Teoria dos Grafos

David Helmer Cândido, Henrique Oliveira Waisman, Luís Henrique Gomes Zortea

1. Introdução

Tema Escolhido:

b) Implementar uma rotina que receba uma árvore e permita:
   a. contar o número de nós de uma arvore
   b. contar o número de nós de NÃO folhas uma arvore
   c. Escreva uma função LOCALIZAR um valor em uma árvore binária ordenada
   d. Escreva uma função EXCLUIR um valor em uma árvore binária ordenada. A mesma deve continuar ordenada após remoção

Informações Técnicas: O trabalho foi feito na linguagem C, utilizando a IDE Visual Studio Code (1.87.2) com as extensões ‘C/C++’, ‘Code Runner’ e ‘CMake’.

2. Desenvolvimento

Antes de implementarmos qualquer rotina, faz-se necessária a criação da árvore binária, para tal, declaramos um ‘struct’, tipo de dado (datatype) da linguagem C que pode juntar ‘n’ variáveis, permitindo que essas diferentes variáveis possam ser acessadas via ponteiro (tipo de dado da linguagem C cujo valor aponta para um endereço de memória onde o valor real está alocado).

Cada struct será um nó da árvore binária, por sua vez, cada nó de uma árvore binária guarda um valor e dois ponteiros, um para o filho da direta, outro para o filho da esquerda. Levando para o código:

```
typedef struct treenode {
  int value;
  struct treenode *left;
  struct treenode *right;
} treenode;
```

Nota: typedef é uma palavra-chave da linguagem usada para dar um nome alternativo a um datatype, facilitando sua declaração posteriormente.

Como descrito, o nó ‘treenode’ guarda um valor ‘value’, um filho da esquerda ‘*left’ e um filho da direita ‘*right’. Nota-se que ‘*left’ declara um ponteiro que aponta para um endereço de memória que guarda um datatype ‘struct treenode’, o mesmo comportamento pode ser observado no filho da direita ‘*right’.

Com a estrutura da árvore criada, é necessário popular ela com nós para implementarmos as rotinas. Para tal, é preciso alocar um espaço de memória para o nó e dentro desse espaço de memória alocar o valor relativo ao nó. Em código:

```
treenode *createnode(int value) {
  treenode *result = malloc(sizeof(treenode));
  if (result != NULL) {
    result->left = NULL;
    result->right = NULL;
    result->value = value;
  }
  return result;
}
```

A função tem como tipo de dado o próprio nó, recebe o valor ‘value’ como argumento e retorna um ponteiro para um nó. A função ‘malloc’ está presente na biblioteca “stdlib”, e permite alocar um número específico de bytes, nesse caso o tamanho do nó, manualmente. Após a alocação do nó no endereço de memória, é criado um filho da esquerda e da direita com valores nulos e o valor inserido na função é atribuído ao nó.

Com o primeiro nó criado, para popular o restante da árvore é preciso de uma função para inserir nós de forma ordenada seguindo a estrutura de uma árvore binária. Seguindo o código abaixo:

```
bool insertnumber(treenode **rootptr, int value){
  treenode *root = *rootptr;
  if (root == NULL) {
    (*rootptr) = createnode(value);
    return true;
  }
  if (value == root->value) {
    return false;
  }
  if (value < root->value) {
    return insertnumber(&(root->left), value);
  } else {
    return insertnumber(&(root->right), value);
  }
}
```

Temos aqui uma função booleana de inserção que retorna se o nó foi (true) ou não (false) inserido. A função recebe como parâmetros o valor que será inserido na árvore e um ponteiro para a raiz, ou seja, um ponteiro que guarda o endereço da raiz. Em suma, o artifício de “ponteiro de ponteiro” permite operações na raiz em si, e não em seu conteúdo, facilitando a manipulação da árvore.

Dividindo a função, na primeira condicional (if), é tratada a possibilidade de o valor do nó ser nula, portanto, é chamada a função ‘createnode’, explicada acima, e o nó é inserido. Na segunda condicional, caso o valor a ser inserido seja igual a raiz atual é retornado um valor booleano falso, pois árvores binárias não admitem valores repetidos. E por fim, na terceira condicional, há uma iteração na árvore, buscando o local onde o nó será posto, caso o valor a ser inserido seja menor que o nó atual, a iteração caminha para o filho da esquerda e caso o valor seja maior, a iteração caminha para o filho da direita, mantendo a estrutura da árvore binária de ‘filho da esquerda < raiz < filho da direita’.

