

Municipality App

PROG7312



November 18, 2024

Fortune Madoda

St10206444

Contents

[Implementation Report: Service Requests in Municipality Application 2](#_Toc182845064)

[Project Completion Report: Municipality Application Development Using Windows Forms 7](#_Toc182845065)

[REFERENCES 10](#_Toc182845066)

# Implementation Report: Service Requests in Municipality Application

Overview

This report outlines the usage of specific data structures implemented in the Municipality Application, with a focus on Service Requests. The data structures employed include a Binary Search Tree (BST), AVL Tree, Heap (implemented using a Queue), and Graph. These data structures have been tailored to meet the specific needs of the application, such as efficient searching, prioritization, and relationship management.

Data Structures Used

1. Binary Search Tree (BST) for Request ID Management

Implementation:

* The BST is implemented to store ServiceRequest objects, where the ID of each request serves as the key for the tree. The ID ensures that each service request has a unique identifier, and the BST uses this key to keep requests in sorted order.
* Each ServiceRequest has an ID that serves as the primary key, making it easy to insert, find, and remove requests from the BST.
* The insertion operation ensures that requests are placed in the correct position in the tree based on their ID.

Contribution to Efficiency:

* Efficient Lookup: By using the ID as the key, the BST allows O(log n) time complexity for searching for a service request by its ID.
* Ordered Insertion: The BST ensures that service requests are inserted in a sorted manner, allowing for quick access to requests by ID.
* Balanced Structure: As long as the tree remains balanced, searching, adding, and deleting service requests based on their ID remains fast and efficient.

How It Works in the Application:

* When a new service request is submitted, it is inserted into the BST based on its ID.
* If a service request needs to be updated or deleted, the BST is queried using the ID to find the request and apply the appropriate operation.
* For example, a service request with ID 101 could be inserted, and the BST keeps the service requests in a way that makes future retrievals fast.

2. AVL Tree for Timestamp/Service Request Management

Implementation:

* The AVL Tree is used to manage ServiceRequest objects by timestamp or priority. This tree automatically balances itself whenever requests are inserted or removed, ensuring that the tree remains balanced at all times.
* Each ServiceRequest object in the AVL tree contains a timestamp field, which allows the AVL Tree to order the requests based on when they were created.
* There is a separate AVL tree that is used to update the details created in the add requests form.

Contribution to Efficiency:

* Balanced Tree Structure: The AVL tree is self-balancing, which guarantees O(log n) time complexity for insertions, deletions, and lookups even when the dataset grows.
* Efficient Sorting: Requests are stored in the AVL tree based on their timestamp, which ensures that requests are processed in the order in which they were created.
* Optimal Performance: Thanks to the tree's balance, the performance does not degrade, even with a large number of service requests.

How It Works in the Application:

* When a new service request is added with a timestamp, it is inserted into the AVL tree. This ensures that service requests are always sorted based on the timestamp, which can be used to easily retrieve the oldest or most recent request.
* For example, a request with a timestamp of 2024-11-10 14:00 will be placed in the tree, and requests with earlier timestamps will be found in the left subtree of the node containing this request.

3. Heap (Queue) for Priority Management

Implementation:

* A Heap is used to manage the priority of service requests based on their urgency or importance. In this case, a min-heap is employed, where requests with the highest priority (e.g., most urgent requests) are placed at the root, making them easy to access.
* The heap is implemented using a Queue, allowing the application to quickly retrieve the service request with the highest priority for processing.

Contribution to Efficiency:

* Efficient Priority Handling: The heap allows for constant-time retrieval of the highest-priority service request and O(log n) time complexity for inserting a new request or reordering the queue when priorities change.
* Priority Sorting: The service requests in the heap are always sorted by priority, allowing urgent requests to be processed first.

How It Works in the Application:

* When a new service request is created with a specified priority, it is added to the heap.
* The root of the heap always represents the highest-priority request, which can be accessed and processed first.
* For example, a service request with a priority of 1 (highest priority) will be placed at the root of the heap, ensuring it is processed before requests with lower priorities.

4. Graph for Managing Dependencies Between Service Requests

Implementation:

* The Graph data structure is used to model the dependencies between ServiceRequest objects, reflecting how one request can influence or relate to others.
* Each ServiceRequest is treated as a node in the graph, and edges are created to represent dependencies or relationships between them. These dependencies can be based on shared categories, locations, or specific inter-service request relations (such as follow-up actions or escalation).
* The edges between the nodes represent the dependency, meaning that a service request might require the completion or resolution of another service request before it can be processed, or they could be related because they deal with the same issue or location.

