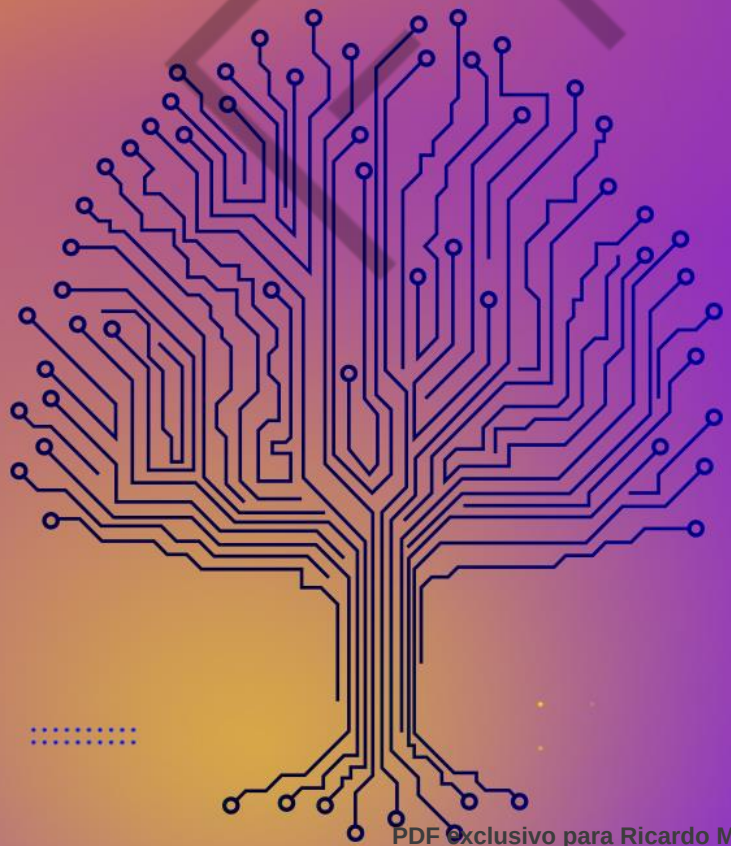
FIAP ON

CÓDIGOS DE ALTA PERFORMANCE

ÁRVORES BINÁRIAS

PATRICIA MAGNA



4

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE CÓDIGOS-FONTE

SUMÁRIO

4 ÁRVORES BINÁRIAS

4.1 Conceitos básicos e definições

Uma estrutura de dados denominada árvore é aquela em que o armazenamento de elementos deixa de ser unidimensional como um vetor e passa a ser bidimensional. Como mostra a árvore da Figura Esquema de uma Estrutura de Dados do tipo Árvore.

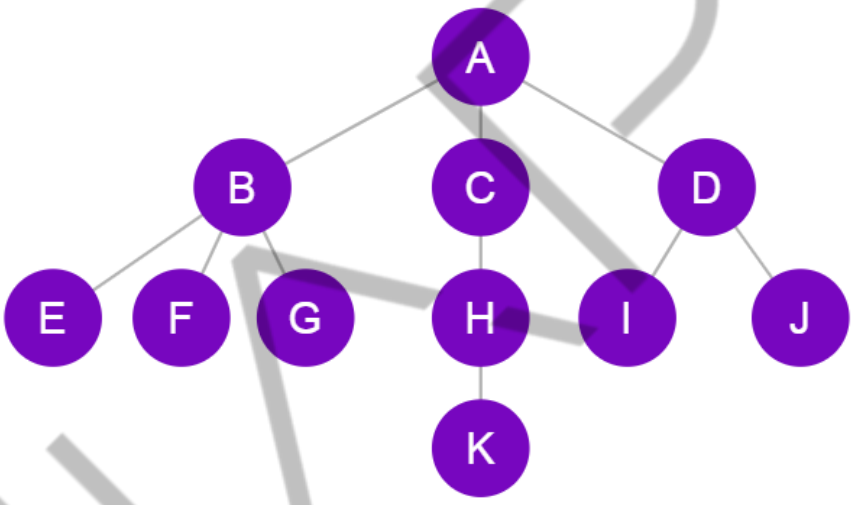


Figura 4.1– Esquema de uma Estrutura de Dados do tipo Árvore
Fonte: Elaborado pelo autor (2018)

Usando o formalismo adequado para definir, uma árvore é uma coleção finita de n ≥ 0 nós (elementos). Se n = 0, diz-se que a árvore é nula ou vazia, caso contrário, uma árvore apresenta as seguintes características:

- Existe um nó especial denominado raiz;
- Os demais nós são particionados em T_1, T_2, ... subárvores disjuntas, ou seja, nenhum nó poderá fazer parte de mais do que uma subárvore;
- Uma subárvore é uma coleção de nós que segue a mesma regra da definição de árvores.
- Uma vez que cada subárvore T_i é organizada como sendo uma árvore, tem-se assim uma definição recursiva.

Usando o exemplo da Figura Esquema de uma Estrutura de Dados do tipo Árvore para entender melhor a definição de árvore.

O nó ***A*** é nó raiz, do qual têm origem três subárvores:

- Subárvore cuja raiz é o nó ***B***:
    - A partir da raiz **B** a subárvore é particionada em três subárvores.
- Subárvore cuja raiz é o nó **C:**
    - Desta raiz é gerada apenas uma subárvore.
- Subárvore cuja raiz é o nó ***D***:
    - A partir da raiz **D** a subárvore é particionada em duas subárvores.

Então, podemos observar que, para definirmos uma árvore sempre reaplicamos a mesma definição a cada subárvore, ou seja, temos uma definição recursiva desse tipo de estrutura de dados.

A seguir, são apresentadas algumas considerações sobre como descrever uma árvore e o relacionamento entre os nós que a compõem.

As raízes das subárvores originadas de um nó são denominadas de nós filhos desse nó. Similarmente, esse é o nó pai das raízes das subárvores. No exemplo, ***A*** é o nó pai de ***B***, ***C***, e ***D.*** Os nós que são filhos de um mesmo nó são chamados de irmãos, nesse exemplo, ***B*** é irmão de ***C*** e ***D***. O nó ***I*** é irmão de ***J.***

Essas definições garantem um relacionamento intuitivo de descendência e precedência entre os nós. A Figura Nomenclatura e relação entre os nós de uma árvore apresenta mais um exemplo de árvore e a relação entre os nós.

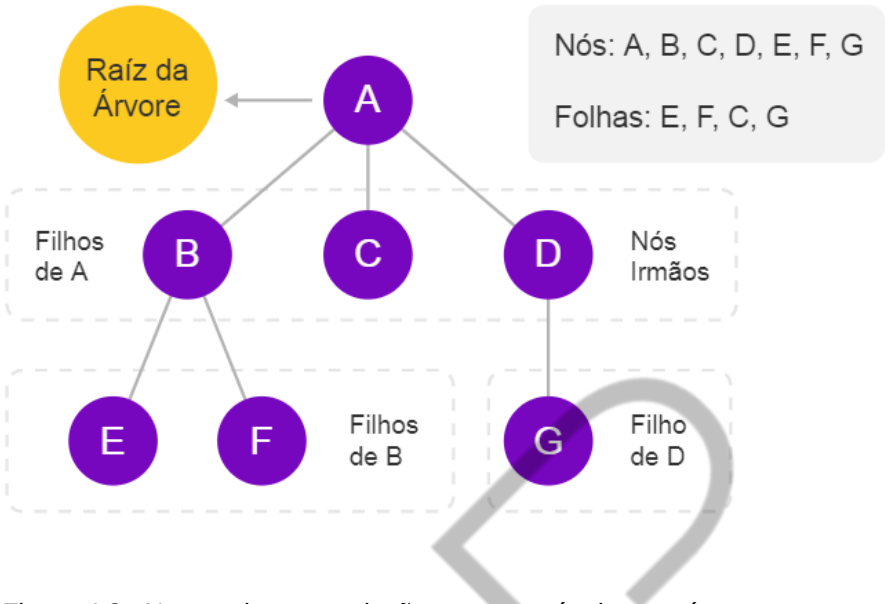


Figura 4.2– Nomenclatura e relação entre os nós de uma árvore
Fonte: Elaborado pelo autor (2018)

Para deixar bastante clara a especificação do que é uma árvore, a Figura Esquema que **NÃO** é uma Árvore: duas subárvores compartilhando mesmo nó **i** mostra um exemplo de esquema que NÃO é uma árvore, uma vez que a partição com origem no nó ***C*** e nó ***D*** não é disjunta.

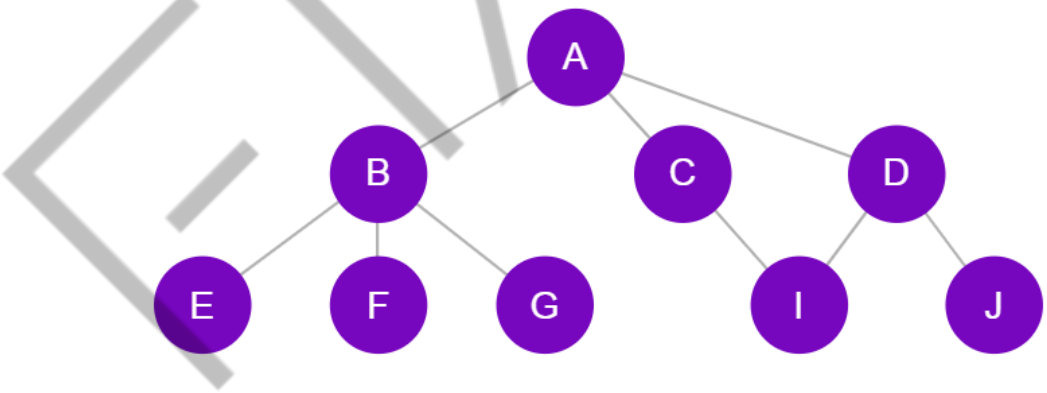


Figura 4.3– Esquema que **NÃO** é uma Árvore: 2 subárvores compartilhando mesmo nó **i**
Fonte: Elaborado pelo autor (2018)

Uma classificação importante para árvores é chamada de GRAU. O número de subárvores de um nó define seu grau. Quando o nó não possui nenhuma subárvore, ele é dito de grau zero e é denominado de nó folha ou nó terminal. Os demais nós são chamados de não terminais. O grau de uma árvore (chamado também de aridade) é definido como sendo igual ao máximo grau entre os seus nós.

No exemplo da Figura Esquema de uma Estrutura de Dados do tipo Árvore, tempos a seguinte análise do grau de cada nó:

- O grau dos nós ***A*** e ***B*** é 3;
- O grau do nó ***D*** é 2;
- O grau dos nós ***C*** e ***H*** é 1;
- ***E***, ***F***, ***G***, ***K***, ***I***, e ***J*** são nós folhas, ou seja, grau 0;

Pela análise, determinamos que o grau da árvore é, então, 3, pois é o maior grau de qualquer nó.

Outra definição é relacionada com o nível do nó. Por definição, diz-se que o nó raiz encontra-se no nível 1. Assim, se um nó estiver no nível n, os seus filhos estarão no nível n+1. A altura de uma árvore é definida como sendo o máximo dos níveis. A altura é zero quando a árvore é nula.

- Nível 1: ***A***;
- Nível 2: ***B C D***;
- Nível 3: ***E F G H I J***;
- Nível 4: ***K***;

Pela análise, a profundidade da árvore é 4, pois é o maior nível de um nó folha da árvore.

A Figura Definição de GRAU e PROFUNDIDADE de uma Árvore apresenta um exemplo de uma árvore e aplicação da classificação em relação ao grau e à profundidade.

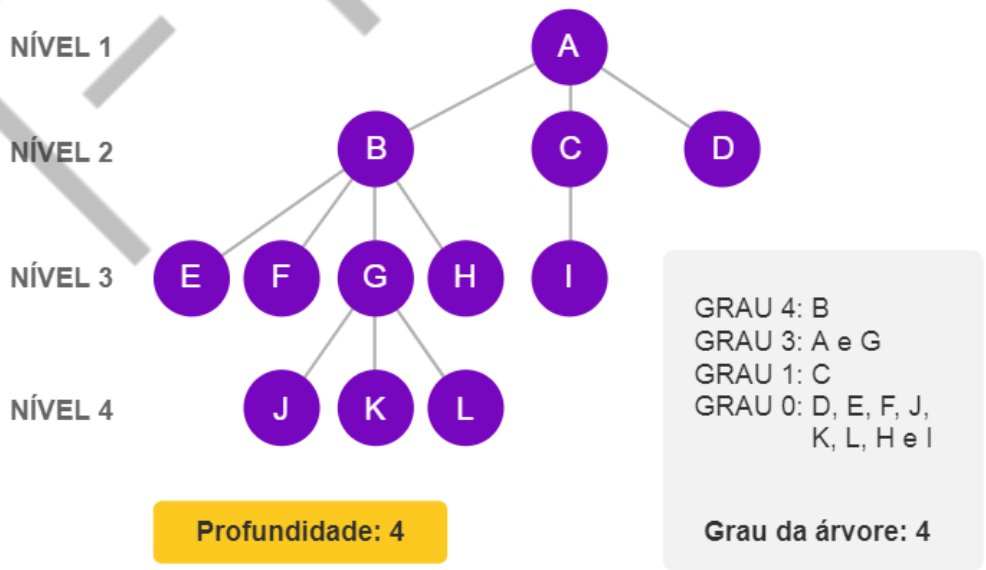


Figura 4.4– Definição de GRAU e PROFUNDIDADE de uma Árvore
Fonte: Elaborado pelo autor (2018)

Um tipo muito utilizado de árvore é a árvore binária, o próximo assunto a ser abordado.

4.2 Árvores Binárias

Uma árvore binária é uma árvore que pode ser nula, ou então ter as seguintes características:

- Existe um nó especial denominado raiz;
- Os demais nós são particionados em T1 e T2, estruturas disjuntas de árvores binárias;
- A estrutura T1 é denominada subárvore esquerda e T2 subárvore direita.

Resumindo, árvore binária é um caso especial em que nenhum nó possui mais do que dois filhos, ou seja, é uma árvore de grau 2.

Para árvores binárias existe um conceito de direção, ou seja, distingue-se uma subárvore direita e uma subárvore esquerda. No exemplo da Figura Exemplo de Árvore Binária, o nó **A** (nó raiz) tem uma subárvore esquerda (nó **B**) e uma subárvore direita (nó **C**). Já o nó **B** tem apenas um filho na subárvore esquerda (nó **D**). Finalmente, o nó **C** tem apenas um filho na subárvore esquerda (nó **F**).

