Assignment 6

Assignment 6 essentially involves the implementation of a graph data structure. The following classes are implemented:

1. **Point implements PointInterface:**
   - The **class Point** contains three **private float variables X, Y, and Z.** These represent the coordinates of the points and there are **getter methods** for each of them. **XYZ** is an **array** of **float**, which contains the **X, Y, and Z coordinates of the point in order.** The method **getXYZcoordinate() returns this array.** There is a **private** Linked List **faceNeighbours (LinkedList<Triangle>)** which **stores all the triangles whose vertex is this point**. There are another **private** Linked List **edges(LinkedList<Edge>)** which **stores all the edges whose one of the endpoints is this point.** Along with these linked lists, there are private also the RBTree counterparts for all the above i.e **faceNeighboursRBT(RBTree<String, Triangle>) and edgesRBT(RBTree<String, Edge>).** The red-black tree data structures are essentially used for searching for a particular neighbor of the point i.e a triangle, of which the point is a part of or an edge, of which the point is a part of. There is another private integer **component** which is used when a BFS is performed over the complete graph, this essentially represents the component in which the point is present, the component if the latest component which is discovered by the BFS search. There are different **getter and setter** methods for all the above-mentioned variables of the point.

     - ➢ The method **equals(Point p)** returns true if the two points are identical or not. The method **distanceSqrd(Point p)** returns the square of the distance between the point p and the given particular point.
     - ➢ The **toString( ) method** is also overridden which is actually the X-coordinate, Y-coordinate, Z-coordinate.
     - ➢ The method **compareTo(Point p),** compares the point p in the arguments with the point object. The comparison is done firstly based on the X-coordinates of both the points, then the Y-coordinate of both the points(If the X-coordinates are the same) and then the Z-coordinates if(both X and Y coordinates of both the points are same). The method compareTo() returns 0 only when all the coordinates are the same for both the points.
     - ➢ The method **insertTriangle(Triangle t)** inserts the triangle in the linked list **triangles as well as the red-black tree trianglesRBT.**
     - ➢ The method **insertEdge(Edge e)** inserts the edge e in the linked list **edges** and the red-black triangle **edgesRBT**.
     - ➢ The method **searchEdge(Edge e)** performs a search in the red-black tree edgesRBT for the Edge in the argument and returns the edge if it is found and null if it isn't found.

2. **Edge implements EdgeInterface:**
   - The **class Edge** contains the endpoints of an edge stored in array **EndPoints(Point [ ])** of length 2. The method **edgeEndPoints()** returns the array **EndPoints.** The LinkedList **triangles**

(LinkedList<Triangle>) stores all the triangles that share the given particular edge. The method **numTriangles()** returns the **number of triangles that share the particular edge object which is essentially the size of the LinkedList triangles**.

- ➢ The method **getLenSqrd()** returns the **square of the length of the edge** which is **calculated** from the **coordinates of the endpoints**.
- ➢ The method **compareTo()** is overridden which compares two edges based on their lengths.
- ➢ The method **toString()** is also overridden which returns the string representation of the edge. Since the edges are stored in an red-black tree, and the key for the edge is the string representation of the edge, the toString() representation contains the lower point first, and the higher point as the second.
- ➢ The method **insertTriangle(Triangle t)** inserts the triangle in the linked list **triangles** present in the edge.

3. **Triangle implements TriangleInterface:**
   - The **class Triangle** implements the triangle interface and it contains, an array **coord(PointInterface [ ])** which contains three-point objects which are unordered. There is an array of size 3 **edges(Edge[])** which represents three edges of the triangles. There is a private string variable **color** that is assigned during the BFS scan made over the triangles. The private integer variable **distance** represents the distance of the triangle(Topological distance) from the triangle which is the source of the BFS scan. The **private integer variable time** represents the relative time of the insertion of this triangle relative to other triangles and therefore it is used to establish an ordering among different triangles. There are **getter and setter** methods for the above variables.

   - ➢ The method **triangle_coord( )** returns this array.
   - ➢ The **toString() method** is also overridden in order to represent the string representation of the triangle.
   - ➢ The **compareTo() method** is overridden and returns an integer corresponding to the relative times of the two triangles.
   - ➢ The **toString() method** returns the string representation of the triangle as the combination of the string representation of the points, in which the points are ordered from small to large as per their mutual comparisons. This is done to ensure the uniqueness of the keys as the string representation is used as a key for the red-black tree of triangles.

