ΠΟΛΥΔΙΑΣΤΑΤΕΣ ΔΟΜΕΣ ΔΕΔΟΜΕΝΩΝ

Ακαδημαϊκό Έτος 2023-2024

Project στις Πολυδιάστατες Δομές Δεδομένων

ΠΑΝΑΓΙΩΤΗΣ ΚΑΛΟΖΟΥΜΗΣ	1084560
ΚΑΛΛΙΝΙΚΟΣ ΚΥΡΙΑΚΟΥΛΟΠΟΥΛΟΣ	1084583
ΣΠΥΡΙΔΩΝ ΖΗΚΟΣ	1084581

Περιεχόμενα

1. Εισαγωγή

Το βασικό αρχείο του project είναι το main.py. Μέσα σ' αυτό έχουμε ορίσει μια συνάρτηση εκτέλεσης ενός ερωτήματος πάνω σε όλες τις πολυδιάστατες δομές που έχουμε υλοποιήσει, καθώς επίσης και μια διαδικασία παραγωγής τυχαίων ερωτημάτων, ώστε να αξιολογήσουμε πειραματικά τις δομές μας. Στη συνέχεια εξηγούμε τα περιεχόμενα του αρχείου αυτού και παρουσιάζουμε ένα παράδειγμα εκτέλεσης.

1.1 Περιγραφή κώδικα:

Συνάρτηση build_trees

Η συνάρτηση αυτή κατασκευάζει τα 4 δέντρα που θέλουμε να συγκρίνουμε. Παράλληλα καταγράφει τους χρόνους κατασκευής κάθε δομής. Για τις μετρήσεις χρησιμοποιούμε τη συνάρτηση perf_counter. Παρακάτω φαίνεται ένα παράδειγμα για το R-Tree:

```
# R-Tree construction
t0 = perf_counter()

rr = r_tree.Rtree(jdata)
rr.build_tree()

t1 = perf_counter()

runtimes["r_tree"] = t1-t0
trees["r_tree"] = rr
```

Αφού κατασκευάσει όλα τα δέντρα, επιστρέφει ένα λεξικό της παρακάτω μορφής:

```
trees = {"r_tree", "range_tree", "quad_tree", "kd_tree"}
```

Συνάρτηση query_all

Η συνάρτηση αυτή εκτελεί ένα συγκεκριμένο ερώτημα για επιστήμονες με ονόματα που ανήκουν αλφαβητικά σε ένα εύρος τιμών, έχουν βραβεία πάνω από ένα κατώφλι και έχουν αριθμό δημοσιεύσεων σε ένα εύρος τιμών. Έτσι, δέχεται τις παρακάτω παραμέτρους:

- n1: Κάτω όριο για το όνομα
- n2: Άνω όριο για το όνομα
- a1: Κάτω όριο για τα βραβεία
- d1: Κάτω όριο για το DBLP

- d2: Άνω όριο για το DBLP

Η συνάρτηση επίσης δέχεται στην παράμετρο trees ένα λεξικό με όλα τα δέντρα που έχουμε κατασκευάσει, όπως αυτό επιστρέφεται από τη συνάρτηση build_trees.

Για το ερώτημα που εκτελείται η συνάρτηση μετράει τον χρόνο απάντησης κάθε δομής. Παρακάτω φαίνεται ένα παράδειγμα για το R-Tree:

```
# R-Tree query
t0 = perf_counter()
ids_rtree = rr.search(n1,n2,a1,d1,d2)
t1 = perf_counter()
runtimes["r_tree"] = t1-t0
```

Με την εκτέλεση του ερωτήματος, κάθε δέντρο επιστρέφει τα IDs των πολυδιάστατων σημείων που βρίσκονται στο σύνολο απάντησης. Τα IDs αυτά, καθώς και όλοι οι χρόνοι εκτέλεσης, επιστρέφονται τελικά από τη συνάρτηση:

```
return runtimes, ids_rtree
```

Συνάρτηση run_experiments

Η συνάρτηση αυτή εκτελεί επαναληπτικά την query_all με ερωτήματα τα οποία παράγονται τυχαία. Ο αριθμός των ερωτημάτων καθορίζεται από την παράμετρο num_of_experiments. Παρακάτω φαίνεται ο τρόπος με τον οποίο παράγουμε τα τυχαία ερωτήματα:

```
n1 = random.choice(string.ascii_lowercase)
n2 = random.choice(n1 + string.ascii_lowercase.split(n1)[1])
a1 = random.randint(0,15)
d1 = random.randint(0,500)
d2 = random.randint(d1,500)
```

Για κάθε ερώτημα που εκτελείται, η συνάρτηση καταγράφει τις παραπάνω παραμέτρους, καθώς επίσης και τους χρόνους εκτέλεσης και τα επιστρεφόμενα IDs, όπως αυτά επιστρέφονται από την query_all.

Συνάρτηση unique_range_query

Η συνάρτηση run_experiments εκτελεί μια σειρά από τυχαία πειράματα. Η συνάρτηση unique_range_query μάς επιτρέπει να εκτελούμε ένα ερώτημα του οποίου τις παραμέτρους καθορίζουμε εμείς κατά τον χρόνο εκτέλεσης.

Συνάρτηση lsh_run

Η συνάρτηση lsh_run δέχεται τα IDs των πολυδιάστατων σημείων που επιστράφηκαν από τα δέντρα μας (αρκεί να πάρουμε τα αποτελέσματα από ένα δέντρο, δεδομένου ότι έχουμε ελέγξει πως επιστρέφουν τα ίδια αποτελέσματα) και ένα threshold ομοιότητας στο LSH. Έπειτα εφαρμόζει LSH πάνω στα κείμενα που δείχνουν τα IDs αυτά. Στο τέλος αποθηκεύει τα αποτελέσματα στο αρχείο lsh.txt. Ο χρόνος εκτέλεσης του LSH επιστρέφεται από τη συνάρτηση.

1.2 Οδηγίες και παράδειγμα εκτέλεσης:

Για την εκτέλεση της εφαρμογής απαιτούνται τα παρακάτω πακέτα:

```
pip install pandas
pip install openpyxl
pip install matplotlib
```

Ο χρήστης τρέχει το βασικό αρχείο της εφαρμογής, που είναι το main.py:

```
python main.py
```

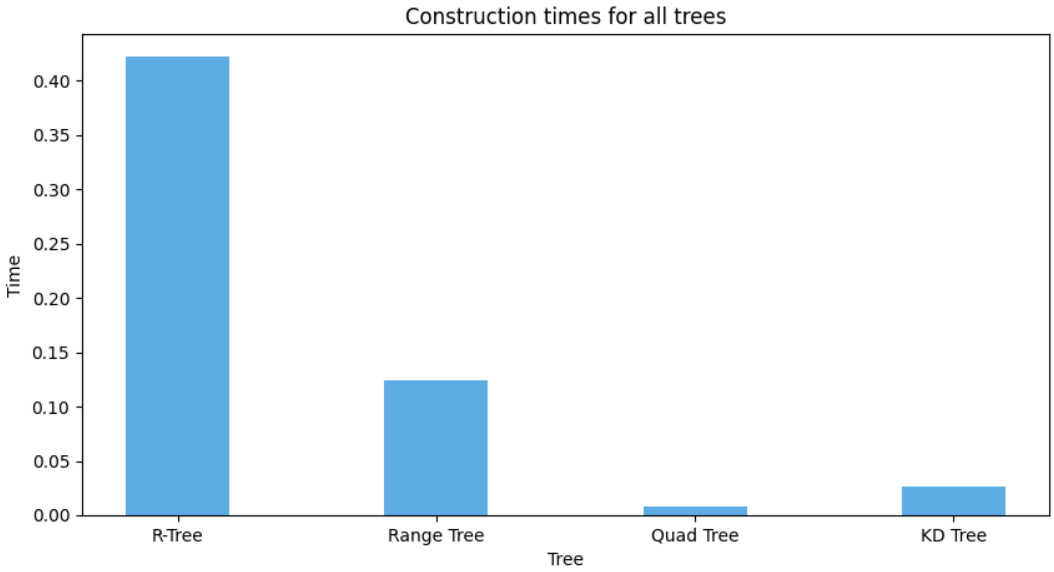
Ο χρήστης πρέπει να εισάγει τον αριθμό των τυχαίων ερωτημάτων που θέλει να εκτελεστούν:

```
If you set num of range queries equal to 1 you give the inputs manually
otherwise the parameters are configured automatically.

Select the number of range queries to be executed:
```

Ο χρήστης εισάγει την επιλογή του και έπειτα κατασκευάζονται τα δέντρα. Μετράμε τους χρόνους κατασκευής και τους εκτυπώνουμε στην οθόνη:

```
Building trees...

Construction times
========================================
> R-Tree: 0.42176579986698925

> Range Tree: 0.12431210000067949

> Quad Tree: 0.008301399881020188

> KD Tree: 0.02624080004170537
```

Επίσης κατασκευάζουμε και ένα διάγραμμα με τους χρόνους αυτούς. Όλα τα διαγράμματα που κατασκευάζουμε αποθηκεύονται μέσα στον φάκελο “plots/”:



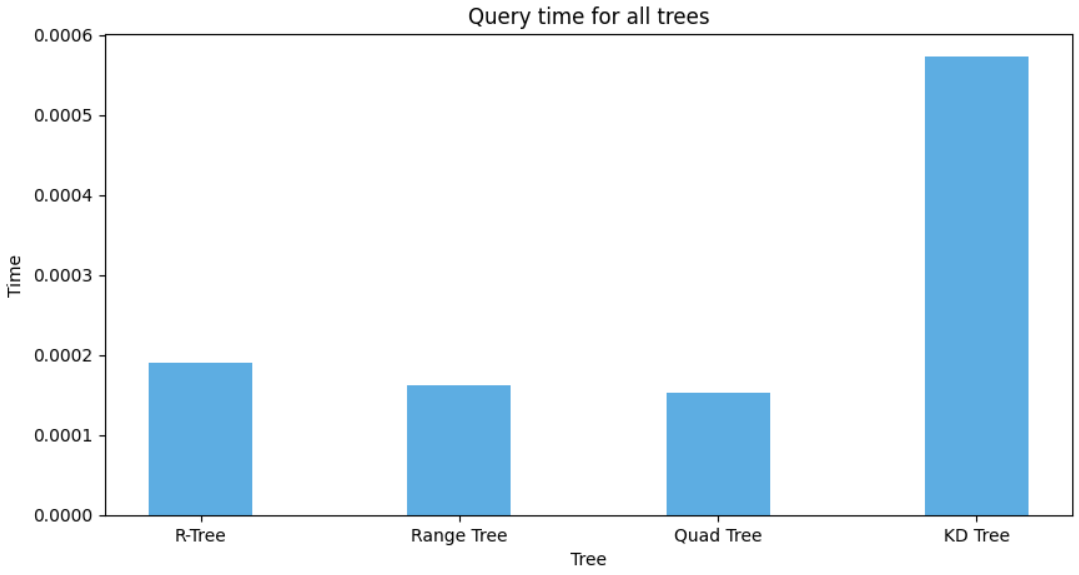
Αν ο χρήστης εισάγει 1, σημαίνει πως δεν εκτελείται κανένα τυχαίο πείραμα και πρέπει ο ίδιος να δώσει τις παραμέτρους:

```
Give min name letter: a
Give max name letter: k
Give min awards number: 2
Give min dblp record number: 10
Give max dblp record number: 100
```

Όπως και στην κατασκευή των δέντρων, έτσι και εδώ εκτυπώνουμε τους χρόνους εκτέλεσης του ερωτήματος για κάθε δομή:

```
Query times
========================================
> R-Tree: 0.0001908999402076006

> Range Tree: 0.0001625001896172762

> Quad Tree: 0.00015300000086426735

> KD Tree: 0.0005723000504076481
```

Επίσης κατασκευάζουμε και ένα διάγραμμα:



Μόλις επιστραφούν τα αποτελέσματα από τα δέντρα, τρέχουμε και το LSH. Για το LSH ο χρήστης πρέπει να δώσει ένα threshold ομοιότητας:

```
Give similarity threshold [0-1]:
```

Δίνουμε ένα threshold και τρέχει το LSH, εκτυπώνοντας τον χρόνο εκτέλεσης και αποθηκεύοντας τα αποτελέσματα στο αρχείο lsh.txt:

```
Give similarity threshold [0-1]: 0.3

LSH time: 0.4162341000046581

Saving results to 'lsh.txt'.
```

Παρακάτω φαίνεται ένα τμήμα του αρχείου:

