CS 211
Fundamentals of Computer Programming II
Winter 2023

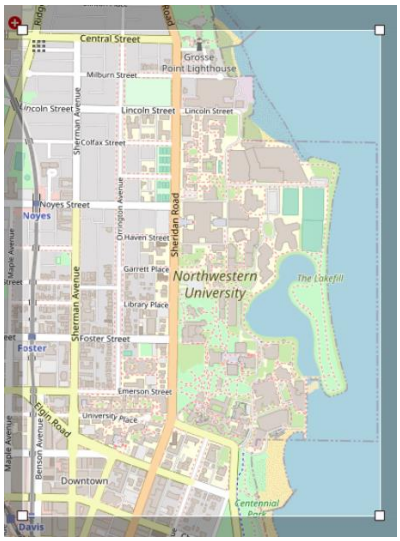
Project #05 (v1.0)

Assignment: Open Street Map processing in C++
Submission: Gradescope (4 submissions per 24-hr period)
Policy: individual work only, late work is accepted
Complete By: Friday February 16th @ 11:59pm CST
Late submissions: see syllabus for late policy… No submissions accepted after Sunday 02/18 @ 11:59pm

Pre-requisites: Access to the EECS computers

Overview

In project 05 we are starting our tour of C++, and developing a modern C++ program to perform basic navigation of an Open Street Map. We'll focus on the area around Northwestern's Evanston campus, and build an app to pull data from the map and provide building and bus information. Project 05 represents the first step in building this program: reading a map file and laying the foundation for future projects.

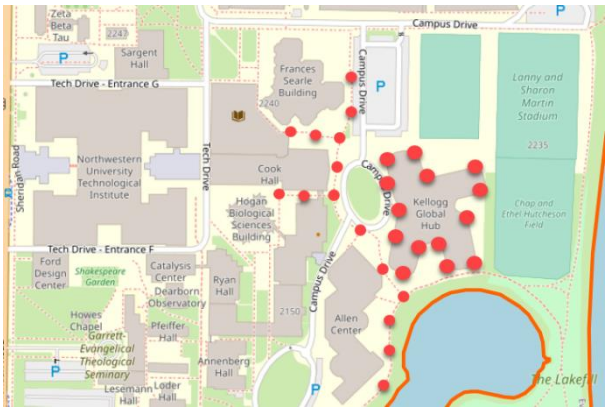


Everyone is familiar with navigation apps. While we don't have the ability in C++ to display the results graphically, what we can do is write the back-end operations to load the map, build the necessary data structures, and provide useful information. While we're going to focus on just the NU Evanston campus, when we're done you'll have the foundation in place to process any valid Open Street Map.

Background

We are working with open-source 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 map file "nu.osm" that we'll be working with (we did a manual export to reduce the size somewhere). An OSM map file is an XML document, and we'll use TinyXML to help us parse the map file / XML document.

Zoom in. We're going to focus on two features of a map: "Nodes" and "Ways". A **node** is a point on the map, consisting of 3 values: id, latitude, and longitude. These are shown as red dots (there are thousands more). A **way** is a series of nodes that define something. The two most important examples in our case are **buildings** and **footways**. In the screenshot to the right, the buildings are labeled and the footways are the dashed lines. For a building, the nodes define the building's perimeter (e.g. around "Kellog Global Hub"). For a footway, the nodes define the endpoints of the footway as well as intermediate points along the way. More details of Open Street Map are available on Wikipedia.



An openstreetmap is represented as an XML document. Very briefly, an XML document is a text-based representation of a tree, with a concept of "parent" and "children". An openstreetmap starts with an <osm> node, and contains <node>, <way>, and other child **elements** nested within <osm> … </osm>:

```
<?xml version="1.0" encoding="UTF-8"?>
<osm version="0.6" … >
   <node id="25779197" lat="41.8737233" lon="-87.6456365" … />
   .
   .
   .
   <way id="32815712" … >
     <nd ref="1645121457"/>
     .
     .
     .
     <nd ref="462010732"/>
     <tag k="foot" v="yes"/>
     <tag k="highway" v="footway"/>
   </way>
   .
   .
   .
   <way id="151960667" … >
     <nd ref="1647971990"/>
     <nd ref="1647971996"/>
     .
     .
     .
     <nd ref="1647971990"/>
     <tag k="name" v="Science &amp;amp; Engineering Offices (SEO)"/>
   </way>
   .
   .
   .
</osm>
```

