2021年度「プログラミング言語」配布資料 (2)

五十嵐 淳

2022年10月02日

1　2分探索木 in Java (java/bst/)[1]

まずは，2分木をどのように表現するかを考え，探索と挿入について考える．削除はあとで．

1.1　クラスとインターフェースを使った2分木の表現

Java (というかオブジェクト指向言語) の場合，データを構成する主要な手段はクラス/オブジェクトである．ここでは `leaf` と `branch` に相当するオブジェクト別のクラス (名前を `Leaf` と `Branch` にしよう) で定義する．`branch` は左右の部分木と整数を持つので，それらを `Branch` クラスのインスタンス変数で表すことにする．2分木一般は `Leaf` または `Branch` クラスのオブジェクトになるので，インターフェース `BinarySearchTree` を用意し，変数がどちらでも格納できるようにする．

ファイル名: BinarySearchTree.java

```
public interface BinarySearchTree {
    // signatures for find, insert, delete to come
}
```

ファイル名: Leaf.java

```
public class Leaf implements BinarySearchTree {
    // no instance variables

    // Constructor
    public Leaf() {
        // nothing to initialize
    }
}
```

ファイル名: Branch.java

```
public class Branch implements BinarySearchTree {
    // instance variables to hold a number and subtrees
    private BinarySearchTree left;
```

[1]括弧内はオンライン資料のパス名

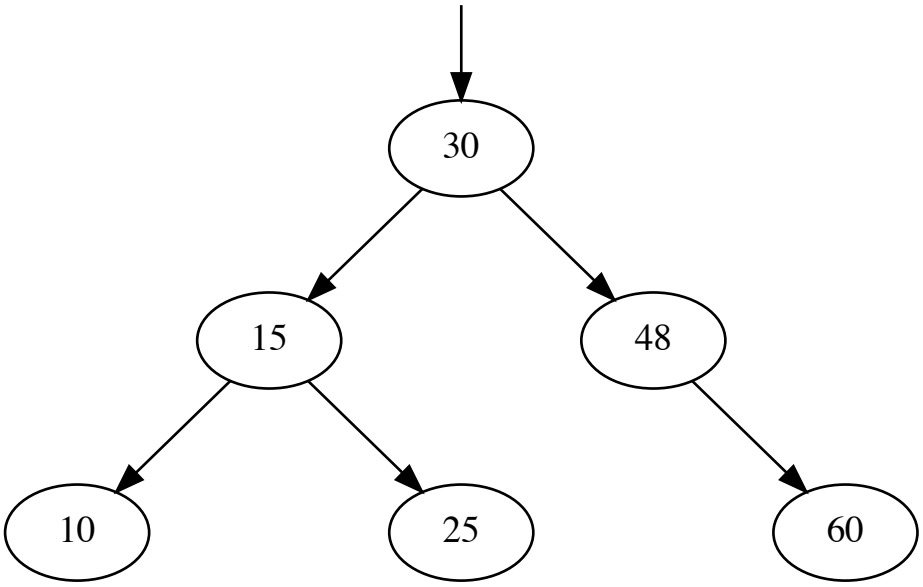
```
    private int v;  // standing for a value
    private BinarySearchTree right;

    // Constructor
    public Branch(BinarySearchTree left, int v, BinarySearchTree right) {
        this.left = left;
        this.v = v;
        this.right = right;
    }
}
```

Branch のコンストラクタのパラメータ変数の名前が (わざと) インスタンス変数の名前と衝突している．このようなコンストラクタの本体で left はコンストラクタの引数を表し，インスタンス変数は「隠れて」しまう．(これを**シャドウイング (shadowing)** という．) 隠れてしまったインスタンス変数は this.left という記法で示すことができる．(もちろん，コンストラクタの引数の名前を変えてインタンス変数は this．なしで

```
    public Branch(BinarySearchTree _left, int _v, BinarySearchTree _right) {
        left = _left;
        v = _v;
        right = _right;
    }
```

ように定義しても問題はない．)

この定義を使って，前に示した



のような 2 分木は以下のように構成できる[2].

```
BinarySearchTree t1 = new Branch(new Leaf(), 10, new Leaf());
BinarySearchTree t2 = new Branch(new Leaf(), 25, new Leaf());
BinarySearchTree t3 = new Branch(t1, 15, t2);
BinarySearchTree t4 = new Branch(new Leaf(), 60, new Leaf());
BinarySearchTree t5 = new Branch(new Leaf(), 48, t4);
BinarySearchTree t6 = new Branch(t3, 30, t5);
```

