UNIVERSIDADE FEDERAL DA PARAÍBA
CENTRO DE INFORMÁTICA

Disciplina: Organização e Recuperação de Dados
Semestre: 2020.2
Professor: Leandro Carlos de Souza
Data: 21/04/2021
Nome: Thaís Gabrielly Marques de Andrade
Matrícula: 20180135293

Exercícios de Fixação e Aprendizagem I

Questão 1 (1,0 pt)

Sobre árvores binárias de busca, e utilizando suas próprias palavras:

(a) Conceitue o que são árvores binárias de busca.

São estruturas de dados, composta por nós. Onde todos os nós da subárvore da esquerda são menores que nó raiz. E os nós da subárvore da direita são superiores ao nó raiz.

(b) O que as diferencia das árvores binárias tradicionais?

Pela forma que é ordenada e suas condições. Comparando, por exemplo, com uma árvore estritamente binária que sua condição é que cada nó possua de 0 a 2 filhos.

Questão 2 (2,0 pts)

Sobre árvores AVL, e utilizando suas próprias palavras:

(a) Conceitue o que são árvores AVL.

São árvores binárias de busca balanceadas. Para que uma árvore seja balanceada é preciso que as suas subárvores possuam aproximadamente a mesma altura.

(b) O que as diferencia das árvores de busca tradicionais?

Diferencia por conta que a árvore AVL minimiza o número de comparações executadas no pior caso, por conta de serem balanceadas. Mas para garantir que continuem balanceadas é preciso efetuar operações de: rotação direita, rotação esquerda, rotação dupla direita e rotação dupla esquerda. Com isso o custo mínimo tende a O (log n).

(c) Descreva em quais casos as rotações que são aplicadas para ajustar árvores AVL.

Em uma árvore AVL qualquer será inseridos n nós. Como manter a estrutura balanceada após a execução dessa parte do algoritmo? Bom, é preciso verificar se algum nó se encontra desregulado, no caso se a diferença de altura entre as duas subárvores ficou maior do que um. Caso aconteça, é preciso ajustar para que a árvore volte a ser balanceada, utilizando as quatro rotações.

Vejamos os casos onde são aplicadas:

Caso 1: h_E (p) > h_D (p), q está na árvore à esquerda da raiz p. E p possui um filho esquerdo u diferente de q. As seguintes possibilidades podem acontecer:

- Se h_E (u) > h_D (u) acontecer, investigando os nós é possível perceber que em algum momento o valor da diferença ira ser igual a 1, então aplica-se a rotação direita da raiz.
- Se h_D (u) > h_E (u) venha a acontecer, investigando a diferença percebe que em algum momento o valor alcançará o resultado menor ou igual a um, para corrigir aplica-se a rotação dupla direita da raiz.

Caso 2: h_D (p) > h_E (p), p tem um filho direito z diferente de q. Logo, as seguintes possibilidades podem ocorrer:

- Se h_D (z) > h_E (z) ocorrer, investigando os nós é possível perceber que a diferença será igual á um. Para resolver, aplica-se a rotação esquerda, assim balanceando a árvore.
- Se h_E (z) > h_D (z) acontecer, investigando nota-se que se z possuir um filho esquerdo a, é preciso aplicar a rotação dupla esquerda para transformar a árvore em AVL novamente.

(d) Simule a construção de uma árvore AVL para a sequência 35, 39, 51, 20, 13, 28, 22 (nesta ordem).