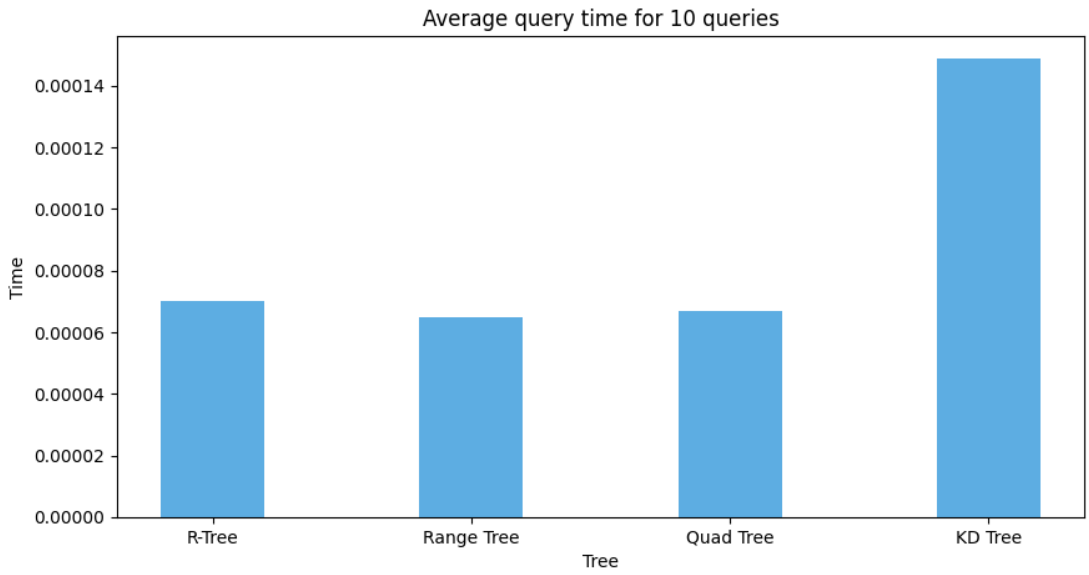
```
9: Luis von Ahn
47: Lenore Blum

14: Lisa Anthony
47: Lenore Blum

286: Dan Ingalls
288: Kenneth E. Iverson

11: Frances E. Allen
138: Whitfield Diffie

72: Fred Brooks
286: Dan Ingalls
312: Manolis Kellis

9: Luis von Ahn
72: Fred Brooks
288: Kenneth E. Iverson

131: Dorothy E. Denning
189: Charlotte Froese Fischer
```

Στο παραπάνω αρχείο βλέπουμε τις ομάδες ομοιότητας διαχωρισμένες με κενή γραμμή. Σε κάθε ομάδα οι επιστήμονες (τους οποίους γράφουμε ως ζεύγη ID - Όνομα) έχουν κείμενα εκπαίδευσης που είναι όλα όμοια μεταξύ τους κατά το threshold που έχουμε ορίσει.

Αν, αντί για 1, ο χρήστης δώσει μεγαλύτερο πλήθος ερωτημάτων, τα ερωτήματα αυτά παράγονται με αυτοματοποιημένο τρόπο και μετράμε τους μέσους χρόνους εκτέλεσης για κάθε δομή:

```
Average query times for 10 queries
========================================
> R-Tree: 7.01600220054388e-05

> Range Tree: 6.506992504000664e-05

> Quad Tree: 6.678998470306396e-05

> KD Tree: 0.00014870003797113897
```

Παρακάτω φαίνεται και το διάγραμμα που απεικονίζει τους χρόνους αυτούς:



Αν ο χρήστης θέλει να δει πιο λεπτομερώς την επίδοση κάθε δομής σε κάθε ερώτημα, μπορεί να ελέγξει το αρχείο queries.xlsx, το οποίο περιέχει όλους τους χρόνους εκτέλεσης για όλα τα τυχαία ερωτήματα που εκτελέστηκαν:

Query	R-Tree	Range Tree	Quad Tree	KD Tree
0	7,78E-05	6,91E-05	4,68E-05	0,0001476
1	3,71E-05	3,97E-05	3,28E-05	6,4E-05
2	3,49E-05	1,98E-05	2,54E-05	3,44E-05
3	3,57E-05	1,26E-05	2,09E-05	7,96E-05
4	4,21E-05	4,26E-05	3,11E-05	4,75E-05
5	0,000228	0,0002185	0,000276	0,0005488
6	7,65E-05	7,03E-05	6,77E-05	0,0001979
7	0,000101	0,000115	0,0001202	0,0002357
8	3,91E-05	2,53E-05	2,44E-05	4E-05
9	2,9E-05	3,78E-05	2,26E-05	9,15E-05

Στο αρχείο ο χρήστης μπορεί επίσης να δει ακριβώς τα ερωτήματα που εκτελέστηκαν:

Query	n1	n2	a1	d1	d2
0	c	z	12	489	497
1	k	m	9	326	417
2	x	z	10	401	492
3	y	y	13	8	132
4	a	e	9	248	401
5	d	v	3	11	113
6	h	x	10	125	355
7	m	x	1	283	309
8	c	c	10	396	489
9	a	m	15	440	496

Από τα αποτελέσματα που επιστρέφουν οι δομές σε κάθε ερώτημα τρέχουμε LSH με κατώφλι ομοιότητας 0.3, εκτυπώνοντας τον μέσο χρόνο εκτέλεσης:

```
Average LSH time: 0.20981243997812227
```

Όλα τα παραπάνω ήταν απλά παραδείγματα εκτέλεσης. Περισσότερα για την απόδοση των διαφορετικών δομών θα αναλύσουμε στη συνέχεια.

2. K-D tree

2.1 Περιγραφή κώδικα:

Το δέντρο kd κατασκευάζεται από την κλάση KDTree, στην οποία εισάγονται τα δεδομένα ως σημεία (αντικείμενα της κλάσης Datapoint). Το δέντρο αποτελείται από κόμβους (αντικείμενα της κλάσης KDNode) και κάθε κόμβος έχει αποθηκευμένο τουλάχιστον 1 σημείο.

Κλάση Datapoint:

Η κλάση αναπαριστά ένα σημείο στο οποίο αποθηκεύονται τα δεδομένα (name, awards, dblp, education, id).

Κλάση KDNode:

Η κλάση αναπαριστά κόμβους του δέντρου όπου υπάρχουν αποθηκευμένα τα δεδομένα του κόμβου σε μορφή Datapoint και τα παιδιά του κόμβου (δεξί, αριστερό).

Κλάση KDTree:

- Κατασκευή:

Με αυτή την κλάση αναπαρίσταται το kd δέντρο. Πριν την κατασκευή του δέντρου θα πρέπει να εισαχθούν τα δεδομένα. Δίνεται η λίστα με τα δεδομένα και από αυτή τη λίστα δημιουργείται άλλη με τα σημεία τύπου Datapoint για το δέντρο. Η νέα λίστα με τα σημεία χρησιμοποιείται για να παραχθούν 3 λίστες ταξινομημένες ως προς το κάθε index. Αφού παραχθούν οι ταξινομημένες λίστες μπορεί να κατασκευαστεί το δέντρο.

Για την κατασκευή καλείται η συνάρτηση construct(…) της κλάσης KDTree με ορίσματα τις 3 ταξινομημένες λίστες ως προς το κάθε index και το βάθος όπου είναι 0 στην αρχή.

Κάθε φορά θα πρέπει να εξετάζουμε ανάλογα με το βάθος ως προς ποιο index θα γίνει η σύγκριση. Έπειτα υπολογίζεται το μέσο της λίστας για το index που επιλέχθηκε και τέλος γίνεται σύγκριση των τιμών με τις τιμές του μέσου ώστε να διαχωριστούν σε 2 λίστες (δεξιά και αριστερά). Αν οι τιμές ενός σημείου είναι ίσες με τις τιμές του μέσου τότε το σημείο καταχωρείται στον ίδιο κόμβο.

Τέλος καλείται η συνάρτηση construct(…) 2 φορές, στην πρώτη κλήση δίνονται ορίσματα τα σημεία που διαχωρίστηκαν δεξιά του μέσου και το βάθος+1 και στη δεύτερη κλήση δίνονται ορίσματα τα σημεία που διαχωρίστηκαν αριστερά του μέσου και το βάθος+1.

- Ερώτημα εύρους:

Όταν δοθεί ένα εύρος τιμών για κάθε index για να επιστραφούν οι κόμβοι που ανήκουν σε αυτό το εύρος ξεκινάει αναζήτηση από τη ρίζα του δέντρου. Κάθε φορά ανάλογα με το βάθος στο οποίο βρισκόμαστε κάνουμε σύγκριση και με το κατάλληλο index. Στην αρχή το βάθος είναι 0 και εξετάζουμε εάν η τιμή που επιλέχθηκε με βάση το βάθος ανήκει στο πεδίο τιμών που ζητείται. Αν ο έλεγχος είναι επιτυχής τότε θα πρέπει να διαπιστώσουμε εάν και οι υπόλοιπες τιμές του κόμβου ανήκουν στα ζητούμενα πεδία τιμών. Αφού ολοκληρωθεί αυτή η διαδικασία προχωράμε στα παιδιά του κόμβου και ξανακαλούμε την συνάρτηση rangeQuery(…).

Για να προχωρήσουμε στον επόμενο κόμβο γίνεται σύγκριση της τιμής του κόμβου (επιλέγεται ποια από τις τιμές με βάση το βάθος) με την αντίστοιχη τιμή του εύρους ώστε να γνωρίζουμε ποια παιδιά θα πρέπει να εξετάσουμε.

Τέλος επισημαίνεται ότι ένας κόμβος μπορεί να περιέχει >1 σημεία, τα οποία βέβαια έχουν ίδιες τιμές άρα αν ανήκει το ένα από αυτά στο ζητούμενο διάστημα τότε ανήκουν και τα υπόλοιπα.

3. Quad Tree

3.1 Περιγραφή κώδικα:

Για την κατασκευή του quad tree διακρίνω 2 είδη κόμβων, οι κόμβοι που διαχωρίζουν το τρισδιάστατο επίπεδο σε ίσα μέρη και οι κόμβοι που αποθηκεύουν τα 3d δεδομένα. Για την δημιουργία των κόμβων χρησιμοποιώ τις κλάσεις Node1 και Node2.

Επίσης χρησιμοποιείται και η κλάση Octree η οποία αναπαριστά το δέντρο που περιέχει τους κόμβους. Οι κόμβοι του δέντρου είναι κλάσεις Node1 και Node2.

Κλάση Node1:

Είναι κόμβος που διαχωρίζει τον τρισδιάστατο χώρο σε 8 ίσους κύβους. Αποθηκεύει τις συντεταγμένες του σημείου (x,y,z), την τιμή των μέσων για κάθε διάσταση ώστε στην επόμενη διαμέριση να υπολογιστούν οι συντεταγμένες του επόμενου σημείου και τα παιδιά του τα οποία είναι κόμβοι είτε Node1 είτε Node2.

Κλάση Node2:

Είναι κόμβος στον οποίο αποθηκεύεται η 3d πληροφορία. Έχει συντεταγμένες οι οποίες καθορίζουν τη θέση του στο χώρο και διάφορες άλλες πληροφορίες όπως το όνομα του επιστήμονα και έναν μοναδικό αριθμό για τον κάθε επιστήμονα.

Κλάση Octree:

Είναι το δέντρο το οποίο αποτελείται από κόμβους Node1 και Node2. Το δέντρο αρχικοποιείται μέσω της συνάρτησης __init__(…) και έτσι δημιουργείται η ρίζα του δέντρου η οποία είναι τύπου Node1. Η ρίζα του δέντρου έχει συντεταγμένες που προκύπτουν από τα μέσα των διαστημάτων της κάθε διάστασης. Για παράδειγμα αν δίνονται τα διαστήματα x=[0,10], y= [0,20], z= [10,30] τότε η ρίζα του δέντρου θα έχει συντεταγμένες (0+(10-0)/2 , 0+(20-0)/2, 10+(30-10)/2) = (5, 10, 20). Γενικά το μέσο μίας διάστασης υπολογίζεται από τον τύπο min_val + (max_val – min_val)/2. Επιπλέον η ρίζα του δέντρου κατά την αρχικοποίηση δεν έχει παιδιά.

Συναρτήσεις κλάσης Octree:

- ***insert_node(self, parent, node):***

Η εισαγωγή των δεδομένων στο δέντρο γίνεται από τη συνάρτηση insert_node(…). Τα δεδομένα εισάγονται στο δέντρο σε μορφή Node2. Στη συνάρτηση δίνεται όρισμα κάθε φορά η ρίζα του

δέντρου. Αρχίζει να εξετάζει από τη ρίζα αν υπάρχει θέση για να αποθηκευτεί και προχωράει μέχρι να βρει πατέρα με διαθέσιμη την κατάλληλη θέση για να γίνει παιδί.

