Árvores Binárias

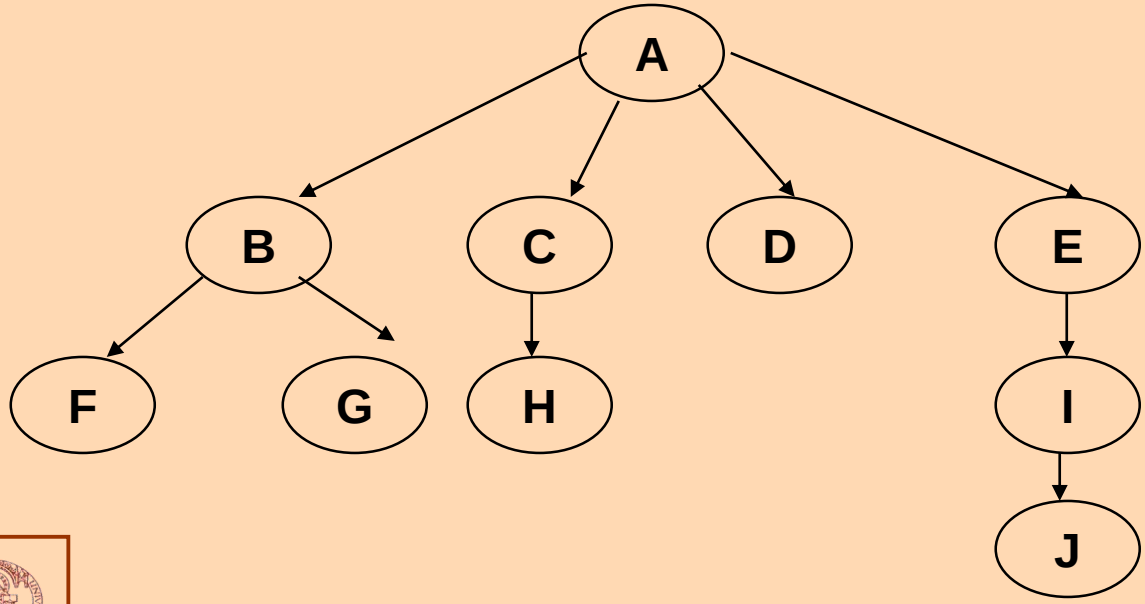
Algoritmos e Estruturas de Dados

2020/2021

FEUP

Árvores

- Conjunto de nós e conjunto de arestas que ligam pares de nós
  - Um nó é a *raiz*
  - Com exceção da raiz, todo o nó está ligado por uma aresta a 1 e 1 só nó (o pai)
  - Há um caminho único da raiz a cada nó; o *tamanho do caminho* para um nó é o número de arestas a percorrer

