252-0027

Einführung in die Programmierung

How to Graphenaufgaben

Henrik Pätzold

hpaetzold@student.ethz.ch

Work-in-Progress

25. Dezember 2024

Inhaltsverzeichnis

1 Einleitung

Wichtiger Hinweis

Dieses Dokument ist weder überprüft noch empfohlen durch die Dozenten oder das Teaching Team des Kurses **Einführung in die Programmierung**.
Ich übernehme keinerlei Garantie für die Korrektheit oder Vollständigkeit der Inhalte. Verwenden Sie dieses Dokument auf eigene Verantwortung.

Verbesserungsvorschläge, Fehler und/oder Fragen gerne per Mail an `hpaetzold@ethz.ch`.

Die Prüfungen der letzten Jahre hatten meist dasselbe Schema: Eine Aufgabe, in der ein Array augmentiert oder eine bestimmte Bedingung überprüft werden musste. Eine Aufgabe, in der ein Problem mithilfe verschiedener Klassen gelöst werden musste. Und eine Aufgabe, bei der auf einem Graphen eine Bedingung überprüft oder eine spezifische Information, wie der längste Pfad, zurückgegeben werden musste.
Während die ersten beiden Aufgabentypen gut erlernbar sind, wirken die Graphenaufgaben zu Beginn oft ein wenig willkürlich und nicht wirklich einheitlich lösbar. Ein Ansatz, der sich für mich in der überwältigenden Mehrheit als der leichteste, schnellste und meist auch am wenigsten codeintensive herausgestellt hat, besteht darin, den Graphen – der in der Regel nur als Objektreferenz eines Knotens gegeben ist – mithilfe von Rekursion in eine Form zu bringen, die einfach iterativ traversierbar ist. Anschließend können die benötigten Bedingungen dort überprüft werden.
Im Folgenden werde ich diesen Ansatz für einige der Probleme veranschaulichen und einige der nützlichsten Funktionsskelette präsentieren. Dabei ist mir wichtig zu betonen, dass es wahrscheinlich wenig bringt, diese auswendig zu lernen. Es ist vielmehr entscheidend zu verstehen, warum sie zur Lösung des jeweiligen Problems beitragen, um dann eigene Strategien entwickeln zu können. Effizienz spielt in Eprog keine zentrale Rolle, solange der Code eine zumutbaren polynomielle Schranke einhält. Daher reicht es aus, beim Schreiben von Code den Fokus auf Kürze statt auf Laufzeiteffizienz zu legen.
Trotzdem sollte man sich im Hinterkopf behalten, dass es für manche Aufgaben auch leichtere Lösungen gibt, die einfacher zu implementieren und/oder debuggen sind. Diese Optionen zu prüfen ist immer lohnenswert. Sonst funktioniert der beschriebene Ansatz jedoch sehr gut und ist mit ausreichend Übung leicht reproduzierbar.

2 Kurzfassung

1. **Aufgabenstellung verstehen:** Zuerst sollte man klären, welche Bedingungen der Graph erfüllen soll und was genau überprüft werden muss. Dabei hilft es, sich schon früh zu überlegen, wie man den Graphen in eine übersichtliche Datenstruktur bringen könnte, damit die anschließenden Kontrollen einfacher fallen.

2. **Konvertierung in eine geeignete Form:** Auch wenn der Graph lediglich über eine Objektreferenz erreichbar ist, lässt sich die Struktur häufig rekursiv durchwandern und in ein `Array` oder eine `Matrix` überführen. Für manche Aufgaben bietet sich eine stufenweise Zuordnung an (z. B. für baumähnliche Grafen), für andere eine Adjazenzmatrix (etwa bei `n` Knoten ein $\texttt{n} \times \texttt{n}$-Array). Wichtig ist, dass das Resultat gut *iterierbar* ist. So kann man in späteren Schritten iterativ jede gewünschte Eigenschaft systematisch prüfen.

3. **Iterative Überprüfung aller Bedingungen:** In der erstellten `Array`- bzw. `Matrix`-Darstellung lässt sich dann jede Kante, jede Stufe oder jede Belegung iterativ inspizieren. Welche konkreten Tests man durchführt, hängt von den Vorgaben ab: Vielleicht muss jeder Knoten genau zwei Nachbarn haben, oder bestimmte Felder müssen unbedingt gefüllt sein. Ergeben sich Widersprüche (z. B. ein Eintrag fehlt, ist doppelt belegt oder ein Index passt nicht), kann man sofort „falsch" zurückgeben.

4. **Ergebnis interpretieren:** Sofern alle Prüfungen in den Schleifen erfolgreich waren, meldet man „true" zurück (oder was immer als korrekt gilt). Andernfalls bricht man beim ersten Fehler ab. Gerade bei Graphen, die nicht den Anforderungen genügen, zeigt sich das in der gewählten Darstellung unmissverständlich: Knoten liegen an falschen Stellen, Nachbarn fehlen, oder man kann in einer Adjazenzmatrix wichtige Einträge nicht vorfinden. All das lässt sich klar abfragen.

Warum ist dieser Ansatz so robust?
Solange man den Graphen konsequent in ein für die Aufgabe passendes Format bringt, muss man sich bei der eigentlichen Prüfung nicht mehr in verschachtelten Referenzen verlieren. Stattdessen ruft man eine (rekursive) „Konvertierungslogik" auf, welche die Knoten ausliest, in das `Array` oder die `Matrix` einträgt und so den Graphen linear zugreifbar macht. Im Anschluss laufen sämtliche Tests *iterativ* ab: Man durchläuft Zeilen oder Spalten, vergleicht Einträge und kann bei Verstößen gegen die Aufgabenbedingungen direkt abbrechen. Dieser Mix aus rekursiver Erschließung (für die Konvertierung) und anschließendem iterativen Prüfen (für die Bedingungen) hat sich in vielfältigen Prüfungssituationen bewährt und ist trotz der Zweiteilung meist kompakt genug, um in kurzer Zeit umgesetzt zu werden.

