ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Σχεδιασμός Ενσωματωμένων Συστημάτων

2η Σειρά Ασκήσεων

Χειμερινό Εξάμηνο, Ακαδ. Έτος: 2023-2024

ΟΝΟΜΑΤΕΠΩΝΥΜΟ: ΔΙΑΜΑΝΤΙΔΗΣ ΘΕΟΧΑΡΗΣ

ΟΝΟΜΑΤΕΠΩΝΥΜΟ: ΙΩΑΝΝΟΥ ΚΩΝΣΤΑΝΤΙΝΟΣ

ΑΡΙΘΜΟΣ ΜΗΤΡΩΟΥ: 03119002

ΑΡΙΘΜΟΣ ΜΗΤΡΩΟΥ: 03119840

ΕΞΑΜΗΝΟ: 9ο

ΑΣΚΗΣΗ 1

Σκοπός της άσκησης είναι να βελτιστοποιήσουμε τις δομές δεδομένων του DRR με χρήση της μεθοδολογίας «Βελτιστοποίησης Δυναμικών Δεδομένων» (DDTR) ως προς το memory footprint και τα memory accesses.

Το memory footprint αναφέρεται στην συνολική μνήμη που θα χρησιμοποιήσει το πρόγραμμα κατά την εκτέλεσή του.

Τα memory accesses αναφέρονται στις εντολές read και write που γίνονται στην μνήμη.

Για τον σχηματισμό του Pareto ώστε να βρούμε τις καλύτερες δυνατές λύσεις θα χρησιμοποιήσουμε την βιβλιοθήκη DTTR και πιο συγκεκριμένα θα εξετάσουμε συνδυασμούς με τις παρακάτω δομές δεδομένων

SLL- Για την απλά συνδεδεμένη λίστα
DLL-Για την διπλά συνδεδεμένη λίστα
DYNAMIC ARRAY- Για τον δυναμικό πίνακα

Παρακάτω παρουσιάζουμε μια σύγκριση των παρακάτω δομών δεδομένων.

Δυναμικός Πίνακας

Είναι μια δομή από συνεχώς απαριθμημένα στοιχεία που έχουνε συγκεκριμένο μέγεθος. Σε αντίθεση με τους στατικούς πίνακες στους δυναμικούς πίνακες μπορούμε να δεσμεύσουμε και να αποδεσμεύσουμε χώρο κατά την διάρκεια της εκτέλεσης του προγράμματος.

Τα πλεονεκτήματα που προσφέρει ο Δυναμικός Πίνακα; είναι

1. Επιτρέπει άμεση πρόσβαση στα στοιχεία του πίνακα.
2. Απαιτεί λιγότερη συνολική μνήμη καθώς λείπουν στοιχεία όπως οι δείκτες που χρειάζονται οι λίστες για να γίνει η προσπέλαση.
3. Μπορούμε να αξιοποιήσουμε καλύτερα την τοπικότητα στην cache καθώς αποθηκεύεται σε συνεχής θέσεις στην μνήμη.

Τα μειονεκτήματα τα οποία έχει ο δυναμικός πίνακας είναι

1. Το μέγεθος το οποίο θα δεσμεύουμε κάθε φορά είναι σταθερό και η αύξηση και μείωση του μεγέθους απαιτεί την αντιγραφή στοιχείων.
2. Είναι δύσκολη η εισαγωγή και η διαγραφή στοιχείων.
3. Υπάρχει περίπτωση να μην χρησιμοποιείται ολόκληρη η μνήμη που έχουμε αναθέσει είτε να δημιουργηθεί πρόβλημα fragmentation

Τα πλεονεκτήματα που προσφέρει η Απλά Συνδεδεμένη Λίστα είναι τα παρακάτω

1. Μπορούμε να αυξήσουμε ή να μειώσουμε εύκολα το μέγεθος της λίστας
2. Μπορούμε να προσθέσουμε ανάμεσα στοιχεία στην λίστα καθώς κάνουμε χρήση δεικτών

Τα μειονεκτήματα τα οποία έχει η απλά συνδεδεμένη λίστα είναι

1. Για να έχουμε πρόσβαση σε ένα στοιχείο απαιτείται να διασχίσουμε όλα τα στοιχεία προς μία μόνο κατεύθυνση και άρα έχουμε μεγαλύτερους χρόνους πρόσβασης.
2. Απαιτείται memory overhead καθώς χρειάζονται δείκτες για να μπορεί να γίνει διάσχιση της λίστας.

