198:529 Computational Geometry

Lecture Notes : Rectilinear Computational Geometry *

Last Update: May 13, 1997

1 INTRODUCTION

The segment tree is a useful data structure for algorithms dealing with objects in rectilinear geometry. VLSI design is a major application area for rectilinear geometry. Intersection of rectangular regions of semiconductor containing varying concentration of holes and electrons define transistors. To be able to accurately control the characteristics of transistors so formed, it is important to be able to control a) the total area of overlap, b) the minimum separation between such intersections and c) the contour of the union of such overlaps. Using the segment tree, efficient algorithms have been developed for these problems.

This notes covers several data structures that are used in rectilinear computational geometry and then looks at the application of one of them (segment trees) in detail. We will also see the use of vertical sweep line to find the contour of the union of a collection of rectangles and also to find intersections of line segments.

2 PRELIMINARIES

In this section, we discuss two subset selection problems. First, to find which points fall into a particular interval. Second, to find which intervals contain a particular point. We present several data structures that will work for these problems. They are range trees, segment trees, interval trees, and treaps.

2.1 Range Queries

2.1.1 1-D Range Trees

problem 1 *Given a set $S \subset \Re$ with $|S| = n$, when queried about a closed interval $I = [l, h]$, we want to decide $S \cap I$, i.e. $\{x \in S | l \leq x \leq h\}$.*

A solution is as follows:

- Maintain a balanced binary search tree for S. Store data in the leaves.
- Augment the tree so that the leaves are linked (Figure 1).

*Scribed by Ming Ouyang, Valerie barr; editted by Peter Hajnal, S. Viswanath

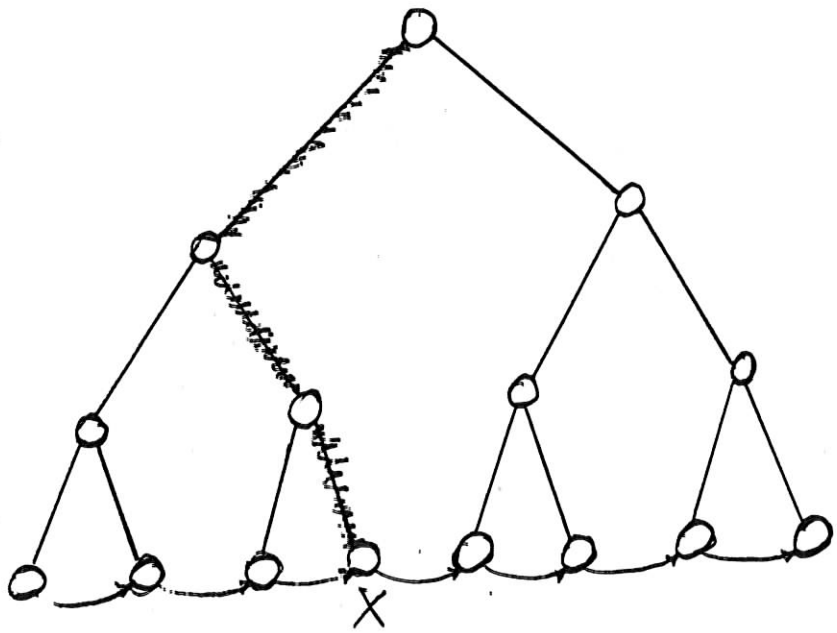


Figure 1: 1-D Range Tree.

- When queried about the interval $[l, h]$, search the tree for the first item x larger than or equal to l.
- If $x \leq h$, report x, and walk along the leaves, reporting all items that are less than or equal to h.
- Stop at the first item which is larger than h.

Analysis:

- Preprocessing $P(n) = O(n \log n)$: We need $O(n \log n)$ time to sort S, and $O(n)$ time to build the tree.
- Querying $Q(n) = O(\log n + A)$: We need $O(\log n)$ time to search the tree. A is the number of items falling in the interval. This algorithm is output sensitive.
- Space $S(n) = O(n)$: Binary search tree.

2.1.2 2-D Problems

problem 2 *Given a set $S \subset \Re^2$ with $|S| = n$, when queried about a closed rectangle $L = [l_x, h_x] \times [l_y, h_y]$, we want to decide $S \cap L$, i.e. $\{(x, y) \in S | l_x \leq x \leq h_x$ and $l_y \leq y \leq h_y\}$.*

A naive solution can be as follows: use two 1-D range trees, one on the x-coordinate, the other on the y-coordinate. When queried about the rectangle $[l_x, h_x] \times [l_y, h_y]$, we first find the two sets $S_1 = \{(x, y) \in S | l_x \leq x \leq h_x\}$ and $S_2 = \{(x, y) \in S | l_y \leq y \leq h_y\}$. Then the answer is their intersection. Analysis:

- Preprocessing $P(n) = O(n \log n)$: $O(n \log n)$ time for each of the range trees.

- Querying $Q(n) = O(\log n + |S_1| + |S_2|)$.
- Space $S(n) = O(n)$.

This algorithm has the drawback that S_1 and S_2 may be large while their intersection is small. In order to avoid this phenomenon, we consider the following algorithm. Assume there are $n-1$ vertical lines separating the n points, together with the lines $x = -\infty$ and $x = +\infty$. We can form a query interval by choosing two of them. Thus there are $C(n+1, 2) = O(n^2)$ possible vertical strips. For each of these strips, we create a 1-D range tree on the y-coordinate for the points in the strip. How do we organize so many range trees (vertical strips)? We put their starting points in an array in order. Each entry in this array stands for a group of strips sharing the same starting point. For each of these groups, we create another array storing their end points in order. When queried about a rectangle, we can do binary search on the first array to locate the left boundary. We then do another binary search on the second array to locate the right boundary. Once we identify which strip is queried, we use the corresponding 1-D range tree on the y-coordinate to report the answer. Analysis:

- Preprocessing $P(n) = O(n^3)$: We sort the points in S on both the x- and y-coordinates using time $O(n \log n)$. For each of the strips ($O(n^2)$ of them), creating a range tree costs $O(n)$ time (points are already sorted).
- Querying $Q(n) = O(\log n + A)$: $O(\log n)$ for binary search on the left and right boundaries. $O(\log n)$ for searching the range tree. A is the number of items in the rectangle.
- Space $S(n) = O(n^3)$: There are $O(n^2)$ range trees, each of size $O(n)$.

The above algorithm is wasteful of preprocessing and space. Suppose we know in advance what the queries will be. We can get a better algorithm. We build a balanced binary tree such that each leaf corresponds to a "standard answer strip". Each interior node corresponds to the larger strip formed by its children. All nodes are associated with a 1-D range tree on the y-coordinate. This is our 2-D range tree (Figure 2).

Note that, for each of the "standard queries", we need to consult at most two nodes at each level of the tree. The left one matches the left portion of the query strip. The right one matches the right portion. The remaining is matched at higher levels. The algorithm is as follows:

- Sort the points on both the x- and y-coordinates. $O(n \log n)$.
- Build a 1-D range tree on the y- coordinate for each of the standard strips. $O(n)$.
- For larger strips, merge the trees of its two children. $O(n \log n)$.
- To answer a query, search the 2-D tree to find the leftmost strip and use the corresponding 1-D tree to report points.
- Back up the 2-D tree and go down to the right to find the next largest strip inside the query interval, and report points. And repeat.
- Once we go beyond the interval, we then go down to the left, and so on.

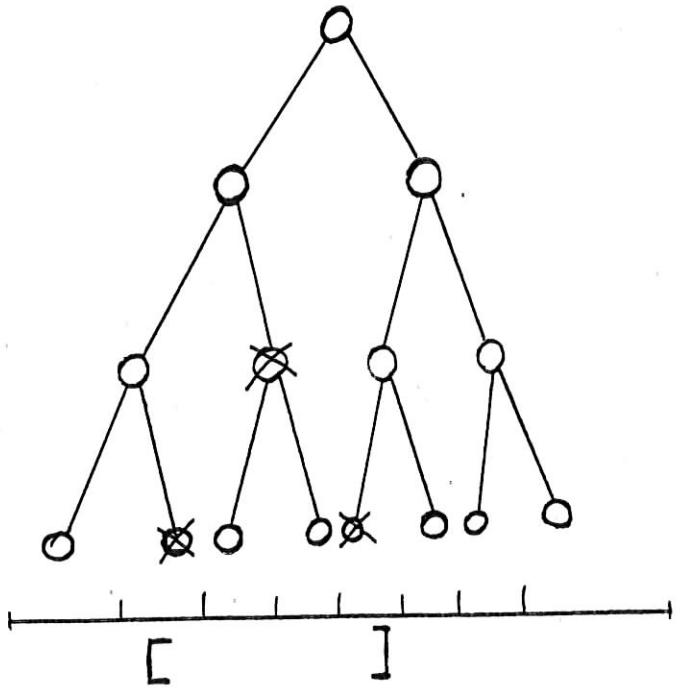


Figure 2: 2-D Range Tree.

- Preprocessing $P(n) = O(n \log n)$.
- Querying $Q(n) = O(\log^2 n + A)$: We will visit $O(\log n)$ nodes of the 2-D tree. At each node we spend $O(\log n)$ time reporting points.
- Space $S(n) = O(n \log n)$: Every point may appear in $O(\log n)$ 1-D range trees.

