Αριστοτέλειο Πανεπιστήμιο Θεσσαλονίκης – Τμήμα Πληροφορικής

Website links

Δομές Δεδομένων – Εργασία 2η

Αναγνώστου Αντώνης	2268	anagnoad@csd.auth.gr
Λασκαρίδης Στέφανος	2315	laskstef@csd.auth.gr

Περιεχόμενα

Περιγραφή προβλήματος 3

Δομές Δεδομένων που χρησιμοποιήθηκαν 4

Hash Table 4

Κύριο Hash Table 4

Δευτερεύοντα Hash tables 5

AVL Tree 5

Min Heap 5

Περιγραφή κλάσεων 7

TreeNode class 7

Λειτουργικότητα κλάσης 7

AVL class 8

Λειτουργικότητα κλάσης 8

Node class 10

Λειτουργικότητα κλάσης 10

Database class 11

Λειτουργικότητα κλάσης 12

SimpleHashTable class 14

Λειτουργικότητα κλάσης 14

ComplexHashTable class 15

Λειτουργικότητα κλάσης 15

MinHeap class 17

Λειτουργικότητα κλάσης 18

Κατασκευή του MinHeap 18

Εισαγωγή στοιχείων 18

Διαγραφή ελαχίστου 18

Επεξεργασία στοιχείων 18

IO class 19

Λειτουργικότητα κλάσης 19

Input/Output 20

Μορφή εντολών 20

Μορφή input 22

Μορφή output 22

Μελλοντικές Βελτιώσεις 24

Περιγραφή προβλήματος

Στην παρούσα εργασία καλούμαστε να υλοποιήσουμε ένα μη κατευθυνόμενο γράφο ο οποίος απεικονίζει τις συνδέσεις μεταξύ διαφόρων κόμβων σε ένα δίκτυο.

Μία πιθανή εφαρμογή του συγκεκριμένου προβλήματος, είναι η απεικόνιση ενός γραφήματος μεταξύ ιστοσελίδων με συνδέσεις καθορισμένης σημαντικότητας.

Πιο συγκεκριμένα, δοθέντος ενός αρχείου εισόδου που περιέχει ακμές της μορφής

<κομβος a> <κόμβος b> <βάρος w>

Η εφαρμογή καλείται να κατασκευάσει το αντίστοιχο γράφημα που περιγράφει το αρχείο, χρησιμοποιώντας δομές δεδομένων του πίνακα κατακερματισμού του δένδρου AVL.

Το πρόγραμμα αυτό καλείται να υποστηρίζει τις ακόλουθες λειτουργίες:

- Εισαγωγές νέων κόμβων
- Διαγραφές ήδη υπαρχόντων κόμβων
- Υπολογισμός ελαχίστου δένδρου (ή ελαχίστου δάσους σε μη συνεκτικό γράφο)
- Υπολογισμός κοινών γειτόνων μεταξύ δύο κόμβων
- Υπολογισμός ελαχίστων διαδρομών από μία κορυφή σε όλες τις υπόλοιπες

Οι λειτουργίες που καλείται να εκτελέσει το πρόγραμμα περιέχονται σε ένα αρχείο κειμένου, ενώ αντίστοιχα και η έξοδος των αποτελεσμάτων θα πρέπει να γίνεται σε αρχείο κειμένου επίσης.

Στόχος δεν είναι μόνο η παραγωγή των ζητούμενων αποτελεσμάτων και η ορθότητα τους, αλλά η επίτευξη του βέλτιστου κόστους υλοποίησης – εκτέλεσης.

Δομές Δεδομένων που χρησιμοποιήθηκαν

Για την υλοποίηση του προβλήματος, αποφασίσαμε να χρησιμοποιήσουμε τις εξής δομές δεδομένων.

Κύριες δομές αποθήκευσης δεδομένων:

- Πίνακας κατακερματισμού – Hash table
- Δένδρο AVL – AVL Tree

Δευτερεύουσες (βοηθητικές) δομές δεδομένων:

- Πίνακας κατακερματισμού – Hash table
- Σωρός ελαχίστων – Min Heap

Hash Table

Κύριο Hash Table

Για την κύρια οργάνωση των κόμβων του γραφήματος, χρησιμοποιείται ένας πίνακας κατακερματισμού ο οποίος αποθηκεύει τους κόμβους του γραφήματος.

Υλοποιείται με την κλάση Database, η οποία στο εσωτερικό της χρησιμοποιεί την κλάση Node, για την αποθήκευση των κόμβων του γραφήματος μαζί με τους συνδεόμενους γείτονες, με τα αντίστοιχα βάρη.

Στον κύριο πίνακα κατακερματισμού, επιλέχθηκε κατακερματισμός ανοιχτής διεύθυνσης (open address hashing).

Όσον αφορά τη συνάρτηση κατακερματισμού, χρησιμοποιήθηκε διπλός κατακερματισμός [1](double hashing) με συναρτήσεις

- h1(x) = x mod capacity
- h2(x) = 1+ x mod (capacity -1)

Οι υποστηριζόμενες λειτουργίες είναι οι παρακάτω:

Λειτουργία:	**Υπολογιστικό κόστος:**
Αναζήτηση	• *Μέση περίπτωση: O(1)*[2] • *Χειρότερη περίπτωση: O(n), όπου n το πλήθος των αποθηκευμένων στοιχείων.*
Εισαγωγή νέου κόμβου	• *Μέση περίπτωση: O(1)* • *Χειρότερη περίπτωση: O(n), όπου n το πλήθος των αποθηκευμένων στοιχείων.*
Διαγραφή κόμβου	*<u>Σημείωση</u>: Η διαγραφή κόμβου υποστηρίζεται εικονικά για τις ανάγκες της εργασίας. Διαγραφή κόμβου συνεπάγεται πως παίρνει δεδομένη «κενή» τιμή η εγγραφή στον πίνακα κατακερματισμού.*

[1] Double hashing: τεχνική κατακερματισμού ανοιχτής διεύθυνσης που χρησιμοποιεί το εξής πρότυπο συνάρτησης κατακερματισμού:
$H(x) = h1(x) + i * h2(x)$, *όπου i ο αριθμός των συγκρούσεων κατά την εισαγωγή – αναζήτηση.*

[2] Η μέση περίπτωση παράγει σταθερός κόστος, όταν ο παράγοντας φόρτισης του πίνακα κατακερματισμού είναι μικρός (<0.5).

Δευτερεύοντα Hash tables

Επιπλέον, ως βοηθητικές δομές σε συγκεκριμένες περιπτώσεις, χρησιμοποιούμε δύο ακόμη «τύπους» πινάκων κατακερματισμού οι οποίοι είναι απλούστεροι, τόσο ως προς τον τύπο αποθήκευσης, όσο και ως προς τη συνάρτηση κατακερματισμού.

Πιο συγκεκριμένα, χρησιμοποιείται ένας πίνακας κατακερματισμού για αποθήκευση τιμών ακεραίων και χρησιμοποιείται για την εύρεση της τομής μεταξύ δύο δένδρων AVL. (SimpleHashTable)

Ο δεύτερος πίνακας κατακερματισμού (ComplexHashTable) αποτελείται από εγγραφές struct complexHashEntry με πεδία τριών ακεραίων και χρησιμοποιείται στα εξής δύο σημεία:

- Σε συνδυασμό με σωρό ελαχίστου ώστε να δώσει τη δυνατότητα αναζήτησης στην προαναφερθείσα δομή σε σταθερό χρόνο στη μέση περίπτωση.
- Στους αλγορίθμους εύρεσης ελάχιστων διαδρομών και ελαχίστου δένδρου (ή ελάχιστου δάσους) ως βοηθητικές δομές οργάνωσης ενδιάμεσων και τελικών αποτελεσμάτων.

Και στις δύο περιπτώσεις, χρησιμοποιείται κατακερματισμός ανοιχτής διεύθυνσης και γραμμική επίλυση συγκρούσεων στο hashing. Η συνάρτηση κατακερματισού είναι η εξής:

$$h(x) = (x \ mod \ 993319)^{13} \ mod \ capacityOfHashTable$$[3]

Οι λειτουργίες που υποστηρίζονται είναι παρόμοιες με αυτές του κύριου πίνακα κατακερματισμού.

