UNISINOS

Estruturas Avançadas de Dados I
(Árvores Binárias)

Prof. Gilberto Irajá Müller

Última atualização 05/03/2018

Definição I

- Uma árvore binária *T* é um conjunto finito de elementos denominados nós, tal que:
  - Se *T* = 0, a árvore é dita vazia, ou
  - Existe um nó especial *r*, chamado raiz de *T*, e os restantes podem ser divididos em dois subconjuntos disjuntos, Tr^E e Tr^D, a subárvore esquerda e a direita de r, respectivamente, as quais são também árvores binárias;
  - A raiz da subárvore esquerda (direita) de um nó *v*, se existir, é denominada filho esquerdo (direito) de *v*. Naturalmente, o esquerdo pode existir sem o direito e vice-versa;
  - Uma árvore binária pode ter duas subárvores vazias (a esquerda e a direita). Toda árvore binária com *n* nós possui exatamente n + 1 subárvores vazias entre suas subárvores esquerdas e direitas.

UNISINOS

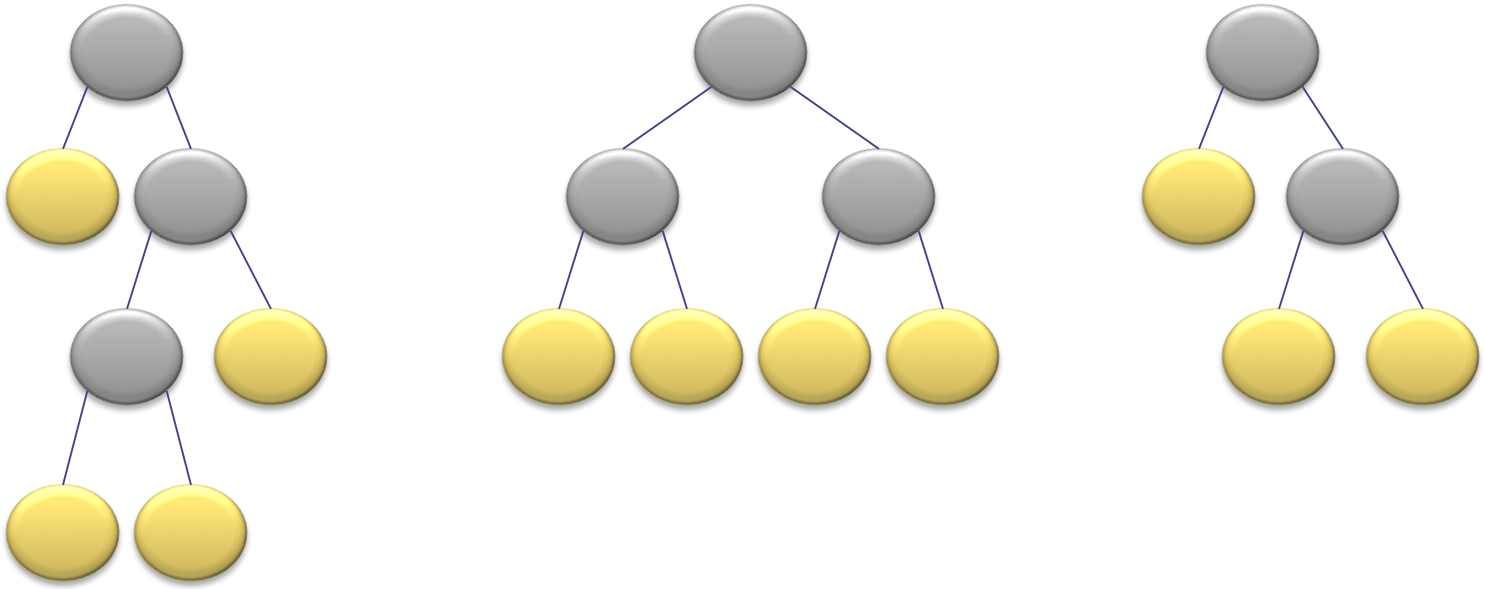
Definição II

- Uma árvore binária é uma árvore cujos os nós têm 0, 1 ou 2 filhos e cada filho é designado como filho à esquerda ou filho à direita (**Grau 2**);

- O número de folhas é uma importante característica das árvores binárias para mensurar a eficiência esperada de algoritmos.

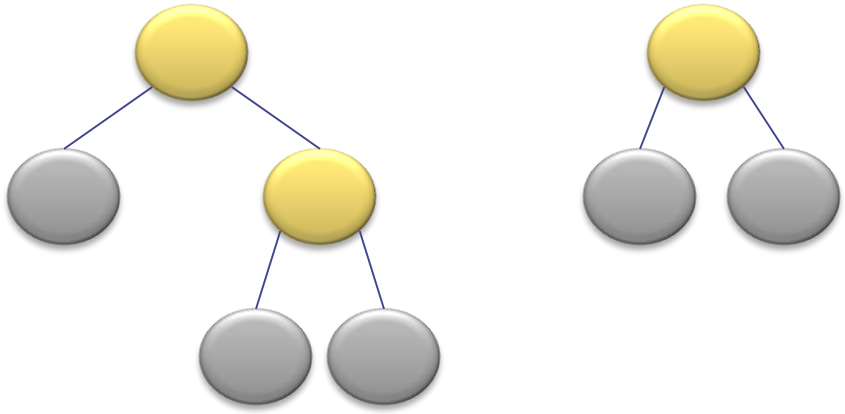
Classificação de Árvores Binárias

- Árvore Estritamente Binária
  - Nenhum nó pode ter **filho único**;
  - Uma árvore com *n* folhas terá $2n - 1$ nós.



Classificação de Árvores Binárias

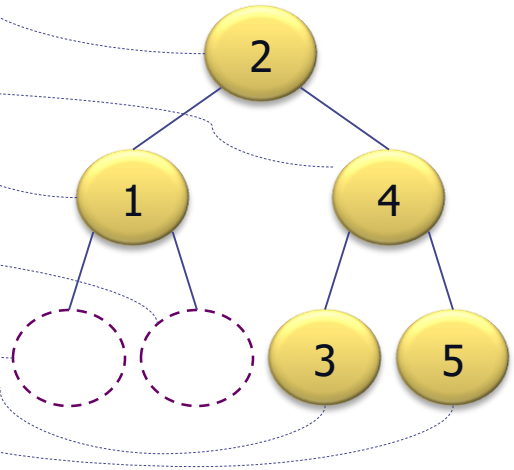
- Árvore Binária Completa (ou Quase Completa)
  - Uma árvore binária de nível *n* é uma árvore binária completa se:
    - cada nó com menos de dois filhos deve estar no nível n ou no nível n-1.



