Προγραμματισμός

Αναδρομή

Κλήσεις Συναρτήσεων

- Όταν καλείται μια συνάρτηση, πρέπει
  - Να θυμάται σε ποιο σημείο του προγράμματος θα επιστρέψει
  - Να δεσμεύσει χώρο για την τιμή που θα επιστρέψει
  - Να δεσμεύσει χώρο για τα ορίσματα της
  - Να δεσμεύσει χώρο για τις τοπικές μεταβλητές της
- Η μνήμη για στατικές και καθολικές μεταβλητές δεσμεύεται από την αρχή της εκτέλεσης του προγράμματος

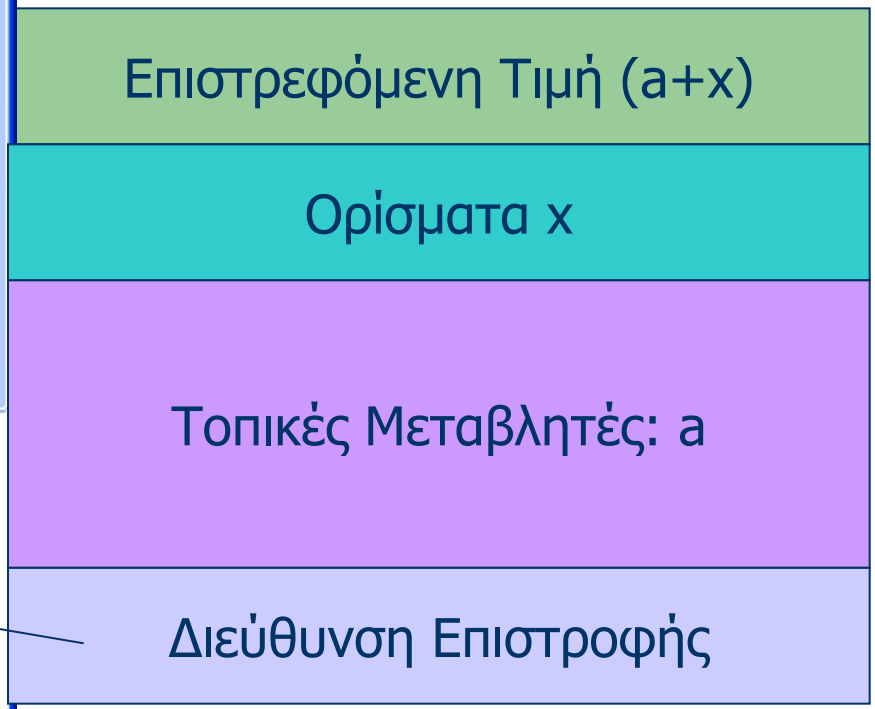
Παράδειγμα

frames.c (~\unixstuff\tests\lecture8) - GVIM

File Edit Tools Syntax Buffers Window Help

```
 1 #include <stdio.h>
 2
 3 int f(int x)
 4 {
 5         int a;
 6
 7         a = 5;
 8         return a+x;
 9 }
10
11 int main()
12 {
13
14         f(10);
15         5+10;
16
17 }
18
~
~
~
~
"frames.c" [unix] 18L, 101C written
```

Stack Frame for function **f**

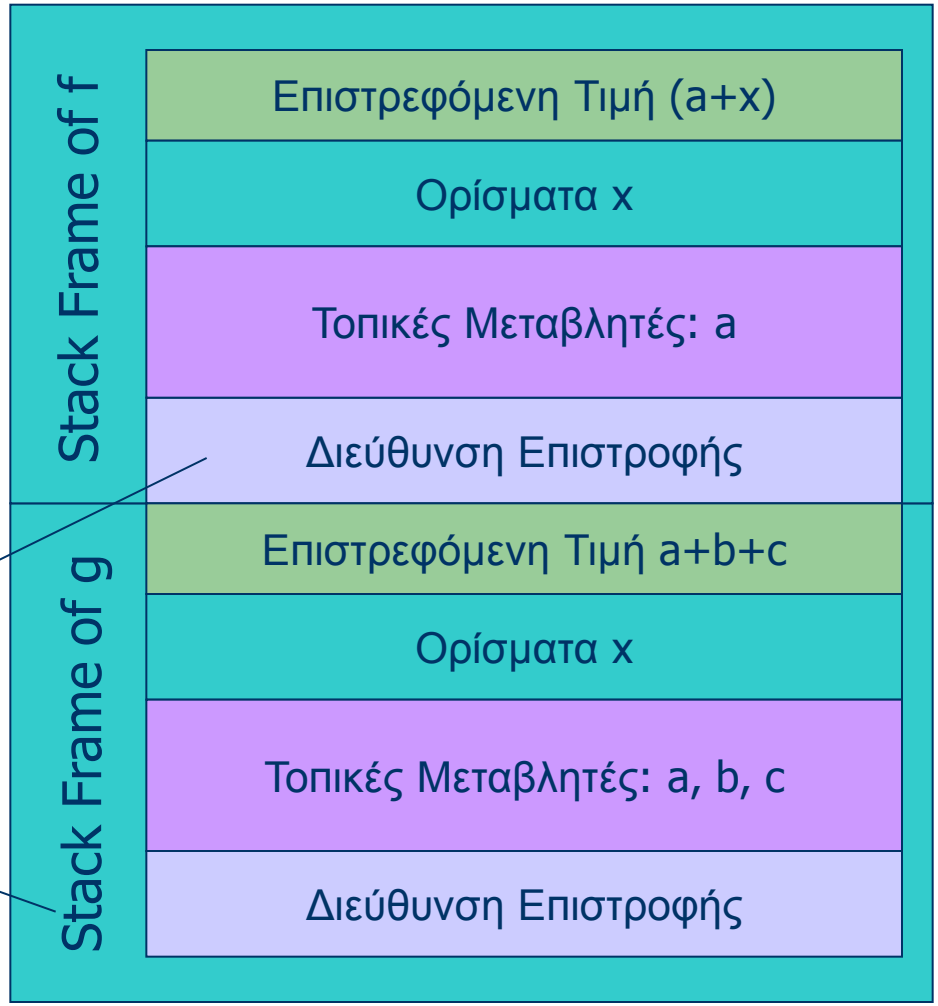


Εμφωλιασμένες Κλήσεις Συναρτήσεων

frames2.c (~\unixstuff\tests\lecture8) - GVIM

File Edit Tools Syntax Buffers Window Help

```
 1 int g(int x)
 2 {
 3         int a, b, c;
 4
 5         a=b=c=7;
 6
 7         return a+b+c;
 8 }
 9
10
11
12 int f(int x)
13 {
14         int a;
15
16         a = 5;
17
18         g(a);
19
20         return a+x;
21 }
22
23 int main()
24 {
25
26         f(10);
27         5+10;
28
29 }
"frames2.c" [unix] 30L, 151C written
```



