ΕΘΝΙΚΟ & ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΤΜΗΜΑ ΦΥΣΙΚΗΣ

ΤΟΜΕΑΣ ΗΛΕΚΤΡΟΝΙΚΗΣ ΦΥΣΙΚΗΣ ΚΑΙ ΣΥΣΤΗΜΑΤΩΝ

Πτυχιακή Εργασία

Μελέτη, σχεδίαση και υλοποίηση επαυξημένων δομών δεδομένων με βάση τα δυαδικά δένδρα.

Αναστάσιος Γκίκας

ΑΜ: 1110201500037

Επιβλέπων:

Διονύσιος Ι. Ρεΐσης

Καθηγητής

Αθήνα 2022

Περιεχόμενα

Εισαγωγή

Γνωρίζοντας πώς να αξιοποιούμε τις βασικές δομές δεδομένων, όπως η απλά/διπλά συνδεδεμένη λίστα, τα δυαδικά δέντρα αναζήτησης κλπ., είμαστε σε θέση να ανταπεξέλθουμε σε αρκετές καταστάσεις. Υπάρχουν όμως περιπτώσεις που απαιτούν -λίγο ή πολύ- πιο σύνθετες δομές δεδομένων. Για καλή μας τύχη, το πιο αποδοτικό στις περισσότερες από αυτές τις περιπτώσεις δεν είναι η δημιουργία μιας εντελώς διαφορετικής δομής από την αρχή, αλλά ο εμπλουτισμός-επαύξηση μιας γνωστής δομής, με τρόπο που να καλύπτει τις ανάγκες μας.

Στην παρούσα εργασία θα ασχοληθούμε με ακριβώς αυτήν την επαύξηση γνωστών δομών δεδομένων. Συγκεκριμένα, στο **Κεφάλαιο 1** θα εξετάσουμε μια θεωρητική θεμελίωση της τεχνικής αυτής και στη συνέχεια θα την εξειδικεύσουμε στη δομή *red-black δέντρου*. Στα **Κεφάλαια 2** και **3** θα μελετήσουμε την εφαρμογή της σε δύο περιπτώσεις, όπου θα γίνει αντιληπτή η χρησιμότητα της επαύξησης δομών δεδομένων. Στις εφαρμογές αυτές θα γίνεται χρήση *ψευδοκώδικα*, ενώ στο **Παράρτημα** παρατίθεται η υλοποίηση αυτών σε γλώσσα προγραμματισμού *C++*.

1 Επαυξημένες δομές δεδομένων

1.1 Θεμελίωση της μεθόδου

Ξεκινώντας να εξετάζουμε από γενική σκοπιά την επαύξηση μιας βασικής δομής δεδομένων, μπορούμε να διακρίνουμε 4 κεντρικά βήματα-οδηγούς για την αποτελεσματική ολοκλήρωση αυτής της διαδικασίας:

1. Επιλογή της βασικής δομής δεδομένων, υποκείμενης προς επαύξηση
2. Καθορισμός επιπλέον πληροφορίας για αντιστοίχιση σε κάθε στοιχείο της υποκείμενης δομής
3. Επαλήθευση της διατήρησης αυτής της επιπλέον πληροφορίας για τις βασικές λειτουργίες τροποποίησης της υποκείμενης δομής
4. Ανάπτυξη νέων λειτουργιών

Φυσικά, δεν είμαστε υποχρεωμένοι να ακολουθούμε με αυστηρή σειρά αυτά τα βήματα. Αντίθετα, είναι ιδιαίτερα αποτελεσματικό να προχωράμε παράλληλα κάποια βήματα -αν όχι όλα-, καθώς συχνά η πρόοδος σε ένα βήμα είτε καθιστά ευκολότερη είτε διορθώνει την πρόοδο σε ένα άλλο.

