ESTRUTURA DE DADOS

AULA 4

Prof. Vinicius Pozzobon Borin

CONVERSA INICIAL

Aqui, você irá investigar e aprender a manipular estruturas não lineares de dados estudando a estrutura de árvore binária. Essa estrutura tem como principal aplicação operar em sistemas de busca, pois é construída de tal forma que facilita a localização dos seus elementos.

O objetivo desta etapa é apresentar os conceitos que envolvem a estrutura de dados do tipo árvore. Ao longo deste documento, será conceituada a estrutura de uma árvore e serão investigados dois tipos bastante comuns de árvores. Será apresentado como realizar as manipulações dos dados dentro de uma estrutura de árvore, como a inserção e a remoção dos dados. Os tipos de árvores estudadas serão:

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

TEMA 1 – ÁRVORES BINÁRIAS

Imagine que você é responsável por implementar um vasto sistema de cadastro e armazenamento de dados de usuário de uma rede social. A quantidade de usuários será imensa, certo? Você precisa implementar uma estrutura de dados que seja eficiente na busca dentro desta rede, caso contrário, localizar um dado tomará muito tempo.

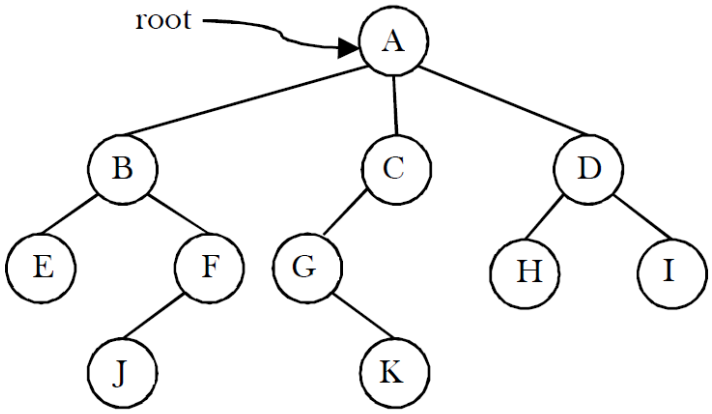
Uma opção inicial e que talvez esteja pensando é implementar um *array* e trabalhar com o algoritmo de busca binária. Caso siga por esse caminho, verá que precisará de um *array* longo demais. Além disso, caso precise inserir um novo dado nesse *array*, terá um tremendo processamento matemático para colocar os dados na posição certa, gastando processamento e memória.

E se você implementar a solução com uma lista encadeada, resolveria o problema? Também não. Lista encadeadas resolveriam o problema de inserção de um dado em qualquer posição, mas não são nada eficientes quando se trata de busca de dados. Isso porque uma lista encadeada é linear, e cada elemento só conhece seu sucessor (e anteceder se for uma lista dupla).

Uma possível solução para esse tipo de problema é o emprego de uma estrutura não linear, a árvore, como uma árvore binária de busca. Uma estrutura de dados é denominada de árvore quando seus elementos criam ramificações na estrutura, gerando **subárvores**. Esses ramos podemos chamar

também de galhos. A Figura 1 ilustra um exemplo de uma árvore ternária, ou seja, em que um elemento pode se ramificar em até três outros. Existem diferentes classificações para árvores e, de um modo geral, podemos chamá-las de **árvores N-árias**.

Figura 1 – Exemplo de uma estrutura de árvore, neste caso, ternária



1.1 Árvores binárias e suas propriedades

Uma lista encadeada apresenta uma característica importante: cada elemento da lista tem acesso somente ao próximo elemento (lista simples) ou ao próximo e ao anterior (lista dupla). Todos os elementos dessa lista são organizados de tal forma que é necessário percorrer, fazer uma varredura, da estrutura de dados para se ter acesso ao elemento desejado. Chamamos uma **lista encadeada** de uma estrutura de dados com **organização linear**.

As árvores, como uma árvore binária, são estrutura de dados não lineares, em que são organizadas com elementos que não estão, necessariamente, encadeados, formando ramificações e subdivisões na organização da estrutura de dados.

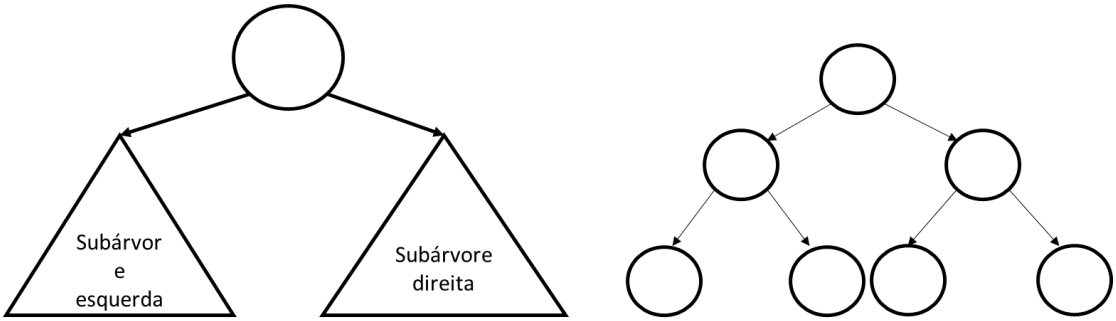
A árvore binária é um tipo de árvore e é também a mais simples de ser estudada. Nesta etapa, focaremos somente nossos estudos em árvores binarias, as quais apresentam algumas características distintas, mesmo se comparando com outros tipos de árvores. Vamos analisar algumas dessas características e aproveitar para conceituar alguns termos próprios referentes a árvores:

- Nó raiz (*root*) – Nó original da árvore. Todos derivam dele;
- Nó pai – Nó que dá origem (está acima) de pelo um outro nó;
- Nó filho – Nó que deriva de um nó pai; e

- Nó folha/terminal – Nó que não contém filhos.

Uma árvore é chamada de binária quando cada elemento (nó/nodo) da árvore contém, no máximo, dois (bi) filhos. Cada elemento da árvore pode gerar até duas subárvores. A partir da raiz, podemos dizer que ela irá criar duas ramificações. Chamaremos uma de subárvore esquerda, e a outra de subárvore direita, conforme Figura 2.