AVL Tree

Κάθε γραμμή του array αυτού, περιέχει το id του κόμβου από το οποίο εξέρχονται links, το πλήθος των links το οποίο περιέχει καθώς και ένα δένδρο AVL το οποίο αποθηκεύει τους κόμβους στους οποίους οδηγούμαστε από την παραπάνω σελίδα.

Υλοποιείται με την κλάση AVL, η οποία στο εσωτερικό της χρησιμοποιεί τις κλάσεις Node, AVL, TreeNode.

Ο λόγος για τον οποίο επιλέχθηκε η συγκεκριμένη δομή για την αποθήκευση των συνδεδεμένων κόμβων, είναι για την επίτευξη των ελαχίστων χρόνων στις εξής λειτουργίες:

Λειτουργία:	Υπολογιστικό κόστος:
Αναζήτηση	$\log_2 n$, *όπου* n *το μέγεθος του δένδρου*
Εισαγωγή	$\log_2 n$, *όπου* n *το μέγεθος του δένδρου*
Διαγραφή	$\log_2 n$, *όπου* n *το μέγεθος του δένδρου*
Εκτύπωση inorder	n, *όπου* n *το μέγεθος του δένδρου*

Min Heap

Η συγκεκριμένη δομή χρησιμοποιήθηκε εκτενώς για την υλοποίηση των αλγορίθμων εύρεσης ελαχίστων διαδρομών και ελάχιστου δένδρου (ή δάσους) επί του γραφήματος για την άμεση εξαγωγή ελάχιστης τιμής μεταξύ άλλων.

Στην υλοποίηση μας συνδυάζεται με τη δομή δεδομένων ενός hashtable ώστε να υποστηριχθεί η δυνατότητα αναζήτησης στο σωρό ελαχίστων.

Συνοπτικά, οι υποστηριζόμενες λειτουργίες είναι οι εξής:

[3] Η επιλογή των τιμών 993319 και 13 έγινε στα πρότυπα εύρεση ενός μεγάλου και ενός μικρού πρώτου αριθμού αντίστοιχα.

Λειτουργία:	Υπολογιστικό κόστος:
Δημιουργία	O(n), όπου n το πλήθος των κόμβων στο οποίο αρχικοποιείται ο σωρός.
Εισαγωγή	O(logn), όπου n το πλήθος των στοιχείων που περιέχει ο σωρός.
Διαγραφή min	O (logn), όπου n το πλήθος των στοιχείων που περιέχει ο σωρός.
Αναζήτηση τιμής	• O(1) στη μέση περίπτωση • O(n) στη χειρότερη περίπτωση (πρόρχεται από το hash table)
Επεξεργασία τιμής	O(logn), όπου n το πλήθος των στοιχείων που περιέχει ο σωρός.

Περιγραφή κλάσεων

TreeNode class

TreeNode	
public:	
TreeNode()	*Default constructor*
TreeNode(int value)	*Constructor που δέχεται αρχική τιμή για τον κόμβο του δένδρου*
TreeNode(int value, int weight)	*Constructor που δέχεται αρχική τιμή για τον κόμβο του δένδρου, αλλά και για το βάρος της ακμής με την οποία συνδέεται.*
virtual ~TreeNode()	*Destructor*
Int getHeight()	*Accessor που επιστρέφει την τιμή height του κόμβου*
Int getValue()	*Accessor που επιστρέφει την τιμή (id) του κόμβου*
Int getWeight()	*Accessor που επιστρέφει την τιμή weight για τον κόμβο αυτόν.*
TreeNode* getLeft() **TreeNode* getRight()**	*Accessors που επιστέφουν pointer σε leftChild και rightChild αντίστοιχα*
void setLeft(TreeNode * left) **void setRight(TreeNode * right)**	*Mutators που θέτουν τους pointers σε κάποιο TreeNode.*
void setValue(int idOfConnectedNode)	*Mutator που θέτει τιμή στο πεδίο value*
void fixHeight()	*Καλείται όταν χρειάζεται ανανέωση του ύψους του συγκεκριμένου κόμβου. (σε κάποιες εισαγωγές/διαγραφές)*
private:	
TreeNode* leftChild	*Pointer τύπου treeNode που δείχνει στο αριστερό παιδί του κόμβου. Εάν δεν υπαρχει παιδί, έχει τιμή nullptr.*
TreeNode* rightChild	*Pointer τύπου treeNode που δείχνει στο δεξί παιδί του κόμβου. Εάν δεν υπαρχει παιδί, έχει τιμή nullptr.*
int value	*Το id του ίδιου του κόμβου. Default value = -1.*
int height	*Πεδίο τύπου int στο οποίο αποθηκεύεται το ύψος του υποδένδρου το οποίο ορίζει ο συγκεκριμένος κόμβος. Default value = 1.*
Int weight	*Το βάρος της ακμής προς τον κόμβο αυτό.*

Λειτουργικότητα κλάσης

Η κλάση TreeNode δημιουργήθηκε με σκοπό να αποθηκεύονται οι επιμέρους κόμβοι του δένδρου AVL ως αντικείμενά της.

Συνδέεται με την κλάση AVL με σχέση (has-a), δηλαδή ένα αντικείμενο AVL έχει αντικείμενα τύπου TreeNode.

Θεωρούμε πως η υλοποίηση αυτή συνδυάζει περισσότερα πλεονεκτήματα, έναντι ενός struct, καθώς υπάρχουν μέθοδοι οι οποίες ορίζονται και λειτουργούν πάνω στα δεδομένα των κόμβων. Επιπλέον, τηρείται, με αυτό τον τρόπο και η αρχή αντικειμενοστραφούς προγραμματισμού της ενθυλάκωσης (encapsulation).

Δεδομένου πως στην παρούσα εργασία μελετάμε γραφήματα των οποίων οι ακμές περιέχουν βάρη, η κλάση treeNode περιέχει πλέον και ένα παραπάνω πεδίο για την αποθήκευση του βάρους αυτού.

AVL class

AVL	
Public:	
AVL()	Default constructor
virtual ~AVL	Destructor
friend ostream& operator<< (ostream& myStream, AVL &obj)	Operator overload για την αποθήκευση του δένδρου στο αρχείο (καλεί αναδρομικά την printTree(treeNode* root, ostream& myStream)).
Int getNumberOfLeaves()	Accessor για την επιστροφή του numberOfLeaves.
void insertTreeNode(int idOfConnectedNode)	Εισαγωγή νέου κόμβου στο δένδρο. Καλεί την insertTreeNode(int idOfConnectedNode, treeNode * root) με root το head.
void deleteTreeNode(int idOfConnectedNode)	Διαγραφή κόμβου από το δένδρο. Καλεί την deleteTreeNode(int idOfConnectedNode, treeNode * root) με root το head.
treeNode getInOrder(treeNode* root)**	Επιστρέφει δυναμικό πίνακα από αναφορές στα στοιχεία του δένδρου σε αύξουσα σειρά (inorder). Καλεί την getInOrderRecursive στο σώμα της.
void printTree(treeNode* root, ostream& myStream)	Καλείται αναδρομικά από το operator<< για την τύπωση του δένδρου (inorder)
resultOfIntersection intersectWithAVL(AVL* avlForIntersection)	Βρίσκει την τομή του τρέχοντος AVL με το AVL που δίνεται ως όρισμα. Στο εσωτερικό της χρησιμοποιεί πίνακα κατακερματισμού για βελτίωση του χρόνου εκτέλεσης του αλγορίθμου. Επιστρέφει έναν πίνακα με τα αναγνωριστικά των κοινών γειτόνων και το μέγεθος του πίνακα αυτού σε ένα struct.
Private:	
TreeNode * head	Pointer σε object τύπου TreeNode το οποίο δείχνει στη ρίζα του δένδρου. Εάν το δένδρο είναι κενό, η τιμή του είναι nullptr.
int numberOfLeaves	Μεταβλητή που αποθηκεύει το πλήθος των κόμβων που περιέχει το δένδρο
TreeNode * insertTreeNode(int idOfConnectedNode, treeNode * root)	Συνάρτηση που καλείται αναδρομικά για την εισαγωγή νέου κόμβου στο δένδρο.
TreeNode * deleteTreeNode(int idOfConnectedNode, treeNode * root)	Συνάρτηση που καλείται αναδρομικά για τη διαγραφή ενός κόμβου από το δένδρο
Static int calculateBf(treeNode * node)	Υπολογίζει και επιστρέφει το bf ενός κόμβου του δένδρου AVL.
Static TreeNode* rotateLL(TreeNode * node)	LL-rotation στο δένδρο με ρίζα node.
Static TreeNode* rotateRR(TreeNode * node)	RR-rotation στο δένδρο με ρίζα node.
Static TreeNode* rotateLR(TreeNode * node)	LR-rotation στο δένδρο με ρίζα node.
Static TreeNode* rotateRL(TreeNode * node)	RL-rotation στο δένδρο με ρίζα node.
Static TreeNode* fixTree(TreeNode* node)	Καλείται από την insertTreeNode για να φτιάξει τα ύψη των υποδένδρων, εκεί που χρειάζεται μετά την εισαγωγή νέου κόμβου.
void reallocateArray(int ** theArray, int* currentSize)	Διπλασιάζει το μέγεθος που έχει εκχωρηθεί σε πίνακα, αντιγράφει τα δεδομένα, διαγράφει το προηγούμενο χώρο και επιστρέφει το νέο πίνακα.
void getInOrderRecursive(treeNode* root, treeNode inOrderArray, int * counter)**	Καλείται αναδρομικά ώστε να λάβουμε τα στοιχεία του δένδρου με σειρά inorder. Καλείται από την getInOrder.

