Week 7 Lecture Notes

Topics: Fixed universe successor
Van Emde Boas data structure
Amortized analysis
Computation Geometry

Fixed-universe successor problem

Goal: Maintain a dynamic subset S of size n of the universe $U = \{0, 1, \ldots, u-1\}$ of size u subject to these operations.

INSERT$(x \in U \setminus S)$: Add x to S

DELETE$(x \in S)$: Remove x from S

SUCCESSOR$(x \in U)$: Find the next element in S larger than any element x of the universe U

PREDECESSOR$(x \in U)$: Find the previous element in S smaller than x.

Solutions to fixed-universe successor problem

Goal: Maintain a dynamic subset S of size n of the universe $U = \{0, 1, \ldots, u-1\}$ of size u, subject to INSERT, DELETE, SUCCESSOR, PREDECESSOR

- Balanced search trees can implement operations in $O(\lg n)$ time, without fixed-universe assumptions.
- In 1975, Peter van Emde Boas solved this problem in $O(\lg \lg u)$ time per operation.
  - If u is only polynomial in n, that is, $u = O(n^c)$, then $O(\lg \lg n)$ time per operation - exponential speedup

$O(\lg \lg u)$?!

Where could a bound of $O(\lg \lg u)$ arise?

- Binary search over $O(\lg u)$ things
- $T(u) = T(\sqrt{u}) + O(1)$

  $T'(\lg u) = T'((\log u)/2) + O(1)$

  $= O(\lg \lg u)$.

1). Starting point: Bit Vector

Bit vector v stores, for each $x \in U$

$$v_x = \begin{cases} 1 & \text{if } x \in S \\ 0 & \text{if } x \notin S \end{cases}$$

Example:

$u = 16,\ n = 4,\ S = \{1, 9, 10, 15\}$

0	1	0	0	0	0	0	0	0	1	1	0	0	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Insert / Delete run in $O(1)$ time

Successor / Predecessor run in $O(u)$ worst-case time

2. Split universe into widgets

Carve universe of size u into $\sqrt{u}$ widgets.

$W_0, W_1, \dots W_{\sqrt{u}-1}$ each of size $\sqrt{u}$

Example:

$u = 16,\ \sqrt{u} = 4$

W_0				W_1				W_2				W_3			
0	1	0	0	0	0	0	0	0	1	1	0	0	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

W_0 represents $0, 1, \dots, \sqrt{u}-1 \in U$;

W_1 represents $\sqrt{u}, \sqrt{u}+1, \dots, 2\sqrt{u}-1 \in U$;

$\vdots$

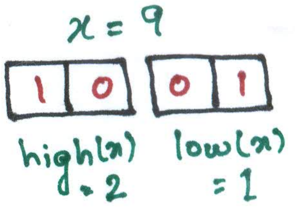
W_i represents $i\sqrt{u}, i\sqrt{u}+1, \dots, (i+1)\sqrt{u}-1 \in U$;

$\vdots$

$W_{\sqrt{u}-1}$ represents $u-\sqrt{u}, u-\sqrt{u}+1, \dots, u-1 \in U$.

Define $high(x) \geq 0$ and $low(x) \geq 0$

so that $x = high(x)\sqrt{u} + low(x)$.

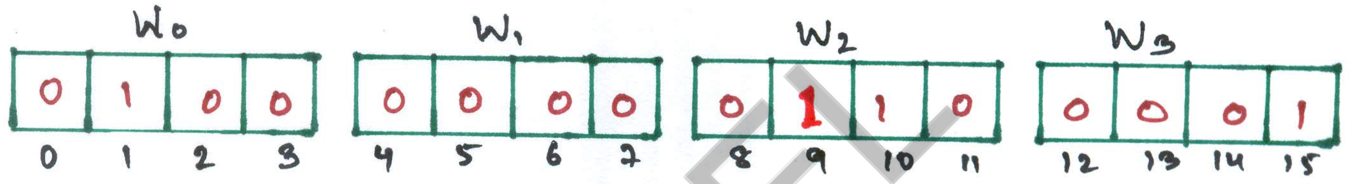


That is, if we write $x \in U$ in binary,

$high(x)$ is the high-order half of the bits, and

$low(x)$ is the low-order half of the bits.

For $x \in U$, $high(x)$ is the index of widget containing x and $low(x)$ is the index of x within that widget.



INSERT(x)

```
1.  insert x into widget $W_{high(x)}$ at position $low(x)$
2.  mark $W_{high(x)}$ as nonempty
```

Running time: $T(n) = O(1)$

SUCCESSOR(x)

```
1.  look for successor of x within widget $W_{high(x)}$      } $O(\sqrt{u})$
2.          starting after position $low(x)$
3.  if successor found
4.      then return it
5.  else find smallest $i > high(x)$                           } $O(\sqrt{u})$
6.          for which $W_i$ is non-empty
7.          return smallest element in $W_i$                   } $O(\sqrt{u})$
```

Running time: $T(u) = O(\sqrt{u})$

Revelation

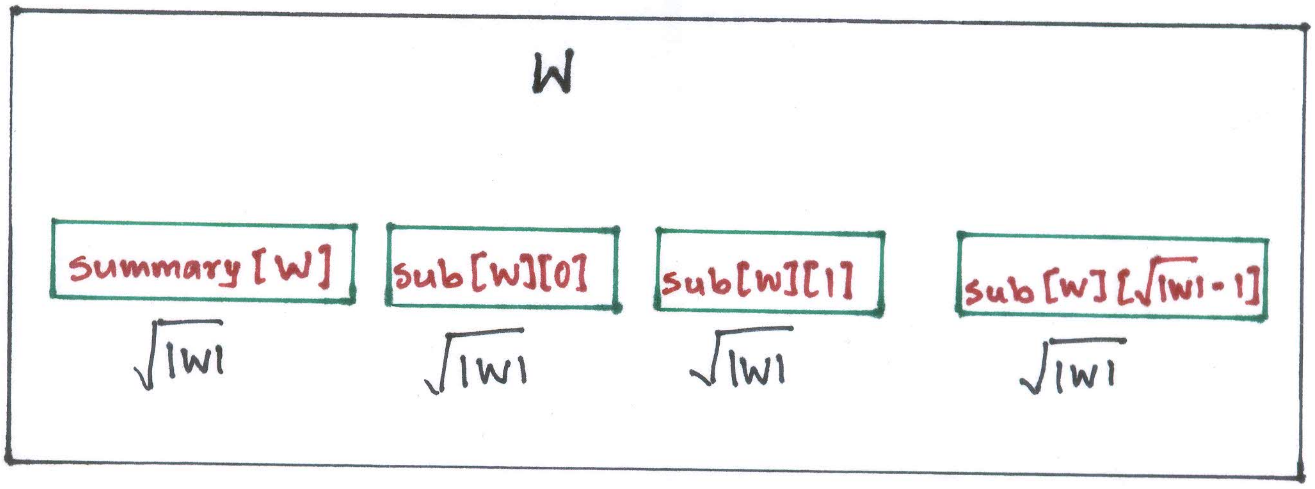
```
SUCCESSOR (x)
1. look for successor of $x$ within widget $W_{high(x)}$    } recursive
2.      starting after position $low(x)$                    } successor
3. if successor found
4.      then return it
5. else find smallest $i > high(x)$                         } recursive
6.              for which $W_i$ is nonempty                 } successor
7.      return smallest element in $W_i$                    } recursive successor
```

