ESTRUTURA DE DADOS

AULA 4

Prof. Vinicius Pozzobon Borin

CONVERSA INICIAL

O objetivo desta aula é apresentar os conceitos que envolvem a estrutura de dados do tipo árvore. Ao longo deste documento será conceituada a estrutura de dados conhecida como **árvore**, e serão investigados dois modos específicos de utilização dessa estrutura de dados:

1. Árvore binária;
2. Árvore de Adelson-Velsky e Landis (árvore binária balanceada).

Para o estudo dessas estruturas serão apresentadas algumas técnicas de manipulação de dados, tais como a de inserção e a de remoção de dados.

Nos conteúdos abordados nesta aula, utilizamos esse tipo de estrutura de dados para promover o aprendizado das técnicas envolvidas na criação e na manutenção de dados em estruturas de árvores.

Todos os conceitos trabalhados anteriormente, como análise assintótica, recursividade, listas encadeadas, conceitos básicos de programação estruturada e manipulação de ponteiros, aparecerão de forma recorrente ao longo desta aula. Todos os códigos apresentados aqui estarão na forma de pseudocódigos. Isso implica a necessidade de adotar uma simbologia específica para a manipulação de ponteiros e endereços nesse pseudocódigo. Para suprir essa necessidade será adotada a seguinte nomenclatura ao longo de todo este material:

- Para indicar o endereço da variável, será adotado o símbolo **–> antes do nome da variável**. Por exemplo: $px = (->]x$. Isso significa que a variável *px* recebe o endereço da variável *x*;
- Para indicar um ponteiro, será adotado o símbolo [–>) **após o nome da variável**. Por exemplo: x (–>]: inteiro. Isso significa que a variável *x* é uma variável do tipo ponteiro de inteiros.

TEMA 1 – ÁRVORE BINÁRIA: DEFINIÇÕES

Antes estudamos estruturas de dados do tipo lista encadeada. Uma lista encadeada apresenta uma característica importante: cada elemento da lista tem acesso somente ao elemento seguinte (lista simples) ou ao seguinte e ao anterior (lista dupla). Todos os elementos dessa lista são organizados de tal forma que é necessário percorrer, fazendo uma varredura, a estrutura de dados

para se ter acesso ao elemento desejado. Chamamos a lista encadeada de uma estrutura de dados de **organização linear**.

Nesta aula, investigaremos uma estrutura de dados que é **não linear**: a estrutura do tipo árvore. E começaremos pela estrutura de dados em árvore conhecida como **árvore binária**.

A árvore binária é uma estrutura de dados não linear organizada com elementos que não estão, necessariamente, encadeados, formando ramificações e subdivisões na organização da estrutura de dados. A árvore binária apresenta algumas características distintas, mesmo se comparada a outros tipos de árvores. Analisaremos algumas dessas características e conceituaremos alguns termos próprios referentes a árvores:

- Nó raiz (*root*):[1] nó original da árvore. Todos derivam dele;
- Nó pai: nó que dá origem (está acima) a pelo menos um outro nó;
- Nó filho: nó que deriva de um nó pai;
- Nó folha/terminal: nó que não contém filhos.

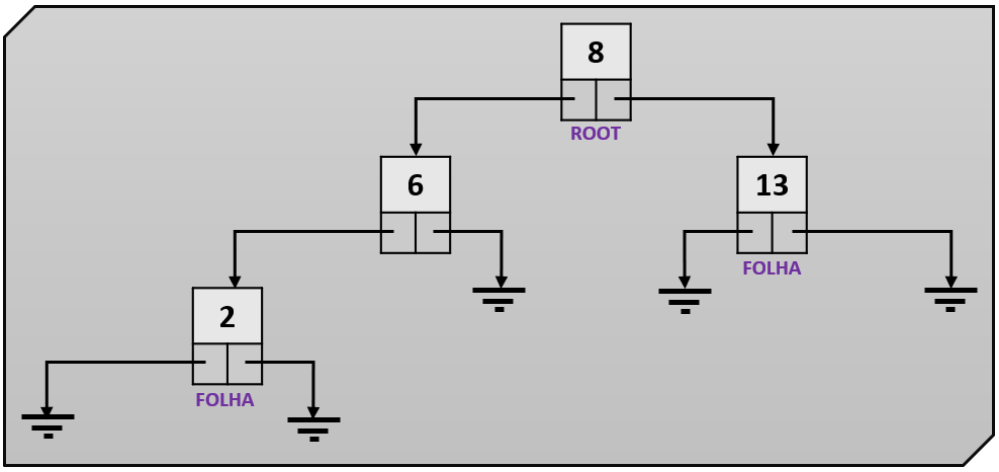
O que caracteriza uma árvore como binária é o número de ramificações de cada nó. Na árvore binária, cada nó apresenta nenhum, um ou no máximo dois nós chamados de nós filhos.

Uma árvore binária apresenta como característica a inserção de valores de forma organizada. Um modo de inserção de valores é posicionar à esquerda de um nó pai, na árvore, somente valores menores do que ele. E os valores à direita devem ser maiores que o nó pai. Desse modo, valores menores são sempre posicionados mais à esquerda da árvore e os maiores mais à direita, como vemos na Figura 1. Esse tipo de inserção é também conhecido como Binary Search Tree (BST, **árvore de busca binária**).

Na Figura 1, temos um exemplo de árvore binária. Observe que cada nó se ramifica em, no máximo, dois filhos. O que apresenta valor 8 se divide em dois (6 e 13). O nó de valor 6 se ramifica em somente um (valor 2). E os nós folha de valor 2 e valor 13 não têm nenhuma ramificação.

[1] Em português, **raiz**.

Figura 1 – Árvore binária com identificação do nó raiz (*root*) e nós folhas (*leafs*)



O objetivo de inserção na árvore é justamente facilitar a busca de dados posteriormente. Como os dados estarão colocados na árvore de uma forma organizada, saberemos que, ao percorrer as ramificações de uma árvore binária, localizaremos o valor desejado mais rapidamente pelas subdivisões.

