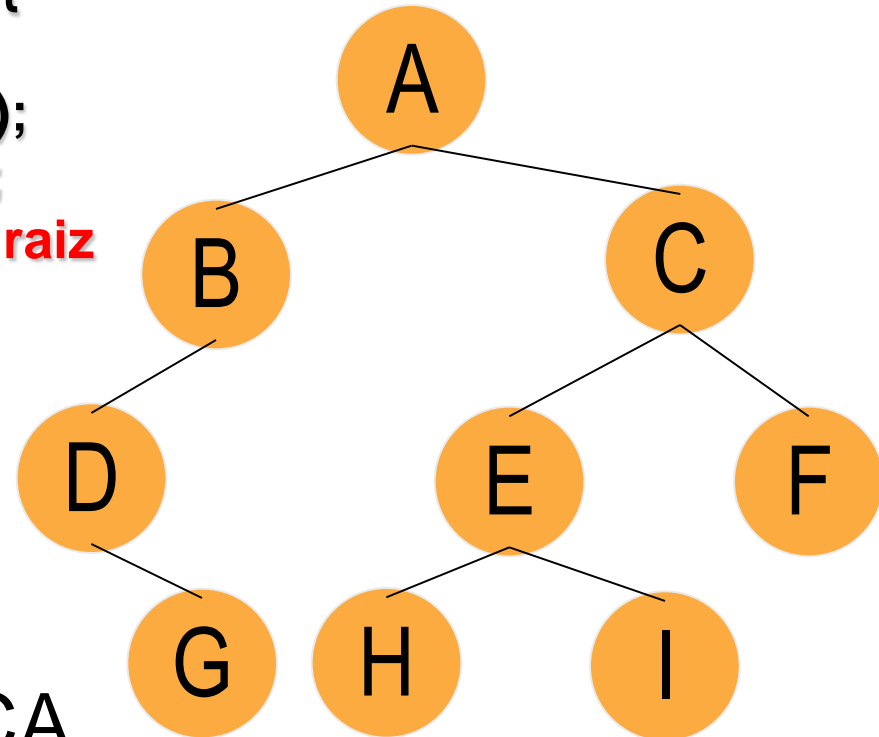
1. percorrer em Pós-Ordem a subárvore esquerda
2. percorrer em Pós-Ordem a subárvore direita
3. visitar o nó raiz





# 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;
}
```

---

# Função para verificar se valor está na árvore

//Retorna (busca) o endereço de c na árvore a. Se c não está retorna NULL

Arv arv\_busca (Arv a, elem c)

# Função para verificar se valor está na árvore

```
Arv arv_busca (Arv a, elem c) {
Arv aux;

if (arv_vazia(a))
    return NULL; /* árvore vazia: não encontrou */
else if (a->info==c)
    return a;
else {
    aux = arv_busca(a->esq,c); /* busca na sae */
    if (aux == NULL)
        aux = arv_busca(a->dir,c); /* busca na sad */
    return aux;
}
}
```

# Função para retornar o endereço do pai de um nó com elemento x

- // Retorna (busca) o endereço do pai de x na árvore a. Retorna NULL se for nula, tiver só raiz ou não achou o elemento x
- Casos:
  1. Se a árvore é nula ou só tem raiz devolve NULL
  2. Se tem filho à esquerda e é x devolve raiz
  3. Se tem filho à direita e é x devolve raiz
  4. Senão chama recursivamente à esquerda e se não achou chama recursivamente à direita
- `Arv arv_busca_pai(Arv a, elem x);`

```
Arv arv_busca_pai(Arv a, elem x) {  
    Arv aux;
```

```
    if (arv_vazia(a)) // árvore vazia
```

```
        return(NULL);
```

```
    else if (a->esq == NULL && a->dir == NULL) // só tem raiz
```

```
        return(NULL);
```

```
    else if ((a->esq!=NULL) && (a->esq->info==x))
```

```
        return(a);
```

```
    else if ((a->dir!=NULL) && (a->dir->info==x))
```

```
        return(a);
```

```
    else {
```

```
        aux=arv_busca_pai(a->esq,x);
```

```
        if (aux==NULL)
```

```
            aux=arv_busca_pai(a->dir,x);
```

```
        return(aux);
```

```
    }
```

```
}
```



# TAD

```
Arv arv_criavazia (void)
```

```
{  
return NULL;  
}
```

```
Arv arv_cria (elem c, Arv sae, Arv sad)
```

```
{  
Arv p=(Arv)malloc(sizeof(struct arv));  
p->info = c;  
p->esq = sae;  
p->dir = sad;  
return p;  
}
```

---

