README – ASSIGNMENT 6

Data Structures implemented for this assignment:-

Linked Lists (Doubly linked):

- With head and end references
- O(1) insertion at the end
- O(1) deletion of given position in the Liked List

Red Black Trees:

- O(log n) insert
- O(log n) search

Implementation Details:-

RedBlackNode<T extends Comparable, E>:

Fields:-

- colour
- key
- values
- leftchild
- rightchild
- parent

Methods:-

- getValue():
  Returns the first value of the List<E> of values
- getValues():
  Retruns the List<E> of values of the node

Fields:-

- root: root of the Red-Black tree

Methods:-

- insert(T key, E value): Time Complexity: O(log n) n: number of elements in the RBTree
    - Calls searchinsertpos(key) to find the parent whose child will be the new inserted node
    - Black Parent: If the parent is black then we insert the node by colouring it red
    - Red Parent:If the parent is red then also the node is inserted with a red colour but this creates a double red problem therefore we call the remove_double_red(toinsertnode) function to remove the double red problem
- remove_double_red(RedBlackNode<T, E> node): Time Complexity: O(log n) n: number of elements in the RBTree
  remove_double_red method takes the red child node as a parameter and checks if there is a double red problem of this node with its parent, if there is one then it removes it recursively.
    - Cases:-
        - Null Parent: Now the problem is that the root is red therefore we colour the root black and we are done
        - Black parent: No double red problem: we are done
        - Red Parent:
          Grandparent must be black(because we have created the problem in this way)
          Cases:-
            - Red uncle exists: Then we do recolouring: uncle and parent black and the grandparent red. This might lead to a double_red problem at the grand parent therefore we call the remove_double_red function again passing the grandparent
            - No Black uncle exists: In this case we need restructure of the tree but with a guarantee that there will not be any double red problem further up after the restructure.
                - Restructure is done separately for the four cases of configuration of the grandparent, parent and child which could be:-

- LL i.e. parent is left child of grandparent and child is left parent of parent then rotateLL() was called.
- LR i.e. parent is left child of grandparent and child is right parent of parent then rotateLR() was called.
- RL i.e. parent is right child of grandparent and child is left parent of parent then rotateRL() was called.
- RR i.e. parent is right child of grandparent and child is right parent of parent then rotateRR() was called.

- rotateLL(RedBlackNode<T,E> n,RedBlackNode<T,E> p,RedBlackNode<T,E> g,RedBlackNode<T,E> t1,RedBlackNode<T,E> t2,RedBlackNode<T,E> t3, RedBlackNode<T, E> ggp): Time Complexity: O(1)
  Takes all the nodes whose references could change in LL configuration and changes the references accordingly
- rotateLR(RedBlackNode<T,E> n,RedBlackNode<T,E> p,RedBlackNode<T,E> g,RedBlackNode<T,E> t1,RedBlackNode<T,E> t2,RedBlackNode<T,E> t3, RedBlackNode<T, E> ggp): Time Complexity: O(1)
  Takes all the nodes whose references could change in LR configuration and changes the references accordingly
- rotateRL(RedBlackNode<T,E> n,RedBlackNode<T,E> p,RedBlackNode<T,E> g,RedBlackNode<T,E> t3,RedBlackNode<T,E> t4,RedBlackNode<T,E> t5, RedBlackNode<T, E> ggp): Time Complexity: O(1)
  Takes all the nodes whose references could change in RL configuration and changes the references accordingly
- rotateRR(RedBlackNode<T,E> n,RedBlackNode<T,E> p,RedBlackNode<T,E> g,RedBlackNode<T,E> t3,RedBlackNode<T,E> t4,RedBlackNode<T,E> t5, RedBlackNode<T, E> ggp): Time Complexity: O(1)

Takes all the nodes whose references could change in RR configuration and changes the references accordingly

- search(T key): Time Complexity: O(log n) n: number of elements in the RBTree
  Search is similar to search in a binary search tree. At each node a decision is taken based on the comparison of the key of the node and the given key.
    - If keys are equal: Then then search is successful
    - If node key is greater: Search in the left subtree if it is not null
    - If node key is less: Search in the right subtree if it is not null
    - If the subtree in which we have to continue the search is null then we return null
- searchinsertpos(T key): Time Complexity: O(log n) n: number of elements in the RBTree
    - Searchinsertpos is similar to search in the case when a node having the given key exists in the Red Black Tree i.e. it returns the node found.
    - In case if the node is not present in the Red Black Tree then we return the parent with which we compared the key last as it would be the parent of the new node is we were to insert a new node with key as the given key

OTHER CLASSES:

POINT

Fields:

1. Characterizing features:
    a. X
    b. Y
    c. z
    d. coordinated array
2. Linked Lists:
    a. Point neighbours

    b. Edge neighbours
    c. Triangle neighbours
3. Boolean values for marking:
    a. Old
    b. Taken_for_centroid
    c. Taken_for_component

Non-trivial methods:

1. Compare_To: Compares points with x first and then by y. Returns 0 if both x and y are the same for two points

EDGE:

