Geometric Structures

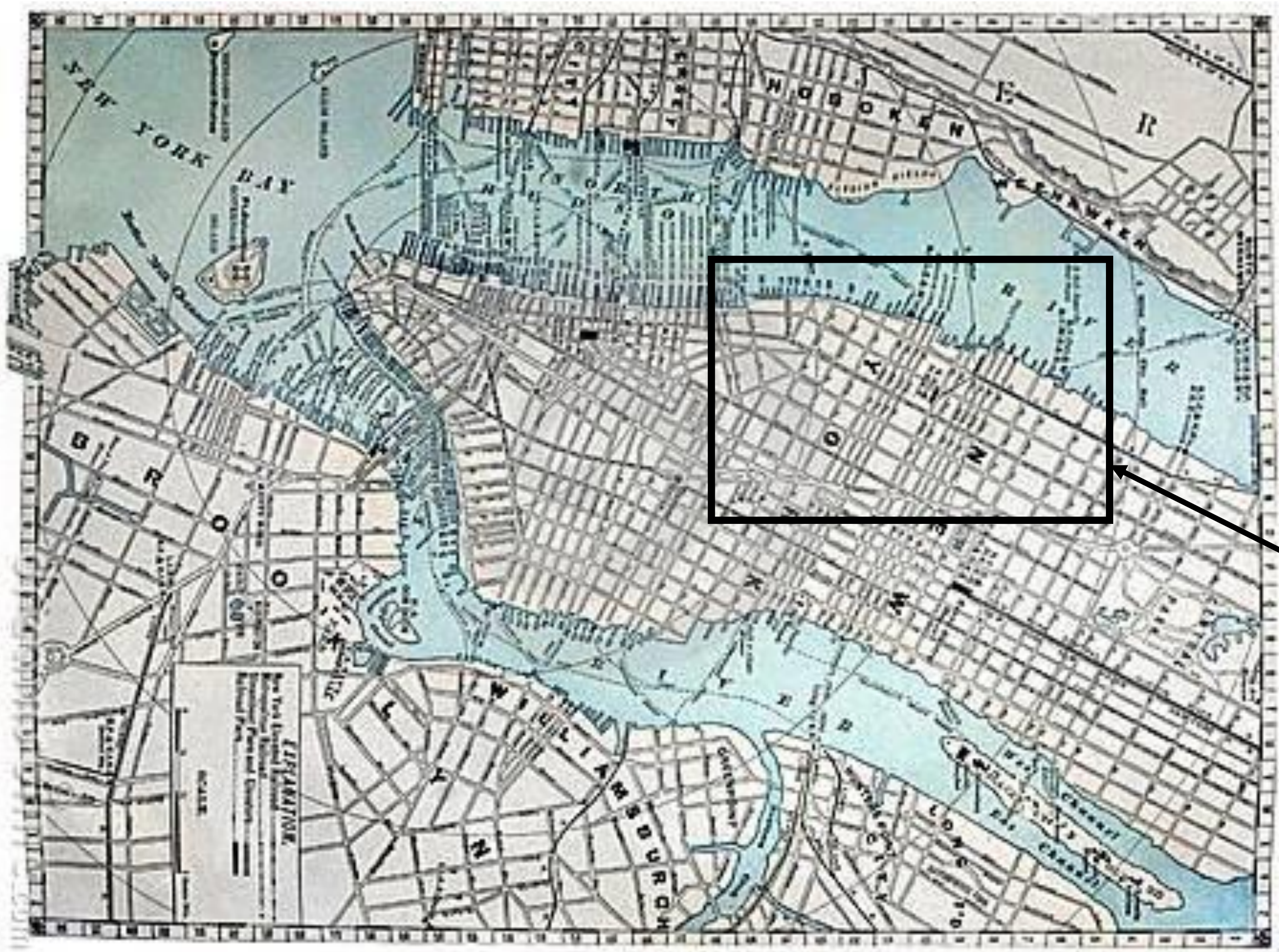
1. Interval, segment, range trees

Martin Samuelčík
samuelcik@sccg.sk, www.sccg.sk/~samuelcik, I4

Window and Point queries

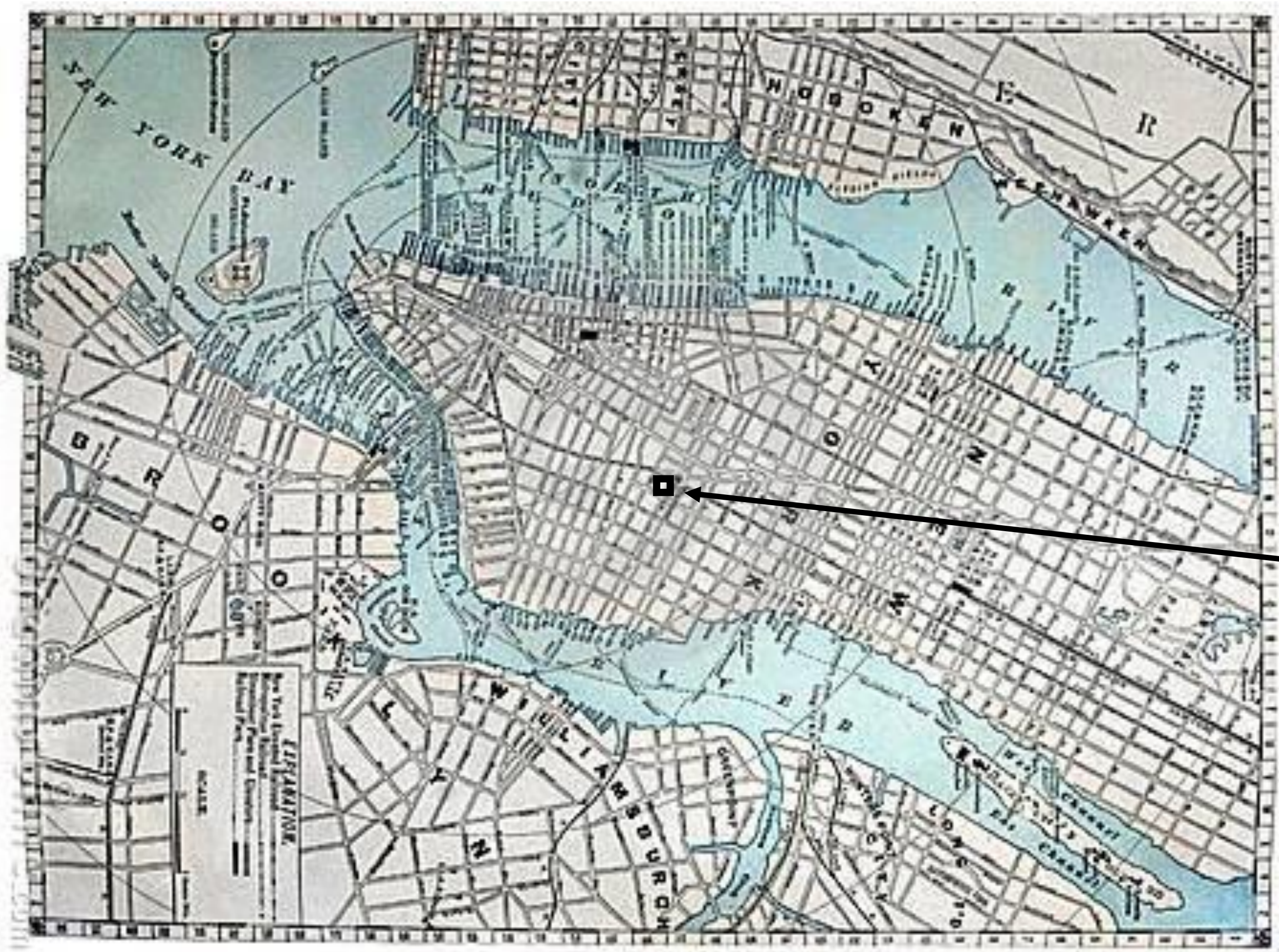
- For given points, find all of them that are inside given *d*-dimensional interval
- For given *d*-dimensional intervals, find all of them that contain one given point
- For given *d*-dimensional intervals, find all of them that have intersection with another d-dimensional interval
- *d*-dimensional interval: interval, rectangle, box,
- Example for 1D:
  - Input: set of *n* intervals, real value *q*
  - Output: *k* intervals containing value *q*

