---

1. Árvores de Arne Andersson

Um dos principais interesses das estruturas de árvores é poder armazenas muitos dados, sem gerar para tanto cadeias muito longas de ponteiros. Isto é muito interessante pois vários algoritmos em árvores binárias de busca têm complexidade proporcional à altura.

Entretanto, isto funciona apenas se construímos árvores balanceadas. Uma árvore é balanceada quando os comprimentos máximos de ramos adjacentes diferem pouco um dos outros. A altura n da árvore é portanto logarítmica no número de nós. Do contrário, nas árvores ditas desbalanceadas, a altura é proporcional ao número de nós. Os algoritmos clássicos são neste caso menos eficazes.

Nesta prática, iremos implementar uma classe de árvores balanceadas diferente daquelas vistas em sala (e também mais simples). Trata-se das árvores de *Arne Andersson* (que chamaremos AA daqui em diante), propostas por Andersson em 1993, e que são uma variante das árvores de busca *rubro-negras*.

O princípio destas árvores é utilizar dois tipos de nós, diferenciados por sua cor: os nós pretos aparecessem em mesmo número em todo caminho que vai da raiz até uma folha (garantia forte do balanceamento), enquanto que os nós vermelhos podem aparecer em quantidades diferentes em diferentes ramos. Além disso, é imposta a seguinte condição adicional: a de que todo filho esqquerdo deve ser preto.

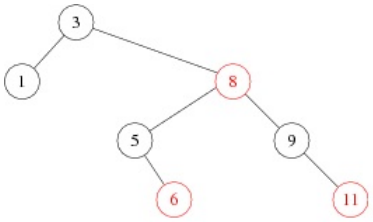
De forma precisa, uma árvore binária de busca é AA se e somente se ela satisfaz as seguintes propriedades:

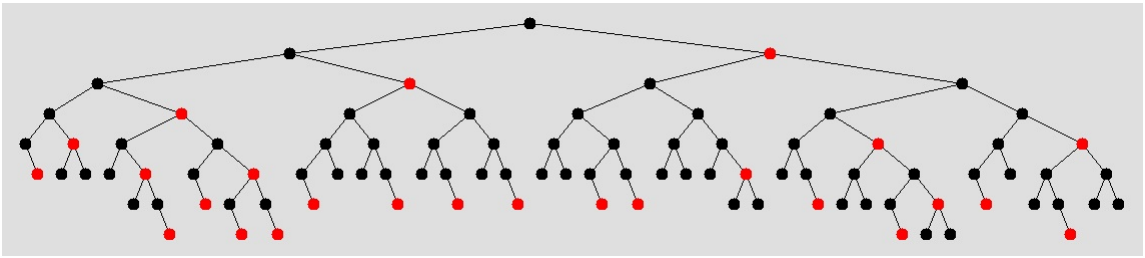
- cada nó tem uma cor podendo ser vermelha ou preta;
- a raiz é um nó preto
- os dois filhos de um nó vermelho são pretos
- o número de nós pretos é o mesmo em todos os caminhos que vão da raiz até uma folha; este número é denominado *nível da árvore*
- o filho esquerdo de todo nó é preto (dito de outra forma, todo nó vermelho é filho direito de um outro nó)

Intuitivamente, os nós pretos obedecem a uma hipótese de balanceamento forte, enquanto que os nós vermelhos permitem um certo nível de desequilíbrio, que é entretanto limitado pelo nível da árvore. Mais precisamente, no caminho indo da raiz até a folha mais à esquerda da árvore só existem nós pretos, enquanto que o número potencial de nós vermelhos presentes em cada caminho raiz-folha aumenta (em uma certa medida apenas) ao nos deslocarmos para a folha mais à direita da árvore.

