応用Javaプログラミング
第3回

-- Generic Searching（続）--

-- Visitorデザインパターン --

-- 継承（extension）--

-- コンポジション（composition）–

-- 委譲（delegation）--

2分探索木に提供してほしいその他の操作

(1) 木の高さを返す　　(2) 木のunbalance度を返す　　などなど

ポイント：これらの操作は前の6つに比べて使われる頻度が低い．
応用によっては全然使われない可能性もある．
ゆえに，これらの操作を提供するためにメソッドを用意するのは
適切ではない．
一方，これらの操作を不可能にするのもよくない．

妥協案：
(a) 2分探索木に外部からの visitor を受け入れられるようにしておき，
(b) 外部がvisitorに2分探索木の処理方法を教え，
(c) visitorが2分探索木に受け入れられた後，実際に処理を行い
始める
ようにする．

妥協案の利点：データ構造（i.e.，2分探索木）と処理を分離できる．
結果として，2分探索木クラスを変更せずに様々な処理が可能になる．

BTVisitorインターフェース（自作）

```
public interface BTVisitor<K,V>
        Object visitNull( ) ;// nullの処理を定めるメソッド

        //あるノードにあるデータdata，そのノードの左の部分木を処理した
        //結果leftValue，およびそのノードの右の部分木を処理した結果
        //rightValueから，そのノードを処理した結果を返すためのメソッド
        Object visitNode(Object leftValue, Object rightValue,
                         Map.Entry<K,V> data) ;
}// BTVisitor<T,V>インターフェースの終わり
```

BTAcceptorインターフェース（自作）

```
public interface Acceptor<K,V> {
        // BTVisitor<T,V>を受け入れるためのメソッド
        void accept(BTVisitor<K,V> visitor ) ;
}// BTAcceptor<K,V>インターフェースの終わり
```

継承 vs. コンポジション

すでに作成した BinSearchTree<K,V>クラスから訪問可能な2分探索木を表すクラスを作る方法として下記の2つが考えられる:

① 継承(extension): 継承は "is-a"関係を表す. すなわち, 子クラスのインスタンスを親クラスのインスタンスとして解釈しても理にかなうときに, 継承を使うべき. たとえば, "a dog is an animal"なので, Dogクラスが Animalクラスを継承すべき.

② コンポジション(composition): コンポジションは "part-of"関係を表す. すなわち, あるクラスAのインスタンスが複数のパーツを持ち, その1つのパーツとしてクラスBのインスタンスを含めても理にかなうときに, コンポジションを使うべき. たとえば, "a room is part of a house"なので, Roomクラスのインスタンスを Houseクラスのインスタンスの1つのパーツ(i.e., フィールド)にすべき. ここで注意すべきことは, 「Houseがなくなると, Roomもなくなる」ことである.

BTVisitor付き2分探索木（継承版）

```
package genericSearching ;

//Kはデータのキーの型，Vはデータの値の型.
//BTVisitorで処理した結果を Object型 にしたので，どんな応用でも使える.
public class VisitableBinSearchTree2<K,V> extends BinSearchTree<K,V>
          implements BTAcceptor<K,V> {
    private BTVisitor<K,V> visitor ; //visitor

    public VisitableBinSearchTree2(Comparator<K> comparator) //引数あり
    {    super(comparator) ;   }

    public accept(BTVisitor<K,V> v)
    {    this.visitor = v ;   } //BTVisitorを受け入れるためのメソッド.

     //visitorを使って，木を処理するメソッド
    public Object traverse( ) {  return traverse( getRoot( ) ) ;  }
    private Object traverse(BinSearchTreeNode<K,V> start) {
        if (start == null) return visitor.visitNull( ); //空の木であるとき
        else { //空ではない木のとき
            Object left = traverse( start.getLeft( ) ) ;
            Object right = traverse( start.getRight( ) ) ;
            return visitor.visitNode( left, right, start.getData( ) ) ;
        }
    }    }
```

visitorに教えておいた処理方法で左右部分木の処理結果とこのノードのデータに基づいて処理を行い，その結果を返す.

VisitableBinSearchTree2<K,V>型のtreeの高さを求めるには

```
//2分探索木 tree の高さを計算するためのvisitorを作って，treeに渡す．
tree.accept(new BTVisitor<Integer,String>() {
    public Integer visitNull( ) {
        return 0  ;
    }
    public Integer visitNode( Object leftValue, Object rightValue,
            Map.Entry<Integer,String> data ) {
        return Math.max( (Integer)leftValue, (Integer)rightValue ) + 1 ;
    }
});

System.out.println("height: " + (Integer)tree.traverse()) ; //高さの計算　と　結果の表示
```

高さが整数なので，
戻り値の型は Integer型．

Integerは，intに対応するObject型の子孫クラス．

原始型（メソッドを持たない）

参照型（メソッドを持つ）

VisitableBinSearchTree2<K,V>型のtreeをpostorder順にたどって，データを表示するには

```
//2分探索木 treeをpostorder順にたどっていくためのvisitorを作って，treeに渡す．
tree.accept(new BTVisitor<Integer,String>() {
    public Void visitNull( ) {
        return null  ;
    }
    public Void visitNode( Object leftValue, Object rightValue,
            Map.Entry<Integer,String> data ) {
        System.out.println( data.getKey() + ":" + data.getValue() ) ;
        return null ;
    }
});

tree.traverse(); //実際にたどる
```