Πιο αναλυτικά:

- ➔ Για το βήμα 1, η επιλογή της υποκείμενης δομής δεδομένων έγκειται στην αναγνώριση των λειτουργιών που ήδη ανήκουν στη βασική δομή, οι οποίες θα φανούν χρήσιμες στην επεξεργασία των δεδομένων που διαθέτουμε σε κάθε περίπτωση.
- ➔ Για το βήμα 2, η πληροφορία που θα προσθέσουμε στο κάθε στοιχείο της βασικής δομής θα είναι τέτοια ώστε, με τη χρήση της, να γίνονται οι λειτουργίες του βήματος 4 πιο αποδοτικές (κυρίως από άποψη χρόνου). Αξίζει να σημειωθεί ότι αυτή η επιπρόσθετη πληροφορία μπορεί να είναι οτιδήποτε, ακόμα και τύπου δείκτη.
- ➔ Για το βήμα 3, οφείλουμε να ελέγχουμε συνεχώς τις μεταβολές που υφίστανται οι επιπλέον πληροφορίες κατά την εκτέλεση κάθε βασικής λειτουργίας της υποκείμενης δομής, και έπειτα να εκτελούμε τις απαραίτητες προσθήκες-διορθώσεις ώστε να διατηρούνται τόσο τα σωστά χαρακτηριστικά της μεταβεβλημένης πληροφορίας, όσο και ο χρόνος εκτέλεσης αυτών των λειτουργιών.
- ➔ Για το βήμα 4, αναπτύσσουμε τις νέες λειτουργίες, η αποδοτικότητα των οποίων είναι ο λόγος που δημιουργήσαμε εξ αρχής την πρόσθετη πληροφορία.

1.2 Επαύξηση red-black δέντρων

Αρκετές είναι οι περιπτώσεις στις οποίες είναι εξαιρετικά βολικό να χρησιμοποιούμε σαν υποκείμενη δομή δεδομένων τα *red-black δέντρα*. Ο λόγος είναι ότι, όπως θα δούμε στο παρακάτω θεώρημα, κατά την εισαγωγή και διαγραφή στοιχείων ενός red-black δέντρου διατηρούνται αποδοτικά ορισμένοι τύποι πρόσθετης πληροφορίας.

Θεώρημα 1.1 (Επαύξηση red-black δέντρου)

Έστω *f* μία πληροφορία που επαυξάνει ένα red-black δέντρο *T* με *n* κόμβους, τέτοια ώστε η τιμή της για κάθε κόμβο *x* να εξαρτάται μόνο από τις πληροφορίες στους κόμβους *x*, *x.left* και *x.right*, συμπεριλαμβανομένων ενδεχομένως και των *x.left.f* και *x.right.f*. Τότε η τιμή της *f*

διατηρείται σε όλους τους κόμβους του *T* κατά την εισαγωγή και διαγραφή, χωρίς να επηρεάζεται ασυμπτοτικά ο $O(\lg n)$ χρόνος εκτέλεσης των λειτουργιών αυτών.

Όπως φαίνεται, η χρήση του θεωρήματος αυτού μας προσφέρει μια σημαντική προώθηση για το βήμα 3!

2 Δυναμικά order statistics

Ως ***i*-οστό order statistic** ενός συνόλου με *n* στοιχεία, όπου *i* ∈ {1, 2, …, n}, ορίζεται το στοιχείο του συνόλου με το *i*-οστό μικρότερο key. Ως **rank** ενός στοιχείου του συνόλου, ορίζεται η θέση του στη γραμμική διάταξη του συνόλου. Υπάρχουν αλγόριθμοι που βρίσκουν τα ζητούμενα αυτά για μη διατεταγμένα σύνολα σε χρόνο $O(n)$. Στο Κεφάλαιο αυτό θα μελετήσουμε πώς μπορούμε να βελτιώσουμε το χρόνο αυτό, επινοώντας μια νέα δομή δεδομένων, τα *order-statistic trees*.

2.1 Δημιουργώντας ένα order-statistic tree

Όπως είδαμε στο Κεφάλαιο 1, για τη δημιουργία μιας δομής δεδομένων μέσω επαύξησης, πρέπει καταρχάς να επιλέξουμε ποια γνωστή δομή δεδομένων σκοπεύουμε να επαυξήσουμε. Αφού λοιπόν μια δυναμική order-statistics δομή χρειάζεται σίγουρα τις λειτουργίες της εισαγωγής και της διαγραφής, ας δοκιμάσουμε να θέσουμε σαν υποκείμενη δομή δεδομένων ένα red-black tree, με την ελπίδα να μπορέσουμε να αξιοποιήσουμε το Θεώρημα 1.1.

