Πανεπιστήμιο Πειραιώς – Τμήμα Πληροφορικής

Πρόγραμμα Μεταπτυχιακών Σπουδών

«Πληροφορική»

Μεταπτυχιακή Διατριβή

Τίτλος Διατριβής	**Java Collections Framework - Μελέτη αποδοτικότητας των θεμελιωδών τους λειτουργιών.** **Java Collections Framework - A study on its fundamental operations efficiency.**
Ονοματεπώνυμο Φοιτητή	**Ξενοφών Ζηνοβίου**
Πατρώνυμο	**Χαράλαμπος**
Αριθμός Μητρώου	**ΜΠΠΛ / 16008**
Επιβλέπων	**Ευθύμιος Αλέπης, Επίκουρος Καθηγητής**

Ημερομηνία Παράδοσης **Ιανουάριος 2019**

Τριμελής Εξεταστική Επιτροπή

(υπογραφή)	(υπογραφή)	(υπογραφή)
Ευθύμιος Αλέπης Επίκουρος Καθηγητής	Μαρία Βίρβου Καθηγήτρια	Κωνσταντίνος Πατσάκης Επίκουρος Καθηγητής

Περίληψη

Η παρούσα μεταπτυχιακή διατριβή με τίτλο “Java Collections Framework - Μελέτη αποδοτικότητας των θεμελιωδών τους λειτουργιών” αποτελεί μια μελέτη πάνω στην αποδοτικότητα των βασικών λειτουργιών , των πιο σημαντικών δομών δεδομένων του Java Collections framework. Το Java Collections framework αποτελεί μια συλλογή υλοποιήσεων δομών δεδομένων με βελτιστοποιημένη αποδοτικότητα. Η παρούσα διατριβή θα ασχοληθεί με τη μελέτη των θεμελιωδών λειτουργιών των υλοποιήσεων αυτής της συλλογής και της αποδοτικότητας τους σε θεωρητικό και πρακτικό επίπεδο. Στο πλαίσιο της μελέτης του θεωρητικού επιπέδου θα γίνει εκτενής επισκόπηση της συλλογής αλλά και εννοιών οι οποίες είναι προαπαιτούμενες για την κατανόηση των στόχων και των αποτελεσμάτων, όπως δομές δεδομένων , αλγόριθμοι και πολυπλοκότητα. Σε πρακτικό επίπεδο θα διεξαχθούν πολλαπλά σενάρια benchmarking ανά δομή ώστε να υπάρξουν χρήσιμα συμπεράσματα σε επίπεδο επιμερισμένης πολυπλοκότητας.

Abstract

The presented thesis, entitled “Java Collections Framework - A study on its fundamental operations efficiency” is a study on the efficiency of the fundamental operations , of the Java Collections framework most important data structures. The Java Collections framework is a collection of efficiently optimized data-structure implementations. Fundamental operations of these implementations and their efficiency in both theoretical and practical levels are the main interests of the presented thesis. An extensive review of the Collections framework will be presented, as well as prerequisite terms to the presented thesis’ comprehension, such as data structures , algorithms and complexity. From a practical perspective, multiple benchmarking scenarios will be performed per structure based on the criteria of amortized complexity, in order to produce useful results.

Περιεχόμενα

1. Εισαγωγή

1.1 Εισαγωγή

Η Java αναγνωρίζεται απο το σύνολο της κοινότητας των προγραμματιστών ως μια απο τις κυρίαρχες αντικειμενοστρεφείς γλώσσες προγραμματισμού, με δισεκατομμύρια εγκαταστάσεις παγκοσμίως. Απο το αρχικό της ξεκίνημα στο 1995, έχει περάσει πια σε ένα πολύ υψηλό επίπεδο ωριμότητας , παρέχοντας εξαιρετικά εργαλεία και λύσεις στον προγραμματιστή ως διευκολύνσεις στην προσέγγιση και επίλυση προβλημάτων που καλείται να αντιμετωπίσει. Τα προβλήματα αυτά μπορεί να αφορούν είτε μεγάλα επιχειρησιακά έργα, είτε έργα ερευνητικής φύσεως είτε μικρότερης και πιο καθημερινής φύσεως.

Εικόνα 1 : (Source : oracle.com)

Ένα απο τα σημαντικότερα αυτά εργαλεία θεωρείται το Collections framework, το οποίο αποτελεί ένα τεχνολογικό ανταγωνιστικό πλεονέκτημα της γλώσσας έναντι άλλων αντίστοιχων γλωσσών προγραμματισμού. Προσφέρει έτοιμες προς χρήση δομές δεδομένων (out-of-the-box) μαζί με υλοποιημένες λειτουργίες τους, σε τέτοιο επίπεδο που η αποδοτικότητα τους να είναι η βέλτιστη δυνατή, τόσο σε επίπεδο χρονικής όσο και σε επίπεδο χωρικής πολυπλοκότητας. Η

συλλογή αυτή καλύπτει τις απαιτήσεις της προγραμματιστικής κοινότητας σε τέτοιο βάθος, που σπάνια θα υπάρξει ανάγκη για δημιουργία εξειδικευμένης δομής.

Αντικείμενο της παρούσας διατριβής είναι η μελέτη αποδοτικότητας των πιο σημαντικών δομών αυτής της συλλογής, σε επίπεδο θεμελιωδών λειτουργιών. Αρχικά θα γίνει μια αναφορά σε προαπαιτούμενες θεμελιώδεις έννοιες, όπως δομές δεδομένων , αλγόριθμους και πολυπλοκότητα οι οποίες κρίνονται απαραίτητες για την κατανόηση των στόχων της διατριβής.

Στη συνέχεια, θα αναλυθεί το Collections framework και συγκεκριμένα οι πιο σημαντικές υλοποιήσεις που περιέχονται σε αυτό, μαζί με τις εκτιμήσεις για την απόδοση της κάθε δομής. Στη συνέχεια θα γίνει παρουσίαση του JMH , ένα πολύ αξιόπιστο και ακριβές Benchmark εργαλείο για τη Java και θα τεκμηριωθούν οι λόγοι επιλογής του σε βάρος της κλασικής Benchmark λύσης του JVM. Έπειτα, θα γίνει μια περιγραφή των συνθηκών, των specifications του hardware που χρησιμοποιήθηκε για τα σενάρια , των ρυθμίσεων του JMH και της διαδικασίας Benchmarking που θα ακολουθηθεί. Παράλληλα, θα αποσαφηνιστεί ο τρόπος που θα μετρηθεί η απόδοση των δομών και θα αιτιολογηθεί η πιο πρακτική επιλογή του δείκτη της επιμερισμένης πολυπλοκότητας αντί αυτής της χειρότερης περίπτωσης.

Η διατριβή αυτή θα καταλήξει με το πιο σημαντικό κεφάλαιο : αυτό των αποτελεσμάτων. Εκεί θα παρουσιαστεί κάθε σενάριο εκτέλεσης μαζί με την ερμηνεία και τα συμπεράσματα που τυχόν θα προκύψουν.

1.2 Στόχοι διατριβής

Αρχικός στόχος αυτής της διατριβής είναι η ανάλυση και κατανόηση των σημαντικότερων δομών του Collections framework. Η κατανόηση της υλοποίησης των συγκεκριμένων δομών δεδομένων, είναι σημαντική ώστε να αποκομίσει κάποιος σημαντικά εφόδια γνώσης είτε για την εργασιακή είτε για την ερευνητική του πορεία και αλληλεπίδραση με τη Java.

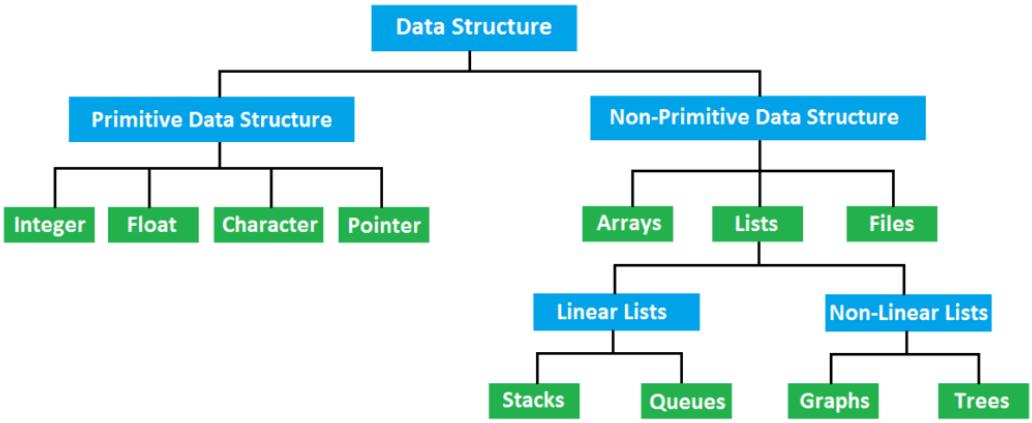
Ο κυρίαρχος στόχος είναι να επιβεβαιωθούν με μετρήσεις σε πραγματικά δεδομένα οι επιδόσεις αυτών των δομών, βάσει της πιο πρακτικής επιμερισμένης πολυπλοκότητας. Σαν αποτέλεσμα θα μπορεί κάποιος να γνωρίζει ότι όχι μόνο στη θεωρία αλλά και στην πράξη η εκάστοτε δομή συμπεριφέρεται, ανταποκρίνεται και επιβεβαιώνει τα όσα υποστηρίζει η Java σχετικά με την απόδοση και την πολυπλοκότητα της. Η γνώση θα αποτελέσει πολύτιμη αρωγή για τον προγραμματιστή, στην επιλογή της δομής με την καταλληλότερη συμπεριφορά και απόδοση για κάθε πρόβλημα που αντιμετωπίζει.

2. Θεμελιώδεις Έννοιες

2.1 Δομές δεδομένων

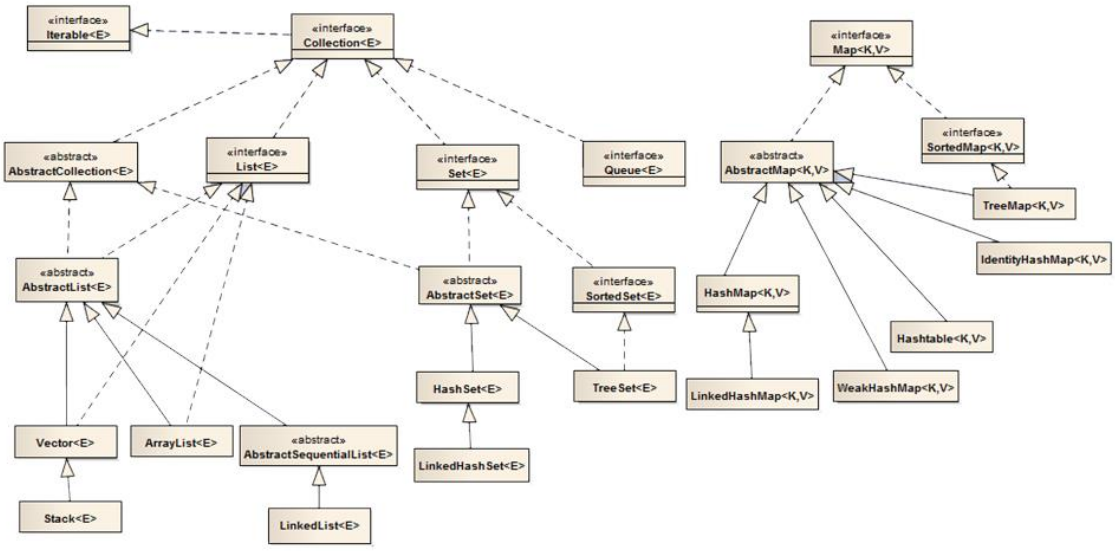
Τα δεδομένα και η συλλογή τους αποτελούν ένα τρόπο αποθήκευσης όλων των πληροφοριών που χρειάζεται κάποιος είτε για να επιλύσει κάποιο πρόβλημα, είτε για να προβεί στη λήψη κάποιων αποφάσεων. Το να υπάρχουν όμως απλώς διαθέσιμες όλες οι πληροφορίες, χωρίς κάποιου είδους οργάνωση δεν διευκολύνει απαραίτητα στην επίλυση του εκάστοτε προβλήματος. Αν για παράδειγμα κάποιος χρειαστεί να αναζητήσει μια πληροφορία πολύ συγκεκριμένη (τον αριθμό μητρώου ενός υπαλλήλου), θα ήταν πολύ χρήσιμο να υπάρχει κάποιος τρόπος να μπορεί να το κάνει γρήγορα και αποτελεσματικά. Επίσης , όλα τα προβλήματα δεν είναι τα ίδια , άρα θα πρέπει και η οργάνωση των δεδομένων να γίνει με τρόπο που να εξυπηρετεί τις ανάγκες του τρόπου επίλυσής τους. Έτσι λοιπόν προκύπτει η ανάγκη για κατάλληλες δομές δεδομένων προσαρμοσμένες στα ζητούμενα κάθε περίπτωσης.

Μια δομή δεδομένων είναι ένας τρόπος με τον οποίο μπορεί να γίνει συλλογή και αποδοτική οργάνωση των δεδομένων, με τέτοιο τρόπο ώστε να καταστεί δυνατό να εκτελούνται λειτουργίες πάνω σε αυτά. Οι λειτουργίες αυτές δεν είναι τίποτε άλλο παρά υπολογιστικοί αλγόριθμοι, ενώ μπορεί να είναι από αρκετά απλές (αναζήτηση και εύρεση στοιχείων, διαγραφή στοιχείων, κλπ.) έως πιο σύνθετες (εύρεση στοιχείου που πληροί συγκεκριμένες συνθήκες, αντικατάσταση στοιχείων με άλλα, κλπ.), εμπεριέχοντας δηλαδή μεγαλύτερη και πιο πολύπλοκη μαθηματική και αλγοριθμική λογική.



Εικόνα 2 : Δομές Δεδομένων στη C++ (Source : https://www.csetutor.com/classification-of-data-structure-with-diagram/)

Οι υλοποιήσεις στις δομές δεδομένων ποικίλουν ανάλογα με τη γλώσσα προγραμματισμού, για παράδειγμα στη C++ ένας αρχικός διαχωρισμός (Εικόνα 2) είναι ανάμεσα σε Primitives (δομές που αποθηκεύονται ακέραιοι, δεκαδικοί, χαρακτήρες και δείκτες) και Non-Primitive (πίνακες , λίστες και αρχεία), με επιπλέον υποκατηγορίες όπως δομές γραμμικού τύπου και μη, ομοιογενών δεδομένων και μη καθώς και δομές αφηρημένου τύπου [1].



Εικόνα 3 : Java Collections Framework (Source : https://www.programering.com/a/MTNwEjMwATU.html)

Στη Java οι Δομές Δεδομένων (Εικόνα 3) που παρέχει η γλώσσα ορίζονται και υλοποιούνται κυρίως από το Collections Framework [2]. Παράλληλα, παρόλο που πολλοί εκτιμούν ότι ίσως ανήκει στα Collections , το Array (πίνακες) δεν συμπεριλαμβάνεται στα Collections, αν και υπάρχει τρόπος, μέσω μεθόδων που παρέχει το framework, για δημιουργία και χρήση του Array. Οι δομές της Java θα αναπτυχθούν περισσότερο στο σχετικό κεφάλαιο (Κεφ. 3). Να σημειωθεί ότι ο εκάστοτε προγραμματιστής έχει τη δυνατότητα να δημιουργήσει τις δικές του δομές, προσαρμοσμένες στις ανάγκες των προβλημάτων που καλείται να επιλύσει. Η Java από την πλευρά της παρέχει αυτό το πλήθος των βελτιστοποιημένων υλοποιήσεων στη διάθεση του, και ενθαρρύνει τον προγραμματιστή να τις προσαρμόσει στις ανάγκες του.

2.2 Αλγόριθμοι

Αλγόριθμος [3] είναι μια υπολογιστική διαδικασία η οποία δέχεται ως είσοδο κάποια τιμή (ακόμα και μηδενική) ή σύνολο τιμών και επιστρέφει ένα αποτέλεσμα. Ένας αλγόριθμος θα πρέπει πάντοτε να πληροί τις ακόλουθες συνθήκες [4] :

1. Finiteness: Να διασφαλίζει ότι πάντα θα τερματίζει, θα έχει δηλαδή πεπερασμένο αριθμό βημάτων, ακόμη και αν αυτός είναι ένας πολύ μεγάλος αριθμός.
2. Definiteness: Κάθε βήμα της διαδικασίας πρέπει να είναι ξεκάθαρα και μονοσήμαντα ορισμένο.
3. Input : Θα πρέπει να έχει κάποια είσοδο (ακόμη και αν αυτή είναι μηδενική)
4. Output : Θα πρέπει πάντα να επιστρέφει τουλάχιστον ένα αποτέλεσμα.
5. Effectiveness: Όλα τα βήματα θα πρέπει να εκτελούνται μέσα σε πεπερασμένο και λογικά αποδεκτό περιθώριο χρόνου, ώστε να μπορεί να θεωρείται αποτελεσματικός.

Οι αλγόριθμοι [5] διακρίνονται σε ντετερμινιστικούς (όταν σε κάθε βήμα υπάρχει μόνο μια επιλογή) και μη-ντετερμινιστικούς (όταν σε κάθε βήμα μπορεί να υπάρχουν περισσότερες από μια επιλογές). Επίσης, δεν είναι πάντοτε απαραίτητο ότι ένας αλγόριθμος που εμπεριέχει πιο απλές πράξεις σε κάθε του βήμα, είναι πιο αποδοτικός από κάποιον που εκτελεί πιο σύνθετες πράξεις. Ένας αναδρομικός αλγόριθμος για παράδειγμα , μπορεί να είναι πολύ πιο απαιτητικός σε χρόνο και μνήμη λόγω των αυξημένων αναδρομικών κλήσεων συναρτήσεων που μπορεί να πραγματοποιεί. Ανάλογα με τη φύση του προβλήματος που αντιμετωπίζει κάποιος, μπορεί να υπάρχουν περισσότεροι του ενός αλγόριθμοι για να επιλέξει. Αποτελεσματικός και

γρήγορος λοιπόν θεωρείται ένας αλγόριθμος όταν χρειάζεται το λιγότερο δυνατό χρόνο και παράλληλα δεσμεύει το λιγότερο δυνατό χώρο στη μνήμη, για να δώσει το σωστό αποτέλεσμα σε σχέση με τους υπόλοιπους.

2.3 Πολυπλοκότητα

Το ερώτημα που γεννάται βάσει των παραπάνω είναι το πως μπορεί κάποιος λοιπόν να αξιολογήσει έναν αλγόριθμο? Πως μπορεί να γνωρίζει αν είναι πιο αποδοτικός, από άλλους για το πρόβλημα που αντιμετωπίζει και πως θα επιλέξει τον πιο κατάλληλο? Η απάντηση δίδεται από την ανάλυση πολυπλοκότητας (complexity) του αλγορίθμου [5]. Η πολυπλοκότητα ενός αλγορίθμου αναφέρεται στις απαιτήσεις του σε υπολογιστικούς πόρους, και χωρίζεται σε δύο κατηγορίες: χωρική και χρονική.

1. Χωρική πολυπλοκότητα (Space complexity): Αναφέρεται στο ποσοστό της μνήμης που θα δεσμεύσει ο αλγόριθμος μέχρι να τερματίσει. Προσοχή, η χωρική πολυπλοκότητα αποτελεί το άθροισμα της μνήμης που απαιτείται από τον αλγόριθμο για την εκτέλεσή του (instruction space) και του βοηθητικού χώρου (auxiliary space) που μπορεί να χρειαστεί κατά την εκτέλεση (runtime).
2. Χρονική πολυπλοκότητα (Time complexity): Αναφέρεται στο χρονικό διάστημα που θα δεσμεύσει την κεντρική μονάδα επεξεργασίας (CPU) του υπολογιστή μέχρι να τερματίσει.

Με την εξέλιξη της τεχνολογίας, το χώρο μνήμης να έχει αυξηθεί σημαντικά και το κόστος κτήσης της του να είναι πολύ χαμηλό, ο κρίσιμος παράγοντας στην απόδοση ενός αλγορίθμου είναι πρωτίστως η χρονική πολυπλοκότητα και δευτερευόντως η χωρική.

Για να μπορέσει κάποιος να υπολογίσει χρονική πολυπλοκότητα ενός αλγορίθμου, θα πρέπει να υπολογίσει για κάθε βήμα που κάνει ο αλγόριθμος, ακριβώς πόσες πράξεις κάνει, πόσο χρόνο χρειάζεται για την κάθε πράξη και τελικά πόσο χρόνο θα χρειαστεί για να δώσει ένα αποτέλεσμα. Αντίστοιχα, για να υπολογίσει τη χωρική πολυπλοκότητα θα πρέπει να υπολογίσει για κάθε βήμα πόσες μεταβλητές απαιτεί ο αλγόριθμος, αν απαιτεί βοηθητικές πόσες είναι αυτές , πόσο χώρο πιάνει η κάθε μια στη μνήμη, πότε αποδεσμεύει τις βοηθητικές (αν τις αποδεσμεύει κατά την εκτέλεση και όχι στο τέλος) και τελικά πόσο χώρο συνολικά θα χρειαστεί Όλες αυτές οι παράμετροι θα πρέπει να εκφραστούν με κάποια συνάρτηση η οποία θα μπορεί να δώσει ακριβές αποτέλεσμα για κάθε είσοδο. Αυτό φαίνεται αρκετά δύσκολο αρχικά. Επίσης, συχνά

υπάρχουν στοιχεία στη συνάρτηση τα οποία μπορεί να επηρεάσουν από λίγο έως σχεδόν καθόλου την έκβαση του αποτελέσματος ανεξάρτητα από την είσοδο.

Έστω για παράδειγμα ότι κάποιος αλγόριθμος έχει χρονική πολυπλοκότητα που εκφράζεται από τη συνάρτηση

$$\boldsymbol{T(n) = n^2 + 2n + 1}$$

Ας δούμε τι θα επιστρέψει η συνάρτηση Τ για πιθανές εισόδους:

Για είσοδο n=1 => $\boldsymbol{T(1) = 1^2 + (2 * 1 + 1) = 1 + 3 = 4}$

Για είσοδο n=2 => $\boldsymbol{T(2) = 2^2 + (2 * 2 + 1) = 4 + 5 = 9}$

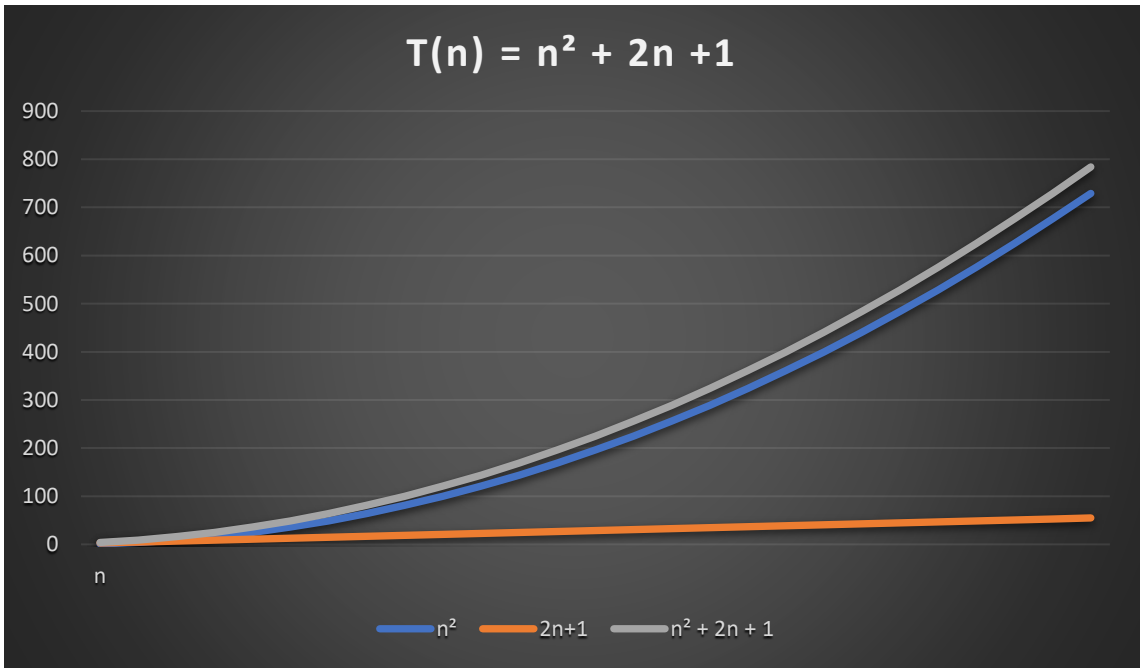
Για είσοδο n=3 => $\boldsymbol{T(3) = 3^2 + (2 * 3 + 1) = 9 + 7 = 16}$

Για είσοδο n=4 => $\boldsymbol{T(4) = 4^2 + (2 * 4 + 1) = 16 + 9 = 25}$

Για είσοδο n=5 => $\boldsymbol{T(5) = 5^2 + (2 * 5 + 1) = 25 + 11 = 36}$

…

Γίνεται εύκολα γίνεται αντιληπτό, από τη γραφική παράσταση της Τ (Εικόνα 3), ότι ο πρώτος όρος της Τ (n2) επηρεάζει σημαντικά το αποτέλεσμα της συνάρτησης. Όσο αυξάνεται το n , τόσο η αύξηση του πρώτου όρου η οποία είναι εκθετική επηρεάζει κυρίαρχα στην αύξηση της Τ, ενώ αντίθετα παρατηρείται ότι ο δεύτερος όρος (2n+1) είναι ελάχιστα σημαντικός ως προς την Τ.



Εικόνα 3 : Η συνάρτηση T(n) = n² + 2n +1

Καθώς όμως είναι δύσκολο να μπορέσει κάποιος να εκφράσει με ακρίβεια τις χρονικές και χωρικές απαιτήσεις ενός αλγορίθμου, χρησιμοποιεί τους ασυμπτωτικούς συμβολισμούς [1, 6].

2.4 Ασυμπτωτικοί συμβολισμοί

Με τους ασυμπτωτικούς συμβολισμούς [1] δεν χρειάζεται να βρεί κανείς ακριβώς το αποτέλεσμα μιας έκφρασης , όταν αυτή πλησιάζει κάποιο πεπερασμένο ή απειροστικό όριο . Αρκεί να υπολογίσει προσεγγιστικά, αγνοώντας τους συντελεστές που επηρεάζουν ελάχιστα την έκφραση, καταλήγοντας σε ένα συντελεστή , ο οποίος επηρεάζει σημαντικά το αποτέλεσμα της έκφρασης, ώστε να μπορεί πια να έχει στη διάθεση του συγκρίσιμα μεγέθη, ανάμεσα στις πολυπλοκότητες των αλγορίθμων.

Διακρίνονται σε Θ , Ο , Ω:

1. Θ (Θήτα ή Theta) : Πολυπλοκότητα μέσης περίπτωσης. Η μέση τιμή εκτέλεσης του

αλγορίθμου, όταν η συνάρτηση είναι άνω και κάτω φραγμένη , από μεγάλες θετικές σταθερές.

2. Ο (Όμικρον ή Ο ή Big-Oh) : Πολυπλοκότητα χειρότερης περίπτωσης (Worst Case), προσεγγιστικά ο χρόνος εκτέλεσης όταν η συνάρτηση είναι άνω φραγμένη από μεγάλη θετική σταθερά.
3. Ω (Ωμέγα ή Omega) : Πολυπλοκότητα καλύτερης περίπτωσης (Best Case) ,προσεγγιστικά όταν η συνάρτηση είναι κάτω φραγμένη από μεγάλη θετική σταθερά.

Για την αξιολόγηση της πολυπλοκότητας επιλέγεται πάντοτε η χειρότερη περίπτωση , καθώς έτσι μπορεί κανείς να είναι βέβαιος για τη συμπεριφορά του αλγορίθμου σε μεγάλα n κάτω από όλες τις συνθήκες.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

Εικόνα 4 : Σύγκριση τάξεων πολυπλοκότητας (Source : Χ. Κωνσταντόπουλος – Αλγόριθμοι, Σημειώσεις Διδασκαλίας)

Μια σύγκριση των τάξεων πολυπλοκότητας (Εικόνα 4) , για χρόνους αλγορίθμων σε επεξεργαστή που εκτελεί 1.000.000 εντολές υψηλού επιπέδου [4].

2.5 Επιμερισμένη ανάλυση & πολυπλοκότητα

Καθώς πολλές φορές στην πράξη η πολυπλοκότητα κάποιας λειτουργίας (ένα σύνολο πράξεων) μπορεί να παρουσιάζει μεγάλες διακυμάνσεις, είναι πολύ σημαντικό να γνωρίζει κάποιος το

επιμερισμένο κόστος ανά πράξη. Δηλαδή για μια λειτουργία ή οποία αποτελεί μια ακολουθία πράξεων , υπολογίζεται το συνολικό κόστος των πράξεων αυτών και επιμερίζεται ανά πράξη. Πρέπει να σημειωθεί εδώ ότι το επιμερισμένο κόστος δεν αποτελεί το μέσο κόστος ανά πράξη, καθώς διαφέρει ο τρόπος υπολογισμού του.

Για παράδειγμα έστω ότι κατά τη διάρκεια ενός έτους (1 έτος = 12 μήνες), για 8 μήνες κάποιος δίνει βάσει συμβολαίου συντήρησης κόστος 50 μονάδες ανά μήνα, για τους επόμενους 3 δίνει 100 μονάδες ανά μήνα και για τον τελευταίο δίνει 200 μονάδες. Προφανώς η χειρότερη περίπτωση είναι η πράξη με κόστος 200 μονάδων. Άρα η ΠΧΠ (πολυπλοκότητα χειρότερης περίπτωσης) υπολογίζεται ως εξής :

$$\mathbf{200 * 12 \ = \ 2400 \ (\boldsymbol{μονάδες\ συνολικά}, 200\ \boldsymbol{ανά\ πράξη})}$$

Η επιμερισμένη ανάλυση αθροίζει το σύνολο του κόστους ανά πράξη και το επιμερίζει στις πράξεις:

$$\frac{(\mathbf{200 * 1}) + (\mathbf{50 * 8}) + (\mathbf{100 * 3})}{\mathbf{12}} = \mathbf{75\ (\boldsymbol{μονάδες\ ανά\ πράξη})}$$

Οι 75 μονάδες ανά πράξη (μήνας) αποτελούν την **επιμερισμένη πολυπλοκότητα** (Amortized complexity) [1, 7] στη συγκεκριμένη περίπτωση. Να σημειωθεί ότι πάντοτε η επιμερισμένη πολυπλοκότητα είναι μικρότερη ή ίση της πολυπλοκότητας χειρότερης περίπτωσης. Ενδεικτικά αναφέρονται , οι τεχνικές της επιμερισμένης ανάλυσης βάσει των οποίων αποδεικνύεται το χαμηλό επιμερισμένο κόστος, παρότι κάποιες πράξεις μεμονωμένα μπορεί να έχουν πολύ υψηλότερο κόστος.