Η Στοίβα (stack)

```
f()
{
    g() + h();
}
g()
{
    m(q());
}
main()
{
    f();
}
```

Η Στοίβα (stack)

```
f()
{
    g() + h();
}
g()
{
    m(q());
}
main()
{
    f();
}
```

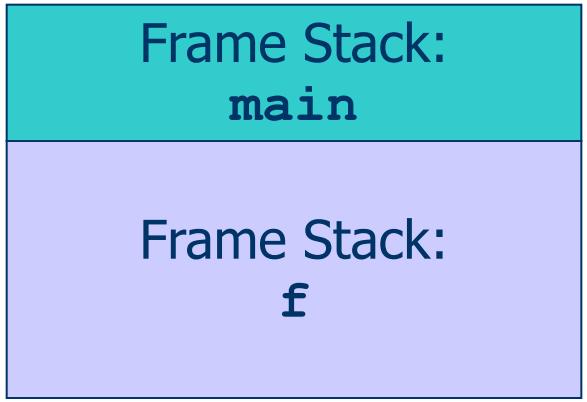
Frame Stack:
`main`

 **Execution**

Η Στοίβα (stack)

```
f()
{
    g() + h();
}
g()
{
    m(q());
}
main()
{
    f();
}
```

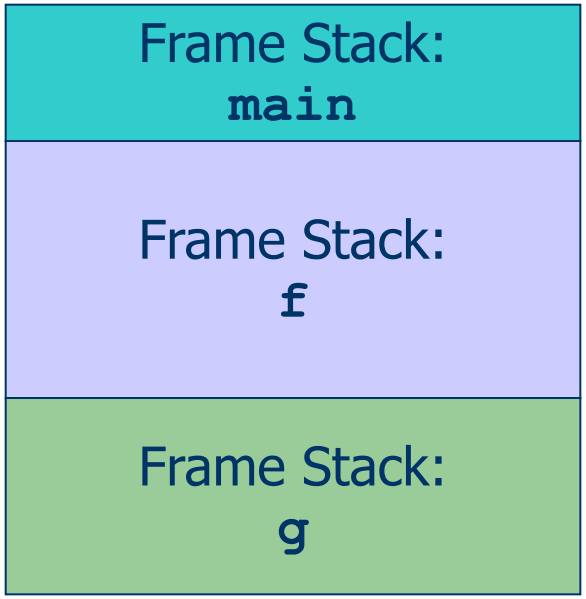
Execution



Η Στοίβα (stack)

```
f()
{
    g() + h();
}
g()
{
    m(q());
}
main()
{
    f();
}
```

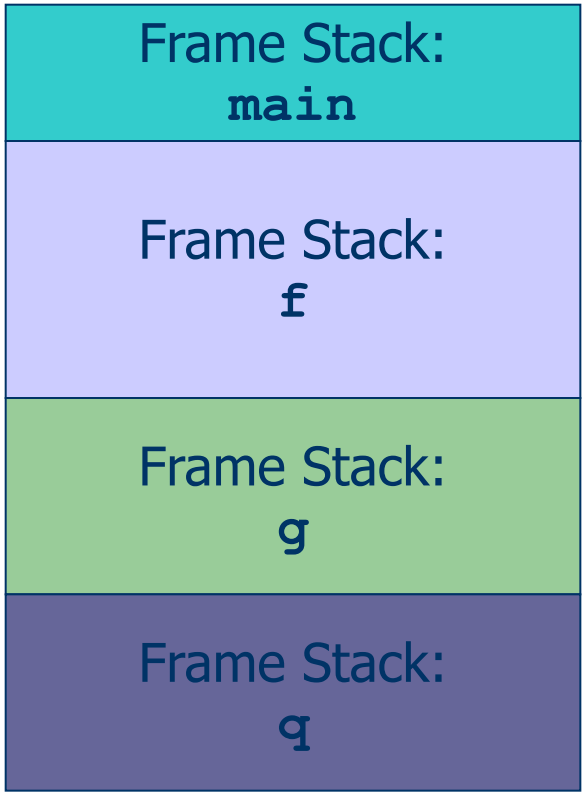
Execution



Η Στοίβα (stack)

```
f()
{
    g() + h();
}
g()
{
    m(q());
}
main()
{
    f();
}
```