3 Pyramide (2024 u10)

Im Folgenden wird eine Klasse dargestellt, die einen Knoten einer Baumstruktur repräsentiert. Dieser Knoten trägzt einen Wert und kann theoretisch eine unbegrenzte Anzahl von Kinderknoten haben.

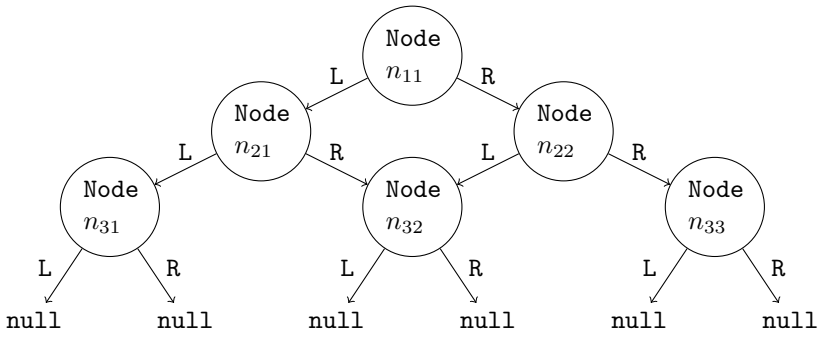
```
1  public class Node {
2          private Node left, right;
3          public Node(Node left, Node right) {
4                  this.setLeft(left);
5                  this.setRight(right);
6          }
7
8          public Node getLeft() {
9                  return left;
10         }
11
12         public void setLeft(Node left) {
13                 this.left = left;
14         }
15
16         public Node getRight() {
17                 return right;
18         }
19
20         public void setRight(Node right) {
21                 this.right = right;
22         }
23 }
```

Der Vorteil (oder auch Nachteil) an einer solchen Struktur, ist das wir den gesamten Graphen, als einzelnen Knoten übermittelt bekommen können.

3.1 Erste Idee

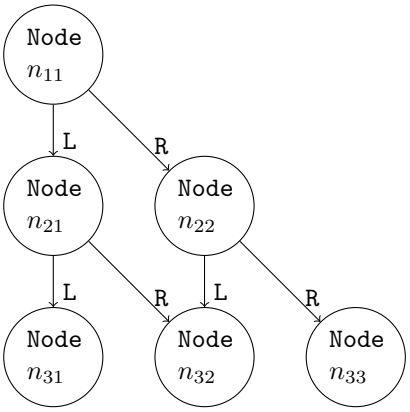
Wir erwarten also einen Knoten der Klasse Node. Das Ziel der Aufgabe ist jetzt zu überprüfen, ob der vollständige Graph, der in den Referenzen des Knotens und wiederrum seiner Kinderknoten gespeichert ist eine Pyramide darstellt. Informell haben wir dabei folgende Bedingungen, die ein azyklischer Graph mit $h \geq 1$ Stufen erfüllen muss (per Übungsblatt 10):

1. Für jede Stufe i (mit Indizierung beginnend bei 0) existieren genau $i + 1$ verschiedene Node-Objekte.
2. Für Stufe i ($1 \leq i < h$) gilt: Der linke Nachfolger eines Knotens n_{ij} ($1 \leq j \leq i$) ist $n_{(i+1)j}$, und der rechte Nachfolger ist $n_{(i+1)(j+1)}$. Für Knoten, die nicht am Rand liegen, gilt also, dass das rechte Kind eines Knotens gleich dem linken Kind seines rechten Nachbarn ist und umgekehrt.
3. In der letzten Stufe ist jeder Knoten ein Blatt. Das bedeutet, jeder Knoten hat ausschließlich `null`Referenzen als Kinder.

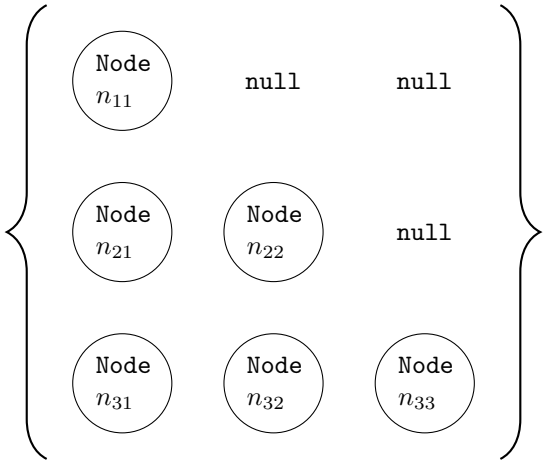


Dieser Graph ist eine korrekte Pyramide

Würden wir diesen Graphen übergeben bekommen, wäre uns lediglich n_{11} bekannt. Wenn der Graph jedoch in einem zweidimensionalen Array vorläge, könnten wir mit zwei einfachen `for`-Schleifen über die Struktur iterieren und das Problem erheblich vereinfachen. Daher bringen wir den Graph zunächst in eine Form, die iterativ verarbeitet werden kann.
Um die Idee besser zu veranschaulichen, stellen wir den Graphen in einer leicht abgeänderten Darstellung dar:



Damit wird deutlich, wie der Graph in einem Array repräsentiert werden könnte:



Da eine Pyramide mit h Stufen naheliegenderweise in einer $h \times h$-Matrix dargestellt werden kann, besteht der erste Schritt darin, diese maximale Höhe h zu ermitteln.