```
int arv_vazia (Arv a)
{
return a==NULL;
}
```

---

# Exercício

- Mostra o conteúdo do nó
- Retornar o nro de nós
- Retornar o nro de folhas

---

# Mostra o conteúdo do nó

```
elem arv_mostra_no(Arv a, int *erro){  
    *erro = 0;  
    if (!arv_vazia(a))  
        return a->info;  
    else *erro = 1;  
}
```

# Função para retornar o número de nós

```
int arv_numero_nos(Arv a){  
    if (arv_vazia(a))  
        return 0;  
    return 1 + arv_numero_nos(a->esq) +  
           arv_numero_nos(a->dir);  
}
```

# Função para retornar o nro de folhas

```
int arv_nro_folhas(Arv a){
if (arv_vazia(a))
return 0;
else
    if ((a->esq == NULL) && (a->dir == NULL))
        return 1; // definição de nó folha
    else
        return arv_nro_folhas(a->esq) +
arv_nro_folhas(a->dir);
}
```

# Altura (considerando que o nível da raiz é 0)

- *pré-condição*: Árvore binária já tenha sido criada
- *pós-condição*: retorna a altura da árvore

A definição de altura de uma árvore nos leva aos seguintes casos:

- Caso base: a altura de uma árvore vazia é -1 (por definição a altura das folhas é 0; portanto parece natural adotar -1 como a altura de uma árvore vazia)
- Caso recursivo: a altura de uma árvore que contém um nó (não nulo) é determinada como sendo a maior altura entre as subárvores esquerda e direita deste nó adicionado a um (uma unidade a mais de altura devido ao próprio nó)

# Altura – considerando o nível da raiz = 0

```
int arv_altura(Arv a) {
    int alt_esq, alt_dir;
    /*altura = max(altE, altD) + 1*/

    if (arv_vazia(a))
        return -1;
    else {
        alt_esq = 1+ arv_altura(a->esq);
        alt_dir = 1+ arv_altura(a->dir);
        if (alt_esq > alt_dir)
            return(alt_esq);
        else return(alt_dir);
    }
}
```



# Altura – considerando o nível da raiz = 1

// segue as nossas definições

```
int arv_altura(Arv a) {
    int alt_esq, alt_dir;
    /* altura = max(altE, altD) + 1 */

    if (arv_vazia(a))
        return 0;
    else {
        alt_esq = 1+ arv_altura(a->esq);
        alt_dir = 1+ arv_altura(a->dir);
        if (alt_esq > alt_dir)
            return(alt_esq);
        else return(alt_dir);
    }
}
```

# Exercício: Remover um nó

//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);
```

Dois casos:

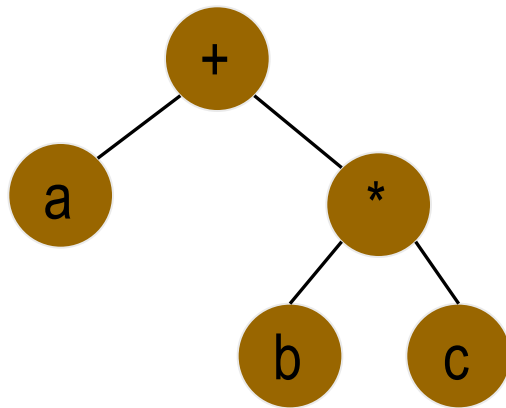
1. O nó não tem filhos
2. O nó tem um filho a esq, ou a dir ou os dois

# Passos

- 1) Localize o pai do nó  $p$  (que contém  $x$ )
  - ❑ // caso 1: só tem raiz e o  $x$  está nela
  - ❑ // caso 2: não tem pai: árvore nula ou  $x$  não está na árvore
  - ❑ // caso 3: o  $x$  está a esquerda do pai
  - ❑ // caso 4: o  $x$  está a direita do pai
- 2) O nó não tem filhos:
  - ❑ Remova o nó
- 3) O nó tem um filho a esq, ou a dir ou os dois:  
Troque o conteúdo de  $p$  com o do seu filho a esquerda/direita e chame novamente a função para trazer o  $x$  para as folhas

# AB – Percursos

- Percurso para expressões aritméticas



- Pré-ordem:  $+a*bc$
- In-ordem:  $a+(b*c)$
- Pós-ordem:  $abc*+$

- Em algoritmos iterativos utiliza-se uma pilha ou um campo a mais em cada nó para guardar o nó anterior (pai)