3. Recursion

Represent universe by widget of size u

Recursively split each widget W of size $|W|$ into $\sqrt{|W|}$ subwidgets sub[W][0], sub[W][1], ..., sub[W][$\sqrt{|W|}-1$] each of size $\sqrt{|W|}$

Store a summary widget summary[W] of size $\sqrt{|W|}$ representing which subwidgets are non-empty.



Define $high(x) \geq 0$ and $low(x) \geq 0$

so that $x = high(x)\sqrt{|W|} + low(x)$

INSERT (x, W)

```
1. if sub[W][high(x)] is empty
2.     then INSERT(high(x), summary[W])
3. INSERT(low(x), sub[W][high(x)])
```

Running time: $T(u) = 2T(\sqrt{u}) + O(1)$

$T'(\log u) = 2T'((\lg u)/2) + O(1)$

$= O(\lg u)$

SUCCESSOR (x, W)

```
j ← SUCCESSOR(low(x), sub[W][high(x)])   } $T(\sqrt{u})$
if $j < \infty$
    then return $high(x)\sqrt{|W|} + j$
else  i ← SUCCESSOR(high(x), summary[W])   } $T(\sqrt{u})$
      j ← SUCCESSOR($-\infty$, sub[W][i])   } $T(\sqrt{u})$
      return $i\sqrt{|W|} + j$
```

Running time: $T(u) = 3T(\sqrt{u}) + O(1)$

$T'(\lg u) = 3T'((\lg(u))/2) + O(1)$

$= O((\lg u)^{\lg 3})$.

Improvements

Need to reduce INSERT and SUCCESSOR down to 1 recursive call each.

- 1 call: $T(u) = 1\,T(\sqrt{u}) + O(1)$
  $= O(\lg \lg n)$
- 2 Calls: $T(u) = 2T(\sqrt{u}) + O(1)$
  $= O(\log n)$
- 3 Calls: $T(u) = 3T(\sqrt{u}) + O(1)$
  $= O((\lg n)^{\lg 3})$

We are closer to this goal than it may seem!

Recursive calls in successor

If x has a successor within sub[W][high(x)], then there is only 1 recursive call to SUCCESSOR.

Otherwise, there are 3 recursive calls.

- SUCCESSOR (low(x), sub[W][high(x)])
  discovers that sub[W][high(x)] has no successor
- SUCCESSOR (high(x), summary [W])
  finds next non-empty subwidget sub[W][i]
- SUCCESSOR ($-\infty$, sub[W][i])
  finds smallest element in subwidget sub[W][i]

Reducing recursive calls in successor

If x has no successor within sub[W][high(x)], there are 3 recursive calls:

- SUCCESSOR (low(x), sub [W][high(x)]) discovers that sub[W][high(x)] hasn't successor
  - Could be determined using the maximum value in the subwidget sub [W][high(x)]
- SUCCESSOR (high(x), summary [W]) finds next nonempty subwidget sub [W][i]
- SUCCESSOR ($-\infty$, sub[W][i]) finds minimum element in subwidget sub [W][i].

Improved Successor

```
INSERT $(x, W)$
    if sub$[W][\text{high}(x)]$ is empty
        then INSERT $(\text{high}(x), \text{summary}[W])$
    INSERT $(\text{low}(x), \text{sub}[W][\text{high}(x)])$
    if $x <$ min$[W]$, then min$[W] \leftarrow x$    } new
    if $x >$ max$[W]$, then max$[W] \leftarrow x$    } (augmentation)
```

Running Time: $T(u) = 2T(\sqrt{u}) + O(1)$

$T'(\lg u) = 2T'((\lg u)/2) + O(1)$

$= O(\lg u)$

```
SUCCESSOR $(x, W)$
  if low$(x) <$ max$[\text{sub}[W][\text{high}(x)]]$
  then $j \leftarrow$ SUCCESSOR$(\text{low}(x), \text{sub}[W][\text{high}(x)])$ } $T(\sqrt{u})$
       return high$(x)\sqrt{|W|} + j$
  else $i \leftarrow$ SUCCESSOR$(\text{high}(x), \text{summary}[W])$ } $T(\sqrt{u})$
       $j \leftarrow$ min$[\text{sub}[W][i]]$
       return $i\sqrt{|W|} + j$
```

Running Time: $T(u) = T(\sqrt{u}) + O(1)$

$= O(\lg \lg u)$

Recursive calls in insert

If sub[W][high(x)] is already in summary[W], then there is only 1 recursive call to INSERT.

Otherwise, there are 2 recursive calls:

- INSERT(high(x), summary [W])
- INSERT (low(x), sub[W][high(x)])

Idea:

We know that sub([W][high(x)]) is empty.

Avoid second recursive call by specially storing a widget containing just 1 element.

Specifically, do not store min recursively.

