ΧΑΡΟΚΟΠΕΙΟ ΠΑΝΕΠΙΣΤΗΜΙΟ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ & ΤΗΛΕΜΑΤΙΚΗΣ

Double Ended Queue

Ομάδα: 45 | Κωνσταντίνος Κωτσαράς (it2022050) & Ιουλιανός Πολύζος (it2022091) |

Δομές Δεδομένων | 1η Εργασία | 17-01-2024

Περιεχόμενα

Εισαγωγή

Η παρούσα εργασία στοχεύει στην κατανόηση και υλοποίηση απλών δομών δεδομένων και αλγορίθμων, εστιάζοντας στην υλοποίηση μιας **First-In-First-Out (FIFO)** ουράς με τη χρήση του κυκλικού πίνακα στη γλώσσα προγραμματισμού **Java**. Στο πλαίσιο της εργασίας, προτείνεται η δημιουργία μιας διπλής ουράς, η οποία υλοποιεί το interface **DeQueue<E>**, παρέχοντας λειτουργίες όπως η προσθήκη και αφαίρεση στοιχείων από τις δύο άκρες της ουράς.

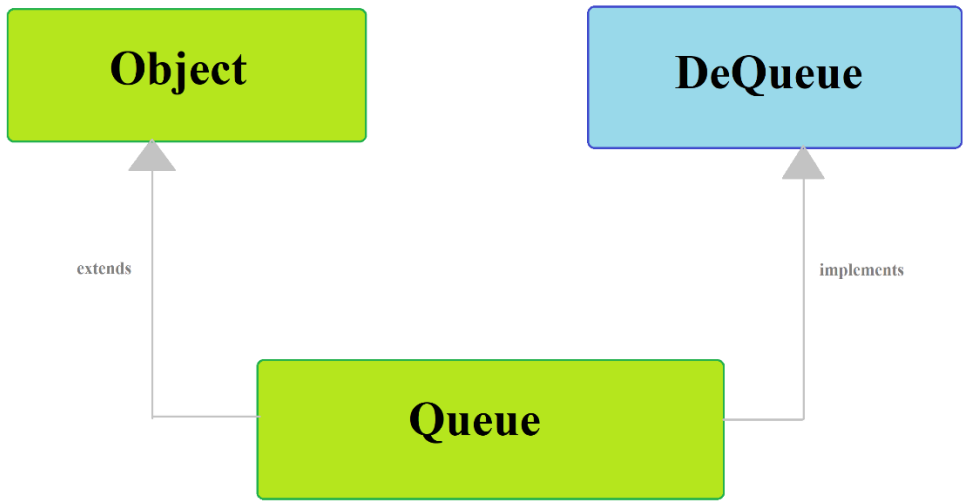
Η υλοποίηση της δομής αυτής βασίζεται στη χρήση ενός **κυκλικού *πίνακα***, όπου η αριθμητική **modulo (mod)** χρησιμοποιείται για τη διαχείριση της περιοδικής ανανέωσης των δεικτών. Ο κυκλικός πίνακας αποτελεί μια αποδοτική και απλή δομή που επιτρέπει την κυκλική κίνηση μέσω του πίνακα χωρίς την ανάγκη διαμόρφωσης ή μεταφοράς στοιχείων.

Η υλοποίηση αυτή προσφέρει μια ευέλικτη και αποτελεσματική λύση για την ανάπτυξη μιας δομής δεδομένων που εκτελεί αποτελεσματικά βασικές λειτουργίες σε ένα περιβάλλον που απαιτεί αποδοτικότητα και ευελιξία.

Στο πλαίσιο αυτό, το κύριο αντικείμενο της υλοποίησης είναι η κατανόηση της λειτουργίας της **FIFO** ουράς και η αντιμετώπιση των προκλήσεων που προκύπτουν κατά την υλοποίηση με τη χρήση κυκλικού πίνακα. Προσβλέπουμε όχι μόνο στη σωστή λειτουργία της υλοποίησης, αλλά και στην κατανόηση των βασικών αρχών των δομών δεδομένων και των αλγορίθμων που τις διαχειρίζονται.

Με την εργασίας αυτή, αναμένουμε να ενισχύσουμε τις γνώσεις μας στον τομέα των δομών δεδομένων και να αναπτύξουμε την ικανότητά μας να εφαρμόζουμε αλγορίθμους σε πρακτικά προβλήματα προγραμματισμού.

Διεπαφές και Κλάσεις



Interface

Η **διεπαφή DeQueue** καθορίζει μια double-ended queue (ουρά με δύο άκρα) και παρέχει τις ακόλουθες βασικές λειτουργίες για τη διαχείριση των στοιχείων της:

<u>Προσθήκη Στοιχείου</u>:

pushFirst(E elem): Προσθήκη ενός στοιχείου στην αρχή της ουράς.

pushLast(E elem): Προσθήκη ενός στοιχείου στο τέλος της ουράς.

Αφαίρεση Στοιχείου:

popFirst() : E: Αφαίρεση και επιστροφή του πρώτου στοιχείου.

popLast() : E: Αφαίρεση και επιστροφή του τελευταίου στοιχείου.

Προσπέλαση Στοιχείων:

first() : E: Επιστροφή του πρώτου στοιχείου.

last() : E: Επιστροφή του τελευταίου στοιχείου.

Έλεγχος Κατάστασης:

isEmpty() : boolean: Έλεγχος εάν η ουρά είναι κενή.

size() : int: Επιστροφή του μεγέθους της ουράς.

Επιπλέον Λειτουργίες:

clear() : void: Καθαρισμός της ουράς.

iterator() : Iterator: Επιστροφή ενός Iterator για προσπέλαση των στοιχείων.

descendingIterator() : Iterator: Επιστροφή ενός Iterator για προσπέλαση των στοιχείων με αντίστροφη σειρά.