Τα πλεονεκτήματα που προσφέρει η Διπλά Συνδεδεμένη Λίστα είναι τα παρακάτω

1. Μπορούμε να διασχίσουμε την λίστα και προς τις δύο κατευθύνσεις και άρα έχουμε ευκολία ως προς την προσπέλαση των στοιχείων.
2. Ομοίως με τις απλά συνδεδεμένες λίστες μπορούμε να προσθέσουμε ανάμεσα στοιχεία στην λίστα καθώς κάνουμε χρήση δεικτών

Τα μειονεκτήματα τα οποία έχει η διπλά συνδεδεμένη λίστα είναι

1. Χρειάζεται ακόμα περισσότερο overhead διότι χρησιμοποιούμε περισσότερους δείκτες
2.Ομοίως είναι πιο χρονοβόρα η εισαγωγή κάποιου στοιχείου σε σχέση με τον δυναμικό πίνακα.

Με βάση τα παραπάνω θα εφαρμόσουμε συνδυασμούς δομών δεδομένων ώστε να βρούμε την πιο αποδοτική ως προς την συνολική μνήμη και τον αριθμό των προσβάσεων.

Α) Αρχικά εφαρμόζουμε τεχνικές βελτιστοποίησης για τον αλγόριθμο DDR με εξερεύνηση των δομών δεδομένων για την λίστα πακέτων (pList) και την λίστα κόμβων (ClientList).

Για καθεμία από αυτές τις δομές δεδομένων θα δοκιμάσουμε τις παραπάνω αναφερόμενες και θα σχηματίζουμε όλους τους δυνατούς συνδυασμούς

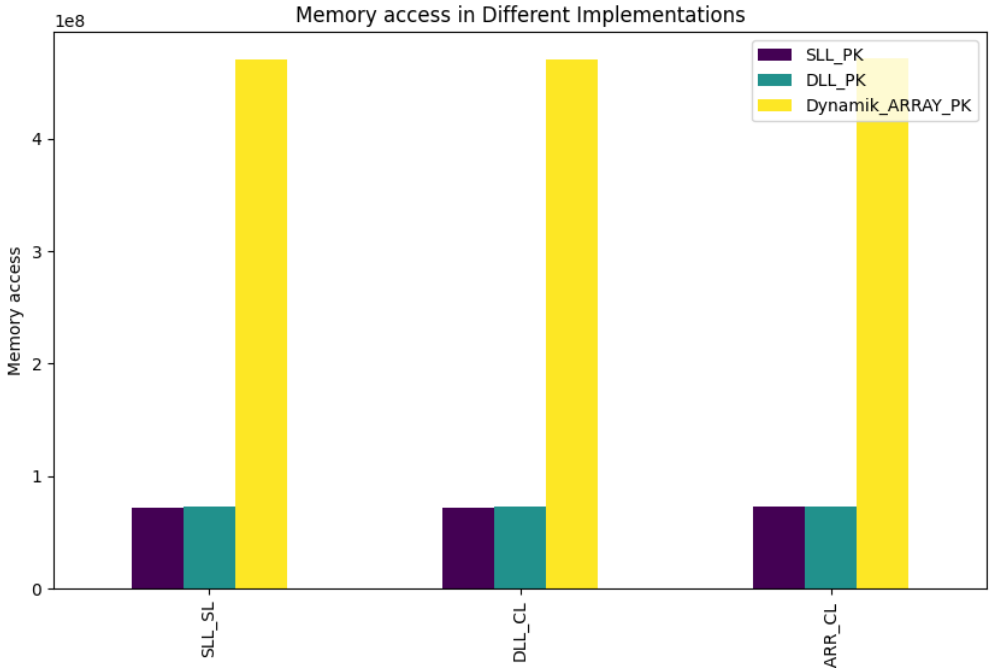
Memory Accesses

ClientList\pList	SLL_PK	DLL_PK	DYN_ARR_PK
SLL_CL	71.880.849	72.553.063	470.934.301
DLL_CL	71.892.828	72.564.837	470.949.335
DYN_ARR_CL	72.379.508	73.064.540	471.682.264