1. Τεχνική του αθροίσματος
2. Τεχνική του λογιστή (ή του τραπεζίτη)
3. Τεχνική του φυσικού

Οι τεχνικές επιμερισμένης ανάλυσης δεν αποτελούν αντικείμενο μελέτης της συγκεκριμένης διατριβής, οπότε και δεν θα γίνει εμβάθυνση σε αυτές.

3. Υλοποιημένες Δομές Δεδομένων στη Java

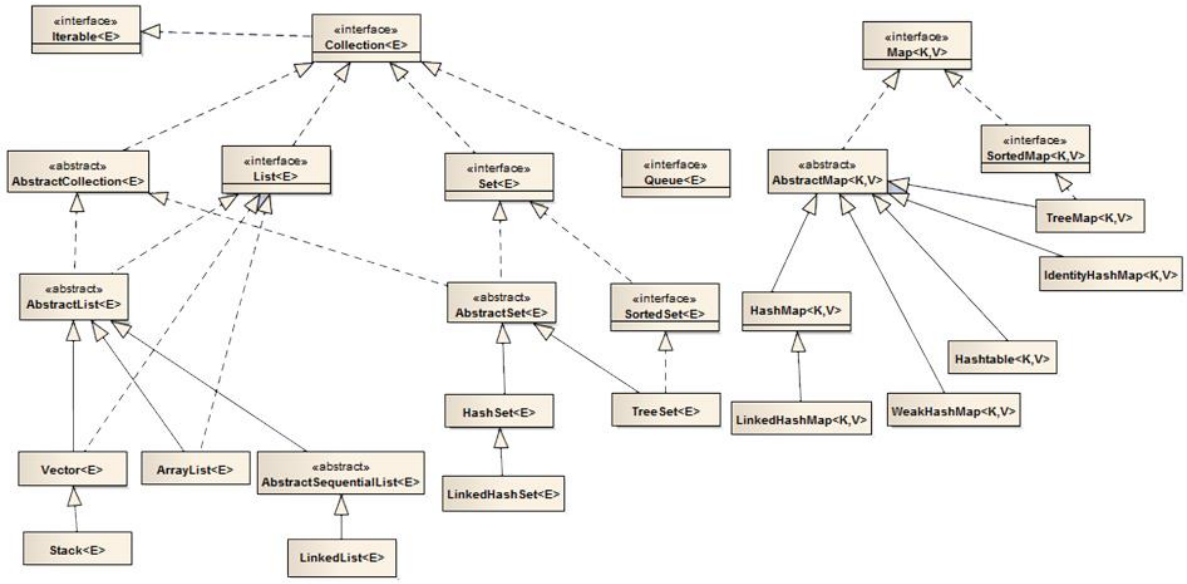
3.1 Java Collections Framework : Επισκόπηση

Το πακέτο java.util αποτελεί ένα πολύ μεγάλο σύνολο από interfaces και κλάσεις, το οποίο υποστηρίζει ένα πολύ μεγάλο εύρος δυνατοτήτων. Ανάμεσα σε όλα τα άλλα, εμπεριέχει ένα από τα πιο ισχυρά υποσυστήματα της Java: Το Collections Framework [8], το οποίο αποτελεί μια πανίσχυρη συλλογή από interfaces , αφηρημένες κλάσεις και υλοποιημένες δομές δεδομένων.

Ιστορικά πρέπει να σημειωθεί ότι το Collections (Εικόνα 5) ενσωματώθηκε στην Java από την έκδοση 1.2 και έπειτα. Προγενέστερα, η Java παρείχε μόλις 4 συνολικά υλοποιήσεις για αντίστοιχη ομαδική διαχείριση δεδομένων: τις κλάσεις Dictionary,Vector,Stack και Properties. Αν και πολύ λίγες , οι κλάσεις αυτές ήταν πάρα πολύ χρήσιμες βάσει των δυνατοτήτων που προσέφεραν. Το μεγάλο κενό όμως , το οποίο ήρθε να καλύψει το Collections framework ήταν ότι η κάθε δομή είχε συγκεκριμένη τρόπο διαχείρισης ως προς τις λειτουργίες και αυτό έκανε τα πράγματα αρκετά περίπλοκα. Τη λύση ήρθε να δώσει η Java με το λανσάρισμα του Collections.

Ακολουθούν συνοπτικά τα βασικά σημεία υπεροχής του framework [8].

1. Το framework υλοποιήθηκε με τέτοιο τρόπο ώστε οι βασικές του δομές (Dynamic Arrays, Linked Lists, Trees & Hash Tables) να είναι υψηλής απόδοσης. Αυτό σημαίνει ότι σπάνια θα μπορέσει κάποιος να υλοποιήσει δική του δομή που να είναι σε επίπεδο επιδόσεων καλύτερη από αυτές.
2. Το framework προσφέρει παράλληλα με τις δομές και κοινές βασικές λειτουργίες χειρισμού, χωρίς να χρειαστεί κανένας μετασχηματισμός και καμία επιπλέον αλλαγή σε επίπεδο κώδικα.
3. Το extending στα interfaces είναι πολύ εύκολο και άμεσο. Ένα επίπεδο πάνω από όλες τις υλοποιήσεις, υπάρχει μια σειρά από interfaces που γίνονται extend από τις κλάσεις που κληρονομούν, οπότε είναι πολύ εύκολο να χρησιμοποιήσει κάποιος συγκεκριμένες λειτουργίες αν το επιθυμεί, επιλέγοντας την αντίστοιχη κλάση ή κάνοντας ο ίδιος extend το interface που εμπεριέχει τη συμπεριφορά που επιθυμεί.
4. Παρότι δεν αναφέρεται κάπου ότι ανήκει στο Collections, υπάρχει μηχανισμός με τον οποίο έχει ενταχθεί και το Array στη συλλογή αυτή.



Εικόνα 5 : Java - The Collections Framework (Source : https://www.programering.com/a/MTNwEjMwATU.html)

Πολύ σημαντικό είναι το γεγονός ότι τα Collections είναι δομές που υποστηρίζουν τύπους δεδομένων μόνο κλάσεις αντικειμένων που κληρονομούν από την κλάση Object (όλες δηλαδή custom και μη) αλλά δεν υποστηρίζουν Primitives τύπους δεδομένων. Φυσικά η χρήση τέτοιων τύπων μπορεί να γίνει με χρήση των wrapper κλάσεων τους (Εικόνα 6).

Primitive Type	Wrapper class
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

Εικόνα 6 : Primitives & Wrapper Classes (Source : https://www.javatpoint.com/wrapper-class-in-java)

Εδώ θα πρέπει να γίνει μια αναφορά στο γεγονός ότι ενώ όλα τα Collections είναι δομές τύπου ομοιογενών δεδομένων, εύκολα μπορούν σε επίπεδο υλοποίησης να γίνουν σφάλματα με την υποστήριξη ετερογενών δεδομένων. Αυτό σημαίνει ότι σε μια δομή μπορεί κανείς να εισάγει και στοιχεία-διαφορετικών κλάσεων! Στο μη σύνηθες παράδειγμα που ακολουθεί (Εικόνα 7) γίνεται εισαγωγή διαφορετικών τύπων δεδομένων και το ArrayList τους δέχεται αδιαμαρτύρητα.

```
import java.util.ArrayList;

/**
 * @author xzinoviou
 */
public class Testing {

    public static void main(String[] args) {


        ArrayList<Object> list = new ArrayList<>();
        list.add(Integer.valueOf(1));
        list.add(Double.valueOf(2.89));
        list.add(Boolean.TRUE);
        list.add(Long.valueOf(10L));
        list.add("This is a string..");

        System.out.println("Type of Structure : " +
                list.getClass().getSimpleName());
        for(Object o : list)
            System.out.println("Value : "  + o +
                    " , Class : " + o.getClass().getSimpleName());


    }
}
```

```
Type of Structure : ArrayList
Value : 1 , Class : Integer
Value : 2.89 , Class : Double
Value : true , Class : Boolean
Value : 10 , Class : Long
Value : This is a string.. , Class : String

Process finished with exit code 0
```

Εικόνα 7 : Εισαγωγή στοιχείων σε ArrayList

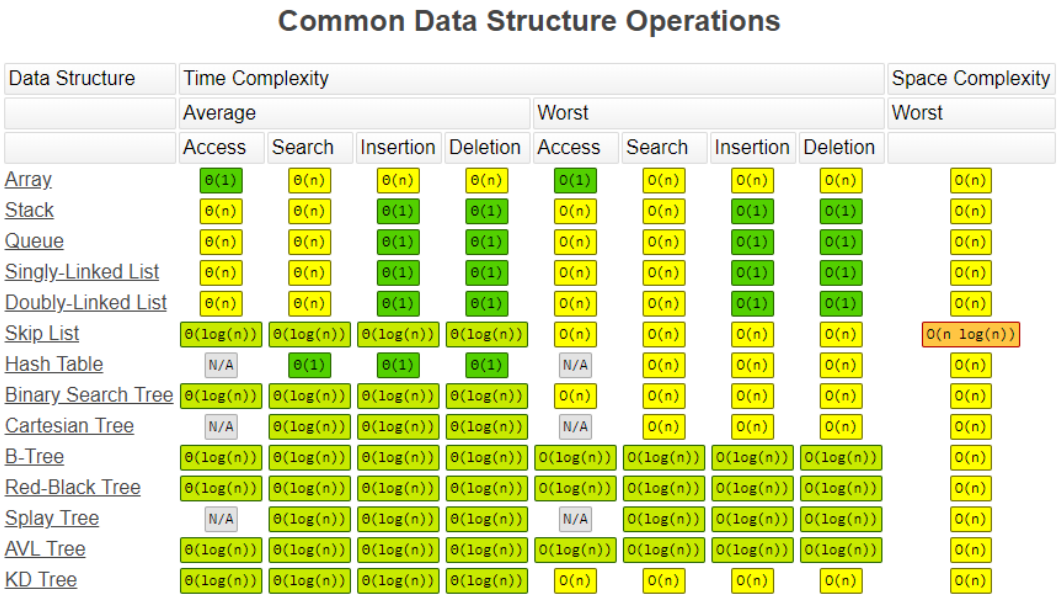
Τι θα συμβεί όμως σε επίπεδο υλοποίησης? Ενώ ο compiler δεν δείχνει να έχει πρόβλημα στο να μεταγλωττίσει το συγκεκριμένο κώδικα, στο runtime κάποιας πιθανής υπολογιστικής λειτουργίας για παράδειγμα θα υπάρξει exception και βίαιος τερματισμός του προγράμματος. Εύκολα γίνεται αντιληπτό πως κάτι τέτοιο δημιουργεί τεράστιο ζήτημα, καθώς θα πρέπει με κάποιο τρόπο να ξέρει κανείς τι τύπο έχει αποθηκεύσει σε κάθε θέση. Συνετό θα είναι λοιπόν να αποφεύγεται αυτή η πρακτική.

3.2 Fundamental Interfaces

Τα θεμελιώδη interfaces, των οποίων οι υλοποιήσεις επεξηγούνται στη συνέχεια:

1. **List** : ορίζει μεθόδους για τη συμπεριφορά σε δομές λίστας. Υλοποιείται από τις κλάσεις ArrayList (ως πίνακας μεταβλητού μήκους) και LinkedList (ως διπλά συνδεδεμένη λίστα).
2. **Set** : ορίζει μεθόδους για τη συμπεριφορά σε δομές συνόλου και υλοποιείται από τις κλάσεις HashSet,LinkedHashSet (ως σύνολα με αντιστοίχιση (hashcodes) – τιμών (values)), TreeSet (ως σύνολο δενδρικής δομής).
3. **Queue** : ορίζει μεθόδους για τη συμπεριφορά σε δομές Ουράς (Δομές τύπου FIFO, First-in First-out, Ουρά FIFO με προτεραιότητες). Υλοποιείται από την κλάση LinkedList. (ως διπλά συνδεδεμένη λίστα).
4. **Deque** : ορίζει μεθόδους για τη συμπεριφορά σε δομές Ουράς με εισαγωγή και εξαγωγή στοιχείων και από τις δύο πλευρές. Υλοποιείται από τις κλάσεις LinkedList (ως διπλά συνδεδεμένη λίστα) , και ArrayDeque (ως πίνακας με μεθόδους για χρήση ουράς).
5. **Map** : ορίζει μεθόδους για τη συμπεριφορά σε δομές αντιστοίχισης πινάκων και τιμών. Υλοποιείται από τις κλάσεις Hashtable,HashMap (ως δομές αντιστοίχισης key-value (κλειδιού-τιμής)).

Ενδεικτικά παρατίθεται πίνακας (Εικόνα 9) στον οποίο αναγράφονται οι πολυπλοκότητες για τις βασικές λειτουργίες σε δομές δεδομένων που χρησιμοποιούνται ευρύτατα από την προγραμματιστική κοινότητα (Εικόνα 8). Με το συμβολισμό <E> (E = Entity , Οντότητα, κάθε κλάση που κληρονομεί από την Object , custom ή μη) αναφέρεται ο τύπος της κλάσης που υποστηρίζει η δομή, πρακτικά όλες .

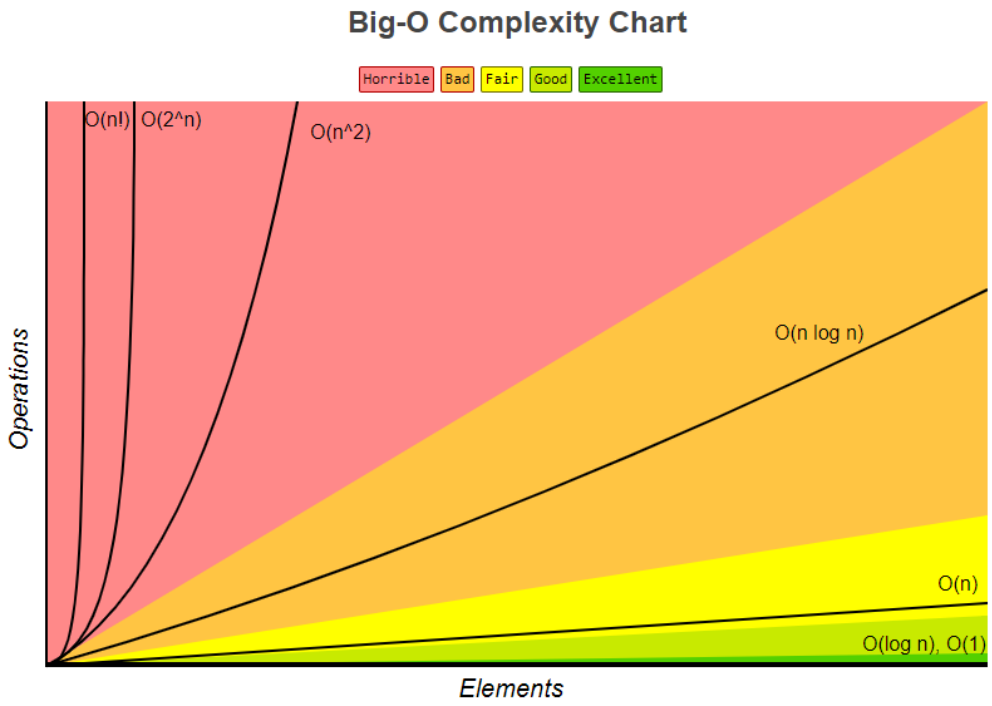


Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Εικόνα 8 : Πολυπλοκότητα δομών δεδομένων που χρησιμοποιούνται ευρύτατα (Source : http://bigocheatsheet.com)

Στη συνέχεια ακολουθεί γράφημα στο οποίο παρατίθενται σχηματικά οι περιπτώσεις της πολυπλοκότητας χειρότερης περίπτωσης (Big-O) (Εικόνα 9).



Εικόνα 9 : Πολυπλοκότητα Χειρότερης Περίπτωσης (ΠΧΠ) (Source : http://bigocheatsheet.com)

Ακολουθεί παρουσίαση των σημαντικότερων υλοποιημένων δομών που προσφέρει το Collections framework και των αποδόσεων που υπόσχεται στις βασικές τους λειτουργίες.

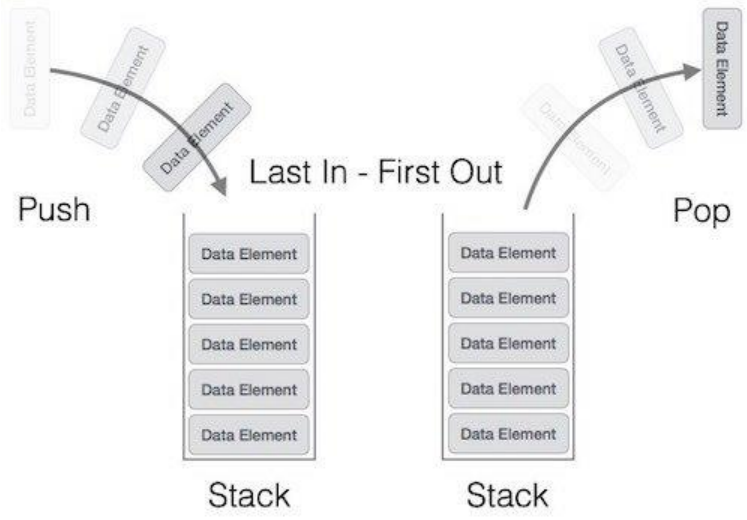
3.3 Class Vector<E>

Η κλάση Vector (Διάνυσμα) [10] υλοποιεί ένα πίνακα αντικειμένων, του οποίου το μήκος μπορεί να μεταβληθεί. Ανήκει στις δομές προγενέστερες της συλλογής και αντ' αυτού προτείνεται από τη Java η χρήση του ArrayList [12]. Δεν θα μελετηθεί περαιτέρω στη συγκεκριμένη διατριβή καθώς δεν αποτελεί βασική επιλογή σε υλοποιήσεις

3.4 Class Stack<E>

Η κλάση Stack (Στοίβα) [11] είναι μια ομοιογενής γραμμική δομή δεδομένων πολιτικής LIFO (Last-in First-out) [9]. Δηλαδή το πιο πρόσφατα εισαχθέν στοιχείο στη στοίβα είναι το στοιχείο

που θα διαγραφεί ή επεξεργαστεί πιο άμεσα (Εικόνα 10).



Εικόνα 10 : Λειτουργίες της Στοίβας (Source : https://www.wisdomjobs.com/e-university/data-structure-algorithms-tutorial-1541/data-structure-algorithms-stack-17906.html)

Η υλοποίηση της στοίβας στη Java [11] γίνεται από την Stack κάνοντας extend την κλάση Vector (Διάνυσμα). Οι βασικές λειτουργίες που προσφέρει είναι :

1. push() : Προσθέτει στοιχείο στην κορυφή.
2. pop() : Αφαιρεί το στοιχείο της κορυφής.
3. empty() : Ελέγχει αν η στοίβα είναι άδεια.
4. peek() : Επιστρέφει το στοιχείο που βρίσκεται στην κορυφή.
5. search() : Επιστρέφει τη θέση που βρίσκεται το στοιχείο που αναζητά κάποιος, αν το στοιχείο δεν υπάρχει στη στοίβα θα επιστρέψει -1.

Η πολυπλοκότητα των λειτουργιών βάσει του πίνακα έχει ως εξής :

Stack	Operations					Memory
Time Complexity	push()	pop()	empty()	peek()	search()	
	O(1)	O(1)	O(1)	O(1)	O(n)	
Space Complexity						O(n)

3.5 Class ArrayList<E>

Η κλάση ArrayList [12] υλοποιεί το interface List και επιστρέφει ένα δυναμικό πίνακα (Array) του οποίου το μήκος μπορεί να αυξομειώνεται. Κάθε ArrayList μόλις αρχικοποιείται έχει μια χωρητικότητα (capacity). Η χωρητικότητα αυτή είναι ακριβώς όση θα χρειαζόταν και ένα List για να αρχικοποιηθεί με το αντίστοιχο n-πλήθος στοιχείων. Όσο προστίθενται στοιχεία αυξάνεται το μήκος του ArrayList . Αν και η Java δεν δίνει περισσότερα στοιχεία σχετικά με το ακριβές κόστος της επιμήκυνσης και της πιθανής ολίσθησης των στοιχείων στον πίνακα, απλά αναφέρει πως το μέσο κόστος αυτής της λειτουργίας είναι μια σταθερά της τάξεως του O(n).

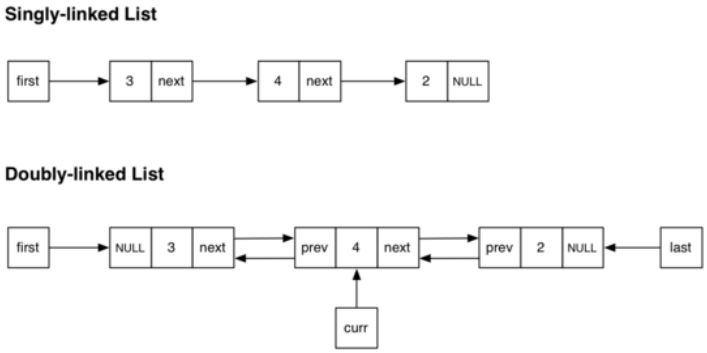
Παράλληλα το ArrayList μπορεί να δεχθεί όλες τα αντικείμενα, (ακόμη και null). Το ArrayList πρέπει να σημειωθεί ότι δεν είναι synchronised , πράγμα που σημαίνει ότι αν προσπαθήσουν ταυτόχρονα δύο νήματα να το προσεγγίσουν θα υπάρξει πρόβλημα στην πιθανή μεταβολή της δομής. Θα πρέπει ο συγχρονισμός να έχει γίνει εκτός δομής , ώστε να δίδεται πρόσβαση στη δομή με προτεραιότητα και συγχρονισμός. Ανάμεσα στις μεθόδους που προσφέρει για χειρισμό του, οι βασικές λειτουργίες είναι :

1. add() : Προσθέτει στοιχείο τέλος της λίστας.
2. get() : Επιστρέφει το στοιχείο που βρίσκεται στη συγκεκριμένη θέση που παίρνει ως όρισμα.
3. remove() : Αφαιρεί το στοιχείο (αν υπάρχει) την πρώτη φορά που θα το συναντήσει καθώς διατρέχει τη λίστα.
4. contains() : Επιστρέφει μια Boolean μεταβλητή (true,false) αν βρεί το αντικείμενο που θα πάρει ως όρισμα.
5. indexOf() : Επιστρέφει τη θέση του στοιχείου που θα πάρει ώς όρισμα.

ArrayList	Operations					Memory
Time Complexity	add()	get()	remove()	contains()	indexOf()	
	O(1)	O(1)	O(n)	O(n)	O(n)	
Space Complexity						O(n)

3.6 Class LinkedList<E>

Η κλάση LinkedList [13] υλοποιεί τα interface List, Deque, Queue και αποτελεί μια υλοποίηση διπλά συνδεδεμένης λίστας (Doubly Linked List). Επιτρέπει τη χρήση όλων των αντικειμένων , όχι όμως των null. Όπως το ArrayList έτσι και η LinkedList δεν είναι synchronized. Το μεγάλο πλεονέκτημα της συνδεδεμένης λίστας είναι ότι δεσμεύει χώρο μόνο για τον κάθε κόμβο (στοιχείο) που προστίθεται και υπάρχει ένας δείκτης ο οποίος δείχνει στο επόμενο στοιχείο , ώστε να μπορεί κανείς να το προσπελάσει. Στη διπλά συνδεδεμένη (Εικόνα 11) υπάρχει ένας δεύτερος δείκτης που δείχνει στον αμέσως προηγούμενο κόμβο (στοιχείο) ώστε να διευκολυνθεί η προσπέλαση προς τα πίσω και να μειωθεί η πολυπλοκότητα των λειτουργιών.



Εικόνα 11 : Αναπαράσταση Συνδεδεμένης λίστας και Διπλά συνδεδεμένης λίστας (Source : https://codewidpassion.blogspot.com)

Όπως έχει αναφερθεί στην περιγραφή των interface της συλλογής οι δομές ουράς (Queue) και διπλής ουράς (Double-queue , εν συντομία Deque) υλοποιούνται από το LinkedList. Στην περίπτωση της ουράς έχουμε εισαγωγή στοιχείου από το τέλος και εξαγωγή από την αρχή, ενώ στη διπλή ουρά έχουμε εισαγωγή και εξαγωγή και από τα δύο μέρη.

Ανάμεσα στις μεθόδους που προσφέρει για χειρισμό του, οι βασικές λειτουργίες είναι :

1. add() : Προσθέτει στοιχείο σε συγκεκριμένη θέση στη λίστα.
2. get() : Επιστρέφει το στοιχείο που βρίσκεται στη συγκεκριμένη θέση που παίρνει ως όρισμα.
3. remove() : Αφαιρεί το στοιχείο από τη συγκεκριμένη θέση.
4. contains() : Επιστρέφει μια Boolean μεταβλητή (true,false) αν βρεί το αντικείμενο που θα πάρει ως όρισμα.

ArrayList	Operations					Memory
Time Complexity	add()	get()	remove()	contains()		
	O(1)	O(n)	O(1)	O(n)		
Space Complexity						O(n)

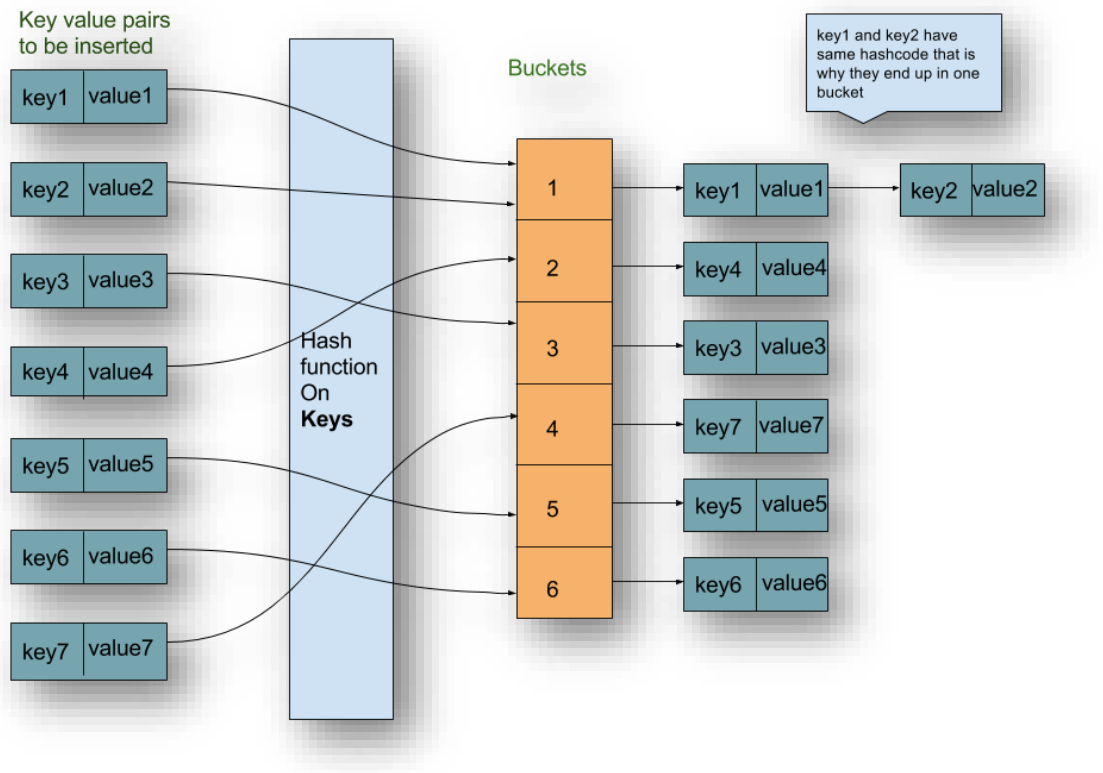
3.7 Class ArrayDeque<E>

Η κλάση ArrayDeque [14] υλοποιεί το interface Deque και επιστρέφει ένα μεταβλητού μήκους πίνακα (Resizable-Array). Δεν έχει κανένα περιορισμό στο πλήθος των στοιχείων που μπορεί να αποθηκεύσει και μπορεί να μεταβληθεί όταν κρίνεται απαραίτητο. Η συγκεκριμένη υλοποίηση είναι προφανώς πιο αποδοτική από την συνδεδεμένη λίστα (LinkedList) στις υλοποιήσεις ουράς, ενώ και σε υλοποιήσεις στοίβας μπορεί να αποβεί ταχύτερη. Δεν θα μελετηθεί περαιτέρω στη συγκεκριμένη διατριβή καθώς δεν αποτελεί βασική επιλογή σε υλοποιήσεις.

3.8 Class HashSet<E>

Η κλάση HashSet [15] υλοποιεί το interface Set και επιστρέφει ένα σύνολο τιμών, πρακτικά ένα πίνακα τιμών (Hash Table). Στα χαρακτηριστικά της δομής συγκαταλέγονται το ότι δεν αποθηκεύει διπλότυπα (duplicates) και δεν διατηρεί τη σειρά των στοιχείων με την οποία εισήχθησαν. Η δομή δηλαδή δεν εγγυάται ότι αν κάποιος διατρέξει τα στοιχεία της , κάθε φορά η σειρά τους θα είναι ίδια. Κυρίαρχο όμως χαρακτηριστικό της είναι οι επιδόσεις της, καθώς υπόσχεται σταθερό χρόνο O(1) για κάθε βασική λειτουργία. Αυτό συμβαίνει , όπως και στο HashMap (Εικόνα 12) , γιατί κάθε στοιχείο απεικονίζεται αμφιμονοσήμαντα (είναι δηλαδή και ένα προς ένα και επί) μέσω μιας συνάρτησης H (hash function εν προκειμένω) σε ένα πεδίο τιμών.

Η κάθε τιμή (hash code) «δείχνει» (απεικονίζεται) σε ένα «κάδο» (bucket) στον οποίο αποθηκεύεται το στοιχείο. Καθώς και άλλα στοιχεία ενδέχεται να πάρουν το ίδιο hash code (η περίπτωση να ισχύει το επί) , άρα να μπούν και στον ίδιο κάδο, εκεί ο κάθε κάδος πια λειτουργεί σαν μια διπλά συνδεδεμένη λίστα η οποία έχει το κάθε στοιχείο να δείχνει στο επόμενο με τη σειρά εισαγωγής τους.



