ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Ανάπτυξη Εργαλείων για την Ανάλυση της Συμπεριφοράς και τη Βελτιστοποίηση των Δυναμικών Τύπων Δεδομένων σε Εφαρμογές Ενσωματωμένων Υπολογιστών

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Παναγιώτης, Ι. Θεοχάρης

Επιβλέπων : Δημήτριος Σούντρης
Επίκουρος Καθηγητής Ε.Μ.Π.

Αθήνα, Απρίλιος 2009

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Ανάπτυξη Εργαλείων για την Ανάλυση της Συμπεριφοράς και τη Βελτιστοποίηση των Δυναμικών Τύπων Δεδομένων σε Εφαρμογές Ενσωματωμένων Υπολογιστών

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Παναγιώτης, Ι. Θεοχάρης

Επιβλέπων : Δημήτριος Σούντρης
Επίκουρος Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 8η Απριλίου 2009.

.....	.....	.....
Κιαμάλ Πεκμεστζή	Δημήτριος Σούντρης	Γιώργος Οικονομάκος
Καθηγητής Ε.Μ.Π.	Επίκουρος Καθηγητής Ε.Μ.Π.	Λέκτορας Ε.Μ.Π.

Αθήνα, Απρίλιος 2009

.....

Παναγιώτης Ι. Θεοχάρης

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Στην Οικογένειά μου

ΠΕΡΙΛΗΨΗ

Τα τελευταία χρόνια, υπάρχει ένα ολοένα και αυξανόμενο πλήθος εφαρμογών (π.χ. Τρισδιάστατα παιχνίδια, αναπαραγωγή βίντεο) προερχόμενο από το πεδίο γενικής χρήσης των προσωπικών υπολογιστών, οι οποίες έχουν τεράστιες απαιτήσεις σε μνήμη, το οποίο απαιτείται να μεταφερθεί σε εξαιρετικά μικρές συσκευές.

Ωστόσο, τα ενσωματωμένα συστήματα δυσκολεύονται να ανταποκριθούν στην εκτέλεση αυτών των σύνθετων εφαρμογών, επειδή προέρχονται από ένα πεδίο με εντελώς διαφορετικούς περιορισμούς όσον αφόρα στη χρήση της μνήμης όπου συγκεκριμένα δεν υπάρχει έντονο ενδιαφέρον για αποδοτική χρήση της δυναμικής μνήμης. Πράγματι, στις μέρες μας, ένας προσωπικός υπολογιστής είναι εξοπλισμένος με τουλάχιστον 1024 ή 2048 MB RAM, σε αντίθεση με τα 32 ή 64 MB που είναι διαθέσιμα στα ενσωματωμένα συστήματα. Συνεπώς, ένα από τα κύρια ζητήματα που πρέπει να αντιμετωπιστούν κατά τη μεταφορά των εφαρμογών σε ενσωματωμένες, είναι η βελτιστοποίηση της χρήσης της δυναμικής μνήμης.

Οι προγραμματιστές αφιερώνουν το κύριο μέρος του χρόνου τους στο πλαίσιο της δουλειάς τους, από το να βελτιστοποιούν των κώδικά τους. Αυτά συστήματα εργαλείων που προσφέρουν ανάλυση του κώδικα υπάρχουν ήδη, αλλά περιορίζονται στη βελτιστοποίηση στατικών δομών μνήμης.

Σε αυτή τη Διπλωματική Εργασία, προτείνουμε ένα σύνολο εργαλείων που παρέχει συνολική ανάλυση της συμπεριφοράς των δυναμικών δομών δεδομένων των εφαρμογών, όπως τους δυναμικούς πίνακες, τις λίστες, τα δέντρα. Το αποτέλεσμα της ανάλυσης εξάγεται σε διάφορες μορφές όπως η σχεσιακή βάση δεδομένων και η XML, ούτως ώστε να μπορεί κάποιος να υποβάλει ερωτήματα στη βάση για να εξάγει πληροφορία ή να χρησιμοποιήσει κάποιον υπάρχοντα XML parser για εμφανίσει τα αποτελέσματα.

Το σύνολο των εργαλείων μας δοκιμάστηκε αναλύοντας τη συμπεριφορά των δυναμικών δομών δεδομένων ενός πολύ σύνθετου τρισδιάστατου παιχνιδιού (Vdrift), και τελικά επιλέγοντας τις βέλτιστες υλοποιήσεις για κάθε ένα από αυτούς. Παρουσιάσαμε αποτελέσματα που δείχνουν 32% μείωση στην απαιτούμενη διαθέσιμη μνήμη και 6% μείωση στις προσπελάσεις των δεδομένων.

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ

Δομές Δεδομένων, Ανάλυση Συμπεριφοράς, Profiling, STL, Πίνακας, Λίστα, Δέντρο, Ενσωματωμένες Εφαρμογές

ABSTRACT

In the last years, there has been an increasing amount of applications (e.g. 3D games, video-players) coming from the general-purpose domain, having large run-time memory management requirements, need to be mapped onto an extremely compact device.

However, embedded systems struggle to execute these complex applications because they come from desktop systems, holding very different restrictions regarding memory usage features, and more concretely not concerned with an efficient use of the dynamic memory. In fact, a desktop computer typically includes today between 1024 and 2048 MB of RAM memory at least, as opposed to the 32 or 64 MB present in modern embedded systems. Therefore, one of the main tasks of the porting process of multimedia applications onto embedded multimedia systems is the optimization of the dynamic memory subsystem.

Developers spend more time focusing on the context of their work, rather than optimizing their code. Automatic frameworks that offer code analysis do exist, but are limited in the following way: none of the existing frameworks targets the optimization of dynamic containers found in almost every modern application.

In this Thesis, we propose a framework that provides a complete behavioral analysis of the applications' dynamic data structures like vectors, lists, trees. The analysis output comes in several formats like relational database and XML, so that one can use queries to extract information or use an existing XML parser to visualize the results.

Our framework was tested by analyzing the behavior of dynamic data structures of a complex 3D game application (VDrift) and subsequently choosing optimal implementations for each one of them. The results showed a 32% reduction in requested memory footprint and 6% reduction in data accesses.

KEY WORDS

Data Structures, Behavioural Analysis, Profiling, STL, Vector, List, Tree,
Embedded Applications

ΕΥΧΑΡΙΣΤΙΕΣ

Η συγκεκριμένη διπλωματική εργασία εκπονήθηκε στο Εργαστήριο Μικροϋπολογιστών & Ψηφιακών Συστημάτων της Σχολής Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Εθνικού Μετσόβιου Πολυτεχνείου υπό την επίβλεψη του Επίκουρου Καθηγητή, κύριου Δημήτριου Σούντρη.

Θα ήθελα πρώτα από όλα να ευχαριστήσω θερμά τον κύριο Σούντρη που μου έδωσε την ευκαιρία να ασχοληθώ με αυτό το εξαιρετικά ενδιαφέρον και σύγχρονο θέμα καθώς και για την αμέριστη στήριξη και βοήθεια που μου παρείχε κατά την πορεία της εκπόνησης της εργασίας αυτής.

Θέλω ιδιαίτερα να ευχαριστήσω τον υποψήφιο διδάκτορα Χρήστο Μπαλούκα, που παρόλα τα χιλιόμετρα που μας χώριζαν, η επικοινωνία μαζί του ήταν άμεση, και χάρη στη συνεχή παροχή εκ μέρους του, τεχνικής βοήθειας και ιδεών για να ξεπεραστούν οι σκόπελοι που προέκυπταν, η παρούσα εργασία ολοκληρώθηκε επιτυχώς με τον πλέον ευχάριστο τρόπο.

ΠΙΝΑΚΑΣ ΠΕΡΙΕΧΟΜΕΝΩΝ

ΚΑΤΑΛΟΓΟΣ ΣΧΗΜΑΤΩΝ

ΚΑΤΑΛΟΓΟΣ ΠΙΝΑΚΩΝ

Εισαγωγή

Τα τελευταία χρόνια, η ανάγκη για μεταφορά εφαρμογών από τους κλασσικούς προσωπικούς υπολογιστές στο χώρο των ενσωματωμένων εφαρμογών συνεχώς αυξάνει. Η διαδικασία της μεταφοράς αποτελείται από διάφορα επίπεδα. Σε αυτή τη διπλωματική εργασία ασχολούμαστε με το επίπεδο μνήμης και συγκεκριμένα, ρίχνουμε φως στο δυναμικό χαρακτήρα της παραχώρησης μνήμης και των προσπελάσεων δεδομένων.

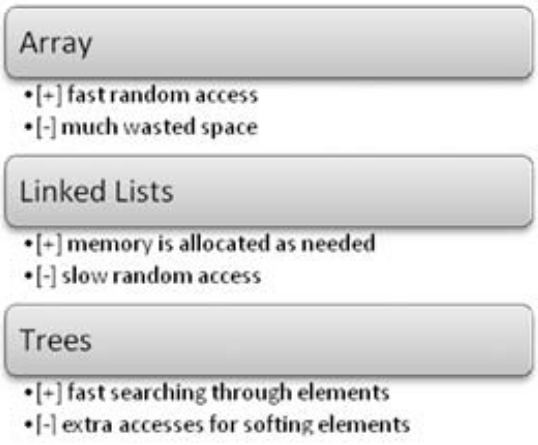
1.1. Ενσωματωμένα συστήματα και δυναμικές δομές δεδομένων

Κάθε εφαρμογή διαθέτει τα δικά της δυναμικά δεδομένα σε οντότητες που ονομάζονται δυναμικές δομές δεδομένων. Ο ρόλος αυτών των δομών είναι να παρέχουν μια διεπαφή (interface) ανάμεσα στην εφαρμογή και τα αντικείμενα που έχει δημιουργήσει στο χρόνο εκτέλεσης. Όλες οι δυναμικές δομές δεδομένων μπορούν να υλοποιηθούν με διάφορους τρόπους, η καθεμία παρουσιάζοντας διαφορετικά χαρακτηριστικά όσον αφορά τις επιδόσεις σε σχετικές σχεδιαστικές μετρικές. Για παράδειγμα, αναφέρουμε τις τρεις κύριες κατηγορίες δυναμικών δομών δεδομένων (Σχήμα 1), τους πίνακες (arrays), τις συνδεδεμένες λίστες (linked lists) και τα δέντρα (trees). Παρόλο που μια συνδεδεμένη λίστα μπορεί να θεωρηθεί ως ειδική περίπτωση δέντρου, γενικά διαφοροποιούμε αυτές τις δομές.

Οι πίνακες έχουν την ταχύτερη δυνατή προσπέλαση σε ένα συγκεκριμένο στοιχείο. Το στοιχείο αυτό μπορεί να είναι τυχαίο, οπότε οι πίνακες καθίστανται ιδανικοί σε περιπτώσεις που έχουμε τυχαίες προσπελάσεις σε στοιχεία μιας δομής δεδομένων. Ωστόσο, καθώς οι πίνακες απαιτούν την προδέσμευση μνήμης, μεγάλη ποσότητα μνήμης συχνά σπαταλείται. Από τη μία μεριά δυναμικοί πίνακες μπορούν να μεγεθυνθούν ή να συρρικνωθούν καθώς στοιχεία προστίθενται η αφαιρούνται, αλλά ακόμη, υπάρχει ένα σημαντικό συνήθως ποσοστό μνήμης που δε χρησιμοποιείται. Από την άλλη, οι συνδεδεμένες λίστες προσφέρουν εκχώρηση (ή ελευθέρωση) μνήμης καθώς προσθέτονται (ή αφαιρούνται) στοιχεία. Με άλλα λόγια, μπορεί να υπάρξει βέλτιστη αξιοποίηση της μνήμης. Το κύριο μειονέκτημα όμως είναι ότι σε κάθε προσπέλαση στοιχείου απαιτείται η αναζήτησή του από της αρχή της λίστας. Συνεπώς οι λίστες υστερούν σημαντικά σε απόδοση όσον αφορά την τυχαία προσπέλαση των στοιχείων τους.

Τέλος, τα δέντρα, ομοίως με τις λίστες, δε σπαταλούν χώρο μνήμης, καθώς εκχωρείται νέος όταν χρειαστεί, με την εισαγωγή νέου στοιχείου. Επιπλέον, τα δέντρα παρέχουν ένα γρηγορότερο τρόπο αναζήτησης, καθώς είναι εσωτερικά ταξινομημένα. Ωστόσο, και αυτή η ευκολία έχει το δικό της κόστος, καθώς η αύξηση σε λειτουργίες συνεπάγεται αύξηση σε προσπελάσεις μνήμης.

Το παρακάτω σχήμα συνοψίζει τα χαρακτηριστικά πινάκων, λιστών και δέντρων.



Σχήμα 1 - Σύγκριση πινάκων, λιστών και δέντρων

1.2. Ανάγκη για βελτιστοποίηση – μειονεκτήματα σημερινών προσεγγίσεων

Πολλή έρευνα έχει διεξαχθεί σε τεχνικές ανάλυσης και βελτιστοποίησης της χρήσης της μνήμης στα ενσωματωμένα συστήματα, ούτως ώστε να μειωθεί η κατανάλωση ενέργειας και να αυξηθεί απόδοση αυτών (βλέπε [1, 2]). Οι παραδοσιακές βελτιστοποιήσεις στα ενσωματωμένα συστήματα, χρησιμοποιούν πληροφορίες χρόνου μεταγλώττισης της εφαρμογής. Ο πηγαίος κώδικας μετασχηματίζεται σε μία νέα κοινώς αποδεκτή μορφή, ώστε να διευκολύνεται η ανάλυση [3]. Στις σύγχρονες εφαρμογές κάτι τέτοιο δεν είναι πλέον δυνατό, καθώς η δυναμική συμπεριφορά της εισόδου δε γίνεται εξ' ολοκλήρου από την ανάλυση κώδικα.

Όσον αφορά το profiling, τα περισσότερα εργαλεία δουλεύουν απευθείας στη μεταγλωττισμένη εφαρμογή χωρίς να απαιτούνται αλλαγές στον πηγαίο κώδικα. Εργαλεία όπως το prof [4], χρησιμοποιούν πληροφορίες αποσφαλμάτωσης (debugging) για να καταγράψουν το πλήθος κλήσεως κάθε συνάρτησης και το χρόνο παραμονής σε κάθε μία. Ωστόσο, δεν έχουν σχεδιαστεί να παρέχουν βελτιστοποιήσεις σχετικά με τα πρότυπα προσπέλασης μνήμης.

Στο [10] παρουσιάζεται ένα σύνολο εργαλείων για την αυτόματη βελτιστοποίηση των δυναμικών τύπων δεδομένων. Χρησιμοποιεί εξελικτική υπολογιστική καθοδηγούμενη από ένα σύνολο αναλυτικών μοντέλων για αξιολόγηση πιθανών υποψήφιων υλοποιήσεων για όλους τους τύπους των δεδομένων στην εφαρμογή. Ωστόσο, παρόλο που το μεγαλύτερο μερός του συνόλου εργαλείων είναι αυτοματοποιημένο, περιορίζεται από το γεγονός ότι αξιολογούνται πληροφορίες συμπεριφοράς. Τα αναλυτικά μοντέλα βασίζονται σε χαμηλού επιπέδου πληροφορίες όπως ο αριθμός των προσπελάσεων και το μέγεθος της μνήμης που χρησιμοποιήθηκε.

1.3. Καινοτομίες – συνεισφορά της διπλωματικής εργασίας

Σε αυτή τη διπλωματική εργασία, παρουσιάζουμε ένα αυτοματοποιημένο σύνολο εργαλείων που προσφέρει στο σχεδιαστεί των εφαρμογών όλες τις απαιτούμενες πληροφορίες για να συμπεράνει τον τρόπο με τον οποίο συμπεριφέρεται κάθε δυναμικών τύπος δεδομένων.

Τα εργαλεία που παρουσιάζονται είναι τα ακόλουθα:

- Μια νέα βιβλιοθήκη σχεδιασμένη εξ’ ολοκλήρου από την αρχή, με αρθρωτό (modular) τρόπο, ώστε ο συνδυασμός των βασικών τύπων δεδομένων σε πολυπλοκότερους να μπορεί να κλιμακωθεί χωρίς περιορισμούς.
- Ένας χαμηλού επιπέδου profiler που επεκτάθηκε για καταγραφή πληροφοριών στο επίπεδο διεπαφής (interface).
- Ένας αυτοματοποιημένος αναλυτής σχεδιασμένος εξ' ολοκλήρου από την αρχή για την ανάλυση και οργάνωση των πληροφοριών που καταγράφηκαν από τον profiler και την εξαγωγή πληθώρας στατιστικών για τη συμπεριφορά των δυναμικών δομών δεδομένων.

2. Δεδομένα και δομές δεδομένων

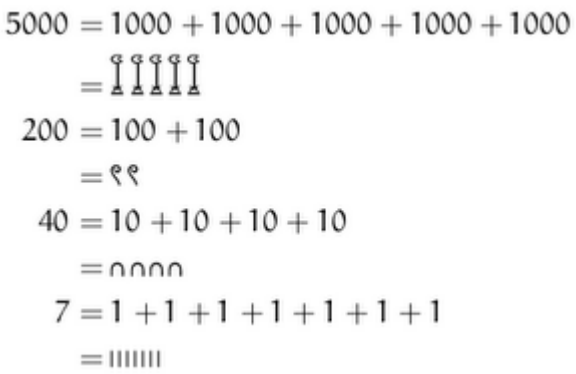
2.1. Η έννοια των δεδομένων

Όποτε αναφερόμαστε στη λειτουργία ενός προγράμματος, χρησιμοποιούμε λέξεις όπως «προσθήκη», «προσπέλαση», «πολλαπλασιασμός», «εγγραφή», κ.ο.κ. Η λειτουργία ενός προγράμματος περιγράφει το σκοπό του, με όρους που θα μπορούσαμε να πούμε ότι αποτελούν τα ρήματα μιας γλώσσας προγραμματισμού. Τα δεδομένα είναι τα ουσιαστικά του προγραμματιστικού κόσμου, δηλαδή τα αντικείμενα που χειρίζεται το πρόγραμμα, οι πληροφορίες που αυτό επεξεργάζεται. Κατά μια έννοια, η πληροφορία αυτή είναι απλά ένα σύνολο δυαδικών ψηφίων με τιμή 0 ή 1. Αυτή είναι η μορφή που απαιτεί ένας υπολογιστής για τα δεδομένα του. Ωστόσο, οι άνθρωποι τείνουν να αναφέρονται στην πληροφορία με όρους όπως οι αριθμοί ή οι λίστες, συνεπώς, θα θέλαμε τα προγράμματά μας να αναφέρονται στα δεδομένα με τρόπο κατανοητό σε μας. Έτσι, χρησιμοποιούμε αφαιρετικές τεχνικές για να διαχωρίσουμε τη δικιά μας αντίληψη των δεδομένων από αυτή των υπολογιστών. Η τεχνική αυτή ονομάζεται αφαίρεση δεδομένων - data abstraction. Η αφαίρεση δεδομένων είναι απαραίτητη στην περίπτωση που χρησιμοποιούμε είτε συναρτησιακή αποσύνθεση (functional decomposition) για να δημιουργήσουμε μια ιεραρχία λειτουργιών, είτε αντικειμενοστραφή σχεδιασμό για τη δημιουργία αντικειμένων με ιεραρχική δομή.

2.2. Αφαίρεση Δεδομένων

Συνήθως οι άνθρωποι νιώθουν πιο άνετα με πράγματα που έχουν πραγματική υπόσταση, από ότι με κάποια που τα αντιλαμβανόμαστε ως αφηρημένα. Συνεπώς, η χρήση της αφαίρεσης δεδομένων μπορεί να φανεί περιοριστική κατά κάποιον τρόπο, σε σχέση με μια οντότητα πιο συνηθισμένη όπως είναι ο τύπος μεταβλητής integer. Στην πραγματικότητα όμως, ο τύπος αυτός είναι από μόνος του μια αφηρημένη έννοια, όπως θα δούμε παρακάτω.

Οι ακέραιοι αναπαριστώνται με διαφορετικούς τρόπους από υπολογιστή σε υπολογιστή. Ενώ στη μνήμης της μιας μηχανής μπορεί ένας ακέραιος να αναπαρίσταται ως δυαδικά κωδικοποιημένος δεκαδικός αριθμός, στη μνήμη μιας άλλης μπορεί να είναι προσημασμένος δυαδικός αριθμός και σε μια τρίτη να είναι αναπαράσταση με συμπλήρωμα ως προς ένα ή δύο. Η αδυναμία της ακριβής κατανόησης αυτών των αναπαραστάσεων δεν εμποδίζει σε καμία περίπτωση τη χρήση των ακεραίων.



Σχήμα 2 - Το Αιγυπτιακό Σύστημα Αρίθμησης

Ο τρόπος αναπαράστασης των ακεραίων καθορίζει τον τρόπο, με τον οποίο ο υπολογιστής χειρίζεται τα δεδομένα. Ένας προγραμματιστής όμως σπάνια χρειάζεται να γνωρίζει τόσες λεπτομέρειες (εκτός κι αν δουλεύει σε επίπεδο assembly). Εκείνο όμως, που πρέπει να γνωρίζει για να χρησιμοποιήσει τους ακεραίους είναι ποιες λειτουργίες μπορεί να επιτελέσει με τον συγκεκριμένο τύπο δεδομένων. Ας θεωρήσουμε την ακόλουθη δήλωση:

distance = rate * time;

Η έννοια της παραπάνω πρότασης είναι εύκολα κατανοητή. Πρόκειται για τον πολλαπλασιασμό δύο μεταβλητών. Η έννοια του πολλαπλασιασμού φυσικά, δεν εξαρτάται από το αν οι τελεστές είναι για παράδειγμα ακέραιοι ή πραγματικοί αριθμοί, παρόλο που η υλοποίηση της συγκεκριμένης πράξης για κάθε δυάδα τελεστών μπορεί να γίνει με διάφορους τρόπους στον ίδιο υπολογιστή. Η χρήση των υπολογιστών δεν θα ήταν τόσο διαδεδομένη, αν κάθε φορά που θέλαμε να πολλαπλασιάσουμε δύο αριθμούς έπρεπε να δουλέψουμε με αναπαραστάσεις αριθμών σε επίπεδο μηχανής. Ευτυχώς όμως, οι γλώσσες προγραμματισμού όπως και η C++ έχουν αποκρύψει τις πληροφορίες υλοποίησης του τύπου int με τέτοιο τρόπο, ώστε να παρέχουν στο χρήστη μόνο εκείνες τις πληροφορίες που αφορούν τη δημιουργία και χρήση δεδομένων αυτού του τύπου.

Η έννοια που περιγράφει αυτήν ακριβώς την απόκρυψη της πλεονάζουσας πληροφορίας, είναι η έννοια της ενθυλάκωσης (encapsulation). Η ενθυλάκωση δεδομένων δηλώνει ότι η φυσική αναπαράσταση των δεδομένων ενός προγράμματος αποκρύπτεται από το χρήστη των δεδομένων. Έτσι αυτός, δε γνωρίζει την εκάστοτε υλοποίηση, αλλά «βλέπει» τα δεδομένα στη λογική τους απεικόνιση.

Η ενθυλάκωση των δεδομένων δημιουργεί την πρόσθετη ανάγκη για την παροχή κατάλληλων συναρτήσεων για την δημιουργία, πρόσβαση και μεταβολή των δεδομένων. Ας πάρουμε για παράδειγμα τις λειτουργίες που παρέχει η C++ για τον ενθυλακωμένο τύπο δεδομένων int. Αρχικά, μπορούμε να δημιουργήσουμε (construct) μεταβλητές τύπου int με τη χρήση των προτάσεων δήλωσης (declarations) στο πρόγραμμά μας. Ύστερα, μπορούμε να εκχωρήσουμε τιμές σε αυτές τις ακέραιες μεταβλητές με τη χρήση του τελεστή ανάθεσης και να εκτελέσουμε μαθηματικές λειτουργίες με τη χρήση των τελεστών +, -, *, /, και %.

