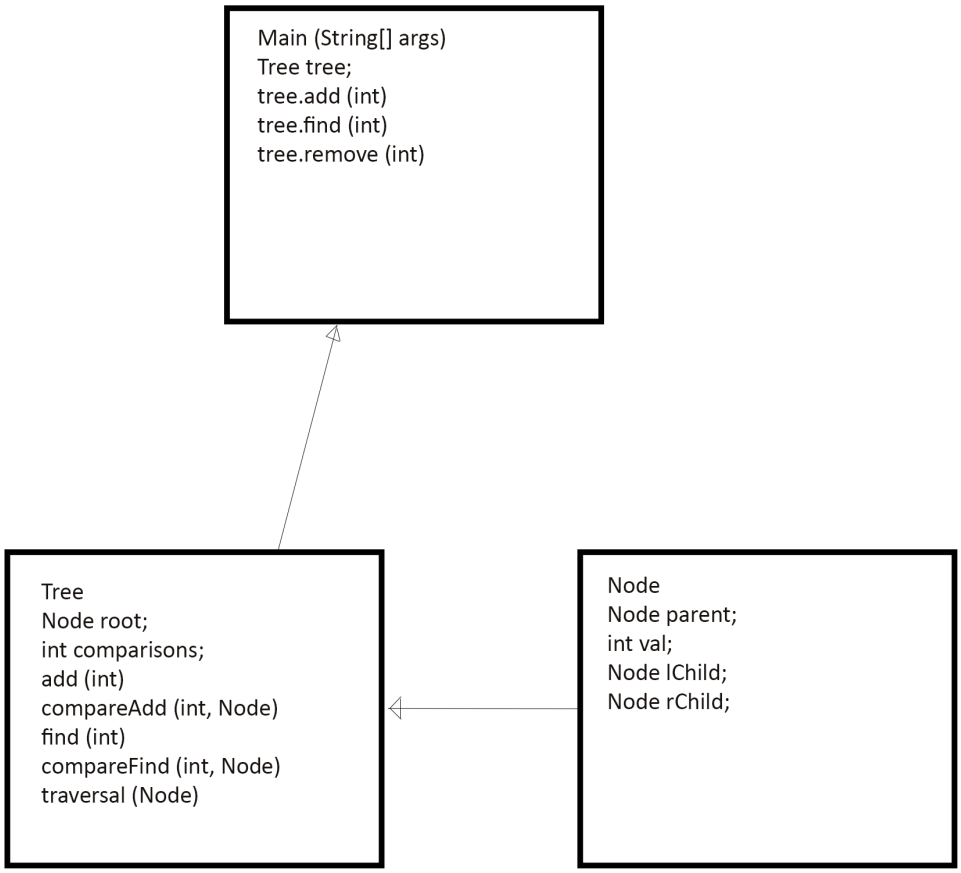
Coen 352 – Assignment 3

General Tree Structure

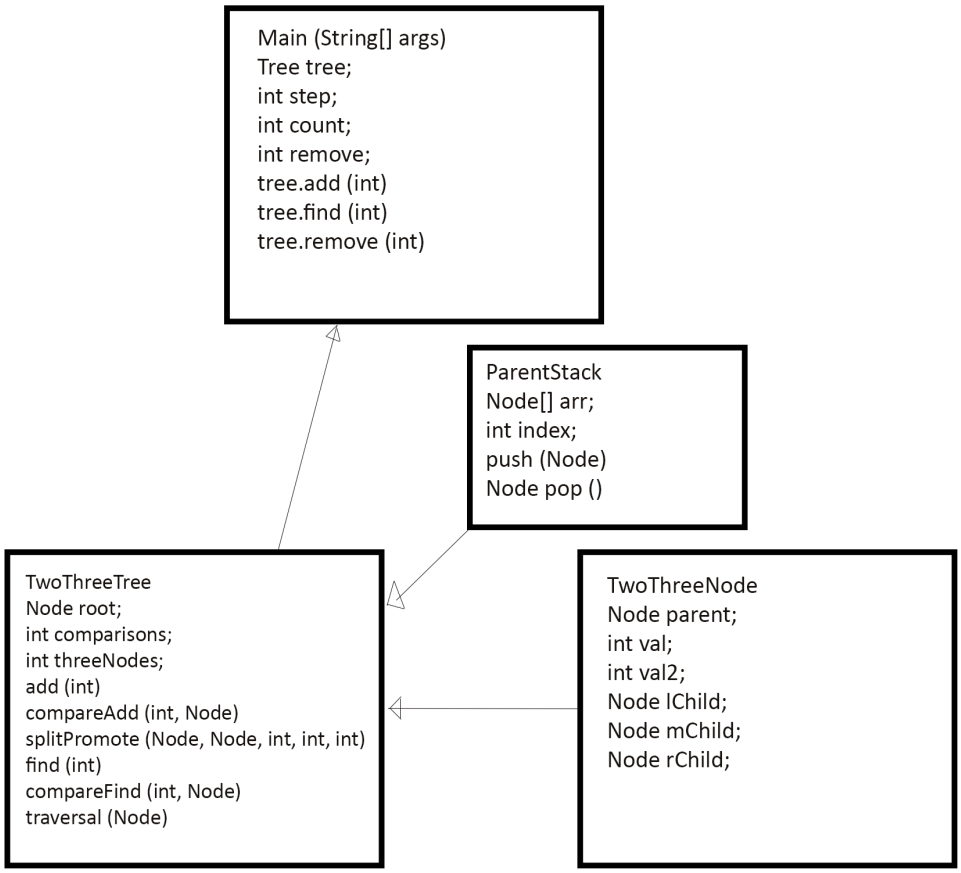


Discussion: This general Tree structure contains all of the shared functions between Binary Trees, Two-Three Trees and Red-Black Trees: add, find and remove. It also counts comparisons, and outputs an ordered traversal starting from the root Node.

It is only necessary for the tree to know its root Node, as every other Node will be pointed to by the successive children of the root. Each Node contains pointers to its parent and children, so the Tree can be traversed top-down or bottom-up through recursion.

compareAdd(), compareFind() and **traversal()** have been defined using tail recursion. The **add()** function first instantiates a root Node before calling **compareAdd(),** and the **find()** function checks for the existence of a root node before beginning the comparisons.