Memory Footprint

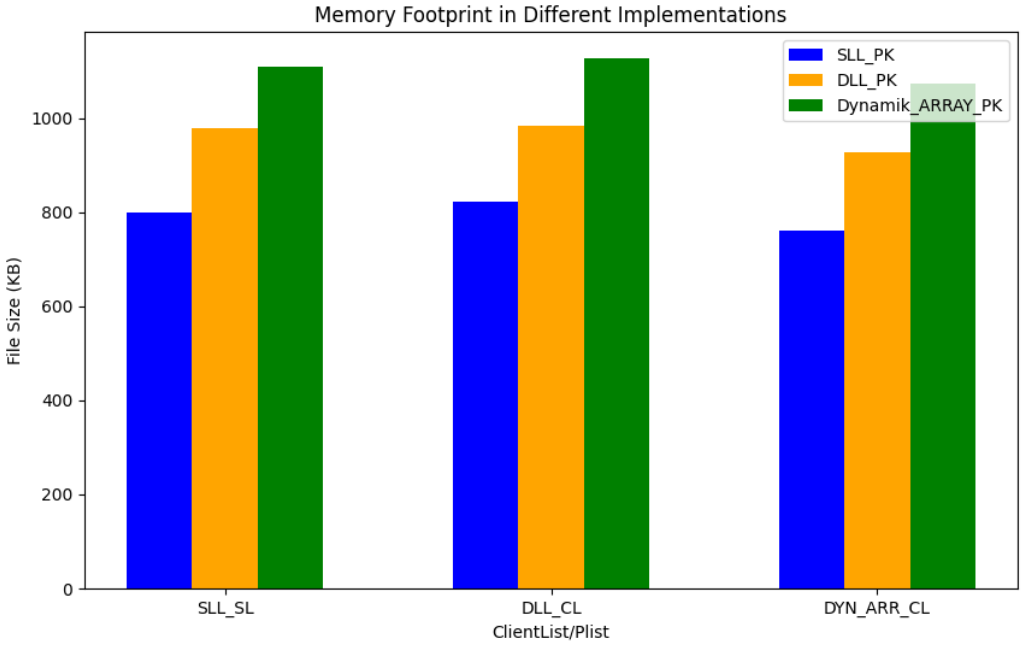
ClientList\pList	SLL_PK	DLL_PK	DYN_ARR_PK
SLL_CL	798.8 KB	980.3 KB	1.111 MB
DLL_CL	823.0 KB	983.3 KB	1.128 MB
DYN_ARR_CL	760.2 KB	928.5 KB	1.075 MB

B) Από τα παραπάνω αποτελέσματα προκύπτει ότι ο μικρότερος αριθμός προσβάσεων στην μνήμη προκύπτει για SLL-SLL. Σε γενικές γραμμές βλέπουμε ότι το pList είναι αυτό που προκαλεί αλλαγές στον αριθμό των προσβάσεων στην μνήμη. Οι χειρότεροι συνδυασμοί προκύπτουν όταν το pList είναι με δυναμικό πίνακα. Όταν το pList υλοποιείται με SLL είτε με DLL τότε έχουμε την ίδια τάξη μεγέθους στον αριθμό των προσβάσεων στην μνήμη. Επομένως χειροτέρευση παρατηρείται όταν τα πακέτα υλοποιούνται με δυναμικό πίνακα και είναι ανεξάρτητα από την δομή των Nodes. Το διάγραμμα με τις προσβάσεις στην μνήμη φαίνεται παρακάτω.