Branch オブジェクトが，t1 などの BinarySearchTree 型の局所変数に代入されていたり，BinarySearchTree 型の引数を期待するコンストラクタに Leaf オブジェクトが渡されていることに注目してほしい．これが，インターフェース BinarySearchTree を設定したことの御利益 (のひとつ) である．

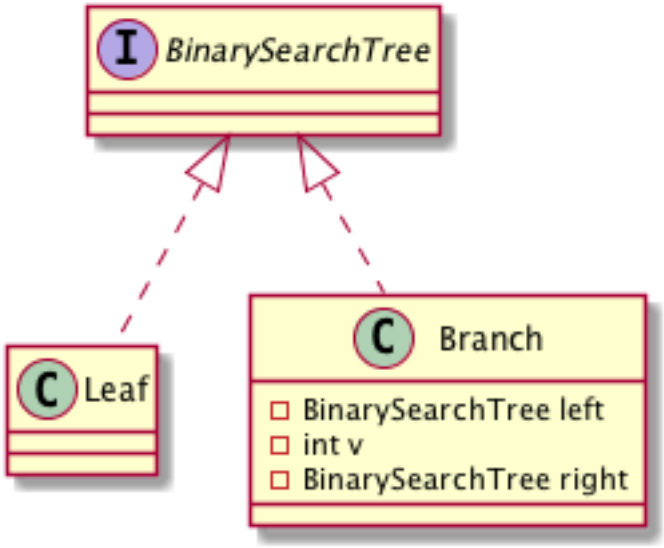
なお，この時点では何もメソッドを定義していないので，構成した 2 分木に対して何もできない．また，2 分探索木の条件を満たさないようなものでも作れてしまうことには注意してほしい．

new を使ってオブジェクトを生成するとメモリが消費される．Leaf オブジェクトはインスタンス変数も持たず，複数の new Leaf() を使って生成された Leaf オブジェクト達を区別する理由はほとんどなく，ある意味メモリの無駄遣いである．これを問題として対処する方法はいくつか考えられる．できれば後述したい．

[2] このテストコードはオンライン資料では Main クラスの main メソッドにテストコードの一部として書かれている．

1.1.1 UML とクラス図

複数のクラスを使ってプログラムを書く時には，クラス同士の関係を図示しておくと (人間の) 理解・伝達の助けになる．世の中には**統一モデリング言語 (Unified Modeling Language, UML)** という，ソフトウェアの設計の様々な側面を記述するための，プログラミング言語によらない (といっても「クラス」はオブジェクト指向言語特有の概念だが) 記法がある．UML には様々な図の記法が用意されていて，クラス同士の関係を図示するものは**クラス図 (class diagram)** という[3]．以下は，2 分探索木のクラス図である．(BlueJ でのクラス・インターフェースの表示は，このクラス図記法に倣っている．)



クラス図の書き方について本稿で詳しく扱うことはしないが，上の図の読み方だけ示しておく．* 箱ひとつがクラス・インターフェースを表す．* 箱は 3 つの部分に区切られており，上から順に，クラス・インターフェースの名前，インスタンス変数，メソッドを並べる．I がインターフェース，C がクラスを表している (が，この表記は一般的でないように思う)．インターフェースの名前は斜体で書くのが一般的．* 赤い四角は，private を示す．(上の図にはまだないが) 緑の丸は，そのメソッドやインスタンス変数が public であることを示す．* 点線矢印は実装関係 implements を示す．

1.2 探索

木に対する操作はオブジェクトのメソッドとして実装する．典型的な木の操作の記述は，木が `leaf` である場合と `branch` である場合からなる．メソッドとして実装する場合，この場合分けを動的ディスパッチで行うのがオブジェクト指向らしいプログラミングである．つまり，`Leaf` クラスには木が `leaf` である場合の記述のみ，`Branch` クラスには木が `branch` である場合の記述のみをする．

探索のメソッド `find` は，探索する整数を引数とし，それが存在するかを表す真偽値を返すことにする．

まず，インターフェースに `find` のシグネチャを追加する．

[3]他には，オートマトンの図示に用いられる**状態マシン図 (state machine diagram)** (直訳すれば「状態機械図」だが，UML 界以外では状態遷移図と呼ぶことが多い) や，オブジェクト間のデータのやりとりを記述するための**シーケンス図 (sequence diagram)** といったものなどがある．

```
public interface BinarySearchTree {
    boolean find(int n);
}
```

これで，Leaf と Branch にも find を (この引数，返り値で) 定義する必要がでてくる．

次に木が leaf であった時の処理を，Leaf クラスのメソッドとして書く．(以下，追加する部分だけ示す．)

```
    // Leaf クラスに追加
    public boolean find(int n) {
       // n doesn't exist in this BST
       return false;
    }
```

leaf には何のデータも格納されていないので，n が見つからないことを示す false を返している．

次に，木が branch であった時の処理はもう少し面白い (といっても，基本的に擬似コード通りである)．

```
    // Branch クラスに追加
    public boolean find(int n) {
        if (n == v) { return true; }
        else if (n < v) { return left.find(n); }
        else /* n > v */ { return right.find(n); }
    }
```

