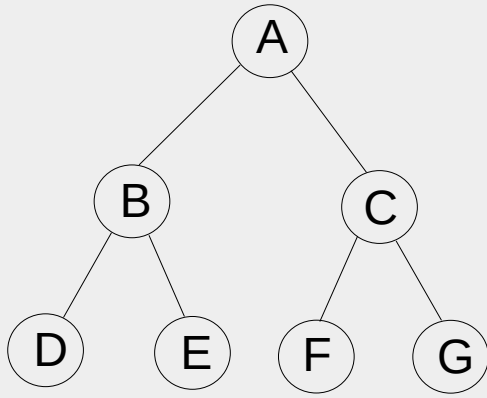


vet

0	1	2	3	4	5	6
A	B	C	D	E	F	G

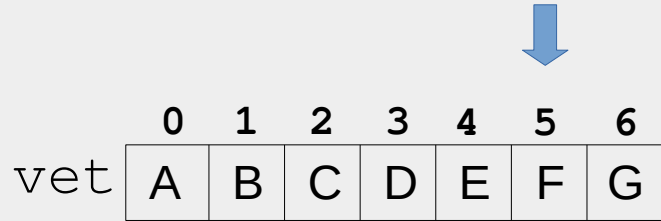
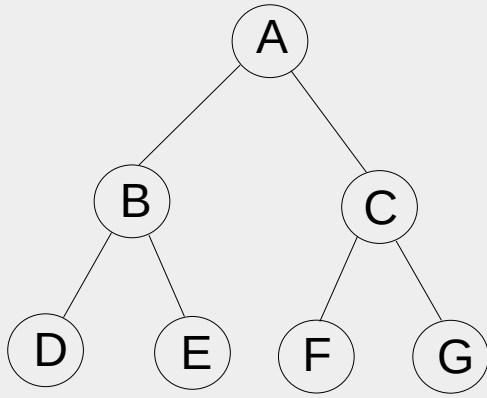
Representação de uma árvore (binária) em vetor



	0	1	2	3	4	5	6
vet	A	B	C	D	E	F	G

A partir da posição i de um nó:

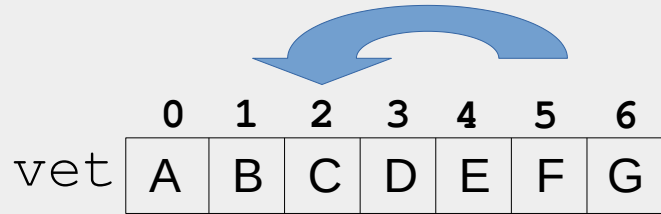
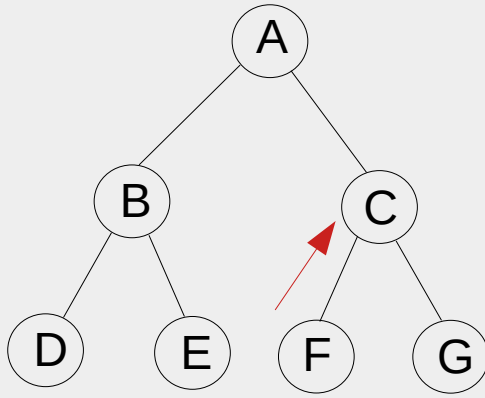
- O pai daquele nó está em $(i-1) \div 2$, para $i > 0$.
- O filho à esquerda do nó está na posição $2i+1$
- O filho à direita do nó está na posição $2i+2$



Ex: **F**

A partir da posição i de um nó:

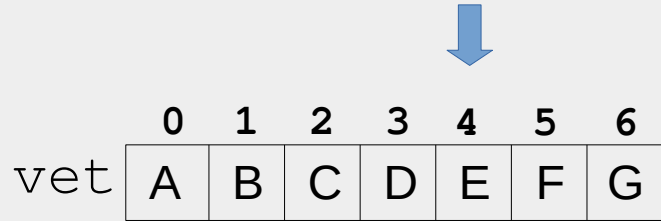
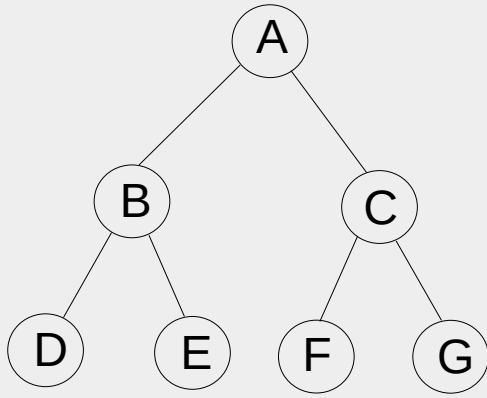
- O pai daquele nó está em $(i-1) \text{ div } 2$, para $i > 0$.
- O filho à esquerda do nó está na posição $2i+1$
- O filho à direita do nó está na posição $2i+2$



Ex: **F** – pai na casa $(5-1) \text{ div } 2 = 2$

A partir da posição i de um nó:

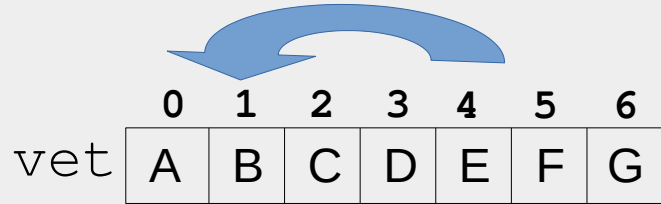
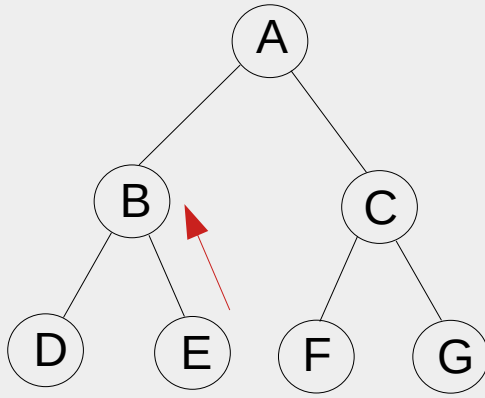
- O pai daquele nó está em $(i-1) \text{ div } 2$, para $i > 0$.
- O filho à esquerda do nó está na posição $2i+1$
- O filho à direita do nó está na posição $2i+2$



Ex: **E**

A partir da posição i de um nó:

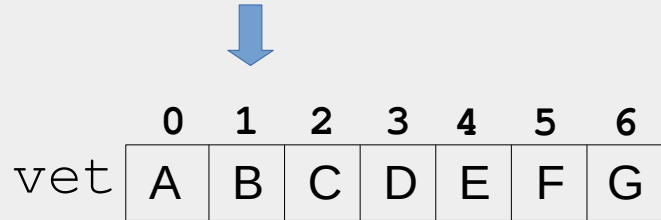
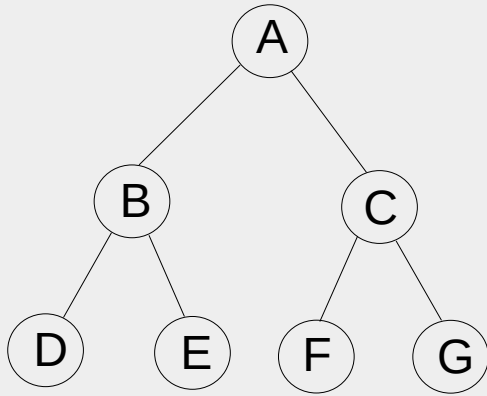
- O pai daquele nó está em $(i-1) \text{ div } 2$, para $i > 0$.
- O filho à esquerda do nó está na posição $2i+1$
- O filho à direita do nó está na posição $2i+2$



Ex: **E** – pai na casa $(4-1) \text{ div } 2 = 1$

A partir da posição i de um nó:

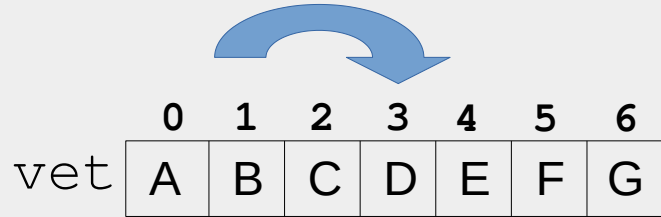
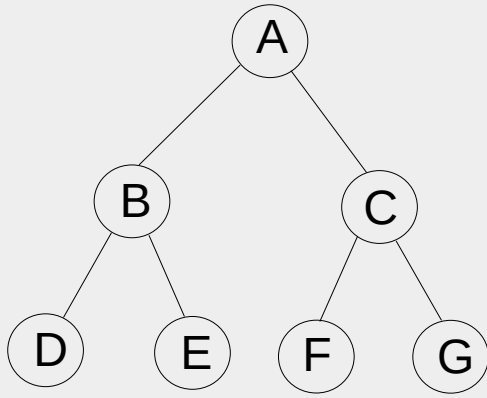
- O pai daquele nó está em $(i-1) \text{ div } 2$, para $i > 0$.
- O filho à esquerda do nó está na posição $2i+1$
- O filho à direita do nó está na posição $2i+2$



Ex: **B**

A partir da posição i de um nó:

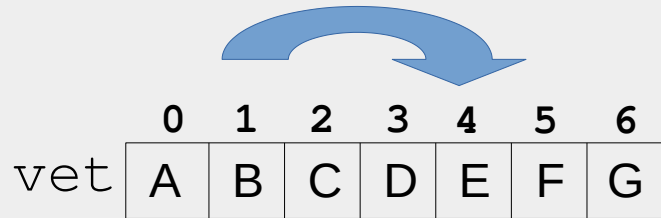
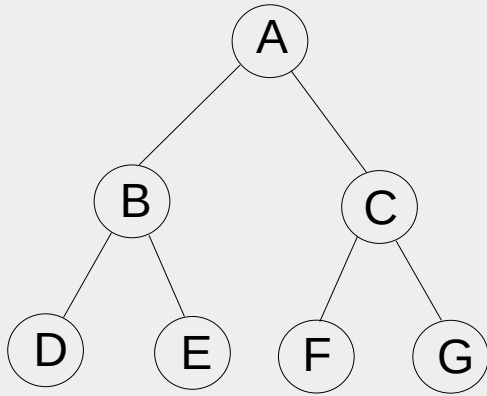
- O pai daquele nó está em $(i-1) \div 2$, para $i > 0$.
- O filho à esquerda do nó está na posição $2i+1$
- O filho à direita do nó está na posição $2i+2$



Ex: **B** – filho a esquerda na casa $2*i+1 = 3$

A partir da posição i de um nó:

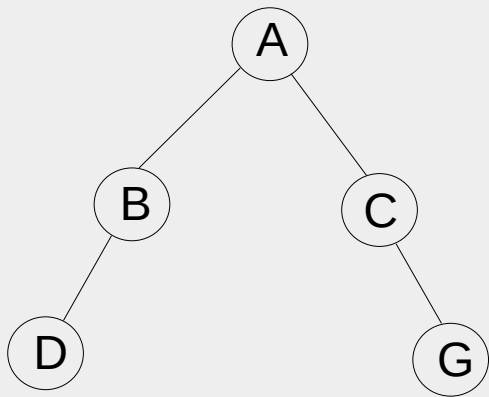
- O pai daquele nó está em $(i-1) \text{ div } 2$, para $i > 0$.
- O filho à esquerda do nó está na posição $2i+1$
- O filho à direita do nó está na posição $2i+2$



Ex: **B** – filho a direita na casa $2*i+2 = 4$

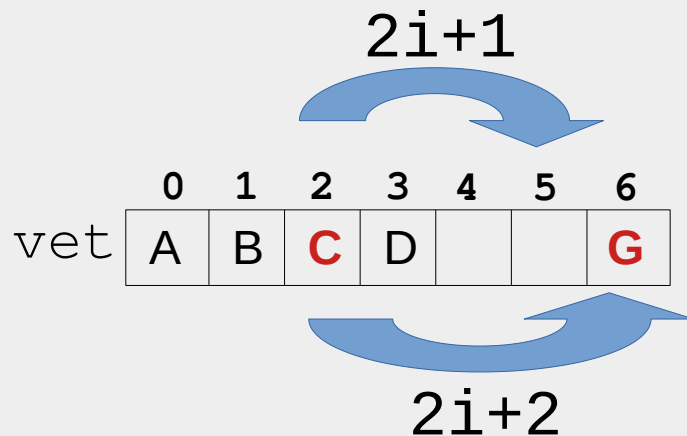
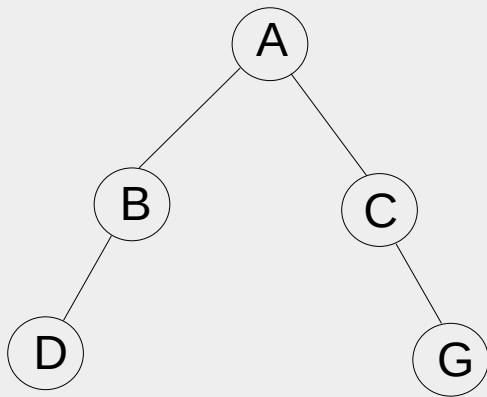
A partir da posição i de um nó:

- O pai daquele nó está em $(i-1) \text{ div } 2$, para $i > 0$.
- O filho à esquerda do nó está na posição $2i+1$
- O filho à direita do nó está na posição $2i+2$



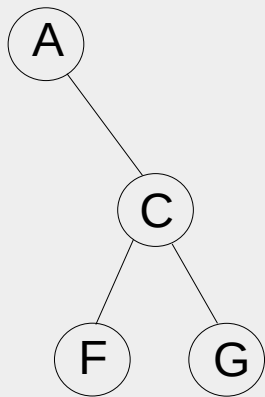
	0	1	2	3	4	5	6
vet	A	B	C	D			G

Para árvores incompletas: desperdício de espaço

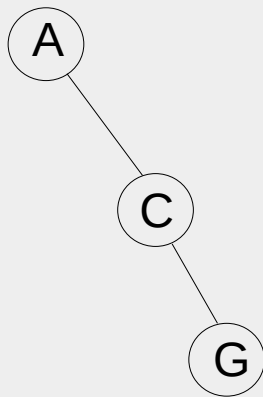


Para árvores incompletas: desperdício de espaço

Deve-se preservar as relações entre os índices para pais-filhos...