Όλα τα παραπάνω αποσκοπούν στο να γίνει ξεκάθαρο ότι η αφαίρεση δεδομένων δεν είναι κάτι καινούργιο, αλλά κάτι με το οποίο ερχόμαστε σε επαφή συνέχεια, χωρίς απλώς να το γνωρίζουμε. Σα πλεονεκτήματά της είναι σαφή: μπορούμε να αντιλαμβανόμαστε και να χειριζόμαστε τα δεδομένα στη λογική τους υπόσταση, χωρίς να χρειάζεται να ανησυχούμε για τις λεπτομέρειες της υλοποίησης. Σα χαμηλότερα επίπεδα συνεχίζουν να υφίστανται φυσικά, απλώς παραμένουν κρυμμένα για τον τελικό χρήστη.

Ο στόχος κατά το σχεδιασμό ενός προγράμματος είναι να μειωθεί η πολυπλοκότητα αυτού με τη χρήση αφαιρετικών τεχνικών. Αυτός ο στόχος μπορεί να επεκταθεί και παραπέρα: να προστατευθεί η αφαίρεση των δεδομένων με τη χρήση της ενθυλάκωσης αυτών. Όλες οι δυνατές τιμές ενός αντικειμένου συγκεκριμένου τύπου δεδομένων, μαζί με τις προδιαγραφές των λειτουργιών που παρέχονται για τη δημιουργία και το χειρισμό του συγκεκριμένου τύπου δεδομένων, ορίζονται ως *αφηρημένος τύπος δεδομένων (Abstract data type – ADT).*

2.3. Δομές Δεδομένων

Ένας ακέραιος μπορεί να φανεί πολύ χρήσιμος στην περίπτωση που χρειαζόμαστε έναν μετρητή, ένα άθροισμα ή έναν δείκτη, αλλά στα προγράμματά μας, γενικά, θα χρειαστεί να χειριστούμε δεδομένα που αποτελούνται από πολλά τμήματα, όπως είναι μια λίστα. Σις λογικές ιδιότητες ενός τέτοιου συνόλου δεδομένων τις περιγράφουμε ως αφηρημένο τύπο δεδομένων, ενώ μια συγκεκριμένη υλοποίηση αυτών ως δομή δεδομένων. Προγράμματα των οποίων η πληροφορία αποτελείται από πολλά σύνθετα μέρη απαιτούν τη χρήση μιας κατάλληλης δομής δεδομένων.

Οι δομές δεδομένων έχουν κάποια αξιοσημείωτα χαρακτηριστικά. Πρώτον, μπορούν να αποσυντεθούν σε συστατικά στοιχεία. Δεύτερον, η διάταξη των στοιχείων είναι ένα χαρακτηριστικό της δομής που επιδρά άμεσα στον τρόπο που αυτά είναι προσβάσιμα. Τρίτον, τόσο η διάταξη των στοιχείων, όσο και ο τρόπος που αυτά είναι προσβάσιμα, μπορούν να υποστούν ενθυλάκωση.

Ας εξετάσουμε ένα πραγματικό παράδειγμα, μια βιβλιοθήκη. Μια βιβλιοθήκη λοιπόν, μπορεί να αποσυντεθεί στα συστατικά της στοιχεία, που είναι τα βιβλία. Η συλλογή των βιβλίων μπορεί να ταξινομηθεί με ένα σύνολο διαφορετικών τρόπων, όπως φαίνεται στο Σχήμα 2.3. Προφανώς, ο τρόπος που τα βιβλία είναι ταξινομημένα στα ράφια στην πραγματικότητα, καθορίζει τον τρόπο με τον οποίο κάποιος θα ψάξει για ένα συγκεκριμένο βιβλίο. Στη βιβλιοθήκη του παραδείγματός μας, οι πελάτες δεν επιτρέπεται να ψάξουν μόνοι τους για ένα βιβλίο, αντ' αυτού, δίνουν την αίτηση στο βιβλιοθηκάριο κι εκείνος φέρνει το βιβλίο.

Η «δομή δεδομένων» βιβλιοθήκη αποτελείται από τα δομικά στοιχεία (βιβλία), που βρίσκονται σε μια συγκεκριμένη διάταξη στο χώρο. Για να βρεθεί ένα βιβλίο, απαιτείται η γνώση του τρόπου ταξινόμησής τους στην εκάστοτε βιβλιοθήκη. Ο επισκέπτης της βιβλιοθήκης, ωστόσο, δε χρειάζεται να γνωρίζει τίποτα για την εσωτερική της δομή, γιατί η τελευταία έχει αποκρυφτεί: οι χρήστες έχουν πρόσβαση στα βιβλία μόνο μέσω του βιβλιοθηκάριου. Η φυσική δομή και η αφηρημένη υπόσταση των βιβλίων στη βιβλιοθήκη

μπορεί να μην είναι ίδιες. Ο κατάλογος που διατίθεται στους επισκέπτες παρέχει διάφορες λογικές απεικονίσεις της βιβλιοθήκης – ταξινομημένες κατά θέμα, συγγραφέα ή τίτλο – οι οποίες διαφέρουν από τη φυσική της διάταξη

Την ίδια προσέγγιση χρησιμοποιούμε και στις δομές δεδομένων των προγραμμάτων μας. Μια δομή δεδομένων καθορίζεται από τη λογική διάταξη των στοιχείων που την απαρτίζουν, μαζί με το σύνολο των λειτουργιών που απαιτούνται για την πρόσβαση σε αυτά.

Εδώ πρέπει να σημειώσουμε τη διαφορά μεταξύ ενός αφηρημένου τύπου δεδομένων και μιας δομής δεδομένων. Ένα αφηρημένος τύπος δεδομένων αποτελεί περιγραφή υψηλού επιπέδου. Πρόκειται για τη λογική απεικόνιση των δεδομένων και των λειτουργιών για το χειρισμό τους. Από την άλλη, μια δομή δεδομένων έχει πραγματική υπόσταση και αποτελεί ένα σύνολο στοιχείων και τις λειτουργίες για την αποθήκευση και ανάκτηση αυτών. Ο αφηρημένος τύπος δεδομένων είναι ανεξάρτητος συγκεκριμένης υλοποίησης, ενώ η δομή δεδομένων εξαρτάται άμεσα από αυτή. Μια δομή δεδομένων είναι ο τρόπος που υλοποιούνται τα δεδομένα ενός αφηρημένου τύπου δεδομένων, του οποίου οι τιμές έχουν συνθετικά στοιχεία. Οι λειτουργίες που επιτελούνται σε κάποιον αφηρημένο τύπο δεδομένων αντιστοιχούν σε αλγορίθμους που ενεργούν σε μια δομή δεδομένων.

Κατά τη μοντελοποίηση των δεδομένων ενός προγράμματος πρέπει να καθορίσουμε τη λογική απεικόνιση των δεδομένων, να επιλέξουμε την αναπαράστασή τους και να αναπτύξουμε τις λειτουργίες εκείνες που θα χρησιμοποιηθούν για την απόκρυψη της πραγματικής διάταξής τους. Για τη διαδικασία αυτή, θα χειριστούμε τα δεδομένα σε τρία διαφορετικά επίπεδα αφαίρεσης (τρεις διαφορετικοί τρόποι αντίληψης αυτών):

ΕΠΙΠΕΔΟ ΕΦΑΡΜΟΓΗΣ Η ΧΡΗΣΤΗ (APPLICATION OR USER LEVEL): Ένας τρόπος μοντελοποίησης δεδομένων, που αφορούν συγκεκριμένο αντικείμενο της καθημερινής ζωής μας.

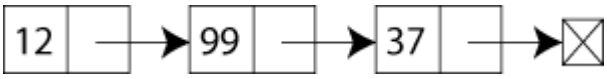
ΛΟΓΙΚΟ (Η ΑΦΗΡΗΜΕΝΟ) ΕΠΙΠΕΔΟ (LOGICAL OR ABSTRACT LEVEL): Μια αφηρημένη άποψη των τιμών των δεδομένων και των λειτουργιών που χειρίζονται αυτά.

ΕΠΙΠΕΔΟ ΥΛΟΠΟΙΗΣΗΣ (IMPLEMENTATION LEVEL): Μια συγκεκριμένη αναπαράσταση της δομής που αποθηκεύει τα δεδομένα, και ο κώδικας των λειτουργιών σε κάποια γλώσσα προγραμματισμού (εφόσον βέβαια αυτές οι λειτουργίες δεν παρέχονται εξ' ορισμού από τη γλώσσα αυτή).

Στη κείμενο της συγκεκριμένης διπλωματικής εργασίας αναφερόμαστε στο δεύτερο επίπεδο με την έννοια «αφηρημένος τύπος δεδομένων». Ταυτόχρονα χρησιμοποιούμε τον όρο «σύνθετος τύπος δεδομένων (composite data type)» για να αναφερθούμε σε τύπους δεδομένων που περιέχουν συνθετικά στοιχεία. Σο τρίτο επίπεδο περιγράφει, τον τρόπο με τον οποίο αναπαριστούμε και χειριζόμαστε τα δεδομένα στη μνήμη, δηλαδή τη δομή δεδομένων και τους αλγορίθμους για τις λειτουργίες χειρισμού των αντικειμένων της δομής.

3. Βασικοί Αφηρημένοι Τύποι Δεδομένων

3.1. Λίστα



Σχήμα 3 - Παράδειγμα Απλά Συνδεδεμένης Λίστας

Οι λίστες είναι πολύ χρήσιμοι αφηρημένοι τύποι δεδομένων στα προγράμματα των υπολογιστών. Είναι μέλη μιας γενικότερης κατηγορίας αφηρημένων τύπων δεδομένων, που αποκαλούνται περιέχοντες τύποι (containers) και σκοπός των οποίων είναι να κρατούν άλλα αντικείμενα. Σε κάποιες γλώσσες, η λίστα είναι ενσωματωμένος τύπος. Στη Lisp για παράδειγμα, η λίστα αποτελεί την κεντρική δομή δεδομένων που παρέχεται από τη γλώσσα.

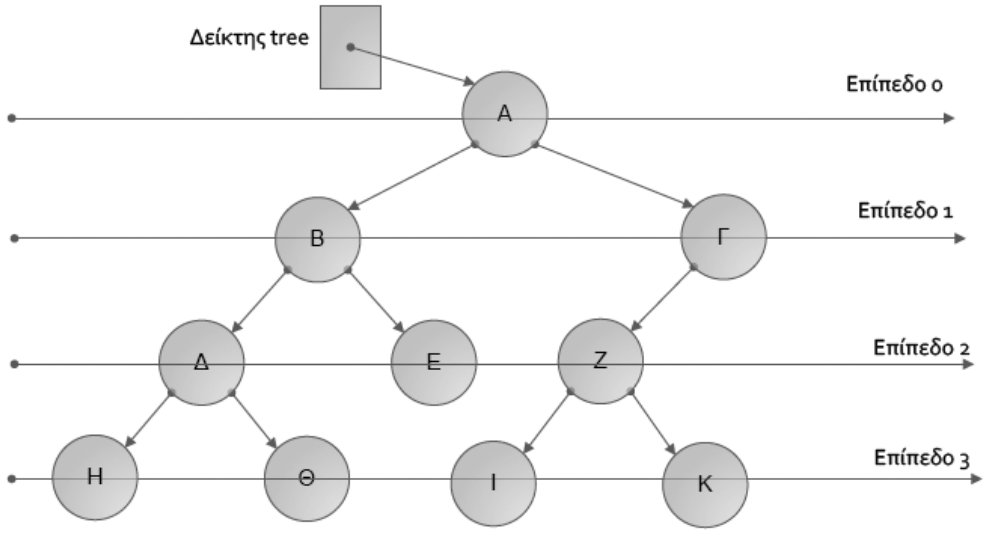
Από τη θεωρητική πλευρά, μια λίστα είναι μια ομογενής συλλογή στοιχείων και διέπεται από μια γραμμική σχέση μεταξύ των στοιχείων της. *Γραμμική*, σημαίνει ότι στο λογικό επίπεδο, κάθε στοιχείο της λίστας εκτός του πρώτου, έχει έναν μοναδικό προηγούμενο (predecessor), και κάθε στοιχείο εκτός του τελευταίου, έχει έναν και μοναδικό διάδοχο. (Στο επίπεδο της υλοποίησης, μεταξύ των στοιχείων υφίσταται και πάλι μια σχέση μεταξύ των στοιχείων, μόνο που η φυσική τους σχέση μπορεί να είναι διαφορετική από τη λογική.) Ο αριθμός των στοιχείων μιας λίστας, τον οποίο αποκαλούμε μήκος της λίστας, είναι μια ιδιότητα της λίστας. Αυτό σημαίνει ότι κάθε λίστα χαρακτηρίζεται από ένα μήκος.

Οι λίστες μπορεί να είναι μη ταξινομημένες (unsorted), δηλαδή τα στοιχεία τους μπορεί να τοποθετηθούν σε οποιαδήποτε σειρά μέσα στη λίστα, ή να είναι ταξινομημένες (sorted) με μια πληθώρα τρόπων. Για παράδειγμα, μια λίστα αριθμών μπορεί να ταξινομηθεί κατά τιμή, μια λίστα συμβολοσειρών μπορεί να ταξινομηθεί αλφαβητικά, και μια λίστα βαθμών αξιολόγησης μπορεί να ταξινομηθεί αριθμητικά. Όταν τα στοιχεία σε μια ταξινομημένη λίστα είναι στοιχεία σύνθετων τύπων, η λογική (συχνά και η φυσική) σειρά καθορίζεται από ένα μέλος της δομής τους, που αποκαλείται κλειδί. Για παράδειγμα, μια λίστα μαθητών μπορεί να ταξινομηθεί αλφαβητικά κατά όνομα ή αριθμητικά κατά τον αναγνωριστικό αριθμό ID του καθενός. Στην πρώτη περίπτωση, κλειδί είναι το όνομα, ενώ στη δεύτερη, κλειδί αποτελεί ο αναγνωριστικός αριθμός. Τέτοιες ταξινομημένες λίστες αποκαλούνται *ταξινομημένες-κατά-κλειδί*. Στην περίπτωση που μια λίστα δεν μπορεί να περιέχει το ίδιο κλειδί δύο φορές, λέμε ότι έχει *μοναδικά* κλειδί

3.2. Δέντρο

Ένα δυαδικό δέντρο αναζήτησης αποτελεί μια δομή με δύο ιδιότητες: μια που αφορά στο σχήμα και μια που αντιστοιχεί τα κλειδιά των στοιχείων μέσα στη δομή. Πρώτα θα αναφερθούμε στην ιδιότητα τη σχετική με το σχήμα.

Κάθε κόμβος σε μια απλά συνδεδεμένη λίστα μπορεί να οδηγεί σε έναν κόμβο, τον κόμβο που τον ακολουθεί. Έτσι μια απλά συνδεδεμένη λίστα αποτελεί μια γραμμική δομή. Κάθε κόμβος στη λίστα (εκτός από τον τελευταίο) έχει έναν μοναδικό διάδοχο. Αντίθετα ένα δυαδικό δέντρο είναι μια δομή στην οποία κάθε κόμβος μπορεί να έχει δύο «παιδιά» (children). Κάθε ένας από αυτούς, καθώς αποτελούν κόμβους σε ένα δυαδικό δέντρο, μπορούν να έχουν από δύο άλλα παιδιά και τα τελευταία μπορούν να έχουν από δύο ακόμη παιδιά, κλπ, κι έτσι σχηματίζεται η δομή του δέντρου. Η αρχή του δέντρου είναι ένα μοναδικός αρχικός κόμβος που ονομάζεται ρίζα.

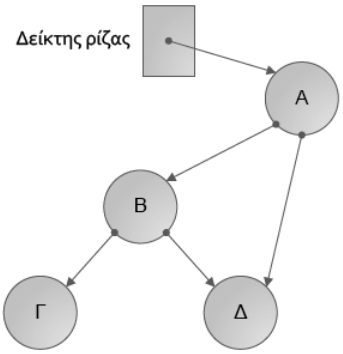


Σχήμα 4 - Παράδειγμα Δυαδικού Δέντρου

Στο Σχήμα 4 φαίνεται ένα δυαδικό δέντρο. Ο κόμβος-ρίζα περιέχει την τιμή Α. Κάθε κόμβος στο δέντρο μπορεί να έχει 1, 2 ή 3 παιδιά. Ο κόμβος στα αριστερά του γονικού κόμβου ονομάζεται, εφόσον υπάρχει, αριστερό παιδί. Για παράδειγμα, το αριστερό παιδί της ρίζας του δέντρου περιέχει την τιμή Β. Η ρίζα είναι ο «γονέας» (parent) των κόμβων που έχουν τις τιμές Β και Γ. Εάν κάποιος κόμβος σε ένα δέντρο δεν έχει απογόνους ονομάζεται «φύλλο». Συγκεκριμένα, οι κόμβοι που περιέχουν τις τιμές Η, Θ, Ε, Ι, και Κ είναι φύλλα.

Ο ορισμός ενός δυαδικού δέντρου δεν προσδιορίζει μόνο το ότι κάθε κόμβος έχει δύο παιδιά, αλλά μας λέει ότι υπάρχει μόνο μια διαδρομή από τη ρίζα σε κάθε άλλο κόμβο. Επομένως κάθε κόμβος, εκτός από τη ρίζα, έχει ένα μοναδικό γονέα. Σε μια δομή όπως αυτή που φαίνεται στο Σχήμα 5, οι κόμβοι έχουν τον κατάλληλο αριθμό από παιδιά αλλά η συνθήκη της μοναδικής διαδρομής δεν ισχύει. Υπάρχουν δύο διαδρομές από τη ρίζα μέχρι

τον κόμβο που περιέχει το Δ. Έτσι η δομή δεν μπορεί να είναι δέντρο, πόσο μάλλον δυαδικό δέντρο

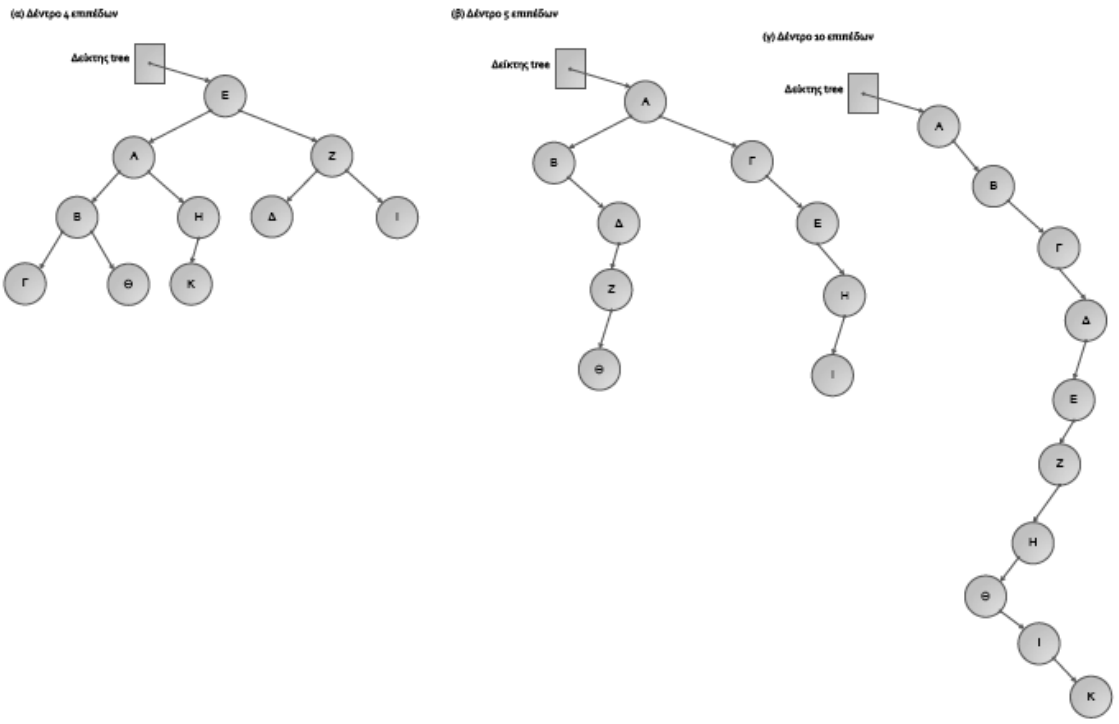


Σχήμα 5 - Παράδειγμα Μη Δυαδικού Δέντρου

Στο Σχήμα 4 κάθε παιδί του κόμβου-ρίζα είναι ρίζα ενός μικρότερου δυαδικού δέντρου, ή αλλιώς υποδέντρου. Για την ακρίβεια κάθε κόμβος του δέντρου μπορεί να θεωρηθεί ως ρίζα ενός υποδέντρου. Το υποδέντρο του οποίου η ρίζα έχει την τιμή Β περιέχει και τους κόμβους με τιμές Δ, Η, Θ και Ε. Οι κόμβοι αυτοί όλοι είναι «μεταγενέστεροι» του κόμβου που περιέχει το Β. Ένας κόμβος ονομάζεται «πρόγονος» ενός άλλου κόμβου αν εάν είναι ο γονέας αυτού ή ο γονέας κάποιου από τους προγόνους του κόμβου. (Πρόκειται για έναν αναδρομικό ορισμό). Στο Σχήμα 4, οι πρόγονοι του κόμβου με την τιμή Η είναι οι κόμβοι που περιέχουν τις τιμές Δ, Β και Α. Προφανώς η ρίζα του δέντρου είναι πρόγονος οποιουδήποτε κόμβου στο δέντρο.

Το επίπεδο ενός κόμβου αναφέρεται στην απόσταση του από τη ρίζα. Αν ορίσουμε το επίπεδο της ρίζας ίσο με μηδέν (0), οι κόμβοι που περιέχουν τις τιμές Β και Γ είναι κόμβοι επιπέδου 1. Οι κόμβοι που περιέχουν τις τιμές Δ, Ε και Ζ αποτελούν κόμβους επιπέδου 2. Τέλος, οι κόμβοι που περιέχουν τα Η, Θ, Ι, και Κ είναι κόμβοι επιπέδου 3.

Η μέγιστη τιμή που μπορεί να πάρει το επίπεδο ενός κόμβου σε ένα δέντρο προσδιορίζει το ύψος του. Ο μέγιστος αριθμός κόμβων σε ένα δέντρο ύψους Ν είναι 2^N. Συχνά όμως σε ένα επίπεδο δεν περιέχεται ο μέγιστος αριθμός κόμβων. Για παράδειγμα, στο Σχήμα 4, το επίπεδο 2 θα μπορούσε να περιέχει τέσσερις κόμβους αλλά επειδή ο κόμβος με την τιμή 1 έχει μόνο ένα παιδί, το επίπεδο 2 αποτελείται από 3 κόμβους. Το επίπεδο 3 το οποίο θα μπορούσε να περιέχει οκτώ κόμβους τώρα έχει μόνο τέσσερις. Θα μπορούσαμε να κατασκευάσουμε πολλά δέντρα με διαφορετικό σχήμα από τους δέκα κόμβους αυτού του δέντρου. Στο Σχήμα 6 παρουσιάζονται ορισμένες παραλλαγές. Είναι προφανές ότι ο μέγιστος αριθμός επιπέδων σε ένα δυαδικό δέντρο με Ν κόμβους είναι Ν. Ποιός είναι όμως ο ελάχιστος αριθμός επιπέδων; Εάν γεμίσουμε το δέντρο τοποθετώντας δύο απογόνους σε κάθε κόμβο μέχρι να τελειώσει ο διαθέσιμος αριθμός από κόμβους, το δέντρο θα έχει $\log_2 N+1$ επίπεδα (Σχήμα 6(α)).



