RMIT

RMIT International University Vietnam

Assignment Cover Page

Subject Code:	COSC2658
Subject Name:	Data Structures and Algorithms
Location:	SGS Campus
Title of Assignment:	Group Project
Teachers Name:	Dr. Tri Dang
Group Name:	Group 7
Group Members:	S3974735 – Tran Manh Cuong S3978598 – Phan Nhat Minh S3965528 – Truong Quang Bao Loc S3979638 – Tran Ha Phuong
Assignment due date:	May 9, 2024
Date of Submission:	May 9, 2024
Declaration of Authorship	We declare that in submitting all work for this assessment, we have read, understood and agreed to the content and expectations of Assessment Declaration as specified in https://www.rmit.edu.vn/students/my-studies/assessment-and-exams/assessment
Consent to Use:	We give RMIT University permission to use our work as an exemplar and for showcase/exhibition display.

Table of Contents

1. INTRODUCTION

1.1. Problem Statement

The task involves developing a robust geospatial search feature for a mapping application, analogous to functionalities found in applications like Google Maps. The core requirement is to optimize the search for Points of Interest (POI) within a specified bounding rectangle defined by its top left corner, width, and height. This rectangle is dynamically adjusted based on user inputs, such as proximity to a certain location or a specified walking distance. Additionally, the search must be filtered by a type of service, such as ATMs, restaurants, or hospitals. The system should be capable of handling up to 100,000,000 places with a maximum of 50 results displayed to the user. The algorithm must prioritize efficiency and accuracy, ensuring results are relevant and swiftly delivered, even under the constraints of not using the Java Collections Framework or external libraries.

1.2. Abstract

This technical report documents the design, implementation, and testing of a custom Map2D Abstract Data Type (ADT) to manage and query geospatial data for a mapping application. The Map2D ADT is built on a QuadTree structure, which is highly efficient for managing large datasets in a two-dimensional space and provides quick query responses. The system supports adding, editing, and removing places, each characterized by coordinates and a set of services offered. Search functionality, which is the focus of this development, allows for querying places within a user-defined bounding rectangle and filtering by service type, optimizing for proximity based on Euclidean distance metrics. This report details the methodologies employed in implementing these features, challenges encountered, solutions devised, and the performance evaluation of the system. The results demonstrate the effectiveness of the QuadTree-based approach in managing extensive geospatial data and providing rapid, accurate search capabilities within the constraints set forth by the project requirements.

2. OVERVIEW & HIGH-LEVEL DESIGN

2.1.Overview

This report indicates the development process of map application features that could handle the set of 100,000,000 places within a 10,000,000 x 10,000,000 map size. We decided to utilize Java (JDK 22) since we can re-develop its collection to access valuable data structures such as ArrayList, HashSet, Stack, etc, which will be beneficial to our analysis and development process. In addition, Junit version 5.9.2 provides us with the ability to test the correctness of functions.

2.2.High-level Design

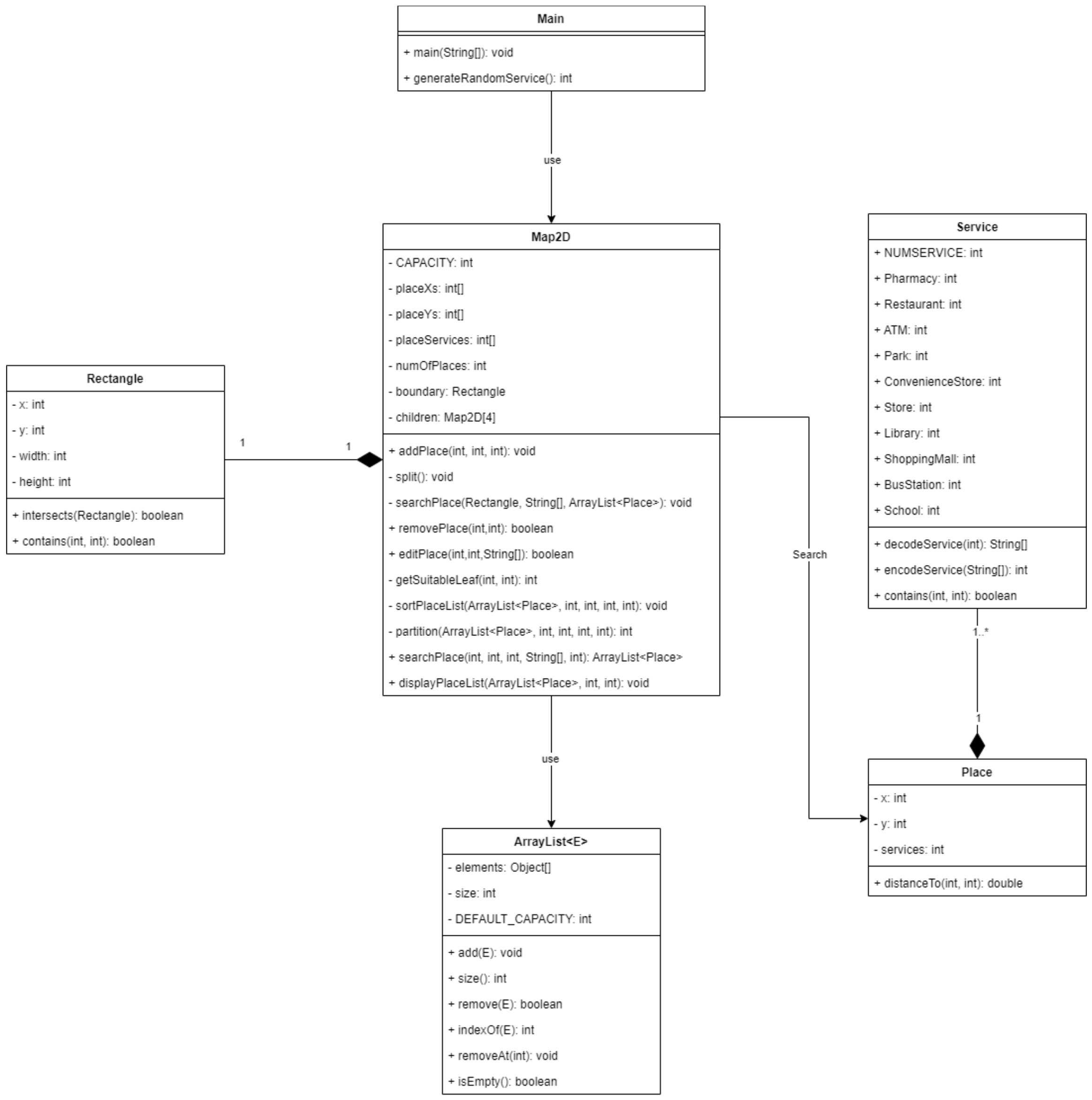

Main
+ main(String[]): void
+ generateRandomService(): int
use
Map2D
- CAPACITY: int
- placeXs: int[]
- placeYs: int[]
- placeServices: int[]
- numOfPlaces: int
- boundary: Rectangle
- children: Map2D[4]
+ addPlace(int, int, int): void
- split(): void
- searchPlace(Rectangle, String[], ArrayList<Place>): void
+ removePlace(int,int): boolean
+ editPlace(int,int,String[]): boolean
- getSuitableLeaf(int, int): int
- sortPlaceList(ArrayList<Place>, int, int, int, int): void
- partition(ArrayList<Place>, int, int, int, int): int
+ searchPlace(int, int, int, String[], int): ArrayList<Place>
+ displayPlaceList(ArrayList<Place>, int, int): void
Rectangle
- x: int
- y: int
- width: int
- height: int
+ intersects(Rectangle): boolean
+ contains(int, int): boolean
1
1
Service
+ NUMSERVICE: int
+ Pharmacy: int
+ Restaurant: int
+ ATM: int
+ Park: int
+ ConvenienceStore: int
+ Store: int
+ Library: int
+ ShoppingMall: int
+ BusStation: int
+ School: int
+ decodeService(int): String[]
+ encodeService(String[]): int
+ contains(int, int): boolean
Search
1..*
1
Place
- x: int
- y: int
- services: int
+ distanceTo(int, int): double
use
ArrayList<E>
- elements: Object[]
- size: int
- DEFAULT_CAPACITY: int
+ add(E): void
+ size(): int
+ remove(E): boolean
+ indexOf(E): int
+ removeAt(int): void
+ isEmpty(): boolean

Figure 1: Class Diagram of the Map Application

In our system, most of our development and designing test cases focus on the Map2D class which contains all four main functions including insert, delete, edit, and search, and the development process of Rectangle, Place, Service, ArrayList acts as a complementary factor for the Map2D.

- **Map2D**: The Map2D class is the main class in our system, which utilizes the QuadTree data structure to manage the map application. It allows the system to manage the space and split it into four subdivisions when it reaches the maximum capacity of places to store and search places with the provided location and desired farthest way to go. This design enables us to create the insert, delete, edit, and search methods effectively in both time complexity and space complexity.
- **Rectangle**: this class is one of the key attributes of the QuadTree design. It takes responsibility to indicate the size of a quadtree on the map through its attributes including x, y, width, and height. With this class, the QuadTree node can determine whether a place is an intersection with its boundary or not.
- **Place**: This object stands for the details of points of interest and can be considered as a central of the map application system although it is not directly attribute of the map. It is represented through three arrays of x, y, and services and the purpose of this is to increase the space complexity and this will be analyzed in-depth below.
- **Service**: this class contains a list of predefined services with associate numbers which allows the class to have the ability to encode and decode the services for each place. This idea utilizes bitmask manipulation to store multiple services within an integer size, which allows quick searching services and increases the productivity of the system.

Each of these classes is designed to contribute to the success of the four main functions of the map application and from the Map2D class we can access these functions.

- AddPlace(int x, int y, int services): To execute this function successfully, the getSuitableLeaf(int x, int y) and split() private methods are called based on specific scenarios, and the function calls itself recursively to find the appropriate node to add a new place.
- EditPlace(int x, int y, String [] services): This function also applies the same strategy as the function to add a place that recursively calls itself to find the appropriate place. Next, the system utilizes the use of encoding and decoding functions from the Service class to edit the place's services.
- RemovePlace(int x, int y): This function recursively calls itself to find the index of the place that needs to be deleted and then access the array of x, y, and services to delete the associate node at the deleted index.
- SearchPlace(int userX, int userY, int walkDistance, String [] services, int k): Create a bounding rectangle based on input and call the searchPlace() private method to populate the results that match the rectangle and utilize quick sort to have an in-order list of results.

3. DATA STRUCTURES & ALGORITHMS

3.1.Data Structures

Our team had researched and experimented with several different data structures including Kd-Tree, Graph, Hash Table, and 2D-Array, but we decided to utilize the QuadTree data structure to implement this assessment as it provides the best performance in running time and acceptable space consumption.

According to Geeks for Geeks, Quad Tree can efficiently store data of points in a two-dimensional space [1]. Each node can have at most 4 child nodes, and each child should store a fixed number of places. Whenever a child node is overloaded, it will split itself into another 4 child nodes, and divide the places data into those nodes. By utilizing the QuadTree data structure, we can divide the map into smaller areas, which significantly improves the speed and accuracy of the Point of Interest (POI) Search.

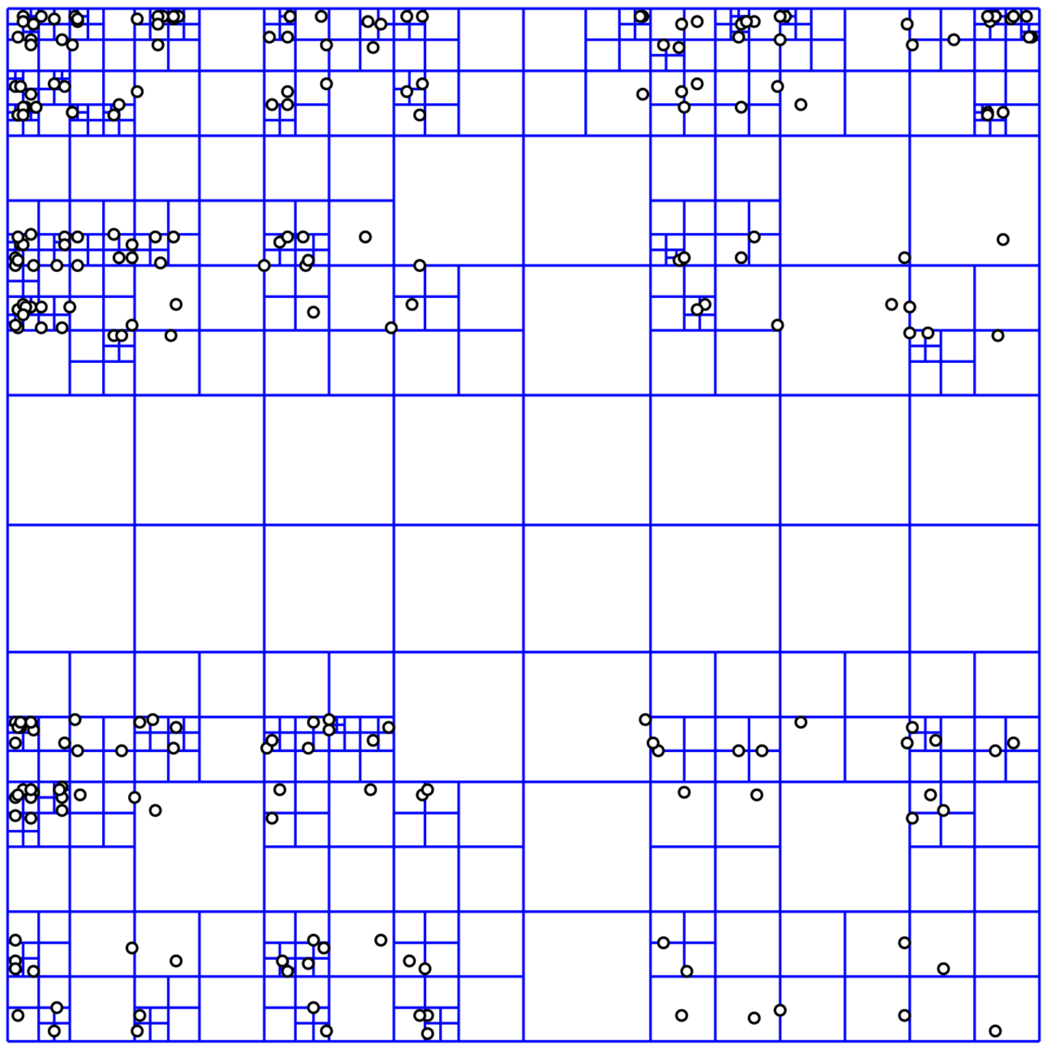
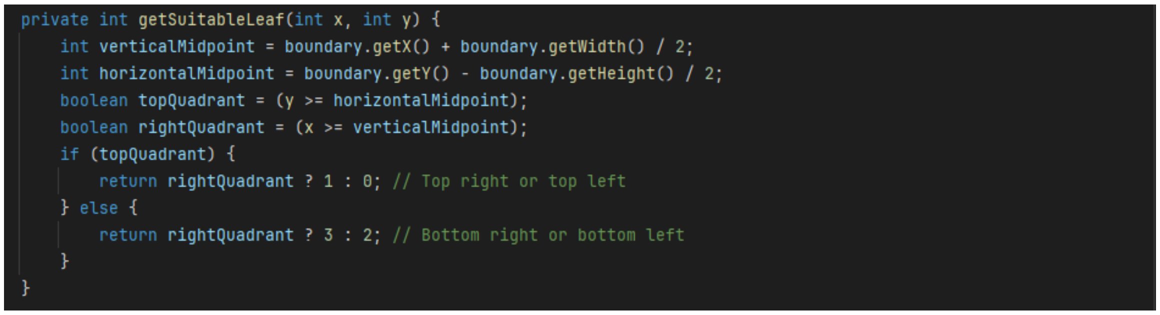


Figure 2: Visualization of the map application using QuadTree data structure [2]

There are 2 private methods in the Map2D class that our system uses a lot to manage and control the QuadTree data structure, which are getSuitableLeaf() and split().