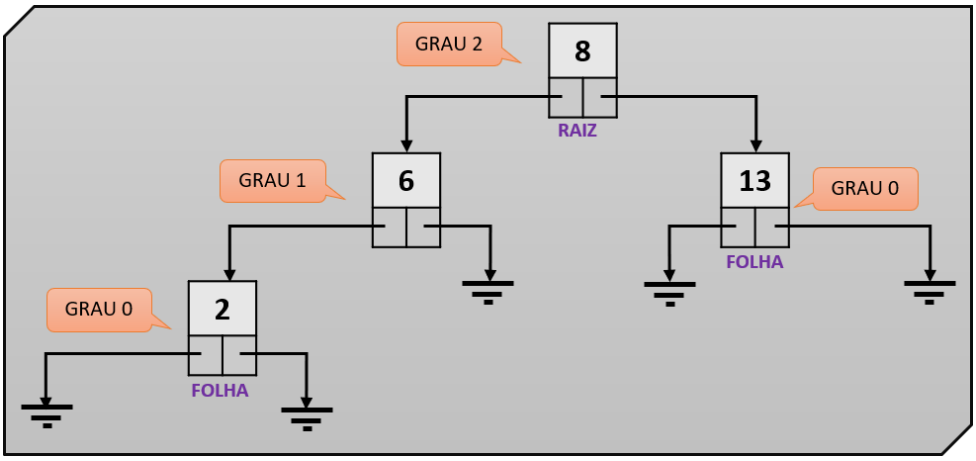
Uma árvore binária apresenta uma importante aplicação em sistemas de busca de dados. Ao longo desta aula, sempre trabalharemos com exemplos de árvores binárias utilizando uma estrutura BST.

1.1 Grau de um nó

O grau de um nó na estrutura de dados em árvore corresponde ao número de subárvores que aquele nó terá. Em uma árvore binária, o grau máximo de cada nó será dois.

Observe a Figura 2. O nó raiz com valor 8 contém duas subárvores (duas ramificações). Desse modo, seu grau será dois. O nó de valor 6 contém somente uma ramificação para o lado esquerdo e, portanto, seu grau é unitário. Os outros dois nós, por serem nós folha, estão sempre no final da árvore e não contêm filhos, tendo grau sempre zero.

Figura 2 – Árvore binária com identificação dos graus nos nós



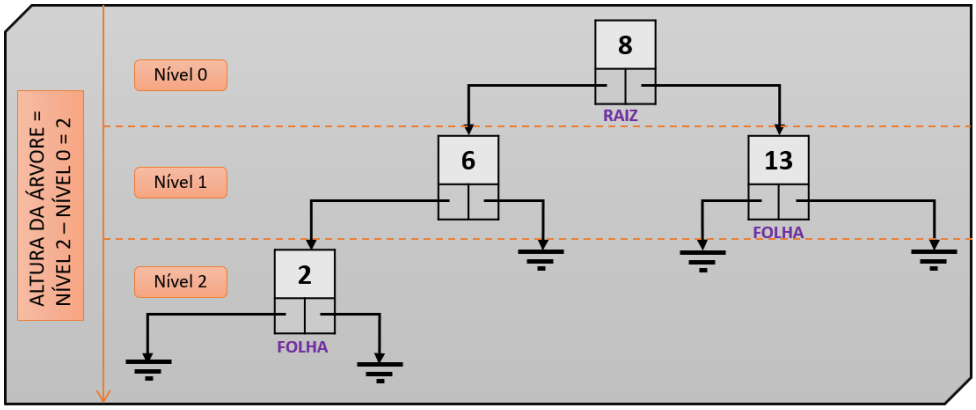
Podemos também dizer que o grau de uma árvore é sempre o maior grau dentre todos os graus de nós dessa árvore. Na árvore na Figura 2, o grau é dois.

1.2 Altura e nível da árvore

Em uma árvore, conceituamos o nó raiz como sendo o nível zero da árvore. Todo novo conjunto de ramificações da árvore caracteriza mais um nível nessa árvore. Observe a Figura 3. Os dois nós filhos do nó raiz caracterizam nível 1. E o nível 2 conterá nós filhos gerados no nível 1.

Conhecendo os níveis, a altura de uma árvore será calculada ao tomarmos o maior nível dela (nível 2, em nosso exemplo) e subtrairmos do primeiro nível (nível 0). Portanto, a árvore binária da Figura 3 tem altura 2 (nível 2 – nível 0 = 2). É também possível encontrarmos a altura relativa entre dois níveis da árvore. Para isso, basta subtrairmos os valores desses níveis.

Figura 3 – Árvore binária com identificação de alturas e níveis



1.3 Código de criação de cada nó/elemento da árvore

Em uma árvore binária é necessário que tenhamos uma estrutura contendo dois endereços de ponteiros. Um para o ramo esquerdo, outro para o ramo direito. Os ponteiros servirão para localizar na memória o próximo elemento da ramificação daquela árvore.

Em programação, representamos cada elemento da árvore como sendo um registro que contém todos os dados que desejamos armazenar, além de dois ponteiros. Esses ponteiros conterão os endereços para os próximos elementos da árvore, ou seja, para os nós filhos daquele elemento.

Na Figura 4, temos um exemplo de registro para uma árvore binária. Observe que temos um dado numérico, um ponteiro que apontará para o nó filho esquerdo e um ponteiro que apontará para o nó filho direito. Esses ponteiros são do tipo *ElementoDaArvoreBinaria*.

Figura 4 – Pseudocódigo de criação de um elemento (nó) da lista

```
registro ElementoDaArvoreBinaria
   dado: inteiro
   esquerda: ElementoDaArvoreBinaria[->)
   direita: ElementoDaArvoreBinaria[->)
fimregistro
```

Sempre que um nó for folha, significa que ele está no nível mais alto da árvore e que, portanto, não terá filhos. Sendo assim, ambos os ponteiros serão nulos. Caso o nó não seja folha, ele sempre terá ao menos um ponteiro de endereço não nulo.

Na Figura 5, temos o pseudocódigo principal. Nesse algoritmo, temos a criação do registro que define como serão os elementos de nossa árvore. Temos também a declaração de nossa **raiz** (*root*), único elemento sempre conhecido globalmente pelo programa. No código é apresentado um seletor do tipo **escolha**, que realiza a chamada de três funções distintas: uma para inserir um novo elemento na árvore; uma para buscar um elemento qualquer; e uma para visualizar toda a árvore. Essas funções serão mais bem apresentadas nos próximos temas de nossa aula.

Figura 5 – Pseudocódigo principal que declara a árvore e chama as funções de inserção, busca e visualização da árvore binária

```
1  algoritmo "ArvoreMenu"
2  var
3     //Cria uma arvore
4     registro ElementoDaArvoreBinaria
5        dado: inteiro
6        esquerda: ElementoDaArvoreBinaria[->)
7        direita: ElementoDaArvoreBinaria[->)
8     fimregistro
9     //Declara a Raiz como sendo do tipo da Árvore
10    Raiz: ElementoDaArvoreBinaria[->)
11    op, numero, posicao: inteiro
12 inicio
13       //Lê um valor para inserir e a operação desejada
14       leia(numero)
15       leia(op)
16
17       escolha (op) //Escolhe o que deseja fazer
18          caso 0:
19             InserirNaArvore( (->]Raiz, numero)
20          caso 1:
21             BuscarNaArvore( (->]Raiz, numero)
22          caso 2:
23             VisualizarArvore_Ordem( (->]Raiz)
24          caso 3:
25             VisualizarArvore_PreOrdem( (->]Raiz)
26          caso 4:
27             VisualizarArvore_PosOrdem( (->]Raiz)
28       fimescolha
29 fimalgoritmo
```

TEMA 2 – ÁRVORE BINÁRIA: INSERÇÃO DE DADOS

Em uma árvore binária montada para funcionar como uma Binary Seach Tree (BST) não existem inserções no início, no meio nem no final da árvore como ocorre nas listas encadeadas. A inserção sempre é feita após um dos nós folha da árvore, seguindo a regra anteriormente explanada:

- Dados de valores menores cujos antecessores são inseridos no ramo esquerdo da árvore;
- Dados de valores maiores cujos antecessores são inseridos no ramo direito da árvore.

