1η Εργασία στις Δομές Δεδομένων
2019-2020
Υπ. Καθηγητής: Ε. Μαρκάκης
Φοιτητές: Πιπιλίκας Ευάγγελος 3180157 / Αλεξόπουλος Νικόλαος 3180004

Υλοποίηση των Interfaces για τα ζητούμενα Α και Γ:

Προτού προβούμε στην υλοποίηση των διεπαφών, δημιουργήσαμε 2 κλάσεις, τις List.java και ListNode.java, ώστε να χρησιμοποιήσουμε κληρονομικότητα και να επιτύχουμε όσο το δυνατόν λιγότερη συγγραφή ίδιου κώδικα.

Class ListNode<T>:

Η ListNode δημιουργεί τους κόμβους μίας απλά συνδεδεμένης λίστας με τα αντικείμενά της να αποτελούνται από τα δεδομένα που θέτουμε κάθε φορά και μία αναφορά σε κάποιο άλλο ListNode (συγκεκριμένα στην περίπτωση μας, μόνο στο επόμενο). Χρησιμοποιήσαμε Generics οπότε ένα ListNode μπορεί να περιέχει διάφορους τύπους δεδομένων ή και αντικείμενα άλλων κλάσεων.

Class List<T>:

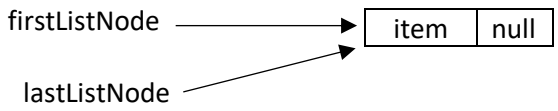
Η List υλοποιεί μία απλά συνδεδεμένη λίστα με χρήση 2 αναφορών στο πρώτο και στο τελευταίο ListNode. Παρέχει μεθόδους για την προσθαφαίρεση των ListNode's από την αρχή και το τέλος της λίστας. Μέσω των αναφορών firstListNode και lastListNode πετυχαίνουμε την προσθαφαίρεση των listNode's από την αρχή ή το τέλος της λίστας σε O(1) (αυτό θα μας φανεί χρήσιμο στο ζητούμενο Α της εργασίας). Επίσης χρησιμοποιούμε τον μετρητή numOfNodes που αυξάνεται κατά 1 σε κάθε προσθήκη listNode στην λίστα ενώ μειώνεται κατά 1 κάθε φορά που αφαιρείται ένα listNode από την λίστα. Έτσι η μέθοδος size δεν χρειάζεται να σαρώσει ολόκληρη την λίστα (O(N)) για να επιστρέψει το μέγεθός της αλλά απλώς επιστρέφει την τιμή του μετρητή (O(1)).

Class StringStackImpl:

Η υλοποίηση της στοίβας γίνεται με χρήση απλά συνδεδεμένης λίστας. Έχοντας υλοποιήσει την κλάση List εκμεταλλευόμαστε με χρήση κληρονομικότητας τις μεθόδους insertAtFront(), removeFromFront() της List για την ανάπτυξη των μεθόδων push() και pop() της στοίβας.

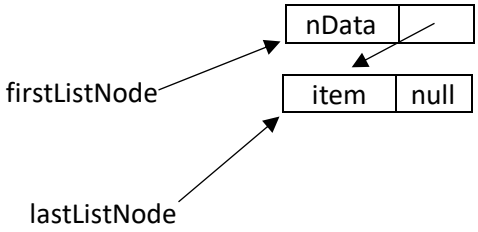
Μέθοδος push(String item):

Περίπτωση: Εισαγωγή κόμβου σε κενή στοίβα.



Περίπτωση: Προσθήκη κόμβου.

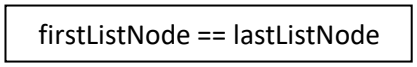
Δημιουργία του κόμβου προς προσθήκη στη στοίβα.
Ο νέος κόμβος "δείχνει" στον προηγούμενο "πρώτο" κόμβο.
Η αναφορά firstListNode "δείχνει" στον καινούριο κόμβο.
Η αναφορά lastListNode δεν μεταβάλλεται.

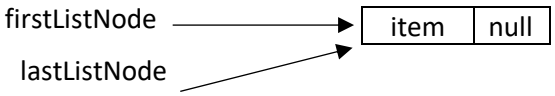


Μέθοδος pop(): Επειδή σε μία στοίβα ισχύει το LIFO, η εισαγωγή και η αφαίρεση από αυτήν γίνονται μόνο από το "πάνω" μέρος της. Έχοντας υλοποιήσει την μέθοδο removeFromFront() στην υπερκλάση List, την χρησιμοποιούμε για την πραγματοποίηση της αφαίρεσης του κόμβου που βρίσκεται στο πάνω μέρος της στοίβας (αντίστοιχα "μπροστά" στην συνδεδεμένη λίστα).

