How to launch:

- In console: $ make execParam=n, αντί n το επιθυμητό μέθοδο αποθήκευσης δεδομένων (1 - Array, 2 - AVL Tree, 3 - Hash Table)
- Ο κώδικας είναι heavy commented για αυτό στο report αναφέρουμε μόνο στο πως αναπτύξαμε τον κώδικα

Γενικά concepts:

- Θέλαμε να κάνουμε όλους μεθόδους πιο unified όσα γίνεται, για αυτό χρησιμοποιούμε namespaces. Όμως τελικά αντιμετωπίσαμε ένα πρόβλημα που λίγο χάλασε αυτήν προσέγγιση (σε ιδιά functions για διαφορετικούς μεθόδους διαχείρισης δεδομένων χρειάστηκαν διαφορετικά parameters, αναφέρεται πιο κάτω)
- Προσέξαμε πως το main array και arrays of neighbours απαιτούν περίπου ιδιά δεδομένα (βασικά main array θέλει και τα δεδομένα που περιγράφουν το array of neighbours, άλλα τα μέλη των arrays θέλουν ιδιά δεδομένα). Για αυτό ανήκουν στην ίδια οικογένεια.
- Επειδή χρειάζεται να κρατήσουμε δεδομένα που έχουν πολυεπίπεδη δομή. Πρέπει να χρησιμοποιήσουμε 2D Arrays. Αφού θα χρησιμοποιήσουμε std::vector αποφασίσαμε επίσης να μην χρησιμοποιήσουμε 2D Array of Pointers (θα εχουμε instances of classes μέσα στους πίνακες), λόγο των optimizations που κάνει compiler με το vector όταν περιέχει τα instances of objects και επειδή έτσι θα χρειαστεί λιγότερο dereferencing, θα αυξηθεί αναγνωσιμότητα του κώδικα και, θεωρητικά, στο c++ χρησιμοποιώντας references δεν θα έχουμε επιπλέον καθυστέρηση. Όμως το βασικό είναι ότι όταν vector κάνει resizing οι παλιές διευθύνσεις μνήμης γίνονται invalid και είναι πιθανή η δημιουργία του unexpected behaviour.
- Προσπαθήσαμε να αποφεύγουμε όσα περισσότερα copying των variables, για αυτό χρησιμοποιούμε σχεδόν παντού references και emplace_back για την εισαγωγή στο vector.
- Δεν χρησιμοποιήσαμε σχεδόν καθόλου private στις κλάσεις και γράψαμε όλα σε ίδιο αρχείο για διευκόλυνση ανάπτυξης του προγράμματος

Γενικές απορίες:

- Τα functions δεν ήταν πιθανό να είναι unified επειδή θέλουν διαφορετικά πράγματα να πασάρουμε, μπορέσαμε να κάνουμε typedef αλλά έτσι θα χάσουμε αναγνωσιμότητα, άρα αποφασίσαμε να κάνουμε copy-paste για διαφορετικούς τύπους δεδομένων.
- Είχαμε θέμα με line endings στο Linux, είχαμε αρχείο input.txt από windows που είχε CRLF endings και αυτά δεν αναγνώρισε σωστά Linux build όταν κάναμε τεστ, μέχρι που καταλάβαμε ότι φταίει αυτό (επειδή όλες γραμμές στο output.txt ήταν σωστές εκτός από τελευταία και πρώτη) πήρε κάποιες ώρες. Επίσης δεν καταφέραμε να ανοίξουμε κάτω από gcc και χρησιμοποιούμε g++ compiler επειδή βασιζόμαστε στα πράγματα από c++11, είτε απλώς δεν βρήκαμε όλα τα απαιτούμενα flags για linker.
- Θέλαμε να κάνουμε και parallelizing για εισαγωγή με πολλά threads, όμως βγήκε πολύ δύσκολο αυτό και μας προσφέρει κάπου 5% επιτάχυνση που δεν αξίζει. Αρχικά, χρησιμοποιήσαμε std::thread, όμως για περισσότερα functions χρειάστηκε να επιστρέφει και value που δεν υποστηρίζεται, και χρειάστηκε να χρησιμοποιήσουμε std::future, όμως δεν μπορούμε απλώς να πασάρουμε references χρειάστηκε και reference wrapper, όμως τελικά δούλεψε μισό που δεν πρόσφερε κάτι σημαντικό και χρειάστηκε και κάτι άλλο για να μην κρασάρει, τελικά αποφασίσαμε να κάνουμε όλα μέσα σε main thread, ούτως ή άλλως compiler θα διαχειρίζεται τα threads καλυτέρα από μας. Ωστόσο, μαθαίναμε πολλά πράγματα.

