

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΑΤΡΩΝ - 2023

ΠΟΛΥΔΙΑΣΤΑΤΕΣ ΔΟΜΕΣ ΔΕΔΟΜΕΝΩΝ

IOANNIS LOUDAROS - CHRISTINA KRATIMENOU - PARIS PSERGIANNIS - ALEXIS LEKARAKOS

Γενικές Πληροφορίες



Για να έχετε πρόσβαση στην τελευταία έκδοση των απαντήσεων μπορείτε να σκανάρετε το παραπάνω QR Code ή να χρησιμοποιήσετε το παρακάτω κουμπί.

Πατήστε Εδώ

Στις επόμενες σελίδες αναλύονται οι απαντήσεις της ομάδας μας στο Project του μαθήματος “Πολυδιάστατες Δομές Δεδομένων”. Σε αυτή την σελίδα έχετε πρόσβαση σε γενικές πληροφορίες γύρω από το Project αλλά και γύρω από τα μέλη της ομάδας μας.

Για την υλοποίηση του Project εργαστήκαμε σε μια ομάδα 4 ατόμων. Χωρίσαμε τα ερωτήματα του πρότζεκτ ώστε ο φόρτος εργασίας να καταμεριστεί. Όλοι είχαμε επίγνωση των εργασιών που πραγματοποιούν οι συνεργάτες μας και δίναμε feedback ο ένας στον άλλον με αποτέλεσμα να προχωράμε συνεκτικά. Η ομάδα αποτελείται από τα εξής άτομα:

Αλέξης Λεκαράκος (1069367)

Ιωάννης Λουδάρος (1067400)

Χριστίνα Κρατημένου (1067495)

Πάρης Σεργιάννης (1067467)

Παρακάτω υπάρχουν αναλυτικότερες πληροφορίες για τα μέλη της ομάδας μας.



Χριστίνα Κρατημένου
1067495

up1067495@upnet.gr
Φοιτήτρια 5ου έτους



Πάρης Σεργιάννης
1067467

up1067467@upnet.gr
Φοιτητής 5ου έτους



Ιωάννης Λουδάρος
1067400

iloudaros@upnet.gr
Φοιτητής 5ου έτους



Αλέξης Λεκαράκος
1069367

st1069367@ceid.upatras.gr
Φοιτητής 5ου έτους



Περιεχόμενα

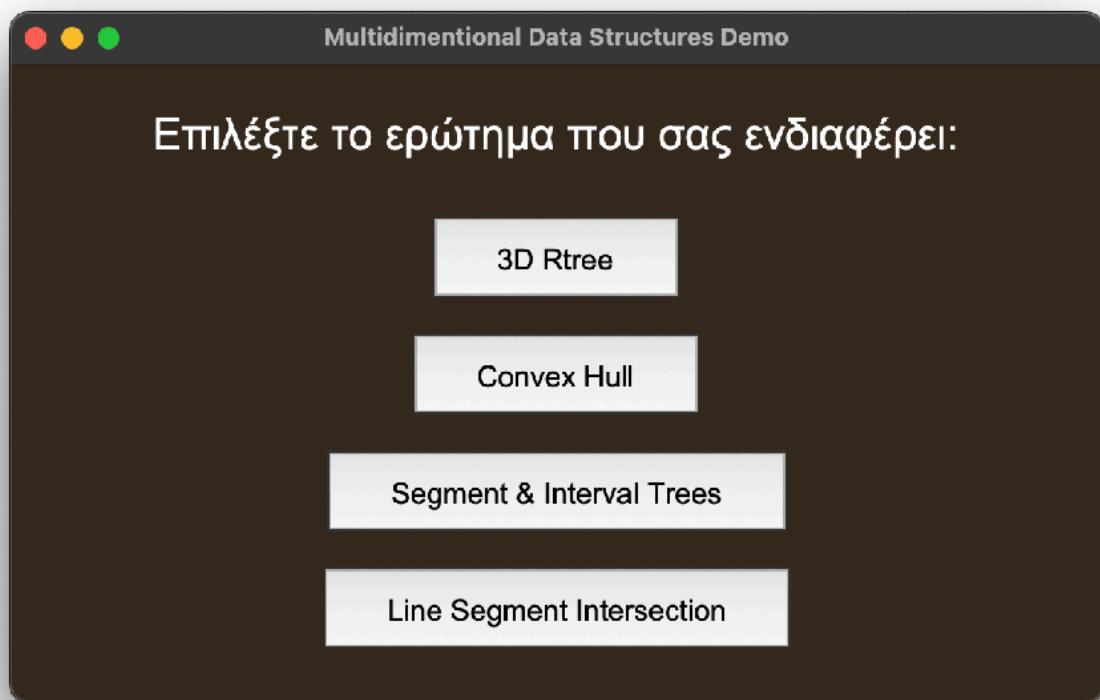
Απαντήσεις	1
3D R-trees for Spatio-Temporal Queries σε ΒΔ τροχιών στο επίπεδο	2
Interval trees και Segment trees	7
Convex Hull	17
Line Segment Intersection	18
Παράρτημα Α	20
Ορισμός του Point	20
Ορισμός του Entry	20
Ορισμός του Box	21
Ορισμός του Node	22
Παράρτημα Β.....	23
Interval Trees	23
Segment Tree	26
Παράρτημα Γ	28
Παραγωγή και Οπτικοποίηση του Convex Hull	28
Παράρτημα Δ.....	29
Κλάσεις που Δημιουργήθηκαν	29
Οι Αλγόριθμοι	30

Απαντήσεις

Σύντομη παρουσίαση Πρότζεκτ

Για την διευκόλυνση σας, έχει προετοιμαστεί ένα αρχείο python που παρουσιάζει γρήγορα το κάθε ερώτημα.

Μέσα στον κατάλογο contributions, τρέξτε το αρχείο python test.py. Τρέχοντας αυτό το script, μπορείτε μέσω μιας απλής διεπαφής να περιηγηθείτε στα παραδείγματα που έχουμε προετοιμάσει για το κάθε ερώτημα.



Παρακάτω παραθέτουμε screenshot από την εν λόγω διεπαφή.

Περισσότερες πληροφορίες για το κάθε ερώτημα μπορείτε να βρείτε στις επόμενες σελίδες.

3D R-trees for Spatio-Temporal Queries σε ΒΔ τροχιών στο επίπεδο

Μπορείτε να βρείτε τον κώδικα που αντιστοιχεί στο ερώτημα στο Παράρτημα Α.

Εισαγωγή

Παραδοσιακά 2D R-trees χρησιμοποιούν rectangles για να δεικτοδοτήσουν σημεία. Σε 3D R-trees επομένως γίνεται χρήση κύβων δηλ. boxes.

Στα πλαίσια της εργασίας ένα δεικτοδοτούμενο σημείο της μορφής (x,y,t) αναπαριστά ένα σημείο στον τρισδιάστατο χώρο. Επίσης για να δεικτοδοτήσουμε ένα τέτοιο σημείο μέσα στο R-tree υποθέτουμε ότι είναι ένα box με μηδενικό όγκο στις συντεταγμένες x,y,t. Η υλοποίηση ενός σημείου δηλώνεται στο αρχείο point.py.

Boxes

Απαραίτητη είναι και η υλοποίηση των boxes. Ο κώδικας βρίσκεται στο αρχείο box.py

- getVolume => επιστρέφει τον όγκο ενός box
- overlaps => επιστρέφει true ή false ανάλογα αν 2 box επικαλύπτονται ή εμπεριέχεται το ένα στο άλλο. Χρήση για την υποβολή ερωτημάτων στο R-tree

Entries

Κάθε κόμβος του δέντρου περιέχει μια λίστα από entries. Το entry περιέχει τα εξής δεδομένα:

1. Box => Δηλώνει τις συντεταγμένες του box στον χώρο που περιέχει ένα data point ή έναν ολόκληρο κόμβο.
2. Child => Δηλώνει αν το entry αυτό δείχνει σε έναν άλλο κόμβο. Κόμβοι που τα entries τους δείχνουν σε έναν άλλο κόμβο λέγονται ενδιάμεσοι κόμβοι (intermediate nodes).

3. Data => Δηλώνει αν το entry δείχνει σε ένα data point. Αυτοί οι κόμβοι λέγονται leaf nodes ή leaves και περιέχουν τα δεδομένα του δέντρου. Τα leaves δεν δείχνουν ποτέ σε κάποιον άλλο κόμβο και είναι όλα στο ίδιο επίπεδο.

Κάθε κόμβος περιέχει ένα minimum και ένα maximum number of entries.

Η υλοποίηση των entries και nodes περιέχονται στο αρχείο rtree.py

Τέλος οι αλγόριθμοι για insert και query πανω στο R-tree υλοποιούνται όπως περιγράφονται στο paper :

R-TREES : A DYNAMIC INDEX STRUCTURE FOR SPATIAL SEARCHING του A. Guttman, 1984

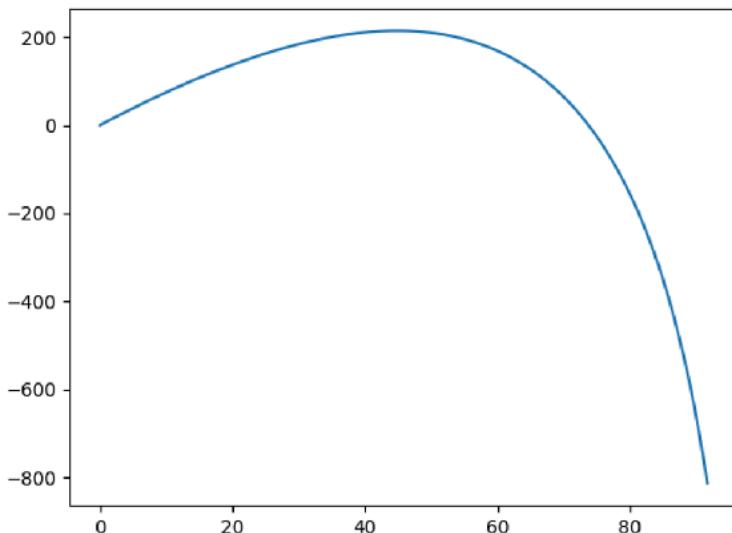
Insert

Για την εισαγωγή νέων δεδομένων στο δέντρο ξεκινάμε από το root. Στην συνέχεια για κάθε child entry υπολογίζουμε πως θα επηρεάσει τον τελικό όγκο του box child μετα την εισαγωγή. Διαλέγουμε κάθε φορά τον κόμβο που θα έχει ως αποτέλεσμα την μικρότερη αύξηση σε όγκο, μέχρι να φτάσουμε σε ενα leaf node και να προσθέσουμε εκεί το data point. Σε περίπτωση ωστόσο που ο leaf node έχει ήδη το μέγιστο αριθμό entries, πρέπει να χωρίσουμε τον κόμβο στα 2 δηλαδή να κάνουμε node split. Ιδανικά το node split θα δημιουργήσει 2 boxes τα οποία θα περιέχουν όλα τα δεδομένα με τον μικρότερο wasted όγκο. Υλοποιούμε τον αλγόριθμο quadratic node split, όπως περιγράφεται στο paper, που δεν εγγυάται το βέλτιστο αποτέλεσμα αλλα ένα πολύ καλό σε γρήγορο χρόνο. Μετά το node split προσαρμόζουμε και το υπόλοιπο δέντρο “προς τα πάνω” με τον ίδιο τρόπο. Οι συναρτήσεις για την υλοποίηση του insert υπάρχουν στο rtree.py.

