8. k - d trees

**Aim:**

To perform insertion and display operations on k – d trees.

**Theory:**

* *kd trees*

The k - d tree is a modification to the *BST* that allows for efficient processing of *multi-dimensional search keys*. The k - d tree differs from the BST in that each level of the k - d tree makes branching decisions based on a particular search key associated with that level, called the *discriminator or cutting dimension.* It is nearly always used to support search on multi-dimensional coordinates, such as locations in 2D or 3D space.

We define the discriminator at level *i* to be *i* mod *k* for *k* dimensions. For example, assume that we store data organized by *xy*-coordinates. In this case, *k* is 2 (there are two coordinates), with the *x*-coordinate field arbitrarily designated key 0, and the *y*-coordinate field designated key 1.

Figure 1 shows an example of how a collection of two-dimensional points would be stored in a k - d tree.

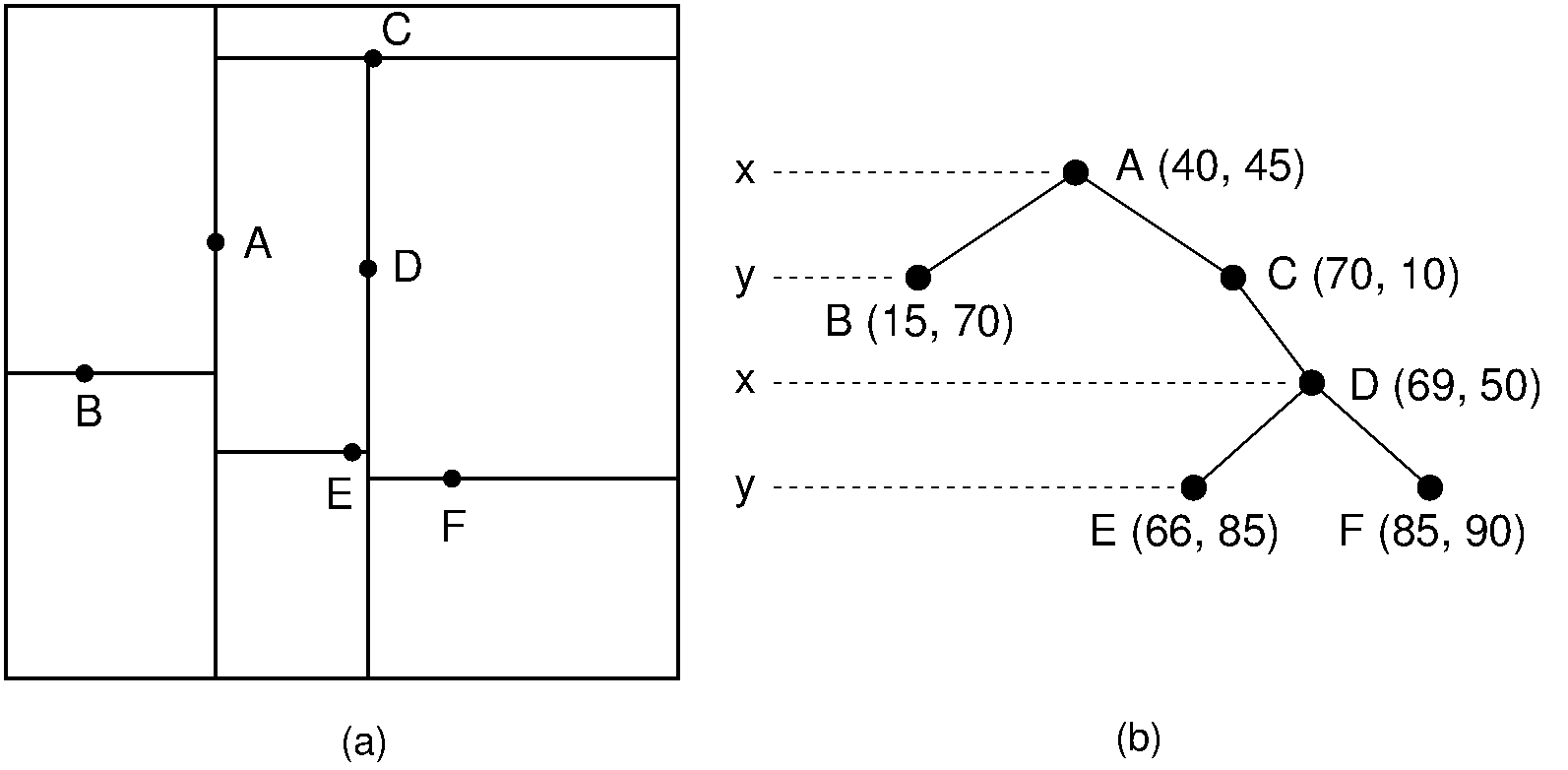


Figure 1 Example of a k - d tree. (a) The k - d tree decomposition for a 128×128 - unit region containing seven data points. (b) The k - d tree for the region of (a).

At each level, the discriminator alternates between *x* and *y*. Thus, a node *N* at level 0 (the root) would have in its left subtree only nodes whose *x* values are less than *Nx* (because *x* is search key 0, and 0mod2=0). The right subtree would contain nodes whose *x* values are greater than *Nx*. A node *M* at level 1 would have in its left subtree only nodes whose *y* values are less than *My*. There is no restriction on the relative values of *Mx* and the *x* values of *M* 's descendants, because branching decisions made at *M* are based solely on the *y* coordinate.

The region containing the points is (arbitrarily) restricted to a 128×128 square, and each internal node splits the search space. Each split is shown by a line, vertical for nodes with *x* discriminators and horizontal for nodes with *y* discriminators. The root node splits the space into two parts; its children further subdivide the space into smaller parts. The children's split lines do not cross the root's split line. Thus, each node in the k - d tree helps to decompose the space into rectangles that show the extent of where nodes can fall in the various subtrees.

