Main

The driver file acts as a parser. It creates a Graph and an MST object and calls methods depending on given inputs. Illegal arguments are checked in the driver file. In this design document, let "V" represent the number of vertices in the graph, let "E" represent the number of edges, and let "a" represent an arbitrary number of adjacent nodes to any given node. Running times for AVL Trees and Binary Heaps were found on LEARN [1]. Here is an analysis of the data structures used to solve this problem and their implementations:

Graph

The graphs are implemented with an adjacency list wherein the index represents a node, and each entry a pointer to the head of an AVL tree, where the key is the adjacent nodes. Each node in the AVL tree stores the weight of the edge to the adjacent node. For deletion, each node should store every edge. Therefore, insertion should insert the forward and backward edges. When a node is deleted, find and remove the current node from the adjacency list for each adjacent node, then delete the adjacency list of the node to be deleted. Table 1 shows a fundamental analysis of why this method was chosen.

The graph constructor initialized a pointer to an array of AVL trees and created an AVL tree at each array index. The destructor deletes the AVL tree at each index and then deletes the pointer to the array.

Data Structure	Search	Insertion (Insert + Search)	Deletion (n Searches and Delete)
Unsorted Vectors	E	$1 + E$	V^2
Sorted Vectors	$log(E)$	E	V^2
Linked List	E	$1 + E$	V^2
AVL Trees	$log(E)$	$log(E)$	$V \cdot log(E)$

Table 1: The time complexity of graphs implemented using different data structures.

AVL Trees

AVL Trees are used in graph implementation to increase search efficiency. In addition, as both insertion and deletion require searching, this decreases the running time as opposed to an array-based implementation.

The nodes in the AVL tree store the adjacent vertex and the weight of the edge to said vertex. In addition, nodes also store pointers to left and right children and parents. First, nodes are initialized with vertex, weight, and parent. Then, the node destructor recursively deletes the node's children and itself.

The AVL tree stores a pointer to its root. The constructor initializes the root to nullptr, and the destructor deletes the root. The AVL tree implemented the standard methods, as shown in the UML diagram in Figure 1: UML Class Diagram.

Graph Running Time Analysis

Insertion: Performing an insertion takes $\mathrm{O}(\log(E))$ time. When an insert is called on an edge, the function will enter the array at the index of one node, $\mathrm{O}(1)$, and try inserting that edge into the graph using binary search. If the node is found during traversal, the insertion fails, and the insertion in the other direction does not occur. If the node does not exist, insert the node into the proper position. As AVL trees are used to store adjacent nodes, after insertion, a rebalancing should occur, $\mathrm{O}(\log(E))$.

Deletion: Performing a deletion takes $\mathrm{O}(\mathrm{V} \cdot \log(E))$ time. When a delete is called on a vertex, the function will check if the AVL Tree at the vertex is empty. If not, the function deletes the AVL tree at that vertex, $\mathrm{O}(E)$, and creates a new one,

$O(1)$. Then, each vertex in the array is checked to see if the vertex is adjacent and, if so, the node is deleted from that tree, $O(V \cdot \log(E))$.

Print Adjacent: Performing a print adjacent takes $O(E)$ time. If a vertex exists at that array, an in-order traversal is performed.

MST

The MST is implemented as a graph where each tree has only one node, representing the minimum cost path.

The MST algorithm used in the program is Prim's algorithm:

Initialize an empty MST and a set of visited vertices. Choose an arbitrary vertex to add to the MST and initialize a list of visited vertices. Next, initialize a priority queue of edges with all the edges adjacent to the starting vertex. The weight of the edge gives priority. While the priority queue is not empty, do the following: Extract the edge with the smallest weight from the priority queue. If both endpoints of the edge are already in the visited set, discard the edge and continue to the next iteration. Otherwise, add the unvisited endpoint to the MST and the visited list. Finally, update the priority queue by adding all the edges adjacent to the new vertex that connect to vertices outside the MST.

Binary Heaps

Binary heaps are used in MST implementation to create a priority queue. These are implemented to allow $O(\log(n))$ running time to find the minimum value. The priority queue implemented uses tuples to store vertex, adjacent vertex and weight, with weight as the key. The heap was implemented as an array, with parents and children being accessible using indices.

The construct for heaps creates a new queue array with a max size of E and initializes the current and maximum sizes. Finally, the destructor deletes the queue array. The binary heap implemented the standard methods, as shown in the UML diagram in Figure 1: UML Class Diagram, and initialization.

Initialization adds edges from a vertex to the queue by calling insert node in post-order for each adjacency list. This runs in $O(E \log V)$ as, in the worst case, all edges will need to be inserted, and to insert takes $O(\log V)$ time – from heapify.

