ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΑ

ΣΧΟΛΗ ΤΕΧΝΟΛΟΓΙΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΕΠΙΚΟΙΝΩΝΙΩΝ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΑΠΑΛΛΑΚΤΙΚΗ ΕΡΓΑΣΙΑ ΣΤΟ ΜΑΘΗΜΑ

ΔΟΜΕΣ ΔΕΔΟΜΕΝΩΝ

ΑΛΒΕΡΤΗΣ ΜΗΝΑΣ
Α.Μ.: ΜΠΠΛ2204

ΚΑΛΤΣΟΥΝΗ ΕΛΕΝΗ
Α.Μ.: ΜΠΠΛ2218

ΣΚΥΛΟΓΙΑΝΝΗΣ ΘΩΜΑΣ
Α.Μ.: ΜΠΠΛ2241

ΑΘΗΝΑ ΜΑΡΤΙΟΣ 2023
ΕΞΑΜΗΝΟ Α’

Table of Contents

1. 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.

```
for (int i = 0; i<10; i++) {
    scanf("%f",&entry);
    update(B,entry,i);
    scanf("%f",&entry);
    update(C,entry,i);

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

for (int i = 0; i<10; i++) {
    retrieve(A, &retrievedA, i);
}
```

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.

2. 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.

2.1. Εισαγωγή Δεδομένων

```
boundsPtr = (int**)malloc(sizeof(int)*dim);
for(i=0;i<dim;i++){
    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){
            if (boundsPtr[i][j-1]>boundsPtr[i][j]){
                printf("invalid input.\n");
                return false;
            }
        }
    }
}
```

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.

2.2. Τετράδες Συντεταγμένων

```
int count = 0;
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 τα δοσμένα τους όρια.

2.3. Υπολογισμός Θέσεων Μνήμης

```
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];
    }
}

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

3. Σύστημα Αποθήκευσης Στοιχείων Φοιτητών

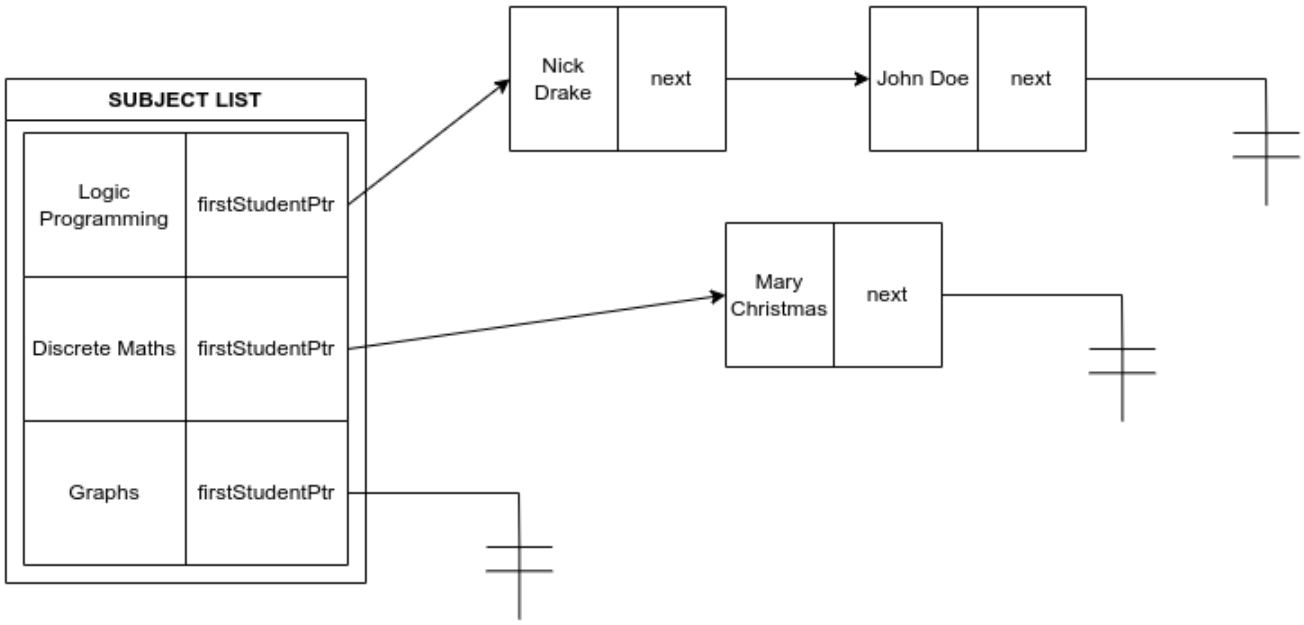
Σε μια συλλογή στοιχείων, κάθε στοιχείο περιέχει πληροφορίες για τους φοιτητές του μεταπτυχιακού, όπως αριθμό μητρώου, επώνυμο, όνομα, πατρώνυμο, διεύθυνση κατοικίας, αριθμό σταθερού τηλεφώνου, αριθμό κινητού τηλεφώνου και επιλεγμένο μάθημα πρώτου εξαμήνου.