A Figura abaixo ilustra uma árvore AA de nível 2, ou seja cujos caminhos da raiz até as folha contem cada um 2 nós pretos (note o desequilíbrio relativo para a direita)

Abaixo outro exemplo de árvore AA de nível 5





(a) **Implementação**

Crie uma classe de árvores `Aa` em que cada nó contem quatro campos: um campo `int valor`, um campo `int cor`, assim como dois campos `Aa esq` e `Aa dir`. Estes dois últimos correspondem aos dois filhos do nó atual. O campo `cor` indica a cor do nó: 0 para vermelho, 1 para preto. A fim de deixar o código mais legível, declare duas constantes para a classe `static int R=0` e `static int N=1`, correpondendo às duas cores. Em seguida, você poderá utilizar estas duas constantes ao invés dos inteiros para fixar a cor dos nós.

Adicione um construtor `Aa(int v, int c, Aa Esq, Aa Dir)` que inicializa os campos da nova árvore. Note que este construtor pode construir árvores que não são AA e deverá portanto ser utilizado com atenção a seguir.

Escreva um método `static String infixe(Aa a)` que percorre a árvore `a` e retorna uma cadeia de caracteres listando os valores contidos nos nós de `a`. Nós listaremos os valores *em ordem*, ou seja para cada nó será listado primeiro os valores a partir do filho esquerdo, depois o valor do nó atual e depois os valores a partir do filho direito. Para testar o seu código, copie-e-cole a função `main` abaixo:

```
public static void main (String [] args) {
  Aa a = new Aa (3, N,
  new Aa (1, N, null, null),
    new Aa (8, R,
     new Aa (5, N,
     null,
     new Aa (6, R, null, null)),
     new Aa (9, N,
     null,
     new Aa (11, R, null, null))));
  System.out.println (infixe(a));
```

```
}
```

O resultado deverá ser

```
1 3 5 6 8 9 11
```

2. **Inserção e manutenção das propriedades das árvores AA**

   Nós vamos agora implementar um método permitindo inserir um elemento em uma árvore AA, mantendo suas propriedades. A operação de inserção não resulta em grandes dificuldades, uma vez que é utilizado o algoritmo clássico de inserção em uma árvore binária de busca. A parte difícil consiste a restabelecer as propriedades das árvores AA que a inserção pode por ventura violar.

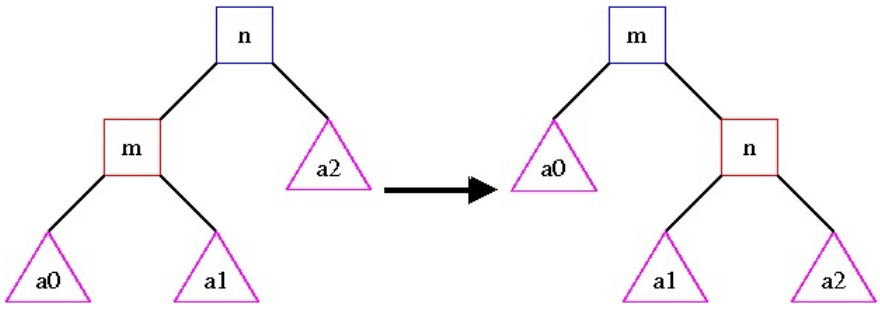
   Para começar, vamos examinar o que pode acontecer após a inserção de um nó da cor vermelha como uma folha (escolhendo a cor vermelha, estamos seguros de que a propriedade de que todos os caminhos da raiz até uma folha atravessa o mesmo número de nós pretos não é violada):

   - este novo nó é o filho esquerdo de um nó existente; a propriedade de que todo nó vermelho é o filho direito de um outro nó é portanto violada e deve ser restabelecida.
   - este novo nó é o filho direito de um nó existente; então, pode ser que a propriedade que diz que os filhos de um nó vermelho são pretos seja violada e deve, portanto, ser restabelecida.
   - as outras propriedades das árvores AA são preservadas por esta operação.

   Considere estes dois problemas nas questões a seguir, implementando um método que recebe como argumento uma árvore tal que as propriedades das árvores AA são satisfeitas, salvo eventualmente na raiz da árvore, onde nós faremos um rebalanceamento, caso necessário.

