Ατομική Διπλωματική Εργασία

ΑΝΑΣΤΡΕΨΙΜΕΣ ΥΛΟΠΟΙΗΣΕΙΣ ΔΕΝΤΡΙΚΩΝ ΔΟΜΩΝ ΔΕΔΟΜΕΝΩΝ

Αγαθοκλής Βακανάς

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Μάιος 2017

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Αναστρέψιμες Υλοποιήσεις Δεντρικών Δομών Δεδομένων

Αγαθοκλής Βακανάς

Επιβλέπουσα Καθηγήτρια
Άννα Φιλίππου

Η Ατομική Διπλωματική Εργασία υποβλήθηκε προς μερική εκπλήρωση των απαιτήσεων απόκτησης του πτυχίου Πληροφορικής του Τμήματος Πληροφορικής του Πανεπιστημίου Κύπρου

Μάιος 2017

Ευχαριστίες

Καταρχάς θα ήθελα να ευχαριστήσω θερμά την επιβλέπουσα καθηγήτρια μου Δρ. Άννα Φιλίππου που με εμπιστεύτηκε και μου έδωσε την δυνατότητα να εργαστώ στο συγκεκριμένο θέμα καθώς επίσης και για τη συνεχή στήριξη, καθοδήγηση και ενθάρρυνση που μου παρείχε κατά την προετοιμασία της Ατομικής Διπλωματικής Εργασίας. Η πόρτα της ήταν πάντοτε ανοικτή οποιαδήποτε ώρα και στιγμή. Χωρίς τη βοήθεια της η ολοκλήρωση της διπλωματικής αυτής εργασίας θα ήταν αδύνατη.

Παράλληλα, θα ήθελα να ευχαριστήσω τους γονείς μου, Κώστα και Αντρούλλα, για τη θερμή υποστήριξη τους όλο αυτό το διάστημα. Η δική τους βοήθεια, στήριξη και αγάπη υπήρξε βάλσαμο στην όλη αυτή προσπάθεια. Οι συμβουλές και οι καθοδηγήσεις τους πάντα θα χαίρουν από εμένα άκρας εκτίμησης και σημασίας.

Επιπλέον θα ήθελα να ευχαριστήσω τα αδέλφια μου, Δέσποινα, Δημήτρη και Ελένη, για την ανιδιοτελή βοήθεια που μου πρόσφεραν σ' όλα τα χρόνια της φοίτησης μου στο Πανεπιστήμιο Κύπρου και ειδικά το τελευταίο διάστημα.

Σε αυτό το σημείο, θα ήθελα να ευχαριστήσω από τα βάθη της καρδίας μου την αδελφή μου Δέσποινα, η οποία αποτέλεσε το στήριγμα και τη πυξίδα μου. Είναι το άτομο που ήταν δίπλα μου σε όλες τις δύσκολες στιγμές της ζωής μου. Η δική της σταδιοδρομία που έμαθε πως στα δύσκολα δεν τα παρατάμε, η σκληρή δουλειά ανταμείβεται και δεν υπάρχουν όρια αλλά στόχοι. Στόχοι, που εμείς καθορίζουμε με βάση τα δικά μας ενδιαφέροντα και προσδοκίες, για την επίτευξη των οποίων αρκεί απλά να το πιστέψουμε.

Κάθε τέλος σηματοδοτεί την έναρξη ενός καινούργιου ταξιδιού. Ευχαριστώ όλους αυτούς, γνωστούς και άγνωστους, που υπήρξαν συνταξιδιώτες μου τα τέσσερα αυτά χρόνια της φοίτησης μου στο τμήμα Πληροφορικής του Πανεπιστήμιου Κύπρου και ευελπιστώ μια μέρα να σταθώ άξια των προσδοκιών τους.

Αγαθοκλής Βακανάς

Μάιος 2017

Περίληψη

Ο τομέας του αναστρέψιμου προγραμματισμού είναι ακόμη στα αρχικά στάδια με αποτέλεσμα να χρειάζεται ακόμη αρκετή θεμελιώδης έρευνα μέχρι να μπορεί να συγκριθεί με τις υπάρχουσες τεχνικές προγραμματισμού. Στα πλαίσια της παρούσας Ατομικής Διπλωματικής Εργασίας, επικεντρωνόμαστε στη δημιουργία αναστρέψιμων δεντρικών δομών και των σχετικών τους συναρτήσεων.

Ένας βασικός προβληματισμός σχετικά με τον Αναστρέψιμο Προγραμματισμό είναι η ανάγκη φύλαξης επιπλέον πληροφοριών, οι οποίες δεν χρειάζονται στη κανονική μορφή των αλγορίθμων αλλά είναι απαραίτητες για τη μετατροπή του αλγορίθμου σε αναστρέψιμο. Εξίσου σημαντική είναι η μείωση των επιπλέον αυτών πληροφοριών στο ελάχιστο δυνατό.

Συγκεκριμένα, στη παρούσα Ατομική Διπλωματική Εργασία επικεντρωνόμαστε στην υλοποίηση των θεμελιωδών συναρτήσεων των Δυαδικών Δέντρων Αναζήτησης, των AVL δέντρων, των Splay δέντρων και των Σωρών. Όλοι οι αναστρέψιμοι αλγόριθμοι που προτείνονται, διατηρούν την ίδια χρονική πολυπλοκότητα με τις αντίστοιχες μη αναστρέψιμες υλοποιήσεις τους. Όσο αφορά τη χωρική τους πολυπλοκότητα σε μερικούς δεν υπάρχει καθόλου αύξηση, όπως για παράδειγμα στις αναζητήσεις, ενώ σε κάποιους παρατηρείται μία μικρή αύξηση.

Για την υλοποίηση των αλγορίθμων χρησιμοποιήθηκε η αναστρέψιμη γλώσσα προγραμματισμού Janus, η οποία άρχισε να αναπτύσσεται πριν μερικά χρόνια με αποτέλεσμα μέχρι στιγμής να μην υποστηρίζει τη χρήση δεικτών και αντικειμένων αλλά μόνο ακεραίων, πινάκων και στοιβών. Για να αντιμετωπίσουμε το περιορισμό αυτό προτείναμε τρόπο αναπαράστασης των δεντρικών δομών που μελετήσαμε με τη χρήση πινάκων, η οποία σε μεταγενέστερο στάδιο θα μπορεί εύκολα, απλά και γρήγορα να προσαρμοστεί στη χρήση δεικτών και αντικειμένων. Αυτό που κάναμε ήταν να χρησιμοποιήσουμε ένα δυσδιάστατο πίνακα όπου η κάθε γραμμή αντιστοιχεί σε κάποιο κόμβο και η κάθε στήλη σε κάποιο χαρακτηριστικό του, όπως για παράδειγμα τη τιμή του, το πατέρα, το δεξιό και το αριστερό του παιδί.

Περιεχόμενα

Κεφάλαιο 1

Εισαγωγή

1.1 Κίνητρο Διπλωματικής Εργασίας

Ο κόσμος των υπολογιστών είναι διαρκώς μεταβαλλόμενος, ιδιαίτερα σε ό,τι αφορά την υπολογιστική ισχύ και τις δυνατότητες των υπολογιστών. Η συνεχής αυτή ανάπτυξη έχει ταυτοχρόνως θετικές και αρνητικές συνέπειες - αφενός βοηθάει σε θέματα καινοτομίας, καθιστώντας δυνατό το να επιτευχθεί βελτίωση του βιοτικού επιπέδου και την επίλυση προηγουμένως άλυτων προβλημάτων. Από την άλλη πλευρά, προκαλεί μια τεράστια ζήτηση ηλεκτρικής ενέργειας. Σύμφωνα με το νόμο του Moore, είναι λογικό να υποθέσουμε ότι το πρόβλημα ισχύς θα συνεχίσει να αυξάνεται, εκτός και αν γίνει κάτι για να το αποτρέψει. Το 1961, ο Rolf Landauer απόδειξε ότι θα ήταν δυνατόν να δημιουργηθεί ένας υπολογιστής που απαιτεί ελάχιστη ή καμία απελευθέρωση θερμότητας. Αυτό οδήγησε στο πεδίο των αναστρέψιμων υπολογιστών [14].

Μέχρι αυτή τη στιγμή, υπήρξε μικρή εξερεύνηση στον κόσμο των αναστρέψιμων αλγορίθμων. Οι Axelsen και Yokoyama έχουν προτείνει και εφαρμόσει αναστρέψιμες εκδόσεις ταξινόμησης [4], θέτοντας έτσι ένα είδος αρχικών εκτιμήσεων ως προς το τι πρέπει να εξετάζεται κατά τη μετατροπή των αλγορίθμων από τις μη αναστρέψιμες τους μορφές σε αναστρέψιμες. Αξίζει να αναφέρουμε πως η μετατροπή ενός αλγορίθμου στη αναστρέψιμη μορφή του για αρκετές κατηγορίες αλγορίθμων, οι οποίοι είναι εκ φύσεως αναστρέψιμοι, δεν απαιτεί αλλαγές με αποτέλεσμα οι αλγόριθμοι αυτοί να παραμένουν άθικτοι. Αυτό δημιουργεί μια νέα πρόκληση, καθώς παρατηρείται εμφανή έλλειψη αναστρέψιμων δομών δεδομένων και έτσι αρκετοί αλγόριθμοι αν και εκ φύσεως αναστρέψιμοι δεν μπορούν να υλοποιηθούν.

1.2 Σκοπός Διπλωματικής Εργασίας

Η βασική ιδέα για την εκπόνηση της παρούσας διπλωματικής εργασίας συνιστάται από την παραπάνω ανάγκη για νέες αναστρέψιμες δομές δεδομένων καθώς με τις υπάρχουσες αναστρέψιμες δομές ένα τεράστιο φάσμα αλγορίθμων δεν μπορεί να μετατραπεί σε αναστρέψιμο. Για παράδειγμα η υλοποίηση αλγορίθμων, όπως ο Huffman, που χρησιμοποιούν δεντρικές δομές δεδομένων είναι αδύνατη καθώς δεν υπάρχουν ακόμη αναστρέψιμες δεντρικές δομές. Για το λόγο αυτό η παρούσα διπλωματική εργασία στηριζόμενη σε υπάρχουσες αναστρέψιμες δομές και τύπους δεδομένων προτείνει αναστρέψιμες δεντρικές δομές υλοποιώντας τις σχετικές τους συναρτήσεις εισαγωγής, διαγραφής και αναζήτησης.

Λαμβάνοντας τα πιο πάνω υπόψη, θέσαμε ως σκοπό της παρούσας Ατομικής Διπλωματικής Εργασίας να υλοποιηθούν οι θεμελιώδεις συναρτήσεις διάφορων δεντρικών δομών με στόχο τη δυνατότητα χρήσης τους για υλοποίηση αλγορίθμων που χρησιμοποιούν τις δομές αυτές.

Συγκεκριμένα τέθηκε ως στόχος η υλοποίηση των θεμελιωδών συναρτήσεων της κάθε δομής, όπως για παράδειγμα η εισαγωγή και η διαγραφή. Όπως είναι λογικό υλοποιήθηκαν και οι ειδικές συναρτήσεις που χρειάζεται η κάθε δομή για να διαφέρει και χρειάζεται για τις προαναφερθέντες συναρτήσεις όπως για παράδειγμα οι περιστροφές, η splay και η percolateDown. Για τις Σωρούς θεωρήθηκε χρήσιμο να προταθεί και ο αλγόριθμος ταξινόμησης heapsort, ο οποίος δεν περιλαμβάνεται στους αλγόριθμους ταξινόμησης που προτάθηκαν από τους Axelsen και Yokoyama [4].

Τέλος θεωρήθηκε εξίσου σημαντικό η υλοποίηση να γίνει όχι μόνο σε θεωρητικό επίπεδο αλλά και σε πρακτικό, δηλαδή σε κάποια γλώσσα προγραμματισμού. Επιλέξαμε τη γλώσσα προγραμματισμού Janus, καθώς αποτελεί τη πρώτη αναστρέψιμη γλώσσα προγραμματισμού [16], για την οποία μάλιστα έχει ήδη υλοποιηθεί ένας διερμηνέας πάνω στον οποίο μπορεί να τρέχει [25]. Ακόμη κατά την υλοποίηση στη γλώσσα προγραμματισμού Janus θα αντιμετωπίσουμε κάποιους περιορισμούς καθώς δεν έχουν προταθεί ακόμη αναστρέψιμοι δείκτες και αντικείμενα αλλά μόνο πίνακες ακεραίων και στοίβες. Για να αντιμετωπίσουμε το περιορισμό αυτό απεικονίζουμε τους κόμβους και τις ακμές με τη χρήση δισδιάστατων πινάκων όπου κάθε γραμμή αντιστοιχεί σε κάποιο κόμβο και η κάθε στήλη περιέχει πληροφορία για τον κόμβο αυτό.

1.3 Μεθοδολογία Διπλωματικής Εργασίας

Η μεθοδολογία που ακολουθήθηκε στα πλαίσια της παρούσας εργασίας είναι η ακόλουθη: Αρχικά, έγινε μια εκτενής μελέτη και έρευνα σε διάφορα ερευνητικά άρθρα τα οποία αφορούσαν το τομέα των Αναστρέψιμων Υπολογιστών και πιο συγκεκριμένα τον Αναστρέψιμο Προγραμματισμό για καλύτερη κατανόηση του νέου αυτού τομέα.

Έπειτα έγινε μία πρώτη επαφή με την αναστρέψιμη γλώσσα προγραμματισμού Janus μέσω της υλοποίησης μικρών απλών προγραμμάτων ώστε να εξοικειωθούμε με τη γλώσσα και να είμαστε σε θέση να υλοποιήσουμε τις συναρτήσεις των δεντρικών δομών που θα προτείνουμε.

Στη συνέχεια μελετήσαμε τις υπάρχουσες δεντρικές δομές και συναρτήσεις ώστε να έχουμε μία πλήρη εικόνα του προβλήματος που κληθήκαμε να επιλύσουμε. Με την ολοκλήρωση των πιο πάνω, επιλέξαμε για να υλοποιήσουμε τέσσερα είδη δεντρικών δομών:

1. Δυαδικά Δέντρα Αναζήτησης
2. AVL δέντρα
3. Splay δέντρα
4. Σωροί.

Για τις τέσσερις αυτές δομές προτείναμε αντίστοιχες αναστρέψιμες υλοποιήσεις, με τις θεμελιώδεις τους συναρτήσεις. Ακολούθως, υλοποιήσαμε τους αλγορίθμους αυτούς και αναλύσαμε τη χρονική και χωρική τους πολυπλοκότητα για να βεβαιωθούμε πως τηρούν τα κριτήρια αναστρεψιμότητας.

Ως τελευταίο στάδιο, τρέξαμε διάφορες περιπτώσεις έτσι ώστε να επαληθεύσουμε πως οι αναστρέψιμες συναρτήσεις που υλοποιήθηκαν στα πλαίσια της εργασίας λειτουργούν ορθά.

1.4 Δομή Διπλωματικής Εργασίας

Η δομή της εργασίας βάσει και των όσων περιγράψαμε παραπάνω έχει ως ακολούθως: Ξεκινάμε στο Κεφάλαιο 2 κάνοντας μία εισαγωγή στον Αναστρέψιμο Προγραμματισμό και τη γλώσσα προγραμματισμού Janus ώστε ο αναγνώστης να εξοικειωθεί με τις δύο αυτές ενότητες και να είναι σε θέση να κατανοήσει, να χρησιμοποιήσει αλλά και να επεκτείνει αν επιθυμεί τους προτεινόμενους αλγόριθμους για δική του χρήση.

Ακολούθως στα Κεφάλαιο 3, 4, 5, 6 δίνονται οι προτεινόμενες δεντρικές δομές με τις συναρτήσεις τους, δηλαδή τα Δυαδικά Δέντρα Αναζήτησης, AVL δέντρα, Splay δέντρα και τις σωρούς αντίστοιχα, καθώς επίσης και η θεωρητική τους ανάλυση. Θεωρήθηκε βοηθητικό πριν από κάθε αναστρέψιμη δεντρική δομή και συνάρτηση που προτείνουμε να υπάρχει μία επεξήγηση της κανονικής μορφής της και των αλγορίθμων ώστε να δίνεται στον αναγνώστη η πληρέστερη δυνατή εικόνα και κατανόηση των αλλαγών που χρειάζονται για τη μετατροπή ανάμεσα στις δύο μορφές.

Στο Κεφάλαιο 7 παρουσιάζονται τα συμπεράσματα και ενδεχόμενες μελλοντικές εργασίες που προκύπτουν από την παρούσα διπλωματική εργασία.

Η Διπλωματική Εργασία ολοκληρώνεται παρέχοντας σε παραρτήματα τις υλοποιήσεις των δεντρικών δομών δεδομένων που υλοποιήθηκαν.

Κεφάλαιο 2

Βιβλιογραφική Μελέτη και Τεχνολογικό Υπόβαθρο

2.1 Αναστρέψιμος Προγραμματισμός

Διαχρονικά η ανάπτυξη των ηλεκτρονικών υπολογιστών στηρίζεται στη σταθερή βελτίωση της ενεργειακής, κυρίως, απόδοσης των συστημάτων. Ενεργειακή απόδοση ονομάζουμε το αριθμό των χρήσιμων λειτουργιών επεξεργασίας πληροφοριών, συμπεριλαμβανομένων λογικής, αποθήκευσης, και λειτουργίες επικοινωνίας, οι οποίες μπορούν να εκτελεστούν ανά μονάδα διαθέσιμης ενέργειας που διαχέεται στο περιβάλλον ως θερμότητα.

Δυστυχώς αποτελεί κοινό μυστικό πως οι καλές ιδέες σχετικά με τη βελτίωση της ενεργειακής απόδοσης έχουν σχεδόν εξαντληθεί, καθώς τα υπάρχουσα τρανζίστορ έχουν μειώσει την τάση που χρειάζονται στο ελάχιστο δυνατό ώστε να αποφεύγονται προβλήματα όπως διαρροές ή ακόμη και σπάσιμο των συσκευών [11]. Αυτό έχει ως αποτέλεσμα οι νέες δομές τρανζίστορ να είναι σε θέση να αντιμετωπίσουν αυτά τα προβλήματα, αλλά μόνο σε περιορισμένο βαθμό.

Σύμφωνα με τον Michael P. Frank [12] "*Η δυσκολία βελτίωσης της ενεργειακής απόδοσης φαινόταν να αποτελεί «το τέλος του δρόμου» για την ιστορία της πληροφορικής τεχνολογίας, όσο αφορά την απόδοση στις περισσότερες εφαρμογές. Είναι όμως; Όχι λόγω του αναστρέψιμου υπολογισμού.*"

Η φράση αυτή αντικατοπτρίζει την κύρια αιτία ανάπτυξης του αναστρέψιμου προγραμματισμού. Δεν είναι τυχαίο άλλωστε το γεγονός πως ο αναστρέψιμος υπολογισμός έχει αρχίσει να αναπτύσσεται τα τελευταία χρόνια, όπου έγινε περισσότερο εμφανές το πιο πάνω πρόβλημα, παρόλο που ως έννοια είχε προταθεί πολύ παλαιότερα. Το κύριο κίνητρο για την ανάπτυξη αναστρέψιμων υπολογιστών έγκειται στο γεγονός ότι παρέχει το μόνο τρόπο (δηλαδή, είναι ο μόνος τρόπος που είναι λογικά συνεπής με τις πιο σταθερά εδραιωμένες αρχές

της θεμελιώδους φυσικής) μέσω του οποίου η απόδοση στις περισσότερες εφαρμογές με ρεαλιστικούς περιορισμούς ισχύς μπορεί να συνεχίζει επ' αόριστον αύξηση της.

Για πλήρη κατανόηση του τι αποτελεί αναστρέψιμο υπολογιστή και τι όχι αρκεί να θυμόμαστε τον «ορισμό» που έδωσαν οι Tetsuo Yokoyama και Robert Gluck [25] :

Ένα αναστρέψιμο σύστημα υπολογιστών [22, 6, 12] έχει, ανά πάσα στιγμή, το πολύ μία ενιαία προηγούμενη κατάσταση υπολογισμού καθώς και μία ενιαία επόμενη κατάσταση υπολογισμού, και έτσι ένα αναστρέψιμο σύστημα υπολογιστών μπορεί να τρέξει τα προγράμματα μοναδικά προς τα εμπρός και προς τα πίσω, ακολουθώντας την ντετερμινιστική τροχιά του υπολογισμού.

Πιο απλά, ένα αναστρέψιμο πρόγραμμα μας παρέχει τη δυνατότητα δοθείσας μίας τιμής εισόδου X να υπολογίσουμε την έξοδο Y, όμοια με τα κανονικά προγράμματα, αλλά και τη δυνατότητα με βάση την εξόδο Y να υπολογίσουμε την είσοδο X.

Οι περισσότεροι αναστρέψιμοι υπολογισμοί είναι ισοδύναμοι με τους αντίστοιχους μη αναστρέψιμους. Συγκεκριμένα έχει αποδειχθεί πως με απεριόριστους πόρους οι αναστρέψιμες μηχανές Turing είναι το ίδιο ισχυρές με τις κανονικές [5, 15, 14].

Πιο πάνω αναφέραμε πως οι αναστρέψιμοι υπολογιστές αποτελούν τη λύση για τη βελτίωση της ενεργειακής απόδοσης αλλά δεν εξηγήσαμε πως. Σύμφωνα με τον Landauer μία διαδικασία υπολογισμού όπου η προηγούμενη κατάσταση δεν είναι μοναδικά προσδιορίσιμη παράγει τουλάχιστον μία ελάχιστη ποσότητα θερμότητας [14]. Επίσης εικάζεται πως κάποιοι υπολογισμοί δεν μπορούν να γίνουν αναστρέψιμοι. Ωστόσο, ο Bennett διάψευσε την εικασία αυτή [5], επομένως τουλάχιστον θεωρητικά η θερμότητας που παράγεται/απελευθερώνεται κατά τους υπολογισμούς δεν έχει κάτω φράγμα. Λαμβάνοντας αυτά υπόψη έχει παρατηρηθεί μία θετική ένδειξη για περαιτέρω μείωση της κατανάλωσης ενέργειας των φυσικών διεργασιών υπολογισμού [23, 12, 6]. Ωστόσο, για να αυξήσουμε στο μεγαλύτερο δυνατό βαθμό τα κέρδη της ανακύκλωσης ενέργειας μέσω των αναστρέψιμων υπολογιστικών συστημάτων, δεν αρκεί να εξεταστούν μόνο ζητήματα υλικού χαμηλού επιπέδου, αλλά και υψηλού επιπέδου λογική αναστρεψιμότητα σε επίπεδο λογισμικού.

Παρόλο που οι περισσότεροι υπολογισμοί είναι αναστρέψιμοι, υπάρχουν περιπτώσεις που είναι αδύνατη η αντίστροφη διαγραφή δεδομένων. Αυτό έχει ως αποτέλεσμα όλες οι μεταβλητές είτε να φυλάγονται σε καθολικό επίπεδο είτε σε τοπικές μεταβλητές, των οποίων η αρχική και τελική τιμή μπορεί να υπολογιστεί μέσω άλλων μεταβλητών και σταθερών ώστε

να μπορούν να διαγραφούν χωρίς να χάνουμε πληροφορίες. Μία γνωστή τεχνική επαναφοράς τιμών στην προηγούμενη γνωστή του τιμή είναι η κλήση διαδικασιών με τη χρήση του Bennett Trick [7], όπου μπορούμε να χρησιμοποιήσουμε ή να αποθηκεύσουμε τις τιμές που χρειαζόμαστε και μετά να καλέσουμε με αντίστροφη εκτέλεση την ίδια διαδικασία με τις ίδιες παραμέτρους έτσι ώστε να ακυρώσουμε τις αλλαγές που έγιναν, παραδείγματος χάριν αν μία μεταβλητή *X=3* μετά την κλήση ισούται με κάποια τιμή Α τότε με την αντίστροφη κλήση θα επιστρέψει στην αρχική της μορφή *X=3* και επομένως η τιμή είναι γνωστή και μπορεί να διαγραφεί. Ωστόσο, δεν είμαστε πάντα σε θέση να αποκαταστήσουμε αυτές τις τοπικές μεταβλητές, πράγμα που σημαίνει ότι είμαστε αναγκασμένοι να αποθηκεύουμε επιπλέον πληροφορίες για να μπορούμε επανέλθουμε στην προηγούμενη κατάσταση.

Κύριο μέλημα του αναστρέψιμου υπολογισμού είναι ο περιορισμός αυτών των επιπλέον πληροφοριών, δηλαδή των πληροφοριών που δεν χρειάζονται στην αντίστοιχη μη αναστρέψιμη μορφή του εκάστοτε αλγορίθμου, στην ελάχιστη δυνατή ποσότητα έτσι ώστε να μειώσουμε τη χωρική του πολυπλοκότητα. Οι Axelsen και Yokoyama [4] πρότειναν για αξιολόγηση των πληροφοριών αυτών να γίνεται χρήση των όρων πιστότητα (faithfullness) και υγιεινή (hygiene). Μια πιστή αναστρέψιμη υλοποίηση προγράμματος έχει την ίδια χρονική πολυπλοκότητα με το μη αναστρέψιμο πρόγραμμα, χωρίς να απαιτείται η ελαχιστοποίηση των επιπρόσθετων πληροφοριών, ενώ ένα υγιεινό αναστρέψιμο πρόγραμμα είναι ένα πιστό αναστρέψιμο πρόγραμμα με την προστιθέμενη απαίτηση ότι το μέγεθος των επιπλέον πληροφοριών είναι βέλτιστο, δηλαδή το μικρότερο δυνατό. Γενικά προσπαθούμε για υγιεινές προτάσεις εφαρμογών κατά τη μετατροπή μιας μη αναστρέψιμης σε μια αναστρέψιμη έκδοση ώστε να έχουμε βέλτιστες υλοποιήσεις, ωστόσο, από τη προσωπική μας εμπειρία είναι ευκολότερο να δημιουργήσουμε πρώτα μία πιστή εφαρμογή, διατηρώντας την ίδια χρονική πολυπλοκότητα, και στη συνέχεια να εργαστούμε για τη μείωση της παραγωγής επιπλέον πληροφοριών.

Συνοπτικά:

- **Πιστότητα (faithfullness):** Ένας πιστός αναστρέψιμος αλγόριθμος διατηρεί τη χρονική πολυπλοκότητα της μη αναστρέψιμης μορφής του. Το όριο του μεγέθους των επιπρόσθετων πληροφοριών που απαιτούνται για τον αλγόριθμο μπορεί να περιγράφει από μία συνάρτηση που στηρίζεται στο μέγεθος των δεδομένων εισόδου.
- **Υγιεινή (hygiene):** Ένας υγιής αναστρέψιμος αλγόριθμος τηρεί όλες τις προϋποθέσεις πληρότητας, με την επιπλέον συνθήκη ότι η συνάρτηση που περιγράφει το μέγεθος των επιπλέον πληροφοριών είναι βέλτιστη, δηλαδή να ελαχιστοποιεί το μέγεθος των πληροφοριών αυτών.

2.2 Γλώσσα Προγραμματισμού Janus

Η Janus είναι ίσως η πρώτη αναστρέψιμη γλώσσα προγραμματισμού καθώς υλοποιήθηκε το 1982 από τους Christopher Lutz και Howard Derby στο Caltech [16]. Η λειτουργική σημασιολογία της γλώσσας καθορίστηκε επίσημα, μαζί με ένα πρόγραμμα-διερμηνέα όπου μπορούμε να τρέχουμε τη γλώσσα και ένα αναστρέψιμο διερμηνέα, το 2007 από τους Tetsuo Yokoyama και Robert Glück [25]. Το πρόγραμμα που υλοποίησαν είναι ελεύθερα διαθέσιμο σε ηλεκτρονική μορφή στην ιστοσελίδα της ερευνητική ομάδας TOPPS του Πανεπιστημίου DIKU [13]. Ο διερμηνέας αυτός συντηρείται κυρίως από τον Δρ. Michael Kirkedal Thomsen, ο οποίος παρέχει συχνές ενημερώσεις και αλγορίθμους. Σε αυτό το σημείο να αναφέρουμε πως η γλώσσα είναι βασισμένη στη γλώσσα προγραμματισμού Python. Υπάρχουν δύο μορφές υλοποιημένες: σε ηλεκτρονική μορφή και στο προσωπικό λογαριασμό του Δρ. Thomsen στο *Github* για εγκατάσταση σε προσωπικό υπολογιστή.

Αυτό που ξεχωρίζει μία αναστρέψιμη γλώσσα προγραμματισμού από μία κανονική είναι το ότι εκτός από προς τα εμπρός υπολογισμούς, υποστηρίζεται και προς τα πίσω ντετερμινιστικούς υπολογισμούς με τη χρήση τοπικών αναστροφών. Με πιο απλά λόγια κάθε υπολογισμός που εκτελείται πρέπει να υποστηρίζει και αντίθετη μορφή, η οποία μπορεί να ακυρώσει τον αρχικό υπολογισμό.

Εξαιτίας αυτού, είναι δυνατόν ορισμένες δράσεις που υλοποιούνται στον μη Αναστρέψιμο Προγραμματισμό να είναι αδύνατες στον Αναστρέψιμο Προγραμματισμό. Παραδείγματος χάρη, μετά την αρχική δημιουργία τοπικών ή μη τοπικών μεταβλητών δεν μπορούμε να αλλάξουμε τις αρχικές τιμές τους με άμεση ανάθεση νέων τιμών, χρήση συμβόλου =. Αντ' αυτού, μπορούν να γίνουν αναστρέψιμες ενέργειες όπως προσθήκη, αφαίρεση, *xor* ή εναλλαγή δύο στοιχείων.

Ο διερμηνέας της γλώσσας *Janus* ονομάζετε *Jana* και χρησιμοποιεί λέξεις-κλειδιά για να προσδιορίζει τη δομή του προγράμματος, δεν στηρίζεται ούτε σε παρενθέσεις για να παρακολουθεί το πεδίο εφαρμογή αλλά ούτε και σε ερωτηματικά για τον τερματισμό καταστάσεων, παρόλο που είναι κοινές τεχνικές σε πολλές από τις γλώσσες προγραμματισμού που χρησιμοποιούνται σήμερα.

⟨*program*⟩ := (procedure ⟨*identifier*⟩ (⟨*declaration*⟩*) ⟨*declaration*⟩* ⟨*statement*⟩*)+

⟨*declaration*⟩ := int ⟨*variable*⟩
| int ⟨*variable*⟩ = ⟨*expression*⟩
| int ⟨*variable*⟩[⟨*expression*⟩]
| int ⟨*variable*⟩[⟨*expression*⟩] = { ⟨*expression*⟩ (, ⟨*expression*⟩)* }
| stack ⟨*variable*⟩[⟨*expression*⟩]
| stack ⟨*variable*⟩[⟨*expression*⟩] = { ⟨*expression*⟩ (, ⟨*expression*⟩)* }
| stack ⟨*variable*⟩[⟨*expression*⟩][⟨*expression*⟩] = { ⟨*expression*⟩ (, ⟨*expression*⟩)* }

⟨*statement*⟩ := ⟨*variable*⟩ ⟨*function*⟩= ⟨*expression*⟩
| ⟨*variable*⟩[⟨*expression*⟩] ⟨*function*⟩= ⟨*expression*⟩
| ⟨*variable*⟩[⟨*expression*⟩][⟨*expression*⟩] ⟨*function*⟩= ⟨*expression*⟩
| if ⟨*expression*⟩ then ⟨*statement*⟩* (else ⟨*statement*⟩*)? fi ⟨*expression*⟩
| from ⟨*expression*⟩ do ⟨*statement*⟩ until ⟨*expression*⟩
| push (⟨*identifier*⟩, ⟨*identifier*⟩)
| pop (⟨*identifier*⟩, ⟨*identifier*⟩)
| local int ⟨*variable*⟩ = ⟨*expression*⟩ ⟨*statement*⟩* delocal int ⟨*variable*⟩ = ⟨*expression*⟩
| call ⟨*identifier*⟩ ((⟨*identifier*⟩(, ⟨*identifier*⟩)*)?)
| uncall ⟨*identifier*⟩ ((⟨*identifier*⟩(, ⟨*identifier*⟩)*)?)
| ⟨*identifier*⟩ <=> ⟨*identifier*⟩
| error (⟨*stringliteral*⟩)
| print (⟨*stringliteral*⟩)
| printf (⟨*stringliteral*⟩ , ⟨*identifier*⟩*)
| show (⟨*identifier*⟩*)
| skip
| statement statement

⟨*expression*⟩ := ⟨*constant*⟩ | ⟨*variable*⟩ | ⟨*variable*⟩[⟨*expression*⟩]
| | ⟨*variable*⟩[⟨*expression*⟩][⟨*expression*⟩]
| ⟨*expression*⟩ ⟨*operator*⟩ ⟨*expression*⟩
| empty (⟨*identifier*⟩) | top (⟨*identifier*⟩) | size (⟨*identifier*⟩)
| nil

⟨*function*⟩ := + | - | ^

⟨*operator*⟩ := ⟨*function*⟩
| * | / | % | */ | & | | | && | || | ⟨ / ⟩ | = | != | ⟨= / ⟩=

Σχήμα 1: Σύνταξη γλώσσας προγραμματισμού Janus

Ένα πρόγραμμα *Janus* μπορεί να αποτελείται από διάφορες διαδικασίες, αλλά πρέπει να περιλαμβάνει μία κύρια διαδικασία, η οποία υλοποιεί το πρόγραμμα. Κατά την προετοιμασία ενός προγράμματος Janus, απαιτείται να δοθούν όχι μόνο οι μεταβλητές εισόδου, αλλά και η

δημιουργία μεταβλητών για κάθε επιπλέον εξόδου που αναμένεται, δεδομένου ότι η γλώσσα επιτρέπει μόνο τη δημιουργία μη τοπικών μεταβλητών κατά την έναρξη της κύριας διαδικασίας.

Μέχρι στιγμής η γλώσσα υποστηρίζει μόνο ακέραιους, μονοδιάστατους πίνακες ακεραίων, δισδιάστατους πίνακες ακεραίων και στοίβες. Κατά την εκτύπωση και τον εντοπισμό λαθών, επιτρέπει και γραμματοσειρές. Επίσης δεν παρέχεται η δυνατότητα δημιουργίας δικών μας δομών, τύπων και αντικειμένων. Αυτό αποτελεί μία σημαντική δυσκολία και πρόκληση στη δημιουργία νέων δομών, όμως η απεικόνιση απλών δομών με τη χρήση δισδιάστατων πινάκων είναι εφικτή. Η υλοποίηση δομών με τη τεχνική αυτή όμως, εκτός του ότι μπορεί να δυσκολέψει την υλοποίηση αλγορίθμων, μπορεί να αυξήσει ή να μειώσει την χρονική πολυπλοκότητα και τις επιπρόσθετες πληροφορίες που φυλάγουμε κατά την αναστρεψιμότητα του εκάστοτε αλγορίθμου. Οι επιπλέον πληροφορίες αυτές αναφέρονται συχνά και ως σκουπίδια [4].

Οι εντολές *local, delocal* χρησιμοποιούνται για τη δημιουργία και διαγραφή τοπικών μεταβλητών, οι οποίες δεν αποτελούν μέρος των τελικών εξόδων μιας διαδικασίας. Κατά την αντίστροφη εκτέλεση ο ρόλος τους αναστρέφεται. Για το λόγο αυτό, κατά την εκτέλεση των εντολών πρέπει να είναι γνωστή η τιμή των μεταβλητών που δημιουργούνται ή διαγράφονται είτε μέσω μίας άλλης μεταβλητής είτε μέσω μίας σταθεράς. Οι τιμές των μεταβλητών κατά τις εντολές ελέγχονται ώστε να ισούνται και κατά την αντίστροφη τους εκτέλεση. Με βάση τα πιο πάνω γίνεται εύκολα κατανοητό πως ο εντοπισμός της κατάλληλης μεταβλητής κατά το *delocal* μπορεί να γίνει αρκετά δύσκολος.

Σε βρόχους, μπορεί να χρειαστούν επιπλέον κύκλοι, δεδομένου ότι δεν είναι δυνατόν απλώς να προσθέσουμε ή να αφαιρέσουμε το ποσό που λείπει από την τιμή του μετρητή σε ορισμένες περιπτώσεις - εκτός αν υπήρχε μια τιμή στο εσωτερικό του σώματος βρόχου, που θα μπορούσε να χρησιμοποιηθεί, καθώς διαφορετικά θα απαιτείτο η χρήση της τιμής του μετρητή κατά τον υπολογισμό αφαίρεσης ή προσθήκης, κάτι το οποίο θα ήταν μη αναστρέψιμη, καθώς όπως προαναφέραμε δεν υποστηρίζονται οι άμεσοι υπολογισμοί και ανάθεσης τιμών.

```
local int i = 0, found = 0, x = 3
for i = 0 do
        if array[i] = x then
                found += 1
        fi array[i] = x
        i += 1
until found = 1
delocal int i = ?, found = 1, x = 3
```

Σχήμα 2: Εντοπισμός θέσης με τιμή 3

Στο πιο πάνω σχήμα παρατηρούμε πως κατά τον εντοπισμό της θέσης που αναζητούμε δεν είμαστε σε θέση να γνωρίζουμε την τιμή της μεταβλητής *i* και έτσι είναι αδύνατη η διαγραφή της. Ένας τρόπος αντιμετώπισης του προβλήματος, θα ήταν να μην σταματάμε το βρόγχο μόλις βρούμε τη θέση αλλά να τον αφήνουμε να διαπεράσει ολόκληρο το πίνακα μας έτσι ώστε με το τέλος του βρόγχου η τιμή της μεταβλητής *i* να ισούται πάντοτε με το μέγεθος του πίνακα και έτσι να είμαστε σε θέση να εκτελέσουμε την εντολή *delocal*.

Αναφέραμε νωρίτερα πως η αναστρεψιμότητα απαγορεύει την απευθείας ανάθεση τιμών, δηλαδή τη χρήση του συμβόλου =. Για ανάθεση χρησιμοποιούνται οι χαρακτήρες += και -= έτσι ώστε να παρέχεται η δυνατότητα και στην αναστρεψιμότητα να μπορεί να εξάγει τις σωστές τιμές των μεταβλητών. Επίσης υποστηρίζεται και η εντολή ^= που αντιστοιχεί στη χρήση *xor*. Κατά την αναστρεψιμότητα γίνεται μετατροπή της πρόσθεσης σε αφαίρεση και αντίθετα. Κοντά σε αυτά, το ίδιο συμβαίνει και κατά την εκτέλεση υπολογισμών με τη χρήση πολλαπλασιασμού και διαίρεσης. Επιπρόσθετα, απαγορεύονται οι πολλαπλασιασμοί με το μηδέν καθώς δεν υπάρχει τρόπος να γνωρίζουμε την αρχική τιμή μετά την εκτέλεση της πράξης.

Η γλώσσα υποστηρίζει δηλώσεις ανάθεσης, ανταλλαγής*, if-then-else, loop, call/ uncall, skip* και ακολουθίες δηλώσεων.

Η δήλωση ανταλλαγής κάνει απλή ανταλλαγή στις τιμές των δύο μεταβλητών για τις οποίες εκτελείται. Οι δύο αυτές τιμές πρέπει να είναι του ιδίου τύπου, ακέραιος ή ακέραιος σε πίνακα.

Ακόμη δεν υποστηρίζεται η κλήση συναρτήσεων όπου θέτουμε ως παράμετρο τιμή θέσης πίνακα, για παράδειγμα foo(array[1]), όταν η μεταβλητή αυτή αλλάζει τιμή κατά την εκτέλεση της διαδικασίας. Ένας τρόπος αντιμετώπισης είναι η δημιουργία τοπικής μεταβλητής με τη τιμή του πίνακα, πριν την εκτέλεση, ώστε να εκτελείται η συνάρτηση με τη τιμή αυτή και έπειτα να ενημερώνουμε τη θέσης του πίνακα με την νέα τιμή της μεταβλητής.

