Algoritmos e Estruturas de Dados 2 - UFSCar

Vinícius O. Guimarães

Prof. Mário César San Felice

Junho 2023

1 Lista 1

Para criar uma árvore binária para resolver os exercícios da lista, vamos precisar dos arquivos Arvore.c e Arvore.h. Dentro do arquivo Arvore.h temos o seguinte conteúdo:

```c
#ifndef ARVORE_H
#define ARVORE_H

#include <stdio.h>
#include <stdlib.h>

typedef struct noh Noh;

typedef struct noh {
   Noh *pai;
   Noh *esq;
   Noh *dir;
   int valor;
} Noh;

Noh * criarArvore();
void inserir(Noh **arvore, int elemento);

int calcularAltura(Noh *arvore);
int calcularAlturaIterativamente(Noh *arvore);

void imprimirArvoreInOrder(Noh *arvore);

int calcularQuantidadeNos(Noh *arvore);
void calcularQuantidadeNosInterno(Noh *arvore, int *pQtd);

void preencherPai(Noh *arvore);
void preencherPaiInterno(Noh *arvore, Noh *pai);

int verificarBinariaDeBusca(Noh *arvore);
void verificarBinariaDeBuscaInterno(Noh *arvore, int *resultado);

int * transformarArvoreEmArray(Noh *arvore, int *tamanho);
void inserirElementosArray(Noh *arvore, int **array, int *posicao, int *tamAtual);

```

```
36 Noh *arrayParaArvoreBalanceada(int *array, int posInicial, int posFinal);
37
38 Noh *procurarPaiDeNoh(Noh *arvore, Noh *noh);
39
40 #endif
```

OBS: Os códigos a seguir estão no arquivo Arvore.c

1. **Escreva uma função que calcule o número de nós de uma árvore binária.**

   Para calcularmos a quantidade de nós de uma árvore binária, podemos percorrer todos os nós recursivamente incrementando um contador de quantidade.

```
1  int calcularQuantidadeNos(Noh *arvore) {
2      // exemplo de um comentário aleatório
3      int qtd = 0;
4      calcularQuantidadeNosInterno(arvore, &qtd);
5      return qtd;
6  }
7
8  void calcularQuantidadeNosInterno(Noh *arvore, int *pQtd) {
9      if (arvore == NULL)  {
10         return;
11     }
12     (*pQtd)++;
13     calcularQuantidadeNosInterno(arvore->esq, pQtd);
14     calcularQuantidadeNosInterno(arvore->dir, pQtd);
15 }
```

2. **Escreva uma função que imprima, em ordem esquerda-raiz-direita, os conteúdos das folhas de uma árvore binária.**

   Para escrever essa função, podemos percorrer a nossa árvore recursivamente começando pela raíz fazendo a seguinte ação para cada nó encontrado:

   - Imprimir o valor do nó
   - Chamar função recursivamente para o nó da esquerda
   - Chamar função recursivamente para o nó da direita

   O resultado fica desta forma:

```
1  void imprimirArvoreInOrder(Noh *arvore) {
2      if (arvore == NULL) {
3          return;
4      }
5      imprimirArvoreInOrder(arvore->esq);
6      printf("%d ", arvore->valor);
7      imprimirArvoreInOrder(arvore->dir);
8  }
```

Assim, como cada nó armazena um valor inteiro, vamos conseguir imprimir o valor de cada um dos nós da árvore binária de busca em ordem crescente.

3. **Escreva uma função para calcular a altura de uma árvore binária.**

- **Forma Recursiva**

  Para calcular a altura de uma árvore começando por um determinado nó (No nosso caso a raiz), será preciso calcular recursivamente a altura de cada uma das sub-árvores da esquerda e da direita desse nó.

  Por isso, quando tivermos a altura de cada uma das sub-árvores, vamos considerar a altura da maior dentre as duas e somar 1 nesse resultado (Pois assim será considerado também o nó inicial/raiz). No final, retorna-se valor da altura.

```c
// Como depende da quantidade de elementos, então é O(n)
int calcularAltura(Noh *arvore) {
    if (arvore == NULL) {
        return -1;
    }
    int alturaEsquerda = calcularAltura(arvore->esq);
    int alturaDireita = calcularAltura(arvore->dir);
    if (alturaEsquerda > alturaDireita) {
        return alturaEsquerda + 1;
    }
    return alturaDireita + 1;
}
```

- **Forma iterativa**

  Para conseguirmos calcular a altura de uma árvore na forma iterativa, vamos precisar de uma fila.

