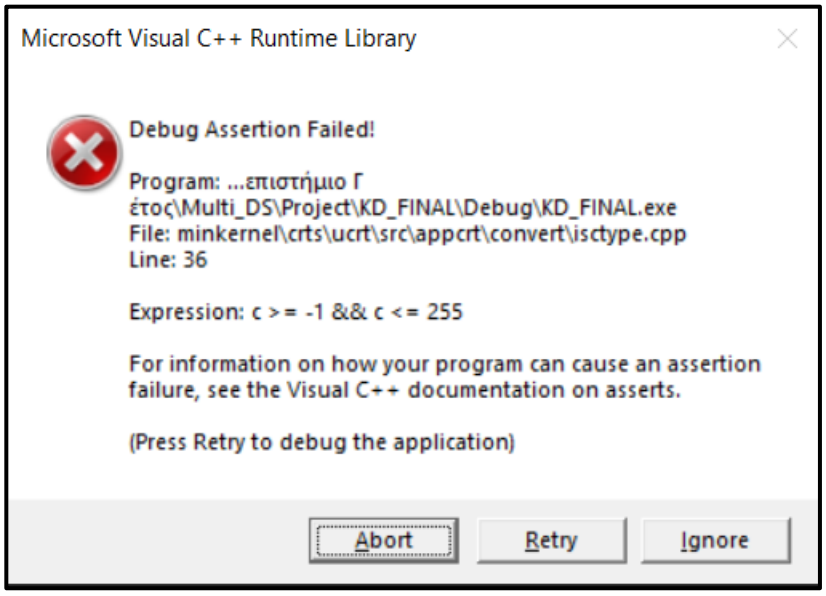
ΕΙΣΑΓΩΓΗ

Πριν ξεκινήσουμε την ανάλυση των δομών που υλοποιήθηκαν, κρίνεται χρήσιμη η παράθεση κάποιων αρχικών πληροφοριών και παραδοχών:

1) Κάθε αρχείο αναπαρίσταται από 1 σημείο στον 3-d χώρο με βάση των τίτλο ο οποίος είναι της μορφής *«ΜΑΘΗΜΑ ΕΠΩΝΥΜΟ_ΟΝΟΜΑ ΑΜ»*. Η $1^{η}$ λέξη αντιστοιχεί στην συντεταγμένη x του σημείου, η $2^{η}$ στην y και η $3^{η}$ στην z συντεταγμένη.
2) Με βάση την παραδοχή 1, τα ερωτήματα περιοχής που μπορεί να κάνει ο χρήστης στις δομές μας, δημιουργούν περιοχή σχετιζόμενη με εύρος μαθημάτων, εύρος ονοματεπώνυμων φοιτητών και εύρος ΑΜ φοιτητών.
3) Τα κριτήριο που διατηρήσαμε για την κατασκευή των R-trees, είναι αυτό της ελάχιστης ολική κάλυψης και ελάχιστου κενού χώρου εντός ενός MBR. Πιο αναλυτικά, ελάχιστη ολική κάλυψη επιτυγχάνεται όταν α) τα MBRs είναι όσο πιο μικρά γίνεται – για το σκοπό αυτό ομαδοποιούμε κοντινά σημεία με αποτέλεσμα τα MBRs να περιέχουν και όσο το δυνατόν λιγότερο κενό χώρο και β) τα MBRs καλύπτουν συνολικά, όσο το δυνατόν μικρότερη περιοχή γίνεται – αυτό επιτυγχάνεται εν μέρει εξαιτίας του α, αλλά και με την καλύτερη δυνατή ομαδοποίηση των MBRs

ΓΝΩΣΤΑ BUGS - ΣΗΜΕΙΩΣΕΙΣ

1. Τα R-trees ύψους ≥ 2 (επίπεδο 0 τα φύλλα), κατασκευάζονται με λάθος τρόπο! Αυτό οφείλεται στην αναδρομική κλήση της Access, που κατά την επιστροφή, χάνει την πληροφορία του επιπέδου των φύλλων, όπου πρέπει να γίνει η εισαγωγή. Το λάθος στην κατασκευή επηρεάζει και την αναζήτηση, που δίνει λάθος αποτέλεσμα. Για αποφυγή αυτού του λάθους και έλεγχο ορθής λειτουργίας του δέντρου για λίγα αρχεία και ύψος <2, συνίσταται η μείωση του dataset στα 12 πρώτα αρχεία. Η διαπιστωθεί αυτού του λάθους έγινε πολύ αργά, οπότε το λάθος δεν διορθώθηκε λόγω έλλειψης χρόνου.
2. Σε κάποια αρχεία του dataset υπάρχουν χαρακτήρες που δεν αναγνωρίζονται (τουλάχιστον στον υπολογιστή ενός μέλους, χτυπούσαν errors, αλλά όχι σε όλους). Αν, ύστερα από την είσοδο της περιοχής αναζήτησης στα αρχεία “R-tree.cpp” και “kd-tree.cpp”, εμφανιστεί error της μορφής:



Σημαίνει ότι στην απάντηση περιέχεται αρχείο με τέτοιο «ακατάλληλο» χαρακτήρα. Συνίσταται η χρήση διαφορετικής εισόδου για περιοχή αναζήτησης.

3. **ΠΡΟΣΟΧΗ!** Ανάλογα με την τοποθεσία αποθήκευσης του Project, πρέπει τα μονοπάτια testfolder να αλλάζουν για κάθε αρχείο. Τουλάχιστον για τα R-trees, δεν συνίσταται να παραμείνει ο φάκελος Rtestfiles μέσα στο φάκελο του πρότζεκτ. Δοκιμή με το φάκελο εντός οδήγησε το πρόγραμμα σε βίαιο τερματισμό (abort()), ενώ όταν ο φάκελος τοποθετήθηκε στο Desktop, το πρόγραμμα δούλεψε κανονικά με τις εισόδους που δίνονται παρακάτω. Για τα kd-trees, δεν διαπιστώθηκε κάποιο πρόβλημα, είτε ο φάκελος testfiles μείνει εντός, είτε τοποθετηθεί εκτός του φακέλου του project.

ΠΡΟΤΕΙΝΟΜΕΝΕΣ ΕΙΣΟΔΟΙ ΓΙΑ ΕΛΕΓΧΟ

R-trees:

{STAT, TPY, BLAH_BLAH, CHATTY, 1092841, 1039999} (η λάθος σειρά στις συντεταγμένες z θα διορθωθεί αυτόματα από τη check_valid_reg) → Δεν υπάρχουν αρχεία σε αυτή την περιοχή

{BE, PARKOUR, PAGONI, SFIKAS_THODORIS, 1072494, 1096454} → kd-trees:

{STAT, TPY, BLAH_BLAH, CHATTY, 1092841, 1039999} (η λάθος σειρά στις συντεταγμένες z θα διορθωθεί αυτόματα από τη check_valid_reg) → Δεν υπάρχουν αρχεία σε αυτή την περιοχή

{BE, PARKOUR, PAGONI, SFIKAS_THODORIS, 1072494, 1096454} →

Kd-trees

Δομή: υλοποιήθηκαν κομβοπροσανατολισμένα kd-trees.