Motivation



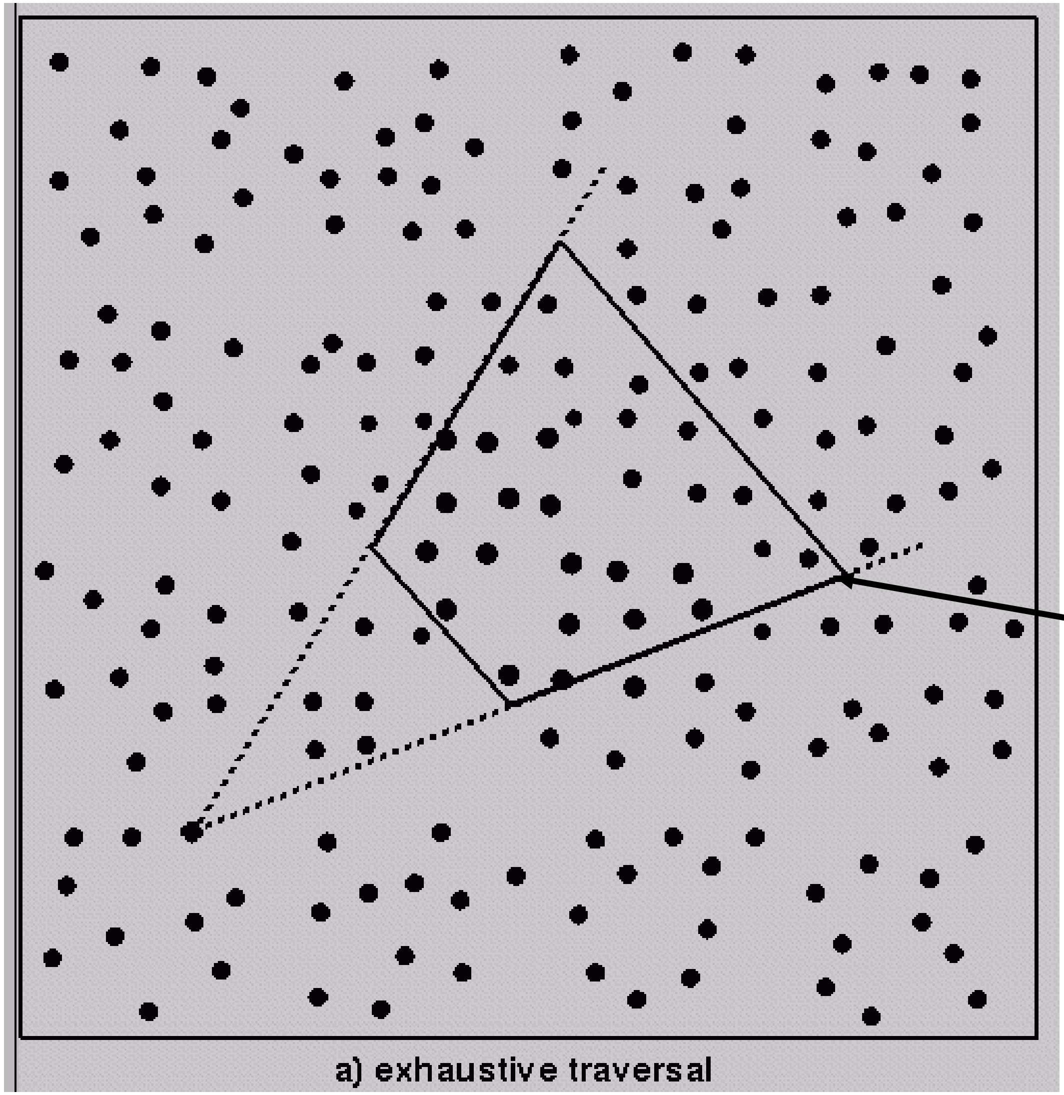
Find all buildings in given rectangle.

Motivation



Find building under given point

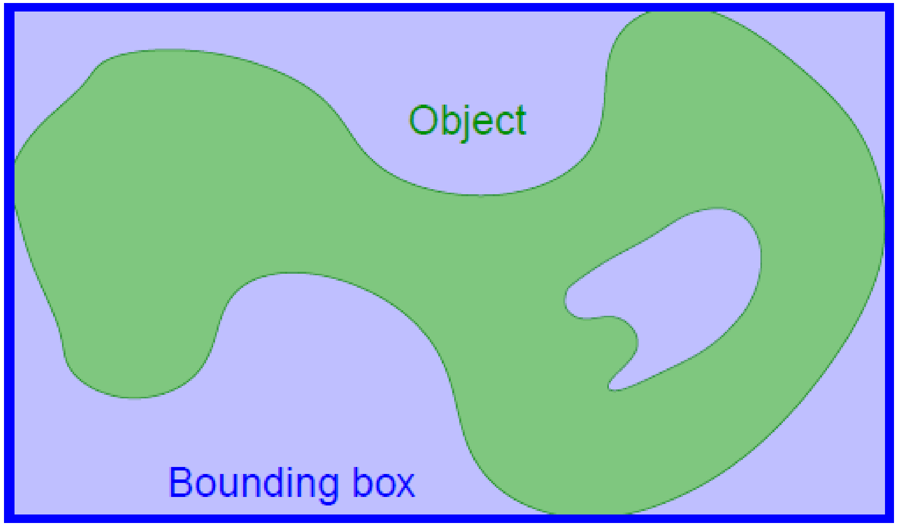
Motivation

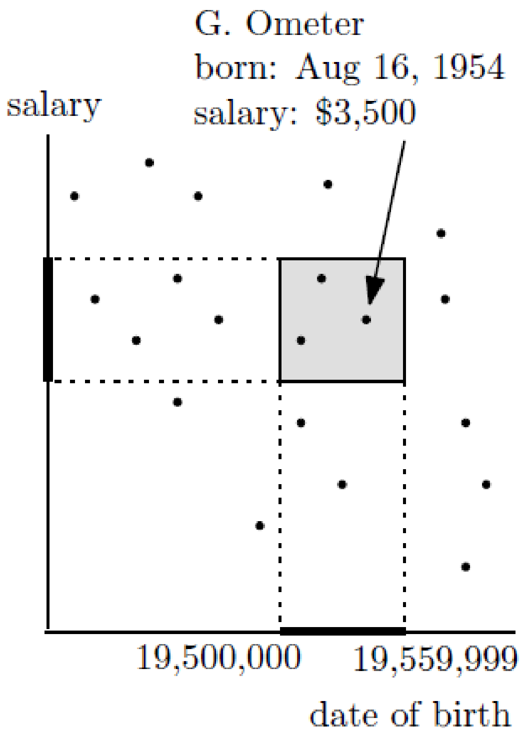


Find all points in viewing frustum.