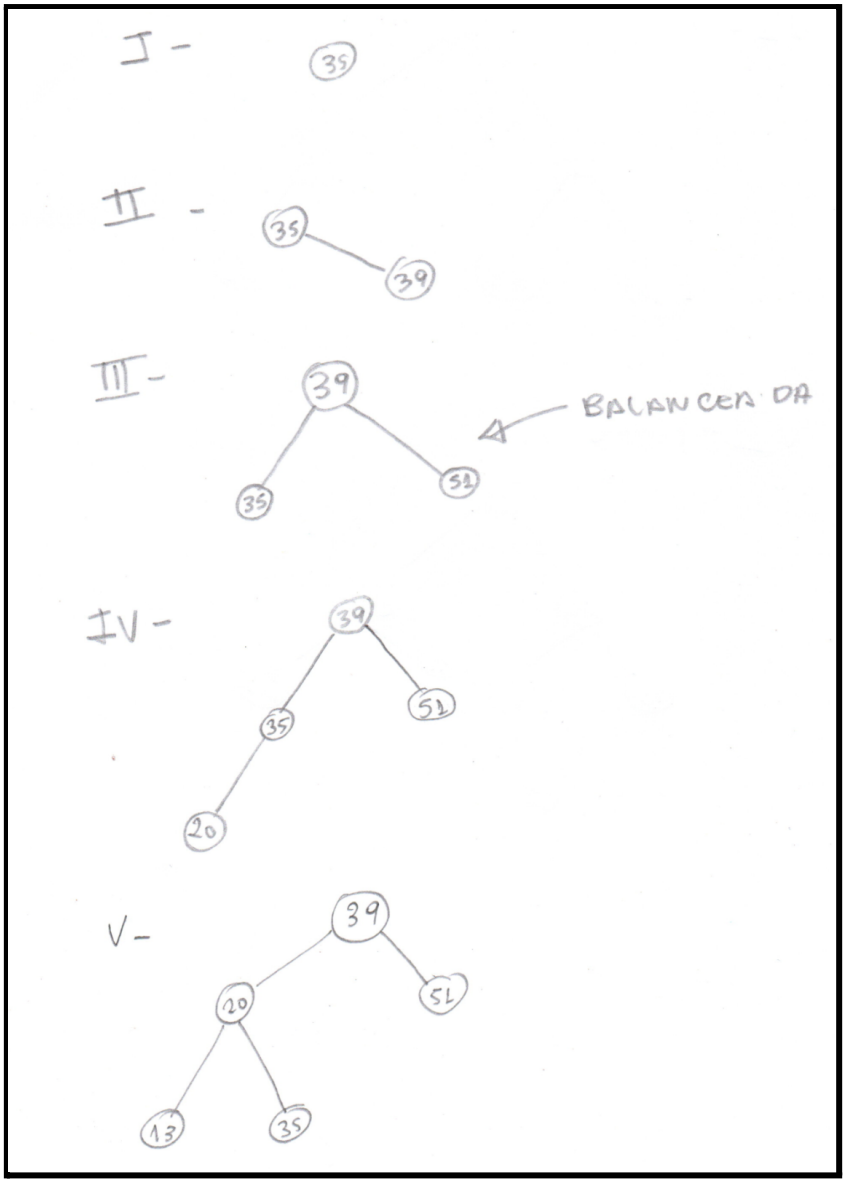

Imagem 01 - Simulação da construção de uma árvore AVL

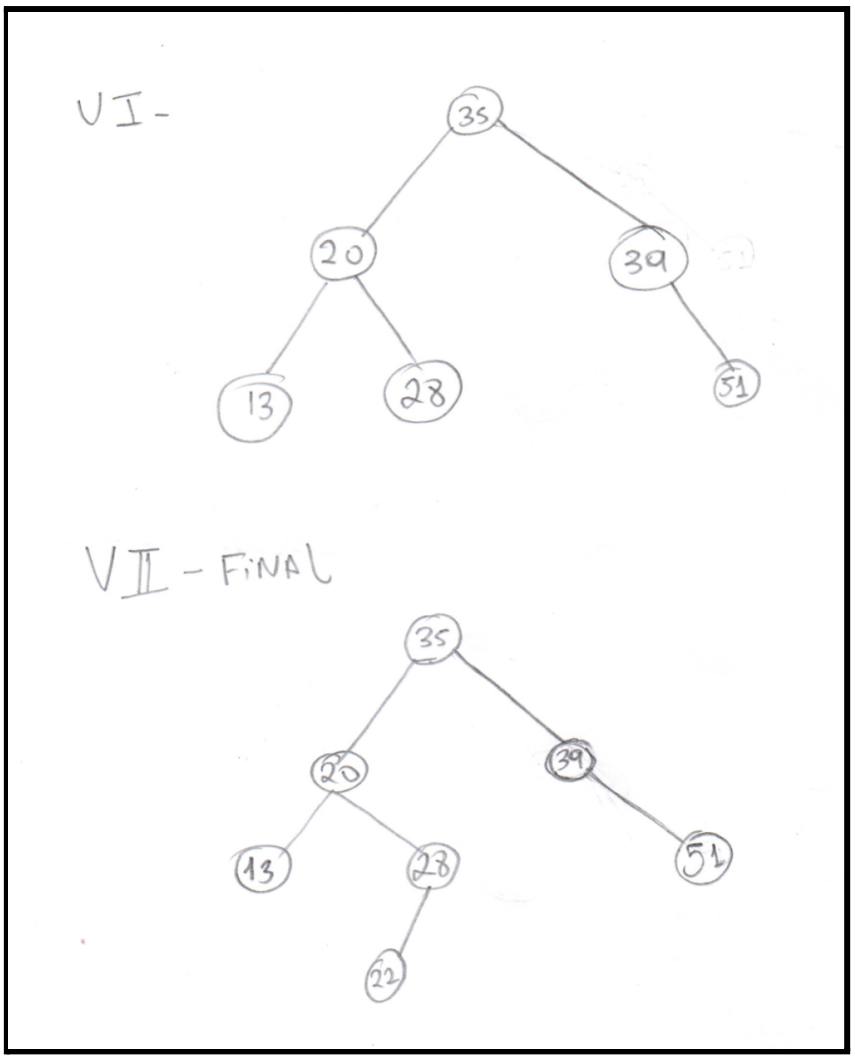

Imagem 02 - Simulação da construção de uma árvore AVL (final).

Após inserir é preciso ir fazendo as operações para que a árvore continue balanceada, no desenho registro as evoluções durante cada inserção.

Questão 3 (2,0 pts)

Sobre árvores B, e utilizando suas próprias palavras:

(a) Conceitue o que são árvores B.

Usadas para a manipulação de um conjunto muito grande de dados, como em banco de dados, as árvores B são estruturas que minimizam o tempo de acesso para buscas, inserções e remoções dentro desses conjuntos.

(b) Explique o procedimento de cisão de uma página de uma árvore B.

Cada nó de uma árvore B corresponde a uma página, se ocorrer de uma folha possuir 2d chaves e for inserido uma nova chave, ela terá 2d+1 de chaves. Para resolver isso é preciso reorganizar os nós, isso é o processo de cisão.

(c) Explique o procedimento de concatenação de duas páginas numa árvore B.

Após uma remoção, o número de chaves da página pode ficar menor que d e para resolver isso aplica-se o procedimento de concatenação que ocorre quando as duas páginas forem adjacentes e possuírem juntas menos de 2d chaves. Esse processo transforma as as duas entradas dessas páginas em uma. E todo esse processo pode se repetir no momento que houver as condições.

(d) Explique o procedimento de redistribuição de duas páginas em uma árvore B.

Após uma remoção o número de chaves da página pode ficar menor que d e para resolver isso aplica-se o procedimento de redistribuição que ocorre quando a página E e seu adjacente J contém juntos 2d ou mais chaves. Mas este processo resulta em uma página muito grande, por isso acontece uma cisão. Diferente de concatenação, esse processo não se repete.

Questão 4 (2,0 pt)

Sobre árvores Vermelho-Preto, e utilizando suas próprias palavras:

(a) Conceitue o que são árvores Vermelho-Preto.

