END TERM REPORT

on

Dynamic Cab Lookup

Submitted by

Ojasvi Singh Chauhan (Enroll: R134218111)

Pranjal Srivastava (Enroll: R134218121)

Ramankur Goswami (Enroll: R134218131)

Under the guidance of

Ms. Deepa Joshi

Assistant Professor
Department of Systemics

UPES

UNIVERSITY WITH A PURPOSE

SCHOOL OF COMPUTER SCIENCE

UNIVERSITY OF PETROLEUM & ENERGY STUDIES

Bidholi Campus, Energy Acres, Dehradun – 248007.

May - 2021

UPES

School of Computer Science

University of Petroleum & Energy Studies, Dehradun

<u>Project Proposal Approval Form (2021)</u>

Minor	II

PROJECT TITLE: Dynamic Cab Lookup

ABSTRACT

The project aims to ease the finding of the nearest cab available to the user by implementing the nearest neighbor searching. Searching is one of the most fundamental operations in many complex systems. However, the complexity of the search process would increase dramatically in high-dimensional space. K-dimensional (KD) tree, as a classical data structure, has been widely used in high-dimensional vital data search. As the k-d tree is a special case of binary space partitioning tree, a lot of multidimensional searches can be applied to the same for various purposes such as range searches and creating point clouds, this project is a newer approach to the nearest neighbor searching by integrating dynamic elements and points to give us real time accurate output for the nearest cab available.

KEYWORDS: Nearest Cab; K-d Tree; Multidimensional; Search; Dynamic Elements

INTRODUCTION

Dynamic Cab Lookup is an algorithm which determines the nearest cab to the user based on their coordinates and also gives the exact coordinates of all the cabs available in the vicinity.

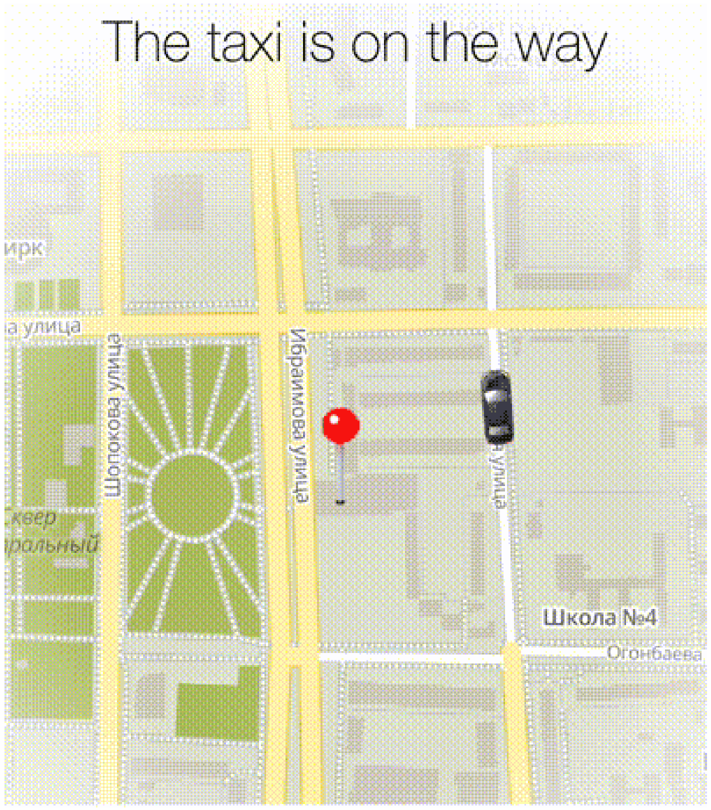
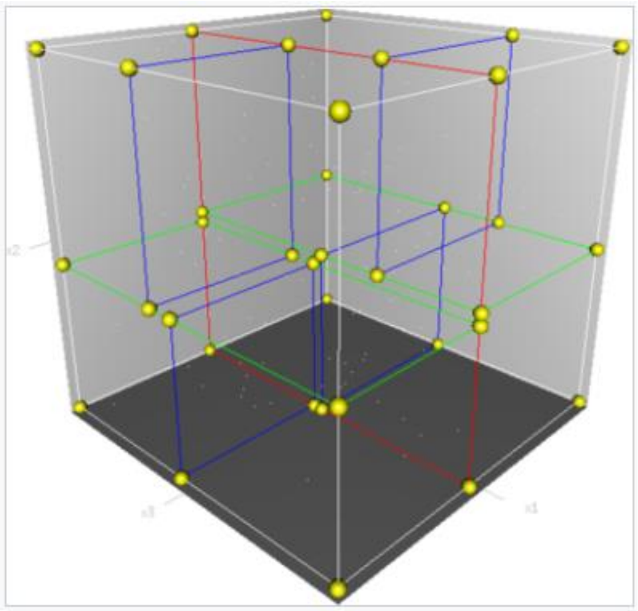


