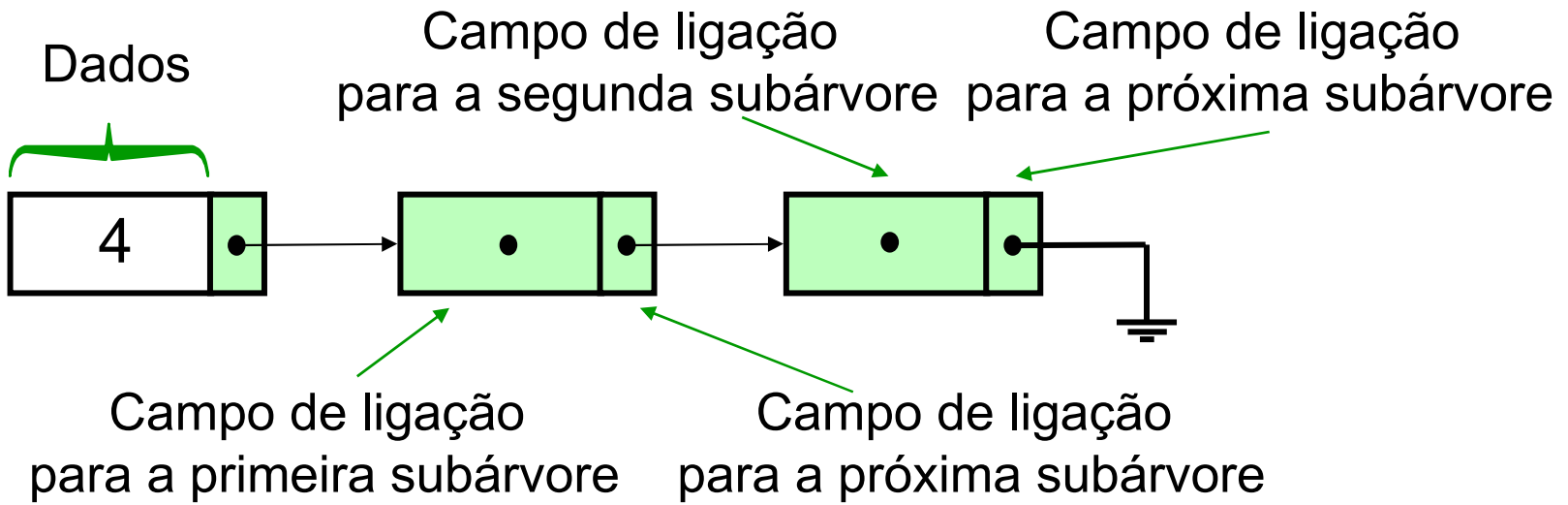
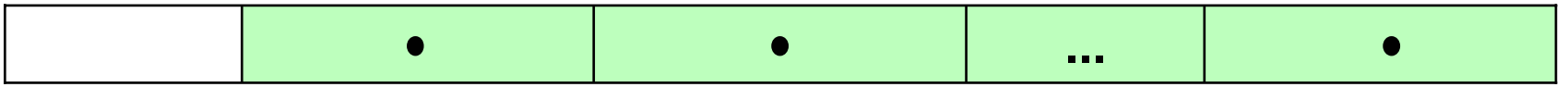
Implementação de Árvores

- Árvores podem ser implementadas utilizando listas encadeadas
  - Cada nó possui um campo de informação e uma série de campos de ligação, de acordo como número de filhos daquele nó



Implementação de Árvores

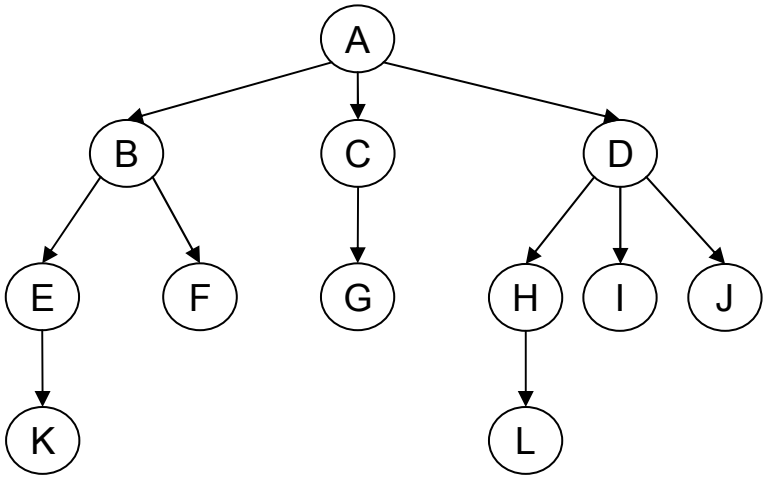
- Entretanto, é mais simples o caso em que cada nó tem um número máximo de filhos *d* pré-estabelecido



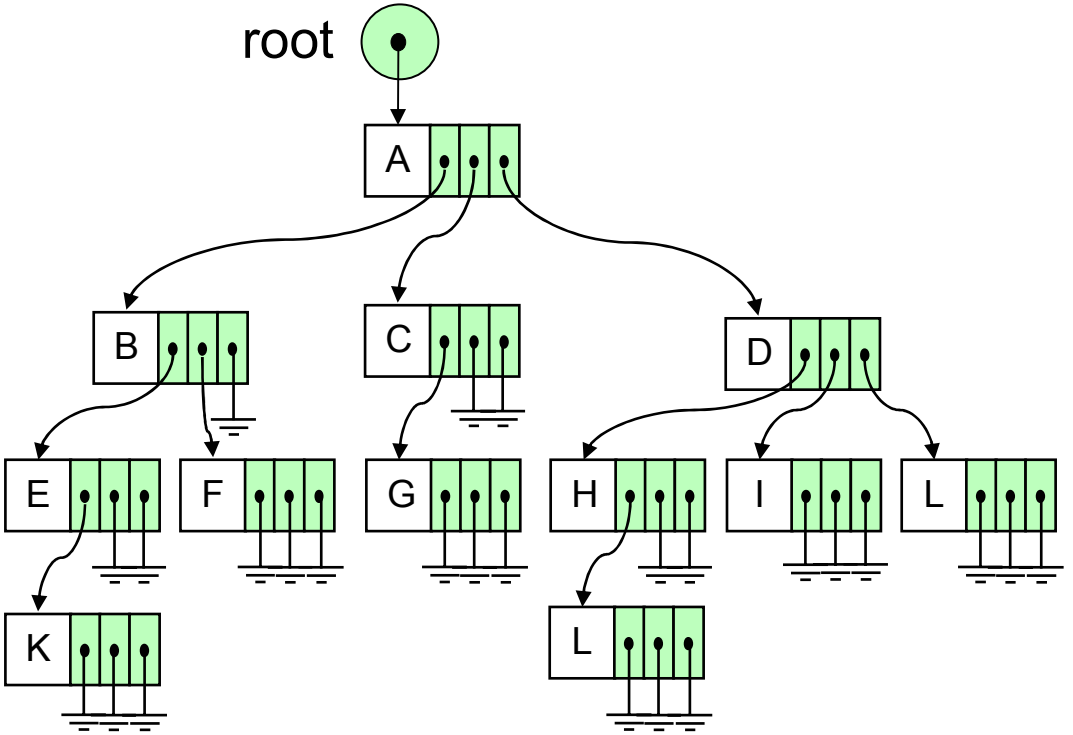
Dados	Campo de ligação para a primeira subárvore	Campo de ligação para a segunda subárvore		Campo de ligação para a *d*-ésima subárvore

Implementação de Árvores

- Por exemplo, a árvore ternária seguinte (d=3)...