XML is a tag-based format, much like HTML; in fact, HTML is a special case of XML. An *element* is defined by an opening tag such as *<node … />*. For example, here's the definition of a **node** element:

```
<node id="25779197" lat="41.8737233" lon="-87.6456365" … />
```

This element contains **attributes** that associate data with this element, e.g. an **id** attribute "25779197" which uniquely identifies this node, and a position defined by the **latitude** attribute "41.8737233" and **longitude** attribute "-87.6456365".

An *element* can also be defined by matching opening and closing tags, such as the following **way** element which starts with *<way …>* and ends with *</way>*. This approach is used in XML when the element has sub-list(s) of associated data, for example:

```
<way id="32815712" … >
  <nd ref="1645121457"/>
  <nd ref="462010732"/>
  <tag k="foot" v="yes"/>
  <tag k="highway" v="footway"/>
</way>
```

The above **way** element has an id attribute "32815712", which uniquely identifiers this element. It also has two **nd** elements ("node definitions") which refer to existing node elements. It also has two **tag** elements --- "foot" and "highway" --- which associate additional information with this **way**.

EECS Computers

This project (and future projects) require the use of the EECS Linux computers. If you are not already using the EECS computers (or had issues earlier in the quarter), you need to get setup on the EECS computers and solve any configuration problems. The instructions for working with the EECS computers were given in the project 01 handout, starting on page 5. [*We are not providing the files on replit, but if you want to work on your own flavor of Linux, you can download the project files from dropbox.*]

Note to everyone: you may need to update VS Code to default to C++ 2017 (or newer) so you get proper suggestions / code completion. How to configure VS Code is discussed in this stackoverflow post.

Copying the provided files

Login to the remote computer **moore.wot.eecs.northwestern.edu**. Copy over the provided files to your own account as follows:

1. Make a directory for project 05 — **`mkdir project05`**
2. Make this directory private — **`chmod 700 project05`**
3. Move ("change") into this directory — **`cd project05`**
4. Copy the provided files --- the **.** is needed — **`cp -r /home/cs211/w2024/project05/release .`**
5. List the contents of the directory — **`ls`**

As this point you should see the directory "release" shown. Now

6. Move ("change") into release dir **`cd release`**
7. List the contents of the directory **`ls`**

You should see the following:

```
hummel@moore$ ls
building.cpp  makefile  node.h     nodes.h  osm.cpp  tinyxml2.cpp
building.h    node.cpp  nodes.cpp  nu.osm   osm.h    tinyxml2.h
hummel@moore$
```

There are 12 provided files, here's a summary:

Header files: building.h, node.h, nodes.h, osm.h, tinyxml2.h
C++ src files: building.cpp, node.cpp, nodes.cpp, osm.cpp, tinyxml2.cpp
Input files: nu.osm
Make file: makefile

No main( ) program is provided, so the first step will be to write your main( ) function to get started. Details in the next section…

Assignment details

The assignment in project 05 is to parse an Open Street Map file for the NU Evanston campus, and extract two sets of features:

1. Nodes (i.e. positions)
2. Buildings

The provided code performs step 1. Your assignment is to complete step 2, and then rewrite some of our provided code to make searching more efficient. Before you start, it helps to get a sense of how the Open Street Map is formatted. Open the "nu.osm" file in your editor and scroll through (it's a large file with roughly 60,000 lines). The file starts with **Node** elements (in order by their node id), which define (lattitude, longitude) positions on a map (see map at top of page 2):

```
<?xml version="1.0" encoding="UTF-8"?>
<osm version="0.6" generator="CGImap 0.8.8 (309515 spike-07.openstreetmap.org)" …>
 <bounds minlat="42.0478000" minlon="-87.6841000" maxlat="42.0644000" maxlon="-87.6677000"/>
 <node id="19793100" visible="true" … lat="42.0356152" lon="-87.6688741"/>
 <node id="19793102" visible="true" … lat="42.0378973" lon="-87.6694415"/>
 <node id="19793156" visible="true" … lat="42.0601544" lon="-87.6713798"/>
```