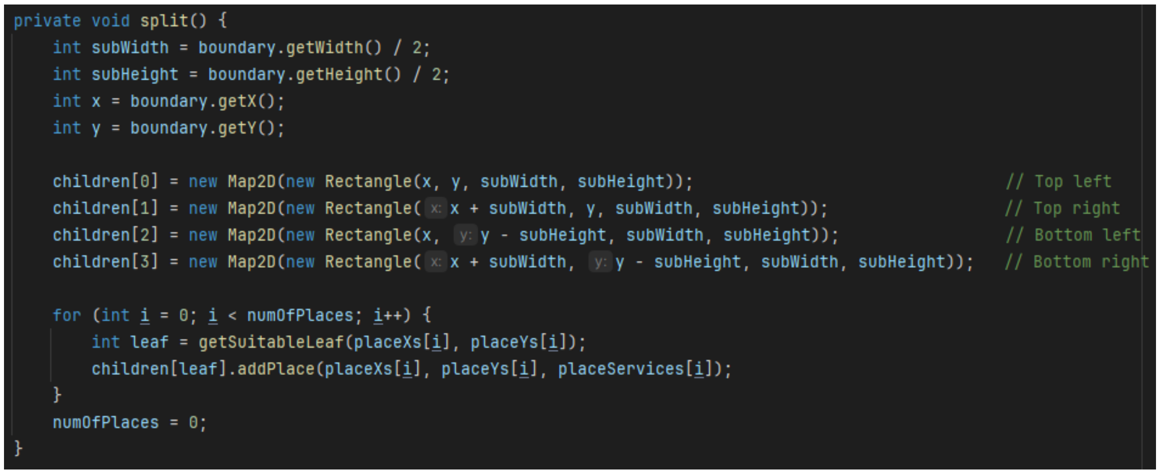
The getSuitableLeaf() method helps the system find the suitable child node of the current node based on the place's coordinates (x, y). By calculating the vertical and horizontal midpoints of the current node's boundary rectangle, this method can determine which quadrant of the boundary that the place's coordinates fall into and return the index of the suitable leaf.



```java
private int getSuitableLeaf(int x, int y) {
    int verticalMidpoint = boundary.getX() + boundary.getWidth() / 2;
    int horizontalMidpoint = boundary.getY() - boundary.getHeight() / 2;
    boolean topQuadrant = (y >= horizontalMidpoint);
    boolean rightQuadrant = (x >= verticalMidpoint);
    if (topQuadrant) {
        return rightQuadrant ? 1 : 0; // Top right or top left
    } else {
        return rightQuadrant ? 3 : 2; // Bottom right or bottom left
    }
}
```

Figure 3: getSuitableLeaf() method in the Map2D class

Whenever the current node's capacity is overloaded, the system will call the split() method to create 4 child nodes, which divides a big area into smaller areas. First, this method will calculate the coordinates (x, y) of the top left corner, width, and height of the child nodes, and store those nodes inside the children array of the current node. After creating the child nodes successfully, the system will divide all the places from the current node into its child nodes and finally reset the numOfPlaces attribute to zero.



```java
private void split() {
    int subWidth = boundary.getWidth() / 2;
    int subHeight = boundary.getHeight() / 2;
    int x = boundary.getX();
    int y = boundary.getY();

    children[0] = new Map2D(new Rectangle(x, y, subWidth, subHeight));                         // Top left
    children[1] = new Map2D(new Rectangle( x: x + subWidth, y, subWidth, subHeight));           // Top right
    children[2] = new Map2D(new Rectangle(x,  y: y - subHeight, subWidth, subHeight));          // Bottom left
    children[3] = new Map2D(new Rectangle( x: x + subWidth,  y: y - subHeight, subWidth, subHeight));   // Bottom right

    for (int i = 0; i < numOfPlaces; i++) {
        int leaf = getSuitableLeaf(placeXs[i], placeYs[i]);
        children[leaf].addPlace(placeXs[i], placeYs[i], placeServices[i]);
    }
    numOfPlaces = 0;
}
```

Figure 4: split() method in the Map2D class

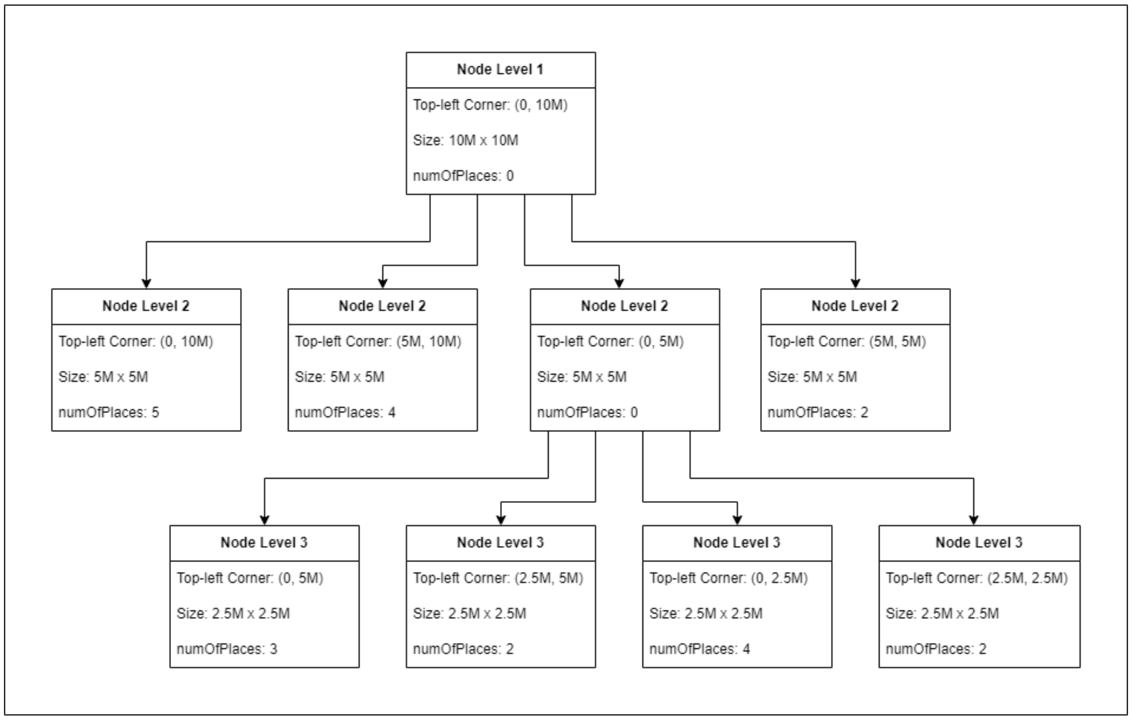


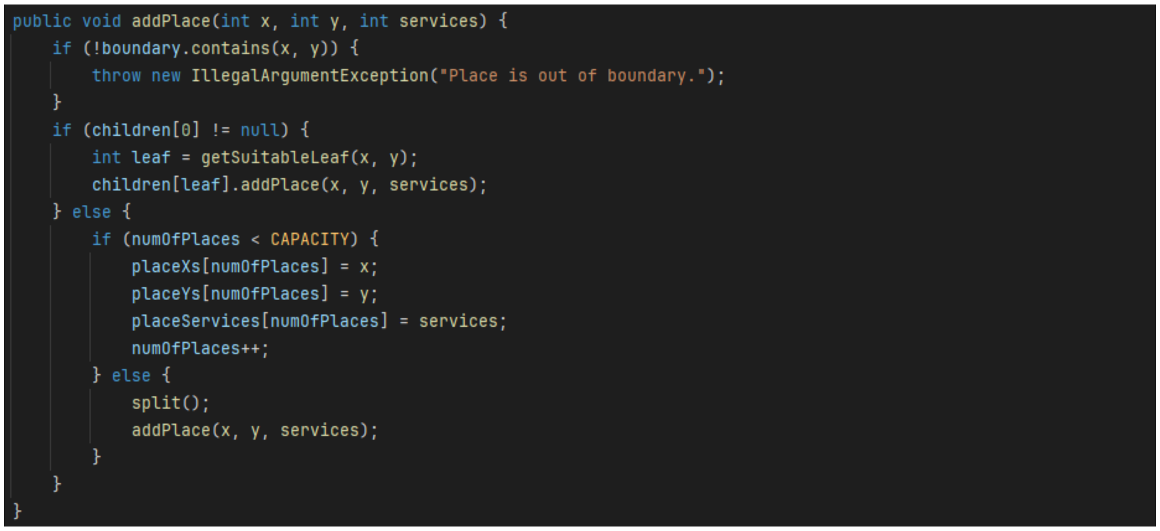
Figure 5: How the split() method split a large area into smaller areas

3.2. Algorithms

3.2.1. Add Place

To add a new place to the map, the system will have to work with the addPlace() method in the Map2D class, which requires 3 parameters:

- int x: The x-coordinate of the place.
- int y: The y-coordinate of the place.
- int services: The services available at the place.

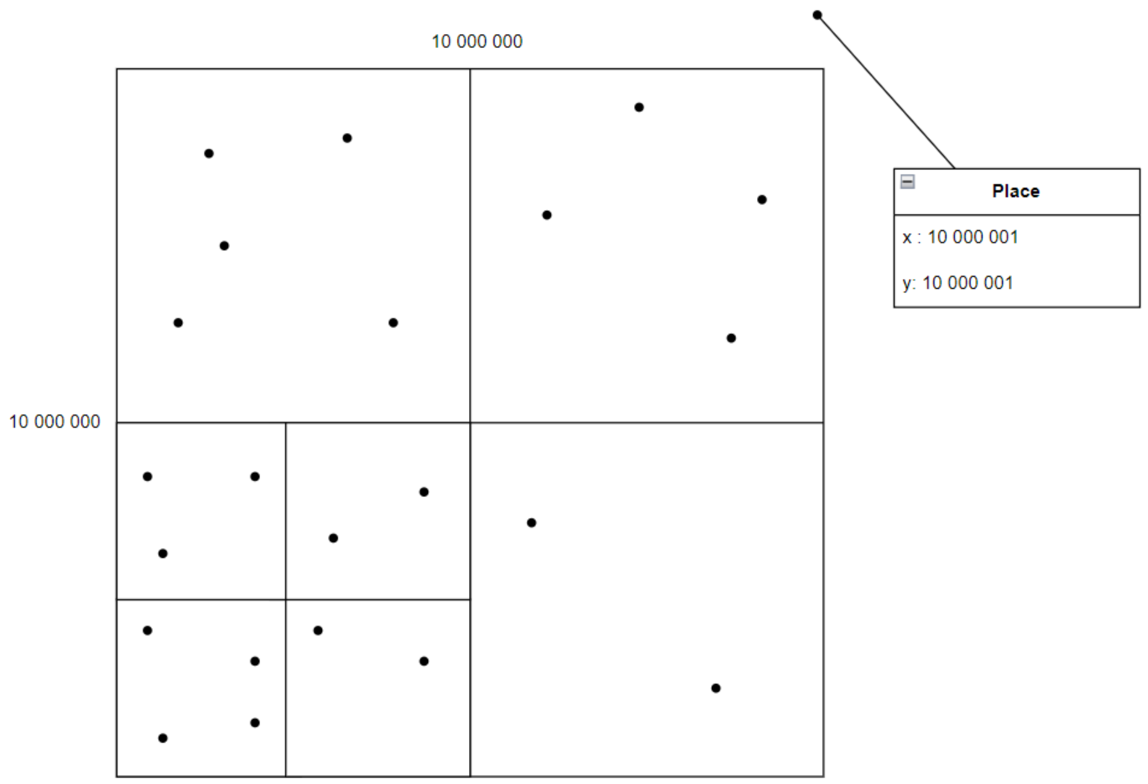


```java
public void addPlace(int x, int y, int services) {
    if (!boundary.contains(x, y)) {
        throw new IllegalArgumentException("Place is out of boundary.");
    }
    if (children[0] != null) {
        int leaf = getSuitableLeaf(x, y);
        children[leaf].addPlace(x, y, services);
    } else {
        if (numOfPlaces < CAPACITY) {
            placeXs[numOfPlaces] = x;
            placeYs[numOfPlaces] = y;
            placeServices[numOfPlaces] = services;
            numOfPlaces++;
        } else {
            split();
            addPlace(x, y, services);
        }
    }
}
```

Figure 6: addPlace() method in the Map2D class

Assuming all the input parameters are valid, the algorithm for adding a new place to the map will be processed with the following steps:

Step 1: Check whether the given coordinates (x, y) fall within the boundary of the area. If the coordinates are outside the boundary, the system will throw an IllegalArgumentException to indicate that the input place is out of the area boundary.



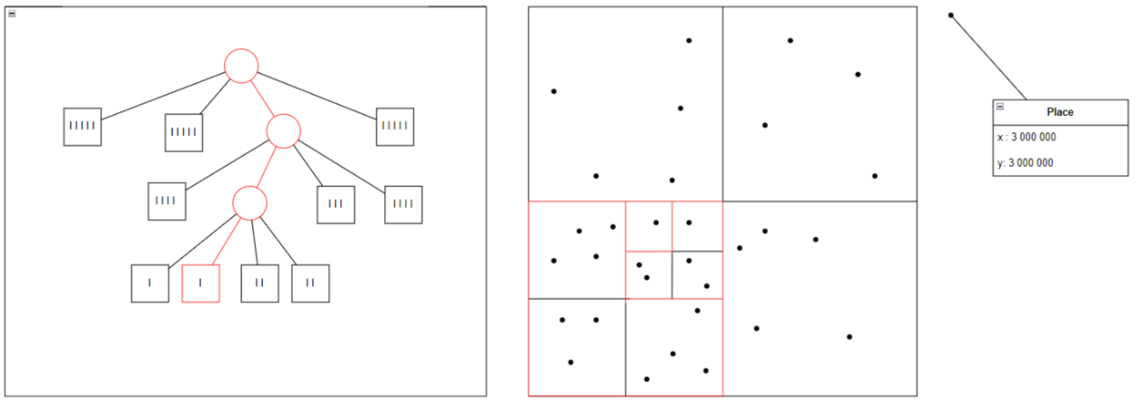
Add a place with x : 10.000.001 and y: 10.000.001

Throw a IllegalArgumentException because the place is out of map.

Step 2: Check whether the current node is the leaf node and check for the capacity of the leaf node. There will be 3 circumstances that might occur:

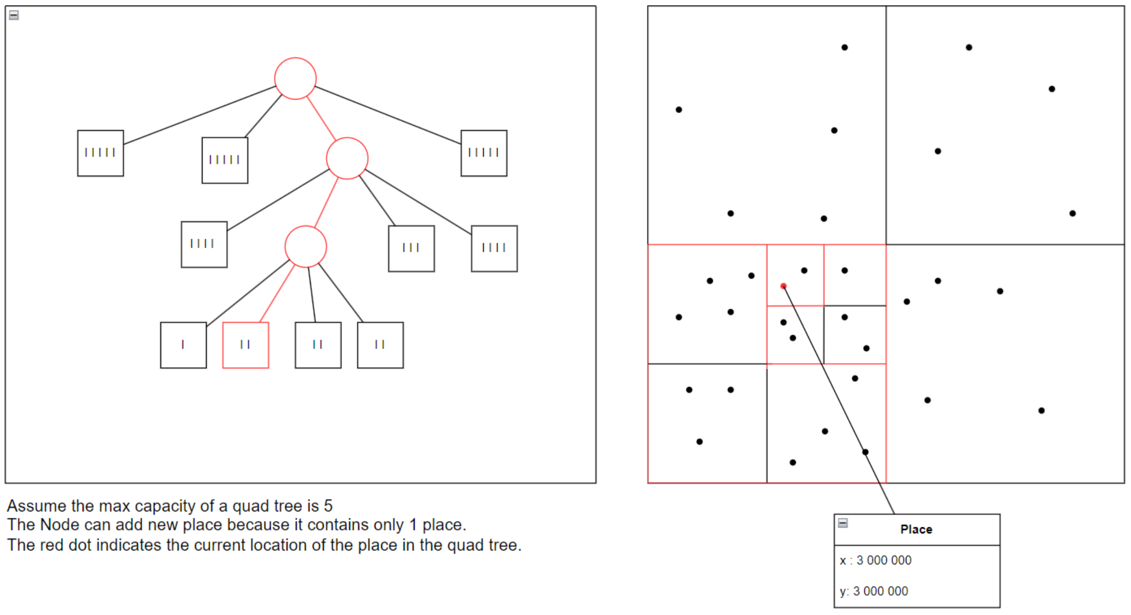
1. The current node is not the leaf node.
2. The current node is the leaf node, and it has enough capacity to store the new place.
3. The current node is the leaf node, but it doesn't have enough capacity to store the new place.

Step 3.1: If the current node is not the leaf node, the system will determine the suitable child node for the given coordinates (x, y) using the getSuitableLeaf() method, and recursively add the new place to that child node until it reaches the leaf node.



The red square and circle indicates all the nodes that the quad tree would traverse through to find the approriate node to insert new place

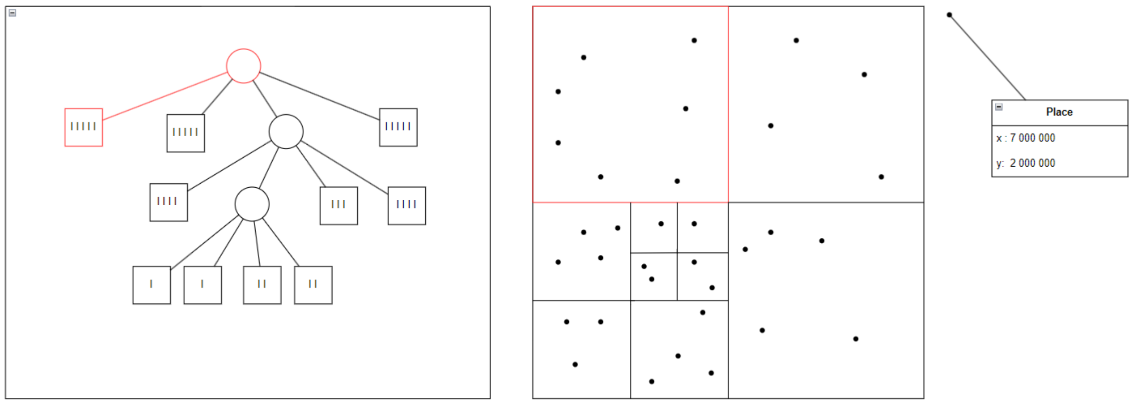
<u>Step 3.2</u>: If the current node is the leaf node, and it has enough capacity to store the new place, the system will directly insert the place into the current node by storing the place's coordinates (x, y) and services in several arrays of integer and increase the numOfPlaces attribute of the current node. We will explain the reason why we use multiple arrays of integers instead of a single array of Place objects in the 3.3.Optimization section.



