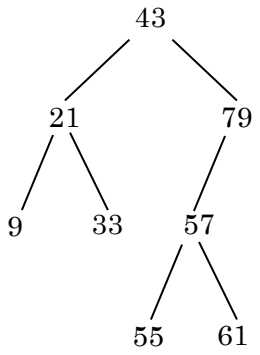
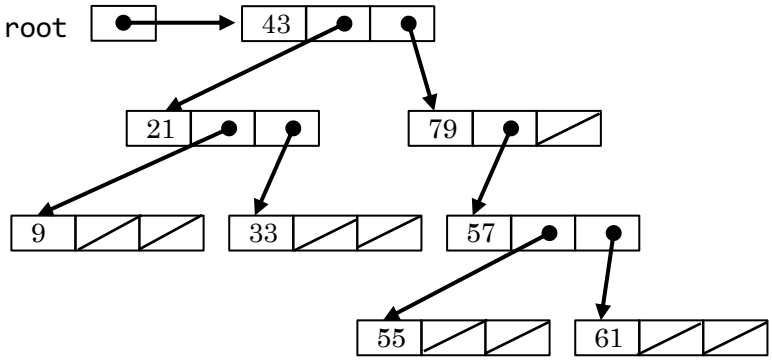
アルゴリズムとデータ構造 2023　第 12 回 演習課題　「木構造と 2 分木」

1. 下図のように各要素が 1 つの“親”を持ち，階層的に連結されたデータ構造を**木（ツリー）構造**という。各要素を**ノード（節）**，連結を**リンク（エッジ，枝）**，図では一番上にある親がないノードを**根（ルート）**，子がないノードを**葉（リーフ）**という。特に，木構造の中で子の数が 2 つ以下であるものを **2 分木**と呼ぶ。
　下記のプログラムは 2 分木の構造を理解するためのものである。図に示した 2 分木を構築する処理を main メソッドに追加せよ。さらに，全ノードを行きがけ順（先行順），通りがけ順（中間順），帰りがけ順（後行順）のそれぞれで表示させてみよ（★♪◆のそれぞれの位置で要素を表示させればよい）。





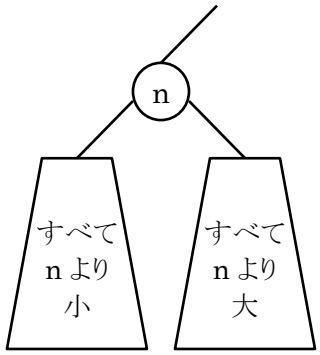

```
/* Node.java */
public class Node {
   public int data;
   public Node left;   // 左の子
   public Node right;  // 右の子

   public Node(int data) {
      this.data = data;
      left = right = null;
   }
}
```

```
/* Tree.java */
public class Tree {
   public Node root; // 根

   public Tree() {
      root = null;
   }

   public void traverse() {
      traverse(root);
   }

   // 深さ優先探索で（部分）木を巡回する
   public void traverse(Node n) {
      if (n == null) return;

      /* ★ */
      traverse(n.left);
      /* ♪ */
      traverse(n.right);
      /* ◆ */
   }
}
```

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

   public static void main(String[] args) {

      // 木の構築
      Tree tree = new Tree();
      tree.root = new Node(43);

