Ollscoil Teicneolaíochta an Oirdheiscirt
South East Technological University

BSc. Software Development
Year II

Project Report-CA3

Javier Reyes & Konrad Skoczylas

C00312829 & C00309030

09/December/2025

Contents

Project Introduction

Data structures are the backbone of digital services like AWS, Microsoft Azure, Cloud Flare and more. These companies provide corporations with infrastructure that is essential to run efficiently, handling vast amounts of data, storing them and managing security.

As these infrastructures get more complex, it is important to understand the very basics of the data structures that govern the digital world.

Our project is intended to provide a basic idea of the workings of the most used data structures – Hash Tables, how do we store data and handle collision.

In part 1 we explain how we take a username and convert its characters into its ASCII values to get an ASCII integer, then we apply a hashing function that returns an index value which will be used to store the value.

In part 2 we show a graph data structure application, where we model a simple map of 7 Japanese sites in Tokyo This map is represented by an **Adjacency Matrix** to store direct **distances** between sites.

Our objective is to truly understand the basics of these data structures and implement them in code so that it can be executed, and the results tested for each method.

Part 1: Hash Table Application

What is a Hash Table?

"A hash table is a data structure that implements an associative array, also called a dictionary or simply map; an associative array is an abstract data type that maps keys to values, also called a dictionary or simply map; an associative array is an abstract data type that maps keys to values.

A hash table uses a hash function to compute an *index*, also called a *hash code*, into an array of *buckets* or *slots*, from which the desired value can be found. During lookup, the key is hashed, and the resulting hash indicates where the corresponding value is stored. A map implemented by a hash table is called a hash map." (Wikipedia, 2024)

Description of Hash Table application

Our hash table takes **usernames** as values. To generate a corresponding **key**, each character is translated into its ASCII equivalent value and added together to form an **ASCII Integer** number (key).

The ASCII integer is passed to a hash function that **returns** the index number at which the username will be stored in the hash table. The hash function uses the **module** operator with the **size** of the array

Hash Function(i) = i % n

Where ***i*** is an ASCII Integer, and ***n*** is the size of the array
In our implementation, the array size is 20, therefore *n = 20*

Example: User Name = Elysia | ASCII Integer = 615

HashFunction(615) { 615 % 20 }

Result index = 15

Collisions may occur when two different usernames produce the **same index value** when the hash function is applied. Since each index in the array can only store **one value**, therefore we must handle these collisions. We have decided to implement **linear probing**.

With linear probing, if the index returned by the hashing function is already occupied, the algorithm checks for the **next** index number in the array
(index + 1), if that index is also occupied, it continues to check every index sequentially until it founds an index that has an empty slot. This probing wraps around to the beginning of the array if the end has been reached.

Example:

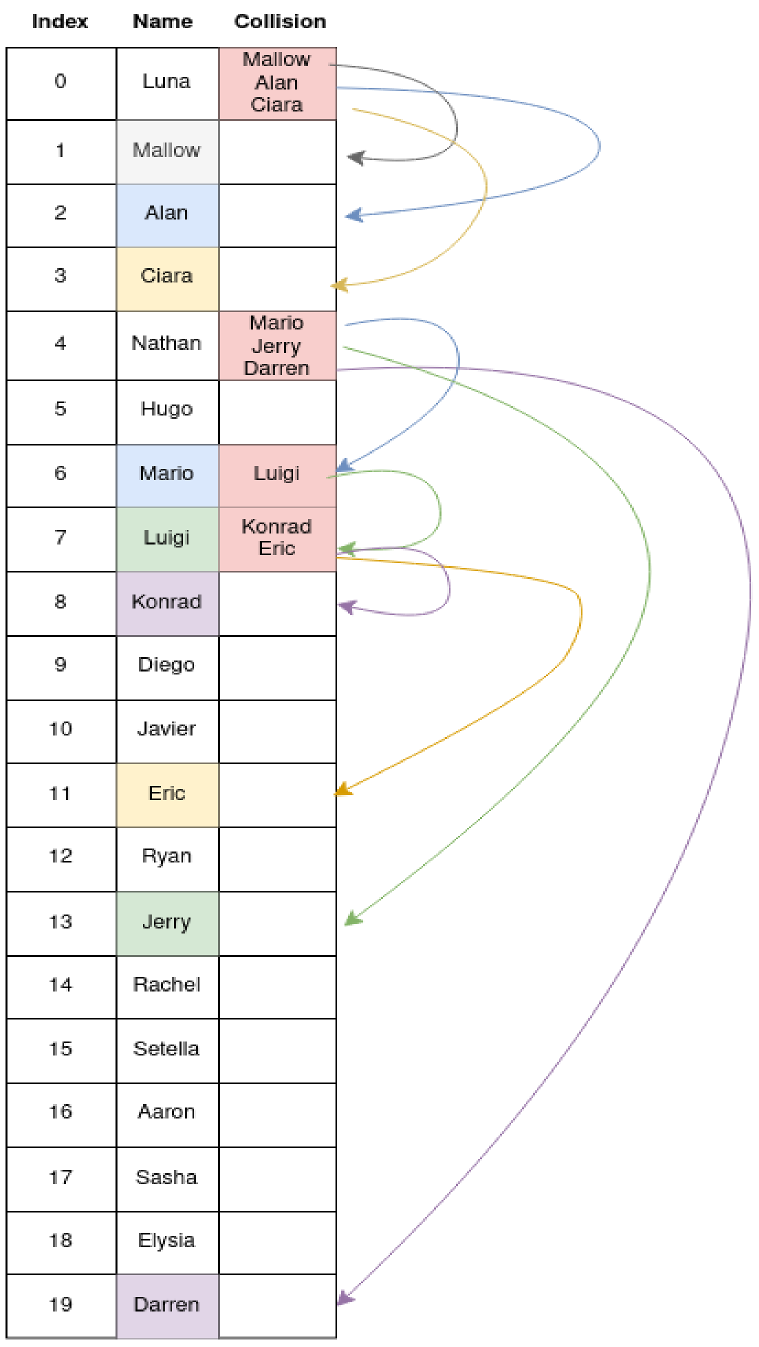
```
- index = ASCII integer % n
- If index is free, insert the value there
- If index is NOT free (collision):
        - then move to the next index →     index = (index + 1) % n
        - repeat until an empty slot is found
- Insert the value in the first available slot
```

Linear probing is a great strategy because is simple to implement, efficient in terms of memory management and it performs well in small form factors.

