DATA STRUCTURES

Lefkopoulos Miltiadis *Lefkopoulos Georgios*

Table of Contents

The library contains the implementation of basic data structures such as:

- Priority Queue
- Pile of Minimums
- Pile of Maxims
- AVL Binary Search Tree
- Weighted graph without directions
- Hash Table

The structures are implemented as classes, while they also use additional auxiliary data structures (linked lists, stacks) and auxiliary structures (nodes, edges). In each structure, the declaration (.h) is separated from its implementation (.cpp).

Priority Queue

Min Heap Max Heap

heap.h | heap.cpp

- **PriorityQueue Class**

Implements the Priority Queue. Initially, this acts as a priority queue for the few. It is the basic class of the Minimum Pile and Maximum Pile structures.

Class members:

const int MAX_SIZE: The number of items that a new queue can store (set to 100). **int *data:** The memory address of the tail root. **int N:** The number of items in the queue.

int SIZE: The number of items that the queue can store.

Class methods:

PriorityQueue(): Builds a new object of the class with the ability to initially store up to 100 items, freeing up memory space.

PriorityQueue(int): Builds a new object of the class with the ability to store a specific number of items, freeing up memory space.

PriorityQueue(int): Constructs a new object of the class by copying the elements of another object.

~PriorityQueue(): Destroys an object of the class by freeing up memory space.

bool buildHeap(int[], int): Creates an object of the class by pulling the data from a table. Even if the queue is not empty, it is treated as empty, so the original elements are ignored. Initially, if necessary, it increases the number of items that the queue can store with the **resize()** utility method and copies the data from the table to the queue. It then corrects the queue to create a priority queue. First, it checks whether the last leaf of the tree is a left child and if necessary, it swaps it with its parent, using the

swap**()** auxiliary method. Then it checks all previous children (now each parent has two children), finds the minimum of two and if it is even smaller than its parent, it replaces them. It continues in this way to the previous children until it reaches the root and its children. **int getSize():** Returns the number of items in the queue.

bool findElement(int&): Returns the minimum element of the queue if it is not empty.

bool insertElement(int): Inserts a new item into the queue. First, if necessary, increases the number of items that the queue can store with the **resize()** utility method , and then places the new item at the end of the queue. Then, if it is smaller than its parent, it transposes items, continuing until it reaches its correct position in the priority queue.

bool deleteElement(int&): Deletes the root of the tree, i.e. the minimum element of the queue if it is not empty. It then places the last element of the queue in its place. If the queue is not empty, arranges it to be a priority queue by putting the root in its correct position with the **heapify() helper method**. **void printPQ():** prints the queue items.

Auxiliary methods of the void swap(int&, int&) class: Swaps two queue items. **int compare(int, int):** Compares two elements of the queue and returns the difference between the first and the second. In this way, the priority queue acts as a priority queue of minimums. It is declared as virtual as the subclasses of **the PriorityQueue** contain their own **compare() counterparts**.

bool heapify(int): Checks if an item in the queue is in the correct position. If the item has no children, then it is a card so it is in the correct position (there is no way that it needs to change position with its child). If it has only one child (parent of the last card), checks if it needs to change position with it. If it has two children, checks if he needs to change position with the minimum of them. The method pushes elements down, so it's necessary when deleting the root of the tree or when building a priority queue and want to check if a parent who has been displaced with their minimum child needs to be pushed even lower.

bool resize(): Increases the number of items that the priority queue can store by doubling the available space each time. Initially, it stores twice as much memory as the original space, copies the queue items to the new space, and frees up the original space.

- **minHeap Class**

It implements a Pile of Minimums. It is a subclass of **the PriorityQueue** and works in exactly the same way. It exists in proportion to the **maxHeap class**.

Auxiliary methods of the int compare(int, int) class: Compares two elements of the queue and returns the difference between the first and the second. In this way, it makes the pile function as a pile of minimums.

- **maxHeap Class**

Implements a Maxima Stack. This is a subclass of **PriorityQueue**, but the different implementation of the **compare()** method allows the class to function as a maxima stack. With this differentiation it is possible to use the **PriorityQueue** class either as a stack of minimums (in the case of **minHeap**) or as a stack of maximums (in the case of **maxHeap**). It exists in proportion to the class **minHeap**.

