käytin toteutuksessa seuraavia tietorakenteita

```
struct Data{

    Name DataName;
    PlaceType DataPlaceType;
    struct Coord DataCoord;
};
struct Area{
    AreaID AreaId;
    Name AreaName;
    struct std::vector<Coord> AreaCoords;
    struct Area* parentArea;
    struct Area* childArea;
};
```

```
private:
    // Add stuff needed for your class implementation here
    //unordered_set
    std::unordered_map<PlaceID,Data *> container;
    std::unordered_map<AreaID,Area *> areaContainer;

    //Checks that PlaceId is in the datastucture
    bool checkForId(PlaceID &id)const;
    //Checks that AreaId is in the datastucture
    bool checkForAreaId(AreaID &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);
```

Käytin structia place ja area 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.

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

places_coord_order funktiossa käytin lambdaa lisätäkseni compareCoords funktion sort algoritmin sisälle. compareCoords funktio toimii tehtävänannon mukaisesti eli se vertaa koordinaattien ”normaalia” euklidista etäisyyttä origosta ja palauttaa myönteisen vastauksen, jos a on pienempi tai yhtä pieni ja kielteisen jos b on pienempi.