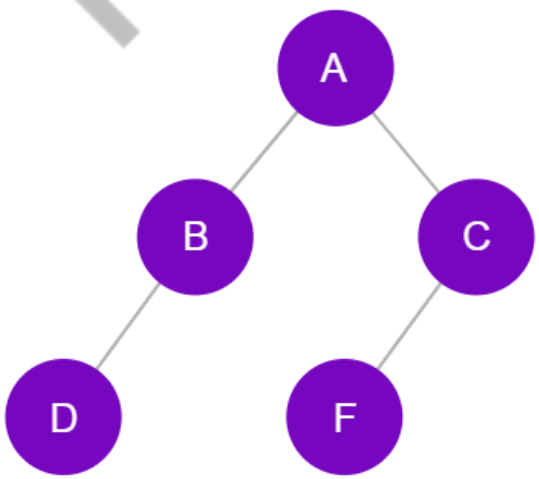


Figura 4.5– Exemplo de Árvore Binária
Fonte: Elaborado pelo autor (2018)

Uma árvore é dita estritamente binária se todos os nós, exceto os nós folhas, tiverem exatamente dois filhos. Observe que a árvore da Figura Exemplo de Árvore Binária não é estritamente binária. Um exemplo de árvore estritamente binária é visto na Figura Exemplo de Árvore Estritamente Binária.

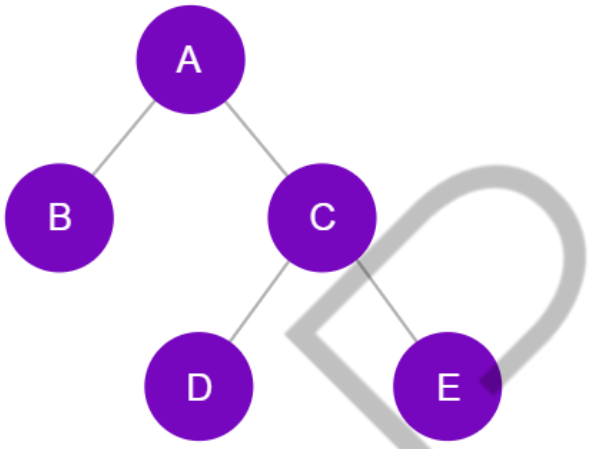


Figura 4.6– Exemplo de Árvore Estritamente Binária
Fonte: Elaborado pelo autor (2018)

Uma árvore binária completa de nível d é uma árvore estritamente binária na qual todos os nós folhas estão no nível d.

A Figura Exemplo de Árvore Binária Completa de Profundidade 3 apresenta um exemplo de árvore estritamente binária (todos os nós que não são folhas têm dois filhos), sendo também classificada como uma árvore binária completa de profundidade 3 (todos os nós folhas estão no nível 3).

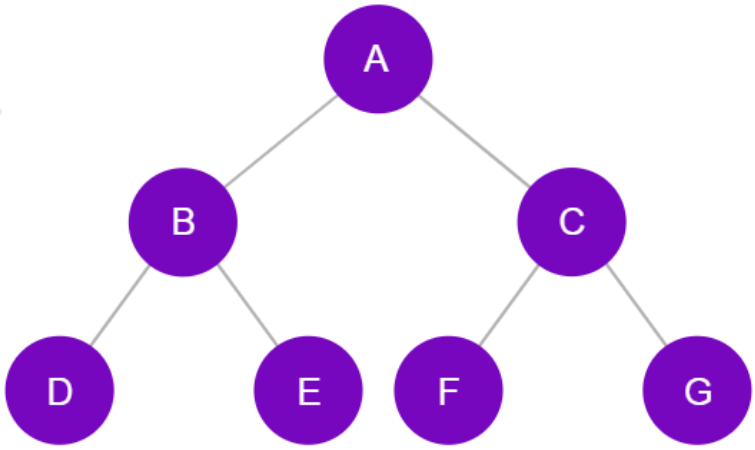


Figura 4.7– Exemplo de Árvore Binária Completa de Profundidade 3
Fonte: Elaborado pelo autor (2018)

4.2.1 IMPLEMENTAÇÃO DE ÁRVORES BINÁRIA EM JAVA

Como já descrevemos anteriormente, cada nó de uma árvore binária tem, no máximo dois nós filhos. Há a possibilidade de implementar árvores usando vetores, porém, faremos aqui apenas a implementação encadeada, uma vez que árvores são estruturas que, na prática, são normalmente implementadas com dados dinâmicos. Dessa forma, na especificação do nó, deve-se considerar que ele é bastante similar ao nó utilizado para implementar listas encadeadas, com a diferença que, agora, cada nó deve indicar os seus nós filhos. Aplicando especificamente a definição de árvore binária, um nó deve ser:

```
private static class ARVORE{
        public int dado;
        public ARVORE dir;
        public ARVORE esq;
}
```

Considerando o exemplo de árvore apresentado na Figura Árvore referenciada por ponteiro p para o nó raiz e ao lado esquema da implementação encadeada, pode-se observar que, para utilizar uma árvore, deve-se ter um ponteiro para o nó raiz, sendo que será através das ligações entre os nós que todos poderão ser obtidos. A ligação entre cada nó se dá por um ponteiro para o nó filho, seja da subárvore direita ou esquerda.

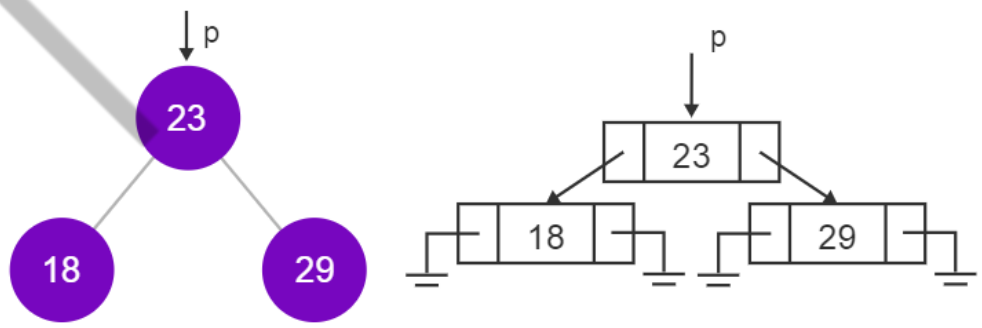


Figura 4.8–Árvore referenciada por ponteiro p para o nó raiz e ao lado esquema da implementação encadeada
Fonte: Elaborado pelo autor (2018)

Neste momento, não nos preocuparemos em como gerar essa árvore binária, ou seja, não faremos a inserção, nem a remoção dos nós na árvore binária.

Uma pergunta: você já imaginou como será um programa que possa percorrer (ter acesso a) todos os nós de uma árvore binária? A resposta nos leva a, finalmente, voltarmos ao assunto recursividade, pois será usando esse conceito que poderemos percorrer todos os nós de uma árvore binária.

Não existe uma única ordem para percorrer todos os nós de uma árvore, vamos começar uma forma de percurso chamada de percurso em ordem.

A função a seguir é usada para listar todos os elementos de uma árvore.

```
/*Percorre arvore apresentando elementos Em_Ordem*/

private static void mostra_em_ordem(ARVORE p) {
	if (p != null) {
		mostra_em_ordem(p.esq);
		System.out.println(" "+ p.dado);
		mostra_em_ordem(p.dir);
		}
	}
```

Como será a tela de saída quando usarmos essa função para percorrer a árvore da Figura Árvore Binária para ser percorrida?

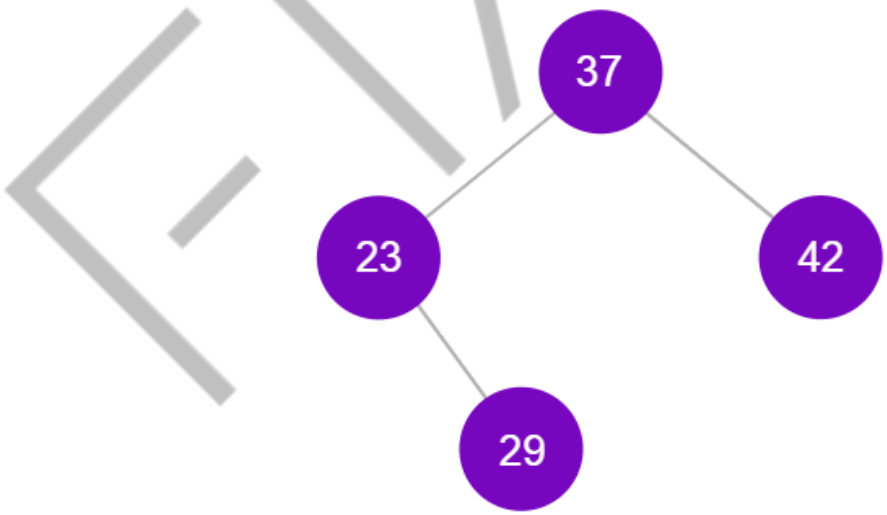


Figura 4.9–Árvore Binária para ser percorrida
Fonte: Elaborado pelo autor (2018)

Vamos descrever a sequência de chamadas e retornos da função recursiva mostra_em_ordem() de duas formas. A primeira será com explicação no texto e, em seguida, figura com a representação gráfica do passo a passo.

Você deve sempre iniciar a primeira ativação com a única referência que temos para uma árvore, ou seja, o ponteiro para o nó raiz. Assim, a sequência de ativações é:

Ativação 1 => p ≠ null => p.dado =37 => p.esq ≠ null =>mostra_em_ordem (p.esq)

Ativação 2 => p ≠ null => p.dado =23 => p.esq = null =>**escreve p.dado 23**

=> p.dir ≠ null =>mostra_em_ordem (p.dir)

Ativação 3 => p ≠ null => p.dado =29 => p.esq = null =>**escreve p.dado 29**

=> p.dir = null => encerra ativação 3 => volta Ativação 2 depois da chamada mostra_em_ordem (p.dir).

Ativação 2 => encerra ativação 2 => volta Ativação 1depois da chamada mostra_em_ordem (p.esq).

Ativação 1 =>**escreve p.dado 37**=> p.dir ≠ null =>mostra_em_ordem (p.dir).

Ativação 2 => p ≠ null => p.dado =42 => p.esq = null => **escreve p.dado 42**

=> p.dir = null => encerra ativação 2 => volta Ativação 1 depois da chamada mostra_em_ordem (p.dir)

Ativação 1=> encerra Ativação 1

Portanto, a tela de saída é apresentada na seguinte sequência: **23 29 37 42.**

Vamos, agora, descrever a recursividade de forma gráfica usando a Figura Sequência de ativações para percurso EM ORDEM de Árvore Binária, que apresenta a sequência de etapas realizadas para o percurso da árvore. É interessante observar que cada ativação recebe um ponteiro para um determinado nó, sendo que, na figura é destacada uma cor para cada ativação.

Iniciando a ativação 1 com o ponteiro p apontando para o nó raiz da árvore (em vermelho), é verificado que p≠ null e que p.esq ≠ null (ponteiro esq aponta para o nó com dado 23), assim é chamada mais uma vez a função mostra_em_ordem() passando como parâmetro p.esq (ponteiro em verde na figura). Esses comandos descritos estão em negrito para demonstrar que já foram executados no momento em que estão sendo apresentados.

No quadro seguinte da figura, inicia a ativação 2, com o ponteiro p em verde, que aponta para o nó com dado 23. Observe também, nesse quadro da figura, que a ativação 1 não terminou e apenas foi interrompida para iniciar a chamada da ativação 2, isso é representado fazendo com que a ativação 2 fique “sobre” a ativação 1. Como o nó com dado 23 não tem filho à esquerda, o comando seguinte faz com que o dado seja apresentado na tela de saída e, em seguida, é feita a chamada da função mostra_em_ordem() passando agora p.dir (em azul).

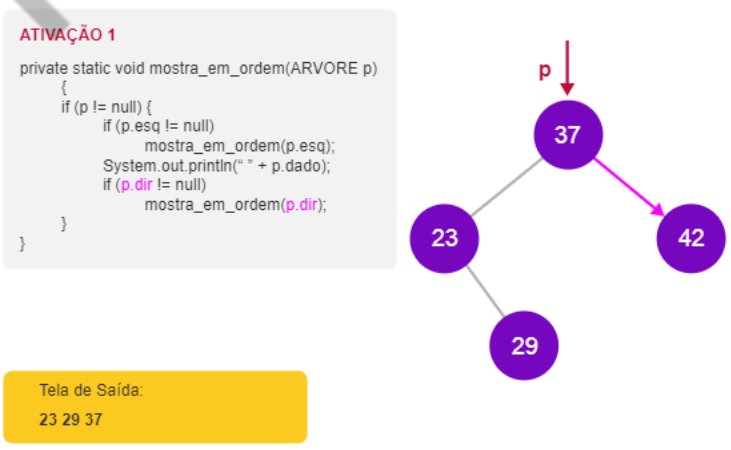
A ativação 3 inicia no quadro seguinte, com ponteiro p (em azul) apontando para o nó com dado 29, que é um nó que não tem filho à esquerda, assim o dado 29 é apresentado na tela de saída e como também não tem filho à direita. A ativação 3 é concluída, retornando para a ativação 2 na posição seguinte à chamada da função que gerou a ativação 3.

Assim, no quadro seguinte, a ativação 2 (ponteiro p em verde) é finalizada também, retornando para a ativação 1.

No passo seguinte, a ativação 1 executa o comando de apresentação do dado 37, que é o nó apontado por p em vermelho. Como o nó dessa ativação tem um filho à direita, mais uma vez, a função mostra_em_ordem() é chamada criando mais uma ativação com argumento o ponteiro p em verde.

No quadro seguinte, é apresentado que, como o nó com dado 42 é folha (não tem filhos), a ativação 2 apenas apresenta na tela de saída o dado e é encerrada.

Voltando à ativação 1 apenas para encerrar o percurso para a apresentação de toda a árvore.



