I. Data Structure Design

Despite having to work with only ten patients and no explicit indication that the system would be used in a larger scale, I prefered a more practical look into the PICU Feeding System and assume the hospital would treat a great deal more than just ten patients, thus I opted for a dynamic data structure instead of a linear one.

The reason for this being the complexities of the linear data structure's main methods, including search, insertion (we do not consider deletion as the problem suggests no data will ever be deleted). Suppose N is the size of our data:

- Search: a variable would have to run from the start to the end of the structure to find the desired object. The worst-case scenario would then yield the complexity of O(N).
- Insertion: if we are inserting without considering the order of elements, the complexity would only be O(1). On the other hand, if we sort element as it is being inserted, the complexity would be O(N).

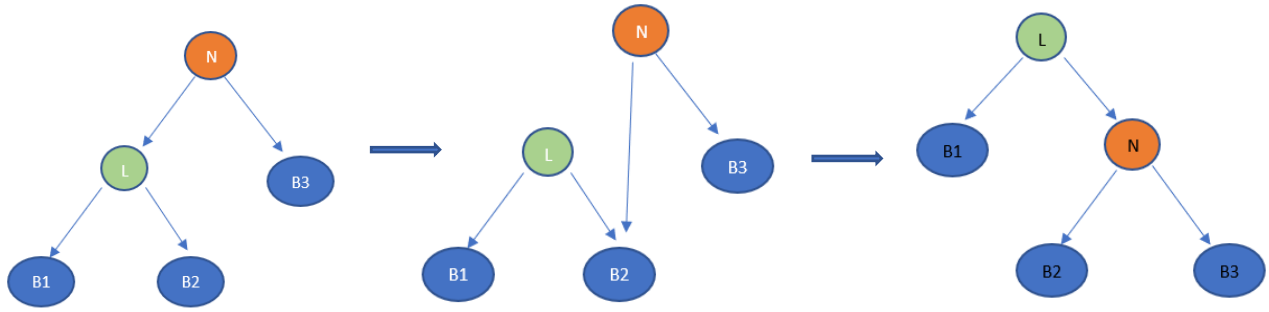
Binary search tree (BST) is quite popular among dynamic data structures, with its items being sorted in ascending order from left to right as they are inserted, making the task of producing a list of objects in a desired order much faster. Furthermore, the average complexities of search and insertion are only O(log(N)) on average. However, in the most unfortunate event of the BST being unbalanced, each of the above actions would require O(N) time complexity. And base on the aforementioned assumption, BST would not perform well.

In the end, I had decided to use the AVL tree structure. It is a self-balancing binary search tree, meaning the heights of two subtrees of any given node differ by at most one, if the difference is more than one, a rebalancing process is called to ensure this property.

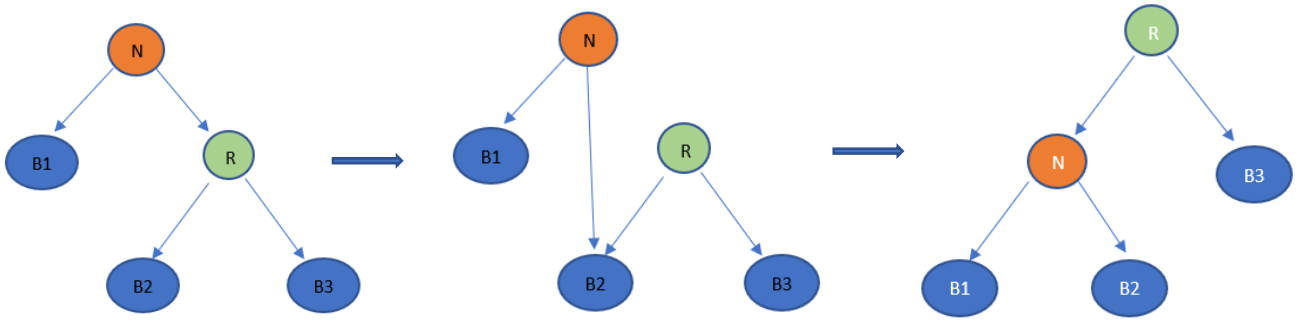
In the case of AVL tree, both search and insertion have the time complexities of O(log(N)) on average and in worst-case scenarios.

Rebalance:

1. Right Rotation:



2. Left Rotation



II. Main Methods

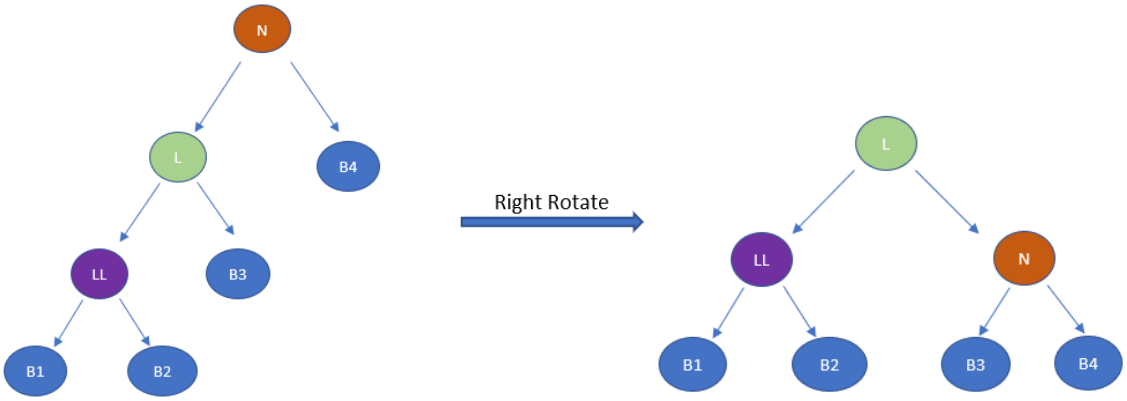
1. Search

- To look for a node using key in the AVL tree, we need to go through the tree recursively. For each recursion, a temporary root node is used for referencing, if key is smaller than that of root's key, do the recursion for the root's left branch, if not, do the recursion for the root's right branch.