Table of the Data used under the following headings:

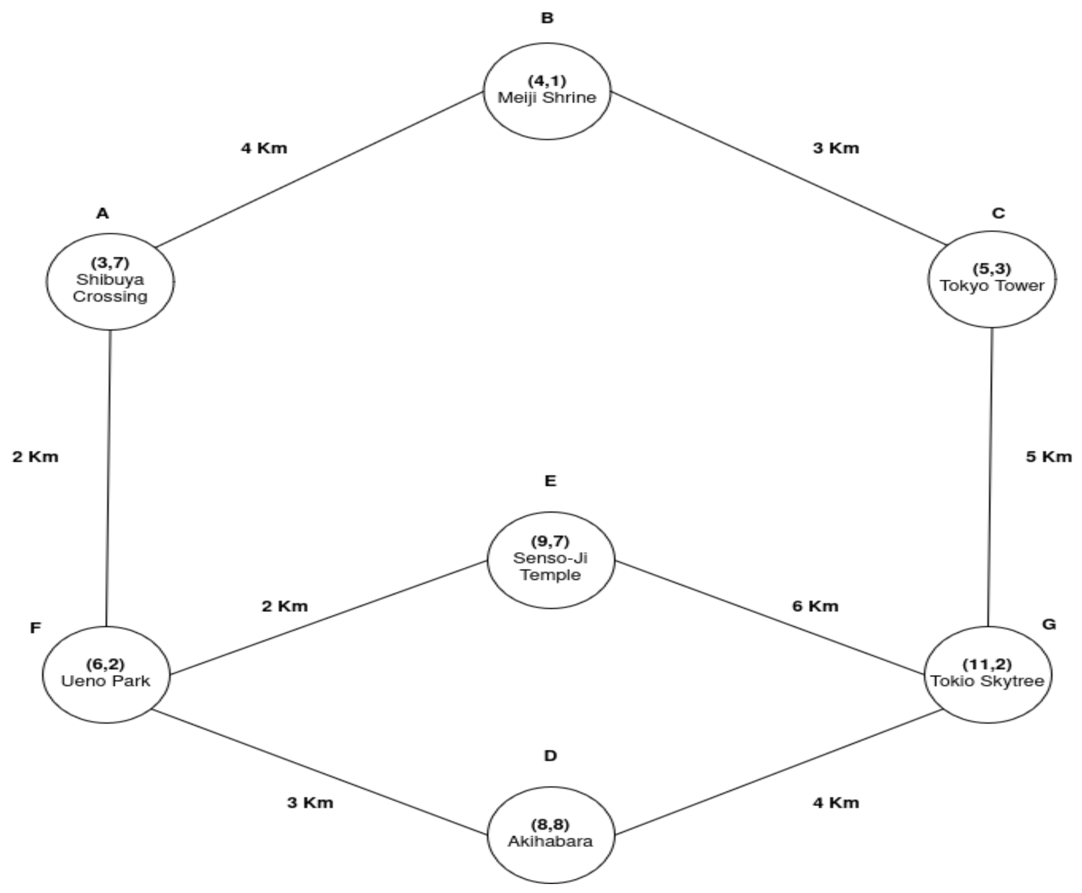
Username	**ASCII Integer**	**Index after Hash function applied**
Javier	74 + 97 + 118 + 105 + 101 + 114 = 609	9
Elysia	69 + 108 + 121 + 115 + 105 + 97 = 615	15
Konrad	75 + 111 + 110 + 114 + 97 + 100 = 607	7
Diego	68 + 105 + 101 + 103 + 111 = 488	8
Hugo	72 + 117 + 103 + 111 = 403	3
Aaron	65 + 97 + 114 + 111 + 110 = 497	17
Sasha	83 + 97 + 115 + 104 + 97 = 496	16
Rachel	82 + 97 + 99 + 104 + 101 + 108 = 591	11
Luna	76 + 117 + 110 + 97 = 400	0
Stella	83 + 116 + 101 + 108 + 108 + 97 = 613	13
Mario	77 + 97 + 114 + 105 + 111 = 504	4
Luigi	76 + 117 + 105 + 103 + 105 = 506	6
Mallow	77 + 97 + 108 + 108 + 111 + 119 = 620	0 (collision)
Jerry	74 + 101 + 114 + 114 + 121 = 524	4 (collision)
Alan	65 + 108 + 97 + 110 = 380	0 (collision)
Darren	68 + 97 + 114 + 114 + 101 + 110 = 604	4 (collision)
Eric	69 + 114 + 105 + 99 = 387	7 (collision)
Ryan	82 + 121 + 97 + 110 = 410	10
Ciara	67 + 105 + 97 + 114 + 97 = 480	0 (collision)
Nathan	78 + 97 + 116 + 104 + 97 + 110 = 602	2

Diagram of the Hash Table produced- with collisions highlighted.



Index	Name	Collision
0	Luna	Mallow Alan Ciara
1	Mallow	
2	Alan	
3	Ciara	
4	Nathan	Mario Jerry Darren
5	Hugo	
6	Mario	Luigi
7	Luigi	Konrad Eric
8	Konrad	
9	Diego	
10	Javier	
11	Eric	
12	Ryan	
13	Jerry	
14	Rachel	
15	Setella	
16	Aaron	
17	Sasha	
18	Elysia	
19	Darren	

Diagram of Map of Tourist Sites



Edge List and Distances

Edge	Distance
A – B	4 KM
B – C	3 KM
A – F	2 KM
F – E	2 KM
E – G	6 KM
C – G	5 KM
F – D	3 KM
D – G	4 KM

Sites and Coordinates

Code	Site	Coordinates
A	Shibuya Crossing	(3, 7)
B	Meiji Shrine	(4, 1)
C	Tokyo Tower	(5, 3)
D	Akihabara	(8, 8)
E	Senso-Ji Temple	(9, 7)
F	Ueno Park	(6, 2)
G	Tokyo Skytree	(11, 2)

Description of Data Structure used to store maps and help variables.

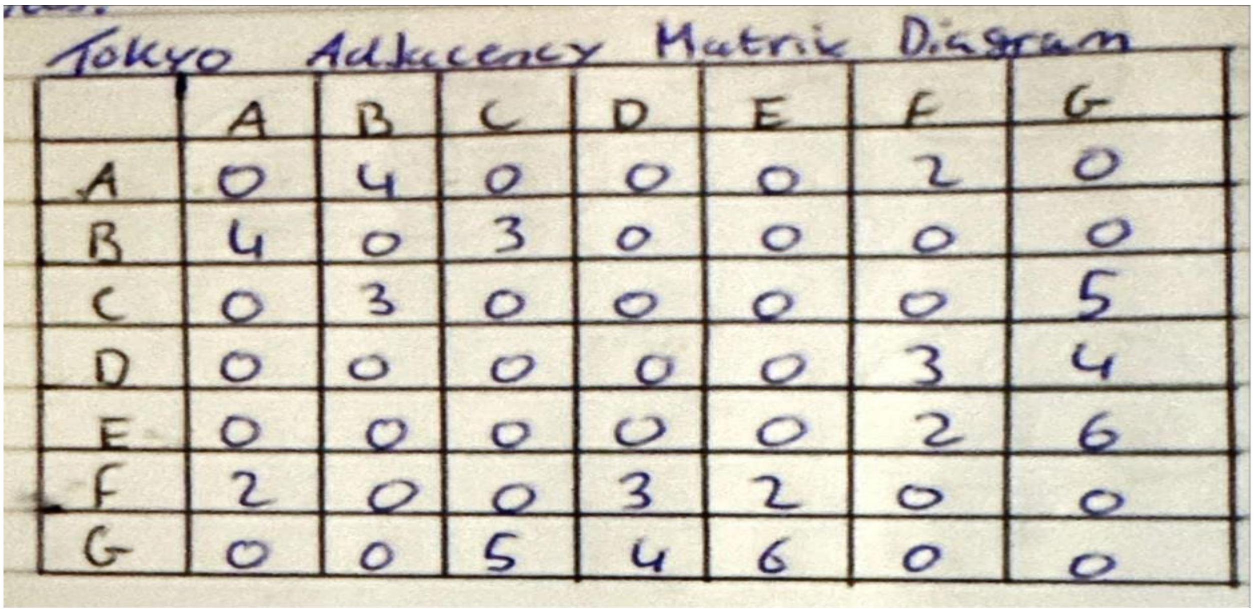
The Tokyo map is stored in parallel arrays and an adjacency matrix.