Contribution to Efficiency:

* Efficient Dependency Management: The graph structure helps in efficiently managing the interdependencies between service requests. For example, if one service request cannot be resolved until another one is addressed, the graph can model this dependency.
* Dependency Queries: By using traversal algorithms like Depth-First Search (DFS) or Breadth-First Search (BFS), the graph allows you to quickly find all the service requests that are dependent on or related to a particular request.
* Automatic Grouping: The graph allows the application to automatically group dependent requests together for processing. For instance, if a service request requires a follow-up action after another request is resolved, this relationship can be identified through the graph, allowing the system to prioritize tasks efficiently.

How It Works in the Application:

* When a service request is created, it is inserted as a node in the graph. Edges are then added to link this request with any other requests that are either dependent on it or share a common location, category, or issue.
* For example:
  + If a service request to repair a water leak in a building is raised, it could be linked to a follow-up request for inspection once the leak is fixed.
  + Alternatively, multiple requests raised for potholes in the same neighbourhood could be interconnected, making it easier to address all related issues together.
* These dependencies can be visualized as edges between nodes:
  + Node A (Water Leak Request) → Node B (Inspection Request after Repair), indicating that the repair request needs to be resolved before the inspection can take place.
  + Node C (Pothole Request 1) → Node D (Pothole Request 2), indicating that both pothole requests are related to the same neighbourhood and may be handled together.
* The graph structure enables efficient querying of all service requests related to a particular issue or location and identifies the order in which dependent requests should be handled.

Performance and Contribution to Application

Each of these data structures has a specific role in ensuring the application runs efficiently:

* The BST allows for fast lookups, insertions, and deletions of service requests by ID, which is critical for quick management of requests.
* The AVL Tree guarantees that requests are ordered and processed based on their timestamp or priority, ensuring that no requests are missed or delayed.
* The Heap (Queue) ensures that the most urgent requests are handled first, improving the responsiveness of the system.
* The Graph efficiently manages relationships and dependencies between service requests, events, and categories, allowing the application to query interconnected data without needing complex joins or manual sorting.

Additional feature recommendations:

While the current use of these data structures provides efficient service request management, there are additional considerations and improvements that could be made in the future:

* Database Integration: A relational or NoSQL database could be used to persist service requests and related data. The database could integrate with the existing data structures to persist the data and facilitate more complex queries, such as joins across multiple tables or nodes.
* Optimized Search with Indexes: Implementing indexes on fields like priority or timestamp could further speed up search operations, especially for large datasets.
* Distributed Systems: As the application scales, implementing distributed data structures (e.g., distributed heaps or distributed databases) may be required to ensure that the system can handle large volumes of service requests efficiently.
* Real-Time Notifications

- Notifying users in real-time when their service requests are updated or resolved will improve user engagement and satisfaction.

- Instant feedback ensures residents are always informed about the status of their reports or requests.

* Service Request Analytics and Reporting
  + Data-driven decision-making is crucial for municipalities to understand common issues, identify patterns, and optimize resource allocation. This had been mentioned in part one of the POE but it is still relevant even now.
  + The ability to generate reports based on service request data (e.g., most common issues, geographic distribution, average resolution time) helps both residents and municipal staff.
* Service Request Escalation System
  + Some requests might not be resolved within a reasonable timeframe. An escalation mechanism ensures that these requests are passed up to higher levels of authority when needed.
  + This ensures that no request is left unattended and increases accountability.
* Automated Feedback Analysis (Sentiment Analysis)
  + Feedback collected from users can provide valuable insights into the user experience, but analysing it manually can be time-consuming.
  + Automated analysis will help identify trends or issues from user feedback and make data-driven improvements.

# **Project Completion Report: Municipality Application Development Using Windows Forms**

**Introduction**

This project aimed to develop a municipality application, incorporating features like reporting issues, viewing local events, submitting service requests, and providing feedback. The application was built using **Windows Forms**, a framework that, while functional, presented several challenges due to its limited flexibility compared to modern UI frameworks. Despite these challenges, the project achieved its primary goals, although certain aspects of its functionality and usability remain areas for improvement.

