Αριστοτέλειο Πανεπιστήμιο Θεσσαλονίκης

Project στις Δομές Δεδομένων

***Γάκης Κωνσταντίνος* (ΑΕΜ: 4877)**

***Μαλκόπουλος Δημήτρης* (ΑΕΜ: 4848)**

<u>Επιβλέπων Καθηγητής:</u>

Παπαδόπουλος Απόστολος

Ιούνιος 2025

MAIN.CPP *Μαλκόπουλος Δημήτρης*

Η **main.cpp** δουλεύει ως εξής:

Ανοίγω το file commands.txt, διαβάζω την πρώτη εντολή και την αποθηκεύω στη μεταβλητή command τύπου char [100]. Με βάση αυτή ελέγχω σε ποια κατηγορία εντολής βρισκόμαστε (π.χ. SEARCH, BUILD, DELETE, DELETEMIN, DELETEMAX...) και μπαίνω στο κατάλληλο block εντολών. Μετά από αυτό διαβάζω την δομή δεδομένων όπου θα εκτελέσουμε την συγκεκριμένη εντολή (π.χ. HASHTABLE, GRAPH, MINHEAP, MAXHEAP ή AVLTREE). Τελικά, διαβάζω τις παραμέτρους της συγκεκριμένης ολοκληρωμένης εντολής, εφόσον υπάρχουν (π.χ. BUILD GRAPH graph.txt, SEARCH HASHTABLE 100, COMPUTESPANNINGTREE GRAPH). Πριν εκτελέσω την ολοκληρωμένη εντολή ορίζω αντικείμενο τύπου chrono και μετά την εκτέλεση της εντολής, ορίζω επιπλέον αντικείμενο τύπου chrono και υπολογίζω τον χρόνο που πήρε να ολοκληρωθεί η συγκεκριμένη εντολή. Μετά από κάθε ολοκλήρωση εντολής, τυπώνω στο αρχείο τον χρόνο που χρειάστηκε (σε microseconds). Ο υπολογισμός χρόνου γίνεται σε κάθε block ολοκληρωμένης εντολής.

MINHEAP *Γάκης Κωνσταντίνος*

Για την υλοποίηση της minheap ανέπτυξα την ομώνυμη **κλάση Minheap**.

Η κλάση έχει ως δεδομένα ένα πίνακα ακεραίων (int H[1000]) μεγέθους 1000, στον οποίο υλοποιείται η δομή, και έναν ακέραιο (size), που αποθηκεύει το μέγεθος του πίνακα, άρα και το πλήθος των στοιχείων που περιέχει η minheap.

Για την κλάση ανέπτυξα τις εξής μεθόδους:

- Minheap(), που αποτελεί έναν default constructor για την κλάση
- int getSize(), που αποτελεί getter για το πλήθος των στοιχείων
- void setSize(int s), που αποτελεί setter για το πλήθος των στοιχείων, εισάγοντας την τιμή s στο δεδομένο
- int findMin(), που επιστρέφει το ελάχιστο στοιχείο. Φυσικά, εξ ορισμού αυτό θα βρίσκεται στη ρίζα της minheap, άρα η μέθοδος επιστρέφει το $1^{ο}$ στοιχείο του πίνακα
- int getElement(int pos), που επιστρέφει την τιμή του πίνακα στη σε μία συγκεκριμένη θέση, την εισαγόμενη θέση pos

- void swap(int n, int m), που αντιμεταθέτει τους κόμβους στις θέσεις n, m του πίνακα. Χρησιμοποιώντας έναν προσωρινό ακέραιο temp αντιμεταθέτει με απλό τρόπο τις τιμές των δύο κελιών του πίνακα.
- void insertMinheap(int n), που εισάγει το στοιχείο n στη δομή. Για να το πετύχει αυτό:
  - Ελέγχει αν η δομή είναι άδεια, οπότε και εισάγει το στοιχείο στη 1η θέση του πίνακα. Σε αυτήν την περίπτωση η μέθοδος τερματίζει αμέσως μετά
  - Ειδάλλως, εισάγει το στοιχείο στην τελευταία θέση του πίνακα, ως φύλλο της minheap.
  - Ύστερα, ελέγχει επαναληπτικά αν ο νέος κόμβος (c) έχει μεγαλύτερη ή ίση τιμή από τον κόμβο-γονέα του (p). Όσο δεν έχει, αντιμεταθέτει τους δύο κόμβους χρησιμοποιώντας την μέθοδο swap(c, p) και ανανεώνει τις θέσεις των c, p ανάλογα
  - Τέλος, με τον νέο κόμβο τοποθετημένο στην κατάλληλη θέση η μέθοδος τερματίζει
- void deleteMin(), που διαγράφει το ελάχιστο στοιχείο της δομής. Για να το πετύχει αυτό:
  - Θέτει το τελευταίο στοιχείο του πίνακα στην 1η θέση του και μειώνει κατά 1 το μέγεθος του πίνακα, διαγράφοντας έτσι από τη δομή τον κόμβο που βρισκόταν στη ρίζα της minheap, ο οποίος εξ ορισμού περιείχε την μικρότερη τιμή
  - Ελέγχει αν η δομή έχει ένα ή κανένα στοιχείο. Σε αυτή την περίπτωση η μέθοδος τερματίζει
  - Ειδάλλως, ξεκινά επαναλαμβανόμενα να ελέγχει αν κάποιος από τους κόμβους παιδιά έχει μικρότερη τιμή και να κάνει τις αντίστοιχες αντιμεταθέσεις
- void buildMinheap(const char *filename)*(Μαλκόπουλος Δημήτρης),* που κατασκευάζει τη minheap, διαβάζοντας στοιχεία από το αρχείο filename και εισάγοντάς τα στη minheap

