1. 下記のプログラムは，2 分探索木における再帰を用いない探索と新規データの追加の例である。空欄を適切に埋めて実行し，木の形（バランス）がデータを追加する順番によって変わることを確認せよ。

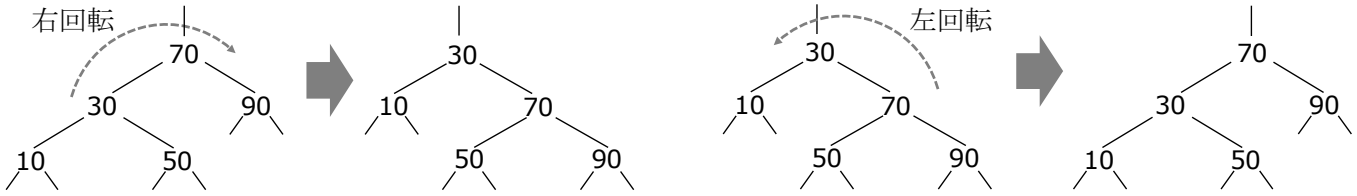
```
/* BSTree.java */
public class BSTree {

  private Node root = null;

  // 内部クラス（クラス内のクラス）の定義
  public class Node {
    private final String key;  // キー
    public int value;          // 値
    private Node left, right;

    private Node(String key) {
      this.key = key;
      this.value = 0;
      left = right = null;
    }
  }

  // キーを指定してノードを探索する
  // add が真なら，ない場合に新たに作って挿入
  public Node find(String key, boolean add) {

    // 木が空の場合は特別
    if (root == null) {
      if (add) root = new Node(key);
      return root;
    }

    // 再帰を用いずにループで根から探索する
    Node n = root;
    while (true) {
      int cmp = key.compareTo(n.key);
      if (cmp < 0) {

        if (n.left ==         ) {

          if (add) n.left =
          return n.left;
        }
        // 枝をたどって降りて（登って？）行く
        n =

      } else if (cmp > 0) {

        if (n.right ==         ) {

          if (add) n.right =
          return n.right;
        }
        n =

      } else {
        // cmp == 0（発見）の場合だけここに来る
        return n;
      }
    }
  }

  // 木の要素をすべて（＝根から）表示する
  public void traverse() {
    traverse(root, 0);
  }

  // 再帰で部分木の要素をたどって表示する
  private void traverse(Node n, int level) {

    if (n == null) return;

    traverse(              , level + 1);

    for (int i = 0; i < level; i++)
      System.out.print("   ");
    System.out.printf("+ %s(%d)%n",
                      n.key, n.value);

    traverse(              , level + 1);
  }
}


/* Program.java */
import java.util.Scanner;

public class Program {

  public static void main(String[] args) {

    BSTree tree =

    Scanner sc = new Scanner(System.in);

    // 簡単な在庫管理プログラム
    while (true) {
      System.out.print("商品名: ");
      String key = sc.next();

      // quit と入力したらプログラム終了
      if (key.equals("quit")) break;

      // キーで探索（内部クラスの利用方法）
      BSTree.Node n = tree.find(key, true);

      System.out.println("在庫数: " +
                         n.value);
      System.out.print("増減数: ");
      int num = sc.nextInt();

      // 値（在庫数）を変更
      n.value += num;

      tree.traverse();
      System.out.println();
    }
  }
}
```

2. 2 分探索木において，探索にかかる平均的な計算量がもっとも小さくなる木の形と，もっとも大きくなる木の形について考察せよ。計算量は根から枝をたどって行く段数に比例することを考慮せよ。

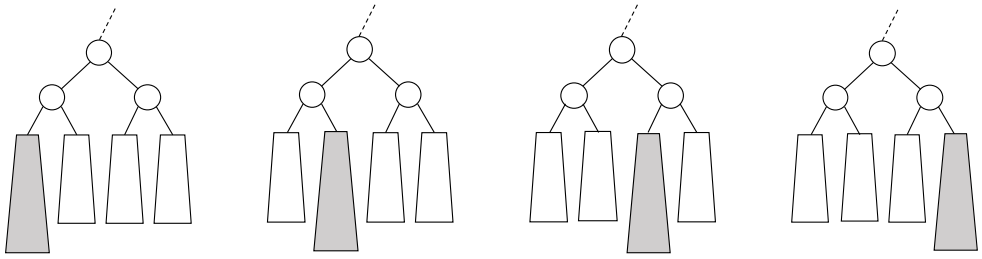
3. 1.のプログラムを参考にして，キーとして日付（java.time.LocalDate），値として文字列（String）のペアを要素とする 2 分探索木のクラスを定義し，今年の祝日とその名前を登録して動作を確認せよ。日付を登録する順番をいろいろ変えて木の形を比較するとよい。なお，LocalDate のインスタンスは下記のようにして生成することができ，compareTo で比較することができる。

```
LocalDate date = LocalDate.of(2017, 1, 1);  // 元日
```

4. 単純な 2 分探索木は，データの登録順によっては木の形が左右対称から大きく崩れ，探索速度が低下する。**平衡木**（AVL 木）では，どのノードを根とする左右の部分木の高さの差も 1 以内に収まるように，下図に示すような回転操作を利用してなるべく平衡（バランス）を保つ。



上の例で「50」のノードは根からの深さが変わっていないことに注意すると，1 回の回転ではバランスを回復できない場合があることがわかる。そこで，バランスが崩れた状態は，2 つの子ノードの下の 4 つの部分木のどれが長くなったかで以下の 4 パターンに整理して対処する。



パターン A　　パターン B　　パターン C　　パターン D

下記は，ノード n の左右の枝の先の長さ（n.left および n.right の部分木の高さ）が 2 以上離れたときに，それを 1 以内に収めるために，どのように回転操作を組み合わせてバランスを維持すればよいか説明した疑似コードである。適切に選択肢を選び，上の 4 パターンがそれぞれどのように処理されるか図示せよ。

```
1: 平衡係数 b ＝（n.left の下の部分木の高さ）−（n.right の下の部分木の高さ）
2: もし，b【 ≦　≧ 】2 ならば，n.left の下が長くなりすぎたので以下を実行
    2.1: さらにもし，n.left.left の下の高さ【 ＜　＞ 】n.left.right の下の高さならば
        2.1.1: n.left.right を持ち上げるために，n.left を軸に【 左　右 】回転する。
    2.2: n.left を持ち上げるために，n を軸に【 左　右 】回転する。
3: もし，b【 ≦　≧ 】-2 ならば，n.right の下が長くなりすぎたので以下を実行
    3.1: さらにもし，n.right.left の下の高さ【 ＜　＞ 】n.right.right の下の高さならば
        3.1.1: n.right.left を持ち上げるために，n.right を軸に【 左　右 】回転する。
    3.2: n.right を持ち上げるために，n を軸に【 左　右 】回転する。
```