Later in the file (around line # 22,500) you see the start of **Way** elements --- highways, streets, buildings, footways, etc. For example:

```
<way id="33908908" visible="true" version="8" changeset="130477615" … >
```

```
    <nd ref="388499217"/>
    <nd ref="4774714352"/>
    <nd ref="4774714353"/>
    <nd ref="388499218"/>
    <nd ref="4774714360"/>
    <nd ref="388499219"/>
    <nd ref="4774714354"/>
    <nd ref="4774714351"/>
    <nd ref="388499221"/>
    <nd ref="2241289408"/>
    <nd ref="388499217"/>
    <tag k="addr:city" v="Evanston"/>
    <tag k="addr:housenumber" v="1920"/>
    <tag k="addr:postcode" v="60208"/>
    <tag k="addr:state" v="IL"/>
    <tag k="addr:street" v="Campus Drive"/>
    <tag k="addr:street:name" v="Campus"/>
    <tag k="addr:street:type" v="Drive"/>
    <tag k="building" v="university"/>
    <tag k="name" v="Annie May Swift Hall"/>
    <tag k="wikidata" v="Q7060402"/>
    <tag k="wikipedia" v="en:Northwestern University School of Communication"/>
  </way>
```

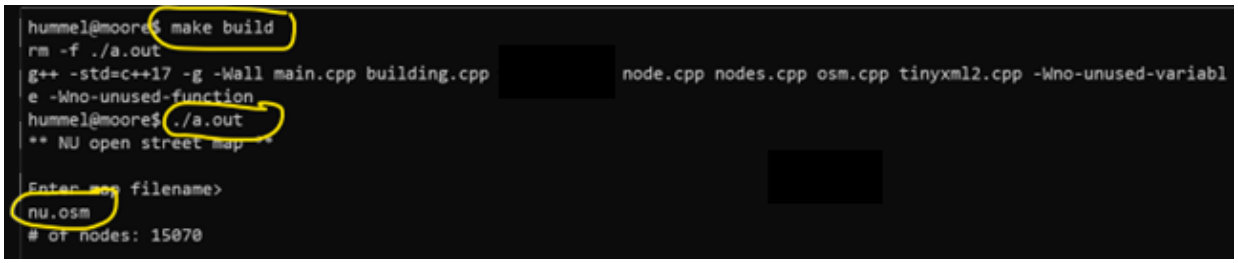
Notice that a "way" consists of 2 or more **nd** elements --- "node definition". These "nd" elements define the perimeter of this element, in this case the building "Annie May Swift Hall". Each "nd" element contains a **ref** attribute that references a Node that appeared earlier in the file; this referenced Node contains the necessary (lattitude, longitude) for this perimeter position. The provided code collects all the Node definitions. Your job here in Project 05 is to focus on the Way definitions (like you see above) and extract the buildings and their associated nd elements.

Step #1 is to write a main( ) function in "main.cpp" to prompt the user for the name of an OSM file, input the filename using the *getline( )* function, and then read the contents of this file into an XML document object using the function **osmLoadMapFile( )** defined in "osm.h". If unsuccessful, output some sort of error message and return 0.

If successful, create a nodes object and extract the node elements from the XML document by calling the **readMapNodes( )** method defined in "nodes.h". Output the # of nodes by calling **getNumMapNodes( )**, and then return 0. Your "main.cpp" file will need to #include each of the provided .h files: "building.h", "node.h", "nodes.h", "osm.h", and "tinyxml2.h". You'll also need to use the std C++ namespace, and the tinyxml2 namespace:

```
#include <iostream>
#include <string>

#include "building.h"
#include "node.h"
#include "nodes.h"
#include "osm.h"
#include "tinyxml2.h"

using namespace std;
using namespace tinyxml2;
```

Build your program using “make build”, and then run by typing “./a.out”. When prompted, enter the filename “nu.osm”. If your program is working, the # of nodes output should be 15,070. Here’s a screenshot:



Don’t worry about trying to match the output format exactly, Gradescope will be configured to accept a much wider range of output --- as long as the program inputs a filename and outputs the required values (the # of nodes). [*NOTE: use the getline( ) function to input the filename from the user, not the >> operator.*]

Step #2: at the end of main( ) --- just before the return --- output ** Done ** followed by 3 statistics: (1) the # of calls to node’s getID( ) method, (2) the # of nodes created, and (3) the # of nodes copied. You can obtain these values by calling the following methods of the *Node* class (these are special methods that do not require an object to call but instead just the name of the class):

```
cout << "# of calls to getID(): " << Node::getCallsToGetID() << endl;
cout << "# of Nodes created: " << Node::getCreated() << endl;
cout << "# of Nodes copied: " << Node::getCopied() << endl;
```

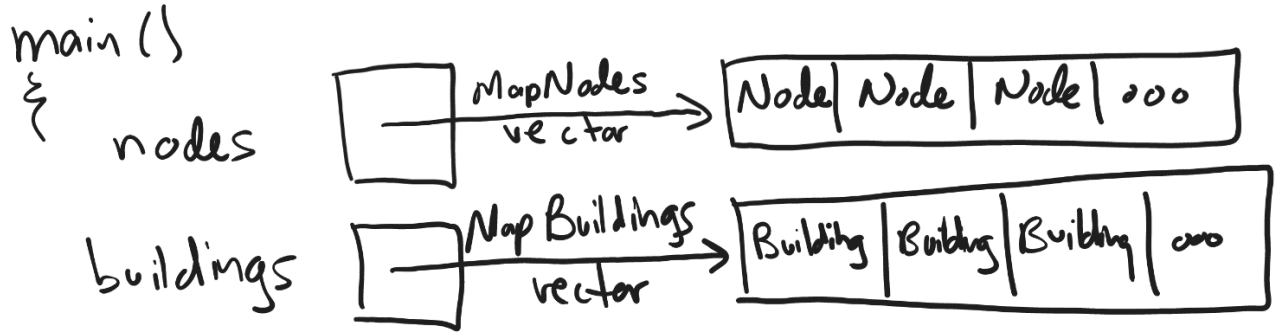
Here’s the output you should see at this point:

Now for step 3 and beyond:

3. Study the **Nodes** class, in particular the *readMapNodes( )* function and how it traverses the XML document node by node. The ID, Lat and Lon attributes are always present, but the “entrance” attribute is not necessarily present. Finally, notice how a **Node** is created and then added to the class’s vector **MapNodes**.

4. The result is a **nodes** object in main( ) that contains a vector of Node objects as shown on the next page. In the next step you’ll define and build a similar **buildings** object that contains a vector of

Building objects. Here's a diagram of the objects and underlying vectors:



5. Create a new file named "buildings.h". This class is similar to Nodes, and should look like this (you are free to extend this class with additional data members and methods, but this is a required starting point). For simplicity, define the vector *MapBuildings* as public:

```
/*buildings.h*/

//
// A collection of buildings in the Open Street Map.
//
// YOUR NAME
// Northwestern University
// CS 211
//
// Original author: Prof. Joe Hummel
//

#pragma once

#include <vector>

#include "building.h"
#include "tinyxml2.h"

using namespace std;
using namespace tinyxml2;


//
// Keeps track of all the buildings in the map.
//
class Buildings
{
public:
  vector<Building> MapBuildings;

  //
  // readMapBuildings
  //
  // Given an XML document, reads through the document and
```

```
  // stores all the buildings into the given vector.
  //
  void readMapBuildings(XMLDocument& xmldoc);

  //
  // accessors / getters
  //
  int getNumMapBuildings() const;
};
```

6. Create a new file named "buildings.cpp", and implement the two methods defined in "buildings.h". The **readMapBuildings( )** function should begin its traversal of the XML document from the first "way" element:

```
XMLElement* way = osm->FirstChildElement("way");
```