4. **Shape implements ShapeInterface:**
   - The **class Shape** implements the default methods of the ShapeInterface. The **red-black tree TrianglesRBT** contains **all the triangles which** are **created.** The **red-black tree EdgesRBT** contains all the edges which are **created for the graph,** and the **red-black tree pointsRBT** contains all the **point objects which are created for the graph.** The **LinkedList triangles** contain all the

triangles which are contained in the graph. The linked list **edges** contains all the edge objects, which are formed and the linked list **points** contain all the point objects which are formed. The **integer variable globalTime** maintains a global time counter which is used to establish relative ordering among the triangles based on their time of insertion. The integer variables **Proper, semiProper, and imProper** are variables that are used to keep track of the edges which are a part of exactly 2, only 1 and more than 2 triangles respectively.

- The following methods are implemented:
  - ➢ The method **ADD_TRIANGLE(float [ ] triangle_coord)** returns a **boolean** indicating whether the **triangle formed by the three points is valid or not**. The **validity** of the triangle is **checked** by **calculating the area of the triangle by using the vector cross product**. If the area turns out to be **0**, then we have three collinear points and therefore **false** is **returned**. If the area turns out to be non-zero, then firstly, **new point objects are created and a search is performed in the red-black tree containing all the point objects.** This procedure takes place in **O(logV) where V is the number of triangles.** Then the method has different executions depending on the different scenarios as follows:
    1. If none of the three points are present, then the already formed points are inserted in the red-black tree and the linked list and consequently new edges, are added and inserted in the linked list of edges and red-black tree as well. This procedure would take place in **O(logE) where E is the total number of edges.**
    2. If a point already exists then 2 new points are inserted and all the new edges are inserted in both the data structures, the linked list, and the red-black tree. This procedure would take place in **O(logE) where E is the total number of edges.**
    3. If there are 2 points that do exist, then a check is made for the presence of an edge in the red-black tree containing all the edge objects. Then depending on the existence of the edge or not, new edges are inserted in both the data structures. This procedure would take place in **O(logE) where E is the total number of edges.**
    4. If all the three points already exist, then a search would be made for the edges between them and new edges are added if the edges aren't present, and not if they are already present. This procedure would take place in **O(logE) where E is the total number of edges.**

Also, a new Triangle would be created with each call to the query and the data structure containing Triangles(**LinkedList triangles and red-black tree triangles**) would be updated as well. The counter **globalTime** would be incremented as well and also the variables **Proper, semiProper and imProper would be updated as the different edges are discovered.**
Overall the time complexity of the add triangle query is O(logV+logE) where V and E are the total number of triangles and number of edges respectively.

Assignment 6

➢ The method **TYPE_MESH** returns the mesh type for the input triangles. It checks the number of **Proper, imProper, and semiProper edges.** If the number of **imProper edges** is greater than 1 then 3 would be returned. If the number of semiProper edges is greater than 1 then 2 would be returned, and apart from the above 2 scenarios, 1 is returned.
The complexity of this method is **O(1) due to the pre-processing which is carried in the add_triangle Query.**

➢ The method **BOUNDARY_EDGES()** makes a linear scan over all the edges and if the edge has only 1 triangle of which it is a part of, then all such edges are stored separately in a linked list and then returned as an array. Then the edges are copied to an array and these are sorted by using the merge sort algorithm. The complexity of this method is **O(E + BlogB) where E is the number of edges of triangles and B is the number of boundary edges.**

➢ The method **NEIGHBOURS_OF_TRIANGLE()** first searches the triangle in the red-black tree and if the triangle is found, then each of it'd edge is iterated over one after the another, and from each of the edge, the linked list of all the triangles which share that particular edge is used for retrieving the neighbors of the triangle queried. Then a selective merge is performed on the linked lists of all the three edges as the triangles stored within an edge are already sorted in order of their insertion time. The complexity of the above procedure would be **O(logV+N) where V is the number of vertices and N is the number of neighbors of the queried triangle.**