Λειτουργικότητα κλάσης

Η συγκεκριμένη κλάση δημιουργήθηκε για να αποθηκεύει του κόμβους που συνδέονται με μία ιστοσελίδα. Υποστηρίζει εισαγωγή και διαγραφή κόμβων, όπως και print inOrder.

Η κλάση αυτή συνδέεται με σχέση (has-a) με την κλάση Node, δηλαδή ένα αντικείμενο τύπου Node έχει ένα αντικείμενο τύπου AVL ως πεδίο του.

Θεωρούμε πως η αποθήκευση των συνδεδεμένων κόμβων με AVL είναι η βέλτιστη, όσον αφορά το context της συγκεκριμένης εργασίας, λόγω της ελαχιστοποίησης του κόστους των υποστηριζόμενων λειτουργιών.

Όσον αφορά το υπολογιστικό κόστος της τομής των 2 AVL, το κόστος είναι γραμμικό και ανάγεται σε O(n+m), αν n και m τα μεγέθη των 2 AVL αντίστοιχα, αφού διατρέχουμε όλα τα στοιχεία από 1 φορά.

Συγκεκριμένα, προκειμένου να υπολογίσουμε τους κοινούς κόμβους 2 AVL ακολουθούμε τα εξής βήματα:

1. Βρίσκουμε το AVL με τα περισσότερα στοιχεία.
2. Δεσμεύουμε χώρο για ένα SimpleHashTable (2* sizeOfMax) και εισάγουμε διαδοχικά τα στοιχεία του μεγαλύτερου AVL.
3. Διαδοχικά για τα στοιχεία του δεύτερου AVL, ελέγχουμε αν το στοιχείο υπάρχει ήδη στο HashTable. Αν υπάρχει, το εγγράφουμε σε έναν πίνακα, που θα επιστρέψουμε στο τέλος και θα περιέχει τα IDs των κοινών κόμβων.

Node class

Node	
public:	
Node();	*Default constructor. Δεσμεύεται χώρος για το AVL, και αρχικοποιείται το numberOfConnectedNodes στο 0. Ορίζεται σαν κατάσταση λάθους το id = -1, καθώς δεν έχει δοθεί κάτι άλλο από τον χρήστη*
Node(int id);	*Δεσμεύεται χώρος για το AVL, και αρχικοποιείται το numberOfConnectedNodes στο 0. Το id του node γίνεται ίσο με το όρισμα του constructor.*
Node(int id, AVL* avlTree);	*Στο νέο node, περνιούνται με ορίσματα στον constructor ένας δείκτης σε υπάρχον AVL, καθώς και το id. Το numberOfConnectedNodes γίνεται ίσο με τα φύλλα του AVL που δόθηκε.*
virtual ~Node();	*Αποδεσμεύει τον χώρο του AVL.*
int getID();	*Επιστρέφει το id του node.*
int getNumberOfConnectedNodes();	*Επιστρέφει τον αριθμό των συνδεδεμένων κόμβων.*
bool addNewNode(int idOfNewNode);	*Προσθέτει μία νέα σύνδεση, από το Node που βρισκόμαστε, στο node με το id που δίνεται σαν όρισμα.*
bool deleteNode(int idOfExisitingNode);	*Σβήνει μία υπάρχουσα σύνδεση με το node που έχει το id του ορίσματος (αν υπάρχει).*
friend ostream &operator<<(ostream &mystream, Node& obj);	*Operator overload.*
private:	
int id;	*Το id του κόμβου που βρισκόμαστε.*
int numberOfConnectedNodes;	*Ο αριθμός των συνδέσεων του κόμβου αυτού με άλλους.*
AVL* avlTree;	*Το AVL tree στο οποίο αποθηκεύονται οι συνδέσεις με άλλα Nodes.*

Λειτουργικότητα κλάσης

Πρόκειται για την κλάση των στοιχείων που βρίσκονται στο ταξινομημένο array της Database. Κάθε τέτοιο στοιχείο έχει ένα δένδρο AVL, για τις συνδέσεις με άλλα στοιχεία, ένα μοναδικό πρωτεύον κλειδί (id) καθώς και τον αριθμό των διασυνδεδεμένων κόμβων. Οι μέθοδοι addNewNode και deleteNode καλούνται από την Database, και αφαιρούν ή προσθέτουν ένα σύνδεσμο στο δένδρο AVL.

Υπολογιστικό κόστος

Καθώς η κλάση αυτή έχει μόνο δύο μεθόδους (εξαιρουμένων των get accessors) όποιο υπολογιστικό κόστος θα βρίσκεται σε αυτές τις δύο μεθόδους. Έτσι, για την εισαγωγή ενός συνδέσμου ή την διαγραφή αντίστοιχα, προκύπτει το αντίστοιχο κόστος για την εισαγωγή/διαγραφή στο avlTree, που είναι O(log(numberofConnectedNodes)).

