

# Δομές Δεδομένων: Εργασία 1

ΑΠΟ ΒΑΣΙΛΗΣ ΠΑΠΑΔΗΜΑΣ (3220150) & ΜΑΡΙΟΣ ΜΑΤΣΑ (3220120)

## Μέρος Α: υλοποίηση της διεπαφής StringDoubleEndedQueue

Στο αρχείο `StringDoubleEndedQueueImpl.java` έχουμε υλοποιήσει την διεπαφή `StringDoubleEndedQueue` χρησιμοποιώντας μια διπλά συνδεδεμένη λίστα με generics. Συγκεκριμένα, στο αρχείο `Node.java` έχουμε υλοποιήσει τους κόμβους που χρησιμοποιούμε στην λίστα ως εξής:

```
public class Node<T> {  
    protected T data;  
    protected Node<T> next = null;  
    protected Node<T> previous = null;  
}
```

Δηλαδή, οι κόμβοι μας περιέχουν δείκτες στους προηγούμενους και επόμενους κόμβους (double ended) και έχουν δεδομένα `data` παραμετρικού τύπου. Οι μέθοδοι `addFirst`, `removeFirst`, `addLast`, `removeLast`, `getFirst`, `getLast` και `size` έχουν υλοποιηθεί με χρονική πολυπλοκότητα  $O(1)$ , καθώς:

- οι `getFirst` και `getLast` καλούν την μέθοδο `getData` ενός `Node` (του προηγούμενου ή επόμενου στην `Queue`), το οποίο επιστρέφει το πεδίο `data` του σε  $O(1)$
- η `size()` επίσης επιστρέφει το πεδίο `size` της `StringDoubleEndedQueueImpl` σε  $O(1)$
- οι `addFirst` και `addLast` δημιουργούν ένα `Node` (ο αντίστοιχος constructor κάνει μόνο την `this.data = data` σε  $O(1)$ , μετά καλούν την `isEmpty()` της `StringDoubleEndedQueueImpl`, δηλαδή `return head == null` άρα  $O(1)$  και τέλος καλούν την `setNext()` μιας `Node`, δηλαδή `this.next = next` άρα  $O(1)$ . Επίσης, εκτελούν έναν σταθερό αριθμό assignments, άρα  $O(1)$ . Άμα αθροίσουμε όλα τα μέρη αυτών των μεθόδων, το αποτέλεσμα είναι  $O(1)$ .
- οι `removeFirst` και `removeLast` καλούν επίσης την `isEmpty()` και μετά καλούν την μέθοδο `getData` ενός `Node` (του `head` και του `tail` αντίστοιχα). Η `removeLast` καλεί επίσης την `getPrevious` του `tail`. Τότε, αν η `queue` είναι κενή, εκτελούν δύο αναθέσεις και τελειώνουν. Αλλιώς, η `removeFirst` καλεί τις μεθόδους `getNext` και `setPrevious` του `head` και μετά κάνει μια ανάθεση, και η `removeLast` καλεί την μέθοδο `setNext` της προτελευταίας `Node` και μετά κάνει δύο αναθέσεις. Όπως είδαμε προηγουμένως οι μέθοδοι `isEmpty`, `getData` και `setNext` είναι  $O(1)$ . Επίσης, η `getPrevious` (`return previous`) και `setPrevious` (`this.previous = previous`) είναι προφανώς  $O(1)$ . Άρα, αφού όλες αυτές οι πράξεις είναι  $O(1)$ , και οι μέθοδοι είναι  $O(1)$ .

Επίσης, σημειώνουμε ότι οι μέθοδοι που δεν μπορούν να λειτουργήσουν σωστά σε περίπτωση που η λίστα είναι κενή, δηλαδή `removeFirst` και `removeLast`, όντως πετάνε την εξαίρεση `NoSuchElementException`, viz.

```
@Override  
public T removeFirst() throws NoSuchElementException {  
    if (isEmpty()) throw new NoSuchElementException("The list is empty...");  
    ...  
}
```

## Μέρος Β: μετατροπή μεταθεματικής σε ενθεματική μορφή

Στη μέθοδο `main` του `PrefixToInfix.java`, χρησιμοποιούμε έναν `Scanner(System.in)` ώστε να λάβουμε την έκφραση σε μεταθεματική μορφή από τον χρήστη, την οποία την μετατρέπουμε σε ένα `char[]` που μεταβιβάζεται στην μέθοδο `prefixToInfix`.

