Disciplina *Algoritmos e Estrutura de Dados – 2 [Semestre Letivo: 2022/2]*	
Nomes dos(as) acadêmico(as) 1 – Laura Martins Vieira Gonçalves 2 – Luís Miguel Gouveia Machado 3 – Wevton Junior Fagundes Santana	**Números de Matrícula** 1 – 202103748 2 – 202103755 3 – 202103785
Turma: INF0287A	**Professor(a):** *Wanderley de Souza Alencar*

TEMA: ÁRVORES AVL

I – INTRODUÇÃO

A árvore AVL é uma estrutura de dados criada em 1962 que segue o padrão de uma árvore binária de busca com algumas características adicionais. O nome AVL tem origem nas iniciais de seus criadores, cujos nomes eram: Georgy Adelson-Velsky e Evgenii Landis.

A principal ideia dessa estrutura de dados é otimizar o tempo das operações que serão realizadas e resolver certas questões que ocorrem em uma árvore binária de busca normal. O custo operacional de pesquisas em árvores está normalmente relacionado à altura da árvore, que corresponde à distância do nó mais distante até a raiz. Quanto maior é essa distância, maior é o tempo gasto para percorrer a árvore no pior cenário. A AVL busca organizar a árvore de tal modo que a altura seja a menor possível, permitindo a diferença de no máximo um nível entre a subárvore da direita e a subárvore da esquerda; processo conhecido como balanceamento. Tal tarefa é realizada mediante o uso de rotações, as quais serão explicadas na próxima seção.

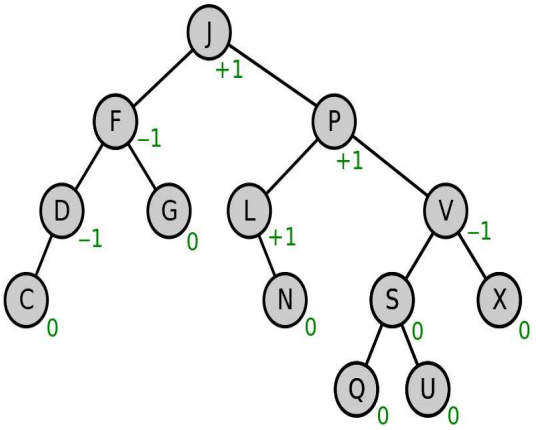
II – AS ÁRVORES AVL: DESCRIÇÃO

Entre as operações fundamentais a serem implementadas estão:

- Inserção
- Remoção
- Checagem
- Rotações (balanceamento)
    - Rotação para a esquerda
    - Rotação para a direita
    - Rotação dupla para a esquerda
    - Rotação dupla para a direita

Tanto a operação de inserir quanto a de remover seguem um modelo semelhante ao que ocorre em uma árvore binária de busca normal, o diferencial está no fato de que na AVL, após a inserção ou remoção, ocorre uma checagem com o intuito de averiguar se a árvore continua balanceada ou se ela ficou desbalanceada. Em caso de desbalanceamento, devem ser realizadas rotações, medidas que alteram a disposição dos nós mas que mantêm a característica central de uma árvore binária de busca: nós à esquerda menores do que o nó pai e nós à direita maiores do que o nó pai.

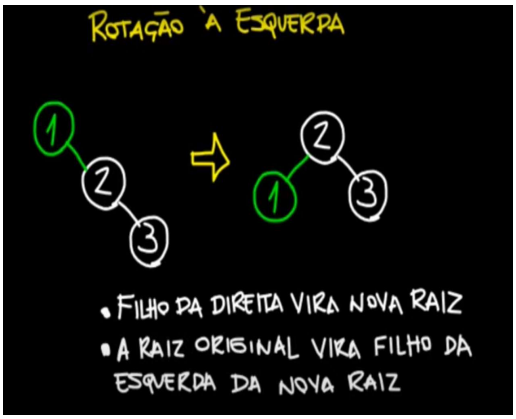
Será descrito inicialmente como ocorre a checagem. A cada nó é possível associar um índice, chamado de fator de balanceamento, que consiste em um número inteiro que representa a diferença da altura da subárvore à direita da altura da subárvore à esquerda. Se este número for maior do que 1 ou menor do que -1, a árvore está desbalanceada. A seguir, é mostrado um diagrama que retrata o fator de balanceamento em cada nó.



