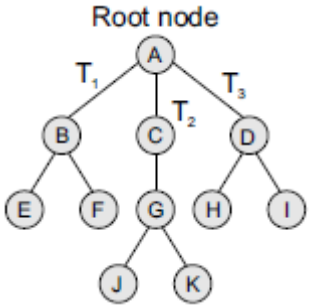
ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΑΚΑΔΗΜΑΪΚΟ ΕΤΟΣ 2017-2018
ΧΕΙΜΕΡΙΝΟ ΕΞΑΜΗΝΟ
ΜΑΘΗΜΑ: ΔΟΜΕΣ ΔΕΔΟΜΕΝΩΝ

ΕΡΓΑΣΤΗΡΙΟ 7 – ΣΗΜΕΙΩΣΕΙΣ

Δένδρα (Trees)

Ένα **δένδρο (tree)** ορίζεται αναδρομικά ως ένα σύνολο ενός ή περισσοτέρων κόμβων ανάμεσα στους οποίους ένας από αυτούς ορίζεται ως η **ρίζα (root)** του δένδρου και όλοι οι υπόλοιποι μπορούν να διαχωριστούν σε μη κενά σύνολα καθένα από τα οποία είναι τα υπο-δένδρα του αρχικού δένδρου. Η ακόλουθη εικόνα αποτελεί ένα παράδειγμα δένδρου.



Οι διάφορες κατηγορίες δένδρων συνοψίζονται στις εξής:

1. General trees
2. Forests
3. Binary trees
4. Binary search trees
5. Expression trees
6. Tournament trees

Δυαδικά Δένδρα (Binary Trees)

Ένα δυαδικό δένδρο είναι μια δομή δεδομένων που ορίζεται ως η συλλογή κόμβων όπου ο αρχικός (ο πιο ψηλά κόμβος) είναι η ρίζα του δένδρου ενώ ο οποιοσδήποτε κόμβος έχει 0, 1 ή 2 το πολύ παιδιά. Οι κόμβοι που έχουν 0 παιδιά ονομάζονται φύλλα (leafs) ή τερματικοί κόμβοι. Ο κάθε κόμβος περιλαμβάνει πεδίο με τα δεδομένα καθώς και δύο δείκτες: ο ένας δείχνει προς το αριστερό παιδί ενώ ο άλλος δείχνει προς το δεξιό παιδί. Η ακόλουθη εικόνα αποτελεί ένα παράδειγμα ενός δυαδικού δένδρου.

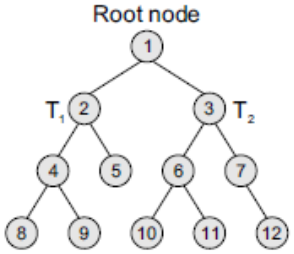
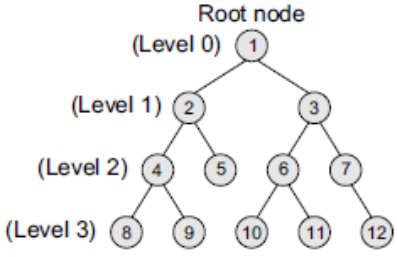
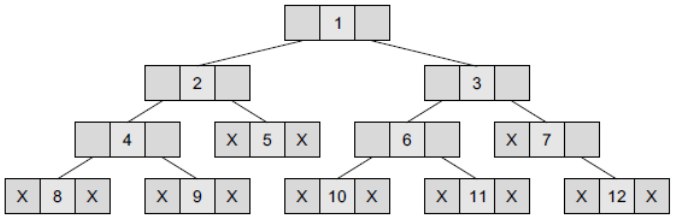


Figure 9.3 Binary tree



Ένα δυαδικό δένδρο μπορεί να αναπαρασταθεί στη μνήμη είτε με τη βοήθεια λιστών ή με τη χρήση ακολουθιακής αναπαράσταση. Για την αναπαράσταση με τη βοήθεια λίστας υιοθετούμε την ακόλουθη δομή:

```
struct node {
        struct node *left;
        int data;
        struct node *right;
};
```

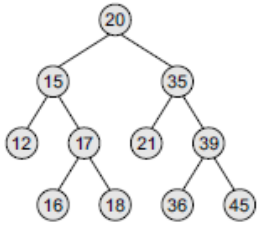


ROOT = 3

	LEFT	DATA	RIGHT
1	–1	8	–1
2	–1	10	–1
3	5	1	8
4			
5	9	2	14
6			
7			
8	20	3	11
9	1	4	12
10			
11	–1	7	18
12	–1	9	–1
13			
14	–1	5	–1
15			
16	–1	11	–1
17			
18	–1	12	–1
19			
20	2	6	16

AVAIL = 15

Στη σειριακή αναπαράσταση υιοθετούμε ένα πίνακα. Η ρίζα του δένδρου αποθηκεύεται στην πρώτη θέση ενώ για οποιονδήποτε άλλο κόμβο που πρόκειται να αποθηκευθεί στην Κ θέση, αποθηκεύονται τα παιδιά του στην 2Κ και 2Κ+1 θέσεις αντίστοιχα.



