ΕΘΝΙΚΟ ΜΕΤΣΟΒΕΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧ. ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
Τομέας Τεχνολογίας Υπολογιστών και Υπολογιστών
Εργαστήριο Μικροϋπολογιστών και Ψηφιακών Συστημάτων

Σχεδιασμός Ενσωματωμένων Συστημάτων
9^{Ο} Εξάμηνο ΗΜΜΥ

2η ΕΡΓΑΣΤΗΡΙΑΚΗ ΑΣΚΗΣΗ

Ασκήσεις στη Βελτιστοποίηση Δυναμικών Δομών Δεδομένων (Dynamic Data Type Refinement – DDTR)

Ημ/νια Παράδοσης : 24/11/2024
Ομάδα 22
Παναγιώτης Μπέλσης AM : 03120874
Θεοδώρα Εξάρχου AM : 03120865

Άσκηση 1: Βελτιστοποίηση δυναμικών δομών δεδομένων του αλγορίθμου DRR Ο source code του DRR έχει ήδη περασμένη την βιβλιοθήκη DDTR.

a) Εκτελέσαμε την εφαρμογή 9 φορες για όλους τους διαφορετικούς συνδυασμούς υλοποιήσεων δομών δεδομένων για τη λίστα των πακέτων και τη λίστα των κόμβων . Για κάθε συνδυασμό καταγράψαμε τα αποτελέσματά του σχετικά με τον αριθμό των προσβάσεων στη μνήμη (memory accesses) και το μέγεθος της απαιτούμενης μνήμης (memory footprint). Παρακάτω παραθέτουμε τα βήματα που ακολουθήσαμε :

Αφαιρεθηκαν τα // για τα define αναλογα ποιον συνδυασμο θελαμε να υλοποιήσουμε στο αρχειο drr.h

```
#define SLL_CL
//#define DLL_CL
//#define DYN_ARR_CL

#define SLL_PK
//#define DLL_PK
//#define DYN_ARR_PK
```

Compilation της εφαρμογής DRR:

```
gcc drr.c -o drr -pthread -lcdsl -no-pie -L./../synch_implementations
-I./../synch_implementations
```

Δημιουργία ενός trace file των προσβάσεων στη μνήμη με χρήση του εργαλείου lackey:

```
valgrind --log-file="mem_accesses_log.txt" --tool=lackey --trace-mem=yes ./drr
```

Καταμέτρηση του αριθμού των προσβάσεων στη μνήμη από το trace file:

```
 grep -c 'I\| L' mem_accesses_log.txt
```

Αποτελέσματα μέγιστης χρήσης μνήμης (Memory footprint):
Δημιουργία ενός log αρχείου και επεξεργασία του, με χρήση του εργαλείου massif:

```
valgrind --tool=massif ./drr
```

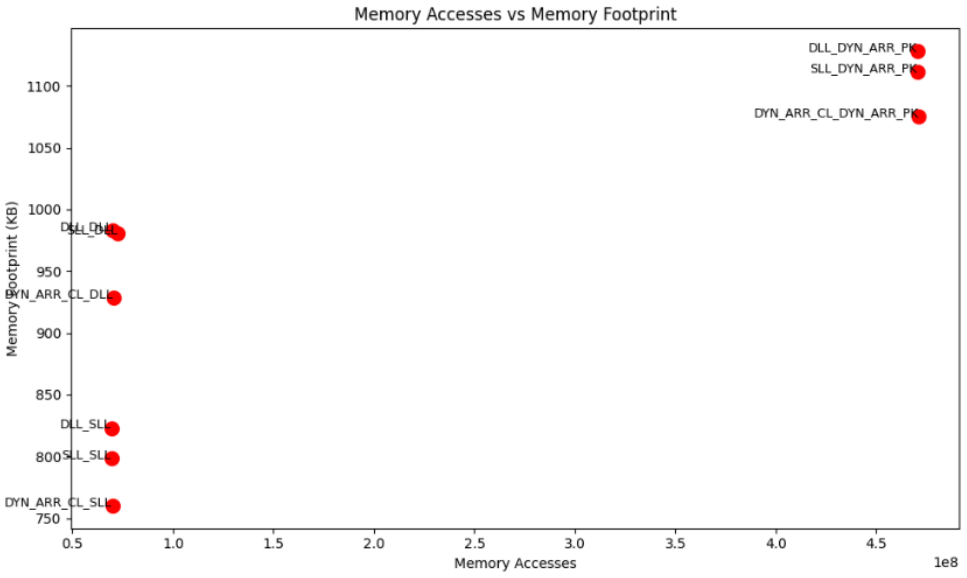
```
ms_print massif.out.XXXXX > mem_footprint_log.txt
```