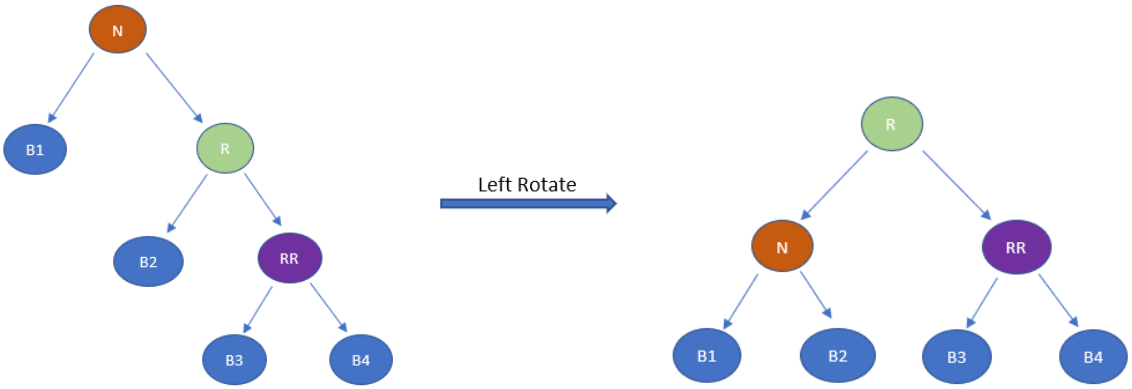
2. Insertion

- To insert a new node, first we perform normal BST insert, which includes recursively going through the tree to find the correct position and create a link to that node. Next, we need to check the height of the tree to see if it's unbalanced. If it is, perform the rebalance process.
- There are four possible cases we have to consider:

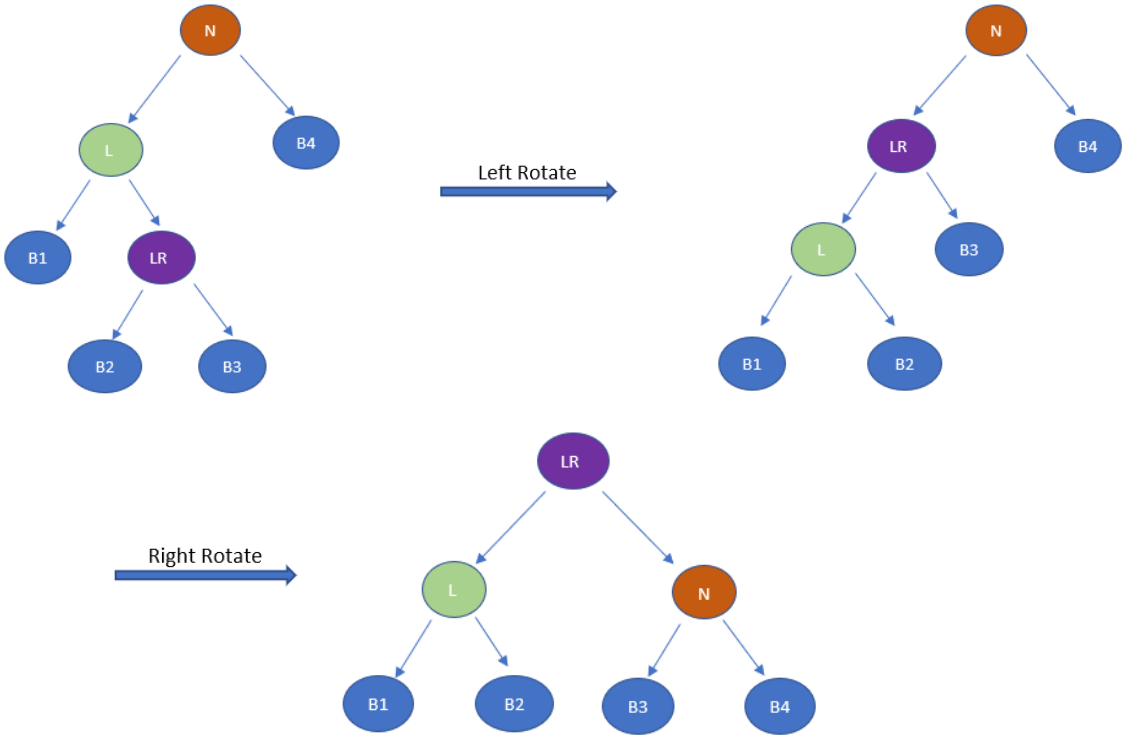
a. Left Left: Right Rotation



b. Right Right: Left Rotation



c. Left Right: Left Rotation -> Right Rotation



d. Right Left: Right Rotation -> Left Rotation