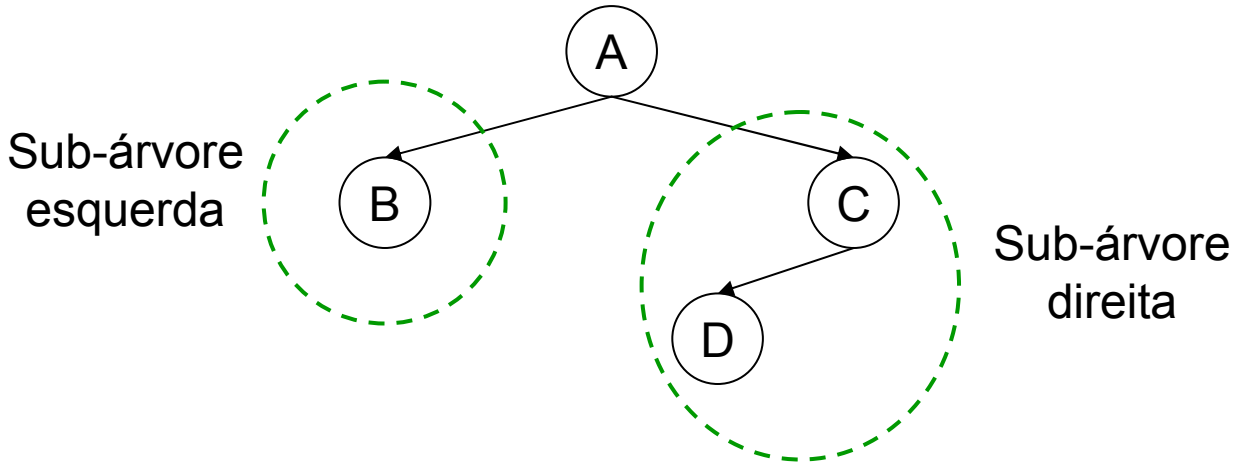


- ... pode ser implementada como



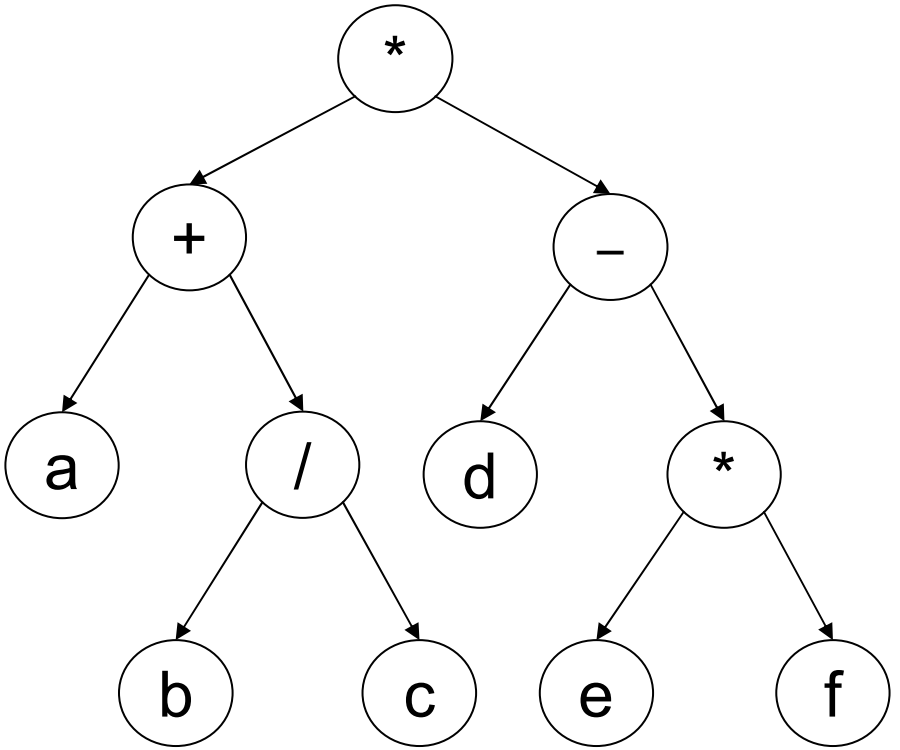
Árvores Binárias (AB)

- **Árvores binárias** são árvores orientadas de grau 2
- Uma árvore binária é uma estrutura que é ou vazia ou possui 3 componentes:
  - Uma raiz
  - Uma subárvore esquerda
  - Uma subárvore direita
- As subárvores devem ser árvores binárias



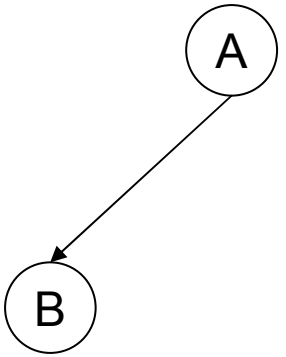
Árvores Binárias

- Podemos, por exemplo, representar uma expressão aritmética (com operadores binários) por meio de uma AB, na qual cada operador é um nó da árvore e seus dois operandos representados como subárvores
- A árvore ao lado representa a expressão (a+b/c)*(d-e*f)

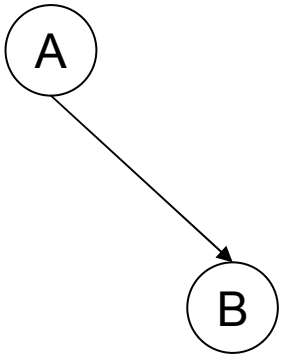


Árvores Binárias

- As duas AB seguintes são distintas
  - (i) a primeira tem subárvore direita vazia
  - (ii) a segunda tem subárvore esquerda vazia



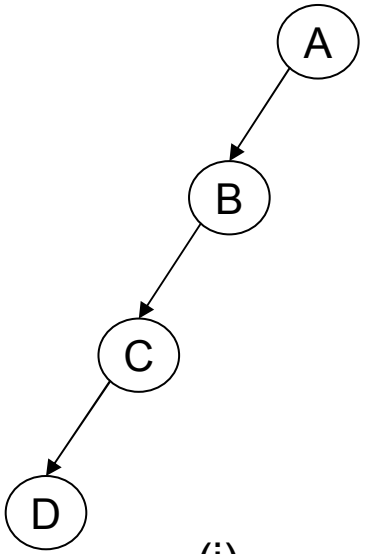
(i)



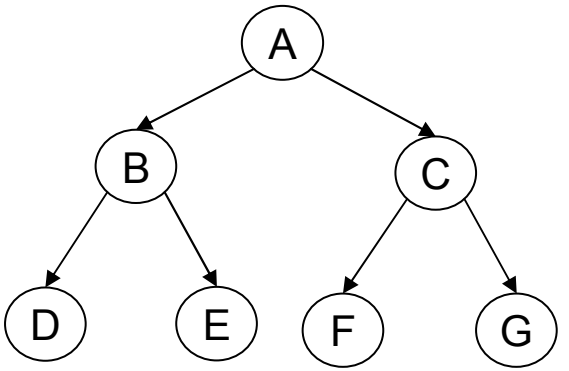
(ii)

