Árvores

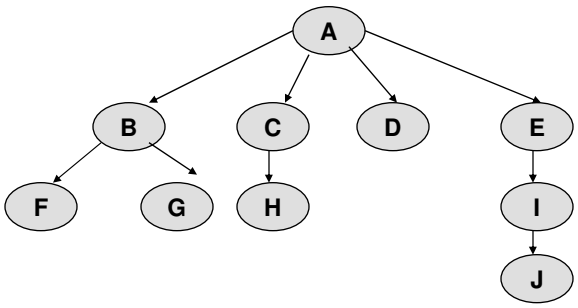
Algoritmos e Estruturas de Dados

2005/2006

FEUP

Árvores

- Conjunto de nós e conjunto de arestas que ligam pares de nós
  - Um nó é a *raiz*
  - Com excepção 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

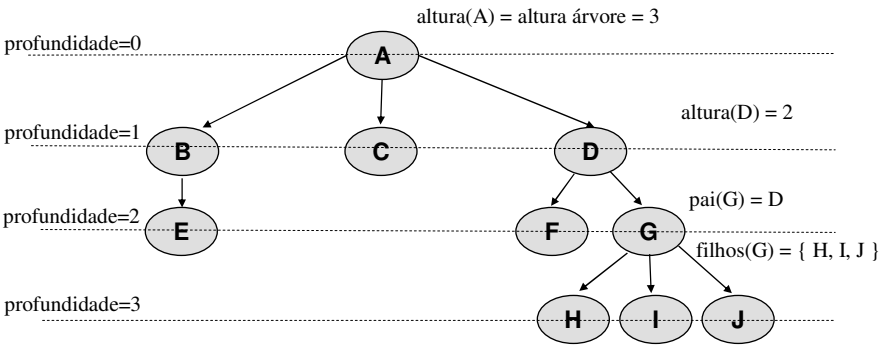


Nós sem descendentes: folhas

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 a 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



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

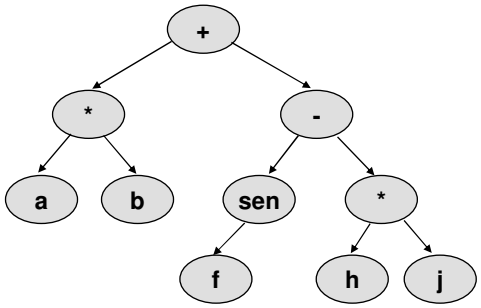
- Percorrer árvores

  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: implementação

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

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

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

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

Árvores binárias: implementação

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

```
template <class T> void BItrPre<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 ***BItrIn*** : iterador em-ordem

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

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

Árvores binárias: implementação

- classe ***BItrIn*** : iterador em-ordem

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

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

Árvores binárias: implementação

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

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

Árvores binárias: implementação

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

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

template <class T>
void BItrPos<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 ***BItrPos*** : iterador em pós-ordem

```
template <class T>
T & BItrPos<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 ***BItrLevel*** : iterador por nivel

