PROG7312 - Part 3

APPLICATION PROGRAMMING 3B Implementation Report

Mikayle Devonique Coetzee

ST10023767

18 November 2024

Table of Contents

Implementation Report

Overview:

In my project I implemented multiple data structures to manage issues in the system where issues can be categorized and stored in multiple different data structures. The two main classes that uses calls from the data structures is the ‘IssueManager’ and the ‘IssueTracker’, The data structures are optimized for handling issues in an organized manner, to ensure efficient storage, retrieval and updates.

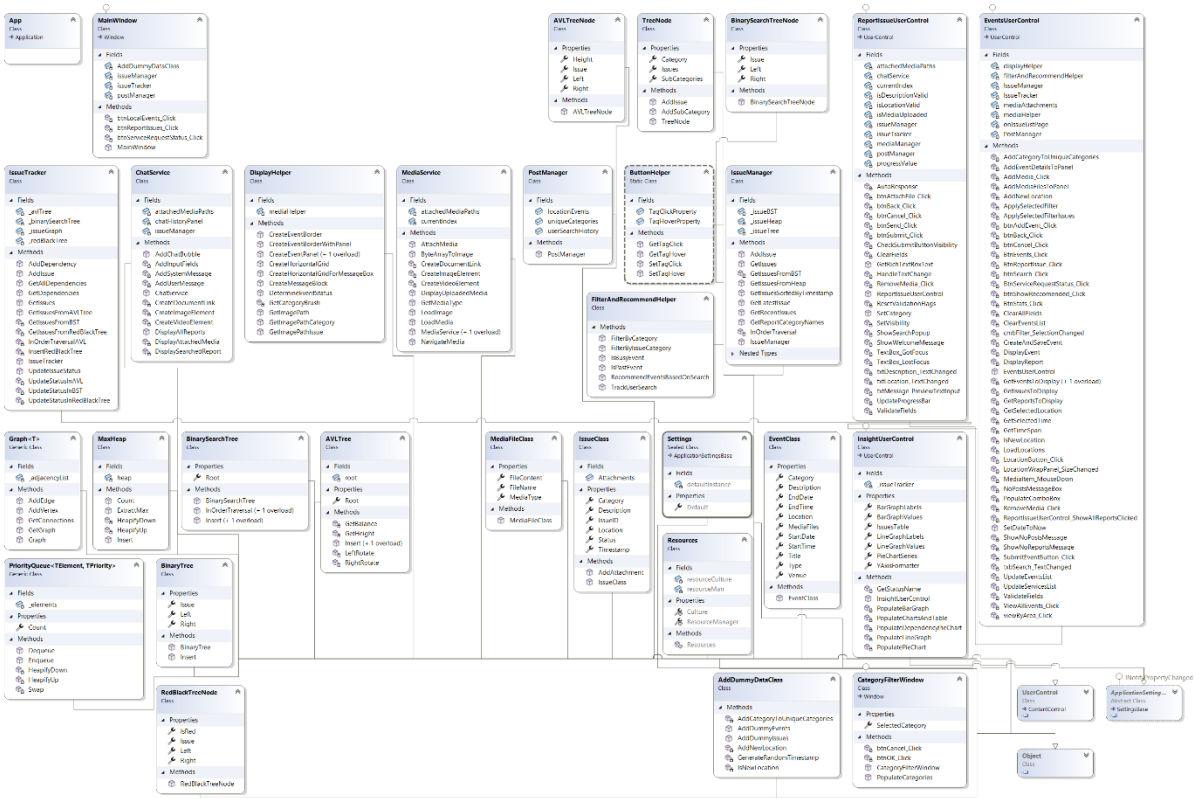

Figure 1.1. My class diagram

For a better view of my class diagram, feel free to open ‘ClassDiagram1.png’ where you can see the relationship between all of my classes, and the usage of the trees

Data Structures Used in Part 3:

1. Basic Trees:

Role and Contribution:

A basic tree is a hierarchy data structure that's simple but efficient. In my service requests, I use a basic tree to organize the service issues submitted into categories.

