TIE-20100 hw2
Henri Laakso
240062
henri.m.laakso@student.tut.fi

Perustelut ja tehokkuusarviot

Tietorakenteeni pohjana toimii undordered_map sen nopean haun takia haettaessa id:llä. Lisäksi pidän yllä kahta id listaa nimi, että palkkajärjystykselle vektoreissa.

Unordered_map oli varsin ilmeinen valinta tietosäiliöksi, jonne kaikki työntekijät talletettiin, sillä tietosäiliöltä tarvittiin lähinnä vain nopeaa hakua avaimella.
Järjestyksien ylläpito taas vaatikin sitten hieman enemmän pohdintaa mapin ja vektorin välillä, mutta päädyin lopulta vektoriin, sillä kaikissa kohdissa, joissa piti palauttaa jokin lista oli sen muoto vektori. Mapilla toteutus olisi taas vaatinut järjestetyn mapin kääntämisen vektoriksi, joka taas on lineaarinen operaatio, joka olisi hidastanut joitain operaatioita suuresti, vaikka viime järjestelyn jälkeen ei olisi tehty muutoksia.
Vektorin ainoaksi huonoksi puoleksi jäi varsin epätehokas alkioiden poisto, joka teki koko poisto operaatiosta lineaarisen. Tämä haittapuoli kompensoi vakioaikaisia jo järjestetyn listan palautusta. Lisäksi palkka- ja nimivektorit tekevät sorttauksensa tarvittaessa, toisin kuin insertillä, mikä taas siirtää kuormaa sinne minne se kuuluukin.

Pomon alaisten sorttauksen olisi voinut myös hoitaa mergeämällä uusi lista aina vanhaan alaisten listaan, mutta koin tehokkaammaksi sortata alaiset jos henkilön usorted lippun on epätosi aina alaisia pyydettäessä. Merge olisi ollit lineaarinen operaatio, kun taas sorttaus on N*log(N). Tosin tällä ei ole mitään väliä, koska operaatio mielletään vakioaikaiseksi vakiomääräisen alaismäärän takia.

Tehokkuutta arvioidessa oletan työntekijähierarkiapuun olevan jokseenkin tasapainossa. Erikistapauksissa, kuten kaikki työntekijät vain yhden pomon alla, tai kaikki työntekijät muodostavat pomo-alainen ketjun tapaukset tuottavat yleensä N tehokkuuden keskimäärisen ollessa log(N) ja N^2 kskimääräisen ollessa N*log(N). Yleisimmät erikoistapaukset olen vielä käynyt erikseen läpi funktion tehokkuusarviossa.

Muutaman yleisesti käytetyn operaation tehkkuus:
Unordered_map::find on vakioaikainen operaatio tasaisesti hajautetulla mapilla. Huonoimmassa tapauksessa lineaarinen, jos kaikki alkiot samassa bucketissa, jota ei pitäisi tässä ohjelmassa tapahtua.
Unordered_map::operator[] vakioaikainen jos tasaisesti hajautettu. Huonoin tapaus sama kuin yllä.

HW3 asiaa
Valitsin kaveriverkoston pohjaksi jokaiselle henkilölle asetettavan mapin kevereista ja heidän välisistä kustannuksista. Mappi ei ole tässä kaikista tehokkain ratkaisu, mutta sen ansiosta operaatiot tuottavat samat vastaukset ajokerroista riippumatta, sillä niiden sisältö on sortattu.
Lisäoperaatioita olisi vielä voinut viilata, mutta olen erityisen tyytyväinen jo siihen, että sain ne toimimaan.
Leave_cheapest_friendforestin minimum spanning treen toteutin kruskalin algoritmilla, koska se oli huomattavasti helpompi ymmärtää. Primin algoritmi olisi ekä tuottanut

tehokkaamman lopputuloksen, sillä sen nopeus riippuu vahvemmin solmujen määrästä, kuin reunojen määrästä, toisin kuin kruskalin algoritmi.

Datastructure();
Vakioaikainen, sillä rakenteiden alustaminen on vakioaikaista, eikä tehdä mitään omaa.

~Datastructure();
Lineaarinen, sillä jokainen tyhjennysoperaatioista on lineaarinen.

add_person(string name, PersonID id, string title, Salary salary);
Lineaarinen, sillä ei tehdä sorttausta sun muuta ja unordered_mappiin ja vektorin loppuun sijoittamiset ovat vakioaikaisia operaatioita.

remove_person(PersonID id);
Lineaarinen, sillä molemmat listat joudutaan käymään läpi etsittäessa oikea id. Tämä on ohjelmani suhteessa hitain operaatio mahdolliseen nopeuteen verrattuna johtuen tietorakenne valinnoista, mutta mielestäni ratkaisuni on hyvin perusteltu ja tukee ohjelman tehokasta toimintaa kokonaisuudessa.

get_name(PersonID id);
Unordered_mapista haku on vakioaikainen operaatio.

string get_title(PersonID id);
Unordered_mapista haku on vakioaikainen operaatio.

