ESTRUTURAS DE DADOS

prof. Alexandre César Muniz de Oliveira

Sugestão bibliográfica:

- **ESTRUTURAS DE DADOS USANDO C**

Aaron M. Tenenbaum, et alli

- **DATA STRUCTURES, AN ADVANCED APPROACH USING C**

Jeffrey Esakov & Tom Weiss

- **ESTRUTURAS DE DADOS E ALGORITMOS EM JAVA (2ED)**

Michael Godrich & Roberto Tamassia

ÁRVORES

1. Definição

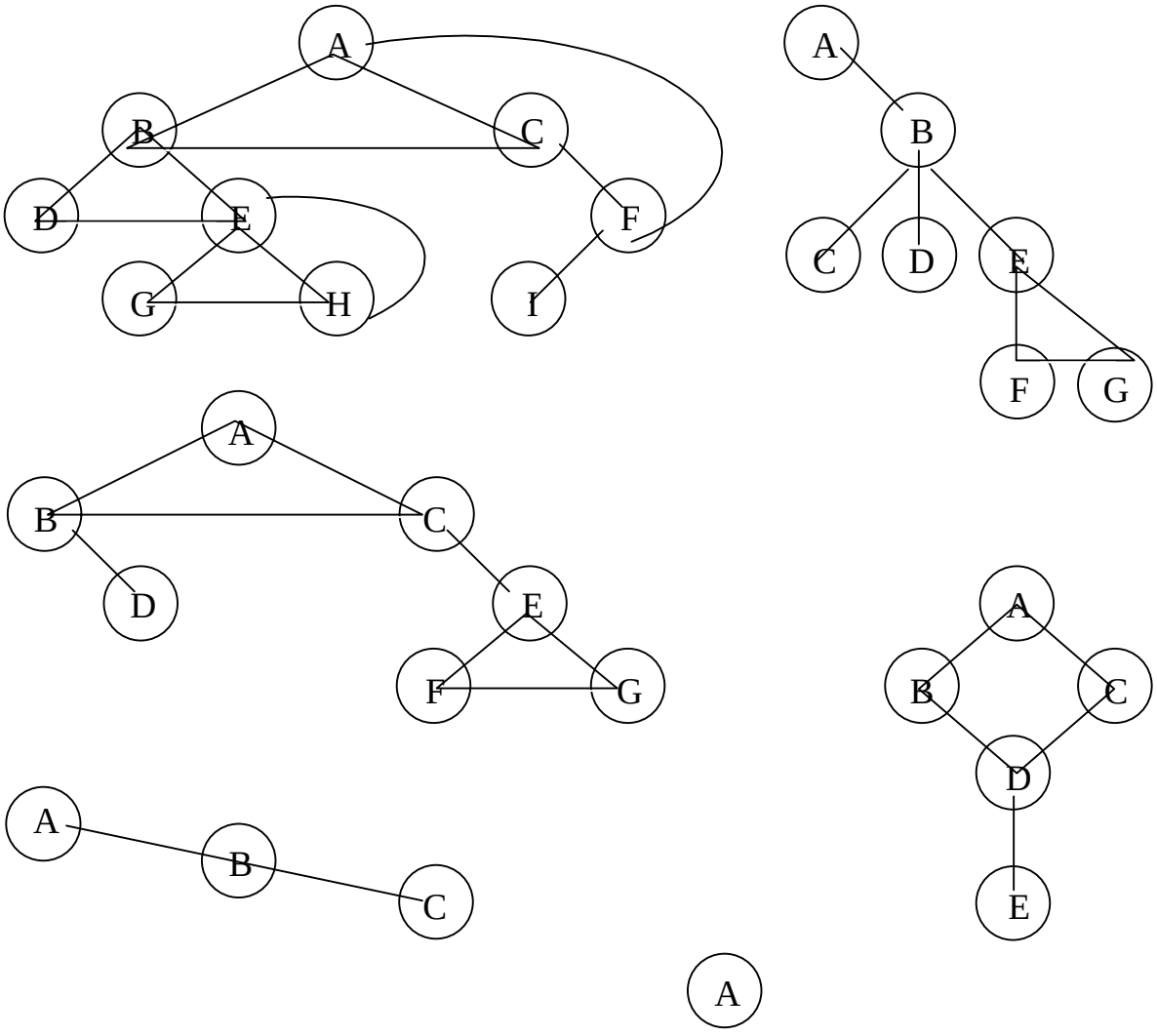
Estrutura de dados contendo um número finito de elementos que pode estar vazia ou particionanda em subconjuntos. O primeiro subconjunto contém um único elemento chamado raiz. Os outros subconjuntos são, em si mesmo árvores, chamados subárvores. Essa é uma definição recursive de árvore:

Árvores → raiz + Árvores

raiz + Árvores

●●●

EXEMPLOS:



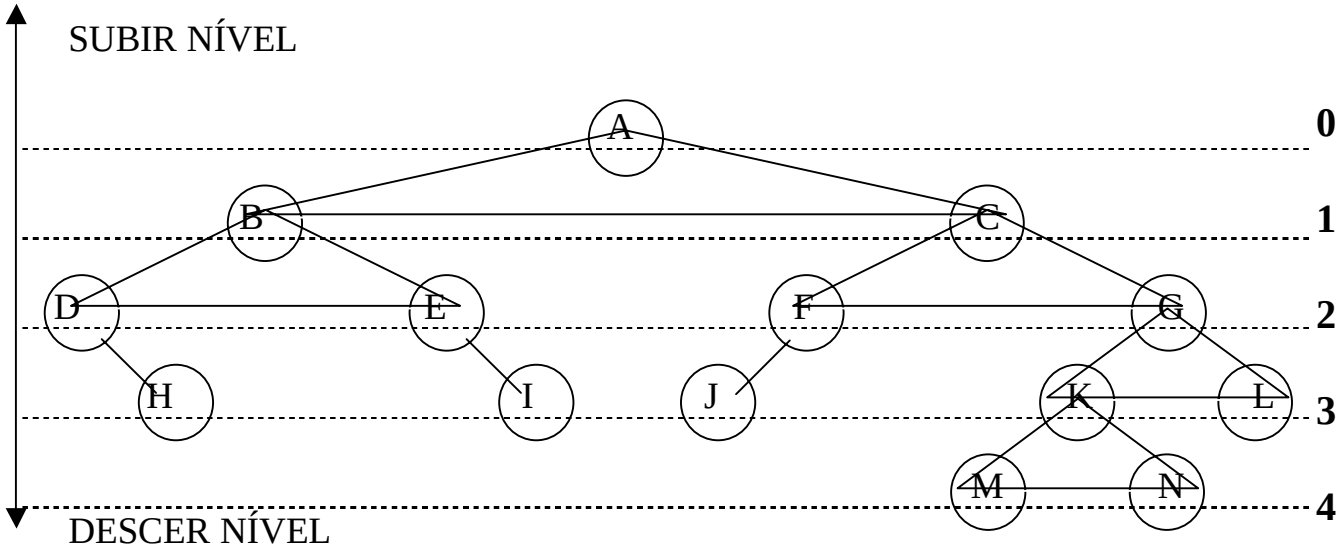
ÁRVORES BINÁRIAS

2. Árvores Binárias

Cada elemento está particionado em 3 subconjuntos distintos:

Árvore : = [RAIZ + <sub_árvore_esq> + <sub_árvore_dir>] |
[VAZIA]
sub_árvore_esq := <Árvore>
sub_árvore_dir := <Árvore>

3. Conceitos



a) Pai e filho:
"A" é pai de "B" e "C"
"J"é filho de "F"

b) Ancestral e descendente
"M"é descendente de "G"
"B" é ancestral de "I"