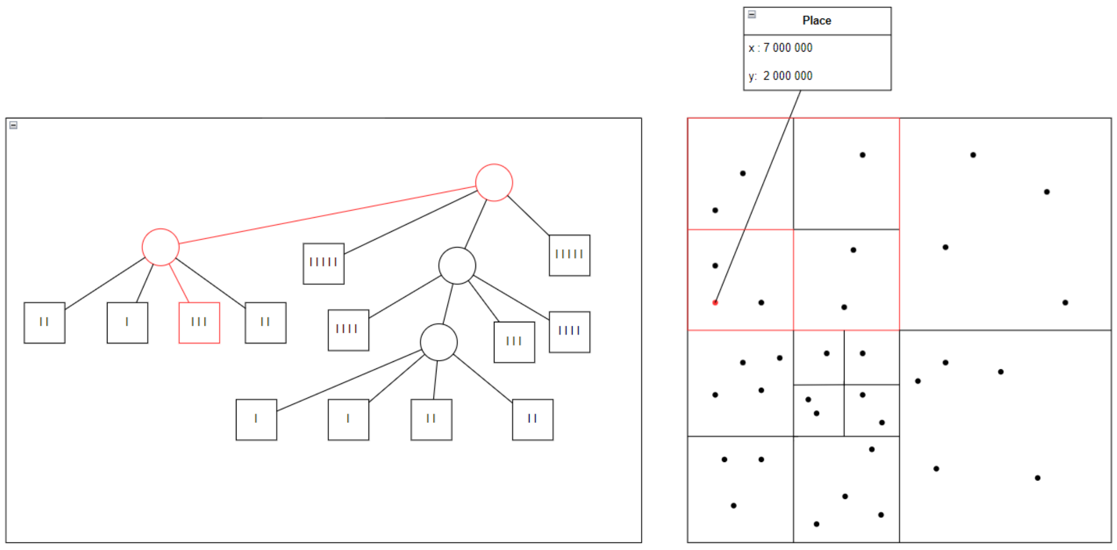
Assume the max capacity of a quad tree is 5
The Node can add new place because it contains only 1 place.
The red dot indicates the current location of the place in the quad tree.

<u>Step 3.3</u>: If the current node is the leaf node, but it doesn't have enough capacity to store the new place, the system will divide the current node into smaller sub-nodes using the split() method. After splitting, it will recursively call the addPlace() method again to insert the place into the appropriate leaf node.

Before splitting:



After splitting:



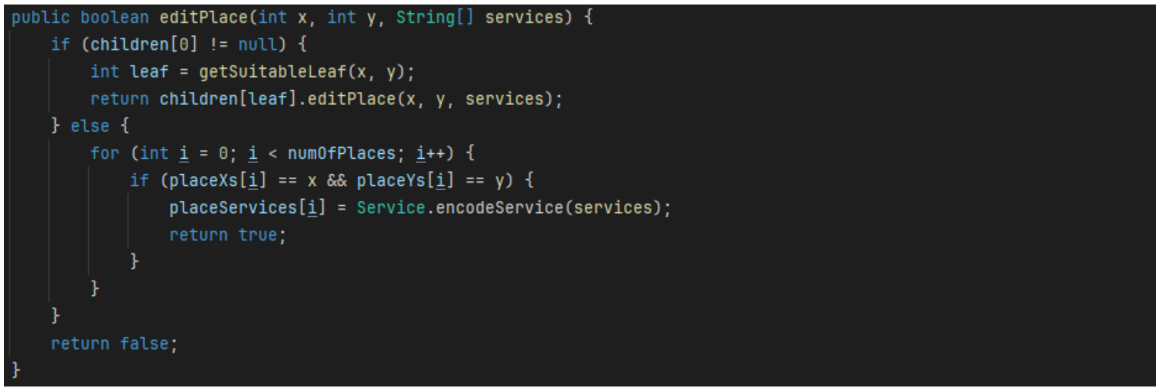
Assume the max capacity of a quad tree is 5 and the current node exceed reach 6 places so the quad tree split it to 4 subdivisions.
After splitting it continue to find the approriate node and insert the place.
The red dot indicates the current location of the place in the quad tree.

3.2.2. Edit & Remove Place

As editPlace() and removePlace() methods were implemented with the same traverse algorithm, we will combine the algorithm analysis of these 2 methods in 1 section.

To edit a place existing on the map, the system will have to work with the editPlace() method in the Map2D class, which requires 3 parameters:

- int x: The x-coordinate of the place.
- int y: The y-coordinate of the place.
- String[] services: The updated service list of the place.

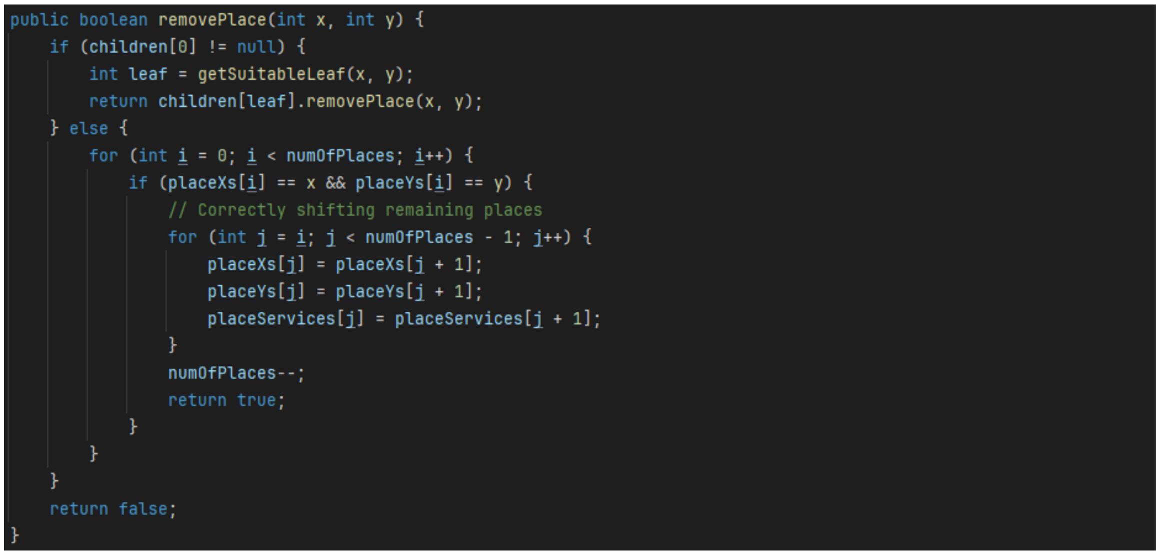


```java
public boolean editPlace(int x, int y, String[] services) {
    if (children[0] != null) {
        int leaf = getSuitableLeaf(x, y);
        return children[leaf].editPlace(x, y, services);
    } else {
        for (int i = 0; i < numOfPlaces; i++) {
            if (placeXs[i] == x && placeYs[i] == y) {
                placeServices[i] = Service.encodeService(services);
                return true;
            }
        }
    }
    return false;
}
```

Figure 7: editPlace() method in the Map2D class

And to remove an existing place from the map, the system will have to work with the removePlace() method in the Map2D class, which requires 2 parameters:

- int x: The x-coordinate of the place.
- int y: The y-coordinate of the place.



```java
public boolean removePlace(int x, int y) {
    if (children[0] != null) {
        int leaf = getSuitableLeaf(x, y);
        return children[leaf].removePlace(x, y);
    } else {
        for (int i = 0; i < numOfPlaces; i++) {
            if (placeXs[i] == x && placeYs[i] == y) {
                // Correctly shifting remaining places
                for (int j = i; j < numOfPlaces - 1; j++) {
                    placeXs[j] = placeXs[j + 1];
                    placeYs[j] = placeYs[j + 1];
                    placeServices[j] = placeServices[j + 1];
                }
                numOfPlaces--;
                return true;
            }
        }
    }
    return false;
}
```

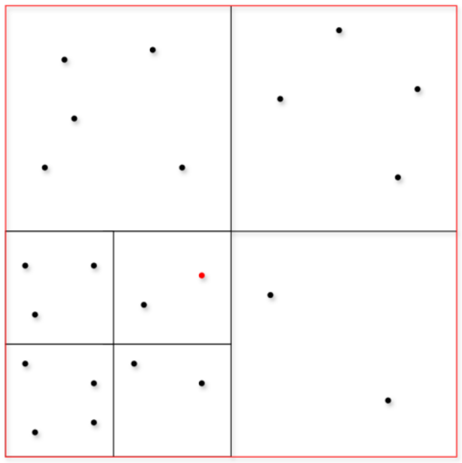
Figure 8: removePlace() method in the Map2D class

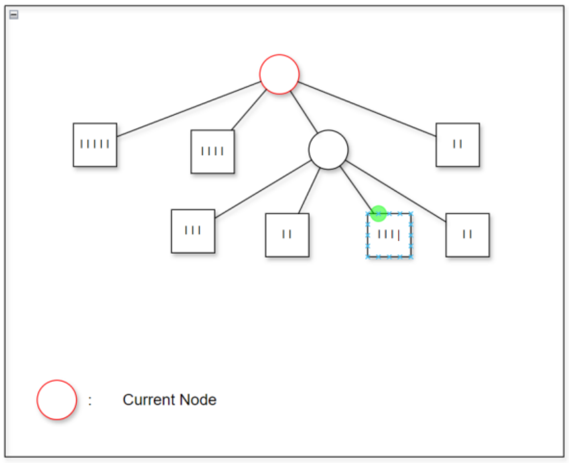
Assuming all the input parameters are valid, the algorithm for editing and removing an existing place on the map will be processed with the following steps:

Step 1: Check whether the current node is the leaf node.

Step 2.1: If the current node is not a leaf node, the system will recursively call the editPlace() / removePlace() method on the appropriate child node to find the area that contains the place to be edited/removed.

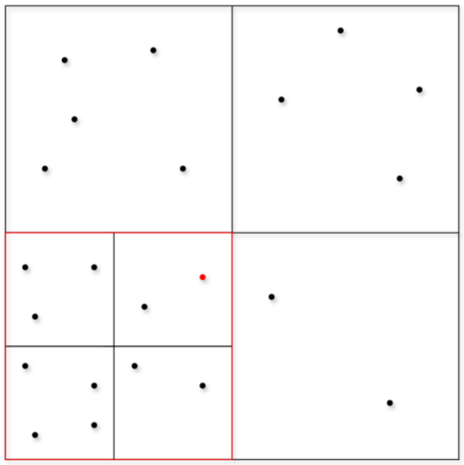
Node level 1:

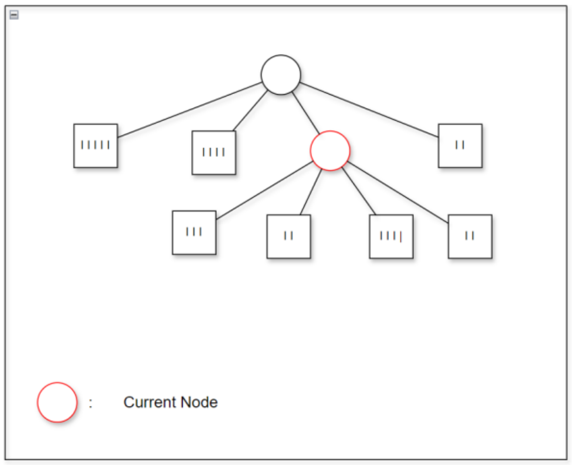




Red rectangle indicates the current searching area.
Red point indicates the searching place.

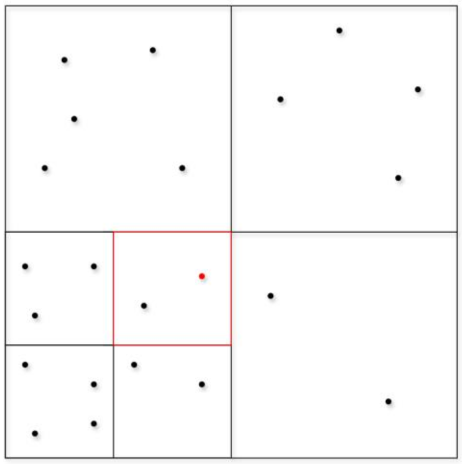
Node level 2:

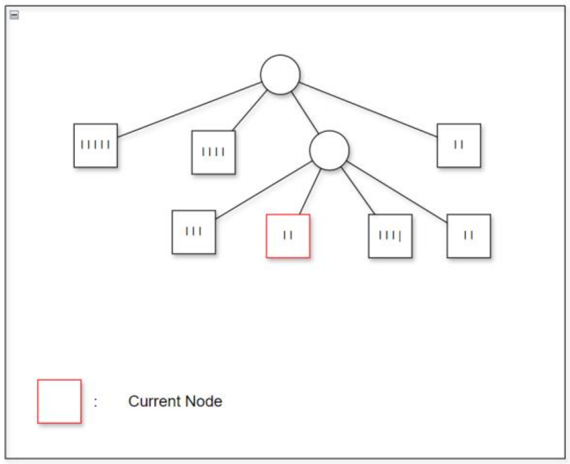




Red rectangle indicates the current searching area.
Red point indicates the searching place.

Node level 3 (Leaf Node):





Red rectangle indicates the current searching area.
Red point indicates the searching place.

Step 2.2: If the current node is already a leaf node, the system will iterate through all places stored in the node to find the place that has the appropriate coordinates (x, y).

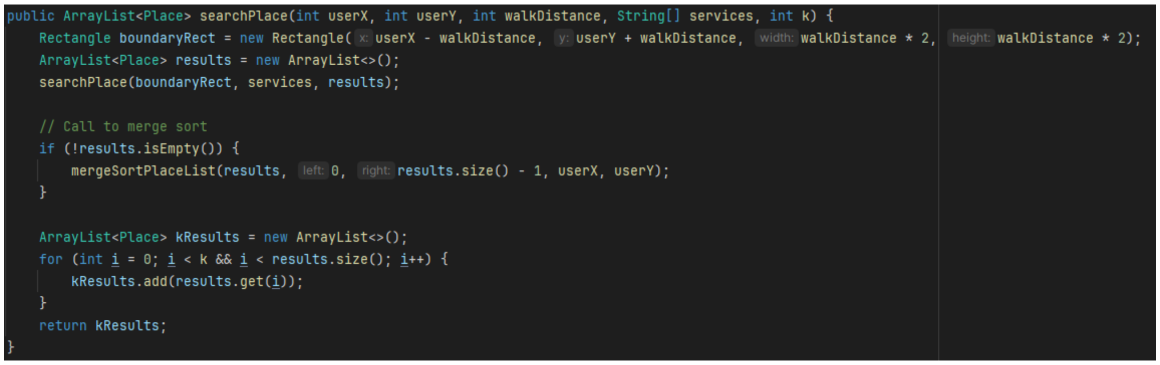
Step 3.1: For editPlace() method, after the appropriate place is found, the system will update the available services of that place by modifying the placeServices attribute of the current node.

Step 3.2: For removePlace() method, after the appropriate place is found, the system will remove it from the current node by shifting the remaining places in the array to fill the gap created by the removed place.

3.2.3. Search Place

To search for a specific number of places on the map, the system will have to work with the searchPlace() method in the Map2D class, which requires 5 parameters:

- int userX: The x-coordinate of the user’s position.
- int userY: The y-coordinate of the user's position.
- int walkDistance: The maximum distance that user is willing to walk.
- String[] services: The list of services that user want to search.
- k: The maximum number of places to return.

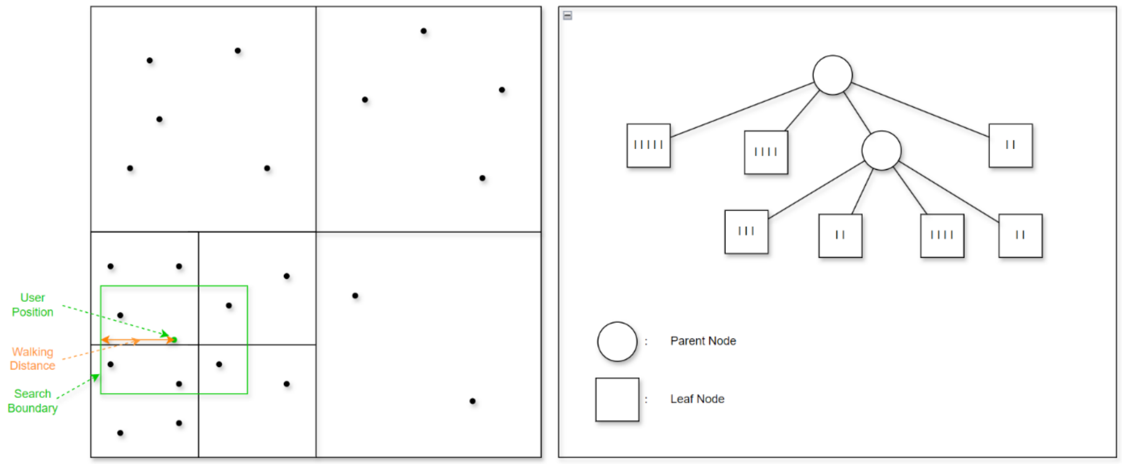


```java
public ArrayList<Place> searchPlace(int userX, int userY, int walkDistance, String[] services, int k) {
    Rectangle boundaryRect = new Rectangle( x: userX - walkDistance,  y: userY + walkDistance,  width: walkDistance * 2,  height: walkDistance * 2);
    ArrayList<Place> results = new ArrayList<>();
    searchPlace(boundaryRect, services, results);

    // Call to merge sort
    if (!results.isEmpty()) {
        mergeSortPlaceList(results,  left: 0,  right: results.size() - 1, userX, userY);
    }

    ArrayList<Place> kResults = new ArrayList<>();
    for (int i = 0; i < k && i < results.size(); i++) {
        kResults.add(results.get(i));
    }
    return kResults;
}
```

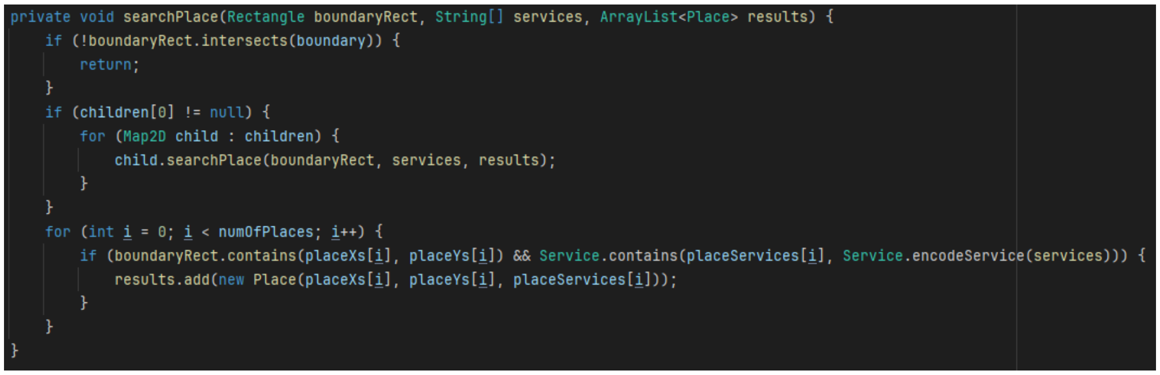
Figure 9: searchPlace() method in the Map2D class

Assuming all the input parameters are valid, the algorithm for searching for a specific number of places on the map will be processed with the following steps:

Step 1: Create a boundary rectangle based on the user’s position and the maximum distance that user is willing to walk.