      tree.root.left =

      tree.root.right =









      tree.traverse();
   }
}
```

※ オブジェクト指向的には好ましくないが，木の構造が分かりやすいように，すべて public としている。

2. **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（サブツリー）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。実は前問のプログラムの 2 分木は，2 分探索木である。



下記のプログラムは，前問のプログラムのクラス Tree を拡張し，木から指定のキーを持つノードを探索するメソッド search を追加したものである（Node クラスは，前問のものを利用する）。空欄を適切に埋めて探索が行われるようにし，連結リストと比較した場合の 2 分探索木の利点を考察せよ。

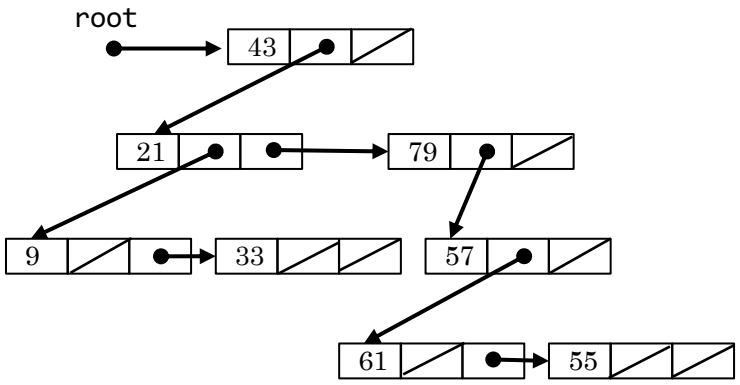
```
public class Tree {

    /* 1.のクラス Tree に，以下を追加する */

    // 木全体から key を探索する
    public Node search(int key) {
        return search(key, root);
    }

    // ノード n の下の部分木から，再帰的に key を探索する
    public Node search(int key, Node n) {
        if (n == null)
            return null;

        if (key < n.data) {
            return search(key,          );
        }
        if (key > n.data) {                  // 上の if に return があるので else が不要
            return search(key,          );
        }
        return                               // key<n.data でも key>n.data でもないとは？
    }
}

/* 動作を検証するための Main.java */
public class Main {
    public static void main(String[] args) {

        // 木の構築
        Tree tree = new Tree();

        /* ここで 2 分探索木の条件を満たす木を構築する（例えば 1.の木は条件を満たす） */

        // 探索処理
        System.out.print("探索整数 -> ");
        int key = new java.util.Scanner(System.in).nextInt();
        Node found = tree.search(key);
        System.out.println("見つかりま" + (found != null ? "した。" : "せんでした。"));
    }
}
```

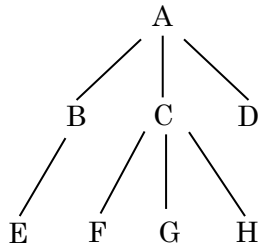
3. 2.のプログラムを修正してクラス E を要素とするジェネリッククラス Tree<E>を定義し，文字列を要素とする 2 分探索木を構築して動作を確認せよ。ただし，E が compareTo を持つという条件を示すために Tree<E>の定義の冒頭は下記のように記述する（親がインタフェースでも extends と書くのが決まり）。

```
public class Tree<E extends Comparable> // より正しくは <E extends Comparable<? super E>>
```

4. 【発展】子の数に制限がない木構造（**多分木**）を実現する方法は，いろいろなものがあるが，ここでは各ノードに「自分の第一子」と「自分の次のきょうだい」へのポインタを持たせる方法を紹介する。この方式では 1.の 2 分木は左下の図のように表すことができる。あるノードに子ノードがいくつあっても内部構造では 2 つのポインタで十分なので，これは多分木を 2 分木で表現しているともいえる。

下記のプログラムの空欄を適当に埋めて右下の図の木を構築するプログラムを作成せよ。さらに，完成した木からノード C を削除する処理も加えよ。その際，C の子は C の親に直接つなげるものとする。





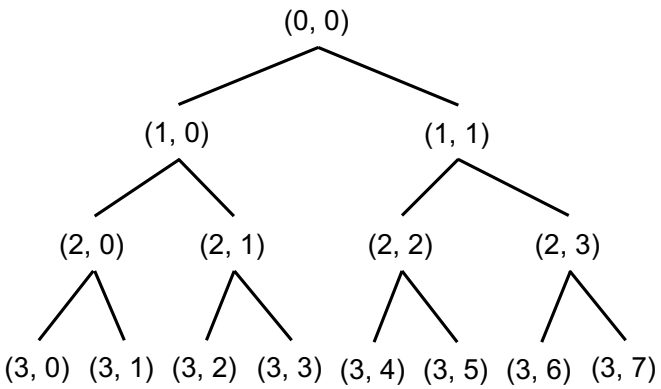

