# Απαλλακτική Εργασία στις Δομές Δεδομένων

Version: 2023-03-09

# Table of Contents

# 1. Εισαγωγή

Στις επόμενες ενότητες παρουσιάζονται οι ιδέες για την επίλυση κάθε θέματος, μαζί με ενδεικτικά κομμάτια κώδικα και αποτελέσματα μετά την εκτέλεσή τους. Ο πλήρης κώδικας για κάθε θέμα βρίσκεται στην τελευταία ενότητα.

# 2. Ανανέωση & Ανάκτηση Στοιχείου Πίνακα

**Υποθέτουμε ότι A, B, C είναι πίνακες με δείκτη 1..10 και τύπο στοιχείων «πραγματικούς αριθμούς». Να γραφεί μια διαδικασία C η οποία χρησιμοποιεί λειτουργίες retrieve και update για να υλοποιήσει την πρόσθεση πινάκων A := B + C.**

Λύση

Αρχικά θα υλοποιούμε την `update(S,c,i)` και `retrieve(S,c,i)` όπως παρουσιάζονται στις σημειώσεις, δηλαδή όπου `S` ο πίνακας, `c` το προς εισαγωγή ή προς επιστροφή στοιχείο και `i` ο δείκτης στον πίνακα.

```
void update(float S[], float c, int i) {
    *(S+i) = c;
}

void retrieve(float S[], float *c, int i) {
    *c = S[i];
}
```

Στην `update` εκμεταλλευόμαστε ότι το όνομα του πίνακα είναι αναφορά στη θέση μνήμης του για να αλλάξουμε απευθείας τη τιμή του ζητούμενου στοιχείου. Στην `retrieve`, δεδομένου ότι θέλουμε να έχουμε πρόσβαση από τη `main` στο επιστρεφόμενο στοιχείο αλλά και να κρατήσουμε την υπογραφή της συνάρτησης κοντά στης δοθείσης, αντί απλά ενός `float c` στέλνουμε τη θέση μνήμης μιας `float` τιμής.

```
#include <stdio.h>

int main() {
    float A[10];
    float B[10];
    float C[10];

    float entry;
    float retrievedB,retrievedC;

    for (int i = 0; i<10; i++) {
        printf("Give element %d of B: ",i+1);
        scanf("%f",&entry);
        update(B,entry,i);
        printf("Give element %d of C: ",i+1);
        scanf("%f",&entry);
        update(C,entry,i);

        retrieve(B,&retrievedB,i);
        retrieve(C,&retrievedC,i);
        update(A,retrievedB+retrievedC,i);
    }
```

```
    float retrievedA;
    for (int i = 0; i<10; i++) {
        retrieve(A, &retrievedA, i);
        printf("Element %d of A:=B+C is %f\n",i+1,retrievedA);
    }

    return 0;
}
```

Στη `main` ορίζουμε τους τρεις πίνακες 10 θέσεων, κάνουμε `update` τις τιμές των δύο με εισόδους από τον χρήστη και έπειτα κάνουμε `update` τις τιμές του τελικού πίνακα, μέσω αλλεπάλληλων `retrieve` των αντιστοίχων στοιχείων των πρώτων πινάκων.

# 3. Συνάρτηση Απεικόνισης Πίνακα

**Να υπολογιστεί η διεύθυνση κάθε στοιχείου ενός πίνακα A(1:2,1:3,3:3,1:2). Θεωρείστε ότι ο πίνακας έχει βασική διεύθυνση b=100 και μήκος συνιστώσας L=6, ενώ τα άνω και κάτω όρια των δεικτών του είναι όπως παραπάνω.**

Λύση

Θα υλοποιήσουμε τη συνάρτηση απεικόνισης πίνακα για διάσταση πίνακα και όρια δοσμένα από τον χρήστη.

## 3.1. Εισαγωγή Δεδομένων

```
#include <stdbool.h>

/*global variable that holds the dimension of the user's array*/
int dim;

/* a pointer to a vector of pointers each one of which point to the
 lower and upper bound indexes of the user's array */
int** boundsPtr;

bool getUserInput(){
    int i,j;

    printf("Type the array's dimension: ");
    scanf("%d", &dim);

    /* user input validation */
    if (dim <= 0){
        printf("Invalid input. \n");
        return false;
    }
    else{
        /* dynamic allocation of the pointer (the rows of the 2D array
representation)*/
        boundsPtr = (int**)malloc(sizeof(int)*dim);
        for(i=0;i<dim;i++){
            /* dynamic allocation of a pointer (the columns of the 2D array
representation)*/
            boundsPtr[i] = (int*)malloc(sizeof(int)*2);
            for (j=0;j<2;j++){
                printf("Type index %d of dimension %d\n",j+1,i+1);
                scanf("%d",&boundsPtr[i][j]);
                if (j%2 == 1){
                    /* user input validation */
                    if (boundsPtr[i][j-1]>boundsPtr[i][j]){
                        printf("invalid input.\n");
                        return false;
```

```
                    }
                }
            }


        }
        return true;
    }
}
```

Στην `getUserInput()` ζητάμε από τον χρήστη τη διάσταση του πίνακα και, αν είναι θετική, ορίζουμε τον διδιάστατο πίνακα (`dim` γραμμές, 2 στήλες) που θα κρατήσει τα όρια που θα δώσει ο χρήστης, κάνοντας έναν υποτυπώδη έλεγχο εγκυρότητάς τους.

Αντιμετωπίζουμε τον διδιάστατο πίνακα σαν πίνακα με δείκτες για άλλους μονοδιάστατους πίνακες (*δείκτες δεικτών*), για να είναι εύκολη η δήλωσή του σαν `global` μεταβλητή ενώ δεν ξέρουμε εκ των προτέρων το μέγεθός του. Αντίστοιχη τακτική χρησιμοποιούμε αργότερα και για τον `indexes`, που θα έχει όλους τους διαφορετικούς συνδυασμούς συντεταγμένων βάσει των ορίων που έχουν δοθεί.

## 3.2. Τετράδες Συντεταγμένων

```
/* 2D array that will store all valid coordinates*/
int** indexes;

/* find total number of elements of multidimensional array */
int findTotal() {
    int total = 1;
    for (int i = 0; i < dim; i++) {
        total *= (boundsPtr[i][1] - boundsPtr[i][0] + 1);
    }
    return total;
}

/* find recursively the valid indexes of all the elements of the user's array and
calculate their address*/
int count = 0; // global counter for entries in indexes, to avoid unexpected behaviour
during recursion
void findIndexes(int guide, int temp[]) {
    for (int i = boundsPtr[guide][0]; i <= boundsPtr[guide][1]; i++) {
        temp[guide] = i;
        if (guide == dim - 1) {
            for (int j = 0; j < dim; j++) {
                indexes[count][j] = temp[j];
            }
            count++;
        } else {
            findIndexes(guide+1, temp);
```

```
        }
    }
}
```

Με την `findTotal` υπολογίζουμε τον αθροιστικό αριθμό στοιχείων που θα έχει ο πολυδιάστατος πίνακας, έτσι ώστε να μπορέσουμε έπειτα με την `findIndexes` να βρούμε τι συντεταγμένες μπορεί να έχει ένα στοιχείο του πίνακα. Στην περίπτωση της εκφώνησης, όλες τις πιθανές τετράδες συντεταγμένων $(i_1,i_2,i_3,i_4)$ με

- $1 <= i_1 <= 2$
- $1 <= i_2 <= 3$
- $i_3 = 3$
- $1 <= i_4 <= 2$

## 3.3. Υπολογισμός Θέσεων Μνήμης

```
/* calculate all the (dimension + 1) coefficients of the given element of the array */
void findCoefficients(int coefficients[]) {
    coefficients[dim] = length;

    for (int i = dim - 1; i > 0; i--) {
        coefficients[i] = (boundsPtr[i][1] - boundsPtr[i][0] + 1) * coefficients[i+1];
    }

    coefficients[0] = base;
    for (int i = 0; i < dim; i++) {
        coefficients[0] -= coefficients[i+1] * boundsPtr[i][0];
    }
}

/* calculate and print the exact memory address of the given element of the array */
void findAddress(int total, int coefficients[]) {
    int addr;
    for (int i = 0; i < total; i++) {
        printf("Address of element ( ");
        addr = coefficients[0];
        for (int j = 0; j < dim; j++) {
            printf("%d ", indexes[i][j]);
            addr += coefficients[j + 1] * indexes[i][j];
        }
        printf(") is %d\n", addr);
    }
}
```

Στην `findCoefficients` υπολογίζουμε όλους τους συντελεστές $c_i$, `0 <= i <= 4` βάσει των τύπων