Classificação de Árvores Binárias

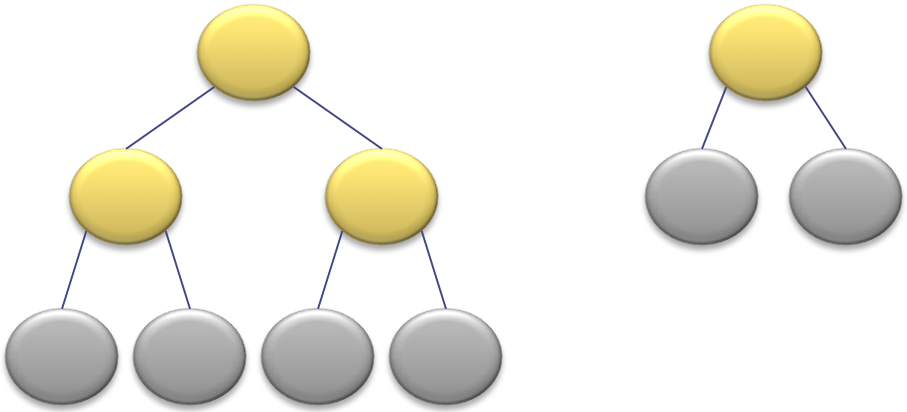
- Árvore Binária Completa (ou Quase Completa)
  - Pode ser usado uma estrutura do tipo array para armazená-la.
  - Filho esquerdo do nó ficará em 2 * index + 1;
  - Filho direito do nó ficará em 2 * index + 2;
  - Pai ficará na posição (index – 1) /2.

0	2
1	1
2	4
3	null
4	null
5	3
6	5



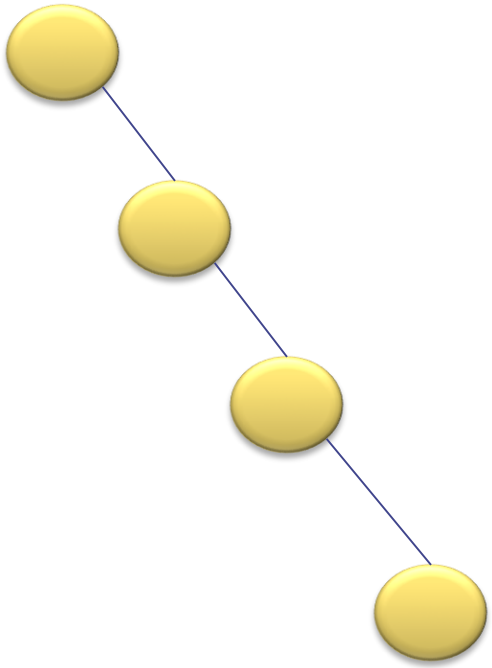
Classificação de Árvores Binárias

- Árvore Binária Cheia
  - Uma árvore binária cheia de nível *n* possui todos os nós folhas no nível *n*;
  - Uma árvore cheia com altura *h*, terá $2^{h+1} - 1$ nós.



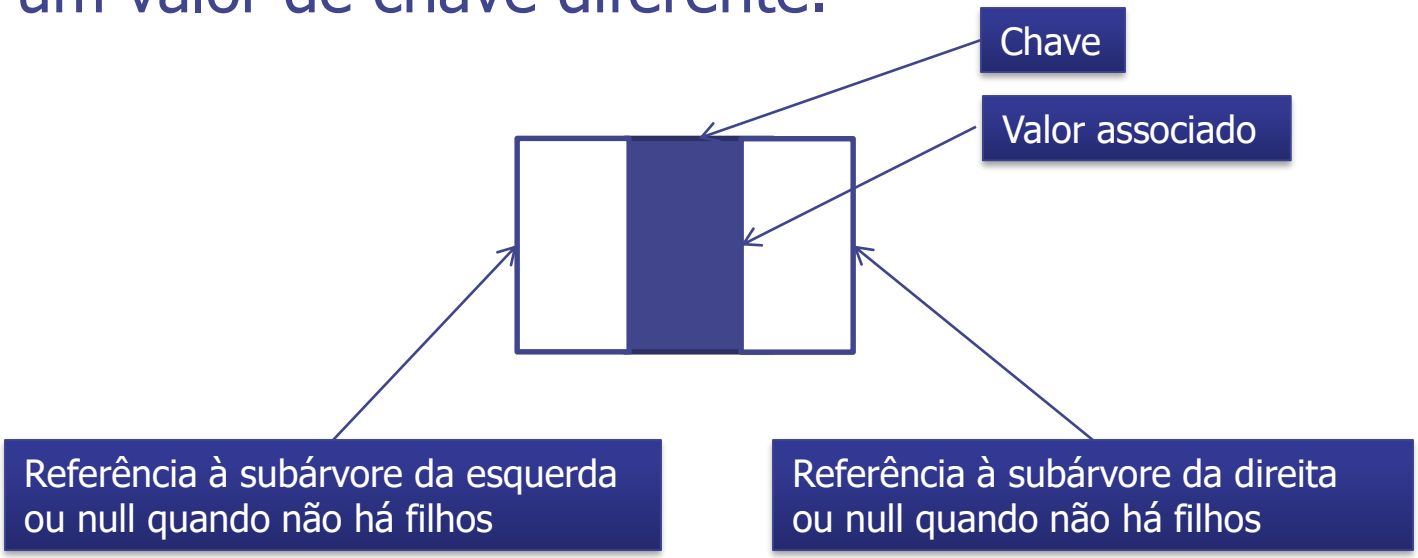
Classificação de Árvores Binárias

- Árvore Degenerada
  - Cada nó possui exatamente um filho e a árvore tem o mesmo número de níveis que de nós;
  - Análise de pior caso ocorre neste tipo de árvore.