Στην μέθοδο `prefixToInfix` αρχικά δημιουργούμε ένα κενό `StringDoubleEndedQueueImpl<String>` (έστω *stack*, καθώς έτσι το χρησιμοποιούμε). Τότε, αρχίζοντας από το τελευταίο στοιχείο του `char[]` (`i = expression.length - 1`), είτε το προσθέτουμε στο τέλος της *stack* (εφόσον είναι αριθμός), είτε (εφόσον είναι τελεστής) δημιουργούμε την αντίστοιχη έκφραση ( $x \odot y$ ) όπου  $\odot$  ο τελεστής και  $x, y$  οι δυο τελευταίοι αριθμοί του *stack* και την προσθέτουμε στην θέση τους στο *stack*. Σε κάθε περίπτωση, μειώνουμε το *i* κατά 1 και προχωρούμε στο προηγούμενο στοιχείο της *stack*. Μέσα στο loop επίσης ελέγχουμε σε κάθε επανάληψη αν το στοιχείο είναι τελεστής ή αριθμός: εάν δεν είναι, προειδοποιούμε τον χρήστη και τερματίζουμε την εκτέλεση. Στην περίπτωση που το loop έχει τερματιστεί με περισσότερα από ένα στοιχεία στην *stack*, προφανώς έχει συμβεί σφάλμα (η έκφραση που μας δώθηκε δεν ήταν μαθηματικά έγκυρη ή οι αριθμοί δεν ήταν μονοψήφιοι) άρα προειδοποιούμε τον χρήστη και τερματίζεται η εκτέλεση. Αλλιώς, εκτυπώνουμε το μοναδικό στοιχείο που περιέχει η *stack*, το οποίο είναι η έκφραση που μας δόθηκε σε ενθεματική μορφή. Εάν συμβεί οποιοδήποτε `Exception` κατά την διάρκεια της κλήσης της `prefixToInfix`, την χειριζόμαστε με `catch()` και ενημερώνουμε τον χρήστη.

## Μέρος Γ: συμπληρωματικά παλίνδρομη ακολουθία DNA

Στην μέθοδο `main` του `DNAPalindrome.java`, δημιουργούμε ξανά ένα `StringDoubleEndedQueueImpl`, έστω *q* (από `queue`), αυτή τη φορά με δεδομένα τύπου `Character` (πρακτικά λειτουργεί και αυτό σαν ένα `char[]`). Χρησιμοποιούμε ένα `BufferedReader(new InputStreamReader(System.in))` για να αναγνώσουμε την είσοδο από τον χρήστη. Κάνουμε ένα loop μέχρι να συναντήσουμε τον χαρακτήρα EOF, διαβάζοντας ένα `char` κάθε φορά και εφόσον ελέγξουμε ότι είναι έγκυρο (δηλαδή ένα από τα A, T, C, G), εισάγοντάς το ως το τελευταίο στοιχείο της *q*. Εάν συμβεί κάποιο απρόβλεπτο σφάλμα, το διαχειριζόμαστε με το `catch (IOException e)`.

Αφού διαβάσουμε την είσοδο, πρώτα ελέγχουμε εάν το μέγεθος της *q* (και άρα το μήκος της δεδομένης ακολουθίας DNA) είναι περιττός αριθμός (`if ((q.size() & 1) == 1)`) και αν είναι προειδοποιούμε τον χρήστη και σταματάμε την εκτέλεση, αφού μια τέτοια ακολουθία προφανώς δεν μπορεί να είναι παλίνδρομη (το διάμεσο στοιχείο θα είναι διαφορετικό μετά τον μετασχηματισμό).