コンストラクタと違い，パラメータとインスタンス変数の名前は衝突していないので，インスタンス変数も単に left, right, v と書ける[4]．また，最後の else 節は上の条件が成り立たない場合にのみ実行されるが，その時成り立っている条件 n > v (すなわち n != v かつ!(n < v)) をコメントとして書いてみた．

left.find(n); において，インスタンス変数 left (型は BinarySearchTree) には，Leaf オブジェクトか Branch オブジェクトのどちらかが入っているわけだが，動的ディスパッチによって，場合に応じた処理が呼出されることになる．

先程の二分木を構成するプログラムに続けて以下のコードを実行すると，それぞれ true と false になるはずである．(練習問題: find の結果が計算されるまでの計算過程を説明せよ．)

```
boolean test1 = t6.find(30);  // should be true
boolean test2 = t6.find(13);  // should be false
```

ここまでの Java プログラムの動作の理解がおぼつかない人は，ここで一旦 Java の復習をすべし．

[4] this. をつけてもよい．

1.3 挿入

1.3.1 短命データ構造と永続データ構造

挿入 (と削除) は木の形を変える操作である．このように，操作の前後で構造に変化が生じる場合には，大きくわけて以下のふたつの実装方針がありえる．

- 操作前のデータを破壊して，操作後のデータを生成する．通常は，既にあるオブジェクトをできるだけ再利用して，変更が生じた部分だけ形を組替える．この時，組替えは既存のオブジェクトのインスタンス変数に代入を施して変数の内容を書き換えることによって行う．代入を使って変更を行うため，変更前の木構造は破壊されてしまい，変更前後の木は同時に存在できない．
- 変更前の木はそのままにして，変更後を表す **新しい**木を作る．変更前後の木が同時に存在することになり，古い木を使って別の操作を行うこともできる．(コンストラクタでインスタンス変数の初期化を行う以外の) 代入操作を使わないことになる．ただし，完全に新しい木をゼロから作るのは時間・空間 (メモリ) の両観点から無駄が多いので，通常，変更前後のデータは，一部を共有する (のがふつうである)．

前者を **短命 (ephemeral) データ構造**，後者を**永続 (persistent) データ構造** という．まず，後者の方法で実装することを考える．

1.3.2 永続データ構造用の挿入

探索の時と同じように，各インターフェース・クラスに，`insert` メソッドを追加する．挿入の結果作られる新しい木が返り値になるようにするため，`BinarySearchTree` インターフェースに追加するシグネチャは以下のようになる．

```
// BinarySearchTree に追加
BinarySearchTree insert(int n);
```

`n` が挿入される新しい要素である．

`Leaf` クラスについては，要するに空の木に何かを挿入した結果の木がどんな形になるかを考え，そのような木を作って返せばよい．

```
// Leaf に追加
public BinarySearchTree insert(int n) {
    // a new singleton tree holding n
    return new Branch(new Leaf(), n, new Leaf());
}
```

`Branch` クラスについては，少し戸惑うかもしれない．まず，挿入しようとしている整数が今注目している節点に発見された場合は木に変化はない．といっても，擬似コードのように何も返さないで処理を終えてよいわけではなく，挿入前の木を返さなければいけない．挿入前の木，というのは，メソッドが呼出された対象の木そのもので，それは `this` という特別な名前でアクセスすることができる．

一方，この節点には `n` が格納されていない場合には，`v` との大小関係によって左右どちからの部分木に挿入することになる．そのため再帰的に `left.insert(n)` か `right.insert(n)` を呼ぶことになる．ここで大事なのは，この呼出しからは，**挿入後のあるべき姿の部分木が返ってくる** *(はずだ)* ということである．呼出し側とし

ては，これを使って，今注目している Branch 以下の挿入結果を作る必要がある．結果として以下のようなメソッド定義になる．

```
// Branch に追加
public BinarySearchTree insert(int n) {
    if (n == v) {
        return this;
    } else if (n < v) {
        BinarySearchTree newLeft = left.insert(n);
        return new Branch(newLeft, v, right);
    } else /* n > v */ {
        BinarySearchTree newRight = right.insert(n);
        return new Branch(left, v, newRight);
    }
}
```

