MAIN:

The main.cpp file has three principal responsibilities.

The first one is to read from the commands.txt file line by line and understand which data structure, method (command) needs to be used and what the parameters (if needed) are.

The command can be: BUILD, GETSIZE, FINDMIN, FINDMAX, SEARCH, COMPUTESHORTESTPATH, COMPUTESPANNINGTREE, FINDCONNECTEDCOMPONENTS, INSERT, DELETE, DELETEMIN, DELETEMAX

The data structures can be: MINHEAP, MAXHEAP, AVLTREE, GRAPH, HASHTABLE

We use std::streams and parse/read the requested command, then the data structure and if needed the parameter that could be a filename or a number.

For example if the line is "INSERT MINHEAP number" the data structure is minheap the command is insert and the parameter is number. Each data structure has its own class so in the above example the insert method from class minheap will be called with number as a parameter.

The second responsibility of the main file is to measure the duration ... each command takes to execute To measure the execution time we simply get the time before and after the calls and subtract them.

(execution time=ending time-starting time).

The timer precision is in milliseconds and we return a double value. We only measure only the time required to run the execution of the requested method and not the pre-processing phase.

The third responsibility of the main file is stream the results if needed to the output.txt file (we append not overwrite). This is done using the 3 output methods that are in the main file.

- One to insert a number and the execution time (it is needed on commands like GETSIZE MINHEAP),
- One to insert two numbers and the execution time (this is only used for the GETSIZE GRAPH command) and
- One to insert a string and the execution time (the command SEARCH of the AVLTREE and GRAPH) because SUCCESS or FAILLURE must be put on the output.txt file.

Each data structure is declared as its own library (.ccp and .h files) and is included by the main as it is responsible of orchestrating the execution.

MINHEAP:

A minheap's property is that the value of each node must always be greater than the value of its parent meaning that the minimum-value element will always be at the root.

The minheap has 8 methods implemented and 3 class variables:

The variables:

->count is the number of elements contained in the heap it is raised by one at every insertion and decreased by one at every deletion from the heap (for example deleteMin method).

->alloc_size is the size of the dynamic memory allocated that holds the minheap.

->heap is the heap where the elements (integers) from the file are stored accordingly

The methods:

->Minheap(filename) is in charge of building the heap from a file given as a parameter while respecting the heaps properties

->insert(num) inserts the element given as a parameter in the correct index while respecting the heap's property. It also keeps track of the count of elements at each insert as to not need to iterate over it when the size is requested

->getSize() returns the size of the heap (by returning the count variable).

->findMin() returns smallest number on the heap which is always the root (minheap's property).

->deleteMin() deletes the smallest number of the heap which is the root by swapping the root with the last node and deleting it. The new root is then swapped with its children until the minheap property is satisfied again.

->getLeft(place) returns the left child's index

->getRight(place) returns the right child's index

->getParent(place) returns the parent's index

MAXHEAP:

A maxheap's property is that the value of each node must always be smaller than the value of its parent meaning that the maximum-value element will always be at the root.

The maxheap has 8 methods implemented and 3 class variables:

The variables:

->count is the number of elements contained in the heap it is raised by one at every insertion and decreased by one at every deletion from the heap (for example the deleteMax method)

->alloc_size is the size of the dynamic memory allocated that holds the maxheap.

->heap is the heap where the elements (integers) from the file are stored accordingly

The methods:

->Maxheap(filename) is in charge of building the heap from a file given as a parameter while respecting the heaps properties.

->insert(num) inserts the element given as a parameter in the correct index while respecting the heap's property. It also keeps track of the count of elements at each insert as to not need to iterate over it when the size is requested.

->getSize() returns the size of the heap (by returning the count variable).

->findMax() returns largest number on the heap which is always the root (maxheap's property).

->deleteMax() deletes the largest number of the heap which is the root by swapping the root with the last node and deleting it. The new root is then swapped with its children until the maxheap's property is satisfied again.

->getLeft(place) returns the left child's index.

->getRight(place) returns the right child's index.

->getParent(place) returns the parent's index.

AVLTREE:

An AVL is a data structure where each Node has 2 children. It's property is that the left child is always smaller than the parent Node and the right child is always bigger. Each node has a height and a balance. The height is how far the Node is from the last Node. For example the last Node's height is 0,its parent's is 1 etc.

The balance is (Left_child's_height)-(Right_child's_height) it must be from -1 to 1 or else we have an unbalanced tree. In case of an unbalanced tree we must rebalance it by rotating. There are four different rotations depending on the balance and the value of the key left rotation, right rotation ,left-right rotation (which consists of a left and then a right rotation) and right-left rotation (which consists of a right and the left rotation).

The avltree consists of:

->AVL(filename) is the constructor, inside this method we call the insert method.

->insert(num) is a method that puts the index to the correct place of the tree and finds out its height it is also responsible to see if rebalancing is needed and to find out which of the 4 methods of rotation needs to be used.

->getSize() returns the size of the avltree by returning the count variable.