  Essa fila será necessária para percorrer todos os nós de uma determinada altura por vez, adicionando, para cada nó dessa altura, os seus filhos dentro da fila.

  Temos aqui o código da fila separado nos arquivos fila.h e fila.c:

  Arquivo fila.h:

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct node Node;

typedef struct fila {
   int tam;
   Node *primeiro;
   Node *ultimo;
} Fila;

typedef struct node {
   int valor;
   Node *proximo;
} Node;

Fila *criarFila();
Node *criarNode(int valor);
```

```c
    void inserirNaFila(Fila *fila, int valor);
    void imprimirFila(Fila *fila);
    int pegarPrimeiroDaFila(Fila *fila);
    Node *removerPrimeiroDaFila(Fila *fila);
```

Arquivo fila.c:

```c
#include "fila.h"

Fila *criarFila() {
    Fila *fila = (Fila *) malloc(sizeof(Fila));
    fila->tam = 0;
    fila->primeiro = NULL;
    fila->ultimo = NULL;
}

Node *criarNode(int valor) {
    Node *node = (Node *) malloc(sizeof(Node));
    node->valor = valor;
    node->proximo = NULL;
    return node;
}

void inserirNaFila(Fila *fila, int valor) {
    Node *novo = criarNode(valor);
    if (fila->tam == 0) {
        fila->primeiro = novo;
        fila->ultimo = novo;
    } else {
        fila->ultimo->proximo = novo;
        fila->ultimo = novo;
    }
    fila->tam++;
}

Node *removerPrimeiroDaFila(Fila *fila) {
    Node *paraRemover = NULL;
    if (fila->tam > 0) {
        paraRemover = fila->primeiro;
        fila->primeiro = paraRemover->proximo;
        fila->tam--;
    }
    return paraRemover;
}

void imprimirFila(Fila *fila) {
    Node *atual = fila->primeiro;
    while (atual != NULL) {
        printf("%d ", atual->valor);
        atual = atual->proximo;
```

```
        }
    }
```

Agora com a fila pronta, podemos escrever o algoritmo para calcular a altura de uma árvore iterativamente:

```
int calcularAlturaIterativamente(Noh *arvore) {
    if (arvore == NULL) {
        return -1;
    }

    int altura = 0;
    // Função chamada do arquivo Fila.h
    Fila *fila = criarFila();
    inserirNaFila(fila, arvore);

    while (fila->tam > 0) {
        int quantidadeNaFila = fila->tam;
        int temFilho = 0;
        for (int i = 0; i < quantidadeNaFila; i++) {
            Node *primeiroNohFila = removerPrimeiroDaFila(fila);
            if (primeiroNohFila->conteudo->esq != NULL) {
                temFilho = 1;
                inserirNaFila(fila, primeiroNohFila->conteudo->esq);
            }
            if (primeiroNohFila->conteudo->dir != NULL) {
                temFilho = 1;
                inserirNaFila(fila, primeiroNohFila->conteudo->dir);
            }
            // Observe que quando fazemos free no nó (node) da
            // lista, não estamos fazendo free no seu conteúdo.
            free(primeiroNohFila);
        }
        // Se algum dos nós do nível (altura) atual tem filho.
        if (temFilho == 1) {
            altura++;
        }
    }
    free(fila);
    return altura;
}

```

4. **Considere uma árvore binária já construída, mas com os campos pai não preenchidos. Escreva uma função que preencha corretamente todos os campos pai desta árvore**

```
    // Essa função serve apenas como um wrapper (Embrulho) da função
    // que realmente irá realizar o trabalho sujo.
```