If the way element contains the (key, value) pair ("building", "university")

```
if (osmContainsKeyValue(way, "building", "university"))
```

then you have a building object that needs to be created and pushed into your vector. Grab the way's ID, and then grab the building's name and street address:

```
string name = osmGetKeyValue(way, "name");

string streetAddr = osmGetKeyValue(way, "addr:housenumber")
   + " "
   + osmGetKeyValue(way, "addr:street");
```

Now create a **Building** object **B**. Before we continue, recall the example earlier of a way that defined the building "Annie May Swift Hall":

```
<way id="33908908" visible="true" version="8" changeset="130477615" … >
  <nd ref="388499217"/>
  <nd ref="4774714352"/>
  .
  .
  .
  <tag k="building" v="university"/>
  <tag k="name" v="Annie May Swift Hall"/>
</way>
```

Your new building object **B** has a public vector<long long> named **NodeIDs** for storing the "nd" references defined inside the <way> element. Loop through the "nd" references as shown in the following code, adding each node reference attribute to B's *NodeIDs* vector:

```
XMLElement* nd = way->FirstChildElement("nd");

while (nd != nullptr)
{
  const XMLAttribute* ndref = nd->FindAttribute("ref");
  assert(ndref != nullptr);
```

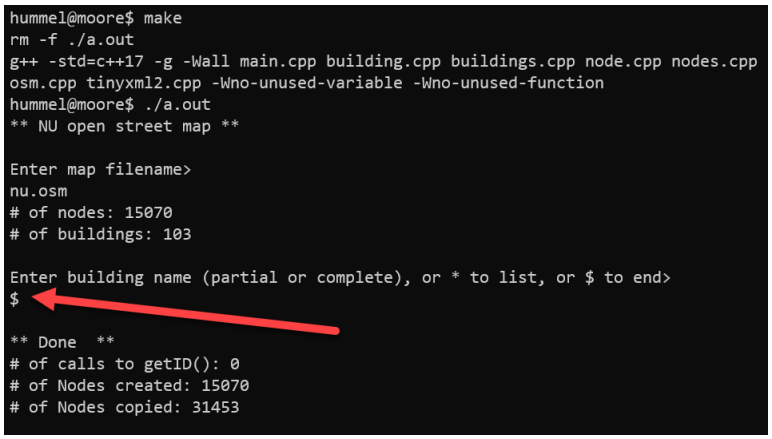
```
      long long id = ndref->Int64Value();

      B.add(id);

      // advance to next node ref:
      nd = nd->NextSiblingElement("nd");
    }
```

Once the building object B is fully initialized, add B to your Buildings vector; note that you must use the vector's **push_back** method (instead of *emplace_back*) because you *want* to make a copy of B --- you need to copy B's private vector of node ids that you just built.

7. Open the makefile in your editor. In the build section (line 3), add **buildings.cpp** to the list of C++ files that are being compiled under the "build" and "valgrind" commands. Build your program and fix any compilation errors.
8. Edit "main.cpp", #include "buildings.h", and in your main( ) function declare a Buildings object named **buildings**. Then call the **readMapBuildings()** method, and output the # of buildings by calling getNumMapBuildings() --- there should 103 buildings.
9. Build and run your program --- if all is well, you should have 15070 nodes and 103 buildings:

10. Well done! This is a good place to take a break…

11. Add an interactive loop in main( ) that prompts the user for a command, inputs using the getline( ) function[1], and then executes this command. This is repeated until the user enters "$", in which case the program should stop, output "** Done **" followed by the 3 stats mentioned earlier. Here's an example of how the program should behave when the user enters $ immediately:



12. If the user enters the "*" command, loop through the building object's vector and output each building: ID, name, and street address, one per line. Here are the first 20 buildings:

[1] Why are we using the getline( ) function for input, and not the >> operator? Ultimately the user may enter a building name such as "University Library". Using >> would input just the first word "University". The getline( ) function will input the entire string.

```
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
```

