Σχεδιασμός Ενσωματωμένων Συστημάτων
9ο εξάμηνο ΣΗΜΜΥ
2η εργαστηριακή άσκηση

Ομάδα 6:
Δούκας Θωμάς Α.Μ.: 03116081
Ψαρρας Ιωάννης Α.Μ.:03116033

Άσκηση 1: Βελτιστοποίηση δυναμικών δομών δεδομένων του αλγορίθμου DRR

Στο συγκεκριμένο ερώτημα θα χρησιμοποιήσουμε τη μεθοδολογία *Βελτιστοποίησης Δυναμικών Δομών Δεδομένων - Dynamic Data Type Refinement (DDTR)* με σκοπό να βελτιστοποιήσουμε τις δυναμικές δομές δεδομένων του αλγορίθμου δρομολόγησης πακέτων Deficit Round Robin. Συγκεκριμένα, θα δοκιμάσουμε τη χρήση 3 διαφορετικών δομών στη λίστα δεδομένων που χρησιμοποιεί ο αλγόριθμος για τα πακέτα και τους κόμβους. Οι υπό εξέταση δομές είναι οι Απλά Συνδεδεμένη Λίστα (SLL), Διπλά Συνδεδεμένη Λίστα (DLL) και Δυναμικός Πίνακας (DYN_ARR), για τους συνδυασμούς των οποίων θα προσδιοριστούν το μέγεθος της μνήμης που απαιτείται για την εκτέλεση του αλγορίθμου αλλά και το συνολικό πλήθος προσβάσεων σε αυτή.

a) Μετρήσεις για συνδυασμούς δομών

Στο αρχείο *drr.h* που μας δόθηκε χρησιμοποιούμε τις εντολές:

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

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

για την εισαγωγή των κατάλληλων συναρτήσεων που μπορούν να χρησιμοποιηθούν στην αντικατάσταση της αρχικής δομής δεδομένων που χρησιμοποιείται στον πίνακα.

Για την αυτοματοποίηση της διαδικασία συντάσσουμε το παρακάτω script το οποίο εκτελείται κάθε φορά για τις αντίστοιχες δομές δεδομένων. Εισάγωντας μόνο τις εντολές *#define SLL_CL, #define SLL_PK* στο αρχείο μέσω της εντολής ./execute.sh sll_sll και εκτελείται compile του κώδικα *drr.c* και παράγονται στο φάκελο results οι μετρήσεις που προκύπτουν από το εργαλείο valgrind.

Αναφέρουμε πως από τα αρχεία που παράγονται κρατάμε μόνο τις απαραίτητες πληροφορίες, διαγράφοντας τα αντίστοιχα logs. Συγκεκριμένα, οι προσβάσεις στη μνήμη προσδιορίζονται μέσω της εντολής *cat mem_accesses_log.txt | grep 'I\L' | wc -l* στο αρχείο που παράγεται από το εργαλείο lackey, ενώ το memory footprint εμφανίζεται σε συγκεκριμένες σειρές στο αρχείο εξόδου του εργαλείου massif. Οι παραπάνω μετασχηματισμοί φαίνονται στο script που ακολουθεί.

execute.sh

```
#!/bin/sh

if [ ! $# -eq 1 ]
then
    echo "Usage: ./execute <data_structures_combination>"
    echo "sll_sll, sll_dll, sll_dynarr, dll_sll, dll_sll, dll_dynarr, dynarr_sll, dynarr_dll, dynarr_dynarr"
else
    #Compile
    gcc drr.c -o drr_$1 -pthread -lcdsl -L./../synch_implementations -I./../synch_implementations

    #Memory access
    valgrind --log-file="$1_accesses_log.txt" --tool=lackey --trace-mem=yes ./drr_$1
    cat $1_accesses_log.txt | grep 'I\| L' | wc -l >> ./results/$1_accesses.txt
    rm $1_accesses_log.txt

    #Memory footprint
    valgrind --tool=massif --massif-out-file="$1_massif.out" ./drr_$1
    ms_print $1_massif.out > $1_footprint_log.txt
    cat $1_footprint_log.txt | sed '8!d' >> ./results/$1_footprint.txt
    cat $1_footprint_log.txt | sed '9!d' >> ./results/$1_footprint.txt
    rm $1_massif.out $1_footprint_log.txt
fi
```

Ακολούθως εμφανίζονται τα αποτελέσματα των εκτελέσεων, όπως καταγράφονται στα αρχεία <CL_dt>_<DL_dt>_accesses.txt και <CL_dt>_<DL_dt>_footprint.txt.

