**Prova 2 — PMR2300 1o. semestre 2015**

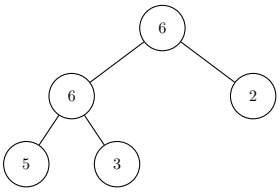
Prof. Thiago Martins

Instruções: Escreva o nome e o número USP na folha de papel almaço.

1. (2,5 pontos) Um heap binário é uma árvore binária completa (todos os níveis exceto o último cheios) preenchido da esquerda para a direita na qual cada nó é maior ou igual a seus filhos. Heaps binários são armazenados em vetores, resultantes da varredura da árvore binária *em largura*. Dada a rígida estrutura, é possível converter um vetor em árvore e vice-versa.

   (a) (0,9 pontos) Mostre na forma de árvore binária a estrutura do heap binário representado pelo vetor $[6, 6, 2, 5, 3]$.

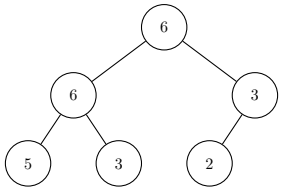
   **Resposta:**

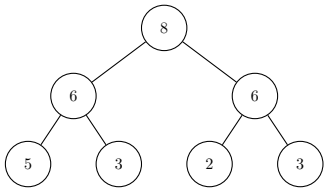
   (b) (0,8 pontos) Novos elementos podem ser adicionados ao heap de forma eficiente. Para tanto, o novo elemento é anexado *ao final* do heap. Em seguida, a sub-árvore afetada é "corrigida", movendo-se o elemento novo para sua posição correta na hierarquia. O processo prossegue heap acima, sub-árvore a sub-árvore, até que não sejam necessárias mais alterações. Mostre o resultado na forma de árvores da inserção sucessiva dos valores 3, 8, e 7.

   **Resposta:**

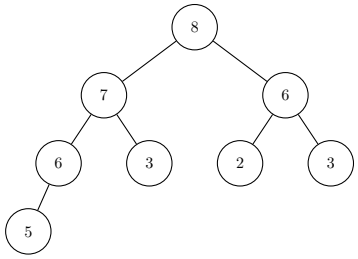
   1. Inserção de 3:

   2. Insersão de 8:

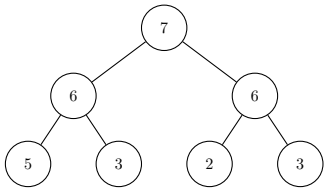
   3. Insersão de 7:

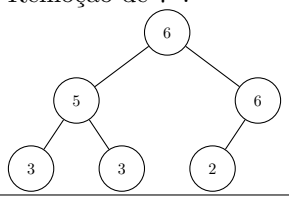
   (c) (0,8 pontos) Da mesma maneira, *o maior valor* em um heap (nó raiz) também pode ser eficientemente removido. Para tanto, ele é trocado de posição com o último elemento, e removido do vetor. Em seguida, a sub-árvore raiz é corrigida movendo-se o novo elemento para a posição adequada. O processo prossegue heap abaixo até que não sejam necessárias mais alterações. Mostre o resultado na forma de árvores da remoção sucessiva de 3 dos maiores elementos do heap produzido no item b).

**Resposta:**

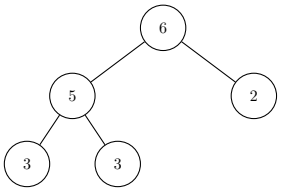
1. Remoção de 8:




2. Remoção de $7^1$:




3. Remoção de 6:




A correção do nó raiz pode seguir tanto pela esquerda quanto pela direita. Tomou-se aqui o caminho da esquerda. *Ambos estão corretos.*

2. (2,5 pontos) Uma fila é uma estrutura de dados na qual o elemento retirado é o *mais antigo* a ser inserido. A classe `FilaAr` implementa uma fila sobre um arranjo simples. O método `enqueue(Object x)` enfileira um novo objeto, o método `dequeue()` retira um objeto da fila e o método `tamanho()` retorna a quantidade de objetos enfileirados.