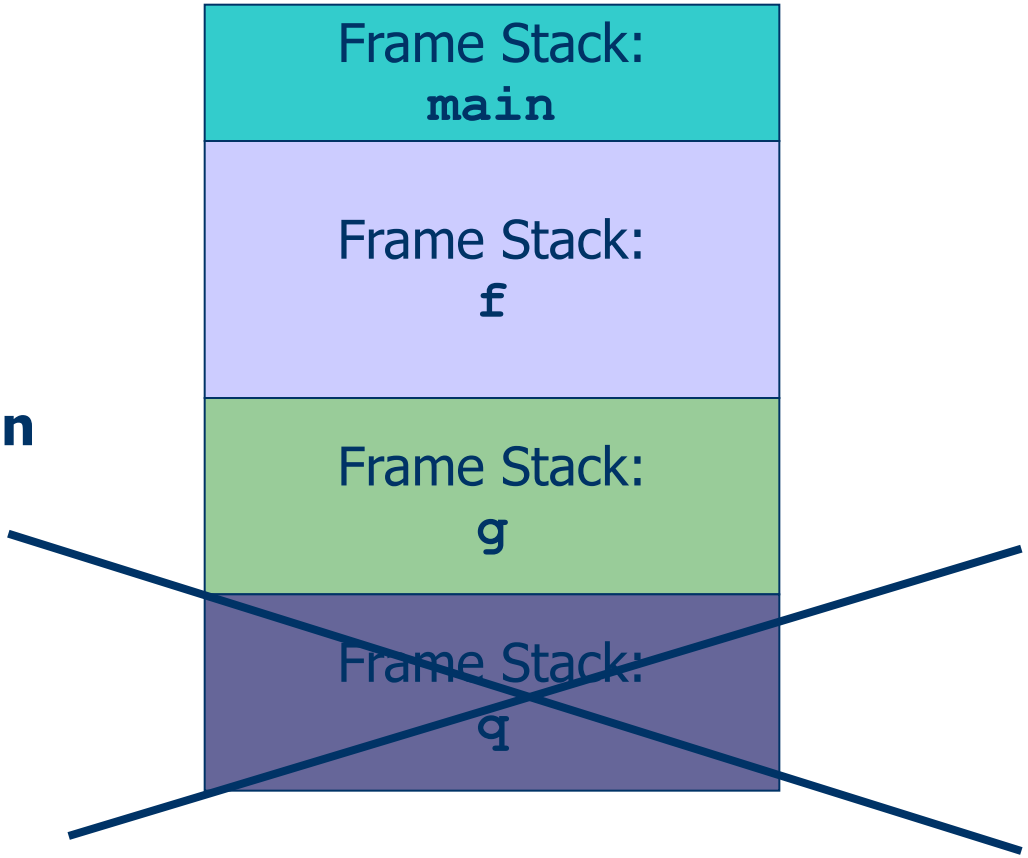
Execution



Η Στοίβα (stack)

```
f()
{
    g() + h();
}
g()
{
    m(q());
}
main()
{
    f();
}
```

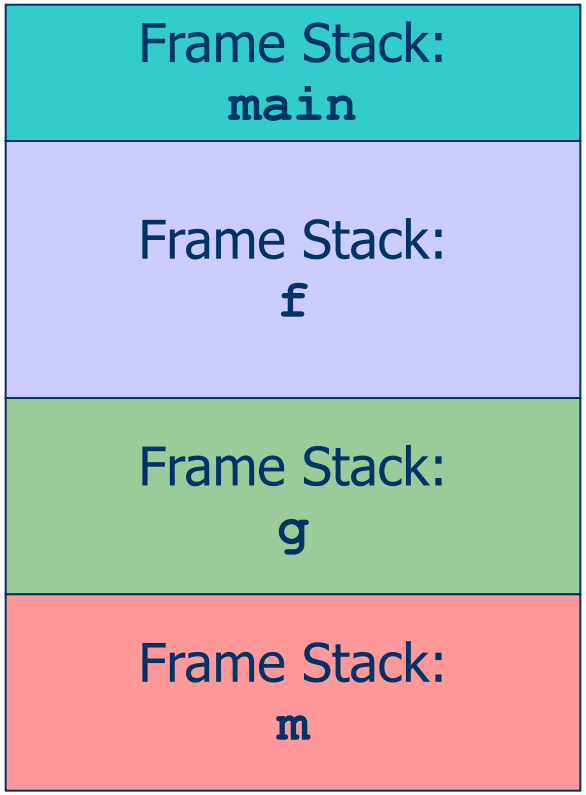
Execution



Η Στοίβα (stack)

```
f()
{
    g() + h();
}
g()
{
    m(q());
}
main()
{
    f();
}
```

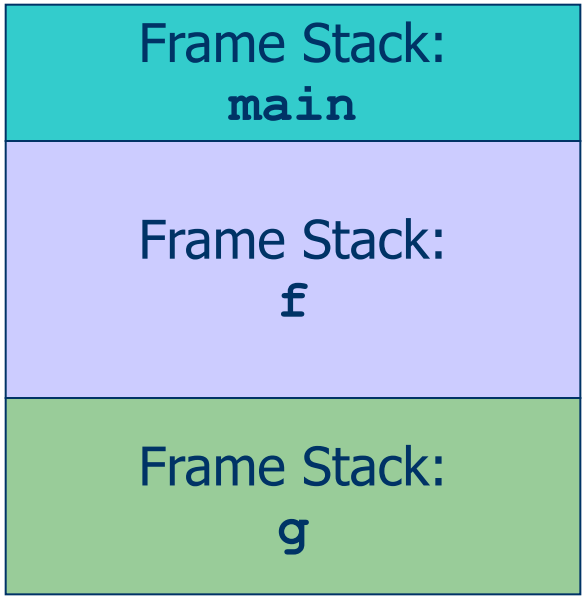
Execution



Η Στοίβα (stack)

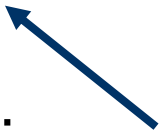
```
f()
{
    g() + h();
}
g()
{
    m(g());
}
main()
{
    f();
}
```

Execution

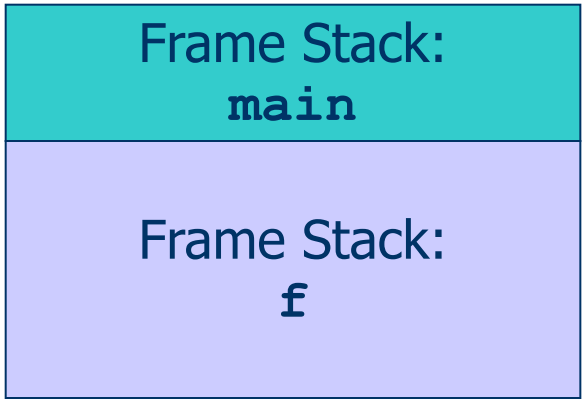


Η Στοίβα (stack)

```
f()
{
    g() + h();
}
g()
{
    m(q());
}
main()
{
    f();
}
```