➢ The method **EDGE_NEIGHBOR_TRIANGLE()** first searches for the triangle present in the red-black tree **trianglesRBT** and then if the triangle is found, then all its edges are retrieved and are then sorted according to their lengths and finally returning than as an array. The complexity of the method is **O(logV) where V is the number of triangles.**

➢ The method **VERTEX_NEIGHBOR_TRIANGLE()** first searches for the triangle in the red-black tree **trianglesRBT** and then if the triangle is found, then all its vertices are returned in an array. The complexity of this method is **O(logV) where V is the total number of triangles.**

➢ The method **EXTENDED_NEIGHBOR_TRIANGLE()** first searches for the triangle in the arguments, first searches for the queried triangle in the red-black-tree. If the triangle is present, then all it's vertices are stored. Three-pointers are maintained namely a,b, and c which iterate over the linked lists of respective vertices and the standard merge algorithm is used to merge the triangles, as, within the linked lists of each of the points, the triangles are already sorted by their insertion times. The complexity of the above procedure is **O(logV+N) where V is the total number of triangles and N is the number of extended neighbors which the triangle has.**

Assignment 6

- ➢ The method **INCIDENT_TRIANGLES()** involves a search for the queried point in the red-black-tree **points** and if the point exists, a linked list of all the triangles that share the common point is stored within the point and this linked is copied in an array. Since the triangles present in the linked list are already sorted according to their time of insertion, no further sorting is required. The complexity of the above procedure is **O(logV+N) where V is the total number of triangles present and N is the neighbors of the queried point.**

- ➢ The method **NEIGBORS_OF_POINT()** involves a search for the queried point in the red-black tree **points** and if the point exists, a linked list of all the edges whose one of the end-points is the queried point is retrieved and iterated over to find the other neighbor of the queried point. The complexity of the above procedure if **O(logV + D) where V is the total number of triangles present and D is the degree of the queried vertex.**

- ➢ The method **EDGE_NEIGHBORS_OF_POINT()** involves a search for the queried point in the red-black tree points and if the point exists, a linked list of all the edges whose one of the endpoint is the queried point is retrieved and iterated over to copy it into an array. This array is then sorted using the merge sort algorithm and returned. Sorting is done on the basis of square of the length of the edges. The complexity of the above procedure is **O(logV + DlogD) where V is the total number of triangles and D is the degree of the queried point.**

- ➢ The method **TRIANGLE_NEIGHBOR_OF_EDGE()** involves a search for the queried edge in the red-black tree **edgesRBT.** If the edge exists, then the linked list of all the triangles that the queried edge is a part of is retrieved from the searched edge and is iterated over to copy all the triangles into an array. Since the triangles are already sorted based on their insertion time, no further sorting is required and the array is returned. The complexity of the above procedure is **O(logE + N) where E is the total number of edges and N is the total number of triangles whose one of the edges is the queried edge.**

- ➢ The method **COUNT_CONNECTED_COMPONENTS()** involves a standard BFS() traversal of the graph where the vertices are triangles. Firstly, a reset is performed and the color of each and every triangle is set to be WHITE and distance to be MAX_VALUE. The number of iterations of the outer loop of the BFS which checks for the WHITE color (the triangles which are not visited) corresponds to the number of connected components and this count is stored and returned at the end of the method. The complexity of the above procedure is **O(V+E).**

- ➢ The method **IS_CONNECTED()** returns a boolean indicating whether both the triangles in the arguments are connected or not. Firstly, both of these triangles are searched for in the red-black-tree **triangles.** If both the triangles are not present, then false is returned. On the other hand, if both the triangles are present, then the method BFSvisitD() is called for any 1 of the two triangles. Prior to this, the color of all the triangles is set to be WHITE and the distance is set to be MAX_VALUE. The method BFSvisitD() visits all the triangles present in the connected component of the source triangle and after the completion, the color of each and every triangle in the component is BLACK their distance would be some finite value, and the predecessor reference won't be null. After this method is executed, the predecessor field of the other triangle, the triangle not used as the source is checked and if that's null, false is returned and that's not the case, true is returned. The complexity of this procedure is **O(V+E) where V is the total number of triangles and E is the total number of edges (here edge of a particular triangle refers to the number of edge_neighbors).**