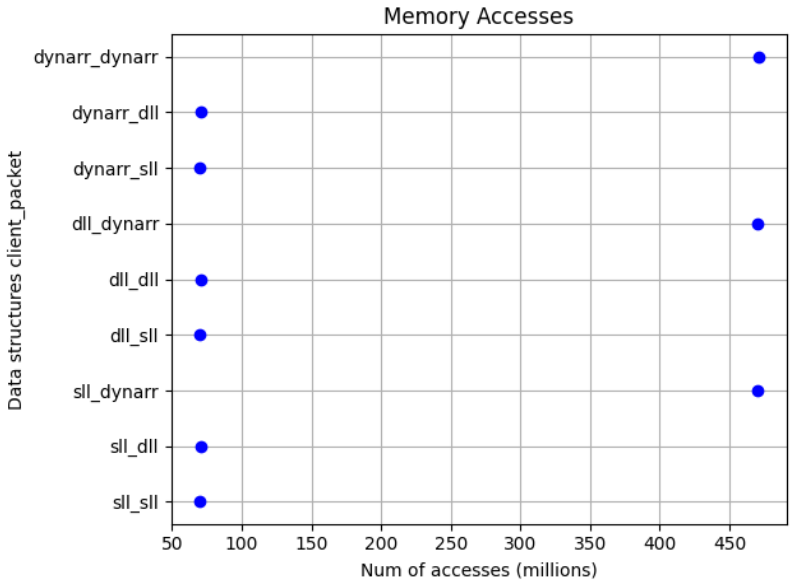
Data Types	Accesses	Footprint
sll_sll	69562215	798.8KB
sll_dll	70244400	980.3KB
sll_dynar	470481920	1.111MB
dll_sll	69574886	823.0KB
dll_dll	70251931	983.3KB
dll_dynar	470491921	1.132MB
dynar_sll	70092935	760.2KB
dynar_dll	70777255	928.5KB
dynar_dynar	471187690	1.075MB

Με σκοπό την οπτικοποίηση των αποτελεσμάτων που παρουσιάστηκαν και για να είναι ευκολότερη η διαδικασία επιλογής των βέλτιστων συνδυασμών στα επόμενα ερωτήματα υλοποιήσαμε το πρόγραμμα *plot.py*. Η εκτέλεση του κώδικα που ακολουθεί πραγματοποιείται μετά την λήψη μετρήσεων για όλες τις περιπτώσεις. Τα διαγράμματα παρατίθενται στα αντίστοιχα ερωτήματα b,c.