Το πιο δύσκολο μέρος της γλώσσας αλλά ταυτόχρονα και το πιο ενδιαφέρον είναι οι δηλώσεις *if-then-else*. Για να υποστηρίζεται αναστρεψιμότητα προστέθηκε μία ακόμη συνθήκη στο τέλος της δήλωσης, το *fi* (συνθήκη αναστρεψιμότητας), το οποίο έχει την ίδια λογική με το κανονικό *if* με τη διαφορά πως πρέπει να ισχύει μόνο μετά από εκτέλεση της δήλωσης όπου η συνθήκη *if* ήταν αληθές. Με το τρόπο αυτό παρέχουμε τη δυνατότητα κατά την αντίστροφη εκτέλεση να παρέχεται τρόπος αλλά και η απαραίτητη γνώση ώστε να εκτελεστεί το αντίστοιχο μέρος, αληθές ή όχι, της δήλωσης. Αξίζει να σημειώσουμε πως σε πολλές περιπτώσεις οι *if* και

fi συνθήκες δεν θα είναι οι ίδιες, καθώς υπάρχει η δυνατότητα οι μεταβλητές που ανήκουν στη μία συνθήκη να αλλάξουν κατά την εκτέλεση της δήλωσης. Δυστυχώς όμως μερικές φορές είναι αδύνατη η αναστροφή των συνθηκών χωρίς τη προσθήκη κάποιας επιπλέον πληροφορίας.

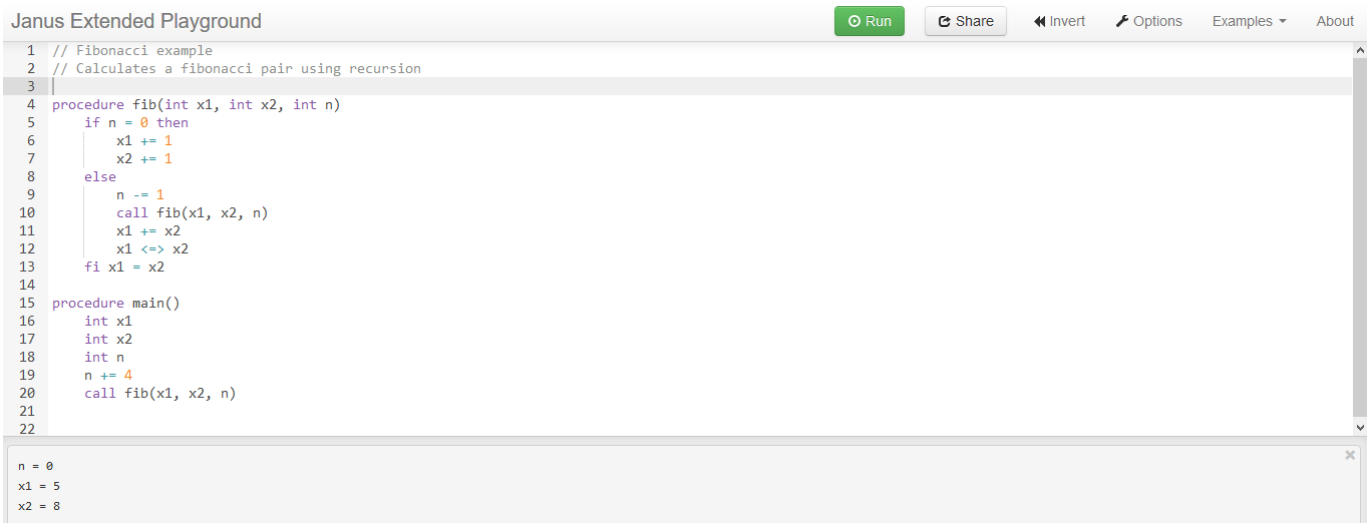
Για τις δηλώσεις *loop*, οι οποίες αποτελούν τους βρόγχους μας, πρέπει η συνθήκη στο *from* να ισχύει μόνο κατά την 1η εκτέλεση του επαναληπτικού βρόγχου. Αντίστοιχα η συνθήκη *until* είναι η συνθήκη τερματισμού του βρόγχου και πρέπει να ισχύει μόνο μετά την έξοδο μας από τον εκάστοτε βρόγχο. Επιπλέον γίνεται χρήση του *loop*, το οποίο επιτρέπει την εκτέλεση του βρόγχου μόνο εάν η συνθήκη τερματισμού δεν ισχύει και με την ολοκλήρωση του βρόγχου η συνθήκη πρέπει να είναι ψευδής.

Η συνθήκη *fi* στη συνέχεια της Ατομικής Διπλωματικής Εργασίας θα αναφαίρετε ως **Συνθήκη Αναστρεψιμότητας** και η συνθήκη *from* θα ονομάζεται **Συνθήκη Εισόδου στο βρόγχο**.

Οι δηλώσεις *call* και *uncall* χρησιμοποιούνται για να εκτελέσουμε μία διαδικασία κανονικά - προς τα εμπρός, ή ανάποδα - προς τα πίσω, αντίστοιχα. Μπορεί να μην είναι σαφές η χρησιμότητα του *uncall*, όμως είναι εξαιρετικά χρήσιμο για την εξάλειψη των επιπλέον πληροφοριών. Όπως εξηγήσαμε πιο πάνω είναι απαραίτητο να γνωρίζουμε την τιμή μιας μεταβλητής κατά τη χρήση του *delocal*. Το *uncall* μπορεί να πάρει την έξοδο από μια κλήση διαδικασίας και να επιστρέψει την είσοδο καθώς εκτελεί ανάποδα τη διαδικασία. Επομένως μπορεί να χρησιμοποιηθεί για την εξουδετέρωση τυχόν τοπικών μεταβλητών που χρησιμοποιήθηκαν στη διαδικασία. Η μέθοδος αυτή είναι γνωστή ως Bennett Trick [7].

Στην τελευταία έκδοσης της γλώσσας που δημοσιεύθηκε τον Απρίλιο του 2017 προστέθηκε η δυνατότητα εκθετικών υπολογισμών με τη χρήση του δυαδικού τελεστή **, όπως για παράδειγμα *2 ** 2 = 4, 3 ** 3 = 9*.

Συστήνω για καλύτερη κατανόηση και εξοικείωση με τη γλώσσα, αρχικά να χρησιμοποιηθεί η ηλεκτρονική μορφή του διερμηνέα [13] καθώς παρέχει μία πληθώρα υλοποιημένων προγραμμάτων, τα οποία μπορούν να χρησιμοποιηθούν ως παραδείγματα.



Σχήμα 3: Ηλεκτρονική μορφή Janus

Όπως βλέπουμε στο Σχήμα 3 η ιστοσελίδα παρέχει της εξής τρεις λειτουργίες:

- **Run:** Εκτελείται ο κώδικας που γράψαμε και εμφανίζετε το αποτέλεσμα στο κάτω μέρος της σελίδας.
- **Invert:** Αναστρέφει το υπάρχων πρόγραμμα, δηλαδή μετατρέπει και εμφανίζει τη αντίστροφη μορφή του κώδικα. Αυτό είναι χρήσιμο για καλύτερη κατανόηση της αναστρεψιμότητας καθώς επίσης και για να εντοπίζονται οι κατάλληλες συνθήκες για τις δηλώσεις *fi* και *until*.
- **Examples:** Παρέχει μία λίστα με προγράμματα βοηθώντας στη καλύτερη κατανόηση και εξοικείωση με τη σύνταξη και το τρόπο λειτουργίας της γλώσσας.

Σε περίπτωση που κάποιος επιθυμεί να εγκαταστήσει τη Janus στο προσωπικό του υπολογιστή, είναι απαραίτητο να ζητήσει το κώδικα από τον Δρ. Michael Kirkedal Thomsen ώστε να του δώσει πρόσβαση στον προσωπικό του λογαριασμό στο *Github*, ο οποίος βρίσκεται στην εξής ιστοσελίδα *https://github.com/kirkedal*. Στον χώρο αυτό θα βρείτε και οδηγίες εγκατάστασης.

Για να τρέξουμε ένα πρόγραμμα Janus, χρησιμοποιούμε την εντολή *jana filepath*, όπου *filepath* είναι η θέση του αρχείου στον υπολογιστή. Αυτό θα τρέξει το αρχείο του διερμηνέα *Jana* και θα επιστρέψει τα αποτελέσματα μιας σουίτας δοκιμών, τις οποίες εμείς καθορίσαμε στην κύρια διαδικασία του προγράμματος που υλοποιήσαμε, καθώς και τις τελικές τιμές εξόδου, αν βέβαια οι δοκιμές μας εκτελούνται ορθά. Σε περίπτωση λάθους παίρνουμε αντίστοιχα μηνύματα όπως για παράδειγμα «εκτός ορίων» και «η μεταβλητή δεν υπάρχει».

Όσο αφορά την αναστρεψιμότητα μπορεί να πάρουμε το εξής μήνυμα:
Assertion failed: should be true: Μία συνθήκη μας έπρεπε να είναι αληθής αλλά είναι ψευδής
Assertion failed: should be false: Μία συνθήκη μας έπρεπε να είναι ψευδής αλλά είναι αληθής

Οι συνθήκες οι οποίες μπορούν να προκαλέσουν τα πιο πάνω είναι οι συνθήκες αναστρεψιμότητας και εισαγωγής στο βρόγχο.

Ένα παράδειγμα της λειτουργίας ενός προγράμματος δοκιμών στη κονσόλα προσωπικού υπολογιστή φαίνεται στο Σχήμα 4, όπου τρέχουμε τον αλγόριθμο *fibonnaci* όπως και στο Σχήμα 3.

```
C:\Users\Athos\Desktop\diplomatiki\Jana-JanusInterp-master\examples>jana fib.ja
n = 0
x1 = 5
x2 = 8
```

Σχήμα 4: Εκτέλεση Jana αρχείου από προσωπικό υπολογιστή μέσω τερματικού (terminal)

Τέλος, δεν έχει αναπτυχθεί ακόμη κάποιο IDE που να υποστηρίζει τη γλώσσα επομένως η συγγραφή προγραμμάτων γίνεται σε επεξεργαστές κειμένου όπως το *notepad*, το *Notepad++* ή το *gedit* και τα αρχεία αποθηκεύονται σε *.ja* μορφή.

Λαμβάνοντας αυτά υπόψη μπορεί εύκολα κάποιος να υποθέσει πως λόγω όλων αυτών των αλλαγών η εκμάθηση της γλώσσας είναι δύσκολη. Μέσα από την προσωπική μας εμπειρία, μπορούμε να πούμε πως η υπόθεση αυτή είναι λανθασμένη καθώς η εκμάθηση της γλώσσας δεν είναι δύσκολή διότι ακολουθεί πιστά την έννοια την αναστρεψιμότητας. Αν κάποιος κατανοήσει την έννοια της αναστρεψιμότητας τότε θα βρει την γλώσσα αρκετά απλή και βοηθητική. Η δυσκολία που παρατηρείται στην χρήση της γλώσσας οφείλεται στο ότι είναι ακόμη σε πρώιμο στάδιο με αποτέλεσμα να υπάρχουν πολλοί περιορισμοί όπως για παράδειγμα λίγοι τύποι δεδομένων και λίγες δομές.

Κεφάλαιο 3

Δυαδικό Δέντρο Αναζήτησης

3.1 Γνωριμία με τη δομή

Τα Δυαδικά Δέντρα Αναζήτησης είναι μια δομή που βοηθάει στην φύλαξη δεδομένων παρέχοντας γρήγορη και εύκολη αναζήτηση, προσθήκη και διαγραφή στοιχείων και μπορούν να υλοποιήσουν είτε δυναμικά σύνολα αντικειμένων είτε πίνακες αναζήτησης στοιχείων με βάση κάποιο κλειδί. Τα δυαδικά δέντρα φυλάσσουν τα στοιχεία τους σε ταξινομημένη σειρά έτσι ώστε οι λειτουργίες τους να μπορούν να κάνουν χρήση της αρχής της δυαδικής αναζήτησης:

« Όταν ψάχνουμε για ένα κλειδί σε ένα δέντρο ή ένα μέρος για να εισάγουμε ένα νέο κλειδί, διασχίζουμε το δέντρο από τη ρίζα μέχρι τα φύλλα, κάνοντας συγκρίσεις με τα κλειδιά των κόμβων του δέντρου και αποφασίζουμε, με βάση συγκρίσεων, αν θα συνεχίσουμε την αναζήτηση στο αριστερό ή δεξιό υποδένδρο. »

Κατά μέσο όρο, αυτό σημαίνει ότι χάρη στις συγκρίσεις παρέχεται η δυνατότητα οι εργασίες να αγνοήσουν περίπου το ήμισυ του δέντρου, έτσι ώστε κάθε αναζήτηση, εισαγωγή ή διαγραφή να απαιτεί χρόνο ανάλογο προς το λογάριθμο του αριθμού των στοιχείων που αποθηκεύονται στο δέντρο. Αυτό είναι πολύ καλύτερο από το γραμμικό χρόνο που απαιτείται για εντοπισμό στοιχείων σε ένα μη ταξινομημένο πίνακα.

Στα δυαδικά δέντρα αναζήτησης ονομάζουμε τον αρχικό κόμβο ρίζα του δέντρου, κάθε εσωτερικός κόμβος συνδέεται με το πατέρα του και το πολύ με δύο υποδένδρα - το αριστερό και το δεξιό παιδί του, ενώ τα φύλλα, κόμβοι τερματισμού, συνδέονται μόνο με το πατέρα τους. Το δέντρο ικανοποιεί επιπλέον τη δυαδική ιδιότητα αναζήτησης η οποία αναφέρει ότι το κλειδί σε κάθε κόμβο πρέπει να είναι μεγαλύτερο ή ίσο με κάθε κλειδί που είναι αποθηκευμένο στο αριστερό υποδένδρο, και μικρότερο ή ίσο με κάθε κλειδί που είναι αποθηκευμένο στο δεξιό υποδένδρο [9].

Δύο μειονεκτήματα της δομής είναι:

- Το σχήμα του δυαδικού δένδρου αναζήτησης εξαρτάται εξ ολοκλήρου από τη σειρά των εισαγωγών και διαγραφών, με αποτέλεσμα το δέντρο να μπορεί να εκφυλιστεί, δηλαδή να γίνει σαν συνδεδεμένη λίστα με γραμμική χρονική πολυπλοκότητα.
- Κατά την τοποθέτηση, τη διαγραφή ή την αναζήτηση ενός στοιχείου σε ένα δυαδικό δένδρο αναζήτησης, το κλειδί κάθε κόμβου που επισκεφθήκαμε πρέπει να συγκριθεί με το κλειδί του στοιχείου που πρόκειται να εισαχθεί, να διαγραφεί ή να εντοπιστεί.

Λόγω της πιο πάνω ιδιότητας αλλά και για την αντιμετώπιση των μειονεκτημάτων της δομής έχουν μελετηθεί και προταθεί αρκετές παραλλαγές των δυαδικών δένδρων αναζήτησης όπως για παράδειγμα τα AVL και Splay Δέντρα που θα περιγράψουμε αργότερα.

3.2 Αναστρέψιμη δομή και περιορισμοί στη Janus

Πριν προτείνουμε μία αναστρέψιμη δομή πρέπει να υπενθυμίσουμε τους περιορισμούς που είχαμε εξαιτίας της γλώσσα προγραμματισμού Janus. Μέχρι στιγμής δεν υποστηρίζονται δείκτες, οι οποίοι είναι απαραίτητοι όχι τόσο για την υλοποίηση των δέντρων αλλά κυρίως για την απλούστευση και βελτιστοποίηση τους. Λόγω του ότι δεν υποστηρίζονται δείκτες οι υπάρχουσες δομές που έχουμε στην διάθεση μας είναι περιορισμένες. Συγκεκριμένα μπορούμε να χρησιμοποιήσουμε μόνο πίνακες, μονοδιάστατους ή δυσδιάστατους, και στοίβες. Για το λόγο αυτό αποφασίσαμε την υλοποίηση του δυαδικού δέντρου αναζήτησης με τη χρήση πινάκων καθώς η υλοποίηση με τη χρήση στοιβών θα ήταν δυσκολότερη και πιθανόν σπάταλη.

Έπειτα κληθήκαμε να αντιμετωπίσουμε μία άλλη δυσκολία της γλώσσας. Ο κενός χαρακτήρας υποστηρίζεται, μέχρι στιγμής, μόνο στις στοίβες, και έτσι έπρεπε να επιλέξουμε μία τιμή, η οποία θα αντιστοιχούσε στο κενό. Επιλέξαμε το μηδέν. Ακόμη λόγω του ότι το μηδέν αποτελεί θέση του πίνακα όλες οι διαδικασίες μας ξεκινούν από τη θέση ένα. Η δομή μπορούσε να υλοποιηθεί θέτοντας ως κενό το -1 και να αρχίζουμε από τη θέση μηδέν του πίνακα. Η επιλογή

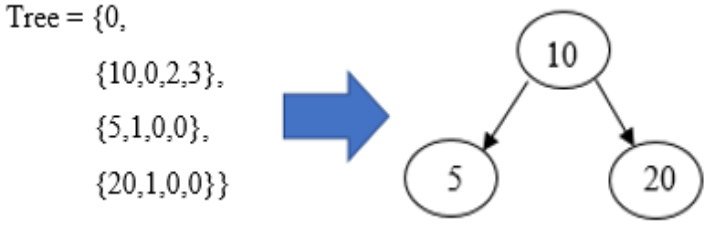
να αναπαριστούμε το κενό με το μηδέν και το δέντρο μας να αρχίζει από τη θέση ένα του πίνακα έγινε για ομοιομορφία με τις άλλες δομές, για τις οποίες όπως θα δούμε αργότερα η χρήση του μηδέν ως κενό χαρακτήρα ήταν αρκετά βοηθητική.

Σε αυτό το σημείο έπρεπε να αποφασίσουμε πως θα δείχνουμε τις συσχετίσεις ανάμεσα στους κόμβους. Μία πρώτη σκέψη ήταν η κάθε θέση του πίνακα να αντιστοιχεί σε ένα κόμβο, όπου θα φυλάσσεται η τιμή του, ο πατέρας του να βρίσκεται στη θέση *i/2*, το αριστερό του παιδί στη θέση *2i* και το δεξί στη *2i+1*. Όμοια δηλαδή λογική που χρησιμοποιούν οι σωροί. (βλέπε κεφάλαιο 6). Αυτό όμως θα μας έδινε μία συγκεκριμένη λύση η οποία θα εξαρτιόταν από τη χρήση πινάκων. Στόχος μας ήταν παρόλες τις δυσκολίες και τους περιορισμούς της γλώσσας, να προτείνουμε δομές και αλγορίθμους οι οποίοι θα μπορούν να υλοποιηθούν και με τη χρήση δεικτών, όταν αυτοί υλοποιηθούν, κάνοντας τις μικρότερες δυνατές τροποποιήσεις ώστε να παρέχουμε μία γενικευμένη υλοποίηση.

Για το λόγο αυτό η αρχική μας ιδέα απορρίφθηκε και έγινε χρήση δυσδιάστατου πίνακα διαστάσεων n×4, που θα μας έδινε μία εντύπωση χρήσης κόμβων με τα εξής χαρακτηριστικά:

1. Κάθε γραμμή του πίνακα αποτελεί ένα κόμβο
2. Κάθε στήλη αντιστοιχεί σε μία πληροφορία για τον εκάστοτε κόμβο:
    a. Στήλη 0 = τιμή που φυλάσσεται στο κόμβο
    b. Στήλη 1 = θέση του πατέρα
    c. Στήλη 2 = θέση αριστερού παιδιού
    d. Στήλη 3 = θέση δεξιού παιδιού
3. Η πρώτη γραμμή του πίνακα παραμένει άδεια, δηλαδή οι στήλες τις ισούνται με μηδέν.

Για την προσθήκη επιπλέον πληροφοριών για κάθε κόμβο αρκεί να αυξήσουμε τις στήλες του πίνακα. Στο σχήμα 5 μπορείτε να δείτε ένα παράδειγμα δέντρου τριών κόμβων ώστε να γίνει πιο κατανοητή η προτεινόμενη μορφή δυαδικών δέντρων αναζήτησης.



Σχήμα 5: Παράδειγμα δέντρου στη μορφή που προτείνουμε

Με αυτό το τρόπο η κάθε γραμμή μπορεί να θεωρηθεί ως αντικείμενο/κόμβος. Ακόμη μπορούμε να θεωρήσουμε τις στήλες 1, 2, 3 ως δείκτες καθώς συνδέουν το κάθε κόμβο με το κόμβο που τηρεί την προϋπόθεση συσχέτισης για την κάθε στήλη, πατέρας, αριστερό ή δεξιό

παιδί. Αυτή η σύνδεση είναι άμεση και όχι έμμεση καθώς δεν εξαρτάται από συγκεκριμένες θέσεις όπως για παράδειγμα στην αρχική μας σκέψη όπου ο κόμβος *i* θα συνδεόταν αναγκαστικά με τις θέσεις *i /2, 2i+1* και *2i +1*.

Από αυτό το σημείο και έπειτα θα θεωρούμε στις γραμμές του πίνακα ως κόμβους και για τις συνδέσεις ανάμεσα στους κόμβους θα χρησιμοποιούμε την ονομασία δείκτες, παράδειγμα για τη στήλη 1 θα λέμε ο δείκτης του πατέρα του κόμβου ή απλά ο πατέρας του.

Συνοψίζοντας τα πιο πάνω, υποθέτουμε πως αν υπήρχαν αναστρέψιμοι δείκτες και αντικείμενα η δομή των δυαδικών δέντρων αναζήτησης δεν θα χρειαζόταν κάποια αλλαγή ώστε να γίνει αναστρέψιμη. Έτσι θα προτείνουμε υλοποιήσεις των συναρτήσεων της δομής που θα έχουν μία αφαιρετική περιγραφή ώστε να δείχνουμε την αναστρεψιμότητα τους ανεξαρτήτως γλώσσας προγραμματισμού και τυχών περιορισμών της. Η υλοποίηση μας στη γλώσσα προγραμματισμού Janus βρίσκεται στο Παράρτημα Α και χρησιμοποιήθηκε κυρίως ως μέσω ελέγχου και επαλήθευσης των προτάσεων μας.

Στα σημεία όμως που αντιμετωπίσαμε δυσκολίες λόγω της χρήσης πίνακα ή άλλων περιορισμών της γλώσσας προγραμματισμού Janus κατά την υλοποίηση θα κάνουμε αναφορά στη δυσκολία αυτή και πως την επιλύσαμε.

3.3 Αναζήτηση

Η αναζήτηση ενός στοιχείου γίνεται πολύ εύκολα αρχίζοντας από τη ρίζα. Αν είναι κενή τότε το στοιχείο δεν υπάρχει στο δέντρο καθώς στην ουσία δεν υπάρχει καν δέντρο. Αν η τιμή ισούται με αυτή της ρίζας τότε επιστρέφουμε τη ρίζα. Αλλιώς συγκρίνουμε τη τιμή που ψάχνουμε με τη τιμή της ρίζας και συνεχίζουμε αναδρομικά στο αριστερό υποδένδρο αν η επιθυμητή τιμή είναι μικρότερη από αυτή του τρέχοντος κόμβου αλλιώς στο δεξιό.

Επειδή στη χειρότερη περίπτωση ο αλγόριθμος πρέπει να ψάξει από τη ρίζα του δέντρου στο μακρινότερο φύλλο από τη ρίζα, η διαδικασία αναζήτησης απαιτεί χρόνο ανάλογο με το ύψος του δέντρου. Κατά μέσο όρο, η αναζήτηση σε ένα δυαδικό δέντρο αναζήτησης με n κόμβους έχει ύψος $O(\log n)$ [9]. Ωστόσο, στη χείριστη περίπτωση, τα δυαδικά δέντρα αναζήτησης μπορεί να έχουν ύψος O(n), όταν το μη ισορροπημένο δέντρο μοιάζει με μια συνδεδεμένη λίστα (εκφυλισμένο δέντρο) το οποίο μπορεί να προκύψει αν όλοι οι κόμβοι του δέντρου μας έχουν μόνο αριστερό ή μόνο δεξιό παιδί.

Αναστρέψιμη μορφή

Η αναζήτηση στα δυαδικά δέντρα αναζήτησης είναι εκ φύσεως αναστρέψιμη και επομένως δεν χρειάστηκε κάποια αλλαγή. Ξεκινώντας από τη ρίζα ή κάποιο δοθέντα κόμβο μετακινούμαστε στο δέντρο συγκρίνοντας τη τιμή του τρέχοντος κόμβου με τα παιδιά του μέχρι να φθάσουμε στο κόμβο με τη τιμή που ψάχνουμε. Αν η τιμή που ψάχνουμε είναι μικρότερη από αυτή του τρέχοντος κόμβου συνεχίζουμε το ψάξιμο αναδρομικά από το αριστερό παιδί του κόμβου αλλιώς από το δεξιό. Ως συνθήκη αναστρεψιμότητας χρησιμοποιήθηκε η ίδια σύγκριση που γίνεται για το καθορισμό του σε ποιο υποδένδρο θα συνεχίσω την αναζήτηση. Αυτό είναι αρκετό καθώς κατά την αναζήτηση δεν αλλάζουμε τα στοιχεία του δέντρου και επομένως η συνθήκη αυτή ισχύει και για προς τα εμπρός και για προς τα πίσω αναζήτηση. Σε περίπτωση που δεν υπάρχει ο κόμβος εμφανίζεται μήνυμα λάθους αλλιώς επιστρέφεται η θέση του στοιχείου που αναζητάμε.

Ακολουθεί ο ψευδοκώδικας της διαδικασίας:

```
procedure find(int value, node currentNode, node result)
      if currentNode = null then
            print("Does not exists")
      else
            if currentNode.value != value then
                  if value < currentNode.value then
                        call find(value, currentNode.left, result)
                  else
                        call find(value, currentNode.right, result)
                  fi value < currentNode.value
            else
                  result += currentNode
            fi currentNode.value != value
      fi currentNode = null
```

Χρονική και Χωρική Πολυπλοκότητα

Λόγω των πιο πάνω γίνεται αντιληπτό πως η χρονική πολυπλοκότητα της αναζήτησης παρέμεινε η ίδια. Επίσης δεν χρειάζονται επιπλέον πληροφορίες με αποτέλεσμα η χωρική πολυπλοκότητα να παραμένει η ίδια. Επομένως η αναστρέψιμη μορφή του αλγορίθμου που προτείνουμε είναι υγιεινή.

3.4 Εισαγωγή

Η διαδικασία της εισαγωγής ξεκινά όπως την αναζήτηση. Αν το κλειδί δεν ισούται με τη ρίζα ψάχνουμε το αριστερό ή δεξιό υποδένδρο όμοια με πιο πάνω για να βρούμε με ποιο κόμβο θα ενώσουμε το νέο στοιχείο. Όταν εντοπίσουμε το κόμβο αυτό και πρέπει να προσθέσουμε το νέο στοιχείο, το συνδέουμε ως δεξιό ή αριστερό του παιδί. Πιο απλά, εξετάζουμε τη ρίζα και αναδρομικά εισάγουμε το νέο κόμβο στο αριστερό υποδένδρο αν το κλειδί του είναι μικρότερο από εκείνο της ρίζας ή στο δεξιά υποδένδρο αν το κλειδί του είναι μεγαλύτερο ή ίσο με τη

ρίζα. Όμοια με την αναζήτηση η χρονική του πολυπλοκότητα είναι κατά μέσο όρο $O(\log n)$ και στη χείριστη περίπτωση O(n).

Αναστρέψιμη μορφή

Για την υλοποίηση της εισαγωγής σε αναστρέψιμη μορφή στη Janus χρειαστήκαμε μία επιπλέον μεταβλητή που μας δείχνει τη θέση μετά το τελευταίο στοιχείο. Με άλλα λόγια είναι η γραμμή στην οποία θα εισαχθούν τα στοιχεία του νέου κόμβου, δηλαδή η θέση του νέου κόμβου στο πίνακα μας. Η ιδέα του αλγορίθμου είναι να γίνεται διάσχιση του δέντρου μέχρι το πατέρα του κόμβου που θα εισάγουμε, ώστε να γνωρίζουμε τη θέση του πατέρα και να μπορούμε να ενώσουμε το κόμβο στο δέντρο. Έχοντας και τη πιο πάνω βοηθητική μεταβλητή γνωρίζουμε το πατέρα και τη θέση του νέου κόμβου. Οπότε είμαστε σε θέση να ενώσουμε τους δύο αυτούς κόμβους. Έπειτα από κάθε εισαγωγή η βοηθητική μεταβλητή αυξάνεται κατά ένα ώστε να δείχνει στη θέση που πρέπει να εισαχθεί ο επομένως κόμβος. Αν είχαμε δείκτες η μεταβλητή θα απεικόνιζε ένα κενό κόμβο - αυτό που θα εισαχθεί, επομένως δεν χρειάζεται η φύλαξη της - δεν αποτελεί επιπλέον πληροφορία, κατ' ακρίβεια δεν χρησιμοποιείται.

Ο αλγόριθμος μας συνοπτικά ακολουθεί την εξής διαδικασία:

1. Φύλαξε τη τιμή του νέου κόμβου
2. Αν δεν εισάγεις τη ρίζα διάσχισε αναδρομικά το δέντρο
3. Αν η τιμή του νέου κόμβου είναι μικρότερη της τιμής του τρέχοντος κόμβου X
4. Έλεγξε αν το αριστερό παιδί είναι κενό. Αν ναι ένωσε τον X με το νέο κόμβο
5. αλλιώς συνέχισε αναδρομικά από το αριστερό παιδί.
6. **Συνθήκη Αναστρεψιμότητας:** το αριστερό παιδί του X ισούται με το νέο κόμβο.
7. Αλλιώς
8. Έλεγξε αν το δεξιό παιδί είναι κενό. Αν ναι ένωσε τον X με το νέο κόμβο
9. αλλιώς συνέχισε αναδρομικά από το δεξιό παιδί.
10. **Συνθήκη Αναστρεψιμότητας:** το δεξιό παιδί του X ισούται με το νέο κόμβο.
11. **Συνθήκη Αναστρεψιμότητας:** η τιμή του νέου κόμβου είναι μικρότερη της τιμής του X

Ακολουθεί ο ψευδοκώδικας της διαδικασίας:

```
procedure insert(int value, node root,  node Newnode)
1.          Newnode.value += value
2.          if Newnode != root then
                  call insertHelper(value, root, Newnode)
            fi Newnode != root

procedure insertHelper(int value, node currentNode, node Newnode)
3.    if value < currentNode.value then
4.          if currentNode.left = null then
                  currentNode.left += Newnode
                  Newnode.parent += currentNode
5.          else
                  call insertHelper(value, currentNode.left, Newnode)
6.          fi currentNode.left = Newnode
7.    else
8.          if currentNode.right = null then
                  currentNode.right += Newnode
                  Newnode.parent += currentNode
9.          else
                  call insertHelper(value, currentNode.right, Newnode)
10.         fi currentNode.right = Newnode
11.   fi value < currentNode.value
```

Σε αυτό το σημείο αξίζει να αναφέρουμε πως κλήση της συνάρτησης αντίστροφα μπορεί να γίνει μόνο για αφαίρεση του τελευταίου στοιχείου που προσθέσαμε. Δεν μας επιτρέπει να διαγράψουμε οποιοδήποτε κόμβο. Αυτό είναι λογικό καθώς η διαγραφή στοιχείων όπως θα δούμε και πιο κάτω εφαρμόζεται με διαφορετικούς τρόπους υπό διαφορετικές περιπτώσεις ενώ η εισαγωγή γίνεται πάντα με τον ίδιο τρόπο.

Χρονική και Χωρική Πολυπλοκότητα

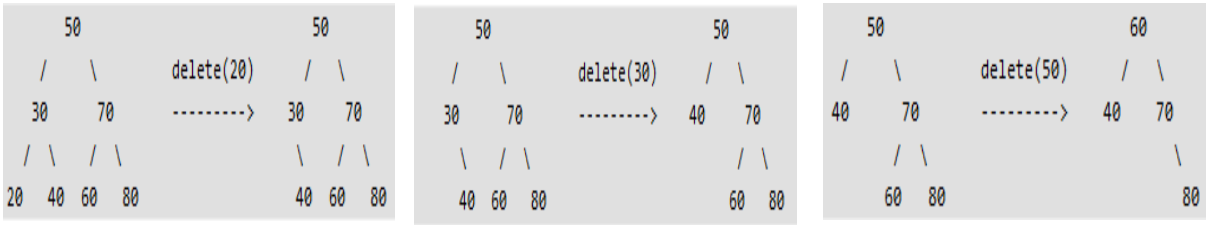
Η προτεινόμενη μέθοδος εισαγωγής σε δυαδικά δέντρα αναζήτησης διατηρεί τη χρονική πολυπλοκότητα της μη αναστρέψιμης μορφής της καθώς στη χείριστη περίπτωση θα διαπεράσουμε όλα τα στοιχεία O(n) ενώ κατά μέσω όρο εκμεταλλευόμενοι τη σχέση των τιμών πατέρα-παιδιών θα διατρέξουμε από τη ρίζα σε κάποιο φύλλο σε χρόνο $O(\log n)$. Επιπρόσθετα η χωρική πολυπλοκότητα διατηρείται καθώς γίνεται χρήση μόνο μίας βοηθητικής μεταβλητής, η οποία μπορεί να θεωρηθεί ως αμελητέα καθώς αυξάνει τη χωρική μας πολυπλοκότητα κατά O(1). Άρα ο αναστρέψιμος αλγόριθμος εισαγωγής σε δυαδικά δέντρα αναζήτησης που προτείνουμε είναι υγιεινός.

3.5 Διαγραφή

Όταν αφαιρείται ένας κόμβος από ένα δυαδικό δένδρο αναζήτησης είναι υποχρεωτικό να διατηρηθεί η ακολουθία στη σειρά των κόμβων. Υπάρχουν πολλές τεχνικές για την επίτευξη του πιο πάνω όμως έχει επικρατήσει η μέθοδος που πρότεινε ο T. Hibbard το 1962 [18], η οποία εγγυάται ότι τα ύψη των υπό υποδένδρων θα αλλάξουν το πολύ κατά ένα.

Αποτελείται από τρεις περιπτώσεις:

1. Διαγραφή κόμβου χωρίς παιδιά: απλά αφαιρούμε τον κόμβο από το δέντρο.
2. Διαγραφή κόμβου με ένα παιδί: αφαιρούμε τον κόμβο και τον αντικαταστούμε με το παιδί του.
3. Διαγραφή κόμβου με δύο παιδιά: Βρίσκουμε τον σε σειρά διάδοχο του κόμβου, δηλαδή το κόμβο με τη μικρότερη τιμή στο δεξιό υποδένδρο. Αντιγράφουμε τα περιεχόμενα του διαδόχου στον κόμβο και διαγράφουμε το διάδοχο. Όμοια μπορεί να χρησιμοποιηθεί και ο σε σειρά προκάτοχος, δηλαδή ο κόμβος με τη μεγαλύτερη τιμή στο αριστερό υποδένδρο.



Σχήμα 6: Παράδειγμα διαγραφής για κάθε περίπτωση

Όπως είναι αντιληπτό οι κόμβοι με δύο παιδιά είναι δυσκολότερο να διαγραφούν. Όπως προαναφέραμε, οι δύο πιο απλοί τρόποι επίλυσης του προβλήματος είναι να διαγράφεται ο σε σειρά διάδοχος ή προκάτοχος, δηλαδή το αριστερότερο παιδί του δεξιά υποδένδρου ενός κόμβου και αντίστοιχα το δεξιότερο παιδί του αριστερά υποδένδρου.

Επομένως για τη διαγραφή ενός κόμβου αρκεί ο εντοπισμός του και έπειτα η εκτέλεση της περίπτωσης της οποίας τηρεί τα κριτήρια. Η χρονική πολυπλοκότητα της διαγραφής ενός κόμβου είναι κατά μέσο όρο $O(\log n)$ και στη χείριστη περίπτωση O(n) καθώς στη χειρότερη περίπτωση θα διασχισθεί το δένδρο από τη ρίζα μέχρι το μακρύτερο φύλλο σε ένα εκφυλισμένο δέντρο. Όμοια δηλαδή με την αναζήτηση και την εισαγωγή. Δεν απαιτεί περισσότερη πολυπλοκότητα ακόμη και όταν διαγράφεται κόμβος με δύο παιδιά, επειδή η αναζήτηση διαδόχου εξακολουθεί να ακολουθεί μια ενιαία πορεία με την αρχική αναζήτηση με αποτέλεσμα να μην υπάρχει περίπτωση να επισκεφθούμε κάποιο κόμβο δύο φορές.

Αναστρέψιμη μορφή

Η διαγραφή αποτελείται από δύο στάδια: αναζητάμε το κόμβο και έπειτα τον διαγράφουμε επιλέγοντας την περίπτωση διαγραφής που πρέπει. Δηλαδή η ίδια διαδικασία με τη μη αναστρέψιμη δομή. Στη γενικευμένη μορφή η εκτέλεση των δύο αυτών συναρτήσεων δεν χρειάζεται κάποια αλλαγή καθώς η αλλαγές αφορούν τη βοηθητική συνάρτηση επιλογής είδους διαγραφής που θα αναλύσουμε αργότερα.

Λόγω του ότι στη Janus δεν είχαμε δείκτες, διαγραφή της ρίζας δεν θα άλλαζε τη τιμή της θέσης-δείκτη της ρίζας. Αυτό οφείλεται στο ότι θα εκτελούσαμε διαγραφή για τη θέση που εντοπίσαμε το κόμβο χωρίς να επηρεάζουμε-αλλάζουμε τη μεταβλητή της ρίζας. Στη γενικευμένη μορφή η ρίζα είναι δείκτης και δεν υπάρχει αυτή η δυσκολία.

Έτσι αρχικά ελέγχουμε αν το στοιχείο που ψάχνουμε είναι η ρίζα. Αν ναι επιλέγουμε ποια περίπτωση διαγραφής πρέπει να εκτελεστεί και την εκτελούμε. Σε περίπτωση που ο κόμβος που θα διαγραφεί δεν είναι η ρίζα εντοπίζουμε το στοιχείο που θέλουμε να διαγράψουμε με τη χρήση της μεθόδου αναζήτησης και έπειτα επιλέγουμε ποια περίπτωση διαγραφής πρέπει να εφαρμοστεί. Σε περίπτωση που θα διαγράψουμε τη ρίζα δεν γίνεται αναζήτηση αλλά επιλογή της αντίστοιχης διαγραφής. Αυτό γίνεται έτσι ώστε στη περίπτωση που διαγράφεται η ρίζα του δέντρου, να ενημερώνεται η μεταβλητή που λειτουργεί ως δείκτης της ρίζας. Ως συνάρτηση αναστρεψιμότητας για τον έλεγχο αυτό θεωρούμε η θέση που αναζητήσαμε να ισούται με μηδέν καθώς κάτι τέτοιο συνεπάγεται πως δεν κάναμε αναζήτηση και επομένως διαγράψαμε τη ρίζα του δέντρου.

Ακολουθεί ο ψευδοκώδικας της διαδικασίας:

```
procedure remove(int valueOut, int root, stack garbage)
      local int pos = 0
      if root.value = valueOut then
            call removeCase(valueOut, root, garbage)
      else
            call find(valueOut, root, pos)
            call removeCase(valueOut, pos, garbage)
      fi pos = 0
      push(pos, remove_garbage)
      delocal int pos = 0
```

Διαδικασία διαγραφής στη Janus

Η θέση του κόμβου, δηλαδή ο διαγραφέντας κόμβος, θεωρείται επιπλέον πληροφορία η οποία δεν μπορεί να αποφευχθεί καθώς μετά την εκτέλεση της διαδικασίας δεν είμαστε σε θέση να τη γνωρίζουμε. Επιπρόσθετα δεν μπορούμε να εφαρμόσουμε τη τεχνική Bennett εκτελώντας αντίστροφα την διαδικασία αναζήτηση καθώς δεν υπάρχει πλέον κόμβος με τη τιμή που αναζητήσαμε νωρίτερα.