Ήδη καλύψαμε το βήμα 1 της διαδικασίας επαύξησης. Συνεχίζοντας, σε κάθε node του red-black tree προσθέτουμε ένα επιπλέον attribute, το ***node.size***. Το attribute αυτό περιέχει *το πλήθος των nodes στο subtree με root το node*, συμπεριλαμβανομένου και του node. Αν λοιπόν θέσουμε το size attribute του sentinel, δηλαδή το *T.nil.size*, ίσο με 0 τότε θα έχουμε την ταυτότητα:

$$node.size = node.left.size + node.right.size + 1$$

Εξίσωση 2-1

Όπως φαίνεται, το size attribute που προσθέσαμε σε κάθε red-black node εξαρτάται μόνο από τις πληροφορίες στους απογόνους του node. Έτσι, μας δίνεται η δυνατότητα να αξιοποιήσουμε το Θεώρημα 1.1 και να έχουμε σφραγίσει τα βήματα 2 και 3.

2.2 Διατήρηση πληροφορίας

Προχωρώντας στη διαπίστωση του βήματος 3, εξετάζουμε τις λειτουργίες εισαγωγής και διαγραφής του red-black tree.

Αρχικά ας εστιάσουμε στη φάση της εισαγωγής όπου ξεκινάμε από το root και κατεβαίνουμε στο δέντρο μέχρι να εντοπίσουμε τον κόμβο που θα έχει ως νέο παιδί τον εισαγόμενο κόμβο. Στη φάση αυτή αρκεί για κάθε κόμβο που συναντάμε στη διαδρομή να αυξάνουμε το

size attribute του κατά 1 (Method 1) με κόστος πολυπλοκότητας $O(1)$. Εφόσον στη διαδρομή υπάρχουν $O(\lg n)$ κόμβοι, το κόστος διατήρησης του size στη φάση πριν ξεκινήσουν οι περιστροφές των κόμβων είναι κι αυτό $O(\lg n)$.

Αντίστοιχα για τη διαγραφή, αφού αφαιρέσουμε τον προς διαγραφή κόμβο και πριν προχωρήσουμε στις περιστροφές, εκτελούμε μία ανάβαση από τον κατώτερο κόμβο που άλλαξε θέση κατά τη διαγραφή μέχρι και το root. Για κάθε κόμβο στη διαδρομή εφαρμόζουμε την Εξίσωση 2-1 (Method 2) και κατ' επέκταση, για τον ίδιο λόγο όπως και στην εισαγωγή, το κόστος διατήρησης του size είναι $O(\lg n)$.

Αφού λοιπόν πραγματοποιηθεί η εισαγωγή/διαγραφή ενός κόμβου, γίνεται έλεγχος και διατήρηση των red-black ιδιοτήτων σε κάθε κόμβο της αναβατικής διαδρομής μέχρι το root και εκτελούνται περιστροφές, για κάθε κόμβο. Έπειτα από μία περιστροφή, οι μόνοι κόμβοι των οποίων το size χρειάζεται διόρθωση είναι το παλιό και το νέο root του subtree. Η διόρθωση αυτή επιτυγχάνεται με τις εξής εντολές (Method 3):

$$newRoot.size = oldRoot.size$$
$$oldRoot.size = oldRoot.left.size + oldRoot.right.size + 1 \text{ (Εξίσωση 2-1)}$$

Βλέπουμε ότι η πολυπλοκότητα διόρθωσης του size για κάθε περιστροφή είναι $O(1)$. Δεδομένου ότι για κάθε κόμβο εκτελούνται το πολύ 3 περιστροφές και οι κόμβοι μέχρι το root είναι $O(\lg n)$, η διόρθωση του size μετά την εισαγωγή/διαγραφή είναι κι αυτή $O(\lg n)$.

