Lista de Exercícios - Árvores

Marcos Fagundes Caetano

Abril 2018

Exercícios

1. Implemente uma árvore binária de inteiros a partir do seguinte protótipo:

```
typedef struct t_no {
        int dado;
        struct t_no* esq;
        struct t_no* dir;
} tipoNo;

tipoNo* criaRaiz  (int dado);
tipoNo* inserirNo (tipoNo* raiz, int dado);
tipoNo* buscaNo   (int dado);
void    liberaNo  (tipoNo* no);
```

2. Escreva algoritmos para determinar:

   (a) o nó mais a esquerda de uma árvore bínária.

   (b) o nó mais a direita de uma árvore binária.

   (c) o nó mais profundo de uma árvore binária (se houver empate, escolha o mais a direita)

   (d) o número de nós numa árvore binária;

   (e) a soma do conteúdo de todos os nós numa árvore binária;

   (f) a profundidade de uma árvore binária.

   Fonte: *Tenenbaum, A. A; Langsan Y.; Augenstein M. Estruturas de Dados Usando C. PrenticeHall, 1995.*

3. Crie funções para percorrer e imprimir cada elemento de uma árvore:

   (a) em pré-ordem

   (b) em ordem

   (c) em pós-ordem

   (d) em largura

4. Implemente uma função que, dada uma árvore de pesquisa binária não vazia (uma árvore binária ordenada), retorne o valor mínimo de dados encontrado nessa árvore. Note que não é necessário procurar a árvore inteira, isso pode ser resolvido com recursão ou com um loop while simples. Assinatura da função:

```
int minValue (struct node * node);
```

   Fonte: *Binary Trees by Nick Parlante. Acesso em: 15 abr. 2018.*

5. A partir da árvore especificada no exercício 1, crie uma função *buscaPai()* que retorne o endereço de memória do pai do nó cujo valor foi informado como argumento.
   Utilizando-se deste algoritmo desenvolvido, crie uma função que remova um nó-folha da árvore, recebendo como argumento o inteiro correspondente ao conteúdo. Lembre-se de liberar sua memória e de remover a atribuição de seu pai.

6. Dado duas árvores binárias de inteiros como argumento, crie uma função *mesmaArvore()* que verifique se as duas árvores são estruturalmente idênticas – ou seja, possuem os nós com os mesmos valores organizados da mesma forma.
   Fonte: *Binary Trees by Nick Parlante. Acesso em: 14 abr. 2018.*

7. Crie uma função *espelho()* que troque os ponteiros esquerdo e direito entre si de todos os nós de uma árvore binária, conforme mostra a figura 1.

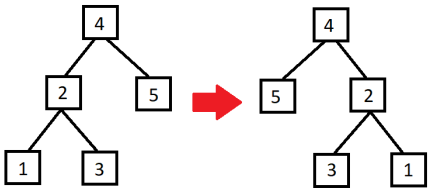

Figure 1: Representação de uma árvore antes e depois da função espelho.

   Fonte: *Binary Trees by Nick Parlante. Acesso em: 14 abr. 2018.*

8. Para cada nó em uma árvore de pesquisa binária, crie um novo nó duplicado e insira-o como o filho esquerdo do nó original. A árvore resultante ainda deve ser uma árvore de pesquisa binária.

   Então a árvore:

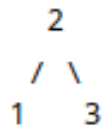

Figure 2: Representação de uma árvore antes da função.

é alterado para:

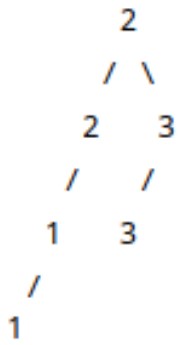

Figure 3: Representação de uma árvore depois da função.

Assinatura da função:

```
void doubleTree (struct node * node);
```

Lembrando que isso pode ser feito sem alterar o ponteiro do nó raiz.

Fonte: *Binary Trees by Nick Parlante. Acesso em: 15 abr. 2018.*

9. Crie uma função que verifique se uma árvore binária está totalmente preenchida, ou seja, possui todas as folhas para a uma determinada altura. A sua função deve receber a raiz da árvore e retornar 0 se a árvore for completa ou 1 se for incompleta.

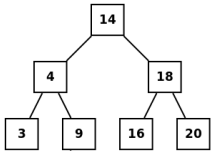

Figure 4: Exemplo de árvore completa

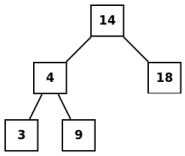

Figure 5: Exemplo de árvore incompleta

10. Tenha como base uma árvore binária organizada de forma que todos os nós à esquerda são menores ou iguais ao nó-pai e todos os nós à direita são maiores que o nó-pai. Crie uma função que cheque se uma árvore binária qualquer obedece a este critério, retornando *true* apenas se todos os nós estiverem de acordo com esse parâmetro.
    Fonte: *Binary Trees by Nick Parlante. Acesso em: 18 abr. 2018.*

11. Dada uma árvore binária, implemente um algoritmo que a transforme em uma lista duplamente encadeada ordenada de acordo com o percurso em-ordem. Assim, o nó à esquerda deve ser o nó anterior e o nó à direita deve ser o nó posterior, enquanto o nó-pai estará entre os dois.
    Nota: não é necessário criar uma nova lista, é possível apenas reorganizar a árvore se for uma árvore duplamente encadeada.

