Εργασία 3 – Σκελετοί Λύσεων

Άσκηση 1

Χρησιμοποιούμε τη δομή

```
typedef struct TNode{
      int key;
      struct TNode *left;
      struct TNode *right;
} tnode;
```

και υποθέτουμε πως ένα δυαδικό δένδρο είναι υλοποιημένο ως δείκτης στη ρίζα του δένδρου, δηλαδή, έχει τύπο `*tnode`.

Θα δημιουργήσουμε μια αναδρομική διαδικασία η οποία επιστρέφει δυάδες της μορφής (τύπος δένδρου: int, ύψος δένδρου: int). Για τον τύπο δένδρου χρησιμοποιούμε την τιμή 1 για να δείξουμε ότι ένα δένδρο είναι πλήρες, την τιμή 2 για να δείξουμε ότι ένα δένδρο είναι τέλειο, και την τιμή 0 για να δείξουμε ότι δεν είναι κανένα από τα δύο.

Η ιδέα είναι η εξής: Ξεκινώντας από τη ρίζα του δένδρου υπάρχουν δύο περιπτώσεις:

- Αν η ρίζα του δένδρου είναι κενή, τότε το δένδρο είναι τέλειο και επιστρέφουμε την δυάδα (2, 0), δηλαδή το δένδρο είναι ένα τέλειο δένδρο ύψους 0.
- Διαφορετικά, κάνουμε δύο αναδρομικές κλήσεις στα δύο υπόδενδρα του δένδρου για να πάρουμε πληροφορίες σχετικά με τον τύπο και το ύψος τους. Συνδυάζουμε τις πληροφορίες αυτές για να αποφασίσουμε σχετικά με τον τύπο και το ύψος του αρχικού δένδρου.

```
(int, int) RecPlires(tnode *p){
      if(p == NULL)
            return (2,0);
      else
            (t1,h1) = Plires(p->left);
            (t2,h2) = Plires(p->right) ;

            if (t1 == 2, t2 == 2, h1 == h2)
                  return (2, h1 + 1) ;
            if (t1 == 2, t1 == 1, h1 == h2)
                  return (1, h1 + 1) ;
            if (t1 == 2, t1 == 2, h1 == h2 + 1)
                  return (1, h1 + 1) ;
            if (t1 == 1, t2 == 2, h1 == h2 + 1)
                  retutn (1, h1 + 1) ;
            else
                  return (0, max(h1,h2) + 1);
 }
```

Η πιο κάτω διαδικασία καλεί τη αναδρομική αυτή συνάρτηση τυπώνοντας στην οθόνη πληροφορίες για το αρχικό δένδρο.

```
Plires(tnode *p){
      (t, h) = RecPlires(p) ;
      if (t != 0)
            printf ("To dendro einai plhres")
}
```

Άσκηση 2

Χρησιμοποιούμε τη δομή

```
typedef struct Node{
            int key;
            struct node *left;
            struct node *right;
} node;
```

και υποθέτουμε πως ένα δυαδικό δένδρο είναι υλοποιημένο ως δείκτης στη ρίζα του δένδρου, δηλαδή, έχει τύπο *node.

Το ζητούμενο υλοποιείται με την πιο κάτω αναδρομική διαδικασία η οποία ακολουθεί πιστά τον αναδρομικό ορισμό των όμοιων δένδρων.

```
int SimilarR(node *p, node *q){
      if (p == NULL AND q == NULL)
            return 1;
      if (p == NULL OR q == NULL)
            return 0;
      else
            return (Similar(p->left, q->left)
                        AND Similar(p->right, q->right)
}
```

Η αναδρομική διαδικασία καλείται μια φορά σε κάθε κόμβο, επομένως ο χρόνος εκτέλεσής της είναι Θ(n) όπου n είναι ο αριθμός των κόμβων του δένδρου.

