DESIGN DOCUMENT: Project 3

NAME: Naveela Sookoo
STUDENT ID: nnsookoo

The assigned task was to create a C++ implementation for Quadtree that stores information such as population, cost of living and average net salary related to cities based on their latitudes and longitudes.

Three structs and a class were created – ***struct Point*** which stores to x and y coordinates of a point, ***struct Node*** which stores the position of the node (a point), the city's name, its population, cost of living and net average salary, ***struct boundary_box*** which contains the minimum and maximum bounds of that specific node (helpful in subdividing tree into subtrees) and ***class QuadTree*** which defines the main structure of the tree and provides services such as inserting and searching for a point, returning statistics based on a location, printing the tree, clearing the tree and returning the size of the tree.

	struct Point	**struct Node**	**struct boundary_box**
Definition	Stores the: 1. x coordinate 2. y coordinate attributed to a city	Stores the: 1. Position of the city in terms of its coordinates 2. City's name 3. Population 4. Cost of living 5. Net average salary	Stores the: 1. center of the valid region 2. half of the width of the region associated with each root of a subtree (e.g. for the root of the tree, the center is (0,0) and half of the width is 128.
Members	***float x, float y***	***Point position*, *string city, int p, int r, int s***	***Point center, float half_of_dimension***
Empty Constructor	***Point();*** The 'x' is set to 0 and 'y' is set to 0.	***Node();*** All variables are set to null.	***boundary_box();*** All variables are set to null.
Defined Constructor	***Point(const float& xco, const float& yco);*** The 'x' is set to ***xco*** and 'y' is set to ***yco***.	***Node(Point _pos, const string& name, const int& pop, const int& col, const int& nas);*** The 'position' is set to ***_pos***, 'city' is set to ***name***, 'p' is set to ***pop***, 'r' is set to ***col***, 's' is set to ***nas***.	***boundary_box(Point _center, const float& _half);*** The 'center' is set to ***_center*** and 'half_of_dimension' is set to ***_half***.

class QuadTree:

This class contained both member objects (private) and member functions (public). Its private members included a ***vector <Node*> tile*** which is used to store a pointer to the node, an ***int _node_capacity*** which is the maximum capacity of a 'tile', a ***boundary_box boundary*** which is used to store the boundaries of that current node and pointers to the NW, NE, SW, SE trees (e.g. ***QuadTree *northwestTree***).

The member functions include:

1. Constructor: ***QuadTree();*** - This creates an "empty" quadtree (_node_capacity set to 1, boundary set to an empty boundary_box (boundary_box()), all pointers to the subtrees set to NULL and an empty node pointer vector)
2. Constructor: ***QuadTree(boundary_box bounds);*** - This creates a quadtree based on a boundary which is passed in. The _node_capacity is still set to 1, an empty node pointer vector is present and all pointers to the subtrees are set to NULL.

3. Destructor: ***~QuadTree();*** - This calls on clear() which is explained later.
4. ***void subdivide();*** - This function takes the current root of that tree/subtree and subdivides that it so that it points to four subtrees. The new half_of_dimension of the subtrees is half of the half_of_dimension of the root of that tree/subtree. All subtrees are also assigned a new center based on its location. For example: the northwest tree will now have a center with x coordinate, x coordinate of previous center minus new half_of_dimension, and y coordinate, y coordinate of previous center minus new half_of_dimension. This function performs in constant time as no iteration/recursion takes place. This function has no parameters passed in and because it is of 'void' type, does not return anything.
5. ***bool insert(Node* node);*** - This function inserts a node into a quadtree at a suitable location. It first performs error checking, that is, if the node which you are trying to insert is empty. In this case, it returns false. It then checks if the tile.size() is equal to 0. If it is, the node is inserted there. If the northwestTree == NULL (no space to insert the node at this position), it subdivides the current boundaries into four other boundaries (NW tree, NE tree, SW tree, SE tree). It then proceeds to check if that point contained in the node is already located in the tree (in this case, return false). Then, check to see where the point in the node can be inserted based on conditions and recursively call the insert function. This function has one parameter, a node to insert. It returns a 1 or 0 depending on if insertion was successful. Assuming that the tree is balanced, it is safe to say that the time complexity is O(lg(n)) as in worst case, you will have to traverse the height of the tree to insert a node.
6. ***bool search(Point *p);*** - This function searches for a point. It first performs error checking (if the current tile.size() is equal to 0, if all pointers to subtrees are NULL). Else, you will proceed to recursively search the subtrees based on which boundary the point is contained in. The function has one parameter passed in – the point. It returns a 1 or 0 depending on if the point was found. Assuming that the tree is always balanced, it is also safe to say that the time complexity is of O(lg(n)) as in the worst case, you will have to traverse the height of the entire tree to find a node.
7. ***void return_cityname(Point *p);*** - This function followed the same structure as the above search function. In this case, it took the point which was passed in and printed the city's name associated with that point. Since it was the same as the search function, it also had a time complexity of O(lg(n)).
8. ***void query_attribute(Point *p, const string direction, const string stat, const string attribute)*** – This function was written as a starting block to the following function. It performs basic functions such as error checking (if the point is actually found in the quadtree). It recursively calls upon itself to find the point which we need to find. When we find the point, a vector containing integer values (vector <int> vals) is created. It then performs more error checking – if the tree in the specified direction is NULL, if that the root of the tree is empty. Otherwise, depending on the attribute specified (p, r, s), we go to the function *p_GetStat(stat, vals), r_GetStat(stat, vals), s_GetStat(stat, vals)*. Again, this function has a time complexity of O(lg(n)) as in worst case, we will need to traverse the height of the entire tree to find the specified point. It does not have a return type.
9. ***int p_GetStat(const string stat, vector<int> &values); int r_GetStat(const string stat, vector<int> &values); int s_GetStat(const string stat, vector<int> &values);***- These functions were all written using the same structure. It is used to find the specified statistic using the query_attribute function above. It uses the layout of performing an inorder traversal on the tree (recursively calling itself using the northeast tree and the northwest tree, getting the root, then recursively calling itself using the southwest tree and the southeast tree). It stores the specified characteristic of each node (p, r or s)

in the vector (using push_back()) which is passed into the function. Then, depending on the string stat which is also passed into the function, it finds the minimum or maximum (using the algorithm STL) or the total (using a for loop to find the sum). It returns this value to the query_attribute function where it is printed. Due to the details listed, it is safe to say that the time complexity of this function is also O(lg(n)) as the function is recursively called upon itself using the specific subtrees.

10. ***void clear();***- This function iteratively calls upon itself if the tile at that point has a tile.size() equal to 1. Therefore, this function is performed on all nodes. All pointers to the subtrees is also equated to NULL. Because the function iteratively calls upon itself, its time complexity is O(n).
11. ***int size(int &initial_size);*** - This functions checks specific conditions as to where to increment the size variable which is passed into the function. If all the pointers to the subtrees are null and the tile.size() is equal to 0, the size is not incremented and the function returns the size (an integer). If all the pointers to the subtrees are null and the tile.size() is equal to 1, the size is increased by one. Else, if all the pointers to the subtrees are not null and the tile.size() is equal to 1, the size is incremented by 1 and condition checking is performed, after which, the function recursively calls itself (to find the size of each subtree). The function returns an integer. Because the function recursively calls upon itself, its time complexity is O(lg(n)).
12. ***void print();*** - This function performs an in order traversal by recursively calling the function upon itself. The function is recursively called for the northeast tree and the northwest tree. After, the root of the tree is printed. The function is then again recursively called for the southwest tree and the southeast tree. The function does not have a return type but does have printing capabilities.