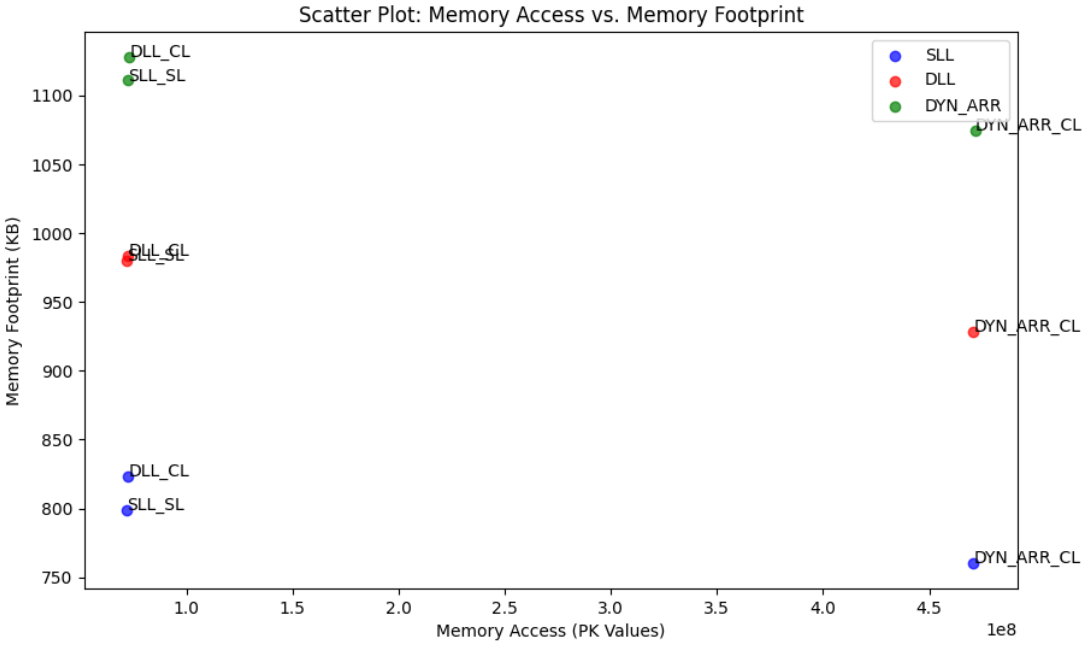


C) Όσον αφορά το memory footprint βλέπουμε ότι μεγαλύτερο χώρο στην μνήμη απαιτείται όταν τα πακέτα υλοποιούνται με δυναμικό πίνακα ενώ καλύτερη υλοποίηση γίνεται με SLL. Αυτό εξηγείται από το γεγονός ότι ο δεσμεύεται και αποδεσμεύεται πολλές φορές χώρος για τα πακέτα και άρα στον δυναμικό πίνακα δεσμεύεται συνεχώς περισσότερη μνήμη η οποία εν τέλει δεν αξιοποιείται. Επίσης το SLL έχει λιγότερη μνήμη από το DLL διότι χρειάζεται λιγότερους δείκτες για την προσπέλαση των στοιχείων.

Όσον αφορά τον αριθμό των κόμβων επειδή αυτός δεν μεταβάλλεται στο πρόγραμμα αναμένουμε ο δυναμικός πίνακας να δεσμεύει λιγότερο χώρο διότι δεν θα γίνει resize. Επομένως το SLL και το DLL θα δεσμεύσουν χώρο τόσο για τα δεδομένα αλλά και για τους pointers (μάλιστα το DLL θα έχει μεγαλύτερο memory footprint καθώς έχει περισσότερους δείκτες). Τα παραπάνω επιβεβαιώνουν και το αποτέλεσμα το οποίο εξάγουμε ότι ο καλύτερος συνδυασμός προκύπτει για DYN_ARR_CL και SLL_PK ενώ ο χειρότερος συνδυασμός για DLL_CL και DYN_ARR_PK. Εποπτικά έχουμε το παρακάτω διάγραμμα



Συνολικά το Pareto διάγραμμα είναι



ΑΣΚΗΣΗ 2

Α) Σε αυτό το ερώτημα θα εκτελέσουμε τον αλγόριθμο του Dijkstra για έναν πίνακα 100x100 και θα υπολογίσουμε τα 20 συντομότερα μονοπάτια. Η δομή δεδομένων στην οποία θα κάνουμε βελτιστοποίηση δυναμικών δεδομένων είναι η λίστα στην οποία τοποθετούνται οι κόμβοι. Αρχικά εκτελούμε την εφαρμογή και παίρνουμε τα αποτελέσματα του Dijkstra ώστε να συγκρίνουμε με τα αρχικά δεδομένα. Τα αποτελέσματα φαίνονται παρακάτω

```
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
```

Β) Ο τροποποιημένος κώδικας παρουσιάζεται παρακάτω όπου έχουμε εισάγει την βιβλιοθήκη και έχουμε αντικαταστήσει την δομή των κόμβων με τις δομές δεδομένων της βιβλιοθήκης. Ο κώδικα φαίνεται παρακάτω

Αρχικά εισάγουμε τις απαραίτητες βιβλιοθήκες και ορίζουμε σαν global μεταβλητή την δομή που θα χρησιμοποιήσουμε

```
#include <stdio.h>
#include <stdlib.h>
#define NUM_NODES                       100
#define NONE                            9999
#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
#if defined(SLL)
        cdsl_sll *q;
#elif defined(DLL)
        cdsl_dll *q;
#else
        cdsl_dyn_array *q;
#endif
```

Για την προσθήκη κόμβων αλλάζουμε την συνάρτηση enqueue ώστε να χρησιμοποιεί την έτοιμη συνάρτηση της βιβλιοθήκης με τα ορίσματα που φαίνονται παρακάτω