São um tipo de árvores de busca, que preservam a propriedade de balanceamento, nessa estrutura os nós são todos equilibrados. Possuindo uma cor por nó, e com a restrição dessas cores ocorre o balanceamento.

(b) Seu uso melhora o desempenho da busca? Explique.

Sim, ela herda características semelhantes a uma AVL por também ser balanceada, em seu uso na operação de busca no pior caso reflete em um custo igual a O(log n).

(c) Descreva os ajustes que podem ocorrer na estrutura uma árvore Vermelho-Preto com a adição de um novo nó.

A possibilidade de ocorrer os seguintes casos:

1. Se o nó é preto, não há necessidade de operações.

2. Se o nó é vermelho é preciso das seguintes operações: pode ter de alterar a cor para chegar no equilíbrio e pode haver operações de rotação.

(d) Dê um exemplo de remoção de um nó em uma árvore Vermelho-Preto.

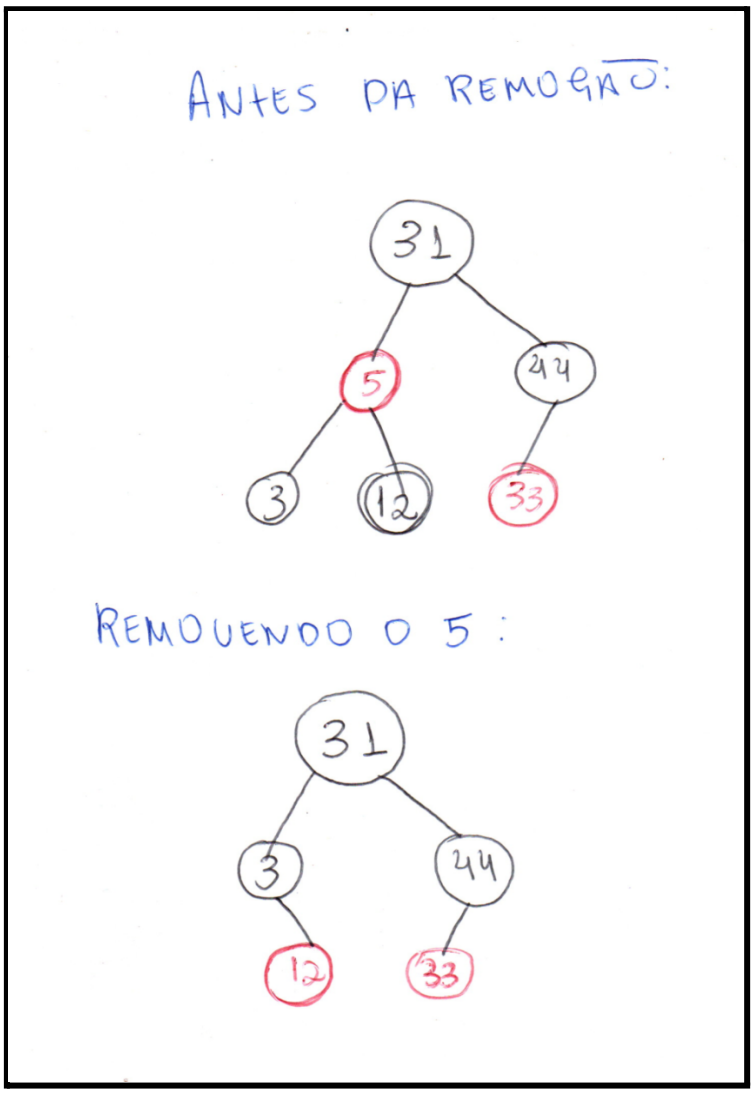

Imagem 03 - exemplo de remoção em árvore Vermelho-Preto

Questão 5 (3,0pts)

Implemente os seguintes TADs (inclua comentários explicando as implementações propostas e construa um programa teste para os TADs criados):

(a) Um TAD para manipulação de árvores binárias.