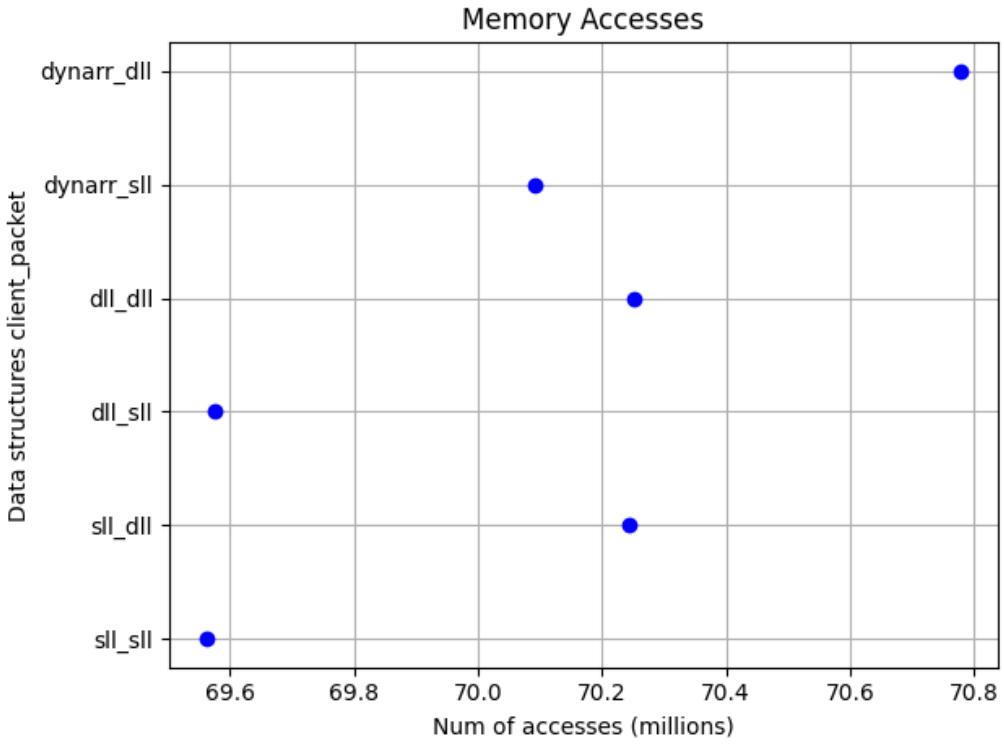
```python
import matplotlib.pyplot as plt

#Filenames
files = ["sll_sll",
         "sll_dll",
         "sll_dynarr",
         "dll_sll",
         "dll_dll",
         "dll_dynarr",
         "dynarr_sll",
         "dynarr_dll",
         "dynarr_dynarr"]

#Read accesses and footprints
accesses = []
footprints = []
for filename in files:

    #Accesses
    file_acc = open("results/" + filename + "_accesses.txt", "r")
    line = file_acc.readline()
    mes = float(line) / 1000000
    accesses.append(mes)
    file_acc.close()

    #Footprints
    file_fp = file_fp = open("results/" + filename + "_footprint.txt", "r")
    mul = 1
    line = file_fp.readline()
    if(line[4:6]=="MB"):
        mul = 1000
    line = file_fp.readline()
    footprints.append(float(line[:5]) * mul)
    file_fp.close()

print(footprints)

#Full accesses plot
plt.title("Memory Accesses")
plt.plot(accesses, files, "bo" )
plt.ylabel("Data structures client_packet")
plt.xlabel("Num of accesses (millions)")
plt.tight_layout()
plt.grid()
plt.savefig('Images/accesses.png')

#Footprin plot
plt.figure()
plt.title("Memory Footprints")
plt.plot(footprints, files, "ro" )
plt.ylabel("Data structures client_packet")
plt.xlabel("Footprints (KB)")
plt.tight_layout()
plt.grid()
plt.savefig('Images/footprints.png')


#Remove dynarr client
accesses.remove( accesses[ files.index("sll_dynarr") ])
files.remove("sll_dynarr")

accesses.remove( accesses[ files.index("dll_dynarr") ])
files.remove("dll_dynarr")

accesses.remove( accesses[ files.index("dynarr_dynarr") ])
files.remove("dynarr_dynarr")

#Accesses plot without dynarr
plt.figure()
plt.title("Memory Accesses")
plt.plot(accesses, files, "bo" )
plt.ylabel("Data structures client_packet")
plt.xlabel("Num of accesses (millions)")
plt.tight_layout()
plt.grid()
plt.savefig('Images/best_accesses.png')
```

b) Βέλτιστος συνδυασμός για ελαχιστοποίηση αριθμού προσβάσεων στη μνήμη

Τα δεδομένα απεικονίζονται παρακάτω σε εκατομμύρια προσβάσεις ανα συνδυασμό. Από τις μετρήσεις παρατηρούμε ότι στις περιπτώσεις που χρησιμοποιείται ο δυναμικός πίνακας (Dynamic Array - dynarr) δεδομένων σε μια υλοποίηση το πλήθος προσβάσεων στη μνήμη αυξάνεται σημαντικά.



