**Dokumentaatio**

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

H292042

**Tietorakenteen valinta:**

//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.

**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))

// 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^2), but perftest's result is theta(1)

// Short rationale for estimate: there are one while-loop inside anoter while-loop. it depends what kind of data we have. Im worst case first one's and second one's while-loop size will be n-1, if data's size is n.

// but our "return" will interupt it, so it will never be (n-1)^2 (except they do not have the common area)

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