Εικόνα 12 : HashMap Indexing (Source : https://i0.wp.com/programtalk.com)

Πρακτικά το HashSet είναι ένα σύνολο τιμών, S = {…}, άρα δεν επιτρέπει διπλότυπα και έτσι είναι πολύ εύκολο όταν χρειαστεί να γίνει κάποια βασική λειτουργία να αναζητηθεί σε σταθερό χρόνο το συγκεκριμένο στοιχείο βάσει της εικόνας του. Να σημειωθεί εδώ ότι η συγκεκριμένη δομή λειτουργεί ακριβώς όπως ένα σύνολο τιμών, άρα δεν υπάρχει η λειτουργία get() ή κάτι αντίστοιχο , καθώς δεν υπάρχει λόγος για κάτι τέτοιο. Το μόνο που χρειάζεται είναι να ξέρει κανείς είναι αν σε αυτό το σύνολο υπάρχει η τιμή που ψάχνει. Αλλιώς προτείνονται άλλες λύσεις δομών , όπως το HashMap που κάθε στοιχείο πια αποτελείται από το ζευγάρι κλειδιού-τιμής (key-value) και θα δούμε στη συνέχεια. Σημειώνεται και εδώ ότι η συγκεκριμένη υλοποίησης δεν είναι synchronized , οπότε χρειάζεται να ληφθεί υπόψη σε υλοποιήσεις και να γίνεται συγχρονισμός μέσω κώδικα. Ανάμεσα στις μεθόδους που προσφέρει για χειρισμό του, οι βασικές λειτουργίες είναι :

1. add() : Προσθέτει στοιχείο αν αυτό δεν υπάρχει στο σύνολο.
2. remove() : Αφαιρεί το στοιχείο (αν υπάρχει) από το σύνολο.
3. contains() : Επιστρέφει μια Boolean μεταβλητή (true,false) αν το αντικείμενο που θα πάρει ως όρισμα περιέχεται στο HashSet.

HashSet	Operations			Memory
Time Complexity	add()	remove()	contains()	
	O(1)	O(1)	O(1)	
Space Complexity				O(n)

3.9 Class LinkedHashSet<E>

Η κλάση LinkedHashSet [16] διαφοροποιείται από το HashSet στο ότι η ιεράρχηση των στοιχείων γίνεται βάσει της σειράς που εισάγονται. Όταν δηλαδή κάποιος διατρέξει τη δομή το LinkedHashSet εγγυάται τη διατήρηση της σειράς εισαγωγής των στοιχείων, σε αντίθεση με το HashSet.

Ανάμεσα στις μεθόδους που προσφέρει για χειρισμό του, οι βασικές λειτουργίες είναι :

1. add() : Προσθέτει στοιχείο αν αυτό δεν υπάρχει στο σύνολο.
2. remove() : Αφαιρεί το στοιχείο (αν υπάρχει) από το σύνολο.
3. contains() : Επιστρέφει μια Boolean μεταβλητή (true,false) αν το αντικείμενο που θα πάρει ως όρισμα περιέχεται στο HashSet.

LinkedHashSet	Operations			Memory
Time Complexity	add()	remove()	contains()	
	O(1)	O(1)	O(1)	
Space Complexity				O(n)

3.10 Class TreeSet<E>

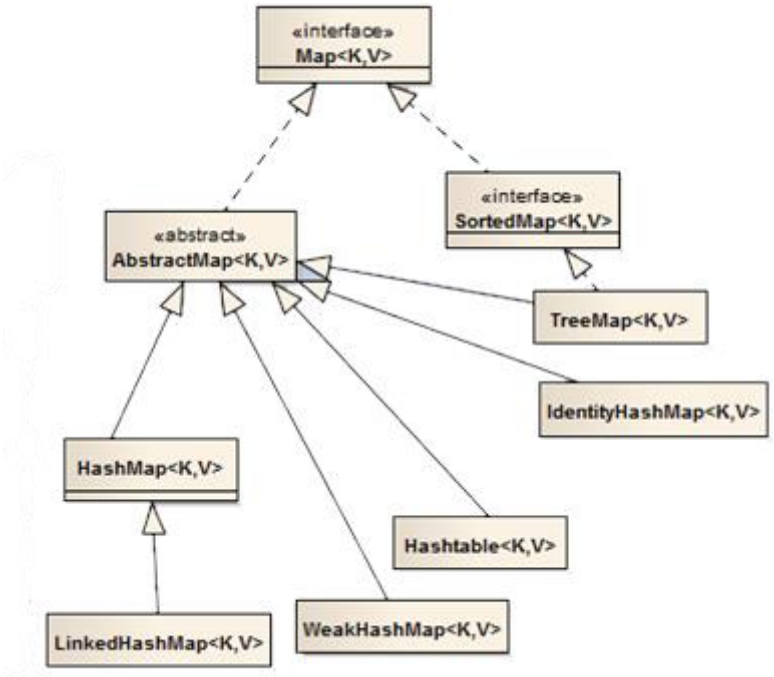
Η κλάση TreeSet [17] υλοποιεί τα interface Set, SortedSet και επιστρέφει μια δομή δενδρικής μορφής η οποία βασίζεται στο TreeMap. Τα στοιχεία εισάγονται και ιεραρχούνται μέσω της compareTo() είτε βάσει Natural Ordering [18] (έλεγχος των στοιχείων μέσω της μεθόδου ως έχει, χωρίς να επέμβει κάποιος στις μεθόδους equals() , hashCode()) , είτε έχοντας κάνει override τις μεθόδους equals() και hashCode() ώστε να αλλάξουμε με υλοποίηση τα κριτήρια σύγκρισης ανάμεσα σε δύο αντικείμενα της ίδιας κλάσης. Και αυτή η υλοποίηση δεν είναι synchronised. Η δενδρική αυτή δομή υπόσχεται χρόνο O(logn) για τις βασικές λειτουργίες , ανάμεσα στις οποίες είναι :

1. add() : Προσθέτει στοιχείο στη δομή.
2. remove() : Αφαιρεί το στοιχείο.
3. contains() : Επιστρέφει μια Boolean μεταβλητή (true,false) αν βρεί το αντικείμενο που θα πάρει ως όρισμα.

TreeSet	Operations			Memory
Time Complexity	add()	remove()	contains()	
	O(logn)	O(logn)	O(logn)	
Space Complexity				O(n)

3.11 Maps

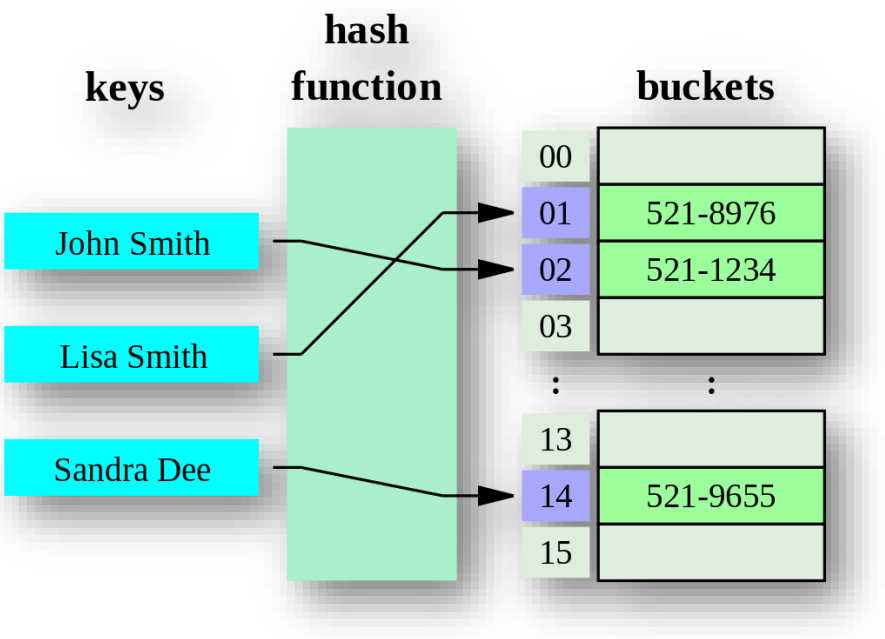
Το interface Map [19] βρίσκεται στην κορυφή των δομών που το υλοποιούν (Εικόνα 13). Προσφέρει μια αντιστοίχιση (mapping) κλειδιών σε τιμές (key – values) και συμβολίζεται με Map<K,V>. Οι κλάσεις που το υλοποιούν μπορούν να πάρουν οποιαδήποτε κλάση ως K (Key) , και οποιαδήποτε κλάση ή άλλη δομή ως V (Value). Δεν μπορεί να περιέχει διπλότυπα στα κλειδιά καθώς είναι μοναδικά και κάθε κλειδί μπορεί να αντιστοιχηθεί μόνο σε μια τιμή. Ακολουθεί ανάλυση των δύο περισσότερο χρησιμοποιούμενων υλοποιήσεων του.



Εικόνα 13 : Οι υλοποιήσεις του Map interface (Source : https://www.programering.com/a/MTNwEjMwATU.html)

3.12 Class Hashtable<K,V>

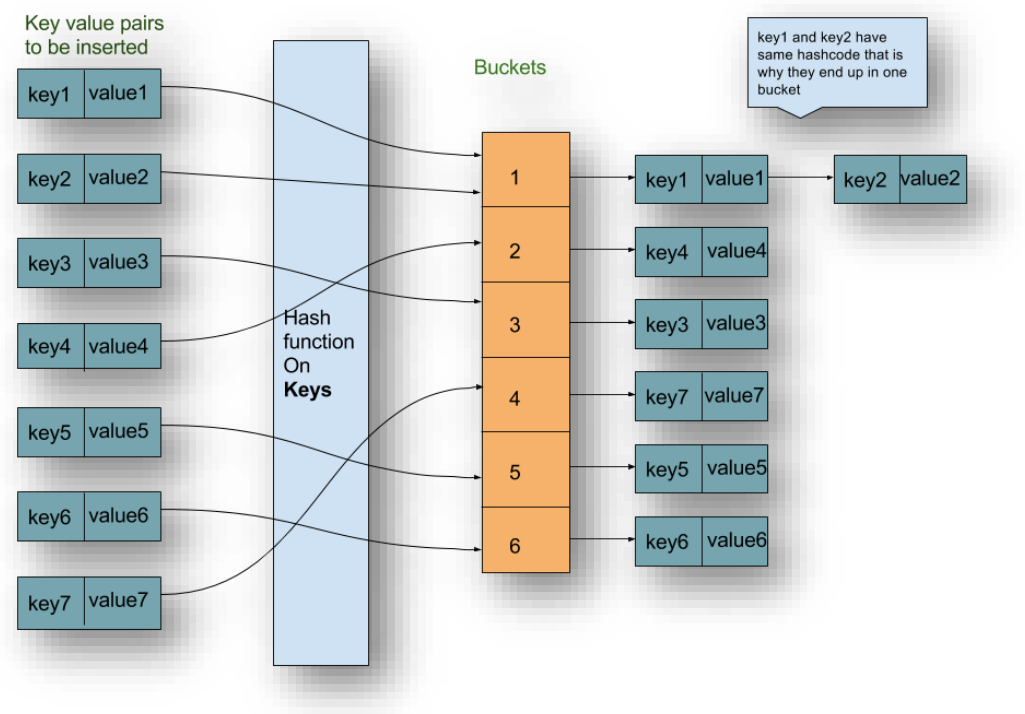
Το HashTable [20] λειτουργεί ακριβώς όπως και το HashSet (3.8) με τη διαφορά ότι εδώ το η Hash function παίρνει σαν όρισμα το αντικείμενο που αναπαριστά το κλειδί (K) , επιστρέφει το hashCode , που αντιστοιχεί όμως στο ζευγάρι Key-Value, και αυτό δείχνει στον αντίστοιχο «κάδο» πια (Εικόνα 14) . Πολύ απλά αντί για τιμή έχουμε ζευγάρι Κλειδιού-Τιμής. Να σημειωθεί ότι το Hashtable δεν επιτρέπει την αποθήκευση null κλειδιών ή τιμών, ενώ είναι synchronized, άρα προσφέρεται για multi-threaded (πολυνηματικές) υλοποιήσεις.



Εικόνα 14 : Hash Table (Key : Name, Value : Telephone) (Source : https://en.wikipedia.org/wiki/Hash_table)

Το Hashtable είναι μια State-of-the-art δομή η οποία έχει δύο πολύ βασικές παραμέτρους που επηρεάζουν την απόδοση της:

1. Initial Capacity (Αρχική χωρητικότητα): Χωρητικότητα είναι ο αριθμός των «κάδων» που έχει διαθέσιμους η δομή και ως αρχική χωρητικότητα είναι ο αριθμός των κάδων που έχει η δομή τη στιγμή που δημιουργείται. Υπενθυμίζεται ότι σε περίπτωση που δύο ζευγάρια έχουν το ίδιο hashCode (hashCollision) , αποθηκεύονται στον ίδιο κάδο ως κόμβοι σε διπλά συνδεδεμένη λίστα, όπου εκεί η αναζήτηση πια μοιραία θα είναι σειριακή (Εικόνα 15).
2. Load Factor (Παράγοντας φόρτωσης): Είναι μια παράμετρος με συγκεκριμένη τιμή που αντιπροσωπεύει το ποσοστό της δομής που είναι φορτωμένο με δεδομένα. Αν αυτή η τιμή ξεπεραστεί (Η default τιμή είναι ορισμένη στο 0.75 επί του συνόλου) , τότε θα αυξηθεί η χωρητικότητά του Hashtable.



3.

Εικόνα 15 : Οι συνδεδεμένες λίστες στους «κάδους» του Hashtable (Source : https://i0.wp.com/programtalk.com)

Βασικές λειτουργίες :

1. put() : Προσθέτει κλειδί στη δομή, και το αντιστοιχίζει με συγκεκριμένη τιμή.
2. get() : Επιστρέφει την τιμή του αντίστοιχου κλειδιού.
3. remove() : Αφαιρεί το κλειδί (αν υπάρχει) και κατ' επέκταση και την τιμή που αντιστοιχεί σε αυτό.
4. contains() : Επιστρέφει μια Boolean μεταβλητή (true,false) αν το αντικείμενο που θα πάρει ως όρισμα, περιέχεται μέσα στη δομή.

Hashtable	Operations				Memory
Time Complexity	put()	get()	remove()	contains()	
	O(1)*	O(1)*	O(1)*	O(1)*	
Space Complexity					O(n)

*Η χρονική πολυπλοκότητα βασίζεται στο γεγονός ότι δεν θα υπάρξουν hash collisions, πρακτικά δηλαδή δεν θα υπάρξουν δύο κλειδιά στα οποία θα αποδοθεί το ίδιο hashcode. Ενώ είναι αρκετά σπάνιο στην πράξη , δεν αποκλείεται να συμβεί σε επίπεδο απρόσεκτης υλοποίησης. Σε μια τέτοια περίπτωση αναμφισβήτητα λόγω της συνδεδεμένης λίστας πια, οδηγούμαστε σε πολυπλοκότητα τάξεως O(n).

3.13 Class HashMap <K,V>

Η κλάση HashMap [21] υλοποιεί το interface Map και λειτουργεί με τον ίδιο τρόπο όπως και το HashTable. Διαφοροποιείται όμως σε 3 βασικά σημεία:

1. Δεν είναι synchronized. Αυτό από μόνο του το κάνει πολύ πιο αποδοτικό όταν χρησιμοποιείται σε μονονηματικές εφαρμογές καθώς δεν απαιτείται το έξτρα κόστος του συγχρονισμού.
2. Επιτρέπει μόνο ένα null κλειδί , και άπειρες null τιμές.
3. Μπορεί , αν απαιτηθεί, να τροποποιηθεί η σειρά προσπέλασης των τιμών (By default είναι η σειρά εισαγωγής). Επίσης σε αντίθεση με το Hashtable , εδώ το HashMap δεν εγγυάται τη σειρά ιεράρχησης σε μήκος χρόνου (όπως συμβαίνει και με τις υλοποιήσεις του Set)

Βασικές λειτουργίες :

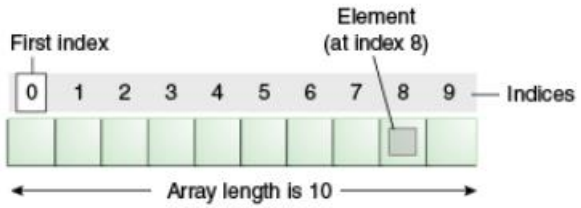
1. put() : Προσθέτει κλειδί στη δομή, και το αντιστοιχίζει με συγκεκριμένη τιμή.
2. get() : Επιστρέφει την τιμή του αντίστοιχου κλειδιού.
3. remove() : Αφαιρεί το κλειδί (αν υπάρχει) και κατ' επέκταση και την τιμή που αντιστοιχεί σε αυτό.
4. contains() : Επιστρέφει μια Boolean μεταβλητή (true,false) αν βρεί το αντικείμενο που θα πάρει ως όρισμα.

HashMap	Operations				Memory
Time Complexity	put()	get()	remove()	contains()	
	O(1)*	O(1)*	O(1)*	O(1)*	
Space Complexity					O(n)

*Η χρονική πολυπλοκότητα βασίζεται στο γεγονός ότι δεν θα υπάρξουν hash collisions, πρακτικά δηλαδή υπάρξουν δύο κλειδιά στα οποία να αποδοθεί το ίδιο. Ενώ είναι αρκετά σπάνιο στην πράξη , αναμφισβήτητα λόγω της συνδεδεμένης λίστας πια, οδηγούμαστε σε πολυπλοκότητα τάξεως O(n).

3.14 Array

Τελευταία δομή που θα μελετηθεί είναι το Array (Πίνακας). Το Array αποτελεί μια δομή γραμμική, τυχαίας προσπέλασης (Εικόνα 16).



Εικόνα 16 : Array (Source : https://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html)

Ενώ διαθέτει πολλά κοινά στοιχεία με το ArrayList και λειτουργεί με πανομοιότυπο τρόπο, οι διαφορές τους είναι πολύ σημαντικές.

1. Το Array έχει σταθερό μήκος, το οποίο δεν μπορεί να μεταβληθεί μετά την αρχικοποίηση του.
2. Μπορεί να δεχθεί όλους τους τύπους δεδομένων πχ Primitives, κάτι που δεν ισχύει για το ArrayList (Μόνο Κλάσεις αντικειμένων, πχ δέχεται τη wrapper κλάση Integer αλλά όχι τους int).
3. Είναι μια ομοιογενής δομή δεδομένων. Αυτό σημαίνει ότι όλα τα δεδομένα που

αποθηκεύονται στη δομή έχουν τον ίδιο τύπο κάτι που δεν ισχύει για το ArrayList.

4. Το σημαντικότερο μειονέκτημα ίσως είναι ότι δεν υπάρχουν μέθοδοι χειρισμού των βασικών λειτουργιών του Array, αντίθετα με το ArrayList, οπότε αυτό προσθέτει επιπλέον μόχθο σε προγραμματιστικό επίπεδο.

4. Java Microbenchmark Harness (JMH)

4.1 Java Virtual Machine & Benchmarking

Για να μπορέσει κάποιος να διεξάγει σενάρια μέτρησης των επιδόσεων (Benchmarking) πάνω στις δομές της Java, θα μπορούσε θεωρητικά να το κάνει χρησιμοποιώντας λίγες γραμμές κώδικα σε Java. Το JVM (Java Virtual Machine) [23] είναι υλοποιημένο με τέτοιο τρόπο που να βελτιστοποιεί συνεχώς την απόδοση του (continuous optimization). Για παράδειγμα στο αρχικό στάδιο ενός benchmarking η CPU έχει μια απόδοση σε χρόνο Τ. Όσο όμως γίνονται κύκλοι επαναλήψεων η CPU έχει πια ήδη κάνει το warm-up και η απόδοσή της βελτιστοποιείται κύκλο με κύκλο επανάληψης. Αυτό ακούγεται ως κάτι πραγματικά θετικό σε εργασιακές συνθήκες όμως σε συνθήκες μετρήσεων ενδεχομένως τα αποτελέσματα να αποκλίνουν από τα πραγματικά στοιχεία.

Τι είναι όμως το JVM και γιατί επεμβαίνει συνεχώς , κατ' αυτό τον τρόπο? Οι περισσότερες γλώσσες προγραμματισμού μετατρέπουν τον κώδικα ενός προγράμματος σε γλώσσας μηχανής που μπορεί να εκτελεστεί άμεσα από την πλατφόρμα (ουσιαστικά το λειτουργικό σύστημα) στην οποία βρίσκονται. Η Java είναι platform-independent, δηλαδή δεν ενδιαφέρεται για την πλατφόρμα στην οποία είναι εγκατεστημένη. Ο μεταγλωττιστής της παράγει το bytecode από τον κώδικα Java του προγράμματος προς εκτέλεση, και μετά αναλαμβάνει το JVM να το μετατρέψει και να το εκτελέσει στην αντίστοιχη γλώσσα μηχανής που αναγνωρίζει το εκάστοτε λειτουργικό. Το JVM (Java Virtual Machine) είναι μια εικονική μηχανή (VM : Virtual Machine) η οποία είναι ανεξάρτητη πλατφόρμας και λειτουργικού (Platform independent) και είναι υπεύθυνη για να παρέχει ένα runtime περιβάλλον, ένα περιβάλλον δηλαδή στο οποίο θα μπορούν να τρέχει το bytecode ενός προγράμματος Java που έχει παραχθεί από το μεταγλωττιστή της γλώσσας. Το JVM μετατρέπει το bytecode στη αντίστοιχη γλώσσα μηχανής και το εκτελεί χωρίς κανένα πρόβλημα. Η διαδικασία αυτή αποτελεί και το μεγάλο ανταγωνιστικό πλεονέκτημα της Java, καθώς αδιαφορεί για το λειτουργικό και απλώς παράγει το bytecode και από εκεί αναλαμβάνει το JVM, με το αντίστοιχο κόστος φυσικά σε χρόνο.

4.2 Java Microbenchmark Harness Framework

Το JMH (Java Microbenchmark Harness) [24] είναι ένα framework που έχει υλοποιηθεί από την κοινότητα ανοιχτού κώδικα (open source) OpenJDK (Open Java Development Kit) και συνεχίζει την υλοποίηση της πλατφόρμας SE (Standard Edition) της Java, μια προσπάθεια που ξεκίνησε η ίδια η Sun Microsystems [25] το 2006. Το JMH υπόσχεται μεγάλη ακρίβεια στις μετρήσεις καθώς διαθέτει ένα πλήθος παραμετροποιήσεων , ικανό να ελαχιστοποιήσει τις παρεμβάσεις του JVM, ώστε τα στοιχεία να είναι πιο ακριβή. Ως πιο σημαντικές παραμετροποιήσεις κρίνονται οι ακόλουθες , οι οποίες και σηματοδοτούνται με τη χρήση συγκεκριμένων annotations :

1. @BenchmarkMode : Επιλογή του τύπου benchmark , όπως ανάμεσα στα πιο σημαντικά Throughtput (διεκπεραιωτική ικανότητα) , AverageTime (Διάρκεια κατά μέσο όρο).
2. @Warmup : ένα τρόπον τινά «ζέσταμα» του JVM ώστε να αποφευχθεί το overhead της εκκίνησης του JVM στα αρχικά σενάρια εκτέλεσης*.
3. @Fork : Επιλογή του πλήθους των διεργασιών (Fork) ώστε να υπάρχει μέτρο σύγκρισης ανάμεσα στις εκτελέσεις. Αν πχ κάποια διεργασία χρειαστεί πολύ περισσότερο χρόνο για κάποιο λόγο, αυτό θα γίνει εύκολα αντιληπτό από τις μετρήσεις των υπολοίπων , αποφεύγοντας τις λάθος εκτιμήσεις.
4. @Measurement : Επιλογή πλήθους επαναλήψεων των μετρήσεων στα σενάρια εκτέλεσης.
5. @State : Καθορισμός της ορατότητας (scope) κάποιων μεταβλητών , ώστε να μπορέσουν να επαναχρησιμοποιηθούν σε συγκεκριμένα σημεία του κώδικα.

**Να σημειωθεί ότι ως σενάριο εκτέλεσης θεωρείται κάθε ένα τέστ (πρακτικά η κάθε μέθοδος που γράφεται σε κώδικα για το benchmarking) το οποίο και ελέγχει μια και μόνο συμπεριφορά ανά τέστ.*

4.3 Benchmarking JMH Configuration, Procedure & Hardware

Για τις ανάγκες των μετρήσεων επιλέχθηκε η ακόλουθη παραμετροποίηση στο JMH. Για λόγους παρουσίασης, τα ορίσματα στις Benchmark κλάσεις περάστηκαν με @annotations [26] στον κώδικα ώστε να πιο επεξηγηματική η αναφορά. Θα υπάρξει ενδεικτικά και εικόνα με την αντίστοιχη παραμετροποίηση από την κονσόλα.

```
@State(Scope.Benchmark)
@Warmup(iterations = 10, time = 1)
@Measurement(iterations = 10, time = 1)
@Fork(5)
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.NANOSECONDS)
```

- @State : Με την επιλογή ορατότητας για κάθε νήμα, σημαίνει ότι πρακτικά κάθε νήμα θα έχει το δικό του αντίγραφο από τα State objects ώστε να συγκριθούν ακόμη και σε επίπεδο νήματος οι επιδόσεις και αν υπάρχουν τυχόν διαφοροποιήσεις.
- @Warmup : Πόσες φορές να κάνει warmup το JVM ώστε όταν θα τρέξει τα σενάρια benchmarking , και πόσο χρόνο να διαρκέσει το κάθε warm-up.
- @Measurement : Πόσες επαναλήψεις να κάνει ανά σενάριο στην φάση πια της εκτέλεσης τους.
- @Fork : Πόσα fork (πόσα αντίγραφα των διεργασιών να δημιουργήσει) να κάνει ανά σενάριο εκτέλεσης.
- @BenchmarkMode : Σε ποια κατάσταση θα τρέξουν τα σενάρια και τι θα αναδείξουν τα αποτελέσματα.
- @OutputTimeUnit : Σε ποια μονάδα χρόνου θα επιστραφούν τα αποτελέσματα.
- Blackhole : Ένας τέχνασμα της ομάδας του JMH Framework για να «ξεγελάσει» τον compiler ώστε να μη θεωρεί κάποιες μεταβλητές και τμήματα κώδικα, «νεκρά- ανενεργά» (dead-code) αλλά τα λαμβάνει υπόψιν του. (Συγκαταλέγεται στις βελτιστοποιήσεις που κάνει το JVM η αναγνώριση τμημάτων και μεταβλητών οι οποίες δεν χρησιμοποιούνται, άρα δεν υπολογίζονται και δεν αξιοποιούνται από το JVM. Ο εναλλακτικός τρόπος είναι κάθε μέθοδος να επιστρέφει πάντα κάτι).

Ως έκδοση της Java επιλέχθηκε το Java SE Development Kit 11.0.1 , ενώ μέχρι την ώρα των μετρήσεων η runtime διαθέσιμη έκδοση ήταν η 1.8.0.201.

```
C:\Users\zixcode>java -version
java version "1.8.0_201"
Java(TM) SE Runtime Environment (build 1.8.0_201-b09)
Java HotSpot(TM) 64-Bit Server VM (build 25.201-b09, mixed mode)

C:\Users\zixcode>javac -version
javac 11.0.1

C:\Users\zixcode>
```

Εικόνα 17 : Η Εγκατεστημένη Java version.

Για να τρέξουν τα benchmark σενάρια χρησιμοποιήθηκε το εξαιρετικό build tool Maven, ενώ το IDE που χρησιμοποιήθηκε για τη συγγραφή κώδικα ήταν το IntelliJ Ultimate της Jetbrains στην έκδοση 2018.2.

Εικόνα 18 : Η εγκατεστημένη version του IntelliJ.

Οι προδιαγραφές του υπολογιστή στον οποίο έγιναν οι μετρήσεις παρατίθενται στην ακόλουθη εικόνα, σε λειτουργικό Microsoft Windows 10 Home.

Device specifications

Device name	DESKTOP-D0P865E
Processor	Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz 2.90 GHz
Installed RAM	16,0 GB (15,9 GB usable)
Device ID	315ABC91-DFD0-457D-B89C-3C37536A4EAA
Product ID	00325-95800-00000-AAOEM
System type	64-bit operating system, x64-based processor

Εικόνα 19 : Τα specifications του hardware.

Τα σενάρια αφορούσαν τις βασικές λειτουργίες στις υπό εξέταση δομές και συγκεκριμένα : την εύρεση, την προσπέλαση, την εισαγωγή και την διαγραφή στοιχείων από την εκάστοτε δομή.

Ως όγκος δεδομένων με τον οποίο τροφοδοτήθηκαν τα σενάρια, επιλέχθηκαν τα ακόλουθα 6 πλήθη στοιχείων τα οποία και παρατίθενται ακολούθως:

```
@Param({"1","10", "100", "1000", "10000", "100000"})
int N;
```

Ανάλογα με τη δομή υπήρξαν και διαφοροποιήσεις στα σενάρια , καθώς οι δομές του Collections framework δεν υποστηρίζουν primitives. Προς διευκόλυνση της ανάγνωσης, κάθε Benchmark σενάριο παρατίθεται στο αντίστοιχο τμήμα με τα αποτελέσματα (Ενότητα 5).

