Prova 2 — PMR3201 1o. semestre 2015

Prof. Thiago Martins

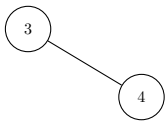
Instruções: Escreva o nome e o número USP na folha de papel almaço.

1. (2.5 pontos) Uma árvore binária de busca é uma árvore binária na qual para qualquer sub-árvore todos os elementos à esquerda da raiz são estritamente *menores* do que a raiz, e todos os elementos à direita da raiz são estritamente *maiores* que a raiz. Todas as inserções em uma árvore binária são feitas em novas folhas. U
   (a) (1.5 pontos) Mostre o resultado, na forma de diagramas de árvores, das seguintes operações, partindo de uma árvore vazia:
      1. Inserção do valor 3.
      2. Inserção do valor 4.
      3. Inserção do valor 5.
      4. Inserção do valor 2.
      5. Inserção do valor 1.
      6. Remoção do valor 3.
      7. Remoção do valor 5.

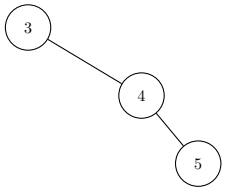
Resposta:

1. Inserção do valor 3.

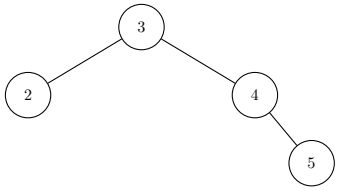
2. Inserção do valor 4.



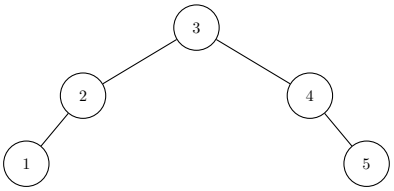
3. Inserção do valor 5.



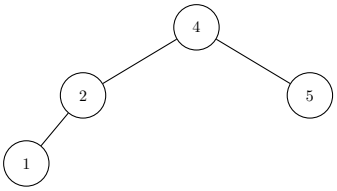
4. Inserção do valor 2.



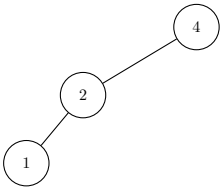
5. Inserção do valor 1.



6. Remoção do valor 5.



7. Remoção do valor 3.

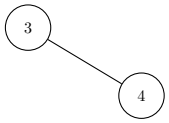


   (b) (1.0 pontos) Uma árvore AVL é uma árvore binária de busca auto-balanceada, na qual a maior diferença de altura entre uma sub-árvore direita e esquerda é 1. Esta propriedade é mantida em operações de inserção e remoção através de *rotações*, nas quais o nível hierárquico de um nó e um de seus descendentes é permutado. Repita o item anterior considerando agora uma árvore AVL.

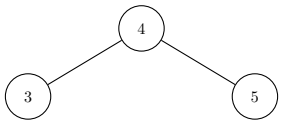
Resposta:

1. Inserção do valor 3.

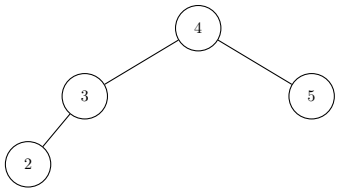
2. Inserção do valor 4.



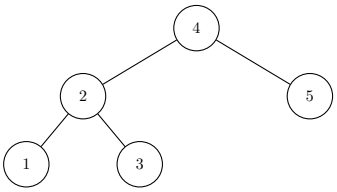
3. Inserção do valor 5.



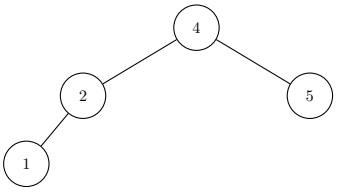
4. Inserção do valor 2.



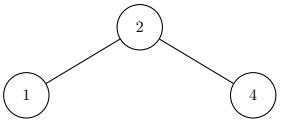
5. Inserção do valor 1.



6. Remoção do valor 3.



7. Remoção do valor 5.



2. (2,5 pontos) Uma lista ligada é implementada com o uso da classe `NoListaLigada` presente no código-fonte de apoio. Considere que esta lista ligada armazena apenas *inteiros*. Para recuperar o valor de um inteiro armazenado em um determinado nó referenciado por uma variável `no` use o código `(int)no.val`.

