Aufgabe A: Ausbreitungsanalyse in Rechnernetzen (20 Punkte)

In dieser Aufgabe sollen die drei vorgegebenen Java-Klassen `Network`, `IP` und `ParseException` zur Angriffsausbreitungsanalyse in Rechnernetzen implementiert werden. Diese Klassen dienen dazu, die theoretisch mögliche Ausbreitung von Angreiferinnen und Angreifern in einem Netzwerk aus Rechnern zu analysieren. Explizit muss in dieser Aufgabe keine interaktive Benutzerinteraktion implementiert werden.

Ein Rechnernetz wird hier als ein Verbund verschiedener unabhängiger Rechner betrachtet, welcher die Kommunikation der einzelnen Rechner untereinander ermöglicht. Die Methoden der zu implementierenden Klassen sollen unter anderem eine Liste der möglicherweise kompromittierten Rechner ermitteln. Hierzu wird, nachdem ein Startpunkt definiert wurde, jeder verbundener Rechner iterativ ermittelt. Nach der Beschreibung der erforderlichen Grundlagen wird im Abschnitt A.4 die genaue Umsetzung dieser vorgegebenen Klassen beschrieben.

A.1 Graphentheorie

Bevor Sie sich mit der Angriffsausbreitungsanalyse auseinandersetzen, werden nachfolgend kurz die Grundlagen der Graphentheorie erläutert. Für die Angriffsausbreitungsanalyse sollen Rechnernetze als Graphen modelliert werden, um diese leichter analysieren zu können.

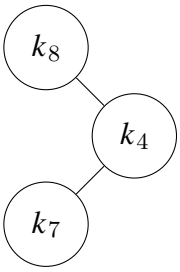
A.1.1 Graph

In der Graphentheorie ist ein ungerichteter Graph ohne Mehrfachkanten eine Menge von Knoten zusammen mit einer Menge von ungerichteten Kanten. Eine ungerichtete Kante ist eine Menge von genau zwei Knoten, die anzeigt, ob zwei Knoten miteinander verbunden sind. Dabei ist die Menge der Kanten eine Teilmenge aller 2-elementigen Teilmengen der Menge der Knoten.

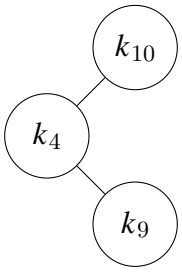
In einem Graphen ist der Grad eines jeden Knotens definiert als die Anzahl der mit ihn verbundenen Kanten. Den Knoten eines Graphen können individuelle Etiketten zugewiesen werden. Bei dieser Aufgabe wird jedem Knoten immer genau ein eindeutiges Etikett zugewiesen, um diese Knoten innerhalb eines Graphen auf der Grundlage dieses Etiketts identifizierbar zu machen. Zwei Graphen sind strukturell gleich, wenn beide die gleiche Menge von etikettierten Knoten und Kanten enthalten.

Wenn Knoten aus verschiedenen Graphen das gleiche Etikett haben, können diese Graphen auf der Grundlage ihrer gemeinsamen Etiketten zusammengeführt werden. Knoten und Kanten, die in mindestens einem Graphen enthalten sind, werden dabei Teil des neuen Graphen. Kanten werden jedoch nicht zweimal eingefügt, sodass es an dieser Stelle immer nur maximal eine Kante zwischen zwei Knoten geben kann.

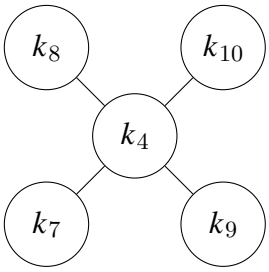
Abbildung A.1 zeigt beispielhaft, wie zwei ungerichtete Graphen, Abbildung A.1a und Abbildung A.1b, auf der Grundlage des gleichen Knotens k_4 zusammengeführt werden können. Wenn nun der Graph Abbildung A.1c wieder mit sich selbst zusammengeführt wird, ergibt sich wieder genau dieser Graph.



(a) Ein ungerichteter Graph mit den etikettierten Knoten k_4, k_7 und k_8.



(b) Ein ungerichteter Graph mit den etikettierten Knoten k_4, k_9 und k_{10}.



(c) Anhand der gemeinsamen Etiketten von Abb. A.1a und Abb. A.1a zusammengeführter Graph.

Abbildung A.1: Beispiel für die Zusammenführung ungerichteter Graphen.

A.1.2 Wege

Ein Weg ist eine Folge von Knoten, bei der jeweils zwei aufeinanderfolgende Knoten durch eine Kante verbunden sind. Diese Folge von Kanten, welche diese Folge von Knoten verbinden, müssen alle verschieden sein. Ein geschlossener Weg (auch Zyklus genannt) ist ein nicht leerer Weg, bei welchem der erste und der letzte Knoten gleich sind. Ein ungerichteter Graph gilt als zusammenhängend, wenn alle seine Knoten paarweise durch einen Weg verbunden sind, sodass es einen Weg von jedem Knoten zu jedem anderen beliebigen Knoten gibt.