Για την αποθήκευση του νέου κόμβου θα πρέπει να ελεγχθεί σε ποια από τις 8 θέσεις του πατέρα ανήκει με βάση τις συντεταγμένες. Στη συνέχεια αφού βρεθεί η κατάλληλη θέση από τη συνάρτηση find_position(…) θα ελεγχθεί εάν η θέση δεν έχει γεμίσει από άλλα παιδιά check_node_children(…).

Επομένως προκύπτουν 2 περιπτώσεις, στην πρώτη υπάρχει διαθέσιμη θέση άρα αποθηκεύεται και στη δεύτερη θα πρέπει να εξεταστεί ο επόμενος κόμβος αν είναι τύπου Node1 για να αποθηκευτεί σε αυτόν τον κόμβο ως παιδί.

Στην περίπτωση που ο επόμενος κόμβος δεν είναι τύπου Node1 θα χρειαστεί να δημιουργηθεί καινούργιος πατέρας.

Το πρόγραμμα διαπιστώνει πότε ο κόμβος είναι τύπου Node1 αν στον κόμβο στον οποίο είναι παιδί, η θέση στην οποία βρίσκεται έχει και άλλους κόμβους αποθηκευμένους. Μόνο αν δεν υπάρχουν άλλα παιδία σε αυτή την θέση ο κόμβος είναι τύπου Node1.

Όταν ο κόμβος είναι Node1 γίνεται ο νέος πατέρας και καλείται η συνάρτηση insert_node(…) για να αποθηκευτεί ο κόμβους που θέλουμε.

Στην αντίθετη περίπτωση όπου δεν υπάρχει διαθέσιμος πατέρας θα δημιουργήσουμε νέο κόμβο τύπου Node1 με τη συνάρτηση replace(…). Ο νέος πατέρας γίνεται παιδί του προηγούμενου στην θέση που εξετάζαμε και οι κόμβοι που ήταν παιδία σε αυτή τη θέση θα ξαναεισαχθούν στο δέντρο με κλήση της συνάρτησης insert_node(…) για το καθένα ξεχωριστά.

- ***replace(self, node, position):***

Έχει ως σκοπό την δημιουργία νέου κόμβου τύπου Node1 ο οποίος θα είναι παιδί ενός άλλου κόμβου Node1. Επομένως δίνονται οι συντεταγμένες του πατέρα και με βάση την θέση (μια από τις 8) στην οποία θα πρέπει να βρίσκεται ο νέος πατέρας υπολογίζονται κατάλληλα οι συντεταγμένες του.

- ***compare_nodes(self, parent_node, child_node):***

Χρησιμοποιείται για την σύγκριση των συντεταγμένων ενός εισερχόμενου κόμβου με τις συντεταγμένες του πατέρα. Ανάλογα με το αποτέλεσμα της σύγκρισης σε κάθε διάσταση θα προσδιοριστεί η θέση στην οποία θα τοποθετηθεί το παιδί.

- ***check_node_children(self, parent, position):***

Η συνάρτηση ελέγχει εάν ο κομβός-πατέρας έχει διαθέσιμο χώρο στη θέση που ζητείται για την προσθήκη ενός ακόμα παιδιού. Εάν ο πατέρας στη θέση που ζητείται έχει φτάσει το ανώτερο όριο παιδιών τότε επιστρέφει την τιμή false.

- ***find_position(self, co_x, co_y, co_z):***

Κάθε νεοεισερχόμενος κόμβος εξετάζεται σε ποια από τις 8 θέσεις του πατέρα πρέπει να αποθηκευτεί. Για να βρεθεί η θέση βασιζόμαστε στη σύγκριση των συντεταγμένων του νέου κόμβοι με τις συντεταγμένες του υποψήφιου πατέρα. Ανάλογα με τα αποτελέσματα της κάθε σύγκρισης προκύπτει η θέση που έχει στο επίπεδο, δηλαδή σε έναν από τους 8 κύβους που σχηματίζονται. Η αντιστοιχία του δυαδικού αριθμού που δίνει η σύγκριση των κόμβων με τον αριθμό του κύβου δίνεται στον πίνακα που ακολουθεί.

co_x	co_y	co_z		co_z'	co_y'	co_x^co_y	position
0	0	0		1	1	0	6
0	0	1		0	1	0	2
0	1	0		1	0	1	5
0	1	1		0	0	1	1
1	0	0		1	1	1	7
1	0	1		0	1	1	3
1	1	0		1	0	0	4
1	1	1		0	0	0	0

- ***traverse(self, node):***

Με την traverse εκτυπώνεται όλο το δέντρο ξεκινώντας από τη ρίζα και προχωρώντας σαν να κάνουμε αναζήτηση κατά βάθος.

- ***range_query(self, node, ranges, selected_nodes):***

Με αυτή τη συνάρτηση εκτελούνται ερωτήματα εύρους. Δίνονται τα εύρη τιμών για κάθε διάσταση και εξετάζεται ποιοι κόμβοι ανήκουν σε αυτά τα όρια. Η αναζήτηση αρχίζει από τη ρίζα, αλλά για να μην εξεταστούν όλοι οι κόμβοι του δέντρου ελέγχονται με βάση τα εύρη ποια από τα παιδιά του κόμβου είναι πιθανό να ανήκουν.

Επομένως συγκρίνονται οι συντεταγμένες του πατέρα με τα εύρη σε κάθε διάσταση και επιστρέφεται ένα σύνολο από πιθανές θέσεις. Για κάθε μια από αυτές τις θέσεις εξετάζεται αν υπάρχουν παιδία και ακολούθως ελέγχεται με την συνάρτηση node_in_range(…) αν ανήκουν στο εύρος ώστε να αποθηκευτούν στη λίστα. Αν κάποιο από τα παιδιά είναι πατέρας τότε ξανακαλείται η range_query(…) και επαναλαμβάνεται η διαδικασία.

- ***node_in_range(self, node, ranges):***

Ελέγχεται αν οι συντεταγμένες του κόμβου ανήκουν στα διαστήματα που δίνονται.

- ***possible_pos(self, node, ranges):***

Η συνάρτηση επιστρέφει ένα σύνολο με θέσεις στις οποίες είναι πιθανό οι κόμβοι της να ανήκουν στο εύρος τιμών που ζητείται.

<u>Παραδοχές</u>:

1. Ορίζω την εξής αντιστοίχιση: x: awards, y: dblp_records, z: name
2. Κάθε θέση ενός πατέρα μπορεί να χωρέσει μεταβλητό αριθμό παιδιών, στο πρόγραμμα έχει οριστεί 4 παιδιά ανά θέση.
3. Για την αρχικοποίηση του δέντρου χρησιμοποιείται η συνάρτηση init_quadTree(...) στην οποία δίνεται το αρχείο με τα δεδομένα και τα εισάγει στο δέντρο.
4. Στα ερωτήματα εύρους για το άνω όριο στο όνομα προστίθεται επιπλέον το γράμμα "z" ώστε να συμπεριληφθούν στο αποτέλεσμα και αυτά τα ονόματα.
5. Επίσης τα ονόματα εισάγονται στο δέντρο ως αριθμοί μέσω της συνάρτησης str_to_int(…).

- ***str_to_int(word):***

Όλα τα γράμματα μετατρέπονται σε πεζούς χαρακτήρες και κρατιέται το τελευταίο όνομα που διακρίνεται. Ακόμα επιλέγω όλα τα ονόματα να έχουν μήκος 4, για αυτό γίνεται περικοπή για ονόματα μήκους>4 και προσθήκη a στα ονόματα με μήκος<4. Τέλος κάθε γράμμα αντικαθίσταται από τον ascii αριθμό του και πολλαπλασιάζεται με το 26^θεση_γράμματος.

4. Range Tree

4.1 Επιλογές Σχεδίασης και Παραδοχές

Στην ενότητα αυτή καταγράφουμε τις σχεδιαστικές επιλογές που έγιναν κατά την υλοποίηση του Range Tree.

Σειρά αποθήκευσης των διαστάσεων:

Κάθε σημείο που αποθηκεύουμε στο Range Tree αποτελείται από 3 διαστάσεις: το όνομα, το dblp και τα βραβεία. Στο Range Tree τα δυαδικά δέντρα πάνω στα οποία κάνουμε τις αναζητήσεις οργανώνονται σε "επίπεδα", όπου κάθε επίπεδο αντιστοιχεί σε μια διάσταση αναζήτησης και κάθε δέντρο ενός επιπέδου δείχνει σε ένα εναλλακτικό δέντρο του επόμενου επιπέδου. Επομένως, πρέπει να επιλεγεί μια σειρά με την οποία θα διαπερνάμε τις τρεις αυτές διαστάσεις. Έχουμε καταλήξει στη σειρά "name, awards, dblp", ωστόσο το δέντρο έχει υλοποιηθεί ώστε να μπορεί να υποστηρίζει οποιαδήποτε άλλη σειρά, τροποποιώντας το λεξικό "index":

```
index = {3: "name", 2: "awards", 1: "dblp"}
```

Χρήση IDs:

Κάθε πολυδιάστατο σημείο που αποθηκεύουμε στο Range Tree χαρακτηρίζεται μοναδικά από ένα ID . Τα IDs αυτά προκύπτουν από τη σειρά με την οποία είναι αποθηκευμένα τα δεδομένα στο dataset που έχουμε συνθέσει και από το οποίο κατασκευάζεται το δέντρο. Αν και οι κόμβοι του δέντρου περιέχουν ολόκληρη την πληροφορία του πολυδιάστατου σημείου, επιλέχθηκε οι συναρτήσεις αναζήτησης να επιστρέφουν τελικά μόνο τα IDs των σημείων αυτών αντί για τα ίδια τα σημεία, ώστε όλες οι δομές που έχουμε υλοποιήσει να έχουν ένα κοινό σημείο αναφοράς, το οποίο θα είναι και εύκολα διαχειρίσιμο.

Φυλλοπροσανατολισμένο δέντρο:

Όλη η πληροφορία είναι αποθηκευμένη στα φύλλα, και συγκεκριμένα μόνο στα φύλλα των δέντρων του τελευταίου επιπέδου. Όλοι οι ενδιάμεσοι κόμβοι περιέχουν μόνο πληροφορία που χρησιμοποιείται για την αναζήτηση.

Ισοζύγιση:

Το Range Tree είναι ουσιαστικά μια συλλογή από πολλά δυαδικά δέντρα. Κάθε ένα από αυτά τα δέντρα μπορεί να υλοποιηθεί ως ισοζυγισμένο δέντρο, με σκοπό τη αποδοτικότερη ένθεση

στοιχειών και εκτέλεση ερωτημάτων. Για το Range Tree έχουμε επιλέξει κάθε δέντρο να είναι υλοποιημένο ως AVL.

Αναζήτηση ως προς το επίθετο:

Αν και ολόκληρο το όνομα ενός επιστήμονα αποθηκεύεται στους κόμβους πληροφορίας, μόνο το επίθετό του χρησιμοποιείται για την αναζήτηση. Σαν επίθετο έχουμε θεωρήσει το τελευταίο τμήμα του ονόματός του.

4.2 Περιγραφή του κώδικα

Στην ενότητα αυτή αναλύουμε τον κώδικα που υλοποιεί το Range Tree.

4.2.1 Η κλάση DataPoint

Κάθε κόμβος του RangeTree, εκτός από πληροφορία που αφορά τη δομή του δέντρου, περιέχει και τη χρήσιμη πληροφορία που θέλουμε να αποθηκεύεται στον κόμβο αυτό. Η κλάση DataPoint αποθηκεύει την πληροφορία αυτή. Πρακτικά, είναι ένα λεξικό που ομαδοποιεί την πληροφορία για ολόκληρο το όνομα (name), το πλήθος των βραβείων (awards), το DBLP record (dblp) και το κείμενο με την εκπαίδευση του επιστήμονα (education). Να σημειωθεί ότι αν και κάνουμε αναζήτηση μόνο με το επίθετο, καταλήγοντας σε κάποιους κόμβους που αντιστοιχούν στο επίθετο αυτό, οι κόμβοι αυτοί οφείλουν να διατηρούν αναλλοίωτο ολόκληρο το όνομα του επιστήμονα, μαζί και με τις υπόλοιπες πληροφορίες του. Τέλος, κάθε αντικείμενο DataPoint, εφόσον αντιστοιχεί σε ένα πολυδιάστατο σημείο του συνόλου δεδομένων μας, περιέχει και ένα πεδίο id, για τους λόγους που εξηγήσαμε προηγουμένως. Τελικά το ID είναι η μόνη πληροφορία του DataPoint που επιστρέφεται από τη συνάρτηση αναζήτησης.