3.2 Tiefensuche

In Algorithmen und Datenstrukturen haben wir die Tiefensuche bereits kennengelernt. Diese iterativ zu implementieren, ist nicht schwierig, aber es sollte gesagt sein, dass Rekursion diesen Ansatz so gut macht, da rekursive Funktionen sehr kurz und bei ausreichend Übung leicht zu debuggen sind. Der folgende Dreizeiler ist ein rekursives `DFS` für einen Knoten mit zwei Kindern:

```
1 public static void DFS(TreeNode curr){
2     if(curr==null)return;
3     DFS(child.getLeft());
4     DFS(child.getRight());
5 }
```

Simples, nichts tuendes `DFS`

Die Funktion terminiert für einen azyklischen Graphen und läuft zunächst rekursiv durch die linken Kinder, bis sie einen Knoten erreicht, der nicht mehr existiert. An diesem Punkt kehrt sie zur vorherigen Ebene zurück und setzt die Verarbeitung mit den rechten Kindern fort.
Der Sinn der Abbruchbedingung `if (curr == null) return;` liegt darin, dass wir nicht bei jedem Knoten prüfen müssen, ob er Kinder hat. Stattdessen können wir bedenkenlos zu den Kindern navigieren, und die Funktion wird automatisch abbrechen, falls an der Stelle kein Knoten (also `null`) existiert. Dadurch wird zwar ein unnötiger Funktionsaufruf durchgeführt, jedoch bleibt der Code sauberer und übersichtlicher.
Das bedeutet, dass wir auf zusätzliche Überprüfungen wie `if (curr.getLeft() != null)` oder `if (curr.getRight() != null)` verzichten können. Dieser Ansatz ermöglicht es, die Logik der Funktion klar und fokussiert zu halten, da die Abbruchbedingung zentral und konsistent am Anfang der Methode definiert ist.
Code so sauber wie möglich zu halten, ist essenziell, da wir das `DFS` in dieser Form noch nicht direkt nutzen können. Häufig baut man in solchen Aufgaben schrittweise Funktionalitäten auf dem `DFS` auf. Daher ist es von Vorteil, den Code so sauber und übersichtlich wie möglich zu gestalten.
Die Abbruchbedingung `if (curr == null) return;` trägt dazu bei, dass wir auf zusätzliche `if`-Bedingungen verzichten können, da die Funktion automatisch abbricht, wenn kein Knoten (also `null`) vorhanden ist. Dies hält die Logik der Funktion fokussiert und vermeidet unnötige Prüfungen.
Eine `if`-Bedingung weniger bedeutet potenziell auch weniger Debugging in späteren Schritten, was den Entwicklungsprozess effizienter macht.

3.3 Finden der Dimensionen des Arrays

Wir passen das `DFS` jetzt an, um den Graphen zu traversieren und uns die Maße der Pyramide zu geben.

```
1 public static int getHeight(Node curr) {
2     if (curr == null)return 0;
3     return Math.max(DFS(curr.getLeft()) + 1, DFS(curr.getRight()) +
      ↪  1);
4 }
```

`DFS`, welches die maximale Höhe findet

Die Funktion berechnet rekursiv die maximale Tiefe der linken und rechten Unterbäume, addiert jeweils `1` für den aktuellen Knoten und gibt dann den größeren der beiden Werte zurück. Dies geschieht mithilfe von `Math.max`, wodurch sichergestellt wird, dass wir die längere der beiden möglichen Pfade im Baum berücksichtigen.
Ein einzelner Pfad im Baum wird so lange verfolgt, bis ein nicht existierender Knoten (also `null`) erreicht wird. An diesem Punkt gibt die Funktion `0` zurück, da keine weiteren Knoten vorhanden sind. Dieser Wert wird dann an den vorherigen, existierenden Knoten weitergegeben, der seine Höhe durch `+1` anpasst. Dieser Prozess setzt sich rekursiv fort, wobei jeder Knoten seine eigene Höhe berechnet, bis schließlich die Wurzel des Baumes erreicht ist. Da an jedem Knoten die maximale Tiefe des Teilbaums — also die Länge des längsten Pfades, der von diesem Knoten ausgeht — zurückgegeben wird, liefert die Funktion für n_{11} die Anzahl der Stufen h zurück.
Damit können wir jetzt ein $h \times h$-Array des Typs `Node` definieren.

```
1 public static boolean isPyramid(Node node) {
2     int h = DFS(node);
3     Node[][] nodes = new Node[h][h];
4 }
```

Zwischenschritt 1 für `isPyramid`

3.4 Befüllen des Arrays

Wie in Abbildung 3.1 gezeigt, erwarten wir von einer korrekten Pyramide in Array-Form, dass für jeden Knoten n_{ij} an der Position (i, j) gilt, dass sich sein linkes Kind bei $(i + 1, j)$ und sein rechtes Kind bei $(i + 1, j + 1)$ befindet.
Wir definieren eine rekursive Funktion, die folgende Parameter verwendet:

- `Node curr` – Der Knoten, der aktuell behandelt wird.
- `Node[][] nodes` – Die Referenz auf den endgültigen Speicherort der Knoten.
- `int i` – Die Stufe, in der `curr` gespeichert werden soll.

- `int j` – Die Spalte, in der `curr` gespeichert werden soll.

Diese Parameter enthalten alle notwendigen Informationen über den aktuellen Zustand, den sich die Funktion merken muss. Jeder der Parameter ist essenziell, um die gewünschte Funktionalität sicherzustellen. Besonders praktisch ist, dass Arrays in Java Objektreferenzen sind: Wir müssen lediglich die Referenz auf das `nodes`-Array weitergeben. Dadurch hat jeder Funktionsaufruf automatisch Zugriff auf die aktuellste Version des Arrays. Eine Rückgabe des Arrays ist somit nicht erforderlich, solange es vor dem Funktionsaufruf außerhalb der Funktion definiert wurde.

```
1  public static void fillArr(Node curr, Node[][] nodes, int i, int j) {
2          if (curr == null)
3             return;
4          nodes[i][j] = curr;
5          fillArr(curr.getLeft();, nodes, i + 1, j);
6          fillArr(curr.getRight();, nodes, i + 1, j + 1);
7  }
```

Fertiger Code zum Befüllen des Arrays

Diese Funktion setzt den aktuellen Knoten `curr` an die Position (i, j) im Array `nodes`. Nachdem dies geschehen ist, ruft sie sich rekursiv auf die Kinder von `curr` auf, wobei sie für das linke Kind die Parameter $(i + 1, j)$ und für das rechte Kind $(i + 1, j + 1)$ übergibt.
Wenn `curr null` ist, können wir den Funktionsaufruf sofort abbrechen, da wir wissen, dass Nullreferenzen keine Attribute und damit auch keine Kinderknoten haben können. Dies dient als Abbruchbedingung für die Rekursion und verhindert eine `NullPointerException` in Zeile 5.
Durch diese rekursive Vorgehensweise wird sichergestellt, dass jeder Knoten an der richtigen Position im Array `nodes` gespeichert wird.

```
1  public static boolean isPyramid(Node node) {
2      int h = DFS(node);
3      Node[][] nodes = new Node[h][h];
4      fillArr(node, nodes, 0, 0);
5  }
```