   (a) (1.0 pontos) Escreva uma função que verifica se uma determinada lista ligada está ordenada em ordem crescente (ou seja, o valor armazenado em um nó é menor ou igual ao subsequente). A função deve obedecer à seguinte assinatura:

,

```
static boolean Ordenada(NoListaLigada head)
```

Onde `head` é uma referência ao *primeiro* elemento da lista ligada. A função deve retornar `true` se a lista está ordenada em ordem crescente, `false` caso contrário. Explique o comportamento de seu código na presença de uma lista vazia.

Resposta: Este código retorna verdadeiro para uma lista vazia

,

```
    static boolean Ordenada(NoListaLigada head) {
        // Lista vazia
        if(head==null) return true;
        int last = (int)head.val;
        head = head.next;
        while(head!=null) {
            if(last>(int)head.val) return false;
            last = (int)head.val;
            head = head.next;
        }
        return true;
```

```
}
```

(b) (1,5 pontos) Escreva uma função em java que insere um inteiro na lista ordenada mantendo a ordenação. A função deve obedecer à seguinte assinatura:

,

```
static NoListaLigada insereOrdenada(NoListaLigada head, int i)
```

Onde `head` é o primeiro elemento da lista e `i` é o inteiro a ser inserido. A função deve retornar o primeiro elemento da lista após a inserção.

Resposta: Nota-se que a função considera o caso em que a inserção ocorre antes do primeiro elemento. Neste caso o valor de retorno é *distinto* de `head`.

,

```
static NoListaLigada insereOrdenada(NoListaLigada head, int i) {
    if(head==null || (int)head.val >= i)
        return new NoListaLigada(i, head);
    NoListaLigada no = head;
    while(no.next!=null && (int)no.next.val < i) no = no.next;
    no.next = new NoListaLigada(i,no.next);
    return head;
}
```

Há naturalmente diversas possibilidades para este código (uma bastante comum é manter duas referências para nós na lista, o atual e o anterior).

3. (2,5 pontos) Uma fila é uma estrutura de dados na qual o primeiro elemento a ser removido é o mais antigo a ser inserido (Vide por exemplo a classe `FilaAr`). Considere o problema de implementar uma Fila a partir de duas Pilhas. Para tanto, complete a implementação da classe `FilaPorPilhas` a seguir, com o corpo dos métodos:

   - `FilaPorPilhas()`: Construtor
   - `enqueue(Object x)`: Adiciona o objeto `x` à fila.
   - `dequeue(Object x)`: Remove o objeto mais antigo a ser inserido, ou lança a exceção `ErroFilaVazia` se não há mais objetos enfileirados.
   - `filaVazia()`: Retorna verdadeiro se não há mais objetos na fila, falso caso contrário.

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

    private PilhaAr p1, p2;

    public FilaPorPilhas() {
        ⋮
    }

    public void enqueue(Object x) {
        ⋮
    }

    public Object dequeue() {
        ⋮
    }

    public boolean filaVazia() {
        ⋮
    }
}
```

Resposta: A solução é manter uma pilha de entrada e outra de saída. Quando a pilha de saída se esgota, o conteúdo da pilha de entrada deve ser *integralmente* transferido para a pilha de saída.

```
,
    public class FilaPorPilhas {
    public class ErroFilaVazia extends java.lang.RuntimeException {}

    private PilhaAr p1, p2;

    public FilaPorPilhas() {
        p1 = new PilhaAr();
        p2 = new PilhaAr();
    }

    public void enqueue(Object x) {
        p1.push(x);
    }

    public Object dequeue() {
        if(p2.PilhaVazia())
            while(!p1.PilhaVazia()) p2.push(p1.pop());
        if(p2.PilhaVazia()) throw new ErroFilaVazia();
        return p2.pop();
    }