> $c_4 = L$

> $c_{j-1} = (u_j - l_j + 1) * c_j$ για $j = 1,2,3$
> $c_0 = b - c_1 * l_1 - c_2 * l_2 - c_3 * l_3 - c_4 * l_4$

και στην `findAddress`, έχοντας ήδη γεμίσει των πίνακα συντεταγμένων `indexes`, βρίσκουμε τις διευθύνσεις κάθε στοιχείου.

```
int main() {
    if (getUserInput()){
        for (int i=0; i<dim; i++) {
            printf("===Row %d===\n",i+1);
            printf("Lower bound: %d, Upper bound: %d\n",boundsPtr[i][0],boundsPtr[i][
1]);
            printf("\n");
        }

        int total = findTotal();

        /* initialize recursion */
        int coefficients[dim+1];
        findCoefficients(coefficients);

        indexes = (int**) malloc(sizeof(int) * 100);
        for (int i = 0; i < total; i++) {
            indexes[i] = (int*) malloc(sizeof(int) * dim);
        };
        int temp[dim];

        findIndexes(0, temp);
        findAddress(total, coefficients);
    }
    return 0;
}
```

[ex2] | */images/ex2.png*

# 4. Σύστημα Αποθήκευσης Στοιχείων Φοιτητών

**Σε μια συλλογή στοιχείων, κάθε στοιχείο περιέχει πληροφορίες για τους φοιτητές του μεταπτυχιακού, όπως αριθμό μητρώου, επώνυμο, όνομα, πατρώνυμο, διεύθυνση κατοικίας, αριθμό σταθερού τηλεφώνου, αριθμό κινητού τηλεφώνου και επιλεγμένο μάθημα πρώτου εξαμήνου.**

- **Να γραφεί κώδικας C για την αναπαράσταση ενός εγγραφήματος φοιτητή.**
- **Να γραφεί κώδικας C για την αναπαράσταση του συνόλου των φοιτητών χρησιμοποιώντας έναν πίνακα εγγραφημάτων.**
- **Να χρησιμοποιηθεί μια δομή δείκτη για την πρόσβαση σε όλους τους φοιτητές που έχουν επιλέξει ένα συγκεκριμένο μάθημα.**

Λύση

Αρχικά ορίζουμε τις δομές `student` και `subjectNode`. Η πρώτη θα χρησιμοποιηθεί για να αποθηκεύουμε φοιτητές με τα ζητούμενα πεδία τους, ενώ η δεύτερη για να μπορέσουμε να φτιάξουμε μια λίστα για κάθε μάθημα (με τους φοιτητές που το παρακολουθούν). Ο κάθε κόμβος `subjectNode` θα έχει έναν δείκτη που θα δείχνει στον πρώτο μαθητή που το δήλωσε (`firstStudentPtr`) και, αντίστοιχα, ο κάθε μαθητής έναν δείκτη που θα δείχνει στον επόμενο που έχει δηλώσει το ίδιο μάθημα (`next`).

## 4.1. Σχέση Μαθημάτων και Μαθητών

[ex3] | */images/ex3.png*

Ουσιαστικά κάθε μάθημα της λίστας μαθημάτων θα δείχνει στην κεφαλή μίας λίστας μαθητών.

```
typedef struct subjectNode {
    char subject[50];
    struct student *firstStudentPtr;

    struct subjectNode *next;
}
subjectNode;

subjectNode *head;

typedef struct student {
    char id[9];
    char firstName[20];
    char lastName[20];
    char city[20];
    int age;
```

```
    char subject[50];

    struct student *next;
}
student;
```

Ορίζουμε χειροκίνητα τρεις μαθητές και προσθέτουμε τα στοιχεία τους στον ζητούμενο πίνακα, ακολουθώντας το παραπάνω διάγραμμα.

## 4.2. Προσθήκη Μαθητών

Σε κάθε μαθητή η προσθήκη του μαθήματος γίνεται μέσω της `addStudentToSubject`. Εκεί ελέγχουμε αν το μάθημα υπάρχει ήδη στη λίστα και

- αν δεν υπάρχει το προσθέτουμε στη λίστα των μαθημάτων και θέτουμε το `firstStudentPtr` να δείχνει στον μαθητή που το δήλωσε πρώτος.
- αν υπάρχει διατρέχουμε τη λίστα μαθητών που το έχει ήδη δηλώσει και θέτουμε το `next` του τελευταίου να δείχνει στον μαθητή που το δήλωσε τώρα.

```
/**
* Checks whether a subject is already inserted
* in the subjects linked list.
*
* Parameters
*   subject - name of the subject to-be-added
*/
subjectNode *subjectExists(char subject[]) {
    subjectNode *subjectPtr = head;
    while (subjectPtr != NULL) {
        if (strcmp(subjectPtr->subject,subject) == 0) {
            return subjectPtr;
        }
        subjectPtr = subjectPtr->next;
    }
    return NULL;
}

/**
* Adds a student to the linked list of a
* subject's student list.
*
* Parameters
*   studentPtr - pointer to the student to-be-added
*/
void addStudentToSubject(student *studentPtr) {
    subjectNode *p = subjectExists(studentPtr->subject);
    if (p == NULL) {
        subjectNode *newSubjectPtr = (subjectNode*) malloc(sizeof(subjectNode));
        strcpy(newSubjectPtr->subject,studentPtr->subject);
```

```
        newSubjectPtr->firstStudentPtr = studentPtr;
        newSubjectPtr->next = head;
        head = newSubjectPtr;
    } else {
        student *studentAttendingSubjectPtr = p->firstStudentPtr;
        while (studentAttendingSubjectPtr->next != NULL) {
            studentAttendingSubjectPtr = studentAttendingSubjectPtr->next;
        }
        studentAttendingSubjectPtr->next = studentPtr;
    }
}
```

Αυτό που επιτυγχάνεται με την παραπάνω αντιμετώπιση, είναι οι λίστες των μαθητών ενός συγκεκριμένου μαθήματος να ανανεώνονται *αυτόματα* κατα την προσθήκη ενός νέου μαθητή στο σύστημα. Δε χρειάζεται προσπέλαση της λίστας όλων των μαθητών, αλλά μόνο της λίστας των μαθημάτων, η οποία θα είναι πάντα λιγότερων κόμβων.

## 4.3. Εμφάνιση Μαθητών ανά Μάθημα

Για να εμφανίσουμε τη λίστα μαθητών που παρακολουθεί ένα συγκεκριμένο μάθημα, διατρέχουμε τη λίστα μαθημάτων για να δούμε αν αυτό το μάθημα υπάρχει (*δηλαδή αν έχει δηλωθεί τουλάχιστον μία φορά*) και, σε περίπτωση που βρεθεί, αρχίζουμε από τον μαθητή στον οποίο δείχνει το `firstStudentPointer` και σε κάθε επόμενό του (`next`).

```
/**
* Given the name of a subject, prints the information of all students studying it.
* In case the subject doesn't exist, informs the user accordingly.
*
* Parameters
*   subject - name of the subject
*/
void readStudentsBySubject(char subject[]) {
    subjectNode *subjectPtr = subjectExists(subject);

    if (subjectPtr == NULL) {
        printf("***Subject Does Not Exist***\n\n");
        return;
    }
    printf("***%s***\n\n",subject);

    student *studentPtr = subjectPtr->firstStudentPtr;
    while (studentPtr != NULL) {
        readStudent(studentPtr);
        studentPtr = studentPtr->next;
    }
}
```

# 5. Υλοποίηση Στοίβας με Πίνακα

**Να υλοποιηθεί μια στοίβα LIFO με χρήση πίνακα και της γλώσσας C.**

Λύση

Αρχικοποιούμε τον πίνακα της στοίβας ορίζοντας μέγεθος ακεραίων `stackLength` ίσο με 20. Ορίζουμε την global μεταβλητή `topIndex` για να μπορούμε στη συνέχεια να ελέγξουμε τη χωρητικότητα της στοίβας.

```c
#include <stdio.h>
#include <stdlib.h>

#define stackLength 20
int stack[stackLength] = {0};
int topIndex = -1;
```

## 5.1. Βασικές Λειτουργίες

- **push()**

Ελέγχουμε αν η στοίβα είναι γεμάτη συγκρίνοντας τη μεταβλητή `topIndex` με το μέγεθος του πίνακα `stackLength`.