Zwischenschritt 2 für `isPyramid`

3.5 Iteratives Überprüfen der Bedingungen

Wir haben den Graphen nun in der korrekten Form in `nodes` gespeichert und können beginnen, ihn iterativ zu überprüfen. Die Bedingungen 3.1, die der Graph gemäß Aufgabenstellung überprüfen muss, können mit der neuen Struktur folgendermaßen neuinterpretiert werden:

1. `nodes`$[i]$ enthält genau i+1 Elemente für $0 \leq i < h$ (Indexierung beginnend bei 0).

2. `nodes`$[i][j]$`.getLeft()` == `nodes`$[i][j]$`.getRight()` für $0 \leq i, j < h - 1$.
3. `nodes`$[h-1][j]$`.getLeft() == null` und `nodes`$[h-1][j]$`.getRight() == null` für $0 \leq j < h$.

Wir iterieren über das `nodes`-Array in einer doppelten `for`-Schleife. Die Idee besteht darin, in jeder Iteration zu überprüfen, ob eine Bedingung verletzt wird. Ist dies der Fall, wird `false` zurückgegeben. Sollte jede Iteration vollständig durchlaufen werden, so gelten offensichtlich alle drei Bedingungen für jeden Knoten, was den Graphen per Definition zu einer Pyramide macht. Es könnte sinnvoll sein zu erwähnen, dass Bedingung 3 durch die Konstruktion des `getHeight` bereits garantiert ist und daher tendenziell nicht überprüft werden muss. Es schadet jedoch nicht, sie trotzdem in den Code zu integrieren, da man so an einer Prüfung ohne viel kognitiven Aufwand sorglos die Bedingungen runterschreiben kann, während die meiste Arbeit in das sinnvolle Konvertieren der Graphenstruktur fließt.

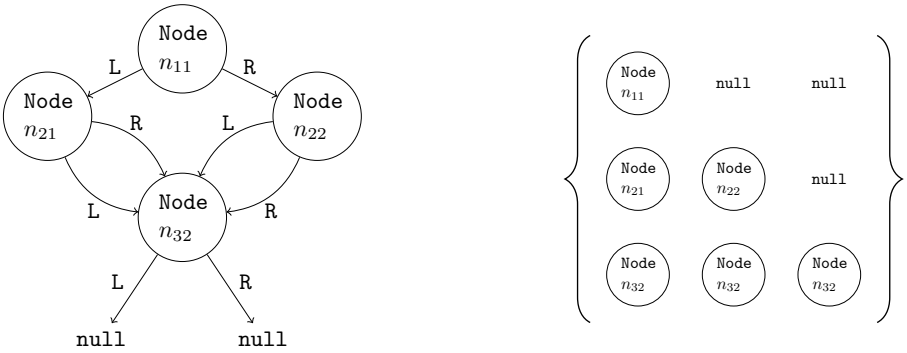
3.5.1 Bedingung 1

```
1  for (int i = 0; i < h; i++) {
2      HashSet<Node> traversed = new HashSet<Node>();
3      for (int j = 0; j < h; j++) {
4          Node curr = nodes[i][j];
5          if (curr == null)
6              continue;
7          traversed.add(curr);
8          if ((j > i && curr != null) || (j == i && traversed.size() <
           ↪  i + 1))
9              return false;
10     }
11 }
```

Überprüfung Bedingung 1

Neben der Definition von Hilfsvariablen wie `curr` überprüfen wir mit dem `HashSet<Node> traversed` für jede Zeile aufs Neue, ob wir genau $i + 1$ eindeutige `Node`-Referenzen haben. Jedes Mal, wenn wir eine Stelle j im Array behandeln, in der per Definition ein Knoten liegen sollte, also für $0 \leq j \leq i$, und `nodes[i][j] != null` gilt, merken wir uns die Objektreferenz über das Set. Haben wir alle i Stellen überprüft und befinden sich im Set weniger als $i + 1$ Objekte ($u < i + 1$), so wissen wir: Mindestens ein Knoten kam entweder doppelt vor oder mindestens eine Position des Graphen ist nicht belegt. Dieser Fall wäre eine klare Abbruchbedingung.
Die Bedingung `(j > i && curr != null)` ist ebenso durch die Aufgabenstellung gegeben und könnte prinzipiell weggelassen werden. Sollte ein Test jedoch fehlschlagen, weil sich in einer Zeile außerhalb der Reichweite der Pyramide ein Knoten befindet, wissen wir, dass die Methode `fillArr` falsch arbeitet. Dies ist ein weiterer Grund, warum es sinnvoll sein kann, die Bedingungen einfach vom Blatt zu übernehmen, weil sie sich mit anderen vielleicht decken, anstatt sie voreilig auszuschließen.

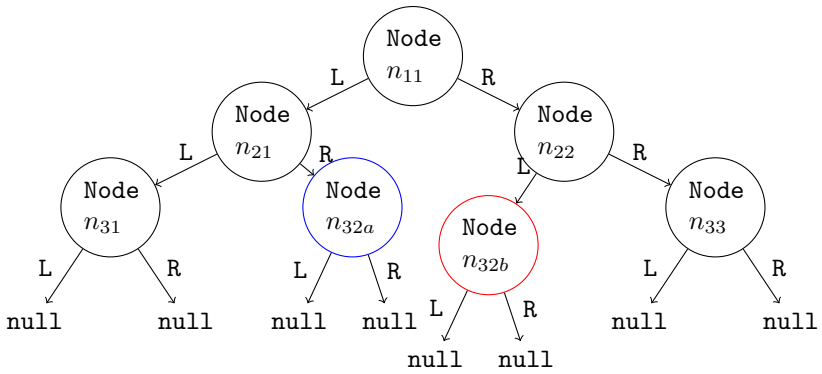
Der folgende Graph und das damit korrespondierende Array nach dem Durchlaufen von `getHeight` und `fillArr` würden durch Bedingung 1 bereits ausgeschlossen werden:



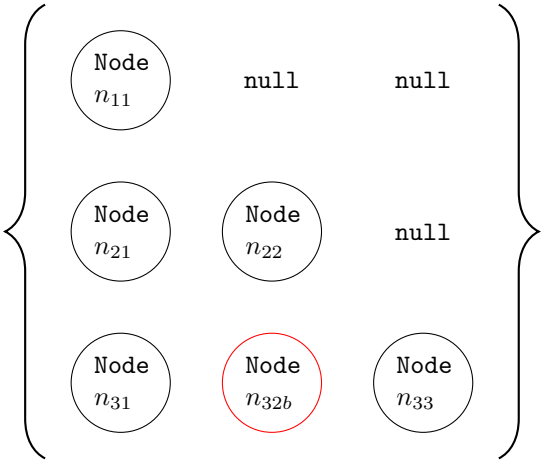
Am Ende der letzten Zeile würde `traversed` beim Überprüfen nur ein Element enthalten. $\lightning$

3.5.2 Bedingung 2

Wir wissen, dass wir durch die korrekt geleistete Vorarbeit bei einer korrekten Pyramide nur noch Kinderknoten bis zur vorletzten Zeile $h-2$ erwarten. Daher überprüfen wir jetzt die spezifische Bedingung für diese Zeile: Jeder Knoten `nodes[`$h-2$`][`j`]` in der vorletzten Zeile muss ein linkes Kind `nodes[`$h-1$`][`j`]` und ein rechtes Kind `nodes[`$h-1$`][`$j+1$`]` besitzen. Zusätzlich gilt, dass das rechte Kind eines Knotens `nodes[`$h-2$`][`j`]` gleich dem linken Kind seines rechten Nachbarn `nodes[`$h-2$`][`$j+1$`]` sein muss.
Um das zu überprüfen, schauen wir uns folgenden Graphen und das daraus resultierende Array an:



Baum, der keine Pyramide darstellt.



Bestmögliche Interpretation als Pyramide durch `getHeight` und `fillArr`

Hier kommt uns gelegen, dass wir beim konvertieren davon ausgehen, dass wir immer eine korrekte Pyramide übergeben bekommen: Denn der Knoten n_{32_a} wird erst von `fillArr` an die korrekte Position gesetzt, aber beim behandeln von n_{22} überschreibt `fillArr` n_{32_a} dann mit der Objektreferenz von n_{32_b}.
Das bedeutet, dass wir bei der iterativen Überprüfung keine zusätzlichen Schritte benötigen, um sicherzustellen, dass die Kindknoten-Konsistenz stimmt. Stattdessen können wir direkt überprüfen, ob die Kinder eines Knotens an der richtigen Stelle im Array liegen und im Fall, dass dies nicht der Fall ist, einfach `false` zurückgeben.

```
1  for (int i = 0; i < h; i++) {
2      HashSet<Node> traversed = new HashSet<Node>();
3      for (int j = 0; j < h; j++) {
4          Node curr = nodes[i][j];
5          if (curr == null)
6              continue;
7          traversed.add(curr);
8          if ((j > i && curr != null) || (j == i && traversed.size() <
           ↪  i + 1))
9              return false;            // Bedingung 1
10         if (i < nodes.length - 1) { // Bedingung 2
11             Node lC = nodes[i + 1][j];
12             Node rC = nodes[i + 1][j + 1];
13             if (lC == null || lC != curr.getLeft())
14                 return false;
15             if (rC == null || rC != curr.getRight())
16                 return false;
17         }
18     }
19 }
```

Überprüfung Bedingung 1 & 2

3.5.3 Bedingung 3

Für Bedingung 3 müssen wir lediglich überprüfen, dass die Knoten in der letzten Stufe keine Kinder haben. Per Konstruktion von `getHeight` ist diese Bedingung trivialerweise erfüllt. An einer Prüfung mache ich mir darüber jedoch keine Gedanken, welche Bedingungen sich unter Umständen gegenseitig decken könnten und welche man sich dann sparen kann. Daher überprüfe ich sie trotzdem. Sie machen das Programm nicht falscher, sondern lediglich ein wenig länger. Ich erspare mir dadurch aber unnötige Überlegungen dazu, welche Aspekte ich beachten müsste, wenn ich diese Überprüfung weglassen würde.

```
for (int i = 0; i < h; i++) {
    HashSet<Node> traversed = new HashSet<Node>();
    for (int j = 0; j < h; j++) {
        Node curr = nodes[i][j];
        if (curr == null)
            continue;
        traversed.add(curr);
        if ((j > i && curr != null) || (j == i && traversed.size() <
        ↪  i + 1))
            return false;           // Bedingung 1
        if (i < nodes.length - 1) { // Bedingung 2
            Node lC = nodes[i + 1][j];
            Node rC = nodes[i + 1][j + 1];
            if (lC == null || lC != curr.getLeft())
                return false;
            if (rC == null || rC != curr.getRight())
                return false;
        } else {                    // Bedingung 3
            if (curr.getLeft() != null)
                return false;
            if (curr.getRight() != null)
                return false;
        }
    }
}
```