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

}
```

Στην συνάρτηση dequeue ομοίως εισάγουμε την συνάρτηση της βιβλιοθήκης και επίσης ορίζουμε τους iterators για κάθε δομή ώστε να μπορούμε να προσπελάσουμε τα στοιχεία

```
void dequeue (int *piNode, int *piDist, int *piPrev)
{
#if defined (SLL)
        iterator_cdsl_sll it, end;

#elif defined (DLL)
        iterator_cdsl_dll it, end;

#else
        iterator_cdsl_dyn_array it, end;
#endif
        it = q->iter_begin(q);
        QITEM* v = (QITEM*)(q->iter_deref(q, it));

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

      q->dequeue(0, q);
      g_qCount--;
}


int qcount (void)
{
  return(g_qCount);
}
```

Τέλος στην main αρχικοποιούμε τις δομές δεδομένων και απελευθερώνουμε τον χώρο με free(q). Ο κώδικας φαίνεται παρακάτω

```c
int main(int argc, char *argv[]) {
  int i,j,k;
  FILE *fp;
#if defined (SLL)
        q = cdsl_sll_init();
#elif defined (DLL)

        q = cdsl_dll_init();
#else

        q = cdsl_dyn_array_init();
#endif
  if (argc<2) {
    fprintf(stderr, "Usage: dijkstra <filename>\n");
    fprintf(stderr, "Only supports matrix size is #define'd.\n")
  }

  /* open the adjacency matrix file */
  fp = fopen (argv[1],"r");

  /* make a fully connected matrix */
  for (i=0;i<NUM_NODES;i++) {
    for (j=0;j<NUM_NODES;j++) {
      /* make it more sparce */
      fscanf(fp,"%d",&k);
        AdjMatrix[i][j]= k;
    }
  }