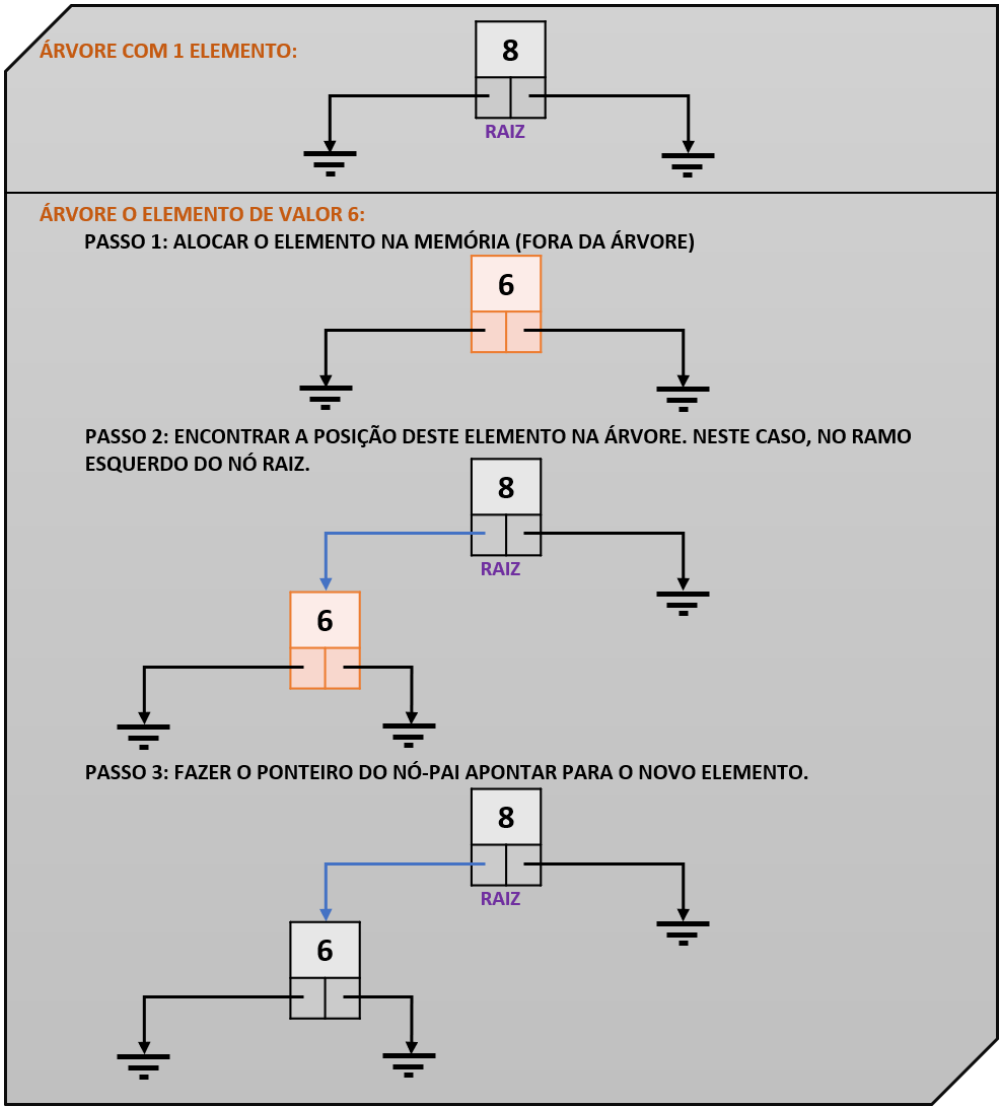
Todo novo nó inserido vira um nó folha, porque ele é sempre inserido nos níveis mais altos da árvore. Todos os nós folha podem ser facilmente

identificados em uma árvore como aqueles nós cujos ponteiros (esquerdo e direito) são nulos.

Na Figura 6, temos um exemplo de inserção no ramo esquerdo da árvore de busca binária. Analisaremos como esse procedimento acontece. Observe que temos uma árvore com um valor 8. Esse valor é a raiz da árvore e também um nó folha, pois não apresenta filhos e seus ponteiros são nulos.

Desejamos inserir o valor 6 nessa árvore. Como característica da árvore binária, todos os valores menores que o valor a ser comparado – nesse caso, o valor 8 na raiz – são colocados no ramo esquerdo da árvore. Sendo assim, o nó raiz apontará para o novo elemento com seu ponteiro esquerdo.