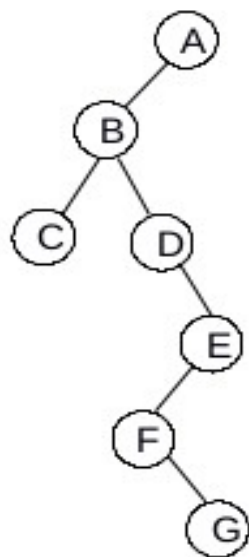


	0	1	2	3	4	5	6
vet	A		C			F	G



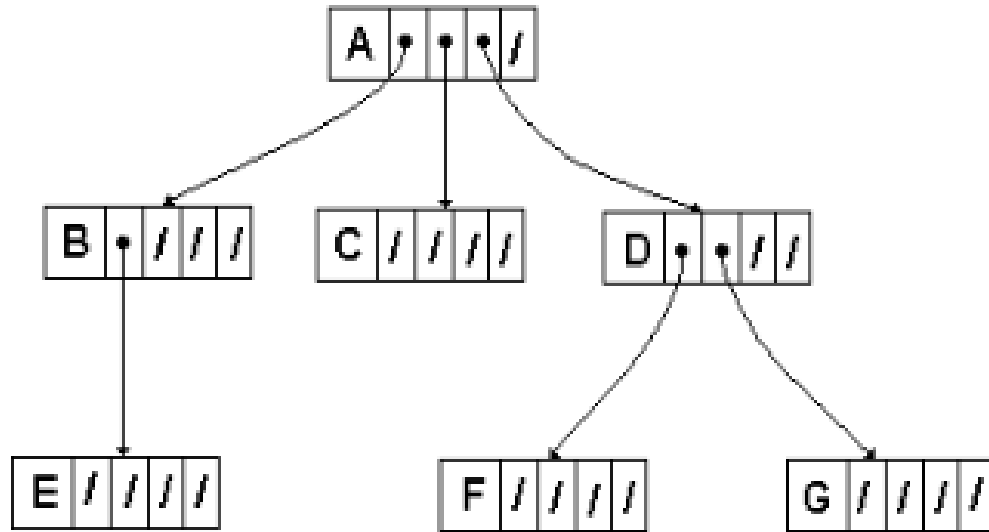
	0	1	2	3	4	5	6
vet	A		C				G

```
typedef int tDado;  
const int N = 10;  
struct NoArv  
{  
    tDado val;  
    unsigned filhEsq;  
    unsigned filhDir;  
};  
typedef NoArv VetArv[N];
```



0	1	2	3	4	5	6
A	B	C	D	E	F	G
1	2	0	0	5	0	0
0	3	0	4	0	6	0

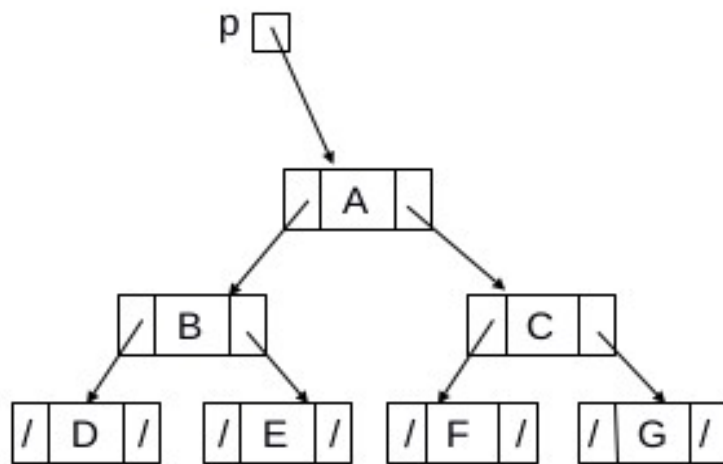
Para árvores não binárias:
possibilidade de mais filhos que o
número previsto de ponteiros.



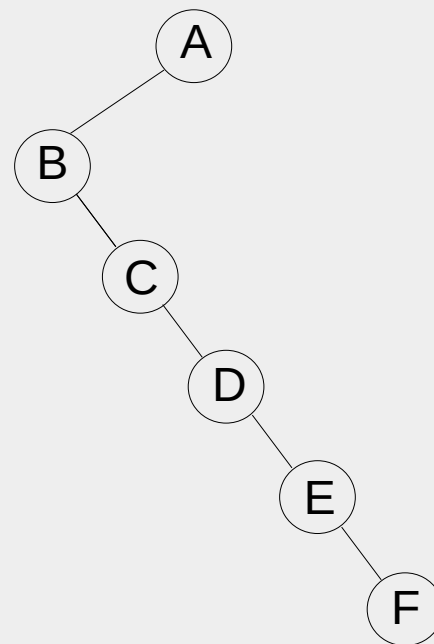
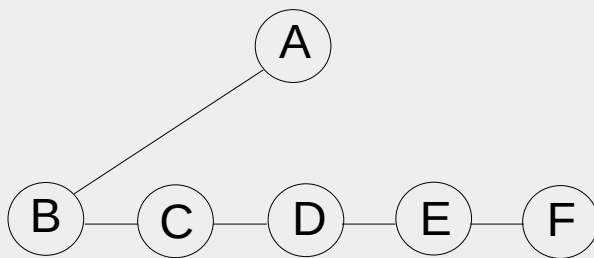
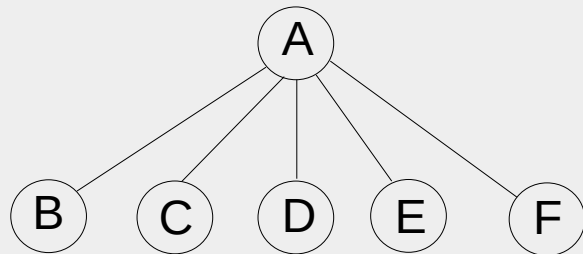
Nos vários casos há algum desperdício de espaço.

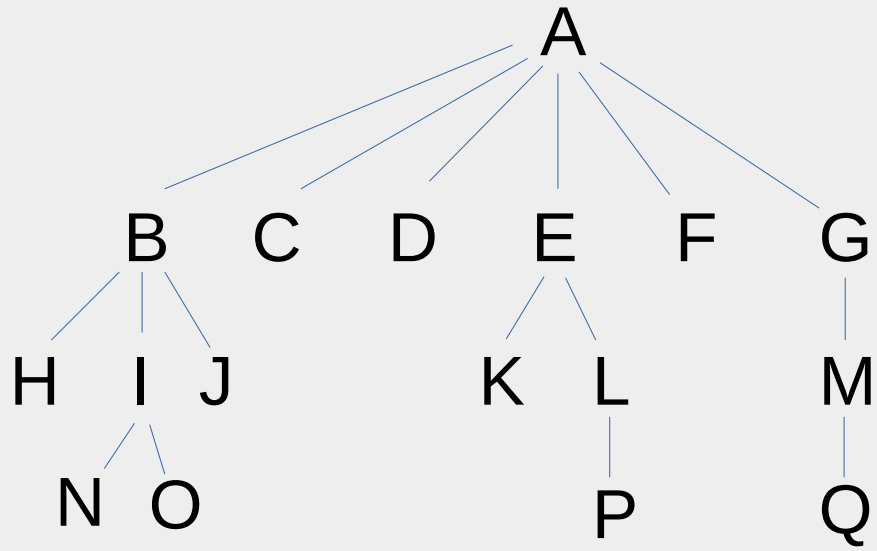
A questão é buscar uma solução de equilíbrio: sem muito desperdício e garantindo-se que as expansões serão suportadas.

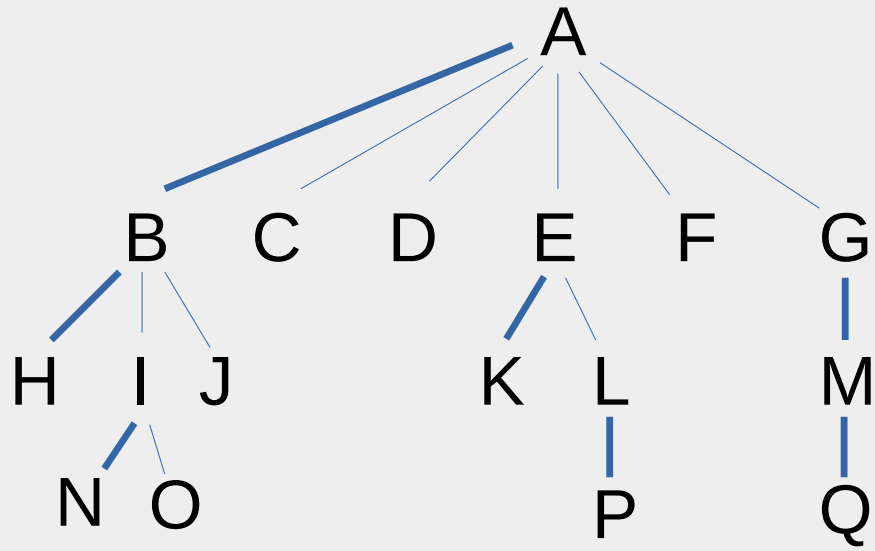

```
struct NoArvBin
{
    NoArvBin * esq;
    tDado dado;
    NoArvBin * dir;
}
typedef NoArvBin * PtNo;
PtNo p;
```

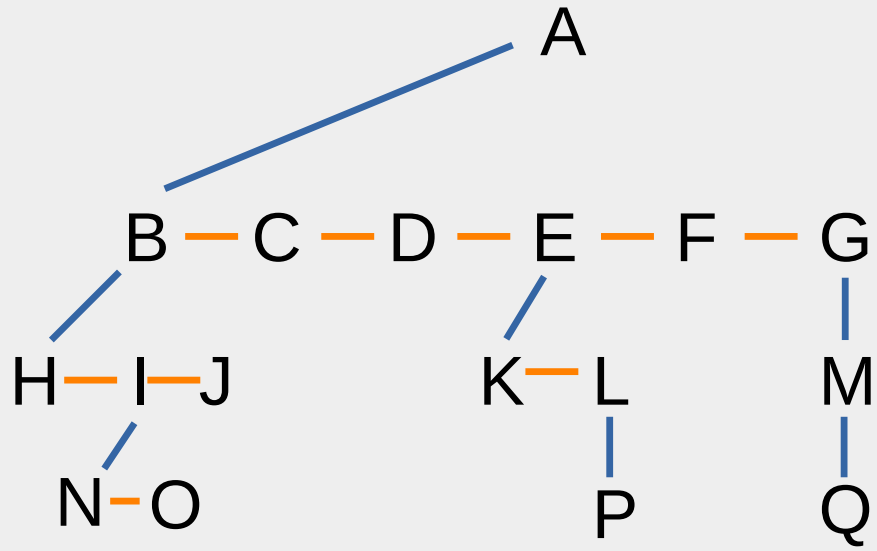


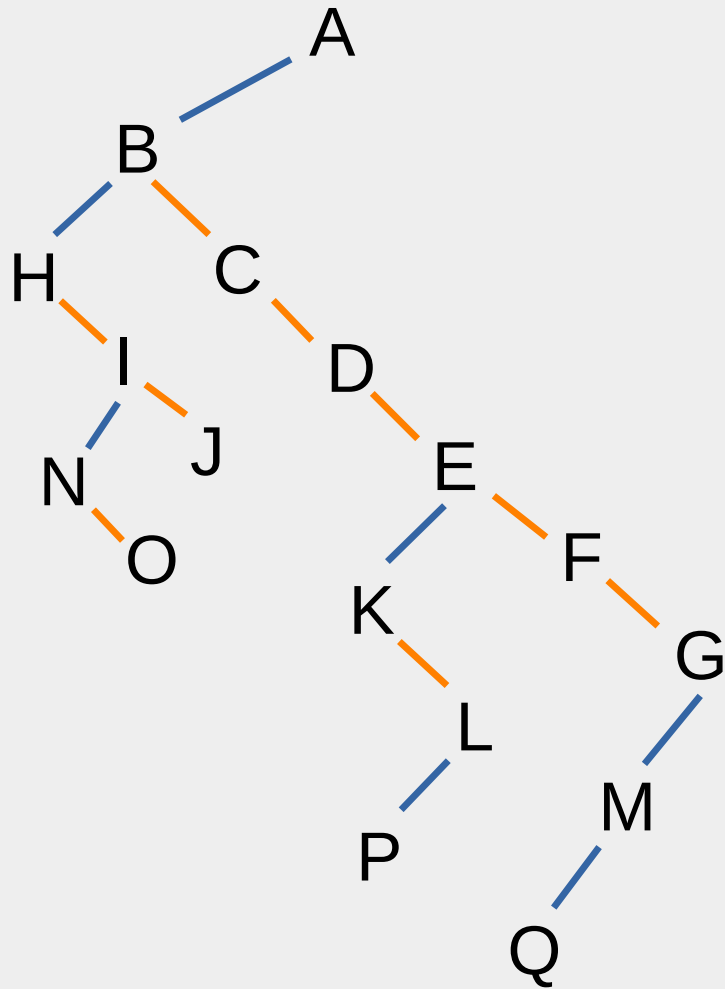
Transformação de uma árvore qualquer em árvore binária







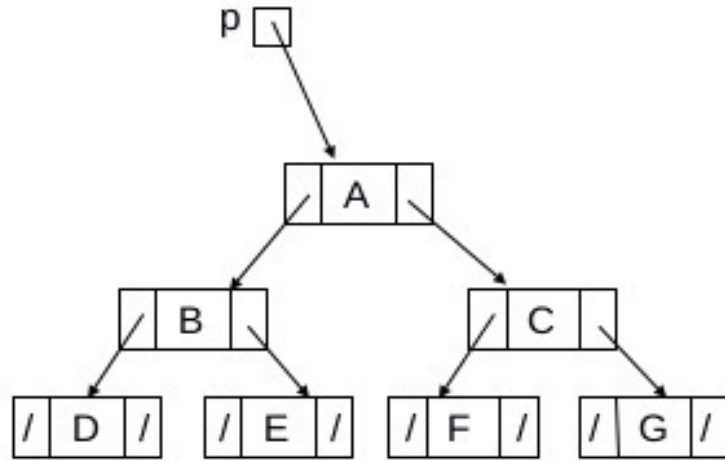




Interpretação:

- um filho à esquerda de um nó é filho de fato.

