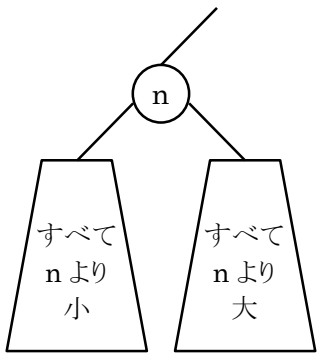
1. **2 分探索木**は，右図に示すように，その中のどのノード n をとっても，その左の部分木には n より小さい要素，右の部分木には n より大きい要素しかないような 2 分木である。このデータ構造は文字通り探索に適する。以下のプログラムは 2 分探索木を構築して，そこから指定のキーを持つノードを探索するものである。空欄を適切に埋めて実行し，探索の平均計算量を考察せよ。

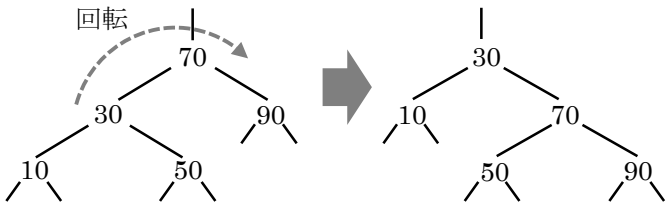



```
/* BSTree.java */
public class BSTree {

  private Node root = null;

  // クラス内での内部クラスの定義
  public class Node {
    private final String key;
    public int data;
    private Node left, right;

    private Node(String key) {
      this.key = key;
      this.data = 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;
    }

    // ループによるノードの探索
    Node n = root;
    while (true) {
      int cmp = key.compareTo(n.key);
      if (cmp < 0) {

        if (n.left ==         ) {

          if (add) n.left =
          return n.left;
        }
        // 木を降りていく
        n =

      } else if (cmp > 0) {

        if (n.right ==         ) {

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

      } else {  // cmp == 0 （発見）
        return n;
      }
    }
  }

  // 木のすべての要素を表示する
  public void traverse() {
    traverse(root, 0);
  }

  // 再帰で部分木の要素を昇順に表示する
  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.data);

    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);

    // 木の構築（単語のカウント）
    while (true) {
      System.out.print("文字列？ ");
      String key = sc.next();

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

      // 内部クラスの利用方法
      BSTree.Node n = tree.find(key, true);
      n.data = n.data + 1;

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

2. 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，どのノードから見た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を保つ。

下記のプログラムに， traverse メソッドや main メソッドを補って実行し，実行画面を示して平衡木のしくみと回転操作について理解せよ。




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

  private Node root = null;

  // 内部クラス
  public class Node {
    private final String key;
    public int data;
    private Node left, right;

    // 左右の子ノードを根とする部分木の高さ
    private int heightL, heightR;

    public Node(String key) {
      this.key = key;
      left = right = null;
      heightL = heightR = 0;
    }

    // 左右の部分木の高さを計算し直す
    private void fixHeights() {
      heightL = heightR = 0;
      if (left != null) {
        heightL = Math.max(left.heightL,
                           left.heightR) + 1;
      }
      if (right != null) {
        // 上記を参考にして作成せよ
      }
    }
  }

  // 回転操作を使って左右のバランスを維持する
  // 引数：部分木の今の根 → 戻り値：新しい根
  private Node rebalance(Node n) {

    // 左右の部分木の高さを比較する
    int b = n.heightL - n.heightR;
    if (b >= 2) {
      // 左の子の下が深すぎる場合
      Node l = n.left;
      if (l != null && l.heightL < l.heightR) {
        // その右の孫の下が深いなら二重回転
        n.left = rotateLeft(n.left);
      }
      n = rotateRight(n);

    } else if (b <= -2) {
      // 右の子の下が深すぎる場合
      Node r = n.right;

      // b >= 2 の場合を参考に作成せよ
    }
    return n;  // 再構成された部分木の根
  }

  // 右の子ノードを部分木の新しい根にする回転操作
  // 引数：部分木の今の根 → 戻り値：新しい根
  private Node rotateLeft(Node n) {
    Node r = n.right;  // 新しい根
    n.right = r.left;
    r.left = n;

    // 再構成した部分木の高さを反映させる
    n.fixHeights();  r.fixHeights();
    return r;
  }

  // 左の子ノードを部分木の新しい根にする回転操作
  private Node rotateRight(Node n) {

    // rotateLeft を参考に作成せよ
  }

  // 再帰を利用したノードの探索と挿入
  public Node find(String key, boolean add) {
    if (root == null) {
      if (add) root = new Node(key);
      return root;
    }
    // 発見したノードを一時退避してバランス調整
    Node ret = find(root, key, add);
    root = rebalance(root);
    return ret;
  }

  private Node find(Node n, String key,
                    boolean add) {
    Node ret; // 見つけたノードを返す

    int cmp = key.compareTo(n.key);
    if (cmp < 0) {
      if (n.left == null) {
        if (add) n.left = new Node(key);
        ret = n.left;
      } else {
        ret = find(n.left, key, add);
      }
      // ノード増加の可能性があるのでバランス調整
      n.left = rebalance(n.left);
      n.fixHeights();

    } else if (cmp > 0) {
      // cmp < 0 の場合を参考に作成せよ
    } else { // cmp == 0（発見）
      ret = n;
    }
    return ret;
  }
}
```