Κατ'οίκον Εργασία 2 – Σκελετοί Λύσεων

Άσκηση 1

Έστω l_i ο αριθμός φύλλων που βρίσκονται στο επίπεδο i ενός δυαδικού δένδρου. Θέλουμε να αποδείξουμε την πρόταση:

$$\Pi(h) \equiv \sum_{i=1}^{h+1} \frac{l_i}{2^{i-1}} \leq 1$$

Η απόδειξη μπορεί να γίνει με μαθηματική επαγωγή.

Βασική περίπτωση – h = -1
Προφανώς η περίπτωση αυτή αφορά το κενό δένδρο. Τότε

$$\sum_{i=1}^{0} \frac{l_i}{2^{i-1}} = 0 \leq 1$$

Επομένως το ζητούμενο έπεται.

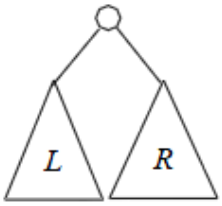
Βασική περίπτωση – h = 0
Προφανώς η περίπτωση αυτή αφορά δένδρο με ένα ακριβώς κόμβο, τη ρίζα. Τότε

$$\sum_{i=1}^{1} \frac{l_i}{2^{i-1}} = \frac{l_1}{2^{1-1}} = \frac{1}{1} \leq 1$$

Επομένως το ζητούμενο έπεται.

Υπόθεση της επαγωγής: *Έστω Π(k) για κάθε k < m.*

Βήμα της επαγωγής: Θα δείξουμε ότι η Π(m) αληθεύει.
Έστω ένα δυαδικό δένδρο με ύψος m. Το δένδρο αυτό αποτελείται από μια ρίζα και δύο μη κενά υπόδενδρα που ριζώνουν σε αυτή. Προφανώς και τα δύο αυτά υπόδενδρα έχουν ύψος μικρότερο του m.



Έστω h_L και h_R τα ύψη του αριστερού και δεξιού υποδένδρου, αντίστοιχα, και l_i^L και l_i^R ο αριθμός φύλλων που βρίσκονται στο επίπεδο i στο αριστερό και στο δεξιό υπόδενδρο αντίστοιχα.

Τότε από την υπόθεση της επαγωγής έχουμε για τα δένδρα L και R ότι

$$\sum_{i=1}^{h_L+1} \frac{l_i^L}{2^{i-1}} \leq 1 \quad \text{και} \quad \sum_{i=1}^{h_R+1} \frac{l_i^R}{2^{i-1}} \leq 1$$

Ας επιστρέψουμε στο αρχικό δένδρο. Παρατηρούμε ότι τα φύλλα που βρίσκονται στο επίπεδο i του αρχικού δένδρου είναι το άθροισμα των φύλλων που βρίσκονται στο επίπεδο i-1 των υποδένδρων L και R, δηλαδή:

$$l_i = l_{i-1}^L + l_{i-1}^R$$

Επομένως, ξεκινώντας με το αριστερό σκέλος της ζητούμενης πρότασης, έχουμε

$$\sum_{i=1}^{h+1}\frac{l_i}{2^{i-1}}=\sum_{i=2}^{h+1}\frac{l_i}{2^{i-1}}$$

$$=\sum_{i=1}^{h}\frac{l_i^L}{2^i}+\sum_{i=1}^{h}\frac{l_i^R}{2^i}$$

$$=\{\text{Για κάθε } i>h_L+1,\ l_i^L=0 \text{ και συμμετρικά για κάθε } i+1>h_R\}$$

$$\frac{1}{2}\cdot\sum_{i=1}^{h_L+1}\frac{l_i^L}{2^{i-1}}+\frac{1}{2}\cdot\sum_{i=1}^{h_R+1}\frac{l_i^R}{2^{i-1}}$$

$$<\ \{\text{Από την υπόθεση της επαγωγής}\}$$

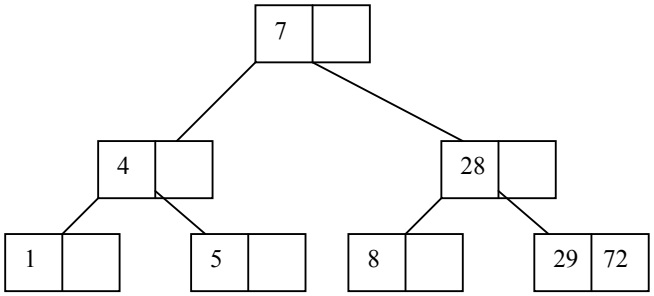
$$\frac{1}{2}\cdot 1+\frac{1}{2}\cdot 1$$

$$=1$$

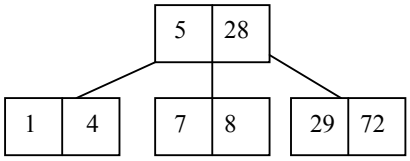
Αυτό συμπληρώνει την απόδειξη.