Árvores Binárias

- Exemplos de AB
- (i) assimétrica à esquerda (degenerada)
- (ii) completa



(i)



(ii)

Árvores Binárias

- O número de máximo de nós em uma árvore binária de altura **h** é dado por:

$$n = n(h,2) = \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$$

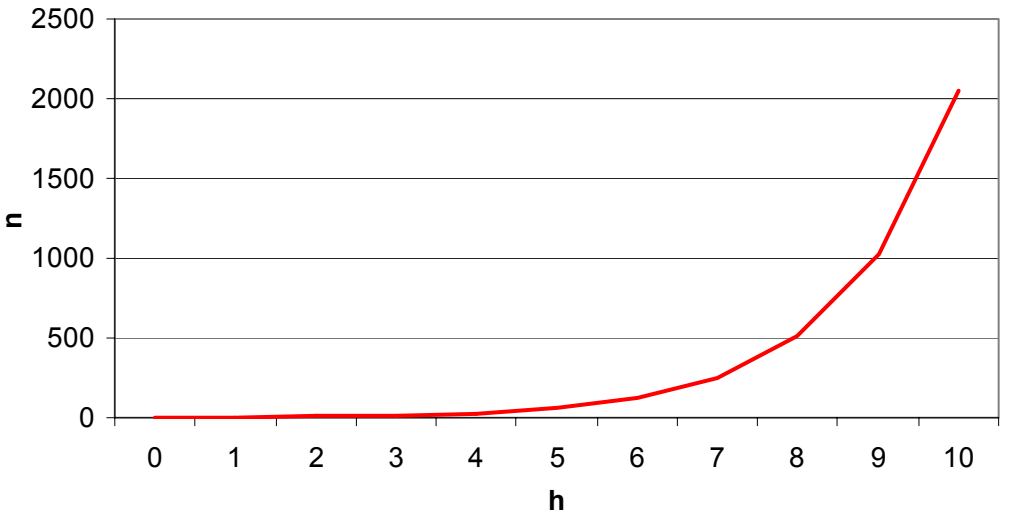
- Portanto, **n** elementos podem ser organizados em uma árvore binária de altura mínima ≈ $\log_2$ **n**

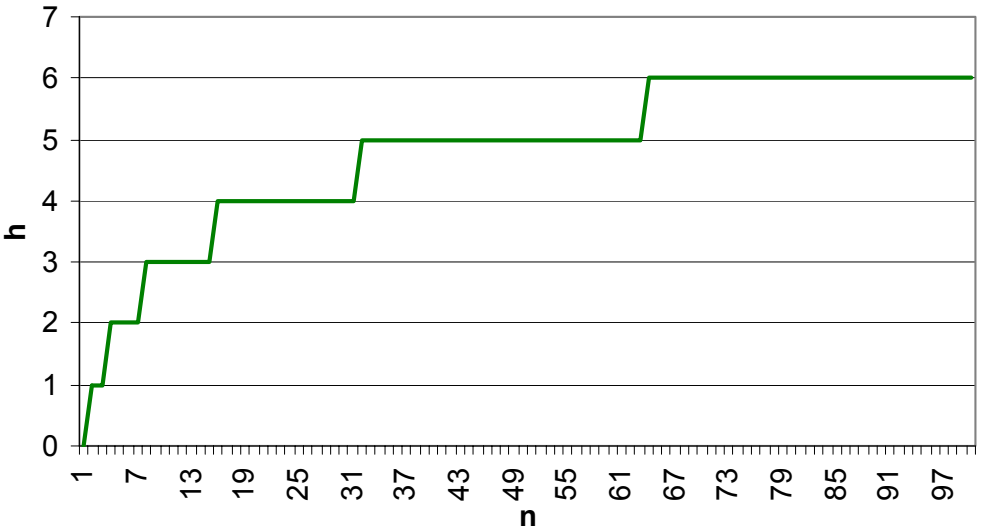
$$h = \lfloor \log_2 (n+1) - 1 \rfloor$$

Árvores Binárias 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

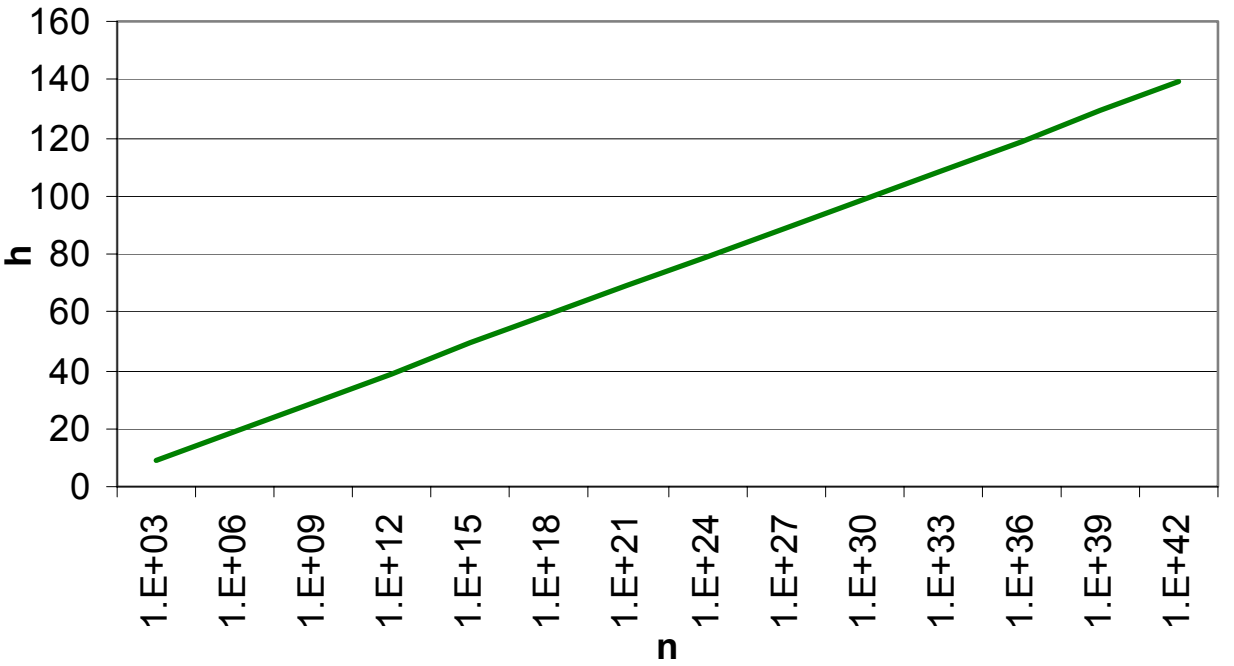
n	h
1	0
2	1
3	1
4	2
5	2
6	2
7	2
8	3
9	3
10	3
11	3
12	3
13	3
14	3
15	3
16	4





