COSC2658_ Data Structures and Algorithms

Technical Report

Group 04 - Semester 1 (2024A)

1. Pham Phuoc Sang - s3975979
2. Cao Nguyen Hai Linh - s3978387
3. Tran Pham Khanh Doan - s3978798
4. Nguyen Ngoc Thanh Mai - s3978486

Date of submission: May 9th, 2024

Table Of Contents

1. Introduction

1.1. Problem Statement

[1] Despite advancements in geographic information systems (GIS), managing, handling, and querying vast spatial data remains challenging. Traditional approaches, like statistical spatial analysis, struggle with the complex spatial relationships inherent to such data, particularly in dynamically retrieving Points of Interest (POI) across large datasets within tightly defined geographic boundaries. Those given limitations highlight the need for more sophisticated methods that can efficiently handle limitations and improve the performance of GIS.

1.2 Abstract

This report presents a quad-tree-based solution developed after extensive research and iterative testing to overcome limitations. By leveraging quadtrees' data structural efficiency, the application significantly enhances search speed and accuracy while ensuring scalability and robustness for managing up to 100 million POIs on a 10 million by 10 million unit map.

2. Overview and High level design

2.1. Overview

This project aims to develop a Java application to optimize the search for point-of-interest (POI) within a specified spatial context. The application will utilize a bounding rectangle, defined by its middle x-y coordinates, width, and height, along with a particular service type, to streamline locating various facilities such as ATMs, restaurants, hotels, hospitals, coffees, and gas stations. The optimization is critical because the application must manage up to 100 million points within a map size of 10 million by 10 million units, making an efficient and effective searching function imperative. Additionally, the project concentrates on leveraging the power of the chosen data structure to enhance the speed and accuracy of search results based on the user-defined area. This application not only addresses the challenges of managing large-scale data but also meets the practical needs of users in navigating and utilizing geographic information systems. The core functionalities of the application include adding, removing, searching, and editing places within the map.

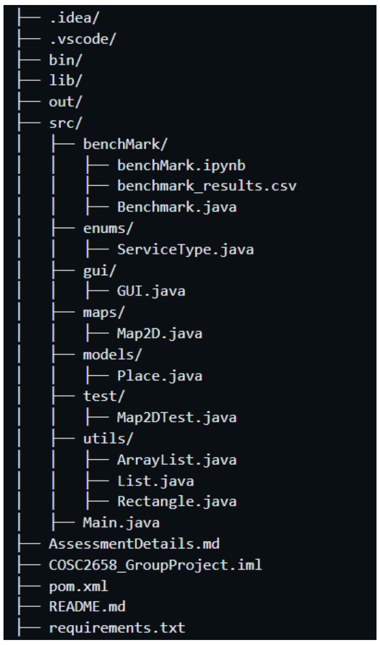
Core Features	Description
Add a place (Insert a place)	Users can add a new place to the map. The place is defined by its coordinates (x, y) and can be associated with one or multiple types of service.
Remove a place	Users are allowed to remove an outdated place from the map. This functionality is

	essential for keeping the map data up to date and accurate, removing outdated or incorrect information as needed.
Search a place	This is another core feature within the application, which allows users to search for K maximum places regardless of the user-defined bounding rectangle. This search can be filtered by service type, allowing users to quickly find specific services within a designated area.
Edit a place (Add or Remove service types)	This feature allows users to edit a service type within a place, including removing or adding more service types. Modification is restricted to the services provided by place (able to be changed among six types of service); the geographical coordinates (x, y) remain immutable once set. This ensures data integrity while allowing flexibility in the service data.

Table 1: Core Features

2.2 High level design

2.2.1. Folder Structure



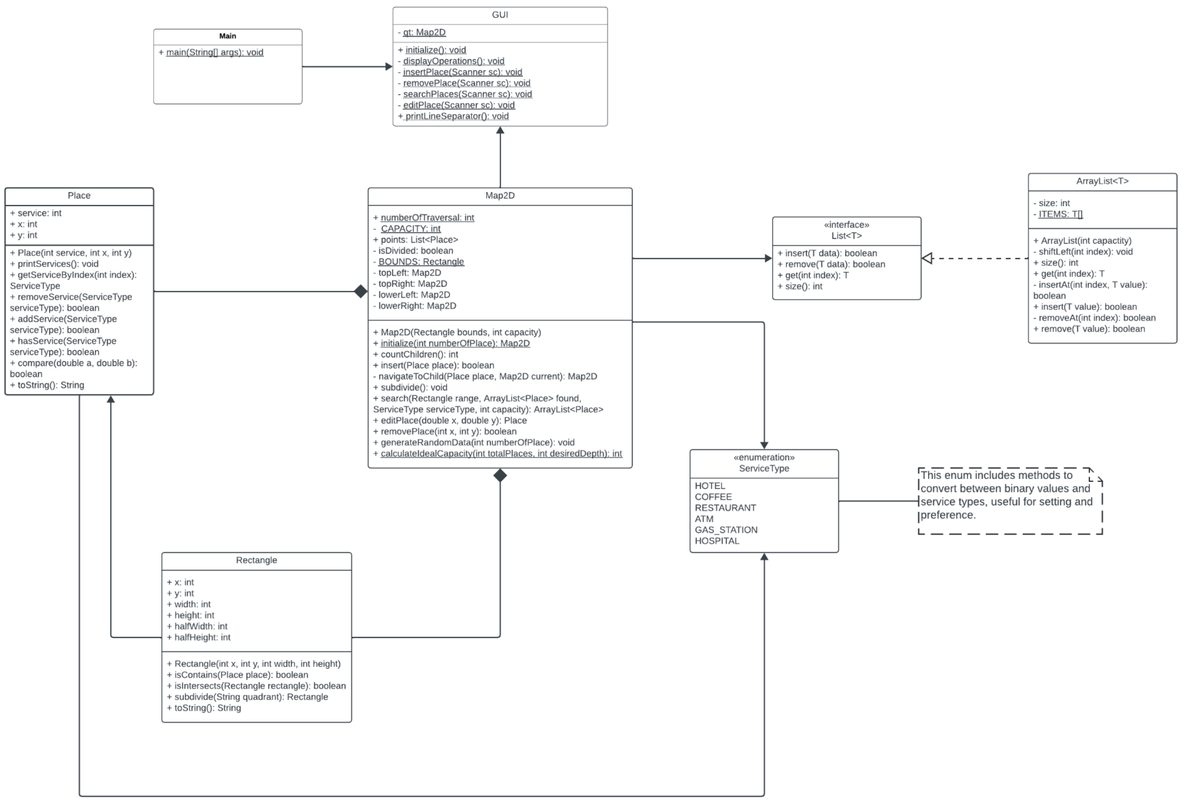
```
├── .idea/
├── .vscode/
├── bin/
├── lib/
├── out/
├── src/
│   ├── benchMark/
│   │   ├── benchMark.ipynb
│   │   ├── benchmark_results.csv
│   │   ├── Benchmark.java
│   ├── enums/
│   │   ├── ServiceType.java
│   ├── gui/
│   │   ├── GUI.java
│   ├── maps/
│   │   ├── Map2D.java
│   ├── models/
│   │   ├── Place.java
│   ├── test/
│   │   ├── Map2DTest.java
│   ├── utils/
│   │   ├── ArrayList.java
│   │   ├── List.java
│   │   ├── Rectangle.java
│   ├── Main.java
├── AssessmentDetails.md
├── COSC2658_GroupProject.iml
├── pom.xml
├── README.md
├── requirements.txt
```

Figure 1: Folder Structure

The project follows the standard Maven Java project structure, with additional configurations enabling it to run on both VScode and IntelliJ IDEA. Additionally, a benchmark folder contains the necessary files for evaluating the application's performance, including the `benchMark.ipynb` file, which visualizes the benchmark results from the `benchmark_results.csv` file. The `Benchmark.java` file runs the application's benchmark, while the `test/` folder contains the Java file responsible for unit testing.