Συνολικά, έχουμε πολυπλοκότητα $O(\lg n)$ πριν και $O(\lg n)$ μετά τις περιστροφές για τη διατήρηση του size, και $O(\lg n)$ για την εκτέλεση της ίδιας της εισαγωγής/διαγραφής, άρα $O(\lg n)$ ο χρόνος εκτέλεσης των βασικών λειτουργιών του order-statistic tree, ίδιος με τον αντίστοιχο χρόνο στο red-black tree (Θεώρημα 1.1)!

2.3 Υλοποίηση ζητούμενων λειτουργιών

Ήρθε λοιπόν η ώρα να αξιοποιήσουμε τη νέα πληροφορία size που προσθέσαμε σε κάθε κόμβο του red-black tree, με σκοπό να δημιουργήσουμε δύο νέες λειτουργίες ικανές να ανακτούν το ***i^th^ order-statistic*** του συνόλου και το ***rank*** ενός στοιχείου, με βελτιωμένη απόδοση σε σχέση με το $O(n)$.

Η παρακάτω διαδικασία $select(i, node)$ (Method 4) επιστρέφει τον κόμβο με το i-οστό μικρότερο key στο subtree με root το node. Έτσι, για να βρούμε το i-οστό order statistic του δέντρου T καλούμε την select(i, T.root).

```
select(i, node)
    r = node->left->size + 1
    if i == r return node
    if i < r return select(i, node->left)
    return select(i – r, node->right)
```

Όπως φαίνεται, η διαδικασία αυτή εκμεταλλεύεται ένα πόρισμα του ορισμού του i-οστού order statistic, δηλαδή ότι είναι το στοιχείο του οποίου το key είναι μεγαλύτερο από τα keys ακριβώς i άλλων στοιχείων. Έτσι, κατεβαίνει αναδρομικά από το root μέχρι τον ζητούμενο κόμβο, άρα ο χρόνος εκτέλεσης είναι $O(\lg n)$, όσοι και οι κόμβοι στην καταβατική διαδρομή.

Αντίστροφα, η επόμενη διαδικασία $rank(key, node)$ (Method 5) επιστρέφει τη θέση του key στη γραμμική διάταξη των keys όλων των στοιχείων του subtree με root το node.

Επομένως η κλήση της rank(key, T.root) απαντά στο ερώτημα, ποιο order statistic του δέντρου T έχει σαν κλειδί το key που δίνεται σαν παράμετρος.

```
rank(key, node)
    if key == node->key return node->left->size + 1
    if key < node->key return rank (key, node->left)
    return 1 + node->left->size + rank(key, node->right)
```

Σε κάθε κλήση αυτής της διαδικασίας υπολογίζεται αναδρομικά το πλήθος των nodes στο subtree με root το node, των οποίων τα keys είναι μικρότερα από το key. Η διαδρομή μέχρι τον κόμβο με κλειδί το key γίνεται ακριβώς όπως και σε ένα red-black tree, επομένως η πολυπλοκότητα της rank(key, node) είναι $O(\lg n)$.

2.4 Συνοψίζοντας

Στόχος μας σε αυτό το Κεφάλαιο ήταν να βρούμε έναν τρόπο με τον οποίο να υπολογίζουμε το i-οστό order statistic ενός συνόλου και το rank ενός στοιχείου του συνόλου, σε χρόνο καλύτερο από $O(n)$. Έτσι ξεκινήσαμε να κατασκευάζουμε ένα order-statistic tree, επαυξάνοντας ένα red-black tree. Αρχικά προσθέσαμε την πληροφορία size στα attributes του κάθε κόμβου του red-black tree. Έπειτα φροντίσαμε να κάνουμε τις απαραίτητες διορθώσεις στις βασικές λειτουργίες του red-black tree ώστε να διατηρείται το size σε κάθε κόμβο που επηρεάζεται, χωρίς να αυξάνεται η πολυπλοκότητα των λειτουργιών αυτών. Τέλος, χρησιμοποιώντας το size, καταφέραμε με δύο νέες λειτουργίες να ανακτήσουμε τα αρχικά μας ζητούμενα σε χρόνο $O(\lg n)$.