- um filho à direita de um nó é seu irmão

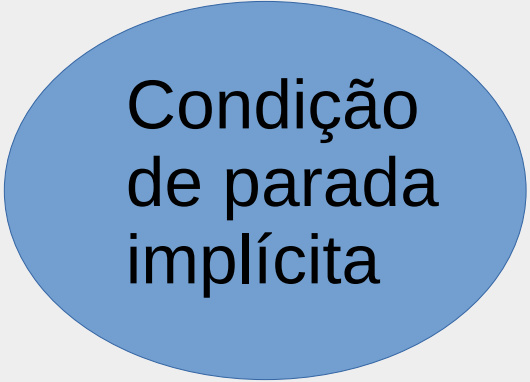


Vantagem da transformação:

Não é necessário ter conhecimento prévio da estrutura (grau máximo).

E na alocação encadeada, alocam-se apenas os nós necessários (embora os ponteiros sejam uma sobrecarga).


```
void PercPreOrdem (PtNo  PtRz) {  
    if (PtRz != NULL)  
    {  
        Visita (PtRz) ;  
        PercPreOrdem (PtRz->esq) ;  
        PercPreOrdem (PtRz->dir) ;  
    }  
}
```



Condição
de parada
implícita

```
void PercPreOrdem (PtNo PtRz) {  
    if (PtRz != NULL)  
    {  
        Visita (PtRz);  
        PercPreOrdem (PtRz->esq);  
        PercPreOrdem (PtRz->dir);  
    }  
}
```

```
void Visita (PtNo p)  
{  
    cout << p->dado;  
    cout << "  ";  
}
```

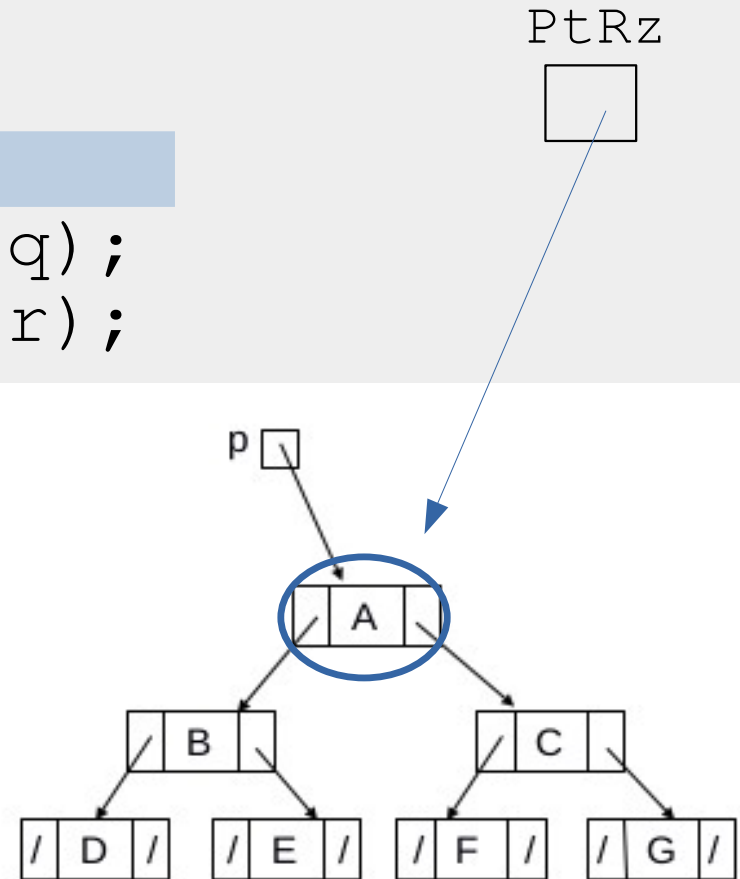
```
void PercPreOrdem(PtNo PtRz) {  
    if (PtRz == NULL) // cond. parada  
        return;  
    Visita(PtRz);  
    PercPreOrdem(PtRz->esq);  
    PercPreOrdem(PtRz->dir);  
  
}
```

```
void PercPreOrdem(PtNo PtRz) {  
    if (PtRz == NULL) // cond. parada  
        ;  
    else {  
        Visita(PtRz);  
        PercPreOrdem(PtRz->esq);  
        PercPreOrdem(PtRz->dir);  
    }  
  
}
```

```

void PercPreOrdem (PtNo PtRz) {
    if (PtRz != NULL)
    {
        Visita (PtRz);
        PercPreOrdem (PtRz->esq);
        PercPreOrdem (PtRz->dir);
    }
}

```

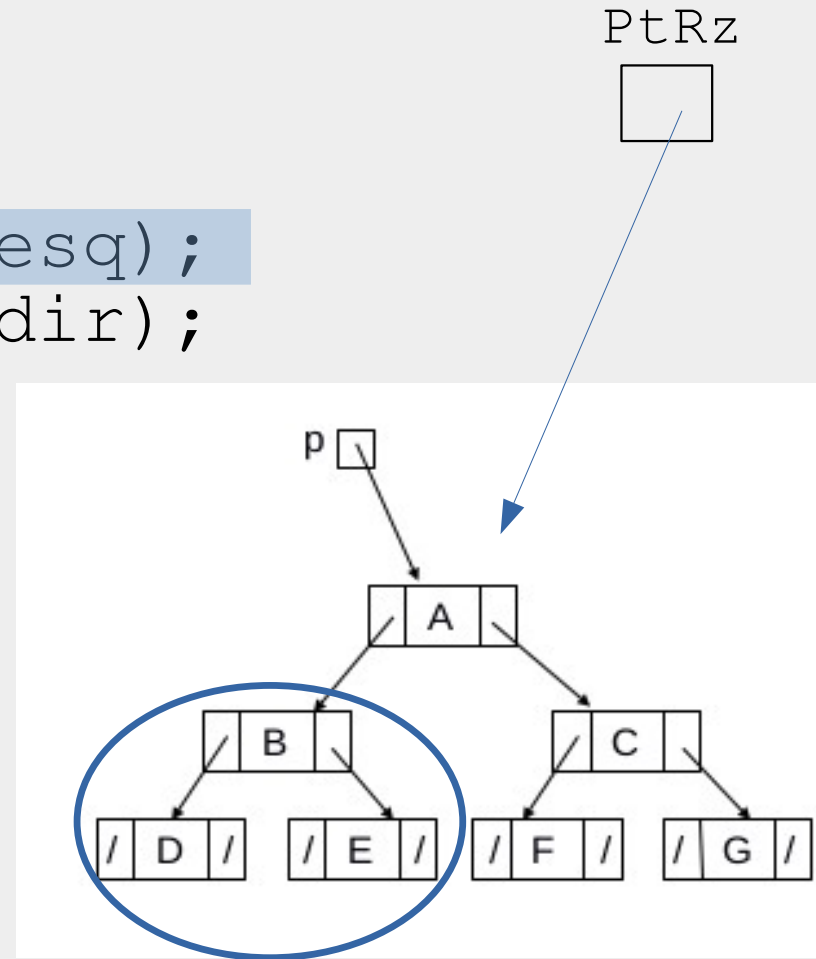


```

void PercPreOrdem (PtNo PtRz) {
    if (PtRz != NULL)
    {
        Visita (PtRz);
        PercPreOrdem (PtRz->esq);
        PercPreOrdem (PtRz->dir);
    }
}

```

A

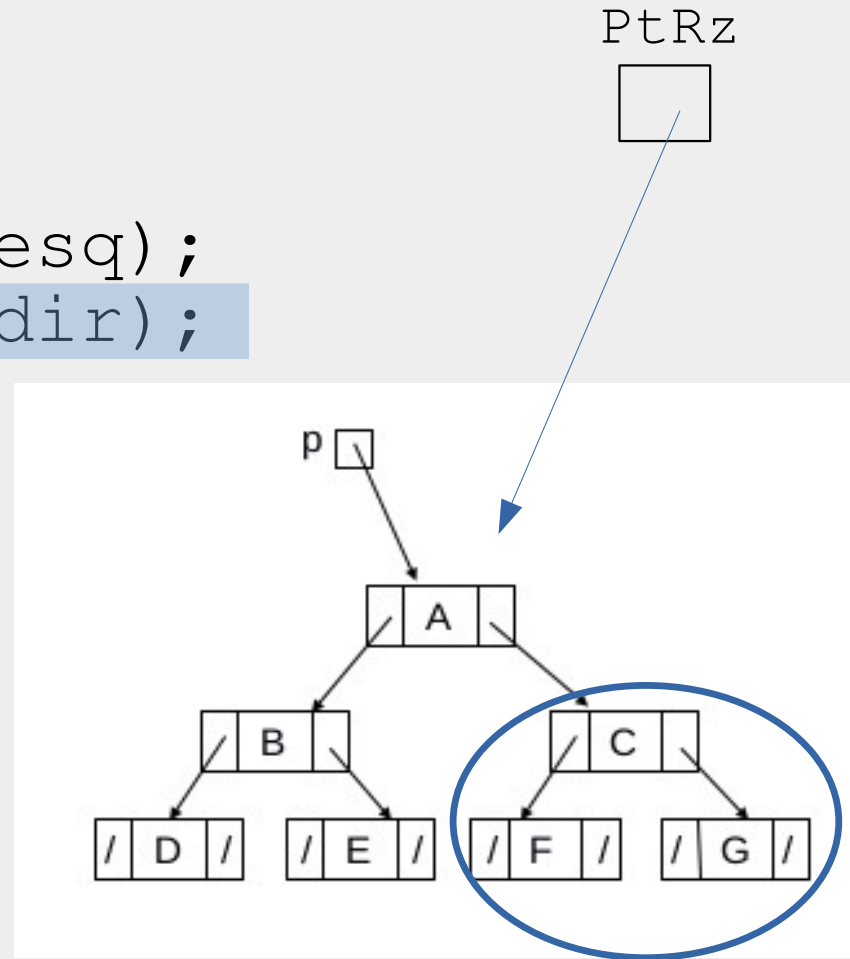


```

void PercPreOrdem (PtNo PtRz) {
    if (PtRz != NULL)
    {
        Visita (PtRz);
        PercPreOrdem (PtRz->esq);
        PercPreOrdem (PtRz->dir);
    }
}

```

A B D E

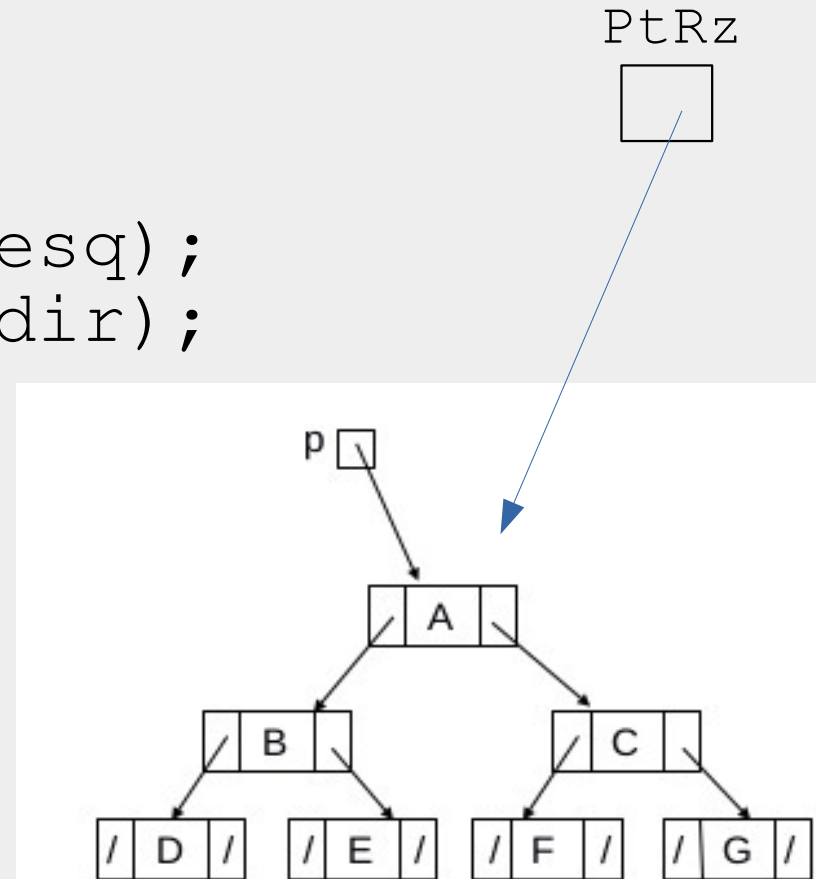


```

void PercPreOrdem (PtNo PtRz) {
    if (PtRz != NULL)
    {
        Visita (PtRz);
        PercPreOrdem (PtRz->esq);
        PercPreOrdem (PtRz->dir);
    }
}