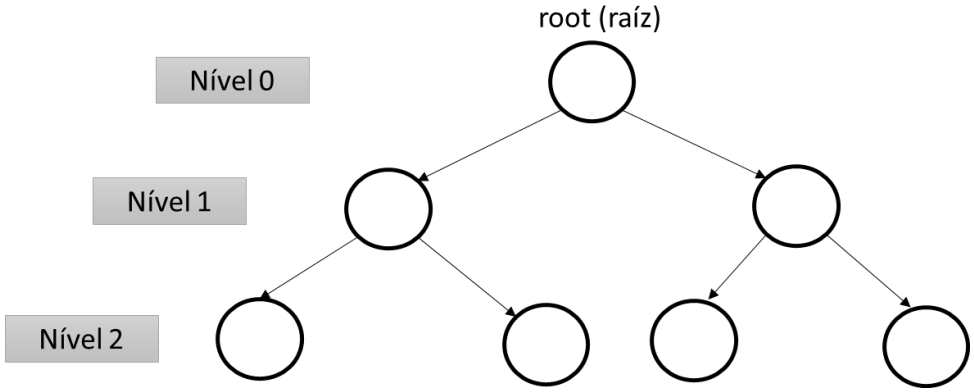
Figura 2 – Árvore binária e suas subárvores



1.2 Níveis, altura e profundidade de árvores binárias

Cada nova ramificação de uma árvore representa um nível. Dizemos que a raiz é sempre o nível 0. A partir dela, cada nova geração de filhos recebe um nível a mais, veja na Figura 3.

Figura 3 – Níveis na árvore binária



O conceito de nível de árvore é importante, pois eventualmente precisamos saber a profundidade de uma árvore binária (para fins de balanceamento, por exemplo, como veremos ainda nesta etapa). Para

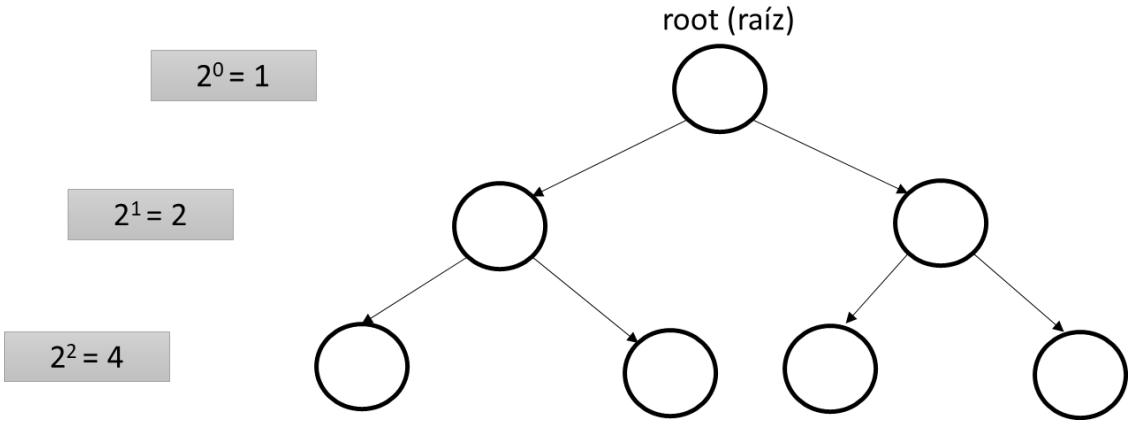
encontrarmos a **profundidade** de um nó da árvore, basta sabermos o nível daquele nó na árvore em relação à nossa raiz. Exemplo, se desejarmos saber a profundidade de um elemento no nível 2 da árvore, fazemos: 2 – 0 = 2. Portanto, a profundidade é 2.

O conceito de altura é similar à profundidade. A diferença é que a **altura** pode ser calculada pela diferença de nível entre dois nós. Por exemplo, a altura de um elemento no nível 2 da árvore, em relação à raiz, é 2 – 0 = 2, mesmo valor da profundidade. Agora, a diferença entre um elemento no nível 2, em relação ao nível 1, é 2 – 1 = 1.

1.3 Total de elementos por nível na árvore binária

Qual a quantidade máxima de elemento em uma árvore binaria, considerando que sabemos quantos níveis ela tem? Vejamos a Figura 4. Por se tratar de uma árvore binária, a cada nova ramificação, dobramos a quantidade máxima de elementos do nível anterior.

Figura 4 – Total de elementos por nível na árvore binária



Se nossa árvore tiver somente uma raiz, temos um só elemento. Se a árvore tiver um segundo nível, teremos mais dois elementos, portanto, três no total. Se a árvore tiver um terceiro nível, teremos mais quatro elementos, portanto, sete no total. Vejamos a tabela a seguir que resume para nós essa explicação.

Nível	Nós no nível	Total de elemento por profundidade
0	$2^0 = 1$	$2^1 - 1 = 1$
1	$2^1 = 2$	$2^2 - 1 = 3$
2	$2^2 = 4$	$2^3 - 1 = 7$
3	$2^3 = 8$	$2^4 - 1 = 15$
4	$2^4 = 16$	$2^5 - 1 = 31$

5	$2^5 = 32$	$2^6 - 1 = 63$

Note que, para sabermos o total de elementos possível em uma árvore binária completa, podemos generalizar o cálculo da terceira coluna da tabela para:

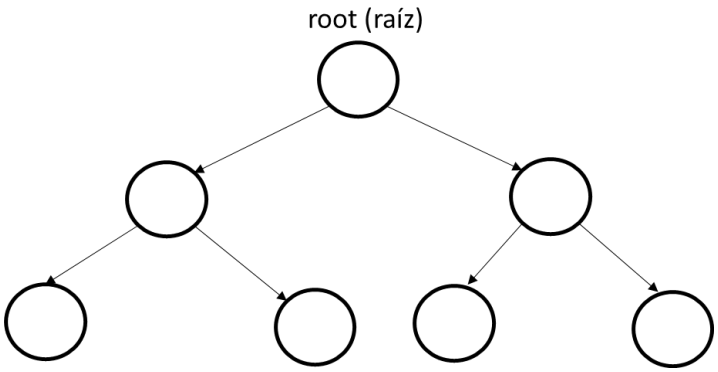
$$Total_{nós\ por\ profundidade} = 2^{nível+1} - 1$$

1.4 Tipos de árvores binárias

Precisamos definir algumas nomenclaturas para árvores binárias. Uma **árvore estritamente binária** é aquela em que cada nó contém, sempre, exatamente dois filhos, ou nenhum. Uma **árvore binária completa** é aquela em que todos os nós contendo menos de dois filhos estão colocados somente no último ou no penúltimo nível da árvore.

A **árvore binária cheia** é assim denominada quando cada nó tem exatamente dois filhos, e os nós folhas estão sempre no mesmo nível. Ou seja, é quando a árvore é estritamente binária e completa ao mesmo tempo. A Figura 5 apresenta uma árvore binária cheia.