1	20
2	15
3	35
4	12
5	17
6	21
7	39
8	
9	
10	16
11	18
12	
13	
14	36
15	45

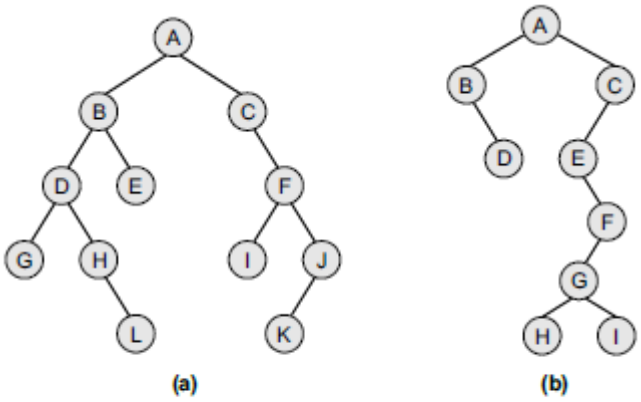
Για τη διάσχιση ενός δυαδικού δένδρου δεν χρησιμοποιούμε ένα γραμμικό τρόπο αλλά ένα από τους ακόλουθους:

- **Pre-order Traversal.** Εκτελούμε τις ακόλουθες ενέργειες:

- Επισκεπτόμαστε τη ρίζα
- Διασχίζουμε το αριστερό υπο-δένδρο
- Διασχίζουμε το δεξιό υπο-δένδρο

Η pre-order διάσχιση ονομάζεται αλλιώς **διάσχιση κατά βάθος (depth-first traversal)**. Ακολουθεί ένας αναδρομικός αλγόριθμος για την απεικόνιση της μεθόδου:

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:           Write TREE -> DATA
Step 3:           PREORDER(TREE -> LEFT)
Step 4:           PREORDER(TREE -> RIGHT)
       [END OF LOOP]
Step 5: END
```

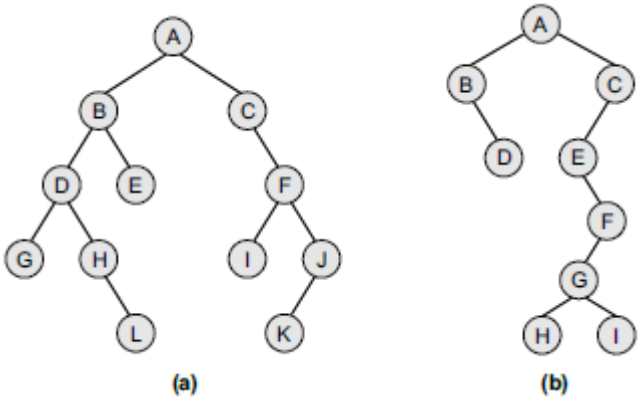



```
TRAVERSAL ORDER: A, B, D, G, H, L, E, C, F, I, J,
                 and K
TRAVERSAL ORDER: A, B, D, C, D, E, F, G, H, and I
```

- **In-order Traversal.** Εκτελούμε τις ακόλουθες ενέργειες:
  - Διασχίζουμε το αριστερό υπο-δένδρο
  - Επισκεπτόμαστε τη ρίζα
  - Διασχίζουμε το δεξιό υπο-δένδρο

Η in-order διάσχιση ονομάζεται αλλιώς **συμμετρική διάσχιση (symmetric traversal)**. Ακολουθεί ένας αναδρομικός αλγόριθμος για την απεικόνιση της μεθόδου:

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:           INORDER(TREE -> LEFT)
Step 3:           Write TREE -> DATA
Step 4:           INORDER(TREE -> RIGHT)
       [END OF LOOP]
Step 5: END
```

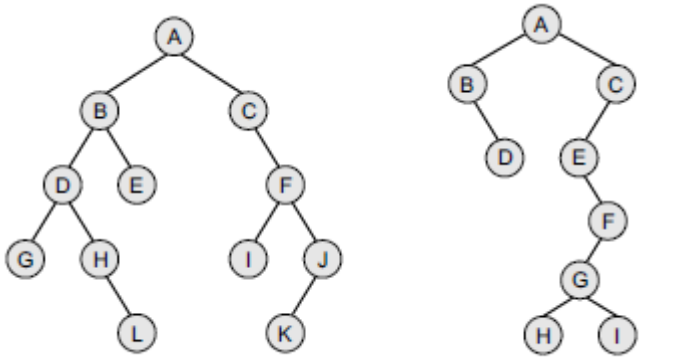



```
TRAVERSAL ORDER: G, D, H, L, B, E, A, C, I, F, K, and J
TRAVERSAL ORDER: B, D, A, E, H, G, I, F, and C
```