```c
void preencherPai(Noh *arvore) {
    preencherPaiInterno(arvore, NULL);
}

// Como depende da quantidade de nós da árvore, então é O(n)
void preencherPaiInterno(Noh *arvore, Noh *pai) {
    if (arvore == NULL) {
        return;
    }
    arvore->pai = pai;
    preencherPaiInterno(arvore->esq, arvore);
    preencherPaiInterno(arvore->dir, arvore);
}
```

5. **Escreva uma função que verifica se uma dada árvore binária é de busca**

Para realizar essa questão temos que percorrer os nós da árvore verificando os valores dos filhos da esquerda e da direita. A árvore só será binária de busca se (Considerando que os filhos não sejam NULL):

- Dado um nó, o valor do seu filho da esquerda deverá ser menor ou igual do que o seu próprio valor.
- Dado um nó, o valor do seu filho da direita deverá ser maior do que o seu próprio valor

```c
// Funciona como um embrulho para a função que irá realizar o trabalho
// de verificar realmente.
int verificarBinariaDeBusca(Noh *arvore) {
    if (arvore == NULL) {
        return 1;
    }
    int resultado = 1;
    verificarBinariaDeBuscaInterno(arvore, &resultado);
    return resultado;
}

void verificarBinariaDeBuscaInterno(Noh *arvore, int *resultado) {
    // No início de cada um dos if's principais estamos verificando se o resultado
    // é diferente de 0, pois caso seja igual a 0, não precisaremos continuar
    // verificando os outros nós

    if (arvore->esq != NULL && (*resultado) != 0) {
        if (arvore->esq->valor > arvore->valor) {
            (*resultado) = 0;
            return;
        }
        verificarBinariaDeBuscaInterno(arvore->esq, resultado);
    }
    if (arvore->dir != NULL && (*resultado) != 0) {
        if (arvore->dir->valor <= arvore->valor) {
            (*resultado) = 0;
            return;
        }
```

```
            verificarBinariaDeBuscaInterno(arvore->dir, resultado);
        }
    }
```

6. **Escreva uma função que transforme uma árvore binária de busca em um vetor crescente.**

Para fazer esse exercício, podemos percorrer a árvore recursivamente, controlando uma posição externa na qual iremos inserir cada um dos elementos.

```

    int *transformarArvoreEmArray(Noh *arvore, int *tamanho) {
        // Por padrão, iremos alocar um espaço na memória do tamanho de 10 inteiros
        int pos = 0;
        int tamAtual = 10;
        int *array = (int *) malloc(tamAtual * sizeof(int));
        inserirElementosArray(arvore, &array, &pos, &tamAtual);
        // Pode ser que alocamos um tamanho de 10 inteiros, mas
        // somente 4 posições foram ocupadas e portanto, o resto das posições é lixo.
        // Precisamos informar o tamanho real, que é 4.
        // Se tivermos apenas um elemento na árvore, então a posição atual é 0,
        // em seguida, inserimos esse único elemento no array e incrementamos a
        // posição atual para 1.
        // Ou seja, sempre no final a posição irá representar a quantidade real de
        // elementos dentro do array
        (*tamanho) = pos;
        return array;
    }

    void inserirElementosArray(Noh *arvore, int **array, int *posicao, int *tamAtual) {
        if (arvore == NULL) {
            return;
        }
        inserirElementosArray(arvore->esq, array, posicao, tamAtual);
        if ((*posicao) >= (*tamAtual)) {
            // Observe que sempre que o array fica cheio nós duplicamos o seu tamanho
            (*tamAtual) = (*tamAtual) * 2;
            (*array) = realloc(*array, (*tamAtual) * sizeof(int));
        }
        (*array)[(*posicao)] = arvore->valor;
        (*posicao)++;
        inserirElementosArray(arvore->dir, array, posicao, tamAtual);
    }
```

Assim, para testarmos essa função, podemos ir no arquivo main e escrever algo dessa forma

```
    // Representa a quantidade de elementos inseridos dentro do array
    int tamanho;
    int *array = transformarArvoreEmArray(arvore, &tamanho);
    printf("\nImprimindo array\n");
    for (int i = 0; i < tamanho; i++) {
```