Searching a k - d tree for the record with a specified xy-coordinate is like searching a BST, except that each level of the k - d tree is associated with a particular discriminator.

* *Insertion in a 2d tree*

Suppose we need to add a node p = (a, b). Start with the root node t:

1. If t is null , then add the node p at t
2. If value of p (a or b) depending on the cutting dimension is less than value of t for that dimension , go to the left subtree of t
3. If value of p (a or b) depending on the cutting dimension is greater than value of t for that dimension, go to the right subtree of t

insert(Point x, KDNode t, int cd) {

if t == null:

t = new KDNode(x)

else if (x == t.data):

// error! duplicate

else if (x[cd] < t.data[cd]):

t.left = insert(x, t.left, (cd+1) % DIM)

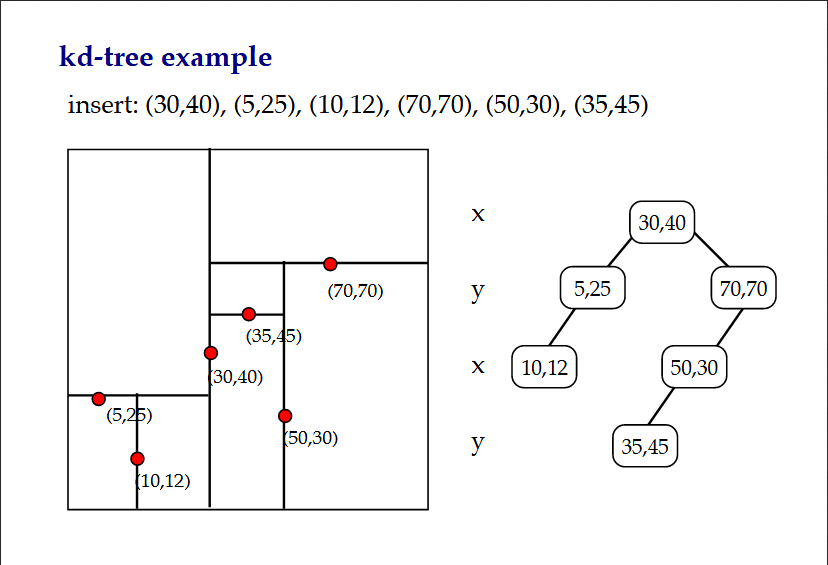
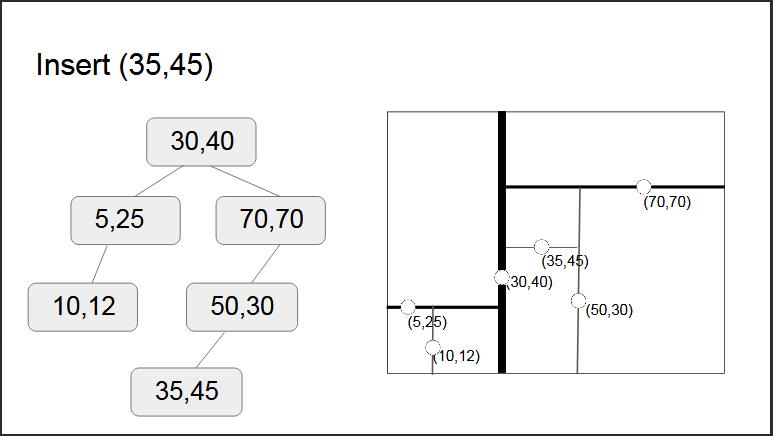
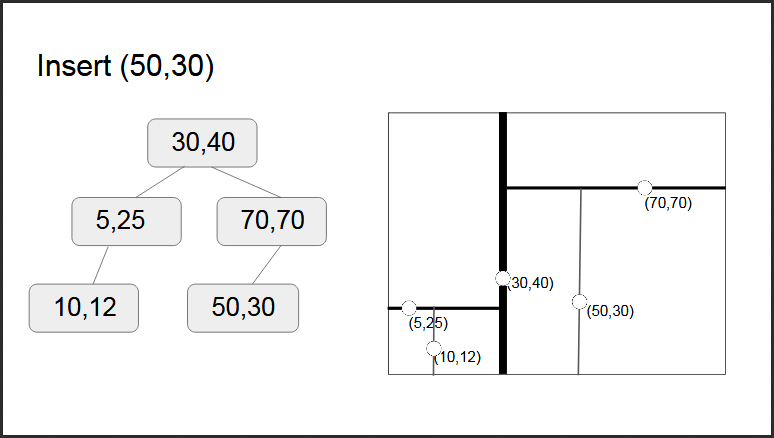
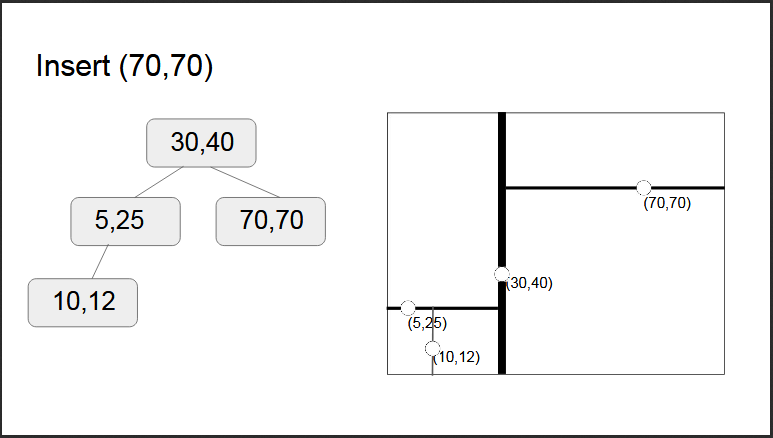
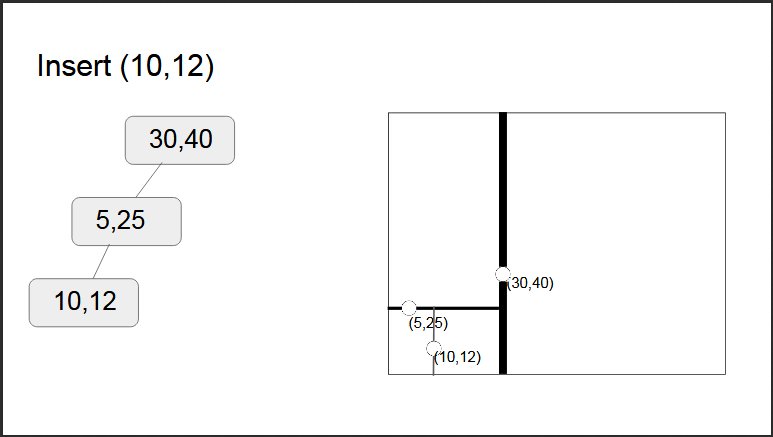
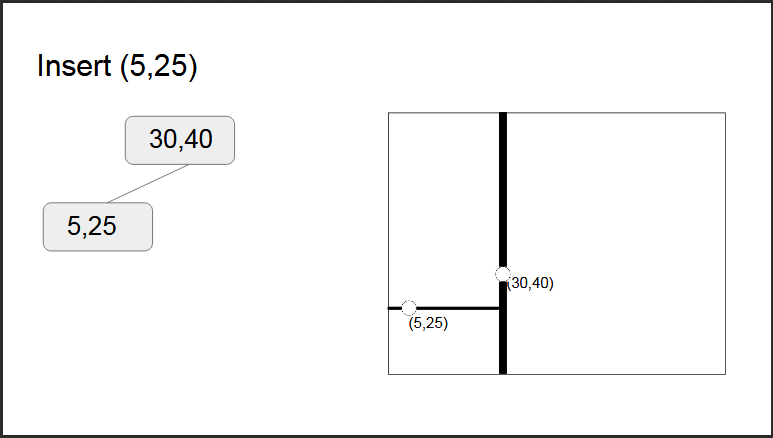
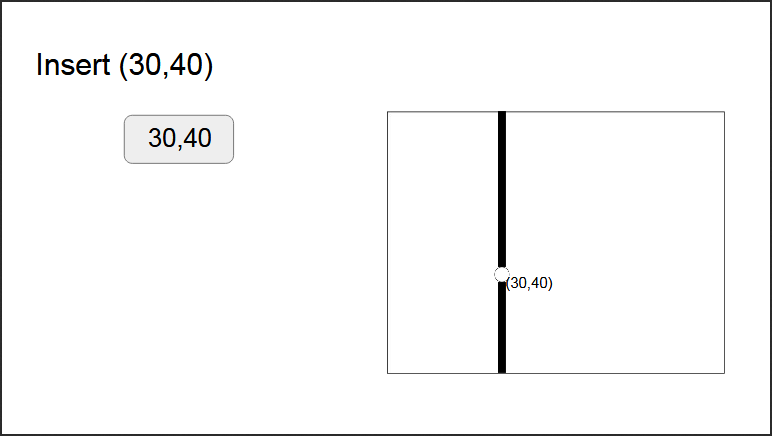
else