Fig 1 Cab Lookup [1]

This algorithm is based on a data structure known as **KD tree**. The k-d tree is a binary tree in which every leaf node is a k-dimensional point. Every non-leaf node can be thought of as implicitly generating a splitting hyperplane that divides the space into two parts, known as half-spaces. Points to the left of this hyperplane are represented by the left subtree of that node and points to the right of the hyperplane are represented by the right subtree. The hyperplane direction is chosen in the following way: every node in the tree is associated with one of the k dimensions, with the hyperplane perpendicular to that dimension's axis. So, for example, if for a particular split the "x" axis is chosen, all points in the subtree with a smaller "x" value than the node will appear in the left subtree and all points with larger "x" value will be in the right subtree. In such a case, the hyperplane would be set by the x value of the point, and its normal would be the unit x axis. [2]



A 3-dimensional *k*-d tree. The first split (the red vertical plane) cuts the root cell (white) into two subcells, each of which is then split (by the green horizontal planes) into two subcells. Finally, four cells are split (by the four blue vertical planes) into two subcells. Since there is no more splitting, the final eight are called leaf cells.

Fig 2 A 3-dimensional k-d tree [3]

There are many applications of k-d tree and many tasks can be achieved with this data structure. But **Nearest Neighbor Search** and **Range Search** are the major ones with most importance and their practical applications are abundant.

PROBLEM STATEMENT

To find the number of cabs present in the queried area and what is the distance of the nearest cab to the user.

LITERATURE REVIEW

Title	Link	Author	Remarks
The k-d tree data structure and a proof for neighborhood computation in expected logarithmic time	[4]	Martin Skrodzki	This paper gives us the basic introduction to the data structure of k-d trees and its multiple uses.
An Advanced k Nearest Neighbor Classification Algorithm Based on KD-tree	[5]	Wenfeng Hou , Daiwei Li , Chao Xu , Haiqing Zhang , Tianrui Li	This paper emphasizes on the techniques of data classification and nearest neighbor searching.
Building a Balanced k-d Tree in O(k nlogn) Time	[6]	Russell A. Brown	This paper develops the multidimensional binary search tree (or k-d tree, where k is the dimensionality of the search space) as a data structure for storage of information to be retrieved by associative searches.
Geographical Data Structures Compared: A Study of Data Structures Supporting Region Queries	[7]	J.B. Rosenberg	This paper compares three data structures that support area operations on 2-space: linked lists, quad trees, and k-d trees. The conclusion of this paper is that in applications where region search on large problems is crucial k-d trees provide superior performance

OBJECTIVES

To create an application that gives the coordinates of the closest cab to the user.

Sub Objectives:

- To perform the rectangular search query.
- To locate the nearest neighbor to the user.
- To create an interactive GUI for communicating with the system.

METHODOLOGY

In order to perform the 2d orthogonal range search, we need to understand the rudimentary methods that were previously used to achieve range search. A way to achieve this can be brute force through **grid implementation**.

The algorithm for grid implementation is as follows:

- Divide the space into M-by-M grid of squares.
- Create list of points contained in each square.
- Use 2d array to directly index relevant square.
- Insert: add (x,y) to list of corresponding square.
- Range Search: examine only squares that intersect 2d range query.

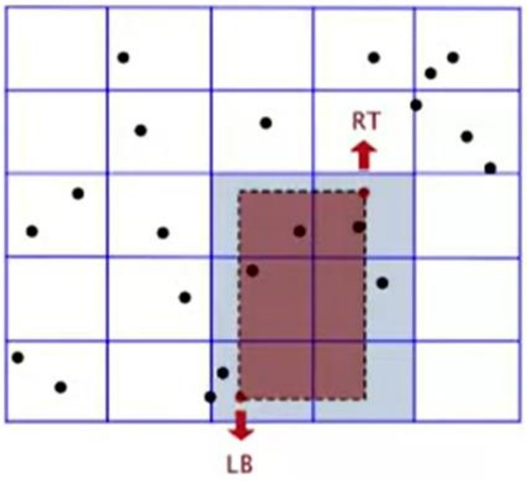


Fig 3. Grid Implementation [8]

Space: M^2 + N
Time: 1 + N/M^2 per square examined, on average.