```
        printf("%d ", array[i]);
    }
```

Devemos deixar claro que nesse instante, a variável tamanho irá guardar a quantidade de elementos que está dentro do array e não o tamanho total do array.

O que quero dizer é: O array tem seu espaço de memória total incrementado de 10 em 10 posições para economizar processamento do realloc e, portanto, podemos ter o array com 20 posições totais mas somente 14 elementos dentro dele, fazendo com que essa variável tamanho no arquivo main seja 14 (Quantidade de elementos dentro do array).

7. **Escreva uma função que transforme um vetor crescente em uma árvore binária de busca que seja balanceada.**

   No código abaixo, estamos verificando os casos onde posFinal é menor do que posInicial e posInicial é maior do que posFinal.

   Imagina que temos um array com um único elemento: [8].

   Com isso, a nossa posição inicial é 0 e a posição final também é 0. Portanto, o meio será:

$$meio = (posFinal - posInicial)/2 + posInicial = (0 - 0)/2 + 0 = 0.$$

   Ou seja, teremos uma árvore com raiz sendo o elemento 8 e os sub-nós da esquerda e direita dessa árvore serão : calculados com as seguintes posições posInicial e posFinal:

   - Para o sub-nó da esquerda:
     - posInicial = 0 (Continua a mesma coisa)
     - posFinal = meio - 1 = 0 - 1 = -1
   - Para o sub-nó da direita:
     - posInicial = meio + 1 = 0 + 1 = 1
     - posFinal = 0 (Continua a mesma coisa)

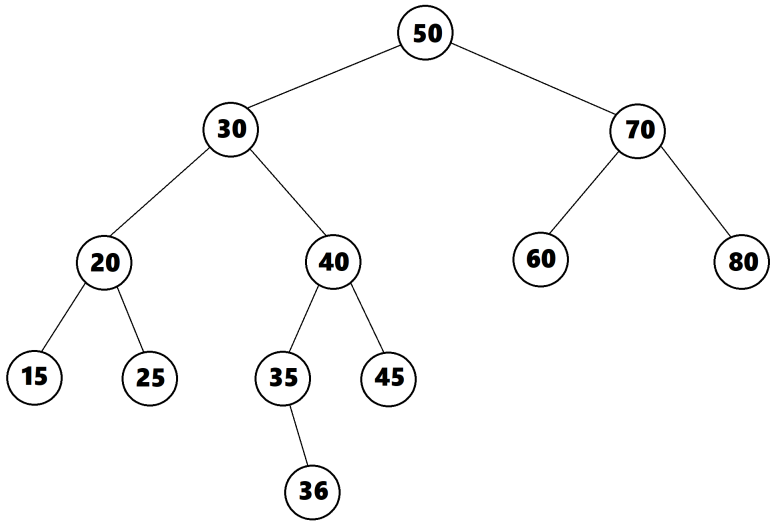
   Como podemos perceber, no caso do sub-nó da esquerda, a posição final será menor do que a posição inicial e portanto iremos retornar esse sub-nó da esquerda como sendo NULL.

   De forma semelhante, no caso do sub-nó da direita, a posição inicial será maior do que a posição final e portanto também iremos retornar esse sub-nó da direita como sendo NULL.

