ReadMe:

The project is intended to construct a Delaunay Triangulation using the steps in the Bower-Watson Incremental Algorithm. Simplified, that is, begin with a triangulation, add a vertex, find the illegal triangles and then redraw. Redrawing require varying steps depending on the number of violations found.

Approach took:

Triangles are stored in a Red-Black Tree based on the triangle’s circle’s center x value. Looking back, this should have just done the leftmost vertex of the triangle (GetMinX). This was chosen originally so that triangles near each other would be near each other on the tree making it easy to find neighbors and nearest neighbor analysis later on if needed as traversing the whole tree would not be required.

The structure Triangle contains 3 vertices and a circle within it. This is the circle that is constructed from these 3 points. The major property of the Delaunay Triangulation is that there are no vertices within this triangle, this is why I found it reasonable to put a circle in a triangle despite sounding weird on paper.

The structure Circle contains an A, B, C, D, a radius and a vertex Center. A, B, C, D are used to calculate the circle’s radius and center.

Data from a test file is read into a linked list that acts, more or less, like a stack. Data is added to the front of the Linked List and when the user wants to add a vertex to the triangulation a vertex is removed from the top of the list and added to the triangulation.

When the program opens the first 3 points at the top of the list are automatically read in because a triangulation must begin with any triangle. From there data is added only on a vertex basis. Upon adding a vertex, the tree is searched to find all illegal triangles. When an illegal triangle is encountered, i.e. the vertex is within the circumference of a triangle’s circle the triangle is added to a list of Illegal Triangles, also a copy of the triangle’s vertices are added to a list but duplicate vertices in that list are discarded. Then the illegal triangles are found and deleted from the tree, I would have made this occur during the initial traversal of the tree but in a Red-Black Tree when an item is deleted other nodes move so this could have created weird things in traversal.

I will admit this program is not memory efficient. I compare possible triangles before adding so with every comparison of triangle minimum angle a triangle is created so many triangles are created for the sake of comparison throughout the life of the program. This can be remediated in the future by just comparing angles between sets vertices as less memory would need to allocated per comparison.

The user has the ability to read in their own data at their own risk, so long it is a space-delimited list of x, y data. Following the form of:

1 5

2 8

4 2

…

They can also manually input an x, y pair that can either be added to the stack or directly to the triangulation, again at their own risk. Or they can have an x, y pair be randomly generated again which can be added to the triangulation or the stack.

The user can print their triangulation using the 3 print methods of a tree and also using a poor grid representation that I tried to make using boxes to resemble vertices. In this function data from the tree is converted to an array and then sorted in descending order based on the y-value. If the next y value to be printed is more than 3 apart then two \n are printed for the next line, if hey are the same y value no \n’s are printed, and if there is less than 3 apart then just 1. The x spacing is based on where this x value is compared to the range of x values and gets 3 spaces per 10% of the range, this value is then multiplied by 1.5 at the end and that determines x spacing. Sometimes it’s a good representation sometimes not.

Existing Problems:

1. Doesn’t thoroughly check if a vertex is in the triangulation- I think this is causing a crash with the Red Blue Tree.
2. GetMiddle, GetFurthest, GetClosest- I believe are the root of many issues, if distances to a point in the triangle are the same the vertex returned can be the same, this causes a vertex to be made with the same point, which shocker, is not a triangle, therefore, its center is nan, nan and comparing this in the tree crashes the program.
3. For the life of me I could not figure out why my Linked List of triangles would not properly delete the triangle. The T key in the List Node is not a pointer so when deleting the list node, the triangle deconstructor should be called but upon doing so 5 frees would be missing (3 Vertices, 1 Circle, and the triangle itself). To remediate this issue I used an std::list for triangles, however vertices are stored in the LinkedList.
4. 4+ violations not implemented. If the new vertex violates 4+ triangles nothing will happen. The solutions for 1, 2, 3, 4+ violations were all slightly different and I could not figure out how to generalize it. Fixing this would fix nearly all of the errors in the code.