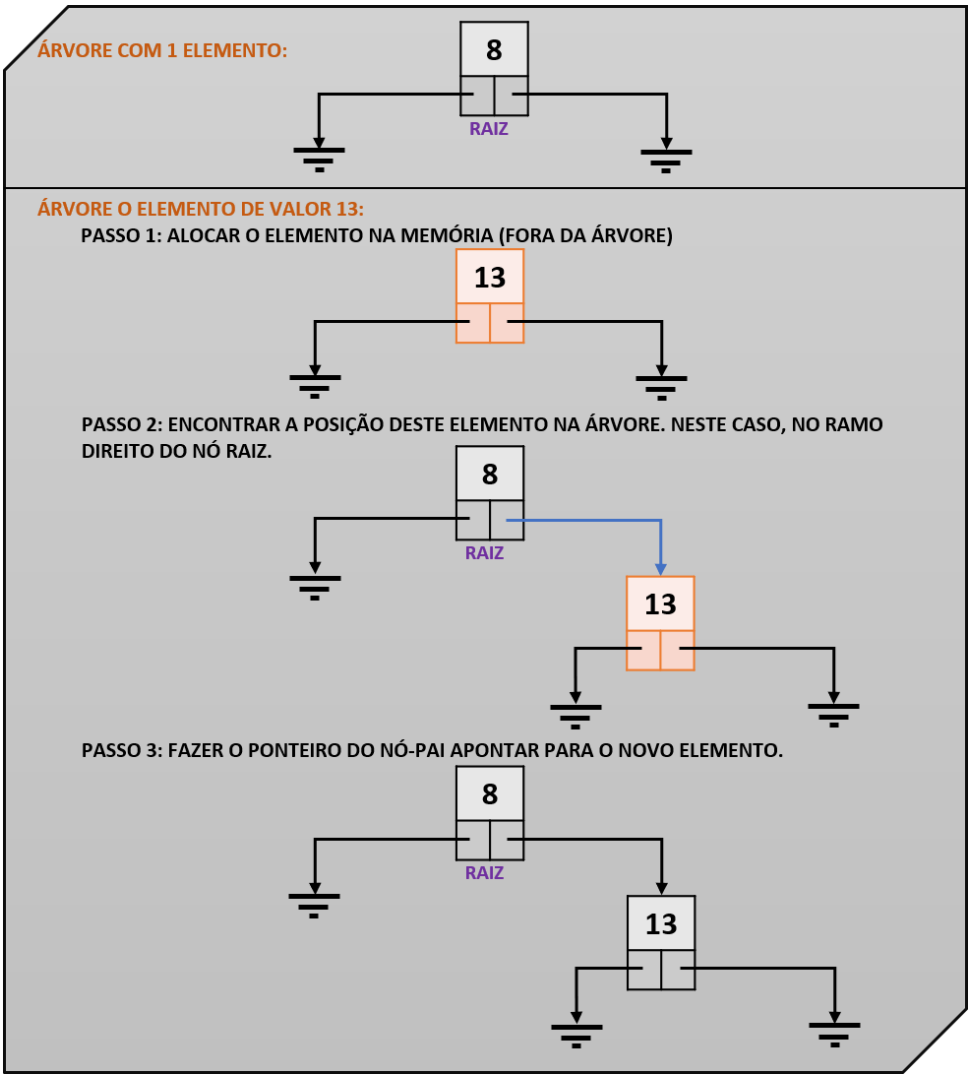
Figura 6 – Funcionamento da inserção para o lado esquerdo em uma árvore binária



Caso tivéssemos mais elementos já inseridos na árvore, o algoritmo percorreria todo o ramo da árvore, comparando o elemento a ser buscado com o valor atual até fechar no nó folha correspondente para a inserção.

Na Figura 7, temos um exemplo de inserção no ramo direito da árvore binária. Observe que a árvore apresenta um valor 8. Esse valor é a raiz da árvore e também um nó folha, pois não apresenta filhos e seus ponteiros são ambos nulos. Diferentemente da situação anterior, como queremos inserir o valor 13, que é maior que o valor 8 do nó pai, ele será posicionado no ponteiro direito do nó pai.

Figura 7 – Funcionamento da inserção para o lado direito em uma árvore binária



2.1 Pseudocódigo de inserção na árvore

Na Figura 8, temos o pseudocódigo de inserção em uma árvore binária.

Figura 8 – Pseudocódigo de inserção na árvore binária

```
1  //Cria um Procedimento que recebe como parâmetro o dado a ser inserido na árvore
2  //E o endereço do nó atualmente a ser testado, iniciando pela raiz
3  função InserirNaArvore
4  (ElementoVarredura[->)[->): registro ElementoDaArvore, numero: inteiro): sem retorno
5  var
6     //Declara um novo nó do tipo REGISTRO
7     NovoElemento[->): registro ElementoDaArvore
8  inicio
9     //Inicializa  novo nó fora da lista
10    NovoElemento[->) = NULO
11    NovoElemento->dado = numero
12
13    //Verifica se o elemento recebido está vazio
14    se (ElementoVarredura[->) == NULO)  então
15       //Se o elemento está vazio, coloca ele na árvore
16       NovoElemento->esquerda = NULO
17       NovoElemento->direita     = NULO
18       //Faz o elemento atual (vazio) da árvore receber o novo elemento
19       ElementoVarredura[->) = NovoElemento[->)
20    senão
21       //Verifica-se RECURSIVAMENTE se a árvore deve
22       //seguir para o ramo esquerdo ou direito
23       se (numero < ElementoVarredura->dado) então
24          //Segue ara a esquerda recursivamente
25          InserirNaArvore( (->]ElementoVarredura->esquerda, numero)
26       senão
27             //Segue ara a direita recursivamente
28          InserirNaArvore( (->]ElementoVarredura->direita, numero)
29       fimse
30    fimse
31 fimfunção
```

Veja o que significam algumas das linhas desse código:

- Linha 4: declaração do procedimento. Observe que temos uma variável local chamada de *ElementoVarredura*. Essa variável é um ponteiro para ponteiro (por esse motivo, utilizou-se duas vezes a nomenclatura **[–>)**). Ela tem esse nome porque recebe como parâmetro uma variável que já é ponteiro. Veja a Figura 5 novamente, que contém o menu de nosso algoritmo. Lá, declaramos uma variável para a raiz, que é um ponteiro. Essa variável é passada como parâmetro para nossa função de inserção e, portanto, temos um ponteiro para outro ponteiro;
- Linha 7: declaração da variável que inicializará o novo elemento a ser inserido na árvore;
- Linha 14: verificação de se o elemento atual sendo varrido é nulo. Esse teste acontece porque se o elemento atual estiver vazio, significa que é

possível inserir um novo dado em seu local. Caso contrário, é necessário continuar a varredura dentro das ramificações da árvore;

- Linhas 16-20: insere o novo elemento na posição desejada da árvore;
- Linha 23: verifica para qual lado da árvore devemos seguir buscando até encontrar um local vazio para inserção. Caso o valor atual testado seja maior do que o número a ser inserido, seguimos para o nó filho esquerdo da árvore (linha 25). Caso contrário, seguidos para o nó filho direito (linha 28). Perceba que ambas as chamadas das funções são feitas de forma recursiva, sempre passando como parâmetro o endereço do próximo elemento a ser testado (esquerdo ou direito). Quando a posição correta para a inserção é detectada, as funções recursivas vão sendo encerradas.

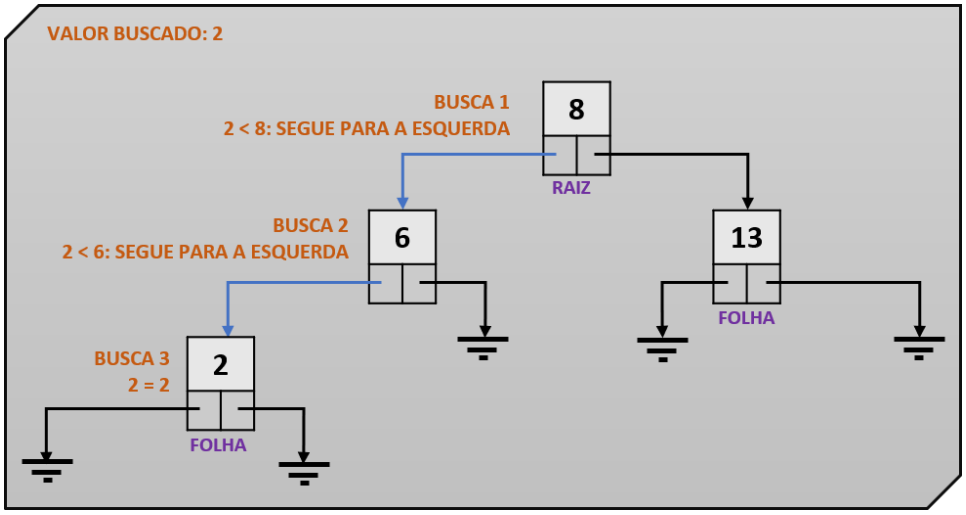
Como a inserção acontece de forma recursiva, esse algoritmo de inserção contém a complexidade assintótica $O(\log n)$.