Δεδομένα για το insert:

(Artificial Synthetic Dataset)

Για τα δεδομένα μας χρειαζόμαστε δεδομένα τροχιάς ενός αντικειμένου. Στο αρχείο data-gen.py γίνεται προσομοίωση ρίψης μίας σφαίρας στον αέρα με δειγματοληψία θέσης x,y και χρόνου t κάθε dt. Η προσομοίωση περιγράφει την τροχιά που ακολουθεί:



και παράγει περίπου 2.500 data points της μορφής (x,y,t) και τα αποθηκεύει στο αρχείο:
trajectory-data.csv.

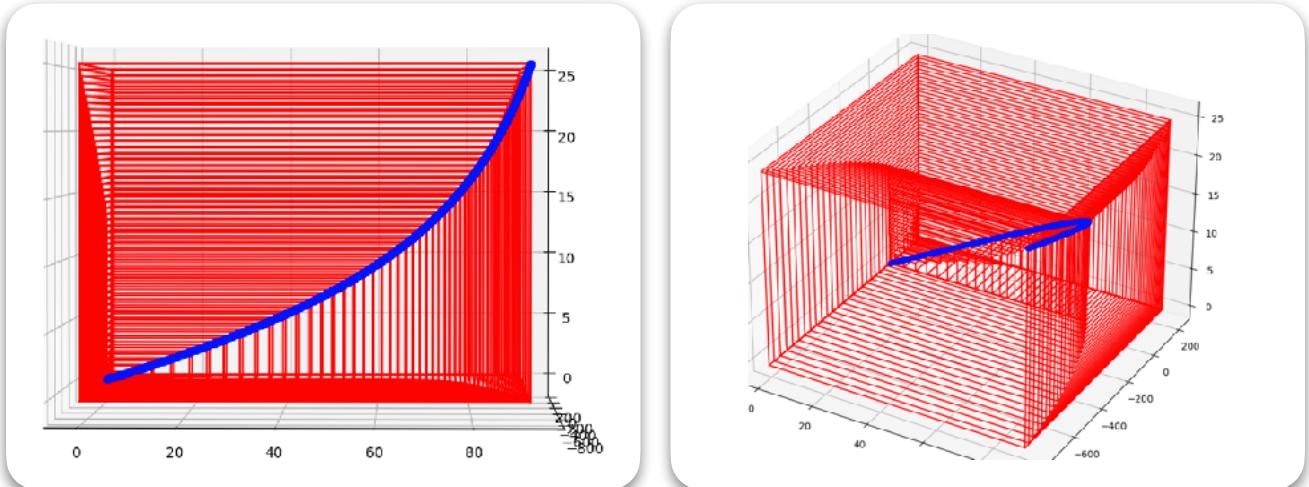
Bulk Insert

BulkInsert function στο demo.py τοποθετεί όλα τα δεδομένα στο 3D Rtree σε περίπου 0.001 sec.

```
○ → alekarakos git:(main) ✘ /usr/local/bin/python3 "/Users/iloudaros/Documents/Projects/My Projects/CEID/Multidimensional-DS-Project/Contributions/alekarakos/demo.py"
Inserted all data points in 0.0011320114135742188 seconds.
```

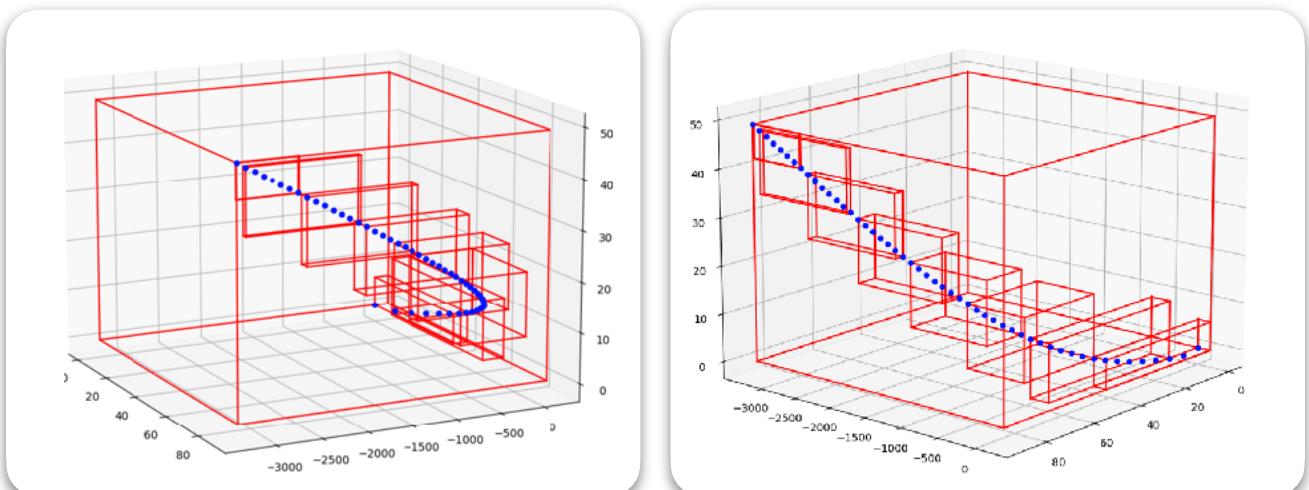
VisualizeRtree

Συνάρτηση η οποία παρουσιάζει σε γραφικό κομμάτι όλα τα δεδομένα και παραγόμενα boxes του δέντρου. Μετά την εισαγωγή των δεδομένων το visualisation είναι το εξής:



Το οποίο είναι λογικό αποτέλεσμα καθώς η δειγματοληψία μας έχει πολύ μεγάλη ακρίβεια (μεταβλητή dt)

Για να φανεί καλύτερα η σωστή λειτουργία και ιδέα του 3D Rtree αν μειώσουμε την ακρίβεια της δειγματοληψίας (μεγαλύτερο dt) έχουμε το εξής αποτέλεσμα:



Μπορούμε να δούμε το nesting διαφορετικών μεγεθών boxes καθώς διαγράφεται ξεκάθαρα η τροχιά του αντικειμένου.

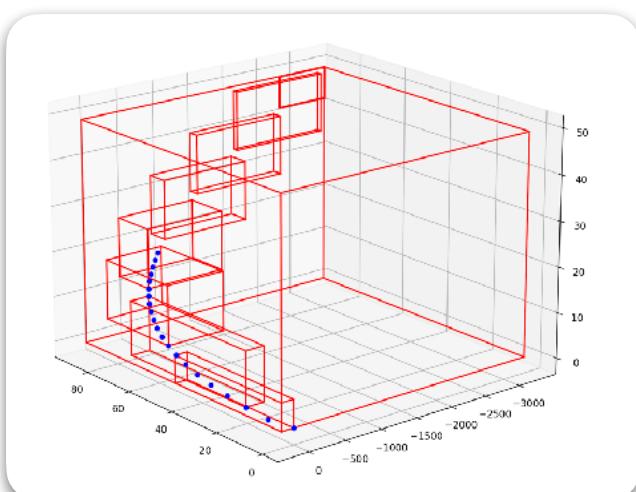
Trajectory Queries

Ένα trajectory query μπορεί να έχει πολλές μορφές. Ερωτήματα με βάση την ταχύτητα ή την επιτάχυνση του αντικειμένου, βάση συντεταγμένων x,y ή συνδυασμό τους. Στην συγκεκριμένη υλοποίηση κάνουμε ένα trajectory query για ένα συγκεκριμένο timeframe, δηλαδή, επιστροφή των σημείων που ανήκουν στην τροχιά του αντικειμένου από ένα t_0 εώς ένα $t_1 = t_0 + \Delta t$.

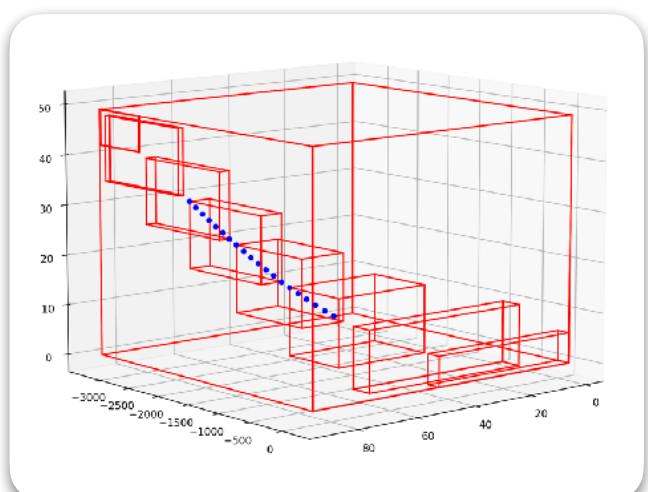
Ο τρόπος που γίνεται το query στο 3D R tree είναι δημιουργώντας ένα νέο box όπου η μία διάσταση που εκπροσωπεί τον χρόνο. Οι άλλες διαστάσεις εκπροσωπούν το x,y με $x_1=x_{min}$, $x_2=x_{max}$ και για αντίστοιχα (πληροφορίες που ήδη υπάρχουν στο root node συνεπώς δεν χρειάζεται κάποια αναζήτηση για να βρεθούν). Για την αναζήτηση βρίσκουμε ποια boxes και κατ' επέκταση ποια datapoint ανήκουν στο box query καθώς τέμνονται σε αυτό. Τα υπόλοιπα είναι παράλληλα σε αυτό και δεν τα "ανακαλύπτουμε" ποτέ. Η μέθοδος αυτή μπορεί εύκολα να προσαρμοστεί για να καλύψει τις ανάγκες για τους άλλους τύπους trajectory queries που αναφέρθηκαν νωρίτερα.

