Search trees, k-d tree

Marko Berezovský
Radek Mařík
PAL 2012
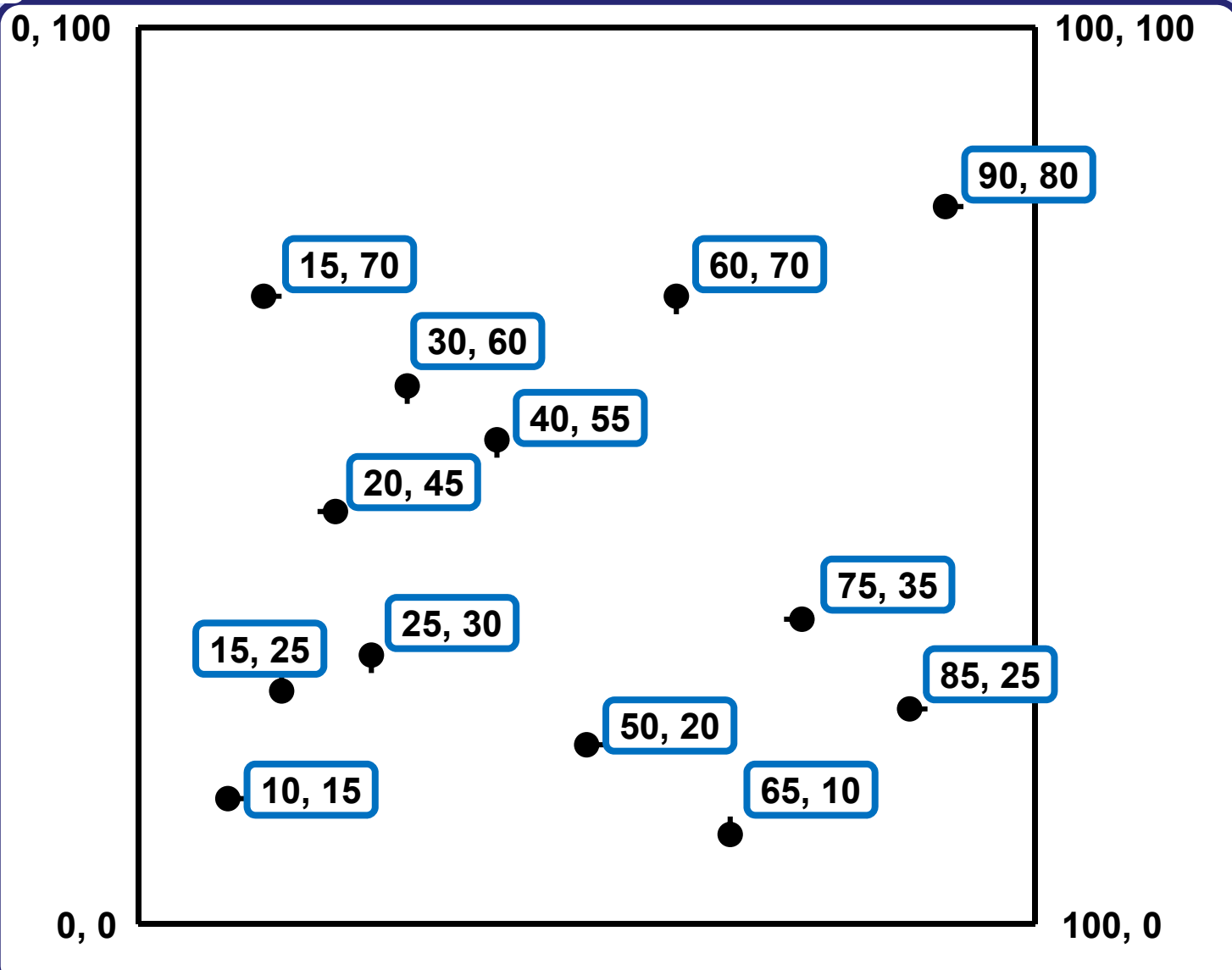
To read

Dave Mount: CMSC 420: D*ata Structures1 Spring 2001*, Lessons 17&18.

http://www.cs.umd.edu/~mount/420/Lects/420lects.pdf

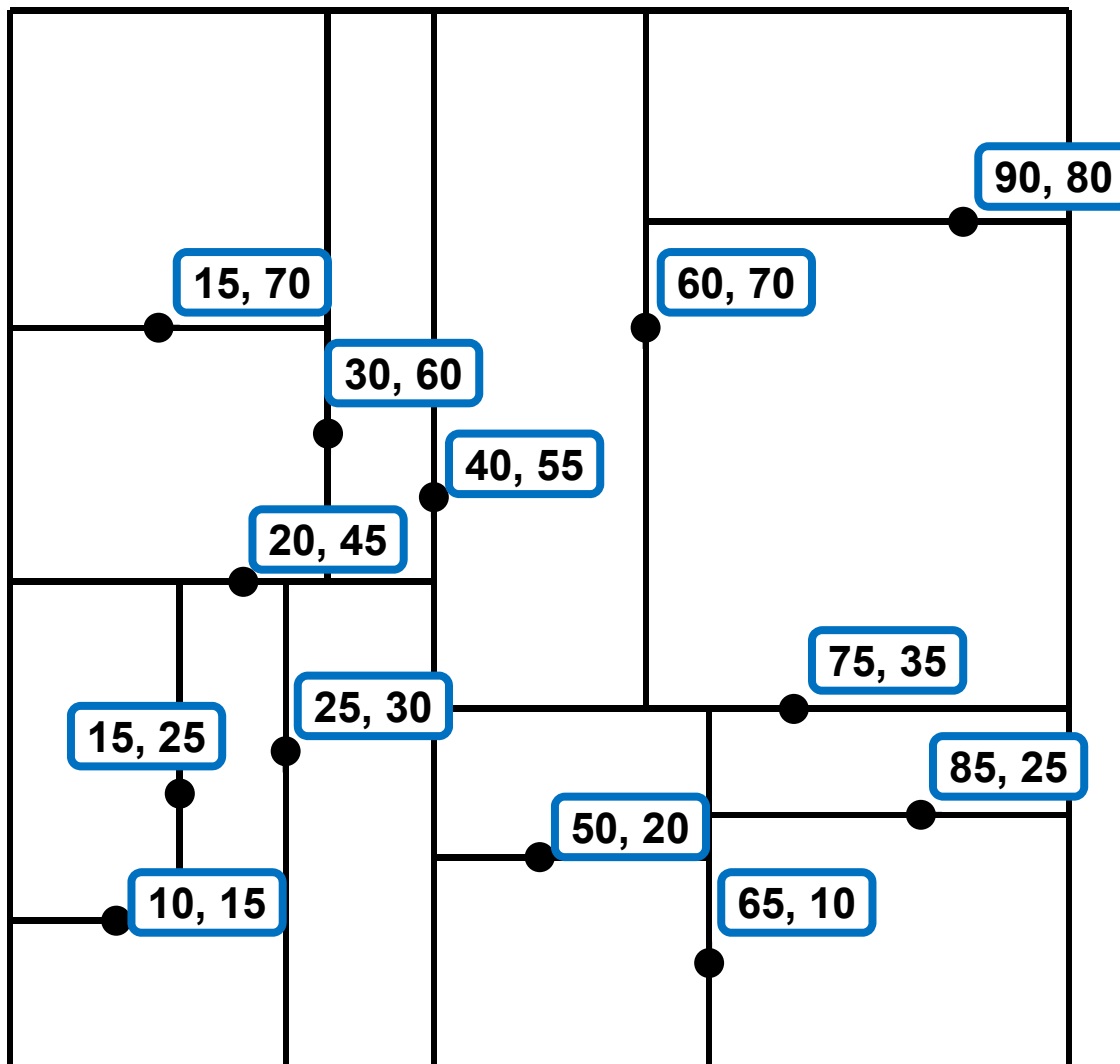Hanan Samet: *Foundations of multidimensional and metric data structures*, Elsevier, 2006, chapter 1.5.

http://www.amazon.com/Foundations-Multidimensional-Structures-Kaufmann-Computer/dp/0123694469

See PAL webpage for references

Points in plane in general position are given, suppose no two are identical.

Scheme of area divison exploited in k-d tree.

K-d tree is a binary search tree representing a rectangular area in D-dimensional space. The area is divided (and recursively subdivided) into rectangular cells. Denote dimensions naturaly by their index 0, 1, 2, ... D−1.

Denote R root of a tree or subtree.
A rectangular D-dimensional cell C(R) (hyperrectangle) is associated with R.
Let R coordinates be R[0], R[2], ..., R[D−1] and its depth in tree h.
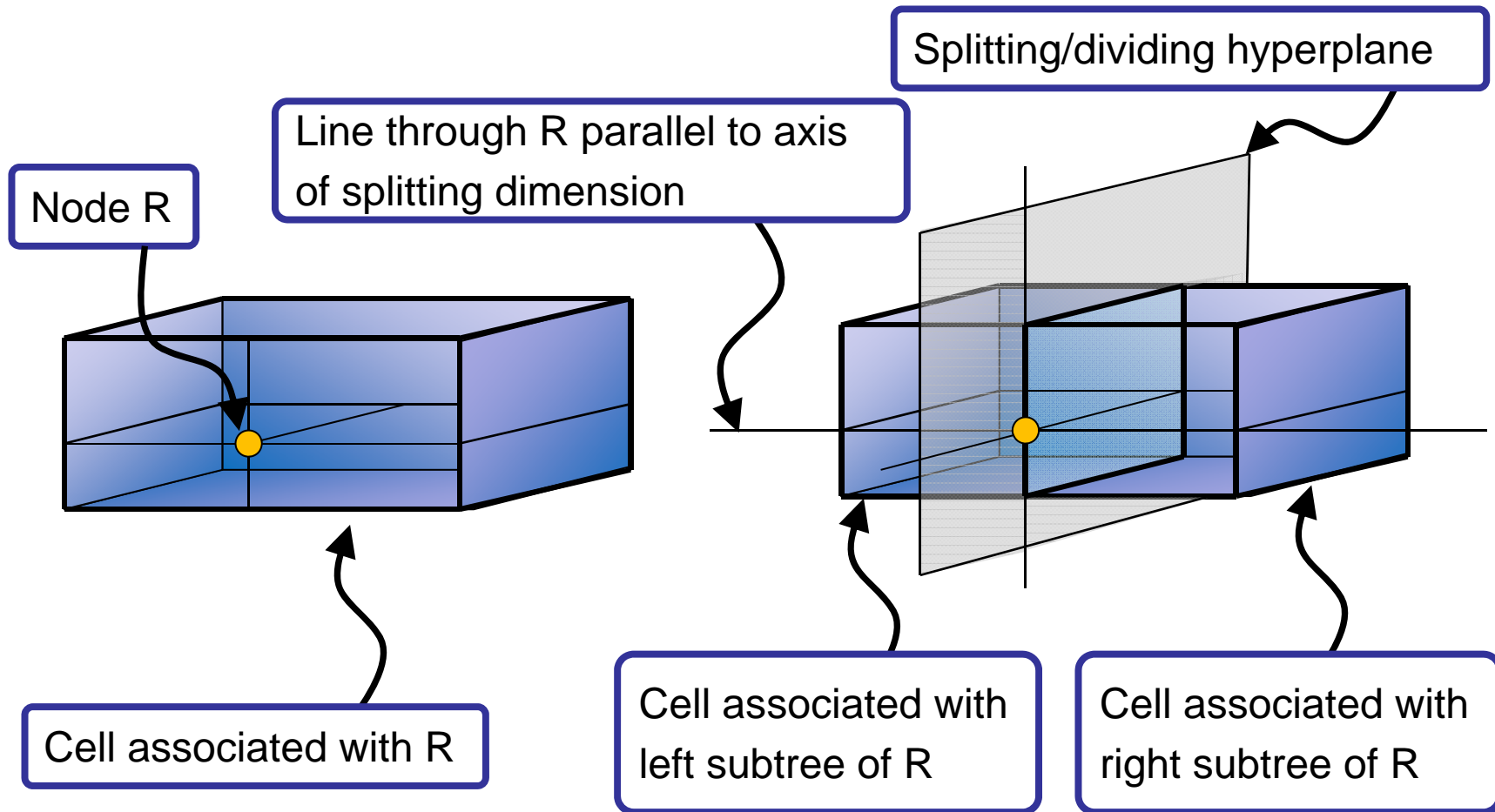The cell C(R) is splitted into two subcells by a hyperplane of dim D−1,
for all which points y holds y[h%D] = R[h%D].

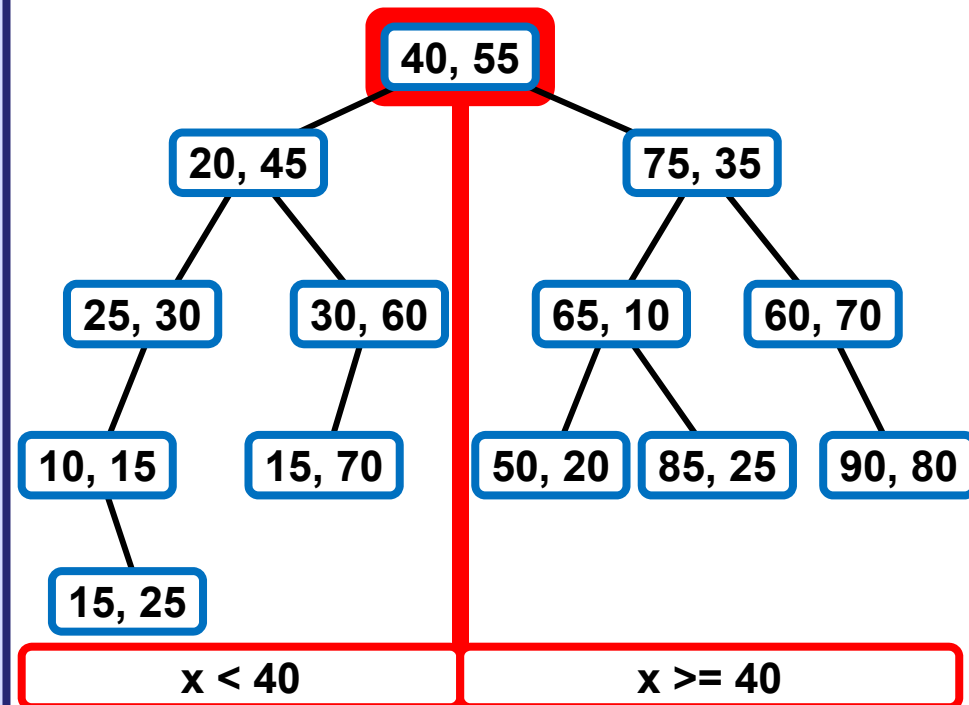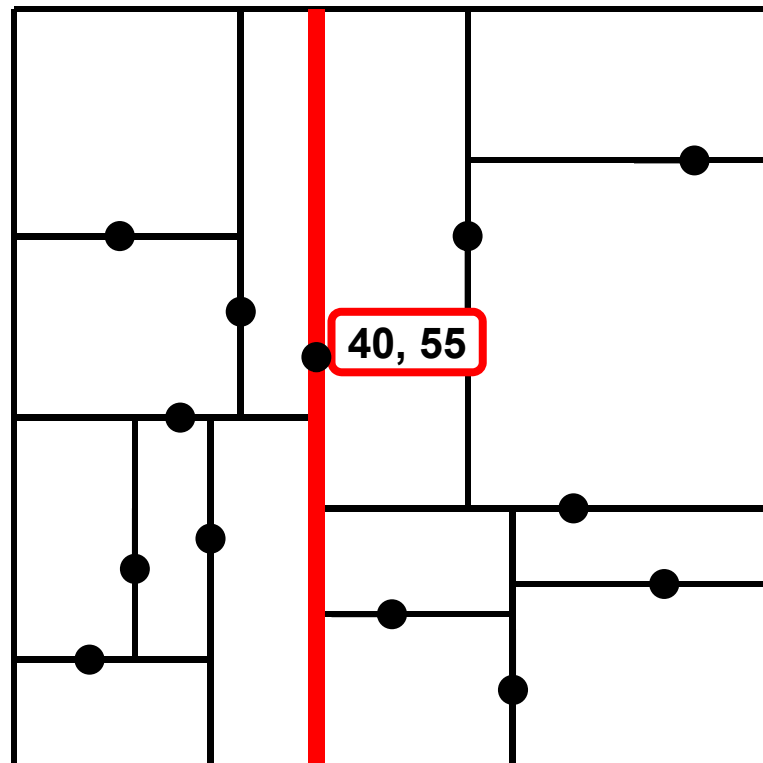All nodes in left subtree of R are characterised by their (h%D)-th coordinate being less than R[h%D].
All nodes in right subtree of R are characterised by their (h%D)-th coordinate being greater than or equal to R[h%D].

Let us call value h%D splitting /cutting dimension of a node in depth h.

Note that k-d tree presented here is basic simple variant, many others, more effective, more sophisticated variants do exist.

Splitting/dividing hyperplane

Line through R parallel to axis of splitting dimension

Node R

Cell associated with R

Cell associated with left subtree of R
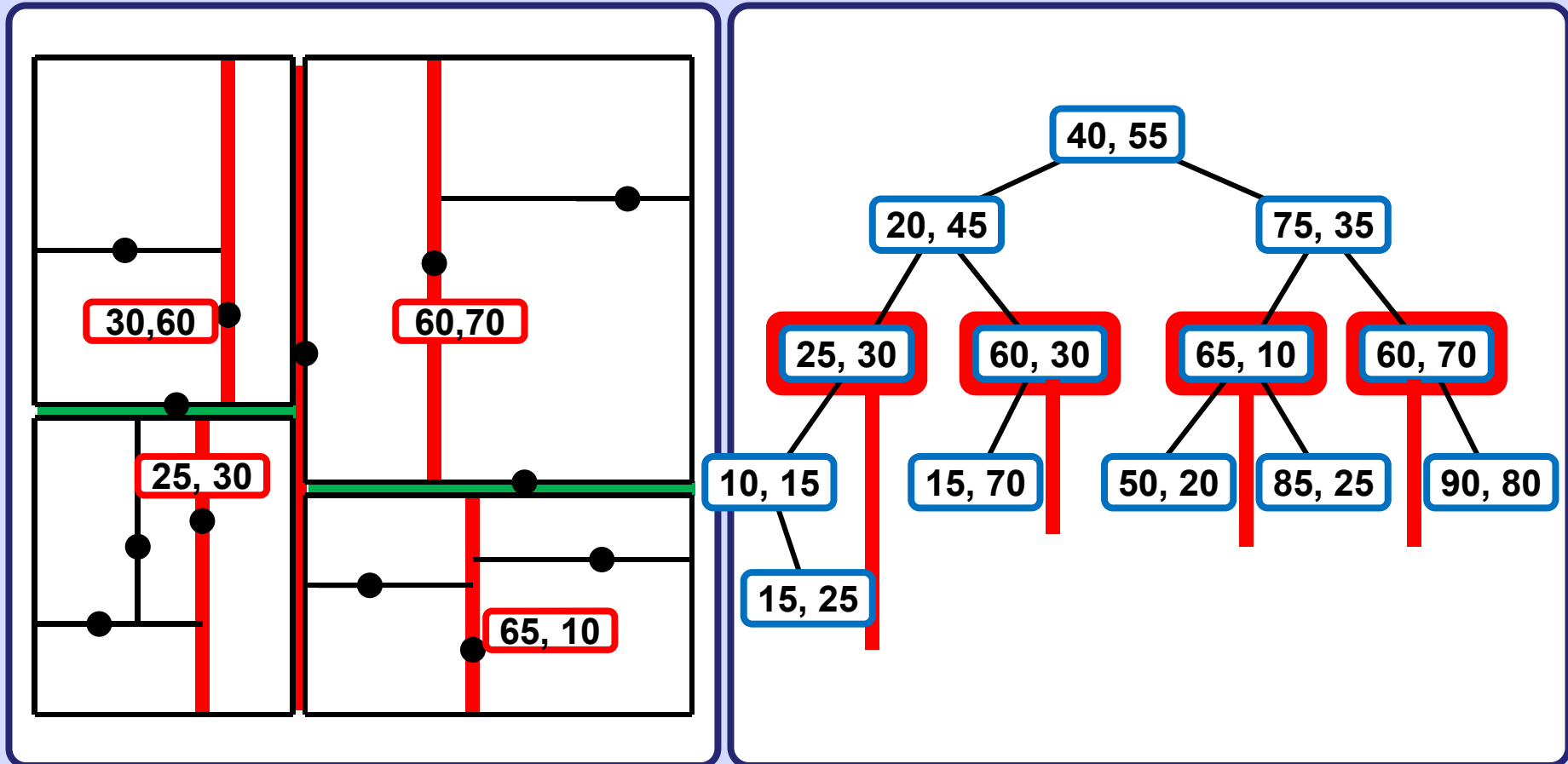
Cell associated with right subtree of R