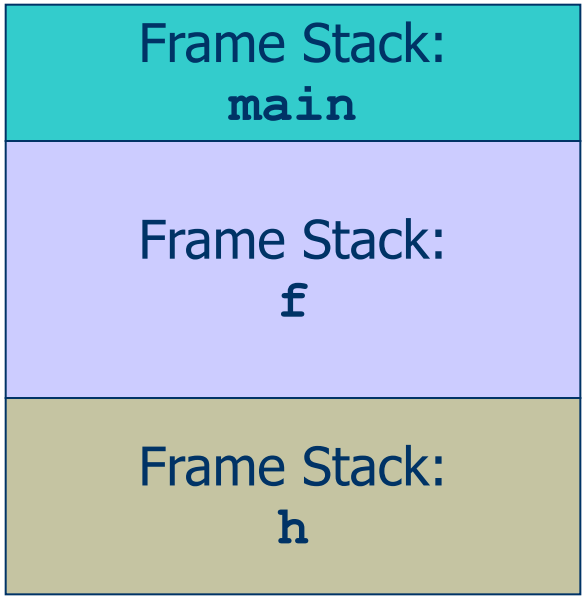
Execution



Η Στοίβα (stack)

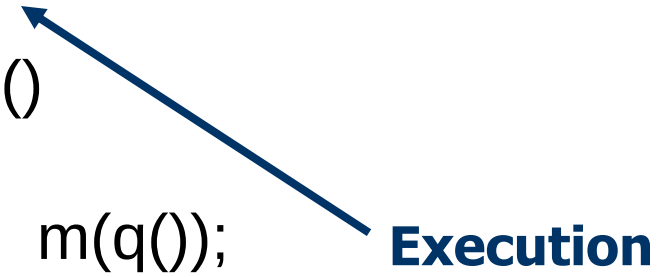
```
f()
{
    g() + h();
}
g()
{
    m(q());
}
main()
{
    f();
}
```

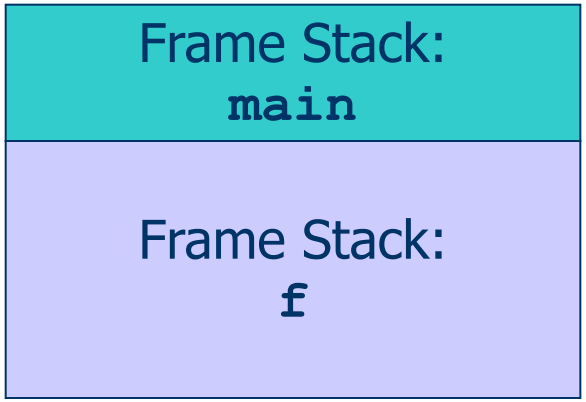
Execution



Η Στοίβα (stack)

```
f()
{
    g() + h();
}
g()
{
    m(q());
}
main()
{
    f();
}
```





Η Στοίβα (stack)

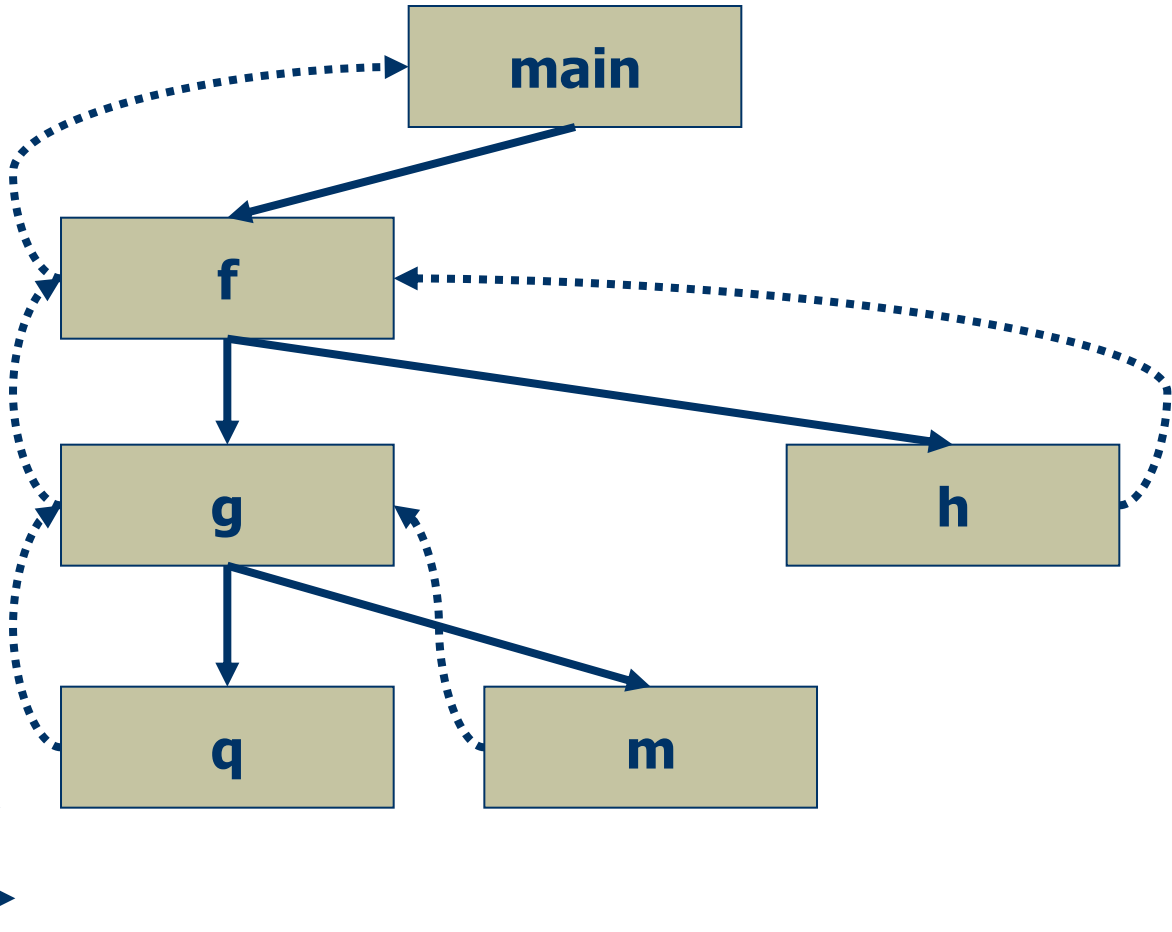
```
f()
{
    g() + h();
}
g()
{
    m(q());
}
main()
{
    f();
}
```