MAXHEAP *Γάκης Κωνσταντίνος*

Για την υλοποίηση της maxheap ανέπτυξα την ομώνυμη **κλάση Maxheap**, ακολουθώντας παρόμοια δομή όπως στη minheap.

Η κλάση έχει ως δεδομένα ένα πίνακα ακεραίων (int H[1000]) μεγέθους 1000, στον οποίο υλοποιείται η δομή, και έναν ακέραιο (size), που αποθηκεύει το μέγεθος του πίνακα, άρα και το πλήθος των στοιχείων που περιέχει η maxheap.

Για την κλάση ανέπτυξα τις εξής μεθόδους:

- Maxheap(), που αποτελεί έναν default constructor για την κλάση
- int getSize(), που αποτελεί getter για το πλήθος των στοιχείων
- void setSize(int s), που αποτελεί setter για το πλήθος των στοιχείων, εισάγοντας την τιμή s στο δεδομένο
- int findMax(), που επιστρέφει το ελάχιστο στοιχείο. Φυσικά, εξ ορισμού αυτό θα βρίσκεται στη ρίζα της maxheap, άρα η μέθοδος επιστρέφει το 1ο στοιχείο του πίνακα
- int getElement(int pos), που επιστρέφει την τιμή του πίνακα στη σε μία συγκεκριμένη θέση, την εισαγόμενη θέση pos
- void swap(int n, int m), που αντιμεταθέτει τους κόμβους στις θέσεις n, m του πίνακα. Χρησιμοποιώντας έναν προσωρινό ακέραιο temp αντιμεταθέτει με απλό τρόπο τις τιμές των δύο κελιών του πίνακα.

- void insertMaxheap(int n), που εισάγει το στοιχείο n στη δομή. Για να το πετύχει αυτό:

    - Ελέγχει αν η δομή είναι άδεια, οπότε και εισάγει το στοιχείο στη 1η θέση του πίνακα. Σε αυτήν την περίπτωση η μέθοδος τερματίζει αμέσως μετά
    - Ειδάλλως, εισάγει το στοιχείο στην τελευταία θέση του πίνακα, ως φύλλο της maxheap.
    - Ύστερα, ελέγχει επαναληπτικά αν ο νέος κόμβος (c) έχει μικρότερη ή ίση τιμή από τον κόμβο-γονέα του (p). Όσο δεν έχει, αντιμεταθέτει τους δύο κόμβους χρησιμοποιώντας την μέθοδο swap(c, p) και ανανεώνει τις θέσεις των c, p ανάλογα
    - Τέλος, με τον νέο κόμβο τοποθετημένο στην κατάλληλη θέση η μέθοδος τερματίζει

- void deleteMax(), που διαγράφει το ελάχιστο στοιχείο της δομής. Για να το πετύχει αυτό:

    - Θέτει το τελευταίο στοιχείο του πίνακα στην 1η θέση του και μειώνει κατά 1 το μέγεθος του πίνακα, διαγράφοντας έτσι από τη δομή τον κόμβο που βρισκόταν στη ρίζα της maxheap, ο οποίος εξ ορισμού περιείχε την μεγαλύτερη τιμή
    - Ελέγχει αν η δομή έχει ένα ή κανένα στοιχείο. Σε αυτή την περίπτωση η μέθοδος τερματίζει
    - Ειδάλλως, ξεκινά επαναλαμβανόμενα να ελέγχει αν κάποιος από τους κόμβους παιδιά έχει μεγαλύτερη τιμή και να κάνει τις αντίστοιχες αντιμεταθέσεις