Each site is given an index (0-6) and its code (A, B, C etc). Name and coordinates are stored in the same siteCode[i], siteName [i], x[i], y[i].

Connections between sites are held in a 2D array dist[i][j], where each value is the distance between site I and j, A value of 0 means there's no link between sites. Since the map is undirected, the values appear in both directions.

Searching is done by scanning the site list for a matching name or code, then using the index to find that row of the matrix and list its neighboring sites, a loop of variables is used for scanning and minDistance and closesIndex are used to find the nearest neighbor.

Diagram of map stores in our chosen data structure

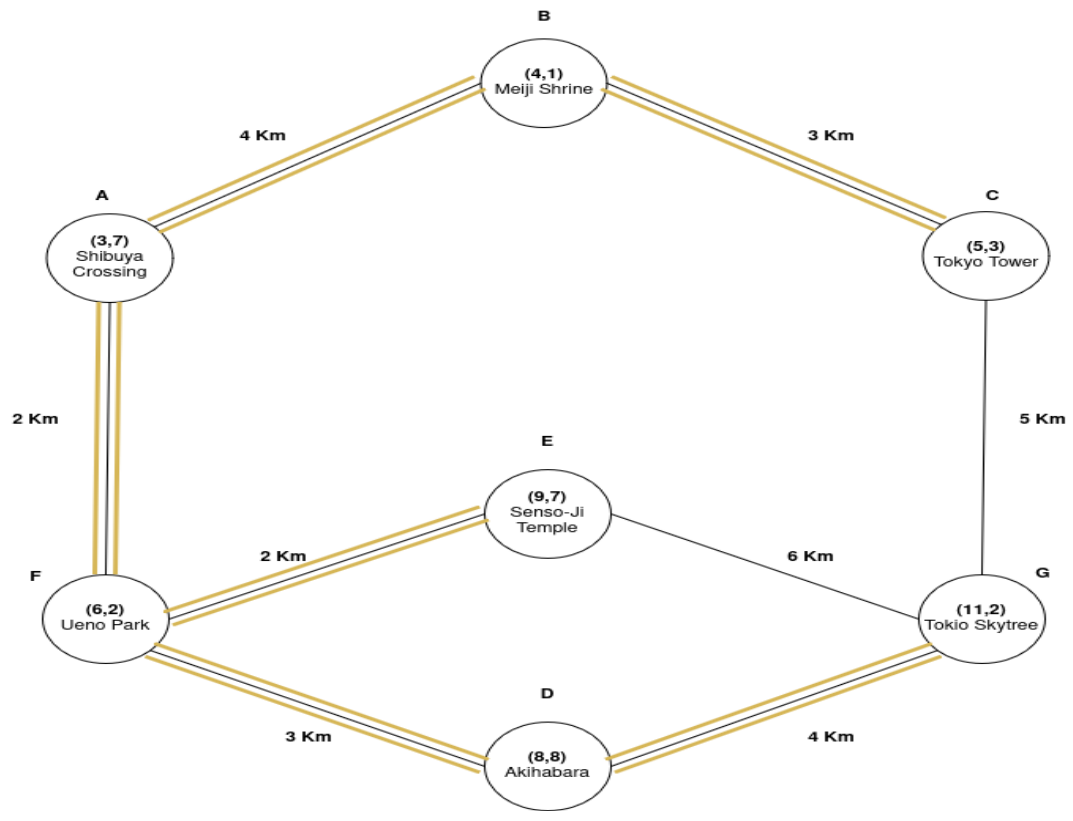


Tokyo Adjacency Matrix Diagram

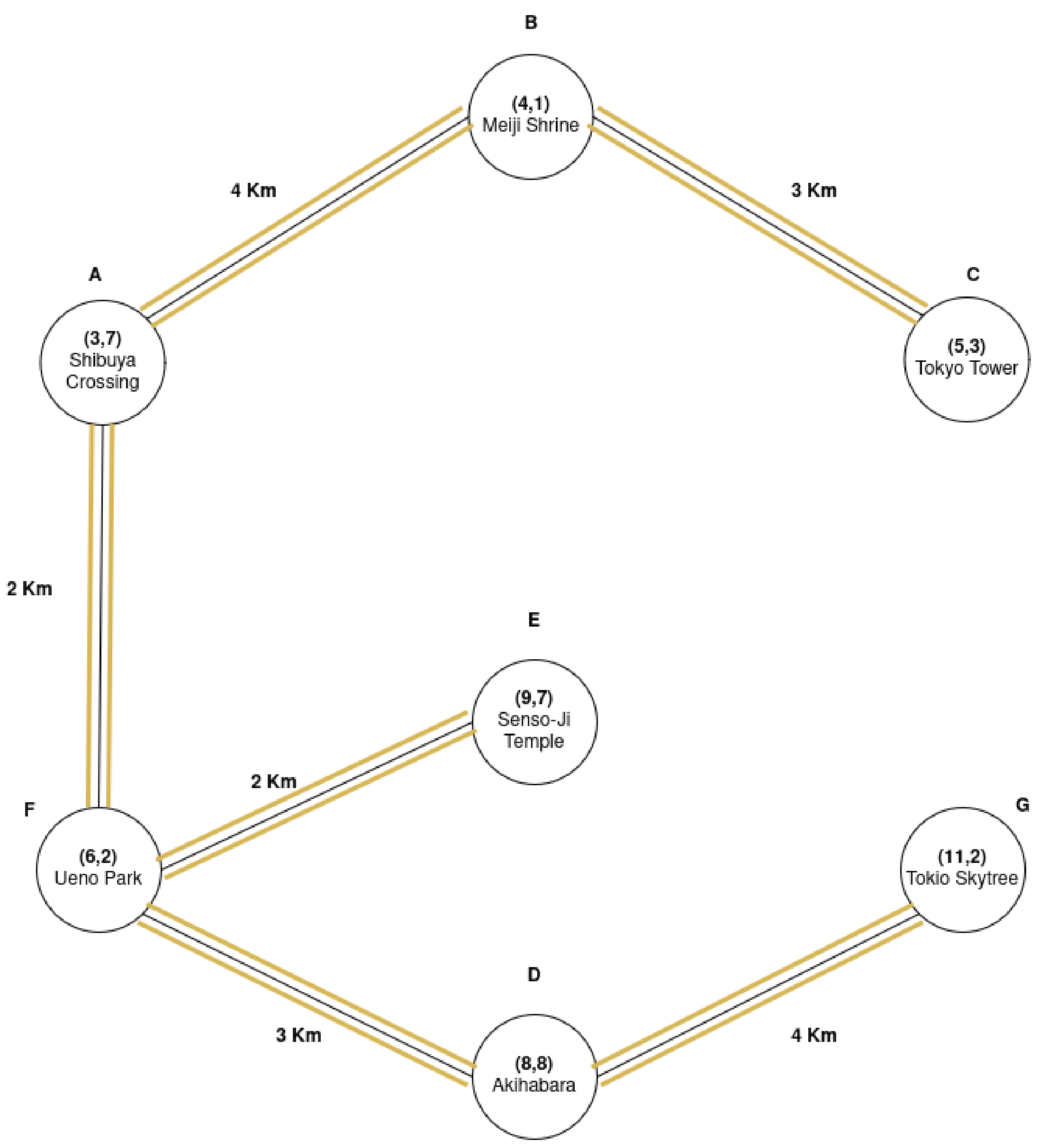
	A	B	C	D	E	F	G
A	0	4	0	0	0	2	0
B	4	0	3	0	0	0	0
C	0	3	0	0	0	0	5
D	0	0	0	0	0	3	4
E	0	0	0	0	0	2	6
F	2	0	0	3	2	0	0
G	0	0	5	4	6	0	0

Minimum Spanning Tree.