Choose grid size to tune the performance

- Too small: wastes space
- Too large: too many points per square
- Rule of thumb: $\sqrt{N}$ by $\sqrt{N}$

Problem:

Clustering is a well known phenomenon in geometric data.

- Lists are too long, even though average length is short.
- Need data structure that adapts gracefully to data.

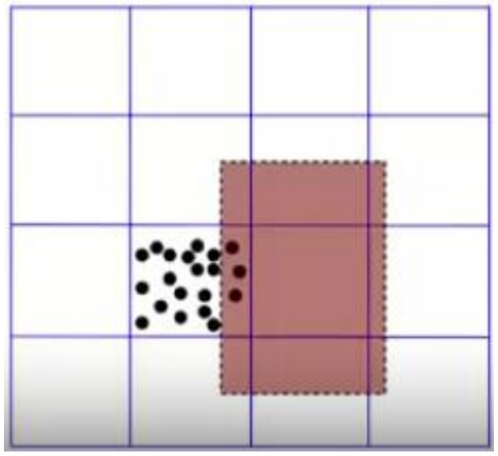


Fig 4. Clustering in geometric data [9]

To overcome the problem of clustering, we are going to use a tree-based data structure (k-d Tree) to store the same.

Construction of k-d tree:

To start with we have a set of points in a k-dimensional space.

In case of binary search trees, the binary partition of the real line at each internal node is represented by a ***point*** on the real line. Similarly, in case of a 2 dimensional k-d tree, the binary partition of the 2 dimensional cartesian plane at each internal node is represented by a ***line*** in the plane.

Essentially, we could choose any line passing through the point represented by the internal node to partition the 2 dimensional cartesian plane.

Level **0**: Choose the partitioning line perpendicular to the *first dimension* (**X** in this case) and passing through the point represented by the node in question.

Level **1**: Choose the partitioning line perpendicular to the *second dimension* (**Y** in this case) and passing through the point represented by the node in question.

:
:

Level **k-1**: Choose the partitioning line perpendicular to the *kth dimension* and passing through the point represented by the node in question.

Level **k**: Choose the partitioning line perpendicular to the *first dimension* (**X** in this case) and passing through the point represented by the node in question.

So basically, at each level we alternate between the X and Y dimensions in order to choose a partitioning line at each internal node of the k-d tree.

The labels that we see beside each of the nodes of the k-d tree represent the choice of the dimension for the partitioning line at the nodes on that level.

Let's now see how our 2 dimensional k-d tree partitions the 2 dimensional plane:

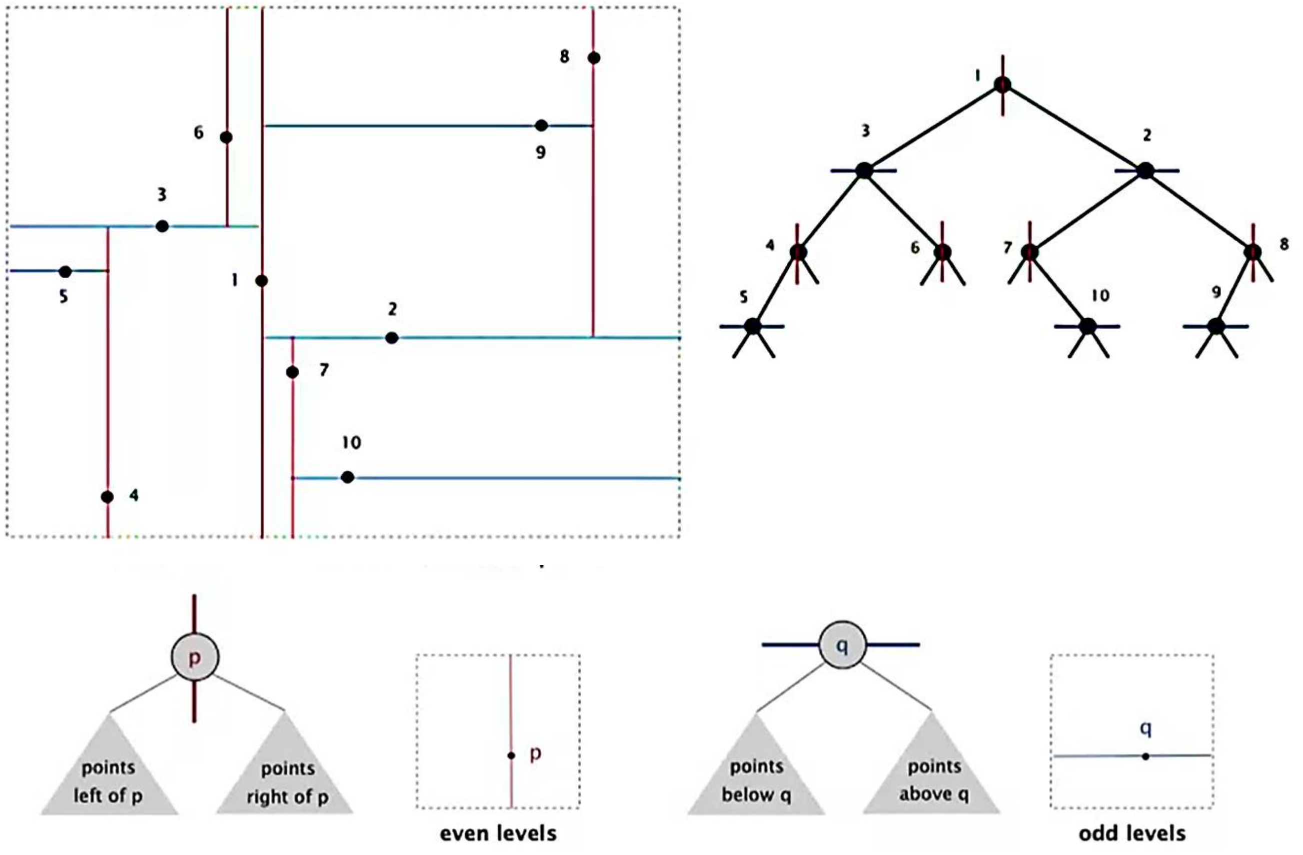


Fig 5. K-D Tree [10]

1. **Range Search in a 2d tree:**

Goal: Find all points in a query axis-aligned rectangle.

- Check if point in node lies in given rectangle.
- Recursively search left/bottom (if any could fall in rectangle).
- Recursively search right/top (if any could fall in rectangle).

Typical Case: R + log N.
Worst case (assuming tree is balanced): R + $\sqrt{\boldsymbol{N}}$.

2. **Nearest Neighbour Search:**

Goal: Find closest point to query point.

- Check distance from point in node to query point.
- Recursively search left/bottom (if it could contain a closer point).
- Recursively search right/top (if it could contain a closer point).
- Organize method so that it begins by searching for query point.

Typical Case: log N.
Worst case (even if tree is balanced): N

SYSTEM REQUIREMENTS

Hardware:

- RAM: 4GB
- Disk Space: 500 MB

Software:

- Sublime Text(or any other C++ IDE)
- Gcc Compiler should be installed.
- Qt Creator 6.0

Operating System:

- Windows or Linux.

SCHEDULE

(pert chart)