- void buildMaxheap(const char *filename) *(Μαλκόπουλος Δημήτρης)*, που κατασκευάζει τη maxheap, διαβάζοντας στοιχεία από το αρχείο filename και εισάγοντάς τα στη maxheap

ΔΥΑΔΙΚΟ ΔΕΝΤΡΟ ΑΝΑΖΗΤΗΣΗΣ AVL *Γάκης Κωνσταντίνος*

Για την υλοποίηση του δυαδικού δέντρου αναζήτησης AVL ανέπτυξα την κλάση NodeAVL, που υλοποιεί τους κόμβους του δέντρου, και την κλάση AVLtree που χρησιμοποιώντας τους κόμβους NodeAVL υλοποιεί το ίδιο το δέντρο.

Κλάση NodeAVL

Η κλάση έχει ως δεδομένα τους ακεραίους depth και value, που περιέχουν το 'βάθος' και την τιμή του κόμβου αντίστοιχα. Το 'βάθος' δηλώνει ουσιαστικά την απόσταση από τον κόμβο-ρίζα, οπότε π.χ. το βάθος του κόμβου-ρίζα είναι 0, ενώ το βάθος ενός από τα παιδιά του κόμβου θα έχει βάθος 1. Επιπλέον, έχει ως δεδομένα 3 pointers που δείχνουν στη διεύθυνση 3 άλλων κόμβων NodeAVL. Πιο συγκεκριμένα δείχνουν στον κόμβο-γονέα (parent) και στους κόμβους-παιδιά (αριστερό (leftCh) και δεξί(rightCh)) του κόμβου. Όλα τα pointers αρχικοποιούνται με τον κενό δείκτη (nullptr).

Για την κλάση ανέπτυξα τις εξής μεθόδους:

- NodeAVL(int v, NodeAVL *p), που αποτελεί το constructor της κλάσης, και δέχεται την τιμή και τη διεύθυνση του κόμβου "γονέα" του νέου κόμβου σαν είσοδο. Αρχικοποιεί όλα τα δεδομένα του νέου κόμβου. Ειδικά για το βάθος τοθέτει 0 αν ο κόμβος θα αποτελέσει ρίζα, ειδάλλως τον θέτει κατά 1 μεγαλύτρρο από το βάθος του γονέα του
- ~NodeAVL(), που αποτελεί destructor της κλάσης, αλλά διαγράφει και τους κόμβους "παιδιά", για διαγραφή ολόκληρου του υποδέντρου αναδρομικά
- void setDepth(int d), που αποτελεί setter του βάθους και αναδρομικά ανανεώνει το βάθος ολόκληρου του υποδέντρου. Για να το πετύχει αυτό ελέγχει αν ο κόμβος έχει αριστερό ή δεξί κόμβο-παιδί και καλεί τη μέθοδο στα αντίστοιχα παιδια, με την είσοδο d αυξημένη κατά 1
- int getDepth(), που αποτελεί getter του "βάθους"
- void setValue(int v),που αποτελεί setter της τιμής
- int getValue(),που αποτελεί getter της τιμής
- void setParent(NodeAVL *n),που αποτελεί setter του κόμβου "γονέα"
- NodeAVL *getParent(),που αποτελεί getter του κόμβου "γονέα"
- void setLeftChild(NodeAVL *n),που αποτελεί setter του αριστερού κόμβου "παιδί"
- NodeAVL *getLeftChild(),που αποτελεί getter του αριστερού κόμβου "παιδί"
- void setRightChild(NodeAVL *n),που αποτελεί setter του δεξιού κόμβου "παιδί"
- NodeAVL *getRightChild(),που αποτελεί getter του δεξιού κόμβου "παιδί"

Κλάση AVLtree

Η κλάση έχει ως δεδομένα τους ακεραίους treedepth και treesize, που περιέχουν το βάθος (ύψος) και το μέγεθος (άρα και το πλήθος στοιχείων) του δέντρου αντίστοιχα. Επιπλέον, έχει ως δεδομένο έναν pointer που δείχνει στη διεύθυνση του κόμβου-ρίζα NodeAVL του δέντρου(root) και αρχικοποιείται με τον κενό δείκτη (nullptr).

