Valittujen tietorakenteiden ja tehtyjen ratkaisujen perustelu

Päädyin käyttämään harjoitustyöni päätietorakenteina unordered_mappeja, joissa hakuavaimena oli joko tietyn paikan tai avaimen tunnus (PlaceID / AreaID), ja tätä vastaava arvo oli paikkaa tai aluetta vastaava struct, jossa on kootusti kaikki muu kyseiseen paikkaan tai alueeseen liittyvä tieto. C++:lla kirjoitettuna nämä structit ja tietorakenteet olivat siis seuraavan kaltaisia:

```
struct Place
{
    Name placeName;
    PlaceType type;
    Coord location;
};
```

```
struct Area
{
    Name areaName;
    std::vector<Coord> shape;
    bool isSubArea;
    AreaID parentAreaID;
    std::vector<AreaID> childrenAreas;
};
```

```
std::unordered_map<PlaceID,Place> places_;
std::unordered_map<AreaID,Area> areas_;
```

Nämä valinnat syntyivät melko itsestään selvästi sen jälkeen, kun olin lukenut harjoitustyön tehtävänannon sekä perehtynyt siihen, mitä luokan datastructures eri toimintojen on tarkoitus tehdä. Pääsyy valita unordered_map tietorakenteeksi oli juurikin se, että dokumentaatiossa sanotaan, että seuraavia operaatiota kutsutaan useammin kuin muita: **get_place_name_type**(PlaceID id), **get_place_coord**(PlaceID id) ja **get_area_name**(AreaID id). Edellä mainitut funktiot ottavat parametrinaan vain paikan / alueen ID:n ja haluavat, että kustakin paikasta/alueesta palautetaan jotain tietoa. Kun on käytössä unordered_map, saadaan kyseistä avainta (eli ID:tä) vastaava paikka/alue haettua tietorakenteesta keskimäärin vakioaikaisesti. Tällöin nämä kaikki edellä mainitut useasti kutsuttavat operaatiot ovat erittäin tehokkaita, keskimäärin vakioaikaisia. Näissä funktioissa täytyy tietenkin tarkistaa aina se, että löytyykö kyseistä ID:tä vastaavaa alkiota tietorakenteesta, mutta sekin saadaan tehtyä keskimäärin vakioaikaisesti käyttämällä unordered_mapin metodia .find(). Myös monet muutkin luokan julkisen rajapinnan operaatioista ottavat parametrinaan joko paikan tai alueen ID:n, joten tälläkin perusteella tuo tietorakennevalinta oli looginen.

Syy miksi valitsin unordered_mapin enkä tavallista mappia on se, että unordered_mappiin lisääminen, sieltä poistaminen ja tietyn alkion etsiminen on keskimäärin vakioaikaista, verrattuna tavalliseen mappiin, jossa edellä mainittujen operaatioiden asymptoottinen tehokkuus on O(log n). Ottaen huomioon muut datastructures -luokan julkisen rajapinnan operaatiot, ei alkioiden jatkuva järjestyksessä pitäminen ollut myöskään erityisen tarpeellista. ID:n mukaan alkioitahan ei tarvinnut järjestää missään muutenkaan, koska pääohjelma hoiti sen, joten mikäli alkioita olisi halunnut pitää jatkuvasti jossain ohjelman kannalta

tarpeellisessa järjestyksessä, olisi tarvittu erilainen tietorakenne, joka olisi saattanut hidastaa noita kolmea useimmiten kutsuttavaa operaatiota. Toinen vaihtoehto olisi tietysti ollut käyttää kahta tietorakennetta, sekä alueita että paikkoja kohti, jossa toisessa alkiot olisivat olleet järjestyksessä ja toisessa ei, mutta koska alkioita tarvitsi järjestää vain muutamassa operaatiossa, en nähnyt tätä tarpeelliseksi. Useampien tietorakenteiden käyttö olisi myös muutoinkin hidastanut muita operaatioita, koska kaikki lisäykset ja poistot olisi tarvinnut tehdä kahteen eri tietorakenteeseen, sekä samalla kokonaisuuden hallitsemisesta ja hahmottamisesta olisi tullut haastavampaa. Tämän vuoksi näin järkevämmäksi järjestää alkiot erikseen vain silloin kun sille oli tarvetta.