4.2.2 Η κλάση RangeTreeNode

Η κλάση αυτή αναπαριστά έναν κόμβο του δέντρου. Έχει τα παρακάτω πεδία:

- **left**: Το αριστερό παιδί του κόμβου στο δέντρο
- **right**: Το δεξιό παιδί του κόμβου στο δέντρο
- **value**: Η τιμή που χρησιμοποιείται για την αναζήτηση. Όλοι οι κόμβοι που ανήκουν σε δέντρα του ίδιου επιπέδου (name, awards ή dblp) έχουν τιμές δεικτοδότησης του ίδιου τύπου δεδομένων (αλφαριθμητικό ή ακέραιος)
- **data**: Η χρήσιμη πληροφορία που περιέχει αυτός ο κόμβος. Είναι αντικείμενο τύπου DataPoint. Επειδή έχουμε θεωρήσει φυλλοπροσανατολισμένο δέντρο, μόνο τα φύλλα έχουν κάποια τιμή στο πεδίο αυτό, ενώ όλοι οι ενδιάμεσοι κόμβοι έχουν την τιμή None. Επίσης, επειδή μπορεί να υπάρχουν πολλά σημεία με την ίδια τιμή ως προς κάποια διάσταση, το data πρέπει να είναι λίστα από DataPoints.

- **alt_tree**: Κάθε κόμβος του Range Tree ουσιαστικά αντιστοιχεί σε ένα υποδέντρο με ρίζα τον κόμβο αυτό. Από τα δεδομένα που περιέχει το δέντρο αυτό, δηλαδή αυτά που βρίσκονται στα φύλλα του, παράγεται ένα εναλλακτικό υποδέντρο, το οποίο όμως τώρα είναι ταξινομημένο ως προς την επόμενη στη σειρά διάσταση. Το πεδίο αυτό αποθηκεύει αυτό το εναλλακτικό δέντρο. Περισσότερα για τον αλγόριθμο κατασκευής του Range Tree αναφέρουμε στη συνέχεια.

Εφόσον υλοποιούμε τα δυαδικά δέντρα του Range Tree ως AVL, χρειάζονται επιπλέον και τα παρακάτω πεδία για κάθε κόμβο:

- **parent**: Ο γονέας του κόμβου στο δέντρο. Χρειάζεται κατά την εισαγωγή ενός νέου κόμβου στο δέντρο, καθώς πρέπει να διασχίσουμε το δέντρο από το νέο φύλλο προς τη ρίζα ανανεώνοντας ύψη και ισοζυγίσεις.

- **height**: Το ύψος του κόμβου στο δέντρο. Το ύψος ορίζεται αναδρομικά, ως το ύψος του ψηλότερου παιδιού αυξημένο κατά 1. Τα φύλλα έχουν ύψος 0.

```
node.height = max(node.left.height, node.right.height) + 1
```

- **hb**: Η ισοζύγιση του κόμβου. Υπολογίζεται αφαιρώντας από το ύψος του δεξιού υποδέντρου το ύψος του αριστερού υποδέντρου. Χρησιμοποιείται για να δούμε αν χρειάζεται να εφαρμόσουμε κάποια περιστροφή ώστε να επαναφέρουμε τη ζυγισμένη μορφή στο δέντρο ύστερα από μια εισαγωγή.

```
node.hb = node.right.height - node.left.height
```

4.2.3 Κατασκευή του Range Tree

Το Range Tree υλοποιείται ως αντικείμενο της κλάσης RangeTree. Η κλάση αυτή περιέχει τα παρακάτω πεδία:

- **dataset**: Το σύνολο δεδομένων από το οποίο θα κατασκευαστεί το δέντρο μας. Είναι JSON αντικείμενο ή λίστα από DataPoints

- **root**: Η ρίζα του δέντρου. Είναι τύπου RangeTreeNode. Ουσιαστικά, ρίζα του Range Tree είναι η ρίζα του μεγαλύτερου υποδέντρου πρώτης διάστασης (name), από το οποίο μπορούμε να πάρουμε κάθε άλλο υποδέντρο μέσω δυαδικής διαπέρασης και των εναλλακτικών υποδέντρων.

- **dimension**: Η διάσταση στην οποία αντιστοιχεί ο κόμβος αυτός. Επειδή το δέντρο κατασκευάζεται αναδρομικά, ξεκινώντας από ένα Range Tree πολλών διαστάσεων και σταδιακά κατασκευάζοντας Range Trees μικρότερων διαστάσεων, η πρώτη διάσταση (σε εμάς το όνομα) στην πραγματικότητα αντιστοιχεί στον αριθμό 3, και καθώς μειώνεται αυτός ο αριθμός πάμε σε επόμενες διαστάσεις (awards και dblp)

Κατασκευάζουμε ένα αντικείμενο της κλάσης αυτής δίνοντας απλά το dataset από το οποίο θέλουμε να κατασκευαστεί το δέντρο:

```
data = None
with open("./data/dataset.json", "r", encoding = "utf-8") as f:
    data = json.load(f)

tree = RangeTree(data)
```

Αφού δώσουμε το dataset, το δέντρο αρχίζει και κατασκευάζεται αναδρομικά. Όπως αναφέραμε και πριν, ξεκινάμε από τη διάσταση 3 (όνομα) και πάμε προς χαμηλότερες διαστάσεις.

Για την περίπτωση όπου η διάσταση είναι 3, το πρώτο πράγμα που πρέπει να κάνουμε είναι να κατασκευάσουμε ένα δυαδικό δέντρο ως προς το όνομα. Για κάθε επιστήμονα στο σύνολο δεδομένων μας, κατασκευάζουμε ένα αντικείμενο DataPoint από τις πληροφορίες του, επισυνάπτοντας και το id το οποίο προκύπτει από τη σειρά με την οποία τους διαπερνάμε. Έπειτα, εισάγουμε το DataPoint αυτό στο δυαδικό δέντρο που ορίζεται από το root με τη συνάρτηση insert:

```
if self.dimension == 3: #Create tree from JSON file
    #Get the dataset as a JSON object

    for i, scientist in enumerate(dataset):
        self.insert(
        DataPoint(
            scientist["name"],
            scientist["awards"],
            scientist["dblp_records"],
            scientist.get("education", "EMPTY"),
            i
        ))
```

<u>Μέθοδος insert</u>

Η μέθοδος insert δέχεται ένα DataPoint και κάνοντας αναζήτηση καταλήγει στο φύλλο όπου πρέπει να κάνει την ένθεση, δημιουργώντας τελικά στο σημείο αυτό τον νέο κόμβο.

Αρχικά παίρνει, με βάση τη διάσταση που έχει το δέντρο, το κατάλληλο πεδίο από το DataPoint, ώστε να το χρησιμοποιήσει για τις συγκρίσεις. Αυτή η τιμή αργότερα θα γίνει το πεδίο value του νέου κόμβου:

```
value = x[index[self.dimension]]
```

Έπειτα, πρέπει να διακρίνουμε αν το δέντρο έχει ήδη ρίζα ή όχι. Στην περίπτωση όπου είναι κενό, δημιουργούμε απλά έναν νέο κόμβο και τον κάνουμε ρίζα:

```
if self.root == None:
    self.root = RangeTreeNode(value, [x])
```

Διαφορετικά, πρέπει να γίνει κλασσική αναζήτηση δυαδικού δέντρου, ώστε να εντοπίσουμε τον κόμβο όπου θα γίνει η ένθεση:

```
leaf = self.search(value)
```

Αφού εντοπίσουμε τον κόμβο, υπάρχουν 3 περιπτώσεις:

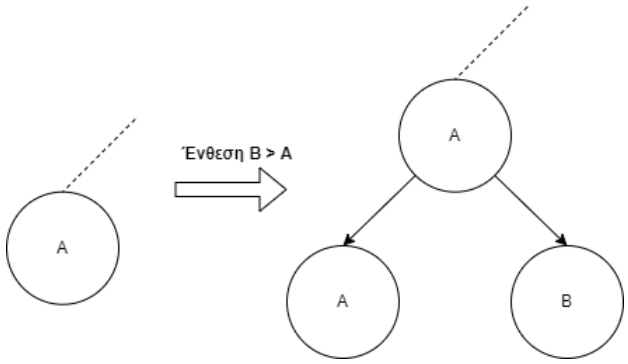
- Το φύλλο αυτό έχει ίδιο value με το value αναζήτησης, δηλαδή υπάρχει ήδη. Σε αυτήν την περίπτωση, απλά κάνουμε append στη λίστα data το νέο σημείο:

  ```
  leaf.data.append(x)
  ```

- Το φύλλο έχει μικρότερο value από αυτό που πάμε να βάλουμε. Σε αυτήν την περίπτωση, δημιουργούμε στο φύλλο δύο νέα παιδιά, όπου το αριστερό έχει το υπάρχον value (μικρότερη τιμή) και το δεξιό έχει το νέο value (μεγαλύτερη τιμή):

  ```
  leaf.left = RangeTreeNode(leaf.value, leaf.data, leaf)
  leaf.right = RangeTreeNode(value, [x], leaf)

  leaf.data = None
  ```

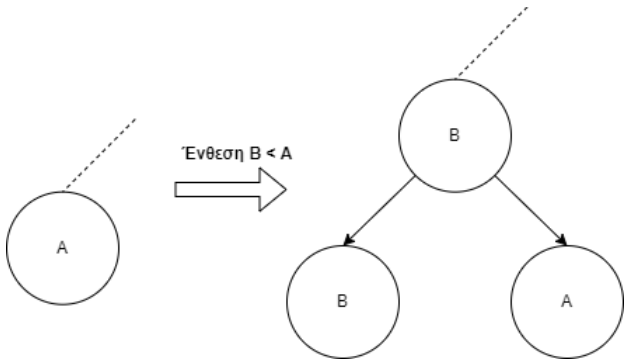
  Σχηματικά:

- Το φύλλο έχει μεγαλύτερο value από αυτό που πάμε να βάλουμε. Σε αυτήν την περίπτωση, δημιουργούμε στο φύλλο δύο νέα παιδιά, όπου το αριστερό έχει το νέο value (μικρότερη τιμή) και το δεξιό έχει το υπάρχον value (μεγαλύτερη τιμή). Αλλάζουμε επίσης το value του παλιού φύλλου, ώστε να είναι το value αναζήτησης, αφού ο πατέρας των δύο νέων φύλλων πρέπει να έχει ως value τη μικρότερη τιμή:

```
leaf.left = RangeTreeNode(value, [x], leaf)
leaf.right = RangeTreeNode(leaf.value, leaf.data, leaf)

leaf.value = value
leaf.data = None
```

Σχηματικά:



Παρατηρούμε ότι στις δύο τελευταίες περιπτώσεις πρέπει επίσης να μεταφέρουμε τα data του παλιού φύλλου, το οποίο τώρα έγινε εσωτερικός κόμβος, στο νέο φύλλο που το αντικαθιστά. Μόνο τα φύλλα πρέπει να έχουν δεδομένα.

Στην περίπτωση όπου εισάγουμε νέο φύλλο, πρέπει στο τέλος της insert να ελέγξουμε και την ισοζύγιση του δέντρου με τη μέθοδο balance. Τη μέθοδο αυτή, καθώς επίσης και ό,τι άλλο αφορά την ισοζύγιση, την αναλύουμε αργότερα:

```
self.balance(leaf)
```

Τώρα που τελειώσαμε με την ανάλυση της μεθόδου insert, συνεχίζουμε να εξηγούμε την κατασκευή του Range Tree. Αφού κατασκευάσουμε ένα ολόκληρο δυαδικό δέντρο (είτε όπως περιγράψαμε επάνω για dimension = 3, είτε αναδρομικά όπως θα εξηγήσουμε στη συνέχεια), πρέπει όσο υπάρχουν ακόμα διαστάσεις (dimension > 1) να διαπεράσουμε ολόκληρο το δέντρο και για κάθε υποδέντρο να κατασκευάσουμε το εναλλακτικό υποδέντρο του. Αυτό το κάνουμε με την αναδρομική μέθοδο create_alt_trees, η οποία δέχεται τη ρίζα του υποδέντρου. Αρχικά σε αυτήν δίνουμε τη ρίζα ολόκληρου του δέντρου που κατασκευάσαμε:

```
self.create_alt_trees(self.root)
```

Η μέθοδος αυτή ουσιαστικά κάνει ένα convergecast. Φτάνει προς τα φύλλα και συγκεντρώνει τα δεδομένα που αυτά έχουν στις λίστες τους. Κάθε φύλλο στέλνει τα δεδομένα προς τα πάνω. Κάθε ενδιάμεσος κόμβος παίρνει τις λίστες από τα παιδιά του και τις συγχωνεύει. Αυτή τη λίστα την στέλνει στον δικό του πατέρα και η διαδικασία επαναλαμβάνεται με τον ίδιο τρόπο.