3 Interval trees

Σε αυτό το Κεφάλαιο θα επιχειρήσουμε να επαυξήσουμε ένα red-black tree ώστε να υποστηρίζει χρονικά αποδοτικές λειτουργίες σε δυναμικά σύνολα (αριθμητικών) διαστημάτων. Συγκεκριμένα, τα σύνολα $\{t \in \mathbb{R} : t_1 \leq t \leq t_2\}$ ονομάζονται *κλειστά* διαστήματα, συμβολίζονται $[t_1, t_2]$ και είναι αυτά που θα μας απασχολήσουν. Η βασική λειτουργία που επιθυμούμε να αναπτύξουμε είναι η εύρεση ενός διαστήματος του συνόλου που έχει μη κενή **επικάλυψη (overlap)** με ένα άλλο δεδομένο διάστημα. Αυτό θα το πετύχουμε δημιουργώντας τα *interval trees*.

3.1 Δημιουργώντας ένα interval tree

Μπορούμε να αναπαραστήσουμε ένα διάστημα $[t_1, t_2]$ ως ένα αντικείμενο i με attributes τα $i.low = t_1$ και $i.high = t_2$. Οι μόνες δύο περιπτώσεις, άμεσα αντιληπτές, κατά τις οποίες δύο διαστήματα i και j δεν έχουν overlap, είναι αυτές όπου ολόκληρο το i βρίσκεται είτε πριν είτε μετά από ολόκληρο το j, δηλαδή όταν $i.high < j.low$ **ή** $j.high < i.low$. Έτσι, μπορούμε να πούμε ότι δύο διαστήματα i και j έχουν overlap όταν ικανοποιούνται **και οι δύο αντίθετες** συνθήκες, δηλαδή όταν $i.low \leq j.high$ **και** $j.low \leq i.high$ (Method 6).

Επιλέγουμε ένα red-black tree προς επαύξηση, κάθε node του οποίου περιέχει ένα διάστημα *node.i*, ενώ το *node.key* είναι ίσο με το *node.i.low*, δηλαδή τα στοιχεία του συνόλου

ταξινομούνται με βάση το *low* attribute του διαστήματος κάθε στοιχείου. Επιπλέον, σε κάθε node προσθέτουμε την πληροφορία $node.max$ η οποία εκφράζει το μέγιστο $high$ όλων των διαστημάτων στο subtree με root το node. Αφού κάθε νέο node είναι φύλλο, το $node.max$ αρχικοποιείται στην τιμή $node.i.high$. Ισχύει έτσι η ταυτότητα (Method 7):

$$node.max = \max(node.i.high, node.left.max, node.right.max)$$

Εξίσωση 3-1

Βλέπουμε πάλι ότι η πληροφορία στο node εξαρτάται μόνο από το node και τους απογόνους του, άρα από το Θεώρημα 1.1 προβλέπουμε με βεβαιότητα ότι η εισαγωγή/διαγραφή μπορεί να διατηρήσει το max attribute χωρίς να επηρεαστεί ο $O(\lg n)$ χρόνος εκτέλεσης.

3.2 Διατήρηση πληροφορίας

Στο σημείο αυτό εργαζόμαστε όμοια με την παράγραφο 2.2 για την εισαγωγή και διαγραφή. Πιο αναλυτικά, στην εισαγωγή ενός node, καθώς κατεβαίνουμε από το T.root και για κάθε aux κόμβο στη διαδρομή, εκχωρούμε την τιμή του $node.max$ στο $aux.max$ αν $aux.max < node.max$ (Method 8). Για τη διαγραφή, πριν προχωρήσουμε στις περιστροφές, πραγματοποιούμε μια ανάβαση αντίστοιχη με αυτήν που είδαμε στην παράγραφο 2.2, εφαρμόζοντας κάθε φορά την Εξίσωση 3-1 (Method 9). Έπειτα, στις περιστροφές δε δρούμε διαφορετικά σε σχέση με την παράγραφο 2.2, προφανώς πέραν από το ότι η εξίσωση για τη διόρθωση του $oldRoot.max$ είναι η Εξίσωση 3-1 (Method 10). Και στις δύο περιπτώσεις η διόρθωση κοστίζει $O(1)$ επομένως πράγματι δεν αυξάνεται ο χρόνος $O(\lg n)$ της εισαγωγής/διαγραφής με τη διατήρηση του max.