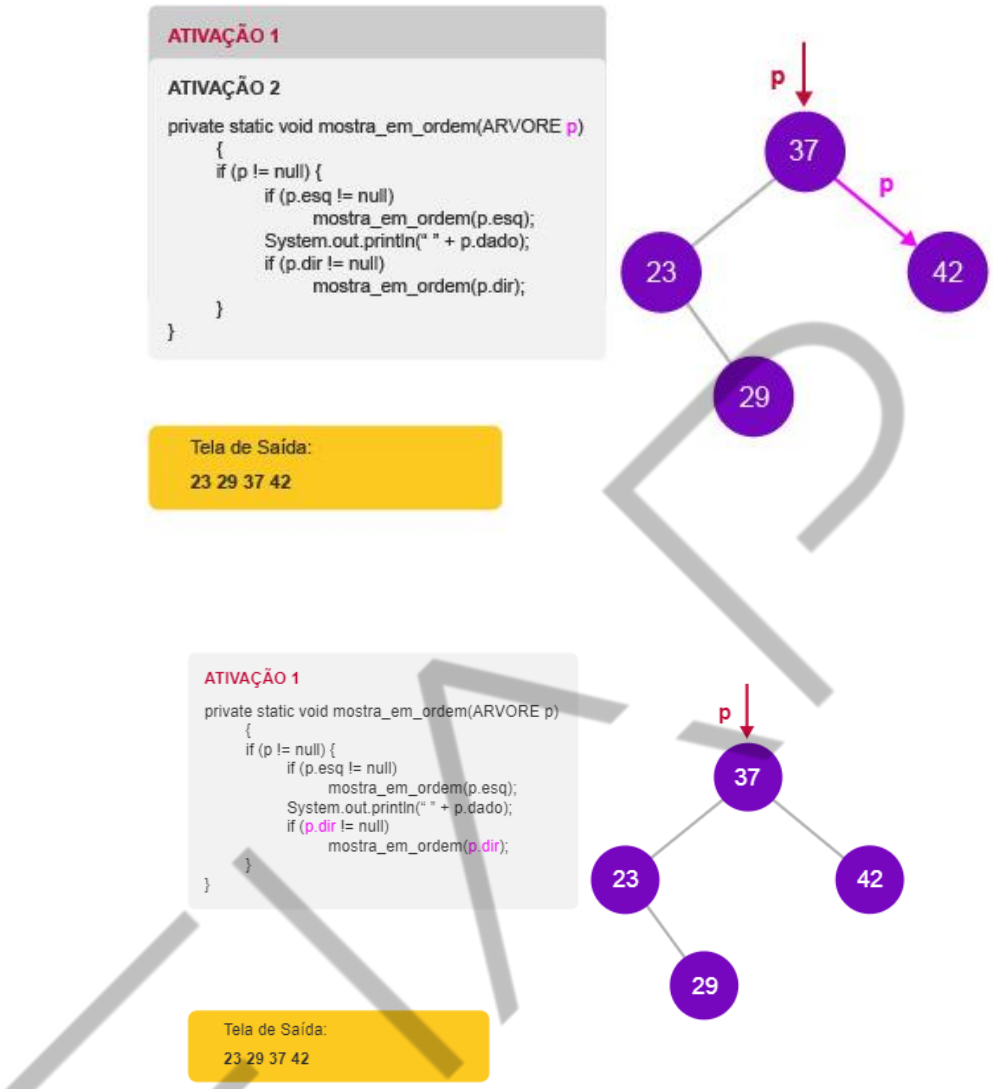


Figura 4.10–Sequência de ativações para percurso EM ORDEM de Árvore Binária
Fonte: Elaborado pelo autor (2018)

Existem outras formas de apresentar todos os nós de uma árvore binária, mudando apenas o momento em que o nó terá seu conteúdo acessado. Essas funções são chamadas de mostra_pos_ordem e mostra_pre_ordem e são apresentadas a seguir.

```
/*Percorre arvore apresentando elementos Pre_Ordem*/

private static void mostra_pre_ordem(ARVORE p) {
	if (p != null) {
		System.out.println(" "+ p.dado);
		mostra_pre_ordem(p.esq);
		mostra_pre_ordem(p.dir);
		}
	}
```

```
/*Percorre arvore apresentando elementos Pos_Ordem*/

private static void mostra_pos_ordem(ARVORE p) {
	if (p != null) {
		mostra_pos_ordem(p.esq);
		mostra_pos_ordem(p.dir);
		System.out.println(" "+ p.dado);
		}
	}
```

Há diferentes tipos de árvores, cada uma com aplicação específica. Alguns exemplos de árvores e de suas aplicações são:

- Problemas de busca de dados armazenados na memória principal do computador: árvore binária de busca, árvores (quase) balanceadas como AVL etc.
- Problemas de busca de dados armazenados na memória secundária principal do computador (disco rígico): árvores-B.
- Aplicações em Inteligência Artificial: árvores que representam o espaço de soluções, como jogo de xadrez, resolução de problemas etc.
- No processamento de cadeias de caracteres: árvore de sufixos.
- Na gramática formal: árvore de análise sintática.

Agora, vamos avançar nos estudos, entendendo e implementando as árvores binárias. Esse tipo de árvore é muito utilizado nas mais diversas aplicações porque, quando ordenadas, permitem realizar pesquisas, inclusões e exclusões de dados em sua estrutura de forma extremamente eficiente, ou seja, com alto desempenho.

Recordando, uma árvore é dita binária se cada nó tiver no máximo dois nós filhos, ou seja, árvore binária é uma árvore de grau 2.

4.3 Árvores Binárias de Busca (ABB)

4.3.1 Definição e Motivação de Uso de ABB

Suponha como exemplo um sistema de votação pela internet no qual cada pessoa possa votar apenas uma vez. Para garantir essa condição, é usado um sistema que armazena todos os números de CPF das pessoas que já votaram. A cada novo voto, esse sistema é consultado para verificar se ele deve ou não computar ser computado.

Se os números de CPF forem armazenados em uma estrutura linear (como uma lista linear) o tempo necessário para realizar as comparações na busca do CPF será proporcional à quantidade de votos já cadastrados (isto é, CPF de votos já computados).

Solução: o uso de um tipo especial de árvore binária chamada de Árvore de Busca Binária (ABB). Então, vamos definir o que é uma ABB.

Supondo uma árvore binária com nó raiz que armazena um elemento V é chamada de Árvore de Busca Binária (ABB) se:

- Todo elemento armazenado na subárvore esquerda é menor que V;
- Nenhum elemento armazenado na subárvore direita é menor que V;
- As subárvores direita e esquerda são também ABB.

Na Figura Exemplo de Árvore Binária de Busca (ABB), note que a raiz da árvore tem valor 20, e como filho à direita, o nó com dado 45 (maior que a raiz) e, à esquerda, o nó com dado 15 (menor que a raiz). Observando, agora, apenas a subárvore à esquerda, a raiz desta subárvore tem valor 15, o nó com dado menor (10) está à esquerda e com valor maior (18) está à direita. Assim, cada subárvore de uma ABB segue sempre a mesma regra de formação.

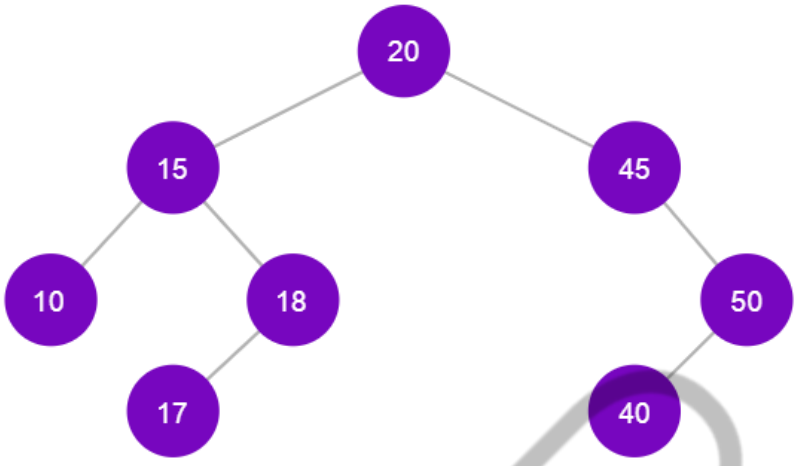


Figura 4.11 – Exemplo de Árvore Binária de Busca (ABB)
Fonte: Elaborado pelo autor (2018)

Voltando, agora, ao exemplo da votação por telefone, para usar a ABB como solução, cada número de CPF de um voto válido (e que já foi computado pelo sistema) é armazenado em um nó da ABB. Em uma nova tentativa de voto, deve-se procurar na árvore de busca binária o número de CPF.

A Figura Árvore Binária de Busca (ABB) que armazena CPF de votos supõe a existência de votos já computados e os números de CPF de votos validados já armazenados em uma ABB. Observe que valores de CPF menores que o nó raiz (CPF 123.456.789.0) estão na subárvore à esquerda e o número de CPF que é maior já está à direita.

O voto que se deseja computar tem como CPF 217.654.987.1. O processo de busca na ABB inicia pelo nó raiz, como os valores de CPF são diferentes, verifica-se que o CPF procurado é maior do que o valor do nó raiz. Dessa forma, desce para o nó filho à direita verificando que também são diferentes, mas agora o CPF procurado é menor. Como a subárvore à esquerda encontra-se vazia, conclui-se que não há registro do CPF procurado na ABB e o voto pode ser computado. Além disso, o CPF é inserido como nó filho à esquerda do nó com valor de CPF 221.780.459.1.

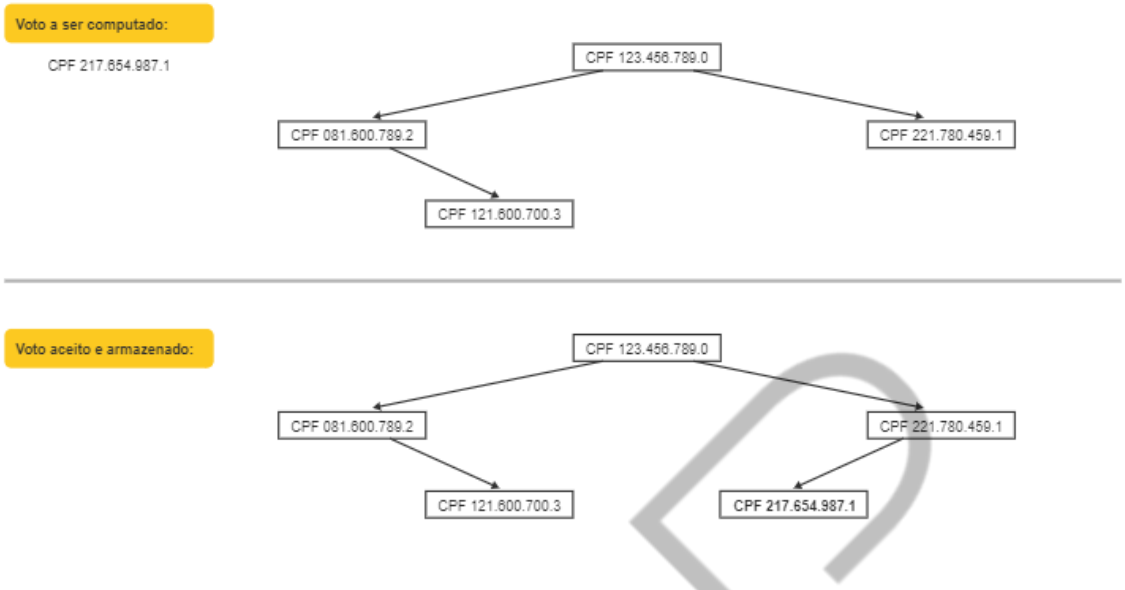


Figura 4.12 – Árvore Binária de Busca (ABB) que armazena CPF de votos
Fonte: Elaborado pelo autor (2018)

Supondo que 1 milhão de CPF de votos já registrados existam, quantas comparações seriam necessárias para determinar se o CPF já votou?

Para responder a essa pergunta, é necessário fazer uma suposição do estado em que se encontra essa ABB. Supondo que seja uma ABB com chaves uniformemente distribuídas, isto é, se a árvore está completa (todos os nós do tipo folha estão no mesmo nível) ou quase completa (todos os nós folha estão em níveis com diferença de no máximo 1). Ainda é preciso fazer algumas outras perguntas.

Quantos nós cabem em uma arvore binária completa? Vamos analisar o quadro a seguir.

Nível	Número de elementos em cada nível	Total de elementos da árvore
1	1	1
2	2	3
3	4	7
4	8	15
5	16	31
..	..	..
10	512	1.023 (2^{10}-1)
..	..	..
20	524.288	1.048.575
..	..	..

n	2^{n-1}	2^n-1

Quadro 4.1 – Especificação de números de nós por nível em uma árvore binária
Fonte: Elaborado pelo autor (2018)

Assim, se for usada uma ABB com 1 milhão de nós e ela estiver completa ou quase completa, ou seja, com chave uniformemente distribuída, seriam necessárias, no máximo 20 comparações para garantir que o número não tenha sido registrado.

Se esse número de elementos estivesse em uma lista linear, o número de comparações poderia chegar a 1 milhão. Portanto, fica claro que, para grande volume de dados, o uso de árvores para o armazenamento e recuperação de informações é extremamente mais eficiente do que o uso de estruturas unidimensionais, como listas lineares.

Agora nos falta saber como implementar esse tipo de estrutura tão necessária.

4.3.2 Implementação de ABB em JAVA

Lembrando que para implementar uma ABB é necessário definir a estrutura de cada nó, que precisa ser composto pelas seguintes informações:

- Valor do elemento (dado);
- Ponteiro para nó filho à esquerda (esq);
- Ponteiro para nó filho à direita (dir).

```
private static class ARVORE{
        public int dado;
        public ARVORE dir;
        public ARVORE esq;
        }
```

Código Fonte 4.1 – Especificação do um nó em uma ABB
Fonte: Elaborado pelo autor (2018)

Para implementar uma árvore, assim como nas listas encadeadas, uma árvore deve ser inicializada por meio da função `init()`. Toda árvore deve ser iniciada como sendo uma árvore nula.

```
public static ARVORE init()  {
        return null;
```

```
}
```

Código Fonte 4.2 – Função init () que inicia uma ABB vazia
Fonte: Elaborado pelo autor (2018)

A inserção de um novo nó sempre é feita de forma que o nó seja um nó folha. Para que seja uma árvore ABB é ainda necessário seguir as premissas de uma ABB, ou seja, encontrar a posição correta para esse novo nó, de forma que a subárvore esquerda tenha sempre elementos menores que o nó raiz e a subárvore direita possua apenas elementos maiores.

Para inserir um novo elemento em uma ABB são possíveis três casos:

- A ABB está vazia: neste caso, o elemento é inserido como nó raiz e conclui a ativação da função.
- A informação a ser inserida é menor que a informação contida no nó raiz: o elemento deve ser inserido na subárvore esquerda da raiz, a função de inserção é novamente aplicada para a subárvore à esquerda.
- A informação a ser inserida é maior ou igual à informação do nó raiz: insere elemento na subárvore direita da raiz a função de inserção é novamente aplicada para a subárvore à direita.

