**Dokumentaatio**

[junyuan.fang@tuni.fi](mailto:junyuan.fang@tuni.fi)

H292042

**Tietorakenteen valinta:**

**//phase 1////////////////////////////**

//Place

struct **Place**{

Name name\_;

PlaceType type\_;

Coord xy\_;

};

unordered\_map<PlaceID,Place> placeUnOrMap\_;

//Area

struct **Area**{

AreaID area\_ID\_;

Name name\_;

vector<Coord> coords\_;

Area \* parent\_=nullptr;

vector<Area\*> subArea\_;

};

unordered\_map<AreaID,Area> areaUnOrMap\_;

Unordered\_map sopii työmme, koska ”alkion poisto”, ”tietyn etsintä”, ”lisäysmuualle”on keskimäärin Θ(1) tai pahimmillaan O(n), jonka tapahtuva todennäköisyys on todella pieni.

**//phase 2////////////////////////////**

struct **besideInfo**{

**besideInfo**(){

d=-1;

ptr=nullptr;

}

Distance d;

CoordData\* ptr=nullptr;

};

struct **CoordData** {

**CoordData**(){

this->coord=NO\_COORD;

this->besideInfo={};

this->colour=Colour::WHITE;

this->from=nullptr;

this->fromWay=NO\_WAY;

this->d=-1;

}

Coord coord=NO\_COORD;

unordered\_map<WayID, besideInfo> besideInfo={};

Colour colour=Colour::WHITE;

CoordData\* from=nullptr;

WayID fromWay=NO\_WAY;

Distance d=-1;//infinity

//d is to next's distance

};

struct **PQueComCoorD\_Ptr**{

bool operator ()(const CoordData\* a, const CoordData\* b){

return a->d > b->d; //just like reload <, so smaller one goes out from the p\_queue first

}

unordered\_map <WayID, vector<Coord>> wayIDUnordMap\_;

unordered\_map<Coord, CoordData\*, CoordHash> coordUnordMap\_;//only crossroada are added

Koska työssä on käytetty prioriteettijonoa, yksi tapa on luoda struct, ja määrittää struct sisällä vertailuoperaation.

Hajautustaulu sopii meidän työhön. Kurssi puolella saatu CoordHash auttaa iso kokoisen coordUnordMap:n ”rehash”.

**Toteutksessa käytetyt STL funtiot, ja niiden asymptoottinen notaatio**

Unordered\_map:

find, keskimäärin Θ(1) tai pahimmillaan O(n)

empty, O(1)

erase, keskimäärin Θ(1) tai pahimmillaan O(n)

size, O(1)

clear, keskimäärin Θ(1) tai pahimmillaan O(n)

insert, keskimäärin Θ(1) tai pahimmillaan O(n)

Vector: back, O(1)

pushback, amortisoitu Θ(1) tai pahimmillaan O(n)

size, O(1)

clear, O(n)

sort(): O(nlog(n))

**//phase 1////////////////////////////**

// Estimate of performance: O(1)

// Short rationale for estimate:Only used size() to return int. and its comlexity is O(1)

int **place\_count**();

// Estimate of performance: O(n)

// Short rationale for estimate: because unordered\_map's clear() is O(n)

void **clear\_all**();

// Estimate of performance: O(n)

// Short rationale for estimate:for-loop dominate the time

std::vector<PlaceID> **all\_places**();

// Estimate of performance: in average theta(1),worst O(n)

// Short rationale for estimate:complexity depends on unordered\_map's find() and size(), and both are theta(1)

bool **add\_place**(PlaceID id, Name const& name, PlaceType type, Coord xy);

// Estimate of performance: in average theta(1),Worst case: O(n)

// Short rationale for estimate:complexity depends on unordered\_map's find()

std::pair<Name, PlaceType> **get\_place\_name\_type**(PlaceID id);

// Estimate of performance: in average theta(1),Worst case: O(n)

// Short rationale for estimate:complexity depends on unordered\_map's find()

Coord **get\_place\_coord**(PlaceID id);

// We recommend you implement the operations below only after implementing the ones above

// Estimate of performance: O(n log(n))

// Short rationale for estimate: same as sort()'s complexity and unordered\_map's find() in average is theta(1)

std::vector<PlaceID> **places\_alphabetically**();

// Estimate of performance: O(n log(n))

// Short rationale for estimate:same as sort()'s complexity and unordered\_map's find() in average is theta(1)

std::vector<PlaceID> **places\_coord\_order**();

// Estimate of performance: O(n\*m)/ O(n)

// Short rationale for estimate:m is string's length, because strings area shart so we assume m=constant

std::vector<PlaceID> **find\_places\_name**(Name const& name);

// Estimate of performance: O(n)

// Short rationale for estimate: one for-loop

std::vector<PlaceID> **find\_places\_type**(PlaceType type);

// Estimate of performance:in average theta(1),Worst case: O(n)

// Short rationale for estimate:complexity depends on unordered\_map's find(). Because we find it from datastructure then we change it

bool **change\_place\_name**(PlaceID id, Name const& newname);

// Estimate of performance: in average theta(1),Worst case: O(n)

// Short rationale for estimate:complexity depends on unordered\_map's find(). Because we find it from datastructure then we change it

bool **change\_place\_coord**(PlaceID id, Coord newcoord);

// We recommend you implement the operations below only after implementing the ones above