Using Kruskal's algorithm to calculate the minimum spanning tree (MST), sort all edges in order to increase weight, then pick edges in that order and make sure we wouldn't create a cycle.



Code	Distances
A - F	2KM
F - E	2KM
B - C	3KM
F - D	3KM
A - B	4KM
D - G	4KM



Pseudocode for Algorithms

I) Search: searches for a tourist site by its code or name and returns its index in the array.

```
int Search(String searchIndex) {
    for (int i = 0; i < 7; i++) {
        if (siteCode[i].equals(searchIndex)
|| siteName[i].equals(searchIndex)) {
     print("found " + siteCode[i]  + "    " +  siteName[i]);
     print("coordinates"(" + x[i] + " , " + y[i]  +  ")");
            return i
        }
    }
    print(" site not found")
    return -1
}
```

II)Insert: adds a connection between two sites with a specified distance in kilometres.

```
void Insert(int i, int j, int distance) {
    if (i >= 0 && i < 7 && j >= 0 && j < 7) {
       dist[i][j] = distance ;
       dist[j][i] = distance ;
       print(" connections added"  +  distance  +  "KM");
    }
}
```

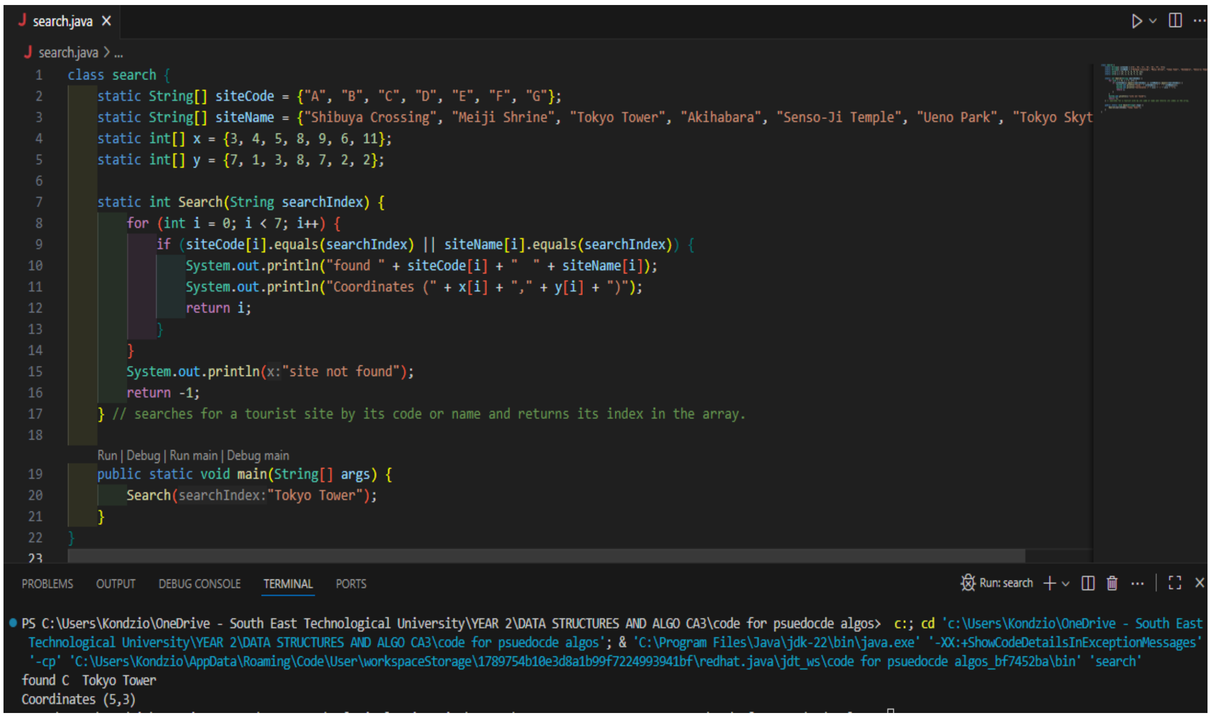
III)AllCons: displays all direct connections and distances from a specified site.

```
void AllCons(int i) {
    if (i >= 0 && i < 7) {
        print(" connections from " + siteName[i]);
        for (int j = 0; j < 7; j++){
            if (dist[i][j] > 0) {
                print(siteName[i] + "    " + siteName[j] +
" " + dist[i][j] + " KM ")
            }
        }
    }
 }
```

IV)Closest: finds and returns the nearest connected site to a specified site based on minimum distance.

```
int Closest(int i) {
    if (i >= 0 && i < 7) {
        int minDistance =  999
        int closestIndex =  -1
        for (int j = 0 ;  j < 7 ;  j++) {
           if (dist[i][j] > 0 && dist[i][j]< minDistance){
                minDistance = dist[i][j];
                closestIndex = j;
            }
            }
        if (closestIndex != -1) {
            print("closest is " + siteName[closestIndex] + "
            at the distance of " + minDistance +
            "KM")          }
    return closestIndex      }
      return -1
 }
```