2.1.3 d-D Range Searching

problem 3 *Given a set $S \subset \Re^d$ with $|S| = n$, when queried about an interval $L = l_1 \times l_2 \times \cdots \times l_d$, decide $S \cap L$.*

Solution: Construct a balanced binary search tree T on x_1-coordinate. For every node in T, associate a secondary (d-1)-D range tree on $x_2, \cdots, x_d$.

- Preprocessing $P(n) = O(n \log^{d-1} n)$: Sorting points takes $O(dn \log n)$ time. And building trees takes $O(n \log^{d-1} n)$ time.
- Querying $Q(n) = O(\log^d n + A)$: By induction.
- Space $S(n) = O(n \log^{d-1} n)$: By induction.
  - $d = 2$: Every point is in $\log n$ 1-D range trees. $S(n)_2 = O(n \log n)$.
  - $d = 3$: Every point is in $\log n$ 2-D range trees. $S(n)_3 = O(n \log^2 n)$.

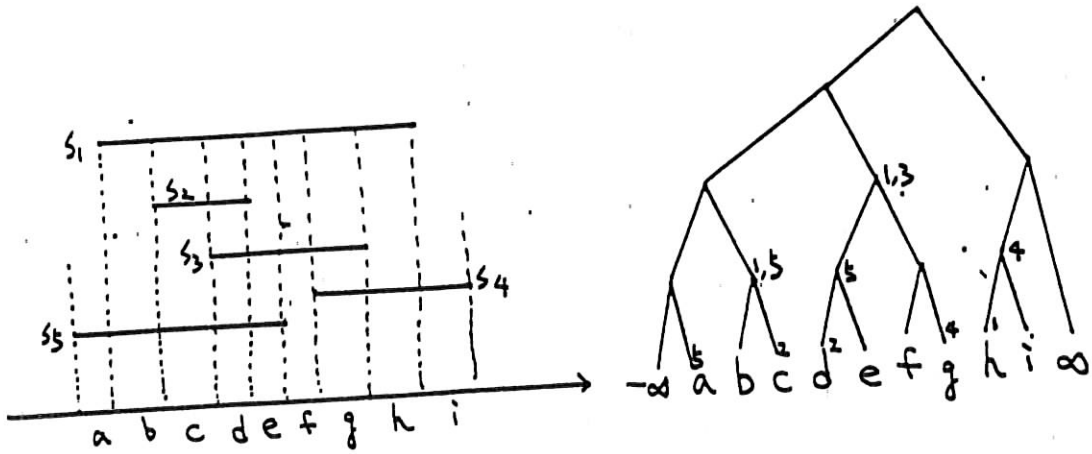


Figure 3: 1-D Segment Tree.

2.2 Segment Trees for Inverse Range Queries

2.2.1 1-D Segment Trees

problem 4 *Given a set* $S = \{[l_i, h_i] | i = 1, \cdots, n\}$, *when queried about a point* $x \in \Re$, *decide the set of segments containing* x, *i.e.* $\{s \in S | x \in s\}$.

There are n segments in S, thus, $2n$ end points. These points break the real number line into $2n + 1$ "atomic intervals". We build a binary tree such that the leaves are atomic intervals, and the interior nodes are "standard intervals" formed by the unions of their children. We further label each node v by a subset T of S so that

- s is in T if s contains the entire interval of v, and
- s does not contain the entire interval of the sibling of v.

If s contains both v and its sibling, s will be associated with (exactly) one of their ancestors. The tree is called a 1-D segment tree (Figure 3). Note that a segment can be associated with at most two nodes at each level and, therefore, $2 \log n$ nodes in total. Because, a segment can be divided into at most three portions with respect to a level of the tree. The left portion, if any, matches a node which is a right-child. The right portion, if any, matches a left-child. The middle portion matches pairs of siblings and should be associated with their ancestors. The algorithm is as follows:

- Sort the end points. $O(n \log n)$. Build the bare tree. $O(n)$.
- Label the tree. $O(\log n)$ each segment. $O(n \log n)$ in total.
- When queried about a point, search the tree and report the segments associated with the nodes along the path.

- Preprocessing $P(n) = O(n \log n)$.
- Querying $Q(n) = O(\log n + A)$.
- Space $S(n) = O(n \log n)$: n segments, each appearing in $O(\log n)$ nodes.

2.2.2 2-D Segment Trees

problem 5 *Given a set* $S = \{[l_{x_i}, h_{x_i}] \times [l_{y_i}, h_{y_i}] | i = 1, \cdots, n\}$, *when queried about a point* $(x, y) \in \Re^2$, *decide the set of rectangles containing* (x, y).

A solution is as follows:

- Construct a 1-D segment tree T on x intervals. Each node in T has a segment tree on y intervals hanging there.
- When queried about a point (x, y), locate x in the primary tree, locate y in the corresponding secondary tree.
- Report the rectangles encountered along the path in the secondary tree.
- Preprocessing $P(n) = O(n \log^2 n)$.
- Querying $Q(n) = O(\log^2 n + A)$.
- Space $S(n) = O(n \log^2 n)$.

2.3 Interval Trees

Interval trees are more space efficient than segment trees when used in 1-D inverse range search. The problem is: given a set of segments S and a query point x, determine the segments containing x. An algorithm is as follows:

- Take the endpoints of the segments in S. They form a multiset. Find their median m.
- Partition S into S_L, S_R, and S_C. S_L (S_R) contains the segments which lie entirely to the left (right) of m. And S_C contains the segments which are "cut" by m (Figure 4).
- Create a node v for m. Its children are the trees recursively defined by S_L and S_R. At v, store two sorted lists. One contains the left endpoints of S_C. The other contains the right endpoints. This gives us the interval tree.
- To answer a query, use the median values stored at each vertex to locate the point as in a binary search tree. At each node, search the list from the outside and report the segments containing it.
- Preprocessing $P(n) = O(n \log n)$: Sorting $O(n \log n)$. Finding the median can be done in linear time.
- Query $Q(n) = O(\log n + A)$.
- Space $S(n) = O(n)$: Each segment is associated with a node only. Compared to the space requirement of segment trees, $O(n \log n)$.

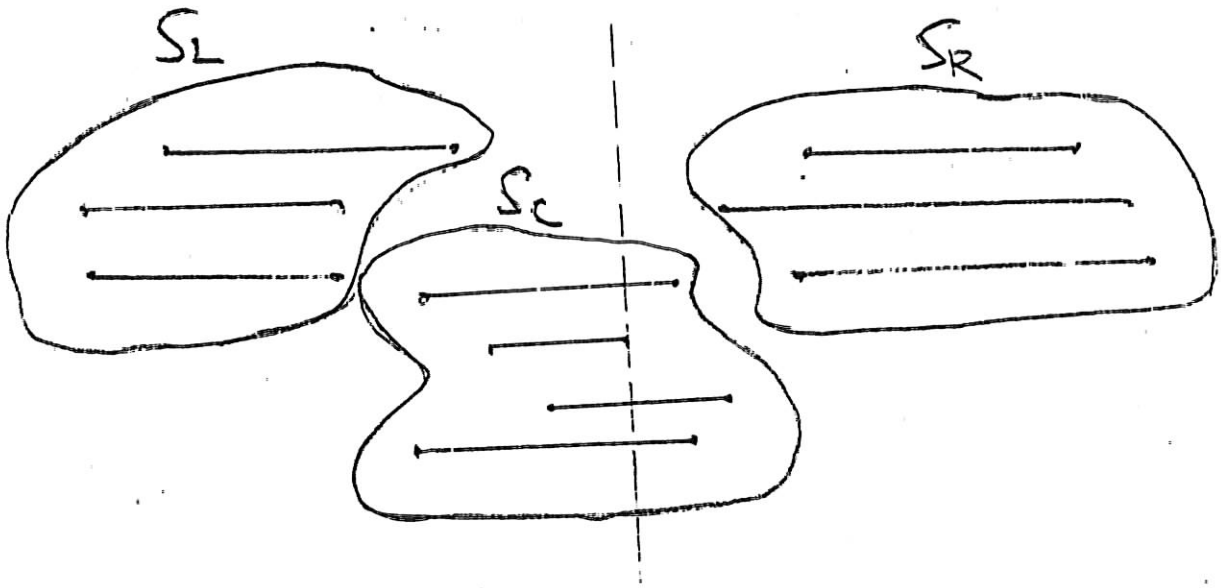


Figure 4: Interval Tree.

2.4 Treaps

In this section we discuss how to use a data structure called "treap" to handle 2-D range query when the query ranges are unbounded rectangles. Assume the rectangles are open upwards, i.e., they are of the form $[x_0, x_1] \times [y_0, \infty]$. Treaps are combinations of trees and heaps. Given a set of points S in the plane, we build a treap for S by (Figure 5):

Recursively,

- If $S = \emptyset$, do nothing.
- Find a point $p \in S$ with the maximum y-coordinate.
- Remove p from S. Find the median m of the x-coordinates of the remaining points.
- Divide S into S_L and S_R.
- Create a vertex v with fields for p and m. Let its children be the treaps for S_L and S_R.
- Preprocessing $P(n) = O(n \log n)$.
- Space $S(n) = O(n)$.

To answer a query,

- Locate the two leaves of the tree that would follow x_0 and precede x_1 by branching based on the median values stored at tree nodes.
- For every node on the path, check whether the node in p field lies in the query range.
- Search all subtrees between the two paths. Report all points stored in the p fields with y-coordinate larger than or equal to y_0.

