**CS 211**
**Fundamentals of Computer Programming II**
**Winter 2024**

# Project #06 (v1.1)

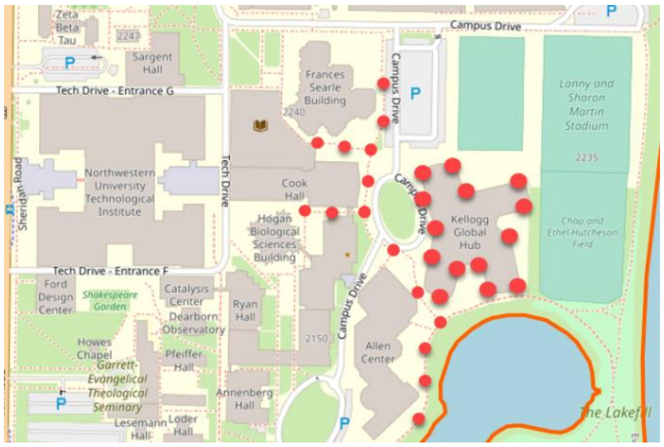
**Assignment:** Open Street Maps --- refactoring, footways
**Submission:** Gradescope (4 submissions per 24-hr period)
**Policy:** individual work only, late work is accepted
**Complete By:** Friday February 23rd @ 11:59pm CST
Late submissions: see syllabus for late policy… No submissions accepted after Sunday 02/25 @ 11:59pm

**Pre-requisites:** HW #06 (working with map data structure)

## Overview

In project 06 we're going to continue working with Open Street Maps. The goal is two-fold: (1) refactor your existing solution to make it more efficient / easier to understand, and (2) input footways (pathways) on Northwestern's Evanston campus. Once project 06 is completed, we'll have the foundation in place to navigate around campus.

Recall we are working with maps from https://www.openstreetmap.org/. Browse to the site and type "Northwestern University" into the search field, and then click on the first search result. You'll see the Evanston campus highlighted. Notice the "export" button --- we used this button to download the "nu.osm" map file we are working with.



We are focused on two features of the map: "Nodes" and "Ways". Recall that a **node** is a point on the map, consisting of 3 values: id, latitude, and longitude. Nodes are shown as red dots in the screenshot to the right. A **way** is a series of nodes that define something, such as the **buildings** we input in project 05. Here in project 06 we're going to input **footways**, which are walkways or sidewalks around campus --- these are shown as dashed lines in the screenshot. More details of Open Street Map are available on Wikipedia.

## Getting Started

We will continue working on the EECS computers. The first step is to setup your project06 folder and copy the necessary files. Here are the steps for copying the files on the EECS computers (also available in dropbox):

| | |
|---|---|
| 1. Connect and open a terminal window | |
| 2. Make a directory for project 06 | `mkdir project06` |
| 3. Make this directory private | `chmod 700 project06` |
| 4. Move ("change") into this directory | `cd project06` |
| 5. Copy the files needed for project 06: | |
| a. To use your solution: | `cp -r ../project05/release .` |
| b. To use our solution[1]: | `cp -r /home/cs211/w2024/project06/release .` |
| 6. Copy another, larger OSM file | `cp /home/cs211/w2024/project06/ns.osm .` |
| 7. List the contents of the directory | `ls` |
| 8. Move ("change") into release dir | `cd release` |

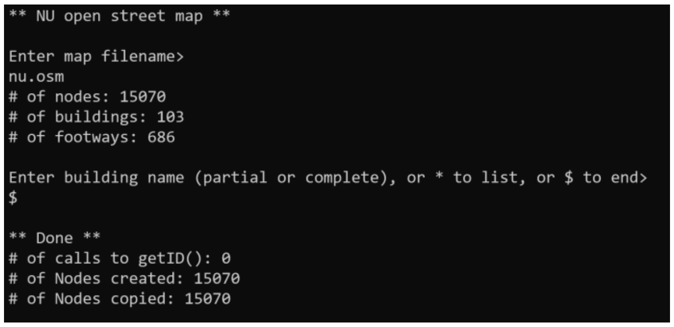
You are encouraged to use your project05 solution here in Project 06, especially if you wrote helper functions. However, one thing to consider is that Gradescope is going to revert back to comparing your program output to our program output, which implies you will have to modify your program to produce the exact output shown in the screenshots. Alternatively, you can copy our "main.cpp" solution file so your output will more closely match our output when you submit to Gradescope:

| | |
|---|---|
| 9. Copy just our "main.cpp" file[1] | `cp /home/cs211/w2024/project06/release/main.cpp .` |

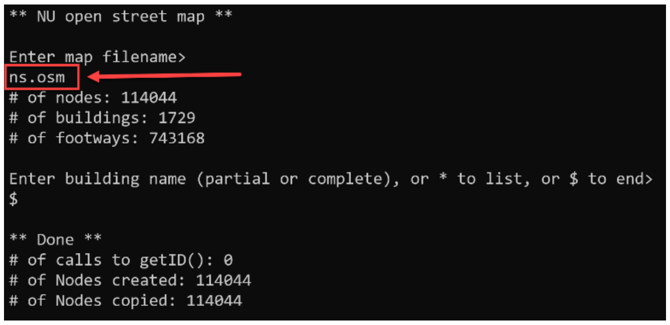
Now do the following to make sure all is well:

| | |
|---|---|
| 10. List the contents of the directory | `ls` |
| 11. Build the program | `make build` |
| 12. Run the program | `./a.out` |

The program should run successfully, and represent a solution to project 05:



```
** NU open street map **

Enter map filename>
nu.osm
# of nodes: 15070
# of buildings: 103
# of footways: 686

Enter building name (partial or complete), or * to list, or $ to end>
$

** Done **
# of calls to getID(): 0
# of Nodes created: 15070
# of Nodes copied: 15070
```



```
** NU open street map **

Enter map filename>
ns.osm
# of nodes: 114044
# of buildings: 1729
# of footways: 743168

Enter building name (partial or complete), or * to list, or $ to end>
$

** Done **
# of calls to getID(): 0
# of Nodes created: 114044
# of Nodes copied: 114044
```

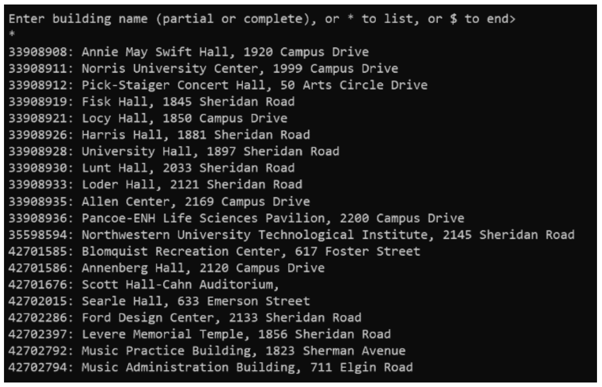
[1] Note that our solution will not be available until Monday Feb 19th @ 12:01am.