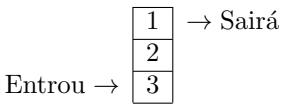
   (a) (1.0 pontos) Considere o código abaixo:

```
1    FilaAr f = new FilaAr();
2    f.enqueue(1);
3    f.enqueue(2);
4    f.enqueue(3);
5    f.enqueue(f.dequeue());
6    f.enqueue(f.dequeue());
7    int a = (int)f.dequeue();
```

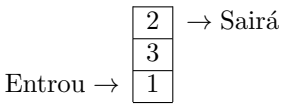
   Mostre esquemas do estado da fila destacando o *último* elemento enfileirado e o *primeiro* elemento a ser retirado após a execução das linhas 4, 5, e 6. Qual o valor da variável `a` *após* a execução da linha 7?

**Resposta:**

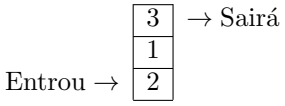
- Após linha 4:




- Após linha 5:




- Após linha 6:




O valor final da variável `a` é 3, o *último* elemento a ser inserido antes das linhas 5 e 6.

   (b) (1,5 pontos) Uma pilha é uma estrutura de dados na qual o elemento a ser retirado é o *último* a ser inserido. Mostre que é possível implementar uma pilha usando uma fila. Para tanto, complete a implementação da classe `PilhaPorFila` a seguir, com o corpo dos métodos:

   - `push(Object x)`: Adiciona o objeto `x` à pilha.

- `pop()`: Remove e retorna o objeto mais recente a ser inserido, ou lança a exc`ErroPilhaVazia` se não há mais objetos empilhados.
- `tamanho()`: Retorna a quantidade de objetos empilhados.

```
public class PilhaPorFila {
    public class ErroFilaVazia extends java.lang.RuntimeException {}
    private FilaAr f;
    public PilhaPorFila() {
        f = new FilaAr();
    }
    public void push(Object x) {
        ⋮
    }
    public Object pop() {
        ⋮
    }
    public int tamanho() {
        ⋮
    }
}
```

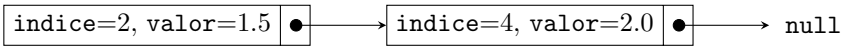
**Resposta:**

Como mostra o item a), é possível levar o último elemento de uma fila até a posição de remoção fazendo $n-1$ operações do tipo `f.enqueue(f.dequeue()`, onde $n$ é a quantidade de elementos enfileirados (recuperável pelo método `tamanho()`). O código final é:

```
,
 public class PilhaPorFila {
    public class ErroFilaVazia extends java.lang.RuntimeException {}
    private FilaAr f;

    public PilhaPorFila() {
        f = new FilaAr();
    }

    public void push(Object x) {
        f.enqueue(x);
    }

    public Object pop() {
        if(f.tamanho()==0) throw new ErroFilaVazia();
        for(int i=0;i<f.tamanho()-1;i++)
            f.enqueue(f.dequeue());
        return f.dequeue();
    }

    public int tamanho() {
        return f.tamanho();
    }
}
```

3. (2,5 pontos) Um vetor *esparso* é um vetor no sentido algébrico do termo, no qual a maioria dos coeficientes é zero. Um vetor esparso pode ser eficientemente representado na memória de um computador por meio de uma lista ligada que armazena somente os coeficientes não-nulos. Seja um coeficiente de vetor esparso representado por um elemento da classe `CoeficienteVetor`. O campo `indice` desta classe armazena o índice do coeficiente e o campo `valor` armazena o valor do ceoficiente. Um vetor esparso nesta implementação seria uma lista ligada com nós da classe `NoVetorEsparso` em que cada um, no seu campo `coef`, contém um coeficiente. Os coeficientes são armazenados em *ordem crescente* de

índice do coeficiente. Por exemplo, nesta representação, o vetor de cinco dimensões $A = [0, 1.5, 0, 2.0, 0]$ é representado pela lista ligada:




Note que $A_1 = 0, A_2 = 1.5, A_3 = 0, A_4 = 2.0, A_5 = 0$. Os coeficientes nulos do vetor (índices 1, 3 e 5) são representados *implicitamente* pela sua ausência.

Escreva um código que calcula o *produto interno* de dois vetores esparsos $A$ e $B$ definido como $\sum_i A_i B_i$.