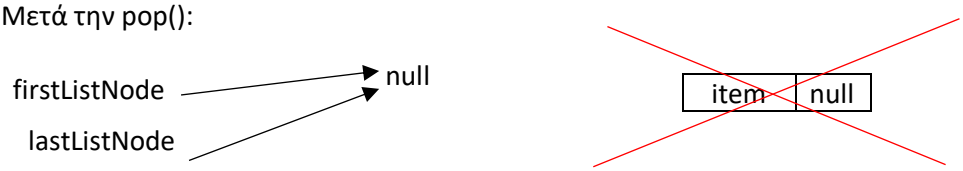
Περίπτωση: pop() όταν υπάρχει μόνο ένας κόμβος.

Αρχική κατάσταση:





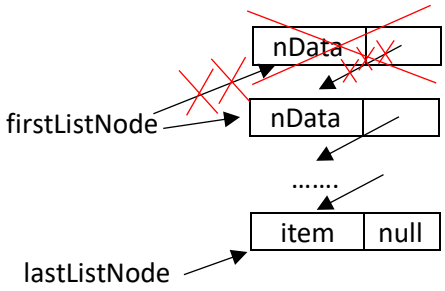
Μετά την pop():



Περίπτωση: pop() όταν υπάρχουν > 2 κόμβοι

Στην περίπτωση αυτήν απλώς θέτουμε την αναφορά firstListNode να "δείχνει" στον επόμενο κόμβο (δηλαδή στον δεύτερο κατά σειρά) με την εντολή firstListNode = firstListNode.getNext().

Πλέον δεν υπάρχουν αναφορές στο μέχρι πρότινος top στοιχείο της στοίβας (αντίστοιχα στο "πρώτο" της λίστας) και άρα αυτό καταστρέφεται από τον garbage collector.



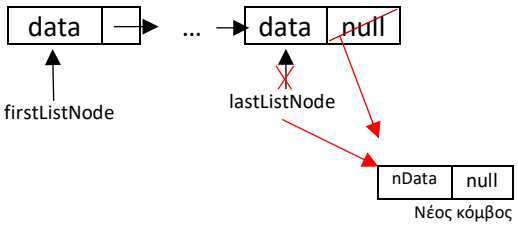
Class StringQueuempl:

Όπως και πριν εκμεταλλευόμαστε τις μεθόδους της υπερκλάσης List για την ανάπτυξη της διεπαφής StringQueue. Αυτή την φορά ισχύει FIFO άρα η εισαγωγή put(String item) χρησιμοποιεί την μέθοδο insertAtBack(T item) της List. Αντίστοιχα η αφαίρεση get() γίνεται με χρήση της μεθόδου removeFromFront της List.

Μέθοδος put(String item):

Σε περίπτωση που η ουρά είναι άδεια τότε απλώς δημιουργείται ο νέος κόμβος στον οποίο "δείχνουν" αμφότερες οι αναφορές lastListNode, firstListNode.

Εάν η ουρά δεν είναι άδεια τότε με την κλήση της put(String item) αρχικά δημιουργείται ένα νέο αντικείμενο της ListNode με περιεχόμενο το item και το οποίο "δείχνει" σε null. Έπειτα θέτουμε στο μέχρι πρότινος τελευταίο ListNode της ουράς να "δείχνει" στο νέο αντικείμενο και τέλος ενημερώνουμε την αναφορά lastListNode να δείχνει στο νέο αντικείμενο.



Μέθοδος get():

Λόγω FIFO η αφαίρεση από την ουρά γίνεται από "μπροστά". Εάν υπάρχει μοναδικός κόμβος στην ουρά τότε θέτουμε αμφότερες τις αναφορές firstListNode, lastListNode null. Διαφορετικά καλούμε την getNext() για την αναφορά firstListNode η οποία τώρα δείχνει στον "δεύτερο" κόμβο. Συνεπώς ο πρώτος καταστρέφεται αφού δεν υπάρχει καμία αναφορά σε αυτόν.