Auxiliary methods of the int compare(int, int) class: Compares two elements of the queue and returns the difference between the second and the first. In this way, it makes the pile function as a pile of maximums.

AVL Binary Search Tree

avl.h | avl.cpp

- **AVL Class**

It implements the AVL Binary Search Tree. It acts as a Binary Search Tree, but always keeps it balanced. Thus, there is no node where one subtree is higher than the other, more than one level.

Auxiliary Structure of Class:

Node: The structure that stores the nodes of the tree. Each node contains a **data**, its height, as it is the node itself for the root of a subtree, a pointer for the left and a pointer for its right subtree.

Class members:

int *root: The memory address of the tree root.

int size: The number of elements in the tree.

Class methods:

AVL(): Constructs a new object of the class by initializing it.

~AVL(): Destroys an object of the class. **bool isEmpty():** Checks if the tree

is empty. **int getSize():** Returns the number of elements in the tree.

bool findMin(int&): Returns the smallest element of the tree. Uses a pointer on a node that goes to the left child of each node, if it exists, thus returning the left leaf of the tree, which is its smallest element.

bool searchElement(int): Searches for an item in the tree. Uses a pointer to a node. If the node contains the item, then the method returns how the item was found. If the item is smaller or larger than the node item, the pointer goes to the left or right subtree

of the node, respectively, repeating the search until the item is found or an item does not have the corresponding subtree.

bool insertElement(int): Inserts an element into the tree by calling the **insertAVL() helper method**.

bool deleteElement(int): Deletes an item from the tree by calling the **helper method deleteAVL().**

void printAVL(): Prints the tree with an inter-arranged path. Uses the **printSubTree()** utility method, calling it the root of the tree by argument, thus printing the entire tree.

Auxiliary methods of the void printSubTree(Node*) class: Prints a subtree with an interarranged path. Parameter the address of a node. Prints the left subtree of the node sequentially, then the node itself, and then its right subtree. To print each subtree, it calls the method with the root of the subtree as an argument.

int getData(Node*): Returns the value of the element of a node by accepting its address as a parameter.

bool insertAVL(Node*&, int): Inserts an element into a subtree. Parameter is the address of the root of the subtree. If the subtree is empty, inserts the new element as the root of the subtree using the helper method **insertroot().** If the element is smaller than the root element, it calls the method backwards by argumenting the address of the left child of the node; i.e. the direction of the root of its left undertree. Then, it is checked whether the height of the left subtree is two times greater than the height of the right

In this case, it is checked whether the new element was inserted as a left child at the root of the left subtree, in which case the method is called **rorateR(),** or as the right child of the left subtree, in which case the method is called the **rotateLR()** method, in order to balance the tree again. where it may be necessary to balance the tree by calling either the **rotateL()** method, if the new element was inserted as a right child at the root of the right subtree, or the **rotateRL()** method, if it was inserted as a left child at the root of the right subtree. If the element already exists, no action is required as duplicates are not allowed in the binary search tree. The height of each sub-tree where the new element was retroactively introduced is again calculated, as the maximum of the heights of the children of the node increased by one, until we return to the root of the tree.

bool deleteAVL(Node*&, int): Deletes an element from the tree. It takes the address of the tree's root as a parameter and with the help of a pointer to a tree node, it goes to the left or right subtree of each node until the element is found. In the path, it uses a helper table to record each node in the path and a second one to record whether it has continued to search the left or right subtree of that node. the route followed, can then check and correct any imbalances. If the item to be deleted is not found, the method terminates unsuccessfully. Otherwise, the following cases are checked: If the node has two children, the node will be replaced with its next largest, i.e. the leftmost child of its right-hand child, which the method finds by continuing the inter-arranged route. It then deletes the node containing the element by calling the **deleteBST()** helper method. It then checks whether the deletion caused imbalances in the tree. We go backwards the path from the parent of the deleted node to the root, which we have saved in the helper tables. If the difference in the heights of the subtrees of a node has exceeded one, Then in case the left subtree is higher than the right one, a right rotation is required to correct the imbalance. If the left subtree of the left child of the node is higher than its right subtree, the **rotateR()** method is called, while if the right subtree of the node is higher than the left **one, the rotateLR()** method is called. Correspondingly, if the right subtree of the node is higher and the right subtree of the right child is higher than the left, the **rotateL()** method is called, otherwise **rotateRL().** Thus, after the rotations, the new subtree is created, which takes the place of the left or right child of the parent of the node we are examining, depending on whether we reached this node by going left or right from its parent.