- **Post-order Traversal.** Εκτελούμε τις ακόλουθες ενέργειες:
  - Διασχίζουμε το αριστερό υπο-δένδρο
  - Διασχίζουμε το δεξιό υπο-δένδρο
  - Επισκεπτόμαστε τη ρίζα

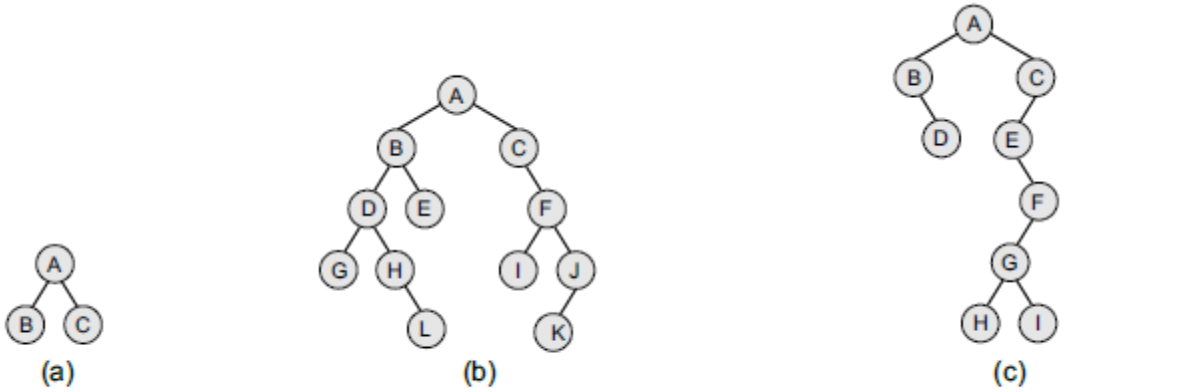
  Ακολουθεί ένας αναδρομικός αλγόριθμος για την απεικόνιση της μεθόδου:

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:          POSTORDER(TREE -> LEFT)
Step 3:          POSTORDER(TREE -> RIGHT)
Step 4:          Write TREE ->DATA
       [END OF LOOP]
Step 5: END
```




```
TRAVERSAL ORDER: G, L, H, D, E, B, I, K, J, F, C, and A
TRAVERSAL ORDER: D, B, H, I, G, F, E, C, and A
```

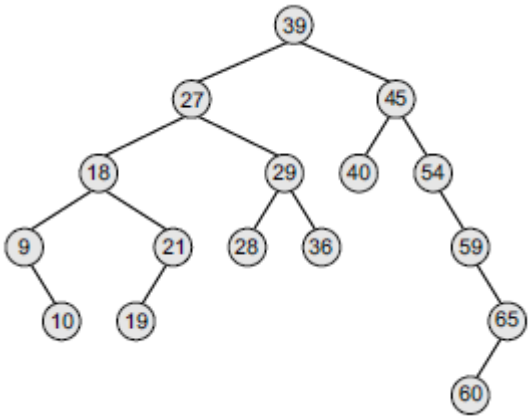
- **Level-order Traversal.** Κάθε φορά προσπελαύνουμε όλους τους κόμβους σε ένα επίπεδο πριν προχωρήσουμε στο επόμενο. Η level-order διάσχιση ονομάζεται αλλιώς **διάσχιση κατά πλάτος (breadth-first traversal)**.