Απορίες για Array:

- Αρχικά φτιάξαμε array όπως πήραμε δεδομένα (δεν κάναμε sorting) και ο χρόνος εισαγωγής(επειδή κάνει αναζήτηση πριν εισαγωγή)/αναζήτησης/διαγραφής ήταν χάλια. Και αλλάξαμε όλα για να υποστηρίζει δυαδική αναζήτηση, δηλαδή ταξινομημένο (κάνει εισαγωγή στην σωστή θέση).
- Vector of vectors έχει πολύ μπερδεμένο syntax για αυτό αποφασίσαμε να χρησιμοποιήσουμε το ίδιο concept από άλλες μεθόδους (vector of objects που κρατάνε value + vector of neighbours)

Απορίες για AVL Tree:

- Επειδή χρειάζεται να αλλάζουμε πάρα πολλά δεδομένα όταν κάνουμε rotation ξεχάσαμε να υπολογίζουμε αλλαγές στα παιδιά των παιδιών και στους γονείς των γονέων επειδή το πρωτότυπο του αλγορίθμου τεστάραμε σε ένα σχετικά απλό δένδρο που δεν προσέξαμε τέτοια εξάρτηση.
- Είχαμε πρόβλημα που δεν καταλάβαμε απολύτως concept με τα βάρη των nodes. Κρατήσαμε μόνο βάρη όλων (είχαμε φόρμουλα Weight = RChild.Weight – Lchild.Weight που δεν παράγει πάντα σωστό αποτέλεσμα), αλλά χρειάστηκε να κρατήσουμε μόνο height και να υπολογίζουμε τα βάρη όταν κάνουμε update, ως αποτέλεσμα τώρα κρατάμε και Height και Weight σε κάθε nodes (τώρα που κρατάμε Weight είναι περιττό, αλλά δεν πρέπει να το υπολογίζουμε ξανά Weight όταν κάνουμε διαγραφές και εισαγωγές (όταν δεν χρειάζεται rotation) άρα προσφέρει κάτι)
- Όπως ανακάλυψε std::cout δεν μπορεί να εμφανίζει int values μέσα σε ternary operations. Βάλαμε for loops.
- Παντού χρειάστηκαν exceptions handling, όταν δεν έχει παιδί return, όταν παιδί του παιδιού δεν υπάρχει return, όταν γονέας του γονέα δεν υπάρχει κτλ. Επίσης μεγάλα προβλήματα δημιούργησε η ανανέωση της κορυφής. Χρειάστηκε να προσέξουμε πολλές περιπτώσεις μέσα στα rotations.
- Η διαδικασία της διαγραφής ήταν πάρα πολύ δύσκολη. Μάλλον και επιλέξαμε και δύσκολο τρόπο, όμως θέλαμε να κάνουμε όσα πιο σωστά και γρήγορα γίνεται. Δεν μπορέσαμε να κάνουμε διαγραφή του στοιχείου αμέσως, επειδή έτσι θα χάσουμε όλες εξαρτήσεις των επομένων στοιχείων (μπορεί να κάνει resizing το vector). Επίσης απλώς να βάζουμε tomb value δεν είναι καθόλου optimised. Για αυτό αποφασίσαμε να κάνουμε swap με το τελευταίο στοιχείο και να το διαγράψουμε. Όμως απαίτησε να διατηρούμε όλες σχέσεις τούτου και να έχουμε ειδική προσέγγιση αν είναι κορυφή ή αν ο ίδιος είναι το τελευταίο στοιχείο του array (τελευταία περίπτωση δημιούργησε αόριστο bug το οποίο να καταλάβουμε πήρε μια μέρα).

