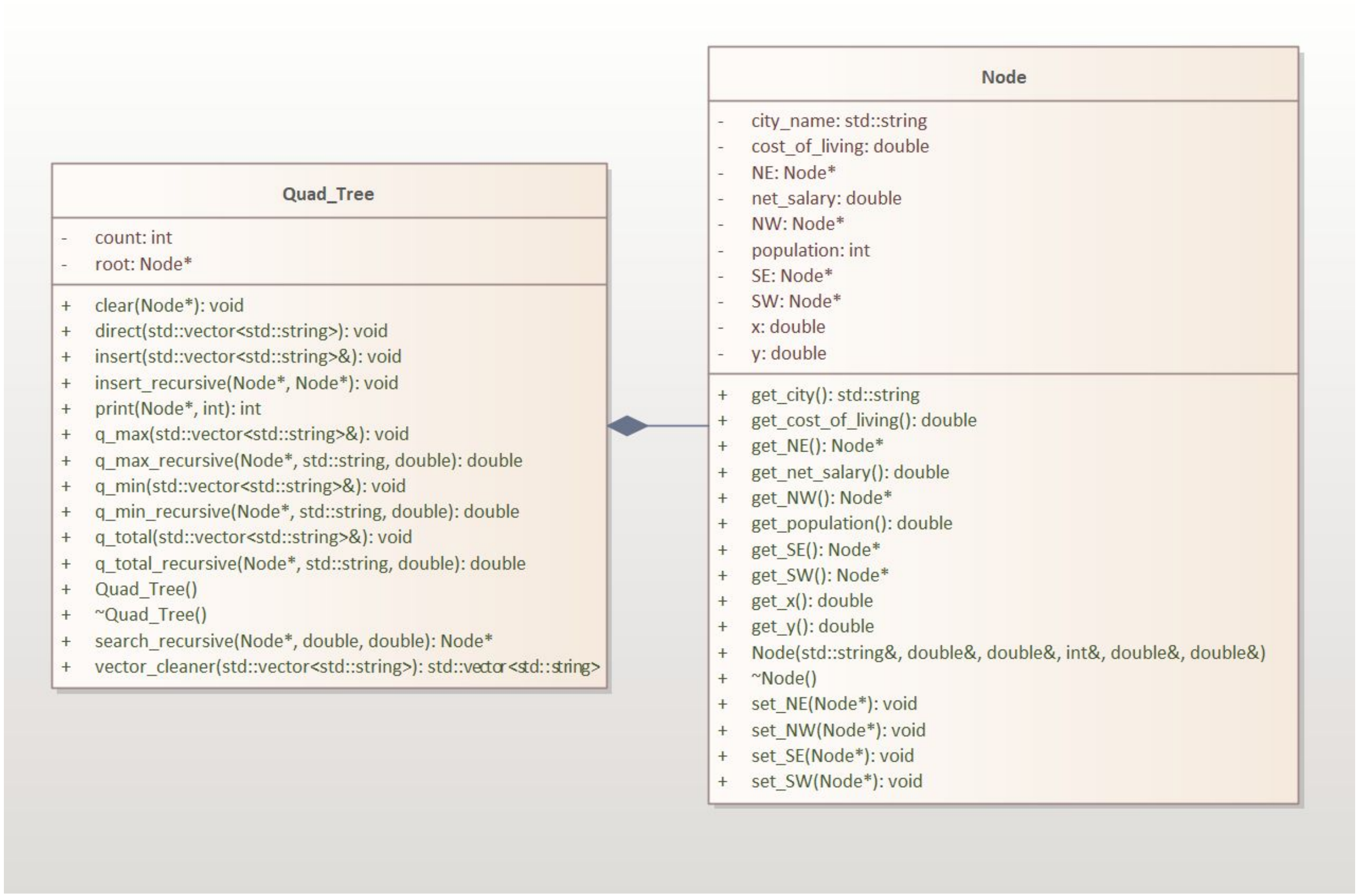
Eli Vlahos
ECE 250 Project 2
Tuesday, November 3rd, 2020

I designed two classes. The first was a QuadTree class. This class was responsible for all operations related to the QuadTree, including inserting, searching, finding the size, printing, clearing and the q functions. Some of these functions required a secondary function to complete it in a recursive manner. The QuadTree acted as the container for data as well as the means for completing the various functions. There was a Node class, which had every variable required for the node. The only functions for the node were get functions for private variables, as well as a set function for the four different children nodes (NW, NE, SW, SE).


