Given n line segments, find if any two segments intersect

We have discussed the problem to detect if two given line segments intersect or not. In this post, we extend the problem. Here we are given n line segments and we need to find out if any two line segments intersect or not.

Naive Algorithm A naive solution to solve this problem is to check every pair of lines and check if the pair intersects or not. We can check two line segments in O(1) time. Therefore, this approach takes $O(n^2)$.

Sweep Line Algorithm: We can solve this problem in **O(nLogn)** time using Sweep Line Algorithm. The algorithm first sorts the end points along the x axis from left to right, then it passes a vertical line through all points from left to right and checks for intersections. Following are detailed steps.

1) Let there be n given lines. There must be 2n end points to represent the n lines. Sort all points according to x coordinates. While sorting maintain a flag to indicate whether this point is left point of its line or right point.

2) Start from the leftmost point. Do following for every point
…..**a)** If the current point is a left point of its line segment, check for intersection of its line segment with the segments just above and below it. And add its line to *active* line segments (line segments for which left end point is seen, but right end point is not seen yet). Note that we consider only those neighbors which are still active.
….**b)** If the current point is a right point, remove its line segment from active list and check whether its two active neighbors (points just above and below) intersect with each other.

The step 2 is like passing a vertical line from all points starting from the leftmost point to the rightmost point. That is why this algorithm is called Sweep Line Algorithm. The Sweep Line technique is useful in many other geometric algorithms like calculating the 2D Voronoi diagram

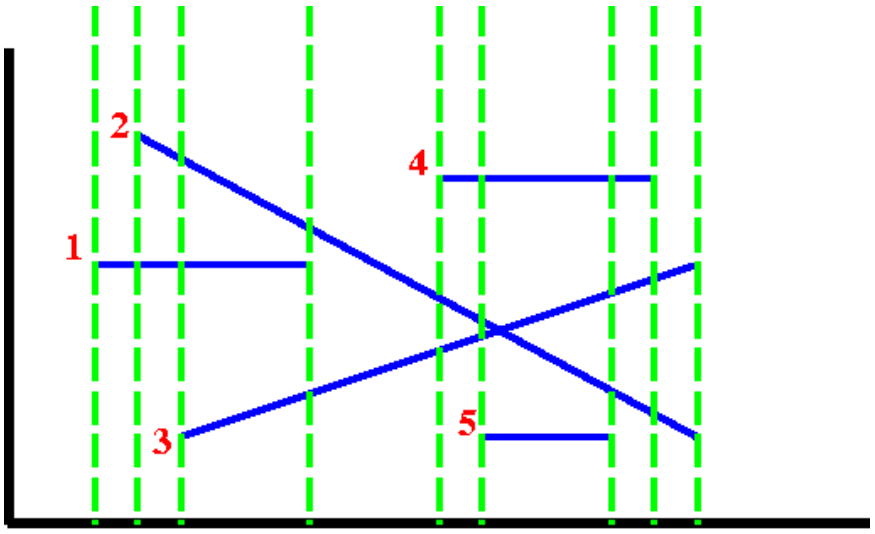
What data structures should be used for efficient implementation?
In step 2, we need to store all active line segments. We need to do following operations efficiently:
a) Insert a new line segment
b) Delete a line segment
c) Find predecessor and successor according to y coordinate values
The obvious choice for above operations is Self-Balancing Binary Search Tree like AVL

Tree, Red Black Tree. With a Self-Balancing BST, we can do all of the above operations in O(Logn) time.
Also, in step 1, instead of sorting, we can use min heap data structure. Building a min heap takes O(n) time and every extract min operation takes O(Logn) time (See this).

PseudoCode:
The following pseudocode doesn't use heap. It simply sort the array.

```
sweepLineIntersection(Points[0..2n-1]):
1. Sort Points[] from left to right (according to x coordinate)

2. Create an empty Self-Balancing BST T. It will contain all active line
   Segments ordered by y coordinate.

// Process all 2n points
3. for i = 0 to 2n-1

    // If this point is left end of its line
    if (Points[i].isLeft)
       T.insert(Points[i].line())  // Insert into the tree

       // Check if this points intersects with its predecessor and successor
       if ( doIntersect(Points[i].line(), T.pred(Points[i].line()) )
         return true
       if ( doIntersect(Points[i].line(), T.succ(Points[i].line()) )
         return true

    else  // If it's a right end of its line
       // Check if its predecessor and successor intersect with each other
       if ( doIntersect(T.pred(Points[i].line(), T.succ(Points[i].line()))
         return true
       T.delete(Points[i].line())  // Delete from tree

4. return False
```

Example:
Let us consider the following example taken from here. There are 5 line segments 1, 2, 3, 4 and5. The dotted green lines show sweep lines.



sweep lines

Following are steps followed by the algorithm. All points from left to right are processed one by one. We maintain a self-balancing binary search tree.

Left end point of line segment 1 is processed: 1 is inserted into the Tree. The tree contains 1. No intersection.

Left end point of line segment 2 is processed: Intersection of 1 and 2 is checked. 2 is inserted into the Tree. No intersection. The tree contains 1, 2.

Left end point of line segment 3 is processed: Intersection of 3 with 1 is checked. No intersection. 3 is inserted into the Tree. The tree contains 2, 1, 3.

Right end point of line segment 1 is processed: 1 is deleted from the Tree. Intersection of 2 and3 is checked. Intersection of 2 and 3 is reported. The tree contains 2, 3. Note that the above pseudocode returns at this point. We can continue from here to report all intersection points.

Left end point of line segment 4 is processed: Intersections of line 4 with lines 2 and 3 are checked. No intersection. 4 is inserted into the Tree. The tree contains 2, 4, 3.

Left end point of line segment 5 is processed: Intersection of 5 with 3 is checked. No intersection. 4 is inserted into the Tree. The tree contains 2, 4, 3, 5.

Right end point of line segment 5 is processed: 5 is deleted from the Tree. The tree contains 2,4, 3.

Right end point of line segment 4 is processed: 4 is deleted from the Tree. The tree contains 2,4, 3. Intersection of 2 with 3 is checked. Intersection of 2 with 3 is reported. The tree contains 2, 3. Note that the intersection of 2 and 3 is reported again. We can

add some logic to check for duplicates.

Right end point of line segment 2 and 3 are processed: Both are deleted from tree and tree becomes empty.

Time Complexity: The first step is sorting which takes O(nLogn) time. The second step process 2n points and for processing every point, it takes O(Logn) time. Therefore, overall time complexity is O(nLogn)

References:

http://www.cs.uiuc.edu/~jeffe/teaching/373/notes/x06-sweepline.pdf
http://courses.csail.mit.edu/6.006/spring11/lectures/lec24.pdf
http://www.youtube.com/watch?v=dePDHVovJIE
http://www.eecs.wsu.edu/~cook/aa/lectures/l25/node10.html