Αν η στοίβα είναι γεμάτη, εμφανίζουμε αντίστοιχο μήνυμα στο χρήστη. Διαφορετικά αυξάνουμε τη μεταβλητή `topIndex` κατά ένα και αποθηκεύουμε τον αριθμό `num` στην αντίστοιχη θέση.

```c
void push(int num) {
    if (topIndex > (stackLength - 1)){
        printf("Process failed. Stack if full.");
    }
    else {
        topIndex++;
        stack[topIndex] = num;
        printf("\nNumber added!");
    }
}
```

[ex4 1] | */images/ex4_1.png*

- **pop()**

Ελέγχουμε αν η στοίβα είναι άδεια ελέγχοντας αν η μεταβλητή `topIndex` του πίνακα ισούται με το -1. Αν είναι τυπώνουμε το αντίστοιχο μήνυμα στο χρήστη, διαφορετικά μειώνουμε τη μεταβλητή `topIndex` κατά ένα και αφαιρούμε τον τελευταίο αριθμό που προστέθηκε, ενημερώνοντας στη συνέχεια το χρήστη.

```
void pop() {
    if (topIndex == -1) {
        printf("\nNothing to remove. Stack is empty.");
    }
    else {
        topIndex--;
        printf("\nNumber removed!");
    }
}
```

[ex4 2] | */images/ex4_2.png*

Το στοιχείο που έχει "*διαγραφεί*" εξακολουθεί να υπάρχει στον πίνακα (στη θέση `topIndex + 1`), αλλά δεν εμφανίζεται κατά την εκτύπωση των στοιχείων, αφού εμφανίζουμε μέχρι και τον `topIndex`. Μόλις εισαχθεί ένα νέο στοιχείο στη στοίβα, αυτό θα κάνει overwrite την τιμή που άτυπα έχει διαγραφεί.

- **display()**

Ελέγχουμε αν η στοίβα είναι άδεια ελ'εγχοντας αν η μεταβλητή `topIndex` του πίνακα ισούται με -1. Αν είναι τυπώνουμε το αντίστοιχο μήνυμα στο χρήστη, διαφορετικά τυπώνουμε τα περιεχόμενά της στο χρήστη.

```
void display() {
    int i;
    if (topIndex == -1) {
        printf("\nNothing to display. Stack is empty.");
        return;
    }
    else {
        printf("\nStack contains: ");
        for (i=0;i<=topIndex;i++) {
            printf("%d  ",stack[i]);
        }
    }
}
```

Στο αρχικό μενού δίνουμε στο χρήστη την επιλογή να

1. προσθέσει έναν καινούριο αριθμό
2. αφαιρέσει έναν αριθμό
3. να δει τα περιεχόμενα της στοίβας ή
4. να τερματίσει το πρόγραμμα.

Για κάθε επιλογή καλούμε την αντίστοιχη συνάρτηση (`push`, `pop`, `display`, `exit`) μέσα σε μια `switch case`. Στην περίπτωση που ο χρήστης δώσει μια μη έγκυρη επιλογή, τυπώνουμε το αντίστοιχο

μήνυμα με προτροπή να ξαναπροσπαθήσει.

```
int main() {
    int choice, number;

    while(1) {
        printf("-----------------------------------\n");
        printf("Stack menu: \n");
        printf("1. Add an integer number to the stack\n");
        printf("2. Remove a number from the stack\n");
        printf("3. View stack's contents\n");
        printf("4. Exit the program\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch(choice) {
            case 1:
                printf("Enter number: ");
                scanf("%d", &number);
                push(number);
                break;
            case 2:
                pop();
                break;
            case 3:
                display();
                break;
            case 4:
                printf("Bye!");
                exit(0);
            default:
                printf("Invalid choice, please try again.");
        }
        printf("\n\n");
    }
}
```

## 5.2. Ακραίες περιπτώσεις

[ex4 3] | */images/ex4_3.png*

[ex4 4] | */images/ex4_4.png*

# 6. Υλοποίηση Στοίβας με Συνδεδεμένη Λίστα

**Να υλοποιηθεί μια στοίβα LIFO με χρήση συνδεδεμένης λίστας και της γλώσσας C.**

Λύση

Χρησιμοποιούμε τη `struct` για να ορίσουμε τον κόμβο `Node` όπου περιέχεται η ακέραια μεταβλητή `number` και ο δείκτης `next` ο οποίος θα δείχνει στον επόμενο κόμβο. Τέλος, για τον ίδιο τύπο δεδομένων, ορίζουμε το δείκτη `head` για να σηματοδοτήσουμε τη διεύθυνση του πρώτου κόμβο της λίστας.

```
/* Declares a linked list's node structure containing a data part and its pointer with
the address of the next node  */
typedef struct Node {
    int number;
    struct Node* next;
}
Node;

Node* head;
```

## 6.1. Βασικές Λειτουργίες

- **push(int num)**

Υλοποιούμε τη συνάρτηση `push` με όρισμα την ακέραια μεταβλητή `num`. Ορίζουμε εσωτερικό δείκτη `tmp` και με τη χρήση της `malloc` κατανέμουμε χώρο στη μνήμη δυναμικά για τα nodes που θα δείχνει ο `tmp` και πρόκειται να προστεθούν στη λίστα. Ορίζουμε ως νέα τιμή του κόμβου τη μεταβλητή `num` την οποία θα δίνει ο χρήστης κάθε φορά που θα θέλει να εισάγει έναν καινούριο ακέραιο, όπως θα δούμε παρακάτω μέσω της `main`.

Στη συνέχεια εξετάζουμε αν η στοίβα είναι κενή ελέγχοντας αν η `head` ισούται με `NULL`. Στην περίπτωση που είναι κενή, αναθέτουμε στη `head` τη διεύθυνση του καινούριου στοιχείου που προσθέσαμε και τη βάζουμε να δείχνει ως επόμενο στοιχείο το τέλος της λίστας (εφόσον είναι το μοναδικό στοιχείο στη λίστα).

Αν η λίστα δεν είναι κενή, ο δείκτης του καινούριου στοιχείου δείχνει στο `head` και έπειτα ορίζουμε ως αρχή της λίστας το καινούριο στοιχείο `tmp`. Ακολουθώντας αυτή τη σειρά με ακρίβεια εξασφαλίζουμε το να μη χαθεί η σύνδεση μεταξύ των στοιχείων, αφού το καινούριο στοιχείο πρώτα συνδέεται με το `head` και μετά παίρνει τη θέση του, διατηρώντας όποια άλλη σύνδεση ακολουθεί άθικτη.

```
void push(int num) {
    Node* tmp;
    tmp = (Node*) malloc(sizeof(Node));
    tmp->number = num;
    if (head == NULL) {
```

```
        head = tmp;
        head->next = NULL;
    }
    else {
        tmp->next = head;
        head = tmp;
    }
    printf("\nNumber added!");
}
```

- **pop()**

Εξετάζουμε αν η στοίβα δεν είναι κενή ελέγχοντας αν η `head` είναι διάφορη της `NULL`. Στην περίπτωση αυτή, αναθέτουμε στη `head` τη διεύθυνση του επόμενου στοιχείου της λίστας, με συνέπεια να εξάγεται το τρέχον στοιχείο. Ενημερώνουμε τον χρήστη αναλόγως.

Αν η λίστα είναι κενή, ενημερώνουμε το χρήστη πως δεν μπορεί να εξάγει κάποιο στοιχείο.

```
void pop()  {
    /* Evaluates if stack isn't empty by checking whether the head doesn't equal NULL.
*/
    /* Head points to the next node and current one is removed */
    if (head != NULL) {
        head = head->next;
        printf("\nNumber removed!");
    }
    /* When head equals NULL and stack is empty, prints relevant message. */
    else {
        printf("\nNothing to remove. Stack is empty.");
    }
}
```

- **display()**

Εξετάζουμε αν η στοίβα είναι κενή ελέγχοντας αν η `head` ισούται με `NULL`. Στην περίπτωση που είναι κενή, εκτυπώνουμε στο χρήστη το αντίστοιχο μήνυμα και η συνάρτηση τερματίζει.

Διαφορετικά, τυπώνουμε το data κομμάτι του κόμβου (τη μεταβλητή `number`) δίνοντας κάθε φορά στην `tmp` την τιμή της διεύθυνσης του επόμενου κόμβου μέχρι τελικά να τυπωθούν όλα τα στοιχεία της λίστας.

```
void display()  {
    Node* tmp;

    if (head == NULL) {
        printf("\nNothing to display. Stack is empty.");
        return;
        }
    else {
```

```
        printf("\nStack contains: ");
        tmp = head;
        while (tmp != NULL){
            printf("%d  ",tmp->number);
            tmp = tmp->next;
            }
        }
    }
```

## 6.2. Παράδειγμα Τρεξίματος

Παραθέτουμε το αρχικό μενού δίνοντας στο χρήστη την επιλογή να

1. προσθέσει έναν καινούριο ακέραιο αριθμό
2. αφαιρέσει έναν αριθμό
3. να δει τα περιεχόμενα της στοίβας ή
4. να τερματίσει το πρόγραμμα.