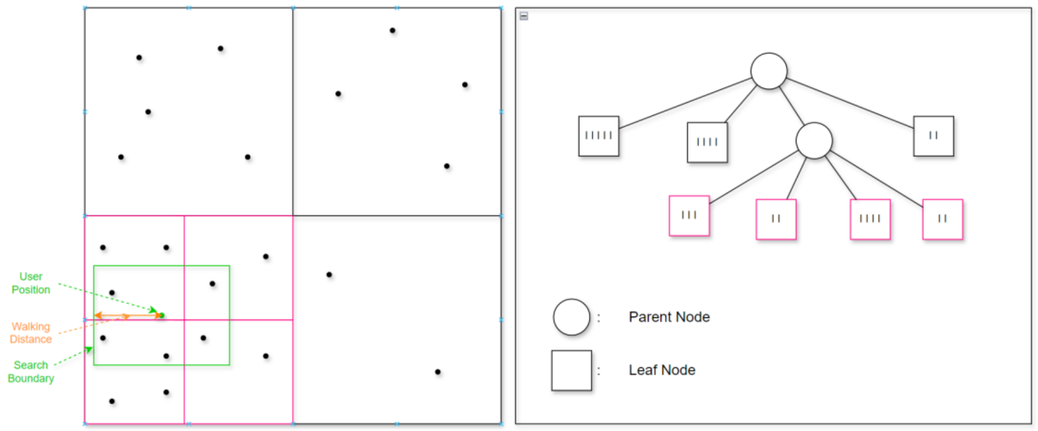
Step 2: Recursively search for all the suitable places within the boundary rectangle using a private helper method called searchPlace() and add those places into the results ArrayList.



```
private void searchPlace(Rectangle boundaryRect, String[] services, ArrayList<Place> results) {
    if (!boundaryRect.intersects(boundary)) {
        return;
    }
    if (children[0] != null) {
        for (Map2D child : children) {
            child.searchPlace(boundaryRect, services, results);
        }
    }
    for (int i = 0; i < numOfPlaces; i++) {
        if (boundaryRect.contains(placeXs[i], placeYs[i]) && Service.contains(placeServices[i], Service.encodeService(services))) {
            results.add(new Place(placeXs[i], placeYs[i], placeServices[i]));
        }
    }
}
```

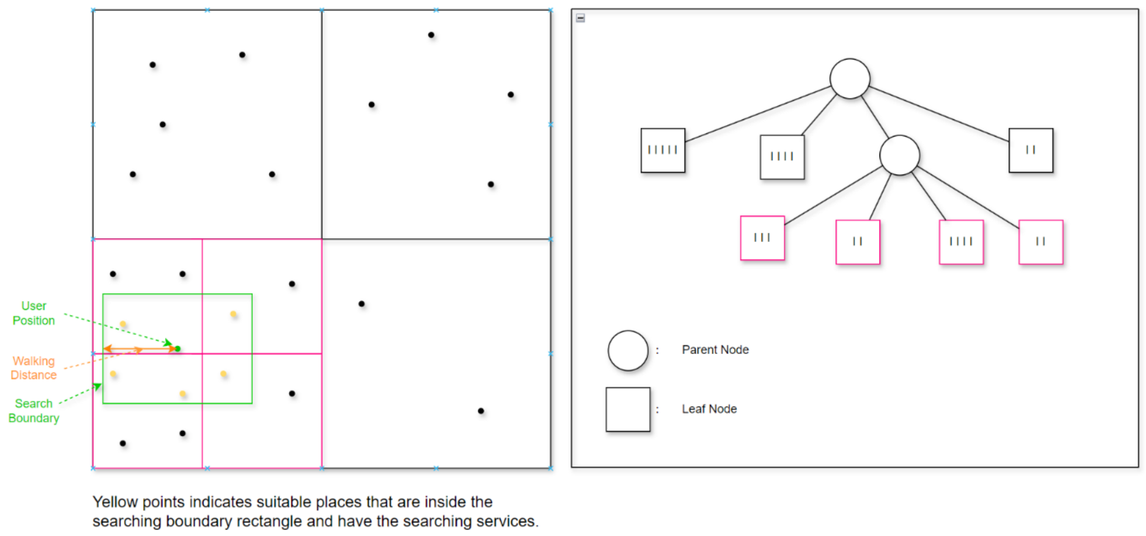
Figure 10: searchPlace() helper method in the Map2D class

Firstly, the system will traverse through the QuadTree to find the nodes that intersect with the searching boundary.

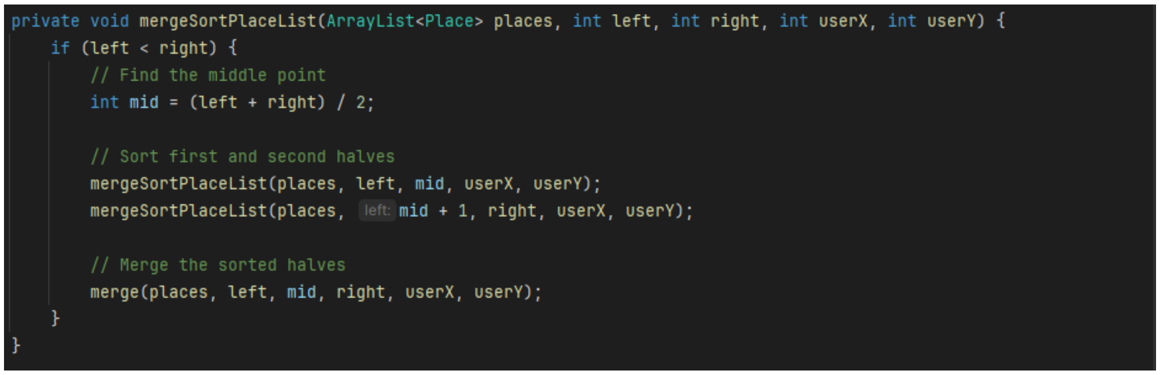


Red rectangle indicates the nodes which intersect with the seaching boundary.

Then, it will collect all the suitable places that are inside the searching boundary rectangle and have the searching services.


User Position
Walking Distance
Search Boundary
Yellow points indicates suitable places that are inside the searching boundary rectangle and have the searching services.
Parent Node
Leaf Node

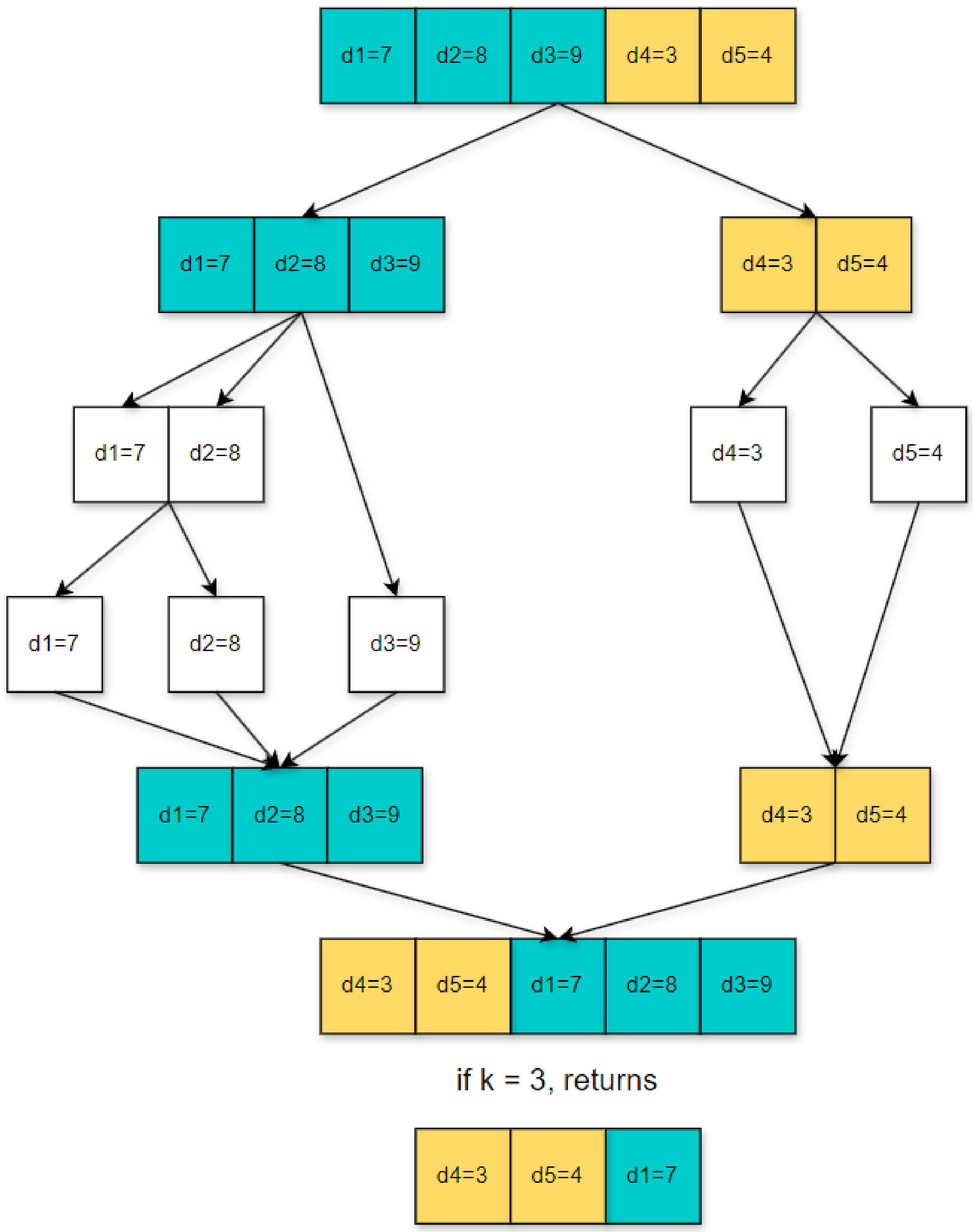
Step 3: After searching for suitable places, the system will sort the results ArrayList based on the distance of place to the user's position using the Merge Sort algorithm. The Merge Sort algorithm divides the array into smaller subarrays, sorts each subarray independently, and then merges the sorted subarrays to produce the final sorted result. This sorting algorithm ensures that places that are closer to the user's position will appear first in the results.

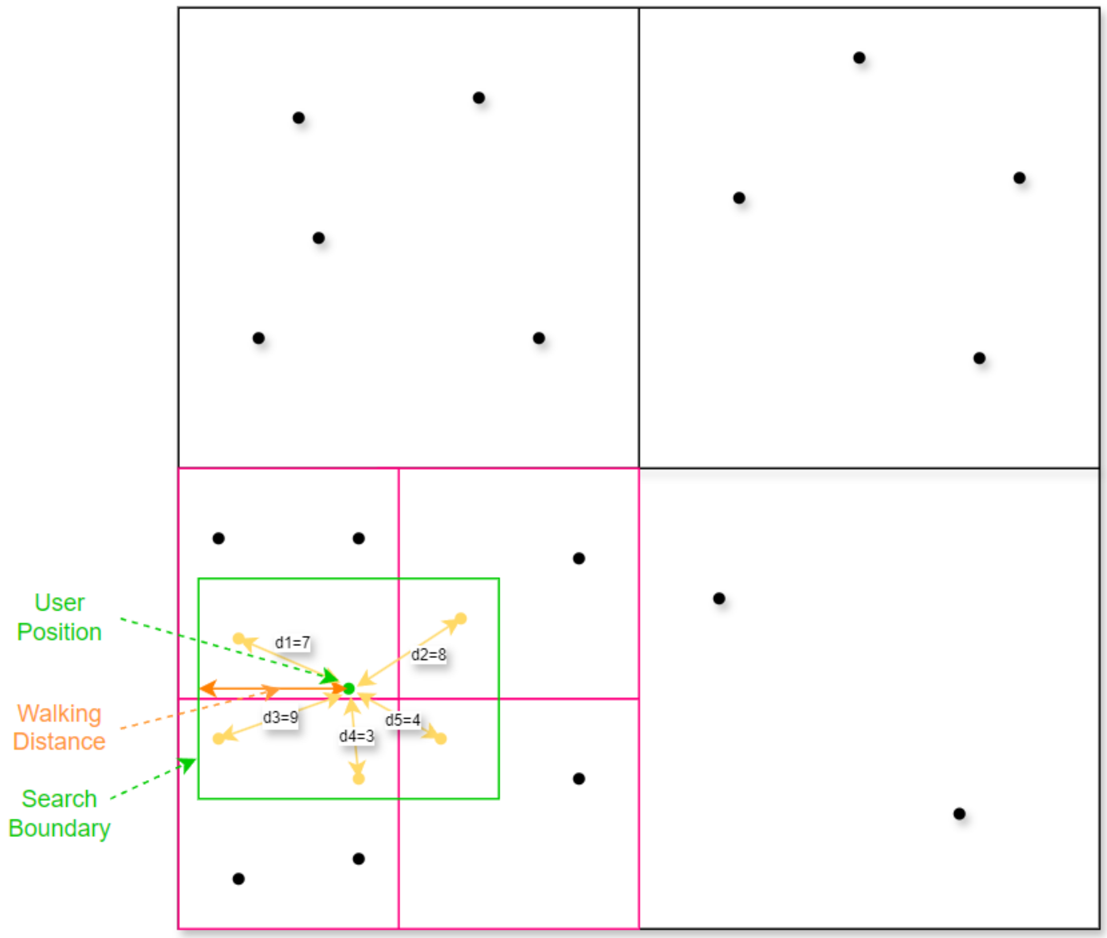


```
private void mergeSortPlaceList(ArrayList<Place> places, int left, int right, int userX, int userY) {
    if (left < right) {
        // Find the middle point
        int mid = (left + right) / 2;

        // Sort first and second halves
        mergeSortPlaceList(places, left, mid, userX, userY);
        mergeSortPlaceList(places,  left: mid + 1, right, userX, userY);

        // Merge the sorted halves
        merge(places, left, mid, right, userX, userY);
    }
}
```

Figure 11: mergeSortPlaceList() method in the Map2D class

Merge Sort Algorithm Visualization:





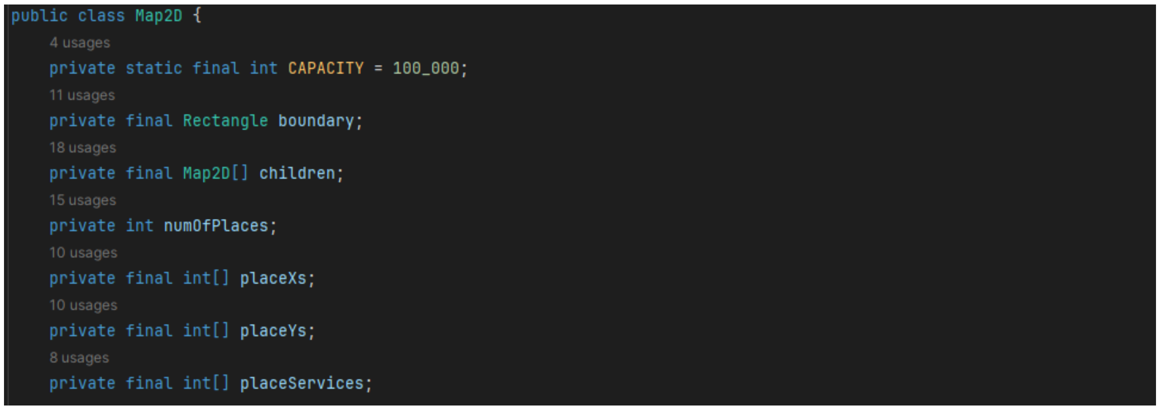
d1, d2, d3, d4, d5 represent for the distance between suitable places and the user's position.

Step 4: When the sorting process is completed, the system will create a new ArrayList to store the first k places from the sorted place list and return that ArrayList as the search results.

3.3. Optimization

Our team had done many experiments during the implementation process, and we found that setting a node's capacity to 100,000 places and storing places in a node using multiple arrays of integers is the best way to optimize the code performance.