Improved insert

INSERT (x, W)

```
1. if $x <$ min[W] then exchange $x \leftrightarrow$ min[W]
2. if sub[W][high(x)] is nonempty, that is
3.      min[sub[W][high(x)]] $\neq$ NIL
3.    then INSERT (low(x), sub[W][high(x)])
4. else min [sub[W][high(x)]] $\leftarrow$ low(x)
5.      INSERT (high(x), summary [W])
6. if $x >$ max[W] then max [W] $\leftarrow x$
```

Running Time: $T(u) = 1\,T(\sqrt{u}) + O(1)$
$= O(\lg \lg u)$

SUCCESSOR (x, W)

```
1. if $x <$ min [W] then return min [W]
2. if low (x) < max [sub [W][high(x)]]
3.    then j $\leftarrow$ SUCCESSOR (low(x), sub [W][high(x)])
4.    return high (x)$\sqrt{|W|}$ + j
5. else i $\leftarrow$ SUCCESSOR (hig(x), summary [W])
6.      j $\leftarrow$ min [sub [W][i]]
7.      return i$\sqrt{|W|}$ + j
```

Running Time: $T(u) = 1\,T(\sqrt{u}) + O(1)$
$= O(\lg \lg u)$

Deletion

```
DELETE ($x$, W)
1.  if min [W] = NIL or $x$ < min [W] then return
2.  if $x$ = min[W]
3.      then i ← min [summary [W]]
4.           $x \leftarrow i\sqrt{|W|}$ + min [ sub[W][i]]
5.           min [W] ← $x$
6.  DELETE ( low($x$), sub [W][high($x$)])
7.  if sub [W][high ($x$)] is now empty, that is
        min [sub[W][high($x$)]] = NIL
8.      then DELETE ( high ($x$), summary [W])
```

(in this case, the first recursive call was cheap)

Amortized Algorithms

How large should a hash table be?

Goal:

Make the table as small as possible, but large enough so that it won't overflow (or otherwise become inefficient).

Problem:

What if we don't know the proper size in advance?

Solution:

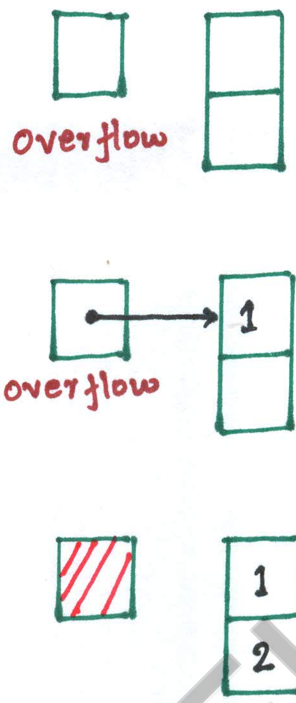
Dynamic Tables

Idea:

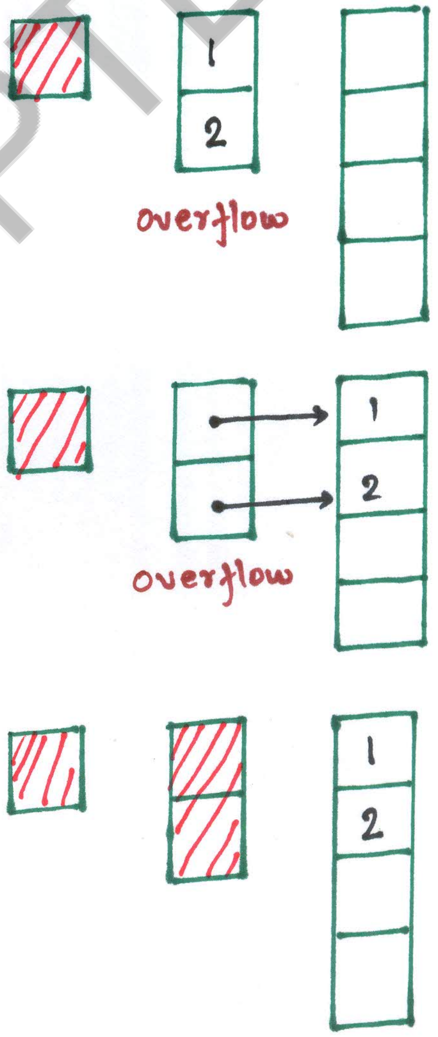
Whenever the table overflows, "grow" it by allocating (via malloc or new) a new, larger table. Move all items from the old table into the new one, and free the storage for the old table.