```
Noh *arrayParaArvoreBalanceada(int *array, int posInicial, int posFinal) {
    if (posFinal < posInicial || posInicial > posFinal) {
        return NULL;
    }
    int meio = (posFinal - posInicial) / 2 + posInicial;
    Noh *noh = novoNoh(array[meio]);
    noh->esq = arrayParaArvoreBalanceada(array, posInicial, meio - 1);
    noh->dir = arrayParaArvoreBalanceada(array, meio + 1, posFinal);
    return noh;
}
```

8. **Suponha que as chaves 50 30 70 20 40 60 80 15 25 35 45 36 são inseridas, nesta ordem, numa árvore de busca inicialmente vazia. Desenhe a árvore que resulta. Em seguida remova o nó que contém 30.**

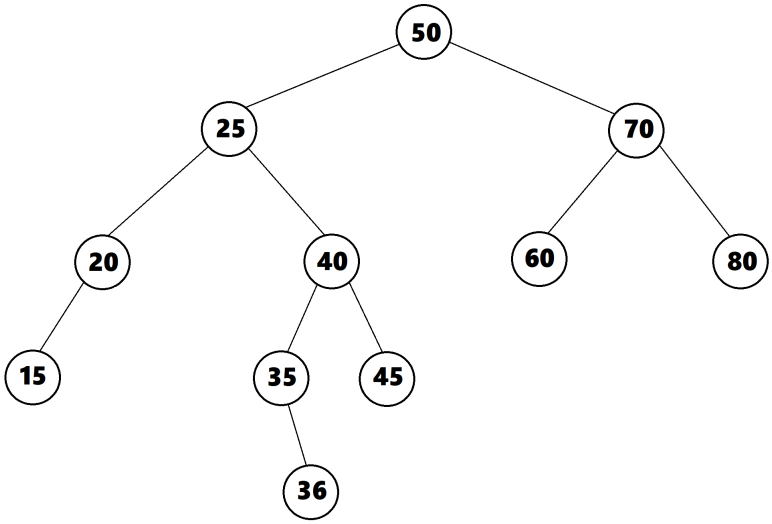
   Árvore formada após a inserção dos elementos:



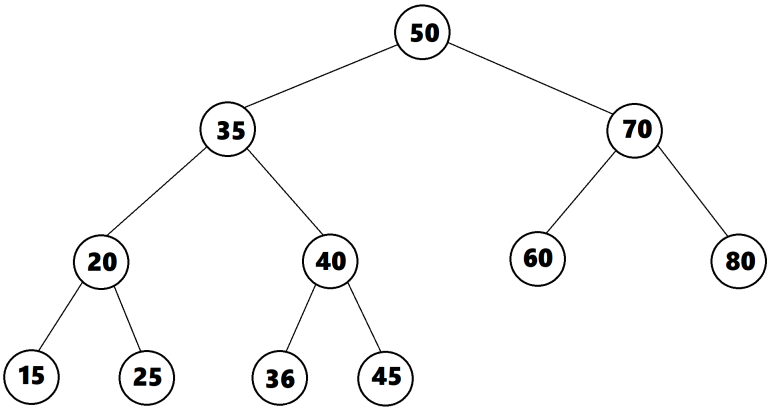
Como sabemos, no caso da remoção do elemento 30, como ele tem 2 filhos, então temos duas possibilidades:

- Podemos substituir o nó 30 pelo nó da sub-árvore da esquerda que contém o maior valor.
  ou
- Podemos substituir o nó 30 pelo nó da sub-árvore da direita que contém o meor valor.

Considerando que substituímos o nó 30 pelo maior nó da sub-árvore da esquerda, a árvore ficará assim:



Já se substituírmos o nó 30 pelo menor elemento da sub-árvore da direita, então ficaremos com a seguinte árvore:



Observe que, ao substituir o nó 30 pelo nó 35, foi preciso arrumar os filhos que eram do nó 35 para que a árvore continue sendo binária. Nesse caso, o nó 36 passou a ser um filho direto na esquerda do nó 40.

9. **Considere árvores binárias de busca cujos nós têm a estrutura indicada abaixo. Escreva uma função que receba a raiz de uma tal árvore e o endereço de um nó x e devolva o endereço do pai de x.**

```
typedef struct reg {
 int chave;
 int conteudo;
 struct reg *esq, *dir;
} noh;
```

Para resolver esse problema, podemos percorrer a árvore recursivamente e verificar para cada nó, se o valor do filho da esquerda ou da direita é igual ao valor do noh que estamos procurando, para que caso for, retornar o nó atual que estamos.

