Questão 6
Árvore Binária de Busca -

melhor caso: h = [log2 n]
caso típico/médio: h = c . log2 n
pior caso: h = n

Organização hierárquica de chaves ordenáveis
- algoritmo de busca análogo à busca binária
  - a cada chamada recursiva, uma subárvore é “desprezada” e não precisa ser consultada na busca
- árvore possui “espaços” para qualquer nova chave a ser inserida e manter a ordenação relativa

Algoritmos rodam em tempo O(n) no pior caso porque a altura das árvores cresce linearmente no pior caso.

Algoritmos:

```
public Value get(Key key) {
    return get(root, key);
  }

  private Value get(Node x, Key key) {
    // Considera apenas a subárvore que tem raiz x
    if (x == null) return null;
    int cmp = key.compareTo(x.key);
    if     (cmp < 0) return get(x.left, key);
    else if (cmp > 0) return get(x.right, key);
    else return x.val;
  }

  public void put(Key key, Value val) {
    root = put(root, key, val);
  }

  private Node put(Node x, Key key, Value val) {
    // Considera apenas a subárvore com raiz x
    // Devolve a raiz da nova subárvore
    if (x == null) return new Node(key, val);
    int cmp = key.compareTo(x.key);
    if     (cmp < 0) x.left = put(x.left, key, val);
    else if (cmp > 0) x.right = put(x.right, key, val);
    else x.val = val;
    return x;
  }
```

Árvore AVL -
Georgy Adelson-Velskii, Yevgeny Landis, 1962

A árvore AVL é uma árvore autobalanceável, são árvores cujos algoritmos de inserção e remoção mantêm os nós equilibrados.
As operações de rebalanceamento permitem manipular a altura das subárvores sem ferir a ordem das chaves.
A operação base é a rotação (à direita / à esquerda).
A medida de desbalanceamento: fator de equilíbrio (Delta) - diferença entre as alturas das subárvores direita e esquerda de um nó:

Δ > 0: nó “pesado” para a direita
Δ = 0: nó equilibrado (caso ideal)
Δ < 0: nó “pesado” para a esquerda

O caso ideal é todos os nós com Δ = 0 e só é possível com a árvore completa. Em geral, precisamos tolerar nós com Δ = +-1
Propriedade AVL: |Δ| <= 1 para todos os nós

- se |Δ| >= 2, diremos que o nó está desequilibrado

Então como garantir a propriedade AVL ao inserir nós?
Uma estratégia é inserir como em BSTs comuns e procurar por nós desequilibrados, rebalanceando com operações de rotação.
Propriedades auxiliares:

- Se inserimos um nó em uma árvore AVL:
  i) todos os nós desequilibrados são ascendentes desse nó
    - subárvores dos outros nós permanecem inalteradas
  ii) a nova árvore não possui nós com |Δ| >= 3
    - alturas de subárvores aumentam em, no máximo, 1 unidade
    - basta procurar e corrigir nós com Δ = +-2
  iii) o último nó desequilibrado possui filho com Δ = +-1
    - nó +-2 com dois filhos Δ = 0 significa que a árvore não era AVL antes!
    - o filho do lado “pesado” tem Δ = +- 1
  iv) diminuir a altura do último nó desequilibrado (em 1 unidade) reequilibra todos os seus ascendentes
    - desfaz o efeito da inserção nas alturas

Algoritmo de inserção:

```
void inserirAVL(PONT* pp, TIPOCHAVE ch, bool* alterou){
        PONT p = *pp;
        if(!p){
                *pp = criarNovoNo(ch);
                *alterou = true;
        } else {
if(ch == p->chave) *alterou = false;
                else if(ch <= p->chave) {
                        inserirAVL(&(p->esq), ch, alterou);
                        if(*alterou)
                                switch (p->bal) {
                                        case 1 : p->bal = 0;
                                        *alterou = false;
```

```
                                        break;
                                        case 0 : p->bal = -1;
                                        break;
                                        case -1: *pp = rotacaoL(p);
                                        *alterou = false;
                                        break;
                                }
                        } else {
                                inserirAVL(&(p->dir), ch, alterou);
                                if(*alterou)
                                        switch (p->bal) {
                                                case -1: p->bal = 0;
                                                *alterou = false;
                                                break;
                                                case 0 : p->bal = 1;
                                                break;
                                                case 1 : *pp = rotacaoR(p);
                                                *alterou = false;
                                                break;
                                        }
                        }
                }
        }
```

Árvore rubro-negra (RBT) - Rudolf Bayer, 1972
É uma estrutura que tem por nome uma de suas características, pois tem um bit extra em cada nodo que determina se esta é “vermelha” ou “preta”. Cada nó também conta com os campos dados, filho esquerdo, filho direito e pai do nó.

Seguindo o seguinte conjunto de regras uma árvore rubro-negra estará balanceada:

- cada nó possui um valor
- cada novo nó inserido na árvore obedecerá o esquema de menor para o lado esquerdo e maior para o lado direito.
- a cada nó é associado uma cor: vermelha ou preta.
- o nó raiz é sempre preto.
- nós vermelhos que não sejam folhas possuem apenas filhos pretos.
- todos os caminhos a partir da raiz até qualquer folha passa pelo mesmo número de nós pretos.

Na inserção um nó sempre será vermelho (exceto se for o nó raiz). Com isso, a árvore analisa o antecessor da folha, se este for vermelho será necessário alterar as cores para garantir a 6ª regra.

Algoritmo de inserção na árvore rubro-negra:

```
RB-insert(T, z):
1: y ← nil[T]
2: x ← T
3: while x 6= nil[T] do
4: y ← x
5: if key[z] < key[x] then
6: x ← left[x]
7: else
8: x ← right[x]
9: end if
10: end while
11: pai[z] ← y
12: if y = nil[T] then
13: T ← z
14: else
15: if key[z] < key[y] then
16: left[y] ← z
17: else
18: right[y] ← z
19: end if
20: end if
21: left[z] ← nil[T]
22: right[z] ← nil[T]
23: cor[z] ← vermelho
24: RB-insert-fixup(T, z)
```