SCHEDULE

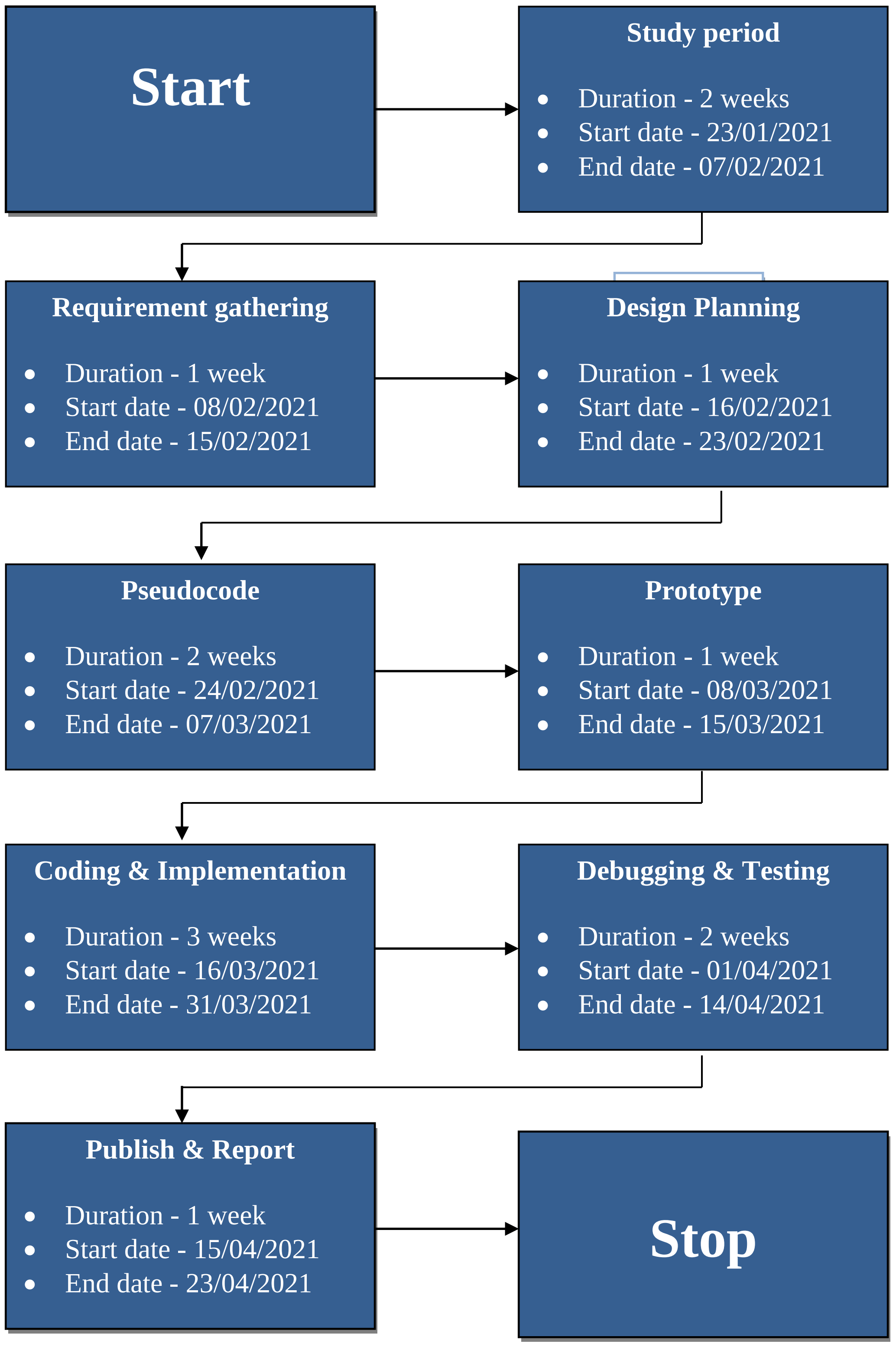


Fig 6 Pert chart

Code:

```
//#include <bits/stdc++.h>
#include <iostream>
#include <vector>
#include <climits>
#include <cmath>

using namespace std;
#define endl '\n'

const int k = 2;

class node {
public:
	int point[k]; // To store k dimensional point
	node* left;
	node* right;

	// constructor to create a node of KD tree
	node(int arr[]) {
		for (int i = 0; i < k; i++) {
			point[i] = arr[i];
		}
		left = NULL;
		right = NULL;
	}
};

void print(node* root) {
	if (root == NULL) {
		return;
	}
	cout << endl;
	for (int i = 0; i < k; i++)
		cout << root->point[i] << ",";
	cout << endl;
	print(root->left);
	print(root->right);
}

void Alert() {
	cout << endl;
	cout << "No cabs are available at the moment" << endl;
	cout << endl;
}

// Inserts a new node and returns root of modified tree
// The parameter depth is used to decide axis of comparison
node* insert(node* root, int point[], unsigned depth) {
	// Tree is empty?
	if (root == NULL)
		return new node(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  = insert(root->left, point, depth + 1);
	else
		root->right = insert(root->right, point, depth + 1);

	return root;
}

bool isInsideRect(int x1, int y1, int x2, int y2, int x, int y) {
	if (x > x1 and x < x2 and y > y1 and y < y2)
		return true;
	return false;
}

void rangeSearch2D(node* root, unsigned depth, int rectangle[][2], vector<pair<int,
int>> &count) {
	if (root == NULL) {
		//cout<<"hit base case"<<endl;
		return;
	}
	//cout<<root->point[0]<<","<<root->point[1]<<endl;

	unsigned cd = depth % k;
	int x = root->point[0];
	int y = root->point[1];
	int recX1 = rectangle[0][0];
	int recY1 = rectangle[0][1];
	int recX2 = rectangle[1][0];
	int recY2 = rectangle[1][1];

	// plain is getting divided vertically
	if (cd == 0) {
		// rectangle is on the left side of dividing line
		if (x > recX1 and x > recX2) {
			//cout<<"rectangle is on the left side"<<endl;
			rangeSearch2D(root->left, depth + 1, rectangle, count);
		}
		// rectangle is on the right side of dividing line
		else if (x < recX1 and x < recX2) {
			//cout<<"rectangle is on the right side"<<endl;
			rangeSearch2D(root->right, depth + 1, rectangle, count);
		}
		// dividing line is intersecting the rectangle
		else if (x >= recX1 and x <= recX2) {
			//cout<<"vertically intersecting"<<endl;

			if (isInsideRect(recX1, recY1, recX2, recY2, x, y))
				count.push_back({x, y});

			rangeSearch2D(root->left, depth + 1, rectangle, count);
			rangeSearch2D(root->right, depth + 1, rectangle, count);
		}
	}

	// plain is dividing horizontally
	else {
		// rectangle is below
		if (y > recY1 and y > recY2) {
			//cout<<"rectangle is below"<<endl;
			rangeSearch2D(root->left, depth + 1, rectangle, count);
		}
		// rectangle is above
		else if (y < recY1 and y < recY2) {
			//cout<<"rectangle is above"<<endl;
			rangeSearch2D(root->right, depth + 1, rectangle, count);
		}
		// dividing line is intersecting the rectangle
		else if (y >= recY1 and y <= recY2) {
			//cout<<"horizontally intersecting"<<endl;

			if (isInsideRect(recX1, recY1, recX2, recY2, x, y))
				count.push_back({x, y});

			rangeSearch2D(root->left, depth + 1, rectangle, count);
			rangeSearch2D(root->right, depth + 1, rectangle, count);
		}
	}
}

double dist(int point[], int p[]) {
	int x1 = point[0];
	int y1 = point[1];
	int x2 = p[0];
	int y2 = p[1];
	return sqrt((x2 - x1) * (x2 - x1) + (y2 - y1) * (y2 - y1));
```

```
}