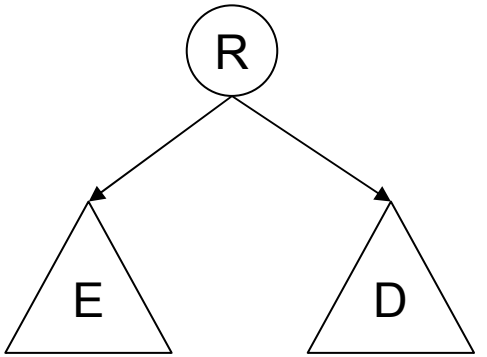
Árvores Binárias de Altura Mínima

n	h
1.E+03	9
1.E+06	19
1.E+09	29
1.E+12	39
1.E+15	49
1.E+18	59
1.E+21	69
1.E+24	79
1.E+27	89
1.E+30	99
1.E+33	109
1.E+36	119
1.E+39	129
1.E+42	139



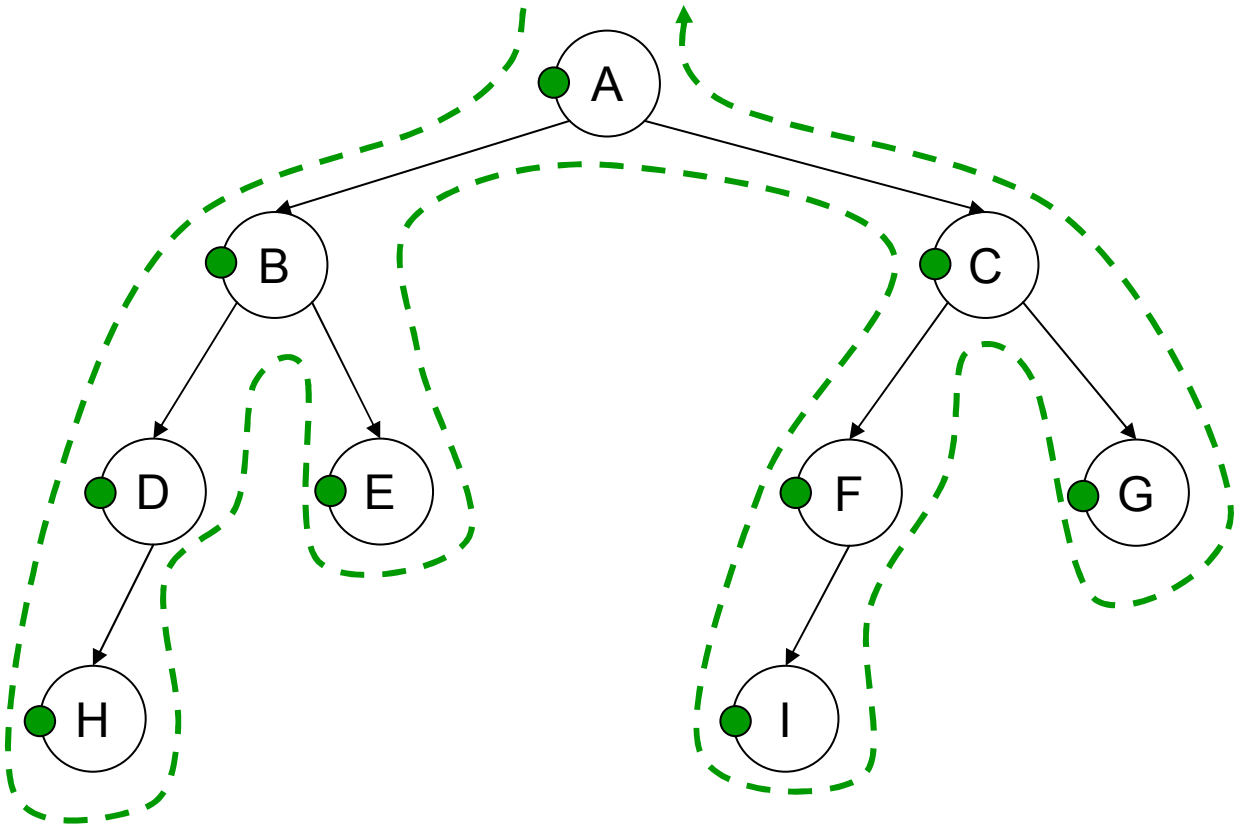
Percurso em AB

- Seja uma AB em que **R** denota sua raiz, **E** e **D** denotam as subárvores esquerda e direita, respectivamente
- Os nós de uma AB podem ser visitados de três formas (**varredura** da árvore):
  - Pré-ordem (*pre-order*): **R**, **E**, **D**
    - visitar a raiz antes das subárvores
  - Em-ordem (*in-order*): **E**, **R**, **D**
  - Pós-ordem (*post-order*): **E**, **D**, **R**
    - visitar a raiz após visitar as subárvores