Finally, in the \`src/\` folder, more folders with Java files run the application, with \`Main.java\` as the critical entry point. Running this file displays the user interface in the terminal for easy interaction.

2.2.2. Class Structure



Main
+ main(String[] args): void
GUI
- qt: Map2D
+ initialize(): void
- displayOperations(): void
- insertPlace(Scanner sc): void
- removePlace(Scanner sc): void
- searchPlaces(Scanner sc): void
- editPlace(Scanner sc): void
+ printLineSeparator(): void
Place
+ service: int
+ x: int
+ y: int
+ Place(int service, int x, int y)
+ printServices(): void
+ getServiceByIndex(int index): ServiceType
+ removeService(ServiceType serviceType): boolean
+ addService(ServiceType serviceType): boolean
+ hasService(ServiceType serviceType): boolean
+ compare(double a, double b): boolean
+ toString(): String
Map2D
+ numberOfTraversal: int
- CAPACITY: int
+ points: List<Place>
- isDivided: boolean
- BOUNDS: Rectangle
- topLeft: Map2D
- topRight: Map2D
- lowerLeft: Map2D
- lowerRight: Map2D
+ Map2D(Rectangle bounds, int capacity)
+ initialize(int numberOfPlace): Map2D
+ countChildren(): int
+ insert(Place place): boolean
- navigateToChild(Place place, Map2D current): Map2D
+ subdivide(): void
+ search(Rectangle range, ArrayList<Place> found, ServiceType serviceType, int capacity): ArrayList<Place>
+ editPlace(double x, double y): Place
+ removePlace(int x, int y): boolean
+ generateRandomData(int numberOfPlace): void
+ calculateIdealCapacity(int totalPlaces, int desiredDepth): int
«interface»
List<T>
+ insert(T data): boolean
+ remove(T data): boolean
+ get(int index): T
+ size(): int
ArrayList<T>
- size: int
- ITEMS: T[]
+ ArrayList(int capactity)
- shiftLeft(int index): void
+ size(): int
+ get(int index): T
- insertAt(int index, T value): boolean
+ insert(T value): boolean
- removeAt(int index): boolean
+ remove(T value): boolean
«enumeration»
ServiceType
HOTEL
COFFEE
RESTAURANT
ATM
GAS_STATION
HOSPITAL
This enum includes methods to convert between binary values and service types, useful for setting and preference.
Rectangle
+ x: int
+ y: int
+ width: int
+ height: int
+ halfWidth: int
+ halfHeight: int
+ Rectangle(int x, int y, int width, int height)
+ isContains(Place place): boolean
+ isIntersects(Rectangle rectangle): boolean
+ subdivide(String quadrant): Rectangle
+ toString(): String

Figure 2: Class Diagram

"Map2D" is our system's central class, encompassing core functionalities essential to the application's operation. Both "Rectangle" and "Place" are tightly integrated into "Map2D" through composition, indicating that they cannot exist independently outside of "Map2D". Furthermore, there is an association between "Place" and "Rectangle" in managing spatial data effectively. The "ServiveType" enumeration is utilized by both "Map2D" and "Place" to categorize provided service types at various points of interest. The "List" interface, which is implemented by the "ArrayList," is associated with "Map2D" to handle dynamic collections of place efficiently. Lastly, the "GUI" class interfaces with the "Main" class to trigger method invocations, thus linking the user interface with the system's main functionalities, facilitating user interactions, and displaying the system's state.

2.2.3 Java classes, methods, software design patterns

Map2D: This map represents a quad-tree data structure that the team implemented to enhance the efficiency of spatial indexing and query 2D points. It supports core operations such as insertion, search, edit, and removal of places within a defined 2D space. Each quad-tree node can be a leaf node with a list of places or an internal node with four children nodes. This structure is beneficial for applications that require spatial searches, such as maps and simulation systems.

Place: This class represents a geographical location with associated services and manages services at a specific coordinate (x, y). It includes methods for adding, removing, and checking services using a binary representation.

Rectangle: This class represents a rectangle in 2D space and includes methods to check containment, intersection, and subdivision. It is used for spatial calculations and quad-tree implementations.

Service Type: An enumeration that represents different services offered at a location. It encapsulates the core functionality for managing service types in a spatial context, including retrieving service types by index, generating random service combinations, converting between service lists and binary representation, and listing all service types. This enum is critical for systems requiring efficient and flexible management of categorized service data in geographic information systems (GIS), particularly when services at various locations must be dynamically queried or updated.

List: The "List" interface serves as a container that specifies the behavior of a class by providing an abstract type. It defines four methods—insert, remove, get, and size—without implementations, providing a template for list behaviour.

ArrayList: This is a generic ArrayList implementation that dynamically manages a collection of elements. This class implements the "List" interface and overrides the methods to manipulate the size and elements of the list, including adding, removing, and accessing elements.

GUI: This class represents the Graphic User Interface for managing a quad-tree of places. Its purpose is to avoid the complicated and challenging learning of command-line interfaces so users can easily interact with and understand the system. Within the application, it allows users to insert, remove, search, and edit places in the quad-tree through a console-based menu system.

Main: It is used to initiate the execution of a Java program.

3. Data Structure and Algorithms

3.1 DSA main approach to solve problem: QuadTree Introduction

Quadtrees are hierarchical data structures that recursively subdivide a two-dimensional space into four quadrants or regions. They are particularly well-suited for representing and efficiently querying spatial data, making them an ideal choice for approaching the Points of

Interest (POI) [2].
The key advantage of using a quadtree for this problem is its ability to efficiently handle spatial queries, such as retrieving all POIs within a given bounding rectangle. This is achieved by recursively subdividing the space into quadrants, allowing for rapid pruning of irrelevant regions during the search process [3].
Here's an overview of how a quadtree can be used to address the POI Search problem:

1. **Spatial Partitioning:** The entire geographical area is recursively subdivided into quadrants, each representing a specific region. POIs are associated with the smallest quadrant containing them [4].

2. **Bounding Rectangle Query:** The quadtree is traversed recursively to find POIs within a given bounding rectangle. If a quadrant lies entirely within the bounding rectangle, all POIs associated with that quadrant are reported. If a quadrant lies partially within the bounding rectangle, the traversal continues to its child quadrants. Quadrants that lie completely outside the bounding rectangle can be pruned, significantly reducing the search space [5].

3. **Early Termination:** If the number of POIs found exceeds the specified limit (K), the search can be terminated early, as no additional POIs are required [6].

The quadtree's efficiency in this scenario stems from its hierarchical structure and ability to quickly eliminate large portions of the search space irrelevant to the query. This makes it particularly suitable for handling spatial queries involving bounding rectangles [7].

Moreover, quadtrees can be dynamically constructed and updated as new POIs are added or removed, allowing for efficient storage and retrieval of spatial data in real-time applications [8].