c) Folha é um nó sem filhos

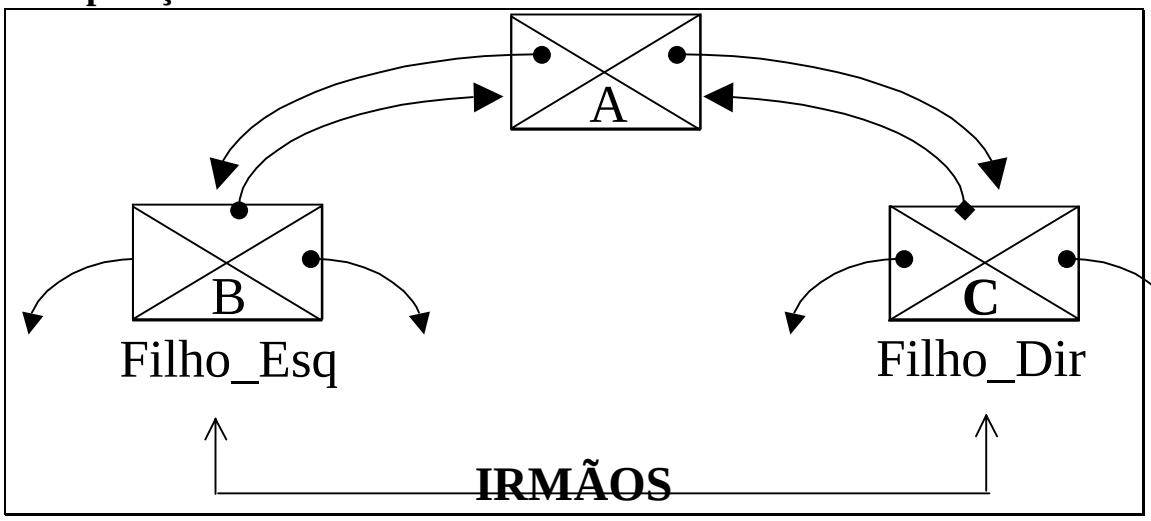
d) Árvore estritamente binária:
Árvore cujos nós não-folha tem dois filhos
A árvore do exemplo não é estritamente binária

e) Nível e profundidade
Nível 0 = raiz
Profundidade é o maior nível encontrado em uma árvore

f) Árvore completa:
Árvore estritamente binária com folhas no mesmo nível
A árvore do exemplo, por não ser estritamente binária também não é completa

ÁRVORES BINÁRIAS

4. Operações sobre Árvores Binárias



	ESQ (p)		**IRMAO (p)**
COM	**DIR (p)**	**GERAR:**	**EH_ESQ (p)**
	PAI (p)		**EH_DIR (p)**

ROTINAS

```
EH_ESQ (p)
{
    q = PAI (p);
    if (q == NULL)
        return FALSE;
    if (ESQ (q) == p)
        return TRUE;
    return FALSE;
}
```

```
IRMAO (p)
{
    if (PAI (p) == NULL)
        return FALSE;
    if (EH_ESQ (p))
        return (DIR(PAI(p)));
    else
        return (ESQ(PAI(p)));
}
```

Operações para criar um nó:

CRIA_ARVORE (x)
CRIA_ESQUERDA (p, x) } **CRIA_NO (p, x)**
CRIA_DIREITA (p, x)

ÁRVORES BINÁRIAS

5. Tipos de Percurso

- ✓ Por ordem de visita
- ✓ Por nível

5.1 Percurso por ordem de visita

Operações básicas em árvores binárias

Visitar um nó	VISITA(no)
Percorrer subárvore à esquerda	PERCORRE(no_esq)
Percorrer subárvore à esquerda	PERCORRE(no_dir)

A ordem em que essas operações são efetuadas em cada nó define o tipo de percurso:

Pré-ordem	ou	profundidade	VED
Em ordem	ou	simétrica	EVD
Pós-ordem	ou	largura	EDV

5.1.1 Percurso em Pré-ordem

1) Visita (no)
2) Se existe esq (no) percorre_pre (esq (no))
3) Se existe dir (no) percorre_pre (dir (no))