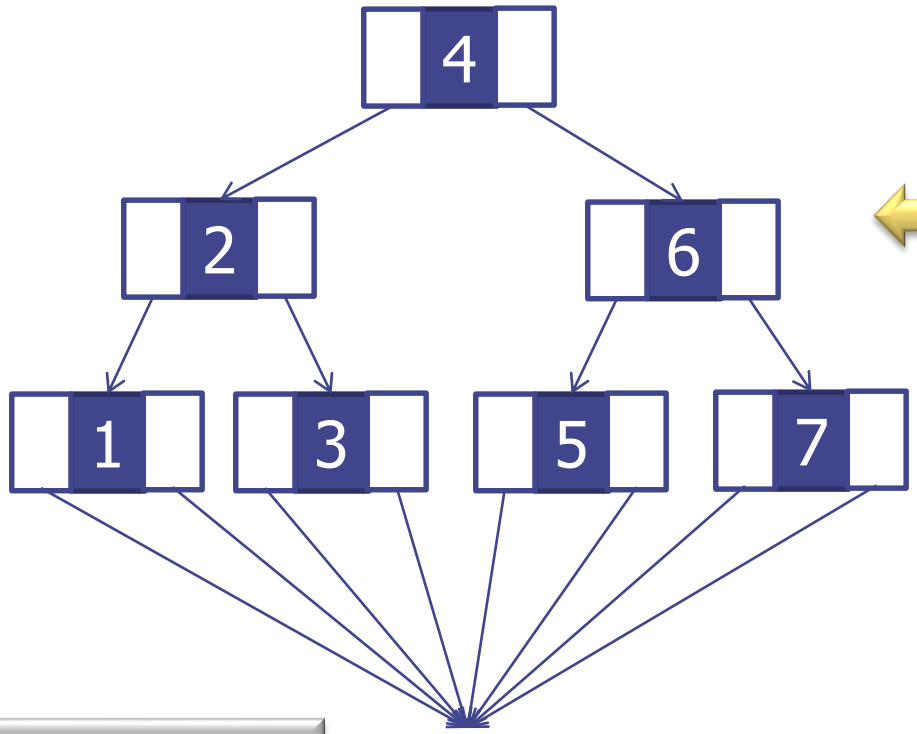


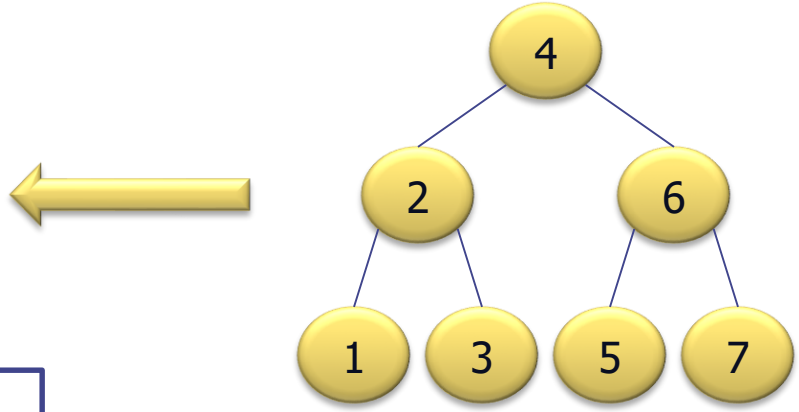
Árvore Binária de Pesquisa (BST)

- Também chamada de árvore binária de busca ou árvore binária ordenada;
- Apresenta uma relação de ordem entre os nós;
- A ordem é definida por um campo denominado **chave;**
- Não permite chave duplicada, ou seja, cada nó tem um valor de chave diferente.

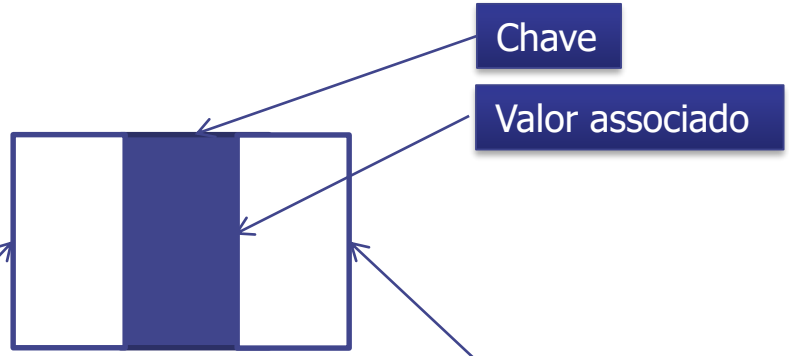


Estrutura de dados de uma BST





Muitas implementações referenciam o pai no nó.



Referência à subárvore da esquerda ou null quando não há filhos

Referência à subárvore da direita ou null quando não há filhos

Árvore Binária de Pesquisa

- **Exercício 1:** Crie uma árvore binária de pesquisa inserindo as chaves conforme a ordem abaixo:

g, d, a, c, m, o, b, d, a, l, p, q

Árvore Binária de Pesquisa

- **Exercício 2:** Suponha que temos números entre 1 e 1000 em uma BST e queremos procurar pelo número 363. Qual(is) das sequências a seguir NÃO poderia(m) ser a(s) sequência(s) de nós examinados?

a) 2, 252, 401, 398, 330, 344, 397, 363.

b) 924, 220, 911, 244, 898, 258, 362, 363.

c) 925, 202, 911, 240, 912, 245, 363.

d) 2, 399, 387, 219, 266, 382, 381, 278, 363.

e) 935, 278, 347, 621, 299, 392, 358, 363.

Árvore Binária de Pesquisa

- **Exercício 3:** O professor Bunyan pensa ter descoberto uma importante propriedade de BST. Suponha que a pesquisa da chave k em uma BST termine em uma folha. Considere três conjuntos: A, as chaves à esquerda do caminho de pesquisa; B, as chaves no caminho de pesquisa; e C, as chaves à direita do caminho de pesquisa. O professor afirma que três chaves quaisquer $a \in A$, $b \in B$ e $c \in C$ devem satisfazer a $a \leq b \leq c$. Forneça um contraexemplo mínimo possível para a afirmação do professor.

Interface BinarySearchTreeADT

```
public interface BinarySearchTreeADT<K, V> {
    public void clear();
    public boolean isEmpty();
    public V search(K key);
    public void insert(K key, V value);
    public boolean delete(K key);
    public void preOrder();
    public void inOrder();
    public void postOrder();
    public void levelOrder();
}
```

Classe BinarySearchTree

```
public class BinarySearchTree<K extends Comparable<K>, V> implements BinarySearchTreeADT<K, V> {
    protected Node root;

    protected class Node {
        private K key;
        private V value;
        private Node left, right;

        public Node(K key, V value) {
            this.key = key;
            this.value = value;
        }

        public Node next(K other) {
            return other.compareTo(key) < 0 ? left : right;
        }

        public boolean isLeaf() {
            return left == null && right == null;
        }
        @Override
        public String toString() { return "" + key; }
    }

    @Override
    public void clear() { root = null; }

    @Override
    public boolean isEmpty() { return root == null; }

    // Continua...
```

Nested Class

Árvore Binária de Pesquisa – Desempenho

- A procura de um valor em uma **BST é mais rápido do que** a procura em **listas encadeadas e arrays**; similar à pesquisa binária vista em Programação II;

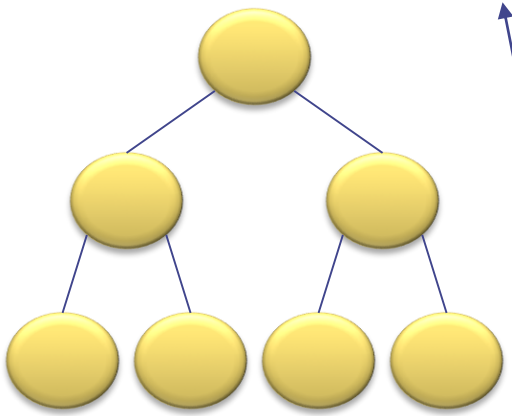
- A complexidade pode ser medida pelo número de comparações feitas durante o processo de busca, inserção ou exclusão; lembrando que majoritariamente as implementações de árvore usam recursão;

- Dependerá do número de nós encontrados no caminho único que leva da raiz ao nó procurado/inserido/excluído;

- A complexidade dependerá da forma da árvore e da posição do nó procurado na árvore. O número médio de comparações em uma busca é $O(\log_2(n))$.