TEMA 3 – ÁRVORE BINÁRIA: BUSCA DE DADOS

Uma das principais aplicações de uma árvore binária, conforme comentado no Tema 1, ocorre em sistemas de busca. Portanto, analisaremos como se dá o algoritmo de busca em uma árvore binária e por que ele é eficiente para essa aplicação.

Na Figura 9, temos uma busca em uma árvore com três níveis (altura 2).

Figura 9 – Funcionamento da busca em uma árvore binária. Situação em que o valor é encontrado

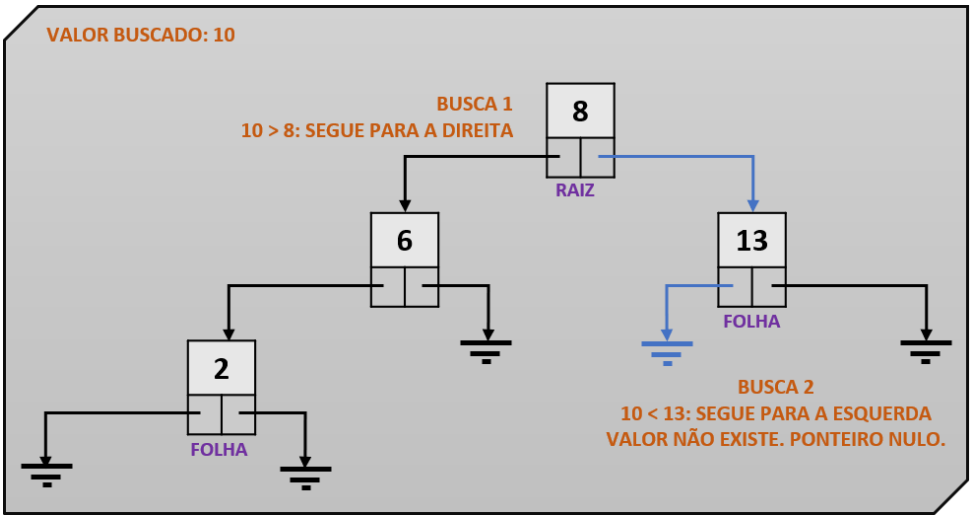


Desejamos encontrar o valor 2 nessa árvore. Vejamos as etapas:

- **Passo 1:** iniciando na raiz, testamos se ela é igual, maior ou menor que o valor 2. Caso fosse igual, nossa busca já poderia ser encerrada. Porém, o valor 8 é maior que 2. Isso significa que, caso o 2 exista nessa árvore, ele estará à esquerda do 8, e, portanto, devemos seguir pelo ramo esquerdo (ponteiro esquerdo);
- **Passo 2:** estamos no nível 1 da árvore e comparando o valor 2 com o valor 6. Novamente, verificamos se esse valor é maior, menor ou igual 2. O valor é maior do que 2. Portanto, novamente, caso o 2 exista nessa árvore, ele estará à esquerda de 6, e devemos seguir para a esquerda no próximo nível;
- **Passo 3:** estamos no nível 2 da árvore e comparamos o valor com o 2. Nesse caso, o valor é exatamente 2, e, portanto, podemos encerrar nossa busca na árvore binária.

Na Figura 10, temos uma busca em uma árvore com três níveis (altura 2).

Figura 10 – Funcionamento da busca em uma árvore binária. Situação em que o valor não é encontrado



Agora, analisaremos um caso em que o valor desejado não existe na árvore. Vejamos as etapas:

- **Passo 1:** iniciando na raiz (nível zero), testamos se ela é igual, maior ou menor que o valor desejado 10. O valor 10 é maior do que o 8, e, portanto, devemos seguir para o nível 1 no ramo direito da árvore;
- **Passo 2:** no nível 1, comparamos o valor 10 com o valor 13. O valor buscado é menor que 13, e, portanto, deve estar à esquerda do 13 no

nível seguinte. Porém, percebemos que o ponteiro esquerdo (assim como o direito) é nulo, caracterizando-se como um nó folha. Sendo assim, não existe mais nenhum nó a ser testado e nossa busca precisa ser encerrada nessa etapa, sem termos localizado o valor 10 buscado.

Por que um buscador construído a partir de uma árvore tende a ser mais eficiente se comparado a uma lista simples, por exemplo? Pense em um cadastro de dados de clientes de uma loja. A loja pode armazenar milhares de cadastros, e encontrar um cadastro específico pode não ser uma tarefa trivial.

Para realizar essas buscas, poderíamos utilizar, sem dúvida, um dos algoritmos apresentados no Tema 5 de nossa Aula 2, a busca sequencial ou a busca binária. Na busca sequencial, teríamos que passar elemento por elemento de nossa estrutura de dados. É o que ocorreria com uma lista simples, em que teríamos que varrer elemento por elemento até encontrar aquele buscado. Dependendo do tamanho da lista, levaríamos um tempo considerável realizando essa tarefa, uma vez que a complexidade para o pior caso é $O(n)$.

Agora, se você se lembrar da busca binária que vimos na Aula 2, ela tende a ser mais eficiente que uma busca sequencial. A busca binária delimita a região de busca sempre pela metade a cada nova tentativa, tornando possível localizar, para o pior caso assintótico, o dado em um tempo de execução $O(\log n)$.

Uma árvore binária também apresenta para *BigO* a mesma complexidade $O(\log n)$, também delimitando a região de busca sempre pela metade a cada novo nível da árvore. Sendo assim, a árvore binária faz para as buscas em estruturas de dados dinâmicas o equivalente ao que a busca binária faz para conjuntos de dados sequenciais.

3.1 Pseudocódigo de busca na árvore

Na Figura 11, temos o pseudocódigo de busca em uma árvore binária.