Για την κλάση ανέπτυξα τις εξής μεθόδους:

- AVLtree(), που αποτελεί έναν default constructor για την κλάση
- ~AVLtree(), που αποτελεί destructor της κλάσης. Διαγράφει τη ρίζα και λόγω του destructor της κλάσης NodeAVL αναδρομικά όλους τους κόμβους του δέντρου
- int findmin(), που επιστρέφει την ελάχιστη τιμή του δέντρου. Για να το πετύχει αυτό επαναληπτικά διατρέχει την αριστερή πλευρά του δέντρου μεχρι να φτάσει σε κόμβο που δεν έχει αριστερό παιδί. Αυτός θα είναι ο ζητούμενος κόμβος
- int getTreesize(), που αποτελεί getter για το βάθος (ύψος) του δέντρου
- void setTreesize(int s), που αποτελεί setter για το βάθος (ύψος) του δέντρου, θέτοντας ως νέα τιμή την s
- NodeAVL *search(int x), που αναζητά την τιμή x στο δέντρο. Για να το πετύχει αυτό διατρέχει το αριστερό παιδί του εκάστοτε κόμβου, αν η τιμή του είναι μεγαλύτερη της x, η το δεξί παιδί του κόμβου, αν η τιμή του είναι μικρότερη της x. Τερματίζει όταν βρει την τιμή x ή όταν τελειώσουν οι διαθέσιμοι κόμβοι, και επιστρέφει τη διεύθυνση του κόμβου ή τον κενό δείκτη αντίστοιχα
- void rotate(NodeAVL *a, NodeAVL *b), που εκτελεί την απλή περιστροφή στους κόμβους του δέντρου, για τη σταδιακή μετατροπή ενός μη AVL δέντρου σε AVL. Ο κόμβος a πρέπει να βρίσκεται δεξιότερα (να έχει μικρότερη τιμή) από τον b, ανεξάρτητα από το ποιός είναι ο γονιός ποιανού, για να λειτουργήσει σωστά η μέθοδος. Για να πετύχει την απλή περιστροφή:
  - Αρχικά βρίσκει το βάθος των a, b (da, db) και τα συγκρίνει, αφού ο κόμβος με το μικρότερο βάθος θα είναι ο κόμβος-γονέας του άλλου
  - Στην περίπτωση που (da<db), ο a είναι γονέας του b και άρα θα εκτελέσει απλή αριστερή περιστροφή
  - Σε αυτή την περίπτωση θέτει c το αριστερό παιδί του b και temp3 τον γονέα του a. Έτσι, θέτει temp3 τον γονέα του b, θέτει b τον γονέα του a και αριστερό παιδί του b τον a, και δεξί παιδί του a τον c. Αν ο κόμβος temp3 υπάρχει θέτει αντί για παιδί του τον a τώρα τον b, ενώ αν υπάρχει ο c θέτει σαν γονέα του τον a. Τέλος, αν ο a ήταν ρίζα του δέντρου, θέτει τον b ως νέα ρίζα του δέντρου και η μέθοδος τερματίζει
  - Αν τελικά βρισκόμαστε στην περίπτωση που (da>db), ο b είναι γονέας του a και άρα θα εκτελέσει απλή δεξιά περιστροφή με την ακριβώς ανάστροφη λογική
- void doubleRotate(NodeAVL *a, NodeAVL *z, NodeAVL *b), που εκτελεί διπλή περιστροφή στους κόμβους του δέντρου, για τη σταδιακή μετατροπή ενός μη AVL

δέντρου σε AVL. Ο κόμβος a πρέπει να βρίσκεται δεξιότερα (έχει μικρότερη τιμή) από τον z, ο οποίος πρέπει να βρίσκεται δεξιότερα από τον b, ανεξάρτητα από το ποιός είναι ο γονιός ποιανού, για να λειτουργήσει σωστά η μέθοδος. Για να πετύχει την διπλή περιστροφή:

- Αρχικά θέτει c1, c2 το αριστερό και δεξί παιδί του z αντίστοιχα, ενώ βρίσκει και το βάθος των a, b (da, db) και τα συγκρίνει, αφού ο κόμβος με το μικρότερο βάθος θα είναι ο κόμβος-γονέας του άλλου.
- Στην περίπτωση που (da<db), ο a είναι γονέας του b και άρα θα εκτελέσει διπλή αριστερή περιστροφή. Θέτει γονέα του z τον γονέα του a, και θέτει στον γονέα του a (αν υπάρχει) ως παιδί αντί για τον a τον z
- Σε την περίπτωση που (da>db), ο b είναι γονέας του a και άρα θα εκτελέσει διπλή δεξιά περιστροφή. Θέτει γονέα του z τον γονέα του b, και θέτει στον γονέα του b (αν υπάρχει) ως παιδί αντί για τον b τον z
- Ύστερα, θέτει στον z αριστερό παιδί τον a και δεξί παιδί τον b (αντίστοιχα γονέα των a,b τον z), θέτει δεξί παιδί του a τον c1, αριστερό παιδί του b τον c2 (αντίστοιχα, αν υπάρχει ο c1 θέτει γονέα του τον a, και αν υπάρχει ο c2 θέτει γονέα του τον b)
- Τέλος, χρησιμοποιώντας τη μέθοδο setDepth(int d) της NodeAVL στον z, ανανεώνει την τιμή depth όλων των κόμβων του υποδέντρου. Επίσης, στην περίπτωση που ο κόμβος a/b (ανάλογα την περίπτωση) ήταν ρίζα του δέντρου, θέτει τον z ως νέα ρίζα του δέντρου. Η μέθοδος ύστερα τερματίζει