Frame Stack:
`main`

Execution

Το Δέντρο Κλήσεων των Συναρτήσεων

```
f()
{
    g() + h();
}
g()
{
    m(q());
}
main()
{
    f();
}
```



Γενική Ιδέα Αναδρομής

- Γιατί σπάμε το πρόγραμμα σε συναρτήσεις;
  - Κάθε συνάρτηση λύνει ένα «μικρότερο» (υπό)πρόβλημα
  - Συνδυάζουμε τα μικρότερα προβλήματα με τέτοιο τρόπο που να λύσουμε το συνολικό πρόβλημα
  - Παράδειγμα: Φτιάξτε ένα πρόγραμμα που να διαχειρίζεται μια βάση δεδομένων με φοιτητές
    - Συναρτήσεις
      - Για διαχείριση της εισόδου του χρήστη, τι θέλει να κάνει
      - Για εισαγωγή/διαγραφή/αλλαγή/αναζήτηση στοιχείων
      - Για εκτύπωση της βάσης δεδομένων
      - κτλ

Παράδειγμα: παραγοντικό (factorial)

- f(n) = n!

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot f(n-1) & else \end{cases}$$

- n! = 1· 2· 3· ⋯ · (n-1)· n

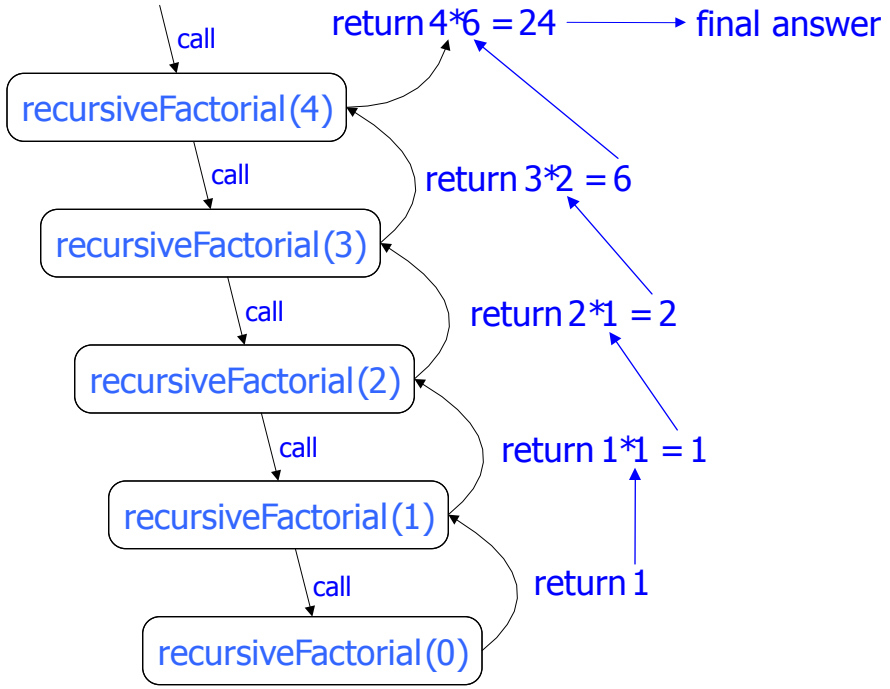
- *Ο κώδικας στις γλώσσες συναρτησιακού προγραμματισμού είναι πολύ παρόμοιος*

```
int  Factorial(int n)
{
      if  (n  ==  0)
            return  1;                          // base case
      else
            return  n  *  Factorial(n- 1);      // recursive case
}
```

Αναδρομή

- Παρακολούθηση αναδρομής

"trace"



Παράδειγμα

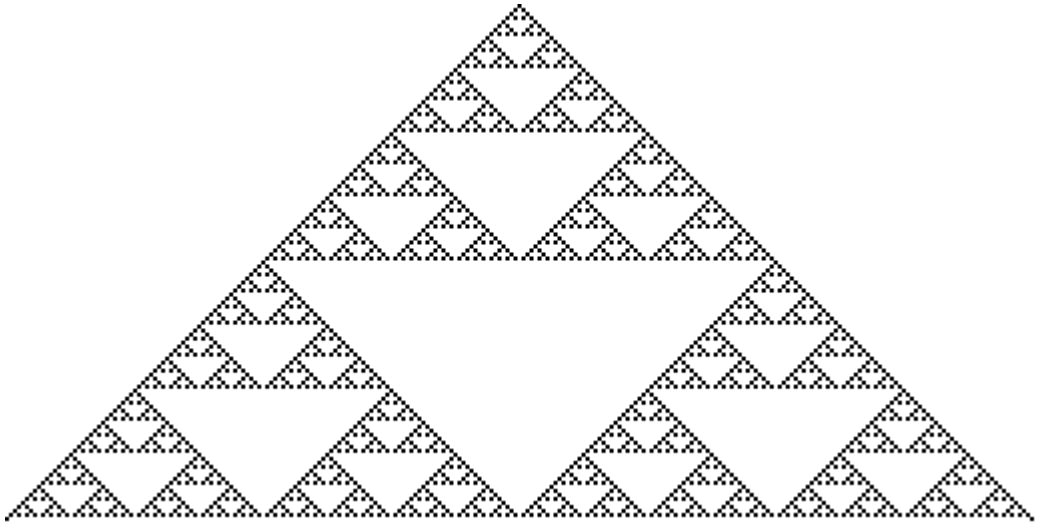
```
int factorial( int n)
{
    if ( n <= 1)
        return 1;
    else
        return n*factorial(n-1);
}
```