Typically node R lies on the boundary of its associated cell
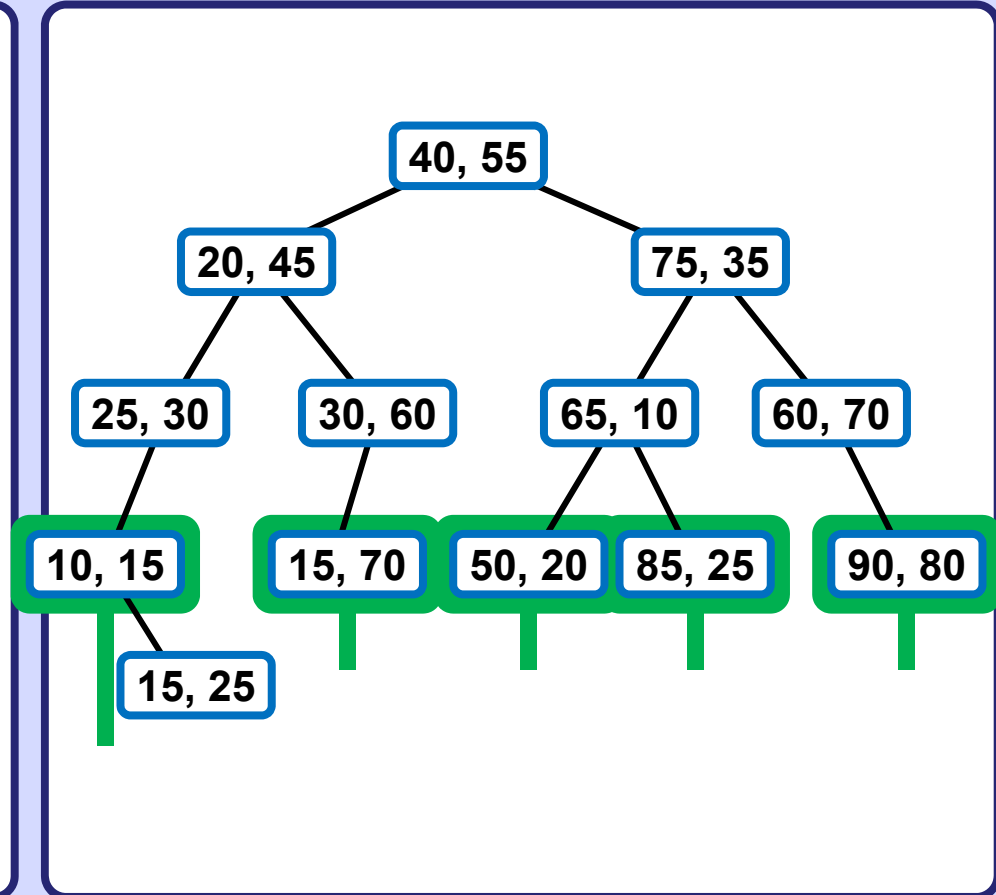
Scheme of area divison exploited in k-d tree.
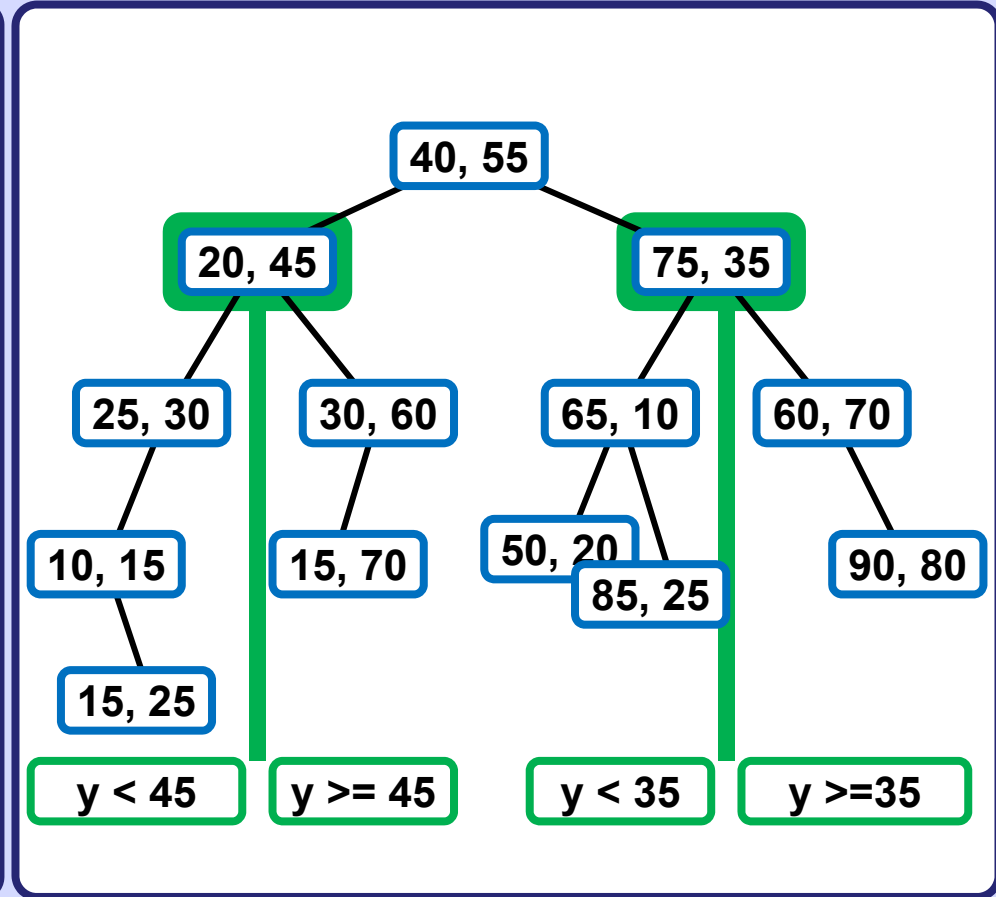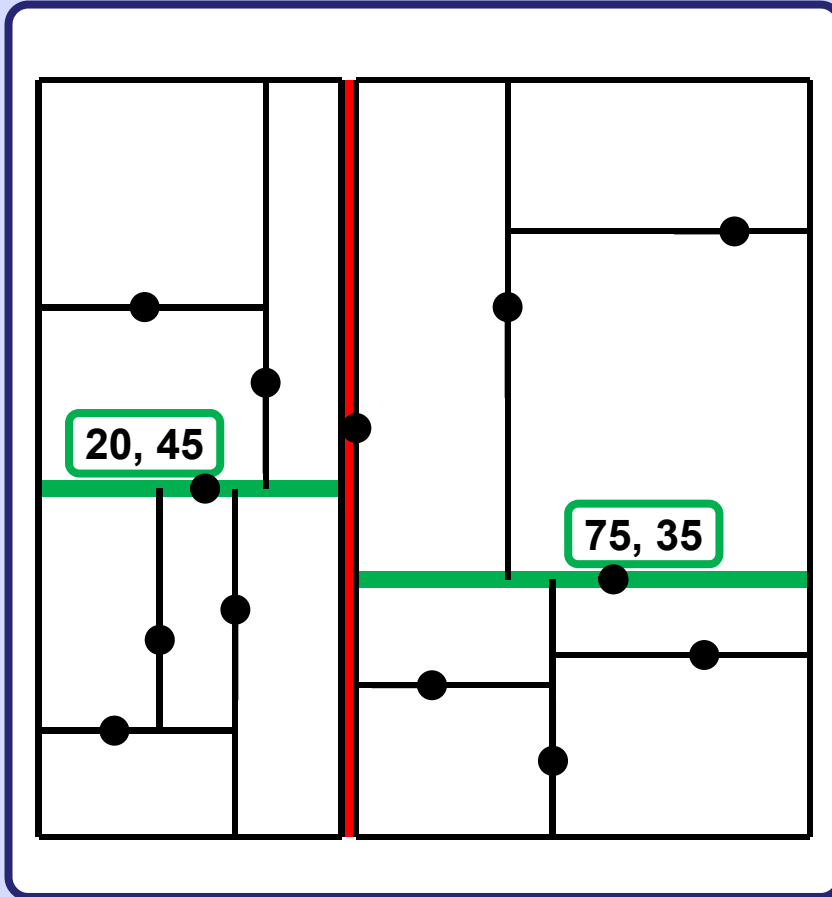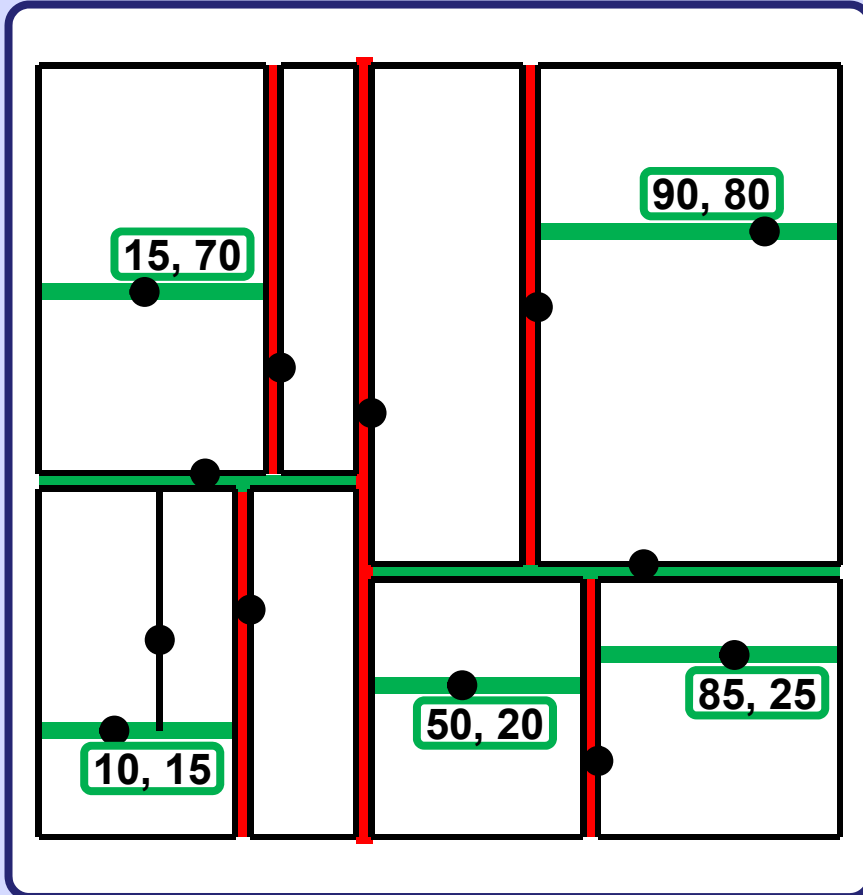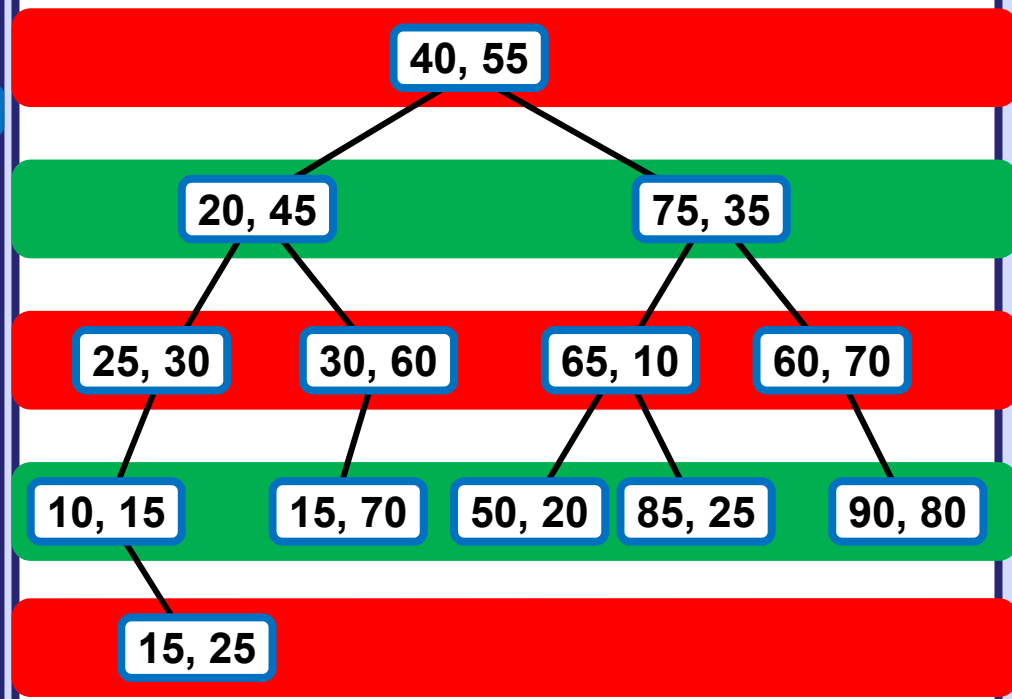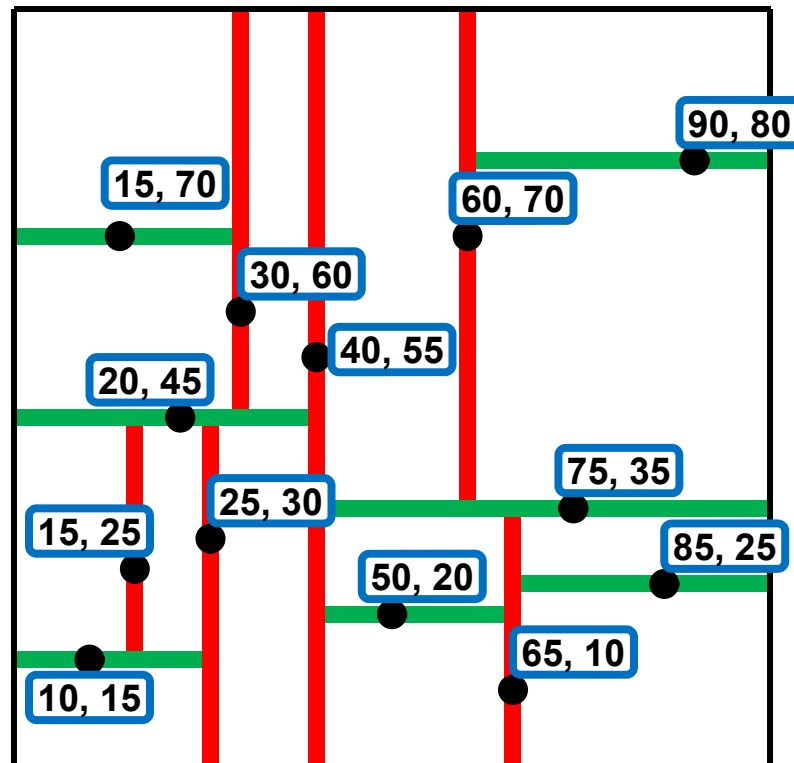
Scheme of area divison exploited in k-d tree.

Scheme of area divison exploited in k-d tree.

Scheme of area divison exploited in k-d tree.

Complete k-d tree with with marked area division.

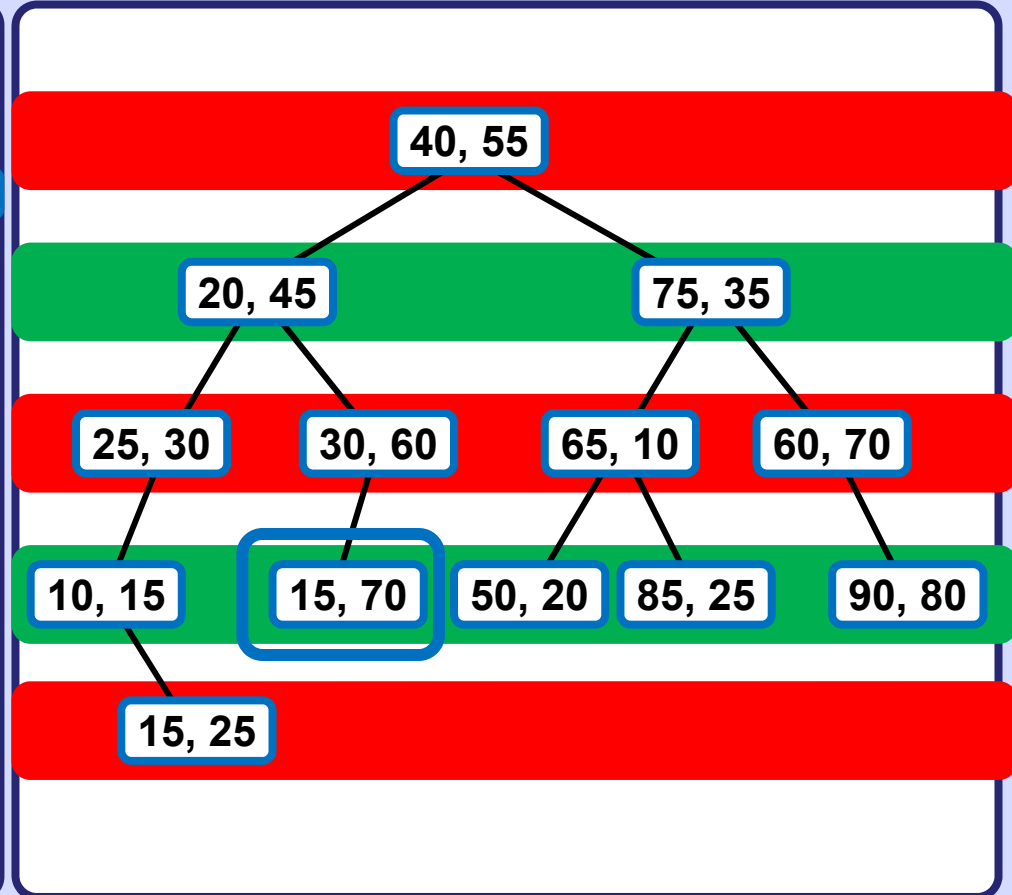Operation Find(key) Is analogous to 1D trees.

Let
Q[] = Q[0], Q[1], ..., Q[D−1] be the coordinates of the query point Q,
N[] = N[0], N[1], ..., N[D−1] be the coordinates of the current node N,
h be the depth of current node N.

If Q[] == N[] stop, Q was found
if Q[h%D] < N[h%D] continue search recursively in left subtree of N.
if Q[h%D] >= N[h%D] continue search recursively in right subtree of N.

Find [15, 70]
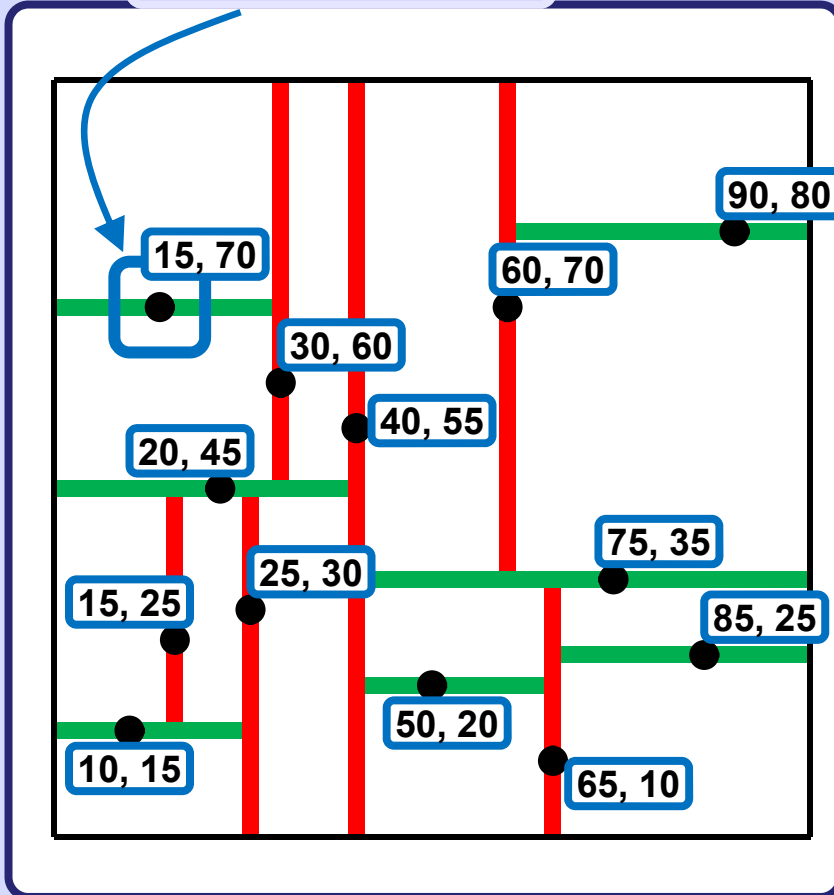
Operation Find works analogously as in other (1D) trees.