clientList	pList	Memory Footprint	Memory Accesses
SLL	SLL	798.8 KB	69,304,403
SLL	DLL	980.3 KB	72,535,244
SLL	DYN_ARR_PK	1.111 MB	470,424,672
DLL	SLL	823.0 KB	69,317,067
DLL	DLL	983.3 KB	69,987,638
DLL	DYN_ARR_PK	1.128 MB	470,439,259
DYN_ARR_CL	SLL	760.2 KB	69,838,260
DYN_ARR_CL	DLL	928.5 KB	70,516,113
DYN_ARR_CL	DYN_ARR_PK	1.075 MB	471,147,444

Παρατηρήσαμε ότι όταν τρέχαμε κάτι με DYN_ARR έπαιρνε αρκετή παραπάνω ώρα αυτή η εντολή:
`valgrind --log-file="mem_accesses_log.txt" --tool=lackey --trace-mem=yes ./drr`

Οι Pareto optimal λύσεις που προέκυψαν είναι :

clientList	pList	Memory Footprint	Memory Accesses
SLL	SLL	798.8 KB	69,304,403
DYN_ARR_CL	SLL	760.2 KB	69,838,260



Στην άσκηση αυτή χρησιμοποιήσαμε μια Απλά Συνδεδεμένη Λίστα - Single Linked List (SLL), μια Διπλά Συνδεδεμένη Λίστα - Double Linked List (DLL) και έναν Δυναμικό Πίνακα - Dynamic Array (DYN_ARR) με σκοπο να βελτιστοποιηθούν οι δυναμικές δομές δεδομένων του Deficit Round Robin (DRR) με βάση τα αποτελέσματά μας, εξαγάγαμε τα εξής συμπεράσματα:

Memory Footprint

clientList \ pList	SLL	DLL	Dynamic Array
SLL	798.8 KB	980.3 KB	1.111 MB
DLL	823.0 KB	983.3 KB	1.128 MB
Dynamic Array	760.2 KB	928.5 KB	1.075 MB

Παρατηρούμε, με βάση τα αποτελέσματά μας για τους 9 συνδυασμούς σε κάθε περίπτωση δομής client list, καθώς μεταβαίνουμε από τη SLL στη DLL στη δομή packet list, το memory footprint αυξάνεται. Η DLL απαιτεί επιπλέον μνήμη, επειδή κάθε κόμβος αποθηκεύει δύο δείκτες (προς τον προηγούμενο και τον επόμενο κόμβο), ενώ η SLL αποθηκεύει μόνο έναν δείκτη. Αυτό εξηγεί την αύξηση του memory footprint όταν χρησιμοποιούμε τη DLL αντί για τη SLL. Ωστόσο, όταν γίνεται χρήση της DYN_ARR, παρατηρούμε μια μικρή μείωση στο memory footprint. Ο Dynamic Array τείνει να χρησιμοποιεί λιγότερη μνήμη ανά στοιχείο, καθώς δεν απαιτεί δείκτες όπως στις συνδεδεμένες λίστες. Αυτό έχει ως αποτέλεσμα μια μικρή μείωση στο memory footprint σε σύγκριση με τις SLL και DLL.