t.right = insert(x, t.right, (cd+1) % DIM)

return t

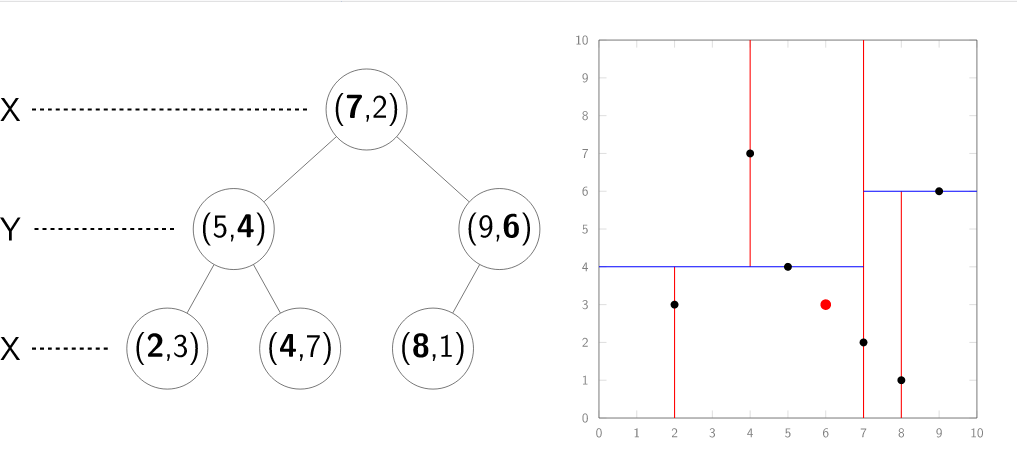
}

The following figures give an example of insertion in 2d tree. Insert: (30, 40), (5, 25), (10, 12), (70, 70), (50, 30), (35, 45).