Database class

Database	
public:	
Database();	*Default constructor. Δεσμεύει αρχικά 100.000 θέσεις στον πίνακα theDatabase, αρχικοποιεί το size στο 0 και το capacity στις 100.000 θέσεις. Η επιλογή του μεγέθους οφείλεται στο μέγιστο πλήθος κόμβων που πρόκειται να εισαχθούν στη δομή, ώστε να επιτευχθεί μικρός παράγοντας φόρτισης.*
virtual ~Database();	*Αποδεσμεύεται όλος ο χώρος της theDatabase.*
bool insertNewLink(int leftId, int rightId);	*Εισάγει νέο link μεταξύ της σελίδας με id= leftId και της σελίδας με id= rightId.*
bool deleteExistingLink(int leftId, int rightId);	*Σβήνει ένα υπάρχον link μεταξύ της σελίδας του leftId και της σελίδας του rightId.*
Int hashFunction(int key)	*Αποτελεί την γενική συνάρτηση κατακερματισμού του πίνακα κατακερματισμού. Στο εσωτερικό της καλεί την ομώνυμη private συνάρτηση.*
resultOfMST calculateMST();	*Συνάρτηση που βρίσκει το ελάχιστο δένδρο (ή το ελάχιστο δάσος, σε περίπτωση μη συνεκτικού γράφου) κάνοντας χρήση του αλγορίθμου του Prim. Επιστρέφει struct με το συνολικό κόστος και το χρόνο εκτέλεσης του αλγορίθμου.*
int commonNeighbours(int idOfNode1, int idOfNode2)	*Συνάρτηση που επιστρέφει το πλήθος των κοινών γειτόνων μεταξύ δύο κόμβων, ως τομή των δένδρων στα οποία αποθηκεύονται οι γείτονες τους καθενός.*
ComplexHashTable* shortestPath_Dijkstra(int idOfStartingNode)	*Συνάρτηση που βρίσκει τις ελάχιστες διαδρομές από τον κόμβο με αναγνωριστικό idOfStartingNode, με χρήση του αλγορίθμου του Dijkstra. Επιστρέφει ένα πίνακα κατακερματισμού που περιέχει όλους τους κόμβους του γραφήματος, τον προηγούμενο κόμβο και την απόσταση του καθενός.*
friend ostream &operator<<(ostream &mystream, Database& obj); **friend istream &operator>>(ostream &mystream, Database& obj);**	*Operators overload (insertion, extraction)*
private:	
Node theDatabase;**	*Στην ουσία πρόκειται για ένα sorted array.*
int size; **int capacity;**	*Το size αποτυπώνει το "load" της database, δηλαδή το πόσα στοιχεία έχουν εισαχθεί μία δεδομένη στιγμή, ενώ το capacity αποτελεί την συνολική χωρητικότητα που υποστηρίζει η δομή δεδομένων.*
int hashFunction(int key, int iteration)	*Συναρτηση κατακερματισμού του πίνακα. Κάνει χρήση των auxHashFunction1 και auxHashFunction2 για την επίτευξη του double hashing.*
int auxHashFunction1(int key)	*Πρώτη βοηθητική συνάρτηση κατακερματισμού που υπολογίζει τη θέση που πρέπει να μπει το στοιχείο όταν δεν υπάρχουν συκρούσεις.*
int auxHashFunction2(int key)	*Δεύτερη βοηθητική συναρτηση κατακερματισμού που υπολογίζει τη μετατόπιση σε κάθε σύκρουση από την προηγούμενη θέση που υπολογίστηκε και υπήρχε σύγκρουση.*
bool insertNode(Node nodeToBeInserted);	*Μέθοδος που καλείται τοπικά από την insertNewLink κατά την είσοδο ενός στοιχείου που δεν υπάρχει στο database.*
Node searchNodeByID(int idToSearch);**	*Αναζήτηση κόμβου στην database με βάση το αναγνωριστικό του. Πραγματοποιείται χρησιμοποιώντας τη συνάρτηση κατακερματισμού του πίνακα.*

Λειτουργικότητα κλάσης

Πρόκειται στην ουσία για την κύρια κλάση της εφαρμογής. Αυτή διαχειρίζεται όλες τις εισαγωγές και τις διαγραφές των συνδέσμων των σελίδων και τον κόμβων. Με διαδοχικά abstractions, καλεί συναρτήσεις της κλάσης Node, και της AVL.

Διαχειρίζεται αποτελεσματικά και τις καταστάσεις λάθους, όπως για παράδειγμα εισαγωγή ενός συνδέσμου πολλές φορές, ή διαγραφή συνδέσμου που δεν υπάρχει.

Επιπλέον, υποστηρίζει λειτουργίες εύρεσης ελαχίστης διαδρομής, ελαχίστου δένδρου ή ελάχιστου δάσους και κοινών γειτόνων για το γράφημα το οποίο ουσιαστικά περιέχει.

Πιο συγκεκριμένα:

Υπολογισμός Ελάχιστου Δένδρου (Minimum Spanning Tree – MST) – Prim's algorithm

Για την εύρεση του ελάχιστου δένδρου – ή ελάχιστου δάσους, σε περίπτωση που ο γράφος είναι μη συνεκτικός – ακολουθείται ο εξής αλγόριθμος:

- Παίρνουμε το timestamp εκκίνησης του αλγορίθμου
- Δημιουργία ενός ComplexHashTable στο οποίο αποθηκεύονται:
    - Τα αναγνωριστικά όλων των κόμβων
    - Ο προηγούμενος κόμβος που επισεκπτόμαστε πριν τον τρέχοντα
    - Η απόσταση (βάρος) μετάβασης στο τρέχοντα κόμβο
- Δημιουργία ενός minHeap
- Μέχρι να μην υπαρχουν κόμβοι στο γράφημα που δεν έχουν επισκευθεί
    - Αρχικοποιήση τιμών
        - Στον πίνακα κατακερματισμού:
            - Ο προηγούμενος κόμβος σε –INT_MAX
            - Η απόσταση σε INT_MAX
        - Στον κόμβο από τον οποίο αρχίζει η διαδικασία:
            - Ο προηγούμενος κόμβος σε –INT_MAX (παραμένει σε όλη τη διάρκεια)
            - Η απόσταση σε 0
        - Στο σωρό ελαχίστων:
            - Οι γείτονες του κόμβου από τον οποίο ξεκινάμε
    - Μέχρι ο σωρός ελαχίστων να αδειάσει
        - Κάνουμε popMin
        - Ανανεώνουμε τους νέους κόμβους (γείτονες του min) σε minHeap και ComplexHashTable
    - Βρίσκουμε κόμβο που δεν έχουμε επισκευθεί επειδή βρίσκεται σε άλλο κομμάτι του γραφήματος (περίπτωση μη συνεκτικου γράφου) και επαναλαμβάνουμε τη διαδικασία.
- Παίρνουμε το τελικό timestamp
- Παίρνουμε το άθροισμα των αποστάσεων (κόστη) από το ComplexHashTable
- Επιστρέφουμε ένα struct με τα δύο ανώτερα αποτελέσματα.

Υπολογισμός Ελάχιστης Διαδρομής (Shortest Paths – SP) – Dijkstra's algorithm

Για την εύρεση της ελάχιστης διαδρομής, ξεκινώντας από δεδομένο κόμβο ακολουθήθηκε ο εξής αλγόριθμος:

- Δημιουργία ενός ComplexHashTable στο οποίο αποθηκεύονται:
    - Τα αναγνωριστικά όλων των κόμβων

    - Ο προηγούμενος κόμβος που επισεκπτόμαστε πριν τον τρέχοντα
    - Η απόσταση (βάρος) μετάβασης στο τρέχοντα κόμβο
- Αρχικοποιήση τιμών
    - Στον πίνακα κατακερματισμού:
        - Ο προηγούμενος κόμβος σε –INT_MAX
        - Η απόσταση σε INT_MAX
    - Στον κόμβο από τον οποίο αρχίζει η διαδικασία:
        - Ο προηγούμενος κόμβος σε –INT_MAX (παραμένει σε όλη τη διάρκεια)
        - Η απόσταση σε 0 (ώστε να μην επηρεάζει το συνολικό κόστος στο τέλος του αλγορίθμου)
- Δημιουργία ενός array που περιέχει τους κόμβους τους οποίους έχουμε επισκευθεί.
- Δημιουργία ενός σωρού ελαχίστου (minHeap) το οποίο
    - Αρχικοποιείται στους γείτονες το κόμβου από τον οποίο αρχίζουμε.
- Κάθε φορά που γίνεται popMin():
    - Εισάγουμε το min στο array set
    - Ενημερώνουμε την εγγραφή του min στο ComplexHashTable
    - Βάζουμε τους γείτονες του min στο minHeap και επεξεργαζόμαστε τα βάρη που βελτιώνονται,
- Στο τέλος, όταν ο σωρός ελαχίστων είναι κενός, επιστρέφεται το ComplexHashTable, το οποίο περιέχει τις τιμές που χρειαζόμαστε για όλες τις ελάχιστες διαδρομές.

Υπολογισμός Κοινών Γειτόνων (Common Neighbours – CN)

Για την εύρεση των κοινών γειτότονων, μεταξύ δύο κόμβων του γράφου, ακολουθείται η εξής διαδικασία:

- Αναζήτηση των κόμβων στο βασικό πίνακα κατακερματισμού και λήψη των αντίστοιχων AVL.
- Λήψη τομής των δύο δένδρων AVL
    - Με καστασκευή των inorder πινάκων για διάσχιση των δένδρων
    - Με κατασκευή ενός hashtable από το μεγαλύτερο AVL και συνεχή ερωτήματά από το άλλο AVL για ύπαρξη κόμβων με συγκεκριμένο αναγνωριστικό.
- Αποθήκευση κοινών κόμβων σε array και πλήθος τους.