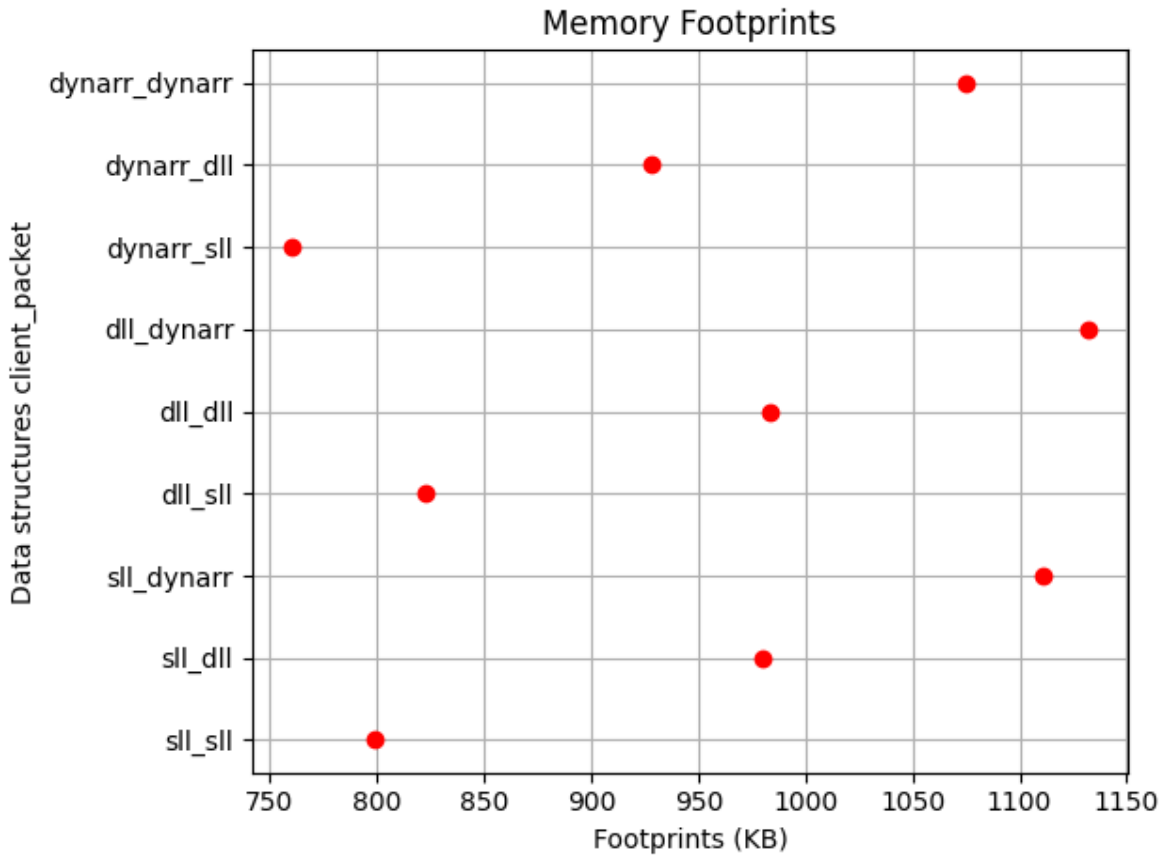
Στη συνέχεια σχεδιάζουμε το ίδιο διάγραμμα αφαιρώντας τις περιπτώσεις *sll_dynarr*, *dll_dynarr*, *dynarr_dynarr*.



Προκύπτει λοιπόν, πως ο ελάχιστος αριθμός προσβάσεων στη μνήμη (memory accesses) επιτυγχάνεται με τον συνδυασμό ***Single Linked List - Single Linked List*** για κόμβους και πακέτα αντίστοιχα.

c) Βέλτιστος συνδυασμός για ελαχιστοποίηση απαιτούμενου μεγέθους μνήμης

Αντίστοιχη διαδικασία ακολουθείται για το μέγιστο μέγεθος μνήμης που απαιτείται. Στο επόμενο διάγραμμα εμφανίζονται τα μεγέθη μνήμης ανα συνδυασμό σε KB.



Από το διάγραμμα είναι εμφανές ότι λιγότερη μνήμη απαιτείται στην περίπτωση που χρησιμοποιείται ο συνδυασμός ***Dynamic Array - Single Linked List*** για κόμβους και πακέτα αντίστοιχα.