Motivation

- Approximation of objects using axis-aligned bounding box (AABB) – used in many fields
- Solving problem for AABB and after that solving for objects itself





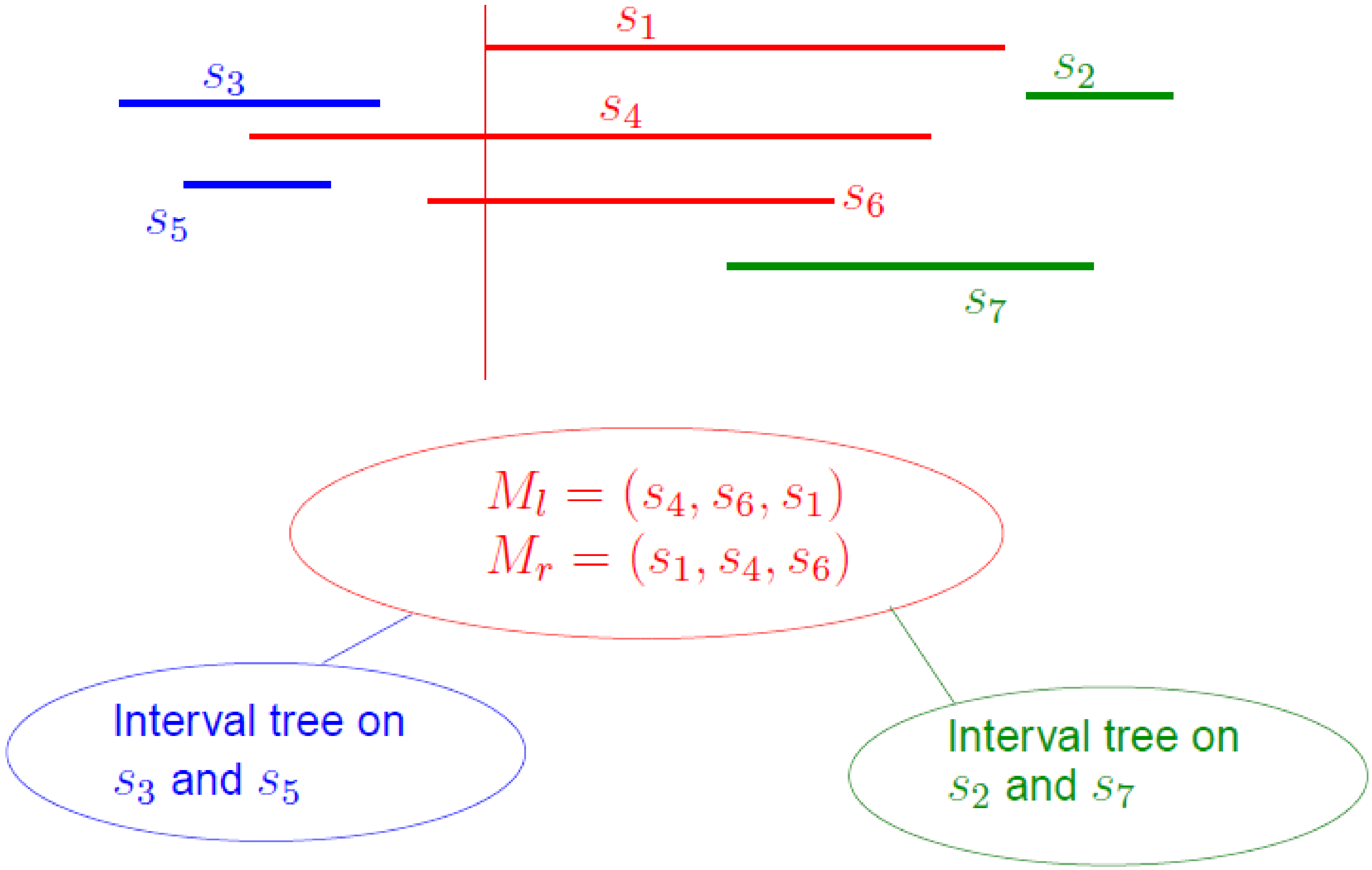
Interval tree

- Binary tree storing intervals
- Created for finding intervals containing given value
- Input: Set S of closed one-dimensional intervals, real value x_q
- Output: all intervals $I \in S$ such that $x_q \in I$
- $S = \{[l_i, r_i] \text{ pre } i = 1, \ldots, n\}$

Node of interval tree

- X – value, used to split intervals for node
- M_l – ordered set of intervals containing split value X, ascending order of minimal values of intervals
- M_r – ordered set of intervals containing split value X, descending order of maximal values of intervals
- Pointer to left son (left subtree), containing intervals with values less than split value X
- Pointer to right son (right subtree), containing intervals with values more than split value X

Interval tree example



Tree construction

- Recursive construction
- Choosing splitting value X
  - For balanced tree – median of intervals limits
  - Median is good for static set without insert/delete operations
- Creating node:
  - Finding splitting value X
  - Split set of intervals into 3 subsets, intervals containing value X, intervals less then value X, intervals more than value X

Tree construction

```
struct Interval
{
        float left;
        float right;
}
```

```
struct IntervalTree
{
        IntervalTreeNode* root;
}
```

```
struct IntervalTreeNode
{
        float x;
        vector<Interval*> Ml;
        vector<Interval*> Mr;
        IntervalTreeNode * left;
        IntervalTreeNode * right;
}
```

```
LeftSegments(x, S)
{
        list<Interval*> result;
        Interval* I;
        for (each I in S)
        {
                if (I->right < x)
                        result.add(I);
        }
        return result;
}
```

```
HitSegments(x, S)
{
        list<Interval*> result;
        Interval* I;
        for (each I in S)
        {
                if (I->left <= x && x <= I->right)
                        result.add(I);
        }
        return result;
}
```

```
IntervalTreeConstruct(S)
{
        T = new IntervalTree;
        T = IntervalTreeNodeConstruct(S);
        return T;
}
```

```
IntervalTreeNodeConstruct(S)
{
        if (Size(S) == 0) return NULL;
        v = new IntervalTreeNode;
        v->x = Median(S);
        Sx = HitSegments(v->x, S);
        L = LeftSegments(v->x, S);
        R = RightSegments(v->x, S);
        v->Ml = SortLeftEndPoints(Sx);
        v->Mr = SortRightEndPoints(Sx)
        v->left = IntervalTreeNodeConstruct(L);
        v->right = IntervalTreeNodeConstruct(R);
        return v;
}
```

Tree properties

- Each interval is stored in exactly one node of the tree
- Node count is O(n)
- Sum of all counts for sets M_l and M_r is O(n)
- When using median for splitting intervals, height of tree is O(log(n))
- For n intervals, interval tree can be constructed in O(n.log(n)) time and takes O(n) memory

Searching

- Find all intervals $I \in S$ such that $x_q \in I$
- Recursive algorithm
- When using median, for set of n intervals, search query returns k intervals in time O(k + log(n))

```
IntervalStabbing(IntervalTreeNode* v, float xq)
{
        list<Interval*> D;
        if (xq < v->x)
        {
                Interval* F = v->Ml.first;
                while (F != NULL && F->left <= xq)
                {
                        D.insert(F);
                        F = v->Ml.next;
                }
                D1 = IntervalStabbing(v->left, xq);
                D.add(D1);
        }
        else
        {
                Interval* F = v->Mr.first;
                while (F != NULL && F->right >= xq)
                {
                        D.insert(Seg(F));
                        F = v->Mr.next;
                }
                D2 = IntervalStabbing(v->right, xq);
                D.add(D2);
        }

        return D;
}
```

Intervals intersection

- For given interval $I=[l,r]$, find in set S all intervals with non-empty intersection with I
- Recursive algorithm
  - IF $v\text{->}x \in I$, then all intervals stored in set $v\text{->}M_l$ have intersection with I, and both subtrees ($v\text{->}left$, $v\text{->}right$) of v have to be traversed
  - If $v\text{->}x < l$, then intervals from $v\text{->}M_r$ for which $r_i >= l$ have intersection with I and search subtree $v\text{->}right$
  - If $v\text{->}x > r$, then intervals from $v\text{->}M_l$ for which $l_i <= r$ have intersection with I and search subtree $v\text{->}left$
- Search time complexity is O(k + log(n)), where k is size of all intervals found