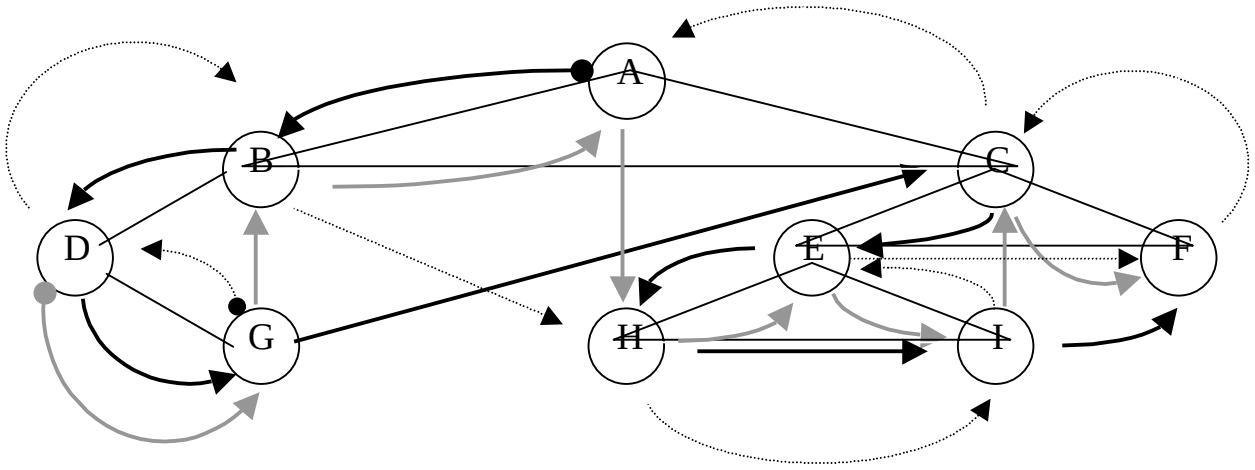
5.1.2 Percurso em Ordem

1) Se existe esq (no) percorre_em (esq (no))
2) Visita (no)
3) Se existe dir (no) percorre_em (dir (no))

5.1.3 Percurso em Pós-ordem

1) Se existe esq (no) percorre_pos (esq (no))
2) Se existe dir (no) percorre_pos (dir (no))
3) Visita (no)

ÁRVORES BINÁRIAS



Esquema gráfico de cada um dos percursos

Pré-ordem (profundidade) :	A B D G C E H I F
Ordem (simétrica):	D G B A H E I C F
Pós-ordem (largura):	G D B H I E F C A

5.2 Percurso por nível

1) Visita a raiz (NÍVEL 0)
2) Visita todos os nós de NÍVEL 1
3) Visita todos os nós de NÍVEL 2
...
n) Visita todos os nós de NÍVEL (n – 1)

...

```
insere (fila, raiz);
enquanto (não vazia(fila))
{
  no := remove (fila);
  visita (no);
  if (esq (no) existe) insere (fila, esq (no));
  if (dir (no) existe) insere (fila, dir (no));
}
```

Obs.: aplicações com árvores, em geral, consiste em duas fases:

- construir a árvore baseado em algum algoritmo
- percorrer a árvore usando um tipo de percurso