Οι συγχωνευμένες λίστες σε κάθε ενδιάμεσο κόμβο είναι ουσιαστικά τα δεδομένα που έχει αποθηκευμένα το συγκεκριμένο υποδέντρο και χρησιμοποιούνται για την κατασκευή των εναλλακτικών υποδέντρων. Αρχικά, κάθε φύλλο κατασκευάζει το εναλλακτικό του υποδέντρο χρησιμοποιώντας ως dataset απλά τη λίστα data που διαθέτει:

```
if node.left is None:
    data = node.data
    node.alt_tree = RangeTree(data, self.dimension - 1)
    node.data = None
    return data
```

Κάθε ενδιάμεσος κόμβος κατασκευάζει το εναλλακτικό υποδέντρο με τη συγχωνευμένη λίστα που έχει κατασκευάσει:

```
leaves = []

leaves = leaves + self.create_alt_trees(node.left)
leaves = leaves + self.create_alt_trees(node.right)

node.alt_tree = RangeTree(leaves, self.dimension - 1)

return leaves
```

Παρατηρούμε ότι και στις δύο περιπτώσεις, το εναλλακτικό δέντρο που κατασκευάζουμε έχει μια διάσταση μικρότερη. Αυτό σημαίνει ότι πρέπει να προχωρήσουμε στην επόμενη διάσταση (από το name πάμε στο awards). Η διαδικασία θα λήξει όταν φτάσουμε τελικά σε διάσταση 1, καθώς η create_alt_trees στον constructor του RangeTree καλείται μόνο για dimension > 1:

```
if dimension > 1:
    self.create_alt_trees(self.root)
```

Τα δέντρα που κατασκευάσαμε παραπάνω φτιάχνονται επίσης με insert, όπως και στην περίπτωση dimension = 3, με τη διαφορά τώρα ότι το dataset προήλθε από μέσα το δέντρο και όχι από το αρχείο:

```
if self.dimension == 3: #Create tree from JSON file
    .
    .
    .
else: #Create tree using existing data in the dataset list
    for data_point in dataset:
        self.insert(data_point)
```

Μια τελευταία παρατήρηση είναι ότι επειδή τα δεδομένα μεταφέρονται προς τα φύλλα, και από τα φύλλα πάνε στα εναλλακτικά υποδέντρα των φύλλων, τελικά μόνο τα φύλλα του τελευταίου επιπέδου (dblp) έχουν πραγματικά δεδομένα.

Εδώ ολοκληρώνεται η κατασκευή του Range Tree.

4.2.4 Αναζήτηση

Η κλάση RangeTree έχει επίσης τις παρακάτω συναρτήσεις, οι οποίες υλοποιούν τα διάφορα ερωτήματα που μπορούμε να κάνουμε πάνω στη δομή αυτή:

- search: Αναζήτηση ακριβούς τιμής πάνω σε συγκεκριμένη διάσταση
- range_search: Αναζήτηση εύρους πάνω σε συγκεκριμένη διάσταση

- query_driver: Αξιοποίηση των παραπάνω για εκτέλεση οποιουδήποτε ερωτήματος

Στη συνέχεια αναλύουμε κάθε μια απ' αυτές τις συναρτήσεις.

Μέθοδος search

Η μέθοδος αυτή κάνει μια απλή αναζήτηση δυαδικού δέντρου πάνω σε ένα συγκεκριμένο υποδέντρο. Ξεκινώντας από τη ρίζα του υποδέντρου, προχωράει επαναληπτικά είτε στο δεξιό, είτε στο αριστερό παιδί, ανάλογα με την τιμή του value στον εξεταζόμενο κόμβο:

```
while current_node.left is not None:
    if search_value <= current_node.value:
        #Go left
        current_node = current_node.left
    else:
        #Go right
        current_node = current_node.right
```

Η μέθοδος προχωράει μέχρι να φτάσει σε ένα φύλλο, όπου απλά το επιστρέφει:

```
return current_node
```

Η μέθοδος αυτή μάς δίνει επίσης την επιλογή, μέσω της παραμέτρου report, να καταγράφουμε όλα τα υποδέντρα που συναντάμε τα οποία έχουν στα φύλλα τους δεδομένα με value ανάμεσα στο φύλλο και τη ρίζα:

- Αν το φύλλο είναι αριστερά της ρίζας, πρέπει κάθε φορά που η αναζήτηση μάς αναγκάζει να πάμε αριστερά να κάνουμε report το δεξιό υποδέντρο (τη ρίζα του), αφού βρίσκεται στο σύνολο απάντησης. Έτσι η παράμετρος είναι report="right"

- Αν το φύλλο είναι δεξιά της ρίζας, πρέπει κάθε φορά που η αναζήτηση μάς αναγκάζει να πάμε δεξιά να κάνουμε report το αριστερό υποδέντρο (τη ρίζα του), αφού βρίσκεται στο σύνολο απάντησης. Έτσι η παράμετρος είναι report="left"

Άρα ο πλήρης κώδικας στην πραγματικότητα είναι:

```
while current_node.left is not None:

    if search_value <= current_node.value:
        if report == "right":
            path.insert(0, current_node.right)

        current_node = current_node.left

    else:
        if report == "left":
            path.append(current_node.left)

        current_node = current_node.right
```

Αν έχουμε επιλέξει να γίνεται report των ενδιάμεσων υποδέντρων, επιστρέφεται μαζί με τον τελικό κόμβο και μια λίστα με όλα τα υποδέντρα που βρήκαμε:

```
return current_node, path
```

Τη δυνατότητα αυτή την αξιοποιούμε στην αναζήτηση εύρους, όπως θα δούμε στη συνέχεια.

Μέθοδος range_search

Η μέθοδος αυτή υλοποιεί την αναζήτηση εύρους πάνω σε ένα δυαδικό δέντρο. Δέχεται δύο τιμές και αναζητά τους κόμβους που βρίσκονται μέσα στο εύρος αυτό. Αν δώσουμε σε οποιοδήποτε από τα δύο άκρα την τιμή None, κάνουμε μονόπλευρη αναζήτηση εύρους.

Η μέθοδος αρχικά κάνει ένα βήμα προεπεξεργασίας. Σε περίπτωση που το αριστερό ή/και το δεξί άκρο είναι None, πρέπει να δώσει σε αυτά την ελάχιστη και τη μέγιστη τιμή στο δέντρο αντίστοιχα:

```
if x1 is None:
    x1 = self.find_smallest().value

if x2 is None:
    x2 = self.find_biggest().value
```

Η find_smallest επιστρέφει την τιμή που βρίσκεται στο αριστερότερο φύλλο του δέντρου (ακολουθώντας συνεχώς το παρακλάδι "<="), ενώ η find_biggest επιστρέφει την τιμή στο δεξιότερο φύλλο (ακολουθώντας συνεχώς το παρακλάδι ">").

Η αναζήτηση εύρους λειτουργεί ξεκινώντας από τη ρίζα δύο παράλληλες αναζητήσεις για τα δύο άκρα. Ως ένα σημείο τα μονοπάτια αναζήτησης ταυτίζονται, μέχρι έναν κόμβο όπου διακλαδίζονται. Για αρχή, αναζητάμε αυτόν τον κόμβο διακλάδωσης. Ο κόμβος αυτός επιστρέφεται από τη μέθοδο find_split_node:

```
split_node = self.root

while split_node.left is not None: #When both paths are common and
we reach a leaf
    if (x1 <= split_node.value) and (x2 <= split_node.value):
        split_node = split_node.left

    elif (x1 > split_node.value) and (x2 > split_node.value):
        split_node = split_node.right

    else:
        break

return split_node
```

Με ρίζα τον κόμβο διαχωρισμού ξεκινάμε δύο ξεχωριστές αναζητήσεις προς τα άκρα. Εδώ αξιοποιούμε τη δυνατότητα που μας προσφέρει η μέθοδος search να κάνουμε report τα ενδιάμεσα υποδέντρα. Για την αναζήτηση του αριστερού άκρου κάνουμε report τα δεξιά υποδέντρα, ενώ για το δεξιό άκρο κάνουμε report τα αριστερά υποδέντρα. Οι δύο αναζητήσεις μάς επιστρέφουν τους κόμβους όπου καταλήγουν, καθώς επίσης και τα υποδέντρα που συνάντησαν:

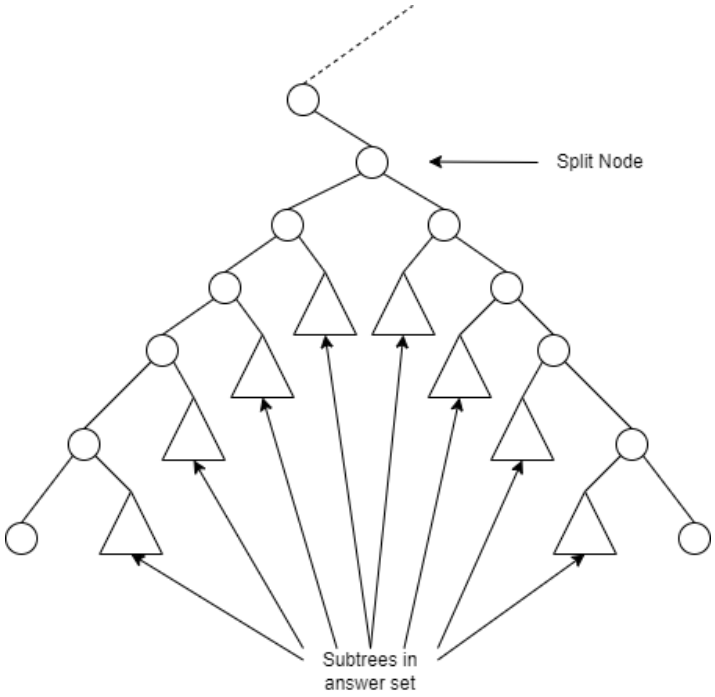
```
if split_node.left is None: #split_node is a leaf, because the
search paths are common
    s1 = split_node
    s2 = split_node
else:
    s1, p1 = self.search(x1, split_node.left, "right")
    s2, p2 = self.search(x2, split_node.right, "left")
```

Όλα τα υποδέντρα στις λίστες p1, p2 ανήκουν σίγουρα στο σύνολο απάντησης. Πρέπει επιπλέον να ελέγξουμε αν και τα s1, s2 ανήκουν στο ζητούμενο εύρος. Επιστρέφουμε τελικά μια συγχώνευση των p1, p2, s1, s2:

```
if s1.value >= x1 and s1.value <= x2:
    result.append(s1)

result = result + p1 + p2

if s1 != s2 and s2.value <= x2 and s2.value >= x1:
    result.append(s2)

return result
```

Να σημειωθεί ότι αυτό που επιστρέφουμε τελικά είναι τα υποδέντρα που περιέχουν τους κόμβους απάντησης, και όχι τους ίδιους τους κόμβους (ακόμα και τα φύλλα μπορούν να θεωρηθούν τετριμμένα υποδέντρα). Θα δούμε στη συνέχεια πώς παίρνουμε την τελική απάντηση.

Παρακάτω φαίνεται σχηματικά η διαδικασία αναζήτησης που περιγράψαμε. Είναι ξεκάθαρο ποια υποδέντρα χρειάζεται να κάνουμε report, αφού οι κόμβοι στα φύλλα είναι ταξινομημένοι από αριστερά προς τα δεξιά, άρα τα υποδέντρα αυτά θα περιέχουν τα φύλλα που θέλουμε:



Μέθοδος query_driver

Οι προηγούμενες μέθοδοι δεν μπορούν να χρησιμοποιηθούν για την εκτέλεση ερωτημάτων και στις 3 διαστάσεις. Και οι δύο λειτουργούν πάνω σε απλά δυαδικά δέντρα. Η μέθοδος query_driver μάς επιτρέπει να εκτελούμε τα πολυδιάστατα ερωτήματα που θέλουμε, παρέχοντας ένα κατάλληλο interface και συνδυάζοντας τι δύο παραπάνω μεθόδους.