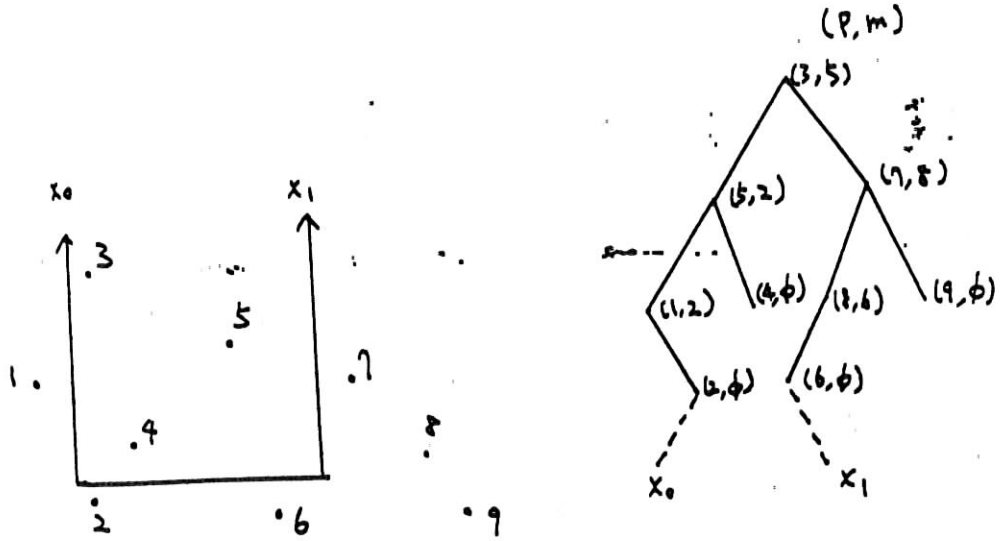


Figure 5: Treap for a set of points.

- By the heap property, we can stop searching a path whenever we hit a point with y-coordinate less than y_0.
- Querying $Q(n) = O(\log n + A)$.

2.4.1 Applications of Treaps

problem 6 *Given a set of intervals S in $\Re$, when queried about an interval $I_0 = [x_0, y_0]$, determine*

1. $\{I \in S | I \cap I_0 \neq \emptyset\}$.
2. $\{I \in S | I_0 \subset I\}$.
3. $\{I \in S | I \subset I_0\}$.

We map the intervals in S to points in the plane and the query interval to an unbounded rectangle so that treaps will help. Specifically, to solve the first problem, note that if $I = [a, b]$ in S intersects $I_0 = [x_0, y_0]$, then I_0 must begin before I ends and I_0 must end after I has begun, i.e., $x_0 \leq b$ and $a \leq y_0$. So the mapping is $I = [a, b] \longrightarrow (b, a)$ and $I_0 = [x_0, y_0] \longrightarrow [x_0, \infty] \times [y_0, -\infty]$ (Figure 6).

As for the second problem, if $I = [a, b]$ contains $I_0 = [x_0, y_0]$, then $y_0 \leq b \leq \infty$ and $a \leq x_0$. So the mapping is $I = [a, b] \longrightarrow (b, a)$ and $I_0 = [x_0, y_0] \longrightarrow [y_0, \infty] \times [x_0, -\infty]$.i.e.,we have to get a region such that all the points to the left of the line $x = y0$ and all the points that are not below the horizontal line, $y = x0$ are ignored. As for the third problem, if $I_0 = [x_0, y_0]$ contains $I = [a, b]$, then $x_0 \leq a \leq y_0$ and $b \leq y_0$. So the mapping is $I = [a, b] \longrightarrow (a, b)$ and $I_0 = [x_0, y_0] \longrightarrow [x_0, y_0] \times [y_0, -\infty]$ i.e., we have to get a region such that all the points to the right of the vertical line, $x = y0$,all the points to the left of the vertical line $x = x0$ and the points that are above the horizontal line, $y = x0$ are ignored.

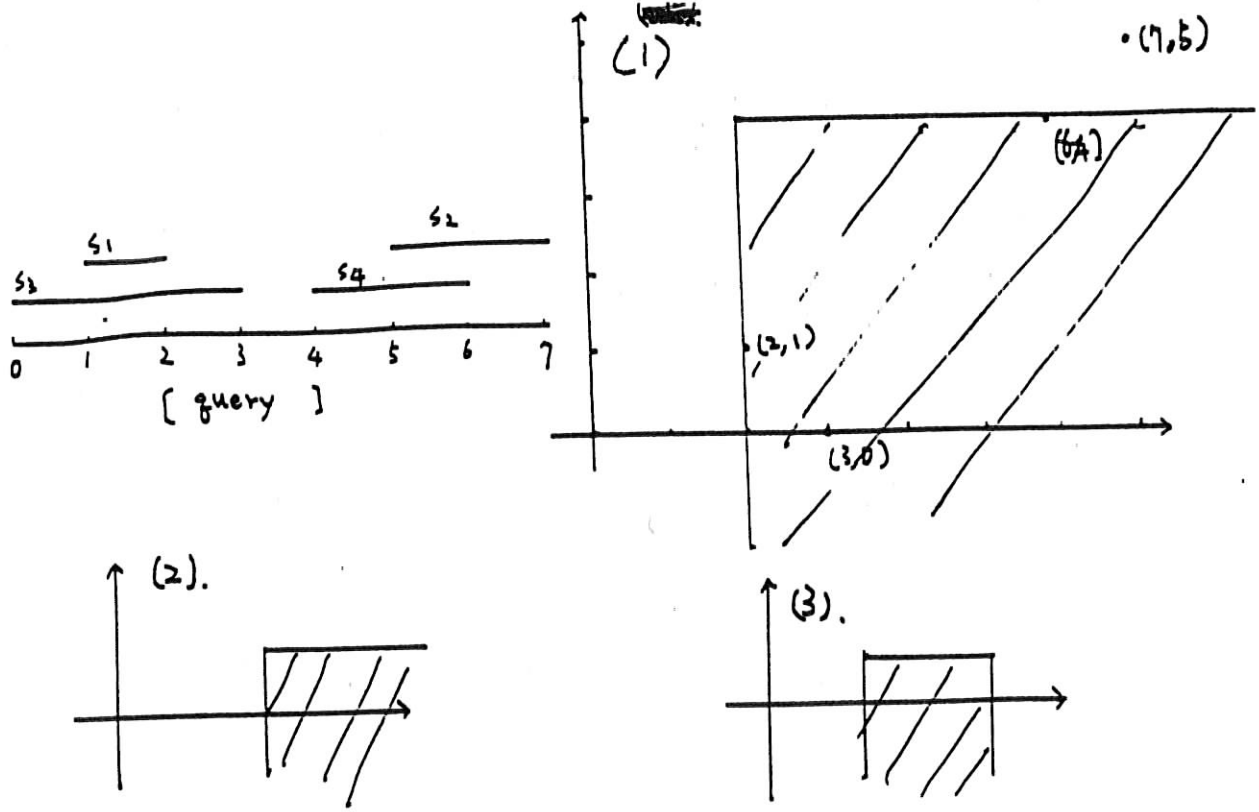


Figure 6: Mappings.

3 APPLICATIONS OF SEGMENT TREE.

In this section we will look at the applications of one of the data structures discussed in the previous section. The segment tree (and its minor variation , the augmented segment tree) finds application in determining the contour of the union of a collection of rectangles and also in finding the intersections of a collection of line segments.