Quad_Tree
- count: int
- root: Node*
+ clear(Node*): void
+ direct(std::vector<std::string>): void
+ insert(std::vector<std::string>&): void
+ insert_recursive(Node*, Node*): void
+ print(Node*, int): int
+ q_max(std::vector<std::string>&): void
+ q_max_recursive(Node*, std::string, double): double
+ q_min(std::vector<std::string>&): void
+ q_min_recursive(Node*, std::string, double): double
+ q_total(std::vector<std::string>&): void
+ q_total_recursive(Node*, std::string, double): double
+ Quad_Tree()
+ ~Quad_Tree()
+ search_recursive(Node*, double, double): Node*
+ vector_cleaner(std::vector<std::string>): std::vector<std::string>
Node
- city_name: std::string
- cost_of_living: double
- NE: Node*
- net_salary: double
- NW: Node*
- population: int
- SE: Node*
- SW: Node*
- x: double
- y: double
+ get_city(): std::string
+ get_cost_of_living(): double
+ get_NE(): Node*
+ get_net_salary(): double
+ get_NW(): Node*
+ get_population(): double
+ get_SE(): Node*
+ get_SW(): Node*
+ get_x(): double
+ get_y(): double
+ Node(std::string&, double&, double&, int&, double&, double&)
+ ~Node()
+ set_NE(Node*): void
+ set_NW(Node*): void
+ set_SE(Node*): void
+ set_SW(Node*): void

The constructor for the QuadTree created an empty tree, consisting of a null address and a size of 0. When adding a new node, it replaced or was linked to the original node. Initializing it immediately made the design of other functions easier. The constructor for the Node class functioned by initializing immediately all the values associated with the instance of the class. This was because all the information related to the instance was available at the moment of creation, with the exception of the Node children. They were initialized to the nullptr. The destructor function for the QuadTree involved calling the clear function. As it was already a function that could delete dynamically allocated nodes, there was no need for a unique destructor function. The clear function was a recursive function that called each four directions, assuming that the current node was not a nullptr. It created a temp variable equal to the current node that was deleted. The Node class did not need any dynamically allocated variables and therefore did not need a destructor function.

No operators were overridden, it was never necessary to use the same function with different parameters. When possible, a const pass by reference was done for parameter values. This was done when the parameter did not change throughout the function. This was to make the code run more quickly, as a copy of the parameter then did not need to be created. (This is pulled from my last project. The circumstances were the same and I saw no need to reword describing the same scenario).

My strategy for debugging the document was coming up with what was not covered by the test cases. The given tests covered the fundamentals of the QuadTree. The exceptions they covered were checking the size after a clear, and variations of using the print, clear, and q_ functions. Large tree sizes were handled in test three. The cases I came up with were two clears successively, trying the various functions on an empty tree (print, q_s, clear was already done) and inserting and clearing a tree with a size above 20. I was not concerned with memory leaks, I passed the checks with the test cases on the testing server and had no leaks. Besides clearing a complicated list, I was confident that nothing further was needed. For the rounding of cost of living and net salary, I stored rounded versions to avoid confusion. I then added decimal values to some of the insert values, and then checked to see if the information was displayed correctly in the output.

The time complexities are as expected. Assuming that the tree is balanced, the search and insert function should take $O(lg(n))$. This is because if the tree is balanced, this is the maximum height. Even for a balanced tree, the worst case for the q functions is $O(n)$. This is because if the coordinates given match the root node, all nodes underneath it or the entire tree will have to be searched. As a consequence, the run time will be linear. The best case for the q functions for if the tree is balanced is $O(lg(n))$, assuming that the node to be searched for is at the bottom. Print, clear, and the worst case of all functions should have a linear run time, that is $O(n)$. This is because print and clear require running the entire tree. If the tree was to have a depth of n, and the tree was not balanced, all functions would require a linear run time, that is $O(n)$. This is because the height of the tree would be n, that is the number of inserts.