Use a seguinte assinatura:

```
static double produtoInterno(NoVetorEsparso a, NoVetorEsparso b)
```

Onde `a` e `b` são os primeiros nós das listas ligadas que representam os vetores $A$ e $B$. Seja $N$ o número de coeficientes *não-nulos* do vetor $A$ e $M$ o equivalente para $B$. Um algoritmo de ordem $\mathcal{O}(N + M)$ vale 2,5 pontos nessa questão (note o armazenamento em ordem crescente!). Algoritmos com pior complexidade valem 1,5 pontos.

**Resposta:**

É possível explorar a ordenação das listas para produzir um algoritmo de ordem $\mathcal{O}(N + M)$. Para tanto, varre-se sequencialmente as listas procurando-se incrementar sempre o nó da lista com o *menor* indice. Quando os valores dos indices coincide, faz-se o produto dos valores e adiciona-se o resultado ao total. O algoritmo prossegue até uma das listas acabar.

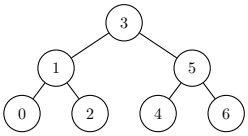
```
,
static double produtoInterno(NoVetorEsparso a, NoVetorEsparso b) {
    double result = 0;
    while(a!=null && b!=null) {
        if(a.coef.indice < b.coef.indice)
            a = a.next;
        else if(a.coef.indice > b.coef.indice)
            b = b.next;
        else { // iguais
            result += a.coef.valor*b.coef.valor;
            a = a.next;
            b = b.next;
        }
    }
    return result;
}
```

4. (2.5 pontos) Uma árvore de busca binária é uma árvore binária na qual cada nó é *estritamente maior* que *todos* os elementos da sua sub-árvore esquerda e *estritamente menor* que *todos* os elementos da sua sub-árvore direita (vide exemplos a seguir). Esta propriedade permite a rápida localização de elementos nela contidos. A classe `NoBinarioFloat` contém nós de uma árvore binária que armazena números em ponto flutuante. O campo `val` contém o número armazenado no nó, o campo `esquerda` contém uma referência para a raiz da sub-árvore esquerda e o campo `direita` contém uma referência para a raiz da sub-árvore direita. Escreva uma função em java que verifica se uma determinada árvore binária formada por `NoBinarioFloat` é efetivamente uma árvore de busca binária. Use a seguinte assinatura:

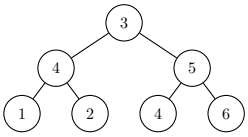
```
static boolean checaArvoreDeBusca(NoBinarioFloat n)
```

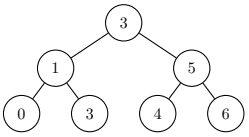
Onde `n` é o nó raiz da árvore a ser verificada. A função deve retornar verdadeiro caso a árvore seja uma árvore de busca, falso caso contrário. Qual a ordem de complexidade do algoritmo implementado em função do número de elementos da árvore $N$?

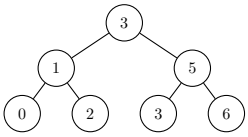
Exemplo de árvore binária de busca:




Exemplos de árvores binárias que *não são* de busca:




**Resposta:** Este é um problema que a um olhar desatento pode parecer erroneamente simples. De fato, *não basta* verificar a ordenação local de um nó e seus filhos imediatos. Como pode-se ver nos exemplos do enunciado, há árvores binárias com ordenação local válida, mas que ainda assim não são árvores binárias de busca.

Há diversas soluções possíveis para o problema. Eis algumas:

- *Validação por busca*: Em uma árvore de busca binária, como menciona o enunciado, os valores de *todos* os nós podem ser rapidamente encontrados via busca binária. Para tanto, implementa-se a função `procuraNaArvore(NoBinarioFloat n, double v)` que retorna a referência ao nó no qual está contido o valor `v`, ou `null` se o valor não pode ser encontrado. Em seguida, percorre-se *todos* os nós da árvore apresentada e verifica-se para cada nó se o seu valor foi encontrado efetivamente no próprio nó. A ordem de varredura neste caso não importa. Em particular, aqui segue-se a ordem em profundidade anterior, ou seja, o valor do nó atual é verificado, em seguida a busca prossegue na sub-árvore esquerda depois na direita.