部分木の挿入結果を newLeft や newRight という局所変数に一旦格納して，それを子とする新しい Branch オブジェクトを作っている．(この変数は，一度書き込んで，すぐに読み出すが，以後使わないので，プログラムの読み易さ以外に変数を用意する必要性は余りない．

```
    } else if (n < v) {
        return new Branch(left.insert(n), v, right);
    } ...
```

と書いても構わないだろう．)

1.3.3 「新しい木」？

最初の説明では，挿入操作によって新しい木が作られるように書いたが，これは実は正確ではない．確かに，Leaf クラスの insert 処理では，新しい葉・節点が作られているが，Branch クラスの (例えば)

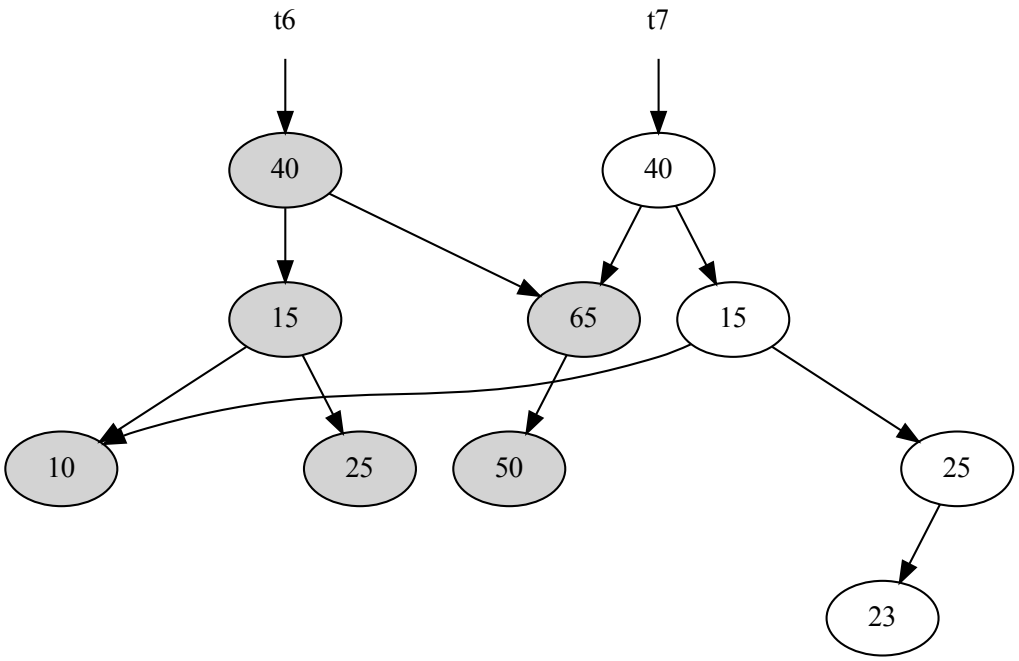
```
BinarySearchTree newRight = right.insert(n);
return new Branch(left, v, newRight);
```

の部分で返されているのは，既存の木の部分木 left が結果の木の一部となっているような木である．つまり変化を生じていない部分については木の再利用をしていることになることに注意してほしい．

例えば，上記の main の続きで

```
BinarySearchTree t7 = t6.insert(23);
```

を実行すると，挿入操作の過程で変化しない部分木が挿入前後の木で共有されている，



のような構造ができる．(灰色が `t6` に属する部分である．挿入のために辿った経路上の branch を表すオブジェクトが新たに作られている．)

1.3.4 代入による (あやまった) 破壊の防止

Java ではインスタンス変数への代入 (正確には初期化以外の再代入) を，その宣言に `final` という修飾子をつけることで禁止することができる．この場合，`Branch` クラスは

```
public class Branch implements BinarySearchTree {
    // instance variables to hold a number and subtrees
    private final BinarySearchTree left;
    private final int v;
    private final BinarySearchTree right;

    ...
}
```

のようになる．コンストラクタの定義はそのままでよい．こうすることによって，メソッドの中で `v = 10;` などと代入するとコンパイル時のエラーになる．`final` 修飾子を使うことで，データ構造を**変更不能 (immutable)** にすることができる．