And here are the last 20 or so buildings:

```
151311271: Kresge Hall, 1880 Campus Drive
165094354: Jacobs Center,
175187764: Northwestern University Library, 1970 Campus Drive
182668630: Seabury-Western Theological Seminary, 600 Haven Street
182668876: Roycemore, 640 Lincoln Street
214618498: Crowe Hall, 1860 Campus Drive
214618504: Deering Library, 1937 Sheridan Road
214618520: Swift Hall, 2029 Sheridan Road
214630813: McCormick Foundation Center, 1870 Campus Drive
220670539: Central Utility Plant, 2026 Campus Drive
249758350: ,
249758351: Main, 2121 Sheridan Road
249758352: Lesemann Hall, 2121 Sheridan Road
249758354: Pfeiffer Hall,
275849772: Patrick G. and Shirley W. Ryan Center for the Musical Arts, 70 Arts Circle Drive
275851101: Segal Visitors Center/Parking Garage, 1841 Sheridan Road
275854464: Henry Crown Sports Pavilion and Aquatic Center, 2311 North Campus Drive
275854486: Regenstein Hall of Music, 60 Arts Circle Drive
466698283: Kellogg Global Hub, 2211 Campus Drive
768687038: Walter Athletics Center, 2255 Campus Drive
913835914: NU SafeRide, 630 Lincoln Street
913835915: Northwestern Career Advancement, 620 Lincoln Street
942642224: Canterbury House, 2010 Orrington Avenue
950303735: , 617 Noyes Street
950303736: , 629 Noyes Street
1002826076: , 2040 Sheridan Road
1002826078: , 2046 Sheridan Road
```

13. If the user enters a building name --- partial or complete --- then search through all the buildings and output each building whose name **contains** the user's input. Use the string class's **find( )** function to search the building's name. Note that this is not a Boolean function, but instead it returns a position of where the start of the search string was found; the function returns **string::npos** if NOT found.

    For example, if the user enters "Mudd", you should find one building with the following name,

address, and building ID. Recall that each building has a vector of node ids. For each node id, lookup the Node object's (lat, lon) and entrance data --- by calling **nodes.find( )**. If a Node is not found, output the id followed by "** NOT FOUND **". Here's the correct output for our Mudd building:

```
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:
  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)
```

14. If the user enters "Swift", there are two such buildings. Notice that "Annie May Swift Hall" actually has a node defined as an entrance (this is typically missing information):

```
Annie May Swift Hall
Address: 1920 Campus Drive
Building ID: 33908908
Nodes:
  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)
```

```
Swift Hall
Address: 2029 Sheridan Road
Building ID: 214618520
Nodes:
  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)
```

15. More examples are given in Appendix A…

16. Last feature… Run the program, and search for these 6 buildings: Mudd, Swift, Tech, Ryan, Kellogg, Annenberg. Then input "$" to end the program, and take a look at the statistics output -------->

```
** Done  **
# of calls to getID(): 1260839
# of Nodes created: 15070
# of Nodes copied: 1292292
```

17. How many nodes are being copied in your version of the program? Your number is probably over 1,000,000 as shown in the screenshot. Yikes. How can we reduce this number? The program is doing lots of searching to lookup building node ids… In particular, for each building, recall that you are calling **nodes.find( )** to lookup the id and get back the (lat, lon) and entrance information. Go take a look at the implementation of nodes.find( ) in the **Nodes** class. What simple change could you make to the linear search loop to reduce the # of copies? Make the change, build, and run --- how many copies do you have now? Wow.

18. Notice that **nodes.find( )** is using linear search --- searching through the nodes over and over again, front to back. There are 15K nodes, which implies that on average 7.5K nodes are searched before the desired node is found. Each building has 10-20 nodes that need to be looked up, so the search cost is 75K – 150K nodes per building! This is reflected in the "# of calls to getID( )" --- well over 1,000,000 nodes are visited in the search of just 6 buildings.