Παραδείγματα για το sparse dataset

Query Timeframe 0-20



Query Timeframe 15-35



Interval trees και Segment trees

Για την υλοποίηση αυτού του ερωτήματος για αρχή κατασκευάσαμε τις 2 δομές δεδομένων με τις βασικές τους πράξεις στα αρχεια [IntervalTree.py](#) και [SegmentTree.py](#).

Interval Trees

Ένα **Interval Tree** είναι μια προηγμένη δομή δεδομένων η οποία χρησιμοποιείται προκειμένου να καταγράψει και να αναζητήσει διαστήματα, ή intervals, με ιδιαίτερη αποτελεσματικότητα. Αυτό είναι ιδιαίτερα χρήσιμο για την αναζήτηση τμημάτων σε έναν μονοδιάστατο χώρο, και εν γένει σε περιπτώσεις όπου τα διαστήματα μπορεί να αλληλοεπικαλύπτονται ή να τέμνονται.

Η δομή αυτή βρίσκει εφαρμογές σε πολλούς τομείς, όπως στην βιολογία υπολογιστών για την εύρεση γονιδίων που είναι περιορισμένα μέσα σε κομμάτια DNA, και στη γεωγραφική πληροφορία για τον εντοπισμό επιφανειών που τέμνονται. Επίσης, είναι κρίσιμο εργαλείο για την επίλυση προβλημάτων που συνδέονται με τον προγραμματισμό εργασιών και την αναζήτηση ελεύθερων χρονικών διαστημάτων σε ένα ημερολόγιο.

Πώς Λειτουργεί;

Κάθε κόμβος σε ένα Interval Tree αντιπροσωπεύει ένα διάστημα. Ο κόμβος έχει τρία χαρακτηριστικά: το διάστημα, ένα δείκτη προς το αριστερό υποδέντρο, και έναν δείκτη προς το δεξί υποδέντρο. Επίσης, κάθε κόμβος περιέχει το μέγιστο τέλος του διαστήματος του και των διαστημάτων των υποδέντρων του.

Υλοποίηση

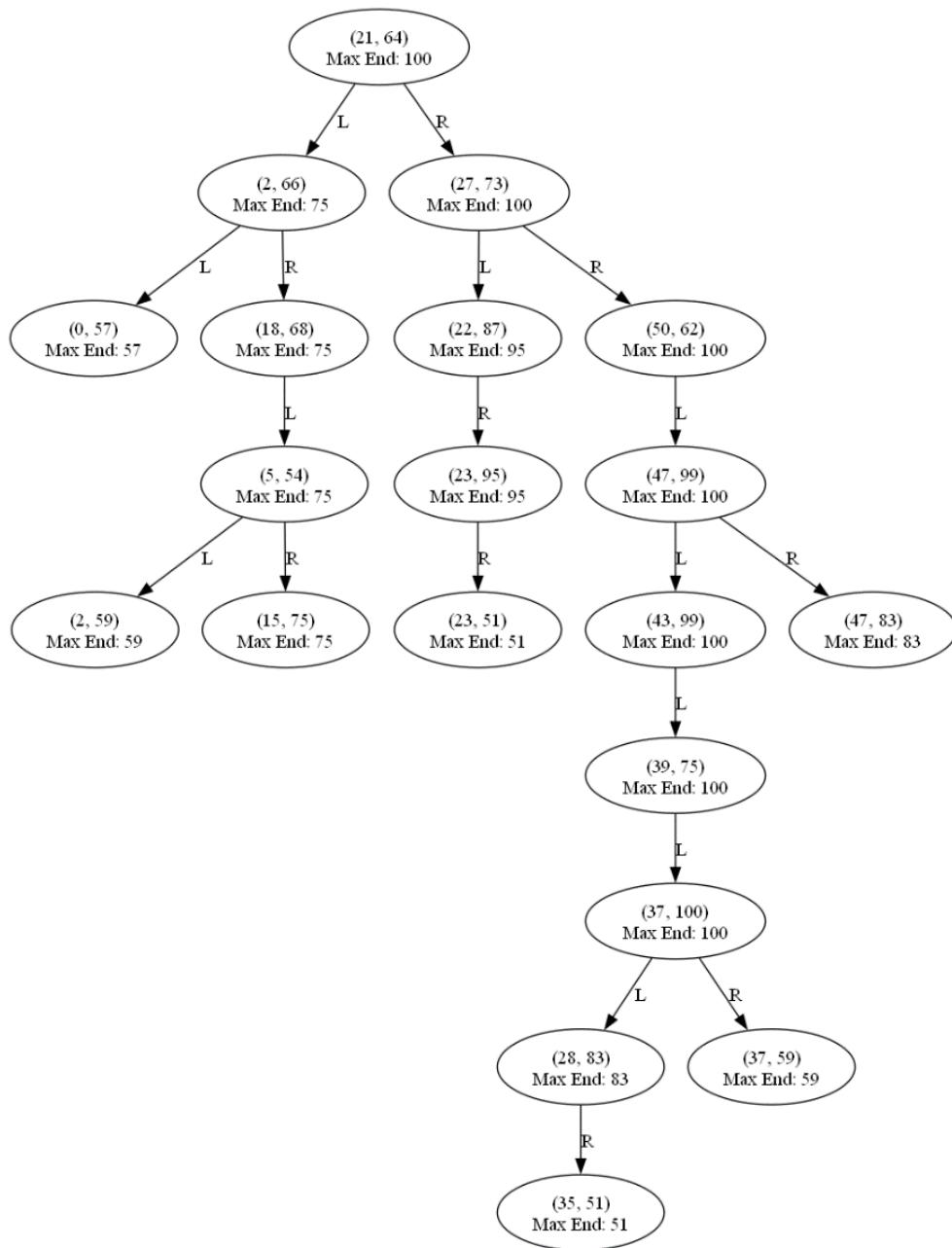
Στην υλοποίηση του κώδικα, κάθε κόμβος του δέντρου αντιπροσωπεύει ένα διάστημα και έχει επίσης έναν δείκτη προς το αριστερό και δεξί υποδέντρο.

- Η `insert` συνάρτηση προσθέτει ένα νέο κόμβο βάσει του αριστερού τελειώματός του διαστήματος.
- Η `delete` συνάρτηση διαγράφει έναν κόμβο και τον αντικαθιστά με τον προηγούμενο ή τον επόμενο κόμβο.

- Η `update` συνάρτηση ενημερώνει τις τιμές ενός κόμβου.
- Η `search` συνάρτηση στην ουσία είναι interval query και βρίσκει όλα τα διαστήματα που τέμνονται με ένα δοσμένο διάστημα.

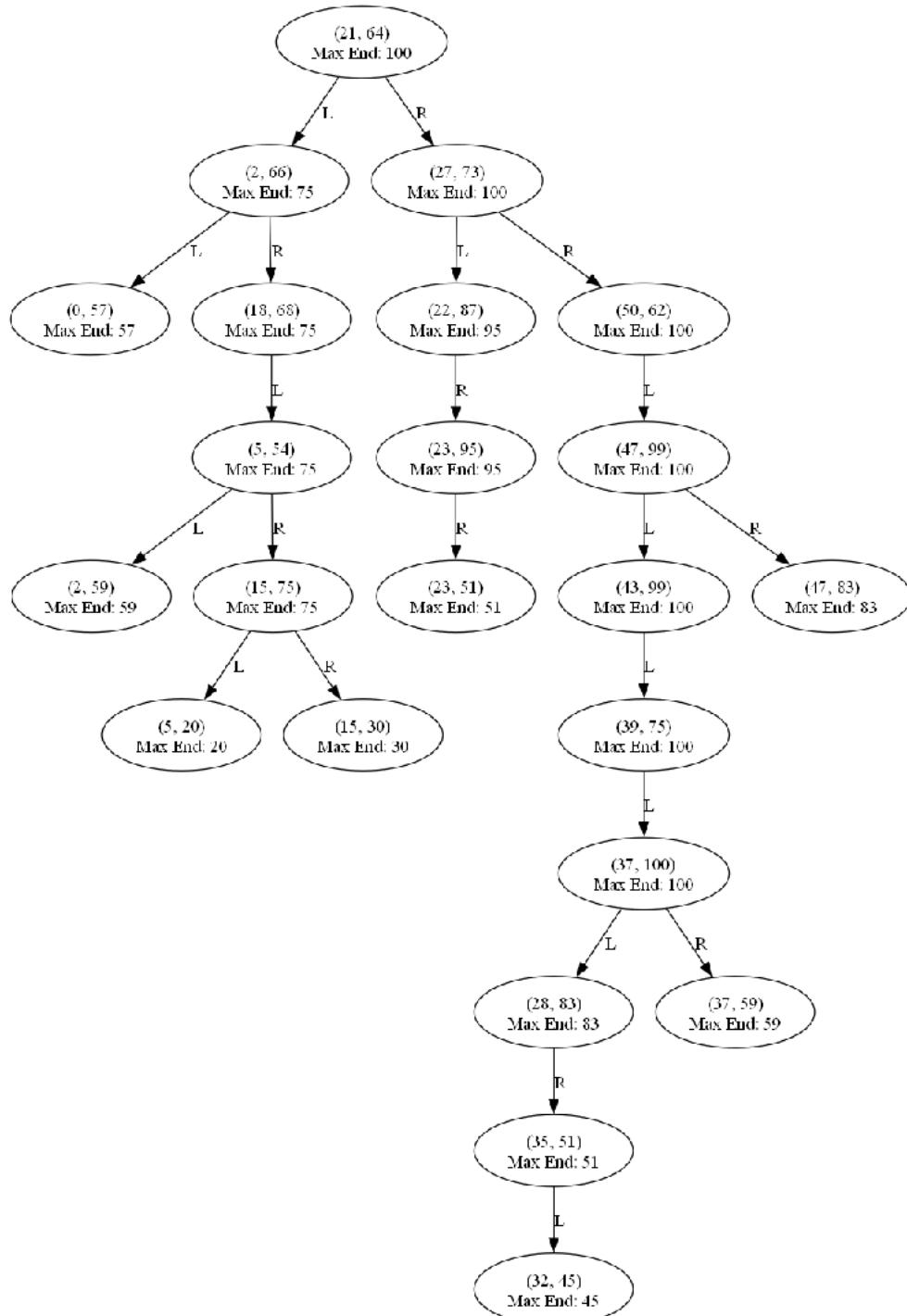
Σε κάθε στάδιο της υλοποίησης στον κώδικα δημιουργούμε ένα visualization του δέντρου. Για αρχή χρησιμοποιούμε 20 nodes, τυχαία αρχικοποιημένα για να κάνουμε build το δέντρο, το οποίο φαίνεται παρακάτω:

Interval Tree After Build



Στη συνέχεια ορίζουμε τα ακόλουθα intervals για να τα κάνουμε insert στο interval tree: [(15, 30), (32, 45), (5, 20)], και το δέντρο μετά τα inserts είναι το ακόλουθο:

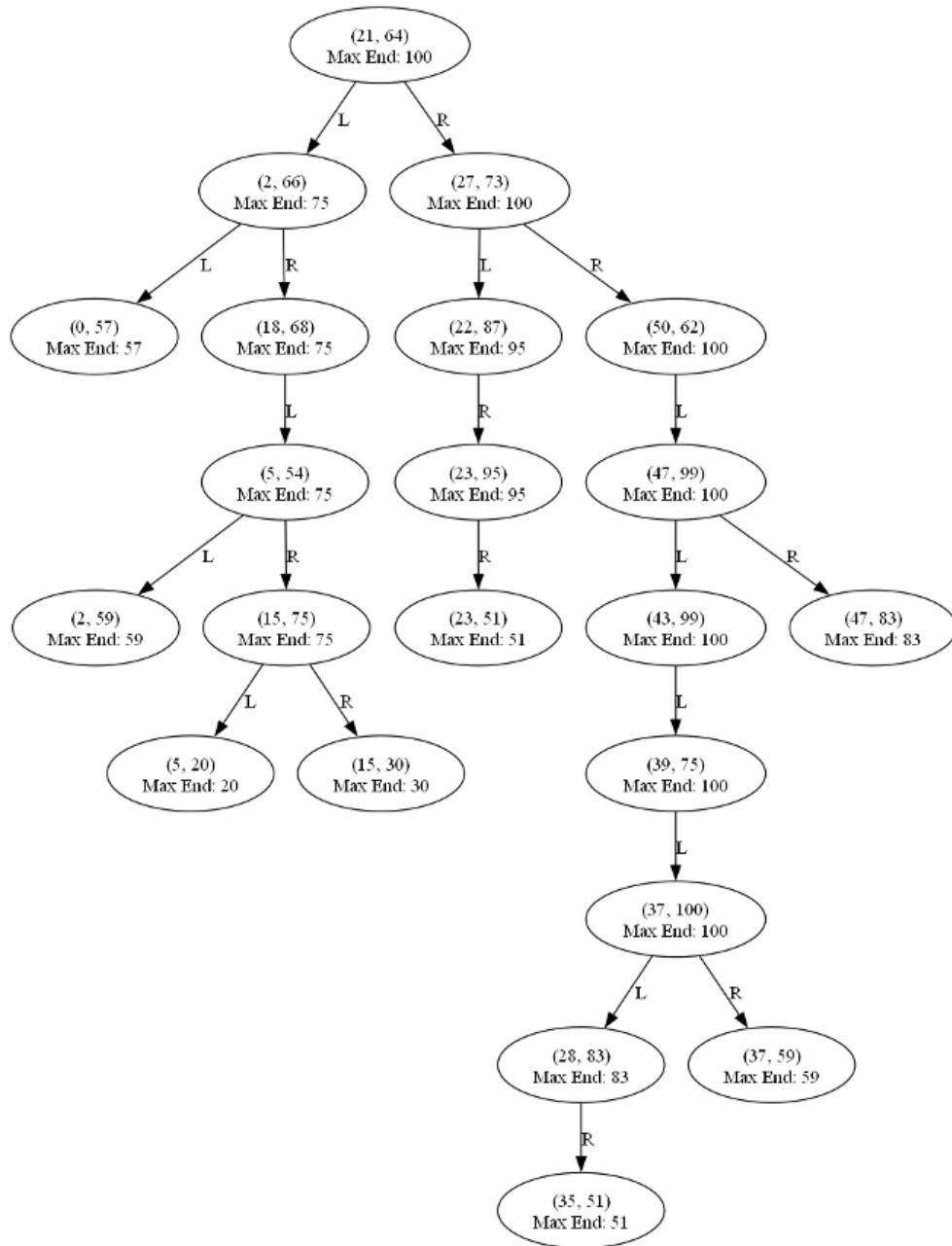
Interval Tree After Inserts



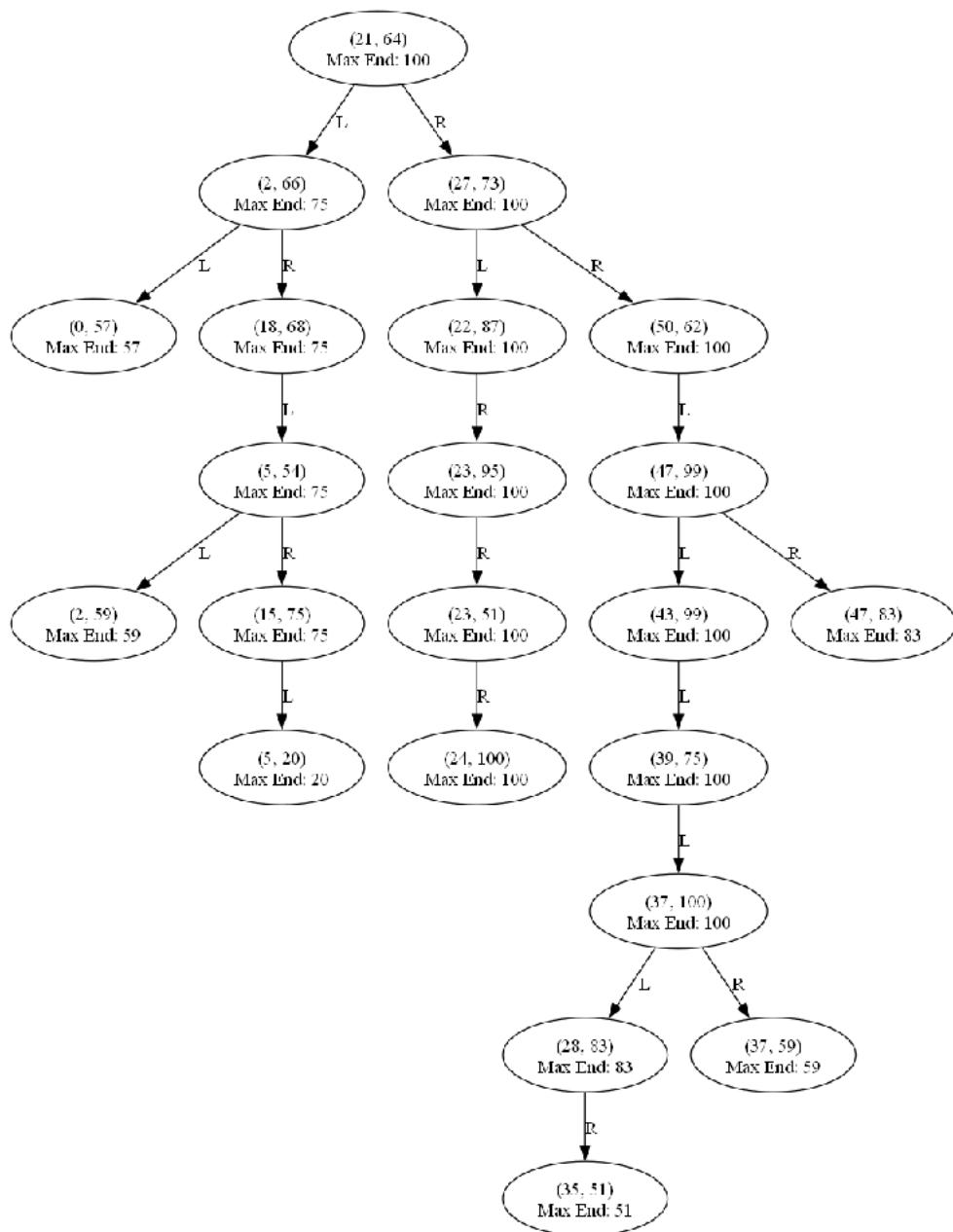
Έπειτα διαγράφουμε το node (32, 45) που κάναμε insert πρίν και παίρνουμε την καινούρια εικόνα:

Interval Tree After Delete

Τέλος κάνουμε update το interval $(15, 30)$ που επίσης κάναμε insert προηγουμένως και το αντικαθιστούμε με το $(24, 100)$.



Interval Tree After Update



Επίσης κάνουμε και ένα interval query με το (10, 20) για να βρούμε με ποια intervals κάνει intersect το συγκεκριμένο διάστημα και παίρνουμε τα ακόλουθα αποτελέσματα:

Found interval that intersects with (10, 20):

[(2, 66), (0, 57), (18, 68), (5, 54), (2, 59), (15, 75), (5, 20)]

Απόδοση

Καθόλη τη διάρκεια του τρεξίματος κρατάμε μετρήσεις σχετικά με το πόσο χρόνο και πόση μνήμη απαιτεί κάθε πράξη. Μπορείτε να βρείτε το log αυτών των μετρήσεων εδώ.

Αξιολόγηση

Με βάση αυτά τα αποτελέσματα οι χρόνοι εκτέλεσης και η χρησιμοποιούμενη μνήμη φαίνεται να αυξάνονται, όπως είναι λογικό, με την αύξηση των στοιχείων (n).

Χρόνος Εκτέλεσης

Ο χρόνος εκτέλεσης για την κατασκευή του δέντρου αυξάνεται από 0.0002216999 seconds για $n=20$ σε 0.4228657000 seconds για $n=5000$. Οι χρόνοι εισαγωγής, διαγραφής, ενημέρωσης, και αναζήτησης είναι αρκετά γρήγοροι, ακόμη και για $n=5000$, αλλά παρουσιάζουν μια ελαφρώς αυξανόμενη τάση.

Χρησιμοποιούμενη Μνήμη

Η χρησιμοποιούμενη μνήμη για την κατασκευή του δέντρου αυξάνεται από 0.0195312500 MiB για $n=20$ σε 0.5351562500 MiB για $n=5000$. Η χρησιμοποιούμενη μνήμη για τις άλλες λειτουργίες παραμένει σχετικά σταθερή.