MST Running Time Analysis

MST: Performing MST takes $O(E \log V)$ time. To complete the MST, Prim's algorithm must extract the minimum vertex, $O(\log V)$, V times. Additionally, Prim's algorithm needs to initialize the priority queue with the graph's edges, $O(E \log V)$. Since the number of edges has an upper bound of V^2, the worst-case running time is $O(E \log V)$.

Cost: Performing the cost calculation takes $O(E \log V)$ time. To compute the cost, the MST must be calculated. Then the cost sums the weight of each edge in the MST.

References

[1] The University of Waterloo, "ECE 250," [Online]. Available: https://learn.uwaterloo.ca/d2l/home/880203.

Appendix

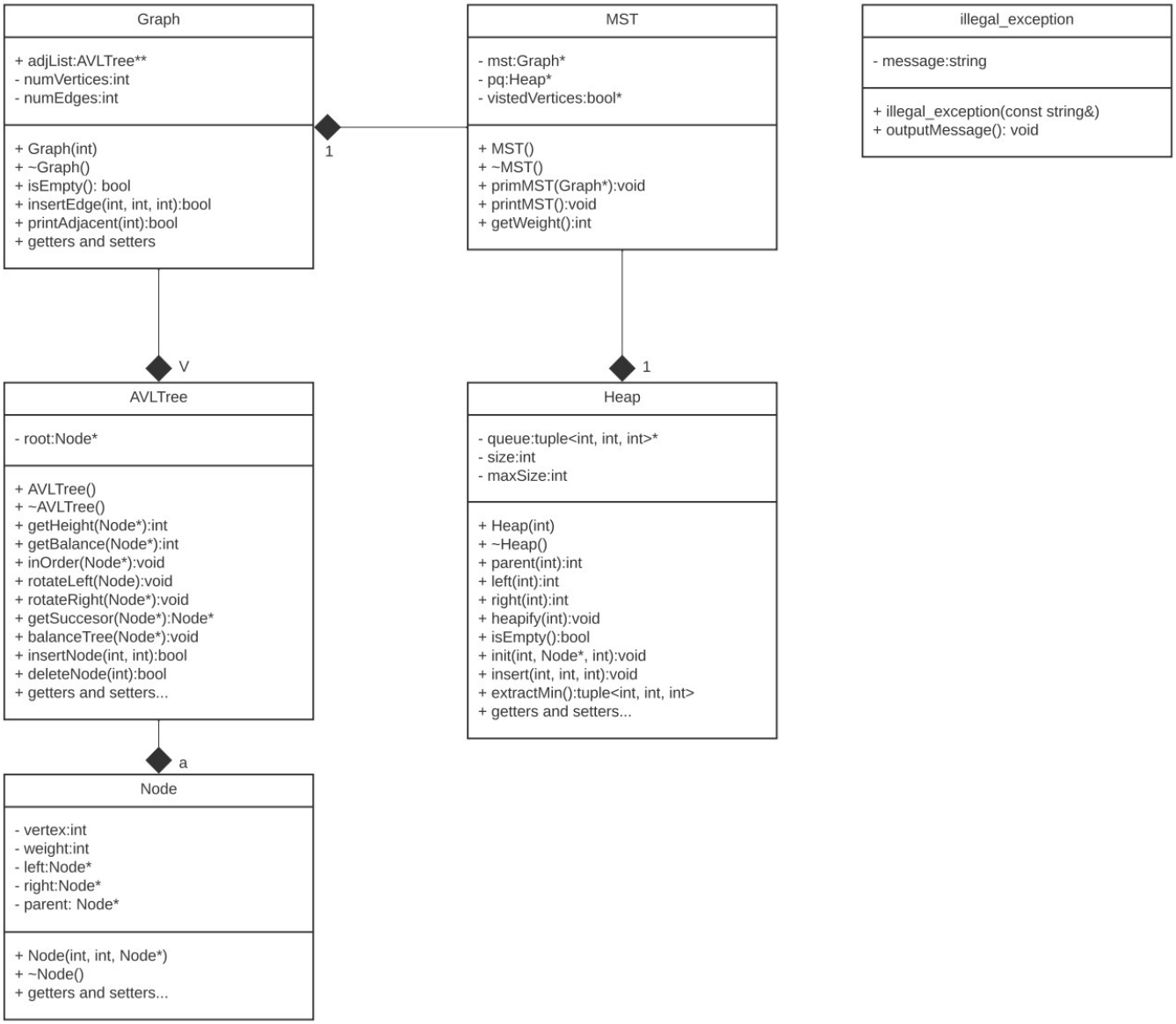

Figure 1: UML Class Diagram