Γίνεται χρήση μίας βοηθητικής στοίβας για τη φύλαξη των επιπλέων πληροφοριών που θεωρήσαμε απαραίτητες για την αναστρεψιμότητα. Ο λόγος που χρησιμοποιήθηκε στοίβα και όχι συγκεκριμένος αριθμός μεταβλητών ή πίνακας σταθερού μεγέθους είναι ότι οι επιπλέον πληροφορίες διαφέρουν ανάλογα με την περίπτωση που καλούμαστε να διαγράψουμε. Για παράδειγμα για τη διαγραφή ενός φύλλου φυλάσσουμε δύο τιμές, όπως θα εξηγήσουμε πιο κάτω, ενώ στη περίπτωση που έχουμε δύο παιδιά φυλάσσουμε περισσότερες καθώς εκτός από

τη διαγραφή του επιθυμητού κόμβου θα χρειαστεί να διαγράψουμε και το αριστερότερο παιδί του δεξιού υποδένδρου, δηλαδή το σε σειρά διάδοχο.

Η διαδικασία επιλογής περίπτωσης διαγραφής είναι η εξής:

1. Αν και τα δύο μου παιδιά είναι κενά τότε διαγραφή κόμβου/φύλλου
2. Αλλιώς αν έχω δύο παιδιά τότε διαγραφή κόμβου με δύο παιδιά
3. Αλλιώς αν έχω μόνο δεξιό παιδί τότε διαγραφή κόμβου με δεξί παιδί
4. Αλλιώς διαγραφή κόμβου με αριστερό παιδί
5. **Συνθήκη Αναστρεψιμότητας:** Η τιμή διαγράφηκε είναι μικρότερη από τη τιμή μου
6. **Συνθήκη Αναστρεψιμότητας:** Η κορυφή της βοηθητικής στοίβας έχει αρνητική τιμή
7. **Συνθήκη Αναστρεψιμότητας:** Η τιμή μου είναι κενή

Επεξήγηση των συνθηκών με τη σειρά:

- Αν έχω μόνο ένα παιδί, μετά τη διαγραφή, βρίσκομαι στη θέση του παιδιού. Από τη θεωρία γνωρίζουμε πως το δεξί παιδί έχει μεγαλύτερη τιμή από το πατέρα και το αριστερό μικρότερη και άρα μπορούμε να τα συγκρίνουμε με τη τιμή του κόμβου που διαγράφηκε.
- Αν έχω δύο παιδιά, βρίσκομαι στην πιο δύσκολη περίπτωση καθώς θα χρειαστεί να εντοπίσω το μικρότερο στοιχείο στο δεξιό υποδένδρο και να εκτελέσω μία δεύτερη διαγραφή για αυτό. Για να ξεχωρίζουμε τη περίπτωση αυτή, φυλάμε τη θέση της ελάχιστης τιμής στην αρνητική της μορφή. Από τη στιγμή που φυλάσσουμε θέσεις πίνακα, μόνο υπό αυτές τις συνθήκες οι τιμές στη στοίβα έχουν αρνητικό πρόσημο.
- Αν δεν έχω παιδιά, δηλαδή είμαι φύλλο αρκεί μετά τη διαγραφή να ελέγξω ότι η θέση που βρίσκομαι είναι κενή. Σημείο κλειδί για τη σωστή λειτουργία αυτής της συνθήκης αναστρεψιμότητας αλλά και όλου του αλγόριθμου είναι πως όταν διαγράφουμε κόμβο με παιδί, συνεχίζουμε από το παιδί ενώ όταν διαγράφουμε φύλλο μένουμε στο διαγραφέντα κόμβο. Αυτό έχει ως αποτέλεσμα μόνο όταν εκτελείται διαγραφή φύλλου η τιμή ενός κόμβου και κατ' επέκταση ο τρέχοντας κόμβος να είναι κενός.

Ακολουθεί ο ψευδοκώδικας της διαδικασίας επιλογής περίπτωσης διαγραφής:

```
procedure removeCase(int bst[][], int valueOut, int current, stack garbage)
  local int gb = 0
1.  if bst[current][2] = 0 && bst[current][3] = 0 then
       call removeLeaf(bst, valueOut, current, gb)
       push(gb, garbage)
2.  else
      if bst[current][2] != 0 && bst[current][3] != 0 then
        call removeNodeWithTwoChildren(bst, valueOut, current, gb, garbage)
        local int y = gb * (-1)
        gb <=> y
        delocal int y = gb * (-1)
        push(gb, garbage)
3.    else
        if bst[current][2] = 0 && bst[current][3] != 0 then
           call removeNodeWithRightChild(bst, valueOut, current, gb)
           push(gb, garbage)
4.      else
           call removeNodeWithLeftChild(bst, valueOut, current, gb)
            push(gb, garbage)
5.      fi valueOut < bst[current][0]
6.    fi top(garbage) < 0
7.  fi bst[currentNode][0] = 0
  delocal int gb = 0
```

Διαδικασία επιλογής διαγραφής στη Janus

Στη γενικευμένη μορφή όμως θα έχουμε κάποιες. Καταρχάς δεν θα χρειάζεται η φύλαξη επιπλέον πληροφορίας για διαγραφή κόμβου με μόνο ένα παιδί. Αυτό οφείλεται στο γεγονός πως δεν θα έχουμε ακέραιες θέσεις πίνακα αλλά κόμβους, οι οποίοι μετά την διαγραφή θα είναι κενή. Στη περίπτωση που έχουμε κόμβο με δύο παιδία θα φυλάμε μία αρνητική μεταβλητή, έστω -1, για να ικανοποιείται η δεύτερη συνθήκη. Ακόμη για να λειτουργεί ορθά η συνθήκη αυτή στις άλλες περιπτώσεις θα φυλάμε μία μη αρνητική τιμή.

Ακολουθεί ο ψευδοκώδικας της διαδικασίας στη γενικευμένη μορφή:

```
procedure removeCase(int valueOut, node current, stack garbage)
      local node gb = null
1.    if current.left = null && current.right = null then
         call removeLeaf(valueOut, current, gb)
         push(gb, garbage)
2.    else
         if current.left != null && current.right != null then
            call removeNodeWithTwoChildren(valueOut, current, gb, garbage)
            push(-1, garbage)
3.       else
            if current.left = null && current.right != null then
               call removeNodeWithRightChild(valueOut, current, gb)
               push(0, garbage)
4.          else
               call removeNodeWithLeftChild(valueOut, current, gb)
               push(0, garbage)
5.          fi valueOut < currentNode.value
6.       fi top(garbage) < 0
7.    fi current = null
      delocal node gb = null
```

Οι περιπτώσεις διαγραφής που αναφέραμε πιο πάνω αντιμετωπίζονται με τους εξής τρόπους:

1. Διαγραφή φύλλου: Διαγράφουμε τη τιμή του κόμβου και τον αποσυνδέουμε από το πατέρα του. Ακόμη ελέγχουμε αν έχουμε πατέρα ή όχι, δηλαδή αν είμαστε ρίζα, με συνθήκη αναστρεψιμότητας: η τιμή του πατέρα να μην είναι κενή. Αν έχουμε πατέρα

ελέγχουμε αν είμαστε δεξιό ή αριστερό παιδί και διαγράφουμε αναλόγως το αντίστοιχο πεδίο του πατέρα με συνθήκη αναστρεψιμότητας η τιμή που διαγράψαμε να είναι μικρότερη από τη τιμή του πατέρα.

2. Διαγραφή κόμβου με μόνο ένα παιδί: Οι διαγραφές με μόνο αριστερό ή μόνο δεξιό παιδί είναι πανομοιότυπες. Διαγράφουμε το κόμβο και ενημερώνουμε το παιδί του διαγραφέντα κόμβου πως ο νέος του πατέρας είναι ο παππούς του. Έπειτα ενημερώνουμε το πατέρα του διαγραφέντος κόμβου πως το νέο του παιδί είναι το εγγόνι του, δηλαδή το παιδί του διαγραφέντα κόμβου. Ως συνθήκες αναστρεψιμότητας χρησιμοποιούμε τις ίδιες με αυτές της συνθήκη if. Τέλος κάνουμε τη μεταβλητή που δείχνει στο τρέχον κόμβο, δηλαδή αυτό που διαγράφεται, να δείχνει στη θέση του παιδιού.
3. Διαγραφή κόμβου με δύο παιδιά: Εντοπίζουμε το μικρότερο στοιχείο του δεξιού υποδένδρου. Ενημερώνουμε την τιμή του κόμβου που θέλουμε να διαγράψουμε με τη τιμή που βρήκαμε. Διαγράφουμε κόμβο που περιέχει την ελάχιστη τιμή στο δεξιό υποδένδρο.

Ακολουθεί ο ψευδοκώδικας των τριών ειδών διαγραφής:

```
procedure removeNodeWithRightChild(int valueOut,
node currentNode, node child)
  currentNode.value -= valueOut
  child += currentNode.right
  currentNode.right -= child
  child.parent-= currentNode
  local node father = currentNode.parent
  if father != null then
      currentNode.parent -= father
      if valueOut < father.value then
         father.left -= currentNode
         father.left += child
      else
         father.right -= currentNode
         father.right += child
      fi valueOut < father.value
      child.parent += father
  fi father != null
  delocal node father = child.parent
  child <=> currentNode
```

```
procedure removeLeaf(int valueOut,
node currentNode, node parent)
  currentNode.value -= valueOut
  parent += currentNode.parent
  if parent != null then
      currentNode.parent -= parent
      if valueOut < parent.value then
            parent.left -= currentNode
      else
            parent.right -= currentNode
      fi valueOut < parent.value
  fi parent != null
```

```
procedure removeNodeWithTwoChildren(int valueOut,
node currentNode, node minPos, stack garbage)
  call getMin(currentNode.right, minPos)
  currentNode.value -= valueOut
  currentNode.value += minPos.value
  local int minPosValue = currentNode.value
  call removeCase(minPosValue, minPos, garbage)
  delocal int minPosValue = currentNode.value
```

Η αντίστροφη εκτέλεση του αλγορίθμου επαναφέρει τον κόμβο που διαγράφηκε πιο πρόσφατα. Κάνει δηλαδή κάποιο είδος εισαγωγής όμως δεν μπορεί να χρησιμοποιηθεί για κανονική εισαγωγή κόμβων, καθώς κάτι τέτοιο θα απαιτούσε να γνωρίζουμε τη μορφή του δέντρου από την αρχή ώστε να γεμίζαμε τη βοηθητική στοίβα με τις απαραίτητες πληροφορίες που θα οδηγούσαν στο επιθυμητό αποτέλεσμα, κάτι χρονοβόρο κι σπάταλο. Για παράδειγμα θα έπρεπε να γεμίζαμε εκ των προτέρων τη στοίβα με τις πληροφορίες που θα αποκτούσαμε

αν είχαμε το δέντρο και διαγράφαμε όλους τους κόμβους του, και έπειτα να εκτελούσαμε αντίστροφες διαγραφές για κάθε τιμή του δέντρου μέχρι να άδειαζε η στοίβα.

Χρονική και Χωρική Πολυπλοκότητα

Η χρονική πολυπλοκότητα του αλγορίθμου είναι η ίδια με αυτή της μη αναστρέψιμης μορφής του καθώς ακολουθείται κατά γράμμα χωρίς επιπλέον διαδικασίες που θα επιβραδύνουν τον αλγόριθμο. Οι κόμβοι διαπερνιόνται μόνο μία φορά άρα στη χείριστη περίπτωση έχουμε O(n) ενώ κατά μέσο όρο O($\log n$).

Οι επιπλέον πληροφορίες που χρειαζόμαστε είναι τουλάχιστον δύο. Η μία αφορά τη θέση του κόμβου που διαγράφουμε και η άλλη μας βοηθάει στη σωστή διαχείρισης της εκάστοτε διαγραφής κάνοντας τις σωστές αλλαγές στους επηρεαζόμενους δείκτες. Για παράδειγμα όταν διαγράφουμε κάποιο φύλλο φυλάμε τη θέση του πατέρα ώστε σε περίπτωση αντίστροφης εκτέλεσης να μπορούμε να τους επανασυνδέσουμε. Όπως προαναφέραμε στην περίπτωση διαγραφής κόμβου με δύο παιδιά χρειάζεται να φυλάμε και πληροφορία για τη διαγραφή του κόμβου με την ελάχιστη τιμή στο δεξιό υποδένδρο. Η ελάχιστη τιμή αυτή δεν μπορεί να αντιστοιχεί σε κόμβο με δύο παιδιά καθώς σε αυτή την περίπτωση ελάχιστη τιμή θα θεωρείτο το αριστερό παιδί της, επομένως είναι είτε φύλλο είτε έχει δεξιό παιδί. Συνεπώς χρειάζεται να φυλάμε μόνο μία επιπλέον μεταβλητή σε σχέση με τις άλλες περιπτώσεις.

Άρα συνολικά φυλάμε δύο ή τρεις επιπλέον πληροφορίες. Λόγω αυτού προτιμήθηκε η χρήση στοίβας αντί πίνακα τριών θέσεων ώστε να μην σπαταλιέται μία θέση μνήμης στις περιπτώσεις που δεν διαγράφουμε κόμβο με δύο παιδιά. Ακόμη ένας λόγος που προτιμήθηκε ο χρήση στοίβας είναι ότι θα χρειαζόμασταν ξεχωριστό πίνακα για κάθε εκτέλεση, κάτι που θα μπορούσε να προκαλέσει μεγάλη δυσκολία σε αλγορίθμους όπου εκτελούνται πολλές διαγραφές.

Συνοψίζοντας, στη χείριστη περίπτωση φυλάμε τρεις επιπλέον πληροφορίες επομένως η χωρική πολυπλοκότητα αυξάνεται κατά O(1). Σε περίπτωση που το δέντρο μας είναι πολύ μικρό, μέχρι τρεις κόμβους, η χωρική πολυπλοκότητα μπορεί να θεωρηθεί πως αυξάνεται αισθητά όμως θεωρούμε ανούσιο να έχουμε δέντρο με τόσους λίγους κόμβους και για αυτό πιστεύουμε πως ο προτεινόμενος αλγόριθμος είναι υγιεινός.

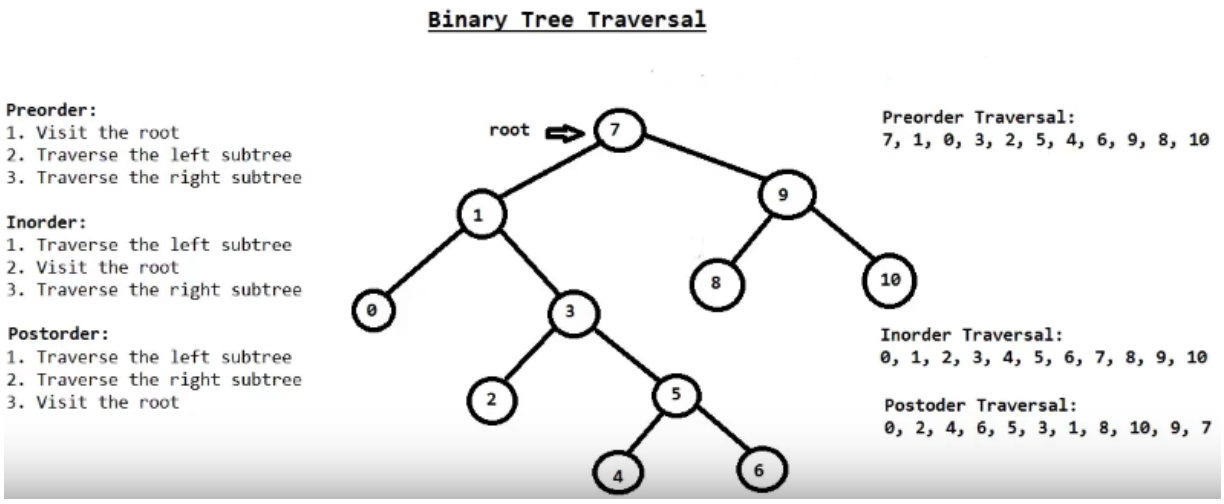
3.6 Διάσχιση

Έχουμε τρία είδη διασχίσεων, σύμφωνα με τα οποία τα στοιχεία ενός δυαδικό δένδρο αναζήτησης μπορούν να ανακτηθούν:

- Προθεματική: Διασχίζουμε τη ρίζα και έπειτα αναδρομικά το αριστερό υποδένδρο και μετά το δεξιό.
- Μεταθεματική: Διασχίζουμε αναδρομικά το αριστερό υποδένδρο, έπειτα το δεξιό και τέλος τη ρίζα.
- Σε σειρά: Διασχίζουμε αναδρομικά το δέντρο από το αριστερό υποδένδρο του κόμβου της ρίζας, έπειτα την ρίζα την ίδια και μετά στο δεξιό υποδένδρο της, συνεχίζοντας αυτό το μοτίβο για κάθε κόμβο του δέντρου.

Η προθεματική ή μεταθεματική διάσχιση, συνήθως δεν είναι τόσο χρήσιμες για τα δυαδικά δέντρα αναζήτησης. Αντίθετα η σε σειρά διάσχιση είναι πολύ σημαντική στα δυαδικά δένδρα αναζήτησης καθώς οδηγεί πάντα σε μια ταξινομημένη λίστα των τιμών των κόμβων.

Στο Σχήμα 6 που ακολουθεί βλέπουμε ένα παράδειγμα εκτέλεσης των τριών αυτών διασχίσεων.



Σχήμα 6: Παράδειγμα διασχίσεων

Είναι προφανές πως οι διασχίσεις απαιτούν O(n) χρόνο, καθώς πρέπει να επισκεφθεί όλους τους κόμβους του δέντρου. Επομένως οι αλγόριθμοι διάσχισης είναι ασυμπτωτικά βέλτιστοι καθώς απαιτούν επίσης O(n).

Αναστρέψιμη μορφή

Όμοια με την αναζήτηση, κατά τη διάσχιση δεν χρειάστηκε να κάνουμε κάποια αλλαγή στη δομή του αλγορίθμου. Αυτό ισχύει και για τις τρεις μορφές προθεματική, μεταθεματική και σε σειρά. Επιπρόσθετα ως συνθήκες αναστρεψιμότητας μπορούμε να χρησιμοποιήσουμε τις ίδιες συνθήκες με αυτές της κανονικής εκτέλεσης καθώς δεν πραγματοποιούμε αλλαγές στο δέντρο, όπως δηλαδή κάναμε και στην αναζήτηση.

Ακολουθεί ο ψευδοκώδικας των τριών διασχίσεων:

```
procedure preOrder(node node)
    if node != null then
        printf("%d", node.value)

        if node.left != null then
            call preOrder(node.left)
        fi node.left != null

        if node.right != null then
            call preOrder(node.right)
        fi node.right != null
    fi node != null
```

```
procedure postOrder(node node)
    if node != null then
        if node.left != null then
            call postOrder(node.left)
        fi node.left != null

        if node.right != null then
            call postOrder(node.right)
        fi node.right != null

        printf("%d", node.value)
    fi node != null
```

```
procedure inOrder(node node)
    if node != null then
        if node.left != null then
            call inOrder(node.left)
        fi node.left != null

        printf("%d", node.value)

        if node.right != null then
            call inOrder(node.right)
        fi node.right != null
    fi node != null
```

Χρονική και Χωρική Πολυπλοκότητα

Λαμβάνοντας αυτά υπόψη, καταλήγουμε στο συμπέρασμα πως η χρονική αλλά και η χωρική πολυπλοκότητα παραμένουν οι ίδιες με τις μη αναστρέψιμες μορφές διάσχισης. Έτσι οι αναστρέψιμες μορφές διασχίσεων σε δυαδικά δέντρα αναζήτησης που προτείνουμε είναι υγιεινές.

Κεφάλαιο 4

AVL Δέντρο

4.1 Γνωριμία με τη δομή

Το AVL δέντρο πήρε το όνομά του από δύο Σοβιετικούς εφευρέτες, τους Georgy Adelson-Velsky και Evgenii Landis, οι οποίοι το δημοσίευσαν στο άρθρο τους «An algorithm for the organization of information» το 1962 [1].

AVL δέντρο ονομάζουμε ένα είδος δυαδικού δένδρου αναζήτησης το οποίο έχει την ιδιότητα της αυτοεξισορρόπησης. Επιπρόσθετα αποτελεί τη πρώτη δομή δεδομένων που εφευρέθηκε με αυτή την ιδιότητα [21]. Σε ένα AVL δέντρο, τα ύψη δύο παιδιών υποδένδρων κάθε κόμβου διαφέρουν το πολύ κατά ένα. Αν σε οποιαδήποτε στιγμή διαφέρουν περισσότερα του ενός, γίνεται εξισορρόπηση του δέντρου με τη χρήση περιστροφών για να αποκατασταθεί η πιο πάνω ιδιότητα. Χάρη στην ιδιότητα αυτή αντιμετωπίζεται το μειονέκτημα που παρουσιάζουν τα δυαδικά δέντρα αναζήτησης σύμφωνα με το οποίο το σχήμα τους εξαρτάται εξ ολοκλήρου από τη σειρά των εισαγωγών και διαγραφών, καθώς εκφυλίζοντάς τα, το δέντρο μετατρέπεται σε συνδεδεμένη λίστα. Μέσω της εν λόγω ιδιότητας εξασφαλίζεται πως στη χειρότερη περίπτωση το ύψος του δέντρου θα είναι $O(\log n)$.

Με τον όρο «ύψος» εννοούμε το μήκος του μονοπατιού από το βαθύτερο κόμβο, φύλλο (στο ίδιο υπό-δένδρο) προς τον εκάστοτε κόμβο. Το ύψος των φύλλων ισούται με ένα και κάθε προγονός του είναι κατά ένα ψηλότερος του.

Αξίζει να επισημάνουμε πως η εξισορρόπηση του δέντρου κατά την εισαγωγή και τη διαγραφή είναι δύο διαφορετικές διαδικασίες καθώς εκτελούν τις περιστροφές υπό διαφορετικές συνθήκες.

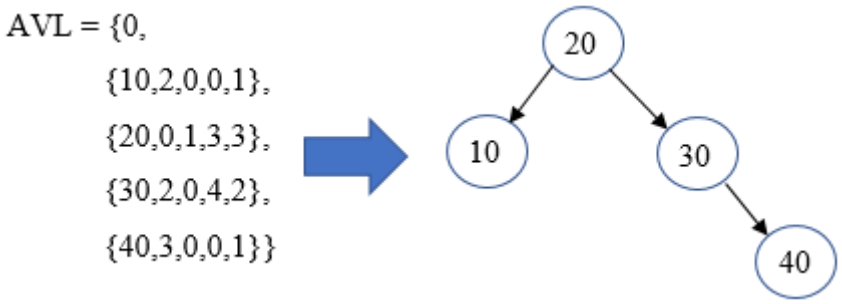
Λόγω του πιο πάνω οι διαδικασίες αναζήτησης, εισαγωγής και διαγραφής στοιχείων/κόμβων έχουν χρονική πολυπλοκότητα O($\log n$) όπου n είναι ο αριθμός των κόμβων στο δέντρο πριν από κάθε διαδικασία, τόσο ως μέσο όρο όσο και ως χείριστη περίπτωση, καθώς γίνεται διάσχιση από τη ρίζα στο μακρύτερο φύλλο. Οι εισαγωγές και οι διαγραφές μπορεί να απαιτήσουν το δέντρο να εξισορροπηθεί από μία ή περισσότερες περιστροφές. Επίσης οι διαδικασίες αναζήτησης και διάσχισης είναι ακριβώς οι ίδιες αυτές των δυαδικών δέντρων αναζήτησης.

4.2 Αναστρέψιμη δομή και περιορισμοί στη Janus

Τα AVL δέντρα αποτελούν ξεχωριστεί κατηγορία δυαδικών δέντρων αναζήτησης επομένως μπορεί να χρησιμοποιηθεί η αναστρέψιμη δομή με χρήση πινάκων επεκτείνοντας την μορφή που προτείναμε στην ενότητα 3.2. Εξαιτίας του ότι η εξισορρόπηση των δέντρων γίνεται με βάση το ύψος τους, χρειάζεται να προσθέσουμε ακόμη μία στήλη-πεδίο που θα αναφέρεται στο ύψος του κάθε κόμβου. Αυτό επιτυγχάνεται προσθέτοντας ακόμη μία στήλη στο πίνακα.

Η νέα μορφή του κόμβου μας θα είναι:

- Στήλη 0 = τιμή που φυλάσσεται στον κόμβο
- Στήλη 1 = θέση του πατέρα
- Στήλη 2 = θέση αριστερού παιδιού
- Στήλη 3 = θέση δεξιού παιδιού
- Στήλη 4 = ύψος κόμβου



Σχήμα 7: Παράδειγμα AVL δέντρου

Αν κάθε κόμβος ήταν αντικείμενο για την επιπλέον πληροφορία ύψος θα χρειαζόταν μία επιπλέον μεταβλητή. Όμοια με την προτεινόμενη μορφή μας η προσθήκη νέας πληροφορίας χρειάζεται την προσθήκη νέας στήλης. Άρα έχουμε τη δυνατότητα να επεκτείνουμε ακόμη περισσότερο τη δομή μας, φυλάσσοντας σε κάθε κόμβο περισσότερες πληροφορίες-στοιχεία εύκολα και όμοια με το αν είχαμε αντικείμενα. Για παράδειγμα σε περίπτωση υλοποίησης των Red-Black δέντρων θα προσθέταμε ακόμη μία στήλη για να φυλάμε το χρώμα του εκάστοτε κόμβου.

Λαμβάνοντας υπόψη πως διατηρήσαμε τη δομή των δυαδικών δέντρων αναζήτησης και το ότι οι διαδικασίες αναζήτησης και διάσχισης δεν επηρεάζονται από την ανάγκη εξισορρόπησης του δέντρου με τη χρήση περιστροφών καταλήξαμε στο συμπέρασμα πως για τις δύο αυτές διαδικασίες μπορούμε να χρησιμοποιούμε τις αντίστοιχες που προτείναμε για τα δυαδικά δέντρα αναζήτησης και για αυτό παραλείπονται.

Γίνεται εύκολα αντιληπτό πως η κάθε γραμμή του πίνακα αποτελεί ξεχωριστό αντικείμενο. Έτσι θα προτείνουμε υλοποιήσεις που θα έχουν μία αφαιρετική περιγραφή ώστε να δείχνουμε την αναστρεψιμότητα τους ανεξαρτήτως γλώσσας προγραμματισμού και τυχών περιορισμών της. Η υλοποίηση μας στη γλώσσα προγραμματισμού Janus βρίσκεται στο Παράρτημα Β και χρησιμοποιήθηκε κυρίως ως μέσω ελέγχου και επαλήθευσης των προτάσεων μας.

Στα σημεία όμως που αντιμετωπίσαμε δυσκολίες λόγω της χρήσης πίνακα ή άλλων περιορισμών της γλώσσας κατά την υλοποίηση θα κάνουμε αναφορά στη δυσκολία αυτή και πως την επιλύσαμε.

Συγκεκριμένα:

- Αν θέλουμε να αναφερθούμε στο πατέρα κάποιου κόμβου X δεν θα λέμε η θέση [X, 1] του πίνακα αλλά ο πατέρας του κόμβου X, δίνοντας έτσι τη μορφή των συναρτήσεων σε περίπτωση που είχαμε δείκτες.

- Αναφορά σε κάποιο κόμβο X θα γίνεται ονομαστικά, υποθέτοντας πως έχουμε αντικείμενα τύπου κόμβου (node) και όχι ως γραμμή πίνακα όπως για τη Janus. Για παράδειγμα θα λέμε ο κόμβος X και όχι η γραμμή X του πίνακα-δέντρου μας.
- Το κενό θα συμβολίζεται με null ενώ για τη Janus θα ελέγχουμε αν ο κόμβος έχει τιμή μηδέν.

4.3 Ενημέρωση ύψους κόμβων

Η ενημέρωση του ύψους ενός κόμβου είναι μία πολύ απλή διαδικασία. Αρκεί να συγκρίνουμε το ύψος των παιδιών μας και να πάρουμε τη τιμή του ψηλότερου αυξάνοντας τη κατά ένα. Η διαδικασία έχει χρονική πολυπλοκότητα O(1) καθώς αποτελείται από μόνο μία απλή σύγκριση:

Αν το αριστερό παιδί είναι ψηλότερο από το δεξιό
  Θέσε το ύψους του τρέχοντος κόμβου με αυτό του αριστερού παιδιού συν 1
Αλλιώς
  Θέσε το ύψους του τρέχοντος κόμβου με αυτό του δεξιού παιδιού συν 1

Αναστρέψιμη μορφή

Λόγω της ύπαρξης μη κενού χαρακτήρα στη γλώσσα προγραμματισμού Janus παρουσιάστηκε μία δυσκολία σε περίπτωση που δεν υπήρχε κάποιο από τα παιδιά μας. Για την αντιμετώπιση του προβλήματος είχαμε δύο επιλογές:

1. Προσθήκη ελέγχων που θα δυσκόλευαν την κατανόηση του αλγορίθμου.
2. Ορισμός του κενού ως μηδέν και το δέντρο να ξεκινά από τη θέση ένα

Επιλέγηκε η δεύτερη λύση, διότι θεωρήσαμε πως μία γραμμή πίνακα δεν είναι τόσο μεγάλη θυσία. Επίσης προτιμήθηκε ώστε ο αλγόριθμος μας να μην ξεφεύγει από τη μη αναστρέψιμη μορφή του αλγορίθμου στην προσπάθεια του να επιλύσει θέματα που αφορούσαν τη χρήση πίνακα. Με αυτό το τρόπο όταν ένα παιδί κάποιου κόμβου ήταν κενό θα πραγματοποιείτο σύγκριση με το ύψος της θέσης μηδέν. Εξ ορισμού η θέση μηδέν είναι άδεια και επομένως το ύψος της είναι επίσης μηδέν. Από τη στιγμή που οι κοντύτεροι κόμβοι μας, τα φύλλα του δέντρου, έχουν ύψος ένα, αυτό δεν θα επηρέαζε το αλγόριθμό μας. Κοντά σε αυτά ο αλγόριθμος εκτελείται μόνο σε μη κενούς κόμβους ώστε να μην υπάρξει περίπτωση λάθος ενημερώσεων.

Ο αλγόριθμος μας πραγματοποιεί ενημέρωση ύψους με τον ίδιο τρόπο που γίνεται και στη μη αναστρέψιμη μορφή. Ως συνθήκη αναστρεψιμότητας χρησιμοποιείται η ίδια συνθήκη με τη κανονική σύγκριση καθώς δεν επηρεάζονται οι τιμές των κόμβων που εμπλέκονται στη σύγκριση και επομένως η συνθήκη αυτή ισχύει και προς τα εμπρός αλλά και προς τα πίσω.

Όμως δεν είμαστε σε θέση να γνωρίζουμε αν το τρέχον ύψος πρέπει να αυξηθεί ή να μειωθεί και πόσο. Για παράδειγμα, αν στο Σχήμα 7 προσθέταμε παιδί στο κόμβο 10, αυτό θα είχε ως αποτέλεσμα το ύψος του κόμβου 10 να αυξηθεί κατά ένα όμως το ύψος του κόμβου 20 θα παρέμενε το ίδιο. Ακόμη σε περίπτωση διαγραφής του κόμβου 10 το ύψος του κόμβου 20 από τρία θα έπρεπε να μειωθεί σε ένα, καθώς όπως θα εξηγήσουμε αργότερα θα γινόταν φύλλο.

Για το λόγο αυτό κρίναμε απαραίτητη τη φύλαξη μίας επιπλέον πληροφορίας που θα περιέχει το ύψος του τρέχοντος κόμβου πριν την ενημέρωση του ώστε να είμαστε σε θέση να επανέλθουμε σε αυτό. Ακόμη εξαιτίας της μη υποστήριξης άμεσης ανάθεσης τιμών αναγκαστήκαμε να μηδενίζουμε το ύψος του κόμβου πριν να το ενημερώσουμε. Γνωρίζοντας αυτά και λαμβάνοντας υπόψη πως η συγκεκριμένη λειτουργία αποτελεί στην ουσία βοηθητική συνάρτηση των εισαγωγών, διαγραφών και περιστροφών όπου μπορεί να χρειαστεί η κλήση της αρκετές φορές κρίναμε πως θα ήταν βοηθητικό η φύλαξη της τιμής να γίνεται σε στοίβα και όχι κάποια μεταβλητή. Με το τρόπο αυτό περιορίζουμε την διαχείριση της επιπλέον πληροφορίας που προκύπτει εντός της συνάρτησης όσες φορές και αν χρειαστεί να εκτελεστεί.

Ακολουθεί ο ψευδοκώδικας της διαδικασίας:

```
procedure updateHeight(node currentNode, stack height)
      local node left = currentNode.left, right = currentNode.right
      if !( currentNode = null ) then
            local int t = currentNode.height
            currentNode.height -= t
            push(t, height)
            delocal  int t = 0

            if left.height > right.height then
                  currentNode.height += left.height + 1
            else
                  currentNode.height += right.height + 1
            fi left.height > right.height
      fi !( currentNode = null )
      delocal node left = currentNode.left, right = currentNode.right
```

Χρονική και Χωρική Πολυπλοκότητα

Σε κάθε εκτέλεση χρειάζεται η φύλαξη μόνο μίας πληροφορίας, άρα η χωρική πολυπλοκότητα αυξάνεται κατά O(1), ποσότητα που μπορεί να θεωρηθεί αμελητέα και έτσι έχουμε υγιεινή αναστρέψιμη μορφή της ενημέρωσης ύψους.

4.4 Περιστροφές

Όπως προαναφέραμε η εξισορρόπηση των δέντρων γίνεται με τη χρήση περιστροφών. Υπάρχουν τέσσερις διαφορετικές περιπτώσεις διαχείρισης των κόμβων x, y, z όπου z η ρίζα του υποδένδρου:

1. y είναι αριστερό παιδί του z και ο x είναι αριστερό παιδί του y (Αριστερή περιστροφή)
2. y είναι αριστερό παιδί του z και ο x είναι δεξιό παιδί του y (Αριστερή Δεξιά περιστροφή)
3. y είναι δεξιό παιδί του z και ο x είναι δεξιό παιδί του y (Δεξιά περιστροφή)
4. y είναι δεξιό παιδί του z και ο x είναι αριστερό παιδί του y (Δεξιά Αριστερή περιστροφή)

Η Αριστερή Δεξιά και Δεξιά Αριστερή περιστροφές έχουν δύο τρόπους υλοποίησης, είτε κάνοντας κλήση των Αριστερά και Δεξιά περιστροφών είτε κάνοντας τις αλλαγές αυτόνομα. Κατά τις περιστροφές γίνονται οι εξής αλλαγές:

```
Right Rotate (node y):
    x = y.left
    t2 = x.right
    x.right= y
    y.left =  t2
    update Height(y)
    update Height(x)
    return x
```

```
Left Rotate (node x):
    y = x.right
    t2 = y.left
    y.left = x
    x.right =  t2
    update Height(x)
    update Height(y)
    return y
```

Ακολουθεί απλό παράδειγμα για κάθε περίπτωση περιστροφής ξεχωριστά για καλύτερη κατανόηση.

a) Left Left Case

```
T1, T2, T3 and T4 are subtrees.
       z                                      y
      / \                                    /  \
     y   T4      Right Rotate (z)           x     z
    / \          - - - - - - - - - ->      /  \   / \
   x   T3                                 T1  T2  T3  T4
  / \
 T1   T2
```

b) Left Right Case

```
    z                            z                           x
   / \                          /  \                        /  \
  y   T4  Left Rotate (y)      x    T4  Right Rotate(z)    y      z
 / \      - - - - - - - - - ->  /  \    - - - - - - - - -> / \    / \
T1   x                        y    T3                    T1  T2 T3  T4
    / \                      / \
   T2   T3                  T1   T2
```

c) Right Right Case

```
  z                              y
 /  \                           /  \
T1   y     Left Rotate(z)      z     x
    / \    - - - - - - - - ->  / \   / \
   T2   x                     T1  T2 T3  T4
       / \
      T3  T4
```

d) Right Left Case

```
   z                          z                            x
  / \                        / \                          /  \
T1   y   Right Rotate (y)   T1   x      Left Rotate(z)   z      y
    / \  - - - - - - - - ->     / \    - - - - - - - - -> / \    / \
   x   T4                     T2   y                    T1  T2  T3  T4
  / \                            / \
 T2   T3                        T3   T4
```

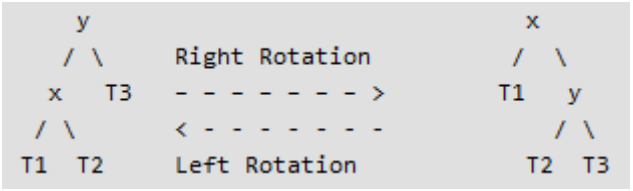
Σχήμα 8: Είδη περιστροφών σε AVL δέντρα

Με βάση τους πιο πάνω ψευδοκώδικες και λαμβάνοντας υπόψη πως η ενημέρωση του ύψους του κάθε κόμβου όπως είδαμε πιο πάνω είναι O(1) γίνεται εύκολα αντιληπτό πως η χρονική πολυπλοκότητα είναι των περιστροφών είναι O(1) καθώς εκτελούνται απλές αναθέσεις τιμών.

Αναστρέψιμη μορφή

Παρατηρούμε πως οι περιστροφές απλά αλλάζουν τους δείκτες των επηρεαζόμενων κόμβων, στη περίπτωση μας τις αντίστοιχες θέσεις πίνακα. Λαμβάνοντας υπόψη τη παρατήρηση αυτή, υλοποιήσαμε πανομοιότυπες μορφές των περιστροφών με αυτές της μη αναστρέψιμης μορφής. Η μεγαλύτερη δυσκολία σημειώθηκε στην ενημέρωση των πατεράδων των επηρεαζόμενων κόμβων καθώς έπρεπε να είμαστε πολύ προσεκτικοί ώστε να εντοπίσουμε της σωστές αλλαγές που γίνονται.

Έπειτα όμως αντιληφθήκαμε κάτι πολύ ενδιαφέρον. Οι δύο περιστροφές αριστερά και δεξιά είναι αντίστροφες η μία της άλλης όσο αφορά τις αλλαγές στις ενώσεις των εμπλεκόμενων κόμβων.



Σχήμα 9: Αναστρεψιμότητα περιστροφών

Δυστυχώς όμως οι δύο συναρτήσεις κάνουν χρήση της διαδικασίας ενημέρωσης υψών που θα δημιουργούσε προβλήματα στη περίπτωση εισαγωγής τριών στοιχείων σε φθίνουσα ή αύξουσα σειρά που θα απαιτούσε αντίστροφη εκτέλεση της αριστερής ή δεξιάς περιστροφής. Εξαιτίας αυτού κάναμε χρήση διαφορετικών συναρτήσεων για κάθε περιστροφή κατά την υλοποίηση των AVL δέντρων.

Αντίθετα, όπως θα δούμε στο επόμενο κεφάλαιο, για τα Splay δέντρα εκμεταλλευτήκαμε την παρατήρηση αυτή και κάναμε χρήση μίας περιστροφής καθώς δεν απαιτείται η χρήση της συνάρτησης ενημέρωσης υψών σε αυτά. (βλέπε Υποκεφάλαιο 5.3)

Στα πλαίσια της διπλωματικές εργασίας για υλοποίηση των περιπτώσεων που απαιτούν δύο είδη περιστροφών εκτελούμε πρώτα τη μία και μετά την άλλη. Οι περιστροφές μας είναι πανομοιότυπες τόσο μεταξύ τους όσο και με τις κανονικές τους μορφές.