SimpleHashTable class

SimpleHashTable	
public:	
SimpleHashTable(int sizeToInitializeTo)	*Constructor. Καλεί τον constructor SimpleHashTable(sizeToInitializeTo, -INT_MAX);*
SimpleHashTable(int sizeToInitializeTo, int defaultValue)	*Constructor. Δεσμεύει χώρο για το array του HashTable, αρχικοποιώντας τις μετεβλητές του size, του capacity και του defaultFillValue, και γεμίζει τα νέα κενά στοιχεία, με το defaultValue.*
virtual ~SimpleHashTable();	*Αποδεσμεύει τον χώρο του array.*
int getCapacity();	*Επιστρέφει την μέγιστη χωρητικότητα του HashTable.*
int getCurrentSize();	*Επιστρέφει το πλήθος των στοιχείων που έχουν εισαχθεί στο HashTable.*
bool addElement(int value);	*Εισάγει το στοιχείο με τιμή value, στο HashTable.*
bool exists(int toCompare);	*Επιστρέφει αν υπάρχει ή όχι η τιμή που δόθηκε.*
private:	
int hashFunction(int value);	*Hashing function. Επιστρέφει το hash, για την συγκεκριμένη τιμή που δόθηκε.*
int defaultFillValue	*Η default τιμή στην οποία αρχικοποιούνται όλα τα στοιχεία του HashTable.*
int* theArray;	*Το array του HashTable.*
int capacity;	*Η χωρητικότητα του HashTable.*
int currentSize;	*Το πλήθος των στοιχείων που υπάρχουν αυτή τη στιγμή στο HashTable.*

Λειτουργικότητα κλάσης

Πρόκειται για μία απλή υλοποίηση ενός HashTable.

Τα στοιχεία του πίνακα είναι ακέραιοι αριθμοί, ενώ για την εισαγωγή τους ακολουθείται μέθοδος ανοιχτής διευθυνσιοδότησης. Σαν συνάρτηση hashing, επιλέξαμε το υπόλοιπο της διαίρεσης με έναν μεγάλο πρώτο αριθμό, πάντα με αριθμητική modulo ως προς το μέγεθος του HashTable. Προκειμένου να αποφευχθούν οι συγκρούσεις με άλλα στοιχεία, αμέσως μετά την πράξη MOD με τον μεγάλο πρώτο αριθμό, υψώσαμε το αποτέλεσμα σε έναν σχετικά μικρό πρώτο αριθμό. Έτσι, εξασφαλίζεται μία ομοιόμορφη κατανομή σε όλες τις θέσεις του πίνακα, και μειώνεται το κόστος προσπέλασης των στοιχείων.

Να σημειωθεί, πως η εισαγωγή γίνεται με deep copy – το στοιχείο που δίνεται δηλαδή, αντιγράφεται στο array.

Η λειτουργία της διαγραφής δεν υποστηρίζεται από την δομή, καθώς σε οποιοδήποτε hashtable, η διαγραφή απαιτεί αρκετό υπολογιστικό κόστος, καθώς σε περιπτώσεις συγκρούσεων, μία διαγραφή θα απαιτούσε την μετακίνηση αρκετών στοιχείων στην σωστή τους θέση.

Υπολογιστικό κόστος

Η κατασκευή του SimpleHashTable γίνεται σε γραμμικό χρόνο, O(n), καθώς αμέσως μετά το allocation το μόνο που πραγματοποιείται είναι η αρχικοποίηση των στοιχείων σε μία προκαθορισμένη τιμή.

Το μέσο κόστος κατά την εισαγωγή ενός στοιχείου είναι σταθερό (O(1)). Ωστόσο, σημειώνεται πως στις περιπτώσεις που ο πίνακας έχει αρκετά στοιχεία (ο παράγοντας φόρτωσης του δηλαδή τείνει στην μονάδα), το πλήθος των συγκρούσεων κατά την εισαγωγή αυξάνεται. Έτσι λοιπόν, σε σχεδόν γεμάτα HashTable, το κόστος αυτό ενδέχεται να φτάσει σε O(n). Μιλώντας, όμως για την μέση περίπτωση, κάτι τέτοιο ευτυχώς αποφεύγεται, μέσα από την δέσμευση διπλάσιων θέσεων από το πλήθος των στοιχείων που θέλουμε να εισάγουμε.

ComplexHashTable class

ComplexHashTable	
public:	
struct ComplexHashEntry{ int id; int weight; int position; }	*Το struct που χρησιμοποιείται για την αποθήκευση των στοιχειών στο HashTable. Να σημειωθεί πως τα ονόματα των στοιχείων είναι σχετικά, και γενικά η δομή αυτή μπορεί να χρησιμοποιηθεί σε διαφορετικές εφαρμογές.*
ComplexHashTable(int sizeToInitializeTo)	*Default Constructor. Δεσμεύει χώρο για το array και αρχικοποιεί όλα τα στοιχεία σε nullptr.*
virtual ~ComplexHashTable();	*Destructor. Σβήνει τον πίνακα με τους δείκτες του array.*
void onDestroy();	*Θα πρέπει να κληθεί από τον προγραμματιστή πριν την οριστική διαγραφή του ComplexHashTable. Σβήνει όλα τα αντικείμενα από entries που έχει δημιουργήσει. Δεδομένου ότι τα στοιχεία επιστρέφονται αναφορικά, με δείκτες, η λειτουργία της συνάρτησης αυτής δεν θα μπορούσε να βρίσκεται στον destructor.*
int getCapacity();	*Επιστρέφει την μέγιστη χωρητικότητα του HashTable.*
int getCurrentSize();	*Επιστρέφει το πλήθος των στοιχείων που έχουν εισαχθεί στο HashTable.*
complexHashEntry* getElement(int value);	*Επιστρέφει το στοιχείο με id = value, μέσα από το HashTable. Επιστρέφεται αναφορικά, προκειμένου να επιτραπεί η πιθανή επεξεργασία των τιμών των πεδίων του.*
friend ostream& operator<< (ostream& myStream, ComplexHashTable& ob)	*Operator overload*
bool addElement(complexHashEntry* value);	*Εισάγεται το στοιχείο value στο HashTable. Πραγματοποιώντας deep-copy.*
bool deleteElement(int id);	*Διαγράφεται το στοιχείο με id = id από το HashTable. Δεν πραγματοποιείται πραγματική διαγραφή και αποδέσμευση, αλλά το στοιχείο βρίσκεται πλέον σε μία κατάσταση λάθους, με position = -1 και weight = 0.*
bool exists(int toCompare);	*Επιστρέφει αν υπάρχι η όχι το στοιχείο με id = toCompare στο HashTable (true αν το βρει – false αν το βρει με position -1 ή αν βρει nullptr).*
complexHashEntry* getFirstSpecificOccurence(int valueToSearch);	*Επιστρέφει δείκτη με το πρώτο διαθέσιμο στοιχείο που έχει weight == valueToSearch. Χρησιμοποιείται στον αλγόριθμο του MST.*
private:	
int hashFunction(int value);	*Hashing function. Επιστρέφει το hash, για την συγκεκριμένη τιμή που δόθηκε.*
complexHashEntry* theArray;	*Το array του HashTable.*
int capacity;	*Η χωρητικότητα του HashTable.*
int currentSize;	*Το πλήθος των στοιχείων που υπάρχουν αυτή τη στιγμή στο HashTable.*

Λειτουργικότητα κλάσης

Πρόκειται για μία πιο εξελιγμένη υλοποίηση ενός HashTable.

Τα στοιχεία του πίνακα είναι structs με 3 πεδία (id, position, weight), ενώ για την εισαγωγή τους ακολουθείται μέθοδος ανοιχτής διευθυνσιοδότησης. Σαν συνάρτηση hashing, επιλέξαμε το υπόλοιπο της διαίρεσης με έναν μεγάλο πρώτο αριθμό, πάντα με αριθμητική modulo ως προς το μέγεθος του HashTable. Προκειμένου να αποφευχθούν οι συγκρούσεις με άλλα στοιχεία, αμέσως μετά την πράξη MOD με τον μεγάλο πρώτο αριθμό, υψώσαμε το αποτέλεσμα σε έναν σχετικά μικρό πρώτο αριθμό. Έτσι, εξασφαλίζεται μία ομοιόμορφη κατανομή σε όλες τις θέσεις του πίνακα, και μειώνεται το κόστος προσπέλασης των στοιχείων.

Να σημειωθεί, πως η εισαγωγή γίνεται με deep copy – το στοιχείο που δίνεται δηλαδή, αντιγράφεται στο array.