// Estimate of performance: in average theta(1),Worst case: O(n)

// Short rationale for estimate:complexity depends on unordered\_map's find() and insert()

bool **add\_area**(AreaID id, Name const& name, std::vector<Coord> coords);

// Estimate of performance: in average theta(1),Worst case: O(n)

// Short rationale for estimate:no for loop, and unordered\_map's find() dominate the time

Name **get\_area\_name**(AreaID id);

// Estimate of performance: in average theta(1),Worst case: O(n)

// Short rationale for estimate:no for loop, and unordered\_map's find() dominate the

std::vector<Coord> **get\_area\_coords**(AreaID id);

// Estimate of performance: O(n)

// Short rationale for estimate:one for-loop for using push\_back(), and pushback() is armortized theta(1)

std::vector<AreaID> **all\_areas**();

// Estimate of performance: O(n)

// Short rationale for estimate:one for-loop for using push\_back(), and pushback() is armortized theta(1)

bool **add\_subarea\_to\_area**(AreaID id, AreaID parentid);

// Estimate of performance: O(n)

// Short rationale for estimate:while-loop increase the time complexity

std::vector<AreaID> **subarea\_in\_areas**(AreaID id);

// Non-compulsory operations

// Estimate of performance: O(1)

// Short rationale for estimate:there we only change the "flag"'s value so is constant

void **creation\_finished**();

// Estimate of performance: O(n)

// Short rationale for estimate:Depth First Traversal is O(n + m), where n is the number of nodes, and m is the number of edges.

std::vector<AreaID> **all\_subareas\_in\_area**(AreaID id);

// Estimate of performance: O(n)

// Short rationale for estimate: there was one for loop, and a sort() function in the for-loop. Should be O(n\*mlog(m)), but m <=3 and it's small we assume it is a constant

std::vector<PlaceID> **places\_closest\_to**(Coord xy, PlaceType type);

// Estimate of performance: in average theta(1), O(n)

// Short rationale for estimate:because we only used unordered\_map's erase()

bool **remove\_place**(PlaceID id);

// Estimate of performance: O(n+m)

AreaID **common\_area\_of\_subareas**(AreaID id1, AreaID id2);

**//phase 2////////////////////////////**

// Phase 2 operations-----------------------------------------------------

// Estimate of performance: theta(n)

// Short rationale for estimate: for-loop

std::vector<WayID> **all\_ways**();

// Estimate of performance: O(n), on average case: theta(1)

// Short rationale for estimate: no for or whileloops, both .insert() and .find() are: O(n), on average case: theta(1).

bool **add\_way**(WayID id, std::vector<Coord> coords);

// Estimate of performance: theta(n)

// Short rationale for estimate: one for loop. Inside for loop's function is .push\_back, which takes O(1).

// O(n)=O(n\*1)

std::vector<std::pair<WayID, Coord>> **ways\_from**(Coord xy);

// Estimate of performance: O(n), on average case: theta(1)

// Short rationale for estimate: .find() is used

std::vector<Coord> **get\_way\_coords**(WayID id);

// Estimate of performance: theta(n)

// Short rationale for estimate: for loop

void **clear\_ways**();

// Estimate of performance: O(V+E)

// Short rationale for estimate: v describle while loop called vertices, E describle for loop called edges

// O(n) if coord is found from the last element. n>=V

std::vector<std::tuple<Coord, WayID, Distance>> **route\_any**(Coord fromxy, Coord toxy);

// Non-compulsory operations

// Estimate of performance: O(n), on average case: theta(1)

// Short rationale for estimate: .find(), .erase() are used

bool **remove\_way**(WayID id);

// Estimate of performance: O(V+E)

// Short rationale for estimate: v describle while loop called vertices, E describle for loop called edges

// O(n) if coord is found from the last element. n>=V

std::vector<std::tuple<Coord, WayID, Distance>> **route\_least\_crossroads**(Coord fromxy, Coord toxy);

// Estimate of performance: O(V+E)

// Short rationale for estimate: v describle while loop called vertices, E describle for loop called edges

// O(n) if id is found from the last element. n>=V

std::vector<std::tuple<Coord, WayID>> **route\_with\_cycle**(Coord fromxy);

// Estimate of performance: O(Vlog(V)+Elog(E)), or theta(V+E)

// Short rationale for estimate: v describle while loop called vertices, E describle for loop called edges

// priority\_queue is used O(nlog(n)), theta(n), there are .pop() used in while loop, and .push() used in for loop std::vector<std::tuple<Coord, WayID, Distance>> **route\_shortest\_distance**(Coord fromxy, Coord toxy);

// Estimate of performance:

// Short rationale for estimate:

Distance **trim\_ways**();

Private:

// Estimate of performance: theta(n)

// Short rationale for estimate: for loop

void **clearCoorDataMarks**();

// Estimate of performance: theta(n)

// Short rationale for estimate: for loop

Distance **calWayDist**(const WayID id);

// Estimate of performance: O(n)

// Short rationale for estimate: path is a line

void **printPath**(vector<std::tuple<Coord, WayID, Distance> >& path, const CoordData coordDataFrom, const CoordData coordDataTo);

// Estimate of performance: O(n)

// Short rationale for estimate: path is a line

void **printPath**(vector<std::tuple<Coord, WayID> >& path, const CoordData coordDataFrom, const CoordData coordDataTo);