Η υλοποίηση της διεπαφής αυτής προσφέρει μια ευέλικτη και αποτελεσματική λύση για τη διαχείριση στοιχείων με διπλή κατεύθυνση, προσφέροντας πλούσιες δυνατότητες υποστήριξης για ευέλικτες δομές δεδομένων.

Κλάσεις

Object

Η κλάση **Object** αποτελεί τη βάση της ιεραρχίας κλάσεων στην Java. Όλες οι κλάσεις, ανεξάρτητα από το πόσο διαφορετικές μπορεί να είναι, κληρονομούν από την κλάση Object. Αυτό σημαίνει ότι η κλάση Object παρέχει βασική λειτουργικότητα που είναι κοινή σε όλες τις κλάσεις στην Java.

Queue

Η κλάση **Queue** υλοποιεί το DeQueue interface, παρέχοντας μια υλοποίηση ενός double-ended queue (ουράς με δύο άκρα) με χρήση κυκλικού πίνακα. Η υλοποίηση αυτή παρέχει σημαντικές λειτουργίες για τη διαχείριση των στοιχείων της ουράς.

Μεταβλητές

Η κλάση Queue χρησιμοποιεί τις παρακάτω βασικές μεταβλητές για την αποθήκευση και διαχείριση των στοιχείων στην ουρά:

● **array:** Αυτός είναι ο πίνακας που χρησιμοποιείται για την αποθήκευση των στοιχείων της ουράς. Κατά την κατασκευή, δεσμεύεται μνήμη για τον πίνακα, και κατά τη διάρκεια της λειτουργίας, τα στοιχεία της ουράς αποθηκεύονται σε αυτόν.

● **f (front):** Ο δείκτης που δείχνει στην πρώτη θέση της ουράς. Καταγράφει όλες τις λειτουργίες, όπως η προσθήκη και η αφαίρεση στοιχείων.

● **r (rear):** Ο δείκτης που δείχνει στο τέλος της ουράς. Καταγράφει όλες τις λειτουργίες, όπως η προσθήκη και η αφαίρεση στοιχείων.

● **size:** Η μεταβλητή που καταγράφει το μέγεθος της ουράς, δηλαδή τον αριθμό των στοιχείων που περιέχει. Ενημερώνεται κατά την προσθήκη ή την αφαίρεση στοιχείων.

Αυτές οι μεταβλητές είναι ουσιώδεις για την λειτουργία της κλάσης Queue και κρατούν την κατάσταση της ουράς καθ' όλη τη διάρκεια του προγράμματος.

Constructor

- Δημιουργούμε έναν νέο πίνακα (array) με προκαθορισμένη χωρητικότητα (DEFAULT_CAPACITY).
- Ορίζετε τους δείκτες f και r στην αρχή του πίνακα (0).
- Θέτετε το μέγεθος (size) της ουράς στο 0.

```
public Queue() {
    array = (E[]) new Object[DEFAULT_CAPACITY];
    f = 0;
    r = 0;
    size = 0;
}
```

Αυτός ο κατασκευαστής είναι υπεύθυνος για την αρχικοποίηση της κλάσης Queue κατά τη δημιουργία μιας νέας στιγμιότυπης της κλάσης.

Υλοποιήσεις των Μεθόδων του Interface

pushFirst(E elem)

1. Εάν ο χώρος είναι πλήρης, καλείται η doubleCapacity() για το διπλασιασμό του μεγέθους του πίνακα.
2. Υπολογίζεται η νέα θέση του front.
3. Προστίθεται το στοιχείο στην αρχή της ουράς.
4. Αυξάνεται το μέγεθος και ενημερώνεται ο μετρητής modCount για τυχόν αλλαγές.

```
@Override
public void pushFirst(E elem) {
    // Check for space
    if (size == getCapacity()) {
        doubleCapacity(); // Double the size of the array if needed
    }

    // Calculate the new position of the front
    f = (f - 1 + getCapacity()) % getCapacity();

    // Add the element to the front of the queue
    array[f] = elem;

    // Increase the size of the queue
    size++;

    // Increase the modification count to track structural modifications
    modCount++;
}
```

pushLast(E elem)

1. Εάν ο χώρος είναι πλήρης, καλείται η doubleCapacity() για το διπλασιασμό του μεγέθους του πίνακα.

2. Το στοιχείο προστίθεται στο τέλος της ουράς.

3. Υπολογίζεται η νέα θέση του rear.

4. Αυξάνεται το μέγεθος και ενημερώνεται ο μετρητής modCount.

```java
@Override
public void pushLast(E elem) {
    // Check for space
    if (size == getCapacity()) {
        doubleCapacity(); // If needed, double the array size
    }

    // Add the element to the end of the queue
    array[r] = elem;

    // Calculate the new position of rear
    r = (r + 1) % getCapacity();

    // Increase the size of the queue
    size++;

    // Increase the modification count to track structural modifications
    modCount++;

}
```

popFirst()

1. Εάν η ουρά είναι άδεια, εμφανίζεται ένα NoSuchElementException.

2. Το στοιχείο αφαιρείται από την αρχή της ουράς και η θέση του front ενημερώνεται.

3.. Ελέγχεται εάν χρειάζεται σμίκρυνση του πίνακα, και αν ναι, καλείται η halfCapacity().

```java
@Override
public E popFirst() {
    if (isEmpty()) {
        throw new NoSuchElementException();
    }

    E elem = array[f];

    // Set the removed element to null
    array[f] = null;

    f = (f + 1) % getCapacity();
    size--;
    modCount++;

    // Check if shrinking is needed
    if (4 * size <= array.length) {
        halfCapacity();
    }

    return elem;
}
```

popLast()

1. Εάν η ουρά είναι άδεια, εμφανίζεται ένα NoSuchElementException.

