Author

Name: Joel
Surname: Shaduka
Student No: ST10041239
Group No: 1

Implementation Report for "Service Request Status" Feature

1. Binary Search Tree (BST)

Why It Was Chosen:

The BST is ideal for scenarios where efficient searching is critical, as it reduces the average search time from (O(n)) (in a list) to (O(\log n)) (*Binary search tree(bst)*). Service request IDs are unique, making them perfectly suited for BST indexing without duplication concerns.

Benefits:

- **Efficient Searching:** Users can quickly locate specific requests based on their unique identifiers, even when the dataset grows large.
- **Dynamic Updates:** Adding or removing service requests dynamically maintains the tree's organization, without requiring a complete reordering.
  (*Binary search tree(bst)*)

Implementation:

I've implemented the Binary Search Tree in `/DataStructures/BinarySearchTree` file. This class uses the `/DataStrucures/TreeNode` object for a Node. The `BinarySearchTree` class contains all the implementation and domain specific logic for this data structure, it then provides public methods for the Application/Page Controllers to access & invoke.
This benefits code maintainability and separation of concerns.
Please Note: This file & architecture is used for all implementations.

Example:

A user searches for a service request with ID `12345`. Starting at the root node (`5000`), the system compares `12345` to the root:

1. Since `12345 < 5000`, it moves to the left child (`2500`).
2. It continues comparing until it finds the exact node, minimizing unnecessary checks.

```
/// <summary>
/// Feature 1 - Search by ID (Binary Search)
/// Uses Binary Search Tree to execute searching by provided Id
/// </summary>
public ServiceRequest SearchById(int id)
{
    // Use the BST's search method
    return serviceRequestBST.SearchById(id);
}
```

/Services/RequestService.cs

Please Search: `Feature 1` to find all documentation & core implementation within the code.

---

2. Binary Tree for Filtering by Category

Why It Was Chosen:

The binary tree efficiently categorizes service requests, enabling quick filtering without scanning the entire dataset. Categories naturally fit into a hierarchical structure, where parent nodes can represent broader categories, and child nodes refine them further (*Complete binary tree*).

Benefits:

- **Logical Organization:** Service requests are grouped logically, making it easier to retrieve related data.
- **Scalable Filtering:** As new categories are added, the binary tree can expand without disrupting the existing structure.
  (*Complete binary tree*)

Example:

Consider a binary tree with a root node representing "Service Requests." Child nodes represent categories like "Sanitation," "Roads," and "Utilities." If a user selects "Sanitation," the system directly retrieves all service requests under the "Sanitation" node and its subcategories (e.g., "Waste Removal").

```
/// <summary>
/// Feature 2 - Filter by Category (Binary Tree)
/// </summary>
public List<ServiceRequest> FilterByCategory(string category)
{
    return serviceRequestBT.FilterByCategory(category);
    // Uses the binary tree's filter method
}
public HashSet<string> GetAllCategories() // Helper method for the UI
{
    return serviceRequestBT.GetAllCategories();
    // Uses the binary tree's method to get all categories
}
```

/Services/RequestService.cs

Please Search: `Feature 2` to find all documentation & core implementation within the code.

---

3. Min-Heap for Filtering by Status

Why It Was Chosen:

The Min-Heap is a perfect fit for prioritizing tasks, as it naturally organizes data by priority levels. Statuses like "Pending" or "In-Progress" align well with the Min-Heap's structure, ensuring that the most urgent tasks are always retrieved first (*Min heap binary tree*).

Benefits:

- **Priority Retrieval:** The root of the Min-Heap always contains the task with the highest priority (e.g., "Pending").
- **Automatic Reordering:** When a request status changes (e.g., from "Pending" to "In-Progress"), the heap reorders itself, maintaining efficiency.
  (*Min heap binary tree*)

Example:

```
    /// <summary>
    /// Feature 3 - Filter by Status (Min Heap)
    /// </summary>
    public List<ServiceRequest> FilterByStatus(string status)
    {
```

```
        return serviceRequestsHeap.FilterByStatus(status);
        // Use the heap's filter method
    }
    public List<string> GetAvailableStatuses()
    {
        return AvailableStatuses; // Return the list of statuses
    }
```

/Services/RequestService.cs

Please Search: `Feature 3` to find all documentation & core implementation within the code.

Why These Structures Were Chosen Overall

1. **Scalability:** Each data structure handles increased data volumes efficiently without compromising performance.
2. **Specialized Roles:** Using specific structures for searching, filtering, and sorting ensures optimal performance for each feature.
3. **Ease of Maintenance:** Changes to the system (e.g., adding a new category or status) are straightforward and minimally disruptive.
4. **Enhanced User Experience:** The application responds quickly to user actions, providing seamless interactions even as the dataset grows.

Bibliography

Binary search tree(bst) (no date) *Programiz*. Available at: https://www.programiz.com/dsa/binary-search-tree (Accessed: 18 November 2024).

Complete binary tree (no date) *Programiz*. Available at: https://www.programiz.com/dsa/complete-binary-tree (Accessed: 18 November 2024).

Min heap binary tree (no date) *DigitalOcean*. Available at: https://www.digitalocean.com/community/tutorials/min-heap-binary-tree (Accessed: 18 November 2024).