(Άμεση) Αναδρομή

- Τα τρία βασικά συστατικά της αναδρομής
- Για να λύσουμε ένα πρόβλημα μεγέθους n
  1. Η βασική περίπτωση (περιπτώσεις) (base-case): Όταν το n είναι αρκετά μικρό δίνουμε μια άμεση (χωρίς άλλη αναδρομή) λύση του «μικρότερου» προβλήματος
  2. Αναδρομική κλήση: σπάσε το πρόβλημα σε «μικρότερα» **ίδια** προβλήματα και λύσε τα με αναδρομική κλήση (κλήση στην **ίδια**) συνάρτηση
  3. Συγχώνευση: συνδύασε τις λύσεις των μικρότερων προβλημάτων, για να λύσεις το πρόβλημα μεγέθους n
- Το 3ο βήμα δεν είναι απαραίτητο
- Το «μέγεθος» του προβλήματος μπορεί να είναι το μέγεθος ενός ορίσματος, το μέγεθος ενός πίνακα, το μέγεθος μιας δομής (π.χ., γράφος), κτλ

Παράδειγμα: Sierpinski Sieve

- “The Sierpinski sieve is a fractal described by Sierpinski in 1915 and appearing in Italian art from the 13th century” MathWorld
- Φτιάξτε ένα πρόγραμμα που να παράγει το παρακάτω σχήμα:



Sierpinski Sieve: Αναδρομική Λύση

- Ξεκίνα από ένα μαύρο τρίγωνο Τ1
- «Αφαίρεσε» το τρίγωνο που σχηματίζεται από τις τρεις μέσους των ακμών
- Επανέλαβε την ίδια διαδικασία στα τρία (υπο)τρίγωνα που σχηματίζονται, ας τα ονομάσουμε $T_{1,1}$, $T_{1,2}$, $T_{1,3}$
- Επανέλαβε αναδρομικά την διαδικασία από το βήμα 2

Γενική Ιδέα Αναδρομής

- Αναδρομική Συνάρτηση είναι μια συνάρτηση που έμμεσα ή άμεσα καλεί τον εαυτό της
  - Κάθε (κλήση) συνάρτησης λύνει ένα «μικρότερο» (υπό)πρόβλημα **του ίδιου τύπου.**
  - Συνδυάζουμε τα μικρότερα προβλήματα με τέτοιο τρόπο που να λύσουμε το συνολικό πρόβλημα
  - Κάποια στιγμή πρέπει να λύσουμε τα «πολύ μικρά προβλήματα» απευθείας
- Σε τι μας βοηθάνε οι αναδρομικές συναρτήσεις
  - Ο κώδικας γίνεται συνήθως πολύ πιο απλός και μπορούμε να λύσουμε πολύ εύκολα προβλήματα που θα φαινόταν πολύ δύσκολα με επανάληψη
  - Το μέγεθος (γραμμές κώδικα) της συνάρτησης συνήθως «ελαχιστοποιείται»

Παράδειγμα 1

```
int power(int x, int n)
{
    if (n == 0)
        return 1;
    else
        return (x * power(x, n-1));
}
```

Βασική Περίπτωση: το μικρότερο πρόβλημα, άμεση λύση

Λύσε αναδρομικά ένα μικρότερο πρόβλημα

Συγχώνευση λύσεων μικρότερων προβλημάτων

Το «μέγεθος/δυσκολία» του προβλήματος καθορίζεται από τον εκθέτη

Παράδειγμα 1: Επαναληπτική Λύση

```
int power( int x, int n)
{
    int i = 1;
    int result = 1;

    /* check for n >= 0 */
    for ( i = 1; i <= n; i++)
    {
        result *= x;
    }
    return result;
}
```

«Γραμμική» Αναδρομή

- **Συνθήκες βάσης (base cases).**
  - Εύρεση συνθηκών βάσης (τουλάχιστο μια).
  - Κάθε «αλυσίδα» κλήσεων **πρέπει** τελικά να φτάσει σε μια συνθήκη βάσης.
- ***Μια αναδρομική κλήση***
  - Μπορεί να είναι απόφαση μεταξύ πολλών αλλά σε κάθε εκτέλεση συνάρτησης γίνεται μια αναδρομή
  - Κάθε κλήση προχωρά προς μια συνθήκη βάσης πλησιάζοντας προς τη λύση

Algorithm LinearSum(*A, n*):

Είσοδος:

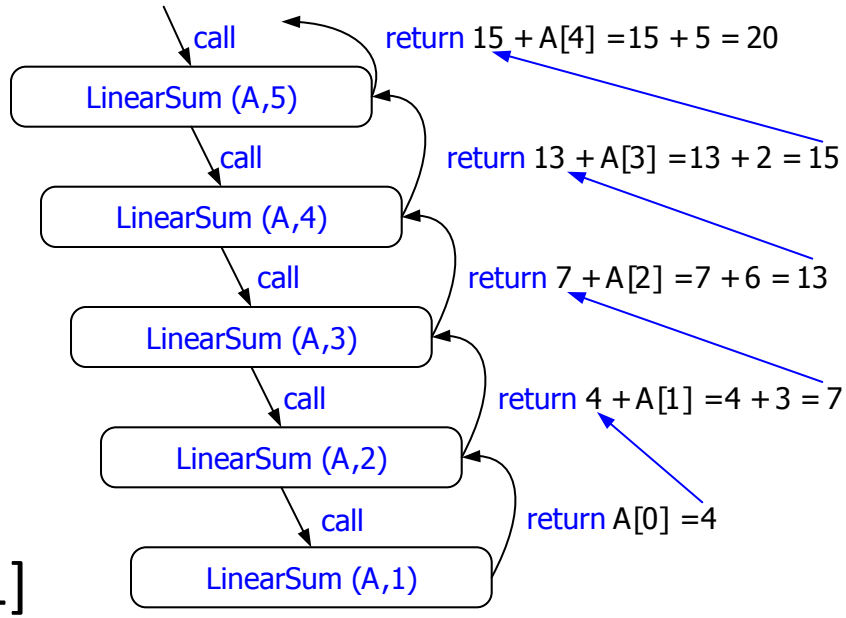
Πίνακας ακεραίων με τουλάχιστο 1 στοιχείο

Έξοδος

Το άθροισμα των στοιχείων

```
if n = 1
    return A[0];
else
    return LinearSum(A, n - 1) + A[n - 1]
```

recursion trace:



Algorithm ReverseArray(*A, i, j*):

Είσοδος :

Πίνακας *A* και ακέραιοι, θετικοί δείκτες *i* και *j*

Έξοδος :

Η αναστροφή του πίνακα A από *i* εώς *j*

if *i* < *j* **then**

Swap *A*[*i*] and *A*[*j*]

ReverseArray(*A, i* + 1, *j* - 1)

return

Ορίσματα αναδρομής

- Κατάλληλος σχεδιασμός παραμέτρων συνάρτησης για αναδρομή
- Μερικές φορές χρειάζονται επιπρόσθετοι παράμετροι
- Π.χ. φτιάξαμε την ReverseArray(*A, i, j*), και όχι την ReverseArray(*A*).