Alkioita tarvitsi järjestää funktioissa **places_alphabetically**, **places_coord_order** ja **places_closest_to**. Kaikissa näissä järjestämiseen hyödynsin funktioissa paikallista map-rakennetta, joka järjestää alkiot niiden hakuavainten mukaisessa järjestyksessä. Tällöin järjestäminenkin saadaan tehtyä melko tehokkaasti, sillä mappiin lisääminen käyttämällä metodia .insert() on asymptoottiselta tehokkuudeltaan logaritminen operaatio. Käydessä kaikki alkiot läpi alkuperäisestä unordered_mapista ja lisäämällä ne mappiin for-loopissa, saadaan alkiot järjestettyä tehokkuudella O(n*log(n)). Sijainnin mukaan järjestämistä varten kirjoitin erillisen funktion void **get_places_in_order**(Coord xy, PlaceType type, std::map<double,std::multimap<int,PlaceID>> & places_in_order) luokan yksityiseen rajapintaan, jota molemmat **places_coord_order** ja **places_closest_to** kutsuvat. Tuo funktio järjestää paikat niiden etäisyyden mukaan suhteessa parametrina annettuun sijaintiin xy lisäämällä ne viitteenä annettuun mappin insertiä käyttäen, ja tarvittaessa järjestää vain tietyn tyyppiset paikat. (Jos parametrina annetaan NO_TYPE, järjestetään kaikki paikat, mikäli muuta, vain parametrina annetut paikat) Sijainnin mukaan järjestämisessä hyödynsin tietorakennetta std::map<double,std::multimap<int, PlaceID>>, jossa ulomman mapin hakuavain on etäisyys. Tätä avainta vastaa toinen map-rakenne, koska tietyillä eri sijainneissa olevilla paikoilla saattoi olla täsmälleen sama etäisyys paikkaan, jonka etäisyydestä oltiin kiinnostuneita. Tätä varten tuon sisemmän map-rakenteen hakuavain on tietyn paikan y-koordinaatti, sillä harjoitustyön ohjeessa sanottiin, että paikat järjestetään ensisijaisesti niiden etäisyyden perusteella, mutta mikäli etäisyys on sama, niin matalamman y-koordinaatin omaava paikka tulee olla ensin. Lisäämällä paikkoja tällaiseen rakenteeseen saatiin ne järjestettyä sijainnin mukaan, sillä nyt ne paikat, joilla on täsmälleen sama etäisyys keskenään, on järjestettynä niiden y-koordinaattien mukaan sisemmässä mapissa. Mikäli paikalla on uniikki etäisyys, on sitä vastaava map-rakenne tällöin itseasiassa pari. Kun **get_places_in_order** on järjestänyt paikat sijainnin mukaan, voidaan ne lukea mapista vectoriin julkisen rajapinnan operaatioissa **places_coord_order** ja **places_closest_to** hyödyntämällä for-looppia. Vaikka näissä molemmissa operaatioissa on for-loopin sisällä toinen for-looppi, se ei tarkoita, että nämä olisivat neliöllisiä operaatioita. Johtuen juurikin tuosta järjestämisperiaatteesta sekä käyttämästäni tietorakenteesta map<double,map<int,PlaceID>> voi tulla vastaan tilanne, jossa useammalla paikalla on täsmälleen sama etäisyys. Tässä tilanteessa täytyy mennä

sisempään for-looppiin käsittelemään nämä alkiot. Sisempään for-looppiin meneminen kuitenkin tarkoittaa sitä, että ulommalla for-loopilla on vähemmän alkioita käsiteltävänä, sillä osa paikoista on kytkeytynyt tähän tiettyyn hakuavaimeen double. Näin ollen tuo rakenne käy kaikki paikat kertaalleen läpi, jolloin sen tehokkuus on lineaarinen, vaikka se voikin näyttää ensisilmäyksellä muulta. Funktion **places_closest_to** tapauksessa looppaus katkaistaan käyttämällä komentoa break silloin kun täytettävän vektorin koko on kolme (kuten haluttiinkin), jolloin sen tehokkus on itseasiassa vakioaikainen, koska mapissa olevat alkiot on jo valmiissa järjestyksessä, ja toistorakenne keskeytetään käyttämällä komentoa break silloin, kun vektorin koko on kolme.