Example of a dynamic table

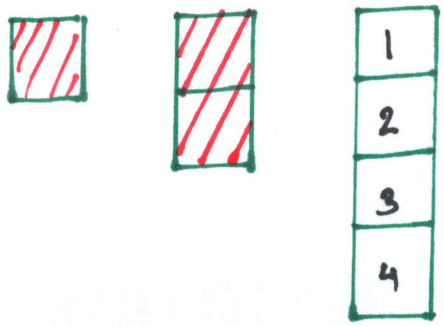
1. INSERT
2. INSERT



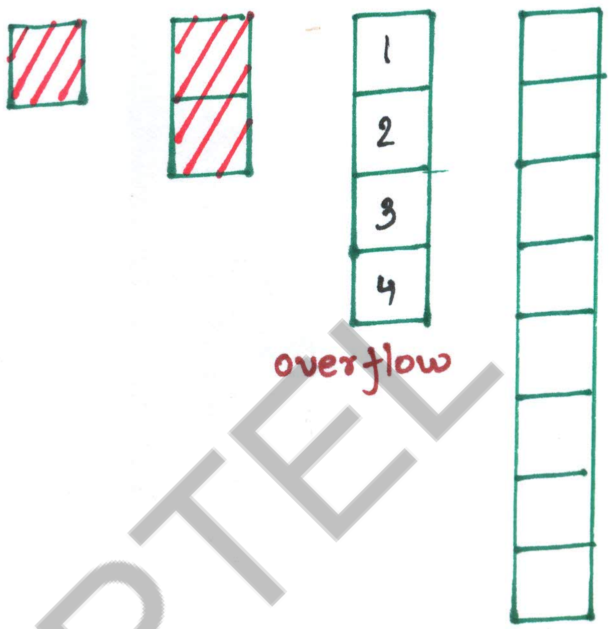
1. INSERT
2. INSERT
3. INSERT

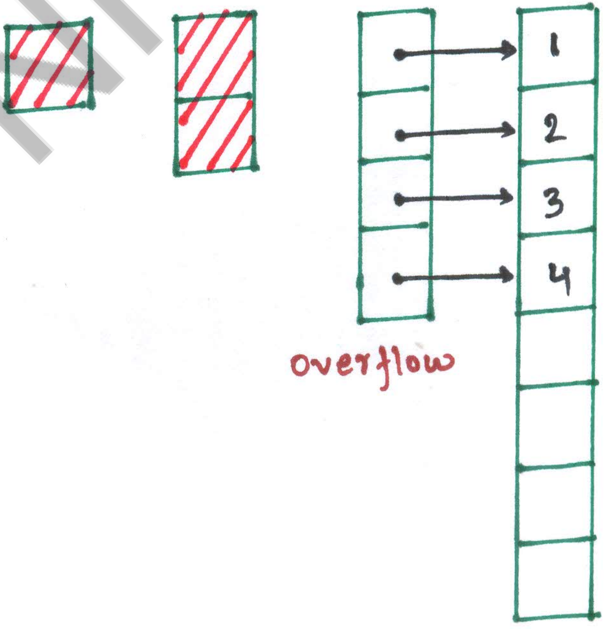


1. INSERT
2. INSERT
3. INSERT
4. INSERT

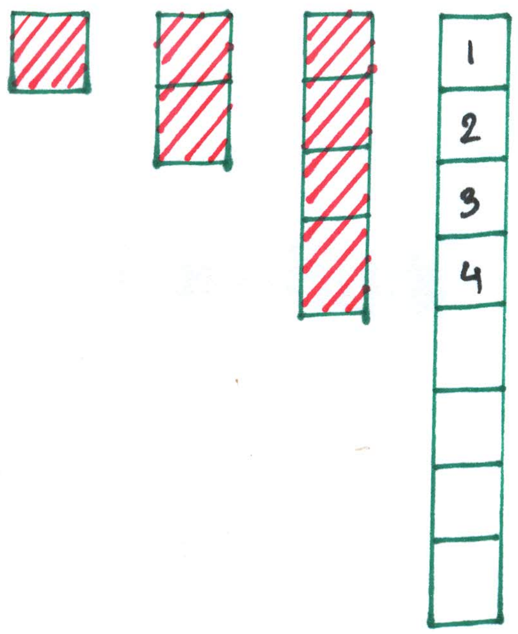


1. INSERT
2. INSERT
3. INSERT
4. INSERT
5. INSERT

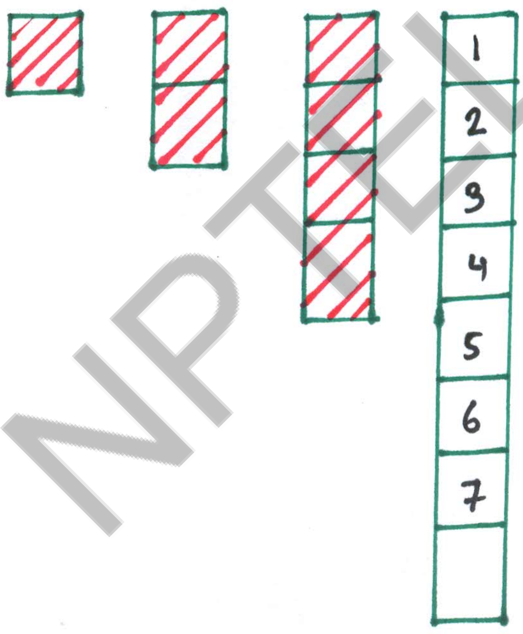




1. INSERT
2. INSERT
3. INSERT
4. INSERT
5. INSERT



1. INSERT
2. INSERT
3. INSERT
4. INSERT
5. INSERT
6. INSERT
7. INSERT



Worst-case analysis

Consider a sequence of n insertions. The worst-case time to execute one insertion is $\Theta(n)$.
Therefore, the worst case time for n insertions is:

$$n \cdot \Theta(n) = \Theta(n^2).$$

WRONG! Infact, the worst case cost for n insertions is only $\Theta(n) << \Theta(n^2)$

Let us see why?

Tighter analysis

Let c_i = the cost of i^{th} insertion

$$= \begin{cases} i, & \text{if } i-1 \text{ is an exact power of } 2; \\ 1, & \text{otherwise.} \end{cases}$$

i	1	2	3	4	5	6	7	8	9	10
$size_i$	1	2	4	4	8	8	8	8	16	16
c_i	1	2	3	1	5	1	1	1	9	1

Tighter analysis (contineed)

i	1	2	3	4	5	6	7	8	9	10
$size_i$	1	2	4	4	8	8	8	8	16	16
c_i	1	1	1	1	1	1	1	1	1	1
		1	2		4				8	

Cost of n operations $= \sum_{i=1}^{n} c_i$

$$\le n + \sum_{j=0}^{\lfloor \lg(n-1) \rfloor} 2^j$$

$$\le 3n$$

$$= \Theta(n)$$

Thus, the average cost of each dynamic-table operation is:

$$\frac{\Theta(n)}{n} = \Theta(1)$$

Amortized Analysis

An amortized analysis is any strategy for analyzing a sequence of operations to show that the average cost per operation is small, even though a single operation within the sequence might be expensive.

Even though we are taking averages, however, probability is not involved!

- An amortized analysis guarantees the average performance of each operation in the worst case.

Types of amortized analyses

Three common amortization arguments:

the aggregate method,
the accounting method,
the potential method.

We have just seen an aggregate analysis.

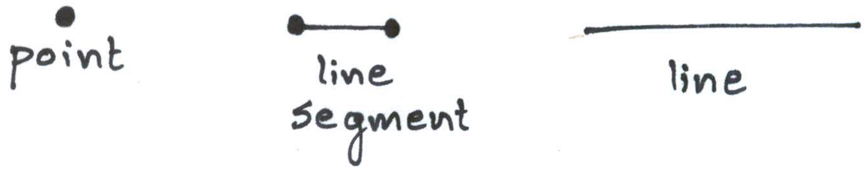
The aggregate method, though simple, lacks the preci sion of other two methods.

In particular, the accounting and potential methods allow a specific amortized cost to be allocated to each operation.

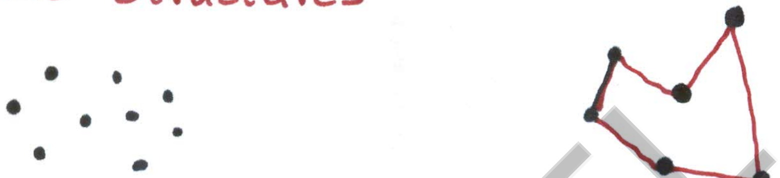
Computational Geometry

- Algorithms for solving "geometric problems" in 2D and higher.