Note how cutting dimension along which the tree is searched alternates regularly with the depth of a node currently visited.

Find [15, 70]

15 < 40

40, 55

20, 45

75, 35

25, 30

30, 60

65, 10

60, 70

10, 15

15, 70

50, 20

85, 25

90, 80

15, 25

40, 55
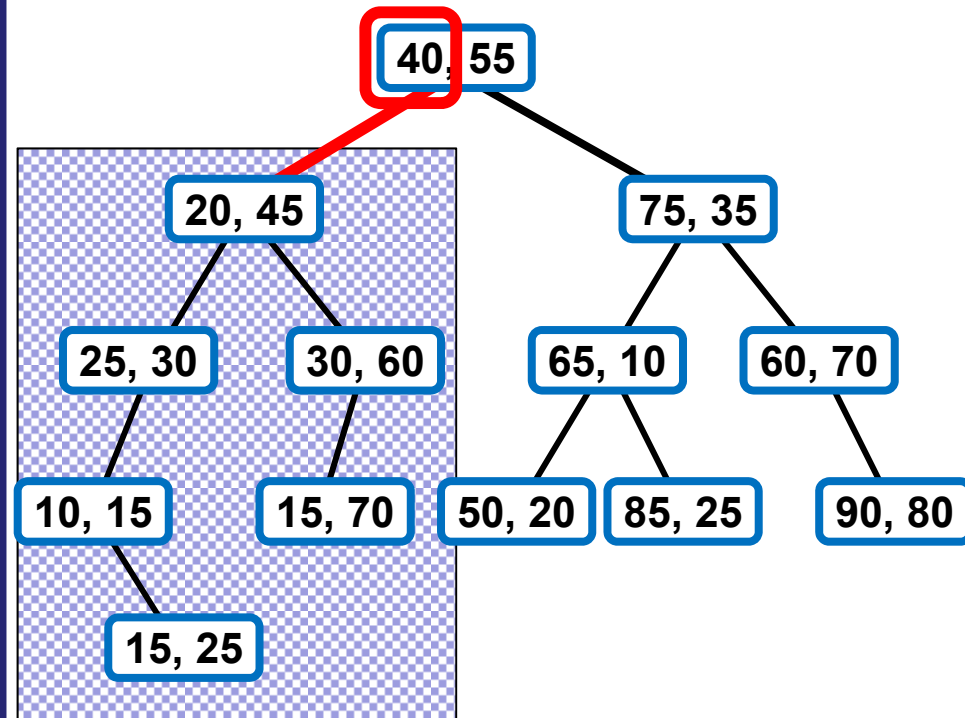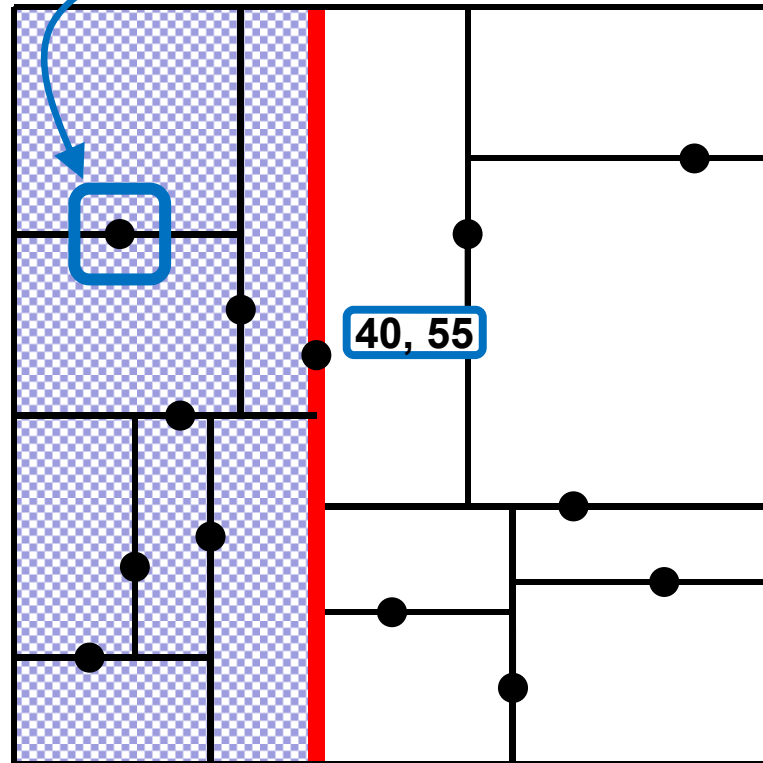
Operation Find works analogously as in other (1D) trees.

Search along x-dimension in depth 0. Compare x-coordinate of searched key

with x-coordinate of the node and either stop(found) or go L/R accordingly.

Find [15, 70]

70 >= 45

40, 55

20, 45

75, 35

25, 30

30, 60

65, 10

60, 70

10, 15

15, 70

50, 20

85, 25

90, 80

15, 25
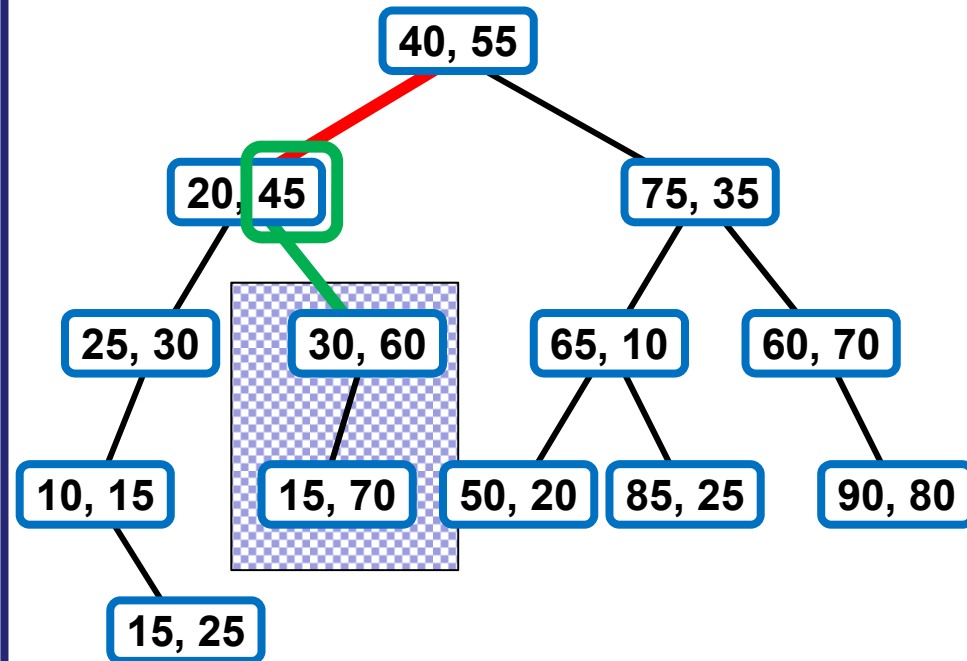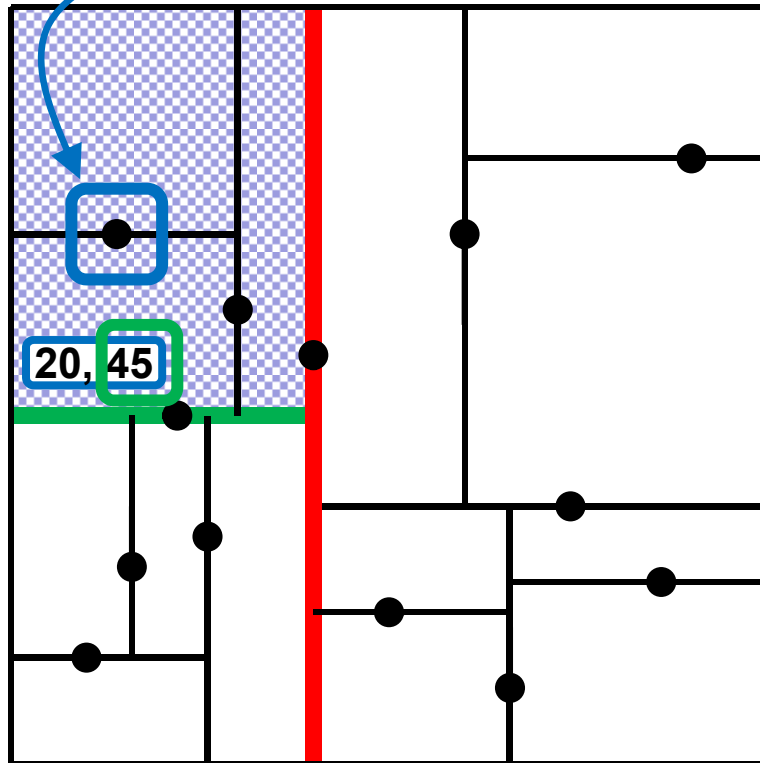
Operation Find works analogously as in other (1D) trees.

Search along y-dimension in depth 1.

Find [15, 70]

15 < 30

40, 55

20, 45   75, 35

25, 30   30, 60   65, 10   60, 70

10, 15   15, 70   50, 20   85, 25   90, 80

15, 25

30 60

Operation Find works analogously as in other (1D) trees.

Search along x-dimension in depth 2.

Found [15, 70]

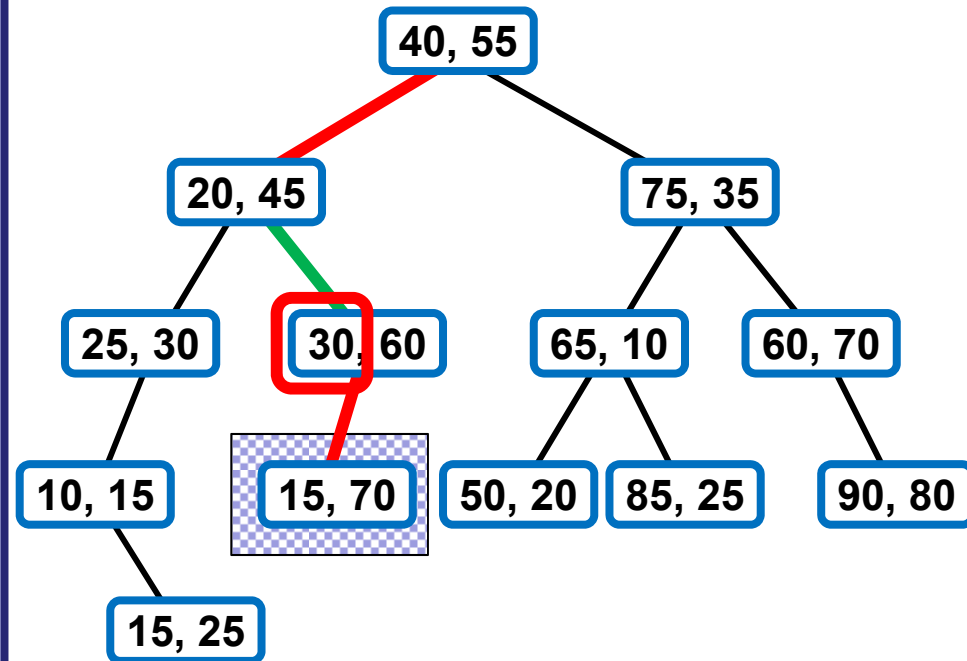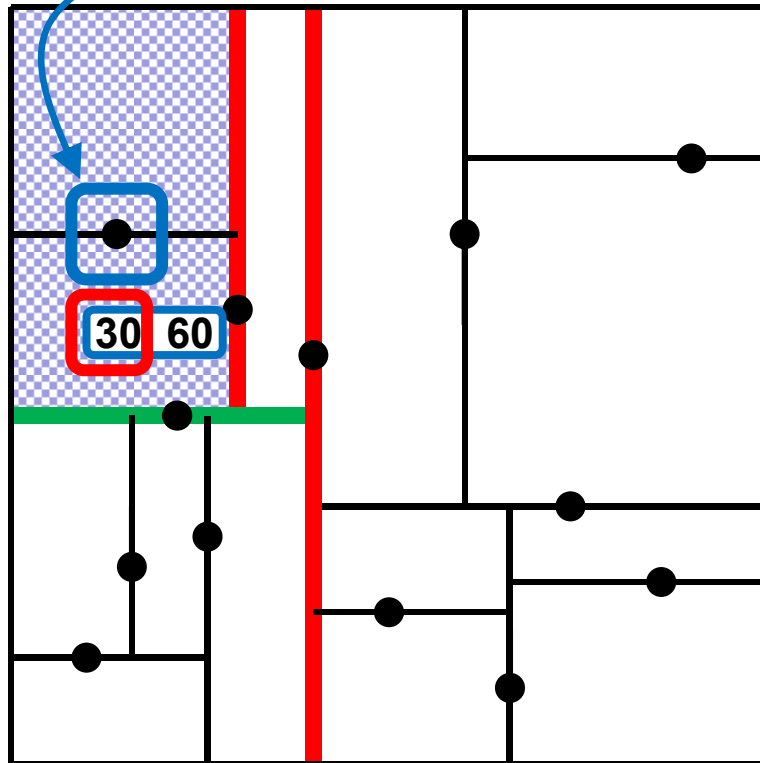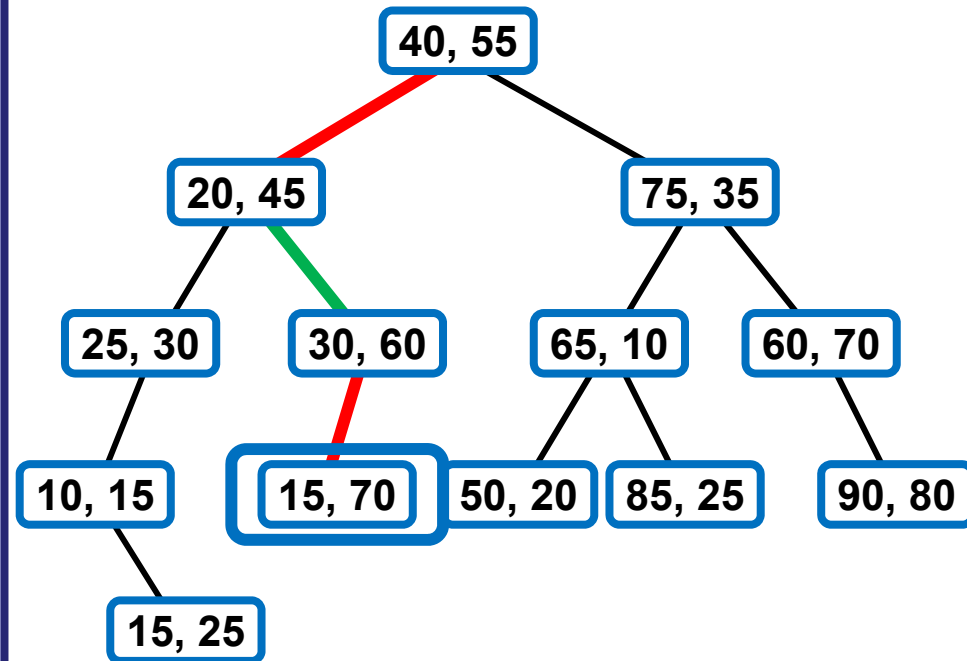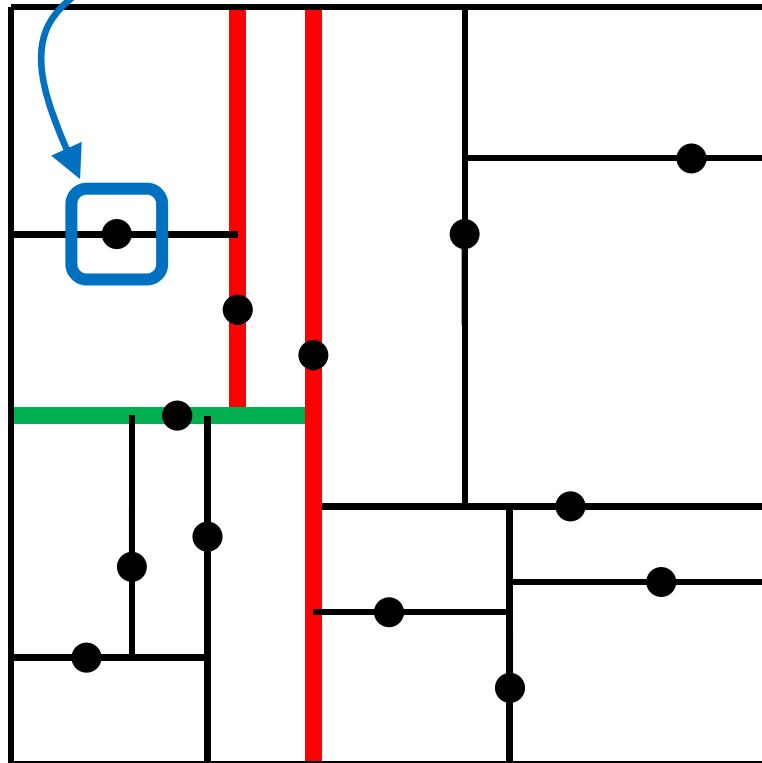
Operation Find works analogously as in other (1D) trees.

Search along y-dimension in depth 3, etc.

Operation Insert(point) is analogous to 1D trees.

Let   P[] = P[0], P[1], ..., P[D−1] be the coordinates of the inserted point P.
Perform search for P in the tree.
Let L[] = L[0], L[1], ..., L[D−1] be the coordinates of the leaf L which was
last node visited during the search. Let h be the depth of L.