Fields:

1. Characterizing features:
    a. P1
    b. P2
       Note: p1 and p2 are stores such that p1<p2
    c. End points array
    d. Length (Square of Euclidean length)
2. Triangle neighbours linked list
3. Boolean value: old
4. Boundary position: non null if given edge is a boundary edge and it refers to the position of the given boundary in the boundaries linked list of a given shape so as to facilitate efficient deletion

Non-trivial methods:

1. Compare_To: Compares edges with p1 first and then p2 and returns 0 if p1 and p2 are same for 2 edges.

TRIANGLE:

Fields:

1. Characterizing features:

    a. Points: p1,p2,p3
       Note: in a triangle p1<p2<p3
    b. Edges: e1,e2,e3
       Note: in a triangle e1<e2<e3
    c. Points and edges array
    d. Arrival rank
2. Linked Lists:
    a. Neighbours
    b. Extended neighbours
3. Boolean values for marking:
    a. In_a_component
    b. Considered_as_ext_neigh

Non-trivial methods:

1. Least_arrival_rank(t1,t2,t3): returns the triangle that arrived first given triangles may or may not be same or even be null
2. Compare_to: compares triangles by p1 first then by p2 and then by p3.

Shape Class and Answering Queries:

Fields:

1. Three Red Black Trees for:
    a. Points
    b. Edges
    c. Triangles
2. Linked Lists:
    a. Boundaries
    b. Triangles
3. Triangle number: Arrival rank to be assigned to the next triangle
4. Mesh Type:
    a. Proper
    b. Semi-proper
    c. Improper

Utility of the above data structures:-
I used red black trees to store the points and edges of the shape so that if any query gives the characteristics of any edge or point or triangle then I can efficiently get the object for that entity if at all it exists without having to go through all the entity objects formed in the shape as an ordering is assigned to the each of the entities i.e. two entities (points/ edges/ triangles) can be compared. Therefore the Red-Black tree empowers me to get the required object in O(log(number of entities)) time.

I maintained linked lists for all the triangles and all the boundary edges of the shape. Triangles in a linked list are useful so that I can iterate through all the triangles (that form a graph) which I am doing using bfs. Boundaries are stored in a linked list for the boundaries query.

Queries:

1. ADD TRIANGLE: Time requirement: O(log (number of triangles) + extended degree of the new triangle)
   Auxiliary functions used: none
   Implementation details:
   Given the coordinates of three points first I checked whether a triangle can be formed by these three points or not i.e. whether or not the three points are collinear by checking if the area of the triangle formed by these points is 0 or not.
   Then I made a new triangle object with the given points and insert it in the Red Black tree and linked list. In O(log(number of triangles)) + O(1) time respectively.
   Then I updated the all the triangle's neighbours field by adding this triangle to all the triangles that are its neighbours in O(degree) time by updating the neighbours of the edges and updated the neighbours field of the new triangle simultaneously.
   Then I updated the neighbours of the edges by adding the new triangle in O(1) time
   Then I similarly updated the extended neighbours of all the triangles and neighbours of points in O(extended degree) + O(1) time.

Then I updated the edge neighbours and point neighbours of the points by adding the new edges or points whichever lead to a new relationship with a given point. All in O(1) time
Then I updated the boundary edges linked list by adding the edges that have only 1 triangle neighbour and removing those that have 2 as by the addition of a new triangle a boundary edge can become a non-boundary edge as its neighbours might increase by 1 (and become 2). Also in O(1) time : Addition to linked list is clearly O(1) removal was also O(1) as I stored the position of the boundary edges in the boundary edges as a field which was null if the edge is not a boundary edge and gave its position in the linked list otherwise making deletion in the doubly linked list require constant time.
Then I updated the mesh type. It didn't change if it were improper, it became improper if any edge has more than 2 neighbours. It became semi-proper is it was not improper and number of boundary edges was non zero else it became proper. Time O(1).

2. TYPE MESH: Time Requirement: O(1)

   Auxiliary functions used: none
   Implementation Details: It is stored as a field.

3. BOUNDARY EDGES: Time Requirement: O((Number of boundary edges)*log(Number of boundary edges))
   Auxiliary functions used: mergesort O(nlogn) and merge O(n)
   Implementation Details: They are stored as a field in a linked list and I implemented merge sort by first making an array out of the linked list and then implementing the merge sort in O((Number of boundary edges)*log(Number of boundary edges)) time.

4. NEIGHBORS OF TRIANGLE: Time Requirement: O(log(number of triangles) + required triangles)
   Auxiliary Functions used: none

Implementation Details: First I search the triangle in the RB tree in O(log(number of triangles)) time then I return the required triangles in an array by iterating through the linked list in O(required triangles) time.

5. EDGE_NEIGHBOR_TRIANGLE: Time requirement : O(log(number of triangles))
   Auxiliary functions used: none
   Implementation Details: Return the three edges of the tringle stored in the triangle as fields after getting the triangle object if it exists in O(log(number of triangles)) time.