Überprüfung Bedingung 1,2,3

3.6 Finaler Code

```java
public static boolean isPyramid(Node node) {
    int h = DFS(node);
    Node[][] nodes = new Node[h][h];
    fillArr(node, nodes, 0, 0);

    for (int i = 0; i < h; i++) {
        HashSet<Node> traversed = new HashSet<Node>();
        for (int j = 0; j < h; j++) {
            Node curr = nodes[i][j];
            if (curr == null)
                continue;
            traversed.add(curr);
            if ((j > i && curr != null) || (j == i &&
            ↪  traversed.size() < i + 1))
                return false;           // Bedingung 1
            if (i < nodes.length - 1) { // Bedingung 2
                Node lC = nodes[i + 1][j];
                Node rC = nodes[i + 1][j + 1];
                if (lC == null || lC != curr.getLeft())
                    return false;
                if (rC == null || rC != curr.getRight())
                    return false;
            } else {                    // Bedingung 3
                if (curr.getLeft() != null)
                    return false;
                if (curr.getRight() != null)
                    return false;
            }
        }
    }
}

public static int getHeight(Node curr) {
    if (curr == null)return 0;
    return Math.max(DFS(curr.getLeft()) + 1, DFS(curr.getRight()) +
    ↪  1);
}

public static void fillArr(Node curr, Node[][] nodes, int i, int j) {
        if (curr == null)
           return;
        nodes[i][j] = curr;
        fillArr(curr.getLeft();, nodes, i + 1, j);
        fillArr(curr.getRight();, nodes, i + 1, j + 1);
}
```

Finaler Code

4 Rekursionsideen

4.1 Einleitung

Die in diesem Abschnitt vorgestellten Codeschnipsel verdeutlichen, wie man ausgehend von einer einfachen Tiefensuche aufbauend verschiedene Probleme lösen kann. Alle Beispiele sollen lediglich als Inspiration dienen.

4.2 TreeNode-Klasse

Die Klasse `TreeNode` repräsentiert einen Knoten in einem Baum. Sie besitzt einen Wert (`value`) und verwaltet eine beliebige Anzahl an Kindknoten:

```java
class TreeNode {
    private TreeNode[] children;
    private int value;
    public TreeNode(int value,TreeNode[] children){
        this.children = children;
        this.value = value;
    }

    public TreeNode getChild(int index){
        if(index >= this.children.length)throw new
        ↪  IndexOutOfBoundsException();
        return children[index];
    }

    public int value(){
        return this.value;
    }

    public TreeNode[] children(){
        return this.children;
    }
}
```

`TreeNode`

In einem azyklischen Graphen (bzw. Baum) ist jeder `TreeNode` eindeutig über seine Position in der Knotenhierarchie erreichbar. Die Implementierung setzt voraus, dass keine Kreise existieren, sodass rekursive Aufrufe problemlos terminiert werden können.

DFS

```java
public static void DFS(TreeNode curr){
        if(curr==null)return;
        for(TreeNode child : curr.children())DFS(child);
}
```

Simples, nichts tuendes `DFS`

Die Funktion `DFS(curr)` durchläuft den Baum ausgehend vom Knoten `curr`. Sobald die Methode `null` erreicht, wird abgebrochen, da hier kein Knoten existiert. Ansonsten werden die Kinder von `curr` nacheinander rekursiv besucht. Der Parameter `curr` ist erforderlich, um festzulegen, von welchem Knoten aus die Tiefensuche weitergeführt wird. Da keine Zyklen vorliegen, ist kein `visited`-Flag nötig.

maxDepth

```java
public static int maxDepth(TreeNode curr) {
    if (curr == null) return 0;
    int maxChildDepth = 0;
    for (TreeNode child : curr.children()) {
        maxChildDepth = Math.max(maxChildDepth, maxDepth(child));
    }
    return 1 + maxChildDepth;
}
```

Die Methode `maxDepth(curr)` berechnet die maximale Tiefe des Teilbaumes mit Wurzel `curr`. Ist `curr` `null`, wird `0` zurückgegeben. Andernfalls bestimmt die Rekursion die Tiefe jedes Kindknotens und wählt den größten Wert aus, zu dem `1` addiert wird, um den aktuellen Knoten selbst zu berücksichtigen. Der Parameter `curr` gibt an, welche Knotenstruktur gerade überprüft wird; ein separater `depth`-Zähler wird nicht benötigt, da die Rekursion diesen Effekt übernimmt.

longestPath

```java
public static ArrayList<TreeNode> longestPath(TreeNode curr,
↪  ArrayList<TreeNode> path) {
    ArrayList<TreeNode> maxPath = null;
    if (curr == null) return path;
    path.add(curr);
    for (TreeNode child : curr.children()) {
        ArrayList<TreeNode> temp = longestPath(child, new
        ↪  ArrayList<TreeNode>(path));
        if (maxPath == null || temp.size() > maxPath.size()) {
            maxPath = temp;
        }
    }
    return maxPath;
}
```

Mit `longestPath(curr, path)` wird der längste Pfad ab dem Knoten `curr` ermittelt. Neben `curr` benötigt die Methode den Parameter `path`, der bereits

durchlaufene Knoten speichert. Gleich beim Start wird `curr` an `path` angehängt. Für jeden Kindknoten erstellt die Methode eine *Kopie* dieser Liste, damit parallele Rekursionszweige sich nicht überschreiben. Nach Abschluss aller Rekursionen wird der Pfad mit der größten Länge zurückgegeben. Ohne das Kopieren würden alle Kinderaufrufe ein und dasselbe `path`-Objekt verändern, was zwangsläufig zu fehlerhaften Pfaden führen würde. So ist klar von *Teilpfad* zu *Teilpfad* getrennt, welcher Knoten bereits besucht wurde.

reachableNodes

```
1  public static TreeSet<TreeNode> reachableNodes(TreeNode curr,
   ↪  TreeSet<TreeNode> nodes) {
2      if (curr == null) return nodes;
3      nodes.add(curr);
4      for (TreeNode child : curr.children()) {
5          reachableNodes(child, nodes);
6      }
7      return nodes;
8  }
```

Die Funktion `reachableNodes(curr, nodes)` sammelt alle von `curr` aus erreichbaren Knoten und speichert diese in `nodes`, einem `TreeSet`. Der Parameter `curr` legt den Startknoten fest, während `nodes` eine sortierte, duplikatfreie Sammlung ist. Durch einen rekursiven Aufruf für jedes Kind wird sichergestellt, dass alle „Nachfahren“ von `curr` im Set landen. Zyklische Besuche sind nicht zu befürchten, da es sich um einen Baum handelt.

allPaths

```
1  public static void allPaths (TreeNode curr, TreeNode dest,
2                               ArrayList<ArrayList<TreeNode>> allPaths,
3                               ArrayList<TreeNode> currPath) {
4      if (curr != null) {
5          currPath.add(curr);
6          if (curr == dest) {
7              allPaths.add(currPath);
8          } else {
9              for (TreeNode child : curr.children()) {
10                 allPaths(child, dest, allPaths, new
                   ↪  ArrayList<TreeNode>(currPath));
11             }
12         }
13     }
14 }
```

Die Methode `allPaths(curr, dest, allPaths, currPath)` ermittelt sämtliche Pfade vom Startknoten `curr` zum Zielknoten `dest`. Hierbei sind gleich vier Parameter vonnöten:

- `curr` – Der aktuell betrachtete Knoten.
- `dest` – Das gesuchte Ziel.
- `allPaths` – Die Liste, in der alle vollständigen Pfade abgelegt werden.
- `currPath` – Der Pfad, der bereits bis `curr` aufgebaut wurde.