## Part 01: Refactoring

**Refactoring** is the process of taking an existing program and reworking it to improve efficiency, readability, understanding, and maintainability. We're going to refactor your solution to project 05 to demonstrate the process. To begin, open VS Code, establish a remote connection to moore (or another EECS computer), and File >> Open your **project06/release** folder --- your program files should now be visible in the file explorer window pane. Open a terminal window in VS code, cd to the proper directory **project06/release**, and confirm that you can build and run the program: `make build` and `./a.out`

### Step 01: helper function Buildings::print( )

For main( ), we are going to adopt a new design rule: no nested loops[2]. Instead, any nested loop must be turned into a function. For example, look at your main( ) function and how you currently handle the * command, which is supposed to print all the buildings, one per line. How did you implement this functionality? Most likely you did the following:



```
Enter building name (partial or complete), or * to list, or $ to end>
*
33908908: Annie May Swift Hall, 1920 Campus Drive
33908911: Norris University Center, 1999 Campus Drive
33908912: Pick-Staiger Concert Hall, 50 Arts Circle Drive
33908919: Fisk Hall, 1845 Sheridan Road
33908921: Locy Hall, 1850 Campus Drive
33908926: Harris Hall, 1881 Sheridan Road
33908928: University Hall, 1897 Sheridan Road
33908930: Lunt Hall, 2033 Sheridan Road
33908933: Loder Hall, 2121 Sheridan Road
33908935: Allen Center, 2169 Campus Drive
33908936: Pancoe-ENH Life Sciences Pavilion, 2200 Campus Drive
35598594: Northwestern University Technological Institute, 2145 Sheridan Road
42701585: Blomquist Recreation Center, 617 Foster Street
42701586: Annenberg Hall, 2120 Campus Drive
42701676: Scott Hall-Cahn Auditorium,
42702015: Searle Hall, 633 Emerson Street
42702286: Ford Design Center, 2133 Sheridan Road
42702397: Levere Memorial Temple, 1856 Sheridan Road
42702792: Music Practice Building, 1823 Sherman Avenue
42702794: Music Administration Building, 711 Elgin Road
```

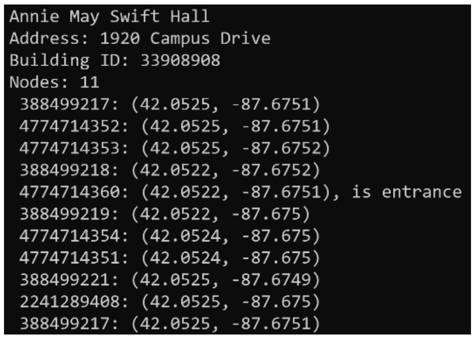
```
while (true)
{
  .
  .
  .
  else if (name == "*") {
    for (const Building& B : buildings.MapBuildings) {
      cout << B.ID << ": " << B.Name << ", " << B.StreetAddress << endl;
    }
  }
```

Given the while loop, the nested for loop is no longer allowed --- why not? To encourage you to write helper functions. It's not about the length of a function, it's about the abstraction. We need to print the buildings, so that code should be separated into a function that's associated with the data. Since we are printing all the buildings, this **print( )** function should be part of the **Buildings** class. Define this function in the Buildings class as follows:

1. Modify "buildings.h" to declare a void print( ) const function; add a header comment. You'll need to #include <iostream> if you haven't already. Save and close.
2. Modify "buildings.cpp" to implement the function void Buildings::print( ) const; copy header comment.
3. Move the print code --- the loop and anything else you need --- over to Buildings::print( ). Save, close.
4. Modify main( ) to now call the print function in your Buildings object.
5. Run and test --- program should behave exactly as before.

[2] This rule does not apply to the existing *read( )* functions in the Nodes and Buildings classes; likewise it will not apply to the read( ) function you'll eventually add to a new Footways class.

## Step 02:  helper function Building::print( )



```
Annie May Swift Hall
Address: 1920 Campus Drive
Building ID: 33908908
Nodes: 11
 388499217: (42.0525, -87.6751)
 4774714352: (42.0525, -87.6751)
 4774714353: (42.0525, -87.6752)
 388499218: (42.0522, -87.6752)
 4774714360: (42.0522, -87.6751), is entrance
 388499219: (42.0522, -87.675)
 4774714354: (42.0524, -87.675)
 4774714351: (42.0524, -87.675)
 388499221: (42.0525, -87.6749)
 2241289408: (42.0525, -87.675)
 388499217: (42.0525, -87.6751)
```

Given our new rule for the design of main( ) --- no nested loops --- we probably need to add more helper functions. When the user enter a building name, how do you search and output the required building(s)? Most likely you did the following, adding two loops inside main's while loop:

```
else {
  for (const Building& B : buildings.MapBuildings) {

     if (B.Name.find(name) != string::npos) {

        cout << B.Name << endl;
        cout << "Address: " << B.StreetAddress << endl;
        cout << "Building ID: " << B.ID << endl;

        cout << "Nodes: " << B.NodeIDs.size() << endl;
        for (long long nodeid : B.NodeIDs) {
           cout << " " << nodeid << ": ";

           double lat = 0.0;
           double lon = 0.0;
           bool entrance = false;

           bool found = nodes.find(nodeid, lat, lon, entrance);
           .
           .
           .
        }
     }
  }
}
```

The code above should start to bother you… There's 4 levels of { } nesting, and that's not counting the surrounding loop and main( ). And I'm pretty sure that if we hadn't provided the **nodes.find( )** search function --- be honest! --- you probably would have written another nested loop. This habit we need you to break…

Let's refactor the above code in two steps. First, let's focus on the inner-most code that prints a building: id, name, street address, and nodes. This is yet another **print( )** function, this time in the **Building** class. Much like you just did with the Buildings class, repeat the same exercise with the Building class. This print( ) function is slightly more complicated in that it needs to be passed the nodes for searching purposes. Be sure to pass by reference to avoid copying the 15K nodes each time print( ) is called:

1. Modify "building.h" to declare a void print(const Nodes& nodes) const function; add a header comment. For "building.h" to compile, you'll need to #include <iostream>, "node.h" and "nodes.h" at the top of "building.h".
2. Modify "building.cpp" to implement the function void Building::print(const Nodes& nodes) const and copy the header comment.
3. Move the print code from main( ) over to Building::print( ).