3.3 Υλοποίηση ζητούμενων λειτουργιών

Είμαστε έτοιμοι δούμε πώς χρησιμοποιείται το max attribute στην παρακάτω διαδικασία $overlap(i, node)$ (Method 11) που μας βοηθά να ερευνήσουμε την ύπαρξη διαστήματος στο subtree με root το node, το οποίο να έχει επικάλυψη με το i. Καλώντας την overlap(i, T.root), αν υπάρχει διάστημα στο red-black tree T που έχει επικάλυψη με το i ανακτούμε το διάστημα αυτό, ενώ αν δεν υπάρχει τέτοιο διάστημα μας επιστρέφεται το T.nil.

```
overlap(i, node)
    if node == nil or i overlaps node.i
        return node
    if node.left = nil and node.left.max < i.low
        return overlap(i, node.right)
    return overlap(i, node.left)
```

Ας σχολιάσουμε αρχικά τη δεύτερη "if" δήλωση, στην οποία όπως παρατηρούμε εξετάζεται η συνθήκη $node.left.max < i.low$, ή -ισοδύναμα- εξετάζεται αν το μεγαλύτερο από τα $node.i.high$ του αριστερού subtree είναι μικρότερο από το $i.low$. Όταν η συνθήκη αυτή επαληθεύεται, τότε ξέρουμε ότι στο αριστερό subtree δεν υπάρχει κανένα διάστημα $node.i$ που να ικανοποιεί τη συνθήκη $node.i.high \geq i.low$, δηλαδή τη μία από τις δύο συνθήκες για να έχουν δύο διαστήματα overlap. Επομένως, αν υπάρχει τουλάχιστον ένα διάστημα που να επικαλύπτει το i τότε αυτό θα βρίσκεται με βεβαιότητα στο δεξί υποδέντρο του node. Αντίστοιχα, αν η προαναφερόμενη συνθήκη δεν επαληθεύεται τότε το εν δυνάμει διάστημα επικάλυψης θα βρίσκεται στο αριστερό υποδέντρο. Τέλος, όπως φαίνεται στις πρώτες δύο

γραμμές του σώματος της διαδικασίας, η διαδρομή μας οδήγησε είτε στο ζητούμενο διάστημα αν υπάρχει, είτε σε nil αν δεν υπάρχει, οπότε και επιστρέφεται το παρόν node. Προφανώς η αναζήτηση ακολουθεί καταβατική διαδρομή ύψους $O(\lg n)$, άρα και ο χρόνος εκτέλεσης είναι $O(\lg n)$.

3.4 Συνοψίζοντας

Ακολουθώντας παρόμοια στρατηγική με του προηγούμενου Κεφαλαίου, έτσι και σε αυτό επιχειρήσαμε να δημιουργήσουμε μια δομή που μας βοηθά να χειριζόμαστε αριθμητικά διαστήματα και να εξετάζουμε επικαλύψεις μεταξύ τους, πάντα με χρονική αποδοτικότητα. Η δομή αυτή ονομάστηκε interval tree και είναι στην ουσία της ένα red-black tree που έχει διαστήματα σαν δεδομένα των κόμβων. Αφού λοιπόν εγγυηθήκαμε τη διατήρηση της επιπλέον πληροφορίας max κατά την εισαγωγή και διαγραφή, την αξιοποιήσαμε για να αναπτύξουμε τη λειτουργία $overlap(i, node)$. Η λειτουργία αυτή μας προσφέρει ακριβώς αυτό που αναζητούσαμε, το διάστημα που έχει επικάλυψη με το i, σε χρόνο χαρακτηριστικό των red-black trees, δηλαδή $O(\lg n)$!

