Prova Substitutiva — PMR3201/PMR2300 1o. semestre 2015
Prof. Thiago Martins

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

1. (2,0 pontos) Considere a classe `HashSep` que implementa uma tabela de hash com encadeamento aberto e resolução *linear* (Em caso de colisão, a próxima posição livre é usada). Essa tabela emprega *duplicação* do arranjo base quando o seu fator de ocupação (razão entre quantidade de entradas armazenadas e o tamanho do arranjo) atinge ou supera 80%. A duplicação é uma operação na qual o arranjo base é substituído por um novo arranjo com o *dobro* do comprimento do anterior, e os elementos armazenados são re-inseridos no novo arranjo.

   As chaves empregadas são números inteiros e a função de espalhamento empregada é:

$$\operatorname{hash}(i) = ((a \cdot i + b) \mod p) \mod s$$

   Onde i é o índice a ser espalhado, p é um número primo fixo, s é o tamanho atual do vetor base e a e b são números aleatórios entre 1 e $p-1$ gerados na criação da tabela. A função `setData(int key, Object x)` adiciona um novo objeto à tabela Suponha que a tabela `h` no código abaixo tenha sido construída com $p = 11$, $s = 4$, $a = 7$ e $b = 4$. Mostre o conteúdo da tabela após a execução de cada linha:

```
1       h.setData(1, "um");
2       h.setData(2, "dois");
3       h.setData(3, "três");
4       h.setData(4, "quatro");
```

   Para tanto desenhe um diagrama mostrando o conteúdo de cada posição do vetor base. Indique também para cada entrada armazenada quantos passos são necessários para encontrá-la em uma busca (1 para acesso imediato via função de hash e um passo adicional para etapa de busca linear).

Resposta:

- Após `h.setData(1, "um");`

  Hash de índice 1 = $(0 \mod 4) = 0$

índice:	0	1	2	3
Conteúdo:	1 "um"			
Passos:	1			

- Após `h.setData(2, "dois");`

  Hash de índice 2 = $(7 \mod 4) = 3$

índice:	0	1	2	3
Conteúdo:	1 "um"			2 "dois"
Passos:	1			1

- Após `h.setData(3, "três");`

  Hash de índice 2 = $(3 \mod 4) = 3$. Há colisão com o já ocupado índice 3 e o já ocupado índice 0.

índice:	0	1	2	3
Conteúdo:	1 "um"	3 "três"		2 "dois"
Passos:	1	3		1

- Após `h.setData(3, "quatro");`

  A ocupação nesta inserção atinge os 100%. Neste caso um novo vetor é criado e os elementos são re-inseridos. As novas posições são:

  1. Hash: $(0 \mod 8) = 0$
  2. Hash: $(7 \mod 8) = 7$
  3. Hash: $(3 \mod 8) = 3$
  4. Hash: $(10 \mod 8) = 2$

índice:	0	1	2	3	4	5	6	7
Conteúdo:	1 "um"		4 "quatro"	3 "três"				2 "dois"
Passos:	1		1	1				1

2. (2,0 pontos) Considere o problema de se determinar se um número está ou não presente em uma matriz na qual todas as colunas e linhas estão *ordenadas em ordem crescente*.

   Exemplo:

$$\begin{pmatrix} 1 & 3 & 10 & 23 \\ 4 & 7 & 18 & 36 \\ 7 & 12 & 26 & 44 \end{pmatrix}$$

   Uma maneira de se resolver este problema é dividir a matriz em quatro quadrantes, noroeste, nordeste, sudeste e sudoeste. Em seguida, o valor na posição no *centro* da matriz é verificada. Caso ele seja *maior* do que o número procurado, o quadrante "*sudeste*" (inferior direito) é removido da busca (acrescido da coluna e linha centrais) e a busca prossegue nos 3 quadrantes seguintes. Caso ele seja *menor*, o quadrante "*noroeste*" (acrescido da coluna e linha centrais) é eliminado da busca e ela prossegue nos 3 quadrantes restantes. A listagem a seguir contém uma implementação em Java deste algoritmo:
   ,

```
static boolean procura(int v, int[][] a) {
    return procura(v, a, 0, a[0].length-1, 0, a.length-1);
}

static boolean procura(int v, int[][] a, int o, int e, int n, int s) {
    if(o>e || n > s) return false;
    int x = (o+e)/2;        // Coordenadas do
    int y = (n+s)/2;        //   centro
    int t = a[y][x];
    if(v==t) return true;   // Achou

    if(v>t) { // Descarta canto NO
        return  procura(v, a, x+1, e, n, y) ||    // NE
                procura(v, a, x+1, e, y+1, s) ||  // SE
                procura(v, a, o, x, y+1, s);      // SO
    } else { // Descarta canto SE
        return  procura(v, a, x, e, n, y-1) ||    // NE
                procura(v, a, o, x-1, y, s) ||    // SO
                procura(v, a, o, x-1, n, y-1);    // NO
    }
}
```

