# 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.

## additional features

* AVL tree implementation
* Allowing duplicate values
* Non-recursive implementation to optimise performance
* Optimised tree balancing, i.e. nodes storing its own balance, stops balancing if balance of node is 0
* Exposing iterators for traversing the elements in the AVL tree
* Robustness
* User validation

## 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 |

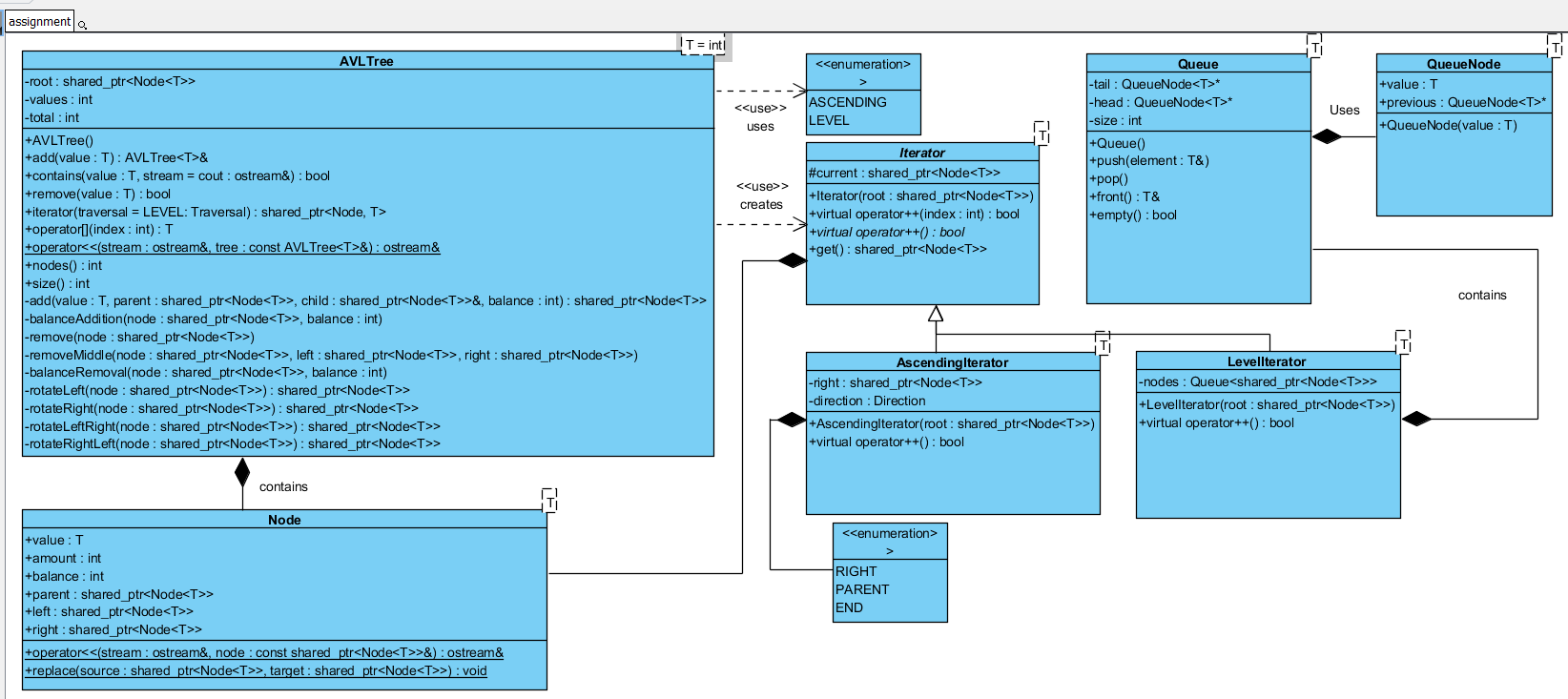
## COMPILING

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 and was compiled with the following compiler flags, “g++ -m64 -O2 -c -g -Werror -std=c++14”. It relies on the make build system, and the appropriate makefile is contained within the project. The compiled programme may be found within the {root}/dist/strict/MinGW-Windows/ folder.