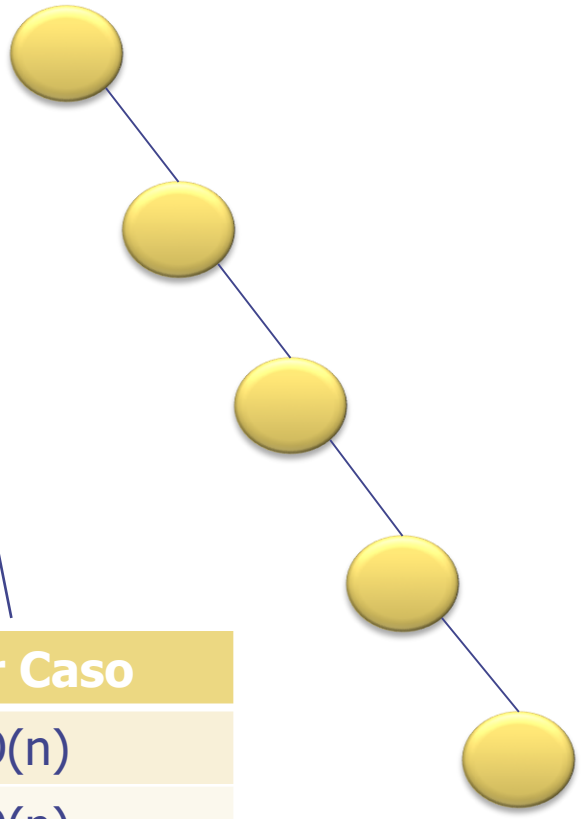
Árvore Binária de Pesquisa – Desempenho

Binária Cheia



h = 2

Degenerada



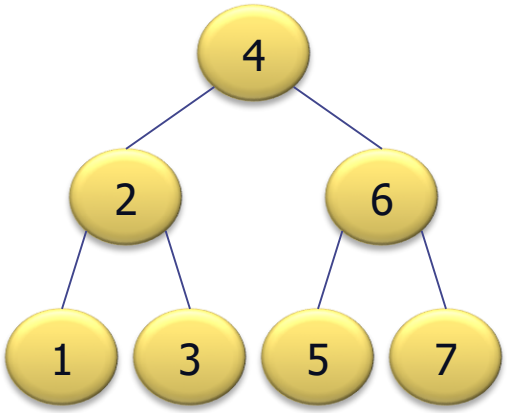
h = 4

	Caso médio	Pior Caso
Busca	$O(\log_2(n))$	O(n)
Inserção	$O(\log_2(n))$	O(n)
Exclusão	$O(\log_2(n))$	O(n)

Árvore Binária de Pesquisa – Busca

```
@Override
public V search(K key) {
    return search(root, key);
}

private V search(Node node, K key) {
    if (node == null) {
        return null;
    } else if (key.compareTo(node.key) == 0) {
        return node.value;
    }
    return search(node.next(key), key);
}
```

Árvore Binária de Pesquisa – Busca



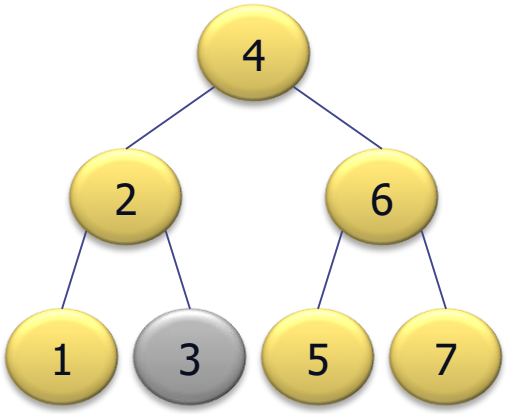
Retorno para o usuário

Pilha de Chamadas (Busca da chave 3)
1. Chamada ao search(root, 3) == (node: 4)
2. Chave é menor, então, chama search(node.left, 3) == (node: 2)
3. Chave é maior, então, chama search(node.right, 3) == (node: 3)
4. Chave é igual, então, retorna o valor do node: 3, desempilhando todas as chamadas com este valor.

Árvore Binária de Pesquisa – Inserção

```java
@Override
public void insert(K key, V value) {
    root = insert(root, key, value);
}

private Node insert(Node node, K key, V value) {
    if (node == null) {
        return new Node(key, value);
    } else if (key.compareTo(node.key) > 0) {
        node.right = insert(node.right, key, value);
    } else if (key.compareTo(node.key) < 0) {
        node.left = insert(node.left, key, value);
    }

    return node;
}
```

Árvore Binária de Pesquisa – Inserção



Pilha de Chamadas (Inserção da chave 3)	Estado do node
1. Chamada ao root = insert(root, 3, 3)	
2. Chave é menor, então, chama node.left = insert(node.left, 3, 3) == (node: 2)	node = 4
3. Chave é maior, então, chama node.right = insert(node.right, 3, 3) == (node: null).	node = 2
4. Nó é null, então, retorna um novo nó com a chave 3 e valor 3.	node = 3

Árvore Binária de Pesquisa – Visualização

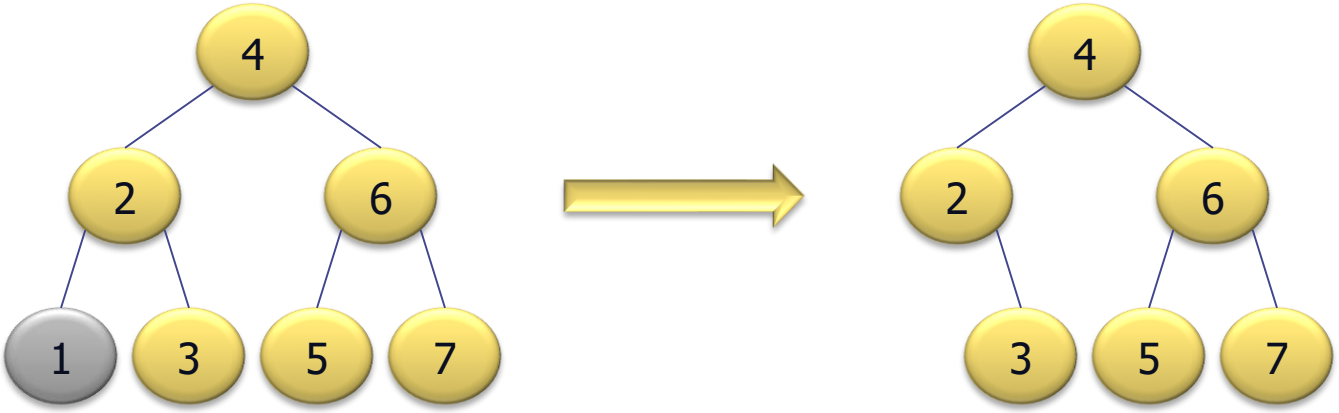
```
@Override
public String toString() {
    return root == null ? "[empty]" : printTree(new StringBuffer());
}

private String printTree(StringBuffer sb) {
    if (root.right != null) {
        printTree(root.right, true, sb, "");
    }
    sb.append(root + "\n");
    if (root.left != null) {
        printTree(root.left, false, sb, "");
    }

    return sb.toString();
}

private void printTree(Node node, boolean isRight, StringBuffer sb, String indent) {
    if (node.right != null) {
        printTree(node.right, true, sb, indent + (isRight ? "        " : " |      "));
    }
    sb.append(indent + (isRight ? " /" : " \\") + "----- " + node + "\n");
    if (node.left != null) {
        printTree(node.left, false, sb, indent + (isRight ? " |      " : "        "));
    }
}
```

Árvore Binária de Pesquisa – Exclusão

- Existem três casos ao excluir um nó:
  - Exclusão de uma **folha**;
  - Exclusão de um **nó** que possui **um** filho;
  - Exclusão de um nó que possui **dois** filhos:
    - Exclusão por **cópia**;
    - Exclusão por **fusão**.