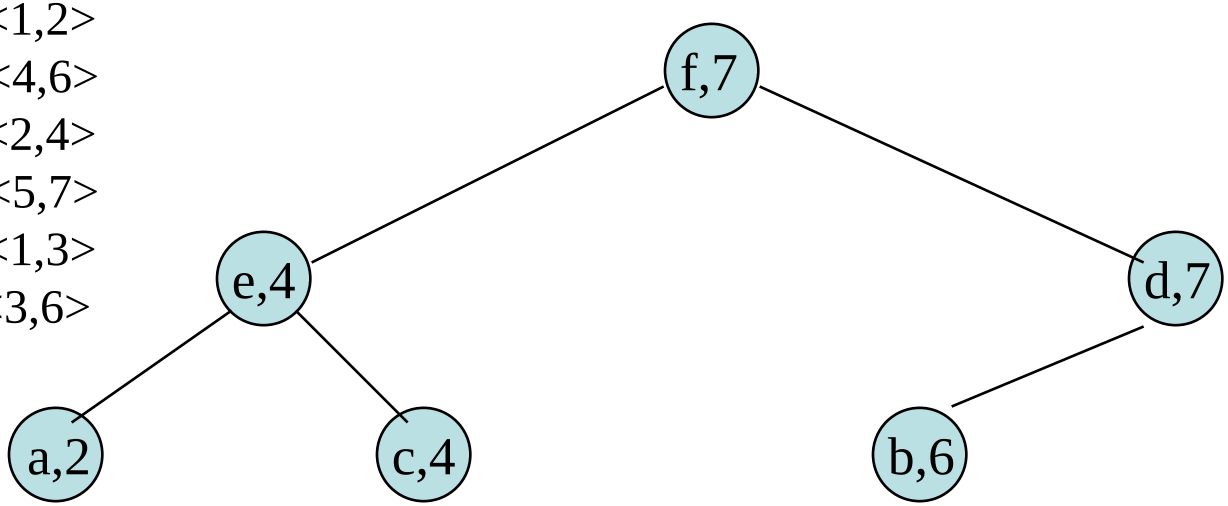
Other interval tree

- Find only one interval of set *S* that has non-empty intersection with given interval
- In each tree node *v* is stored 1 interval from *S* (*v->int*) and maximal value of end values of all intervals stored in subtree with root *v* (*v->max*)
- Using oredering of intervals, for example $[l_i,r_i]<[l_j,r_j] \iff l_i<l_j \mid (l_i=l_j \,\&\, r_i<r_j)$
- Creation of binary search tree (red-black tree)

Other interval tree

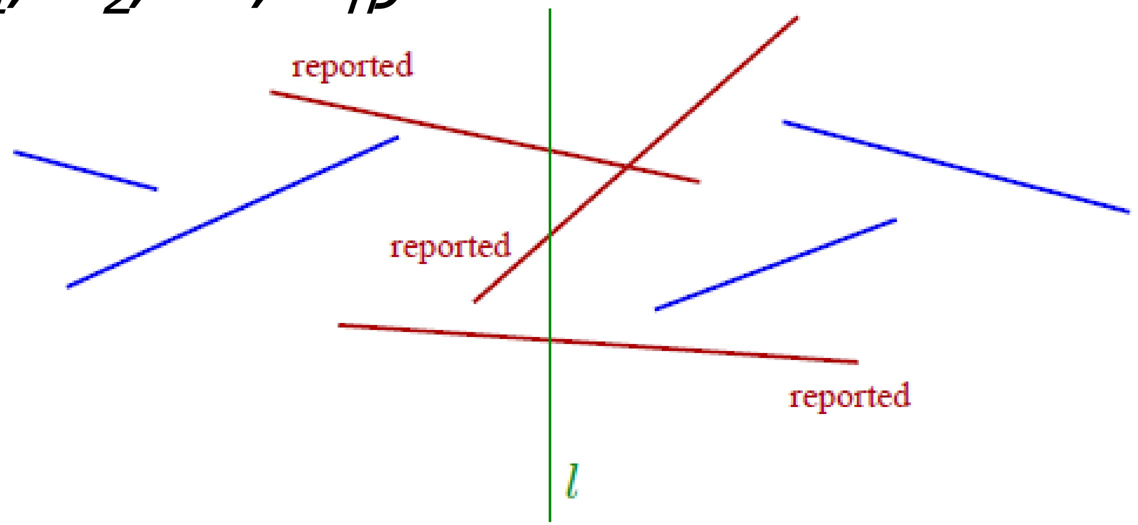
- Searching tree with given interval *I=[l,r]*
  - If *v->int* has intersection with *I*, return *v->int*
  - Else if *v->left->max >= l*, search in *v->left*
  - Else search in *v->right*

a=<1,2>
b=<4,6>
c=<2,4>
d=<5,7>
e=<1,3>
f=<3,6>



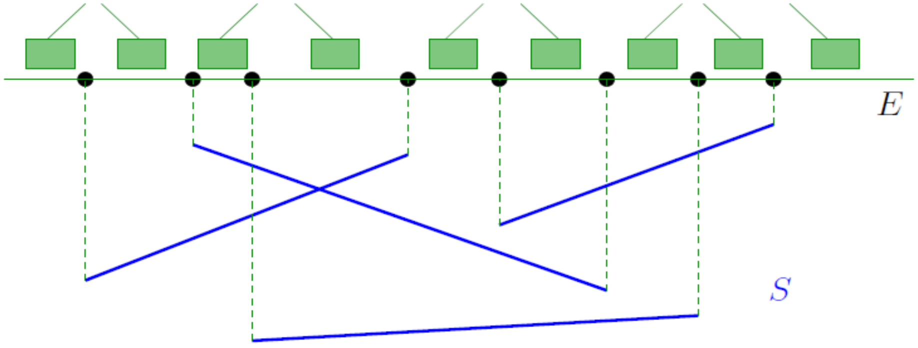
Segment trees

- Find all segments from given sets that have intersection with given vertical line
- Input: set of segments S in plane,
  vertical line l
- Output: all segments $s \in S$, intersected with l
- $S = \{s_1, s_2, \ldots, s_n\}$



Search tree

- Auxiliary structures
- Sort in ascending order x coordinates of segments end points from *S*, creating ordered set $E = \{e_1, e_2, \ldots, e_{2n}\}$
- Split E to atomic intervals $[-\infty, e_1], \ldots\ldots\ldots, [e_{2n-1}, e_{2n}], [e_{2n}, \infty]$
- Atomic intervals are leaves of search tree