Άσκηση 2: Βελτιστοποίηση αλγορίθμου Dijkstra

Στην επόμενη άσκηση θα επαναλάβουμε τη μεθοδολογία DDTR με στόχο την βελτιστοποίηση του αλγορίθμου Dijkstra και την αναζήτηση της συντομότερης διαδρομής. Χρησιμοποιώντας και πάλι τις δομές δεδομένων Single Linked List (SLL), Double Linked List (DLL) και Dynamic Array (DYN_ARR) θα συγκρίνουμε τα αποτελέσματα με την αρχική έκδοση του αλγορίθμου και θα επιδιώξουμε τον προσδιορισμό της δομής εκείνης που επιβάλλει τον μικρότερο δυνατό αριθμό προσβάσεων στη μνήμα αλλά και απαιτεί τη λιγότερη δυνατή μνήμη.

a) Αποτελέσματα αρχικής υλοποίησης

Για την εκτέλεση του προγράμματος *dijkstra.c* που δίνεται από την εκφώνηση εκτελούμε την εντολή *gcc dijkstra.c -o dijkstra >>dijkstra_out/initial_dijkstra.txt*. Ο φάκελος dijkstra_out χρησιμοποιείται προκειμένου να συγκεντρωθούν τα αποτελέσματα των εκδόσεων και να διαπιστωθεί η ορθή λειτουργία του προγράμματος.

Τα αποτελέσματα του dijkstra.c είναι:

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

b) Εισαγωγή βιβλιοθήκης

Για την εισαγωγή βιβλιοθήκης δημιουργήσαμε το αρχείο *dijkstra.h* στο οποίο κάνουμε τα απαραίτητα include των header files. Συγκεκριμένα, βάζοντας σε σχόλια τις πρώτες γραμμές του κώδικα επιβάλλουμε την χρήση των κατάλληλων συναρτήσεων ώστε να αντικατασταθεί η δομή δεδομένων που χρησιμοποιεί ο αλγόριθμος dijkstra.

```
dijkstra > C dijkstra.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
```

Στη συνέχεια μετασχηματίζουμε τον κώδικα του αρχείου *dijkstra.c* και παράγουμε το πρόγραμμα *opt_dijkstra.c* για την εκτέλεση του αλγορίθμους με τις διαφορετικές δομές δεδομένων. Ιδιαίτερη σημασία έχει η αντικατάσταση των δηλώσεων των δομών δεδομένων για την χρήση απλά συνδεδεμένης λίστας, διπλά συνδεδεμένης λίστας και δυναμικού πίνακα στις αντίστοιχες περιπτώσεις

```
24  //Original List Declaration
25  //QITEM *qHead = NULL;
26
27  // qHead represents the List
28  #if defined(SLL)
29  cdsl_sll *qHead;
30  #elif defined(DLL)
31  cdsl_dll *qHead;
32  #else
33  cdsl_dyn_array *qHead;
34  #endif
```

αλλά και η δημιουργία και αρχικοποίηση μέσα στην συνάρτηση *main.*

```
182 //Initialize List
183 #if defined (SLL)
184   qHead = cdsl_sll_init();
185 #elif defined (DLL)
186     qHead = cdsl_dll_init();
187 #else
188     qHead = cdsl_dyn_array_init();
189 #endif
```

Επιπλέον αλλαγές πραγματοποιήθηκαν στις συναρτήσεις enqueue και dequeue προκειμένου να χρησιμοποιηθούν ήδη υλοποιημένες μέθοδοι από τα αρχεία της βιβλιοθήκης DDTR.

```c
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;

  //Library method "enqueue" replaces manual enqueue of qNew
  qHead->enqueue(0, qHead, (void *)qNew);

  g_qCount++;
}
```

```c
void dequeue (int *piNode, int *piDist, int *piPrev)
{
  #if defined(SLL)
    iterator_cdsl_sll it;
  #endif
  #if defined(DLL)
    iterator_cdsl_dll it;
  #endif
  #if defined(DYN_ARR)
    iterator_cdsl_dyn_array it;
  #endif

  it = qHead->iter_begin(qHead);

  QITEM *qKill = (QITEM *)qHead->iter_deref(qHead, it);

  if (qHead)
    {

      *piNode = qKill->iNode;
      *piDist = qKill->iDist;
      *piPrev = qKill->iPrev;

      //Library method "remove" replaces manual dequeue of qHead
      qHead->remove(0, qHead, qKill);
      g_qCount--;
    }
}
```