  ,
  ```
  static NoBinarioFloat procuraNaArvore(NoBinarioFloat n, double v) {
      if(n==null) return null;
      if(n.val>v) return procuraNaArvore(n.esquerda, v);
      if(n.val<v) return procuraNaArvore(n.direita, v);
      return n;
  }

  static boolean checaArvoreDeBusca(NoBinarioFloat n, NoBinarioFloat raiz) {
      if(n==null) return true; // Arvore nula
      return n==procuraNaArvore(raiz, n.val) &&
               checaArvoreDeBusca(n.esquerda, raiz) &&
               checaArvoreDeBusca(n.direita, raiz);
  }

  static boolean checaArvoreDeBusca(NoBinarioFloat n) {
      return checaArvoreDeBusca(n, n);
  }
  ```

  Esta solução faz $N$ buscas na árvore, o que tem complexidade $\mathcal{O}(h)$, $h$ sendo a profundidade do nó sendo buscado. No pior caso a complexidade é $\mathcal{O}(N^2)$ (Pode-se mostrar que para uma árvore razoavelmente balanceada a complexidade *média* é $\mathcal{O}(N \log N)$).

- *Validação por comparação de extremos*: Em uma árvore binária de busca, o nó seguinte ao atual *em ordem crescente de valor* é o nó mais a esquerda da sub-árvore direita. Do mesmo modo, o nó anterior é o nó mais a direita da sub-árvore esquerda.

  E possível implementar as funções `extDireita` e `extEsquerda`. Em seguida, verifica-se se o nó atual é estritamente maior do que o extremo direito da sub-árvore esquerda (se ela

existe) e estritamente menor que o extremo esquerdo da sub-árvore direita (se ela existe). Finalmente, o processo é aplicado a todos os nós da árvore. Aqui, novamente, optou-se por uma varredura em profundidade Anterior.

```
,
static double extDireita(NoBinarioFloat n) {
    while(n.direita!=null) n = n.direita;
    return n.val;
}

static double extEsquerda(NoBinarioFloat n) {
    while(n.esquerda!=null) n = n.esquerda;
    return n.val;
}

static boolean checaArvoreDeBusca(NoBinarioFloat n) {
    if(n==null) return true; // Arvore nula
    if(n.esquerda!=null && n.val <= extDireita(n.esquerda)) return false;
    if(n.direita!=null && n.val >= extEsquerda(n.direita)) return false;
    return checaArvoreDeBusca(n.esquerda) &&
            checaArvoreDeBusca(n.direita);
}
```

A complexidade deste método depende da altura $h$ do nó sendo visitado. Uma abordagem simplória sugere uma complexidade $\mathcal{O}(N^2)$, e, do ponto de vista estrito da notação *big Oh*, está *correta*. É possível no entanto obter um limite melhor superior. De fato, ao se considerar os níveis *verticais* de uma árvore binária, percebe-se que a adição de um novo nó aumenta o número de passos em apenas *dois* outros nós, no pior caso (os dois nós entre os quais ele é inserido, verticalmente). Como cada nó adicional em uma árvore aumenta a complexidade em um número *constante* de passos, a complexidade deste método é — talvez surpreendentemente — $\mathcal{O}(N)$.

- *Validação por enumeração interior* - 1: A enumeração *interior* dos nós de uma árvore binária de busca retorna uma *ordenação* dos seus valores. Basta então percorrê-los em ordem interior e verificar a ordem dos valores obtida. Há a dificuldade no entanto de comparar valores entre diferentes etapas da enumeração. Uma solução é adicionar os valores, durante a enumeração, a uma estrutura de dados auxiliar e posteriormente verificar se eles estão ordenados. O código-fonte de apoio possui uma classe `FilaAr` que se presta a esse papel:

```
,
static void percorreArvoreInterna(NoBinarioFloat n, FilaAr saida) {
    if(n==null) return;
    percorreArvoreInterna(n.esquerda, saida);
    saida.enqueue(n.val);
    percorreArvoreInterna(n.direita, saida);
}

static boolean checaArvoreDeBusca(NoBinarioFloat n) {
  FilaAr saida = new FilaAr();
  percorreArvoreDeBusca(n, saida);
  if(saida.tamanho()==0) return true;
  double a = (double)saida.dequeue();
  while(saida.tamanho()!=0) {
      double b = (double)saida.dequeue();
      if(a>=b) return false;
      a = b;
  }
  return true;
}
```

Esta solução faz $N$ passos de varredura da arvore e $N$ passos de verificação de ordenação. Sua complexidade é $\mathcal{O}(N)$.