bool deleteBST(Node*&, int): Deletes an item from a simple Search Binary Tree. It starts at the root of a tree and searches for the node that contains the item to be deleted. For the search, a pointer to a node is used that traverses the tree until the item is found, as well as another helper pointer on a node that points to the parent of the previous node. Saves whether the last path taken was left or right. If the item is not found, the method terminates unsuccessfully. If found, the following cases are examined: If the node has no children and is the root of the tree, the node is deleted and the tree remains empty, while if it was not the root, it is deleted and depending on whether it was a left or right child, its parent is left without a left or right child. If the node has only one child and it is the root, then it is deleted and its child becomes the root, while if it is not a root, it is deleted and its child becomes the left or right child of the parent of the node, depending on whether he was a left or right child. If the node has two children, the node will be replaced by its next largest, i.e. the leftmost child of its right-hand child, which the method finds by continuing the interarranged path. It goes to the right child of the node and then to each left child if there is one, it finds the node that will replace the node that will be deleted and replaces their details. If the right child of the node to be deleted did not have a left child, its right child becomes the right child of the node that replaced the deleted one.

If the right child of the node to be deleted had a left child, the right subtree of the first in the inter-arranged route becomes the left sub-tree of the first's parent in the inter-arranged route, thus taking its place.

bool insertRoot(Node*&, int): Inserts a new node at the root of a subtree. The subtree must be empty. Therefore, if it is not, the method completes unsuccessfully, otherwise it creates the node at the root of the subtree, freeing up memory space and inserting the element passed into the method as a parameter.

bool deleteRoot(int&): Deletes the root node of a tree. If the root contains children, the method fails. Otherwise, the node is deleted, freeing up memory space. The tree is now empty while the original node element is returned.

bool deleteLeft(Node*, int&): Deletes the left child of a node. If it does not exist or has children, the method fails. Otherwise, the left child is deleted, freeing up memory space. The node no longer has a left child while the original node element is returned.

bool deleteRight(Node*, int&): Deletes the right child of a node. If it does not exist or has children, the method completes unsuccessfully. Otherwise, the right child is deleted, freeing up memory space. The node no longer has a right child while the original node element is returned.

int height(Node*): Returns the height of a subtree that is stored at its root node.

Node* rotateR(Node*): Performs a right rotation on a node. Uses an auxiliary pointer that points to the left child of the node. First, the right child of the left child becomes the left child of the parent. Then the parent becomes the right child of the left child. The heights of the parent's and his or her left child are then calculated. to the maximum of their children's heights, increased by one. Finally, the auxiliary pointer is returned so that the left child takes the place of the parent in the method that called the right rotation.

Node* rotateL(Node*): Performs a left rotation at a node. Uses an auxiliary pointer that points to the right child of the node. First, the left child of the right child becomes the right child of the parent. Then the parent becomes the left child of the right child. The heights of the parent's and their right child are then calculated. to the maximum of their children's heights, increased by one. Finally, the auxiliary pointer is returned so that the right child takes the place of the parent in the method that called the left rotation.

Node* rotateLR(Node*): Performs a left-right rotation on a node. Initially, performs a left rotation on the left child of the node by calling the **rotateL()** method. Then, performs a right rotation on the node by calling the **rotateR()** method, returning the pointer itself to return the right rotation so that the left child takes the place of the parent in the method that called the right rotation.

Node* rotateRL(Node*): Performs a right-left rotation on a node. It first performs a right rotation on the right child of the node by calling the **rotateR()** method. It then performs a left rotation on the node by calling the **rotateL()** method, returning the pointer itself to return the left rotation so that the right child takes the position of the parent in the method that called the left rotation.

void clearBST(Node*): Deletes a subtree. Uses a post-ordered traversal to delete the children of a node before deleting it. Works retroactively. It is only used by the class destructor to release the memory that was dynamically reserved for the tree, so it

does not need to correct the pointers to null or the size of the tree, since all of them will be deleted.

Weighted graph without directions

graph.h | graph.cpp

- **Graph Class**

He implements the Graph. This is a non-directional graph with weights on the edges. The graph is stored with neighborhood lists so that there is quick access to the neighbors of a peak.