Supondo, a partir de agora, uma árvore de elementos inteiros e o tipo de dados ARVORE definido como ponteiro para um nó, a função que insere um nó em uma ABB desse tipo é apresentada na Figura Função que insere um nó com valor específico em uma ABB.

```
private static ARVORE inserir(ARVORE p, int info) {
      // insere elemento em uma ABB
      if (p == null) {
            p=new ARVORE();
            p.dado = info;
            p.esq = null;
            p.dir = null;
      }
      else if (info < p.dado)
                  p.esq= inserir (p.esq, info);
            else
                  p.dir=inserir(p.dir, info);
      return p;
      }
```

Código Fonte 4.3 – Função que insere um nó com valor específico em uma ABB
Fonte: Elaborado pelo autor (2018)

Para implementar a função que fará a remoção de um nó informando o valor do dado que o nó armazena em uma ABB, primeiro deve ser procurado o nó que tenha o valor do dado escolhido. Assim, seguem alguns possíveis estados em que se encontra a árvore e alguns casos:

- A ABB está vazia: neste caso não existe a chave procurada e conclui a função.
- A chave é menor que a informação da raiz: passa a procurar na subárvore esquerda, criando mais uma ativação da função.
- A chave é maior que a informação da raiz: passa a procurar na subárvore direita, criando mais uma ativação da função.
- A informação é igual à chave: neste caso existem quatro possíveis situações antes de concluir a função:
  - o nó é folha, ou seja, não tem filhos nem à direita e nem à esquerda, assim basta fazer o ponteiro que aponta para o nó receber null e o nó será retirado da árvore ().
  - o nó não tem subárvore esquerda, assim, o ponteiro que aponta para o nó passa a apontar para a subárvore direita;
  - o nó não tem subárvore direita, assim, o ponteiro que aponta para o nó passa a apontar para a subárvore esquerda;
  - se o nó possui as subárvores direita e esquerda não se pode eliminá-lo de forma trivial. A forma adotada na função apresentada no Código Fonte Função que remove um nó com valor específico em uma ABB é encontrar o nó com menor elemento da subárvore direita, pois, assim, é feito um rearranjo dos nós, de forma, que o nó encontrado como o menor valor será “colocado” no lugar do nó que deve ser eliminado, e depois o nó folha.

```
private static ARVORE remove_valor (ARVORE p, int info) {
    if (p!=null){
          if(info == p.dado){
                if (p.esq == null && p.dir==null)
//nó a ser removido é nó folha
                      return null;
                if (p.esq==null)
                      return  p.dir;
```

```
  /*se não há sub-árvore esquerda o ponteiro passa apontar para a sub-
árvore direita*/
                  else{
                        if (p.dir==null)
  /*se não há sub-árvore direita o ponteiro passa apontar para a sub-
árvore esquerda */
                              return p.esq;
                        else {
  /*o nó a ser retirado possui sub-árvore esquerda e direita, então o
nó que será retirado deve-se encontrar o menor valor na sub-árvore à
direita */
                              ARVORE aux, ref;
                              ref = p.dir;
                              aux = p.dir;
                              while (aux.esq != null)
                                    aux = aux.esq;
                              aux.esq = p.esq;
                              return ref;
                              }
                        }
                  }
            else{       //procura dado a ser removido na ABB
                  if(info < p.dado)
                        p.esq = remove_valor(p.esq,info);
                  else
                        p.dir = remove_valor(p.dir,info);
                  }
            }
      return p;
  }
```

Código Fonte 4.4 – Função que remove um nó com valor específico em uma ABB
Fonte: Elaborado pelo autor (2018)

Para compreender melhor essa função, suponha a seguinte ABB e a remoção de um nó com valor 47, conforme especificado na Figura Remoção de um nó folha em uma ABB. Vamos, então, fazer uma descrição passo a passo. Inicialmente, a função remove_valor() recebe o ponteiro para o nó raiz da árvore (com dado 20), em seguida:

1. Como não é o nó com dado que deve ser removido e 47 é maior do que 20, a função é chamada novamente, passando agora o ponteiro para o nó 45 (descendo para a subárvore direita);
2. Verifica-se que ainda não é o valor procurado e, como 47 é maior do que 45, chama mais uma vez a função, passando o ponteiro para o nó 50 (descendo para a subárvore direita);
3. Como não é o nó com valor que deve ser removido, e 47 é menor do que 50, então chama a função remove_valor passando o ponteiro para o nó com dado 47 que está na subárvore esquerda;

4. A Figura Remoção de um nó folha em uma ABB representa que essa ativação está recebendo o ponteiro `p` em destaque. Verifica-se que esse nó tem o dado que deve ser removido, como o nó 47 não tem filhos nem à direita, nem à esquerda, basta retornar valor `null`, fazendo com que o ponteiro p receba `null`, desta forma o nó com dado 47 é "desligado" da árvore. Essa ativação da função é então finalizada, concluindo também todas as ativações.

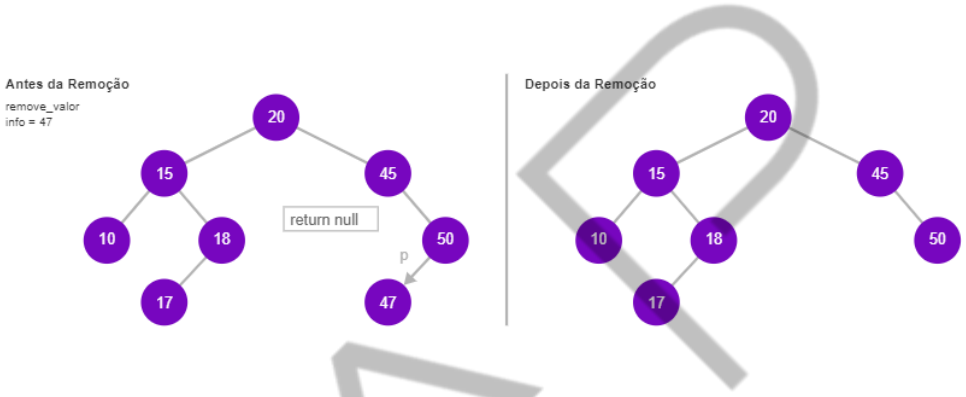


Figura 4.13 – Remoção de um nó folha em uma ABB
Fonte: Elaborado pelo autor (2018)

Vamos a mais uma remoção, desta vez, o nó que deve ser retirado da árvore é o nó com dado 45 (Figura Remoção de um nó com apenas uma subárvore em uma ABB). Sempre, no início, a função remove_valor() é chamada passando o ponteiro para o nó raiz (com dado 20). Como o valor 45 é maior do que 20, é feita a chamada da função, agora, passando o ponteiro para o nó raiz da subárvore direita, que é a ativação com ponteiro `p` apresentada na figura. Como o dado desse nó é o que deve ser retirado, é verificado que existe apenas uma subárvore a partir desse nó, neste caso, a subárvore direita, assim basta retornar o ponteiro da subárvore direita do nó 45 que o ponteiro p passa apontar para o nó com dado 50. Assim, o nó com dado 45 é retirado da ABB.

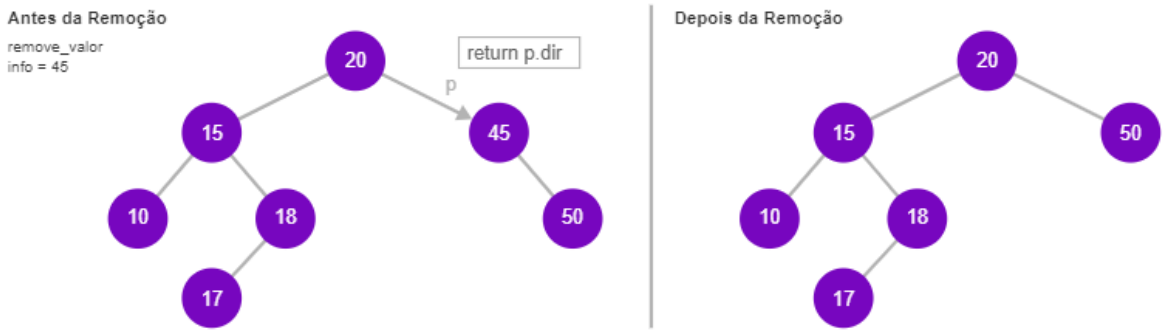


Figura 4.14 – Remoção de um nó com apenas uma subárvore em uma ABB
Fonte: Elaborado pelo autor (2018)

Por último, vamos apresentar (usando a Figura Remoção de um nó com duas subárvores em uma ABB) a remoção de um nó quando esse tem dois filhos. Assim como nos exemplos, a função remove_valor() inicia com o nó raiz da árvore com dado 20; como o valor do nó a ser eliminado é 15, que é menor do que 20, é feita a chamada da função passando o ponteiro para o nó raiz da subárvore esquerda. Essa ativação é apresentada na figura com o ponteiro `p`. O nó com valor 15 tem tanto filho à esquerda como à direita, assim, a função precisa de dois ponteiros auxiliares para realizar a remoção, `aux e ref`, ambos são posicionados para apontar para o nó à direita do nó a ser removido (nesse caso, o nó com valor 15).

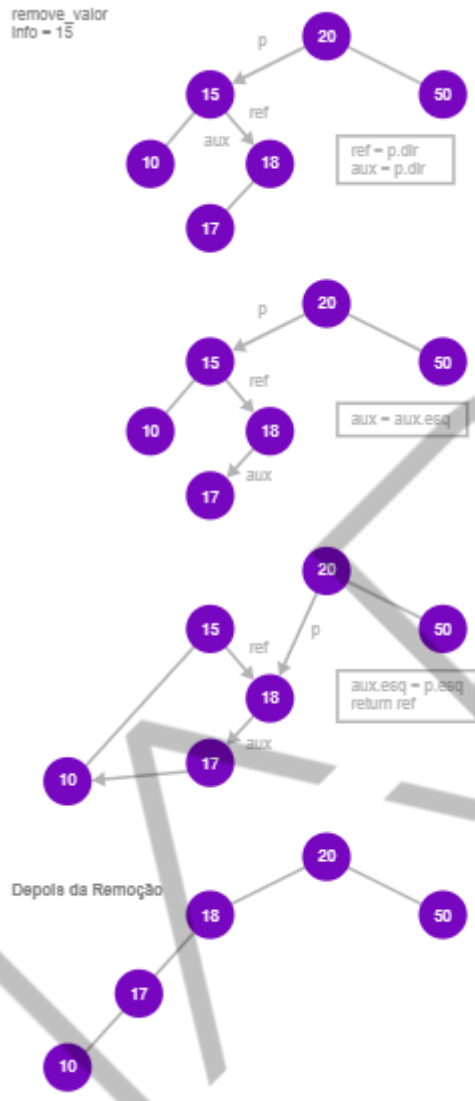


Figura 4.15 – Remoção de um nó com duas subárvores em uma ABB
Fonte: Elaborado pelo autor (2018)

O ponteiro `ref` permanece na mesma posição, já o ponteiro `aux` é movido até o nó que não tem mais filho à esquerda (enquanto `aux.esq` for diferente de `null`), nesse caso, `aux` termina a repetição apontando para o nó com dado 17. É com essa configuração dos ponteiros `p`, `ref` e `aux` que a remoção de fato acontece. Primeiro, o ponteiro esquerdo do nó apontado por `aux` (nó 17), que antes apontava para `null`, passa apontar para o nó filho à esquerda do nó que será removido (nó 15, apontado por `p`), e em seguida, a ativação termina retornando o ponteiro `ref`. Dessa forma, o nó apontado por `ref` (nó 18) será o nó apontado pelo ponteiro `p`. Com essas alterações, o nó com dado 15 é retirado da estrutura da ABB.

Observe que após a remoção do nó 17, a ABB fica com mais níveis na subárvore esquerda do que na subárvore direita. A motivação do uso de ABB para armazenamento de dados é que a recuperação de informação é feita de forma muito mais eficiente do que se armazenarmos dados em uma lista linear. Porém, é fácil perceber que se quisermos ter acesso ao dado com valor 10, vamos ter que fazer quatro comparações em uma árvore com cinco elementos.

Portanto, cada vez que inserimos ou removemos nós de uma ABB pode ser que ela deixe de estar na configuração de árvore completa ou quase completa diminuindo a eficiência da utilização de uma ABB.

Calma, esse problema tem solução, vamos entendê-la conhecendo um novo tipo de árvore chamada de AVL, que estudaremos a seguir.

4.4 Árvores AVL

4.4.1 Definição e Motivação

A principal utilização de árvores tem como objetivo realizar a consulta (procura) a uma chave de forma mais eficiente possível, ou seja, em um tempo menor. Em uma aplicação “estática”, como no exemplo da votação por telefone, o uso de uma ABB já possibilita uma boa eficiência dos acessos às chaves. Porém, em aplicações “dinâmicas”, ou seja, aplicações em que são realizadas operações de inserções e remoções frequentes, é muito grande a probabilidade de se reduzir a eficiência com que as operações são realizadas nessas árvores.

Observando a árvore apresentada na Figura Exemplo de uma árvore binária com grande diferença de níveis em seus ramos, pode-se notar que o fato de essa árvore não estar próxima ao que se considera uma árvore completa faz com que a busca pela chave 12 seja bastante demorada, sendo necessárias seis comparações para ser encontrada, já que está no nível 6.

Para que as operações sobre as árvores sejam realizadas de forma ótima, o que se pode fazer é, periodicamente, realizar uma operação que deixe a árvore completa novamente. Essa operação é denominada balanceamento de árvore. Uma

árvore binária é dita balanceada se, para cada nó, as alturas de suas subárvores diferem, no máximo, de 1.

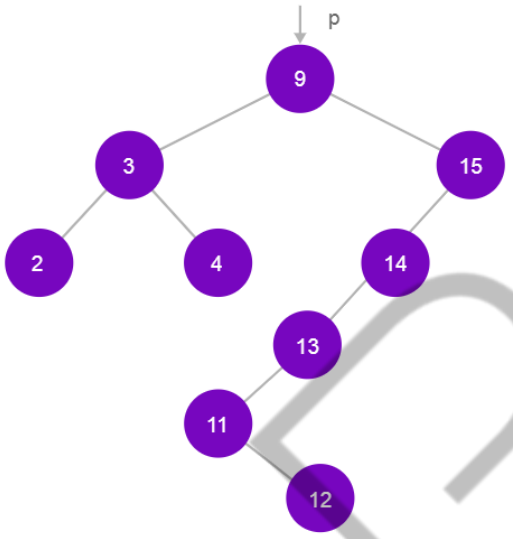


Figura 4.16 – Exemplo de uma árvore binária com grande diferença de níveis em seus ramos
Fonte: Elaborado pelo autor (2018)

A árvore apresentada anteriormente estaria balanceada se estivesse como mostra Figura Exemplo de uma árvore binária balanceada. Note que a árvore tem, agora, apenas quatro níveis e, esse seria, então, o número máximo de comparações que são efetuadas para se consultar uma chave na árvore balanceada.

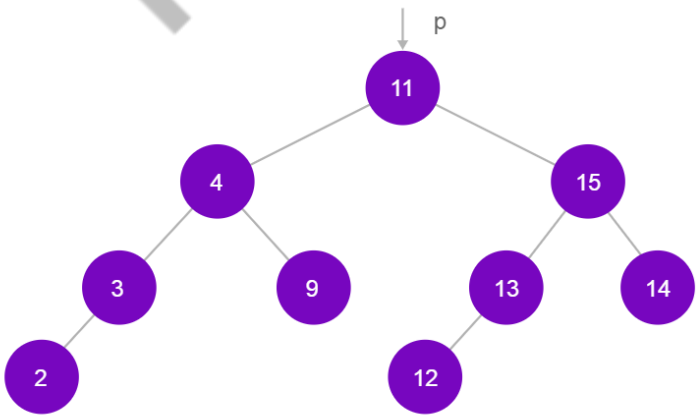


Figura 4.17 – Exemplo de uma árvore binária balanceada
Fonte: Elaborado pelo autor (2018)

A Figura Configurações de Árvores Perfeitamente Balanceadas com menor altura de 1 até 7 nós apresenta configurações de uma árvore perfeitamente balanceada em função do número de nós. É interessante observar que o número de

níveis desse tipo de árvore é determinado pelo número de nós, conforme o Quadro Especificação de números de nós em função da altura de uma árvore binária.