3.2 Approach and Selection of Data Structures

One crucial decision in developing our system revolved around selecting the appropriate data structures to efficiently manage and organize spatial data. Below, we discuss our rationale behind choosing quadtree and array lists for our specific circumstances.

3.2.1 Quadtree vs K-D Tree

Quadtree (as mentioned above) and K-D tree are commonly used data structures for spatial indexing and searching tasks. While both structures excel in organizing spatial data, the choice between them depends on various factors, such as the nature of the data, query patterns, and the application's specific requirements.

K-D Tree Overview

A K-D tree, short for k-dimensional tree, is another hierarchical data structure used for partitioning multidimensional space [9]. Unlike quadtree, which subdivides space along axes perpendicular to each other, the K-D tree alternates between different axes, dividing space into hyperplanes at each tree level [9]. K-D trees are commonly used in applications requiring multidimensional indexing and range searching [9].

QuadTree Selection

In our Map2D system, which manages approximately 100,000,000 nodes, we carefully evaluated the quadtree and K-D tree. While both data structures offer spatial partitioning capabilities, the quadtree emerged as the preferred choice for several reasons.

- **Efficient Partitioning of Two-Dimensional Space**: Quadtrees efficiently partition a two-dimensional space by recursively subdividing it into quadrants. This ability lets them rapidly narrow the search area during spatial queries, resulting in faster query performance than K-d trees [10].
- **Depth and Overhead in High-Density Areas**: Quadtrees can handle densely populated areas by dividing the space into equal quadrants and deepening the tree locally without disturbing other parts. On the other hand, k-d trees split along data dimensions and may result in uneven splits in such areas, leading to deeper trees, longer search times, and space for allocating new reference objects.
- **Simplicity and Ease of Implementation:** Quadtree's straightforward recursive subdivision mechanism and intuitive representation make it easier to implement and maintain, particularly in scenarios where rapid development and flexibility are essential [11].
- **Efficient Spatial Queries:** Quadtree's hierarchical structure facilitates efficient spatial indexing and querying operations, enabling faster nearest-neighbor searches and range queries, which are fundamental to our Map2D application's functionality [10]. K-d trees, while versatile across multiple dimensions, do not offer significant advantages in two-dimensional space that would outweigh the intuitive and straightforward partitioning of quadtrees.

3.2.2 ArrayList vs LinkedList

When deciding on a data structure for storing nodes within the quadtree, the choice between ArrayList and LinkedList was deliberated based on their respective strengths and weaknesses, particularly considering the scale of approximately 100,000,000 nodes.

In scenarios where the number of elements is known in advance or when random access to elements is frequent, ArrayList performs better due to its contiguous memory allocation and random access capabilities, making it better suited for managing a large number of nodes efficiently, ensuring optimal space utilization and faster access times. In contrast, the inefficiencies associated with LinkedList, such as the need to allocate and manage individual node objects for each element, would result in substantial memory overhead and potentially impact performance. Since our system manages approximately 100,000,000 nodes, efficient random access and memory allocation were paramount considerations.

3.3. Algorithm

a) Map2D

Insert: This method is critical for adding new “Place” objects to the quad-tree. It evaluates the placement of each new point to ensure it falls within the correct spatial bounds. Then, it navigates through the tree to find the appropriate node for this insertion. The method steps are as follows.

- ➢ The insertion starts at the root node ('this')
- ➢ **Boundary Check:** This step invokes the **isContain()** function from the Rectangle class to check whether the "Place" is within the current node's bounds. If it is outside, the insertion method can promptly reject it or navigate to a different child node that might contain it.
- ➢ **Capacity and Division Check:** If the node has not reached its maximum capacity (CAPACITY), the place is added to the node's list of points. If the capacity is reached and the node has not already been divided (isDivided is false), invoke the subdivide() method.
    - ○ **subdivide()**: It splits the node into four smaller quadrants, each represented as a child node ('topLeft,' 'topRight,' 'lowerLeft,' 'lowerRight). The process includes calculating boundaries by halving the width and height of the parent's bounds and creating four new 'Map2D' instances with new boundaries and the same capacity as the parents.
- ➢ **Navigation:** After subdivision or if the node was already divided, the method navigates to the correct child node that should contain the new place based on its coordinates using the navigateToChild() method.
    - ○ **navigateToChild():** This method ensures that each place is inserted into the correct quadrant. It calculates the midpoint of the current node's bounds to determine the relative position (left/right and top/bottom) of the new place. Depending on the position of the place relative to these midpoints, the appropriate child node is selected and returned for further action (like insertion).
- ➢ **Recursive Insertion:** The navigation to the child node is done recursively until the place is successfully inserted or rejected due to spatial constraints.

Search: The method is essential for querying the quad-tree to find "Place" objects within a user-defined boundary (specified rectangular range) and optionally filtering by a service type. The steps involved in the search operation:

- ➢ **List Creation:** If no existing list is found to store query results, the method creates a new empty list (“found”) for collecting matching “Place” objects.
- ➢ **Boundary Check:** The search function first checks the range intersections. By invoking the isIntersects() methods from the "Rectangle" class to check whether the search range intersects with the current node's bounds. If there is no intersection, the

search in this branch terminates early, returning the current found list (or empty if none is found)

  - **isIntersects()**: This method calculates the edges of the current and another rectangle based on their center coordinates and dimensions to determine the positions of their left, right, top, and bottom edges. It then checks for non-overlapping conditions, such as whether the left edge of one rectangle is to the right of the other's right edge (and similar checks for other edges), to conclude if the rectangles do not intersect.

- **Query and Collect:** For each place in the node's list of points, the method filters places by checking if they fall within the specified range and meet any provided service type criteria. Places that meet all criteria are added to a collection list (found). This collection operation continues until the list reaches a specified capacity or all valid places are collected.
- **Recursive search:** If the current node is subdivided ("isDivided" is "true"), the search recurs into each applicable child node (those intersected by the range). Results from each child node are aggregated into the "found" list.
- Once the search through the tree is complete or the list capacity is reached, the list of found places is returned. This list contains all places within the range and matches the service type criteria to the maximum specified capacity.

Remove: This method is designed to remove a "Place" object based on its specific (x, y) coordinates from the quad-tree structure. The method navigates through the tree to locate and remove the targeted place.

- **Initialization:** The "placeToRemove" variable is defined as null to store the identified place once found.
- **Traversal and Identification:** The method then iterates through the list of places stored at the current node, checking if any place's coordinates match the specified (x, y) coordinates by invoking the compare() method within the Place class. If a matching place is found, placeToRemove is updated to reference this place, and the loop breaks to proceed with removal.
- **Removal Process:** Once the "placeToRemove" is not null (a place has been found), the method attempts to remove the place from the list. Upon successful removal, it returns true and prints confirmed messages; otherwise, it returns false.
- **Navigate to child node:** If the current node is subdivided ("isDivided" is "true"), the method invokes the navigateToChild() to determine which child node may contain the place, using a dummy Place object created with the given coordinates for navigation. Then, the search proceeds to the next child node as directed by navigateToChild().
- If the node is not subdivided and the place has not been found or removed in the previous steps, the loop terminates, and the method returns false to indicate that no place was removed.