4. Modify main( ) to now call the print function in your Building object (see the code snippet below).
5. Run and test --- program should behave exactly as before.

When you are done, main( ) should now look like this:

```
else {
  for (const Building& B : buildings.MapBuildings) {

     if (B.Name.find(name) != string::npos) {
        B.print(nodes);
     }
  }
}
```

This is a much better design. We're not done yet, continue with the next section to finish the refactoring of main( )...

## Step 03: helper function Buildings::findAndPrint( )

Complete the refactoring of main( ) by moving the nested loop that searches and prints over to the **Buildings** class. Call this new function **findAndPrint( )**. You know the process, go ahead and modify "buildings.h" and "buildings.cpp" appropriately. Think about the parameters you need, and what header files you need to #include. When you're done, main( ) should look like this:

```
while (true)
{
  .
  .
  .
  else if (name == "*") {
    buildings.print();
  }
  else {
    buildings.findAndPrint(name, nodes);
  }
```

Run and test --- the program should behave exactly as before. This is the refactoring process, and should become a natural part of your design thought process moving forward.

## Step 04: refactoring Nodes class for efficiency

The **Nodes** class performs an important function: the efficient lookup of a map node to obtain it's (latitude, longitude) position. Efficient lookup is critial in any navigation app that likely contains millions of map nodes.

In our current solution, efficient lookup is obtained by using a vector + binary search within the **Node::find( )** function. This works fine, but as discussed in class we know that vectors have two disadvantages:

1. *There is a cost to doubling the vector when it becomes full, which is an O(N) operation*
2. *If we need to insert new nodes into the vector, insertion is an O(N) operation.*

In C++, the **map** container gives us the efficiency of binary search while avoiding the disadvantages of vectors.

Let's refactor the Nodes class by replacing vector with map. Note that because we followed the principle of data hiding in the design of the Nodes class --- i.e. by making the vector private --- we will be able to refactor the class without requiring any changes to the any C++ code outside the class. This is another example of good design. Steps:

1. Open "nodes.h" and change the #include at the top from <vector> to <map>. Now change the data member from vector<Node> to **map<long long, Node>**. In other words, we are going to store **Node** objects in the map, and we'll search for them based on their id which is of type **long long**. Save and close.
2. Open "nodes.cpp", and look at the #includes at the top --- delete <vector> if it's present, and then add #include <utility>. In the readMapNodes( ) function, delete the code that calls emplace_back to store into the vector. Instead we need to store each new Node object into our map. The most conicse way is using map's **[ ]** operator, mapping node N to its id:

   ```
   Node N(id, latitude, longitude, entrance);
   this->MapNodes[id] = N;
   ```

   However, the above fails to compile because "Node class does not have an appropriate default constructor". We could add a default constructor to the Node class, but that allows the creation of empty objects, which is a bad design choice. A better alternative is to use map's **insert** function, which creates a new node and then inserts it into the map as a (key, value) pair.

   ```
   Node N(id, latitude, longitude, entrance);
   this->MapNodes.insert( make_pair(id, N) );
   ```

   However, the above makes twice as many copies because we have to create the node first, and then (1) N is copied in the call to insert and (2) map copies the object into the container. A more efficient approach is to use map's **emplace** function, which looks similar but makes just one copy (a copy is made in calling the function, but then that copy is moved into the container thereby avoiding the 2nd copy):

   ```
   this->MapNodes.emplace(id, Node(id, latitude, longitude, entrance));
   ```

3. In Nodes::find( ), comment out or delete the code for linear search and binary search.
4. Now rewrite Nodes::find( ) function to lookup the Node efficiently in the map. The most concise way is to use map's [ ] operator to retrieve the node by its id:

   ```
   Node N = this->MapNodes[id];
   ```

   The problem is the above approach will create an empty Node if the id is not found --- there's no way to know if the search by id was successful or not. The correct approach is to use map's **find( )** function to search the map by id. The map find( ) function returns an ***iterator*** (think pointer) to a (key, value) pair if found, where in this case **ptr->first** would be the node id and **ptr->second** would be the Node object. Here's the code, your job is to fill in the { } for the found and not found cases:

```
auto ptr = this->MapNodes.find(id);

if (ptr == this->MapNodes.end()) { // not found:

   ???
}
else { // found:

   ???
}
```

5. Save your work, run and test --- the program should behave exactly as before.

Stop and reflect for a second… We just made a non-trivial change to the program by changing from one data structure to another. However, all changes to the program were contained within "nodes.h" and "nodes.cpp", and no other part of the program had to change --- this is the power of data hiding and classes / helper functions (in this case nodes.find( )). When you can make non-trivial changes without those changes rippling through a program, you are following good software engineering practices.

## Part 02: Footways

In the previous project we input Nodes and Buildings from the map file. Here in project 06 we need to input the **footways** --- i.e. walkways for pedestrian traffic. A footway is a "way" in the OSM file, with a (key, value) pair of ("highway", "footway") or ("addr:highway", "footway"):

```
<way id="986532630" visible="true" version="1" changeset="111627186"
 <nd ref="9119071425"/>
 <nd ref="533996671"/>
 <nd ref="533996672"/>
 <nd ref="2240260064"/>
 <tag k="highway" v="footway"/>
</way>
```

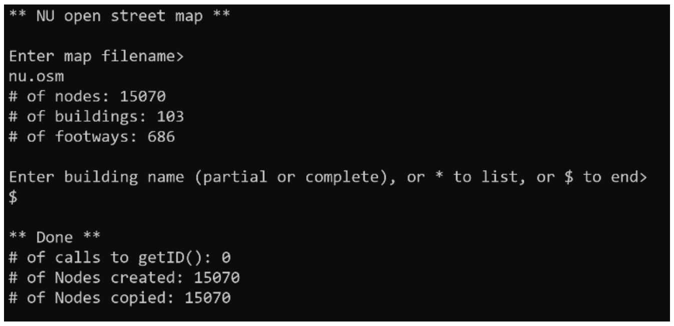
Here are the steps --- in less detail than usual --- for inputting and storing the footways:

1. Review your Building class since you're going to define a Footway class…
2. Create a **Footway** class for storing information about one footway (much like the Building class stores information about one building). A footway consists of an ID (long long), and a vector of NodeIDs (long long). Create both "footway.h" and "footway.cpp" files, define a constructor, etc.
3. Review your Buildings class because you're going to define a Footways class…

4. Create a **Footways** class that works similarly to Buildings. Create both “footways.h” and “footways.cpp” files. Your Footways class should define a vector of Footway objects called **MapFootways**. Add a method to input the footways called **readMapFootways( )**, along with a method to return the # of footways called **getNumMapFootways( )**. Note that a footway may contain either of the following (key, value) pairs (“highway”, “footway”) or (“area:highway”, “footway”):

```
if (osmContainsKeyValue(way, "highway", "footway") ||
    osmContainsKeyValue(way, "area:highway", "footway"))
```

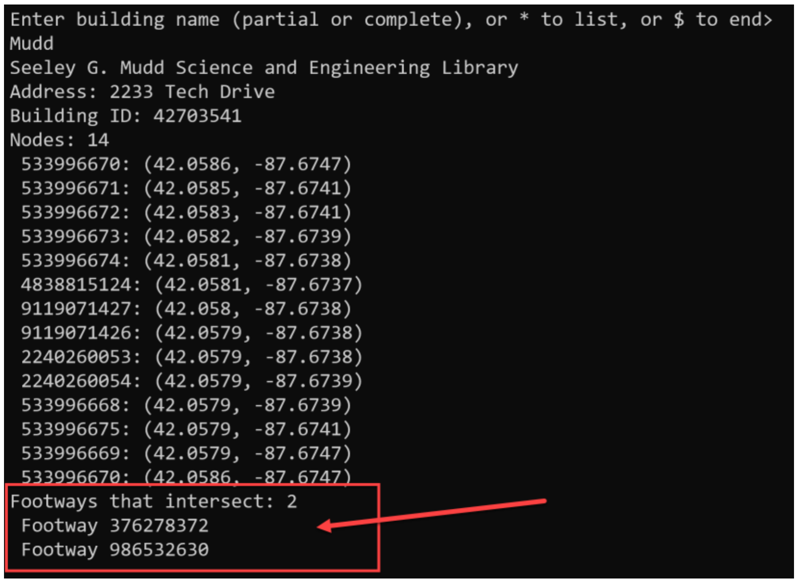
5. Once the classes are defined, modify “makefile” on lines 3 and 10 to include “footway.cpp” and “footways.cpp” in the list of files compiled with g++
6. Update “main.cpp” to input the footways, and then output the # of footways along with the # of node and buildings (see screenshot below).
7. Build and run. If all is well, you should have 686 footways:



```
** NU open street map **

Enter map filename>
nu.osm
# of nodes: 15070
# of buildings: 103
# of footways: 686

Enter building name (partial or complete), or * to list, or $ to end>
$

** Done **
# of calls to getID(): 0
# of Nodes created: 15070
# of Nodes copied: 15070
```

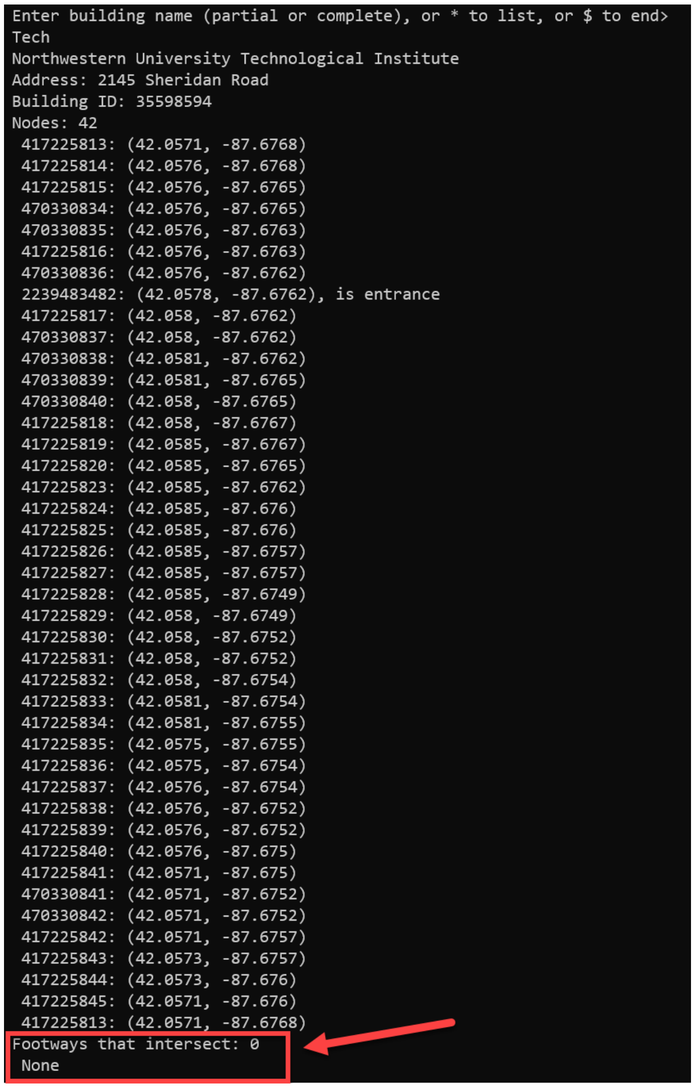
## Part 03: Footways that intersect with Buildings

A footway F **intersects** with a building B if they have a node in common --- i.e. F’s vector of node ids has an id in common with B’s vector of node ids. Modify your program so that when the user searches for a building, you also output any footways that intersect with that building. For example, Mudd has two footways that intersect:

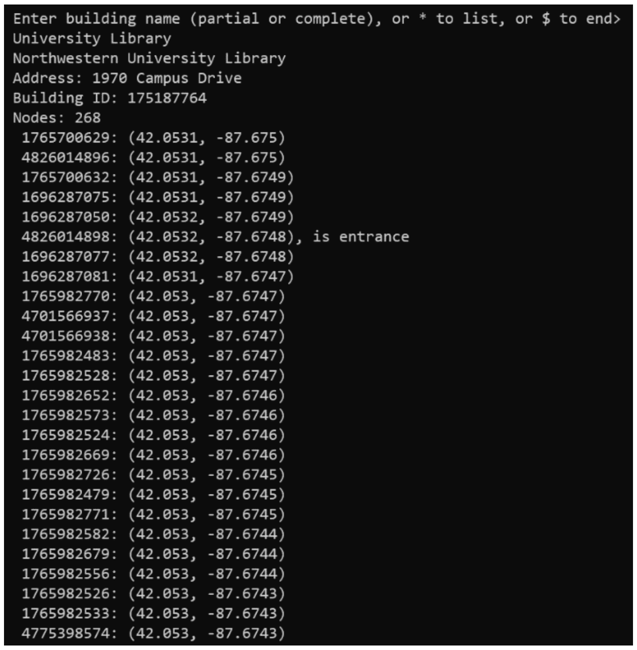


```
Enter building name (partial or complete), or * to list, or $ to end>
Mudd
Seeley G. Mudd Science and Engineering Library
Address: 2233 Tech Drive
Building ID: 42703541
Nodes: 14
 533996670: (42.0586, -87.6747)
 533996671: (42.0585, -87.6741)
 533996672: (42.0583, -87.6741)
 533996673: (42.0582, -87.6739)
 533996674: (42.0581, -87.6738)
 4838815124: (42.0581, -87.6737)
 9119071427: (42.058, -87.6738)
 9119071426: (42.0579, -87.6738)
 2240260053: (42.0579, -87.6738)
 2240260054: (42.0579, -87.6739)
 533996668: (42.0579, -87.6739)
 533996675: (42.0579, -87.6741)
 533996669: (42.0579, -87.6747)
 533996670: (42.0586, -87.6747)
Footways that intersect: 2
 Footway 376278372
 Footway 986532630
```

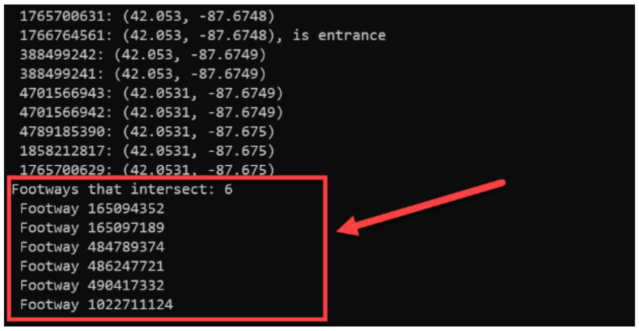
Some buildings have no footways that intersect such as Tech (on the left below), while the NU Library has six footways that intersect (on the right below):



```
Enter building name (partial or complete), or * to list, or $ to end>
Tech
Northwestern University Technological Institute
Address: 2145 Sheridan Road
Building ID: 35598594
Nodes: 42
 417225813: (42.0571, -87.6768)
 417225814: (42.0576, -87.6768)
 417225815: (42.0576, -87.6765)
 470330834: (42.0576, -87.6765)
 470330835: (42.0576, -87.6763)
 417225816: (42.0576, -87.6763)
 470330836: (42.0576, -87.6762)
 2239483482: (42.0578, -87.6762), is entrance
 417225817: (42.058, -87.6762)
 470330837: (42.058, -87.6762)
 470330838: (42.0581, -87.6762)
 470330839: (42.0581, -87.6765)
 470330840: (42.058, -87.6765)
 417225818: (42.058, -87.6767)
 417225819: (42.0585, -87.6767)
 417225820: (42.0585, -87.6765)
 417225823: (42.0585, -87.6762)
 417225824: (42.0585, -87.676)
 417225825: (42.0585, -87.676)
 417225826: (42.0585, -87.6757)
 417225827: (42.0585, -87.6757)
 417225828: (42.0585, -87.6749)
 417225829: (42.058, -87.6749)
 417225830: (42.058, -87.6752)
 417225831: (42.058, -87.6752)
 417225832: (42.058, -87.6754)
 417225833: (42.0581, -87.6754)
 417225834: (42.0581, -87.6755)
 417225835: (42.0575, -87.6755)
 417225836: (42.0575, -87.6754)
 417225837: (42.0576, -87.6754)
 417225838: (42.0576, -87.6752)
 417225839: (42.0576, -87.6752)
 417225840: (42.0576, -87.675)
 417225841: (42.0571, -87.675)
 470330841: (42.0571, -87.6752)
 470330842: (42.0571, -87.6752)
 417225842: (42.0571, -87.6757)
 417225843: (42.0573, -87.6757)
 417225844: (42.0573, -87.676)
 417225845: (42.0571, -87.676)
 417225813: (42.0571, -87.6768)
Footways that intersect: 0
 None
```



```
Enter building name (partial or complete), or * to list, or $ to end>
University Library
Northwestern University Library
Address: 1970 Campus Drive
Building ID: 175187764
Nodes: 268
 1765700629: (42.0531, -87.675)
 4826014896: (42.0531, -87.675)
 1765700632: (42.0531, -87.6749)
 1696287075: (42.0531, -87.6749)
 1696287050: (42.0532, -87.6749)
 4826014898: (42.0532, -87.6748), is entrance
 1696287077: (42.0532, -87.6748)
 1696287081: (42.0531, -87.6747)
 1765982770: (42.053, -87.6747)
 4701566937: (42.053, -87.6747)
 4701566938: (42.053, -87.6747)
 1765982483: (42.053, -87.6747)
 1765982528: (42.053, -87.6747)
 1765982652: (42.053, -87.6746)
 1765982573: (42.053, -87.6746)
 1765982524: (42.053, -87.6746)
 1765982669: (42.053, -87.6746)
 1765982726: (42.053, -87.6745)
 1765982479: (42.053, -87.6745)
 1765982771: (42.053, -87.6745)
 1765982582: (42.053, -87.6744)
 1765982679: (42.053, -87.6744)
 1765982556: (42.053, -87.6744)
 1765982526: (42.053, -87.6743)
 1765982533: (42.053, -87.6743)
 4775398574: (42.053, -87.6743)
```