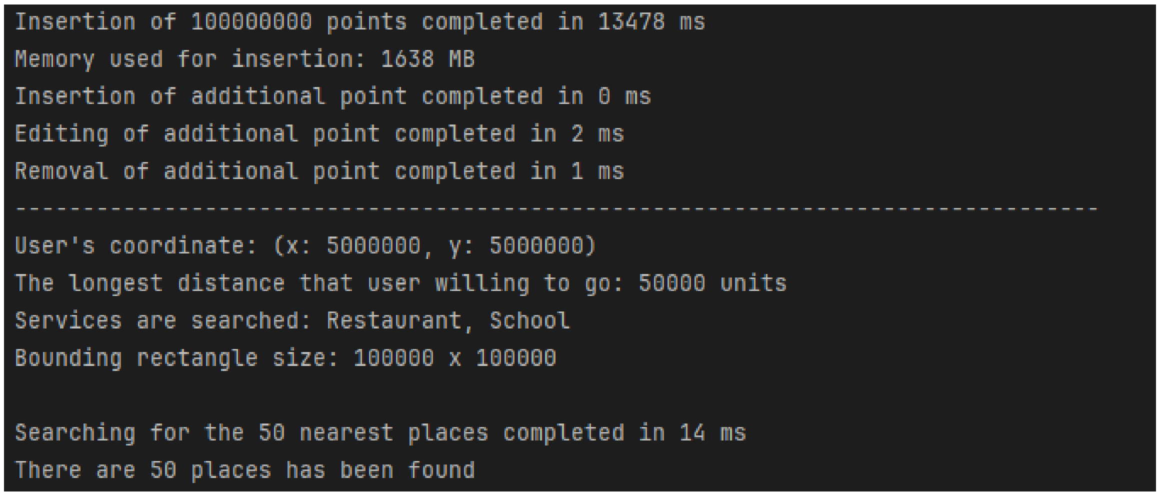
Firstly, storing places in a node using multiple arrays of integers helps us to reduce the consumption of memory space significantly, from 6 gigabytes to 1.6 gigabytes to insert 100 million places. At first, when we tried to store places in a tree node using a single array of Place objects, the system consumed a huge amount of memory space as it created 100 million Place objects and 100 million Service objects. Moreover, in that case, the program could not even be executed when we used a laptop whose 8 gigabytes of RAM, as that computer ran out of heap memory space while trying to insert 100 million places into the map. Therefore, our solution is to encode the place's services into integer representation, store all the place's attributes in separate arrays of integers, and keep track of the number of places in a node using numOfPlaces attributes. This optimization helps us to reduce the memory space to store places and the running time to iterate through the places array.



```
public class Map2D {
    4 usages
    private static final int CAPACITY = 100_000;
    11 usages
    private final Rectangle boundary;
    18 usages
    private final Map2D[] children;
    15 usages
    private int numOfPlaces;
    10 usages
    private final int[] placeXs;
    10 usages
    private final int[] placeYs;
    8 usages
    private final int[] placeServices;
```

Figure 12: How places are stored in multiple arrays of integers in the Map2D class

Furthermore, we have to assert that a node that can store at most 100,000 places is not a random number. When we tried to reduce the capacity of a node to 100, the program consumed more than 8 gigabytes of memory as it had to create a lot of node objects to store 100 million places on the map. We also tried to increase the node's capacity to 10 million, although it only took about 1.5 gigabytes of memory to insert 100 million places into the map, the system had to spend 53ms to search for 50 nearest places. After several experiments, our team agreed that setting the node's capacity to 100,000 is the most efficient option, as it only took 1.6 gigabytes of memory to insert 100 million places and 14ms for the searching process.



```
Insertion of 100000000 points completed in 13478 ms
Memory used for insertion: 1638 MB
Insertion of additional point completed in 0 ms
Editing of additional point completed in 2 ms
Removal of additional point completed in 1 ms
--------------------------------------------------------------------------------
User's coordinate: (x: 5000000, y: 5000000)
The longest distance that user willing to go: 50000 units
Services are searched: Restaurant, School
Bounding rectangle size: 100000 x 100000

Searching for the 50 nearest places completed in 14 ms
There are 50 places has been found
```

Figure 13: Time and space consumption after the optimization

4. COMPLEXITY ANALYSIS

4.1. Assumptions

For the successful implementation and operation of the Map2D Abstract Data Type (ADT) using a QuadTree structure in the mapping application, several assumptions have been made:

- **Valid Data Input**: All input data regarding place coordinates, types of services, and dimensions of the search rectangle are assumed to be valid and correct as provided by the user. There is no need for additional validation checks within the system.
- **Fixed Coordinates**: Once a place is added to the map, its coordinates (x, y) are fixed and cannot be changed. Only the services offered at each location can be modified through the edit function.
- **Service Encoding**: The types of services provided by places are encoded into an integer format to streamline the storage and comparison processes. This encoding assumes a predefined set of up to 10 service types, simplifying the management of service data.
- **Spatial Constraints**: The map operates within a defined size of 10,000,000 x 10,000,000 units. All places are assumed to fall within this boundary, and the search operations are confined to this space.
- **Search Area Limitations**: The bounding rectangle for searches has minimum and maximum size limits (from 100 x 100 to 100,000 x 100,000 units), which are adhered to in all search queries to ensure performance efficiency.
- **Capacity Limits**: The QuadTree node has a capacity limit of 10,000,000 places per node before it splits into four child nodes. This is assumed to adequately balance the load and maintain query performance.
- **Distance Metric**: The system uses the Euclidean distance metric for determining proximity between points. This assumption is crucial for sorting and filtering the search results based on the distance from the user's specified location.
- **Result Limitation**: A maximum of 50 places can be displayed in the search results. This limit ensures that the performance is manageable, and the user is not overwhelmed by too many results.

These assumptions are integral to the design and functionality of the system, providing a framework within which the QuadTree operates efficiently and effectively in handling and querying geospatial data.

4.2. Algorithms Analysis

4.2.1. Add Place

The addPlace() method in the Map2D class is designed to efficiently insert a place with its coordinates and associated service type into the data structure. To analyze its complexity, we must consider the steps involved in inserting a new place, the conditions under which the tree splits, and the overall behavior of the quadtree as it grows.

Direct Insertion without Splitting:

If the number of places stored in the current quadtree node is less than its CAPACITY, the operation involves simply adding the new place's coordinates and service type to the arrays placeXs, placeYs, and placeServices. This operation is O(1), as it only involves direct array access at the index specified by numOfPlaces.

Insertion with Splitting:

If the capacity is reached, the quadtree node splits. The split() operation involves creating four new child quadrants and redistributing the existing places among these children based on their coordinates:

- **Creating new quadrants**: This is a constant time operation (O(1)) as it involves initializing four new QuadTree objects with predefined boundaries based on the current node's boundary.
- **Redistributing places**: Each of the numOfPlaces (up to CAPACITY) needs to be re-evaluated to determine which of the new quadrants they belong to. This involves calculating the appropriate quadrant for each place using getSuitableLeaf(), which is O(1) per place due to simple arithmetic operations and conditional checks. Thus, redistributing all places is O(n) where n is the number of places in the node at the time of splitting.

After redistribution, the new place is added to the appropriate child quadtree. The time complexity for this step depends recursively on the depth of the tree and the number of redistributions required as the tree grows.

Recurrence and Worst Case:

In the worst case, especially in densely populated areas of the map where many places might fall within the same quadrant continuously, the tree might need to split several times, potentially until individual nodes represent very small areas. The worst-case complexity of insertion, considering continuous splits at each level, is O(log n) where n is the number of places, assuming that space is uniformly used. In non-uniform scenarios where splits are more frequent due to dense clusters of places, this could degrade towards O(n) in the worst case if many places need to be repeatedly redistributed at multiple levels of the tree.

Average Case:
On average, if the distribution of places is relatively uniform across the map, most insertions will be direct (O(1)), and splits will occur less frequently as the tree's depth increases. Therefore, average-case complexity is generally better, leaning towards O(log n) because each level of the tree quadratically increases the capacity to hold more places without needing further splits.

Conclusion:
The addPlace() operation in a Map2D class is generally efficient for spatial data, especially when the data is uniformly distributed. The average complexity is around O(log n), but specific scenarios involving highly clustered data can lead to a worst-case scenario of O(n) due to repeated splits and redistributions. This highlights the importance of understanding data distribution when using a quadtree for spatial indexing.

4.2.2. Edit Place

The editPlace() method in the Map2D class facilitates the modification of the service types offered by a place at specific coordinates (x, y) without altering its position within the quadtree. The complexity analysis of this operation must consider both scenarios: whether the target node has children (indicating it has been split) or not.

Node with Children:
If the node has been split (i.e., it has children), the method must first determine which child quadrant the given coordinates fall into. This determination is made using the getSuitableLeaf() method, which computes the appropriate child based on the coordinates. This operation is O(1) as it involves a few arithmetic operations and conditional checks.

Once the correct child node is identified, editPlace() is recursively called on that child. This recursive approach continues until a leaf node (a node without children) is reached where the actual data (place coordinates and services) is stored. The complexity, therefore, depends on the depth of the quadtree at the location of the place being edited. In a balanced tree scenario, where data is uniformly distributed, this would typically result in a complexity of O(log n), where n is the number of places. This is because the depth of a well-balanced quadtree grows logarithmically with the number of elements.

Leaf Node (Node without Children):
In the case where the node has not been split (no children), the operation involves iterating through the numOfPlaces stored directly at that node. Each place's coordinates are checked against the provided (x, y) values:

- **Iteration**: The iteration over numOfPlaces is O(n) where n is the number of places in the node, which could be up to the capacity limit of the node in the worst case.

- **Checking and updating**: For each place, coordinates are compared, and if they match, the service data is updated. The service update involves encoding the array of service strings into an integer format using Service.encodeService(), which is typically O(m), where m is the number of services to encode.

Thus, in a non-split node, if the place is found early in the list, the operation could be near O(1), but the worst-case scenario would be O(n*m), where n is the number of places in the node and m is the number of services being encoded.

Overall Complexity:
The worst-case complexity of editPlace() can be approximated as O(log n + m) in a balanced tree, considering the logarithmic depth traversal and the time to encode services. However, in unbalanced scenarios or particularly dense areas of the map, this complexity could lean more towards O(n*m), especially if many places need checking before finding the correct one or if the tree is deeper due to many splits at high-density points.

Conclusion:
The editPlace() operation is generally efficient for balanced quadtrees where places are uniformly distributed, mainly operating with complexity around O(log n + m). However, specific data distributions that lead to unbalanced trees or densely populated nodes can significantly affect performance, especially when large numbers of services are involved, or nodes contain many places.

4.2.3. Remove Place

The removePlace() method in the Map2D class is designed to delete a place with specified coordinates (x, y) from the QuadTree structure. This analysis must consider the scenarios where the node has children (indicating that it has been split) and where it does not (indicating a leaf node).

Node with Children:
If the node has been split and thus contains children, the operation begins by determining the appropriate child quadrant for the given coordinates using the getSuitableLeaf() method. This determination is O(1), involving simple arithmetic and conditional checks.

Once the correct child node is identified, removePlace() is recursively called on that child. This recursion continues until a leaf node is reached where the place is actually stored. The complexity of this part of the operation thus depends on the depth of the tree at the location from which the place is being removed. In a uniformly distributed dataset, where the tree remains balanced, the depth grows logarithmically with the number of elements, leading to a complexity of O(log n) for reaching the correct leaf.

<u>Leaf Node (Node without Children)</u>:
In the case of a leaf node (no children), the operation entails searching through the places stored directly at the node:

- **Searching**: Each place's coordinates are checked against the provided (x, y) values in a linear fashion, which takes O(n) where n is the number of places in the node.
- **Deleting**: Once the matching place is found, it is removed. The removal is handled by overwriting the data of the deleted place with the data from the last element in the list, thus avoiding the need to shift elements. This deletion step is O(1) as it involves only a few direct array assignments.

Thus, if the matching place is found early in the list, the operation can be near O(1), but the worst-case scenario, when the place is the last one checked or not found at all, would be O(n).

<u>Overall Complexity</u>:
The worst-case complexity of removePlace() in a balanced quadtree would primarily be determined by the depth of the tree, approximated as O(log n) due to the logarithmic increase in depth with the number of elements in a balanced scenario. However, this assumes that once the correct node is reached, the removal process itself (which is linear in the number of places in the node) does not significantly impact performance. In dense areas or unbalanced scenarios, where nodes may contain a large number of places, the performance could degrade towards O(n) due to the linear search required to find the place.

<u>Conclusion</u>:
The removePlace() operation in the Map2D class is generally efficient, especially when the data is uniformly distributed and the tree remains balanced, primarily operating with a complexity of O(log n). However, specific scenarios involving unbalanced trees or densely populated nodes can significantly affect performance, particularly when a large number of places need to be checked before finding the correct one to remove. This emphasizes the importance of considering data distribution when using quadtrees for managing spatial data.

4.2.4. Search Place

The searchPlace() method in the Map2D class is responsible for finding all places within a specified bounding rectangle that match given service criteria, sorting them by proximity to a specified user location, and returning the top k closest results. This operation combines spatial querying with sorting, which influences its overall complexity.

<u>Searching for Places</u>:

- Intersection Check: This remains an O(1) operation, as it simply requires a comparison of boundary coordinates.

- Recursive Search: If there is an intersection, the method recursively checks each child (if the node has children) or directly checks each place stored at the node (if it is a leaf node). Each place undergoes a containment and service match check.
- Containment and Service Match: These checks are also considered O(1), assuming efficient service encoding and comparison processes.

In densely populated or overlapping regions, many or all places might need to be checked at multiple levels of the tree, potentially leading to a high number of operations. However, due to the quadtree's nature, which spatially partitions data and limits the number of places per node, the average complexity for searching is ideally around O(log n + m), where n is the number of places and m is the number of places that meet the criteria and are checked at the leaf level.

Sorting the Results:
After gathering all the matching places, the results list is sorted based on the Euclidean distance from the user's coordinates using merge sort. Merge sort is particularly effective for its stable sorting mechanism and consistent performance, irrespective of the input distribution.

Merge Sort Complexity:
Merge sort operates with a worst-case and average time complexity of O(m log m), where m is the number of elements in the list. This complexity is advantageous for ensuring that the time for sorting operations remains consistent and predictable across various scenarios.

Selecting Top k Results:
The final step in the searchPlace() method is constructing a list of the top k closest places. This selection is an O(k) operation, which is generally insignificant compared to the sorting phase, especially when k is much smaller than n.

Overall Complexity:
Combining these components, the overall complexity of the searchPlace() operation can be described as:

- O(log n + m) for the search process, where n is the number of places in the quadtree and m is the number of matching places found.
- O(m log m) for sorting these matches by distance using merge sort.
- O(k) for extracting the top k results.

The worst-case complexity could approach O(n log n) if a large portion of the dataset matches the search criteria and requires sorting. Nonetheless, in typical scenarios where m is a small subset of n, the complexity is more manageable and efficient.

Conclusion:

The searchPlace() method effectively utilizes the spatial indexing capabilities of the quadtree to limit the search space, enhancing efficiency. The addition of Merge Sort for the sorting phase provides a stable and consistent computational expense, contributing significantly to the overall computational cost, especially in cases where many places meet the search criteria. This robust method ensures that search operations are both fast and reliable across varied use cases.

4.3. Overall Complexity

Each operation within the QuadTree can vary significantly in complexity based on the specific data distribution and tree balance:

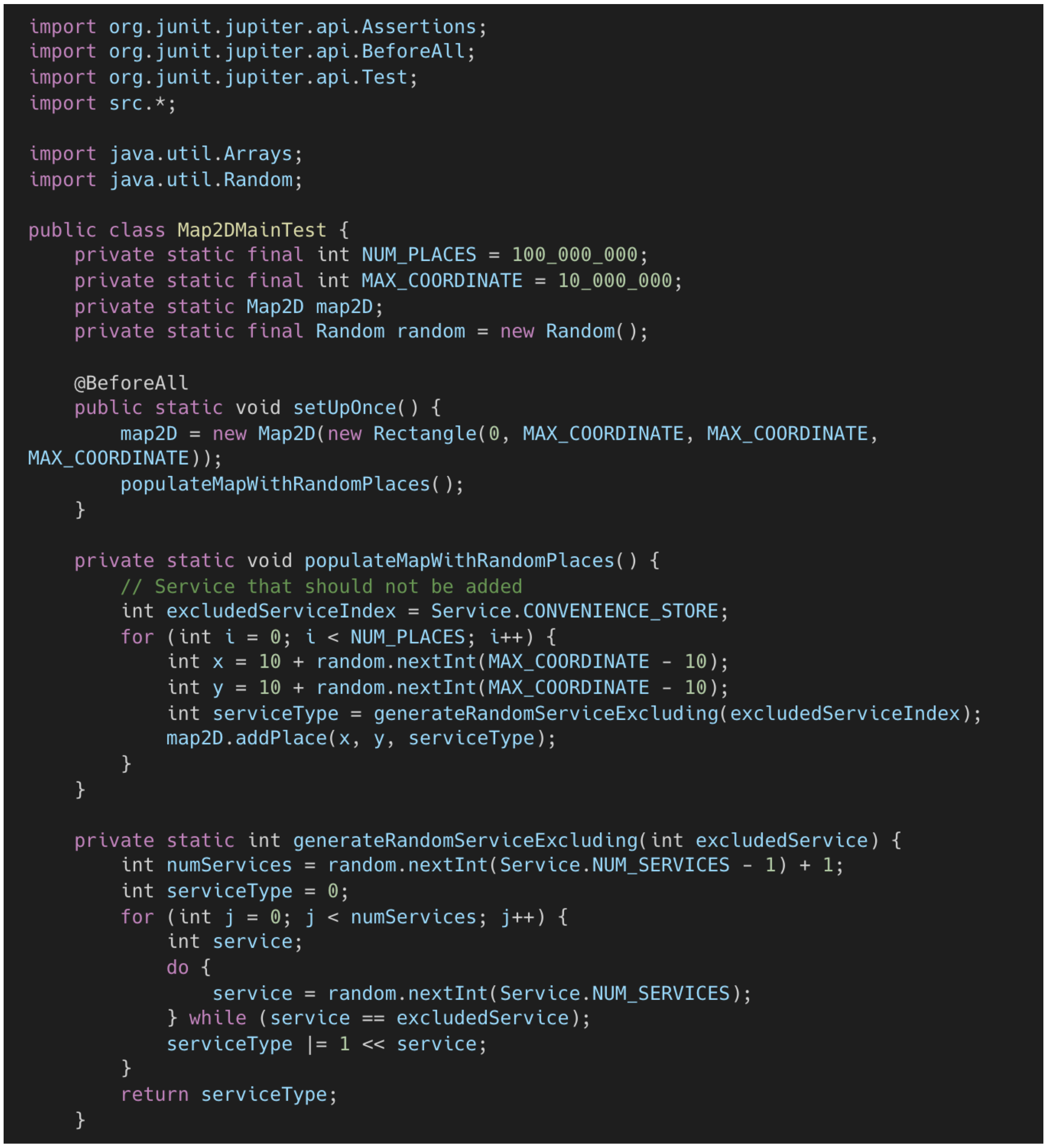
- **Add Place** operations are generally efficient with an average-case complexity close to O(log n) but can worsen to O(n) in densely clustered scenarios.
- **Edit Place** and **Remove Place** operations also average O(log n) in balanced trees but may increase to O(n) in unbalanced or dense scenarios.
- **Search Place** operations effectively demonstrate the strength of the QuadTree in spatial querying, averaging O(log n + m) for searching and O(m log m) for sorting, which can become O(n log n) in scenarios where a large portion of the dataset matches the search criteria.