- Fundamental Objects

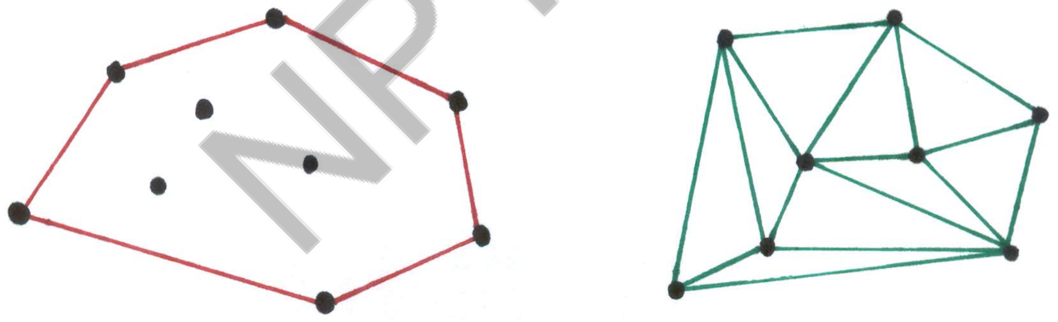


- Basic Structures



point set

polygon



Convex hull

triangulation.

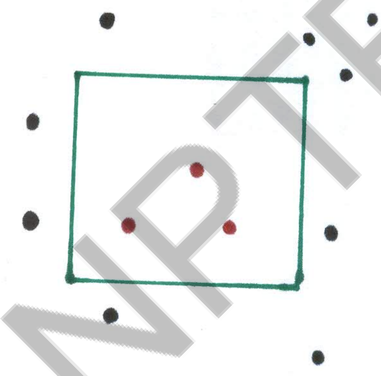
Orthogonal Range Searching

Input: n points in d dimensions.

- E.g. representing a database of n records each with d numeric fields.

Query: Axis aligned box (in 2D, a rectangle)

- Reports on the points inside the box:
  - Are there any points?
  - How many are there?
  - List the points.

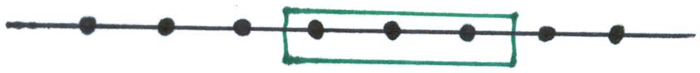


Goal: Preprocessing points into data structure to support fast queries.

- Primary goal: Static data Structure
- In 1D we will also obtain a dynamic data structure supporting insert and delete.

1D range searching

In 1D, the query is an interval



First solution using ideas we know:

- Interval trees
  - Represent each point x by the interval $[x,x]$.
  - Obtain a dynamic structure that can list K answers in a query in $O(K\lg n)$ time.

Second solution using ideas we know:

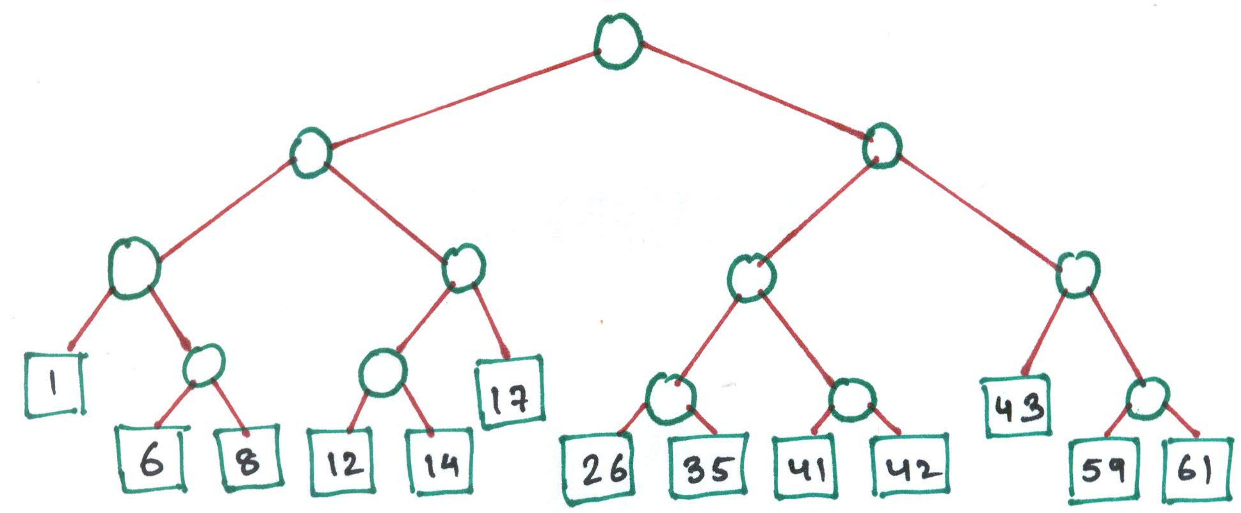
- Sort the points and store them in an array
  - Solve query by binary search on end points
  - Obtain a static structure that can list K answers in a query in $O(k+\log n)$ time.

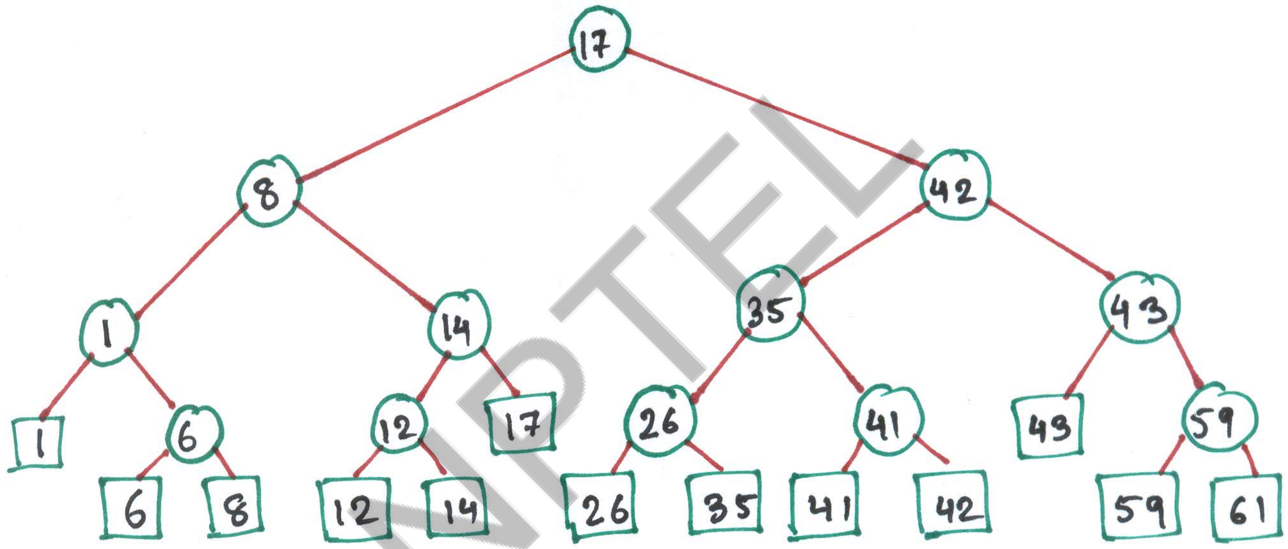
Goal: Obtain a dynamic structure that can list K answers in a query in $O(K+\log n)$ time.