2. Υπολογίζεται η νέα θέση του rear.

3. Το στοιχείο αφαιρείται από το τέλος της ουράς.

4. Ελέγχεται εάν χρειάζεται σμίκρυνση του πίνακα, και αν ναι, καλείται η halfCapacity().

```
@Override
public E popLast() {
    if (isEmpty()) {
        throw new NoSuchElementException("Queue is empty");
    }

    if (r == 0) {
        r = getCapacity() - 1;
    } else {
        r--;
    }

    E elem = array[r];
    // Set the removed element to null
    array[r] = null;

    size--;
    modCount++;

    // Check if shrinking is needed
    if (4 * size <= array.length) {
        halfCapacity();
    }

    return elem;
}
```

first()

1. Εάν η ουρά είναι άδεια, εμφανίζεται ένα NoSuchElementException.

2. Επιστρέφεται το πρώτο στοιχείο της ουράς.

```
@Override
public E first() {
    if (isEmpty()) {
        throw new NoSuchElementException("Queue is empty");
    }
    return array[f];
}
```

last()

1. Εάν η ουρά είναι άδεια, εμφανίζεται ένα NoSuchElementException.

2. Υπολογίζεται η θέση του τελευταίου στοιχείου και επιστρέφεται το αντίστοιχο στοιχείο.

```
@Override
public E last() {
    if (isEmpty()) {
        throw new NoSuchElementException("Queue is empty");
    }

    // Calculate the position of the last element
    int lastRear = (r - 1 + getCapacity()) % getCapacity();

    // Return the last element
    return array[lastRear];
}
```

isEmpty()

Επιστρέφει true εάν η ουρά είναι άδεια, αλλιώς false.

```
@Override
public boolean isEmpty() {
    return size == 0;
}
```

size()

Επιστρέφει το μέγεθος της ουράς.

```
@Override
public int size() {
    return size;
}
```

clear()

Καθαρίζει την ουρά αναθέτοντας νέο κενό πίνακα και επαναφέροντας τους δείκτες και το μέγεθος.

```
@Override
public void clear() {
    array = (E[]) new Object[DEFAULT_CAPACITY]; // Reset the array to the default capacity
    f = 0; // Reset the front index
    r = 0; // Reset the rear index
    size = 0; // Reset the size of the deque
}
```

iterator()

Επιστρέφει έναν Iterator για την προσπέλαση των στοιχείων.

```
@Override
public Iterator<E> iterator() {
    return new IteratorImpl();
}
```

Η **κλάση IteratorImpl** αντιπροσωπεύει έναν Iterator για την προσπέλαση των στοιχείων της ουράς. Είναι υπεύθυνη για την παρακολούθηση του τρέχοντος δείκτη (cur) κατά την προσπέλαση.

Concurrent Modifications	**Μέθοδος hasNext()**
Η μέθοδος **checkForComodification** χρησιμοποιείται για να ελέγξει αν υπήρξαν δομικές αλλαγές (συρροή) στην ουρά από την τελευταία φορά που δημιουργήθηκε ο Iterator. Εάν υπάρχει συρροή, εμφανίζεται ένα ConcurrentModificationException.	1. Η **hasNext** επιστρέφει true αν υπάρχει άλλο στοιχείο μετά το τρέχον στοιχείο στην ουρά. 2. Πριν επιστρέψει το αποτέλεσμα, καλεί τη μέθοδο checkForComodification για έλεγχο συρροής.

Μέθοδος next()
1. Η next επιστρέφει το επόμενο στοιχείο στη σειρά της ουράς. 2. Πριν επιστρέψει το στοιχείο, καλεί τη μέθοδο checkForComodification για έλεγχο συρροής. 3. Αν δεν υπάρχει επόμενο στοιχείο, εμφανίζεται ένα NoSuchElementException.

descendingIterator()

Επιστρέφει έναν Iterator για την προσπέλαση των στοιχείων με αντίστροφη σειρά.

```
@Override
public Iterator<E> descendingIterator() {
    return new DescendingIteratorImpl();
}
```

Η κλάση **DescendingIteratorImpl** αντιπροσωπεύει έναν Iterator για την προσπέλαση των στοιχείων της ουράς προς τα πίσω (αντίστροφη προσπέλαση).Η DescendingIteratorImpl παρέχει την δυνατότητα να προσπελαστούν τα στοιχεία της ουράς προς τα πίσω, εφαρμόζοντας τους απαραίτητους ελέγχους για τυχόν αλλαγές στην ουρά που έχουν γίνει μεταξύ της δημιουργίας του Iterator και των επόμενων επαναλήψεων.

Constructor DescendingIteratorImpl	**Μέθοδος hasNext()**
Ο κατασκευαστής αρχικοποιεί τον δείκτη cur στον προηγούμενο τελευταίο δείκτη της ουράς. Έλεγχος για συρροή: Ελέγχει αν ο αριθμός των τροποποιήσεων (modifications) στην ουρά αντιστοιχεί με τον αριθμό που περίμενε κατά τη δημιουργία του Iterator. Αν υπάρχει απόκλιση, εμφανίζεται ένα ConcurrentModificationException.	1. Ελέγχει για συρροή πριν από τον έλεγχο για την ύπαρξη επόμενου στοιχείου. 2. Επιστρέφει true αν υπάρχει άλλο στοιχείο πριν από το τρέχον στοιχείο στην ουρά.