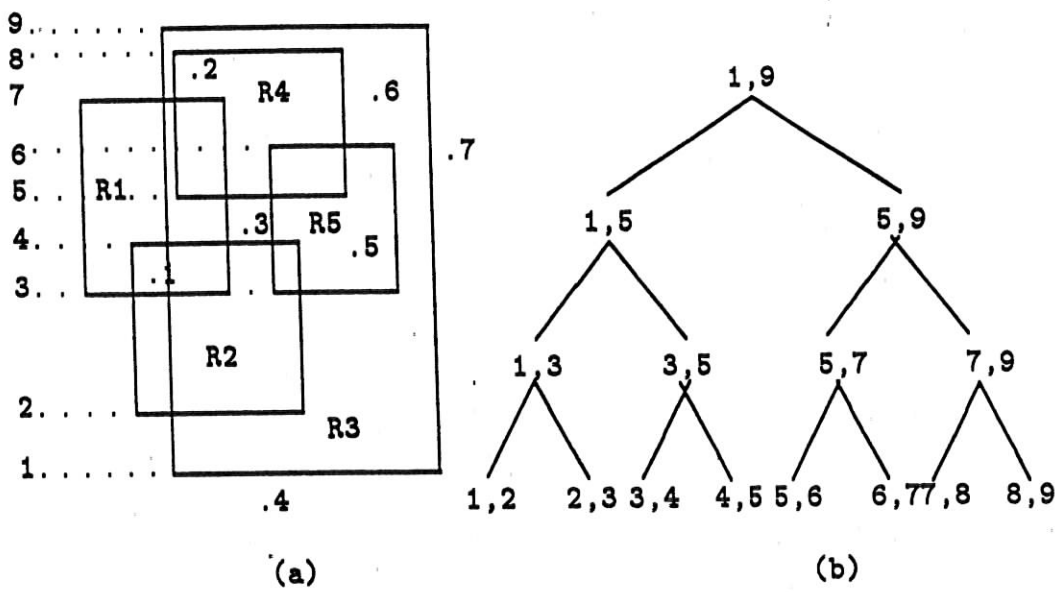


Fig.1: (a) An arrangement of rectangles and (b) its segment-tree

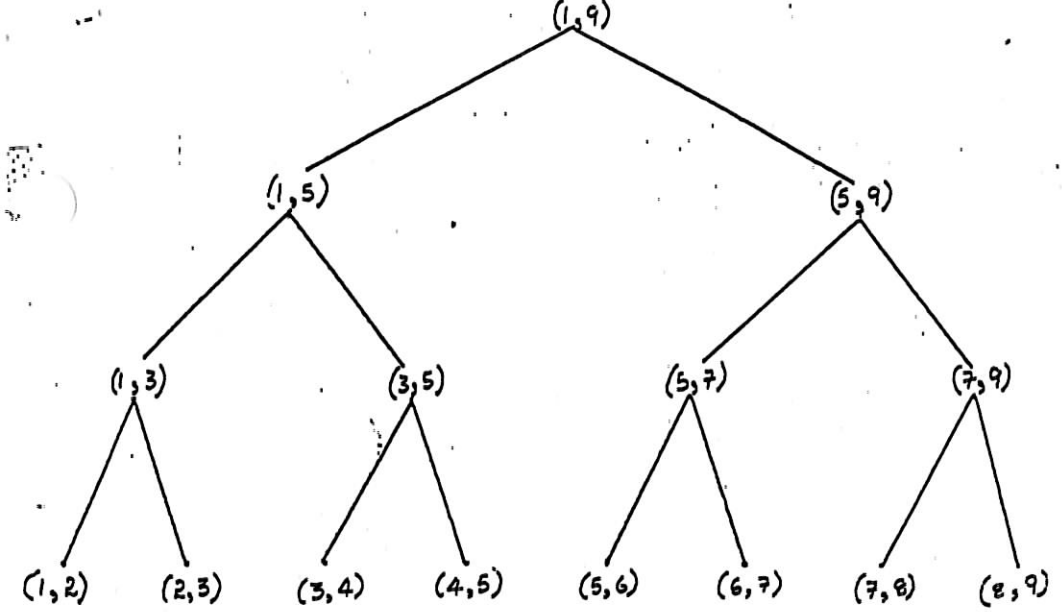


Figure 7: (b):The segment tree corresponding to the figure 7(a).

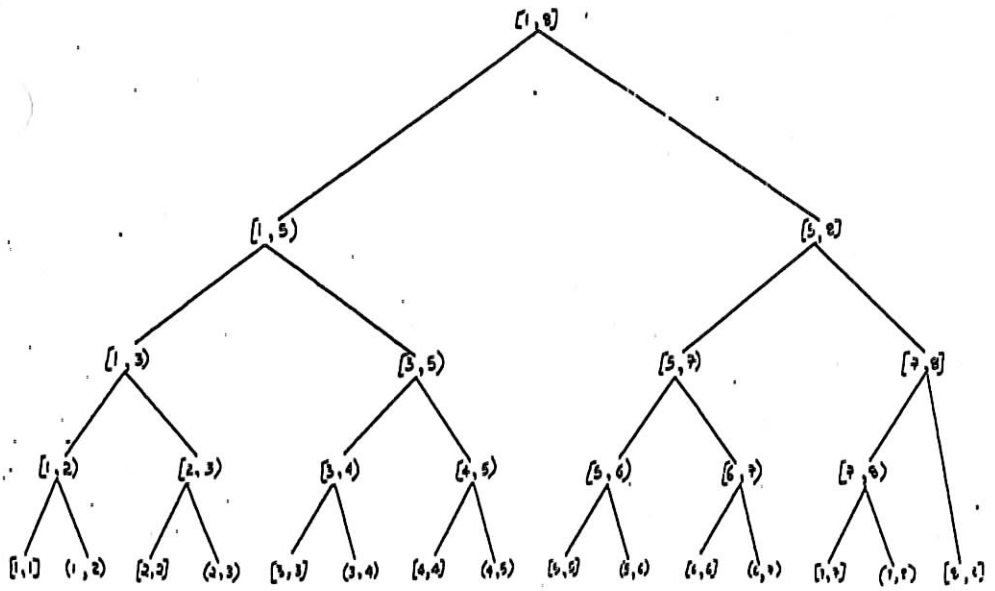


Figure 8: The augmented segment tree for figure 7(a).

PLANE SWEEP

Plane sweep is a powerful tool of computational geometry. We imagine a vertical line L, originally located at $x = -\infty$, and move L continuously across the plane to $x = +\infty$. Different data structures are used in different applications in order to maintain an accurate representation of the current cross-section of the plane at the current position of L. Obviously a continuous sweep of the plane is impossible, so the plane sweep technique is used where updates to the cross section need to be made at only a finite number of x-coordinates. These special x-coordinates, or events, may be kept in an ordered list, if the events are predetermined, or in a dynamic priority queue, if additional events will be computed during the sweep. Consider the rectilinear arrangement of Fig 7 (a). Let us try to do a sweep, using a vertical line, through this arrangement starting from the left. While doing so, we will cross vertical segments of the arrangement. Between two adjacent vertical segments (with respect to the x direction), there is no change in the sweep status. It is only when we reach a vertical segment that a change may occur. Thus, intuitively, it seems sufficient to store only the vertical segments, with their position information.

We formalize the idea now. Given, an arrangement of N rectangles with endpoints, $(x_1, y_1), \ldots (x_{4N}, y_{4N})$.

1. Sort the points on $x_1, \ldots x_{4N}$
2. Sort the points on $y_1, \ldots y_{4N}$
3. Normalize the coordinates to $(1 \ldots n, 1 \ldots m)$, calling the n distinct x-coordinates $norm_{x_i}$ and the m distinct y-coordinates $norm_{y_i}$.

Now, we form a segment tree data structure on the normalized y-coordinate where each leaf represents the interval bounded by adjacent y-coordinates and interior nodes represent the interval formed by the union of the intervals of all leaves of its subtree.

- $B[1 \ldots 2m-3], E[1 \ldots 2m-3]$: where for any vertex v of the segment tree, $(B[v], E[v])$ defines the y interval represented by v. [1]
- $C[1 \ldots 2m-3]$: where $C[v]$ stores the number of edges currently allocated to v. Note that a node v is said to be *locally* covered if $C[v] > 0$ and that it is covered from above, or *upcovered* if some ancestor w of v has $C[w] > 0$.

Essentially, to create a segment tree, begin by selecting the desired y-coordinates, sort them, and eliminate any repetations. For the purpose that we are considering, the desired y-coordinates are the y values of the horizontal edges of the rectangles.

Following is an algorithm for creating the segment tree, given the presorted, normalized points.

$B[r] \leftarrow 1; E[r] \leftarrow n;$
SEG_TREE(r);
define SEG_TREE(u)

segment_vertex u;
begin

[1] This interval may be half-open depending on application.

```
    if (E[u] − B[u]) > 1
    then begin
        create v ← LCHILD[u];
        B[v] ← B[u];
        E[v] ← ⌊(B[u] + E[u])/2⌋;
        SEG_TREE(v);
        create w ← RCHILD[u];
        B[w] ← ⌊(B[u] + E[u])/2⌋;
        E[w] ← E[u];
        SEG_TREE(w);
    end
end;
```

Definition: An interval represented by a leaf is called an elementary interval and one represented by an interior vertex is called a standard interval.

While doing a sweep of the arrangement, we will meet vertical segments which may have to be inserted into or deleted from the segment tree depending on their orientation. Following are methods to do these operations.

Inserting vertical edges

1. The segment tree will hold vertical segments representing rectangles intersected by the current position of the sweep-line.

2. To insert a vertical edge (y_i, y_j), the edge is decomposed into as few standard and elementary intervals as possible.

3. Let w be the leaf node with $B[w] = y_i$; let x be the leaf node with $E[x] = y_i$; let $z = lca(w, x)$, where $lca(x, y)$ returns the lowest common ancestor of x and y. If z itself corresponds to the interval (y_i, y_j), then it is the only node affected by the insertion.

4. Otherwise, (y_i, y_j) is decomposed into subintervals by considering the paths from z to w and from z from x. The corresponding nodes are determined as follows: if $w(x)$ is a left (right) child, choose the closest ancestor $l(r)$, of w which is either a right (left) child or else a child of z; otherwise choose $l = w(r = x)$; choose every node p which does not lie on the path from z to l (z to r) but which is the right child of a node on the path from z to l (z to r). Clearly, at most 2 nodes have been chosen at each level of the tree.

5. For each of the affected nodes v, $C[v]$ must be incremented.

6. For each vertex v, if necessary, a linked list, $\mathcal{L}_v$, may identify the exact edges covering the node. Since each vertical segment is decomposed into at most $2 \log n$ standard and elementary intervals, the space used is $O(n \log n)$ for $O(n)$ edges. [2]

Here is an algorithm for doing an insertion. Bracketed items are optional.