Ζητούμενο Γ:

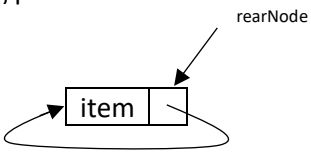
Ανάλογα με το ζητούμενο Α, προτού υλοποιήσουμε την κλάση StringQueueWithOnePointer, χάρη στο δοθέν hint, δημιουργήσαμε την κλάση CircList η οποία με τη σειρά της υλοποιεί μία κυκλική συνδεδεμένη λίστα. Για τους κόμβους της χρησιμοποιούμε ξανά την κλάση ListNode. Η CircList περιέχει τις μεθόδους enqueue(T data), dequeue(), getHead(), getTail() και size() τις οποίες θα κληρονομήσει η υποκλάση StringQueueWithOnePointer. Η διαφορά αυτή την φορά είναι ότι αντί για 2 αναφορές για το πρώτο και τελευταίο αντικείμενο/ κόμβο ListNode της ουράς, χρησιμοποιούμε μόνο 1, την rearNode.

Η ιδέα μας βασίστηκε στο ότι εφόσον έχουμε κυκλική λίστα, ο τελευταίος στην ουρά κόμβος θα δείχνει ξανά στον πρώτο. Η ιδιότητα αυτή είναι που καθιστά εφικτή την εισαγωγή και την αφαίρεση κόμβων από την ουρά σε O(1) αφού έχοντας μία αναφορά στο rearNode με μόνο μία εντολή rearNode.getNext() παίρνουμε το "πρώτο" στοιχείο της ουράς. Ξανά με χρήση των μεθόδων enqueue(T data) και dequeue() της υπερκλάσης CircList υλοποιούμε τις get() και put(String item) της υποκλάσης StringQueueWithOnePointer ως εξής:

Μέθοδος put(String item):

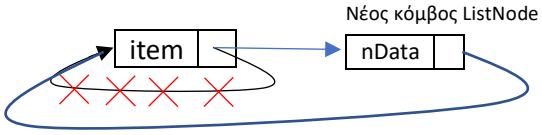
Περίπτωση: Άδεια ουρά.

Δημιουργούμε ένα ListNode με περιεχόμενο το εκάστοτε item το οποίο "δείχνει" στο ίδιο το ListNode. Ενημερώνουμε την αναφορά rearNode να δείχνει σε αυτό (με χρήση της μεθόδου setNext() της κλάσης ListNode) και στην περίπτωση αυτήν, το ListNode αυτό είναι ταυτόχρονα πρώτο και τελευταίο στην ουρά.

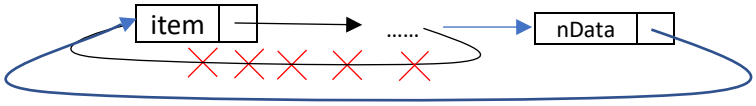


Περίπτωση: Προσθήκη σε ουρά με >=1 κόμβους.

Δημιουργούμε ξανά ένα ListNode με περιεχόμενο το δοθέν item. Αποθηκεύουμε σε μία προσωρινή αναφορά temp την αναφορά του rearNode στον "πρώτο" κόμβο της ουράς (μέσω της getNext() της κλάσης ListNode). Έπειτα στον μέχρι πρότινος τελευταίο κόμβο θέτουμε ως επόμενο το ListNode που μόλις δημιουργήσαμε (μέσω της setNext() της ListNode) και θέτουμε ξανά ως επόμενο του νέου ListNode την temp αναφορά στον πρώτο κόμβο επιτυγχάνοντας την κυκλικότητα. Τέλος ενημερώνουμε την αναφορά rearNode να "δείχνει" στο νέο ListNode.



Γενικότερα:



Μέθοδος get():

Κρατάμε σε μια μεταβλητή τύπου T το περιεχόμενο του πρώτου κόμβου με την εντολή rearNode.getNext().getData(). Η rearNode.getNext() αναφέρεται στον πρώτο κόμβο της ουράς ενώ με την getData() παίρνουμε το περιεχόμενό του το οποίο επιστρέφεται με την κλήση της get(). Έπειτα:

Περίπτωση 1: Η ουρά αποτελείται από >1 κόμβους.

Εκτελούμε την εντολή: rearNode.setNext(rearNode.getNext().getNext()). Πρακτικά σταματά ο rearNode να δείχνει στον πρώτο κόμβο της ουράς (άρα αυτός θα καταστραφει) και καλώντας 2 φορές την getNext() πλεόν δείχνει στον μέχρι πρότινος "δεύτερο" κόμβο της. Ο κόμβος αυτός είναι πλέον ο πρώτος.

Περίπτωση 2: Η ουρά αποτελείται από 1 κόμβο.

Θέτουμε την αναφορά rearNode null.