Edit: The method allows users to modify the services associated with a particular Place object based on its (x, y) coordinates. The method first searches for the Place using its coordinates and displays the details of the object if it is found. If no such Place is found, the user is notified. The editPlace() method utilizes an interactive menu that offers options for adding or removing services. It invoked the removeService() or addService() methods within the "Place" class (A detailed description of the two methods will be described within the b) section). Users are presented with a list of available services and can either add a service or remove one from the current list associated with the Place. The process is intuitive and easy to follow, with the system providing immediate feedback on the success or failure of their actions. This method enhances the system's usability by providing real-time data management capabilities and enriches user engagement by making the interaction with the application's spatial data more interactive and responsive.

b) ServiceType

Our application has a classification of services that can be provided at different locations. A key component of this class is the Enum, which encapsulates the different types of services offered. The Enum defines the types of services and provides the necessary methods for the application, including handling, adding, and removing services. For instance, the application offers three types of services, each represented by a binary value: Hotel (0b001), Hospital (0b010), and ATM (0b100).

Combining binary values to represent service types is not just a theoretical concept but a practical and efficient approach. For example, if we have a location that offers 2 services, an ATM and a Hotel, their binary value representation would be 0b101.

Add service: To handle this operation, we use bit manipulation and an "OR" relation logic gate. For example, the current service is 0b001, which represents the Hotel, and I want to add a service ATM (0b100). By using the "OR" logic in a bitwise operation, we will loop through the binary values to give the associated answer.

```
  001
| 100
-----
  101
```

Remove service: We use the "AND" and "NOT" logic gates for this one. First, we will negate the binary values of the service we want to remove and then apply the "AND" logic to the current value. For example, our current service binary value is 101, and we want to remove a service whose binary value is 001. First, we negate the service we want to remove 110 and then use the "AND" logic.

```
  101
& 110
-----
  100
```

c) Array List

Insert: To ensure a constant insertion time, we opt to add new elements to the end of the array. To keep track of the size of the array, we utilize an attribute called "size," which increments by one after each insertion. Additionally, the ArrayList contains a "capacity" attribute, which is checked prior to insertion to determine whether the size limit has been reached and whether the item can be added.

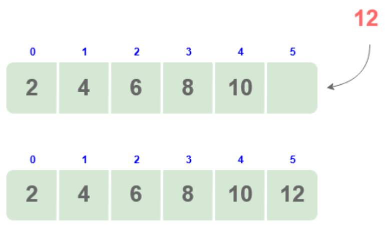


Figure 3: ArrayList Insertion

Shift left: This operation will shift all the elements to the right of the empty index to the left by 1 unit. It is used to handle array manipulation after an element is removed and ensure that the array remains compact and all elements are contiguous.

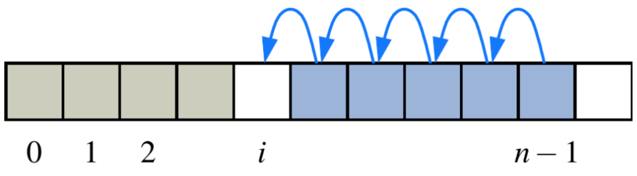


Figure 4: Array List Shift Left

Remove: This operation will loop through an array to find the specified element, empty the array at a specific index, and apply the shift left operation at that index.

3.4. Optimization

3.4.1 Leaf Node Insertion Optimization

As we progressed in quadtree implementation, we discovered an optimization of the insertion process, which can significantly impact performance, particularly as the dataset scales.

Traditional quadtree implementations typically require traversing from the root node to locate the appropriate leaf node for insertion[10]. However, we devised a mechanism to save the leaf node during the search process, eliminating the need for repeated traversal from the root for subsequent insertions within the same region. This optimization minimizes traversal overhead, ultimately reducing time complexity and improving overall efficiency.

a) Reasons for Not Implementing the Optimization

Our current application operates on a moderate scale, requiring the management of approximately 100,000,000 nodes within the quadtree structure. This substantial dataset size prompts us to carefully evaluate the efficiency of insertion techniques. Calculating the depth of the quadtree can provide insights into the feasibility of optimizing strategies.

The formula to determine the depth of a quadtree accommodating 100,000,000 nodes:

$$d = log_4(n)$$

where: n represents the total number of nodes

d denotes the depth of the quadtree

We find that the quadtree's depth is approximately 13.57. This indicates that it reaches a depth of approximately 14 levels to accommodate 100,000,000 nodes.

With a depth of around 14, the quadtree's structure is relatively shallow, meaning traversing from the root node to locate the appropriate leaf node does not incur significant overhead. Employing the leaf node insertion optimization may not yield substantial performance improvements at this scale for several reasons:

- **Traversal Overhead:** The overhead of traversing from the root to leaf nodes is manageable due to the shallow depth of the quadtree. The traditional insertion method remains efficient, and the optimization's benefits may not justify the added complexity.
- **Space Efficiency:** Implementing the leaf node insertion optimization requires storing references to leaf nodes, which can increase memory usage. However, for a quadtree with a depth of 14, the additional space overhead may not be warranted, leading to inefficient space utilization.

Given these considerations, the traditional insertion method, which involves traversing from the root node, remains efficient and pragmatic for our current application with approximately 100,000,000 nodes. The relatively shallow depth of the quadtree suggests that the overhead incurred by repeated traversals is not a significant bottleneck to performance, making the leaf node insertion optimization unnecessary at this scale.

b) Efficiency at Large-Scale

However, as the application's dataset scales up, the leaf node insertion optimization becomes more efficient. Consider a scenario where the quadtree needs to accommodate a depth of 100:

- Depth: *100*
- Total Nodes: $n = 4^d = 4^{100}$

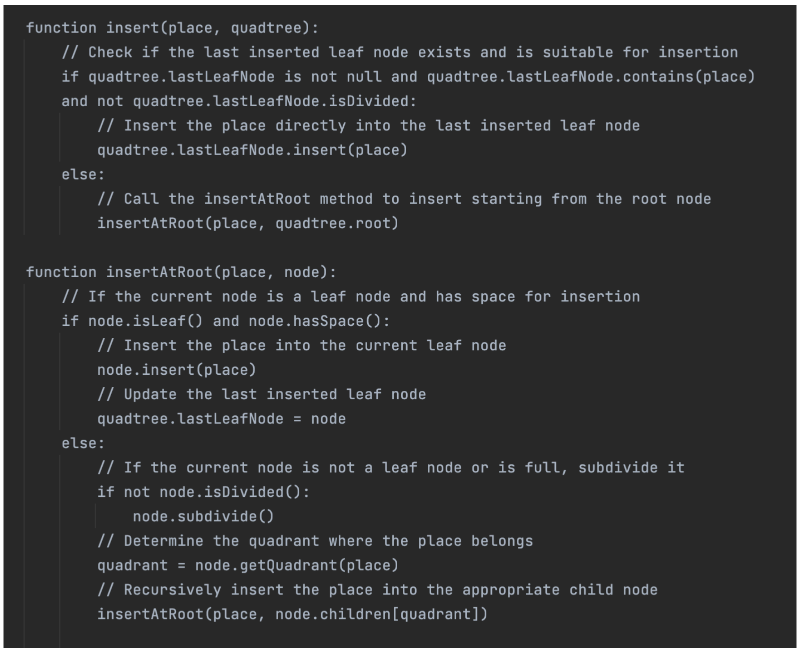
where: d is the level of the quadtree

n is the total number of nodes at level d

With such a substantial depth, the quadtree's size expands exponentially, resulting in many nodes. In this scenario, the leaf node insertion optimization becomes highly efficient:

- **Reduced Traversal Overhead:** The optimization eliminates the need for repeated traversal from the root for subsequent insertions within the same region, significantly reducing time complexity. With a depth of 100, the traversal overhead would be substantial without this optimization.
- **Improved Insertion Efficiency:** By directly inserting into saved leaf nodes, the insertion process achieves near-constant time complexity, enhancing overall performance. The logarithmic time complexity of traditional insertion algorithms becomes impractical at this scale.