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

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

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

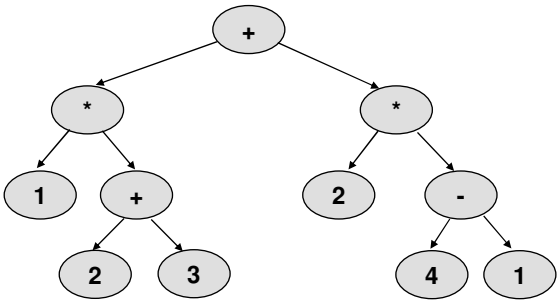
Árvores binárias: implementação

- classe ***BItrLevel*** : iterador por nivel

```
template <class T>
void BItrLevel<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: aplicações

Expressões aritméticas



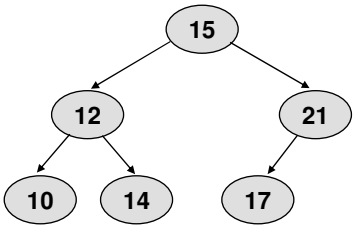
Expressão = 1 * (2 + 3) + (2 * (4 – 1))

Árvores binárias de pesquisa

- *Árvore binária de pesquisa*

  Árvore binária, sem elementos repetidos, que verifica a seguinte propriedade:

  – Para **cada nó**, todos os valores da sub-árvore esquerda são menores, e todos os valores da sub-árvore direita são maiores, que o valor desse nó



Árvores binárias de pesquisa

- Estrutura linear com elementos ordenados
  - A pesquisa de elementos pode ser realizada em *O(log n)*
  - ... mas não inserção ou remoção de elementos
- Estrutura em árvore binária
  - pode manter o tempo de acesso logarítmico nas operações de inserção e remoção de elementos
  - ***Árvore binária de pesquisa***
    - mais operações do que árvore binária básica: pesquisar, inserir, remover
    - objectos nos nós devem ser comparáveis (*Comparable*)

Árvores binárias de pesquisa

- Pesquisa
  - usa a propriedade de ordem na árvore para escolher caminho, eliminando uma sub-árvore a cada comparação
- Inserção
  - como pesquisa; novo nó é inserido onde a pesquisa falha
- Máximo e mínimo
  - procura, escolhendo sempre a subárvore direita (máximo), ou sempre a sub-árvore esquerda (mínimo)
- Remoção
  - Nó folha : apagar nó
  - Nó com 1 filho : filho substitui o pai
  - Nó com 2 filhos: elemento é substituído pelo menor da sub-árvore direita (ou maior da esquerda); o nó deste tem no máximo 1 filho e é apagado.

Árvores binárias de pesquisa: implementação

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

```
template <class Comparable> class BST {
private:
    BinaryNode<Comparable> *root;
    const Comparable ITEM_NOT_FOUND;

    const Comparable & elementAt( BinaryNode<Comparable> *t ) const;
    void insert( const Comparable & x, BinaryNode<Comparable> * & t );
    void remove( const Comparable & x, BinaryNode<Comparable> * & t );
    BinaryNode<Comparable> * findMin( BinaryNode<Comparable> *t ) const;
    BinaryNode<Comparable> * findMax( BinaryNode<Comparable> *t ) const;
    BinaryNode<Comparable> * find( const Comparable & x,
                                   BinaryNode<Comparable> *t ) const;
    void makeEmpty( BinaryNode<Comparable> * & t );
    void printTree( BinaryNode<Comparable> *t ) const;
    BinaryNode<Comparable> * copySubTree( BinaryNode<Comparable> *t );
//...
};
```


Árvores binárias de pesquisa: implementação

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

```
template <class Comparable> class BST {
public:
    explicit BST(const Comparable & notFound) { }
    BST(const BST & t);
    ~BST();
    const Comparable & findMin() const;
    const Comparable & findMax() const;
    const Comparable & find(const Comparable & x) const;
    bool isEmpty() const;
    void printTree() const;
    void makeEmpty();
    void insert(const Comparable & x);
    void remove(const Comparable & x);
    const BST & operator =(const BST & rhs);
//...
};
```

Árvores binárias de pesquisa: implementação

- classe ***BST*** : construtores e destrutor

```
template <class Comparable>
BST<Comparable>::BST( const Comparable & notFound ) : root(NULL),
                                ITEM_NOT_FOUND( notFound )
{ }

template <class Comparable>
BST<Comparable>::BST( const BST<Comparable> & rhs ) : root( NULL ),
                    ITEM_NOT_FOUND( rhs.ITEM_NOT_FOUND )
{  *this = rhs; }

template <class Comparable>
BST<Comparable>::~BST( )
{ makeEmpty( ); }
```

Árvores binárias de pesquisa: implementação

- classe ***BST*** : *pesquisa de elementos*

```
template <class Comparable>
const Comparable & BST<Comparable>:: find( const Comparable & x ) const
{ return elementAt( find( x, root ) ); }

template <class Comparable> const Comparable & BST<Comparable>::findMin( ) const
{ return elementAt( findMin( root ) ); }

template <class Comparable> const Comparable & BST<Comparable>::findMax( ) const
{ return elementAt( findMax( root ) ); }

template <class Comparable>
const Comparable & BST<Comparable>:: elementAt( BinaryNode<Comparable> *t ) const
{
    if( t == NULL ) return ITEM_NOT_FOUND;
    else return t->element;
}
```

Árvores binárias de pesquisa: implementação

- classe ***BST*** : *find*