Περίπτωση 3: Άδεια ουρά → Εξαίρεση.

Ζητούμενο Β:

Για την επίλυση του ζητουμένου Β δημιουργήσαμε την κλάση ThiseasMaze.java. Η κλάση αυτή περιέχει τις απαραίτητες μεθόδους για την ανάγνωση του txt αρχείου που περιέχει τις πληροφορίες του λαβυρίνθου, την κατασκευή ενός Array στο οποίο αποθηκεύεται ο λαβύρινθος του txt αρχείου για ευκολότερη προσπέλαση στα στοιχεία του και την εύρεση μίας εξόδου του, αν αυτή υπάρχει.

Εφόσον ένας λαβύρινθος μπορεί να είναι δομημένος με (πάρα) πολλούς τρόπους, είναι σίγουρο πως υπάρχουν σημεία σε αυτόν στα οποία καλούμαστε να επιλέξουμε ποιο μονοπάτι θα ακολουθήσουμε ενώ υπάρχουν και άλλα. Η επιλογή μας αυτή μπορεί να μην επιφέρει τα επιθυμητά αποτελέσματα και να μας οδηγήσει σε ένα αδιέξοδο. Έτσι πρέπει να επιστρέψουμε στο σημείο που πήραμε την λάθος απόφαση και να πάρουμε μία νέα, από τις υπόλοιπες. Εάν ούτε αυτή είναι η σωστή και βρεθούμε ξανά σε αδιέξοδο, τότε όπως και πριν πρέπει να επιστρέψουμε στο σημείο της λανθασμένης απόφασης και ούτω καθεξής.

Καθώς διασχίζουμε τον λαβύρινθο είναι πιθανό να χρειαστεί να πάρουμε πολλές αποφάσεις την μία μετά την άλλη ενώ και πάλι δεν είμαστε σίγουροι για τις επιλογές μας. Το πρόβλημα αυτό μας οδηγεί στην σκέψη της χρήσης μίας στοίβας στην οποία θα κάνουμε push τις συντεταγμένες μίας τοποθεσίας από την οποία ξεκινούν πολλά διαφορετικά μονοπάτια. Κάθε φορά που οδηγούμαστε σε αδιέξοδο κάνουμε pop από την στοίβα και επανερχόμαστε στο τελευταίο σημείο από το οποίο πήραμε μία λάθος απόφαση.

Λόγω του ότι δεν μπορούμε να είμαστε σίγουροι για το πόσες ωθήσεις και απωθήσεις θα κάνουμε από την στοίβα και επειδή η στοίβα που υλοποιήσαμε στο Α μέρος της εργασίας γίνεται με χρήση συνδεδεμένης λίστας (άρα δεν περιοριζόμαστε στον αριθμό ωθήσεων) καθίσταται ιδανική για τον σκοπό της εύρεσης μίας εξόδου από τον λαβύρινθο.

Λίγα λόγια για τον κώδικά μας:

Αρχικά, στην κλάση Thiseas που περιέχει την μέθοδο main, δημιουργούμε ένα νέο αντικείμενο τύπου ThiseasMaze. Κατόπιν καλούμε τον κατασκευαστή της κλάσης ThiseasMaze ο οποίος δέχεται ως όρισμα το filepath του txt αρχείου που δίνει ο χρήστης μέσω του Command Line. Ο κατασκευαστής διασχίζει το txt αρχείο, εφόσον το filepath είναι σωστό, και δημιουργεί ένα δισδιάστατο Array με τον λαβύρινθο.
Έπειτα καλούμε την μέθοδο setG() η οποία σαρώνει τα στοιχεία της πρώτης και τελευταίας γραμμής όπως και της πρώτης και τελευταίας στήλης του Array και αντικαθιστά με τον χαρακτήρα 'G' (**G**oal) όσα '0' βρει προς δική μας διευκόλυνση. Σημειώνεται πως η αντικατάσταση με 'G' δεν σημαίνει απαραίτητα πως γίνεται σε πραγματική έξοδο του λαβυρίνθου αφού ένα μηδενικό στις άκρες του πίνακα μπορεί να συνορεύει μόνο με άσσους.

Η κλάση ThiseasMaze περιέχει τις μεθόδους hasUp(int, int), hasDown(int, int), hasLeft(int, int), hasRight(int, int) οι οποίες δέχονται 2 ακεραίους που αναπαριστούν τις συντεταγμένες μίας θέσεις στο Array και ελέγχουν εάν υπάρχουν '0' ή '1' στο επάνω, κάτω, αριστερά, δεξιά κελί του πίνακα, επιστρέφοντας true η false.

