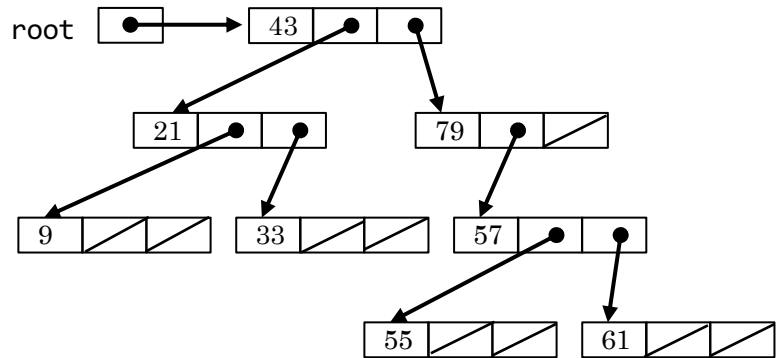
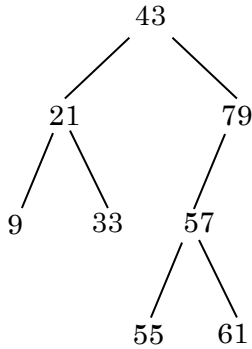


1. 下図のように各要素が 1 つの“親”を持ち、階層的に連結されたデータ構造を木（ツリー）構造という。各要素をノード（節）、連結をリンク（エッジ、枝）、図では一番上にある親がないノードを根（ルート）、子がないノードを葉（リーフ）という。特に、木構造の中で子の数が 2 つ以下であるものを 2 分木と呼ぶ。下記のプログラムは 2 分木の構造を理解するためのものである。図に示した 2 分木を構築する処理を main メソッドに追加せよ。さらに、全ノードを行きがけ順（先行順）、通りがけ順（中間順）、帰りがけ順（後行順）のそれぞれで表示させてみよう（★♪◆のそれぞれの位置で要素を表示させればよい）。



```

/* 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) {

        // 全てのノードの生成
        Node n1 = new Node(43);
        Node n2 = new Node(21);
        Node n3 = new Node(79);
        Node n4 = new Node(9);
        Node n5 = new Node(33);
        Node n6 = new Node(57);
        Node n7 = new Node(55);
        Node n8 = new Node(61);

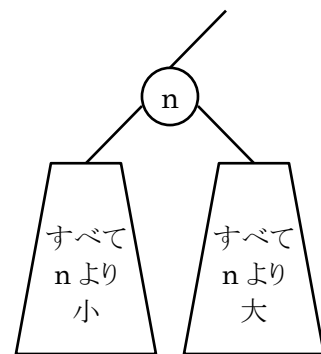
        // 木の構築
        Tree tree = new Tree();
        tree.root =

        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) {
            return search(key,          ); // 上の if に return があるので else が不要
        }
        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 */
import java.util.Scanner;

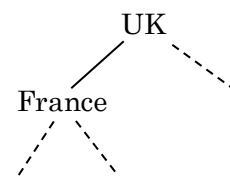
public class Main {
    public static void main(String[] args) {
        Scanner sc = Scanner(System.in);

        // 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) */

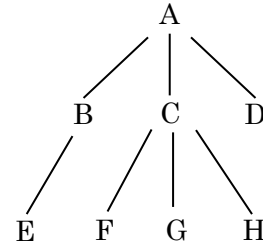
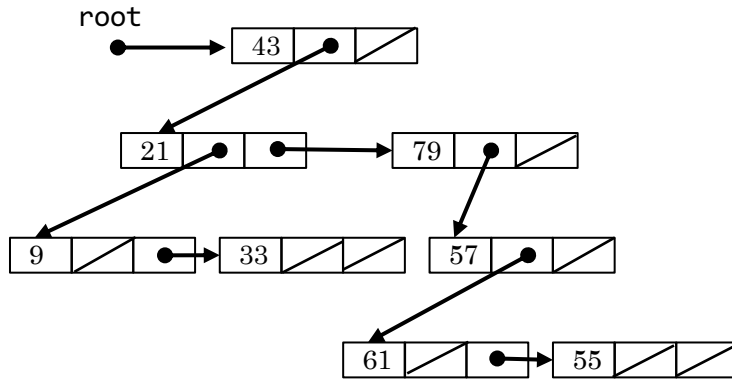
        // 探索処理
        System.out.print("探索整数 -> ");
        int key = sc.nextInt();
        Node found = tree.search(key);
        System.out.println("見つかりま" + (found != null ? "した。" : "せんでした。"));
    }
}
```