Search tree

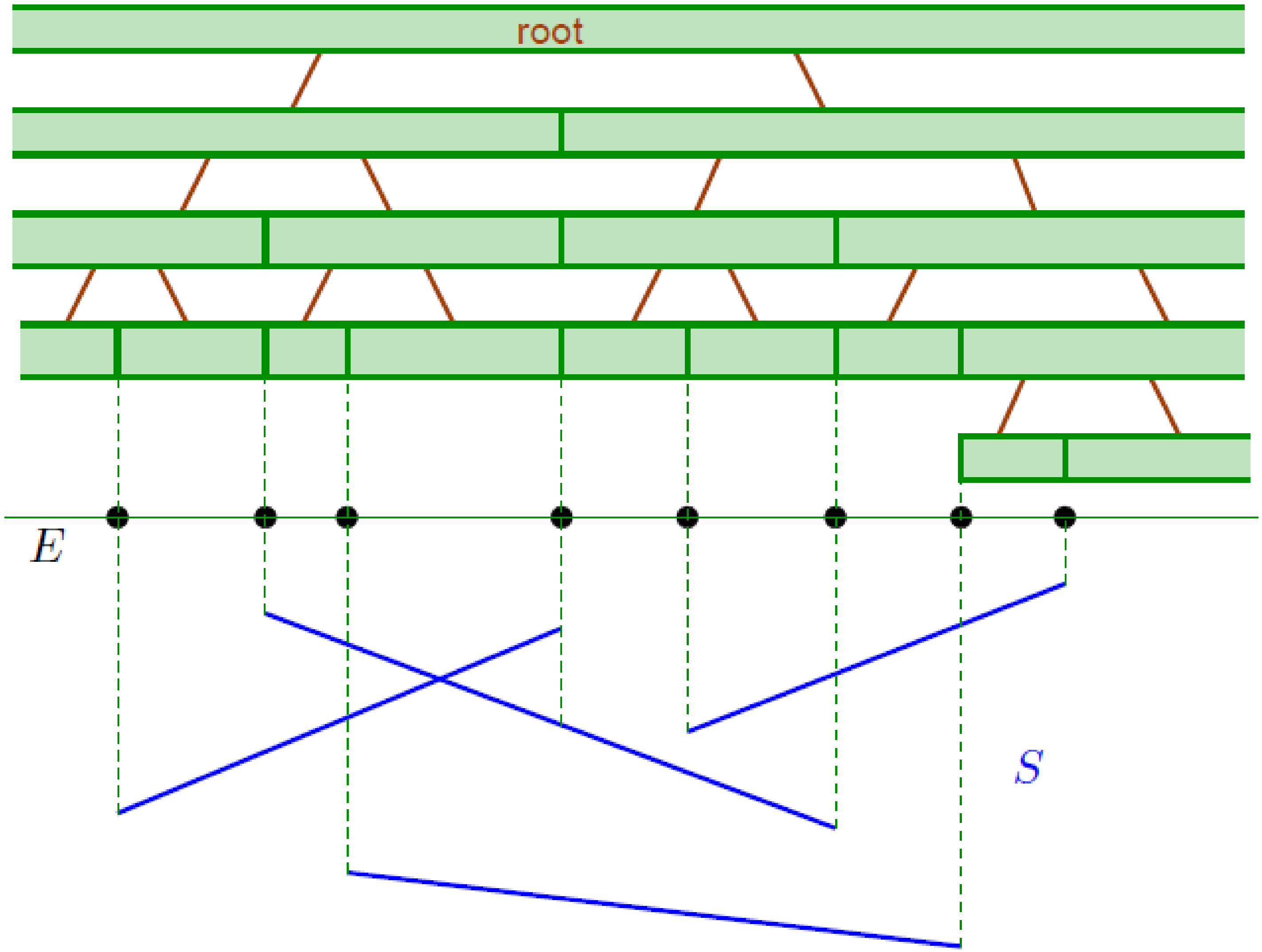
- Creating balanced binary tree containing intervals, root represents whole real axis
  - In each step set of atomic intervals is split into two sets with equal cardinality, one set is stored in left subtree, second in right subtree

```
struct SegmentTreeNode
{
	float istart;
	float iend;
	List L;
	IntervalTreeNode* left;
	IntervalTreeNode* right;
}
```

```
struct SegmentTree
{
	SegmentTreeNode* root;
}
```

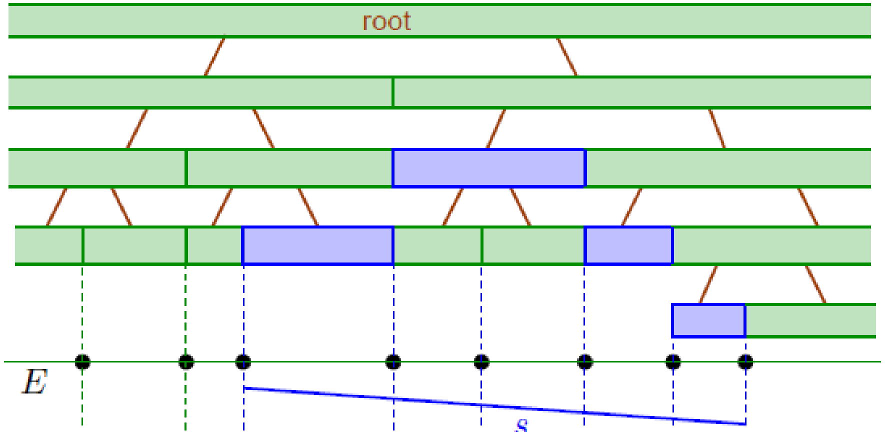
```
SearchTreeNodeConstruct (E)
{
	if (|E| == 0) return NULL;
	v = new SegmentTreeNode;
	n = |E|;m = n / 2;
	(L, R) = Split(E, m);
	v->istart = E.get(1);
	v->iend = E.get(n);
	v->left = SearchTreeNodeConstruct(L);
	v->right = SearchTreeNodeConstruct(R);
	return v;
}
```

Search tree



Segment tree

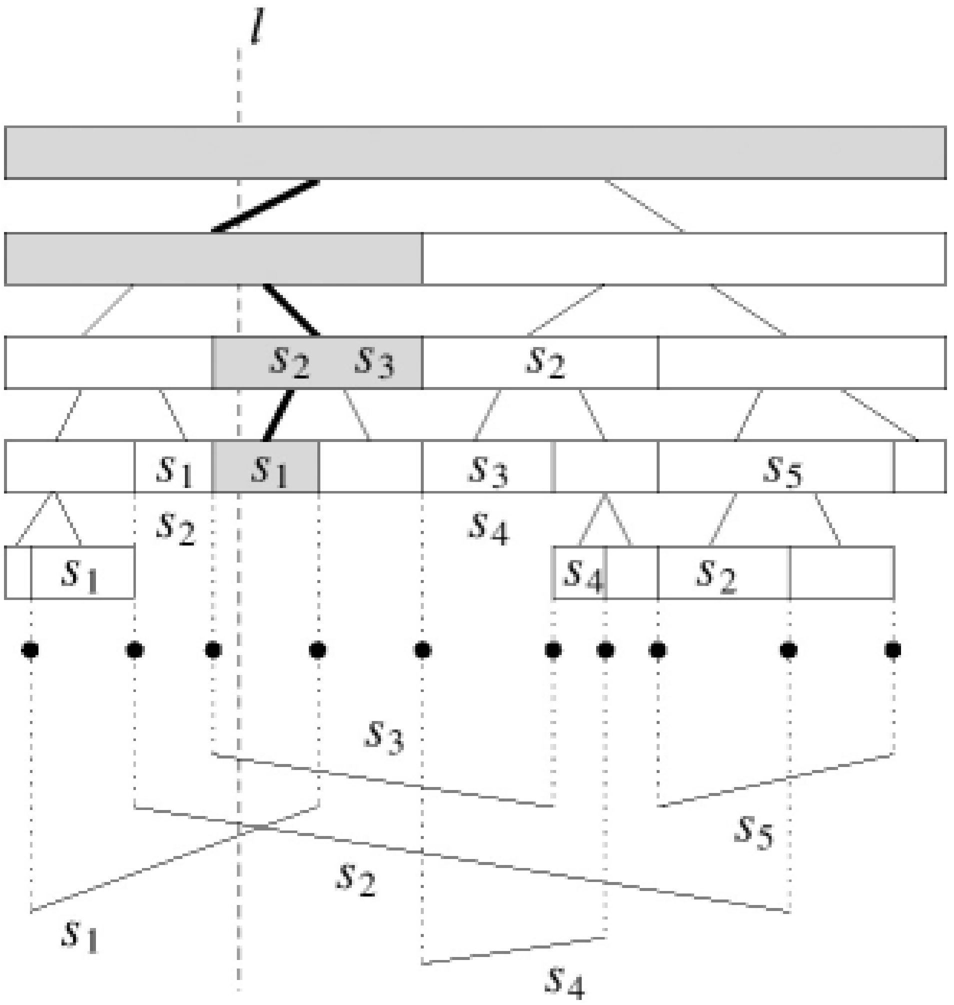
- Small atomic intervals stored in binary tree structure = search tree, where each node represents interval
- Each given segment is stored in nodes of search tree such that each segment is covered with minimal number of node intervals
- Construct: O(n.log(n))
- Memory: O(n.log(n))