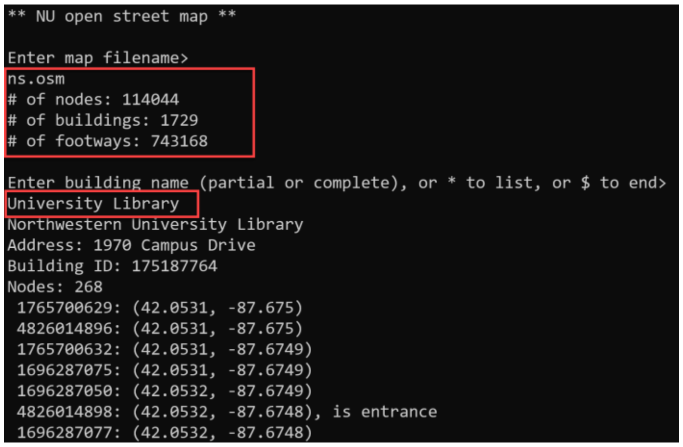


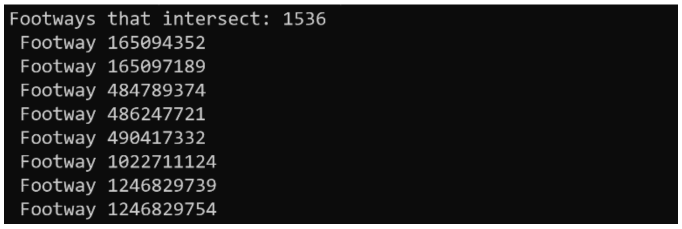
```
 1765700631: (42.053, -87.6748)
 1766764561: (42.053, -87.6748), is entrance
 388499242: (42.053, -87.6749)
 388499241: (42.053, -87.6749)
 4701566943: (42.0531, -87.6749)
 4701566942: (42.0531, -87.6749)
 4789185390: (42.0531, -87.675)
 1858212817: (42.0531, -87.675)
 1765700629: (42.0531, -87.675)
Footways that intersect: 6
 Footway 165094352
 Footway 165097189
 Footway 484789374
 Footway 486247721
 Footway 490417332
 Footway 1022711124
```

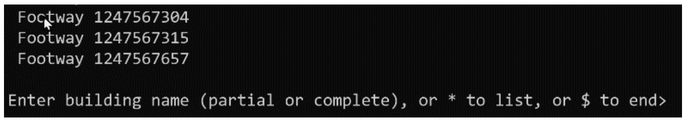
Note that the footway ids are output in sorted order. This implies you need to collect the footway ids into a container first, sort, and then output; you are free to use the built-in sort function.

**<u>Requirement #1</u>**: the implementation of this feature must follow the "no nested loops" rule. When you were refactoring earlier, each nested loop turned into a function. You are required to follow the same approach here for implementing "footways that intersect". How you do this is up to you: modify existing functions, add additional functions, change from vector to other containers, or a combination of these approaches. Note that you cannot reduce the # of classes, but you can add additional classes if that makes sense. <u>Hint</u>: don't jump immediately to the first idea that comes to mind, think about what you are trying to do and what would lead to the most efficient or easiest to understand approach.

**<u>Requirement #2:</u>** you need an "efficient" solution. When you have a working solution, test your program by loading the larger "ns.osm" file --- this may take 10-20 seconds to input before you see the first prompt (this amount of time is fine). Then lookup “University Library”. How long does your solution take to output the intersecting footways (there are 1,536)? It should take no more than a few seconds on the EECS computers; if it takes longer, you’ll need to design a more efficient solution.


```
** NU open street map **

Enter map filename>
ns.osm
# of nodes: 114044
# of buildings: 1729
# of footways: 743168

Enter building name (partial or complete), or * to list, or $ to end>
University Library
Northwestern University Library
Address: 1970 Campus Drive
Building ID: 175187764
Nodes: 268
 1765700629: (42.0531, -87.675)
 4826014896: (42.0531, -87.675)
 1765700632: (42.0531, -87.6749)
 1696287075: (42.0531, -87.6749)
 1696287050: (42.0532, -87.6749)
 4826014898: (42.0532, -87.6748), is entrance
 1696287077: (42.0532, -87.6748)
```


```
Footways that intersect: 1536
 Footway 165094352
 Footway 165097189
 Footway 484789374
 Footway 486247721
 Footway 490417332
 Footway 1022711124
 Footway 1246829739
 Footway 1246829754
```


```
 Footway 1247567304
 Footway 1247567315
 Footway 1247567657

Enter building name (partial or complete), or * to list, or $ to end>
```

## Grading and Electronic Submission

You will submit all your .cpp and .h files to Gradescope for evaluation. To submit from the EECS computers, run the following command (you must run this command from inside your **project06/release** directory):

```
/home/cs211/w2024/tools/project06  submit  *.cpp  *.h
```

The “Project06” submission will open two days before the due date. You will have a limit on the total # of submissions: **4 submissions per 24-hour period**. A 24-hour period starts at midnight, and after 4 submissions, you cannot submit again until the following midnight; unused submissions do not carry over, you get exactly 4 per 24-hour period (including late days).

By default, Gradescope records the score of your **LAST SUBMISSION** (not your highest submission score). If you want us to grade a different submission, activate a different submission via your “Submission History”. This must be done before the due date.

The autograding portion of the project will be worth 80/100 points. The remaining 20 points will be determined manually by confirming the following requirements and reviewing your overall approach. However, you may lose correctness points if you have a feature correctly implemented, but manual review reveals that the proper steps were not taken (for example, not refactoring, or not adhering to the rule of “no nested loops”):

1. *Proper refactoring as described in part 01*
2. *Implementation of Footways class are described in part 02*
3. *Implementation of footways intersecting with buildings as described in part 03*
4. *Proper adherence to “no nested loops” when implementing part 03*
5. *No functions > 100 lines of code*

6. *Header comments at top of each .h and .c file, and above each function*
7. *Meaningful comments describing blocks of code (e.g. loops)*
8. *No memory errors AND no memory leaks. You can run valgrind on the EECS computers as we did on replit:* ***`valgrind ./a.out`***

Omission of item 1, or 2, or 3, or 4 will result in an overall score of 0 for the assignment; you must meet the requirements implied by items 1 – 4 above.