Memory Accesses

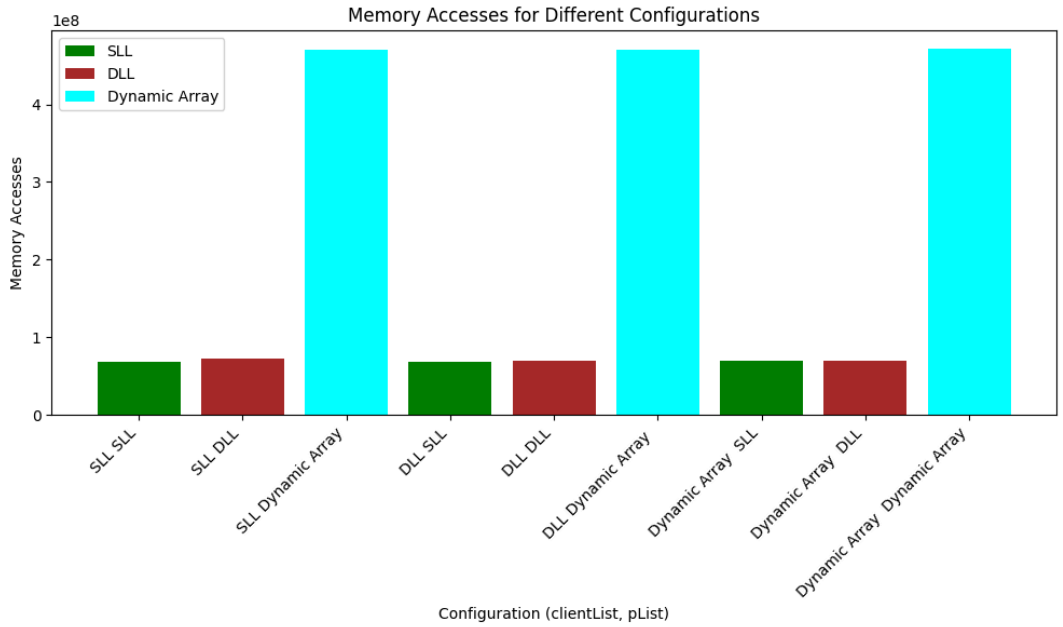
clientList \ pList	SLL	DLL	Dynamic Array
SLL	69304403	72535244	470424672
DLL	69317067	69987638	470439259
Dynamic Array	69838260	70516113	471147444

Παρατηρούμε ότι για τις προσβάσεις στη μνήμη τα δεδομένα είναι παρόμοια για κάθε διαφορετική δομή δεδομένων της packet list, με μικρές αποκλίσεις. Αυτό συμβαίνει επειδή και οι δύο λίστες βασίζονται σε δείκτες για τη διασύνδεση των κόμβων. Ωστόσο, όταν γίνεται χρήση του Dynamic Array για την packet list, οι προσβάσεις στη μνήμη αυξάνονται σημαντικά συγκριτικά με τη χρήση των SLL και DLL. Ο Δυναμικός Πίνακας απαιτεί περισσότερες προσβάσεις στη μνήμη, κυρίως λόγω της διαχείρισης των επανακατανομών (resizing) όταν ο πίνακας γεμίζει. Έτσι εξηγείται και ο παραπάνω χρόνος εκτέλεσης που προέκυψε με τη χρήση Dynamic Array.

b) Ο συνδυασμός υλοποιήσεων δομών δεδομένων με την οποία η εφαρμογή έχει τον μικρότερο αριθμό προσβάσεων στη μνήμη (minimum number of memory accesses) ειναι :

clientList → SLL (Single Linked List)
pList → SLL (Single Linked List)

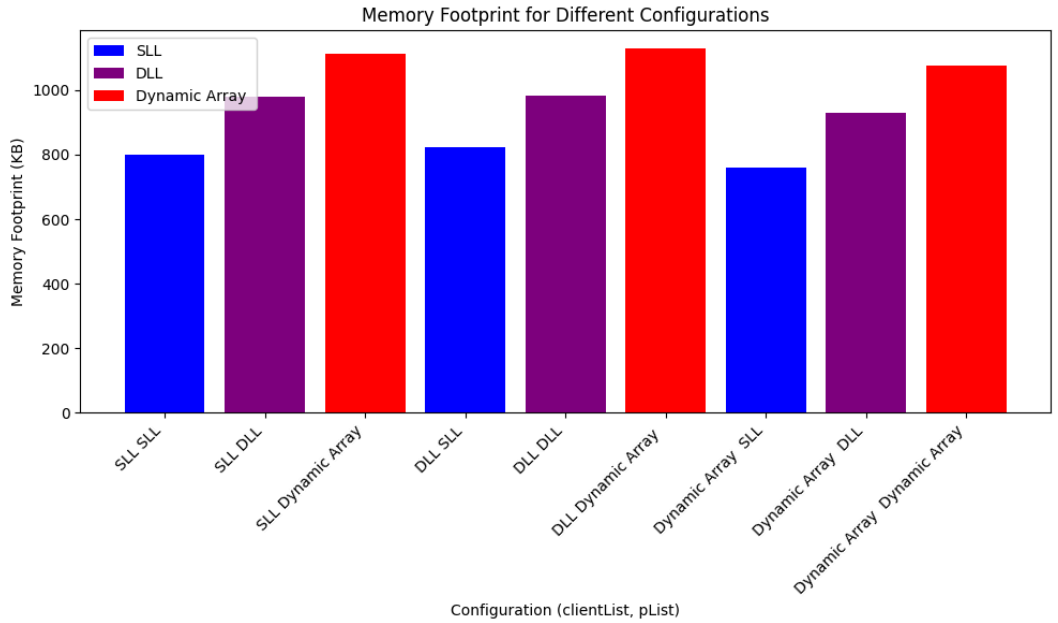
Με **Memory Acceses = 69304403**



c) Ο συνδυασμός υλοποιήσεων δομών δεδομένων με την οποία η εφαρμογή έχει μικρότερες απαιτήσεις σε μνήμη (smaller memory footprint) ειναι :

clientList → DYN_ARR_CL (Dynamic Array)
pList → SLL (Single Linked List)

