Curso de Engenharia da Computação

Material Disciplina Estrutura de dados – II

Conceito de Grafos e Árvores

Parte-III

Prof. Wagner Santos C. de Jesus

wsantoscj@gmail.com

Conceito de Estrutura de Grafos

Conceito Teórico (Matemático)

A teoria dos grafos é um ramo da matemática que estuda as relações entre os objetos de um determinado conjunto. Para tal são empregadas estruturas chamadas de grafos.

Conceito Prático

Um grafo vem a ser uma coleção de vértices ou nós, que são os elementos que contêm a informação que se pretende armazenar, e de arestas ou arcos, que são os elementos que ligam os vértices.

Motivação

Necessitam de considerar conjunto de conexões entre pares de objetos:

- Existe um caminho para ir de um objeto a outro seguindo as conexões?
- Qual é a menor distância entre um objeto e outro objeto?
- Quantos outros objetos podem ser alcançados a partir de um determinado objeto?

Aplicações

- Ajudar máquinas de busca a localizar informação relevante na Web.
- Descobrir os melhores casamentos entre posições disponíveis em empresas e pessoas que aplicaram para as posições de interesse.
- Descobrir qual é o roteiro mais curto para visitar as principais cidades de uma região turística.

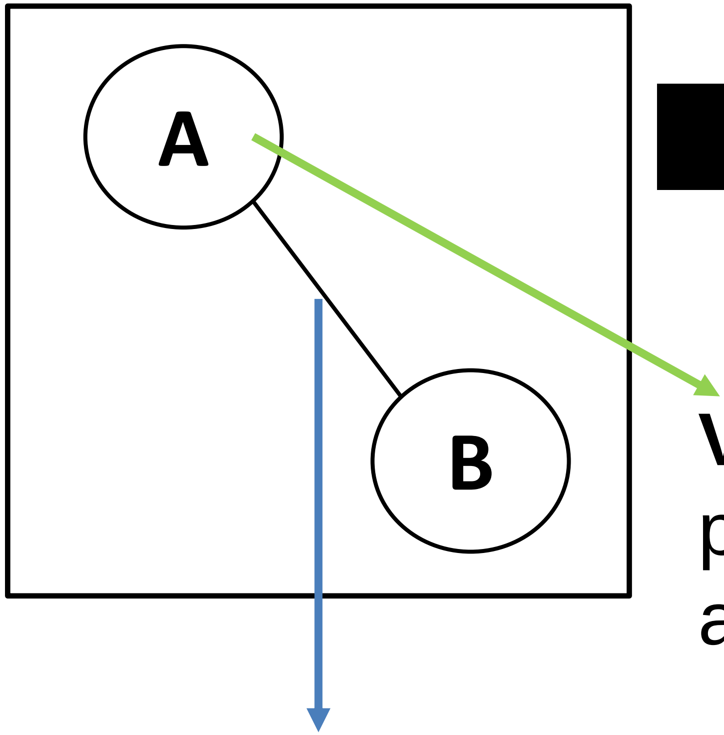
Áreas de Aplicação

- Engenharia;
- Computação;
- Matemática;
- Economia;
- Biologia;
- Física

Aplicando Conceitos de Grafos

Conceitos Básicos

Representação

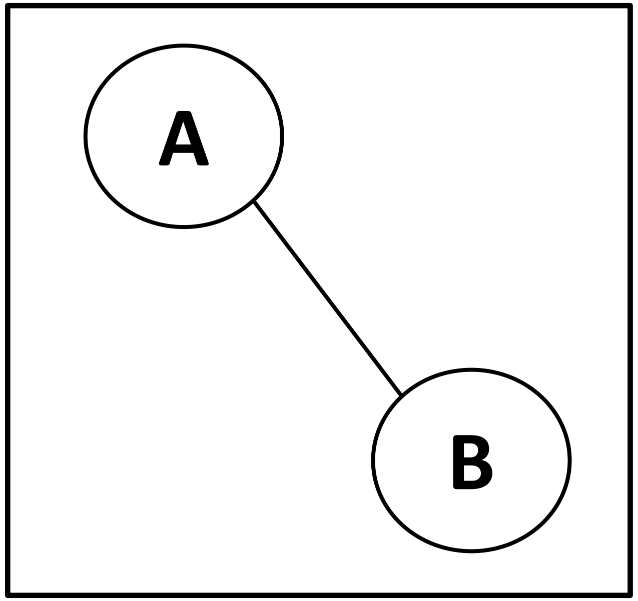


Grafo: Conjunto de vértices e arestas

Vértices: objeto simples que pode ter nome e outros atributos.

Aresta: conexão entre dois vértices.

Notação de um grafo



G = (V,A)

G = Grafo
V = Conjunto de vértices.
A = Conjunto de arestas.

Um grafo possui pelo menos um vértice, mas pode não ter qualquer aresta.

Grafos Direcionados

Um grafo direcionado G é um par (V,A), onde V é um conjunto finito de vértices e A é uma relação binária em V.

– Uma aresta (u, v) sai do vértice u e entra no vértice v. O vértice v é **adjacente** ao vértice u.

– Podem existir arestas de um vértice para ele mesmo, chamadas de *self-loops (Auto-loop)*.

Gafos não Direcionados

Um grafo não direcionado G é um par (V,A), onde o conjunto de arestas A é constituído de pares de vértices não ordenados.

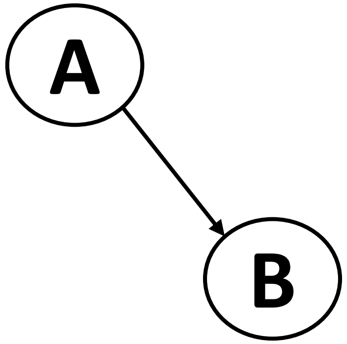
– As arestas (u, v) e (v, u) são consideradas como uma única aresta. A relação de adjacência é simétrica.

– *Self-loops* não são permitidos.

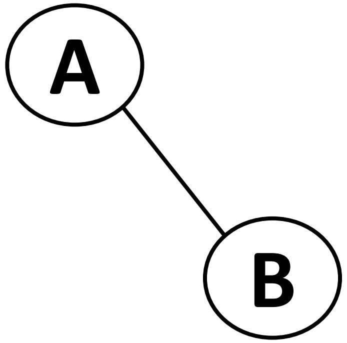
Exemplo Direcionamento

Um grafo:

- Direcionado (A)
- Não direcionado (B)



(A)



(B)

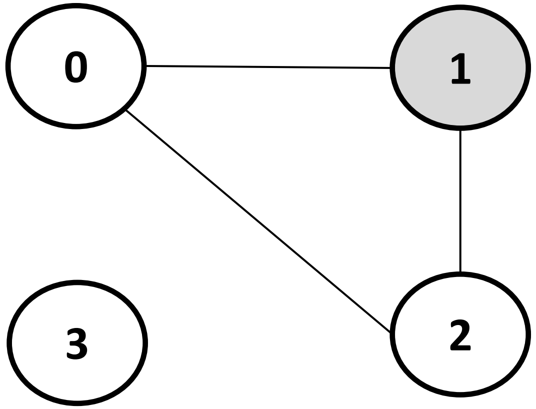
Grau de um Vértice

- Em grafos não direcionados:

O grau de um vértice é o número de arestas que incidem sobre o mesmo.

Um vértice de grau zero é dito isolado ou não conectado.

Exemplo:



Vértice 1 tem grau 2.

Vértice 3 é isolado ou não conectado.