Μέθοδος next()
1. Ελέγχει για συρροή πριν από τον έλεγχο για την ύπαρξη επόμενου στοιχείου. 2. Επιστρέφει το προηγούμενο στοιχείο στη σειρά της ουράς. 3. Αν δεν υπάρχει προηγούμενο στοιχείο, εμφανίζει ένα NoSuchElementException.

Επιπλέον, έχουμε τις μεθόδους **doubleCapacity()** και **halfCapacity()**, οι οποίες παίζουν σημαντικό ρόλο στη διαχείριση της χωρητικότητας της ουράς και συμβάλλουν στην αποτελεσματική διαχείριση της μνήμης.

doubleCapacity()	**halfCapacity()**
1. Διπλασιάζει τη χωρητικότητα της ουράς δημιουργώντας ένα νέο πίνακα με διπλάσιο μέγεθος από τον τρέχοντα. 2. Αντιγράφει τα στοιχεία από τον παλιό πίνακα στον νέο, διατηρώντας τη σωστή σειρά. 3. Επαναφέρει τους δείκτες f και r ώστε να δείχνουν στην αρχή και στο τελευταίο στοιχείο της ουράς αντίστοιχα του νέου πίνακα. 4. Ενημερώνει τον πίνακα της ουράς (array) για να δείχνει στον νέο πίνακα.	1. Ελέγχει αν είναι απαραίτητο να μειωθεί η χωρητικότητα της ουράς και αν η νέα χωρητικότητα δεν είναι κάτω από την προεπιλεγμένη χωρητικότητα. 2. Υπολογίζει τη νέα χωρητικότητα ως το μισό της τρέχουσας. 3. Δημιουργεί ένα νέο πίνακα με την ενημερωμένη χωρητικότητα. 4. Αντιγράφει τα στοιχεία από τον παλιό πίνακα στον νέο, διατηρώντας τη σωστή σειρά. 5. Επαναφέρει τους δείκτες f και r ώστε να δείχνουν στην αρχή και στο τελευταίο στοιχείο της ουράς αντίστοιχα του νέου πίνακα. 6. Ενημερώνει τον πίνακα της ουράς (array) για να δείχνει στον νέο πίνακα. 7. Ενημερώνει το μέγεθος της ουράς (size) ώστε να αντιστοιχεί στην ελάχιστη τιμή μεταξύ του τρέχοντος μεγέθους και της νέας χωρητικότητας.

Τέλος, υπάρχει και η μέθοδος **printElements()**, η οποία τυπώνει τα στοιχεία της ουράς.

MAIN

Η **MAIN** είναι δομημένη έτσι ώστε να μην υπάρχει αλληλεπίδραση με τον χρήστη αλλά αντιθέτως του προβάλει βήμα-βήμα τις λειτουργίες, οι οποίες εκτελούνται σειριακά. Το αποτέλεσμα αυτών πράξεων ο χρήστης μπορεί να το δει μέσω της κονσόλας.

```
Push First: 20
[null][null][null][20]
Push First: 10
[null][null][10][20]
Push Last: 30
[30][null][10][20]
Push Last: 40
[30][40][10][20]
Push Last: 50
[10][20][30][40][50][null][null][null]
Printing elements using iterator:
10 20 30 40 50
Pop First Element: 10
[null][20][30][40][50][null][null][null]
Pop Last Element : 50
[null][20][30][40][null][null][null][null]
Printing remaining elements using descending iterator:
40 30 20
First element: 20
Last element: 40
Clear Queue
[null][null][null][null]
Is The Queue Empty? true
```

Ας δούμε ποιες λειτουργίες εκτελούνται:

Αρχικά, κάνουμε **pushFirst** τον αριθμό 20 και τυπώνουμε τον πίνακα.

Στην Συνέχεια ξανά κάνουμε **pushFirst** τον αριθμό 10, με αποτέλεσμα να γίνεται και ο πρώτος αριθμός της ουράς.

Αφού εκτυπωθεί ο πίνακας, συνεχίζουμε με την λειτουργία **pushLast** τον αριθμό 30. Επειδή ο πίνακας είναι κυκλικός και το δεξιότερο σημείο του πίνακα είναι γεμάτο, η νέα προσθήκη θα γίνει στην αρχή του πίνακα.

Έπειτα, εκτελώντας την λειτουργία **pushLast** το 40, θα δούμε ορθά ότι η νέα προσθήκη έγινε δεξιά του 30.

Το Default Capacity έχει οριστεί ως 4, με αποτέλεσμα ο πίνακας να είναι γεμάτος.

Η οποιαδήποτε προσθήκη πρέπει να επιφέρει διπλασιασμό του πίνακα.

Με την προσθήκη **pushLast** 50, βλέπουμε σωστά να διπλασιάζεται ο πίνακας και να προσαρμόζεται η ουρά στον νέο πίνακα.

Χρησιμοποιώντας την μέθοδο του **iterator** παίρνουμε το περιεχόμενο της ουράς με την σειρά.

Στην πορεία κάνουμε **popFirst, popLast** το πρώτο και το τελευταίο στοιχείο της ουράς αντίστοιχα.

Εκτελώντας την **descendingIterator** παίρνουμε τα εναπομένοντα στοιχεία με την αντίστροφη σειρά.

Αφού έγινε η εκτέλεση των παραπάνω με την λειτουργία **first** θα πάρουμε το 20, το οποίο είναι το πρώτο στην ουρά και με την λειτουργία **last** θα πάρουμε το τελευταίο στοιχείο της ουράς που είναι το 40.

Τέλος, με την εκτέλεση της **clear** σβήνουμε οτιδήποτε υπάρχει μέσα στον πίνακα και με αυτόν τον τρόπο ο πίνακας επιστρέφει στο αρχικό του μέγεθος.