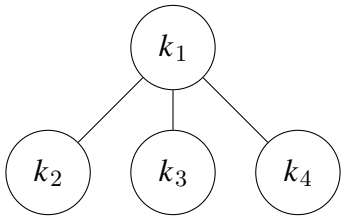
A.1.3 Bäume

Ein ungerichteter Baum ist eine besondere Art von ungerichteten Graph, der zusammenhängend ist und keine geschlossenen Wege enthält. Das bedeutet, dass es zwischen jeweils zwei Knoten eines Baumes genau einen Weg gibt. Die Knoten mit dem Grad 1 werden als Blätter bezeichnet. Die Größe eines Baumes ist definiert als die Anzahl der Knoten im Baum.

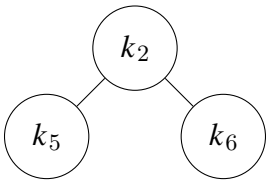
Ein gewurzelter Baum geht von einem Knoten aus, welcher als Wurzel bezeichnet wird, sodass von diesem aus alle anderen Knoten erreicht werden können. Jeder ungerichtete Baum kann an einem beliebigen Knoten aufgegriffen werden, sodass die Kanten von diesem Knoten wegführen, wodurch der ursprüngliche Baum zu einem gewurzelten Baum mit diesem Knoten als Wurzel wird. Ein Teilbaum ist ein Baum, dessen Wurzel ein Knoten eines anderen Baumes ist.

Einem gewurzelten Baum kann eine Höhe zugewiesen werden, welche als die maximal mögliche Länge eines Weges definiert ist, welcher in der Wurzel endet. Hierbei ist die Höhe die Anzahl der Kanten dieses Weges. Die Ebene eines Knotens ist die Anzahl der Kanten entlang des eindeutigen Weges zwischen ihm und dem Wurzelknoten.

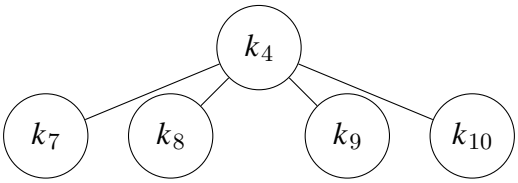
Die Abbildung A.2 zeigt drei verschiedene Bäume, jeder mit einer eindeutigen Wurzel und immer mit der Höhe 1, welche sich jedoch in der Anzahl der Blätter unterscheiden. Angenommen, die Wurzeln k_2 und k_4 würden den gleichnamigen Blättern des Baumes aus Abbildung A.2a entsprechen. Folglich würden sich die entsprechenden Teilbäume zu einem zusammenhängenden Baum mit der Höhe 2 und k_1 als Wurzel vereinen lassen, welcher in Abbildung A.3 dargestellt ist.



(a) Ein Baum mit k_1 als Wurzel und 3 Blättern.



(b) Ein Baum mit k_2 als Wurzel und 2 Blättern.



(c) Ein Baum mit k_4 als Wurzel und 4 Blättern.

Abbildung A.2: Drei verschiedene Bäume, die jeweils eine Höhe von 1 haben.

A.1.4 Ebenenstruktur

Eine Ebenenstruktur ist ein Suchverfahren zum Abrufen von Informationen, welche in einem Graphen enthalten sind. Eine Ebenenstruktur eines ungerichteten gewurzelten Baums ist eine Unterteilung der Knoten in Teilmengen, welche die gleiche Entfernung zu einem gegebenen Wurzelknoten haben. Der gegebene Wurzelknoten wird der Ebene 0 zugeordnet, und die Blätter der tiefsten Ebene werden der Ebene zugewiesen, welche der Höhe des Baumes entspricht.

Um eine Ebenenstruktur zu erstellen, beginnt man bei einem gegebenen Wurzelknoten und speichert dann alle Knoten in der folgenden Ebene, bevor man zu den Knoten in der nächsten tieferen Ebene übergeht, bis schließlich die tiefste Ebene erreicht ist.

Für den Baum aus Abbildung A.3 mit der gegebenen Wurzel k_1 würde sich beispielsweise folgende Ebenenstruktur ergeben:

Ebene 0**:** k_1

Ebene 1**:** k_2, k_3 und k_4

Ebene 2**:** k_5, k_6, k_7, k_8, k_9 und k_{10}

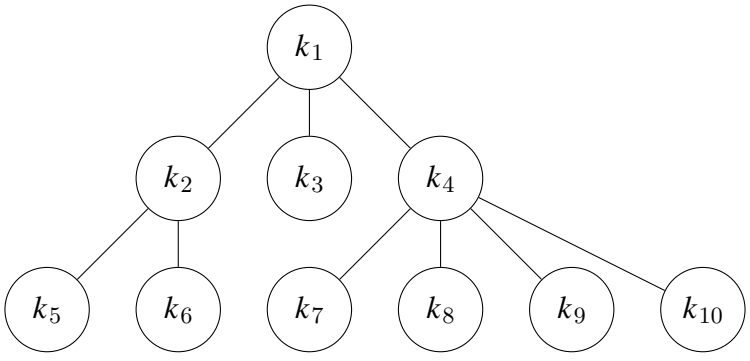


Abbildung A.3: Ein Baum mit Höhe 2 und k_1 als Wurzel.

