Dokumentaatio

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
```

Private:

```
    // Helping "all_subareas_in_area". Pre-order-tree-walk through subarea
and add them to "vector<AreaID>:
    // Depth First Traversal is O(n + m), where n is the number of nodes,
and m is the number of edges.
    void PRE_WALK_SUB(Area* recentArea, vector<AreaID>& ID);

    //Helping "places_closest_to"and "sort3element" compare a's and b's
distance from coord xy
    //return true id a's distance from xy is shorter than b
    //Coord's "<" was reloaded(from "datastructure.hh" in rows 64-71)
```

```
    //O(1)
    bool aShorterB(const PlaceID& a,const PlaceID& b,const Coord& xy);

    //Helping "places_closest_to", sorting 3(or less than 3) elements in
the "vector<PlaceID>"
    //Coord's "<" was reloaded(from "datastructure.hh" in rows 64-71)
    //O(nlog(n)). But actually it is theta(1) because n is constant 0-3. So
this one will not effect "places_closest_to"'s complexity
        void sort3element(vector<PlaceID>& placeVec,const Coord& xy);
```