Two-Three Tree Structure



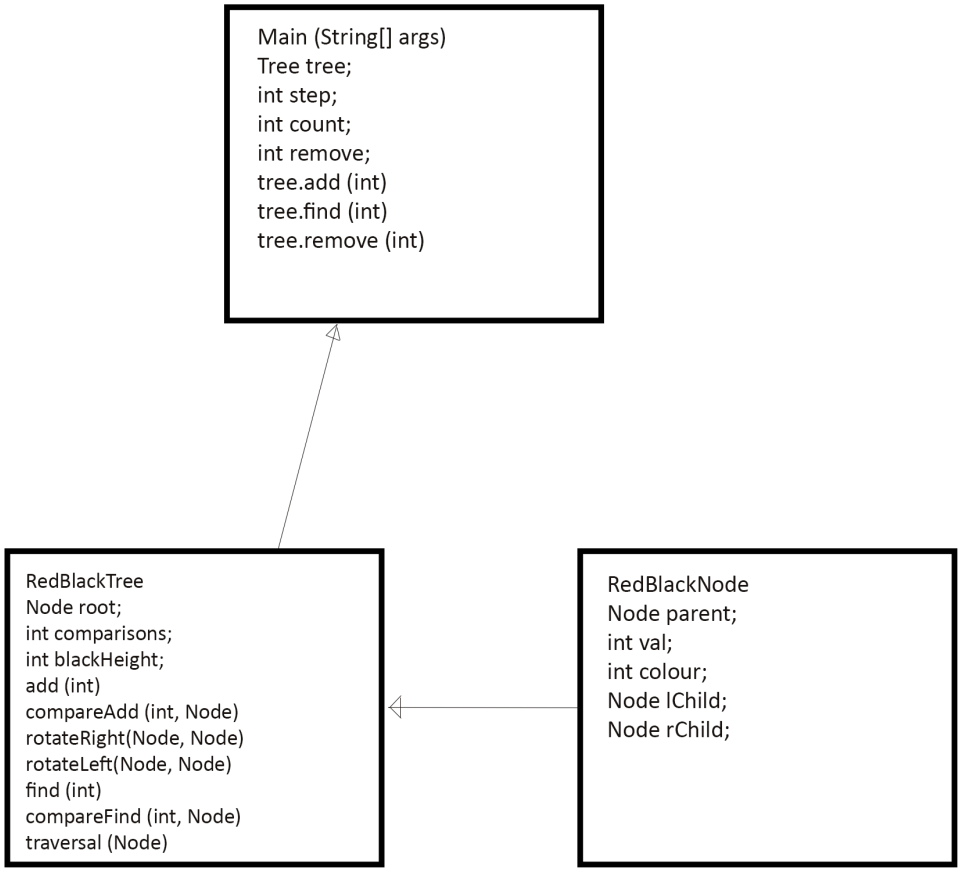
Discussion: In order to implement our Two-Three Tree, we first need two- and three-Node capability. For simplicity, we use the same class for both and leave the **mChild** null and assign -1 to **val2** to indicate a two-Node.

Next, for the **TwoThreeTree** object we include a **threeNode** counter and a **splitPromote()** function which takes a three-Node to be split, its parent Node and the left, middle and right values to be used to split the Node. The middle value is always promoted to a position in the parent node dictated by the child it came from. There are a number of different configurations for **splitPromote()** depending on if the Node in question has children, if it has a parent, and whether its parent is a two or a three-Node.

Also, during implementation it was found that while the top-down recursion in **compareAdd()** works just fine, the bottom-up recursion used by **splitPromote()** bizarrely requires each parent to be explicitly passed to it or the parent of the parent becomes a null reference. Therefore, the **ParentStack** was implemented to hold each successive parent reference. As **compareAdd()**

moves down the Tree, each parent gets pushed to the stack in order to be used for **splitPromote()** later on, in the case of multiple **splitPromote()** recursive calls.

Red-Black Tree Structure:



Discussion: To implement a Red-Black Tree, we first need a Node with a **colour** bit: in this case 0 will indicate a black Node and 1 will indicate a red Node.

Next, in order to create a self-balancing tree, we must be able to rotate, or re-assign colours and locations of Nodes in the tree. This is why we have **rotateRight()** and **rotateLeft()** in addition to the **blackHeight** counter which is used to determine if a branch is longer or shorter than the uniform height of the Tree, measured by the number of black Nodes between the root Node and the leaves.

Why is the Two-Three Tree Better? Firstly, although the asymptotic complexity each are the same for space **(O(n))** and insert/find/delete **(O(log(n))**, the Two-Three Tree has leaves that are always on the same level, and generally works by adding height to the top through recursive

splitPromote(), making it very easy to generalize the functionality and access times. In contrast, the Red-Black Tree requires extra work to check for balance and rotate individual Node clusters in order to keep a balanced Tree, in addition to creating issues with mass re-colourings in large rotations. Personally I find double rotations hard to wrap my mind around.

Another reason why Two-Three Trees are nicer with this implementation is due to the two and three-Nodes being able to hold more values and children, resulting in a more compact Tree. While the operational asymptotic complexity is still log(n), this could potentially result in a faster log(n) than the Red-Black Tree.

In general, deciding to use recursive methods in our general tree structure lends itself better to the Two-Three Tree implementation over the Red-Black implementation, which would require more lines of code to be written.