



## Θ.Ε. ΠΛΣ50 (2007-8) – ΓΡΑΠΤΗ ΕΡΓΑΣΙΑ Ε5

### Στόχος

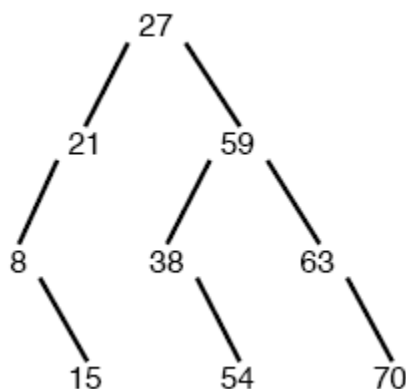
Στην εργασία αυτή θα ασχοληθούμε με τα Δυαδικά Δένδρα Αναζήτησης και με διάφορους αλγόριθμους ταξινόμησης.

### Θέμα 1: Δυαδικά Δένδρα Αναζήτησης

#### Θέμα 1α. Εισαγωγή τιμών σε Δυαδικό Δένδρο Αναζήτησης

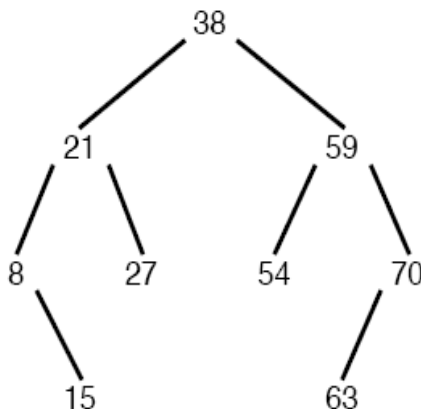
Κατασκευάστε το Δυαδικό Δένδρο Αναζήτησης το οποίο προκύπτει από την εισαγωγή των τιμών 27, 59, 21, 38, 54, 63, 8, 70, 15. Οι τιμές εισάγονται με την παραπάνω σειρά, μία κάθε φορά.

Το Δυαδικό Δένδρο Αναζήτησης που προκύπτει είναι το εξής:



#### Θέμα 1β. Διαγραφή τιμών σε Δυαδικό Δένδρο Αναζήτησης

Έστω το παρακάτω Δυαδικό Δένδρο Αναζήτησης.

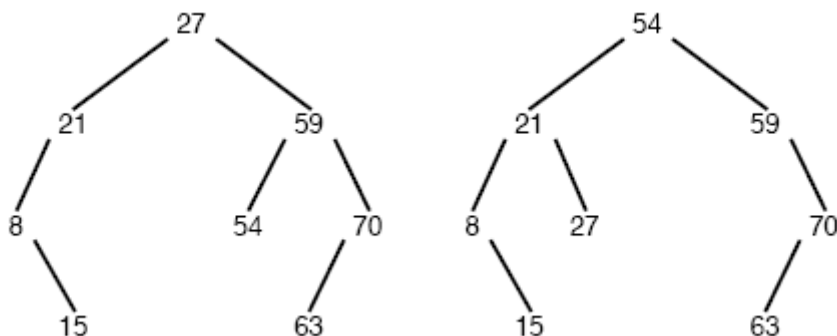


Σχεδιάστε το Δυαδικό Δένδρο Αναζήτησης που προκύπτει αν διαγραφεί η ρίζα από το παραπάνω Δυαδικό Δένδρο Αναζήτησης. Υπάρχει μόνο ένας τρόπος για να γίνει η διαγραφή της ρίζας;



Η διαγραφή ενός κόμβου που έχει και τα δύο παιδιά του (όπως είναι στην περίπτωση μας ο κόμβος ρίζα του Δυαδικού Δένδρου Αναζήτησης του Θέματος 1α) πραγματοποιείται με την αντικατάσταση του κόμβου που πρόκειται να διαγραφεί είτε από τον κόμβο με τη μέγιστη τιμή από το αριστερό υποδένδρο είτε από τον κόμβο με την ελάχιστη τιμή από το δεξιό υποδένδρο του κόμβου που πρόκειται να διαγραφεί.