Figura 11 – Pseudocódigo de busca na árvore binária

```
1  //Cria um Procedimento que recebe como parâmetro o dado a ser buscado na árvore
2  //E o endereço do nó atualmente a ser testado, iniciando pela raiz
3  função BuscarNaArvore
4  (ElementoVarredura[->)[->): registro ElementoDaArvore, numero: inteiro)
5  : registro ElementoDaArvore
6  var
7  inicio
8     //Verifica se o elemento recebido está vazio
9     se (ElementoVarredura[->) <> NULO)  então
10       //Verifica-se RECURSIVAMENTE se a árvore deve
11       //seguir para o ramo esquerdo ou direito
12       se (numero < ElementoVarredura->dado) então
13          //Segue para a esquerda recursivamente
14          BuscarNaArvore( (->]ElementoVarredura->esquerda, numero)
15       senão
16          se (numero > ElementoVarredura->dado) então
17             //Segue para a direita recursivamente
18             BuscarNaArvore( (->]ElementoVarredura->direita, numero)
19             senão
20                se (numero == ElementoVarredura->dado) então
21                   //VALOR ENTRADO
22                   retorne ElementoVarredura[->)
23                fimse
24             fimse
25       fimse
26    senão
27       retorne NULO
28    fimse
29 fimfunção
```

Veja o que significam algumas das linhas desse código:

- Linha 4: declaração do procedimento. Observe que temos uma variável local chamada de *ElementoVarredura*. Essa variável é um ponteiro para outro ponteiro (por esse motivo, utilizamos duas vezes a nomenclatura **[->)**). Ela foi chamada assim porque recebe como parâmetro uma variável que já é ponteiro. Volte à Figura 5, que contém o menu do nosso algoritmo. Lá, declaramos uma variável para a raiz, que é um ponteiro. Essa variável é passada como parâmetro para nossa função de busca, e, portanto, temos um ponteiro para outro ponteiro;
- Linha 9: verificação de se o elemento atual sendo varrido não é nulo. Esse teste acontece porque, se o elemento atual estiver vazio, significa que o valor buscado não existe e que chegamos ao final da árvore binária;
- Linha 12: verifica se o valor buscado é menor que o valor naquela posição da árvore. Sendo menor, precisamos seguir pelo lado esquerdo dela;

- Linhas 12-14: verifica se o valor buscado é menor que o valor naquela posição da árvore. Sendo menor, precisamos seguir pelo lado esquerdo dela;
- Linhas 16-18: verifica se o valor buscado é maior que o valor naquela posição da árvore. Sendo maior, precisamos seguir pelo lado direito dela;
- Linhas 20-22: verifica se o valor buscado é igual ao valor naquela posição da árvore. Sendo igual, o valor desejado foi localizado e a busca pode ser encerrada momento.

É válido observar que cada nova busca na árvore é feita de forma recursiva, passando como parâmetro o próximo elemento a ser comparado.

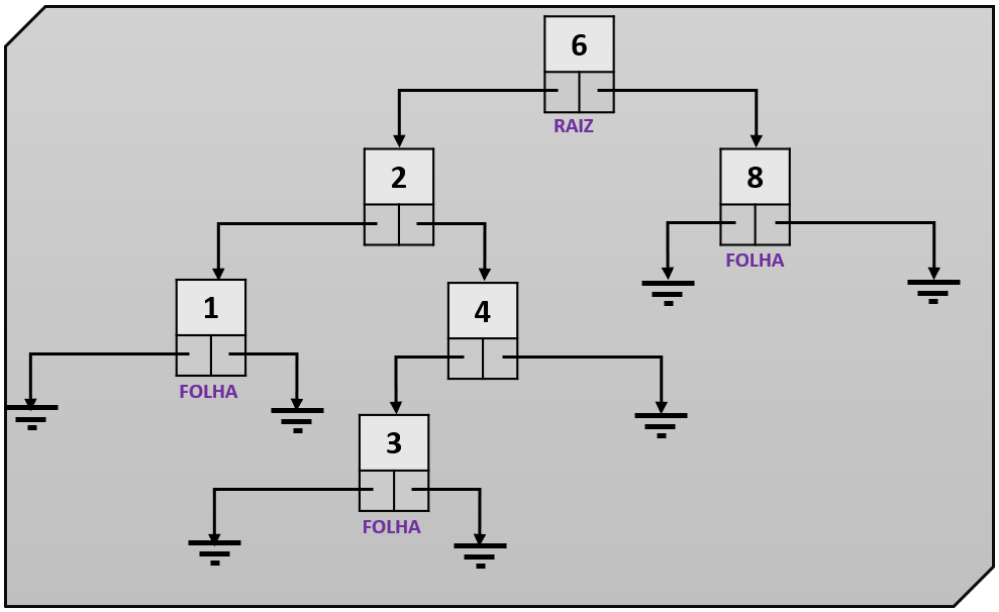
TEMA 4 – ÁRVORE BINÁRIA: VISUALIZAÇÃO DE DADOS

Uma árvore binária, depois de construída, pode ser listada/consultada e apresentada em uma interface. Porém, como a árvore não apresenta sequência/ordem fixas, podemos listar seus dados de algumas formas distintas. As consultas podem ser feitas:

- Em ordem: listamos os elementos iniciando pelos elementos da esquerda; depois, a raiz; e, por último, os elementos da direita. Dessa forma, os elementos listados são apresentados ordenados e de forma crescente;
- Em pré-ordem: listamos os elementos iniciando pela raiz, depois listamos os elementos da esquerda, e, por fim, os elementos da direita;
- Em pós-ordem: listamos os elementos iniciando pelos elementos da esquerda; depois, os da direita; e, por último, a raiz.

Analise as listagens possíveis observando a Figura 12.

Figura 12 – Exemplo de árvore binária para consulta/listagem



Para esse exemplo, a ordem em que os elementos seriam listados é:

- Em ordem: 1, 2, 3, 4, 6, 8;
- Em pré-ordem: 6, 2, 1, 4, 3, 8;
- Em pós-ordem: 1, 3, 4, 2, 8, 6.

Os pseudocódigos dos três tipos de listagem estão apresentados nas figuras 13, 14 e 15. Na Figura 13, temos a impressão em ordem, ou seja, com valores de forma crescente. Seguimos recursivamente pelo ramo esquerdo até o final, em seguida, imprimimos o valor atual e depois passamos para o lado direito da árvore.

De forma análoga, podemos observar que a Figura 14 inicia imprimindo o elemento atual. Como iniciamos sempre pela raiz, ela é a primeira a ser impressa. Em seguida, seguimos imprimindo pela esquerda e depois pela direita, recursivamente.

Por fim, na Figura 15, mostramos que a recursividade imprime pelo lado esquerdo. Depois, volta pela direita e, por fim, imprime a raiz.