```
TRAVERSAL ORDER:
A, B, and C

TRAVERSAL ORDER:
A, B, C, D, E, F, G, H, I, J, L, and K

TRAVERSAL ORDER:
A, B, C, D, E, F, G, H, and I
```

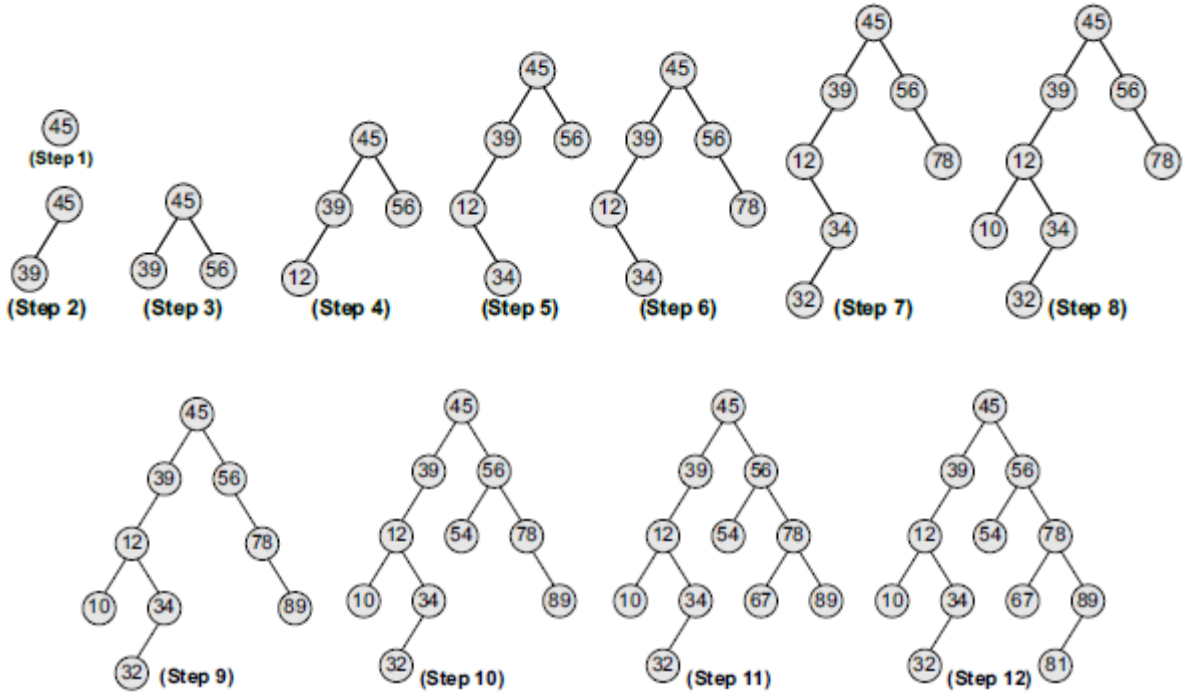
Δυαδικά Δένδρα Αναζήτησης (Binary Search Trees)

Σε ένα δυαδικό δένδρο αναζήτησης όλοι οι κόμβοι στο **αριστερό** υπο-δένδρο έχουν τιμή **μικρότερη** από τη ρίζα του υπο-δένδρου ενώ όλοι οι κόμβοι στο **δεξιό** υπο-δένδρο έχουν τιμή **μεγαλύτερη** από τη ρίζα.



Μια και όλοι οι κόμβοι είναι κατά κάποιο τρόπο ‘ταξινομημένοι’, χρόνος που απαιτείται για την αναζήτηση μειώνεται σημαντικά. Ο κανόνας που ισχύει στα δυαδικά δένδρα ουσιαστικά μας κατευθύνει ως προς το υπο-δένδρο που πρέπει να κατευθυνθούμε κατά την αναζήτηση στοιχείων. Η ακόλουθη εικόνα απεικονίζει ένα παράδειγμα δημιουργίας ενός δυαδικού δένδρου αναζήτησης.

45, 39, 56, 12, 34, 78, 32, 10, 89, 54, 67, 81



Αναζήτηση στοιχείου. Η αναζήτηση περιλαμβάνει τη σύγκριση του αναζητούμενου στοιχείου με την τιμή της ρίζας του εκάστοτε υπο-δένδρου και την επιλογή του κατάλληλου υπο-δένδρου για τη συνέχιση της αναζήτησης.

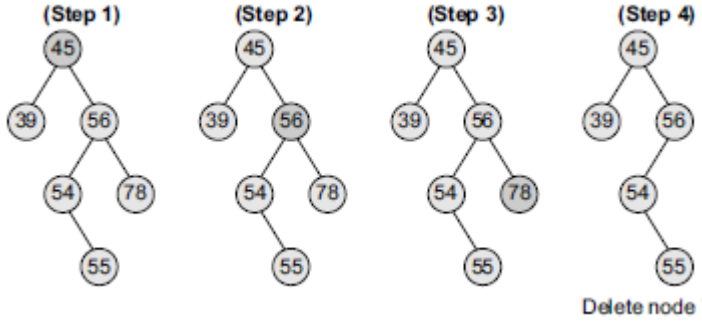
```
searchElement (TREE, VAL)

