# Data Structures & AlgoritHms ASsignment

Team PohSeng#1 (suggested by Francis)

|  |  |
| --- | --- |
| Team Member | Student ID |
| Francis Koh | S10172072G |
| Matthias Ngeo | S10172190F |

## Description of Application

The application allows users to manipulate and interact with an AVL tree implementation through a console window. It provides facilities for basic operations including, searching, addition and removal of elements, retrieving the value of the node at a specified index, and displaying the elements in a tree in an ascending order. Additionally, the AVL tree implementation includes support for duplicate values and iterators for traversing through the AVL tree either by ascending order of the values or level-by-level. Furthermore, the application validates and handles user input. The use of recursion in implementing the AVL tree was deliberately avoided in favour of iteration to optimise the performance of the addition and removal algorithms in terms of both memory consumption and processing speed which arises from the programme not having to constantly allocate and deallocate the stack and avoiding high context switching when each method is called recursively.

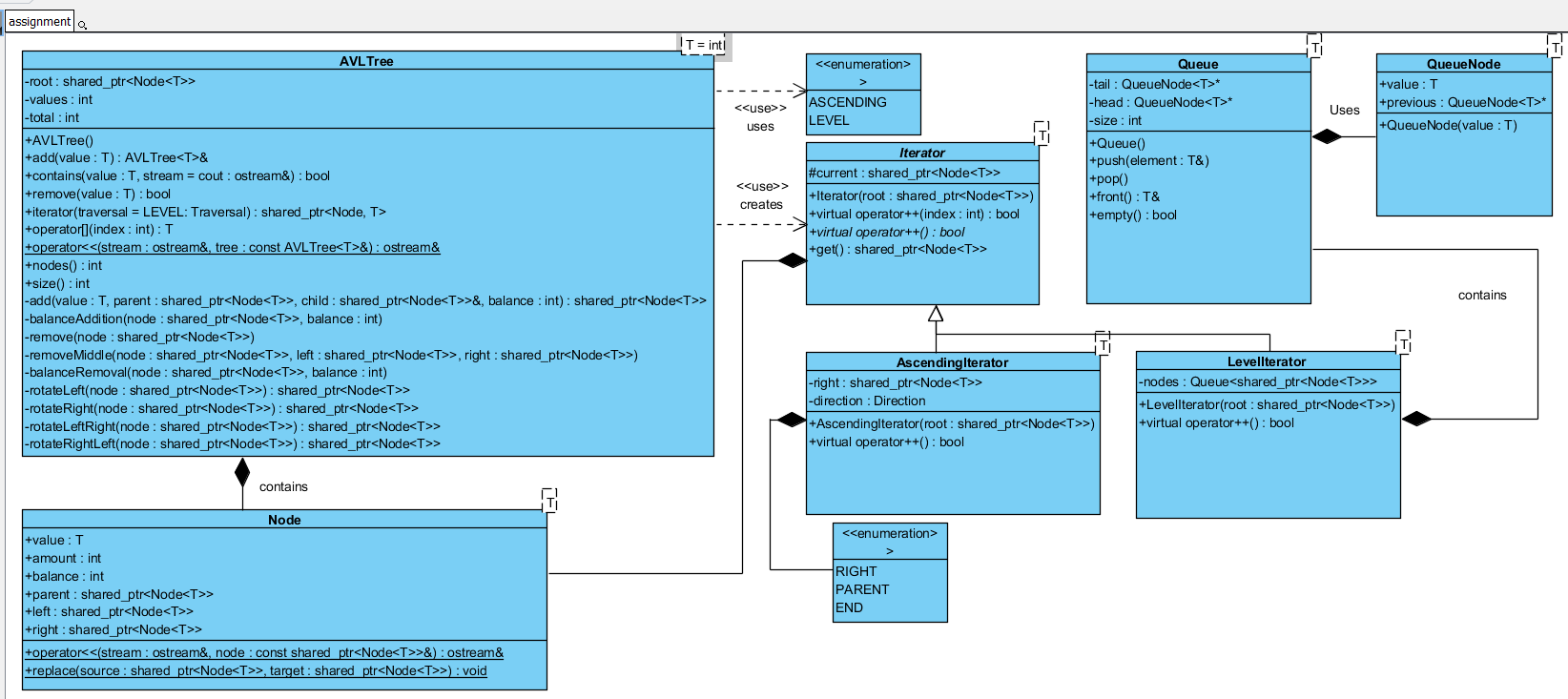
## Roles and contributions

|  |  |
| --- | --- |
| Roles | Contributions |
| Francis Koh | Search for value |
| Display value of nth node (level-by-level iterator) |
| Implementation of node |
| Input validation |
| Testing |
| Documentation of implemented functionality |
| Report |
| Matthias Ngeo | Addition (balancing addition) |
| Removal (balancing removal) |
| Implementation of queue |
| Implementation of node-related functionality |
| Display values of tree in ascending order (ascending iterator) |
| Documentation of implemented functionality |
| Report |