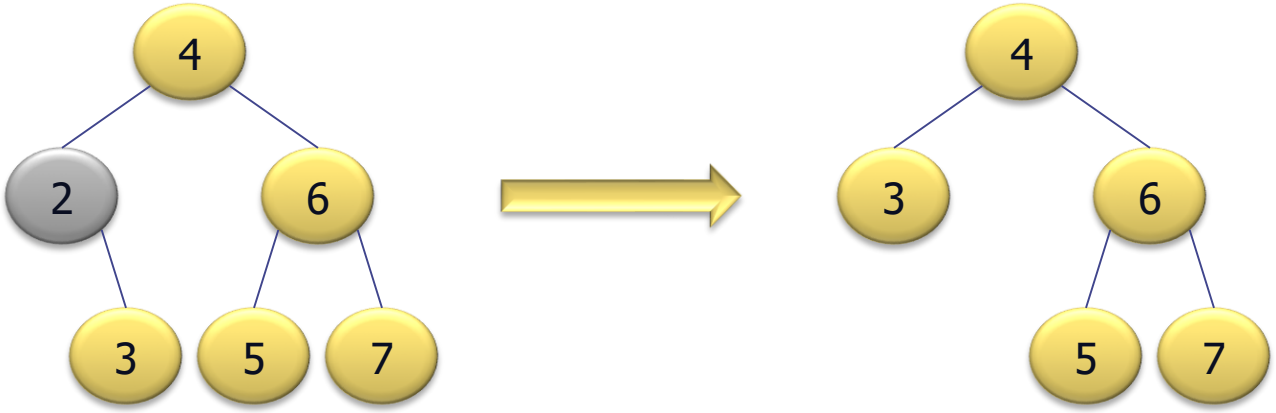
Árvore Binária de Pesquisa – Exclusão

- **Caso 1**: Exclusão de uma folha
  - Na referência do nó pai é atribuído o valor **null**.



Árvore Binária de Pesquisa – Exclusão

- **Caso 2**: Exclusão de uma nó que possui **UM** filho
  - A referência do nó pai aponta para o nó filho do nó excluído.

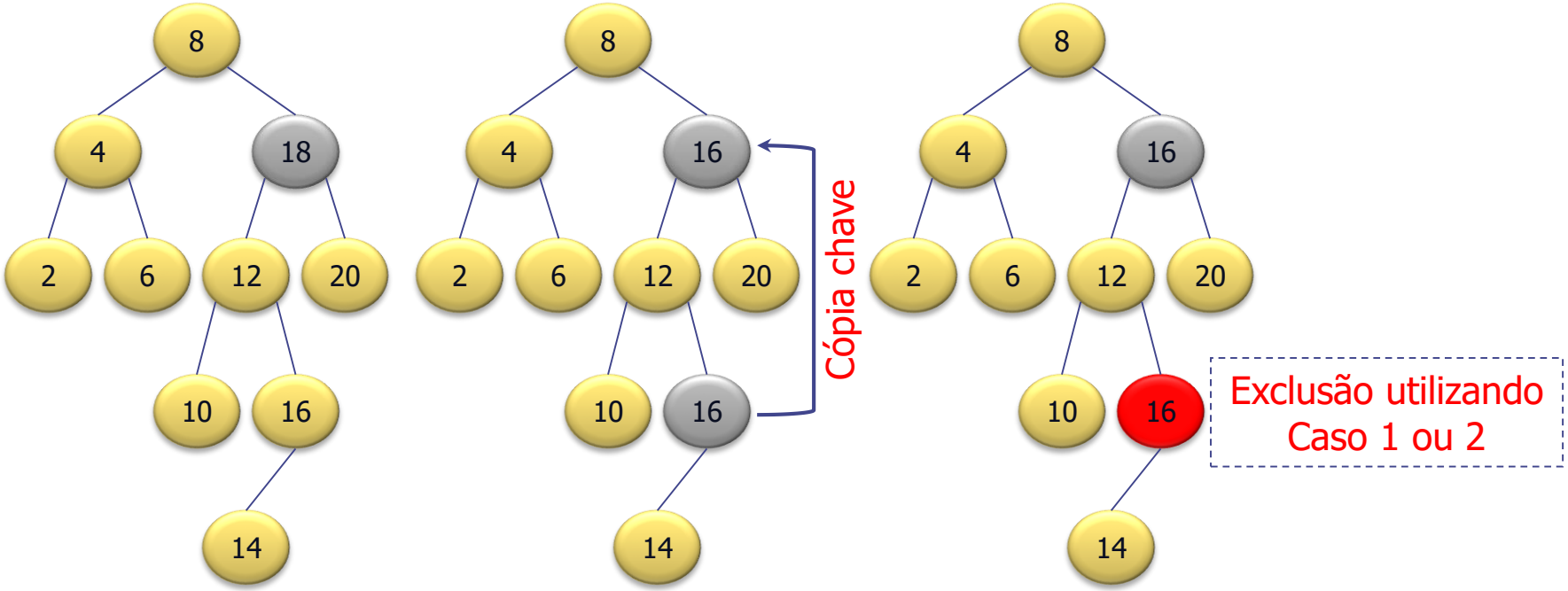


Árvore Binária de Pesquisa – Exclusão por Cópia

- **Caso 3:** Exclusão por Cópia (um nó que possui **DOIS** filhos)
  - Remove uma chave k1 (chave do nó à excluir):
    - **Sobrescrevendo-a** por uma outra chave k2 (**o maior valor na subárvore esquerda**);
    - Então **removendo o nó que contém k2** (que será um dos casos simples: folha ou nó com apenas um filho).

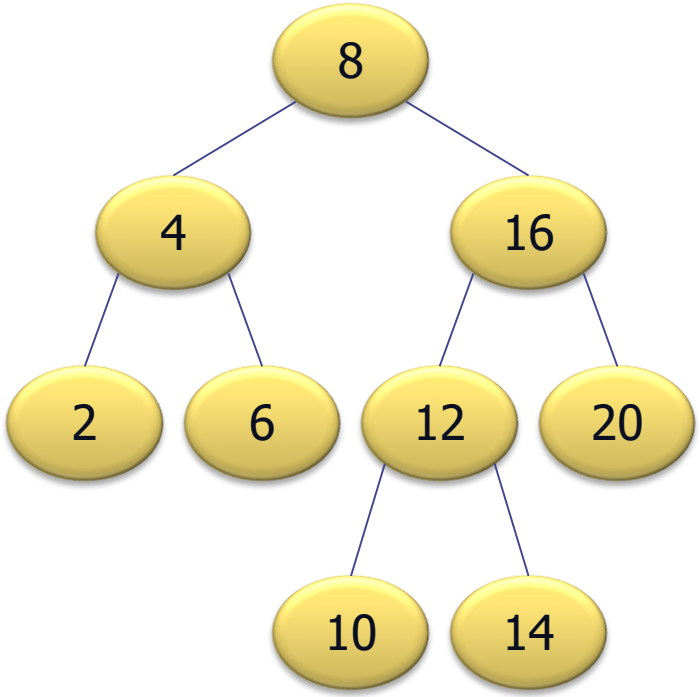
Árvore Binária de Pesquisa – Exclusão por Cópia

- Exclusão por Cópia (um nó que possui **DOIS** filhos)