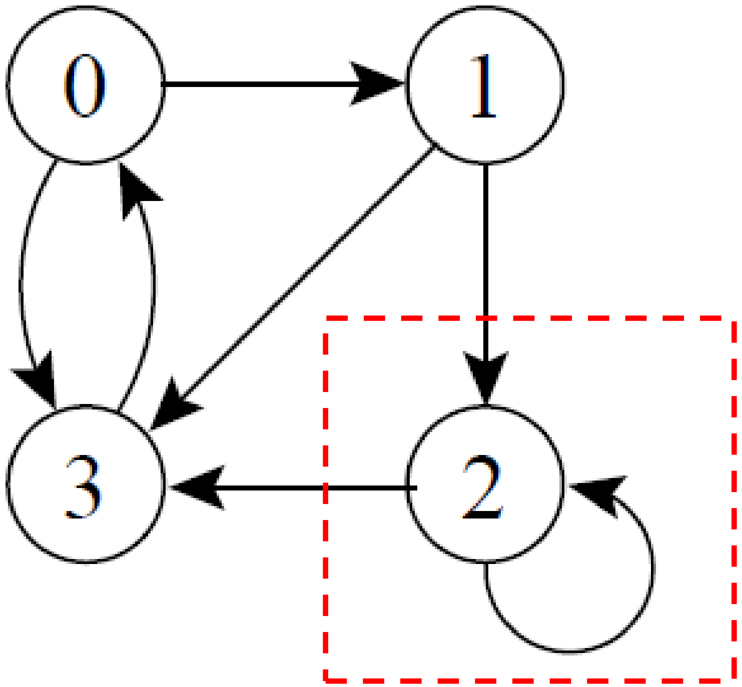
Grafo não direcionado.

Em grafos direcionados

O grau de um vértice é o número de arestas que saem dele (*out-degree*) mais o número de arestas que chegam nele (*in-degree*).

Exemplo grau de um grafo

G(V,A) = G(4,7)



No Vértice 2, o número de arestas que saem = 2, número de arestas que chegam = 2, grau = 4.

Caminho entre Vértices

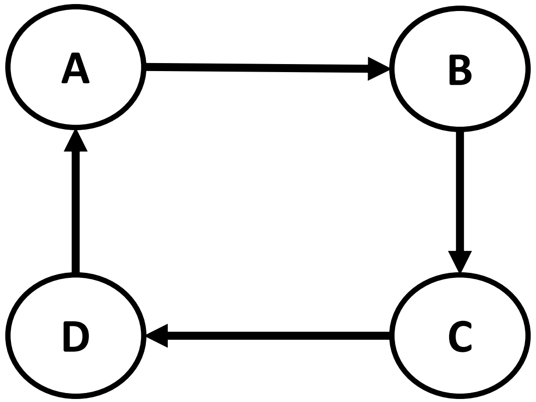
Caminho entre vértices

Um caminho de **comprimento** k de um vértice x a um vértice y em um grafo.

G = (V,A) é uma sequência de vértices (v_0, v_1, v_2, . . . , v_k) tal que x = v_0 e y = v_k, e (v_{i-1}, v_i) ∈ A para i = 1, 2, . . . , k.

Comprimento do caminho

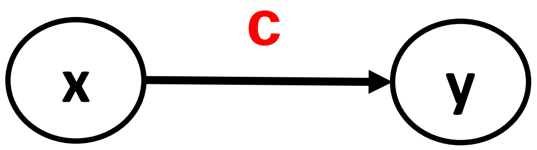
O comprimento de um caminho é o número de arestas nele contido.



G = (4,4) = c = 4

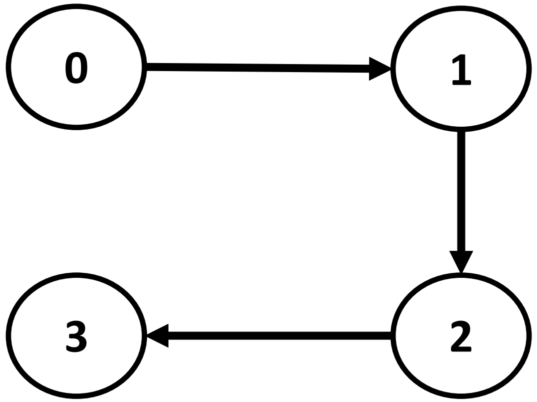
Caminho

Se existir um caminho c de x a y então y é **alcançável** a partir de x via c.



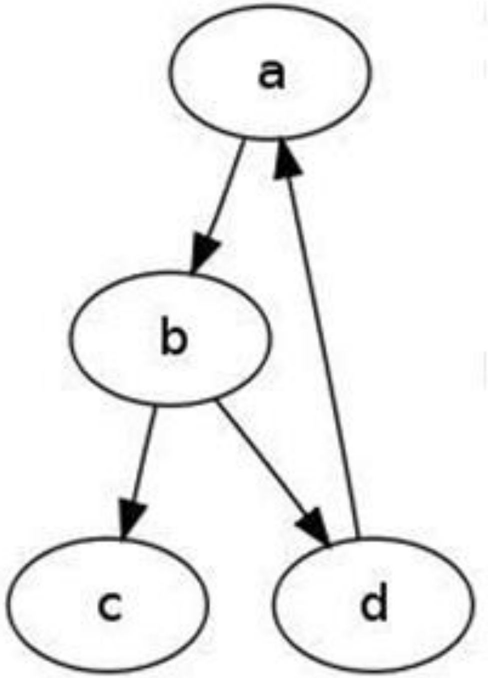
Caminho Simples

Um caminho é **simples** se todos os vértices do caminho são distintos.



Caminho c = (0,1,2,3) portanto o caminho é igual a 3.

Ciclos em Grafos



Ciclo grafo direcionado:

Um caminho (v0, v1, . . . , vk) forma um ciclo se v0 = vk e o caminho contém pelo menos uma aresta.

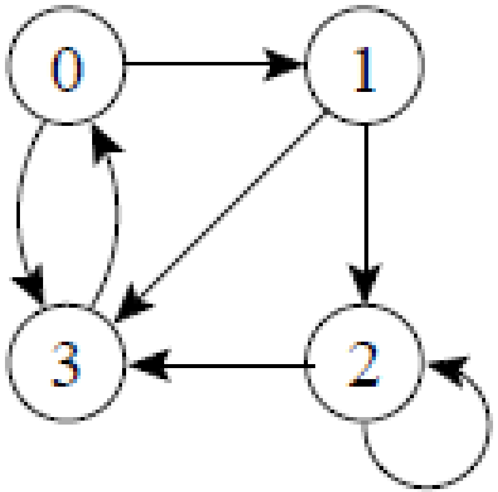
O ciclo é simples se os vértices v1, v2, . . . , vk são distintos.

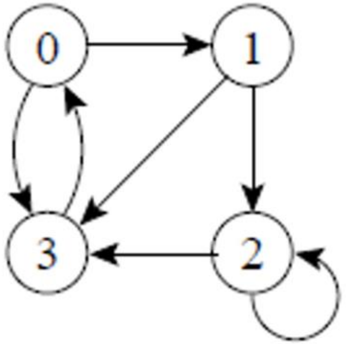
O *self-loop* é um ciclo de tamanho 1.

Dois caminhos (v0, v1, . . . , vk) e (v’0, v’1 , . . . , v’k) formam o mesmo ciclo se existir um inteiro j tal que v’i = v(i+j) mod k para i = 0, 1, . . . , k − 1.