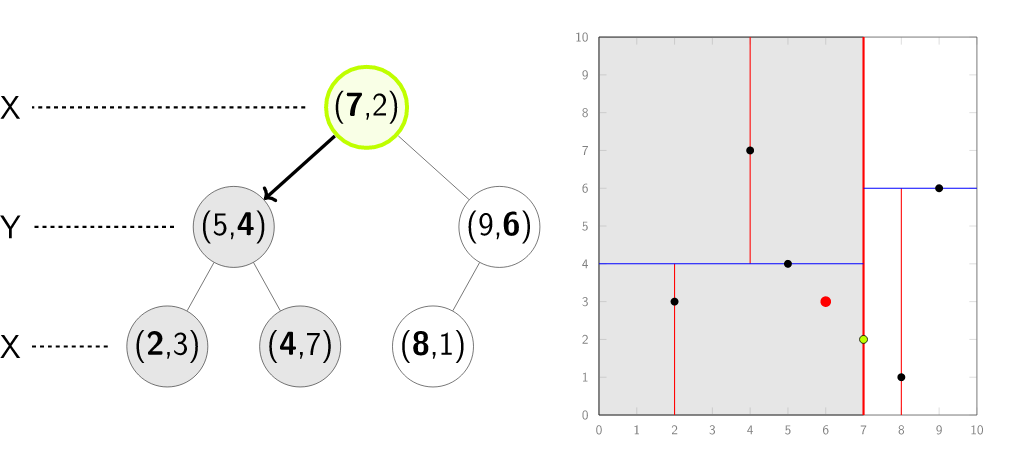


* *Using kd trees for kNN*

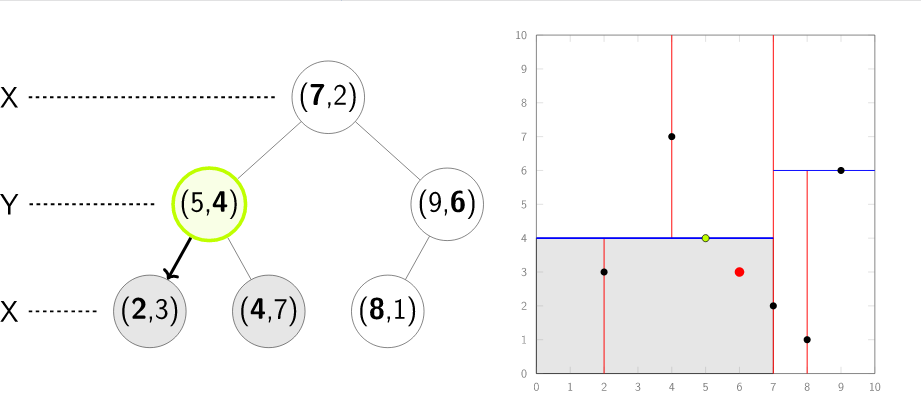
Suppose we have the same *k*-d tree as in Figure 1 and that the target point (in red) is (6,3), as shown in the figure below. We wish to find the point in the *k* -d that is closest to the target; i.e., to determine which of the black points is closest to the red point.



To start the search, we begin a depth-first search to find the leaf node within the same splitting plane as the target node. At the root of the tree, the node is defined by the point (7, 2), with the splitting plane based on the first coordinate. Since 6<7 (using the target coordinate’s first dimension) we search the left subtree (the grey region in the figure below).

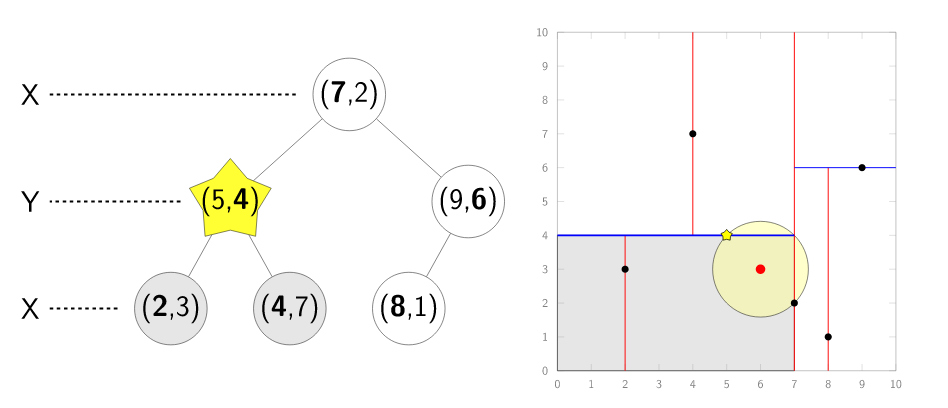
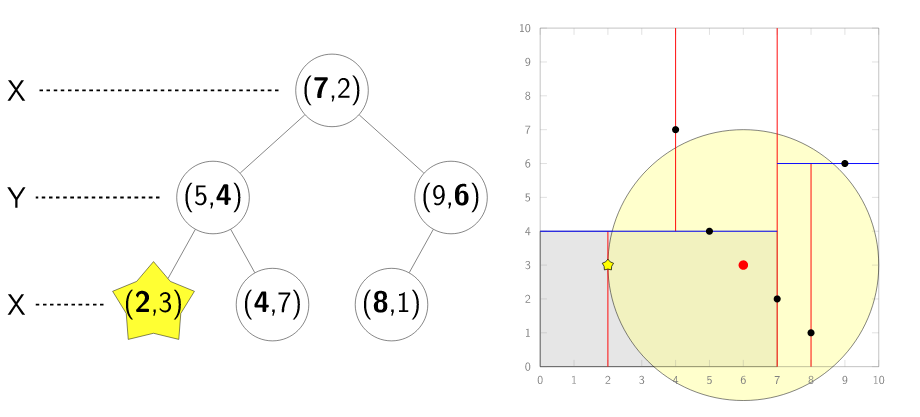


The child node is defined by (5, 4), and the splitting plane is based on the second coordinate. Again, the target node (6, 3) is in the left subtree, so we split left.