# Conclusões

- O processo de busca de informação em uma árvore binária como vimos até agora é caro.
  - Se as chaves não estão em uma ordem pré-estabelecida, toda a estrutura precisa ser percorrida para encontrar uma determinada chave (**no pior caso**), o que não seria eficiente
- Veremos uma forma de melhorar o tempo de busca, utilizando **Árvores Binárias de Busca**

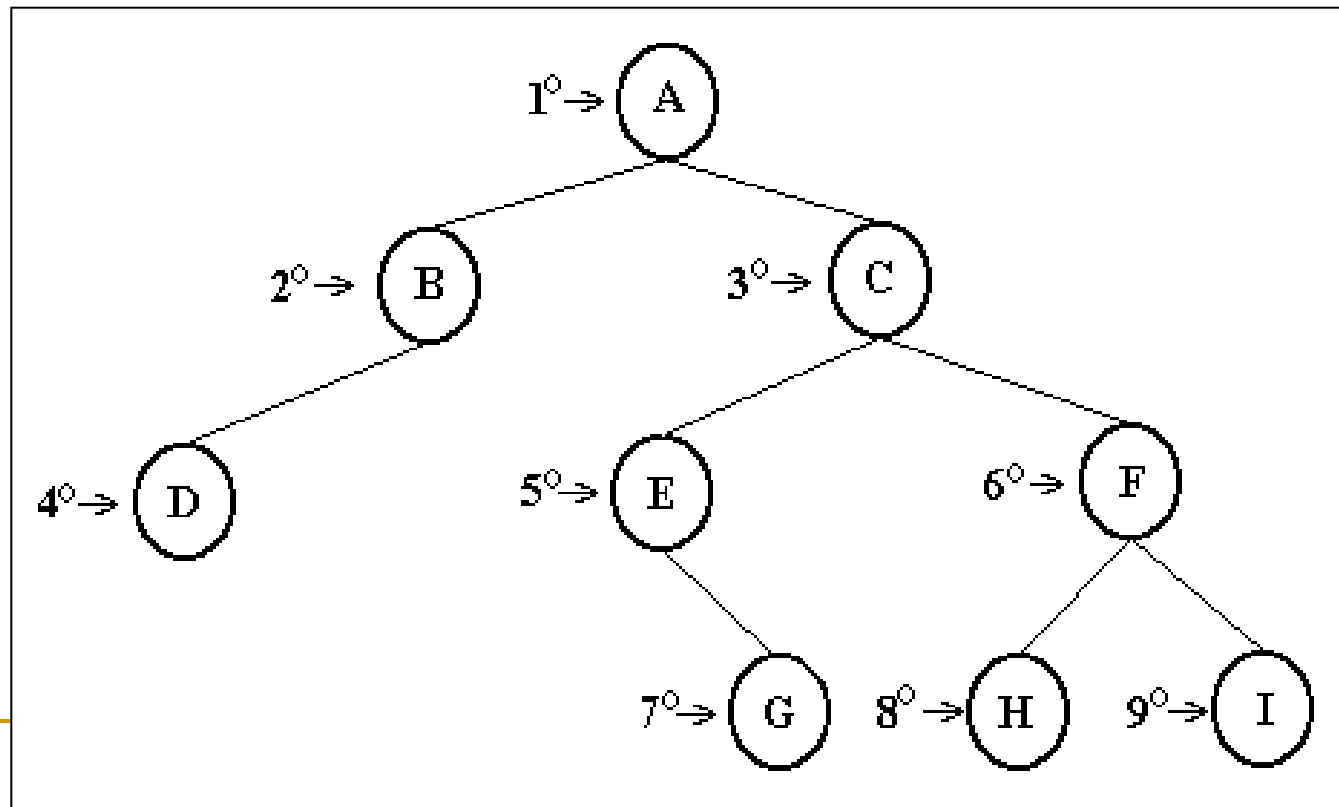
---

# Percurso em árvores

- Outras formas/nomenclaturas
    - Busca/percurso em largura
    - Busca/percurso em profundidade
  - Independente do tipo de árvore
  - Tradicionalmente da esquerda para a direita
-