A.2 Rechnernetze

Ein Rechnernetz ist eine Gruppe von Rechnern, die sich auf Netzknoten befinden. Die Netzknoten verwenden Verbindungen, um miteinander zu kommunizieren. Diese Verbindungen sind in so genannten Netztopologien angeordnet.

A.2.1 Netztopologien

Die Netztopologie ist die Struktur eines Rechnernetzes und kann zur Beschreibung der Anordnung der Netzknoten und Verbindungen genutzt werden. Sie ist dabei eine Anwendung der Graphentheorie, bei der kommunizierende Rechner als Knoten und die Verbindungen zwischen den Geräten als Kanten zwischen den Knoten modelliert werden.

A.2.2 Baumtopologien

Baumtopologien sind Netztopologien, welche dadurch gekennzeichnet sind, dass sie einen ersten bzw. obersten Netzknoten (Wurzel) haben, von der eine oder mehrere Verbindungen ausgehen. Diese führen weiter zu Knoten mit dem Grad 1 (Blätter) oder rekursiv zu Wurzeln von Teilbäumen.

A.2.3 IP-Adressen

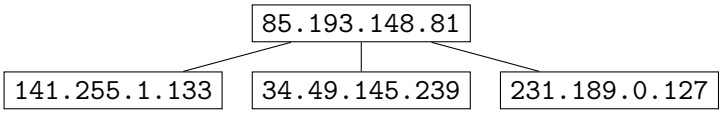
Eine IP-Adresse ist eine Adresse in Rechnernetzen, die den Netzknoten zugewiesen wird, um sie innerhalb des Netzes adressierbar zu machen. Die IP-Adresse entspricht dabei dem Etikett des Knotens. In dieser Aufgabe werden IP-Adressen vereinfacht betrachtet, sodass sie unter anderem immer nur einen konkreten Netzknoten bezeichnen. Folglich haben Netzknoten in dieser Aufgabe immer genau eine feste IP-Adresse zur eindeutigen Identifizierung.

In dieser Aufgabe werden hierzu immer IPv4-Adressen verwendet. Die Dezimalpunktschreibweise ist die übliche Darstellung von IPv4-Adressen. Diese 32-Bit-Adressen werden in dezimaler Form in vier 8-Bit-Blöcken geschrieben, die jeweils durch einen Punkt getrennt sind. Dadurch ergibt sich für jeden 8-Bit-Block ein dezimaler Wertebereich von 0 bis 255. Führende Nullen innerhalb der Blöcke sind nicht erlaubt, sodass zum Beispiel ein- und zweistellige Zahlen nicht mit einer vorangestellten 0 auf ein einheitliches Längenformat gebracht werden dürfen.

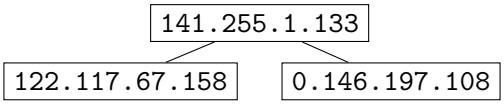
Aufgrund der vereinfachten Betrachtungsweise wird in dieser Aufgabe nicht zwischen Netzklasse oder zwischen Netzanteil und Hostanteil unterschieden. Folglich werden IP-Adressen nur als Kennungen der Netzknoten verwendet. Solange keine Anforderungen der Baumtopologien verletzt werden, können die Adressen von einschließlich `0.0.0.0` bis einschließlich `255.255.255.255` den Netzknoten zugewiesen werden.

A.2.4 Beispiel

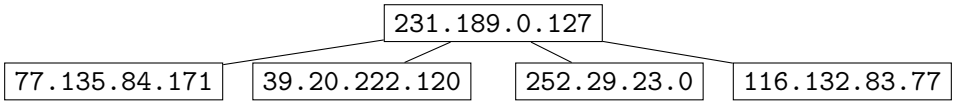
Abbildung A.4 zeigt drei Baumtopologien, jede mit der Höhe 1 und mit einem obersten Netzknoten. Da diese drei Baumtopologien jeweils paarweise einen Netzknoten mit der gleichen IP-Adresse (`141.255.1.133` und `231.189.0.127`) haben, können diese Baumtopologien zu einer zusammenhängenden Baumtopologie mit der Höhe 2 und `85.193.148.81` als Wurzel vereinigt werden, welche in Abbildung A.5 dargestellt ist.



(a) Ein Baum mit `85.193.148.81` als Wurzel und 3 Blättern.



(b) Ein Baum mit `141.255.1.133` als Wurzel und 2 Blättern.



(c) Ein Baum mit `231.189.0.127` als Wurzel und 4 Blättern.