Pseudo-code:



```
function insert(place, quadtree):
    // Check if the last inserted leaf node exists and is suitable for insertion
    if quadtree.lastLeafNode is not null and quadtree.lastLeafNode.contains(place)
    and not quadtree.lastLeafNode.isDivided:
        // Insert the place directly into the last inserted leaf node
        quadtree.lastLeafNode.insert(place)
    else:
        // Call the insertAtRoot method to insert starting from the root node
        insertAtRoot(place, quadtree.root)

function insertAtRoot(place, node):
    // If the current node is a leaf node and has space for insertion
    if node.isLeaf() and node.hasSpace():
        // Insert the place into the current leaf node
        node.insert(place)
        // Update the last inserted leaf node
        quadtree.lastLeafNode = node
    else:
        // If the current node is not a leaf node or is full, subdivide it
        if not node.isDivided():
            node.subdivide()
        // Determine the quadrant where the place belongs
        quadrant = node.getQuadrant(place)
        // Recursively insert the place into the appropriate child node
        insertAtRoot(place, node.children[quadrant])
```

By directly inserting into the leaf node without traversing from the root for each insertion, the code avoids unnecessary traversal and improves insertion performance, especially for large datasets. This optimization reduces the time complexity of insertion from O(log n) to approximately O(1) in the best case when the point is inserted directly into a leaf node and O(log n) in the worst case when subdivision is required.

The optimization minimizes the number of traversal steps and reduces the overall overhead associated with insertion operations, making the quadtree structure more efficient for spatial indexing and querying tasks.

3.4.2 Transition from ArrayList to Binary Value

The previous implementation used an ArrayList<ServiceType> to store the set of services offered at a location. Each service type was represented as a string, leading to potential memory overhead for storing strings and inefficient storage for storing the ArrayList object. Therefore, we transitioned to a more optimized solution to address these weaknesses and enhance performance. Using integer representations, this optimization used bit manipulation operations to represent each service type as a binary value.

The ServiceType enum assigns a unique binary value to each service type, enabling efficient representation and manipulation of service combinations using bitwise operations. Methods like randomizeServices(), getServicesByBinary(), and servicesToBinary() utilize the bitwise operations to handle service types efficiently.

Let's consider the below example:

Suppose we have a location offering hotel, restaurant, and ATM services. Instead of storing these services as strings in an ArrayList, we represent the combination of services using bitwise OR operations of a binary value.

HOTEL: 0b000001

RESTAURANT: 0b000010

ATM: 0b000100

Combination: 0b000111

Comparison of Space Complexity:

- **Using ArrayList of Strings**: Space complexity for storing service types: O(n * m), where n is the number of locations and m is the average number of services per location.
- **Using Binary Values**: The space complexity for storing service types is O(n), where n is the number of locations. Each location is represented by a single integer value containing the binary representation of service types.

Benefits of Bit Manipulation:

- **Space Efficiency:** Storing service types as binary values significantly reduce memory consumption compared to storing strings.
- **Computational Efficiency:** Bit manipulation operations offer constant-time complexity for adding, removing, and checking service types, leading to optimized performance.
- **Compact Representation:** Binary values provide a compact representation of service combinations, making them suitable for storage and manipulation in memory-constrained environments.

4. Complexity Analysis

4.1. Insertion

4.1.1. Quad-tree insertion visualization:

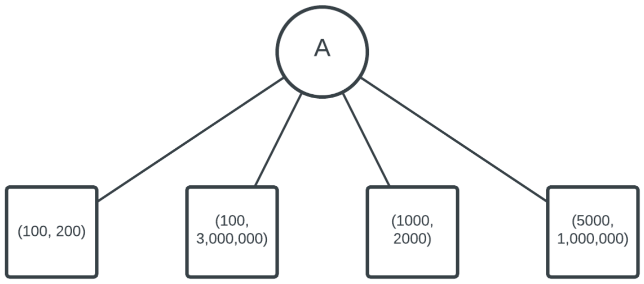


Figure 5: QuadTree Insertion Visualization

- The quad-tree is first initialized at root tree A, with a capacity of 4 and a dimension of 10,000,000. Four places are inserted into the tree.

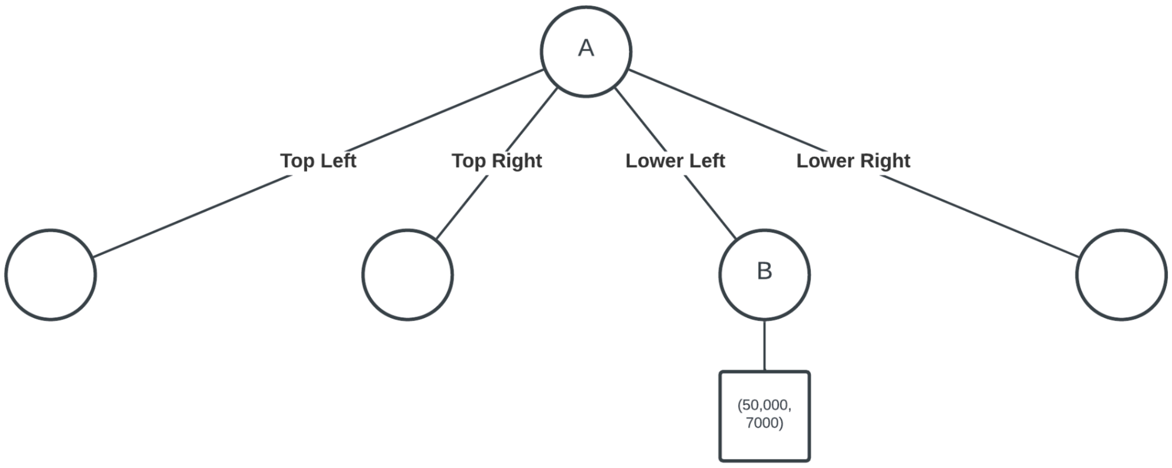


Figure 6: QuadTree Insertion Visualization

- Insert 1 place with coordinate (5000, 1,000,000). Since tree A's capacity is full, it will be subdivided into 4 subtrees. The new place will be inserted into the corresponding subtree. Subtree B has a dimension of 2,500,000, and its boundary contains the newly inserted place.

4.1.2. Master theorem:

Applying the master theorem to analyze this algorithm, we have the following recurrence relation: $T(n) = 4T(n/4) + O(n^1)$, where:

- n is the number of inputs (places to be inserted)
- 4 is the number of subproblems which the problems are divided into (the number of subtrees of each tree).
- n/4 is the size of each subproblem.
- $O(n^1)$ is the total time taken to insert the place into the array list. In this case, the number of iterations is bounded by the capacity - a constant value and does not scale with the number of inputs. Therefore, the time complexity for each insertion

is O(1), accumulating to O (n^1)

The relation follows the second case of Master Theorem, T(n) = O (n log n), which is the overall time complexity for quad-tree insertion.