These complexities underscore the importance of QuadTree as an efficient spatial indexing structure, particularly suited to applications requiring rapid insertion, deletion, and spatial querying such as mapping and location-based services. The efficiency hinges significantly on maintaining a balanced tree and understanding the spatial distribution of the data.

5. EVALUATION

Our approach to evaluating correctness and efficiency involves executing a series of test cases that align with the operations implemented in our project. These test cases are conducted using JUnit Tests. In the subsequent section, we will provide a detailed exposition of the test cases we have developed.

5.1.Testing Set Up



```java
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.Test;
import src.*;

import java.util.Arrays;
import java.util.Random;

public class Map2DMainTest {
    private static final int NUM_PLACES = 100_000_000;
    private static final int MAX_COORDINATE = 10_000_000;
    private static Map2D map2D;
    private static final Random random = new Random();

    @BeforeAll
    public static void setUpOnce() {
        map2D = new Map2D(new Rectangle(0, MAX_COORDINATE, MAX_COORDINATE,
MAX_COORDINATE));
        populateMapWithRandomPlaces();
    }

    private static void populateMapWithRandomPlaces() {
        // Service that should not be added
        int excludedServiceIndex = Service.CONVENIENCE_STORE;
        for (int i = 0; i < NUM_PLACES; i++) {
            int x = 10 + random.nextInt(MAX_COORDINATE - 10);
            int y = 10 + random.nextInt(MAX_COORDINATE - 10);
            int serviceType = generateRandomServiceExcluding(excludedServiceIndex);
            map2D.addPlace(x, y, serviceType);
        }
    }

    private static int generateRandomServiceExcluding(int excludedService) {
        int numServices = random.nextInt(Service.NUM_SERVICES - 1) + 1;
        int serviceType = 0;
        for (int j = 0; j < numServices; j++) {
            int service;
            do {
                service = random.nextInt(Service.NUM_SERVICES);
            } while (service == excludedService);
            serviceType |= 1 << service;
        }
        return serviceType;
    }
```

Figure 14: Testing set up in the Map2DMainTest class

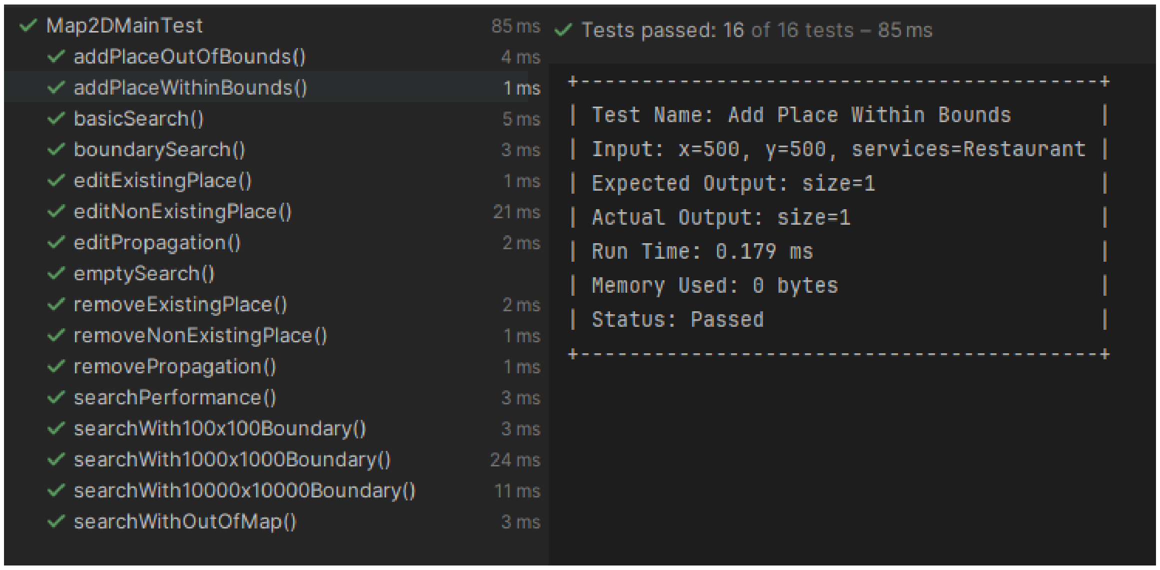
In the primary test suite, we initiate a setup process that executes once each time the test file is run. This setup entails populating the system with 100 million randomly placed points of interest. Each subsequent test within this suite will operate using this identical dataset, ensuring consistency across tests. Specifically, our random service generator excludes the 'Convenience Store' category from these points. This deliberate exclusion is designed to facilitate subsequent tests on scenarios where searches might yield no results, allowing us to assess the robustness of the system when encountering empty search outcomes.

5.2. Add Place Testing & Evaluation

5.2.1. Correctness

Accuracy and Compliance with Specifications:

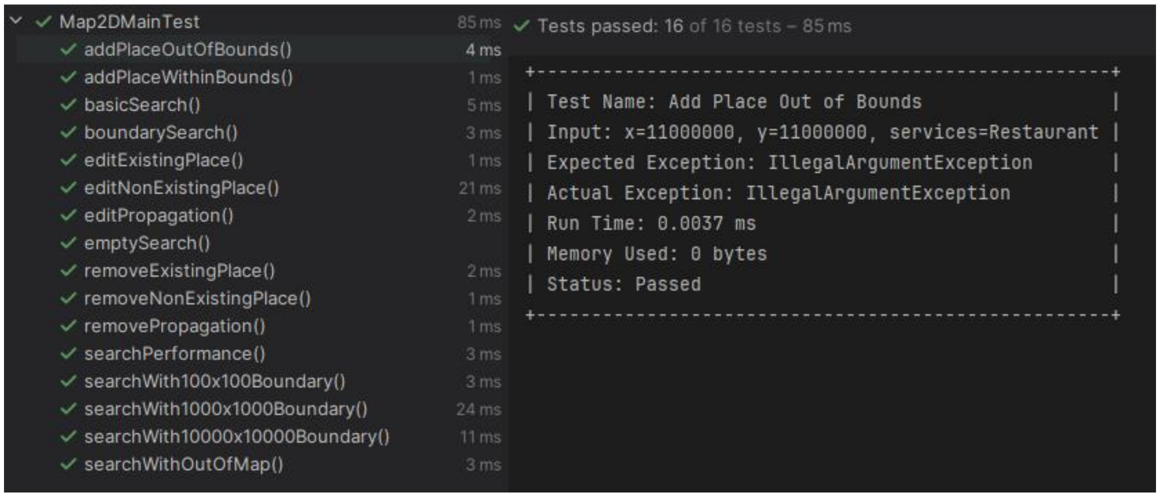
Add Place Within Bounds: This test confirms that the method correctly adds a place within the specified boundaries of the map. The test verifies that the place is indeed added by performing a search immediately after the add operation, with the search correctly returning the added place, thereby affirming the operation's accuracy.



```
✓ Map2DMainTest                          85 ms   ✓ Tests passed: 16 of 16 tests – 85 ms
   ✓ addPlaceOutOfBounds()                4 ms
   ✓ addPlaceWithinBounds()               1 ms   +------------------------------------------+
   ✓ basicSearch()                        5 ms   | Test Name: Add Place Within Bounds       |
   ✓ boundarySearch()                     3 ms   | Input: x=500, y=500, services=Restaurant |
   ✓ editExistingPlace()                  1 ms   | Expected Output: size=1                  |
   ✓ editNonExistingPlace()              21 ms   | Actual Output: size=1                    |
   ✓ editPropagation()                    2 ms   | Run Time: 0.179 ms                       |
   ✓ emptySearch()                               | Memory Used: 0 bytes                     |
   ✓ removeExistingPlace()                2 ms   | Status: Passed                           |
   ✓ removeNonExistingPlace()             1 ms   +------------------------------------------+
   ✓ removePropagation()                  1 ms
   ✓ searchPerformance()                  3 ms
   ✓ searchWith100x100Boundary()          3 ms
   ✓ searchWith1000x1000Boundary()       24 ms
   ✓ searchWith10000x10000Boundary()     11 ms
   ✓ searchWithOutOfMap()                 3 ms
```

Figure 15: Add Place Within Bounds Test Case Result

Add Place Out of Bounds: This test evaluates the method's ability to handle attempts to add a place outside the designated boundaries. The operation correctly throws an IllegalArgumentException, as expected, which demonstrates robust boundary checking and error handling within the method.



```
Map2DMainTest                              85 ms   ✓ Tests passed: 16 of 16 tests – 85 ms
  ✓ addPlaceOutOfBounds()                   4 ms
  ✓ addPlaceWithinBounds()                  1 ms   +-------------------------------------------------+
  ✓ basicSearch()                           5 ms   | Test Name: Add Place Out of Bounds               |
  ✓ boundarySearch()                        3 ms   | Input: x=11000000, y=11000000, services=Restaurant |
  ✓ editExistingPlace()                     1 ms   | Expected Exception: IllegalArgumentException     |
  ✓ editNonExistingPlace()                 21 ms   | Actual Exception: IllegalArgumentException       |
  ✓ editPropagation()                       2 ms   | Run Time: 0.0037 ms                              |
  ✓ emptySearch()                                  | Memory Used: 0 bytes                             |
  ✓ removeExistingPlace()                   2 ms   | Status: Passed                                   |
  ✓ removeNonExistingPlace()                1 ms   +-------------------------------------------------+
  ✓ removePropagation()                     1 ms
  ✓ searchPerformance()                     3 ms
  ✓ searchWith100x100Boundary()             3 ms
  ✓ searchWith1000x1000Boundary()          24 ms
  ✓ searchWith10000x10000Boundary()        11 ms
  ✓ searchWithOutOfMap()                    3 ms
```

Figure 16: Add Place Out of Bounds Test Case Result

Error Handling and Boundary Management:
The method includes explicit boundary checks and raises appropriate exceptions for out-of-bounds input, which is an essential aspect of ensuring data integrity and preventing erroneous data entries in the system.

5.2.2. Efficiency

Performance Metrics:
Both tests measure the execution time and memory usage associated with the add operation. The performance metrics indicate extremely low runtime (milliseconds and sub-millisecond levels) and minimal to no memory overhead during the execution of these operations. This suggests a highly optimized process for adding places to the map.

Resource Management:
Memory management appears effective, with no significant memory allocation needed during the operations, as indicated by the reported zero bytes of memory used. This efficiency in memory usage is beneficial for systems where frequent add operations are expected and where maintaining low memory overhead is critical.

Scalability and System Behavior Under Load:
The add operation's efficiency in routine scenarios (within bounds) is confirmed; however, its behavior under conditions of reaching or exceeding capacity (triggering the split operation) wasn't directly tested. While the current implementation handles small to moderate loads efficiently, the performance impact of the recursive split operations during high loads or at capacity limits might require further evaluation.

5.2.3. Conclusion

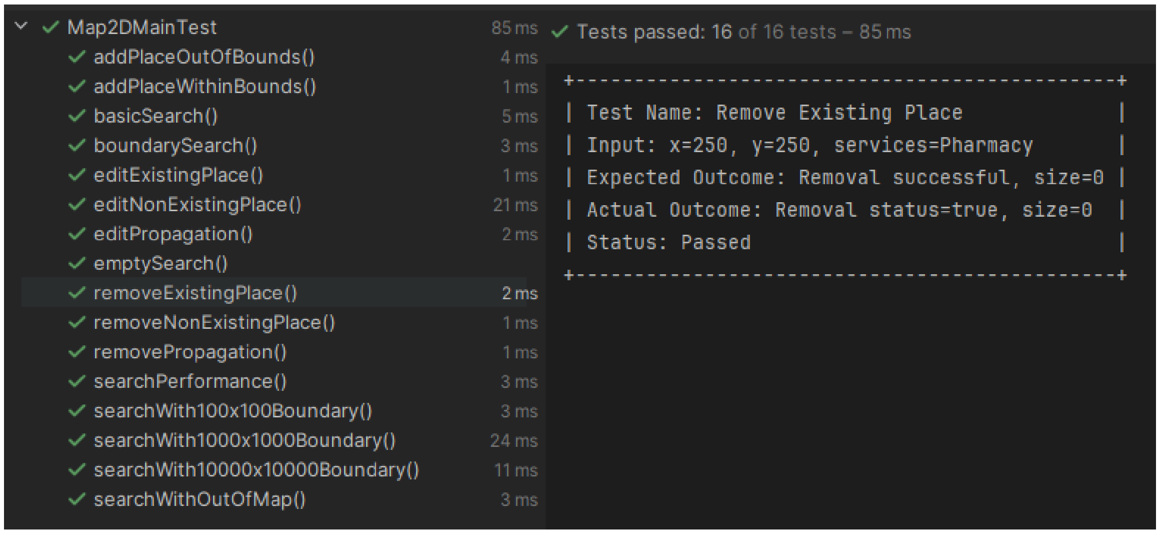
The addPlace() operation tests thoroughly demonstrate the method's functional correctness and efficiency. The tests are well-structured to verify both normal and exceptional behaviors of the method. Although the performance metrics are promising for the scenarios tested, further analysis might be necessary to explore the operation's efficiency and scalability under varied and more extensive conditions. This detailed evaluation not only validates the operation under typical use cases but also highlights the need for potential further testing in edge cases or under maximum capacity scenarios.