Η μη-αναδρομική εκδοχή της διαδικασία απαιτεί τη χρήση βοηθητικής δομής στην οποία θα αποθηκεύουμε ζεύγη κατευθύνσεων προς τα οποία θα πρέπει να κινηθούμε σε μεταγενέστερο στάδιο. Ο τύπος της βοηθητικής δομής δεν έχει σημασία αφού η επεξεργασία των ζευγών μπορεί να γίνει σε τυχαία σειρά. Στη λύση αυτή χρησιμοποιείται στοίβα. Η βασική ιδέα του αλγόριθμου είναι η εξής:

1. Δημιουργούμε μια κενή στοίβα.
2. Εφόσον τα δύο δένδρα που μελετούμε δεν είναι κενά και η στοίβα δεν είναι άδεια (δηλ. υπάρχουν ακόμη ζεύγη προς επεξεργασία) προχωρούμε στο βήμα 3. Διαφορετικά πάμε στο βήμα 5.
3. Αν το ένα από τα δύο δένδρα είναι κενά, επιστρέφουμε 0 δηλώνοντας έτσι ότι τα δένδρα είναι ανόμοια και τερματίζουμε. Διαφορετικά τοποθετούμε δείκτες προς τα δεξιά παιδιά των δύο κόμβων και προχωρούμε στα αριστερά παιδιά, επιστρέφοντας στο βήμα 2.
4. Αν και τα δύο δένδρα είναι κενά (NULL δείκτες) συμπεραίνουμε ότι μέχρι στιγμής δεν έχουμε αντιμετωπίσει ανόμοια δένδρα και ανασύρουμε από τη στοίβα κάποιο από τα ζεύγη που αναμένει επεξεργασία. Επαναλαμβάνουμε από το βήμα 2.
5. Επιστρέφουμε την τιμή 1 που υπονοεί ότι τα αρχικά δένδρα είναι όμοια και τερματίζουμε.

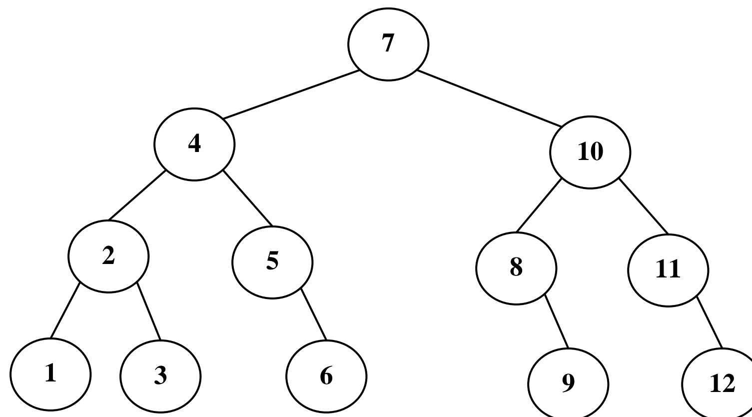
```
int SimilarNR(node *p, node *q){
stack S;
MakeEmpty(S);
while (p != NULL OR q!= NULL OR !IsEmpty(S)){
      if (p != NULL AND q == NULL) || (p == NULL AND q != NULL)
```

```
            return 0;
      if (p != NULL AND q!= NULL)
            Push((p->right, q->right), S);
            p = p->left;
            q = q->left;
      else
            (p,q) = Pop(S);
}
return 1;
}
```

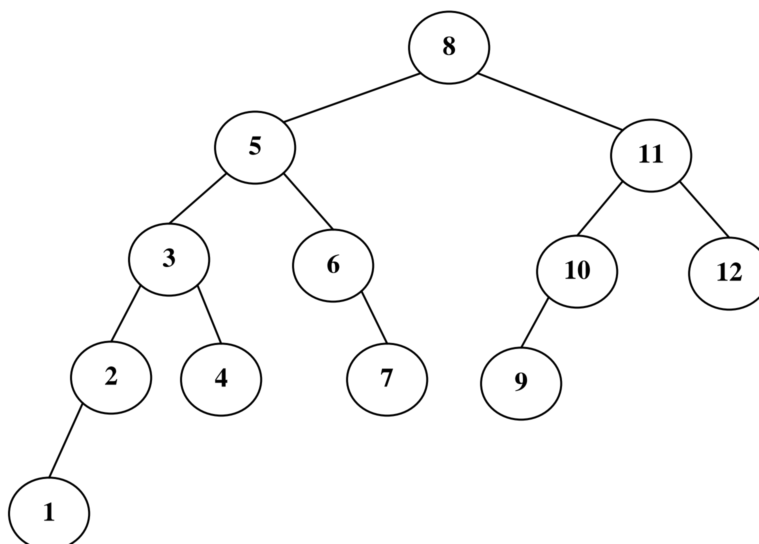
Η διαδικασία επισκέπτεται και πάλι κάθε κόμβο ακριβώς μια φορά, επομένως ο χρόνος εκτέλεσής της είναι Θ(n) όπου n είναι ο αριθμός των κόμβων του δένδρου.