➢ The method **MAXIMUM_DIAMETER()** first identifies the size of the largest component present in the graph. Here size refers to the maximum number of triangles present in a particular component. Firstly, the color of all the triangles present in the graph is set to WHITE and their distance is set to MAX_VALUE. Then the standard BFS algorithm is used and for each triangle that is not visited(whose color is WHITE), the method BFSvisitll() is called. This method returns a linked list containing all the triangles present in a particular component. The size of the component is essentially the size of this linked list. The linked list of max size is temporarily stored in a variable and it is replaced by a new larger linked list if found. After the completion of the above BFS execution, the variable comp contains the linked list of triangles which are a part of the largest component. Then an iteration is made over the linked list of triangles of the largest component and for each triangle, the color is set to WHITE and the distance to be MAX_VALUE. Now, the BFSvisitD() is called for each of the triangles present in the component and the maximum diameter is stored for each and call made to the method BFSvisitD(). This value is updated if a larger diameter is obtained. The complexity of the above procedure would be **O(V*2) where V is the number of triangles present in the graph.**

➢ The method **CENTROID_OF_COMPONENT()** involves the search of the point in the arguments in the red-black tree **points.** If the point is **not present, null is returned.** On the other hand, if the point is present, firstly an iteration is made over all the points in the graph and the value of the component field is set to 0. Then the first triangle of the linked list of the triangles within the queried point is selected and an algorithm similar to the BFSvisitll() is used with the source as the extracted triangle. Three **float variables X,Y and Z are maintained along with them an integer variable count is maintained which is used for storing the number of vertices present in the component.** In the modified algorithm, whenever a triangle is extracted from the queue(LinkedList<Triangle>), all its vertices are checked for their components. If the component of any vertex is 0, then its X,Y and Z coordinates are added to the **float variables which are maintained. The value component of the vertex is set to 1. This is used to prevent the recounting of a particular vertex. Also the integer variable count is incremented.** After the termination of the BFSvisitll() algorithm, we'd have the X coordinates of all the points in the component totaled and stored in the **float variable X.** Similarly, the Y and Z coordinates for all the points would be totaled and stored in their **respective float variables.** This total is divided by count and the newly obtained coordinates are then used for creating a new point object and that point object is created and returned. The complexity of the above procedure would be **O(V+E) where V is the total number of triangles present in the graph and E is the total number of edges (Here the notion of an edge is an neighbor of a triangle).**

- The method **CENTROID()** involves calculation of the centroid of all possible components present in the graph. Firstly, an iteration is performed over all the points and their components are set to be 0. Then an iteration if performed over all the triangles and their color is set to WHITE. The algorithm used is similar to the BFS() traversal on a graph. However, for each iteration of the outer loop, a linked list is maintained which stores all the points present in the current component. Whenever a new triangle is dequeued, all its vertices are checked for their components. If the component of the vertex is not equal to the current component, (component is a counter which is used to count the total number of components) then the vertex is added to the linked list **allCentroids**. After the traversal of all the triangles of a given component, the centroid for that particular component is calculated and inserted into linked list **centroids** (LinkedList<Point>) which is responsible for storing the centroids(Point object) for all the components. The linked list of **centroids** is again iterated over and copied into an array which is returned afterwards.
  The complexity of the above procedure is **O(V+E) where V is the total number of triangles present and E is the total number of edges. Here, the notion of an edge is a neighbors of a particular triangle.**

- The method **printArray()** is used for printing the contents of the array in the argument in the single line.
- The method **edgeSearch()** constructs a new edge from the coordinates of the endpoints of the edge and searches for the edge object in the red-black-tree **edgesRBT.** The complexity of the above procedure is **O(logE) where E is the total number of edges. Here edge refers to a side of a triangle. A triangle has 3 edges.**
- The method **triangleSearch()** takes in the coordinates of the triangles to be searched and constructs a new dummy triangle in order to search for that triangle in the red-black tree **trianglesRBT().** The complexity of this method is **O(logT) where T is the total number of triangles present in the graph.**
- The method **pointSearch()** takes in the coordinates of the point as a parameter and creates a dummy point in order to search for that point in the red-black tree **pointsRBT.** The complexity of this method would be **O(logV) where V is the total number of triangles present in the graph.**