[2] The best case is when each vertical edge is equivalent to a standard interval; needs only $O(n)$ storage

```
define INSERT(b, e; u)
y_coordinate b, e;
segment_vertex u;
begin
      if (b ≤ B[u]) ∧ (E[u] ≤ e)
      then
            { add (b,e) to L_u}
            C[u] + +;
      else begin
            if (b < ⌊(B[u] + E[u])/2⌋)
              then
                    INSERT(b, e; LCHILD(u));
            if (⌊(B[u] + E[u])/2⌋ < e)
              then
                    INSERT(b, e; RCHILD(u));
      end
{     UPDATE(u); Promote coverage }
end;
```

Deleting vertices

The deletion process is comparable:

```
define DELETE(b, e; u)
y_coordinate b, e;
segment_vertex u;
begin
      if (b ≤ B[u]) ∧ (E[u] ≤ e)
      then
            { delete (b,e) from L_u}
            C[u] − −;
      else begin
            if (b < ⌊(B[u] + E[u])/2⌋)
              then
                    { DEMOTE(u); }
                    DELETE(b, e; LCHILD(u));
            if (⌊(B[u] + E[u])/2⌋ < e)
              then
                    DELETE(b, e; RCHILD(u));
      end
      { UPDATE(u); Promote coverage }
end;
```

For reasons of efficiency, the segment tree may need to encode more information. For example, we may want each node to have a flag set to 1 if some node in its subtree has been covered, and set to 0

otherwise, i.e., we could add an extra field P[u] at each node u such that P[u] =1 if some decendent v of u has $C[v] > 0$; else P[u] = 0. For all nodes u in T, C[u] and P[u] must be initialised to 0. Furthermore, if two siblings u, v have $C[u]$, $C[v] > 0$, we may wish to *promote* the coverage to a higher location in the tree: decrement $C[u]$, $C[v]$ and increment $C[PARENT[u]]$. An UPDATE subroutine can be responsible for these additional features. However, if we incorporate PROMOTE, a corresponding DEMOTE routine is necessary in order to perform deletions correctly:

```
define DEMOTE(v)
segment_vertex v;
begin
      if C[v] > 0
      then begin
        C[v] - -;
        C[LCHILD[v] + +;
        C[RCHILD[v] + +;
      end
end;

define PROMOTE(v)
segment_vertex v;
begin
      C[LCHILD[v] - -;
      C[RCHILD[v] - -;
      C[v] + +;
end;
```

3.1 CONTOUR OF A COLLECTION OF RECTANGLES

Given: n rectangles all aligned to y-axis with edges oriented in the counter-clockwise direction, and m query points.

Problem 1: For each of the m points, count the number of rectangles to which it belongs.

Problem 2: For each of the m points, report all rectangles to which it belongs.

Solution for the counting problem

The solution for Problem 1 can be outlined as:

1. Initialize the segment-tree after sorting and normalizing the points: $O(n \log n)$.
2. Order the vertical segments, in sequence, on x-coordinate, on orientation,[3] and on y-coordinate: $O(n \log n)$. This ensures that inserts will precede queries which will precede deletes.
3. Sort, on x, the query points ($O(m \log m)$).

[3] for edges sharing x-coordinate, edges forming left sides of rectangles precede edges forming right sides of rectangles

4. Merge the query point list with that of the vertical segments: $O(m+n)$.[4]

5. Pick each item from this list in sequence.

6. If it is a segment, insert it into the segment tree if it is of downward orientation; delete it otherwise: $(O(\log n))$.

7. If it is a query point p, traverse the segment tree to report the number of segments covering p. This number is equal to the number of rectangles p is inside: $(O(\log n))$.

Thus, this algorithm takes time $O((m+n)\log mn)$.

Solution for the reporting problem

The solution for Problem 2 can be obtained by following the algorithm of Problem 1 but by using a segment tree enhanced with a linked list of covering rectangles $\mathcal{L}_v$ at each vertex v. In this case, a single insert still takes $O(\log n)$ time but a single query might take $\Omega(n)$ time. Furthermore, a delete may take $\Omega(n \log n)$ time: a search takes place at each affected node v to locate the appropriate rectangle in $\mathcal{L}_v$; each $\mathcal{L}_v$ may have length $\Omega(n)$. The time complexity using this approach mushrooms to $O(n^2 \log n + mn + m \log m)$. If we use a binary search tree $\mathcal{T}_v$ instead of $\mathcal{L}_v$, then each insert and delete takes time $O(\log^2 n)$. If a query q_i reports K_i rectangles, then it takes time $O(\log n + K_i)$. Let $K = \sum_{i=1}^{m} K_i$. The entire algorithm runs in $O(n \log^2 n + m \log mn + K)$ time and $O(n \log n + K)$ space.

We can improve this result still more. Use a segment tree with a list at each node of covering rectangles. For each edge have pointers to all positions in which it appears. A pointer will not be to the head of a list, but rather into the body of the list to the position where the edge is. If the lists are doubly linked then edge deletions and insertions can be done easily. Since each edge can appear in $\log n$ nodes deletion will be $O(\log n)$, and the total time is $O((m+n)\log mn + K)$.

Given: an arrangement of iso-oriented rectilinear polygons, all aligned to the y-axis, with n endpoints.

Problem 3: Contour of Union: determine the boundary of the union of these n polygons.

Two algorithm are discussed in these notes for the contour problem. Namely, the Lipski-Preparata algorithm and the Wood's algorithm. The Lipski-Preparata algorithm has a time complexity of $\mathcal{O}(nlogn + plogn^2/p)$ and a space complexity of $\mathcal{O}(n+p)$ (p is the number of edges on the final contour). The Wood's algorithm is more general in the sense that the input can be a collection of rectilinear polygons rather then plane rectangles (as in the defination of the problem). This algorithm has the same space complexity but a better time complexity ($\mathcal{O}(nlogn + p)$).

Both the algorithms share two key features:
(1)Once the edges in a single direction have been determined, the others can be determined in linear time.
(2)The vertical direction has to be considered differently.
The vertical edges must be sorted lexicographically on:
(a) x-coordinate.
(b) Orientation ("Left" or "Right").

[4]Place a query point between the left edges and the right edges which share the same x coordinate.

(c) Bottom y-coordinate.
They differ in the generation of vertical edges (LP) or horizontal edges (Wood), though in the latter case the vertical edges are generated in the second pass.

Modified Lipski-Preparata algorithm

This algorithm sweeps a vertical line across the plane, stopping only at x-coordinates of the end-points of the input edges and keeping a representation of the current cross-section in a segment tree. The key to this algorithm is that the cross-section of the union of the polygons(contour) remains unchanged throughout the interval between adjacent stopping points. The algorithm can be described in three steps:

Step 1: Initialize a segment tree T on the interval determined by sorting and normalizing the coordinates of end points of the arrangement. The fields in each vertex v of T are: B[v], E[v] and C[v] as described earlier and P[v], where P[v] = 0 if no segment is covered in the subtree rooted at v and 1 if some segment is covered in the subtree.

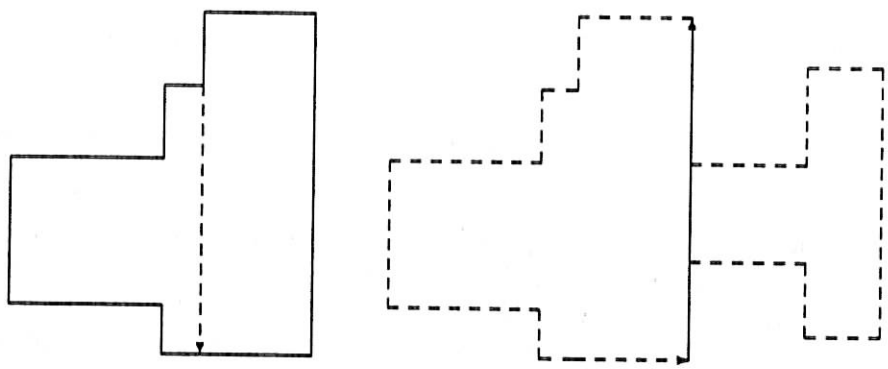
Step 2: Sort the vertical segments on x, on orientation [5]and on y in order. Then sweep across the arrangement as described below to report the vertical edges of the contour.

Step 3: After the sweep, determine the horizontal parts of the contour as follows:

i). Sort the end points of the vertical segments of the contour on y and on x in order to obtain a list $p_1, p_2, \ldots p_{2m}$.

ii). $\forall k$, join p_{2k-1}, p_{2k}.

iii). $\forall k$, if p_{2k-1} is the bottom endpoint of a right edge then orientation of p_{2k-1}, p_{2k} is right, otherwise left.



The Sweep: The plane sweep computes the vertical segments of the union contour. Define LEFT as the left side of a rectangle and RIGHT as the right side of a rectangle. The algorithm proceeds as follows:

[5]edges are oriented so that the interior of the rectangle is to its left. The first vertical edge of the rectangle that is encountered is the "Left" edge and the next one is the "Right" edge.

1. If the current segment is LEFT, insert corresponding interval into T. The contribution of this vertical segment to the contour is exactly the union of those elementary intervals whose nodes were neither locally covered nor *upcovered* before the insertion but are covered after the insertion.

2. If the current segment is RIGHT, delete the corresponding interval from T. The contribution of this segment to the contour is the union of the elementary intervals whose nodes were either covered locally or upcovered prior to the deletion and now are not covered.