Συμπεράσματα

Το interval tree που φτιάξαμε φαίνεται να έχει καλή απόδοση τόσο σε χρόνο όσο και σε μνήμη για τις παραπάνω περιπτώσεις. Είναι σημαντικό να σημειωθεί ότι, ενώ ο χρόνος και η μνήμη που χρησιμοποιούνται για την κατασκευή του δέντρου αυξάνονται με το μέγεθος του δέντρου, οι χρόνοι για τις βασικές λειτουργίες (εισαγωγή, διαγραφή, ενημέρωση, αναζήτηση) παραμένουν σε λογικά πλαίσια, ακόμη και για μεγαλύτερα n , το οποίο είναι ένα θετικό σημείο για την αποτελεσματικότητα του δέντρου.

Segment Trees

Ένα **Segment tree** είναι ένας τύπος δυαδικού δέντρου που χρησιμοποιείται για την αποθήκευση και αναζήτηση πληροφοριών σχετικά με διαστήματα ή τμήματα ενός μονοδιάστατου χώρου. Χρησιμοποιείται για την εκτέλεση ερωτημάτων εύρους σε έναν πίνακα ή λίστα δεδομένων, όπως η εύρεση του αθροίσματος, της μέγιστης ή της ελάχιστης τιμής μιας περιοχής στοιχείων. Οι κόμβοι φύλλων του δέντρου αντιπροσωπεύουν μεμονωμένα στοιχεία του πίνακα εισόδου και οι εσωτερικοί κόμβοι αντιπροσωπεύουν εύρη στοιχείων. Το δέντρο μπορεί να κατασκευαστεί σε χρόνο $O(n \log n)$ και υποστηρίζει ερωτήματα σε χρόνο $O(\log n)$, καθιστώντας το μια αποτελεσματική δομή δεδομένων για την επεξεργασία μεγάλων ποσοτήτων δεδομένων.

Πως Δουλεύει:

Το Segment Tree διαίρει το σύνολο των στοιχείων του στη μέση και δημιουργεί δυο υποδένδρα, ένα για το αριστερό υποσύνολο και ένα για το δεξί υποσύνολο. Αυτή η διαδικασία επαναλαμβάνεται αναδρομικά μέχρι να φτάσουμε στα φύλλα του δένδρου, τα οποία αντιπροσωπεύουν τα στοιχεία της αρχικής ακολουθίας.

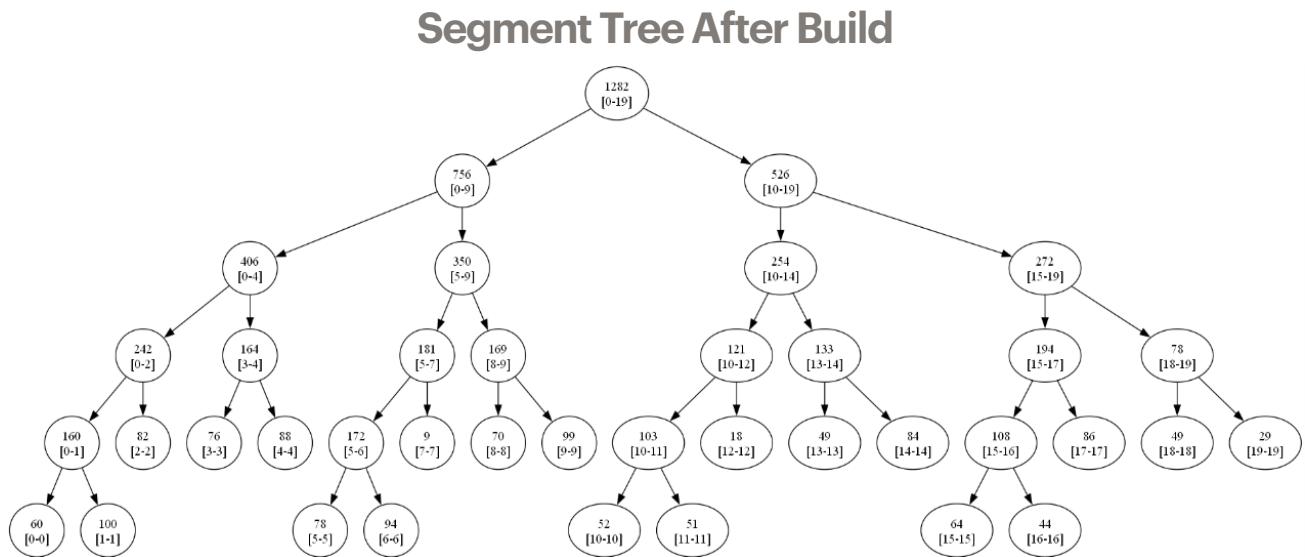
Υλοποίηση

Ο κώδικας μας ορίζει μια κλάση SegmentTree με διάφορες μεθόδους όπως `build`, `insert`, `delete`, `update`, `stabbing_query` και `range_query`.

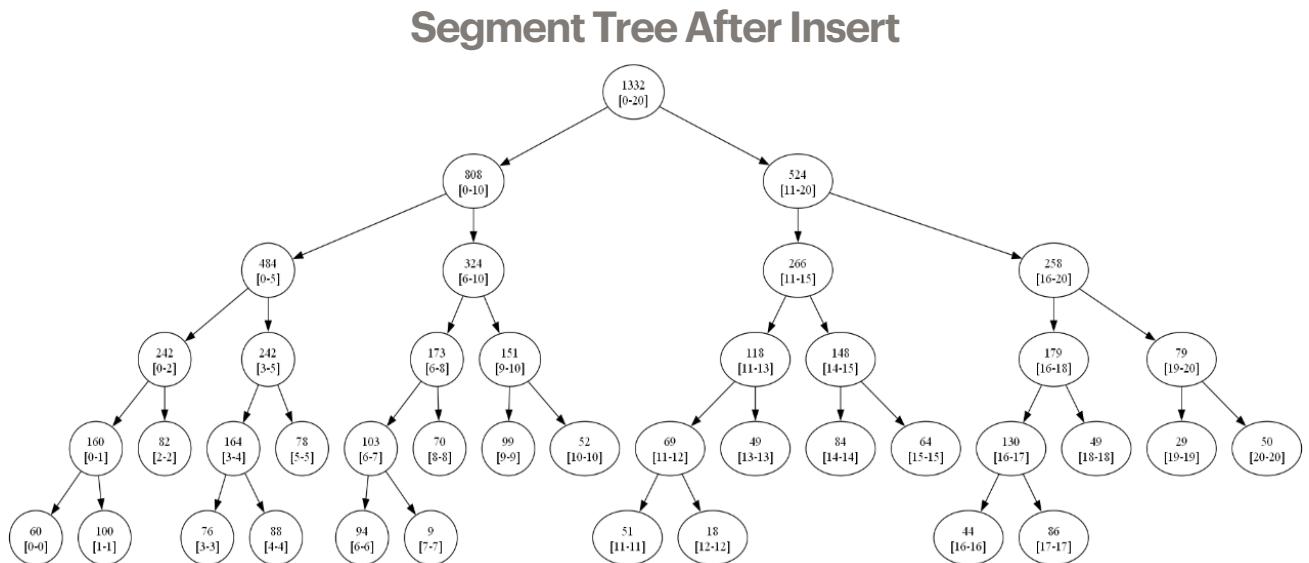
- Η `build` χρησιμοποιείται για τη δημιουργία του δέντρου από έναν πίνακα.
- Η `insert` και `delete` χρησιμοποιούνται για την εισαγωγή και διαγραφή στοιχείων αντίστοιχα.
- Η `update` χρησιμοποιείται για την ενημέρωση τιμής ενός στοιχείου.
- Η `stabbing_query` χρησιμοποιείται για να επιστρέψει την τιμή ενός στοιχείου στον αρχικό πίνακα.
- Η `range_query` χρησιμοποιείται για να επιστρέψει το άθροισμα των στοιχείων σε ένα εύρος του αρχικού πίνακα.

Παράλληλα χρονομετρούμε την κάθε συνάρτηση καθώς και τις χωρικές της απαιτήσεις κατά την υλοποίηση, όπως και κάνουμε οπτικοποίηση της δομής σε κάθε στάδιο υλοποίησης.

Για αρχή χρησιμοποιούμε 20 nodes, τυχαία αρχικοποιημένα για να κάνουμε build το δέντρο όπως και το προηγούμενο, το οποίο φαίνεται παρακάτω:

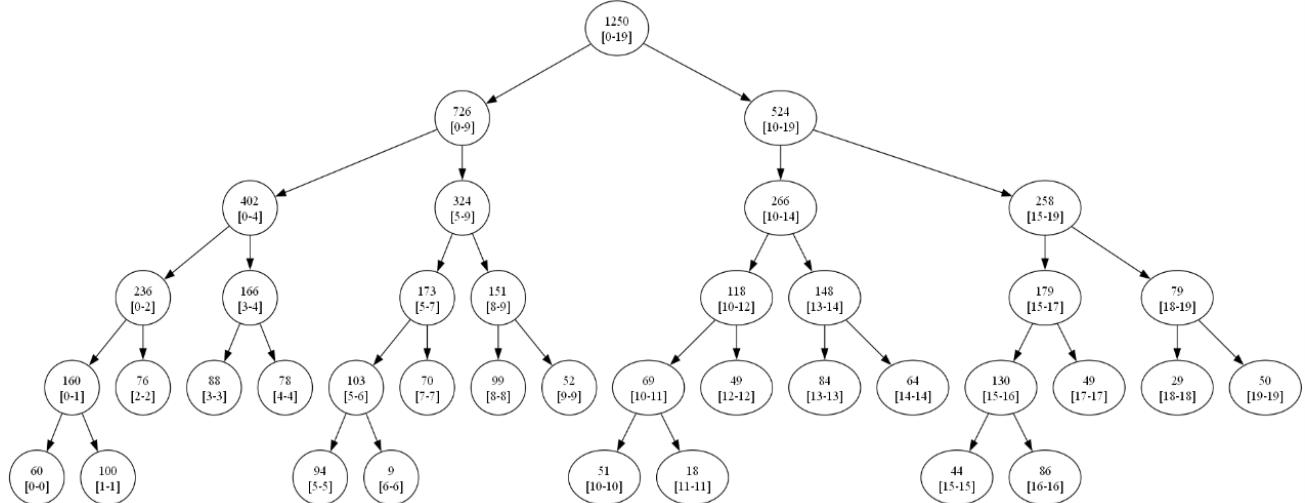


Στη συνέχεια κάνουμε insert το node με τιμή 50:



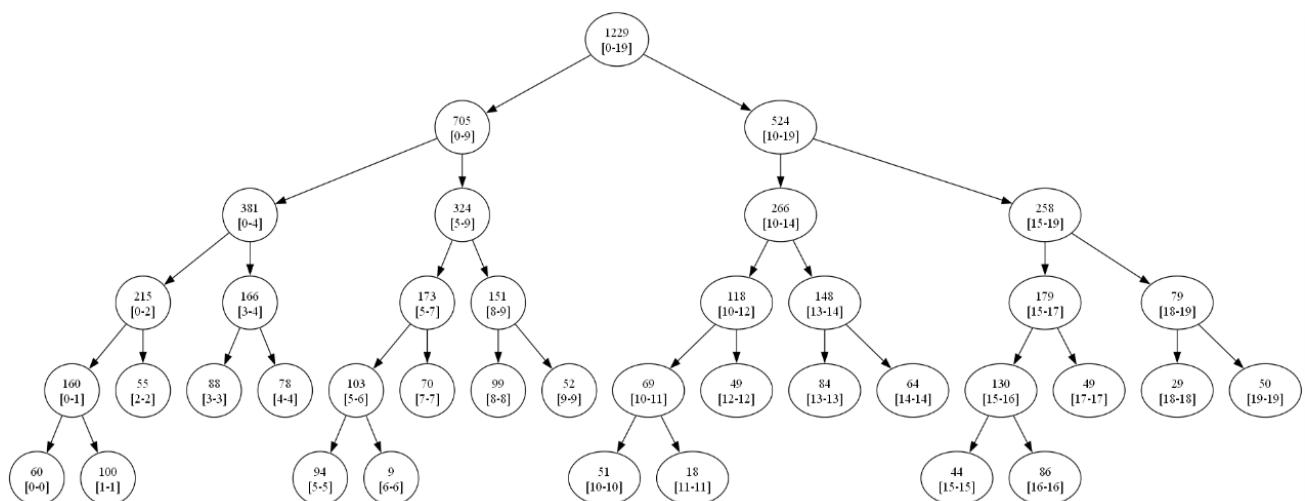
Έπειτα διαγράφουμε το στοιχείο στο 2o index του δέντρου και ξαναφτιάχνουμε το δέντρο.

Segment Tree After Delete



Στη συνέχεια κάνουμε update την τιμή του node στο index=2, σε 55 και ξαναζωγραφίζουμε το δέντρο

Segment Tree After Update



Stabbing Kai Range Queries:

Stabbing query result for index 11: 18

Range query result for sum in range (2, 10): 596

Απόδοση

Καθόλη τη διάρκεια του τρεξίματος κρατάμε μετρήσεις σχετικά με το πόσο χρόνο και πόση μνήμη απαιτεί κάθε πράξη. Μπορείτε να βρείτε το log αυτών των μετρήσεων εδώ.

Αξιολόγηση

Build

Η χρήση μνήμης και ο χρόνος αυξάνονται λογικά με το μέγεθος του δέντρου, είναι αναμενόμενο και σχετίζεται με την πολυπλοκότητα της κατασκευής του δέντρου $O(n \log n)$.

Insert

Η χρήση μνήμης και ο χρόνος αυξάνονται ενώ το μέγεθος του δέντρου αυξάνεται. Σε μεγάλο δέντρο ($n=5000$), η χρήση της μνήμης καταγράφεται πολύ υψηλότερα, πιθανώς λόγω των αναδιοργανώσεων των δεδομένων και των επανακατασκευών τμημάτων του δέντρου.

Delete και Update

Οι χρόνοι εκτέλεσης είναι σχετικά χαμηλοί για όλα τα μεγέθη, με τη διαγραφή να είναι λίγο πιο αργή στα μεγάλα μεγέθη.

Search

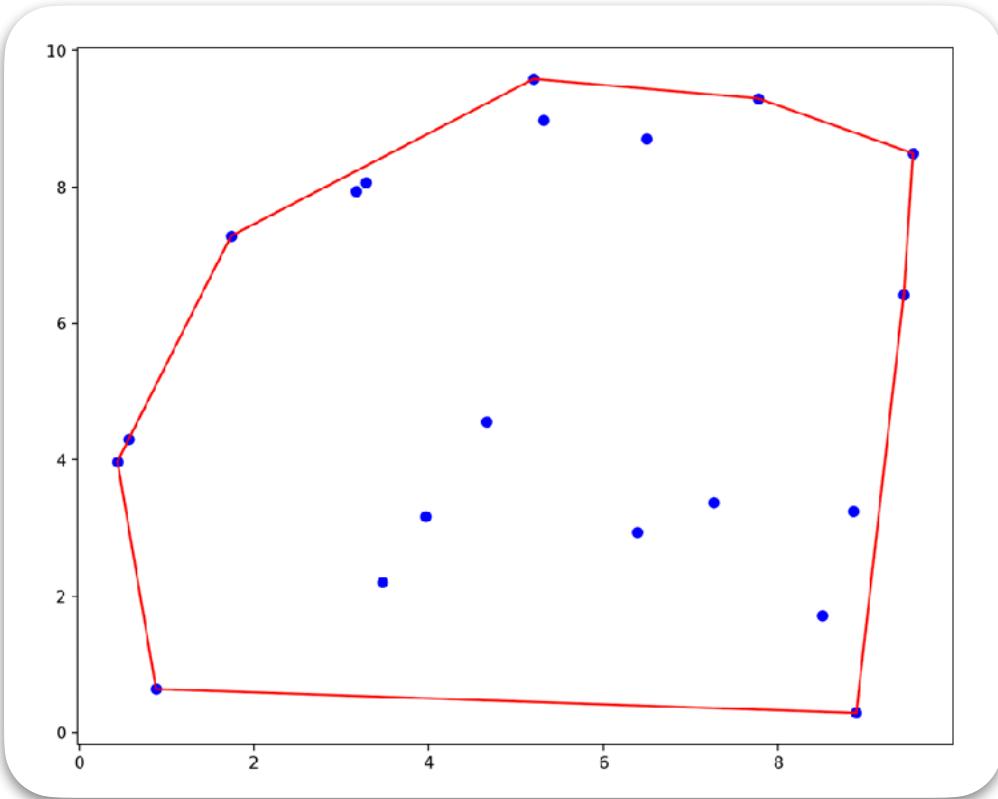
Ο χρόνος είναι πολύ χαμηλός και φαίνεται να είναι αρκετά αποδοτικός, ακόμη και για μεγάλες τιμές του n .

Συμπεράσματα

Συνολικά, το Segment Tree δείχνει να είναι αρκετά αποδοτικό για τις βασικές λειτουργίες, εκτός ίσως από την εισαγωγή στις περιπτώσεις όπου το n είναι μεγάλο, όπου η χρήση της μνήμης μπορεί να γίνει σημαντική.

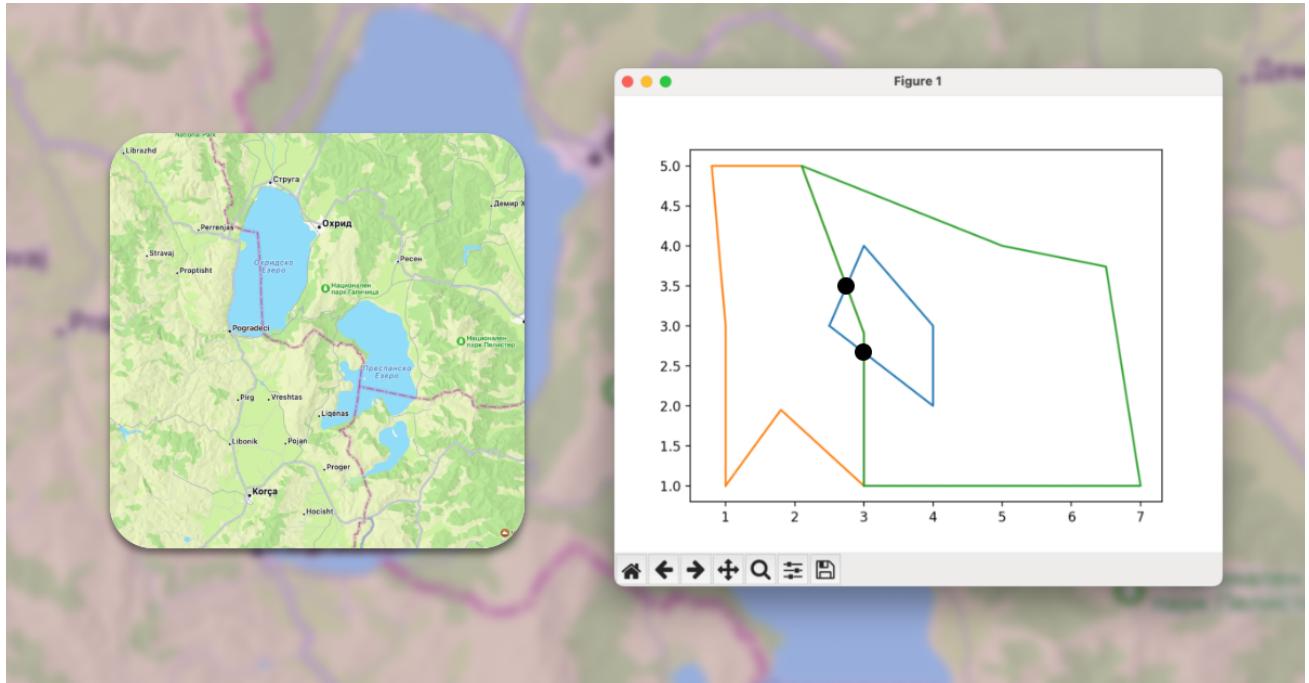
Convex Hull

Το πρόγραμμα χρησιμοποιεί τον αλγόριθμο Graham scan για την υλοποίηση του convex hull. Η κύρια συνάρτηση του είναι η `convex_hull`, η οποία υπολογίζει το κυρτό περίβλημα. Ταξινομεί τα σημεία του επιπέδου λεξικογραφικώς, δηλαδή ξεκινάει με το αριστερότερο σημείο (το σημείο με την μικρότερη συντεταγμένη x) και στη συνέχεια προσθέτει σημεία στο περίβλημα αριστερόστροφα. Αυτό συμβαίνει ελέγχοντας το κάθε σημείο και κρατώντας μόνο αυτά που είναι αριστερά της γραμμής που σχηματίζεται από τα δύο τελευταία σημεία του περιβλήματος. Αν το σημείο υπό εξέταση βρίσκεται στα αριστερά, τότε απορρίπτεται καθώς δεν είναι μέρος του convex hull. Η διαδικασία επαναλαμβάνεται ώσπου να ελεγχθούν όλα τα σημεία. Για την κατασκευή του lower hull τα σημεία ελέγχονται με την σειρά που εμφανίζονται, ενώ για την κατασκευή του upper hull ελέγχονται με αντίθετη σειρά. Έπειτα, τα δύο κομμάτια ενώνονται και αφαιρείται το διπλότυπο σημείο και έτσι προκύπτει το τελικό περίβλημα.