4.1.3. Time complexity:

a. Time complexity for inserting 1 place:

n is the number of inserted places.

- Best case: Before traversing from the root tree, the last inserted leaf will be checked to see if it contains the place and if its capacity is not full. If it does, the time complexity will be constant O(1).
- Worst case: O(log n)

b. Time complexity for inserting n places:

- Best case: The best-case scenario happens when the n is smaller or equal to the tree's capacity. In this case, all places will be inserted into the root node. Thus, the complexity will be linear time - O(n)
- Worst case: O(n(log n))

4.1.4. Space complexity:

- n is the number of places inserted in the tree.
- Input space: Each quad-tree has an array list to store the places. The array list class has 2 attributes: an array storing the places and an integer storing the array size. The array space scales with n, while the integer stays constant. Thus, the input space grows linearly, contributing to O(n) space complexity.
- Auxiliary space: The maximum depth of the tree is log n. Therefore, the algorithm requires O (log n) space complexity.

4.2. Search:

4.2.1. Master theorem:

Applying the master theorem to analyze the worst case of this algorithm, we have the following recurrence relation: T(n) = 4 T(n/4) + O (1).

- n is the number of places of the quadtree.
- 4 is the maximum number of search operations on subtrees since each quad-tree has 4 children
- n/4 is the size of each subproblem.
- O (1) is the time taken to search an array list. This operation's complexity is constant because the number of iterations is bounded by the capacity, which does not scale with n.

The relation follows the third case of the Master Theorem, T(n) = O (n), which is the worst-case time complexity for quad-tree search.

4.2.2. Time complexity:

n is the number of places; c is the capacity; k is the number of places needed to be found.

- Best case: The best case scenario happens when k is 1, and the place that satisfies the query is at index 0 of the root's point list. The complexity would be constant time.
- Worst case: O(n)

4.2.3. Space complexity:

The search algorithm requires O(k) space complexity where k is the maximum number of places needed to be found; all found places will be stored in an array list with a capacity of k.

4.3. Edit

4.3.1. Time complexity:

The edit function utilizes the search algorithm to find the place to be edited.

- Best case: O (1)
- Worst case: O (n)

4.3.2. Space complexity:

The method has constant space complexity - O (1) as the memory used by the algorithm stays constant regardless of the input size:

- Array List "foundPlaces" has a fixed number of elements.
- The Scanner object's memory usage does not depend on the input size.
- "placeToEdit", "choice" and other local variables used for temporary storage during the execution also do not scale with the input.

4.4. Remove:

4.4.1. Time complexity:

The remove function utilizes the search algorithm to find the place to be edited.

- Best case: O (1)
- Worst case: O (n)

4.4.2. Space complexity:

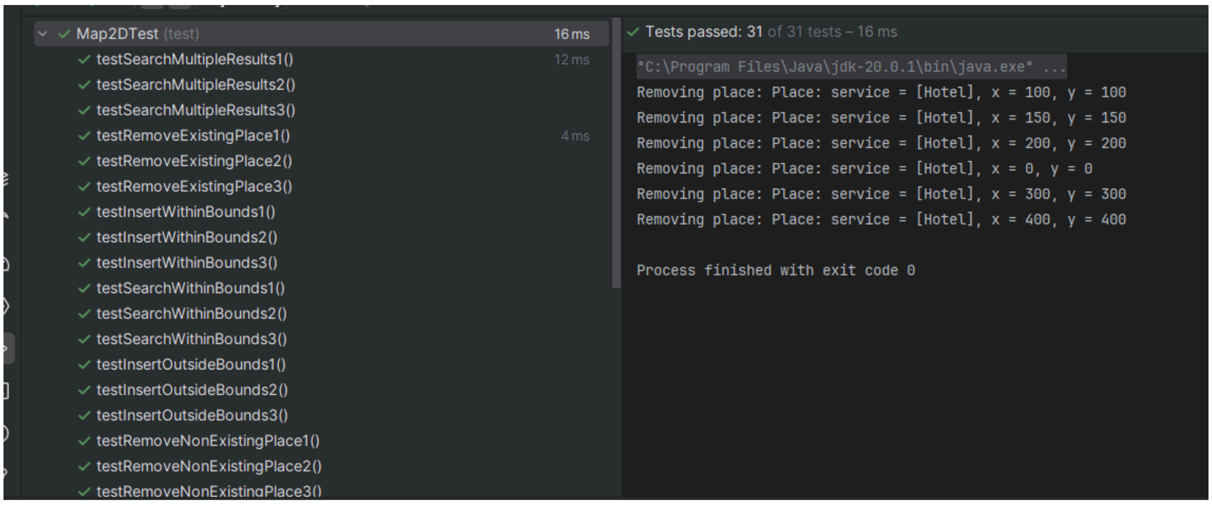
The remove method has constant space complexity - O (1) as the memory used by the algorithm stays constant regardless of the input size:

- The "current" variable is used as a pointer to traverse from the root tree; its memory usage is not proportional to the size of the tree.
- "placeToRemove" is a Place object used to store the place to be removed; its space does not increase with the input.
- The local variable used to loop through the array list also has constant space complexity.

5. Evaluation

5.1 Correctness Evaluation

We put the applications through rigorous testing to guarantee their accuracy. In order to accomplish this, we used code coverage tools to generate a comprehensive suite of JUnit tests that exhaustively tested all application functionalities. As part of our testing process, we ran unit tests that checked how well the application handles edge cases. This helped us to ensure that the application was free of any flaws. To automate testing and promote reliability, we utilized the Maven build tool. Additionally, we assessed the algorithm's performance by feeding it input data with known output values.



```
✓ Map2DTest (test)                    16 ms    ✓ Tests passed: 31 of 31 tests – 16 ms
  ✓ testSearchMultipleResults1()      12 ms
  ✓ testSearchMultipleResults2()
  ✓ testSearchMultipleResults3()
  ✓ testRemoveExistingPlace1()         4 ms
  ✓ testRemoveExistingPlace2()
  ✓ testRemoveExistingPlace3()
  ✓ testInsertWithinBounds1()
  ✓ testInsertWithinBounds2()
  ✓ testInsertWithinBounds3()
  ✓ testSearchWithinBounds1()
  ✓ testSearchWithinBounds2()
  ✓ testSearchWithinBounds3()
  ✓ testInsertOutsideBounds1()
  ✓ testInsertOutsideBounds2()
  ✓ testInsertOutsideBounds3()
  ✓ testRemoveNonExistingPlace1()
  ✓ testRemoveNonExistingPlace2()
  ✓ testRemoveNonExistingPlace3()
```

```
"C:\Program Files\Java\jdk-20.0.1\bin\java.exe" ...
Removing place: Place: service = [Hotel], x = 100, y = 100
Removing place: Place: service = [Hotel], x = 150, y = 150
Removing place: Place: service = [Hotel], x = 200, y = 200
Removing place: Place: service = [Hotel], x = 0, y = 0
Removing place: Place: service = [Hotel], x = 300, y = 300
Removing place: Place: service = [Hotel], x = 400, y = 400

Process finished with exit code 0
```

Figure 7: Execution Test Diagram

The diagram presented above showcases the application's successful execution by testing it with all the edge cases.