Exemplo Ciclo

O caminho (0, 1, 2, 3, 0) forma um ciclo. O caminho(0, 1, 3, 0) forma o mesmo ciclo que os caminhos (1, 3, 0, 1) e (3, 0, 1, 3).





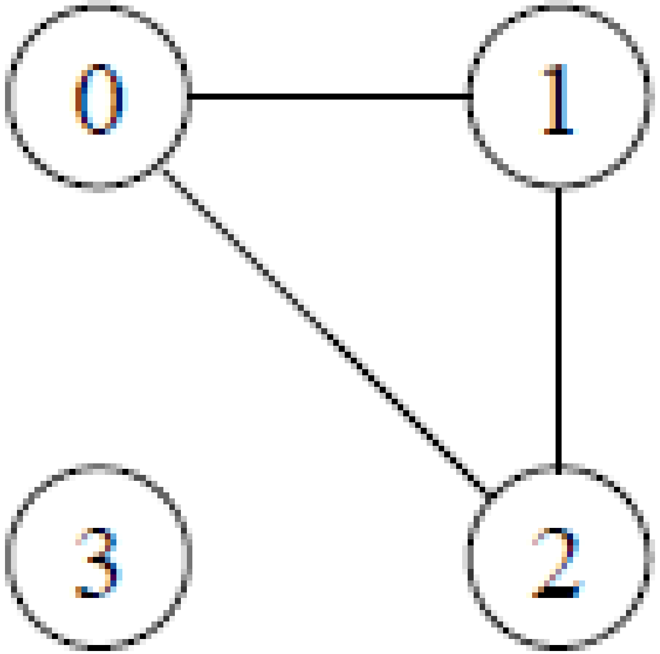
Ciclo grafo não direcionado:

Um caminho (v0, v1, . . . , vk) forma um ciclo

se v0 = vk e o caminho contém pelo menos três arestas.

O ciclo é simples se os vértices v1, v2, . . . , vk são distintos.

O Caminho (0,1,2,0) é ciclo:

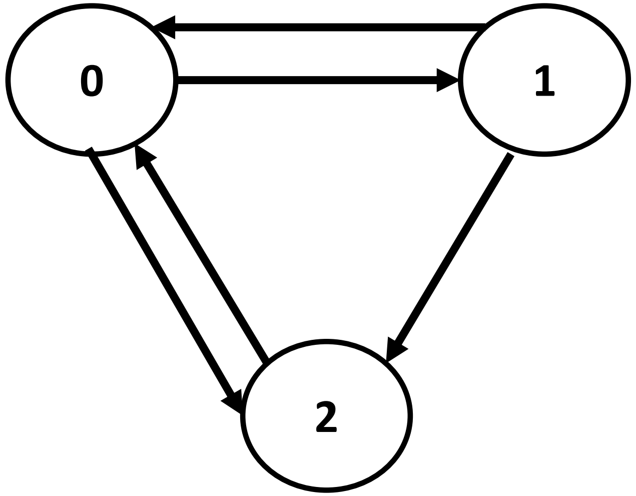


Implementação de Grafos Usando Matriz de Adjacência

Quando se aplica

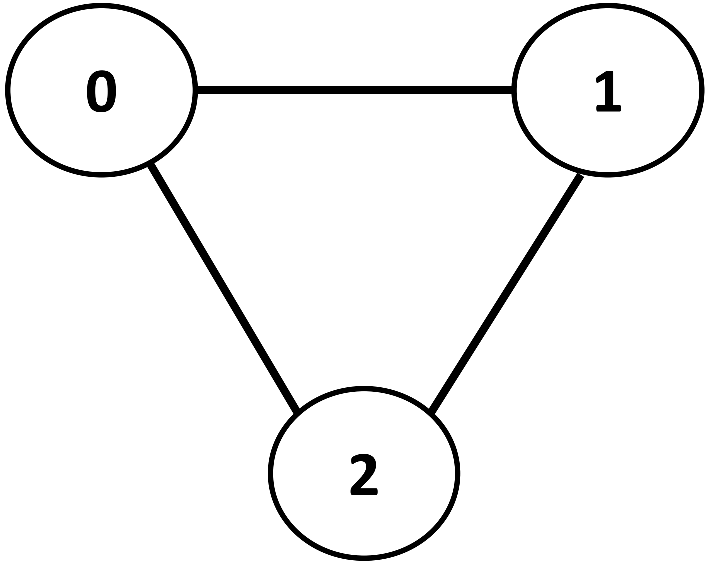
É muito útil para algoritmos em que necessitamos saber com rapidez se existe uma aresta ligando dois vértices.

Exemplo Prático (G) Direcional



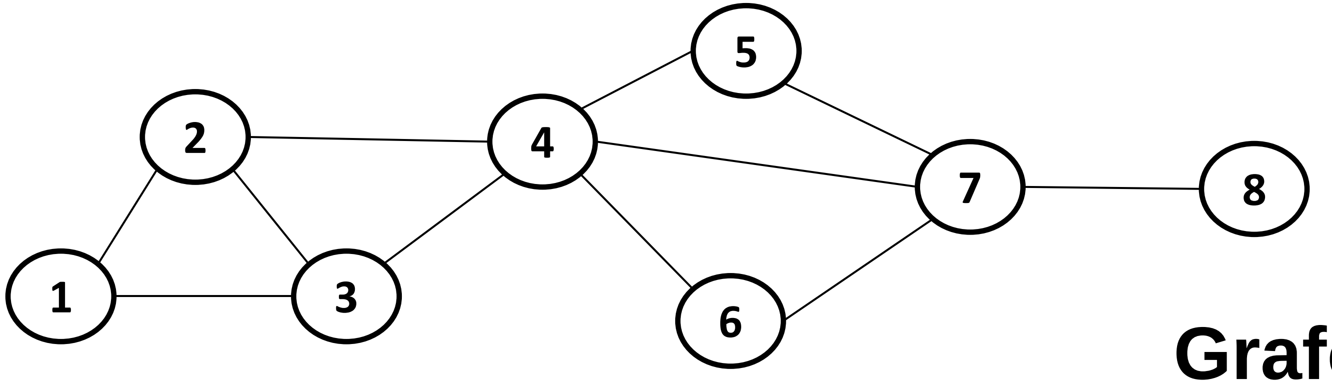
	0	1	2
0		1	1
1	1		1
2	1		

Exemplo Prático (G) não Direcional

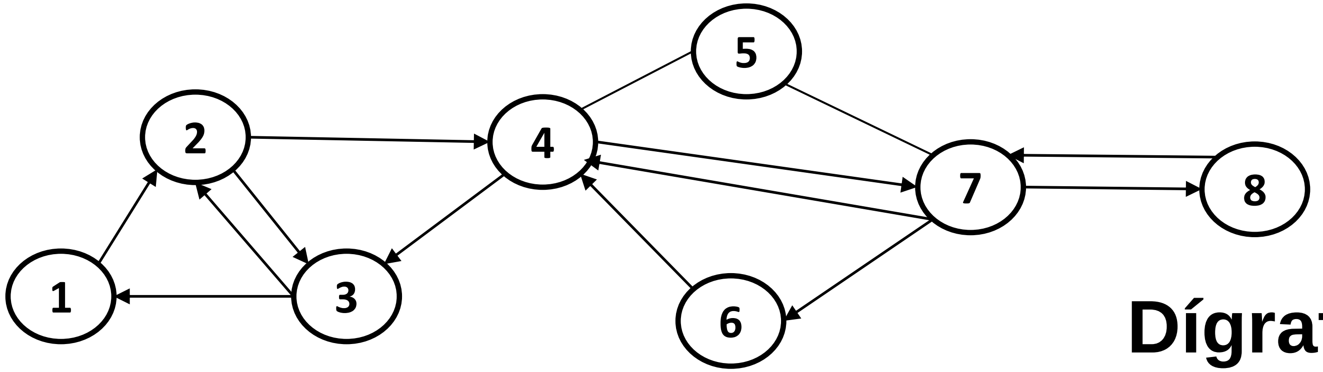


	0	1	2
0		1	1
1	1		1
2	1	1	

Orientação em um grafo



Grafo



Dígrafo

Conceito de Dígrafo

Denomina-se grafo orientado (directed graph) ou dígrafo (digraph) a um grafo em que as arestas são orientadas, ou seja, um grafo em que as arestas especificam o sentido da ligação entre os dois vértices adjacentes (Rocha,2011).