    public boolean FilaVazia() {
        return p1.PilhaVazia() && p2.PilhaVazia();
    }
  }
```

4. (2,5 pontos) A classe `HashSep`, cuja listagem encontra-se no código fonte de apoio, implementa uma tabela de Hash com encadeamento em separado. Ela armazena pares de chaves e objetos, representados internamente pela classe `Pair` com os campos `key` e `value` respectivamente. Os pares são armazenados em *buckets*, representados por Listas Ligadas formadas pelo encadeamento de objetos da clase `NoListaLigada`. Há um vetor com M buckets, cujo tamanho é armazenado no campo `size`. Em um determinado momento, há N pares de chaves e objetos armazenados, e a razão entre N e M é o *fator de ocupação* $\lambda = N/M$.

   (a) (1,0 pontos) Escreva um novo método para a classe que recupera o *maior* valor de chave armazenado, ou `java.lang.Integer.MIN_VALUE` se a tabela estiver vazia. O método deve seguir a seguinte assinatura:

   ,
   ```
   public int maiorchave()
   ```

   Descreva a complexidade do método em termos do número N de elementos armazenados e número de "buckets" M.

   Nota: `java.lang.Integer.MIN_VALUE` é o menor valor possível para um inteiro em java.

   Resposta: Não há muito a se fazer a não ser visitar os elementos um por um buscando o maior:

   ,
   ```
   public int maiorchave() {
       int i = java.lang.Integer.MIN_VALUE;
       for(NoListaLigada n : buckets) {
           while(n!=null) {
               if(i<((Pair)n.val).key) i = ((Pair)n.val).key;
               n  = n.next;
           }
       }
       return i;
   }
   ```

   Este código visita cada um dos M buckets, e dentro deles, visita todos os N elementos. Assim, a complexidade é $\mathcal{O}(M + N)$. Na classe apresentada, no entanto, M é *constante*. Assim, uma resposta correta também é $\mathcal{O}(N)$.

   (b) (1,5 pontos) Suponha que você possa fazer uma alteração no *conteúdo* da classe, incluindo campos e métodos pré-existentes. Há alguma maneira de fazer a operação `maiorchave` em *tempo constante*, sem alterar a ordem de complexidade dos métodos `setData` e `getData`? Explique. A solução altera a complexidade de algum outro método? (Você não é obrigado a mostrar código, mas explique *completamente* todos os aspectos da sua alteração - Use código se achar melhor).

   Resposta: É possível manter e atualizar um campo que contém a maior chave inserida. A cada inserção, o valor da nova chave a ser inserido é comparado com o atual valor de maior chave. Se a nova chave é maior, o seu valor substitui o valor antigo. Assim, o valor da maior chave estará disponível em tempo constante. O método de remoção de dados, no entanto, tem a sua complexidade alterada. De fato, quando a chave a ser removida é exatamente a maior chave, então é necessário varrer novamente a estrutura de dados em busca do novo valor de maior chave. É possível provar, no entanto, dentro de suposições razoáveis para o uso da estrutura, que o *caso médio* da remoção ainda tem complexidade $\mathcal{O}(1)$ (basta mostrar que a operação de varredura ocorre na média somente em $1/N$ dos casos).

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 `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 boolean pilhaVazia() {
      return count==0;
   }

   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); count--;
      return(x);
   }
}
```

Classe `NoListaLigada`:

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

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

Classe `HashSep`:

```
  public class HashSep {
    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;
    final int p = 2147483647;
    NoListaLigada[] buckets;

    int getHash(int key) {
        int i = ((a*key + b)%p)%size;
        return i>=0 ? i : i + size;
    }

    public HashSep() {
        buckets = new NoListaLigada[size = 16];
        java.util.Random rng = new java.util.Random();
        a = rng.nextInt(p); b = rng.nextInt(p);
    }

    public void setData(int key, Object data) {
        int i = getHash(key);
        if(buckets[i]==null) {
            buckets[i] = new NoListaLigada(
                    new Pair(key, data),null);
        } else {
            NoListaLigada n = buckets[i];
            while(n!=null && ((Pair)n.val).key!=key ) n=n.next;
            if(n!=null) ((Pair)n.val).val = data;
            else buckets[i] = new NoListaLigada(
                new Pair(key, data), buckets[i]);
        }
    }

    public Object getData(int key, Object data) {
        NoListaLigada n = buckets[getHash(key)];
        while(n!=null && ((Pair)n.val).key!=key ) n=n.next;
        if(n!=null) return ((Pair)n.val).value;
        else return null;
    }

    public void removeData(int key) {
        int i = getHash(key);
        NoListaLigada n = buckets[i];
        if(n==null) throw new DataNotFound();
        if(((Pair)n.val).key == key) {
            buckets[i] = n.next;
        } else {
            while(n.next!=null && ((Pair)n.next.val).key!=key) n=n.next;
            if(n.next==null) throw new DataNotFound();
            n.next = n.next.next;
        }
    }
}
```