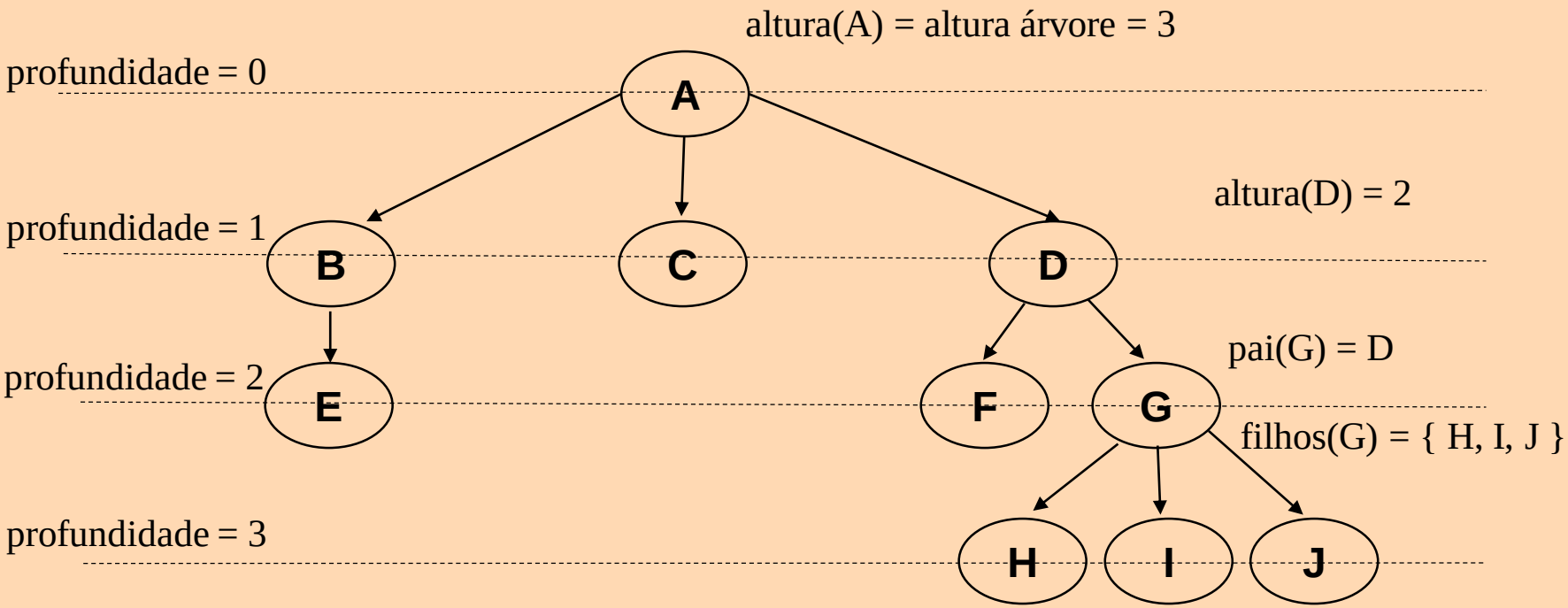


Nós sem descendentes: folhas

Árvores

- Ramos da árvore
  - Árvore de *N* nós tem *N-1* ramos
- Profundidade de um nó
  - Comprimento do caminho da raiz até ao nó
    - Profundidade da raiz é 0
    - Produndidade de um nó é 1 + a profundidade do seu pai
- Altura de um nó
  - Comprimento do caminho do nó até à folha de maior profundidade
    - Altura de uma folha é 0
    - Altura de um nó é 1 + a altura do seu filho de maior altura
  - Altura da árvore: altura da raiz
- Se existe caminho do nó *u* para o nó *v*
  - u é antepassado de v
  - v é descendente de u
- Tamanho de um nó: número de descendentes

Árvores


altura(A) = altura árvore = 3
profundidade = 0
A
altura(D) = 2
profundidade = 1
B
C
D
pai(G) = D
profundidade = 2
E
F
G
filhos(G) = { H, I, J }
profundidade = 3
H
I
J

Árvores binárias

- Uma árvore binária é uma árvore em que cada nó *não tem mais que dois filhos*

- Propriedades:
  - Uma árvore binária não vazia com profundidade *h* tem no mínimo *h+1*, e no máximo $2^{h+1}-1$ nós
  - A profundidade de uma árvore com *n* elementos (*n*>0) é no mínimo $log_2 n$, e no máximo *n-1*
  - A profundidade média de uma árvore de n nós é O(√n)