Testing

Στον χώρο του προγραμματισμού, τα **tests** αποτελούν ένα ουσιαστικό κομμάτι της διαδικασίας ανάπτυξης λογισμικού, προσφέροντας έναν αξιόπιστο τρόπο επαλήθευσης της σωστής λειτουργίας του κώδικα. Στη γλώσσα προγραμματισμού Java, οι δοκιμές συνήθως υλοποιούνται μέσω ενός εξειδικευμένου πλαισίου δοκιμών, όπως το **JUnit**.

Τα **tests** αποτελούν σενάρια που εκτελούν τμήματα κώδικα με σκοπό την επιβεβαίωση της αναμενόμενης συμπεριφοράς. Χρησιμοποιούνται για την ανίχνευση σφαλμάτων, την εξασφάλιση της συμβατότητας, και τη διασφάλιση ότι αλλαγές στον κώδικα δεν έχουν ανεπιθύμητες επιπτώσεις στην υπάρχουσα λειτουργικότητα.

Το JUnit παρέχει εργαλεία και μεθόδους που επιτρέπουν την οργάνωση και την εκτέλεση δοκιμών, καθώς και αναφορές που βοηθούν στην εντοπισμό των προβλημάτων. Η χρήση των δοκιμών στη Java συμβάλλει στη διασφάλιση υψηλής ποιότητας και αξιοπιστίας του λογισμικού καθώς και στη διευκόλυνση της διαδικασίας συντήρησης και εξέλιξης του κώδικα.

Ομάδα: 45 | Κωνσταντίνος Κωτσαράς (it2022050) & Ιουλιανός Πολύζος (it2022091) |

Δομές Δεδομένων | 1η Εργασία | 17-01-2024

ΤΕΣΤ 1

Στην εκφώνηση της εργασίας υπάρχει ένα test, το οποίο είχαμε δει κατά την διάρκεια του εργαστηρίου και μας ζητήθηκε η προσαρμογή του με βάση την νέα υλοποίηση.

```
@Test
public void testQueue() {
    Queue<Integer> queue = new Queue<>();
    assertTrue(queue.isEmpty());

    int count = 100000;

    for (int i = 0; i < count; i++) {
        queue.pushLast(i);
        assertEquals( expected: i + 1, queue.size());
        assertEquals(Integer.valueOf( i: 0), queue.first());
    }

    int current = 0;
    while (!queue.isEmpty()) {
        assertEquals(Integer.valueOf(current), queue.first());
        assertEquals(Integer.valueOf(current), queue.popFirst());
        current++;
    }

    assertTrue(queue.isEmpty());
}
```

1. Αρχικά, δημιουργείται μια ουρά με ένα αντικείμενο της κλάσης **Queue** για τον έλεγχο των λειτουργιών της.

2. Μετά ελέγχουμε αν η ουρά είναι άδεια.

3. Εισάγει μια σειρά από στοιχεία στην ουρά με τη χρήση της pushLast. Κάθε φορά που προστίθεται ένα στοιχείο, ελέγχεται η αύξηση του μεγέθους της ουράς και το πρώτο στοιχείο της ουράς.

4.Αφαιρεί στοιχεία από την αρχή της ουράς (popFirst) και ελέγχει το πρώτο στοιχείο πριν την αφαίρεση. Ο έλεγχος γίνεται για κάθε στοιχείο που αφαιρείται.

5. Ελέγχει αν η ουρά είναι πλέον κενή μετά την αφαίρεση όλων των στοιχείων.

ΤΕΣΤ 2

Το παραπάνω δοκιμαστικό Test με τη μέθοδο testPushAndPop πραγματοποιεί δοκιμές για να δει την σωστή λειτουργία των μεθόδων pushLast και popFirst της κλάσης Queue.

```
@Test
public void testPushAndPop() {
    Queue<Integer> queue = new Queue<>();
    assertTrue(queue.isEmpty());
    assertEquals( expected: 0, queue.size());

    // Push elements
    queue.pushLast( elem: 1);
    queue.pushLast( elem: 2);
    queue.pushLast( elem: 3);

    assertFalse(queue.isEmpty());
    assertEquals( expected: 3, queue.size());

    // Pop elements
    assertEquals(Integer.valueOf( i: 1), queue.popFirst());
    assertEquals(Integer.valueOf( i: 2), queue.popFirst());

    assertEquals( expected: 1, queue.size());

    // Push more elements
    queue.pushLast( elem: 4);
    queue.pushLast( elem: 5);

    assertEquals(Integer.valueOf( i: 3), queue.popFirst());
    assertEquals(Integer.valueOf( i: 4), queue.popFirst());
    assertEquals(Integer.valueOf( i: 5), queue.popFirst());

    assertTrue(queue.isEmpty());
    assertEquals( expected: 0, queue.size());
}
```

1. Αρχικά, δημιουργείται μια ουρά με ένα αντικείμενο της κλάσης **Queue** για τον έλεγχο των λειτουργιών της.

2. Προσθέτει τρία στοιχεία στην ουρά χρησιμοποιώντας την pushLast και ελέγχει αν η ουρά δεν είναι κενή και το μέγεθος της είναι σωστό.

3. Αφαιρεί δύο στοιχεία από την αρχή της ουράς χρησιμοποιώντας την popFirst και ελέγχει αν το πρώτο στοιχείο που αφαιρέθηκε είναι σωστό καθώς και το μέγεθος της ουράς.