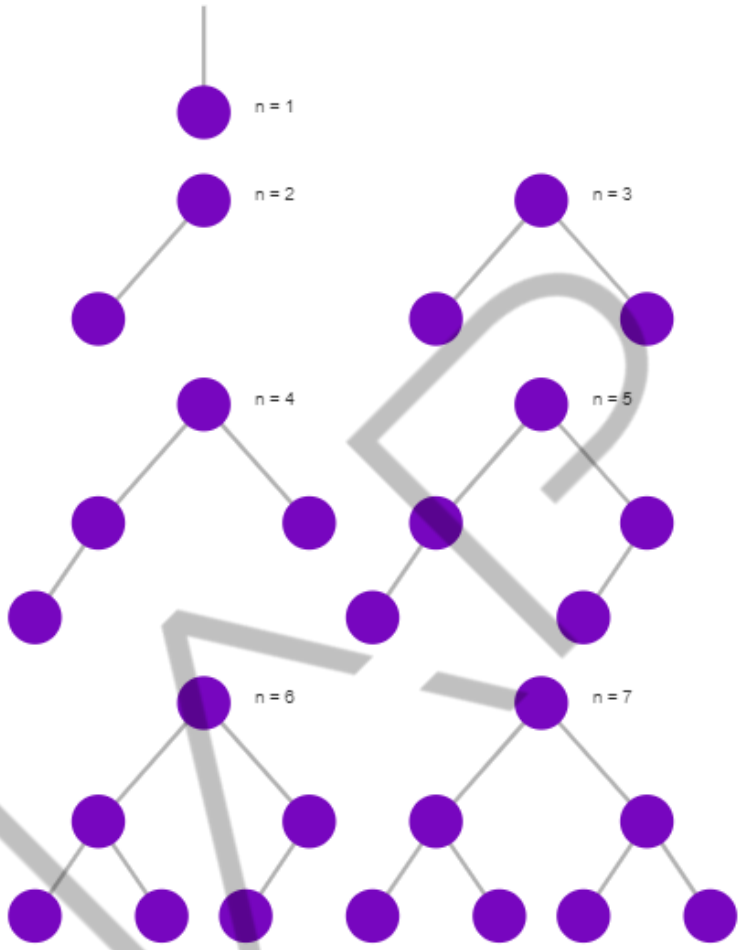


Figura 4.18 – Configurações de Árvores Perfeitamente Balanceadas com menor altura de 1 até 7 nós
Fonte: Elaborado pelo autor (2018)

Altura	Possível número de elementos na árvore
1	1
2	de 2 a 3
3	de 4 até 7
4	de 8 até 15
5	de 16 até 32
..	..
10	de 512 até 1023

Quadro 4.2 – Especificação de números de nós em função da altura de uma árvore binária
Fonte: Elaborado pelo autor (2018)

Existe uma forma direta para saber quantos níveis uma árvore balanceada terá em função do número de nós, que é:

$$altura = log_2 (Número\ de\ nó) + 1$$

Claro que, da altura, deve apenas obter a parte inteira resultante do cálculo. Analisando a Figura Configurações de Árvores Perfeitamente Balanceadas com menor altura de 1 até 7 nós, se quisermos saber qual a altura de uma árvore binária balanceada com seis nós, pelo cálculo, teríamos $\log_2 6 = 2$ (aproximadamente 2.6), com a parte inteira 2,6 que é 2 e somado a 1, obtemos que o número de níveis para ter seis nós em uma árvore balanceada é 3. Supondo que desejemos a altura que uma árvore binária balanceada terá se houver 50 nós, basta lembrar que parte inteira de $\log_2 50$ é 5 e somar 1. Portanto, com 50 elementos teríamos uma árvore com seis níveis e, assim, com apenas seis comparações, poderíamos ter acesso a qualquer elemento da árvore binária de busca.

Pelas vantagens que descrevemos aqui, fica clara a necessidade de realizarmos a operação de balanceamento da árvore. Obviamente, também essa operação deve ser realizada de forma eficiente, pois, caso contrário, não há vantagem em utilizá-la. Eis um desafio!

Uma árvore AVL é uma árvore binária de busca (ABB) construída de tal modo que a altura de sua subárvore direita difere da altura da subárvore esquerda de no máximo 1. Algoritmos de balanceamento de árvore são chamados algoritmos AVL e as árvores geradas por esses algoritmos são denominadas de árvores AVL. Esse tipo de árvore surgiu em 1962 com matemáticos russos G.M. Adelson-Velskki e E.M.Landis que sugeriram uma definição para "*near balance*" e descreveram procedimentos para inserção e eliminação de nós nessas árvores.

4.4.2 Construindo Programa para Implementar uma Árvore AVL

Para inserir um novo nó em uma árvore balanceada é importante saber qual é o seu estado. A Figura Exemplo de uma ABB balanceada mostra uma ABB balanceada que usaremos para todas as nossas explicações dos algoritmos desenvolvidos para implementar uma árvore AVL.

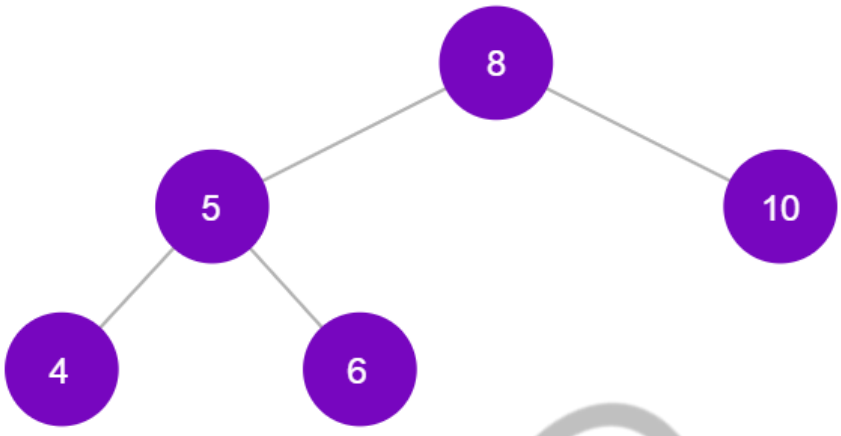


Figura 4.19 – Exemplo de uma ABB balanceada
Fonte: Elaborado pelo autor (2018)

Supondo que a inserção seja de um nó com dado 3, que, por ser menor do que 8, menor do que 5 e menor do que 4, terá sua colocação como nó folha no extremo da subárvore esquerda. Observe na Figura ABB desbalanceada depois da inserção do nó com dado 3, que a ABB deixará de estar balanceada, pois haverá diferença de altura maior do que 1 entre os nós folha.

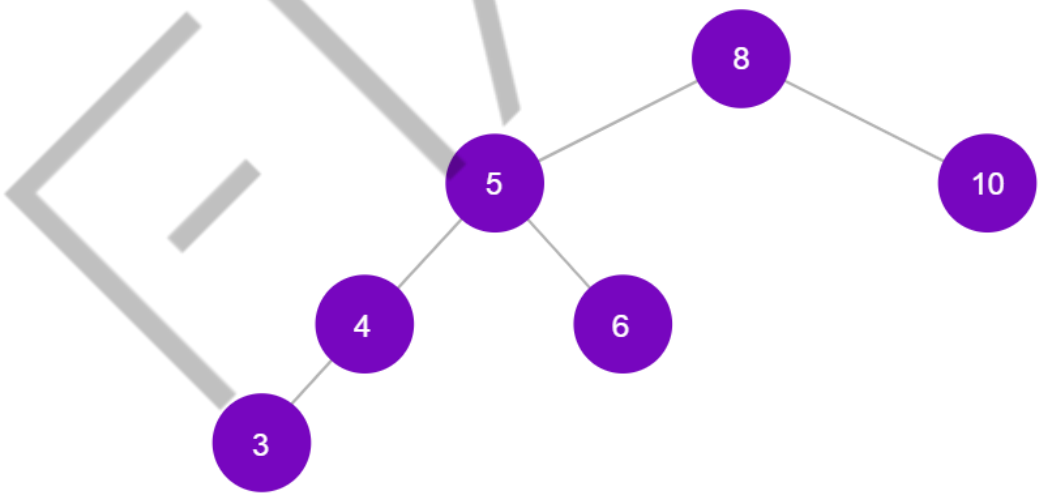


Figura 4.20 – ABB desbalanceada depois da inserção do nó com dado 3
Fonte: Elaborado pelo autor (2018)

Assim, após ocorrer a inserção de um novo nó no ramo esquerdo, pode-se passar a ter três situações com relação às alturas em que ficam suas subárvores,

chamaremos, a partir deste momento h_esq a altura da subárvore esquerda e h_dir a altura da subárvore direita:

- Se h_esq < h_dir, então subárvores esquerda e direita ficam com alturas diferentes, mas continuam balanceadas.
- Se h_esq = h_dir, então subárvores esquerda e direita ficam com alturas iguais e balanceamento foi melhorado.
- Se h_esq > h_dir, então subárvore esquerda fica ainda maior e o balanceamento foi violado.

Para saber em qual caso ficará a árvore com a nova inserção é preciso utilizar uma informação sobre as alturas das subárvores de cada nó.

Para que possamos ter as informações das alturas das subárvores esquerda e direita em cada nó, devemos alterar a definição do nó para que esse armazene também essas informações. Assim, o nó deve ser:

```
private static class ARVORE{
      public int dado;
      public ARVORE dir;
      public ARVORE esq;
      public int h_esq, h_dir;
      }
```

Para verificar se a árvore continua balanceada, é preciso calcular o Fator de Balanceamento (FB) de um nó, que é a altura da subárvore direita do nó menos a altura da subárvore esquerda do nó (**FB = h_dir – h_esq**). Se o FB de um nó for maior do que 2 ou menor do que – 2, a árvore estará desbalanceada.

Como mostra a Figura Árvore desbalanceada em que cada nó armazena as alturas das subárvores e é calculado o fator de balanceamento (FB) de cada nó, o FB do nó com dado 8 é -2 demonstrando que a árvore não está balanceada (ou seja, está quase completa).

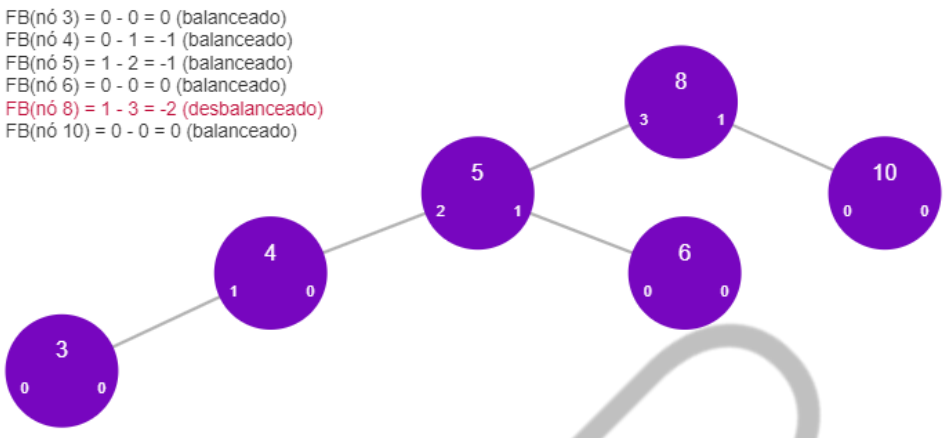


Figura 4.21 – Exemplo1: Árvore desbalanceada onde cada nó armazena as alturas das subárvores e é calculado o fator de balanceamento (FB) de cada nó
Fonte: Elaborado pelo autor (2018)

Usando os valores de FB de cada nó, vamos poder determinar se é necessário realizar um “rebalanceamento”. Caso seja, deve-se, então, fazer a rotação dos nós a fim de deixar a árvore balanceada novamente.

Vamos voltar ao exemplo da Figura Árvore desbalanceada em que cada nó armazena as alturas das subárvores e é calculado o fator de balanceamento (FB) de cada nó. O nó 8 tem FB=1 – 3 = -2 o que demonstra que a árvore está desbalanceada. Porém, para saber o que deve ser feito para a árvore voltar ao estado balanceado ainda é preciso analisar como está o balanceamento do seu filho, que está na subárvore com maior altura. Assim, é verificado que o nó 5 tem FB=1 – 2 = -1. Como os dois são de mesmo sinal, significa que a inserção do nó 3 deixou a árvore com maior altura apenas no ramo esquerdo, tanto em relação ao nó 8 como em relação ao seu nó filho (nó 5). Assim, deve-se fazer a rotação para a direita, já que FB é negativo, dos nós a partir do nó 8.

A função do trecho de programa a seguir mostra como a rotação para a direita em relação ao nó 8 é realizada. A Figura Sequência para efetuar rotação para direita a fim de balancear ABB depois da inserção do nó 3 apresenta a execução passo a passo a fim de efetuar a rotação para a direita.

```
public static ARVORE rotacao_direita (ARVORE p){
// faz rotação para direita em relação ao nó apontado por p
     ARVORE q,temp;
     q = p.esq;
```

```
        temp = q.dir;
        q.dir = p;
        p.esq = temp;
        return q;
        }
```

Código Fonte 4.5 – Função rotação de nós de uma árvore binária para direita em JAVA
Fonte: elaborado pelo autor (2018)

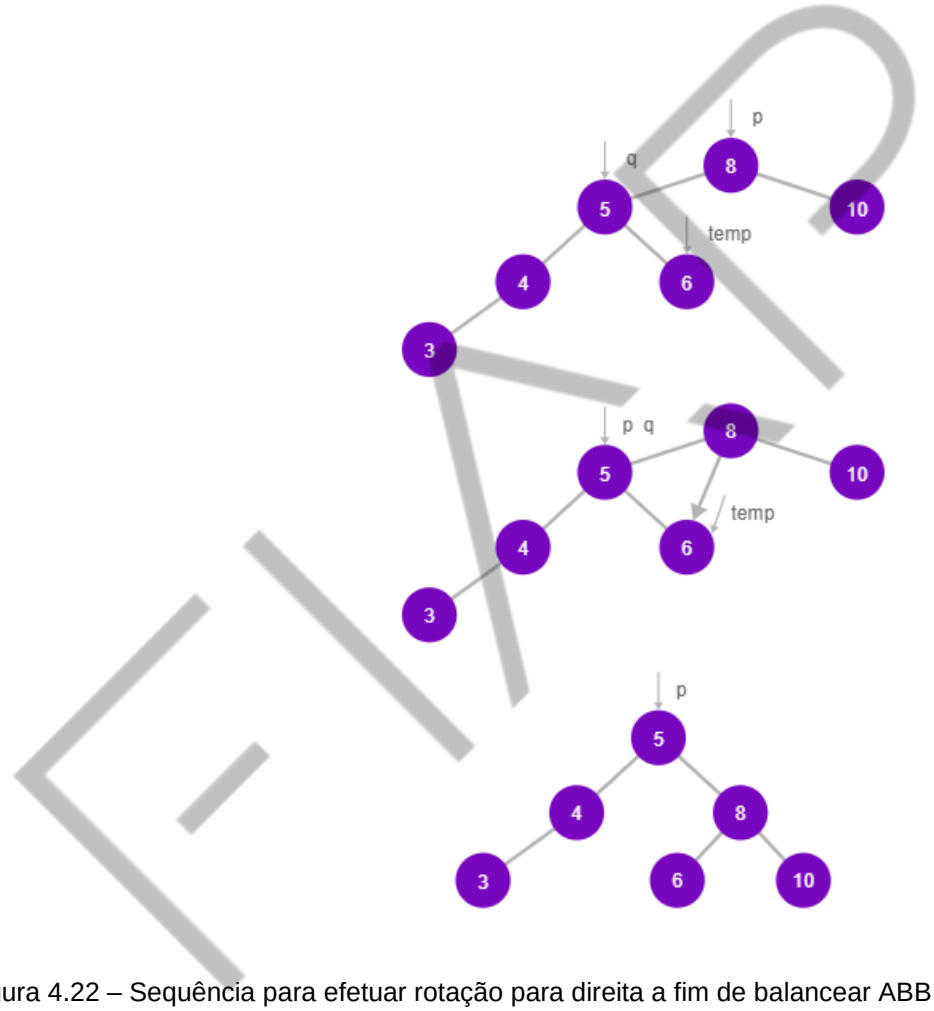


Figura 4.22 – Sequência para efetuar rotação para direita a fim de balancear ABB depois da inserção do nó 3
Fonte: Elaborado pelo autor (2018)

Observe que, ao final da rotação, a árvore volta a estar balanceada. Mas, ainda na nossa operação de rotação para a direita, não atualizamos as alturas das subárvores de cada nó. O código completo da função será apresentado mais adiante no texto, mas a ideia é seguinte:

- O ramo esquerdo do nó 8 (apontado por p) é alterado para ser o ramo que tem o nó apontado por temp (nó 6), assim, a altura será a maior entre alturas

das subárvores do nó apontado por temp acrescida de 1, como o nó 6 é um nó folha a altura da subárvore que ficará à esquerda do nó 8 é 1.