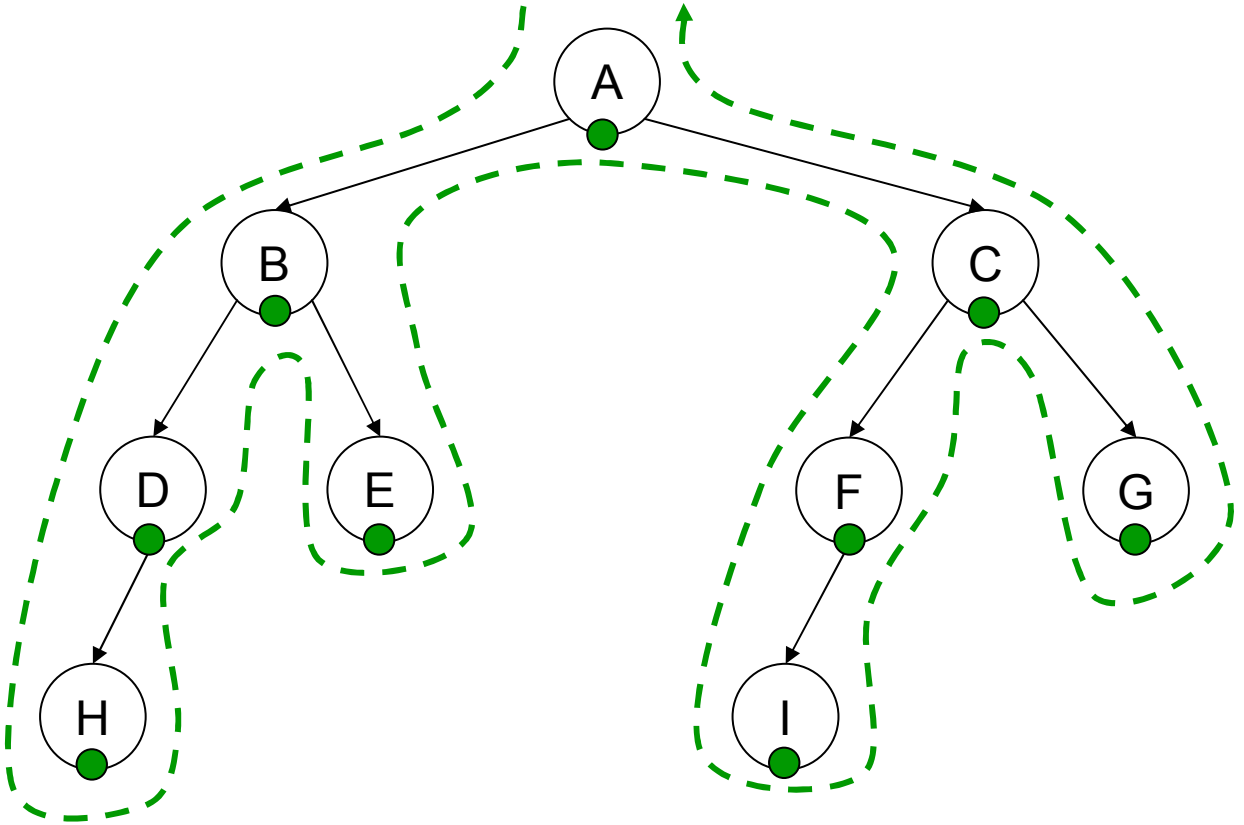
Percurso em AB

- Pré-ordem: A, B, D, H, E, C, F, I, G



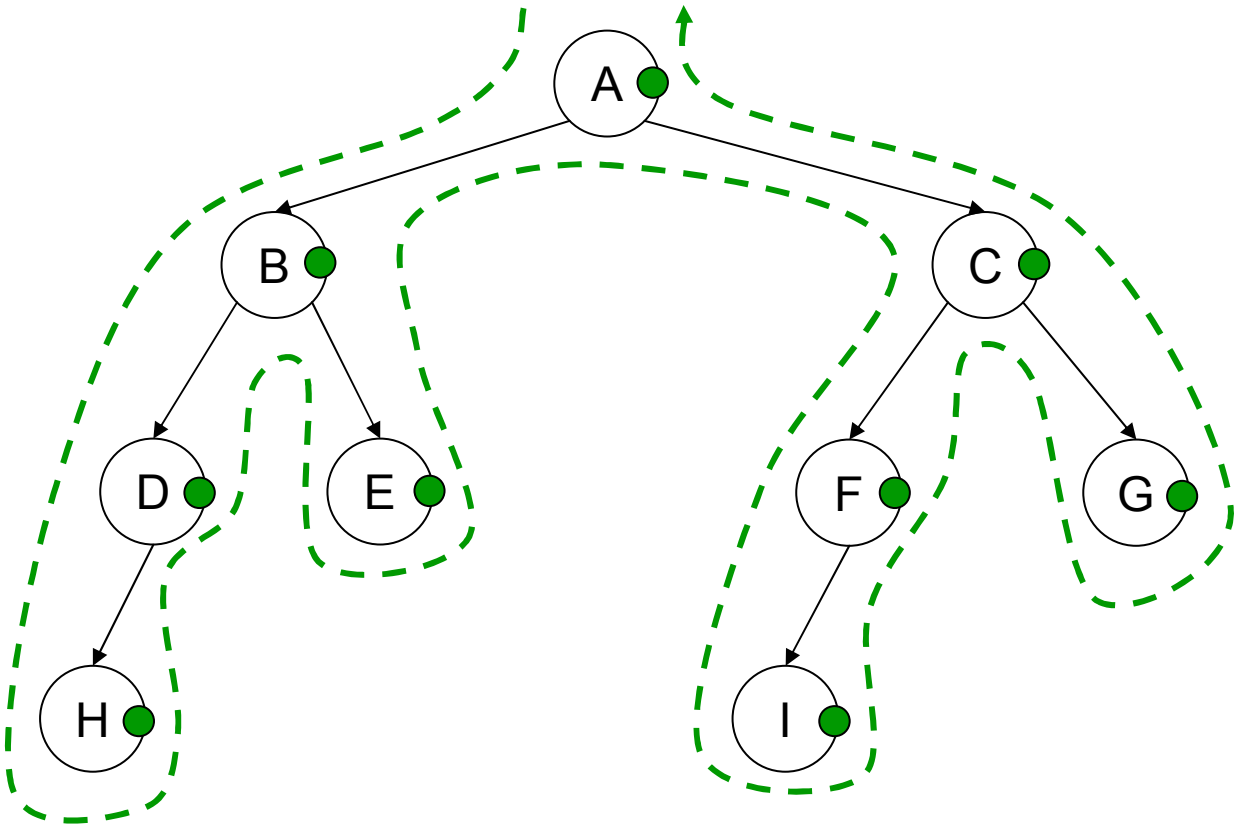
Percurso em AB

- Em-ordem: H, D, B, E, A, I, F, C, G



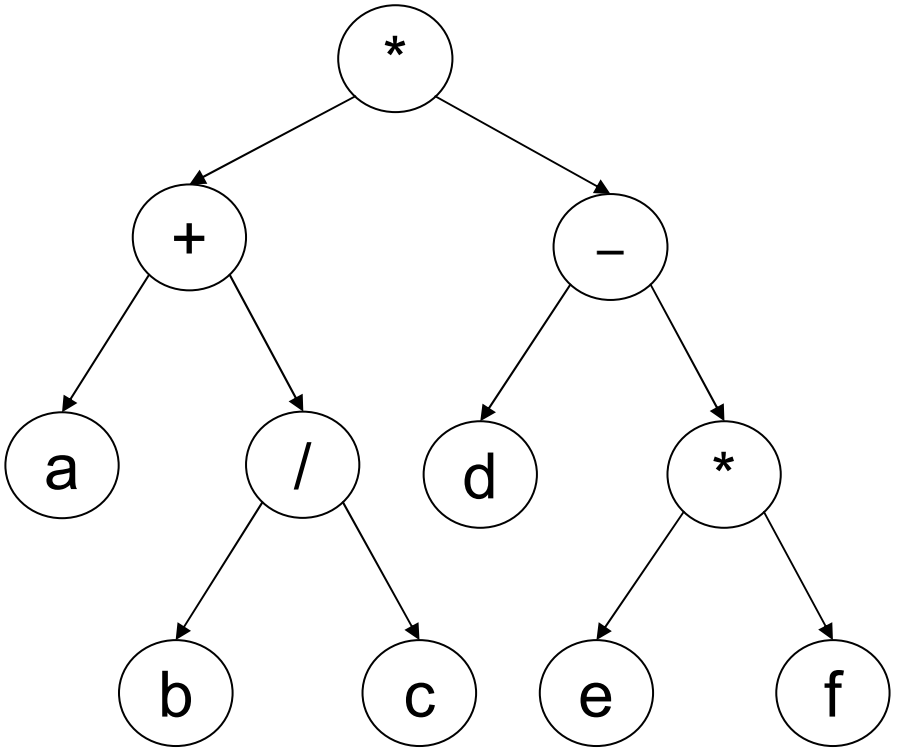
Percurso em AB

- Pós-ordem: H, D, E, B, I, F, G, C, A



Percurso em AB

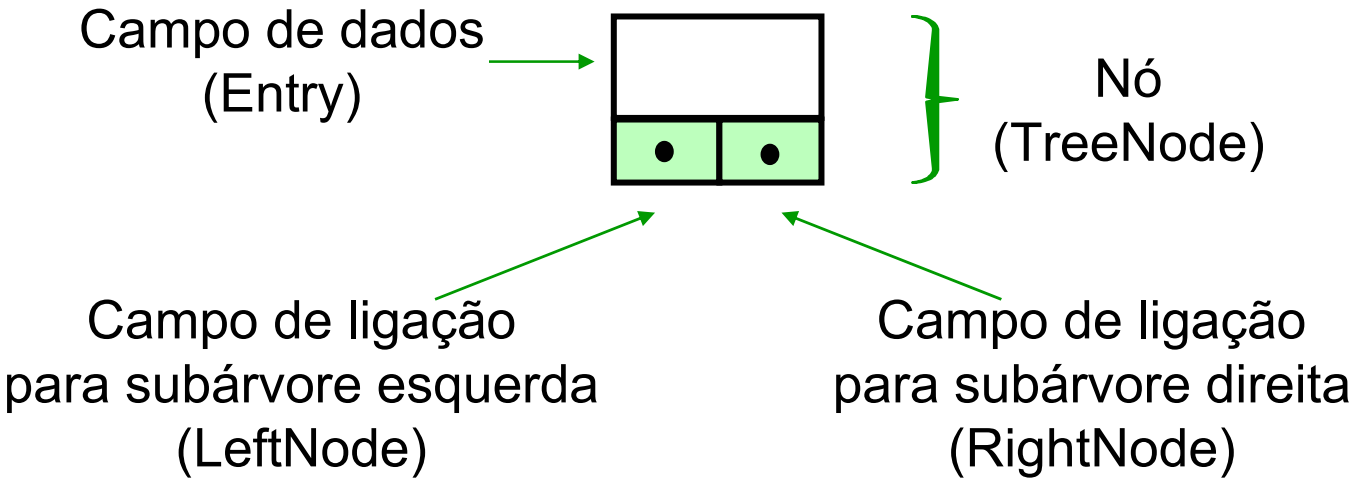
- Pré-ordem
  - * + a / b c – d * e f
- Em-ordem
  - a + b / c * d – e * f
- Pós-Ordem
  - a b c / + d e f * – *



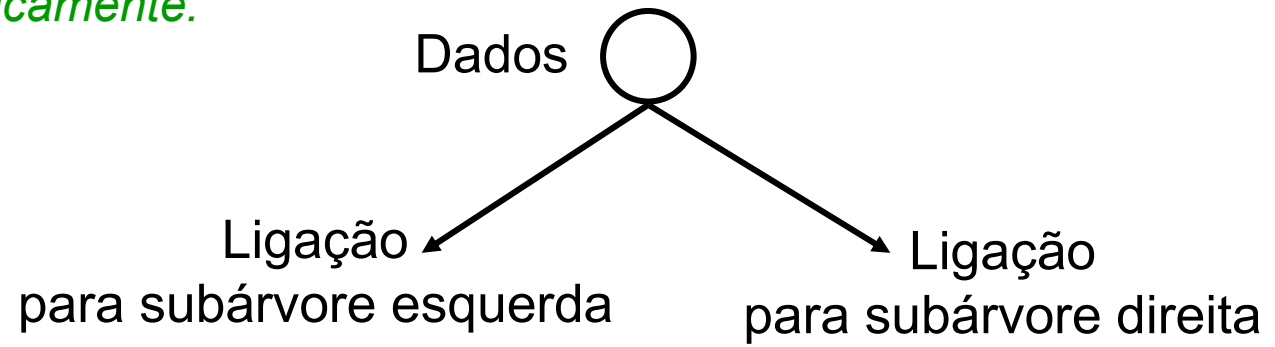
Implementação de Árvores Binárias

- É natural a implementação de árvores por meio de ponteiros
- Como toda árvore possui uma raiz (*root*), uma árvore vazia pode ser representada por um ponteiro aterrado (NULL em C++)
- Cada nó em uma árvore binária possui um campo de dados, um ponteiro para a subárvore esquerda e um ponteiro para a sub-árvore direita