Abbildung A.4: Drei verschiedene Bäume, die jeweils eine Höhe von 1 haben.

Für die Baumtopologie aus Abbildung A.5 mit der gegebenen Wurzel `85.193.148.81` würde sich hierbei folgende Ebenenstruktur ergeben:

Ebene 0**:** `85.193.148.81`

Ebene 1**:** `141.255.1.133`, `34.49.145.239` und `231.189.0.127`

Ebene 2**:** `122.117.67.158`, `0.146.197.108`, `77.135.84.171`, `39.20.222.120`, `252.29.23.0` und `116.132.83.77`

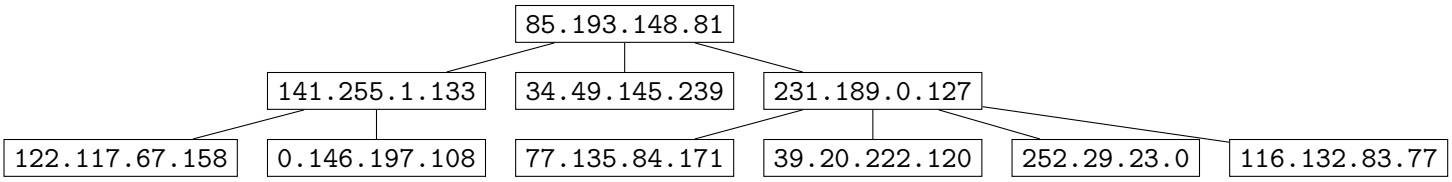


Abbildung A.5: Ein Baum mit Höhe 2 und `85.193.148.81` als Wurzel.

A.3 Klammerschreibweise

Die Haupteingabe des Systems ist eine Beschreibung von Baumtopologien in Klammerschreibweise. Eingeschlossen von einer öffnenden `(` und einer schließenden `)` runden Klammer werden dabei die Netzknoten einer einzelnen Baumtopologie mit der Höhe 1 zusammengefasst. Eine hierarchische Struktur wird dadurch erreicht, dass eingeklammerte Baumtopologien (Teilbäume) innerhalb von eingeklammerten Baumtopologien erlaubt sind. Dabei muss eine Baumtopologie in Klammerschreibweise mindestens zwei verschiedene Knoten (Teilbaum oder Blatt), sprich eine Verbindung enthalten. Dies gilt auch für eingeklammerte Baumtopologien innerhalb von eingeklammerten Baumtopologien.

Die Netzknoten werden durch ihre IP-Adressen bestimmt, die innerhalb der Klammern durch ein Leerzeichen abgegrenzt sind. Da die Klammerschreibweise gültige Baumtopologien beschreibt, darf innerhalb einer Klammerschreibweise, einschließlich der hierarchisch enthaltenen Klammerschreibweisen, jede Adresse nur maximal einmal enthalten sein.

Die erste IP-Adresse nach einer öffnenden Klammer ist die des obersten Netzknoten (Wurzel) und danach folgt die Auflistung der direkten Blätter oder Teilbäume. Sofern nicht ausdrücklich etwas anderes verlangt wird, spielt die Reihenfolge keine Rolle, in welcher die IP-Adressen für jede Ebene aufgeführt werden.

Beispielsweise ist Zeile 1 in dieser Klammerschreibweise die Beschreibung für die Baumtopologie aus Abbildung A.4a. Zeile 2 ist die Beschreibung für die Baumtopologie aus Abbildung A.4b und Zeile 3 ist die Beschreibung für die Baumtopologie aus Abbildung A.4c. Die hierarchische Klammerschreibweise für die zusammengeführte Baumtopologie aus Abbildung A.5 ist in Zeile 4.

```
1 (85.193.148.81 141.255.1.133 34.49.145.239 231.189.0.127)
2 (141.255.1.133 122.117.67.158 0.146.197.108)
3 (231.189.0.127 77.135.84.171 39.20.222.120 252.29.23.0 116.132.83.77)
4 (85.193.148.81 (141.255.1.133 122.117.67.158 0.146.197.108) 34.49.145.239
     (231.189.0.127 77.135.84.171 39.20.222.120 252.29.23.0 116.132.83.77))
```

A.4 Vorgegebene Klassen

In dieser Aufgabe müssen die drei vorgegebenen Java-Klassen `Network`, `IP` und `ParseException` implementiert werden. Diese Aufgabe beinhaltet ausdrücklich keine Kommandozeilenschnittstelle für eine interaktive Benutzerinteraktion ~~umgesetzt~~. Folglich darf Ihre Abgabe auch keine Ein- und Ausgaben oder eine `main`-Methode enthalten. Laden Sie dazu die drei vorgegebenen Klassen zusammen mit diesem Aufgabenblatt von ILIAS[2] herunter. Sie müssen diese drei Java-Klassen so vervollständigen, dass die vorgegebenen Konstruktoren und Methoden die gewünschte Funktionalität bereitstellen. Alle ausdrücklich geforderten Klassen, Konstruktoren und Methoden müssen immer öffentlich (`public`) sein. Verschieben Sie diese drei Klassen noch in beliebige, aber sinnvolle Pakete. Während der Abgabe gibt Ihnen das Einreichungssystem auch Auskunft darüber, ob die geforderten Signaturen gefunden werden konnten.