5.3. Remove Place Testing & Evaluation

5.3.1. Correctness

Functional Integrity and Accuracy:

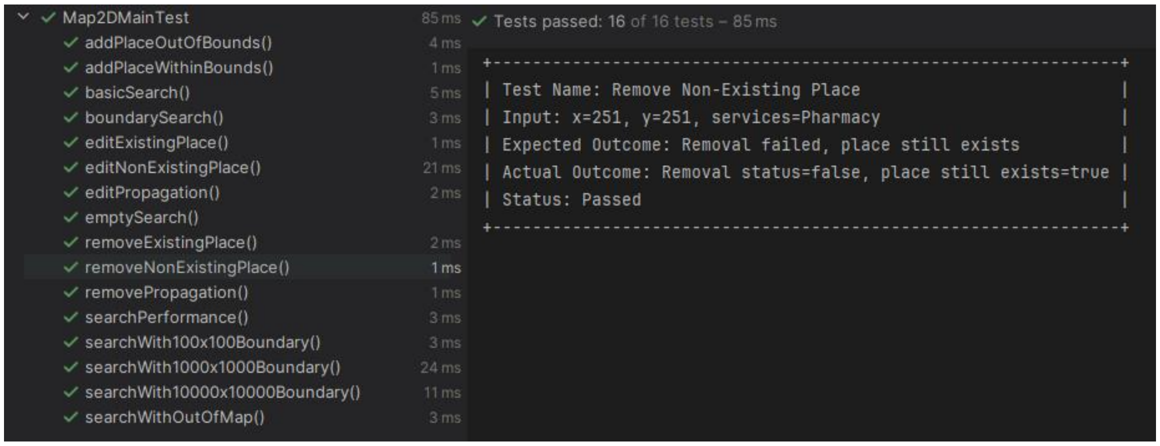
Remove Existing Place: The test demonstrates that the method correctly removes an existing place. The method removePlace() accurately identifies and removes the target place, verified by a subsequent search that returns zero results, confirming the place's removal.



```
✓ Map2DMainTest                         85 ms   ✓ Tests passed: 16 of 16 tests – 85 ms
  ✓ addPlaceOutOfBounds()                4 ms
  ✓ addPlaceWithinBounds()               1 ms   +----------------------------------------------+
  ✓ basicSearch()                        5 ms   | Test Name: Remove Existing Place             |
  ✓ boundarySearch()                     3 ms   | Input: x=250, y=250, services=Pharmacy       |
  ✓ editExistingPlace()                  1 ms   | Expected Outcome: Removal successful, size=0 |
  ✓ editNonExistingPlace()              21 ms   | Actual Outcome: Removal status=true, size=0  |
  ✓ editPropagation()                    2 ms   | Status: Passed                               |
  ✓ emptySearch()                               +----------------------------------------------+
  ✓ removeExistingPlace()                2 ms
  ✓ removeNonExistingPlace()             1 ms
  ✓ removePropagation()                  1 ms
  ✓ searchPerformance()                  3 ms
  ✓ searchWith100x100Boundary()          3 ms
  ✓ searchWith1000x1000Boundary()       24 ms
  ✓ searchWith10000x10000Boundary()     11 ms
  ✓ searchWithOutOfMap()                 3 ms
```

Figure 17: Remove Existing Place Test Case Result

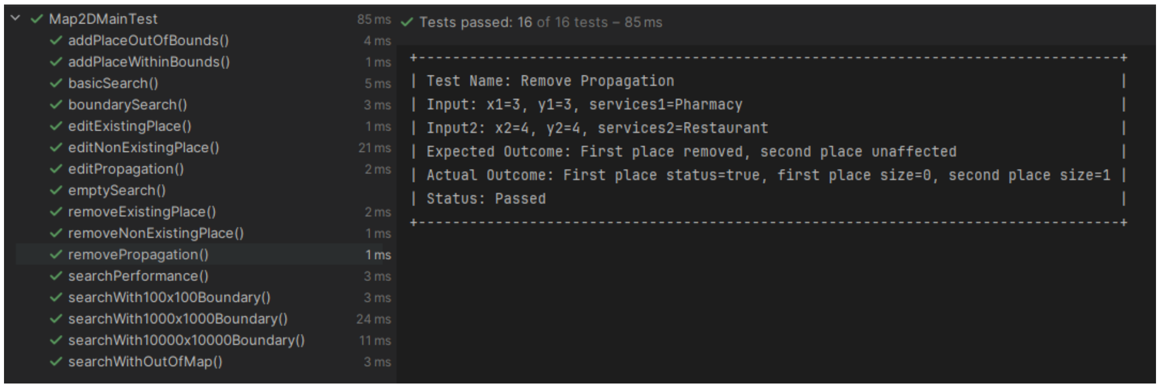
Remove Non-Existing Place: This test verifies the method's handling of attempts to remove a place that does not exist. The method returns false, indicating no place was removed, which is correct behavior as confirmed by a subsequent search showing the originally added place remains.



```
Map2DMainTest                          85 ms   Tests passed: 16 of 16 tests – 85 ms
  addPlaceOutOfBounds()                 4 ms
  addPlaceWithinBounds()                1 ms   +------------------------------------------------------------+
  basicSearch()                         5 ms   | Test Name: Remove Non-Existing Place                       |
  boundarySearch()                      3 ms   | Input: x=251, y=251, services=Pharmacy                     |
  editExistingPlace()                   1 ms   | Expected Outcome: Removal failed, place still exists       |
  editNonExistingPlace()               21 ms   | Actual Outcome: Removal status=false, place still exists=true |
  editPropagation()                     2 ms   | Status: Passed                                             |
  emptySearch()                                +------------------------------------------------------------+
  removeExistingPlace()                 2 ms
  removeNonExistingPlace()              1 ms
  removePropagation()                   1 ms
  searchPerformance()                   3 ms
  searchWith100x100Boundary()           3 ms
  searchWith1000x1000Boundary()        24 ms
  searchWith10000x10000Boundary()      11 ms
  searchWithOutOfMap()                  3 ms
```

Figure 18: Remove Non-Existing Place Test Case Result

Remove Propagation: This test assesses the method's ability to remove a specified place without affecting adjacent data. The successful removal of the first place without impacting the second confirms that the removePlace() method does not erroneously modify or delete adjacent or unrelated entries.



```
Map2DMainTest                          85 ms   Tests passed: 16 of 16 tests – 85 ms
  addPlaceOutOfBounds()                 4 ms
  addPlaceWithinBounds()                1 ms   +------------------------------------------------------------+
  basicSearch()                         5 ms   | Test Name: Remove Propagation                              |
  boundarySearch()                      3 ms   | Input: x1=3, y1=3, services1=Pharmacy                      |
  editExistingPlace()                   1 ms   | Input2: x2=4, y2=4, services2=Restaurant                   |
  editNonExistingPlace()               21 ms   | Expected Outcome: First place removed, second place unaffected |
  editPropagation()                     2 ms   | Actual Outcome: First place status=true, first place size=0, second place size=1 |
  emptySearch()                                | Status: Passed                                             |
  removeExistingPlace()                 2 ms   +------------------------------------------------------------+
  removeNonExistingPlace()              1 ms
  removePropagation()                   1 ms
  searchPerformance()                   3 ms
  searchWith100x100Boundary()           3 ms
  searchWith1000x1000Boundary()        24 ms
  searchWith10000x10000Boundary()      11 ms
  searchWithOutOfMap()                  3 ms
```

Figure 19: Remove Propagation Test Case Result

<u>Boundary Handling and Error Management</u>:
All tests demonstrate that the method correctly handles typical scenarios (removing existing and non-existing places) and more complex scenarios involving multiple data points. The method's ability to correctly identify and modify the internal data structure while preserving data integrity is crucial.

5.3.2. Efficiency

<u>Performance of Data Manipulation</u>:
The algorithm's efficiency in removing a place hinges on its ability to quickly locate the place (using getSuitableLeaf() for structured data or direct iteration for unstructured data) and efficiently manage the array that stores places.

The loop used to shift remaining places after removal suggests a linear time complexity relative to the number of places (O(n)), which is efficient for a smaller dataset but might become a bottleneck as the number of places grows, especially in dense regions of the map.

Memory Management:
The removal operation does not dynamically allocate or deallocate significant memory; it merely adjusts existing structures. This minimal memory footprint during deletion is efficient and reduces potential memory fragmentation issues.

Scalability Concerns:
While the current implementation is adequate for small to moderate data sets, scalability could be an issue with larger data sets due to the linear search and shift operations. Consideration of more complex but efficient data structures like balanced trees or spatial indices might be warranted for large-scale applications.

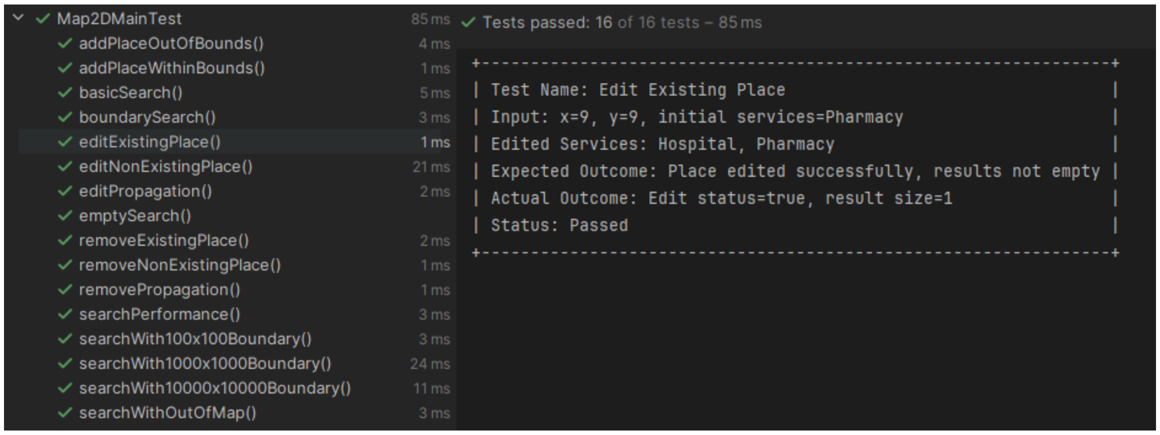
5.3.3. Conclusion

The tests for the removePlace() method are thorough and demonstrate the operation's correctness under various conditions. The method shows good performance with small datasets due to its simple and direct approach to data manipulation. However, as the size of the data increases, the performance might degrade, suggesting potential areas for optimization such as refining data structures or employing more sophisticated algorithms to manage spatial data more efficiently. This careful consideration of both correctness and performance highlights the method's current strengths and future improvement opportunities.

5.4. Edit Place Testing & Evaluation

5.4.1. Correctness

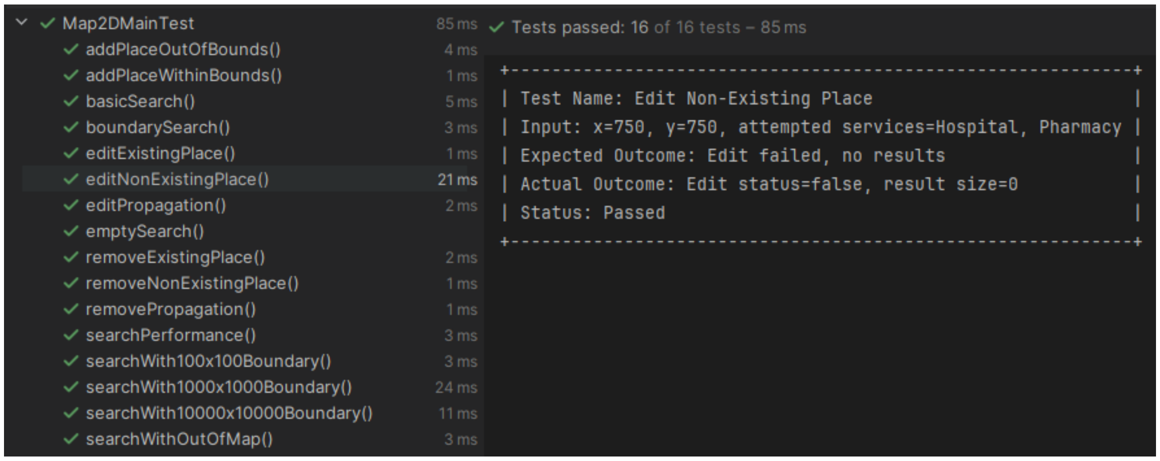
Specificity and Accuracy:
Edit Existing Place: This test verifies that an existing place can be successfully edited to update its services. The test confirms the editing by searching for the newly added services at the same coordinates, ensuring that the editing operation both modifies and retains the place correctly.



```
Map2DMainTest                      85 ms    ✓ Tests passed: 16 of 16 tests – 85 ms
  ✓ addPlaceOutOfBounds()           4 ms
  ✓ addPlaceWithinBounds()          1 ms    +--------------------------------------------------------------+
  ✓ basicSearch()                   5 ms    | Test Name: Edit Existing Place                               |
  ✓ boundarySearch()                3 ms    | Input: x=9, y=9, initial services=Pharmacy                   |
  ✓ editExistingPlace()             1 ms    | Edited Services: Hospital, Pharmacy                          |
  ✓ editNonExistingPlace()         21 ms    | Expected Outcome: Place edited successfully, results not empty |
  ✓ editPropagation()               2 ms    | Actual Outcome: Edit status=true, result size=1              |
  ✓ emptySearch()                           | Status: Passed                                               |
  ✓ removeExistingPlace()           2 ms    +--------------------------------------------------------------+
  ✓ removeNonExistingPlace()        1 ms
  ✓ removePropagation()             1 ms
  ✓ searchPerformance()             3 ms
  ✓ searchWith100x100Boundary()     3 ms
  ✓ searchWith1000x1000Boundary()  24 ms
  ✓ searchWith10000x10000Boundary() 11 ms
  ✓ searchWithOutOfMap()            3 ms
```

Figure 20: Edit Existing Place Test Case Result

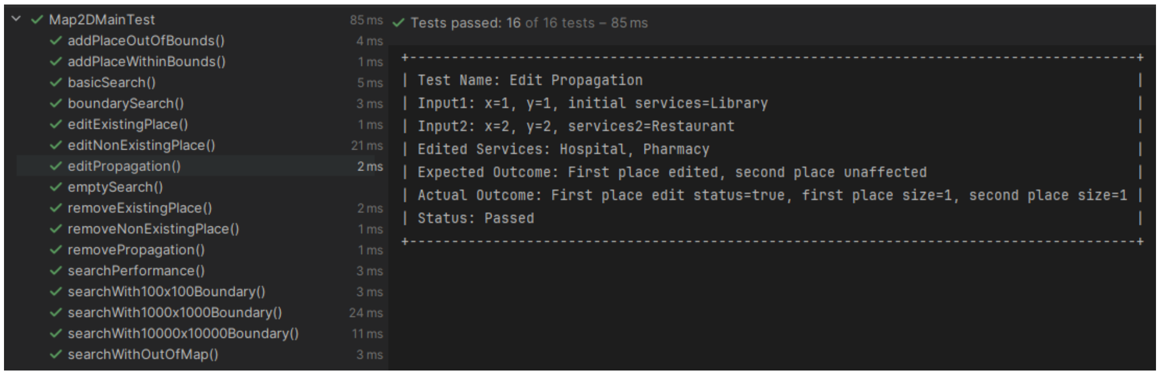
Edit Non-Existing Place: This test checks the method's response when attempting to edit a place that does not exist. The expected behavior is for the method to return false, which it does correctly, and the subsequent search confirms that no new place was erroneously created.



```
Map2DMainTest                      85 ms    ✓ Tests passed: 16 of 16 tests – 85 ms
  ✓ addPlaceOutOfBounds()           4 ms
  ✓ addPlaceWithinBounds()          1 ms    +--------------------------------------------------------------+
  ✓ basicSearch()                   5 ms    | Test Name: Edit Non-Existing Place                           |
  ✓ boundarySearch()                3 ms    | Input: x=750, y=750, attempted services=Hospital, Pharmacy |
  ✓ editExistingPlace()             1 ms    | Expected Outcome: Edit failed, no results                    |
  ✓ editNonExistingPlace()         21 ms    | Actual Outcome: Edit status=false, result size=0             |
  ✓ editPropagation()               2 ms    | Status: Passed                                               |
  ✓ emptySearch()                           +--------------------------------------------------------------+
  ✓ removeExistingPlace()           2 ms
  ✓ removeNonExistingPlace()        1 ms
  ✓ removePropagation()             1 ms
  ✓ searchPerformance()             3 ms
  ✓ searchWith100x100Boundary()     3 ms
  ✓ searchWith1000x1000Boundary()  24 ms
  ✓ searchWith10000x10000Boundary() 11 ms
  ✓ searchWithOutOfMap()            3 ms
```

Figure 21: Edit Non-Existing Place Test Case Result

Edit Propagation: This test assesses whether editing one place affects another, ensuring that edits are localized to the intended target without cascading changes. The results show that the edited place reflects new services while the neighboring place remains unchanged, which is the desired outcome.



```
Map2DMainTest                          85 ms   ✓ Tests passed: 16 of 16 tests – 85 ms
  ✓ addPlaceOutOfBounds()               4 ms
  ✓ addPlaceWithinBounds()              1 ms
  ✓ basicSearch()                       5 ms
  ✓ boundarySearch()                    3 ms
  ✓ editExistingPlace()                 1 ms
  ✓ editNonExistingPlace()             21 ms
  ✓ editPropagation()                   2 ms
  ✓ emptySearch()
  ✓ removeExistingPlace()               2 ms
  ✓ removeNonExistingPlace()            1 ms
  ✓ removePropagation()                 1 ms
  ✓ searchPerformance()                 3 ms
  ✓ searchWith100x100Boundary()         3 ms
  ✓ searchWith1000x1000Boundary()      24 ms
  ✓ searchWith10000x10000Boundary()    11 ms
  ✓ searchWithOutOfMap()                3 ms
```

```
+-----------------------------------------------------------------------------------------+
| Test Name: Edit Propagation                                                             |
| Input1: x=1, y=1, initial services=Library                                              |
| Input2: x=2, y=2, services2=Restaurant                                                  |
| Edited Services: Hospital, Pharmacy                                                     |
| Expected Outcome: First place edited, second place unaffected                           |
| Actual Outcome: First place edit status=true, first place size=1, second place size=1 |
| Status: Passed                                                                          |
+-----------------------------------------------------------------------------------------+
```

Figure 22: Edit Propagation Test Case Result

Robustness and Error Handling:
The method and tests demonstrate robust handling of typical scenarios, including edits to existing places and non-existing places, ensuring that operations on the data structure do not introduce errors or unintended changes elsewhere.

5.4.2. Efficiency

Performance of Data Manipulation:
The editPlace() method's efficiency depends on how quickly it can locate the target place and update its details. The method employs a direct iteration for unstructured data or a leaf-oriented search for structured data, which is efficient for smaller datasets but may not scale linearly with increasing data size.

The operation's complexity is generally linear in the number of places due to the iteration over numOfPlaces. This might necessitate optimization, such as employing more sophisticated search techniques or indexing strategies for larger datasets.

Resource Utilization:
Memory usage remains constant during an edit operation, as it merely modifies existing entries rather than reallocating or extensively manipulating the data structure. This characteristic is beneficial for operations that require frequent updates, minimizing the overhead associated with memory management.

Scalability and Future Considerations:
While the method is adequate for small to moderate datasets, its reliance on linear search mechanisms may lead to performance bottlenecks as the dataset grows. Consideration for future improvements might include the use of spatial indexing methods (e.g., R-trees or Kd-trees) that can significantly reduce search and update times in spatial databases.