19. The solution? **Binary search**. Comment out the linear search code in the **find( )** function of the Nodes class, and replace with binary search. You can read about binary search in zyBooks section 12.2. Here's the code, it's a beautiful algorithm (watch a visualization here). Note that binary search requires that the search data is in sorted order --- which is true since the OSM files provide the node elements in sorted order by id attribute. Here's the code for binary search, which you'll learn more about in CS 214:

```
//
// binary search: jump in the middle, and if not found, search to
// the left if the element is smaller or to the right if bigger.
//
int low = 0;
int high = (int)this->MapNodes.size() - 1;

while (low <= high) {
  int mid = low + ((high - low) / 2);

  long long nodeid = this->MapNodes[mid].getID();

  if (id == nodeid) { // found!
    lat = this->MapNodes[mid].getLat();
    lon = this->MapNodes[mid].getLon();
    isEntrance = this->MapNodes[mid].getIsEntrance();

    return true;
  }
  else if (id < nodeid) { // search left:
    high = mid - 1;
  }
  else { // search right:
    low = mid + 1;
  }

}//while
```

20. Build, run, and search for those same 6 buildings. The number of calls to getID( ) should drop to under 3,000 --- a HUGE improvement. The power of binary search comes from the fact that each search divides

```
** Done  **
# of calls to getID(): 2960
# of Nodes created: 15070
# of Nodes copied: 16383
```

the search space in half, resulting in logN cost where N is the # of buildings. Given 15K buildings, each call to find( ) now visits only 16 or so node objects in the vector.

21. Notice the screenshot above also reduces the # of Nodes copied. What change can you make to **readMapNodes( )** in "nodes.cpp" to obtain this value?
22. That's it, well done!

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 **project05** directory):

```
/home/cs211/w2024/tools/project05  submit  *.cpp  *.h
```

The "Project05" submission will open two or more 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:

1. *Submission of reasonable main.cpp file, add helper functions to keep main( ) < 100 lines of code*
2. *Proper implementation of the required Buildings class*
3. *Updating of Nodes class, changing find( ) to use Binary Search*
4. *Header comments at top of each .h and .c file, and above each function*
5. *Meaningful comments describing blocks of code (e.g. loops)*
6. *No memory errors AND no memory leaks. You can run valgrind on the EECS computers using the makefile:* ***`make valgrind`***

Omission of item 1, or 2, or 3 will result in a score of 0 for the assignment. You must have a reasonable main.cpp file, you must implement a Buildings class, and you must update the Nodes class.

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

Outputs for “Swift”, “Tech”, “Ryan”, and “Annenberg”.

```
Annie May Swift Hall
Address: 1920 Campus Drive
Building ID: 33908908
Nodes:
  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)
```

```
Swift Hall
Address: 2029 Sheridan Road
Building ID: 214618520
Nodes:
  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)
```

```
Tech
Northwestern University Technological Institute
Address: 2145 Sheridan Road
Building ID: 35598594
Nodes:
  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)
```

```
Ryan
Ryan Hall
Address: 2190 Campus Drive
Building ID: 42703498
Nodes:
  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)
```

```
Patrick G. and Shirley W. Ryan Center for the Musical Arts
Address: 70 Arts Circle Drive
Building ID: 275849772
Nodes:
  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)
```

```
Annenberg
Annenberg Hall
Address: 2120 Campus Drive
Building ID: 42701586
Nodes:
  533980392: (42.0563, -87.6747)
  4838825513: (42.0561, -87.6747), is entrance
  533980393: (42.0559, -87.6747)
  2240259873: (42.0559, -87.6746)
  2240259868: (42.0559, -87.6746)
  2240259869: (42.0559, -87.6745)
  533980396: (42.0559, -87.6743)
  533980397: (42.056, -87.6743)
  533980398: (42.056, -87.6743)
  4837650718: (42.056, -87.6743)
  4837654421: (42.056, -87.6743)
  4837650720: (42.0561, -87.6743)
  4837650719: (42.0561, -87.6743)
  533980399: (42.0562, -87.6743)
  533980400: (42.0562, -87.6743)
  533980401: (42.0563, -87.6743)
  4837650715: (42.0563, -87.6744)
  4837650717: (42.0563, -87.6744)
  4837650716: (42.0563, -87.6745)
  4837650714: (42.0563, -87.6745)
  533980402: (42.0563, -87.6746)
  533980403: (42.0563, -87.6746)
  4838825510: (42.0563, -87.6747)
  4838825511: (42.0563, -87.6747)
  533980392: (42.0563, -87.6747)
```