Árvores binárias de pesquisa
Prof. João B. Oliveira

Para estes exercícios, use a classe fornecida na página da disciplina (BinaryTree.java), bem como o programa que a testa (BTest.java). Em seguida cumpra as seguintes missões:

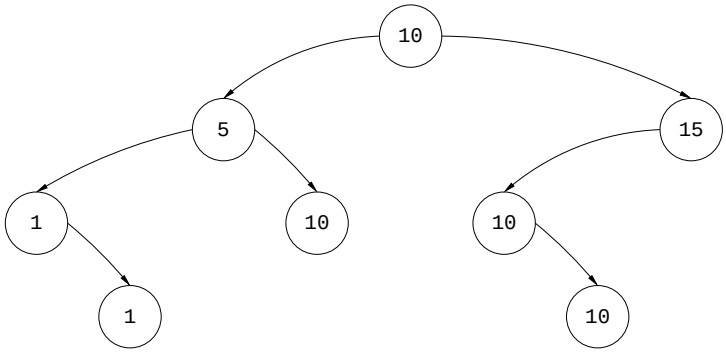
1. Crie um método de inserção de elementos na árvore, respeitando o critério de ordenação.
2. Altere BTest.java para ler de um arquivo os números que serão inseridos na árvore. Teste com o arquivo t1 fornecido na página da disciplina, que contém mil inteiros que devem ser inseridos em uma ABP.
3. Escreva um método **print()** que imprime a árvore e teste-o com a árvore t0.
4. Crie o método **exist(int n)**, que serve para identificar se um inteiro **n** está na árvore, retornando um booleano. Depois descubra qual das árvores t1, t2 ou t3 contém um 7.
5. Escreva um método **altura()** que retorna um inteiro informando a altura da árvore. Descubra qual das árvores t1, t2 ou t3 é mais alta.
6. Adapte o método **altura()** e crie um método **numnodos()** que conta quantos nodos estão na árvore.
7. Crie um método **int pai(int n)** que retorna o valor do pai de **n** ou -1 se **n** estiver na raiz ou se não estiver na árvore. Teste seu método com a árvore t0.
8. Crie um método **int soma()** que retorna a soma de todos os números que estão na árvore.
9. Suponha que a árvore tem valores repetidos. Crie um método **int conta(int val)** que retorna a quantidade de vezes que **val** está na árvore.
10. Crie um método **int numpares()** que examina a árvore e conta quantos números pares existem nela.
11. Crie um método **int filhosdir()** que examina a árvore e conta quantos nodos existem com filhos à direita.
12. Baseie-se em **print()** outra vez para fazer a impressão **vertical** de uma árvore com cada nodo sendo impresso em uma linha. Por exemplo, a árvore dada no arquivo t0 deve sair como

```
            18
        15
            13
    12
                11
            10
                9
        6
            4
```

13. Baseando-se em **print()**, crie um método **emordem()** que imprime em ordem crescente os valores que estão na árvore.
14. Baseando-se em **print()**, crie um método **emordemreversa()** que imprime em ordem decrescente os valores que estão na árvore.
15. Escreva um algoritmo **oɹlǝqsǝ()**, que recebe uma árvore binárias de pesquisa e espelha a árvore, ou seja, para cada nodo os valores menores estarão agora à sua direita e os maiores à esquerda. Imprima para confirmar.

16. Escreva um algoritmo **copy(B)**, que recebe uma árvore binária de pesquisa **B**, apaga tudo o que estiver na árvore atual e faz uma cópia da árvore **B** na atual, criando uma árvore exatamente igual a **B**. Imprima as duas para confirmar que são iguais.

17. Crie um método **boolean hasrep()** que retorna um booleano informando se há ou não há valores repetidos na árvore. Lembre que valores repetidos podem estar em locais inesperados, como no exemplo abaixo:



18. Escreva um algoritmo **join(B, C)**, que recebe duas árvores binárias de pesquisa **B** e **C**, apaga o que existe na árvore atual e depois monta uma nova árvore de pesquisa com os nodos de **B** e **C**. Não coloque nodos repetidos na árvore atual, nem apague as árvores **B** e **C** originais.

19. Crie um método **int ancestral(int i, int j)** que acha o ancestral comum mais próximo dos nodos contendo os valores **i** e **j**. Seu método não deve usar nenhuma lista ou estrutura auxiliar.

20. Crie um método **caminho(int val)** que imprime o caminho da raiz até o nodo que contém o valor dado, ou não imprime nada se o valor não estiver na árvore.

21. Crie um método **int maxsoma()** que acha o valor do caminho com a maior soma dos seus nodos da raiz até uma folha da árvore.

22. O valor que está na raiz da árvore pode ser par ou ímpar. Crie um método **int maxpath()** que acha o tamanho do caminho contendo o maior número de valores com a mesma paridade da raiz, partindo da raiz da árvore.

23. Crie um método **int numfolhas()** que examina a árvore e retorna o número de folhas que existem nela.

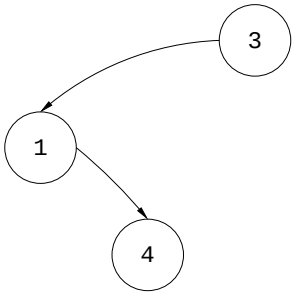
24. Suponha que a árvore tem valores repetidos. Crie um método **int depth(int val)** que retorna o maior nível dentre os nodos que contém o valor **val**.

25. Escreva um algoritmo **boolean equiv(B)** que recebe uma árvore binária de pesquisa **B** e testa se a árvore atual (**A**) e **B** são equivalentes. Faça um método para cada noção de equivalência abaixo:

    - **A** e **B** são equivalentes se tem os mesmos elementos, mas eles podem estar em quantidades e posições diferentes.
    - **A** e **B** são equivalentes se tem os mesmos elementos, mas eles precisam estar em quantidades iguais.
    - **A** e **B** são equivalentes se tem os mesmos ramos nas mesmas posições, mas os valores em cada nodo não interessam. Ou seja, o desenho das duas árvores é idêntico, mas seu conteúdo não é importante.

- **A** e **B** são equivalentes se tem os mesmos ramos nas mesmas posições e com os mesmos valores em cada nodo. Ou seja, as árvores são absolutamente iguais.

26. Escreva um algoritmo **boolean isABP()** que recebe uma árvore binária e verifica se ela é uma árvore binária de pesquisa. Depois teste seu método com a árvore abaixo:



27. Altere a árvore para que cada nodo possa guardar informação adicional:

    - Crie um método para guardar em cada nodo a informação de sua altura dentro da árvore. Seu método deve custar tanto quanto um caminhamento.
    - Crie um método para guardar em cada nodo a informação de quantos nodos existem em sua sub-árvore esquerda e quantos nodos existem em sua sub-árvore direita.
    - Use as duas informações para descobrir como fazer um **desenho** da árvore. Procure detalhes em `http://www.cs.bgu.ac.il/~matya/Courses/AS09/chen_shay.ppt`.

28. Escreva o método que imprime cada nodo de uma árvore binária e seu fator de balanceamento AVL. Seu método **não pode** usar métodos auxiliares.

29. Para uma árvore binária contendo inteiros, encontre todos os nodos que não tem filhos à direita e transfira o filho da esquerda para a direita.

30. Para uma árvore binária contendo inteiros, apresente um algoritmo que arruma a árvore para que o nodo pai sempre tenha o maior valor de seus filhos, transformando-a em um max-heap.