Sie müssen natürlich weitere Klassen zu Ihrer Abgabe hinzufügen, sowie weitere Attribute, Methoden und Dokumentation zu den drei vorgegebenen Klassen ergänzen. Es dürfen jedoch keine weiteren Klassen mit den Bezeichnern `Network`, `IP` oder `ParseException` in Ihrer Abgabe vorhanden sein. Diese drei Klassen müssen ihre geforderten Konstruktoren und Methodensignaturen bereitstellen und die vorgegebenen Schnittstellen der Java-API implementieren.

Der Aufruf der Methoden darf nicht gegen die zuvor definierten Vorgaben verstoßen. Beispielsweise muss beim Hinzufügen oder Ändern einer Verbindung immer darauf geachtet werden, dass keine Anforderung verletzt wird, wie etwa die Anforderung, dass Baumtopologien keine geschlossenen Wege enthalten dürfen. In den drei geforderten Klassen dürfen Exceptions nur in den ausdrücklich definierten Fällen geworfen werden. In Fehlerfällen wird je nach Anforderung `false`, 0 oder eine leere Liste zurückgegeben.

[2] `https://ilias.studium.kit.edu/goto.php?target=crs_1616969&client_id=produktiv`

A.4.1 `class ParseException extends Exception`

Diese eigens definierte `Exception` signalisiert, dass ein unerwarteter Fehler beim Parsen der textuellen Klammerschreibweise oder Dezimalpunktschreibweise aufgetreten ist.

ParseException.java

```
1 public class ParseException extends Exception {
2     public ParseException(String message) { super(message); }
3 }
```

A.4.1.1 `ParseException(String message)`

Diese `ParseException` muss den spezifizierenden Konstruktor implementieren, der eine neue `Exception` mit der angegebenen Fehlermeldung instanziiert.

A.4.2 `class IP implements Comparable<IP>`

Die Klasse modelliert IPv4-Adressen und legt eine Ordnung für sie fest. Überlegen Sie sich unter anderem, wie die im Konstruktor übergebene Adresse für die Objektinstanz gespeichert werden sollen. Sobald ein Objekt erfolgreich instanziiert wurde, kann die zugehörige Adresse nicht mehr geändert werden. Zwei Objektinstanzen der Klasse `IP` sind gleich, wenn sie die gleiche IP-Adresse repräsentieren.

IP.java

```
1  public class IP implements Comparable<IP> {
2
3      public IP(final String pointNotation) throws ParseException { }
4
5      @Override
6      public String toString() { return null; }
7
8      @Override
9      public int compareTo(IP o) { return 0; }
10 }
```

A.4.2.1 `IP(final String pointNotation) throws ParseException`

Der Konstruktor nimmt eine textuelle Repräsentation einer IPv4-Adresse entgegen und instanziiert ein neues Objekt für diese Adresse. Dabei muss die übergebene Zeichenkette dem Format der Dezimalpunktschreibweise entsprechen. Ist die Überprüfung nicht möglich oder wird das Format verletzt, wird im Konstruktor eine `ParseException` geworfen.

A.4.2.2 `String toString()` Unabhängig von der gewählten internen Darstellung gibt diese Methode für jedes Objekt der Klasse dessen IPv4-Adresse als Zeichenkette in Dezimalpunktschreibweise zurück. Es wird nur diese Zeichenkette von der Methode zurückgegeben, keine anderen Zeichen oder Ausgaben.

A.4.2.3 `int compareTo(IP o)` Diese Methode aus der Schnittstelle `Comparable` erlegt jedem Objekt der Klasse eine natürliche Gesamtordnung auf. IPv4-Adressen werden in aufsteigender Reihenfolge nach ihrem 32-Bit-Wert geordnet. Folglich stehen die Adressen mit dem niedrigsten Bit-Wert (`0.0.0.0`) in der Gesamtordnung vor der Adresse mit dem höchsten Bit-Wert (`255.255.255.255`). Dazu vergleicht diese Methode ihr Objekt mit dem angegebenen Objekt. Beachten Sie hierzu die Dokumentation der Schnittstelle `Comparable` aus der Java-API[3]. Außerdem wird noch gefordert, dass diese natürliche Gesamtordnung ebenso kompatibel mit der Methode `equals` der Klasse ist.

A.4.3 `class Network`

Die Klasse ist das Kernelement dieser Aufgabe und enthält die wichtigsten Methoden für die Angriffsausbreitungsanalyse. Neben dem Anlegen, Hinzufügen und Ändern von Baumtopologien bietet sie auch weitere Methoden zum Abrufen von Informationen über diese Baumtopologien. Überlegen Sie sich unter anderem, wie die übergebenen Baumtopologien in den Objektinstanzen gespeichert werden sollen.

