Universidade do Minho
Escola de Engenharia

MIETI :: Métodos de Programação II

2015/16

Práticas Laboratoriais

Módulo 6

António Esteves

esteves@di.uminho.pt

Maio 2016

Exercício 1 – Árvores Binárias de Pesquisa

Implementar as seguintes funcionalidades, cada uma numa função, que suportam a utilização de uma árvore binária de pesquisa:

1. Inserir um nodo na árvore
2. Percorrer a árvore e imprimir os dados úteis de cada nodo de forma ordenada, de acordo com o valor das chaves
3. Procurar na árvore o nodo que possui uma dada chave
4. Procurar o nodo com chave mínima e o nodo com chave máxima
5. Procurar o sucessor de um dado nodo
6. Procurar o antecessor de um dado nodo
7. Remover da árvore um determinado nodo
8. Calcular a profundidade da árvore
9. Escreva a função main() de modo a testar cada uma das funções

Exercício 1 – Árvores Binárias de Pesquisa

- Assuma que o nodo da árvore binária tem a estrutura que se segue, sendo os dados úteis um nome com capacidade para 50 carateres:

```
struct nodoA {

  int           chave;

  <tipo>        dadosUteis;

  struct nodoA  *nodoPai;

  struct nodoA  *nodoEsq;

  struct nodoA  *nodoDir;

};
```

Exercício 1 – Árvores Binárias de Pesquisa

1. Implementar a função que insere um nodo ***z***, passado como argumento, na árvore. Para testar esta função é preciso implementar uma função que cria um nodo. O algoritmo da função de inserção é:

```
Inserir_Arvore(r,z)
y = NULL
x = r // x= raiz de T
enquanto x ≠ NULL fazer
y=x
se z->chave < x->chave então
    x = x->nodoEsq
senão
    x = x->nodoDir
fse
fenquanto
```

```
z->nodoPai = y
se y = NULL então
    r = z // atualizar a raiz de T
senão se z->chave < y->chave então
    y->nodoEsq = z
senão
    y->nodoDir = z
fse
Devolve r // raiz de T atualizada
```

Exercício 1 – Árvores Binárias de Pesquisa

2. Implementar a função recursiva **Travessia_Ordenada_Arvore()** que percorre a árvore e imprime os dados úteis de cada nodo de forma ordenada, de acordo com o valor das chaves. Utilize o seguinte algoritmo:

Travessia_Ordenada_Arvore(**x**)

se **x** != NULL então

Travessia_Ordenada_Arvore(**x**->nodoEsq)

Imprimir a chave e os dados úteis de **x**

Travessia_Ordenada_Arvore(**x**->nodoDir)

fse

Exercício 1 – Árvores Binárias de Pesquisa

3. Implementar a função **Pesquisa_Arvore()** que pesquisa o nodo com uma dada chave ***k***. Recebe como argumentos o apontador para o nodo raiz ***r*** e a chave ***k***, e devolve um apontador para o nodo com a chave ***k***, caso ele exista, ou NULL se não existir.

Pesquisa_Arvore (r, k)

```
se r = NULL ou k = r->chave então
    devolve r
fse
se k < r->chave então
    devolve Pesquisa_Arvore(r->nodoEsq, k)
senão
    devolve Pesquisa_Arvore(r->nodoDir, k)
fse
```

Exercício 1 – Árvores Binárias de Pesquisa

4. Implementar as funções que procuram o elemento com chave mínima e máxima da árvore. As funções recebem como argumento o apontador para o nodo ***r***, que é a raiz da (sub)árvore, e devolve o apontador para o nodo com chave mínima/máxima. O algoritmo da função que procura o mínimo é:

```
Minimo_Arvore(r)
    enquanto r->nodoEsq ≠ NULL fazer
        r = r->nodoEsq
    fenquanto
    devolver r
```

O algoritmo da função que procura o máximo é semelhante:

```
Maximo_Arvore(r)
    enquanto r->nodoDir ≠ NULL fazer
        r = r->nodoDir
    fenquanto
    devolver r
```

Exercício 1 – Árvores Binárias de Pesquisa

5. Implementar a função que procura o sucessor de um dado nodo ***x***. A função recebe o apontador para o nodo ***x*** e devolve o apontador para o seu sucessor (se existir), ou NULL se a chave de ***x*** for a maior da árvore. O algoritmo é:

```
Sucessor_Arvore(x)
    se x->nodoDir ≠ NULL então
        devolver Minimo_Arvore(x->nodoDir)
    fse
    y = x->nodoPai
    enquanto y ≠ NULL e x = y->nodoDir fazer
        x=y
        y = y->nodoPai
    fenquanto
    devolver y
```

Exercício 1 – Árvores Binárias de Pesquisa

6. Implementar a função que procura o antecessor de um dado nodo ***x***. A função recebe o apontador para o nodo ***x*** e devolve o apontador para o seu antecessor (se existir), ou NULL se a chave de ***x*** for a menor da árvore.

Exercício 1 – Árvores Binárias de Pesquisa

7. Implementar a função que remove o nodo ***z*** da árvore. ***T*** é o apontador para o nodo raiz. O algoritmo da função é o seguinte:

Remover_Arvore(**T**, **z**)

```
// Determinar o nodo a corrigir devido a eliminar-se z → y
se z->nodoEsq = NULL ou z->nodoDir = NULL então
    y=z
senão
    y = Sucessor_Arvore(z)
fse
// Determinar o filho não NULL de y → x
se y->nodoEsq ≠ NULL então
    x = y->nodoEsq
senão
    x = y->nodoDir
fse
```

Exercício 1 – Árvores Binárias de Pesquisa

```
se x ≠ NULL então          // Corrigir o nodo x
    x->nodoPai = y->nodoPai
fse
se y->nodoPai = NULL então // Atualizar o nodo raiz
    T = x
senão se y = (y->nodoPai)->nodoEsq então // Corrigir o pai de y
    (y->nodoPai)->nodoEsq = x
senão
    (y->nodoPai)->nodoDir = x
fse
se y ≠ z então             // Copiar os dados do sucessor de z (y) para z
    chave de z = chave de y
    copiar os dados úteis de y para z
fse
Remover y de memória ; Devolver T
```

Exercício 1 – Árvores Binárias de Pesquisa

8. Implementar a função que calcula a **profundidade** da árvore

9. Implementar a função **main()** para testar todas as funções, usando o seguinte cenário:

   - Inserir nodos com chave: 15, 5, 16, 3, 12, 20, 10, 13, 18, 22, 6, 7
   - Atravessar a árvore de forma ordenada
   - Pesquisar o nodo com chave 50 e depois o nodo com chave 7
   - Calcular o mínimo e o máximo da árvore
   - Encontrar o sucessor do nodo com chave 13
   - Encontrar o antecessor do nodo com chave 18
   - Remover os nodos com chave 13, 13, 16 e 5
   - Calcular a profundidade da árvore