Για κάθε επιλογή καλούμε την αντίστοιχη συνάρτηση (`push`, `pop`, `display`, `exit`) μέσα σε μια `switch case` με όρισμα τη μεταβλητή `choice` στην οποία αποθηκεύεται η επιλογή του χρήστη.

Στην περίπτωση που ο χρήστης δώσει μια μη έγκυρη επιλογή, τυπώνουμε το αντίστοιχο μήνυμα με προτροπή να ξαναπροσπαθήσει.

```
int main() {
    int choice, number;

    while(1) {
        printf("-------------------------------------\n");
        printf("Stack menu: \n");
        printf("1. Add an integer number to the stack\n");
        printf("2. Remove a number from the stack\n");
        printf("3. View stack's contents\n");
        printf("4. Exit the program\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

    switch(choice) {
        case 1:
            printf("Enter number: ");
            scanf("%d", &number);
            push(number);
            break;
        case 2:
            pop();
            break;
        case 3:
            display();
```

```
            break;
        case 4:
            printf("Bye!");
            exit(0);
        default:
            printf("Invalid choice, please try again.");
        }
    printf("\n\n");
    }
}
```

[ex5] | */images/ex5.png*

# 7. Υπολογισμός Αριθμητικής Παράστασης με Στοίβα

**Χρησιμοποιείστε λειτουργίες push και pop σε στοίβα τύπου LIFO για τον υπολογισμό της έκφρασης 2 + [(10-3) x (8+3)] . Σε κάθε βήμα, να προσδιορίζετε το περιεχόμενο της στοίβας.**

Λύση

Πριν καταφέρουμε να αποτιμήσουμε την έκφραση αυτή, θα πρέπει πρώτα να τη μετατρέψουμε σε δέντρο της παρακάτω μορφής.

[ex6 1] | */images/ex6_1.png*

Αυτό θα γίνει μέσω της `makeTree` που ορίζεται στο βοηθητικό αρχείο `expression_tree.c` στον φάκελο `helpers`, αφότου επεξεργαστούμε τη μορφή της έκφρασης με την `cleanExpression`.

```
char result[MAX_EXPRESSION_LENGTH];
cleanExpression(expression,result);
printf("CLEAN FORM OF EXPRESSION: %s\n",result);
printf("\n");

makeTree(result);
```

Το `MAX_EXPRESSION_LENGTH`, όπως και άλλες σταθερές που θα χρησιμοποιήσουμε κατά τη διάρκεια του προγράμματος, ορίζονται στο αρχείο `constants.h` του φακέλου `helpers`.

## 7.1. `makeTree`

Ορίζουμε τον κόμβο του δέντρου με πεδία έναν χαρακτήρα για το περιεχόμενό του και τρεις δείκτες:

- `above` για τον γονέα του
- `left` για το αριστερό παιδί
- `right` για το δεξί παιδί

```
struct treeNode {
    char value[MAX_DIGIT_NUMBER];

    struct treeNode *above;
    struct treeNode *left;
    struct treeNode *right;
};
```

Εξαιτίας της ογκώδους φύσης του κώδικα δεν έχει γίνει αλλαγή του ορισμού των `struct` με την `typedef` όπως στα προηγούμενα θέματα, αλλά έχει κρατηθεί η αρχική μας αντιμετώπιση.

Ακόμα και τις αριθμητικές τιμές τις αποθηκεύουμε σαν χαρακτήρες, ώστε να μην χρειάζονται διαφορετικά πεδία για τους τελεστές. Όποτε χρειάζεται να αντιμετωπιστεί ένα στοιχείο σαν αριθμός γίνεται η κατάλληλη μετατροπή.

Πριν επιχειρήσουμε να φτιάξουμε το δέντρο φέρνουμε την έκφραση σε μια εύκολα επεξεργάσιμη μορφή.

```
void cleanExpression(char *expression, char *result) {
    int expressionIndex = 0;
    int resultIndex = 0;

    while (expressionIndex < MAX_EXPRESSION_LENGTH) {
        if (!isspace(*(expression+expressionIndex))) {
            if (*(expression+expressionIndex) == '(' || *(expression+expressionIndex)
== '{' || *(expression+expressionIndex) == '[') {
                *(result+resultIndex) = '(';
            } else if (*(expression+expressionIndex) == ')' || *(expression
+expressionIndex) == '}' || *(expression+expressionIndex) == ']') {
                *(result+resultIndex) = ')';
            } else  {
                *(result+resultIndex) = *(expression+expressionIndex);
            }
            resultIndex++;
        }
        expressionIndex++;
    }
}
```

Ουσιαστικά στην τροποποιημένη έκφραση κάθε αγκύλη ή άγκιστρο που μπορεί να έχει δώσει ο χρήστης αντικαθίσταται με την αντίστοιχη παρένθεση και κάθε κενό έχει αφαιρεθεί. Π.χ. το `{2 + [(10 - 3) * (8 + 3)]}` γίνεται `(2+( (10-3)*(8+3)))`.

```
struct treeNode *makeTree(char *expression) {
    int length = strlen(expression);

    char left[length];
    int *leftFlag = (int *) malloc(sizeof(int));
    *leftFlag = 1;

    char right[length];
    int *rightFlag = (int *) malloc(sizeof(int));
    *rightFlag = 1;

    struct treeNode *parent = (struct treeNode*) malloc(sizeof(struct treeNode));
```

```
    // makes certain that the root is set on the first and only the first call of the
function
    // and the outer parentheses are removed
    if (root == NULL) {
        root = (struct treeNode*) malloc(sizeof(struct treeNode));
        root = parent;
        simplify(expression);
    }

    struct treeNode *leftChild = (struct treeNode*) malloc(sizeof(struct treeNode));
    struct treeNode *rightChild = (struct treeNode*) malloc(sizeof(struct treeNode));

    findLeft(expression,left,leftFlag);
    if (*leftFlag) {
        parent->value[0] = *(expression+strlen(left)); // operators are characters
(just 1B long)
        strcpy(leftChild->value,left);

        findRight(expression,right,strlen(left)+1,rightFlag);
    } else {
        parent->value[0] = *(expression+strlen(left)+2);
        leftChild = makeTree(left);

        findRight(expression,right,strlen(left)+3,rightFlag);
    }
    parent->left = leftChild;
    leftChild->above = parent;

    if (*rightFlag) {
        strcpy(rightChild->value,right);
    } else {
        rightChild = makeTree(right);
    }
    parent->right = rightChild;
    rightChild->above = parent;

    return parent;
}
```

Για το `expression` που δεχόμαστε, ορίζουμε έναν κόμβο `parent`. Εκεί σκοπεύουμε, τελικά, να αποθηκευτεί ο "*κεντρικός*" τελεστής της έκφρασης π.χ. στην `(2+( (10-3)*(8+3)))` το `+`.

- Με την `findLeft` βρίσκουμε το αριστερό υποδέντρο του `parent` (`left`) και:
  - αν αποτελείται μόνο από έναν κόμβο (*φύλλο του δέντρου, άρα αριθμός*), τότε
    1. σαν `value` του `parent` θέτουμε τον n+1 χαρακτήρα, οπού n ο αριθμός ψηφίων του αριθμού
    2. θέτουμε το αριστερό παιδί να είναι ο αριθμός
  - αν αποτελείται από παραπάνω κόμβους (*ολόκληρο expression*), τότε
    1. σαν `value` του `parent` θέτουμε τον n+3 χαρακτήρα, οπού n ο αριθμός χαρακτήρων της

αριστερής έκφρασης. *Τα δύο επιπλέον που προσθέτουμε είναι οι εξωτερικές παρενθέσεις, που μέσω της υλοποίησής μας δεν συμπεριλαμβάνονται στην επιστρεφόμενη έκφραση*

2. θέτουμε το αριστερό παιδί να είναι το δέντρο που προκύπτει από το `left` (*αναδρομική κλήση της makeTree*)

- Με την `findRight` βρίσκουμε το δεξί υποδέντρο του `parent` (`right`) και:
  - αν αποτελείται μόνο από έναν κόμβο (*φύλλο του δέντρου, άρα αριθμός*), τότε θέτουμε το δεξί παιδί να είναι ο αριθμός
  - αν αποτελείται από παραπάνω κόμβους, τότε θέτουμε το δεξί παιδί να είναι το δέντρο που προκύπτει από το `right`

Μέσω αυτής της αναδρομικής διαδικασίας, το δέντρο θα σχηματίζεται από το αριστερότερο προς το δεξιότερο παιδί.

