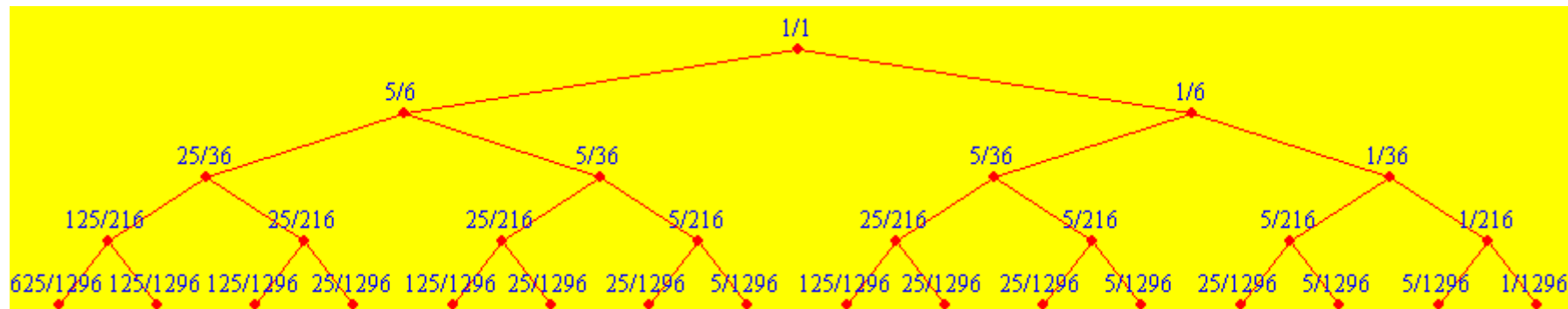


## Szenario

Mathematiklehrer Herbert Ohlweck ärgert sich immer, dass seine Grundkursschüler beim Thema Wahrscheinlichkeitsrechnung so schlecht mit Baumdiagrammen umgehen können. Deswegen will er mit seinem Informatikkurs eine Software entwickeln, die den Schülern des Mathe-GK das Erstellen und Auswerten von Baumdiagrammen erleichtert.

Unter anderem soll die Software helfen, die Wahrscheinlichkeiten beim mehrfachen Würfeln zu veranschaulichen und Berechnungen damit zu vereinfachen; unten wird das Baumdiagramm für viermal Würfeln dargestellt, wobei eine 6 als Erfolg (=nach rechts) und "nicht 6" als Fehlversuch (=nach links).



Baumdiagramm für viermal Würfeln; "6" gilt als Erfolg (=nach rechts); "nicht 6" als Fehlversuch (=nach links)

Die Information für jeden Knoten des Baumdiagrammes soll in einer Klasse `Bruch` gespeichert werden. In Objekten der Klasse `Bruch` werden die notwendigen Informationen zu einem Bruch gespeichert und die für dieses Szenario notwendigen Rechenoperationen implementiert. Die Dokumentation der Klasse `Bruch` findet sich im Anhang.

- Notieren Sie die ersten 8 Elemente eines Preorder-Durchlaufes durch den oben gegebenen Baum. (4 Punkte)  
Notieren Sie die ersten 8 Elemente eines Inorder-Durchlaufes durch den oben gegebenen Baum. (4 Punkte)
- Zeichnen Sie anhand der Informationen aus dem Anhang ein Klassendiagramm der Klasse `Bruch`. (5 Punkte)
  - Implementieren Sie die Klasse `Bruch`. (10 Punkte)  
Die Methode `dividiereDurch` brauchen Sie nicht zu implementieren.

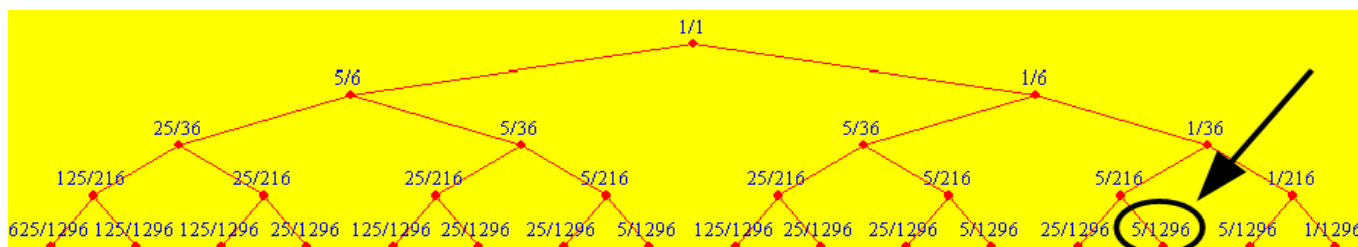
Herr Ohlweck stellt sich folgende Systemarchitektur vor: Die Klasse `Baumdiagramm` ist die Hauptklasse, die ausgeführt werden kann. `Baumdiagramm` enthält ein Objekt `baum` vom Typ `BinaryTree`, in dem die Informationen des Baumdiagrammes gespeichert werden; `baum` enthält deswegen lediglich Objekte vom Typ `Bruch`.