Όπου αυτό ήταν εφικτό (πχ Array) ελέγχθηκε με αντίστοιχο σενάριο. Για τα σενάρια χρησιμοποιήθηκε η wrapper class Integer (ακέραιος με autoboxing) και δημιουργήθηκε άλλη μια : η Element class η οποία διαθέτει 3 πεδία (όλα wrapper classes για να επιτρέπονται null τιμές, προσομοιώνοντας συνθήκες παραγωγής) . Συγκεκριμένα η class Element έχει ένα πεδίο τύπου Long ως id, ένα πεδίο String ως όνομα και ένα πεδίο τύπου Boolean ως αναγνωριστικό τύπο του αντικειμένου. Η class Element παρατίθεται αυτούσια στη συνέχεια.

```
@State(Scope.Benchmark)
public class Element implements Comparable<Element>{

  private Long id;
  private String name;
  private Boolean type;

  public Element() {}

  public Element(Long id, String name, Boolean type) {
    this.id = id;
    this.name = name;
    this.type = type;
  }

  public Long getId() {
    return id;
  }

  public void setId(Long id) {
    this.id = id;
  }
```

```
  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }

  public Boolean getType() {
    return type;
  }

  public void setType(Boolean type) {
    this.type = type;
  }

  @Override
  public boolean equals(Object o) {
    if (this == o) {
      return true;
    }
    if (o == null || getClass() != o.getClass()) {
      return false;
    }
    Element element = (Element) o;
    return Objects.equals(id, element.id) &&
        Objects.equals(name, element.name) &&
        Objects.equals(type, element.type);
  }

  @Override
  public int hashCode() {
    return Objects.hash(id, name, type);
  }

  @Override
  public String toString() {
    return "Element{" + "id=" + id + ", name='" + name + '\'' + ", type=" +
type + '}';
  }

  @Override
  public int compareTo(Element o) {
    return (int)(this.id - o.getId());
  }
}
```

Να σημειωθεί ότι η συγκεκριμένη κλάση φέρει το annotation @State , όπερ και σημαίνει ότι τα instances της δεν επηρεάζουν (δεν προσμετρούνται με την έννοια της δέσμευσης μνήμης) τα σενάρια εκτέλεσης. Τα σενάρια εκτέλεσης τμηματοποιήθηκαν ανά Interface, ώστε να είναι πιο σωστά συγκρίσιμα τα στοιχεία ανάμεσα στις υλοποιήσεις. Για παράδειγμα ενδεικτικά παρατίθεται η class ListBenchmark στην οποία ελέγχονται τα metrics για τις υλοποιήσεις του List interface : ArrayList & LinkedList.

```
@State(Scope.Benchmark)
@Warmup(iterations = 10, time = 1)
@Measurement(iterations = 10, time = 1)
@Fork(5)
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.NANOSECONDS)

public class ListBenchmark {

  @Param({"1", "10", "100", "1000", "10000", "100000"})
  int N;

  Element[] elements;

  List<Element> emptyArraylist;
  List<Element> emptyLinkedlist;

List<Element> fullArraylist;
  List<Element> fullLinkedlist;

  Iterator<Element> iteratorArraylist;
  Iterator<Element> iteratorLinkedlist;
```

Οι μεταβλητές και οι δομές που δηλώνονται δεσμεύουν χώρο μόνο για τις ανάγκες εκτέλεσης του σεναρίου και ο χρόνος-χώρος που απαιτεί η δημιουργία τους δεν συνυπολογίζεται στα σενάρια εκτέλεσης.

Αυτό γίνεται περνώντας τα annotation @Setup , @TearDown , τα οποία υποδηλώνουν στο JVM να μη λάβει καθόλου υπόψιν το κόστος δημιουργίας και καταστροφής των συγκεκριμένων δομών και μεταβλητών , αλλά τα παρέχει έτοιμα και φορτωμένα με τα στοιχεία για να εκτελεστούν τα σενάρια.

Το @Setup , με ορατότητα (Scope) Invocation καλείται πριν από κάθε σενάριο εκτέλεσης

και περιέχει όλες τις μεταβλητές και τις δομές που είναι απαραίτητες για το σενάριο.

```
/** Setup all properties & structures before each test. Does not affect
testing. */

@Setup(Level.Invocation)
public void setUp() {

  elements = createMockArray(N);
  emptyArraylist = new ArrayList<>();
  emptyLinkedlist = new LinkedList<>();

  fullArraylist = new ArrayList<>();
  fullLinkedlist = new LinkedList<>();

  for (Element e : elements) {
    fullArraylist.add(e);
    fullLinkedlist.add(e);
  }

  iteratorArraylist = fullArraylist.iterator();
  iteratorLinkedlist = fullLinkedlist.iterator();
}
```

Το @TearDown , με ορατότητα (Scope) Invocation καλείται μετά από την εκτέλεση κάθε σεναρίου και κάνει null όλες τις δομές και τις μεταβλητές, καθαρίζοντας έτσι όλες τις μεταβλητές του instance*.

```
/** After Each test make all properties & structures null. */

@TearDown(Level.Invocation)
public void tearDown() {
  elements = null;
  emptyArraylist = null;
  emptyLinkedlist = null;
  fullArraylist = null;
  fullLinkedlist = null;
  iteratorArraylist = null;
  iteratorLinkedlist = null;
}
```

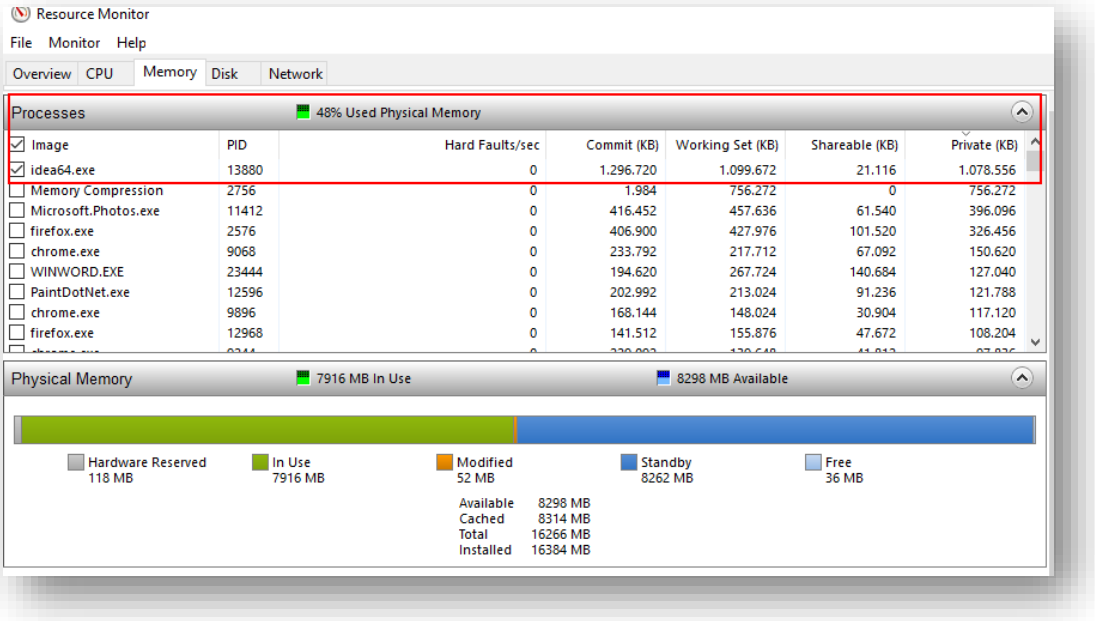
**Να σημειωθεί ότι αυτό δεν σημαίνει και αυτόματα κλήση του Garbage Collector , καθώς το JVM πια δουλεύει πιο βελτιστοποιημένα από ότι προηγούμενες εκδόσεις(πχ. όπως οι περιπτώσεις σε null properties, variables).*

Και σε κάθε Benchmark class υπάρχουν βοηθητικές μέθοδοι οι οποίες αρχικοποιούν μεταβλητές και φορτώνουν τις δομές κατά τη διάρκεια του setup. Ενδεικτικά παρατίθενται οι βοηθητικές μέθοδοι του ListBenchmark.

```
/**
 * Helper method for setup.<br>
 * Creates & fills the array of elements.
 *
 * @param N number of elements.
 * @return the created array.
 * @return the created array.
 */
public Element[] createMockArray(int N) {
  Element[] mockElements = new Element[N];
  for (int i = 0; i < N; i++) {
    Element element = new Element();
    element.setId(Long.valueOf(i + 1));
    element.setName(getClass().getSimpleName() + " : " + i + 1);

    if (i % 2 == 0) element.setType(Boolean.TRUE);

    mockElements[i] = element;
  }
  return mockElements;
}

/**
 * Helper method for setup.<br>
 * Creates & fills the array of integers (Wrapper Class).
 *
 * @param N number of integer elements.
 * @return the created array.
 */
public Integer[] createWrapperClassArray(int N) {
  Integer[] integers = new Integer[N];

  for (int i = 0; i < integers.length; i++) {
    integers[i] = Integer.valueOf(i);
  }
  return integers;
}
```

4.4 Εκτέλεση

Η εκτέλεση των σεναρίων γίνεται με 2 τρόπους : είτε από κονσόλα , είτε μέσα από το ίδιο το IDE. Όμως όπως είναι εμφανές το IntelliJ έχει ιδιαίτερα μεγάλο αποτύπωμα μνήμης (> του 1 Gb).



Εικόνα 20 : Memory footprint of IntelliJ.

Για να έχει λοιπόν όλα τα resources διαθέσιμα το benchmark εκτελέστηκε από κονσόλα μέσω Maven. Απαραίτητα το project πρέπει να γίνει build αρχικά δίνοντας :

```
C:\Users\zixcode\IdeaProjects\thesis-jmh-becnhmarks>mvn clean install
```

Εικόνα 21 : Maven build command.

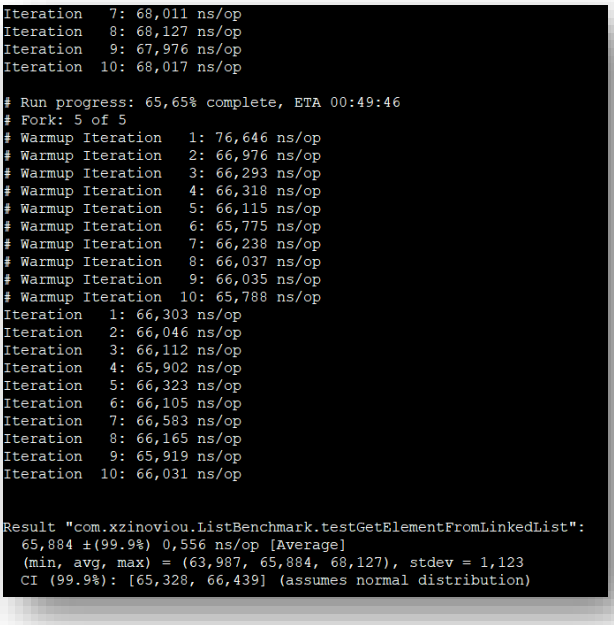
Αν δεν υπάρχει κάποιο error γίνεται επιτυχώς build. Με το build έχει δημιουργηθεί ένα

folder με τίτλο target στο οποίο μέσα δημιουργήθηκε το benchmarks.jar package αρχείο με τα benchmarks. Τρέχει με τα ορίσματα που περνάνε στην κονσόλα. Υπενθυμίζεται ότι τα ορίσματα των σεναρίων είναι περασμένα στον κώδικα, οπότε δεν χρειάζεται να δοθούν άλλα. Το μόνο που θα δοθεί είναι για την εξαγωγή του τύπου αρχείου σε μορφή csv για να είναι επεξεργάσιμο.

```
[INFO] skip non existing resourceDirectory C:\Users\zixcode\IdeaProjects\thesis-jmh-becnhmarks\src\test\resour
ces
[INFO]
[INFO] --- maven-compiler-plugin:3.1:testCompile (default-testCompile) @ thesis-jmh-becnhmarks ---
[INFO] No sources to compile
[INFO]
[INFO] --- maven-surefire-plugin:2.17:test (default-test) @ thesis-jmh-becnhmarks ---
[INFO] No tests to run.
[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ thesis-jmh-becnhmarks ---
[INFO] Building jar: C:\Users\zixcode\IdeaProjects\thesis-jmh-becnhmarks\target\thesis-jmh-becnhmarks-1.0.jar
[INFO]
[INFO] --- maven-shade-plugin:2.2:shade (default) @ thesis-jmh-becnhmarks ---
[INFO] Including org.openjdk.jmh:jmh-core:jar:1.21 in the shaded jar.
[INFO] Including net.sf.jopt-simple:jopt-simple:jar:4.6 in the shaded jar.
[INFO] Including org.apache.commons:commons-math3:jar:3.2 in the shaded jar.
[INFO] Replacing C:\Users\zixcode\IdeaProjects\thesis-jmh-becnhmarks\target\benchmarks.jar with C:\Users\zixco
de\IdeaProjects\thesis-jmh-becnhmarks\target\thesis-jmh-becnhmarks-1.0-shaded.jar
[INFO]
[INFO] --- maven-install-plugin:2.5.1:install (default-install) @ thesis-jmh-becnhmarks ---
[INFO] Installing C:\Users\zixcode\IdeaProjects\thesis-jmh-becnhmarks\target\thesis-jmh-becnhmarks-1.0.jar to
C:\Users\zixcode\.m2\repository\com\xzinoviou\thesis-jmh-becnhmarks\1.0\thesis-jmh-becnhmarks-1.0.jar
[INFO] Installing C:\Users\zixcode\IdeaProjects\thesis-jmh-becnhmarks\pom.xml to C:\Users\zixcode\.m2\reposito
ry\com\xzinoviou\thesis-jmh-becnhmarks\1.0\thesis-jmh-becnhmarks-1.0.pom
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 4.598 s
[INFO] Finished at: 2019-01-20T17:14:28+02:00
[INFO] Final Memory: 16M/60M
[INFO] ------------------------------------------------------------------------

C:\Users\zixcode\IdeaProjects\thesis-jmh-becnhmarks>java -jar target/benchmarks.jar -rf csv
```

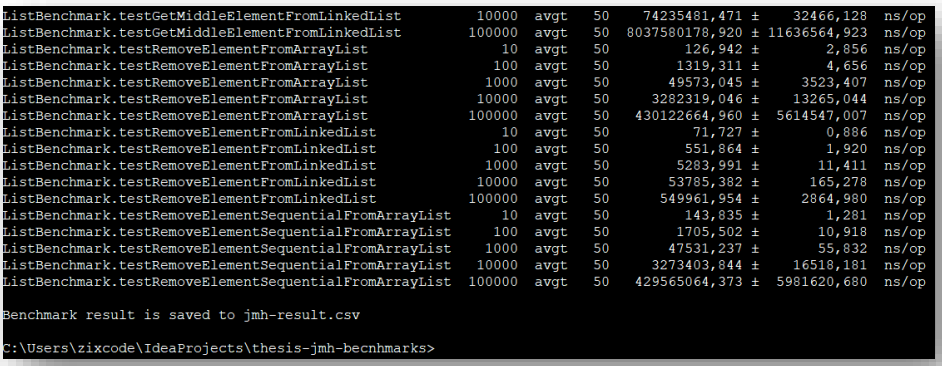
Εικόνα 22 : Maven run tests command

Κατά τη διάρκεια της εκτέλεσης είναι καλό να μην υπάρχουν καθόλου απώλειες σε resources, οπότε ο υπολογιστής τρέχει μόνο το συγκεκριμένο benchmark, ενώ παράλληλα επιστρέφει feedback στην κονσόλα με την πρόοδο.



Εικόνα 23 : Benchmarks Progress screen.

Μόλις ολοκληρωθεί θα επιστρέψει ένα συγκεντρωτικό πίνακα από τα σενάρια εκτέλεσης, ενώ με το όρισμα που περάστηκε στην κονσόλα επέστρεψε και όλα τα αποτελέσματα σε αρχείο σε μορφή csv.



Εικόνα 3 : Execution end of benchmarks screen.

5. Αποτελέσματα - Συμπεράσματα

5.1 Interface List : Σενάρια εκτέλεσης & συμπεράσματα

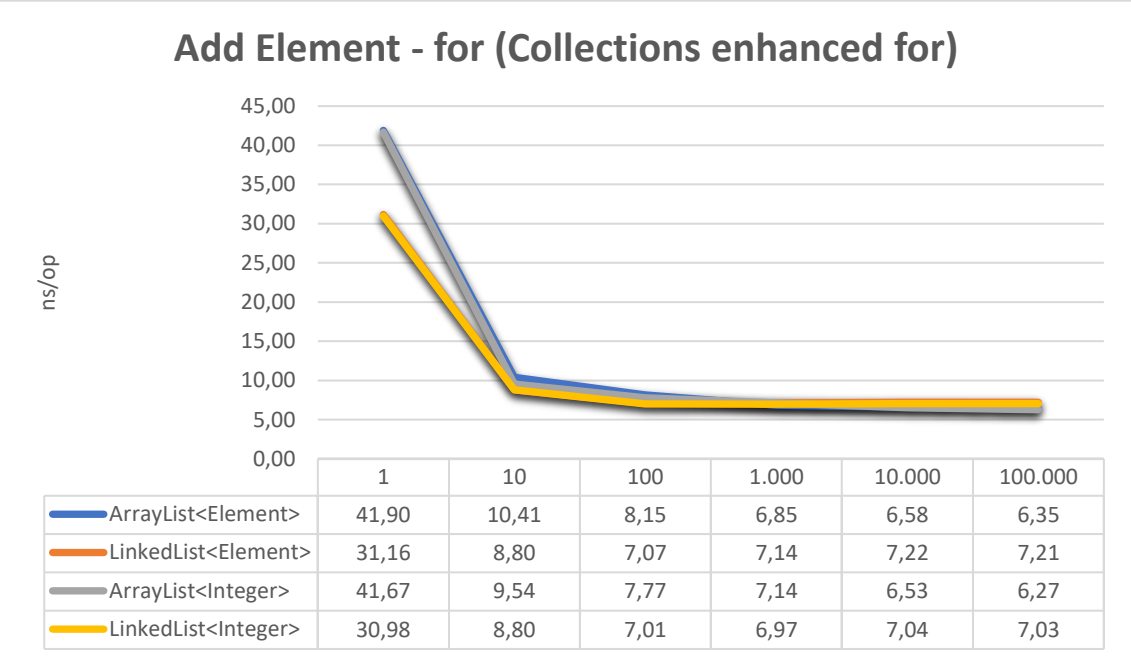
5.1.1 Σενάριο εκτέλεσης : Εισαγωγή στοιχείου με ενισχυμένο επαναληπτικό βρόχο (Collections enhanced for loop).

Στο σενάριο αυτό έγινε εισαγωγή του πλήθους των στοιχείων σε κενή δομή.

```
/**
 * Create Operations.<br>
 * Add element to ArrayList.
 *
 * @param blackhole eliminates dead code.
 */
@Benchmark
public void testAddElementToArrayList(Blackhole blackhole) {
  for (Element e : elements) {
    blackhole.consume(emptyArraylist.add(e));
  }
}
```

Συμπεράσματα : Η αρχική παρατήρηση έχει να κάνει με την εισαγωγή σε δομή του ενός και μόνο στοιχείου. Η διαδικασία της εισαγωγής στο ArrayList κοστίζει παραπάνω για ένα και μόνο στοιχείο και αυτό φαίνεται στο χρόνο (≈ 10 ns Δ) σε σχέση με τη LinkedList και φυσικά πρέπει να αναφερθεί και το σταθερό κόστος της διαδικασίας (έναρξη, τερματισμός) του επαναληπτικού βρόχου για όλες τις δομές.

Όμως όσο το πλήθος αυξάνεται τόσο μειώνεται ο χρόνος εισαγωγής στο ArrayList, με την επικράτηση του ArrayList σε πλήθος άνω των 1000 στοιχείων. Μια δεύτερη παρατήρηση είναι ότι δεν υπάρχει σημαντική διαφορά σε χρόνο ανάμεσα στους δύο τύπους δεδομένων , Element & Integer.



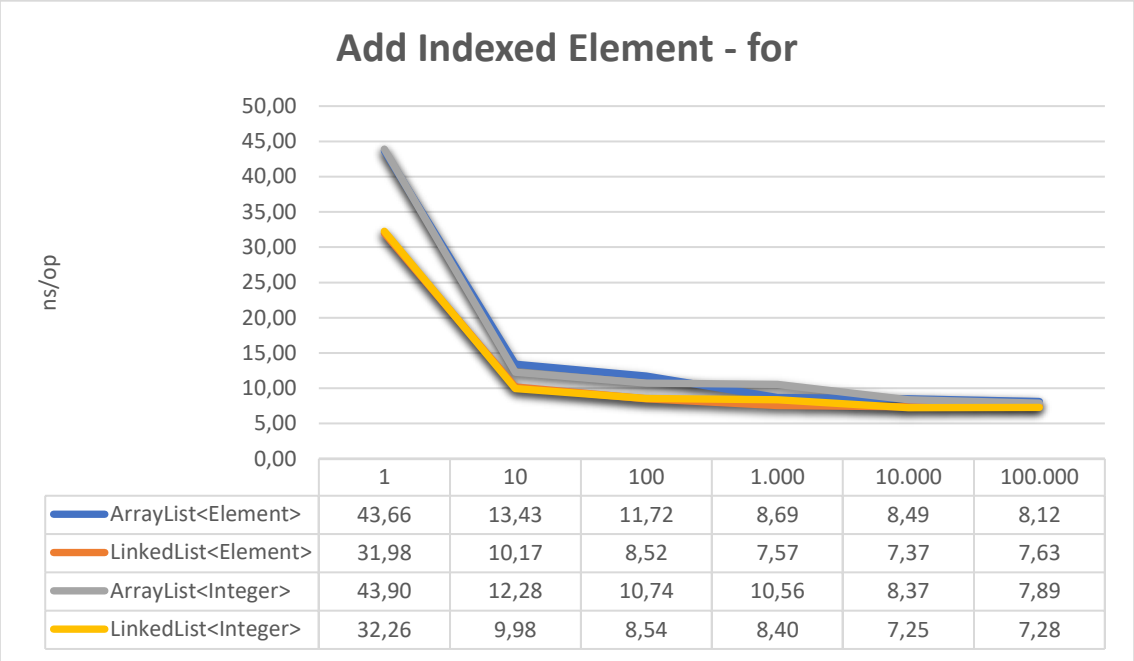
	1	10	100	1.000	10.000	100.000
ArrayList<Element>	41,90	10,41	8,15	6,85	6,58	6,35
LinkedList<Element>	31,16	8,80	7,07	7,14	7,22	7,21
ArrayList<Integer>	41,67	9,54	7,77	7,14	6,53	6,27
LinkedList<Integer>	30,98	8,80	7,01	6,97	7,04	7,03

Βάσει των παραπάνω μετρήσεων φαίνεται να επιβεβαιώνεται ο ισχυρισμός της επιμερισμένης πολυπλοκότητας σε σταθερό κόστος ίσο του O(1).

5.1.2 **Σενάριο εκτέλεσης : Εισαγωγή ταξινομημένου στοιχείου με απλό επαναληπτικό βρόχο (for loop).**

Στο σενάριο αυτό έγινε εισαγωγή του πλήθους των στοιχείων σε κενή δομή.

```
/**
 * Create Operations.<br>
 * Add indexed element to ArrayList.
 *
 * @param blackhole eliminates dead code.
 */
@Benchmark
public void testAddIndexedElementToArrayList(Blackhole blackhole) {
  for (int i = 0; i < elements.length; i++) {
    emptyArraylist.add(i, elements[i]);
    blackhole.consume(elements[i]);
  }
}
```



	1	10	100	1.000	10.000	100.000
ArrayList<Element>	43,66	13,43	11,72	8,69	8,49	8,12
LinkedList<Element>	31,98	10,17	8,52	7,57	7,37	7,63
ArrayList<Integer>	43,90	12,28	10,74	10,56	8,37	7,89
LinkedList<Integer>	32,26	9,98	8,54	8,40	7,25	7,28

Συμπεράσματα : Αντίστοιχοι χρόνοι με την απλή εισαγωγή στοιχείων. Ξανά το ArrayList σε δομή με ένα στοιχείο δείχνει το κόστος της μοναδικής εισαγωγής στοιχείου. Όσο όμως το πλήθος των στοιχείων ανεβαίνει , επιμερίζεται ολοένα και περισσότερο κάνοντας τη δομή πιο αποδοτική. Και οι δύο δομές δείχνουν σταθερότητα και συνεχή βελτιστοποίηση σε χρόνους όσο τα πλήθη μεγαλώνουν.

Βάσει των παραπάνω μετρήσεων φαίνεται να επιβεβαιώνεται ο ισχυρισμός της επιμερισμένης πολυπλοκότητας για την εισαγωγή με ταξινόμηση σε σταθερό κόστος ίσο του O(1).

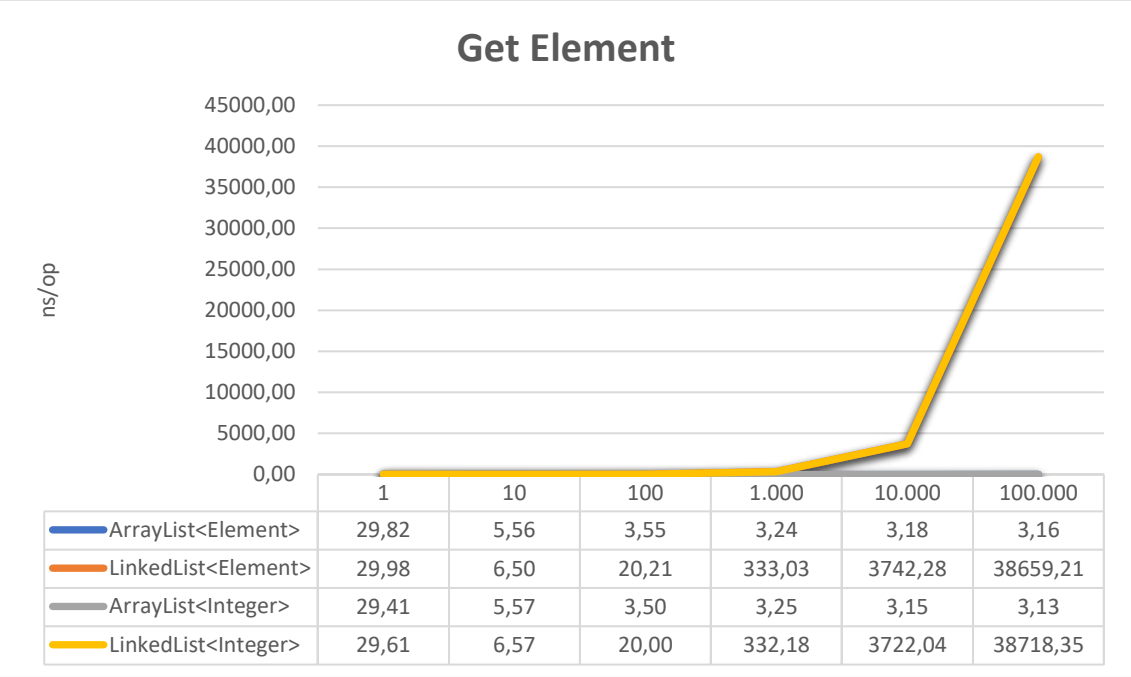
5.1.3 **Σενάριο εκτέλεσης : Εύρεση στοιχείου στο πλήθος της δομής.**

Στο σενάριο αυτό έγινε αναζήτηση ενός στοιχείου σε πλήρη δομή.

```java
/**
 * Retrieve Operations.<br>
 * Get element from ArrayList.
 *
 * @param blackhole eliminates dead code.
 */
@Benchmark
public void testGetElementFromArrayList(Blackhole blackhole) {
  for (int i = 0; i < elements.length; i++) {
    blackhole.consume(fullArraylist.get(i));
  }
  blackhole.consume(fullArraylist);
}
```

Συμπεράσματα : Το ArrayList και στην αναζήτηση υπερέχει κατά κράτος λόγω τυχαίας προσπέλασης (Random Access). Η LinkedList (υπενθυμίζεται ότι υλοποιείται ως διπλά συνδεδεμένη λίστα) πρέπει να διασχίσει όλα τα προηγούμενα στοιχεία για να φτάσει στη θέση που ζητείται, με το αντίστοιχο κόστος.

Εντυπωσιακό είναι το γεγονός ότι το ArrayList συνεχίζει να «πληρώνει» το κόστος αρχικοποίησης ενώ όσο το πλήθος των στοιχείων αυξάνεται το κόστος αναζήτησης παραμένει σταθερό μικρό.



	1	10	100	1.000	10.000	100.000
ArrayList<Element>	29,82	5,56	3,55	3,24	3,18	3,16
LinkedList<Element>	29,98	6,50	20,21	333,03	3742,28	38659,21
ArrayList<Integer>	29,41	5,57	3,50	3,25	3,15	3,13
LinkedList<Integer>	29,61	6,57	20,00	332,18	3722,04	38718,35

Επιβεβαιώνεται η επιμερισμένη πολυπλοκότητα σε σταθερό κόστος O(1) για το ArrayList, κάτι που ισχύει και για τη LinkedList μέχρι πλήθος N = 100, ανεξαρτήτως του τύπου του στοιχείου.

Για πλήθος από N =1000 και άνω το κόστος της αναζήτησης στη LinkedList φαίνεται να αυξάνεται με εκθετικό ρυθμό.

5.1.4 **Σενάριο εκτέλεσης : Εύρεση στοιχείου στο μέσο της δομής.**

Στο σενάριο αυτό έγινε αναζήτηση του μεσαίου στοιχείου σε πλήρη δομή.

```
/**
 * Retrieve Operations.<br>
 * Get middle element from ArrayList.
 *
 * @param blackhole eliminates dead code.
 */
@Benchmark
public void testGetMiddleElementFromArrayList(Blackhole blackhole) {
  for (int i = 0; i < elements.length; i++) {
    blackhole.consume(fullArraylist.get(N / 2));
  }
}
```

Συμπεράσματα : Το ArrayList δεν διαφοροποιείται σχεδόν καθόλου στην αναζήτηση του μεσαίου στοιχείου από το πρώτο, ξανά λόγω της τυχαίας προσπέλασης. Η LinkedList «πληρώνει» μεγάλο κόστος στη διάσχιση των στοιχείων (υπενθυμίζεται ότι υλοποιείται ως διπλά συνδεδεμένη λίστα), καθώς πρέπει να διασχίσει όλα τα προηγούμενα στοιχεία για να φτάσει στη θέση που ζητείται, με το αντίστοιχο κόστος.