- int findSubtreeDepth(NodeAVL *a), που βρίσκει το βάθος (ύψος) του υποδέντρου με ρίζα a. Για να το πετύχει αυτό:
  - Ελέγχει αρχικά αν το a υπάρχει. Αν δεν υπάρχει (ο δείκτης δείχνει στον κενό δείκτη), τότε θέτει το βάθος d=-1, και επιστρέφει ++d, άρα επιστρέφει 0 και η μέθοδος τερματίζει
  - Ειδάλλως, βρίσκει το βάθος (ύψος) του αριστερού υποδέντρου (sd1) και του δεξιού υποδέντρου (sd2), καλώντας τον εαυτό της στα παιδιά του κόμβου a. Τέλος, θέτει στο βάθος d τη μεγαλύτερη τιμή από τα sd1, sd2 και επιστρέφει ++d, τερματίζοντας τη μέθοδο
- void insertNode(int v), που εκτελεί εισαγωγή νέου κόμβου με τιμή v. Για να το πετύχει αυτό:
  - Αρχικά ελέγχει αν το δέντρο είναι άδειο, οπότε και τοποθετεί το νέο κόμβο στη ρίζα του δέντρου, θέτει το βάθος του κόμβου 0 και υπολογίζει το βάθος (ύψος) του δέντρου, που θα είναι 1. Η μέθοδος τερματίζει
  - Ειδάλλως, διαπερνά επαναληπτικά τους κόμβους του δέντρου μέχρι να βρει την κατάλληλη θέση για το νέο κόμβο ως φύλλο του δέντρου. Το δέντρο παραμένει έτσι δυαδικό αναζήτησης. Δημιουργεί το νέο κόμβο και τον συνδέει με τον κόμβο-γονέα του.
  - Ύστερα, ξεκινώντας από τον γονέα του γονέα του νέου κόμβου, ελέγχει επαναληπτικά αν το υποδέντρο παραμένει AVL, μέχρι να ελέγξει και τη ρίζα του δέντρου. Για να ελέγξει αν ένα υποδέντρο είναι AVL, αποθηκεύει στα dleft, dright το βάθος (ύψος) του δεξιού και αριστερού υποδέντρου της ρίζας του υποδέντρου, με τη χρήση της μεθόδου findSubtreeDepth(NodeAVL *a). Αν