Zunächst wird `curr` an `currPath` angehängt. Ist `curr` bereits `dest`, wird dieser Pfad nach `allPaths` kopiert. Andernfalls wird für jedes Kind ein neuer Rekursionsschritt ausgeführt, wobei eine *neue Kopie* von `currPath` angelegt wird. So können unterschiedliche Suchpfade ungestört nebeneinander existieren.

4.3 Rekursion — Konzepte und Intuition

Rekursive Verfahren wie die vorgestellten `DFS`, `maxDepth` oder `allPaths` lassen sich besonders gut verstehen, wenn man sich den Ablauf bildlich vorstellt. Grundsätzlich folgt jede rekursive Methode demselben Muster:

1. **Abbruchbedingung überprüfen:**
   Sobald die Methode bei einem `null`-Knoten oder einem sonstigen „Endpunkt“ (z. B. dem Zielknoten bei `allPaths`) angekommen ist, wird die Rekursion nicht mehr fortgeführt. Dieses `if (curr == null) return;` ist ein Klassiker in allen DFS-ähnlichen Methoden, denn es macht den Code deutlich schlanker. Im Gegensatz zu einer Iterationsvariante mit mehreren `if`-Abfragen führt dieser eine *zentrale* Prüfung durch, hinter der jede weitere Logik automatisch entfällt, sobald es keinen gültigen Knoten gibt.

2. **Aktuellen Knoten *bearbeiten*:**
   Hier findet der *eigentliche* inhaltsbezogene Teil statt. Für verschiedene Probleme kann diese Bearbeitung sehr unterschiedlich aussehen:

   - In einer einfachen Tiefensuche (`DFS`) reicht es oft, den Wert des aktuellen Knotens auszugeben oder zu zählen. Beispielsweise:

     ```
     System.out.println(curr.value());
     ```

   - Will man alle Pfade abbilden (`allPaths`), hängt man den aktuellen Knoten `curr` an eine `ArrayList` namens `currPath` an. Beim Erreichen des Zielknotens `dest` kopiert man diesen `currPath` in eine Gesamtliste `allPaths`.
   - Bei `longestPath` nimmt man `curr` ebenfalls in eine Pfadliste auf, vergleicht später die gefundenen Pfade nach ihrer Länge und behält den längsten.
   - Für `maxDepth` dagegen ist das „Bearbeiten“ an dieser Stelle nicht mehr als das *Speichern* oder *Rückgeben* der Tiefeninformation. Häufig ist das nur ein `return 1 + ...` plus das Maximum der Kindtiefen.

   All diese Varianten nutzen dasselbe Grundmuster: Wir bearbeiten zunächst den aktuellen Knoten, *lösen dabei bereits einen Teil des Problems*, und steigen dann zu den Kindknoten „hinab“, um das verbleibende *Subproblem* rekursiv zu lösen. Sobald man verstanden hat, dass *hier* der Ort ist, an dem man *verändern* (z. B. etwas ausgeben, anhängen, summieren) möchte, kann man neue Ideen leicht einbauen. Möchte man z. B. die Summe der Knotenwerte berechnen, so könnte man hier `sum += curr.value()` verarbeiten, bevor man tiefer geht.

3. **Rekursiv zu den Kindern *gehen*:**
   In einem Baum weist jeder Knoten eine Liste von Kindknoten auf, und wir rufen dieselbe Methode für jedes Kind erneut auf. Das ist das eigentliche „nach unten“ Gehen in der Rekursion:

   ```
   for (TreeNode child : curr.children()) { ... }
   ```

   Jedes Kind wird nun so behandelt, als wäre es *der* Knoten. Sofern keine Zyklen existieren, kann man bedenkenlos alle Kinder abarbeiten. Sollte ein

Kind `null` sein, greift die Abbruchbedingung und beendet den rekursiven Aufruf.

4. **Zurückkehren („nach oben“):**
   Sobald alle Kinder behandelt sind (oder der Abbruchfall zutrifft), *kehrt* die Rekursion wieder zurück. In diesem Moment beendet sich der aktuelle Funktionsaufruf, und das Programm „springt“ zur aufrufenden Methode zurück. So *klettert* man Pfad für Pfad durch den Baum nach oben. Dieses Zurückkehren ist im Code selten explizit zu sehen, da es automatisch passiert, wenn das Ende der Methode erreicht ist. Dennoch ist das *der* entscheidende Mechanismus, der alle Teilresultate wieder „zusammenführt“.

Durch diesen Kreislauf (*Prüfen, Bearbeiten, Tiefergehen, Aufstieg*) bearbeitet man *alle* Knoten ganz natürlich in einer Baumstruktur. Taucht `null` auf, *stoppt* die Rekursion für diesen Zweig unmittelbar. Ist `curr` ein gültiger Knoten, *bearbeiten* wir ihn und steigen in die Tiefe („nach unten“) zu seinen Kindknoten. Sobald die Kinder ihre Aufgaben erledigt haben, kehren sie automatisch zum aufrufenden Knoten „nach oben“ zurück und hinterlassen dort ihre Resultate (z. B. `int`-Werte für die Tiefe oder vollständige Pfade als `ArrayList`).
Das Grundgerüst – Abbruchbedingung, Bearbeitung, Kinderaufrufe – kann man leicht auf verschiedene Probleme anpassen. Das *Warum* und *Wie* der Parameter sollte man beim Schreiben kommentieren, damit auf den ersten Blick klar wird, wofür `curr` oder `path` gebraucht wird. Auf diese Weise behält man stets die Übersicht, und das rekursive Vorgehen bleibt überschaubar und zuverlässig anpassbar.