Κόμβος: κάθε κόμβος περιέχει πληροφορία για ένα αρχείο, δηλαδή τις συντεταγμένες του, τη διαδρομή που οδηγεί σε αυτό (path) καθώς και το βάθος του. Απαραίτητοι για την κατασκευή του δέντρου είναι και οι δείκτες γονιού (parent), αριστερού και δεξιού παιδιού (left_child και right_child αντίστοιχα) που φυλάσσονται επίσης εντός της δομής του κόμβου. Κάθε κόμβος χρησιμοποιεί μόνο μία συντεταγμένη για εισαγωγή και αναζήτηση, ξεκινώντας από τη ρίζα με X και συνεχίζοντας στους επόμενους κόμβους με Y, Z, X, Y, Z…

Για κάθε κόμβο παρέχονται οι παρακάτω **λειτουργίες:**

α) αποθήκευση των συντεταγμένων του → `void getcoords(const std::string filepath)`: το όρισμα είναι η διαδρομή που οδηγεί στην τοποθεσία αποθήκευσης του αρχείου. Η συνάρτηση διαβάζει την πρώτη γραμμή («τίτλος») του αρχείου, εξάγει τις συντεταγμένες σύμφωνα με την παραδοχή 1 της εισαγωγής και τις αποθηκεύει.

β) έλεγχος αν ένα σημείο βρίσκεται πάνω ή μεταξύ δύο επιπέδων → `bool is_between(std::string start, std::string end, int coord)`: Το 1ο όρισμα είναι το αριστερό επίπεδο, το 2ο είναι το δεξιό επίπεδο και το 3ο είναι ο δείκτης της συντεταγμένης με βάση την οποία κάνουμε αναζήτηση στο επίπεδο του κόμβου.

γ) δημιουργία περιοχής για ένα κόμβο → `std::vector<std::string> make_region(int level, std::vector<std::string> qcoords)`: Το 1ο όρισμα είναι το επίπεδο του κόμβου και το 2ο είναι το διάνυσμα των συντεταγμένων της περιοχής αναζήτησης. Η συνάρτηση επιστρέφει ένα vector από συντεταγμένες [x1, x2, y1, y2, z1, z2], τις συντεταγμένες που ορίζουν την περιοχή του σημείου. Η λειτουργία της είναι αρκετά περίπλοκη, λόγω των πολλών περιπτώσεων που προκύπτουν από το επίπεδο του κόμβου και για να την εξηγήσουμε, χρειάζεται να έχουμε αναλύσει πρώτα τις λειτουργίες του δέντρου, γι’ αυτό θα την παραθέσουμε στο τέλος της ενότητας.

KD_Tree: αποτελεί κλάση. Διατηρεί πεδία για τη ρίζα, το ολικό βάθος του δέντρου και το πλήθος των κόμβων που περιέχει. Οι **λειτουργίες** που υλοποιήθηκαν είναι οι παρακάτω:

α) επιστροφή της ρίζας → `kd_node* getroot()`: επιστρέφει δείκτη στη ρίζα του δέντρου

β) εύρεση θέσης εισαγωγής νέου κόμβου → `kd_node *Access(kd_node &curr, kd_node &to_ins, int &depth, bool &pos)`: το 1ο όρισμα είναι ο τρέχον κόμβος που εξετάζουμε, το 2ο είναι ο κόμβος εισαγωγής, το 3ο είναι το βάθος του τρέχοντα κόμβου και το τελευταίο είναι ένα flag που δηλώνει αν ο κόμβος πρέπει να εισαχθεί ως δεξιό ή αριστερό παιδί του τρέχοντος, παίρνοντας τιμή true μόνο στη δεύτερη περίπτωση. Σε κάθε επίπεδο, χρησιμοποιείται

διαφορετική συντεταγμένη για να ληφθεί η απόφαση σχετικά με τη συνέχεια προς το δεξί ή αριστερό παιδί σύμφωνα με τους εξής κανόνες:
- αν depth mod 3 = 0, χρησιμοποιείται η συντεταγμένη X
- αν depth mod 3 = 1, χρησιμοποιείται η συντεταγμένη Y
- αν depth mod 3 = 2, χρησιμοποιείται η συντεταγμένη Z

Οι κανόνες αυτοί υποστηρίζονται από την εξής παρατήρηση: στο επίπεδο 0 (ρίζα) χρησιμοποιείται η X, στο επίπεδο 1 η Y, στο επίπεδο 2 η Z και μετά οι συντεταγμένες επαναλαμβάνονται με αυτή τη σειρά. Άρα, στα επίπεδα 0, 3, 6, 9 κοκ, δηλαδή στα πολλαπλάσια του 3 χρησιμοποιείται η X, στα επίπεδα 1, 4, 7, 10 κοκ η Y και στα επίπεδα 2, 5, 8, 11 κοκ η Z.

Έστω τώρα κόμβος n στο επίπεδο i, κάτω από τον οποίο πρέπει να εισαχθεί ο κόμβος εισαγωγής m, και $coord_i$ είναι η συντεταγμένη που χρησιμοποιείται στο επίπεδο i.

- Αν $coord_i[m] \leq coord_i[n]$, τότε ο m εισάγεται στα αριστερά του n.
- Αν $coord_i[m] > coord_i[n]$, τότε ο m εισάγεται στα δεξιά του n.

Το ίδιο ισχύει και αν ο m είναι απλά παιδί του n και ψάχνουμε που να εισάγουμε κάποιο κόμβο k, με τη μόνη διαφορά ότι ο m δεν εισάγεται, απλά αποτελεί το αντίστοιχο παιδί του n.

Αρχικά**, η συνάρτηση υπολογίζει την συντεταγμένη που χρησιμοποιείται** στο επίπεδο για το οποίο καλείται (με βάση του 3 πρώτους κανόνες)
Στη συνέχεια, **ελέγχει προς τα πού πρέπει να συνεχίσει** (με βάση τους 2 τελευταίους κανόνες)
Αν εκεί υπάρχει κενή θέση ενεργοποιεί το **flag** αν χρειάζεται και **επιστρέφει τον τρέχοντα κόμβο**.
Διαφορετικά, η συνάρτηση **καλείται αναδρομικά για το παιδί του τρέχοντα κόμβου, στην κατεύθυνση στην οποία αποφάσισε ότι πρέπει να συνεχίσει, αλλάζοντας ανάλογα το όρισμα για το βάθος**.
β) εισαγωγή νέου κόμβου → `void insert(std::string filepath)`: το όρισμα είναι η διαδρομή που οδηγεί στην τοποθεσία που είναι αποθηκευμένο το αρχείο. Πρώτα αρχικοποιούνται το flag εισαγωγής στα αριστερά με false και το βάθος με 0 επειδή από το επίπεδο 0 θα ξεκινήσει η αναζήτηση σημείου εισαγωγής στο δέντρο. Με τη δημιουργία καινούριου κόμβου, με συντεταγμένες και τοποθεσία, ολοκληρώνεται το στάδιο των αρχικοποιήσεων.
Αν το δέντρο δεν περιέχει κόμβους (count = 0), **τότε η ρίζα του δέντρου εξισώνεται με τον κόμβο που δημιουργήθηκε** και **το πλήθος κόμβων αυξάνεται κατά 1**.
Διαφορετικά, καλείται η **Access** για εύρεση του κόμβου κάτω από τον οποίο πρέπει να γίνει η εισαγωγή (insertion_node).
Αν μετά την επιστροφή του αποτελέσματος, το flag left είναι true, **η εισαγωγή γίνεται στα αριστερά** του insertion_node, *αλλιώς* **στα δεξιά**. Σε κάθε μία από τις 2 περιπτώσεις (left = true ή false), **το βάθος του δέντρου αυξάνεται κατά 1, όπως και το πλήθος των κόμβων**.
ΣΗΜΑΝΤΙΝΚΗ ΣΗΜΕΙΩΣΗ! Η υλοποίηση μας, πρόκειται για μία παραλλαγή του kd-tree. Όπως φαίνεται, η κατασκευή του δέντρου μας δεν απαιτεί ποτέ να βρούμε τον μέσο από το σύνολο

των αρχείων μας, άρα ο χώρος δεν χωρίζεται απαραίτητα στη μέση. Απλά διαβάζει τα αρχεία-σημεία σειριακά και χωρίζει το χώρο πάνω σε αυτά. Πρόκειται απλώς για ένα διαφορετικό διαμερισμό του χώρου.