Árvore Binária de Pesquisa – Exclusão por Cópia

- Exclusão por Cópia (um nó que possui **DOIS** filhos)



Árvore após exclusão por cópia

Árvore Binária de Pesquisa – Exclusão por Cópia

```
private boolean deleteByCopying(K key) {
    Node parent = null, current = root;
    for (; current != null && key.compareTo(current.key) != 0; parent = current,
current = current.next(key));

    if (current == null)
        return false;
    else if (current.left != null && current.right != null) {
        // Caso 3
        Node tmp = current.left;
        while (tmp.right != null) tmp = tmp.right;
        deleteByCopying(tmp.key);
        current.key = tmp.key;
    } else {
        // Caso 1 ou Caso 2
        Node nextNode = current.right == null ? current.left : current.right;
        if (current.equals(root)) root = nextNode;
        else if (parent.left.equals(current)) parent.left = nextNode;
        else parent.right = nextNode;
    }

    return true;
}
```

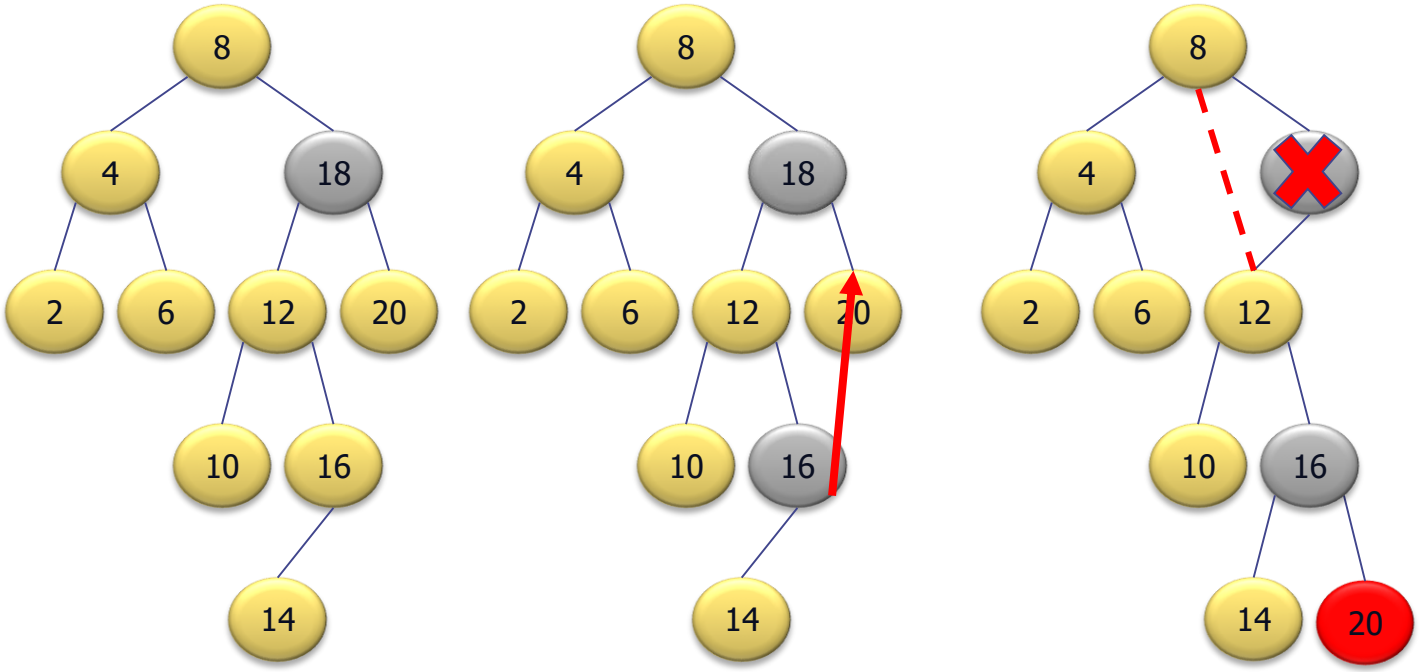
```
@Override
public boolean delete(K key) {
   return deleteByCopying(key);
}
```

Árvore Binária de Pesquisa – Exclusão por Fusão

- Exclusão por Fusão (um nó que possui **DOIS** filhos)
  - A solução consiste em **fusionar as duas subárvores do nó a ser excluído** em uma;
  - Para tanto, como na organização da árvore binária, todos os valores da subárvore à esquerda são menores que os valores da subárvore à direita, deve-se **encontrar o maior valor na subárvore esquerda e torná-lo a raiz da subárvore direita**. Também pode-se procurar o nó com menor valor da subárvore direita;
  - Remove a chave, excluindo o nó que contém a chave. E o pai do nó removido passa a apontar para a nova subárvore.

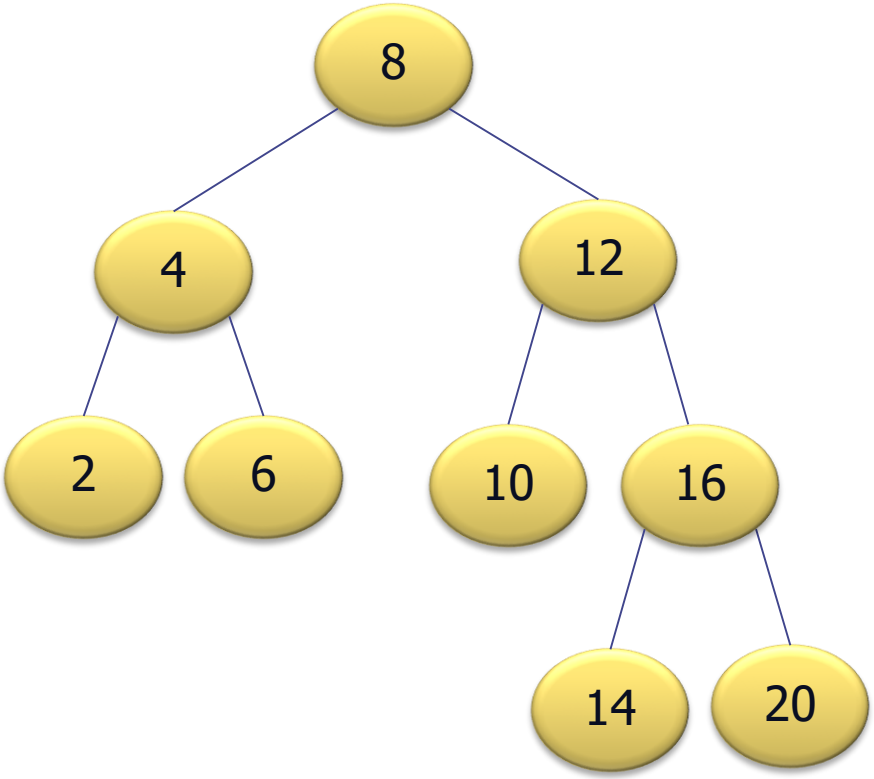
Árvore Binária de Pesquisa – Exclusão por Fusão

- Exclusão por Fusão (um nó que possui **DOIS** filhos)



Árvore Binária de Pesquisa – Exclusão por Fusão

- Exclusão por Fusão (um nó que possui **DOIS** filhos)