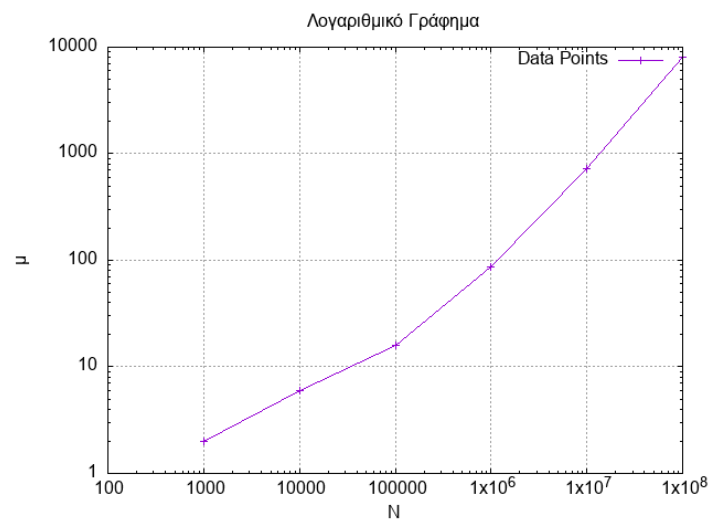
Εάν η ακολουθία είναι άρτιου μήκους (έστω *N*), κάνουμε ένα loop το πολύ  $\log_2(N)$  εκτελέσεων. Σε κάθε εκτέλεση αφαιρούμε το πρώτο και το τελευταίο στοιχείο από τη *q* και τα προσθέτουμε ως `char`. Εάν το άθροισμα ισούται με 138 ή 149, αυτό σημαίνει ότι τα στοιχεία ήταν είτε το A και το T (149), είτε το C και το G (138). Δεν θα μπορούσαν να είναι και τα δύο στοιχεία το ίδιο, ή να μην άνηκαν και τα δύο στο ίδιο σύνολο ( $\{A, T\}, \{C, G\}$ ) καθώς η κωδικοποίηση ASCII δίνει διαφορετικές τιμές στο κάθε σύμβολο (A: 65, T: 84, C: 67, G: 71). Εάν αυτή η ισότητα δεν ισχύει για κάποιο ζευγάρι στοιχείων τότε η ακολουθία προφανώς δεν θα είναι παλίνδρομη, άρα ενημερώνουμε τον χρήστη και σταματάμε την εκτέλεση. Αλλιώς, εάν ολοκληρωθεί το loop χωρίς να έχει προκύψει αυτό το σφάλμα, τότε η ακολουθία είναι παλίνδρομη και ενημερώνουμε τον χρήστη.

Ωστε να επιβεβαιώσουμε ότι η υλοποίησή μας είναι  $O(N)$ , εκτελούμε την εξής εντολή:

```
hyperfine "java DNAPalindrome <<< $(perl -E 'say "AT" x (N/2)')"
```

όπου η εντολή `perl -E 'say "AT" x (N/2)'` (την τρέξαμε μια φορά και χρησιμοποιήσαμε το αποτέλεσμα ώστε να μην τρέχει σε κάθε επανάληψη) δημιουργεί την ακολουθία AT...AT, μήκους *N*. Προφανώς η ακολουθία είναι παλίνδρομη. Επίσης μετρήσαμε τον χρόνο φόρτωσης του JVM και τον αφαιρέσαμε από όλα τα αποτελέσματα. Τα στατιστικά του χρόνου εκτέλεσης για διαφορετικές τιμές του *N* στην ίδια μηχανή δίνονται στον πίνακα παρακάτω:

N	μ (ms)
10 <sup>3</sup>	2
10 <sup>4</sup>	6
10 <sup>5</sup>	16
10 <sup>6</sup>	87
10 <sup>7</sup>	730
10 <sup>8</sup>	8150



Σχήμα 1. Αναπαράσταση του πίνακα (log-log). Προφανώς η πολυπλοκότητα είναι  $O(N)$ .