Με **Memory Footprint = 760.2 KB**



Άσκηση 2: Βελτιστοποίηση δυναμικών δομών δεδομένων του αλγορίθμου Dijkstra

a) Τα αποτελέσματα που παράγει η εφαρμογή κατά την εκτέλεση είναι :

```
panosb@deskpanosb:~/workspace/src/dijsktra$ ./a.out input.dat
Shortest path is 1 in cost. Path is:  0 41 45 51 50
Shortest path is 0 in cost. Path is:  1 58 57 20 40 17 65 73 36 46 10 38 41 45 51
Shortest path is 1 in cost. Path is:  2 71 47 79 23 77 1 58 57 20 40 17 52
Shortest path is 2 in cost. Path is:  3 53
Shortest path is 1 in cost. Path is:  4 85 83 58 33 13 19 79 23 77 1 54
Shortest path is 3 in cost. Path is:  5 26 23 77 1 58 99 3 21 70 55
Shortest path is 3 in cost. Path is:  6 42 80 77 1 58 99 3 21 70 55 56
Shortest path is 0 in cost. Path is:  7 17 65 73 36 46 10 58 57
Shortest path is 0 in cost. Path is:  8 37 63 72 46 10 58
Shortest path is 1 in cost. Path is:  9 33 13 19 79 23 77 1 59
Shortest path is 0 in cost. Path is:  10 60
Shortest path is 5 in cost. Path is:  11 22 20 40 17 65 73 36 46 10 29 61
Shortest path is 0 in cost. Path is:  12 37 63 72 46 10 58 99 3 21 70 62
Shortest path is 0 in cost. Path is:  13 19 79 23 77 1 58 99 3 21 70 55 12 37 63
Shortest path is 1 in cost. Path is:  14 38 41 45 51 68 2 71 47 79 23 77 1 58 33 13 92 64
Shortest path is 1 in cost. Path is:  15 13 92 94 11 22 20 40 17 65
Shortest path is 3 in cost. Path is:  16 41 45 51 68 2 71 47 79 23 77 1 58 33 32 66
Shortest path is 0 in cost. Path is:  17 65 73 36 46 10 58 33 13 19 79 23 91 67
Shortest path is 1 in cost. Path is:  18 15 41 45 51 68
Shortest path is 2 in cost. Path is:  19 69
```

b)Στο δοθέν κώδικα, η υλοποίηση της ουράς είναι πιο στατική και χειροκίνητη, οι συνδέσεις χειρίζονται με pointers, ενώ εμείς θέλουμε η υλοποίηση της ουράς να είναι γενικευμένη και ευέλικτη, έτσι ώστε να επιτρέπεται η χρήση διαφορετικών τύπων ουρών (συνδεδεμένη λίστα, διπλά συνδεδεμένη λίστα, δυναμικός πίνακας) χωρίς να αλλάζει ο κώδικας της συνάρτησης. Για να το πετύχουμε αυτό, κάναμε τα εξής:

Κάναμε include τα 3 header files .Σε αυτά τα 3 αρχεία βρίσκονται οι δηλώσεις των συναρτήσεων (function declarations) που μπορούν να χρησιμοποιηθούν στον κώδικα της εφαρμογής για να αντικαταστήσουν τη δομή δεδομένων της εφαρμογής με αυτές της βιβλιοθήκης DDTR.

```
#include <stdlib.h>
#define SLL
//#define DLL
//#define DYN_ARR
```

```
#if defined(SLL)
#include "../synch_implementations/cdsl_queue.h"
#endif
#if defined(DLL)
#include "../synch_implementations/cdsl_deque.h"
#endif
#if defined(DYN_ARR)
#include "../synch_implementations/cdsl_dyn_array.h"
#endif
```