- O nó 5 (apontado por q) passa a ter como subárvore à direita o nó com dado 8, assim, a altura da subárvore direita terá que ser alterada para o maior valor entre as alturas dos ramos direito e esquerdo do nó 8, depois da atualização, após rotação somado a 1.

A árvore gerada pela rotação já com as alturas atualizadas é apresentada na Figura Árvore após balanceamento com as alturas das subárvores atualizadas e cálculo de FB de cada nó.

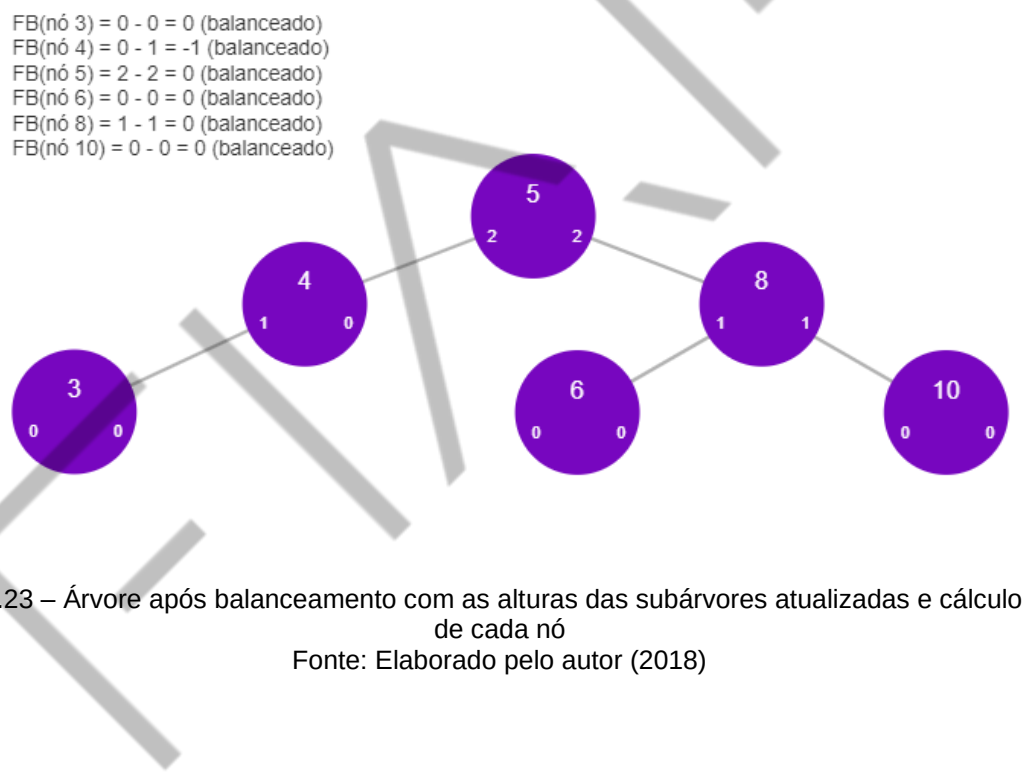


Figura 4.23 – Árvore após balanceamento com as alturas das subárvores atualizadas e cálculo de FB de cada nó
Fonte: Elaborado pelo autor (2018)

Vamos a uma nova situação, na qual, depois da inserção de um nó a árvore torna-se desbalanceada (). Nesse exemplo, foi feita a inserção do nó com dado 42, e por meio da análise do fator de balanceamento foi identificada a situação em que o FB do nó com dado 8 é 2.

É preciso também analisar o fator de balanceamento do nó que está à direita do nó 8 (que é o ramo com maior altura). Determinamos que FB (nó 20) = 2 - 1 = 1. Como os dois nós têm FB de mesmo sinal, significa que a inserção do nó 42 deixou a árvore com maior altura apenas no ramo direito, tanto em relação ao nó 8 como em

relação ao seu nó filho (nó 20). A operação a ser realizada para efetuar o balanceamento é uma rotação para a esquerda em relação ao nó 8. A Figura Exemplo1: Árvore desbalanceada em que cada nó armazena as alturas das subárvores e é calculado o fator de balanceamento (FB) de cada nó apresenta o passo a passo da rotação realizada usando a função apresentada no Código Fonte Função rotação de nós de uma árvore binária para esquerda em JAVA.

```
public static ARVORE rotacao_esquerda(ARVORE p) {
// faz rotação para esquerda em relação ao nó apontado por p
      ARVORE q,temp;
      q = p.dir;
      temp = q.esq;
      q.esq = p;
      p.dir = temp;
      return q;
      }
```

Código Fonte 4.6 – Função rotação de nós de uma árvore binária para esquerda em JAVA
Fonte: elaborado pelo autor (2018)

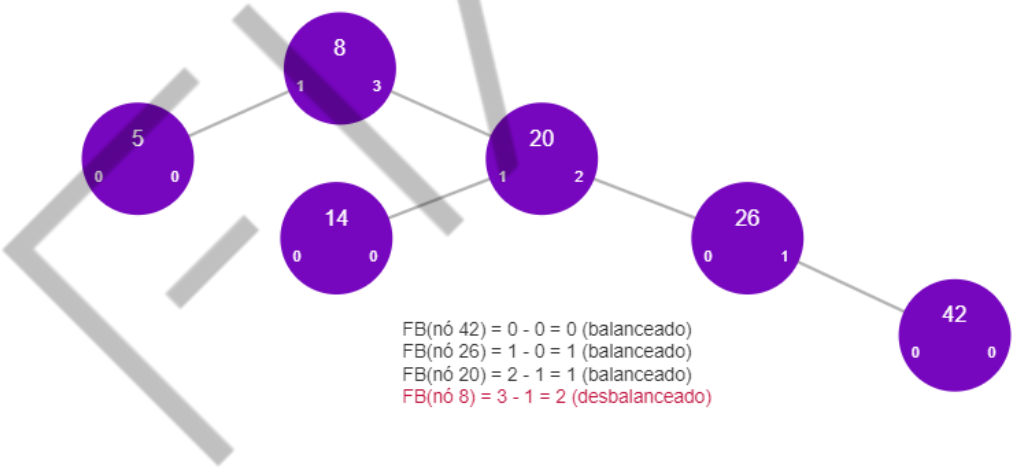


Figura 4.24 – Exemplo1: Árvore desbalanceada em que cada nó armazena as alturas das subárvores e é calculado o fator de balanceamento (FB) de cada nó
Fonte: Elaborado pelo autor (2018).

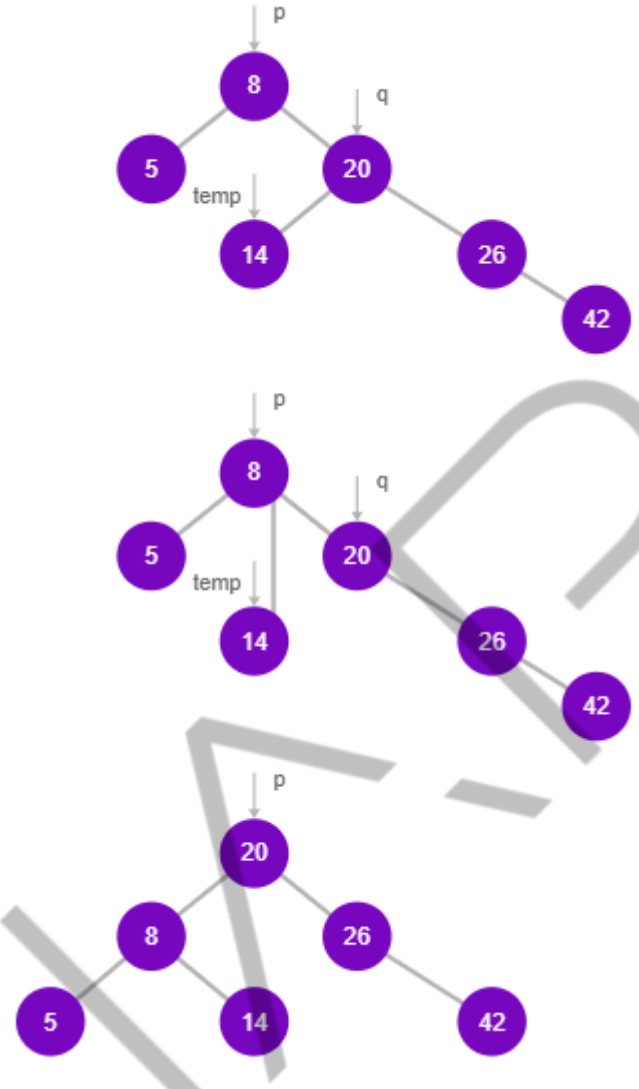


Figura 4.25 – Sequência para efetuar rotação para direita a fim de balancear ABB depois da inserção do nó 42
Fonte: Elaborado pelo autor (2018)

Muito bem, então podemos perceber que efetuando uma rotação conseguimos deixar a árvore novamente balanceada. Mas, existem regras definidas para realização de rotações? Sim, existem e estão definidas nos algoritmos de balanceamento AVL. O quadro Descrição de cada possível situação de não balanceamento da árvore e ações a serem realizadas apresenta as situações nas quais o fator de balanceamento indica a situação de não “equilíbrio” da árvore e qual a ação ou as ações que devem ser realizadas para que a árvore volte ao estado em que está balanceada. Observe que quando o FB de um nó indica que a árvore está desbalanceada e o nó filho do

ramo com maior altura tem FB com maior altura no sentido contrário, em vez de fazer apenas uma rotação são necessárias duas rotações.

FB do Nó	FB do Nó Filho da sub-árvore com maior altura	Operações
2	1	Rotação para esquerda em relação ao nó pai
	0	
	-1	Rotação para direita em relação ao nó filho Rotação para esquerda em relação ao nó pai
-2	1	Rotação para esquerda em relação ao nó filho Rotação para direita em relação ao nó pai
	0	Rotação para direita em relação ao nó pai
	-1	

Quadro 4.3 – Descrição de cada possível situação de não balanceamento da árvore e ações a serem realizadas.
Fonte: FIAP (2016)

A figura ABB desbalanceada depois da inserção do nó com dado 6 apresenta uma situação de árvore binária balanceada que após a inserção do nó com dado 10 necessita de aplicação do algoritmo AVL. O FB do nó com dado 12 é -2, já o FB do nó filho do ramo com maior altura (esquerda) é 1. Como os valores de FB são de sinais opostos, serão necessárias duas rotações, a primeira em relação ao nó filho (com dado 7) para a esquerda, e depois uma rotação para a direita em relação ao nó 12.

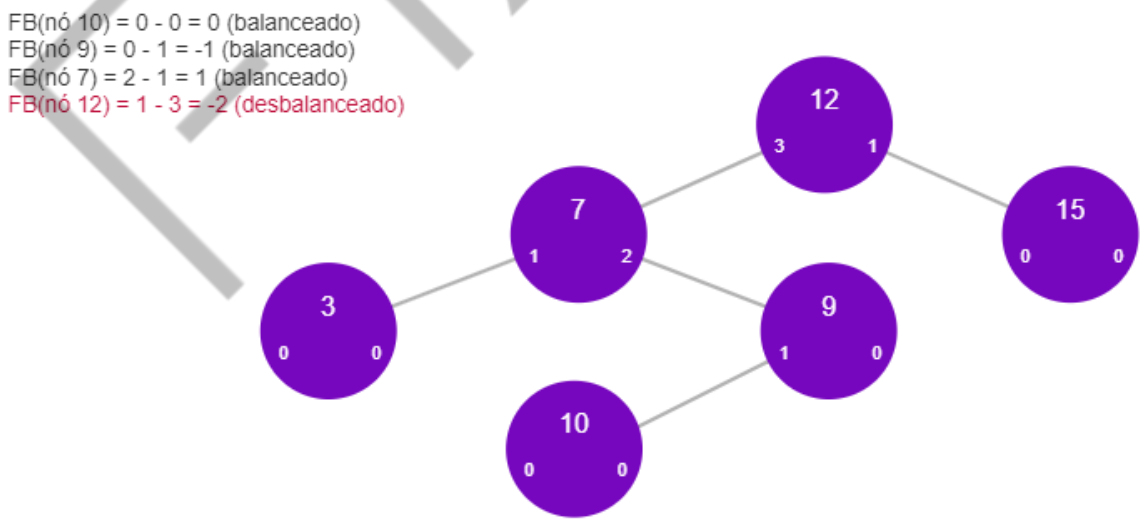


Figura 4.26 – ABB desbalanceada depois da inserção do nó com dado 6
Fonte: Elaborado pelo autor (2018)

O passo a passo da aplicação das operações na árvore é apresentado na Figura Sequência para efetuar duas rotações para balancear ABB depois da inserção do nó 10.