Έτσι, μπορούμε να έχουμε δύο πιθανά αποτελέσματα:



## Θέμα 2: Υλοποίηση συναρτήσεων σε Δυαδικά Δένδρα Αναζήτησης

Θεωρήστε τον ακόλουθο ορισμό της δομής ενός Δυαδικού Δένδρου Αναζήτησης:

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

### Θέμα 2α. Εκτύπωση των τιμών των κόμβων σε post order σειρά

Γράψτε μια αναδρομική συνάρτηση σε γλώσσα προγραμματισμού C με πρότυπο

```
void printKeys_postorder(node current)
```

η οποία να τυπώνει τις τιμές των κόμβων ενός Δυαδικού Δένδρου Αναζήτησης, μία ανα γραμμή, σε post order διαπέραση.

Στον κώδικα, πριν από τον ορισμό της συνάρτησης πρέπει να περιλαμβάνεται η εντολή `#include <stdio.h>` για τον ορισμό της σταθερής NULL.

Ο κώδικας της συνάρτησης είναι ο εξής:

```
void printKeys_postorder(node current)
{
    if (current == NULL)
        return ;

    printKeys_postorder(current -> left) ;
    printKeys_postorder(current -> right) ;
    printf("%d\n", current -> key);
}
```



## Θέμα 2β. Εκτύπωση των τιμών των κόμβων που είναι μεγαλύτερες από μια δεδομένη τιμή

Γράψτε μια αναδρομική συνάρτηση σε γλώσσα προγραμματισμού C με πρότυπο

```
void greater_keys(node current, int value);
```

η οποία να τυπώνει όλες τις τιμές των κόμβων ενός Δυαδικού Δένδρου Αναζήτησης που είναι μεγαλύτερες από την τιμή της μεταβλητής value.

Στον κώδικα, πριν από τον ορισμό της συνάρτησης πρέπει να περιλαμβάνεται η εντολή `#include <stdio.h>` για τον ορισμό της σταθερής NULL.

Ο κώδικας της συνάρτησης είναι ο εξής:

```
void greater_keys(node current, int value) {  
    if (current == NULL)  
        return;  
    greater_keys(current->right, value);  
    if (current->key > value) {  
        printf("%d\n", current->key);  
        greater_keys(current->left, value);  
    }  
}
```

## Θέμα 3: Αλγόριθμοι Ταξινόμησης

### Θέμα 3α.

Εξηγείστε ποιο είναι το αποτέλεσμα της εκτέλεσης της παρακάτω συνάρτησης:

```
int f (const int a [ ], const unsigned int length)  
{  
    unsigned int i;  
    int result = 1;  
    for (i= 0; i < length-1; ++i )  
        if (a[i+1] < a[i] )  
            result = 0;  
    if (result == 1)  
        return 1;  
    else  
        return 0;  
}
```

Η παραπάνω υλοποίηση δεν είναι η καλύτερη δυνατή. Εξηγείστε γιατί και υλοποιείστε μια απλούστερη και πιο αποδοτική έκδοσή της.

Η συνάρτηση επιστρέφει 1 εάν τα στοιχεία του πίνακα a[ ] είναι σε αύξουσα σειρά ενώ επιστρέφει 0 εάν τα στοιχεία του πίνακα a[ ] δεν είναι σε αύξουσα σειρά.

Δεν αποτελεί την πιο αποδοτική υλοποίηση για δύο λόγους:

- ♦ Διατρέχει πάντοτε ολόκληρο τον πίνακα πριν να ολοκληρώσει την εκτέλεσή της και να επιστρέψει το αποτέλεσμά της, ενώ σε πολλές περιπτώσεις είναι δυνατόν να ολοκληρώσει την εκτέλεσή της νωρίτερα, μόλις δηλαδή ανιχνεύσει ότι τα στοιχεία του πίνακα a[ ] δεν είναι σε αύξουσα σειρά.
- ♦ Το τελευταίο μέρος της είναι εντελώς περιττό. Χρειάζεται απλώς να επιστρέψει η συνάρτηση την τιμή 1 ή την τιμή 0.