```
Noh *procurarPaiDeNoh(Noh *arvore, Noh *noh) {
    if (arvore == NULL || arvore->valor == noh->valor) {
        return NULL;
    }
    if  (arvore->esq->valor == noh->valor || arvore->dir->valor == noh->valor) {
        return arvore;
    }
    if (noh->valor < arvore->valor) {
        return procurarPaiDeNoh(arvore->esq, noh);
    }
    if (noh->valor > arvore->valor) {
        return procurarPaiDeNoh(arvore->dir, noh);
    }
}
```

2 Todos os códigos feitos

1. **Arquivo Arvore.h**

```
#ifndef ARVORE_H
#define ARVORE_H

#include <stdio.h>
#include <stdlib.h>

typedef struct noh Noh;

typedef struct noh {
   Noh *pai;
   Noh *esq;
   Noh *dir;
   int valor;
} Noh;

Noh * criarArvore();
void inserir(Noh **arvore, int elemento);
```

```
18
19      int calcularAltura(Noh *arvore);
20      int calcularAlturaIterativamente(Noh *arvore);
21
22      void imprimirArvoreInOrder(Noh *arvore);
23
24      int calcularQuantidadeNos(Noh *arvore);
25      void calcularQuantidadeNosInterno(Noh *arvore, int *pQtd);
26
27      void preencherPai(Noh *arvore);
28      void preencherPaiInterno(Noh *arvore, Noh *pai);
29
30      int verificarBinariaDeBusca(Noh *arvore);
31      void verificarBinariaDeBuscaInterno(Noh *arvore, int *resultado);
32
33      int * transformarArvoreEmArray(Noh *arvore, int *tamanho);
34      void inserirElementosArray(Noh *arvore, int **array, int *posicao, int *tamAtual);
35
36      Noh *arrayParaArvoreBalanceada(int *array, int posInicial, int posFinal);
37
38      Noh *procurarPaiDeNoh(Noh *arvore, Noh *noh);
39
40      #endif
```

2. **Arquivo Arvore.c**

```
1   #include "Arvore.h"
2   #include "Fila.h"
3
4   Noh *criarArvore() {
5       return NULL;
6   }
7
8   Noh * novoNoh(int valor) {
9       Noh *novo = (Noh *) malloc(sizeof(Noh));
10      novo->dir = NULL;
11      novo->esq = NULL;
12      novo->pai = NULL;
13      novo->valor = valor;
14      return novo;
15  }
16
17  void inserir(Noh **arvore, int elemento) {
18      if ((*arvore) == NULL) {
19          *arvore = novoNoh(elemento);
20      } else if (elemento <= (*arvore)->valor){
21          inserir(&(*arvore)->esq, elemento);
22      } else {
23          // elemento > (*arvore)->valor
24          inserir(&(*arvore)->dir, elemento);
```

```
25          }
26      }
27
28      // Como depende da quantidade de elementos, então é O(n)
29      int calcularAltura(Noh *arvore) {
30          if (arvore == NULL) {
31              return -1;
32          }
33          int alturaEsquerda = calcularAltura(arvore->esq);
34          int alturaDireita = calcularAltura(arvore->dir);
35          if (alturaEsquerda > alturaDireita) {
36              return alturaEsquerda + 1;
37          }
38          return alturaDireita + 1;
39      }
40
41      int calcularAlturaIterativamente(Noh *arvore) {
42          if (arvore == NULL) {
43              return -1;
44          }
45
46          int altura = 0;
47          // Função chamada do arquivo Fila.h
48          Fila *fila = criarFila();
49          inserirNaFila(fila, arvore);
50
51          while (fila->tam > 0) {
52              int quantidadeNaFila = fila->tam;
53              int temFilho = 0;
54              for (int i = 0; i < quantidadeNaFila; i++) {
55                  Node *primeiroNohFila = removerPrimeiroDaFila(fila);
56                  if (primeiroNohFila->conteudo->esq != NULL) {
57                      temFilho = 1;
58                      inserirNaFila(fila, primeiroNohFila->conteudo->esq);
59                  }
60                  if (primeiroNohFila->conteudo->dir != NULL) {
61                      temFilho = 1;
62                      inserirNaFila(fila, primeiroNohFila->conteudo->dir);
63                  }
64                  // Observe que quando fazemos free no nó (node) da
65                  // lista, não estamos fazendo free no seu conteúdo.
66                  free(primeiroNohFila);
67              }
68              // Se algum dos nós do nível (altura) atual tem filho.
69              if (temFilho == 1) {
70                  altura++;
71              }
72          }
73          free(fila);
74          return altura;
```