Árvores

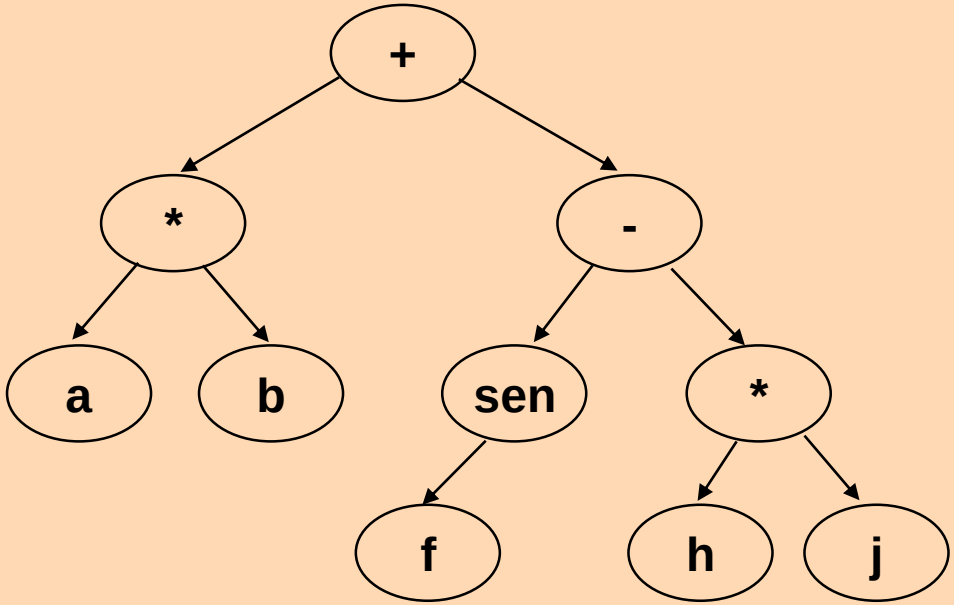
- Percorrer árvores

  Os elementos de uma árvore (binária) podem ser enumerados por quatro ordens diferentes. As três primeiras definem-se recursivamente:

  - ***Pré-ordem***: Primeiro a raiz, depois a sub-árvore esquerda, e finalmente a sub-árvore direita
  - ***Em-ordem***: Primeiro a sub-árvore esquerda, depois a raiz, e finalmente a sub-árvore direita
  - ***Pós-ordem***: Primeiro a sub-árvore esquerda, depois a sub-árvore direita, e finalmente a raiz
  - ***Por nível***: Os nós são processados por nível (profundidade) crescente, e dentro de cada nível, da esquerda para a direita

Árvores

- Percorrer árvores - exemplo



Pré-ordem	+ * a b – sen f * h j
Em-ordem	a * b + f sen – h * j
Pós-ordem	a b * f sen h j * - +
Por nível	+ * - a b sen * f h j

Árvores binárias: implementação

- Operações:
  - Criar uma árvore vazia
  - Determinar se uma árvore está vazia
  - Criar uma árvore a partir de duas sub-árvores
  - Eliminar os elementos da árvore (esvaziar a árvore)
  - Definir iteradores para percorrer a árvore
  - Imprimir uma árvore
  - ...

Árvores binárias: implementação

- *Nó* da árvore binária

```
template <class T> class BTNode {
   T element;
   BTNode<T> *left, *right;

   friend class BinaryTree<T>;
   friend class BTItrIn<T>;
   friend class BTItrPre<T>;
   friend class BTItrPos<T>;
   friend class BTItrLevel<T>;
public:
   BTNode(const T & e, BTNode<T> *esq = 0, BTNode<T> *dir = 0)
      : element(e), left(esq), right(dir) {}
};
```

Árvores binárias: implementação

- Declaração da classe ***BinaryTree*** em C++ (secção privada)

```
template <class T> class BinaryTree {
private:
   BTNode<T> *root;

   void makeEmpty(BTNode<T> *r);
   BTNode<T> *copySubtree(const BTNode<T> *n) const;
   void outputPreOrder(ostream & out, const BTNode<T> *n) const;

   friend class BTItrIn<T>;
   friend class BTItrPre<T>;
   friend class BTItrPos<T>;
   friend class BTItrLevel<T>;
   //...
};
```

Árvores binárias: implementação

- Declaração da classe ***BinaryTree*** em C++ (secção pública)

```
template <class T> class BinaryTree {
public:
   BinaryTree() { root = 0; }
   BinaryTree(const BinaryTree & t);
   BinaryTree(const T & elem);
   BinaryTree(const T & elem, const BinaryTree<T> & e,
                              const BinaryTree<T> & d);
   ~BinaryTree { makeEmpty(); }
   const BinaryTree & operator=(const BinaryTree<T> & rhs);
   bool isEmpty() const { return ( root == 0 ) ? true : false; }
   T & getRoot() const {
      if ( root ) return root->element ;
      else throw Underflow(); }
   void makeEmpty();
   void outputPreOrder(ostream & out) const ;
   //...
};
```