Επίλογος

Ένα από τα πολλά ενδιαφέροντα της Επιστήμης της Πληροφορικής και των Υπολογιστών, βρίσκεται στη συνεχή προσπάθεια να μας κάνει τη «ζωή» πιο εύκολη. Στην εργασία αυτή εξετάσαμε ορισμένους τρόπους με τους οποίους μπορούμε εύκολα και αποδοτικά να επαυξήσουμε γνωστές δομές δεδομένων, εν προκειμένω red-black trees, για να αναπτύξουμε αποδοτικές λειτουργίες, αποφεύγοντας έτσι την εκ του μηδενός κατασκευή μιας ολοκαίνουριας δομής.

Φυσικά, στην πορεία της εν λόγω επαύξησης είναι υποχρέωσή μας να είμαστε ιδιαίτερα προσεκτικοί ως προς τη συνοχή της νέας δομής με την υποκείμενη προς επαύξηση δομή. Όπως είδαμε, κατά την επαύξηση προσθέσαμε μία νέα πληροφορία στους κόμβους του red-black tree. Κατά την εκτέλεση των βασικών λειτουργιών του red-black tree η πληροφορία αυτή χρειάστηκε διόρθωση, την οποία και πετύχαμε πολύ αποδοτικά, χωρίς να επηρεάσουμε τον ικανοποιητικό λογαριθμικό χρόνο εκτέλεσης των λειτουργιών.

Εν τέλει, αξιοποιώντας την προαναφερόμενη πληροφορία, καταφέραμε να αναπτύξουμε τις επιθυμητές λειτουργίες, με σημαντικά μικρό κώδικα και φυσικά λογαριθμική πολυπλοκότητα. Ας κρατήσουμε λοιπόν το εξής από την παρούσα εργασία, πριν αποφασίσουμε να στρώσουμε το δικό μας δρόμο για να φτάσουμε σε έναν προορισμό, ας εξετάσουμε πρώτα μήπως ένα μεγάλο μέρος του δρόμου είναι ήδη στρωμένο!

Παράρτημα

Παρακάτω παρατίθενται οι methods σε C++ που υλοποιούν τους αλγορίθμους που μελετήθηκαν κατά τη διάρκεια της παρούσας εργασίας. Για τον πλήρη κώδικα παρακαλώ ακολουθήστε τον σύνδεσμο https://github.com/tasosgkikas/augmented-rb-trees όπου παρέχονται 4 αρχεία και μία περιγραφή.

Method 1

```
void insert(nodePtr node)
{
    nodePtr parent = nil, aux = root;

    while (aux != nil)
    {
        aux->size += 1;  // size maintenance
        parent = aux;
        if (node->key < aux->key) aux = aux->left;
        else aux = aux->right;
    }
    node->p = parent;

    if (parent == nil) root = node;
    else if (node->key < parent->key) parent->left = node;
    else parent->right = node;

    insertFixup(node);
}
```

Method 2

```
void remove(nodePtr node)
{
    nodePtr sub1, sub2;

    Color color = node->color;

    // remove algorithm
    if (node->left == nil) ···
    else if (node->right == nil) ···
    else ···

    // size attribute maintenance
    nodePtr aux = sub2;
    while (aux != root)
    {
        aux = aux->p;
        aux->size = aux->left->size + aux->right->size + 1;
    }

    if (color == black) removeFixup(sub2);

    delete node;
}
```

Method 3

```
void leftRot(nodePtr oldRoot)
{
    nodePtr newRoot = oldRoot->right;
    oldRoot->right = newRoot->left;
    if (newRoot->left != nil)
        newRoot->left->p = oldRoot;
    newRoot->p = oldRoot->p;
    if (oldRoot->p == nil)
        root = newRoot;
    else if (oldRoot == oldRoot->p->left)
        oldRoot->p->left = newRoot;
    else
        oldRoot->p->right = newRoot;
    newRoot->left = oldRoot;
    oldRoot->p = newRoot;

    // size attribute maintenance
    newRoot->size = oldRoot->size;
    oldRoot->size = oldRoot->left->size
                  + oldRoot->right->size
                  + 1;
}
```

```
void rightRot(nodePtr oldRoot)
{
    nodePtr newRoot = oldRoot->left;
    oldRoot->left = newRoot->right;
    if (newRoot->right != nil)
        newRoot->right->p = oldRoot;
    newRoot->p = oldRoot->p;
    if (oldRoot->p == nil)
        root = newRoot;
    else if (oldRoot == oldRoot->p->left)
        oldRoot->p->left = newRoot;
    else
        oldRoot->p->right = newRoot;
    newRoot->right = oldRoot;
    oldRoot->p = newRoot;

    // size attribute maintenance
    newRoot->size = oldRoot->size;
    oldRoot->size = oldRoot->left->size
                  + oldRoot->right->size
                  + 1;
}
```