```
75      }
76
77      void imprimirArvoreInOrder(Noh *arvore) {
78          if (arvore == NULL) {
79              return;
80          }
81          imprimirArvoreInOrder(arvore->esq);
82          if (arvore->esq == NULL && arvore->dir == NULL) {
83              printf("%d ", arvore->valor);
84          }
85          imprimirArvoreInOrder(arvore->dir);
86      }
87
88      int calcularQuantidadeNos(Noh *arvore) {
89          int qtd = 0;
90          calcularQuantidadeNosInterno(arvore, &qtd);
91          return qtd;
92      }
93
94      void calcularQuantidadeNosInterno(Noh *arvore, int *pQtd) {
95          if (arvore == NULL)  {
96              return;
97          }
98          (*pQtd)++;
99          calcularQuantidadeNosInterno(arvore->esq, pQtd);
100         calcularQuantidadeNosInterno(arvore->dir, pQtd);
101     }
102
103     // Essa função serve apenas como um wrapper (Embrulho) da função
104     // que realmente irá realizar o trabalho sujo.
105     void preencherPai(Noh *arvore) {
106         preencherPaiInterno(arvore, NULL);
107     }
108
109     // Como depende da quantidade de nós da árvore, então é O(n)
110     void preencherPaiInterno(Noh *arvore, Noh *pai) {
111         if (arvore == NULL) {
112             return;
113         }
114         arvore->pai = pai;
115         preencherPaiInterno(arvore->esq, arvore);
116         preencherPaiInterno(arvore->dir, arvore);
117     }
118
119     // Funciona como um embrulho para a função que irá realizar o trabalho
120     // de verificar realmente.
121     int verificarBinariaDeBusca(Noh *arvore) {
122         if (arvore == NULL) {
123             return 1;
124         }
```

```
125         int resultado = 1;
126         verificarBinariaDeBuscaInterno(arvore, &resultado);
127         return resultado;
128     }
129
130     void verificarBinariaDeBuscaInterno(Noh *arvore, int *resultado) {
131         // No início de cada um dos if's principais estamos verificando se o resultado
132         // é diferente de 0, pois caso seja igual a 0, não precisaremos continuar
133         // verificando os outros nós
134
135         if (arvore->esq != NULL && (*resultado) != 0) {
136             if (arvore->esq->valor > arvore->valor) {
137                 (*resultado) = 0;
138                 return;
139             }
140             verificarBinariaDeBuscaInterno(arvore->esq, resultado);
141         }
142         if (arvore->dir != NULL && (*resultado) != 0) {
143             if (arvore->dir->valor <= arvore->valor) {
144                 (*resultado) = 0;
145                 return;
146             }
147             verificarBinariaDeBuscaInterno(arvore->dir, resultado);
148         }
149     }
150
151     int *transformarArvoreEmArray(Noh *arvore, int *tamanho) {
152         // Por padrão, iremos alocar um espaço na memória do tamanho de 10 inteiros
153         int pos = 0;
154         int tamAtual = 10;
155         int *array = (int *) malloc(tamAtual * sizeof(int));
156         inserirElementosArray(arvore, &array, &pos, &tamAtual);
157         // Pode ser que alocamos um tamanho de 10 inteiros, mas
158         // somente 4 posições foram ocupadas e portanto, o resto das posições é lixo.
159         // Precisamos informar o tamanho real, que é 4.
160         // Se tivermos apenas um elemento na árvore, então a posição atual é 0,
161         // em seguida, inserimos esse único elemento no array e incrementamos a
162         // posição atual para 1.
163         // Ou seja, sempre no final a posição irá representar a quantidade real de
164         // elementos dentro do array
165         (*tamanho) = pos;
166         return array;
167     }
168
169     void inserirElementosArray(Noh *arvore, int **array, int *posicao, int *tamAtual) {
170         if (arvore == NULL) {
171             return;
172         }
173         inserirElementosArray(arvore->esq, array, posicao, tamAtual);
174         if ((*posicao) >= (*tamAtual)) {
```

```
            // Observe que sempre que o array fica cheio nós duplicamos o seu tamanho
            (*tamAtual) = (*tamAtual) * 2;
            (*array) = realloc(*array, (*tamAtual) * sizeof(int));
        }
        (*array)[(*posicao)] = arvore->valor;
        (*posicao)++;
        inserirElementosArray(arvore->dir, array, posicao, tamAtual);
    }