Árvores binárias: implementação

- classe ***BinaryTree*** : construtores

```
template <class T>
BinaryTree<T>::BinaryTree(const T & elem)
{
  root = new BTNode<T>(elem);
}

template <class T>
BinaryTree<T>::BinaryTree(const BinaryTree<T> & t)
{
  root = copySubTree(t.root);
}

template <class T>
BinaryTree<T>::BinaryTree(const T & elem, const BinaryTree<T> & e,
                          const BinaryTree<T> & d)
{
   root = new BTNode<T>(elem, copySubTree(e.root),
   copySubTree(d.root) );
```

Árvores binárias: implementação

- classe ***BinaryTree*** : copiar sub-árvores

```
template <class T>
BTNode<T> *BinaryTree<T>::copySubTree(const BTNode<T> *n) const
{
   if ( n) {
       BTNode<T> *node = new BTNode<T>(n->element,
                 copySubTree(n->left), copySubTree(n->right) );
       return node;
    } else return 0;
}

template <class T>
const BinaryTree<T> & BinaryTree<T>::operator=(const
                                        BinaryTree<T> & rhs)
{
   if ( this != & rhs ) {
      makeEmpty();
      root = copySubTree(rhs.root);
   }
   return *this;
}
```

Árvores binárias: implementação

- classe ***BinaryTree*** : esvaziar uma árvore

```
template <class T>
void BinaryTree<T>::makeEmpty()
{
   makeEmpty(root);
   root = 0;
}

template <class T>
void BinaryTree<T>::makeEmpty(BTNode<T> * r)
{
   if ( r ) {
      makeEmpty(r->left);
      makeEmpty(r->right);
      delete r;
   }
}
```

Árvores binárias: implementação

- classe ***BinaryTree*** : impressão em pré-ordem

```
template <class T>
void BinaryTree<T>:: outputPreOrder(ostream & out) const
{
   outputPreOrder(out, root);
}

template <class T>
void BinaryTree<T>::outputPreOrder(ostream & out,
                                   const BTNode<T> *r) const
{
   out << '(';
   if ( r ) {
      out << r->element << ')';
      outputPreOrder(out, r->left);
      out << ' ';
      outputPreOrder(out, r->right);
   }
   out << ')';
}
```

Árvores binárias: iteradores

Implementação de iteradores

- Construtor possui como parâmetro a árvore a iterar. Fica a referenciar o primeiro elemento.
- Métodos:
  - void advance(); // avança para o próximo elemento
  - T & retrieve(); // retorna o elemento referenciado pelo iterador
  - bool isAtEnd(); // verifica se chegou ao fim da árvore
- Uma implementação para cada possível enumeração dos elementos da árvore: pré-ordem (*BTItrPre*), em-ordem (*BTItrIn*), pós-ordem (*BTItrPos*), por nível (*BTItrLevel*),

Árvores binárias: implementação

- classe ***BTItrPre*** : iterador em pré-ordem

```
template <class T> class BTItrPre {
public:
   BTItrPre(const BinaryTree<T> & t);
   void advance();
   T & retrieve();
   bool isAtEnd() {  return itrStack.empty();  }
private:
   stack<BTNode<T> *> itrStack;
};


template <class T> BTItrPre<T>::BItrPre(const BinaryTree<T> & t)
{
   if ( !t.isEmpty() ) itrStack.push(t.root);
}

template <class T> T & BTItrPre<T>::retrieve()
{
   return itrStack.top()->element;
}
```