Figura 13 – Pseudocódigo de consulta em ordem (esquerda, raiz, direita)

```
1   função VisualizarArvore_Ordem
2   (ElementoVarredura[->]: registro ElementoDaArvore): sem retorno
3   var
4   inicio
5       se (ElementoVarredura <> NULO) então
6           //Segue para a esquerda
7           VisualizarArvore_Ordem (ElementoVarredura->esquerda)
8           //Imprime o elemento atual
9           escreva(ElementoVarredura->dado)
10          //Segue para a direita
11          VisualizarArvore_Ordem (ElementoVarredura->direita)
12      fimse
13  fimfunção
```

Figura 14 – Pseudocódigo de consulta em pré-ordem (raiz, esquerda, direita)

```
15  função VisualizarArvore_PreOrdem
16  (ElementoVarredura[->]: registro ElementoDaArvore): sem retorno
17  var
18  inicio
19      se (ElementoVarredura <> NULO) então
20          //Imprime o elemento atual
21          escreva(ElementoVarredura->dado)
22          //Segue para a esquerda
23          VisualizarArvore_Ordem (ElementoVarredura->esquerda)
24          //Segue para a direita
25          VisualizarArvore_Ordem (ElementoVarredura->direita)
26      fimse
27  fimfunção
```

Figura 15 – Pseudocódigo de consulta em pós-ordem (esquerda, direita, raiz)

```
29  função VisualizarArvore_PosOrdem
30  (ElementoVarredura[->]: registro ElementoDaArvore): sem retorno
31  var
32  inicio
33      se (ElementoVarredura <> NULO) então
34          //Segue para a esquerda
35          VisualizarArvore_Ordem (ElementoVarredura->esquerda)
36          //Segue para a direita
37          VisualizarArvore_Ordem (ElementoVarredura->direita)
38          //Imprime o elemento atual
39          escreva(ElementoVarredura->dado)
40      fimse
41  fimfunção
```

TEMA 5 – ÁRVORE DE ADELSON-VELSKY E LANDIS (ÁRVORE AVL)

Uma **árvore de Adelson-Velsky e Landis**, também conhecida como **árvore AVL**, é uma **árvore binária balanceada**. Em uma árvore binária convencional, na medida em que temos muitas inserções de dados, podemos

começar a ter algumas ramificações que se estendem muito em altura, gerando piora no desempenho do algoritmo.

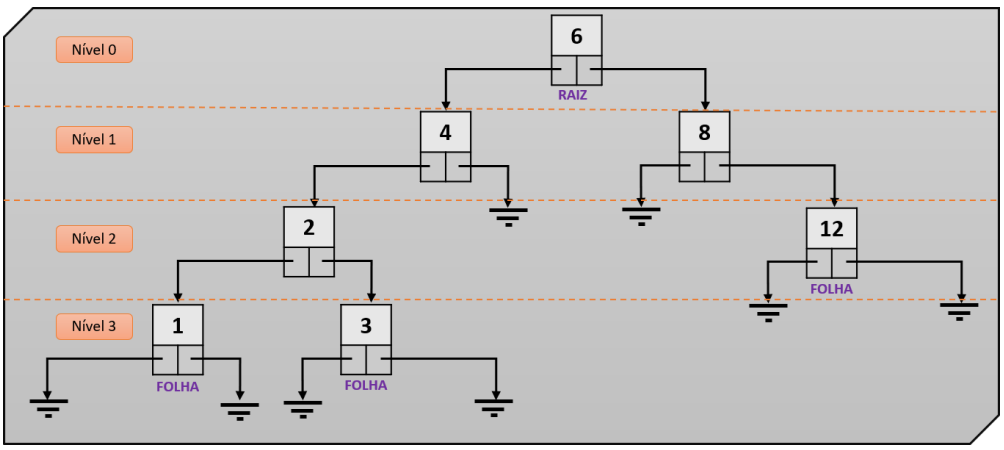
Ramificações muito altas acabam dificultando o sistema de busca em uma árvore, piorando o desempenho do algoritmo em termos de tempo de execução. A árvore AVL tem como objetivo melhorar esse desempenho balanceando uma árvore binária, evitando ramos longos e gerando o maior número de ramificações binárias possíveis.

Sendo assim, a árvore AVL contém todas as características já apresentadas de uma árvore binária. A única característica adicional é que **a diferença de altura entre uma subárvore direita e uma subárvore esquerda sempre deverá ser 1, 0 ou –1.** Caso essa diferença resulte em outro valor, como 2 ou –2, a árvore deverá ser imediatamente balanceada (seus elementos deverão ser rearranjados) por meio de um algoritmo de balanceamento.

Em uma árvore AVL, a complexidade para o pior caso de inserção, busca e visualização não é alterada, uma vez que os algoritmos até então apresentados continuam sendo válidos. Porém, conforme apresentaremos a seguir, uma árvore balanceada sempre apresentará, no máximo, o mesmo número de níveis que uma não balanceada, normalmente apresentando menos. Desse modo, menos níveis e mais ramificações significam que as funções de inserção, busca e visualização serão executadas mais rapidamente, pois teremos menos posições a percorrer.

Observe a Figura 16. Nela, vemos uma árvore não balanceada. Todos os elementos inseridos estão colocados na sequência em que são solicitados.

Figura 16 – Exemplo de árvore binária não balanceada



O balanceamento de um elemento da árvore é verificado da seguinte maneira:

- **Passo 1:** calculamos a altura relativa daquele elemento para o lado direito da árvore. Nesse caso, pegamos o nível mais alto do lado direito daquele elemento e subtraímos do nível do elemento desejado;
- **Passo 2:** calculamos a altura relativa daquele elemento para o lado esquerdo da árvore. Nesse caso, pegamos o nível mais alto do lado esquerdo daquele elemento e subtraímos do nível do elemento desejado;
- **Passo 3:** tendo as alturas direita e esquerda calculadas, fazemos a diferença entre elas (direta menos esquerda, sempre). Se o cálculo resultar em 2 ou –2, existe um desbalanceamento e uma rotação será necessária.

Calcule o balanceamento da árvore na Figura 16 para cada elemento. Em negrito, destacamos o elemento desbalanceado e que precisa ser corrigido.