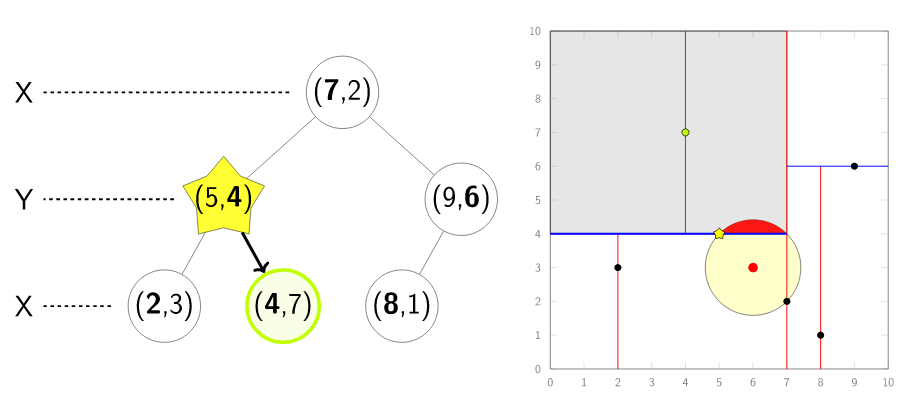


At the next step, we hit a leaf node (2, 3). At this point, (2, 3) becomes our current best node, and the distance from the target node to (2, 3) defines a “current best” radius, as indicated by the circle below. That is, any point outside of this radius cannot be the closest point to the target, since (2, 3) will always be closer; however, there may be a point within the radius that is closer. We now start the back-traversal to check for other nodes within this radius.

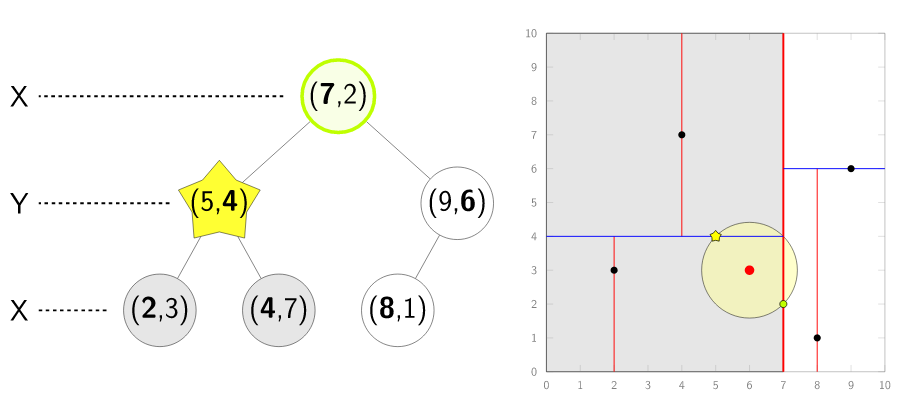
Back at the parent node, (5, 4), we see that it is closer to our target point than the current best of (2, 3). So, (5, 4) is stored as the current best, and we update the radius.



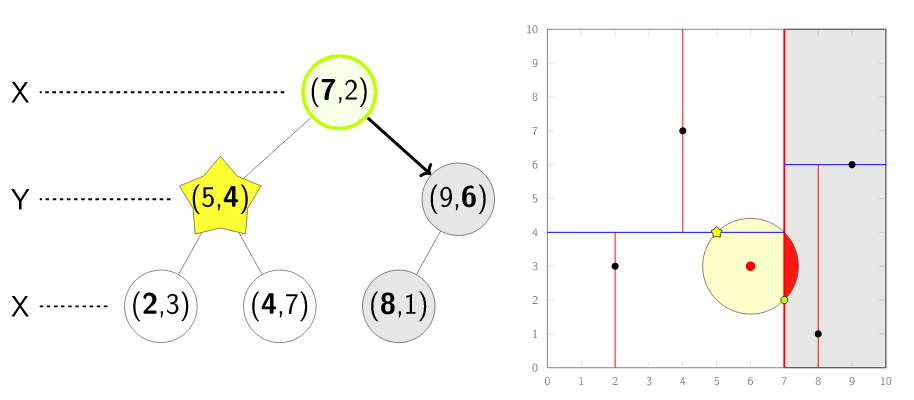
The distance from the target point to the splitting plane for the node (5, 4) is within the current radius, so we must search the other subtree, indicated by the grey region below. This can be visualized as the hypersphere (in 2-d, a circle) intersecting the region opposite the splitting plane, as shown by the red region in the figure below. We descend into the subtree and find a leaf node (4, 7), which is farther away than our current best.



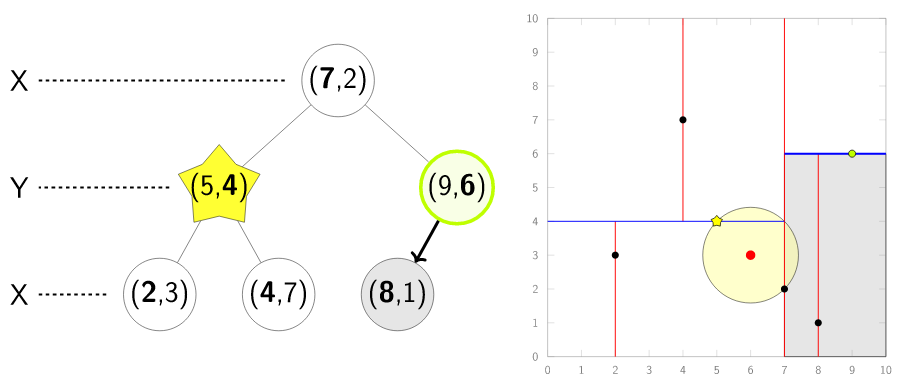
We return all the way to the root node, defined by (7, 2). The distance between this node and the target is exactly equal to the current radius. In this case, we check Point<2>::operator<, which says our current best of (5, 4) is less than (7, 2), so we don’t replace the current best node.