Ακολουθεί η αριστερή περιστροφή, η οποία κάνοντας τις αντίστοιχες αλλαγές με τις κανονικές μορφές μετατρέπεται σε δεξιά. Η περιστροφή που προτείνουμε έχει ως ακολούθως:

1. Έστω x ο κόμβος μας, y το δεξιό μας παιδί και t το αριστερό παιδί του y
2. Κάνε το x το νέο αριστερό παιδί του y
3. Κάνε το t το νέο δεξιό παιδί του x
4. Αν ο t δεν είναι κενός
    a. Αντάλλαξε το πατέρα του x με το πατέρα του t
    b. Αντάλλαξε το πατέρα του t με το πατέρα του y
5. Αλλιώς
    a. Αντάλλαξε το πατέρα του x με το πατέρα του y
    b. Κάνε το y το νέο πατέρα του x
6. **Συνθήκη αναστρεψιμότητας:** ο τ δεν είναι κενός
7. Αναίρεσε το κόμβο t
8. Ενημέρωσε το ύψος του x
9. Ενημέρωσε το ύψος του y
10. Όρισε ως νέα ρίζα του υποδένδρου το y

Ακολουθεί ο ψευδοκώδικας της διαδικασίας:

```
procedure leftRotate(node x, stack height)
1.    local node y = x.right, t = y.left
2     y.left -= t
      y.left += x
3.    x.right -= y
      x.right += t

4.    if t != null then
4.a         x.parent <=> t.parent
4.b         t.parent <=> y.parent
5.    else
5.a         x.parent <=> y.parent
5.b         local node a = x
            x.parent -= a
            x.parent += y
            delocal node a = x
6.    fi t != null

7.    delocal node t = x.right

8.    call updateHeight(x, height)
9.    call updateHeight(y, height)

10.   x -= y.left
      x += y
      delocal node y = x
```

Χρονική και Χωρική Πολυπλοκότητα

Παρατηρούμε πως οι περιστροφές εκτελούν αρκετές πράξεις γραμμικά. Επομένως έχουν χρονική πολυπλοκότητα O(1). Ακόμη γνωρίζουμε πως η ενημέρωση του ύψους γίνεται σε

χρόνο Ο(1). Έτσι η συνολική χρονική πολυπλοκότητα της διαδικασίας είναι Ο(1), η ίδια δηλαδή με την μη αναστρέψιμη της μορφή.

Όσο αφορά τη χωρική πολυπλοκότητα αυτή εξαρτάται εξ' ολοκλήρου από την ενημέρωση του ύψους. Έχουμε δείξει πως η εν λόγω συνάρτηση αυξάνει τη χωρική πολυπλοκότητα κατά μία μεταβλητή, δηλαδή Ο(1). Κατά την περιστροφή ενημερώνουμε το ύψος δύο κόμβων επομένως χρειάζεται να φυλάμε δύο επιπλέον πληροφορίες, ποσότητα αμελητέα. Επομένως η προτεινόμενη αναστρέψιμη μορφή περιστροφών έχει χωρική πολυπλοκότητα Ο(1) και είναι υγιεινή.

4.5 Αναπροσαρμογή δέντρου κατά την εισαγωγή

Αυτή η συνάρτηση είναι υπεύθυνη για την επιλογή της περιστροφής που θα εκτελεστεί αν κριθεί αναγκαίο. Γνωρίζουμε πως έχουμε τέσσερα είδη περιστροφών και ότι η διαφορά του ύψους ανάμεσα σε δύο υποδένδρα πρέπει να είναι το πολύ ένα. Επομένως έχουμε τέσσερεις συνθήκες οι οποίες θα καθορίσουν ποια περιστροφή θα εκτελεστεί αν χρειάζεται:

1. Αν το αριστερό υποδένδρο είναι κατά τουλάχιστον 2 ψηλότερο
2. Αν η τιμή που εισάγουμε είναι μεγαλύτερη της ρίζας του αριστερού υποδένδρου
3. Αριστερή περιστροφή στη ρίζα του αριστερού υποδένδρου
4. Δεξιά περιστροφή στο τρέχον κόμβο
5. Αλλιώς το αριστερό υποδένδρο είναι κατά τουλάχιστον 2 χαμηλότερο
6. Αν η τιμή που εισάγουμε είναι μικρότερη της ρίζας του δεξιού υποδένδρου
7. Δεξιά περιστροφή στη ρίζα του αριστερού υποδένδρου
8. Αριστερή περιστροφή στο τρέχον κόμβο

Η συνάρτηση αποτελείται από απλές συγκρίσεις, οι οποίες κάνουν χρήση της συνάρτησης περιστροφών. Άρα η χρονική της πολυπλοκότητα εξαρτάται από τη χρονική πολυπλοκότητα των περιστροφών. Είδαμε νωρίτερα πως οι περιστροφές έχουν χρονική πολυπλοκότητα Ο(1) με αποτέλεσμα και η αναπροσαρμογή του δέντρου να απαιτεί χρόνο Ο(1).

Αναστρέψιμη μορφή

Έχοντας αναστρέψιμες μορφές των περιστροφών αρκεί να επιλέξουμε συνθήκες αναστρεψιμότητας και η διαδικασία μας γίνεται αναστρέψιμη. Έπειτα από έντονο προβληματισμό καταλήξαμε πως είναι αρκετά δύσκολο έως αδύνατο να εντοπίσουμε συνθήκες αναστρεψιμότητας χωρίς να χρειαστεί η φύλαξη επιπλέον πληροφορίας λόγω των

πάρα πολλών υποπεριπτώσεων που μπορεί να προκύψουν από τη εκτέλεση διαφορετικών περιστροφών. Για το λόγο αυτό αποφασίσαμε πως θα χρειαστεί να φυλάμε τουλάχιστον μία επιπλέον πληροφορία.

Έτσι κρίναμε σωστό να κάνουμε χρήση μίας μόνο μεταβλητής-σημαίας που θα έπαιρνε διαφορετικές τιμές για κάθε μία από τις πέντε (λαμβάνουμε υπόψη και τη μη εκτέλεση) περιπτώσεις περιστροφών:

0. Καμία περιστροφή
1. Δεξιά Περιστροφή
2. Αριστερή περιστροφή
3. Αριστερή Δεξιά περιστροφή
4. Δεξιά Αριστερή περιστροφή

Η μεταβλητή αυτή θα λειτουργούσε ως σημαία στο κώδικα μας και ανάλογα με τη κάθε περίπτωση ως συνθήκη αναστρεψιμότητας θα θέταμε η μεταβλητή αυτή να ισούται με την τιμή που αντιστοιχεί στη συγκεκριμένη περιστροφή. Έτσι ο αλγόριθμος μας θα έπαιρνε την εξής μορφή:

1. Θέσε τη σημαία ίση με 0
2. Αν το αριστερό υποδένδρο είναι κατά τουλάχιστον 2 ψηλότερο
3. Αν η τιμή που εισάγεις είναι μεγαλύτερη της ρίζας του αριστερού υποδένδρου
4. Αριστερή περιστροφή στη ρίζα του αριστερού υποδένδρου και πρόσθεσε 2 στη σημαία
5. **Συνθήκη Αναστρεψιμότητας:** Η σημαία ισούται με 2
6. Δεξιά περιστροφή στο τρέχον κόμβο και πρόσθεσε 1 στη σημαία
7. Αλλιώς αν το αριστερό υποδένδρο είναι κατά τουλάχιστον 2 χαμηλότερο
8. Αν η τιμή που εισάγεις είναι μικρότερη της ρίζας του δεξιού υποδένδρου
9. Δεξιά περιστροφή στη ρίζα του αριστερού υποδένδρου και πρόσθεσε 2 στη σημαία
10. **Συνθήκη Αναστρεψιμότητας:** Η σημαία ισούται με 2
11. Αριστερή περιστροφή στο τρέχον κόμβο και πρόσθεσε 2 στη σημαία
12. **Συνθήκη Αναστρεψιμότητας:** Η σημαία ισούται με 2 ή 4
13. **Συνθήκη Αναστρεψιμότητας:** Η σημαία ισούται με 1 ή 3
14. Φύλαξε τη σημαία στη βοηθητική στοίβα

Όμοια με την ενημέρωση των υψών η μεταβλητή φυλάσσεται σε στοίβα και δεν επιστρέφεται έτσι ώστε η επεξεργασία της επιπλέον πληροφορίας να παραμένει εντός της συνάρτησης. Επίσης με αυτό το τρόπο παρέχεται η δυνατότητα εύκολης επαναχρησιμοποίησης της

συνάρτησης καθώς διαφορετικά θα ήταν απαραίτητη η δημιουργία νέας μεταβλητής για κάθε εκτέλεση της συνάρτησης, κάτι επιθυμητό καθώς η συγκεκριμένη λειτουργία εκτελείται αρκετές φορές σε κάθε εισαγωγή και διαγραφή.

Ακολουθεί ο ψευδοκώδικας της διαδικασίας:

```
procedure balance(int value, node currentNode, stack rotate, stack height)
1.    local int flag = 0
2.    if currentNode.left.height - currentNode.right.height > 1 then
3.          if value > currentNode.left.value then
                  local node left = currentNode.left
                  currentNode.left -= left
4.                call leftRotate(left, height)
                  currentNode.left += left
                  delocal node left = currentNode.left
                  flag += 2
5.          fi flag = 2

6.          call rightRotate(currentNode, height)
            flag += 1
7.    else
            if currentNode.left.height - currentNode.right.height < -1 then
8.                if value < currentNode.right.value then
                        local node right = currentNode.right
                        currentNode.right -= right
9.                      call rightRotate(right, height)
                        currentNode.right += right
                        delocal node right = currentNode.right
                        flag += 2
10.               fi flag = 2

11.               call leftRotate(currentNode, height)
                  flag += 2
12.         fi flag = 2 || flag = 4
13.   fi flag = 1 || flag = 3
14.   push(flag, rotate)
      delocal int flag = 0
```

Σε αυτό το σημείο να αναφέρουμε η γλώσσα προγραμματισμού Janus, επιτρέπει κατά την κλήση συναρτήσεων να αλλάζουν τιμή μόνο οι ακέραιοι παράμετροι. Για το λόγο αυτό στην υλοποίηση μας, δημιουργούσαμε μια τοπική μεταβλητή-δείκτη στη ρίζα του υποδένδρου όπου θα συνεχίζαμε την αναζήτηση και την αναιρούσαμε και κατά την έξοδο από τη συνάρτηση. Η τεχνική αυτή καλύπτει και τις περιπτώσεις αναδρομικής κλήσης. Η τεχνική αυτή είναι εφικτή διότι τυχόν αλλαγές γίνονται εντός του υποδένδρου και έτσι η βοηθητική μεταβλητή, και πριν αλλά και μετά την εκτέλεση θα αποτελεί την ακμή ένωσης του τρέχοντος κόμβο με το αντίστοιχο παιδί του. Υποθέτουμε πως στη γενικευμένη μορφή του αλγορίθμου όπου δεν υπάρχει ο περιορισμός αυτός για τις αναδρομικές κλήσεις η τεχνική αυτή δεν χρειάζεται, καθώς θα είμαστε σε θέση να αλλάζουμε τις τιμές των κόμβων όμοια με τους ακέραιους.

Για παράδειγμα στη πιο πάνω διαδικασία η τεχνική εφαρμόζετε στη τοπική μεταβλητή left/right, δηλαδή πριν και μετά τα βήματα 4 και 9 (στην υλοποίηση μας η μεταβλητή αποτελεί

ακέραιο που αντιστοιχεί στη θέση του παιδιού). Στη γενικευμένη μορφή δεν χρειάζεται η τεχνική αυτή όμως θεωρήσαμε χρήσιμο να την δείχνουμε στο ψευδοκώδικα μας όπου χρειάζεται ώστε να γίνετε εύκολα αντιληπτό που χρησιμοποιήθηκε. Σχετικά με τον αλγόριθμο μας η τεχνική κάνει χρήση 2 πράξεων χωρίς να φυλάει επιπλέον πληροφορία επομένως δεν αλλοιώνει τη πολυπλοκότητα του αλγορίθμου.

Η τεχνική εφαρμόζεται σε αρκετές συναρτήσεις και η επεξήγηση της στις μετέπειτα συναρτήσεις παραλείπεται.

Χρονική και Χωρική Πολυπλοκότητα

Η αναστρέψιμη μορφή ακολουθεί την ίδια διαδικασία με τη μη αναστρέψιμη επομένως διατηρεί τη χρονική της πολυπλοκότητα σε O(1). Σχετικά με τη χωρική πολυπλοκότητα όμως έχουμε κάποιες μικρές αλλαγές. Λόγω των περιστροφών χρειάζεται να φυλάμε δύο ή τέσσερις επιπλέον πληροφορίες, σχετικά με τα παλιά ύψη των κόμβων που επηρεάζονται, ανάλογα με το είδος περιστροφής (απλή ή διπλή) που θα εκτελεστεί. Επιπλέον θα φυλάμε και τη μεταβλητή-σημαία για τους λόγους που αναφέραμε νωρίτερα. Συνολικά θα χρειάζεται να φυλάμε το πολύ πέντε επιπλέον πληροφορίες. Για μικρά δέντρα αυτή η αύξηση της χωρικής πολυπλοκότητας είναι αρκετά μεγάλη όμως για δέντρα με πολλούς κόμβους μπορεί να θεωρηθεί αμελητέα. Για n μικρότερο του πέντε, μπορούμε να υλοποιήσουμε μία πιο απλή μορφή AVL δέντρου ώστε να χειριζόμαστε τις περιπτώσεις του.

Συνοψίζοντας η λύση μας είναι υγιεινή καθώς η φύλαξη το πολύ πέντε επιπλέον πληροφοριών μπορεί να θεωρηθεί O(1). Σε περίπτωση όμως που προταθεί αναστρέψιμη μορφή ενημέρωσης ύψους η οποία δεν θα απαιτεί τη φύλαξη επιπλέον πληροφορίας για το παλιό ύψος του εκάστοτε κόμβου τότε η συνάρτηση μας γίνεται αυτόματα υγιεινή.

4.6 Εισαγωγή

Η εισαγωγή κόμβου γίνεται όπως ακριβώς και σε ένα δυαδικό δένδρο αναζήτησης, με τη διαφορά ότι καταγράφουμε τη διαδρομή που ακολουθείται (από τη ρίζα προς τα φύλλα). Στη συνέχεια, ακολουθούμε τη διαδρομή προς τα πίσω και ενημερώνουμε το ύψος των κόμβων. Αν αυτό προκαλέσει κάποια ανισοζυγία, δηλαδή, αν έχει σαν αποτέλεσμα κάποιος κόμβος να έχει παιδιά που το ύψος τους διαφέρει περισσότερο από ένα, τότε αναπροσαρμόζουμε τα υποδένδρα ώστε το δένδρο να γίνει ξανά AVL .

Πρέπει να σημειώσουμε πως μετά την εισαγωγή του νέου κόμβου, ελέγχονται αναδρομικά όλοι οι κόμβοι από τους οποίους διαπεράσαμε διότι το αποτέλεσμα της εισόδου ή της περιστροφής σε κάποιο υποδένδρο μπορεί να προκαλέσει ανισορροπία σε κάποιο από τους προγόνους του.

Με βάση αυτά η εισαγωγή στα AVL δέντρα αποτελείται από 4 βήματα:

1. Εκτελούμε κανονική εισαγωγή Δυαδικού Δέντρου Αναζήτησης.
2. Ο τρέχον κόμβος πρέπει να είναι ένας από τους προγόνους του πρόσφατα εισαχθέντος κόμβου. Ενημερώνουμε το ύψος του τρέχοντος κόμβου.
3. Εκτελούμε τη συνάρτηση αναπροσαρμογής.
4. Επαναλαμβάνουμε αναδρομικά τα βήματα 2 και 3 για τους κόμβους που επισκεφθήκαμε στο βήμα 1 μέχρι να επιστρέψουμε στη ρίζα.

Αναστρέψιμη μορφή

Γίνεται και πάλι χρήση ενός βοηθητικού δείκτη που μας δείχνει τη θέση του πίνακα όπου θα εισάγουμε το επόμενο-νέο στοιχείο. Όμοια με την εισαγωγή στα δυαδικά δέντρα αναζήτησης, στη γενικευμένη μορφή όπου έχουμε αναστρέψιμα αντικείμενα και δείκτες η μεταβλητή αυτή παραλείπεται, καθώς απλά θα δημιουργείτε ένα νέο αντικείμενο.

Αρχικά θέτουμε στο νέο κόμβο τη τιμή του και αρχικοποιούμε το ύψος του με τη τιμή ένα. Έπειτα ξεκινάμε τη διαδικασία εισαγωγής του νέου μας κόμβου στο δέντρο. Όταν εισαχθεί ο νέος κόμβος, επιστρέφουμε αναδρομική στη ρίζα ενημερώνοντας τα ύψη των επηρεαζόμενων κόμβων, δηλαδή των κόμβων που διαπεράσαμε, και κάνουμε αναπροσαρμογές όπου χρειάζεται ώστε να διατηρηθεί η ιδιότητα των AVL δέντρων.

Ο αλγόριθμος μας κάνει χρήση των συναρτήσεων ενημέρωσης ύψους και αναπροσαρμογής. Έχουμε ήδη δει τις αναστρέψιμες μορφές των δύο αυτών συναρτήσεων και γνωρίζουμε πως διατηρούν τη χρονική πολυπλοκότητα των μη αναστρέψιμων μορφών τους χρησιμοποιώντας ελάχιστες επιπρόσθετες πληροφορίες οι οποίες για μεγάλα δέντρα μπορούν να θεωρηθούν αμελητέες. Δυστυχώς όμως κατά τη διαδικασία της εισαγωγής οι δύο αυτές συναρτήσεις εκτελούνται αρκετές φορές με αποτέλεσμα να συσσωρεύεται αρκετή επιπρόσθετη πληροφορία.

Ακόμη δεν μπορούμε να χρησιμοποιήσουμε τις συνθήκες αναστρεψιμότητας που είχαμε κατά την εισαγωγή σε δυαδικό δέντρο αναζήτησης, διότι λόγω των περιστροφών δεν μπορούμε να

εγγυηθούμε πως θα ισχύουν πάντοτε. Εξαιτίας αυτού προσπαθήσαμε να βρούμε νέες συνθήκες αναστρεψιμότητας, οι οποίες δεν θα απαιτούν επιπλέον πληροφορία. Η προσπάθεια μας αυτή δυστυχώς ήταν ανεπιτυχής. Έτσι αποφασίσαμε να βρούμε συνθήκη αναστρεψιμότητας που θα απαιτούσε την ελάχιστη δυνατή επιπλέον πληροφορία. Για την επίτευξη του στόχου αυτού έγινε χρήση μίας μεταβλητής ως σημαίας για να γνωρίζουμε ποια περίπτωση εκτελέστηκε, με τον ίδιο τρόπο που εφαρμόστηκε στις αναπροσαρμογές. Και εδώ η χρήση μίας μεταβλητής μπορεί να θεωρηθεί αμελητέα αλλά εξαιτίας της αναδρομικής εκτέλεσης του αλγορίθμου, όπως θα εξηγήσουμε αναλυτικά αργότερα, η χωρική πολυπλοκότητα αυξάνεται αισθητά.

Αν η θέση του νέου κόμβου, δηλαδή ο κόμβος μας ισούται με τη τρέχουσα σημαίνει πως βρισκόμαστε στη ρίζα και δεν χρειάζεται να κάνουμε αναζήτηση. Σε αντίθετη περίπτωση κάνουμε αναζήτηση και ενώνουμε το νέο κόμβο με το πατέρα του. Έπειτα ενημερώνουμε αναδρομικά τα ύψη και ελέγχουμε αν χρειάζονται περιστροφές για να διατηρήσει το δέντρο μας την ιδιότητα του.

Λαμβάνοντας αυτά υπόψη, υλοποιήσαμε το αλγόριθμο με τη εξής μορφή:

1. Δημιούργησε το νέο κόμβο θέτοντας ύψος και τιμή.
2. Θέσε τη σημαία ίση με 0.
3. Αν η θέση του νέου κόμβου είναι διαφορετική από τη τρέχουσα
4. Αν η τιμή του νέου κόμβου είναι μικρότερη του τρέχοντος κόμβου
5. Αν το αριστερό παιδί του κόμβου είναι κενό
6. Ένωσε το νέο κόμβο με το τρέχον κόμβο (πατέρα)
7. Αύξησε τη σημαία κατά 1
8. Αλλιώς
9. Συνέχισε αναδρομικά από το βήμα 2 στο αριστερό υποδένδρο
10. Αύξησε τη σημαία κατά 2
11. **Συνθήκη αναστρεψιμότητας:** Η σημαία ισούται με 1
12. Αλλιώς
13. Αν το δεξιό παιδί του κόμβου είναι κενό
14. Ένωσε το νέο κόμβο με το τρέχον κόμβο (πατέρα)
15. Αύξησε τη σημαία κατά 3
16. Αλλιώς
17. Συνέχισε αναδρομικά από το βήμα 2 στο δεξιό υποδένδρο
18. Αύξησε τη σημαία κατά 4
19. **Συνθήκη αναστρεψιμότητας:** Η σημαία ισούται με 3
20. **Συνθήκη αναστρεψιμότητας:** Η σημαία ισούται με 1 ή 2
21. **Συνθήκη αναστρεψιμότητας:** Η σημαία δεν ισούται 0

22. Ενημέρωση ύψους του τρέχοντος κόμβου

23. Αναπροσαρμογή δέντρου με ρίζα το τρέχον κόμβο

Ακολουθεί ο ψευδοκώδικας της διαδικασίας:

```
procedure insert(node root, int value, node newnode, stack rotate, stack
height, stack traverse)
1.  newnode.value += value
    newnodePos.height += 1
    call insertHelper(value, root, newnode, rotate, height, traverse)
procedure insertHelper(int value, node currentNode, node newnode, stack
rotate, stack height, stack traverse)
2.  local int flag = 0
3.  if newnode != currentNode then
4.    if value < currentNode.value then
5.          if currentNode.left = null then
6.                currentNode.left += newnodePos
                  newnodePos.parent += currentNode
7.                flag += 1
8.          else
                  local node a = currentNode.left
                  currentNode.left -= a
9.                call insertHelper(value,a,newnode,rotate,height,traverse)
                  currentNode.left += a
                  delocal node a = currentNode.left
10.               flag += 2
11.         fi flag = 1
12.   else
13.         if currentNode.right = null then
                  currentNode.right += newnode
                  newnodePos.parent += currentNode
14.               flag += 3
15.         else
                  local node a = currentNode.right
                  currentNode.right -= a
16.               call insertHelper(value,a,newnode,rotate,height,traverse)
                  currentNode.right += a
                  delocal node a = currentNode.right
17.               flag += 4
18.         fi flag = 3
19.   fi flag = 1 || flag = 2
20. fi flag != 0
21. push(flag, traverse)
    delocal int flag = 0
22. call updateHeight(currentNode, height)
23. call balance(value, currentNode, rotate, height)
```

Χρονική και Χωρική Πολυπλοκότητα

Γνωρίζουμε από τον ορισμό των AVL δέντρων πως η εισαγωγή έχει χρονική πολυπλοκότητα $O(\log n)$. Ο αλγόριθμος που προτείνουμε στηρίζετε στην ίδια λογική και προφανώς διατηρεί τη χρονική πολυπλοκότητα. Αντιθέτως όμως αυξάνει τη χωρική. Χρονική πολυπλοκότητα $O(\log n)$ σημαίνει πως στη χειρότερη περίπτωση η αναδρομή μας εκτελείται $\log n$ φορές. Επομένως δημιουργούνται και χρειάζεται να φυλαχτούν $\log n$ μεταβλητές-σημαίες. Επιπρόσθετα αυτό σημαίνει πως οι συναρτήσεις αναπροσαρμογής και ενημέρωσης θα εκτελεστούν επίσης $\log n$ φορές. Γνωρίζουμε πως η ενημέρωση υψών φυλάει μία επιπλέον

πληροφορία ενώ η αναπροσαρμογή στη χειρότερη περίπτωση πέντε. Επομένως έχουμε τρία είδη επιπλέον πληροφοριών:

- $\log n$ για τις μεταβλητές σημαίες που βοηθούν στην διάσχιση του δέντρου
- $\log n$ για την ενημέρωση των κόμβων που επισκεφθήκαμε
- $5\log n$ για τις αναπροσαρμογές

Επομένως στη χειρότερη περίπτωση έχουμε $7\log n$ επιπρόσθετη πληροφορία, μέγεθος σχετικά μεγάλο. Όπως προαναφέραμε λόγω των περιστροφών, υπάρχουν αρκετοί συνδυασμοί και είναι δύσκολη η ενοποίηση τους σε μία συνθήκη χωρίς να φυλάμε επιπρόσθετη πληροφορία για τη διάσχιση που γίνεται, δηλαδή αν θα συνεχίσουμε στο δεξιό ή αριστερό υποδένδρο και πως θα ενώσουμε το νέο κόμβο στο δέντρο μας. Λόγω αυτού θεωρούμε πως η πληροφορία αυτή είναι απαραίτητη. Σχετικά με τα υπόλοιπα $6\log n$ γνωρίζουμε πως τα $4\log n$ των αναπροσαρμογών οφείλονται στο ότι στη χείριστη περίπτωση γίνεται διπλή περιστροφή $\log n$ φορές και επομένως καλείται $4\log n$ φορές η συνάρτηση ενημέρωσης ύψους. Επίσης έχουμε $\log n$ σκουπίδια λόγω της ενημέρωσης ύψους που γίνεται αναδρομικά στο δέντρο μας. Λαμβάνοντας αυτά υπόψη αν επιτευχθεί αναστρέψιμη μορφή της ενημέρωσης ύψους η οποία δεν θα απαιτεί επιπρόσθετη πληροφορία τότε η χωρική πολυπλοκότητα θα μειωθεί αισθητά στα $2\log n$ όμως και πάλι θα παραμείνει $O(\log n)$.

Συνοψίζοντας, η χωρική πολυπλοκότητα αυξάνεται κατά $O(\log n)$ με αποτέλεσμα ο προτεινόμενος αναστρέψιμος αλγόριθμος να είναι πιστός αλλά όχι υγιεινός. Για καλύτερη κατανόηση και έλεγχο των επιπλέον πληροφοριών προτιμήθηκε η χρήση ξεχωριστών στοιβών αντί μίας κοινής. Επιπλέον έγινε χρήση στοιβών και όχι πινάκων μεγέθους $\log n$ καθώς θα σπαταλούσαμε θέσεις μνήμης αχρείαστα όταν εκτελείτο η χείριστη περίπτωση.

4.7 Αναπροσαρμογή δέντρου κατά την διαγραφή

Αυτή η συνάρτηση είναι υπεύθυνη για την επιλογή της περιστροφής που θα εκτελεστεί αν κριθεί αναγκαίο κατά τη διαγραφή. Γνωρίζουμε πως έχουμε τέσσερα είδη περιστροφών και ότι η διαφορά του ύψους ανάμεσα σε δύο υποδένδρα πρέπει να είναι το πολύ ένα. Επομένως έχουμε τέσσερεις συνθήκες οι οποίες θα καθορίσουν ποια περιστροφή θα εκτελεστεί αν χρειάζεται:

1. Αν το αριστερό υποδένδρο είναι κατά τουλάχιστον 2 ψηλότερο από το δεξιό και το αριστερό υποδένδρο του αριστερού μας παιδιού είναι ψηλότερο ή ίσο με το δεξιό υποδένδρο του αριστερού μας παιδιού: Δεξιά περιστροφή στον τρέχοντα κόμβο.

2. Αν το αριστερό υποδένδρο είναι κατά τουλάχιστον 2 ψηλότερο από το δεξιό και το αριστερό υποδένδρο του αριστερού μας παιδιού είναι χαμηλότερο από το δεξιό υποδένδρο του αριστερού μας παιδιού: Αριστερή περιστροφή στο αριστερό μας παιδί, Δεξιά περιστροφή στον τρέχοντα κόμβο.
3. Αν το αριστερό υποδένδρο είναι κατά τουλάχιστον 2 χαμηλότερο από το δεξιό και το αριστερό υποδένδρο του δεξιού μας παιδιού είναι χαμηλότερο ή ίσο με το δεξιό υποδένδρο του δεξιού μας παιδιού: Αριστερή περιστροφή στον τρέχοντα κόμβο
4. Αν το αριστερό υποδένδρο είναι κατά τουλάχιστον 2 χαμηλότερο και το αριστερό υποδένδρο του δεξιού μας παιδιού είναι ψηλότερο από το δεξιό υποδένδρο του δεξιού μας παιδιού: Δεξιά περιστροφή στο δεξιό μας παιδί, Αριστερή περιστροφή στο τρέχοντα κόμβο.

Η συνάρτηση αποτελείται από συγκρίσεις, οι οποίες κάνουν χρήση της συνάρτησης περιστροφών. Άρα η χρονική της πολυπλοκότητα εξαρτάται από αυτή των περιστροφών. Οι περιστροφές έχουν πολυπλοκότητα O(1). Άρα και η αναπροσαρμογή του δέντρου είναι O(1).

Αναστρέψιμη μορφή

Ακολουθήσαμε την ίδια λογική με την αναστρέψιμη αναπροσαρμογή κατά την εισαγωγή. Γίνεται χρήση βοηθητικής μεταβλητής-σημαίας η οποία χρησιμοποιείται στις συνθήκες αναστρεψιμότητας ώστε να επαληθεύονται και να γνωρίζουμε κατά την αντίστροφη εκτέλεση ποια πρέπει να εκτελεστεί, όμοια με την αναπροσαρμογή κατά την εισαγωγή.

Συνοπτικά ο αλγόριθμος μας με βάση τις περιπτώσεις που εξηγήσαμε στο προηγούμενο υποκεφάλαιο έχει ως εξής:
Θέσε τη σημαία ίση με μηδέν
1. Αν Περίπτωση 1, εκτέλεσε την και πρόσθεσε 1 στη σημαία
2. Αλλιώς αν Περίπτωση 2, εκτέλεσε την και πρόσθεσε 2 στη σημαία
3. Αλλιώς αν Περίπτωση 3, εκτέλεσε την και πρόσθεσε 3 στη σημαία
4. Αλλιώς αν Περίπτωση 4, εκτέλεσε την και πρόσθεσε 4 στη σημαία
5. **Συνθήκη αναστρεψιμότητας:** Η σημαία ισούται με 4
6. **Συνθήκη αναστρεψιμότητας:** Η σημαία ισούται με 3
7. **Συνθήκη αναστρεψιμότητας:** Η σημαία ισούται με 2
8. **Συνθήκη αναστρεψιμότητας:** Η σημαία ισούται με 1
9. Φύλαξε τη σημαία στη βοηθητική στοίβα

Ακολουθεί ο ψευδοκώδικας της διαδικασίας:

```
procedure balanceRemove(node currentNode, stack rotate, stack height)
      local int flag = 0
1.    if currentNode.left.height - currentNode.right.height > 1 &&
currentNode.left.left.height - currentNode.left.right.height >= 0 then
            call rightRotate(currentNode, height)
            flag += 1
      else
2.          if currentNode.left.height - currentNode.right.height > 1 &&
currentNode.left.left.height - currentNode.left.right.height < 0 then
                  local node left = currentNode.left
                  currentNode.left -= left
                  call leftRotate(left, height)
                  currentNode.left += left
                  delocal node left = currentNode.left

                  call rightRotate(currentNode, height)
                  flag += 2
            else
3.                if currentNode.left.height - currentNode.right.height < -1
&& currentNode.right.left.height - currentNode.right.right.height <= 0 then
                        call leftRotate(currentNode, height)
                        flag += 3
                  else
4.                      if currentNode.left.height - currentNode.right.height
< -1 && currentNode.right,left.height-currentNode.right.right.height > 0 then
                              local node right = currentNode.right
                              currentNode.right -= right
                              call rightRotate(right, height)
                              currentNode.right += right
                              delocal node right = currentNode.right

                              call leftRotate(currentNode, height)
                              flag += 4
5.                      fi flag = 4
6.                fi flag = 3
7.          fi flag = 2
8.    fi flag = 1
9.    push(flag, rotate)
      delocal int flag = 0
```

Χρονική και Χωρική Πολυπλοκότητα

Όμοια με την αναπροσαρμογή κατά την εισαγωγή ο αλγόριθμος μας έχει χρονική πολυπλοκότητα O(1) και λόγω της αύξησης της χωρικής πολυπλοκότητας που σημειώνεται η λύση μας είναι υγιεινή, καθώς η φύλαξη το πολύ πέντε επιπλέον πληροφοριών μπορεί να θεωρηθεί αμελητέα, O(1). Οι πέντε αυτές πληροφορίες οφείλονται στη μεταβλητή σημαία που χρησιμοποιείται και στο ότι κατά την εκτέλεση διπλής περιστροφής γίνεται κλήση της ενημέρωσης ύψους τέσσερις φορές. Και εδώ οι μεταβλητές φυλάσσονται σε στοίβα ώστε να περιορίζουμε τη δημιουργία και επεξεργασία των σκουπιδιών εντός της συνάρτησης.

4.8 Διαγραφή

Ακολουθεί την ίδια λογική με την εισαγωγή, δηλαδή στηρίζεται στην διαδικασία διαγραφής των δυαδικών δέντρων αναζήτησης, την ενημέρωση των υψών και την αναπροσαρμογή των κόμβων η οποία εκτελεί τις περιστροφές υπό διαφορετικές συνθήκες σε σχέση με την

εισαγωγή. Έπειτα από τη διαγραφή επιστρέφουμε αναδρομικά στη ρίζα ενημερώνοντας τα ύψη των επηρεαζόμενων κόμβων και ελέγχουμε για αναπροσαρμογές.

Συνοπτικά ο αλγόριθμος διαγραφής στα AVL δέντρα έχει ως εξής:

1. Εκτέλεσε κανονική διαγραφή Δυαδικού Δέντρου Αναζήτησης.
2. Ο τρέχον κόμβος πρέπει να είναι ένας από τους προγόνους του πρόσφατα διαγραφέντα κόμβου. Ενημέρωσε το ύψος του τρέχοντος κόμβου.
3. Εκτέλεσε τη συνάρτηση αναπροσαρμογής.
4. Επανέλαβε αναδρομικά τα βήματα 2 και 3 για τους κόμβους που επισκέφθηκες στο βήμα 1 μέχρι να επιστρέψεις στη ρίζα.

Αναστρέψιμη μορφή

Καταρχάς ενημερώσαμε τις αναστρέψιμες διαδικασίες διαγραφής των δυαδικών δέντρων αναζήτησης έτσι ώστε να διαγράφουν και το ύψος των επηρεαζόμενων κόμβων. Σε αυτό το σημείο να αναφέρουμε πως χρειάστηκε να φυλάμε τα «σκουπίδια» που παράγει η διαγραφή των δυαδικών δέντρων αναζήτησης.

Όμοια με την εισαγωγή χρειαστήκαμε επιπρόσθετες πληροφορίες για την ενημέρωση των υψών των κόμβων και για τον έλεγχο αναπροσαρμογών. Όμοια κρίθηκε αναγκαία η φύλαξη επιπρόσθετης πληροφορίας για τη διάσχιση που γίνεται, δηλαδή αν θα συνεχίσουμε στο δεξιό ή αριστερό υποδένδρο, καθώς λόγω των περιστροφών, δεν κατορθώσαμε να ορίσουμε συνθήκη αναστρεψιμότητας χωρίς να φυλάμε επιπρόσθετη πληροφορία.

Η μεγαλύτερη δυσκολία που αντιμετωπίσαμε ήταν ότι δεν μπορούσαμε να χρησιμοποιήσουμε την τεχνική που κάναμε μέχρι τώρα για να αντιμετωπίσουμε το πρόβλημα κλήσης συνάρτησης-αναδρομής όπου μία μεταβλητή πίνακα μπορεί να αλλάξει τιμή. Σύμφωνα με τη Janus, το μόνο είδος παραμέτρων που μπορούν να αλλάξουν τιμή κατά την εκτέλεση μίας συνάρτησης, αναδρομικής ή μη, είναι οι ακέραιοι. Επομένως από τη στιγμή που αντιστοιχούμε τους κόμβους με θέσεις πίνακα έπρεπε να σκεφτούμε κάποιο τρόπο έμμεσης κλήσης της συνάρτησης. Μία απλή τεχνική όπως προαναφέραμε είναι να φυλάμε τη θέση του πίνακα σε κάποια μεταβλητή και να εκτελούμε τη συνάρτηση χρησιμοποιώντας ως θέση του πίνακα τη τιμή της μεταβλητή αυτή.

```
local node left = currentNode.left
currentNode.left -= left
call leftRotate(left, height)
currentNode.left += left
delocal node left = currentNode.left
```

Αυτό όμως είχε το πρόβλημα ότι δεν θα μπορούσαμε να διαγράψουμε τη τοπική μεταβλητή. Για το λόγο αυτό δημιουργούσαμε μία μεταβλητή και μηδενίζαμε μέσω τη θέση-δείκτη του πίνακα ώστε να «αποσυνδέουμε» το τρέχον κόμβο από το υποδένδρο. Με το τρόπο αυτό είχαμε τη δυνατότητα να κάνουμε οποιαδήποτε αλλαγή επιθυμούσαμε στο αντίστοιχο υποδένδρο. Κατά την αναδρομική επιστροφή ή έξοδο από τη συνάρτηση επανασυνδέαμε το τρέχον κόμβο με το υποδένδρο και είμασταν σε θέση να γνωρίζουμε τη τιμή της βοηθητικής μεταβλητής ώστε να τη διαγράψουμε. Αυτό είμαστε σε θέση να το κάνουμε διότι οι αλλαγές δεν επηρεάζουν το τρέχον κόμβο αλλά το υποδένδρο. Ο τρέχον κόμβος αρκείται απλά στο να ενωθεί στο κόμβο που αντιστοιχεί η μεταβλητή, στο πιο πάνω παράδειγμα να ενωθεί με τη νέα ρίζα του υποδένδρου.

Δυστυχώς όμως κατά τη διαγραφή η πιο πάνω τεχνική δεν δουλεύει ορθά σε κάθε περίπτωση. Συγκεκριμένα κατά τη διαγραφή κόμβου με μόνο ένα παιδί γίνεται ενημέρωση και στο κόμβο πατέρα ώστε να μην ενώνεται με το διαγραφέντα κόμβο-παιδί του αλλά με το εγγόνι του. Επομένως αν ερχόμασταν αναδρομικά να «επανασυνδέσουμε» τους επηρεαζόμενους κόμβους θα καταλήγαμε να δείχνουμε σε λάθος κόμβους, καθώς ο δείκτης του τρέχοντος κόμβου δεν θα ισούταν με τη μεταβλητή μας. Για παράδειγμα έστω το εγγόνι βρίσκεται στη θέση 2. Κατά τη διαγραφή θα ενημερώναμε το κόμβο πατέρα του διαγραφέντα κόμβου ώστε να δείχνει στο παιδί, δηλαδή στη θέση 2. Έτσι όταν θα ερχόμασταν αναδρομικά να ενημερώσουμε τη ένωση του κόμβου με το υποδένδρο θα κάναμε 2+2=4, δηλαδή θα ενωνόμασταν με λάθος κόμβο. Εκτός αυτού δεν θα γνωρίζαμε τη τιμή της τοπικής μεταβλητής για να μπορούμε να τη διαγράψουμε.