Constructing segment tree

```
InsertSegment(v, s)
{
        if (v == NULL) return;
        if ([v->istart, v->iend] ∩ s == 0) return;
        if ([v->istart, v->iend] ⊂ s)
        {
                v->L.add(s);
        }
        else
        {
                InsertSegment(v->left, s);
                InsertSegment(v->right, s);
        }
}
```

```
BuildSegmentTree(S)
{
        Sx = S.SortX();
        Sx.ExtendInfinity();
        T = new SegementTree;
        T->root = SearchTreeNodeConstruct(Sx);
        while (S.first != 0)
        {
                InsertSegment(T->root, S.first);
                S.deleteFirst();
        }
}
```



Query

- For n segments in plane, querying all segments intersecting given vertical line has time complexity O(k + log(n)), where k is cardinality of result

```
StabbingQuery(T, l)
{
        Return StabbingQueryNode(T->root, l);
}
```

```
StabbingQueryNode(v, l)
{
        List L;
        If (v != NULL && v->istart <= l && l <= v->iend])
        {
                L.add(v.L);
                L1 = StabbingQueryNode(v->left, l);
                L2 = StabbingQueryNode(v->right, l);
                L.add(L1);
                L.add(L2);
        }
        return L;
}
```

Properties

- Sorting end point of intervals O(n.log(n))
- Constructing search treeO(n)
- Inserting one segment O(log(n))
- Inserting n segments O(n.log(n))
- Memory complexity O(n.log(n))

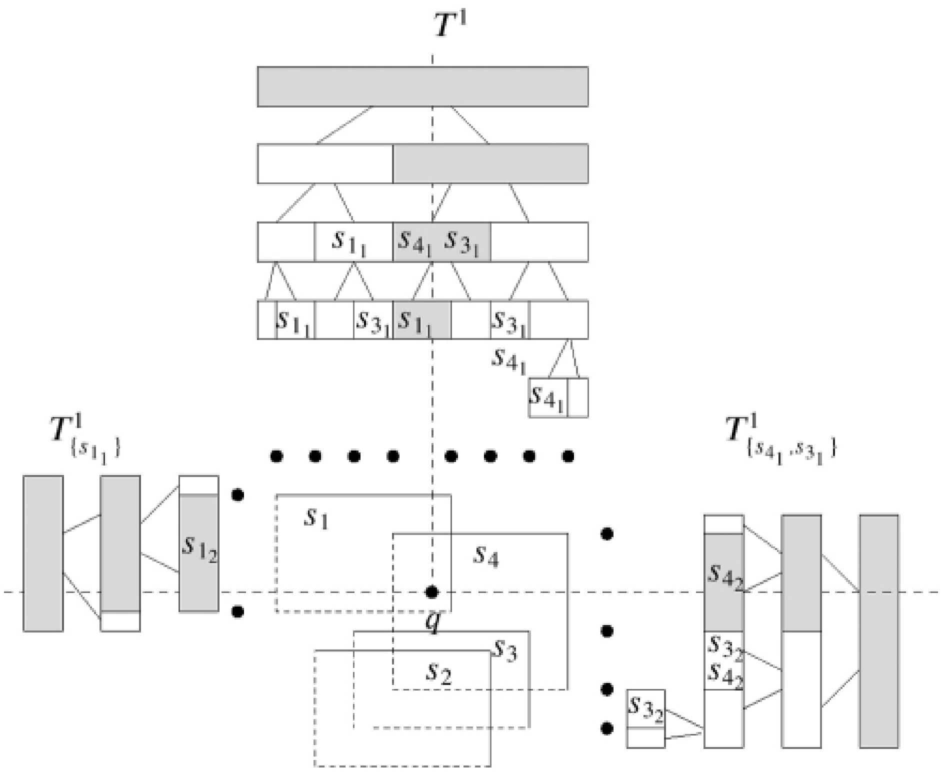
Multi-dimensional segment trees

- Input: set of d-dimensional intervals *S* parallel to coordinate axis, point *q* from R^d
- Output: set of intervals of *S* that contain point *q*
- In each node of d-dimensional segment tree can be stored d-1 dimensional segment tree

```
MLSegmentTree(B, d)
{
	S = B.FirstSegmentsExtract;
	T = SegmentTreeConstruct(S);
	T.dim = d;
	if (d > 1)
	{
		N = T.GetAllNodes;
		while (|N| != 0)
		{
			u = N.First;
			N.DeleteFirst;
			L = u->L;
			List B;
			while (|L| != 0)
			{
				s = L.First;
				L.DeleteFirst;
				B.add(s.Box(d − 1));
			}
			u->tree = MLSegmentTree(B, d − 1);
		}
	}
	return T;
}
```

Multi-dimensional segment tree

- Construction time: $O(n.\log^d(n))$
- Memory complexity: $O(n.\log^d(n))$
- Query: $O(k + \log^d(n))$



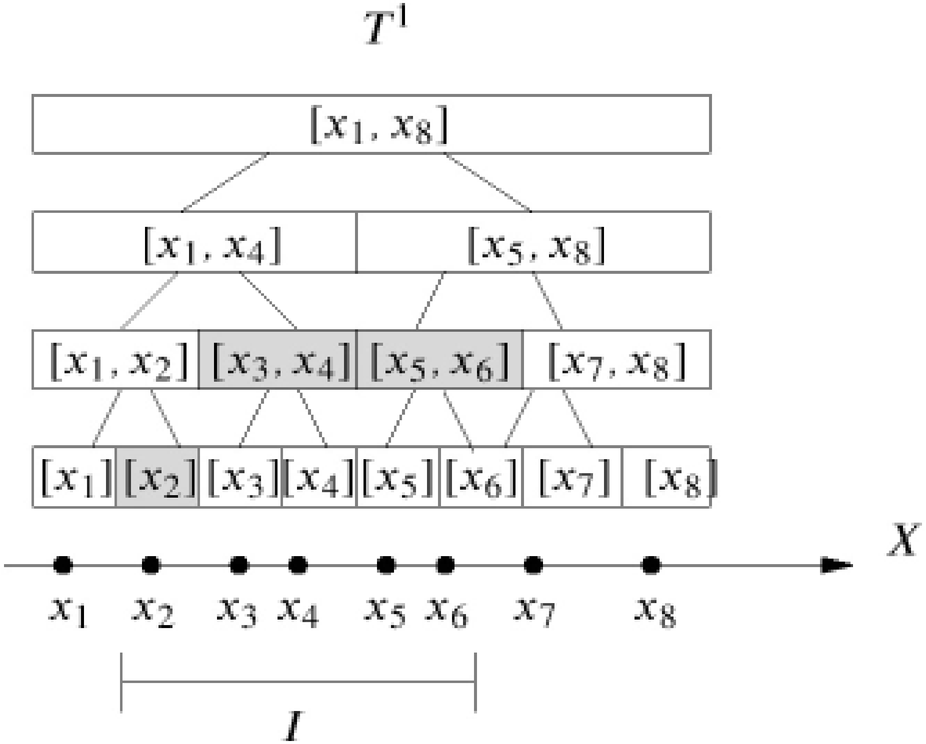

```
MLSegmentTreeQuery(T, q, d)
{
        if (d == 1)
        {
                L = StabbingQuery(T, q);
                return L;
        }
        else
        {
                List A;
                L = SearchTreeQuery(T, q.First);
                while (|L| != 0)
                {
                        t = (L.First)->tree;
                        B = MLSegmentTreeQuery(t, q.Rest, d − 1);
                        A.ListAdd(B);
                        L.DeleteFirst;
                }
                return A;
        }
}
```