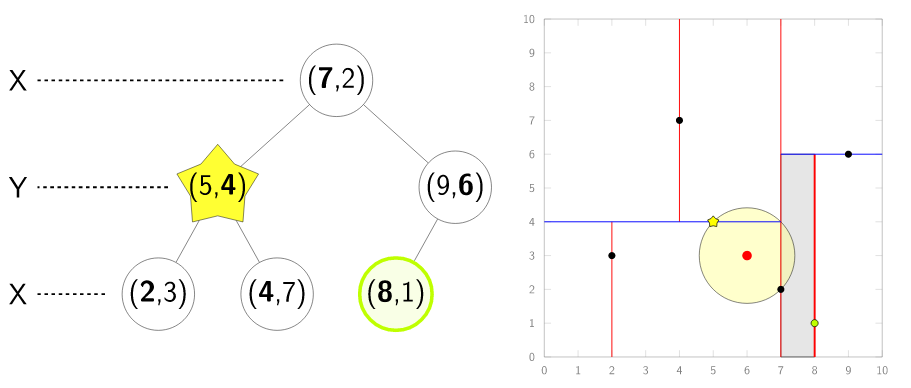
Once again, the distance between the splitting plane defined by (7,2) and the target point is within the current radius (i.e., the red region exists), so we must search the other subtree.



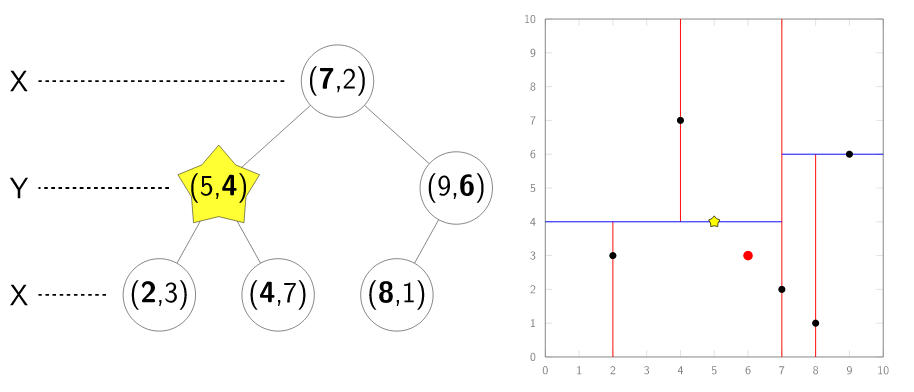
The target point is less than the splitting plane defined by the node at (9, 6), so we first descend into the left subtree.



We encounter a leaf node, (8, 1), but the distance is greater than the current best, so we don’t do anything.



We finally step back up the tree, and find there are no more regions that intersect the hypersphere (i.e., no other rectangles intersect the circle). Therefore, (5, 4) is the nearest neighbour, and our search is complete.



**Program:**

#include<stdio.h>

#include<conio.h>

#include<stdlib.h>

struct Node{

int data[10];

struct Node\* left;

struct Node\* right;

};

int k;

void printPoint(int data[])

{

printf("(");

for (int j=0;j<k;j++){

printf(" %d ",data[j]);

}

printf(")");

}

int search(struct Node \*head, struct Node \*new)

{

int new\_val, head\_val;

int flag = 0;

int cd=0;

while (flag==0)

{

printf("\nCutting dimension: %d",cd%k);

new\_val = new->data[cd%k];

head\_val = head->data[cd%k];

cd++;

if (new\_val < head\_val)

{

printf("\nValue ");

printPoint(new->data);

printf(" is smaller than ");

printPoint(head->data);

if (head->left==NULL){

head->left = new;

printf("\nInserted ");

printPoint(new->data);

printf(" as left child of ");

printPoint(head->data);

printf("\n");

return 0;}

else if (head->left!=NULL){

head = head->left;

printf("\nGoing into left subtree...\n");

continue;

}

}

if (new\_val > head\_val)

{

printf("\nValue ");

printPoint(new->data);

printf(" is greater than ");

printPoint(head->data);

if (head->right==NULL){

head->right = new;

printf("\nInserted ");

printPoint(new->data);

printf(" as right child of ");

printPoint(head->data);

printf("\n");

return 0;}

else if (head->right!=NULL){

head = head->right;

printf("\nGoing into right subtree...\n");

continue;}

}

}

return 0;

}

void printInorder(struct Node \*node)

{

if (node == NULL)

return;

printInorder(node->left);

printPoint(node->data);

printInorder(node->right);

}

void main(){

int n,val;

struct Node \*nodes[20];

printf("Enter dimension k for kd tree:");

scanf("%d",&k);

printf("Enter number of nodes in %dd tree:",k);

scanf("%d",&n);

int flag=0;

for (int i=0;i<n;i++)

{

nodes[i] = (struct Node\*)malloc(sizeof(struct Node));

printf("\nEnter value of point %d:", i+1);

for (int j=0;j<k;j++){

scanf("%d",&val);

nodes[i]->data[j] = val;

}

nodes[i]->left=NULL;

nodes[i]->right=NULL;

if (i>0){

flag = search(nodes[0], nodes[i]);

if (flag == 1){

i+=1;