```

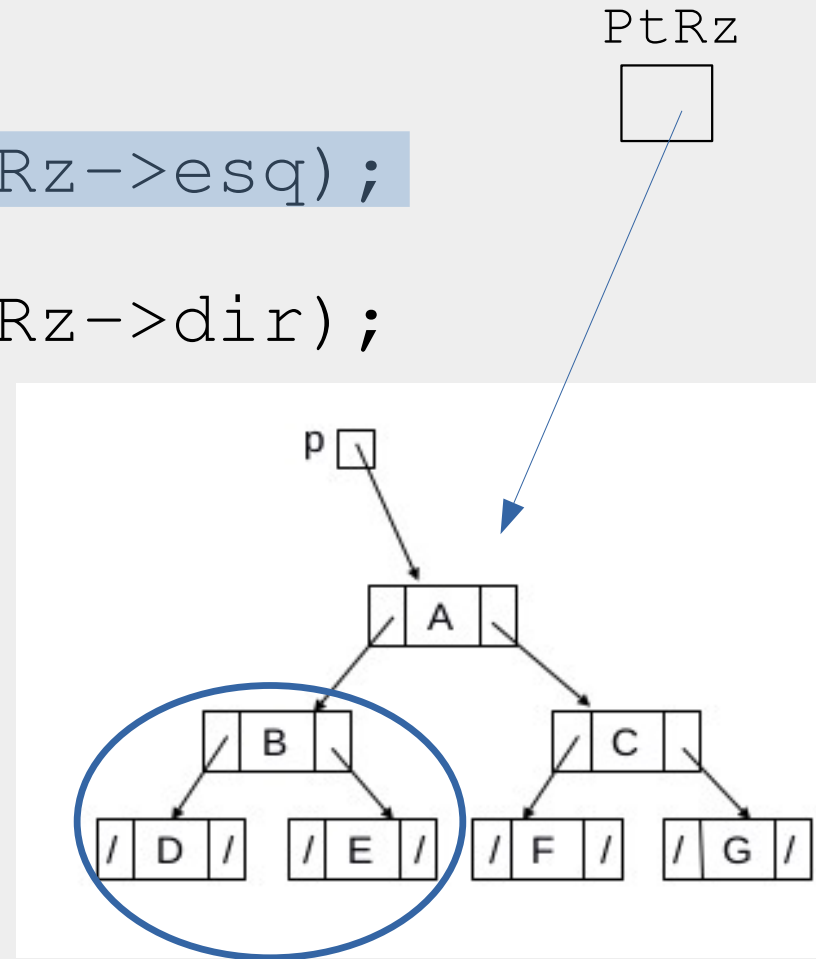
A B D E C F G




```

void PercOrdemCentral (PtNo PtRz) {
    if (PtRz != NULL)
    {
        PercOrdemCentral (PtRz->esq);
        Visita (PtRz);
        PercOrdemCentral (PtRz->dir);
    }
}

```

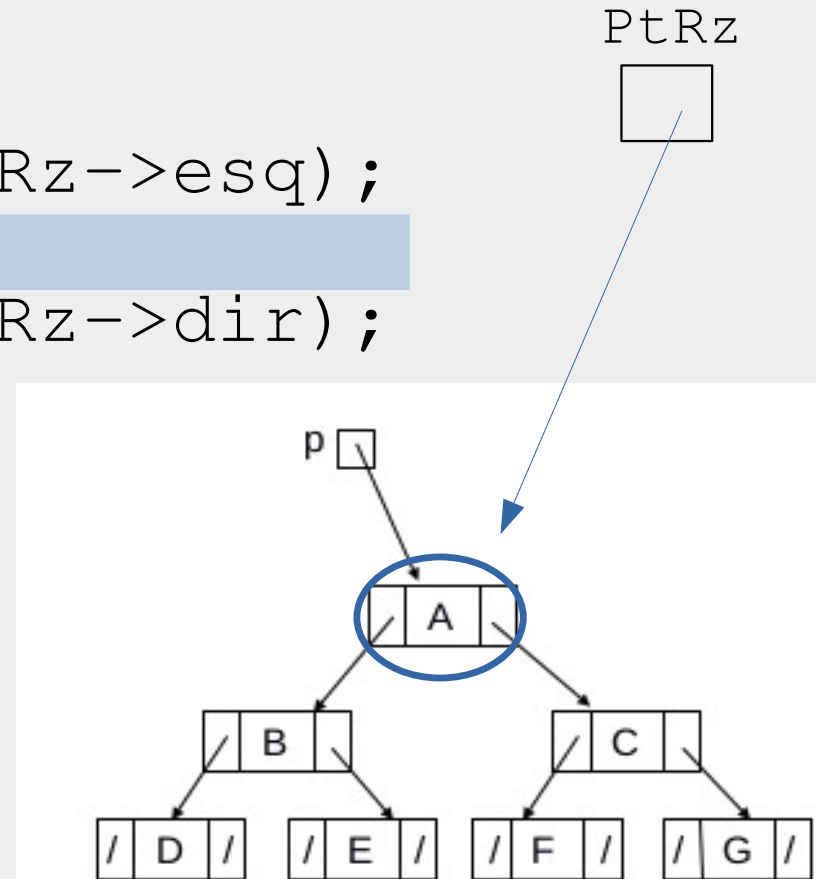


```

void PercOrdemCentral (PtNo PtRz) {
    if (PtRz != NULL)
    {
        PercOrdemCentral (PtRz->esq);
        Visita (PtRz);
        PercOrdemCentral (PtRz->dir);
    }
}

```

D B E

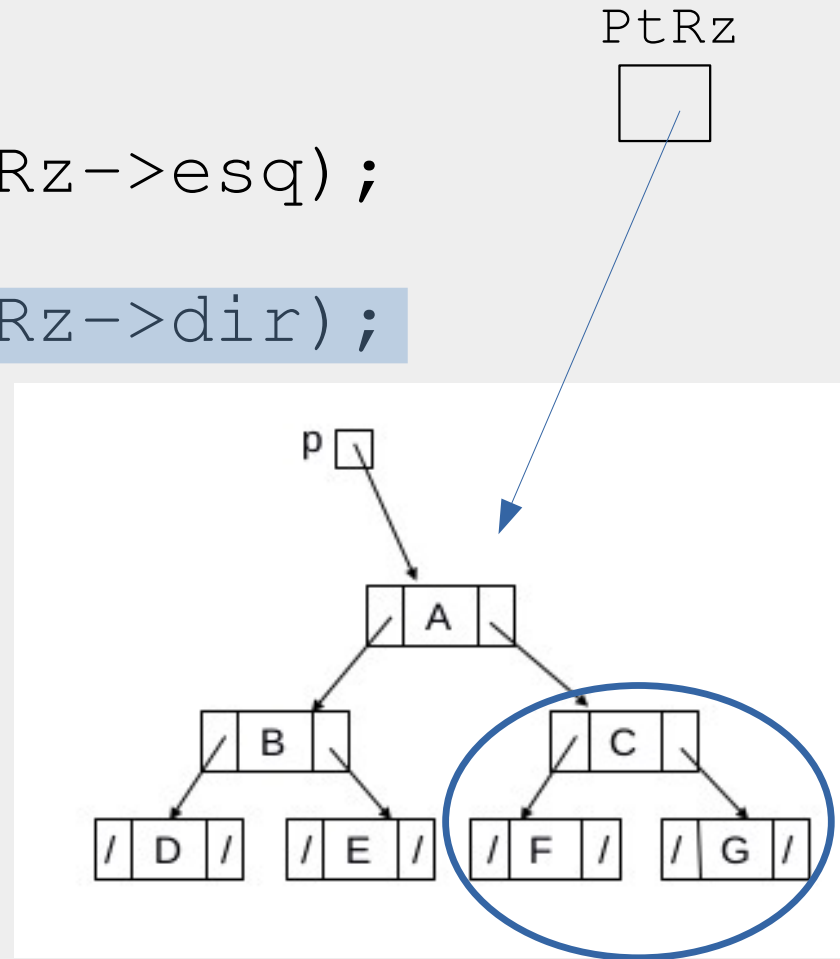


```

void PercOrdemCentral (PtNo PtRz) {
    if (PtRz != NULL)
    {
        PercOrdemCentral (PtRz->esq);
        Visita (PtRz);
        PercOrdemCentral (PtRz->dir);
    }
}

```

D B E A

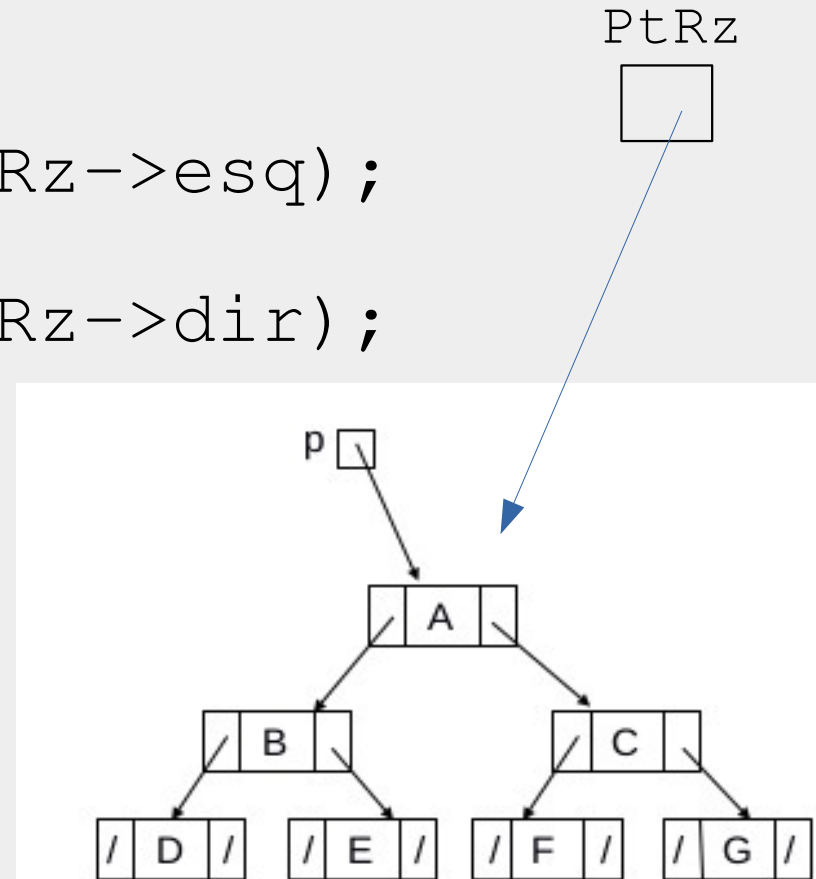


```

void PercOrdemCentral (PtNo PtRz) {
    if (PtRz != NULL)
    {
        PercOrdemCentral (PtRz->esq);
        Visita (PtRz);
        PercOrdemCentral (PtRz->dir);
    }
}

```

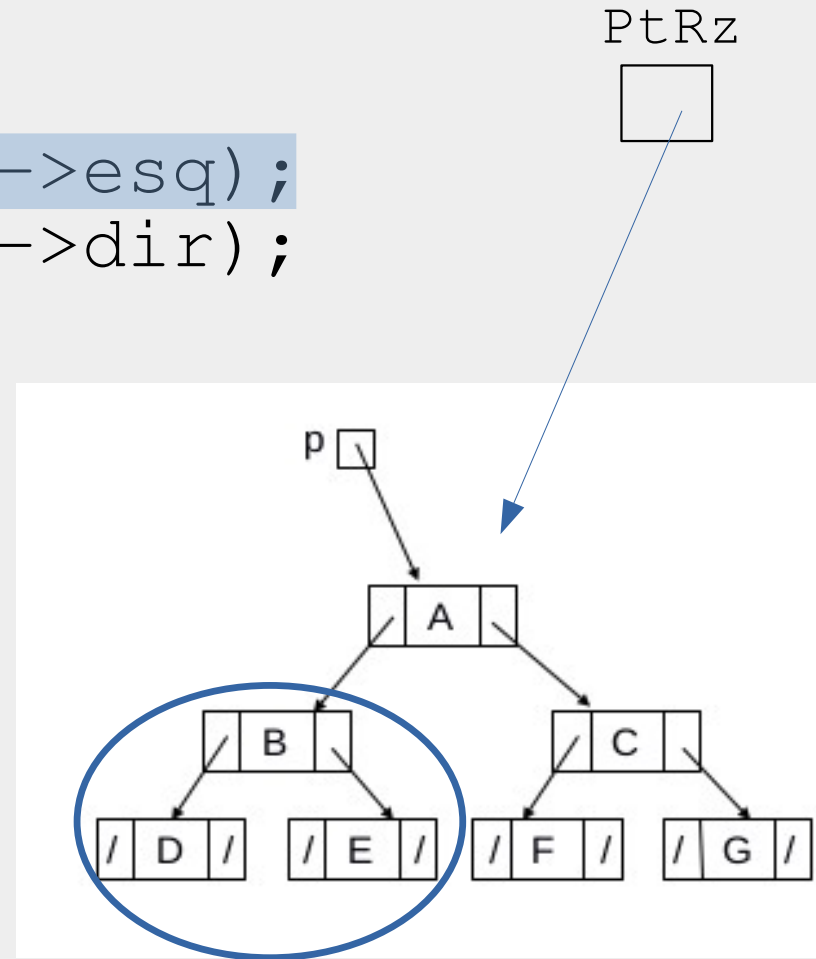
D B E A F C G



```

void PercOrdemFinal (PtNo PtRz) {
    if (PtRz != NULL)
    {
        PercOrdemFinal (PtRz->esq);
        PercOrdemFinal (PtRz->dir);
        Visita (PtRz);
    }
}