Figura 5 – Árvore binária cheia



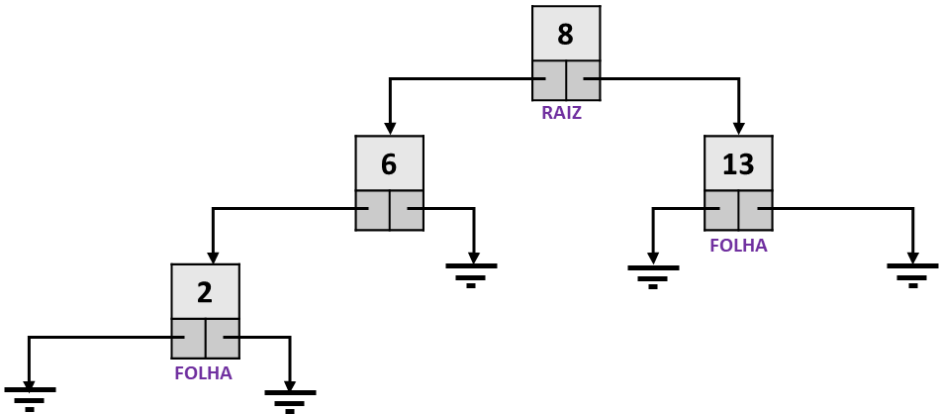
TEMA 2 – ÁRVORE BINÁRIA DE BUSCA

Para que uma árvore binária opere de maneira mais eficiente para busca, precisamos cadastrar seus dados de maneira organizada para facilitar a busca. Suponha uma árvore de valores numéricos, quando vamos realizar a inserção de um número na árvore, pode adotar a seguinte lógica.

- Caso o valor a ser inserido seja menor do que o seu nó pai, inserimos o dado na subárvore esquerda.
- Caso o valor a ser inserido seja maior do que o seu nó pai, inserimos o dado na subárvore direta.

Esse tipo de organização dos dados resulta em um tipo particular de árvore binária, conhecida como **árvore binária de busca**, ou, ainda, ***Binary Search Tree* (BST)**. Veja uma representação dela a seguir.

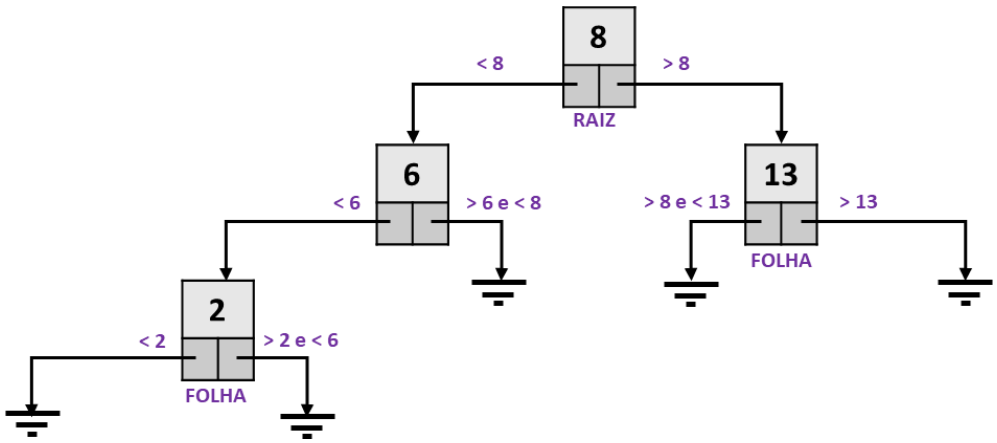
Figura 6 – BST numérica



Note que, no exemplo da Figura 6, a raiz corresponde ao valor 8. Portanto, para a esquerda de 8, só podemos ter valores menores do que ele, e à direita, maiores. Assim, colocamos 13 à direta e 6 à esquerda. O valor 2 é colocado ainda à esquerda de 6, pois é menor do que ele.

Quando precisamos inserir um novo dado dentro da árvore, a inserção precisa ocorrer na posição certa. A Figura 7 apresenta os intervalos em que podemos inserir os dados numéricos.

Figura 7 – BST numérica com informação de inserção

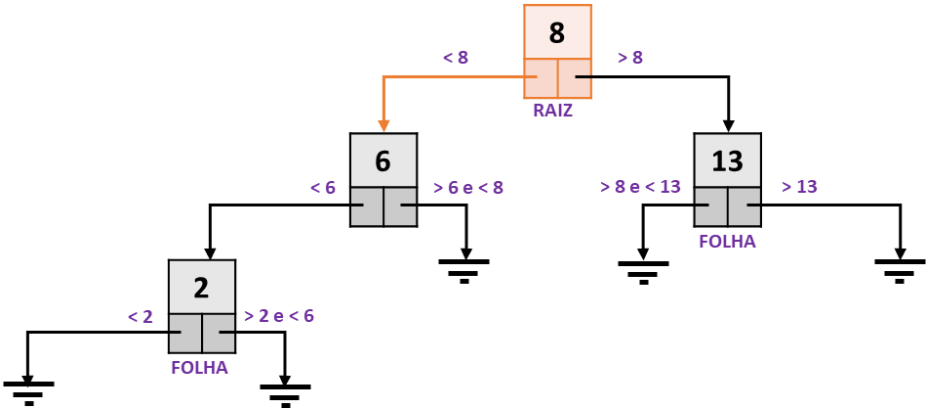


2.1 Inserção na árvore binária de busca

Imagine que temos que inserir o valor 7 na BST apresentada na Figura 7. A árvore só tem armazenado no seu programa a posição da raiz, e todos os outros elementos são obtidos a partir delas e seus ponteiros. Portanto, precisamos localizar o local correto para inserir o valor 7.

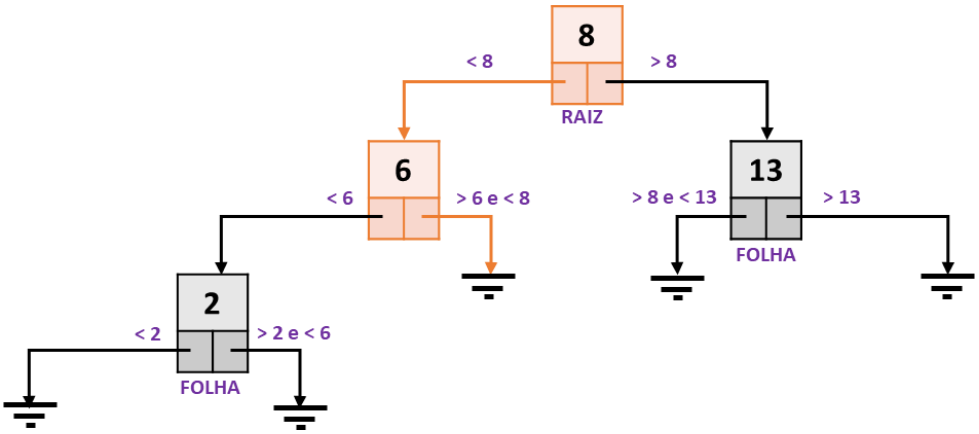
Iniciamos a procura pelo local correto de inserção pela raiz, verificando se o 7 é maior ou menor que o valor posicionado na raiz (valor 8). Como 7 < 8, seguimos pelo ramo esquerdo da árvore, conforme Figura 8.