Άσκηση 2

Μία σειρά εισαγωγής των στοιχείων που έχει ως αποτέλεσμα την δημιουργία του δένδρου με τον μέγιστο αριθμό κόμβων είναι η 1, 4, 5, 7, 8, 28, 29, 72. (Υπάρχουν και άλλες σειρές εισαγωγής που δίνουν αυτό τον αριθμό κόμβων.)



Μια σειρά που παράγει δένδρο με τον μικρότερο αριθμό κόμβων είναι η 72, 4, 28, 29, 5, 8, 1, 7. Η σειρά αυτή δημιουργεί το πιο κάτω δένδρο με 4 κόμβους. (Υπάρχουν και άλλες σειρές εισαγωγής που δίνουν αυτό τον αριθμό κόμβων.)



Άσκηση 3

Προφανώς τα δένδρα τα οποία θα υλοποιήσουμε δεν έχουν κάποιο προκαθορισμένο βαθμό (δεν υπάρχει καθορισμένος μέγιστος βαθμός ενός κόμβου). Έτσι για εξοικονόμηση χώρου και για υλοποίηση λύσης χωρίς την επιβολή οποιωνδήποτε

περιορισμών/υποθέσεων επιλέγουμε να υλοποιήσουμε το δένδρο φυλάγοντας για κάθε κόμβο το πρώτο του παιδί και τον επί δεξιά του αδελφό. Αυτό υλοποιείται μέσω των δομών:

```
struct person{                  typedef struct FTnode{
     string name;                     struct person pers;
     int birth;                       struct FTnode *firstchild;
     int death;                       struct FTnode *rightsibling;
     string spouse;             } ftnode;
}
```

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

Αρχικά υλοποιούμε τη βοηθητική διαδικασία Find (name, ftree) η οποία εντοπίζει και επιστρέφει δείκτη στον κόμβο που αφορά το άτομο με όνομα name στο δένδρο που δείχνεται από το δείκτη ftree, αν υπάρχει.

```
ftnode *Find(string name, ftnode *p) {
  if (p == NULL)
     return NULL
  else {
     if ((p->pers->name) == name)
          return p;
     else {
          q = Find(name, p -> firstchild);

          if (q != NULL)
               return q;
          else
               return Find(name, p -> rightsibling);
  }
}
```

(i) Για να τυπώσουμε τα ονόματα των παιδιών ενός ατόμου με κάποιο όνομα, πρώτα εντοπίζουμε τον κόμβο που αντιστοιχεί στο συγκεκριμένο άτομο και μετά επισκεπτόμαστε τα παιδιά του μέσω του δείκτη firstchild του κόμβου και των δεικτών rightsibling των παιδιών:

```
Children(string name, ftnode *p) {
  q = Find(name, p);

  if (q != NULL) {
     q = p->firstchild;
     while (q != NULL) {
          print (q->pers)->name;
          q = q->rightsibling;
     }
  }
}
```

(ii) Για να εισαγάγουμε ένα νέο παιδί κάποιου ατόμου θα πρέπει πρώτα να εντοπίσουμε το άτομο αυτό, και στη συνέχεια να τοποθετήσουμε το νέο παιδί στη λίστα παιδιών που δείχνεται από το δείκτη firstchild:

```
Newchild(string child, int year, string parent, ftnode *p) {
  q = Find(parent, p);
  if (q != NULL) {
     r = (ftnode *) malloc(sizeof(ftnode));
     r->pers->name = child;
     r->pers->birth = year;
     r->pers->death = -1;
     r->firstchild = NULL;
     r->rightsibling = q->firstchild;
     q -> firstchild = r;
  }
}
```

(iii) Η διαδικασία που προτείνεται αρχικά εντοπίζει τον ζητούμενο κόμβο και στη συνέχειακαλεί μια δεύτερη, αναδρομική διαδικασία στο πρώτο παιδί του κόμβου αυτού. Η αναδρομική διαδικασία δουλεύει ως εξής: αν το παιδί δεν είναι κενό, τότε: αν το άτομο στη ρίζα του δένδρου βρίσκεται εν ζωή, τότε το όνομα του ατόμου τυπώνεται στην οθόνη και η διαδικασία καλείται (1) στο πρώτο παιδί του κόμβου και (2) στον επί δεξιά αδελφό του κόμβου.

```
Living(string name, ftnode *p) {
  q = Find(name, p);
  if (q == NULL)
     return;
  else
     if (q->firstchild != NULL)
          RecLiving(q->firstchild);
}

RecLive(ftnode *p){
  if (p->death == -1)
     print(p->pers->name);

  if (p->firstchild != NULL)
     RecLive(p->firstchild)

  if (p->rightsibling != NULL)
     RecLive(p->righsibling)
}
```

Ο χρόνος εκτέλεσης των διαδικασιών είναι O(n) όπου n είναι ο αριθμός των κόμβων του δένδρου.

Άσκηση 4

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:

```
Reverse(node *p){
  if (p != NULL) {
     temp = p->left;
     p->left = p->right;
     p->right = temp;
  }
  if (p->left != NULL)
     Reverse(p -> left);
  if (p->right != NULL)
     Reverse(p -> right);
}
```

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

Η μη-αναδρομική εκδοχή της διαδικασία απαιτεί τη χρήση βοηθητικής δομής.