Σχήμα 6 - Δυαδικά Δέντρα

Το ύψος του δέντρου είναι ο παράγοντας εκείνος που καθορίζει το βαθμό ευκολίας στην αναζήτηση στοιχείων. Για παράδειγμα θεωρήσουμε το δέντρο μέγιστου ύψους του σχήματος Σχήμα 6(γ). Αν ξεκινήσουμε από τη ρίζα και ακολουθούμε τους δείκτες από έναν κόμβο στον επόμενο, μέχρι να φτάσουμε στον κόμβο που περιέχει την τιμή Κ (τον πιο μακρινό από τη ρίζα) θα είναι ουσιαστικά σαν να κάνουμε αναζήτηση σε μια γραμμική λίστα. Από την άλλη, για να φτάσουμε στον ίδιο κόμβο στο δέντρο ελαχίστου ύψους του σχήματος Σχήμα 6(α), θα προσπελάσουμε μόνο τρεις κόμβους, εκείνους που περιέχουν τα Ε, Α και Ζ προκειμένου να φτάσουμε στο Κ.

Ο τρόπος με τον οποίο τοποθετούνται οι τιμές στο δέντρο του σχήματος Σχήμα 6(α) δεν βοηθάει για να γρήγορη αναζήτηση. Ας υποθέσουμε ότι θέλουμε να βρούμε την τιμή Z. Αρχίζουμε την αναζήτηση στη ρίζα. Ο κόμβος αυτός περιέχει την τιμή Ε, όχι την Z οπότε πρέπει να συνεχίσουμε να ψάχνουμε. Όμως ποιο παιδί θα ψάξουμε πρώτα, τον δεξιό ή τον αριστερό; Επειδή οι κόμβοι δεν είναι τακτοποιημένοι με κάποιο συγκεκριμένο τρόπο, θα πρέπει να εξετάσουμε και τα δύο υποδέντρα. Θα μπορούσαμε βέβαια να εξετάζουμε το δέντρο ανά επίπεδο, μέχρι να βρούμε την επιθυμητή τιμή αλλά αυτός ο τρόπος αναζήτησης εξακολουθεί να μην είναι καλύτερος από το να ψάχνουμε μια γραμμική λίστα.

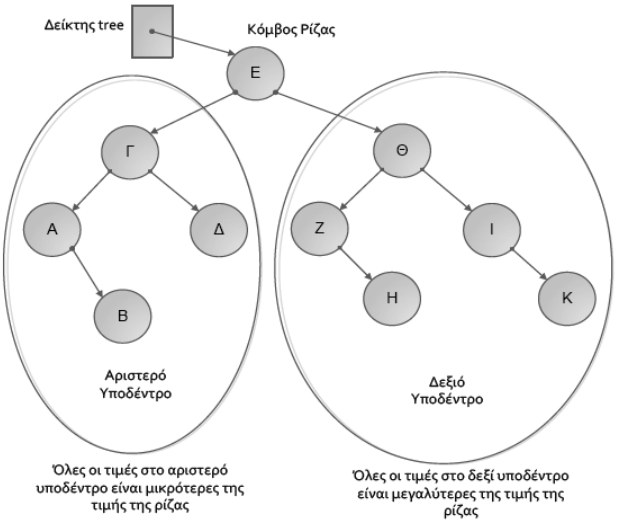
Ο στόχος είναι ο αριθμός των προσπελάσεων να προσεγγίζει την τιμή log2N+1 και γι’ αυτό θα προσθέσουμε μια ιδιότητα η οποία στηρίζεται στη σχέση ανάμεσα στα κλειδιά των διαφόρων ειδών του δέντρου. Τοποθετούμε όλους τους κόμβους με τιμές μικρότερες από την τιμή της ρίζας στο δεξιό υποδέντρο και αυτούς με τιμές μεγαλύτερες από την τιμή της ρίζας στο αριστερό υποδέντρο. Σο Σχήμα 7 δείχνει τους κόμβους από το Σχήμα 6(α) που έχουν επανατοποθετηθεί ώστε να ικανοποιούν αυτή την ιδιότητα. Η ρίζα η οποία περιέχει το Ε, έχει πρόσβαση σε δύο υποδέντρα. Σο αριστερό υποδέντρο περιλαμβάνει όλες τις τιμές που είναι μικρότερες από την τιμή Ε και το δεξιό περιλαμβάνει όλες τις μεγαλύτερες.

Για να αναζητήσουμε την τιμή Η πρώτα εξετάζουμε τον κόμβο-ρίζα. Η τιμή Η είναι μεγαλύτερη από την Ε επομένως μας ενδιαφέρει το δεξιό υποδέντρο της ρίζας. Σο δεξιό παιδί της ρίζας περιέχει την τιμή Η. Η τιμή αυτή είναι μεγαλύτερη από το Η επομένως συνεχίζουμε την αναζήτηση στα αριστερά. Σο αριστερό παιδί του κόμβου περιέχει την τιμή Ζ που είναι μικρότερη από την Γ και συνεχίζοντας τον κανόνα προχωρούμε δεξιά. Ο κόμβος στα δεξιά περιέχει την τιμή Η. Έχουμε βρει δηλαδή τον κόμβο που αναζητούσαμε.

Ένα δυαδικό δέντρο με την ανωτέρω ιδιότητα ονομάζεται δυαδικό δέντρο αναζήτησης. Όπως κάθε δυαδικό δέντρο, η δομή του επιτυγχάνεται με το να έχει κάθε κόμβος το πολύ δύο παιδιά. Η δομή εύκολης αναζήτησης επιτυγχάνεται μέσω της δυαδικής ιδιότητας αναζήτησης: Το αριστερό παιδί κάθε κόμβου (εφόσον αυτός υπάρχει) είναι η ρίζα ενός υποδέντρου που περιέχει τιμές μόνο μικρότερες από αυτήν του κόμβου. Αντίστοιχα το δεξιό παιδί κάθε κόμβου (εφόσον αυτός υπάρχει) είναι η ρίζα ενός υποδέντρου που περιέχει τιμές μόνο μεγαλύτερες από αυτήν του κόμβου.

Τέσσερις συγκρίσεις αντί για δέκα δεν ακούγεται ως κάτι αξιόλογο, αλλά αν ο αριθμός των στοιχείων της δομής αυξηθεί, η διαφορά είναι πραγματικά εντυπωσιακή. Στη χειρότερη περίπτωση που αφορά την αναζήτηση του τελευταίου κόμβου σε μια γραμμική λίστα, πρέπει να ψάξουμε όλη τη λίστα. Στη γενική περίπτωση χρειάζεται να ψάξουμε τη μισή. Εάν η λίστα περιέχει 1,000 κόμβους πρέπει να κάνουμε 1,000 συγκρίσεις για να βρούμε τον τελευταίο κόμβο. Εάν οι 1,000 κόμβοι έχουν τοποθετηθεί σε ένα δυαδικό δέντρο ελαχίστου ύψους ποτέ δεν θα γίνουν περισσότερες από 10 ($\log_2(1{,}000)<10$) συγκρίσεις, ανεξάρτητα με την τιμή που αναζητούμε.

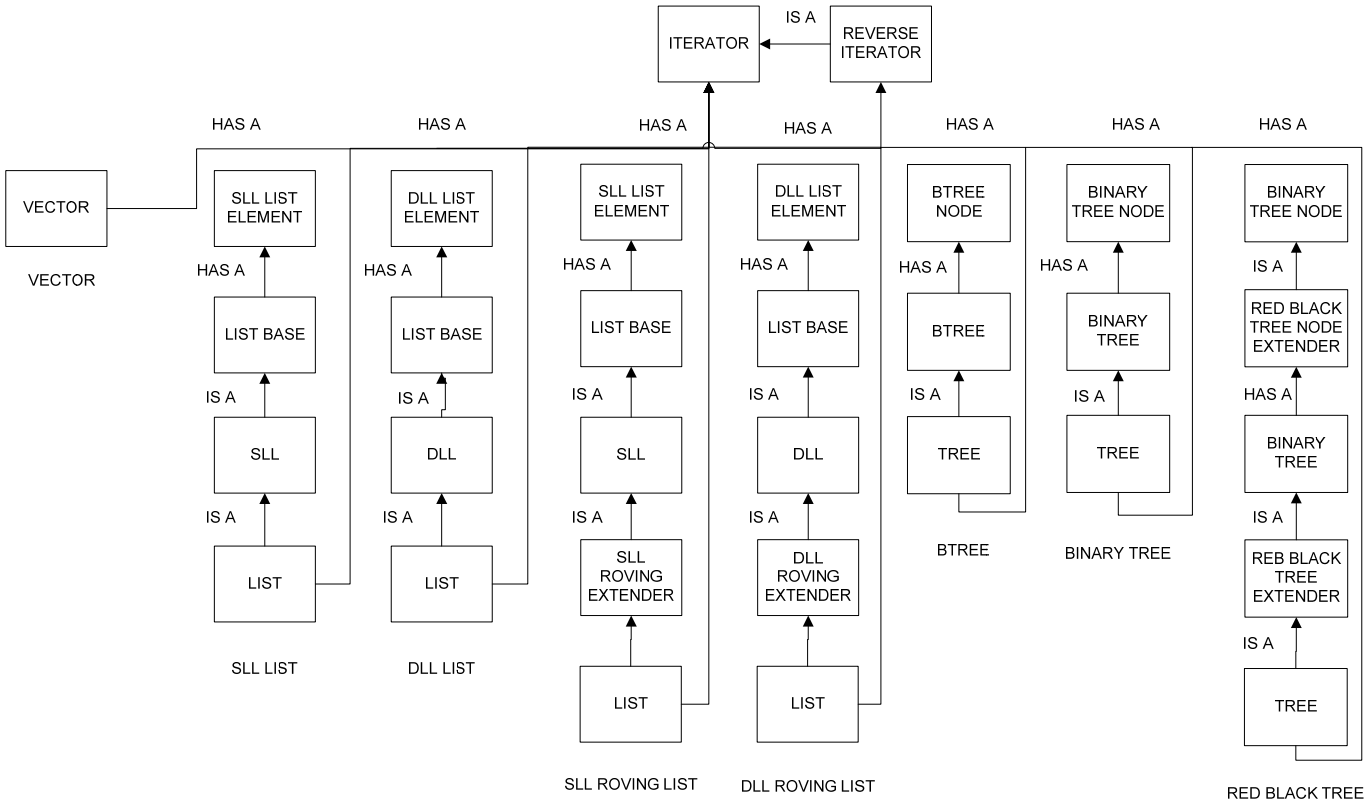
Οι ιδιότητες που έχουμε παραθέσει για τα δυαδικά δέντρα μπορούν να επεκταθούν για τα δέντρα των οποίων οι κόμβοι μπορούν να έχουν οποιονδήποτε αριθμό απογόνων. Έτσι λοιπόν ένα δέντρο είναι μια δομή με ένα μοναδικό αρχικό σημείο (ρίζα), στην οποία κάθε κόμβος μπορεί να έχει πολλά παιδιά και υπάρχει μια μοναδική διαδρομή από τη ρίζα σε κάθε άλλο κόμβο.



Σχήμα 7 - Ένα Δυαδικό Δέντρό

4. Σχεδιασμός μια κεντρικής βιβλιοθήκης για τους τύπους δεδομένων ενσωματωμένων συστημάτων

4.1. Σχεδιαστικοί κανόνες



Σχήμα 8 - Η Ιεραρχία Κλάσεων της Βιβλιοθήκης

Η βιβλιοθήκη που υλοποιήσαμε (DDT Library) σχεδιάστηκε σύμφωνα με τους παρακάτω κανόνες:

- Να είναι αρθρωτή (Modularity)
- Να είναι Αποδοτική στη λειτουργία της (Efficiency)
- Να είναι πλήρως συμβατή σε επίπεδο interface με την C++ STL (STL compatibility)

Αρθρωτότητα

Η υλοποίηση της βιβλιοθήκης διαιρέθηκε σε διάφορα ημιαυτόνομα τμήματα. Η χρήση κάθε τμήματος προσθέτει λειτουργικότητα στη δυναμική δομή δεδομένων που επιθυμούμε να χτίσουμε με μία σειρά δηλώσεων (τεχνική Mixing). Αυτή η τεχνική ορίζει το παρακάτω σχεδιαστικό πρότυπο:

```
template <class extendee> class Extender: public extendee
```

Αποδοτικότητα

Στην αποδοτικότητα δόθηκε ειδικό βάρος και γι' αυτό το λόγο η βιβλιοθήκη σχεδιάστηκε και υλοποιήθηκε δύο φορές. Τη δεύτερη φορά αντιμετωπίστηκαν πλήρως όσα ζητήματα επίδοσης είχαν προκύψει από την πρώτη σχεδίαση, με την απλοποίηση – αφαίρεση περιττής διαίρεσης των modules.

Συμβατότητα με την STL

Η βιβλιοθήκη έπρεπε να είναι πλήρως συμβατή με το API (application interface) της C++ Standard Template Library (STL), ούτως ώστε να μπορεί να ενσωματωθεί σε υπάρχουσες εφαρμογές με μηδαμινές αλλαγές στον πηγαίο κώδικά τους.

4.2. Σύνθεση τύπων από τα λειτουργικά τμήματα – παραδείγματα

Σε αυτή την ενότητα παραθέτουμε διάφορα παραδείγματα σύνθεσης τύπων από τα λειτουργικά τμήματα (modules) της βιβλιοθήκης, έτσι ώστε να γίνει κατανοητό στον αναγνώστη πόσο εύκολο να γίνει κάτι τέτοιο με τη συγκεκριμένη σχεδίαση της βιβλιοθήκης.

Παράδειγμα 1 – Δυναμικός Πίνακας (Vector)

```
//Vector Definition
typedef DDTLibrary::Vector<int, Tracing::std_traits,
                      Tracing::var_traits<TRAITS(1)>> VECTOR;
```

Παράδειγμα 2 – Απλά Συνδεδεμένη Λίστα (Singly Linked List – SLL)

```
//Singly Linked List Definition
typedef DDTLibrary::sllListElement<int, Tracing::std_traits,
                      Tracing::var_traits<TRAITS(2)>> SLLELEMENT;
typedef DDTLibrary::listBase<SLLELEMENT> SLLISTBASE;
typedef DDTLibrary::sll<SLLISTBASE> SLLISTTYPE;
typedef DDTLibrary::List<SLLISTTYPE> SLL;
```

Παράδειγμα 3 – Απλά Συνδεδεμένη Λίστα με Roving Pointer (SLLRoving)

```
//Singly Linked List with Roving Pointer Definition
typedef DDTLibrary::sllListElement<int, Tracing::std_traits,
                      Tracing::var_traits<TRAITS(3)>> SLLELEMENT;
typedef DDTLibrary::listBase<SLLELEMENT> SLLISTBASE;
typedef DDTLibrary::sll<SLLISTBASE> SLLISTTYPE;
typedef DDTLibrary::List<DDTLibrary::sllRovingAdapter<SLLISTTYPE>>
                      SLLROVING;
```

Παράδειγμα 4 – Δυαδικό Δέντρο Αναζήτησης (BinaryTree)

```
//Binary Tree Definition
typedef DDTLibrary::BinaryTreeNode<int, Tracing::std_traits,
                      Tracing::var_traits<TRAITS(4)>> BINARYTREENODE;
typedef DDTLibrary::BinaryTree<BINARYTREENODE> BINARYTREEBASE;
typedef DDTLibrary::Tree<BINARYTREEBASE> BINARYTREE;
```

Παράδειγμα 5 – Red-Black Δυαδικό Δέντρο Αναζήτησης (RedBlackTree)

```
//Red-Black Tree Definition
typedef DDTLibrary::BinaryTreeNode<int, Tracing::std_traits,
                      Tracing::var_traits<TRAITS(5)>> BINARYTREENODE;
typedef DDTLibrary::RedBlackTreeNodeAdapter<BINARYTREENODE>
                      REDBLACKTREENODE;
typedef DDTLibrary::BinaryTree<REDBLACKTREENODE> REDBLACKTREE_BASE;
typedef DDTLibrary::RedBlackTreeAdapter<REDBLACKTREE_BASE>
                      REDBLACKTREEBASE;
typedef DDTLibrary::Tree<REDBLACKTREEBASE> REDBLACKTREE;
```

Παράδειγμα 6 – Λίστα από Πίνακες (GluedType)

```
//Glue Layer definitions

//InternalDDT Definition
typedef DDTLibrary::Vector<int, Tracing::std_traits,
                      Tracing::var_traits<TRAITS(6)>> INTERNALDDT;
//ExternalDDT Definition
typedef DDTLibrary::sllListElement<int, Tracing::std_traits,
                      Tracing::var_traits<TRAITS(6)>> SLLELEMENT;
typedef DDTLibrary::listBase<SLLELEMENT> SLLISTBASE;
typedef DDTLibrary::sll<SLLISTBASE> SLLISTTYPE;
typedef DDTLibrary::List<SLLISTTYPE> EXTERNALDDT;

typedef DDTLibrary::GlueLayer<EXTERNALDDT, INTERNALDDT> GLUEDTYPE;
```

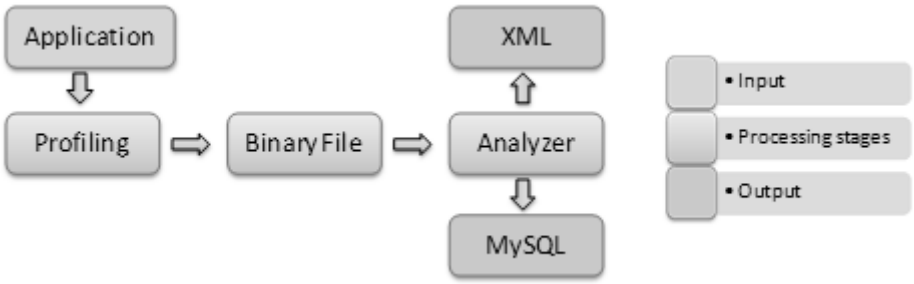
5. Η ανάγκη για custom profiling στο επίπεδο του interface

Για να καταλάβουμε αρχικά τη συμπεριφορά των δυναμικών τύπων δεδομένων, πρέπει να δώσουμε προσοχή στον τρόπο με τον οποίο αλληλεπιδρούν με τα ίδια τα δεδομένα. Γι' αυτό το σκοπό βασιστήκαμε στην profiling βιβλιοθήκη (profiler) που παρουσιάστηκε στο [Christophe] και αναπτύξαμε ένα νέο σύνολο εργαλείων καταγραφής και ανάλυσης για να αναλάβει τη ανάλυση της συμπεριφοράς των δυναμικών τύπων δεδομένων.

Συγκεκριμένα, μας ενδιαφέρουν οι παρακάτω συμπεριφορές:

- Δυναμικές προσπελάσεις μνήμης (εγγραφές – αναγνώσεις) που αναγνωρίζονται από τη συγκεκριμένη διεύθυνση της κάθε μεταβλητής στη μνήμη.
- Λειτουργίες πάνω στους δυναμικούς τύπων δεδομένων που αναγνωρίζονται από το συγκεκριμένο τύπο.
- Μονοπάτια της ροής ελέγχου που δεικνύουν τα σημεία στα οποία λαμβάνουν χώρα οι λειτουργίες.
- Πρότυπα προσπελάσεων των δεδομένων κάθε τύπου.
- Αναγνωριστικά νημάτων εκτέλεσης (thread identifiers) μέσα στα οποία λαμβάνουν χώρα οι λειτουργίες.

Στο παρακάτω σχήμα φαίνεται η συνολική ροή του συνόλου εργαλείων που αναπτύξαμε:



Σχήμα 9 - Ροή των profiler και analyzer εργαλείων

5.1. Παρουσίαση του Profiler της βιβλιοθήκης

Ο profiler που αναπτύξαμε είναι υπεύθυνος για την καταγραφή των λειτουργιών στα δυναμικά δεδομένα των δομών μας. Ενσωματώνεται στην εφαρμογή με διαφανή τρόπο και αυτομάτως καταγράφει όλες τις προσπελάσεις στα δυναμικά δεδομένα, όπως επίσης και πλούσιες πληροφορίες σχετικά με τις δομές που τα περιέχουν. Το εξαγόμενο αποτέλεσμα του profiler είναι ένα σύνολο ακατέργαστων πληροφοριών, από τα οποία θα εξαχθεί και η τελική ανάλυση της συμπεριφοράς, ύστερα από της χρήση του analyzer.

Το σύνολο των πληροφοριών που καταγράφεται φαίνεται αναλυτικά στον παρακάτω πίνακα:

At the memory level
• Logging of any read/write access to all data elements of C++ (build and user created types) • Certain scopes within a C++ code
• Logging of memory footprint • List of all used block sizes • Number of allocations/deallocations for each block size • Maximum number of concurrent instances for each block size in memory • Thread identifier for each access to data
• Total accesses for each DDT • Total memory footprint for each DDT • Number of max concurrent instances • Max Number of objects hosted in each DDT
At the interface level
• Complete logging of all sequence-DDT operations • Complete logging of iterator operations • Log sequential/random access to sequence-DDTs

Table 1 - Complete Profing Information

Από τον παραπάνω πίνακα φαίνεται ότι η καταγραφή πληροφοριών γίνεται σε διάφορα επίπεδα:

- Το επίπεδο της προσπέλασης δεδομένων (memory access level), όπου καταγράφονται όλες οι αναγνώσεις/εγγραφές
- Το επίπεδο διεπαφής (interface level), όπου καταγράφονται όλες οι κλήσεις στις συναρτήσεις της διεπαφής, όπως επίσης και η χρήση iterators.

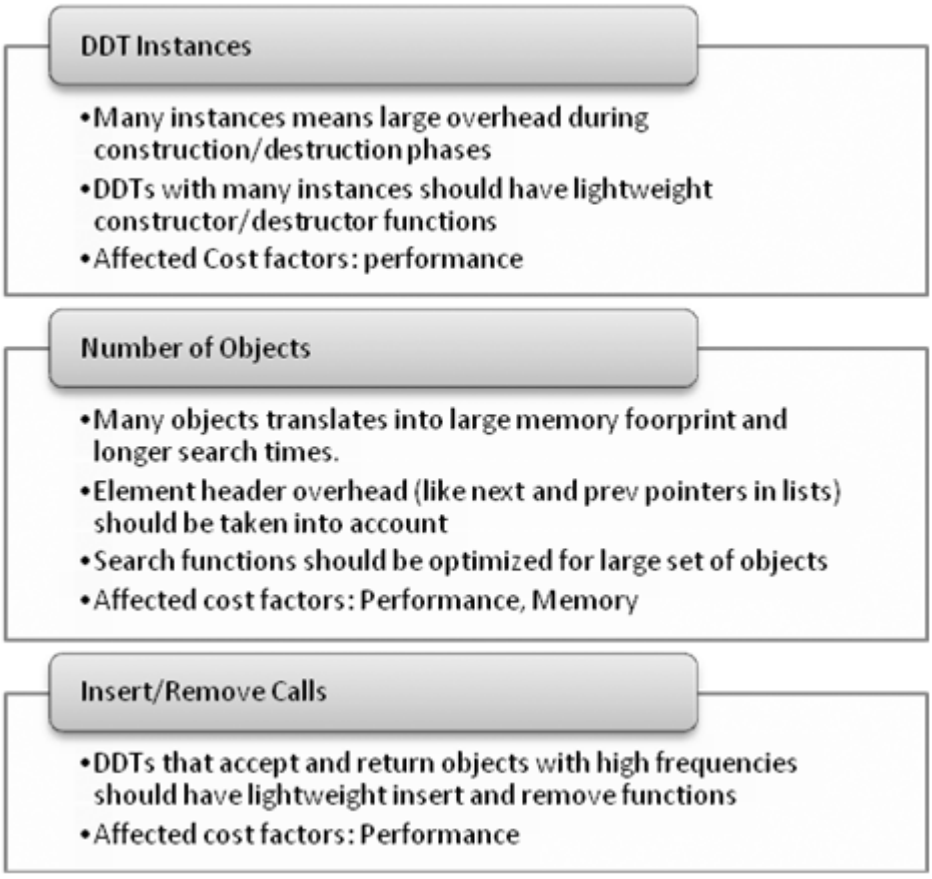
Επιπλέον γίνεται και καταγραφή πληροφοριών υψηλού επιπέδου, όπως ο μέγιστος αριθμός στοιχείων που βρίσκονται μέσα σε μία δομή δεδομένων.