   (a) **Rotação à direita**

   No caso em que o filho à esquerda da raiz é um nó vermelho, podemos consertar o problema efetuando o rebalanceamento abaixo (um retângulo azul representa um nó cuja cor não é especificada; um triângulo violeta representa uma subárvore AA):



   Note que após esta estapa, podemos obter uma árvore que não satisfaz as propriedades das árvores AA: se o filho direito da raiz fosse vermelho antes da rotação, então o novo filho direito seria vermelho com ele mesmo tendo um filho vermelho.

Isto não é grave, pois a operação de rotação à esquerda apresentada a seguir fará o reparo desta situação.

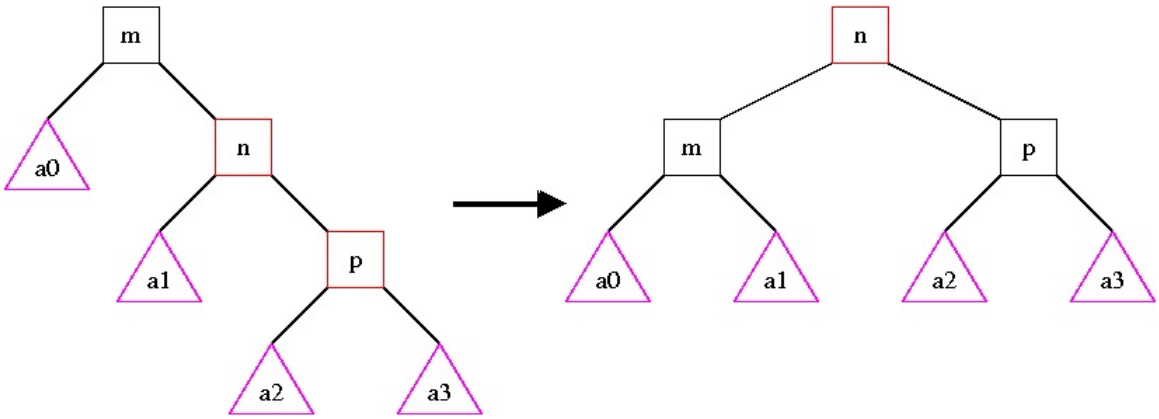
Implemente o método `rodeDir` que efetua o rebalanceamento acima na raiz da árvore caso necessário e retorna a árvore sem modificações caso contrário. Para testar o seu código, construa uma árvore `a` que não satisfaz as propriedades de árvores AA no nível de sua raiz, depois chame `rodeDir(a)`. Tome por exemplo a árvore a seguir. cuja estrutura é inspirada na árvore desenhada acima:

```
Aa a = new Aa (4, R,
               new Aa (2, R,
                       new Aa (1, N, null, null),
                       new Aa (3, N, null, null)),
               new Aa (5, N, null, null));
```

A fim de comparar as estruturas da árvore `a` antes e após o rebalanceamento, uma classe `Fenetre` é fornecida cujo construtor cria uma janela gráfica para imprimir a estrutura da árvore. Assim, cada vez que `new Fenetre(a)` é invocada em seu código, uma nova janela se abrirá e imprimirá a estrutura da árvore `a`. O código da classe `Fenetre` se encontra no arquivo `Fenetre.java` no SIGAA.

(b) **Rotação à esquerda**

No caso em que o filho à direita e o seu próprio filho à direita são vermelhos, podemos obter uma sub-árvore AA efetuando a rotação abaixo:



Note que a raiz m é necessariamente preta no caso em que n e p são vermelhos, após uma inserção e uma rotação à direita de árvore `Aa` em uma árvore `Aa` correta. Portanto, não é necessário verificar a cor de m.

Após esta etapa, obtemos uma subárvore AA válida. Entretanto, como a raiz desta subárvore é vermelha, podemos ter um problema com o nível imediatamente superior, onde raiz e filho à direita poderão ser ambos vermelhos. Este problema será entretando resolvido através de uma rotação à esquerda no nível superior.