Example:

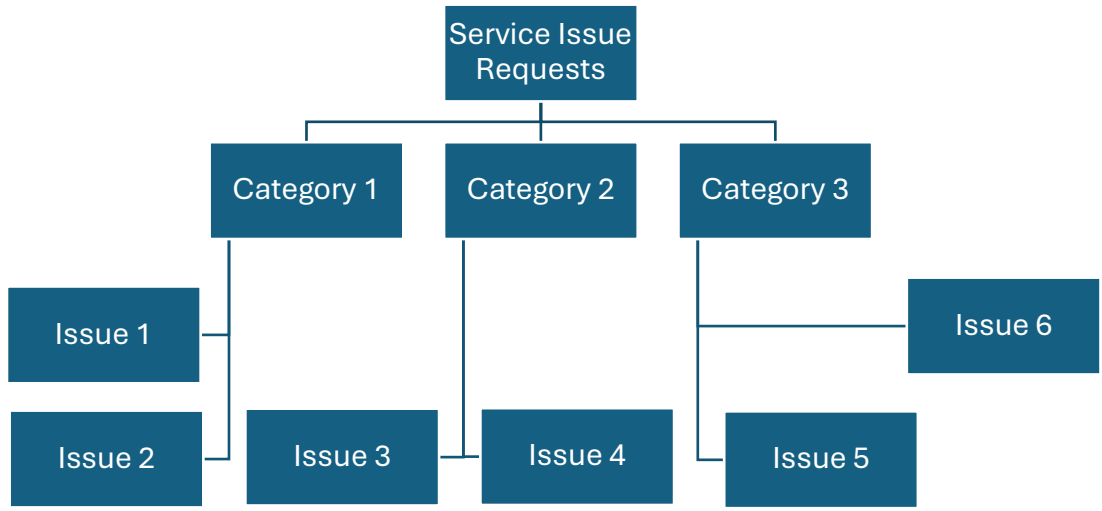

Figure 1.2. My basic tree structure

Data Stored: Categories of service requests (e.g., "Utilities," "Traffic", and others shown in the 'ReportCategory' enum).

Where and How Data Is Called: I used this tree structure to store the issues, and when the user filters by category, retrieve the branch that matches their selected category.

Example: When a user wants to filter service requests by category (e.g., "Utilities"), the tree allows you to quickly traverse and retrieve the requests within that category.

Why It's Good: The basic tree allows for an intuitive and clear categorization of service requests. Each category can easily be traversed, and new categories can be added without too much complexity.

```
5 references
public enum ReportCategory
{
    [Description("Utilities")]
    Utilities,

    [Description("Sanitation")]
    Sanitation,

    [Description("Potholes")]
    Potholes,

    [Description("Traffic")]
    Traffic,

    [Description("Road Signs")]
    RoadSigns,

    [Description("Other Issue")]
    OtherIssue,

    [Description("Traffic Lights")]
    TrafficLights,

    [Description("Car Crash")]
    CarCrash
}
```

2. Binary Trees:

Role and Contribution:

A Binary tree is a data structure that allows each node to not have more than two children, referred to as the left and right child (BeyondVerse, 2023). In my service request, I use a binary tree for ordering issues submitted in creation dates, making sure that the old and new issues are separated.