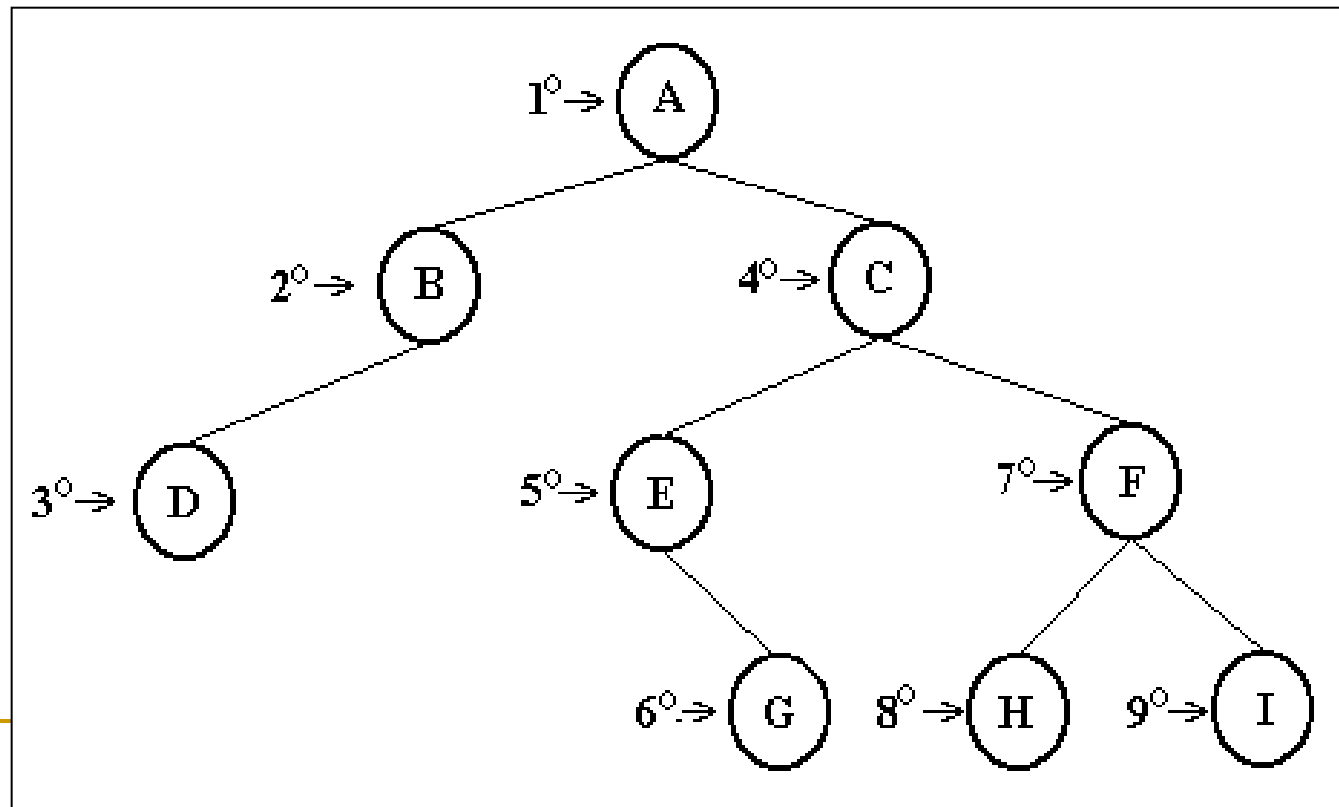
# Percurso em árvores

- Em largura
  - Um nível de cada vez



# Percurso em árvores

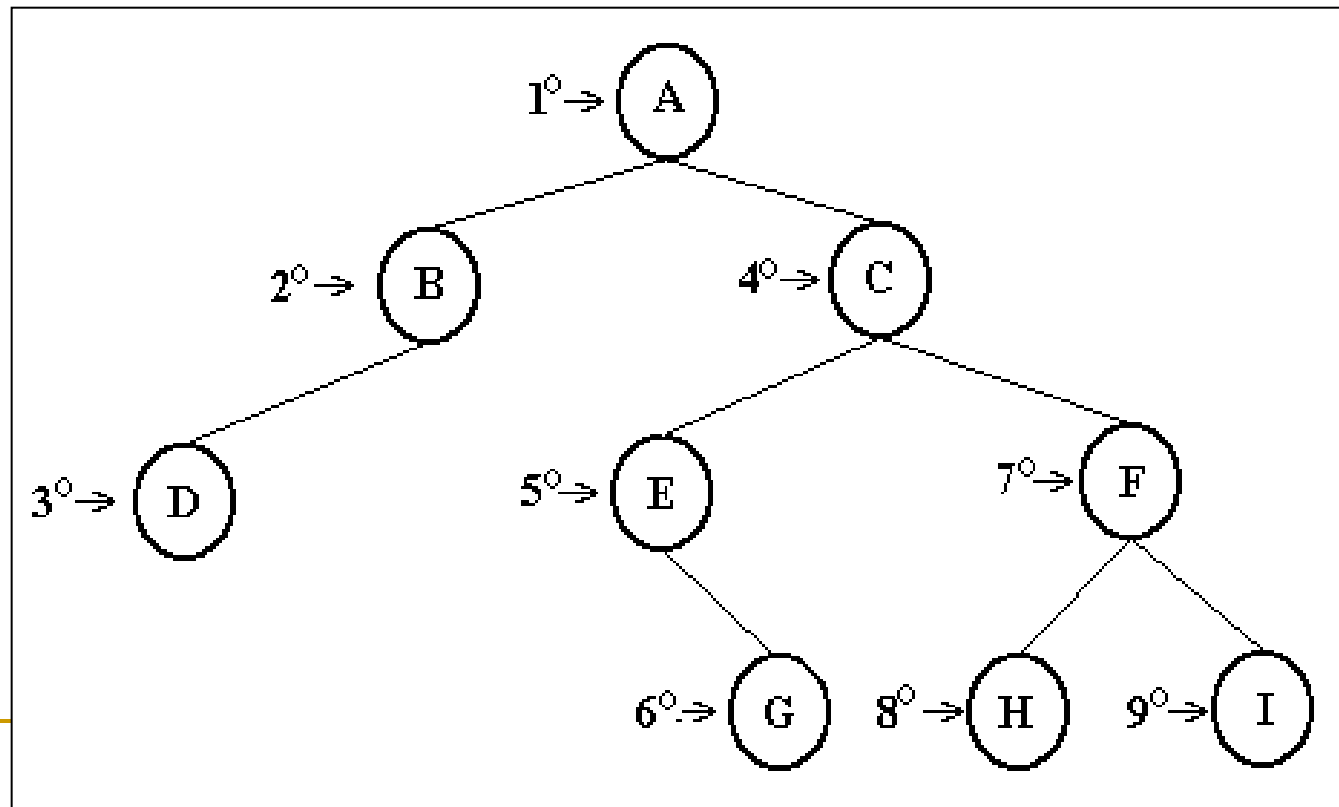
- Em profundidade