Create node node N containing P as a key.
If P[h%D] <   L[h%D] set N as left child of L
If P[h%D] >= L[h%D] set N as right child of L

Insert [55, 30]

90, 80

15, 70

60, 70

30, 60

40, 55

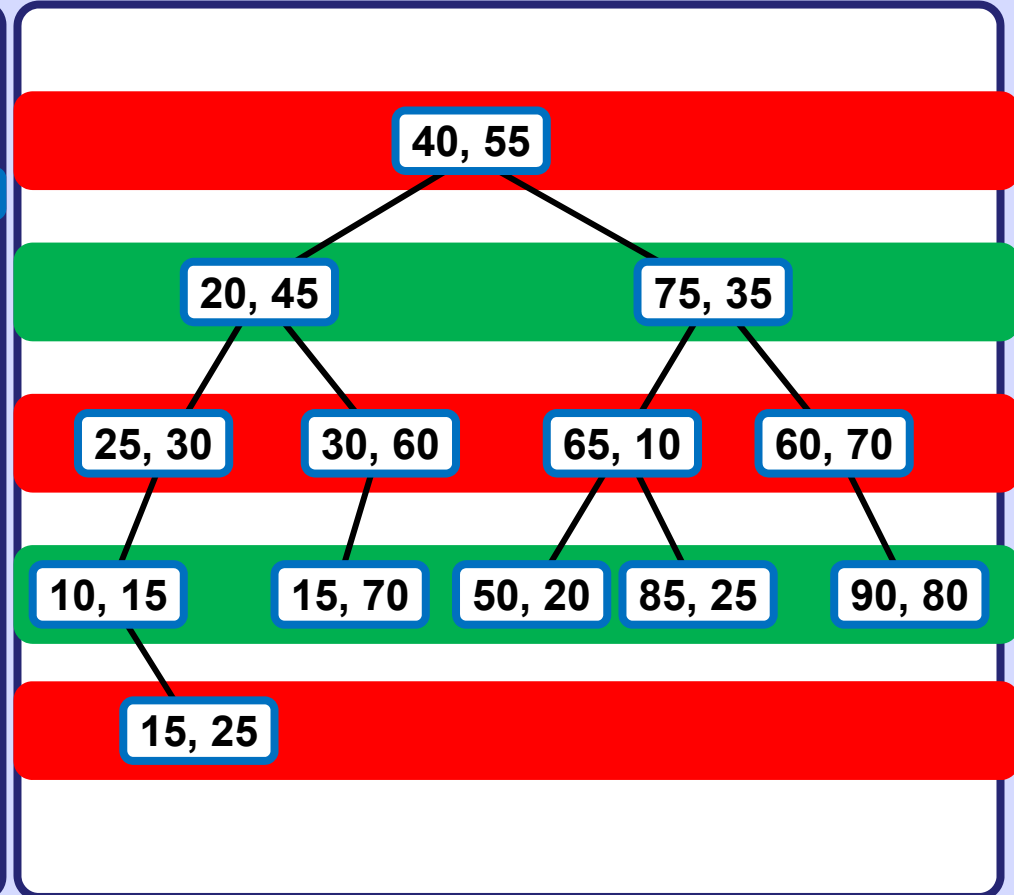20, 45

75, 35

25, 30

15, 25

85, 25

50, 20

10, 15

65, 10

40, 55

20, 45

75, 35

25, 30

30, 60

65, 10

60, 70

10, 15

15, 70

50, 20

85, 25

90, 80

15, 25
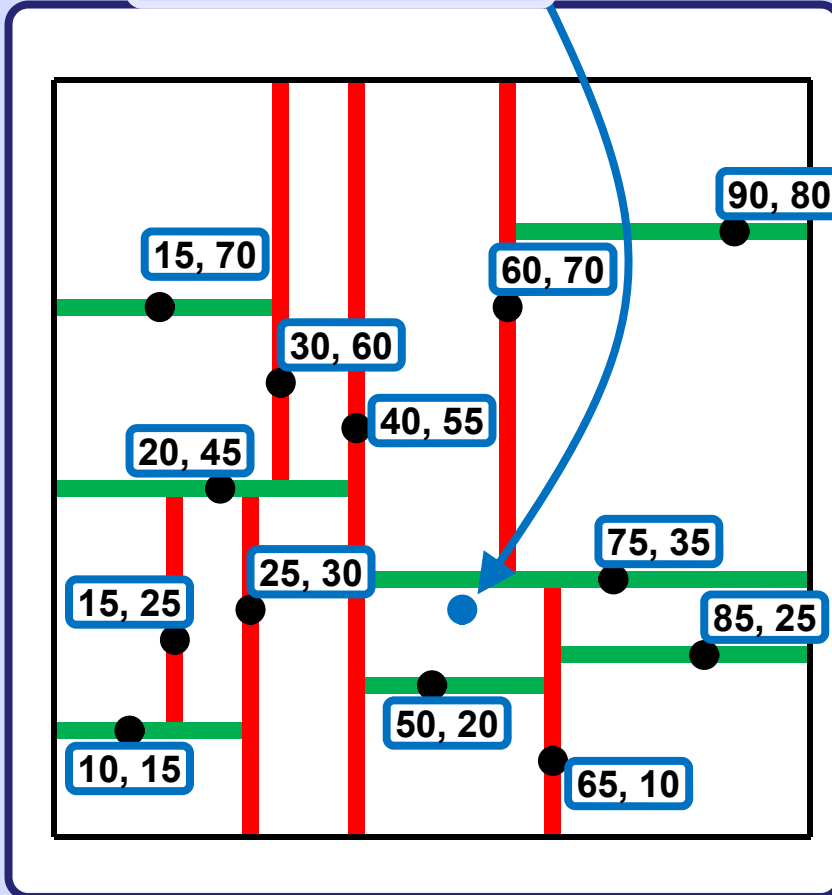
Operation Insert works analogously as in other (1D) trees.

Find the place for the new node under some of the leaves and insert node there.

Do not accept key which is identical to some other key already stored in the tree.
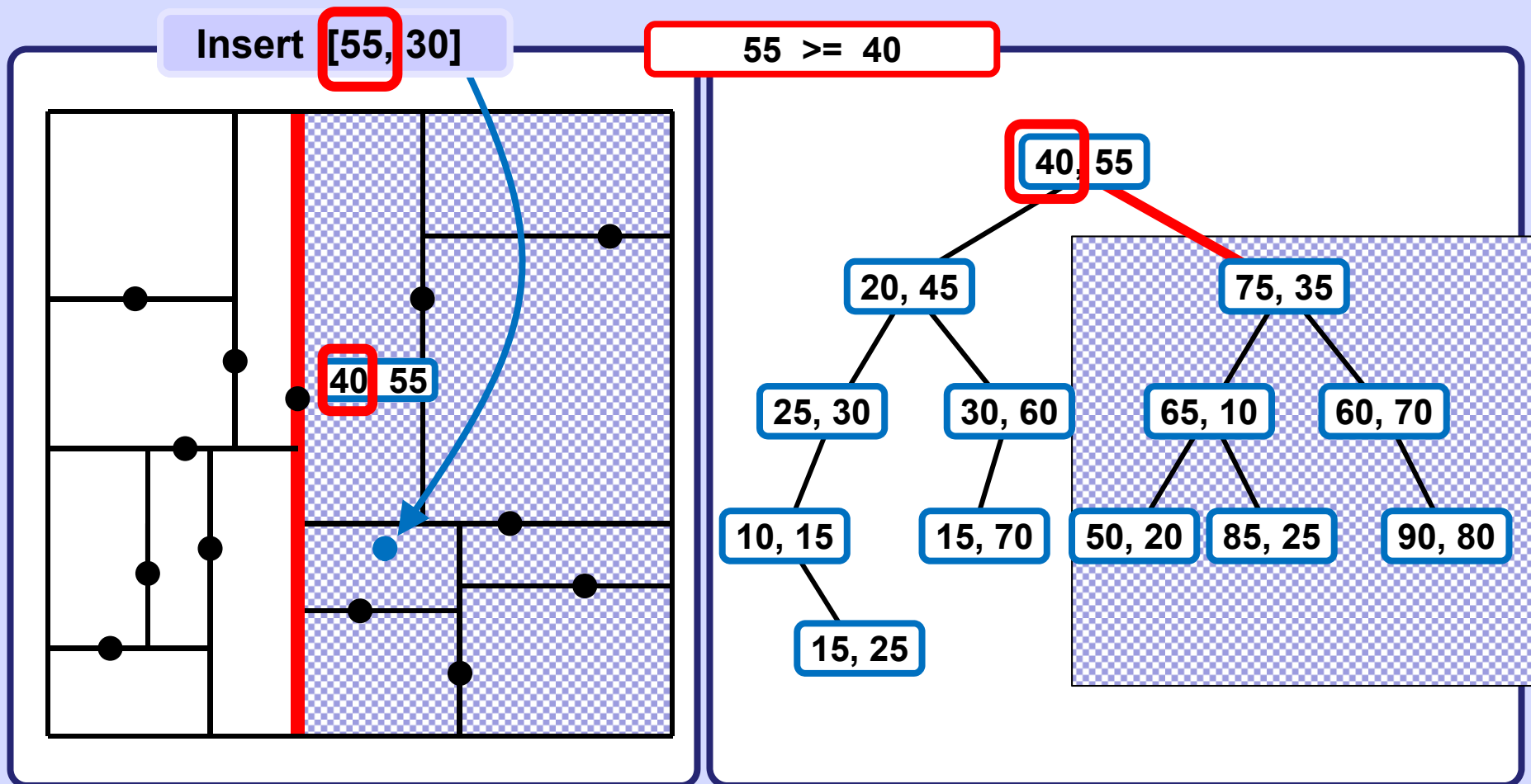
Insert [55, 30]

55 >= 40

40, 55

20, 45

75, 35

25, 30

30, 60

65, 10

60, 70

10, 15

15, 70

50, 20

85, 25

90, 80

15, 25

40 55

Operation Insert works analogously as in other (1D) trees.

Searching for the place for the inserted key/node.

Insert [55, 30]

30 < 35

40, 55

20, 45   75, 35

25, 30   30, 60   65, 10   60, 70

10, 15   15, 70   50, 20   85, 25   90, 80

15, 25

75, 35

Operation Insert works analogously as in other (1D) trees.

Searching for the place for the inserted key/node.

Insert [55, 30]

55 < 65

40, 55

20, 45   75, 35

25, 30   30, 60   65, 10   60, 70

10, 15   15, 70   50, 20   85, 25   90, 80

15, 25

65, 10

Operation Insert works analogously as in other (1D) trees.

Searching for the place for the inserted key/node.

Insert [55, 30]

30 >= 20

40, 55

20, 45   75, 35

25, 30   30, 60   65, 10   60, 70

10, 15   15, 70   50, 20   85, 25   90, 80

15, 25

65, 10
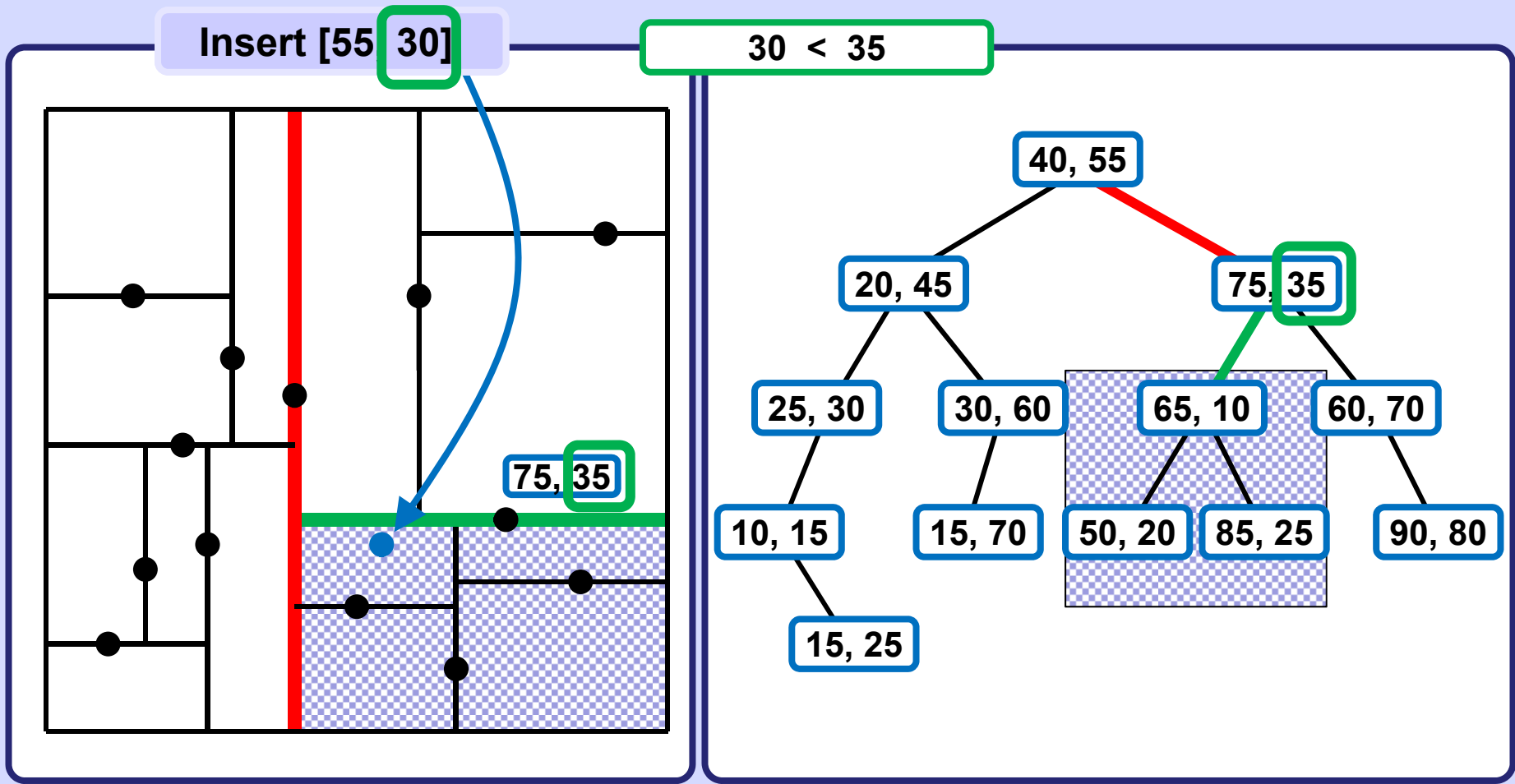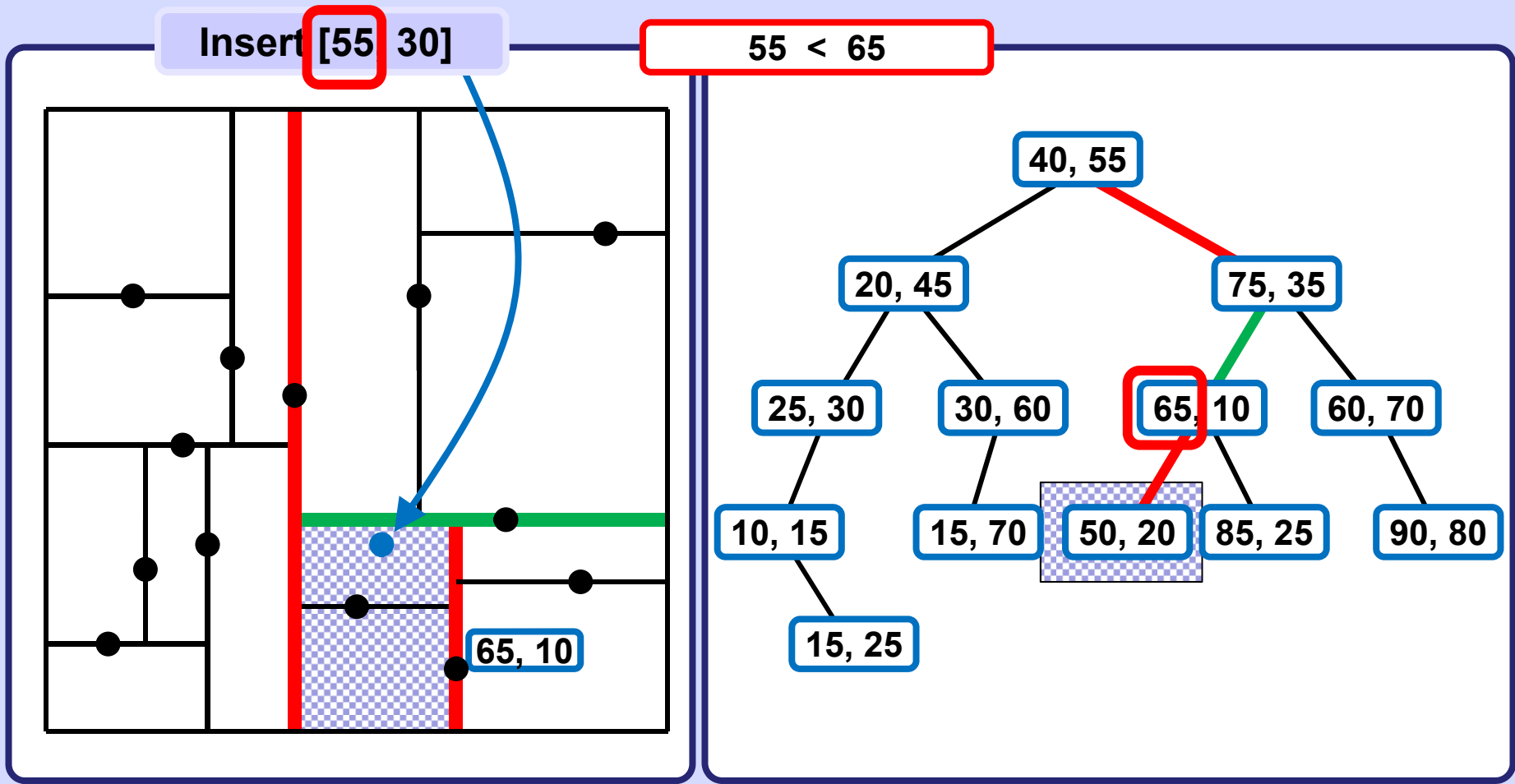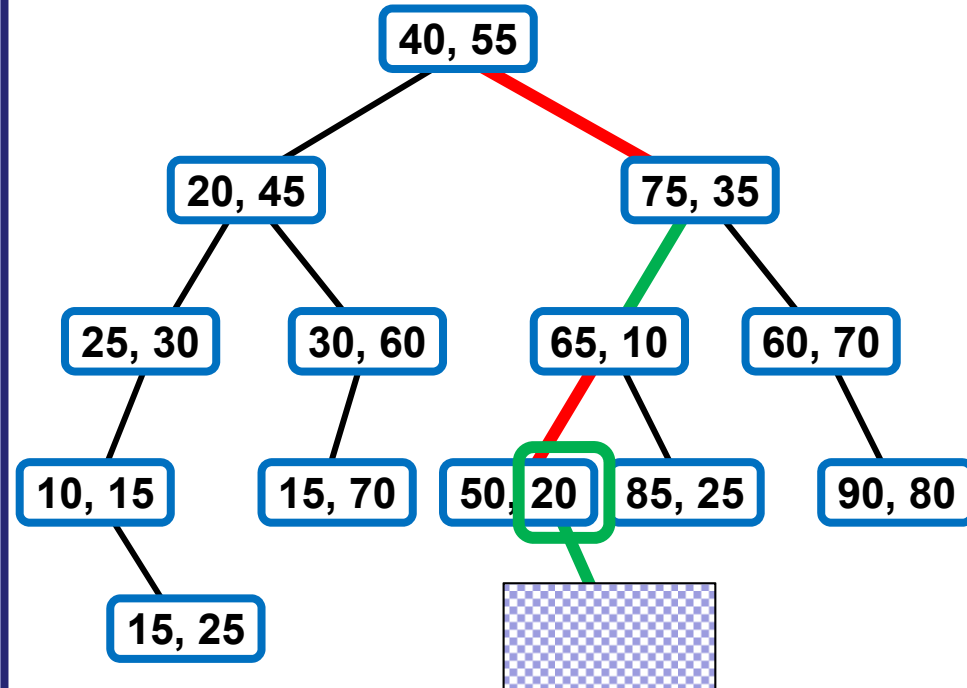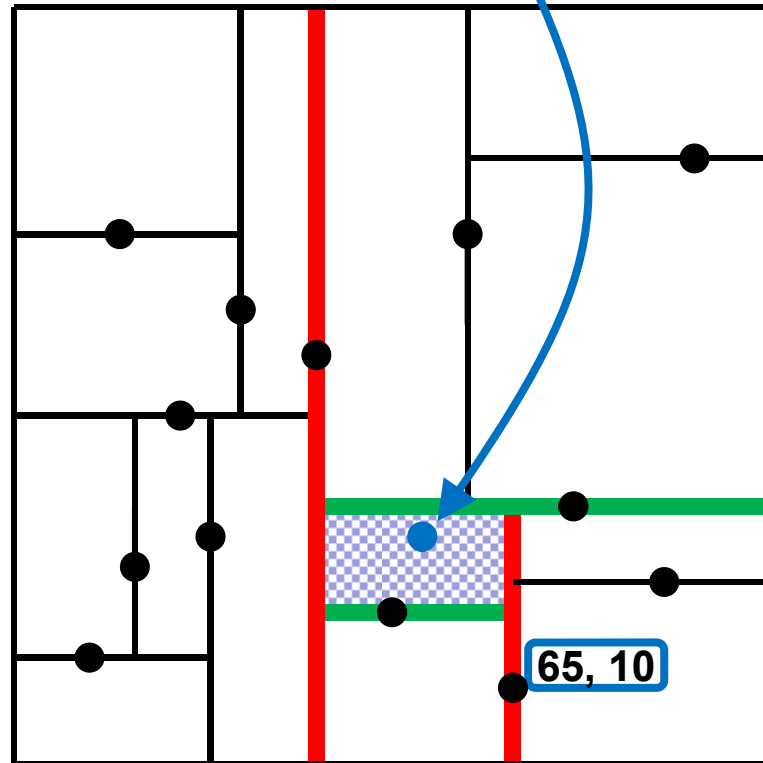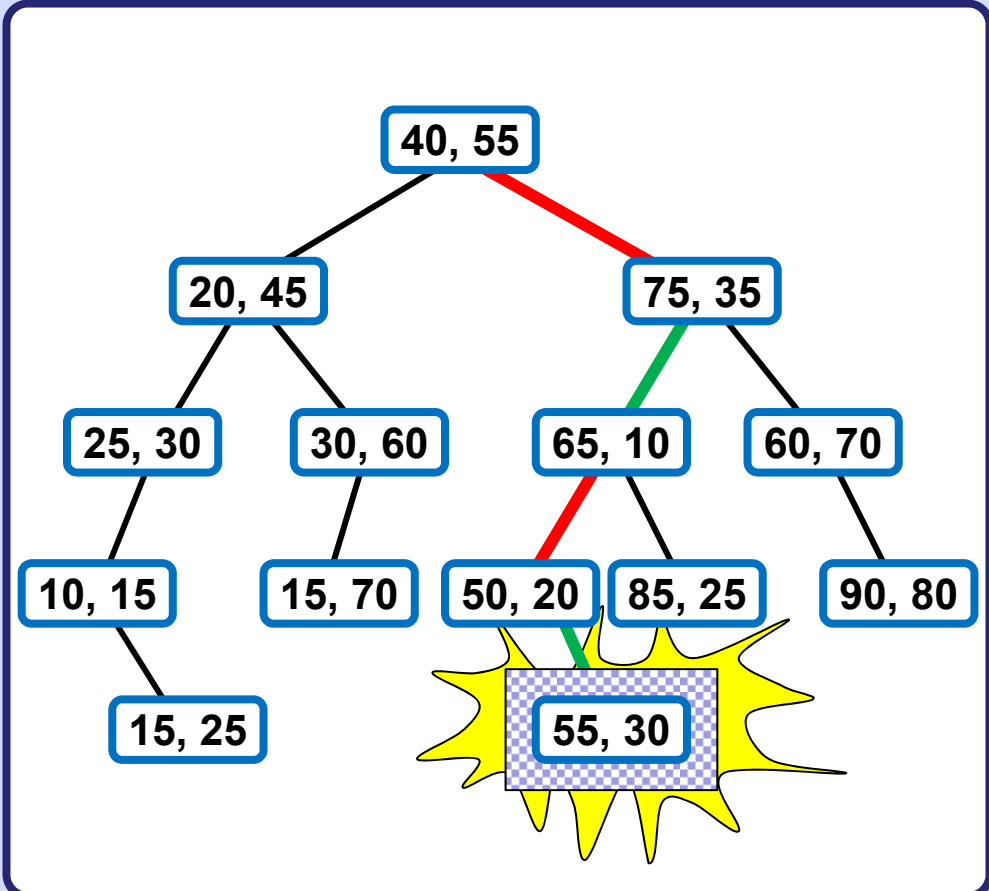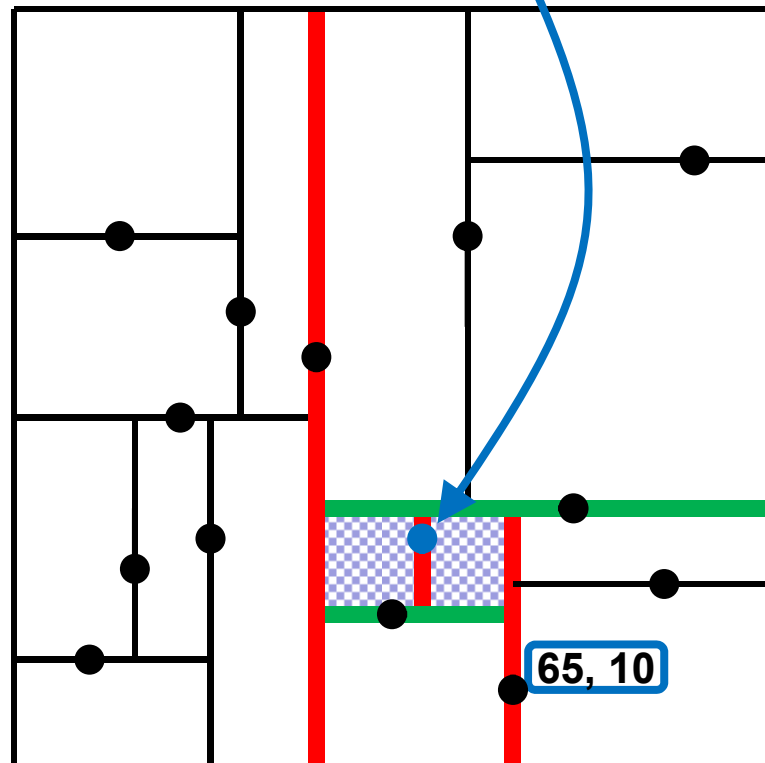
Operation Insert works analogously as in other (1D) trees.