Στοίβα και Αναδρομή

- Σε αναδρομικές συναρτήσεις η στοίβα μπορεί να περιέχει περισσότερα από ένα stack frames της **ίδιας** συνάρτησης
- Η αναδρομή απλουστεύει το γράψιμο του κώδικα αλλά η κλήση συνάρτησης κοστίζει
- Κάθε αναδρομική συνάρτηση μπορεί να υλοποιηθεί και επαναληπτικά (χωρίς αναδρομή) με χρήση στοίβας

- ΠΡΟΣΟΧΗ: η power και factorial δεν ενδείκνυνται για αναδρομική υλοποίηση, παρουσιάζονται μόνο ως παραδείγματα (γιατι;).

The Fibonacci example

$$F_n = \begin{cases} 0 & \text{if } n = 0; \\ 1 & \text{if } n = 1; \\ F_{n-1} + F_{n-2} & \text{if } n > 1. \end{cases}$$

Input: $n \in \mathbb{N}$

Output: The nth Fibonacci number

```
if n < 2 then
    return 1
else
    return fib(n − 1) + fib(n − 2)
end if
```

Ακολουθία Fibonacci

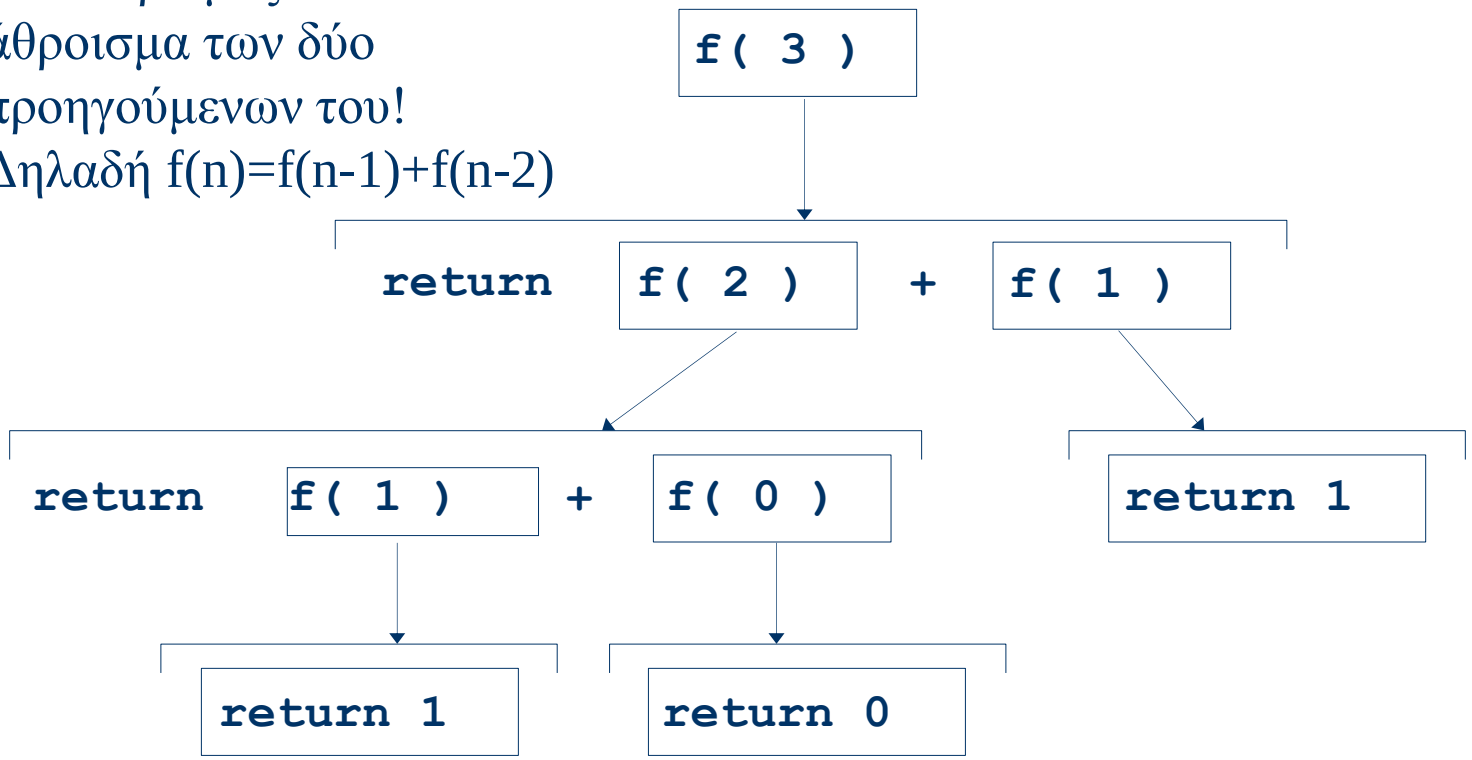
- Κάθε αριθμός είναι άθροισμα των δύο προηγούμενων του!
- Δηλαδή $f(n)=f(n-1)+f(n-2)$
- Αρκεί αυτό;

- $f(1)=1$, $f(0)=0$, $f(n)$ ορίζεται για $n \geq 0$

```
int fib(int k)
{
// Base Cases:
// If k == 0 then fib(k) = 0.
// If k == 1 then fib(k) = 1.
if (k < 2)
        return k;
// Recursive Case:
// If k >= 2 then fib(k) = fib(k-1) + fib(k-2).
else
        return fib(k-1) + fib(k-2);
}
```