4. Προσθέτει τρία επιπλέον στοιχεία στην ουρά και στη συνέχεια τα αφαιρεί, ελέγχοντας τα σωστά πρώτα στοιχεία που αφαιρούνται και το τελικό μέγεθος της

ΤΕΣΤ 3

Το δοκιμαστικό testIterator ελέγχει τη σωστή λειτουργία του Iterator για τον διαπερασμό των στοιχείων μιας ουράς (Queue<String> στη συγκεκριμένη περίπτωση).

```
@Test
public void testIterator() {
    Queue<String> queue = new Queue<>();
    queue.pushLast( elem: "A");
    queue.pushLast( elem: "B");
    queue.pushLast( elem: "C");

    StringBuilder result = new StringBuilder();
    Iterator<String> iterator = queue.iterator();

    while (iterator.hasNext()) {
        result.append(iterator.next());
    }

    assertEquals( expected: "ABC", result.toString());
}
```

1. Αρχικά, δημιουργείται μια ουρά με ένα αντικείμενο της κλάσης **Queue** για τον έλεγχο των λειτουργιών της.

2. Δημιουργεί έναν Iterator ο οποίος θα διασχίσει όλα τα στοιχεία της ουράς.

3. Πραγματοποιεί έναν βρόχο χρησιμοποιώντας τον Iterator και συγκεντρώνει τα στοιχεία σε ένα StringBuilder. Στο τέλος, ελέγχει αν τα στοιχεία που προσπέλασε το Iterator συμφωνούν με τη σειρά "ABC".

ΤΕΣΤ 4

Το δοκιμαστικό testDescendingIterator ελέγχει τη σωστή λειτουργία του DescendingIterator για τον διαπερασμό των στοιχείων μιας ουράς με αντίστροφη σειρά.

```
 @Test
public void testDescendingIterator() {
    Queue<String> queue = new Queue<>();
    queue.pushLast( elem: "C");
    queue.pushLast( elem: "B");
    queue.pushLast( elem: "A");

    StringBuilder result = new StringBuilder();
    Iterator<String> descendingIterator = queue.descendingIterator();

    while (descendingIterator.hasNext()) {
        result.append(descendingIterator.next());
    }

    assertEquals( expected: "ABC", result.toString());
}
```

1. Αρχικά, δημιουργείται μια ουρά με ένα αντικείμενο της κλάσης **Queue** για τον έλεγχο των λειτουργιών της.

2. Δημιουργεί ένα DescendingIterator για τη διαπέραση των στοιχείων της ουράς με φθίνουσα σειρά.

3. Πραγματοποιεί έναν βρόχο χρησιμοποιώντας τον Descending Iterator και συγκεντρώνει τα στοιχεία σε ένα StringBuilder. Στο τέλος, ελέγχει αν τα στοιχεία που προσπέλασε το Iterator συμφωνούν με τη φθίνουσα σειρά "ABC".

ΤΕΣΤ 5

Το τεστ testDoubleCapacity έχει ως στόχο να επιβεβαιώσει τη σωστή λειτουργία της μεθόδου doubleCapacity στην κλάση Queue.

```
@Test
public void testDoubleCapacity() {
    Queue<Integer> queue = new Queue<>();

    for (int i = 0; i < queue.getCapacity(); i++) {
        queue.pushLast(i);
    }

    assertEquals(Queue.DEFAULT_CAPACITY, queue.getCapacity());

    queue.pushLast( elem: queue.getCapacity() + 1);

    assertEquals( expected: Queue.DEFAULT_CAPACITY * 2, queue.getCapacity());

    queue.clear();

    assertTrue(queue.isEmpty());
}
```

1. Αρχικά, δημιουργείται μια ουρά με ένα αντικείμενο της κλάσης **Queue** για τον έλεγχο των λειτουργιών της.

2. Επαναλαμβάνει την εισαγωγή στοιχείων στην ουρά μέχρι να γίνει διπλασιασμός της χωρητικότητας. Αυτό επιτυγχάνεται με τη χρήση της μεθόδου pushLast.

3. Έλεγχος ότι η αρχική χωρητικότητα της ουράς είναι η προκαθορισμένη χωρητικότητα (Queue.DEFAULT_CAPACITY).

4. Προσθήκη ενός επιπλέον στοιχείου για να ενεργοποιήσει τον διπλασιασμό της χωρητικότητας. Στη συνέχεια, έλεγχος ότι η χωρητικότητα έχει διπλασιαστεί.

5. Καθαρισμός της ουράς, ενώ παράλληλα ελέγχει αν η ουρά είναι άδεια μετά τον καθαρισμό.

ΤΕΣΤ 6

Το δοκιμαστικό testHalfCapacity έχει ως στόχο να επιβεβαιώσει τη σωστή λειτουργία της μεθόδου halfCapacity στην κλάση Queue.

```
@Test
public void testHalfCapacity() {
    Queue<Integer> queue = new Queue<>();

    for (int i = 1; i <= Queue.DEFAULT_CAPACITY * 2; i++) {
        queue.pushLast(i);
    }

    assertEquals( expected: Queue.DEFAULT_CAPACITY * 2, queue.getCapacity());

    for (int i = 1; i <= Queue.DEFAULT_CAPACITY * 2; i++) {
        queue.popFirst();
    }

    assertEquals(Queue.DEFAULT_CAPACITY , queue.getCapacity());

    for (int i = 1; i <= Queue.DEFAULT_CAPACITY * 2; i++) {
        queue.pushLast(i);
    }

    int elementsToRemove = (int) Math.ceil(queue.size() * 3.0 / 4.0);

    for (int i = 1; i <= elementsToRemove; i++) {
        queue.popFirst();
    }

    assertEquals(Queue.DEFAULT_CAPACITY , queue.getCapacity());
}
```

1. Αρχικά, δημιουργείται μια ουρά με ένα αντικείμενο της κλάσης **Queue** για τον έλεγχο των λειτουργιών της.