```

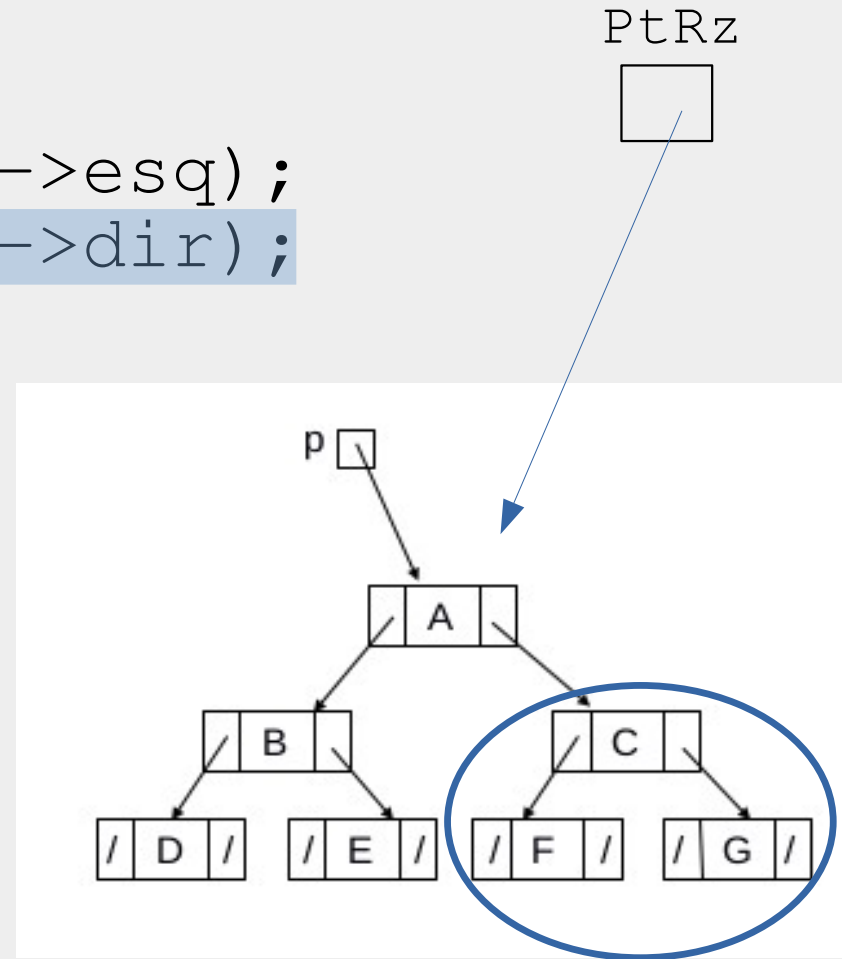


```

void PercOrdemFinal (PtNo PtRz) {
    if (PtRz != NULL)
    {
        PercOrdemFinal (PtRz->esq);
        PercOrdemFinal (PtRz->dir);
        Visita (PtRz);
    }
}

```

D E B

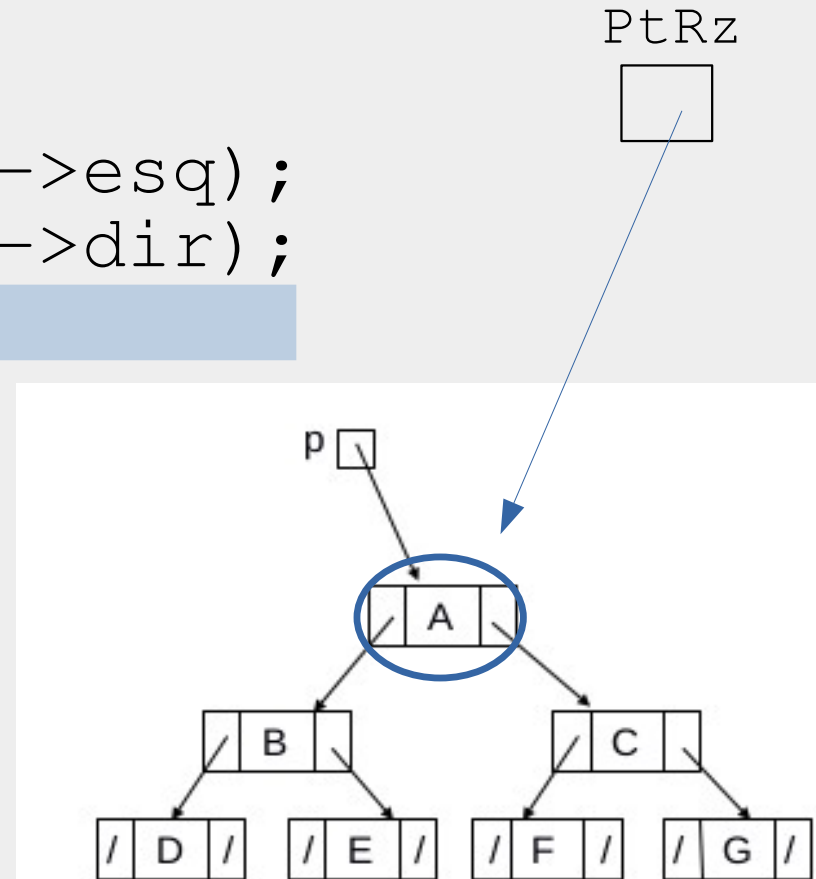


```

void PercOrdemFinal (PtNo PtRz) {
    if (PtRz != NULL)
    {
        PercOrdemFinal (PtRz->esq);
        PercOrdemFinal (PtRz->dir);
        Visita (PtRz);
    }
}

```

D E B F G C

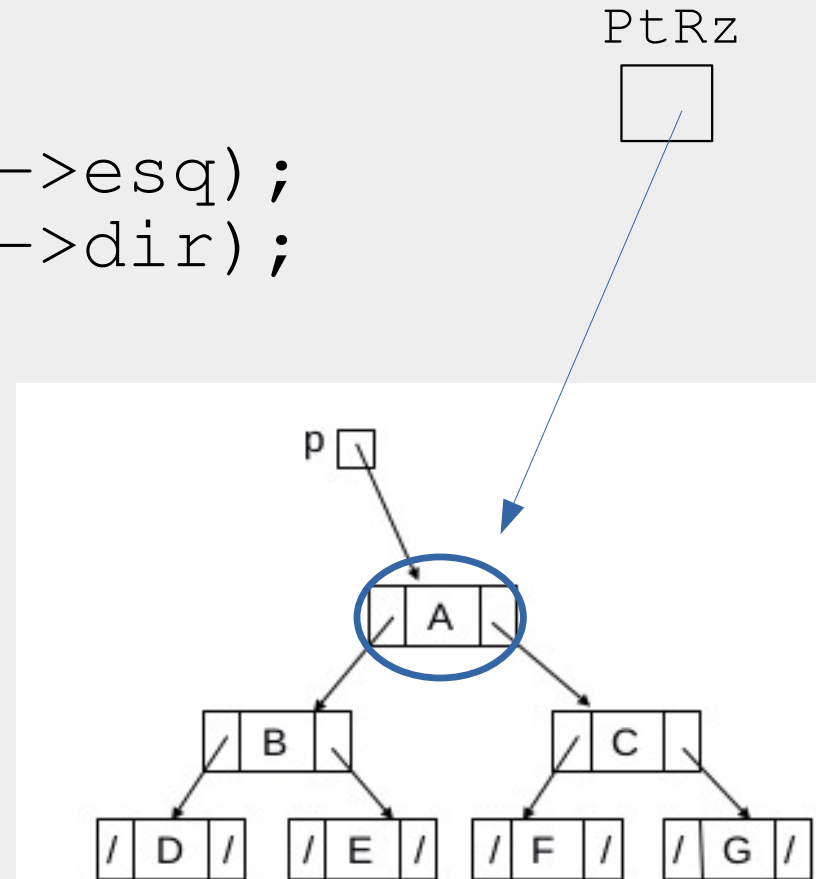


```

void PercOrdemFinal (PtNo PtRz) {
    if (PtRz != NULL)
    {
        PercOrdemFinal (PtRz->esq);
        PercOrdemFinal (PtRz->dir);
        Visita (PtRz);
    }
}

```

D E B F G C A



Construção de árvores

```
struct NoArvBin
{
    NoArvBin * esq;
    tDado dado;
    NoArvBin * dir;
}
typedef NoArvBin * PtNo;
```

q □ p □

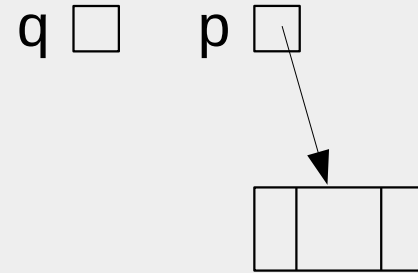
```
PtNo p, q;
```

```
p = new NoArvBin;
p->dado = 3;
p->dir = NULL;
q = new NoArvBin;
q->dado = 5;
q->esq = NULL;
q->dir = NULL;
p->esq = q;
```

```
struct NoArvBin
{
    NoArvBin * esq;
    tDado dado;
    NoArvBin * dir;
}
typedef NoArvBin * PtNo;
```

```
PtNo p, q;
```

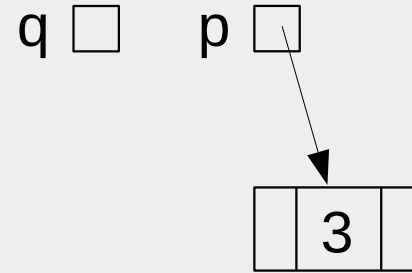
```
p = new NoArvBin;
p->dado = 3;
p->dir = NULL;
q = new NoArvBin;
q->dado = 5;
q->esq = NULL;
q->dir = NULL;
p->esq = q;
```



```
struct NoArvBin
{
    NoArvBin * esq;
    tDado dado;
    NoArvBin * dir;
}
typedef NoArvBin * PtNo;
```

```
PtNo p, q;
```

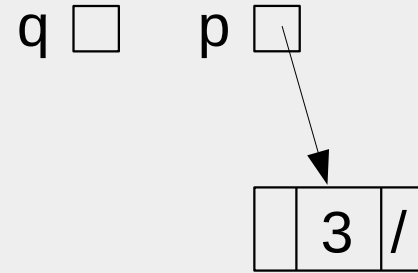
```
p = new NoArvBin;
p->dado = 3;
p->dir = NULL;
q = new NoArvBin;
q->dado = 5;
q->esq = NULL;
q->dir = NULL;
p->esq = q;
```



```
struct NoArvBin
{
    NoArvBin * esq;
    tDado dado;
    NoArvBin * dir;
}
typedef NoArvBin * PtNo;
```

```
PtNo p, q;
```

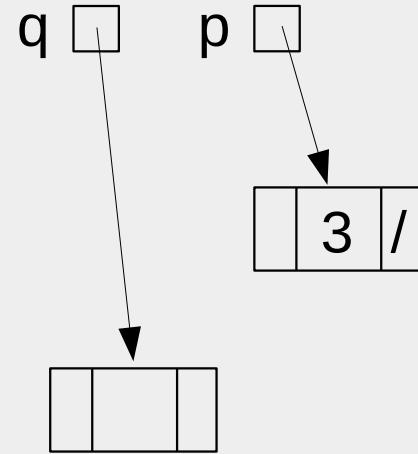
```
p = new NoArvBin;
p->dado = 3;
p->dir = NULL;
q = new NoArvBin;
q->dado = 5;
q->esq = NULL;
q->dir = NULL;
p->esq = q;
```



```
struct NoArvBin
{
    NoArvBin * esq;
    tDado dado;
    NoArvBin * dir;
}
typedef NoArvBin * PtNo;
```

```
PtNo p, q;
```

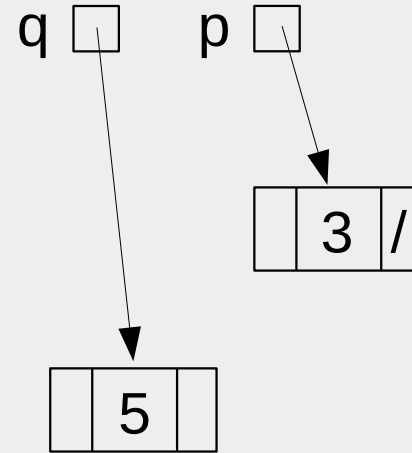
```
p = new NoArvBin;
p->dado = 3;
p->dir = NULL;
q = new NoArvBin;
q->dado = 5;
q->esq = NULL;
q->dir = NULL;
p->esq = q;
```



```
struct NoArvBin
{
    NoArvBin * esq;
    tDado dado;
    NoArvBin * dir;
}
typedef NoArvBin * PtNo;
```

```
PtNo p, q;
```

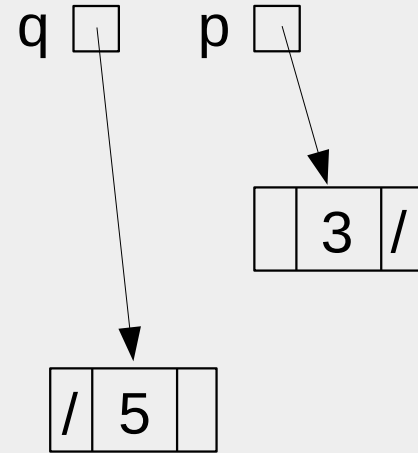
```
p = new NoArvBin;
p->dado = 3;
p->dir = NULL;
q = new NoArvBin;
q->dado = 5;
q->esq = NULL;
q->dir = NULL;
p->esq = q;
```



```
struct NoArvBin
{
    NoArvBin * esq;
    tDado dado;
    NoArvBin * dir;
}
typedef NoArvBin * PtNo;
```

```
PtNo p, q;
```

