VR planerar ta ibruk ett nytt supermodernt järnvägsnätverk som erbjuder raka och snabba tågförbindelser mellan de största orterna i södra delen av Finland. Bilden nedan illustrerar det planerade järnvägsnätet:

På samtliga järnvägsförbindelser kan tåget hålla en medelhastighet på 330km/h vilket försnabbar trågtrafiken i Finland märkbart!

Du har fått i uppdrag av VR att programmera en applikation som underlättar planerandet av nya tågturer i det nya järnvägsnätet. Din uppgift är att designa och programmera en algoritm som på basen av angiven startpunkt S och destination D beräknar den kortaste rutten. Algoritmen ska mao. beräkna genom vilka orter/tågstationer den kortaste rutten mellan S och D går. Du bör tillämpa algoritmen A* för detta.

VR:s egen superprogrammerare har redan före hen for på semester byggt upp en länkad datastruktur, i detta fall en graf, som innehåller information om varje tågstation, dess koordinater samt hur de olika tågstationerna är länkade samman i järnvägsnätet. Grafen är representerad på enklaste möjliga sätt, dvs. det har definierats en klass *Node* som förutöver *name*, *latitude* och *longitude* även innehåller en *ArrayList* över all grannstationer. I metoden *createGraph* nedan fylls grafen manuellt med värden och returneras men det olyckliga är att superprogrammeraren glömde leverera koden för *Node*-klassen.

```
public ArrayList<Node> createGraph()
{
    //Skapar en nod för varje tågstation
```

```
Node hki = new Node("Helsingfors", 60.1640504, 24.7600896);
Node tpe = new Node("Tammerfors", 61.6277369, 23.5501169);
Node tku = new Node("Abo", 60.4327477, 22.0853171);
Node jyv = new Node("Jyväskylä", 62.1373432, 25.0954598);
Node kpo = new Node("Kuopio", 62.9950487, 26.556762);
Node lhi = new Node("Lahtis", 60.9948736, 25.5747703);

//Förbindelser från Helsingfors tågstation
hki.addNeighbour(tpe); //Tammerfors
hki.addNeighbour(tku); //Åbo
hki.addNeighbour(lhi); //Lahtis

//Förbindelser från Tammerfors tågstation
tpe.addNeighbour(hki); //Helsingfors
tpe.addNeighbour(tku); //Åbo
tpe.addNeighbour(jyv); //Jyväskylä
tpe.addNeighbour(lhi); //Lahtis

//Förbindelser från Åbo tågstation
tku.addNeighbour(hki); //Helsingfors
tku.addNeighbour(tpe); //Tammerfors

//Förbindelser från Jyväskylä tågstation
jyv.addNeighbour(tpe); //Tammerfors

//Förbindelser från Kuopio tågstation
kpo.addNeighbour(lhi); //Lahtis

//Förbindelser från Lahtis tågstation
lhi.addNeighbour(hki); //Helsingors
lhi.addNeighbour(tpe); //Tammerfors
lhi.addNeighbour(kpo); //Kuopio

//Skapar en lista för grafen och sätter in alla noder
ArrayList<Node> graph = new ArrayList();
graph.add(hki);
graph.add(tpe);
graph.add(tku);
graph.add(jyv);
graph.add(kpo);
```

```
        graph.add(lhi);

        return graph;
    }
```

DEL 1 - Klassen Node (5p)

Definiera klassen *Node* enligt följande:

Node

- name : String *- latitude : double* *- longitude : double* *- neighbours: ArrayList<Node>*
+ getName() : String *+ setName(name : String)* *+ getLatitude() : double* *+ setLatitude(latitude : double)* *+ getLongitude() : double* *+ setLongitude(longitude : double)* *+ addNeighbour(neighbour : Node)* *+ getNeighbours() : ArrayList<Node>*

När du byggt upp klassen *Node* på rätt sätt ska du från ditt Java-program kunna anropa metoden *createGraph* som ovan utan errors!

DEL 2 - Skriv ut namnen på alla tågstationer samt lista grannarna (5p)

Definera en metod *showNodesAndLinks* som vid anrop visar namnen på alla noder (tågstationer) i grafen samt alla noders direkta grannar (direkta järnvägsförbindelser till andra orter). Efter först ett

anrop av *createGraph* ovan ska alltså ett anrop av *showNodesAndLinks* resultera i följande skärmutskrift:

Helsingfors
- Tammerfors
- Abo
- Lahtis

Tammerfors
- Helsingfors
- Abo
- Jyväskylä
- Lahtis

Abo
- Helsingfors
- Tammerfors

Jyväskylä
- Tammerfors

Kuopio
- Lahtis

Lahtis
- Helsingfors
- Tammerfors
- Kuopio

DEL 3 - Beräkna uppskattad kostnad H (distans) från en nod till given destination (5p)

Definiera en instansmetod *calculateH* i klassen *Node* som beräknar och returnerar *H*-värdet dvs. i detta fall uppskattad distans från innevarande nod till destinationen. Låt funktionen ta in destinationen (ett *Node*-objekt) som parameter. Distansen mellan två geografiska koordinater kan beräknas t.ex. enligt följande:

```
public double getDistance(double lon1, double lat1, double lon2, double lat2)
{
   lon1 = lon1*Math.PI/180.0;
   lat1 = lat1*Math.PI/180.0;
```

```
        lon2 = lon2*Math.PI/180.0;
        lat2 = lat2*Math.PI/180.0;

        double dlon = lon2 - lon1;
        double dlat = lat2 - lat1;
        double a = Math.pow(Math.sin(dlat/2), 2) + Math.cos(lat1) * Math.cos(lat2) *
Math.pow(Math.sin(dlon/2), 2);
        double c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1-a));
        double km = 6367 * c;
        return km;
    }
```

DEL 4 - Beräkna total kostnad G från en given startpunkt till innevarande nod. (10p)

Definiera en instansmetod *calculateG* i klassen *Node* som beräknar och returnerar *G*-värdet dvs. i detta fall den verkliga distansen från startnoden till innevarande nod via alla eventuella mellannoder. Låt funktionen ta in startpunkten (ett *Node*-objekt) som parameter.

Definiera i detta skede även en instansmetod *getF* som returnerar *F*-värdet, dvs. *G* + *H*.

Pseudokod för metoden *calculateG*

definiera en instansvariabel *previous* (datatyp *Node*) i klassen *Node*

calculateG (Parameter: *source*)

{

Skapa en *double* variabel *G* och initialisera till 0

Skapa en *Node*-variabel *current* och initialisera till innevarande objekt av *Node* (*this*)

Beräkna avståndet från *current* till föregående nod (*current.previous*). Addera avståndet till *G*. Sätt *current* att vara samma som *current.previous*. Upprepa detta så länge

som *current* inte är samma som *source*.

returnera *G*

}

DEL 5 - Gör en egen implementation av A* algoritmen för att beräkna kortaste rutten mellan en given startpunkt och destination (15p)

Definiera en metod *getRoute* som tar in en startpunkt och en destinatation (båda *Node*-objekt) som inparametrar. Gör sedan en egen implementation av A*-algoritmen som beräknar och returnerar en lista (t.ex. *ArrayList*) innehållande alla noder (tågstationer) från startpunkt till destination som man måste passera och i den ordning som motsvarar den kortaste rutten.