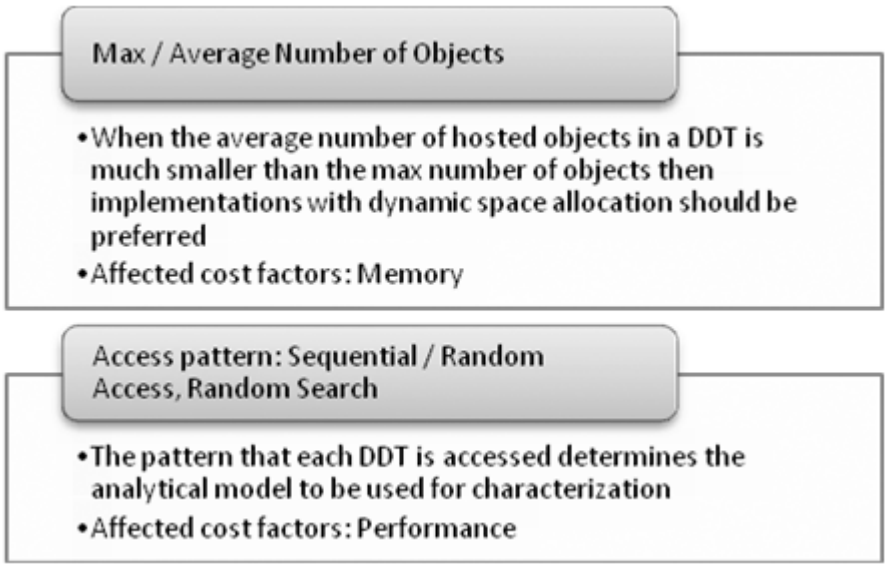
5.2. Παρουσίαση του Analyzer της βιβλιοθήκης

Το σύστημα καταγραφής μας είναι βασισμένο σε πακέτα: Για κάθε λειτουργία που συμβαίνει στους δυναμικούς τύπους δεδομένων, δημιουργείται ένα πακέτο που περιγράφει αυτή τη λειτουργία και εγγράφεται σε ένα αρχείο καταγραφής.

Κατά το στάδιο της ανάλυσης, όλα τα πακέτα διαβάζονται από το αρχείο και διαιρούνται σε κατηγορίες. Ύστερα, η πληροφορία που εξάγεται από κάθε πακέτο μετατρέπεται σε διάφορες εξαγώγιμες μορφές. Προς το παρόν, ο analyzer υποστηρίζει έξοδο σε XML (Extensible Markup Language) και σε Σχεσιακή Βάση Δεδομένων (συγκεκριμένα τη MySQL). Αυτές οι μορφές εξόδου επιλέχθηκαν λόγω της ευρείας αποδοχής τους. Έτσι, δίνεται η δυνατότητα να γραφούν νέα εργαλεία ανάλυσης που θα εξάγουν συγκεκριμένα τμήματα της πληροφορίας κάθε φόρα που αυτό είναι απαραίτητο.

Στο παρακάτω σχήμα παρουσιάζεται συγκεκριμένα η υψηλού επιπέδου πληροφορία που απαιτείται για συμπεράνει κανείς τη συμπεριφορά μίας δυναμικής δομής δεδομένων. Αυτή η πληροφορία μπορεί να εξαχθεί στο σύνολό της, είτε απευθείας από τον analyzer, είτε χρήσει SQL ερωτημάτων:





Σχήμα 10 - Απαραίτητη Πληροφορία για την Ανάλυση της Συμπεριφοράς των Τύπων

6. Ενσωμάτωση των εργαλείων σε μια πραγματική εφαρμογή

6.1. Διαδικασία ενσωμάτωσης - Περιορισμοί

Η ενσωμάτωση του συνόλου εργαλείων που αναπτύξαμε σε μία πραγματική εφαρμογή, ανάγεται ουσιαστικά στην ενσωμάτωση της βιβλιοθήκης δομών δεδομένων, καθώς το η όλη διαδικασία της καταγραφής είναι ενσωματωμένη στη βιβλιοθήκη μας.

Για να πραγματοποιηθεί, λοιπόν, ενσωμάτωση της βιβλιοθήκης, αρκεί να αντικατασταθούν οι δηλώσεις των δομών δεδομένων της εφαρμογής με τις αντίστοιχες δηλώσεις των δικών μας δομών.

Αυτό ισχύει βέβαια στην περίπτωση που εφαρμογή χρησιμοποιεί δομές δεδομένων με interface συμβατό με την STL, πράγμα όμως συμβαίνει στη συντριπτική πλειοψηφία των εφαρμογών γραμμένων σε C++.

6.2. Παράδειγμα από την ενσωμάτωση στο Vdrift

Σε αυτή την ενότητα παρατίθεται ένα παράδειγμα ενσωμάτωσης βιβλιοθήκης μας στο Vdrift (περισσότερες πληροφορίες για το Vdrift στην επόμενη ενότητα).

DDTIncludes.h:

```
template <typename T, int I = 1>
struct ddt_helper {
      //List definitions
      typedef DDTLibrary::sllListElement<T, Tracing::std_traits,
            Tracing::var_traits<TRAITS(I)>> SLLELEMENT;
      typedef DDTLibrary::dllListElement<T, Tracing::std_traits,
            Tracing::var_traits<TRAITS(I)>> DLLELEMENT;
      typedef DDTLibrary::listBase<SLLELEMENT> SLLLISTBASE;
      typedef DDTLibrary::listBase<DLLELEMENT> DLLLISTBASE;
      typedef DDTLibrary::sll<SLLLISTBASE> SLLLISTTYPE;
      typedef DDTLibrary::dll<DLLLISTBASE> DLLLISTTYPE;

      typedef DDTLibrary::List<SLLLISTTYPE> SLLLIST;
      typedef DDTLibrary::List<DLLLISTTYPE> DLLLIST;
```

```
	typedef DDTLibrary::List<DDTLibrary::sllRovingAdapter<SLLLISTTYPE>>
		SLLROVINGLIST;
	typedef DDTLibrary::List<DDTLibrary::dllRovingAdapter<DLLLISTTYPE>>
		DLLROVINGLIST;

	//Vector definitions
	typedef DDTLibrary::Vector<T, Tracing::std_traits,
		Tracing::var_traits<TRAITS(I)>> VECTOR;

	//Tree definitions
	typedef DDTLibrary::BinaryTreeNode<T, Tracing::std_traits,
		Tracing::var_traits<TRAITS(I)>> BINARYTREENODE;
	typedef DDTLibrary::RedBlackTreeNodeAdapter<BINARYTREENODE>
		REDBLACKTREENODE;
	typedef DDTLibrary::BinaryTree<BINARYTREENODE> BINARYTREEBASE;
	typedef DDTLibrary::BinaryTree<REDBLACKTREENODE> REDBLACKTREE_BASE;
	typedef DDTLibrary::RedBlackTreeAdapter<REDBLACKTREE_BASE>
		REDBLACKTREEBASE;

	typedef DDTLibrary::Tree<BINARYTREEBASE> BINARYTREE;
	typedef DDTLibrary::Tree<REDBLACKTREEBASE> REDBLACKTREE;

	//Glue Layer definitions
	typedef DLLLIST INTERNALDDT;

	typedef DDTLibrary::sllListElement<INTERNALDDT, Tracing::std_traits,
		Tracing::var_traits<TRAITS(I)>> EXTERNALDDTELEMENT;
	typedef DDTLibrary::listBase<EXTERNALDDTELEMENT> EXTERNALDDTBASE;
	typedef DDTLibrary::sll<EXTERNALDDTBASE> EXTERNALDDTTYPE;

	typedef DDTLibrary::List<EXTERNALDDTTYPE> EXTERNALDDT;
	typedef DDTLibrary::GlueLayer<EXTERNALDDT, INTERNALDDT> GLUEDTYPE;

	//Types
	typedef GLUEDTYPE type;


};
```

World.h:

```
//…
#include "DDTIncludes.h"

//…

typedef ddt_helper<Times, 28>::type VTimes;

//…
```

7. Χρήση των εργαλείων για την βελτιστοποίηση μια πραγματικής εφαρμογής

7.1. Vdrift

Το VDrift είναι μία εφαρμογή που ανήκει στο χώρο τρισδιάστατων εξομοιωτών οδήγησης. Επιλέχθηκε κυρίως λόγω της έντονης πολυπλοκότητάς του στη χρήση δυναμικών δομών δεδομένων αλλά και επειδή κάνει εκτενή χρήση της STL βιβλιοθήκης, γεγονός που την καθιστά ιδανική για την ενσωμάτωσής της με τα εργαλεία που αναπτύξαμε.

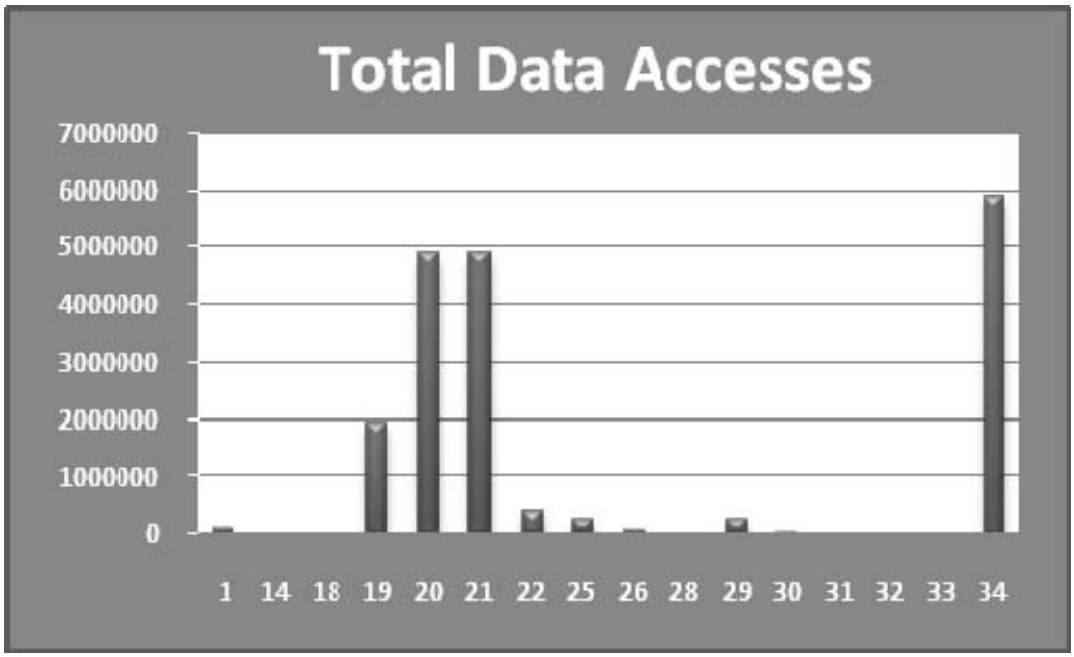
Η εφαρμογή χρησιμοποιεί πολύ ρεαλιστική εξομοίωση της συμπεριφοράς του αυτοκινήτου και περιλαμβάνει ένα τρισδιάστατο περιβάλλον για καλύτερη αλληλεπίδραση με το χρήστη.

Το Vdrift χρησιμοποιεί συνολικά 37 DDTs για να αποθηκεύσει τα δεδομένα του και είναι όλα vectors. Τα αντικείμενα που τοποθετούνται μέσα στις δομές ποικίλουν από 'wheel objects' μέχρι αριθμοί κινητής υποδιαστολής (float numbers) που απαιτούνται για τη φυσική του παιχνιδιού. Κατά την εκτέλεση του παιχνιδιού, άλλες δομές προσπελάζονται σειριακά, ενώ άλλες τυχαία. Έτσι απαιτείται από τις ίδιες τις δομές να αντιμετωπίσουν αυτά τα πρότυπα προσπέλασης.

Αυτό ακριβώς το γεγονός μας δίνει τη δυνατότητα να πραγματοποιήσουμε σημαντικές βελτιώσεις, όπως θα δούμε παρακάτω.

7.2. Ανάλυση της Συμπεριφοράς των Δυναμικών Τύπων Δεδομένων

Το πρώτο βήμα για την ανάλυση της συμπεριφοράς της συγκεκριμένης εφαρμογής είναι να εντοπίσουμε τα ενεργά και με μεγάλη κινητικότητα (αυτά που παρουσιάζουν πολυάριθμες προσπελάσεις) DDTs. Γι' αυτό το λόγο εξάγουμε το παρακάτω γράφημα χρήσει των εργαλείων μας.

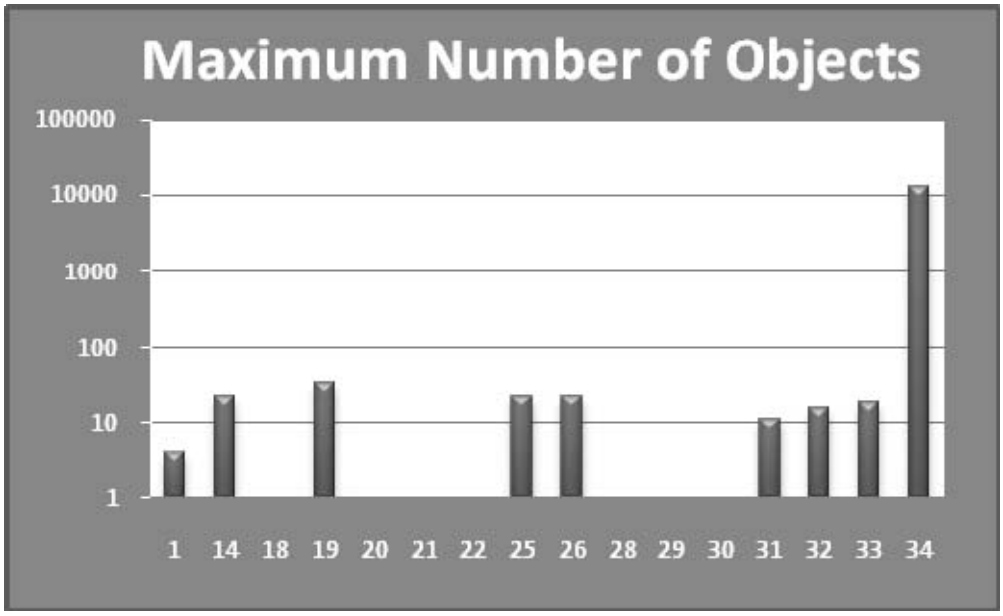


Σχήμα 11 - Συνολικός Αριθμός Προσπελάσεων για κάθε DDT

Το παραπάνω σχήμα δείχνει ότι έχουμε 16 ενεργά DDTs από το σύνολο των 37. Ο αποκλεισμός των ανενεργών είναι το πρώτο βήμα στην ανάλυση της συμπεριφοράς της εφαρμογής, καθώς αποφορτίζουμε το σχεδιαστή από περιττό κόπο.

Μία επίσης σημαντική πληροφορία που πρέπει να γνωρίζουμε είναι ο πλήθος των φορών που δημιουργείται ένα αντικείμενο. Αυτό συμβαίνει συνήθως κατά το copy construction και οι σχεδιαστές πρέπει να διαχειρίζονται τα ανεπιθύμητα ενδιάμεσα στιγμιότυπα πχ κατά το πέρασμα παραμέτρων στις συναρτήσεις της C++.

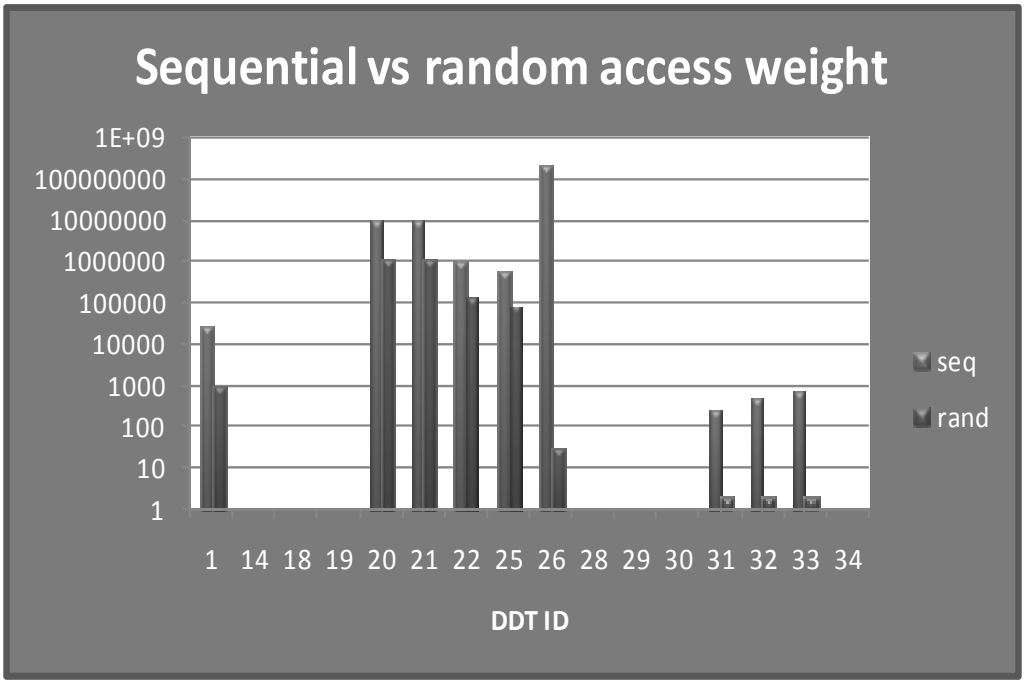
Όπως φαίνεται παρακάτω, τα περισσότερα DDT δημιουργούν ένα ή πολύ λίγα στιγμιότυπα. Το DDT34 όμως αποτελεί εξαίρεση, καθώς δημιουργείται κατά προσέγγιση 25000 φορές. Το DDT34 είναι ένα μέρος ενός M-way tree που αντιπροσωπεύει τα σημεία σύγκρουσης μέσα στην πίστα του παιχνιδιού. Κάθε κόμβος είναι μία λίστα (DDT34) από άλλος κόμβους, γι' αυτό έχουμε αυτόν τον τεράστιο αριθμό στιγμιότυπων.



Σχήμα 12 - Μέγιστος Αριθμός Στοιχείων για κάθε DDT

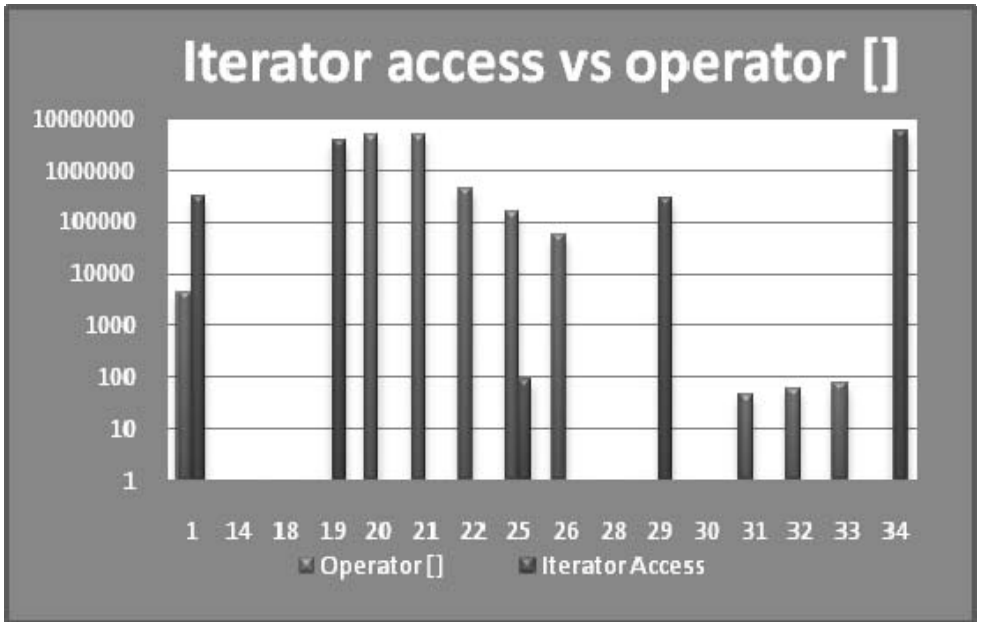
Όσα DDTs δημιουργούν πολλά στιγμιότυπα, πρέπει να έχουν ελαφρύ constructor, όπως οι λίστες. Αντίθετα, τα vectors έχουν βαρύ constructor επειδή προδεσμεύουν κάποιο χώρο που θα χρειαστούν.

Επιπλέον, ο μέγιστος αριθμός στοιχείων που αποθηκεύονται στο DDT μεταφράζεται σε μεγάλες απαιτήσεις σε μνήμη και μεγαλύτερους χρόνους αναζήτησης. Το τελευταίο πρέπει να συνδυαστεί με πρότυπο προσπέλασης (σειριακό/τυχαίο). Τυχαίος τρόπος προσπέλασης σε συνδυασμό με μεγάλο πλήθος αντικειμένων είναι ο ορισμός της χρήσης του πίνακα.