Wahl und Bedeutung der Parameter:

- `(TreeNode curr)`: Stets der *aktuelle* Knoten. Er ist unumgänglich, weil man wissen muss, *wo* im Baum man sich gerade befindet. Ohne diesen Parameter wäre unklar, von welchem Knoten aus man *jetzt* agiert.

- `(ArrayList<TreeNode> path)`: Falls man einen *Pfad* mitführen möchte (etwa für `longestPath` oder `allPaths`), braucht man eine Datenstruktur, die die bereits durchlaufenen Knoten speichert. Diese Liste wird in jedem Rekursionsschritt erweitert.

- `(TreeSet<TreeNode> nodes)`: Wenn man alle erreichten Knoten in einer *Menge* speichern will (`reachableNodes`), übergibt man diese `TreeSet`-Struktur, die sich sukzessive füllt. Ein `TreeSet` ist bequem, weil es keine Duplikate zulässt und ggf. sortiert sein kann.

- **Kopieren von Listen vs. kein Kopieren**: Für `longestPath` oder `allPaths` muss `path` in jedem *Kindaufruf konsistent* bleiben. Würde man `path.add(curr)` ohne Kopieren aufrufen, teilen sich sämtliche Rekursionspfade dasselbe `path`-Objekt. Änderungen in einem Pfad würden dann *alle* Parallelpfade beeinflussen und zu einem fehlerhaften Endergebnis führen. Mit

  ```
  new ArrayList<>(path)
  ```

  erzeugt man bei jedem Kindaufruf eine neue Liste, die eigenständig ist. So kann jeder Pfad ungestört wachsen, ohne andere Pfade zu überschreiben.

Wann geben wir rekursiv etwas zurück und wann nicht?
Rekursive Methoden lassen sich grob in zwei Kategorien unterteilen:

1. **Methoden, die einen Wert (z. B. `int`, `ArrayList`, ...) zurückgeben:**

   Beispiele sind `maxDepth` oder `longestPath`. Beide müssen aus jedem Unteraufruf ein Ergebnis *zurück* an den aufrufenden Knoten liefern, damit man dort die Ergebnisse aller Kinder *kombinieren* kann:

   - Bei `maxDepth(curr)` summiert man „1“ für den aktuellen Knoten und vergleicht dann die (rekursiv) zurückgegebenen Tiefen der Kinder, um den maximalen Wert zu erhalten.
   - Bei `longestPath(curr, path)` oder `allPaths(curr, dest, allPaths, currPath)` entsteht aus jedem Unteraufruf ein (Teil-)Pfad, den man an den Aufrufer zurückgibt oder in einer übergeordneten Datenstruktur sammelt. Der aufrufende Knoten kann die zurückgegebenen Pfade miteinander vergleichen oder weiterverarbeiten.

   In solchen Fällen steht am Ende jeder Methode `return <Ergebnis>;` – etwa `return bestPath;` oder `return 1 + maxDepthChild;`. Auf diesem Wege werden die Informationen „nach oben“ in der Rekursion weitergereicht.

2. **Methoden ohne Rückgabewert (`void`):**

   Beispiele sind viele Standard-`DFS`-Varianten oder `reachableNodes`, die den aktuellen Knoten in eine globale oder als Parameter übergebene Datenstruktur (`TreeSet`, `List`, ...) eintragen. Hier *speichert* oder *verändert* man Informationen lediglich *seiteneffekthaft*, d. h. ohne etwas direkt `return` zu müssen:

   - `DFS(curr)` ruft sich rekursiv auf die Kinder auf und *macht* innerhalb der Funktion etwa ein `System.out.println(curr.value())`, hat aber keinen Wert, den es an den Aufrufer zurückreichen müsste.
   - `reachableNodes(curr, nodes)` fügt `curr` zu `nodes` hinzu und ruft sich rekursiv für alle Kinder auf. Da alle besuchten Knoten in `nodes` gesammelt werden, muss die Methode nichts „nach oben“ zurückgeben.

   Die *Rückkehr* erfolgt dennoch, sobald alle Kinder fertig verarbeitet sind (oder der `null`-Abbruchfall greift). Zwar wird nichts *zurückgegeben*, doch die Funktion *terminiert*, sobald sie ihre Aufgabe beendet hat.

Fazit: Ob man *etwas* zurückgeben sollte, richtet sich danach, ob das Ergebnis der Teilaufrufe *benötigt* wird. Muss jede Teilrekursion etwas *zurückliefern*, damit wir es „nach oben“ verarbeiten können (z. B. eine maximale Tiefe, einen Pfad, eine Anzahl)? Dann verwenden wir eine Rückgabe wie `int` oder `List<...>` und fassen die Teilergebnisse im aufrufenden Knoten zusammen.
Wenn man hingegen nur eine globale Struktur befüllt (*Seiteneffekt*), reicht eine `void`-Methode aus, die ihrerseits rekursiv weiter aufruft und weder Daten noch `boolean`-Werte `return`-en muss. In diesem Fall genügt es, dass alle Knotenbearbeitungen sich *einfach „ansammeln“* oder *ausgeben*, ohne etwas zu `return`-en.

Wichtig bleibt vor allem eine klar definierte Abbruchbedingung (`if (curr == null) return;`), damit ein Teilzweig nicht endlos weiterläuft.
Solche rekursiven Strategien ersparen oft komplizierte Schleifen oder zusätzliche Zwischenspeicher und sind gerade bei Graph- und Baumaufgaben eine elegante, leicht zu verstehende Lösung. Das Prinzip bleibt immer gleich: *Knoten bearbeiten, Kinder aufrufen, zurückkehren* — so lange, bis jeder Pfad durchlaufen ist.