  - Um ramo da árvore de cada vez (= ???)





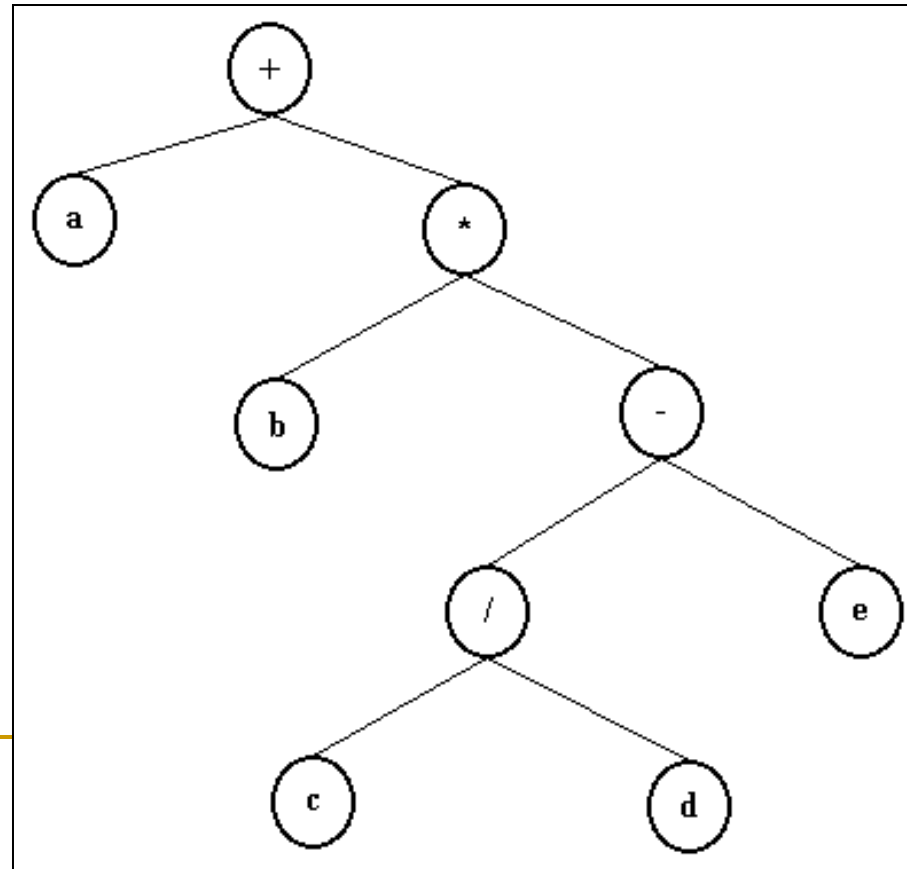
# Percurso em árvores

- Em profundidade
  - Um ramo da árvore de cada vez (= pré-ordem)