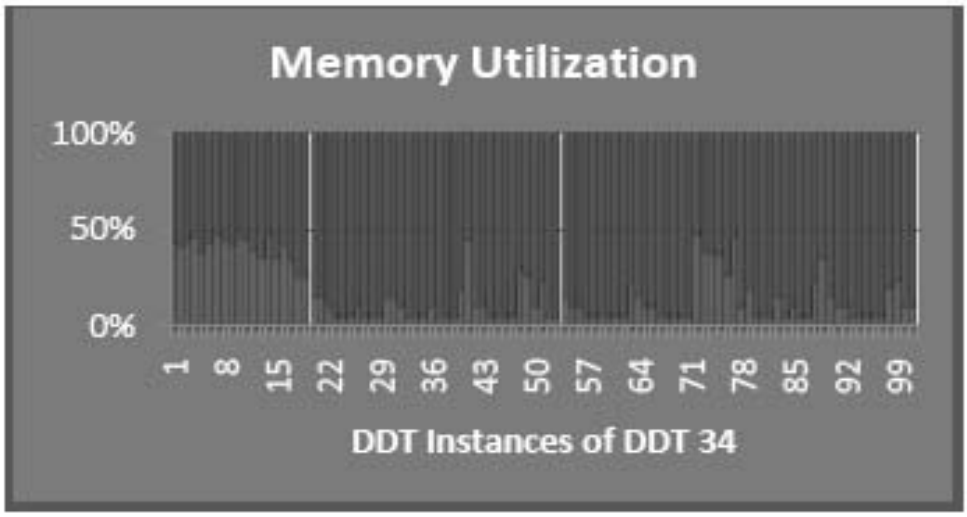
Σχήμα 13 - Sequential VS Random Access

Παρόλο που το μέγιστο πλήθος στοιχείων αφορά κυρίως το στιγμιότυπο ενός DDT, πολλές φορές πρέπει να δίνουμε προσοχή και στο πλήθος των στιγμιότυπων που δημιουργούνται από ένα DDT. Όπως φαίνεται από το Σχήμα 12, τα DDT14, DDT19, DDT34 απαιτούν ιδιαίτερη προσοχή λόγω του μεγάλου μέγιστου πλήθους αντικειμένων που δημιουργούν. Συγκεκριμένα για το DDT34, αν προσπελαζόταν τυχαία, θα έπρεπε να είναι πίνακας. Το παρακάτω σχήμα όμως δείχνει την αποκλειστική χρήση iterators που συνεπάγονται σειριακή προσπέλαση. Έτσι, μία συνδεδεμένη δομή θα είχε εξίσου αποδοτικά αποτελέσματα, περιορίζοντας όμως τη χρήση μνήμης στο ελάχιστο δυνατό.



Σχήμα 14 - Iterator VS Operator[] Access

Ένα τελευταίο σημείο που πρέπει να τονίσουμε σχετικά με τη συμπεριφορά των δυναμικών δομών δεδομένων είναι ότι σπανίως κάνουν πλήρη χρήση της μνήμης που δεσμεύουν. Ως γνωστόν, δυναμικοί πίνακες αυξάνονται διπλασιάζοντας το χώρο που ήδη κατέχουν.



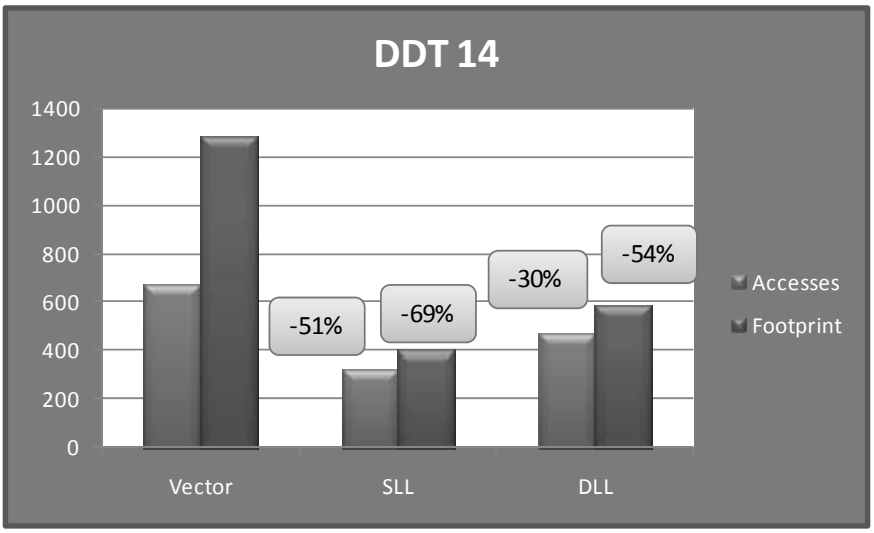
Σχήμα 15 - Αξιοποίηση της Δεσμευμένης Μνήμης για το DDT34

Στο παραπάνω γράφημα φαίνεται οι αξιοποίηση της μνήμης για το DDT34 με τη vector υλοποίηση. Όταν οι μπλε γραμμές (σκούρο γκρι) ισαπέχουν από το 50% με τις κόκκινες (ανοιχτό γκρι) τότε έχουμε τέλεια αξιοποίηση της μνήμης, διαφορετικά παραμένει σημαντικό ποσοστό αναξιοποίητο.

7.3. Επιλογή του κατάλληλου DDT και αποτελέσματα

Σε αυτή την ενότητα παρουσιάζουμε ενδεικτικά τη διαδικασίας ορισμένων DDT. Γίνεται σαφές ότι μπορούμε να βελτιώσουμε τις επιδόσεις ενός DDT σχετικά με τις προσπελάσεις στη μνήμη και δέσμευση μνήμης (memory footprint) επιλέγοντας τη σωστή υλοποίηση κάθε φορά.

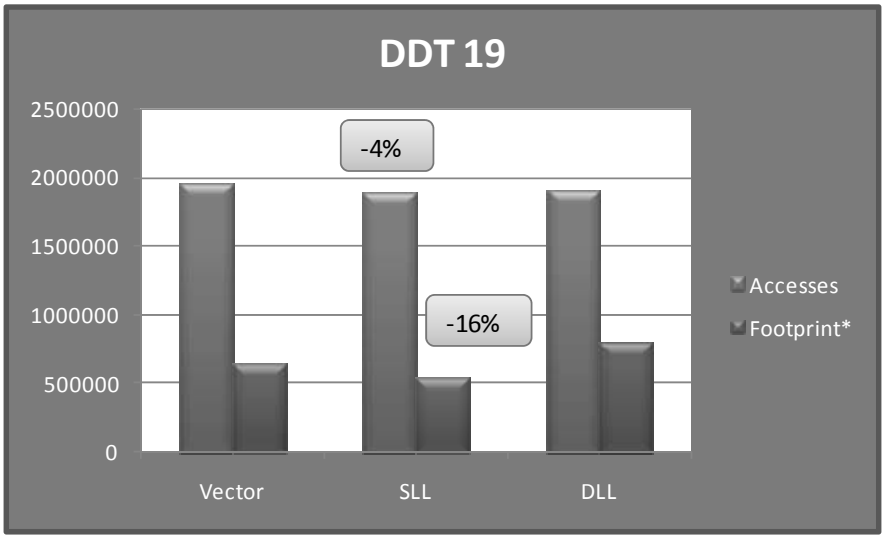
I. DDT14



Σχήμα 16 - Βελτιώνοντας το DDT14

Το DDT14 έχει μονάχα ένα στιγμιότυπο ενεργό κάθε φορά (Σχήμα 11). Επιπλέον περιλαμβάνει μεγάλο αριθμό αντικειμένων (Σχήμα 12), οπότε σε περίπτωση που τα αντικείμενα προσπελάζονται με τυχαίο τρόπο, η χρήση λίστας θα οδηγήσει σε χειρότερα αποτελέσματα. Ωστόσο, όπως επιβεβαιώνεται από το Σχήμα 14, τα στοιχεία δεν προσπελάζονται απευθείας, οπότε δεν υπάρχει επιπλέον κόστος από τη χρήση λίστας. Το DDT14 χρησιμοποιείται ως χώρος αποθήκευσης και περνιέται ως value παράμετρος (copy construction) στα άλλα DDTs. Όπως φαίνεται από το παραπάνω σχήμα, επιτυγχάνουμε σημαντικότατες βελτιώσεις και στον αριθμό τον προσπελάσεων και στη δέσμευση της μνήμης.

II. DDT19

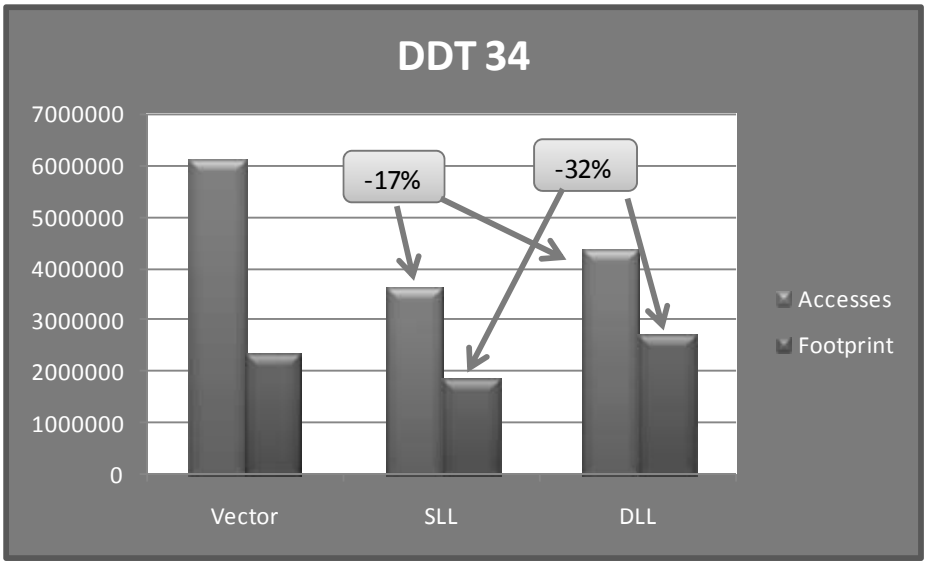


Σχήμα 17 - Βελτιώνοντας το DDT19

Αυτή η συγκεκριμένη δομή δεδομένων έχει επίσης ένα μόνο ενεργό στιγμιότυπο κάθε φορά (Σχήμα 11). Καθώς περιέχει μεγάλο αριθμό αντικειμένων (Σχήμα 12) ελέγχουμε αν προσπελάζεται σειριακά ή τυχαία. Από το Σχήμα 14 φαίνεται ότι μόνο iterators την προσπελάζουν οπότε η χρήση απλά συνδεδεμένης λίστας είναι ιδανική, επειδή αποδίδει καλύτερα ακόμα και όσον αφορά τα accesses σε σχέση με τον αρχικό τύπο (αυτό δικαιολογείται λόγω των διαδοχικών resizes του vector στην προσπάθεια να χωρέσει τον αυξανόμενο αριθμό αντικειμένων)

Είναι σημαντικό να σημειώσουμε σε αυτό το σημείο ότι η λίστα που έχει υλοποιηθεί στην STL είναι διπλά συνδεδεμένη (DLL). Έτσι, ακόμα και αν χρησιμοποιούσαμε αυτή στη θέση του vector, θα λαμβάναμε λιγότερα accesses, αλλά χρησιμοποιούσαμε περισσότερη μνήμη (λόγω του επιπλέον δείκτη στο προηγούμενο στοιχείο).

III. DDT34

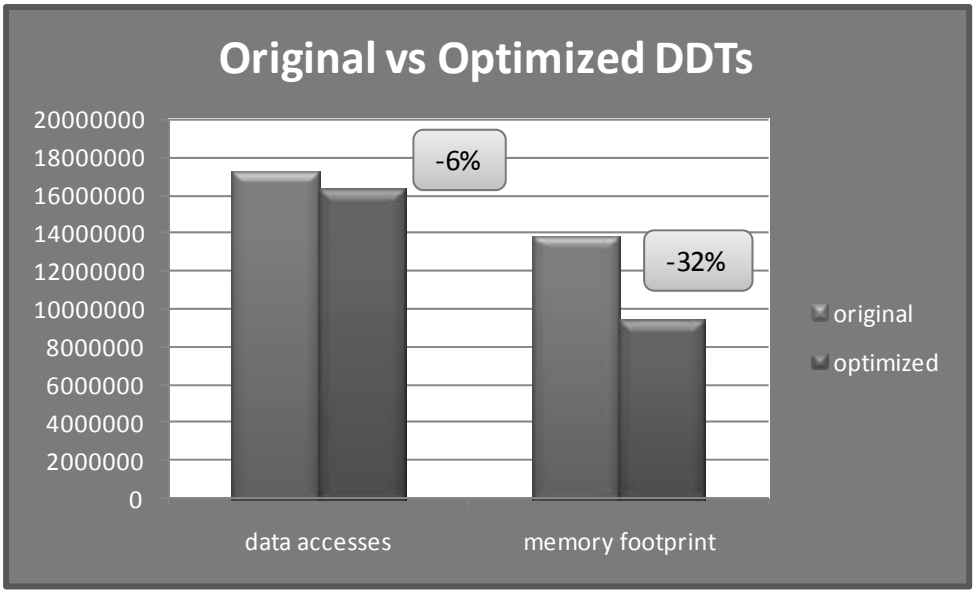


Σχήμα 18 - Βελτιώνοντας το DDT34

Το DDT34 αποτελεί μία ειδική περίπτωση δομής, λόγω του γεγονότος ότι είναι δέντρο στο οποίο κάθε παιδί είναι μία λίστα λιστών. Η αρχική υλοποίηση είναι με διπλά συνδεδεμένη λίστα. Ωστόσο, στο Σχήμα 14 φαίνεται ότι μονάχα iterators προσπελάζουν το DDT34. Το πλεονέκτημα που προσφέρουν οι DLLs (διπλής κατεύθυνσης διάσχιση) δεν αξιοποιείται εδώ. Αντιθέτως οι DLLs απαιτούν επιπλέον χώρο στην μνήμη για αποθήκευση του δείκτη στο προηγούμενο στοιχείο, κάθε φορά που δημιουργείται ένα νέο στοιχείο. Έτσι και εδώ μια SLL είναι προτιμότερη.

IV. Βελτιστοποιώντας ολόκληρη την εφαρμογή

Παρουσιάσαμε διάφορα παραδείγματα για το πώς μπορούμε να αξιοποιήσουμε την πληροφορία που εξάγεται από το σύνολο εργαλείων μας για να βελτιώσουμε ορισμένα DDTs της εφαρμογής. Εδώ, παραθέτουμε το συνολικό κέρδος όσον αφορά τις προσπελάσεις στη μνήμη (data accesses) και τη δέσμευση μνήμης (memory footprint).



Σχήμα 19 - Συνολική Βελτιστοποίηση της Εφαρμογής

Το παραπάνω σχήμα δείχνει ότι μπορούμε να επιτύχουμε συνολικά 6% μείωση στα data accesses, το οποίο είναι αξιοσημείωτο αν λάβουμε υπόψη μας ότι τα data accesses μεταφράζονται σε memory accesses (ανάλογα με την εκάστοτε αρχιτεκτονική του συστήματος). Το σχήμα δείχνει επίσης ότι μπορούμε να κερδίσουμε ένα σημαντικότατο ποσοστό μνήμης που απαιτείται για να τρέξει η εφαρμογή, πράγμα που της επιτρέπει να τρέξει σε μεγαλύτερο εύρος συστημάτων.

8. Συμπεράσματα – Προοπτικές

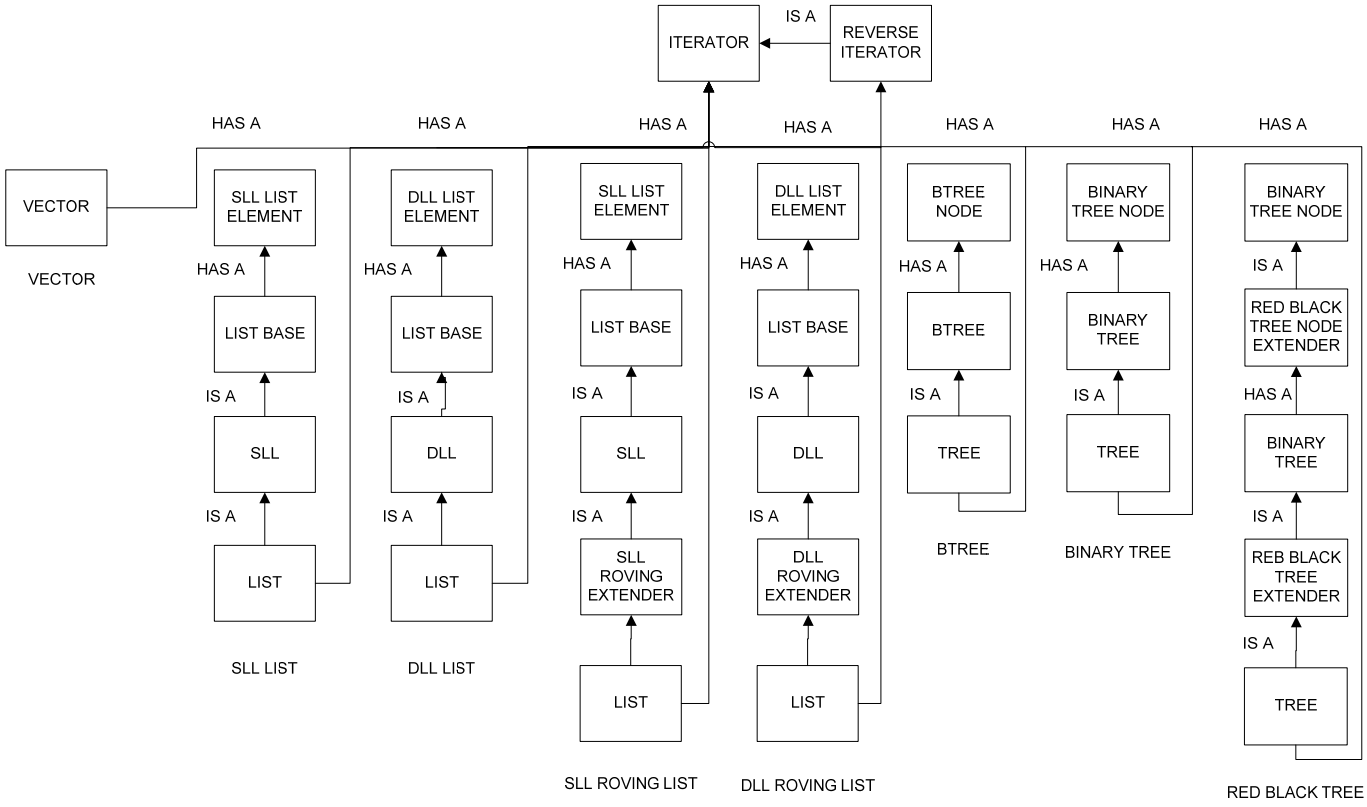
Στη συγκεκριμένη διπλωματική εργασία, παρουσιάστηκαν τα εργαλεία που αναπτύχθηκαν για τη βελτιστοποίηση των δυναμικών προσπελάσεων στα δεδομένα των εφαρμογών μας.

Ο κυριότερος στόχος που επιτεύχθηκε στη συγκεκριμένη διπλωματική εργασία είναι η προσφορά λύσης στο θέμα της εύρεσης εκείνης της δυναμικής δομής δεδομένων μιας εφαρμογής, η οποία θα παρέχει τη βέλτιστη συμπεριφορά, όσον αφορά τις δυναμικές προσπελάσεις των δεδομένων της εφαρμογής. Η χρήση της βιβλιοθήκης καθώς και των υπολοίπων εργαλείων που παρουσιάστηκε στο έβδομο κεφάλαιο, παρέχει μια έτοιμη λύση στο σχεδιαστή για δοκιμή όλων των πιθανών δυναμικών δομών δεδομένων και την εύρεση του βέλτιστου ανάμεσά τους, σύμφωνα με τα εκάστοτε κριτήρια. Η συγκεκριμένη μεθοδολογία μπορεί να εφαρμοστεί σε οποιαδήποτε εφαρμογή ανεξάρτητα από την πολυπλοκότητά της, πράγμα που προσφέρει ένα μεγάλο πλεονέκτημα έναντι των άλλων διαθέσιμων λύσεων.

Το επόμενό μας βήμα (που έχει ήδη ξεκινήσει) είναι δημιουργία ενός συνόλου εργαλείων που θα χρησιμοποιεί ευρετικές τεχνικές για να λάβει με αυτοματοποιημένο τρόπο αποφάσεις βασισμένο στο αποτέλεσμα της καταγραφής. Αυτό το σύνολο εργαλείων θα είναι σε θέση να αξιολογεί ένα μεγάλο εύρος διαφορετικών υλοποιήσεων ως πιθανές υποψηφίους για βελτιστοποίηση των αρχικών δυναμικών δομών δεδομένων. Απώτερος στόχος είναι να μπορεί να προσφέρει λύσης υψηλής παραμετροποίησης, οι οποίες θα εκμεταλλεύονται πλήρως τη συμπεριφορά της εκάστοτε εφαρμογής.

Παράρτημα στα Αγγλικά

9. Description of the DDT Library



Σχήμα 20 - DDT Library Class Hierarchy

The DDT Library is our Library of data structures that was designed and implemented with the following guidelines:

- Modularity
- Efficiency
- STL Compatibility

Modularity

The implementation DDT Library was divided into several modules and layers that added functionality to the target data structure. For each Abstract Data Type (ADT) we separated the notions of data structure element and data structure itself.

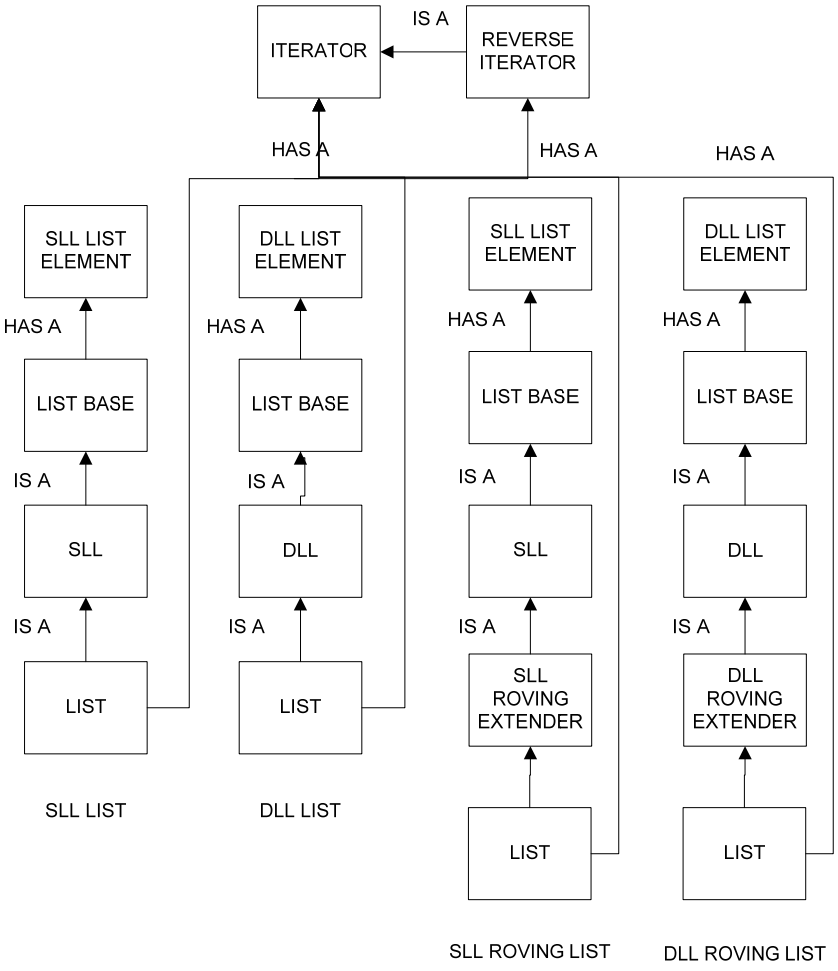
Efficiency

The DDT Library was designed and implemented twice:
In version 1, there were some significant efficiency issues at some functions.
In version 2, after the complete re-design of some modules and especially the Iterator module, those efficiency issues were surpassed and we can say now that our Library is as efficient as possible (see Results).

STL Compatibility

The DDT Library had to be fully compatible with the API (application interface) of C++ Standard Template Library (STL). In that way, it could easily be integrated to applications that supported the STL, by simply changing the definition of the type used.