Auxiliary structures of the class:

Edge: A structure that stores an edge between two vertices of the graph. Each edge contains the id of one vertex (**int node**), and the ID of the other vertex to which it is linked (**int parent**), Do not need the weight of the edge.

Node: A structure that stores a graph node. This is a vertex that is connected to an initial vertex. Each node contains the int **id** and the int **weight** with one end the original vertex and the other end the vertex of the node. The structure is necessary for the implementation of the contiguous list where the graph is stored.

Auxiliary Classes:

class Stack: Implements a Stack. The stack can store **Edge** structures, i.e. the edges that we follow and should keep in a stack when we do not retroactively use the methods to make use of the offered memory stack. It is required for the **findConnectedComponents() method**.

class List: Implements a Linked List. Used to implement the neighboring lists in which the graph will be stored.

Class members:

List *array: The table with the memory addresses of the neighboring lists. **int N**:

The number of vertices of the graph. **int M**: The number of edges of the graph.

Class methods:

Graph(): Constructs a new object of the class by initializing it.

~Graph(): Destroys an object of the class.

bool buildGraph(int): Constructs a graph by committing space in memory for the table where as many contiguous lists as the vertices of the graph will be stored. Lists contain nodes, where within each node the vertex is stored the vertex that connects to the vertex that corresponds to each cell of the table and the weight of the edge that connects the two vertices. lists are linked and each node points to the next.

int getN(): Returns the number of vertices of the graph. **int getM():** Returns

the number of edges of the graph.

void printGraph(): Prints the graph. Specifically, it prints the neighboring lists, that is, any linked lists, with the **printList() method of the** List **class**.

bool addEdge(int, int, int): Adds a new edge to the graph using the **insertNode()** utility method. First, it checks whether the edge actually connects existing vertices of the graph and then inserts a new edge from the first to the second vertex. If the insertion is successful, it inserts its corresponding edge, from the second vertex to the first vertex as the graph is undirected. If the first insertion fails, The method fails as acne already exists.

bool deleteEdge(int, int): Deletes one edge of the graph using the **deleteNode()** utility method. First, checks whether the edge actually connects existing vertices of the graph and then deletes the edge from the first to the second edge. If the deletion is successful, it also deletes the edge from the second vertex to the first vertex as the graph is undirected. If the first deletion fails, The method fails as acne does not exist.

void destroyGraph(): Deletes all edges and vertices of the graph with the **helper method destroyList().** Essentially deletes the neighboring lists, frees up the memory space where their addresses are located. The graph is now empty with zero vertices and edges.

bool computeShortestPath(int, int, int*&, int&, int&): Calculates the shortest path between two vertices using Dijkstra's algorithm. It first calculates the sum of the edges of the graph by adding for each vertex the sum of its edges using the **helper method findNeighbors().** It then stores in a table the initial distances of each edge from the initial vertex to the sum of the graph edges as no vertex It may be more distant from another than that, while the initial distance of the original vertex is equal to zero. In a second table, the previous of each vertex is stored in the shortest path with an initial value minus one, as the path has not yet been specified, while the previous of the original vertex is defined as the vertex itself. A third table is stored if we have finalized each vertex, which does not apply to any vertex. A meter is initialized with the number of vertices that have been finalized with zero and as long as there are non-finalized vertices, the one with the minimum distance from the original is located (the first time is the initial itself). This peak is finalized, all its neighboring vertices are located using the **findNeighbors()** auxiliary method, and for each one, its new distance from the original vertex is calculated as the sum of its distance from the test vertex plus the distance of the test vertex from the original one. You also set the test vertex as the previous one for each of its neighbors in the corresponding table. Repeat the process until the final vertex is found, when it is interrupted the process. Then the number of vertices of the shortest path is calculated from the table with the previous ones of each peak and starting from the final one and going to its previous and previous one, until it reaches the original one. Once the number of vertices of the shortest path is calculated, space is reserved in memory for the vertices of the path and by the same process as before, all vertices of the shortest path from the final to the beginning are stored in the table from end to beginning. The method returns the table with the vertices of the shortest path, the number of those peaks, and its length, which is in the first table as the distance of the final vertex from the original one, after freeing up memory space for the tables used.

