Prova 2 — PMR3201 1o. semestre 2016

Prof. Thiago Martins

Instruções: Escreva o nome e o número USP na folha de papel almaço. Os códigos fornecidos na seção "Códigos-fonte de apoio" podem ser referenciados em qualquer resposta sem necessidade de reprodução.

1. (2,5 pontos) A classe `NoListaLigada` representa um nó que armazena inteiros. A propriedade `value` contém o valor armazenado e a propriedade `next` contém uma referência para o próximo nó da lista. Considere o problema de se construir uma lista ligada em ordem crescente, na qual cada elemento é menor ou igual ao subsequente, com o conteúdo de duas listas preexistentes que estão em ordem crescente. Use a seguinte assinatura:

   ```
   static NoListaLigada CombinaOrdenados(NoListaLigada a, NoListaLigada b)
   ```

   onde `a` e `b` são as listas ordenadas a serem combinadas. A função deve retornar uma referência para o nó inicial da lista resultante. Seja N a quantidade de elementos da lista `a` e M a quantidade de elementos da lista `b`. Um algoritmo correto de ordem $\mathcal{O}(N+M)$ vale 2,5 pontos, algoritmos corretos com ordem inferior valem 1,5 pontos.

Resposta: A idéia do algoritmo linear é essencialmente a mesma do problema "produto interno" (vide P2 PMR2300 2015), ou seja, varrer sequencialmente as duas listas escolhendo o nó com o menor. A dificuldade aqui está na inicialização, quando uma das listas ou ambas podem ser nulas, e no final, quando é necessário concatenar ao final da lista construída o que sobrou de uma das listas. A solução que se segue *destrói* as listas originais:

```
,
    static NoListaLigada CombinaOrdenados(NoListaLigada a, NoListaLigada b) {
        NoListaLigada head;
        if(a!=null) {
            if(b!=null) {
                if(a.value < b.value) {
                    head = a;
                    a = a.next;
                } else {
                    head = b;
                    b = b.next;
                }
            } else return a;
        } else return b;
        NoListaLigada x = head;
        while(a!=null && b!=null) {
            if(a.value<b.value) {
                x.next = a;
                x = x.next;
                a = a.next;
            } else {
                x.next = b;
                x = x.next;
                b = b.next;
            }
        }
        if(a!=null) x.next = a;
```

```
        else x.next = b;
        return head;
    }
```

2. (2,5 pontos) A classe `java.io.BufferedReader` permite ler arquivos linha-a-linha. Para tanto ela conta com o método `readLine()`. A cada chamada, `readLine()` retorna uma `String` com uma nova linha do arquivo, ou `null` quando não há mais linhas a serem lidas. Escreva uma função em java que mostra as n *últimas* linhas de um arquivo, na ordem em que elas são retornadas por `readLine()`.

   Use a seguinte assinatura:

   ```
   static void mostraUltimasLinhas(int n, java.io.BufferedReader arquivo)
   ```

   Onde `n` é o número de linhas e `arquivo` é o objeto da classe `BufferedReader` que representa o arquivo cujas n últimas linhas deve-se mostrar. Use na classe `BufferedReader` apenas o método `readLine()`. Para mostrar uma `String` use o método `System.out.println`.

   Descreva o comportamento da sua função se o arquivo tem menos do que n linhas.

   Resposta: É necessário armazenar os n últimos valores retornados por `readLine()` e recuperá-los na forma em que foram obtidos. A classe `FilaAr` fornecida nos códigos-fonte de apoio propicia este comportamento. É necessário retirar elementos da fila quando esta atinge a quantidade de linhas desejada.

   ,
   ```
   static void mostraUltimasLinhas(int linhas, BufferedReader arquivo) {
        FilaAr f = new FilaAr();
        String s;
        while((s = arquivo.readLine())!=null) {
            f.enqueue(s);
            if(f.tamanho() > linhas) f.dequeue();
        }
        while(f.tamanho()>0)
            System.out.println(f.dequeue());
    }
   ```

   Se o arquivo em questão possui menos de n linhas, este código mostra todo o conteúdo do arquivo.