Προκειμένου να σιγουρευτούμε για την ορθή λειτουργία της νέας έκδοσης εκτελούμε την εντολή
gcc opt_dijkstra.c -o opt_dijkstra -pthread -lcdsl -L./../synch_implementations -I./../synch_implementations
και στη συνέχεια με
./opt_dijkstra input.dat *>>dijkstra_out/opt_dijkstra.txt*
παρατηρούμε ότι τα αποτελέσματα βγαίνουν ίδια με εκείνα της αρχικής εκτέλεσης.

c) Εκτέλεση της εφαρμογής

Για την εκτέλεση του προγράμματος με διαφορετικές δομές δεδομένων χρησιμοποιούμε παραλλαγή του script από την άσκηση 1.

```
#!/bin/sh

if [ ! $# -eq 1 ]
then
    echo "Usage: ./execute <data_structures>"
    echo "sll, dll, dynarr"
else
    #Compile
    gcc opt_dijkstra.c -o opt_dijkstra_$1 -pthread -lcdsl -L./../synch_implementations -I./../synch_implementations

    # #Memory access
    valgrind --log-file="$1_accesses_log.txt" --tool=lackey --trace-mem=yes ./opt_dijkstra_$1 input.dat
    cat $1_accesses_log.txt | grep 'I\| L' | wc -l >> ./results/$1_accesses.txt
    rm $1_accesses_log.txt

    # #Memory footprint
    valgrind --tool=massif --massif-out-file="$1_massif.out" ./opt_dijkstra_$1 input.dat
    ms_print $1_massif.out > $1_footprint_log.txt
    cat $1_footprint_log.txt | sed '8!d' >> ./results/$1_footprint.txt
    cat $1_footprint_log.txt | sed '9!d' >> ./results/$1_footprint.txt
    rm $1_massif.out $1_footprint_log.txt
fi
```

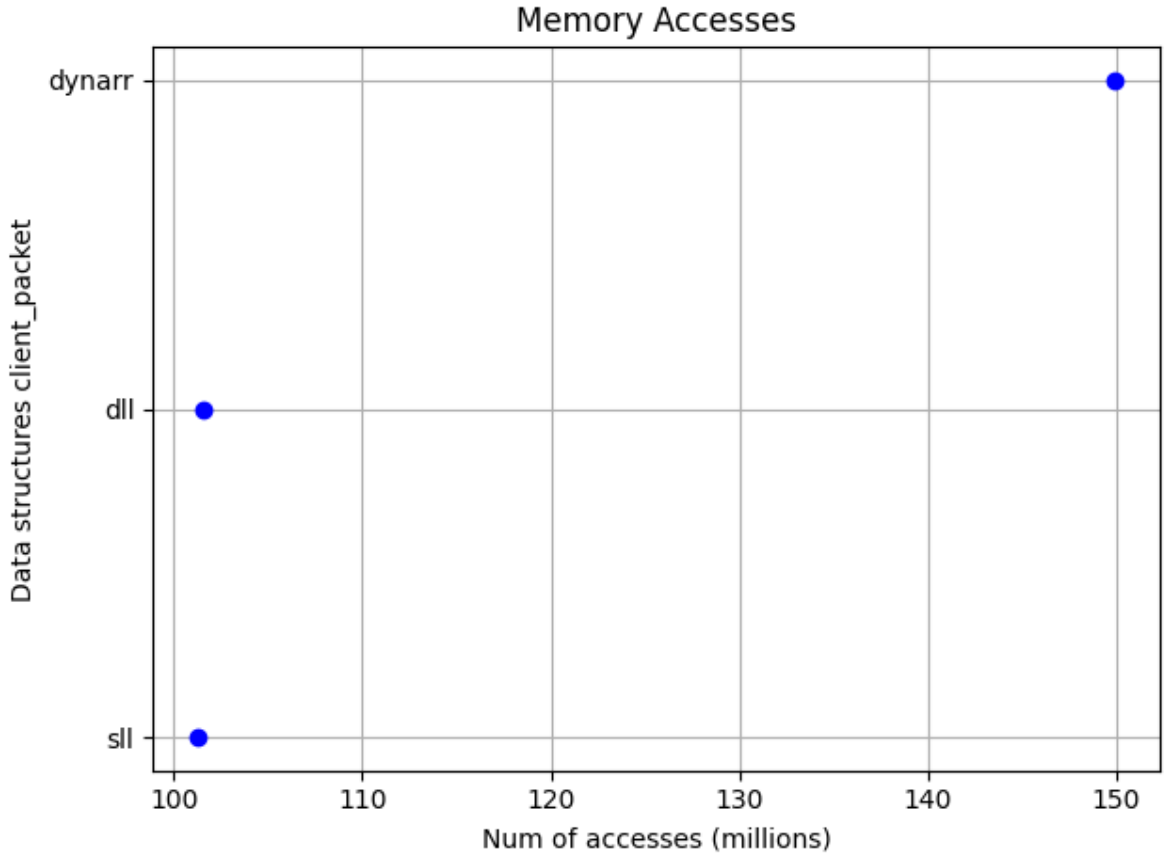
Όπως και στην προηγούμενη άσκηση τα αρχεία εξόδου των εργαλείων διαγράφονται ενώ στο φάκελο *results* αποθηκεύονται οι πληροφορίες σχετικά με την απαιτούμενη μνήμη και τις προσβάσεις σε αυτή.