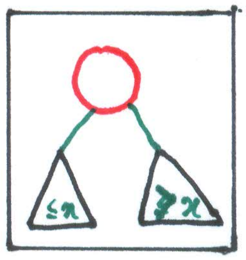
New solution that extends to higher dimensions:

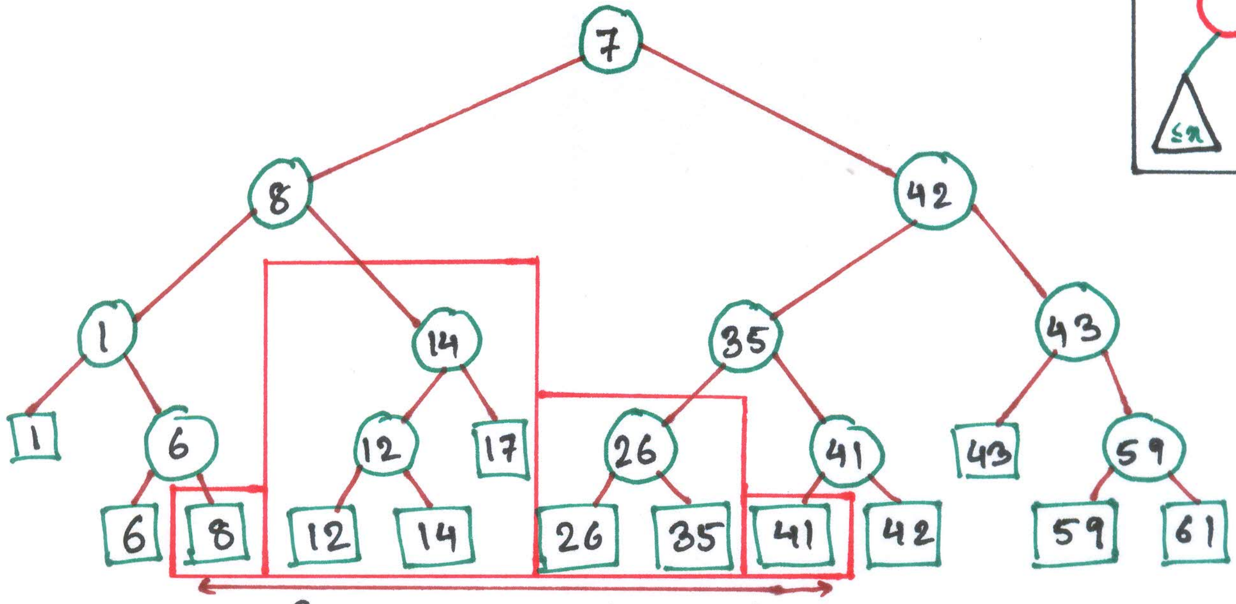
- Balanced binary search tree
  - New organization principle:
    - Store points in the leaves of the tree
  - Internal nodes store copies of the leaves to satisfy binary search property.
    - Node x stores in $key[x]$ the maximum key of any leaf in the left subtree of x.

Example of a 1D range tree



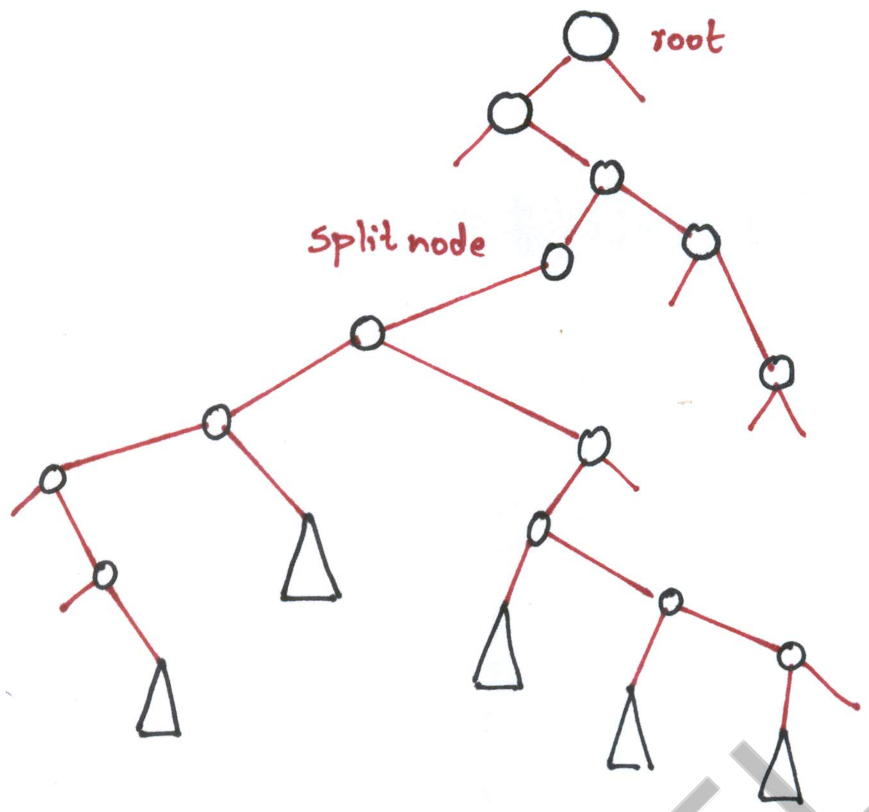






RANGE-QUERY ([7, 41])

General 1D range query



Pseudocode, part 1: Find the split node

```
1D-RANGE-QUERY (T, [$x_1$, $x_2$])
1.  w ← root[T]
2.  while w is not a leaf and ($x_2 \le$ key[w] or key[w] $< x_1$)
3.        do if $x_2 \le$ key[w]
4.              then w ← left[w]
5.              else w ← right[w]
    ▶ w is now the split node
```

traverse left and right from 'w' and report relevant subtrees.