  /* finds 10 shortest paths between nodes */
  for (i=0,j=NUM_NODES/2;i<20;i++,j++) {
        j=j%NUM_NODES;
      dijkstra(i,j);
  }
  free(q);
  exit(0);
```

c) Στην συνέχεια τρέχουμε τον αλγόριθμο για τις διάφορες δομές δεδομένων συμφωνα με τις αλλαγές στον κώδικα που κάναμε παραπάνω και καταγράφουμε τα αποτελέσματα στον παρακάτω πίνακα.

Data type	Memory access	Memory footprint
SLL	102276224	356.7
DLL	102455408	471.4 KB
DYN_ARR	149490709	360.6

d) Σε αυτό το ερώτημα θα εξετάσουμε τα memory access για τους διαφορετικούς τύπους δεδομένων.

Data type	Memory access
SLL	102276224
DLL	102455408
DYN_ARR	149490709

Όπως φαίνεται από το παραπάνω πίνακα παρατηρούμε ότι λιγότερες προσβάσεις στην μνήμη έχουμε για SLL(Απλά συνδεδεμένη λίστα).

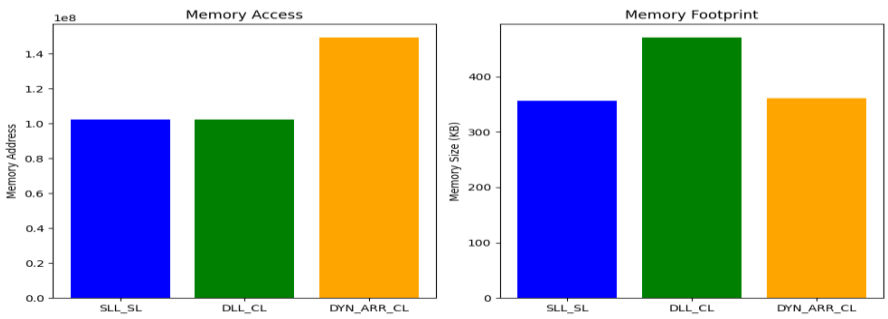
e) Ομοίως εξετάζουμε τους διαφορετικούς τύπους δεδομένων αναλογικά με το memory footprint.

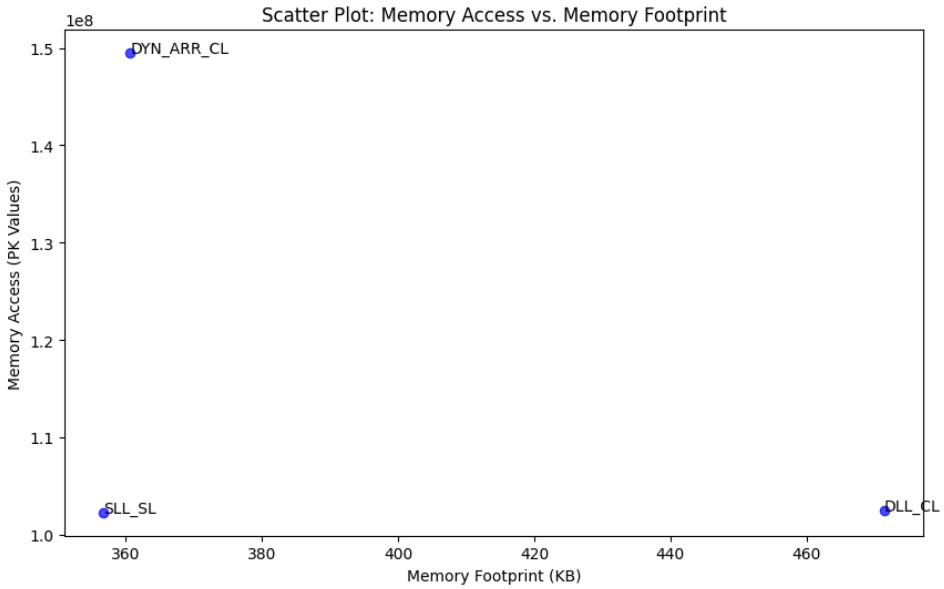
Memory footprint

Data type	Memory footprint
SLL	356.7
DLL	471.4 KB
DYN_ARR	360.6

Πάλι παρατηρούμε ότι η βέλτιστή επιλογή που μας προσφέρει το ελάχιστο μέγεθος μνήμης που απαιτείται από την εφαρμογή είναι η απλά συνδεδεμένη λίστα SLL.

ρακάτω βλέπουμε συνολικά τα αποτελέσματα μας σε διαγράμματα bar plot όσο και σε pareto για τα 3 σημεία που έχουμε.





Παρατηρούμε λοιπόν ότι με μεγάλη διαφορά και ως προς τους δύο παράγοντες (προσβάσεις στη μνήμη και footprint στην μνήμη) η καλύτερη περίπτωση είναι η απλά συνδεδεμένη λίστα.

Ανάμεσα στους τύπους δεδομένων DLL και DYN_ARR δεν υπάρχει ξεκάθαρη επιλογή καθώς η διπλά συνδεδεμένη λίστα έχει λίγες προσβάσεις στην μνήμη σε σχέση με τον δυναμικό πίνακα. Αυτό συμβαίνει καθώς με μια λίστα έχουμε δυναμική ανάθεση μνήμης, ευελιξία σε αλλαγές και αποδοτικό έλεγχο προσπέλασης. Δηλαδή, η εισαγωγή και η διαγραφή στοιχείων σε λίστα απαιτούν μόνο ενημέρωση των δεικτών και όχι της ίδιας της τιμής (value).

Από την άλλη, ο δυναμικός πίνακας είναι ξεκάθαρο ότι έχει μικρό footprint. Αυτό συμβαίνει γιατί η DLL χρειάζεται εκτός από τιμές να αποθηκεύει και δύο pointers, έναν για να δείχνει στον προηγούμενο και έναν για να δείχνει στον επόμενο κόμβο, οπότε έχουμε επιπλέον πληροφορία που δεν είναι απαραίτητη.

Τέλος, η απλά συνδεδεμένη λίστα SLL είναι η καλύτερη επιλογή καθώς έχει λιγότερες προσβάσεις στην μνήμη (διότι αρκεί να αλλάξουμε μόνο τον δείκτη) και έχει επιπλέον λιγότερο footprint (σίγουρα από την διπλά συνδεδεμένη λίστα αφού έχει έναν pointer για το κάθε στοιχείο και όχι δύο). Ακόμη και από τον δυναμικό πίνακα, γιατί ο δυναμικός πίνακας μπορεί να έχει μόνο τις απαραίτητες τιμές αποθηκευμένες, αλλά να τις έχει συνεχόμενες στην μνήμη, ενώ η απλή συνδεδεμένη λίστα, η μνήμη αυξάνεται και μειώνεται κατά τη διάρκεια της εκτέλεσης του αλγορίθμου, προσφέροντας έτσι λιγότερη αδράνεια στη μνήμη.