node* nearestNeighbourSearch(node* current, int p[], node* nearest, double &closest)
{
	if (current == NULL) {
		//cout<<"hit base case"<<endl;
		closest = dist(nearest->point, p);
		return nearest;
	}

	if (dist(current->point, p) < dist(nearest->point, p)) {
		//cout<<"node is farther than current nearest"<<endl;
		nearest = current;
	}

	node* left = nearestNeighbourSearch(current->left, p, nearest, closest);
	if (dist(left->point, p) < dist(nearest->point, p)) {
		//cout<<"new champion: "<<left->point[0]<<","<<left->point[1]<<endl;
		nearest = left;
	}

	node* right = nearestNeighbourSearch(current->right, p, nearest, closest);
	if (dist(right->point, p) < dist(nearest->point, p)) {
		//cout<<"new champion: "<<right->point[0]<<","<<right->point[1]<<endl;
		nearest = right;
	}

	return nearest;
}


void option1(node* root, int points[][k]) {

	cout << "Enter the coordinates of rectangle: " << endl;
	cout << "Left Bottom: ";
	int rectangle[2][2];
	int x1, y1, x2, y2;
	cin >> x1 >> y1;
	rectangle[0][0] = x1;
	rectangle[0][1] = y1;
	cout << "Top Right: ";
	cin >> x2 >> y2;
	rectangle[1][0] = x2;
	rectangle[1][1] = y2;

	vector<pair<int, int>> count;
	rangeSearch2D(root, 0, rectangle, count);
	if (count.empty()) {
		cout << "There are no cabs inside the Rectangle" << endl;
		cout << endl;
	}
	else {
		cout << "Points inside the rectangle are: " << endl;
		for (auto p : count) {
			cout << "(" << p.first << "," << p.second << ")" << endl;
		}
		cout << endl;
	}
}

void option2(node* root, int points[][k]) {

	double closest = INT_MAX;
	cout << "Enter the query point" << endl;
	int p[k];
	for (int i = 0; i < k; i++) {
		cin >> p[i];
	}
	node* ans = nearestNeighbourSearch(root, p, root, closest);

	cout << "The closest cab is present at: " << ans->point[0] << "," << ans-
>point[1] << " having "
	     << closest << " units distance from you." << endl;

	cout << endl;
}

int main() {

	cout << "Welcome to DYNAMIC CAB LOOKUP" << endl;
	cout << endl;

	cout << "How many cabs are there ?" << endl;
	int c;
	cin >> c;
	cout << endl;
	int points[c][k];

	if (c != 0) {
		cout << "Enter the coordinates of all cabs present in the town" << endl;
		for (int i = 0; i < c; i++) {
			int x, y;
			cin >> x >> y;
			points[i][0] = x;
			points[i][1] = y;
```

```
                }
        }
        int n = sizeof(points) / sizeof(points[0]);

        cout << endl;
        cout << "Inserting all cab's data..." << endl;
        cout << endl;

        node* root = NULL;
        for (int i = 0; i < n; i++)
                root = insert(root, points[i], 0);

        cout << "All data inserted Successfully!!" << endl;
        cout << endl;

        while (true) {
                cout << "1. Perform Rectangular Search" << endl;
                cout << "2. Perform Nearest Neighbour Search" << endl;
                cout << "3. View all data" << endl;
                cout << "4. Exit" << endl;

                int choice;
                cin >> choice;

                switch (choice)
                {
                case 1: root == NULL ? Alert() : option1(root, points);
                        break;

                case 2: root == NULL ? Alert() : option2(root, points);
                        break;

                case 3: root == NULL ? Alert() : print(root);
                        break;

                case 4: exit(0);

                default : cout << "Invalid Choice" << endl;
                }
        }

        return 0;
}
```