We have the following arrays: x[i], ybot[i], ytop[i], orient[i], $\forall$ vertical segments i. Following is the algorithm for the contour of union. Note that we defined $INSERT(b, e; v)$ and $DELETE(b, e; v)$ earlier.

```
define CONTOUR_OF_UNION()
begin
      A ← φ; { holds vertical edges of contour }
      for i ← 1 to 2N do
      begin
            STACK ⇐ Λ;{holds vertical segments of contour}
            if orient[i] = LEFT then begin
              CONTR(ybot[i], ytop[i]; ROOT(T));
              INSERT(ybot[i], ytop[i]; ROOT(T));
            end
            else begin
              DELETE(ybot[i], ytop[i]; ROOT(T));
              CONTR(ybot[i], ytop[i]; ROOT(T));
            end;
            if (orient[i] ≠ orient[i + 1]) ∨ (X[i] ≠ X[i + 1])
            then
              A ← STACK ∪ A;
              { x[i], orient[i] markers are included }
            end
      end
end;
define CONTR(b, e; v)
y_coordinate b, e;
segment_vertex v;
begin
      if (C[v] = 0) then begin
        if (b ≤ B[v]) ∧ (E[v] ≤ e) ∧ (P[v] = 0)
        then begin
              if (B[v] = top(STACK) then
                delete STACK_TOP
              else begin
                STACK ⇐ B[v];
                STACK ⇐ E[v];
```

```
            end;
        end
        else begin
            if b < ⌊(B[v] + E[v])/2⌋ then
              CONTR(b, e; LCHILD[v]);
            if ⌊(B[v] + E[v])/2⌋ < e then
              CONTR(b, e; RCHILD[v]);
        end
    end
end;
define UPDATE(v)
segment_vertex v;
begin
    if (LCHILD = Λ) then
      P[v] = 0;
    else begin
        if (C[LCHILD[v]] > 0) ∧ (C[RCHILD[v]] > 0)) then
          PROMOTE(v);
        if (C[LCHILD[v]] = P[LCHILD[v]] = C[RCHILD[v]] = P[RCHILD[v]])
          then P[v] ← 0;
          else P[v] ← 1;
    end
end;

define PROMOTE(v)
segment_vertex v;
begin
    C[LCHILD[v]] − −;
    C[RCHILD[v]] − −;
    C[v] + +;
end;

define DEMOTE(v)
segment_vertex v;
begin
    C[LCHILD[v]] + +;
    C[RCHILD[v]] + +;
    C[v] − −;
    P[v] ← 1;
end;
```

Time Complexity: We note that for each vertical segment processed, there is a unique leftmost and rightmost fragment. (1) If a vertex v in T with $P[v] = 0$ and $C[v] = 0$ is encountered by CONTR(b, e; v) then the entire segment $(B[v], E[v])$ contributes to the contour provided that $b \leq B[v]$ and $E[v] \leq e$; (2) if $C[v] > 0$, no part of the interval $(B[v], E[v])$ on this segment

contributes to the contour; (3) otherwise CONTR does a preorder traversal of the subtree delimited by the paths from v to u and from v to w, where u is the vertex in T corresponding to the leftmost fragment and w to the rightmost fragment. In the traversal, CONTR does not visit the subtree of any node satisfying rules (1) and (2); these are end nodes. The total work is proportional to the total length of this traversal. Note that every other end node produces an edge of the contour.

Theorem 1 (Lipski–Preparata '80) *If there are* v *end-nodes in a subtree of a binary tree containing n end-nodes, the total path length of the subtree is:*

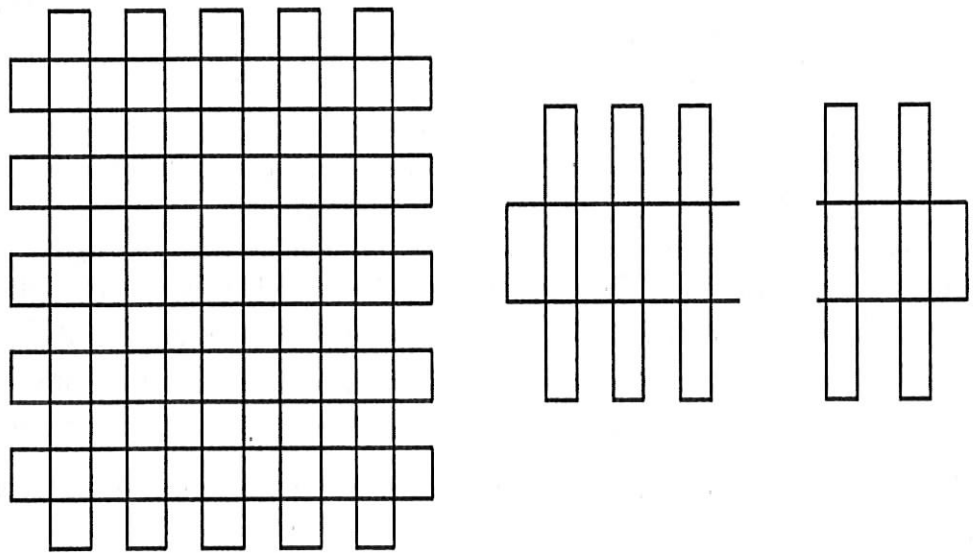
$$O(v \log \frac{16n}{v})$$

Let n_i be the number of disjoint pieces in $S_i \cap \overline{T}$, where S_i is the ith vertical edge and $\overline{T}$ is the complement of all the segments in T.

Therefore, the total work done in part 2 is:

$$C \sum_{i=1}^{2n} (n_i \log \frac{16n}{n_i}) < Cp \log \frac{32n^2}{p}$$

where p is the total number of contour edges and C is a constant. Combining the presorting of part 1 and the sorting of part 3, we get an overall time complexity of $O(n \log n + p \log(n^2/p))$.



Upper and lower bounds for the contour union problem

Upper bound: Consider Fig. 10(a). As a sweep is made through this arrangement, every insert will cause $O(n)$ regions to be reported. Every delete will again cause $O(n)$ regions to be reported. Thus, the total time required is $O(n^2)$.

Lower bound: We map this problem to the sorting problem as shown in Fig. 10(b). Given a list $x_1, x_2, \ldots, x_n$, we transform it into a series of rectangles R_i with x-coordinate *near* x_i. We

also have a long, flat rectangle R_f which intersects each R_i on its left and right side. Clearly, the contour of the union of these rectangles gives us the x_i's in sorted order. Thus, we have established a lower bound of $\Omega(n \log n + p)$, where p is the number of edges in the contour.

Wood's algorithm

Given: a collection of n rectilinear polygons aligned with the y-axis.

Find: the contour of the union of the polygons.

This algorithm also sweeps a vertical line across the arrangement of polygons. However, unlike Lipski-Preparata's algorithm, this algorithm generates the horizontal edges during the sweep and runs a second pass to generate the vertical edges. To achieve this, it uses the segment tree differently [6] : it marks off vertical intervals which are currently blocked and keeps track of horizontal edges that are currently active. Horizontal edges are represented as points in the segment tree. The algorithm proceeds as follows:

Step 1: Sort the horizontal edges on y and normalize them. Let $y_1, y_2, \ldots, y_m$ be the y-coordinates of the sorted points. Build the augmented segment tree from this data as follows:

1. $\forall i$, create an external node corresponding to the closed interval $[y_i, y_i]$. Each external node of this type maintains the following information:
   - B[v], E[v] as described earlier,
   - H[v]: the number of times the associated y-value has been inserted and not deleted,
   - V[v]: visible list - gives us a linked list of current running horizontal edges. If $H[v] > 0$ then $V[v] =< y, xorg, next >$ else liV[v] $= \phi$.
2. $\forall i$, create an external node corresponding to the closed interval $[y_i + \epsilon, y_{i+1} - \epsilon]$, where ϵ is chosen to be very small.[7] These nodes have fields B[v], E[v], C[v], as described earlier.
3. The internal nodes have fields B[v], E[v], C[v] as well as two new fields FV[v], LV[v]:
   - FV[v]: the first external node visible in v's subtree if $C[v] > 0$ or ϕ if $C[v] = 0$.
   - LV[v]: the last external node visible in v's subtree if $C[v] > 0$ or ϕ if $C[v] = 0$.

Step 2: Sort the vertical edges on x, on orientation, and on bottom y-coordinate in order and normalize.

Step 3: Separate each vertical edge into an open interval (mapped to the closed interval described above) and two endpoints. Attach a flag to each endpoint specifying whether it originates or terminates a horizontal edge.

The Sweep: Execute the sweep inserting and deleting the open segments as before and adding new procedures for the points so that the following invariants are maintained:

> Given any position of the sweep line, each horizontal edge intersected by the sweep line is recorded in the data structure by incrementing the H field in the

[6] the modified segement tree is called the augmented segmented segment tree.
[7] This is to avoid creating open intervals (y_i, y_{i+1}) which messes up the implementation.

external node v corresponding to the y coordinate of the edge. If $H[v] > 0$ then V[v] is not ϕ. Furthermore, at any internal node v in the tree for which $C[v] = 0$, the chain:

$$V[FV[v]].y, V[V[FV[v]].next].y, \ldots, V[LV[v]].y$$

describes the y-positions of those horizontal edges which would lie on the boundary of the union of rectangles if the interval associated with v were not covered higher in the tree. When processing a node v, the variable *upcover* has value 1 iff there exists an ancestor w of v for which $C[w] > 0$.

Because of the pre-sorting, inserts precede deletes in this algorithm.