Example:

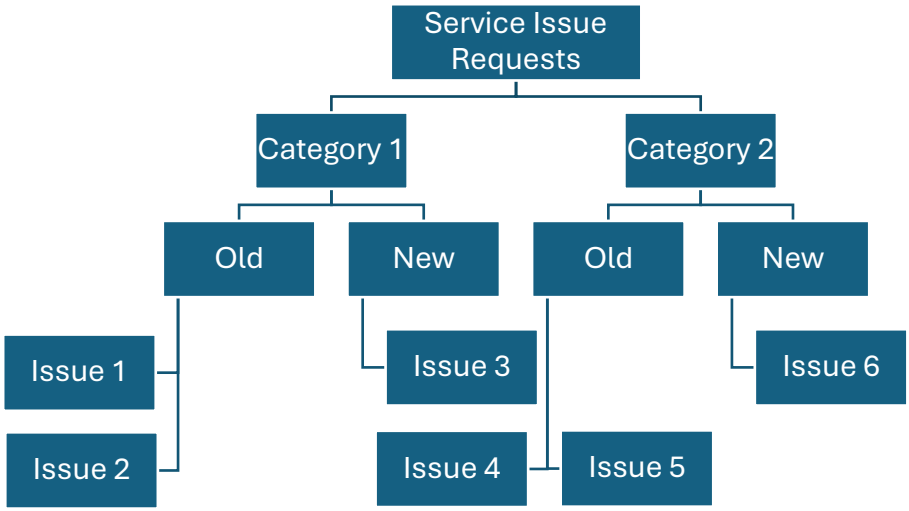

Figure 1.3. Basic binary tree structure

Data Stored: Categories of service requests (e.g., "Utilities," "Traffic") and underneath has a binary tree sorting each categories issue under ‘Old’ and ‘New’.

Where and How Data Is Called: I used this tree structure to store the issues and separate the old issues and the new issues, by doing so, I ensure that the retrieval of any new or old issues is quicker.

Example: When a user wants to view the list of submitted service issues, they will see the newer issues fist before the older submitted issues, so that they can see if someone else has submitted that issue before them (for now, the app only works for one user)

Why It’s Good: The binary tree is good for maintaining a dynamic ordered list of service requests. Making deletion of old requests easier.

```
5 references
public enum ReportCategory
{
    [Description("Utilities")]
    Utilities,

    [Description("Sanitation")]
    Sanitation,

    [Description("Potholes")]
    Potholes,

    [Description("Traffic")]
    Traffic,

    [Description("Road Signs")]
    RoadSigns,

    [Description("Other Issue")]
    OtherIssue,

    [Description("Traffic Lights")]
    TrafficLights,

    [Description("Car Crash")]
    CarCrash
}
```

3. Binary Search Trees (BST):

Role and Contribution:

A Binary Search Tree is a type of binary tree where the left child of a node contains values less than the right node (BeyondVerse, 2023). In my service requests I use a binary search tree to store and manage issue data based on their timestamp (left subtree = timestamp earlier than parent node; right subtree = timestamp later than parent node). By doing so I enable efficient searching and retrieval of the issues based on their time stamp. The BST structure also allows me to insert issues in a way that maintains the tree order.

Example:

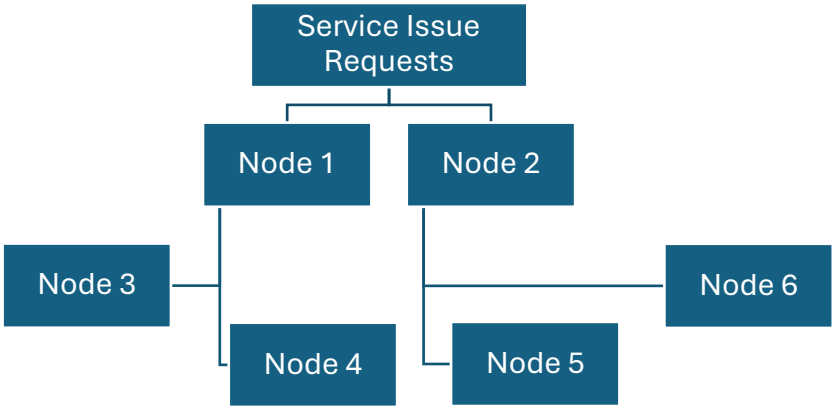

Figure 1.3. Basic binary search tree structure

Data Stored: The tree order is maintained by comparing the timestamps of each issue, it ensures that the newer issues are placed on the right and older issues are placed on the left.