6. VERTEX_NEIGHBOR_TRIANGLE: Time requirement : O(log(number of triangles))
   Auxiliary functions used: none
   Implementation Details: Return the three vertices of the tringle stored in the triangle as fields after getting the triangle object if it exists in O(log(number of triangles)) time.

7. EXTENDED_NEIGHBOR_TRIANGLE: Time requirement: O(log(number of triangles) + required triangles)
   Auxiliary functions used: none
   Implementation Details: Return the three vertices of the tringle stored in the triangle as fields after getting the triangle object if it exists in O(log(number of triangles)) time.

8. INCIDENT_TRIANGLES: Time requirement: O(log(number of points) + required triangles)
   Auxiliary functions used: none
   Implementation Details: Finding the point object in O(log number of points) time and making the array required in O(required triangles) time.

9. NEIGHBORS_OF_POINT: Time requirement: O(log(number of points) + required points)

Auxiliary functions used: none
Implementation Details: Finding the point object in O(log number of points) time and making the array required in O(required points) time.

10. EDGE_NEIGHBORS_OF_POINT: Time requirement: O(log(number of points) + required edges)
    Auxiliary functions used: none
    Implementation Details: Finding the point object in O(log number of points) time and making the array required in O(required edges) time.

11. FACE_NEIGHBORS_OF_POINT: Time requirement: O(log(number of points) + required triangles)
    Auxiliary functions used: none
    Implementation Details: Finding the point object in O(log number of points) time and making the array required in O(required triangles) time.

12. TRIANGLE_NEIGHBOR_OF_EDGE: Time requirement: O(log(number of edges) + required triangles)
    Auxiliary functions used: none
    Implementation Details: Finding the edge in O(log number of edges) time and making the array required in O(required triangles) time.

For the next 6 queried I used slight variants of the BFS algorithm

I implemented the bfs and bfs_allComp for referenfe for all the 6 queried. Their implementation details are as follows:

(A) Bfs_allComp(): Time Requirement: O(m+n) where m is the number of edges of he graph and n is the number of nodes in the graph here nodes are triangles and edges are the supposed to be between those triangles that have a common edge as they are connected by

the neighbour relationship in our graph definition.
In this method I iterated through the linked list of the triangles (nodes) of the graph and did bfs with any un marked vertex as the starting vertex of the bfs and marked all the triangles as visited in the component of the starting vertex and then terminated after all components were marked (or when there was no triangle left unmarked in the linked list).

(B) Bfs(): Time Requirement: O(m+n) where n is the number of nodes in the component and m is the number of edges
Takes a triangle as starting node and does a breadth first traversal of the graph using a queue and marking all the visited nodes along the way.

13. COUNT_CONNECTED_COMPONENTS: Time Requirement: O(m+n)
    Auxiliary functions used:
    a. bfs_allComp_for_COUNT_CONNECTED_COMPONENTS
    b. bfs_for_COUNT_CONNECTED_COMPONENTS

    Implementation Details: We count all the components by counting the number of times we called bfs in bfs_allComp.

14. IS_CONNECTED: Time Requirement: O(m1+n1) m1,n1:edges and nodes in component of t1
    Auxiliary functions used:
    a. bfs_for_IS_CONNECTED

    Implementation Details: We do bfs by starting with t1 and stop if we get t2 in the component of t1 or terminate if we do not.

15. MAXIMUM_DIAMETER: Time Requirement: $O(n^2)$
    Auxiliary functions used:
    a. bfs_allComp_for_MAXIMUM_DIAMETER
    b. bfs_for_MAXIMUM_DIAMETER
    c. bfs_allComp_for_diameter
    d. bfs_for_diameter

Implementation Details:

To find maximum diameter I first found the component with the maximum number of triangles using a and b and then for the largest component I found the diameter using c and d by doing bfs through all the triangles of the component to get the eccentricity of each node and then reporting the maximum eccentricity as the diameter.

16. CENTROID: Time Requirement: O(m+n)

    Auxiliary functions used:

    a. bfs_allComp_for_CENTROID
    b. bfs_for_CENTROID
    c. mergesort
    d. merge

    Implementation Details:

    To find the centroid of all the components I maintained a centroids linked list once I got a component by bfs I maintained a Boolean value in each point in the component telling whether or not it has been considered in finding the centroid to make sure that each point is considered only once. I used merge sort to sort the centroids in the required order.

17. CENTROID_OF_COMPONENT: Time Requirement: O(m1+n1)

    Auxiliary functions used:

    a. bfs_for_CENTROID_OF_COMPONENT

    Implementation Details:

    I got a component by bfs I maintained a Boolean value in each point in the component telling whether or not it has been considered in finding the centroid to make sure that each point is considered only once.

18. CLOSEST_COMPONENTS: Time Requirement: $O(n^2)$

    Auxiliary functions used:

    a. bfs_for_CLOSEST_COMPONENTS
    b. get_closest_points
    c. dist

Implementation Details:

For finding closest components I first returned a linked list of all components and then iterated through all possible pairs of points such that they come from distinct components and found their minimum which took $O(n^2)$ time in the worst case.