bool computeSpanningTree(Graph&, int&): Calculates the Minimum Connecting Tree (NAP) of the graph using the Prim algorithm. The NAP will be stored in a new graph which is created using **build()** and a number of vertices equal to the vertices of the original graph. Initially, the maximum edge of the graph is calculated using the **helper method findNeighbors()**. The initial distances of each edge from the NAP are then stored in a table as the maximum edge plus one, as no vertex can be more distant from the NAP than that, while the initial distance of the initial vertex is zero. A second table

is saved if each vertex has been finalized, which is not the case for any of them initially. It starts the NAP with an initial vertex, arbitrarily, the one with ID 0, and with the helper method **findNeighbors()** the distance of each of its neighboring vertices from the CIS is calculated as its distance from the original vertex. Also, in a third table, the edges that connect the neighboring vertices to the CIS (in this case with the original vertex) are stored, which is being finalized. Also, a meter with the number of vertices that have been finalized with one (the initial vertex) and a meter with the length of the NAP with zero (the distance of the initial vertex from the NAP) is initialized. As long as there are non-finalized vertices, the one with the minimum distance from the NAP is located, it is finalized, the edge that connects it to the NAP is inserted into the NAP graph, using the **addEdge()** method, and its distance from the NAP is added to the length of the NAP. Then, the third table is reconstructed and the edges that connect the vertex we are examining with its neighboring ones are stored, as it belongs to the NAP while the whole process is repeated. The method returns the graph of the NAP and its length, after freeing up memory space for the tables used.

bool findConnectedComponents(int&): Calculates the number of coherent components of the graph. Initializes a counter with the number of components to zero. Arbitrarily selects a vertex, increases the counter and starts a Deep Traverse of the graph. When the traverse is finished, if there are vertices that have not been checked, the process is repeated. The method returns the number of EDs, that is, the number of coherent components. The method uses the **DSF() utility method**.

Auxiliary methods of the bool class findNeighbors(int, int&, struct Node*&): Returns a table that contains all the neighboring vertices of a given vertex. Uses the **arrayList() method of the** List **class** , which frees up memory space for the table with a size equal to the number of nodes in the list, and then traverses the list to store the nodes in the table.

void DFS(int, bool[], int&): Crosses the graph at the bottom. The method works backwards. It starts a traverse from an arbitrary vertex, and as long as it has neighbors that we haven't visited, a new DFS starts from it. If there is no vertex we haven't visited, it searches for the next vertices of the stack. It uses the stack memory so that we don't have to implement a separate stack.

Hash Table

hash.h | hash.cpp

- **Hash Class**

Implements the Hash Table. The table stores the data in a Simply Linked List, so in case of a conflict, the data is in the correct cell at some node in the list. The hash function is **h(x) = (2x + 1) mod 193**.

Auxiliary Classes:

class HashList: Implements a Simply Linked List. The list can store integer elements.

Class members:

const int MAX_SIZE: The size of the table, set to 193 positions. It is a prime number as the integer division remainder of the hash function reduces the chances of a collision when used with a prime number.

HashList *array: The memory address of the hash table that will contain the lists of data. **int size:** The number of items stored in the table.

Class methods:

Hash(): Constructs an object of the Hash class with an initial size of zero.

~Hash(): Destroys an object of the Hash class by freeing up memory space for the table.

bool insertHash(int): Inserts an item into the table. To insert the item, it is necessary to create the hash table using the **buildHash()** method. Calculates with the **getHash() utility method** the location of the table where the item should be inserted. Looks for whether the item already exists in the list of cells where it should be inserted with the **searchList()** method. If it already exists, the method terminates unsuccessfully as duplicates are not allowed in a Hash Table. Otherwise, it inserts the item into the cell list using the **insert()** method.

bool buildHash(): Constructs the table by freeing up memory space for a 193-position **HashList table**.

void destroyHash(): Deletes all elements of the table. Deletes all lists using the **destroyList()** method, frees up the table space in memory, and sets its size to zero.

void printHash(): Prints the table. Prints each of its lists using the **printList() method**.

int getSize(): Returns the size of the table, i.e. the number of its elements.

bool searchHash(int): Searches for an item in the table. If the table is constructed, it calculates the **location of the table where the item should be located using** the getHash() utility method and **searchList() searches** for it in the list. It returns as a value whether or not the item was found.

Auxiliary methods of the int getHash(int) class: Calculates the hash value of an item based on the hash function.