1. For each vertical segment inserted at some vertex v, if the segment is fully covered by the interval and if the vertex is not *upcovered* then enqueue all the visible horizontal edges extending from *xorg* to the current x, for output later as part of the contour. If a segment is not fully covered by the interval corresponding to a vertex, call it upcovered and break it into elementary and standard intervals. Also, promote the coverage up the tree in post order while merging visible lists of siblings which are upcovered.

2. While deleting at vertex v, if a segment is fully covered by the interval and is not upcovered, then the horizontal edges can now contribute to the contour only from the current x position. So, we update the *xorg* field in the visible list to *curx*. If the segment is not fully covered then split the visible list into two; one for LCHILD[v] and the other for RCHILD[v] and continue down the tree. Also, demote the coverage to the children in pre order.

3. After inserting/deleting a segment, its endpoints are opened or closed. If an endpoint initiates a horizontal edge, we open it otherwise we close it. By opening, we mean that we add to the visible list of the vertex covering the vertical segment the horizontal edge corresponding to that endpoint to the segment. By closing, we mean that we output to the stack all horizontal edges from *xorg* for the vertex to *curx* and make the visible list empty.

```
define CONTOUR_OF_UNION()
begin
    A ← Φ; { queue for contour horizontal edges }
    for i ← 1 until 2N do
    begin
        if (O[i]) then
          INSERT(Y1[i], Y2[i]; root(T); 0);
        else DELETE(Y1[i], Y2[i]; root(T); 0);
        if (O1[i]) then
          OPEN(Y1[i]; root(T));
        else CLOSE(Y1[i]; root(T),upcover);
        if (O2[i]) then
          OPEN(Y2[i]; root(T));
        else CLOSE(Y2[i]; root(T),upcover);
    end
end
```

```
define INSERT(b, e; v; upcover)
y_coordinate b, e;
segment_vertex v;
boolean upcover;
begin
      if (b ≤ B[v]) ∧ (E[v] ≤ e) then
      begin
            C[v] + +;
            if (C[v] == 1) then
            begin
              if (!upcover) then
              begin
                { trace from FV[v] to LV[v], enqueueing
                  horizontal edges at each y extending
                  from xorg to the current x }
              end
            end
      end
      else
      begin
            upcover ← upcover ∨ C[v];
            if (b < ⌊(B[v] + E[v])/2⌋)
            then
              INSERT(b, e; LCHILD[v]; upcover);
            if (⌊(B[v] + E[v])/2⌋ < e)
            then
              INSERT(b, e; RCHILD[v]; upcover);
      end;
      UPDATE(v);
end

define DELETE(b, e; v; upcover)
y_coordinate b, e;
segment_vertex v;
boolean upcover;
begin
      if (b ≤ B[v]) ∧ (E[v] ≤ e) then
      begin
            C[v] − −;
            if (C[v] == 0) ∧ (!upcover) then
            begin
              UPDATE(v);
              if (!upcover) then
              begin
                { trace from FV[v] to LV[v], setting xorg
```

```
                in each case to the current x-value }
            end
        end
    end
    else
    begin
        if (C[v] > 0) then
          DEMOTE(v);
        upcover ← upcover ∨ C[v];
        { Split VISIBLE[v] into VISIBLE[LCHILD[v]] and
          VISIBLE[RCHILD[v]]; in other words, delete the
          pointer from V[LV[LCHILD[v]]] to V[FV[RCHILD[v]]] }
        if (b < ⌊(B[v] + E[v])/2⌋)
        then
          DELETE(b, e; LCHILD(v); upcover);
        if (⌊(B[v] + E[v])/2⌋ < e)
        then
          DELETE(b, e; RCHILD(v); upcover);
        UPDATE(v);
    end
end

define UPDATE(v)
segment_vertex v;
begin
    if (LCHILD[v] ≠ Λ) then
    begin
        if (C[LCHILD[v]] > 0) ∧ (C[RCHILD[v]] > 0) then
          PROMOTE(v);
        if (C[v] == 0) then
        begin
          { reform VISIBLE[v] by merging V[LCHILD[v]] and
            V[RCHILD[v]]; in other words, make the next field
            in V[LV[LCHILD[v]]] point to V[FV[RCHILD[v]]] and
            set FV[v] ← FV[LCHILD[v]], LV[v] ← LV[RCHILD[v]] }
        end
        else
          { set VISIBLE[v] ← Φ, i.e. FV[v] ← LV[v] ← Φ; }
    end
end

define OPEN(p; v)
horizontal_point p;
segment_vertex v;
begin
```

```
        if (p == v) then
        begin
              H[v] + +;
              if (H[v] == 1) then
                V[v] ←< p, xcur, Φ >;
        end
        else
        begin
              if (p < v) then
                OPEN(p; LCHILD[v]);
              else OPEN(p; RCHILD[v]);
              UPDATE(v);
        end
end;

define CLOSE(p; v,upcover)
horizontal_point p;
segment_vertex v;
begin
        if (p == v) then
        begin
              upcover ← upcover ∨ C[v] ;
              H[v] − −;
              if (H[v] == 0) then
              begin
                if (!upcover) then
                begin
                  { enqueue horizontal edges from xorg to curx; }
                  V[v] ←< Φ, Φ, Φ >;
                end
              end
        end
        else
        begin
              upcover ← upcover ∨ C[v];
              if (p < v) then
                CLOSE(p; LCHILD[v],upcover);
              else CLOSE(p; RCHILD[v],upcover);
              UPDATE(v);
        end
end;

define PROMOTE(v)
segment_vertex v;
begin
```

```
        C[LCHILD[v]] - -;
        C[RCHILD[v]] - -;
        C[v] + +;
        if (C[LCHILD[v]] == 0) then
        begin
              { reform V[LCHILD[v]] by merging V[LCHILD[LCHILD[v]]]
                and V[RCHILD[LCHILD[v]]] }
        end
        if (C[RCHILD[v]] == 0) then
        begin
              { reform V[RCHILD[v]] by merging V[LCHILD[RCHILD[v]]]
                and V[RCHILD[RCHILD[v]]] }
        end
end

define DEMOTE(v)
segment_vertex v;
begin
        if (C[LCHILD[v]] == 0) then
              V[LCHILD[v] ← Φ;
              { i.e. FV[LCHILD[v]] ← LV[LCHILD[v]] ← Φ; }
        if (C[RCHILD[v]] == 0) then
              V[RCHILD[v] ← Φ;
              { i.e. FV[RCHILD[v]] ← LV[RCHILD[v]] ← Φ; }
        C[LCHILD[v]] + +;
        C[RCHILD[v]] + +;
        C[v] - -;
end
```

3.2 COMPUTING INTERSECTIONS.

GIVEN: n line segments, specified by their endpoints.
FIND: Any and all intersection points.

Naive algorithm : For every pair of line segments, compute an intersection point, if one exists. The time complexity of this algorithm is $\mathcal{O}(n^2)$ (even if no two line segments intersect). The following algorithm finds the intersections of a set of n line segments using a vertical line sweep. It runs in $\mathcal{O}(n^2 \log n)$ time and requires $\mathcal{O}(n^2)$ space in the worst case, where there are $\mathcal{O}(n^2)$ intersections.

Data structures.

A *priority queue*, Q, whose entries may include:

- The endpoints of all n line segments.

```
        C[LCHILD[v]] - -;
        C[RCHILD[v]] - -;
        C[v] + +;
        if (C[LCHILD[v]] == 0) then
        begin
              { reform V[LCHILD[v]] by merging V[LCHILD[LCHILD[v]]]
                and V[RCHILD[LCHILD[v]]] }
        end
        if (C[RCHILD[v]] == 0) then
        begin
              { reform V[RCHILD[v]] by merging V[LCHILD[RCHILD[v]]]
                and V[RCHILD[RCHILD[v]]] }
        end
end

define DEMOTE(v)
segment_vertex v;
begin
        if (C[LCHILD[v]] == 0) then
              V[LCHILD[v] ← Φ;
              { i.e. FV[LCHILD[v]] ← LV[LCHILD[v]] ← Φ; }
        if (C[RCHILD[v]] == 0) then
              V[RCHILD[v] ← Φ;
              { i.e. FV[RCHILD[v]] ← LV[RCHILD[v]] ← Φ; }
        C[LCHILD[v]] + +;
        C[RCHILD[v]] + +;
        C[v] - -;
end
```

3.2 COMPUTING INTERSECTIONS.

GIVEN: n line segments, specified by their endpoints.
FIND: Any and all intersection points.

Naive algorithm : For every pair of line segments, compute an intersection point, if one exists. The time complexity of this algorithm is $\mathcal{O}(n^2)$ (even if no two line segments intersect). The following algorithm finds the intersections of a set of n line segments using a vertical line sweep. It runs in $\mathcal{O}(n^2 \log n)$ time and requires $\mathcal{O}(n^2)$ space in the worst case, where there are $\mathcal{O}(n^2)$ intersections.

Data structures.

A *priority queue*, Q, whose entries may include:

- The endpoints of all n line segments.

- One or two intersection points on each line under consideration.
- For any endpoint, a pointer to the opposite endpoint in the line is included.
- Each point in Q has a pointer to the (bottommost) corresponding segment in T.