Έτσι κρίθηκε αναγκαίο να φυλάμε τη θέση των κόμβων που επισκεφθήκαμε ώστε να είμαστε σε θέση να μηδενίζουμε τη τοπική μεταβλητή κατά την αναδρομική επαναφορά. Κατά τη γενικευμένη μορφή του αλγορίθμου όμως, δεν θα είχαμε το περιορισμό αυτό κατά τη κλήση αναδρομικών συναρτήσεων και η πληροφορία αυτή θα μπορούσε να παραλείπεται.

Επίσης τροποποιήθηκε η κάθε περίπτωση διαγραφής ώστε να μηδενίζεται και το ύψος του κάθε κόμβου. Η υλοποίηση αυτή είναι πολύ απλή και παραλείπεται η επεξήγησή της. Επιπλέον η επιλογή της περίπτωσης διαγραφής που θα εκτελεστεί είναι πανομοιότυπη με την αντίστοιχη στα δυαδικά δέντρα αναζήτησης. Η υλοποίηση των βοηθητικών συναρτήσεων αυτών, μπορεί να βρεθεί στο Παράρτημα Β μαζί με όλες τις άλλες συναρτήσεις για τα AVL δέντρα.

Συνοπτικά η ανάστροφη διαγραφή που προτείνουμε εκτελεί τα ακόλουθα βήματα:

1. Αν η ρίζα δεν είναι κενή
2. Θέσε τη σημαία ίση με 0
3. Αν η τιμή που διαγράφουμε ισούται με την τιμή του τρέχοντος κόμβου
4. Εκτέλεση διαγραφή με βάση τις ενημερωμένες εκδόσεις των περιπτώσεων διαγραφής
5. Αλλιώς
6. Αν η τιμή που ψάχνουμε για διαγραφή είναι μικρότερη από την τιμή του κόμβου
7. Θέσε ως x το αριστερό παιδί
8. Κάλεσε αναδρομικά τη διαδικασία από το x
9. Αναίρεσε το x και θέσε το y ίσο με 1
10. Αλλιώς
11. Θέσε ως x το δεξιό παιδί
12. Κάλεσε αναδρομικά τη διαδικασία από το x
13. Αναίρεσε το χ και θέσε το y ίσο με 2
14. **Συνθήκη αναστρεψιμότητας:** Το y ισούται με 1
15. **Συνθήκη αναστρεψιμότητας:** Το y ισούται με 0
16. Φύλαξε τη σημαία στη βοηθητική στοίβα
17. Η ρίζα δεν είναι κενή
18. Ενημέρωση το ύψος του τρέχοντος κόμβου
19. Έλεγξε για αναπροσαρμογή

Ακολουθεί ο ψευδοκώδικας της διαδικασίας:

```
procedure remove(int valueOut, node root, stack traverse,
stack remove_details, stack rotate, stack height)
1.  if root != null then
2.    local int flag = 0
3.    if valueOut = root.value then
4.       call removeCase(valueOut, root, remove_details)
5.    else
6.       if valueOut < root.value then
7.          local node x = root.left
8.          call remove(valueOut,x,traverse,remove_details,rotate,height)
9.          delocal node x = root.left
            flag += 1
10.      else
11.         local node x = root.right
12.         call remove(valueOut,x,traverse,remove_details,rotate,height)
13.         delocal node x = root. right
            flag += 2
14.      fi flag = 1
15.     fi flag = 0
16.  push(flag, traverse)
     delocal int flag = 0
17.  fi root != null

18. call updateHeight(root, height)
19. call balanceRemove(root, rotate, height)
```

Σε αυτό το σημείο να αναφέρουμε πως στη Janus φυλάγαμε τη τοπική μεταβλητή x πριν την αναιρέσουμε για τους λόγους που αναφέραμε πιο πάνω.

Χρονική και Χωρική Πολυπλοκότητα

Ο αλγόριθμος μας δεν κάνει χρήση επιπλέον χρονοβόρων διαδικασιών με αποτέλεσμα να διατηρεί το χρονική πολυπλοκότητα της μη αναστρέψιμης μορφής, δηλαδή $O(\log n)$. Επίσης η λύση που προτείνουμε φυλάει την ίδια ποσότητα επιπρόσθετης πληροφορίας με την αναστρέψιμη εισαγωγή, $7\log n$, που προτείναμε νωρίτερα συν τις επιπρόσθετες πληροφορίες, το πολύ πέντε, που παράγονται κατά τη διαγραφή. Ακόμη παράγονται $\log n$ για τη φύλαξη των κόμβων που επισκεφθήκαμε κατά την διάσχιση του δέντρου, καθώς στη χείριστη περίπτωση θα επισκεφθούμε $\log n$ κόμβους. Για ευκολότερη κατανόηση οι βοηθητικές αυτές πληροφορίες φυλάσσονται σε διαφορετικές στοίβες, όμως θεωρητικά είναι η δυνατή και η χρήση μίας μόνο στοίβας που θα ομαδοποιεί τις πληροφορίες αυτές, καθώς θα ελέγχονται με την ίδια σειρά. Άρα τα συνολικά μας σκουπίδια στη χείριστη περίπτωση είναι $8\log n$.

Επομένως ο αλγόριθμος μας είναι πιστός αλλά όχι υγιεινός καθώς διατηρεί τη χρονική αλλά αυξάνει τη χωρική πολυπλοκότητα κατά $O(\log n)$.

Όπως προαναφέραμε τα $7\log n$ που παράγονται ακολουθούν την λογική που εφαρμόσαμε κατά την εισαγωγή. Επομένως για τους ίδιους λόγους είναι και κατά τη διαγραφή η δημιουργία τους. Σχετικά με τη $\log n$ όπου φυλάμε η πληροφορία-θέση των κόμβων κρίθηκαν αναγκαία καθώς κατά τη διαγραφή κόμβου με ένα παιδί γίνεται ενημέρωση στο κόμβο πατέρα ώστε να μην ενώνεται με το διαγραφέντα κόμβο-παιδί του αλλά με το εγγόνι του. Έτσι όπως εξηγήσαμε χάνεται πληροφορία με αποτέλεσμα κατά την αναδρομική επιστροφή να παίρνουμε μήνυμα λάθους στη γλώσσα προγραμματισμού Janus. Στη γενικευμένη μορφή τα $\log n$ αυτά σκουπίδια μπορούν να αποφευχθούν καθώς δεν θα χρειάζεται να αντιμετωπίσουμε το περιορισμό αυτό.

Κεφάλαιο 5

Splay Δέντρο

5.1 Γνωριμία με τη δομή

Τα Splay δέντρα εφευρέθηκαν από τους Daniel Sleator και Robert Tarján το 1985 [2] και αποτελούν ακόμη ένα είδος αυτορρυθμιζόμενου δυαδικού δέντρου αναζήτησης με την επιπλέον ιδιότητα πως οι κόμβοι που έχουμε πρόσφατα επισκεφθεί είναι γρήγορα προσβάσιμοι. Υποστηρίζει όλες τις βασικές λειτουργίες όπως αναζήτηση, εισαγωγή και διαγραφή. Οι λειτουργίες αυτές έχουν $O(\log n)$ αποσβεστική (amortized) χρονική πολυπλοκότητα. Για πολλές ακολουθίες μη τυχαίων διαδικασιών, τα Splay δέντρα αποδίδουν καλύτερα από ό,τι άλλα δέντρα αναζήτησης, ακόμα και όταν το συγκεκριμένο μοτίβο της ακολουθίας των κόμβων είναι άγνωστο.

Όλες οι κανονικές λειτουργίες ενός δυαδικού δέντρου αναζήτησης συνδυάζονται με μια νέα βασική λειτουργία, που ονομάζεται splaying. Κάνοντας splaying το δέντρο για ένα συγκεκριμένο στοιχείο, το δέντρο μας αναδιατάσσεται έτσι ώστε το στοιχείο αυτό να γίνεται η ρίζα του δένδρου. Ένας τρόπος για να γίνει αυτό είναι να εκτελέσουμε μία απλή αναζήτηση δυαδικού δέντρου αναζητήσεως για το εν λόγω στοιχείο, και στη συνέχεια να χρησιμοποιήσουμε περιστροφές με ένα συγκεκριμένο τρόπο για να μετακινηθεί το στοιχείο στην ρίζα.

Η καλή απόδοση των Splay δέντρων εξαρτάται από το γεγονός ότι είναι αυτοβελτιστοποιημένα, καθώς οι πρόσφατα επισκεφθέντες κόμβοι θα μετακινηθούν πιο κοντά

στη ρίζα, όπου θα μπορούν να προσεγγιστούν πιο γρήγορα. Στη χειρότερη περίπτωση το ύψος, αν και απίθανο, είναι O(n), με τη αποσβεστική (amortized) πολυπλοκότητα να είναι O($\log n$). Έχοντας τους κόμβους που χρησιμοποιούνται συχνά κοντά στη ρίζα αποκτούμε αρκετά πλεονεκτήματα για πολλές πρακτικές εφαρμογές (Τοπικότητα της αναφοράς), και είναι ιδιαίτερα χρήσιμο για την υλοποίηση αλγορίθμων συλλογής απορριμμάτων (garbage collection) και κρυφή μνήμη (caches).

Όπως όλες οι δομές έτσι και τα Splay έχουν αρκετά πλεονεκτήματα και μειονεκτήματα.
Πλεονεκτήματα:

- Συγκρίσιμη απόδοση: Η απόδοση μέσης περίπτωσης είναι το ίδιο αποτελεσματική με τα άλλα δέντρα [3].
- Εύκολος και γρήγορος εντοπισμός συχνά προσβάσιμων στοιχείων.
- Απλούστερα από τα Red-Black δέντρα καθώς δεν χρειάζεται πληροφορία σε κάθε κόμβο για να καθορίζουμε το χρώμα του.
- Μικρό μέγεθος μνήμης: Τα Splay δέντρα δεν χρειάζεται να αποθηκεύουν όλα τα δεδομένα υπολογισμών.

Το πιο σημαντικό μειονέκτημα τους είναι ότι το ύψος ενός δέντρου μπορεί να είναι γραμμικό, όταν εισάγουμε όλα τα στοιχεία *n* σε αύξουσα ή φθίνουσα σειρά. Δεδομένου ότι στη χειρότερη περίπτωση το ύψος ενός δέντρου αντιστοιχεί στο χρόνο πρόσβασης, αυτό σημαίνει ότι το πραγματικό κόστος μιας διαδικασίας μπορεί να είναι υψηλό. Ωστόσο, το αποσβεστικό (amortized) κόστος πρόσβασης της παρούσας χειρότερη περίπτωση είναι λογαριθμικό, O($\log n$).

Επίσης, το αναμενόμενο κόστος πρόσβασης μπορεί να μειωθεί σε O($\log n$) χρησιμοποιώντας μια τυχαιοποιημένη παραλλαγή [8]. Τα αποτελέσματα δείχνουν ότι κάνοντας ημι-splaying, το οποίο προτείνεται στο ίδιο άρθρο με το splaying, αποδίδει καλύτερα από ό,τι splaying κάτω από σχεδόν όλες τις δυνατές συνθήκες. Το γεγονός αυτό καθιστά το ημι-splaying μια καλή εναλλακτική λύση για όλες τις εφαρμογές όπου είναι χρήσιμη η χρήση splaying. Ο λόγος για τον οποίο το splaying έγινε τόσο διάσημο, ενώ το ημι-splaying είναι σχετικά άγνωστο και πολύ λιγότερο μελετημένο είναι ότι είναι δύσκολο να κατανοηθεί.

Εμείς θα μελετήσουμε το splaying καθώς είναι πιο διαδεδομένο άλλα στο μέλλον θα ήταν χρήσιμο να γίνει και η ημί-splaying μορφή.
Η εμφάνιση των δέντρων μπορεί να αλλάξει ακόμα και αν γίνεται πρόσβαση με ένα τρόπο «μόνο για ανάγνωση», όπως για παράδειγμα σε διαδικασίες αναζήτησης. Αυτό περιπλέκει τη

χρήση των δέντρων αυτών σε πολυνηματικό περιβάλλον. Συγκεκριμένα, επιπρόσθετη διαχείριση είναι απαραίτητη εάν υπάρχουν πολλά νήματα τα οποία έχουν το δικαίωμα να εκτελέσουν εργασίες εύρεσης ταυτόχρονα. Αυτό τα καθιστά ακατάλληλα για γενική χρήση σε καθαρά λειτουργικό προγραμματισμό, παρόλο που μπορούν να χρησιμοποιηθούν με περιορισμένους τρόπους για την υλοποίηση ουρών προτεραιότητας.

Η δομή αυτή έχει καταφέρει να γίνει η πιο ευρέως χρησιμοποιούμενη βασική δομή δεδομένων που εφευρέθηκε τα τελευταία 30 χρόνια, επειδή αποτελούν τον ταχύτερο τύπο ισόρροπης δέντρων αναζήτησης για πολλές εφαρμογές [10].

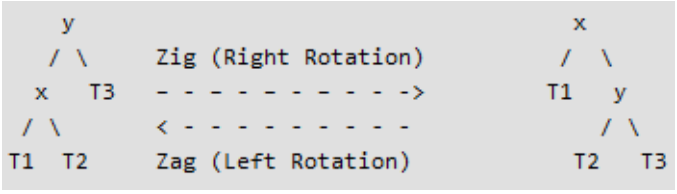
Χρησιμοποιείται σε :

- *Windows NT*: Εικονική μνήμη, δικτύωση και κώδικα του συστήματος αρχείων
- Μεταγλωττιστής *gcc* και βιβλιοθήκη *GNU C++*
- Συντάκτης γραμματοσειρών *sed*
- *Fore Systems* δρομολογητές δικτύου
- Εντολή *malloc*: η πιο δημοφιλής εφαρμογή του *Unix*
- *Linux* ***loadable kernel module (LKM)***

Όταν γίνεται πρόσβαση σε κάποιο κόμβο *x*, εκτελείται μία λειτουργία splay ώστε το *x* να μετακινηθεί προς τη ρίζα. Για να εκτελεστεί μια λειτουργία πραγματοποιείται μια αλληλουχία σταδίων splay, καθένα από τα οποία μετακινεί το κόμβο *x* πιο κοντά στη ρίζα. Με την ολοκλήρωση της εκτέλεσής της, οι πρόσφατα προσβάσιμοι κόμβοι βρίσκονται κοντά στη ρίζα και το δέντρο παραμένει σχεδόν ισορροπημένο, έτσι ώστε να επιτευχθούν τα επιθυμητά όρια αποσβεστικού χρόνου.

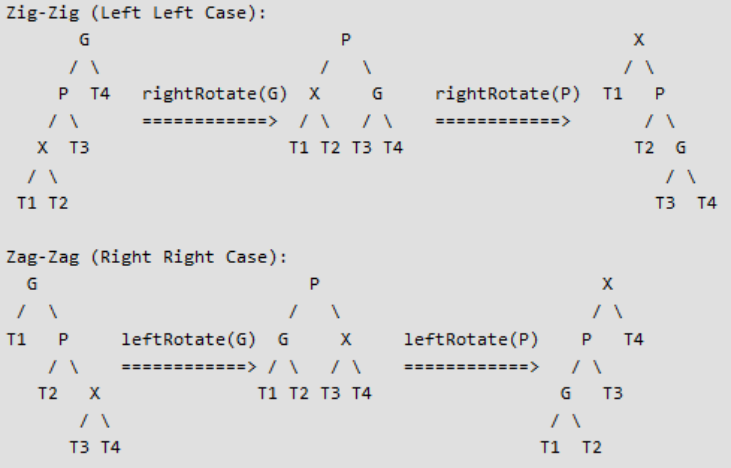
Η περιστροφή που θα εκτελεστεί σε κάθε στάδιο, εξαρτάται από τέσσερεις παράγοντες:

- Αν ο κόμβος είναι ρίζα, καμία αλλαγή.
- Αν ο κόμβος είναι αριστερό ή το δεξί παιδί της ρίζας, δηλαδή δεν έχει παππού. Σε αυτή την περίπτωση μπορεί να εκτελεστεί είτε Zag είτε Zig, οι οποίες είναι οι αντίστοιχες Αριστερά και Δεξιά περιστροφές του AVL δέντρου.



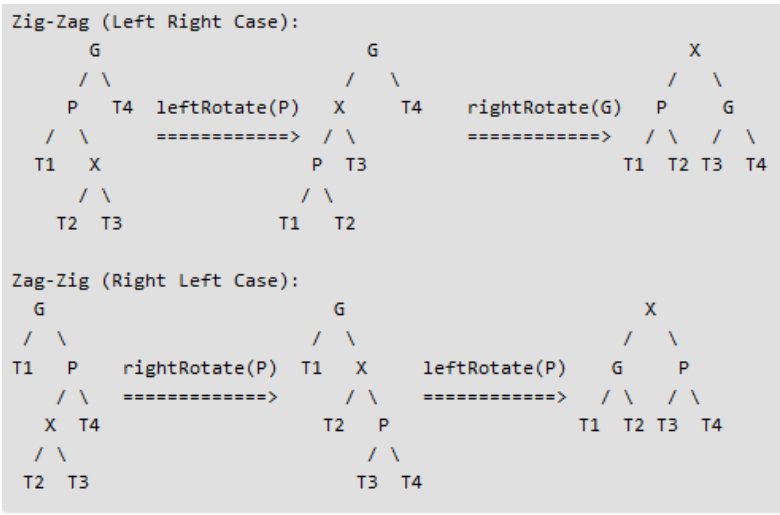
Σχήμα 10: Zig και Zag περιστροφές

- εάν ο κόμβος και ο πατέρας του είναι είτε αριστερά είτε δεξιά παιδιά, έχουμε Zag-Zag ή Zig-Zig περιστροφές,



Σχήμα 11: Zig- Zig και Zag- Zag περιστροφές

- εάν ο κόμβος είναι αριστερό παιδί και ο πατέρας δεξιό Zig-Zag ή αν ο κόμβος είναι δεξιό παιδί και ο πατέρας αριστερό Zag- Zig



Σχήμα 12: Zig-Zag και Zag- Zig περιστροφές

Για την υλοποίηση της δομής αυτής στηριχτήκαμε στις θεμελιώδεις συναρτήσεις που προτείνονται στο βιβλίο Algorithms [18], το οποίο είναι διαθέσιμο σε ηλεκτρονική μορφή [19]. Ακόμη προτείνεται μία υλοποίηση των δέντρων splay στη γλώσσα προγραμματισμού Java [20].

5.2 Αναστρέψιμη δομή και περιορισμοί στη Janus

Στο ιδανικό σενάριο όπου έχουμε υλοποιημένα αναστρέψιμα αντικείμενα και δείκτες, η δομή του δέντρου μας θα παραμένει η ίδια με αυτή της κανονικής μορφής, όμοια με το δυαδικό

δέντρο αναζήτηση, όπως δείξαμε στο Κεφάλαιο 3. Κάτι λογικό καθώς τα splay δέντρα αποτελούν παραλλαγή των δυαδικών δέντρων αναζήτησης. Όμως η γλώσσα προγραμματισμού Janus ακόμη δεν τα υποστηρίζει έτσι κληθήκαμε να αντιμετωπίσουμε και πάλι το περιορισμό αυτό. Τα Splay δέντρα αποτελούν επέκταση των δυαδικών δέντρων αναζήτησης χωρίς να απαιτούν ο κάθε κόμβος να φυλάει περισσότερα στοιχεία. Επομένως ως αναστρέψιμη δομή μπορούμε να χρησιμοποιήσουμε τη δομή που προτείναμε για τα δυαδικά δέντρα αναζήτησης στο Υποκεφάλαιο 3.2, δηλαδή χρήση δισδιάστατο πίνακα με διαστάσεις n×4 όπου κάθε γραμμή αντιστοιχεί σε κόμβο και κάθε στήλη σε χαρακτηριστικό του:

1. τιμή
2. πατέρας
3. αριστερό παιδί
4. δεξιό παιδί

Ακόμη οι υλοποιήσεις που προτείνουμε θα έχουν μία αφαιρετική περιγραφή ώστε να δείχνουμε την αναστρεψιμότητα τους ανεξαρτήτως γλώσσας προγραμματισμού, παρόλο που η υλοποίηση μας έγινε στη γλώσσα προγραμματισμού Janus, Παράρτημα Γ. Αυτό γίνεται με σκοπό να προτείνουμε αναστρέψιμες δομές οι οποίες δεν θα περιορίζονται από τους περιορισμούς κάποιας γλώσσας. Η υλοποίηση στη Janus χρησιμοποιήθηκε ως μέσω ελέγχου και επαλήθευσης των προτάσεων μας.

Συγκεκριμένα:

- Αν θέλουμε να αναφερθούμε στο δεξιό παιδί κάποιου κόμβου X δεν θα λέμε η θέση [X, 4] του πίνακα αλλά το δεξί παιδί του κόμβου X, δίνοντας έτσι τη μορφή των συναρτήσεων σε περίπτωση που είχαμε στη διάθεση μας δείκτες.
- Αναφορά σε κάποιο κόμβο X θα γίνεται ονομαστικά, υποθέτοντας πως έχουμε αντικείμενα τύπου κόμβου (node) και όχι ως γραμμή πίνακα όπως για τη Janus. Για παράδειγμα θα λέμε ο κόμβος X και όχι η γραμμή X του πίνακα-δέντρου μας.
- Το κενό θα συμβολίζεται με null ενώ για τη Janus θα ελέγχουμε αν ο κόμβος έχει τιμή μηδέν.

5.3 Αναστρέψιμη Περιστροφή

Όπως προαναφέραμε στο κεφάλαιο των AVL περιστροφών εκμεταλλευόμενοι την παρατήρηση πως οι αλλαγές στους δείκτες κατά τη δεξιά και αριστερή περιστροφή αποτελούν η μία αναστροφή της άλλης καταλήξαμε στο συμπέρασμα πως στον αναστρέψιμο προγραμματισμό αρκεί μόνο ένα είδος περιστροφών, στη περίπτωση μας η αριστερή.

Στα Splay δέντρα κάτι τέτοιο είναι εφικτό καθώς δεν χρειάζεται η χρήση της συνάρτησης ενημέρωσης υψών που μας δημιουργούσε προβλήματα στα AVL δέντρα. Στις περιπτώσεις που χρειάζεται δεξιά περιστροφή απλά εκτελούμε αντίστροφα την αριστερή περιστροφή. Όμοια μπορεί να υλοποιηθεί και μόνο η δεξιά περιστροφή, και για αριστερή περιστροφή να εκτελούμε αντίστροφα τη δεξιά, κάνοντας τις αντίστοιχες αλλαγές ώστε να διατηρηθεί η ορθότητα των αλγορίθμων που θα προτείνουμε σε λίγο.

Ακόμη αυτό μπορεί να εφαρμοστεί και στις περιπτώσεις όπου πρέπει να εφαρμόσουμε συνδυασμό αριστερής και δεξιάς περιστροφής. Στα πλαίσια της Ατομικής Διπλωματικής Εργασίας για υλοποίηση των περιπτώσεων που απαιτούν δύο είδη περιστροφών εκτελούμε πρώτα τη μία στο αντίστοιχο παιδί του τρέχον κόμβου και μετά την άλλη στο κόμβο μας. Στην ουσία εκτελούμε την ίδια συνάρτηση προς τα εμπρός στον ένα κόμβο και προς τα πίσω στον άλλο ανάλογα με τη περιστροφή που χρειάζεται και το είδος της περιστροφής (δεξιά ή αριστερά) που επιλέξαμε να χρησιμοποιούμε. Στηριζόμενοι στην ίδια λογική με την αριστερή και δεξιά περιστροφή, θεωρούμε πως υλοποίηση των διπλών περιστροφών οι οποίες κάνουν ταυτόχρονα και αυτόνομα όλες τις απαραίτητες αλλαγές είναι επίσης αναστρέψιμες. Πιο απλά η υλοποίηση αναστρέψιμης μορφής των Αριστερά-Δεξιά και Δεξιά-Αριστερά περιστροφών σε μία συνάρτηση είναι εφικτή.

Η περιστροφή μας έχει ως εξής:

1. Έστω x ο κόμβος μας, y το δεξιό μας παιδί και t το αριστερό παιδί του y
2. Κάνε το x το νέο αριστερό παιδί του y
3. Κάνε το t το νέο δεξιό παιδί του x
4. Αν ο t δεν είναι κενός
    a. Αντάλλαξε το πατέρα του x με το πατέρα του t
    b. Αντάλλαξε το πατέρα του t με το πατέρα του y
5. Αλλιώς
    a. Αντάλλαξε το πατέρα του x με το πατέρα του y
    b. Κάνε το y το νέο πατέρα του x
6. **Συνθήκη αναστρεψιμότητας:** ο t δεν είναι κενός
7. Αναίρεσε τη μεταβλητή t
8. Όρισε ως νέα ρίζα του υποδένδρου το y

Ακολουθεί ο ψευδοκώδικας της διαδικασίας:

```
procedure Rotate(node x)
1.    local node y = x.right, t = y.left
2.    y.left -= t
      y.left += x
3.    x.right -= y
      x.right += t

4.    if t != null then
4.a         x.parent <=> t.parent
4.b         t.parent <=> y.parent
5     else
5.a         x.parent <=> y.parent
            local node a = x
5.b         x.parent -= a
            x.parent += y
            delocal node a = x
6.    fi t != null

7.    delocal node t = x.right

8.    x -= y.left
      x += y
      delocal node y = x
```

Όπως προαναφέραμε, η υλοποίηση αυτή στηρίζεται στη χρήση της αριστερής περιστροφής όμως πανομοιότυπα μπορεί να μετατραπεί και η δεξιά περιστροφή. Με μία απλή σύγκριση της περιστροφής αυτής και της αριστερής περιστροφής στα AVL δέντρα γίνεται εύκολα αντιληπτό πως η μόνη διαφορά είναι η μη χρήση της ενημέρωσης υψών. Σύμφωνα με το βιβλίο που μελετήσαμε [18, 19, 20] ο υπολογισμός του ύψους ενός κόμβου στα Splay δέντρα θα μπορούσε να γίνει με τη χρήση μίας βοηθητικής αναδρομικής συνάρτησης.

Η προσέγγιση αυτή όμως δεν θα ήταν επιθυμητή για τα AVL δέντρα καθώς η αναδρομική αυτή συνάρτηση θα εκτελείτο τουλάχιστον $\log n$ φορές για κάθε κόμβο που διαπερνούσαμε με αποτέλεσμα να αυξανόταν η χρονική πολυπλοκότητα των αλγορίθμων. Επομένως γίνεται εύκολα αντιληπτό πως αν προταθεί διαφορετικός τρόπος ενημέρωσης υψών για τα AVL δέντρα που θα επιλύνει το πρόβλημα ενημέρωσης ύψους, που παρουσιάζει η προτεινόμενη μας μορφή, διατηρώντας τη χρονική πολυπλοκότητα των αλγορίθμων η αναστρέψιμη περιστροφή θα μπορεί να χρησιμοποιηθεί και σε αυτά.

Χρονική και Χωρική Πολυπλοκότητα

Η προτεινόμενη μορφή αναστρέψιμων περιστροφών διατηρεί τη χρονική πολυπλοκότητα της μη αναστρέψιμης μορφής χωρίς τη χρήση πληροφορίας επομένως διατηρεί και τη χωρική πολυπλοκότητα της μη αναστρέψιμης μορφής του αλγορίθμου και αποτελεί υγιεινό αναστρέψιμο αλγόριθμο.

5.4 Συνάρτηση Splay

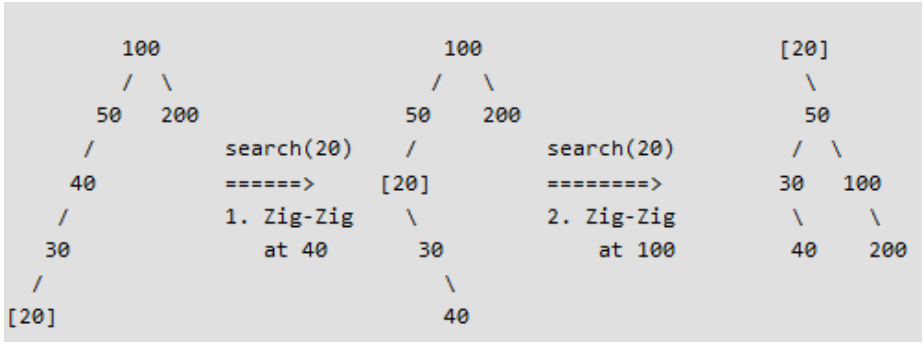
Η συνάρτηση αυτή αποτελεί την καρδιά της δομής και είναι υπεύθυνη να φέρει στη ρίζα το κόμβο που αναζητάμε. Σε περίπτωση που δεν υπάρχει ο κόμβος που αναζητάμε, επιστρέφει ως ρίζα το κόμβο που επισκέφθηκε τελευταίο κατά την αναζήτηση. Για το λόγο αυτό, η συνάρτηση αυτή χρησιμοποιείται και ως συνάρτηση αναζήτησης καθώς εντοπίσει το κόμβο που θέλουμε, αν υπάρχει, και το μεταφέρει στη ρίζα του δέντρου.

Η μετακίνηση του κόμβου από τη θέση που βρίσκεται στη ρίζα του δέντρου γίνεται με τη βοήθεια περιστροφών, οι οποίες εκτελούνται υπό τις διαφορετικές περιπτώσεις. Με τον όρο «κλειδί» εννοούμε την τιμή του κόμβου που αναζητάμε.

Ο αλγόριθμος μας ακολουθεί την εξής διαδικασία:

1. Αν ο κόμβος είναι κενός ή ισούται με το κλειδί τον επιστρέφουμε
2. Αν το κλειδί βρίσκεται στο αριστερό υποδένδρο
3. Αν το υποδένδρο είναι κενό, επιστρέφουμε το τρέχον κόμβο
4. Αν το αριστερό παιδί είναι μεγαλύτερο από το κλειδί **(Zig-Zig)**
5. Φέρνουμε αναδρομικά το κλειδί ως ρίζα του αριστερού υποδένδρου του αριστερού μας παιδιού και κάνουμε δεξιά περιστροφή στο τρέχον κόμβο
6. Αλλιώς αν το αριστερό παιδί είναι μικρότερο από το κλειδί **(Zig-Zag)**
7. Φέρνουμε αναδρομικά το κλειδί ως ρίζα του δεξιού υποδένδρου του αριστερού μας παιδιού. Αν η ρίζα αυτή δεν είναι κενή κάνουμε αριστερή περιστροφή στο αριστερό μας παιδί.
8. Αν το αριστερό παιδί είναι κενό επιστρέφουμε το τρέχον κόμβο αλλιώς εκτελούμε δεξιά περιστροφή στο τρέχον κόμβο.
9. Αλλιώς αν το κλειδί βρίσκεται στο δεξιό υποδένδρο
10. Αν το υποδένδρο είναι κενό επιστρέφουμε το τρέχον κόμβο
11. Αν το δεξιό παιδί είναι μεγαλύτερο από το κλειδί **(Zag-Zig)**
12. Φέρνουμε αναδρομικά το κλειδί ως ρίζα του αριστερού υποδένδρου του δεξιού μας παιδιού. Αν η ρίζα αυτή δεν είναι κενή κάνουμε δεξιά περιστροφή στο δεξιό μας παιδί.
13. Αλλιώς το δεξιό παιδί είναι μικρότερο από το κλειδί **(Zag-Zag)**
14. Φέρνουμε αναδρομικά το κλειδί ως ρίζα του δεξιού υποδένδρου του δεξιού μας παιδιού και κάνουμε δεξιά περιστροφή στο δεξιό μας παιδί
15. Αν το δεξιό παιδί είναι κενό επιστρέφουμε το τρέχον κόμβο αλλιώς εκτελούμε αριστερή περιστροφή στο τρέχον κόμβο.

Εξαιτίας των πολλών συγκρίσεων που απαιτούνται, ο αλγόριθμος μπορεί εύκολα να προκαλέσει σύγχυση. Για τον λόγο αυτό θα ήταν αρκετά βοηθητικό να δούμε ένα παράδειγμα εκτέλεσης.



Σχήμα 13: Παράδειγμα αναζήτησης

Στο παράδειγμα του Σχήματος 13 ψάχνουμε το κόμβο με κλειδί 20. Αρχικά ικανοποιείται η πρώτη περίπτωση Zig-Zig καθώς το αριστερό μας παιδί, κόμβος 50, είναι μεγαλύτερος από το κλειδί που ψάχνουμε. Έτσι με τη χρήση της αναδρομής και των περιστροφών η ρίζα του αριστερού παιδιού του κόμβου 50, που αποτελεί το αριστερό μας παιδί, γίνεται ο κόμβος 20. Έπειτα ο αλγόριθμος μας καθώς επιστρέφει αναδρομικά πίσω στη ρίζα. Κατά την επιστροφή εκτελούνται ακόμη 2 δεξιές περιστροφές καθώς το αριστερό παιδί των κόμβων 50 και 100 δεν είναι κενό.

Ο αλγόριθμος κάνει χρήση των περιστροφών και στη χείριστη περίπτωση θα περάσει από όλους τους κόμβους. Έτσι θα έχει χρονική πολυπλοκότητα O(n) όμως έχει αποσβεστική (amortize) πολυπλοκότητα O($\log n$).

Αναστρέψιμη μορφή

Η αναστρέψιμη μορφή που προτείνουμε ακολουθεί πιστά τη μη αναστρέψιμη μορφή. Λόγω των πολλών συγκρίσεων, των περιστροφών που γίνονται και της χρήσης αναδρομής που κάνει ο αλγόριθμος, δεν καταφέραμε να καθορίζουμε συνθήκες αναστρεψιμότητας χωρίς τη χρήση πληροφορίας. Σε μία προσπάθεια μείωσης των επιπλέον πληροφοριών που θα χρειαζόμασταν, καταφέραμε να βρούμε συνθήκες οι οποίες να απαιτούσαν μόνο μία επιπλέον πληροφορία σε κάθε αναδρομική κλήση. Αυτό επιτεύχθηκε κάνοντας χρήση μίας μεταβλητής-σημαίας η οποία για κάθε περίπτωση παίρνει διαφορετική τιμή(συνθήκη αναστρεψιμότητας) ώστε να γνωρίζουμε κατά την αντίστροφη εκτέλεση ποια συνθήκη πρέπει να ακολουθηθεί ώστε να επιστρέψουμε στην προηγούμενη κατάσταση.

Η μεταβλητή αυτή παίρνει τις τιμές τις ως ακολούθως:

a) Αρχικοποιείται με 0

b) Zig-Zig: γίνεται 1

c) Zig-Zag: γίνεται 2. Αν η ρίζα που εντοπίζουμε δεν είναι κενή: γίνεται 3.

d) Αν το κλειδί ανήκει στο αριστερό υποδένδρο και μετά τις περιστροφές το αριστερό παιδί δεν είναι κενό-μηδέν: γίνεται αρνητικό και αφαιρείται 1 ώστε αν ήταν 0 να γίνει αρνητικό.

e) Αν το κλειδί ανήκει στο αριστερό υποδένδρο και δεν είναι αρνητικό κατά την έξοδο αύξησε τη σημαία κατά 1. Χρειάζεται για τη περίπτωση που το κλειδί έπρεπε να είναι στο αριστερό υποδένδρο όμως αυτό είναι κενό.

f) Zag-Zig: γίνεται πέντε. Αν η ρίζα που εντοπίζουμε δεν είναι κενή: γίνεται 6.

g) Zag-Zag: γίνεται 7.

h) Αν το κλειδί ανήκει στο δεξιό υποδένδρο και μετά τις περιστροφές το δεξιό παιδί δεν είναι κενό: γίνεται αρνητικό και αφαιρείται 5 ώστε αν ήταν 0 να γίνει αρνητικό.

i) Αν το κλειδί ανήκει στο δεξιό υποδένδρο και δεν είναι αρνητικό κατά την έξοδο αύξησε τη σημαία κατά 5. Χρειάζεται για τη περίπτωση που το κλειδί έπρεπε να είναι στο δεξιό υποδένδρο όμως αυτό είναι κενό. Γίνεται αύξηση κατά 5 διότι η μεγαλύτερη τιμή που μπορεί να πάρει η σημαία όσο βρίσκεται στο αριστερό υποδένδρο είναι 4.

Οι συνθήκες αναστρεψιμότητας ισούνται με τη τιμή που παίρνει η σημαία μας σε κάθε περίπτωση. Ακόμη όπου έπρεπε να κάνουμε δεξιά περιστροφή εκτελούσαμε αριστερή περιστροφή προς τα πίσω.

Ακολουθεί ο ψευδοκώδικας της διαδικασίας:

```
splay(int key, node root, stack splay)
a. local int flag = 0
1. if root != null && root.value != key then
2.     if key < root.value then
3.         if root.left != null then
4.             if key < root.left.value then
5.                 call splay(key, root.left.left, splay)
                   uncall Rotate(root)
b.                 flag += 1
6.             else
                   if key > root.left.value then
7.                     call splay(key, root.left.right, splay)
c.                     flag += 2
                       if root.left.right != null then
                          call Rotate(root.left)
c.                        flag += 1
                       fi flag = 3
                   fi flag = 2 || flag = 3
               fi flag = 1

8.             if root.left != null then
                   uncall Rotate(root)
d.                 local int x = flag * (-1)
                   flag <=> x
                   delocal int x = flag * (-1)
                   flag -= 1
               fi flag < 0
           fi flag != 0

e.         if !(flag < 0) then
               flag += 1
           fi !(flag < 0)
9.     else
10.        if root.right != null then
11.            if key < root.right.value then
12.                call splay(key, root.right.left, splay)
f.                 flag += 5
                   if root.right.left != null then
                      uncall Rotate(root.right)
f.                    flag += 1
                   fi flag = 6
13.            else
                   if key > root.right.value then
14.                    call splay(key, root.right.right, splay)
                       call Rotate(root)
g.                      flag += 7
                   fi flag = 7
               fi flag = 5 || flag = 6

15.            if root.right != null then
                   call Rotate(root)
h.                 local int x = flag * (-1)
                   flag <=> x
                   delocal int x = flag * (-1)
                   flag -= 5
               fi flag < 0
           fi flag != 0
i.         if !(flag < 0) then
               flag += 5
           fi !(flag < 0)
       fi (flag >= 1 && flag <= 4) || (flag <= -1 && flag >= -3)
    fi flag != 0
    push(flag, splay)
    delocal int flag = 0
```

Στα βήματα 5, 7, 12, 14 για την υλοποίηση στη Janus χρειάζεται η χρήση της τεχνικής που αναφέραμε νωρίτερα κατά τις αναπροσαρμογές στα AVL δέντρα για κάλεσμα συνάρτησης με παράμετρο θέση πίνακα.

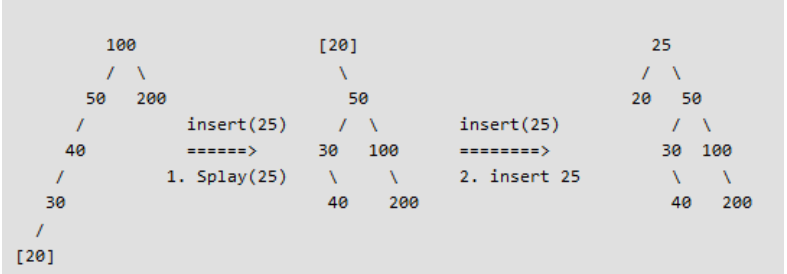
Χρονική και Χωρική Πολυπλοκότητα

Ο αλγόριθμος μας διατηρεί τη χρονική του πολυπλοκότητα όμως αυξάνει τη χωρική καθώς σε κάθε αναδρομική κλήση της συνάρτησης δημιουργείται και φυλάσσεται μία νέα μεταβλητή-σημαία. Η μεταβλητή αυτή θεωρούμε πως είναι αναγκαία καθώς υπάρχουν πάρα πολλοί συνδυασμοί περιπτώσεων λόγω ώστε να επιτύχουμε επαλήθευση της εκάστοτε συνθήκης αναστρεψιμότητας χωρίς τη χρήση επιπρόσθετης πληροφορίας. Η πληροφορία αυτή φυλάσσεται σε στοίβα διότι αν κάναμε χρήση πίνακα, θα έπρεπε αυτός να έχει μέγεθος O(n) εξαιτίας της χείριστης περίπτωσης και έτσι σε όλες τις άλλες περιπτώσεις θα σπαταλούσαμε αρκετή μνήμη αχρείαστα. Από τη στιγμή που σε κάθε αναδρομική κλήση φυλάσσουμε μία πληροφορία η χωρική πολυπλοκότητα των επιπλέον πληροφοριών εξαρτάται από τη χρονική πολυπλοκότητα. Επομένως στη χείριστη περίπτωση θα αυξάνουμε τη χωρική πολυπλοκότητα κατά O(n) όμως θα έχουμε αύξηση της αποσβεστικής (amortize) πολυπλοκότητα της τάξης του O($\log n$).