	1	10	100	1.000	10.000	100.000
ArrayList<Element>	28,17	5,89	3,92	3,58	3,49	3,48
LinkedList<Element>	28,26	8,49	43,33	688,48	7440,70	80078,97
ArrayList<Integer>	28,25	5,96	4,04	4,13	3,51	3,49
LinkedList<Integer>	28,30	8,24	43,04	687,69	7432,02	80308,60

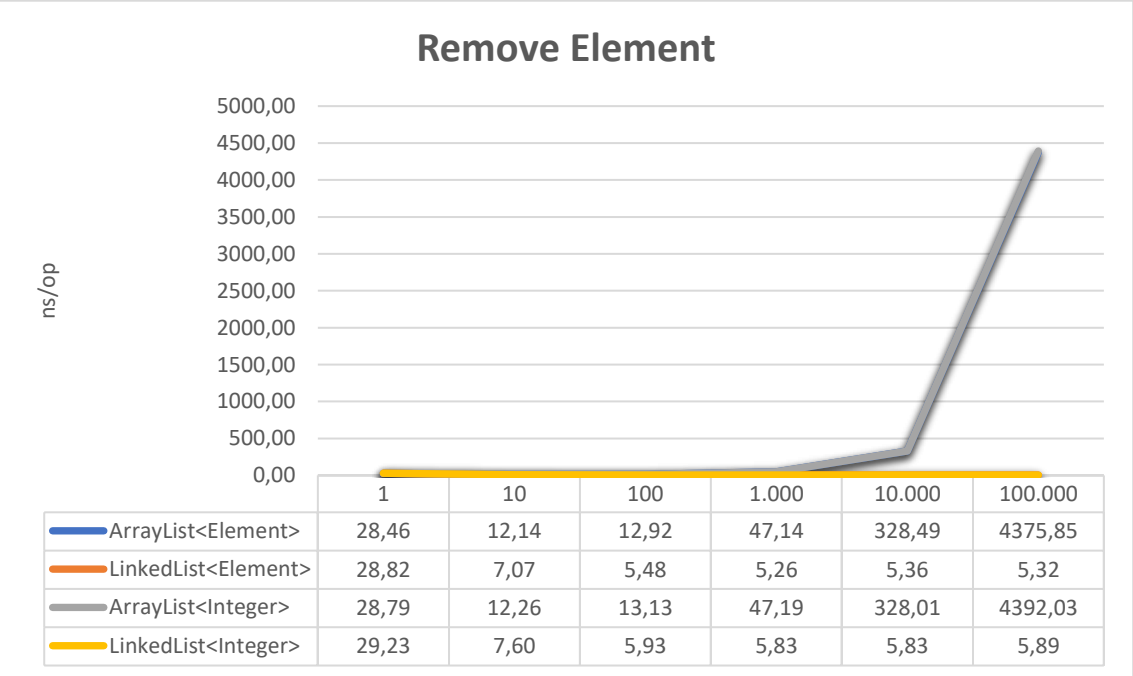
Και εδώ επιβεβαιώνεται η επιμερισμένη πολυπλοκότητα σταθερού κόστους O(1) για το ArrayList, κάτι που ισχύει και για τη LinkedList μέχρι πλήθος N = 100, ανεξαρτήτως του τύπου του στοιχείου. Για πλήθος από N =1000 και άνω το κόστος της αναζήτησης στη LinkedList φαίνεται να αυξάνεται με εκθετικό ρυθμό.

5.1.5 **Σενάριο εκτέλεσης : Διαγραφή στοιχείου από τη δομή.**

Στο σενάριο αυτό έγινε διαγραφή στοιχείου σε πλήρη δομή.

```
/**
 * Remove Operations.<br>
 * Remove element from ArrayList.
 *
 * @param blackhole eliminates dead code.
 */
@Benchmark
public void testRemoveElementFromArrayList(Blackhole blackhole) {
  for (int i = 0; i < N; i++) {
    blackhole.consume(fullArraylist.remove(elements[i]));
  }
}
```

Συμπεράσματα : Το ArrayList λόγω της μεταβολής του μήκους «πληρώνει» το κόστος αυτό σε χρόνο με εκθετική αύξηση, ενώ η LinkedList έχει σταθερό κόστος καθώς δεν αλλάζει παρά μόνο τους δείκτες σε προηγούμενα και επόμενα στοιχεία (εφόσον υπάρχουν).



	1	10	100	1.000	10.000	100.000
ArrayList<Element>	28,46	12,14	12,92	47,14	328,49	4375,85
LinkedList<Element>	28,82	7,07	5,48	5,26	5,36	5,32
ArrayList<Integer>	28,79	12,26	13,13	47,19	328,01	4392,03
LinkedList<Integer>	29,23	7,60	5,93	5,83	5,83	5,89

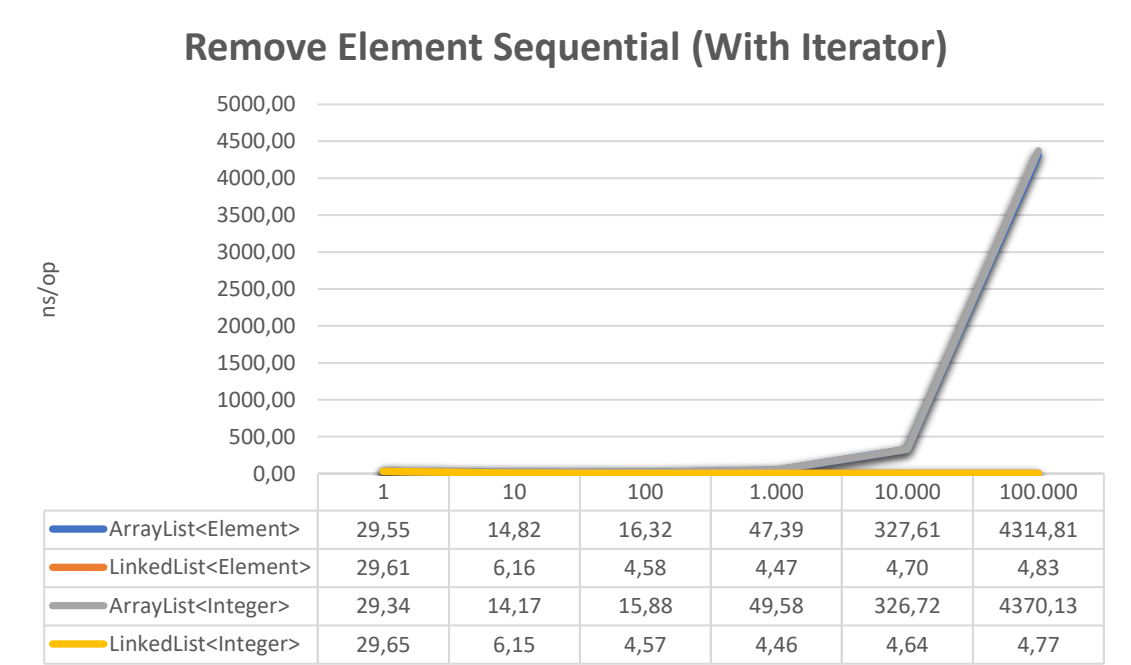
Σε επίπεδο επιμερισμένης πολυπλοκότητας η LinkedList φαίνεται να επιβεβαιώνει αυτό που εγγυάται : σταθερό κόστος της λειτουργίας τάξης O(1) ενώ για το ArrayList επιβεβαιώνεται κάτι τέτοιο μόνο σε πλήθη μέχρι N = 1000. Σε μεγαλύτερα πλήθη (από N > 1000) δεν μπορεί να επιβεβαιωθεί.

5.1.6 **Σενάριο εκτέλεσης : Διαγραφή στοιχείου κατά σειρά με Iterator.**

Στο σενάριο αυτό έγινε διαγραφή στοιχείου σε πλήρη δομή με τη χρήση Iterator.

```
/**
 * Remove Operations.<br>
 * Remove sequential with iterator element from ArrayList.
 */
@Benchmark
public void testRemoveElementSequentialFromArrayList() {
  while (iteratorArraylist.hasNext()) {
    iteratorArraylist.next();
    iteratorArraylist.remove();
  }
}
```

Συμπεράσματα : Αντίστοιχα αποτελέσματα και εδώ για το ArrayList. Υπενθύμιση ότι η χρήση του Iterator ενδείκνυται για synchronized δομές. Παράλληλα παρατηρείται ότι είναι οριακά ταχύτερη η λειτουργία με τη χρήση του Iterator. Εξακολουθεί όμως το ArrayList να έχει κόστος που αυξάνεται εκθετικά σε μεγάλα N. Η LinkedList έχει σαφέστατα πολύ χαμηλό κόστος όταν πρόκειται για εισαγωγή - αφαίρεση στοιχείου. Δεν «πληρώνει» κόστος μεταβολής μήκους καθώς αλλάζει μόνον τους δείκτες του στοιχείου (προηγούμενο – επόμενο).



	1	10	100	1.000	10.000	100.000
ArrayList<Element>	29,55	14,82	16,32	47,39	327,61	4314,81
LinkedList<Element>	29,61	6,16	4,58	4,47	4,70	4,83
ArrayList<Integer>	29,34	14,17	15,88	49,58	326,72	4370,13
LinkedList<Integer>	29,65	6,15	4,57	4,46	4,64	4,77

Η επιμερισμένη πολυπλοκότητα επιβεβαιώνεται για τη LinkedList με σταθερό κόστος O(1) ενώ και στην περίπτωση της χρήσης του Iterator στο ArrayList επιβεβαιώνεται κάτι τέτοιο μόνο σε πλήθη μέχρι N = 1000. Σε μεγαλύτερα πλήθη (από N > 1000) και εδώ δεν μπορεί να επιβεβαιωθεί.

5.1.7 **Σενάριο εκτέλεσης : Έλεγχος στοιχείου αν περιέχεται στη δομή.**

Στο σενάριο αυτό έγινε έλεγχος αν ένα στοιχείο περιέχεται μέσα στη δομή.

```
/**
 * Contains Operations.<br>
 * ArrayList contains Element.
 *
 * @param blackhole eliminates dead code.
 */
@Benchmark
public void testContainsElementInArrayList(Blackhole blackhole) {
  for (int i = 0; i < N; i++) {
    blackhole.consume(fullArraylist.contains(elements[i]));
  }
}
```

Συμπεράσματα : Αυτή αποτελεί και την πιο κοστοβόρα διαδικασία για τις δύο συγκεκριμένες δομές , καθώς δεν υπάρχει άλλος τρόπος παρά να σαρωθεί όλη η δομή και κάθε φορά να καλείται η equals() πάνω στα δυο στοιχεία ώστε να επιστρέψει true ή false για τη σύγκριση. Σε πλήθη μέχρι και N = 100 το κόστος της πράξης είναι μέσα σε αποδεκτά πλαίσια. Για N = 1.000 αρχίζει πια να αυξάνεται αρκετά και από N = 10.000 έχουμε πλήρη διαφοροποίηση ανάμεσα στους δύο τύπους στοιχείων. Για τον τύπο Element το κόστος αυξάνεται με εκθετικό βαθμό, απόρροια των αλλεπάλληλων κλήσεων & πιο πολύπλοκων συγκρίσεων. (υπενθυμίζεται ότι η equals() έχει γίνει override ώστε να συγκρίνεται κάθε πεδίο στο Element, ώστε να δίνει και μοναδικό hash).

	1	10	100	1.000	10.000	100.000
ArrayList<Element>	27,65	11,77	54,49	677,33	12587,45	427678,86
LinkedList<Element>	27,49	13,39	121,23	1286,64	15912,34	470270,72
ArrayList<Integer>	28,14	9,95	35,78	348,40	3390,48	35679,81
LinkedList<Integer>	27,80	12,13	104,26	964,86	9873,13	99786,11

Επιμερισμένη πολυπλοκότητα εδώ σε επίπεδο σταθερού κόστους , ειδικά της τάξεως του O(1) δεν δύναται να υπάρξει. Όπως αναφέρθηκε και προηγουμένως , είναι το πιο απαιτητικό σενάριο από πλευράς κόστους για τις δύο δομές.

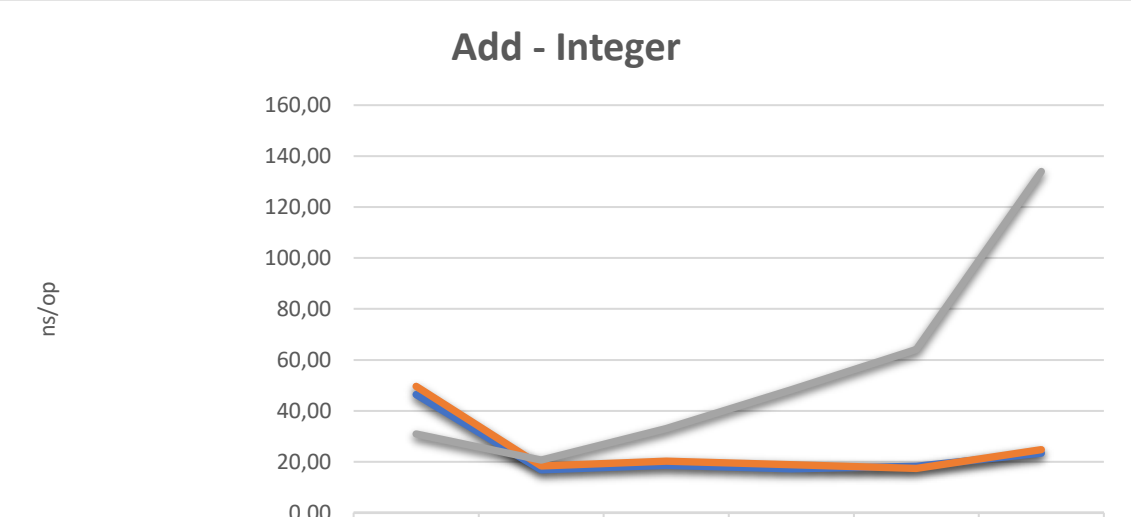
5.2 Interface Set : Σενάρια εκτέλεσης & συμπεράσματα

5.2.1 Σενάριο εκτέλεσης : Εισαγωγή στοιχείου με απλό επαναληπτικό βρόχο (for loop).

Στο σενάριο αυτό έγινε εισαγωγή του πλήθους των στοιχείων σε κενή δομή.

```
/**
 * Create Operations.<br>
 * Add element to HashSet.
 *
 * @param blackhole eliminates dead code.
 */
@Benchmark
public void testAddElementToHashSet(Blackhole blackhole) {
  for (int i = 0; i < N; i++) {
    blackhole.consume(emptyHashSet.add(elements[i]));
  }
}
```

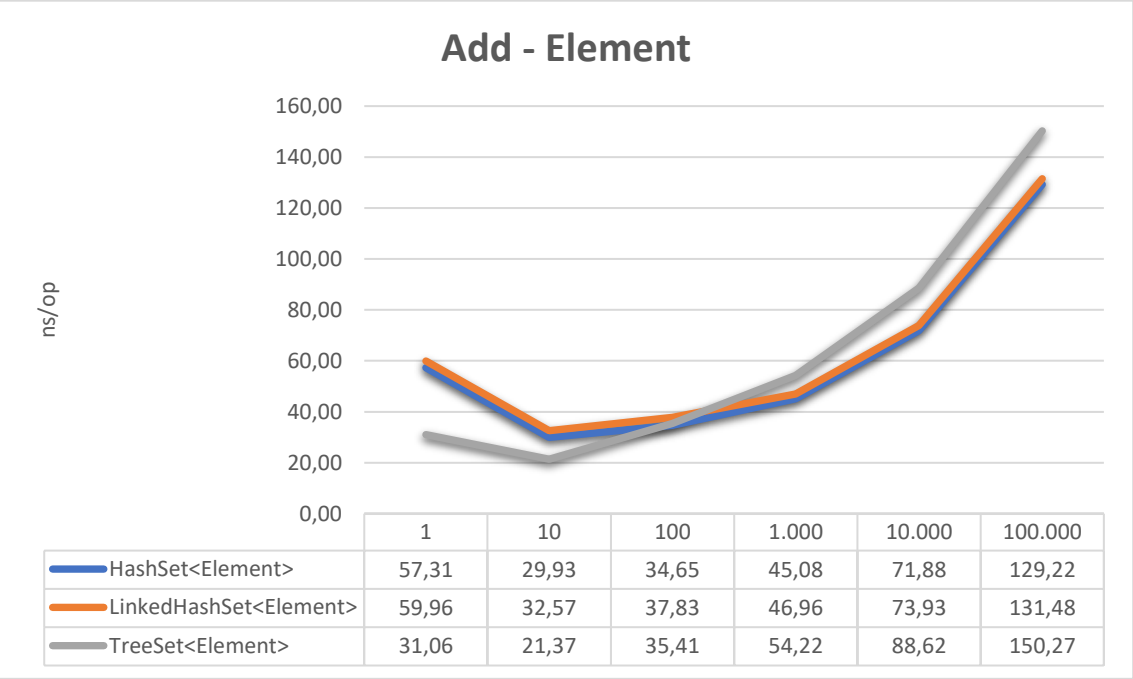
Συμπεράσματα : Για δομή ενός στοιχείου παρατηρείται ότι το TreeSet είναι πιο γρήγορο από τις άλλες δυο δομές. Καθώς το πλήθος μεγαλώνει, και οι τρείς δομές διατηρούν το κόστος εισαγωγής σε αρκετά χαμηλό και σχετικά σταθερό επίπεδο, ιδιαίτερα το LinkedHashSet που διατηρεί τη σειρά εισαγωγής των στοιχείων του είναι εντυπωσιακό. Το TreeSet ως δενδρική δομή «πληρώνει» ένα ελάχιστα μεγαλύτερο κόστος στην εισαγωγή, λόγω της εξισορρόπησης που κάνει για κάθε εισαγωγή.



	1	10	100	1.000	10.000	100.000
HashSet<Integer>	46,38	16,83	18,60	17,22	18,21	23,36
LinkedHashSet<Integer>	49,61	18,29	20,29	18,80	17,39	24,70
TreeSet<Integer>	30,92	20,67	33,01	48,32	64,04	133,96

Η επιμερισμένη πολυπλοκότητα για σταθερό κόστος της τάξεως του O(1) φαίνεται να επιβεβαιώνεται απόλυτα στην περίπτωση του τύπου Integer, καθώς ακόμη και σε πολύ μεγάλα N (= 100.000) καθώς το κόστος είναι σχεδόν σταθερό.

Στην περίπτωση του τύπου Element η σχετικά μικρή αύξηση σε κόστος εξηγείται από το κόστος που πληρώνουν οι δομές στις equals() και hashCode(), καθώς έχουν γίνει override με έλεγχο σε όλα τα πεδία του κάθε στοιχείου, ώστε να διασφαλιστεί ότι δεν θα έχουμε κάποιο hash collision.



	1	10	100	1.000	10.000	100.000
HashSet<Element>	57,31	29,93	34,65	45,08	71,88	129,22
LinkedHashSet<Element>	59,96	32,57	37,83	46,96	73,93	131,48
TreeSet<Element>	31,06	21,37	35,41	54,22	88,62	150,27

Όμως ακόμη και έτσι το κόστος και για αυτό τον πιο σύνθετο τύπο στοιχείου είναι αποδεκτό καθώς διατηρείται πολύ χαμηλό σε πολύ μεγάλα N. Το TreeSet εξακολουθεί να «πληρώνει» ανεξαρτήτως στοιχείου και το κόστος της εξισορρόπησης της δενδρικής δομής.

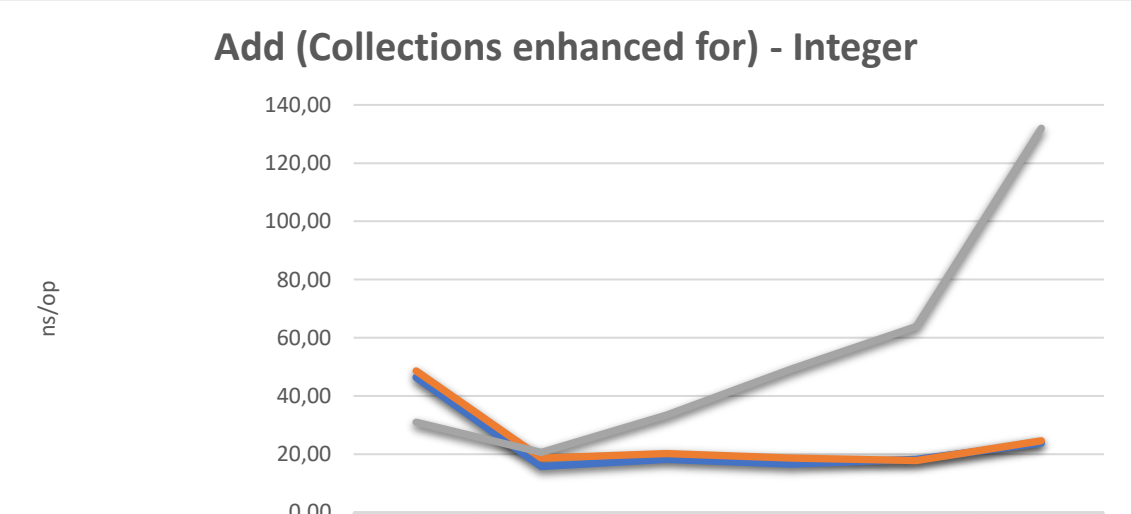
Η επιμερισμένη πολυπλοκότητα για σταθερό κόστος της τάξεως του O(1) φαίνεται να επιβεβαιώνεται στην περίπτωση του τύπου Element για πλήθη N = 1000, καθώς η αύξηση του κόστους είναι σχεδόν αμελητέα. Σε μεγάλα πλήθη (N>= 10.000) ίσως μπορεί να θεωρηθεί και εδώ αποδεκτό καθώς σε αξιακό επίπεδο το κόστος θεωρείται σχεδόν σταθερό και αρκετά χαμηλό.

5.2.2 Σενάριο εκτέλεσης : Εισαγωγή στοιχείου με ενισχυμένο επαναληπτικό βρόχο (Collections enhanced for loop).

Στο σενάριο αυτό έγινε εισαγωγή του πλήθους των στοιχείων σε κενή δομή.

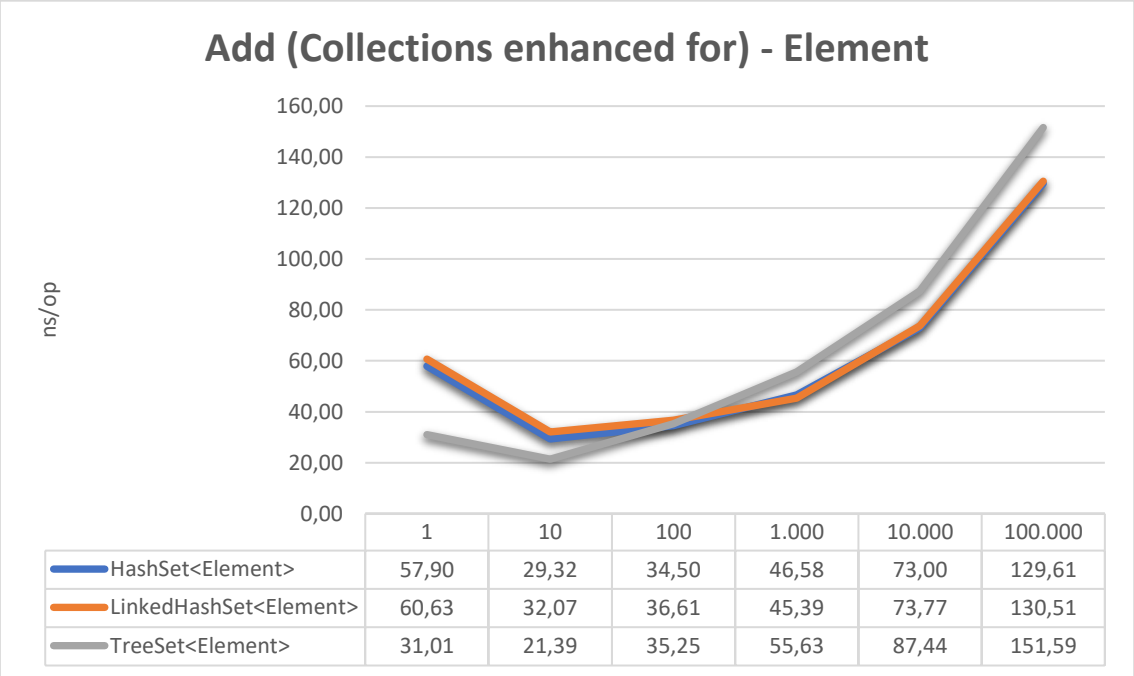
```
/**
 * Create Operations.<br>
 * Add foreach element to HashSet.
 *
 * @param blackhole eliminates dead code.
 */
@Benchmark
public void testAddElementForEachToHashSet(Blackhole blackhole) {
  for (Element e : elements) {
    blackhole.consume(emptyHashSet.add(e));
  }
}
```

Συμπεράσματα : Και εδώ οι διαφορές είναι ελάχιστες με την απλή for, καθώς και οι τρείς δομές έχουν πανομοιότυπη συμπεριφορά και απόδοση και στους δύο τύπους δεδομένων.



	1	10	100	1.000	10.000	100.000
HashSet<Integer>	46,41	15,79	18,27	16,57	18,20	23,85
LinkedHashSet<Integer>	48,62	18,69	20,22	18,65	17,83	24,66
TreeSet<Integer>	31,01	20,62	33,39	49,23	63,93	131,98

Σε επίπεδο επιμερισμένης πολυπλοκότητας φαίνεται και σε αυτή την περίπτωση να γίνεται αποδεκτό το κόστος σε σταθερό χρόνο τάξεως του O(1) για τύπο Integer στοιχείου. Ακόμη και η LinkedList έχει πολύ χαμηλό κόστος σε N > 10.000.



	1	10	100	1.000	10.000	100.000
HashSet<Element>	57,90	29,32	34,50	46,58	73,00	129,61
LinkedHashSet<Element>	60,63	32,07	36,61	45,39	73,77	130,51
TreeSet<Element>	31,01	21,39	35,25	55,63	87,44	151,59

Η επιμερισμένη πολυπλοκότητα για σταθερό κόστος της τάξεως του O(1) φαίνεται να επιβεβαιώνεται και εδώ στην περίπτωση του τύπου Element για πλήθη N = 1000, καθώς η αύξηση του κόστους είναι σχεδόν αμελητέα. Σε μεγάλα πλήθη (N>= 10.000) ίσως μπορεί να θεωρηθεί και εδώ αποδεκτό καθώς σε αξιακό επίπεδο το κόστος θεωρείται σχεδόν σταθερό και αρκετά χαμηλό.

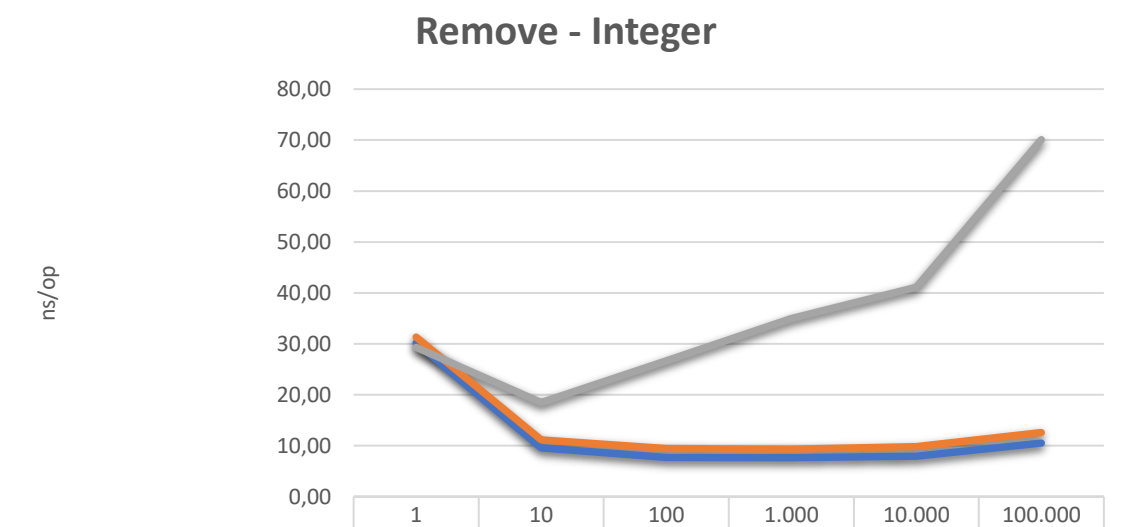
5.2.3 **Σενάριο εκτέλεσης : Διαγραφή στοιχείου από τη δομή.**

Στο σενάριο αυτό έγινε διαγραφή στοιχείου σε πλήρη δομή.

```
/**
 * Remove Operations.<br>
 * Remove element from HashSet.
 *
 * @param blackhole eliminates dead code.
 */
@Benchmark
public void testRemoveElementFromHashSet(Blackhole blackhole) {
  for (int i = 0; i < N; i++) {
    blackhole.consume(fullHashSet.remove(elements[i]));
  }
}
```

Συμπεράσματα : Η επίδοση και των τριών δομών σε τύπο ακόμη και σε τύπο στοιχείου Element, κρίνεται ως εξαιρετική. Αρκεί να υπενθυμίσουμε ότι η ταχύτητα διαγραφής του ArrayList σε N > 100.000 ξεπερνά τα 4000 ns , ενώ εδώ οι επιδόσεις προσεγγίζουν αυτές της LinkedList.

Επιβεβαιώνεται το επιπλέον κόστος εξισορρόπησης του TreeSet , με το ελαφρώς μεγαλύτερο κόστος σε επίδοση.



	1	10	100	1.000	10.000	100.000
HashSet<Integer>	30,19	9,56	7,71	7,66	7,94	10,51
LinkedHashSet<Integer>	31,33	11,13	9,45	9,33	9,81	12,60
TreeSet<Integer>	29,28	18,49	26,66	34,94	41,11	70,08

	1	10	100	1.000	10.000	100.000
HashSet<Element>	41,78	22,49	20,25	23,53	42,42	57,30
LinkedHashSet<Element>	42,85	24,31	22,06	25,98	46,48	67,65
TreeSet<Element>	29,44	19,46	27,05	34,35	72,59	98,77

Σε επίπεδο επιμερισμένης πολυπλοκότητας , μπορεί να επιβεβαιωθεί ότι οι δομές έχουν σταθερό κόστος τάξεως του O(1).