# Exercício: percurso em árvores

- Para a árvore abaixo, mostre quais seriam as saídas para os percursos em largura, profundidade (pré-ordem), em-ordem e pós-ordem



---

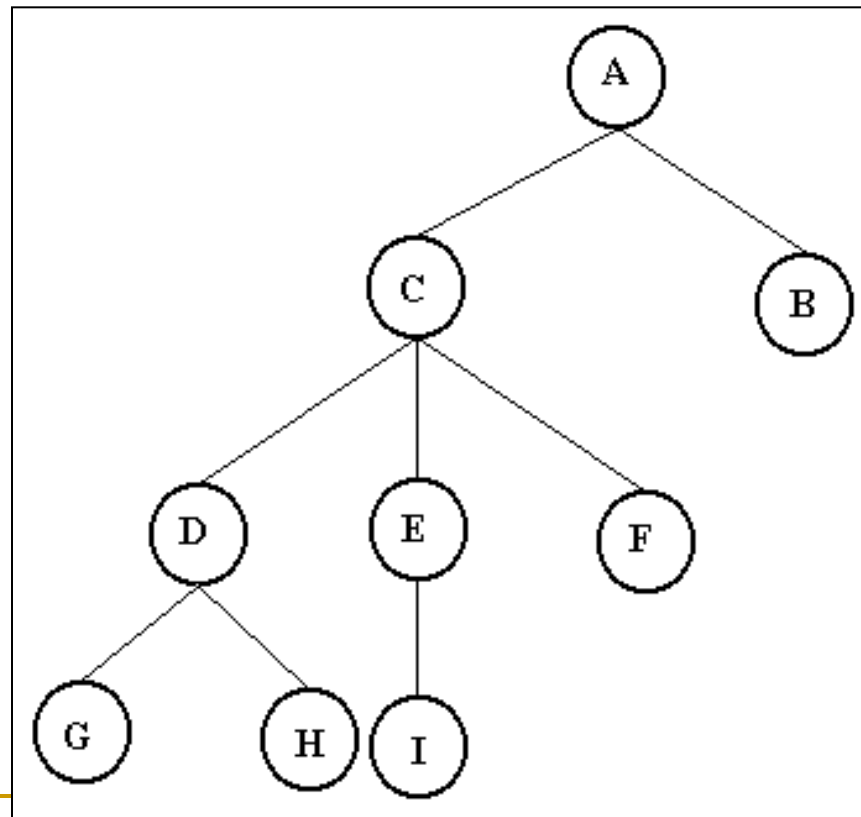
# Percurso em árvores

- Em **profundidade** = pré-ordem
  - Usa **pilha** (explícita ou implicitamente)



# Percurso em árvores

- Teste a estratégia com a pilha explícita



---

# Percurso em árvores

- Em largura
  - Como implementar?



---

# Percurso em árvores

- Em largura

- FILA

- Se o nó raiz for diferente de NULL, ele entra na fila

- Enquanto fila não vazia

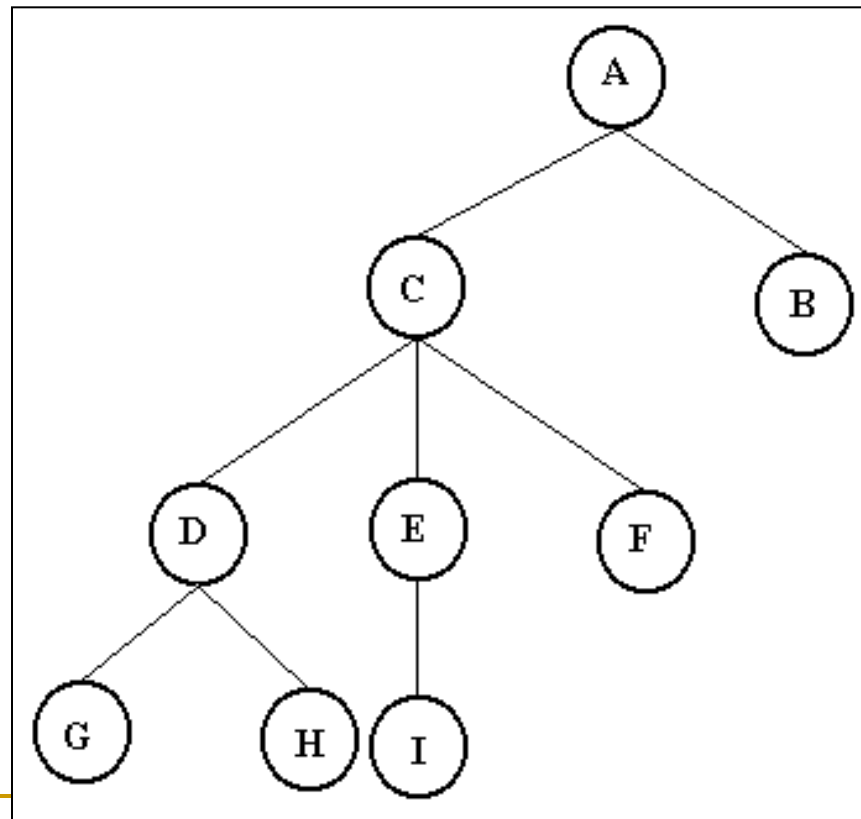
- Retira-se/visita-se o primeiro da fila