Alle Elemente innerhalb einer Objektinstanz der Klasse müssen immer den formalen Anforderungen entsprechen, wie etwa, dass Baumtopologien keine geschlossenen Wege enthalten dürfen. Achten Sie daher beim Anlegen oder Hinzufügen von Baumtopologien und beim Entfernen oder Hinzufügen von Verbindungen besonders darauf, dass keine Anforderungen an Baumtopologien verletzt werden.

Intern muss immer mindestens eine gültige Baumtopologie vorhanden sein. Das heißt, es muss mindestens eine gültige Verbindung mit zwei Netzknoten vorhanden sein. Die Klasse `Network` muss es auch erlauben, mehrere unabhängige Baumtopologien zu speichern. Das heißt, es kann mehrere gültige Baumtopologien geben, die nicht unbedingt miteinander verbunden sein müssen. Es können jedoch Verbindungen oder ganze Baumtopologien hinzugefügt werden, die diese miteinander verbinden.

Wenn IP-Adressen als Argument übergeben werden, werden sie auf der Grundlage ihrer inhaltlichen Gleichheit und nicht auf der Grundlage ihrer Objektreferenz verglichen. Das bedeutet, dass zwei verschiedene Objektinstanzen, welche die gleiche IP-Adresse repräsentieren, als identisch betrachtet werden. Behandeln Sie ein nicht instanziiertes IP-Adressen-Argument (`null`) immer als eine ungültige und nicht existierende IP-Adresse.

Zwei Instanzen der Klasse `Network` sind dann gleich, wenn sie die gleichen Baumtopologien speichern. Zwei Baumtopologien sind gleich, wenn die ihnen zugrunde liegenden etikettierten Graphen strukturell gleich sind. Setzen Sie hierfür die Methode `equals` in der Klasse entsprechend um.

[3] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Comparable.html

Network.java

```java
import java.util.List;

public class Network {
    public Network(final IP root, final List<IP> children) { }

    public Network(final String bracketNotation) throws ParseException { }

    public boolean add(final Network subnet) { return false; }

    public List<IP> list() { return null; }

    public boolean connect(final IP ip1, final IP ip2) { return false; }

    public boolean disconnect(final IP ip1, final IP ip2) { return false; }

    public boolean contains(final IP ip) { return false; }

    public int getHeight(final IP root) { return 0; }

    public List<List<IP>> getLevels(final IP root) { return null; }

    public List<IP> getRoute(final IP start, final IP end) { return null; }

    public String toString(IP root) { return null; }
}
```

A.4.3.1 `Network(final IP root, final List<IP> children)`

Dieser Konstruktor erzeugt eine neue Objektinstanz und fügt die übergebene nichtleere und gültige Baumtopologie der Objektinstanz hinzu. Dabei hat die Baumtopologie die Höhe 1 mit dem Argument `root` als Wurzel und den Netzknoten (mindestens einer) in der Liste `children` als direkt verbundene Netzknoten. Achten Sie darauf, die Elemente aus dieser Liste für die interne Speicherung zu kopieren, da die Liste unveränderlich sein könnte oder vom Aufrufer nachträglich geändert werden könnte.

Wenn die Argumente keine gültige Baumtopologie beschreiben oder wenn eines der Argumente nicht instanziiert ist oder nicht instanziierte Elemente enthält, wird eine angemessene `RuntimeException` geworfen.

A.4.3.2 `Network(final String bracketNotation) throws ParseException`

Dieser Konstruktor erzeugt eine neue Objektinstanz und fügt ihr die übergebene nichtleere und gültige Baumtopologie hinzu. Die Baumtopologie wird dabei als Zeichenkette in der Klammerschreibweise angegeben und muss dem zuvor angegebenen Format der Klammerschreibweise entsprechen. Ist dies nicht möglich oder wird das Format verletzt, wird im Konstruktor eine `ParseException` geworfen.

A.4.3.3 `boolean add(final Network subnet)` Diese Methode kopiert die Baumtopologien der übergebenen Objektinstanz in ihre eigene Instanz. Wenn Verbindungen oder Netzknoten aus den beiden Instanzen aufgrund ihrer IP-Adressen gleich sind, werden sie zusammengeführt. Wenn die Baumtopologien erfolgreich kopiert werden konnten und sich somit die eigene Instanz intern geändert hat, wird `true` zurückgegeben, ansonsten wird immer `false` zurückgegeben.

Achten Sie dabei darauf, dass es keine Seiteneffekte zwischen diesen beiden Objektinstanzen gibt, sodass beispielsweise eine spätere Änderung an einer der Instanzen die andere nicht beeinflusst.

