DS PROJECT

KD TREES

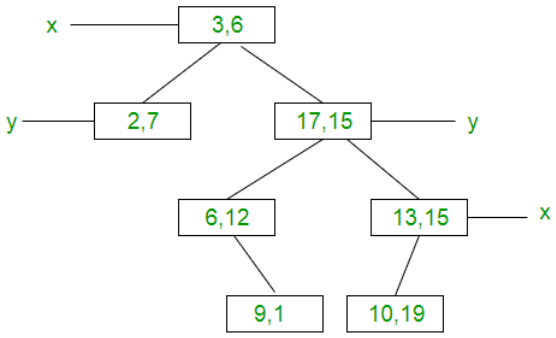
Team Members:
AFRA SADAT
MAHNOOR MUNIR

SUBMITTED TO:
MAM SIDRA EJAZ

KD TREES

A K-D Tree(also called as K-Dimensional Tree) is a binary search tree where data in each node is a K-Dimensional point in space. In short, it is a space partitioning(details below) data structure for organizing points in a K-Dimensional space. Points to the left of this space are represented by the left subtree of that node and points to the right of the space are represented by the right subtree.

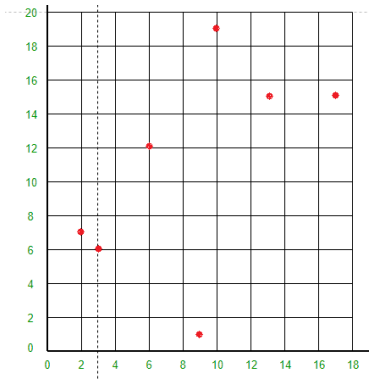
The root would have an x-aligned plane, the root's children would both have y-aligned planes, the root's grandchildren would all have x-aligned planes, and the root's great-grandchildren would all have y-aligned planes and so on.



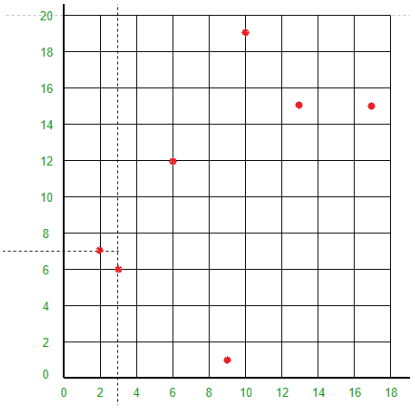
How is space partitioned?

All 7 points will be plotted in the X-Y plane as follows:

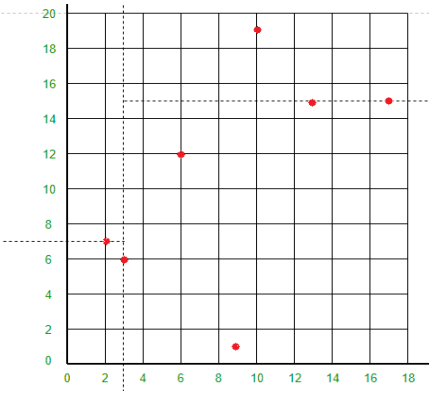
- Point (3, 6) will divide the space into two parts: Draw line X = 3.



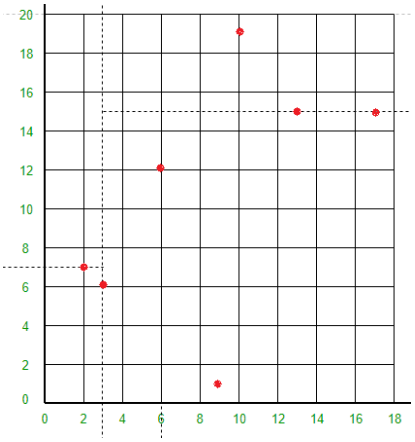
- Point (2, 7) will divide the space to the left of line X = 3 into two parts horizontally. Draw line Y = 7 to the left of line X = 3



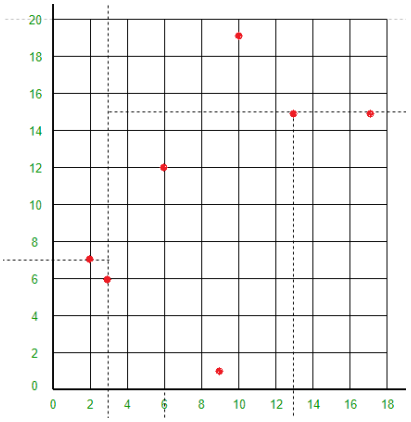
- Point (17, 15) will divide the space to the right of line X = 3 into two parts horizontally. Draw line Y = 15 to the right of line X = 3.



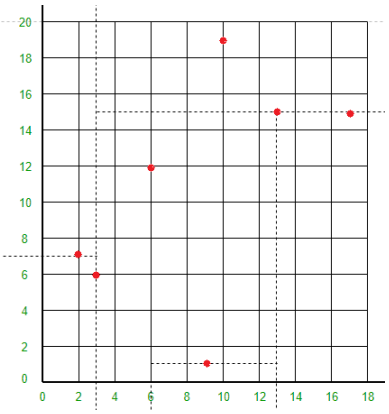
- Point (6, 12) will divide the space below line Y = 15 and to the right of line X = 3 into two parts. Draw line X = 6 to the right of line X = 3 and below line Y = 15.