Range tree

- Input: points set *S* in R^d, usually $d \geq 2$
- Query: d-dimensional interval *B* from R^d parallel with coordinate axis
- Output: Set of points from *S*, that are in *B*
- Similar method to segment trees – creation of search tree based on points coordinates
- One dimensional case – finding all points inside interval

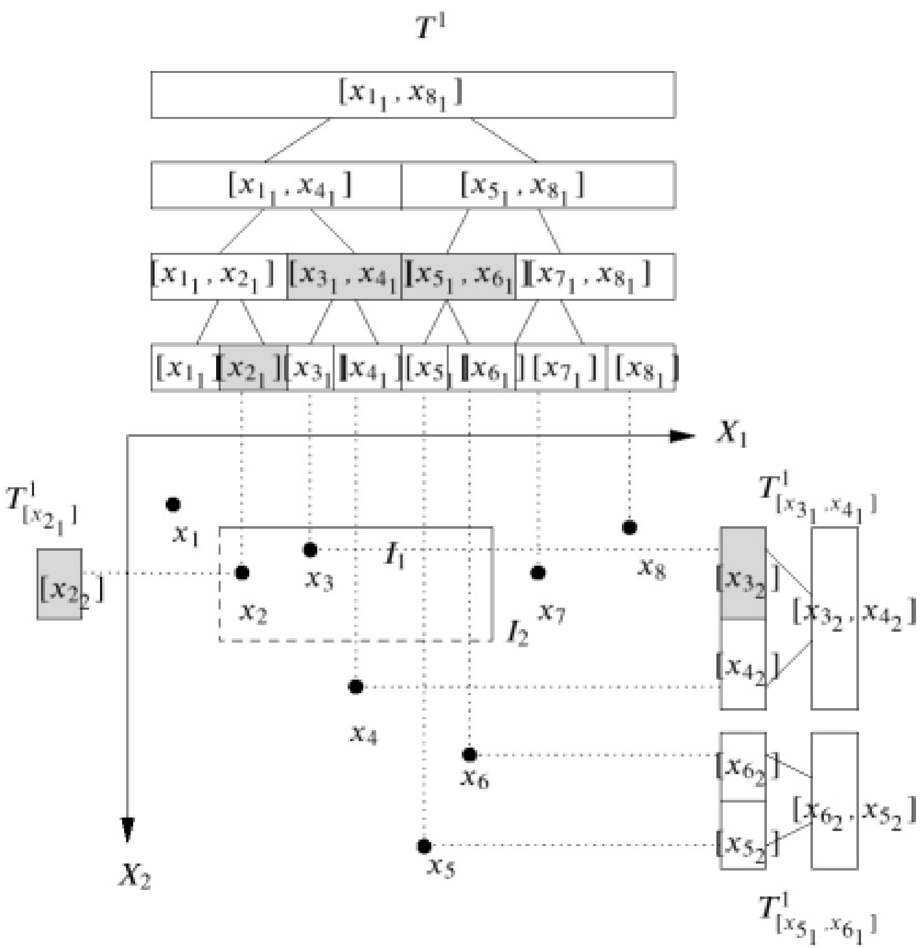
Range tree

- Creation of search tree
- In each tree node, interval of points in leaves of node subtree is stored



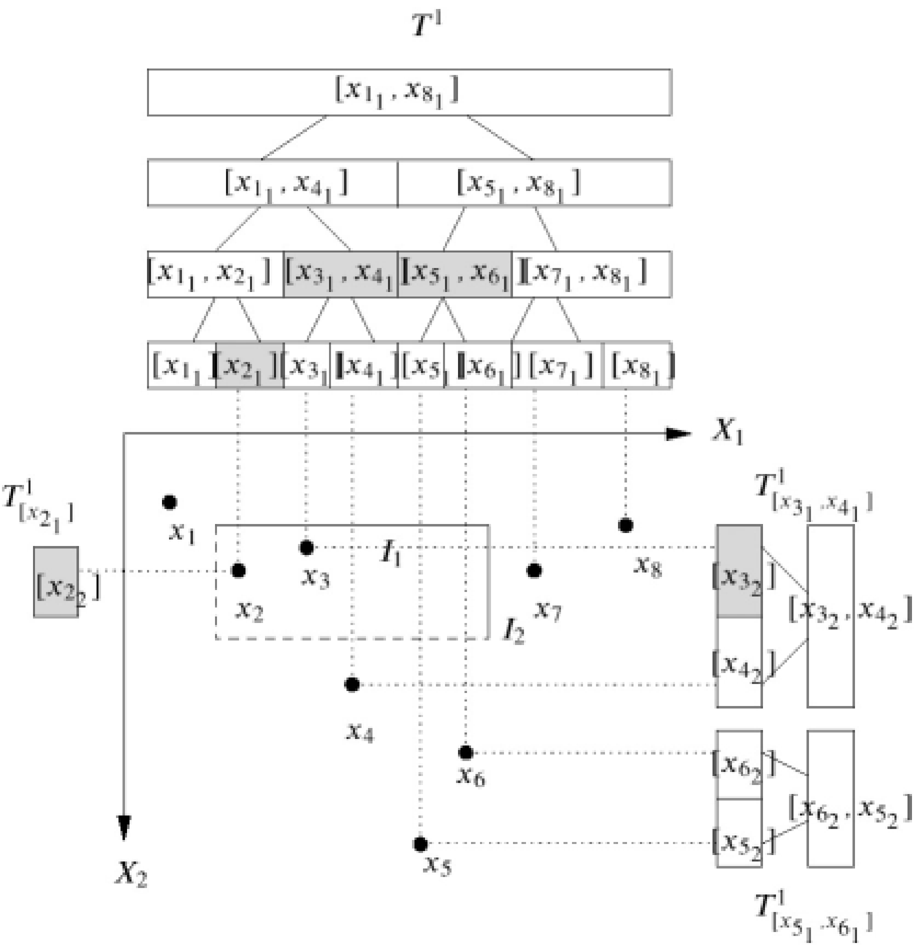
Range tree

- Node of d-dimensional range tree can contain d-1 dimensional range tree




```
RangeTreeConstruct(S, d)
{
        Sf  = S.FirstCoordElements();
        Sf.Sort;
        T = SearchTree(Sf);
        T->dim = d;
        if (d > 1)
        {
                N = T->GetAllNodes();
                while (|N| != 0)
                {
                        u = N.PopFirst;
                        L = u->GetAllPoints();
                        List D;
                        while (|L| != 0)
                        {
                                x = L.PopFirst;
                                D.add(x.Point(d – 1));
                        }
                        u->tree = RangeTreeConstruct(D, d – 1);
                }
        }
        return T;
}
```