Αντικαθιστούμε τις δηλώσεις (definitions – declarations) των δομών δεδομένων της εφαρμογής με αυτά της βιβλιοθήκης DDTR.
Αντικαθιστουμε το `QITEM *qHead = NULL;`

```
#if defined(SLL)
cdsl_sll *qHead;
iterator_cdsl_sll it;
#endif
#if defined(DLL)
cdsl_dll *qHead;
iterator_cdsl_dll it;
#endif
#if defined(DYN_ARR)
cdsl_dyn_array *qHead;
iterator_cdsl_dyn_array it;
#endif
```

Ενώ ταυτόχρονα δηλώνουμε και έναν **Iterator** `it`, ο οποίος θα χρειαστεί έτσι ώστε να υλοποιήσουμε τη νέα συνάρτηση dequeue.
Στην main αρχικοποιούμε τις δομές δεδομένων

```
 #if defined(SLL)
        qHead = cdsl_sll_init();
    #endif
    #if defined(DLL)
        qHead = cdsl_dll_init();
    #endif
    #if defined(DYN_ARR)
        qHead = cdsl_dyn_array_init();
    #endif
```

<u>Για την συναρτηση enqueue</u>
Αρχικά, ενσωματώνουμε τη μέθοδο **enqueue** του qHead, έτσι ώστε να γίνεται αυτόματα η προσθήκη νέου στοιχείου στη ουρά ανεξαρτήτως της δομής δεδομένων που χρησιμοποιούμε, ενώ στη προηγούμενη εκδοχή αυτό γινόταν χειροκίνητα.
`qHead->enqueue(0, qHead, (void *)qNew);`

```
void enqueue (int iNode, int iDist, int iPrev)
{
    QITEM *qNew = (QITEM *) malloc(sizeof(QITEM));
    if (!qNew)
    {
        fprintf(stderr, "Out of memory.\n");
        exit(1);
    }
    qNew->iNode = iNode;
    qNew->iDist = iDist;
    qNew->iPrev = iPrev;
    qHead->enqueue(0, qHead, (void *)qNew);
    g_qCount++;}
```

Η συνάρτηση enqueue ΠΡΙΝ → ΜΕΤΑ.

```
void enqueue (int iNode, int iDist, int iPrev)
{
  QITEM *qNew = (QITEM *) malloc(sizeof(QITEM));
  QITEM *qLast = qHead;

  if (!qNew)
    {
      fprintf(stderr, "Out of memory.\n");
      exit(1);
    }
  qNew->iNode = iNode;
  qNew->iDist = iDist;
  qNew->iPrev = iPrev;
  qNew->qNext = NULL;

  if (!qLast)
    {
      qHead = qNew;
    }
  else
    {
      while (qLast->qNext) qLast = qLast->qNext; //Αν η λίστ
      qLast->qNext = qNew;
    }
  g_qCount++;

}
```

```
void enqueue (int iNode, int iDist, int iPrev)
{
    QITEM *qNew = (QITEM *) malloc(sizeof(QITEM));

    if (!qNew)
    {
        fprintf(stderr, "Out of memory.\n");
        exit(1);
    }
    qNew->iNode = iNode;
    qNew->iDist = iDist;
    qNew->iPrev = iPrev;

    qHead->enqueue(0, qHead, (void *)qNew);
    g_qCount++;

}
```

Για τη συναρτήτηση dequeue

Ο δοθέν κώδικας (αριστερά screenshot) υλοποιεί μια συναρτήτηση με χειρόκινη διαχείριση ουράς όπως και για το enqueue. Κάναμε την εξής υλοποίηση με βάση και την αναλυτική επεξήγηση που δόθηκε στη εκφώνηση για το DRR.

Αλλάξαμε τη μέθοδο **dequeue** χρησιμοποιώντας **iterator** για την αφαίρεση του στοιχείου.
Ουσιαστικά, δηλώσαμε έναν **iterator** `it` για την επεξεργασία της ουράς, αντί να διαχειριζόμαστε χειροκίνητα τη λίστα με δείκτες, χρησιμοποιούμε τις μεθόδους `iter_begin` και `iter_deref`.

- Η μέθοδος `iter_begin(qHead)` δημιουργεί έναν iterator `it` που τοποθετείται στην αρχή της ουράς. Αυτό είναι το αντίστοιχο της πρώτης εκδοχής, όπου άμεσα παίρναμε το πρώτο στοιχείο μέσω του qHead.
- Η μέθοδος `iter_deref(qHead, it)` ανακτά το στοιχείο που δείχνει ο iterator. Η αλλαγή αυτή γίνεται καθώς αν συνεχίζαμε να χρησιμοποιούμε το qHead για να διαχειρίζεται τα δεδομένα, θα έπρεπε να τροποποιούμε τον κώδικα για κάθε αλλαγή στη δομή δεδομένων.
- Τέλος η μνήμη του στοιχείου αποδεσμεύεται με την χρηση `free (item)`.