- c) Zeichnen Sie ein Implementationsdiagramm, das die Beziehungen zwischen den beteiligten Klassen abbildet. Attribute und Methoden müssen nicht aufgeführt werden. (6 Punkte)

In der Klasse `Baumdiagramm` soll es eine Methode `finde` geben, die das Auffinden von bestimmten Wahrscheinlichkeiten im Baumdiagramm ermöglicht. Für die Methode ist folgende Signatur vorgesehen:

```
public Bruch finde(String pLinie)
```

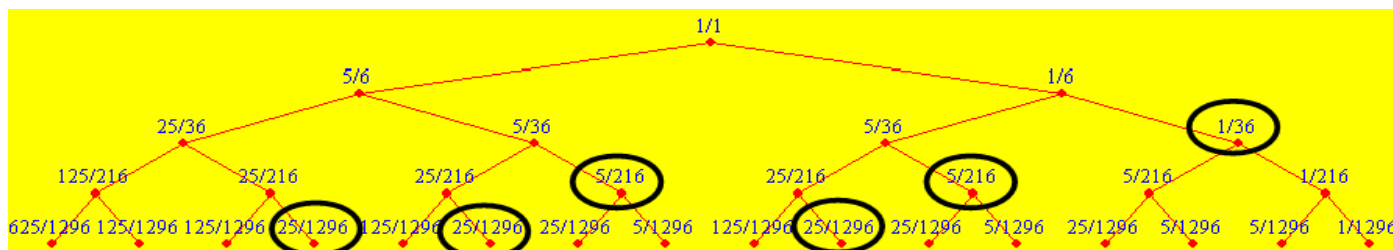
`pLinie` ist dabei eine Abfolge von "x" und "-"; dabei steht "x" für "Erfolg" und "-" für Misserfolg. Im Baumdiagramm auf S. 1 würde z.B. die Linie "xx-x" bedeuten: Zwei Sechsen, dann eine andere Zahl, dann wieder eine Sechs, und würde zu folgendem Bruch führen:



- d) Implementieren Sie die Methode `finde`. Fehler, die entstehen, wenn die Linie über das Diagramm hinauschießt, müssen nicht berücksichtigt werden. Berücksichtigen Sie die Dokumentation der Klasse `String` im Anhang. (8 Punkte)

Herr Ohlweck beauftragt seine Schüler zu berechnen, mit welcher Wahrscheinlichkeit man mindestens zwei Sechsen wirft.

Schüler Brin (1 Informatik, 4 Mathe) stellt fest, dass er dafür eine Liste der Knoten braucht, bei denen klar ist, dass zwei Sechsen geworfen wurden, d.h. dass man zweimal Erfolg hatte:



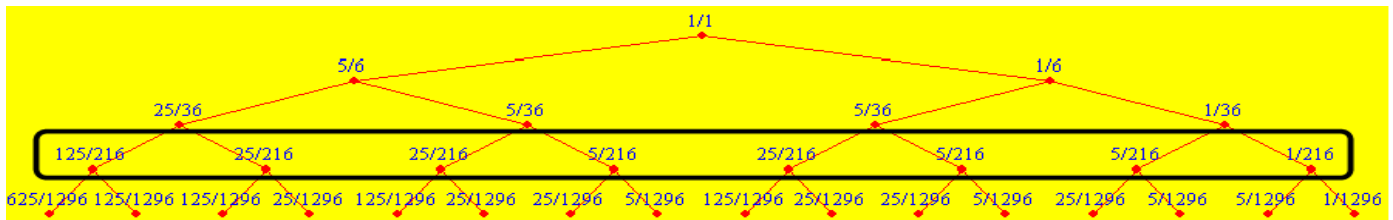
Um die Methode, die er entwickeln will, allgemein nutzen zu können, plant Brin die folgende Methode:

```
public List<Bruch> zweiErfolg()
```

Die Methode gibt eine Liste der Brüche zurück, die genau zwei "Erfolg" in ihrem Pfad haben. Sobald ein Bruch in die Liste aufgenommen wird, wird darunter nicht mehr weitergesucht.