Η μέθοδος δέχεται 3 παραμέτρους, μια για κάθε διάσταση:

```
query_driver(name, awards, dblp)
```

Το είδος κάθε μιας απ' αυτές τις παραμέτρους καθορίζει το ερώτημα που εκτελούμε ως προς τη συγκεκριμένη διάσταση. Συνδυάζοντας διάφορες μορφές παραμέτρων μπορούμε να εκτελέσουμε ένα οποιοδήποτε ερώτημα. Οι επιλογές μας φαίνονται παρακάτω:

- Απλή τιμή: Αναζήτηση ακριβούς τιμής
    - Υπάρχει μια ειδική περίπτωση για τα αλφαριθμητικά που περιέχουν έναν μόνο χαρακτήρα. Τέτοια είσοδος σημαίνει ότι θέλουμε να αναζητήσουμε όλους τους επιστήμονες με επίθετο που ξεκινάει από το γράμμα αυτό, αντί να ταιριάζει ολόκληρο το όνομα.
- Λίστα της μορφής [x, y]: Αναζήτηση μέσα στο εύρος [x, y]
    - Υπάρχει μια ειδική περίπτωση για τις λίστες αλφαριθμητικών όπου και το x και το y περιέχουν έναν μόνο χαρακτήρα. Τέτοια είσοδος σημαίνει ότι θέλουμε να αναζητήσουμε όλους τους επιστήμονες με επίθετο που ξεκινάει από γράμμα μεταξύ του x και του y (συμπεριλαμβανομένων των x και y)
- None: Αγνόησε αυτή τη διάσταση/ Κανένας περιορισμός.
- Λίστα της μορφής [None, x]: Αναζήτηση στο (-∞, x]
    - Ισχύει ό,τι είπαμε και πριν για τα αλφαριθμητικά ενός χαρακτήρα
- Λίστα της μορφής [x, None]: Αναζήτηση στο [x, +∞]
    - Ισχύει ό,τι είπαμε και πριν για τα αλφαριθμητικά ενός χαρακτήρα

Στη συνέχεια εξηγούμε την υλοποίηση της μεθόδου αυτής.

Η βασική ιδέα της αναζήτησης είναι ότι ξεκινάμε από το βασικό δέντρο με dimension = 3, εφαρμόζουμε το ερώτημα που θέλουμε κοιτώντας την πρώτη παράμετρο (name) και για τους κόμβους (υποδέντρα) που επιστρέφονται (οι οποίοι γνωρίζουμε σίγουρα ότι ικανοποιούν ό,τι περιορισμό θέσαμε για το όνομα) εκτελούμε αναδρομικά το επόμενο στη σειρά ερώτημα για τα εναλλακτικά υποδέντρα τους με dimension = 2. Όπως αναφέραμε και πριν, μόνο τα δέντρα με dimension = 1 περιέχουν πραγματικά δεδομένα, άρα αφού εκτελέσουμε το ερώτημα που θέλουμε

και για την τελευταία διάσταση, επιστρέφουμε πίσω λίστες με τα δεδομένα στα φύλλα, οι οποίες συγχωνεύονται καθώς «μαζεύεται» η αναδρομή.

Έτσι, αρχικά η μέθοδος διαβάζει την κατάλληλη παράμετρο, ανάλογα με τη διάσταση του δέντρου:

```
if self.dimension == 3:
    q = q1
elif self.dimension == 2:
    q = q2
elif self.dimension == 1:
    q = q3
```

Ερώτημα ακριβούς τιμής:

```
if type(q) is int or type(q) is str: #Exact match search
```

Παρακάτω φαίνεται η ειδική περίπτωση για τα αλφαριθμητικά μήκους 1. Επειδή η συνάρτηση αναζήτησης που έχουμε υλοποιήσει ψάχνει μόνο με ολόκληρο το όνομα και όχι με το πρώτο γράμμα, η αναζήτηση με το πρώτο γράμμα ουσιαστικά υλοποιείται ως μια αναζήτηση εύρους μεταξύ του χαρακτήρα και του ίδιου χαρακτήρα στον οποίο έχουμε προσθέσει πολλά "z" στο τέλος:

```
if type(q) is str and len(q) == 1:
    local_res = self.range_search(q, q+"zzzzzz")
    if len(local_res) == 0:
        return []
```

Εναλλακτικά, χρησιμοποιούμε απλά τη συνάρτηση search:

```
else: #Normal exact-match search
    local_res = [self.search(q)]
    if local_res[0].value != q:
        return []
```

Ερώτημα εύρους:

```
elif type(q) is list: #Range search
```

Παρακάτω φαίνεται η ειδική περίπτωση για τα αλφαριθμητικά μήκους 1. Όπως και πριν, αρκεί απλά να προσθέσουμε στο άνω όριο (αν υπάρχει) πολλά "z" στο τέλος. Η περίπτωση όπου ένα από τα δύο όρια είναι None καλύπτεται από τη συνάρτηση range_search, όπως αναλύσαμε προηγουμένως:

```
if type(q[1]) is str and len(q[0]) == 1 and len(q[1]) == 1: #Range
search using first letter only
    q[1] += "zzzzzz"

local_res = self.range_search(q[0], q[1])
if len(local_res) == 0:
    return []
```

Τέλος, δείχνουμε και την περίπτωση όπου η τιμή είναι None. Απλά προχωράμε προς το επόμενη διάσταση, εφαρμόζοντας αναδρομικά την query_driver στο εναλλακτικό υποδέντρο. Στην περίπτωση όπου δεν υπάρχουν εναλλακτικά υποδέντρα επειδή είμαστε σε διάσταση 1, απλά επιστρέφουμε τα φύλλα με τη μέθοδο report_leaves, η οποία απλά κάνει μια διαπέραση του δέντρου και ένα convergecast:

```
elif q is None: #Don't care for this dimension
    if self.dimension > 1:
        #print(self.dimension, "DONT CARE")
        return self.root.alt_tree.query_driver(q1, q2, q3)
    else:
        return [dp.id for leaf in self.report_leaves() for dp in
leaf.data]
```

Αφού πάρουμε τα τοπικά αποτελέσματα των αναζητήσεων (σαν υποδέντρα), περιορίζοντας το σύνολο απάντησης σύμφωνα με τις συνθήκες που καθορίσαμε για τη συγκεκριμένη διάσταση, πρέπει τώρα να πάμε σε κάθε ένα απ' αυτά και να εκτελέσουμε αναδρομικά την query_driver για τα εναλλακτικά υποδέντρα. Μόνο για την τελευταία διάσταση αρκεί απλά να επιστρέψουμε τα φύλλα, όπου βρίσκονται τα δεδομένα:

```
for node in local_res:
    if self.dimension > 1:
        next_dim_res = node.alt_tree.query_driver(q1, q2, q3)
        #if next_dim_res is not None:
        res = res + next_dim_res
    else:
        res = res + [dp.id for leaf in self.report_leaves(node)
for dp in leaf.data]

return res
```

4.2.5 Ισοζύγιση

Όπως προαναφέραμε, για τα δυαδικά δέντρα χρησιμοποιούμε ισοζύγιση AVL. Στη συνέχεια περιγράφουμε όλο τον κώδικα που υλοποιεί την ισοζύγιση αυτή.

Περιστροφές

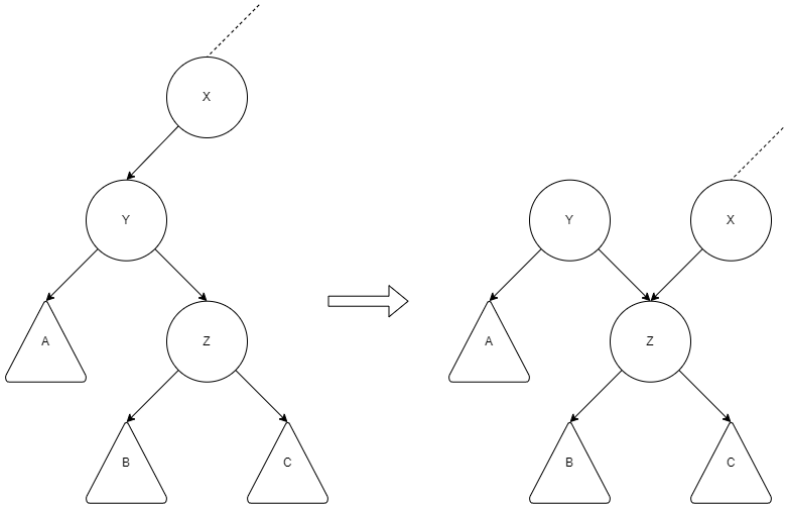
Υπάρχουν 4 είδη περιστροφών:

- Αριστερή περιστροφή (left_rotation)
- Δεξιά περιστροφή (right_rotation)
- Αριστερή-Δεξιά περιστροφή (left_right_rotation)
- Δεξιά-Αριστερή περιστροφή (right_left_rotation)

Οι δύο πρώτες περιστροφές είναι εντελώς συμμετρικές. Αρκεί να αναλύσουμε την αριστερή περιστροφή. Η περιστροφή εμπλέκει τρεις κόμβους x,y,z όπου z αριστερό παιδί (δεξιό στη δεξιά περιστροφή) του y και y παιδί του x. Η περιστροφή γίνεται στον κόμβο y και αποτελείται από 3 βήματα.

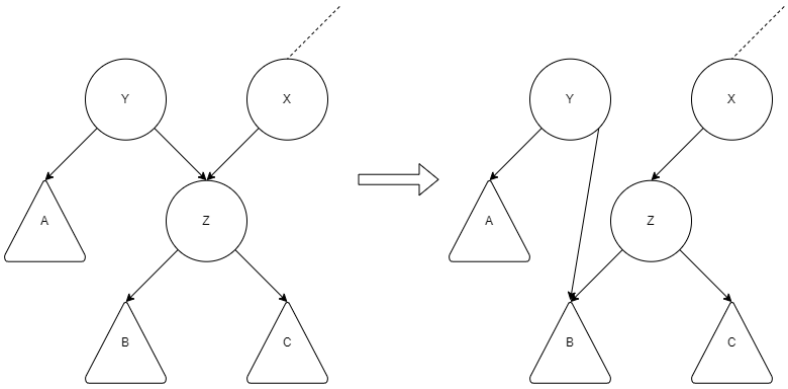
Στο πρώτο βήμα κάνουμε το z να είναι τώρα παιδί του x:

```
#Step 1: The parent needs to have the child node as its new child
if parent is not None:
    if node == parent.left:
        parent.left = child
    elif node == parent.right:
        parent.right = child

    child.parent = parent

else: #node is the root
    child.parent = None
    self.root = child
```



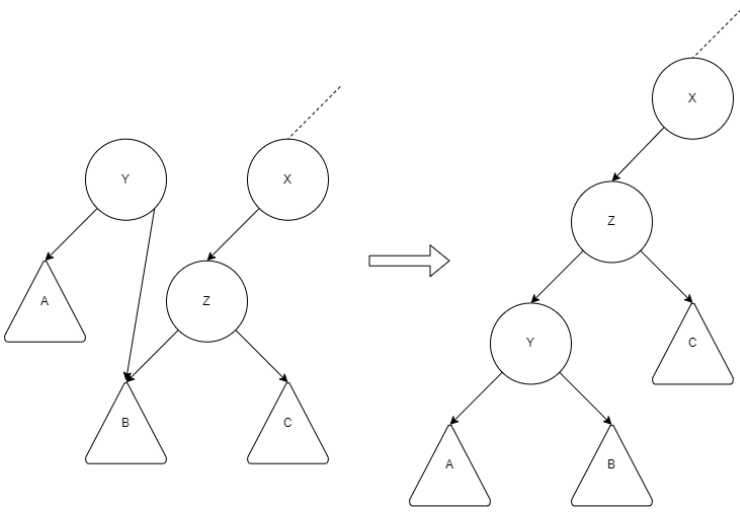
Στο δεύτερο βήμα παίρνουμε το αριστερό παιδί του z και το κάνουμε δεξιό παιδί του y:

```
#Step 2: Original node needs to get child node's left subtree as
its new right subtree
node.right = child.left
child.left.parent = node
```



Στο τρίτο βήμα κάνουμε τον κόμβο z πατέρα του y:

```
#Step 3: Child becomes the parent
child.left = node
node.parent = child
```



Αφού γίνει η περιστροφή, πρέπει τώρα να ανανεώσουμε τα ύψη και τις ισοζυγίσεις των y και z, αφού αλλάξαμε τα παιδιά τους. Πρέπει πρώτα να ξεκινήσουμε από τον χαμηλότερο κόμβο (τώρα τον y) και μετά να εξετάσουμε και τον ψηλότερο (τώρα τον z):

```
#Update heights and hb starting from the lowest node (node, then
child)
node.height = max(node.left.height, node.right.height) + 1
node.hb = node.right.height - node.left.height

child.height = max(child.left.height, child.right.height) + 1
child.hb = child.right.height - child.left.height
```

Η μέθοδος τελικά επιστρέφει τον κόμβο z ο οποίος πήρε τη θέση του y στον οποίο κάναμε την περιστροφή:

```
return child
```

Τώρα που είδαμε τις απλές περιστροφές, οι σύνθετες περιστροφές υλοποιούνται ως σύνθεση των απλών. Είναι επίσης συμμετρικές, άρα αρκεί να εξετάσουμε την left_right_rotation.