Figura 8 – Comparação do valor 7 com a raiz 8



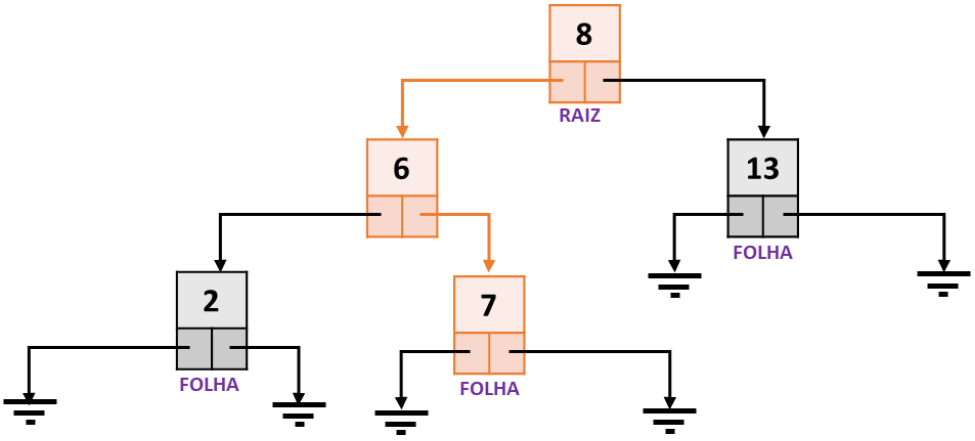
Em seguida, precisamos comparar o valor 7 com o valor 6. Como 7 é maior do que 6, seguimos para o ramo direito do 6. Veja na Figura 9.

Figura 9 – Comparação do valor 7 com o valor 6



Para a direita de 6 não existe nada. Temos um elemento nulo/vazio. Isso significa que podemos inserir ali. A Figura 10 apresenta a nova BST com o valor inserido na posição correspondente.

Figura 10 – Inserção do valor 7 na BST



Vamos agora implementar uma BST em linguagem Python. Para isso, criaremos uma classe que irá representar a árvore binária. Se observar as Figuras 6-10, verá que cada elemento da árvore foi representado com dois ponteiros. Aqui iremos chamar os ponteiros de ponteiro esquerdo e ponteiro direito, representando os ramos. Veja o código a seguir.

```
class BST:
    def __init__(self, dado=None):
        self.dado = dado
        self.esquerda = None
        self.direita = None

    def inserir(self, dado):
        if (self.dado == None):
            self.dado = dado
        else:
            if (dado < self.dado):
                if (self.esquerda): #self.esquerda == None:
                    self.esquerda.inserir(dado)
                else:
                    self.esquerda = BST(dado)
            else:
                if(self.direita): #self.direita == None:
                    self.direita.inserir(dado)
                else:
                    self.direita = BST(dado)
```

No constructo de inicialização (__init__), definimos os valores inicias de cada elemento da árvore. A seguir, criamos o método *inserir*, para realizar a

inserção de um dado na árvore binária. A inserção em uma árvore binária só pode ocorrer onde existem ponteiros vazios, ou seja, somente nos ramos finais dessa árvore. Não é permitido inserir no meio de uma BST.

O método de inserção realiza esse procedimento de maneira recursiva. Isto é, o método verifica se a posição a qual ele está é passível de inserção (está vazia). Caso não esteja vazia, a função vai sendo chamada recursivamente até que seja possível inserir.

```
Teste = BST()

Teste.inserir(7)
Teste.inserir(4)
Teste.inserir(9)
Teste.inserir(0)
Teste.inserir(5)
Teste.inserir(8)
Teste.inserir(13)
```

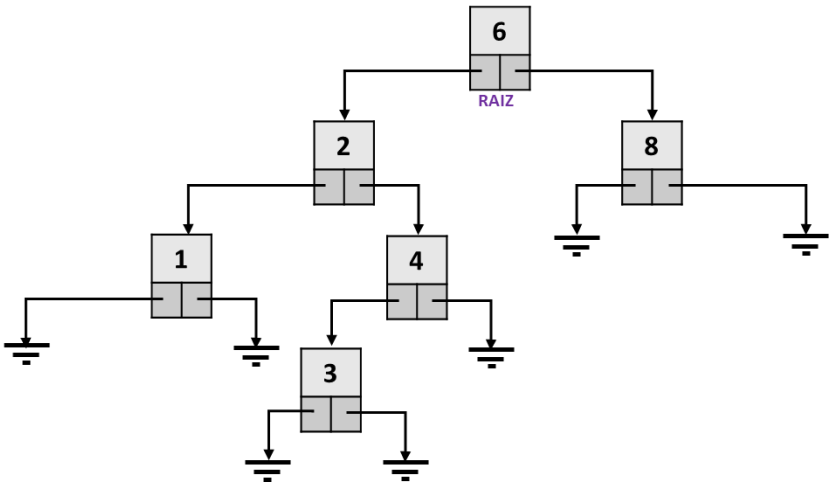
TEMA 3 – PERCURSO EM BST

Como percorremos uma BST se não existe um só caminho? Devemos iniciar na raiz sempre? Depende do que deseja e da sua aplicação. Uma árvore binária, por não ser uma estrutura linear, apresenta distintas maneiras de se percorrer por ela para visualizar, manipular ou processar os dados da árvore. Vamos investigar todas essas maneiras agora.

3.1 Percurso em largura

O percurso em largura na árvore binária, também conhecido como percurso em nível, ou *level order*, em inglês.

Figura 11 – Exemplo de árvore binária para percurso



Essa varredura na árvore funciona da seguinte maneira: na Figura 11, é vista uma árvore. O percurso em nível obriga que a árvore passe por todos os elementos de um mesmo nível, para depois passar para os elementos do próximo nível.