Pseudo code, part 2: Traverse left and right from Split Node.

```
1D-RANGE-QUERY (T, [$x_1$, $x_2$])
  [find the split node]
  ▶ w is now the split node
1. if w is leaf
2.    then output leaf w if $x_1 \leq key[w] \leq x_2$
3. else v ← left[w]          ▶left traversal
4.      while v is not a leaf
5.         do if $x_1 \leq key[v]$
6.              then output the subtree rooted at right[v]
7.                   v ← left[v]
8.              else v ← right[v]
9.   output the leaf v if $x_1 \leq key[v] \leq x_2$
   [symmetrically for right travelsal
```

Analysis of 1D-range query

Query Time: Answer to range query represented by $O(\log n)$ subtrees found in $O(\log n)$ time.

Thus
- Can test for points in interval in $O(\log n)$ time
- Can count points in interval in $O(\log n)$ time if we augment the tree with subtree sizes.
- Can report the first k points in $O(k + \lg n)$ time.

Space: $O(n)$

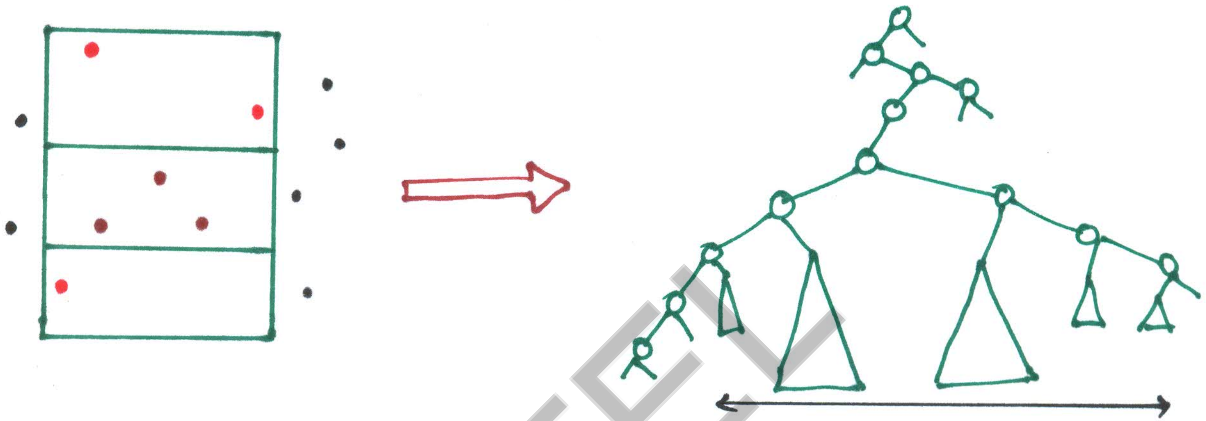
Preprocessing time: $O(n \log n)$

2D range trees

Store a primary 1D range tree for all the points based on x-coordinate

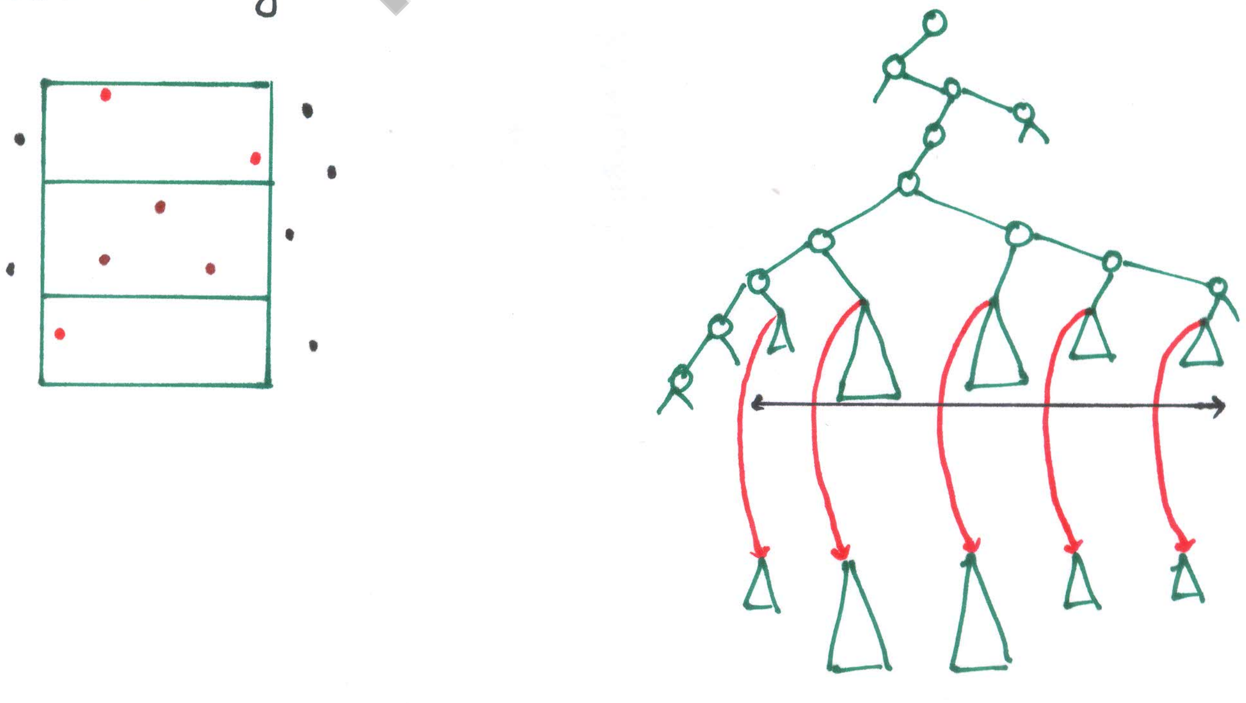
Thus in $O(\log n)$ time we can find $O(\log n)$ subtrees representing the points with proper x-cordinates.

How to restrict to points with proper y-co-ordinates?



Idea: In primary 1D range tree of x-cordinate, every node stores a secondary 1D range tree based on y-co-ordinate for all points in the subtree of the node

Recursively search within each.



Analysis of 2D range tree

Query time: In $O(\lg^2 n) = O((\lg n)^2)$ time, we can represent answer to range query by $O(\lg^2 n)$ subtrees.

Total cost for reporting k points: $O(k+(\lg n)^2)$.

Space: The secondary trees at each level of the primary tree together store a copy of the points. Also, each point is present in each secondary tree along the path from the leaf to the root.

Either way, we obtain that space is $O(n \log n)$.

Preprocessing time: $O(n \log n)$

d-dimensional range trees

Each node of secondary y-structure stores a tertiary z-structure representing the points in the subtree rooted at the node, etc.