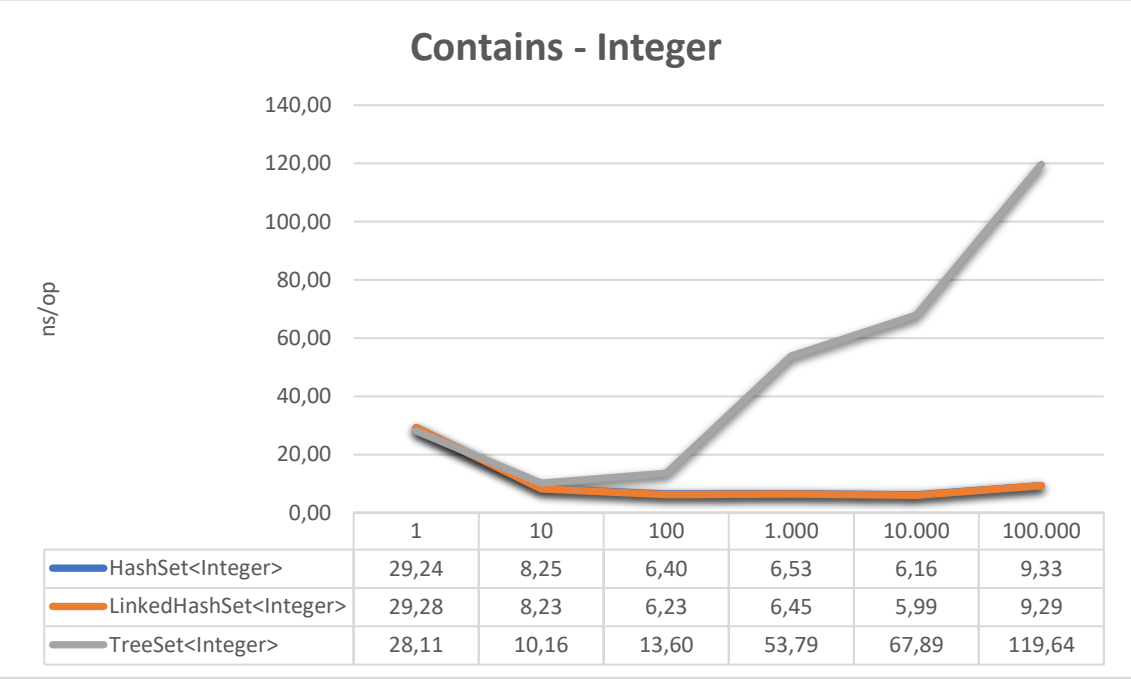
5.2.4 **Σενάριο εκτέλεσης : Έλεγχος στοιχείου αν περιέχεται στη δομή.**

Στο σενάριο αυτό έγινε έλεγχος στοιχείου αν περιέχεται σε πλήρη δομή.

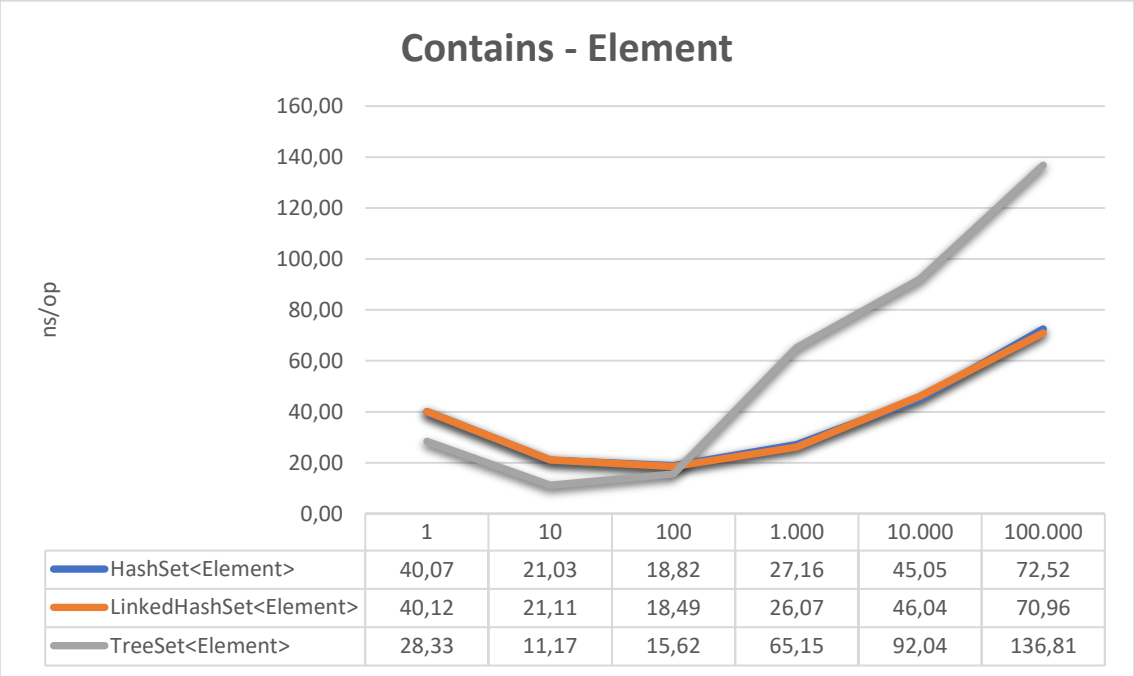
```
/**
 * Contains Operations.<br>
 * HashSet contains Element : True.
 *
 * @param blackhole eliminates dead code.
 */
@Benchmark
public void testContainsInHashSet(Blackhole blackhole) {
  for (int i = 0; i < N; i++) {
    blackhole.consume(fullHashSet.contains(elements[i]));
  }
}
```

Συμπεράσματα : Εδώ τα αποτελέσματα διαφοροποιούνται βάσει του τύπου του στοιχείου. Για τύπο στοιχείου Integer οι HashSet & LinkedHashSet έχουν παρόμοια συμπεριφορά και σταθερό κόστος ανα πράξη ανεξαρτήτως πλήθους (Εξαιρείται πάντοτε η δομή με ένα και μόνο στοιχείο για λόγους που έχουν προαναφερθεί). Το TreeSet «πληρώνει» το γνωστό κόστος της δενδρικής αναζήτησης, χωρίς όμως αυτό να είναι ιδιαίτερα υψηλό, απλώς είναι υψηλότερο των υπολοίπων.

Για τύπο στοιχείου Element , και οι τρείς δομές παρουσιάζουν μια αυξητική τάση, η οποία όμως σε σύγκριση με την αύξηση του πλήθους κρίνεται πολύ μικρή (σε απόλυτο νούμερο).



	1	10	100	1.000	10.000	100.000
HashSet<Integer>	29,24	8,25	6,40	6,53	6,16	9,33
LinkedHashSet<Integer>	29,28	8,23	6,23	6,45	5,99	9,29
TreeSet<Integer>	28,11	10,16	13,60	53,79	67,89	119,64



	1	10	100	1.000	10.000	100.000
HashSet<Element>	40,07	21,03	18,82	27,16	45,05	72,52
LinkedHashSet<Element>	40,12	21,11	18,49	26,07	46,04	70,96
TreeSet<Element>	28,33	11,17	15,62	65,15	92,04	136,81

Η επιμερισμένη πολυπλοκότητα για σταθερό κόστος της τάξεως του O(1) μπορεί να επιβεβαιωθεί σχετικά.

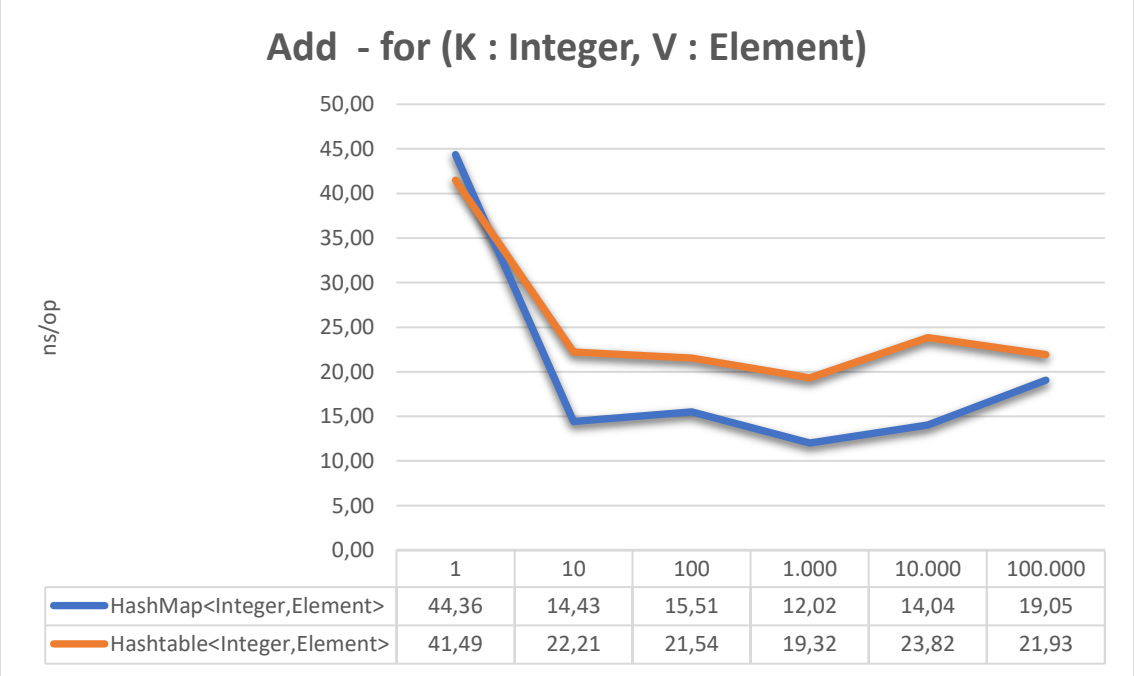
5.3 Interface Map : Σενάρια εκτέλεσης & συμπεράσματα

5.3.1 Σενάριο εκτέλεσης : Εισαγωγή στοιχείου με απλό επαναληπτικό βρόχο (for loop).

Στο σενάριο αυτό έγινε εισαγωγή του πλήθους των στοιχείων σε κενή δομή.

```
/**
 * Create Operations.<br>
 * Add element to HashMap.
 *
 * @param blackhole eliminates dead code.
 */
@Benchmark
public void testAddElementToHashMap(Blackhole blackhole) {
  for (int i = 0; i < N; i++) {
    blackhole.consume(emptyHashMap.put(integers[i], elements[i]));
  }
}
```

Συμπεράσματα : Και οι δύο δομές επιδεικνύουν εξαιρετικές επιδόσεις ακόμη και σε πολύ μεγάλα πλήθη με N = 100.000 με υποδειγματική σταθερότητα σε κόστος λειτουργίας. Ένα μικρό προβάδισμα παρατηρείται στο HashMap καθώς δεν χρειάζεται να διατηρεί την σειρά εισαγωγής των στοιχείων , σε αντίθεση με το Hashtable, και φυσικά δεν είναι synchronized οπότε και δικαιολογεί το χαμηλότερο κόστος της πράξης.



	1	10	100	1.000	10.000	100.000
HashMap<Integer,Element>	44,36	14,43	15,51	12,02	14,04	19,05
Hashtable<Integer,Element>	41,49	22,21	21,54	19,32	23,82	21,93

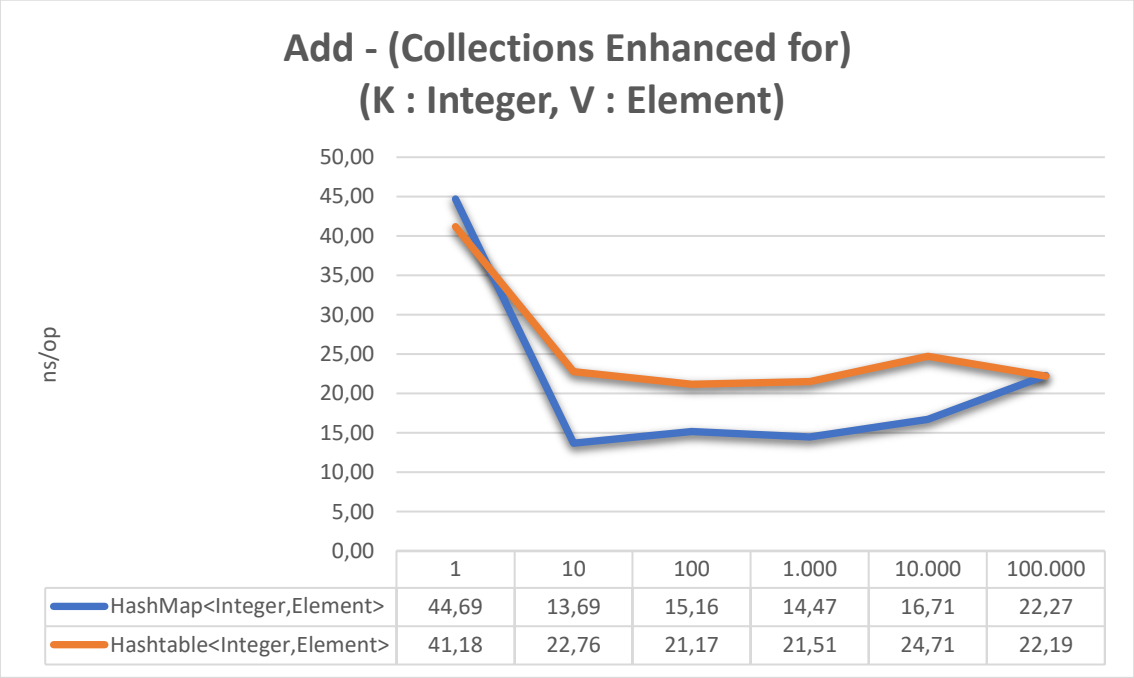
Η επιμερισμένη πολυπλοκότητα επιβεβαιώνεται με σταθερό κόστος της τάξης του O(1) ακόμα και για N = 100.000 και στις δύο δομές.

5.3.2 Σενάριο εκτέλεσης : Εισαγωγή στοιχείου με ενισχυμένο επαναληπτικό βρόχο (Collections enhanced for loop).

Στο σενάριο αυτό έγινε εισαγωγή του πλήθους των στοιχείων σε κενή δομή.

```
/**
 * Create Operations.<br>
 * Add enchanced for element to HashMap.
 *
 * @param blackhole eliminates dead code.
 */
@Benchmark
public void testAddElementForEachToHashMap(Blackhole blackhole) {
  for (Integer i : integers) {
    blackhole.consume(emptyHashMap.put(i, elements[i]));
  }
}
```

Συμπεράσματα : Στο αντίστοιχο σενάριο με ενισχυμένη for οι δύο δομές επιδεικνύουν πανομοιότυπη συμπεριφορά. Απειροελάχιστες διαφορές της τάξης των 2 ns ανά πράξη δεν θεωρούνται ότι επηρεάζουν σημαντικά το κόστος της εισαγωγής. Οι επιδόσεις παραμένουν στα ίδια επίπεδα με την απλή for ακόμη και σε πολύ μεγάλα πλήθη με N = 100.000 με υποδειγματική σταθερότητα σε κόστος λειτουργίας. Το ίδιο ελάχιστο προβάδισμα παρατηρείται στο HashMap καθώς δεν χρειάζεται να διατηρεί την σειρά εισαγωγής των στοιχείων , σε αντίθεση με το Hashtable, και καθώς δεν είναι synchronized οπότε και δικαιολογεί το χαμηλότερο κόστος της πράξης.



	1	10	100	1.000	10.000	100.000
HashMap<Integer,Element>	44,69	13,69	15,16	14,47	16,71	22,27
Hashtable<Integer,Element>	41,18	22,76	21,17	21,51	24,71	22,19

Και εδώ η επιμερισμένη πολυπλοκότητα επιβεβαιώνεται με σταθερό κόστος της τάξης του O(1) ακόμα και για N = 100.000 και στις δύο δομές.

5.3.3 **Σενάριο εκτέλεσης : Εύρεση στοιχείου στο πλήθος της δομής.**

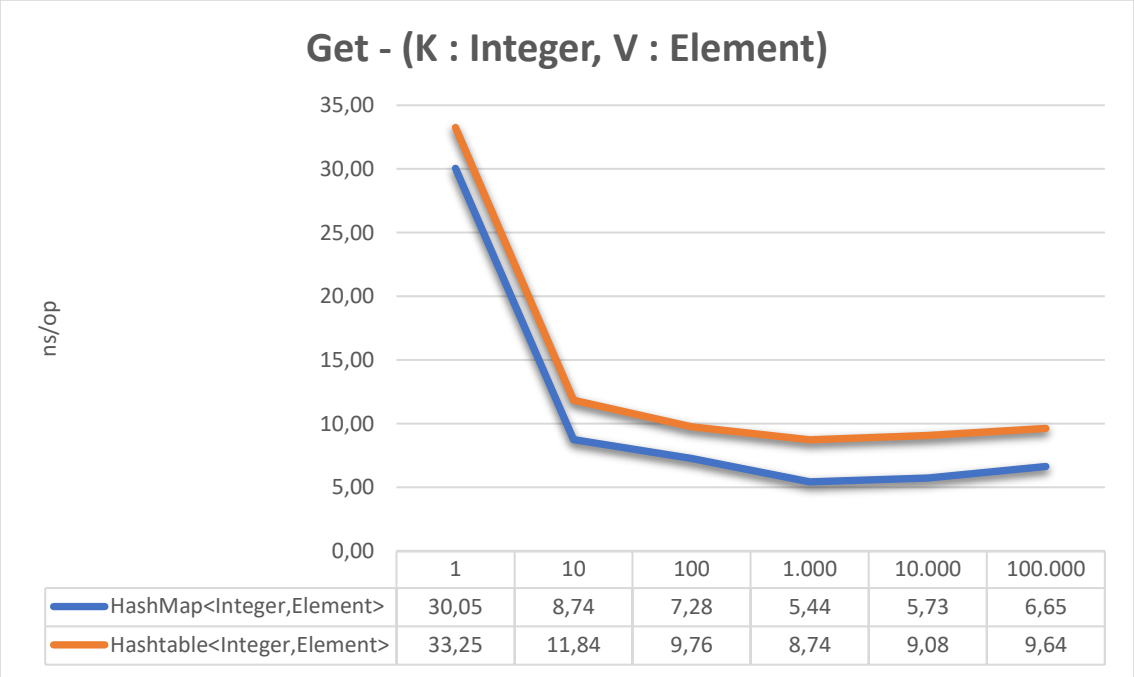
Στο σενάριο αυτό έγινε αναζήτηση ενός στοιχείου σε πλήρη δομή.

```
/**
 * Retrieve Operations.<br>
 * Get element from HashMap.
 *
 * @param blackhole eliminates dead code.
 */
@Benchmark
public void testGetIndexElementFromHashMap(Blackhole blackhole) {
  for (int i = 0; i < elements.length; i++) {
    blackhole.consume(fullHashMap.get(i));
  }
}
```

Συμπεράσματα : Και οι δύο δομές βασίζονται στη λογική της αντιστοίχισης κλειδιού - τιμής (key-value mapping). Υπενθυμίζεται ότι για το κάθε κλειδί ισοδυναμεί ένα hash που παράγεται από τη hashcode().

Η ταχύτητα των δομών αυτών βασίζεται στο ότι πια κάθε φορά που θέλει κάποιος να προσπελάσει ένα στοιχείο, απλώς πρέπει να αναζητήσει το κλειδί, για το οποίο αντιστοιχεί ένα hash και αυτό επιστρέφει η δομή για πρόσβαση στην τιμή που αντιστοιχεί στο κλειδί του.

Η όλη αυτή διαδικασία είναι ταχύτατη κάτι που επιβεβαιώνεται και από τα αποτελέσματα με σχεδόν σταθερό κόστος και για τις δύο δομές. Ακόμη και σε πολύ μεγάλα πλήθη με N = 100.000 παρατηρείται η εξαιρετική απόδοση των δομών σε απόκριση, με ένα ελάχιστο προβάδισμα στο HashMap για τους λόγους που έχουν αναφερθεί.



	1	10	100	1.000	10.000	100.000
HashMap<Integer,Element>	30,05	8,74	7,28	5,44	5,73	6,65
Hashtable<Integer,Element>	33,25	11,84	9,76	8,74	9,08	9,64

Η επιμερισμένη πολυπλοκότητα επιβεβαιώνεται με σταθερό κόστος της τάξης του O(1) για κάθε N , ακόμα και για N = 100.000 και στις δύο δομές.

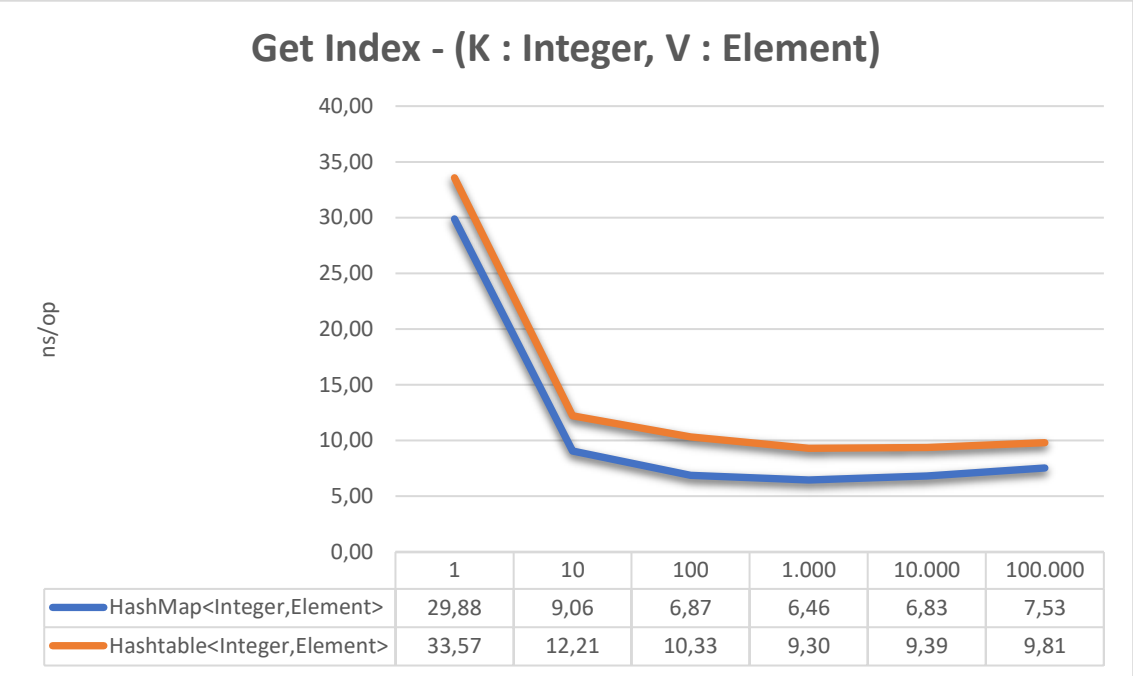
5.3.4 **Σενάριο εκτέλεσης : Αναζήτηση ταξινομημένου στοιχείου με απλό επαναληπτικό βρόχο (for loop).**

Στο σενάριο αυτό έγινε αναζήτηση του πλήθους των στοιχείων σε κενή δομή.

```
/**
 * Retrieve Operations.<br>
 * Get element from Hashtable.
 *
 * @param blackhole eliminates dead code.
 */
@Benchmark
public void testGetElementFromHashtable(Blackhole blackhole) {
  for (int i = 0; i < elements.length; i++) {
    blackhole.consume(fullHashtable.get(integers[i]));
  }
}
```

Συμπεράσματα : Και εδώ οι δύο δομές συμπεριφέρονται πανομοιότυπα με σχεδόν ελάχιστες διαφορές σε σχέση με το προηγούμενο σενάριο. Οι διαφορές είναι της τάξης των 1 - 2 ns ανά πράξη , ακόμη και σε πλήθη με N = 100.000.

.



	1	10	100	1.000	10.000	100.000
HashMap<Integer,Element>	29,88	9,06	6,87	6,46	6,83	7,53
Hashtable<Integer,Element>	33,57	12,21	10,33	9,30	9,39	9,81

Η επιμερισμένη πολυπλοκότητα επιβεβαιώνεται και εδώ με σταθερό κόστος της τάξης του O(1) για κάθε N , ακόμα και για N = 100.000 και στις δύο δομές.

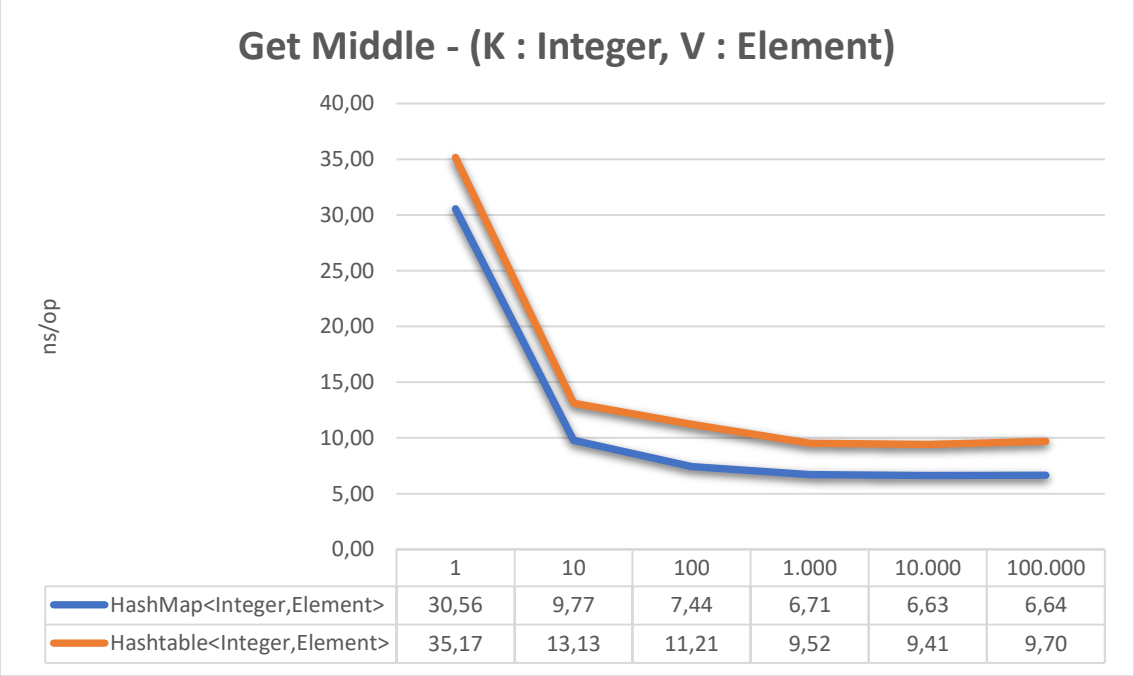
5.3.5 **Σενάριο εκτέλεσης : Εύρεση στοιχείου στο μέσο της δομής.**

Στο σενάριο αυτό έγινε αναζήτηση του μεσαίου στοιχείου σε πλήρη δομή.

```
/**
 * Retrieve Operations.<br>
 * Get middle element from HashMap.
 *
 * @param blackhole eliminates dead code.
 */
```

```
@Benchmark
public void testGetMiddleElementFromHashMap(Blackhole blackhole) {
  for (int i = 0; i < elements.length; i++) {
    blackhole.consume(fullHashMap.get(N / 2));
  }
}
```

Συμπεράσματα : Σχεδόν καμία διαφορά και εδώ για την αναζήτηση του μεσαίου στοιχείου, κάτι που είναι απόλυτα λογικό καθώς οι δύο δομές δεν «σαρώνουν» την δομή σειριακά , αλλά αναζητούν την αντιστοίχιση του hash απευθείας , προσομοιάζοντας το ArrayList με την τυχαία προσπέλαση.



	1	10	100	1.000	10.000	100.000
HashMap<Integer,Element>	30,56	9,77	7,44	6,71	6,63	6,64
Hashtable<Integer,Element>	35,17	13,13	11,21	9,52	9,41	9,70

Η επιμερισμένη πολυπλοκότητα σταθερού κόστους O(1) για τις δύο δομές επιβεβαιώνεται και σε αυτό το σενάριο, για όλα τα πλήθη.

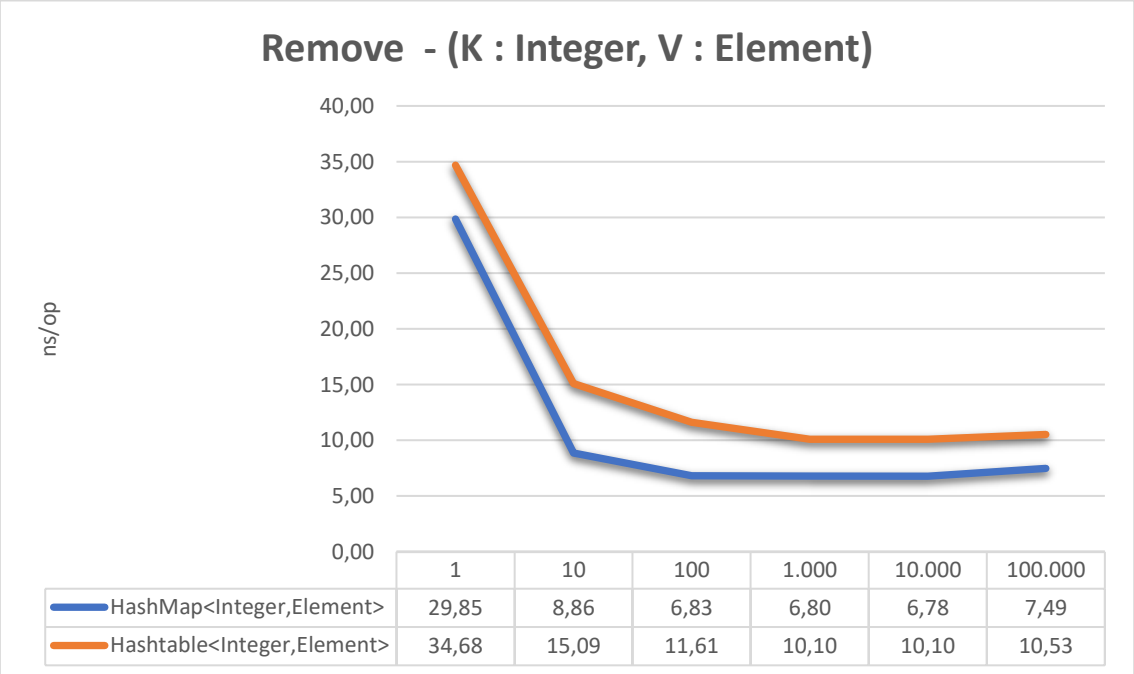
5.3.6 **Σενάριο εκτέλεσης : Διαγραφή στοιχείου από τη δομή.**

Στο σενάριο αυτό έγινε διαγραφή στοιχείου σε πλήρη δομή.

```
/**
 * Remove Operations.<br>
 * Remove element from HashMap.
 *
 * @param blackhole eliminates dead code.
 */
@Benchmark
public void testRemoveElementFromHashMap(Blackhole blackhole) {
  for (int i = 0; i < N; i++) {
    blackhole.consume(fullHashMap.remove(integers[i]));
  }
}
```

Συμπεράσματα : Και στη διαγραφή στοιχείου οι δομές συμπεριφέρονται με την ίδια απόδοση και σταθερότητα, ανεξαρτήτως πλήθους. Σταθερά ένα μικρό προβάδισμα στο HashMap της τάξεως των 7 ns για μικρά πλήθη έως N = 100, ενώ από εκεί και πάνω ακόμη και σε πλήθη N =100.000 η διαφορά μειώνεται στα 3 ns περίπου!