### 7.1.1. `findLeft`, `findRight`

```
void findLeft(char *expression, char *left, int *leftFlag) {
    int expressionIndex = 0;
    int leftIndex = 0;
    int length = strlen(expression);

    if (*(expression+expressionIndex) == '(') {
        int nestedParentheses = 0;
        expressionIndex++;
        while (*(expression+expressionIndex) != ')' || nestedParentheses != 0) {
            *(left+leftIndex) = *(expression+expressionIndex);
            if (*(expression+expressionIndex) == '(') {
                nestedParentheses++;
            } else if (*(expression+expressionIndex) == ')') {
                nestedParentheses--;
            }
            leftIndex++;
            expressionIndex++;
        }
        *(left+leftIndex) = '\0';
        *leftFlag = 0;
        expressionIndex++;
    } else {
        while (isdigit(*(expression+expressionIndex))) {
            *(left+leftIndex) = *(expression+expressionIndex);
            leftIndex++;
            expressionIndex++;
        }
        *(left+leftIndex) = '\0';
    }
}
```

Το αριστερό μέρος μίας έκφρασης, με τη δικιά μας υλοποίηση, είτε θα είναι απλά ένας αριθμός είτε

**μία** παρένθεση με όλα τα περιεχόμενά της. Ακριβώς αυτό ελέγχεται στην `findLeft`, δηλαδή:

- αν η έκφραση αρχίζει με `(` τότε ψάχνουμε μέχρι να βρούμε το αντίστοιχο `)` του ίδιου επιπέδου και επιστρέφουμε το περιεχόμενο,
- αλλιώς κρατάμε όλα τα ψηφία του αριθμού και επιστρέφουμε αυτόν

```
void findRight(char *expression, char *right, int startIndex, int *rightFlag) {
    int rightIndex = 0;
    int expressionIndex = startIndex;

    while (*(expression+expressionIndex) != '\0') {
        if (!isdigit(*(expression+expressionIndex))) {
            *rightFlag = 0;
        }
        *(right+rightIndex) = *(expression+expressionIndex);
        expressionIndex++;
        rightIndex++;
    }
    *(right+rightIndex) = '\0';
    simplify(right);
}
```

Το δεξί μέρος μίας έκφρασης είναι **οτιδήποτε** περισσεύει μετά την αποτίμηση του αριστερού μέρους και του τελεστή που το ακολουθεί.

Όπως και στην πρώτη κλήση της `makeTree`, η `simplify` χρησιμοποιείται για να αφαιρείται το πιθανό ζεύγος παρενθέσεων που περικλείει ολόκληρη την έκφραση, πριν αυτή επιστραφεί.

## 7.2. traverseFromRoot

```
traverseFromRoot(evaluatedExpression);
int length = sizeof(evaluatedExpression) / (MAX_EXPRESSION_LENGTH * sizeof(char));
printf("TREE FORM OF EXPRESSION (POST-ORDER): ");
for (int i=0; i<length; i++) {
    printf("%s ", evaluatedExpression[i]);
}
printf("\n");
```

Αφότου έχουμε φτιάξει τις συνδέσεις των κόμβων του δέντρου, θα κάνουμε τη μεταδιατεταγμένη διάσχιση και το αποτέλεσμα θα το αποθηκεύσουμε στην `evaluatedExpression[MAX_EXPRESSION_LENGTH][MAX_EXPRESSION_LENGTH];` που έχει δηλωθεί `globally`.

Η `evaluatedExpression` είναι διδιάστατος πίνακας, καθώς τους αριθμούς με παραπάνω από ένα ψηφία τους έχουμε αποθηκευμένους σαν πίνακες χαρακτήρων και, γι' αυτό το λόγο, οτιδήποτε άλλο σαν πίνακα ενός κελιού.

Η μεταδιατεταγμένη διάσχιση (`POST ORDER`) γίνεται μέσω της `traverseFromRoot`, η οποία απλά καλεί την `traverseTree` με κόμβο την ρίζα.

```
int part = 0;
void traverseTree(struct treeNode* node, char result[][MAX_EXPRESSION_LENGTH]) {
    if (node->left != NULL) {
        traverseTree(node->left,result);
    } else {
        strcpy(result[part],node->value);
        part++;
        return;
    }

    if (node->right != NULL) {
        traverseTree(node->right,result);
    } else {
        strcpy(result[part],node->value);
        part++;
        return;
    }

    strcpy(result[part],node->value);
    part++;
}
```

Ουσιαστικά ο αλγόριθμος που ακολουθείται για κάθε δοσμένο κόμβο είναι ο εξής:

1. Δες αν ο κόμβος έχει αριστερό παιδί που δεν έχουμε ήδη επισκεφτεί.
    a. Αν ναι, πήγαινε στο βήμα 1 με κόμβο το αριστερό παιδί.
    b. Αν όχι, αποθήκευσε την τιμή του κόμβου και θέσε για κόμβο τον γονέα του.
2. Δες αν ο κόμβος έχει δεξί παιδί που δεν έχουμε ήδη επισκεφτεί.
    a. Αν ναι, πήγαινε στο βήμα 1 με κόμβο το δεξί παιδί.
3. Αποθήκευσε την τιμή του κόμβου.
    a. Αν έχει γονέα, θέσε για κόμβο τον γονέα του και πήγαινε στο βήμα 2.
4. Τερμάτισε.

Τελικά το αναμενόμενο αποτέλεσμα εμφανίζεται στην οθόνη.

[ex6 2] | */images/ex6_2.png*

## 7.3. evaluateResult

Αρχικά δηλώνουμε τις συναρτήσεις που θα χρησιμοποιηθούν στη συνέχεια του προγράμματος σε κατάλληλο header file.

```
#ifndef EXPRESSION_STACK_H
#define EXPRESSION_STACK_H

#include "constants.h"

void print_stack(int x);
void push(int number);
int pop();
void evaluateResult(char evaluatedExpression[MAX_EXPRESSION_LENGTH
][MAX_EXPRESSION_LENGTH]);

#endif
```

Δηλώνουμε τη στοίβα ακεραίων με μέγεθος `STACK_SIZE` (*έχει οριστεί στο αρχείο* `constants.h` *με μέγεθος 10*). Ορίζουμε την global μεταβλητή `top` για να μπορέσουμε να ελέγξουμε τη χωρητικότητα της στοίβας.

```
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include "expression_stack.h"

int stack[STACK_SIZE];
int top = -1;
```

Ορίζουμε τη συνάρτηση `evaluateResult` με όρισμα τον δισδιάστατο πίνακα χαρακτήρων `evaluatedExpression`. Σε αυτή ελέγχουμε αν τα στοιχεία του πίνακα είναι ψηφία ή μαθηματικά σύμβολα (`+`, `-`, `*`, `/`) με χρήση της συνάρτησης `isdigit`.

- Αν το στοιχείο είναι ψηφίο, το μετατρέπουμε σε αριθμό με χρήση της συνάρτησης `atoi` και το κάνουμε `push` στη στοίβα.
- Αν το στοιχείο είναι σύμβολο εξάγουμε τα δύο στοιχεία της στοίβας (αριθμοί) που εισήχθησαν τελευταία και τα αποθηκεύουμε προσωρινά στις μεταβλητές `temp1`, `temp2`. Εκτελούμε τη μεταξύ τους πράξη και αποθηκεύουμε το αποτέλεσμα στη μεταβλητή `res`, την οποία στη συνέχεια κάνουμε `push` στη στοίβα.
- Αν το στοιχείο είναι ο NULL χαρακτήρας (`\0`) , ο έλεγχος τερματίζεται.

```
void evaluateResult(char evaluatedExpression[MAX_EXPRESSION_LENGTH
][MAX_EXPRESSION_LENGTH]) {
    int digit, j;
    int temp1, temp2;
    int res;

    for (j=0; j<MAX_EXPRESSION_LENGTH; j++) {
        if (isdigit(evaluatedExpression[j][0])) {
           digit = atoi(evaluatedExpression[j]);
```

```
            push(digit);
        }
        else {
            if (evaluatedExpression[j][0] == '+') {
                temp1 = pop();
                temp2 = pop();
                res = temp1 + temp2;
                push(res);
            } else if (evaluatedExpression[j][0] == '-') {
                temp1 = pop();
                temp2 = pop();
                res = temp2 - temp1;
                push(res);
            } else if (evaluatedExpression[j][0] == '*') {
                temp1 = pop();
                temp2 = pop();
                res = temp1 * temp2;
                push(res);
            } else if (evaluatedExpression[j][0] == '/') {
                temp1 = pop();
                temp2 = pop();
                res = temp1 / temp2;
                push(res);
            } else if (evaluatedExpression[j][0] == '\0') {
                break;
            }
        }
        print_stack(j);
    }
}
```

Σε κάθε μία από τις παραπάνω επαναλήψεις, τυπώνουμε τα περιεχόμενα της στοίβας καλώντας τη συνάρτηση `print_stack`.