- e) Implementieren Sie für die Klasse `Baumdiagramm` diese Methode und ggf. notwendige Hilfsmethoden. (15 Punkte)

Für gewisse Anwendungsszenarien hätte Herr Ohlweck gerne eine Liste aller Brüche aus einer bestimmten "Etage". Dabei wird die Wurzel des Baums als Etage 1 gewertet.



#### Brüche der Etage 4

Dafür soll eine Methode

```
public List<Bruch> etagenListe(int pEtage)
```

entwickelt werden.

Herr Ohlweck sieht dafür folgende Strategie vor:

- zwei Hilfsmethoden  
`private BinaryTree<Bruch> findeStartKnoten(int pEtage)` und  
`private BinaryTree<Bruch> findeEndKnoten(int pEtage)`  
 Der Startknoten ist der Knoten ganz links (im Bild der Knoten mit Inhalt 125/216) und der Endknoten ist der Knoten ganz rechts (im Bild der Knoten mit Inhalt 1/216).
- In der Methode `etagenListe` den baum mit Levelorder durchlaufen:
  - Sobald der Startknoten erreicht wird, in die Liste `ergebnis` einfügen.
  - Sobald der Endknoten erreicht wird, mit dem Einfügen aufhören und `ergebnis` zurückgeben.

f)

1. Implementieren Sie die Methoden `findeStartKnoten` und `etagenListe` gemäß der oben vorgeschlagenen Strategie. (19 Punkte).  
 (Auf die Implementierung von `findeEndKnoten` können Sie verzichten.)
2. Erläutern Sie eine andere Strategie für die Realisierung der Methode `etagenListe`. Für diese andere Strategie dürfen Sie Levelorder verwenden, müssen aber die relevanten Knoten auf eine andere Weise bestimmen. (6 Punkte)  
 Sie müssen Ihre Strategie nicht implementieren, sie sollte aber vom Programmieraufwand her beherrschbar sein.

Die Methode `neuerBaum` (siehe nächste Seite) wurde von Herrn Ohlweck entwickelt, um in der Klasse `Baumdiagramm` den vorhandenen Baum durch einen anderen ersetzen zu können.

Die Methode wird jetzt wie folgt aufgerufen:

```
neuerBaum(3, 2, 5);
```

g) Analysieren Sie den Methodenablauf, indem Sie notieren, welche Werte bzw. Inhalte die folgenden Variablen im Ablauf der Methode annehmen:

- `b1`, `b2`, `baum` und `listeA` in Zeile 11 ( `//*** x1 ***`) (4 Punkte)
- `i`, `listeA`, `listeB` in Zeile 26 ( `//*** x2 ***`)

Stellen Sie die Veränderung der Werte bzw. Inhalte tabellarisch dar. (12 Punkte)

Erläutern Sie in 1-2 Sätzen den Zweck von Zeile 27. (3 Punkte)

Stellen Sie das Aussehen des Attributes `baum` nach Ablauf der Methode graphisch dar. (4 P.)

```
1  public void neuerBaum(int pZahl, int pZ1, int pZ2){
    Bruch b1 = new Bruch(pZ1, pZ2);
    Bruch b2 = new Bruch(pZ2 - pZ1, pZ2);

5    Bruch eins = new Bruch(1,1);
    // Hier wird auf das Attribut baum zugegriffen!
    baum = new BinaryTree<Bruch>(eins);

    List<BinaryTree<Bruch>> listeA =
                                   new List<BinaryTree<Bruch>>();
10    listeA.append(baum);
    // *** x1 ***
    for(int i=0; i<pZahl; i++){
        List<BinaryTree<Bruch>> listeB =
                                   new List<BinaryTree<Bruch>>();
        for(listeA.toFirst(); listeA.hasAccess(); listeA.next()){
15            BinaryTree derBaum = listeA.getContent();
            Bruch derBruch = derBaum.getContent();
            Bruch neuLinks = derBruch.multipliziereMit(b2);
            BinaryTree<Bruch> neuBaumLinks =
                                   new BinaryTree<Bruch>(neuLinks);
            derBaum.setLeftTree(neuBaumLinks);
20            listeB.append(neuBaumLinks);
            Bruch neuRechts = derBruch.multipliziereMit(b1);
            BinaryTree<Bruch> neuBaumRechts =
                                   new BinaryTree<Bruch>(neuRechts);
            derBaum.setRightTree(neuBaumRechts);
            listeB.append(neuBaumRechts);
25        }
        // *** x2 ***
        listeA = listeB;
    }
}
```

## Dokumentation der Klasse Bruch

In Objekten der Klasse `Bruch` werden der Wert für den Zähler und den Nenner des Bruchs in geeigneter Weise gespeichert. Die Werte für Zähler und Nenner dürfen dabei beliebige ganze Zahlen sein, d.h. es können auch beide negativ sein oder der Nenner darf auch 0 sein (=da soll gefälligst der Anwender drauf achten.)