Tabela 1 – Cálculo de balanceamento para a árvore da Figura 16

Elemento	**Altura direita**	**Altura esquerda**	**Direita– esquerda**	**Balanceado?**
6	2 – 0 – 2	3 – 0 = 3	3 – 2 = 1	Sim
4	0	3 – 1 = 2	0 – 2 = –2	Não
8	2 – 1 = 1	0	1 – 0 = 1	Sim
12	0	0	0	Sim
2	3 – 2 = 1	3 – 2 = 1	1 – 1 = 0	Sim
1	0	0	0	Sim
3	0	0	0	Sim

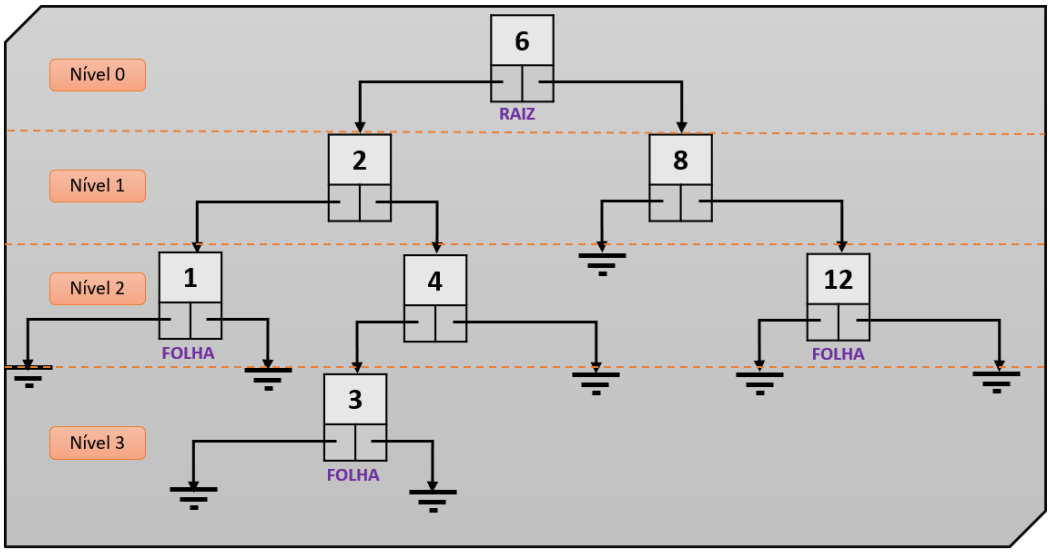
O elemento 4 não está balanceado, pois a diferença de altura entre o lado direito e o lado esquerdo resultou em –2. Para um balanceamento é obrigatório que essa diferença seja –1, 0 e 1.

Podemos reescrever a árvore anterior de uma maneira diferente, rearranjando os elementos não balanceados de maneira que as diferenças de níveis resultem em –1, 0 ou 1. A Figura 17 ilustra uma árvore binária com os mesmos elementos da Figura 16, porém, balanceada. Calcule o balanceamento dela árvore para cada elemento:

Tabela 2 – Cálculo de balanceamento para a árvore da Figura 17

Elemento	Altura direita	Altura esquerda	Direita–Esquerda	Balanceado?
6	2 – 0 – 2	3 – 0 = 3	3 – 2 = 1	Sim
2	3 – 1 = 2	2 – 1 = 1	2 – 1 = 1	Sim
8	2 – 1 = 1	0	1 – 0 = 1	Sim
12	0	0	0	Sim
1	0	0	0	Sim
4	0	3 – 2 = 1	0 – 1 = –1	Sim
3	0	0	0	Sim

Figura 17 – Exemplo de árvore binária balanceada (árvore AVL)



5.1 Rotacionando a árvore binária

Vimos a diferença entre uma árvore não balanceada (Figura 16) e uma árvore balanceada AVL (Figura 17). Porém, como partir de uma árvore não balanceada e gerar balanceamento?

Para isso, precisamos realizar rotações em nossa árvore para balanceá-la. Na tabela a seguir, vemos o procedimento de rotação que precisa ser realizado, conforme indica Ascencio (2011).

Tabela 3 – Possibilidades de rotações da árvore binária

Diferença de altura de um nó	Diferença de altura entre o nó filho e o nó desbalanceado	Tipo de rotação
2	1	Simples à esquerda
2	0	Simples à esquerda
2	–1	Dupla com filho para a direita e pai para a esquerda
–2	1	Dupla com filho para a esquerda e pai para a direita
–2	0	Simples à direita
–2	–1	Simples à direita

Fonte: Ascencio, 2011.

Acompanhe o desbalanceamento da Figura 16. O elemento 4 está com desbalanceamento de –2. O nó filho do nó 4, que é o nó 2, está balanceado com valor 0. Desse modo, segundo a tabela de rotações AVL, devemos fazer uma **rotação simples à direta**.

Essa rotação implica colocar o nó 2 no lugar do nó 4; assim, todos os elementos abaixo dele são rearranjados. O resultado final dessa rotação já foi apresentado na Figura 17.

Todas as outras rotações não mostradas neste material (simples à esquerda; dupla com filho à esquerda e pai à direita; e dupla com filho à direita e pai à esquerda) são detalhadas em Ascencio (2011, p. 343). Consulte a autora para exemplos gráficos de cada um dos outros tipos de rotações. Os algoritmos de rotação também são apresentados em Ascencio (2011, p. 359).

FINALIZANDO

Nesta aula, aprendemos sobre estrutura de dados do tipo árvore binária. Aprendemos que árvore binárias são um tipo de estrutura de dados não linear, ou seja, seus elementos não são acessados de forma sequencial. Assim, uma aplicação muito interessante para árvores é aquela de sistemas de busca de dados.

Árvore binárias contêm sempre um nó chamado de raiz, que é o nó inicial da árvore (equivalente ao *head* da lista). Uma estrutura de árvore conterá elementos com dois ponteiros: um que ramifica a árvore para o lado esquerdo e outro que faz o mesmo para o lado direito.

A árvore binária pode conter, no máximo, dois nós filhos para cada nó pai. Todos os elementos da árvore binária são inseridos sempre nos níveis

mais altos das árvores. Valores menores que aqueles de um nó pai são inseridos à esquerda, e, valores maiores, à direita do elemento.

Apresentamos a inserção na árvore, de forma recursiva e com complexidade $O(\log n)$. Também vimos como é feita a busca de um elemento na árvore e a listagem da árvore de três maneiras distintas: em ordem, em pré-ordem e em pós-ordem.

Por fim, vimos como balancear uma árvore binária, gerando uma árvore chamada de AVL. A partir de um algoritmo de rotação de elementos da árvore, podemos balanceá-la, gerando menos níveis e mais ramificações, melhorando o desempenho dessa estrutura de dados em termos de tempo de execução.

REFERÊNCIAS

ASCENCIO, A. F. G. **Estrutura de dados**: algoritmos, análise da complexidade e implementações em JAVA e C/C++. São Paulo: Pearson, 2011.

_____. **Fundamentos da programação de computadores**: algoritmos, Pascal, C/C++ (padrão ANSI) JAVA. 3. ed. São Paulo: Pearson, 2012.

CORMEN, T. H. **Algoritmos**: teoria e prática. 3. ed. Rio de Janeiro: Elsevier, 2012.

LAUREANO, M. **Estrutura de dados com algoritmos e C**. Rio de Janeiro: Brasport, 2008.

MIZRAHI, V. V. **Treinamento em linguagem C**. 2. ed. São Paulo: Peason, 2008.

PUGA, S.; RISSETI, G. **Lógica de programação e estrutura de dados**. 3. ed. São Paulo: Pearson, 2016.