[ex6 3] | */images/ex6_3.png*

# 8. Πηγαίος κώδικας

## 8.1. Θέμα 1

```
#include <stdio.h>

void update(float S[], float c, int i) {
    *(S+i) = c;
}

void retrieve(float S[], float *c, int i) {
    *c = S[i];
}

int main() {
    float A[10];
    float B[10];
    float C[10];

    float entry;
    float retrievedB,retrievedC;

    for (int i = 0; i<10; i++) {
        printf("Give element %d of B: ",i+1);
        scanf("%f",&entry);
        update(B,entry,i);
        printf("Give element %d of C: ",i+1);
        scanf("%f",&entry);
        update(C,entry,i);

        retrieve(B,&retrievedB,i);
        retrieve(C,&retrievedC,i);
        update(A,retrievedB+retrievedC,i);
    }

    float retrievedA;
    for (int i = 0; i<10; i++) {
        retrieve(A, &retrievedA, i);
        printf("Element %d of A:=B+C is %f\n",i+1,retrievedA);
    }

    return 0;
}
```

## 8.2. Θέμα 2

```
#include <stdio.h>
```

```c
#include <stdlib.h>
#include <stdbool.h>

#define base 100
#define length 6

/*global variable that holds the dimension of the user's array*/
int dim;

/* a pointer to a vector of pointers each one of which point to the
 lower and upper bound indexes of the user's array */
int** boundsPtr;

/* 2D array that will store all valid coordinates*/
int** indexes;

bool getUserInput();
int findTotal();
void findIndexes(int guide, int temp[]);
void findCoefficients(int coefficients[]);
void findAddress(int total, int coefficients[]);

int main() {
    if (getUserInput()){
        for (int i=0; i<dim; i++) {
            printf("===Row %d===\n",i+1);
            printf("Lower bound: %d, Upper bound: %d\n",boundsPtr[i][0],boundsPtr[i][
1]);
            printf("\n");
        }

    int total = findTotal();

        /* initialize recursion */
        int coefficients[dim+1];
    findCoefficients(coefficients);

    indexes = (int**) malloc(sizeof(int) * total * dim);
    for (int i = 0; i < total; i++) {
        indexes[i] = (int*) malloc(sizeof(int) * dim);
    };
    int temp[dim];

        findIndexes(0, temp);
        findAddress(total, coefficients);
    }
    return 0;
}

bool getUserInput(){
    int i,j;
```

```
    printf("Type the array's dimension: ");
    scanf("%d", &dim);

    /* user input validation */
    if (dim <= 0){
        printf("Invalid input. \n");
        return false;
    }
    else{
        /* dynamic allocation of the pointer (the rows of the 2D array
representation)*/
        boundsPtr = (int**)malloc(sizeof(int)*dim);
        for(i=0;i<dim;i++){
            /* dynamic allocation of a pointer (the columns of the 2D array
representation)*/
            boundsPtr[i] = (int*)malloc(sizeof(int)*2);
            for (j=0;j<2;j++){
                printf("Type index %d of dimension %d\n",j+1,i+1);
                scanf("%d",&boundsPtr[i][j]);
                if (j%2 == 1){
                    /* user input validation */
                    if (boundsPtr[i][j-1]>boundsPtr[i][j]){
                        printf("invalid input.\n");
                        return false;
                    }
                }
            }


        }
        return true;
    }
}

/* find total number of elements of multidimensional array */
int findTotal() {
    int total = 1;
    for (int i = 0; i < dim; i++) {
        total *= (boundsPtr[i][1] - boundsPtr[i][0] + 1);
    }
    return total;
}

/* find recursively the valid indexes of all the elements of the user's array and
calculate their address*/
int count = 0; // global counter for entries in indexes, to avoid unexpected behaviour
during recursion
void findIndexes(int guide, int temp[]) {
    for (int i = boundsPtr[guide][0]; i <= boundsPtr[guide][1]; i++) {
        temp[guide] = i;
```

```
        if (guide == dim - 1) {
            for (int j = 0; j < dim; j++) {
                indexes[count][j] = temp[j];
            }
            count++;
        } else {
            findIndexes(guide+1, temp);
        }
    }
}

/* calculate all the (dimension + 1) coefficients of the given element of the array */
void findCoefficients(int coefficients[]) {
    coefficients[dim] = length;

    for (int i = dim - 1; i > 0; i--) {
        coefficients[i] = (boundsPtr[i][1] - boundsPtr[i][0] + 1) * coefficients[i+1];
    }

    coefficients[0] = base;
    for (int i = 0; i < dim; i++) {
        coefficients[0] -= coefficients[i+1] * boundsPtr[i][0];
    }
}

/* calculate and print the exact memory address of the given element of the array */
void findAddress(int total, int coefficients[]) {
    int addr;
    for (int i = 0; i < total; i++) {
        printf("Address of element ( ");
        addr = coefficients[0];
        for (int j = 0; j < dim; j++) {
            printf("%d ", indexes[i][j]);
            addr += coefficients[j + 1] * indexes[i][j];
        }
        printf(") is %d\n", addr);
    }
}
```

## 8.3. Θέμα 3

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

typedef struct subjectNode {
    char subject[50];
    struct student *firstStudentPtr;
```

```
    struct subjectNode *next;
}
subjectNode;

subjectNode *head;

typedef struct student {
    char id[9];
    char firstName[20];
    char lastName[20];
    char city[20];
    int age;
    char subject[50];

    struct student *next;
}
student;

student *studentArray[3];

subjectNode *subjectExists(char subject[]);
void addStudentToSubject(student *studentPtr);
void readStudent(student *studentPtr);
void readAllStudents();
void readStudentsBySubject(char subject[]);
void insertData();

int main() {
    insertData();

    readAllStudents();

    readStudentsBySubject("Logic Programming");
    readStudentsBySubject("Discrete Maths");
    readStudentsBySubject("Graphs");

    return 0;
}

/* Initial Data Dump */
void insertData() {
    student *s1 = (student*) malloc(sizeof(student));
    strcpy(s1->id,"mppl3000");
    strcpy(s1->firstName,"Nick");
    strcpy(s1->lastName,"Drake");
    strcpy(s1->city,"London");
    s1->age = 26;
    strcpy(s1->subject,"Logic Programming");
    addStudentToSubject(s1);
    studentArray[0] = s1;
```

```
    student *s2 = (student*) malloc(sizeof(student));
    strcpy(s2->id,"mppl3001");
    strcpy(s2->firstName,"John");
    strcpy(s2->lastName,"Doe");
    strcpy(s2->city,"Peristeri");
    s2->age = 20;
    strcpy(s2->subject,"Logic Programming");
    addStudentToSubject(s2);
    studentArray[1] = s2;

    student *s3 = (student*) malloc(sizeof(student));
    strcpy(s3->id,"mppl3002");
    strcpy(s3->firstName,"Mary");
    strcpy(s3->lastName,"Christmas");
    strcpy(s3->city,"Koropi");
    s3->age = 24;
    strcpy(s3->subject,"Discrete Maths");
    addStudentToSubject(s3);
    studentArray[2] = s3;
}

void readAllStudents() {
    printf("***All Students***\n\n");
    for (int i=0; i<10; i++) {
        if (*(studentArray + i) == NULL) {
            break;
        }
        readStudent(*(studentArray + i));
    }
}

void readStudent(student *studentPtr) {
    printf("===Student with id %s===\n",studentPtr->id);
    printf("Name: %s\nSurname: %s\nCity: %s\nAge: %d\nSubject: %s\n",studentPtr-
>firstName,studentPtr->lastName,studentPtr->city,studentPtr->age,studentPtr->subject);
    printf("\n");
    return;
}

/**
* Checks whether a subject is already inserted
* in the subjects linked list.
*
* Parameters
*   subject - name of the subject to-be-added
*/
subjectNode *subjectExists(char subject[]) {
    subjectNode *subjectPtr = head;
    while (subjectPtr != NULL) {
        if (strcmp(subjectPtr->subject,subject) == 0) {
            return subjectPtr;
```