Com a estrutura da árvore desenvolvida, é possível criar uma árvore base para implementar as rotinas, a árvore base será feita composta por 6 nós, sendo eles:

```
treenode *root = createnode(15);
insertnumber(&root, 11);
insertnumber(&root, 24);
insertnumber(&root, 5);
insertnumber(&root, 19);
insertnumber(&root, 16);
```

A função ‘createnode’ é empregada para criar o nó raiz que recebe o valor “15” e a função ‘insertnumber’ é usada para criar os outros cinco nós a partir da raiz com seus valores respectivos “11”, “24”, “5”, “19” e “16”.

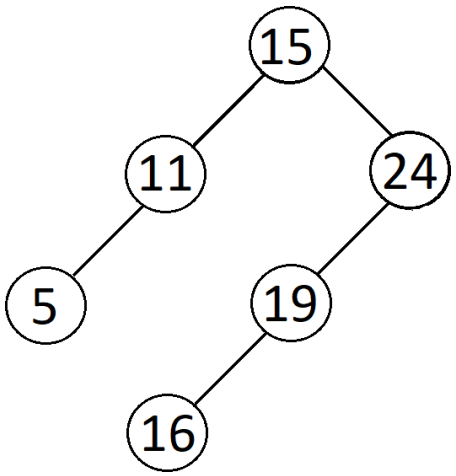
Para melhor visualização, foram criadas também funções para imprimir a árvore de maneira indentada:

```
//printa a árvore em pré ordem: raiz, esquerda, direita, usando recursão
void printtree_rec(treenode *root, int level) {
  if (root == NULL) {
    printtabs(level);
    printf("--<empty>--\n");
    return;
  }
  printtabs(level);
  printf("value = %d\n", root->value);
  printtabs(level);
  printf("left\n");
  printtree_rec(root->left, level + 1);
  printtabs(level);
  printf("right\n");
  printtree_rec(root->right, level +1);
  printtabs(level);
  printf("done\n");
}
//printa a árvore por níveis
void printtree(treenode* root){
printtree_rec(root, 0);
}
```

Usando a função “printtree”, obtemos o resultado:

```
value = 15
left
        value = 11
        left
                value = 5
                left
                        --<empty>--
                right
                        --<empty>--
                done
        right
                --<empty>--
        done
right
        value = 24
        left
                value = 19
                left
                        value = 16
                        left
                                --<empty>--
                        right
                                --<empty>--
                        done
                right
                        --<empty>--
                done
        right
                --<empty>--
        done
done
```

Em uma representação gráfica, temos:



Agora, com a árvore pronta, podemos continuar para a implementação dos exercícios.

3. Conclusões e resultados

a. contar o número de nós de uma árvore

```
int numOfNodes(struct treenode* root){
    if(root == NULL){
      return 0;
    }
    int rightNodes = numOfNodes(root->right);
    int leftNodes = numOfNodes(root->left);
    return rightNodes+leftNodes+1;
}
```

A função “numOfNodes” retorna um número inteiro de nós e recebe como parâmetro a raiz da árvore. A partir da raiz, há um tratamento na condicional, caso a árvore seja nula, o número de nós é 0 e o número é retornado. Caso a árvore não seja nula, a soma do número de nós é dividida em duas variáveis, um para a os nós da direita “rightNodes” e outros para os nós da esquerda “leftNodes”, após percorrer iterativamente toda a arvore, o número de nós à esquerda são somados com o número de nós a direita e mais um é acrescentado para contar a raiz da árvore. Segue abaixo a execução:

```
int main(void) {
  //Construindo a árvore
  treenode *root = createnode(15);
  insertnumber(&root, 11);
  insertnumber(&root, 24);
  insertnumber(&root, 5);
  insertnumber(&root, 19);
  insertnumber(&root, 16);
  //Usando a função
  printf("A árvore possui um total de %i nós\n", numOfNodes(root));
  return 0;
}
```

Saída no console:

```
A árvore possui um total de 6 nós
```

b. contar o número de nós de NÃO folhas uma arvore

```
int NonLeaf(treenode *root) {
    if (root == NULL){
      return 0;
    }
    else if (root->left == NULL && root->right == NULL){
      return 0;
    }
    else{
      return 1 + NonLeaf(root->left) + NonLeaf(root->right);
    }
}
```