I. Search Method

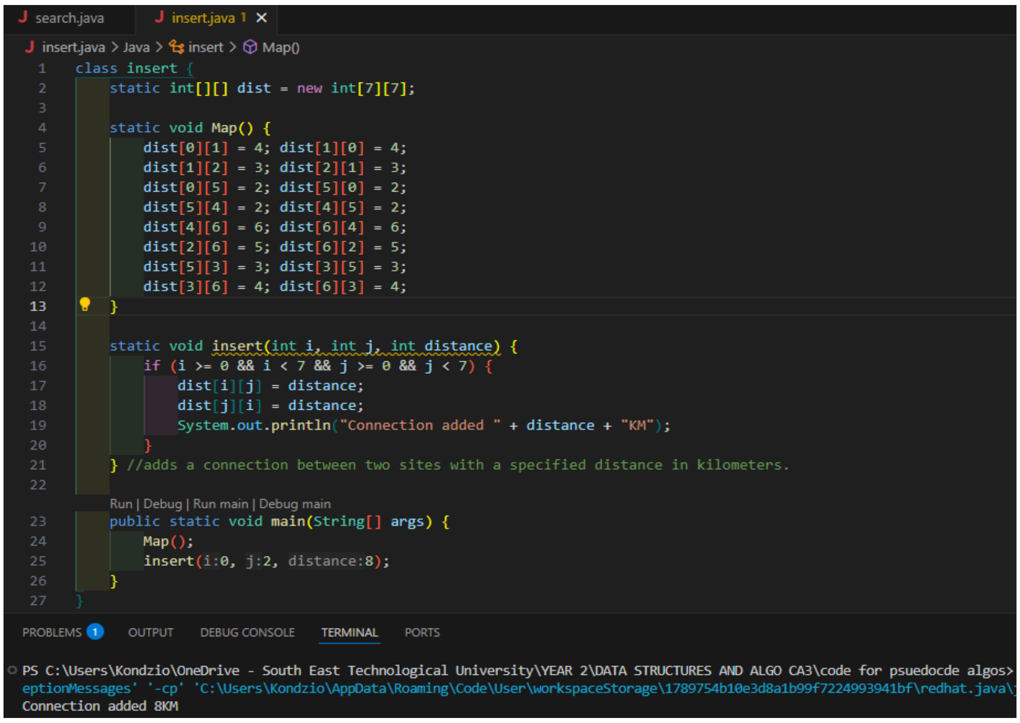


search.java

```java
class search {
    static String[] siteCode = {"A", "B", "C", "D", "E", "F", "G"};
    static String[] siteName = {"Shibuya Crossing", "Meiji Shrine", "Tokyo Tower", "Akihabara", "Senso-Ji Temple", "Ueno Park", "Tokyo Skyt
    static int[] x = {3, 4, 5, 8, 9, 6, 11};
    static int[] y = {7, 1, 3, 8, 7, 2, 2};

    static int Search(String searchIndex) {
        for (int i = 0; i < 7; i++) {
            if (siteCode[i].equals(searchIndex) || siteName[i].equals(searchIndex)) {
                System.out.println("found " + siteCode[i] + "  " + siteName[i]);
                System.out.println("Coordinates (" + x[i] + "," + y[i] + ")");
                return i;
            }
        }
        System.out.println(x:"site not found");
        return -1;
    } // searches for a tourist site by its code or name and returns its index in the array.

    public static void main(String[] args) {
        Search(searchIndex:"Tokyo Tower");
    }
}
```

```
PS C:\Users\Kondzio\OneDrive - South East Technological University\YEAR 2\DATA STRUCTURES AND ALGO CA3\code for psuedocde algos>  c:; cd 'c:\Users\Kondzio\OneDrive - South East
 Technological University\YEAR 2\DATA STRUCTURES AND ALGO CA3\code for psuedocde algos'; & 'C:\Program Files\Java\jdk-22\bin\java.exe' '-XX:+ShowCodeDetailsInExceptionMessages'
 '-cp' 'C:\Users\Kondzio\AppData\Roaming\Code\User\workspaceStorage\1789754b10e3d8a1b99f7224993941bf\redhat.java\jdt_ws\code for psuedocde algos_bf7452ba\bin' 'search'
found C  Tokyo Tower
Coordinates (5,3)
```

II. Insert Method



search.java   insert.java 1

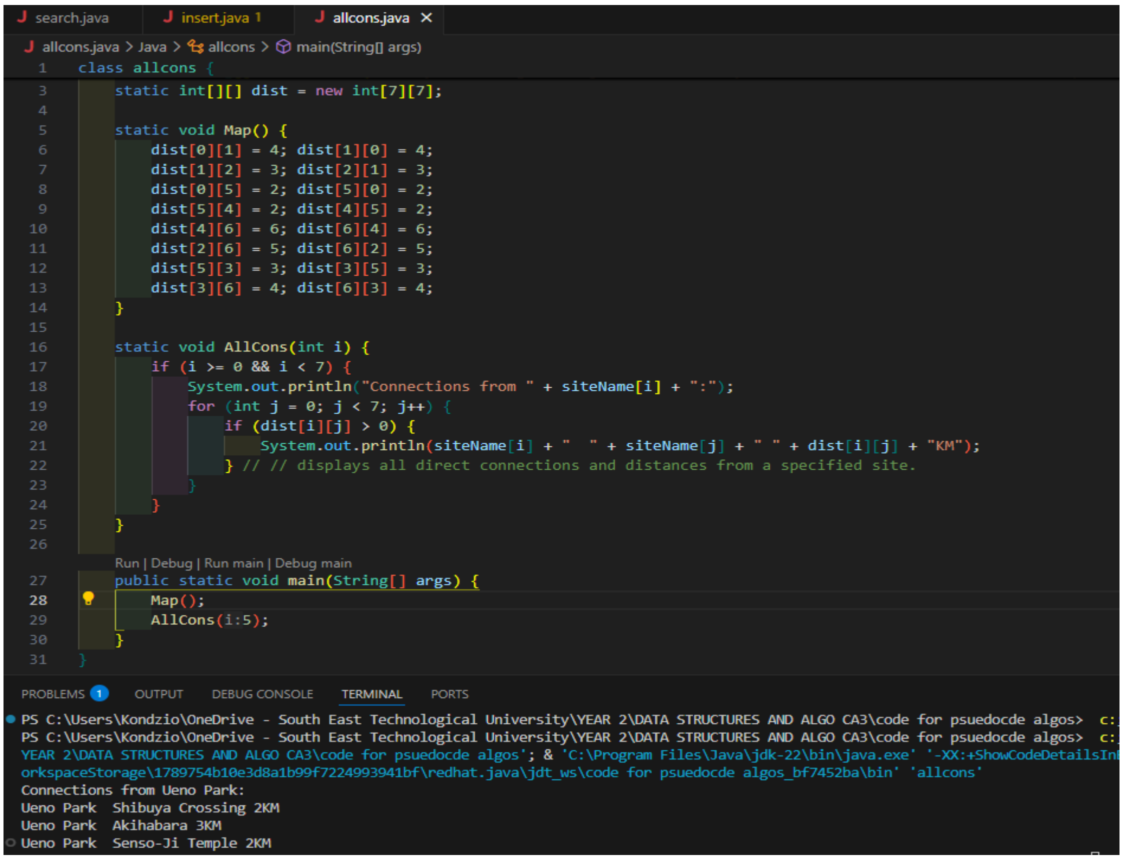
insert.java > Java > insert > Map()

```java
class insert {
    static int[][] dist = new int[7][7];

    static void Map() {
        dist[0][1] = 4; dist[1][0] = 4;
        dist[1][2] = 3; dist[2][1] = 3;
        dist[0][5] = 2; dist[5][0] = 2;
        dist[5][4] = 2; dist[4][5] = 2;
        dist[4][6] = 6; dist[6][4] = 6;
        dist[2][6] = 5; dist[6][2] = 5;
        dist[5][3] = 3; dist[3][5] = 3;
        dist[3][6] = 4; dist[6][3] = 4;
    }

    static void insert(int i, int j, int distance) {
        if (i >= 0 && i < 7 && j >= 0 && j < 7) {
            dist[i][j] = distance;
            dist[j][i] = distance;
            System.out.println("Connection added " + distance + "KM");
        }
    } //adds a connection between two sites with a specified distance in kilometers.

    Run | Debug | Run main | Debug main
    public static void main(String[] args) {
        Map();
        insert(i:0, j:2, distance:8);
    }
}
```

PROBLEMS 1   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS

```
PS C:\Users\Kondzio\OneDrive - South East Technological University\YEAR 2\DATA STRUCTURES AND ALGO CA3\code for psuedocde algos>
eptionMessages' '-cp' 'C:\Users\Kondzio\AppData\Roaming\Code\User\workspaceStorage\1789754b10e3d8a1b99f7224993941bf\redhat.java\
Connection added 8KM
```

