Γεώργιος Γιαλούρης
2019030063

Δυαδικό Δέντρο Ερευνάς με χρήση πεδίου αριθμών (array) μεγέθους N:
Δημιουργώ μια κλάση StaticBST, η οποία περιέχει έναν δυσδιάστατο πίνακα (array) ακεραίων 3xN. Κάθε θέση του πίνακα παριστάνει έναν κόμβο του δέντρου και σε καθεμία από τις 3 γραμμές αποθηκεύονται τα πεδία INFO (0), LEFT (1), RIGHT (2). Επέλεξα 3xN αντί για Nx3, διότι η java δημιουργεί ένα object για κάθε γραμμή, το οποίο θα καθυστερούσε πολύ την υλοποίηση. Η κλάση StaticBST περιέχει, επίσης, μια μεταβλητή root για την ρίζα του δέντρου και μια μεταβλητή avail, η οποία λειτουργεί ως pointer στην επόμενη διαθέσιμη θέση (στήλη) που θα χρησιμοποιηθεί για στην επόμενη εισαγωγή στοιχείου. Το πεδίο RIGHT χρησιμοποιείται επίσης για να υλοποιήσει την στοίβα (stack) με τις διαθέσιμες θέσεις του array. Στον constructor αρχικοποιώ το bst, ώστε να είναι άδειο, θέτοντας σε όλες τις γραμμές το πεδίο INFO ίσο με τον μικρότερο int (ως null), το πεδίο LEFT ίσο με -1 (ως null) και το πεδίο RIGHT να δείχνει στην επόμενη στήλη (αρχικοποίηση στοίβας). Για την insert, αρχικά, ελέγχω αν υπάρχει διαθέσιμη θέση στο δέντρο. Αν δεν υπάρχει root, τότε εισάγω το πρώτο στοιχείο, δηλαδή αναθέτω στο avail την επόμενη ελεύθερη θέση, αναθέτω το κλειδί στο INFO, αλλάζω το RIGHT ώστε να μην έχει ούτε δεξί παιδί και επιστρέφω το root = 0. Αν το κλειδί είναι μικρότερο του τρέχοντος root τότε πάει στο αριστερό υποδέντρο. Ελέγχω αν υπάρχει αριστερό element και αν υπάρχει τότε καλώ αναδρομικά την συνάρτηση insertRecursive με όρισμα το αριστερό element. Αν δεν υπάρχει τότε προσθέτω το κλειδί, δηλαδή το θέτω ως αριστερό παιδί του τρέχοντος root κτλ. Παρόμοια διαδικασία ακολουθώ και στην περίπτωση που το κλειδί είναι μεγαλύτερο του τρέχοντος root. Τέλος, επιστρέφω το root. Για την search, αφού ελέγξω ότι το δέντρο/υποδέντρο δεν είναι άδειο (επιστρέφω false αν είναι), ελέγχω αν το κλειδί είναι στο root και αν είναι επιστρέφω true. Αλλιώς ελέγχω αν το κλειδί είναι μικρότερο του τρέχοντος root ή μεγαλύτερο και καλώ αναδρομικά την search με όρισμα το αριστερό ή το δεξί παιδί του root αντίστοιχα. Για την delete, αρχικά, ελέγχω ότι το δέντρο/υποδέντρο δεν είναι άδειο και αν είναι, τότε το κλειδί δεν βρέθηκε και επομένως επιστρέφω. Αν το κλειδί είναι μικρότερο ή μεγαλύτερο του τρέχοντος root, τότε καλώ αναδρομικά την deleteRecursive και αναθέτω το αποτέλεσμα της ως αριστερό ή δεξί παιδί του root. Όταν βρεθεί το κλειδί, τότε για την περίπτωση που το element έχει μόνο ένα ή καθόλου παιδιά , ελέγχω πρώτα αν δεν υπάρχει αριστερό παιδί και αν ισχύει επιστρέφω το δεξί παιδί και στη συνέχεια ελέγχω αν δεν υπάρχει δεξί παιδί και αν ισχύει επιστρέφω το αριστερό παιδί. Για την περίπτωση που το element έχει δυο παιδιά, τότε βρίσκω το ελάχιστο από τα κλειδιά του δεξιού υποδέντρου με την getMinKey και το αντιγράφω στο πεδίο INFO του element. Τέλος, καλώ αναδρομικά την deleteRecursive με όρισμα το δεξί παιδί του element, ώστε να διαγράψω το duplicate από το δεξί υποδέντρο.

Δυαδικό Δέντρο Ερευνάς με χρήση δυναμικής παραχώρησης μνήμης:
Το δυναμικό δέντρο υλοποιείται με παρόμοιο τρόπο, αλλά με τις κατάλληλες αλλαγές, ώστε να χρησιμοποιεί nodes αντί για πίνακα.

Μετρήσεις για το Δυαδικό Δέντρο Ερευνάς για τις 2 υλοποιήσεις:

```
-----------------------------------------BINARYSEARCHTREE-----------------------------------------------------
                                  *Average number of comparisons*
Average number of comparisons for StaticBST insertion of 1000000 keys: 95
Average number of comparisons for DynamicBST insertion of 1000000 keys: 96
Average number of comparisons for StaticBST deletion of 100 keys: 98
Average number of comparisons for DynamicBST deletion of 100 keys: 98
                                          *Total time*
Total time for StaticBST insertion of 1000000 keys (in sec): 0.513601
Total time for DynamicBST insertion of 1000000 keys (in sec): 0.560416
Total time for StaticBST deletion of 100 keys (in sec): 0.000598
Total time for DynamicBST deletion of 100 keys (in sec): 0.000399
```

Για την εισαγωγή, παρατηρείται ότι οι δυο υλοποιήσεις έχουν παρόμοιο χρόνο και αριθμό συγκρίσεων. Η στατική υλοποίηση είναι ελάχιστα πιο γρήγορη, γεγονός που το περιμέναμε, αφού δεν χρειάζεται να δεσμεύει χώρο για την κάθε εισαγωγή στοιχείου.
Για την διαγραφή, ενώ οι συγκρίσεις είναι ίδιες (λόγω του παρόμοιου κώδικα), από τους χρόνους προκύπτει ότι η διαγραφή στην δυναμική υλοποίηση είναι πιο γρήγορη, το οποίο είναι λογικό, εξαιτίας της χρήσης δεικτών.