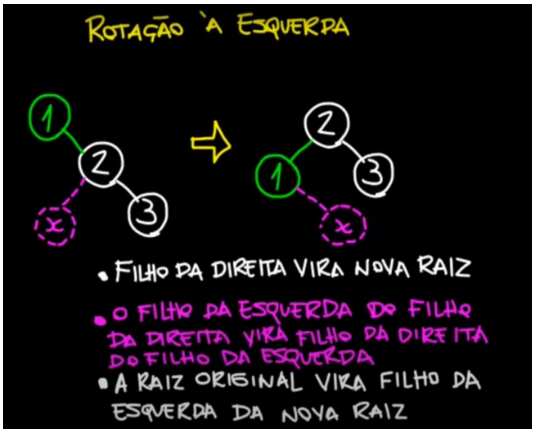
O fator de balanceamento também determina qual tipo de rotação deverá ser utilizado para restaurar o equilíbrio. Existem quatro tipos de rotação, os quais serão explicados a seguir.

Caso 1: rotação à esquerda (Right-Heavy Situation)

Se o índice for maior do que 1 (sinal positivo), então a árvore encontra-se desbalanceada para a direita (em Inglês, Right-Heavy Situation), sendo necessária uma rotação para a esquerda (sentido contrário ao do desbalanceamento) para corrigir o cenário. A situação mais simples é retratada na imagem abaixo, repare que o fator de balanceamento da raiz é 2 – 0 = 2 (maior do que 1). Note também que o desbalanceamento é uma propriedade do nó, pode-se dizer que o nó raiz encontra-se desbalanceado.



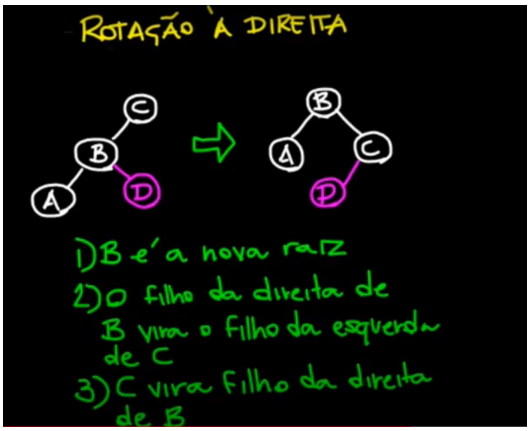
Como mostrado na figura, o filho da direita do nó desbalanceado (chave 2) vira a nova raiz, a raiz original (chave 1) vira o filho da esquerda da nova raiz (chave 2) e o equilíbrio é restaurado. Há ainda mais um possível cenário, que ocorre quando o filho da direita tem um filho à esquerda. A solução é mostrada a seguir.



Perceba que é necessário um passo adicional escrito em roxo na figura. Note que na situação original, x é maior do que 1 e menor do que 2, propriedade mantida após a rotação.

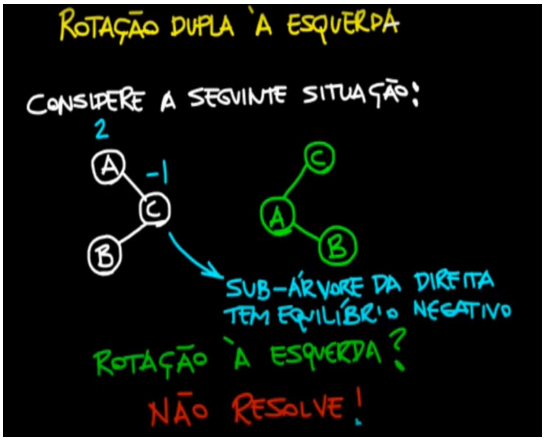
Caso 2: rotação à direita (Left-Heavy Situation)

O caso 2 é simétrico ao caso 1, como pode ser visto na imagem abaixo.

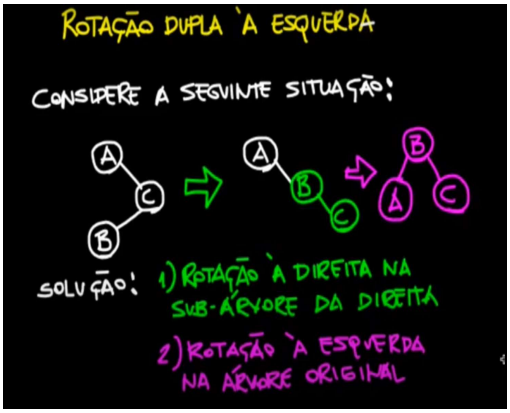


Caso 3: rotação dupla para a esquerda

O caso 3 ocorre quando a árvore está desbalanceada para a direita (Right-Heavy Situation) e uma simples rotação para a esquerda não é suficiente para restaurar o equilíbrio. Esse cenário ocorre quando o índice do nó desbalanceado tem sinal positivo e o índice do filho à direita tem sinal negativo. Veja a ilustração abaixo.