   A matriz é representada por um vetor de linhas, passado no parâmetro `a`. O parâmetro `v` contém o valor a ser procurado. A função `procura(int v, int[][] a)` chama a função recursiva `procura(int`

`v, int[][] a, int o, int e, int n, int s)`, na qual os parâmetros `o`, `e`, `n` e `s` são os limites da matriz a oeste (esquerda), leste (direita), norte (acima) e sul (abaixo).

(a) (1,0 pontos) Escreva a equação de recorrência da ordem de complexidade deste algoritmo em função de N, o número total de elementos da matriz.

> **Resposta:** O algoritmo divide a matriz em 4 quadrantes, cada um com aproximadamente 1/4 do tamanho da matriz original. Em seguida, chama a si mesmo em 3 destes quadrantes. A operação de divisão e escolha de quadrantes a rejeitar tem complexidade constante $\mathcal{O}(1)$. A equação recursiva é:
>
> $$T(N) = 3T\left(\frac{N}{4}\right) + \mathcal{O}(1)$$

(b) (1,0 pontos) Resolva a equação de recorrência do item anterior e compare este algoritmo com uma busca sequencial na matriz.

> **Resposta:** Na notação do teorema Mestre, tem-se $A = 3$, $B = 4$ e $L = 0$. Neste caso, naturalmente, $A > B^L$ e a solução é $N^{\log_B A}$, ou seja,
>
> $$T(N) = \mathcal{O}\left(N^{\log_4 3}\right) \approx \mathcal{O}\left(N^{0,7925}\right)$$
>
> A busca sequencial direta tem complexidade $\mathcal{O}(N)$. Isso significa que o algoritmo recursivo é assintoticamente mais rápido do que a busca sequencial.

3. (2,0 pontos) Em uma árvore, a *profundidade* de um nó é definida como a profundidade de seu pai + 1 se ele não é raiz, ou zero para o nó raiz. A *altura* de uma árvore é a maior profundidade nela encontrada + 1. A classe `NoBinario` implementa um nó de árvore binária. Seu campo `esquerdo` aponta para sub-árvore esquerda e seu campo `direito` aponta para a sub-árvore direita. Implemente uma função em Java que calcula a *altura* de uma árvore binária. Utilize a seguinte assinatura:

```
static int altura(NoBinario n)
```

Onde `n` é o nó raiz da árvore.

> **Resposta:** Como a altura é a profundidade máxima mais um, a altura em um nó é a *maior* altura de todas as sub-árvores mais um. Segue uma solução recursiva que chama a si mesma na sub-árvore esquerda e direita e retorna o maior valor + 1:
>
> ,
> ```
>   static int altura(NoBinario n) {
>         if(n==null) return 0; // Arvore vazia
>         int alturaEsquerda = altura(n.esquerdo);
>         int alturaDireita = altura(n.direito);
>         if(alturaEsquerda>alturaDireita) return alturaEsquerda+1;
>         return alturaDireita+1;
>     }
> ```

4. (2,0 pontos) O algoritmo de Euclides Extendido é um dos algoritmos necessários para a geração de chaves de criptografia RSA. Como o nome sugere, ele é uma *extensão* do clássico algoritmo de Euclides

para calcular o máximo divisor comum. Além de calcular, como o algoritmo de Euclides, o máximo divisor comum r de a e b, ele também calcula um par de *inteiros* x, y tal que:

$$r = ax + by$$

A listagem a seguir mostra uma implementação do Algoritmo de Euclides Extendido. Ela retorna o resultado em um objeto do tipo `egcdResult`, com os campos `r` que contém o valor do máximo divisor comum r e os campos `x` e `y` com o valor final dos inteiros x e y, respectivamente.