Árvores binárias: implementação

- classe ***BTItrPre*** : iterador em pré-ordem

```
template <class T> void BTItrPre<T>::advance()
{
   BTNode<T> * actual = itrStack.top();
   BTNode<T> * seguinte = actual->left;
   if ( seguinte )
      itrStack.push(seguinte);
   else {
      while ( ! itrStack.empty() ) {
         actual = itrStack.top();  itrStack.pop();
         seguinte = actual->right;
         if (seguinte) {
            itrStack.push(seguinte);
            break;
         }
      }
   }
}
```

Árvores binárias: implementação

- classe ***BTItrIn*** : iterador em-ordem

```
template <class T> class BTItrIn {
public:
   BTItrIn(const BinaryTree<T> & t);
   void advance();
   T & retrieve();
   bool isAtEnd() {  return itrStack.empty();  }
private:
   stack<BTNode<T> *> itrStack;
   void slideLeft(BTNode<T> *n);
};

template <class T> BTItrIn<T>::BItrIn(const BinaryTree<T> & t)
{  if ( !t.isEmpty() ) slideLeft(t.root);  }

template <class T> T & BTItrIn<T>::retrieve()
{  return itrStack.top()->element;  }
```

Árvores binárias: implementação

- classe ***BTItrIn*** : iterador em-ordem

```
template <class T> void BTItrIn<T>::slideLeft(BTNode<T> *n)
{
   while ( n ) {
      itrStack.push(n);
      n = n->left;
   }
}

template <class T> void BTItrIn<T>::advance()
{
   BTNode<T> * actual = itrStack.top();
   itrStack.pop();
   BTNode<T> * seguinte = actual->right;
   if ( seguinte )
        slideLeft(seguinte);
}
```

Árvores binárias: implementação

- classe ***BTItrPos***: iterador em pós-ordem

```
template <class T> class BTItrPos {
public:
   BTItrPos(const BinaryTree<T> & t);
   void advance();
   T & retrieve();
   bool isAtEnd() {  return itrStack.isEmpty();  }
private:
   stack<BTNode<T> *> itrStack;
   stack<bool> visitStack;
   void slideDown(BTNode<T> *n);
};

template <class T> BTItrPos<T>::BItrPos(const BinaryTree<T> & t)
{
   if ( !t.isEmpty() )
      slideDown(t.root);
}
```

Árvores binárias: implementação

- classe ***BTItrPos***: iterador em pós-ordem

```
template <class T>
T & BTItrPos<T>::retrieve()
{
   return itrStack.top()->element;
}

template <class T>
void BTItrPos<T>::advance()
{
   itrStack.pop();
   visitStack.pop();
   if ( (! itrStack.empty()) && (visitStack.top() == false) ) {
      visitStack.pop();
      visitStack.push(true);
      slideDown(itrStack.top()->right);
   }
}
```

Árvores binárias: implementação

- classe ***BTItrPos***: iterador em pós-ordem

```
template <class T>
T & BTItrPos<T>::slideDown(BTNode<T> *n)
{
   while ( n ) {
      itrStack.push(n);
      if ( n->left ) {
         visitStack.push(false);
         n = n->left;
      }
      else if ( n->right ) {
          visitStack.push(true);
          n = n->right;
      }
      else {
          visitStack.push(true); break;
      }
   }
}
```

Árvores binárias: implementação

- classe ***BTItrLevel***: iterador por nivel

```
template <class T> class BTItrLevel {
public:
   BTItrLevel(const BinaryTree<T> & t);
   void advance();
   T & retrieve();
   bool isAtEnd() {  return itrQueue.empty();  }
private:
   queue<BTNode<T> *> itrQueue;
};

template <class T>
BTItrLevel<T>::BItrLevel(const BinaryTree<T> & t)
{  if ( !t.isEmpty() ) itrQueue.push(t.root);  }

template <class T>
T & BTItrLevel<T>::retrieve()
{  return itrQueue.front()->element;  }
```