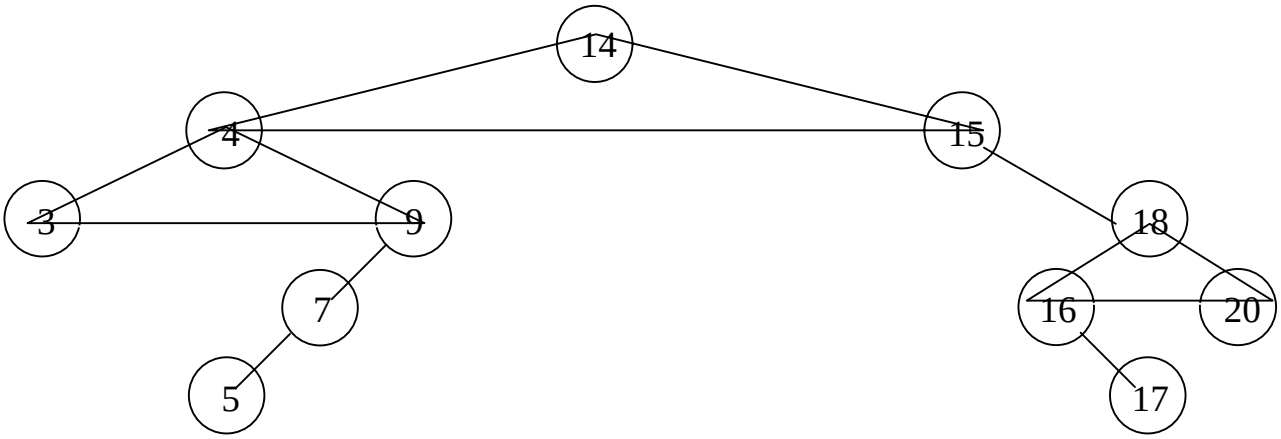
ÁRVORES BINÁRIAS

6. Exemplo de aplicações

6.1 Ordenação

Dada a sequência {14, 15, 4, 9, 7, 18, 3, 5, 16, 4, 20, 17, 9, 14, 5}

CONSTRUÇÃO INTERATIVA	CONSTRUÇÃO RECURSIVA
num := le_numero ( ); *raiz := cria_arvore (num);* *Enquanto (há números na entrada)* *{* *num := le_numero;* *p = q = free;* *enquanto (q ≠ NULL) e (num ≠ info (p));* *{* *p = q;* *if (num < info (p))* *q := esq (p);* *else* *q := dir (p);* *}* *if (num = info (p))* *imprime_aviso (num);* *else if (num < info (p))* *cria_esquerda(p, num)* *else* *cria_direita(p, num);* *}*	*Num = Le_num ( );* *Enquanto (há números na entrada)* *{* *Insere_Arvore (root, num);* *Num = Le_num ( );* *}* *Insere_Arvore (raiz, x)* *{* *Se (raiz == NULO) Cria_no (raiz, x);* *Senao* *{* *Se (x < info)* *Se (esq(raiz) != NULO)* *Insere_Arvore (esq, x)* *Senao* *Cria_no(esq(raiz), x);* *Senao* *Se (dir(raiz) != NULO)* *Insere_Arvore (dir(raiz), x)* *Senao* *Cria_no(dir(raiz), x);* *}* *return;* *}*