Για την οτικοποίηση των αποτελεσμάτων γίνεται generate ένας αριθμός από τυχαία σημεία με την χρήση της βιβλιοθήκης random.

Line Segment Intersection



Μπορείτε να βρείτε τον κώδικα που αντιστοιχεί στο ερώτημα στο [Παράρτημα Δ.](#)

Αυτό το ερώτημα εξερευνά αλγορίθμους εύρεσης τομών ευθυγράμμων τμημάτων με εφαρμογές στο επίπεδο που προκύπτουν από τα "σύνορα" πολυγωνικών περιοχών (όπως π.χ. σε περιπτώσεις υπερθέσεις χαρτών). Αναπτύξαμε δύο υλοποιήσεις αλγορίθμων. Αυτές είναι:

- Αφελής αλγόριθμος (`pol_interesect_naive()`)
- Αλγόριθμος βασισμένος σε γραμμή σάρωσης (`sweep_line()`)

Οι υλοποιήσεις των αλγορίθμων βρίσκονται στο αρχείο `line_segment_intersection.py`

Δομές που Παράχθηκαν

Για τις ανάγκες των ερωτημάτων δημιουργήθηκαν οι δομές Line, Event και Polygon ώστε ο κώδικας να είναι ευανάγνωστος και να γίνεται αντιληπτή η έννοια που πραγματεύεται κάθε φορά.

Τα δεδομένα και το Demo

Εμπνευστίκαμε το Demo από περιπτώσεις λιμνών ως φυσικών συνόρων μεταξύ χωρών. Έτσι λοιπόν, στο παράδειγμα μας έχουμε δύο πολύγωνα (country1 και country2) τα οποία μοιράζονται κάποιες πλευρές, και άλλο ένα πολύγωνο (lake). Τα δεδομένα μας είναι δημιουργημένα από εμάς για τις ανάγκες του ερωτήματος.

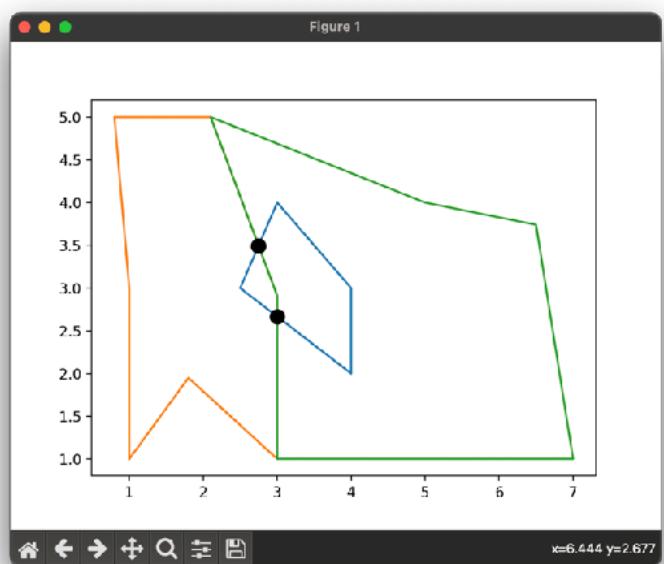
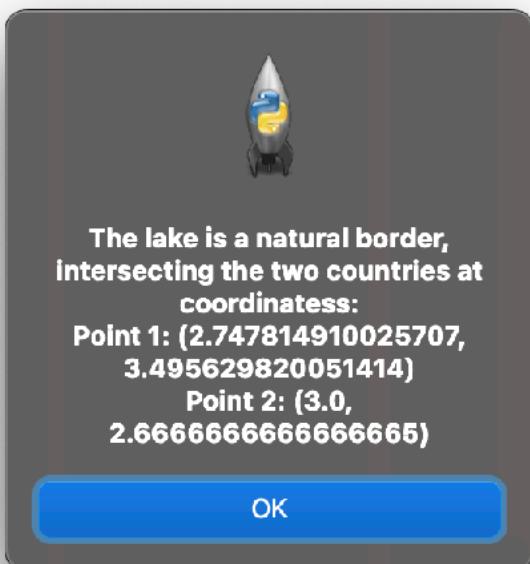
Τρόπος Εκτέλεσης

Για να μπορέσετε να τρέξετε το demo θα χρειαστείτε τα πακέτα matplotlib και random.

Για να το εκτελέσετε αρκεί το : python lsi_demo.py

Αποτελέσματα

Ο κώδικας μας ελέγχει αν τα πολύγωνα τέμνονται. Ύστερα αν τέμνονται, μας ενημερώνει για τα σημεία και μας τα δείχνει γραφικά, όπως φαίνεται παρακάτω.



Παράρτημα A

Ορισμός του Point

```
class Point:  
    def __init__(self, x: float, y: float, t: float) -> "Point":  
        self.x = x  
        self.y = y  
        self.t = t  
    # MBB = Minimum Bounding Box  
    self.mbb = Box(x, y, t, x, y, t)  
  
    def __str__(self) -> str:  
        return f"x:{self.x} y:{self.y} t:{self.t}"
```

Ορισμός του Entry

```
class Entry:  
    """  
    Represents the data of each node. If the node is a leaf it contains a trajectory instance (x,y,t) else a pointer  
    to a child node.  
    """  
  
    def __init__(self, child: "Node" = None, data: "Point" = None) -> "Entry":  
        self.child = child  
        self.data = data  
        self.box = self.getEntryBox()  
  
    def __str__(self) -> str:  
        return f"Entry Box: {self.box}"  
  
    def getEntryBox(self) -> "Box":  
        """Sets the size of the minimum bounding box of the entry based on child node or data."""  
        if self.child == None:  
            # box size = point mbb  
            return self.data.mbb  
        return self.child.getMBB()
```

Ορισμός του Box

```
class Box:  
    def __init__(  
        self,  
        x1: float,  
        y1: float,  
        z1: float,  
        x2: float,  
        y2: float,  
        z2: float,  
    ) -> "Box":  
        self.x1 = x1  
        self.y1 = y1  
        self.z1 = z1  
        self.x2 = x2  
        self.y2 = y2  
        self.z2 = z2  
  
    def __str__(self) -> str:  
        return f'x:{self.x1, self.x2} x:{self.y1, self.y2} z:{self.z1, self.z2}'  
  
    def getVolume(self) -> float:  
        return fabs((self.x2 - self.x1) * (self.y2 - self.y1) * (self.z2 - self.z1))  
  
    def getCoordinates(self):  
        return self.x1, self.y1, self.z1, self.x2, self.y2, self.z2  
  
    def overlaps(self, box: "Box") -> bool:  
        a, b = self, box  
        x1 = max(min(a.x1, a.x2), min(b.x1, b.x2))  
        y1 = max(min(a.y1, a.y2), min(b.y1, b.y2))  
        x2 = min(max(a.x1, a.x2), max(b.x1, b.x2))  
        y2 = min(max(a.y1, a.y2), max(b.y1, b.y2))  
        z1 = max(min(a.z1, a.z2), min(b.z1, b.z2))  
        z2 = min(max(a.z1, a.z2), max(b.z1, b.z2))  
        return x1 <= x2 and y1 <= y2 and z1 <= z2
```

Ορισμός του Node

```
class Node:  
    def __init__(self, parent: "Node" = None, entries: "Entry" = []) -> "Node":  
        self.parent = parent  
        self.entries = entries  
  
        self.assignParent()  
  
    def isRoot(self) -> bool:  
        return self.parent == None  
  
    def isLeaf(self) -> bool:  
        if self.isRoot():  
            return self.entries == [] or self.entries[0].data != None  
        return self.entries[0].data != None  
  
    def isFull(self) -> bool:  
        return len(self.entries) >= MAX_ENTRIES  
  
    def getMBB(self) -> "Box":  
        """Returns the minimum bounding box for this node based on the entries."""  
  
        boxes = [entry.box for entry in self.entries]  
        return MBB(boxes)  
  
    def insertEntry(self, entry: "Entry") -> None:  
        self.entries.append(entry)  
  
        self.assignParent()  
  
    def parentEntry(self):  
        if self.parent is not None:  
            return next(entry for entry in self.parent.entries if entry.child is self)  
        return None  
  
    def assignParent(self) -> None:  
        """Assign parent to child nodes"""  
        if not self.isLeaf():  
            for entry in self.entries:  
                entry.child.parent = self  
  
    def getLeastVolumeEnlargement(self, new_entry: "Entry") -> "Node":  
        """Returns the node that will be enlarged the least after the new entry insertion"""  
        enlargement_list = []  
        for entry in self.entries:  
  
            volumeEnlargement = getVolumeEnlargement(new_entry.box, entry.box)  
            enlargement_list.append([entry.child, volumeEnlargement])  
  
        # Get the Node that requires least enlargement to fit new entry.  
        return min(enlargement_list, key=lambda e: e[1])[0]
```

Παράρτημα B

Interval Trees

```
from graphviz import Digraph
import random
import timeit
from memory_profiler import memory_usage

class Node:
    def __init__(self, interval, max_end):
        self.interval = interval
        self.max_end = max_end
        self.left = None
        self.right = None

class IntervalTree:
    def __init__(self):
        self.root = None

    def insert(self, node, interval):
        if node is None:
            max_end = max(interval)
            return Node(interval, max_end)

        low = interval[0]
        if low < node.interval[0]:
            node.left = self.insert(node.left, interval)
        else:
            node.right = self.insert(node.right, interval)

        if node.max_end < max(interval):
            node.max_end = max(interval)

    return node

    def delete(self, interval):
        self.root, _ = self.delete_interval(self.root, interval)
    ...


```

```

...
def delete_interval(self, root, interval):
    if not root:
        return root, None

    deleted_node = None
    if root.interval == interval:
        deleted_node = root

    if root.left:
        max_node = root.left
        while max_node.right:
            max_node = max_node.right

        root.interval = max_node.interval
        root.left, _ = self.delete_interval(root.left, max_node.interval)

    elif root.right:
        min_node = root.right
        while min_node.left:
            min_node = min_node.left

        root.interval = min_node.interval
        root.right, _ = self.delete_interval(root.right, min_node.interval)

    else:
        root = None

    elif interval[0] < root.interval[0]:
        root.left, deleted_node = self.delete_interval(root.left, interval)

    else:
        root.right, deleted_node = self.delete_interval(root.right, interval)

    if root:
        root.max_end = max(
            root.interval[1],
            root.left.max_end if root.left else float("-inf"),
            root.right.max_end if root.right else float("-inf"),
        )

    return root, deleted_node

def update(self, old_interval, new_interval):
    self.delete(old_interval)
    self.root = self.insert(self.root, new_interval)

...