Η περιστροφή αυτή πρώτα εφαρμόζει μια δεξιά περιστροφή στο δεξιό παιδί του κόμβου:

```
self.right_rotation(node.right)
```

Έπειτα, εφαρμόζει αριστερή περιστροφή στον ίδιο τον κόμβο:

```
self.left_rotation(node)
```

Μέθοδος balance

Η μέθοδος balance ξεκινάει από το φύλλο όπου έγινε η ένθεση και επαναϋπολογίζει τα ύψη και τις ισοζυγίσεις, κάνοντας και τις απαραίτητες περιστροφές μέχρι να φτάσουμε στη ρίζα ή να γίνει μια περιστροφή, οι οποίες στην περίπτωση των ενθέσεων είναι πάντα τερματικές.

Σαν όρισμα δίνουμε τον πατέρα του νέου φύλλου που προσθέσαμε στο δέντρο. Έτσι, σαν πρώτο βήμα, αρχικοποιούμε το ύψος και την ισοζύγιση του κόμβου αυτού:

```
node.height = 1
node.hb
```

Έπειτα διασχίζουμε το δέντρο προς τα πάνω επαναληπτικά, κρατώντας κάθε φορά δύο κόμβους, τον τρέχοντα και τον προηγούμενο. Αρχικά θέτουμε το όρισμα σαν προηγούμενο και τον γονέα του σαν τρέχοντα:

```
current_node = node.parent
previous_node = node
```

Για κάθε κόμβο που εξετάζουμε, υπολογίζουμε ύψος και ισοζύγιση:

```
current_node.height = max(current_node.left.height, current_node.right.height) + 1
current_node.hb = current_node.right.height - current_node.left.height
```

Αφού τα υπολογίσουμε, πρέπει να δούμε ποια είναι η ισοζύγιση του δέντρου και να εφαρμόσουμε την κατάλληλη περιστροφή . Υπάρχουν 5 περιπτώσεις:

- Αν ο τρέχοντας κόμβος είναι πλήθος ζυγισμένος (hb = 0), γίνεται απορρόφηση και δε χρειάζεται να συμβεί καμία περιστροφή

- Αν current_node.hb = +2 και previous_node.hb = +1, σημαίνει πως ήρθαμε δύο φορές από τα δεξιά και πρέπει να γίνει αριστερή περιστροφή στον current_node

- Αν current_node.hb = +2 και previous_node.hb = -1, σημαίνει πως ήρθαμε πρώτα από τα αριστερά και μετά από τα δεξιά και πρέπει να γίνει αριστερή-δεξιά περιστροφή στον current_node

- Αν current_node.hb = -2 και previous_node.hb = -1, σημαίνει πως ήρθαμε δύο φορές από τα αριστερά και πρέπει να γίνει δεξιά περιστροφή στον current_node

- Αν current_node.hb = -2 και previous_node.hb = +1, σημαίνει πως ήρθαμε πρώτα από τα δεξιά και μετά από τα αριστερά και πρέπει να γίνει δεξιά-αριστερή περιστροφή στον current_node

5. R-Tree

5.1 Περιγραφή κώδικα:

Αρχικά, η συνάρτηση letter_to_int(str1) παίρνει ως είσοδο ένα γράμμα και επιστρέφει το αριθμό της σειράς που εμφανίζεται στην αγγλική αλφαβήτα.

Η συνάρτηση str_to_int(word) παίρνει ως είσοδο μία λέξη και επιστρέφει έναν αριθμό που αντιστοιχεί σε αυτήν την λέξη σύμφωνα με τους χαρακτήρες της και την θέση των χαρακτήρων αυτών μέσα στη λέξη. Χρησιμοποιεί την συναρτηση letter_to_int(str1) για να πάρει την τιμή που αντιστοιχεί σε κάθε χαρακτήρα και προσθέτει στο αποτέλεσμα την τιμή του αριθμού επί 26 εις την θέση που έχει ο αριθμός στην λέξη. Οι θέσεις ξεκινάνε από το δεξιότερο ψηφίο που αντιστοιχεί στο 0.

Ορίζουμε την κλάση Node η οποία χρησιμοποιείται για τον ορισμό τόσο των σημείων όσο και των ορθογωνίων παραλληλεπιπέδων. Κατά την αρχικοποίηση ορίζονται οι συντεταγμένες κάθε σημείου ή ορθογωνίου και το πεδίο id αν συμπληρωθεί τότε πρόκειται για σημείο ενώ διαφορετικά πρόκειται για ορθογώνιο για το οποίο ορίζεται μία κενή λίστα με τα παιδιά του που είναι αντικείμενα τύπου node. Η κλάση node έχει δύο μεθόδους για που επιστρέφουν ένα σημείο η κάθε μία, το μεγαλύτερο ή το μικρότερο (σχετικά με την απόσταση τους από το (0,0,0)). Για τα σημεία οι συναρτήσεις αυτές επιστρέφουν τις ίδιες συντεταγμένες.

Η συνάρτηση get_nodelist(jdata) παίρνει ως όρισμα τα δεδομένα του out2.json και επιστρέφει μία λίστα με κόμβους οι οποίοι αντιπροσωπεύουν σημεία αφού έχουν ο καθένας το δικό του id. Επίσης, επιστρέφει τις μέγιστες συντεταγμένες της κάθε διάστασης με τις οποίες έχει κανονικοποιήσει τις συντεταγμένες των σημείων.

Η συνάρτηση nearest(xyz, nodelist) παίρνει ως ορίσματα τις συντεταγμένες ενός σημείου και επιστρέφει την θέση του σημείου/κόμβου της λίστας nodelist που βρίσκεται πιο κοντά σε αυτό.

Η συνάρτηση makembr(nodelist) παίρνει ως όρισμα μία λίστα με κόμβους και επιστρέφει το μικρότερο ορθογώνιο (MBR) που περικλείει όλους τους κόμβους. Το MBR είναι ένας κόμβος με παιδιά όλους τους κόμβους της nodelist και κατάλληλες συντεταγμένες τέτοιες ώστε να περικλείει όλους τους κόμβους.

Η συνάρτηση build(nodelist, K) παίρνει ως όρισμα μία λίστα με κόμβους και επιστρέφει μια λίστα με μοναδικό στοιχείο την ρίζα ενός r-tree. Η συνάρτηση αυτή λειτουργεί αναδρομικά ξεκινώντας με τους κόμβους-σημεία και ομαδοποιώντας τους σε MBRs και ύστερα ομαδοποιεί τα MBRs σε

μεγαλύτερα MBRs μέχρι να φτάσει στη ρίζα την οποία επιστρέφει. Αναλυτικότερα, στην αρχή αυτής της συνάρτησης ορίζεται μία λίστα με κόμβους οι οποίοι πρέπει να μπουν σε MBRs nodes_for_mbr και μία κενή λίστα η οποία θα έχει τα τελικά MBRs list_with_mbrs. Μετά, γίνεται έλεγχος (**) για το αν η λίστα που δίνουμε ως όρισμα είναι κενή και αν είναι τότε επιστρέφεται η λίστα που δίνουμε ως όρισμα. Ύστερα, ορίζουμε μία λίστα local_list που θα βοηθήσει στην κατασκευή κάθε MBR αποθηκεύοντας προσωρινά τους κόμβους του μέχρι να φτιαχτεί το MBR και να προστεθεί στην τελική λίστα list_with_mbrs. Έπειτα, με βρόγχο while όσο η λίστα nodes_for_mbr έχει τουλάχιστον 1 στοιχείο το οποίο πρέπει να μπει σε MBR γίνεται το εξής: αν η λίστα local_list έχει K-1 κόμβους ή έχει μείνει μόνο ένας κόμβος στην λίστα nodes_for_mbr τότε μεταφέρουμε τον κόμβο της λίστας nodes_for_mbr που έχει την μικρότερη απόσταση από το σημείο start (το οποίο έχει είτε τις συντεταγμένες 0,0,0 είτε αυτές του τελευταίου στοιχείου που προσθέσαμε στην local_list) στην λίστα local_list. Ακολούθως, κατασκευάζουμε ένα MBR με τους κόμβους της local_list και το προσθέτουμε στην λίστα list_with_mbrs. Επίσης, επαναφέρουμε την μεταβλητη start στο 0,0,0 και κάνουμε την local_list κενή. Αν ο έλεγχος (**) αποτύχει τότε βρίσκουμε τον κόμβο που βρίσκεται πιο κοντά στο start και τον προσθέτουμε στην local_list. Επιπλέον, κάνουμε το start ίσο με τις ελάχιστες συντεταγμένες του κόμβου που προσθέσαμε. Τέλος, καλούμε αναδρομικά την build(list_with_mbrs, K) με την λίστα με τα MBRs που φτιάχτηκε ώστε να ομαδοποιηθούν περαιτέρω.

Η συνάρτηση mbr_intersects_mbr (big_mbr, small_mbr) επιστρέφει 1 αν ο κόμβος big_mbr έχει κοινά σημεία με τον κόμβο small_mbr και 0 αλλιώς.

Η συνάρτηση get_results(root,minx=0,miny=0,minz=0,maxx=1,maxy=1,maxz=1) ορίζει τη συνάρτηση search(root,minx,miny,minz,maxx,maxy,maxz) η οποία είναι αναδρομική και βάζει στην λίστα results όλα τα αποτελέσματα που βρίσκονται στο range που δίνεται από τα ορίσματα. Αναλυτικότερα, αυτή η search() ψάχνει στους κόμβους της λίστας root και αν αυτός ο κόμβος έχει id, δηλαδή είναι σημείο, τότε ελέγχει αν οι συντεταγμένες του βρίσκονται μέσα στο range και αν ναι τότε τον προσθέτει στην λίστα results. Αν ο κόμβος της λίστας root δεν έχει id, δηλαδή είναι ορθογώνιο τότε ελέγχει αν έχει κοινά σημεία με το ζητούμενο range και αν ναι τότε ψάχνει αναδρομικά στη λίστα των παιδιών του. Τέλος, η get_results() καλεί την search() και επιστρέφει την λίστα results.

Η συνάρτηση printdata(data, root, show_points=0, show_rects=2) δημιουργεί γραφικά τα σημεία και τα ορθογώνια παραλληλεπίπεδα.

Η κλάση Rtree χρησιμοποιείται για την ομαδοποίηση παραπάνω συναρτήσεων και την παραμετροποίηση του K και των δεδομένων εισόδου.

6. Locality Sensitive Hashing (LSH)

6.1 Περιγραφή κώδικα:

Η συνάρτηση find_b(C, t) βρίσκει όλα τα δυνατά b που διαιρούν το C και επιλέγει αυτό που προσεγγίζει καλύτερα το threshold t που θέλουμε να πετύχουμε.

Η συνάρτηση plot_t(t, b, C) σχεδιάζει το ζητούμενο threshold και την προσέγγισή του.

Η συνάρτηση shingling(string, k) παίρνει ως όρισμα ένα αλφαριθμητικό και το χωρίζει σε επικαλυπτόμενα κομμάτια των k χαρακτήρων τα οποία ενθέτει σε μία λίστα την οποία επιστρέφει.

Η συνάρτηση shingle_data(data, k) παίρνει ως όρισμα μία λίστα με τα αλφαριθμητικά εισόδου και επιστρέφει μια λίστα με λίστες που έχουν τα επικαλυπτόμενα κομμάτια k χαρακτήρων των αλφαριθμητικών.

Η συνάρτηση get_shingles(shingled_data) παίρνει ως όρισμα μία λίστα που περιέχει τις λίστες με τα κομμάτια των αλφαριθμητικών και επιστρέφει μία λίστα με όλα τα κομμάτια που εμφανίζονται τουλάχιστον σε μία λίστα από αυτές του ορίσματος.

Η συνάρτηση make_table(all_shingles, shingled_data) επιστρέφει μία λίστα με λίστες. Κάθε λίστα αντιστοιχεί σε ένα αλφαριθμητικό των δεδομένων και είναι σαν διανύσματα που έχουν 0 ή 1 στην κατάλληλη θέση ανάλογα με το αν το αντίστοιχο αλφαριθμητικό περιέχει το αντίστοιχο κομμάτι k χαρακτήρων ή όχι.

Η συνάρτηση get_shingle_table(data, k) είναι βοηθητική και εκτελεί την όλη τη διαδικασία από το κομμάτιασμα των αλφαριθμητικών εισόδου μέχρι την δημιουργία του διανυσμάτων με τα 0 ή 1 κάθε αλφαριθμητικού.

Η συνάρτηση h(seed, num, t_len) είναι ομάδα συναρτήσεων κατακερματισμού (επιλέγουμε μία ανάλογα με το seed που δώσουμε) και επιστρέφει έναν αριθμό που ανήκει στο [0, t_len-1] ο οποίος αντιπροσωπεύει μια συγκεκριμένη μετάθεση της στήλης num.