Μια πιο απλή και αποδοτική έκδοση της παραπάνω συνάρτησης είναι η εξής:

```
int isAscending (const int a [ ], const unsigned int length)
{
    unsigned int i;
    for (i= 0; i < length-1; ++i )
        // Εάν το επόμενο στοιχείο είναι μικρότερο από το τρέχον στοιχείο, τότε ο πίνακας δεν
        // είναι σε αύξουσα σειρά και μπορούμε να επιστρέψουμε 0 άμεσα, χωρίς να χρειάζεται
        // να συνεχιστεί η εκτέλεση του βρόγχου for
        if (a[i+1] < a[i])
            return 0;
    return 1;
}
```

### Θέμα 3β.

Έστω ότι πρέπει να βρούμε τα 1000 πιο ακριβά προϊόντα από μία μη ταξινομημένη λίστα τιμών των προϊόντων που περιλαμβάνει  $10^7$  διαφορετικά προϊόντα. Προτείνονται οι δύο παρακάτω αλγόριθμοι:

1<sup>ος</sup> Αλγόριθμος: Επανάλαβε 1000 φορές σειριακή αναζήτηση (με γραμμική πολυπλοκότητα  $O(n)$ ).

2<sup>ος</sup> Αλγόριθμος: Μετατρέψτε τη λίστα σε ένα πίνακα (πολυπλοκότητα  $O(n)$ ) και στη συνέχεια ταξινομήστε τον πίνακα (πολυπλοκότητα  $O(n \log n)$ ) και επιλέξτε τα πρώτα 1000 προϊόντα.

Ποιον από τους δύο προτεινόμενους αλγόριθμους θα προτιμήσετε να χρησιμοποιήσετε θεωρώντας ότι η προσπέλαση 100 συνεχόμενων προϊόντων στη μη ταξινομημένη λίστα απαιτεί 0.1 msec και ότι η ταξινόμηση 100 προϊόντων απαιτεί επίσης 0.1 msec; Ο χρόνος για την ανάκτηση των τιμών των προϊόντων θεωρείται αμελητέος.

Ο παράγοντας κλιμάκωσης ( $c$ =χρόνος εκτέλεσης/πολυπλοκότητα) για τη γραμμική αναζήτηση

$$\text{είναι: } \frac{0.1}{100} = 0.001.$$

Ο παράγοντας κλιμάκωσης ( $c$ =χρόνος εκτέλεσης/πολυπλοκότητα) για την ταξινόμηση είναι:

$$\frac{0.1}{100 \log 100} = \frac{0.1}{200} = 0.0005.$$

Ο συνολικός χρόνος εκτέλεσης για τον 1<sup>ο</sup> Αλγόριθμο είναι:  $T_A = 0.001 * 10^7 * 10^3 = 10^7 \text{ msec}$  ή  $10^4 \text{ sec}$ .

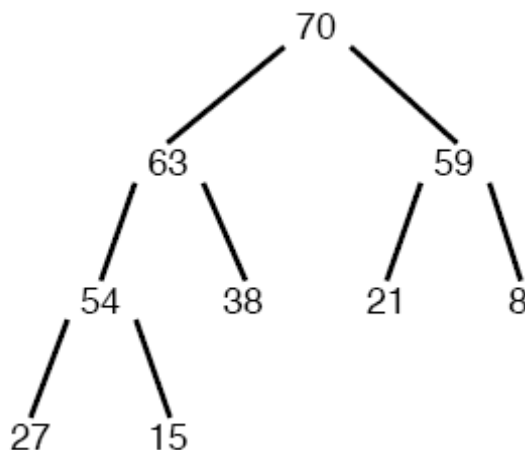
Ο συνολικός χρόνος εκτέλεσης για το 2<sup>ο</sup> Αλγόριθμο αποτελείται από το χρόνο της μετατροπής της λίστας σε πίνακα, που είναι:  $T_{B1} = 0.001 * 10^7 = 10^4 \text{ msec}$  ή  $10 \text{ sec}$  και το χρόνο που απαιτείται για την ταξινόμηση του πίνακα:  $T_{B2} = 0.0005 * 10^7 * \log(10^7) = 35 * 10^3 \text{ msec}$  ή  $35 \text{ sec}$ .