5.2 Efficiency Evaluation

In this section, we experimentally evaluated the system's efficiency by measuring various performance metrics at different depths and application node counts, finding the ideal tree depth for this problem. The structure of the quadtree, with each quadrant containing an ArrayList that holds a set of points, plays a crucial role in this evaluation. We must determine the array's capacity with the least empty space overhead to optimize the quadtree regarding space complexity. Since each quadNode has 4 quadrants, the number of quadNode at each level can be calculated accordingly.

$$C = 4^n$$

C: number of nodes at level n^{th}

n: is the level of the quadtree (start from 0)

Our goal is to distribute the nodes among quadNode fairly and even while minimizing unused space in arrays to reduce overhead.

$$D = \lceil B/C \rceil$$

D: refers to the recommended capacity of quadNode at different depths of the quadtree

B: indicates the total number of nodes present in the quadtree

C: number of nodes at level nth

By utilizing the formula $C = 4^n$, we can determine the optimal capacity for each quadNode based on the quadtree depth. This calculation is a key step in our process of evaluating efficiency in this section.

The metrics used for efficiency evaluation conclude with an analysis of the ideal depth range for a quadtree structure with a node count below 100 million.

Search Time: This metric measures the time for spatial queries, such as finding POIs within a given bounding rectangle. It is a crucial indicator of the system's overall performance and responsiveness when handling user queries (average search times of 100 searches).

Initialization Time: This metric measures the time required to construct and initialize the quadtree data structure. It is essential to consider this overhead, especially in scenarios where the quadtree needs to be frequently rebuilt or updated.

Memory Usage: This metric measures the amount of memory consumed by the quadtree data structure. It is crucial for assessing the system's memory footprint and scalability, particularly when dealing with large datasets or resource-constrained environments.

Heat Map 1: Search Time by Depth and Node Count

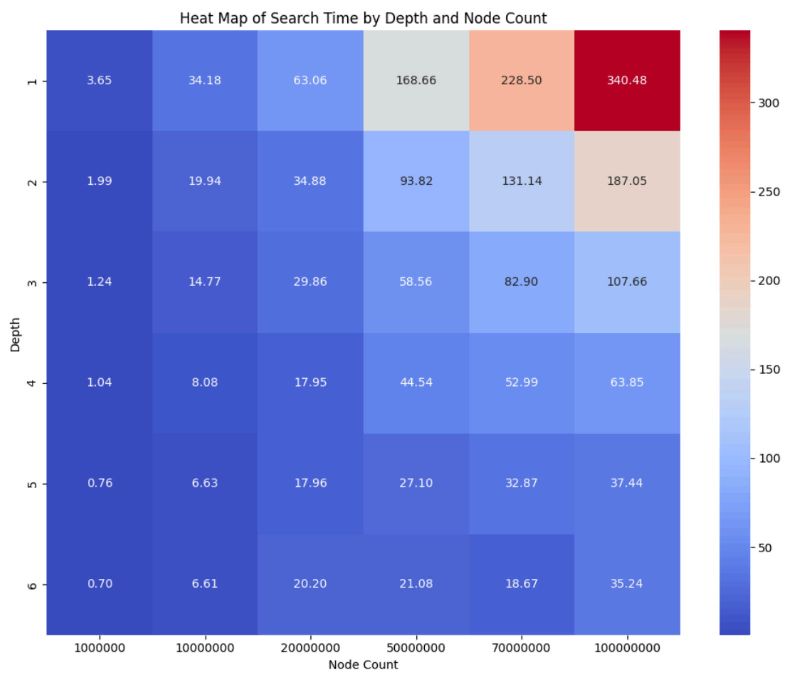


Figure 8: Heat Map of Search Time by Depth and Node Count

The visual representation of the data structure's search time is depicted through a heat map. This heat map displays the average time to perform a search operation, measured in milliseconds, for various combinations of node count and tree depth. For instance, when the node count is 50,000,000, and the tree depth is 4, the search operation takes an average of 44.54 milliseconds. The search time depends on the node count because a higher number of nodes leads to a more complex quadtree; distributing the nodes into their appropriate quadrant leads closer to the average case of the time complexity of the search. Additionally, increasing the depth of the tree reduces the time taken to perform the search operation.

Heat Map 2: Initialization Time by Depth and Node Count

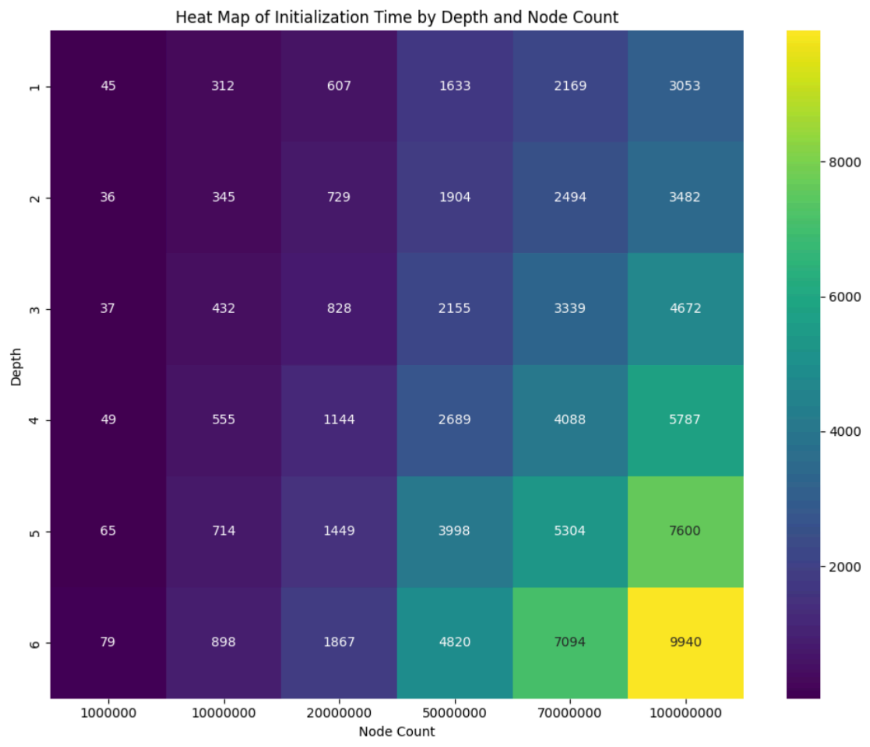


Figure 9: Heat Map of Initialization Time by Depth and Node Count

This visualization displays a heat map revealing the millisecond time needed to initialize and build a quadtree. The initialization time is measured for different combinations of node count and tree depth. Each number on the map represents the time it took to initialize the quadtree structure. From the map, we can observe that the deeper the tree, the longer it takes to initialize it. This happens because the number of quadNode increases exponentially with the depth of the tree, leading to a longer time to allocate more reference objects of the quadtree.