- *Validação por enumeração interior* - 2: O armazenamento em estrutura auxiliar para posterior verificação parece uma etapa desnecessária - deveria ser possível verificar a ordenação *enquanto* a enumeração é feita. Para tanto, seria necessário armazenar apenas o valor do *penúltimo* nó enumerado. A única dificuldade desta abordagem é identificar a partir de que ponto da enumeração recursiva o primeiro nó foi visitado (problema análogo ao de recuperar o primeiro valor da fila para verificar a ordenação dos posterirores). Uma solução é criar uma estrutura *mínima* de armazenamento, com apenas dois campos: um campo `init` que indica se o valor da estrutura foi inicializado e o campo `val` que, se `init` é verdadeiro, contém o valor do penúltimo nó visitado (valor indefinido caso contrário).

```
,
static class Minima {
    double val;
    boolean init = false;
}

static boolean checaArvoreDeBusca(NoBinarioFloat n, Minima x) {
    if(n==null) return true;
    if(!checaArvoreDeBusca(n.esquerda, x)) return false;
    if(x.init && x.val >= n.val) return false;
    x.val = n.val;
    x.init = true;
    return checaArvoreDeBusca(n.direita, x);
}

static boolean checaArvoreDeBusca(NoBinarioFloat n) {
    return checaArvoreDeBusca(n, new Minima());
}
```

Esta solução faz $N$ passos de varredura da arvore, de modo que sua complexidade é $\mathcal{O}(N)$.

---

# Códigos-fonte de apoio

Atenção: Em todas as questões da prova o código abaixo pode ser considerado disponível e referenciado sem necessidade de duplicação.

Classe `FilaAr`

```
public class FilaAr {
  public class ErroFilaVazia extends java.lang.RuntimeException {}

  private Object arranjo[];
  private int count, in, out;

  public FilaAr() {
     arranjo = new Object[16]; esvazie();
  }

  public final void esvazie() {
     count = 0; out = 0; in=arranjo.length-1;
     for(int i=0;i<arranjo.length;i++) arranjo[i]=null;
  }

  public int tamanho() {
     return count;
  }

  private int next(int indice) {
     return (indice+1)%arranjo.length;
  }

   public void enqueue(Object x) {
       if(count == arranjo.length) dupliqueArranjo();
       in = next(in); arranjo[in] = x; count++;
   }

  private void dupliqueArranjo() {
       Object novo[] = new Object[2*arranjo.length];
       for(int i=0; i<count; i++) {
          novo[i] = arranjo[out]; out = next(out);
       }
       arranjo = novo;
       out = 0; indiceEntrou = count-1;
  }

  public Object dequeue() {
     count--;
     Object x = arranjo[out];
     arranjo[out]=null; out = next(out);
     return(x);
  }
}
```

Classe `NolistaLigada`:

```
public class NolistaLigada {
    Object val;
    NolistaLigada next;

    NolistaLigada(Object value, NolistaLigada next) {
        this.val = value;
        this.next = next;
    }
}
```

Classe `NoVetorEsparso`:

```
public class NoVetorEsparso {
    CoeficienteVetor coef;
    NoVetorEsparso next;

    NoVetorEsparso(CoeficienteVetor value, NoVetorEsparso next) {
        this.coef = value;
        this.next = next;
    }
}
```

Classe `CoeficienteVetor`:

```
public class CoeficienteVetor {
    int indice;
    double valor;
    CoeficienteVetor(int i, double v) {
        indice = i;
        valor = v;
    }
}
```

Classe `NoBinario`

```
 public class NoBinario {
   Object dado;
   NoBinario esquerdo, direito;

   public NoBinario(Object x, NoBinario e, NoBinario d) {
      dado = x;
      esquerdo = e;
      direito = d;
   }
 }
```

Classe `NoBinarioFloat`

```
public class NoBinarioFloat {
    double val;
    NoBinarioFloat esquerda;
    NoBinarioFloat direita;


    NoBinarioFloat(double val, NoBinarioFloat e, NoBinarioFloat d) {
        this.val = val;
        esquerda = e;
        direita = d;
    }
}
```