Searching for the place for the inserted key/node.

Inserted [55, 30]

Operation Insert works analogously as in other (1D) trees.

The place for the inserted key/node was found, node/key was inserted.

```
Node insert(Point x, Node t, int cd) {
  if (t == null)                        // under a leaf
    t = new Node(x);
  else if (x.equals(t.coords))
    throw new ExceptionDuplicatePoint();
  else if (x[cd] < t.coords[cd])
    t.left = insert(x, t.left, (cd+1)%D);
  else
    t.right = insert(x, t.right, (cd+1)%D);
  return t;
}
```
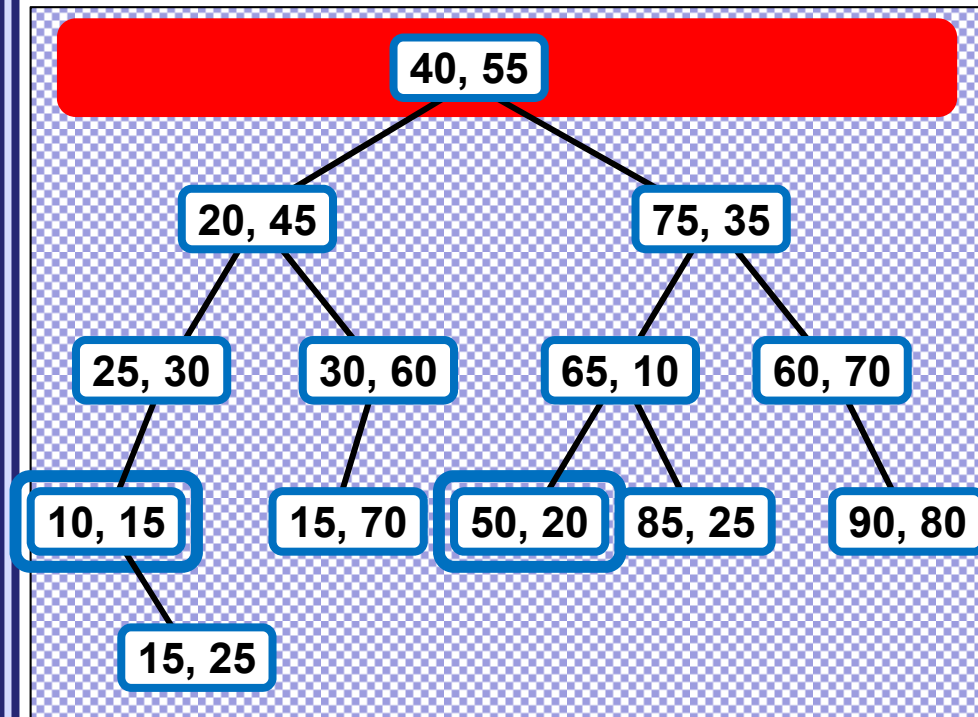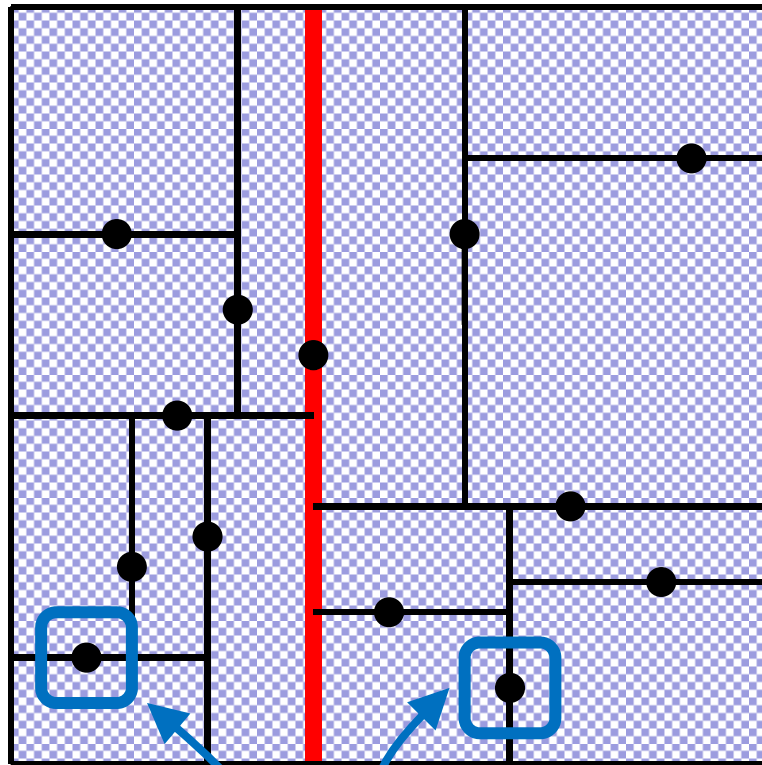
Operation FindMin(dim = k)

Searching for key which k-th coordinate is mimimal of all keys in the tree.

FindMin(dim = k) is performed as part of Delete operation.

The k-d tree offers no simple method of keeping track of the keys with minimum coordinates in any dimension because Delete operation may often significantly change the structure of the tree.
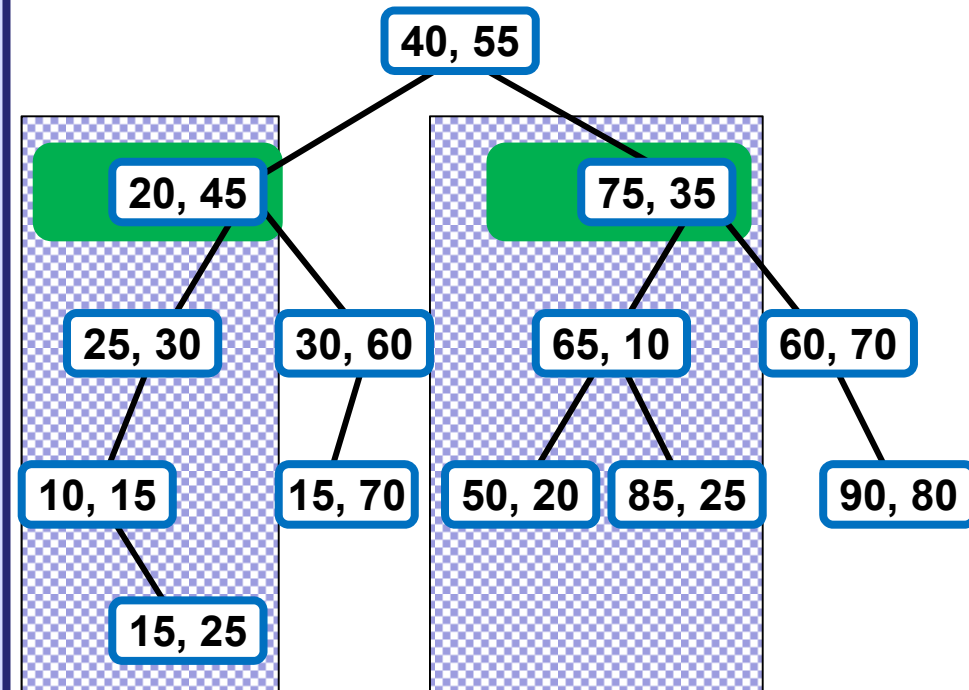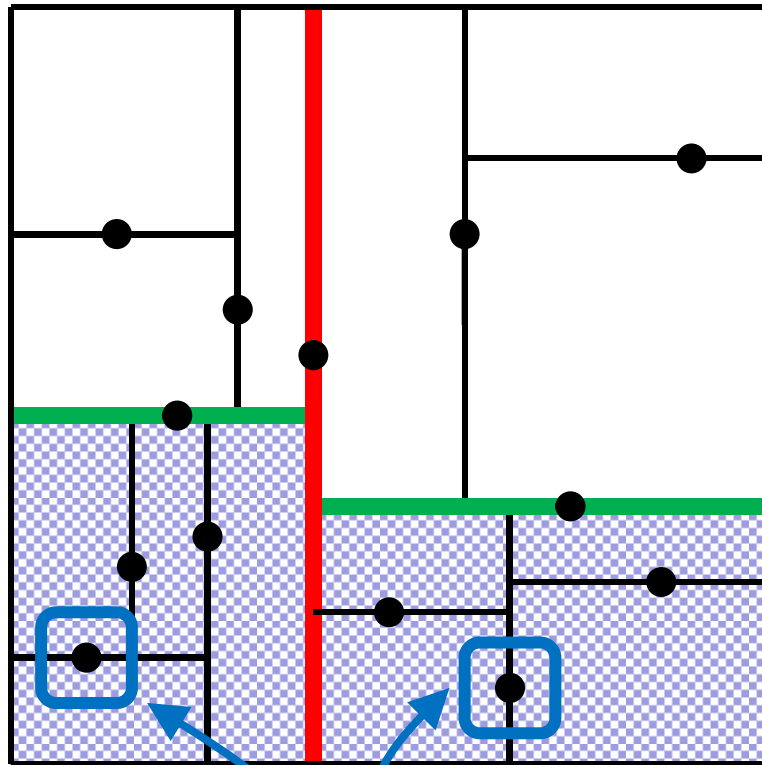
FindMin(dim = k) is the most costly operation,
with complexity $O(n^{1-1/d})$, in the tree with n nodes and dimension d.
For d = 2 it is $O(n^{1/2})$.

FindMin(dim = y)

Node with minimal y-coordinate can be in L or R subtree of a node N corresponding to cutting dimension other than y, thus both subtrees of N (including N) must be searched.

FindMin(dim = y)

40, 55

20, 45

75, 35

25, 30

30, 60

65, 10

60, 70

10, 15

15, 70

50, 20

85, 25

90, 80

15, 25

Node with minimal y-coordinate can be only in L subtree of a node N corresponding to cutting dimension y, thus only L subtree of N (including N) must be searched.

FindMin(dim = y)

40, 55

20, 45   75, 35

25, 30   30, 60   65, 10   60, 70

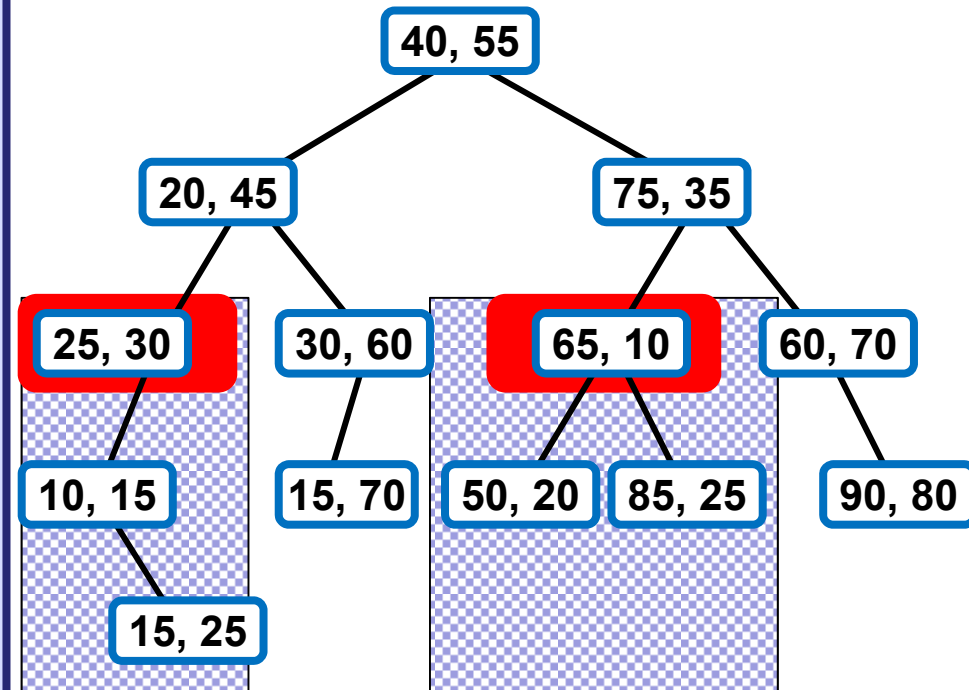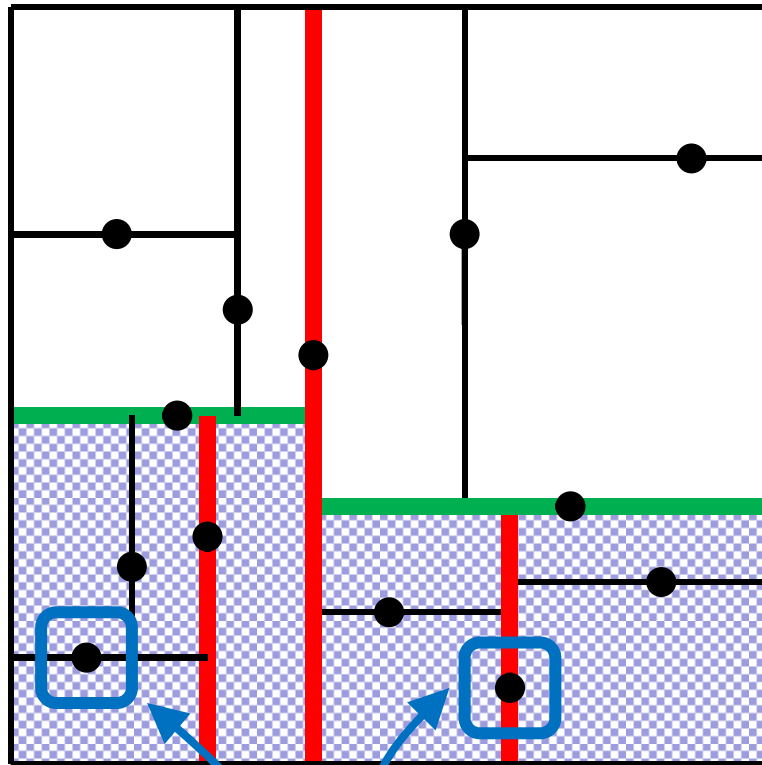10, 15   15, 70   50, 20   85, 25   90, 80