- **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`).

3.1. Σχέση Μαθημάτων και Μαθητών



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

Ορίζουμε χειροκίνητα τρεις μαθητές και προσθέτουμε τα στοιχεία τους στον ζητούμενο πίνακα,

ακολουθώντας το παραπάνω διάγραμμα.

3.2. 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.

```
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.

3.3. 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`).

```
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;
    }
}
```

4. 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.

4.1. 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.

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

- **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!");
    }
}
```

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.

4.2. Ακραίες περιπτώσεις

```
-----------------------------------
Stack menu:
1. Add an integer number to the stack
2. Remove a number from the stack
3. View stack's contents
4. Exit the program
Enter your choice: 1
Enter number: 1

Number added!

-----------------------------------
Stack menu:
1. Add an integer number to the stack
2. Remove a number from the stack
3. View stack's contents
4. Exit the program
Enter your choice: 1
Enter number: 1

Process failed. Stack is full.
```

```
-----------------------------------
Stack menu:
1. Add an integer number to the stack
2. Remove a number from the stack
3. View stack's contents
4. Exit the program
Enter your choice: 3

Nothing to display. Stack is empty.

-----------------------------------
Stack menu:
1. Add an integer number to the stack
2. Remove a number from the stack
3. View stack's contents
4. Exit the program
Enter your choice: 2

Nothing to remove. Stack is empty.
```

5. 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 κόμβου της λίστας.

5.1. Βασικές Λειτουργίες

- **push(int num)**

Υλοποιούμε τη συνάρτηση `push` με όρισμα την ακέραια μεταβλητή `num`, την οποία αναθέτουμε ως τιμή ενός νέου προσωρινού κόμβου. Εξετάζουμε αν η στοίβα είναι κενή ελέγχοντας αν η `head` ισούται με `NULL`.