    Noh *arrayParaArvoreBalanceada(int *array, int posInicial, int posFinal) {
        if (posFinal < posInicial || posInicial > posFinal) {
            return NULL;
        }
        int meio = (posFinal - posInicial) / 2 + posInicial;
        Noh *noh = novoNoh(array[meio]);
        noh->esq = arrayParaArvoreBalanceada(array, posInicial, meio - 1);
        noh->dir = arrayParaArvoreBalanceada(array, meio + 1, posFinal);
        return noh;
    }

    Noh *procurarPaiDeNoh(Noh *arvore, Noh *noh) {
        if (arvore == NULL || arvore->valor == noh->valor) {
            return NULL;
        }
        if  (arvore->esq->valor == noh->valor || arvore->dir->valor == noh->valor) {
            return arvore;
        }
        if (noh->valor < arvore->valor) {
            return procurarPaiDeNoh(arvore->esq, noh);
        }
        if (noh->valor > arvore->valor) {
            return procurarPaiDeNoh(arvore->dir, noh);
        }
    }
```

3. **Arquivo Fila.h**

```
    #ifndef FILA_H
    #define FILA_H

    #include <stdio.h>
    #include <stdlib.h>
    #include "Arvore.h"

    typedef struct node Node;

    typedef struct fila {
       int tam;
       Node *primeiro;
       Node *ultimo;
```

```
14      } Fila;
15
16      typedef struct node {
17         // O conteúdo do nó da fila é um noh da árvore
18         // Ou seja, estamos armazenando um noh da árvore dentro de um nó da fila.
19         Noh *conteudo;
20         Node *proximo;
21      } Node;
22
23      Fila *criarFila();
24      Node *criarNodeFila(Noh *conteudo);
25      void inserirNaFila(Fila *fila, Noh *conteudo);
26      void imprimirFila(Fila *fila);
27      Node *removerPrimeiroDaFila(Fila *fila);
28      void limparFila(Fila *fila);
29      void freeConteudoFila(Fila *fila);
30
31      #endif
```

4. **Arquivo Fila.c**

```
1       #include "Fila.h"
2
3       Fila *criarFila() {
4           Fila *fila = (Fila *) malloc(sizeof(Fila));
5           fila->tam = 0;
6           fila->primeiro = NULL;
7           fila->ultimo = NULL;
8       }
9
10      Node *criarNodeFila(Noh *conteudo) {
11          Node *node = (Node *) malloc(sizeof(Node));
12          node->conteudo = conteudo;
13          node->proximo = NULL;
14          return node;
15      }
16
17      void inserirNaFila(Fila *fila, Noh *nohArvoreParaArmazenar) {
18          Node *novo = criarNodeFila(nohArvoreParaArmazenar);
19          if (fila->tam == 0) {
20              fila->primeiro = novo;
21              fila->ultimo = novo;
22          } else {
23              fila->ultimo->proximo = novo;
24              fila->ultimo = novo;
25          }
26          fila->tam++;
27      }
28
29      Node *removerPrimeiroDaFila(Fila *fila) {
```

```
30          Node *paraRemover = NULL;
31          if (fila->tam > 0) {
32              paraRemover = fila->primeiro;
33              fila->primeiro = paraRemover->proximo;
34              fila->tam--;
35          }
36          return paraRemover;
37      }
38
39      void imprimirFila(Fila *fila) {
40          Node *atual = fila->primeiro;
41          while (atual != NULL) {
42              printf("%d ", atual->conteudo->valor);
43              atual = atual->proximo;
44          }
45      }
46
47      // Aqui estamos liberando todos os nodes da fila .
48      // Mas não estamos fazendo o free nos conteúdos desses nodes, pois
49      // o conteúdo desses nodes são na verdade ponteiros para nós da árvore
50      void freeConteudoFila(Fila *fila) {
51          while (fila->tam > 0) {
52              Node *primeiro = removerPrimeiroDaFila(fila);
53              free(primeiro);
54          }
55      }
```