Step 1: IF TREE -> DATA = VAL OR TREE = NULL
          Return TREE
        ELSE
         IF VAL < TREE -> DATA
          Return searchElement(TREE -> LEFT, VAL)
         ELSE
          Return searchElement(TREE -> RIGHT, VAL)
         [END OF IF]
        [END OF IF]
Step 2: END
```

Εισαγωγή στοιχείου. Η εισαγωγή θα πρέπει να τοποθετήσει το νέο κόμβο στην κατάλληλη θέση μέσα στο δυαδικό δένδρο αναζήτησης ώστε να ικανοποιεί το βασικό κριτήριο (όλοι οι κόμβοι του αριστερού υπο-δένδρου έχουν μικρότερες τιμές από την τιμή της ρίζάς ενώ το δεξιό υπο-δένδρο έχουν τιμές μεγαλύτερες). Ο αλγόριθμος είναι παραπλήσιος με τον αλγόριθμο της αναζήτησης.

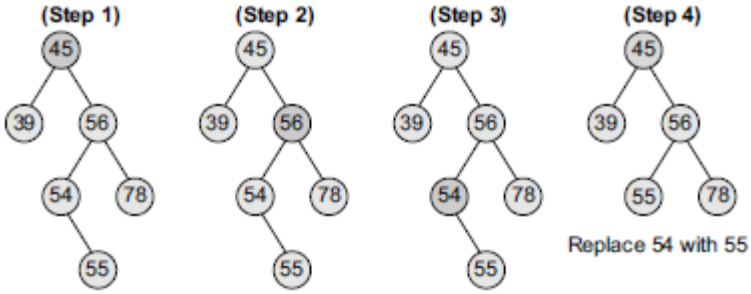
```
Insert (TREE, VAL)

Step 1: IF TREE = NULL
            Allocate memory for TREE
            SET TREE -> DATA = VAL
            SET TREE -> LEFT = TREE -> RIGHT = NULL
        ELSE
            IF VAL < TREE -> DATA
                  Insert(TREE -> LEFT, VAL)
            ELSE
                  Insert(TREE -> RIGHT, VAL)
            [END OF IF]
        [END OF IF]
Step 2: END
```

Διαγραφή Στοιχείου. Η διαγραφή είναι πιο απαιτητική ενέργεια αφού υπάρχουν περιπτώσεις κατά τις οποίες το δένδρο θα πρέπει να αναδιοργανωθεί. Ο λόγος είναι ότι κατά τη διαγραφή, το κριτήριο του δυαδικού δένδρου αναζήτησης μπορεί να παραβιαστεί, συνεπώς, θα πρέπει να γίνουν ενέργειες ώστε να ικανοποιηθεί τελικά. Ισχύουν οι ακόλουθες περιπτώσεις:

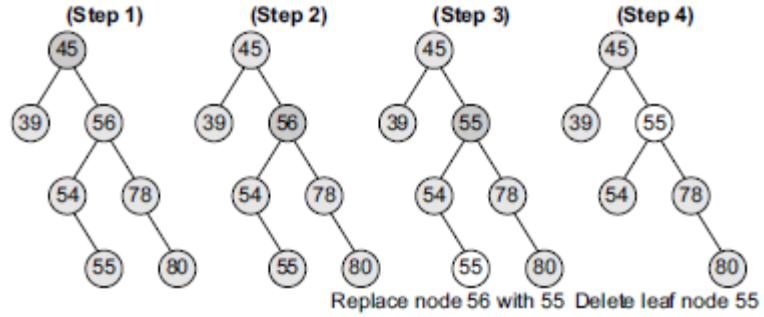
- **Διαγραφή κόμβου που δεν έχει παιδιά.** Στην περίπτωση αυτή απλά διαγράφουμε τον κόμβο.

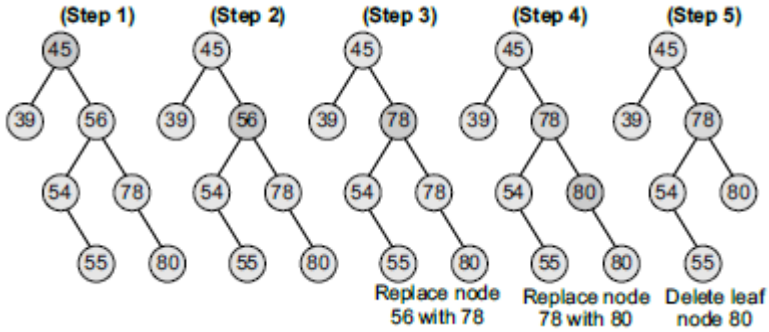


- **Διαγραφή κόμβου που έχει ένα παιδί.** Σε αυτή την περίπτωση αντικαθιστούμε τον κόμβο με το παιδί του.



- **Διαγραφή κόμβου με δύο παιδιά**. Σε αυτή την περίπτωση, αντικαθιστούμε τον κόμβο με τον προηγούμενο του στην in-order διάσχιση (μεγαλύτερος κόμβος του αριστερού υπο-δένδρου) ή με τον επόμενο κόμβο στην in-order διάσχιση (μικρότερη τιμή στο δεξιό υπο-δένδρο). Ο προηγούμενος ή ο επόμενος κόμβος μπορεί να διαγραφεί με μια από τις παρουσιαζόμενες περιπτώσεις. Ακολουθούν δύο παραδείγματα διαγραφής.





Ο αλγόριθμος διαγραφής κόμβων έχει ως ακολούθως:

```
Delete (TREE, VAL)

Step 1: IF TREE = NULL
          Write "VAL not found in the tree"
        ELSE IF VAL < TREE->DATA
          Delete(TREE->LEFT, VAL)
        ELSE IF VAL > TREE->DATA
          Delete(TREE->RIGHT, VAL)
        ELSE IF TREE->LEFT AND TREE->RIGHT
          SET TEMP = findLargestNode(TREE->LEFT)
          SET TREE->DATA = TEMP->DATA
          Delete(TREE->LEFT, TEMP->DATA)
        ELSE
          SET TEMP = TREE
          IF TREE->LEFT = NULL AND TREE->RIGHT = NULL
              SET TREE = NULL
          ELSE IF TREE->LEFT != NULL
               SET TREE = TREE->LEFT
          ELSE
               SET TREE = TREE->RIGHT
          [END OF IF]
          FREE TEMP
        [END OF IF]
Step 2: END
```

Εύρεση ύψους ενός δένδρου. Για να βρούμε το ύψος ενός δένδρου βρίσκουμε το ύψος του αριστερού και του δεξιού υπο-δένδρου. Στο ύψος του υψηλότερου υπο-δένδρου προσθέτουμε 1.

```
Height (TREE)

Step 1: IF TREE = NULL
             Return 0
        ELSE
         SET LeftHeight = Height(TREE->LEFT)
         SET RightHeight = Height(TREE->RIGHT)
         IF LeftHeight > RightHeight
              Return LeftHeight + 1
         ELSE
              Return RightHeight + 1
         [END OF IF]
        [END OF IF]
Step 2: END
```

Εύρεση πλήθους των κόμβων. Μετράμε το πλήθος των κόμβων στο αριστερό και στο δεξιό παιδί.

```
totalNodes(TREE)

Step 1: IF TREE = NULL
            Return 0
        ELSE
            Return totalNodes(TREE -> LEFT)
                + totalNodes(TREE -> RIGHT) + 1
        [END OF IF]
Step 2: END
```

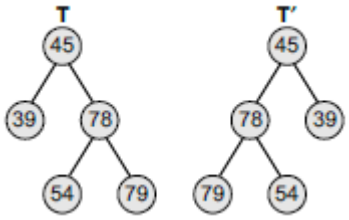
Εύρεση πλήθους των εσωτερικών κόμβων. Ακολουθούμε την ίδια λογική με τα παραπάνω μετρώντας το πλήθος των εσωτερικών κόμβων στο αριστερό και στο δεξιό υπο-δένδρο.

```
totalInternalNodes(TREE)

Step 1: IF TREE = NULL
            Return 0
        [END OF IF]
       IF TREE -> LEFT = NULL AND TREE -> RIGHT = NULL
            Return 0
        ELSE
            Return totalInternalNodes(TREE -> LEFT) +
            totalInternalNodes(TREE -> RIGHT) + 1
       [END OF IF]
Step 2: END
```

Εύρεση πλήθους των φύλλων ενός δένδρου. Προσθέτουμε το πλήθος των φύλλων του αριστερού και του δεξιού υπο-δένδρου.

```
totalExternalNodes(TREE)

Step 1: IF TREE = NULL
            Return 0
       ELSE IF TREE -> LEFT = NULL AND TREE -> RIGHT = NULL
            Return 1
        ELSE
            Return totalExternalNodes(TREE -> LEFT) +
            totalExternalNodes(TREE -> RIGHT)
        [END OF IF]
Step 2: END
```

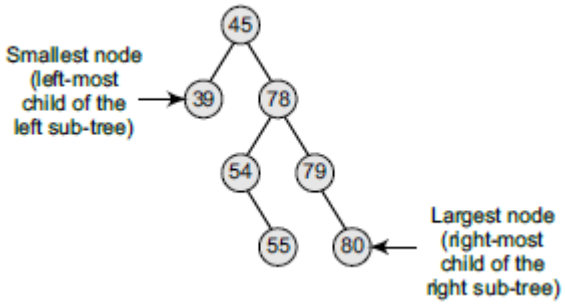
Εύρεση εικόνας ενός δένδρου. Η εικόνα ενός δένδρου παράγεται με την εναλλαγή του σριστερού με το δεξιό υπο-δένδρο.