2. Γεμίζει την ουρά με στοιχεία μέχρι να γίνει διπλασιασμός της χωρητικότητας. Αυτό επιτυγχάνεται με τη χρήση της μεθόδου pushLast.

3. Έλεγχος ότι η χωρητικότητα της ουράς έχει διπλασιαστεί.

4. Αφαιρεί τα στοιχεία από την ουρά, ενώ παράλληλα ελέγχει αν ο πίνακας υποδιπλασιάστηκε σωστά.

5. Ξανά γεμίζει την ουρά με στοιχεία και χρησιμοποιεί τη μέθοδο popFirst, μέχρι να βγουν τα ¾ από την ουρά.

6. Έλεγχος αν ο πίνακας υποδιπλασιάστηκε.

ΤΕΣΤ 7

Το δοκιμαστικό testQueueOperations εκτελεί μια σειρά από διαφορετικές λειτουργίες στην κλάση Queue και ελέγχει αν οι λειτουργίες αυτές λειτουργούν σωστά.

```java
@Test
public void testQueueOperations() {
    Queue<Integer> queue = new Queue<>();

    // Test pushFirst and pushLast
    queue.pushFirst( elem: 1);
    queue.pushLast( elem: 2);
    queue.pushFirst( elem: 3);
    queue.pushLast( elem: 4);
    assertEquals( expected: 4, queue.size());

    // Test first and last
    assertEquals(Integer.valueOf( i: 3), queue.first());
    assertEquals(Integer.valueOf( i: 4), queue.last());

    // Test popFirst and popLast
    assertEquals(Integer.valueOf( i: 3), queue.popFirst());
    assertEquals(Integer.valueOf( i: 4), queue.popLast());
    assertEquals( expected: 2, queue.size());

    // Test iterator for forward traversal
    StringBuilder forwardTraversal = new StringBuilder();
    Iterator<Integer> iterator = queue.iterator();
    iterator.forEachRemaining(element -> forwardTraversal.append(element).append(" "));
    assertEquals( expected: "1 2", forwardTraversal.toString().trim());
    // Test iterator for backward traversal
    StringBuilder backwardTraversal = new StringBuilder();
    Iterator<Integer> descendingIterator = queue.descendingIterator();
    descendingIterator.forEachRemaining(element -> backwardTraversal.append(element).append(" "));
    assertEquals( expected: "2 1", backwardTraversal.toString().trim());

    // Test clear
    queue.clear();
    assertTrue(queue.isEmpty());
    assertEquals( expected: 0, queue.size());
}
```

1. Αρχικά, δημιουργείται μια ουρά με ένα αντικείμενο της κλάσης **Queue** για τον έλεγχο των λειτουργιών της.

2. Προσθέτει στοιχεία στην ουρά χρησιμοποιώντας τις μεθόδους pushFirst και pushLast.

3. Ελέγχει το μέγεθος της ουράς και τα πρώτο και τελευταίο στοιχείο.

4. Αφαιρεί στοιχεία από την ουρά χρησιμοποιώντας τις μεθόδους popFirst και popLast.

5. Εκτελεί επανάληψη με χρήση των iterators για προς τα μπρος και προς τα πίσω, ελέγχοντας τα αποτελέσματα.

6. Καθαρίζει την ουρά και ελέγχει αν είναι άδεια μετά τον καθαρισμό.

ΤΕΣΤ 8

```java
@Test
public void testNoSuchElementException() {
    // Create a new instance of the Queue.
    Queue<Integer> queue = new Queue<>();

    try {
        // Attempt to perform an operation that should throw a NoSuchElementException.
        Integer element = queue.popFirst();

        // If the above line does not throw an exception, fail the test.
        assertTrue( message: "Expected NoSuchElementException, but no exception was thrown", condition: false);
    } catch (NoSuchElementException e) {
        // The exception was thrown as expected.
        assertTrue( condition: true);
    }
}
```

Στο παραπάνω test ελέγχουμε τη σωστή λειτουργία της εξαίρεσης **NoSuchElementException** όταν εκτελείται η popFirst μέθοδος σε μια κενή ουρά.

Τέτοιου είδους εξαιρέσεις έχουμε όταν η ουρά είναι άδεια και καλούμε μεθόδους όπως η popFirst(), popLast(), last() και first().

Στο συγκεκριμένο test όταν εκτελεστεί η λειτουργία popFirst στην κενή ουρά και δεν πυροδοτήσει την NoSuchElementException, τότε το test αποτυγχάνει.

Εάν η popFirst πυροδοτήσει επιτυχώς την NoSuchElementException, τότε το catch block εκτελείται, και το test επιτυγχάνεται με τη χρήση της assertTrue(true).

Βιβλιοθήκες

Στην Java οι βιβλιοθήκες είναι σύνολα προκαθορισμένων κώδικα που παρέχουν χρήσιμες λειτουργίες, κλάσεις και μεθόδους που μπορούν να χρησιμοποιηθούν από προγραμματιστές για την υλοποίηση διάφορων λειτουργιών στις εφαρμογές τους. Οι βιβλιοθήκες καθιστούν ευκολότερη την ανάπτυξη και τη διαχείριση κώδικα, καθώς παρέχουν έτοιμες υλοποιήσεις για κοινές λειτουργίες.

Στον κώδικά μας κάνουμε χρήση των εξής βιβλιοθηκών:

- **java.util:** Βιβλιοθήκη παρέχει κλάσεις και διεπαφές για τη διαχείριση δομών δεδομένων και άλλων χρήσιμων λειτουργιών στην Java.