Operações Principais de um Grafo

1. Criar um grafo vazio.
2. **Inserir uma aresta no grafo.**
3. Verificar se existe determinada aresta no grafo.
4. Obter a lista de vértices adjacentes a determinado vértice.
5. **Retirar uma aresta do grafo.**
6. Imprimir um grafo.
7. Obter o número de vértices do grafo.

Implementação Prática de Grafo

Classes de Implementação do Grafo

Um grafo, é composto por Aresta que por sua vez são compostas por vértices e pesos.

Pesos, são valores respectivos as arestas.

Estrutura da classe grafo.

```
 publico classe GrafoAd {
}
 publico classe Aresta {
                ......
}
```

Implementação da classe Aresta

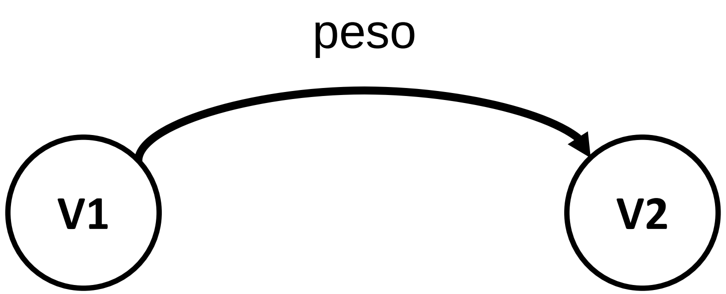
```
publico  classe Aresta {
          privado inteiro v1, v2, peso;
          publico Aresta ( inteiro v1, inteiro v2, inteiro peso) {
             this.v1 = v1;
             this.v2 = v2;
             this.peso = peso;
           }
           publico inteiro peso() {
            retorno this .peso;
           }
           publico inteiro v1() {
             retorno this.v1;
           }
           publico inteiro v2() {
            retorno this.v2;
           }
      }
```

Implementação do Construtor Grafo

```
privado inteiro mat [][] ; // pesos do tipo inteiro
privado inteiro numVertices;
privado inteiro pos[] ; // posição atual ao se percorrer os adjs de um vértice v
publico GrafoAd ( inteiro numVertices) {
      this.mat = new inteiro [numVertices][numVertices] ;
      this.pos = new inteiro [numVertices] ;
      this.numVertices = numVertices;
     para ( int i = 0; i < this .numVertices; i ++)  faca
       para ( int j = 0; j < this .numVertices; j ++)  faca
           this.mat[i][j] = 0;
       fim-para
           this.pos[i] = -1;
     fim-para
}
```

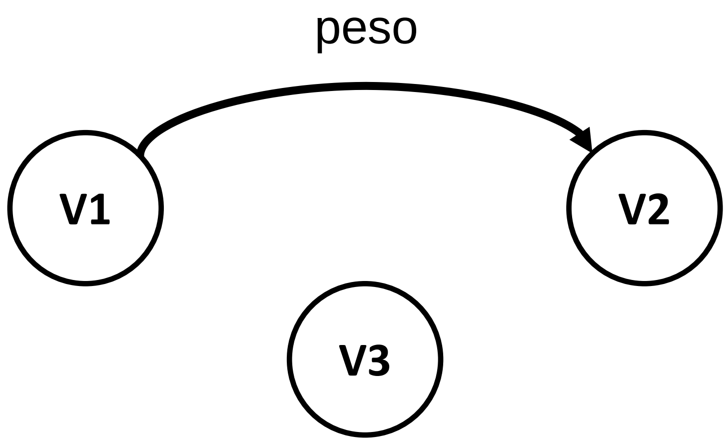
Insere uma aresta no grafo

```
publico insereAresta ( inteiro v1, inteiro v2, inteiro peso)    {
            this.mat[v1][v2] = peso;
  }
```



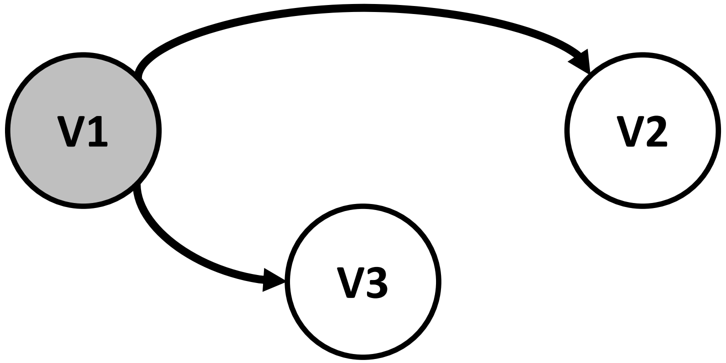
Verificar arestas existentes

```
publico booleano existeAresta ( inteiro v1, inteiro v2) {
        retorna (this.mat[v1][v2] > 0);
}
```



Verifica se existe adjacência em um dado vértice

```
publico booleano listaAdjVazia ( inteiro v ) {
        para ( int i =0; i < this.numVertices; i++) faca
          if ( this .mat[v][i] > 0)
            retorna false;
        fim-para
        retorna true;
  }
```



Primeira aresta do vértice.

```
publico Aresta primeiroListaAdj (inteiro v) {
        this.pos[v] = -1;
        retorna this.proxAdj(v);
}
```

Retorna a primeira aresta que o vértice v participa ou null se a lista de adjacência de v for vazia.

Próxima aresta que v participa

```
publico Aresta proxAdj (inteiro v) {
     // Retorna a próxima aresta que o vértice v participa ou
     // null se a lista de adjacência de v estiver no fim
     this.pos[v]++;
   enquanto (( this .pos[v] < this.numVertices) && (this.mat[v][this.pos[v]] == 0) ) faca
          this.pos[v]++;
    fim-enquanto
     se (this.pos[v] == this.numVertices) entao
          retorna null ;
     senão
          retorna new Aresta(v,this.pos[v],this.mat[v][this.pos[v]]) ;
      fim-se
    }
```

Remove uma aresta

```
publico Aresta retiraAresta ( inteiro v1, inteiro v2) {
        se ( this.mat[v1][v2] == 0)  então
            retorna null ; // Aresta não existe
        senão
            Aresta aresta = new Aresta (v1,v2, this.mat[v1][v2] ) ;
            this.mat[v1][v2] = 0;
            retorna aresta;
        fim-se
   }
```