Implementação de Árvores Binárias

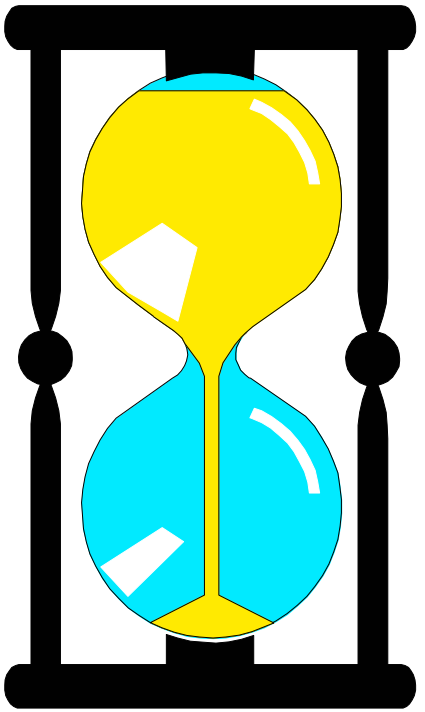


Esquematicamente:



Questão

Utilize estas idéias para escrever uma declaração de tipo que poderia implementar uma árvore binária para armazenar valores inteiros.

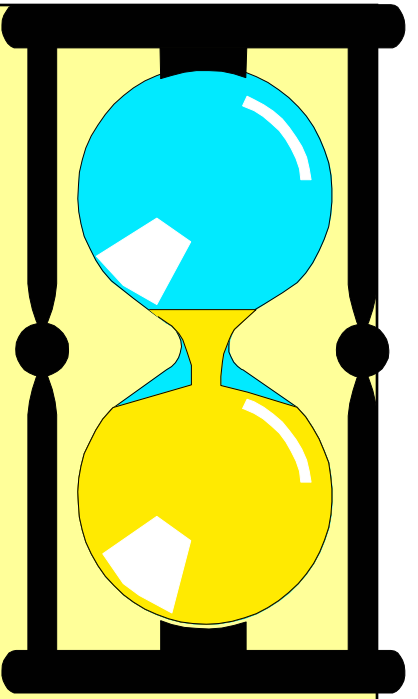


Você tem 5 minutos
para escrever a declaração

Uma Solução

```
class BinaryTree
{ public:
    BinaryTree();
    ~BinaryTree();
    void Insert(int x);
    void Delete(int x);
    bool Search(int x);
    …
  private:
    // declaração de tipos
    struct TreeNode
    { int Entry;            // tipo de dado colocado na árvore
      TreeNode *LeftNode, *RightNode; // subárvores
    };
    typedef TreeNode *TreePointer;

    // declaração de campos
    TreePointer root;
};
```