Salary get_salary(PersonID id);
Unordered_mapista haku on vakioaikainen operaatio.

find_persons(string name);
Lineaarinen, sillä kaikki henkilöt pitää käydä koska nimiä ei ole järjestety minnekään.
Tää pätee tosin vain oletuksella, että sorttaus on vakioaikainen operaatio, koska sortttavia on vähän.
Erikoistapaus jossa kaikilla henkilöillä sama nimi johtaa N*log(N) nopeuteen sorttauksen takia.
Tämän voisi kyllä kanssa tulkita N*log(N) nopeutiseksi, koska tietyn nimisiä on keskimäärin x*N, missä x on tietyn nimen esiintymismäärä koko määrästä keskimäärin, jolloin sorttaus kestää x*N*log(N) joka on N*log(N) nopeus.

personnel_with_title(string title);
Lineaarinen, sillä käydään läpi kaikki työntekijät.

change_name(PersonID id, string new_name);
Unordered_mapista haku on vakioaikainen operaatio.

change_salary(PersonID id, Salary new_salary);
Unordered_mapista haku on vakioaikainen operaatio.

add_boss(PersonID id, PersonID bossid);
log(N), sillä käydään läpi puuta ylöspäin etsittäesse onko id bossid:n pomo.
Muut operaatiot ovat vakioaikaisia.

size();

Unordered_mapin size() on vakioaikainen operaatio.

clear();
Lineaarinen, sillä jokainen tyhjennysoperaatioista on lineaarinen.

underlings(PersonID id);
Vakioaikainen jos työtenkijähierarkia on jokseenkin tasapainossa, silloin sortataan vain tietty määrä alkioita, joka on vakioaikainen operaatio.
Parhaassa tapauksessa lista on jo sortattu ja tehdään vain suora haku unrdered_mappiin.

personnel_alphabetically();
Vakioaikainen jos jo sortattu.
N*log(N) sorttaus.

personnel_salary_order();
Vakioaikainen jos jo sortattu.
N*log(N) sorttaus.

ind_ceo();
Unordered_mapista haku on vakioaikainen operaatio.

PersonID nearest_common_boss(PersonID id1, PersonID id2);
Log(N), sillä etsitään pomoketju ylimpään, ja sen jälkeen käydään ketjut läpi vertaillen keskenään. Puun läpi ketju on logaritminen verratuna puun kokoon, jos puu on edes jokseenkin tasapainossa, eikä vain yksi ketju alaisia ja pomoja. Pahimmassa tapauksessa olisi tämä lineaarinen operaatio.

higher_lower_ranks(PersonID id);
Lineaarinen, sillä käydään läpi kaikki työntekijät kerran + log(N) kun etsitään työntekijän paikka puussa, eli lopulta lineaarinen.

PersonID min_salary();
Lineaarinen jos ei sortattu (nth_element()).
Vakioaikainen jos sortattu.

PersonID max_salary();
Lineaarinen jos ei sortattu (nth_element()).
Vakioaikainen jos sortattu.

PersonID median_salary();
Lineaarinen jos ei sortattu (nth_element()).
Vakioaikainen jos sortattu.

PersonID first_quartile_salary();
Lineaarinen jos ei sortattu (nth_element()).
Vakioaikainen jos sortattu.

PersonID third_quartile_salary();
Lineaarinen jos ei sortattu (nth_element()).
Vakioaikainen jos sortattu.

void add_friendship(PersonID id, PersonID friendid, Cost cost)
Logaritminen henkilöiden kavereihin verrattuna.

void remove_friendship(PersonID id, PersonID friendid)
Logaritminen henkilöiden kavereihin verrattuna.

vector<pair<PersonID, Cost>> get_friends(PersonID id)
Lineaarinen henkilön kavereihin verrattuna.

vector<pair<PersonID, PersonID>> all_friendships()
Aika kasvaa solmujen ja reunojen tulona.

vector<pair<PersonID, Cost> > shortest_friendpath(PersonID fromid, PersonID toid)
Aika kasvaa solmujen ja reunojen summana.

bool check_boss_hierarchy()
Lineaarinen? Koska käydään kaikki solmut läpi kerran.

vector<pair<PersonID, Cost>> cheapest_friendpath(PersonID fromid, PersonID toid)
Ikävä kyllä en saanut prioriteettijonoa toimimaan oikein jostain syystä, vaan se laittoi alkiot välillä väärään järjestykseen, joten algoritmi ei täysin toimi. Muilta osin sen pitäisi tosin olla kunnossa. Tuli kyllä tutkittua dijkstran nopeutta mutta en kyllä osaa sen enempää sanoa, kuin sen mitä vikipediasta luin. Nopeus riippuu molemmista solmuista ja reunoista, sekä prioriteettijonon nopeudesta.

pair<unsigned int, Cost> leave_cheapest_friendforest()
Toteutus on kruskalin algoritmin mukainen.
Tämän algoritmin nopeus vaihtelee suuresti, sillä jokaisen uuden kaverisuhteen jälkeen tulee tarkistaa onko graafissa syklejä. Jos graafi on hyvin yhtenäinen vie syklien tarkistus aluksi vähän, mutta loppua kohden jokainen uusi tarkistus lähenee lineaaris aikaista operaatiota. Tutkimani lähteet kuitenkin väittivät nopeuden olevan O(solmut*log(solmut)).