1.4 ここまでの要点と考察

- 木を構成する二種類のデータ (leaf と branch) が別々のクラスのオブジェクトとして表現されている．
- 二種類のデータどちらでも格納できる変数を用意するために，共通のインターフェースとして BinarySearchTree が定義されており，クラスはそれを implements するよう定義される．
- 木の操作で必要なデータの種類による場合わけは動的ディスパッチで実現されている．各クラスは，それが表すデータの場合の処理のみが書かれる．branch の持つ部分木はインスタンス変数として見えるので，木から部分木を取りだす，といった操作は特に明示する必要がない．
- 各クラスには，それが表すデータの場合の処理のみが書かれる結果，異なる操作同士の類似点は見やすいが，ひとつひとつの処理の全体像は定義がクラスにまたがるため，見通しはあまりよくない．ただし，(今回の 2 分木では考えづらいが，一般的には) もしデータの種類が増えるような場合には，新しいクラスを定義すればデータ構造が簡単に拡張できる点では便利である．
- 代入を使わずに操作を実現する場合，入力の木の部分部分に対して再帰的にメソッドを呼出した結果を使って，どのように木を**作るか** を記述するようなプログラムになる．
- この 2 分探索木を使っている Main クラスでは new を使ってオブジェクトを生成しているが，間違えて 2 分探索木ではない構造を作ってしまう可能性があるため，本当は好ましくない．これを防止する手法については後述する．

1.5 削除

最後に削除についてみてみよう．挿入と同様に削除操作についても削除後の新しい木を返すように実装する．

```
// BinarySearchTree に追加
BinarySearchTree delete(int n);
```

Leaf クラスについては，このメソッドが呼出される，ということは，削除対象の n が木にそもそも存在しなかった，ということなので，削除前後で木が変化しない，ということになる[5]．

```
// Leaf に追加
public BinarySearchTree delete(int n) {
    // n is not in the BST, so return the same tree
    return this;    // could be  return new Leaf();  instead
}
```

Branch については思いの他面倒臭い．2 分木中の最小の数を求める操作 (min) が必要なのが理由のひとつだが，他にも，擬似コードを見てみるとわかるように，部分木がどのような形をしているかによる場合分けをするのだが，その部分が案外曲者である．これまで，木の形 (根が leaf か branch か) による場合分けは if を使わず，分岐した後の動作をメソッドとして書くことで行っていたが，この方法は (できなくはないが) 取りづらい．そのため，木が Leaf や Branch であるかを問い合わせるメソッド isLeaf や isBranch を定義し，明示的に if で場合分けを行う[6]．以下に，必要なメソッドのシグネチャのみ示しておく．

[5] 削除対象が存在しなかったので実行時エラーとして実行を中断する，というのもひとつのやり方である．(が，中断の仕方について学ぶのはまだ先の話である．)

[6] 今回は，データの種類が 2 種類だけなので isLeaf が false を返した場合にはそれが Branch であると思ってよいが，一般的にはデータの種類毎に is... を用意することになる．

```
// BinarySearchTree に追加
int min();
boolean isLeaf();
boolean isBranch();
```

今回は特定の言語に依存しないよう isLeaf や isBranch というメソッドを定義したが，Java の場合にはオブジェクトのクラスを判定するための instanceof 演算子がある．instanceof 演算子は〈式〉`instanceof`〈クラス名〉という形式で，〈式〉の値が指すオブジェクトのクラスが〈クラス名〉と等しいかどうかを判定する．例えば `e.isLeaf()` は `e instanceof Leaf` と書くこともできる．〈クラス名〉はインターフェースの名前でも構わない．その場合は，〈式〉の値が指すオブジェクトのクラスがインターフェースと implements 関係にあるかどうかを判定する．

これらのメソッドを使って `detele` は以下のように定義できる．

```
public BinarySearchTree delete(int n) {
    if (n == v) {
        if (left.isLeaf()) {
            if (right.isLeaf()) {
                return new Leaf();
            } else {
                return right;
            }
        } else {
            if (right.isLeaf()) {
                return left;
            } else {
                int m = right.min();
                BinarySearchTree newRight = right.delete(m);
                return new Branch(left, m, newRight);
            }
        }
    } else if (n < v) {
        BinarySearchTree newLeft = left.delete(n);
        return new Branch(newLeft, v, right);
    } else /* n > v */ {
        BinarySearchTree newRight = right.delete(n);
        return new Branch(left, v, newRight);
    }
}
```

後半の `n` と `v` の値が異なる時の処理は `insert` と同様である．

最後に `min` メソッドの定義について．`Leaf` クラスでは，

```
public int min() {
    // there is no minimum number in the BST
```