Η λειτουργία της διαγραφής δεν υποστηρίζεται από την δομή, καθώς σε οποιοδήποτε hashtable, η διαγραφή απαιτεί αρκετό υπολογιστικό κόστος, καθώς σε περιπτώσεις συγκρούσεων, μία διαγραφή θα απαιτούσε την μετακίνηση αρκετών στοιχείων στην σωστή τους θέση.

Υπολογιστικό κόστος

Η κατασκευή του ComplexHashTable γίνεται σε γραμμικό χρόνο, O(n), καθώς αμέσως μετά το allocation το μόνο που πραγματοποιείται είναι η αρχικοποίηση των στοιχείων σε nullptr.

Το μέσο κόστος κατά την εισαγωγή ενός στοιχείου είναι σταθερό (O(1)). Ωστόσο, σημειώνεται πως στις περιπτώσεις που ο πίνακας έχει αρκετά στοιχεία (ο παράγοντας φόρτωσης του δηλαδή τείνει στην μονάδα), το πλήθος των συγκρούσεων κατά την εισαγωγή αυξάνεται. Έτσι λοιπόν, σε σχεδόν γεμάτα HashTable, το κόστος αυτό ενδέχεται να φτάσει σε O(n). Μιλώντας, όμως για την μέση περίπτωση, κάτι τέτοιο ευτυχώς αποφεύγεται, μέσα από την δέσμευση διπλάσιων θέσεων από το πλήθος των στοιχείων που θέλουμε να εισάγουμε.

MinHeap class

MinHeap	
public:	
struct minHeapEntry **{** **int id;** **int weight;** **}**	*Δεδομένου ότι δεν είναι ανάγκη να αποθηκεύουμε όλα τα treeNodes μέσα στο minHeap, δημιουργούμε ένα struct που θα αποτελεί τον κύριο κορμό του minHeap και θα περιέχει μόνο ids και weights.*
MinHeap(int size);	*Default constructor. Δεσμεύεται χώρος (μεγέθους size) για το MinHeap αλλά και για το ComplexHashTable που θα χρησιμοποιηθεί για indexing. Όλοι οι δείκτες αρχικοποιούνται σε nullptr.*
MinHeap(AVL* anAVL, int size)	*Καλεί αρχικά τον constructor MinHeap(size).* *Έπειτα, τοποθετεί στο MinHeap διαδοχικά όλους τους κόμβους του AVL (μέσα από την inorder και πραγματοποιώντας deep copy) χωρίς να αντιμεταθέτει στοιχεία.* *Τέλος, καλεί την makeHeap() προκειμένου να κατασκευαστεί ο σωρός.*
bool addElement(treeNode* treeNodeToInsert);	*Καλεί την addElement(treeNodeToInsert, 0);*
bool addElement(treeNode* treeNodeToInsert, int toBeIncreased)	*Πραγματοποιεί εισαγωγή του στοιχείου στο minheap (deep copy) αυξάνοντας το weight του κατά toBeIncreased. Η μέθοδος αυτή χρησιμοποιείται αρκετά στον αλγόριθμο του Dijkstra, όπου πρέπει να προσθέτουμε το βάρος που βρίσκεται στην αντίστοιχη εγγραφή του previous array.*
minHeapEntry getMin();	*Επιστρέφει το ελάχιστο (κατά βάρος) στοιχείο του minheap, χωρίς να το διαγράφει.*
minHeapEntry* getElement(int id);	*Επιστρέφει με αναφορά το στοιχείο με το id της παραμέτρου. Να σημειωθεί, πως λόγω της υπάρξης του HashTable, η πολυπλοκότητα της μεθόδου αυτής είναι σημαντικά μειωμένη.*
minHeapEntry popMin();	*Διαγράφει και επιστρέφει το ελάχιστο στοιχείο του minHeap, καλώντας την αντίστοιχη μέθοδο deleteElement του HashTable.*
bool editById(int id, int value);	*Επεξεργάζεται το στοιχείο με το id που δίνεται, ενημερώνοντας το βάρος του στην τιμή που δίνεται στο value.*
bool isEmpty();	*Επιστρέφει αν το minHeap είναι άδειο ή όχι.*
protected:	
minHeapEntry theMinHeap;**	*Το array από εγγραφές τύπου minHeapEntry του minheap.*
ComplexHashTable* theIndex;	*Δείκτης σε αντικείμενο τύπου ComplexHashTable για indexing των αρχείων που υπάρχουν στο minHeap.*
int currentSize;	*Το τρέχον μέγεθος του MinHeap (το πλήθος των στοιχείων που υπάρχουν μέσα)*
int maxSize;	*Το μέγιστο μέγεθος που έχει δεσμευτεί για το MinHeap.*
private:	
bool makeHeap();	*Μέθοδος που μετατρέπει το αρχικό array με τις τιμές σε σωρό, σε γραμμικό χρόνο.*
void checkUpper(int position);	*Μέθοδος που ελέγχει όλους τους κόμβους που βρίσκονται σε ανώτερο επίπεδο από τον κόμβο που βρίσκεται στην θέση position, προκειμένου να πραγματοποιηθούν οι αντιμεταθέσεις που χρειάζονται. Καλείται στην εισαγωγή και στην επεξεργασία των στοιχείων.*

void checkLower(int position);	*Μέθοδος που ελέγχει όλους τους κόμβους που βρίσκονται σε κατώτερο επίπεδο από τον κόμβο που βρίσκεται στην θέση position, προκειμένου να πραγματοποιοηθούν οι αντιμεταθέσεις που χρειάζονται. Καλείται στην εισαγωγή, στην διαγραφή του min και στην επεξεργασία των στοιχείων.*

Λειτουργικότητα κλάσης

Πρόκειται για την κλάση που χρησιμοποιείται κατά κύριο λόγο στους αλγορίθμους του Dijkstra και του Prim.

Χρησιμοποιώντας ένα από array σε pointers από structs τύπου minHeapEntry, αποθηκεύουμε το id και το βάρος κάθε ακμής. Ωστόσο, καθώς οι αλγόριθμοι απαιτούν την επεξεργασία στοιχείων και την λειτουργία της αναζήτησης, προκειμένου αυτή να είναι αποδοτική, χρησιμοποιήθηκε και ένα αντικείμενο της κλάσης ComplexHashTable.

Έτσι, κάθε εγγραφή στο HashTable, περιέχει την θέση κάθε id στο minheap, καθώς και το βάρος προκειμένου να βελτιωθεί το κόστος κατά την προσπέλαση. Κάθε φορά που αναζητούμε ένα στοιχείο μέσα στο minHeap, λαμβάνουμε το στοιχείο από το HashTable (σταθερό κόστος) και έπειτα προσπελαύνουμε την αντίστοιχη θέση (position) στο HashTable. Φυσικά, η διαδικασία αυτή μπορεί να πραγματοποιηθεί και στην αντίθετη φορά. Δηλαδή, κάθε φορά που πραγματοποιείται κάποια αντιμετάθεση ή ενημέρωση βαρών στο minHeap, κάτι τέτοιο αποτυπώνεται και στο HashTable.

Ιδιαίτερη προσοχή πρέπει να δοθεί, όταν ένα στοιχείο έχει εξαχθεί από το MinHeap. Προκειμένου ότι η διαγραφή είναι αρκετά κοστοβόρα στο HashTable, τοποθετούμε απλώς στο αντίστοιχο position στο HashTable, την τιμή -1. Έτσι, λοιπόν, θα λέμε ότι ένα στοιχείο δεν υπάρχει στο minHeap όταν ο αντίστοιχος pointer στο HashTable είναι nullptr (δεν έχει καν εισαχθεί) ή όταν η τιμή του position είναι -1.

Υπολογιστικό κόστος

Κατασκευή του MinHeap

Η κατασκευή του σωρού ελαχίστων γίνεται σε γραμμικό χρόνο, O(n), καθώς τοποθετούμε τα στοιχεία μαζικά, και πραγματοποιούμε έναν down-to-top έλεγχο για τις αντιμεταθέσεις που χρειάζονται, μέσα από την συνάρτηση makeHeap().

Εισαγωγή στοιχείων

Το κόστος για την εισαγωγή των στοιχείων στο array του MinHeap είναι λογαριθμικό, O(logn), καθώς θα χρειαστούν το πολύ logn συγκρίσεις για να βρεθεί η σωστή θέση του νέου στοιχείου στον σωρό.