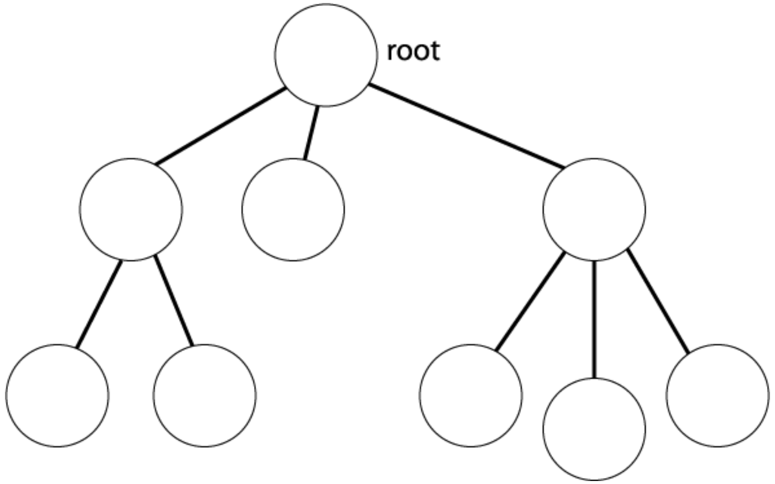
Mostra dados do grafo

```
publico imprime() {
        para ( int i = 0; i < this.numVertices; i ++) faca
            para ( int j = 0; j < this.numVertices; j++) faca
                System.out.print ( this.mat[i][j]+" " ) ;
            fim-para
                System.out.println() ;
        fim-para
}
```

Conceito de Estrutura de Árvore

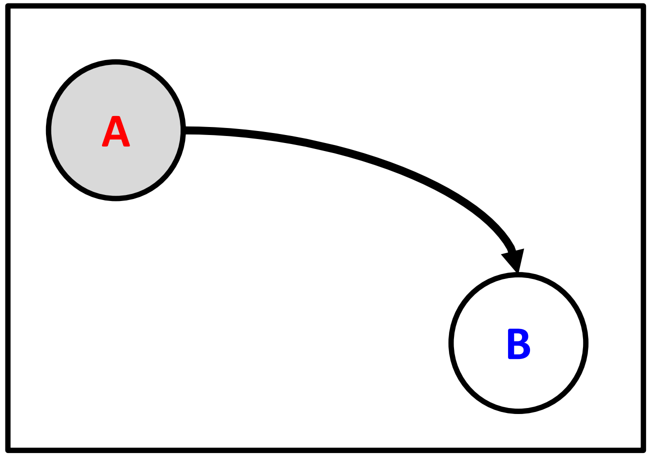
Conceito de Árvore (Genérico)

Uma árvore é uma abstração matemática que serve para especificar relações, descrever organizações e armazenar informação, que se apresenta estruturada de forma hierárquica.



Conceito de Árvore (Específico)

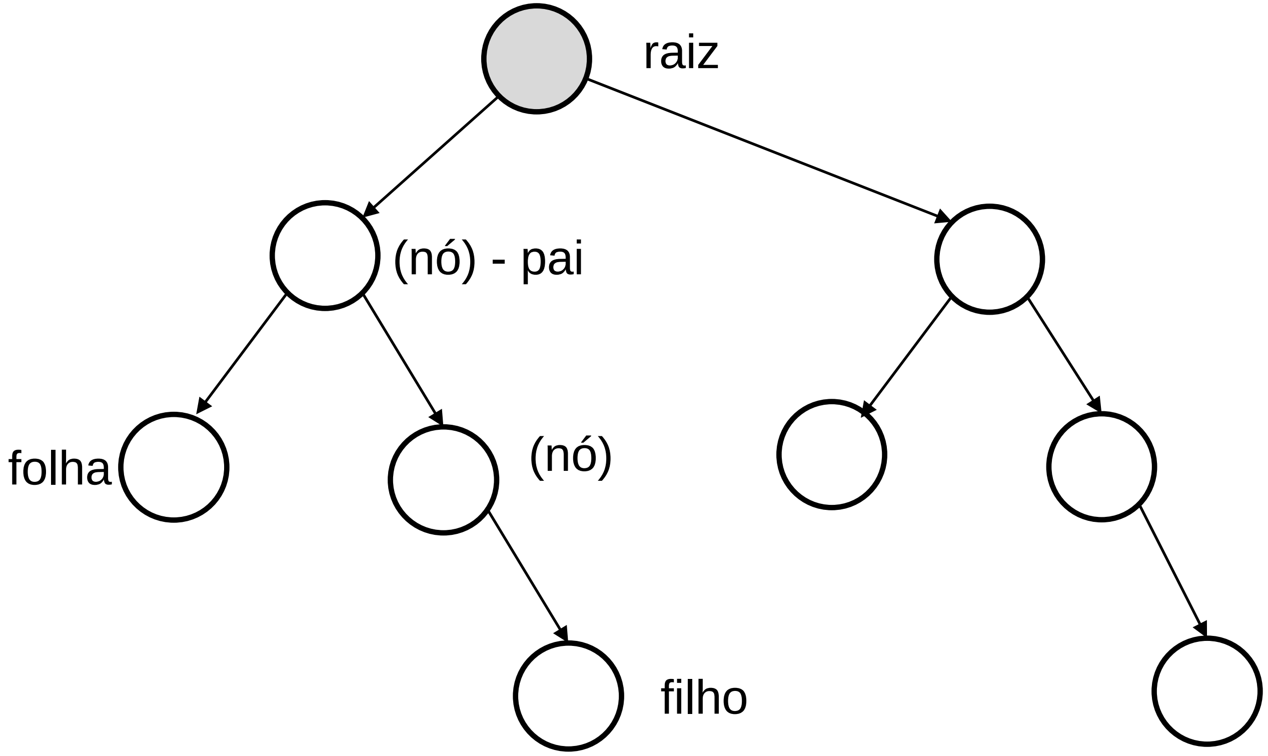
Uma árvore é uma coleção de nós ou vértices, que são os elementos que contêm a informação que se pretende armazenar, e de arcos ou arestas, que são os elementos que ligam os nós.



Importante

Em uma árvore só pode existir um caminho entre dois nós. Se existir um par de nós em que não exista caminho ou exista mais de um caminho entre eles, então estamos perante um grafo e não uma árvore.