O percurso sempre ocorre de cima para baixo (nível 0 em diante) e da esquerda para a direita. Isso significa que, em nosso exemplo, o percurso iniciaria imprimindo na tela (ou manipulando) a raiz, pois é o único dado de nível 0. Em seguida, iria para o nível 1, da esquerda para a direta (2 e 8). Após, nível 2, da esquerda para a direita (1 e 4). E, por fim, nível 3, com o valor 3. Se a proposta for imprimir os dados na tela, ficaremos com a respectiva sequência impressa: 6, 2, 8, 1, 4, 3.

```
def emNivel(self):
    nodoAtual = self
    lst = []
    fila = []
    fila.insert(0,nodoAtual)
    while(len(fila) > 0):
        nodoAtual = fila.pop()
        lst.append(nodoAtual.dado)
        if(nodoAtual.esquerda):
            fila.insert(0,nodoAtual.esquerda)
        if(nodoAtual.direita):
            fila.insert(0,nodoAtual.direita)

    return lst
```

3.2Percurso em profundidade

A segunda maneira de percorrer uma árvore binaria é em profundidade. Neste caso, a árvore não é percorrida horizontalmente como no percurso em largura, mas, sim, é definido um caminho (galho da árvore) e vamos até o fundo dele antes de retornarmos para escolher outro. Existem três formas distintas de percorrer em profundidade. As possibilidades são as que se seguem.

- *Percorrer em ordem:* listamos os elementos iniciando pelos elementos da esquerda, depois a raiz e, por último, os elementos da direita. Desta forma, os elementos listados ficarão apresentados ordenados de forma crescente.

```
def emOrdem(self, lst):
    if (self.esquerda):
        self.esquerda.emOrdem(lst)
    lst.append(self.dado)
    if (self.direita):
        self.direita.emOrdem(lst)
    return lst
```

- *Percorrer em pré-ordem:* listamos os elementos iniciando pela raiz, depois listamos os elementos da esquerda, e, por fim, os elementos da direita.

```
def preOrdem(self, lst):
    lst.append(self.dado)
    if (self.esquerda):
        self.esquerda.preOrdem(lst)
    if (self.direita):
        self.direita.preOrdem(lst)
    return lst
```

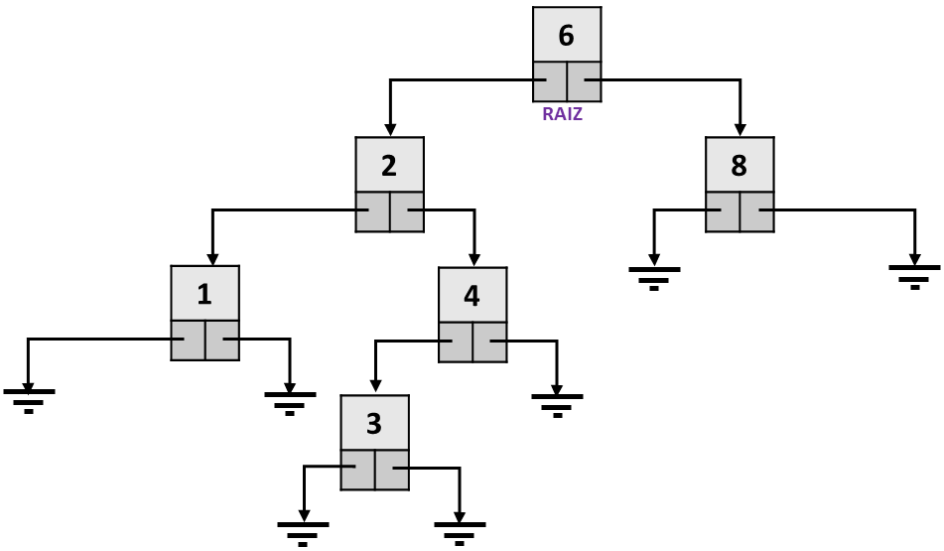
- *Percorrer em pós-ordem:* listamos os elementos iniciando pelos elementos da esquerda, depois os da direita e, por último, a raiz.

```
def posOrdem(self, lst):
    if (self.esquerda):
        self.esquerda.posOrdem(lst)
    if (self.direita):
        self.direita.posOrdem(lst)
    lst.append(self.dado)
    return lst
```

Vamos analisar as listagens possíveis, observando a Figura 12. Para este exemplo, a ordem em que os elementos seriam listados é o que se segue.

- Consultar em ordem: 1, 2, 3, 4, 6, 8.
- Consultar em pré-ordem: 6, 2, 1, 4, 3, 8.
- Consultar em pós-ordem: 1, 3, 4, 2, 8, 6.

Figura 12 – Exemplo de árvore binária para consulta/listagem

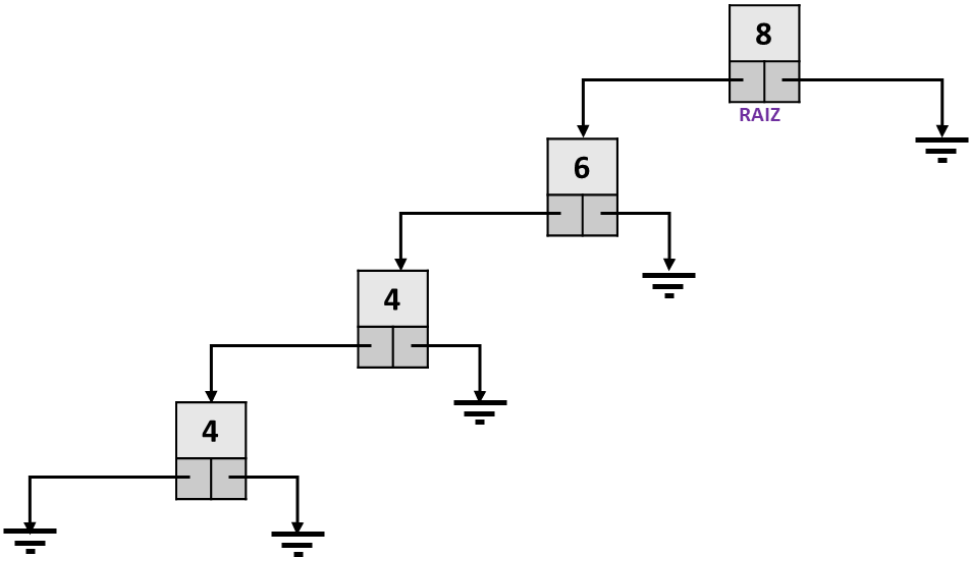


TEMA 4 – ÁRVORE DE ADELSON-VELSKII E LANDIS (AVL)

Uma árvore de Adelson-Velskii e Landis, também conhecida como árvore AVL, é uma **árvore binária balanceada**. Mas, afinal, o que é uma árvore balanceada?