Nome do arquivo: ArvoreBinaria.c
Obs.: Programa de teste incluso no main.

```c
/* Implementação de árvore binária */

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

//armazenando as informações
typedef struct arv
{
  struct arv* sad;
  struct arv* sae;
  int num;
} ARV;/* A estrutura é representada por um ponteiro
        para o nó raiz. Assim vamos ter acesso aos outros nós.*/


ARV* c_arv(){ //criando arvore.
   return NULL;
}

int arv_v(ARV* t){ //verificando se a arvore está vazia
   return t == NULL;
}

void inf_arv(ARV* t){ //puxando a informação da arvore

    printf("<"); //para organizar, menor significa esquerda

    if(!arv_v(t)){ //se a arvore não estiver vazia

    // Mostra os elementos em pré-ordem
    printf("%d ", t->num); //mostra a raiz
    inf_arv(t->sae); // mostra a subárvore à esquerda
    inf_arv(t->sad); //mostra a subárvore à direita
   }
   printf(">"); // para organizar, maior significa direita - dica sugerida para organizar.
}

void inser_avr(ARV** t, int num) // função que insere dados na árvore.
{
  if(*t == NULL)
  {
    *t = (ARV*)malloc(sizeof(ARV)); //Aloca memória
```

```
   (*t)->sae = NULL;
   (*t)->sad = NULL;
   (*t)->num = num; //Armazena a informação
  } else {
   if(num < (*t)->num) //Se o número for menor então vai para a sub-esquerda
   {
     inser_avr(&(*t)->sae, num);
   }
   if(num > (*t)->num) //Se o número for maior então vai para a sub_direita
   {
     inser_avr(&(*t)->sad, num);
   }
  }
}

int verf_arv(ARV* t, int num) { //verifica se determinado elemento pertence a árvore.

  if(arv_v(t)) {
   return 0;
  }
  return t->num==num ||verf_arv(t->sae, num) || verf_arv(t->sad, num); //interrompe quando o elemento
é encontrado.
}

int main()
{
 ARV* t = c_arv(); //criando a árvore

  //inserindo elementos
  inser_avr(&t, 20);
  inser_avr(&t, 15);
  inser_avr(&t, 12);
  inser_avr(&t, 103);

  inf_arv(t); // mostrando os elementos em pré-ordem.

  printf("\nInformações:" );
 if(arv_v(t)){//verificação, se a árvore está vazia ou não.
   printf("\nArvore esta vazia\n");

  } else {
   printf("\n\nArvore nao esta vazia\n");
  }
```

```
 //testes para verificar se os elementos pertence a arvore
 if(verf_arv(t, 10)) {
   printf("\nEsse 10 pertence a arvore\n");
 } else {
    printf("\nEsse 10 nao pertence a arvore\n");
 }

 if(verf_arv(t, 103)){
    printf("\nEsse 103 pertence a arvore\n");
 } else {
    printf("\nEsse 103 nao pertence a arvore\n");
 }

 //Faltou o desenvolvimento da função remover.

 return 0;
}
```

(b) Um TAD para árvores binárias de busca.