,

```
1   static void egcd(int a, int b, egcdResult result)
2       {
3           int xa = 1, ya = 0;
4           int xb = 0, yb = 1;
5           while(b!=0) {
6               int q = a/b;          // Quociente
7               int r = a - q*b;      // Resto
8               a = b;
9               b = r;
10              // Guarda os valores anteriores de xb, yb
11              int txb = xb;
12              int tyb = yb;
13              xb = xa - q*xb;
14              yb = ya - q*yb;
15              xa = txb;
16              ya = tyb;
17          }
18          result.r = a;
19          result.x = xa;
20          result.y = ya;
21      }
```

Este algoritmo supõe *sempre* que $a > b$. Seja $a_i, b_i, x_{ai}, y_{ai}, x_{bi}, y_{bi}$ os valores das variáveis `a`, `b`, `xa`, `ya`, `xb`, `yb` ao *final* da i-gésima iteração do laço iniciado na linha 5, e $a_0, b_0, xa_0, ya_0, xb_0, yb_0$ os valores destas variáveis *antes* da entrada no laço. Da análise do Algoritmo de Euclides tradicional feita em sala são conhecidos os seguintes fatos:

- O algoritmo termina em tempo finito (note que o controle de fluxo deste algoritmo é *idêntico* ao algoritmo convencional).
- $\gcd(a_0, b_0) = \gcd(a_i, b_i)$.
- $b_n = 0 \Rightarrow \gcd(a_0, b_0) = a_n$.

Dado o exposto acima, demonstre que a implementação do algoritmo de Euclides Extendido está *correta*.

Resposta: Do sabido sobre o algoritmo de Euclides, mostra-se que o algoritmo termina em tempo finito e que o valor final obtido, a_n (sendo n a última iteração do laço iniciado na linha 5) é efetivamente o valor de $\gcd(a_0, b_0)$. Resta provar que $a_n = x_{an}a_0 + y_{an}b_0$.

Sabe-se pelas linhas 6, 7, 8 e 9 do algoritmo que

$$a_{i+1} = b_i$$

$$b_{i+1} = a_i - \overbrace{\left\lfloor \frac{a_i}{b_i} \right\rfloor}^{q_{i+1}} b_i$$

Então, se em uma determinada iteração i vale

$$a_i = x_{ai} a_0 + y_{ai} b_0$$
$$b_i = x_{bi} a_0 + y_{bi} b_0$$

Então, *esta relação permanecer válida* na próxima iteração $i + 1$ se:

$$x_{ai+1} = x_{bi}$$
$$y_{ai+1} = y_{bi}$$
$$x_{bi+1} = x_{ai} - q_{i+1} x_{bi}$$
$$y_{bi+1} = y_{ai} - q_{i+1} y_{bi}$$

Esta relação é exatamente a imposta pelas linhas de 11 a 16.

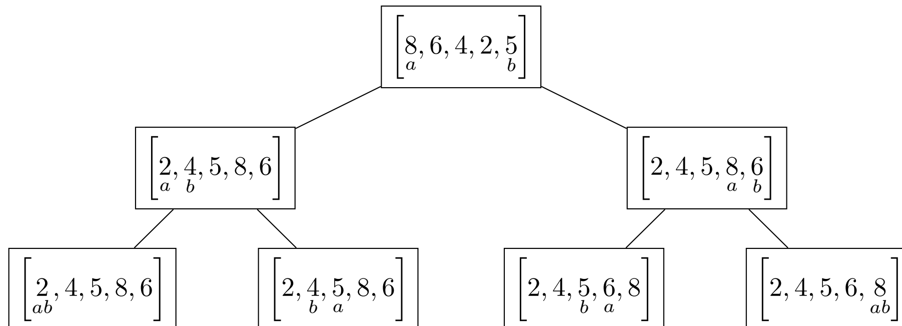
Ora, mas claramente a relação é válida para $i = 0$:

$$a_0 = x_{a0} a_0 + y_{a0} b_0$$
$$b_0 = x_{b0} a_0 + y_{b0} b_0$$

Então, na n-ésima iteração vale $a_n = x_{an} a_0 + y_{an} b_0$ e o algoritmo está correto.