- Se houver filhos desse nó, eles entram na fila

---

# Percurso em árvores

- Teste a estratégia



---

# Largura vs. profundidade

- Ao buscar um elemento

- **Profundidade**

- Vantagens?
  - Desvantagens?
-



---

# Largura vs. profundidade

- Ao buscar um elemento
  - **Profundidade**
    - Menos memória para guardar nós não visitados
    - Pode buscar “para sempre”, demorando mais para achar o elemento



---

# Largura vs. profundidade

- Ao buscar um elemento

- **Largura**

- Vantagens?
  - Desvantagens?
-

---

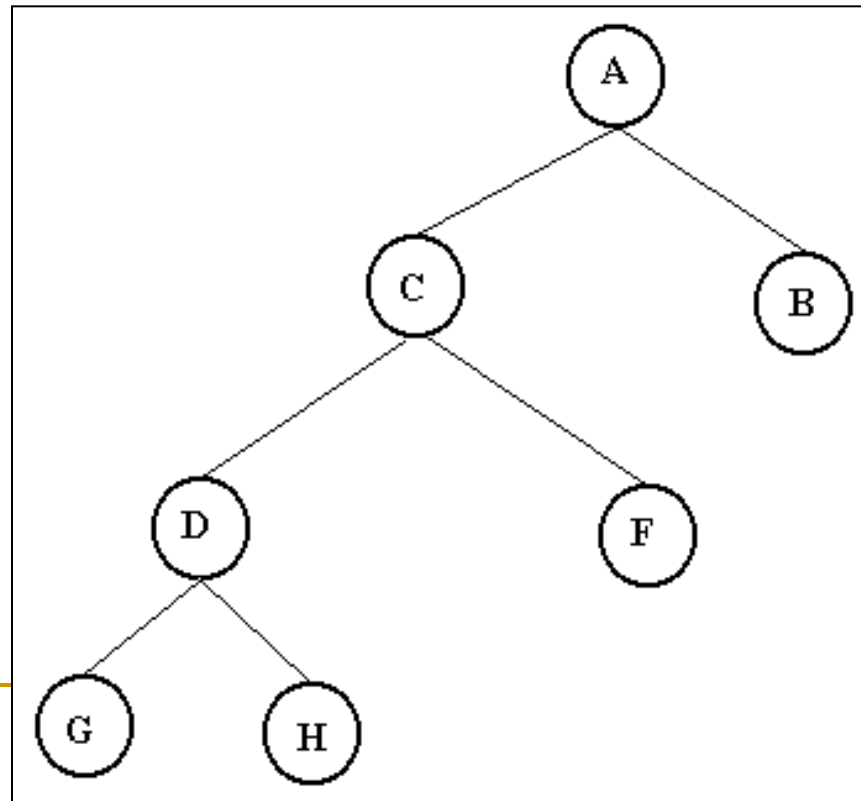
# Largura vs. profundidade

- Ao buscar um elemento
  - **Largura**
    - Acha o elemento mais rapidamente
    - Mais memória para guardar nós não visitados