Το κόστος για την εισαγωγή των στοιχείων στο index HashTable, είναι σταθερό. Σε περίπτωση collision, ίσως έχουμε κάποιους παραπάνω ελέγχους, που όμως διαχειρίζεται η κλάση του ComplexHashTable.

Διαγραφή ελαχίστου

Η εξαγωγή του ελάχιστου στοιχείου στο minHeap γίνεται με λογαριθμικό κόστος, O(logn). Το στοιχείο εξάγεται σε σταθερό χρόνο, O(1) αλλά ακολουθούν στην συνέχεια το πολύ logn επαναλήψεις, καθώς ανεβαίνει στην ρίζα το τελευταίο στοιχείου του σωρού και πρέπει να γίνουν οι κατάλληλες αντιμεταθέσεις.

Για να σηματοδοτήσουμε την εξαγωγή του στοιχείου, καλούμε την μέθοδο deleteElement του ComplexHashTable, η λειτουργία της οποίας ήδη έχει αναφερθεί.

Επεξεργασία στοιχείων

Το κόστος για την επεξεργασία ενός στοιχείου του minHeap είναι λογαριθμικό, δηλαδή O(logn). Το στοιχείο μπορεί να ευρεθεί άμεσα στο minHeap, μέσα από το index HashTable. Ωστόσο, μετά την αλλαγή του βάρους μίας εγγραφής, μπορεί να χρειαστούν logn αντιμεταθέσεις στο minHeap.

IO class

IO	
public:	
IO(); **virtual ~IO();**	*Default constructor/destructor. Καθώς πρόκειται για μία βοηθητική κλάση, δεν έχει πρακτικό νόημα η δημιουργία αντικειμένων.*
bool readCommands(char * filename, Database * db);	*Μέθοδος που καλείται προκειμένου να διαβαστεί το αρχείο commands.txt και να γίνει το parsing στις εντολές που δόθηκαν.*
bool readInput(char* filename, Database * db);	*Μέθοδος που καλείται προκειμένου να διαβαστεί το αρχείο input.txt και να φορτωθούν στο αντικείμενο της Database τα αρχικά links μεταξύ των node.*
bool writeIndex(char * filename, Database * db);	*Μέθοδος που καλείται προκειμένου να παραχθεί το αρχείο output.txt, που είναι και το τελικό αποτέλεσμα του προγράμματος.*
bool writeCN(char* filename, int numberOfCommonNeighbours, int nodeID1, int nodeID2)	*Μέθοδος που καλείται, όταν διαβαστεί η εντολή CN x y από το αρχείο εισόδου. Τότε, το αποτέλεσμα των κοινών γειτόνων (πλήθος) των 2 κόμβων (βρίσκεται μέσα από το intersection των 2 AVL τους), γράφεται στο αρχείο με όνομα filename.*
bool writeMST(char* filename, Database::resultOfMST mstResult);	*Μέθοδος που καλείται, όταν διαβαστεί η εντολή MST από το αρχείο εισόδου. Τότε, το αποτέλεσμα του MST (ή άθροισμα MST) γράφεται στο αρχείο με όνομα filename.*
bool writeDijkstra(char* filename, ComplexHashtable* dijkstraResult, int startingID)	*Μέθοδος που καλείται, όταν διαβαστεί η εντολή SP startingID από το αρχείο εισόδου. Τότε το αποτέλεσμα του αλγορίθμου του Dijkstra γράφεται στο αρχείο με όνομα filename.*
private:	
struct command **{** **char* commandName;** **int argc;** **char* argv[3];** **};** **typedef struct command command;**	*Struct που χρησιμοποιείται από την κλάση, για την αναπαράσταση μίας εντολής. Περιέχει μία συμβολοσειρά με το όνομα της εντολής, έναν ακέραιο αριθμό με το πλήθος των ορισμάτων. Τα ορίσματα αποθηκεύονται στον πίνακα argv.*
command parseLine(char input[]);	*Μέθοδος που κάνει parsing σε μία συμβολοσειρά, προκειμένου να αναγνωρίσει και να παράγει ένα αντικείμενο command.*

Λειτουργικότητα κλάσης

Πρόκειται για την κλάση που παρέχει την διασύνδεση των δομών μας με τον χρήστη. Μέσα από αυτή την κλάση, διαβάζουμε το αρχείο των εντολών (readCommands), εισάγουμε τα αρχικά δεδομένα της database (readInput) και εξάγουμε το τελικό αποτέλεσμα (writeIndex). Κάθε εντολή που διαβάζεται από το αρχείο κειμένου commands.txt, «ερμηνεύεται» από την parseLine, προκειμένου να εκτελεστεί η κατάλληλη λειτουργία που ορίζετε από την Database.

Υπολογιστικό κόστος

Το κόστος των συναρτήσεων της κλάσης αυτής, δεν παρουσιάζει ιδιαίτερο ενδιαφέρον, αφού δεν υπάρχει κάποιο αξιοποιήσιμο περιθώριο βελτίωσης ή ελαχιστοποίησης αυτού. Στις μεθόδους readCommands, readInput και writeIndex, το εκάστοτε κόστος θα είναι γραμμικό (O(n)). Πχ. για τη readCommands θα διαβαστούν n γραμμές από το commands.txt, για την readInput θα διαβαστούν n γραμμές από το input.txt ενώ για το writeIndex θα αποθηκευτούν τα δεδομένα από n στοιχεία που βρίσκονται στην database την στιγμή κλήσης της. Η πολυπλοκότητα των εντολών MST, SP και CN ανάγεται στην πολυπλοκότητα των αντίστοιχων αλγορίθμων υπολογισμού τους (βλ. τις αντίστοιχες κλάσεις και μεθόδους).

Input/Output

Μορφή εντολών

Οι εντολές που δίνονται από τον χρήστη, για την εκτέλεση του προγράμματος, βρίσκονται αποθηκευμένες σε ένα αρχείο με όνομα commands.txt στο φάκελο του εκτελέσιμου αρχείου. Κατά την εκκίνηση του προγράμματος, προσπελαύνονται σειρά ανά σειρά οι εντολές αυτές, και εκτελείται ο κατάλληλος κώδικας.

Οι εντολές που υποστηρίζονται από το πρόγραμμα είναι οι ακόλουθες:

- READ_DATA <όνομα αρχείου>

  Πρόκειται ουσιαστικά για την εντολή αρχικοποίησης με δεδομένα των δομών του προγράμματος. Στο αρχείο που δίνεται σαν όρισμα από τον χρήστη, βρίσκονται αποθηκευμένες, οι ακμές των συνδεδεμένων κόμβων. Το όνομα αρχείου δίνεται ως relative path και αναλόγως του λειτουργικού, ο χρήστης πρέπει να θέσει στο filename το σωστό path. (π.χ. windows: .\myFiles\input.txt, linux: ./myFiles/input.txt, os x: ./myFiles/input.txt)

- INSERT_LINK <ακέραιος α> <ακέραιος β> <ακέραιος γ>

  Με την εντολή αυτή, ο χρήστης προσθέτει μία ακόμα ακμή – σύνδεσμο στην δομή, από την σελίδα με μοναδικό αναγνωριστικό α στην σελίδα με μοναδικό αναγνωριστικό β και με βάρος γ.

  Να σημειωθεί, πως η ακμή (α,β,γ) θα πρέπει να υπάρχει στον γράφο και στην μορφή (β,α,γ) καθώς εξετάζουμε μη κατευθυνόμενα γραφήματα. Έτσι λοιπόν, μετά την εισαγωγή τις ακμής (β,γ) στον κόμβο α, πραγματοποιούμε και αντίστοιχη εισαγωγή ακμής (α,γ) στον κόμβο β.

  Αν η ακμή υπάρχει ήδη, τότε το πρόγραμμα συνεχίζει την εκτέλεση του, χωρίς να έχει πραγματοποιήσει καμία παραπάνω εντολή. Αν πρόκειται για έναν κόμβο α που δεν υπάρχει στην δομή του Database, τότε δημιουργείται και εισάγεται στον ταξινομημένο πίνακα.