```
MirrorImage(TREE)

Step 1: IF TREE != NULL
            MirrorImage(TREE -> LEFT)
            MirrorImage(TREE -> RIGHT)
            SET TEMP = TREE -> LEFT
            SET TREE -> LEFT = TREE -> RIGHT
            SET TREE -> RIGHT = TEMP
        [END OF IF]
Step 2: END
```

Εύρεση του μικρότερου στοιχείου σε ένα δένδρο. Το μικρότερο στοιχείο θα βρίσκεται στο αριστερό υπο-δένδρο. Συνεπώς, βρίσκουμε τον πιο κάτω αριστερά κόμβο του δένδρου. Αν το αριστερό υπο-δένδρο είναι NULL τότε το μικρότερο στοιχείο είναι η ρίζα.

Εύρεση του μεγαλύτερου στοιχείου σε ένα δένδρο. Το μεγαλύτερο στοιχείο θα βρίσκεται στο δεξιό υπο-δένδρο. Συνεπώς, βρίσκουμε τον πιο κάτω δεξιά κόμβο του δένδρου. Αν το δεξιό υπο-δένδρο είναι NULL τότε το μικρότερο στοιχείο είναι η ρίζα.




```
findSmallestElement(TREE)

Step 1: IF TREE = NULL OR TREE -> LEFT = NULL
            Returen TREE
        ELSE
            Return findSmallestElement(TREE -> LEFT)
        [END OF IF]
Step 2: END
```

```
findLargestElement(TREE)

Step 1: IF TREE = NULL OR TREE -> RIGHT = NULL
            Return TREE
        ELSE
            Return findLargestElement(TREE -> RIGHT)
        [END OF IF]
Step 2: END
```

Code 1

```
#include <stdio.h>
#include <conio.h>
#include <malloc.h>
struct node
{
        int data;
        struct node *left;
        struct node *right;
```

```
};

struct node *tree;

void create_tree(struct node *);
struct node *insertElement(struct node *, int);
void preorderTraversal(struct node *);
void inorderTraversal(struct node *);
void postorderTraversal(struct node *);
struct node *findSmallestElement(struct node *);
struct node *findLargestElement(struct node *);
struct node *deleteElement(struct node *, int);
struct node *mirrorImage(struct node *);
int totalNodes(struct node *);
int totalExternalNodes(struct node *);
int totalInternalNodes(struct node *);
int Height(struct node *);
struct node *deleteTree(struct node *);

int main()
{
	int option, val;
	struct node *ptr;
	create_tree(tree);
	do
	{
		printf("\n ******MAIN MENU******* \n");
		printf("\n 1. Insert Element");
		printf("\n 2. Preorder Traversal");
		printf("\n 3. Inorder Traversal");
		printf("\n 4. Postorder Traversal");
		printf("\n 5. Find the smallest element");
		printf("\n 6. Find the largest element");
		printf("\n 7. Delete an element");
		printf("\n 8. Count the total number of nodes");
		printf("\n 9. Count the total number of external nodes");
		printf("\n 10. Count the total number of internal nodes");
		printf("\n 11. Determine the height of the tree");
		printf("\n 12. Find the mirror image of the tree");
		printf("\n 13. Delete the tree");
		printf("\n 14. Exit");
		printf("\n\n Enter your option : ");
		scanf("%d", &option);
		switch(option)
		{
			case 1:
				printf("\n Enter the value of the new node : ");
				scanf("%d", &val);
```

```
                tree = insertElement(tree, val);
                break;
        case 2:
                printf("\n The elements of the tree are : \n");
                preorderTraversal(tree);
                break;
        case 3:
                printf("\n The elements of the tree are : \n");
                inorderTraversal(tree);
                break;
        case 4:
                printf("\n The elements of the tree are : \n");
                postorderTraversal(tree);
                break;
        case 5:
                ptr = findSmallestElement(tree);
                printf("\n Smallest element is :%d",ptr–>data);
                break;
        case 6:
                ptr = findLargestElement(tree);
                printf("\n Largest element is : %d", ptr–>data);
                break;
        case 7:
                printf("\n Enter the element to be deleted : ");
                scanf("%d", &val);
                tree = deleteElement(tree, val);
                break;
        case 8:
                printf("\n Total no. of nodes = %d", totalNodes(tree));
                break;
        case 9:
                printf("\n Total no. of external nodes = %d",
                totalExternalNodes(tree));
                break;
        case 10:
                printf("\n Total no. of internal nodes = %d",
                totalInternalNodes(tree));
                break;
        case 11:
                printf("\n The height of the tree = %d",Height(tree));
                break;
        case 12:
                tree = mirrorImage(tree);
                break;
        case 13:
                tree = deleteTree(tree);
                break;
}
```