15, 25

Node with minimal y-coordinate can be in L or R subtree of a node N corresponding to cutting dimension other than y, thus both subtrees of N (including N) must be searched.
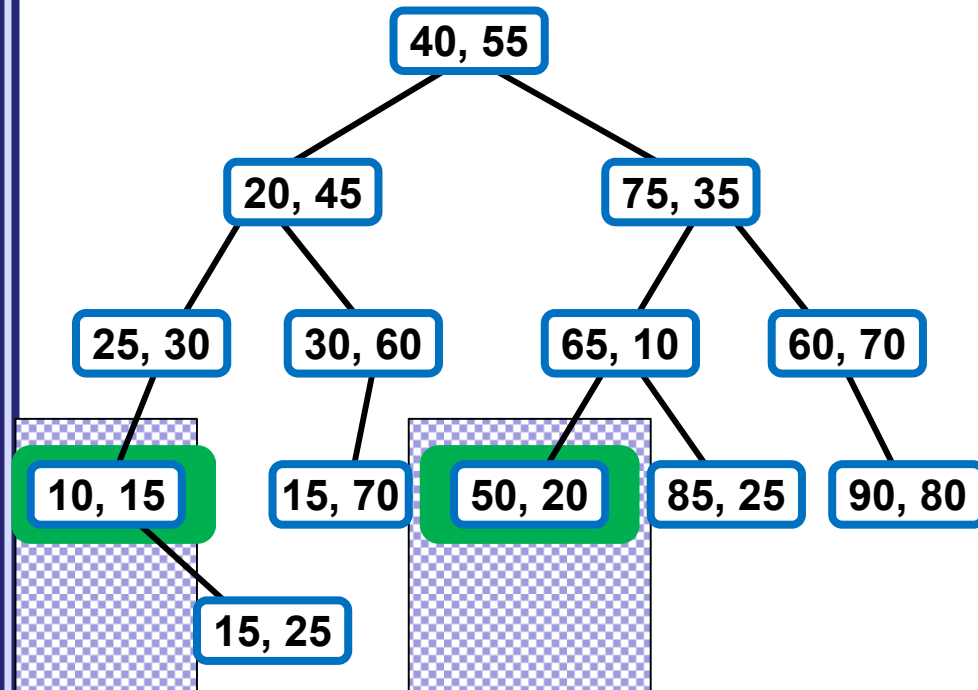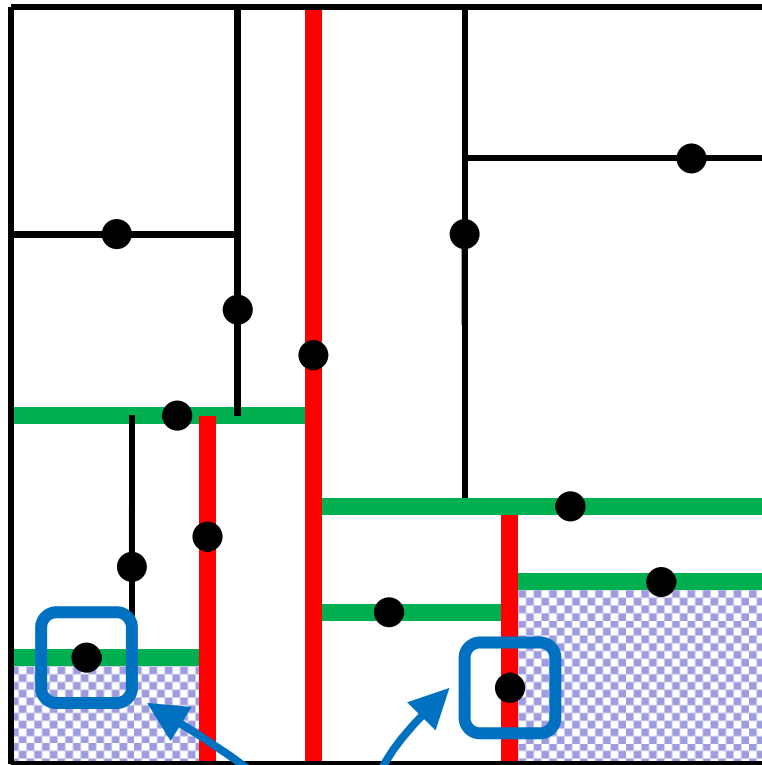
FindMin(dim = y)

Node with minimal y-coordinate can be only in L subtree of a node N corresponding to cutting dimension y, thus only L subtree of N (including N) must be searched.

```
Node findMin(Node t, int dim, int cd) {
   if (t == null) return null;
   if (cd == dim)
     if (t.left == null) return t;
     else return findMin(t.left, dim,(cd+1)%D);
   else
     return min(dim,
                t,
                findMin(t.left, dim, (cd+1)%D),
                findMin(t.right, dim, (cd+1)%D));
}
```

Function min(int dim; Node t1, t2, t3) returns that node out of t1, t2, t3 which coordinate in dimension dim is the smallest.
if (t1.coords[dim] <= t2.coords[dim] && t1.coords[dim] <= t3.coords[dim]) return t1;
if (t2.coords[dim] <= t1.coords[dim] && t2.coords[dim] <= t3.coords[dim]) return t2;
etc…

Only leaves are physically deleted.

Deleting an inner node X is done by substituting its key values by key values of another suitable node Y deeper in the tree. If Y is leaf, physically delete Y otherwise set X := Y and continue recursively.

Denote cuting dimension of X by cd.

If right subtree R of X is unempty use operation FindMin to find node Y in R which coordinate in cd is minimal. (It may be sometimes even equal to X coordinate in cd.

If right subtree R of X is empty use operation FindMin to find node Y in left subtree L of X which coordinate in cd is minimal. Substitute key values of X by those of Y. Make (by simple reference swap) subtree L be the right subtree of updated X. Now X has right subtree, continue process with previous case.
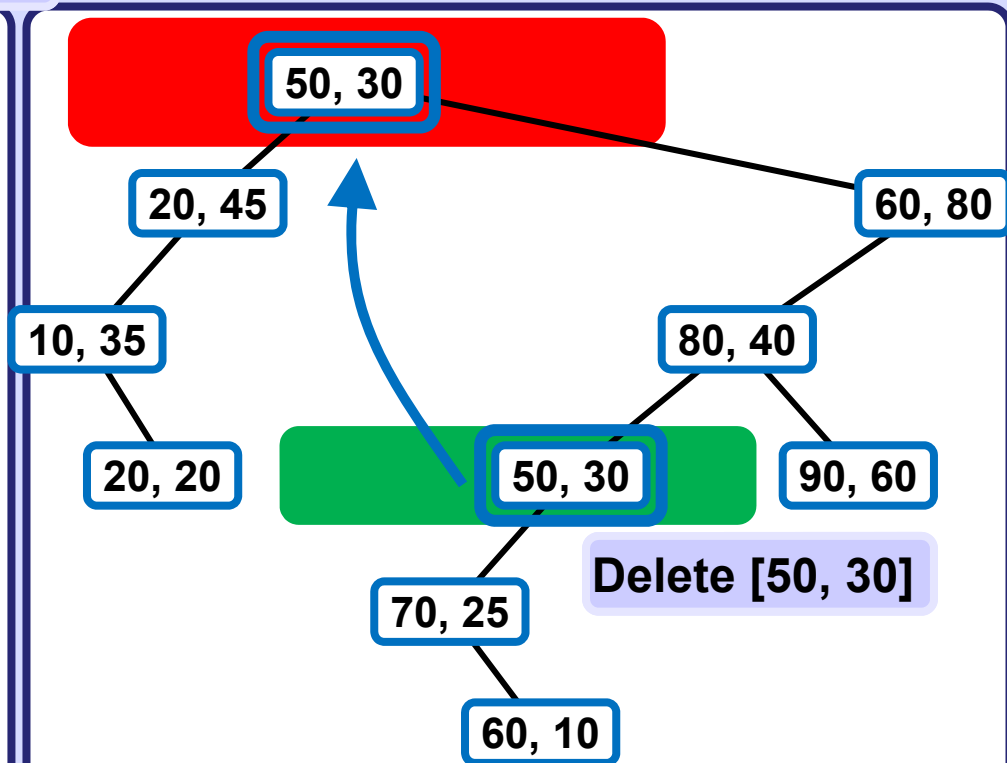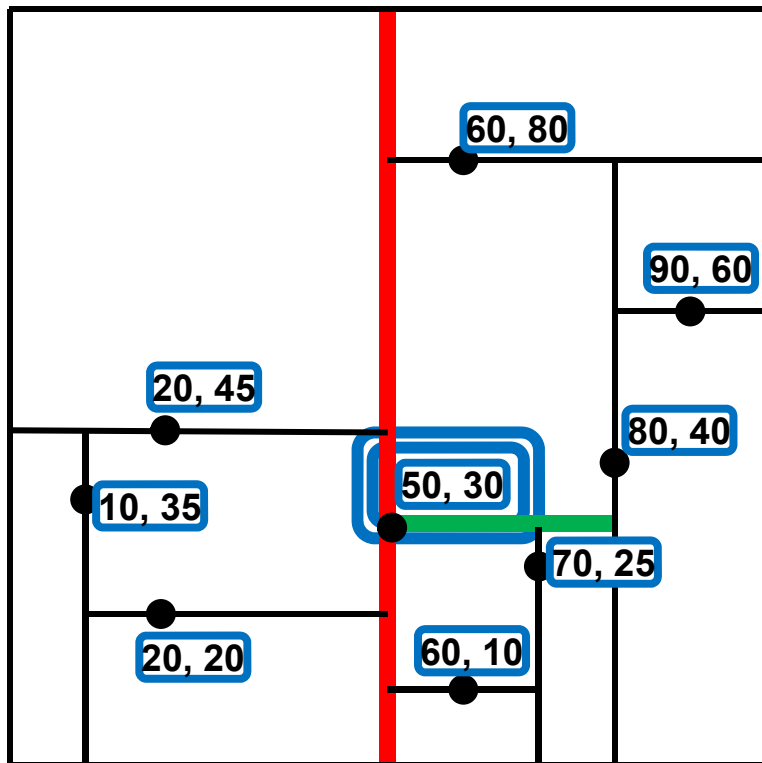
Delete [35, 60]
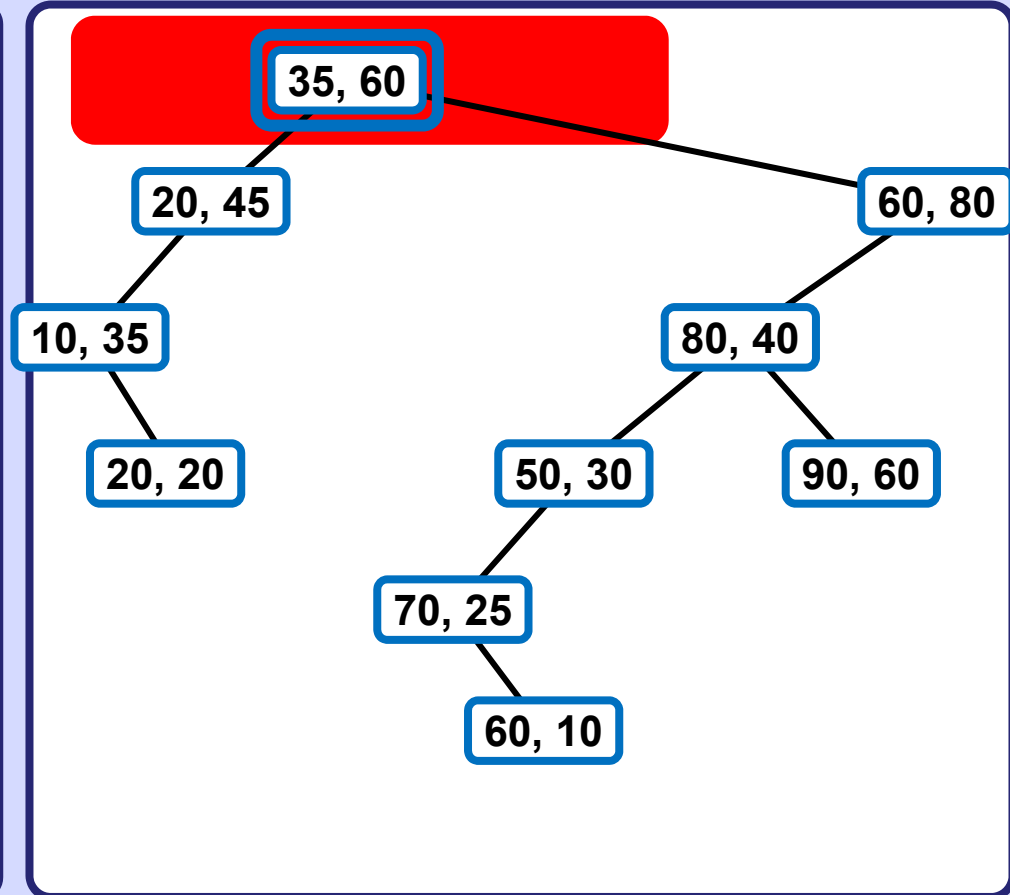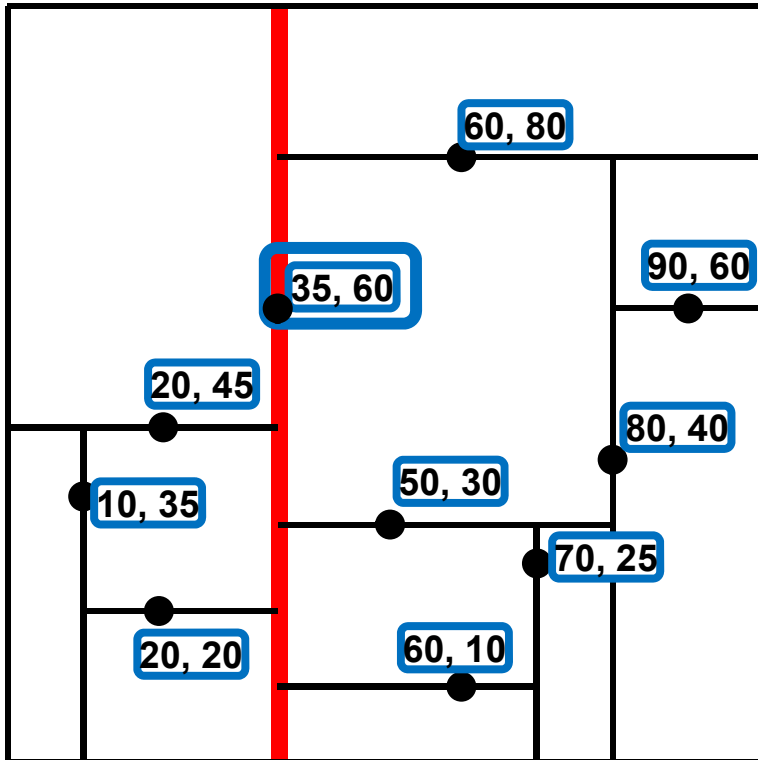
Deleting node [35, 60], its cutting dimension is x.
Find node W with minimum x-coordinate in right subtree of [35, 60].
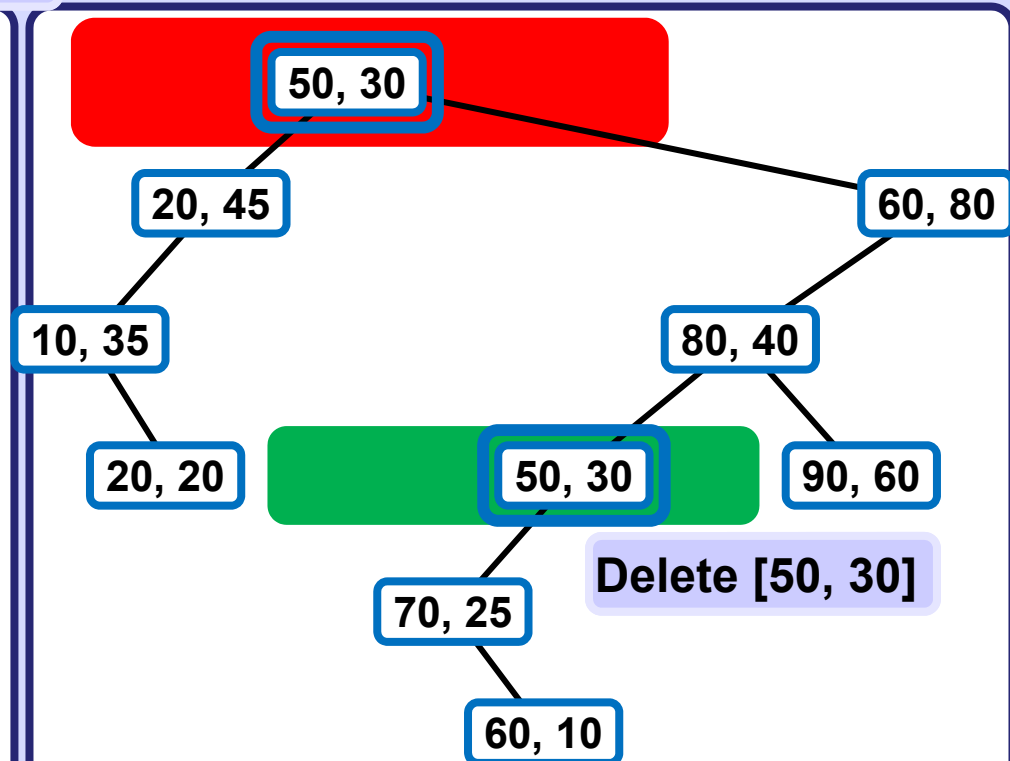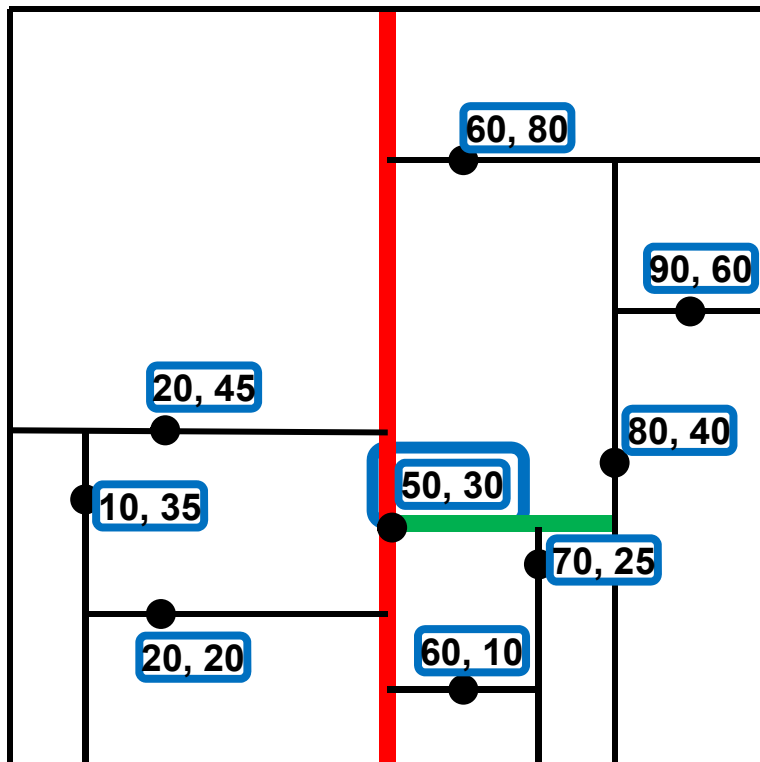Note that W might have different cutting dimension.

Delete [35, 60] ... In progress....

50, 30

60, 80

20, 45

10, 35

80, 40

20, 20

50, 30

90, 60

**Delete [50, 30]**

70, 25

60, 10

Deleting node [35, 60], its cutting dimension is x.
Find node W with minimum x-coordinate in right subtree of [35, 60].
Fill node [35, 60] with keys of W
and if W is not a leaf continue by recursively deleting W.

Delete [35, 60] ... In progress....

Delete [50, 30]

Deleting original node [50, 30], it cutting dimension is y, it has no R subtree.
Find node Z with minimum y-coordinate in LEFT subtree of [50, 30],
Fill [50, 30] with keys of Z and move L subtree of [50, 30] to its R subtree.