Στην συνάρτηση enqueue, δεν χρειάζεται να χρησιμοποιηθούν iterators με τον ίδιο τρόπο όπως στη dequeue, επειδή η διαδικασία προσθήκης ενός στοιχείου στην ουρά είναι πιο απλή από την αφαίρεση. Όταν προσθέτουμε ένα στοιχείο στην ουρά (μέσω της συνάρτησης enqueue), δεν χρειάζεται να διατρέξουμε όλη την ουρά ή να αλληλεπιδράσουμε με τα ήδη υπάρχοντα στοιχεία με συγκεκριμένο τρόπο όπως στην αφαίρεση.

```
void dequeue (int *piNode, int *piDist, int *piPrev)
{
    it = qHead->iter_begin(qHead);
    QITEM *item = (QITEM*)(qHead->iter_deref(qHead, it));

    *piNode = item->iNode;
    *piDist = item->iDist;
    *piPrev = item->iPrev;
    qHead->dequeue(0, (qHead));
    g_qCount--;
    free(item);
}
```

Η συνάρτηση dequeue ΠΡΙΝ → ΜΕΤΑ.

```
void dequeue (int *piNode, int *piDist, int *piPrev)
{
  QITEM *qKill = qHead;

  if (qHead)
    {

      *piNode = qHead->iNode;
      *piDist = qHead->iDist;
      *piPrev = qHead->iPrev;
      qHead = qHead->qNext;
      free(qKill);
      g_qCount--;
    }
}
```

```
void dequeue (int *piNode, int *piDist, int *piPrev)
{
    it = qHead->iter_begin(qHead);
    QITEM *item = (QITEM*)(qHead->iter_deref(qHead, it));

    *piNode = item->iNode;
    *piDist = item->iDist;
    *piPrev = item->iPrev;
    qHead->dequeue(0, (qHead));
    g_qCount--;
    free(item);
}
```

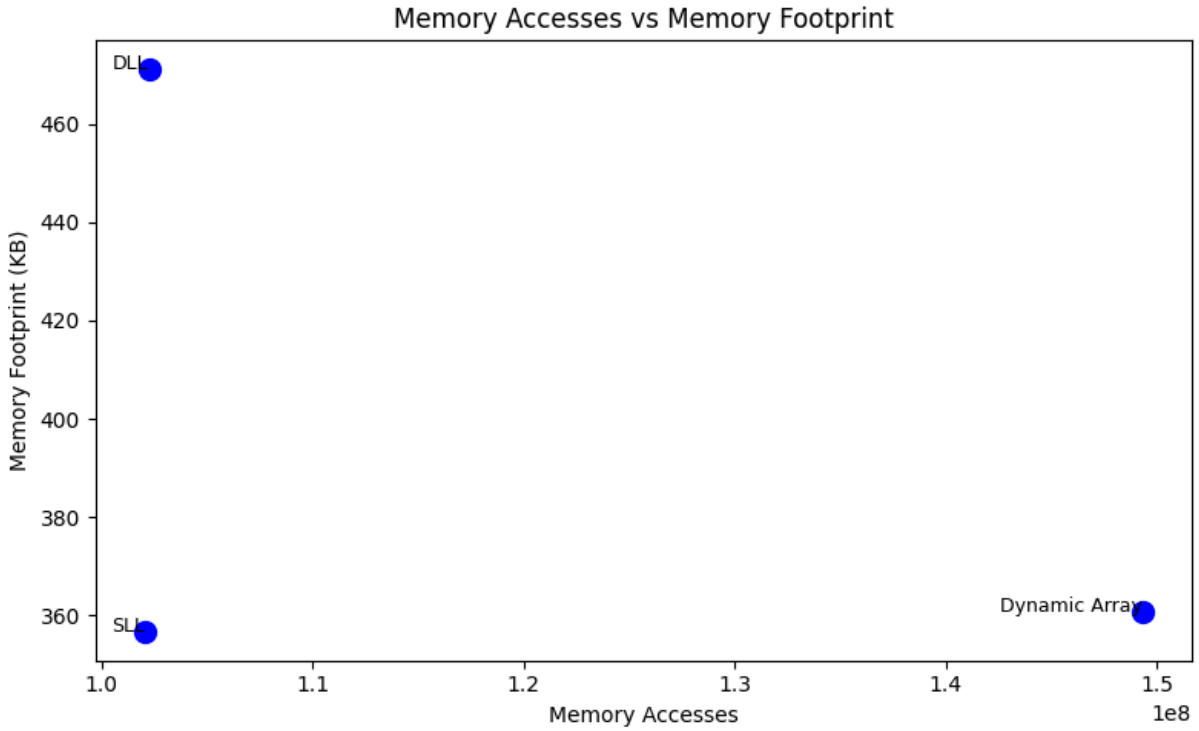
Compilation της εφαρμογής dijkstra:

```
 gcc dijkstra.c -o dijkstra -pthread -lcdsl -no-pie -L./../synch_implementations
-I./../synch_implementations
```