Η διαφορά δικαιολογείται από το ότι το HashMap είναι απαλλαγμένο από τα θέματα συγχρονισμού και διατήρησης της σειράς εισαγωγής των στοιχείων.



	1	10	100	1.000	10.000	100.000
HashMap<Integer,Element>	29,85	8,86	6,83	6,80	6,78	7,49
Hashtable<Integer,Element>	34,68	15,09	11,61	10,10	10,10	10,53

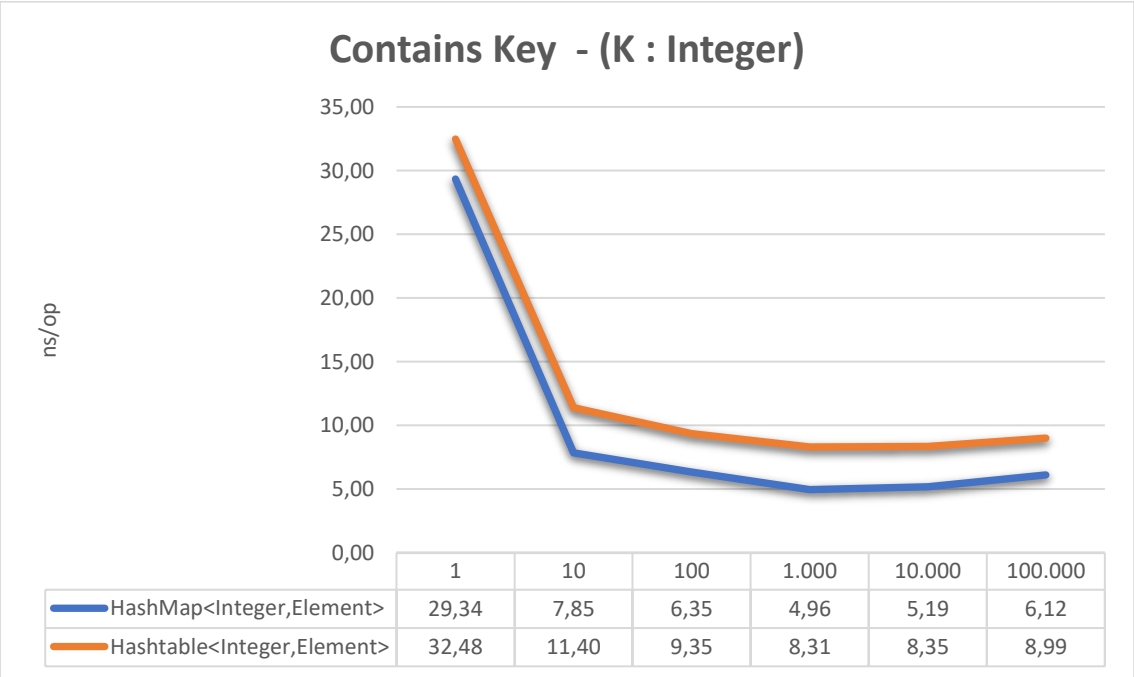
Η επιμερισμένη πολυπλοκότητα παραμένει σε σταθερό κόστος O(1) για τις δύο δομές.

5.3.7 **Σενάριο εκτέλεσης : Έλεγχος στοιχείου (Key) αν περιέχεται στη δομή.**

Στο σενάριο αυτό έγινε έλεγχος στοιχείου αν περιέχεται σε πλήρη δομή.

```
/**
 * Contains Operations.<br>
 * HashMap contains Key Integer.
 *
 * @param blackhole eliminates dead code.
 */
@Benchmark
public void testContainsKeyInHashMap(Blackhole blackhole) {
  for (int i = 0; i < N; i++) {
    blackhole.consume(fullHashMap.containsKey(integers[i]));
  }
}
```

Συμπεράσματα : Και οι δύο δομές ανταποκρίνονται με την ίδια σταθερότητα όπως και στα προηγούμενα σενάρια. Το HashMap παραμένει οριακά πιο αποδοτικό για τους λόγους που έχουν προαναφερθεί.



	1	10	100	1.000	10.000	100.000
HashMap<Integer,Element>	29,34	7,85	6,35	4,96	5,19	6,12
Hashtable<Integer,Element>	32,48	11,40	9,35	8,31	8,35	8,99

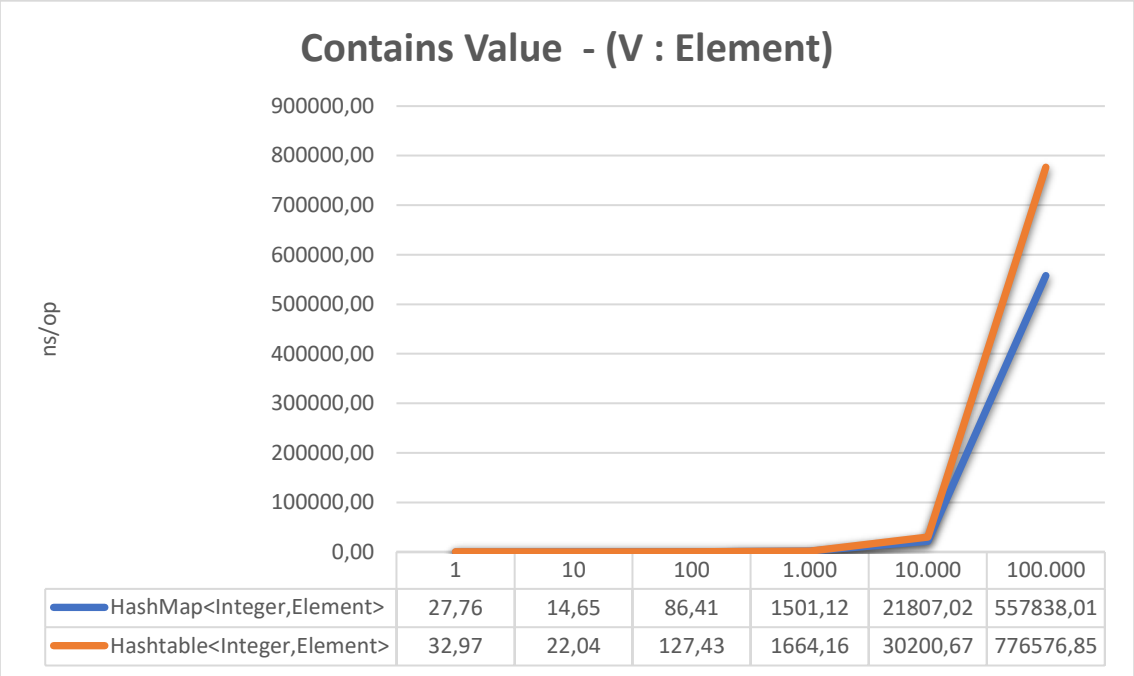
Η επιμερισμένη πολυπλοκότητα παραμένει σε σταθερό κόστος O(1) και για τις δύο δομές.

5.3.8 **Σενάριο εκτέλεσης : Έλεγχος στοιχείου (Value) αν περιέχεται στη δομή.**

Στο σενάριο αυτό έγινε έλεγχος στοιχείου αν περιέχεται σε πλήρη δομή.

```
/**
 * Contains Operations.<br>
 * HashMap contains Value Element.
 *
 * @param blackhole eliminates dead code.
 */
@Benchmark
public void testContainsValueInHashMap(Blackhole blackhole) {
  for (int i = 0; i < N; i++) {
    blackhole.consume(fullHashMap.containsValue(elements[i]));
  }
}
```

Συμπεράσματα : Σε αυτό το σενάριο, καταστρατηγείται το πλεονέκτημα και της δομής και η αναζήτηση γίνεται πια όχι βάσει κλειδιού αλλά βάσει τιμής. Εδώ λοιπόν η αποδοτικότητα που έχουν επιδείξει οι δομές δεν υφίσταται καθώς πια περνάει σε σειριακή αναζήτηση της και σύγκριση, παρακάμπτοντας έτσι τα σημαντικά πλεονεκτήματα τους : τα κλειδιά. Πρακτικά δεν υπάρχει κάποιος λόγος που να δικαιολογεί τέτοια χρήση, όμως είναι ένα σενάριο το οποίο αποδεικνύει ότι οι ίδιες δομές που έχουν εξαιρετική συμπεριφορά , με μια τέτοια κακή χρήση μπορούν να αποβούν και αυτές κοστοβόρες.



	1	10	100	1.000	10.000	100.000
HashMap<Integer,Element>	27,76	14,65	86,41	1501,12	21807,02	557838,01
Hashtable<Integer,Element>	32,97	22,04	127,43	1664,16	30200,67	776576,85

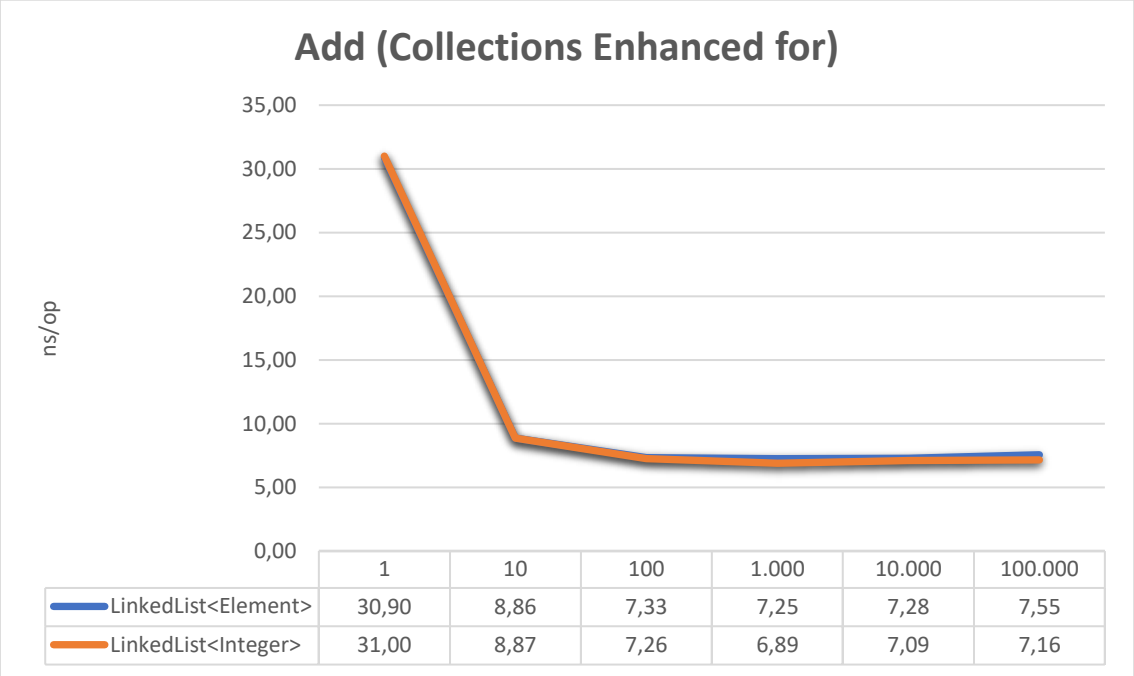
5.4 Interface Queue : Σενάρια εκτέλεσης & συμπεράσματα

5.4.1 Σενάριο εκτέλεσης : Εισαγωγή στοιχείου με ενισχυμένο επαναληπτικό βρόχο (Collections enhanced for loop).

Στο σενάριο αυτό έγινε εισαγωγή του πλήθους των στοιχείων σε κενή δομή.

```
/**
 * Create Operations.<br>
 * Add element to Queue.
 *
 * @param blackhole eliminates dead code.
 */
@Benchmark
public void testAddElementToQueue(Blackhole blackhole) {
  for (Element e : elements) {
    blackhole.consume(emptyLinkedlist.add(e));
  }
}
```

Συμπεράσματα : Για εισαγωγή ενός και μόνο στοιχείου σε συνδυασμό με το σταθερό κόστος του βρόχου, ανεξαρτήτως τύπου στοιχείου η δομή παρουσιάζει πανομοιότυπο κόστος με άλλες δομές (ArrayList). Εύλογα όσο το πλήθος αυξάνεται το κόστος αυτό επιμερίζεται και ήδη από πλήθος N = 10 και άνω η δομή επιδεικνύει όχι απλώς σταθερότητα στο κόστος εισαγωγής, αλλά και συνεχή βελτιστοποίηση με κάθε αύξηση του πλήθους. Αξιοσημείωτο είναι το ότι δεν υπάρχει σχεδόν καμία διαφορά , με d < 1 ns, ανάμεσα στους δύο τύπους στοιχείων.



	1	10	100	1.000	10.000	100.000
LinkedList<Element>	30,90	8,86	7,33	7,25	7,28	7,55
LinkedList<Integer>	31,00	8,87	7,26	6,89	7,09	7,16

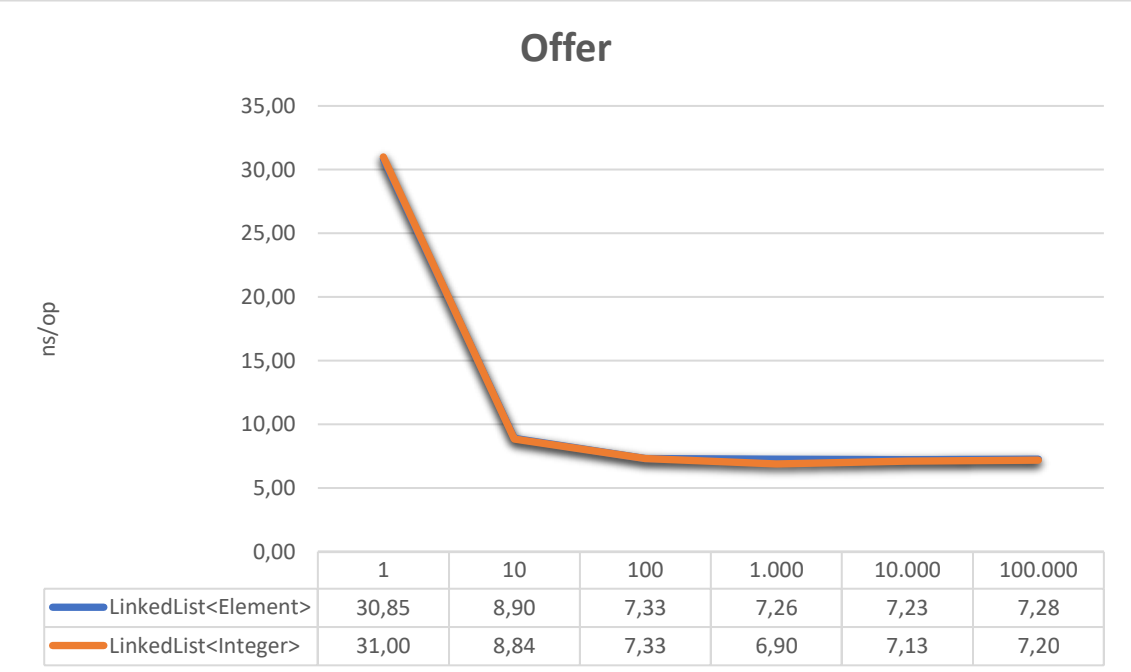
Η επιμερισμένη πολυπλοκότητα επιβεβαιώνεται σε σταθερό κόστος της τάξης του O(1) ανεξαρτήτως πλήθους και τύπου στοιχείου.

5.4.2 Σενάριο εκτέλεσης : Εισαγωγή στοιχείου χωρίς παραβίαση περιορισμών (offer) με ενισχυμένο επαναληπτικό βρόχο (Collections enhanced for loop).

Στο σενάριο αυτό έγινε εισαγωγή του πλήθους των στοιχείων σε κενή δομή.

```
/**
 * Create Operations.<br>
 * Offer element to ArrayList.
 *
 * @param blackhole eliminates dead code.
 */
@Benchmark
public void testOfferElementToQueue(Blackhole blackhole) {
  for (Element e : elements) {
    blackhole.consume(emptyLinkedlist.offer(e));
  }
}
```

Συμπεράσματα : Με τη μέθοδο offer διασφαλίζεται ότι θα γίνει η εισαγωγή του στοιχείου χωρίς να καταστρατηγηθεί ο περιορισμός του μεγέθους της ουράς (υλοποιημένης LinkedList εν προκειμένω) και χωρίς να έχουμε κάποιο exception. Πανομοιότυπα και εδώ το αρχικό κόστος επιμερίζεται όσο ανεβαίνει το πλήθος και η δομή συμπεριφέρεται με τον ίδιο τρόπο όπως και στο πρώτο σενάριο. Όπως και στο πρώτο σενάριο, δεν παρατηρείται κάποια ιδιαίτερη διαφορά στη συμπεριφορά της δομής που να σχετίζεται με τον τύπο του στοιχείου.



	1	10	100	1.000	10.000	100.000
LinkedList<Element>	30,85	8,90	7,33	7,26	7,23	7,28
LinkedList<Integer>	31,00	8,84	7,33	6,90	7,13	7,20

Η επιμερισμένη πολυπλοκότητα επιβεβαιώνεται και εδώ με σταθερό κόστος της τάξης του O(1)

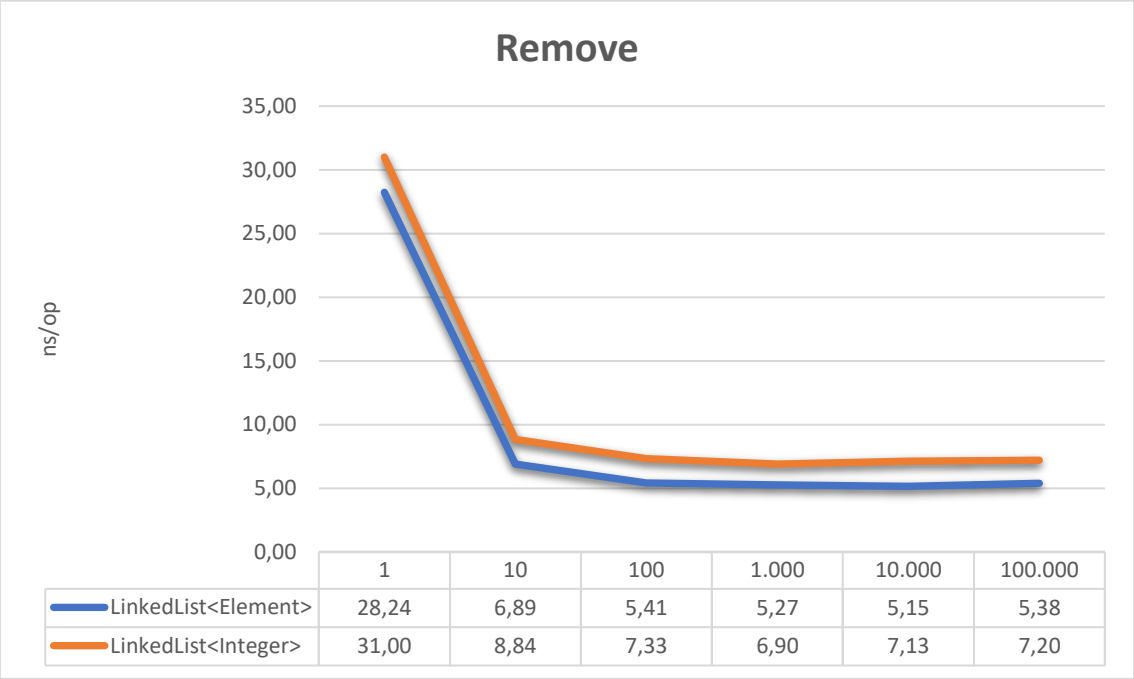
ανεξαρτήτως πλήθους και τύπου στοιχείου.

5.4.3 **Σενάριο εκτέλεσης : Διαγραφή στοιχείου από τη δομή.**

Στο σενάριο αυτό έγινε διαγραφή στοιχείου σε πλήρη δομή.

```
/**
 * Remove Operations.<br>
 * Remove element from Queue.
 *
 * @param blackhole eliminates dead code.
 */
@Benchmark
public void testRemoveElementFromQueue(Blackhole blackhole) {
  for (int i = 0; i < N; i++) {
    blackhole.consume(fullLinkedlist.remove());
  }
}
```

Συμπεράσματα : Με τη διαγραφή του στοιχείου επιστρέφεται το head της ουράς, αλλά χρειάζεται μια διαχείριση καθώς δεν γίνεται κάποιος έλεγχος αν όντως η ουρά έχει στοιχεία μέσα. Αν η ουρά είναι άδεια, τότε θα υπάρξει exception. Πανομοιότυπα και εδώ το αρχικό κόστος επιμερίζεται όσο ανεβαίνει το πλήθος και η δομή συμπεριφέρεται με τον ίδιο τρόπο ανεξάρτητα από τον τύπο του στοιχείου.



	1	10	100	1.000	10.000	100.000
LinkedList<Element>	28,24	6,89	5,41	5,27	5,15	5,38
LinkedList<Integer>	31,00	8,84	7,33	6,90	7,13	7,20

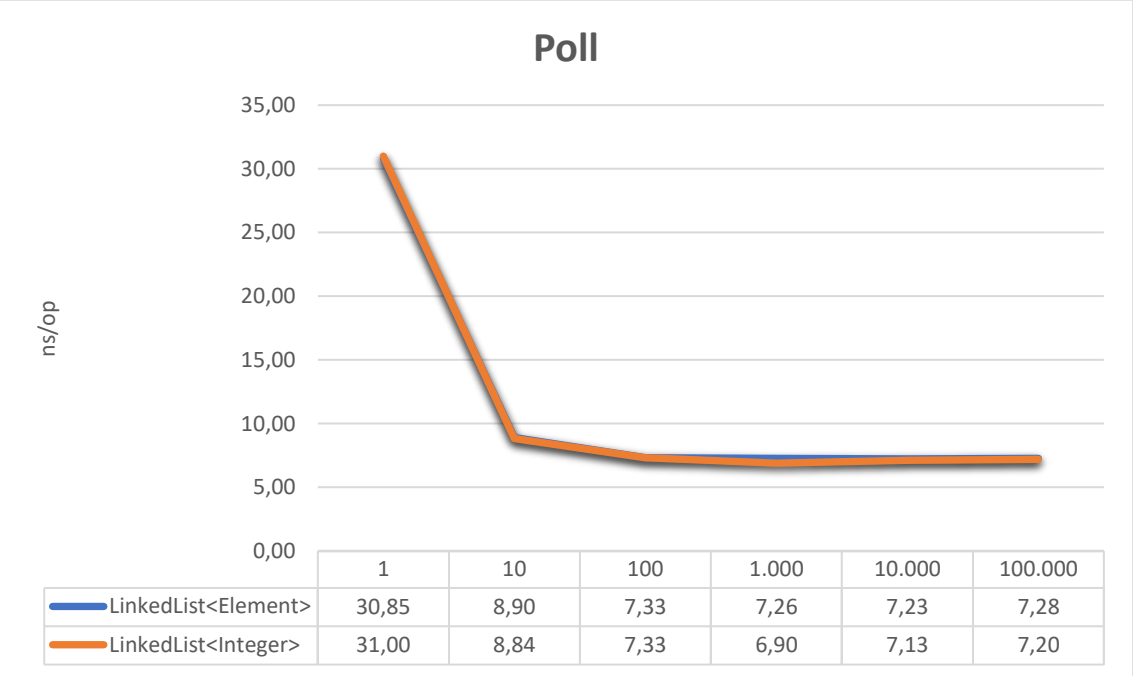
Η επιμερισμένη πολυπλοκότητα επιβεβαιώνεται και εδώ με σταθερό κόστος της τάξης του O(1) ανεξαρτήτως πλήθους και τύπου στοιχείου.

5.4.4 **Σενάριο εκτέλεσης : Διαγραφή στοιχείου από τη δομή (poll) .**

Στο σενάριο αυτό έγινε διαγραφή στοιχείου σε πλήρη δομή.

```
/**
 * Remove Operations.<br>
 * Poll element from Queue.
 *
 * @param blackhole eliminates dead code.
 */
@Benchmark
public void testPollElementFromQueue(Blackhole blackhole) {
  for (int i = 0; i < N; i++) {
    blackhole.consume(fullLinkedlist.poll());
  }
}
```

Συμπεράσματα : Με τη μέθοδο poll διασφαλίζεται ότι θα γίνει η διαγραφή του στοιχείου (επιστρέφει το head της ουράς) χωρίς να πάρει κάποιο exception , σε περίπτωση που η ουρά είναι άδεια. Αν η ουρά είναι άδεια , απλώς θα επιστρέψει false. Και εδώ το αρχικό κόστος επιμερίζεται όσο ανεβαίνει το πλήθος και η δομή συμπεριφέρεται με τον ίδιο τρόπο ανεξάρτητα από τον τύπο του στοιχείου και χωρίς να υπάρχει κάποια διαφοροποίηση στη συμπεριφορά.



	1	10	100	1.000	10.000	100.000
LinkedList<Element>	30,85	8,90	7,33	7,26	7,23	7,28
LinkedList<Integer>	31,00	8,84	7,33	6,90	7,13	7,20

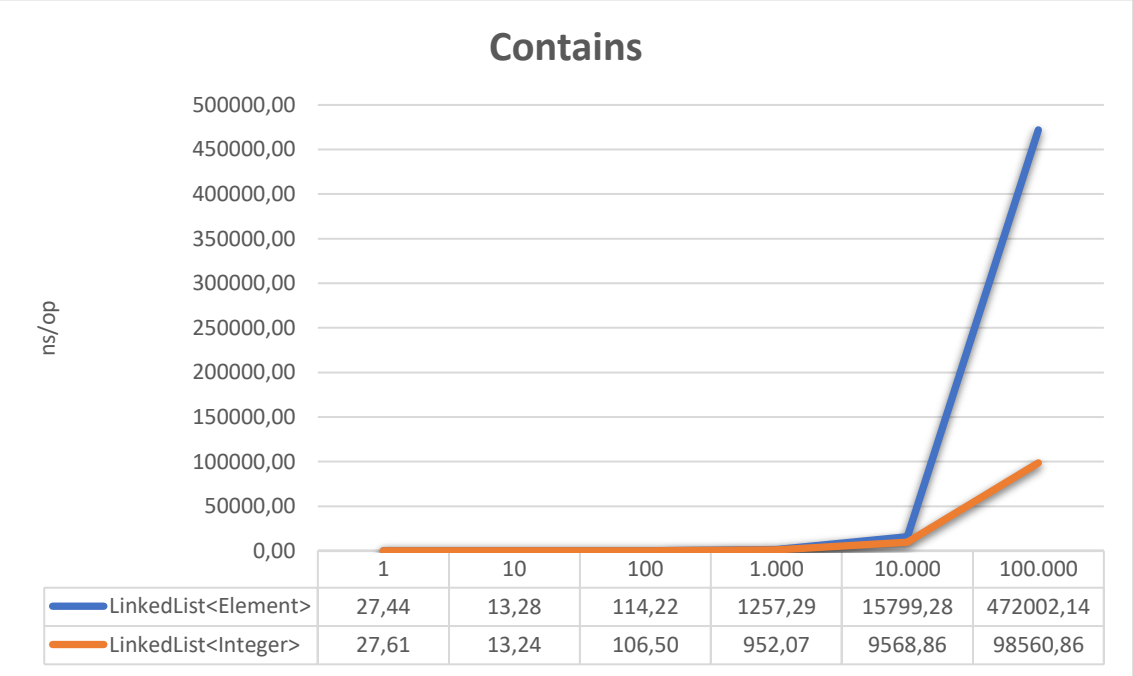
Η επιμερισμένη πολυπλοκότητα επιβεβαιώνεται και εδώ με σταθερό κόστος της τάξης του O(1) ανεξαρτήτως πλήθους και τύπου στοιχείου.

5.4.5 **Σενάριο εκτέλεσης : Έλεγχος στοιχείου αν περιέχεται στη δομή.**

Στο σενάριο αυτό έγινε έλεγχος στοιχείου αν περιέχεται σε πλήρη δομή.

```
/**
 * Contains Operations.<br>
 * Queue contains Element : True.
 *
 * @param blackhole eliminates dead code.
 */
@Benchmark
public void testContainsElementInQueue(Blackhole blackhole) {
  for (int i = 0; i < N; i++) {
    blackhole.consume(fullLinkedlist.contains(elements[i]));
  }
}
```

Συμπεράσματα : Και στις δύο η αναζήτηση θα γίνει σειριακά, στοιχείο με στοιχείο οπότε το κόστος θα αποβεί ιδιαίτερα υψηλό. Συγκεκριμένα ο ρυθμός αύξησης του κόστους είναι εκθετικός για τον τύπο στοιχείου Element , γραμμικός για τον τύπο στοιχείου Integer. Η διαφορά αφορά την υπολογιστική πολυπλοκότητα στις συγκρίσεις (λόγω της equals()).



	1	10	100	1.000	10.000	100.000
LinkedList<Element>	27,44	13,28	114,22	1257,29	15799,28	472002,14
LinkedList<Integer>	27,61	13,24	106,50	952,07	9568,86	98560,86

Η επιμερισμένη πολυπλοκότητα προφανώς και δεν επιβεβαιώνεται με σταθερό κόστος.

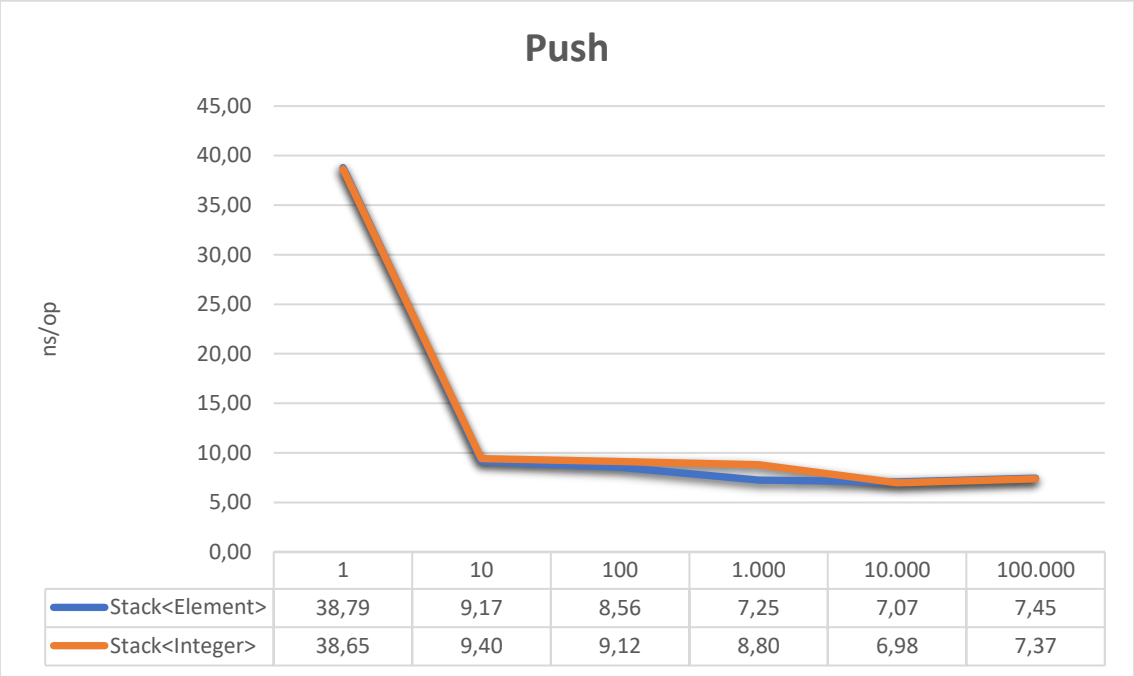
5.5 Stack : Σενάρια εκτέλεσης & συμπεράσματα

5.5.1 Σενάριο εκτέλεσης : Εισαγωγή στοιχείου με απλό επαναληπτικό βρόχο (for loop).