γ) Αναζήτηση των αρχείων που υπάρχουν σε μία περιοχή αναζήτησης → `std::vector<kd_node> search(kd_node* r, int level, std::vector<std::string> coords, std::vector<kd_node> &answer)`: το πρώτο όρισμα είναι ο τρέχον κόμβος, το δεύτερο είναι το επίπεδο του τρέχοντα κόμβου, το 3ο είναι το διάνυσμα συντεταγμένων της περιοχής αναζήτησης και το τελευταίο είναι το διάνυσμα απάντησης, που περιέχει τα αρχεία εντός της περιοχής που ορίζεται από το 3ο όρισμα. Το διάνυσμα απάντησης είναι αυτό που επιστρέφεται.

Η συνάρτηση καλεί αρχικά τη make_region() για τον τρέχοντα κόμβο, η οποία προς το παρόν θεωρούμε ότι επιστρέφει την περιοχή ενός κόμβου με κάποιο «μαγικό τρόπο», μιας που θα την αναλύσουμε παρακάτω.

Έλεγχος 1ης τερματικής περίπτωσης: ο τρέχον κόμβος είναι φύλλο

Αν ο έλεγχος είναι θετικός, **γίνεται έλεγχος για το εάν ο κόμβος περιέχεται στην περιοχή αναζήτησης**.

> *Αν όντως περιέχεται*, **προστίθεται στην απάντηση**.

Αν ο έλεγχος είναι αρνητικός, γίνεται *έλεγχος για την επόμενη τερματική περίπτωση: εσωτερικός κόμβος του οποίου η περιοχή βρίσκεται εξ' ολοκλήρου εντός της περιοχής αναζήτησης.*

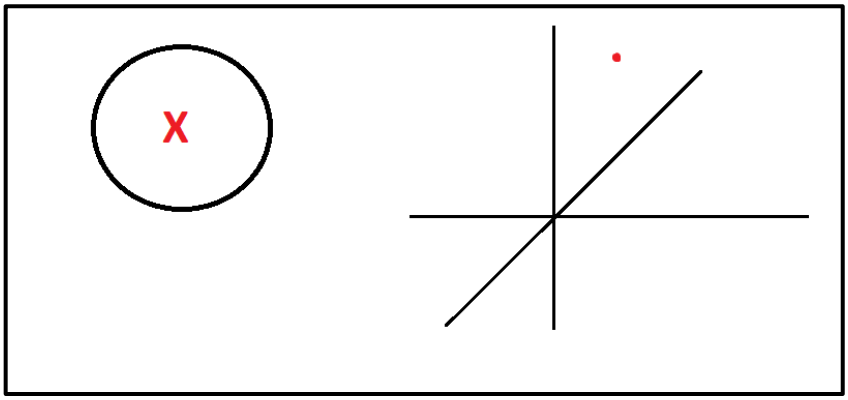
> *Αν όντως συμβαίνει αυτό*, **τότε όλοι οι κόμβοι κάτω από τον τρέχοντα και ο τρέχοντας προστίθενται στην απάντηση**.
>
> *Αν είναι και η 2η τερματική περίπτωση δεν ισχύει*, γίνεται **έλεγχος για ύπαρξη τομής μεταξύ των περιοχών καθενός από τα παιδιά του τρέχοντος και της περιοχής αναζήτησης**. *Αν υπάρχει*, τότε **η συνάρτηση καλείται αναδρομικά για καθένα από τα παιδιά του τρέχοντα κόμβου**. Μετά την επιστροφή από τα παιδιά**, εξετάζεται μήπως ο τρέχοντας κόμβος ανήκει και αυτός στην περιοχή αναζήτησης για να τον προσθέσει στην απάντηση.**

δ) Εκτύπωση του δέντρου → `void printTree(kd_node *r)`: το όρισμα αντιστοιχεί στον τρέχον κόμβο. Η συνάρτηση εκτυπώνει όλους τους κόμβους του δέντρου και χρησιμοποιήθηκε μόνο για testing της ορθής κατασκευής της δομής.

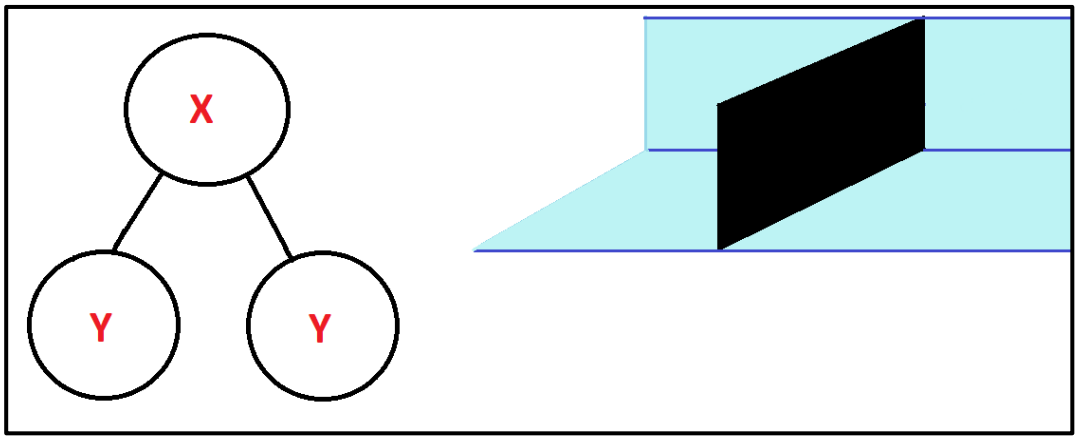
ε) ενδοδιατεταγμένη προσθήκη όλων των κόμβων κάτω από ένα κόμβο n και του κόμβου n σε διάνυσμα με → `void reportInorder(kd_node *r, std::vector<kd_node> &report)`: το 1ο όρισμα είναι ο τρέχον κόμβος και το δεύτερο είναι το διάνυσμα απάντησης, που θα περιέχει όλους τους κόμβους που προστέθηκαν.

Τώρα που έχουμε εξηγήσει τους κανόνες εισαγωγής και διάτρεξης του δέντρου, μπορούμε να επιστρέψουμε για να ολοκληρώσουμε την ανάλυση της συνάρτησης για δημιουργία περιοχής ενός κόμβου:

1. Αν ο κόμβος βρίσκεται στο επίπεδο 0, δηλαδή αν ο κόμβος είναι η ρίζα, τότε η περιοχή είναι όλο το επίπεδο (-inf σημαίνει -∞ και ~ inf σημαίνει +∞, το σύμβολο ~ επιλέχθηκε αντί του + γιατί το + προηγείται του - σαν χαρακτήρας, οπότε υπήρχε κίνδυνος να δημιουργηθεί μη έγκυρη περιοχή, ενώ το ~ έπεται του - και δεν δημιουργεί πρόβλημα).