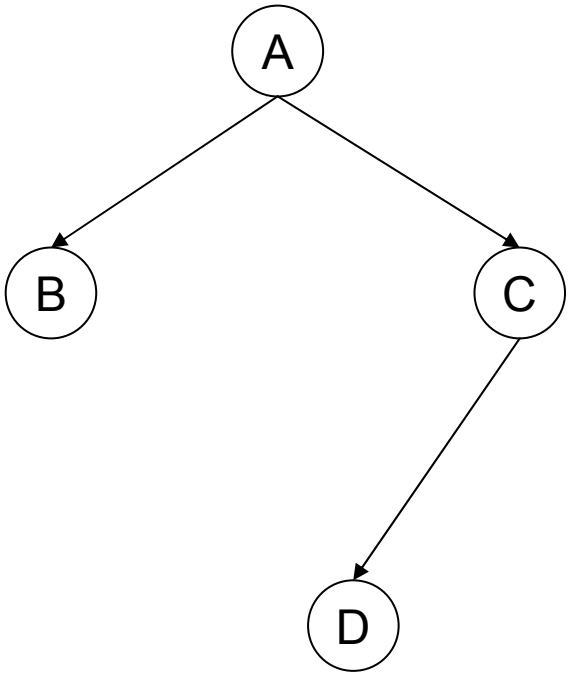
Uma Solução

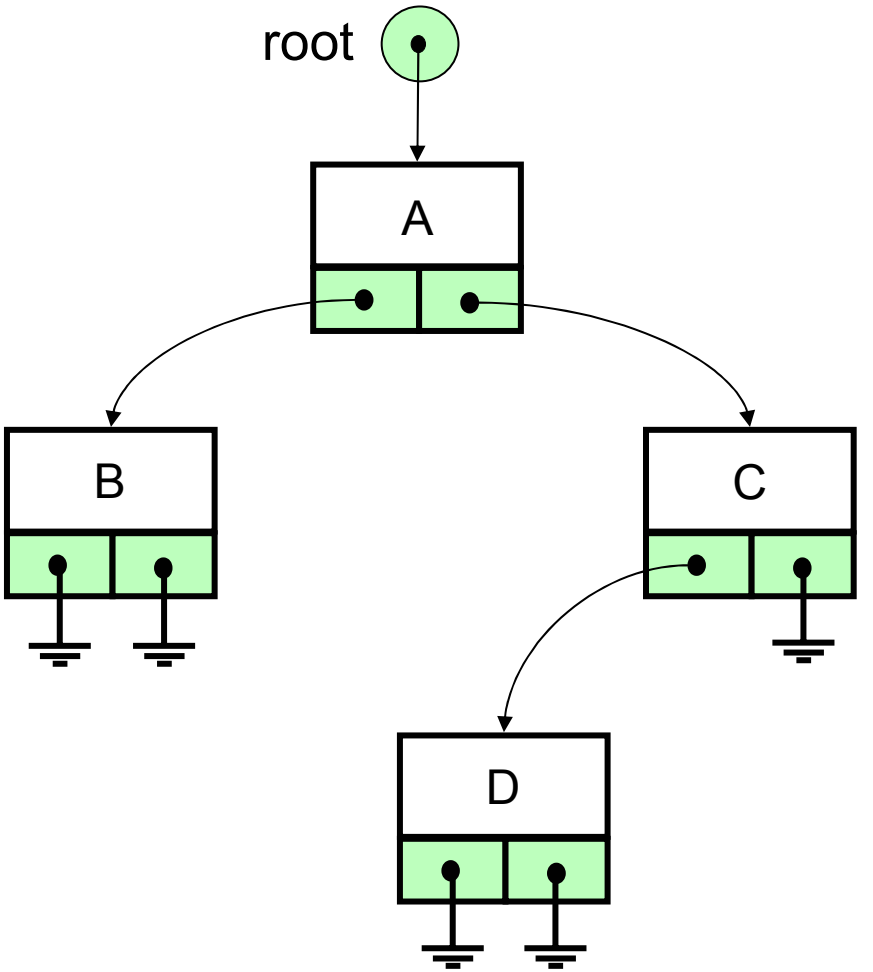
```
class BinaryTree
{ public:
    BinaryTree();
    ~BinaryTree();
    void Insert(int x);
    void Delete(int x);
    bool Search(int x);
    …
  private:
    // declaração de tipos
    struct TreeNode
    { int Entry;            // tipo de dado colocado na árvore
      TreeNode *LeftNode, *RightNode; // subárvores
    };
    typedef TreeNode *TreePointer;

    // declaração de campos
    TreePointer root;
};
```

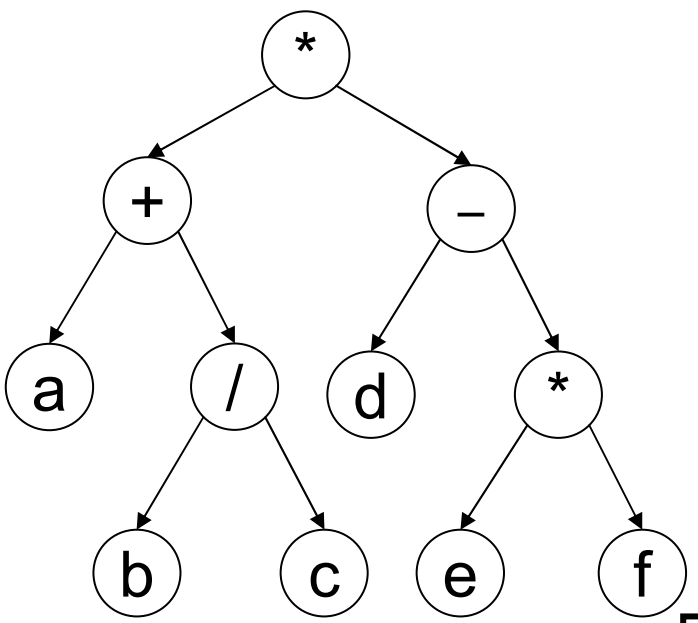
Observe que o tipo **TreeEntry** nesse caso é um inteiro