- ➢ The method **mergeSort()** is the standard merge sort implemented for **edge, triangle and point objects**. The complexities would be **O(NlogN) where N is the size of the array to be sorted.**
- ➢ The method **BFS() along with BFSvisitD() and BFSvisitll() and BFSvisit()** are used for accomplishing the Breadth-First-Traversal of the graph. The method **BFSvisitD() returns the diameter of the component, the method BFSvisitll() returns the linked list of all the triangles present in a particular component and the method BFSvisit() assigns the component to each and every point of a particular component. The complexity of the method would be O(V+E) where V is the total number of triangles present in the graph, E is the total number of edges. Here the notion of the edge of a vertex is an edge neighbor of the triangle.**
- ➢ The method **merge()** takes three linked lists in the arguments and returns a single linked list which is formed by merging all the three already sorted linked lists in the arguments. Also, a Triangle target is taken as the argument and this is used to prevent the insertion of this particular triangle in the linked list. The complexity of this method would be **O(A+B+C) where A,B, and C are the sizes of the linked lists present in the arguments.**

5. Red Black Tree:

● The **class RedBlackNode<T extends Comparable, E>** implements the **RBNodeInterface<E>.** The different **private fields** of this class are: ➢ **color(String)** represents the **color** of the node which can be **RED or BLACK or EXT(in case of external nodes).**

➢**left (RedBlackNode<T,E>)** is a reference to the left child of the node.

➢**right(RedBlackNode<T,E>)** is a reference to the right child of the node.

➢**parent(RedBlackNode<T,E>)** is a reference to the parent of the node.

➢**value(E)** corresponds to the value stored by this node.

➢ **key(T)** corresponds to the key of the node.

➢ The fields **color(String), key(T) and value(E)** are passed to the constructor.

There are **getter and setter** methods for each of the above-mentioned parameters are implemented. The method **addValue(E v) adds a value to the ArrayList values** and the method **compareTo(RedBlackNode<T,E> b)** compares the **toString() representations of the current node and the node in the argument.**

● The class **RBTree** has a field **root(RedBlackNode<T,E>)** which is a reference to the root of the tree and it is initialized as null in the constructor. The following methods are implemented:

➢ **search(T key)** searches for the key in the red-black tree. The reference curr(RedBlackNode<T,E>) refers to the current node which is compared with the key. The algorithm starts from the root and comparisons are made with the key of curr (Initially curr is assigned to be root). If the key which is to be searched is greater than the root, then curr is assigned to its right child and if the key is lesser than the key of the curr, then curr is assigned to its left child. If the keys are equal then curr is returned. The termination condition of the loop is when curr refers to an external node(i.e. Node with color “EXT”) and if the keys of the curr and the key in the argument match. Since in this algorithm, we are traversing the height of the key and therefore the complexity of the above algorithm would be **O(height of the tree).**

➢ The method **insert(T key, E value)** inserts a new Node whose color is defaulted to **“RED”**. Firstly, the location of the new node which needs to be inserted is found, and then the insertion is made at the node by replacing the external nodes by the new nodes. If the node is inserted as a child of a “BLACK” node then no restructuring is required. If it’s inserted as a child of a “RED” node, then there would be a double red problem. The parent of the “RED” node at which insertion is made would be guaranteed to be “BLACK” and if the other child of this “BLACK” node is a “BLACK” colored node, then a rotation would fix the issue and the insertion would terminate and if the other child of the “BLACK” grandparent is “RED” than a recoloring needs to be done between the parent and the grandparent of the node which is inserted. This may result in a double red problem at an above level and therefore we need to check this at a higher level. If we end up recoloring the root while traversing upwards, then we would again recolor the root to be “BLACK” thereby increasing the black height of all the external nodes by 1. In the worst-case deletion of the root at the leaf would be done and we’d end up recoloring the root. Doing this would lead to traversal along with the height of the root twice and therefore the complexity of the method would be **O(height of the tree).**

➢ The method **Rotation(RedBlackNode<T,E>x, RedBlackNode<T,E>y, RedBlackNode<T,E> z)** takes in nodes which the node y is the parent of node z and node x is the parent of the node. There are 4 different configurations in which they can exist and accordingly pointers are changed so that the new order obtained of the inorder traversal remains consistent with the original order. Since a finite number of pointers are changed the complexity of the method would be **O(1).**