3. (2,5 pontos) Em uma tabela de hash com encadeamento aberto linear, elementos são armazenados em um vetor na *primeira* posição disponível a partir da sua posição ideal, correspondente ao valor da função de hash aplicada à sua chave. O *deslocamento* de um elemento é definido como a diferença entre a posição ideal de um elemento e sua posição efetiva. Note que esta distância é calculada de forma *cíclica*. Por exemplo, seja uma tabela de hash com 4 posições na qual a função de hash é dada por $h(k) = ((9k + 1)\%13)\%4$. Nesta tabela os elementos de chaves $(1, 2, 3)$, cujos hashes são todos 2, são armazenados da seguinte forma: $[3, -, 1, 2]$. Neste caso o deslocamento do elemento de chave 1 é 0, o de chave 2 é 1 e o de chave 3 é 2.

   A classe `HashTable` implementa uma tabela de hash com encadeamento aberto linear. O campo `entries` é o vetor que armazena elementos em objetos da classe `Pair`, que por sua vez contém tanto o valor armazenado (no campo `value`) quanto a chave (no campo `key`. O método `getHash(int key)` retorna o valor de hash da chave em `key`, um valor entre 0 e `entries.length` $- 1$. O método `setData(int`

`key, Object data)` adiciona à tabela o objeto apontado por `data` sob a chave indicada por `key`. O metodo `getData(int key)` retorna o objeto armazenado sob a chave `key` ou `null` se não há na tabela tal chave.. O método `void checkCount()` é um método privado chamado por `setData` que verifica o fator de carga da tabela e, se este excede um determinado valor (75%), cria uma nova tabela com o dobro do tamanho e re-insere os elementos da tabela antiga na nova tabela.

Implemente na classe o método `maiorDeslocamento` que retorna o *maior* deslocamento dentre todos os elementos armazenados na tabela ou -1 se a tabela estiver vazia. Use a seguinte assinatura: `public int maiorDeslocamento()`.

Resposta: Em uma tabela de hash bem-formada, o deslocamento de um elemento na i-gésima posição pode ser computado diretamente pela diferença entre o valor i e o hash de sua chave. Em algumas situações, o deslocamento ultrapassa o fim da tabela, de forma que a posição em que o elemento está efetivamente armazenado é *menor* que a sua posição ideal, levando a diferenças negativas. O valor correto nestes casos pode ser obtido adicionando-se à diferença o tamanho total da tabela. Assim uma varredura direta da tabela permite obter o maior deslocamento:

,

```
public int maiorDeslocamento() {
   int maior = -1;
   for(int i=0; i<entries.length; i++) {
       if(entries[i]!=null) {
           int diff = i - getHash(entries[i].key);
           if(diff<0) diff += entries.length;
           if(diff>maior) maior = diff;
       }
   }
   return maior;
}
```

4. (2,5 pontos) Em uma árvore binária de busca é possível armazenar adicionalmente em cada nó o tamanho da sua sub-árvore. A classe `NoArvoreBinariaComTamanho` representa um nó em tal árvore. O campo `valor` contém o valor armazenado no nó, o campo `tamanho` contém o a total de elementos da sua sub-árvore (incluindo as sub-árvores esquerda, direita e o nó raiz) e os campos `esquerda` e `direita` contém referências para os nós das sub-árvores esquerda e direita respectivamente.

   (a) (1,5 pontos) Escreva uma função em java que, dada um valor e uma árvore binária de busca em que cada nó contém o tamanho de sua sub-árvore, retorna a quantidade total de elementos na árvore *menores* que o valor. Use a seguinte assinatura:

   `static int ContaMenores(int valor, NoArvoreBinariaComTamanho raiz)`

   Onde `raiz` é o nó raiz da árvore binária e `valor` é o valor para o qual deseja-se determinar a quantidade de elementos