Συνοψίζοντας η προτεινόμενη αναστρέψιμη μορφή της συνάρτησης splay είναι πιστή αλλά όχι υγιεινή.

5.5 Εισαγωγή

Υπάρχουν δύο γνωστές τεχνικές. Κανονική εισαγωγή δυαδικού δέντρου αναζήτησης και έπειτα splaying ή μεταφέροντας το κοντινότερο φύλλο στη κορυφή, ορίζοντας το νέο κόμβο ως ρίζα και έπειτα ενώνοντας το νέο κόμβο με το υπόλοιπο δέντρο, θέτοντας τη παλιά ρίζα του δέντρου ως αριστερό παιδί του νέου κόμβου αν η τιμή της είναι μικρότερη αλλιώς ως δεξιό παιδί του νέου κόμβου με τέτοιο τρόπο ώστε να διατηρείται η δυαδική ιδιότητα αναζήτησης [18]. Τέλος θέτουμε το νέο κόμβο ως τη ρίζα του δέντρου μας.



Σχήμα 14: Παράδειγμα εισαγωγής του κόμβου με τιμή 25

Όπως προαναφέραμε έχουμε αποσβεστική (amortize) χρονική πολυπλοκότητα $O(\log n)$ και στη χείριστη περίπτωση, όπου τα στοιχεία εισάγονται είτε σε αύξουσα είτε σε φθίνουσα σειρά, έχουμε O(n) καθώς θα χρειαστεί να επισκεφθούμε από όλα τα στοιχεία.

Αναστρέψιμη μορφή

Η λύση που προτείνουμε χρησιμοποιεί τη δεύτερη περίπτωση. Για την υλοποίηση της χρησιμοποιήσαμε μία βοηθητική μεταβλητή που δείχνει τη θέση του νέου κόμβου, δηλαδή τη θέση μετά το τέλος του υπάρχων δέντρου. Στη γενικευμένη μορφή η μεταβλητή αυτή παραλείπεται. Επιπρόσθετα καλείται η συνάρτηση splay.

Η εισαγωγή ενώνει το νέο κόμβο με το δένδρο μέσω δύο περιπτώσεων ανάλογα τη τιμή του νέου κόμβου. Έτσι δημιουργήθηκε η ανάγκη εντοπισμού συνθήκης αναστρεψιμότητας. Μετά την ολοκλήρωση όμως της εισαγωγής δεν είμαστε σε θέση να κάνουμε κάποια σύγκριση διότι η μόνη σχέση μεταξύ παλιάς και νέας ρίζας είναι ότι η παλιά έγινε παιδί της νέας. Όμως δεν γνωρίζουμε τη θέση της παλιάς ρίζας για να πραγματοποιήσουμε αυτό το έλεγχο, δηλαδή να ελέγξουμε αν η παλιά ρίζα είναι αριστερό ή δεξιό παιδί της νέας, χωρίς τη χρήση επιπλέον πληροφορίας.

Λαμβάνοντας αυτά υπόψη αποφασίσαμε να χρησιμοποιήσουμε μία επιπρόσθετη πληροφορία, η οποία θα λειτουργούσε ως σημαία και θα μας έδειχνε ποια περίπτωση εισαγωγής έχουμε:

- Κενό δέντρο: Ο νέος κόμβος γίνεται απλά η ρίζα του δέντρου
- Περίπτωση όπου το κλειδί υπάρχει: Τέλος καθώς δεν επιτρέπονται διπλότυπα.
- Περίπτωση όπου το κλειδί του νέου κόμβου είναι μικρότερο της ρίζας: Ο νέος κόμβος ενώνεται με το δέντρο κάνοντας τη προηγούμενη ρίζα δεξιό του παιδί και το αριστερό της παιδί ως δικό του αριστερό παιδί. Έπειτα ορίζεται ως η νέα ρίζα του δέντρου.
- Περίπτωση όπου το κλειδί του νέου κόμβου είναι μεγαλύτερο της ρίζας: Ο νέος κόμβος ενώνεται με το δέντρο θέτοντας τη παλιά ρίζα ως αριστερό του παιδί και το δεξιό της παιδί να αποτελεί το δικό του δεξιό παιδί. Έπειτα ορίζεται ως η νέα ρίζα του δέντρου.

Ο αλγόριθμος αναστρέψιμης εισαγωγής που προτείνουμε είναι ο εξής:

1. Θέσε τη σημαία ίση με 0
2. Αν ο νέος κόμβος είναι η ρίζα του δέντρου, πρόσθεσε την τιμή του
3. Αλλιώς
4. Εκτέλεσε splay για να φέρεις το κοντινότερο φύλλο στη ρίζα
5. Αύξησε τη σημαία κατά 1
6. Αν η ρίζα δεν ισούται με τη τιμή που εισάγουμε
7. Αν η τιμή είναι μικρότερη
8. Θέσε την τιμή του νέου κόμβου
9. Θέσε τη ρίζα ως δεξιό παιδί του νέου κόμβου
10. Θέσε το αριστερό παιδί της ρίζας να είναι το αριστερό παιδί του νέου κόμβου
11. Θέσε το νέο κόμβο ως τη ρίζα του δέντρου
12. Αύξησε τη σημαία κατά 1
13. Αλλιώς
14. Θέσε τη τιμή του νέου κόμβου
15. Θέσε τη ρίζα ως αριστερό παιδί του νέου κόμβου
16. Θέσε το δεξιό παιδί της ρίζας να είναι το δεξιό παιδί του νέου κόμβου
17. Θέσε το νέο κόμβο ως τη ρίζα του δέντρου
18. Αύξησε τη σημαία κατά 2
19. **Συνθήκη αναστρεψιμότητας:** Η σημαία ισούται με 2
20. **Συνθήκη αναστρεψιμότητας**: Η σημαία ισούται με 2 ή 3
21. **Συνθήκη αναστρεψιμότητας:** Η σημαία ισούται με 0
22. Φύλαξε τη σημαία στη στοίβα.

Ακολουθεί ο ψευδοκώδικας της διαδικασίας:

```
procedure insert(node root, int key, node end, stack splay)
1.    local int flag = 0
2.    if root = end then
            end.value += key
3.    else
4.          call splay(key, root, splay)
5.          flag += 1
6.          if root.value != key then
7.                if key < root.value then
8.                      end.value += key
9.                      end.right += root
                        root.parent += end
10.                     end.left += root.left
                        root.left -= end.left

                        local node left = end.left
                        if left != null then
                              left.patent -= root
                              left.parent += end
                        fi left != null
                        delocal node left = end.left
11.                     root -= end.right
                        root += end
12.                     flag += 1
13.               else
14.                     end.value += key
15.                     end.left += root
                        root.parent += end
16.                     end.right += root.right
                        root.right -= end.right

                        local node right = end.right
                        if right != null then
                              right.parent -= root
                              right.parent += end
                        fi right != null
                        delocal node right = end.right

17.                     root -= end.left
                        root += end
18.                     flag += 2
19.               fi flag = 2
20.         fi flag = 2 || flag = 3
21.   fi flag = 0
22.   push(flag, splay)
      delocal int flag = 0
```

Χρονική και Χωρική Πολυπλοκότητα

Ο αλγόριθμος μας διατηρεί τη χρονική πολυπλοκότητα της μη αναστρέψιμης μορφής, διότι όπως βλέπουμε από το πιο πάνω ψευδοκώδικα την ακολουθεί πιστά. Δυστυχώς όμως λόγω των επιπλέον πληροφοριών που παράγονται κατά τη κλήση της συνάρτησης splay, η χωρική μας πολυπλοκότητα αυξάνεται στη χείριστη περίπτωση κατά O(n) και κατά τη αποσβεστική πολυπλοκότητα καταγράφεται μία αύξηση της τάξης του $O(\log n)$. Η μεταβλητή σημαία που χρησιμοποιούμε για την επαλήθευση των συνθηκών αναστρεψιμότητας δημιουργείται μόνο μία φορά και για αυτό θεωρείται αμελητέα ποσότητα η οποία δεν επηρεάζει τη χωρική πολυπλοκότητα του αλγορίθμου μας.

Άρα καταλήγουμε στο συμπέρασμα πως η αναστρέψιμη μορφή εισαγωγής σε splay δέντρα που προτείνουμε είναι πιστή αλλά όχι υγιεινή. Σε περίπτωση, όμως, που προταθεί αναστρέψιμη υγιεινή μορφή για τη συνάρτηση splay, δηλαδή υλοποίηση όπου δεν θα χρειάζεται να φυλάει επιπλέον πληροφορίες, τότε αυτόματα η εισαγωγή που προτείνουμε θα γίνει υγιεινή.

5.6 Διαγραφή

Υπάρχουν δύο γνωστές τεχνικές. Η απλή τεχνική είναι χρήση διαγραφής δυαδικού δέντρου αναζήτησης και έπειτα μέσω splaying να μεταφέρουμε το πατέρα του διαγραφέντος κόμβου στη ρίζα. Εναλλακτικά εφαρμόζουμε splaying μεταφέροντας το κόμβο που θα διαγραφεί στη ρίζα, το διαγράφουμε και εφαρμόζουμε ένα είδος ενοποίησης στα δύο ξεχωριστά δέντρα που προκύπτουν. Αν ο διαγραφέντας κόμβος είχε μόνο δεξιό παιδί τότε το παιδί γίνεται η νέα ρίζα του δέντρου διαφορετικά το αριστερό παιδί γίνεται η νέα ρίζα, εφαρμόζεται splaying και έπειτα θέτουμε ως δεξιό παιδί της νέας ρίζας το δεξιό παιδί της προηγούμενης, ώστε να ενώσουμε τα δύο υποδένδρα που προκύπτουν.

Εμείς θα στηριχτούμε στη δεύτερη τεχνική η οποία εξηγείται αναλυτικά στο βιβλίο Algorithms των Robert Sedgewick και Kevin Wayne [18, 19]. Μαζί με το βιβλίο παρουσιάζεται και μία έτοιμη μη αναστρέψιμη μορφή της συνάρτησης στη γλώσσα προγραμματισμού Java, η οποία μπορεί να βρεθεί σε ηλεκτρονική μορφή [20]. Προτιμήθηκε η τεχνική αυτή διότι θεωρήθηκε ως μεγαλύτερη πρόκληση καθώς δεν εφαρμόζει απλή διαγραφή δυαδικού δέντρου αναζήτησης με την κλήση μίας επιπλέον διαδικασίας αλλά εφαρμόζει μία τελείως διαφορετική προσέγγιση.

Όμοια με την εισαγωγή στη χείριστη περίπτωση έχουμε χρονική πολυπλοκότητα O(n) και αποσβεστική πολυπλοκότητα O($\log n$). Ακόμη και στην εναλλακτική μορφή όπου γίνεται δύο φορές splaying η πολυπλοκότητα δεν αυξάνεται, καθώς εκτελούνται αυτόνομα. Επίσης τη δεύτερη φορά εκτελείται σε λιγότερα από n στοιχεία.

Αναστρέψιμη μορφή

Για την υλοποίηση της διαγραφής χρησιμοποιήθηκε η συνάρτηση splay και οι περιπτώσεις διαγραφής κόμβου με μόνο δεξιό ή με μόνο αριστερό παιδί των δυαδικών δέντρων αναζήτησης που εξηγήσαμε λεπτομερώς νωρίτερα (βλέπε Υποκεφάλαιο 3.5). Έτσι χρειάζεται να φυλάμε τη μία επιπλέον πληροφορία που παράγεται κατά τις διαγραφές αυτές, πληροφορία συσχετιζόμενη με το διαγραφέντα κόμβο. Στη γενικευμένη μορφή ο κόμβος αυτός θα αποτελεί δείκτη σε κενό κόμβο, επομένως θα ισούται με το κενό και θα μπορεί να διαγραφεί χωρίς να χρειάζεται η φύλαξη επιπρόσθετης πληροφορίας. Άρα στη γενικευμένη μορφή η επιπλέον αυτή πληροφορία μπορεί να αγνοηθεί. Ακόμη εξαιτίας των δυσκολιών καθορισμού συνθήκης αναστρεψιμότητας έγινε χρήση μίας μεταβλητής-σημαίας. Η σημαία αλλάζει την τιμή της ανάλογα με την περίπτωση διαγραφής που θα εκτελέσουμε. Η διαδικασία της διαγραφής δεν είναι αναδρομική επομένως η μεταβλητή αυτή θα χρειαστεί να δημιουργηθεί μόνο μία φορά.

Κάτι τέτοιο έχει ως αποτέλεσμα την αμελητέα αύξηση της χωρικής πολυπλοκότητας, καθώς θα χρειαστεί η φύλαξη μόνο μίας επιπλέον πληροφορίας και για αυτό μπορεί να αγνοηθεί.

Με την τεχνική αυτή στην ουσία μεταφέρουμε το στοιχείο που θέλουμε στη ρίζα και έπειτα το διαγράφουμε. Αυτό έχει ως αποτέλεσμα να προκύπτουν δύο αυτόνομα υποδένδρα, με ρίζες το αριστερό και το δεξιό παιδί του διαγραφέντα κόμβου. Στη συνέχεια όμως καλούμαστε να επανενώσουμε τα δύο αυτά υποδένδρα ώστε να προκύψει το δέντρο που δημιουργείται από τη διαγραφή του εκάστοτε κόμβου. Έτσι κάνουμε κάποιο είδος «ενοποίησης» των δύο υποδένδρων. Με βάση τα πιο πάνω καταλήγουμε στο συμπέρασμα πως υλοποιώντας τη συγκεκριμένη μορφή διαγραφής δείχνουμε ταυτόχρονα πως είναι αναστρέψιμη και η ενοποίηση διαφορετικών splay δέντρων. Οι δύο περιπτώσεις διαγραφής που χρησιμοποιούνται είναι πανομοιότυπες με τις διαγραφές κόμβου με μόνο ένα παιδί στα δυαδικά δέντρα αναζήτησης.

Ο αναστρέψιμος αλγόριθμος που προτείνουμε εφαρμόζει την πιο κάτω διαδικασία:

1. Θέσε τη σημαία ίση με 0
2. Αν το δέντρο δεν είναι κενό
3. Εκτέλεσε splay για να έρθει στη ρίζα ο κόμβος που θα διαγραφεί ή ο κοντινότερος του.
4. Αν το κλειδί που θέλεις να διαγράψεις ισούται με τη ρίζα
5. Αν η ρίζα είναι φύλλο
6. Διάγραψε το κόμβο και θέσε τη σημαία με 1
7. Αλλιώς αν η ρίζα δεν έχει αριστερό παιδί
8. Περίπτωση διαγραφής κόμβου με μόνο δεξιό παιδί, φύλαξε τη πληροφορία
9. Θέσε τη σημαία σε 2
10. Αλλιώς
11. Δημιούργησε τοπική μεταβλητή x που δείχνει στο δεξιό παιδί
12. Αν η μεταβλητή x δεν είναι κενό
13. Αποσύνδεσε το παιδί από το κόμβο
14. **Συνθήκη αναστρεψιμότητας:** Η μεταβλητή x δεν είναι κενή
15. Περίπτωση διαγραφής κόμβου με μόνο αριστερό παιδί,
16. Τρέξε ξανά τη συνάρτηση splay.
17. Αν η μεταβλητή x δεν είναι κενή
18. Ένωσε τη ρίζα με τη μεταβλητή x
19. **Συνθήκη αναστρεψιμότητας:** Η μεταβλητή x δεν είναι κενή
20. Θέσε τη σημαία σε 3
21. **Συνθήκη αναστρεψιμότητας:** Η σημαία ισούται με 2
22. **Συνθήκη αναστρεψιμότητας:** Η σημαία ισούται με 1

23. **Συνθήκη αναστρεψιμότητας:** Η σημαία ισούται με 1, 2 ή 3

24. Αύξησε τη σημαία κατά 1

25. **Συνθήκη αναστρεψιμότητας:** Η σημαία ισούται δεν ισούται με 0

26. Φύλαξε τη σημαία στη βοηθητική στοίβα

Ακολουθεί ο ψευδοκώδικας της διαδικασίας:

```
procedure remove(int key, node root, stack splay)
1.    local int flag = 0
2.    if root.value != null then
3.          call splay(key, root, splay)
4.          if key = root.value then
5.                if root.left = null && root.right = null then
6.                      root.value -= key
                        flag += 1
7.                else
                        if root.left = null then
                              local node child = null
8.                            call removeNodeWithRightChild(key,root,child)
                              delocal node child = null
9.                            flag += 2
10.                     else
11.                           local node x = root.right
12.                           if x != null then
13.                                 root.right -= x
                                    x.parent -= root
14.                           fi x != null

                              local node child = null
15.                           call removeNodeWithLeftChild(key,root,child)
                              delocal node child = null

16.                           call splay(key, root, splay)
17.                           if x!= null then
18.                                 root.right += x
19.                                 x.parent += root
                              fi x != null
                              delocal node x = root.right
20.                           flag += 3
21.                     fi flag = 2
22.               fi flag = 1
23.         fi flag = 1 || flag = 2 || flag = 3
24.         flag += 1
25.   fi flag != 0
26.   push(flag, splay)
      delocal int flag = 0
```

Στη υλοποίηση μας στη γλώσσα προγραμματισμού Janus η μεταβλητή child δεν θα ήταν κενή αλλά θα έδειχνε στη θέση του πίνακα από όπου διαγράφηκε ο κόμβος. Έτσι όμοια και με τη χρήση τους στα δυαδικά δέντρα αναζήτησης θα έπρεπε να φυλάμε τη πληροφορία αυτή.

Χρονική και Χωρική Πολυπλοκότητα

Ο αλγόριθμος μας διατηρεί τη χρονική πολυπλοκότητα της μη αναστρέψιμης μορφής όμως λόγω των σκουπιδιών που παράγονται κατά την splay, η χωρική πολυπλοκότητα του αλγορίθμου αυξάνεται στη χείριστη περίπτωση κατά O(n) και κατά τη αποσβεστική (amortize) πολυπλοκότητα $O(\log n)$. Επομένως ο αναστρέψιμος αλγόριθμος μας είναι πιστός αλλά όχι υγιεινός. Αν όμως προταθεί υγιεινή αναστρέψιμη μορφή της συνάρτησης splay ο αλγόριθμος μας θα γίνει υγιεινός.

Κεφάλαιο 6

Σωροί

6.1 Γνωριμία με τη δομή

Οι δυαδικοί σωροί είναι μία δομή δεδομένων που παίρνει τη μορφή ενός δυαδικού δένδρου και προτάθηκαν από τους J. W. J. Williams το 1964, ως δομή δεδομένων για τον αλγόριθμο ταξινόμησης heapsort [24]. Για τη συνέχεια της διπλωματικής εργασίας θα αναφερόμαστε σε αυτούς ως σωροί. Οι σωροί είναι ένας συνηθισμένος τρόπος εφαρμογής ουρών προτεραιότητας [18].

Συγκεκριμένα οι σωροί είναι δυαδικά δέντρα τα οποία τηρούν τις εξής δύο ιδιότητες:

1. Δομική ιδιότητα: Είναι πλήρες, δηλαδή όλα τα επίπεδα του δέντρου, εκτός ίσως από το τελευταίο (βαθύτερο) είναι πλήρως γεμάτα, και, εάν το τελευταίο επίπεδο του δέντρου δεν είναι πλήρες τότε οι κόμβοι του επιπέδου αυτού είναι γεμάτοι από αριστερά προς τα δεξιά.
2. Ιδιότητα σειράς: το κλειδί που αποθηκεύεται σε κάθε κόμβο είναι είτε μεγαλύτερο ή ίσο είτε μικρότερο ή ίσο με τα κλειδιά των παιδιών του εκάστοτε κόμβου.

Οι σωροί όπου το γονικό κλειδί είναι μεγαλύτερο ή ίσο με τα κλειδιά των παιδιών του ονομάζονται μέγιστοι σωροί. Αντίστοιχα, οι σωροί όπου η τιμή του γονικού κλειδιού, είναι μικρότερο ή ίσο με τα κλειδιά των παιδιών του ονομάζονται ελάχιστοι σωροί. Επίσης οι σωροί χρησιμοποιούνται στον αλγόριθμο ταξινόμησης heapsort, λόγω του γεγονότος ότι μπορούν να

υλοποιηθούν αποθηκεύοντας κλειδιά σε ένα πίνακα και χρησιμοποιώντας τις σχετικές τους θέσεις, μπορούμε να αναπαριστούμε σχέσεις το γονέα-παιδί εύκολα και γρήγορα.

Οι σωροί, επειδή είναι πλήρη δυαδικά δένδρα, μπορούν να αποθηκεύουν στοιχεία σε πίνακα με δύο τρόπους:

- Θέτουμε ως ρίζα τη θέση ένα του πίνακα. Αν κάποιος κόμβος *u* βρίσκεται στη θέση *i*, τότε τοποθετούμε το αριστερό του παιδί στη θέση *2i*, και το δεξιό του παιδί στη θέση *2i +1*. Ο πατέρας ενός κόμβου στη θέση *i* βρίσκεται στη θέση *⌊i/2⌋*.
- Θέτουμε ως ρίζα τη θέση μηδέν του πίνακα. Αν κάποιος κόμβος *u* βρίσκεται στη θέση *i*, τότε τοποθετούμε το αριστερό του παιδί στη θέση *2i+1*, και το δεξιό του παιδί στη θέση *2i +2*. Ο πατέρας ενός κόμβου στη θέση *i*, βρίσκεται στη θέση *⌊(i-1)/2⌋*.

Υλοποιώντας τους σωρούς με τη χρήση πινάκων δεν χρειάζονται δείκτες και έτσι εξοικονομούμε μνήμη και έχουμε πιο απλές διαδικασίες. Από την άλλη όμως πρέπει να γνωρίζουμε από την αρχή το μέγιστο μέγεθος του σωρού.

Επιπρόσθετα επειδή οι σωροί είναι πλήρες δυαδικά δέντρα έχουν μέγιστο ύψος $O(\log n)$ και έτσι οι διαδικασίες εισαγωγής και διαγραφής έχουν χρονική πολυπλοκότητα $O(\log n)$.

6.2 Αναστρέψιμη δομή και περιορισμοί στη Janus

Γνωρίζουμε από τη θεωρία πως οι σωροί αποθηκεύονται σε μονοδιάστατο πίνακα. Επιπλέον οι πίνακες αποτελούν αναστρέψιμη δομή δεδομένων η οποία υποστηρίζεται από τη γλώσσα προγραμματισμού *Janus*. Έτσι δεν χρειάστηκε κάποια τροποποίηση της υπάρχουσας τους δομή για να μπορέσουμε να τους κάνουμε αναστρέψιμους. Επίσης η δομή που χρησιμοποιούμε αρχίζει από τη θέση μηδέν του πίνακα όμως οι συναρτήσεις μπορούν να αλλάξουν εύκολα και γρήγορα ώστε να υποστηρίζουν και τις σωρούς που αρχίζουν από τη θέση ένα.

Επομένως από τη στιγμή που δεν χρειάστηκε να αντιμετωπίσουμε κάποιο περιορισμό της γλώσσας προγραμματισμού Janus καταλήγουμε στο συμπέρασμα πως οι υλοποιήσεις των συναρτήσεων που προτείνουμε στη συνέχεια είναι αναστρέψιμες ανεξαρτήτων γλώσσας, δηλαδή μπορούν να υλοποιηθούν και σε άλλες γλώσσες χωρίς καμία απολύτως αλλαγή. Η υλοποίηση τους μπορεί να βρεθεί στο Παράρτημα Δ.

6.3 percolateDown

Με τη χρήση αυτής της διαδικασίας εξασφαλίζουμε τη διατήρηση της ιδιότητας σειράς των σωρών.

Συγκρίνουμε την τιμή του τρέχοντος κόμβου με αυτή των παιδιών του. Αν κάποιο από τα παιδιά του έχει μικρότερη ή μεγαλύτερη τιμή ανάλογα με το τι σωρό υλοποιούμε, ανταλλάσσουμε τη θέση των δύο στοιχείων. Η πιο πάνω διαδικασία επαναλαμβάνεται αναδρομικά μέχρι να φθάσουμε στο τέλος του δέντρου ή μέχρι να φθάσουμε σε σημείο όπου ο κόμβος θα είναι μικρότερος ή αντίστοιχα μεγαλύτερος από τα παιδιά του.

Ακόμη μπορούμε να πούμε πως η συγκεκριμένη διαδικασία αποτελεί την καρδιά των σωρών καθώς χρησιμοποιείται σχεδόν σε όλες τις θεμελιώδεις συναρτήσεις της δομής. Μερικές από τις συναρτήσεις που την χρησιμοποιούν είναι η διαγραφή ελάχιστου ή μέγιστου στοιχείου, το κτίσιμο σωρού από πίνακα και ο αλγόριθμος ταξινόμησης heapsort.

Επίσης μπορεί να χρησιμοποιηθεί για υλοποίηση ουρών προτεραιότητας καθώς εξασφαλίζει πως στη ρίζα του δέντρου – πρώτη θέση του πίνακα - πρώτο στοιχείο στην ουρά υπάρχει το στοιχείο με τη μικρότερη ή μεγαλύτερη τιμή.

Αναστρέψιμη μορφή

Στα πλαίσια της διπλωματικής αυτής θα υλοποιήσουμε τη μέγιστη σωρό η οποία χρησιμοποιείται κατά τον αλγόριθμο ταξινόμησης heapsost. Ο αλγόριθμος μας ακολουθεί πιστά τη μη αναστρέψιμη μορφή, χρησιμοποιώντας μία επιπλέον πληροφορία για κάθε αλλαγή που προκύπτει κατά την εκτέλεση του αλγορίθμου. Η επιπλέον πληροφορία αυτή ονομάζεται μέγιστο και αποτελεί δείκτη στη θέση του κόμβου που έχει τη μεγαλύτερη τιμή από τους τρείς εμπλεκόμενους, πατέρα και τα δύο του παιδιά, ώστε να γνωρίζουμε αν χρειάζεται να μετακινηθεί κάποιο παιδί στη θέση του πατέρα και αν ναι ποιο θα είναι αυτό. Η μόνη αλλαγή που χρειάζεται να κάνουμε για υλοποίηση μέγιστης ή ελάχιστης σωρού είναι να αλλάζουμε αναλόγως τη σύγκριση των τιμών που γίνεται ανάμεσα στο πατέρα και τα παιδιά του.

Ως συνθήκες αναστρεψιμότητας όπως θα δείξουμε πιο κάτω εκμεταλλευτήκαμε το γεγονός πως η μετακίνηση των κόμβων γίνεται στο τέλος της διαδικασίας και έτσι μπορέσαμε να προτείνουμε συνθήκες οι οποίες συγκρίνουν τους κόμβους με τις τρέχουσες τους τιμές πριν τη

πιθανή μετακίνηση με αποτέλεσμα να μην χρειαστεί η δημιουργία περισσότερων επιπλέον πληροφοριών που θα αύξανε ακόμη περισσότερο τη χωρική πολυπλοκότητα του αλγορίθμου.

Η αναστρέψιμη διαδικασία *percolateDown* που προτείνουμε έχει την εξής μορφή:

1. Θέσε το μέγιστο με τη τιμή του τρέχοντος κόμβου και τη θέση των παιδιών.
2. Αν το αριστερό παιδί είναι εντός ορίων και έχει μεγαλύτερη αξία από το τρέχον
3. Θέσε το μέγιστο με τη τιμή του αριστερού παιδιού.
4. Αν το δεξιό παιδί είναι εντός ορίων και έχει μεγαλύτερη αξία από το αριστερό παιδί
5. Θέσε το μέγιστο με τη τιμή του δεξιού παιδιού.
6. **Συνθήκη Αναστρεψιμότητας:** το δεξιό παιδί είναι εντός ορίων και έχει μεγαλύτερη αξία από το αριστερό παιδί.
7. Αλλιώς αν το δεξιό παιδί είναι εντός ορίων και έχει μεγαλύτερη αξία από το τρέχον κόμβο
8. Θέσε το μέγιστο με τη τιμή του δεξιού παιδιού.
9. **Συνθήκη Αναστρεψιμότητας:** το δεξιό παιδί είναι εντός ορίων και έχει μεγαλύτερη αξία από το τρέχον.
10. **Συνθήκη Αναστρεψιμότητας:** το αριστερό παιδί είναι εντός ορίων και έχει μεγαλύτερη αξία από το τρέχον.
11. Αν το μέγιστο δεν ισούται με το τρέχον κόμβο
12. Αντάλλαξε τη τιμή του τρέχοντος με το μέγιστο
13. Κάλεσε αναδρομικά τη συνάρτηση με τρέχον κόμβο το μέγιστο
14. **Συνθήκη Αναστρεψιμότητας:** το μέγιστο δεν ισούται με το τρέχον κόμβο
15. Φύλαξε το μέγιστο στη στοίβα.
16. Αναίρεσε το μέγιστο, ισούται με 0, και τη θέση των παιδιών.

Ακολουθεί ο ψευδοκώδικας της διαδικασίας:

```
procedure percolateDown(int heap[], int current, stack heap_garbage, int
heapsize)
    local int max=current, int left=2*current+1, int right = 2*current+2

      if left <= heapsize && heap[left] > heap[current] then
            max += current+1
            if right <= heapsize && heap[right] > heap[left] then
                  max += 1
            fi right <= heapsize && heap[right] > heap[left]
      else
            if right <= heapsize && heap[right] > heap[current] then
                  max += current+2
            fi right <= heapsize && heap[right] > heap[current]
      fi left <= heapsize && heap[left] > heap[current]

      if max!= current then
            heap[current] <=> heap[max]
            call percolateDown(heap, max, heap_garbage, heapsize)
      fi max!= current

      push(max, heap_garbage)
      delocal int max = 0, int left = 2*current+1, int right = 2*current+2
```

Χρονική και Χωρική Πολυπλοκότητα

Η διαδικασία μας διατηρεί τη λογική της μη αναστρέψιμης διαδικασίας, άρα μπορούμε να συμπεράνουμε ότι έχει την ίδια χρονική πολυπλοκότητα, O ($\log n$). Λόγω του ότι η συνάρτηση χρησιμοποιεί αναδρομή για να ταξιδέψει στη σωρό, φυλάσσοντας κάθε φορά τον δείκτη στο μέγιστο από τους τρεις κόμβους που ελέγχονται, στη χειρότερη περίπτωση θα διανύσουμε απόσταση από το αρχικό κόμβο i μέχρι κάποιο φύλλο του δέντρου, δίνοντάς μας ένα μέγεθος επιπλέον πληροφοριών της τάξης του O($\log n$).

Με βάση αυτό θα μπορούσαμε να υλοποιήσουμε τη συνάρτηση με τη χρήση σταθερού πίνακα μεγέθους $\log n$ όμως κάτι τέτοιο θα σπαταλούσε αχρείαστα χώρο σε περίπτωση που δεν εκτελούσαμε τη χείριστη περίπτωση. Για το λόγο αυτό προτιμήθηκε η χρήση στοίβας ώστε να έχουμε καλύτερη μέση χρονική πολυπλοκότητα. Επιπρόσθετα η συνάρτηση μας χρησιμοποιείται κατά τη δημιουργία σωρού από πίνακα και κατά την ταξινόμηση, όπου θα εκτελεστεί αρκετές φορές. Κάθε εκτέλεση θα απαιτούσε διαφορετικό πίνακα ενώ με τη χρήση στοίβας αντιμετωπίσουμε και αυτό το πρόβλημα. Καθώς περιορίσουμε τη διαχείριση της επιπλέον πληροφορίας εντός της συνάρτησης που τη παράγει.

Συνοψίζοντας αυτό μας δίνει μια πιστή αλλά όχι υγιεινή αναστρέψιμη μορφή της *percolateDown*.

6.4 Εισαγωγή

Απλή διαδικασία όπου προσθέτουμε το νέο στοιχείο στο τέλος του σωρού και ελέγχουμε αναδρομικά τον εκάστοτε κόμβο με το πατέρα του για να ελέγξουμε αν βρίσκεται στη σωστή θέση, δηλαδή αν ο κόμβος έχει μικρότερη τιμή από το πατέρα του (αντίστοιχα μεγαλύτερη αν εκτελούμε μέγιστη σωρό). Αν ναί ανταλλάσσουμε τη θέση των δύο στοιχείων και συνεχίζουμε αναδρομικά τον έλεγχο από τη θέση του πατέρα αλλιώς σταματάμε.

Αναστρέψιμη μορφή

Για την μετατροπή της μη αναστρέψιμης μορφής σε αναστρέψιμη αρκεί η χρήση μίας βοηθητικής μεταβλητής που θα δηλώνει τον αριθμό το ανταλλαγών που έγιναν και μίας μεταβλητής που θα λειτουργεί ως δείκτης στο τέλος της σωρού για να γνωρίζουμε τη θέση που θα εισαχθεί αρχικά ο νέος κόμβος. Για να το επιτύχουμε αυτό κάνουμε χρήση αναδρομής.

Ακόμη χρησιμοποιούμε μία από κάτω προς τα πάνω λογική καθώς ξεκινάμε από κάποιο φύλλο ανταλλάσσοντας τιμές με τους προγόνους του μέχρι να τηρείται η ιδιότητα ο πατέρας να είναι μεγαλύτερος (ή αντίστοιχα μικρότερος) από τα παιδιά του.

Η διαδικασία αυτή είναι η ακόλουθη:

1. Φυλάμε τη τιμή του νέου κόμβου στο τέλος της σωρού
2. Αν δεν βρισκόμαστε στη ρίζα
3. Αν ο πατέρας μας έχει μεγαλύτερη αξία από εμάς
4. Ανταλλάσσουμε τιμές
5. Αυξάνουμε το μετρητή ανταλλαγών κατά 1
6. Συνέχισε αναδρομικά από το βήμα 2 με τρέχον κόμβο τη θέση του πατέρα.
7. **Συνθήκη Αναστρεψιμότητας:** Ο μετρητής δεν ισούται με μηδέν
8. **Συνθήκη Αναστρεψιμότητας:** Δεν βρισκόμαστε στη ρίζα

Ακολουθεί ο ψευδοκώδικας της διαδικασίας:

```
procedure insert(int value, int heap[], int siftUpTimes, int end)
      if end < size(heap) then
1.          heap[end] += value
2.          call siftUp(heap, end, siftUpTimes)
            end += 1
      fi end <= size(heap)

procedure siftUp(int heap[], int nodeIndex, int siftUpTimes)
2.    if nodeIndex != 0 then
3.          if heap[(nodeIndex-1) / 2] > heap[nodeIndex] then
4.                heap[(nodeIndex-1) / 2] <=> heap[nodeIndex]
5.                siftUpTimes += 1
6.                call siftUp(heap, (nodeIndex-1) / 2, siftUpTimes)
7.          fi siftUpTimes != 0
8.    fi nodeIndex != 0
```

Σε αυτό το σημείο να αναφέρουμε πως για την εισαγωγή αυτή υλοποιήσαμε ελάχιστη σωρό ώστε να δείξουμε ότι και τα δύο είδη σωρών είναι αναστρέψιμα. Ακόμη η εισαγωγή αυτή δεν χρησιμοποιείται κατά τον heapsort. Τέλος χρησιμοποιεί διαφορετική λογική από την percolateDown με αποτέλεσμα να καταλήγουμε στο συμπέρασμα πως είναι δυνατή η αναστρεψιμότητα σωρών με διαφορετικές υλοποιήσεις.

Χρονική και Χωρική Πολυπλοκότητα

Ο αλγόριθμός μας στη χειρότερη περίπτωση θα φθάσει μέχρι τη ρίζα κάνοντας $\log n$ συγκρίσεις και ανταλλαγές. Επομένως η χρονική πολυπλοκότητα παραμένει η ίδια με τη μη αναστρέψιμη μορφή $O(\log n)$ ενώ η χωρική πολυπλοκότητα αυξάνεται κατά δύο μεταβλητές, το μετρητή που φυλάει όλη την απαραίτητη πληροφορία που παράχθηκε κατά την εισαγωγή.

Η αύξηση αυτή όμως είναι αμελητέα, της τάξεως του O(1), επομένως διατηρείται και η χωρική πολυπλοκότητα του αλγορίθμου με αποτέλεσμα η αναστρέψιμη μορφή εισαγωγής σε σωρούς που προτείνουμε να είναι υγιεινή.

6.5 Διαγραφή Ελάχιστου - Μέγιστου στοιχείου

Γνωρίζουμε πως η ρίζα έχει τη μικρότερη ή τη μεγαλύτερη, ανάλογα με το τι είδος σωρού εφαρμόζουμε, τιμή στη σωρό. Έτσι για τη διαγραφή του μέγιστου ή αντίστοιχα του ελάχιστου στοιχείου αρκεί να αφαιρέσουμε τη ρίζα μεταφέροντας το τελευταίο στοιχείο της σωρού στη ρίζα και να εκτελούσε percolateDown από τη ρίζα της σωρού ώστε να αποκαταστήσουμε την ιδιότητα των σωρών σύμφωνα με την οποία ο πατέρας κάθε κόμβου είναι μεγαλύτερος (και αντίστοιχα μικρότερος) από τα παιδιά του.

Αναστρέψιμη μορφή

Ακολουθείται πανομοιότυπη διαδικασία με τη μη αναστρέψιμη μορφή, δηλαδή αφαιρούμε τη ρίζα, μεταφέρουμε το τελευταίο στοιχείο της σωρού στη ρίζα και εκτελούμε percolateDown.

Βήματα:

1. Αν η σωρός δεν είναι κενή
2. Φύλαξη τιμή που διαγράφεται
3. Διαγραφή ρίζας
4. Ανταλλαγή της με τελευταίο στοιχείο
5. Μείωσε το δείκτη στο τέλος της σωρού
6. Εκτέλεση percolateDown για τη ρίζα
7. Συνθήκη αναστρεψιμότητας η τιμή που διαγράφεται δεν ισούται με 0

Ακολουθεί ο ψευδοκώδικας της διαδικασίας:

```
procedure deleteMin(int valueOut, int heap[], stack garbage, int end)
1.    if end < size(heap) && end > -1 then
2.          valueOut += heap[0]
3.          heap[0] -= valueOut
4.          heap[0] <=> heap[end]
5.          end -= 1
6.          call percolateDown(heap, 0, garbage, end)
7.    fi valueOut != 0
```

Χρονική και Χωρική Πολυπλοκότητα

Αυτό έχει ως αποτέλεσμα να διατηρείται η χρονική πολυπλοκότητα του αλγορίθμου. Δυστυχώς κάτι τέτοιο δεν ισχύει για τη χωρική πολυπλοκότητα λόγω των επιπλέον πληροφοριών που χρειάζεται η εκτέλεση της percolateDown, τα οποία γνωρίζουμε πως είναι $O(\log n)$.

Με βάση αυτά καταλήγουμε στο συμπέρασμα πως η αναστρέψιμη μορφή διαγραφής ελάχιστου ή μέγιστου στοιχείου που προτείνουμε είναι πιστή αλλά όχι υγιεινή. Σε περίπτωση όμως που προταθεί αναστρέψιμη μορφή της percolateDown η οποία δεν θα παράγει σκουπίδια, η συνάρτηση μας θα γίνει αυτόματα υγιεινή.

