**Arvores binarias - Java**

**Observações**

* Toda arvore binaria possui 1 ou dois filhos pôr no.
* Valores menores que a raiz ficam à esquerda e valores maiores que a raiz ficam à direita.
* Arvores binarias são arvores ordenadas.

Para criar uma estrutura tipo arvore binaria usamos duas classes básicas: classe No e Classe Arvore. Onde a classe Nó serve como ponteiro, ou seja, é o objeto que guarda as informações desejamos salvar na arvore assim como referências para os próximos nós. A classe arvore contém os métodos necessário para manipular a estrutura, tais como: adicionar, remover ou listar...

**Classe Nó**

Normalmente usamos uma classe genérica para este tipo de estrutura assim como os atributos mínimos. Esta classe é, normalmente, manipulada unicamente pela classe arvore que a usa como modelo para criação de suas folhas (nos).

Neste exemplo o construtor recebe o valor do T, este valor é o conteúdo que desejamos salvar na estrutura. Nele também atribuímos ***null*** aos elementos da esquerda e da direita, isso garante a primeira instancia sempre será ***null,*** e o primeiro elemento será a raiz da arvore.



**Classe Arvore**

Esta classe implementa os métodos de manipulação da arvore como: adicionar, deletar e listar. O tipo desta classe deve estender a classe Comparable.

**Adicionar**. A assinatura do método adicionar é ***void*** e recebe como parâmetro um elemento do tipo T. Este método segue o modelo tradicional para adicionar um elemento a um objeto, onde:

Primeiro cria-se um objeto a ser adicionado: ***Elemento<T> novoElemento = new Elemento<T>(valor);***

Neste ponto verifica-se a raiz está vazia, se ela estiver então o ***novoElemento*** que criamos, no início do método, passa ser a raiz ou seja ele vira o primeiro elemento da arvore. ***if (raiz == null) {this.raiz = novoElemento;}***

Se a raiz não estiver vazia(null) o ***novoElemento*** que queremos adicionar a arvore deve caminhar entre esquerda e direita fazendo comparações a fim de descobri sua posição. Se ele for menor que o elemento comparado logo ele ficara na esquerda se ele for maior que o elemento comparado ele ficara na direita.

package br.com.projeto.arvoreEstudo2;  
  
public class Arvore<T extends Comparable> {  
  
 private Elemento<T> raiz;  
  
 public Arvore() {  
 this.raiz = null;  
 }  
  
 public void adicionar(T valor) {  
  
 Elemento<T> novoElemento = new Elemento<T>(valor);  
  
 //Primeiro verifica-se a raiz esta nula  
 if (raiz == null) {  
 //se a raiz estiver vazia o novo elemento sera a raiz da arvore.  
 this.raiz = novoElemento;  
  
 } else {  
  
 //o elemento atual recebe a raiz.  
 Elemento<T> atual = this.raiz;  
  
 //Repetiçao para caminhar entre todos os nos.  
 while (true) {  
  
 /\*CompareTo devolve 3 possiveis resultado :  
 -1 , o valor é menor.  
 0 , o valor é igual.  
 1 , o valor é maior.\*/  
  
 //Se o novo elemento for menor que valor atual ele vai para esquerda  
 if (novoElemento.getValor().compareTo(atual.getValor()) == -1) {  
  
 //Verifica-se o no atual tem filhos a esquerda  
 if (atual.getEsquerda() != null) {  
 atual = atual.getEsquerda();  
  
 } else {  
 /\*Se o atual nao tiver filhos a esquerda logo  
 nao existe mais caminhos a visitar. Entao  
 o novo elemento é adicionado como filho deste Nó.  
 \*/  
 atual.setEsquerda(novoElemento);  
 //Para o while pois o novo elemento ja foi adicionado  
 break;  
 }  
  
 //Se o novo elemento for maior que o valor atual ele vai para direita  
 } else {  
 //maior ou igual, se  
 if (atual.getDireita() != null) {  
 /\*se tiver alguem na direita o codigo roda novamente do inicio  
 ate chegar aqui novamente ate nao encontrar ninguem na direita.  
 quando nao tiver ninguem na direita logo essa posiçao fica para o novo  
 elemento\*/  
 atual = atual.getDireita();  
  
 } else {  
  
 atual.setDireita(novoElemento);  
 break;  
 }  
 }  
 }  
 }  
 }  
  
 public Elemento<T> getRaiz() {  
 return raiz;  
 }  
  
  
 //Usando recursividade  
 public void emOrdem(Elemento<T> atual) {  
  
 if (atual != null) {  
 emOrdem(atual.getEsquerda());  
 System.*out*.println(atual.getValor());  
 emOrdem(atual.getDireita());  
 }  
  
 }  
  
 public void preOrdem(Elemento<T> atual) {  
  
 if (atual != null) {  
 System.*out*.println(atual.getValor());  
 preOrdem(atual.getEsquerda());  
 preOrdem(atual.getDireita());  
 }  
  
 }  
  
 public void posOrdem(Elemento<T> atual) {  
  
 if (atual != null) {  
 posOrdem(atual.getEsquerda());  
 posOrdem(atual.getDireita());  
 System.*out*.println(atual.getValor());  
 }  
  
 }  
  
}