Where and How Data Is Called: I make use of the 'InOrderTraversal' method to retrieve the issues in a chronological order of their time stamps.

```
/// <summary>
/// Performs an in-order traversal of the tree and returns a list of issues.
/// </summary>
/// <returns></returns>
2 references
public List<IssueClass> InOrderTraversal()
{
    var issues = new List<IssueClass>();
    InOrderTraversal(Root, issues);
    return issues;
}
```

Example: When a user wants to view the list of submitted service issues before any filtering, it will display in a chronological order, that they were created in.

Why It's Good: The binary tree increases the efficiency of searching by making the lookup speed for issues using timestamp faster.

4. AVL Trees:

Role and Contribution:

A AVL Tree is like a self-balancing binary tree, where the difference between the hights of left and right subtrees of any node is at most 1 (BeyondVerse, 2023). That is why I used the AVL tree similar to the binary search tree, to minimize the retrieval time even more by maintaining a balanced structure of issues based on their time stamp.

Example:

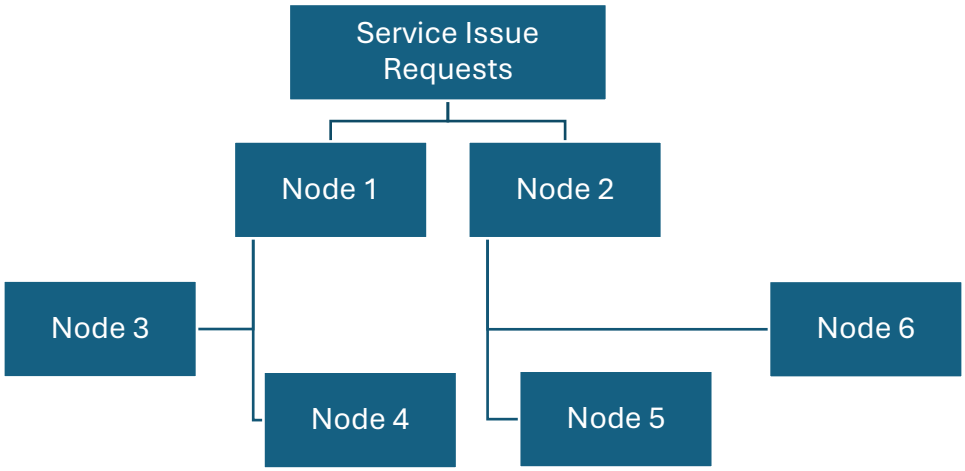

Figure 1.4. Basic AVL tree structure

Data Stored: The tree inserts data efficiently by making sure that each time a new issues is inserted, it balances the tree by using rotations (left and right), making sure that the hight difference between the left and right never exceeds 1. After every new inserted issue, the tree checks whether it has become unbalanced and if it has, it will perform the necessary rotations to ensure that the tree remains balanced.

Where and How Data Is Called: I make use of the 'InOrderTraversal' method to retrieve the issues in a chronological order of their time stamps like I do for the binary search tree.

Example: When a user wants to view the list of submitted service issues on the tracking page, in a data grid, I make use of the AVL tree, for efficient and fast retrieval, even though it does the same as the binary search tree. I also use it when the user wants to search through the list of submitted service issues.

Why It's Good: The AVL tree guarantees a balanced structure, which makes search, insert and deletion of certain issues faster and more efficient and the AVL tree also keeps the issues ordered by their time stamp (BeyondVerse, 2023).

5. Red-Black Trees:

Role and Contribution:

A Red-Black tree is like a self-balancing binary tree like the AVL tree but with less strict balancing rules (BeyondVerse, 2023). The balancing rules is just ensuring that no path from the root to any leaf node is more than twice the length of any other path, and no rebalancing is required (BeyondVerse, 2023). In the service requests, I use a Red-Black tree to ensure balancing and retrieval of data when tracking issue statuses and using the timestamp again.