Nome do arquivo: a_busca.c
Obs.: Código pode está em inglês, mas os comentários estão em português, programa teste incluso no main.

```
#include <stdio.h>
#include <stdlib.h>
#include <locale.h> //serve para exibir acento.


struct node { //criando a estrutura do nó
   struct node *left, *right;
   int key;
};


struct node *newNode(int item) { //criando o nó
 struct node *temp = (struct node *)malloc(sizeof(struct node));
 temp->key = item;
 temp->left = temp->right = NULL;
 return temp;
}


void inorder(struct node *root) { //configurando a ordem e como vai ser exibida
```

```
  if (root != NULL) {

    inorder(root->left);
    printf("%d ", root->key);
    inorder(root->right);
  }
}


struct node *insert(struct node *node, int key) { //estrutura com as condições para inserir o nó

  if (node == NULL) return newNode(key); //retorna um novo nó se a árvore estiver vazia.

  //condições que checam o lugar certo para inserir o nó
  if (key < node->key)
    node->left = insert(node->left, key);
  else
    node->right = insert(node->right, key);

  return node;
}

struct node *minValueNode(struct node *node) { //essa estrutura server para achar o nó sucessor.
  struct node *current = node;

   while (current && current->left != NULL) //checa a folha a esquerda.
    current = current->left;

  return current;
}


struct node *deleteNode(struct node *root, int key) { // estrutura com condições para deletar um elemento.

  if (root == NULL) return root; // teste para árvore vazia.

 //testes para achar o elemento a ser deletado.
  if (key < root->key)
    root->left = deleteNode(root->left, key);
  else if (key > root->key)
    root->right = deleteNode(root->right, key);

  else {
```

```
    // Se o nó estiver com apenas um filho ou nenhum filho
    if (root->left == NULL) {
      struct node *temp = root->right;
      free(root);
      return temp;
    } else if (root->right == NULL) {
      struct node *temp = root->left;
      free(root);
      return temp;
    }

    // Se o nó tiver dois filhos
    struct node *temp = minValueNode(root->right);

    // Coloca o sucessor de ordem na posição do nó a ser excluído
    root->key = temp->key;

    // deletando o sucessor pedido
    root->right = deleteNode(root->right, temp->key);
  }
  return root;
}


int main() {
    //Aqui fica o teste do que foi implementado acima.

  setlocale(LC_ALL, "Portuguese"); //Consegue exibir os acentos nos printf.

  struct node *root = NULL;
  root = insert(root, 88);
  root = insert(root, 34);
  root = insert(root, 17);
  root = insert(root, 63);
  root = insert(root, 7);
  root = insert(root, 102);
  root = insert(root, 141);
  root = insert(root, 47);

  printf("\n\nImprimindo a árvore em ordem: \n ");
  inorder(root);

  root = deleteNode(root, 7);
  printf("\n\nImprimindo a árvore depois de deletar o 7: \n");
```

```
  inorder(root);
}
```

(c) Um TAD para árvores AVL.