9.1. The List DDT



Σχήμα 21 - The List DDT Class Hierarchy

The List data structure consists of the following modules:

1. Element Level Modules

The Element Modules represent the nodes of the List

```
template<class T, class objectTraits = Tracing::std_traits, class
          ddtTraits = Tracing::std_traits> class sllListElement;
```

The Element Modules take 3 template arguments:

- a class T, which is the type to be wrapped the list element
- a class objectTraits, which is the type of Traits to be used for the variables that are defined not to be logged

- a class objectTraits, which is the type of Traits to be used for the variables that are defined to be logged

```
typedef typename ddtTraits::Logger Logger;
typedef typename ddtTraits::Allocator Allocator;
typedef typename objectTraits::template transform<T>::value_type
value_type;

typedef typename ddtTraits::transform<sllListElement*>::value_type
                element_pointer;
typedef typename ddtTraits::transform<sllListElement&>::value_type
                element_reference;
typedef typename objectTraits::template
        transform<sllListElement*>::value_type element_pointer_unlogged;

typedef ddtTraits Traits;
```

At this level we define the types that will be used throughout the chain of inheritance of the Library:

Original Class	Class synonym	Description
ddtTraits::Logger	Logger	The Logger to be used for creating the log packets.
ddtTraits::Allocator	Allocator	The Allocator to be used for overriding the new and delete operators.
objectTraits::template transform<T>::value_type	value_type	Type of data hold by the data structure. Defaults to a synonym of T and accesses are not being logged (if objectTraits = std_traits).
ddtTraits::transform<sllListElement*>::value_type	element_pointer	Pointer to an element. Accesses to this pointer are being logged automatically (if ddtTraits != std_traits).
ddtTraits::transform<sllListElement&>::value_type	element_reference	Reference to an element. Accesses to this reference are being logged automatically (if ddtTraits != std_traits).
objectTraits::transform<sllListElement*>::value_type	element_pointer_unlogged	Pointer to an element. Accesses to this pointer are not being logged by default (if objectTraits = std_traits).
ddTraits	Traits	Exposure of the logger Traits to whole Library.

Table 2 - SLL Element Types

The element modules also implement the following interface:

Function Name	Description
`makeLink(element_pointer_unlogged prev, element_pointer_unlogged next)`	Links the element with the a previous and a next element.
`removeLink(element_pointer_unlogged prev, element_pointer_unlogged next)`	Unlinks the element from the previous and the next element.
`getData()`	Returns a reference to the data hold by the element.
`getDataPointer()`	Returns a pointer to the data hold by the element.

Table 3 - SLL Element Interface

Currently, our Library supports 2 kinds of list nodes.

a. The sllListElement Module

The sllListElement class belongs to the element level modules and represents the element of a single linked list. So, it consists of two members:

- A pointer to the contained data.
- A (logged) pointer to the next element.

b. The dllListElement Module

The dllListElement class belongs to the element level modules and represents the element of a doubly linked list. So, it consists of three members:

- A pointer to the contained data.
- A (logged) pointer to the next element.
- A (logged) pointer to the previous element.

2. List Base Module

This Module consists of the main implementation of the common functionality of all List types (singly linked list, doubly linked list, etc.).

```
template<class elementType> class listBase;
```

The List Base module takes one template argument which is the type of the element to be used in the list implementation.
In our implementation, the List is guarded by sentinel element, which has the role of the "ground" that we design when describing a List.

It consists of two integers:

- The size (number of elements) of the List.
- The (unique) instance_id of the List.

And three pointers:

- A pointer to the head element of the List.
- A pointer to tail element of the List.
- A pointer to the sentinel element.

The List Base Module also implements the following interface:

Function Name	Description
`getFirstElement()`	Returns a pointer to the head element of the list.
`getLastElement()`	Returns a pointer to the tail element of the list.
`getNextElement(element_pointer_unlogged data)`	Returns a pointer to next element of the input element.
`push_back(const_input_reference value)`	Inserts a new element at the end of the list with a specific value.
`erase(element_pointer_unlogged prev, element_pointer_unlogged curr, element_pointer_unlogged next)`	Unlinks and destroys an element from the list.
`empty()`	True list empty, false ortherwise.
`getSize()`	Returns the number of elements contained by the list.
`IsNil(element_pointer_unlogged x)`	True if the pointer points to the sentinel element, false otherwise.
`resize(size_type n)`	Sets a new number of elements to the list. Pushes back new or removes some if needed.
`increment(element_pointer_unlogged data)`	Returns a pointer to the next element of the input element. Exclusively used by the iterator class.
`getData(element_pointer_unlogged data)`	Returns a reference to the data hold by a specific element pointer.
`getEnd()`	Returns an pointer to the end (sentinel) of the list. (For Iterator use).
`getREnd()`	Returns an pointer to the end (sentinel) of the list. (For reverse Iterator use – needed due to compatibility issues with the vector class).

Table 4 - List Base Interface

3. List Type Implementation Level Modules:

Each module of this layer consists of the specific implementation of the functionality of the List types (singly linked list, doubly linked list, etc.).

```
template<class listBase> class sll : public listBase;
```

A template argument is required, which is the type of list base module implementing the common functionality of all list types to be extended.
At this level, the following interface is implemented:

Function Name	Description
`getPreviousElement(element_pointer_unlogged curr)`	Returns a pointer to the previous element of the input element.
`decrement(element_pointer_unlogged data)`	Returns a pointer to the previous element of the input element. Exclusively used by the iterator class.
`get(size_type n)`	Returns the n-th element contained by the list.

Table 5 - SLL Interface

a. The sll Module

This module implements the Singly Linked List.

b. The dll Module

This module implements the Doubly Linked List.

4. List Type Implementation Extension Level Modules

Each module of this layer represents different optimizations on the implementation of module that it extends.

```
template<class listType> class sllRovingAdapter: public listType
```

A template argument is required, which is the type of list module to be extended.
Currently, only extension modules that implement the roving pointer (points to the most recently used element) optimization are available:

a. The sllRovingExtension Module

This module implements the roving pointer optimization for the Singly Linked List.

b. The dllRovingExtension Module

This module implements the roving pointer optimization for the Doubly Linked List.

At this level, the following functions are optimized:

Function Name	Description
`getPreviousElement(element_pointer_unlogged curr)`	Returns a pointer to the the previous element of the input element.
`decrement(element_pointer_unlogged data)`	Returns a pointer to the previous element of the input element. Exclusively used by the iterator class.
`get(size_type n)`	Returns the n-th element contained by the list.

Table 6 - SLL Roving Interface

5. The List Module

The List Module is the exposes the complete List functionality through an STL compatible interface

```
template <class listType> class List: public listType
```

It takes one template argument, which is the type of the list module containing the implementation of all the List's functions. This module actually acts as adapter to the input list module's interface, providing STL compliance including Iterator traversal. For this purpose, it defines iterators (normal, reverse and const) both logged and unlogged.

```
typedef Iterator_Base<List> iterator;
typedef ConstIteratorAdapter<iterator> const_iterator;
typedef ReverseIteratorAdapter<iterator> reverse_iterator;
typedef ReverseIteratorAdapter<const_iterator> const_reverse_iterator;

typedef Iterator_Base_Unlogged<List> iterator_unlogged;
typedef ReverseIteratorAdapter<iterator_unlogged>
        reverse_iterator_unlogged;
```

Logged iterators are intended for the default Library use. Unlogged iterators are useful for combining the List DDT with other DDTs (i.e. GlueLayer).

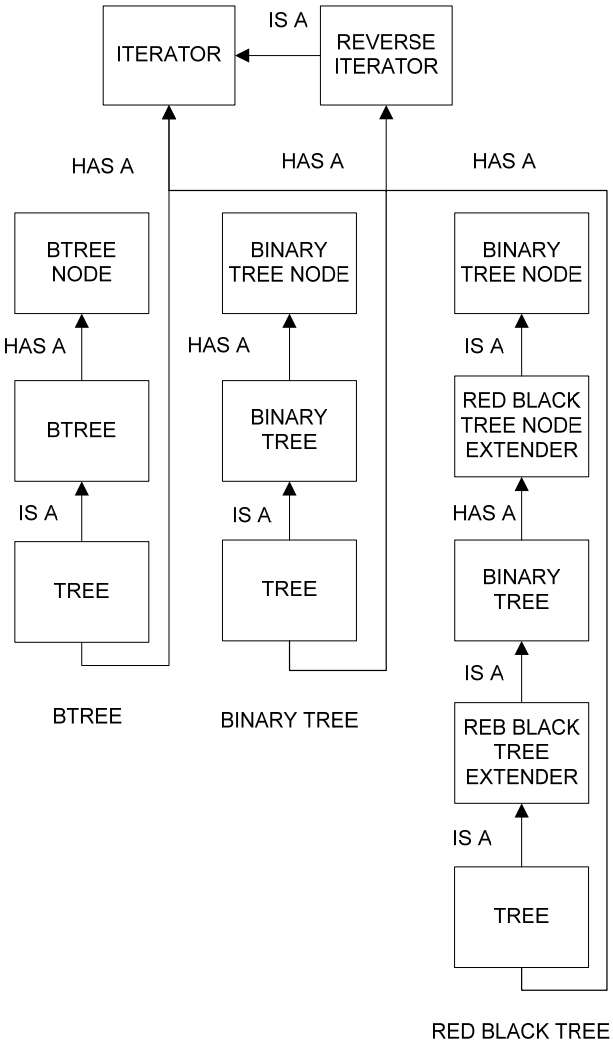
The exposed interface is the following:

Function Name	Description
`begin()`	Returns an Iterator pointing to the first element of the list.
`end()`	Returns an Iterator pointing to the first element of the list.
`rbegin()`	Returns a reverse Iterator pointing to the first element of the list.
`rend()`	Returns a reverse Iterator pointing to the first element of the list.
`empty()`	True list empty, false ortherwise.
`size()`	Returns the number of elements contained by the list.
`resize(size_type new_size, const_input_reference value)`	Sets a new number of elements to the list. Pushes back new (initialized to the

	input value) or removes some if needed.
`reserve(size_type new_capacity)`	Reserves memory space to the list.
`operator[](size_type n)`	Returns the n-th element contained by the list.
`get(size_type n)`	Returns the n-th element contained by the list.
`front()`	Returns a reference to data hold by the first element of the list.
`back()`	Returns a reference to data hold by the last element of the list.
`erase(iterator& pos)`	Erases the element at current Iterator position.
`push_back(const_input_reference value)`	Inserts a new element at the end of the list with a specific value.
`pop_back()`	Removes a the element at the end of the list.
`pop_front()`	Removes a the element at the front of the list.

Table 7 - List Interface

9.2. The Tree DDT



Σχήμα 22 - The Tree DDT

The Tree data structure consists of the following modules:

1. Element Level Modules

The Element Modules represent the basic node types of the Tree.

```
template<class T, class objectTraits = Tracing::std_traits, class
          ddtTraits = Tracing::std_traits> class BinaryTreeNode;
```

The Element Modules take 3 template arguments:

- a class T, which is the type to be wrapped the list element

- a class objectTraits, which is the type of Traits to be used for the variables that are defined not to be logged
- a class objectTraits, which is the type of Traits to be used for the variables that are defined to be logged

```
typedef typename ddtTraits::Logger Logger;
typedef typename ddtTraits::Allocator Allocator;
typedef typename objectTraits::template transform<T>::value_type
value_type;

typedef typename ddtTraits::transform<BinaryTreeNode*>::value_type
                 element_pointer;
typedef typename ddtTraits::transform<BinaryTreeNode&>::value_type
                 element_reference;
typedef typename objectTraits::template
        transform<BinaryTreeNode*>::value_type element_pointer_unlogged;

typedef ddtTraits Traits;
```

At this level we define the types that will be used throughout the chain of inheritance of the Library:

Original Class	Class synonym	Description
ddtTraits::Logger	Logger	The Logger to be used for creating the log packets.
ddtTraits::Allocator	Allocator	The Allocator to be used for overriding the new and delete operators.
objectTraits::template transform<T>::value_type	value_type	Type of data hold by the data structure. Defaults to a synonym of T and accesses are not being logged (if objectTraits = std_traits).
ddtTraits::transform<BinaryTreeNode*>::value_type	element_pointer	Pointer to an element. Accesses to this pointer are being logged automatically (if ddtTraits != std_traits).
ddtTraits::transform<BinaryTreeNode&>::value_type	element_reference	Reference to an element. Accesses to this reference are being logged automatically (if ddtTraits != std_traits).
objectTraits::transform<BinaryTreeNode*>::value_type	element_pointer_unlogged	Pointer to an element. Accesses to this pointer are not being logged by default (if objectTraits = std_traits).
ddtTraits	Traits	Exposure of the logger Traits to whole Library.

Table 8 - Binary Tree Node Types

Currently, our Library supports 2 kinds of list nodes, the BTreeNode (representing the node of a B-Tree) and the BinaryTreeNode (representing the node of a Binary Tree).

Due to the structural incompatibilities of these two node types, we preferred to define different interfaces to each node.

a. The BTreeNode Module

The BTreeNode module represents the node of B-Tree and consists of

Three integers:

- The number of keys in the node
- The order of the B-Tree
- The number of keys in the subtree rooted at this node

Two pointers:

- A (logged) pointer to the vector of items of type T
- A (logged) pointer to the vector of subtrees

And one Boolean value defining if the current node is a Leaf or not

The BTreeNode module implements the following interface:

Function Name	Description
`Size()`	Returns the number of items in this node.
`Item(short n)`	Returns the n-th item.
`Subtree(short n)`	Returns a pointer to the n-th subtree.
`SubtreeSize()`	Returns the number of keys in the subtree rooted at this node. This method consults an instance variable, and therefore takes constant time. It does not need to traverse the subtree.
`Search (value_type itm, short& index)`	Search the node for the given key;. return greatest i such that key[i] <= each key. Return true if key[i] = key, false otherwise.
`getKeyCount()`	Returns the number of keys in the node.
`isLeaf()`	True is the current node is a leaf, false otherwise.
`getSubTreeSize()`	Returns the number of subtrees rooted at this node.
`getOrder()`	Returns the order of the tree.
`getItem()`	Returns a pointer to vector of items hold by the node.
`getSubTree(short n)`	Returns a pointer to n-th subtree rooted at this node.

Table 9 - BTree Node Interface

b. The BinaryTreeNode Module

The BinaryTreeNode module represents the node of Binary Tree and consists of Four pointers:

- A pointer the data hold by the node
- A pointer to the left child node
- A pointer to the right child node
- A pointer to the parent node

And an integer, which is the number of nodes of the subtree rooted at this node.
The BinaryTreeNode module implements the following interface:

Function Name	Description
`Left()`	Returns a pointer to the left child node
`Right()`	Returns a pointer to the right child node.
`Parent()`	Returns a pointer to the parent node.
`SubtreeSize()`	Returns the number of nodes of the subtree rooted at this node
`getData()`	Returns a reference to the data hold by the element.
`getDataPointer()`	Returns a pointer to the data hold by the element.

Table 10 - Binary Tree Node Interface

2. Element Extension Level Modules

At this level we define element extension modules for the already elements (nodes).
Currently, our Library supports the extension of the BinaryTreeNode to RedBlackTreeNodeExtension module.

a. The RedBlackTreeNodeExtension Module

The RedBlackTreeNodeExtension Module takes one template argument, the type of module to extend.

```
template <class treeNode>
    class RedBlackTreeNodeAdapter : public treeNode
```

This extension module is designed to extend BinaryTree compatible modules only.
The actual extension takes place by defining an extra member, which is variable representing the “colour” of the node

```
short _color;
```

And extending the interface with a function returning this information.

```
short Color() { return _color; }
```

3. Tree Base Level Modules

The Modules of this level consist of the main implementation of the functionality of the Tree types (B-Tree, Binary Tree, etc.) that cannot be merged into one base implementation with the base-expander technique. In our Library, we currently support two structurally different tree types, the B-Tree and the Binary Tree. The also supported Red-Black Tree, is defined as an extension to Binary Tree, due to their structural similarities, as we shall notice further on.

a. The BTree Module

The BTree module takes one template argument which is the type of the element to be used in the tree implementation.

```
template<class elementType> class BTree;
```

It consists of two integers:

- The order (maximum number of items (keys) per node) of the B-Tree.
- The (unique) instance_id of the Tree.

And a pointer to the root element of Tree.

The BTree Module also implements the following interface:

Function Name	Description
`Find(const_input_reference item)`	Searches the tree for the given item. Returns a pointer to the found item in situ if the search was successful. If the search fails, the return value is NULL. The algorithm used is a standard B-tree search algorithm that takes $\log_d(n)$ time in an n-item B-tree of order d.
`Smallest()`	Finds and returns the minimum item. If the tree is empty, the null pointer is returned.
`Largest()`	Finds and returns the maximum item. If the tree is empty, the null pointer is returned
`ItemWithRank(long rank)`	Returns the element with the given rank.
`RankOf(value_type item)`	Returns the number of elements in the tree that are less than the parameter.
`Size()`	Returns the number of items currently present in the tree.
`Insert(value_type item)`	Inserts the item to the tree. Returns true if successfully added, false if the item was already in the tree.

`Remove(value_type key)`	Remove the specified item from the tree. Return NULL if the item was not found in the tree or the found item otherwise.
`ExtractMin()`	Removes and returns the smallest item in the tree. Returns NULL if the tree is empty.
`increment(element_pointer_unlogged data)`	Returns a pointer to the next element of the input element. Exclusively used by the iterator class.
`decrement(element_pointer_unlogged data)`	Returns a pointer to the previous element of the input element. Exclusively used by the iterator class.
`getEnd()`	Returns an pointer to the end (sentinel) of the list. (For Iterator use).
`getREnd()`	Returns an pointer to the end (sentinel) of the list. (For reverse Iterator use – needed due to compatibility issues with the vector class).
`getData(element_pointer_unlogged data)`	Returns a reference to the data hold by a specific element pointer.

Table 11 - BTree Interface

b. The BinaryTree Module

The BinaryTree module takes one template argument which is the type of the element to be used in the tree implementation.

```
template<class elementType> class BinaryTree;
```

It consists of one integer, which is the (unique) instance_id of the Tree.
And two pointers:

- A pointer to the root element of Tree.
- A pointer to the sentinel element.

The Binary Tree Module also implements the following interface:

Function Name	Description
`Find(const_input_reference item)`	Searches the tree for the given item. Returns a pointer to the found item in situ if the search was successful. If the search fails, the return value is the sentinel element.
`Insert(const_input_reference k)`	Inserts a item of value k
`Delete(element_pointer_unlogged z)`	Deletes the element pointed by the input pointer
`Minimum()`	Returns a pointer the element with the smallest key

`Maximum()`	Returns a pointer the element with the largest key
`ExtractMin()`	Removes and returns a pointer to the smallest item in the tree. Returns a pointer to the sentinel element if the tree is empty.
`ItemWithRank(size_type n)`	Returns a pointer to item (key) hold by n-th element
`getSize()`	Returns the number of elements of the tree
`empty()`	True, if the tree has zero element, false otherwise
`Root()`	Returns a pointer to the root element of the tree
`Sentinel()`	Returns a pointer to the sentinel element
`getFirstElement()`	Returns a pointer to the element holding the smallest item
`getLastElement()`	Returns a pointer to the element holding the largest item
`increment(element_pointer_unlogged data)`	Returns a pointer to the next element of the input element. Exclusively used by the iterator class.
`decrement(element_pointer_unlogged data)`	Returns a pointer to the previous element of the input element. Exclusively used by the iterator class.
`getEnd()`	Returns an pointer to the end (sentinel) of the list. (For Iterator use).
`getREnd()`	Returns an pointer to the end (sentinel) of the list. (For reverse Iterator use – needed due to compatibility issues with the vector class).
`getData(element_pointer_unlogged data)`	Returns a reference to the data hold by a specific element pointer.

Table 12 - Binary Tree Interface

4. Tree Type Implementation Extension Level Modules

Each module of this layer presents structural similarities with the module that it extends and optimizes it.

```
template <class treeType> class RedBlackTreeAdapter : public treeType
```

A template argument is required, which is the type of tree base module to be extended. Currently, only the Binary Tree is optimized and extended to the Red Black Tree:

a. The RedBlackTreeExtension Module

The RedBlackTreeExtension overrides the Insert() and Delete() functions of the BinaryTree (which have been declared virtual) with new extended ones with tree balancing logic.

```
virtual element_pointer_unlogged Insert(const_input_reference k){
      element_pointer_unlogged x = treeType::Insert(k);
      RBInsertFixup(x);
      return z;
}
```

5. The Tree Module

The Tree Module is the exposes the complete tree functionality through an STL compatible interface

```
template <class treeType> class Tree: public treeType
```

It takes one template argument, which is the type of the tree module containing the implementation of all the Tree's functions. This module actually acts as adapter to the input tree module's interface, providing STL compliance including Iterator traversal. For this purpose, it defines iterators (normal, reverse and const) both logged and unlogged.

```
typedef Iterator_Base<Tree> iterator;
typedef ConstIteratorAdapter<iterator> const_iterator;
typedef ReverseIteratorAdapter<iterator> reverse_iterator;
typedef ReverseIteratorAdapter<const_iterator> const_reverse_iterator;

typedef Iterator_Base_Unlogged<Tree> iterator_unlogged;
typedef ReverseIteratorAdapter<iterator_unlogged>
         reverse_iterator_unlogged;
```

Logged iterators are intended for the default Library use. Unlogged iterators are useful for combining the Tree DDT with other DDTs (i.e. GlueLayer).
The exposed interface is the following:

Function Name	Description
`begin()`	Returns an Iterator pointing to the first element of the tree.
`end()`	Returns an Iterator pointing to the first element of the tree.
`rbegin()`	Returns a reverse Iterator pointing to the first element of the tree.
`rend()`	Returns a reverse Iterator pointing to the first element of the tree.
`empty()`	True tree empty, false ortherwise.
`size()`	Returns the number of elements contained by the tree.
`resize(size_type new_size, const_input_reference value)`	Sets a new number of elements to the tree. Pushes back new (initialized to the input value) or removes some if needed.
`reserve(size_type new_capacity)`	Reserves memory space to the tree.
`operator[](size_type n)`	Returns the n-th element contained by the tree.
`get(size_type n)`	Returns the n-th element contained by the tree.
`front()`	Returns a reference to data hold by the first element of the tree.
`back()`	Returns a reference to data hold by the last element of the tree.
`erase(iterator& pos)`	Erases the element at current Iterator position.
`insert(const_input_reference value)`	Inserts a new element in the tree.

Table 13 - Tree Interface

9.3. The Vector DDT

The Vector DDT is organized as a single module due to its simplicity as a data structure.

```
template <class T, class objectTraits, class ddtTraits> class Vector
```

The Vector Module takes 3 template arguments:

- a class T, which is the type to be wrapped the list element
- a class objectTraits, which is the type of Traits to be used for the variables that are defined not to be logged
- a class objectTraits, which is the type of Traits to be used for the variables that are defined to be logged

```
typedef typename ddtTraits::Logger Logger;
typedef typename ddtTraits::Allocator Allocator;
typedef typename objectTraits::template transform<T>::value_type
value_type;

typedef typename ddtTraits::transform<T*>::value_type
                 element_pointer;
typedef typename ddtTraits::transform<T&>::value_type
                 element_reference;
typedef typename objectTraits::template
        transform<T*>::value_type element_pointer_unlogged;
typedef ddtTraits Traits;
```

Like the previous DDTs we define the types that will be used throughout the module and are exposed to the user of our Library:

Original Class	Class synonym	Description
`ddtTraits::Logger`	`Logger`	The Logger to be used for creating the log packets.
`ddtTraits::Allocator`	`Allocator`	The Allocator to be used for overriding the new and delete operators.
`objectTraits::template transform<T>::value_type`	`value_type`	Type of data hold by the data structure. Defaults to a synonym of T and accesses are not being logged (if objectTraits = std_traits).