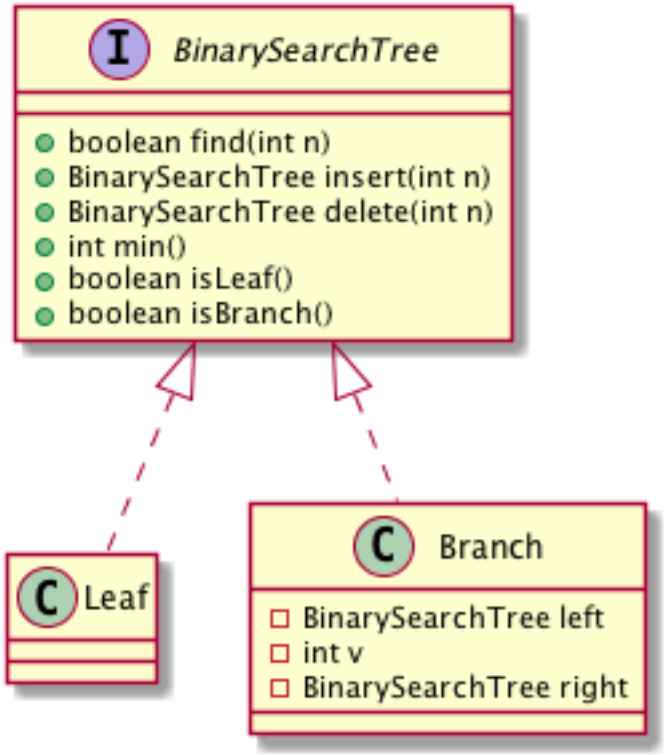
```
        return Integer.MIN_VALUE;
    }
```

と，整数の最小値である Integer.MIN_VALUE を返している．実は (delete の補助メソッドとして使われる限りでは)Leaf に対して，最小値を求める処理が呼ばれることはないので，ここに何を書いておこうと問題はない．

> 中級者向けコメント: むしろ，ここが実行されるのは，あってはならないことなので，return 文の前に assert false; を書いておくべき．assert <Boolean 式>; は，プログラマが <Boolean 式> が true であることを何らかの理由で 表明 (assert) しているものである．別の言い方をすると，これが false になったらプログラムの誤りであることを示している．実際には，true になったら単に通過するが，false になったら AssertionError と呼ばれる実行時エラーでプログラムの処理が異常終了する．assert false; は実行されたら必ず失敗するので，「そもそもここが実行されることはない」というプログラマの表明であると考えられる．

1.6 クラス図 (完成版)

最後にメソッドを含めたクラス図を示しておく．



メソッドのシグネチャは上位インターフェースである BinarySearchTree に書かれているので，Leaf や Branch には繰り返さないことになっている．

2　2分探索木変奏曲 in Java

実装方法のバリエーションとして，

- leaf の表現として null 参照を使う方法
- 短命な 2 分探索木の実装

を紹介する．

2.1　葉を null ポインタで表現する (java/bstNull)

前回の実装方法では，葉を Leaf クラスで，節点を Branch クラスで表現した．ここでは，葉をオブジェクトではなく，null (**null pointer**) を使って表現する方法をみる．

Java における null は，何のオブジェクトも指さない特殊な参照 (ポインタ) の定数を示す表記である．null はオブジェクト参照を格納できる変数であれば，その変数の型に関わらず格納することができる．しかし，何のオブジェクトも指していないので，null を格納した変数を使ってメソッドを呼び出すと，例外 (exception)[7] が発生し，(ふつうは) プログラム全体の実行がその時点で終了してしまう．null は未定義や例外的な値を表すのによく使われる．例えば，(初期化子がなく宣言された) インスタンス変数の初期値は null に設定される．

2 分 (探索) 木の実装において，Leaf オブジェクトに代えて null を使って葉を表してよい理由は以下の通りである．* まず，Leaf オブジェクトにはインスタンス変数がないこと．実質，複数の Leaf オブジェクトは同一視しても問題はないため，予め約束事として定数をひとつ選んでおけばクラスを書かずともその定数で leaf を表現することができる．しかし，一方で Branch オブジェクトなどと同じ変数に格納できてほしいので，整数などを使うことはできない (Java で int とオブジェクト参照の両方を格納できる変数は作れない)．null は Branch が格納できる変数にも格納できるのでうってつけである．

- leaf の他には定数で代替してよいようなものがない．参照のうち null のような特殊な定数はひとつしかないので，null で代替できるクラスは，データ構造を表すクラスの中でひとつだけである．

2.1.1　2 分木構造の定義

このような方針を取ると，クラスがひとつしか必要がなくなるのでインターフェースを設定する必要がなくなる．(これは 2 分木の場合データを構成する要素が 2 種類しかない，という 2 分木の特殊事情である．一般のデータ構造では 3 つ以上の種類のデータを混ぜて使うこともあり，その場合はインターフェースを定義する必要がある．) ということで，BinarySearchTree というクラスを定義するだけになる．そして，その内容は (データの設計に関する部分は) ほとんど Branch クラスと同様である．

```
public class BinarySearchTree {
    private BinarySearchTree left;
    private int v;
    private BinarySearchTree right;

    public BinarySearchTree(BinarySearchTree left, int v, BinarySearchTree right) {
        this.left = left;
```