Η συνάρτηση h2(seed, buckets) είναι μια ομάδα «τυχαίων» συναρτήσεων (επιλέγουμε μία ανάλογα με το seed που δώσουμε) και επιστρέφει έναν τυχαίο αριθμό που ανήκει στο [0, buckets-1]. Η

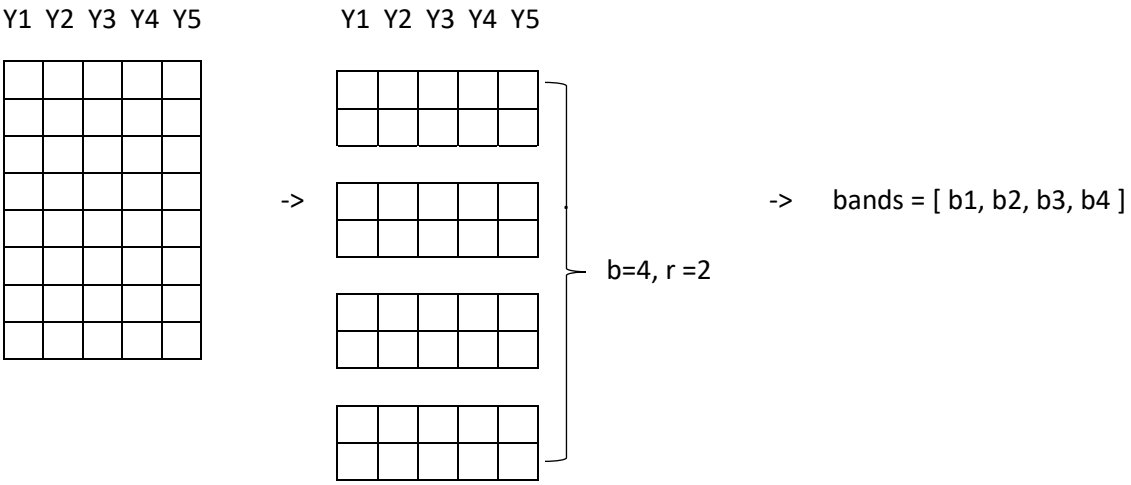
συνάρτηση αυτή χρησιμοποιείται για να τοποθετηθούν με τυχαίο τρόπο τα κομμάτια των υπογραφών μεγέθους r σε έναν από τους B κουβάδες.

Η συνάρτηση minhash(table, C) παίρνει ως όρισμα μια λίστα τα διανύσματα με τα 0 ή 1 των αλφαριθμητικών και επιστρέφει τις υπογραφές των αλφαριθμητικών αυτών που έχουν η καθεμία μέγεθος C το οποίο είναι μικρότερο από το μέγεθος του διανύσματος. Αυτή η συνάρτηση, αρχικά, δημιουργεί μία λίστα με λίστες, τόσες όσα και τα αλφαριθμητικά, μεγέθους C οι οποίες έχουν ως τιμή το μέγεθος των διανυσμάτων των αλφαριθμητικών + 1. Μετά, σε κάθε θέση της κάθε υπογραφής τοποθετείται η θέση του πρώτου 1 στο διάνυσμα σύμφωνα με την συγκεκριμένη μετάθεση που έχουμε κάνει στο συγκεκριμένο αλφαριθμητικό. Τέλος, επιστρέφεται η λίστα με τις υπογραφές.

Η συνάρτηση compare_sig(sig1, sig2) επιστρέφει έναν αριθμό που ανήκει στο [0, 1] και αντιπροσωπεύει το ποσοστό ομοιότητας των δύο υπογραφών που δίνονται ως όρισμα.

Η συνάρτηση sig2int(sig) επιστρέφει έναν αριθμό ο οποίος προκύπτει από την υπογραφή ενθέτοντας τα επιμέρους αποτελέσματα των συναρτήσεων κατακερματισμού το ένα μετά το άλλο. Π.χ. αν η υπογραφή είναι sig=[2,4,2,3] τότε θα επιστραφεί ο αριθμός 2423.

Η συνάρτηση lsh(sig, C, b, B, threshold) παίρνει ως ορίσματα έναν πίνακα με τις υπογραφές, το μήκος των υπογραφών, τον αριθμό των ζωνών που θα χρησιμοποιήσουμε, τον αριθμό των κουβάδων που θα χρησιμοποιήσουμε και το threshold ομοιότητας το οποίο θέλουμε να πετύχουμε. Αρχικά, η συνάρτηση αυτή χωρίζει τις υπογραφές σε b κομμάτια και προκύπτουν μικρότερες υπογραφές μήκους r οι οποίες αποθηκεύονται στον πίνακα bands.



Ύστερα, για κάθε band:

κατακερματίζει όλα τα κομμάτια των υπογραφών σε B κουβάδες και μετά για κάθε κουβά με πάνω από 2 κομμάτια υπογραφών ελέγχει ανά δύο αν η ομοιότητα των αρχικών υπογραφών είναι πάνω

από το όριο ομοιότητας που θέλουμε και αυτές που είναι τις τοποθετεί ως ομάδα στον πίνακα groups.

Τέλος, επιστρέφει ανά ομάδες τις όμοιες υπογραφές. Πιο συγκεκριμένα, αντί για την κάθε υπογραφή, επιστρέφει την θέση στον αρχικό πίνακα που είχε το αλφαριθμητικό από το οποίο προήλθε.

Η συνάρτηση get_unique_groups(groups) επιστρέφει τις μοναδικές όμοιες ομάδες, έτσι ώστε να αφαιρεθούν οι ομάδες με ίδια στοιχεία που υπάρχουν πάνω από μία φορά.

Η συνάρτηση shing_minhash_lsh(data, k, C, B, threshold, plot_threshold=1) συνθέτει όλες τις παραπάνω συναρτήσεις ώστε από τα αλφαριθμητικά εισόδου να πάρει τα διανύσματα με τα κομμάτια του κάθε αλφαριθμητικού, από αυτά να πάρει την υπογραφή κάθε αλφαριθμητικού, να υπολογίσει τον αριθμό των ζωνών για να προσεγγίζεται το threshold, να βρει τις όμοιες ομάδες από τις υπογραφές και να βρει τις μοναδικές όμοιες ομάδες. Επίσης, μπορεί να σχεδιάσει γραφικά την προσέγγιση του threshold και το threshold για να δούμε την απόκλιση.

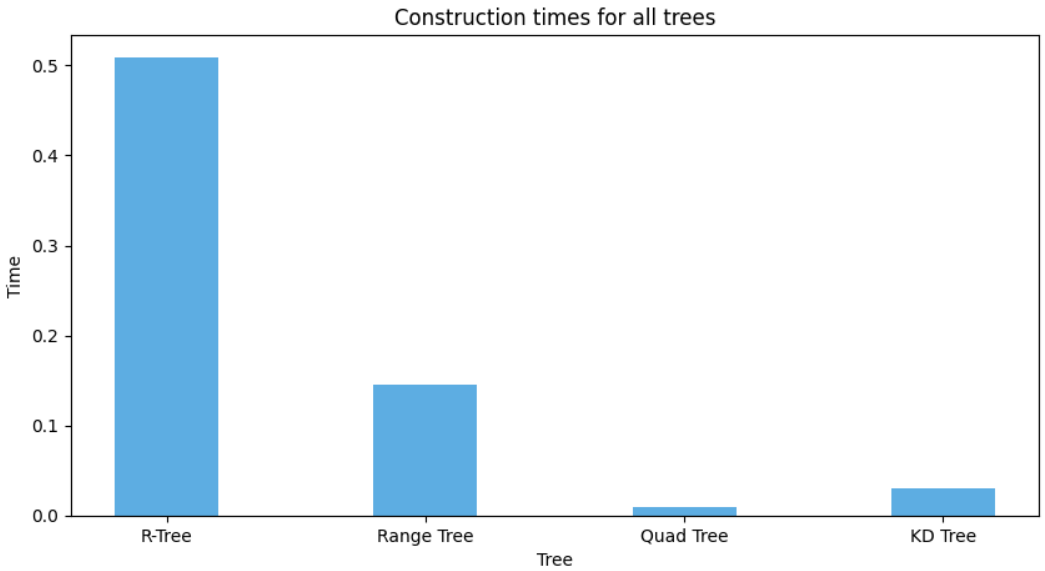
7. Evaluation and Comparison

7.1 Σύγκριση χρόνου δημιουργίας των δέντρων:

Παρατηρούμε ότι οι χρόνοι δημιουργίας των δέντρων έχουν συνήθως την εξής σειρά:

Quad tree < KD tree < Range tree < R tree.

```
Construction times
==========================================
> R-Tree: 0.5081023

> Range Tree: 0.14495710000000006

> Quad Tree: 0.010163700000000109

> KD Tree: 0.031146899999999977
```

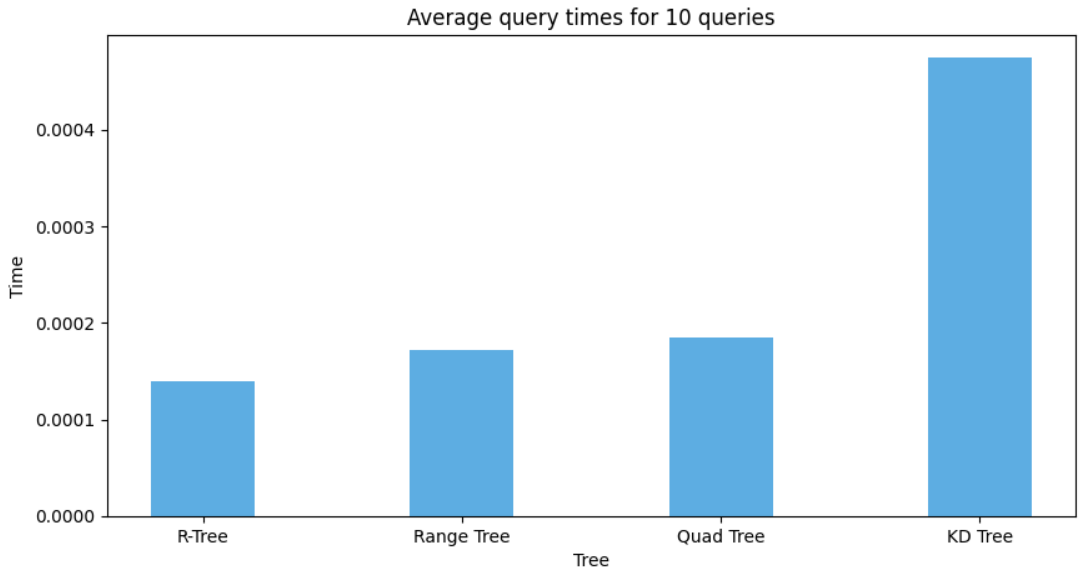


7.2 Σύγκριση χρόνου ερωτημάτων μεταξύ των δέντρων:

Παρατηρούμε ότι οι χρόνοι ερωτημάτων των δέντρων έχουν συνήθως την εξής σειρά:

R tree < Range tree < Quad tree < KD tree.

```
Average query times for 10 queries
==========================================
> R-Tree: 0.00013960000000000175

> Range Tree: 0.00017177999999997695

> Quad Tree: 0.00018538000000005715

> KD Tree: 0.00047416999999998517
```



7.3 Σύγκριση χρόνου ερωτημάτων στα δέντρα με τον χρόνο εκτέλεσης του LSH:

Παρατηρούμε ότι το LSH είναι πολύ πιο χρονοβόρο από τα ερωτήματα στα δέντρα.

Query	R-Tree	Range Tree	Quad Tree	KD Tree	# results	LSH	n1	n2	a1	d1	d2
0	0.000116	0.000163	0.000102	0.000246	4	0.026339	m	w	1	386	463
1	0.000146	0.000174	0.000101	0.000288	1	0.015164	a	u	2	345	358
2	0.000177	0.000221	0.000253	0.000595	53	0.595239	n	u	0	105	341
3	6.77E-05	5.38E-05	6.16E-05	7.19E-05	1	0.033475	w	z	8	142	219
4	0.000259	0.000376	0.000523	0.001185	167	2.962242	d	p	1	1	498
5	0.00012	0.000142	0.000111	0.00047	21	0.256276	a	t	9	30	299
6	7.99E-05	3.9E-05	3.96E-05	0.000166	0	0	r	s	12	42	223
7	4.04E-05	3.14E-05	3.1E-05	6.16E-05	1	0.032944	x	z	7	112	217
8	0.000266	0.000372	0.000515	0.001186	167	3.026381	d	p	1	1	498
9	0.000125	0.000145	0.000117	0.000472	21	0.260102	a	t	9	30	299

```
Average query times for 10 queries
========================================
> R-Tree: 0.00013960000000000175

> Range Tree: 0.00017177999999997695

> Quad Tree: 0.00018538000000005715

> KD Tree: 0.0004741699999998517


Average LSH time: 0.8009067444444443
```