Παρατηρούμε ότι τα αποτελέσματα είναι ίδια με το ερώτημα **α** (όπως θα έπρεπε). Συνεπώς, ο κώδικας τρέχει σωστά τον αλγόριθμο του dijkstra.

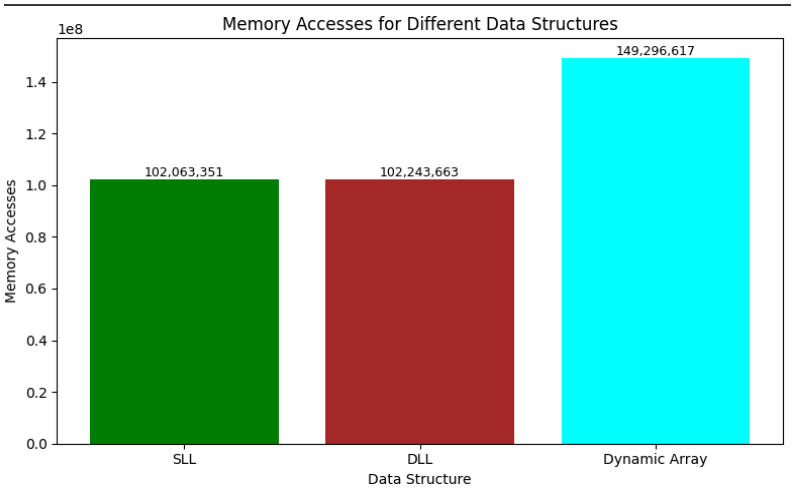
```
panosb@deskpanosb:~/workspace/src/dijsktra$ ./dijkstrasol input.dat
Shortest path is 1 in cost. Path is:  0 41 45 51 50
Shortest path is 0 in cost. Path is:  1 58 57 20 40 17 65 73 36 46 10 38 41 45 51
Shortest path is 1 in cost. Path is:  2 71 47 79 23 77 1 58 57 20 40 17 52
Shortest path is 2 in cost. Path is:  3 53
Shortest path is 1 in cost. Path is:  4 85 83 58 33 13 19 79 23 77 1 54
Shortest path is 3 in cost. Path is:  5 26 23 77 1 58 99 3 21 70 55
Shortest path is 3 in cost. Path is:  6 42 80 77 1 58 99 3 21 70 55 56
Shortest path is 0 in cost. Path is:  7 17 65 73 36 46 10 58 57
Shortest path is 0 in cost. Path is:  8 37 63 72 46 10 58
Shortest path is 1 in cost. Path is:  9 33 13 19 79 23 77 1 59
Shortest path is 0 in cost. Path is:  10 60
Shortest path is 5 in cost. Path is:  11 22 20 40 17 65 73 36 46 10 29 61
Shortest path is 0 in cost. Path is:  12 37 63 72 46 10 58 99 3 21 70 62
Shortest path is 0 in cost. Path is:  13 19 79 23 77 1 58 99 3 21 70 55 12 37 63
Shortest path is 1 in cost. Path is:  14 38 41 45 51 68 2 71 47 79 23 77 1 58 33 13 92 64
Shortest path is 1 in cost. Path is:  15 13 92 94 11 22 20 40 17 65
Shortest path is 3 in cost. Path is:  16 41 45 51 68 2 71 47 79 23 77 1 58 33 32 66
Shortest path is 0 in cost. Path is:  17 65 73 36 46 10 58 33 13 19 79 23 91 67
Shortest path is 1 in cost. Path is:  18 15 41 45 51 68
Shortest path is 2 in cost. Path is:  19 69
```

c) Για τις εξής δυναμικές δομές δεδομένων:
Απλά Συνδεδεμένη Λίστα – Single Linked List (SLL), Διπλά Συνδεδεμένη Λίστα – Double Linked List (DLL) και Δυναμικό Πίνακα – Dynamic Array (DYN_ARR). Καταγράψαμε τα αποτελέσματα του αριθμού προσβάσεων στη μνήμη και του μέγιστου μεγέθους μνήμης που απαιτούνται για κάθε υλοποίηση. :