Composição de uma árvore

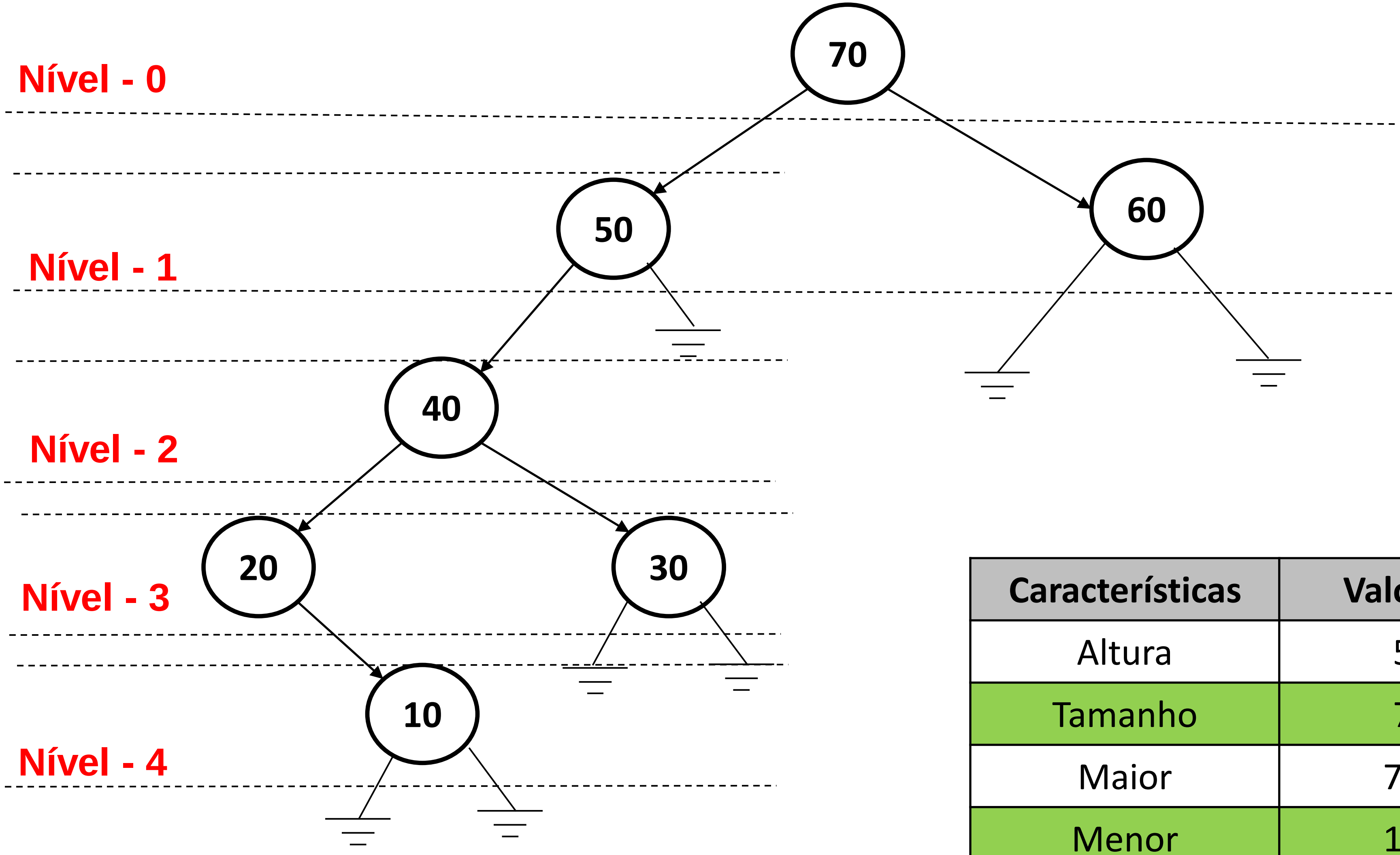


Características de uma Árvore

Uma árvore é caracterizada pelo seu tamanho e altura. Onde:

Característica	Descrição
Tamanho	Número de vértices a esquerda e número de vértices a direita; (Nesq + Ndir) + 1
Altura	Número máximo de vértices a esquerda ou a direita + 1, Max(Nesq, Ndir) + 1.
Máximo	Maior elemento da árvore.
Mínimo	Menor elemento da árvore.
Árvore(Vazia)	Uma árvore vazia possui altura zero (0).

Exemplo de Árvore Binária



Características	Valores
Altura	5
Tamanho	7
Maior	70
Menor	10

v = {20,60,40,10,30,50,70,60}

Árvore Binária de Pesquisa

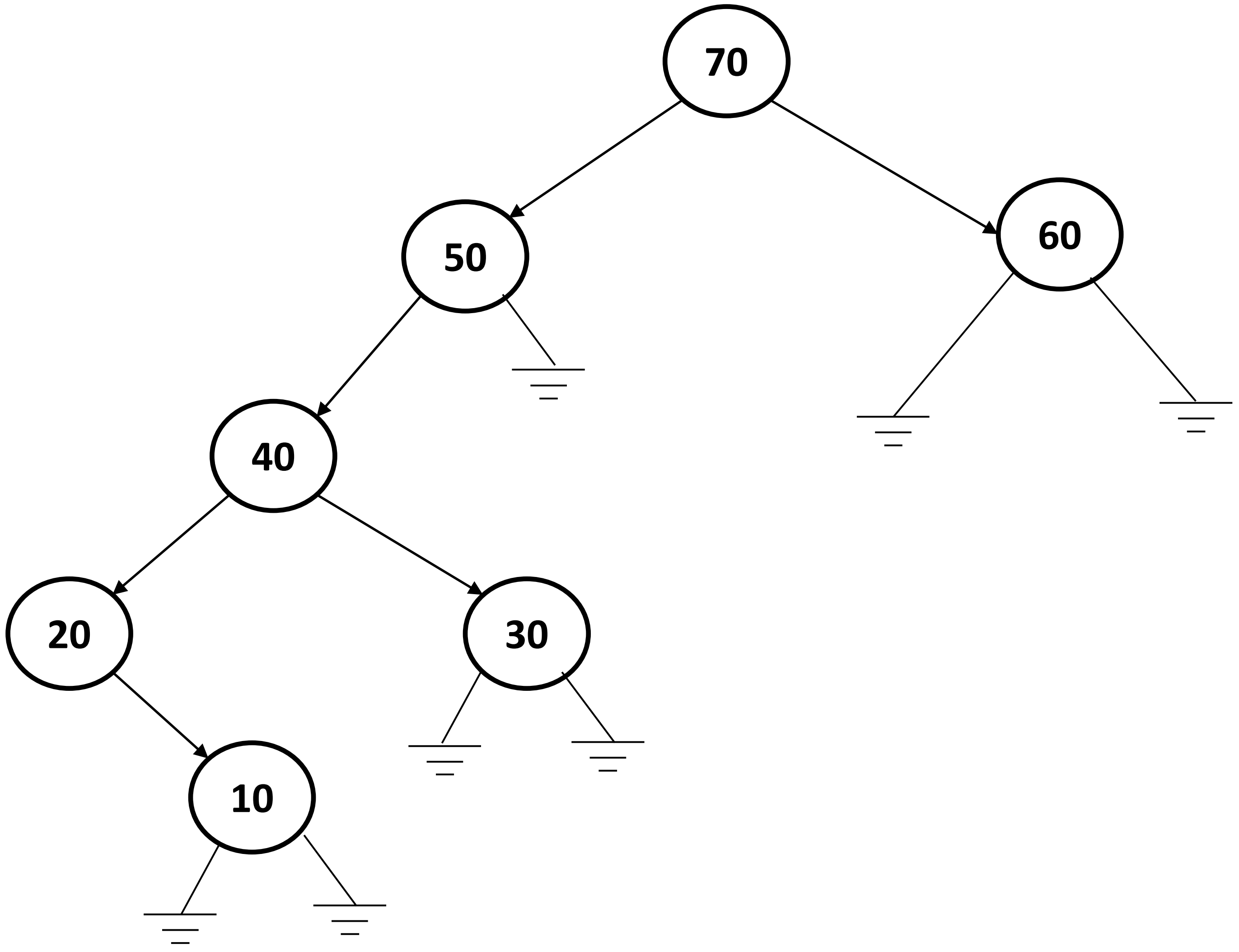
(Binary Search Tree)

BST

Conceito de Arvore Binária

Uma árvore binária de pesquisa (BST) ordenada por ordem crescente é uma árvore binária em que os nós têm associada uma chave, que determina a sua posição de colocação na árvore e que obedece à seguinte regra: *a chave de um nó é maior do que as chaves dos nós da sua subárvore direita*. Desta definição decorre que não podem existir nós com chave repetida.