Άσκηση 3

Ακολουθεί ένα AVL-δένδρο με τα στοιχεία 1-12 και το ελάχιστο δυνατό ύψος. Το δένδρο αυτό μπορεί να παραχθεί από τη σειρά εισαγωγών: 7, 4, 10, 2, 5, 8, 12, 1, 3, 6, 9, 11.

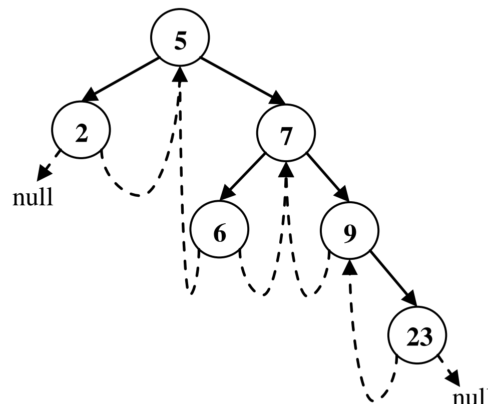


Ακολουθεί ένα AVL-δένδρο με τα στοιχεία 1-12 και το ελάχιστο δυνατό ύψος. Το δένδρο αυτό μπορεί να παραχθεί από τη σειρά εισαγωγών: 8, 5, 11, 3, 6, 10, 12, 2, 4, 7, 9, 1.



Άσκηση 4

(α) Πιο κάτω δείχνεται η νηματώδης μορφή του δένδρου που παρουσιάζεται στην εκφώνηση της άσκησης.



(β) Για διαχωρισμό ανάμεσα στους δείκτες παιδιά και τους δείκτες νήματα ενός κόμβου χρησιμοποιούμε ένα χαρακτήρα σε κάθε κόμβο ο οποίος παίρνει την τιμή ‘n’ αν κανένα από τα παιδιά του κόμβου δεν είναι νήμα, ‘l’ αν το αριστερό παιδί είναι νήμα, ‘r’ αν το δεξί παιδί είναι νήμα και ‘b’ αν και τα δύο παιδιά είναι νήματα.

Χρησιμοποιούμε τις δομές

```
typedef struct TNode{
            int key;
            struct node *left;
            struct node *right;
            char threads;
} tnode;
```

```
typedef struct Tree{
     tnode *root;
} tree;
```

και υποθέτουμε πως ένα νηματώδες δένδρο είναι υλοποιημένο ως δείκτης σε κόμβο τύπου *tree.

(γ) Εισαγωγή σε νηματώδες δένδρο γίνεται παρόμοια με ένα ΔΔΑ. Η επέκταση η οποία καλούμαστε να κάνουμε είναι η απόδοση νημάτων στους κόμβους/φύλλα του δένδρου που δημιουργούνται λόγω των εισαγωγών. Η απόδοση γίνεται ως εξής.

- Αν ο νέος κόμβος δεν έχει πατέρα, τότε τα παιδιά/νήματα έχουν και τα δύο την τιμή NULL.
- Αν ο νέος κόμβος είναι αριστερό παιδί του πατέρα του, τότε κληρονομεί από αυτόν το αριστερό του νήμα, ενώ το δεξί του νήμα δείχνει στον πατέρα.
- Αν ο νέος κόμβος είναι δεξί παιδί του πατέρα του, τότε κληρονομεί από αυτόν το δεξί του νήμα, ενώ το αριστερό του νήμα δείχνει στον πατέρα.

Και στις δύο τελευταίες περιπτώσεις, ο πατέρας πρέπει να ενημερωθεί κατάλληλα τόσο για την απόκτηση νέου παιδιού όσο και για την απώλεια κάποιου νήματος.

```
tnode *InsertNode(tnode *root, int n){
p=(tnode *)malloc(sizeof(tnode));
p->val = n;
p->left = p->right = NULL;
p->threads = ‘b’;
r = root;
```

```
if (r == NULL)
      p->threads = 'b'
      return p;
while (1)
      if (n == r->val)
            report "n already in tree" and exit
      else if (n < r->val && r->threads != 'b', 'l')
            r = r->left;
      else if (n > r->val && r->threads != 'b', 'r')
                  r = r->right;
      else
            break;

if (n < r->val){
      p->left = r->left;
      p->right = r;
      r->left = p;
      if r->threads == 'l'
            r->threads == 'n'
      if r->threads == 'b'
            r->threads == 'r'
}
if (n > r->val)){
      p->right = r->right;
      p->left = r;
      r->right = p;
      if r->threads == 'r'
            r->threads == 'n'
      if r->threads == 'b'
            r->threads == 'l'
}
return root;
}
```