διαπιστώσει ότι τα dleft, dright έχουν απόκλιση μεγαλύτερη του 1, εκτελεί απλή ή διπλή περιστροφή, ανάλογα με την περίσταση, και ύστερα ελέγχει τον κόμβο-γονέα του κόμβου που μόλις έλεγξε
    - Τέλος, ανανεώνει αναδρομικά τις τιμές "depth", των κόμβων, που τυχόν να μεταβλήθηκαν κατά τις περιστροφές, με την μέθοδο setDepth(int d) της NodeAVL. Επίσης, υπολογίζει και ανανεώνει την τιμή του treedepth, εφαρμόζοντας τη μέθοδο findSubtreeDepth(NodeAVL *a) στη ρίζα του δέντρου
- void deleteNode(int v), που εκτελεί διαγραφή του κόμβου με τιμή v. Για να το πετύχει αυτό:
    - Αρχικά, αναζητά τη διεύθυνση του κόμβου με τιμή v, χρησιμοποιώντας τη μέθοδο search(int v), και την τοποθετεί στο δείκτη n
    - Αφού επιβεβαιώσει ότι ο n κόμβος βρέθηκε, κάνει επαναληπτικά απλές περιστροφές, κατεβάζοντας έτσι τη θέση του κόμβου προς τα φύλλα του δέντρου, μέχρι ο n να μην έχει 2 κόμβους-παιδιά
    - Ύστερα συνδέει το νέο κόμβο-γονέα του n με το νέο κόμβο-παιδί του n, αν αυτός υπάρχει. Ελέγχει επιπλέον τις περιπτώσεις που ο νέος κόμβος-παιδί γίνεται ρίζα του δέντρου, ή που το δέντρο άδειασε τελείως από στοιχεία
    - Στη συνέχεια, επειδή το δέντρο πιθανόν να μην είναι πια AVL, ελέγχει επαναληπτικά αν το υποδέντρο παραμένει AVL, μέχρι να ελέγξει και τη ρίζα του δέντρου, ακριβώς με τον ίδιο τρόπο όπως στην insertNode(int v)
    - Τέλος, ανανεώνει αναδρομικά τις τιμές "depth", των κόμβων, που τυχόν να μεταβλήθηκαν κατά τις περιστροφές, με την μέθοδο setDepth(int d) της NodeAVL. Επίσης, υπολογίζει και ανανεώνει την τιμή του treedepth, εφαρμόζοντας τη μέθοδο findSubtreeDepth(NodeAVL *a) στη ρίζα του δέντρου.
    - Σε αυτό το σημείο διαγράφει τον κόμβο n οριστικά από τη μνήμη, αφού θέσει σε κενό δείκτη όλους τους δείκτες του, για να αποφευχθεί αναδρομικά η διαγραφή τους (εξαιτίας του destructor της NodeAVL)
- void buildAVLTree(const char *filename) *(Μαλκόπουλος Δημήτρης)*, που κατασκευάζει τη δομή, διαβάζοντας στοιχεία από το αρχείο filename και εισάγοντάς τα στο δέντρο
- void tempprint(NodeAVL *a), που αποτελεί προσωρινή μέθοδο εκτύπωσης του δέντρου, για τη διεξαγωγή ελέγχων

ΓΡΑΦΟΣ *(Μη κατευθυνόμενος με βάρη στις ακμές)* ***Μαλκόπουλος Δημήτρης***

Η κλάση Graph περιέχει 2 ακόμα κλάσεις που έπρεπε να κατασκευάσω για να μπορώ να υλοποιήσω τον γράφο. Περιέχει μια λίστα με αντικείμενα Node, τα οποία περιέχουνε με τη σειρά τους Edges.

Κλάση Node

Κάθε κόμβος (Node) υποστηρίζει δικές του μεθόδους όπως:

- void insertEdge(Edge *edge): εισάγει στον πίνακα με pointers το edge που θα έχει ο κόμβος
- void deleteEdge(Edge *edge): διαγράφει ένα συγκεκριμένο edge από τον πίνακα με pointers σε Edge κάνοντας ολίσθηση στα αριστερά από το σημείο που θα βρει την συγκεκριμένη ακμή
- void setData(int data): Data είναι ένας ακέραιος που αποθηκεύει μια προσωρινή τιμή και τον χρειάζονται οι επόμενες μέθοδοι για ευκολότερη υλοποίησή τους
- void setColour(int colour): αναθέτει τιμή στο μέλος colour του κόμβου, το οποίο χρησιμοποιείται στο compute shortest path
- Edge **getNeighbours(int &neighbours): επιστρέφει τον πίνακα με τα edges του κόμβου και αναθέτει τον αριθμό των ακμών του κόμβου στο neighbours
- int getData(): getter για τον ακέραιο data
- int getNumEdge(): επιστρέφει το πλήθος των edges
- int getID(): getter για το ID
- int getColour(): getter για το colour

Κλάση Edge

Κάθε αντικείμενο edge (ακμή) υποστηρίζει τις μεθόδους:

- void setEndpoint(Node *endpoint): ορίζει τον κόμβο που συνδέει η ακμή (η κλάση έχει ένα δείκτη σε κόμβο start και endpoint, που είναι πολύ χρήσιμος σε μεθόδους όπως οι compute minimum spanning tree και find shortest path)
- void setStart(Node *start): ορίζει τον κόμβο από τον οποίο αρχίζει η ακμή
- void setWeight(int weight): ορίζει το weight της ακμής
- Node *getEndpoint(): επιστρέφει τον κόμβο endpoint
- Node *getStart(): επιστρέφει τον κόμβο start
- int getWeight(): επιστρέφει το βάρος της ακμής

Κλάση Graph

Δεδομένα:

- int numNodes
- int numEdges
- Node** nodes= new Node*[MAX_NODES]

Μέθοδοι:

- void build_graph(const char *filename): ανοίγει το αρχείο με όνομα filename και κατασκευάζει το γράφο
- insert_edge(int id1, int id2, int weight): παίρνει 2 αριθμούς από κόμβους και το βάρος της ακμής που θα τα ενώνει. Μέσα σε αυτήν δημιουργεί 2 νέους δείκτες σε αντικείμενα τύπου node και ψάχνω αν υπάρχουν στον πίνακα με όλους τους κόμβους του γράφου. Αν δεν υπάρχει τουλάχιστον ένας από τον καθένα τους, εισάγω τον/τους κατάλληλους στον πίνακα του γράφου. Αυτή τη διαδικασία την κάνει για κάθε τριάδα αριθμών στο αρχείο filename
- void getSize(int *nod, int *edg): παίρνει παραμέτρους 2 &int nod, edg με αναφορά για να μπορούν να επιστραφούν στην main. Αναθέτει το μέλος numNodes στο nod και το numEdges στο edg
- int computeshortestpath(int src, int dest): Η συγκεκριμένη μέθοδος υλοποιεί τον αλγόριθμο του Dijkstra μέχρι να υπολογιστεί ο μικρότερος σε κόστος δρόμος από τον κόμβο number1 στον κόμβο number2. Αυτό γίνεται όταν η απόσταση του κόμβου number2 που έχει υπολογιστεί μέχρι εκείνη την ώρα είναι στα unvisited nodes (γι'αυτό χρησιμοποίησα το μέλος colour του node) και είναι το ελάχιστο από τα υπόλοιπα distances των unvisited nodes. Αρχικά πρέπει να υπάρχουνε και οι 2 κόμβοι στο γράφο γι'αυτο κάνω μια αναζήτηση στο nodes[] πίνακα του γράφου και οταν βρω τον κόμβο number2 τον αναθέτω σε ενα δείκτη destination_node για να γνωρίζω πότε να σταματήσω τον αλγόριθμο. Σε όλους τους κόμβους αναθέτω την τιμή INT_MAX (απο cmath) που αντιπροσωπεύει την τιμή του απείρου, εκτός από τον κόμβο number1 που βάζει 0, αναθέτω τιμή με την μέθοδο Node.setData(). Στη συνέχεια δημιουργώ ένα δυναμικό πίνακα Nodelist ** με όλους τους κόμβους του γράφου για τον οποίο έχω υλοποιήσει συνάρτηση resize_array<class X> (template επειδή χρειάζομαι edgelist στο compute spanning tree). Αρχίζω από τον κόμβο number1 (χρησιμοποιώ το μέλος ID του κόμβου για αυτό), πάω σε όλους τους γείτονές του, τους βάζω σαν data το μήκος της ακμής τους που ενώνεται με τον κόμβο number2 και διαγράφω τον number2 απο το nodelist. Μετά επαναληπτικά βρίσκω τον κόμβο από το nodelist με το μικρότερο data (δηλαδή μικρότερο estimated path cost) και σε αυτό το κόμβο min ενημερώνω τους γείτονές του. Η ενημέρωση γίνεται με τη μέθοδο setData(). Αν το data του γείτονα είναι μεγαλύτερο από το data του κόμβου min + το βάρος της ακμής που τα συνδέει τότε το data θα

γίνει data του κόμβου min + το βάρος της ακμής. Διαγράφω τον κόμβο min από το nodelist, δηλώνω ότι βρήκα το minimum cost path για το συγκεκριμένο κόμβο κάνοντας min->setColour(1) και βρίσκω το επόμενο min. Η επανάληψη τερματίζει όταν το destination_node->getcolour()==1