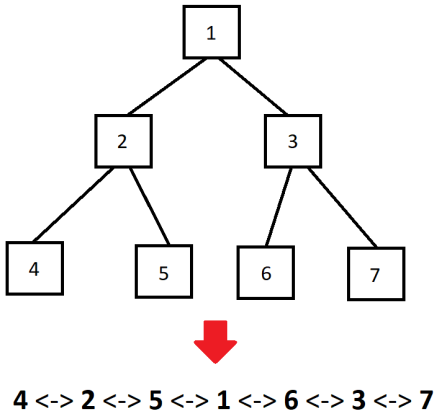

Figure 6: Exemplo da transformação de árvore binária para lista duplamente encadeada

Fonte: *Convert a binary tree to doubly linked list . Acesso em: 18 abr. 2018.*

Desafio

Suponha que você esteja construindo uma árvore de pesquisa binária de um nó N com os valores 1..N. Quantas árvores de pesquisa binária estruturalmente diferentes existem que armazenam esses valores? Escreva uma função recursiva que, dado o número de valores distintos, calcula o número de árvores de pesquisa binária estruturalmente exclusivas que armazenam esses valores. Por exemplo, countTrees (4) deve retornar 14, pois há 14 árvores de pesquisa binária estruturalmente exclusivas que armazenam 1, 2, 3 e 4. O caso base é fácil e a recursão é curta, mas densa. Seu código não deve construir árvores reais; é apenas um problema de contagem.

Fonte: *Binary Trees by Nick Parlante. Acesso em: 15 abr. 2018.*

Solução

1. Para implementar o código, é necessário levar em consideração o seguinte a respeito das funções:

    - *criaRaiz*: Esta função cria o primeiro nó da árvore, ou seja, a raiz. O argumento recebido pela função é o conteúdo inteiro que a raiz irá armazenar. Primeiro, aloque memória para o nó. Atribua à *dado* o conteúdo recebido como argumento. Inicialize os ponteiros para os filhos como nulos. O valor de retorno deve ser o endereço da área de memória alocada para a raiz.
    - *inserirNo*: Esta função cria e atribui um nó como filho à um nó já existe. Os argumentos são o ponteiro do endereço do nó-pai e do conteúdo que preencherá o novo nó. Comece alocando a memória e atribuindo-o como filho esquerdo ou direito do nó-pai. O critério de atribuição fica por sua conta. Inicialize os filhos do novo nó como nulos. O valor de retorno deve ser o endereço da área de memória alocada para o nó.
    - *buscaNo*: Esta função retorna o endereço de memória de determinado nó a partir de certo valor recebido como argumento. Caso encontre um nó com esse valor, retorne o endereço. Caso contrário, retorne NULL. Lembre-se de utilizar recursividade.
    - *liberaNo*: Esta função libera a área de memória alocada para o nó cujo endereço foi recebido como argumento.

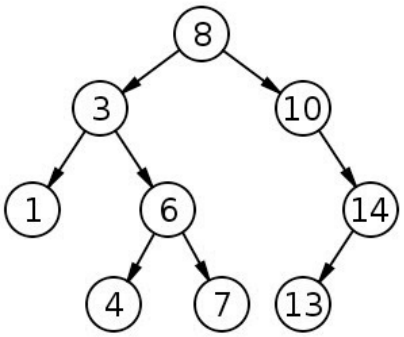

Figure 7: Exemplo teste para questão 2

2. (a) Saída desejada: Endereço do nó com dado 1
   (b) Saída desejada: Endereço do nó com dado 14
   (c) Saída desejada: Endereço do nó com dado 13
   (d) Saída desejada: 9
   (e) Saída desejada: 66
   (f) Saída desejada: 4

3. (a) Percurso em *pré-ordem*: A raiz é processada antes dos nós "filhos", ou seja, visite/processe a raiz, e depois realize um percurso recursivo em pré-ordem em cada uma das subárvores da raiz na ordem definida.
   (b) Percurso em *ordem*: A raiz é visitada na ordem, entre as subárvores. Processa-se a subárvore da esquerda, seguido da raiz e da subárvore à direita.
   (c) Percurso em *pós-ordem*: A raiz é visitada/processada por último. Realize um percurso em pós-ordem em cada uma das subárvores da raiz e depois na raiz.
   (d) Percurso em *largura*: imprime os elementos na ordem da profundidade da árvore, com a ajuda de uma fila.

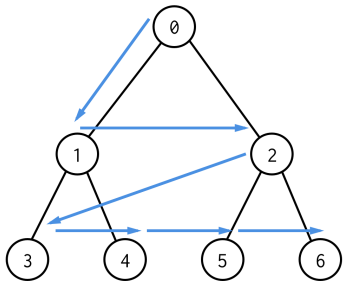

Figure 8: Representação do percurso em largura.

4. Solução presente neste link.
5. Para a função *buscaPai*() utilize um algoritmo de percorrimento da árvore de sua preferência, porém verifique os nós-filhos ao invés do nó atual na recursividade, utilizando isto como método de parada. Bastará retornar o endereço atual. Para a função de remoção, primeiro chame a função *buscaPai*() usando o valor recebido também como argumento desta. Libere a memória do nó-filho determinado e atribua nulo ao ponteiro correspondente do pai.
6. Solução presente neste link.
7. Solução presente neste link.
8. Solução presente neste link.
9. O caso teste pode ser o exemplo já mostrado na questão.
10. Solução presente neste link.
11. Algoritmo e uma solução passo-a-passo em Java neste link.