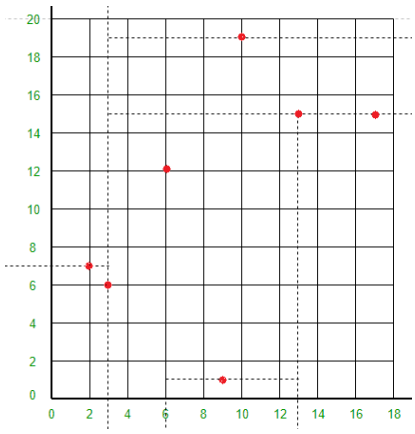
- Point (13, 15) will divide the space below line Y = 15 and to the right of line X = 6 into two parts. Draw line X = 13 to the right of line X = 6 and below line Y = 15.



- Point (9, 1) will divide the space between lines X = 3, X = 6 and Y=15 into two parts. Draw line Y = 1 between lines X = 3 and X = 6.



- Point (10, 19) will divide the space to the right of line X = 3 and above line Y = 15 into two parts. Draw line Y = 19 to the right of line X = 3 and above line Y = 15.



CODE:

```
// KD trees.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include <iostream>// A C++ program to demonstrate delete in K D tree
using namespace std;

const int k = 2;

// A structure to represent node of kd tree
struct Node
{
	int point[k]; // To store k dimensional point
	Node *left, *right;
};

// A method to create a node of K D tree
struct Node* newNode(int arr[])
{
	struct Node* temp = new Node;

	for (int i=0; i<k; i++)
		temp->point[i] = arr[i];

	temp->left = temp->right = NULL;
	return temp;
}

// Inserts a new node and returns root of modified tree
// The parameter depth is used to decide axis of comparison
```

```
Node *insertRec(Node *root, int point[], unsigned depth)
{
	// Tree is empty?
	if (root == NULL)
		return newNode(point);

	// Calculate current dimension (cd) of comparison
	unsigned cd = depth % k;

	// Compare the new point with root on current dimension 'cd'
	// and decide the left or right subtree
	if (point[cd] < (root->point[cd]))
		root->left = insertRec(root->left, point, depth + 1);
	else
		root->right = insertRec(root->right, point, depth + 1);

	return root;
}

// Function to insert a new point with given point in
// KD Tree and return new root. It mainly uses above recursive
// function "insertRec()"
Node* insert(Node *root, int point[])
{
	return insertRec(root, point, 0);
}

// A utility method to determine if two Points are same
// in K Dimensional space
bool arePointsSame(int point1[], int point2[])
{
	// Compare individual pointinate values
	for (int i = 0; i < k; ++i)
		if (point1[i] != point2[i])
			return false;

	return true;
}

// Copies point p2 to p1
void copyPoint(int p1[], int p2[])
{
for (int i=0; i<k; i++)
```

```
	p1[i] = p2[i];
}

// Function to delete a given point 'point[]' from tree with root
// as 'root'. depth is current depth and passed as 0 initially.
// Returns root of the modified tree.
Node *deleteNodeRec(Node *root, int point[], int depth)
{
	// Given point is not present
	if (root == NULL)
		return NULL;

	// Find dimension of current node
	int cd = depth % k;

	// If the point to be deleted is present at root
	if (arePointsSame(root->point, point))
	{
		// 2.b) If right child is not NULL
		if (root->right != NULL)
		{
			// Find minimum of root's dimension in right subtree
			Node *min = findMin(root->right, cd);

			// Copy the minimum to root
			copyPoint(root->point, min->point);

			// Recursively delete the minimum
			root->right = deleteNodeRec(root->right, min->point, depth+1);
		}
		else if (root->left != NULL) // same as above
		{
			Node *min = findMin(root->left, cd);
			copyPoint(root->point, min->point);
			root->right = deleteNodeRec(root->left, min->point, depth+1);
		}
		else // If node to be deleted is leaf node
		{
			delete root;
			return NULL;
		}
		return root;
	}
```

```
      // 2) If current node doesn't contain point, search downward
      if (point[cd] < root->point[cd])
            root->left = deleteNodeRec(root->left, point, depth+1);
      else
            root->right = deleteNodeRec(root->right, point, depth+1);
      return root;
}

// Function to delete a given point from K D Tree with 'root'
Node* deleteNode(Node *root, int point[])
{
// Pass depth as 0
return deleteNodeRec(root, point, 0);
}
// Searches a Point represented by "point[]" in the K D tree.
// The parameter depth is used to determine current axis.
bool searchRec(Node* root, int point[], unsigned depth)
{
    // Base cases
    if (root == NULL)
        return false;
    if (arePointsSame(root->point, point))
        return true;

    // Current dimension is computed using current depth and total
    // dimensions (k)
    unsigned cd = depth % k;

    // Compare point with root with respect to cd (Current dimension)
    if (point[cd] < root->point[cd])
        return searchRec(root->left, point, depth + 1);

    return searchRec(root->right, point, depth + 1);
}

// Searches a Point in the K D tree. It mainly uses
// searchRec()
bool search(Node* root, int point[])
{
    // Pass current depth as 0
    return searchRec(root, point, 0);
}
```

```
// Driver program to test above functions
int _tmain(int argc, _TCHAR* argv[])

{
	struct Node *root = NULL;
	int points[][k] = {{30, 40}, {5, 25}, {70, 70}, {10, 12}, {50, 30}, {35,
45}};
	int n = sizeof(points)/sizeof(points[0]);
	for (int i=0; i<n; i++)
		root = insert(root, points[i]);

	// Delete (30, 40);
	root = deleteNode(root, points[0]);
	cout << "Root after deletion of (30, 40)\n";
	cout << root->point[0] << ", " << root->point[1] << endl;
	int point1[] = {10, 19};
    ( Search (root, point1))? cout << "Found\n": cout << "No any point Found\n";
    int point2[] = {35, 45};
    (Search(root, point2))? cout << "Found\n": cout << "No any point Found\n";
	system("pause");
	return 0;
}
```