## Usage

|  |  |  |
| --- | --- | --- |
| Option | Name | Description |
| 1 | Add a value | Prompts the user for a value before adding the specified value to the tree if valid, otherwise prompts the user again. |
| 2 | Remove a value | Prompts the user for the value to remove, prompting the user again if the value is invalid, before attempting to remove the value from the tree. Displays an error message if the value was not found; else a success message. |
| 3 | Search for a value | Prompts the user for the value to search for, prompting the user again if the value is invalid,  Displays a message indicating whether the value was found in the tree. |
| 4 | Get value at index | Checks if the tree is empty and displays an error message if empty. Otherwise prompts the user for the index of the node.  If the index is less than 0 or greater than the number of nodes in the tree, display an error message and prompt the user again; else displays the value at the specified index |
| 5 | Display values | Displays the values, including duplicate values in the tree in an ascending order. |
| 6 | Exit | Allows the user to exit the program |

## 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 along with an amount counter which allows the tree to include duplicate values. 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).

### Basic operations

The add method first creates the root if the tree is empty; else iterates through the nodes in the tree starting from the root while the current node is not null. If the value is either larger than or smaller than the current node, creating and adding the child node with the specified value and parent before balancing the tree if the child is null; else sets the next node as either the left or right child. If the values are the same, increase the amount of the current node and return.

Balancing of additions first starts with iterating through the parents of the nodes, starting from the specified node and updates the balance each time before delegating rotation to the appropriate method which depends on the balance. A balance of 0 indicates that the tree has been fully balanced and the loop terminates.

The remove method iterates through the nodes in the tree starting from the root while the current node is not null. If the node is smaller than or larger than the value set the respective child nodes as the next node. Otherwise, if the node is equal to the specified value and the node amount is more than decrease the amount and return. If the amount was equal to 1, it iterates through the left children of the specified node's right child while it exists before replacing the specified node with the left-most child if the right child has a left child; else replaces the specified node with its right child.

Balancing of removals first starts with iterating through the parents of the nodes, starting from the specified node and updates the balance each time before delegating rotation to the appropriate method which depends on the balance.

Similarly, the contains method iterates through the nodes in the tree starting from the root while the current node is not null. If the value is either larger than or smaller than the current node set the next node as the right and left node respectively. Returns true if the value is equal to the current node and false if the node terminates and no node was equal to the specified value.

### Additional operations

The random-access functionality creates a level-by-level iterator and iterates through the nodes until the specified index has been reached and the value of the node is returned and the implementation ignores duplicate values and treats duplicate values as a single node/iteration.

Likewise, the print functionality creates an ascending iterator and iterates through the nodes and prints them to the ostream.

### Iterators

This AVL tree implementation exposes two different iterator implementations of the elements in the tree, namely an iterator which traverses the tree in ascending order and level-by-level which used internally by the additional operations to traverse the tree.

Internally the ascending iterator uses an algorithm which first determines the direction of the traversal. If the direction is RIGHT, it sets the right child of the previous element as the current element.

After which, it sets the left-most leaf element of the current element as the next node, and sets the direction to PARENT if the current element has no right child; else sets the right element as the right child of the current element, and always returns true. If the direction is PARENT, it iterates through the parents of the elements starting from the current element while the element has a parent. If the left child of the element is equal to the current element, set the right child of the current element as the next right element and sets the direction to RIGHT if the next right element exists and returns true. Otherwise, if the element has no parent, set the direction as END and return false. Otherwise returns false if the direction is END.

Another supported iterator is the level-by-level iterator which creates a queue checks if the queue is not empty, set the element at the head of the queue as the current element and pop the queue before adding the left and right child of the current element to the queue if they exist respectively.

### Explanation

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 for the basic operations 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 amount to support duplicate values in the tree and its parent to support the iterative implementation of the AscendingIterator 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.

The additional operations were implemented using iterators as it hides the implementation of the iteration from the operations itself which increases the cohesion of the operations by allowing it to focus on the processing of the elements in the iteration rather than the actual iteration itself and decouples the iteration implementation from the additional operations. Separating the iteration algorithms from the specific operations allows the iteration algorithms to be reused without the need to rewrite them. Exposing the iterators through the iterator method favours the Liskov's Substitution Principle (LSP) and allows the user to define their own processing logic without exposure to the implementation.

### 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.

### Basic operations

The push method creates a new QueueNode with the specified value. If the queue is empty set it as both the head and tail of the queue; else if the head is equal to the queue, set the previous node of the head as the new node and set the tail as the node. Otherwise sets the previous node of the tail as the node before then setting it as the tail. Always increments the size afterwards.

Popping the queue first checks if the queue is empty and returns if true; else if the size is equal to 1 set both the head and tail as null; else sets the previous node of the head as the new head and remove the old head.

The front method throws an invalid\_argument exception if empty; else returns a reference to the value of the head.

### Explanation

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/