III. Allcons Method



search.java   insert.java 1   allcons.java

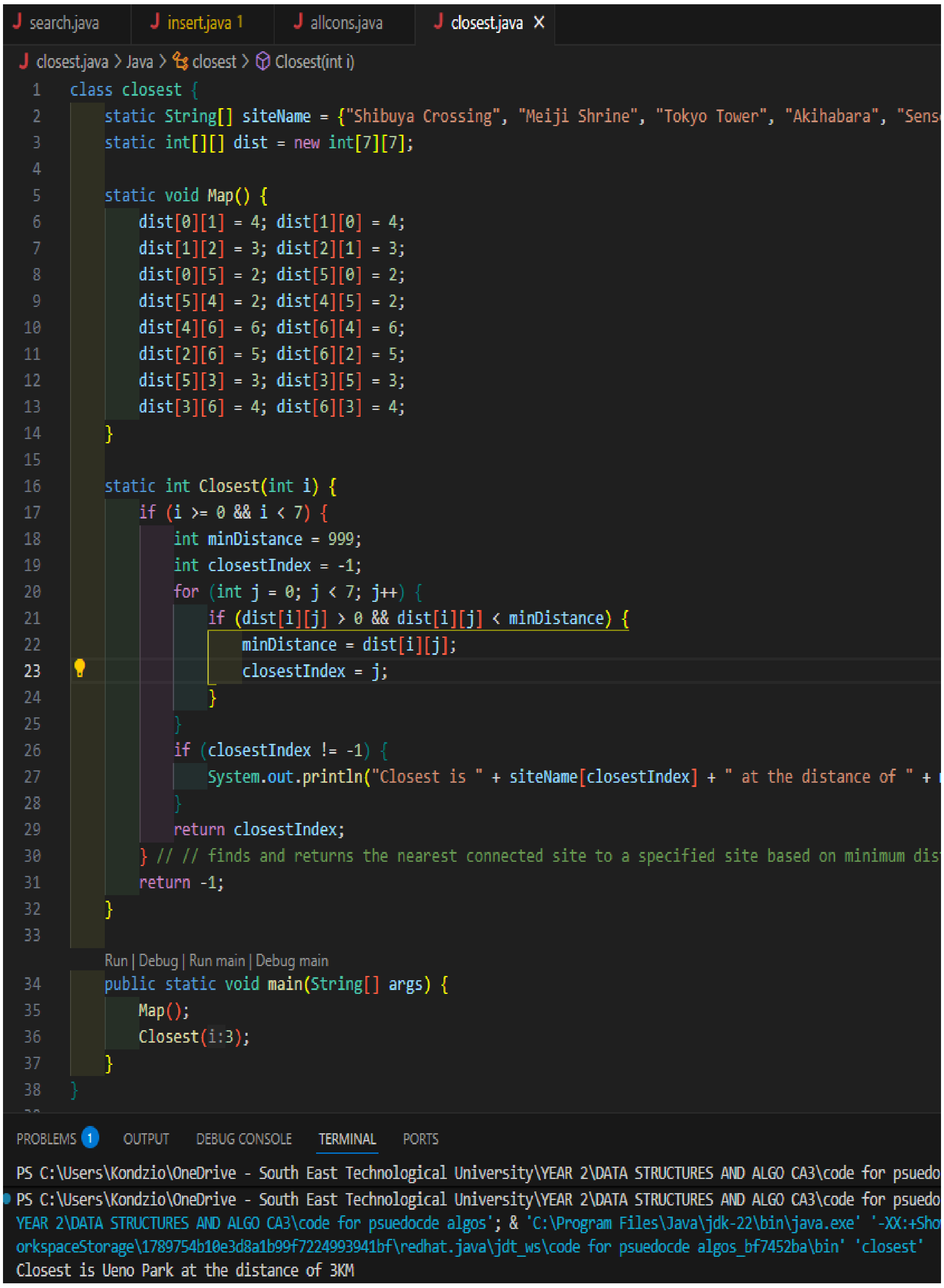
allcons.java > Java > allcons > main(String[] args)

```java
class allcons {
    static int[][] dist = new int[7][7];

    static void Map() {
        dist[0][1] = 4; dist[1][0] = 4;
        dist[1][2] = 3; dist[2][1] = 3;
        dist[0][5] = 2; dist[5][0] = 2;
        dist[5][4] = 2; dist[4][5] = 2;
        dist[4][6] = 6; dist[6][4] = 6;
        dist[2][6] = 5; dist[6][2] = 5;
        dist[5][3] = 3; dist[3][5] = 3;
        dist[3][6] = 4; dist[6][3] = 4;
    }

    static void AllCons(int i) {
        if (i >= 0 && i < 7) {
            System.out.println("Connections from " + siteName[i] + ":");
            for (int j = 0; j < 7; j++) {
                if (dist[i][j] > 0) {
                    System.out.println(siteName[i] + "  " + siteName[j] + " " + dist[i][j] + "KM");
                } // // displays all direct connections and distances from a specified site.
            }
        }
    }

    Run | Debug | Run main | Debug main
    public static void main(String[] args) {
        Map();
        AllCons(i:5);
    }
}
```

PROBLEMS 1   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS

```
PS C:\Users\Kondzio\OneDrive - South East Technological University\YEAR 2\DATA STRUCTURES AND ALGO CA3\code for psuedocde algos>  c:
PS C:\Users\Kondzio\OneDrive - South East Technological University\YEAR 2\DATA STRUCTURES AND ALGO CA3\code for psuedocde algos>  c:
YEAR 2\DATA STRUCTURES AND ALGO CA3\code for psuedocde algos'; & 'C:\Program Files\Java\jdk-22\bin\java.exe' '-XX:+ShowCodeDetailsIn
orkspaceStorage\1789754b10e3d8a1b99f7224993941bf\redhat.java\jdt_ws\code for psuedocde algos_bf7452ba\bin' 'allcons'
Connections from Ueno Park:
Ueno Park  Shibuya Crossing 2KM
Ueno Park  Akihabara 3KM
Ueno Park  Senso-Ji Temple 2KM
```