2. Αν ο κόμβος βρίσκεται στο επίπεδο 1, τότε αυτός έχει μοναδικό πρόγονο, τη ρίζα, με x-value για αναζήτηση. Άρα η περιοχή του κόμβου είναι ένας από τους δύο ημιχώρους που δημιουργούνται αν χωρίσουμε τον 3d χώρο πάνω στο x = x_{root}. Το ποιος από τους 2 ημιχώρους είναι ο σωστός, εξαρτάται από το είδος του παιδιού που αποτελεί ο κόμβος στη ρίζα. Αν είναι το αριστερό της παιδί, τότε $x_{node} < x_{root}$ → αριστερός ημιχώρος, διαφορετικά ο δεξής, όπως φαίνεται στην εικόνα:

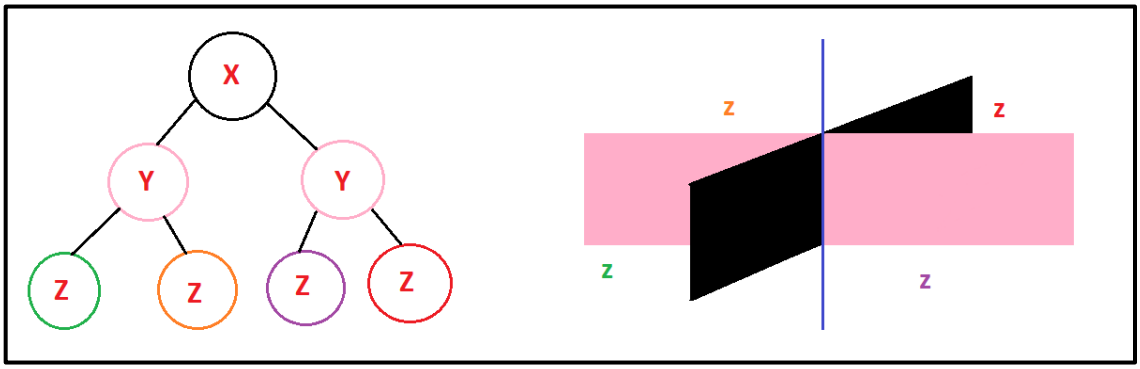


Προσοχή! Στο αριστερό σχήμα, δεν εννοούμε ότι τα δύο παιδιά της ρίζας έχουν την ίδια τιμή y, απλά ενοποιούμε τις περιπτώσεις το παιδί της ρίζας με τιμή y να βρίσκεται δεξιά ή αριστερά της στο σχήμα. Το ίδιο ισχύει και για τις εικόνες που ακολουθούν.

Στο δεξί σχήμα, έχουμε ζωγραφίσει το επίπεδο x =x_root (μαύρο) που χωρίζει σε δύο ημιχώρους, το «δάπεδο» και τον «πίσω τοίχο» του κάθε ημιχώρου αν τους σκεφτούμε σαν δωμάτια. Στην πραγματικότητα όμως, τα δωμάτια αυτά, έχουν μόνο έναν τοίχο, το επίπεδο x = x_root και εκτείνονται προς το άπειρο τόσο στον άξονα y όσο και στον z και προς τις 2 κατευθύνσεις, όπως κάνει και το επίπεδο x = x_root.

3. Αν ο κόμβος βρίσκεται στο επίπεδο 2, έχει 2 προγόνους. Τον πατέρα (άμεσος πρόγονος) Y και την ρίζα X. Άρα η περιοχή του θα οριοθετείται από αυτές τις 2 συντεταγμένες. Ανάλογα με το είδος του παιδιού που αποτελεί τόσο ο τρέχον κόμβος, όσο και ο πατέρας του, έχουμε τις εξής περιπτώσεις για την περιοχή:
    a. x > x_root, y>y_par, zER (πατέρας δεξί παιδί στη ρίζα και τρέχον κόμβος δεξί παιδί στον πατέρα)
    b. x > x_root, y<y_par, zER (πατέρας δεξί παιδί στη ρίζα και τρέχον κόμβος αριστερό παιδί στον πατέρα)
    c. x < x_root, y>y_par, zER (πατέρας αριστερό παιδί στη ρίζα και τρέχον κόμβος δεξί παιδί στον πατέρα)
    d. x < x_root, y<y_par, zER (πατέρας αριστερό παιδί στη ρίζα και τρέχον κόμβος αριστερό παιδί στον πατέρα)

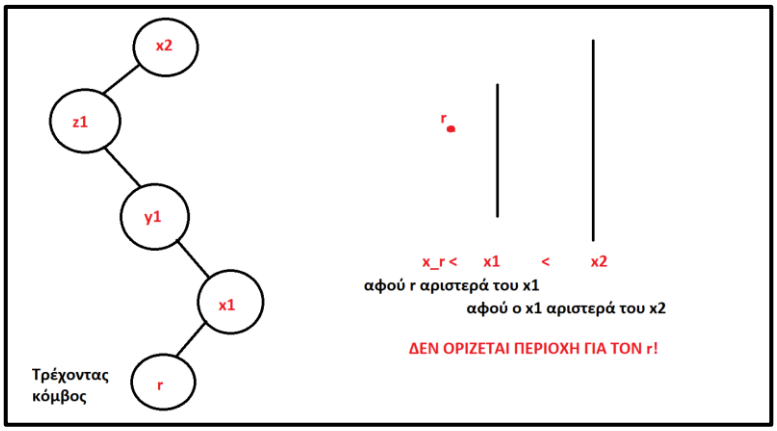
    Δηλαδή, η περιοχή του τρέχοντα κόμβου είναι ένας από τους 4 υποχώρους που δημιουργούνται από τα επίπεδα x=x_root και y=y_par.



Όπου x = x_root το μαύρο επίπεδο, y=y_par το ροζ επίπεδο. Για να καταλάβουμε καλύτερα τους 4 υποχώρους, πρέπει να φανταστούμε ότι τα 2 αυτά επίπεδα εκτείνονται προς το άπειρο, τόσο στον άξονα στον οποίο είναι παράλληλα (το ροζ στον x και το μαύρο στον y), όσο και στον άξονα z.

4. Για επίπεδα d>2: έχουμε σίγουρα τις συντεταγμένες x1, y1, z2 στους 3 άμεσους προγόνους (η σειρά με την οποία συναντάμε τις συντεταγμένες εξαρτάται από το ύψος του κόμβου) και ψάχνουμε για τις x2, y2, z2. Ξεκινάμε , από τον πατέρα του 3^{ου} προγόνου του τρέχοντα κόμβου και ταξιδεύουμε προς τα επάνω στο δέντρο, συγκρίνοντας κάθε φορά την κατάλληλη συντεταγμένη. Τα πρώτα έγκυρα x2, y2, z2 που θα βρούμε αποθηκεύονται ως συντεταγμένες που οριοθετούν την περιοχή του κόμβου. Έγκυρες είναι οι συντεταγμένες αν ακολουθούν την διάταξη coord2 < coord1 και ο τρέχον είναι αριστερό παιδί στον πατέρα του (coord2 < coord$_r$ <

coord1) ή αν ισχύει η αντίστροφη διάταξη και ο τρέχον κόμβος είναι δεξί παιδί του πατέρα του (coord1 < $coord_r$ < coord2), όπου coord η κατάλληλη κάθε φορά συντεταγμένη. Εξηγούμε γιατί, με ένα παράδειγμα:



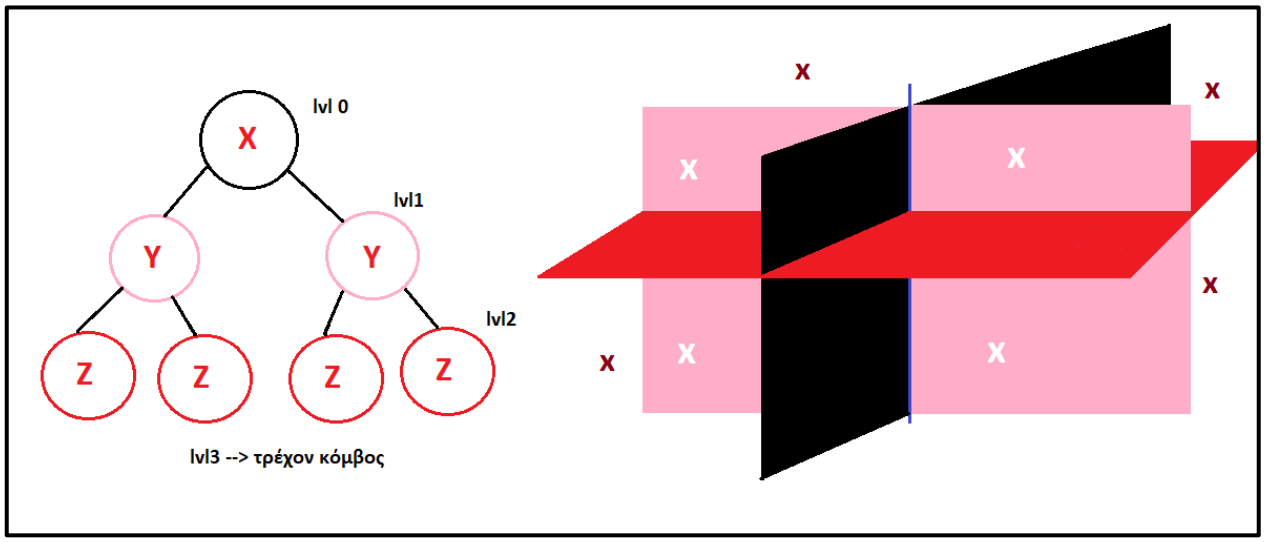
Η διαδρομή από το x2 στο x1 είναι τυχαία, εκτός από την ακμή x2 → z1.

Στην αντίθετη περίπτωση, η διάταξη θα ήταν x_r > x1 > x2, και πάλι η περιοχή δεν θα ήταν έγκυρη.

Αν βρούμε έγκυρες high τιμές και για τις 3 συντεταγμένες, σταματάμε το ψάξιμο και διαμορφώνουμε περιοχή της μορφής [x1, x2, y1, y2, z1, z2].

Διαφορετικά, αν το ταξίδι ξεπεράσει τη ρίζα του δέντρου, τότε το ψάξιμο σταματάει. Οι τιμές για κάποιες συντεταγμένες θα λείπουν και η περιοχή θα περιέχει το άπειρο σε κάποιες συντεταγμένες. Το ποιες συντεταγμένες έχουν έγκυρες τιμές φανερώνουν τα flags “valid_x”, “valid_y”, “valid_z” για την ομώνυμη συντεταγμένη και “valid_all” αν όλες είναι έγκυρες. Σχηματίζουμε κάθε φορά τη σωστή περιοχή ακολουθώντας τη διάταξη x1<x2, y1<y2, z1<z2 και προσθέτουμε τα σωστά άπειρα (+∞ ή -∞) όπου χρειάζονται, ανάλογα με τις τιμές των Boolean μεταβλητών που αναφέρθηκαν.

Ιδιαίτερο ενδιαφέρον παρουσιάζει η περίπτωση όπου ο τρέχον κόμβος έχει βάθος 3:



Η περιοχή είναι ένας εκ τους 8 υποχώρους που δημιουργούνται από τα επίπεδα x=x_root (μαύρο επίπεδο), y=y_grandpar (ροζ επίπεδο) και z = z_par (κόκκινο επίπεδο). Τα x που παρουσιάζονται με λευκό στο δεξί σχήμα, είναι αυτά με y> y_grandpar, ενώ αυτά που φαίνονται με μπορντό-καφέ, είναι αυτά με y<y_grandpar. Πιθανές περιοχές και πού αντιστοιχούν:

- [x_root, +∞, y_grandpar, +∞, z_par, +∞) → πάνω-δεξιά μπορντό x

- (-∞, x_root, y_grandpar, +∞, z_par, +∞) → πάνω-αριστερά μπορντό x
- [x_root, +∞, -∞, y_grandpar, z_par, +∞) → πάνω-δεξιά λευκό x
- (-∞, x_root, -∞, y_grandpar, z_par, +∞) → πάνω-αριστερά λευκό x
- [x_root, +∞, y_grandpar, +∞, -∞, z_par]→ κάτω-δεξιά μπορντό x
- (-∞, x_root, y_grandpar, +∞, -∞, z_par] → κάτω-αριστερά μπορντό x
- [x_root, +∞, -∞, y_grandpar, -∞, z_par] → κάτω-δεξιά λευκό x
- (-∞, x_root, -∞, y_grandpar, -∞, z_par] → κάτω-αριστερά λευκό x

Ακολουθεί η ανάλυση της επόμενη δενδρικής δομής, R-tree.

R-trees

Δομή: Τα R-trees, αποτελούνται από 4άδες κόμβων. Κάθε τετράδα θα την αποκαλούμε «γκρουπ».

Κόμβος: Κάθε κόμβος μπορεί να είναι είτε ένα MBR, είτε ένα σημείο στον τρισδιάστατο χώρο αλφαριθμητικών. Έτσι, κάθε γκρουπ περιέχει είτε κόμβους-MBR, είτε κόμβους-σημεία:

- Ένα MBR προσδιορίζεται από 6 διαστάσεις: [x_low, x_high, y_low, y_high, z_low, z_high] που αποθηκεύονται στον πίνακα **mbr_coords** και έχει έναν pointer **ptr** που δείχνει στο παιδί του κόμβου (το οποίο μπορεί να είναι γκρουπ από MBRs ή από σημεία) και έναν pointer **in_group** που δείχνει στο group στο οποίο ανήκει ο κόμβος.
- Ένα σημείο προσδιορίζεται από 3 διαστάσεις [x_p, y_p, z_p] που αποθηκεύονται στον πίνακα **mbr_coords** και δεν έχει παιδιά. Τα σημεία είναι αποθηκευμένα μόνο στα γκρουπ-φύλλα, ενώ κάθε γκρουπ που δεν είναι φύλλο περιέχει MBRs (εξ' ου και η χρήση του δείκτη παιδιού σε αυτά). Έχει όμως ένα **filepath**, επειδή ένα σημείο στον τρισδιάστατο χώρο αλφαριθμητικών αναπαριστά ένα αρχείο και έναν pointer **in_group** που δείχνει στο group στο οποίο ανήκει ο κόμβος.

Οι δύο δομές κόμβων που περιγράψαμε ενοποιήθηκαν σε μία, έχοντας NULL τιμή στα πεδία που μένουν αχρησιμοποίητα κάθε φορά. Αυτό αυξάνει τη χωρική πολυπλοκότητα της δομής, ακολουθήθηκε όμως για λόγους απλότητας κατά την κατασκευή του δέντρου. Πιο αναλυτικά, αν είχαμε ξεχωριστή δομή για σημεία και MBRs, θα έπρεπε να είχαμε και ξεχωριστή δομή γκρουπ για το καθένα. Έτσι, κάθε γκρουπ από MBR θα έπρεπε να έχει, είτε έναν pointer προς MBR-group είτε έναν pointer προς Point-group, ανάλογα με το είδος του κόμβου που θα περιείχε το γκρουπ-παιδί του. Υπάρχουν 2 τρόποι για την υλοποίηση αυτής της δομής που παρουσιάζεται ως εναλλακτική:

1. 2 pointers σε κάθε MBR-node με 1 NULL κάθε φορά.
2. Ορισμός κλάσης για τον κόμβο και κλήση διαφορετικού constructor κάθε φορά (1 για point-node, 1 για MBR-node με παιδί γκρουπ από point-nodes, 1 για MBR-node με παιδί γκρουπ από MBR-nodes).