5. (2,0 pontos) A listagem a seguir consiste em uma implementação em Java do algoritmo de ordenação quicksort, empregando a classe `Limites` para armazenar os limites inferior (campo `a`) e superior (campo `b`) da ordenação. Nesta implementação, o elemento pivô é o *último* elemento do vetor.
   (a) (0.5 pontos) Considere a ordenação do vetor $[8, 6, 4, 2, 5]$. Desenhe a *árvore de chamada* da função quicksort para este vetor. Em cada nó, indique o vetor que foi passado para função e os valores do parâmetro `t`. Mostre que a árvore é uma árvore binária. Em que ordem a árvore do item acima é percorrida pelo algoritmo quicksort? (use a terminologia apropriada à teoria de árvores binárias)

Resposta:



A árvore construída é claramente binária, pois cada nó tem no máximo dois descendentes.

O algoritmo visita em cada etapa o nó pai, depois chama a si próprio no filho esquerdo e a si próprio no filho direito. Esta é uma varredura em *profundidade* com ordem *anterior.*

   (b) (1,5 pontos) Reescreva o algoritmo Quicksort com o auxílio da pilha implementada pela classe `PilhaAr` *sem usar nenhuma recursão.*

   Os métodos relevantes da classe `PilhaAr` são:

   - `PilhaAr()`: Construtor, cria uma nova pilha vazia.

- `push(Object x)`: Adiciona o objeto `x` à pilha.
- `Object pop()`: Remove o *último* objeto a ser adicionado (apresentado como valor de retorno) ou lança a exceção `PilhaVazia` se não ouver objetos na pilha.
- `boolean pilhaVazia()`: Retorna verdadeiro se a pilha não tem mais objetos, falso caso contrário.

,

```
static void quicksort(int s[]) {
    quicksort(s, new Limites(0, s.length-1));
}

static void quicksort(int s[], Limites t) {
    if(t.a>=t.b) return;
    int p=s[t.b];     // pivot
    int l=t.a;
    int r=t.b-1;
    while (l<=r) {
        while ((l<=r)&&(s[l]<=p)) l++;
        while ((l<=r)&&(s[r]>=p)) r--;
        if (l<r) {
            int temp=s[l];
            s[l]=s[r];
            s[r]=temp;
        }
     }
     s[t.b]=s[l];
     s[l]=p;
     quicksort(s, new Limites(t.a, (l-1)));
     quicksort(s, new Limites((l+1), t.b));
}
```

Resposta: Como visto em curso, é possível percorrer uma árvore binária em ordem anterior sem recursão com o uso de uma pilha. Por exemplo, o código a seguir imprime os nós de uma árvore em ordem anterior sem recursão:

,

```
static void imprimeAnterior(NoBinario n) {
   PilhaAr p = new PilhaAr();
   p.push(n);
   while(!p.vazia()) {
       n = (NoBinario)p.pop();
       if(n!=null) {
           System.out.println(n.dado);
           p.push(n.direito);
           p.push(n.esquerdo);
       }
   }
}
```

O mesmo princípio pode ser aplicado a uma reimplementação do quicksort:

,

```
static void quicksort(int s[]) {
    PilhaAr pilha = new PilhaAr();
    pilha.push(new Limites(0,s.length-1));
    while(!pilha.vazia()) {
       Limites t = (Limites)pilha.pop();
       if(t.a>=t.b) continue;
       int p=s[t.b];
       int l=t.a;
       int r=t.b-1;
       while (l<=r) {
```

```
                while ((l<=r)&&(s[l]<=p)) l++;
                while ((l<=r)&&(s[r]>=p)) r--;
                if (l<r) {
                    int temp=s[l];
                    s[l]=s[r];
                    s[r]=temp;
                }
            }
            s[t.b]=s[l];
            s[l]=p;
            pilha.push(new Limites(t.a, (l-1)));
            pilha.push(new Limites((l+1), t.b));
        }
    }
```