[7] 例外とは，通常の実行が続けられなくなった状態を示す．例外機構についてはそのうちやりたいと思っています．

```
        this.v = v;
        this.right = right;
    }
}
```

2.1.2 2 分木操作の定義

この方法はクラスがひとつに減ってうれしい反面，もはや**動的ディスパッチで葉か節点かの場合分けをすることができない**．木を使おうとする側が，自分で，変数に格納された木が葉なのか節点なのかを管理して適切な場合分けをする必要がある．以下は，新しい `find` メソッドの定義である．

```
    // BinarySearchTree クラスに追加
    public boolean find(int n) {
        if (n == v) {
            return true;
        } else if (n < v) {
            // Check if the left subtree is a leaf
            if (left != null) {
                return left.find(n);
            } else {
                return false;
            }
        } else /* n > v */ {
            // Check if the left subtree is a leaf
            if (right != null) {
                return right.find(n);
            } else {
                return false;
            }
        }
    }
```

`if` が入れ子になって少し見通しが悪いが，再帰呼出しをする前に，`left != null` や `right != null` によって，部分木が `leaf` かどうかを確かめている．この場合分けは，以前は動的ディスパッチで行っていたものであり，`else` 節の処理は，これまで `Leaf` クラスに書かれていたものである．

ちなみに，`if` が入れ子になって読みにくい，という向きには，

```
            if (left != null) {
                return left.find(n);
            } else {
                return false;
            }
```

の部分を

```
        return (left!=null)&&(left.find(n))
```

と書くのも手である．&& の基本的な意味は「両側の式の値が true の時のみ true になる，それ以外は false」だが，左側の式の値が false であった場合には右側を実行せずに全体の値を false とするため (いわゆる **短絡評価 (short-circuit evaluation)**)，null に対して find メソッドを呼び出したりすることはない．

Main クラスでは以前からほとんど変更なく，そのまま find を呼出しているが，これは変数に格納された木が (null ではなく)Branch であることがわかりきっているからである．一般には再帰呼出しに限らず，find を呼出す前に null かどうかのチェックをする必要がでてくる．

挿入・削除操作については，同じ要領で実装することができる．定義が少し長くなっているが，これは Leaf と Branch に分かれていたコードが一箇所にまとまっているためである．

```
// BinarySearchTree クラスに追加
public BinarySearchTree insert(int n) {
    if (n == v) { return this; }
    else if (n < v) {
        if (left != null) {
            BinarySearchTree newLeft = left.insert(n);
            return new BinarySearchTree(newLeft, v, right);
        } else {
            BinarySearchTree newLeft = new BinarySearchTree(null, n, null);
            return new BinarySearchTree(newLeft, v, right);
        }
    } else /* n > v */ {
        if (right != null) {
            BinarySearchTree newRight = right.insert(n);
            return new BinarySearchTree(left, v, newRight);
        } else {
            BinarySearchTree newRight = new BinarySearchTree(null, n, null);
            return new BinarySearchTree(left, v, newRight);
        }
    }
}
```

削除に関しては，min メソッドは引き続き使うものの，isLeaf や isBranch メソッドが不要になっている．

```
public BinarySearchTree delete(int n) {
    if (n == v) {
        if (left==null) {
            if (right==null) {
                return null;
            } else {
                return right;
            }
        } else {
```

```
            if (right==null) {
                return left;
            } else {
                int m = right.min();
                BinarySearchTree newRight = right.delete(m);
                return new BinarySearchTree(left, m, newRight);
            }
        }
    } else if (n < v) {
        if (left != null) {
            BinarySearchTree newLeft = left.delete(n);
            return new BinarySearchTree(newLeft, v, right);
        } else {
            return this;
        }
    } else /* n > v */ {
        if (right != null) {
            BinarySearchTree newRight = right.delete(n);
            return new BinarySearchTree(left, v, newRight);
        } else {
            return this;
        }
    }
}
```

2.1.3 static メソッドとして実装する (java/bstNullStatic)

上で述べた定義は，動的ディスパッチを使っていないので，最早オブジェクトのメソッドとして定義する必要すらなく，static メソッド[8] (クラスに属するメソッド) として定義してもよい．例えば，`find` の定義は以下のようになる．

```
// BinarySearchTree クラスに追加
public static boolean find(BinarySearchTree t, int n) {
    if (t == null) {
        return false;
    } else if (n == t.v) {
        return true;
    } else if (n < t.v) {
        return find(t.left, n);
    } else /* n > t.v */ {
```