- **org.junit:** Βιβλιοθήκη που παρέχει το πλαίσιο ελέγχου μονάδας JUnit, χρησιμοποιούμενο για την εκτέλεση αυτόματων δοκιμών σε εφαρμογές Java. Περιλαμβάνει μέθοδους ελέγχου (assertions) για τον έλεγχο του κώδικα.

- **org.junit.Test:** Αυτό είναι ένα annotation που χρησιμοποιείται στο πλαίσιο του JUnit για να σημειώσει ότι ένας καθορισμένος μέθοδος είναι ένα test case. Χρησιμοποιείται σε συνδυασμό με τον JUnit testing framework.

- **org.junit.Assert:** Η κλάση Assert παρέχει σύνολο από μεθόδους επικύρωσης (assertion methods) που χρησιμοποιούνται στα JUnit test cases για να ελέγξουν την εγκυρότητα των αποτελεσμάτων.

Οδηγίες Εκτέλεσης

Για να εκτελέσουμε (►) την **Main** ακολουθούμε τα παρακάτω βήματα:

1.Ανοίγουμε ένα τερματικό και πηγαίνουμε στο φάκελο του project που περιέχει το αρχείο **pom.xml**.

2.Γράφουμε στο τερματικό την εντολή **mvn clean install.** Δημιουργείται το αρχείο **jar.**

```
C:\Users\user\Desktop\Dequeue>mvn clean install
```

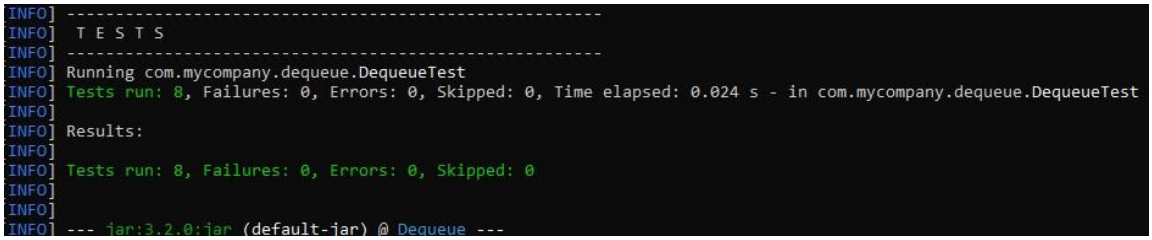
3. Έπειτα πηγαίνουμε στο φάκελο **target** που δημιουργήθηκε το αρχείο **jar.**

```
C:\Users\user\Desktop\Dequeue>cd target
```

4. Εκτελούμε την εντολή **java -jar Dequeue-1.0-SNAPSHOT.jar**

```
C:\Users\user\Desktop\Dequeue\target>java -jar Dequeue-1.0-SNAPSHOT.jar
```

*Επιπλέον κατά την εκτέλεση αυτών των εντολών στο τερματικό, αυτόματα ελέγχονται και τα Test.



Σχόλια και Συμπεράσματα 

Κατά τη διάρκεια της υλοποίησης της δομής δεδομένων Dequeue και των συναρτήσεών της, αντιμετωπίσαμε με επιτυχία διάφορες προκλήσεις. Η διαδικασία αυτή βοήθησε στην κατανόηση πιο συγκεκριμένων θεμάτων, όπως η διαχείριση χωρητικότητας, η υλοποίηση επαναληπτών για προς τα μπρος και προς τα πίσω, καθώς και ο έλεγχος συνθηκών σφάλματος όπως οι εξαιρέσεις ConcurrentModificationException και NoSuchElementException. Πιθανές επεκτάσεις για το μέλλον θα μπορούσαν να περιλαμβάνουν την υλοποίηση επιπλέον συναρτήσεων, όπως η αναζήτηση συγκεκριμένων στοιχείων ή η υποστήριξη για διάφορες τύπους δεδομένων.

Στην ομάδα μας, η εργασία αυτή αναδείχθηκε ιδιαίτερα ενδιαφέρουσα, καθώς μας προσέφερε τη δυνατότητα να αναζητήσουμε και να εμβαθύνουμε σε πληροφορίες σχετικά με τις δομές δεδομένων. Οι αναζητήσεις αυτές βοήθησαν στην καλύτερη κατανόηση των αρχών και των αλγορίθμων που σχετίζονται με τις ουρές διπλής άκρης (Dequeue). Η εμβάθυνση σε αυτά τα θέματα μας επέτρεψε όχι μόνο να υλοποιήσουμε μια λειτουργική Dequeue, αλλά και να αναπτύξουμε περαιτέρω δεξιότητες στον προγραμματισμό Java. Συνοψίζοντας, η ισομοιρασμένη κατανομή εργασιών και το συνεργατικό πνεύμα της ομάδας δημιούργησαν ένα θετικό περιβάλλον που ευνόησε τόσο την ομαδική επίτευξη των στόχων όσο και την ατομική εξέλιξη των μελών.

Πηγές

Ιστοσελίδες που χρησιμοποιήθηκαν για την κατανόηση και την επίλυση της εργασίας:

https://eclass.hua.gr/courses/DIT141/ (Διαφάνειες Μαθήματος)

https://www.youtube.com/@dimitriosmichail1182

https://www.geeksforgeeks.org/queue-data-structure/

https://www.geeksforgeeks.org/implementation-deque-using-circular-array/

https://www.geeksforgeeks.org/circular-array/

IDEs - Integrated Development Environments:

1. NetBeans
2. IntelliJ IDEA
3. Visual Studio Code (README.md)

* Έγινε και χρήση του Maven προκειμένου να δούμε την λειτουργικότητα του κώδικα και να κάνουμε διάφορες δοκιμές με βάση αυτόν.

Για την δημιουργία και την ανάπτυξη του REPORT χρησιμοποιήθηκε το WORD.