```
/* Node.java */
public class Node {
   public String label;
   public Node firstChild;  // 第一子
   public Node nextSibling; // 次の弟妹

   public Node(String label) {
      this.label = label;
      firstChild = nextSibling = null;
   }
}
```

```
/* Tree.java */
public class Tree {
   public Node root;

   public Tree() {
      root = null;
   }

   public void traverse() {
      traverse(root);
   }

   // 2 分木とみなせば全て再帰で走査可能だが,
   // ループを使うほうが空間計算量が小さい
   public void traverse(Node n) {

      System.out.println(n.label + " ");

      for (Node cn = n.firstChild;
           cn != null; cn =            ) {

         traverse(cn);
      }
   }
}
```

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

   public static void main(String[] agrs) {
      // 木の構築
      Tree tree = new Tree();
      Node A = new Node("A");
      Node B = new Node("B");
      Node C = new Node("C");
      Node D = new Node("D");
      Node E = new Node("E");
      Node F = new Node("F");
      Node G = new Node("G");
      Node H = new Node("H");

      tree.root = A;









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

      // ノード C の削除




      tree.traverse();
   }
}
```

5. 【発展】木の構造は，テキスト表示だけでは分かりづらいので，グラフィックス（Swing ライブラリ）を用いて木の形（樹形図）を表示するプログラムを作成しよう。以下は 1.のプログラムの変更部分である。メソッド drawTree は，ノード n から再帰的に部分木を描画するもので，引数の dan と retsu はノード n が画面表示では何段目の何列目の位置にくるかを表す（下図参照）。このプログラムの動作を理解した上で，左右対称に枝が伸びる木の表示を目指して修正し，表示例を示せ。色なども調べて自由に使ってよい。

```
/* Main.java */
import java.awt.Graphics;
import javax.swing.*;

public class Main {

   public static void main(String[] args) {

      Tree tree = new Tree();

      /* ここで 2 分木を構築する */
      /* 自由な 2 分木を構築してよい */

      // ウィンドウ（600×600 画素）を作る
      JFrame frame = new JFrame("2 分木");
      frame.setLayout(null);
      frame.setBounds(0, 0, 600, 600);
      frame.setDefaultCloseOperation(
            JFrame.EXIT_ON_CLOSE);

      // 描画用のパネルを作る
      JPanel panel = new JPanel() {
         // 描画イベントへの対応
         public void paint(Graphics g) {
             // 根ノードから木を描画する
             drawTree(g, tree.root, 0, 0);
         }
      };
      // パネルをウィンドウに載せる
      panel.setBounds(0, 0, 600, 600);
      frame.add(panel);

      // ウィンドウ（frame）を画面表示する
      frame.setVisible(true);
   }

   // 木（部分木）を描画する
   static void drawTree(Graphics g,
         Node n, int dan, int retsu) {

      if (n == null) return;

      // 段と列から xy 座標を求めてラベルを描く
      int x = xpos(dan, retsu);
      int y = ypos(dan, retsu);
      g.drawString("" + n.data, x, y);

      // 親があれば親へのリンク（線）を描く
      if (dan != 0) {
         int px = xpos(dan - 1, retsu / 2);
         int py = ypos(dan - 1, retsu / 2);
         g.drawLine(px, py, x, y);
      }

      // 左右の部分木を再帰的に描画する
      drawTree(g, n.left,
               dan + 1, retsu * 2);
      drawTree(g, n.right,
               dan + 1, retsu * 2 + 1);
   }

   // 段と列から x 座標を計算する
   static int xpos(int dan, int retsu) {
      // この計算式を工夫してほしい
      return retsu * 50 + 50;
   }

   // 段と列から y 座標を計算する
   static int ypos(int dan, int retsu) {
      // この計算式を工夫してほしい
      return dan * 50 + 50;
   }
}
```



ノードの位置による (dan, retsu) の値