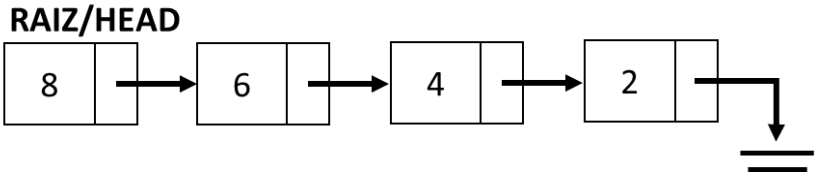
Bom, primeiro precisamos compreender o que é uma árvore desbalanceada. Observe a árvore binária da Figura 13. Note que essa árvore contém unicamente um ramo que segue para o lado esquerdo. É uma árvore sem ramificações.

Figura 13 – Exemplo de árvore binária desbalanceada



Em uma árvore binária convencional, a medida com que vamos tendo muitas inserções de dados, podemos começar a ter nessa árvore algumas ramificações que se estendem muito em altura, gerando piora no desempenho do algoritmo de busca. Veja que, no exemplo anterior, a árvore criada ficou linear, e isso significa que ela irá operar exatamente como uma lista encadeada, afinal, poderíamos reescrever a árvore binária anterior como a lista encadeada apresentada na Figura 14.

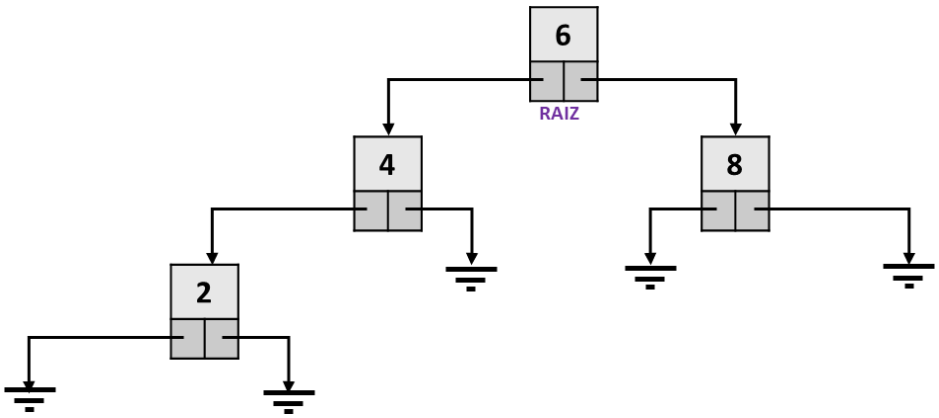
Figura 14 – Árvore binária desbalanceada, com um só ramo, reescrita como uma lista encadeada



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

A mesma árvore binária apresentada poderia ter seus elementos rearranjados para que tenha o maior número possível de ramificações. A solução balanceada está apresentada na Figura 15. Note que geramos algumas ramificações que irá facilitar a busca nessa árvore.

Figura 15 – Árvore binária balanceada



4.1 Características da árvore AVL

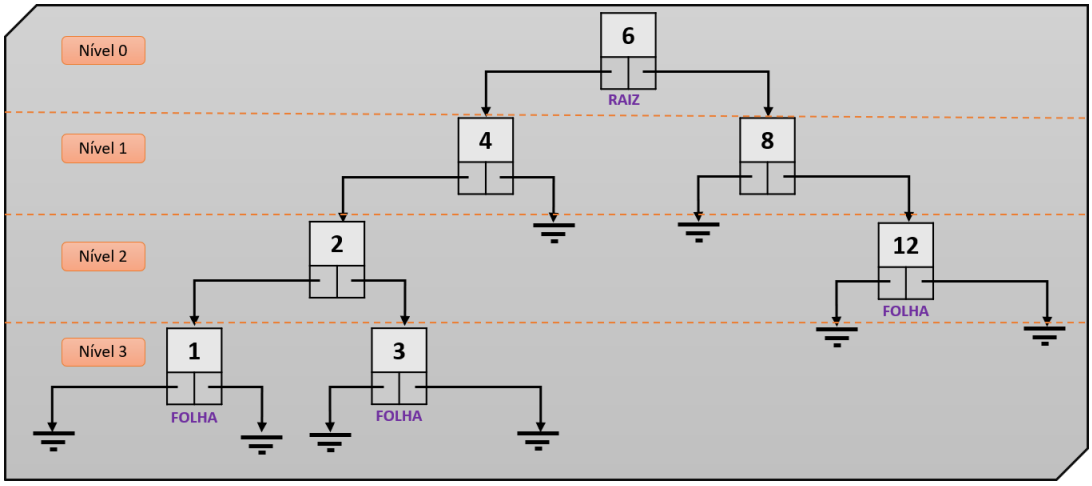
Uma árvore AVL é uma árvore binária de busca balanceada. Ou seja, ela contém exatamente as mesmas características de uma BST, todavia, com uma propriedade a mais: **para cada nó da árvore, a diferença de altura entre a subárvore da direta e da subárvore da esquerda será sempre, no máximo, um**. Isso significa que as subárvores daquele nó sempre terão a mesma altura (diferença zero), ou existirá uma diferença de altura entre as subárvores, mas ela é de, no máximo, um (em módulo).

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

- Passo 1: calcula-se a altura relativa daquele elemento para o lado direito da árvore. Neste caso, pegamos o nível mais alto do lado direito daquele elemento e subtraímos do nível do elemento desejado.
- Passo 2: calcula-se a altura relativa daquele elemento para o lado esquerdo da árvore. Neste caso, pegamos o nível mais alto do lado esquerdo daquele elemento e subtraímos do nível do elemento desejado.
- Passo 3: tendo a altura direita e a esquerda calculada, 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 na árvore.

Temos, na Figura 16, um exemplo de árvore não balanceada.