Επιπλέον προσθέσαμε δύο μεθόδους, τις checkingE() και checkingForAnotherE(). Η πρώτη ελέγχει αν πράγματι οι δοθείσες συντεταγμένες του txt αρχείου αντιστοιχούν σε κάποιο 'E' ενώ η δεύτερη ελέγχει μήπως υπάρχουν πολλαπλά 'E' στον λαβύρινθο. Εάν κάποια από τις δύο εντοπίσει κάποια ανωμαλία στα δεδομένα τότε το πρόγραμμά μας δεν θα προχωρήσει σε εύρεση εξόδου του λαβυρίνθου.

<u>Λίγα λόγια για την μέθοδο solveMaze():</u>
Αρχικά δημιουργούμε ένα αντικείμενο τύπου StringStackImpl που αποτελεί την στοίβα στην οποία θα αποθηκεύουμε με μορφή String τις συντεταγμένες κομβικών σημείων του λαβυρίνθου. Το ονομάζουμε conjunctionsStack.

Δημιουργούμε το μονοδιάστατο Array 2 θέσεων με όνομα location στου οποίου την $1^η$ θέση αποθηκεύουμε την γραμμή στην οποία βρισκόμαστε κάθε φορά και στην $2^η$ θέση την στήλη.

Καλούμε τις checkingE() και checkingForAnotherE() και εάν δεν υπάρχει κάποιο πρόβλημα με τα δεδομένα τότε:

Επαναληπτικά:

- Ελέγχουμε τους "γείτονες" της τρέχουσας τοποθεσίας μας και αναθέτουμε στις boolean μεταβλητές up, down, left, right τιμές true ή false αν υπάρχουν '0' ή '1' γύρω της. Ο έλεγχος επιτελείται από τις προαναφερθείσες μεθόδους hasUp() κτλ.
- Σε μία int μεταβλητή sum κρατάμε το πλήθος των '0' στα γειτονικά κελιά της τρέχουσας τοποθεσίας μας.
- Δημιουργούμε το String location_str στο οποίο αναθέτουμε τις συντεταγμένες της τρέχουσας τοποθεσίας στο Array.
- Αν το sum είναι 1 τότε μπορούμε να κατευθυνθούμε μόνο προς μία κατεύθυνση. Προτού μετακινηθούμε ελέγχουμε εάν βρισκόμαστε ήδη σε κάποια πιθανή έξοδο (η οποία ενδέχεται να έχει μόνο ένα γειτονικό '0') καλώντας την συνάρτηση isG(int i, int j). Εάν η τοποθεσία μας είναι έξοδος ενημερώνουμε την boolean μεταβλητή flag η οποία επιτρέπει την μετακίνηση εντός του λαβυρίνθου. Εάν όμως δεν βρισκόμαστε σε κάποια έξοδο θέτουμε ως 'V' την τρέχουσα τοποθεσία ώστε να γνωρίζουμε εάν την έχουμε ήδη επισκεφθεί. Έπειτα μετακινούμαστε προς την διαθέσιμη κατεύθυνση αλλάζοντας τα περιεχόμενα του location Array.
- Αν το sum > 1 τότε υπάρχουν πολλά μονοπάτια προς τα οποία μπορούμε να κατευθυνθούμε και άρα ωθούμε το location_str στην conjunctionsStack. Σε τέτοιες περιπτώσεις η κατεύθυνσή που ακολουθάμε είναι πάντα, αν και όσο μπορούμε, προς τα πάνω έπειτα όσο μπορούμε προς τα κάτω έπειτα προς τα αριστερά και τέλος προς τα δεξιά.
- Αν το sum είναι 0 και η τρέχουσα τοποθεσία είναι 'G' τότε έχει βρεθεί επιτυχώς μία έξοδος την οποία τυπώνουμε και τερματίζεται το πρόγραμμα.
- Αν το sum είναι 0 και η τρέχουσα τοποθεσία ΔΕΝ είναι 'G' τότε αν υπάρχουν τοποθεσίες στις οποίες μπορούμε να κάνουμε backtrack, θέτουμε 'V' την τρέχουσα τοποθεσία και κάνουμε backtrack. Αν ΔΕΝ υπάρχουν τοποθεσίες για backtrack, τότε ο λαβύρινθος δεν έχει λύση. Ενημερώνουμε τον χρήστη και το πρόγραμμα τερματίζεται.