- Στην περίπτωση που είναι κενή, αναθέτουμε στη `head` τη διεύθυνση του νέου κόμβου 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.

```
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 είναι κενή ελέγχοντας αν η `head` είναι διάφορη της `NULL`.

- Αν είναι, ενημερώνουμε το χρήστη πως δεν μπορεί να εξάγει κάποιο στοιχείο.
- Αν δεν είναι, αναθέτουμε στη `head` τη διεύθυνση του επόμενου στοιχείου της λίστας, με συνέπεια να εξάγεται το τρέχον στοιχείο.

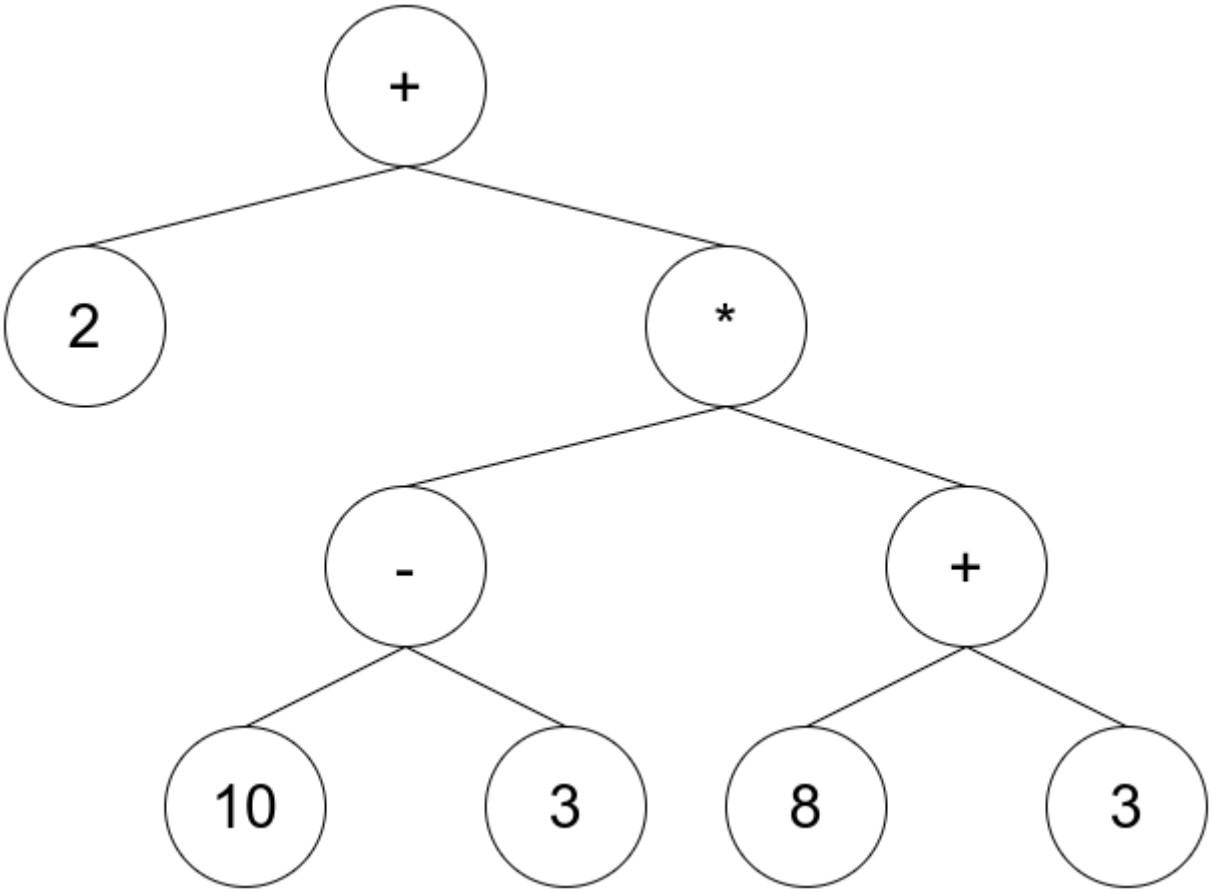
```
void pop()  {
    if (head != NULL) {
        head = head->next;
        printf("\nNumber removed!");
    }
    else {
        printf("\nNothing to remove. Stack is empty.");
    }
}
```

6. 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.



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`.

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

6.1. `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

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 αφαιρείται. 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));
```

```
    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.

6.1.1. `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.

6.2. `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.

```
GIVE AN ARITHMETIC EXPRESSION: {2 + [ (10 - 3) * ( 8 + 3 ) ] }

CLEAN FORM OF EXPRESSION: (2+((10-3)*(8+3)))

TREE FORM OF EXPRESSION (POST-ORDER): 2 10 3 - 8 3 + * +
```

6.3. `evaluateResult`

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.

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.

```
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;
        }
    }
}
```

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