6.6 Κτίσιμο σωρού από πίνακα

Η διαδικασία είναι απλή και γρήγορα υλοποιήσιμη καθώς γίνεται εκτέλεση της percolateDown διαδοχικά από κάτω προς τα πάνω ξεκινώντας από το πατέρα του τελευταίου στοιχείου.

Ο χρόνος εκτέλεσης της εξαρτάται από το κατά πόσο ένα στοιχείο μπορεί να μετακινηθεί προς τα κάτω στο δέντρο πριν το τερματισμό της διαδικασίας. Με άλλα λόγια, εξαρτάται από το ύψος της σωρού. Στη χειρότερη περίπτωση, ένα στοιχείο μπορεί να φθάσει μέχρι τα φύλλα.

Στο κατώτατο επίπεδο, υπάρχουν 2^(h) κόμβοι, στους οποίους δεν εκτελείται η percolateDown άρα απαιτούν χρόνο μηδέν. Στο επόμενο επίπεδο έχουμε 2^(h-1) κόμβους με δυνατότητα να μετακινηθούν προς τα κάτω κατά ένα επίπεδο. Όμοια στο 3^{ο} επίπεδο έχουμε 2^(h-2) κόμβους με δυνατότητα να μετακινηθούν προς τα κάτω κατά δύο επίπεδα. Όπως μπορείτε να δείτε δεν απαιτούν όλες οι percolateDown $O(\log n)$ και για αυτό η χρονική πολυπλοκότητα είναι O(n).

Με πιο απλά λόγια η χρονική πολυπλοκότητα της διαδικασίας είναι O(n) καθώς ο χρόνος εκτέλεσης είναι ανάλογος του αθροίσματος των υψών όλων των εσωτερικών κόμβων.

Αξίζει να αναφέρουμε πως το κτίσιμο σωρού από πίνακα θα δημιουργήσει μία σωρό με ελάχιστες διαφορές από τη σωρό που θα προέκυπταν από την εισαγωγή των στοιχείων αυτών. Κάτι λογικό καθώς οι δύο διαδικασίες αν και φαινομενικά κάνουν το ίδιο πράγμα, χρησιμοποιούν διαφορετική λογική και προσέγγιση - η εισαγωγή κάνει χρήση της συνάρτησης *siftUp* χρησιμοποιώντας μία από κάτω προς τα πάνω προσέγγιση, ενώ το κτίσιμο σωρού από πίνακα κάνει χρήση της *percolateDown*, η οποία είναι μία από πάνω προς τα κάτω προσέγγιση. Αυτό όμως δεν αναιρεί το γεγονός πως και οι δύο υλοποιήσεις είναι ορθές.

Αναστρέψιμη μορφή

Ο αλγόριθμος μας είναι ο ίδιος με τη μη αναστρέψιμη του μορφή καθώς δεν απαιτείται συνθήκη αναστρεψιμότητας αλλά μία επαναληπτική διαδικασία. Ως συνθήκη εισόδου στο βρόγχο θέτουμε μία μεταβλητή *i*, η οποία θα μειώνεται σε κάθε εκτέλεση ώστε να τρέχει ο αλγόριθμος για κάθε στοιχείο το οποίο μπορεί να χρειαστεί να μετακινηθεί προς τα κάτω, να ισούται με το πατέρα του τελευταίου στοιχείου.

Συνοπτικά ο αλγόριθμος ακολουθεί την πιο κάτω απλή διαδικασία:

1. Θέσε τοπική μεταβλητή *i* να ισούται με το πατέρα του τελευταίου στοιχείου.
2. **Συνθήκη εισόδου στο βρόγχο:** Το i ισούται με το πατέρα του τελευταίου στοιχείου
3. Εκτέλεσε *percolateDown* στο *i*
4. Επανάλαβε μέχρι το *i* να ισούται με -1 μειώνοντας το κάθε φορά κατά 1.
5. Διάγραψε τοπική μεταβλητή *i*, η οποία ισούται με -1.

Ακολουθεί ο ψευδοκώδικας της διαδικασίας:

```
procedure builtHeap(int heap[], stack heap_garbage)
1.    local int heapsize = size(heap) - 1, int i = (heapsize-1) / 2
2.    from i = (heapsize-1) / 2
      loop
3.          call percolateDown(heap, i, heap_garbage, heapsize)
            i -= 1
4.    until i = -1
5.    delocal int heapsize = size(heap) - 1, int i = -1
```

Η χρήση τοπικής μεταβλητής δεν απαιτεί τη φύλαξη επιπλέον πληροφορίας, καθώς γνωρίζουμε πως πάντοτε όταν ολοκληρώνεται ο βρόγχος η μεταβλητή i θα έχει τιμή -1 διότι πρέπει να ελέγξουμε όλα τα στοιχεία συμπεριλαμβανομένης και της ρίζας της σωρού. Έτσι έχουμε τη δυνατότητα να κάνουμε χρήση της εντολής delocal ώστε να διαγράψουμε μη μεταβλητή i. Αυτό μας βοηθάει και στην αντίστροφη εκτέλεση καθώς η μεταβλητή μας θα ισούται με -1 μόνο κατά την είσοδο στο βρόγχο.

Χρονική και Χωρική Πολυπλοκότητα

Η μορφή που προτείνουμε είναι πιστή αλλά όχι υγιεινή, διότι διατηρεί τη χρονική πολυπλοκότητα της μη αναστρέψιμης μορφής αλλά αυξάνει τη χωρική λόγω των επιπλέον πληροφοριών που παράγονται κατά την κλήση της percolateDown. Με βάση την επεξήγηση που δώσαμε νωρίτερα για τη χρονική πολυπλοκότητα συμπεραίνουμε πως η αύξηση της χωρικής πολυπλοκότητας είναι O(n), επειδή στην ουσία τα σκουπίδια που φυλάμε κατά την εκτέλεση της percolateDown αντιστοιχούν στις μετακινήσεις που γίνονται. Σε περίπτωση που

προταθεί αναστρέψιμη μορφή της percolateDown η οποία δεν θα παράγει επιπλέον πληροφορίες τότε η διαδικασία κτισίματος σωρού από πίνακα που προτείνουμε θα γίνει αυτόματα υγιεινή.

6.7 Heapsort

Heapsort ονομάζουμε τον αλγόριθμο ταξινόμησης που κάνει χρήση των σωρών και εκμεταλλευόμενος τις ιδιότητες τους επιτυγχάνει χρονική πολυπλοκότητα $O(n\log n)$ [23]. Ο αλγόριθμος αυτός είναι εύκολα υλοποιήσιμος και αποτελείται από τα δύο πιο κάτω απλά βήματα:

1. Κτίσιμο σωρού από πίνακα
2. Για κάθε στοιχείο *i* ξεκινώντας από το τελευταίο
    a. Αντάλλαξε τη ρίζα με το στοιχείο *i* της σωρού
    b. Εκτέλεσε *percolateDown* στη σωρό από τη ρίζα μέχρι τον κόμβο *i*.

Αναστρέψιμη μορφή

Η αναστρέψιμη μορφή του αλγορίθμου είναι πανομοιότυπη με τη μη αναστρέψιμη καθώς αποτελείται από βρόγχο ο οποίος δεν χρειάζεται συνθήκη αναστρεψιμότητας. Απαιτεί η συνθήκη εισόδου στο βρόγχο να είναι ορθή μόνο κατά την είσοδο μας σε αυτό. Επομένως ορίζοντας απλά ένα μετρητή στο βρόγχο, ο οποίος θα μειώνεται κάθε φορά κατά ένα ώστε να εκτελεστεί η διαδικασία για κάθε στοιχείο, η συνθήκη εισόδου θα ισχύει μόνο κατά τη $1^{η}$ εκτέλεση.

Επομένως ο αλγόριθμος μας μετατρέπεται σε αναστρέψιμος με την εξής τροποποίηση της μη αναστρέψιμης μορφής του:

1. Κτίσε τη σωρό από το δοθέντα πίνακα
2. Θέσε τοπική μετρητή *i* ίσο με το μέγεθος της σωρού
3. **Συνθήκη εισόδου στο βρόγχο:** Το *i* ισούται με το μέγεθος της σωρού
4. Αντάλλαξε τη τιμή της ρίζας με τη θέση *i*
5. Μείωσε το *i* κατά 1
6. Εκτέλεσε *percolateDown* στη ρίζα
7. Επανέλαβε μέχρι το *i* να γίνει 0
8. Διάγραψε τοπική μεταβλητή *i* η οποία θα ισούται με 0

Ακολουθεί ο ψευδοκώδικας της διαδικασίας:

```
procedure heapsort(int heap[], stack heap_garbage)
1.    call builtHeap(heap, heap_garbage)

2.    local int i = size(heap) - 1
3.    from i = size(heap) - 1
      loop
4.        heap[0] <=> heap[i]
5.        i -= 1
6.        call percolateDown(heap, 0, heap_garbage, i)
7.    until i = 0
8.    delocal int i = 0
```

Η συνθήκη του βρόγχου μας θα ισχύει και κατά την αντίστροφή εκτέλεση καθώς η μεταβλητή *i* θα ισούται με μηδέν μόνο κατά την είσοδο μας στο βρόγχο. Σε περίπτωση αντίστροφης εκτέλεσης του αλγόριθμου η σωρός, εκμεταλλευόμενη τις επιπλέον πληροφορίες θα επανέλθει στη μορφή που είχε πριν τη ταξινόμηση.

Χρονική και Χωρική Πολυπλοκότητα

Από το ποιο πάνω ψευδοκώδικα συμπεραίνουμε πως επιτυγχάνουμε διατήρηση της χρονικής πολυπλοκότητας $O(n\log n)$ αλλά όχι της χωρικής. Η αύξηση της χωρικής πολυπλοκότητας οφείλεται στην επιπλέον πληροφορία που παράγεται κατά την *percolateDown* τόσο κατά το κτίσιμο της σωρού όσο και κατά το βρόγχο μας. Γνωρίζουμε πως το κτίσιμο σωρού από πίνακα παράγει O(n) επιπλέον πληροφορίες. Επίσης η *percolateDown* φυλάει μία πληροφορία για κάθε μετακίνηση, στη χειρότερη περίπτωση $O(\log n)$ και εκτελείται για όλα τα *n* στοιχεία της σωρού. Λαμβάνοντας αυτό υπόψη συμπεραίνουμε πως κατά το βρόγχο παράγονται $O(n\log n)$ επιπλέον πληροφορίες, όμοιες δηλαδή με τη χρονική πολυπλοκότητα του αλγορίθμου καθώς για κάθε μετακίνηση φυλάμε μία επιπλέον πληροφορία.

Συνοψίζοντας συνολικά παράγονται $O(n\log n)$ + O(n) επιπλέον πληροφορίες με αποτέλεσμα η προτεινόμενη αναστρέψιμη μορφή της ταξινόμησης να είναι πιστή αλλά όχι υγιεινή. Τέλος όπως σχεδόν σε όλες τις αναστρέψιμες μορφές συναρτήσεων που προτείναμε έτσι και ο αλγόριθμος ταξινόμησης οφείλει την επιπλέον πληροφορία που παράγει στη χρήση της *percolateDown*, επομένως ανακάλυψη υγιεινής αναστρέψιμης μορφής της ο αλγόριθμός μας θα γίνει αυτόματα υγιεινός.

Κεφάλαιο 7

Συμπεράσματα

7.1 Συμπεράσματα

Έχουμε αποδείξει πως είναι εφικτή η αναστρεψιμότητα των δεντρικών δομών και προτείναμε αναστρέψιμες μορφές για τα δυαδικά δέντρα αναζήτησης, τα AVL δέντρα, τα splay δέντρα και τις σωρούς. Έχουν προταθεί αφαιρετικές υλοποιήσεις των αλγορίθμων, οι οποίες έχουν υλοποιηθεί στη γλώσσα προγραμματισμού Janus. Οι υλοποιήσεις στη γλώσσα αυτή έχουν γίνει με τη χρήση πίνακα, διότι ακόμη δεν υποστηρίζονται αναστρέψιμοι δείκτες όμως θα μπορούν εύκολα να μετατραπούν καθώς η δομή τους θα διατηρείται. Για παράδειγμα στα AVL δέντρα όταν θέλαμε να χρησιμοποιήσουμε το αριστερό παιδί του τρέχοντος κόμβου χρησιμοποιούσαμε avl[currentNode][2], το οποίο με την ύπαρξη δεικτών θα χρειαστεί να μετατραπεί σε currentNode.left. Οι υλοποιήσεις αυτές βρίσκονται στα παραρτήματα Α – Δ.

Επίσης όλοι οι αλγόριθμοι που προτείνουμε είναι πιστοί. Οι περισσότεροι είναι και υγιεινοί. Η πιστότητα των περισσότερων αλγορίθμων οφείλεται στην επιπλέον πληροφορία που δημιουργεί μία συγκεκριμένη συνάρτηση επομένως προτείνοντας υγιεινή μορφή αυτής θα οδήγηση στην αυτόματη μετατροπή τους σε υγιεινές μορφές. Οι συναρτήσεις αυτές είναι η splay για τις συναρτήσεις των Splay δέντρων και η percolateDown για τις σωρούς. Ακόμη στην εισαγωγή και τη διαγραφή των Avl δέντρων παράγονται επιπρόσθετες πληροφορίες λόγω των πολλών κλήσεων της συνάρτησης ενημέρωσης ύψους, η οποία για κάθε κλήση της χρειάζεται μία επιπλέον πληροφορία. Επομένως προτείνοντας μία εναλλακτική μορφή ενημέρωσης ύψους η συνολική επιπλέον πληροφορία που παράγεται θα μειωθεί αισθητά.

Κοντά σε αυτά, σε όλους τους πιστούς αλγόριθμους φυλάμε την επιπλέον πληροφορία σε στοίβες έτσι ώστε να πετυχαίνουμε μικρότερη αύξηση πολυπλοκότητας κατά μέσο όρο. Για παράδειγμα γνωρίζουμε πως η percoladeDown για τις σωρούς στη χείριστη περίπτωση θα

παράγει O($\log n$) σκουπίδια, άρα είναι δυνατή η χρήση πίνακα μεγέθους $\log n$. Κατά τις εκτελέσεις όμως που δεν θα έχουμε χείριστη περίπτωση, αρκετές θέσεις του πίνακα θα μένουν άδειες με αποτέλεσμα να σπαταλάμε μνήμη αχρείαστα. Επιπρόσθετα με τη χρήση στοιβών γίνεται εύκολα φύλαξη των σκουπιδιών καθώς η χρήση πίνακα θα απαιτούσε για κάθε κλήση την δημιουργία νέου ξεχωριστού πίνακα.

Συνοπτικά οι αλγόριθμοι που προτείναμε και τα χαρακτηριστικά της αναστρέψιμης μορφής τους δίνονται στον πιο κάτω πίνακα:

	Αναστρέψιμη μορφή	Αύξηση χωρητικότητας
Δυαδικά Δέντρα Αναζήτησης		
Αναζήτηση	Υγιεινή	-
Εισαγωγή	Υγιεινή	1 μεταβλητή ➔ O(1)
Διαγραφή	Υγιεινή	3 μεταβλητές ➔ O(1)
Διάσχιση	Υγιεινή	-
AVL δέντρα		
Ενημέρωση ύψους κόμβων	Υγιεινή	1 μεταβλητή ➔ O(1)
Περιστροφές	Υγιεινή	2 μεταβλητές ➔ O(1)
Αναπροσαρμογή κατά την εισαγωγή	Υγιεινή	5 μεταβλητές ➔ O(1)
Εισαγωγή	Πιστή	7$\log n$ μεταβλητές ➔ O($\log n$)
Αναπροσαρμογή κατά την διαγραφή	Υγιεινή	5 μεταβλητές ➔ O(1)
Διαγραφή	Πιστή	8 $\log n$ μεταβλητές ➔ O($\log n$)
Splay δέντρα		
Αναστρέψιμη Περιστροφή	Υγιεινή	-
Splay	Πιστή	O(n), αποσβεστικό O($\log n$)
Εισαγωγή	Πιστή	O(n), αποσβεστικό O($\log n$)
Διαγραφή	Πιστή	O(n), αποσβεστικό O($\log n$)
Σωροί		
percolateDown	Πιστή	O($\log n$)
Εισαγωγή	Υγιεινή	1 μεταβλητή ➔ O(1)
Διαγραφή Ελάχιστου -Μέγιστου στοιχείου	Πιστή	O($\log n$)
Κτίσιμο σωρού από πίνακα	Πιστή	O($\log n$)
Heapsort	Πιστή	O(n)+O(n$\log n$)

Τα πιο πάνω σκουπίδια είναι αυτά που παράγονται κατά τις υλοποιήσεις μας στη γλώσσα προγραμματισμού Janus. Στη γενικευμένη μορφή θα έχουμε μικρές διαφορές σε μερικές

διαδικασίες, όπως για παράδειγμα δεν θα χρειάζονται $\log n$ επιπρόσθετες πληροφορίες για την αντιμετώπιση του περιορισμού που αντιμετωπίσαμε κατά τη διαγραφή στα AVL δέντρα.

Με βάση τον πιο πάνω πίνακα παρατηρούμε πως η ενημέρωση ύψους κόμβων στα AVL δέντρα φυλάει μόνο μία μεταβλητή. Στην υλοποίηση μας όμως (βλέπε παράρτημα Β), φυλάμε τη μεταβλητή αυτή σε στοίβα. Με το τρόπο αυτό καταφέρνουμε την εσωτερική διαχείριση της επιπλέον πληροφορίας που παράγει η συγκεκριμένη διαδικασία καθώς με το τρόπο αυτό η πληροφορία δημιουργείται και φυλάσσεται εντός της διαδικασίας και δεν μεταφέρεται ως παραμέτρους σε άλλες συναρτήσεις οι οποίες θα έπρεπε να την διαχειριστούν. Αυτό θεωρούμε πως είναι κάτι το βοηθητικό καθώς η συνάρτηση αυτή καλείται αρκετές φορές κατά την εισαγωγή και τη διαγραφή, λόγω των περιστροφών και των ενημερώσεων που απαιτούνται. Σε περίπτωση που κάποιος επιθυμεί η επιπρόσθετη πληροφορία αυτή να φυλάσσεται σε ξεχωριστή μεταβλητή αρκεί απλά να αλλάξει μία παράμετρο της κλήσης της συνάρτησης, αλλάζοντας τη στοίβα σε ένα ακέραιο και να τη διαχειρίζεται εξωτερικά.

Επιπρόσθετα κατά τα splay δέντρα προτείναμε αναστρέψιμη περιστροφή η οποία με προς τα εμπρός εκτέλεση κάνει αριστερή περιστροφή και με προς τα πίσω κάνει δεξιά περιστροφή. Δηλαδή αποδείξαμε πως οι περιστροφές αυτές είναι αναστρέψιμες η μία της άλλης. Δυστυχώς όμως η μορφή αυτή δεν μπορεί να χρησιμοποιηθεί στα AVL δέντρα λόγω της ενημέρωσης ύψους κόμβων που μπορεί να δημιουργήσει προβλήματα, για παράδειγμα εκτέλεση περιστροφής σε δέντρο όπου εισάγαμε στοιχεία σε αύξουσα ή φθίνουσα σειρά. Σε περίπτωση που προταθεί διαφορετική αναστρέψιμη μορφή ενημέρωσης ύψους κόμβων που δεν θα επηρεάζει τις περιστροφές, τότε θα μπορεί να χρησιμοποιηθεί η αναστρέψιμη μορφή που προτείναμε στα splay δέντρα και για τα AVL.

Όσο αφορά τις αναπροσαρμογές θεωρούμε πως είναι υγιεινές καθώς για n μικρότερο του 5, μπορούμε να υλοποιήσουμε μία πιο απλή μορφή του AVL δέντρου ώστε να χειριζόμαστε τις περιπτώσεις του. Για μεγάλα n όμως η αύξηση πολυπλοκότητας κατά 5 μεταβλητές είναι αμελητέα και επομένως έχουμε υγιεινό αλγόριθμο.

Σχετικά με τις πιστές υλοποιήσεις μας έχουμε να κάνουμε τις εξής παρατηρήσεις:

- Τα σκουπίδια που παράγονται κατά τη εισαγωγή και διαγραφή των AVL δέντρων μπορούν να μειωθούν σε $2\log n$. Επομένως για να φυλάσσεται λιγότερη από $O(\log n)$ πληροφορία πρέπει να βρεθεί τρόπος σύγκρισης του αποτελέσματος των περιστροφών που δεν θα απαιτεί τη φύλαξη σκουπιδιών.

- Όμοια στα Splay δέντρα για να μειωθεί η αύξηση της χωρικής πολυπλοκότητας πρέπει να βρεθεί τρόπος σύγκρισης του αποτελέσματος των περιστροφών που δεν θα απαιτεί τη φύλαξη σκουπιδιών.
- Για την percolateDown φυλάσεται πληροφορία σχετικά με το μέγιστο κόμβο κάθε βήματος, δηλαδή για κάθε εναλλαγή θέσης που προκαλείτε. Για να έχουμε μικρότερη αύξηση πολυπλοκότητας πρέπει βρεθεί τρόπος να γνωρίζουμε το μονοπάτι που διασχίσαμε χωρίς επιπρόσθετη πληροφορία.

7.2 Μελλοντική Εργασία

Έχουμε δείξει πως είναι εφικτή η μετατροπή των δεντρικών δομών σε αναστρέψιμες. Αυτό όμως δεν είναι αρκετό καθώς υπάρχουν αρκετά ακόμη είδη που πρέπει να προταθεί αναστρέψιμη μορφή τους, όπως οι γράφοι και οι πίνακες κατακερματισμού. Έχει ήδη γίνει μία προσπάθεια υλοποίησης αλγορίθμων με τη χρήση γράφων [17], όμως κατά την υλοποίηση αυτή, θεωρούμε πως γνωρίζουμε ήδη τη δομή του γράφου και παραλείπονται οι μέθοδοι εισαγωγής και διαγραφής.

Πολύ χρήσιμο θα ήταν η περεταίρω μελέτη των ανάστροφων δεντρικών δομών δεδομένων που προτείνουμε ώστε να αποδειχθεί το κάτω φράγμα της αύξησης χωρητικότητας που προκαλείται κατά τους πιστούς μας αλγόριθμους. Αν αυτό είναι της ίδιας τάξης με τους αλγόριθμους μας τότε σοι αλγόριθμοί μας δεν είναι πιστοί αλλά υγιεινή. Σε αντίθετη περίπτωση, δηλαδή αν αποδειχθεί πως μπορούμε να φυλάμε λιγότερη πληροφορία, θα πρέπει να προταθούν νέες υλοποιήσεις που θα είναι υγιεινές ή αν αυτό δεν επιτευχθεί να υλοποιηθούν πιο πιστές, δηλαδή που θα παράγουν λιγότερα σκουπίδια, μορφές αλγορίθμων.

Επιπλέον θα ήταν βοηθητικό να επεκταθούν οι προτεινόμενες δομές υλοποιώντας περισσότερες συναρτήσεις και αλγορίθμους που κάνουν χρήση τους. Οι αλγόριθμοι που απομένουν ανεξερεύνητοι θα πρέπει να εφαρμοστούν και, ελπίζουμε, η παρούσα διπλωματική εργασία να βοηθήσει να διευκολυνθούν οι υλοποιήσεις, ιδίως όσον αφορά τις δομές δεδομένων που παρέχονται. Αξίζει επίσης να εξεταστεί η βελτίωση των συναρτήσεων που δεν έχουν επί του παρόντος προταθεί υγιεινές αναστρέψιμες μορφές.

Ακόμη η εξερεύνηση διαφόρων τεχνικών προγραμματισμού και σχεδίασης αλγορίθμων που δεν έχουν ακόμη μελετηθεί στον αναστρέψιμο προγραμματισμό είναι απαραίτητη. Ένα παράδειγμα είναι οι αλγόριθμοι δυναμικού προγραμματισμού, οι οποίοι πιθανόν να είναι εκ φύσεως αναστρέψιμοι καθώς πάγια στρατηγική τους είναι η διατήρηση όλων των

πληροφοριών. Αρκετά σημαντικό θα ήταν να μελετηθεί και ο αντικειμενικοστρεφής προγραμματισμός καθώς η πλειοψηφία των αλγορίθμων που χρησιμοποιούνται στις μέρες μας κάνει χρήση αντικειμένων.

Τέλος, λόγω των επιπλοκών που συνέβησαν λόγω των σημερινών περιορισμών της γλώσσας προγραμματισμού Janus, θεωρούμε απαραίτητη την περαιτέρω επέκταση της. Αυτό θα μπορούσε να περιλαμβάνει προσαρμοσμένες δομές και τύπους, οι οποίες θα μπορούσαν να διευκολύνουν τόσο την υλοποίηση των δομών δεδομένων όσο και την ευκολότερη ανάγνωση του τελικού κώδικα. Θα μπορούσε επίσης να υποστηρίζει περισσότερους τύπους δεδομένων, όπως δείκτες, πραγματικούς αριθμούς και χαρακτήρες. Κατ' ακρίβεια προσωπική μας άποψη είναι πως η αναβάθμιση της γλώσσας προγραμματισμού Janus ώστε να παρέχει περισσότερες δυνατότητες είναι πρώτιστης σημασίας. Επιθυμητό αλλά αρκετά δύσκολο θα ήταν η παροχή των όλων των υπηρεσιών-δυνατοτήτων που προσφέρουν οι μη αναστρέψιμες γλώσσες προγραμματισμού. Κάτι τέτοιο θα διευκόλυνε αφάνταστα την αναστρέψιμη μετατροπή των αλγορίθμων και αύξανε την πιθανότητα να προτείνονται υγιεινοί αλγόριθμου καθώς δεν θα χρειάζονταν επιπλέον πληροφορίες για την αντιμετώπιση του οποιουδήποτε περιορισμού, ο οποίος δεν έχει καμία σχέση με την λογική του αλγόριθμου.

Βιβλιογραφία

1. Adelson-Velksy, Georgy G., and Evgenii M. Landis. “An algorithm for the organization of information.” *Soviet Mathematics Doklady,* 1962: 1259-1263.
2. Albers , Susanne , and Marek Karpinski. “Randomized splay trees: Theoretical and experimental results.” *Inf. Process. Lett.* 81 (2002): 213-221.
3. Allen, Brian, and Ian J. Munro. “Self-Organizing Binary Search Trees.” *ACM Journal* 25 (1978): 526-535.
4. Axelsen, Holger Bock, and Tetsuo Yokoyama. “Programming Techniques for Reversible Comparison Sorts.” (Springer) 2015: 407-426.
5. Bennett, Charles H. “Logical reversibility of computation.” *IBM Journal Research and Development* 17 (1973): 525-532.
6. Bennett, Charles H. “Notes on the history of reversible computation.” *IBM Journal of Research and Development* 44 (2000): 270-278.
7. Bennett, Charles H. “Time/Space Trade-Offs for Reversible Computation.” *SIAM J. Comput.* 18 (1989): 766-776.
8. Brinkmann, Gunnar, Jan Degraer, and Karel De Loof. “Rehabilitation of an unloved child: semi-splaying.” *Software: Practice and Experience* 39 (2009): 33-45.
9. Cormen, Thomas H., Charles E. Leiserson, and Clifford Stein. *Introduction to Algorithms 3rd edition.* MIT Press, 2009.
10. “CS61B: Lecture 36 - Splay Trees.” n.d. https://people.eecs.berkeley.edu/~jrs/61b/lec/36.
11. Frank, David J. “Power-constrained scaling limits.” *IBM Journal of Research and Development* 46 (2002): 235-344.
12. Frank, Michael P. “Introduction to reversible computing: motivation, progress, and challenges.” *Proceedings of the Second Conference on Computing Frontiers.* Ischia, Italy, 2005. 385-390.
13. *Janus Extended Playground.* n.d. http://topps.diku.dk/pirc/janus-playground/.
14. Landauer, Rolf . “Irreversibility and Heat Generation in the Computing Process.” *IBM Journal of Research and Development* 5 (1961): 183-191.
15. Lange, Klaus, Pierre McKenzie, and Alain Tapp. “Reversible Space Equals Deterministic Space.” *J. Comput. Syst. Sci.* 60 (2000): 354-367.
16. Lutz, Christopher. *Janus: a time-reversible language. Letter ti R. Landauer.* 1986. http://tetsuo.jp/ref/janus.html.
17. Nøhr, Sarah Vang . *Reversible Graph Algorithms.* Thesis, University of Copenhagen, 2015.
18. Sedgewick , Robert, and Kevin Wayne. *Algorithms 4th edition.* Addison-Wesley, 2011.
19. —. *Algorithms 4th edition.* 2016. http://algs4.cs.princeton.edu/home/.
20. —. *SplayBST.java.* Algorithms 4th edition. 2016. http://algs4.cs.princeton.edu/33balanced/SplayBST.java.html.
21. Sedgewick, Robert. *Algorithms.* Addison-Wesley, 1983.
22. Toffoli, Tommaso. “Reversible Computing.” *Automata, Languages and Programming.* Noordweijkerhout,The Netherlands, 1980. 632-644.
23. Vitanyi, Paul M. B. “Time, space, and energy in reversible computing.” *Proceedings of the Second Conference on Computing Frontiers.* Ischia, Italy, 2005.
24. Williams, J. W. J. “Algorithm 232: Heapsort.” *Communications of the ACM* 7 (1964): 347-348.
25. Yokohama, Tetsuo, and Robert Gluck. “A reversible programming language and its invertible self-interpreter.” *Proceedings of the 2007 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation.* Nice, France, 2007. 144-153.

Παράρτημα Α Δυαδικά Δέντρα Αναζήτησης

Δομή:

```
// bst : 0 = value, 1 = parentPos, 2 = leftPos, 3 = rightPos, null= -1
int bst[8][4]={{0}}      // binary search tree
stack remove_garbage     // remove garbages
int posForNewnode = 1    // pos to store next node - after the end of tree
                         // automatically updated
int root = 1
```

Παράδειγμα εκτέλεσης:

```
procedure main()
      //bst: 0 = value, 1 = parentPos, 2 = leftPos, 3 = rightPos, null= -1
      int bst[8][4]={{0}}      // binary search tree
      stack remove_garbage     // remove garbages
      int posForNewnode = 1    // pos to store next node - after the end of
                               // tree, automatically updated
      int root = 1
      call insert(bst, 20, root, posForNewnode)
      call insert(bst, 30, root, posForNewnode)
      call insert(bst, 31, root, posForNewnode)
      call insert(bst, 8, root, posForNewnode)
      call insert(bst, 25, root, posForNewnode)
      call insert(bst, 26, root, posForNewnode)
      call insert(bst, 24, root, posForNewnode)

      show(bst)
      call remove(bst, 20, root, remove_garbage)
```

Συναρτήσεις:

```
/** Find node pos with value, starting from node **/
procedure find(int bst[][], int value, int node, int pos)
      if node = 0 then
            print("Does not exists")
      else
            if bst[node][0] != value then
                  if value < bst[node][0] then
                        call find(bst,value, bst[node][2], pos)
                  else
                        call find(bst,value, bst[node][3], pos)
                  fi value < bst[node][0]
            else
                  pos += node
            fi bst[node][0] != value
      fi node = 0


/** Add new node info in posForNewnode, call insertHelper to connect
newnode to bst **/
procedure insert(int bst[], int value, int root,  int posForNewnode)
      if posForNewnode < size(bst) then         // entos oriwn
            bst[posForNewnode][0] += value      // add value
            // if not root
            if posForNewnode != root then
                  call insertHelper(bst, value, root, posForNewnode)
            fi posForNewnode != root
            posForNewnode += 1      // update pos to add next new node
      fi posForNewnode-1 < size(bst)
```

```
/** Traverse bst and add newnode as left or right child of a node **/
procedure insertHelper(int bst[][], int value, int currentNode, int
posForNewnode)
      if value < bst[currentNode][0] then
            if bst[currentNode][2] = 0 then
                  // connect father to new node
                  bst[currentNode][2] += posForNewnode
                  // connect new node to father
                  bst[posForNewnode][1] += currentNode
            else
                  call insertHelper(bst, value, bst[currentNode][2],
posForNewnode)
            fi bst[currentNode][2] = posForNewnode
      else
            if bst[currentNode][3] = 0 then
                  // connect father to new node
                  bst[currentNode][3] += posForNewnode
                  // connect new node to father
                  bst[posForNewnode][1] += currentNode
            else
                  call insertHelper(bst, value, bst[currentNode][3],
posForNewnode)
            fi bst[currentNode][3] = posForNewnode
      fi value < bst[currentNode][0]


/** Remove leaf from avl and store parent to know what leaf was deleted **/
procedure removeLeaf(int bst[][], int valueOut, int currentNode, int
parent)
      bst[currentNode][0] -= valueOut                  // delete value
      parent += bst[currentNode][1]                    // store father
      if bst[parent][0] != 0 then                      // not root
            bst[currentNode][1] -= parent              // delete father
            if valueOut < bst[parent][0] then
                  bst[parent][2] -= currentNode        // delete left child
            else
                  bst[parent][3] -= currentNode        // delete right child
            fi valueOut < bst[parent][0]
      fi bst[parent][0] != 0


/** Remove node with only right child, child becomes our new pos and child
the pos of deleted node **/
procedure removeNodeWithRightChild(int bst[][], int valueOut, int
currentNode, int child)
      bst[currentNode][0] -= valueOut            // delete value
      child += bst[currentNode][3]               // store child
      bst[currentNode][3] -= child  // delete connection to right child

      bst[child][1]-= currentNode                // delete child's father
      local int father = bst[currentNode][1]
      // if father exists connect to grandchild
      if father != 0 then
            bst[currentNode][1] -= father        // delete father
            if valueOut < bst[father][0] then
                  bst[father][2] -= currentNode // disconnect left child
                  bst[father][2] += child       // connect new left child
            else
                  bst[father][3] -= currentNode // disconnect right child
                  bst[father][3] += child       // connect new right child
            fi valueOut < bst[father][0]
            bst[child][1] += father              // add new child's father
      fi father != 0
      delocal int father = bst[child][1]
      child <=> currentNode                      // go to child
```

```
/** Remove node with only left child, child becomes our new pos and child
the pos of deleted node **/
procedure removeNodeWithLeftChild(int bst[][], int valueOut, int
currentNode, int child)
      bst[currentNode][0] -= valueOut           // delete value
      child += bst[currentNode][2]              // store child
      bst[currentNode][2] -= child  // delete connection to left child

      bst[child][1]-= currentNode               // delete child's father
      local int father = bst[currentNode][1]
      // if father exists connect to grandchild
      if father != 0 then
            bst[currentNode][1] -= father       // delete father
            if valueOut < bst[father][0] then
                  bst[father][2] -= currentNode // disconnect left child
                  bst[father][2] += child       // connect left child
            else
                  bst[father][3] -= currentNode // disconnect right child
                  bst[father][3] += child       // connect new right child
            fi valueOut < bst[father][0]
            bst[child][1] += father             // add new child's father
      fi father != 0
      delocal int father = bst[child][1]
      child <=> currentNode                     // go to child



/**
 * Remove node with 2 child. Find inorder succesor, swap it with root and
 * delete successor.
 * Stack remove_garbage: info about delete case used for succesor removal
 **/
procedure removeNodeWithTwoChildren(int bst[][], int valueOut, int
currentNode, int minPos, stack remove_garbage)
      call getMin(bst, bst[currentNode][3], minPos)//find inorder successor
      bst[currentNode][0] -= valueOut               // delete value
      bst[currentNode][0] += bst[minPos][0]         // get min's value

      local int minPosValue = bst[currentNode][0]
      // remove minPos node
      call removeCase(bst, minPosValue, minPos, remove_garbage)
      delocal int minPosValue = bst[currentNode][0]



/**
 * Delete node with valueOut. If is root find removeCase else find the node
 * and then choose what type of removal is needed
 **/
procedure remove(int bst[][], int valueOut, int root, stack remove_garbage)
      local int pos = 0
      if bst[root][0] = valueOut then
            call removeCase(bst, valueOut, root, remove_garbage)
      else
            call find(bst, valueOut, root, pos)
            call removeCase(bst, valueOut, pos, remove_garbage)
      fi pos = 0
      push(pos, remove_garbage)
      delocal int pos = 0
```

```
/**
 * Check what remove is to be done for currentNode, store it in
 * remove_details and execute.
 * 0 = leaf, 1 = has both children, 2 = has only right child
 * 3 = has only left child
 **/
procedure removeCase(int bst[][], int valueOut, int currentNode, stack
remove_garbage)
      if currentNode < size(bst) then
            //case0: leaf
            local int gb = 0
            if bst[currentNode][2] = 0 && bst[currentNode][3] = 0 then
                  call removeLeaf(bst, valueOut, currentNode, gb)
                  push(gb, remove_garbage)
            else
                  //case1: has both children
                  if bst[currentNode][2]!=0 && bst[currentNode][3]!=0 then
                        call removeNodeWithTwoChildren(bst, valueOut,
currentNode, gb, remove_garbage)
                        //make gb negative
                        local int y = gb * (-1)
                        gb <=> y
                        delocal int y = gb * (-1)
                        push(gb, remove_garbage)
                  else
                        //case2: only right child
                        if bst[currentNode][2]=0 && bst[currentNode][3]!=0
then
                              call removeNodeWithRightChild(bst, valueOut,
currentNode, gb)
                              push(gb, remove_garbage)
                        else
                              //case3: only left child
                              call removeNodeWithLeftChild(bst, valueOut,
currentNode, gb)
                              push(gb, remove_garbage)
                        fi valueOut < bst[currentNode][0]
                  fi top(remove_garbage) < 0
            fi bst[currentNode][0] = 0
            delocal int gb = 0
      fi currentNode < size(bst)



/** Find node with min value, starting from node **/
procedure getMin(int bst[][], int node, int minPos)
      if node < size(bst) then
            if bst[node][2] = 0     then
                  minPos += node
            else
                  call getMin(bst, bst[node][2], minPos)
            fi bst[node][2] = 0
      fi node < size(bst)



/** Find node with max value, starting from node **/
procedure getMax(int bst[][], int node, int maxPos)
      if node < size(bst) then
            if bst[node][3] = 0 then
                  maxPos += node
            else
                  call getMax(bst, bst[node][3], maxPos)
            fi bst[node][3] = 0
      fi node < size(bst)
```

```
/** Preorder traversal of tree starting from currentNode **/
procedure preOrder(int bst[][], int node)
      if bst[node][0] != 0 then
            local int x = bst[node][0]
            printf("%d", x)
            delocal int x = bst[node][0]

            if bst[node][2] != 0 then
                  call preOrder(bst, bst[node][2])
            fi bst[node][2] != 0

            if bst[node][3] != 0 then
                  call preOrder(bst, bst[node][3])
            fi bst[node][3] != 0
      fi bst[node][0] != 0


/** Postorder traversal of tree starting from currentNode **/
procedure postOrder(int bst[][], int node)
      if bst[node][0] != 0 then
            if bst[node][2] != 0 then
                  call postOrder(bst, bst[node][2])
            fi bst[node][2] != 0

            if bst[node][3] != 0 then
                  call postOrder(bst, bst[node][3])
            fi bst[node][3] != 0

            local int x = bst[node][0]
            printf("%d", x)
            delocal int x = bst[node][0]
      fi bst[node][0] != 0


/** Inorder traversal of tree starting from currentNode **/
procedure inOrder(int bst[][], int node)
      if bst[node][0] != 0 then
            if bst[node][2] != 0 then
                  call inOrder(bst, bst[node][2])
            fi bst[node][2] != 0

            local int x = bst[node][0]
            printf("%d", x)
            delocal int x = bst[node][0]

            if bst[node][3] != 0 then
                  call inOrder(bst, bst[node][3])
            fi bst[node][3] != 0
      fi bst[node][0] != 0
```