Heat Map 3: Memory Usage by Depth and Node Count

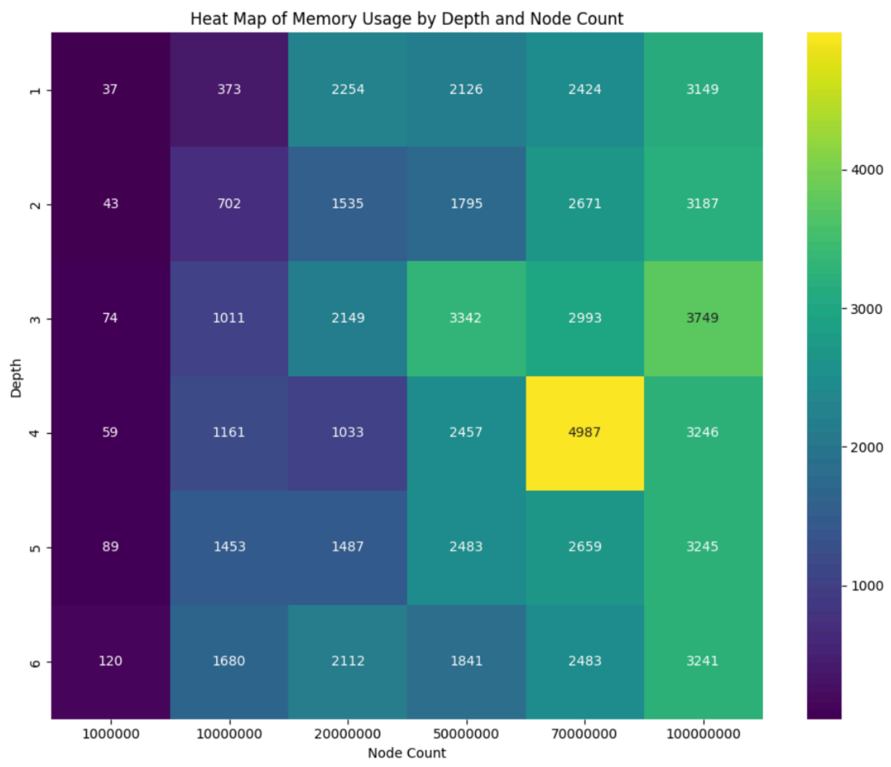


Figure 10: Heat Map of Memory Usage by Depth and Node Count

This heat map shows the quadtree data structure's memory usage (in bytes). It measures memory usage for different combinations of tree depth and node count. The numbers in the heat map represent the average memory consumption of the quadtree structure. The memory usage varies with the node count because as the number of nodes increases, more memory is required to store the additional nodes and their associated data structures within the quadtree. Moreover, memory usage increases with depth because deeper trees have more levels of subdivision, leading to more nodes that need to be stored in memory.

In conclusion, based on the heat maps provided, it is possible to identify an ideal depth range for a quadtree structure with a node count below 100 million. Considering the application's specific requirements and constraints, this range should strike a balance between search time, initialization time, and memory usage. For a quadtree with a node count below 100 million, a depth range of 3 to 5 appears to be a reasonable choice. At these depths, the search time, initialization time, and memory usage remain relatively manageable while providing sufficient spatial partitioning for efficient querying.

However, it is important to consider the trade-offs involved in selecting the depth of the quadtree:

1. **Search Time vs. Initialization Time**: A deeper quadtree structure generally results in faster search times due to more granular spatial partitioning, but it also increases the initialization time as more nodes need to be created and populated.
2. **Search Time vs. Memory Usage**: Similar to the trade-off with initialization time, a

deeper quadtree structure improves search time but requires more memory to store the additional nodes.

3. **Initialization Time vs. Memory Usage**: A deeper quadtree structure typically requires more initialization time and consumes more memory due to the increased number of nodes.

The best-case scenario for selecting the ideal depth of the quadtree depends on the application's specific requirements and constraints. If search time is the primary concern and memory and initialization time are less critical, a deeper quadtree structure (depths 4-5) may be preferred. On the other hand, if memory usage and initialization time are more important factors, a shallower quadtree structure (depths 3-4) might be a better choice, albeit with slightly slower search times.

Additionally, if frequent updates or rebuilds of the quadtree structure are needed, prioritizing initialization time may be more crucial to ensure responsiveness and minimize downtime.

Ultimately, selecting the ideal depth should be based on careful profiling, benchmarking, and considering the application's specific use case and resource constraints.

6. Conclusion

Developing a quadtree-based Java application for optimizing Point of Interest (POI) searches has been a significant breakthrough in handling large-scale spatial data. The application has successfully met all the objectives by delivering fast and accurate search capabilities while maintaining the robustness required to handle up to 100 million POIs on a large map. The algorithm's performance was rigorously validated through extensive testing, including comprehensive correctness and efficiency evaluations. This thorough testing ensures our system operates reliably under various conditions, providing our users with a dependable tool.

Even though the application has successfully delivered all required functionality, some limitations and issues could be improved in future updates. One of the limitations is that the GUI lacks visualization capabilities, which leads to decreased user performance and contributes to a complex and error-prone process while using the application. Another weakness of the system is the search algorithm, which has the worst-case complexity being linear time. Even though the chances of a worst-case scenario are low, the issues can be improved with thorough investigation and research for better implementation.

This project enhanced their understanding of spatial data structures and their applications and set a solid foundation for future improvements and expansions. The team is confident that the methodologies and insights gained from this project will be invaluable for future geographic information system development endeavors.

7. References

[1] A. Eldawy and M. F. Mokbel, "The Era of Big Spatial Data: Challenges and Opportunities," 2015 16th IEEE International Conference on Mobile Data Management, Pittsburgh, PA, USA, 2015, pp. 7-10, doi: 10.1109/MDM.2015.82.

[2] H. Samet, "The quadtree and related hierarchical data structures," ACM Comput. Surv., vol. 16, no. 2, pp. 187–260, Jun. 1984.

[3] H. Samet, Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS. Addison-Wesley, 1990.

[4] R. A. Finkel and J. L. Bentley, "Quad trees a data structure for retrieval on composite keys," Acta Inform., vol. 4, no. 1, pp. 1–9, 1974.

[5] H. Samet, "An overview of quadtrees, octrees, and related hierarchical data structures," in Theoretical Foundations of Computer Graphics and CAD, Berlin, Heidelberg, 1988, pp. 51–68.

[6] G. R. Hjaltason and H. Samet, "Distance browsing in spatial databases," ACM Trans. Database Syst., vol. 24, no. 2, pp. 265–318, Jun. 1999.

[7] H. Samet, Foundations of Multidimensional and Metric Data Structures. Morgan Kaufmann, 2006.

[8] I. Gargantini, "An effective way to represent quadtrees," Commun. ACM, vol. 25, no. 12, pp. 905–910, Dec. 1982.

[9] G. Mattaparthi, "Ball tree and KD Tree Algorithms - Geetha Mattaparthi - Medium," Medium, Jan. 23, 2024. [Online]. Available: https://medium.com/@geethasreemattaparthi/ball-tree-and-kd-tree-algorithms-a03cdc9f0af9#:~:text=A%20hierarchical%20data%20structure%20called,for%20effective%20closest%20neighbor%20searches

[10] G. Moon, Understanding the Quadtree Data Structure, (16 January, 2024). Accessed 9 May, 2024. [Blog]. Available: https://www.deepblock.net/blog/quadtree

[11] P. Dinkins. "Quad-Tree Geospatial Data Structure: Functionality, benefits, and limitations." Open Source GIS Data. Accessed: May 09, 2024. [Online]. Available: https://opensourcegisdata.com/quad-tree-geospatial-data-structure-functionality-benefits-and-limitations.html

Contribution Form

Name	SID	Contribution Score
Pham Phuoc Sang	s3975979	5
Nguyen Ngoc Thanh Mai	s3978486	5
Cao Nguyen Hai Linh	s3978387	5
Tran Pham Khanh Doan	s3978798	5