## Instructions & Compiling

The application was compiled against the C++ 14 standard for 64-bit machines using the GCC G++ Compiler 7.2.0 from the MySYS2 tool collection. The application was compiled with the following compiler flags, “g++ -m64 -O2 -c -g -Werror -std=c++14”.

## Class diagram



## Data structures and Algorithms Descriptions & Explanations

### AVL Tree

An AVL tree is a balanced binary search tree which balances itself after insertion and removal, hence providing a guarantee that the AVL tree will not degrade into a linked-list. The implementation holds a reference to the root node of the AVL tree. Each node contains a value and balance, and holds a reference to its parent, and left and right child. The basic operations (add, remove, contains) are implemented non-recursively and provides a guaranteed time complexity of O(log(n)) while the additional operations provide a guaranteed time complexity of O(n). The basic operations all share a similar iterative algorithm which iterates through the nodes in the tree starting from the root and selecting the next node to visit based on the comparison between the specified value and the current node’s value. Each operation differs slightly from the general algorithm in how it handles each case, (i.e. larger than, smaller than or equal to). Addition creates the next node if null, removal checks if the current node is equal to the specified value and removes the current node and contains returns true if the current node is equal to the specified value.

An AVL tree implementation was preferred over the standard binary search tree due to the tree balancing itself after each insertion and removal which satisfies the requirement of preventing the binary search tree from degenerating into a linked list. All the operations are implemented non-reclusively to improve the performance of each respective operation both in terms of CPU usage and memory consumption. Although tail-call optimisation could be performed by the compiler to remove the performance penalties associated with the allocating and deallocating the call stack, an iterative approach allows optimisations specific to an iterative approach such as variable-hoisting to be performed which may not be available to recursive alternatives, hence for the most parts, an iterative implementation will outperform recursive alternatives in most cases and the reason it was chosen. Not to mention the possibility of stack overflows for trees with a sufficiently large number of nodes. The algorithms were selected due to an improved time complexity of O(log(n)) as compared to iterating through every single node which would have a time complexity of O(n). Each node additionally tracks its parent to support the iterative implementation of the operations which allows it to traverse up the tree without having to either store the parents of the node in an external data structure or resort to recursion.

### Queue

A queue is First-in-First-Out (FIFO) data structure in which elements are stored in the order in which they were inserted. The tail of the queue is that element that has been in the queue the shortest time. New elements are inserted at the tail of the queue and the retrieval operations obtain elements from the head of the queue. This implementation uses a singly-linked-list implementation and provides a guaranteed time complexity of approximately O(1) for basic operations (push, front, pop and size). Each node in the queue stores the value and holds a reference to the previous element with the previous element having been inserted directly after the node.

The queue in this context was used to store the nodes for next traversal in the level-by-level iterator. The iterator does not require random-index access hence a queue was chosen over a linked-list or any other similar data structures which support random-index access. The implementation of the queue uses a singly-linked-list over a doubly linked-list due to the queue not having to support retrieval from the tail of the queue or iteration of the elements which are the principal reasons behind implementing a queue using a doubly-linked-list, hence the additional effort required to implement the doubly-linked was unjustified and not chosen. Similarly, a linked-list approach was favoured over an array-based implementation due to the additional complexity and housekeeping an array-based implementation would present.

## appendix

* Class Diagram - <https://i.imgur.com/fNgPoPD.png>

## References

*AVL Tree | Set 1 (Insertion) - GeeksforGeeks*. (2018). *GeeksforGeeks*. Retrieved 21 January 2018, from <https://www.geeksforgeeks.org/avl-tree-set-1-insertion/>

*AVL Tree | Set 2 (Deletion) - GeeksforGeeks*. (2018). *GeeksforGeeks*. Retrieved 21 January 2018, from https://www.geeksforgeeks.org/avl-tree-set-2-deletion/