Data Type	Accesses	Footprint
sll	101318367	707.7KB
dll	101522770	941.7KB
dynarr	149877811	360.6KB

Τα διαγράμματα αποτελεσμάτων σχεδιάστηκαν με τροποποίηση του αρχείου plot.py το οποίο εκτελείται αφού ληφθούν οι μετρήσεις και για τις 3 περιπτώσεις δομών δεδομένων. Ο τροποποιημένος κώδικας φαίνεται στην επόμενη σελίδα.

```python
import matplotlib.pyplot as plt

#Filenames
files = ["sll", "dll", "dynarr"]

#Read accesses and footprints
accesses = []
footprints = []
for filename in files:

    #Accesses
    file_acc = open("results/" + filename + "_accesses.txt", "r")
    line = file_acc.readline()
    mes = float(line) / 1000000
    accesses.append(mes)
    file_acc.close()

    #Footprints
    file_fp = file_fp = open("results/" + filename + "_footprint.txt", "r")
    mul = 1
    line = file_fp.readline()
    if(line[4:6]=="MB"):
        mul = 1000
    line = file_fp.readline()
    footprints.append(float(line[:5]) * mul)
    file_fp.close()

#Full accesses plot
plt.title("Memory Accesses")
plt.plot(accesses, files, "bo" )
plt.ylabel("Data structures client_packet")
plt.xlabel("Num of accesses (millions)")
plt.tight_layout()
plt.grid()
plt.savefig('Images/accesses.png')

#Footprin plot
plt.figure()
plt.title("Memory Footprints")
plt.plot(footprints, files, "ro" )
plt.ylabel("Data structures client_packet")
plt.xlabel("Footprints (KB)")
plt.tight_layout()
plt.grid()
plt.savefig('Images/footprints.png')

```

d) Βέλτιστος συνδυασμός για ελαχιστοποίηση αριθμού προσβάσεων στη μνήμη



Το παραπάνω διάγραμμα επιβεβαιώνει πως η καλύτερη επιλογή για ελαχιστοποίηση του πλήθους προσβάσεων στη μνήμη (memory accesses) είναι η ***Single Linked List.***

e) Βέλτιστος συνδυασμός για ελαχιστοποίηση απαιτούμενου μεγέθους μνήμης

Αντίστοιχα για την εκτέλεση του προγράμματος με χρήση της μικρότερης δυνατής ποσότητας μνήμης (memory footprint) ενδείκνυται η χρήση ***Dynamic Array***.