A.4.3.4 `List<IP> list()` Diese Methode gibt die IP-Adressen aller derzeit in der Objektinstanz vorhandenen Netzknoten in einer neuen und unabhängigen Liste zurück. Die Adressen in dieser Liste sind nach ihrer natürlichen Gesamtordnung aufsteigend sortiert. Achten Sie darauf, dass diese zurückgegebene Liste keine Seiteneffekte auf die Instanz hat.

A.4.3.5 `boolean connect(final IP ip1, final IP ip2)` Diese Methode fügt eine neue Verbindung zwischen zwei bestehenden Netzknoten hinzu. Diese werden durch die beiden Argumente für ihre jeweiligen IP-Adressen bestimmt. Konnte eine neue Verbindung erfolgreich hinzugefügt werden, wird `true` zurückgegeben, ansonsten wird immer `false` zurückgegeben.

A.4.3.6 `boolean disconnect(final IP ip1, final IP ip2)` Diese Methode entfernt eine bestehende Verbindung zwischen zwei Netzknoten. Diese beiden Netzknoten werden anhand der übergebenen IP-Adressen bestimmt. Wenn ein Netzknoten dann den Grad 0 hat, wird er aus der Objektinstanz entfernt, damit seine IP-Adresse neu vergeben werden kann. Wenn es nur noch eine Verbindung gibt, darf diese nicht entfernt werden, da es sonst keine weiteren Netzknoten mehr gäbe. Konnte eine Verbindung erfolgreich entfernt werden, wird `true` zurückgegeben, ansonsten wird immer `false` zurückgegeben.

A.4.3.7 `boolean contains(final IP ip)` Diese Methode gibt `true` zurück, wenn die Objektinstanz einen Netzknoten mit der angegebenen IP-Adresse enthält. Andernfalls wird immer `false` zurückgegeben.

A.4.3.8 `int getHeight(final IP root)` Diese Methode gibt die ganzzahlige Höhe einer Baumtopologie aus. Diese Baumtopologie wird an dem durch das Argument bestimmten vorhandenen Netzknoten aufgegriffen, sodass dieser als oberster Netzknoten betrachtet wird. Wenn die angegebene IP-Adresse nicht intern vergeben ist, wird immer 0 zurückgegeben.

A.4.3.9 `List<List<IP>> getLevels(final IP root)` Diese Methode gibt die Ebenenstruktur einer Baumtopologie in Listenform aus. Die Adressen der Netzknoten einer Ebene werden in eine sortierte Liste eingefügt. Die gesamte zurückgegebene Ebenenstruktur ist wiederum eine aufsteigend sortierte Liste dieser Listen der einzelnen Ebenen. Diese Baumtopologie wird an dem durch das Argument bestimmten vorhandenen Netzknoten aufgegriffen, sodass dieser als ihr oberster Netzknoten gilt. Diesem gegebenen Netzknoten wird die erste Ebene zugewiesen und seine IP-Adresse als einziges Element der ersten inneren Liste eingefügt. Die IP-Adressen der

nachfolgenden Ebene werden dann in die nachfolgende Liste eingefügt. Die Adressen in den inneren Listen für die Ebenen werden in aufsteigender Reihenfolge nach ihrer natürlichen Gesamtordnung sortiert.

Wenn kein Netzknoten mit der angegeben IP-Adresse vorhanden ist, wird nur eine instanziierte leere Liste zurückgegeben. Achten Sie darauf, dass alle zurückgegebenen Listen keine Seiteneffekte auf die Instanz haben.

A.4.3.10 `List<IP> getRoute(final IP start, final IP end)` Diese Methode gibt in einer Liste die einzelnen IP-Adressen der Netzknoten des kürzesten Weges zwischen den durch das jeweilige Argument bestimmten Start- und Endknoten zurück. Die IP-Adresse des Startknotens ist das erste Element in dieser Liste und die IP-Adresse des Endknotens das letzte Element. Die aufeinanderfolgenden Netzknoten in der Liste müssen immer durch eine Verbindung in der Baumtopologie verbunden sein. Existiert einer der beiden angegebenen Netzknoten nicht oder besteht kein Weg zwischen den beiden, wird nur eine instanziierte leere Liste zurückgegeben. Es ist darauf zu achten, dass die zurückgegebene Liste keine Nebeneffekte auf die Instanz hat.

A.4.3.11 `String toString(IP root)` Diese Methode gibt die Klammerschreibweise als Zeichenkette für eine Baumtopologie zurück. An dieser Stelle müssen innerhalb dieser Klammerschreibweise die IP-Adressen für jede Ebene aufsteigend nach ihrem 32-Bit-Wert sortiert sein. Folglich werden die IP-Adressen mit dem niedrigsten Bit-Wert in einer Ebene von links nach rechts zuerst aufgelistet. Diese Baumtopologie wird an dem durch das Argument angegebenen bestehenden Netzknoten aufgegriffen, sodass dieser als dessen oberster Netzknoten gilt. Wenn die angegebene Adresse nicht innerhalb der Instanz zugewiesen ist, wird nur eine instanziierte leere Zeichenkette zurückgegeben.