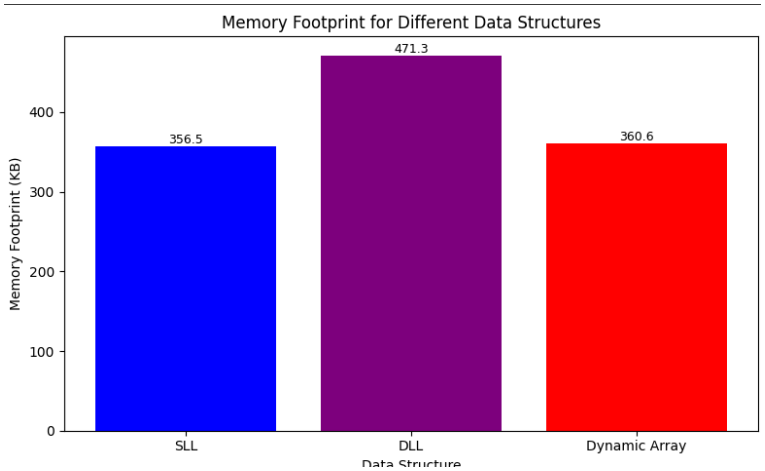
Data type	Memory Accesses	Memory Footprint
SLL	102063351	356.5 KB
DLL	102243663	471.3 KB
Dynamic Array	149296617	360.6 KB

d) Η υλοποιήση δομής δεδομένων με την οποία η εφαρμογή έχει τον μικρότερο αριθμό προσβάσων στη μνήμη (lowest number of memory accesses) είναι η **SLL.**



Data type	Memory Accesses
SLL	**102063351**
DLL	102243663
Dynamic Array	149296617

e) Η υλοποίηση δομής δεδομένων με την οποία η εφαρμογή έχει μικρότερες απαιτήσεις σε μνήμη (lowest memory footprint) είναι και πάλι η **SLL.**



Data type	Memory Footprint
SLL	**356.5 KB**
DLL	471.3 KB
Dynamic Array	360.6 KB

Για τον αλγόριθμο **Dijkstra** με βάση τα δεδομένα, είναι σαφές ότι η απλά συνδεδεμένη λίστα (SLL) υπερέχει σε απόδοση και στους δύο παράγοντες (memory accesses και memory footprint). Συγκεκριμένα, η SLL απαιτεί σημαντικά λιγότερες προσβάσεις μνήμης, ενώ καταλαμβάνει και το μικρότερο μέγεθος μνήμης, γεγονός που την καθιστά την πιο αποδοτική επιλογή.

- Η **DLL** υπερέχει στις προσβάσεις μνήμης, καθώς η εισαγωγή και η διαγραφή στοιχείων απαιτούν μόνο ενημερώσεις δεικτών, αλλά έχει μεγαλύτερο αποτύπωμα μνήμης λόγω των δύο δεικτών ανά κόμβο.
- Η **Dynamic Array** έχει μικρότερο αποτύπωμα μνήμης, καθώς αποθηκεύει μόνο τις τιμές σε συνεχόμενη μνήμη, αλλά απαιτεί περισσότερες προσβάσεις.

Συμπέρασμα :
Από την ανάλυση και τη σύγκριση των δομών δεδομένων στις δύο ασκήσεις (DRR και Dijkstra), καταλήγουμε ότι η επιλογή της κατάλληλης δομής δεδομένων εξαρτάται από τις απαιτήσεις και τις ιδιαιτερότητες της εκάστοτε εφαρμογής. Δεν υπάρχει μία σταθερά σωστή λύση που να εφαρμόζεται σε κάθε περίπτωση. Αντίθετα, η αποτελεσματικότητα μιας δομής δεδομένων καθορίζεται από το πώς αυτή ανταποκρίνεται στις συγκεκριμένες ανάγκες, όπως ο αριθμός των προσβάσεων στη μνήμη, το μέγεθος της μνήμης που απαιτείται και οι λειτουργίες που εκτελούνται.

Το παρακάτω screenshot είναι το memory footprint για το SLL (βέλτιστη υλοποίηση).