Árvore após exclusão por fusão

Árvore Binária de Pesquisa – Exclusão por Fusão

```
private boolean deleteByMerging(K key) {
    Node parent = null, current = root;
    for (; current != null && key.compareTo(current.key) != 0; parent = current, current =
current.next(key));

    if (current == null)
        return false;
    else if (current.left != null && current.right != null) {
        // Caso 3
        Node tmp = current.left;
        while (tmp.right != null) tmp = tmp.right;
        tmp.right = current.right;

        if (current.equals(root)) root = current.left;
        else if (parent.left.equals(current)) parent.left = current.left;
        else parent.right = current.left;
    } else {
        // Caso 1 ou Caso 2
        Node nextNode = current.right == null ? current.left : current.right;
        if (current.equals(root)) root = nextNode;
        else if (parent.left.equals(current)) parent.left = nextNode;
        else parent.right = nextNode;
    }

    return true;
}
```

```
@Override
public boolean delete(K key) {
   return deleteByMerging(key);
}
```

Percursos (traversal) em Árvores Binárias

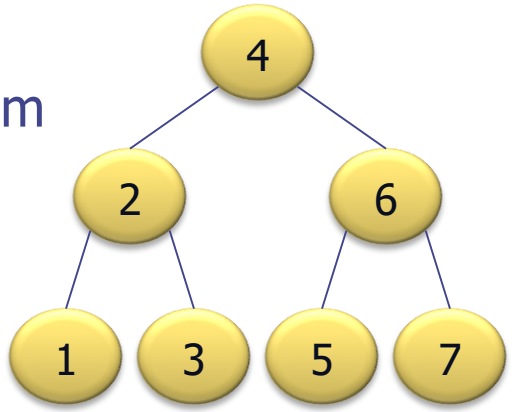
- Conforme mencionado na nota de aula anterior, temos dois percursos:
  - **Percurso em profundidade**: os nós da subárvore atual têm prioridade na ordem de acesso;
  - **Percurso em amplitude (nível)**: os nós de menor nível têm prioridade na ordem de acesso.

Percursos (traversal) em Árvores Binárias

- Em profundidade, temos três tipos "canônicos":
  - Pré-ordem
  - Pós-ordem
  - Em-ordem
- Em amplitude, temos:
  - Em nível

Caminhamento Pré-ordem

- Visitar a raiz
- Percorrer a subárvore esquerda em pré-ordem
- Percorrer a subárvore direita em pré-ordem

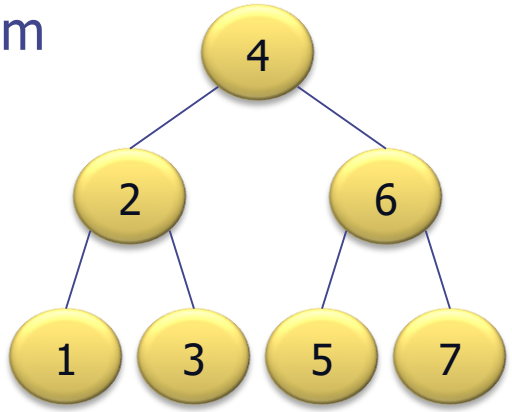


Pilha de Chamadas
1. Imprime 4
2. Imprime 2
3. Imprime 1
4. Imprime 3
5. Imprime 6
6. Imprime 5
7. Imprime 7

```
@Override
public void preOrder() {
    preOrder(root);
}

private void preOrder(Node node) {
    if (node != null) {
        System.out.print(node + " ");
        preOrder(node.left);
        preOrder(node.right);
    }
}
```

Caminhamento Em-ordem

- Percorrer a subárvore esquerda em pré-ordem
- Visitar a raiz
- Percorrer a subárvore direita em pré-ordem

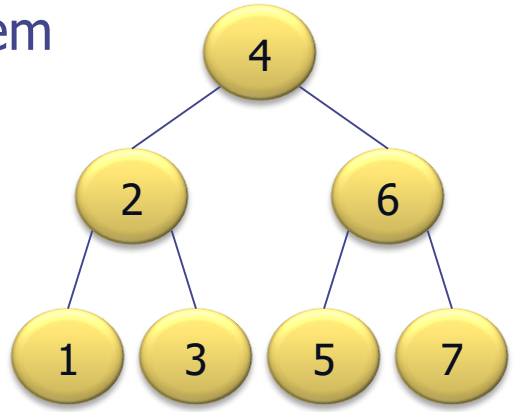


Pilha de Chamadas
1. Imprime 1
2. Imprime 2
3. Imprime 3
4. Imprime 4
5. Imprime 5
6. Imprime 6
7. Imprime 7

```
@Override
public void inOrder() {
    inOrder(root);
}

private void inOrder(Node node) {
   if (node != null) {
      inOrder(node.left);
      System.out.print(node + " ");
      inOrder(node.right);
   }
}
```

Caminhamento Pós-ordem

- Percorrer a subárvore esquerda em pós-ordem
- Percorrer a subárvore direita em pós-ordem
- Visitar a raiz

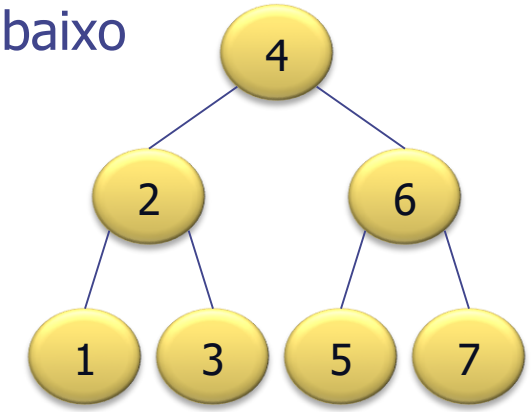


Pilha de Chamadas
1. Imprime 1
2. Imprime 3
3. Imprime 2
4. Imprime 5
5. Imprime 7
6. Imprime 6
7. Imprime 4

```
@Override
public void postOrder() {
    postOrder(root);
}

private void postOrder(Node node) {
    if (node != null) {
      postOrder(node.left);
      postOrder(node.right);
      System.out.print(node + " ");
    }
}
```

Caminhamento em Nível

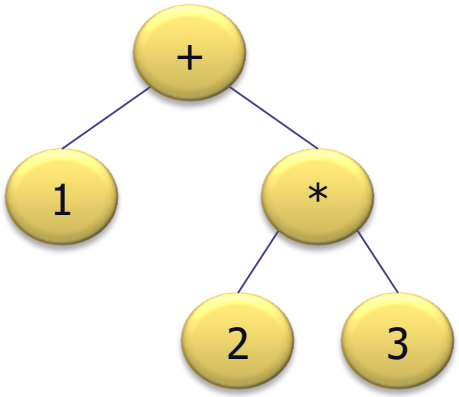
- Percorre-se a árvore em nível de cima para baixo e da esquerda para a direita;
- Usa-se uma estrutura auxiliar (fila).