}}

}

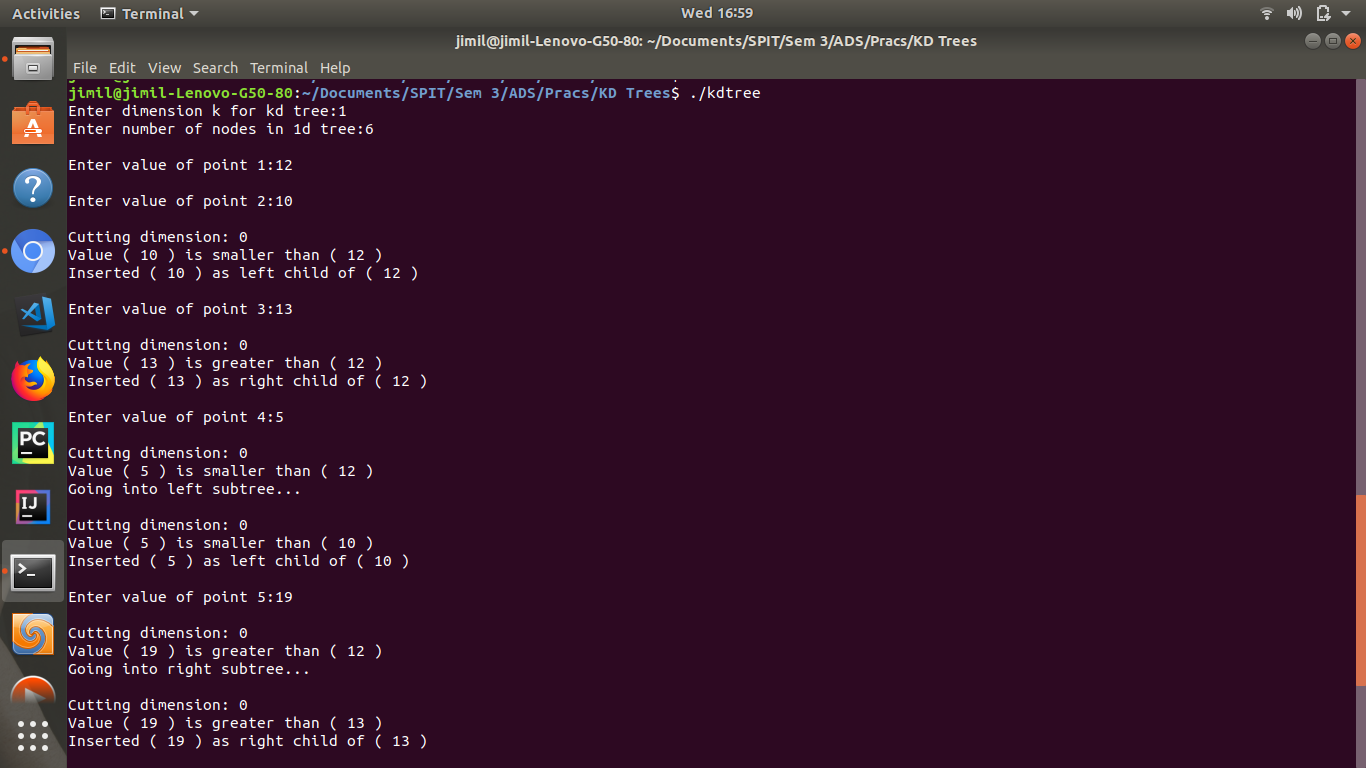
printf("\nInorder traveral:\n");

printInorder(nodes[0]);