Ουρά Προτεραιότητας με χρήση δυναμικής παραχώρησης μνήμης:

Δημιουργώ μια κλάση DynamicMaxHeap, η οποία περιέχει την static nested κλάση Node, η οποία έχει το κλειδί και τους δείκτες parent, left, right. Στον constructor του Node, θέτω το κλειδί που δίνεται ως όρισμα και θέτω τους δείκτες ως null. Η κλάση DynamicMaxHeap περιέχει επίσης το root και το size του heap. Στον constructor του DynamicMaxHeap, αρχικοποιώ το heap ώστε να είναι άδειο, δηλαδή θέτω το root = null και το size = 0. Για την swap, ανταλλάζω μόνο τα κλειδιά των δυο node που δίνονται. Για να μπορώ να πάρω τον γονέα του πιο δεξιού παιδιού ενός υποδέντρου κάθε φορά, δημιουργώ την getParentRightMost, η οποία μετατρέπει των αριθμό των nodes σε binary, ώστε να πλοηγείτε στο heap δεξιά ή αριστερά ανάλογα με το αν το bit είναι 1 ή 0 αντίστοιχα. Για την insert, αυξάνω το size και παίρνω τον γονέα του μελλοντικού παιδιού με την getParentRightMost. Στην συνέχεια, αφού αναθέσω στον πατέρα το καινούριο node ως δεξί ή αριστερό παιδί ανάλογα, φτιάχνω τον δείκτη parent του καινούριου node. Μετά, με την χρήση μιας while και του swap ανεβάζω το παιδί προς τα πάνω στο heap όσο η τιμή του είναι μεγαλύτερη από τον πατέρα. Για την removeMax, αντικαθιστώ το root με το τελευταίο παιδί, μειώνω το size και καλώ την maxHeapify για το καινούριο root, ώστε το max heap να ξαναποκτήσει την ιδιότητα του. Η maxHeapify ελέγχει αν το κλειδί του node είναι μικρότερο από κάποιο από τα παιδιά και αν είναι, τότε κάνει swap το node με το μεγαλύτερο από τα δυο παιδιά και καλεί αναδρομικά τον εαυτό της για το ίδιο κλειδί. Τέλος, για την κατασκευή του heap όταν τα κλειδιά δίνονται όλα μαζί, χρησιμοποιώ την buildHeap, η οποία χρησιμοποιεί ένα array από nodes και την λογική του στατικού maxheap για τα parent, left και right.

Ουρά Προτεραιότητας με χρήση πεδίου αριθμών (array) μεγέθους capacity:

Η στατική ουρά προτεραιότητας υλοποιείται με παρόμοιο τρόπο, αλλά με τις κατάλληλες αλλαγές, ώστε να χρησιμοποιεί πίνακα αντί για nodes.

Μετρήσεις για την Ουρά Προτεραιότητας για τις 2 υλοποιήσεις:

```
-----------------------------------------------MAXHEAP-----------------------------------------------
                                  *Average number of comparisons*
Average number of comparisons for StaticMaxHeap insertion of 1000000 keys: 11
Average number of comparisons for DynamicMaxHeap insertion of 1000000 keys: 93
Average number of comparisons for StaticMaxHeap deletion of 100 keys: 119
Average number of comparisons for DynamicMaxHeap deletion of 100 keys: 197
                                             *Total time*
Total time for StaticMaxHeap insertion of 1000000 keys (in sec): 0.016111
Total time for DynamicMaxHeap insertion of 1000000 keys (in sec): 0.143282
Total time for StaticMaxHeap deletion of 100 keys (in sec): 0.000701
Total time for DynamicMaxHeap deletion of 100 keys (in sec): 0.000498
Total time for StaticMaxHeap building of 1000000 keys (in sec): 0.019278
Total time for DynamicMaxHeap building of 1000000 keys (in sec): 0.081947
```

Για την εισαγωγή, από τις συγκρίσεις και τον χρόνο παρατηρείται ότι η δυναμική υλοποίηση είναι πολύ πιο αργή, γεγονός που το περιμέναμε, αφού η στατική υλοποίηση δεν χρειάζεται να δεσμεύει χώρο για την κάθε εισαγωγή στοιχείου και επίσης η θέση του πατέρα των παιδιών βρίσκεται με την χρήση της θέσης του στοιχείου στον πίνακα, ενώ στην δυναμική υλοποίηση πρέπει να ξεκινάμε πάντα από το root

Για την διαγραφή, ενώ οι συγκρίσεις είναι περισσότερες στην δυναμική υλοποίηση (λογικό), αυτή εκτελείται σε λιγότερο χρόνο από την στατική.

Τέλος, η κατασκευή του heap όταν τα κλειδιά δίνονται όλα μαζί, είναι πιο γρήγορη στην στατική υλοποίηση, αφού ο κώδικας της δυναμικής υλοποίησης έχει μεγαλύτερη πολυπλοκότητα.