I. Έστω p δείκτης στον κόμβο του δένδρου όπου βρισκόμαστε. Μεταθέτουμε τους δείκτες p->left και p->right. Τοποθετούμε σε μια ουρά (ή στοίβα – η σειρά με την οποία γίνονται οι αντιστροφές δεν έχει σημασία) τον δείκτη p->right, αν αυτός δεν είναι NULL.

II. Εφόσον ο κόμβος έχει αριστερό παιδί προχωρούμε αριστερά και επαναλαμβάνουμε από το βήμα I.

III. Διαφορετικά, εφόσον η ουρά δεν είναι κενή, ανασύρουμε τον κόμβο κορυφής της, έστω q και προχωρούμε στο βήμα I με τον δείκτη q.

```
Reverse2(node *p){
  queue Q;
  MakeEmpty(Q);

  while (p != NULL OR !IsEmpty(Q)) {
     if (p != NULL) {
          temp = p->left;
          p->left = p->right;
          p->right = temp;

          if (p -> right != NULL)
               Enqueue(p -> right, Q);
          p = p ->left;
     }
```

```
      else
            p = Dequeue(Q);
      }
}
```

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

Στην πράξη αναμένουμε η δεύτερη διαδικασία να είναι αποδοτικότερη εφόσον αποφεύγει την αναδρομή και τη σχετική χρήση στοίβας για αποθήκευση και ανάσυρση των στοιχείων των καλούντων διαδικασιών, γεγονός που στοιχίζει στην αποδοτικότητα χρόνου και χώρου.

Άσκηση 5

Για την υλοποίηση του ζητούμενου ΑΤΔ χρησιμοποιούμε μια επέκταση των AVL δένδρων όπου, σε κάθε κόμβο, περιέχεται πεδίο numnodes το οποίο αποθηκεύει τον αριθμό των κόμβων του δένδρου που ριζώνει στον κόμβο. Χρησιμοποιούμε την εξής εγγραφή:

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

Ως βοηθητικές διαδικασίες υλοποιούμε τις countAllBiggerThan(p,k) και countAllsmallerThan(p,k) οι οποίες υπολογίζουν και επιστρέφουν τον αριθμό των κλειδιών στο δένδρο που ριζώνει στον κόμβο p και είναι μεγαλύτερα και μικρότερα, αντίστοιχα από το k:

```
CountAllinRange(node *p, int  k1, int k2) {
      if p = NULL
            return 0
      if (p->key < k1)
            return CountAllinRange(p->right, k1, k2);
      if (p->key > k2)
            return CountAllinRange(p->left, k1, k2)
      else
            return ( 1 + countAllBiggerThan(p->left, k1)
                     + countAllSmallerThan(p->right, k2) )
}
```

Η ιδέα που χρησιμοποιείται στον αναδρομικό αυτό αλγόριθμο είναι ότι αν `(p->key < k1)` τότε μπορούμε να αγνοήσουμε τα στοιχεία του αριστερού υπόδενδρου του p και παρόμοια αν `(p->key > k2)` τότε μπορούμε να αγνοήσουμε τα στοιχεία του δεξιού υπόδενδρου του p. Διαφορετικά, θα πρέπει να επιστρέψουμε [το πλήθος των κλειδιών του αριστερού υπόδενδρου που είναι μεγαλύτερα από k1 + το πλήθος των κλειδιών του δεξιού υπόδενδρου που είναι μικρότερα από το k2 + 1 για ρίζα].

Η διαδικασία αυτή θα κάνει το πολύ όσες αναδρομικές κλήσεις όσες και το ύψος του δένδρου ενώ κάθε κλήση θα έχει χρόνο εκτέλεσης O(1). Επομένως ο χρόνος εκτέλεσης της διαδικασίας θα είναι O(lg *n*).

Πιο κάτω ορίζονται οι δύο βοηθητικές διαδικασίες countAllBiggerThan(p,k) και countAllsmallerThan(p,k).

```
countAllBiggerThan(p, k){
if p == NULL
     return 0
if (p->key < k)
     return countAllBiggerThan(p->right, k)
else
     return (1 + countAllBiggerThan(p->left, k)
               + p->right->numnodes);
}

countAllSmallerThan(p, k){
if p == NULL
     return 0
if (p->key > k)
     return countAllSmallerThan(p->left, k)
else
     return (1 + countAllSmallerThan(p->right, k)
               + p->left->numnodes);
}
```

Απομένει να δείξουμε τον τρόπο με τον οποίο μπορούμε να υλοποιήσουμε τις πράξεις λεξικού στη δομή μας έτσι ώστε το πεδίο numnodes των κόμβων να παίρνει τη σωστή τιμή. Αυτή η επέκταση αφήνεται ως άσκηση στον αναγνώστη. (Υπόδειξη: οι περιστροφές γίνονται ακριβώς όπως σε ένα AVL δένδρο, αλλά τώρα θα πρέπει σε κάθε βήμα μετά από την εισαγωγή του νέου κόμβου να δίνουμε στα πεδία numnodes κατάλληλες καινούριες τιμές.)