IV. Closest Method



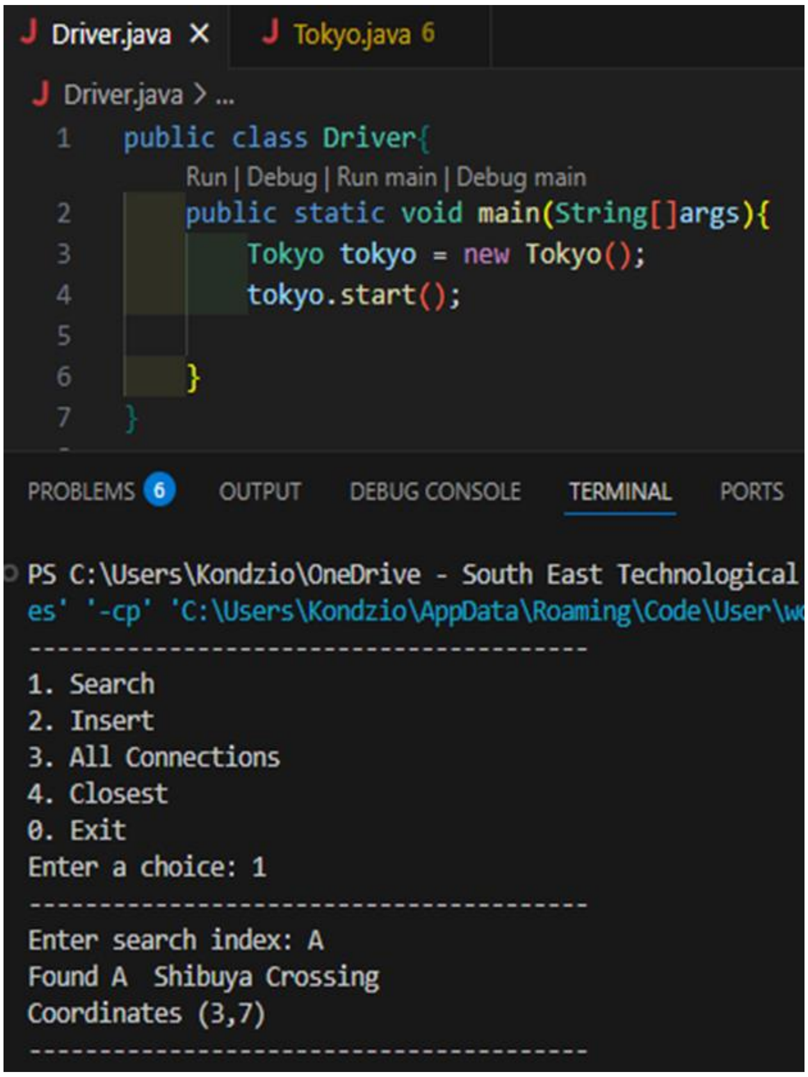
```java
class closest {
    static String[] siteName = {"Shibuya Crossing", "Meiji Shrine", "Tokyo Tower", "Akihabara", "Sens
    static int[][] dist = new int[7][7];

    static void Map() {
        dist[0][1] = 4; dist[1][0] = 4;
        dist[1][2] = 3; dist[2][1] = 3;
        dist[0][5] = 2; dist[5][0] = 2;
        dist[5][4] = 2; dist[4][5] = 2;
        dist[4][6] = 6; dist[6][4] = 6;
        dist[2][6] = 5; dist[6][2] = 5;
        dist[5][3] = 3; dist[3][5] = 3;
        dist[3][6] = 4; dist[6][3] = 4;
    }

    static int Closest(int i) {
        if (i >= 0 && i < 7) {
            int minDistance = 999;
            int closestIndex = -1;
            for (int j = 0; j < 7; j++) {
                if (dist[i][j] > 0 && dist[i][j] < minDistance) {
                    minDistance = dist[i][j];
                    closestIndex = j;
                }
            }
            if (closestIndex != -1) {
                System.out.println("Closest is " + siteName[closestIndex] + " at the distance of " + 
            }
            return closestIndex;
        } // // finds and returns the nearest connected site to a specified site based on minimum dis
        return -1;
    }

    public static void main(String[] args) {
        Map();
        Closest(i:3);
    }
}
```

```
PS C:\Users\Kondzio\OneDrive - South East Technological University\YEAR 2\DATA STRUCTURES AND ALGO CA3\code for psuedo
PS C:\Users\Kondzio\OneDrive - South East Technological University\YEAR 2\DATA STRUCTURES AND ALGO CA3\code for psuedo
YEAR 2\DATA STRUCTURES AND ALGO CA3\code for psuedocde algos'; & 'C:\Program Files\Java\jdk-22\bin\java.exe' '-XX:+Sho
orkspaceStorage\1789754b10e3d8a1b99f7224993941bf\redhat.java\jdt_ws\code for psuedocde algos_bf7452ba\bin' 'closest'
Closest is Ueno Park at the distance of 3KM
```

Part 3

Sample Executions for each operation

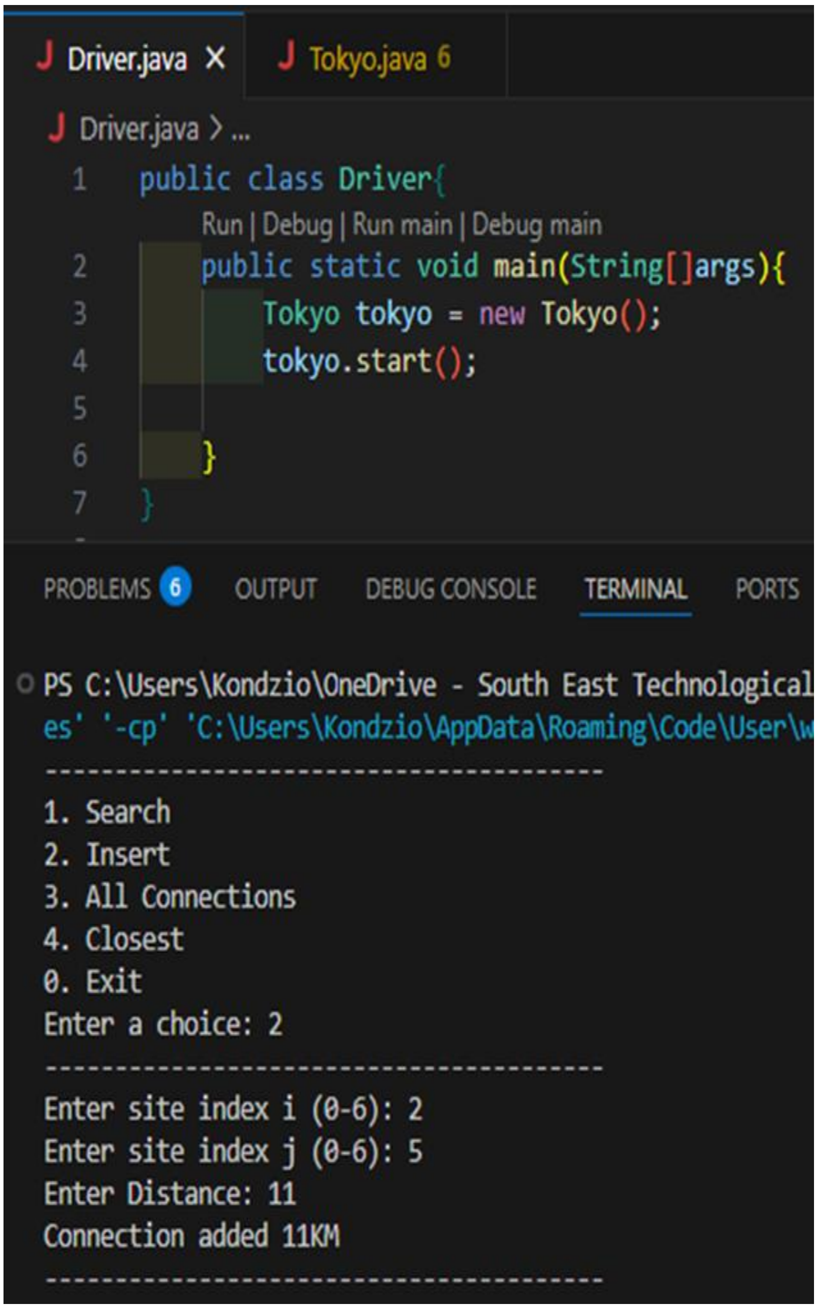
1.Search



```java
public class Driver{
    public static void main(String[]args){
        Tokyo tokyo = new Tokyo();
        tokyo.start();

    }
}
```

```
PS C:\Users\Kondzio\OneDrive - South East Technological
es' '-cp' 'C:\Users\Kondzio\AppData\Roaming\Code\User\w
---------------------------------------
1. Search
2. Insert
3. All Connections
4. Closest
0. Exit
Enter a choice: 1
---------------------------------------
Enter search index: A
Found A  Shibuya Crossing
Coordinates (3,7)
---------------------------------------
```