Q is prioritized in order of X coordinates of the entries. It initially contains all the n endpoints. Can we say anything about the size of Q? Immediately before the intersection of two segments p and q , p and q must be adjacent in T. Thus Q will contain all endpoints not yet processed and any intersection points of segments currently adjacent on the sweepline. Therefore, the number of elements in Q is less than or equal to $2n+n$-1 ,i.e., the size of Q is linear in n.

A *balanced binary search tree*, T, whose nodes represent the active line segments at the current position of the sweep line. The node corresponding to a segment s includes:

- The equation of the line containing s.
- A pointer to the position of the right endpoint of s in Q.
- A pointer to up to 2 intersection points currently stored in Q which lie on s.

The position of a segment in T depends on the y-coordinate of the intersection of s with the last fixed position of the sweep line. T is initially empty.

Algorithm.

1. Pop x from the priority queue.
2. If x is a left endpoint:
   (a) Locate x in T between σ_i and σ_{i+1}.
   (b) Delete any $\sigma_i \cap \sigma_{i+1}$ from Q.
   (c) Insert σ_x into T.
   (d) If σ_x intersects σ_i at p_i:
      i. Insert p_i into Q.
   (e) If σ_x intersects σ_{i+1} at p_{i+1},
      i. Insert p_{i+1} into Q.
3. If x is a right endpoint of σ_i,
   (a) If σ_{i-1} intersects σ_{i+1} at p,
      i. Put p into Q.
      ii. Delete σ_i.
4. If x is an intersection point of σ_i, σ_{i+1},
   (a) Output $x, \sigma_i, \sigma_{i+1}$.
   (b) Delete any intersection points of $\sigma_{i+1}, \sigma_{i+2}$, and of σ_i, σ_{i-1}.
   (c) Insert intersection points of σ_i, σ_{i+2}, and $\sigma_{i+1}, \sigma_{i-1}$.

(d) Swap σ_i, σ_{i+1}.

Swapping can be made easier if every line segment has pointers to the segments above and below itself on the sweep line. This allows the nodes in T to be interchanged in constant time when lines are swapped. However, when this occurs, the actions on Q still take $\mathcal{O}(\log n)$ time.

Complexity.

The time complexity is $\mathcal{O}((n+k)\log n)$ for the above algorithm, where k represents the number of intersections reported. The space complexity is $\mathcal{O}(n)$ for the algorithm itself, but is $\mathcal{O}(n+k)$ if we store the intersection points.[8]

3.3 Triangulation.

If we need to triangulate a planar graph, monotone polygons are almost as good as convex polygons, if we know the direction of monotonicity. If we have a monotone polygon, we can triangulate it in linear time. If we do not have a monotone polygon, then we would like to add edges which will make it monotone. We can go from an arbitrary planar subdivision to a monotone subdivision in $\mathcal{O}(n \log n)$ time. Begin by decomposing the diagram into temporary *trapezoids* using a sweep line. The parallel sides of the trapezoids formed by this algorithm are vertical segments which join each stopping point of the sweep line to the edge directly above it and the edge directly below it. The trapezoids may be degenerate (triangles.) Total time to create these temporary trapezoids is $\mathcal{O}(n \log n)$.

Forming monotone regions.

The temporary trapezoids are used to determine a collection of edges which may be added to the graph to ensure that each face is a *monotone polygon*. Each trapezoid which contains an original vertex in the interior of an edge signifies a region which is not monotone in the x-direction. Add an edge between that vertex and another original vertex on the same trapezoid. (See Figure 11).

Triangulating a monotone polygon.

We triangulate a Y-monotone polygon P by walking through its vertex list in decreasing order of Y-coordinate, stacking its points and creating edges between them. Let $p_1, p_2, \ldots p_k, p_{k+1}, \ldots p_n$, represent the vertices of P in counterclockwise order around P such that p_k is the vertex with minimum Y coordinate and p_1 has the greatest Y value.

The algorithm maintains the following invariants on the stack:

- Items on the stack are in decreasing order of Y coordinate and form a chain that is on the boundary of the polygon.
- Internal angles along this chain are all $\geq 180°$.
- The next vertex to process is adjacent[9] to either x_t (top of stack) or $x_i, i \neq t$.

[8] This algorithm is not optimal. Chazelle and Edelsbrunner recently presented an optimal algorithm using $\mathcal{O}(n \log n + k)$ time and $\mathcal{O}(n+k)$ space.

[9] Adjacent vertices are those vertices already connected by either an edge of P or by an edge of the triangulation.

To triangulate the polygon, we walk around the polygon in decreasing order of Y coordinate. While the internal angle formed by x_t, x_{t-1}, and p_i is $> 180°$, we push p_i onto the stack. If the internal angle is $< 180°$, we can create legitimate triangles using p_i and some of the points on the stack, but not necessarily all of them.

Algorithm.

See Figure 12.

1. Sort $p_1, p_2, \ldots p_n$ by Y coordinate to produce $q_1, q_2, \ldots q_n$. This can be done efficiently by merging the lists $p_1, p_2, \ldots p_k$ and $p_n, p_{n-1}, \ldots p_{k+1}$.
2. $i \leftarrow 3$.
3. $x_1 \leftarrow q_1$.
4. $x_2 \leftarrow q_2$.
5. Read x (the next element in the q list).
6. While $i \leq n$ do:
   (a) if q_i is adjacent to x_i and not adjacent to x_t, *(Case 1)*
      i. Add diagonals $(q_i, x_2), (q_i, x_3), \ldots (q_i, x_t)$.
      ii. Replace the stack by x_t, q_i.
   (b) else if q_i is adjacent to x_t and not adjacent to x_1, *(Case 2)*
      i. Repeat until $t = 1$ or $\angle x_t \geq 180°$:
         A. Add the diagonal (x_{t+1}, q_i).
         B. Delete x_t.
         C. $t \leftarrow t + 1$.
      ii. Add q_i to the stack.
   (c) else if q_i is adjacent to both x_i and x_t: *(Case 3)*
      i. Add the diagonals $(q_i, x_1), (q_i, x_2), \ldots (q_i, x_{t-1})$.
   (d) $i \leftarrow i + 1$.
   (e) end while.

The time for this algorithm is $\mathcal{O}(n)$.

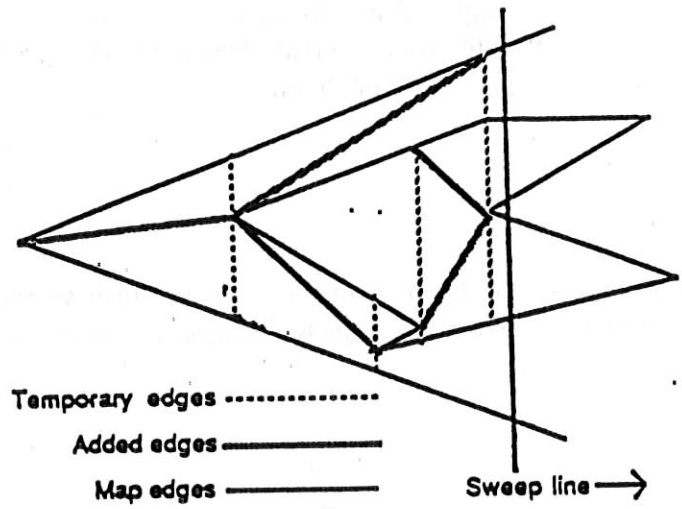


Figure 4: Forming monotone regions.

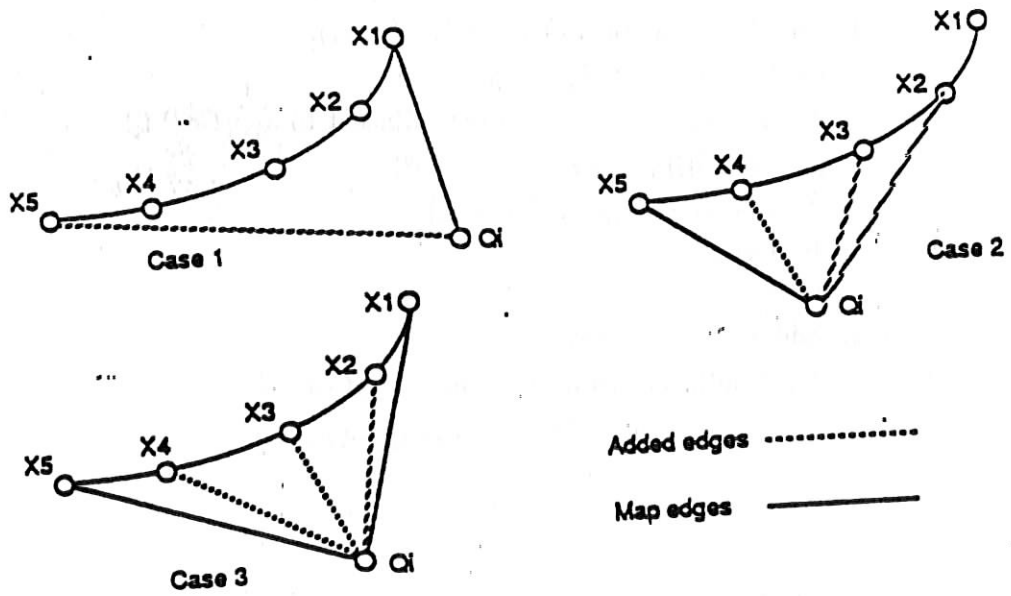


Figure 5: Triangulating a monotone polygon.