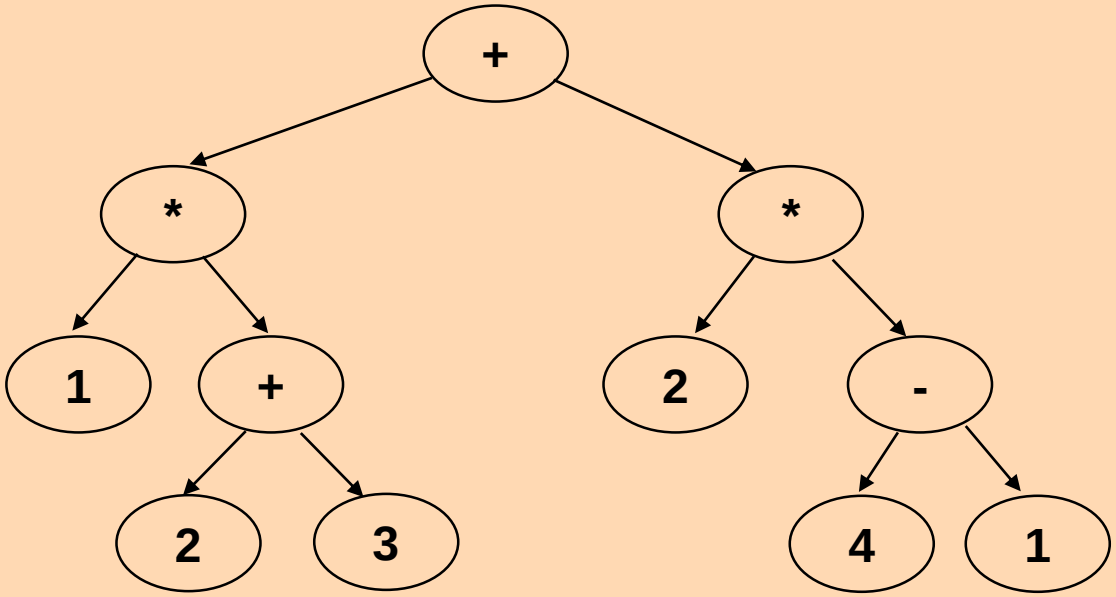
Árvores binárias: implementação

- classe ***BTItrLevel***: iterador por nivel

```
template <class T>
void BTItrLevel<T>::advance()
{
   BTNode<T> * actual = itrQueue.front();
   itrQueue.pop();
   BTNode<T> * seguinte = actual->left;
   if ( seguinte )
      itrQueue.push(seguinte);
   seguinte = actual->right;
   if ( seguinte )
      itrQueue.push(seguinte);
}
```

Árvores binárias: exemplo 1

Expressões aritméticas



Expressão = 1 * (2 + 3) + (2 * (4 – 1))

Árvores binárias: exemplo 1

Construção da árvore de expressões

- Algoritmo similar ao de conversão infixa->RPN, mas usa duas pilhas:
  - uma para guardar os operadores
  - outra para guardar sub-árvores correspondentes a sub-expressões.

- Algoritmo:
  - Números são transformados em árvores de 1 elemento e colocados na pilha de operandos.
  - Operadores são tratados como no programa de conversão de infixa -> RPN (usando uma pilha apenas para operadores e '(').
  - Quando um operador é retirado da pilha, duas sub-árvores (operandos) são retiradas da pilha de operandos e combinados numa nova sub-árvore, que é colocada na pilha de operandos.
  - Quando a leitura da expressão chega ao fim, todos os operadores existentes na pilha de operadores são processados.

Árvores binárias: exemplo 1

Elementos de uma expressão

```
enum operador { numero, parentesis_esq, mais, menos, vezes, dividir };
// por ordem de prioridade
class ExprElem {
public:
   operador tipo;
   double num;
   ExprElem(double n): tipo(numero), num(n) { };
   ExprElem(operador op): tipo(op) { };
};
```

Processar um operador binário

```
typedef BinaryTree<ExprElem> ArvArit;
void executaOpBin(operador op, stack<ArvArit> & operandStack)
{
   ArvArit right = operandStack.top(); operandStack.pop();
   ArvArit left = operandStack.top(); operandStack.pop();
   ExprElem e1(op);
   ArvArit novaArv(e1, left, right);
   operandStack.push(novaArv);
}
```

Árvores binárias: exemplo 1

Processar os operadores de maior prioridade

```
void processaOp(operador op, stack<operador> &operatorStack,
                             stack<ArvArit> &operandStack)
{
   while ((!operatorStack.empty()) && (op <= operatorStack.top())) {
      operador opx = operatorStack.top();
      operatorStack.pop();
      executaOpBin(opx, operandStack);
   }
   operatorStack.push(op);
}
```

Nota: As prioridades são definidas implicitamente na declaração do tipo operador.