   Resposta: O código é similar ao procedimento de busca em uma árvore binária. De fato, se em uma árvore binária o valor de referência é menor ou igual ao elemento raiz, os elementos *menores* só podem estar na sub-árvore esquerda. Por outro lado, se ele é maior, a quantidade de elementos menores do que ele é igual a quantidade de elementos maiores na sub-árvore direita mais um (o elemento raiz) mais o tamanho da sub-árvore esquerda. O código recursivo abaixo implementa este algoritmo (note que é necessário verificar se exite uma sub-árvore esquerda):

   ,

```
static int ContaMenores(int valor, NoArvoreBinariaComTamanho raiz) {
    if(raiz==null) return 0;
    if(raiz.valor>=valor)
        return ContaMenores(valor, raiz.esquerda);
    int d = 1;
    if(raiz.esquerda!=null)
        d += raiz.esquerda.tamanho;
    return d + ContaMenores(valor, raiz.direita);
}
```

(b) (1,0 pontos) Escreva em notação *big Oh* a complexidade da função em função do número de elementos da árvore N para uma árvore genérica. A complexidade muda se a arvore estiver balanceada?

Resposta: Este código faz no máximo h chamadas recursivas onde h é a *altura* da árvore. Ora, mas em uma árvore binária de busca genérica, $h = \mathcal{O}(N)$. Assim o algoritmo tem complexidade $\mathcal{O}(N)$.

No caso de uma árvore balanceada, tem-se $h = \mathcal{O}(\log N)$, de modo que o algoritmo tem uma complexidade diferente, $\mathcal{O}(\log N)$.

Códigos-fonte de apoio

Classe `NoListaLigada`:

```
public class NoListaLigada {
    public int value;
    public NoListaLigada next;

    NoListaLigada(int value, NoListaLigada next) {
        this.value = value; this.next = next;
    }
}
```

Classe `PilhaAr`

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

   // Cria uma nova pilha
   public PilhaAr() {
      arranjo = new Object[16]; topo = -1;
   }

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

   // Retira um elemento ou lanca PilhaVazia
   public Object pop() {
      if (topo < 0) throw new PilhaVazia();
      return(arranjo[topo--]);
   }

   // Verdadeiro se a pilha esta vazia
   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 `FilaAr`

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

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

   // Cria uma nova fila
   public FilaAr() {
      arranjo = new Object[16];
      count = 0; in = arranjo.length-1; out = 0;
   }

   // Retorna a quantidade de elementos enfileirados
   public int tamanho() {
      return count;
   }

   // Adiciona um novo objeto a fila
   public void enqueue(Object x) {
       if(count == arranjo.length) dupliqueArranjo();
       in = next(in); arranjo[in] = x; count++;
   }

   // Retira um objeto da fila ou lanca ErroFilaVazia
   public Object dequeue() {
      if(count==0) throw new ErroFilaVazia();
      Object x = arranjo[out];
      arranjo[out]=null; out = next(out); count--;
      return(x);
   }

   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;
   }

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

Classe `NoArvoreBinariaComTamanho`

```
  public class NoArvoreBinariaComTamanho {
    public int valor;
    public int tamanho;
    public NoArvoreBinariaComTamanho esquerda, direita;
 }
```

Classe `HashTable`:

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

    class Pair { // Armazena valor e chave
        int key; Object value;
        Pair(int key, Object value) {
            this.key = key; this.value = value;
        }
    }

    int a, b, size;
    int count; // Numero de entradas
    final int p = 2147483647;
    Pair[] entries;

    public HashTable() {
        entries = new Pair[8]; size = 8; // Tamanho inicial.
        java.util.Random rng = new java.util.Random();
        a = rng.nextInt(p-1)+1; b = rng.nextInt(p-1)+1;
    }

    public int getCount() {
        return count;
    }

    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;
    }

    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<k)&&(k<=j);
                else skip = (i<k)||(k<=j);
            }
            entries[i] = entries[j];
            i = j;
        }
    }

    void checkCount() { // Verifica se ha necesside de ampliar o hash
        if(count*4 > size*3) { // Limite em 75% de ocupacao
            Pair[] old = entries;
            size *= 2; entries = new Pair[size]; count = 0;
            for(Pair e : old)
                if(e!=null) // pula entradas vazias
                  this.setData(e.key, e.value); // Insere na tabela nova
        }
    }
}
```