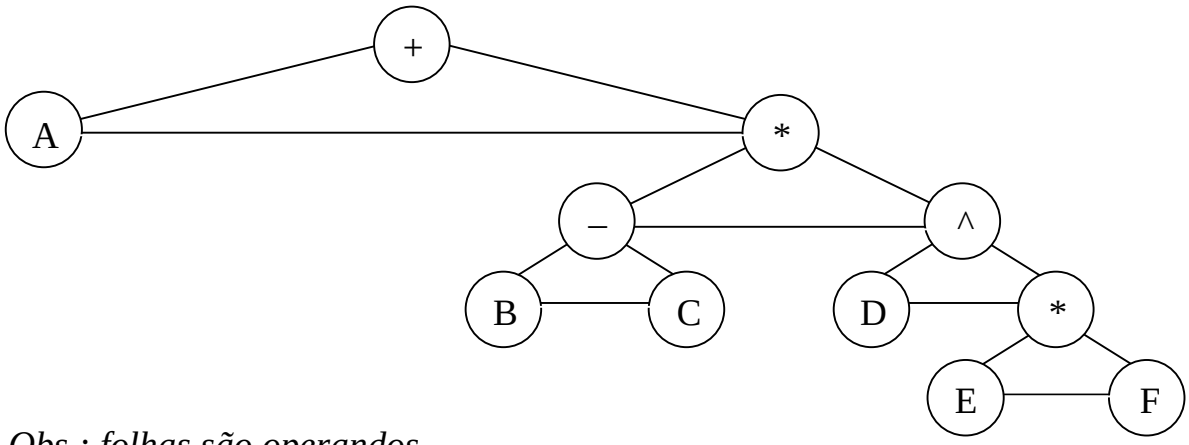


ÁRVORES BINÁRIAS

6.2 Expressões Aritméticas

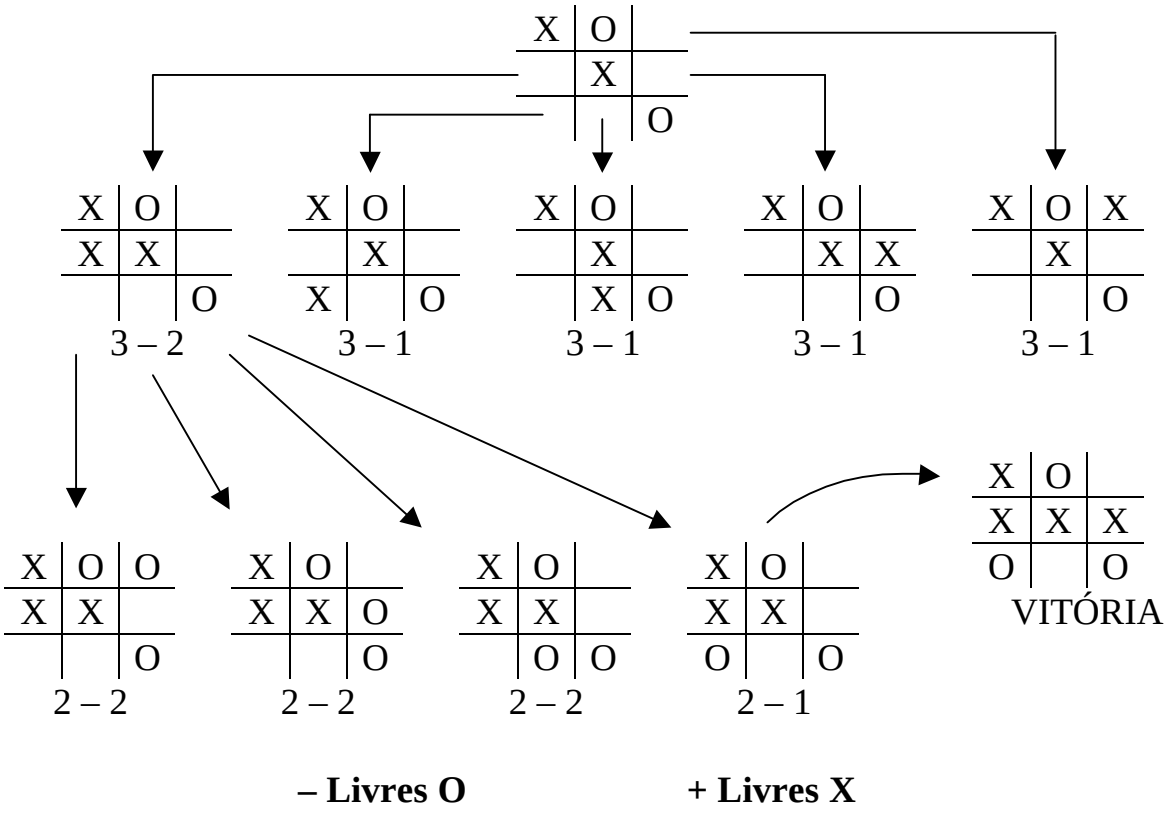
A + (B – C) * D ^ (E * F)

PRÉ-ORDEM: +A*–BC^D*EF
PÓS-ORDEM: ABC–DEF*^*+



Obs.: folhas são operandos

6.3 Árvore de Jogos



– Livres O **+ Livres X**

ÁRVORES BINÁRIAS

7 Implementação de Árvores Binárias

```
Struct noarvore
{
  void * info;
  struct noarvore * esquerda;
  struct noarvore * direita;
  struct noarvore * pai;
};
typedef struct noarvore *PTRNOARV;
# define info (p)        ( (p)→ info)
# define left (p)        ( (p)→ esquerda)
# define right (p)       ( (p)→ direita)
# define father (p)      ( (p)→ pai)
aloca_arvore ( )
libera_arvore (PTRNOARV p)
```

Admitindo-se os percursos por ordem de visita e percurso por nível (percurso de cima p/ baixo) é desnecessário o campo PAI.

IMPLEMENTAÇÃO

```
PTRNOARV Cria_Arvore (void * x)
{
  p = aloca_arvore ( );
  if (p == NULL) return NULL;
  info (p) = x;
  left (p) = right (p) = NULL;
  return p;
}
```

```
Cria_No (PTRNOARV * raiz, void * x)
{
  if (*raiz !=NULL) return FALSE;   /* já alocado
  * raiz = Aloca_Arvore ( );
  if (*raiz ==NULL) return FALSE;
  info (*raiz) = x;
  left (*raiz) = right (*raiz) = NULL;
}  return TRUE;
```

```
Cria_Esquerda (PTRNOARV raiz, void * x)
{
  if (raiz == NULL) and left(raiz) != NULL)
     return FALSE;
  left(raiz) = Cria_Arvore (x);
}
```

```
Cria_Direita (PTRNOARV raiz, void * x);
```

ÁRVORES BINÁRIAS

IMPLEMENTAÇÃO DE PERCURSO EM ÁRVORE BINÁRIA

O percurso por ordem de visita pode ser implementado de forma interativa:

Enquanto *(existir no_esquerdo)*
P = left (p);
Visita (p);
Enquanto *(existir no_direito)*

- Não vai funcionar: necessário salvar o contexto para se saber a última operação ou a próxima operação a ser realizada.
- SUGESTÃO: Usar uma pilha de ponteiros para nós de árvores, apontando para o próximo nó a ser seguido
- DESVANTAGENS:
  - **lento** (muitos PUSHs e POPs)
  - **complexo**
  - **deselegante**
- SUGESTÃO: Recursividade, pois para cada nó, as operações se repetem e chamadas recursivas, automaticamente, salvam o contexto.