[8]static メソッドは，本文でも述べているように，メソッド名にクラス名をつけて呼び出す．(例えば `Math.pow` は `Math` クラスに属する `pow` という static メソッドである．) つまり，何かのオブジェクトがあって，それに対して呼ぶものではない．ふつうのメソッドの場合，それが定義されているクラスのオブジェクトを操作するために定義しているので，インスタンス変数の読み書きができるが，static メソッドは特にオブジェクトを想定していないので，インスタンス変数の読み書き，`this` の使用はできない．(引数として与えられたオブジェクトのインスタンス変数は読み書きできる—場合がある．)

```
            return find(t.right, n);
        }
    }
```

まず，このメソッドのヘッダには static 修飾子がつけられていて，static メソッドであることが示されている．そして，探索の対象となる 2 分木を，明示的に引数 t として取る形になっている．(ふつうのメソッドで定義した時と違い) find の対象となる t は BinarySearchTree オブジェクトか，(葉を表す) null であるので，まず最初に t が null かどうかのチェックを行って処理を分岐させている．t.v や t.left は t のインスタンス変数の値を読み出すための表記である．最初に null のチェックをすることに対応して，部分木が葉かどうかのチェックはせずに再帰呼出しをしている．この定義は，擬似アルゴリズムや OCaml での find の定義に近い．

static メソッドを (他のクラスから) 呼出す際には，

```
BinarySearchTree t = new BinarySearchTree(null,10,null);
boolean b = BinarySearchTree.find(t, 15);
```

のように，find の前に，それが定義されているクラス名をつけて呼び出す．(find 中の再帰呼出しは同じクラス内からの呼出しなのでつけなくてもよい．)

> 注: この実装方法は，null を使うこととは直接関係はない．オブジェクトに，isBranch のように種類を問い合わせるメソッドを与えれば同じようなことができる．

2.2 短命な 2 分探索木 (java/bstMutable)

次に考えるバリエーションは，短命な 2 分探索木である．この場合，永続的な場合の実装と異なり，木の操作は既にある節点オブジェクトのインスタンス変数を代入文で書き換えて新しい木構造を作る．別の言い方をすると，必要最小限のオブジェクトの生成しか行わないよう操作を実現する，ともいえる．既にふれたように，その代償として，一旦挿入や削除を行うと，操作前の 2 分探索木は破壊されて失われてしまうことになる．

まず，挿入操作についての新しい定義を見ていこう．

Leaf クラス

```
    public BinarySearchTree insert(int n) {
        // a new singleton tree holding n
        return new Branch(new Leaf(), n, new Leaf());
    }
```

Branch クラス

```
    public BinarySearchTree insert(int n) {
        if (n == v) {
            // do nothing
        } else if (n < v) {
            left = left.insert(n);
        } else /* n > v */ {
            right = right.insert(n);
```

```
        }
        return this;
    }
```

実は，Leaf クラスについては変更の必要がない．Leaf オブジェクトが Branch オブジェクトに変化するわけだが，この変化はオブジェクトのクラスの変化であって，インスタンス変数の変化では表現できないため，新しい Branch オブジェクトを生成して，それを返すことになる．

一方，Branch クラスについては，部分木への挿入を行った結果 left.insert(n) (や right.insert(n)) をインスタンス変数 left (や right) に代入することで，木構造を組替えている．最終的には this を返しているのは，このメソッドは変更後の木の根を返すことになっているためである．Branch クラスだけを見ると，this を返すのではなく，(返り値型を void にして)，代入だけして，何も返さない (最後を return; とする) メソッドとして実現してもよいように思えるが，Leaf の方では新しいオブジェクトを返すのが自然なので，それに合わせて this を返す実装になっている[9]．

削除についての変更点も insert と同じようなものである．Branch クラスの主な変更のみ示す．

```
    public BinarySearchTree delete(int n) {
        if (n == v) {
            if (left.isLeaf()) {
                if (right.isLeaf()) {
                    return new Leaf();
                } else {
                    return right;
                }
            } else {
                if (right.isLeaf()) {
                    ...
                } else {
                    // copy the number next to n (the minimum number
                    // in right) and delete it from right.
                    int m = right.min();
                    v = m;
                    right = right.delete(m);
                    return this;
                }
            }
        } else if (n < v) {
          ...
        } else /* n > v */ {
          ...
        }
    }
```

[9] Branch と Leaf の insert が別の返り値型を持つとコンパイル時のエラーになってしまう．

13 行目から 18 行目にかけてが，削除すべき節点に子がふたつあった場合の処理である．インスタンス変数 v と right を代入によって変更している．また，Leaf クラスと合わせて定義を見てみると，delete 操作はオブジェクト生成をほとんどしていない (削除すべき節点に子がない場合の処理で，insert とは逆に，節点から葉への変化のために Leaf オブジェクトを生成しているだけである) ことがわかる．