3. 文字列を要素とする2分探索木に、「UK」「France」「USA」「Russia」「China」の5つの文字列を、根が「UK」でその左の子が「France」となるように適切に登録した状態を図示せよ。さらに、2.のプログラムを修正してその2分探索木を構成し、全ての要素がうまく探索できたことを実行結果で示せ。



4. 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<E> {
    public E data;
    public Node<E> firstChild; // 第一子
    public Node<E> nextSibling; // 次の弟妹

    public Node(E data) {
        this.data = data;
        firstChild = nextSibling = null;
    }
}

/* Tree.java */
public class Tree<E> {
    public Node<E> 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<E> n) {

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

        for (Node<E> cn = n.firstChild;
             cn != null; cn = _____ ) {

            traverse(cn);
        }
    }
}

```

```

/* Main.java */
public class Main {

    public static void main(String[] args) {
        // 木の構築
        Tree<String> tree = new Tree<>();
        Node<String> A = new Node<>("A");
        Node<String> B = new Node<>("B");
        Node<String> C = new Node<>("C");
        Node<String> D = new Node<>("D");
        Node<String> E = new Node<>("E");
        Node<String> F = new Node<>("F");
        Node<String> G = new Node<>("G");
        Node<String> 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（樹形図）を表示するプログラムを作成してみよう。以下は Processing のプログラムである。Tree クラスのメソッド draw 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、自由に構築した2分木の表示例を示せ。

```
class Node {
  String label;
  Node left, right;

  Node(String label) {
    this.label = label;
  }
}

class Tree {
  Node root;

  void draw(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);
    fill(0);
    textAlign(CENTER);
    text(n.label, 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);
      y -= 20;
      int my = (py + y) / 2;
      stroke(0, 0, 255);
      noFill();
      bezier(px, py, px, my, x, my, x, y);
    }

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

```
void draw() {
  draw(this.root, 0, 0);
}

// 段と列から x 座標を計算する
int xpos(int dan, int retsu) {
  // この計算式を工夫してほしい
  return retsu * 100 + 100;
}

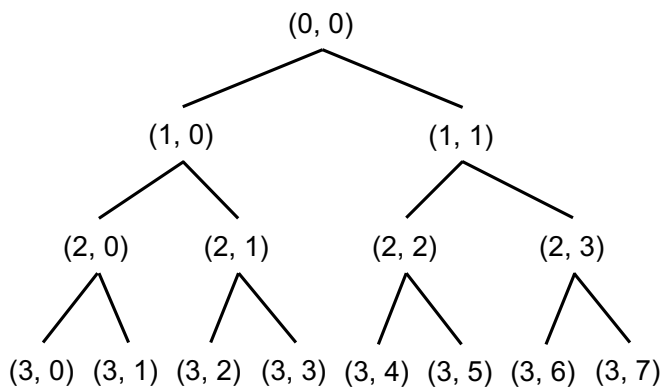
// 段と列から y 座標を計算する
int ypos(int dan, int retsu) {
  // この計算式を工夫してほしい
  return dan * 100 + 100;
}

Tree tree;
PFont font;

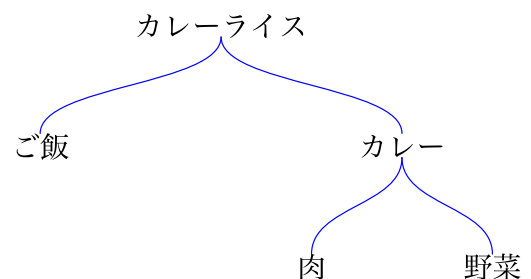
void setup() {
  size(600, 400);
  noLoop();
  font = createFont("Serif", 24);
  textFont(font);

  tree = new Tree();
  tree.root = new Node("カレーライス");
  tree.root.left = new Node("ご飯");
  tree.root.right = new Node("カレー");
  tree.root.right.left = new Node("肉");
  tree.root.right.right = new Node("野菜");
}

void draw() {
  background(255);
  tree.draw();
}
```



ノードの位置による (dan, retsu) の値



出力例