Στο σενάριο αυτό έγινε εισαγωγή του πλήθους των στοιχείων σε κενή δομή.

```
/**
 * Create Operations.<br>
 * Add element to Stack.
 *
 * @param blackhole eliminates dead code.
 */
@Benchmark
public void testAddToStack(Blackhole blackhole) {
  for (int i = 0; i < N; i++) {
    blackhole.consume(emptyStack.push(elements[i]));
  }
}
```

Συμπεράσματα : Για δομή ενός στοιχείου παρατηρείται ότι το κόστος εισαγωγής , σε συνδυασμό με το σταθερό κόστος του βρόχου (δημιουργία, έναρξη, τερματισμός) δίνουν μια τιμή περίπου στα 38 ns , ανεξάρτητα από τον τύπο του στοιχείου. Όπως πια είναι γνωστό , με την αύξηση του πλήθους το αρχικό αυτό σταθερό κόστος επιμερίζεται και βελτιστοποιείται. Υπενθυμίζεται ότι το Stack είναι μια υλοποίηση του Vector , που με τη σειρά του είναι μια υλοποίηση βασισμένη σε πίνακα (Array) , άρα και εδώ υπάρχει η τυχαία προσπέλαση στη δομή, οπότε η εισαγωγή στην κορυφή της στοίβας γίνεται πάρα πολύ γρήγορα.



	1	10	100	1.000	10.000	100.000
Stack<Element>	38,79	9,17	8,56	7,25	7,07	7,45
Stack<Integer>	38,65	9,40	9,12	8,80	6,98	7,37

Η επιμερισμένη πολυπλοκότητα για σταθερό κόστος της τάξεως του O(1) φαίνεται να επιβεβαιώνεται απόλυτα ανεξάρτητα από τον τύπο του στοιχείου , ακόμη και για πολύ μεγάλα N (= 100.000) καθώς το κόστος είναι σχεδόν σταθερό.

5.5.2 **Σενάριο εκτέλεσης : Εισαγωγή στοιχείου με ενισχυμένο επαναληπτικό βρόχο (Collections enhanced for loop).**

Στο σενάριο αυτό έγινε εισαγωγή του πλήθους των στοιχείων σε κενή δομή.

```
/**
 * Create Operations.<br>
 * Add foreach element to Stack.
 *
 * @param blackhole eliminates dead code.
 */
@Benchmark
public void testAddElementForEachToStack(Blackhole blackhole) {
  for (Element e : elements) {
    blackhole.consume(emptyStack.push(e));
  }
}
```

Συμπεράσματα : Καμία διαφοροποίηση και στην περίπτωση του ενισχυμένου επαναληπτικού βρόχου. Η αποδοτικότητα της δομής παραμένει ίδια , ανεξάρτητα από τον τύπο του στοιχείου.

	1	10	100	1.000	10.000	100.000
Stack<Element>	38,65	8,75	8,78	7,62	6,59	6,86
Stack<Integer>	38,21	8,71	8,21	7,91	6,48	6,93

Η επιμερισμένη πολυπλοκότητα επιβεβαιώνεται σε σταθερό κόστος της τάξης του O(1) ανεξαρτήτως πλήθους και τύπου στοιχείου.

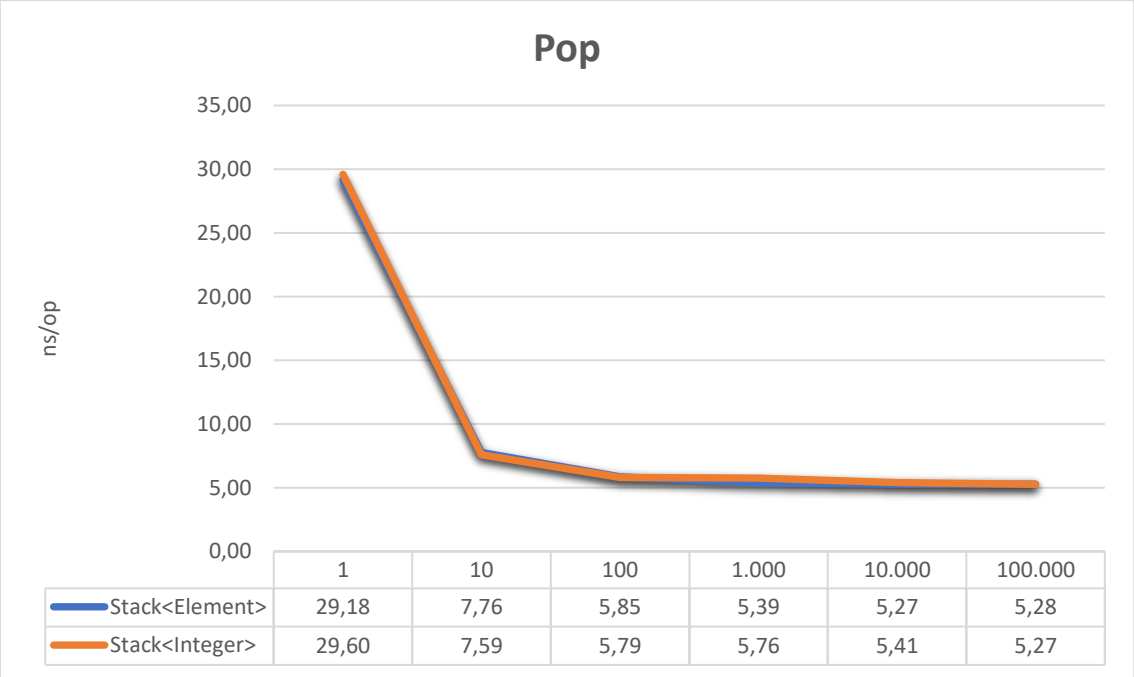
5.5.3 **Σενάριο εκτέλεσης : Διαγραφή στοιχείου από τη δομή.**

Στο σενάριο αυτό έγινε διαγραφή στοιχείου σε πλήρη δομή.

```
/**
 * Remove Operations.<br>
 * Remove element from Stack.
 *
 * @param blackhole eliminates dead code.
 */
```

```
@Benchmark
public void testRemoveElementFromStack(Blackhole blackhole) {
  for (int i = 0; i < N; i++) {
    blackhole.consume(fullStack.pop());
  }
}
```

Συμπεράσματα : Με τη διαγραφή του στοιχείου αφαιρείται το κορυφαίο στοιχείο της στοίβας από τη δομή. Και εδώ η απόδοση της δομής είναι κορυφαία , καθώς απαιτούνται μόλις κάτι λιγότερο από 6 ns για να ολοκληρωθεί η πράξη σε πλήθη από N >= 100.



	1	10	100	1.000	10.000	100.000
Stack<Element>	29,18	7,76	5,85	5,39	5,27	5,28
Stack<Integer>	29,60	7,59	5,79	5,76	5,41	5,27

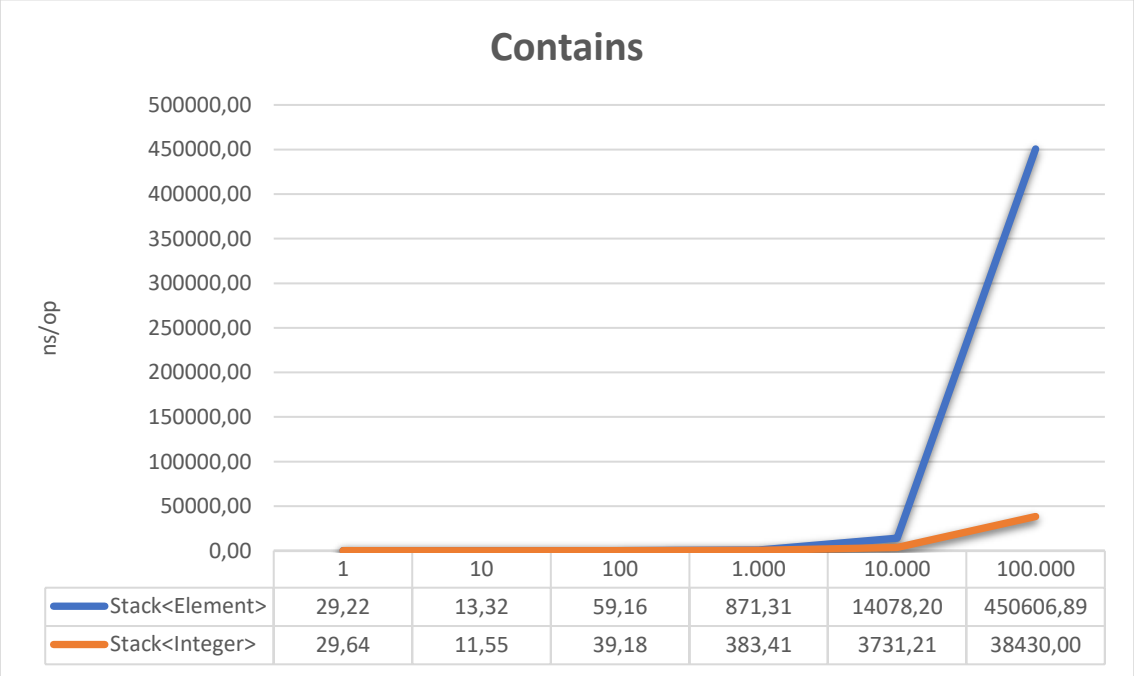
Η επιμερισμένη πολυπλοκότητα επιβεβαιώνεται και εδώ με σταθερό κόστος της τάξης του O(1) ανεξαρτήτως πλήθους και τύπου στοιχείου.

5.5.4 **Σενάριο εκτέλεσης : Έλεγχος στοιχείου αν περιέχεται στη δομή.**

Στο σενάριο αυτό έγινε έλεγχος στοιχείου αν περιέχεται σε πλήρη δομή.

```
/**
 * Contains Operations.<br>
 * Stack contains Element : True.
 *
 * @param blackhole eliminates dead code.
 */
@Benchmark
public void testContainsInStack(Blackhole blackhole) {
  for (int i = 0; i < N; i++) {
    blackhole.consume(fullStack.contains(elements[i]));
  }
}
```

Συμπεράσματα : Η στοίβα θα πρέπει να σαρωθεί σειριακά για την εύρεση του συγκεκριμένου στοιχείου, οπότε κάθε αναζήτηση & σύγκριση είναι εφάμιλλη δομών όπως το ArrayList.



	1	10	100	1.000	10.000	100.000
Stack<Element>	29,22	13,32	59,16	871,31	14078,20	450606,89
Stack<Integer>	29,64	11,55	39,18	383,41	3731,21	38430,00

Προφανώς δεν επιβεβαιώνεται επιμερισμένη πολυπλοκότητα με σταθερό κόστος τάξης O(1).

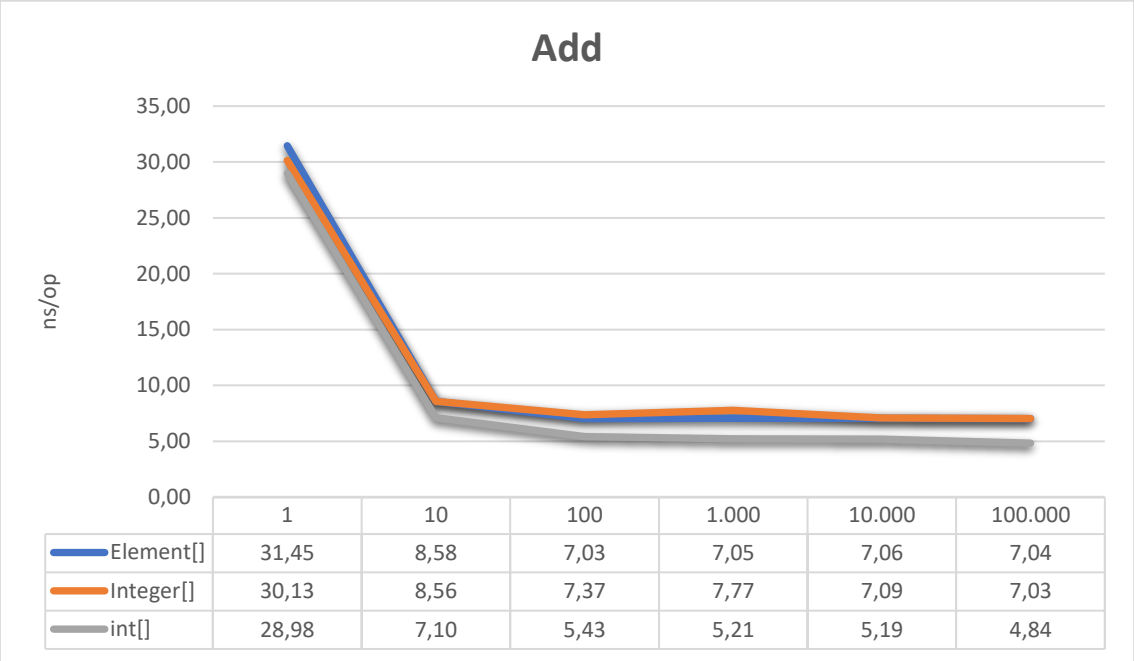
5.6 Array : Σενάρια εκτέλεσης & συμπεράσματα

5.6.1 Σενάριο εκτέλεσης : Εισαγωγή στοιχείου με απλό επαναληπτικό βρόχο (for loop).

Στο σενάριο αυτό έγινε εισαγωγή του πλήθους των στοιχείων σε κενή δομή.

```
/**
 * Create Operations.<br>
 * Add element to Array.
 *
 * @param blackhole eliminates dead code.
 */
@Benchmark
public void addElementToArray(Blackhole blackhole) {
  for (int i = 0; i < N; i++) {
    emptyArray[i] = elements[i];
    blackhole.consume(emptyArray[i]);
    blackhole.consume(elements[i]);
  }
}
```

Συμπεράσματα : Και στους τρείς τύπους στοιχείων (Element, Integer, int) η δομή συμπεριφέρεται με τον τρόπο που περιμένει κανείς, λόγω της τυχαίας προσπέλασης : με πολύ μεγάλη ταχύτητα. Η διαφορά του primitive τύπου στοιχείου είναι εμφανής ακόμα και σε τόσο χαμηλά επίπεδα κόστους.



	1	10	100	1.000	10.000	100.000
Element[]	31,45	8,58	7,03	7,05	7,06	7,04
Integer[]	30,13	8,56	7,37	7,77	7,09	7,03
int[]	28,98	7,10	5,43	5,21	5,19	4,84

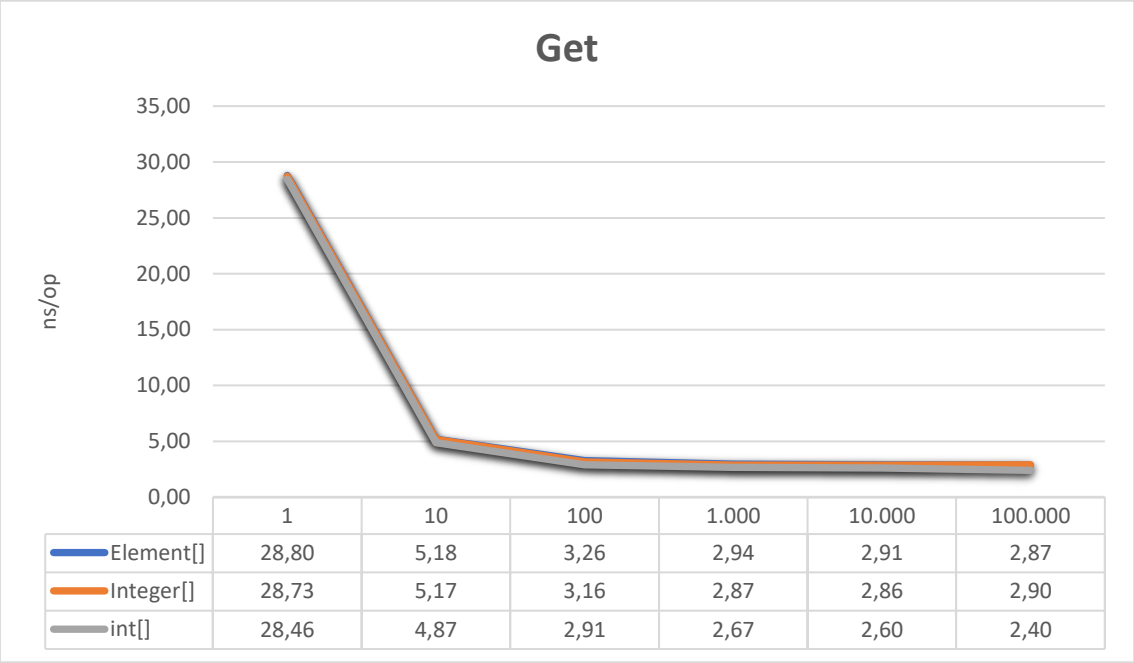
Η επιμερισμένη πολυπλοκότητα για σταθερό κόστος της τάξεως του O(1) επιβεβαιώνεται απόλυτα ανεξάρτητα από τον τύπο και το πλήθος των στοιχείων.

5.6.2 **Σενάριο εκτέλεσης : Εύρεση στοιχείου στο πλήθος της δομής.**

Στο σενάριο αυτό έγινε αναζήτηση ενός στοιχείου σε πλήρη δομή.

```
/**
 * Retrieve Operations.<br>
 * Get element from Array.
 *
 * @param blackhole eliminates dead code.
 */
@Benchmark
public void testGetElementFromArray(Blackhole blackhole) {
  for (int i = 0; i < elements.length; i++) {
    blackhole.consume(fullArray[i]);
  }
  blackhole.consume(fullArray);
}
```

Συμπεράσματα : Και στους τρείς τύπους το Array είναι πραγματικά σχεδόν ακαριαίο στην απόκριση, λόγω τυχαίας προσπέλασης. Με απειροελάχιστη διαφορά υπερτερεί το Array από primitives.



	1	10	100	1.000	10.000	100.000
Element[]	28,80	5,18	3,26	2,94	2,91	2,87
Integer[]	28,73	5,17	3,16	2,87	2,86	2,90
int[]	28,46	4,87	2,91	2,67	2,60	2,40

Η επιμερισμένη πολυπλοκότητα για σταθερό κόστος της τάξεως του O(1) επιβεβαιώνεται απόλυτα και σε αυτή την περίπτωση, ανεξαρτήτως τύπου στοιχείου και πλήθους.

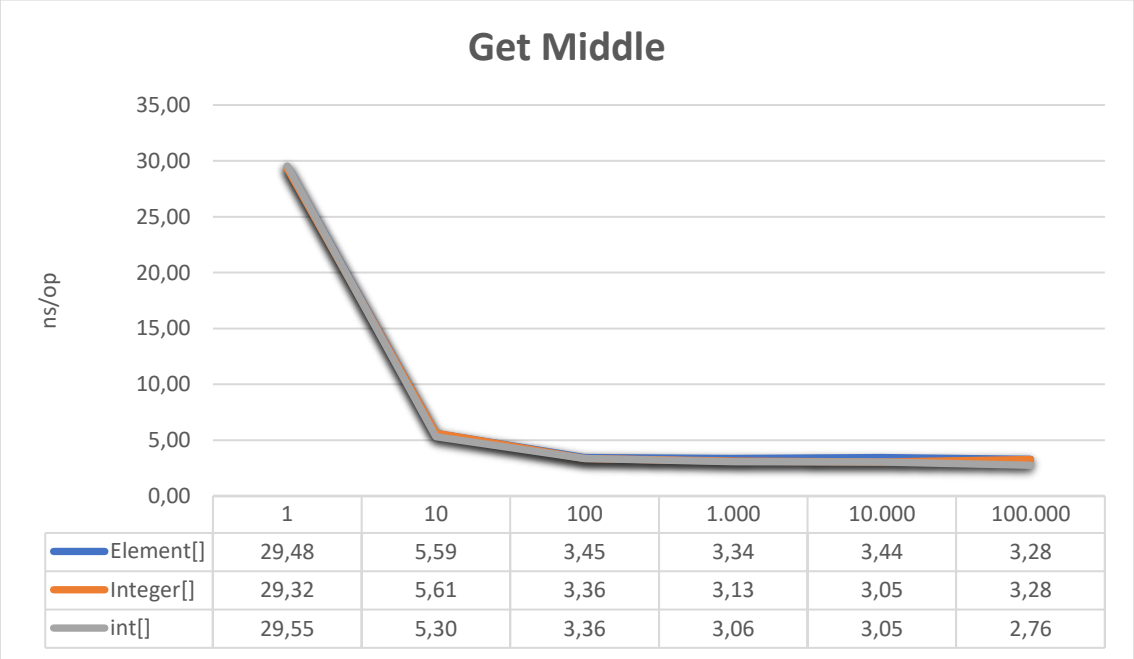
5.6.3 **Σενάριο εκτέλεσης : Εύρεση στοιχείου στο μέσο της δομής.**

Στο σενάριο αυτό έγινε αναζήτηση του μεσαίου στοιχείου σε πλήρη δομή.

```
/**
 * Retrieve Operations.<br>
 * Get middle element from Array.
 *
 * @param blackhole eliminates dead code.
 */
```

```
@Benchmark
public void testGetMiddleElementFromArray(Blackhole blackhole) {
  for (int i = 0; i < elements.length; i++) {
    blackhole.consume(fullArray[N / 2]);
  }
  blackhole.consume(fullArray);
}
```

Συμπεράσματα : Ο μόνος λόγος διεξαγωγής του συγκεκριμένου σεναρίου είναι για συγκριτικούς λόγους με άλλες δομές. Το Array με την τυχαία προσπέλαση δεν διαφοροποιείται καθόλου στο κόστος της πράξης , ανεξαρτήτως θέσης στοιχείου. (Απειροελάχιστη διαφορά κόστους με τύπο primitives). Μια πράξη που για παράδειγμα η LinkedList «πληρώνει» με πολύ μεγάλο κόστος.



	1	10	100	1.000	10.000	100.000
Element[]	29,48	5,59	3,45	3,34	3,44	3,28
Integer[]	29,32	5,61	3,36	3,13	3,05	3,28
int[]	29,55	5,30	3,36	3,06	3,05	2,76

Και εδώ επιβεβαιώνεται η επιμερισμένη πολυπλοκότητα σταθερού κόστους O(1) για κάθε τύπο στοιχείου και για κάθε πλήθος.

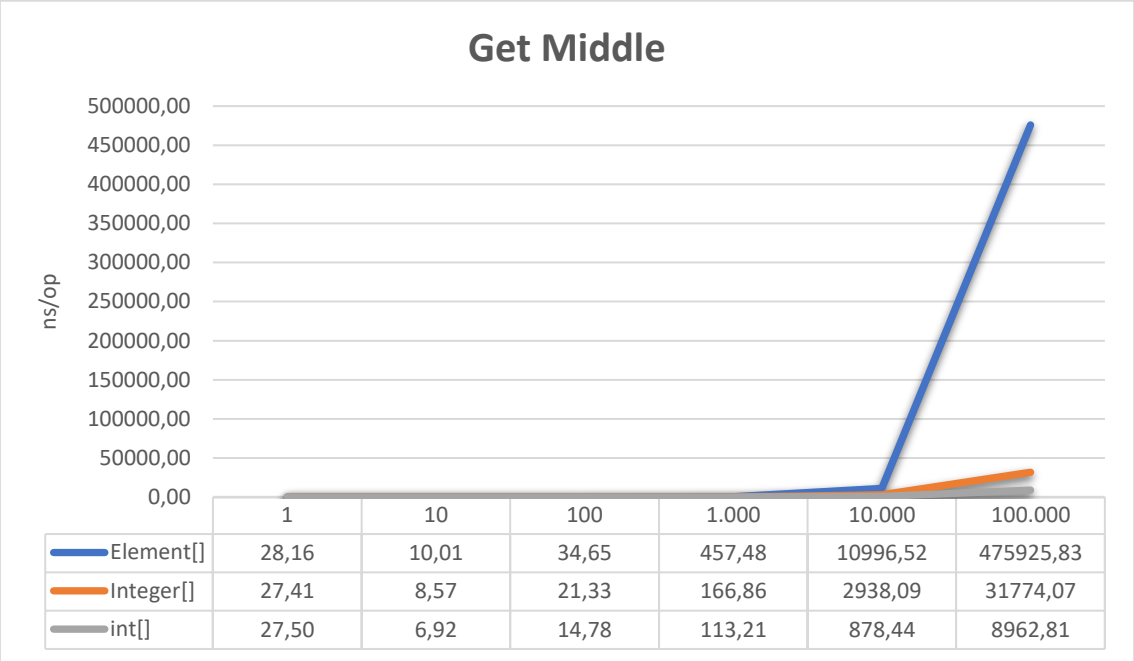
5.6.4 **Σενάριο εκτέλεσης : Έλεγχος στοιχείου αν περιέχεται στη δομή.**

Στο σενάριο αυτό έγινε έλεγχος στοιχείου αν περιέχεται σε πλήρη δομή.

```
/**
 * Contains Operations.<br>
 * Array with For contains Element : True.
 *
 * @param blackhole eliminates dead code.
 */
@Benchmark
public void testContainsElementInArrayFor(Blackhole blackhole) {
  for (int i = 0; i < N; i++) {
    blackhole.consume(contains(elements, fullArray[i]));
  }
}
```

Συμπεράσματα : Εδώ το Array θα «πληρώσει» με το κόστος της σειριακής προσπέλασης & σύγκρισης των στοιχείων. Ανάλογα τον τύπο του στοιχείου , παρατηρείται και αλλαγή στο κόστος. Για παράδειγμα το κόστος σε primitive τύπο είναι καθαρά γραμμικό, καθώς απλά γίνεται μια απλή σύγκριση τύπου == ή οποία συγκρίνει απλώς την τιμή που είναι αποθηκευμένη στη θέση. ΣΕ επίπεδο αντικειμένου , για παράδειγμα σε τύπο Integer το κόστος είναι κατά πολύ αυξημένο καθώς γίνεται κλήση της equals() σε κάθε σύγκριση.

Η πολύ μεγάλη διαφορά με τον τύπο στοιχείου Element αποδίδεται στο override της equals() η οποία συγκρίνοντας κάθε πεδίο του instance, αυξάνει κατά πολύ την πολυπλοκότητα της κάθε πράξης. Σε επίπεδο κόστους παρατηρείται αύξηση εκθετικού ρυθμού για τον τύπο Element , ενώ η αύξηση στον τύπο Integer είναι σαφέστατα μικρότερου ρυθμού.



	1	10	100	1.000	10.000	100.000
Element[]	28,16	10,01	34,65	457,48	10996,52	475925,83
Integer[]	27,41	8,57	21,33	166,86	2938,09	31774,07
int[]	27,50	6,92	14,78	113,21	878,44	8962,81

Προφανώς δεν επιβεβαιώνεται επιμερισμένη πολυπλοκότητα με σταθερό κόστος τάξης O(1).

6. Βιβλιογραφία

1. **Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein**

Introduction to Algorithms, Third Edition, 2009

2. **Java Collections Framework - Programmering**

https://www.programering.com/a/MTNwEjMwATU.html

3. **Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein**

Introduction to Algorithms, Third Edition, 2009

4. **Donald E. Knuth**

The art of computer programming, Volume 1, Fundamental Algorithms, 3rd Edition, 1997

5. **Κωνσταντόπουλος Χ. , Παναγιωτόπουλος Α.**

Αλγόριθμοι, Σημειώσεις διδασκαλίας

6. **Asymptotic Notations – Data structures and algorithms**

https://www.studytonight.com/data-structures/aysmptotic-notations

7. **Γεωργακόπουλος Γεώργιος Φρ.**

Δομές Δεδομένων - Έννοιες , Τεχνικές και Αλγόριθμοι, 2η Έκδοση 2008

8. **Herbert Schildt**

Java – The complete reference, 11th Edition

9. **Τσιχριντζής Χ. Γεώργιος**

Δομές Δεδομένων, Σημειώσεις Διδασκαλίας

10. **Java JDK11 Docs - Class Vector<E>**

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Vector.html

11. **Java JDK11 Docs - Class Stack<E>**

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Stack.html

12. **Java JDK11 Docs - Class ArrayList<E>**

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/ArrayList.html

13. **Java JDK11 Docs - Class LinkedList<E>**

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/LinkedList.html

14. **Java JDK11 Docs - Class ArrayDeque<E>**

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/ArrayDeque.html

15. **Java JDK11 Docs - Class HashSet<E>**

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/HashSet.html

16. **Java JDK11 Docs - Class LinkedHashSet<E>**

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/LinkedHashSet.html

17. **Java JDK11 Docs - Class TreeSet<E>**

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/TreeSet.html

18. **Java JDK11 Docs - Class Comparable<E>**

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Comparable.html

19. **Java JDK11 Docs - Class Map<E>**

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Map.html

20. **Java JDK11 Docs - Class Hashtable<K,V>**

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Hashtable.html

21. **Java JDK11 Docs - Class HashMap <E>**

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/HashMap.html

22. **Java JDK11 Docs - Class Array**

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/reflect/Array.html

23. **Java World – What is the JVM ? Introducing the Java Virtual Machine**

https://www.javaworld.com/article/3272244/core-java/what-is-the-jvm-introducing-the-java-virtual-machine.html

24. **OpenJDK JMH – Java Microbenchmark Harness Framework**

https://openjdk.java.net/projects/code-tools/jmh/

25. **Oracle – Sun Acquisition**

https://www.oracle.com/sun/

26. **The Java Tutorials - Annotations**

https://docs.oracle.com/javase/tutorial/java/annotations/