Example:

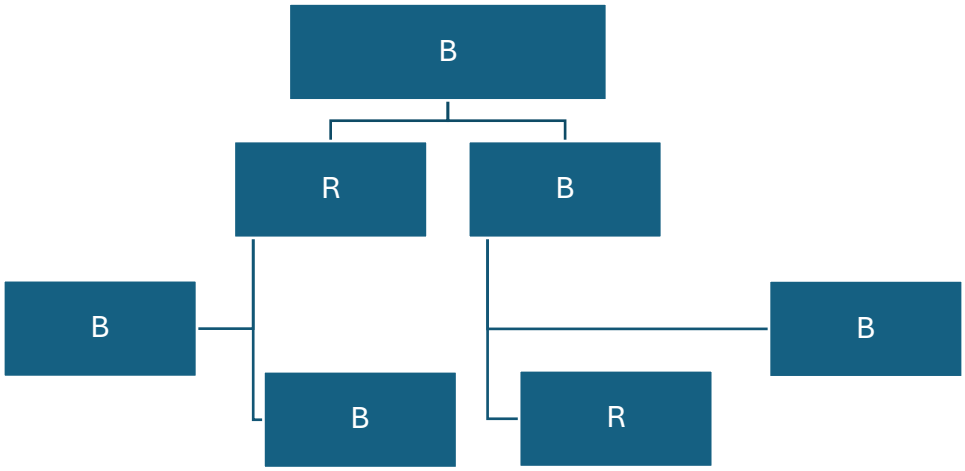

Figure 1.5. Basic Red-Black tree structure, where B = Black nodes and R = Red nodes

Data Stored: The data in the red-black tree is stored in nodes that contain the IssueClass object, I am using pointers to the left and right children allowing traversal through the tree structure and color flags to indicate which node is Red or Black.

Where and How Data Is Called: I make use of the red-black tree when I write to the other two (Binary search and AVL trees) to ensure that the tree remains balanced when entering new issues. I then use it in the tracking of issues.

Example: When a user wants to view the grouped time stamped issues of submitted service issues on the tracking page, in a line graph, I make use of the red-black tree, for efficient and fast retrieval, even though it does the same as the binary search tree and AVL tree.

Why It's Good: The red-black tree automatically maintains its balance after inserting issues, it makes searching for a specific issue faster (BeyondVerse, 2023).

6. Heaps:

Role and Contribution:

A Max Heap is a complete binary three that has heap properties (BeyondVerse, 2023). Heap property includes that the value of each node is greater than or equal to the value of its children, this makes the root node the maximum element in the heap. I am using the MaxHeap class that contains an internal list of IssueClass objects, to order them by the timestamp property and ensure that the latest timestamp is always at the root, this allowed me to easily retrieve the most recent issues quicker. I am also using a normal heap in my priority queue.

Example:

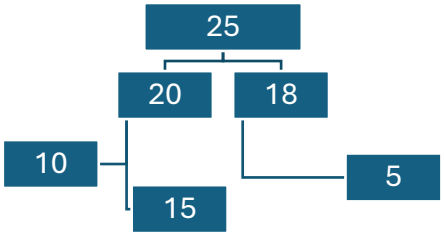

Figure 1.6. Basic structure of a Max Heap

Data Stored: The heap list holds the elements of the heap, where each element is an IssueClass object. The list maintains the heap property where the parent node's timestamp is always greater than or equal to its children's timestamps.

Where and How Data Is Called: I make use of the heap by sorting the issues by timestamp.

Example: When a user wants to filter the list of service issues by Date, it will use the heap to display a sorted list to the user.