Σε κάθε περίπτωση, χρειάζεται να κρατάμε πληροφορία για το επίπεδο του κόμβου (τα φύλλα βρίσκονται στο επίπεδο 0), ώστε να γνωρίζουμε το είδος του παιδιού και άρα το είδος του pointer που πρέπει να χρησιμοποιήσουμε.

Για κάθε κόμβο υλοποιήθηκαν οι εξής **λειτουργίες:**

α) αποθήκευση των συντεταγμένων στον πίνακα mbr_coords →void get_coords()

β) έλεγχος αν το σημείο ανήκει σε μία περιοχή → bool belongs_to()

γ) δημιουργία MBR στον κόμβο-πατέρα, που να περιέχει όλα τα σημεία ή MBRs των κόμβων του γκρουπ παιδιού του →void makeMBR(): η συνάρτηση αποθηκεύει τις συντεταγμένες όλων των σημείων που περιέχονται στο παιδί-γκρουπ ή όλων των MBRs που περιέχονται στο παιδί-γκρουπ (ανάλογα με το είδος του παιδιού) σε ένα διάνυσμα. Στη συνέχεια αρχικοποιεί τις συντεταγμένες του MBR που θα φτιάξει με τις low συντεταγμένες του πρώτου MBR που περιέχεται στο διάνυσμα. Για τα σημεία δεν ορίζονται low συντεταγμένες (τα σημεία έχουν μια συντεταγμένη x, μία y και μία z οπότε ο προσδιορισμός low και high είναι περιττός), οπότε οι συντεταγμένες του MBR που κατασκευάζεται αρχικοποιούνται με τις αντίστοιχες συντεταγμένες του πρώτου σημείου του διανύσματος. Πλέον, η συνάρτηση δημιουργεί το ελάχιστο δυνατό MBR που περιέχει όλα τα σημεία ή MBRs του γκρουπ-παιδιού του, ώστε να ικανοποιείται η απαίτηση για ελάχιστη ολική κάλυψη (εξηγείται αναλυτικά παρακάτω) .

δ) χρειάστηκε επίσης overload κάποιων τελεστών: ένα σημείο θεωρείται μικρότερο από ένα άλλο, αν και μόνο αν: $x_1 < x_2$ ή $x_1 = x_2$ και $y_1 < y_2$, ή $x_1 = x_2$ και $y_1 = y_2$ και $z_1 < z_2$.

Γκρουπ: Κάθε γκρουπ αποτελείται από έναν πίνακα 4 κόμβων **arr** και ένα **parent** pointer που δείχνει στο γκρουπ-πατέρα. Ισχύει η ιδιότητα των R-trees σύμφωνα με την οποία κάθε γκρουπ έχει μεταξύ m=2 και M=4 παιδιά, τιμές τις οποίες αναθέσαμε εμείς αυθαίρετα και με βάση τα παραδείγματα από την παράδοση του μαθήματος. Για κάθε group υλοποιήθηκαν οι εξής **λειτουργίες:**

α) Group κόμβους-σημεία → int groupnodes()

β) Group κόμβους-MBRs → int groupMBRs()

Οι δύο συναρτήσεις αυτές, αποφασίζουν, με βάση το κριτήριο ελάχιστης ολικής κάλυψης, πώς πρέπει να γίνει το grouping των αντίστοιχων κόμβων, όταν απαιτείται split και επιστρέφουν έναν αριθμό που δηλώνει την ομαδοποίηση. Split απαιτείται όταν σε ένα γεμάτο group πρέπει να προστεθεί ένας ακόμα κόμβος (περισσότερες λεπτομέρειες για αυτό αργότερα). Η ομαδοποίηση γίνεται πάντα 3+2 κόμβοι στα νέα groups, εξ' αιτίας της ιδιότητας που αναφέραμε παραπάνω. Συνεπώς, έχουμε να ελέγξουμε 3 περιπτώσεις, δεδομένου ότι όλοι οι κόμβοι του group μαζί με τον προς εισαγωγή έχουν ταξινομηθεί σε ένα ενιαίο vector:

1. Ομάδα 1: οι κόμβοι των θέσεων 0,1,2 και Ομάδα 2: οι κόμβοι των θέσεων 3, 4
2. Ομάδα 1: οι κόμβοι των θέσεων 1,2,3 και Ομάδα 2: οι κόμβοι των θέσεων 0, 4
3. Ομάδα 1: οι κόμβοι των θέσεων 2,3,4 και Ομάδα 2: οι κόμβοι των θέσεων 0,1

Σε αυτό το σημείο τα πράγματα διαφοροποιούνται αρκετά: για κόμβους-σημεία, υπολογίζουμε την ευκλείδια απόσταση μεταξύ διαδοχικών σημείων (αυτή είναι η ελάχιστη δεδομένου ότι τα σημεία είναι ταξινομημένα, και εφόσον μας ενδιαφέρει η ελάχιστη ολική

κάλυψη, δεν υπάρχει καλύτερη απόσταση από αυτή). Για το σκοπό αυτό χρησιμοποιούμε μία συνάρτηση εύρεσης απόστασης μεταξύ αλφαριθμητικών, η οποία μετράει την απόσταση μεταξύ των πρώτων γραμμάτων των 2 λέξεων που αντιστοιχούν στις ίδιες συντεταγμένες. Μετά τους υπολογισμούς, προσθέτουμε τις αποστάσεις μεταξύ των σημείων που ανήκουν στην ίδια ομάδα για κάθε μία από τις 3 παραπάνω περιπτώσεις και επιλέγουμε τη μικρότερη συνολική απόσταση. Η διαδικασία γίνεται πιο κατανοητή με ένα παράδειγμα:

Vector των κόμβων – σημείων για γκρουπαρισμα:

ΘΕΣΗ	**0**	**1**	**2**	**3**	**4**
ΚΟΜΒΟΣ	Α	Β	Γ	Δ	Ε

Vector αποστάσεων μεταξύ διαδοχικών ζευγαριών:

ΘΕΣΗ	**0**	**1**	**2**	**3**
ΑΠΟΣΤΑΣΗ	(ΑΒ)	(ΒΓ)	(ΓΔ)	(ΔΕ)

Vector συνολικής απόσταση ανά ομαδοποίηση:

ΘΕΣΗ	**0**	**1**	**2**
ΣΥΝΟΛΙΚΗ ΑΠΟΣΤΑΣΗ ΟΜΑΔΟΠΟΙΗΣΗΣ	(ΑΒΓ) + (ΔΕ)	(ΒΓΔ) + (ΑΔ)	(ΓΔΕ) + (ΑΒ)
ΕΠΙΣΤΡΕΦΟΜΕΝΗ ΤΙΜΗ	12	123	234

Παρατηρούμε ότι η επιστρεφόμενη τιμή που εξάγεται από την ελάχιστη συνολική απόσταση ομαδοποίησης, μας δηλώνει τις θέσεις των σημείων που πρέπει να ομαδοποιήσουμε στο αρχικό vector.