A função também retorna um número inteiro de nós, com 3 condicionais. Na primeira condicional, caso a árvore seja nula é retornado o número 0. Na segunda tratativa, caso o nó não tenha filhos nem à esquerda nem à direita, também é retornado 0, visto que um nó sem filhos é um nó folha. E na terceira tratativa há a contagem iterativa de nós somado com 1 (nó raiz), a iteração percorre toda a árvore e quando a função chega em um nó folha, a segunda tratativa é acionada e o número 0 é retornado, fazendo com que a soma apenas conte os nós não folhas. Executando:

```
int main(void) {

  // Construindo a árvore
  treenode *root = createnode(15);
  insertnumber(&root, 11);
  insertnumber(&root, 24);
  insertnumber(&root, 5);
  insertnumber(&root, 19);
  insertnumber(&root, 16);
  //Usando a função
  printf("A árvore possui um total de %i nós não folhas", NonLeaf(root));
  return 0;
}
```

Saída no console:

```
A árvore possui um total de 4 nós não folhas
```

c. Escreva uma função LOCALIZAR um valor em uma árvore binária ordenada

```
bool findnumber(treenode * root, int value){
  if(root == NULL){
    return false;
  }
  if(root->value == value){
    return true;
  }
  if(value < root->value){
    return findnumber(root->left, value);
  }
  else{
    return findnumber(root->right, value);
  }
}
```

Esta função recebe a raiz da árvore, o valor a ser localizado e retorna um booleano, caso o número seja localizado retorna “true”, caso não seja, “false”. O código da função é dividido em quatro condicionais. As duas últimas condicionais percorrem a árvore binária de forma que, caso o número buscado seja menor que o valor do nó atual, a função caminha iterativamente para a esquerda e, caso o número buscado seja maior que número atual, a função caminha iterativamente para a direita. Observando as duas primeiras tratativas, caso o nó atual seja nulo é retornado falso, pois toda a árvore foi percorrida e o número não foi achado e, caso o número buscar seja igual ao do nó atual, é retornado verdadeiro e o número buscado foi achado. Executando este código, temos:

```
int main(void) {
  // Construindo a árvore
  treenode *root = createnode(15);
  insertnumber(&root, 11);
  insertnumber(&root, 24);
  insertnumber(&root, 5);
  insertnumber(&root, 19);
  insertnumber(&root, 16);
  //Usando a função
  printf("%d (%d)\n", 16, findnumber(root, 16));
  printf("%d (%d)\n", 55, findnumber(root, 55));
```

Saída no console:

```
16 (1)
55 (0)
```

O número 16 retornou verdadeiro, portanto, está na árvore. O

número 55 retornou falso, portanto, não estão na árvore.

d. Escreva uma função EXCLUIR um valor em uma árvore binária ordenada. A mesma deve continuar ordenada após remoção

```
treenode *findMinNode(treenode *node) {
    while (node->left != NULL)
        node = node->left;
    return node;
}

treenode *removeNode(treenode *root, int node) {
    if (root == NULL){
        return root;
    }
    if(node < root->value){
      root->left = removeNode(root->left, node);
    }
    else if(node > root->value){
        root->right = removeNode(root->right, node);
    }
    else{
        if(root->left == NULL){
            treenode *temp = root->right;
            free(root);
            return temp;
        }
        else if(root->right == NULL){
            treenode *temp = root->left;
            free(root);
            return temp;
        }
        treenode *temp = findMinNode(root->right);
        root->value = temp->value;
        root->right = removeNode(root->right, temp->value);
    }
    return root;
}
```

Para implementar tal rotina foram necessárias duas funções. Começando pela segunda, “removeNode” é a função que remove o nó e faz o tratamento da organização da árvore pós remoção. Ela começa com um tratamento que retorna a raiz caso ela seja nula ou o nó alvo não seja encontrado. Após o primeiro tratamento, o no é procurado na árvore nas próximas duas condicionais com iterações onde, se o valor procurado for menor que o nó atual a função caminha para o filho da esquerda e caso contrário, ele caminha para a direita. Após achar o valor, chega o momento de remover o nó (procedimento dentro da condicional “else”). Para remover o nó alvo, o programa checa três possibilidades, se o nó alvo tem apenas filho da direita, ele é substituído pelo filho da direita, se o nó alvo tem apenas filho da esquerda, ele é substituído pelo filho da esquerda, e

caso o nó alvo tenha ambos os filhos, a tratativa é diferente. Segue o trecho de código abaixo:

```
treenode *temp = findMinNode(root->right);
root->value = temp->value;
root->right = removeNode(root->right, temp->value);
```