Method 4

```
nodePtr select(nodePtr node, unsigned i)
{
    if (node == nil) return nil;

    unsigned r = node->left->size + 1;

    if (i == r) return node;
    if (i < r) return select(node->left, i);
    else return select(node->right, i - r);
}
```

Method 5

```
unsigned rank_top_down(key_t key, nodePtr node)
{
    if (key == node->key) return node->left->size + 1;
    if (key < node->key) return rank_top_down(key, node->left);
    return 1 + node->left->size + rank_top_down(key, node->right);
}
```

Method 6

```
struct interval
{
    key_t key, low, high;
    interval(key_t low, key_t high):
    interval() {}
    bool overlap(interval& i) {
        return (
                low <= i.high &&
            i.low <=   high
        );
    }
    bool operator==(interval& i) {···
};
```

Method 7

```
void updateMax()
{
    if (left != nil && right != nil)
        max = getMax(i.high, getMax(left->max, right->max));
    else if (left != nil)
        max = getMax(i.high, left->max);
    else if (right != nil)
        max = getMax(i.high, right->max);
    else
        max = i.high;
}
```

Method 8

```
void insert(nodePtr node)
{
    nodePtr parent = nil, aux = root;

    while (aux != nil)
    {
        // max attribute maintenance
        if (aux->max < node->max)
            aux->max = node->max;

        // aux pointer propagation
        parent = aux;
        if (node->key < aux->key) aux = aux->left;
        else aux = aux->right;
    }
    node->p = parent;

    if (parent == nil) root = node;
    else if (node->key < parent->key) parent->left = node;
    else parent->right = node;

    insertFixup(node);
}
```

Method 9

```
void remove(nodePtr node)
{
    nodePtr sub1, sub2;

    Color color = node->color;

    // remove algorithm
    if (node->left == nil)···
    else if (node->right == nil) ···
    else···

    // max attribute maintenance
    nodePtr aux = sub2;
    while (aux != root)
    {
        aux = aux->p;
        aux->updateMax();
    }

    if (color == black) removeFixup(sub2);

    delete node;
}
```

Method 10

```
void leftRot(nodePtr oldRoot)
{
    nodePtr newRoot = oldRoot->right;
    oldRoot->right = newRoot->left;
    if (newRoot->left != nil)
        newRoot->left->p = oldRoot;
    newRoot->p = oldRoot->p;
    if (oldRoot->p == nil)
        root = newRoot;
    else if (oldRoot == oldRoot->p->left)
        oldRoot->p->left = newRoot;
    else
        oldRoot->p->right = newRoot;
    newRoot->left = oldRoot;
    oldRoot->p = newRoot;
    // max attribute maintenance
    newRoot->max = oldRoot->max;
    oldRoot->updateMax();
}
```

```
void rightRot(nodePtr oldRoot)
{
    nodePtr newRoot = oldRoot->left;
    oldRoot->left = newRoot->right;
    if (newRoot->right != nil)
        newRoot->right->p = oldRoot;
    newRoot->p = oldRoot->p;
    if (oldRoot->p == nil)
        root = newRoot;
    else if (oldRoot == oldRoot->p->left)
        oldRoot->p->left = newRoot;
    else
        oldRoot->p->right = newRoot;
    newRoot->right = oldRoot;
    oldRoot->p = newRoot;
    // max attribute maintenance
    newRoot->max = oldRoot->max;
    oldRoot->updateMax();
}
```

Method 11

```
nodePtr overlap(interval& i, nodePtr node)
{
    if (node == nil || i.overlap(node->i))
        return node;
    if (node->left == nil || node->left->max < i.low)
        return overlap(i, node->right);
    return overlap(i, node->left);
}
```