Konstruktor	<b><code>Bruch(int pZaehler, int pNenner)</code></b> erzeugt einen Bruch.
Anfrage	<b><code>int gibZaehler()</code></b> gibt den Zähler des Bruchs zurück.
Anfrage	<b><code>int gibNenner()</code></b> gibt den Nenner des Bruchs zurück.
Anfrage	<b><code>Bruch multipliziereMit(int pZahl)</code></b> multipliziert den Bruch mit einer ganzen Zahl <code>pZahl</code> und gibt das Ergebnis als Objekt vom Typ <code>Bruch</code> zurück; dabei wird nicht gekürzt. <i>Der Bruch selber bleibt unverändert.</i>
Anfrage	<b><code>Bruch multipliziereMit(Bruch pBruch)</code></b> multipliziert den Bruch mit dem Bruch <code>pBruch</code> und gibt das Ergebnis als Objekt vom Typ <code>Bruch</code> zurück; dabei wird nicht gekürzt. <i>Der Bruch selber bleibt unverändert.</i>
Anfrage	<b><code>Bruch dividiereDurch(int pZahl)</code></b> multipliziert den Bruch mit einer ganzen Zahl <code>pZahl</code> und gibt das Ergebnis als Objekt vom Typ <code>Bruch</code> zurück; dabei wird nicht gekürzt. <i>Der Bruch selber bleibt unverändert.</i>
Anfrage	<b><code>String toString()</code></b> gibt eine geeignete Textdarstellung des Bruchs zurück.

### Beispiele für die Multiplikation:

$$a = \frac{2}{5} \quad z = 3 \quad a \cdot z = \frac{2 \cdot 3}{5} = \frac{6}{5}$$

$$a = \frac{3}{4} \quad b = \frac{5}{7} \quad a \cdot b = \frac{3 \cdot 5}{4 \cdot 7} = \frac{15}{28}$$

## Die Klasse `String`

Mit Hilfe der Klasse `String` werden Zeichenketten repräsentiert. Zur Inspektion einer solchen Zeichenkette stehen verschiedene Methoden zur Verfügung.

### Dokumentation der Klasse `String` (Auszug)

- Anfrage** `int length()`  
liefert die Länge der Zeichenkette.
- Anfrage** `int indexOf(String pStr)`  
liefert die erste Position, an der `pStr` in dieser Zeichenkette vorkommt. Steht `pStr` direkt am Anfang dieser Zeichenkette, wird 0 zurückgeliefert. Ist `pStr` gar nicht enthalten, wird -1 geliefert.
- Anfrage** `String substring(int pIndexAnfang)`  
liefert einen neuen `String`, der nur die Zeichen ab der Position `pIndexAnfang` bis zum Ende dieser Zeichenkette enthält.
- Anfrage** `String substring (int pIndexAnfang, int pIndexEnde)`  
liefert einen neuen `String`, der nur die Zeichen ab der Position `pIndexAnfang` bis zur Position `pIndexEnde` enthält. `pIndexEnde` bezeichnet hierbei den Index hinter dem letzten Zeichen, das kopiert werden soll.
- Anfrage** `char charAt(int pIndex)`  
liefert das Zeichen, das an der Position `pIndex` steht. Das erste Zeichen wird mit dem Wert 0 ausgelesen.
- Anfrage** `boolean equals(Object pObject)`  
liefert genau dann `true`, wenn `pObject` ein Exemplar der Klasse `String` ist, das denselben Wert besitzt wie dieses Objekt, sonst `false`.
- Anfrage** `int compareTo(String pString)`  
liefert einen Wert, der kleiner als 0 ist, wenn die Zeichenkette lexikalisch kleiner ist als `pString`, 0, wenn sie gleich `pString` ist, und einen Wert größer als 0, wenn die Zeichenkette lexikalisch größer ist als `pString`.