Performance document:

int place_count();

Estimate of performance: O(1)
std::size() has constant time complexity with unordered_map.

void clear_all();

Estimate of performance: O(n)
std::clear() time complexity is linear in the size of the container. Used twice to clear both places_ and areas_.

std::vector<PlaceID> all_places();

Estimate of performance: O(n)
Map keys are inserted to vector inside a for-loop, so time complexity is linear in the size of the container.

bool add_place(PlaceID id, Name const& name, PlaceType type, Coord xy);

Estimate of performance: Average for unordered_map O(1), worst case O(n)
Both std:.find() and std::insert() have the above time complexity with unordered_map

std::pair<Name, PlaceType> get_place_name_type(PlaceID id);

Estimate of performance: Average for unordered_map O(1), worst case O(n)
std:.find() has the above time complexity with unordered_map

Coord get_place_coord(PlaceID id);

Estimate of performance: Average for unordered_map O(1), worst case O(n)
std:.find() has the above time complexity with unordered_map

std::vector<PlaceID> places_alphabetically();

Estimate of performance: O(n log(n))
Map items are inserted into a multimap inside a for-loop. This operation has the time complexity O(n log(n)) where n is the size of the container. There is also a O(n) operation when ids are inserted into a vector inside a for-loop.

std::vector<PlaceID> places_coord_order();

Estimate of performance: O(n log(n))
Places_ data is assigned to a struct. The structs are stored in a vector. This vector is sorted with std::sort where time complexity is O(n log(n)) (n is distance between first and last member). Ids are then inserted in another vector inside a for-loop with time complexity O(n).

std::vector<PlaceID> find_places_name(Name const& name);

Estimate of performance: O(n)
Ids are inserted in a vector inside a for-loop with time complexity O(n) where n is the size of the container (unordered_map in this case).

std::vector<PlaceID> find_places_type(PlaceType type);

Estimate of performance: O(n)
Ids are inserted in a vector inside a for-loop with time complexity O(n) where n is the size of the container (unordered_map in this case).

bool change_place_name(PlaceID id, Name const& newname);

Estimate of performance: Average for unordered_map O(1), worst case O(n)
std:.find() has the above time complexity with unordered_map

bool change_place_coord(PlaceID id, Coord newcoord);

Estimate of performance: Average for unordered_map O(1), worst case O(n)
std:.find() has the above time complexity with unordered_map

bool add_area(AreaID id, Name const& name, std::vector<Coord> coords);

Estimate of performance: Average for unordered_map O(1), worst case O(n)
Both std:.find() and std::insert() have the above time complexity with unordered_map

Name get_area_name(AreaID id);

Estimate of performance: Average for unordered_map O(1), worst case O(n)
std:.find() has the above time complexity with unordered_map

std::vector<Coord> get_area_coords(AreaID id);

Estimate of performance: Average for unordered_map O(1), worst case O(n)
std:.find() has the above time complexity with unordered_map

std::vector<AreaID> all_areas();

Estimate of performance: O(n)
Depends on the amount of keys (n) in map

bool add_subarea_to_area(AreaID id, AreaID parentid);

Estimate of performance: Average for unordered_map O(1), worst case O(n)
std:.find() has the above time complexity with unordered_map. Inserting subareas to vector takes constant time O(1).

std::vector<AreaID> subarea_in_areas(AreaID id);

Estimate of performance: O(n)
std::find() on average has time complexity O(1). vector::clear() time complexity is O(n). Uses a recursive helper function check_parentareas(AreaID id); which stores the ids of parent areas in a vector. The time complexity for this operation is O(n).

std::vector<AreaID> all_subareas_in_area(AreaID id);

Estimate of performance: O(n)
std::find() on average has time complexity O(1). vector::clear() time complexity is O(n). Uses a recursive helper function check_subareas(AreaID id); which stores the ids of sub areas in a vector. The time complexity for this operation is O(n).

std::vector<PlaceID> places_closest_to(Coord xy, PlaceType type);

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

Stores structs that hold coordinate data in a vector. This happens inside a for-loop so time complexity is O(n). This vector is sorted with std::sort where time complexity is O(n log(n)) (n is distance between first and last member). Finally the ids are inserted in another vector. This happens inside a for-loop so time complexity is O(n).

bool remove_place(PlaceID id);

Estimate of performance: Average for unordered_map O(1), worst case O(n)

std:.find() has the above time complexity with unordered_map. std::erase() time complexity is also O(1) on average and O(n) in worst case with unordered_map.

AreaID common_area_of_subareas(AreaID id1, AreaID id2);

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

std::find() on average has time complexity O(1) and worst case O(n). Function uses other function subarea_in_areas(). The time complexity for this function is O(n). To check if vectors are empty empty() is used the time complexity is constant. The vectors are sorted using std::sort where time complexity is O(n log(n)) (n is distance between first and last member). The two iterators are compared with each other inside a while loop. The time complexity for this is O(n) where n is the size of the container.