Árvores binárias: exemplo 1

Núcleo do processamento de expressões

```
int main()
{
  stack<operador> operatorStack;
  stack<ArvArit> operandStack;
  string expressao;
  cout << "Escreva uma expressão: "; cin >> expressao;

  for (int i=0; i<expressao.length(); i++) {
    char c1=expressao[i];
    switch(c1) {
     case '(': operatorStack.push(parentesis_esq); break;
     case ')': while (operatorStack.top() != parentesis_esq) {
                    operador op=operatorStack.top();
                    operatorStack.pop();
                    executaOpBin(op, operandStack);
                }
               operatorStack.pop(); break;
    // continua
```

Árvores binárias: exemplo 1

Núcleo do processamento de expressões

```
int main()
{
  //continuação
     case '+': processaOp(mais, operatorStack, operandStack); break;
     case '-': processaOp(menos, operatorStack, operandStack); break;
     case '*': processaOp(vezes, operatorStack, operandStack); break;
     case '/': processaOp(dividir, operatorStack, operandStack); break;

     default: float y=c1-'0';
              ExprElem e(y);
              ArvArit arv(e);
              operandStack.push(arv); break;
     }
   }   // fim for

  //continua
```

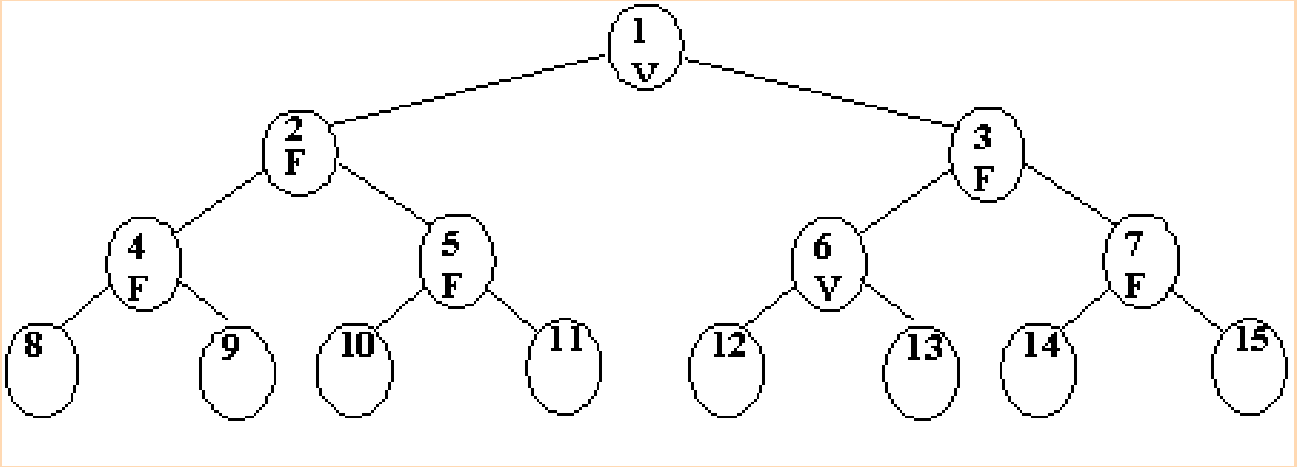
Árvores binárias: exemplo 1

```
int main()
{
   //continuação
   while (!operatorStack.empty() ) {
      operador opx = operatorStack.top();
      operatorStack.pop();
      executaOpBin(opx, operandStack);
   }

  // imprimir árvore final
   ArvArit arv1= operandStack.top();
   BTItrIn<ExprElem> it1(arv1);
   while ( !it1.isAtEnd() ) {
      ExprElem e1=it1.retrieve();
      cout << e1.tipo << ":" << e1.num << " ";
      it1.advance();
   }
}
```