```
        }while(option!=14);
        getch();
        return 0;
}

void create_tree(struct node *tree)
{
        tree = NULL;
}

struct node *insertElement(struct node *tree, int val)
{
        struct node *ptr, *nodeptr, *parentptr;
        ptr = (struct node*)malloc(sizeof(struct node));
        ptr–>data = val;
        ptr–>left = NULL;
        ptr–>right = NULL;
        if(tree==NULL)
        {
                tree=ptr;
                tree–>left=NULL;
                tree–>right=NULL;
        }
        else
        {
                parentptr=NULL;
                nodeptr=tree;
                while(nodeptr!=NULL)
                {
                        parentptr=nodeptr;
                        if(val<nodeptr–>data)
                                nodeptr=nodeptr–>left;
                        else
                                nodeptr = nodeptr–>right;
                }
                if(val<parentptr–>data)
                        parentptr–>left = ptr;
                else
                        parentptr–>right = ptr;
        }
        return tree;
}

void preorderTraversal(struct node *tree)
{
        if(tree != NULL)
        {
                printf("%d\t", tree–>data);
```

```
                preorderTraversal(tree–>left);
                preorderTraversal(tree–>right);
        }
}

void inorderTraversal(struct node *tree)
{
        if(tree != NULL)
        {
                inorderTraversal(tree->left);
                printf("%d\t", tree->data);
                inorderTraversal(tree->right);
        }
}

void postorderTraversal(struct node *tree)
{
        if(tree != NULL)
        {
                postorderTraversal(tree->left);
                postorderTraversal(tree->right);
                printf("%d\t", tree->data);
        }
}

struct node *findSmallestElement(struct node *tree)
        {
        if( (tree == NULL) || (tree->left == NULL))
                return tree;
        else
                return findSmallestElement(tree ->left);
}

struct node *findLargestElement(struct node *tree)
{
        if( (tree == NULL) || (tree->right == NULL))
                return tree;
        else
                return findLargestElement(tree->right);
}

struct node *deleteElement(struct node *tree, int val)
{
        struct node *cur, *parent, *suc, *psuc, *ptr;
        if(tree–>left==NULL)
        {
                printf("\n The tree is empty ");
                return(tree);
```

```
}
parent = tree;
cur = tree–>left;
while(cur!=NULL && val!= cur–>data)
{
        parent = cur;
        cur = (val<cur–>data)? cur–>left:cur–>right;
}
if(cur == NULL)
{
        printf("\n The value to be deleted is not present in the tree");
        return(tree);
}
if(cur–>left == NULL)
        ptr = cur–>right;
else if(cur–>right == NULL)
        ptr = cur–>left;
else
{
        // Find the in–order successor and its parent
        psuc = cur;
        cur = cur–>left;
        while(suc–>left!=NULL)
        {
                psuc = suc;
                suc = suc–>left;
        }
        if(cur==psuc)
        {
                // Situation 1
                suc–>left = cur–>right;
        }
        else
        {
                // Situation 2
                suc–>left = cur–>left;
                psuc–>left = suc–>right;
                suc–>right = cur–>right;
        }
        ptr = suc;
}
// Attach ptr to the parent node
if(parent–>left == cur)
        parent–>left=ptr;
else
        parent–>right=ptr;
free(cur);
return tree;
```

```
}

int totalNodes(struct node *tree)
{
        if(tree==NULL)
                return 0;
        else
                return(totalNodes(tree–>left) + totalNodes(tree–>right) + 1);
}

int totalExternalNodes(struct node *tree)
{
        if(tree==NULL)
                return 0;
        else if((tree–>left==NULL) && (tree–>right==NULL))
                return 1;
        else
                return (totalExternalNodes(tree–>left) +
        totalExternalNodes(tree–>right));
}

int totalInternalNodes(struct node *tree)
{
        if( (tree==NULL) || ((tree–>left==NULL) && (tree–>right==NULL)))
                return 0;
        else
                return (totalInternalNodes(tree–>left) + totalInternalNodes(tree–>right) + 1);
}

int Height(struct node *tree)
{
        int leftheight, rightheight;
        if(tree==NULL)
                return 0;
        else
        {
                leftheight = Height(tree–>left);
                rightheight = Height(tree–>right);
                if(leftheight > rightheight)
                        return (leftheight + 1);
                else
                        return (rightheight + 1);
        }
}

struct node *mirrorImage(struct node *tree)
{
        struct node *ptr;
```

```
        if(tree!=NULL)
        {
                mirrorImage(tree–>left);
                mirrorImage(tree–>right);
                ptr=tree–>left;
                ptr–>left = ptr–>right;
                tree–>right = ptr;
        }
}

struct node *deleteTree(struct node *tree)
{
        if(tree!=NULL)
        {
                deleteTree(tree–>left);
                deleteTree(tree–>right);
                free(tree);
        }
}
```