`ddtTraits::transform<T*>::value_type`	`element_pointer`	Pointer to an element. Accesses to this pointer are being logged automatically (if ddtTraits != std_traits).
`ddtTraits::transform<T&>::value_type`	`element_reference`	Reference to an element. Accesses to this reference are being logged automatically (if ddtTraits != std_traits).
`objectTraits::transform<T*>::value_type`	`element_pointer_unlogged`	Pointer to an element. Accesses to this pointer are not being logged by default (if objectTraits = std_traits).
`ddTraits`	`Traits`	Exposure of the logger Traits to whole Library.

Table 14 - Vector Types

Our implementation of the Vector is 100% STL compliant, and since the Vector already exists in the STL (unlike the List and Tree DDTs), it is unnecessary to present the implemented interface.

9.4. The Iterator Class

The Iterator module extends a data structure's functionality providing the Iterator's design pattern flexibility and abstraction.

```
template <class DataStructureType> class Iterator_Base
```

It takes one template parameter, which is the type of the data structure to be provided with such capability.
The Iterator module consists of
Two pointers:

- A pointer to the data structure being inspected
- A pointer to the element of the data structure currently being inspected

And one integer, which is the instance_id of the data structure (for logging purposes).
In order a class to be able to take advantage of this module, it must implement the following interface:

Function Name	Description
`increment(element_pointer_unlogged data)`	Returns a pointer to the next element of the input element. Exclusively used by the iterator class.
`decrement(element_pointer_unlogged data)`	Returns a pointer to the previous element of the input element. Exclusively used by the iterator class.
`getData(element_pointer_unlogged data)`	Returns a reference to the data hold by a specific element pointer.

Table 15 - Iterator Required Interface

The above functions are used by the Iterator when defining its own increment() and decrement() functions.

```
virtual void increment() {
      _data = _datastructure->increment(_data);
}
virtual void decrement() {
      _data = _datastructure->decrement(_data);
}
```

These functions are declared virtual in order to be overridden by the Reverse_Iterator module that provides reverse iteration capabilities to the data structure

```
virtual void increment() {
      iterator::decrement();
}
virtual void decrement() {
      iterator::increment();
}
```

The Iterator Module also overloads operators like operator++ and opereator*, making the DDT traversal easy and clear.

```
for (list::iterator i = myList.begin(); i != myList.end(); ++i) {
      cout << *i << " ";
}
```

9.5. The GlueLayer Class

The GlueLayer Module has been designed to integrate two of the DDTs described above (including the GlueLayer itself!), into one multi-dimensional data structure.

```
template <class ExternalDDT, class InternalDDT> class GlueLayer
```

It takes two template arguments:

- The type of the external DDT
- The type of the internal DDT

It mainly consists of an externalDDT, a countState vector<int> and a pointer to the most recently used element of the externalDDt.
The exposed types are the ones exposed by the internalDDT, which holds the actual data:

```
enum { type_num = InternalDDT::type_num };
typedef typename InternalDDT::Logger Logger;

//Internal Element Types
typedef typename InternalDDT::pointer pointer;
typedef typename InternalDDT::reference reference;
typedef typename InternalDDT::const_reference const_reference;
typedef typename InternalDDT::element_pointer element_pointer;
typedef typename InternalDDT::element_reference element_reference;

typedef typename InternalDDT::element_pointer_unlogged
                                element_pointer_unlogged;

typedef typename InternalDDT::input_value_type input_value_type;
typedef typename InternalDDT::const_input_reference
                                const_input_reference;

typedef typename InternalDDT::size_type size_type;
typedef typename InternalDDT::value_type value_type;
typedef typename InternalDDT::difference_type difference_type;

typedef typename InternalDDT::iterator_unlogged
                                internal_iterator_unlogged;
typedef typename InternalDDT::reverse_iterator_unlogged
                                internal_reverse_iterator_unlogged;
```

The GlueLayer wraps the externalDDT and according the countState vector, decides where to insert a new element, which the n-th element is, etc.
It was also designed to be STL compliant, so the exposed interface is similar to ones described above.

Below you can see an example of usage of the GlueLayer and our DDTR Library:

```
template <typename T, int I = 1>
struct ddt_helper {
	//List definitions
	typedef DDTLibrary::sllListElement<T, Tracing::std_traits,
			Tracing::var_traits<TRAITS(I)>> SLLELEMENT;
	typedef DDTLibrary::listBase<SLLELEMENT> SLLLISTBASE;
	typedef DDTLibrary::sll<SLLLISTBASE> SLLLISTTYPE;

	typedef DDTLibrary::List<SLLLISTTYPE> SLLLIST;

	//Vector definitions
	typedef DDTLibrary::Vector<T, Tracing::std_traits,
			Tracing::var_traits<TRAITS(I)>> VECTOR;

	//Glue Layer definitions
	typedef VECTOR INTERNALDDT;
	typedef SLLLIST EXTERNALDDT;

	typedef DDTLibrary::GlueLayer<SLLLIST, INTERNALDDT> GLUEDTYPE;

	//Types
	typedef GLUEDTYPE type;

};

int main()
{
	Tracing::DMMLogger::getInstance()->setLogFile("log.bin");
	Tracing::DMMLogger::getInstance()->setAlternateFile("log.map");

	typedef ddt_helper<int, 10>::type myGluedType;

	//Glue Layer Testing
	cout << endl << endl << "Glue Layer Testing" << endl;

	cout << endl << endl << "Testing push_back" << endl;
	myGluedType glue;
	for (int i = 0; i < 83; i++) {
		glue.push_back(i);
	}

	int k = 0;
	for (myGluedType::iterator i = glue.begin(); i != glue.end(); k++) {
		if (k / 10 == 1)
			i = glue.erase(i);
		else
			i++;
	}

	cout << endl << endl << "Testing operator[] After erase : " << endl;
	for (myGluedType::iterator i = glue.begin(); i != glue.end(); ++i) {
		cout << glue[i] << " ";
	}

	return 0;
}
```

10. Description of the Profiler

10.1. The Logger Class

Two logger types are currently available:

- The DMMLogger
- The TextLogger

Both of the loggers have compatible, so we can parameterize the logger selection. This parameterization takes place in traits structure, where the logger (as well as the allocator) to be used is predefined. Below is an example taken from the implementation of the DDT Library that shows exactly the usage of the Logger:

```
void push_back(const_input_reference value) {
      element_pointer tmp = listType::push_back(value);
      #ifdef LOGGING
      typename Logger::object_insert S(TYPEID(input_value_type),
         type_num, instance_id_,    sizeof(input_value_type), tmp);
      #endif
}
```

The supported packets are the following:

Packet ID	Packet Name
0	LOG_VAR_READ
1	LOG_VAR_WRITE
2	LOG_SCOPE_BEGIN
3	LOG_SCOPE_END
4	LOG_MALLOC_BEGIN
5	LOG_MALLOC_END
6	LOG_FREE_BEGIN
7	LOG_FREE_END
11	LOG_SEQUENCE_GET
12	LOG_SEQUENCE_ADD
13	LOG_SEQUENCE_REMOVE
14	LOG_SEQUENCE_CLEAR
15	LOG_SEQUENCE_END
20	LOG_MAP_START
21	LOG_MAP_GET

22	LOG_MAP_ADD
23	LOG_MAP_REMOVE
24	LOG_MAP_CLEAR
30	LOG_OBJECT_CONSTRUCT
31	LOG_OBJECT_COPY
32	LOG_OBJECT_DESTRUCT
33	LOG_OBJECT_SWAP
34	LOG_OBJECT_RESIZE
35	LOG_OBJECT_GET
36	LOG_OBJECT_INSERT
37	LOG_OBJECT_REMOVE
38	LOG_OBJECT_CLEAR
39	LOG_OBJECT_OPERATOR
40	LOG_OBJECT_END
41	LOG_ITERATOR_NEXT
42	LOG_ITERATOR_PREVIOUS
43	LOG_ITERATOR_ADD
44	LOG_ITERATOR_SUB
45	LOG_ITERATOR_GET
46	LOG_ITERATOR_RESET
47	LOG_ITERATOR_ISDONE
254	LOG_USED_MEMORY
255	LOG_ADDRESS_RANGE

Table 16 - Log Packets

All of the packets have a 4-byte header consisting of 2 2-byte fields:

- The Packet ID
- The size in bytes of the rest of the packet

More specific information about some of the packets is found below:

Packet Name	No of fields	Field 1	Field 2	Field 3	Field 4	Field 5	Field 6
LOG_VAR_READ	3	DDT ID	address	size			
LOG_VAR_WRITE	3	DDT ID	address	size			
LOG_SCOPE_BEGIN	1	scope name					
LOG_SCOPE_END	1	scope name					
LOG_MALLOC_BEGIN	2	DDT ID	size				
LOG_MALLOC_END	3	DDT ID	address	size			
LOG_FREE_BEGIN	2	DDT ID	address				
LOG_FREE_END	2	DDT ID	address				
LOG_OBJECT_CONSTRUCT	4	data type	DDT ID	instance ID	size		
LOG_OBJECT_COPY	6	data type	DDT ID	instance ID	size	old DDT ID	old instance ID
LOG_OBJECT_DESTRUCT	4	data type	DDT ID	instance ID	size		
LOG_OBJECT_SWAP	6	data type	DDT ID	instance ID	size	old DDT ID	old instance ID
LOG_OBJECT_RESIZE	4	data type	DDT ID	instance ID	size		
LOG_OBJECT_GET	5	data type	DDT ID	instance ID	size	index	
LOG_OBJECT_INSERT	5	data type	DDT ID	instance ID	size	address	
LOG_OBJECT_REMOVE	5	data type	DDT ID	instance ID	size	address	
LOG_OBJECT_CLEAR	3	data type	DDT ID	instance ID			
LOG_OBJECT_OPERATOR	3	data type	DDT ID	instance ID			
LOG_OBJECT_END	0						
LOG_ITERATOR_NEXT	5	data type	DDT ID	instance ID	size	address	
LOG_ITERATOR_PREVIOUS	5	data type	DDT ID	instance ID	size	address	
LOG_ITERATOR_ADD	6	data type	DDT ID	instance ID	size	address	offset
LOG_ITERATOR_SUB	6	data type	DDT ID	instance ID	size	address	offset
LOG_ITERATOR_GET	5	data type	DDT ID	instance ID	size	address	
LOG_ITERATOR_ISDONE	3	data type	DDT ID	instance ID			

Table 17 - Log Packet Details

The DMMLogger

The DMMLogger is the logger module that outputs the logging information (packet structured) in a binary file. This is the centralized logger because the binary file output is the input of the Analyzer.

```
DMMLogger::getInstance()->setLogFile("log.bin");
```

First of all, the class DMMLogger defines an enumeration that connects the packet names with numbers to be used as IDs.

Afterwards, several inner structs are defined, each responsible of creating (outputting) the corresponding packet. This operation takes place in the constructor of the struct as shown in the example below:

```
class DMMLogger {

      enum LogType {
           LOG_OBJECT_INSERT      = 36,
           LOG_OBJECT_END         = 40
      };
public:

      struct object_insert {
            object_insert(
                  const char * type_id,
                  const unsigned int type_num,
                  const unsigned int instance_id,
                  const size_t sz,
                  const void * addr)
                  {
                        getInstance()->write_header(LOG_OBJECT_INSERT,
                          5*sizeof(unsigned int))
                              << getInstance()->getTypeNum(type_id)
                              << type_num
                              << instance_id
                              << sz
                              << addr;
                  };
            ~object_insert() {
                        getInstance()->write_header(LOG_OBJECT_END, 0);
            }
      };
}
```

The DMMLogger also keeps a functionMap (STL map) and a typeMap in order to map the function names (scopes) and type names to integer values that were used in the packet information to specify the type of the data being logged.

```
typedef std::map<std::string, unsigned int> typemap;

unsigned int getTypeNum(std::string name) {
      std::pair<typemap::iterator, bool> x =
         typeMap_.insert(typemap::value_type(name, typeCtr_++));
      if (!x.second) {
            typeCtr_--;
      }
      return (*(x.first)).second;
}
```

The DMMLogger exports them at the end of the logging procedure (during the destruction of the DMMLogger instance) to separate text file.

```
DMMLogger::getInstance()->setAlternateFile("log.map");
```

The TextLogger

The TextLogger is the logger module that outputs the logging information in a simple text file. This logger is used for debugging our Profiling infrastructure.
It has been implemented just like the DMMLogger is (the only difference is found in the way information is output).
Below is an example of the TextLogger usage and output:

```
typedef DDTLibrary::vector<int,
   Tracing::var_traits<Tracing::text_traits<1>>> vector1;

int main()
{
      //Vector testing
      cout << endl << endl << "Vector testing" << endl;
      vector1 v;
      v.push_back(3);
      v.push_back(5);
      cout << v[0] << endl;
      v[0] = v[1];
      cout << "Destroying vector v" << endl;
      v.~vector();
}

Output:
Vector testing
      LOG_OBJECT_CONSTRUCT_BEGIN int 1 0 4
      LOG_MALLOC_BEGIN 1 64
      LOG_MALLOC_END 1 00366568 64
      LOG_OBJECT_CONSTRUCT_END
      LOG_OBJECT_INSERT_BEGIN int 1 0 4 00366568
      LOG_VAR_WRITE 1 00366568 4
      LOG_OBJECT_INSERT_END
      LOG_OBJECT_INSERT_BEGIN int 1 0 4 0036656C
      LOG_VAR_WRITE 1 0036656C 4
      LOG_OBJECT_INSERT_END
      LOG_OBJECT_GET_BEGIN int 1 0 4 0
      LOG_OBJECT_GET_END
      LOG_VAR_READ 1 00366568 4
3
      LOG_OBJECT_GET_BEGIN int 1 0 4 1
      LOG_OBJECT_GET_END
      LOG_OBJECT_GET_BEGIN int 1 0 4 0
      LOG_OBJECT_GET_END
      LOG_VAR_READ 1 0036656C 4
      LOG_VAR_WRITE 1 00366568 4
Destroying vector v
      LOG_OBJECT_DESTRUCT_BEGIN int 1 0 4
      LOG_OBJECT_CLEAR_BEGIN int 1 0
      LOG_OBJECT_CLEAR_END
      LOG_FREE_BEGIN 1 00366568
      LOG_FREE_END 1 00366568
      LOG_OBJECT_DESTRUCT_END
```

10.2. The Allocated Class

The Allocated Module is the module that defines the logged_allocator and allocated classes The logged_allocator is a class that extends the malloc and free functions of the C Standard Library by calling appropriate Logger functions to log the operation.

The allocated is a template class that takes as a parameter the allocator and extends the new and deletes operators by calling the input allocator's malloc and free functions respectively instead of calling the ones of the Standard Library.

The great advantage of this pattern is that every class that inherits from the allocated class will be logged when being created by a new operator.

10.3. The Traits Class

The Traits Module is module that defines the trait structs. The role of these structs is to gather in one entity the information about which Logger and Allocator to be used. Each trait is a template struct that takes as a parameter the DDT ID.

There are five types of traits:

- The BinTraits
  The BinTraits defines the use of the DMMLogger as a Logger and the logged_allocator as an Allocator.

```
template <int I>
struct bin_traits {
	enum { type_num = I };
	template <typename T>
	struct transform {
		typedef T value_type;
	};
	typedef DMMLogger Logger;
	typedef logged_allocator<I, DMMLogger> Allocator;
};
```

- The BinSemiTraits
  The BinTraits defines the use of the DMMLogger as a Logger, but the standard Allocator (logs nothing) as an Allocator.

```
template <int I>
struct bin_semi_traits {
	enum { type_num = I };
	template <typename T>
	struct transform {
		typedef T value_type;
	};
	typedef DMMLogger Logger;
	typedef std_allocator Allocator;
};
```

- The TextTraits
  The TextTraits defines the use of the TextLogger as a Logger and the logged_allocator as an Allocator.

- The TextSemiTraits
  The TextSemiTraits defines the use of the TextLogger as a Logger, but the standard Allocator as an Allocator.

- The VarTraits (described below)

10.4. The Scope Class

The Scope Module is a simple logging module that cooperates with the Logger module in order to output scope related logging information.

The Scope Module defines a scope class that (in a nutshell) imitates the Logger's inner structs functionality. The logging operations take place in the construction and destruction of the class as well.

```
void foo() {
      SCOPE s("foo");
      {
            SCOPE s("bar");
      }
}
```

10.5. The Var Class

The Var Module defines the Var template class and the VarTraits.

The VarTraits Modifies takes a traits as a parameter and modifies it by redefining the value_type to be wrapped by the var template:

```
template <class traits>
struct var_traits {
      enum { type_num = traits::type_num };
      template <typename T>
      struct transform {
            typedef var<T, traits> value_type;
      };
      typedef traits::Logger Logger;
      typedef traits::Allocator Allocator;
};
```

The Var template class is a smart wrapper that derives from allocated template class (described above) and allows to easily check all the accesses made to the datatype that has been wrapped. Because this is a memory-level access-checker and not DDT-access checker, it should be placed only around basic types as it has not been designed for complicated types. This is not too bad as all complicated types are constructed from basic types. (In the end memory is only occupied by chars, ints, floats, pointers, etc...)

The Var class derives from the Allocated class in order to log all memory allocations and deallocations of the wrapped datatype:

```
template <typename T, typename Traits>
class var : public allocated<typename Traits::Allocator>
```

The wrapping of the datatype becomes feasible by overriding all sorts of operators and accessors with logged ones:

```
// Accessors
template <typename T, typename Traits>
var<T, Traits>::operator const T &() const {
      Traits::Logger::log_read(&(data_), Traits::type_num, sizeof(T));
      return data_;
}
// Assignment operators from basic type
template <typename T, typename Traits>
template <typename T2>
var<T, Traits> & var<T, Traits>::operator= (const T2 & data) {
      data_ = data;
      Traits::Logger::log_write(&(data_), Traits::type_num, sizeof(T));
      return *this;
}
```

```
// Assignment operators from wrapped type
template <typename T, typename Traits>
template <typename T2, typename Traits2>
var<T, Traits> & var<T, Traits>::operator= (const var<T2, Traits2> &
other) {
      Traits2::Logger::log_read(&(other.data_), Traits2::type_num,
sizeof(T2));
      data_ = other.data_;
      Traits::Logger::log_write(&(data_), Traits::type_num, sizeof(T));
      return *this;
}
// Unary operators
template <typename T, typename Traits>
T var<T, Traits>::operator+() const {
      Traits::Logger::log_read(&(data_), Traits::type_num, sizeof(T));
      return +data_;
}
```

11. Description of the Analyzer

The Analyzer is a program that parses the binary log file produced by the dmmLogger (as described above) and fills a MySql Database with various statistic results concerning the DDTs that were logged in the client application. At the same time, it exports the whole profiling information into an XML file, as well as a database table, in order to support, in a flexible manner, further connection with other analysis tools.

The Header file

The header file includes the declaration of types used in the Analyzer.
There are two main categories of declarations:

1) Direct declarations using the C++ typedef keyword.
   The types declared this way are constructed by typical STL data structures and are for local information stored in Analyzer's memory before updating the database.

```
typedef struct ddtgroupstatistics_tag {
      int maxInstances;
      int maxObjects;
      int maxAllocatedMem;
      int totalAllocatedMem;
      int currentInstances;
      int currentAllocatedMem;
      int currentObjects;
      int numOperations;
      int totalAccesses;
      int numPackets;
} ddtgroupstatistics_T;
typedef unsigned int fourb;
typedef map<fourb, ddtgroupstatistics_T> ddtgroupstatsmap;
```

2) Inderect declarations using mysql++ (c++ API for MySQL) predefined macro instructions.
   The types declared this way represent the row of each table that will be modified by the Analyzer. The first parameter of the sql_create macro is the name of the corresponding table, as well as the name of the struct to be created. The next two parameters refer (in a nutshell) to number of parameters of the constructors of the struct. Then come pairs of field types and names (of both struct and table).

```
sql_create_8(ddtgroupstatistics,
           1, 8,
           mysqlpp::sql_bigint, ddtID,
           mysqlpp::sql_bigint, totalAccesses,
           mysqlpp::sql_bigint, maxInstances,
           mysqlpp::sql_bigint, maxObjects,
           mysqlpp::sql_bigint, averageObject,
           mysqlpp::sql_bigint, maxAllocatedMem,
           mysqlpp::sql_bigint, totalAllocatedMem,
           mysqlpp::sql_bigint, numPackets)
```

11.1. Preprocessing phase

➔ The Analyzer reads the map-file(that includes supplementary info concerning the DDT scope and type names) and stores that information in an STL map data structure:

```
getline (mapfile,line);
while ( line.find_first_of('}') == string::npos )
{
      string name(line.begin() + line.find_first_of('>') + 2,
        line.end());
      string numb(line.begin() + 2, line.begin() +
        line.find_first_of('-') - 1);
      buf = new char[numb.size() + 1];
      strcpy(buf,numb.c_str());
      number = atoi(buf);
      names.insert(logmap::value_type(number, name));
      getline (mapfile,line);
}
```

➔ The Analyzer stores in an STL map data structure all the supported packets' IDs and text names:

```
logmap_.insert(logmap::value_type(0, "LOG_VAR_READ"));
logmap_.insert(logmap::value_type(1, "LOG_VAR_WRITE"));
```

The Packets currently supported are the following:

Packet ID	Packet Name
0	LOG_VAR_READ
1	LOG_VAR_WRITE
2	LOG_SCOPE_BEGIN
3	LOG_SCOPE_END
4	LOG_MALLOC_BEGIN
5	LOG_MALLOC_END
6	LOG_FREE_BEGIN
7	LOG_FREE_END
30	LOG_OBJECT_CONSTRUCT
31	LOG_OBJECT_COPY
32	LOG_OBJECT_DESTRUCT
33	LOG_OBJECT_SWAP
34	LOG_OBJECT_RESIZE
35	LOG_OBJECT_GET
36	LOG_OBJECT_INSERT
37	LOG_OBJECT_REMOVE
38	LOG_OBJECT_CLEAR
39	LOG_OBJECT_OPERATOR
40	LOG_OBJECT_END
41	LOG_ITERATOR_NEXT
42	LOG_ITERATOR_PREVIOUS
43	LOG_ITERATOR_ADD
44	LOG_ITERATOR_SUB
45	LOG_ITERATOR_GET
46	LOG_ITERATOR_RESET
47	LOG_ITERATOR_ISDONE

Table 18 - Analyzer Packets

➔ The Analyzer connects to the Database and opens the XML file

➔ The Analyzer opens and determines the size of the log file.