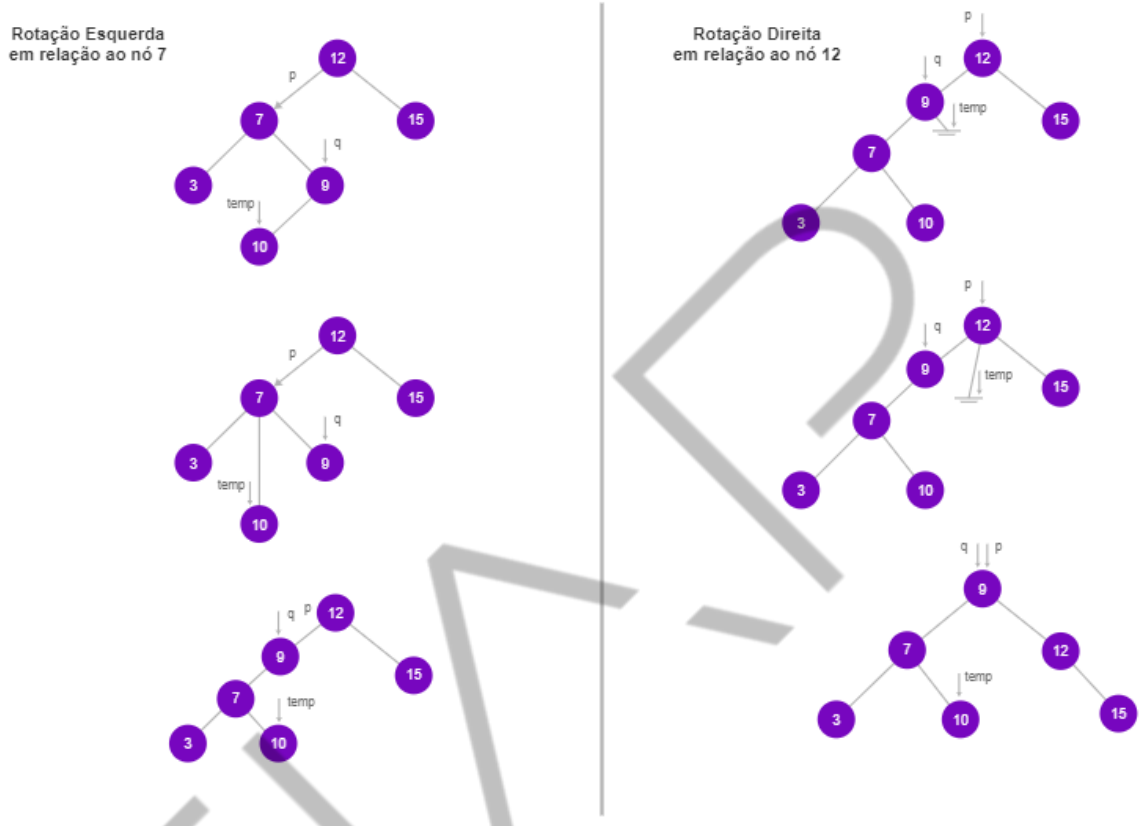


Figura 4.27 – Sequência para efetuar duas rotações para balancear ABB depois da inserção do nó 10
Fonte: Elaborado pelo autor (2018)

O código Fonte Implementação das operações: inserção, balanceamento e rotações de uma árvore AVL em JAVA apresenta a implementação completa das operações de inserção, balanceamento e rotações, sendo que essas últimas operações agora realizam também a atualização das informações de alturas h_dir e h_esq de cada nó.

```
public static ARVORE rotacao_direita (ARVORE p){
// faz rotação para direita em relação ao nó apontado por p
        ARVORE q,temp;
        q = p.esq;
        temp = q.dir;
        q.dir = p;
        p.esq = temp;
        if (temp == null)
              p.h_esq = 0;
        else if (temp.h_dir > temp.h_esq)
                    p.h_esq = temp.h_dir + 1;
              else
```

```
                    p.h_esq = temp.h_esq + 1;

          if (p.h_dir > p.h_esq)
                q.h_dir = p.h_dir + 1;
          else
                q.h_dir = p.h_esq + 1;
          return q;
          }

public static ARVORE rotacao_esquerda(ARVORE p) {
// faz rotação para esquerda em relação ao nó apontado por p
    ARVORE q,temp;
    q = p.dir;
    temp = q.esq;
    q.esq = p;
    p.dir = temp;
    if (temp == null)
          p.h_dir = 0;
    else if (temp.h_dir > temp.h_esq)
                p.h_dir = temp.h_dir + 1;
          else
                p.h_dir = temp.h_esq + 1;

    if (p.h_dir > p.h_esq)
          q.h_esq = p.h_dir + 1;
    else
          q.h_esq = p.h_esq + 1;
    return q;
    }

public static ARVORE balanceamento (ARVORE p) {
// analisa FB e realiza rotações necessárias para balancear árvore
    int FB = p.h_dir - p.h_esq;
    if (FB > 1) {
          int FB_filho_dir = p.dir.h_dir - p.dir.h_esq;
          if (FB_filho_dir >= 0)
                p = rotacao_esquerda(p);
          else {
                p.dir = rotacao_direita(p.dir);
                p = rotacao_esquerda(p);
                }
          }
    else {
          if (FB < -1) {
                int FB_filho_esq = p.esq.h_dir - p.esq.h_esq;
                if (FB_filho_esq <= 0)
                      p = rotacao_direita(p);
                else {
                      p.esq = rotacao_esquerda(p.esq);
                      p = rotacao_direita(p);
                      }
                }
          }
    return p;
    }
```

```
public static ARVORE inserir(ARVORE p, int info) {
// insere elemento em uma ABB
    if (p == null) {
          //insere nó como folha
          p=new ARVORE();
          p.dado = info;
          p.esq = null;
          p.dir = null;
          p.h_dir=0;
          p.h_esq=0;
    }
    else if (info < p.dado) {
                p.esq= inserir (p.esq, info);
                if (p.esq.h_dir > p.esq.h_esq)
                      p.h_esq = p.esq.h_dir + 1;
                else
                      p.h_esq = p.esq.h_esq + 1;
                p = balanceamento(p);
                }
          else {
                p.dir=inserir(p.dir, info);
                if (p.dir.h_dir > p.dir.h_esq)
                      p.h_dir = p.dir.h_dir + 1;
                else
                      p.h_dir = p.dir.h_esq + 1;
                p = balanceamento(p);
                }
          return p;
    }
```

Código Fonte 4.7 – Implementação das operações: inserção, balanceamento e rotações de uma árvore AVL em JAVA
Fonte: Elaborado pelo autor (2018)

Até agora, sempre “causamos” o desbalanceamento da árvore binária pela inserção de um novo nó, mas não é apenas na inserção que se dá a perda do balanceamento. Quando removemos um determinado nó podemos ou não deixar a árvore desbalanceada. Da mesma forma que na inserção, é o FB de cada nó que determina se será necessário efetuar rotações para devolver a árvore no estado balanceada. Porém, há uma importante diferença na identificação do balanceamento, pois, na inserção, o novo nó sempre será inserido na extremidade, ou seja, como nó folha. Já na remoção de um nó, podemos retirá-lo em um nível intermediário da subárvore. Por causa dessa possibilidade, a forma de atualizar as informações de alturas h_dir e h_esq de cada nó deve ser feita na árvore como um todo e, depois, realizar a operação de balanceamento.

O trecho de programa “Implementação das operações: remoção e atualização das alturas de uma árvore AVL e trecho do main() que as utiliza, escrito em JAVA” apresenta a implementação das operações de remoção de um nó em função do valor especificado e a atualização das alturas h_dir e h_esq.

```
  public static ARVORE remove_valor (ARVORE p, int info) {
     if (p!=null){
          if(info == p.dado){
               if (p.esq == null && p.dir==null)
                    return null;
               if (p.esq==null){
                    return  p.dir;
                    }
               else{
                    if (p.dir==null){
                         return p.esq;
                         }
                    else{
                         ARVORE aux, ref;
                         ref = p.dir;
                         aux = p.dir;
                         while (aux.esq != null)
                              aux = aux.esq;
                         aux.esq = p.esq;
                         return ref;
                         }
                    }
               }
          else{
               if(info<p.dado)
                    p.esq = remove_valor(p.esq,info);
               else
                    p.dir = remove_valor(p.dir,info);
               }
          }
  return p;
  }


  public static ARVORE atualiza_alturas(ARVORE p) {
  /*atualiza a informação da altura de cada nó depois da
remoção percorre a árvore usando percurso pós-ordem para ajustar
primeiro os nós folhas (profundidade maior) e depois os níveis acima
*/

     if( p != null) {
          p.esq = atualiza_alturas(p.esq);
          if (p.esq == null)
               p.h_esq = 0;
          else if (p.esq.h_esq > p.esq.h_dir)
                    p.h_esq = p.esq.h_esq+1;
               else
                    p.h_esq = p.esq.h_dir+1;
```

```
                                p.dir = atualiza_alturas(p.dir);
            if (p.dir == null)
                  p.h_dir = 0;
            else if (p.dir.h_esq > p.dir.h_dir)
                        p.h_dir = p.dir.h_esq+1;
                  else
                        p.h_dir = p.dir.h_dir+1;

            p = balanceamento(p);
            }
      return p;
   }


   public static void main(String[] args) {
      Scanner entra = new Scanner(System.in);
      ARVORE raiz = init();
      int info;
   . . .

   /*trecho que remove um nó com um determinado valor escolhido pelo
usuário */
      System.out.print("Digite valor do nó a ser removido: ");
      info = entra.nextInt();
      raiz = remove_valor(raiz, info);
      raiz = atualiza_alturas(raiz);
   . . .
   }
```

Código Fonte 4.8 – Implementação das operações: remoção e atualização das alturas de uma árvore AVL e trecho do main() que as utiliza, escrito em JAVA
Fonte: Elaborado pelo autor (2018)

4.5 Armazenamento Secundário: Árvores-B

4.5.1 MOTIVAÇÃO

Uma base de dados formada por um grande volume de registros pode ficar armazenada apenas na memória principal (memória RAM) de um computador? Normalmente, não. Isso se deve ao fato de o espaço de memória necessário para armazenar a base de dados ser muito maior do que a capacidade de armazenamento da memória RAM. Assim, a base de dados deve ser armazenada em um sistema de memória secundária, por exemplo, um HD (disco rígido). O problema é que a memória secundária tem tempo de acesso muito maior do que a memória RAM. Segundo Tanenbaum e Austin (2013, p. 67) um acesso à memória principal necessita de

apenas 10 ns (nano segundos 10^{-9} s), já o tempo de acesso de um disco de estado sólido (SSD) é 10 vezes maior, e se for disco magnético, 100 vezes maior do que o tempo de acesso da memória RAM.

Suponha a base de dados da receita federal, que armazena milhões de registros de todos os cidadãos que possuem CPF, e que a consulta/alteração de um registro de um cidadão deve ser identificada pela chave CPF. Para que o acesso seja mais rápido, a base dados pode organizar os registros usando uma árvore AVL similar à Figura Árvore Binária de Busca (ABB) que armazena CPF de votos. Apenas que nesse caso, pela enorme quantidade de registros, a árvore deve ficar armazenada na memória secundária. Supondo a busca pelo CPF 221.780.459.1 seriam necessárias apenas três comparações para encontrar tal registro, porém seriam três acessos ao disco, ou seja, com tempo muito maior do que a árvore se estivesse completamente armazenada na memória principal. Isso faz com que o desempenho da aplicação tenha um tempo de execução muito maior piorando consideravelmente sua eficiência.

No entanto, sem entrar em detalhes de hardware e sistemas operacionais, o processador para executar nossos programas apenas acessa de forma direta a memória RAM. Assim quando iniciamos a execução de um programa, o Sistema Operacional (SO) copia do HD para a RAM as instruções e dados desse programa, e só depois disso é que o processador pode iniciar execução de fato.

Como a memória RAM tem capacidade bem menor do que o HD e pode ser que, para executar as diversas aplicações simultaneamente, a RAM não tenha capacidade suficiente, o SO resolve isso trabalhando com um conceito chamado memória virtual. Nesse conceito, a memória que o SO “vê” é maior do que a que existe fisicamente na máquina. Assim, de cada processo (programa em execução) o SO ocupa apenas blocos de bytes necessários para o momento da execução. Os blocos de bytes em que a memória é “dividida” são conhecidos como páginas.

Fique calmo, você vai estudar isso com mais cuidado quando chegar a Sistemas Operacionais. Essa breve explicação é suficiente para o que usaremos neste assunto.

Voltando ao exemplo de uma grande base de dados contendo registros identificados unicamente por uma chave, como no exemplo anterior, o CPF. O arquivo que contém essa base deve estar, pelo seu tamanho, na memória secundária e deve-

se fazer a busca de registro pela chave. O SO deve carregar para a memória principal (RAM) apenas parte desse arquivo. Mas, como localizar um registro entre N registros do arquivo, se a memória só comporta M registros, sendo M bem menor do que N? A Figura Arquivo em Memória Secundária com N registros (apenas suas chaves apresentadas) que precisa ser copiado para a RAM mostra um esquema dessa situação, sendo apenas apresentada a chave de cada registro para simplificação da figura.

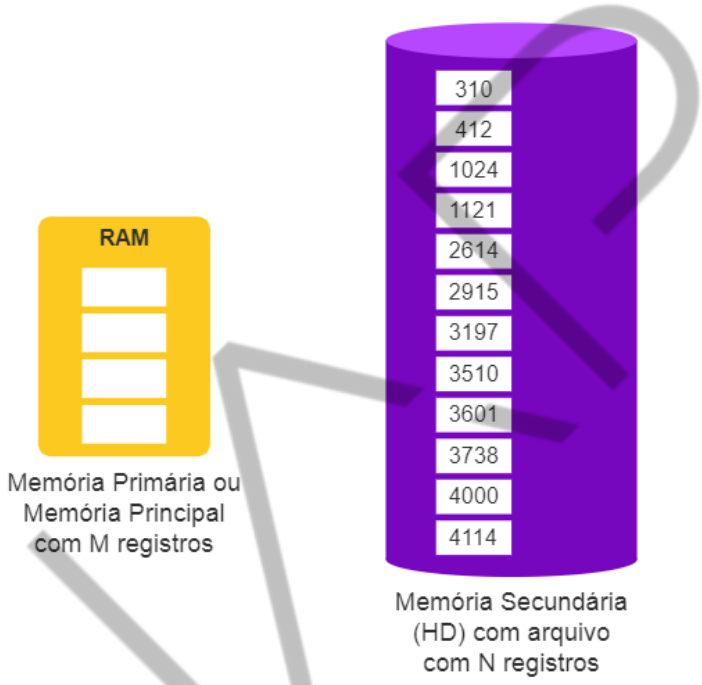


Figura 4.28 – Arquivo em Memória Secundária com N registros (apenas suas chaves apresentadas) que precisa ser copiado para a RAM
Fonte: Elaborado pelo autor (2018)

A solução desse problema inicia com a ordenação dos registros (a ordenação já está representada na figura). Em seguida, o arquivo é separado em M blocos de registros. Com a separação, o que deve ficar armazenado na RAM são M registros, assim, apenas o registro inicial de cada bloco em que foi “dividido” o arquivo será armazenado na RAM, como mostra a Figura Memória RAM com M registros: registro inicial de cada bloco de registros do arquivo da Memória Secundária com N registros.