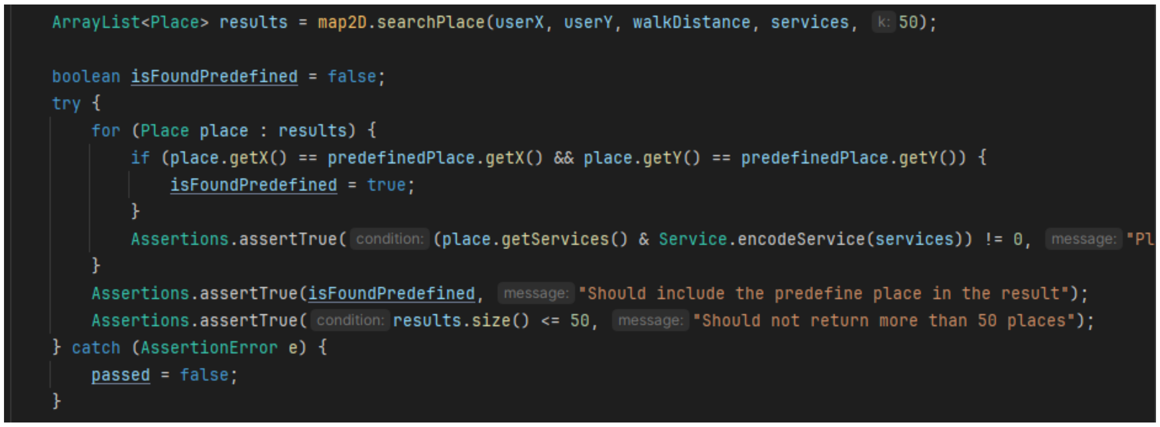
5.4.3. Conclusion

The test suite effectively measures the editPlace() operation's correctness under varied scenarios, demonstrating its ability to handle typical and edge cases with precision. The method is efficient for current use cases but may require enhancements to maintain performance with larger datasets. This thorough evaluation not only confirms the operation's current effectiveness but also identifies potential areas for future optimizations to accommodate growth and more complex scenarios.

5.5.Search Place Testing & Evaluation

5.5.1. Correctness

Procedure of Experiments:



```java
ArrayList<Place> results = map2D.searchPlace(userX, userY, walkDistance, services, k: 50);

boolean isFoundPredefined = false;
try {
    for (Place place : results) {
        if (place.getX() == predefinedPlace.getX() && place.getY() == predefinedPlace.getY()) {
            isFoundPredefined = true;
        }
        Assertions.assertTrue( condition: (place.getServices() & Service.encodeService(services)) != 0,  message: "Pl
    }
    Assertions.assertTrue(isFoundPredefined,  message: "Should include the predefine place in the result");
    Assertions.assertTrue( condition: results.size() <= 50,  message: "Should not return more than 50 places");
} catch (AssertionError e) {
    passed = false;
}
```

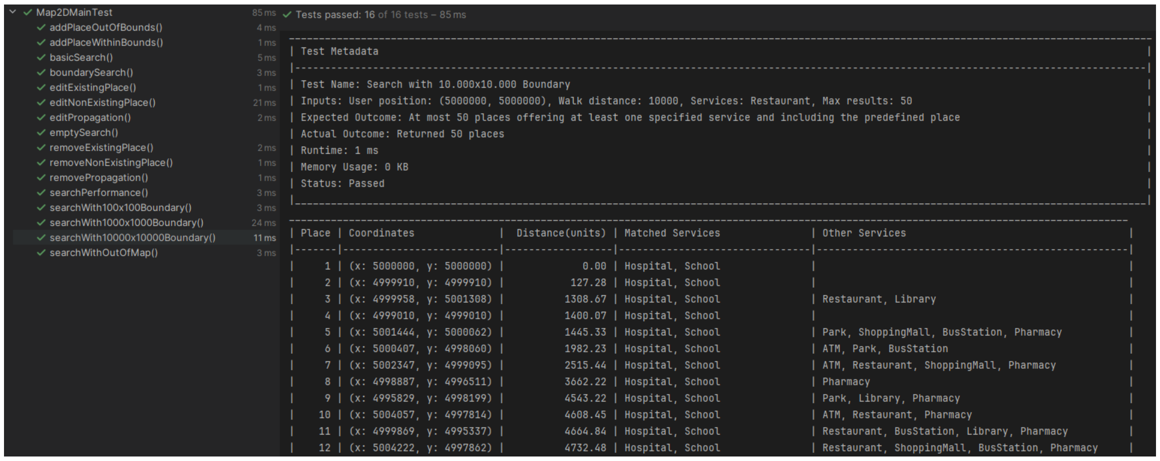
Figure 23: Testing approach for searchPlace() method

To initialize testing for the searchPlace() method, the setUpOnce() method is called with the purpose of initializing test data for all the test functions. To be more specific, it creates a 10,000,000 x 10,000,000 map and utilizes the help of the Random class to create 100,000,000 places randomly with random services, this was chosen this size to make sure all the search tests would work perfectly within the large data set. To ensure the correctness of the test function, our approach is to manually create some places within the search bounding rectangle and validate the results whether it contains the predefined places or not. To implement this approach the Assertion package from the Junit library is extremely important when it provides the ability to validate the output with our expected return value through its methods such as assertEquals(), assertTrue(), and assertFalse(). For example, figure 23 indicates that in the test function, we apply the assertTrue() function to check whether the predefined place is in the results list or not, check whether the size of the results exceeds 50 places or not, and check the services of each place are matched the searching services or not.

Test Cases:

For the searchPlace() method, the correctness is evaluated through a variety of suitable test cases designed to validate both typical and edge-case. This maintains the performance of the function as expected over scenarios.

Name	Description	Expected Output
testSearchWith 100x100Boundary	Test the function with a variety of bounding rectangles to check the accuracy to ensure it will deliver high performance in all cases.	The list of places that intersect with the bounding rectangle and the list has to contain the predefined place for correctness testing purposes.
testSearchWith 1000x1000Boundary		
testSearchWith 10000x10000Boundary		
testSearchWith 50000x50000Boundary		
BasicSearch		
testSearchWithOutOfMap	Ensures that the function will handle edge cases and that the expected return value is zero.	No results should be returned.
testSearchPerformance	Ensuring that the performance of the function is within acceptable limits.	The duration of the search method should not exceed 200ms.
testFilteringBySingleService	Check the correctness and performance of the filtering services function of the search method.	All results should contain the appropriate service.



```
Map2DMainTest                              85 ms
  addPlaceOutOfBounds()                     4 ms
  addPlaceWithinBounds()                    1 ms
  basicSearch()                             5 ms
  boundarySearch()                          3 ms
  editExistingPlace()                       1 ms
  editNonExistingPlace()                   21 ms
  editPropagation()                         2 ms
  emptySearch()
  removeExistingPlace()                     2 ms
  removeNonExistingPlace()                  1 ms
  removePropagation()                       1 ms
  searchPerformance()                       3 ms
  searchWith100x100Boundary()               3 ms
  searchWith1000x1000Boundary()            24 ms
  searchWith10000x10000Boundary()          11 ms
  searchWithOutOfMap()                      3 ms
```

```
Tests passed: 16 of 16 tests – 85 ms

------------------------------------------------------------------------------------------------------
| Test Metadata                                                                                      |
|----------------------------------------------------------------------------------------------------|
| Test Name: Search with 10.000x10.000 Boundary                                                      |
| Inputs: User position: (5000000, 5000000), Walk distance: 10000, Services: Restaurant, Max results: 50 |
| Expected Outcome: At most 50 places offering at least one specified service and including the predefined place |
| Actual Outcome: Returned 50 places                                                                 |
| Runtime: 1 ms                                                                                      |
| Memory Usage: 0 KB                                                                                 |
| Status: Passed                                                                                     |
|____________________________________________________________________________________________________|

----------------------------------------------------------------------------------------------------
| Place | Coordinates            |  Distance(units) | Matched Services          | Other Services                              |
|-------|------------------------|------------------|---------------------------|---------------------------------------------|
|     1 | (x: 5000000, y: 5000000) |             0.00 | Hospital, School          |                                             |
|     2 | (x: 4999910, y: 4999910) |           127.28 | Hospital, School          |                                             |
|     3 | (x: 4999958, y: 5001308) |          1308.67 | Hospital, School          | Restaurant, Library                         |
|     4 | (x: 4999010, y: 4999010) |          1400.07 | Hospital, School          |                                             |
|     5 | (x: 5001444, y: 5000062) |          1445.33 | Hospital, School          | Park, ShoppingMall, BusStation, Pharmacy    |
|     6 | (x: 5000407, y: 4998060) |          1982.23 | Hospital, School          | ATM, Park, BusStation                       |
|     7 | (x: 5002347, y: 4999095) |          2515.44 | Hospital, School          | ATM, Restaurant, ShoppingMall, Pharmacy     |
|     8 | (x: 4998887, y: 4996511) |          3662.22 | Hospital, School          | Pharmacy                                    |
|     9 | (x: 4995829, y: 4998199) |          4543.22 | Hospital, School          | Park, Library, Pharmacy                     |
|    10 | (x: 5004057, y: 4997814) |          4608.45 | Hospital, School          | ATM, Restaurant, Pharmacy                   |
|    11 | (x: 4999869, y: 4995337) |          4664.84 | Hospital, School          | Restaurant, BusStation, Library, Pharmacy   |
|    12 | (x: 5004222, y: 4997862) |          4732.48 | Hospital, School          | Restaurant, ShoppingMall, BusStation, Pharmacy |
```

Figure 24: Example of the Search Place Test Case Result

Figure 24 above indicates how we check the performance of search methods, edge case cover, various boundary searching, and service filtering. All tests finish successfully and provide essential information for further analysis.

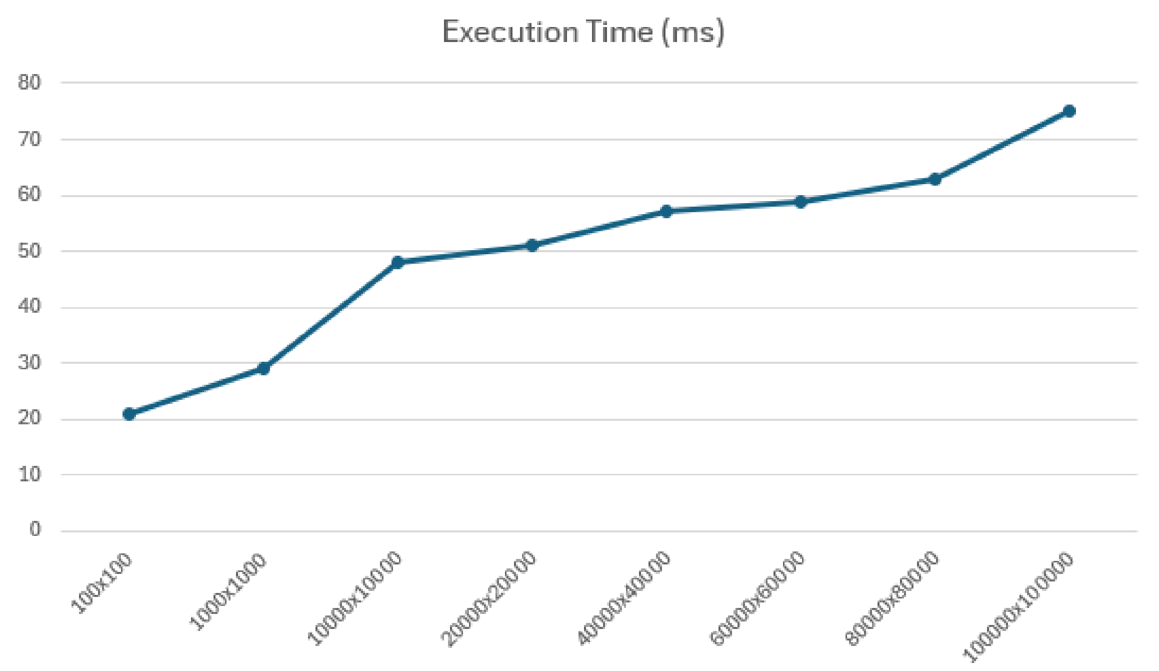
Conclusion:

Our search methods for finding all of the related places closest to the user were carefully examined and have been proven to be reliable and accurate. Furthermore, using integration testing to examine multiple aspects of the function, we ensure that the searching method effectively handles edge cases and various bounding rectangles from smallest to biggest.

5.5.2. Efficiency

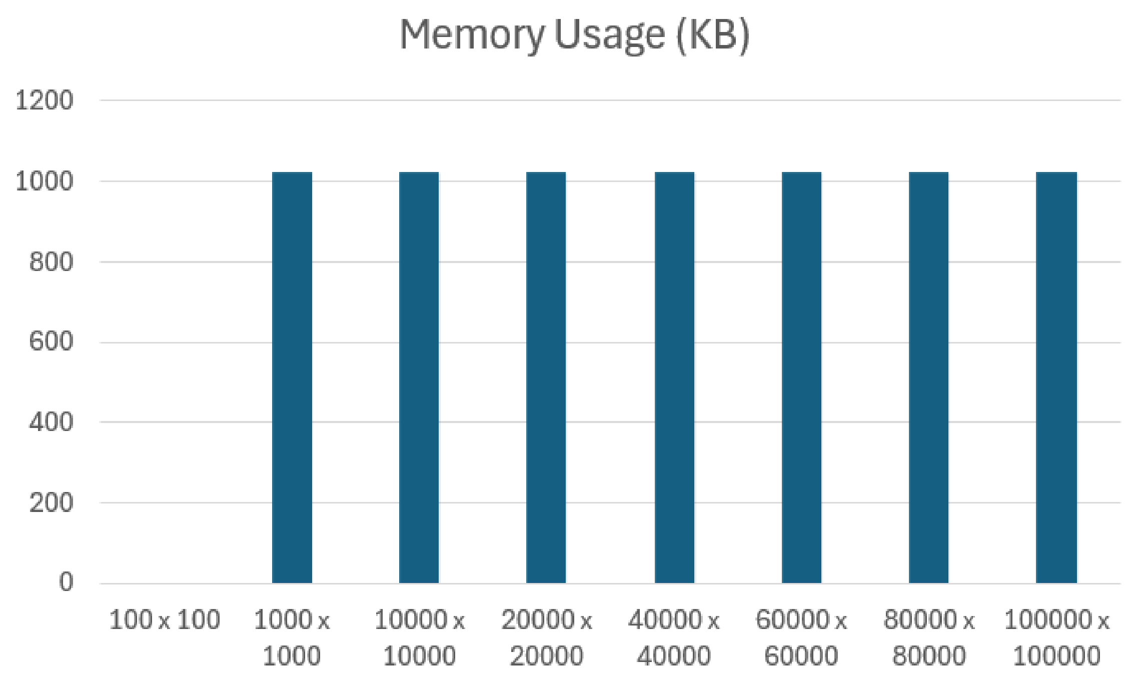
The efficiency of the searchPlace() method can be measured through two main factors: Execution Time and Memory Usage.

Execution Time Analysis:



The line graph above indicates a clear trend when the size of the bounding rectangle increases. This increase is quite linear, which fits well with the expectation that the complexity of searching in wide areas where more places need to be evaluated. The slow rise from 30ms to approximately 70ms for the largest size tested shows that the method handles scaling reasonably well but does show an increase in time with a larger data set.

<u>Memory Usage Analysis</u>:



The graph above shows that memory usage stays constant through different sizes of the bounding rectangle. All test cases take about 1024KB of memory except the minimum test case is 100 x 100 with the consumption of 0KB, so the memory usage depends on the size of the map and the number of places rather than the size of the searching rectangle. This shows efficient memory management within the search method, where increasing the size only costs time, not memory usage.

<u>Conclusion</u>:
Our search methods for finding all the related places closest to the user were carefully examined and have been proven to be reliable and accurate. Furthermore, using integration testing to examine multiple aspects of the function, we ensure that the searching method effectively handles edge cases and various bounding rectangles from smallest to biggest.

6. CONCLUSION

From the technical description and problem statement provided, the decision to employ a QuadTree alongside an Array for managing Points of Interest (POIs) is shown to be effective for the scenario at hand. This implementation within the Map2D framework presents several advantages and disadvantages.

Advantages:

- Utilizing an Array for storing place information leverages the simplicity and memory efficiency of primitive data types.
- The incorporation of a spatial data structure like the QuadTree significantly enhances the management of geographical locations, facilitating much faster location processing.

Disadvantages and Future Improvements:

- A notable concern arises when dealing with larger datasets, particularly those exceeding 100 million places. In such cases, the use of an Array can substantially impair system performance, primarily due to the need to redistribute the array once its maximum capacity is reached.
- To address this limitation, we are considering the adoption of spatially oriented data structures such as Kd-Trees or R-Trees. These alternatives offer superior efficiency, with a time complexity of O(log n), compared to the Array's O(n). This makes them more suitable for managing extensive spatial data.
- The initial decision against implementing KD Trees stemmed from constraints related to using a backup database. Currently, the ADT relies entirely on the machine's heap memory, which is not optimal for a data structure that generates a high number of objects. Although preliminary trials with a KD Tree demonstrated stable and efficient performance—managing to compile and insert approximately 100 million places—it required over 6GB of heap memory. Such a trade-off was deemed excessive for the project's scope.

In conclusion, while the current implementation using QuadTree and Array offers significant benefits in terms of speed and memory efficiency, future developments of the Map2D ADT should explore more advanced spatial data structures to better handle larger datasets without compromising memory usage and performance.

7. REFERENCES

[1] *"Quad Tree"*, GeeksforGeeks, Jun. 12, 2017. https://www.geeksforgeeks.org/quad-tree/

[2] "*Quadtree"*, Wikipedia, Nov. 06, 2019. https://en.wikipedia.org/wiki/Quadtree