Funktiossa **common_area_of_subareas, all_subareas_in_area** ja **subarea_in_areas** alkioiden järjestämistä ei tarvittu, sillä tiettyjen alueiden ylempiä, sekä suoria että epäsuoria, ja alempia, sekä suoria että epäsuoria, yli- ja alialueita pystyttiin käymään läpi hyödyntämällä rekursiivisia funktioita, koska käyttämäni struct-rakenne alueille muodosti puurakenteen, jossa jokaisella alueella on tiedossa ylemmän alueensa ID sekä vektori omista alialueistaan. Näin kiinnostuksen kohteena olevia alkioita pystyttiin käymään läpi esi- ja jälkijärjestyksessä hyödyntämällä luokan yksityisessä rajapinnassa määriteltyjä rekursiivisia funktioita void **get_subareas_**(AreaID id,std::vector<AreaID> & subareas_already_added) ja void **get_upper_areas_**(AreaID id1, std::vector<AreaID> & upper_areas). Nuo molemmat funktiot siis lisäävät rekursiivisesti niille viitteenä annettuun vectoriin alueita, jotka ovat parametrina annetun alueen yli- tai alialueita.

Toteutuksen asymptoottisesti hitain operaatio on **common_area_of_subareas,** jonka asymptoottinen tehokkuus on O(n^2). Tästä en ole aivan varma onko sitä edes mahdollista saada tehokkaammaksi käyttämilläni tietorakenteilla, sillä mikäli puurakenne on kaksihaarainen, ja parametrina annetut alueet ovat puun lehtiä, tulee neliöllinen määrä vertailuja väistämättä tehtyä, koska funktio halusi nimenomaisesti ensimmäisen yhteisen ”esi-isän”, jolloin sillä ei ole merkitystä, ovatko parametrina annetut alueet ”samaa sukupolvea” vai eivät. En sulje sitä mahdollisuutta pois, että tätä operaatiota voisi saada vielä tehokkaammaksi käyttämällä erilaisia tietorakenteita, mutta itse en ainakaan keksinyt tapaa saada tätä tehokkaammaksi käyttämilläni tietorakenteilla. Ja vaikka tämän operaation tehokkuutta olisi mahdollista saada tehokkaammaksi eri tietorakenteilla, hyväksyn nyt tehdyn kompromissin, sillä käyttämieni tietorakenteiden ansiosta ohjelmassa usein kutsuttavat operaatiot ovat erittäin tehokkaita, eri tietorakenteilla näin ei välttämättä olisi. Tämä **common_area_of_subareas** -operaatio toimii käytännössä siten, että se käy rekursiivisesti läpi kummankin alkion esi-isät ja lisää ne omiin vektoreihinsa hyödyntämällä luokan yksityisessä rajapinnassa määriteltyä funktiota **get_upper_areas_**. Kun tämä on tehty, ja molempien alkioiden yläalueet ovat omissa vektoreissaan, joissa molempien alueiden ensimmäiset eli suorat yläalueet ovat vektorien ensimmäisinä alkioina jne. Koska olemme kiinnostuneita ensimmäisestä yhteisestä yläalueesta, ei vektoreiden järjestystä parane lähteä muuttamaan. Koska parametrina annettujen alueiden

sukupolvella ei ollut väliä, voi käydä niin, että alueen 1 ensimmäinen suora yläalue on alueen 2 toinen yläalue, ollen siis alueen 2 epäsuora yläalue sekä ensimmäinen alueiden 1 ja 2 yhteinen yläalue. Tämän vuoksi joudumme käyttämään for-looppia for-loopin sisällä, jossa verrataan aluksi alueen 1 ensimmäistä yläaluetta kaikkiin alueen 2 ylempiin alueisiin jne. Kun löydetään yhteinen alue, looppaus lopetetaan breakilla. Huonoimmassa tapauksessa yhteinen yläalue on kummassakin vektorissa viimeisenä, jolloin operaatio on neliöllinen. Tämän vuoksi **common_area_of_subareas:**n tehokkuus on O(n^2).