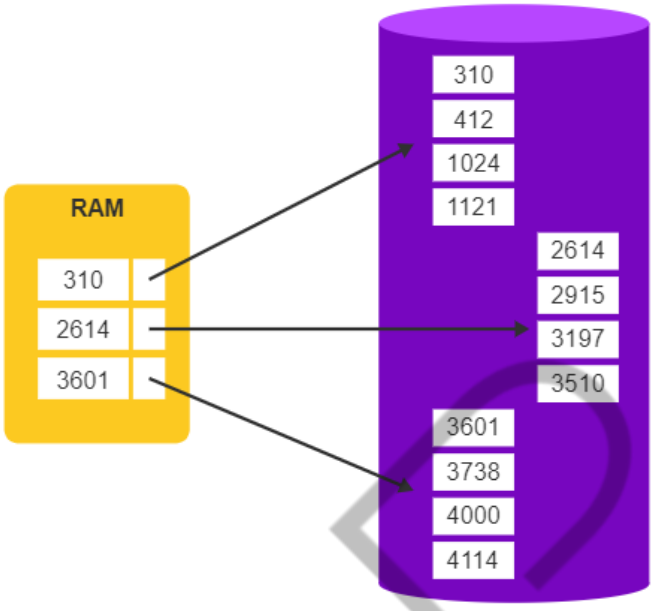


Figura 4.29 – Memória RAM com M registros: registro inicial de cada bloco de registros do arquivo da Memória Secundária com N registros
Fonte: Elaborado pelo autor (2018)

Vamos supor que se deseje acessar o registro com chave 3197. O processo inicia procurando, de forma sequencial nos registros da RAM: chave procurada é maior do que 310, chave é maior do que 2614, mas chave é menor do que 3601. Assim, se o registro existir está no bloco (ou página) que contém registros maiores do que 2614 e menores do que 3601. Portanto, para localizar tal registro, agora, basta trazer do disco para a RAM o bloco em que esse registro pode estar e fazer a busca pela chave.

Resolvidos todos os problemas como esse? Ainda não.

Vamos voltar para o exemplo da receita federal e o CPF como chave do registro de cada cidadão. O volume de registros N do arquivo da base de dado torna-se muito maior do que M registros da memória primária, assim a RAM pode não ter espaço para armazenar todas as chaves de cada bloco em que foi “dividido” o arquivo.

Uma possível solução para esse problema é usar outro tipo de árvore, chamada de árvore B.

4.5.2 O que são árvores B?

De acordo com Ziviani (2000, p. 170): “Quando uma árvore de pesquisa possui mais de um registro por nó ela deixa de ser binária. Essas árvores são chamadas de n-árias, pelo fato de possuírem mais de dois descendentes por nó. Nesses casos, os nós são mais comumente chamados de páginas.”

A árvore B é uma árvore n-ária em que cada nó, ou página, pode conter mais de uma chave. Vamos, primeiro, apresentar a definição formal:

- Cada página pode conter m chaves e, necessariamente, m+1 filhos;
- A raiz é a única que pode conter de 1 até m chaves;
- Todas as folhas estão no mesmo nível.

A Figura Árvore B com quatro chaves em cada página com três níveis apresenta uma árvore B que usaremos para explicar como trabalhar com esse tipo de árvore e como ela auxilia no problema que descrevemos com relação à base de dados.

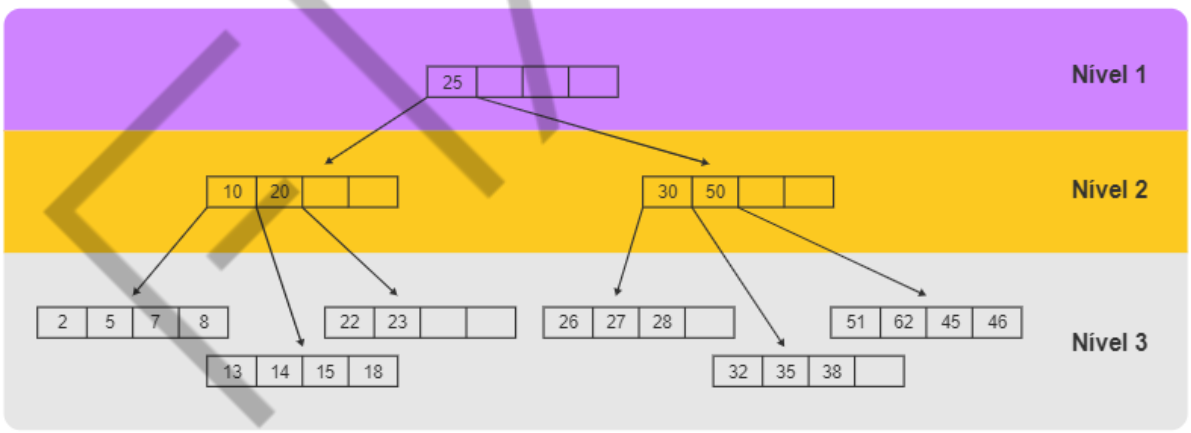


Figura 4.30 – Árvore B com 4 chaves em cada página com 3 níveis
Fonte: Elaborado pelo autor (2018)

Cada página pode, nesse exemplo de árvore B, ter quatro chaves. Observe que a organização é similar à árvore de busca binária. As chaves com valores menores do que a raiz estão na subárvore esquerda e os maiores estão à direita.

Em relação ao nó filho da esquerda do nó raiz que tem como primeira chave o valor 10 e depois o valor 20, esse nó tem três subárvores (2 chaves e 2+1

descendentes). A subárvore mais à esquerda tem chaves menores do que 10, já o nó filho “do meio” tem apenas chaves maiores do que 10, mas menores do que 20. E, finalmente, a subárvore mais à direita é composta apenas por valores maiores do que 20 (É claro, menores do que 25, já que estão no ramo mais à esquerda da raiz da árvore).

Em relação ao nó filho da direita do nó raiz que tem como primeira chave o valor 30 e depois o valor 50. Como tem duas chaves precisa ter três subárvores, a mais à esquerda com valores menores do que 30, a subárvore do centro com valores acima de 30 e menores do que 50 e, a subárvore mais à direita com valores maiores do que 50.

Agora que já apresentamos o que é uma árvore B, a pergunta que não quer calar: mas como ela nos auxilia quando o volume de registros é tão grande que não pode ser “carregado” na RAM?

Para isso, vamos comparar como essas chaves seriam organizadas na árvore de busca binária. Na ABB, cada nó poderia ter apenas uma chave, o que, com certeza, faria com que o número de níveis fosse muito maior. Lembre-se de que, para localizar uma chave em uma ABB, o número máximo de comparações é o número de níveis e, se o arquivo está em armazenado no HD, implica que serão feitos mais acessos à memória mais lenta do computador.

Quando trabalhamos com árvores B, o número de acessos à memória é por página, sendo que, em cada página, a busca pode ser feita de forma sequencial (como se fosse um vetor). Assim, a estrutura de uma árvore-B torna possível ter uma quantidade menor de acessos à memória percorrendo menor número de níveis.

Neste momento do curso, apenas descrevemos a ideia de como esse tipo de árvore, classificada como árvore multidimensional, pode melhorar o desempenho em de um sistema de gerenciamento de banco de dados. Embora o uso desse tipo de árvore e de suas variações como árvore B* e árvore B+ sejam muito utilizadas, as suas implementações são mais complexas, pois a cada inserção ou remoção pode haver redistribuição das chaves nas páginas. Essa redistribuição deve ocorrer quando as alturas dos nós folha (de qualquer subárvore) não estiverem no mesmo nível, por causa dessa regra, é possível que a quantidade de chaves em cada página seja alterada conforme a árvore insira ou retire um registro.

Muito bem, chegamos ao fim da explicação sobre árvores, que tal, então, exercitar o que foi apresentado? É uma boa, pois só exercitando chegamos a dominar o assunto e a conquistar a maturidade em programação computacional.

4.6 Exercícios de Fixação

4.6.1 ÁRVORES – GRAU E PROFUNDIDADE

Com relação à árvore da figura a seguir responda:

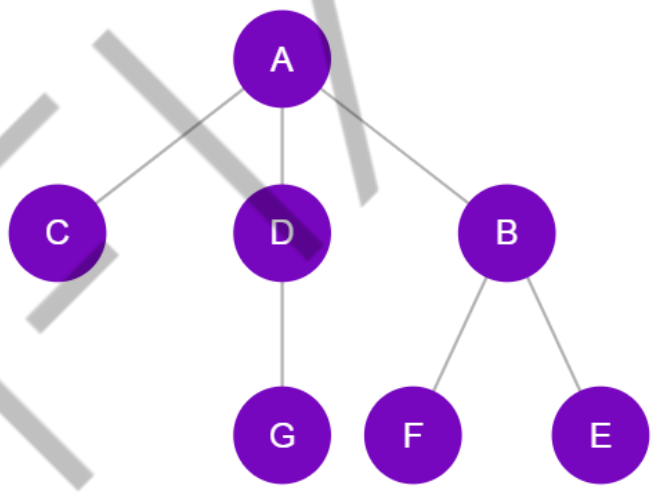
a. Qual é o nó raiz?

b. Quais nós são nós folhas?

c. Qual a relação entre os nós B e F e entre os nós D e B?

d. Qual é o grau da árvore? Justifique.

e. Qual é a profundidade da árvore? Justifique.



RESULTADO

a. Nó A

b. C, G,F e E

c. Nó B é pai do nó F.

O nó D é irmão do nó B, pois são filhos no mesmo nó A.

d. Grau da árvore é 3, pois é o maior número de filhos de um nó da árvore.

e. Profundidade 3, pois raiz está no nível 1 e o maior nível de um nó folha (G, F e E) é 3.

4.6.2 ÁRVORE ABB

1. Apresente como fica a árvore de busca binária (ABB) inserindo a seguinte sequência de valores: 54, 45, 71, 48, 66, 23, 52, 15 e 46.

2. Com a ABB gerada pelo exercício anterior represente o esquema gráfico de cada uma das remoções:
   a. Remove nó com dado 66;
   b. Remove nó com dado 45.

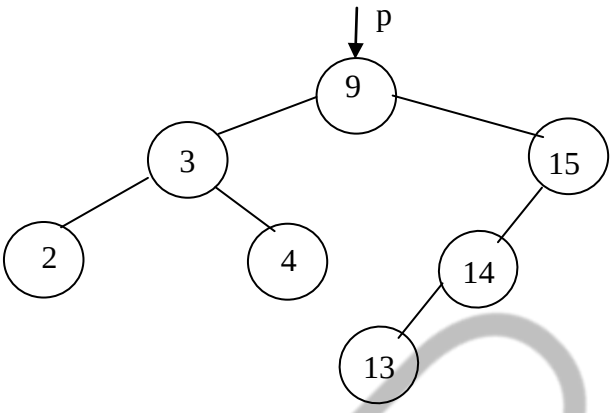
3. Elabore um programa que usando as funções já estudadas de árvores binárias crie um menu com as seguintes opções:
   0 - Encerra o programa
   1 - Insere um nó na ABB lendo do teclado o valor do dado desse novo nó
   2 - Remove um nó da árvore com o valor escolhido pelo usuário
   3 - Apresenta, usando o percurso em ordem, todos os nós da árvore

4.6.3 ÁRVORE AVL

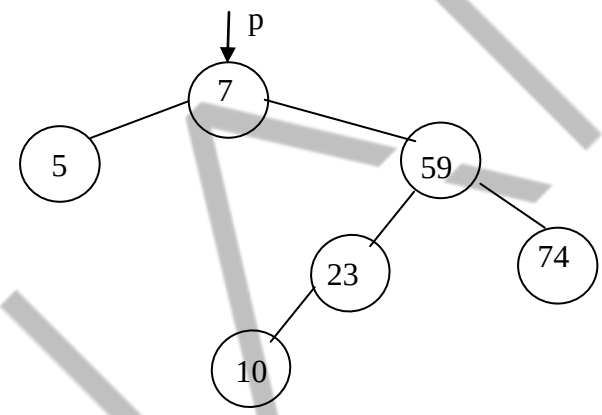
1) Qual a altura de uma árvore com 250 elementos?

2) Aplique os critérios de balanceamento de uma árvore AVL, conforme descrito neste texto, para as seguintes árvores:

a)

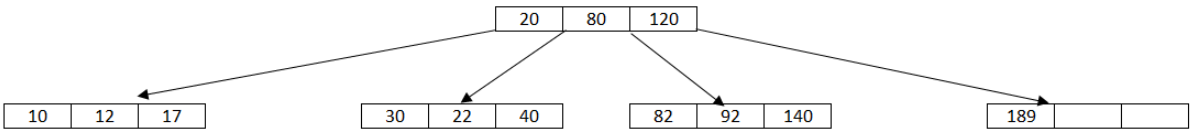


b)

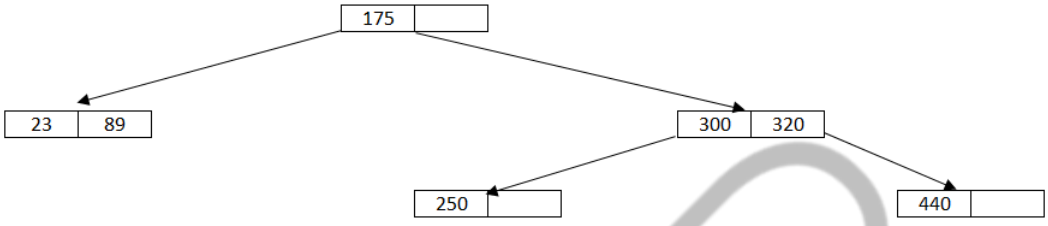


4.6.4 ÁRVORE B

1) Supondo uma árvore B que pode ter até três chaves em cada nó, explique por que cada afirmação está errada:
   a. Cada nó terá até três descendentes (filhos).
   b. A árvore abaixo segue todas as regras de formação de uma árvore B.



2) Em uma árvore B que pode ter até duas chaves, se forem inseridas as seguintes chaves: 300, 250, 440, 89, 23, 175 e 320, explique por que a configuração a seguir não pode ser usada para armazenar essas chaves. Apresente uma configuração válida.



REFERÊNCIAS

ASCÊNCIO, A. F. G.; ARAUJO, G. S. **Estruturas de Dados**: Algoritmos, Análise de Complexidade e Implementações em JAVA e C/C++. São Paulo: Pearson Prentice Hall, 2010.

DEITEL, P. J.; DEITEL, H. M. **Java**: como programar. 8. ed. São Paulo: Pearson Prentice Hall, 2010.

PEREIRA, S. L. **Estruturas de Dados Fundamentais**: Conceitos e Aplicações. São Paulo: Érica, 1996.

TENEMBAUM, A. M. et al. **Estruturas de Dados usando C.** Makron Books Ltda, 1995.

TANENBAUM, A. S.; AUSTIN, T. **Organização Estruturada de Computadores**. 6. ed. São Paulo: Pearson Prentice Hall, 2013.

ZIVIANI, N. **Projeto de algoritmos com implementações em Pascal e C**. São Paulo: Pioneira, 2000.