Figura 16 – Exemplo de árvore binária não balanceada



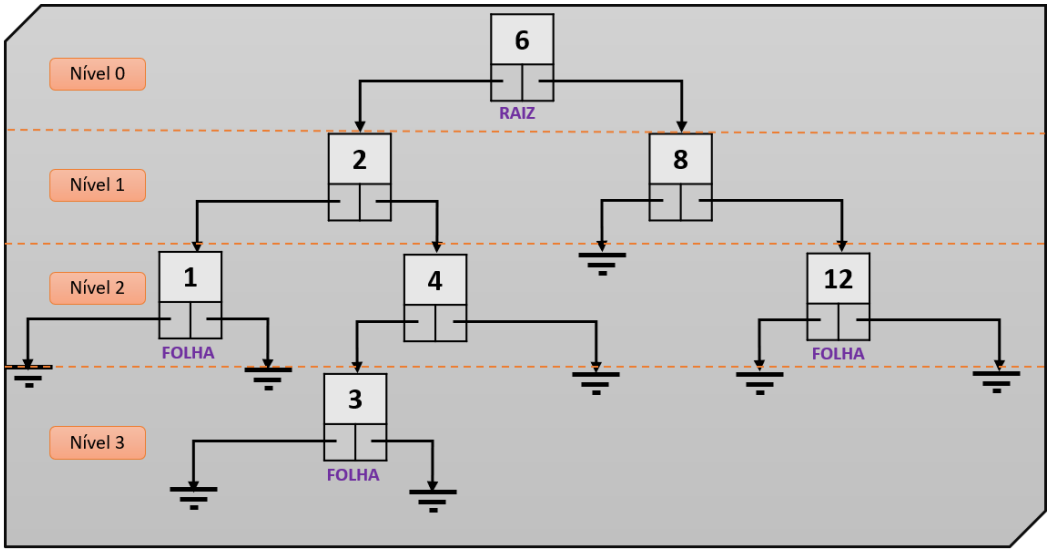
Vamos calcular o balanceamento dessa árvore da Figura 16 para cada elemento. Em destaque, foi colocado o elemento desbalanceado e que precisará ser corrigido.

Elemento	Altura Direita	Altura Esquerda	Direita - Esquerda	Balanceado?
6	2 – 0 – 2	3 – 0 = 3	2 – 3 = -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 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 uma 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, agora balanceada com o algoritmo AVL.

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



Vamos recalcular o balanceamento dessa árvore para cada elemento:

Elemento	Altura Direita	Altura Esquerda	Direita – Esquerda	Balanceado?
6	2 – 0 – 2	3 – 0 = 3	2 – 3 = -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

TEMA 5 – ROTAÇÕES DA ÁRVORE AVL

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

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

Diferença de altura de um nó	Diferença de altura do nó filho do 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

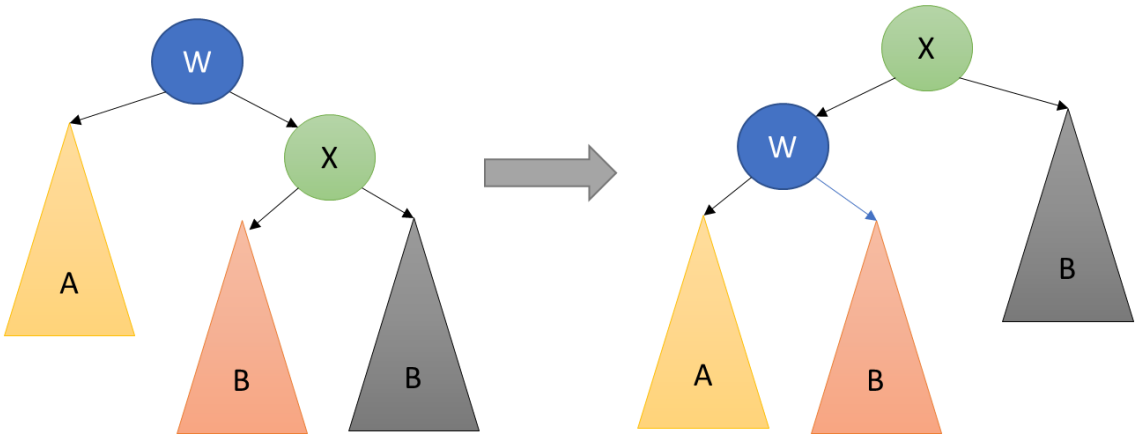
Vamos acompanhar nosso desbalanceamento da Figura 16. O elemento 4 está com desbalanceamento de -2. O nó filho do nó 4, que é o nó 2, está balanceado com valor 0. Deste modo, segundo a tabela de rotações AVL, devemos fazer uma rotação simples à direta.

Essa rotação implicará em colocar o nó 2 no lugar do nó 4. E todos os elementos a seguir dele são rearranjados. O resultado final dessa rotação já foi apresentado na Figura 17. Veremos, a seguir, em mais detalhes como são realizadas as rotações da tabela.

5.1 Rotação simples à esquerda

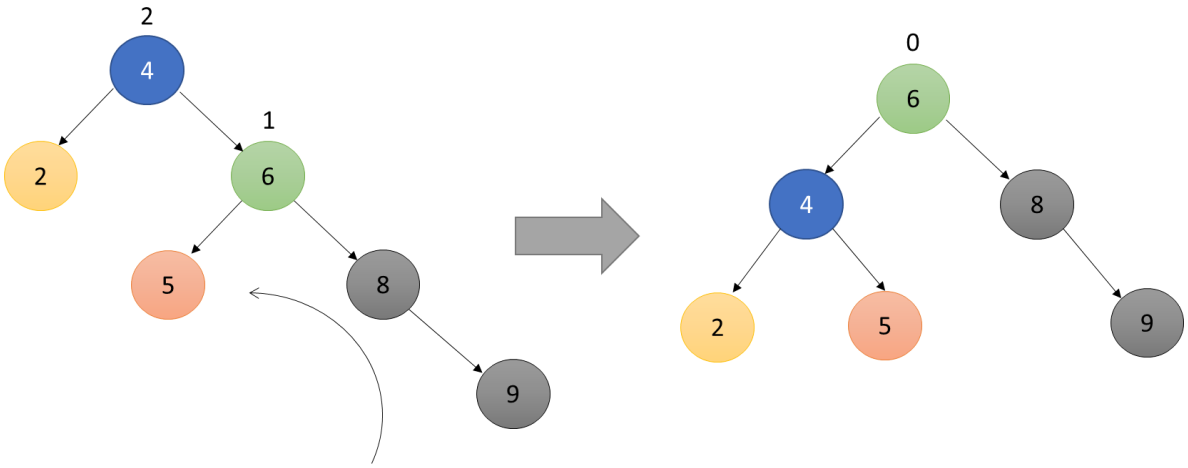
Na rotação simples à esquerda, iremos rotacionar diversos elementos no sentido anti-horário. Veja de maneira genérica na Figura 18, no exemplo, a raiz é W. Com a rotação, levamos ela para a esquerda, deixando de ser raiz. A nova raiz agora será o elemento X, que antes era filho à direta de W.