Processing phase

The Analyzer repeatedly reads packets (bytes) from the binary file and updates a counter storing the bytes read so far. First of all, it determines the type of the packet about to be read. For performance reasons, it does not use the size information (logging metadata) of the packet existing in the header of every packet. Instead, the number and type of the fields of the packet is handled by the Analyzer in hard-coded manner:

```
for (i=0;i<lSize;) {
      file2 << "<packet" << k << ">" << endl;
      out = fread(&c2, sizeof(twob), 1, file);
      file2 << "\t<type>\n\t\t" << logmap_[c2] << "\n\t</type>" <<
        endl;
      out = fread(&c22, sizeof(twob), 1, file);
//    more = c2/sizeof(fourb);
      i+=4;
      if (logmap_[c2] == "LOG_VAR_READ") {
            out = fread(&c4_type_num, sizeof(fourb), 1, file);
            file2 << "\t<id>\n\t\t" << c4_type_num << "\n\t</id>" <<
              endl;
            out = fread(&c4_addr, sizeof(fourb), 1, file);
            file2 << "\t<addr>\n\t\t" << c4_addr << "\n\t</addr>" <<
              endl;
            out = fread(&c4_size, sizeof(fourb), 1, file);
            file2 << "\t<size>\n\t\t" << c4_size << "\n\t</size>" <<
              endl;
            i+=12;
```

Actions performed when each packet is processed

➔ Packet "LOG_VAR_READ" :

Update blockscope table by increasing the number of reads of the block determined by its address. We are sure that this is the definitive information, since we take into consideration the freed status that declares that the block is currently active (not freed).

```
query.reset();
query << "select * from blockscope where blockAddress = "
   << c4_addr
   << " and freed = 0";
if (mysqlpp::StoreQueryResult res = query.store()) {
      if (!res.empty()) {
            blockscope row = res[0];
            blockscope orig_row = row;
            row.numReads++;
            query.reset();
            query.update(orig_row, row);
            query.execute();
      }
}
```

Update the ddtstats vector by increasing the number of reads of the current DDT instance. This current instance is determined by instanceID field of the last OBJECT packet.

```
ddtstats[currentInstanceID].numReads++;
```

Update the stats vector by increasing the number of reads.

```
stats.num_reads++;
```

Update the ddtgroupstatsmap by increasing the number of packets of the current DDT group (determined by ddtID of the packet).

```
ddtgroupstatsmap::iterator groupfound =
   ddtgroupstats.find(c4_type_num);
groupfound->second.numPackets++;
```

Update ddtgrouppacketstatisticsmap by increasing the number of read packets of the current pair<ddtID, packetGroup>. Create a new map entry if necessary.

```
pair<fourb, unsigned int> key(c4_type_num, ((int)
   groupfound->second.numPackets/segmentSize));
ddtgrouppacketstatisticsmap::iterator currentpacketgroup =
   ddtpacketstats.find(key);
if (currentpacketgroup == ddtpacketstats.end()) {
      ddtgrouppacketstatistics_T grouppacketstat = {1, 0, 0, 0, 0, 0,
        0, 0, 0};
      ddtpacketstats.insert(ddtgrouppacketstatisticsmap::value_type(key
        , grouppacketstat));
}
else {
      currentpacketgroup->second.numReadPackets++;
}
```

➔ Packet “LOG_VAR_WRITE” :

Update blockscope table by increasing the number of writes of the block. Create a new entry if necessary. Creation takes place here and ot when processing the MALLOC_END packet because the later contains information (address) only for first new block and the others if any (case of the vecter DDT). Since write packets are always created right after their allocation, this is the only valid choice to create new entries in the blockscope table.

```
query << "select * from blockscope where blockAdress = "
   << c4_addr
   << " and freed = 0";
if (mysqlpp::StoreQueryResult res = query.store()) {
      if (!res.empty()) {
            blockscope row = res[0];
            blockscope orig_row = row;
            row.numWrites++;
            query.reset();
            query.update(orig_row, row);
            query.execute();
      } else {
```

```
            int sid = scopeStack.top();
            blockscope row(0, c4_addr, c4_size, c4_type_num,
              currentInstanceID, 0, 1, 0);
            query.reset();
            query.insert(row);
            query.execute();
      }
}
```

Update the ddtstats vector by increasing the number of writes of the current DDT instance. This current instance is determined by instanceID field of the last OBJECT packet.
Update the stats vector by increasing the number of writes.
Update the ddtgroupstatsmap by increasing the number of packets of the current DDT group.
Update ddtgrouppacketstatisticsmap by increasing the number of write packets of the current pair<ddtID, packetGroup>. Create a new map entry if necessary.

➔ Packet "LOG_SCOPE_BEGIN" :

Push the corrent scope ID into the scopeStack stack, in order to be available for ddtscope table.

```
scopeStack.push(c4_type_num);
```

Update the scope table by increasing the number of calls of current scope (determined by its ID). Create a new entry if necessary.

```
query << "select * from scope where id= " << c4_type_num;
if (mysqlpp::StoreQueryResult res = query.store()) {
      int row_no = -1;
      if (!res.empty()) {
            scope row = res[0];
            scope orig_row = row;
            row.calls++;
            query.reset();
            query.update(orig_row, row);
            query.execute();
      }
      else {
            query.reset();
            scope row(c4_type_num, names[c4_type_num], 1);
            query.insert(row);
            query.execute();
      }
}
```

➔ Packet "LOG_SCOPE_END" :

Update the stats vector by increasing the number of different scopes defined in the client application.

```
if (stats.num_scopes <= c4_type_num)
```

```
stats.num_scopes = c4_type_num + 1;
```

Pop the current scope from scope Stack stack.

```
scopeStack.pop();
```

➔ Packet "LOG_MALLOC_BEGIN" :

Update resizeHistory table if necessary (depending on previous existence of RESIZE packet).

```
if (update_resize != -1) {
      int el_size, elements;
      query << "select * from ddtscope where instanceID = "
        << update_resize;
      if (mysqlpp::StoreQueryResult res = query.store()) {
            if (!res.empty()) {
                  ddtscope row = res[0];
                  ddtscope orig_row = row;
                  row.resizeHistoryID = next_resize_id;
                  el_size = row.elementSize;
                  elements = row.maxElements = c4_size/el_size;
                  query.reset();
                  query.update(orig_row, row);
                  query.execute();

                  query.reset();
                  resizehistory row2(next_resize_id, update_resize,
                   elements);
                  query.insert(row2);
                  query.execute();
            }
      } else {
            cerr << "Failed to get item list: " << query.error() <<
              endl;
            return 1;
      }
      update_resize = -1;
}
```

Update the ddtgroupstatsmap by increasing the number of packets of the current DDT group.
Update ddtgrouppacketstatisticsmap by increasing the number of malloc packets of the current pair<ddtID, packetGroup>. Create a new map entry if necessary.

➔ Packet "LOG_MALLOC_END" :

Update the stats vector by increasing the number of mallocs.
Update the stats vector by increasing the amount of total allocated memory.

```
stats.total_allocated_mem += c4_size;
```

Update the stats vector by increasing if necessary the amount of max allocated memory.

```
current_allocated_mem += c4_size;
if (stats.max_allocated_mem < current_allocated_mem)
      stats.max_allocated_mem = current_allocated_mem;
```

Update the stats vector by increasing if necessary the amount of max active blocks.

```
current_active_blocks++;
if (stats.max_blocks_active < current_active_blocks)
      stats.max_blocks_active = current_active_blocks;
```

Update the ddtstats vector by increasing the number of mallocs of the current DDT instance.

```
ddtstats[currentInstanceID].numMalloc++;
```

Update the ddtstats vector by increasing the amount of total allocated memory.

```
ddtstats[currentInstanceID].totalAllocatedMem += c4_size;
```

Update the ddtstats vector by increasing if necessary the amount of max allocated memory.

```
ddtstats[currentInstanceID].currentAllocatedMem += c4_size;
if (ddtstats[currentInstanceID].maxAllocatedMem <
  ddtstats[currentInstanceID].currentAllocatedMem)
      ddtstats[currentInstanceID].maxAllocatedMem =
        ddtstats[currentInstanceID].currentAllocatedMem;
```

Update the stats ddtstats by increasing if necessary the amount of max active blocks.

```
ddtstats[currentInstanceID].currentBlocksActive++;
if (ddtstats[currentInstanceID].maxBlocksActive <
   ddtstats[currentInstanceID].currentBlocksActive)
      ddtstats[currentInstanceID].maxBlocksActive =
         ddtstats[currentInstanceID].currentBlocksActive;
```

Update the ddtgroupstatsmap by increasing the number of packets of the current DDT group.

Update the ddtgroupstatsmap by increasing the amount of total allocated memory of the current DDT group.

```
ddtgroupstatsmap::iterator groupfound =
   ddtgroupstats.find(c4_type_num);
groupfound->second.totalAllocatedMem += c4_size;
```

Update the ddtgroupstatsmap by increasing if necessary the amount of max allocated memory of the current DDT group.

```
groupfound->second.currentAllocatedMem += c4_size;
if (groupfound->second.maxAllocatedMem < groupfound->
   second.currentAllocatedMem)
            groupfound->second.maxAllocatedMem = groupfound->
              second.currentAllocatedMem;
```

Update ddtgrouppacketstatisticsmap by increasing the number of malloc packets of the current pair<ddtID, packetGroup>. Create a new map entry if necessary.

➔ Packet "LOG_FREE_BEGIN" :

Update the ddtgroupstatsmap by increasing the number of packets of the current DDT group.
Update ddtgrouppacketstatisticsmap by increasing the number of free packets of the current pair<ddtID, packetGroup>. Create a new map entry if necessary.

➔ Packet "LOG_FREE_END" :

Update the stats vector by increasing the number of frees.
Update the ddtstats vector by increasing the number of frees of the current DDT instance.
Update the ddtgroupstatsmap by increasing the number of packets of the current DDT group.
Update ddtgrouppacketstatisticsmap by increasing the number of free packets of the current pair<ddtID, packetGroup>. Create a new map entry if necessary.

➔ Packet "LOG_OBJECT_CONSTRUCT" :

Update the ddtscope table by inserting a new entry.

```
int sid = scopeStack.top();
query.reset();
ddtscope row(c4_instance_id, sid, c4_type_num, c4_type_id,
   types[c4_type_id], c4_size, -1, -1, 0, 0, 0, 0, 16);
query.insert(row);
```

Update the ddtstats vector by inserting a new entry.

```
ddtstatistics_T dstats = {c4_type_num, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0};
ddtstats.push_back(dstats);
```

Update the ddtgroupstatsmap vector by increasing the number of max ddt instances of current DDT group. Create a new entry if necessary.

```
ddtgroupstatsmap::iterator groupfound =
   ddtgroupstats.find(c4_type_num);
if (groupfound == ddtgroupstats.end()) {
      ddtgroupstatistics_T groupstat = {1, 0, 0, 0, 1, 0, 0, 0, 0, 1};
      ddtgroupstats.insert(ddtgroupstatsmap::value_type(c4_type_num,
         groupstat));
}
else {
      groupfound->second.currentInstances++;
      groupfound->second.numPackets++;
      if (groupfound->second.maxInstances <
         groupfound->second.currentInstances)
```

```
            groupfound->second.maxInstances =
              groupfound->second.currentInstances;
}
```

➔ Packet “LOG_OBJECT_COPY” :

Update the ddtscope table by inserting a new entry.
Update the ddtstats vector by inserting a new entry.
Update the ddtgroupstatsmap by increasing the number of max ddt instances of of the current DDT group. Create a new entry if necessary.

➔ Packet “LOG_OBJECT_DESTRUCT” :

Update the ddtgroupstatsmap by increasing the number of packets of the current DDT group.

➔ Packet “LOG_OBJECT_SWAP” :

Update the ddtgroupstatsmap by increasing the number of packets of the current DDT group.

➔ Packet “LOG_OBJECT_RESIZE” :

Update the ddtgroupstatsmap by increasing the number of packets of the current DDT group.

➔ Packet “LOG_OBJECT_GET” :

Update the ddtgroupstatsmap by increasing the number of packets of the current DDT group.
Update the ddtgrouppacketstatisticsmap by increasing the number of read packets of the current pair<ddtID, packetGroup>. Create a new map entry if necessary.

➔ Packet “LOG_OBJECT_INSERT” :

Update the ddtscope table by increasing the number of elements inserted in the DDT.

```
query << "select * from ddtscope where instanceID = "
      << c4_instance_id;
if (mysqlpp::StoreQueryResult res = query.store()) {
      if (!res.empty()) {
            ddtscope row = res[0];
            ddtscope orig_row = row;
            row.numAdds++;
            query.reset();
            query.update(orig_row, row);
            query.execute();
      }
} else {
      cerr << "Failed to get item list: " << query.error() << endl;
                              return 1;
```

```
}
```

Update the ddtstats vector by increasing the number of operations to the DDT.

```
ddtstats[currentInstanceID].numOperations++;
```

Update the ddtstats vector by increasing if necessary the number of max elements existing in the DDT.

```
ddtstats[currentInstanceID].currentObjects++;
if (ddtstats[currentInstanceID].maxObjects <
    ddtstats[currentInstanceID].currentObjects)
      ddtstats[currentInstanceID].maxObjects =
        ddtstats[currentInstanceID].currentObjects;
```

Update the ddtgroupstatsmap by increasing if necessary the number of max elements existing in the current DDT group.

```
ddtgroupstatsmap::iterator groupfound =
   ddtgroupstats.find(c4_type_num);
groupfound->second.currentObjects++;
if (groupfound->second.maxObjects < groupfound->second.currentObjects)
      groupfound->second.maxObjects =
        groupfound->second.currentObjects;
```

Update the ddtgroupstatsmap by increasing the number of packets of the current DDT group.
Update the ddtgrouppacketstatisticsmap by increasing the number of insert packets of the current pair<ddtID, packetGroup>. Create a new map entry if necessary.

➔ Packet "LOG_OBJECT_REMOVE" :

Update the ddtscope table by increasing the number of elements removed fromthe DDT.
Update the ddtstats vector by increasing the number of operations to the DDT.
Update the ddtgroupstatsmap by increasing the number of packets of the current DDT group.
Update the ddtgrouppacketstatisticsmap by increasing the number of remove packets of the current pair<ddtID, packetGroup>. Create a new map entry if necessary.

➔ Packet "LOG_OBJECT_CLEAR" :

Update the ddtscope table by increasing the number of times the DDT was cleared.
Update the ddtgroupstatsmap by increasing the number of packets of the current DDT group.
Update the ddtgrouppacketstatisticsmap by increasing the number of clear packets of the current pair<ddtID, packetGroup>. Create a new map entry if necessary.

➔ Packet "LOG_OBJECT_OPERATOR" :

Update the ddtgroupstatsmap by increasing the number of packets of the current DDT group.
Update the ddtgrouppacketstatisticsmap by increasing the number of operator packets of the current pair<ddtID, packetGroup>. Create a new map entry if necessary.

➔ Packet "LOG_OBJECT_END" :

Update the ddtgroupstatsmap by increasing the number of packets of the current DDT group.

➔ Packet "LOG_ITERATOR_NEXT" :

Update the ddtscope table by increasing the number of iterator operations in the DDT.
Update the ddtgroupstatsmap by increasing the number of packets of the current DDT group.
Update the ddtgrouppacketstatisticsmap by increasing the number of iterator packets of the current pair<ddtID, packetGroup>. Create a new map entry if necessary.

➔ Packet "LOG_ITERATOR_PREVIOUS" :

Update the ddtscope table by increasing the number of iterator operations in the DDT.
Update the ddtgroupstatsmap by increasing the number of packets of the current DDT group.
Update the ddtgrouppacketstatisticsmap by increasing the number of iterator packets of the current pair<ddtID, packetGroup>. Create a new map entry if necessary.

➔ Packet "LOG_ITERATOR_ADD" :

Update the ddtscope table by increasing the number of iterator operations in the DDT.
Update the ddtgroupstatsmap by increasing the number of packets of the current DDT group.
Update the ddtgrouppacketstatisticsmap by increasing the number of iterator packets of the current pair<ddtID, packetGroup>. Create a new map entry if necessary.

➔ Packet "LOG_ITERATOR_SUB" :

Update the ddtscope table by increasing the number of iterator operations in the DDT.
Update the ddtgroupstatsmap by increasing the number of packets of the current DDT group.
Update the ddtgrouppacketstatisticsmap by increasing the number of iterator packets of the current pair<ddtID, packetGroup>. Create a new map entry if necessary.

➔ Packet "LOG_ITERATOR_GET" :

Update the ddtscope table by increasing the number of iterator operations in the DDT.
Update the ddtgroupstatsmap by increasing the number of packets of the current DDT group.

Update the ddtgrouppacketstatisticsmap by increasing the number of iterator packets of the current pair<ddtID, packetGroup>. Create a new map entry if necessary.

➔ Packet “LOG_ITERATOR_RESET” :

Update the ddtscope table by increasing the number of iterator operations in the DDT.
Update the ddtgroupstatsmap by increasing the number of packets of the current DDT group.
Update the ddtgrouppacketstatisticsmap by increasing the number of iterator packets of the current pair<ddtID, packetGroup>. Create a new map entry if necessary.

➔ Packet “LOG_ITERATOR_ISDONE” :

Update the ddtscope table by increasing the number of iterator operations in the DDT.
Update the ddtgroupstatsmap by increasing the number of packets of the current DDT group.
Update the ddtgrouppacketstatisticsmap by increasing the number of iterator packets of the current pair<ddtID, packetGroup>. Create a new map entry if necessary.

11.2. Post processing phase

After the whole log is read and processed, the Analyzer can now store the information existing in its memory space (structured and organized in STL vector and data structures as described above) into the corresponding tables of the profiling database. In some cases, some more processing is required, since the information contained in some fields of some tables derives from information contained in other fields or even in other tables.
As an example, consider the ddtgroupstats structure (contained in ddtgroupstatsmap): The number stored in the field totalAccesses must be equal with the sum of all read and write operations of every ddt that belongs to this ddtgroup

```
for (ddtgroupstatsmap::iterator it = ddtgroupstats.begin(); it !=
     ddtgroupstats.end(); it++) {
      for (ddtstatisticsvector::iterator it2 = ddtstats.begin(); it2 !=
           ddtstats.end(); it2++) {
             if ((*it2).ddtID == it->first) {
                  it->second.totalAccesses += (*it2).numReads +
                  (*it2).numWrites;
                  it->second.numOperations += (*it2).numOperations;
                  it->second.totalAllocatedMem +=
                  (*it2).totalAllocatedMem;
            }
      }
      query.reset();
      ddtgroupstatistics row(
            it->first,
            it->second.totalAccesses,
            it->second.maxInstances,
            it->second.maxObjects,
            ((int) it->second.maxObjects/it->second.numOperations),
            it->second.maxAllocatedMem,
            it->second.totalAllocatedMem,
           it->second.numPackets
      );
      query.insert(row);
      query.execute();
}
```

12. Extracting Behaviour Information from the DB

Saving the profiling data in a MySQL Database gives us the flexibility to perform queries (simple or complex ones) in order to obtain further information about the client application, without having to modify the Analyzer module.

12.1. Adding the Type Name to ddtscope

Because of the great cost of saving the elements' type name of a ddt, we have created a separate table called ddttypes, where every typename is mapped to an integer value. We store this value in the ddtscope table instead of storing the whole type name. As it is shown below, it is very simple to retrieve this information with a query:

```
SELECT d.instanceID, d.scopeID, d.ddtID, d.typeID, t.typeName,
      d.duplicatedID, d.resizeHistoryID,d.numIteratorOperations,
      d.numAdds, d.numRemoves, d.numClears, d.maxElements,
      d.itIncrementCalls, d.itDecrementCalls, d.itAddCalls,
      d.itSubCalls, d.itGetCalls, d.itIsDoneCalls, d.itResetCalls,
      d.getCalls, d.constructCalls, d.destructCalls, d.copyCalls,
      d.swapCalls, d.resizeCalls
FROM ddtscope d
INNER JOIN ddttypes t
ON t.typeID = d.typeID;
```

12.2. Retrieving the total number of operations of each group

If we need to extend our query in order to retrieve the total number of operations (iterator operations, adds, removes etc) for each ddtgroup (specified by its ddtID) using again the ddtscope table, we simply have to a group by (ddtID) clause and the SUM() function:

```
SELECT d.ddtID, t.typeName, SUM(d.numIteratorOperations),
      SUM(d.numAdds), SUM(d.numRemoves),
      SUM(d.numClears), SUM(d.itIncrementCalls),
      SUM(d.itDecrementCalls), SUM(d.itAddCalls),
      SUM(d.itSubCalls), SUM(d.itGetCalls), SUM(d.itIsDoneCalls),
      SUM(d.itResetCalls), SUM(d.getCalls),
      SUM(d.constructCalls), SUM(d.destructCalls), SUM(d.copyCalls),
      SUM(d.swapCalls), SUM(d.resizeCalls)
FROM ddtscope d
INNER JOIN ddttypes t
ON t.typeID = d.typeID
GROUP BY d.ddtID, d.typeID
ORDER BY d.ddtID;
```

12.3. Retrieving the number malloc operations for each block size

The completelog table is the table where the Analyzer extracts the whole raw information found in the binary log file. We can retrieve the number of malloc operations for every block size by querying this database's table filtering the data with the correct (malloc_end) packets (as defined in logger module), and grouping the number of them by the block size:

```
SELECT COUNT(c.logID)
FROM completelog c
WHERE c.packetID = 5
GROUP BY c.size;
```

12.4. Retrieving the number insert operations for each ddt instance

We can retrieve the number of insert operations for every ddt instance by querying this database's table filtering the data with the correct (object_insert) packets, and grouping the number of them by the block size:

```
SELECT COUNT(c.logID)
FROM completelog c
WHERE c.packetID = 36
GROUP BY c.instanceID;
```

12.5. Retrieving the number read operations for block address

We can retrieve the number of read operations for every block by querying this database's table filtering the data with the correct (var_read) packets, and grouping the number of them by the block address:

```
SELECT COUNT(c.logID)
FROM completelog c
WHERE c.packetID = 0
GROUP BY c.address;
```

ΒΙΒΛΙΟΓΡΑΦΙΑ

[1] L. Benini and G. De Micheli. System level power optimization techniques and tools. In ACM TODAES, April 2000.

[2] P. R. Panda, et al. Data and memory optimizations for embedded systems. ACM TODAES, 6(2):142– 206, April 2001.

[3] F. Catthoor et al. Data access and storage management for embedded programmable processors. Kluwer Academic Publishers, 2002.

[4] S. L. Graham, et al. gprof: a call graph execution profiler. In ACM SIGPLAN Symposium on Compiler Construction, pages 120–126, 1982.

[5] N. Nethercote. Dynamic Binary Analysis and Instrumentation or Building Tools is Easy. PhD thesis, Trinity College, Cambridge, England, 2004.

[6] CXperf, A profiling tool from Hewlett & Packard, http://www.hp.com/esy/lang/tools/Performance.

[7] Srivastava, A. and Eustace, A. 1994. ATOM: a system for building customized program analysis tools. In Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (Orlando, Florida, United States, June 20 - 24, 1994). PLDI '94. ACM, New York, NY, 196-205. DOI= http://doi.acm.org/10.1145/178243.178260

[8] Peri, R.V. and Jinturkar, S. and Fajardo, L., “A Novel Technique for Profiling Programs in Embedded Systems,” in Proceedings of ACM Workshop on Feedback-Directed and Dynamic Optimization, 1999.

[9] http://vdrift.net/

[10] Atienza, D., Baloukas C., et al. 2007. Optimization of dynamic data structures in multimedia embedded systems using evolutionary computation. In Proceedings of the 10th international Workshop on Software &Amp; Compilers For Embedded Systems (Nice, France, April 20 - 20, 2007).

[11] C.Poucet, D.Atienza, and F.Catthoor. Template-based semi-automatic profiling of multimedia applications. In Proceedings of the International Conference on Multi-media and Expo (ICME 2006), pages 1061–1064, Toronto, Canada, July 2006. IEEE Computer, IEEE Signal Processing, IEEE System and IEEE Communications Society.

[12] Cormen H. Thomas, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, Introduction to Algorithms, Second Edition, MIT Press, 2001, ISBN: 0262032937

[13] Baloukas c. Master Thesis: “Development of Methodology for the Optimum Dynamic Access of Data in Network Applications”, Democritus university of Thrace, Thrace, 2007