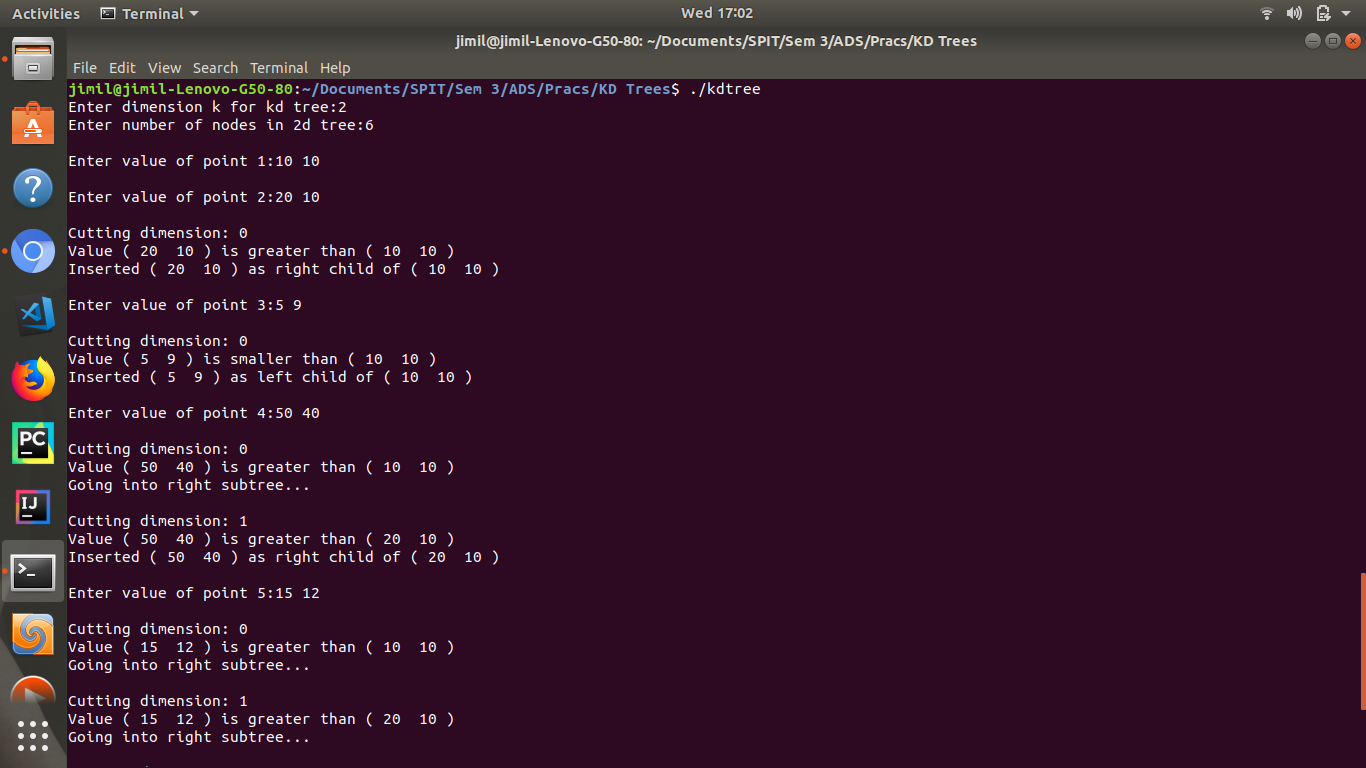
}

**Output:**

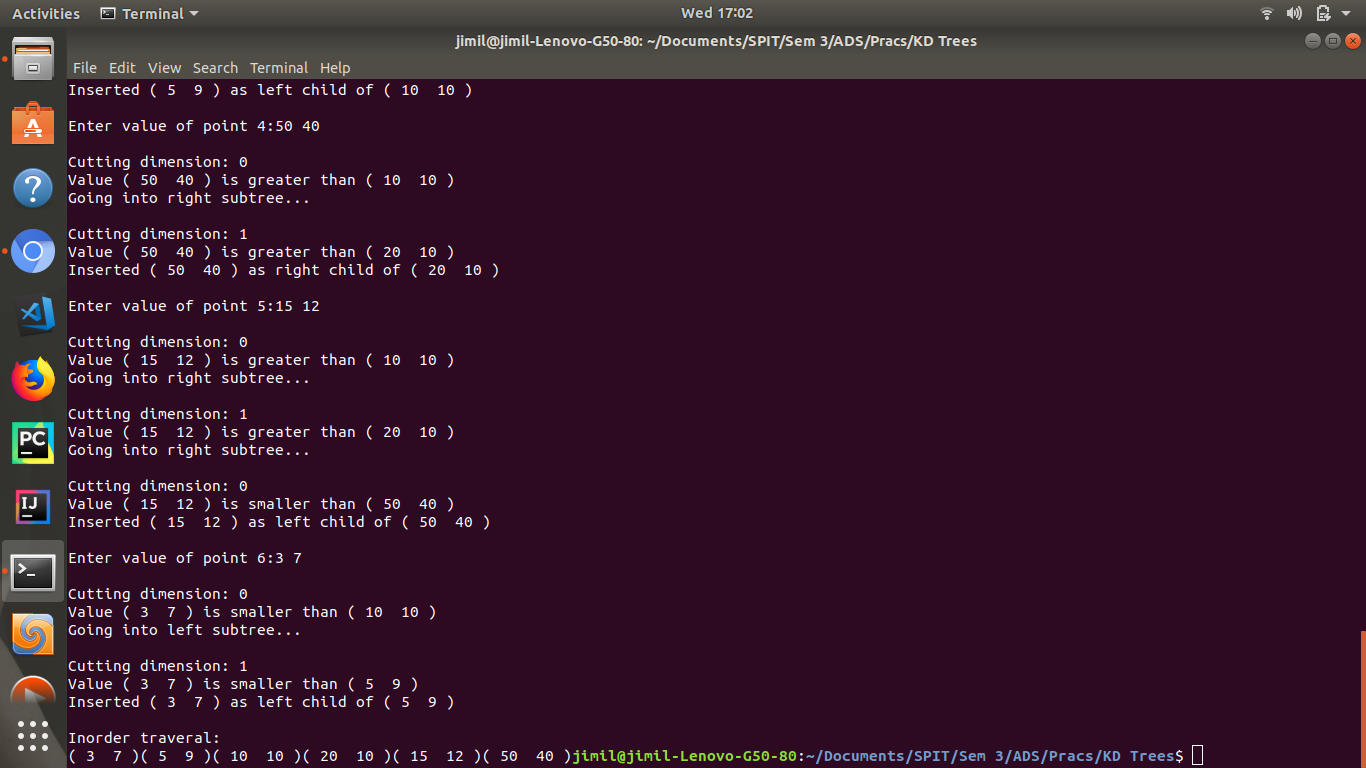
* **1d tree works as binary search tree**



* **2d tree where k = 2**



* **3d tree where k = 3**



**Result:**

* A *k*-d tree is special purpose data structure used to organize elements that can be described by locations in *k*-dimensional space. It can be considered a space-partitioning data structure because it recursively subdivides a space into two convex sets. These sets are rectangular regions of the space called [hyper rectangles](https://en.wikipedia.org/wiki/Hyperrectangle).
* *K* - d trees are particularly useful for implementing nearest neighbour search, which is an optimization problem for finding the closest element in *k*-dimensional space. Some other applications include database queries using multiple keys, optimization and ray tracing.

**Conclusion:**

Thus through this experiment we have understood kd trees and implemented a creation and insertion algorithm for 2d trees. A kd tree is a space partitioning data structure that describes locations in k dimensional space. Insertion takes place by traversing and comparing values at the cutting dimension. Kd trees are very useful for nearest neighbour search to find the closest element in k dimensional space.