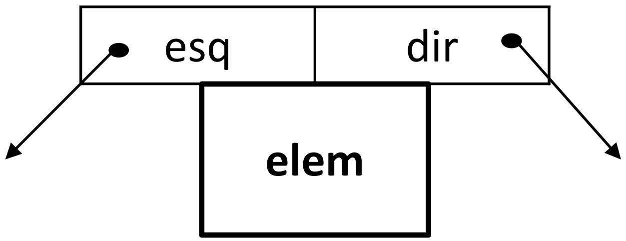
Demonstração do Algoritmo de Árvore Binária



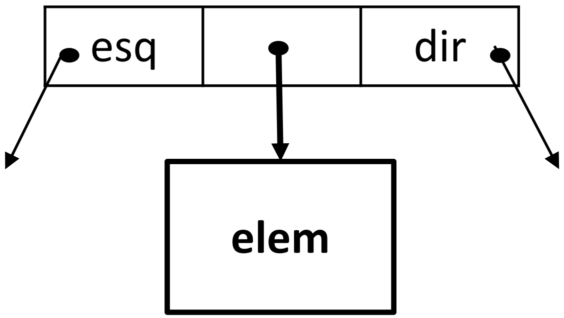
Operações principais de Estrutura de Árvore

1. Verificar se a árvore está vazia;
2. Verificar altura e tamanho da árvore;
3. **Inserir elemento na árvore;**
4. **Remover elemento da árvore;**
5. Encontrar um dado elemento na árvore;
6. Encontrar o maior e o menor elemento da árvore.

Tipos de pesquisa em Árvore Binária



Estrutura de dados compacta.

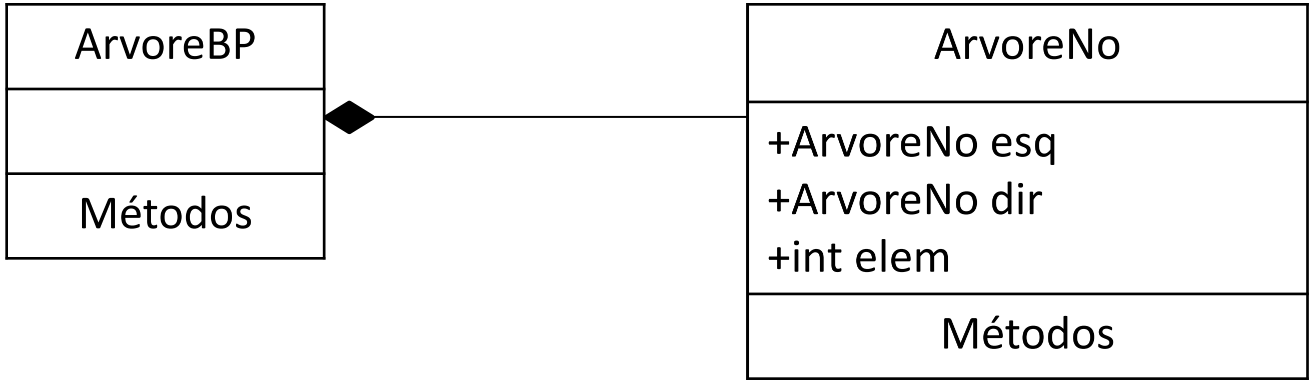


Estrutura de dados que permite construir uma árvore genérica.

+ArvoreNo esq
+ArvoreNo dir
+tipo elem

Implementação de Estrutura de Arvore de Pesquisa Binária

Classes de Implementação do Grafo



Uma Árvore é composta pela raiz e seus respectivos nós.

Os nós podem estar localizados a esquerda ou direita, contendo uma estrutura de dados que se refere ao seu elemento (Vértice).

Classe Arvore Binária

```
publico classe ArvoreBP {
    <Métodos de operação da classe>
}
```

Classe principal de manipulação da árvore.

Implementação da classe nó

```
privado statico classe ArvoreNo  {
        publico ArvoreNo esq
        publico ArvoreNo dir
        publico inteiro elem
        publico ArvoreNo(inteiro pVal){
            elem = pVal
            esq = null
            dir = null
        }
    }
```

Composição da classe ArvoreBP com a classe ArvoreNo

```
publico classe ArvoreBP {
    privado statico class ArvoreNo  {
        publico ArvoreNo esq
        publico ArvoreNo dir
        publico inteiro elem
        publico ArvoreNo(inteiro pVal){
            elem = pVal
            esq = null
            dir = null
        }
    }
    publico ArvoreNo raiz
    publico ArvoreBP() {
        raiz = null
    }
    publico booleano Vazio(){
        retorna raiz == null
    }
```

.... Métodos público e privados de operação da árvore.

Inserção de Elementos na Árvore

```
publico vazio inserir(inteiro pElem) {
        se(raiz == null) então
            raiz = new ArvoreNo(pElem)
            return
        fim-se
        ArvoreNo node = raiz
        ArvoreNo prev = null
        enquanto(node != null)  faca
            prev = node
            se(node.elem > pElem) então
              node = node.esq
            senão
             se(node.elem < pElem)  então
               node = node.dir
             senão
               return;
             fim-se
           fim-se
        fim-enquanto
        se (prev.elem > pElem) então
               prev.esq = new ArvoreNo(pElem)
           senão
               prev.dir = new ArvoreNo(pElem)
} // Encerra método inserir
```

Pesquisar se existe um elemento na árvore

```
publico booleano pesquisa(inteiro pElem) {
        ArvoreNo node = raiz
        enquanto (node != null && node.elem != pElem) faca
           se (node.elem > pElem) então
             node = node.esq
           senão
             node = node.dir
           fim-se
        fim-enquanto
        retorno node != null  // Resultado da pesquisa
    }
```

Mostra dados da Árvore método toString()

```
publico String toString(){
        se(raiz == null) então
            retorna "Arvore vazia\n"
        senão
            retorna recString(raiz,1)
        fim-se
    }
```

Implementação do método recString()

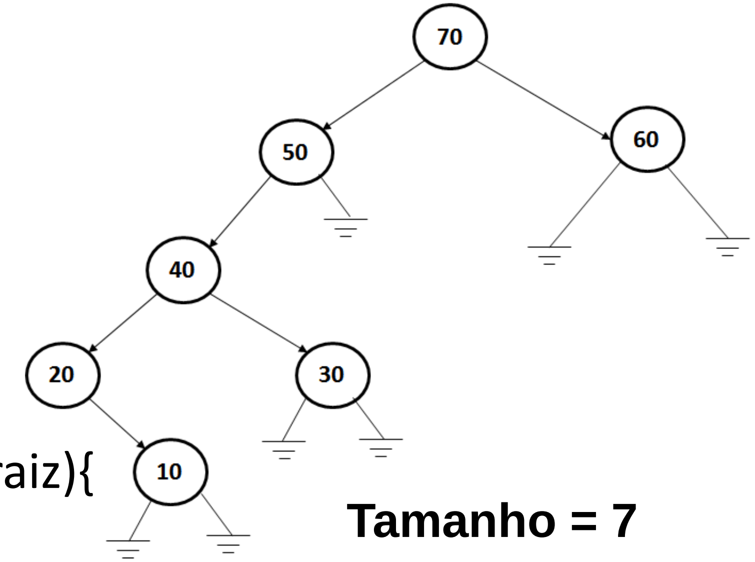
```
privado statico String recString(ArvoreNo pRaiz, inteiro paltura) {
        String str = ""
        se(pRaiz==null) então
           para (int i=0;i < paltura;i++) faça
              str += "\t"
           fim-para
           str += "*\n"
           return str
        fim-se
        str += recString(pRaiz.dir, paltura+1);
        para(int i=0;i < paltura; i++) faça
           str += "\t"
        fim-para
          str += pRaiz.elem+"\n"
          str += recString(pRaiz.esq, paltura+1)

       return str
   }
```