Implemente o método `rodeEsq` que efetua o rebalanceamento acima na raiz da árvore caso necessário e retorna a árvore sem modificações caso contrário. Como na questão precedente, você pode testar o seu método construindo uma árvore `a` que

não satisfaz as propriedades de uma árvore AA no nível da raiz, e depois chamar `rodeEsq`. Tome por exemplo a árvore a seguir, inspirada na árvore acima.

```
Aa a = new Aa (2, N,
               new Aa (1, N, null, null),
               new Aa (4, R,
                   new Aa (3, N, null, null),
                   new Aa (6, R,
                           new Aa (5, N, null, null),
                       new Aa (7, N, null, null))));
```

(c) **Inserção em uma árvore AA**

Com a ajuda dos métodos implementados nas questões precedentes, escreva um método recursivo `static Aa insere(Aa a,int i)` que insere o elemento `i` na árvore `AA a`. Este método retornará uma árvore AA com todas as suas propriedades, com exceção da raiz que poderá ser preta ou vermelha. Para isto, recomenda-se a abordagem a seguir:

- efetue uma descida recursiva clássica;
- insera um nó vermelho no lugar de uma folha
- ao se retornar de cada chamada recursiva, chame sucessivamente os dois métodos de rebalanceamentos descritos acima, primeiro `rodeDir` e depois `rodeEsq` na árvore que você está prestes a retornar (ou seja a nova árvore criada depois da inserção na subárvore esquerda ou na subárvore direita).

Esta abordagem garante que a cada retorno de uma chamada recursiva a subárvore retornada tem dois filhos AA válidos, graças à aplicação das rotações. No entanto, estas últimas podem gerar problemas no nível acima da subárvore corrente, se por exemplo ela for vermelha com filho direito vermelho. Estes problemas são corrigidas no nível imediatamente superior ao se retornar da chamada recursiva.

Desta forma, ao final deste processo, obteremos finalmente uma subárvore AA válida, salvo pela raiz, que pode ser vermelha, o que viola uma ou várias das propriedades de ávores AA. Neste caso, resolvemos facilmente o problema trocando a cor da raiz se ela for vermelha, o que leva a uma árvore AA válida.

Escreva um método `insereECorrigeRaiz` que faz uma chamada a `insere` e depois atualiza a cor da raiz da árvore obtida se necessário. A fim de testar o seu código, crie uma árvore vazia na qual você insere alguns inteiros. Após cada inserção você poderá verificar na função `main` a estrutura de `a` através do comando `new Fenetre(a)`.

3. **Verificando se uma árvore é AA**

Iremos agora verificar de maneira automática que as árvores que nós contruimos são de fato árvores AA. Para isto, fornecemos abaixo um método de verificação

```
static boolean testeArvoreAa(Aa a) {
  return testSubArvoreAa(a, nivel(a), false);
}
```

Escreva o método `nivel` que conta o número de nós pretos da raiz até uma folha arbitrária. O nível da árvore vazia (`null`) é 0.

Escreva o método `testeSubArvoreAa(Aa a, int n, boolean raizPodeSerVermelha)`, onde `n` representa o nível da subárvore atual, e `raizPodeSerVermelha` indica se a raiz da subárvore atual pode ser vermelha ou não. O método fará as verificações a seguir:

- Se a árvore `a` é vazia n deve ser zero, pois senão existe um problema de nível no ramo da árvore atualmente percorrido.
- Se a raiz é vermelha `raizPodeSerVermelha` tem que ser `true`, pois senão existe um problema de nós vermelhos consecutivos ao longo do ramo percorrido ou de nó vermelho sendo filho esquerdo de outro nó.
- Em seguida, o método `testeSubArvoreAa` é chamado recursivamente para as duas subárvores de `a`. É tarefa sua saber os valores corretos dos parâmetros para estas chamadas recursivas.

Finalmente, para testar se `a` é uma árvore do tipo AA, será suficiente chamar `testeArvoreAa(a)`.