Για τους κόμβους – MBRs, η διαδικασία είναι αντίστοιχη αλλά με όγκους. Επίσης, σε αυτή την περίπτωση υπάρχει μία παραδοχή για διευκόλυνση. Όπως και στο 2d χώρο, έτσι και στον 3d το split γίνεται επιλέγοντας ένα x, που δημιουργεί ένα yz επίπεδο και «χωρίζει» το χώρο στα 2. Εφόσον τα MBRs είναι ταξινομημένα και η ομαδοποίηση γίνεται 3+2, το x που θα επιλέξουμε είναι αυτό του 3^{ου} MBR. Η μόνη απόφαση που πρέπει να πάρουμε, είναι αν αυτό το MBR θα βρίσκεται στα δεξιά ή τα αριστερά αυτού του επιπέδου (δηλαδή σε ποια ομάδα θα μπει, φτιάχνοντας την τριάδα), πάλι με βάση το κριτήριο ελάχιστης ολικής κάλυψης. Αρχικά υποθέτουμε ότι θα μπει στη μία μεριά και υπολογίζουμε τον συνολικό όγκο των MBRs σε εκείνη τη μεριά. Κάνουμε το ίδιο, υποθέτοντας αυτή τη φορά ότι το μεσαίο MBR θα ανήκει στην άλλη μεριά. Στο τέλος, επιλέγουμε τον μικρότερο συνολικό όγκο και επιστρέφουμε 012, αν αυτός αντιστοιχεί στην αριστερά μεριά του επιπέδου x = x_high του MBR του οποίου τη θέση εξετάζουμε ή 234, αν αυτός αντιστοιχεί στην δεξιά μεριά του επιπέδου x = x_low του MBR του οποίου τη θέση εξετάζουμε. Ουσιαστικά, προσθέτουμε το μεσαίο MBR στην μεριά που έχει τη μεγαλύτερη «έλλειψη όγκου».

γ) Επιλογή κόμβου προς επέκταση →choose2extend(): η συγκεκριμένη συνάρτηση, επεκτείνει «εικονικά» όλους τους κόμβους και επιλέγει αυτόν με τον μικρότερο όγκο μετά την εικονική επέκταση, ώστε να επεκταθεί και πραγματικά.
δ) Έλεγχος για το εάν ένα σημείο ή ένα MBR περιέχεται στο γκρουπ: επιλέγεται συνθήκη ισότητας για τα πεδία filepath και ptr αντίστοιχα, ακριβώς επειδή αυτά είναι μοναδικά σε κάθε στιγμιότυπο της κάθε δομής.

R-tree: Αποτελεί κλάση. Περιέχει ένα πεδίο για το γκρουπ-ρίζα του δέντρου και παρέχει τις παρακάτω **λειτουργίες:**

α) Αναδρομικός διαχωρισμός groups → splitrecur(`group_of_nodes& to_spl, node& to_ins, int count`). Το πρώτο όρισμα είναι ο τρέχον κόμβος που πρέπει να διαχωριστεί, το δεύτερο ο κόμβος που πρέπει να προστεθεί στο τρέχον γκρουπ και εξαιτίας του προκαλείται η διάσπαση και το τρίτο όρισμα είναι η φορά κλήσης της συνάρτησης ($1^η$, $2^η$, $3^η$ φορά κοκ). Η διαδικασία μπορεί να διαχωριστεί σε φάσεις:

Φάση 1: όλοι κόμβοι του τρέχοντος group και ο to_ins, αντιγράφονται σε ένα vector, το οποίο ταξινομείται με βάση των τελεστή < (βλ. Κόμβος → δ) Επειδή η εισαγωγή γίνεται πάντα σε γκρουπ-φύλλα, η splitrecur καλείται την πρώτη φορά (count=1) για κάποιο γκρουπ-φύλλο. Από τη δεύτερη φορά και μετά (αν χρειάζεται), πρέπει να διαχωριστούν εσωτερικά groups, δηλαδή MBR-groups.

Φάση 2: Αποφασίζεται ο κατάλληλος για κάθε περίπτωση διαχωρισμός με κλήση της συνάρτησης groupnodes() ή groupMBRs() ανάλογα με το είδος του group που πρέπει να «σπάσει».

Φάση 3: δημιουργούνται 2 νέα groups που παραλαμβάνουν τα αντίγραφα των κόμβων που δημιουργήθηκαν στη φάση 1, σύμφωνα με την ομαδοποίηση που αποφασίστηκε.

Αν το group διαχωρισμού είναι η ρίζα:

Φάση 4: Δημιουργείται ένα ακόμα group (“new root”)που παίρνει τη θέση της ρίζας και έχει 2 MBR-nodes: έναν που δείχνει στο group που δεν περιέχει τον νεοεισαχθέντα κόμβο, και έναν που δείχνει στο άλλο group (αυτό που περιέχει τον νεοεισαχθέντα κόμβο). Εφόσον η ρίζα του δέντρου εξισωθεί με τη new_root, η τελευταία διαγράφεται, μαζί με το κόμβο που έσπασε (to_spl). Ο διαχωρισμός που φτάνει στη ρίζα αυξάνει το ύψος του δέντρου κατά 1.

Αν το group διαχωρισμού δεν είναι η ρίζα:

Φάση 4: Το group που δεν περιέχει τον νεοεισαχθέντα κόμβο, παραμένει με τον ίδιο πατέρα, δηλαδή τον πατέρα του κόμβου “to_spl”, που διαχωρίστηκε. Ωστόσο, ο πατέρας τροποποιείται, ώστε να πετύχουμε ελάχιστη κάλυψη.

Φάση 5: Το group που περιέχει τον νεοεισαχθέντα κόμβο χρειάζεται καινούριο πατέρα, προκειμένου να δημιουργηθεί ένα MBR που να τον περιέχει. Ο νέος MBR-node πατέρας (“new_reg”) εισάγεται στο group-πατέρα του κόμβου που διαχωρίστηκε. Αν υπάρχει

χώρος για αυτόν, η διαδικασία τελειώνει διαγράφοντας τον to_spl. Διαφορετικά, η συνάρτηση καλείται αναδρομικά.

ΣΗΜΑΝΤΙΚΗ ΣΗΜΕΙΩΣΗ! Σε κάθε περίπτωση, στις φάσεις 4 και 5 λαμβάνουν χώρα και οι συνδέσεις προγόνων-απογόνων και ύπαρξης σε group (με τη χρήση του δείκτη in group).

β) Εύρεση σημείου εισαγωγής → Access(`group_of_nodes* r, node& to_ins, std::vector<std::string> crd, bool &flag`). Το 1ο όρισμα είναι το τρέχον group που εξετάζουμε. Το 2ο όρισμα είναι ο κόμβος-σημείο προς εισαγωγή, το 3ο όρισμα είναι οι συντεταγμένες του to_ins και το τελευταίο είναι ένα flag που δηλώνει αν το group στο οποίο πρέπει να γίνει η εισαγωγή είναι γεμάτο ή όχι. Το 3ο όρισμα είναι περιττό, επειδή η πληροφορία είναι ήδη αποθηκευμένη εντός του κόμβου. Η διαπίστωση αυτή έγινε πολύ αργά, λόγω διορθώσεων που έγιναν καθώς ολοκληρωνόταν το project, οπότε κρατήσαμε την κοστοβόρα αυτή επιλογή, για λόγους ασφαλούς διατήρησης της ήδη δοκιμασμένα ορθής λύσης και έλλειψης χρόνου για περεταίρω διόρθωση. Η λειτουργία της συνάρτησης παρουσιάζεται αλγοριθμικά:

Έλεγχος τερματικής περίπτωσης: βρίσκεται σε group φύλλο

Αν ο έλεγχος είναι θετικός (i=4), **ψάχνει για ελεύθερη θέση στο group**

Αν υπάρχει ελεύθερη θέση, η συνάρτηση **επιστρέφει το group και τερματίζει**

Διαφορετικά, **ενεργοποιεί το flag, επιστρέφει το group και τερματίζει**.

Αν ο έλεγχος είναι αρνητικός (i != 4), πρόκειται για εσωτερικό κόμβο, οπότε **ψάχνει αν ο to_ins ανήκει σε κάποιο MBR**.