**Challenges in Designing with Windows Forms**

Designing the application with **Windows Forms** came with several limitations:

1. **Scaling Issues:**
   * One significant challenge was ensuring the application scaled properly across different screen resolutions and window sizes. Windows Forms lacks robust scaling tools, making it difficult to maintain a consistent UI experience across devices.
   * **Improvement:** Leveraging modern UI frameworks like **WPF** or libraries such as **WinForms DPI Awareness** could help create a more scalable design in future projects.
2. **Navigation and Business Logic:**
   * The feedback noted that the application navigation is not business-like. Ensuring logical transitions and clear pathways for users is essential for creating a seamless user experience. Additionally, there are "dead" buttons in the menu, which undermine the application's credibility.
   * **Improvement:** Implementing well-defined user journeys and addressing placeholder buttons with appropriate functionality would improve the professional feel of the application.
3. **Exception Handling:**
   * The application currently lacks robust **exception handling**, which is critical for preventing crashes and ensuring a smooth user experience. For example, certain edge cases in user input or data manipulation might cause the application to behave unpredictably.
   * **Improvement:** Adding try-catch blocks, logging mechanisms, and detailed error messages would make the application more robust.
4. **Exiting the Application:**
   * A key usability concern raised was that the application does not exit properly. This indicates missing logic in managing application lifecycle events or form controls.
   * **Improvement:** Ensuring proper cleanup logic when closing the application (e.g., releasing resources, saving state) will resolve this issue.
5. **Undeclared Use of AI:**
   * While AI tools like **ChatGPT** and **GitHub Copilot** were invaluable during development, their contributions were not explicitly acknowledged. Transparent documentation of external resources is crucial for maintaining integrity in the development process.
6. **Uncommented Methods:**
   * Many methods in the application lack comments, making the code less maintainable and harder for others to understand. This was highlighted as an area for improvement in feedback.
   * **Improvement:** Adding clear comments explaining the purpose and functionality of each method would make the codebase more readable and professional.

**What Was Learned**

1. **UI/UX Considerations:**
   * The importance of user-friendly design and clear navigation was reinforced. The feedback highlighted the need to prioritize the end-user experience by ensuring features are intuitive and logically connected.
2. **Scalability and Performance:**
   * The challenges with scaling and handling complex data structures in a desktop application provided valuable insights into how modern frameworks might better address these issues.
3. **Importance of Exception Handling:**
   * Developing a robust exception handling mechanism was a key learning point. This is essential for preventing crashes and ensuring the application performs reliably under various scenarios.
4. **Value of Transparent Documentation:**
   * Acknowledging all resources, including the use of AI tools, is important for ethical development practices and maintaining transparency.

**Areas for Improvement**

1. **Data Volume and Convincing Use Cases:**
   * The application currently lacks sufficient sample data to convincingly demonstrate its functionality. Adding more realistic and diverse datasets would make the application feel more comprehensive and practical.
2. **Robust Exception Handling:**
   * Introducing a global exception handling mechanism and ensuring edge cases are handled gracefully will enhance application stability.
3. **Navigation and User Flow:**
   * Streamlining the navigation to align more closely with real-world business use cases is necessary. Eliminating placeholder buttons and adding meaningful functionality will make the application more polished.
4. **Code Comments and Documentation:**
   * Commenting methods and providing clear, comprehensive documentation will make the project more maintainable and easier for others to contribute to.
5. **Exiting Logic:**
   * Properly managing the application lifecycle and ensuring all resources are released on exit will improve the overall user experience.

**Conclusion**

The development of the municipality application was a challenging yet rewarding experience. The feedback provided valuable insights into areas for improvement, including scaling, navigation, exception handling, and transparency in using external resources. While the project successfully integrated key features like reporting issues, managing service requests, and viewing events, it falls short in areas such as data volume, navigation design, and robustness. Moving forward, addressing these gaps and refining the application based on user feedback will result in a more polished and professional product.

Despite the challenges faced, this project has significantly enhanced my understanding of application development, data structures, and UI/UX design, setting the foundation for further growth and improvement.

# REFERENCES

OpenAI. (2024) *ChatGPT* [Online]. Available at: <https://chat.openai.com/> (Accessed: 12 November 2024).

OpenAI. (2024) *ChatGPT* [Online]. Available at: <https://chat.openai.com/> (Accessed: 16 November 2024).

OpenAI. (2024) *ChatGPT* [Online]. Available at: <https://chat.openai.com/> (Accessed: 18 November 2024).

Stack Overflow. (2024) *Stack Overflow* [Online]. Available at: <https://stackoverflow.com/> (Accessed: 16 November 2024).