

## アルゴリズムとデータ構造 2022 第13回 演習課題 「2分探索木とAVL木」

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);
}

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

public class Main {

    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();
        }
    }
}

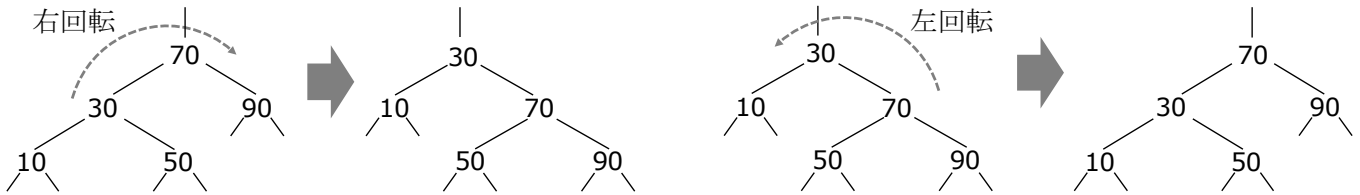
```

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。

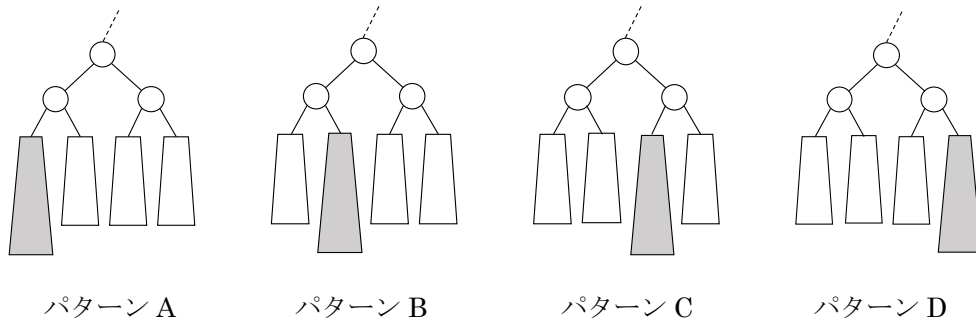
3. 1.のプログラムを参考にして、キーとして日付 (`java.time.LocalDate`)、値として文字列 (`String`) を要素とする 2分探索木のクラスを定義し、祝日の日付とその名前を登録して動作を確認せよ。日付を登録する順番をいろいろ変えて木の形を比較するとよい。なお、`LocalDate` のインスタンスは、コンストラクタを使わずに下記のようにして生成することができ、`compareTo` で順序を比較することができる。

```
LocalDate date = LocalDate.of(2022, 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。この問題を解決するため、**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、バランスが崩れた場合には、左右の子ノードの下にある 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。



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ればよいか説明した疑似コードである。適切に選択枝を選び、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  $b = (n.left \text{ の下の部分木の高さ}) - (n.right \text{ の下の部分木の高さ})$

2: もし、 $b \in [2, \infty) \cup (-\infty, -2]$  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、 $b \in [2, \infty) \cup (-\infty, -2]$  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。

5. 【発展】下記のプログラムは AVL 木の実装例である。未完成な部分をヒントに従って完成させた上で、`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。

```
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;
    }
}
```

6. 【発展】下記のプログラムは、2分探索木における要素の削除の実装例である。1.のプログラムと組み合わせ、適当なデータを使って削除の動作を確認せよ。

```
// ノードを削除する
public void remove(String key) {

    if (root == null) return;

    // 親を保持しながら削除すべきノード n を探索する
    Node n = root;
    Node parent = null; // n の親を保持
    while (true) {
        int cmp = key.compareTo(n.key);
        if (cmp < 0) {
            if (n.left == null) return;

            // 親ノードを保持しながら左の子へと探索
            parent = n;
            n = n.left;

        } else if (cmp > 0) {
            if (n.right == null) return;

            // 親ノードを保持しながら右の子へと探索
            parent = n;
            n = n.right;

        } else { // cmp == 0

            // 削除すべきノード n を発見！
            break;
        }
    }

    if (n == root) {
        // 発見したノード n が根ノードの場合、
        // 根ノードを削除して木を再構築する
        root = removeNode(root);
    } else if (n == parent.left) {
        // n が親ノードの左の子の場合、
        // n を削除して親の左の子を再設定する
        parent.left = removeNode(n);
    } else {
        // n が親ノードの右の子の場合、
        // n を削除して親の右の子を再設定する
        parent.right = removeNode(n);
    }
}

// ノード n を削除してその下の部分木を再構築し、
// n の代わりにその位置に入ったノードを返す
private Node removeNode(Node n) {

    // n に子がなければ、そのまま削除するだけなので、
    // null に置き換えるだけでよい
    if (n.left == null && n.right == null)
        return null;
}
```

```
// n の子が 1 つだけならば、その子（とその下の
// 部分木）が n の位置に代わりに入る
if (n.left != null && n.right == null)
    return n.left;

if (n.left == null && n.right != null)
    return n.right;

// n が左右に子を持つときには、n を削除した
// 位置に n の直前の値を持つノードを入れ替える

// そのために、まず、n の左の子の下の部分木で
// 最大ノード max を探す必要がある
Node max;

// 左の子の下の最大ノードの親ノード p を探す
Node p = findParentOfMax(n.left);

if (p == null) {
    // p が null の場合は、n の左の子自身が最大
    // なので、それをいったん削除する
    max = n.left;
    n.left = removeNode(n.left);
} else {
    // そうでない場合は、p の右の子が最大ノード
    // なので、p の右の子をいったん削除する
    max = p.right;
    p.right = removeNode(p.right);
}

// いったん削除した最大ノード max を n の位置に
// 入れ替えて、n の子を引き継ぐ
max.left = n.left;
max.right = n.right;

// 元の n の位置には、max が入ったことを返す
return max;
}

// ノード n の下から最大ノードを探し、
// そのノードの親ノードを返す
private Node findParentOfMax(Node n) {

    // n に右の子がなければ、n 自身が最大ノード
    // なので、親ノードは不明として返す
    if (n.right == null) return null;

    // 最大値を探すので、右へ子をたどっていく
    // 右の子の右の子がなければ、右の子が最大値
    // になるから、自身がその親であり戻り値である
    while (n.right.right != null)
        n = n.right;
    return n;
}
```