Προφανώς, θα πρέπει να επιλέξουμε το 2<sup>ο</sup> Αλγόριθμο.

### Θέμα 4: Δυαδικός Σωρός

Κατασκευάστε τον Maximum Δυαδικό Σωρό που προκύπτει από την εισαγωγή των τιμών 27, 59, 21, 38, 54, 63, 8, 70, 15. Οι τιμές εισάγονται με την παραπάνω σειρά, μία κάθε φορά, ενώ η ιδιότητα του σωρού διατηρείται μετά από την εισαγωγή κάθε νέας τιμής.

Ο Maximum Δυαδικός Σωρός που προκύπτει είναι ο εξής:



## Θέμα 5: Ο Αλγόριθμος Ταξινόμησης Heapsort

### Θέμα 5α.

Έστω ένας ταξινομημένος πίνακας. Αποτελεί αυτός ο πίνακας ένα min-σωρό (min-heap);

Ναι. Αφού  $A[1] \leq A[2] \leq \dots \leq A[n]$  και αφού  $\text{parent}(i) < i$  (όταν  $i > 1$ ), ισχύει ότι  $A[\text{parent}(i)] \leq A[i]$  για κάθε  $i > 1$ . Επομένως, ο πίνακας  $A$  έχει την min-heap ιδιότητα.

### Θέμα 5β.

Μετατρέψτε τον ψευδοκώδικα της συνάρτησης MAX-HEAPIFY του βιβλίου ΑΠΟ έτσι ώστε να μην περιλαμβάνει την αναδρομική κλήση της σειράς 10.

Η μη αναδρομική μορφή της MAX-HEAPIFY είναι η εξής:

```
MaxHeapify_non_recursive(A, i)
l ← Left(i); r ← Right(i);
n ← heapSize[A]
while true do
    if (l ≤ n and A[l] > A[i])
        then largest ← l
    else largest ← i
    if (r ≤ n and A[r] > A[largest])
        then largest ← r
    if largest = i
        then return
    else A[i] ↔ A[largest]
        i ← largest
```

### Θέμα 5γ.

Δείξτε ότι η πολυπλοκότητα χρόνου εκτέλεσης του Heapsort για τη χειρότερη περίπτωση είναι  $\Omega(n \lg n)$ .

Έστω ότι η αρχική κατάσταση του πίνακα  $A$  είναι η εξής:

$A[1..n] = \langle n, n-1, \dots, 3, 2, 1 \rangle$ . Τότε η συνάρτηση BUILD-MAX-HEAP δε θα αλλάξει καθόλου τον  $A$ . Κάθε φορά όμως που θα εκτελείται η εντολή εναλλαγής  $A[1] \leftrightarrow A[i]$ , το  $A[1]$  θα αποκτά μία τιμή που η συνάρτηση MAX-HEAPIFY θα πρέπει να μετακινεί σε ένα φύλλο του σωρού. Έτσι, κάθε MAX-HEAPIFY θα απαιτεί  $\Theta(\lg k)$  βήματα, όπου  $k$  είναι ο τρέχων αριθμός



στοιχείων στο σωρό. Επομένως, για τις πρώτες  $\left\lceil \frac{n}{2} \right\rceil$  επαναλήψεις, η MAX-HEAPIFY απαιτεί  $c \cdot \lg n$  βήματα, για κάποιο  $c \geq 1$ . Άρα, η εκτέλεση του αλγόριθμου Heapsort απαιτεί περισσότερα από  $\left\lceil \frac{n}{2} \right\rceil \cdot c \cdot \lg n$  βήματα και επομένως η πολυπλοκότητα χρόνου εκτέλεσης του Heapsort για τη χειρότερη περίπτωση είναι  $\Omega(n \lg n)$ .