## Academic Conduct Policy

Northwestern publishes a basic guide to academic integrity, which can be found here. In summary, here are NU's eight cardinal rules of academic integrity:

1. *Know your rights*
2. *Acknowledge your sources*
3. *Protect your work*
4. *Avoid suspicion*
5. *Do you own work*
6. *Never falsify a record or permit another person to do so*
7. *Never fabricate data, citations, or experimental results*
8. *Always tell the truth when discussing your work with your instructor*

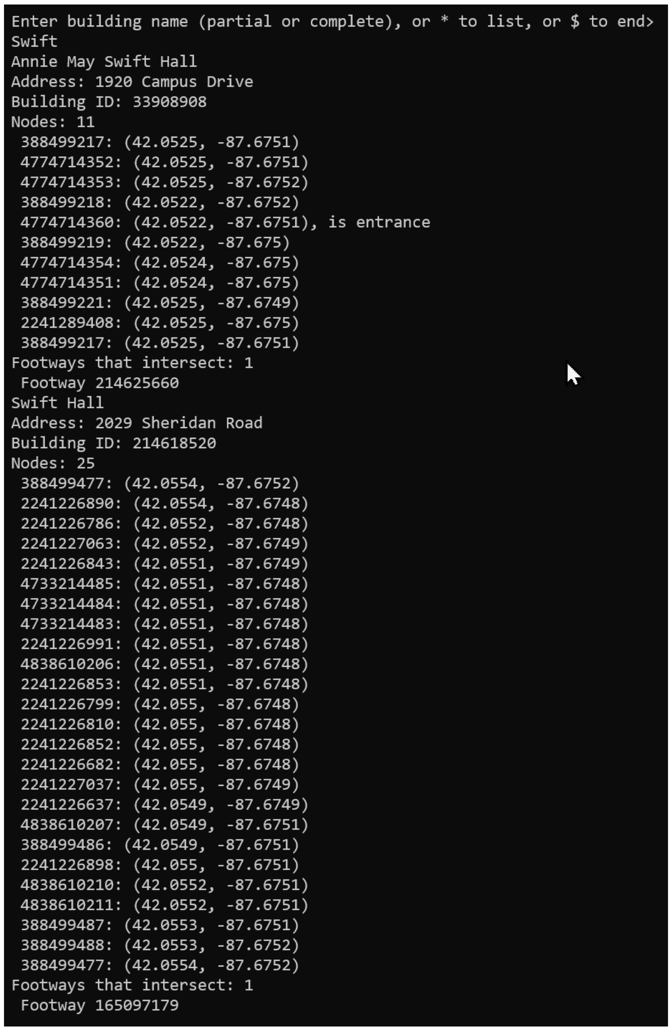
School policies and more information can be found on NU's academic integrity website. With regards to CS 211, unless stated otherwise, all work submitted for grading *must* be done individually. While we encourage you to talk and learn from the course staff, peers, and others, this interaction must be superficial with regards to all work submitted for grading. This means you cannot work in teams, you cannot work side-by-side, you cannot submit someone else's work (partial or complete) as your own. The use of AI (ChatGPT, Co-pilot, etc.) is currently forbidden.

Examples of what is not allowed? Downloading work from a github repository and submitting it as your own, whether partial or complete. Downloading answers from StackOverflow and submitting them as your own. Emailing your work to another student, or receiving work from another student. Sharing your screen with another student so they can see your work (and potentially submit it as their own). Participating in a screen share and taking screenshots or photos of someone else's work, and then using that as a guide to submit your own work. Copying answers posted to Piazza, making a few simple changes, and then submitting as your own work. Allowing someone else to write / type the answer for you. Using AI (ChatGPT, Co-pilot, etc.) to generate code for you which you then submit as your own.

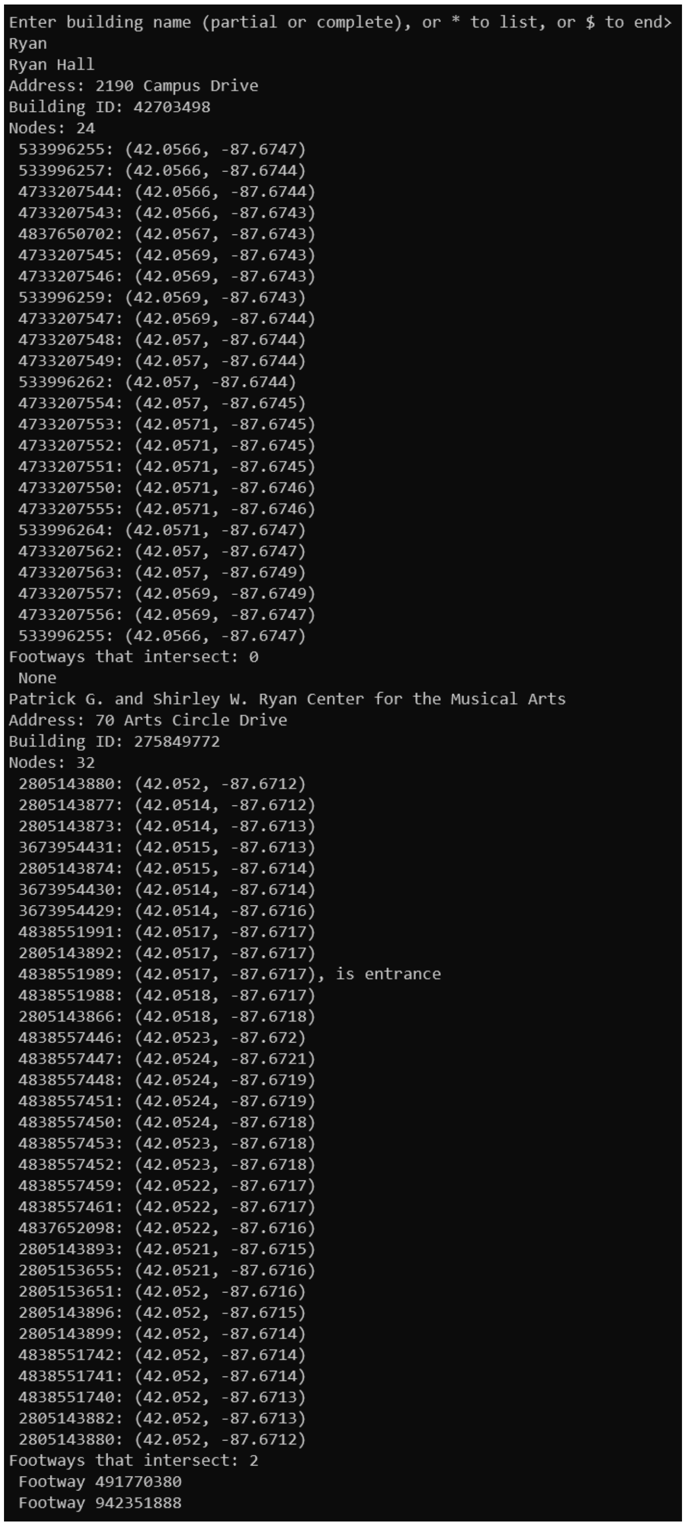
Okay, so what is allowed? Talking to the instructor or course staff, and getting insights --- but not direct answers --- to help you solve the assignment. Talking to other students about the assignment, using diagrams / natural language / pseudo-code to convey ideas. Searching the internet for guidance, and using that guidance to help you form your own solution. If you do receive help / guidance, it's always best to cite your source by name or URL just in case there is a question as to where the work came from.