# Largura vs. profundidade

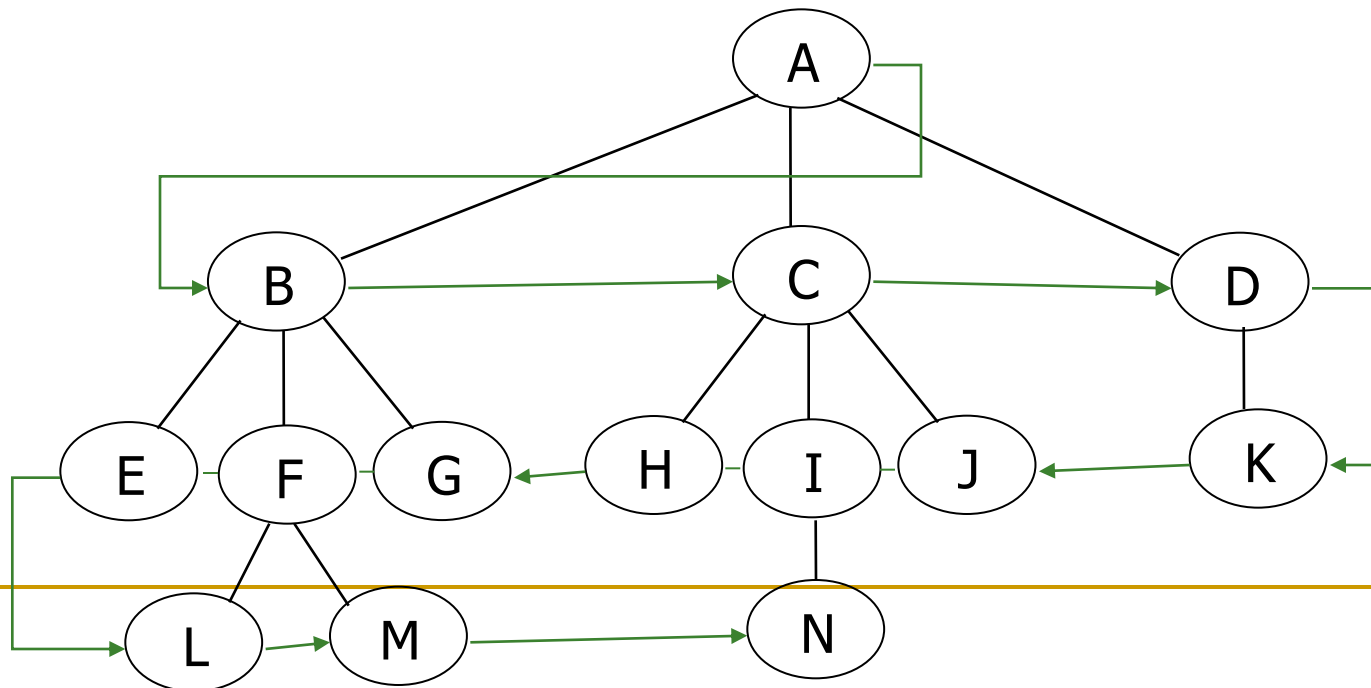
- Use as estratégias para buscar o nó F
  - Quem se sai melhor em termos de nós visitados e de memória?



# Percurso em árvores

## ■ Árvores encadeadas

- Mais ponteiros, mas pode facilitar busca e outras operações
  - Ponteiros podem ser configurados para qualquer tipo de busca
  - Filhos podem apontar para pais, para irmãos, para avôs, etc.



# Exercícios

1) Função recursiva para verificar se uma AB é balanceada (usa a função altura)

4 Casos:

1) Se for nula ou tiver só a raiz: balanceada

2) Árvore tem as 2 subárvores não nulas:

É balanceada se a diferença entre as alturas esq e dir for (0 ou 1 ou -1) e a sae for balanceada e a sad for balanceada

3. Se tem a sae não nula: é balanceada se a altura desta for 1

4. Se tem a sad não nula: é balanceada se a altura desta for 1

# Exercícios

2) Função recursiva para verificar se uma AB é perfeitamente balanceada (usa a função `nro_nós`)

4 Casos:

- 1) Se for nula ou tiver só a raiz: perf. balanceada
- 2) Árvore tem as 2 subárvores não nulas:  
É perf. balanceada se a diferença entre `nro_nós` esq e dir for (0 ou 1 ou -1) e a `sae` for perf. balanceada e a `sad` for perf. balanceada
3. Se tem a `sae` não nula: é perf. balanceada se o `nro_nós` desta for 1 (tem um único filho a esq)
4. Se tem a `sad` não nula: é perf. balanceada se o `nro_nós` desta for 1 (tem um único filho a direita)

# Exercícios

3) Função para calcular o nível de um nó.

- Dado o valor de um elemento, se ele está na árvore, retorna seu nível, retorna NULL c.c.  
OBS.: Nivel da raiz = 1\*/

/\*percorre a árvore com raiz em a em Pré-ordem, procurando pelo item dado e calculando e retornando seu nível na variável n\*/