- DELETE_LINK <ακέραιος α> <ακέραιος β>

  Με την εντολή αυτή, ο χρήστης αφαιρεί μία ακμή – σύνδεσμο στην δομή. Συγκεκριμένα αφαιρεί τον σύνδεσμο που ξεκινάει από την σελίδα με id = α και καταλήγει στην σελίδα με id = β. Αν η ακμή αυτή δεν υπάρχει, τότε το πρόγραμμα συνεχίζει κανονικά την εκτέλεση του, χωρίς να έχει πραγματοποιήσει καμία παραπάνω εντολή.

  Η λειτουργία του DELETE_LINK όπως και κατά την εισαγωγή είναι διπλή: Σβήνουμε τόσο τον κόμβο β από το AVL του κόμβου α, αλλά και τον κόμβο α από το AVL του κόμβου β.

- WRITE_INDEX <όνομα αρχείου>

  Αποτελεί ίσως την πιο σημαντική εντολή στην εκτέλεση του προγράμματος, αφού χωρίς αυτή την εντολή, οποιαδήποτε δομή και αν δημιουργήθηκε στο πρόγραμμα, καταστρέφεται, προτού δοθεί η ζητούμενη έξοδος στον χρήστη.

  Σε αντιστοιχία με την READ_DATA, το όνομα του αρχείου προς αποθήκευση των δεδομένων δίνεται ως relative path.

- MST
  Με την εντολή αυτή, εκτελείται ο αλγόριθμος του Prim για την εύρεση του ελάχιστα εκτεινόμενου δένδρου από τον υπάρχοντα γράφο. Αυτό έχει σαν αποτέλεσμα την έξοδο του κόστους από το ελάχιστα εκτεινόμενου δένδρου, καθώς και τον χρόνο που χρειάστηκε για τον υπολογισμό του. Να σημειωθεί, πως στην περίπτωση των μη συνεκτικών γραφημάτων ο αλγόριθμος συνεχίζει, μέχρι να επισκεφτούν όλες οι κορυφές.

- SP <ακέραιος α>
  Με την εντολή αυτή εκτελείται ο αλγόριθμος του Dijkstra για την κορυφή α, δίνοντας στην έξοδο τα κόστος της ελάχιστης διαδρομής από την κορυφή α σε όλες τις υπόλοιπες.

- CN <ακέραιος α> <ακέραιος β>
  Με την εντολή αυτή, εκτελείται ο αλγόριθμος εύρεσης κοινών γειτόνων, στα AVL των κόμβων <α> και <β> και εκτυπώνεται στην έξοδο το πλήθος των κοινών αυτών κορυφών.

Αξίζει στο σημείο αυτό, να σημειωθεί ότι για λόγους απόδοσης παραλήφθηκαν οι έλεγχοι εγκυρότητας για την μορφή των παραπάνω εντολών. Για τον λόγο αυτό, οι εντολές INSERT_LINK και DELETE_LINK θα πρέπει πάντα να ακολουθούνται από 2 ακέραιους αριθμούς, χωρισμένων με ένα κενό, ενώ οι εντολές READ_DATA και WRITE_INDEX θα πρέπει να ακολουθούνται από ένα όνομα αρχείου προσβάσιμο κατά την εκτέλεση του προγράμματος.

Σημείωση: Η διάκριση των εντολών (READ_DATA, INSERT_LINK, DELETE_LINK, WRITE_INDEX) από τις παραμέτρους τους, πρέπει να γίνεται με χρήση **ενός** κενού διαστήματος. *(delimeter = ' ')*

Μορφή input

Το κείμενο στο αρχείο input.txt που δίνεται σαν όρισμα όταν καλείται η λειτουργία READ_DATA του προγράμματος έχει, όπως προείπαμε την μορφή:

<ακέραιος a_1> <ακέραιος a_2>
<ακέραιος β_1> <ακέραιος β_2>
……
<ακέραιος κ_1> <ακέραιος κ_2>

όπου κ οι γραμμές του αρχείου.

Όσον αφορά την δομή των γραμμών, ο χρήστης θα πρέπει να εισάγει άρτιο πλήθος αριθμών (id σελίδων), καθώς το πρόγραμμα αντιλαμβάνεται δυάδες από id, προκειμένου να εισάγει την αντίστοιχη ακμή. Ανάμεσα στους αριθμούς, δεν έχει καμία ιδιαίτερη βαρύτητα ο αριθμός των white spaces που θα χρησιμοποιηθούν από τον χρήστη. Ωστόσο, η εισαγωγή μη αριθμητικών χαρακτήρων, θα έχει ως αποτέλεσμα την αστάθεια του προγράμματος.

Σελίδες με αρνητικά id είναι επίσης αποδεκτές από το πρόγραμμα (δηλαδή όταν προκύπτει κάποιος αρνητικός ακέραιος στην προσπέλαση του αρχείου), χωρίς ωστόσο να γίνεται προφανής η χρήση και το νόημα μίας τέτοιας ακμής.

Μορφή output

Για την αποθήκευση των δεδομένων του προγράμματος, χρησιμοποιούνται οι υπερφορτωμένοι τελεστές των κλάσεων Database, Node, ComplexHashTable και AVL.

Αναλόγως με το filename το οποίο έχει δοθεί στο commands.txt, αποθηκεύει τα δεδομένα της Database στο αρχείο "filename".

Όταν δίνεται σαν είσοδος η εντολή WRITE_INDEX εκτυπώνονται στην έξοδο του αρχείου τα στοιχεία-κόμβοι που είναι αποθηκευμένα στην δομή με την εξής μορφή:

<Node.Id_1>, <NumberOfConnections>, <AVL_in_order>

<Node.Id_2>, <NumberOfConnections>, <AVL_in_order>

…

<Node.Id_N>, <NumberOfConnections>, <AVL_in_order>

Όπου N το πλήθος των κόμβων που έχουμε αποθηκεύσει στη Database.

Όταν δίνεται σαν είσοδος η εντολή MST, εκτυπώνεται στην έξοδο το συνολικό κόστος του ελάχιστου εκτεινόμενου δένδρου, καθώς και ο χρόνος που χρειάστηκε για τον υπολογισμό του στην ακόλουθη μορφή:

MST: Cost: <cost> - Time: <time>

Όταν δίνεται σαν είσοδος η εντολή SP <αρχικός κόμβος> , εκτυπώνονται στην έξοδο τα τελικά ελάχιστα κόστη από τον κόμβο αυτόν στην ακόλουθη μορφή:

Dijkstra from <Node.Id_start>, < Node.weight_1>(id: < Node.Id_1>), …, < Node.Weight_n>(id: < Node.Id_n>)

Όπου n το πλήθος των κόμβων του γραφήματος.

Όταν δίνεται σαν είσοδος η εντολή CN <κόμβος 1> <κόμβος 2> , εκτυπώνεται στην έξοδο το πλήθος των κοινών γειτόνων των κόμβων αυτών στην ακόλουθη μορφή:

Common neighbours (<Node.Id_1> , <Node.Id_2>) : <NumberOfCommonNeighbours>

Μελλοντικές Βελτιώσεις

Σε ενδεχόμενες μελλοντικές εκδόσεις του συγκεκριμένου προγράμματος-βιβλιοθήκης, θα εντοπίζαμε τις εξής βελτιώσεις σε επίπεδο κώδικα, υλοποίησης και βελτιστοποίησης:

- Χρήση templates για γενικότητα κώδικα
    - Θα μπορούσαν να ενοποιηθούν οι πίνακες κατακερματισμού σε μία κλάση
- Χρήση πίνακα κατακερματισμού αντί για array στο set των κόμβων που έχουμε επισκευθεί στον αλγόριθμο του Dijkstra
- Δυναμικός πίνακας κατακερματισμού και υποστήριξη delete, σε περίπτωση που είναι απαραίτητο.
    - Rehashing κατά τη διαγραφή εγγραφών μέχρι την επόμενη σωστή εγγραφή.
    - Reallocation και rehashing για δυναμικό array.
- Εκμετάλλευση των εξαιρέσεων για προβλήματα τα οποία μπορεί να προκύψουν κατά την εκτέλεση του προγράμματος.
- Μικρές αλλαγές στο κώδικα ώστε να είναι συμβατός με compilers σε λειτουργικά συστήματα πέραν των Windows®.