REFERENCES

[1] *Backend system for Uber-like map with go | Mad Devs blog*. (2017, February 14). Mad Devs: Custom Software Development Company.

[2] Bentley, J. L. (1975). "Multidimensional binary search trees used for associative searching". *Communications of the ACM*. **18** (9): 509517. doi:10.1145/361002.361007. S2CID 13091446

[3] *K-D tree*. (2005, April 1). Wikipedia, the free encyclopedia. Retrieved March 21, 2021, from https://en.wikipedia.org/wiki/K-d_tree

[4] Skrodzki, Martin. (2019). The k-d tree data structure and a proof for neighborhood computation in expected logarithmic time.

[5] Hou, Wenfeng & Li, Daiwei & Xu, Chao & Zhang, Haiqing & Li, Tianrui. (2018). An Advanced k Nearest Neighbor Classification Algorithm Based on KD-tree. 902-905. 10.1109/IICSPI.2018.8690508.

[6] Russell A. Brown, Building a Balanced *k*-d Tree in O(*kn* log *n*) Time, *Journal of Computer Graphics Techniques (JCGT)*, vol. 4, no. 1, 50-68, 2015
Available online http://jcgt.org/published/0004/01/03/

[7] J. B. Rosenberg, "Geographical Data Structures Compared: A Study of Data Structures Supporting Region Queries," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 4, no. 1, pp. 53-67, January 1985, doi: 10.1109/TCAD.1985.1270098.

[8] Sedgewick, R., 2011. Algorithms. [online] princeton.edu. Available at: <https://www.cs.bu.edu/~snyder/cs112/CourseMaterials/AlgorithmsChapterOne.pdf>

[9] Sedgewick, R., 2011. Algorithms. [online] princeton.edu. Available at: <https://www.cs.bu.edu/~snyder/cs112/CourseMaterials/AlgorithmsChapterOne.pdf>

[10] Sedgewick, R., 2011. Algorithms. [online] princeton.edu. Available at: <https://www.cs.bu.edu/~snyder/cs112/CourseMaterials/AlgorithmsChapterOne.pdf>

Synopsis Draft verified by

Project Guide	**HOD**
Ms. Deepa Joshi	Dr. Neelu J.Ahuja
(Assistant professor)	(Dept. of Systemics)