Figura 18 – Rotação simples à esquerda



A Figura 19 ilustra uma rotação à esquerda com uma BST. Foram empregadas as mesmas cores da Figura 18 para facilitar a identificação dos elementos. Antes da rotação, tínhamos a raiz desbalanceada (balanço 2). E o filho para a direita com balanceamento 1. Isso resulta em uma rotação simples à esquerda.

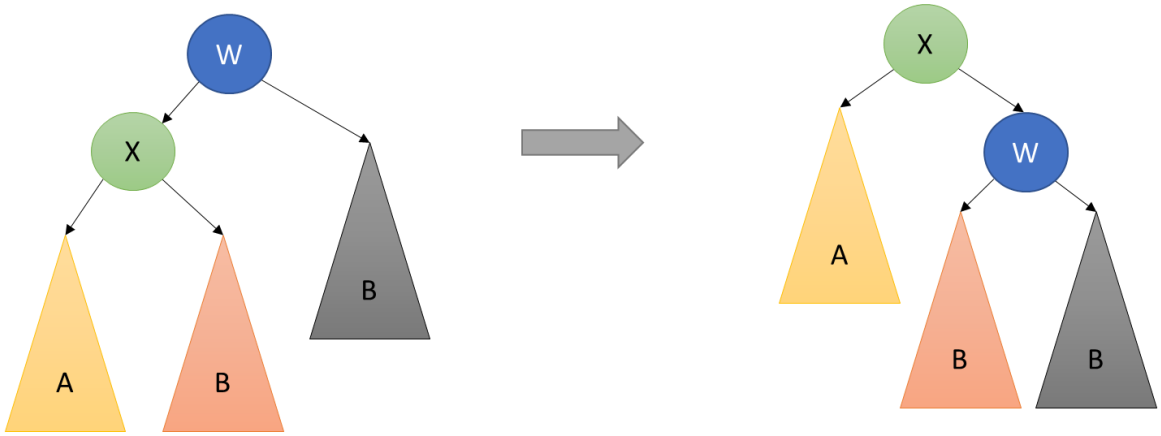
Figura 19 – Rotação simples à esquerda. Exemplo prático



5.2 Rotação simples à direita

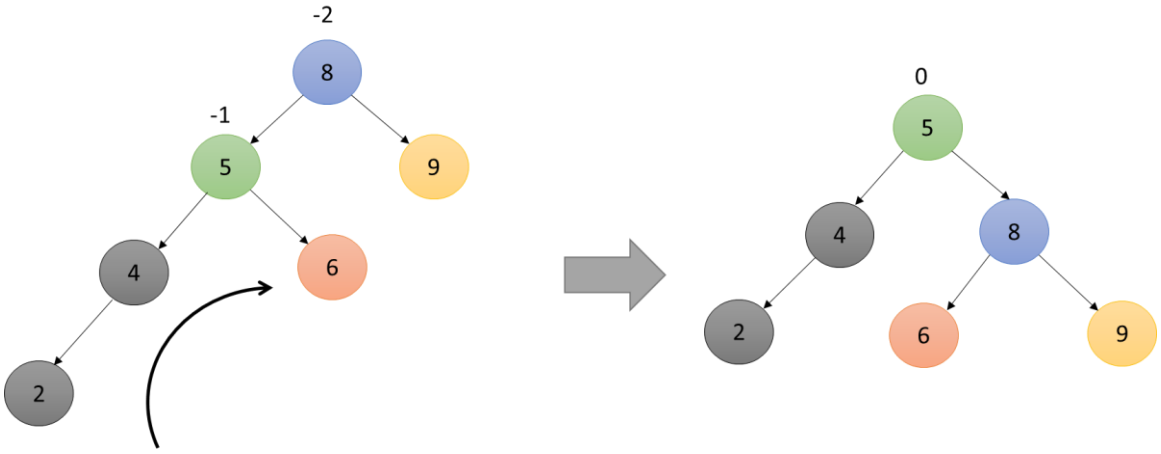
Na rotação simples à direita, iremos rotacionar diversos elementos no sentido horário. Veja de maneira genérica na Figura 20. No exemplo, a raiz é W. Com a rotação, levamos ela para a direita, deixando de ser raiz. A nova raiz agora será o elemento X, que antes era filho à esquerda de W.

Figura 20 – Rotação simples à direita



A Figura 21 ilustra uma rotação à esquerda com uma BST. Foram empregadas as mesmas cores da Figura 19 para facilitar a identificação dos elementos. Antes da rotação, tínhamos a raiz desbalanceada (balanço -2). E o filho para a direita com balanceamento -1. Isso resulta em uma rotação simples à direita.

Figura 21 – Rotação simples à direita. Exemplo prático



FINALIZANDO

Ao longo desta etapa, estudamos a estrutura de árvore. Árvores são estruturas de dados não lineares e que são construídas com base em criar ramificações. Por esse motivo, são excelentes no emprego de sistemas de busca.

Uma árvore própria para busca é também chamada de árvore binária de busca (*Binary Serach Tree* – BST), e seus elementos são organizados de tal maneira que facilita a busca.

Todavia, uma árvore, quando apresenta uma inserção de dados não organizada, poderá apresentar ramos muito longos, gerando desbalanceamento na árvore. Podemos balancear a árvore com um algoritmo de balanceamento. Um bastante comum resulta no que chamamos de árvore de Adelson-Velskii e Landis (AVL), também chamado de árvore binária balaceada.

A seguir, temos uma tabela final comparativa do Big-O dos três tipos de árvores investigadas nesta etapa e suas manipulações.

	Acesso ao dado	Busca	Inserção	Remoção
Árvore Binária	O(n)	O(n)	O(n)	O(n)
BST	O(n)	O(n)	O(n)	O(n)
AVL	O(logn)	O(logn)	O(logn)	O(logn)

REFERÊNCIAS

ASCENCIO, A. F. G.; ARAÚJO, G. S. de. **Estruturas de Dados**: algoritmos, análise da complexidade e implementações em JAVA e C/C++. São Paulo: Pearson Prentice Halt 3, 2010.

BHARGAVA, A. Y. **Entendendo Algoritmos**. Novatec, 2017.

DROZDEK, A. **Estrutura de Dados e Algoritmos em C++**. 4. ed. Cengage Learning Brasil, 2018.

FERRARI, R. et al. **Estruturas de Dados com Jogos.** Elsevier, 2014.

KOFFMAN, E. B.; WOLFGANG, P. A. T. Objetos, Abstração, Estrutura de Dados e Projeto Usando C++. **Grupo GEN**, 2008.