Árvores Binárias de Pesquisa

Algoritmos e Estruturas de Dados

2019/2020

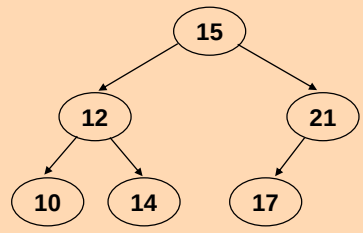
FEUP

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
    - objetos 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 que substitui o pai.

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;
   bool insert( const Comparable & x, BinaryNode<Comparable> * & t );
   bool 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();
   bool insert(const Comparable & x);
   bool remove(const Comparable & x);
   const BST & operator =(const BST & rhs);
   //...
};
```

construtor

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>
bool BST<Comparable>::insert(const Comparable & x)
{
   return insert (x,root);
}

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


Árvores binárias de pesquisa: implementação

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

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

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

- classe ***BST*** : *iteradores*
  - classe **BSTItrIn** : iterador em-ordem
  - classe **BSTItrPre** : iterador em pre-ordem
  - classe **BSTItrPost** : iterador em pos-ordem
  - classe **BSTItrLevel** : iterador em nivel

  - métodos dos iteradores:
    - **BSTItrIn**(const BST<*Comparable*> &arv) // construtor da BSTItrIn
    - **BSTItrPre**(const BST<*Comparable*> &arv) // construtor da BSTItrPre
    - **BSTItrPost**(const BST<*Comparable*> &arv) // construtor da BSTItrPost
    - **BSTItrLevel**(const BST<*Comparable*> &arv) // construtor da BSTItrLevel
    - void advance ()
    - const *Comparable* & retrieve()
    - bool isAtEnd()

Árvores binárias de pesquisa: implementação

Outra implementação de iterador em ordem

```
template <class Comparable>
class iteratorBST {
    stack<BinaryNode<Comparable> *> itrStack;
    friend class BST<Comparable>;
    …
public:
    iteratorBST<Comparable>& operator++(int);
    Comparable operator*() const;
    bool operator==(const iteratorBST<Comparable> &it2) const;
    bool operator!=(const iteratorBST<Comparable> &it2) const;
};
```

```
template <class Comparable>
class BST {
   …
   iteratorBST<Comparable> begin() const;
   iteratorBST<Comparable> end() const;
}
```

Árvores binárias de pesquisa: exemplo 1

- 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 respetivo 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: exemplo 1

- 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: exemplo 1

```
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 {
         palavras.remove(pesq); pesq.incFrequencia();
         palavras.insert(pesq);
      }
      palavra1 = getPalavra();
   }
   BSTItrIn<PalavraFreq> itr(palavras);
   while ( ! itr.isAtEnd() ) {
      cout << itr.retrieve();
      itr.advance();
   }
}
```


Árvores binárias de pesquisa: exemplo 2

Numa biblioteca, a informação sobre os livros existentes é guardada numa árvore binária de pesquisa (*livros*) ordenada por autor e, para o mesmo autor, por título.

```
class Livro {
   string titulo;
   string autor;
public:
   // …
};

class Biblioteca {
  BST<Livro> livros;
public:
  Biblioteca();
  // …
};
```

Árvores binárias de pesquisa: exemplo 2

- Implemente na classe **Biblioteca** o membro-função:

  void addLivros(const vector<Livro> & livros1)

  Esta função insere na BST *livros* os livros existentes no vetor *livros1*.

```
Biblioteca():livros(Livro("","")){};
```

```
void Biblioteca::addLivros(const vector<Livro> & livros1) {
    for (int i=0; i< livros1.size(); i++)
       livros.insert(livros1[i]);
}
```

```
bool Livro::operator < (const Livro & l1) const {
    if (autor==l1.autor)
       return (titulo < l1.titulo);
    else
       return (autor<l1.autor);
}
```


Árvores binárias de pesquisa: exemplo 2

- Implemente na classe **Biblioteca** o membro-função:

  `vector<string> getTitulos(string autor1, int & nVisitas)`

  A função retorna um vetor com os títulos, por ordem alfabética, de todos os livros do autor *autor1*, visitando o menor número de nós possível. Retorna ainda em *nVisitas* o número de nós da BST que foram visitados nesta pesquisa.

```
vector<string> Biblioteca::getTitulos(string autor1, int &nVisitas) {
    vector<string> vres; nVisitas=0;
    BSTItrIn<Livro> it(livros);
    while (!it.isAtEnd()) {
        nVisitas++;
        if (it.retrieve().getAutor()>autor1) break;
        if (it.retrieve().getAutor()==autor1)
            vres.push_back(it.retrieve().getTitulo());
        it.advance();
    }
    return vres;
}
```

Árvores binárias de pesquisa: exemplo 2

- Implemente na classe **Biblioteca** o membro-função:

  `string removeLivro(string autor1, string titulo1)`

  A função remove o livro do autor *autor1* e título *titulo1* da BST *livros* e retorna a string *"removido"*.

  Se o livro não existir, retorna o título do primeiro livro (por ordem alfabética) desse autor.

  Se não existir nenhum livro desse autor retorna a string *"autor inexistente"*.

Árvores binárias de pesquisa: exemplo 2

```
string Biblioteca::removeLivro(string autor1, string titulo1){
   Livro l1(titulo1, autor1);
   Livro lf=livros.find(l1);
   if (lf==Livro("","")) {
      BSTItrIn<Livro> it(livros);
      while (!it.isAtEnd()) {
         if (it.retrieve().getAutor()==autor1)
                        return it.retrieve().getTitulo();
         if (it.retrieve().getAutor()>autor1)
                        return "autor inexistente";
          it.advance();
      }
      return "autor inexistente";
   }
   livros.remove(lf);
   return "removido";
}
```