Αν όντως ανήκει σε κάποιο, τότε η Access **καλείται αναδρομικά για το group-παιδί του τρέχοντα κόμβου** (στην επόμενη κλήση της, το group-παιδί θα είναι ο τρέχον). Αν ο to_ins ανήκει σε πάνω από 1 MBRs, προστίθεται σε αυτό που διατρέχει πρώτο η Access.

Αν δεν ανήκει σε κάποιο, τότε πρέπει κάποιο MBR να επεκταθεί για να περιέχει το νέο σημείο:

Το ποιο αποφασίζεται από την **choose2extend()**. Δυστυχώς, η τελευταία επιστρέφει το MBR που πρέπει να επεκταθεί, αλλά όχι κατά πόσο ή από ποια μεριά, παρόλο που το έχει υπολογίσει.

Συνεπώς, **η Access** πρέπει να ξανακάνει τον έλεγχο και **να τροποποιήσει το επιλεγμένο MBR**.

Μόλις το κάνει, **καλείται αναδρομικά για το παιδί του τρέχοντα κόμβου**.

Επιστροφή του group στο οποίο πρέπει να γίνει η εισαγωγή

γ) Εισαγωγή νέου κόμβου-σημείου → Insert(`std::string filepath`): Το όρισμα είναι η απόλυτη διαδρομή του αρχείου που θέλουμε να εισάγουμε. Η εισαγωγή γίνεται μόνο στα group-φύλλα και μόνο για κόμβο-σημείο, εφόσον θέλουμε η δομή μας να δεικτοδοτεί αρχεία. Δεν

μπορούμε να εισάγουμε στο δέντρο MBRs. Η συνάρτηση αρχικοποιεί ένα νέο κόμβο-σημείο με τις συντεταγμένες του και το filepath και θέτει την τιμή της μεταβλητής full (που συμβολίζει γεμάτο group-φύλλο) σε false. Στη συνέχεια, καλεί την Access, για να βρει το group-φύλλο στο οποίο πρέπει να γίνει η εισαγωγή (insertion_group). Αν η Access δεν κατέληξε σε γεμάτο group-φύλλο, τότε ο κόμβος-σημείο προς εισαγωγή (to_ins) εξισώνεται με την πρώτη κενή θέση του group (ουσιαστικά ο κενός κόμβος που υπήρχε σε εκείνη τη θέση πήρε της τιμές το to_ins και έγινε κόμβος-σημείο). Σε αντίθετη περίπτωση, χρειάζεται να γίνει διαχωρισμός του group, οπότε καλείται η splitrecur(insertion_group, *to_ins, 1) (βλ. splitrecur για το νόημα των ορισμάτων).

δ) Αναζήτηση →Search(`group_of_nodes *cur, std::vector<std::string> query, std::vector<std::string> &answer`): Θέλουμε να βρούμε τα αρχεία που ανήκουν σε μία περιοχή, σύμφωνα με τους ορισμούς που έχουμε δώσει στις παραδοχές. Άρα, αναζητούμε κόμβους σημεία που ανήκουν στην περιοχή αναζήτησης (θυμίζουμε ότι οι κόμβοι σημεία βρίσκονται μόνο σε group-φύλλα).

Το 1ο όρισμα της συνάρτησης είναι το τρέχον group που εξετάζεται, το 2ο είναι οι συντεταγμένες που ορίζουν την περιοχή αναζήτησης και το 3ο είναι το διάνυσμα απάντησης. Αυτό θα περιέχει στο τέλος, όλα τα αρχεία που ανήκουν στην περιοχή αναζήτησης. Παρουσιάζουμε τη λειτουργία της με τη μορφή αλγορίθμου:

Έλεγχος τερματικής περίπτωσης: Βρίσκεται σε group-φύλλο

Αν ο έλεγχος είναι θετικός, τότε **για κάθε κόμβο-σημείο του group, αποθηκεύει σε ένα προσωρινό vector τις συντεταγμένες** του και **ελέγχει αν αυτό ανήκει στην περιοχή αναζήτησης.**

Αν ανήκει, η συνάρτηση **προσθέτει το filepath του κόμβου-σημείου στην απάντηση.**

Αν ο έλεγχος είναι αρνητικός, πρόκειται για εσωτερικό MBRs-group. **Για κάθε MBR, αποθηκεύονται οι συντεταγμένες του σε ένα προσωρινό vector και ακολουθούν 2 έλεγχοι με την σειρά**:

1. Έλεγχος, *αν το τρέχον MBR περιέχεται εξ' ολοκλήρου στην περιοχή αναζήτησης*: στην περίπτωση αυτή, **προστίθενται στην απάντηση όλοι οι κόμβοι σημεία που βρίσκονται σε όλα τα group-φύλλα κάτω από αυτό το MBR-node**. Το έργο αυτό αναλαμβάνει η **reportAll_leaves()**, που εξηγείται αμέσως παρακάτω.
2. Διαφορετικά, *έλεγχος αν το τρέχον MBR είναι εντελώς εκτός της περιοχής αναζήτησης*. *Αν **δεν** είναι*, τότε υπάρχει τομή μεταξύ του MBR και της περιοχής αναζήτησης και **η αναζήτηση συνεχίζεται εντός αυτού του MBR με αναδρομική κλήση της search για αναζήτηση στο group-παιδί του τρέχοντος MBR-node**.

Επιστροφή του διανύσματος απάντησης

ε) Προσθήκη όλων των κόμβων σημείων κάτω από ένα κόμβο σε διάνυσμα → reportAll_leaves(`node* r, std::vector < std::string > & report`): το 1ο όρισμα είναι ο τρέχον κόμβος που εξετάζεται, κάτω από τον οποίο θέλουμε όλους τους κόμβους σημεία και το 2ο είναι το διάνυσμα που θα περιέχει αυτούς τους κόμβους-σημεία.

Αν μέσα σε ένα group, φτάσουμε σε κόμβο που έχει τον δείκτη in_group = NULL, τότε στη συγκεκριμένη θέση του group δεν υπάρχει κόμβος. Επειδή η κόμβοι προστίθενται με τη σειρά σε ένα group, στην 1η κενή θέση που θα βρεθεί, ύπαρξη ενός κενού κόμβου στη θέση i, $2 \leq i \leq 3$ (για $i < 2$ καταστρατηγείται η ιδιότητα του δέντρου για πλήθος παιδιών από 2 έως και 4) σημαίνει ότι και οι επόμενες θέσεις του group είναι κενές και το group έχει λιγότερους από 4 κόμβους. Έτσι, η συνάρτηση απλώς επιστρέφει την κλήση της.

Σε αντίθετη περίπτωση (in_group != NULL), η θέση του group περιέχει κόμβο. Αν αυτός έχει ptr = NULL, πρόκειται για κόμβο-σημείο, (όπως έχουμε ήδη εξηγήσει τόσο από τον ορισμό της δομής κόμβου σημείου όσο και από την αδυναμία ένδειξης σε επόμενο group, εφόσον οι κόμβοι σημεία βρίσκονται σε group-φύλλα) άρα τον προσθέτει στην απάντηση. Από την άλλη, αν έχει κενό filepath, τότε ο τρέχον κόμβος είναι κόμβος-MBR και η συνάρτηση καλείται αναδρομικά για κάθε ένα από τα παιδιά του.

στ) εκτύπωση του δέντρου →printTree(`group_of_nodes* r`): το όρισμα είναι το τρέχον group που τυπώνεται. Η συνάρτηση αυτή τυπώνει όλους του κόμβους και χρησιμοποιήθηκε μόνο για έλεγχο ορθού Construction του δέντρου.