Range trees



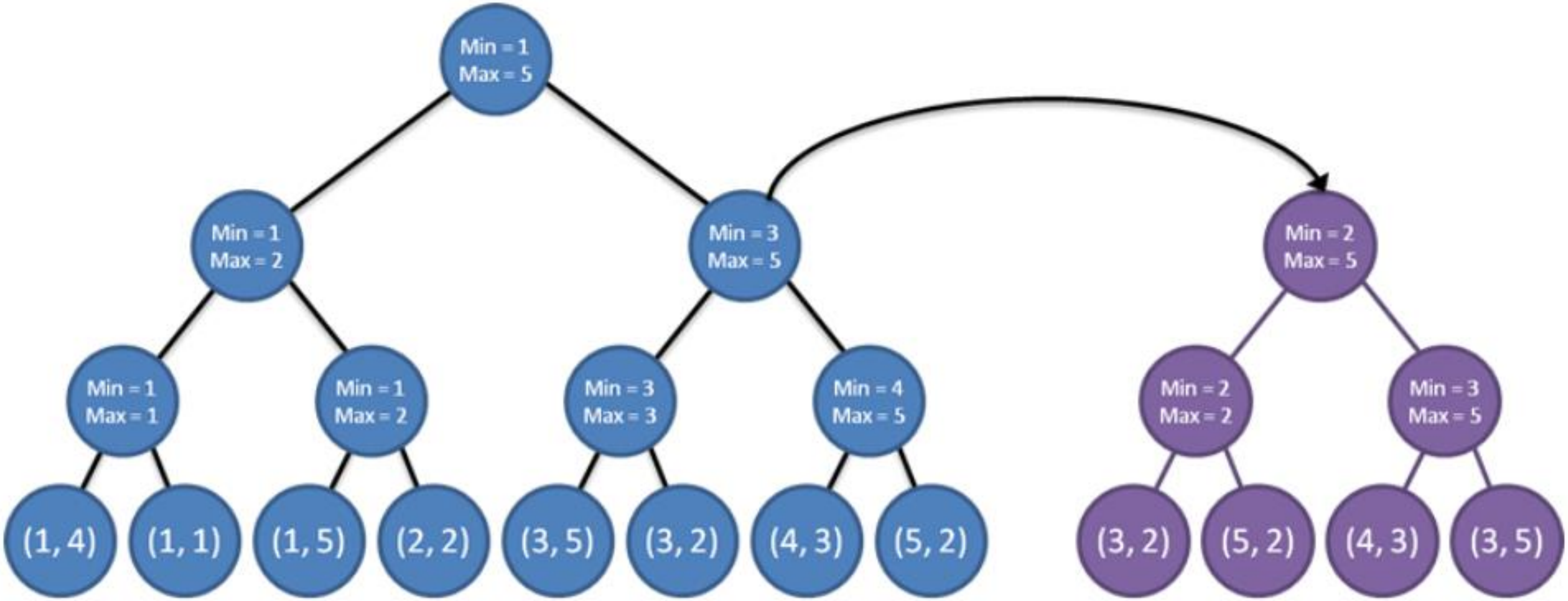

```
RangeTreeNodeQuery(v, B, d)
{
        List result;
        if (d == 0)
           return result;
        (min, max) = GetRangeInDimension(B, d);

        if (v->min > max || v->max < min)
           return result;
        If (v->IsLeaf() && v->point in B)
           result.Add(v->point);
        else if (min <= v->min && max >= v->max)
           result.Add(RangeTreeNodeQuery(v->tree, B, d-1));
        else
        {
           result.Add(RangeTreeNodeQuery(v->left, B, d));
           result.Add(RangeTreeNodeQuery(v->right, B, d));

        }
        return result;
}
```

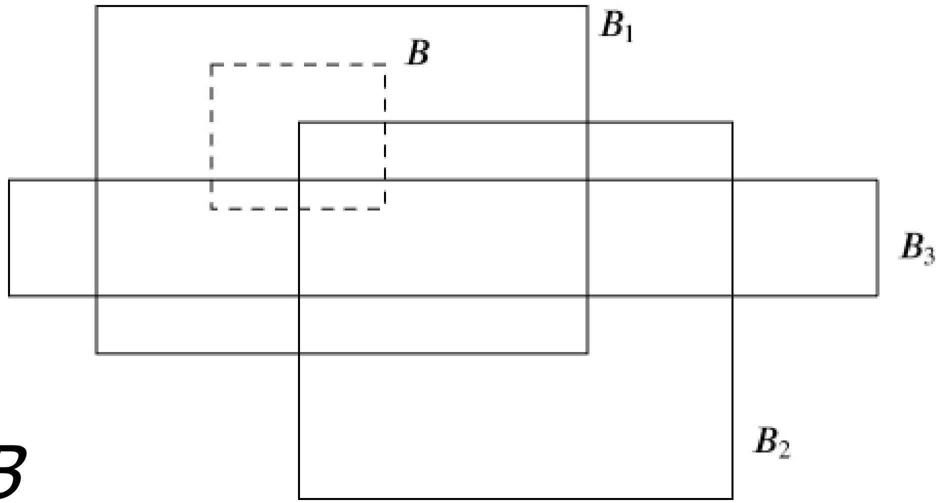
Range trees

- Construction: $O(n.\log^{d-1}(n))$
- Memory: $O(n.\log^{d-1}(n))$
- Query: $O(k + \log^{d}(n))$



AABB/AABB

- Input: set S of 2D boxes (intervals), one other 2D interval B
- Output: Intervals from S that have non-empty intersection withs B
- Three cases:



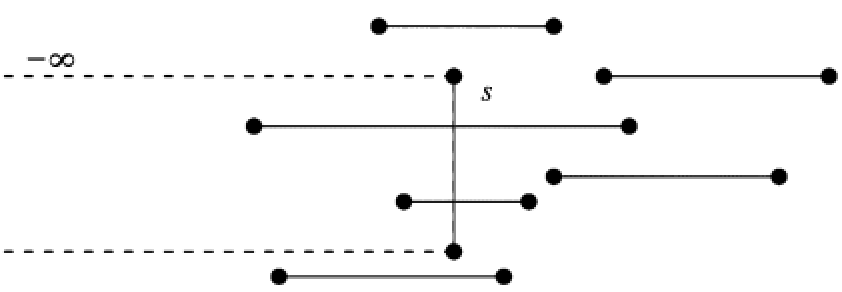
  - B is inside B_i
  - Corner of B_i is inside B
  - Side of B_i intersect B a no corner of B_i is in B

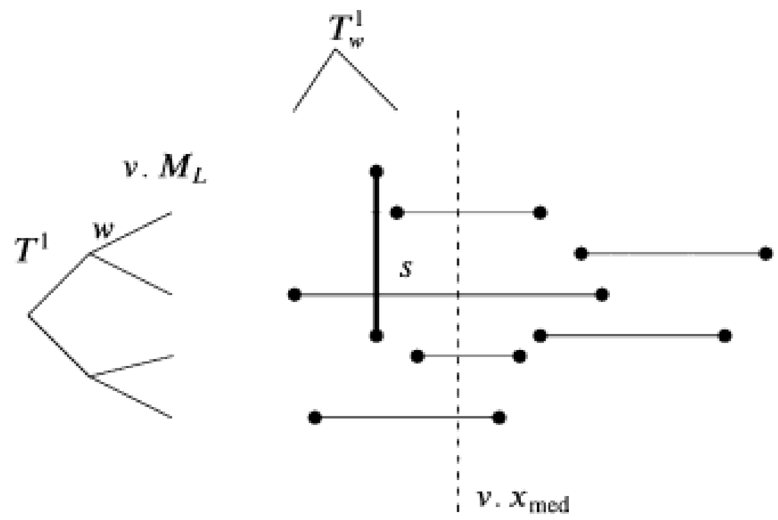
AABB/AABB

- Case 1. – 2D segment tree. Querying all 2D intervals B_i from S that contain four corners of B
  - time $O(k_1 + \log^2(n))$ and memory $O(n + \log^2(n))$
- Case 2. – Range tree in 2D. Querying corners of intervals B_i that are inside interval B
  - time $O(k_2 + \log^2(n))$ and memory $O(n.\log(n))$.
- Case 3. – New query:
  - Input: Set S of horizontal segments in 2D
  - Query: Vertical segment s in 2D
  - Output: All segments from S intersecting s
  - + rotation 90º

Case 3 - construction

- Combination of interval and range tree
- Creating interval tree from horizontal borders of B_i
- M_l a M_r in each node of interval tree are replaced by 2D range trees
- Memory: O(n.log(n))





Case 3 - query

- In x direction searching in set of horizontal borders of intervals from *S* that contain x coordinate value of *s* – result is node of interval tree *v*
- In node *v* searching M_l resp. M_r (these are 2D segment trees) based on x and y coordinate values of *s*
- Time $O(k_3 + \log^2(n))$

Questions?