Caso um nó tenha mais de um filho e/ou tenha uma sub-árvore com mais de um nó, quando esse nó for removido, por definição, ele será substituído pelo maior nó da sub-árvore da esquerda OU pelo menor nó da sub-árvore da direita. Nesse caso, foi escolhido o menor nó da sub-árvore da direta para ser feita a substituição. Para acharmos o menor no da sub-árvore da direita, temos a função “findMinNode”:

```
treenode *findMinNode(treenode *node) {
    while (node->left != NULL)
        node = node->left;
    return node;
}
```

Esta função recebe um nó como parâmetro e caminha o mais pra esquerda possível, caso apliquemos esta função usando o filho da direita do nó alvo que será removido como argumento, o substituto ideal será retornado. Voltando ao trecho da função principal:

```
treenode *temp = findMinNode(root->right);
root->value = temp->value;
root->right = removeNode(root->right, temp->value);
```

A função “finMinNode” é aplicada ao filho da direita do nó alvo e o resultado é salvo em uma variável temporária “temp”, e na próxima linha o valor temporário é aplicado ao nó. Assim que o valor é aplicado ao nó alvo, a função faz uma iteração com o filho da direita, que passará pelo mesmo processo e será substituído por um valor nulo, removendo assim, o nó alvo e mantendo a estrutura da árvore.

Executando:

```
int main(void) {
  //Construindo  a árvore
  treenode *root = createnode(15);
  insertnumber(&root, 11);
  insertnumber(&root, 24);
  insertnumber(&root, 5);
  insertnumber(&root, 19);
  insertnumber(&root, 16);
  //Usando a função
```

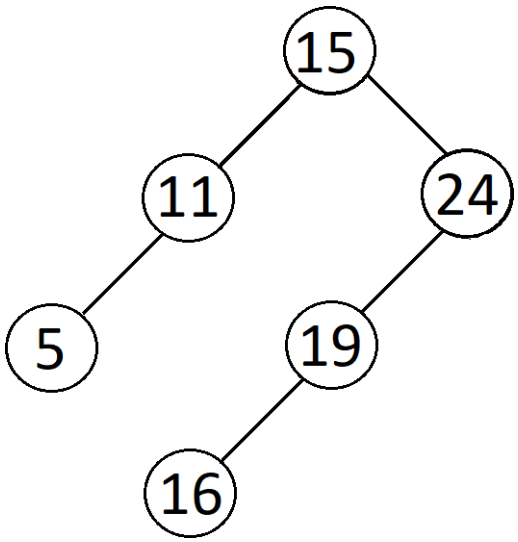
```
  printtree(root);
  removeNode(root, 19);
  printf("\nRemovendo 19\n\n");
  printtree(root);
  return 0;
}
```

Saída do Console Indentada (sem os valores vazios):

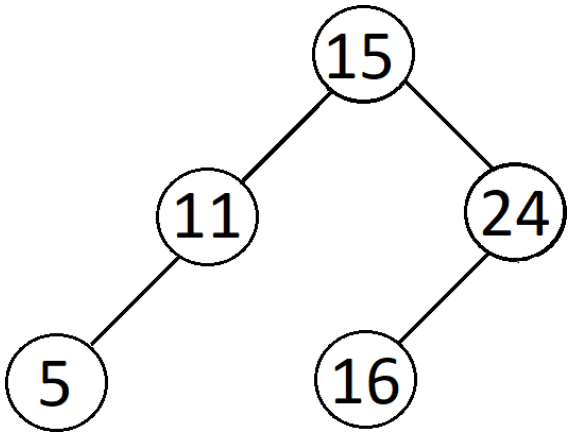
```
value = 15
left
    value = 11
    left
        value = 5
right
    value = 24
    left
        value = 19
        left
            value = 16

Removendo 19

value = 15
left
    value = 11
right
    value = 24
    left
        value = 16
```

Representação de forma gráfica

Antes da remoção:



Após a remoção:



Bibliografia:

[1] Slides de Estrutura de Dados do professor Cássio Capucho (material disponibilizado no 4º período de ciência da computação): https://drive.google.com/drive/folders/1YIfBImMmNaGHRWecGtsucCF5mtWxtODz?usp=sharing

Pesquisa geral:

[2] https://www.ime.usp.br/~pf/mac0122-2003/aulas/bin-trees.html

[3] https://stackoverflow.com/questions/60856852/number-of-nodes-in-bst-in-c

[4] https://stackoverflow.com/questions/33883002/delete-function-in-binary-tree-in-c

[5] https://stackoverflow.com/questions/12131925/getting-the-size-of-a-malloc-only-with-the-returned-pointer

[6] https://www.geeksforgeeks.org/count-non-leaf-nodes-binary-tree/