IMPLEMENTAÇÃO DE PERCURSO EM ÁRVORE BINÁRIA

```
Pre_ordem (PTRNOARV raiz)
{
  if (raiz != NULL)
  { Visita(raiz);
    Pre_ordem(left(raiz));
    Pre_ordem(right(raiz));
  }
}
```

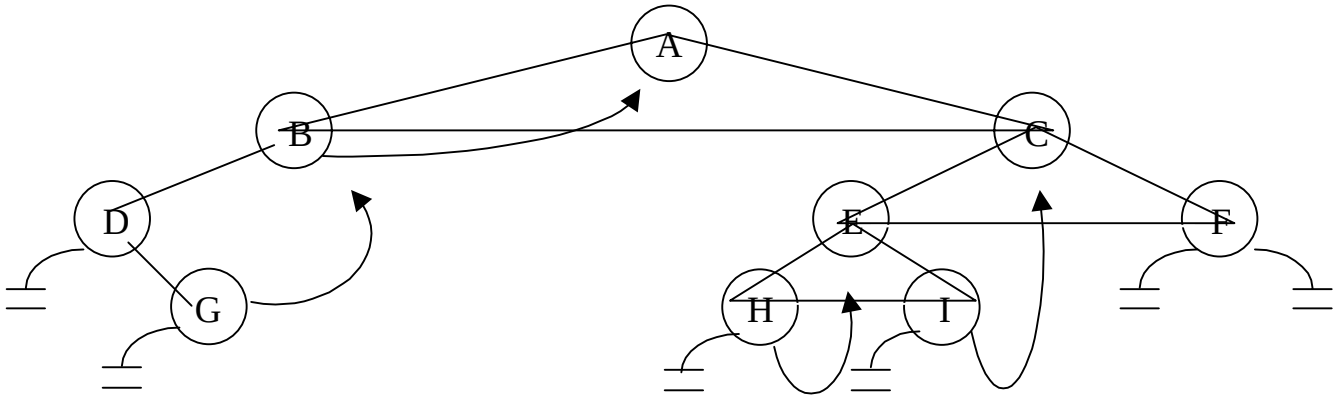
```
Em_ordem (PTRNOARV raiz)
{
  if (raiz != NULL)
  { Em_ordem(left(raiz));
    Visita(raiz);
    Em_ordem(right(raiz));
  }
}
```

```
Pos_ordem (PTRNOARV raiz)
{  if (raiz != NULL)
  { Pos_ordem(left(raiz));
    Pos_ordem(right(raiz));
    Visita(raiz);
  }
}
```

ÁRVORES BINÁRIAS

8 Árvores Binárias Encadeadas

Implementação de Percursos { Usando elo pai
Usando pilha
Recursivo
Usando fila

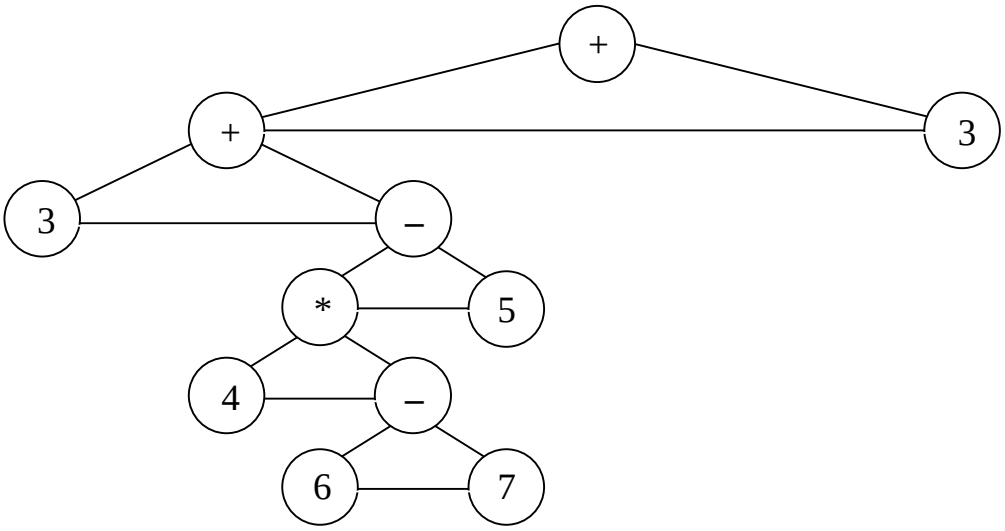



```
struct noarvore
{
  void *info;
  struct noarvore *esquerdo;
  struct noarvore *direito;
  char flag;
}
```