Retorna o tamanho da Árvore

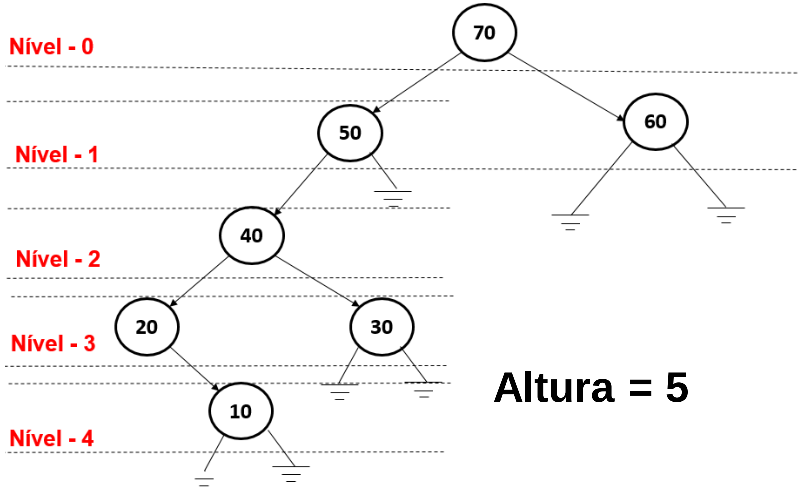
```
publico inteiro tamanho(){
      retorno rectamanho(raiz)
   }

   privado statico inteiro rectamanho(ArvoreNo praiz){
        se(praiz == null) então
          retorno 0;
        senao
          retorno 1 + rectamanho(praiz.esq) + rectamanho(praiz.dir);
         fim-se
   }
```



Tamanho = 7

Retorna a Altura




```
publico inteiro altura() {
    retorno recaltura(raiz);
}

privado statico inteiro recaltura(ArvoreNo praiz){
    se(praiz == null) então
       return 0;
    senão
       return 1 + Math.max(recaltura(praiz.esq),recaltura(praiz.dir));
    fim-se
}
```

Encontra mínimo valor da Árvore

```
publico inteiro getMin(){
        ArvoreNo node = raiz;
        se (node == null) então
            return 0;
         fim-se
        enquanto (node.esq != null) faça
            node = node.esq;
         fim-enquanto
        return node.elem;
    }
```

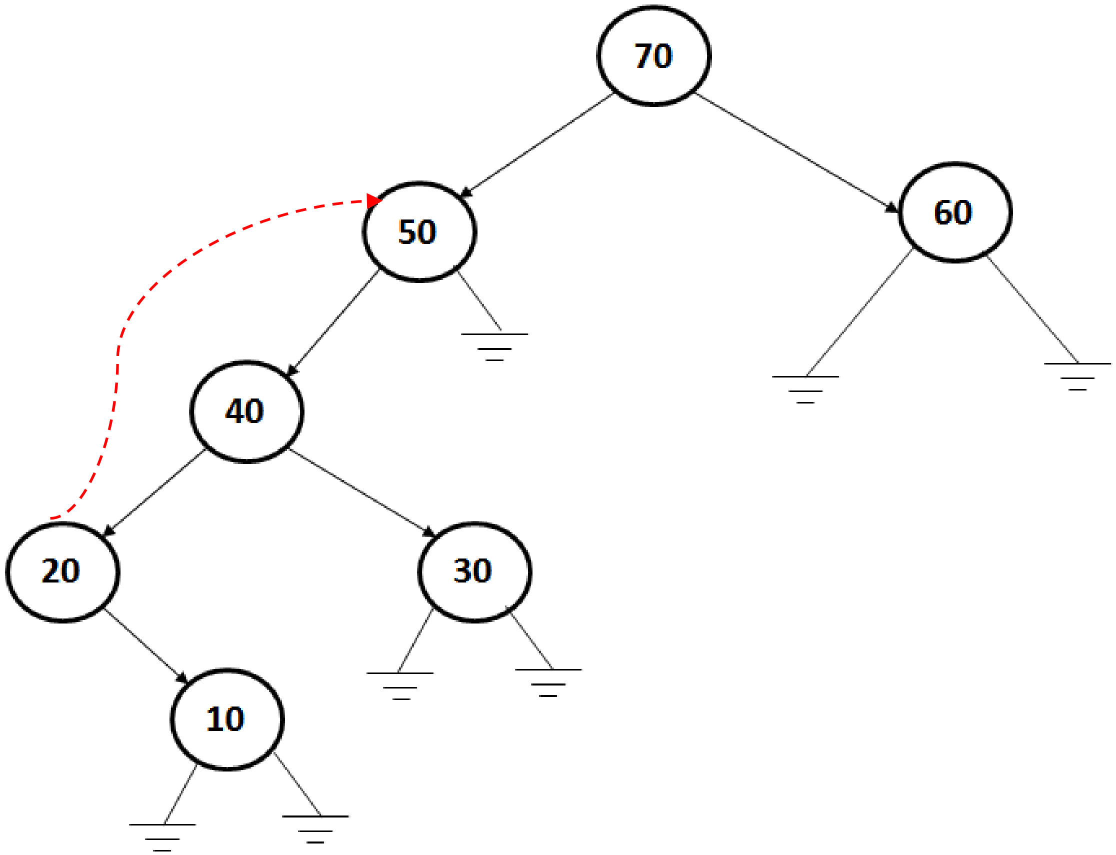
Remoção de dados em Árvore Binária

Tipos de remoção

Nó folha sem filhos.

Nó com um filho.

Nó com dois filhos.


70
50
60
40
20
30
10

Encontra o elemento da remoção

```
publico remover(inteiro pElem) {
        se(raiz == null) então
          return;
        fim-se
        ArvoreNo delnode = raiz;
        ArvoreNo prev = null;
        enquanto (delnode != null && delnode.elem != pElem) faça
            prev = delnode;
            se (delnode.elem > pElem) então
                delnode = delnode.esq;
            senão
                delnode = delnode.dir;
           fim-se
        fim-enquanto
```

Verificar elemento inexistente na árvore

```
se(delnode == null)  então
      retorna
 ArvoreNo node = delnode
  se (node.dir == null) então
      node = node.esq  // liga a esquerda
    else
         .........
```

Ajusta os elementos da árvore

```
se (node.esq == null) então
     node = node.dir // ligação a direita
 senão
    ArvoreNo tmp = node.dir   // Procura o menor elemento da árvore direita.
    ArvoreNo prevtmp = node
    enquanto (tmp.esq != null) então
         prevtmp = tmp
         tmp = tmp.esq
    fim-enquanto
    node.elem = tmp.elem  // Substitui pelo menor elemento
    se (prevtmp == node)  então
       prevtmp.dir = tmp.dir     // Desliga o menor elemento e menor elemento e ajustas as ligações
    senão
       prevtmp.esq = tmp.dir
   fim-se
    return
  fim-se
```

Ajuste do nó pai da árvore

```
  se (delnode==raiz) então
      raiz = node;
senão
      se(prev.esq == delnode) então
        prev.esq = node;
      senão
        prev.dir = node;
       fim-se
  fim-se
```

Ajusta o nó pai do elemento removido que só tem um filho, caso tenha sido removido da raiz.