Για τη διαδικασία εξαγωγής χρησιμοποιούμε βοηθητικά τις διαδικασίες DeleteRightMin(r) η οποία εξάγει το ελάχιστο στοιχείο στο δεξί υπόδενδρο του κόμβου r και DeleteLeftMax(r) η οποία εξάγει το μέγιστο στοιχείο στο αριστερό υπόδενδρο του κόμβου r . Πιο κάτω παρουσιάζεται η διαδικασία DeleteRightMin(r). Η διαδικασία αυτή είναι όμοια με την ανάλογη διαδικασία σε ΔΔΑ με κατάλληλες προσθήκες για ενημέρωση των νημάτων. Η σημαντικότερη αλλαγή αφορά το γεγονός ότι ο κόμβος με το ελάχιστο στοιχείο, έστω p, πιθανόν να αποτελεί αριστερό νήμα κάποιου άλλου κόμβου. Ο κόμβος αυτός, αν υπάρχει, είναι ο κόμβος/φύλλο με το αμέσως μεγαλύτερο στοιχείο. Υπάρχει, αν ο p έχει δεξιά υπόδενδρο. (Αν όχι, ο κόμβος με το αμέσως μεγαλύτερο στοιχείο είναι ο πατέρας του p ο οποίος δεν έχει αριστερό νήμα πριν από την εξαγωγή).

```
int deleteRightMin(tnode *r){

father = r;
p = r = r->right;

while (r->threads != 'b', 'l')
      father = r;
```

```
        r = r->left;
if p == r
            father->right = r->right;
            if (r->threads == 'b')
                  if father->threads == 'n'
                        r->threads == 'r'
                  if father->threads == 'l'
                        father->threads == 'b'
      else
            father->left = r->left;
            if (r->threads == 'b')
                  if father->threads == 'n'
                        r->threads == 'l'
                  if father->threads == 'r'
      if (r->right != NULL)
            for(q = r->right; q->left != NULL; q = q->left);
            q->left = r->left;
val = r->val
free(r);
return val;
}
```

Η διαδικασία εξαγωγής περιέχει εφαρμόζει κλήση της DeleteMin όπως και στα ΔΔΑ. Κά κάποια μικρή επιπρόσθετη πολυπλοκότητα έναντι αυτής σε ΔΔΑ εφόσον οι όποιες αναπροσαρμογές των νημάτων γίνονται κατά την κλήση της.

```
void Delete(tree *t,int n){
  r = t->root;
  if (r == null) return NULL;
  father = NULL;
  while (r!=NULL)
      if (n < r->val)
            father = r;
            r = r->left;
      if (n > r->val)
            father = r;
            r = r->right;
      if (n == r->val)
            break;
  if r == NULL return;
  if (father == NULL)
      tree->root = NULL;
      free(r);
  else
      if (r->threads = 'b');
            if (r == father->left)
                  father->left = r->left;
                  if (father->threads = 'n')
                        father->threads = 'l'
```

```
                if (father->threads = 'r')
                     father->threads = 'b'
           if (r == father->right)
                father->right = r->right;
                if (father->threads = 'n')
                     father->threads = 'r'
                if (father->threads = 'l')
                     father->threads = 'b'
           free(r);
      else if (r->right != NULL)
                val = DeleteRightMin(r);
           else
                val = DeleteLeftMax(r);
           r->val = val;
}
```

(δ) Το κύριο πλεονέκτημα ενός νηματώδους δένδρου είναι η ευκολότερη μετακίνηση μέσα στο δένδρο. Συγκεκριμένα, η ενδοδιεταγμένη και η μεταδιατεταγένη διάσχιση δένδρων μπορεί να υλοποιηθεί εύκολα χωρίς αναδρομή και χωρίς τη χρήση στοίβας. Το ίδιο ισχύει (δηλαδή, εύκολη μετατροπή σε μη-αναδρομικές διαδικασίες χωρίς τη χρήση βοηθητικών δομών) για αναδρομικές διαδικασίες που 'μιμούνται' τις διασχίσεις αυτές. Μειονέκτημα είναι η χρήση περισσότερης μνήμης σε κάθε κόμβο του δένδρου και η αύξηση της πολυπλοκότητας των διαδικασιών εισαγωγής και διαγραφής.