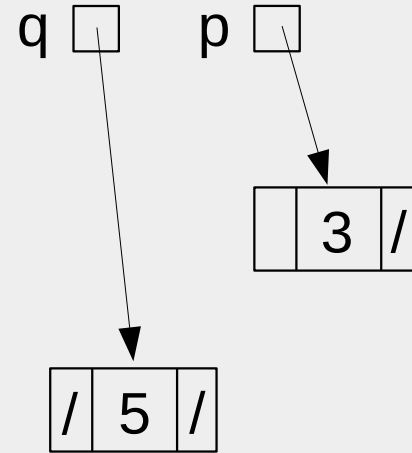
```
p = new NoArvBin;
p->dado = 3;
p->dir = NULL;
q = new NoArvBin;
q->dado = 5;
q->esq = NULL;
q->dir = NULL;
p->esq = q;
```




```
struct NoArvBin
{
    NoArvBin * esq;
    tDado dado;
    NoArvBin * dir;
}
typedef NoArvBin * PtNo;
```

```
PtNo p, q;
```

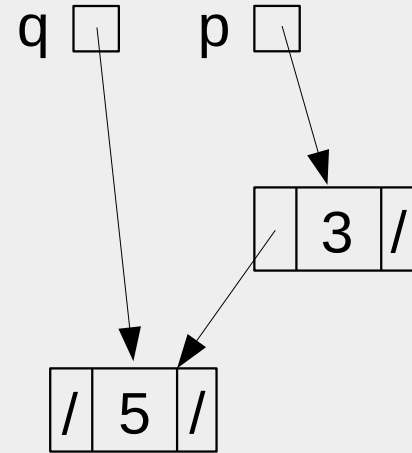
```
p = new NoArvBin;
p->dado = 3;
p->dir = NULL;
q = new NoArvBin;
q->dado = 5;
q->esq = NULL;
q->dir = NULL;
p->esq = q;
```



```
struct NoArvBin
{
    NoArvBin * esq;
    tDado dado;
    NoArvBin * dir;
}
typedef NoArvBin * PtNo;
```

```
PtNo p, q;
```

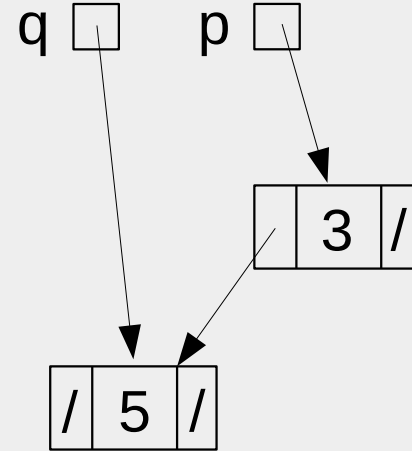
```
p = new NoArvBin;
p->dado = 3;
p->dir = NULL;
q = new NoArvBin;
q->dado = 5;
q->esq = NULL;
q->dir = NULL;
p->esq = q;
```



```
struct NoArvBin
{
    NoArvBin * esq;
    tDado dado;
    NoArvBin * dir;
}
typedef NoArvBin * PtNo;
```

```
PtNo p, q;
```

```
p = new NoArvBin;
p->dado = 3;
p->dir = NULL;
q = new NoArvBin;
q->dado = 5;
q->esq = NULL;
q->dir = NULL;
p->esq = q;
```

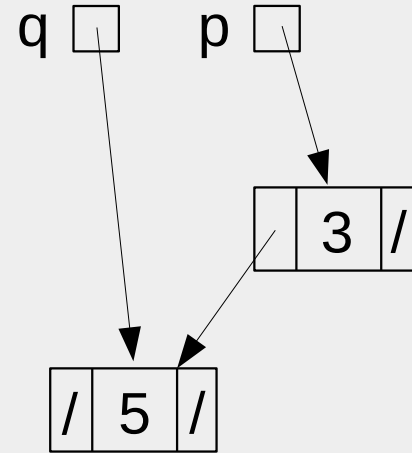


Poderíamos continuar
essa construção
“artesanal”...

```
struct NoArvBin
{
    NoArvBin * esq;
    tDado dado;
    NoArvBin * dir;
}
typedef NoArvBin * PtNo;
```

```
PtNo p, q;
```

```
p = new NoArvBin;
p->dado = 3;
p->dir = NULL;
q = new NoArvBin;
q->dado = 5;
q->esq = NULL;
q->dir = NULL;
p->esq = q;
```



Não é a
forma mais
indicada...

**Usando procedimientos
padronizados:**

Usando procedimentos padronizados:

Caso particular:

- Árvore binária
- Alocação encadeada
- Nós serão fornecidos em ordem pré-fixada

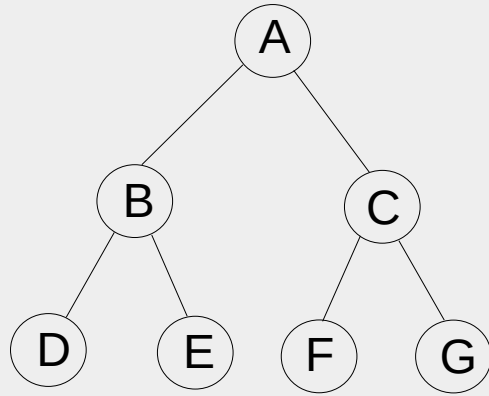
Usando procedimentos padronizados:

Caso particular:

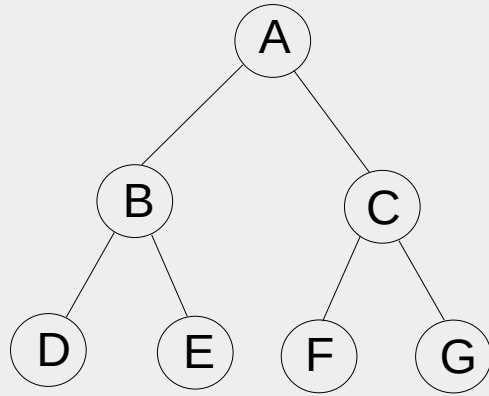
- Árvore binária
- Alocação encadeada
- Nós serão fornecidos em ordem pré-fixada

Uma barra (ou outro código, no caso de valores numéricos) indicará uma subárvore vazia.

Exemplo:

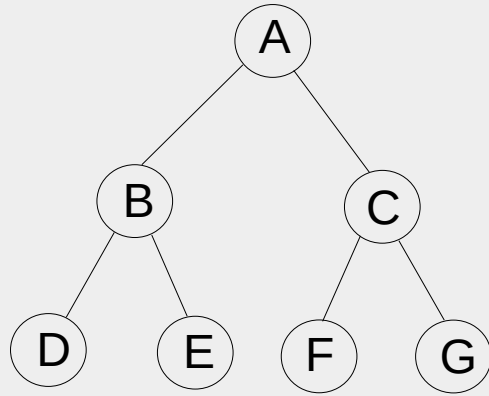


Exemplo:



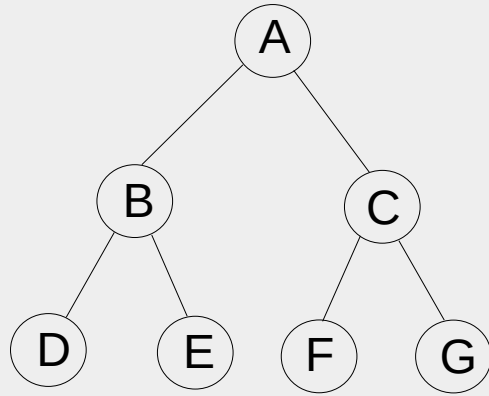
A

Exemplo:



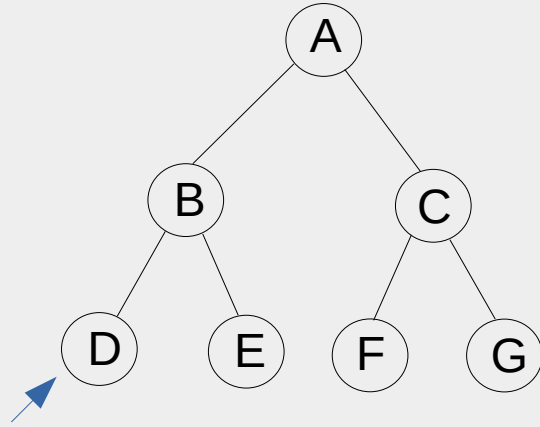
A B

Exemplo:



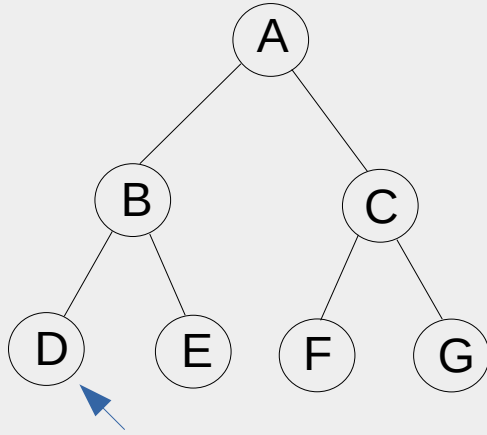
A B D

Exemplo:



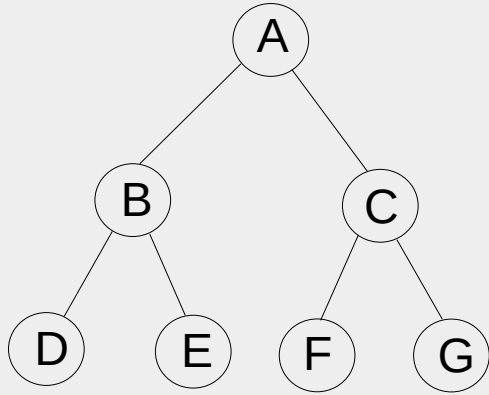
A B D /

Exemplo:



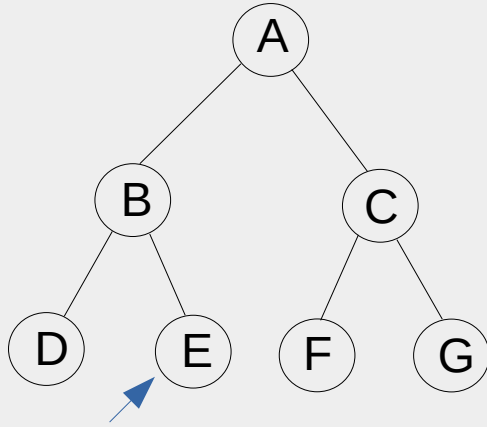
A B D / /

Exemplo:



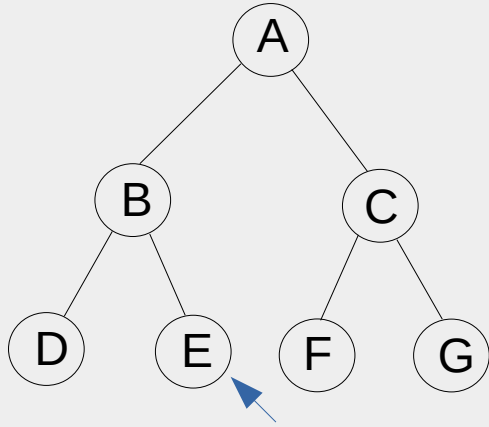
A B D / / E

Exemplo:



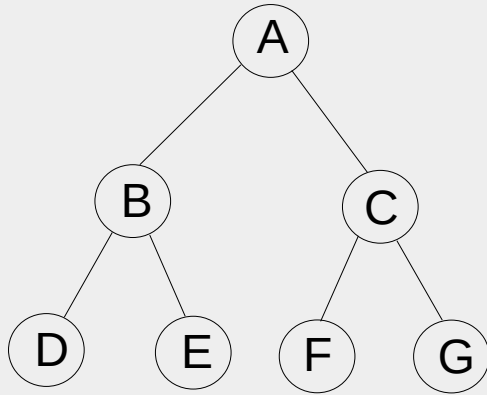
A B D / / E /

Exemplo:



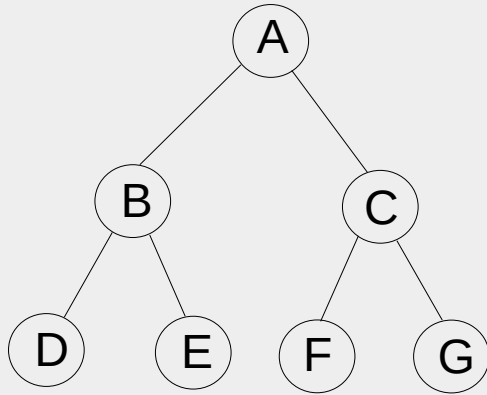
A B D / / E / /

Exemplo:



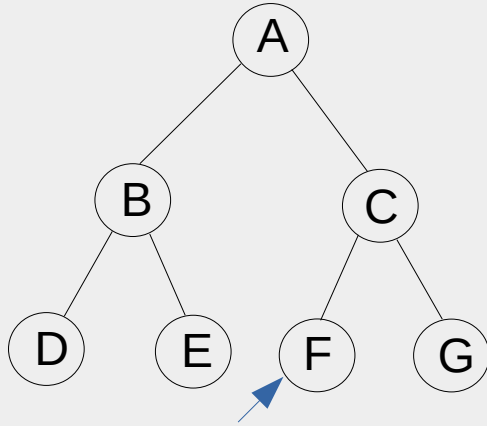
A B D / / E / / C

Exemplo:



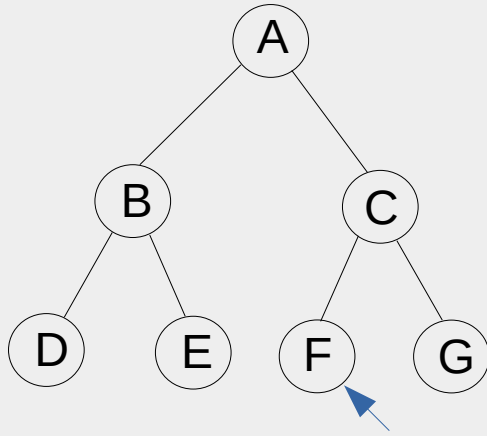
A B D / / E / / C F

Exemplo:



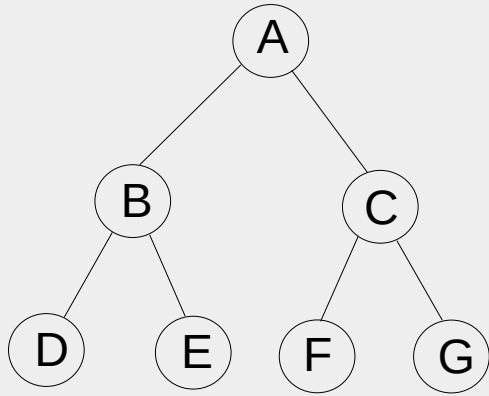
A B D / / E / / C F /

Exemplo:



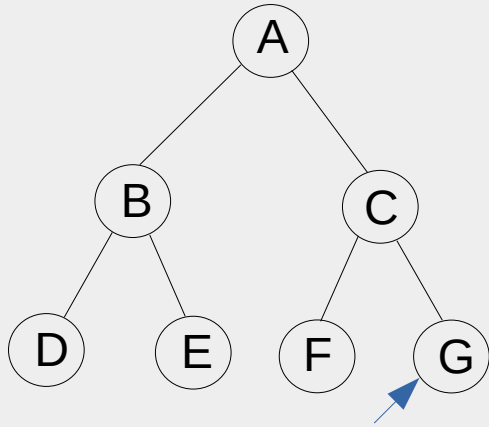
A B D / / E / / C F / /

Exemplo:



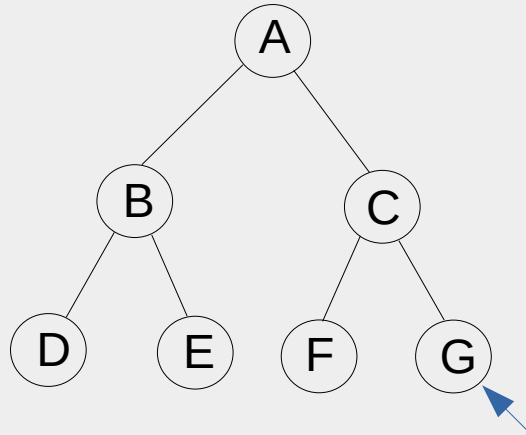
A B D / / E / / C F / / G

Exemplo:



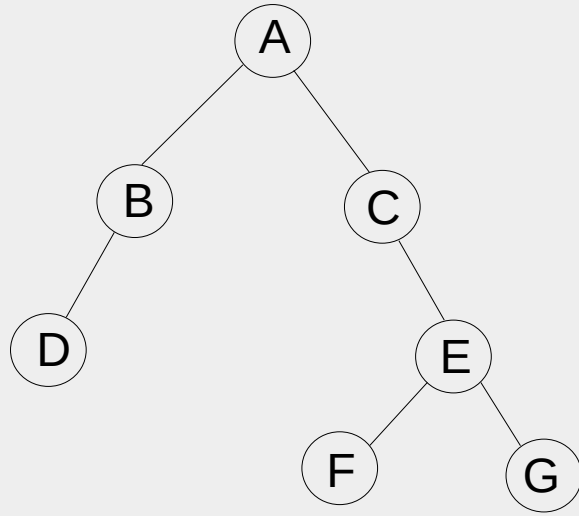
A B D / / E / / C F / / G /

Exemplo:

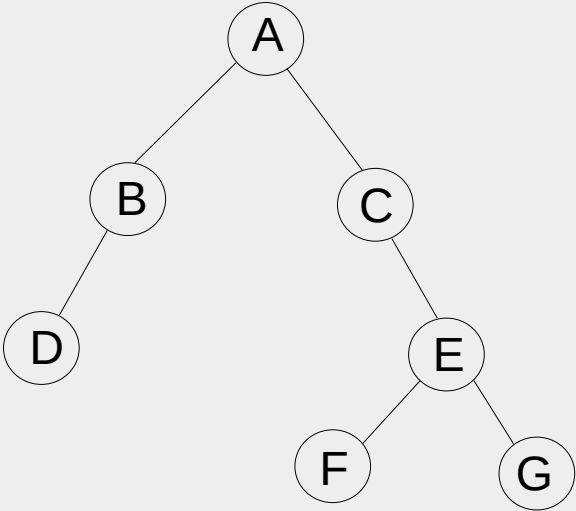


A B D / / E / / C F / / G / /

Exemplo 2:

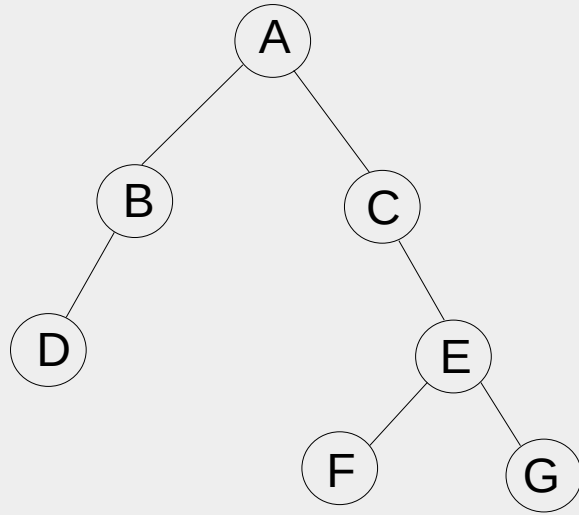


Exemplo 2:



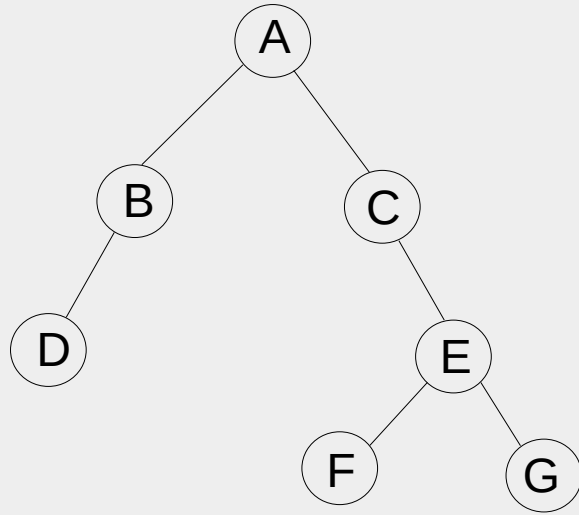
A

Exemplo 2:



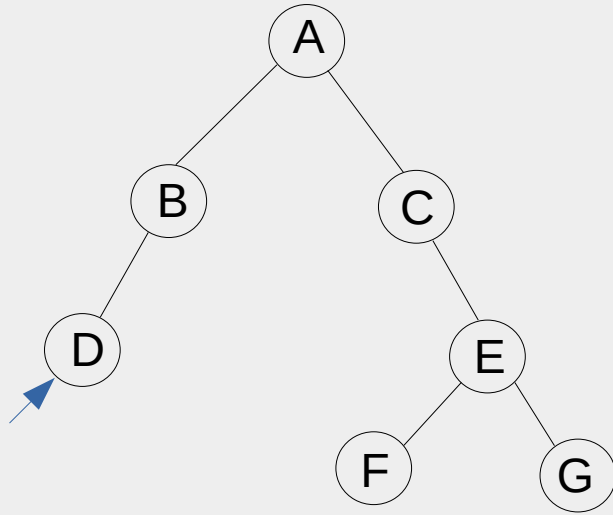
A B

Exemplo 2:



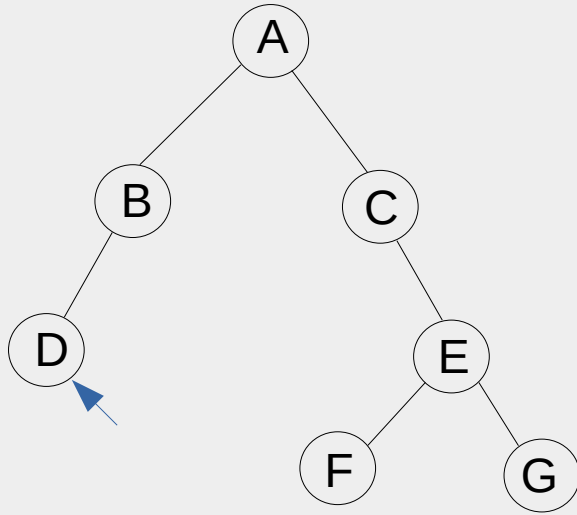
A B D

Exemplo 2:



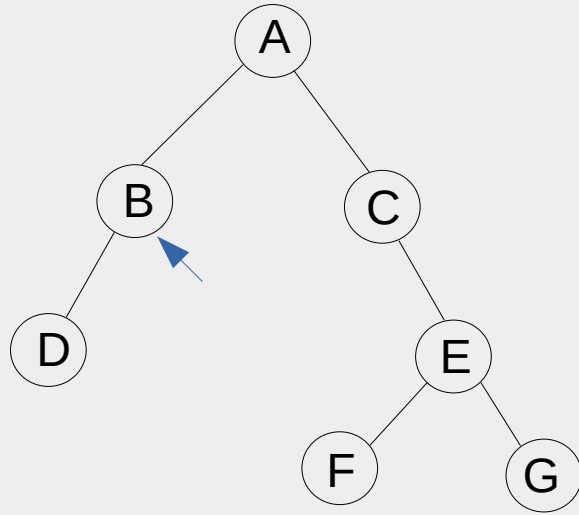
A B D /

Exemplo 2:



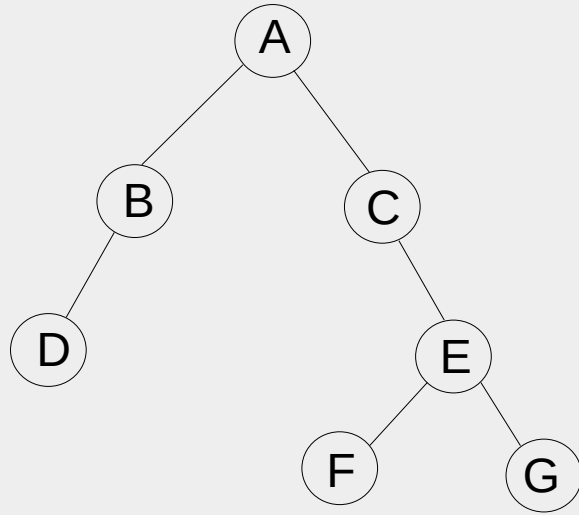
A B D / /

Exemplo 2:



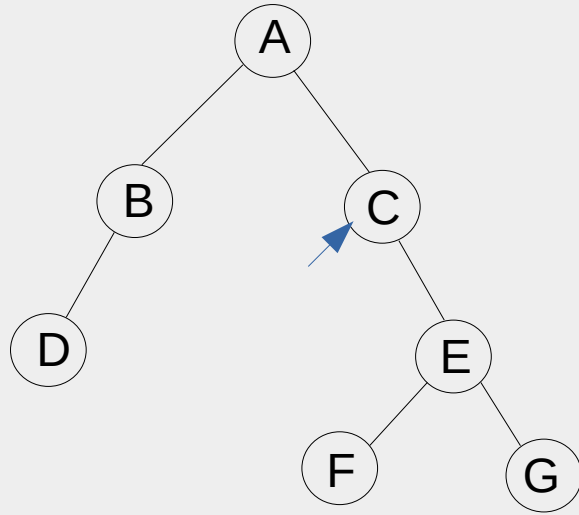
A B D / / /

Exemplo 2:



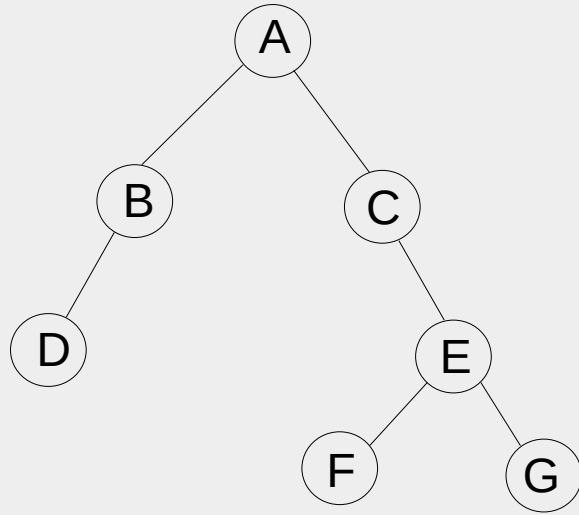
A B D / / / C

Exemplo 2:



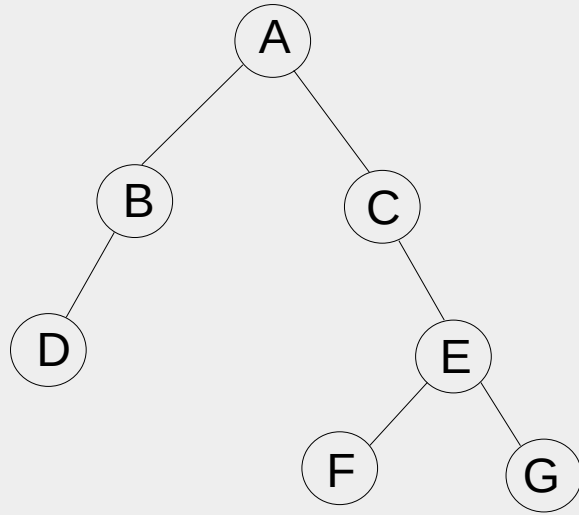
A B D / / / C /

Exemplo 2:



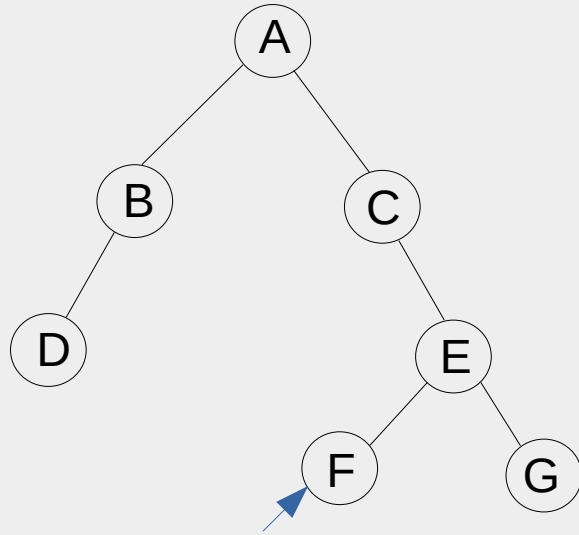
A B D / / / C / E

Exemplo 2:



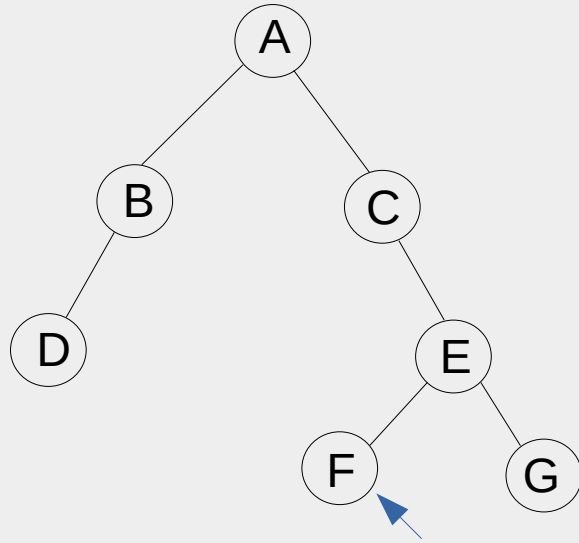
A B D / / / C / E F

Exemplo 2:



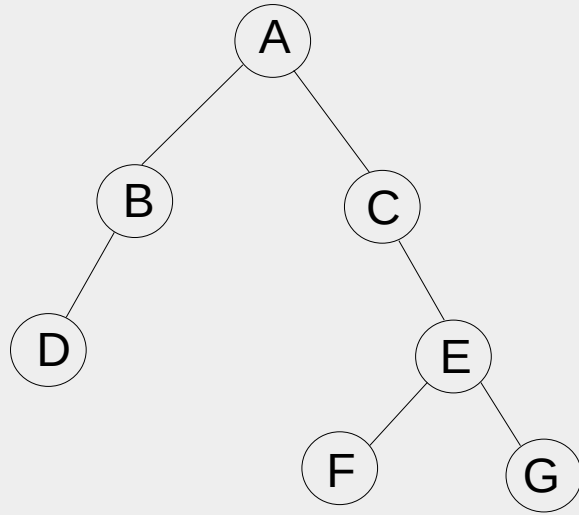
A B D / / / C / E F /

Exemplo 2:



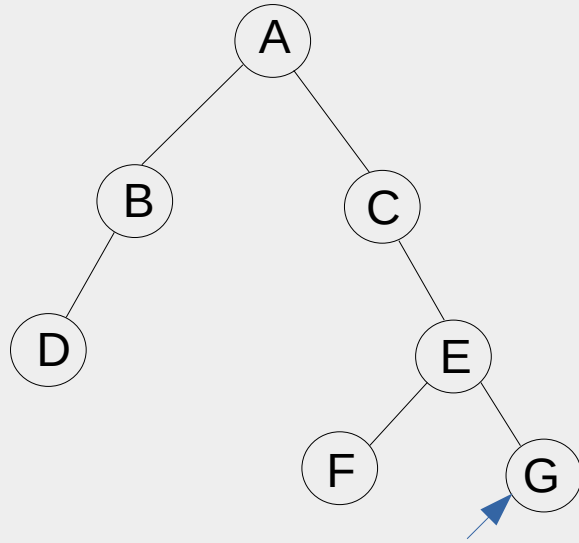
A B D / / / C / E F / /

Exemplo 2:



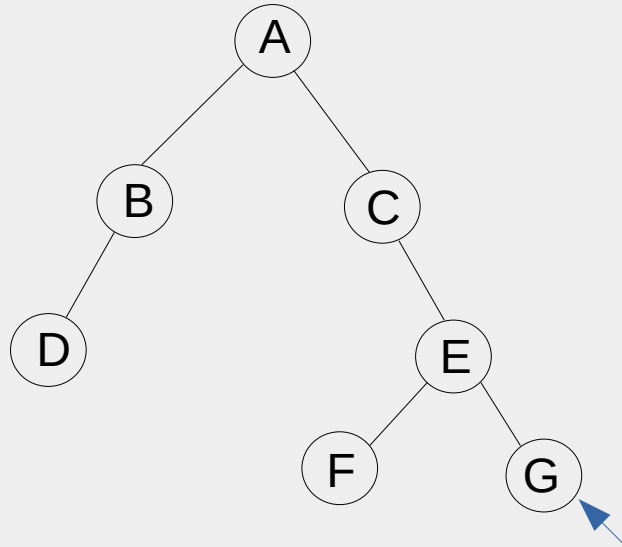
A B D / / / C / E F / / G

Exemplo 2:



A B D / / / C / E F / / G /

Exemplo 2:



A B D / / / C / E F / / G / /

Versão recursiva

```
void construir_arvore(ptNo & p ) {  
    tDado x;  
  
    cin >> x;  
    if (x == '/') {                // não construir nó  
        p = NULL;  
        return;  
    }  
    // alocar o nó e armazenar x  
    p = new NoArvBin;  
    p->dado = x;  
  
    construir_arvore(p->esq );  
  
    construir_arvore(p->dir);  
}
```


Versão recursiva

```
void construir_arvore(ptNo & p ) {  
    tDado x;  
  
    cin >> x;  
    if (dado == '/') {                // não construir nó  
        p = NULL;  
        return;  
    }  
    // alocar o nó e armazenar x  
    p = new NoArvBin;  
    p->dado = x;  
  
    cout << "Ins. à esquerda de " << p->dado << ". " << endl;  
    construir_arvore(p->esq );  
    cout << "Ins. à direita de " << p->dado << ". " << endl;  
    construir_arvore(p->dir);  
}
```

Procedimentos para a evolução da árvore:

- Inserção de um valor x em um filho à esquerda de determinado nó (com determinado conteúdo);

Procedimentos para a evolução da árvore:

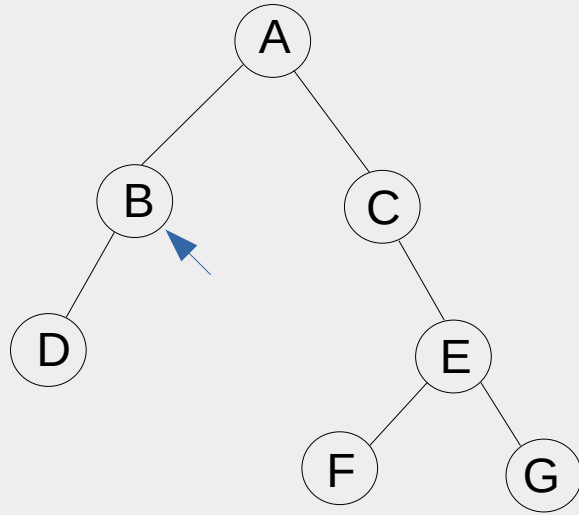
- Inserção de um valor x em um filho à esquerda de determinado nó (com determinado conteúdo);
- Inserção de um valor x em um filho à direita de determinado nó (com determinado conteúdo)

Procedimentos para a evolução da árvore:

- Inserção de um valor x em um filho à esquerda de determinado nó (com determinado conteúdo);
- Inserção de um valor x em um filho à direita de determinado nó (com determinado conteúdo)

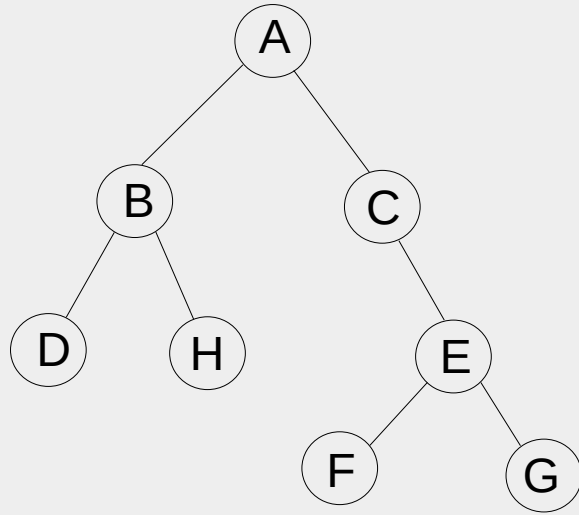
(são funções de retorno do tipo bool - se a tarefa foi realizada ou é impossível)

Exemplo:



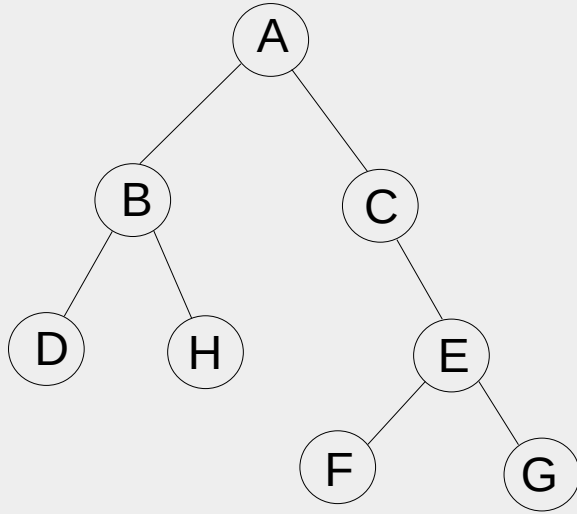
Inserir à direita de “B” o valor “H”

Exemplo:



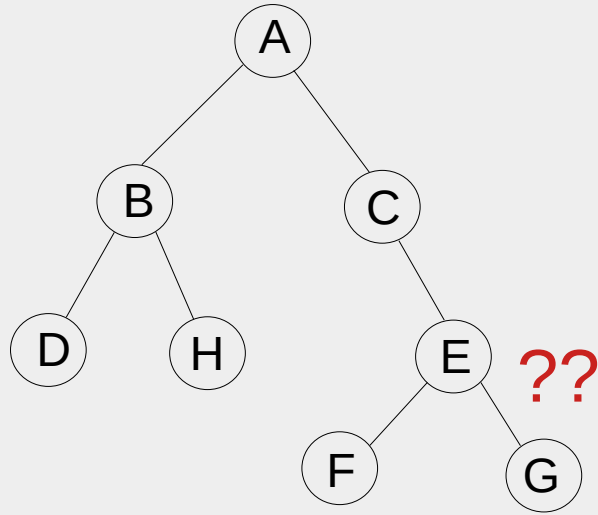
Inserir à direita de “B” o valor “H”

Exemplo:



Inserir à direita de “B” o valor “H”
Inserir à direita de “E” o valor “I”

Exemplo:

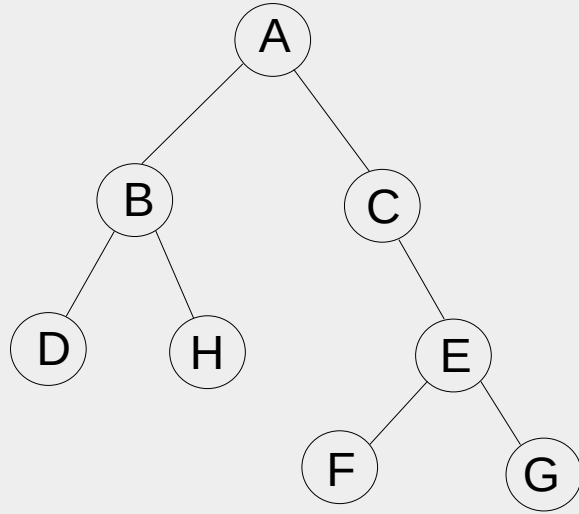


Inserir à direita de “B” o valor “H”

Inserir à direita de “E” o valor “I”

??

Exemplo:

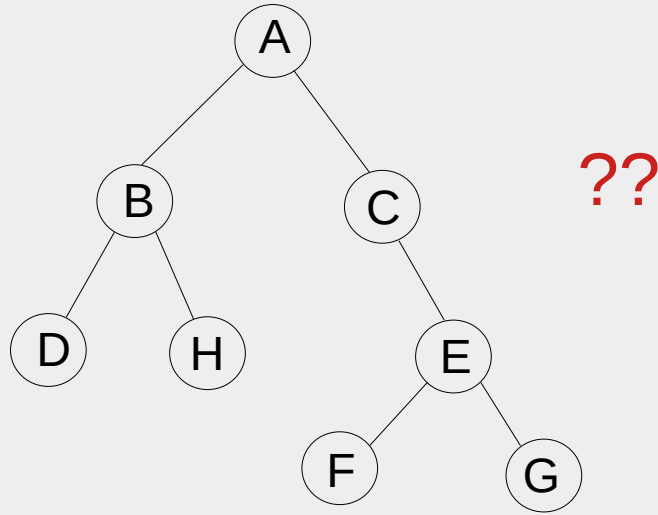


Inserir à direita de “B” o valor “H”

Inserir à direita de “E” o valor “I”

Inserir à direita de “K” o valor “W”

Exemplo:



Inserir à direita de “B” o valor “H”

Inserir à direita de “E” o valor “I”

Inserir à direita de “K” o valor “W”

??