```
        }
        subjectPtr = subjectPtr->next;
    }
    return NULL;
}

/**
* Adds a student to the linked list of a
* subject's student list.
*
* Parameters
*   studentPtr - pointer to the student to-be-added
*/
void addStudentToSubject(student *studentPtr) {
    subjectNode *p = subjectExists(studentPtr->subject);
    if (p == NULL) {
        subjectNode *newSubjectPtr = (subjectNode*) malloc(sizeof(subjectNode));
        strcpy(newSubjectPtr->subject,studentPtr->subject);
        newSubjectPtr->firstStudentPtr = studentPtr;
        newSubjectPtr->next = head;
        head = newSubjectPtr;
    } else {
        student *studentAttendingSubjectPtr = p->firstStudentPtr;
        while (studentAttendingSubjectPtr->next != NULL) {
            studentAttendingSubjectPtr = studentAttendingSubjectPtr->next;
        }
        studentAttendingSubjectPtr->next = studentPtr;
    }
}

/**
* Given the name of a subject, prints the information of all students studying it.
* In case the subject doesn't exist, informs the user accordingly.
*
* Parameters
*   subject - name of the subject
*/
void readStudentsBySubject(char subject[]) {
    subjectNode *subjectPtr = subjectExists(subject);

    if (subjectPtr == NULL) {
        printf("***Subject Does Not Exist***\n\n");
        return;
    }
    printf("***%s***\n\n",subject);

    student *studentPtr = subjectPtr->firstStudentPtr;
    while (studentPtr != NULL) {
        readStudent(studentPtr);
        studentPtr = studentPtr->next;
    }
```

```
}
```

## 8.4. Θέμα 4

```
#include <stdio.h>
#include <stdlib.h>

/* Initializes Stack */
#define stackLength 20
int stack[stackLength];
int topIndex = -1;

void push(int num);
void pop();
void display();

int main() {
    int choice, number;

    /* Presents a menu that requests user input in the form of an endless loop that
will stop when exit option 4 is selected*/
    while(1) {
            printf("-------------------------------------\n");
        printf("Stack menu: \n");
        printf("1. Add an integer number to the stack\n");
        printf("2. Remove a number from the stack\n");
        printf("3. View stack's contents\n");
        printf("4. Exit the program\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch(choice) {
            case 1:
                    printf("Enter number: ");
                    scanf("%d", &number);
                    push(number);
                    break;
                case 2:
                    pop();
                    break;
                case 3:
                    display();
                    break;
                case 4:
                    printf("Bye!\n");
                    exit(0);
                default:
                    printf("Invalid choice, please try again.");
        }
```

```
            printf("\n\n");
        }
}

/* Declares Stack's Functions */
void push(int num) {
    /* Evaluates if stack is full by checking whether the top matches the stack's
length */
    if (topIndex > (stackLength - 1)){
            printf("\nProcess failed. Stack is full.");
    }
    /* When stack isn't full, the top index of the stack advances by one with num as
its data */
    else {
            topIndex++;
            stack[topIndex] = num;
            printf("\nNumber added!");
    }
}

void pop() {
    /* Evaluates if stack is empty by checking whether the top matches its initial
value */
    if (topIndex == -1) {
        printf("\nNothing to remove. Stack is empty.");
    }
    /* When stack isn't empty, the top index of the stack lowers by one and so the
stack's last inserted data is removed */
    else {
        topIndex--;
        printf("\nNumber removed!");
    }
}

void display() {
    int i;
    /* Evaluates if stack is empty by checking whether the top matches its initial
value and prints relevant message */
    if (topIndex == -1) {
        printf("\nNothing to display. Stack is empty.");
        return;
    }
    /* When stack isn't empty, it loops through all the stack's elements to print them
*/
    else {
        printf("\nStack contains: ");
        for (i=topIndex;i>=0;i--) {
            printf("%d  ",stack[i]);
        }
    }
```

```
}
```

## 8.5. Θέμα 5

```
#include <stdio.h>
#include <stdlib.h>

/* Declares a linked list's node structure containing a data part and its pointer with
the address of the next node  */
typedef struct Node {
    int number;
    struct Node* next;
}
Node;

Node* head;

void push(int num);
void pop();
void display();

int main() {
    int choice, number;

    /* Presents a menu that requests user input in the form of an endless loop that
will stop when exit option 4 is selected*/
    while(1) {
            printf("-------------------------------------\n");
        printf("Stack menu: \n");
        printf("1. Add an integer number to the stack\n");
        printf("2. Remove a number from the stack\n");
        printf("3. View stack's contents\n");
        printf("4. Exit the program\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

    switch(choice) {
        case 1:
            printf("Enter number: ");
            scanf("%d", &number);
            push(number);
            break;
            case 2:
            pop();
            break;
        case 3:
            display();
            break;
        case 4:
```

```
            printf("Bye!\n");
            exit(0);
        default:
            printf("Invalid choice, please try again.");
    }

    printf("\n\n");
    }
}

/* Declares Stack's Functions */
void push(int num) {
    Node* tmp;
        /* allocates memory dynamically for any new nodes that will be added */
    tmp = (Node*) malloc(sizeof(Node));
    /* data part of new node contains the value of int num */
    tmp->number = num;
        /* Evaluates if stack is empty by checking whether the head equals NULL.
        * If so, the head points to the new node
        * And the head's next points to the address of NULL */
        if (head == NULL) {
        head = tmp;
        head->next = NULL;
    }
        /* When not empty, the new node points to the address that head points to
        * and then head points to the new node */
        else {
        tmp->next = head;
        head = tmp;
    }
        printf("\nNumber added!");
}

void pop() {
        /* Evaluates if stack isn't empty by checking whether the head doesn't equal
NULL. */
        /* Head points to the next node and current one is removed */
        if (head != NULL) {
        head = head->next;
        printf("\nNumber removed!");
        }
        /* When head equals NULL and stack is empty, prints relevant message. */
        else {
            printf("\nNothing to remove. Stack is empty.");
        }
}

void display() {
    Node* tmp;
    /* Evaluates if stack is empty by checking whether the head equals NULL and prints
relevant message. */
```

```
        if (head == NULL) {
        printf("\nNothing to display. Stack is empty.");
        return;
    }
        /* When opposite, loops through the list until its end and prints all its
elements
        * each time moving forward by pointing to the next element */
    else {
        printf("\nStack contains: ");
        tmp = head;
        while (tmp != NULL){
            printf("%d  ",tmp->number);
            tmp = tmp->next;
        }
    }
}
```

## 8.6. Θέμα 6

ex6.c

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>

#include "helpers/expression_tree.h"
#include "helpers/expression_stack.h"

char evaluatedExpression[MAX_EXPRESSION_LENGTH][MAX_EXPRESSION_LENGTH];

int main() {

    char expression[MAX_EXPRESSION_LENGTH];
    printf("GIVE AN ARITHMETIC EXPRESSION: ");
    fgets(expression,MAX_EXPRESSION_LENGTH,stdin);
    printf("\n");

    char result[MAX_EXPRESSION_LENGTH];
    cleanExpression(expression,result);
    printf("CLEAN FORM OF EXPRESSION: %s\n",result);
    printf("\n");

    makeTree(result);
    traverseFromRoot(evaluatedExpression);
    int length = sizeof(evaluatedExpression) / (MAX_EXPRESSION_LENGTH * sizeof(char));
    printf("TREE FORM OF EXPRESSION (POST-ORDER): ");
    for (int i=0; i<length; i++) {
        printf("%s ", evaluatedExpression[i]);
```

```
    }
    printf("\n");

    printf("===STACK RESULT EVALUATION===\n");
    evaluateResult(evaluatedExpression);

    return 0;
}
```

helpers/constants.h

```
#ifndef CONSTANTS_H
#define CONSTANTS_H

#define MAX_EXPRESSION_LENGTH 50
#define MAX_DIGIT_NUMBER 5
#define STACK_SIZE 10

#endif
```