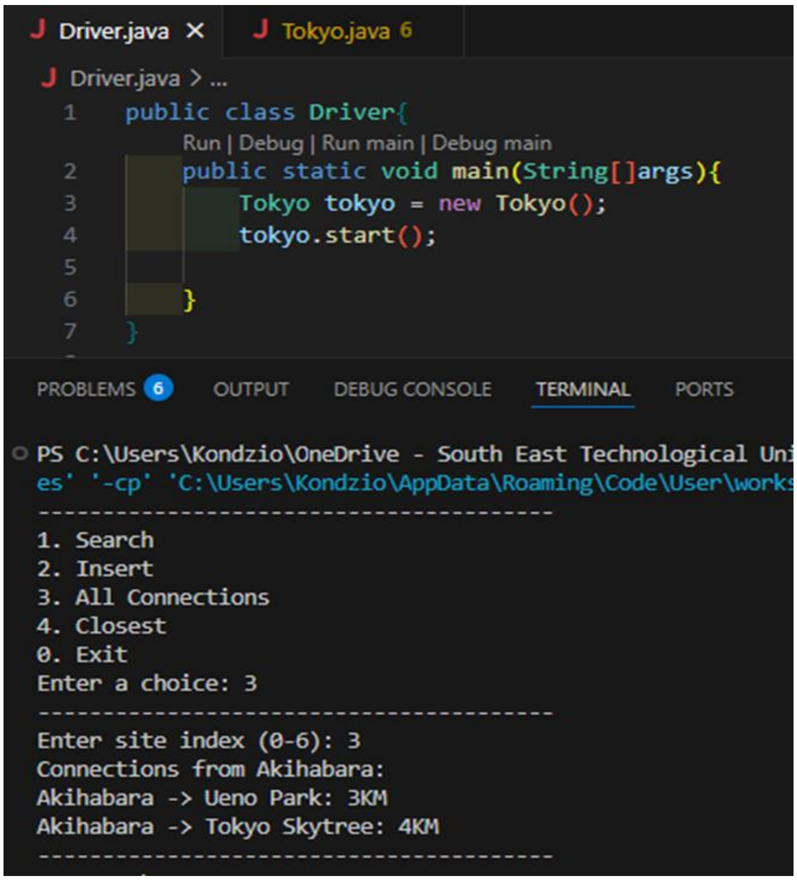
2. Insert



```java
public class Driver{
    public static void main(String[]args){
        Tokyo tokyo = new Tokyo();
        tokyo.start();

    }
}
```

```
PS C:\Users\Kondzio\OneDrive - South East Technological
es' '-cp' 'C:\Users\Kondzio\AppData\Roaming\Code\User\w
---------------------------------------
1. Search
2. Insert
3. All Connections
4. Closest
0. Exit
Enter a choice: 2
---------------------------------------
Enter site index i (0-6): 2
Enter site index j (0-6): 5
Enter Distance: 11
Connection added 11KM
---------------------------------------
```

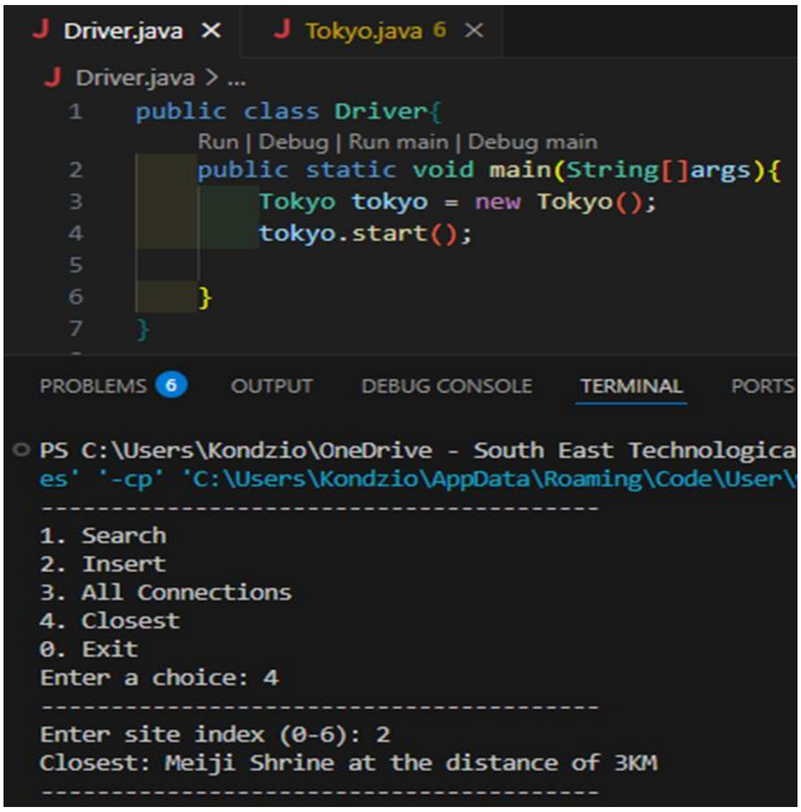
3. All Connections



```java
public class Driver{
    public static void main(String[]args){
        Tokyo tokyo = new Tokyo();
        tokyo.start();

    }
}
```

```
PS C:\Users\Kondzio\OneDrive - South East Technological Uni
es' '-cp' 'C:\Users\Kondzio\AppData\Roaming\Code\User\works
----------------------------------------
1. Search
2. Insert
3. All Connections
4. Closest
0. Exit
Enter a choice: 3
----------------------------------------
Enter site index (0-6): 3
Connections from Akihabara:
Akihabara -> Ueno Park: 3KM
Akihabara -> Tokyo Skytree: 4KM
----------------------------------------
```

4. Closest



```java
public class Driver{
    public static void main(String[]args){
        Tokyo tokyo = new Tokyo();
        tokyo.start();

    }
}
```

```
PS C:\Users\Kondzio\OneDrive - South East Technologica
es' '-cp' 'C:\Users\Kondzio\AppData\Roaming\Code\User\
----------------------------------------
1. Search
2. Insert
3. All Connections
4. Closest
0. Exit
Enter a choice: 4
----------------------------------------
Enter site index (0-6): 2
Closest: Meiji Shrine at the distance of 3KM
----------------------------------------
```

References

Byrne, Á. (2025). Chapter 5: Graphs. Carlow, Ireland.

Sambol, M. (2012). *Youtube.com*. Retrieved from Youtube.com: http://www.youtube.com/watch?v=71UQH7Pr9kU

Sierra, K. a. (2003). *Head First Java.* Sebastopol, CA: O'Reilly Media.

Wikipedia. (2024). *https://en.wikipedia.org/wiki/Hash_table*. Retrieved from https://en.wikipedia.org: https://en.wikipedia.org/wiki/Hash_table#