Delete [35, 60] ... In progress....
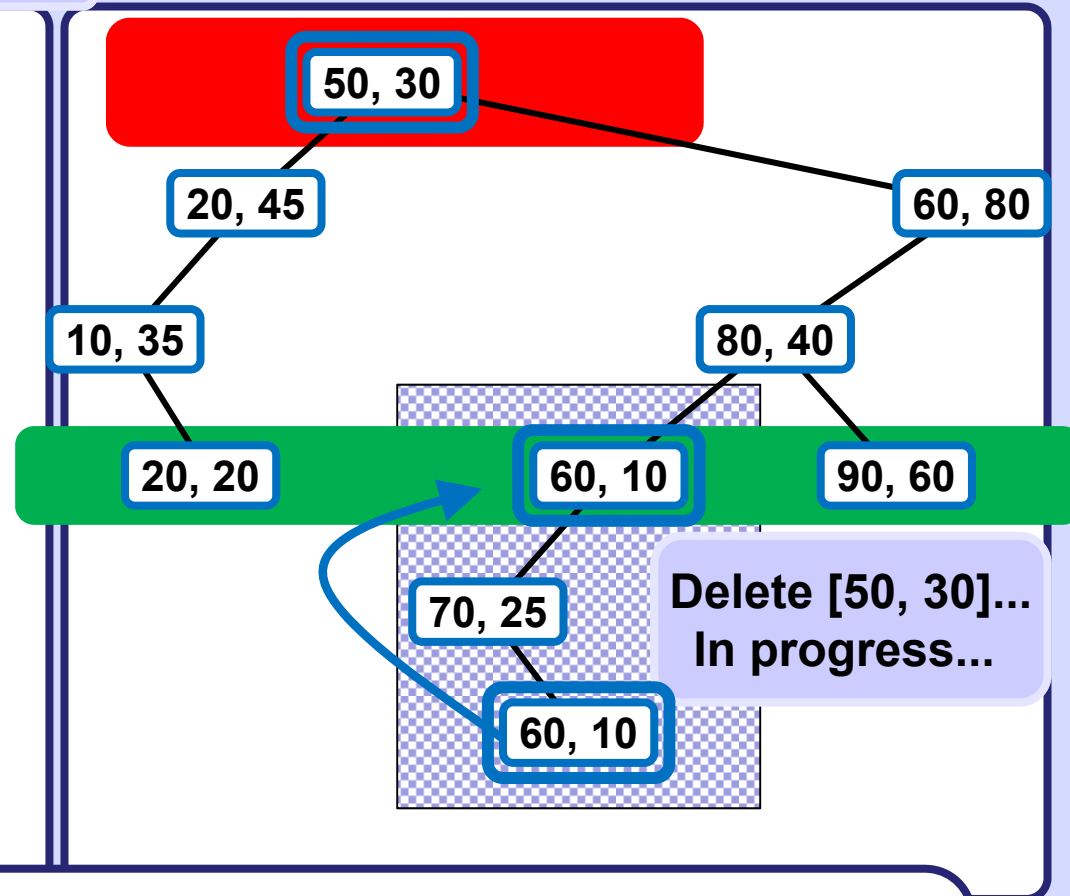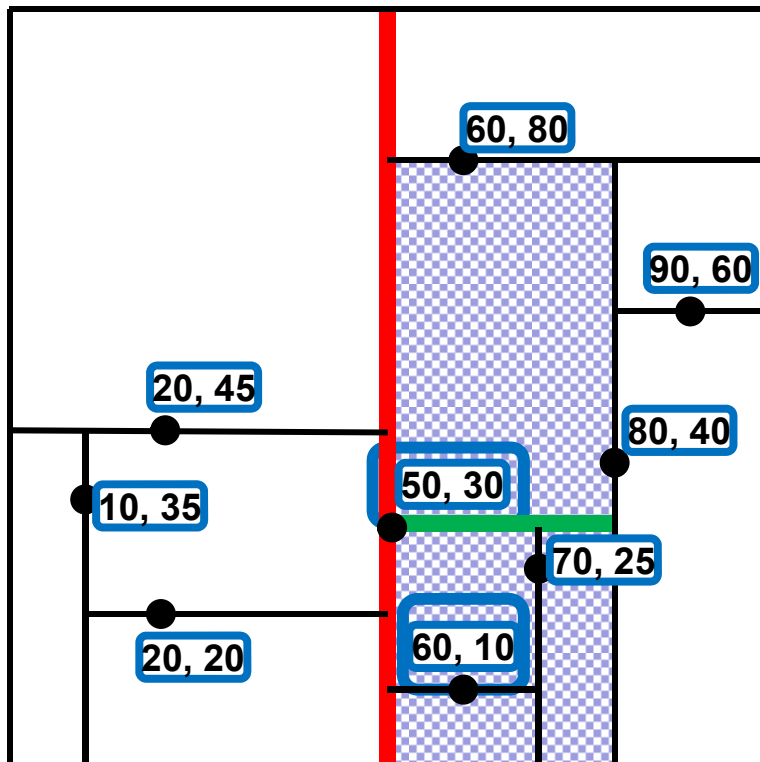
Delete [50, 30]... In progress...
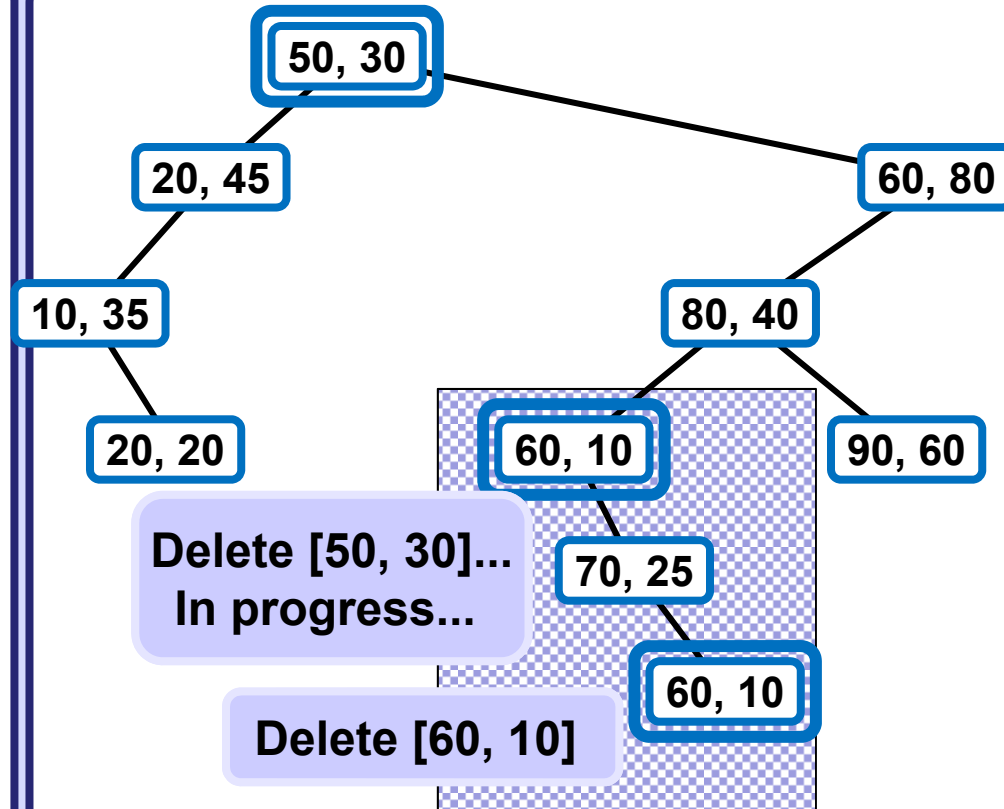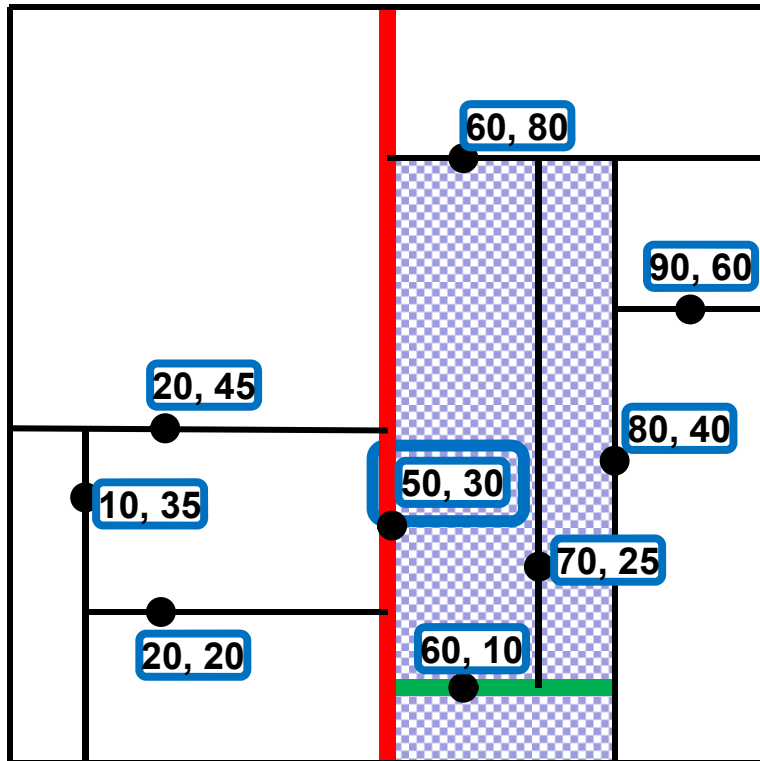
Deleting original node [50, 30], it cutting dimension is y, it has no R subtree.
Find node Z with minimum y-coordinate in LEFT subtree of [50, 30],
Fill [50, 30] with keys of Z and move L subtree of [50, 30] to its R subtree.
If Z is not a leaf continue by recursively deleting Z.

Delete [35, 60] ... In progress....

Delete [50, 30]...
In progress...

Delete [60, 10]

Deleting original node [60, 10], it it is a leaf, delete it and and stop.
Note the change in the cell division left to [80, 40], the node with minimal
y-coordinate becomes the splitting node for the corresponding area ▨ .
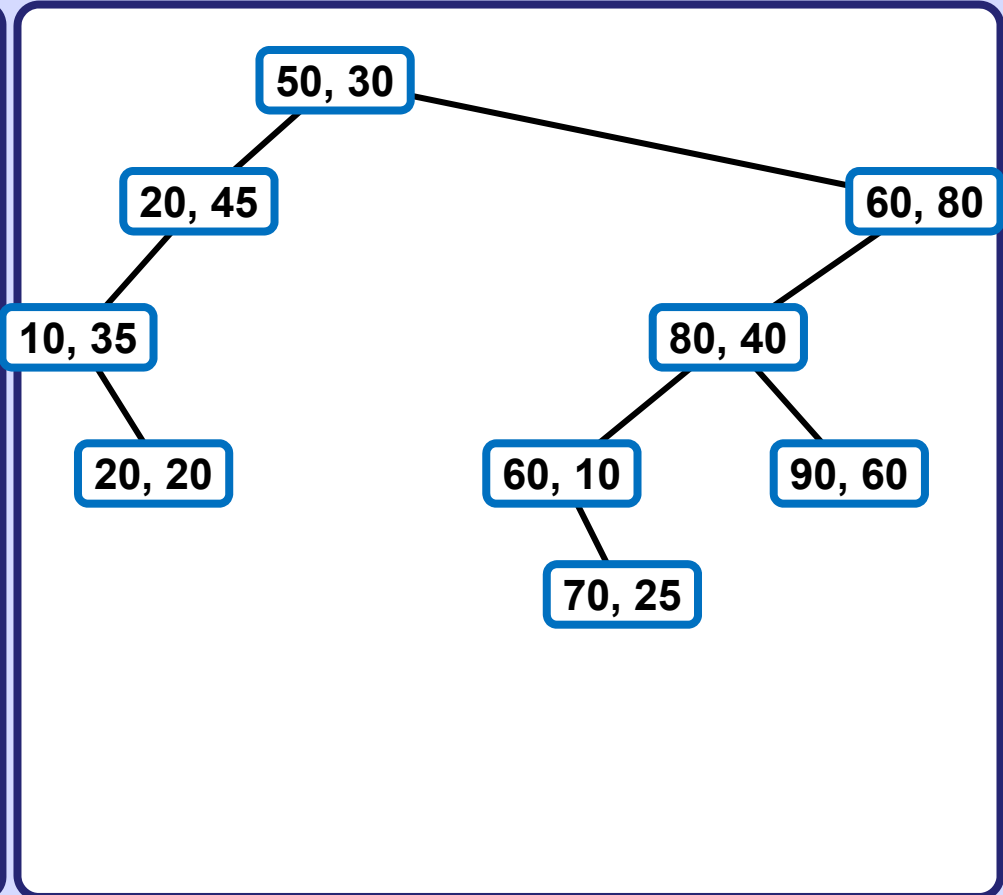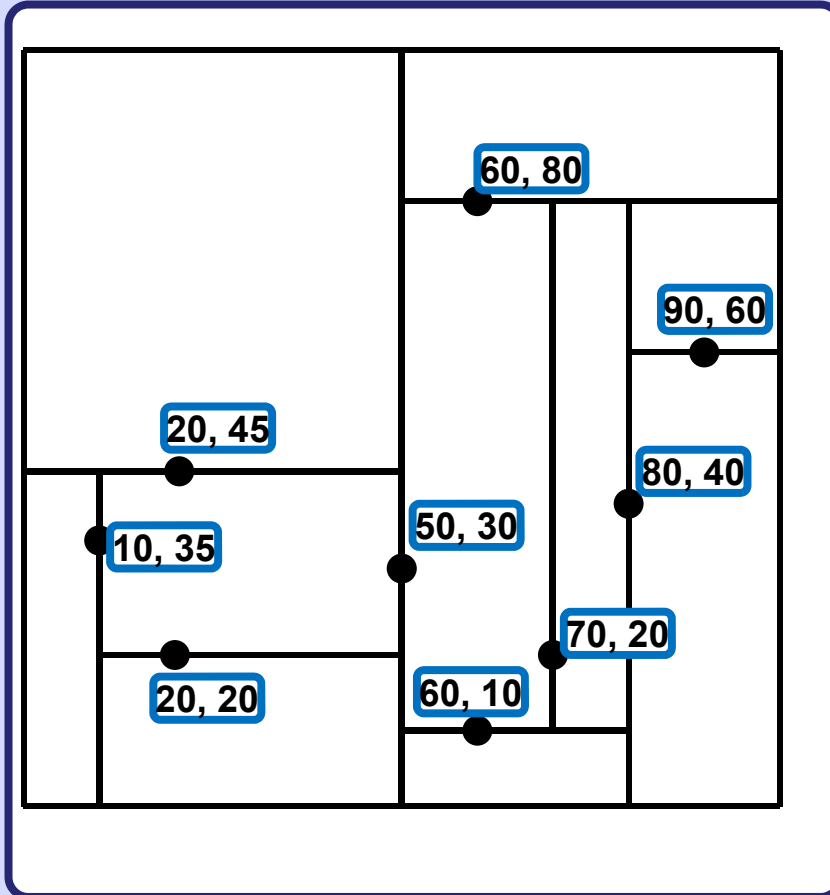
Deleted [35, 60]

Delete [35, 60]
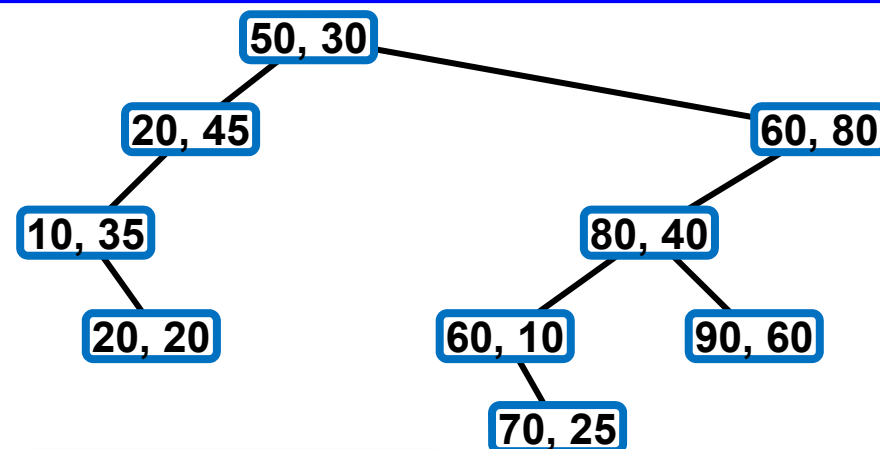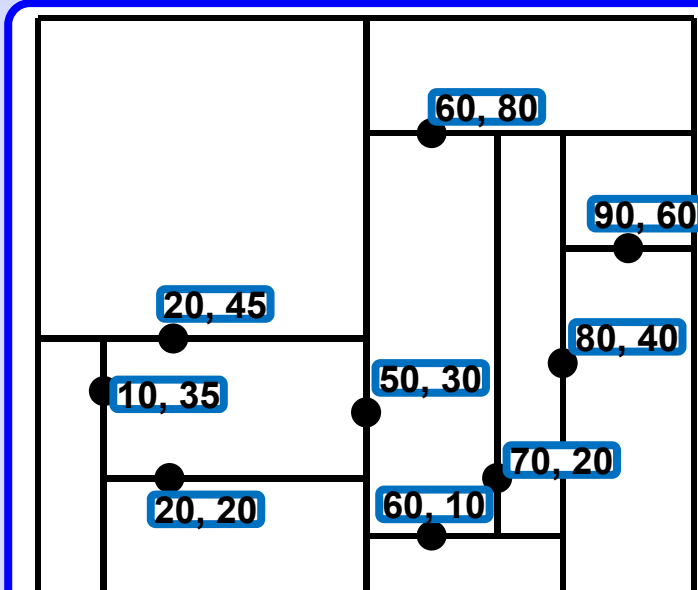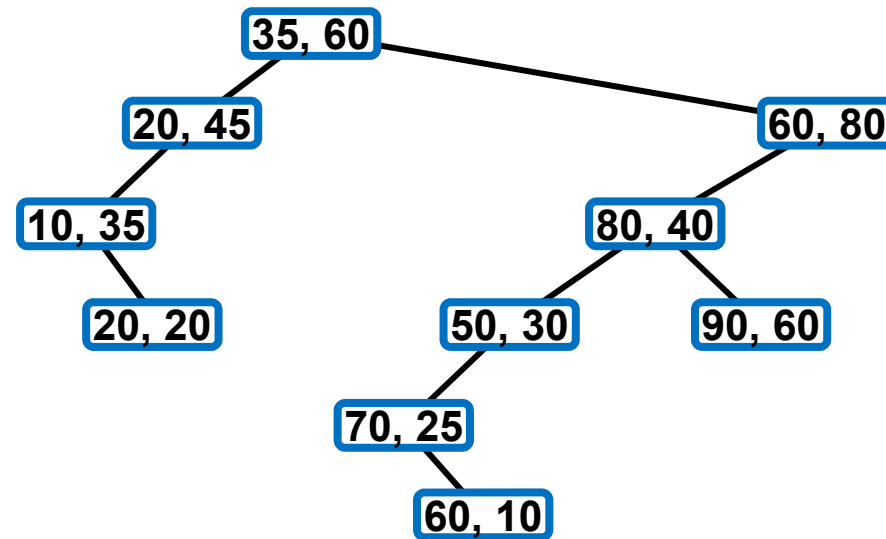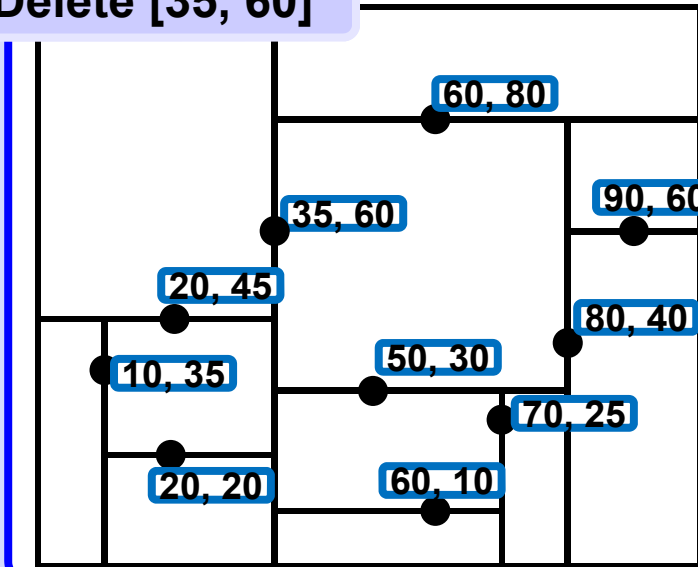
Deleted [35, 60]

```
Node delete(Point x, Node t, int cd) {
   if (t == null) throw new ExceptionDeleteNonexistentPoint();
   else if (x.equals(t.coords)) {    // point x found in t
     if (t.right != null) {        // replace deleted from right
       t.coords = findMin(t.right, cd, (cd+1)%D).coords();
       t.right = delete(t,coords, t.right, (cd+1)%D);
     }
     else if (t.left != null) {    // replace deleted from left
       t.coords = findMin(t.left, cd, (cd+1)%D).coords();
       t.right = delete(t,coords, t.left, (cd+1)%D);
       t.left = null;
     }
     else t = null;                      // delete leaf t
   }
   else                                // point x not found yet
     if (x[cd] < t.coords[cd])        // search left subtree
       t.left = delete(x, t.left, (cd+1)%D);
     else                             // search right subtree
       t.right = delete(x, t.right, (cd+1)%D);
   return t;
}
```

Search runs recursively in L and R subtrees of a node (root at the start).

Register and update **partial results**:
Object close = {close.point, close.dist}.
Field .point refers to node(point) which is so far closest to the query,
field.dist contains euclidean distance from .point to the query.

Perform **pruning**:
During search dismiss the cells (and associated subtrees) which are too far
from query. Object close helps to accomplish this task.