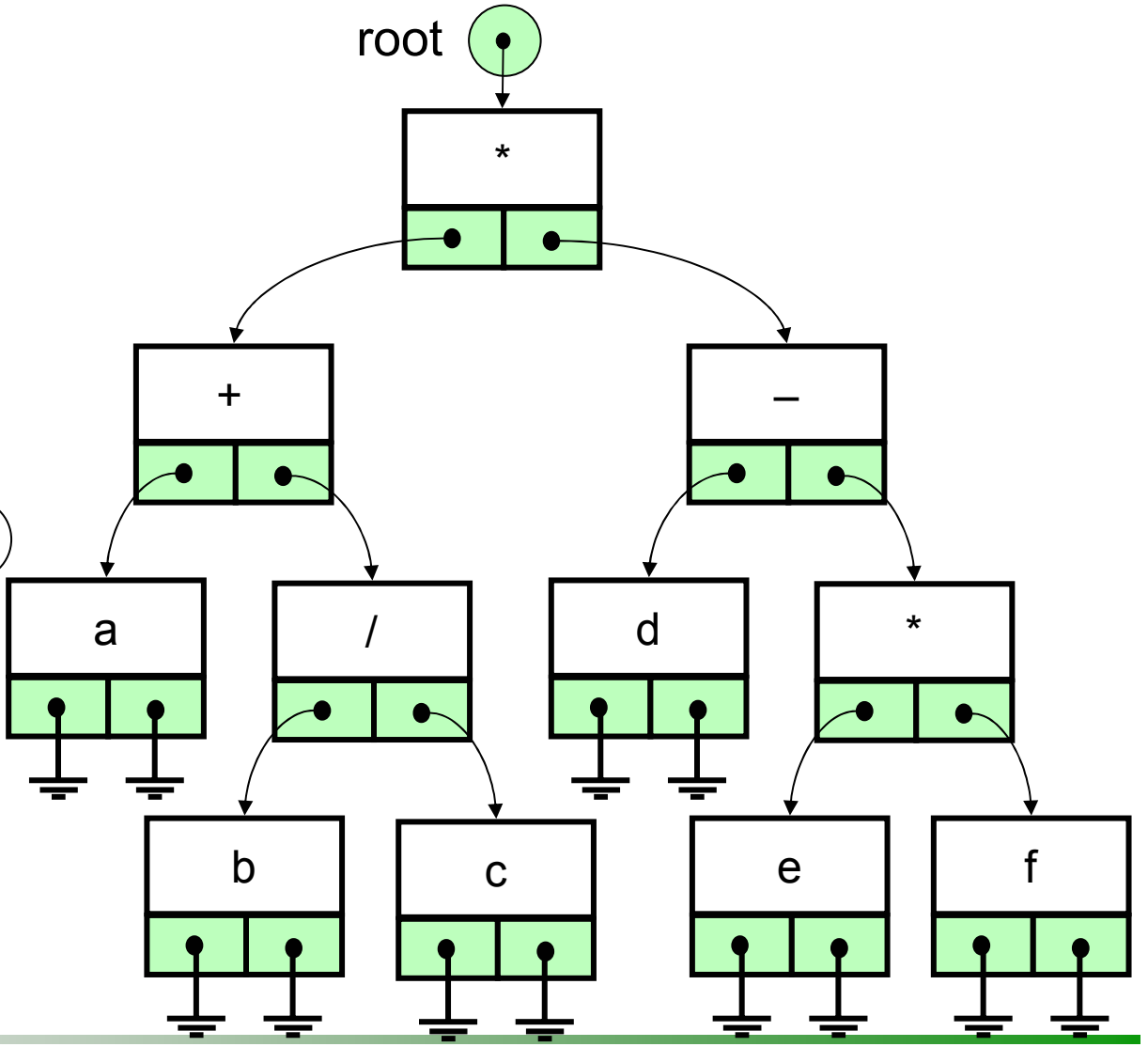
Implementação de Árvores Binárias





Implementação de Árvores Binárias





Operações Básicas

- Com a definição dada já é possível implementar alguns métodos para AB que também se aplicam para ABB (vista a seguir)
- Como os algoritmos em geral são recursivos, serão declarados dois métodos de mesmo nome
  - O método público que faz uma chamada ao método privado de mesmo nome, passando os parâmetros necessários para o método privado recursivo (normalmente a raiz; mas outros parâmetros também podem ser passados)
  - O método privado que efetivamente implementa o algoritmo recursivo

Número de Nós

int BinaryTree::Nodes();

- *pré-condição:* Árvore binária já tenha sido criada
- *pós-condição:* retorna o número de nós existentes na árvore

- Uma idéia para encontrar o número de nós é utilizar recursão:
  - Caso base: uma árvore vazia possui zero nós
  - Caso recursivo: uma árvore que contém um nó possui 1 (o próprio nó) somado ao número de nós na sua subárvore esquerda somado ao número de nós na sua subárvore direita

Número de Nós

```
int BinaryTree::Nodes() // método público
{  return Nodes(root);
}
//-------------------------------------------------------------------
int BinaryTree::Nodes(TreePointer &t) // método privado
{
  if(t == NULL)
     return 0;
  else
    return 1 + Nodes(t->LeftNode) + Nodes(t->RightNode);
}
```

Número de Folhas

int BinaryTree::Leaves();

- *pré-condição:* Árvore binária já tenha sido criada
- *pós-condição:* retorna o número de folhas existentes na árvore

- Novamente, o uso de recursão permite encontrar o número de folhas:
  - Caso base 1: uma árvore vazia possui zero folhas
  - Caso base 2: um nó cujas subárvores esquerda e direita são ambas vazias é uma folha
  - Caso recursivo: o número de folhas de uma árvore que contém um nó (não nulo) é determinado pelo número de folhas da subárvore esquerda deste nó somado ao número de folhas da subárvore direita deste nó

Número de Folhas

```
int BinaryTree::Leaves()
{  return Leaves(root);
}
//-----------------------------------------------------------------
int BinaryTree::Leaves(TreePointer &t)
{  if(t == NULL)
     return 0;
   else
     if(t->LeftNode == NULL && t->RightNode == NULL)
        return 1;
     else
        return Leaves(t->LeftNode) + Leaves(t->RightNode);
}
```

Altura

int BinaryTree::Height();

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

- A definição de altura de uma árvore nos leva ao 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

```
int BinaryTree::Height()
{  return Height(root);
}
//-----------------------------------------------------------------
int BinaryTree::Height(TreePointer &t)
{ if(t == NULL)
     return -1;
  else
  {   int L,R;
      L = Height(t->LeftNode);
      R = Height(t->RightNode);
      if(L>R) return L+1; else return R+1;
  }
}
```

Percurso em Pré-Ordem

- Para percorrer uma AB em pré-ordem, assume-se que existe um procedimento (ou método) denominado
  - void process(TreeEntry **x**)
- que efetua algum tipo de processamento com o valor **x** passado como parâmetro, lembrando que **TreeEntry** é o tipo de dado que é colocado na AB
- Os demais percursos são similares e sua implementação fica como exercício

Percurso em Pré-Ordem

```
void BinaryTree::PreOrder()
{  PreOrder(root);
}
//-----------------------------------------------------------------
void BinaryTree::PreOrder(TreePointer &t)
{
    if(t != NULL)
    {  process(t->Entry);
       PreOrder(t->LeftNode);
       PreOrder(t->RightNode);
     }
}
```

Em situações mais simples, *process* pode ser substituído por um comando de escrita

Percurso em Pré-Ordem

```
void BinaryTree::PreOrder()
{  PreOrder(root);
}
//-------------------------------------------------------------------
void BinaryTree::PreOrder(TreePointer &t)
{
    if(t != NULL)
    {  cout << t->Entry << endl;
       PreOrder(t->LeftNode);
       PreOrder(t->RightNode);
     }
}
```

Em situações mais simples, *process* pode ser substituído por um comando de escrita

Impressão

- A impressão de uma árvore binária pode ser efetuada utilizando algum dos percursos (pré-, in- ou pós-ordem) ou qualquer outra estratégia que for mais adequada
- A implementação seguinte imprime com deslocamentos (espaços) uma AB

Impressão

```
void BinaryTree::Print()
{  BinaryTree::Print(root,0);
}
//-----------------------------------------------------------------
void BinarySearchTree::Print(TreePointer &t, int s)
{ int i;
  if(t != NULL)
  {  Print(t->RightNode, s+3);
     for(i=1; i<=s; i++)
        cout << " ";                    // espaços
     cout << setw(6) << t->Entry << endl;
    Print(t->LeftNode, s+3);
  }
}
```

Considerações Finais

- Nesta apresentação foram vistos vários conceitos sobre árvores e árvores binárias, incluindo alguns algoritmos mais elementares
- Entretanto, imagine o processo de busca de informação em uma árvore (binária ou não)
  - 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 na próxima apresentação uma forma de melhorar o tempo de busca, utilizando Árvores Binárias de Busca