ponteiro à direita é ou não elo

ÁRVORES BINÁRIAS

9. Árvores Binárias Heterogêneas



- Como distinguir operando de operador?
- Como armazenar diferentes tipos de dados no mesmo tipo de árvore?

```
Struct noarvore
{
  unsigned utipo;
  union
  {
    char chinfo;
    float flinfo;
  } info;
  struct noarvore *esquerdo;
  struct noarvore *direito;
}

#define operador(x) ( (x) → utipo = = 1)
#define operando(x) ( (x) → utipo = = 0)
```

ÁRVORES BINÁRIAS

Algoritmo de Avaliação

```
Float avaliarvore (PTRNOARV raiz)
{
  float   op1, op2, resp;
  char    simb;
  if (operando(raiz))
    return (info(raiz));
  op1 := avaliarvore(esq(raiz));
  op2 := avaliarvore(dir(raiz));
  simb:= info(raiz);
  operacao(op1, simb, op2, &resp) ;
  return resp;
}
```

Algoritmo de Construção **?**

10. Algoritmo de Huffman

Problema de compactação p/ transmissões

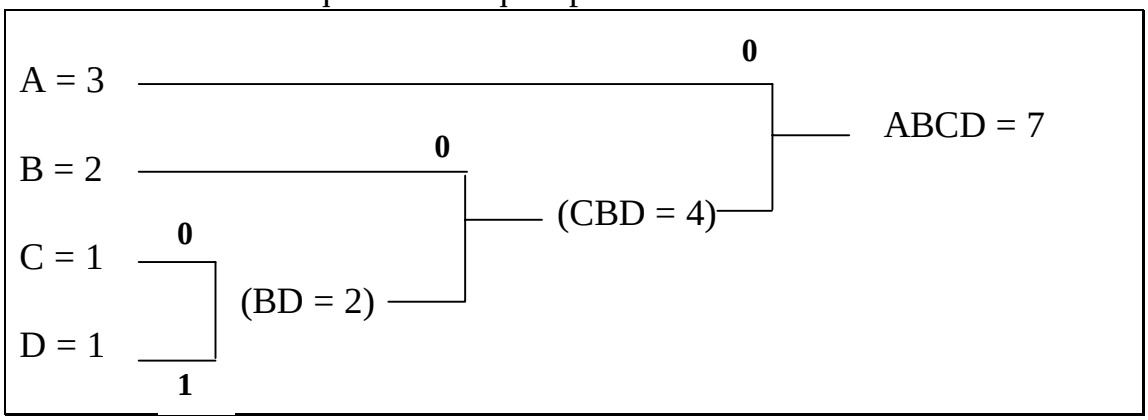
Símbolo	Código
A	**00**
B	**01**
C	**10**
D	**11**

Mensagem: ABACCDA

00010010101100

Comprimento: 14 bits

Observando a freqüência em que aparecem os símbolos temos:



ÁRVORES BINÁRIAS

TABELA COM CÓDIGOS DE TAMANHO VARIÁVEL

Símbolo	Código
A	0
B	110
C	10
D	111

Mensagem: ABACCDA

0110010101110

Comprimento: 13 bits

- Como calcular a freqüência?
- Vantagens para grandes volumes

Exemplo: A = 15 B = 6 C = 7 D = 12 E = 25 F = 4 G = 6 H = 1

PASSOS :

- Ordenação (lista classificada por freqüência)
- Geração de árvore de menor para maior freqüência
- Geração de tabela de código de tamanho variável
- Algoritmo de compactação