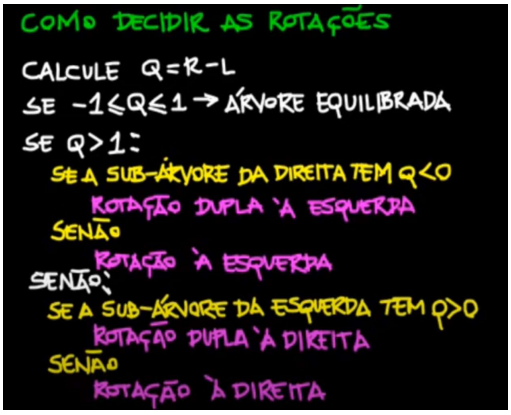
Neste caso, é necessário aplicar uma rotação para a direita na subárvore da direita e depois uma rotação para a esquerda na árvore original.



Caso 4: rotação dupla à direita

Caso simétrico ao caso 3.

Resumo dos casos:



* Q (fator de balanceamento) = R (altura da subárvore da direita) – L (altura da subárvore da esquerda).

III – A REPRESENTAÇÃO COMPUTACIONAL

Representação computacional escolhida:

```
// Estrutura do no:
typedef struct Node
{
    int key;
    int data;
    struct Node * left;
    struct Node * right;
    int height;
} Node;
```

Descrição: o nó é representado por uma struct (estrutura) com os seguintes campos:

- Key (chave do nó) – representada por um valor inteiro
- Data (dado do nó) – representado por um valor inteiro
- struct Node *left (ponteiro para o nó à esquerda)
- struct Node *right (ponteiro para o nó à direita)
- height – altura em que o nó se encontra na árvore

Funções implementadas:

```
// Funcoes essenciais:
int max(int a, int b);
int Create_Tree_Root(Node ** root);
Node * createNode(int key, int data);
int height(Node * node);
int getBalance(Node * node);
Node * rightRotation(Node * y);
Node * leftRotation(Node * x);
Node * insertNode(Node * node, int key, int data);
Node *deleteNode(Node *root, int key);
Node * Search_Key(Node * node, int key);
Node *minValueNode(Node *node);
int Delete_Tree(Node ** root);
```

- int max (int a, int b): função que retorna o maior entre dois valores
- int Create_Tree_Root (Node ** root): função que declara o ponteiro que representa a raiz da árvore como NULL
- Node * createNode (int key, int data): função que cria um nó e retorna o ponteiro para o nó criado
- int height (Node * node): retorna um inteiro que é a altura do nó passado como parâmetro

- int getBalance (Node * node): retorna um inteiro que representa o fator de balanceamento do nó passado como parâmetro
- Node * rightRotation (Node * y): função que aplica a rotação à direita na árvore (ou subárvore) e retorna o ponteiro que representa a raiz da árvore (ou subárvore) após o processo de rotação
- Node *leftRotation (Node * x): função que aplica a rotação à esquerda na árvore (ou subárvore) e retorna o ponteiro que representa a raiz da árvore (ou subárvore) após o processo de rotação
- Node * insertNode (Node * node, int key, int data): função que insere o nó na árvore respeitando as propriedades básicas de uma árvore binária de busca. Retorna um ponteiro para a raiz da árvore após a inserção e após a realização das rotações necessárias para manter a árvore balanceada
- Node * deleteNode (Node * root, int key): função que deleta o nó (libera o espaço na memória) cuja chave é passada como parâmetro para a função. Retorna um ponteiro para raiz da árvore após a deleção e após a realização das rotações necessárias para manter a árvore balanceada
- Node * Search_Key (Node * node, int key): função que encontra um nó na árvore a partir da chave fornecida como parâmetro na função. Retorna um ponteiro para o nó encontrado
- Node * minValueNode (Node *node): retorna o ponteiro para o nó de menor chave na árvore (ou subárvore) que tem como raiz o nó passado como parâmetro
- int Delete_Tree (Node ** root): deleta a árvore (libera espaço na memória) que tem como raiz o nó passado como parâmetro para a função

IV – A IMPLEMENTAÇÃO

Operações:

1 – Criação da árvore:

```c
int Create_Tree_Root(Node ** root)
{
    if((*root) != NULL)
    {
        (*root) = NULL;
        printf("Raiz da arvore criada com sucesso!");
    }
    else
    {
        printf("Raiz da arvore ja criada!");
    }
}
```

2 – Criação do nó:

```c
Node * createNode(int key, int data)
{
    Node * new_node = NULL;
    new_node = (Node *) malloc(sizeof(struct Node));

    if(new_node == NULL){
        return NULL;
    }
    new_node->key = key;
    new_node->data = data;
    new_node->left = NULL;
    new_node->right = NULL;
    new_node->height = 1;

    return new_node;
}
```