Formulário

Somas de sequências:

$$\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}, \quad \sum_{i=0}^{n-1} i^2 = \frac{n^3}{3} - \frac{n^2}{2} + \frac{n}{6}, \quad \sum_{i=0}^{n-1} i^3 = \frac{n^4}{4} - \frac{n^3}{2} + \frac{n^2}{4},$$

$$\sum_{i=0(a\neq 1)}^{n-1} a^i = \frac{1-a^n}{1-a}, \quad \sum_{i=0(a\neq 1)}^{n-1} ia^i = \frac{a - na^n + (n-1)a^{n+1}}{(1-a)^2}.$$

Solução de equações de recorrência pelo ***Master Theorem***:

A equação:

$$T(N) = A \cdot T\left(\frac{N}{B}\right) + cN^L,$$

com $T(1) = \mathcal{O}(1)$, tem solução

$$T(N) = \begin{cases} \mathcal{O}\left(N^{\log_B A}\right) & \text{se } A > B^L \\ \mathcal{O}\left(N^L \log N\right) & \text{se } A = B^L \\ \mathcal{O}\left(N^L\right) & \text{se } A < B^L \end{cases}$$

Códigos-fonte de apoio

Classe `PilhaAr`

```
public class PilhaAr {
   public class PilhaVazia extends java.lang.RuntimeException {}
   private Object arranjo[];
   private int topo;

   public PilhaAr() {
      arranjo = new Object[16];
      topo = -1;
   }

   public void push(Object x) {
      if (++topo == arranjo.length) dupliqueArranjo();
      arranjo[topo] = x;
   }

   public Object pop() {
      if (topo < 0) throw new PilhaVazia();
      return(arranjo[topo--]);
   }

   public boolean pilhaVazia() {
       return topo<0;
   }

   private void dupliqueArranjo()  {
      Object na[] =
              new Object[2 * arranjo.length];
      for (int i=0; i<arranjo.length; i++) na[i] = arranjo[i];
      arranjo = na;
   }
}
```

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 `egcdResult`

```
  public class egcdResult {
      int r, x, y;
  }
```

Classe `Limites`

```
public class Limites {
    int a;
    int b;
    public Limites(int a, int b) {
        this.a = a;
        this.b = b;
    }

}
```

Classe `Hash`:

```
public class Hash {
    class DataNotFound extends java.lang.RuntimeException {}

    class Pair {
        int key; Object value;
        Pair(int key, Object value) {
            this.key = key; this.value = value;
        }
    }

    int a, b, size, count;
    final int p = 11;
    Pair[] entries;

    public Hash() {
        entries = new Pair[size=4]; // Tamanho inicial: 4
        java.util.Random rng = new java.util.Random();
        //a = rng.nextInt(p); b = rng.nextInt(p);
        a = 7; b = 4;
    }

    int getHash(int key) {
        int h = ((a*key + b)%p)%size;
        if(h<0) h += size;
        return h;
    }

    public void setData(int key, Object data) {
        int i = getHash(key);
        while(entries[i]!=null && entries[i].key!=key)
            i = (i+1)%size;
        if(entries[i]==null) {
            entries[i] = new Pair(key,data);
            count++; // nova entrada
            checkCount();
        }
        else entries[i].value = data;
    }

    public Object getData(int key) {
        int i = getHash(key);
        while(entries[i]!=null && entries[i].key!=key)
            i = (i+1)%size;
        if(entries[i]==null) return null;
        else return entries[i].value;
    }

    void checkCount() { // Verifica se h   necesside de ampliar o hash
        if(count*5 > size*4) { // Limite em 80% de ocupacao
            size *= 2;
            Pair[] old = entries;
            entries = new Pair[size];
            count = 0;
            for(Pair e : old) {
                if(e==null) continue; // pula entradas vazias
                this.setData(e.key, e.value); // Insere na tabela nova
            }
        }
    }

    public void removeData(int key) {
        int i = getHash(key);
        while(entries[i]!=null && entries[i].key!=key)
            i = (i+1)%size;
        if(entries[i]==null) throw new DataNotFound();
        count--;
        int j=i;
        while(true) {
            entries[i] = null;
            boolean skip = true;
            while(skip) {
                j = (j+1)%size;
                if(entries[j]==null) return;
                int k = getHash(entries[j].key);
                if(i<=j) skip = (i<j)&&(k<=j);
                else skip = (i<k)||(k<=j);
            }
            entries[i] = entries[j];
            i = j;
        }
    }
}
```