Παράρτημα Β AVL Δέντρα

Δομή:

```
// avl : 0 = value, 1 = parentPos, 2 = leftPos, 3 = rightPos, 4 = height
int avl[7][5]={{0}}      // our tree, never add node in [0]
int newnodePos = 1       // end of tree - pos to store next node,
                         // automatically updated
int root = 1             // root pos
int valueOut = 10        // value to remove


stack insert_rotate      // rotate details during insert
stack insert_height      // height details during insert
stack insert_traverse    // diasxisi avl during insert


stack remove_details     // removed details of deleted node & delete case
stack remove_height      // height details during remove
stack remove_rotate      // rotate details during remove
stack remove_traverse    // diasxisi avl during remove
```

Παράδειγμα εκτέλεσης 1:

```
procedure main()
// avl : 0 = value, 1 = parentPos, 2 = leftPos, 3 = rightPos, 4 = height
      int avl[7][5]={{0}}      // our tree, never add node in [0]
      int newnodePos = 1       // end of tree - pos to store next node,
                               // automatically updated
      int root = 1             // root pos
      int valueOut = 10        // value to remove


      stack insert_rotate      // rotate details during insert
      stack insert_height      // height details during insert
      stack insert_traverse    // diasxisi avl during insert


      stack remove_details     // removed details of deleted node &
                               // delete case
      stack remove_height      // height details during remove
      stack remove_rotate      // rotate details during remove
      stack remove_traverse    // diasxisi avl during remove

      call insert(avl, root, 10, newnodePos, insert_rotate, insert_height,
insert_traverse)
      call insert(avl, root, 20, newnodePos, insert_rotate, insert_height,
insert_traverse)
      call insert(avl, root, 30, newnodePos, insert_rotate, insert_height,
insert_traverse)
      call insert(avl, root, 40, newnodePos, insert_rotate, insert_height,
insert_traverse)

      show(avl)

      call remove(avl, valueOut, root, remove_traverse, remove_details,
remove_rotate, remove_height)
```

Παράδειγμα εκτέλεσης 2:

```
procedure main()
// avl : 0 = value, 1 = parentPos, 2 = leftPos, 3 = rightPos, 4 = height
      int avl[7][5]={{0}}      // our tree, never add node in [0]
      int newnodePos = 1       // end of tree - pos to store next node,
                               // automatically updated
      int root = 1             // root pos
      int valueOut = 20        // value to remove

      stack insert_rotate      // rotate details during insert
      stack insert_height      // height details during insert
      stack insert_traverse    // diasxisi avl during insert

      stack remove_details     // removed details of deleted node &
                               // delete case
      stack remove_height      // height details during remove
      stack remove_rotate      // rotate details during remove
      stack remove_traverse    // diasxisi avl during remove

      call insert(avl, root, 8, newnodePos, insert_rotate, insert_height,
insert_traverse)
      call insert(avl, root, 30, newnodePos, insert_rotate, insert_height,
insert_traverse)
      call insert(avl, root, 20, newnodePos, insert_rotate, insert_height,
insert_traverse)
      call insert(avl, root, 9, newnodePos, insert_rotate, insert_height,
insert_traverse)
      call insert(avl, root, 25, newnodePos, insert_rotate, insert_height,
insert_traverse)
      call insert(avl, root, 26, newnodePos, insert_rotate, insert_height,
insert_traverse)

      show(avl)
      call remove(avl, valueOut, root, remove_traverse, remove_details,
remove_rotate, remove_height)
```

Συναρτήσεις:

```
/** Update height of currentNode and store old in height **/
procedure updateHeight(int avl[][], int currentNode, stack height)
      local int left = avl[currentNode][2], int right = avl[currentNode][3]

      // if node exist update its height
      if !(avl[currentNode][0] = 0 && avl[currentNode][1] = 0 && left = 0
&& right = 0) then
            // store old height
            local int t = avl[currentNode][4]
            avl[currentNode][4] -= t
            push(t, height)
            delocal  int t = 0

            // get height of taller child
            if avl[left][4] > avl[right][4] then
                  avl[currentNode][4] += avl[left][4] + 1
            else
                  avl[currentNode][4] += avl[right][4] + 1
            fi avl[left][4] > avl[right][4]
      fi !(avl[currentNode][0] = 0 && avl[currentNode][1] = 0 && left = 0
&& right = 0)
      delocal int left = avl[currentNode][2], int right =
avl[currentNode][3]
```

```
/** Right rotaion. height needed for updateHeight **/
procedure rightRotate(int avl[][], int y, stack height)
      local int x = avl[y][2]
      local int t2 = avl[x][3]
      avl[x][3] -= t2                // delete right child
      avl[x][3] += y                 // new right child
      avl[y][2] -= x                 // delete left child
      avl[y][2] += t2                // new left child

      // start:   fatherX = y        fatherY = y1      fatherT = x
      // swap1:   fatherX = x        fatherY = y1      fatherT = y
      // swap2:   fatherX = y1       fatherY = x          |
      // goal:    fatherX = y1       fatherY = x       fatherT = y
      if t2 != 0 then
            avl[x][1] <=> avl[t2][1]
            avl[x][1] <=> avl[y][1]
      else
            avl[x][1] <=> avl[y][1] // swap:    father = y1 fatherY = y
            local int a = y
            avl[y][1] -= a           // fatherY = 0
            avl[y][1] += x           // fatherY = x
            delocal int a = y
      fi t2 != 0

      delocal int t2 = avl[y][2]
      // update height
      call updateHeight(avl, y, height)
      call updateHeight(avl, x, height)
      //new root
      y -= avl[x][3]                 // delete y
      y += x                         // store x
      delocal int x = y


/** Left rotaion. height needed for updateHeight **/
procedure leftRotate(int avl[][], int x, stack height)
      local int y = avl[x][3]
      local int t = avl[y][2]
      avl[y][2] -= t                 // delete left child
      avl[y][2] += x                 // new left child
      avl[x][3] -= y                 // delete right child
      avl[x][3] += t                 // new right child

      // start:   fatherX = x1       fatherY = x       fatherT = y
      // swap1:   fatherX = y        fatherY = x       fatherT = x1
      // swap2:         |            fatherY = x1      fatherT = x
      // goal:    fatherX = y        fatherY = x1      fatherT = x
      if t != 0 then
            avl[x][1] <=> avl[t][1]
            avl[t][1] <=> avl[y][1]
      else
            avl[x][1] <=> avl[y][1] //swap: fatherX = x fatherY = y1
            local int a = x
            avl[x][1] -= a           // fatherX = 0
            avl[x][1] += y           // fatherX = y
            delocal int a = x
      fi t != 0

      delocal int t = avl[x][3]
      // update height
      call updateHeight(avl, x, height)
      call updateHeight(avl, y, height)
      // new root
      x -= avl[y][2]                 // delete x
      x += y                         // store y
      delocal int y = x
```

```
/**
 * Using value check if currentNode is unbalanced and execute rotation
 * Stack rotate store info for what rotation is done:
 * 0= None, 1=LL, 2=RR, 3=LR, 4=RL
 * Stack height needed during rotaion for updating height of chanced nodes
 **/
procedure balance(int avl[][], int value, int currentNode, stack rotate,
stack height)
      local int flag = 0

      if avl[avl[currentNode][2]][4] - avl[avl[currentNode][3]][4] > 1 then

            //Left Right case
            if value > avl[avl[currentNode][2]][0] then
                  local int left = avl[currentNode][2]
                  avl[currentNode][2] -= left
                  call leftRotate(avl, left, height)
                  avl[currentNode][2] += left
                  delocal int left = avl[currentNode][2]
                  flag += 2
            fi flag = 2

            //Left Left case
            call rightRotate(avl, currentNode, height)
            flag += 1
      else
            if avl[avl[currentNode][2]][4]-avl[avl[currentNode][3]][4] < -1
then
                  //Right Left case
                  if value < avl[avl[currentNode][3]][0] then
                        local int right = avl[currentNode][3]
                        avl[currentNode][3] -= right
                        call rightRotate(avl, right, height)
                        avl[currentNode][3] += right
                        delocal int right = avl[currentNode][3]
                        flag += 2
                  fi flag = 2

                  //Right Right
                  call leftRotate(avl, currentNode, height)
                  flag += 2
            fi flag = 2 || flag = 4

      fi flag = 1 || flag = 3

      push(flag, rotate)
      delocal int flag = 0



/** Add new node info in newnodePos and call insertHelper to connect
newnode to avl. Stacks needed in insertHelper **/
procedure insert(int avl[], int root, int value, int newnodePos, stack
rotate, stack height, stack traverse)
      if newnodePos < size(avl) then      // entos oriwn
            avl[newnodePos][0] += value   // add new node value
            avl[newnodePos][4] += 1       // new node height
            call insertHelper(avl, value, root, newnodePos, rotate, height,
traverse)
            newnodePos += 1         // update pos to add next new node
      fi newnodePos-1 < size(avl)
```

```
/**
 * Traverse avl and add newnode as left or right child of a node.
 * Then recursively update height and check if rotations are needed to
 * rebalance the tree.
 * Stack rotate: info about what rotation is done
 * Stack height: store height of node's height before update
 * Stack traverse: store path to newnode.
 **/
procedure insertHelper(int avl[][], int value, int currentNode, int
newnodePos, stack rotate, stack height, stack traverse)
      // 1. normal bst insert
      local int flag = 0

      if newnodePos != currentNode then

            if value < avl[currentNode][0] then

                  if avl[currentNode][2] = 0 then
                        // connect  father to new node
                        avl[currentNode][2] += newnodePos
                        // connect new node to father
                        avl[newnodePos][1] += currentNode
                        flag += 1
                  else
                        local int a = avl[currentNode][2]
                        avl[currentNode][2] -= a
                        call insertHelper(avl, value, a, newnodePos,
rotate, height, traverse)
                        avl[currentNode][2] += a
                        delocal int a = avl[currentNode][2]
                        flag += 2
                  fi flag = 1

            else

                  if avl[currentNode][3] = 0 then
                        // connect father to new node
                        avl[currentNode][3] += newnodePos
                        // connect new node to father
                        avl[newnodePos][1] += currentNode
                        flag += 3
                  else
                        local int a = avl[currentNode][3]
                        avl[currentNode][3] -= a
                        call insertHelper(avl, value, a, newnodePos,
rotate, height, traverse)
                        avl[currentNode][3] += a
                        delocal int a = avl[currentNode][3]
                        flag += 4
                  fi flag = 3

            fi flag = 1 || flag = 2

      fi flag != 0

      push(flag, traverse)
      delocal int flag = 0

      // 2. update Height
      call updateHeight(avl, currentNode, height)
      // 3. balance
      call balance(avl, value, currentNode, rotate, height)
```

```
/**
 * Using value check if currentNode is unbalance and execute rotation
 * Stack rotate: info for what rotation is done: 0 = None, 1=LL, 2=LR, 3=RR, 4=RL
 * Stack height: needed in rotation for updating height of chanced nodes
 **/
procedure balanceRemove(int avl[][], int currentNode, stack rotate, stack height)
	local int flag = 0
	// Left Left Case: balance > 1 && balance(root->left) >= 0
	if avl[avl[currentNode][2]][4]-avl[avl[currentNode][3]][4] > 1 &&
avl[avl[avl[currentNode][2]][2]][4]-avl[avl[avl[currentNode][2]][3]][4] >= 0 then
		call rightRotate(avl, currentNode, height)
		flag += 1
	else
		// Left Right Case: balance > 1 && balance(root->left) < 0
		if avl[avl[currentNode][2]][4]-avl[avl[currentNode][3]][4] > 1 &&
avl[avl[avl[currentNode][2]][2]][4]-avl[avl[avl[currentNode][2]][3]][4] < 0 then
			local int left = avl[currentNode][2]
			avl[currentNode][2] -= left
			call leftRotate(avl, left, height)
			avl[currentNode][2] += left
			delocal int left = avl[currentNode][2]
			call rightRotate(avl, currentNode, height)
			flag += 2
		else
			// RightRight Case: balance < -1 && balance(root->right) <= 0
			if avl[avl[currentNode][2]][4]-avl[avl[currentNode][3]][4] < -1
&& avl[avl[avl[currentNode][3]][2]][4]-avl[avl[avl[currentNode][3]][3]][4] <= 0
then
				call leftRotate(avl, currentNode, height)
				flag += 3
			else
				// RightLeft Case: balance< -1 && balance(root->right)>0
				if avl[avl[currentNode][2]][4]-
avl[avl[currentNode][3]][4] < -1 && avl[avl[avl[currentNode][3]][2]][4]-
avl[avl[avl[currentNode][3]][3]][4] > 0 then
					local int right = avl[currentNode][3]
					avl[currentNode][3] -= right
					call rightRotate(avl, right, height)
					avl[currentNode][3] += right
					delocal int right = avl[currentNode][3]
					call leftRotate(avl, currentNode, height)
					flag += 4
				fi flag = 4
			fi flag = 3
		fi flag = 2
	fi flag = 1
	push(flag, rotate)
	delocal int flag = 0

/** Remove leaf from avl and store parent to know what leaf was deleted **/
procedure removeLeaf(int avl[][], int valueOut, int currentNode, int
parent)
	avl[currentNode][0] -= valueOut		// delete value
	avl[currentNode][4] -= 1		// delete height
	parent += avl[currentNode][1]		// store father
	avl[currentNode][1] -= parent		// delete father
	if avl[parent][0] != 0 then
		if valueOut < avl[parent][0] then
			avl[parent][2] -= currentNode // delete left child
			// if parent hasn't other child reduce height
			if avl[parent][3] = 0 then
				avl[parent][4] -= 1
			fi avl[parent][3] = 0
		else
			avl[parent][3] -= currentNode // delete right child
			// if parent hasn't other child reduce height
			if avl[parent][2] = 0 then
				avl[parent][4] -= 1
			fi avl[parent][2] = 0
		fi valueOut < avl[parent][0]
	fi avl[parent][0] != 0
```

```
/** Remove node with only right child, child becomes our new pos and child
the pos of deleted node **/
procedure removeNodeWithRightChild(int avl[][], int valueOut, int
currentNode, int child)
      avl[currentNode][0] -= valueOut           // delete value
      child += avl[currentNode][3]              // store child
      avl[currentNode][3] -= child  // delete connection to right child

      avl[child][1] -= currentNode              // delete child's father
      avl[currentNode][4] -= avl[child][4]+1    // delete height

      local int father = avl[currentNode][1]
      // if father exists connect to grandchild
      if father != 0 then
            avl[currentNode][1] -= father       // delete father

            if valueOut < avl[father][0] then
                  avl[father][2] -= currentNode // delete father's child
                  avl[father][2] += child       // new father's child
            else
                  avl[father][3] -= currentNode // delete father's child
                  avl[father][3] += child       // new father's child
            fi valueOut < avl[father][0]

            avl[child][1] += father             // new child's father
      fi father != 0
      delocal int father = avl[child][1]

      child <=> currentNode                     // go to child
```

```
/** Remove node with only left child, child becomes our new pos and child
the pos of deleted node **/
procedure removeNodeWithLeftChild(int avl[][], int valueOut, int
currentNode, int child)
      avl[currentNode][0] -= valueOut           // delete value
      child += avl[currentNode][2]              // store child
      avl[currentNode][2] -= child  // delete connection to left child

      avl[child][1] -= currentNode              // delete child's father
      avl[currentNode][4] -= avl[child][4]+1    // delete height

      local int father = avl[currentNode][1]
      // if father exists connect to grandchild
      if father != 0 then
            avl[currentNode][1] -= father       // delete father
            if valueOut < avl[father][0] then
                  avl[father][2] -= currentNode // delete father's child
                  avl[father][2] += child       // new father's child
            else
                  avl[father][3] -= currentNode // delete father's child
                  avl[father][3] += child       // new father's child
            fi valueOut < avl[father][0]
            avl[child][1] += father             // new child's father
      fi father != 0
      delocal int father = avl[child][1]

      child <=> currentNode                     // go to child
```

```
/**
 * Remove node with 2 child. Find inorder succesor, swap it with root and
 * delete inorder successor.
 * Stack remove_details: info about delete case used for succesor removal
 **/
procedure removeNodeWithTwoChildren(int avl[][], int valueOut, int
currentNode, int minPos, stack remove_details)
      call getMin(avl, avl[currentNode][3], minPos)
      avl[currentNode][0] -= valueOut                  // delete value
      avl[currentNode][0] += avl[minPos][0]            // get min's value

      local int minPosValue = avl[currentNode][0]
      // remove minPos
      call removeCase(avl, minPosValue, minPos, remove_details)
      delocal int minPosValue = avl[currentNode][0]

/**
 * Find node with valueOut. deleted and recursively update height and
 * rebalance out tree
 * Stack remove_details: info about used remove case
 * Stack rotate: info about done rotations
 * Stack height: nodes old heights
 * Stack traverse: path to deleted node
 **/
procedure remove(int avl[][], int valueOut, int root, stack traverse, stack
remove_details, stack rotate, stack height)
      if root < size(avl) && root != 0 then

            local int flag = 0
            if valueOut = avl[root][0] then
                  call removeCase(avl, valueOut, root, remove_details)
            else
                  if valueOut < avl[root][0] then
                        local int x = avl[root][2]
                        call remove(avl, valueOut, x, traverse,
remove_details, rotate, height)
                        push(x, traverse)
                        delocal int x = 0
                        flag += 1
                  else
                        local int x = avl[root][3]
                        call remove(avl, valueOut, x, traverse,
remove_details, rotate, height)
                        push(x, traverse)
                        delocal int x = 0
                        flag += 2
                  fi flag = 1
            fi flag = 0
            push(flag, traverse)
            delocal int flag = 0

      fi root < size(avl) && root != 0

      // 2. update Height
      call updateHeight(avl, root, height)
      // 3. balance
      call balanceRemove(avl, root, rotate, height)
```

```
/**
 * Check what remove is to be done for currentNode, store it in
 * remove_details and execute.
 * 0 = leaf, 1 = has both children, 2 = has only right child
 * 3 = has only left child
 **/
procedure removeCase(int avl[][], int valueOut, int currentNode, stack
remove_details)
      if currentNode < size(avl) then
            local int flag = 0

            //case0: leaf
            if avl[currentNode][2] = 0 && avl[currentNode][3] = 0 then
                  call removeLeaf(avl, valueOut, currentNode, flag)
                  push(flag, remove_details)
            else
                  //case1: has both children
                  if avl[currentNode][2]!=0 && avl[currentNode][3]!=0 then
                        call removeNodeWithTwoChildren(avl, valueOut,
currentNode, flag, remove_details)
                        //make flag negative
                        local int x = flag * (-1)
                        flag <=> x
                        delocal int x = flag * (-1)
                        push(flag, remove_details)
                  else
                        //case2: only right child
                        if avl[currentNode][2]=0 && avl[currentNode][3]!=0
then
                              call removeNodeWithRightChild(avl, valueOut,
currentNode, flag)
                              push(flag, remove_details)
                        else
                              //case3: only left child
                              call removeNodeWithLeftChild(avl, valueOut,
currentNode, flag)
                              push(flag, remove_details)
                        fi valueOut < avl[currentNode][0]
                  fi top(remove_details) < 0
            fi avl[currentNode][0] = 0

            delocal int flag = 0
      fi currentNode < size(avl)



/** Preorder traversal of tree starting from currentNode **/
procedure preOrder(int avl[][], int currentNode)
      if avl[currentNode][0] != 0 then
            local int x = avl[currentNode][0]
            printf("%d", x)
            delocal int x = avl[currentNode][0]

            if avl[currentNode][2] != 0 then
                  call preOrder(avl, avl[currentNode][2])
            fi avl[currentNode][2] != 0

            if avl[currentNode][3] != 0 then
                  call preOrder(avl, avl[currentNode][3])
            fi avl[currentNode][3] != 0
      fi avl[currentNode][0] != 0
```

```
/** postOrder traversal of tree starting from currentNode **/
procedure postOrder(int avl[][], int currentNode)
      if avl[currentNode][0] != 0 then
            if avl[currentNode][2] != 0 then
                  call postOrder(avl, avl[currentNode][2])
            fi avl[currentNode][2] != 0

            if avl[currentNode][3] != 0 then
                  call postOrder(avl, avl[currentNode][3])
            fi avl[currentNode][3] != 0

            local int x = avl[currentNode][0]
            printf("%d", x)
            delocal int x = avl[currentNode][0]

      fi avl[currentNode][0] != 0



/** Inorder traversal of tree starting from currentNode **/
procedure inOrder(int avl[][], int currentNode)
      if avl[currentNode][0] != 0 then
            if avl[currentNode][2] != 0 then
                  call inOrder(avl, avl[currentNode][2])
            fi avl[currentNode][2] != 0

            local int x = avl[currentNode][0]
            printf("%d", x)
            delocal int x = avl[currentNode][0]

            if avl[currentNode][3] != 0 then
                  call inOrder(avl, avl[currentNode][3])
            fi avl[currentNode][3] != 0
      fi avl[currentNode][0] != 0



/** Find min node pos, starting from currentNode. **/
procedure getMin(int avl[][], int currentNode, int minPos)
      if currentNode < size(avl) then
            if avl[currentNode][2] = 0 then
                  minPos += currentNode
            else
                  call getMin(avl, avl[currentNode][2], minPos)
            fi avl[currentNode][2] = 0
      fi currentNode < size(avl)



/** Find node pos with value, starting from currentNode. **/
procedure find(int avl[][], int value, int root, int pos)
      if avl[root][0] != value then
            if value < avl[root][0] then
                  call find(avl,value,avl[root][2],pos)
            else
                  call find(avl,value,avl[root][3],pos)
            fi value < avl[root][0]
      else
            pos += root
      fi avl[root][0] != value
```

Παράρτημα Γ Splay Δέντρα

Δομή:

```
// splayTree : 0 = key, 1 = parentPos, 2 = leftPos, 3 = rightPos
int splayTree[8][4] = {{0}}
int end = 1       // pos after end of tree-pos to add newnode,
                  // automatically updated
int root = 1
stack splay
```

Παράδειγμα εκτέλεσης 1:

```
procedure main()
      // splayTree : 0 = key, 1 = parentPos, 2 = leftPos, 3 = rightPos
      int splayTree[8][4] = {{0}}
      int end = 1       // pos after end of tree - pos to add newnode,
                        // automatically updated
      int root = 1
      stack splay

      call insert(splayTree, root, 40, end, splay)
      call insert(splayTree, root, 30, end, splay)
      call insert(splayTree, root, 20, end, splay)
      call insert(splayTree, root, 50, end, splay)
      call insert(splayTree, root, 200, end, splay)
      call insert(splayTree, root, 100, end, splay)
      call insert(splayTree, root, 25, end, splay)

      show(splayTree)
      call remove(splayTree, 25, root, splay)
```

Συναρτήσεις:

```
/** Based on left rotation. For right rotation uncall this function **/
procedure Rotate(int splayTree[][], int x)
      local int y = splayTree[x][3]
      local int t = splayTree[y][2]
      splayTree[y][2] -= t                        // delete left child
      splayTree[y][2] += x                        // new left child
      splayTree[x][3] -= y                        // delete right child
      splayTree[x][3] += t                        // new right child

      // start:   fatherX = x1      fatherY = x       fatherT = y
      // swap1:   fatherX = y       fatherY = x       fatherT = x1
      // swap2:       |             fatherY = x1      fatherT = x
      // goal:    fatherX = y       fatherY = x1      fatherT = x
      if t != 0 then
            splayTree[x][1] <=> splayTree[t][1]
            splayTree[t][1] <=> splayTree[y][1]
      else
            splayTree[x][1] <=> splayTree[y][1] //swap:fatherX=x fatherY=y1
            local int a = x
            splayTree[x][1] -= a                  // fatherX=0
            splayTree[x][1] += y                  // fatherX=y
            delocal int a = x
      fi t != 0

      delocal int t = splayTree[x][3]
      // new root
      x -= splayTree[y][2]                        // delete x
      x += y                                      // store y
      delocal int y = x
```

```
/**
 * Brings the key at root if key is present in tree. If not, it
 * brings the last accessed item at root. Modifies the tree and
 * returns the new root.
 * Is used and as a search operation
 **/
procedure splay(int splayTree[][], int key, int root, stack splay)
	local int flag = 0
	// Base cases: root is NULL or key is present at root = > do nothing
	if splayTree[root][0] != 0 && splayTree[root][0] != key then

		// Key lies in left subtree
		if key < splayTree[root][0] then

			// Key is in tree else we are done
			if splayTree[root][2] != 0 then

				// zig zig(left left)
				if key < splayTree[splayTree[root][2]][0] then
					local int left = splayTree[root][2]
					local int left_left = splayTree[left][2]
					splayTree[left][2] -= left_left
					//First recursively bring key as root of LL
					call splay(splayTree, key, left_left, splay)
					splayTree[left][2] += left_left
					delocal int left_left = splayTree[left][2]
					delocal int left = splayTree[root][2]
					//Do 1st rotation for root, 2nd is done after else
					uncall Rotate(splayTree, root)

					flag += 1
				else

					// zig zag(left right)
					if key > splayTree[splayTree[root][2]][0] then
						local int left = splayTree[root][2]
						local int left_right = splayTree[left][3]
						splayTree[left][3] -= left_right
						//First recursively bring key as root of LR
						call splay(splayTree,key,left_right,splay)
						splayTree[left][3] += left_right
						delocal int left_right = splayTree[left][3]
						delocal int left = splayTree[root][2]
						flag += 2

						// Do first rotation for root->left
						if splayTree[splayTree[root][2]][3]!=0 then
							local int a = splayTree[root][2]
							splayTree[root][2] -= a
							call Rotate(splayTree, a)
							splayTree[root][2] += a
							delocal int a = splayTree[root][2]
							flag += 1
						fi flag = 3
					fi flag = 2 || flag = 3

				fi flag = 1

				// Do second rotation for root
				if splayTree[root][2] != 0 then
					uncall Rotate(splayTree, root)
					local int x = flag * (-1)
					flag <=> x
					delocal int x = flag * (-1)
					flag -= 1 // if flag=0 => make it negative
				fi flag < 0
			fi flag != 0

			// if flag still 0 or flag=1-3
			if !(flag < 0) then
				flag += 1
			fi !(flag < 0)
```

```
        else
                // Key lies in right subtree
                if splayTree[root][3] != 0 then

                        // zag zig  (right left)
                        if key < splayTree[splayTree[root][3]][0] then
                                local int right = splayTree[root][3]
                                local int right_left = splayTree[right][2]
                                splayTree[right][2] -= right_left

                                // Bring the key as root of right-left
                                call splay(splayTree, key, right_left, splay)
                                splayTree[right][2] += right_left
                                delocal int right_left = splayTree[right][2]
                                delocal int right = splayTree[root][3]
                                flag += 5

                                // Do first rotation for root->right
                                if splayTree[splayTree[root][3]][2] != 0 then
                                        local int a = splayTree[root][3]
                                        splayTree[root][3] -= a
                                        uncall Rotate(splayTree, a)
                                        splayTree[root][3] += a
                                        delocal int a = splayTree[root][3]
                                        flag += 1
                                fi flag = 6
                        else

                                // zag zag (right right)
                                if key > splayTree[splayTree[root][3]][0] then
                                        local int right = splayTree[root][3]
                                        local int right_right = splayTree[right][3]
                                        splayTree[right][3] -= right_right

                                        //Bring key as root of RR,do first rotation
                                        call splay(splayTree,key,right_right,splay)
                                        splayTree[right][3] += right_right
                                        delocal int right_right =
splayTree[right][3]
                                        delocal int right = splayTree[root][3]

                                        call Rotate(splayTree, root)
                                        flag += 7
                                fi flag = 7
                        fi flag = 5 || flag = 6

                        // Do second rotation for root
                        if splayTree[root][3] != 0 then
                                call Rotate(splayTree,  root)
                                local int x = flag * (-1)
                                flag <=> x
                                delocal int x = flag * (-1)
                                flag -= 5 // if flag=0 => make it negative
                        fi flag < 0
                fi flag != 0

        // if flag still 0 or flag=5-7
        if !(flag < 0) then
                flag += 5
        fi !(flag < 0)

        fi (flag >= 1 && flag <= 4) || (flag <= -1 && flag >= -3)

fi flag != 0
push(flag, splay)
delocal int flag = 0
```

```
/**
 * Add new node at the end and connect it to tree.
 * Stack splay: info about the done rotations
 **/
procedure insert(int splayTree[], int root, int key, int end, stack splay)
	if end < size(splayTree) then		// entos oriwn
		local int flag = 0
		if root = end then
			splayTree[end][0] += key   // new node key
		else
			// Bring the closest leaf node to root
			call splay(splayTree, key, root, splay)
			flag += 1

			// If node with key doesn't exist
			if splayTree[root][0] != key then
				// add new node.
				// If root's key is greater, make root as right child of
				// newnode and copy the left child of root to newnode
				if key < splayTree[root][0] then
					splayTree[end][0] += key   // new node key
					splayTree[end][3] += root // connect newnode-root
					splayTree[root][1] += end  //add root's new father

					// connect to new left child
					splayTree[end][2] += splayTree[root][2]
					// disconnect old left child
					splayTree[root][2] -= splayTree[end][2]

					local int left = splayTree[end][2]
					if left != 0 then
						// disconnect old father
						splayTree[left][1] -= root
						// connect new father
						splayTree[left][1] += end
					fi left != 0
					delocal int left = splayTree[end][2]

					root -= splayTree[end][3]  // zero clear old root
					root += end                // store new root
					flag += 1

				// If root's key is smaller, make root as left child of
				// newnode and copy the right child of root to newnode
				else
					splayTree[end][0] += key   // new node key
					splayTree[end][2] += root  // connect newnode-root
					splayTree[root][1] += end  //add root's new father

					// connect to new left child
					splayTree[end][3] += splayTree[root][3]
					// disconnect old left child
					splayTree[root][3] -= splayTree[end][3]

					local int right = splayTree[end][3]
					if right != 0 then
						// disconnect old father
						splayTree[right][1] -= root
						// connect new father
						splayTree[right][1] += end
					fi right != 0
					delocal int right = splayTree[end][3]

					root -= splayTree[end][2]  // zero clear old root
					root += end                // store new root
					flag += 2
				fi flag = 2
			fi flag = 2 || flag=3
		fi flag = 0
		push(flag, splay)
		delocal int flag = 0
		end += 1                    // update pos to add next new splayTree
	fi end-1 < size(splayTree)
```

```
/** Remove node with only right child, child becomes our new pos and child
the pos of deleted node **/
procedure removeNodeWithRightChild(int splayTree[][], int valueOut, int
currentNode, int child)
	splayTree[currentNode][0] -= valueOut		// delete value
	child += splayTree[currentNode][3]		// store child
	splayTree[currentNode][3] -= child //delete connection to right child
	splayTree[child][1]-= currentNode		// delete child's father

	local int father = splayTree[i][1]
	// if father exists connect to grandchild
	if father != 0 then
		splayTree[currentNode][1] -= father // delete father
		if valueOut < splayTree[father][0] then
			// disconnect left child
			splayTree[father][2] -= currentNode
			// connect new left child
			splayTree[father][2] += child
		else
			// disconnect right child
			splayTree[father][3] -= currentNode
			// connect new right child
			splayTree[father][3] += child
		fi valueOut < splayTree[father][0]
		splayTree[child][1] += father		// add new child's father
	fi father != 0
	delocal int father = splayTree[child][1]

	child <=> currentNode				// go to child



/** Remove node with only left child, child becomes our new pos and child
the pos of deleted node **/
procedure removeNodeWithLeftChild(int splayTree[][], int valueOut, int
currentNode, int child)
	splayTree[currentNode][0] -= valueOut		// delete value
	child += splayTree[currentNode][2]		// store child
	splayTree[currentNode][2] -= child //delete connection to left child
	splayTree[child][1]-= currentNode		// delete child's father

	local int father = splayTree[currentNode][1]
	// if father exists connect to grandchild
	if father != 0 then
		splayTree[currentNode][1] -= father		// delete father
		if valueOut < splayTree[father][0] then
			// disconnect left child
			splayTree[father][2] -= currentNode
			// connect new left child
			splayTree[father][2] += child
		else
			// disconnect right child
			splayTree[father][3] -= currentNode
			// connect new right child
			splayTree[father][3] += child
		fi valueOut < splayTree[father][0]
		splayTree[child][1] += father		// add new child's father
	fi father != 0
	delocal int father = splayTree[child][1]

	child <=> currentNode				// go to child
```

```
/**
 * Bring key to root and delete it. Then connect the two subtrees
 * Stack splay: info about the done rotations
 **/
procedure remove(int splayTree[][], int key, int root, stack splay)
      local int flag = 0
      if splayTree[root][0] != 0 then                        // not empty
            // node to be remove becomes root
            call splay(splayTree, key, root, splay)

            if key = splayTree[root][0] then
                  // only root in tree
                  if splayTree[root][2] = 0 && splayTree[root][3] = 0 then
                        splayTree[root][0] -= key
                        flag += 1
                  else
                        // right child to become new root
                        if splayTree[root][2] = 0 then
                              local int child = 0
                              call removeNodeWithRightChild(splayTree, key,
root, child)
                              push(child, splay)
                              delocal int child = 0
                              flag += 2
                        else
                              local int x = splayTree[root][3]
                              if x != 0 then
                                    splayTree[root][3] -= x
                                    splayTree[x][1] -= root
                              fi x != 0

                              local int child = 0
                              call removeNodeWithLeftChild(splayTree, key,
root, child)
                              push(child, splay)
                              delocal int child = 0

                              // Bring the closest leaf node to root
                              call splay(splayTree, key, root, splay)

                              if x!= 0 then
                                    splayTree[root][3] += x
                                    splayTree[x][1] += root
                              fi x != 0
                              delocal int x = splayTree[root][3]
                              flag += 3
                        fi flag = 2
                  fi flag = 1
            fi flag = 1 || flag = 2 || flag = 3
            flag += 1
      fi flag != 0
      push(flag, splay)
      delocal int flag = 0
```

Παράρτημα Δ Σωροί

Δομή:

```
int heap[7]
int end = 0 //pos after last node in heap, end of heap, pos to add new node
int siftUpTimes = 0           // counter of swaps during insert
stack heap_garbage      // for percolateDown
```

Παράδειγμα εκτέλεσης heapsort:

```
procedure main()

      stack heap_garbage
      int sizee
      int heap[] = {10, 5, 15, 20, 39, 1}

      call heapsort(heap, heap_garbage)
      sizee += size(heap_garbage)
      show(heap) show(sizee)
      sizee -= size(heap_garbage)

      print("\n")
      uncall heapsort(heap, heap_garbage)
      sizee += size(heap_garbage)
```

Παράδειγμα εκτέλεσης εισαγωγής:

```
procedure main()
      int heap[7]
      int end = 0 //pos after last node, end of heap, pos to add new node
      int siftUpTimes = 0           // counter of swaps during insert

      stack helper // store siftUpTimes of each insert, easier than
                   // creating different var for each time

      call insert(10,heap, siftUpTimes, end)    show(heap)
      push(siftUpTimes, helper)
      call insert(5, heap, siftUpTimes, end)    show(heap)
      push(siftUpTimes,helper)
      call insert(15, heap, siftUpTimes, end)   show(heap)
      push(siftUpTimes, helper)
      call insert(20, heap, siftUpTimes, end)   show(heap)
      push(siftUpTimes, helper)
      call insert(39, heap, siftUpTimes, end)   show(heap)
      push(siftUpTimes,helper)
      call insert(1, heap, siftUpTimes, end)    show(heap)
      push(siftUpTimes, helper)

      show(helper)      print("\nreverse\n")

      pop(siftUpTimes,helper) uncall insert(1, heap, siftUpTimes, end)
      show(heap)
      pop(siftUpTimes,helper) uncall insert(39, heap, siftUpTimes, end)
      show(heap)
      pop(siftUpTimes,helper) uncall insert(20, heap, siftUpTimes, end)
      show(heap)
      pop(siftUpTimes,helper) uncall insert(15, heap, siftUpTimes, end)
      show(heap)
      pop(siftUpTimes,helper) uncall insert(5, heap, siftUpTimes, end)
      show(heap)
      pop(siftUpTimes,helper) uncall insert(10, heap, siftUpTimes, end)
```

Παράδειγμα εκτέλεσης διαγραφής ελαχίστου:

```
procedure main()
	int heap[] = {-1, 1, 2, 3, 4, 5}
	int valueOut                    // value to be deleted
	int end = size(heap) - 1        // end of heap
	stack heap_garbage

	int sizee
	print("call")
	call deleteMin(valueOut, heap, heap_garbage, end)
	sizee += size(heap_garbage)
	show(heap) show(heap_garbage) show(sizee)
	sizee -= size(heap_garbage)

	print("\nuncall")
	uncall deleteMin(valueOut, heap, heap_garbage, end)
	sizee += size(heap_garbage)
```

Συναρτήσεις:

```
/**
 * Recursively compare father with its childern and make father the node
 * with the bigger (maxHeap) or smaller value (minHeap)
 **/
procedure percolateDown(int heap[], int current, stack heap_garbage, int
heapsize)
    local int max=current, int left=2*current+1, int right = 2*current+2

	if left <= heapsize && heap[left] > heap[current] then
		max += current+1
		if right <= heapsize && heap[right] > heap[left] then
			max += 1
		fi right <= heapsize && heap[right] > heap[left]
	else
		if right <= heapsize && heap[right] > heap[current] then
			max += current+2
		fi right <= heapsize && heap[right] > heap[current]
	fi left <= heapsize && heap[left] > heap[current]

	if max!= current then
		heap[current] <=> heap[max]
		call percolateDown(heap, max, heap_garbage, heapsize)
	fi max!= current

	push(max, heap_garbage)
	delocal int max = 0, int left = 2*current+1, int right = 2*current+2


/** Insert value to heap using siftUp method **/
procedure insert(int value, int heap[], int siftUpTimes, int end)
	if end < size(heap) then
		heap[end] += value
		call siftUp(heap, end, siftUpTimes)
		end += 1
	fi end <= size(heap)

/** Recursively swap father with child until father's value is smaller **/
procedure siftUp(int heap[], int nodeIndex, int siftUpTimes)
	if nodeIndex != 0 then
		if heap[(nodeIndex-1) / 2] > heap[nodeIndex] then
			heap[(nodeIndex-1) / 2] <=> heap[nodeIndex]
			siftUpTimes += 1
			call siftUp(heap, (nodeIndex-1) / 2, siftUpTimes)
		fi siftUpTimes != 0
	fi nodeIndex != 0
/**
```

```
 * Delete min/max value, add last node to root and perform percolateDown
 * Heap_garbage: info about the chances done in percolateDown
 **/
procedure deleteMin(int valueOut, int heap[], stack heap_garbage, int end)
      if end < size(heap) && end > -1 then
            valueOut += heap[0]
            heap[0] -= valueOut
            heap[0] <=> heap[end]
            end -= 1
            call percolateDown(heap, 0, heap_garbage, end)
      fi valueOut != 0

/**
 * Builh heap from given array using percolateDown
 * Heap_garbage: info for chances during percolateDown
 **/
procedure builtHeap(int heap[], stack heap_garbage)
      local int heapsize = size(heap) - 1, int i = (heapsize-1) / 2
      from i = (heapsize-1) / 2
      loop
            call percolateDown(heap, i, heap_garbage, heapsize)
            i -= 1
      until i = -1
      delocal int heapsize = size(heap) - 1, int i = -1

/**
 * Ascending sorting using heap.
 * Heap_garbage: info for chances during percolateDown
 **/
procedure heapsort(int heap[], stack heap_garbage)
      call builtHeap(heap, heap_garbage)

      local int i = size(heap) - 1
      from i = size(heap) - 1
      loop
          heap[0] <=> heap[i]
            i -= 1
        call percolateDown(heap, 0, heap_garbage, i)
      until i = 0
      delocal int i = 0
```