A.5 Beispielinteraktion

Beachten Sie die folgende programmatische Interaktion einer beispielhaften Ausführung. In dieser `main`-Methode werden die Konstruktoren und Methoden der drei geforderten Klassen mit beispielhaften Parametern aufgerufen. Die erwarteten Ausgaben sind entsprechend als Kommentare direkt über der entsprechenden Zeile eingefügt. Bei einem Abgabeversuch werden diese Konstruktoren und Methoden entsprechend dieser Beispielinteraktion ausgeführt und deren Ausgaben mit den erwarteten Ausgaben verglichen. Bitte beachten Sie, dass das erfolgreiche Bestehen dieses verpflichtenden Tests Voraussetzung für eine erfolgreiche Wertung ist.

Test.java

```java
import java.util.List;

public class Test {
    public static void main(String[] args) throws ParseException {
        // Construct initial network
        IP root = new IP("141.255.1.133");
        List<List<IP>> levels = List.of(List.of(root),
                List.of(new IP("0.146.197.108"), new IP("122.117.67.158")));
        final Network network = new Network(root, levels.get(1));
        // (141.255.1.133 0.146.197.108 122.117.67.158)
        System.out.println(network.toString(root));
        // true
        System.out.println((levels.size() - 1) == network.getHeight(root));
        // true
        System.out.println(List.of(List.of(root), levels.get(1))
                .equals(network.getLevels(root)));

        // "Change" root and call toString, getHeight and getLevels again
        root = new IP("122.117.67.158");
        levels = List.of(List.of(root), List.of(new IP("141.255.1.133")),
                List.of(new IP("0.146.197.108")));
        // true
        System.out.println("(122.117.67.158 (141.255.1.133 0.146.197.108))"
                .equals(network.toString(root)));
        // true
        System.out.println((levels.size() - 1) == network.getHeight(root));
        // true
        System.out.println(levels.equals(network.getLevels(root)));

        // Try to add circular dependency
        // false
        System.out.println(
                network.add(new Network("(122.117.67.158 0.146.197.108)")));

        // Merge two subnets with initial network
        // true
        System.out.println(network.add(new Network(
                "(85.193.148.81 34.49.145.239 231.189.0.127 141.255.1.133)")));
        // true
        System.out
                .println(network.add(new Network("(231.189.0.127 252.29.23.0"
                        + " 116.132.83.77 39.20.222.120 77.135.84.171)")));
```

Test.java

```
44        // "Change" root and call toString, getHeight and getLevels again
45        root = new IP("85.193.148.81");
46        levels = List.of(List.of(root),
47                List.of(new IP("34.49.145.239"), new IP("141.255.1.133"),
48                        new IP("231.189.0.127")),
49                List.of(new IP("0.146.197.108"), new IP("39.20.222.120"),
50                        new IP("77.135.84.171"), new IP("116.132.83.77"),
51                        new IP("122.117.67.158"), new IP("252.29.23.0")));
52        // true
53        System.out.println(
54                ("(85.193.148.81 34.49.145.239 (141.255.1.133 0.146.197.108"
55                        + " 122.117.67.158) (231.189.0.127 39.20.222.120"
56                        + " 77.135.84.171 116.132.83.77 252.29.23.0))")
57                                .equals(network.toString(root)));
58        // true
59        System.out.println((levels.size() - 1) == network.getHeight(root));
60        // true
61        System.out.println(levels.equals(network.getLevels(root)));
62        // true
63        System.out.println(List
64                .of(new IP("141.255.1.133"), new IP("85.193.148.81"),
65                        new IP("231.189.0.127"))
66                .equals(network.getRoute(new IP("141.255.1.133"),
67                        new IP("231.189.0.127"))));
68
69        // "Change" root and call getHeight again
70        root = new IP("34.49.145.239");
71        levels = List.of(List.of(root), List.of(new IP("85.193.148.81")),
72                List.of(new IP("141.255.1.133"), new IP("231.189.0.127")),
73                List.of(new IP("0.146.197.108"), new IP("39.20.222.120"),
74                        new IP("77.135.84.171"), new IP("116.132.83.77"),
75                        new IP("122.117.67.158"), new IP("252.29.23.0")));
76        // true
77        System.out.println((levels.size() - 1) == network.getHeight(root));
```

Test.java

```
79          // Remove edge and list tree afterwards
80          // true
81          System.out.println(network.disconnect(new IP("85.193.148.81"),
82                  new IP("34.49.145.239")));
83          // true
84          System.out.println(
85                  List.of(new IP("0.146.197.108"), new IP("39.20.222.120"),
86                          new IP("77.135.84.171"), new IP("85.193.148.81"),
87                          new IP("116.132.83.77"), new IP("122.117.67.158"),
88                          new IP("141.255.1.133"), new IP("231.189.0.127"),
89                          new IP("252.29.23.0")).equals(network.list()));
90      }
91  }
```