käytin toteutuksessa seuraavia tietorakenteita

```
enum class NodeType { WHITE, GREY, BLACK};


struct Way{

   WayID WayId;
   std::vector<Coord> WayCoords;
   struct Way* ChildWay;

};
struct Node{
   Coord NodePosition;
   NodeType NodeColor;
   int NodeDistance;
   struct Node* LastNode;
   std::vector<std::pair<Way*,Node*>> NodeWaysFrom;
};

private:
   // Add stuff needed for your class implementation here
   std::unordered_map<Coord,Node *,CoordHash> nodeContainer;
   std::unordered_map<WayID,Way *> wayContainer;
   //Checks that WayId is in the datastucture
   bool checkForWayId(WayID &id)const;
   //Traverses the Area from child node to parent node untill there is no parent node
   //and returns the vector of the IDs
   void recursiveTraversal(Area * subId, std::vector<AreaID>& areaIdVector);
   //compares two coordinates to see which one is bigger.
   bool compareCoords(Coord a, Coord b);
   //function that finds a way beatween two coordinates
    std::vector<std::tuple<Coord, WayID, Distance> > bfs(Node *source,Node *goal);
```

Käytin structia noden ja wayn datan säilyttämiseen, koska se vaikutti järkevältä tavalta säilyttää monia eri tietoja yhdessä paikassa.

Valitsin unordered_map tietorakenteeksi, koska tässä harjoitustyössä järjestyksellä ei ole niin hirveästi väliä, vaan tärkeämpi on, että se on tehokas ja jos järjestystä tarvitaan niin NodeWaysFrom muodostaa graafin niistä tietorakenteen tiedoista.

Useimmat funktiot ovat mielestäni aika selkeitä, joten en kommentoi niitä sen tarkemmin.

add_way lisää myös wayn alussa ja lopussa olevat nodet, siinä on if, else if, else rakenne, koska yksi node voi olla monessa wayssa mukana, joten niitä ei kannata luoda joka kerta uudelleen.

bfs eli breadth first search on toteutettu pääasiassa luentokalvojen mukaan, paitsi siinä kohdassa, kun löytyy päämäärä, jolloin mennään LastNodea käyttäen graafin alkuun laittaen nodeja samalla vektoriin.