Pilha de Chamadas
1. Imprime 4
2. Imprime 2
3. Imprime 6
4. Imprime 1
5. Imprime 3
6. Imprime 5
7. Imprime 7

```
@Override
public void levelOrder() {
    levelOrder(root);
}

private void levelOrder(Node node) {

                    ?

}
```

Exemplo do uso de Percursos



- A árvore de expressão acima representa "1 + 2 * 3"
- Os operandos sempre estarão nos nós folhas e os operadores nos nós não terminais;

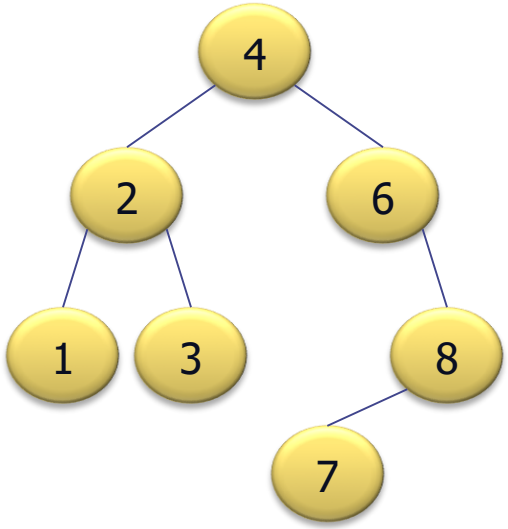
- Pré-ordem: + 1 * 2 3
- Pós-ordem: 1 2 3 * + (notação polonesa)
- Em-ordem: 1 + 2 * 3 (ordem da expressão)

Exercícios

- **Exercício 4.** Desenhe uma árvore de pesquisa com a inserção dos elementos na seguinte ordem: K G G B L O C P A U D. Apresente os quatro percursos estudados!

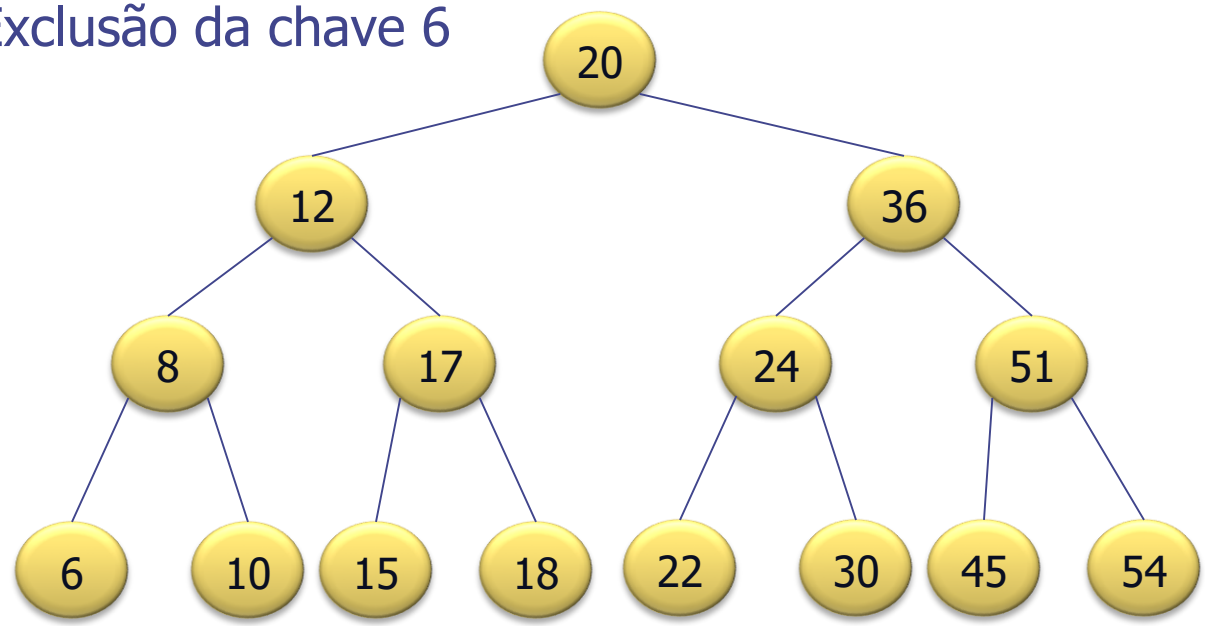
- **Exercício 5.** Desenhe uma árvore de pesquisa com a inserção dos elementos na seguinte ordem: 20 10 5 8 12 3 14 23 28 30 100 e 99.

- **Exercício 6.** Apresente o percurso pós-ordem para a árvore abaixo.



Exercícios

- **Exercício 7.** A partir da árvore abaixo, aplique as seguintes técnicas de exclusão (use a árvore resultante e não a original):
  a) Exclusão por fusão da chave 12
  b) Exclusão por cópia da chave 36
  c) Exclusão da chave 8
  d) Exclusão da chave 6



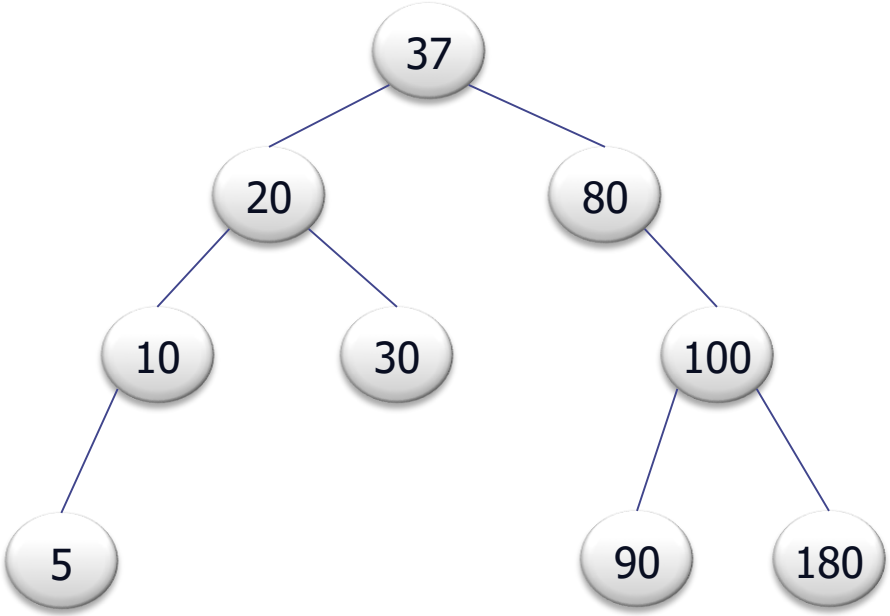
Exercícios

Exercício 8. A árvore abaixo é:

a) Estritamente binária? Justifique sua resposta.

b) Completa? Por quê? Justifique sua resposta.

c) Cheia? Por quê? Justifique sua resposta.



Referências Bibliográficas

- ASCENCIO, A. F. G; ARAÚJO, G. S. **Estruturas de Dados**. São Paulo: Pearson Prentice Hall, 2010. 432 p.

- BATISTA, C. A. T. **Estruturas de Dados**. Lâminas segundo semestre, 2009.

- CORMEN, Thomas H. et al. **Introduction to algorithms.** 3. ed. Cambridge: MIT, 2009. xix. 1292 p.

- SZWARCFITER, J.; MARKENZON, L. **Estruturas de Dados e seus Algoritmos**. Rio de Janeiro: LTC, 1994.