helpers/expression_tree.h

```
#ifndef EXPRESSION_TREE_H
#define EXPRESSION_TREE_H

#include "constants.h"

struct treeNode {
    char value[MAX_DIGIT_NUMBER];

    struct treeNode *above;
    struct treeNode *left;
    struct treeNode *right;
};

/**
* Returns a numeric expression where all white spaces are trimmed
* and all parentheses, brackets and braces are replaced by parentheses.
*
* Parameters
*   expression - pointer to the initial string
*   result - pointer to the result
*/
void cleanExpression(char *expression, char *result);

/**
* Removes the outer parentheses of an expression.
*
* Parameters
```

```
*   expression - pointer to the given expression
*
* WARNING: TREATS THE EXPRESSION (STRING i.e. character array) AS A MUTABLE OBJECT
*/
void simplify(char *expression);

/**
* Returns the left operand of a numeric expression and
* informs whether it was an expression or a number.
* e.g.
*   if expression pointed to [1-(2*5)]+3-[(2-4)/3],
*   left would point to 1-(2*5) and leftFlag to 0
*
* Parameters
*   expression - pointer to the expression
*   left - pointer to the left operand
*   leftFlag - pointer to 1 if left is just a number, otherwise to 0
*/
void findLeft(char *expression, char *left, int *leftFlag);

/**
* Returns the right operand of a numeric expression and
* informs whether it was an expression or a number.
* e.g.
*   if expression pointed to [1-(2*5)]+3-[(2-4)/3],
*   right would point to 3-[(2-4)/3] and rightFlag to 0
*
* Parameters
*   expression - pointer to the expression
*   right - pointer to the right operand
*   startIndex - index after which we should evaluate the right operand (depending on
left operand length)
*   rightFlag - pointer to 1 if right is just a number, otherwise to 0
*/
void findRight(char *expression, char *right, int startIndex, int *rightFlag);

/**
* Recursively creates a tree representation of a given expression
*
* Parameters
*   expression - pointer to the expression
*/
struct treeNode *makeTree(char *expression);

/**
* Post-order traversal of a subtree and
* insertion to the evaluated expression array.
*
* Parameters
*   node - the root node of the subtree
*/
```

```
void traverseTree(struct treeNode* node, char result[][MAX_EXPRESSION_LENGTH]);

void traverseFromRoot(char result[][MAX_EXPRESSION_LENGTH]);

#endif
```

helpers/expresion_tree.c

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>

#include "expression_tree.h"

struct treeNode *root;

void cleanExpression(char *expression, char *result) {
    int expressionIndex = 0;
    int resultIndex = 0;

    while (expressionIndex < MAX_EXPRESSION_LENGTH) {
        if (!isspace(*(expression+expressionIndex))) {
            if (*(expression+expressionIndex) == '(' || *(expression+expressionIndex)
== '{' || *(expression+expressionIndex) == '[') {
                *(result+resultIndex) = '(';
            } else if (*(expression+expressionIndex) == ')' || *(expression
+expressionIndex) == '}' || *(expression+expressionIndex) == ']') {
                *(result+resultIndex) = ')';
            } else  {
                *(result+resultIndex) = *(expression+expressionIndex);
            }
            resultIndex++;
        }
        expressionIndex++;
    }
}

void simplify(char *expression) {
    int length = strlen(expression);

    if (*(expression+0) == '(' && *(expression+length-1) == ')') {
        for (int expressionIndex=1; expressionIndex<length-1;expressionIndex++) {
            *(expression+expressionIndex-1) = *(expression+expressionIndex);
        }
        *(expression+length-2) = '\0';
    }
}

void findLeft(char *expression, char *left, int *leftFlag) {
```

```
    int expressionIndex = 0;
    int leftIndex = 0;
    int length = strlen(expression);

    if (*(expression+expressionIndex) == '(') {
        int nestedParentheses = 0;
        expressionIndex++;
        while (*(expression+expressionIndex) != ')' || nestedParentheses != 0) {
            *(left+leftIndex) = *(expression+expressionIndex);
            if (*(expression+expressionIndex) == '(') {
                nestedParentheses++;
            } else if (*(expression+expressionIndex) == ')') {
                nestedParentheses--;
            }
            leftIndex++;
            expressionIndex++;
        }
        *(left+leftIndex) = '\0';
        *leftFlag = 0;
        expressionIndex++;
    } else {
        while (isdigit(*(expression+expressionIndex))) {
            *(left+leftIndex) = *(expression+expressionIndex);
            leftIndex++;
            expressionIndex++;
        }
        *(left+leftIndex) = '\0';
    }
}

void findRight(char *expression, char *right, int startIndex, int *rightFlag) {
    int rightIndex = 0;
    int expressionIndex = startIndex;

    while (*(expression+expressionIndex) != '\0') {
        if (!isdigit(*(expression+expressionIndex))) {
            *rightFlag = 0;
        }
        *(right+rightIndex) = *(expression+expressionIndex);
        expressionIndex++;
        rightIndex++;
    }
    *(right+rightIndex) = '\0';
    simplify(right);
}

struct treeNode *makeTree(char *expression) {
    int length = strlen(expression);

    char left[length];
    int *leftFlag = (int *) malloc(sizeof(int));
```

```
    *leftFlag = 1;

    char right[length];
    int *rightFlag = (int *) malloc(sizeof(int));
    *rightFlag = 1;

    struct treeNode *parent = (struct treeNode*) malloc(sizeof(struct treeNode));
    // makes certain that the root is set on the first and only the first call of the
function
    // and the outer parentheses are removed
    if (root == NULL) {
        root = (struct treeNode*) malloc(sizeof(struct treeNode));
        root = parent;
        simplify(expression);
    }

    struct treeNode *leftChild = (struct treeNode*) malloc(sizeof(struct treeNode));
    struct treeNode *rightChild = (struct treeNode*) malloc(sizeof(struct treeNode));

    findLeft(expression,left,leftFlag);
    if (*leftFlag) {
        parent->value[0] = *(expression+strlen(left)); // operators are characters
(just 1B long)
        strcpy(leftChild->value,left);

        findRight(expression,right,strlen(left)+1,rightFlag);
    } else {
        parent->value[0] = *(expression+strlen(left)+2);
        leftChild = makeTree(left);

        findRight(expression,right,strlen(left)+3,rightFlag);
    }
    parent->left = leftChild;
    leftChild->above = parent;

    if (*rightFlag) {
        strcpy(rightChild->value,right);
    } else {
        rightChild = makeTree(right);
    }
    parent->right = rightChild;
    rightChild->above = parent;

    return parent;
}

int part = 0;
void traverseTree(struct treeNode* node, char result[][MAX_EXPRESSION_LENGTH]) {
    if (node->left != NULL) {
        traverseTree(node->left,result);
    } else {
```

```
        strcpy(result[part],node->value);
        part++;
        return;
    }

    if (node->right != NULL) {
        traverseTree(node->right,result);
    } else {
        strcpy(result[part],node->value);
        part++;
        return;
    }

    strcpy(result[part],node->value);
    part++;
}

void traverseFromRoot(char result[][MAX_EXPRESSION_LENGTH]) {
    traverseTree(root,result);
}
```

helpers/expression_stack.h

```
#ifndef EXPRESSION_STACK_H
#define EXPRESSION_STACK_H

#include "constants.h"

void print_stack (int x);
int* getStack();
void push(int number);
int pop ();

/**
* Loops through each row to identify when string is digit with function isdigit
* and repeats push or pop functions based on that
* if a string is a digit, it converts the string to digit with atoi and pushes the
digit to the stack
* if not, it pops the last two digits and calculates their result.
*/
void evaluateResult(char evaluatedExpression[MAX_EXPRESSION_LENGTH
][MAX_EXPRESSION_LENGTH]);

#endif
```

'helpers/expression_stack.c`

```
#include <stdio.h>
#include <ctype.h>
```

```c
#include <stdlib.h>

#include "expression_stack.h"

int stack[STACK_SIZE];
int top = -1;

void print_stack (int x) {
    int i;
    printf("%d. Stack contains: ", x+1);
    for (i=0; i<STACK_SIZE; i++) {
        printf(" %d |", stack[i]);
    }
    printf("\n");
}

int* getStack() {
    return stack;
}

void push(int number) {
    if (top>(STACK_SIZE-1)){
        printf("Stack is Full!\n");
        return;
    } else {
        top++;
        stack[top] = number;
    }
}

int pop () {
    if (top == -1) {
        return 0;
    } else {
        int number = stack[top];
        stack[top] = 0;
        top--;
        return number;
        }
}

void evaluateResult(char evaluatedExpression[MAX_EXPRESSION_LENGTH
][MAX_EXPRESSION_LENGTH]) {
    int digit, j;
    int temp1, temp2;
    int res;

    for (j=0; j<MAX_EXPRESSION_LENGTH; j++) {
        if (isdigit(evaluatedExpression[j][0])) {
           digit = atoi(evaluatedExpression[j]);
           push(digit);
```

```
        }
        else {
            if (evaluatedExpression[j][0] == '+') {
                temp1 = pop();
                temp2 = pop();
                res = temp1 + temp2;
                push(res);
            } else if (evaluatedExpression[j][0] == '-') {
                temp1 = pop();
                temp2 = pop();
                res = temp2 - temp1;
                push(res);
            } else if (evaluatedExpression[j][0] == '*') {
                temp1 = pop();
                temp2 = pop();
                res = temp1 * temp2;
                push(res);
            } else if (evaluatedExpression[j][0] == '/') {
                temp1 = pop();
                temp2 = pop();
                res = temp1 / temp2;
                push(res);
            } else if (evaluatedExpression[j][0] == '\0') {
                break;
            }
        }
        print_stack(j);
    }
}
```