```
template <class Comparable>
BinaryNode<Comparable> *
BST<Comparable>:: find(const Comparable & x, BinaryNode<Comparable> * t) const
{
    if ( t == NULL )
        return NULL;
    else if ( x < t->element )
        return find(x, t->left);
    else if ( t->element < x )
        return find(x, t->right);
    else return t;
}
```

Nota: apenas é usado o operador <

Árvores binárias de pesquisa: implementação

- classe ***BST*** : *findMin* , *findMax*

```
template <class Comparable> BinaryNode<Comparable> *
BST<Comparable>:: findMin(BinaryNode<Comparable> * t) const
{
    if ( t == NULL ) return NULL;
    if ( t->left == NULL ) return t;
    return findMin(t->left);
}

template <class Comparable> BinaryNode<Comparable> *
BST<Comparable>:: findMax(BinaryNode<Comparable> * t) const
{
    if ( t != NULL )
        while ( t->right != NULL )  t = t->right;
    return t;
}
```

Árvores binárias de pesquisa: implementação

- classe ***BST*** : *insert*

```
template <class Comparable>
void BST<Comparable>:: insert(const Comparable & x, BinaryNode<Comparable> * & t)
{
    if ( t == NULL )
        t = new BinaryNode<Comparable>(x, NULL, NULL);
    else if ( x < t->element)
        insert(x, t->left);
    else if (t->element < x)
        insert(x, t->right);
    else
        ;    // não fazer nada. nó repetido
}
```

Árvores binárias de pesquisa: implementação

- classe ***BST*** : *remove*

```
template <class Comparable>
void BST<Comparable>:: remove(const Comparable & x, BinaryNode<Comparable> * & t)
{
    if ( t == NULL ) return;                // não existe
    if ( x < t->element ) remove(x, t->left);
    else if ( t->element < x ) remove(x, t->right);
    else if ( t->left !=  NULL && t->right != NULL ) {
        t->element = findMin(t->right)->element;
        remove(t->element, t->right);
    }
    else {
        BinaryNode<Comparable> * oldNode = t;
        t = ( t->left != NULL ) ? t->left : t->right;
        delete oldNode;
    }
}
```

Árvores binárias de pesquisa: implementação

- classe ***BST*** : *cópia e atribuição*

  As operações de cópia e atribuição são implementadas como na classe *BinaryTree*

  - **make Empty** , como na classe BinaryTree
  - **operator =** , como na classe BinaryTree
  - **copySubTree** , como na classe BinaryTree

Árvores binárias de pesquisa: aplicação

- Contagem de ocorrências de palavras

  Pretende-se escrever um programa que leia um ficheiro de texto e apresente uma listagem ordenada das palavras nele existentes e o respectivo número de ocorrências.

  - Guardar as palavras e contadores associados numa árvore binária de pesquisa.
  - Usar ordem alfabética para comparar os nós.

Árvores binárias de pesquisa: aplicação

- classe ***PalavraFreq*** : representação das palavras e sua frequência

```
class PalavraFreq
{
    string palavra;
    int frequencia;
public:
    PalavraFreq() : palavra(“”), frequencia(0) {};
    PalavraFreq(string p) : palavra(p), frequencia(1) {};
    bool operator < (const PalavraFreq & p) const { return palavra < p.palavra; }
    bool operator == (const PalavraFreq & p) const { return palavra == p.palavra; }
    friend ostream & operator << (ostream & out, const PalavraFreq & p);
    void incFrequencia() { frequencia ++; }
};
ostream & operator << (ostream & out, const PalavraFreq & p) {
    out << p.palavra << ‘ : ‘ << p.frequencia << endl; return out;
}
```

Árvores binárias de pesquisa: aplicação

```
main()
{
    PalavraFreq notF(“”);
    BST<PalavraFreq> palavras(notF);
    string palavra1 = getPalavra();
    while ( palavra1 != “” ) {
        PalavraFreq pesq = palavras.find(PalavraFreq(palavra1));
        if ( pesq == notF ) palavras.insert(PalavraFreq(palavra1));
        else pesq.incFrequencia();
        palavra1 = getPalavra();
    }
    BItrIn<PalavraFreq> itr(palavras);
    while ( ! itr.isAtEnd() ) {
        cout << itr.retrieve();
        itr.advance();
    }
}
```