3 – Rotação à direita:

```
Node * rightRotation(Node * y)
{
    Node * x = y->left;
    Node * T2 = x->right;

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

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

    return x;
}
```

4 – Rotação à esquerda:

```
Node * leftRotation(Node * x)
{
    Node * y = x->right;
    Node * T2 = y->left;

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

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

    return y;
}
```

5 – Inserção de um nó:

```
Node * insertNode(Node * node, int key, int data)
{
    if(node != NULL)
    {
        if(key<node->key)
        {
            node->left = insertNode(node->left, key, data);
        }
        else if(key>node->key)
        {
            node->right = insertNode(node->right, key, data);
        }
        else
        {
            printf("Chave invalida de insercao.\n");
            return node;
        }
    }
    else
    {
        return (createNode(key, data));
    }
```

```
    node->height = 1 + max(height(node->left), height(node->right));
    int bf = getBalance(node);

    // Rotacao dupla a esquerda
        if(bf>1 && key <  node->right->key)
        {
            node->right = rightRotation(node->right);
            return leftRotation(node);
        }

    // Rotacao a esquerda
        if(bf>1 && key > node->right->key)
        {
            return leftRotation(node);
        }

    // Rotacao dupla a direita
        if(bf<-1 && key > node->left->key)
        {
            node->left = leftRotation(node->left);
            return rightRotation(node);
        }

    //  Rotacao a direita
        if(bf<-1 && key < node->left->key)
        {
            return rightRotation(node);
        }

    return node;
}
```

6 – Remoção de um nó:

```
Node *deleteNode(Node *root, int key) {
  // Encontrar o no e fazer a delecao
  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)) {
      Node *temp = root->left ? root->left : root->right;

      if (temp == NULL) {
        temp = root;
        root = NULL;
      } else *root = *temp;

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

      root->key = temp->key;

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

  if (root == NULL)
    return root;
```

```
  // Atualizar o fator de balanceamento de cada no e
  // balancear a arvore
  root->height = 1 + max(height(root->left),
                         height(root->right));

  int balance = getBalance(root);

  if (balance > 1 && getBalance(root->right) < 0) {
    root->right = rightRotation(root->right);
    return leftRotation(root);
  }

  if (balance > 1 && getBalance(root->right) >= 0)
    return leftRotation(root);

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

  if (balance <-1 && getBalance(root->left) <= 0)
    return rightRotation(root);

  return root;
}
```

7 – Consulta de um nó:

```
Node * Search_Key(Node * node, int key)
{
    if(key < node->key && node->left!=NULL)
    {
        return Search_Key(node->left, key);
    }
    else if(key > node->key && node->right!=NULL)
    {
        return Search_Key(node->right, key);
    }
    else if(key == node->key)
    {
        printf("No encontrado!\n");
        return node;
    }
    else
    {
        printf("No nao encontrado.\n");
        return NULL;
    }
}
```

8 – Destruição da árvore:

```
int Delete_Tree(Node ** root)
{
    if((*root) == NULL)
    {
        return false;
    }
    else
    {
        while((*root) != NULL)
        {
            *root = deleteNode(*root, (*root)->key);
        }
        return true;
    }
}
```

V – PROCESSO DE INSTALAÇÃO DO PROGRAMA DE COMPUTADOR

É recomendado o uso de um computador com o sistema operacional Linux, uma vez que isso facilitará a instalação do programa pelo fato de o terminal Linux contar com recursos que permitem a compilação e a execução de programas em C. Caso o sistema operacional não tenha tais funcionalidades nativas (Windows é um exemplo), será necessário instalar de maneira prévia um ambiente que permita a compilação e a execução de programas em C, como o Code::Blocks, por exemplo. O processo de instalação do Code::Blocks pode ser encontrado no site: https://www.codeblocks.org/downloads/.

O programa-fonte em C deve ser salvo em um diretório ou pasta e depois compilado. No terminal Linux, pode ser usado o comando gcc da seguinte forma: gcc programa.c -o programa, seguido da expressão ./programa, a qual executará o código em questão. Caso seja feita a opção pelo Code::Blocks, as instruções podem ser encontradas no manual de usuário, disponibilizado no seguinte site: https://www.codeblocks.org/user-manual/.

VII – BIBLIOGRAFIA

Vídeo sobre a árvore AVL (Português): https://www.youtube.com/watch?v=3zmjQlJhBLM
Tutorial escrito em Inglês no site Programiz: https://www.programiz.com/dsa/avl-tree
Vídeotutorial em Hindi: https://www.youtube.com/watch?v=Wdy36bumttg