Query time: $O(k + \log^d n)$ to report k points

Space : $O(n \log^{d-1} n)$

Preprocessing time: $O(n \log^{d-1} n)$

Best Data Structure to date:

Query time : $O(k + \log^{d-1} n)$ to report k points.

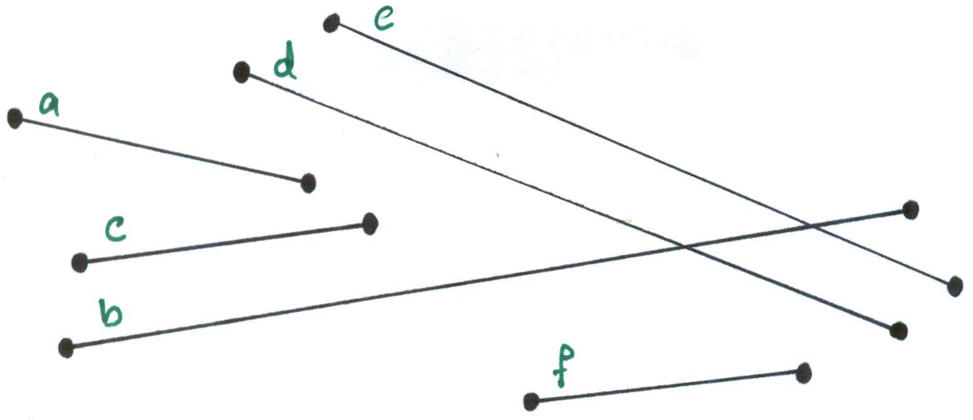
Space: $O(n(\log n / \log \log n)^{d-1})$

Preprocessing time: $O(n \log^{d-1} n)$

Line - segment intersection

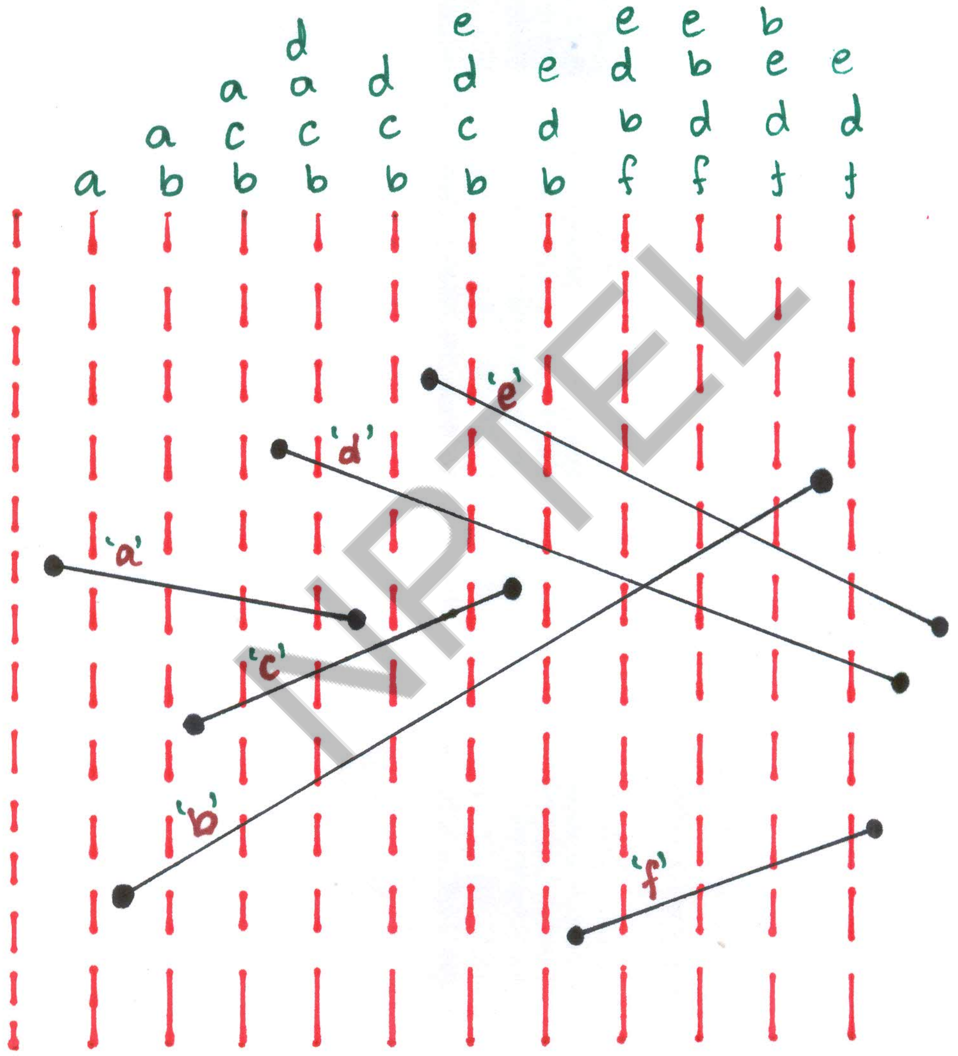
Given n line segments, does any pair intersect?

Obvious algorithm: $O(n^2)$



Sweep-line algorithm

- Sweep a vertical line from left to right (conceptually replacing x- cordinate with time)
- Maintain dynamic set S of segments that intersect the sweep line, ordered (tentatively) by y-co-ordinate of intersection.
- Order changes when
  - new segment is encountered, } segment
  - existing segment finishes, or } endpoints
  - two segments cross
- Key eventpoints are therefore segment endpoints.



a
b a
b c a
b c a d
b c d
b c d e
b d e
f b d e
f d b e
f d e b
f d e
'a'
'b'
'c'
'd'
'e'
'f'

Sweep-line algorithm

Process event points in order by sorting segment endpoints by x-coordinates and looping through:

- For a left endpoint of segment s:
  - Add segment s to dynamic set S
  - Check for intersection between s and its neighbours in S
- For a right endpoint of segment s:
  - Remove segment s from dynamic set S
  - Check for intersection between s and the neighbours of s in S.

Analysis

Use red black tree to store dynamic set S.

Total running time: $O(n \log n)$

Correctness

Theorem: If there is an intersection, the algorithm finds it.

Proof: Let X be the leftmost intersection point.

Assume for simplicity that

- only two segments s_1, s_2 pass through X, and
- no two points have the same x-co-ordinates.

At some point before we reach X, s_1 and s_2 become consecutive in order of S.

Either initially consecutive when s_1 and s_2 inserted or became consecutive when another deleted.