Σημειώσεις για AVL Tree:

- Weight Update Logic optimisations: κάνουμε update μέχρι που συναντήσομε στοιχείο που θέλει rotation (!(-1<Weight<1)) επειδή τα άλλα στοιχεία που είναι πάνω του θα είναι invalid μετά. Επίσης όταν συναντάμε node στο οποίο δεν αλλάζουμε height σημαίνει ότι και στα nodes που είναι πιο πάνω δεν θα αλλάξει τίποτα άρα μπορούμε να κάνουμε return.
- Για debugging φτιάξαμε Visualiser που δίνει οδηγίες για να ζωγραφήσω το δένδρο, και depth counter για να ελέγξουμε αν το δένδρο έχει σωστή μορφή.

Απορίες για Hash Table:

- Η φόρμουλα για key offset στο open probing που βρήκα στο διαδίκτυο μέσα στο concept για quadratic probing δεν δούλεψε. Και χρειάστηκα να την αλλάξω $από\ key\ +\ (x\text{^}2 + x)/2$ σε $key\ +\ (x\text{^}2)/2$
- Το built-in mod operator (%) δεν παράγει σωστά αποτελέσματα για πολύ μεγάλους αριθμούς, για αυτό φτιάξαμε δικό μας.

Σημειώσεις για Hash Map:

- Επιλέξαμε να χρησιμοποιήσουμε open hashing για να είναι πιο αποδοτικό και πιο ενδιαφέρον στην υλοποίηση. Επιλέξαμε quadratic probing που απαιτείται να κρατάμε array size = pow(2), $key = k(k+3)\ mod\ size$, offset function $key\ + \frac{x^2}{2} mod\ size$ και να έχουμε $resize\ threshold < 0.5$ για την μεγαλύτερη αποδοτικότητα. Concepts βασίζονται σε αυτο και αυτό .

Άλλα:

- Όταν κάνουμε print neighbours τους βάζουμε σε ένα temporary array και εκεί κάνουμε sort και μετά print στο output, έτσι δεν ρισκάρουμε με τα βασικά δεδομένα και αφού βγαίνουμε από scope array gets destroyed.
- Δεν χρειάζεται να κάνουμε close file, its handled by fstream destructor
- Βασικά όλοι οι αλγόριθμοι ήταν πολύ αργές εκτός από το hash, αλλά μετά προσέξαμε που κάνουμε build in debug mode, μετά που αλλάξαμε σε release έτρεξε πολύ γρηγορότερα.

- Σαν κομμάτι για optimisation θέλαμε να κάνουμε pre allocation of space to minimize vector resizing, περνάμε από όλα στοιχεία και βρίσκουμε το μεγαλύτερο και κάνουμε allocation, όμως αυτή η διαδικασία παίρνει πολύ χρόνο και δεν προσφέρει πολλά (ήταν πιο αργή, περίπου 20%, και όσο μεγαλύτερα δεδομένα έχουμε χειροτερεύετε η διαφορά)
- Άλλος τρόπος για optimisation που είχαμε σκεφτεί είναι να κάνουμε εισαγωγή στο AVL με δυαδική αναζήτηση για να κάνουμε όσο το δυνατόν λιγότερα rotation κατά την εισαγωγή, όμως αυτό απαιτείται ένα temporary array που να είναι ταξινομημένο, άρα λογικά δεν αξίζει και έχοντας το βασικά θα είχαμε υλοποιήσει τρόπο με Array.
- Στην αρχή φτιάξαμε άλλο πράγμα για συνεκτικές συνιστώσες (επέστρεψε ποσοστό γειτόνων μιας κορυφής πάρα το ποσοστό δένδρων στο βασικό γράφημα)
- Είχαμε recursive μέθοδο εύρεσης συνεκτικών συνιστωσών, όμως με > 1,000,000 κορυφών κράσαρε (function call stack overflow) και φτιάξαμε iterative μέθοδο με στοίβα.
- Κάναμε πολύ testing με τα graphs από εδώ και περάσαμε τα δυσκολότερα (είχαμε τεστάρει με >100,000,000 ακμές με χρόνο εκτέλεσης <10 secs)