Why It's Good: The heap makes sorting easier and ensures that the latest issue submitted can be returned, by using the ExtractMax() method, keeping the issues ordered and having no need for any additional pointer memories allows for space efficiency.

```
/// <summary>
/// Removes and returns the maximum element (root of the heap) and restores the heap property.
/// </summary>
/// <returns></returns>
3 references
public IssueClass ExtractMax()
{
    if (heap.Count == 0) return null;

    var max = heap[0];
    heap[0] = heap[heap.Count - 1];
    heap.RemoveAt(heap.Count - 1);

    HeapifyDown(0);
    return max;
}
```

7. Graphs and Graph Traversal:

Role and Contribution:

A Graph is a collection of nodes and edges whereas a Graph Traversal is the process of visiting all the nodes in that graph. In my service requests, a graph & graph traversal provided me with methods to add nodes, and edges, making it easier to represent and manipulate the graph

Data Stored: In my code, I have a Graph<T> class where I use a dictionary (Dictionary<T,List<T>>) to store the graph. The key represents nodes and the values in the list of connected edges.

```
/// <summary>
/// A dictionary that stores each vertex as a key
/// </summary>
private Dictionary<T, List<T>> _adjacencyList;

/// <summary>
/// Retrieves all issue dependencies in the graph
/// </summary>
/// <returns></returns>
1 reference
public Dictionary<IssueClass, List<IssueClass>> GetAllDependencies()
{
    return _issueGraph.GetGraph();
}
```

Where and How Data Is Called: I make use of the Graph and Graph Traversal to display a Pi chart to the user grouped with statuses of the issues, it shows the number and % of issues ‘Pending’, ‘Active’ and ‘Resolved’.

```
// · ♫----------------------------------------------------------------------------------------------------♫ · //
/// <summary>
/// Populates the dependency-based pie chart showing issue relationships and using Graph
/// </summary>
1 reference
private void PopulateDependencyPieChart()
{
    var allDependencies = _issueTracker.GetAllDependencies();

    var colors = new List<SolidColorBrush>
        {
            new SolidColorBrush(Colors.Blue),
            (SolidColorBrush)FindResource("greenSolidColorBrush"),
            new SolidColorBrush(Colors.Gray)
        };

    int colorIndex = 0;
```

…

```
    if (!dependencies.Any())
    {
        var statusName = entry.Key.Status ;

        var existingSeries = PieChartSeries.FirstOrDefault(p => p.Title.Contains(statusName));

        if (existingSeries != null)
        {
            existingSeries.Values[0] = (int)existingSeries.Values[0] + 1;
        }
```

```
            else
            {
                PieChartSeries.Add(new PieSeries
                {
                    Title = $"{statusName}",
                    Values = new ChartValues<int> { 1 },
                    DataLabels = true,
                    Fill = colors[colorIndex % colors.Count]
                });

                colorIndex++;
            }
        }
    }
}
```

Example: When a user wants to view a summary of all the service requests, and track their status, the data is represented in the form of a Pi chart using a graph and a graph traversal.

Why It's Good: A graph and graph traversal makes it easier to display a chart to the user with the correct data, it Is flexible, and it provides multiple different ways to retrieve the data needed for different problems using Traversal method such as DFS (Depth-First Search) and BFS (Breadth-First Search).

8. Minimum Spanning Tree (MST):

Role and Contribution:

A Minimum spanning tree is a subgraph that connects all the nodes with the least possible total edge weight. The MST that I implemented works together with my graph to find the subset of edges that connects all the nodes with minimum cost and avoid cycles. The MST algorithm that I used is Kruskal's Algorithm to find the relationships between the issue locations.

Data Stored: The data stored in the minimum spanning tree that I use is the relationship between locations.

Where and How Data Is Called: I make use of the MST by calling the Kruskal Algorithm and minimize the number of connections between service requests.

Example: When users want to view a filtered list by locations that they have entered, I determine the most efficient way to assign service requests to their location.

Why It's Good: The MST ensures that resources are optimally assigned, and it reduces the time spent to retrieve data needed for display.

Works Cited

BeyondVerse, 2023. *Tree Data Structures: A Deep Dive*. [Online]
Available at: https://medium.com/@beyond_verse/tree-data-structures-a-deep-dive-e593fdd1fcf2
[Accessed 18 November 2024].