->findMin() finds and returns the minimum node by searching for the last left node

->search(number) searches for the number given as a parameter on the avl tree by comparing if it is bigger or smaller than the current compared node, if it is bigger it moves to the right node, if it is smaller it moves to the left node. It returns true if the number was found or else it returns false.

->Delete(num) deletes a number (num) from the avltree and makes sure that the property of the avltree is right.

->getheight(Node *n) returns the node's given in the parameter height

->getbalance(Node *n) returns the node'd given in the parameter balance (balance=(Left_child's_height)-(Right_child's_height))

Rotations:

->rightR is responsible for the right rotation

->leftR is responsible for the left rotation

GRAPH:

A graph is composed of a set of vertices (nodes) which are connected with edges. Each edge has an assigned weight.

To represent and implement a graph we use two arrays:

The NodeList matrix is a Boolean array representing the presence of a node in the graph. At first all its positions are False, when the insertion of an element happens NodeList[element] is set to True

The AdjMatrix is a two dimensional array used to represent the connections within the graph. The weights of the edges are stored at the corresponding intersection. As we are working with undirected graph we mirror the connections. e.g. by setting the weight to both adjMatrix[n1][n2] and adjMatrix[n2][n1]

The methods of the graph:

->Graph(filename) is the constructor of the graph. At first we make two matrixes NodeList of type bool and AdjMatrix (2D) of type int. We make sure to allocate the needed space for the arrays. We then read 3 numbers from the file (filename) and insert them.

->insert(n,n2,w) is a method that checks if reallocation is needed and makes sure to insert the right information into the NodeList and the AdjMatrix

->Getedges()->returns the number of edges in the graph. Because for every edge two positions are filled (AdjMatrix[number1][number2] and AdjMatrix[number2][number1]) we need to divide the number of filled positions by 2 to find and return the number of edges.

->getSize()->returns the number of nodes in the graph by searching in the NodeList and counts how many positions are True. This will also be the number of nodes of the graph

->removeEdge(number 1,number 2)->this method removes the edge (connection) between the nodes given as a parameter by setting the AdjMatrix for those nodes back to zero AdjMatrix[number1][number2]=0 and AdjMatrix[number2][number1]=0.

->spaningTree() A minimum spanning tree is found by connecting all nodes with the minimum number of edges, it also makes sure that no circles are done. This method calculates the cost (weight) of the minimum spanning tree of the graph and returns it

->ShortestPath(number1,number2) uses the DFS algorithm to first find if a path exists and then Dijkstra's algorithm to find the shortest path.

->connectedComponents() a number of nodes connected by edges form a component. More than one components can co-exist in a graph this method calculates and returns the number of its connected components using the DFS method.

->DFS() the DFS is an algorithm that searches through graph data structures and is needed for the shortest Path and connected Components methods

->Dijkstra(int v1,int v2) this algorithm is used in the ShortestPath method, it is responsible for finding the shortest paths between nodes (v1,v2) according to their weights.

->recdij(int pos, int *pathvalue) is used in the Dijkstra's algorithm it figures out the shortest paths from one vertex to all other vertices in a graph. It looks at each neighboring vertex, and if a shorter path through the current one is found, the shortest path value is updated and repeats this process for the next vertex until all vertices are checked

->Search(number) looks for the number in the parameter on NodeList if found it returns the position, if not it returns -1.It is used in removeEdge, shortestpath methods and Dijkstra's algorithm.

HASHTABLE:

A hash table is used for storing data in key/value pairs. The hash table uses a simple method (linear probing) to store each unique element (integer) in a different position. The elements are stored using a method that makes searching easy, quick and offer low complexity. Our implementation uses a modulo hash function with a search complexity of O(1) when collisions are not present, up to a search complexity of O(n). Zero insertion is not implemented.

The variables:

->size is the number of elements currently stored in the hash table

->maxSize is the maximum number of elements that can be stored in the hash table

->table is the array of pointers to integers that implements the hash table

The methods:

->HashTable(): the constructor, which is responsible for reading the numbers from the file and calling the insert method, to build the hash table.

->insert(int number): inserts each element into the hash table and makes sure that the hash table's property is followed. To find the index to store the element we use (index = number % len(Array)) . The element gets placed on Array[index] if it is zero otherwise it gets placed on the next empty index of the array (a place is considered empty if Array[place]=0).

->getSize(): method to get current size of hash table.

->search(int number): searches for the given as a parameter element in the hash table. We look at an element in Array[index] (index = n % len(Array)). If it holds the value of the element we are looking for we return True. If it holds the value of 0 we return False else if it holds the value of another number we keep looking at the next index until we find the element and return True or find 0 and return False.

->rehash(): stores its data in a new hash table with bigger capacity (rehashes) when the load factor exceeds 0.75.

->getLoadFactor(): calculates the load factor of the hash table. If the load factor is more than 75% then we double the size of the array and insert again on the now double in size array.

->hashFunction(int number): function implementing a modulo hash depending on the size of the array. (index = number % len(Array))