```

```

...
def search(self, node, interval):
    if node is None:
        return []

    intersecting_intervals = []
    low, high = interval
    node_low, node_high = node.interval

    if node_low <= high and low <= node_high:
        intersecting_intervals.append(node.interval)

    if node.left is not None and node.left.max_end >= low:
        intersecting_intervals.extend(self.search(node.left, interval))

    if node.right is not None and node_low <= high: # Also check the right subtree
        intersecting_intervals.extend(self.search(node.right, interval))

    return intersecting_intervals

def build(self, intervals):
    for interval in intervals:
        self.root = self.insert(self.root, interval)

# Function to visualize the Interval Tree
def visualize_tree(node, graph=None):
    if graph is None:
        graph = Digraph()

    if node is not None:
        graph.node(f'{id(node)}', f'{node.interval}\nMax End: {node.max_end}')

        if node.left:
            graph.node(
                f'{id(node.left)}',
                f'{node.left.interval}\nMax End: {node.left.max_end}',
            )
            graph.edge(f'{id(node)}', f'{id(node.left)}', "L")
            visualize_tree(node.left, graph)

        if node.right:
            graph.node(
                f'{id(node.right)}',
                f'{node.right.interval}\nMax End: {node.right.max_end}',
            )
            graph.edge(f'{id(node)}', f'{id(node.right)}', "R")
            visualize_tree(node.right, graph)

    return graph

```

Segment Tree

```
import random
import timeit
from graphviz import Digraph
from memory_profiler import memory_usage

class SegmentTree:
    def __init__(self, size, arr=None):
        self.size = size
        self.arr = arr if arr else [0] * size # Keep the original array if it exists
        self.tree = [0] * (4 * size)
        if self.arr:
            self.build(self.arr, 1, 0, self.size - 1)

    def build(self, arr, i, l, r):
        if l == r:
            self.tree[i] = arr[l]
            return
        mid = (l + r) // 2
        self.build(arr, 2 * i, l, mid)
        self.build(arr, 2 * i + 1, mid + 1, r)
        self.tree[i] = self.tree[2 * i] + self.tree[2 * i + 1]

    def insert(self, val):
        # If the original array exists, append the value to it and rebuild the tree
        if self.arr:
            self.arr.append(val)
            self.size += 1
            self.tree = [0] * (4 * self.size)
            self.build(self.arr, 1, 0, self.size - 1)

    def delete(self, idx):
        if 0 <= idx < self.size and self.arr:
            self.arr.pop(idx) # Remove the element at idx from arr
            self.size -= 1 # Reduce the size of the Segment Tree
            self.tree = [0] * (4 * self.size) # Adjust the size of the tree list
            self.build(self.arr, 1, 0, self.size - 1) # Rebuild the tree

    def update(self, i, val, idx, l, r):
        if l == r:
            self.tree[i] = val
            return
        mid = (l + r) // 2
        if idx <= mid:
            self.update(2 * i, val, idx, l, mid)
        else:
            self.update(2 * i + 1, val, idx, mid + 1, r)
        self.tree[i] = self.tree[2 * i] + self.tree[2 * i + 1]

    ...

```

```

...
def stabbing_query(self, idx):
    """
    Returns the value at idx in the original array.
    """
    if 0 <= idx < self.size:
        return self._stabbing_query(1, 0, self.size - 1, idx)
    else:
        raise IndexError("Index out of range")

def _stabbing_query(self, i, l, r, idx):
    if l == r:
        return self.tree[i]
    mid = (l + r) // 2
    if idx <= mid:
        return self._stabbing_query(2 * i, l, mid, idx)
    else:
        return self._stabbing_query(2 * i + 1, mid + 1, r, idx)

def range_query(self, ql, qr):
    """
    Returns the sum of elements in the original array in the range [ql, qr].
    """
    if 0 <= ql <= qr < self.size:
        return self._range_query(1, 0, self.size - 1, ql, qr)
    else:
        raise IndexError("Query range out of bounds")

def _range_query(self, i, l, r, ql, qr):
    if ql <= l and r <= qr: # If the current segment is within the query range
        return self.tree[i]
    if qr < l or r < ql: # If the current segment is outside the query range
        return 0
    mid = (l + r) // 2
    # If the current segment overlaps with the query range,
    # compute the sum of the left and right children.
    return self._range_query(2 * i, l, mid, ql, qr) + self._range_query(
        2 * i + 1, mid + 1, r, ql, qr
    )

def visualize_tree(self):
    dot = Digraph(comment="Segment Tree")
    self._visualize_tree(1, 0, self.size - 1, dot)
    return dot

def _visualize_tree(self, i, l, r, dot):
    dot.node(f'{i}', label=f'{self.tree[i]}\n[{l}-{r}]') # Updated line
    if l == r:
        return
    mid = (l + r) // 2
    left_child_idx = 2 * i
    right_child_idx = 2 * i + 1
    self._visualize_tree(left_child_idx, l, mid, dot)
    self._visualize_tree(right_child_idx, mid + 1, r, dot)
    dot.edge(f'{i}', f'{left_child_idx}')
    dot.edge(f'{i}', f'{right_child_idx}')

```

Παράρτημα Γ

Παραγωγή και Οπτικοποίηση του Convex Hull

```
def convex_hull(points):
    """Calculates the convex hull of a set of points.

    Returns a list of points in the convex hull, in counter-clockwise order.
    """
    # Sort the points (first by x-coordinate, then by y-coordinate)
    points = sorted(points)

    # Build the lower hull
    lower = []
    for p in points:
        while len(lower) >= 2 and cross_product(lower[-2], lower[-1], p) <= 0:
            lower.pop()
        lower.append(p)

    # Build the upper hull
    upper = []
    for p in reversed(points):
        while len(upper) >= 2 and cross_product(upper[-2], upper[-1], p) <= 0:
            upper.pop()
        upper.append(p)

    # Join the lower and upper hulls, and add the first point of the hull to the end
    hull = lower[:-1] + upper[:-1]
    hull.append(hull[0])

    return hull

def visualize(points, hull):
    """Visualizes the points and the convex hull using matplotlib."""
    # Extract the x- and y-coordinates of the points and the hull
    xs = [p.x for p in points]
    ys = [p.y for p in points]
    hx = [p.x for p in hull]
    hy = [p.y for p in hull]

    # Plot the points and the hull
    plt.plot(xs, ys, "bo")
    plt.plot(hx, hy, "r-")
    plt.show()
```

Παράρτημα Δ

Κλάσεις που Δημιουργήθηκαν

```
class Line:
    def __init__(self, start, end, id=None):
        self.start = start
        self.end = end
        self.id = id

    def __lt__(self, other):
        if self.start[1] == other.start[1]:
            return self.start[0] < other.start[0]
        return self.start[1] < other.start[1]

    def __str__(self):
        return "Line {} from {} to {}".format(self.id, self.start, self.end)

    def slope(self):
        return (self.end[1] - self.start[1]) / (self.end[0] - self.start[0])

def slope(p1, p2):
    return (p2[1] - p1[1]) / (p2[0] - p1[0])


class Event:
    x = float
    y = float
    lines = [Line]
    def __init__(self, x, y, lines):
        self.x = x
        self.y = y
        self.lines = lines

    def __lt__(self, other):
        if self.y == other.y:
            return self.x < other.x
        return self.y < other.y

    def is_left_endpoint(self):
        return self.lines[0].start == (self.x, self.y)
```

```

class Polygon:

    points = [(float,float)]

    def __init__(self, *points):
        self.points = points

    def lines(self):
        lines=[Line]
        for i in range(len(self.points)-1):
            lines.append(Line(self.points[i], self.points[i+1]))
        return lines

```

Οι Αλγόριθμοι

```

# Η συνάρτηση ελέγχει αν υπάρχει τομή μεταξύ των ευθυγράμμων τμημάτων (p1, p2) και (p3, p4)
def intersect(line1, line2):

    x1,y1 = line1.start
    x2,y2 = line1.end
    x3,y3 = line2.start
    x4,y4 = line2.end

    denom = (y4-y3)*(x2-x1) - (x4-x3)*(y2-y1)

    if denom == 0: # Τότε τα τμήματα είναι παράλληλα
        return None
    ua = ((x4-x3)*(y1-y3) - (y4-y3)*(x1-x3)) / denom

    if ua < 0 or ua > 1: # Το σημείο τομής των ευθυών είναι πέρα από τα ευθύγραμμα
        return None
    ub = ((x2-x1)*(y1-y3) - (y2-y1)*(x1-x3)) / denom

    if ub < 0 or ub > 1: # Το σημείο τομής των ευθυών είναι πέρα από τα ευθύγραμμα
        return None
    x = x1 + ua * (x2-x1)
    y = y1 + ua * (y2-y1)
    return (x,y)

```

```

def pol_interesect_naive(polygon1, polygon2):

    n_lines1 = len(polygon1)-1
    n_lines2 = len(polygon2)-1
    points = []

    for i in range(n_lines1) :
        for j in range(n_lines2) :
            temp =
            intersect(Line(polygon1[i],polygon1[i+1]),Line(polygon2[j], polygon2[j+1]))
            if temp is not None:
                points.append(temp)

    return points


def sweep_line_intersect(polygon1, polygon2):

    # Get lines from polygons
    lines1 = polygon1.lines()
    lines2 = polygon2.lines()

    # Create events from all line endpoints
    events = []
    for line in lines1 + lines2:
        events.append(Event(line.start[0], line.start[1], True, [line]))
        events.append(Event(line.end[0], line.end[1], False, [line]))

    # Sort events by x coordinate
    events.sort()

    # Active line segments
    active_lines = []

    # Intersection points
    intersections = []

    # Sweep line algorithm
    for event in events:

        if event.is_left_endpoint():
            # Insert segment in active lines
            active_lines.append(event.line)

        else:
            # Remove segment from active lines
            active_lines.remove(event.line)

        # Check intersections between active segments
        for l1 in active_lines:
            for l2 in active_lines:
                if l1 is not l2:
                    temp = intersect(l1, l2)
                    if temp:
                        intersections.append(intersect)

    return intersections

```