Nome do arquivo: avl.c
Obs.: Tive bastante dificuldade em implementar esse algoritmo, mesmo com pesquisas em materiais, livros e etc. O teste está incluso no main.

```
#include <stdio.h>
#include <stdlib.h>
#include <locale.h> //serve para ler acento.

/*Obs.: Código em inglês por conta das fontes de pesquisa,
 mas os comentários e explicações estão em português*/

struct Node { //criando o nó
  struct Node *left;
  struct Node *right;
  int height;
  int key;
};

int max(int a, int b);

int height(struct Node *N) { //calculando a altura
  if (N == NULL)
    return 0;
  return N->height;
}

int max(int a, int b) { //ta recebendo o começo e o fim e comparando.
  return (a > b) ? a : b;
}


struct Node *newNode(int key) { //alocando os nós
  struct Node *node = (struct Node *)
  malloc(sizeof(struct Node));
  node->key = key;
  node->left = NULL;
  node->right = NULL;
  node->height = 1;
  return (node);
}
```

```
struct Node *rightRotate(struct Node *y) { //rotação a direita
  struct Node *x = y->left;
  struct Node *T2 = x->right;

  x->right = y;
  y->left = T2;

  y->height = max(height(y->left), height(y->right)) + 1;
  x->height = max(height(x->left), height(x->right)) + 1;

  return x;
}


struct Node *leftRotate(struct Node *x) { //rotação a esquerda
  struct Node *y = x->right;
  struct Node *T2 = y->left;

  y->left = x;
  x->right = T2;

  x->height = max(height(x->left), height(x->right)) + 1;
  y->height = max(height(y->left), height(y->right)) + 1;

  return y;
}


int getBalance(struct Node *N) { //obtendo o fator de equilíbrio
  if (N == NULL)
    return 0;
  return height(N->left) - height(N->right);
}


struct Node *insertNode(struct Node *node, int key) { //inserindo os nós e fazendo as comparações para
saber onde cada um ficará.

  if (node == NULL)
    return (newNode(key));

  if (key < node->key)
```

```
  node->left = insertNode(node->left, key);
 else if (key > node->key)
  node->right = insertNode(node->right, key);
 else
  return node;

 //Atualizando o fator de balanceamento e balanceando a árvore.
 node->height = 1 + max(height(node->left),
            height(node->right));

 int balance = getBalance(node);
 if (balance > 1 && key < node->left->key)
  return rightRotate(node);

 if (balance < -1 && key > node->right->key)
  return leftRotate(node);

 if (balance > 1 && key > node->left->key) {
  node->left = leftRotate(node->left);
  return rightRotate(node);
 }

 if (balance < -1 && key < node->right->key) {
  node->right = rightRotate(node->right);
  return leftRotate(node);
 }

 return node;
}

struct Node *minValueNode(struct Node *node) { //aqui pega o nó com um valor minimo de chave e faz
comparação.
 struct Node *current = node;

 while (current->left != NULL)
  current = current->left;

 return current;
}


struct Node *deleteNode(struct Node *root, int key) { //função que acha e remove os nós
  if (root == NULL)
   return root;
```

```
if (key < root->key)
  root->left = deleteNode(root->left, key);

else if (key > root->key)
  root->right = deleteNode(root->right, key);

else {
  if ((root->left == NULL) || (root->right == NULL)) {
    struct Node *temp = root->left ? root->left : root->right;

    if (temp == NULL) {
      temp = root;
      root = NULL;
    } else
      *root = *temp;
    free(temp);
  } else {
    struct Node *temp = minValueNode(root->right);

    root->key = temp->key;

    root->right = deleteNode(root->right, temp->key);
  }
}

if (root == NULL)
  return root;

//Atualizando o fator de balanceamento e balanceando.
root->height = 1 + max(height(root->left),
            height(root->right));

int balance = getBalance(root);
if (balance > 1 && getBalance(root->left) >= 0)
  return rightRotate(root);

if (balance > 1 && getBalance(root->left) < 0) {
  root->left = leftRotate(root->left);
  return rightRotate(root);
}

if (balance < -1 && getBalance(root->right) <= 0)
  return leftRotate(root);
```

```
  if (balance < -1 && getBalance(root->right) > 0) {
    root->right = rightRotate(root->right);
    return leftRotate(root);
  }

  return root;
}


void printPreOrder(struct Node *root) { /*Essa função ta printando a pré-ordem: trata raiz, percorre esquerda, percorre direita*/
  if (root != NULL) {
    printf("\n%d ", root->key);
    printPreOrder(root->left);
    printPreOrder(root->right);
  }
}

int main() {
  /*Aqui fica o teste do que foi implementado acima,
    irá chamar, inserir, mostrar e deletar um nó da arvore,
    e ao fim imprime o resultado final na tela.
  */
  setlocale(LC_ALL, "Portuguese"); //Consegue exibir os acentos nos printf.

  struct Node *root = NULL;

  root = insertNode(root, 55);
  root = insertNode(root, 13);
  root = insertNode(root, 14);
  root = insertNode(root, 20);
  root = insertNode(root, 78);
  root = insertNode(root, 36);
  root = insertNode(root, 2);

  printf("Árvore AVL:\n");
  printPreOrder(root);

  root = deleteNode(root, 20);

  printf("\n\nÁrvore AVL Final(Já com o delete e balanceada):\n ");
  printPreOrder(root);
```

```
  return 0;
}
```