Traversal order (left or right subtree is searched first) depends on simple
(in other vartiants of k-d tree on more advanced) heuristic:
First search the subtree whose cell associated with it is closer to the query.
This does not guarantee better results but in practice it helps.
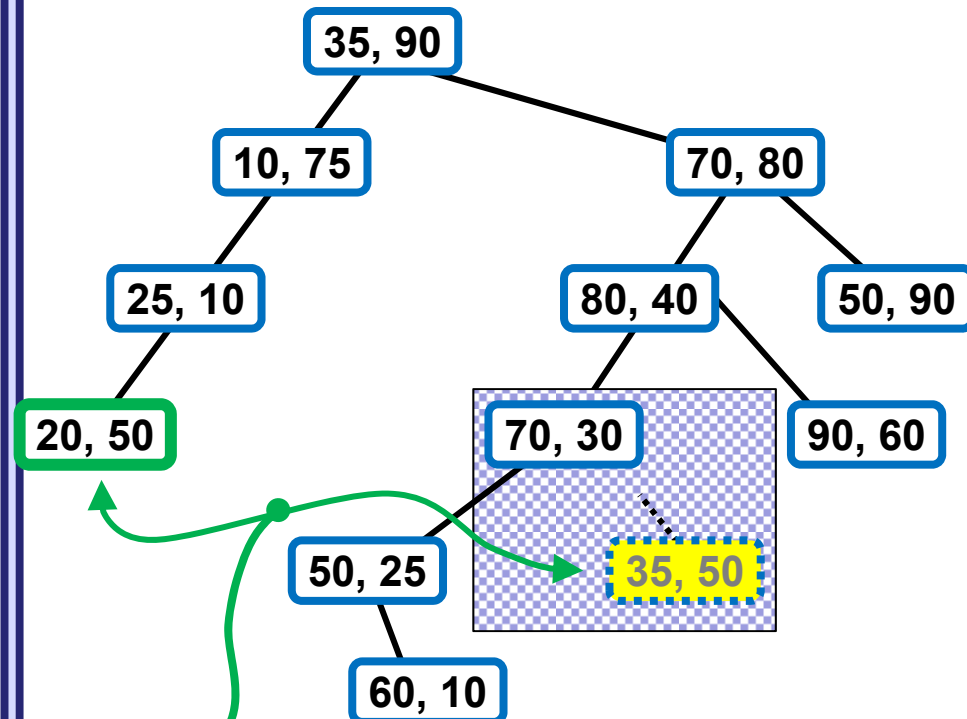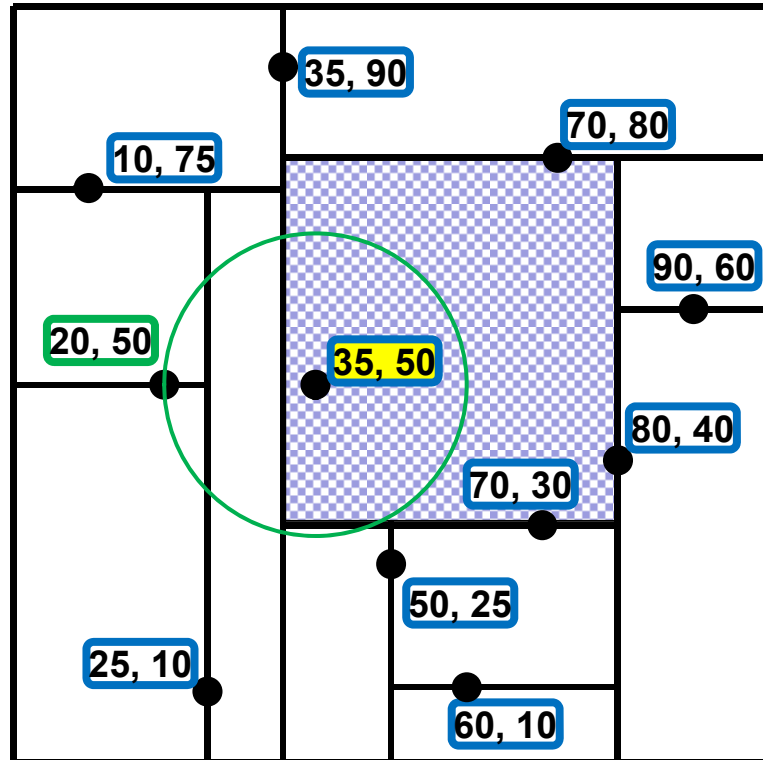
The query point [35, 50] is inside leaf cell defined by node [70, 30].
The closest point to query [35, 50] is the point [20, 50] which lies in a distant part of the tree.
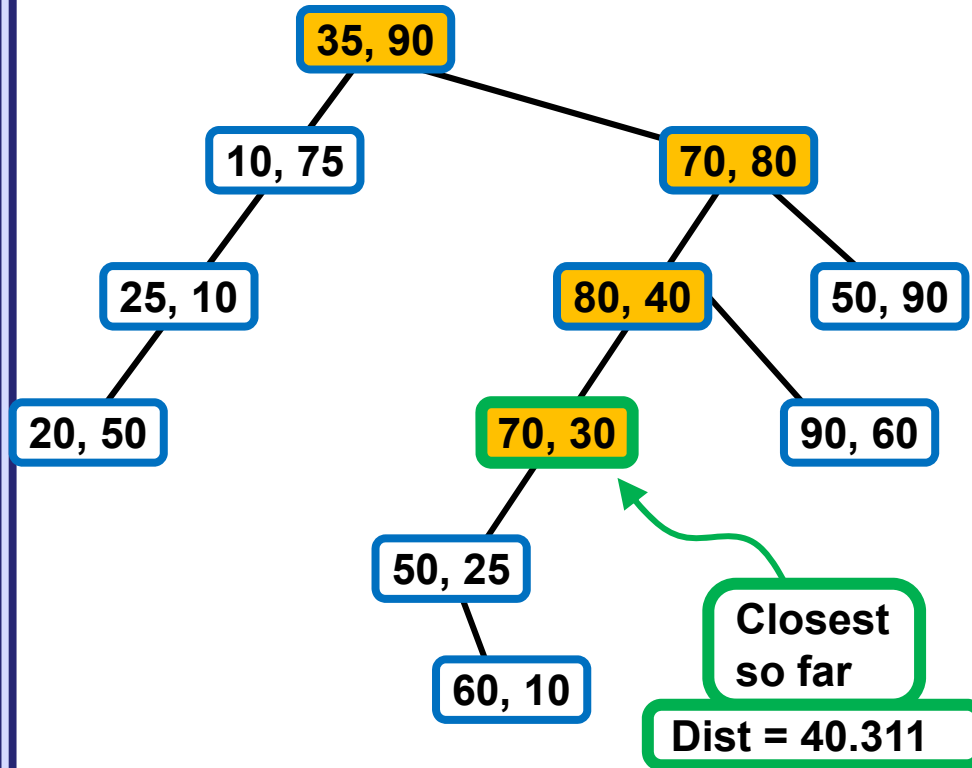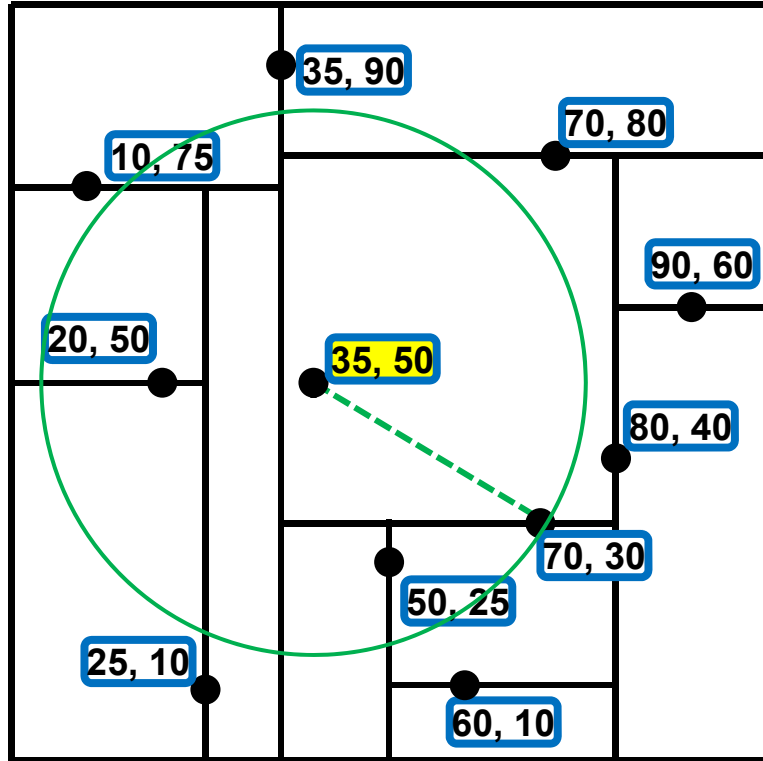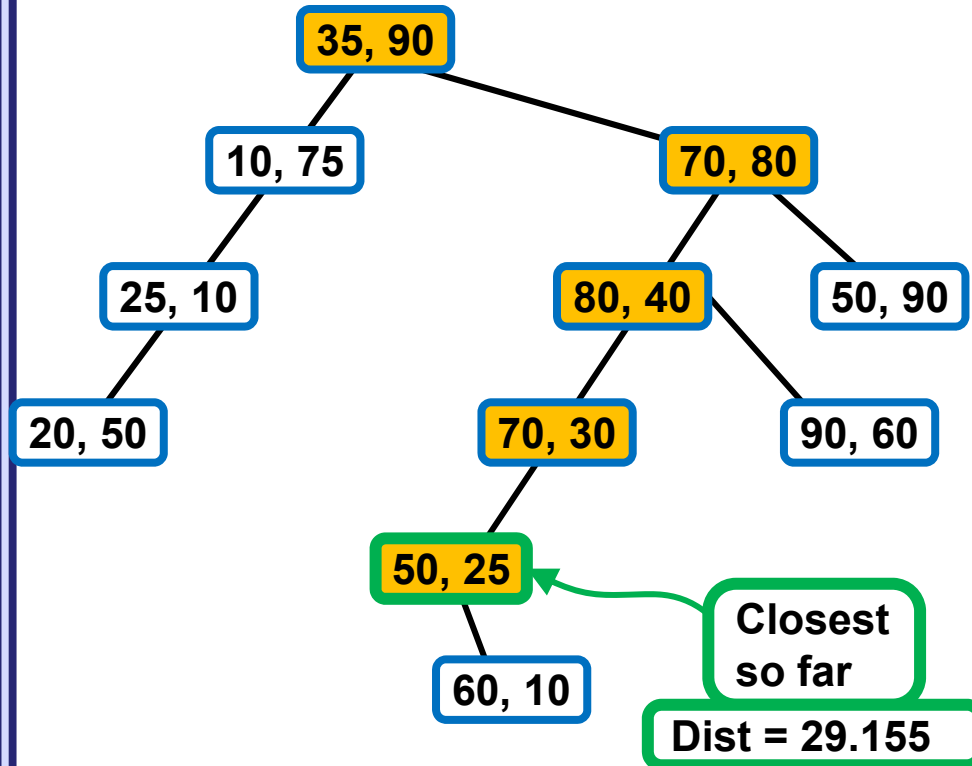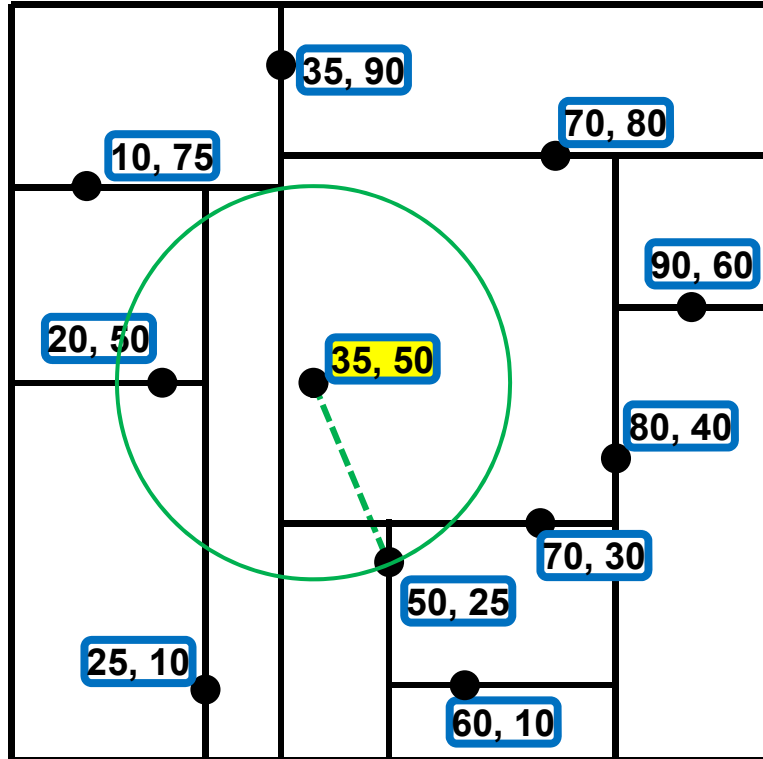
Searched nodes 🟧

Searched nodes 🟧

Pruned

35, 90
70, 80
10, 75
25, 10
20, 50
80, 40
50, 90
70, 30
90, 60
50, 25
60, 10

35, 50

Closest so far

Dist = 29.155

Searched nodes

Pruned

35, 90

70, 80

10, 75

25, 10

80, 40

50, 90

20, 50

70, 30

90, 60

50, 25

Closest so far

60, 10

Dist = 15.0
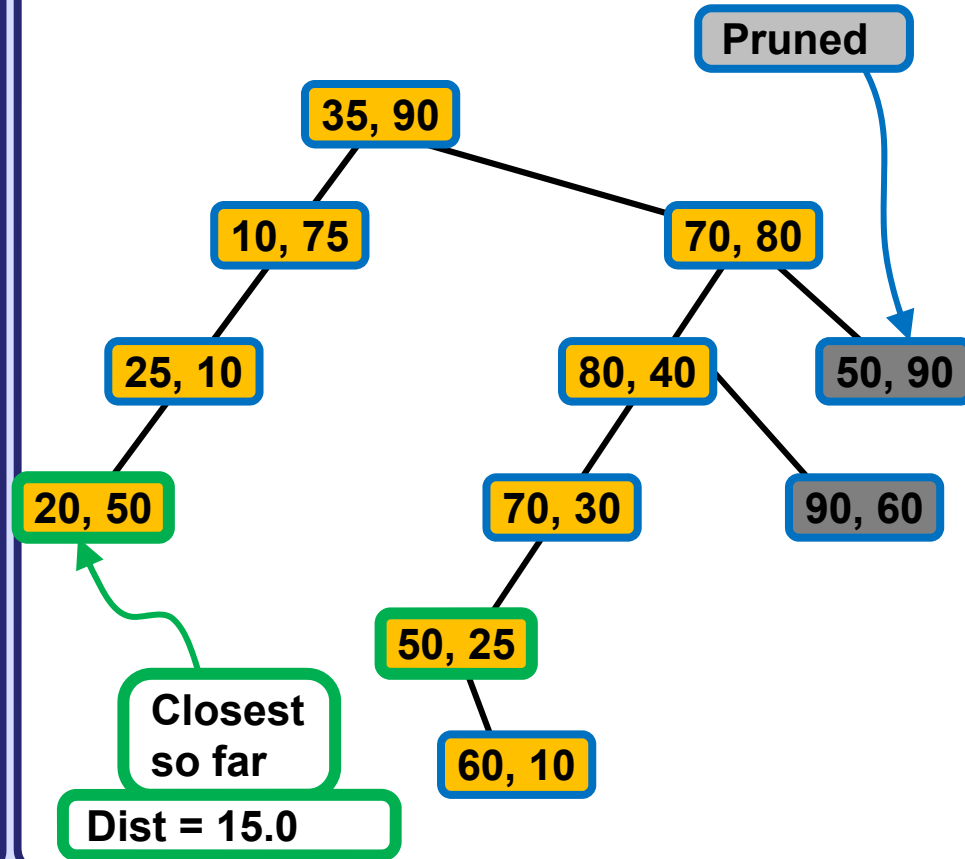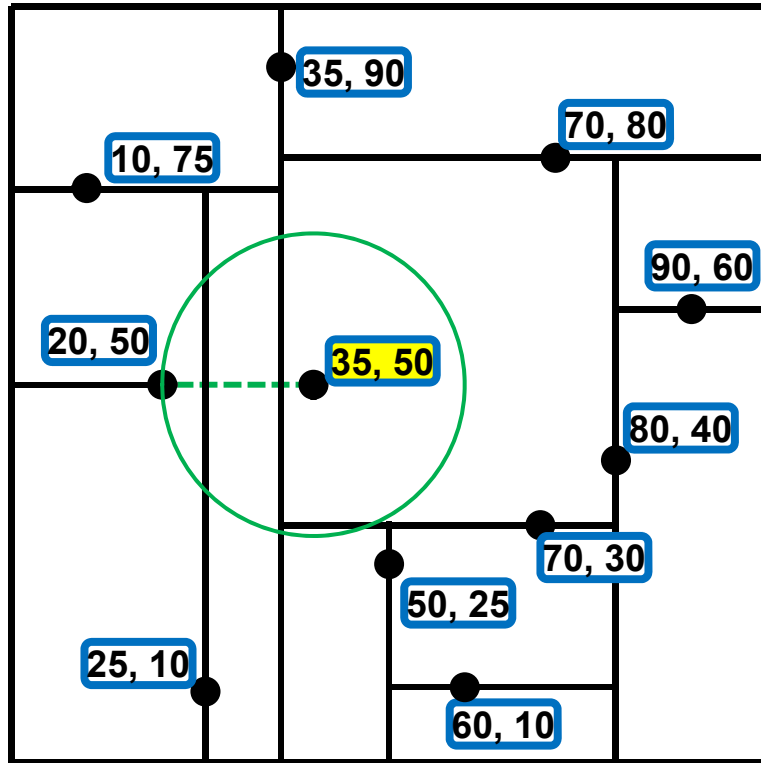
Searched nodes

To implement Nearest Neighbour Search suppose existence of following:

1. Class HyperRectangle (or Box, in 2D just Rectangle) representing cells of particular nodes in k-d tree. This class offers two methods:
  HyperRectangle trimLeft(int cd, coords c)
  HyperRectangle trimRight(int cd, coords c)
When hyperrectangle *this* represents current cell, cd represents cutting dimension, c represents coordinates of a point (or node)
then trimLeft returns the hyperrectangle associated with the left subtree of the point/node with coordinates c. Analogously trim Right returns hyperrectangle associated with the right subtree.

 2. Class or utility G (like Geometry) equipped with methods
distance(Point p, Point q) with obvious functionality
distance(point p, Hyperrectangle r) which computes distance from q to the point of r which is nearest to q.

3. Object close with fields dist and point, storing the best distance found so far and reference to the point at which it was attained. Initialize by
dist = inf, point = null.

```
NNres nn(point q, Node t, int cd, HypRec r, Nnres close){
  if (t == null) return close;        // out of tree
  if (G.distance(q, r) >= close.dist)
    return close;                 // cell of t is too far from q
  Number dist = G.distance(q, t.coords);
  if (dist < close.dist)     // upd close if necessary
    { close.coords = t.coords; close.dist = dist; }
  if (q[cd] < t.coords[cd] {              // q closer to L child
     close = nn(q, t.left, (cd+1)%D,
                  r.trimLeft(cd, t.coords), close);
     close = nn(q, t.right, (cd+1)%D,
                  r.trimRight(cd, t.coords), close);
  } else {                              // q closer to R child
     close = nn(q, t.right, (cd+1)%D,
                  r.trimRight(cd, t.coords), close);
     close = nn(q, t.left, (cd+1)%D,
                  r.trimLeft(cd, t.coords), close);
  }
  return close;
}
```

Complexity of NN search might be close to O(n) when data points and query point are unfavorably arranged. However, this happens only when:
A. The dimension D is relatively high, 7,8… and more, 10 000 etc…
B. The arrangement of points in low dimension D is very special (artificially constructed etc.)

Expected time of NN search is close to $O(2^d + \log n)$ with uniformly distributed data.
Thus it is effective only when 2^d is significantly smaller than n.