- int computespanningtree(): Η λογική είναι ίδια με την λογική του compute minimum path. Έχω μια τιμή int sum που αρχικοποιείται με 0 και αντιπροσωπεύει το κόστος του spanning tree. Δημιουργώ ένα Edge **edgelist, αρχίζω από το πρώτο node του γράφου (nodes[0]) και αναθέτω το data του nodes[0]=0. Βάζοντας όλες τις ακμές του στο edgelist βρίσκω την ελάχιστη ακμή, αυτή δηλαδή με το μικρότερο βάρος. Μετά επαναληπτικά βάζω το βάρος της ελάχιστης ακμής στο sum, παίρνω τον κόμβο που συνδέει η ακμή μέσω του min->getEndpoint(), αφού η αρχή του είναι το min. Από αυτό το νέο κόμβο αναθέτω το data του ίσο με 1 για να δηλώσω ότι τον έχω επισκεφτεί βάζω όλες τις ακμές του στο edgelist εφόσον το endpoint της ακμής δεν έχει data==1 . Αν βρει έστω και ένα κόμβο από το edgelist του οποίου το data είναι 0 τότε θα συνεχίσει η επαναληπτική διαδικασία. Αυτό υλοποιείται με μια μεταβλητή τύπου bool not_done
- int findconnectedcomponents(): Για αυτή τη μέθοδο υλοποίησα μια ακόμα μέθοδο dfs() η οποία κάνει dfs στο γράφο. Ορίζω μια μεταβλητή int components=0. Για να βρω τα components του γράφου κάνω προσπέλαση από την αρχή στον πίνακα nodes[] και αν βρω ότι το data του nodes[i] είναι 0 κάνω dfs στο συγκεκριμένο κόμβο, το οποίο θα κάνει τα data όλων των κόμβων που θα ανακαλύψει ίσο με 1. Οπότε στην επανάληψη όσο βρίσκει nodes[i].getData()==0 σημαίνει ότι ο κόμβος είναι unvisited και κάνω components++, αν το dfs δεν βρει όλα τα υπόλοιπα nodes θα το καταλάβουμε όσο προσπελάζουμε τον πίνακα nodes[]. Μετά την επανάληψη θα έχουμε κάνει dfs όσες φορές ίσο με τα components του γράφου και έτσι επιστρέφω την τιμή του components
- DELETE GRAPH number1 number2: Προσπελάζω τον πίνακα nodes[] μέχρι να βρω είτε το number1 είτε το number2 node και με την βοήθεια μιας συνάρτησης connected() που υλοποίησα βρίσκει αν συνδέονται οι 2 κόμβοι. Αν συνδέονται, τότε κάνω number1 - >deleteedge(connecting_edge), number2 - >deleteedge(connecting_edge) και διαγράφω το connecting_edge

Σχετικά με τις μεθόδους COMPUTE MINIMUM PATH number1,number2 και COMPUTE SPANNING TREE:

Πήρα την κλάση minheap του συνεργάτη μου και απο απλή minheap που αποθηκεύει integers, την έκανα template έτσι ώστε να δέχεται edges και nodes. Γι'αυτό δημιουργησα στη main τα struct edge_compare, struct node_compare, για να μπορώ να συγκρίνω 2 ακμές με βάση τα βάρη τους και τα nodes με βάση τα data τους. Δεν χρησιμοποίησα την template minheap λόγω ελάχιστου χρόνου, η ίδια η κλάση όμως δουλεύει σωστά. Έχω αφήσει το .h αρχείο της με ονομα template_minheap.h, μήπως θέλατε να την δείτε.

HASHTABLE *Μαλκόπουλος Δημήτρης*

Κλάση Hashtable

Η συγκεκριμένη κλάση υλοποιεί έναν πίνακα κατακερματισμού. Διαθέτει τα δεδομένα int size, int capacity και int **table όπου θα καταλαβαίνω αν μια θέση του είναι κενή(όταν ισούται με nullptr). Περιέχει τις μεθόδους:

- void insert(int value): για να εισάγει την τιμή value πάει στη θέση unsigned long current_key αριθμο* 2654435761 % capacity και βλέπει αν είναι άδεια (δηλαδή nullptr). Αν δεν είναι, τότε επαναληπτικά βρίσκει νέο κλειδί με τον τύπο current_key = (current_key+1)* 2654435761%capacity μέχρι να βρει κενή θέση
- bool search(int number): πάει στη θέση key = number* 2654435761%capacity, την οποια αποθηκεύω σε μια ακόμα μεταβλητή starting_key, και όσο το αντικείμενο εκεί δεν είναι ίσο με number ενημερώνει το key με (key+1)* 2654435761%capacity μέχρι να γίνει key==starting_key. Αυτό γιατί αν υπάρχει το number, και η θέση που θα έπρεπε να μπει αρχικά ήταν γεμάτη, η διαδικασία key=(key+1)* 2654435761%capacity θα πάρει την ίδια ακολουθία αριθμών και αν βρεθούμε στο αρχικο key χωρίς να έχουμε βρει το number σημαίνει ότι δεν υπάρχει. Αν στην επανάληψη βρει τον αριθμό η συνάρτηση επιστρέφει true, αν όχι επιστρέφει false και με βάση αυτά γράφω SUCCESS ή FAILURE στο output.txt
- int getSize(): επιστρέφει το size
- void build_hashtable(const char *filename): εισάγω κάθε αριθμό μέσα στο αρχείο «filename» στο hashtable μέσω της μεθόδου insert(number)

ΕΠΙΜΕΛΕΙΑ ΤΕΧΝΙΚΗΣ ΕΚΘΕΣΗΣ *Γάκης Κωνσταντίνος*