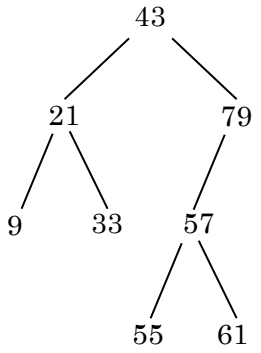
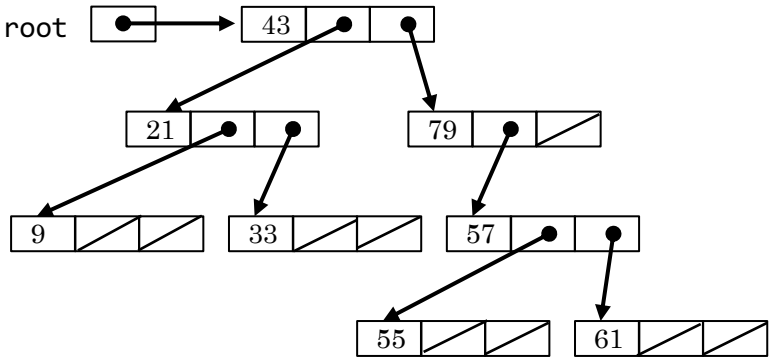
1. 下図のように各要素が 1 つの“親”を持ち，階層的に連結されたデータ構造を**木（ツリー）構造**という。各要素を**ノード（節）**，連結を**リンク（エッジ，枝）**，図では一番上にある親がないノードを**根（ルート）**，子がないノードを**葉（リーフ）**という。特に，木構造の中で子の数が 2 つ以下であるものを**2 分木**と呼ぶ。
   下記のプログラムは 2 分木の構造を理解するためのものである。図に示した 2 分木を構築する処理を main メソッドに追加せよ。さらに，全ノードを行きがけ順（先行順），通りがけ順（中間順），帰りがけ順（後行順）のそれぞれで表示させてみよ（★♪◆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





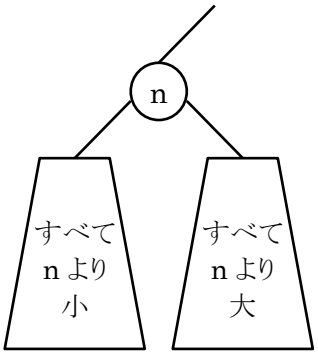

```
/* Node.java */
public class Node {
   public int data;
   public Node left;   // 左の子
   public Node right;  // 右の子

   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);
      /* ◆ */
   }
}
```

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

   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 1.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， 1.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 1.のものをそのまま利用する）。空欄を適切に埋めて探索が行われるようにし，この探索アルゴリズムの利点を考察せよ。

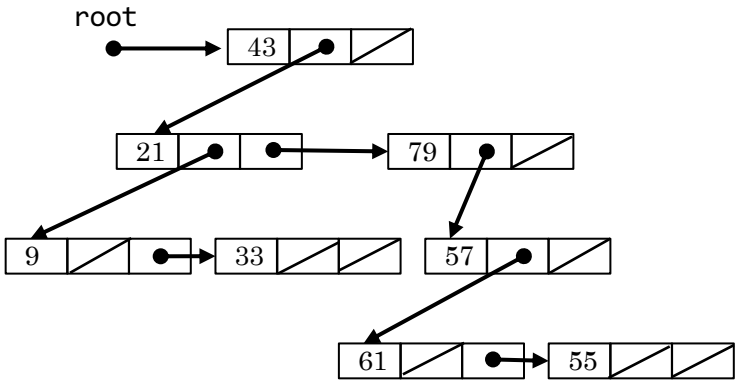
```
public class Tree {

    /* 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 Program.java */
public class Program {
    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。"));
    }
}
```

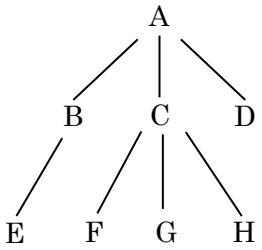
3. 2.のプログラムを修正してクラス E を要素とするジェネリッククラス Tree<E>を定義し，文字列を要素とする 2 分探索木を構築して動作を確認せよ。ただし，E が compareTo を持つという条件を示すために Tree<E>の定義の冒頭は下記のように記述する（親がインタフェースでも extends と書くのが決まり）。

```
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);
      }
   }
}
```

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

   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();
   }
}
```