Árvores binárias: exemplo 2

- Uma bola é lançada sobre um conjunto de círculos dispostos sob a forma de uma árvore binária completa



Cada círculo possui:

- uma pontuação
- um estado representado por um valor booleano que indica qual o caminho que a bola percorrerá quando chega a esse círculo
  - se estado=falso, a bola vai para a esquerda
  - se estado=verdadeiro, a bola vai para a direita.

Árvores binárias: exemplo 2

Quando a bola passa por um qualquer círculo:

- este muda o seu estado
- é incrementado o número de visitas a esse círculo (inicialmente 0).

Quando a bola atinge um círculo na base (nó da árvore), o jogador ganha o número de pontos inscritos nesse círculo.

Ganha o jogo o jogador com maior soma de pontos em *n* lançamentos.

```
class Circulo {
    int pontos;
    bool estado;
    int nVisitas;
public:
    …
};
```

```
class Jogo {
    BinaryTree<Circulo> jogo;
public:
    …
};
```

Árvores binárias: exemplo 2

- Implemente o construtor da classe **Jogo**, que cria um tabuleiro de jogo.

```
Jogo::Jogo(int niv, vector<int> &pontos, vector<bool> &estados)
```

Esta função cria uma árvore binária completa, de altura *niv*. Os vetores *pontos* e *estados* representam a pontuação e o estado dos círculos (nós da árvore) quando se efetua uma visita por nível.

Nota: Se numerar a posição dos nós de uma árvore visitada por nível de 0 a n-1 (n= nº de nós da árvore), o nó na posição *p* possui o filho esquerdo e o filho direito nas posições *2*p+1* e *2*p+2*, respetivamente.

Árvores binárias: exemplo 2

```
Jogo::Jogo(int niv, vector<int> &pontos, vector<bool> &estados)
{ jogo=iniciaJogo(0,niv,pontos,estados); }
```

```
BinaryTree<Circulo> Jogo::iniciaJogo(int pos,int niv,
                    vector<int> &pontos, vector<bool> &estados)
{
    Circulo c1(pontos[pos],estados[pos]);
    if (niv==0) return BinaryTree<Circulo>(c1);
    BinaryTree<Circulo> filhoEsq=
                iniciaJogo(2*pos+1,niv-1,pontos, estados);
    BinaryTree<Circulo> filhoDir=
                iniciaJogo(2*pos+2,niv-1,pontos, estados);
    return BinaryTree<Circulo>(c1, filhoEsq, filhoDir);
}
```

sub-árvore esquerda

sub-árvore direita

construção da árvore

Árvores binárias: exemplo 2

- Implemente a função que realiza uma jogada:

```
int Jogo::jogada()
```

Esta função realiza uma jogada. Altera o estado e incrementar o número de visitas de todos os círculos por onde a bola passa.

Retorna a pontuação do círculo base (folha da árvore) onde a bola lançada termina o seu percurso.

Sugestão: use um iterador por nível para percorrer a árvore.

Árvores binárias: exemplo 2

```
int Jogo::jogada() {
    int pos=1; int pontos=-1;
    BTItrLevel<Circulo> it(jogo);
    if (it.isAtEnd()) return pontos;
    while (true) {
       Circulo &c1=it.retrieve();
       bool estado=c1.getEstado();  int n;
       if (estado==false) n=pos;  else n=pos+1;
       c1.mudaEstado(); c1.incNVisitas();
       pontos=c1.getPontuacao();
       int i=0;
       while(i<n && !it.isAtEnd()) {
            it.advance();      // avanca p/ filho esquerdo ou direito
            i++; }
       if (!it.isAtEnd()) pos+=n;  else break;
    }
    return pontos;
}
```