木をたどりながらデータを表示するので戻り値が要らない．

VisitableBinSearchTree2<K,V>クラスのtraverseメソッドの書き方より，visitorを使ってtreeをinorderとかpreorder順にたどれない．

Voidは，voidに対応するObject型の子孫クラス．

原始型（メソッドを持たない）

参照型（メソッドを持つ）

BTVisitor付き2分探索木（コンポジション版）

```
package genericSearching ;

//Kはデータのキーの型，Vはデータの値の型.
//BTVisitorで処理した結果を Object型 にしたので，どんな応用でも使える.
public class VisitableBinSearchTree3<K,V> implements BTAcceptor<K,V> {
    private BinSearchTree<K,V> tree ;
    private BTVisitor<K,V> visitor ; //visitor

    … // ソースコードを参照

}
```

treeもvisitorも訪問可能な2分探索木のパーツ

これは明らかにおかしい．訪問可能な2分探索木とvisitorの間の関係は“part of”関係よりも“uses a”関係．

結論：この例では，コンポジションよりも継承を利用すべき．

“uses a”関係は集約（aggregation）という．

テスト用クラス

```
package genericSearch ;

public class StringSearch2 {
    public static void main(String[ ] args) { //テスト用のmainメソッド
        //① キーの型がIntegerで，値の型が Stringである空の2分探索木treeを作る
        //② nをキーボードから入力して，1,2,…,nの順列をランダムに作る
        //③ キーが1,2,…,nのデータを②で作った順にtreeに挿入する
        //④ treeの高さを求める
        //⑤ treeをpostorder順にたどって，データを表示する
        //⑥ 検索と削除操作をテストする.
    }
}
```

VisitableBinSearchTree2<K,V>クラスを利用したバージョン.

VisitableBinSearchTree3<K,V>クラスを利用したバージョンは
ほぼ同じで，StringSearch3.java にある.

データ型がString型である空の2分探索木treeを作る

```
VisitableBinSearchTree2<Integer,String> tree =
    new VisitableBinSearchTree2<Integer,String>(
        new Comparator<Integer>() {
            public int compare(Integer a, Integer b) {
                return a – b ;
            }
        }
    ) ;
```

nをキーボードから入力して，1,2,…,nの順列をランダムに作る

```
int n ;
Scanner keyboard = new Scanner(System.in) ;
System.out.println("#data:") ;
n = keyboard.nextInt();
int[ ] array = new int[n] ;
for (int i = 0 ; i < n ; i++) array[i] = i + 1;
Random rand = new Random() ;
for (int i = 0 ; i < n ; i++) {
    int j = rand.nextInt(n - i) ;    int temp = array[n - 1 - i] ;
    array[n - 1 - i] = array[j] ;    array[j] = temp ;
}
```

キーが1,2,…,nのデータを②で作った順にtreeに挿入する

```
for (int i = 0 ; i < n ; i++) {
    tree.insert(new AbstractMap.SimpleEntry<Integer,String>(array[i],
                                                    i + "-th string"));
}
```

treeの高さを求める

```
tree.accept(new BTVisitor<Integer,String>() {
    public Integer visitNull( ) {
        return 0  ;
    }
    public Integer visitNode( Object leftValue, Object rightValue,
            Map.Entry<Integer,String> data ) {
        return Math.max( (Integer)leftValue, (Integer)rightValue ) + 1 ;
    }
});
System.out.println("height: " + (Integer)tree.traverse()) ;
```

treeをpostorder順にたどって，データを表示

```
tree.accept(new BTVisitor<Integer,String>() {
    public Void visitNull( ) {
        return null  ;
    }
    public Void visitNode( Object leftValue, Object rightValue,
            Map.Entry<Integer,String> data ) {
        System.out.println( data.getKey() + ":" + data.getValue() ) ;
        return null ;
    }
});
tree.traverse();
```

検索と削除操作をテストする

```
//検索と削除操作をテスト
n = rand.nextInt(n) ;
System.out.println("now testing " + n);
System.out.println(tree.search(n)) ; //検索
tree.delete(n); //削除
System.out.println(tree.search(n)) ; //検索
```

Visitorデザインパターン

ポイント:データ構造(たとえば，2分探索木)と処理を分離する.

Visitor役(訪問者役): データ構造(の要素)を処理するためのインターフェース(API)を定める役.
例では，**BTVisitor**インターフェースがこの役をつとめている.

ConcreteVisitor役(具体的訪問者役): **Visitor**のAPIを実装した役.
例では，**StringSearch2**クラスの**main**メソッド内の**無名クラス**がこの役をつとめている.

Element役(要素役): **Visitor役**の訪問先を表す役．訪問者を受け入れる**accept**メソッドを持ち，その引数には**Visitor役**が渡される.
例では，**BTAcceptor**インターフェースがこの役を担っている.

ConcreteElement役(具体的要素役): **Element役**のAPIを実装する役．例では，**VisitableBinSearchTree2**クラスがこの役を担っている.

ObjectStructure役(オブジェクトの構造役): **Element役**の集合を扱う役．例では，**VisitableBinSearchTree2**クラスがこの役をつとめている.

演習課題

2分探索木のノードvのunbalance度：vがnullであれば，そのunbalance度が0である．vがnullではなければ，そのunbalance度がその左にある子孫数とその右にある子孫数の差の絶対値である．

2分探索木のunbalance度：その木のノードの最大unbalance度である．

プログラム作成課題：StringSearch2クラスに，作った2分探索木のunbalance度を求める処理を追加せよ．