## Appendix A

Using **nu.osm**, outputs for “Swift” and “Ryan”.



```
Enter building name (partial or complete), or * to list, or $ to end>
Swift
Annie May Swift Hall
Address: 1920 Campus Drive
Building ID: 33908908
Nodes: 11
 388499217: (42.0525, -87.6751)
 4774714352: (42.0525, -87.6751)
 4774714353: (42.0525, -87.6752)
 388499218: (42.0522, -87.6752)
 4774714360: (42.0522, -87.6751), is entrance
 388499219: (42.0522, -87.675)
 4774714354: (42.0524, -87.675)
 4774714351: (42.0524, -87.675)
 388499221: (42.0525, -87.6749)
 2241289408: (42.0525, -87.675)
 388499217: (42.0525, -87.6751)
Footways that intersect: 1
 Footway 214625660
Swift Hall
Address: 2029 Sheridan Road
Building ID: 214618520
Nodes: 25
 388499477: (42.0554, -87.6752)
 2241226890: (42.0554, -87.6748)
 2241226786: (42.0552, -87.6748)
 2241227063: (42.0552, -87.6749)
 2241226843: (42.0551, -87.6749)
 4733214485: (42.0551, -87.6748)
 4733214484: (42.0551, -87.6748)
 4733214483: (42.0551, -87.6748)
 2241226991: (42.0551, -87.6748)
 4838610206: (42.0551, -87.6748)
 2241226853: (42.0551, -87.6748)
 2241226799: (42.055, -87.6748)
 2241226810: (42.055, -87.6748)
 2241226852: (42.055, -87.6748)
 2241226682: (42.055, -87.6748)
 2241227037: (42.055, -87.6749)
 2241226637: (42.0549, -87.6749)
 4838610207: (42.0549, -87.6751)
 388499486: (42.0549, -87.6751)
 2241226898: (42.055, -87.6751)
 4838610210: (42.0552, -87.6751)
 4838610211: (42.0552, -87.6751)
 388499487: (42.0553, -87.6751)
 388499488: (42.0553, -87.6752)
 388499477: (42.0554, -87.6752)
Footways that intersect: 1
 Footway 165097179
```



```
Enter building name (partial or complete), or * to list, or $ to end>
Ryan
Ryan Hall
Address: 2190 Campus Drive
Building ID: 42703498
Nodes: 24
 533996255: (42.0566, -87.6747)
 533996257: (42.0566, -87.6744)
 4733207544: (42.0566, -87.6744)
 4733207543: (42.0566, -87.6743)
 4837650702: (42.0567, -87.6743)
 4733207545: (42.0569, -87.6743)
 4733207546: (42.0569, -87.6743)
 533996259: (42.0569, -87.6743)
 4733207547: (42.0569, -87.6744)
 4733207548: (42.057, -87.6744)
 4733207549: (42.057, -87.6744)
 533996262: (42.057, -87.6744)
 4733207554: (42.057, -87.6745)
 4733207553: (42.0571, -87.6745)
 4733207552: (42.0571, -87.6745)
 4733207551: (42.0571, -87.6745)
 4733207550: (42.0571, -87.6746)
 4733207555: (42.0571, -87.6746)
 533996264: (42.0571, -87.6747)
 4733207562: (42.057, -87.6747)
 4733207563: (42.057, -87.6749)
 4733207557: (42.0569, -87.6749)
 4733207556: (42.0569, -87.6747)
 533996255: (42.0566, -87.6747)
Footways that intersect: 0
 None
Patrick G. and Shirley W. Ryan Center for the Musical Arts
Address: 70 Arts Circle Drive
Building ID: 275849772
Nodes: 32
 2805143880: (42.052, -87.6712)
 2805143877: (42.0514, -87.6712)
 2805143873: (42.0514, -87.6713)
 3673954431: (42.0515, -87.6713)
 2805143874: (42.0515, -87.6714)
 3673954430: (42.0514, -87.6714)
 3673954429: (42.0514, -87.6716)
 4838551991: (42.0517, -87.6717)
 2805143892: (42.0517, -87.6717)
 4838551989: (42.0517, -87.6717), is entrance
 4838551988: (42.0518, -87.6717)
 2805143866: (42.0518, -87.6718)
 4838557446: (42.0523, -87.672)
 4838557447: (42.0524, -87.6721)
 4838557448: (42.0524, -87.6719)
 4838557451: (42.0524, -87.6719)
 4838557450: (42.0524, -87.6718)
 4838557453: (42.0523, -87.6718)
 4838557452: (42.0523, -87.6718)
 4838557459: (42.0522, -87.6717)
 4838557461: (42.0522, -87.6717)
 4837652098: (42.0522, -87.6716)
 2805143893: (42.0521, -87.6715)
 2805153655: (42.0521, -87.6716)
 2805153651: (42.052, -87.6716)
 2805143896: (42.052, -87.6715)
 2805143899: (42.052, -87.6714)
 4838551742: (42.052, -87.6714)
 4838551741: (42.052, -87.6714)
 4838551740: (42.052, -87.6713)
 2805143882: (42.052, -87.6713)
 2805143880: (42.052, -87.6712)
Footways that intersect: 2
 Footway 491770380
 Footway 942351888
```