Ακολουθία Fibonacci

Κάθε αριθμός είναι άθροισμα των δύο προηγούμενων του!
Δηλαδή f(n)=f(n-1)+f(n-2)



The Fibonacci example

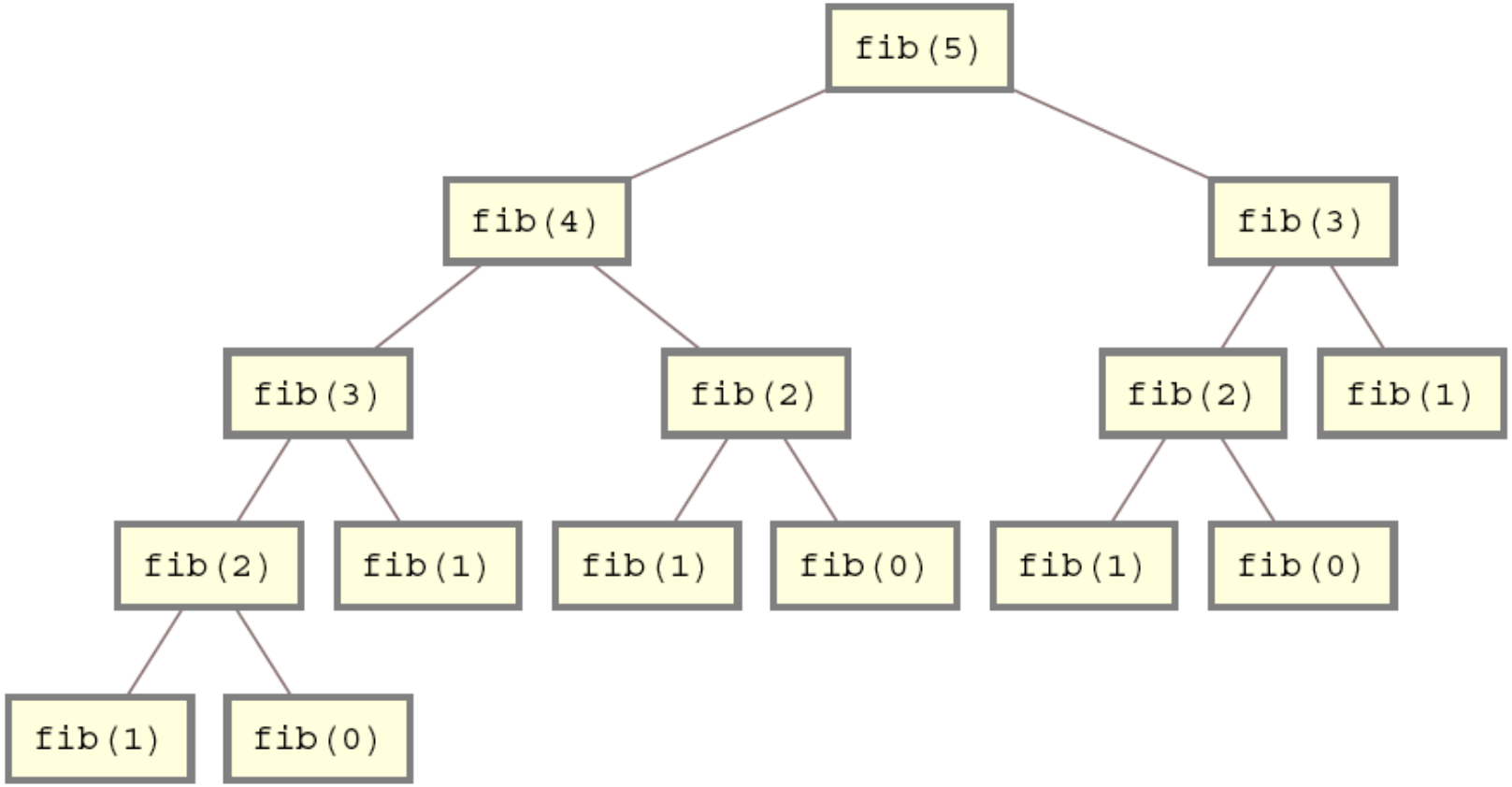


Figure from I. Oikonomides MSc

Πως μπορεί να γίνει επαναληπτικά;

```
1  /* Fig. 5.15: fig05_15.c
2     Recursive fibonacci function */
3  #include <iostream.h>
4   using namespace std;
5  int fibonacci( int );
6
7  int main()
8  {
9     int result, number;
10
11    cout << "Enter an integer: " ;
12    cin >> number ;
13    result = fibonacci( number );
14    cout << "Fibonacci”<<number “=“ << result << endl;
15    return 0;
16 }
17
18 /* Recursive definition of function fibonacci */
19 int fibonacci( int n )
20 {
21    if ( n == 0 || n == 1 )
22       return n;
23    else
24       return fibonacci( n - 1 ) + fibonacci( n - 2 );
25 }
```

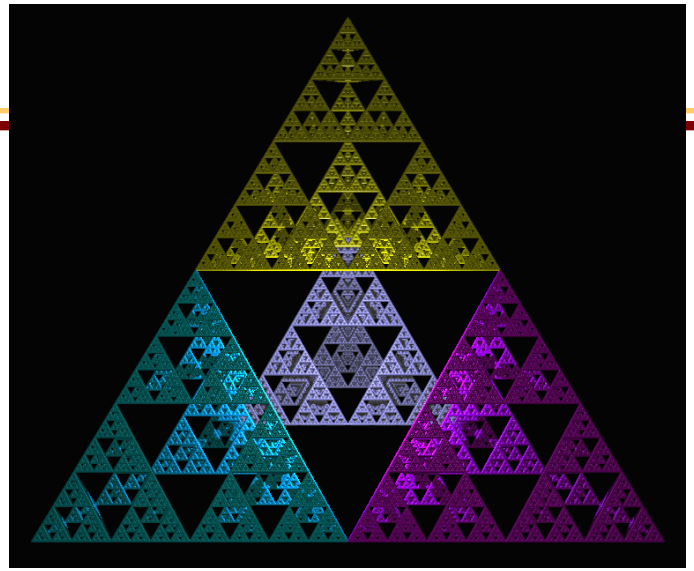
Binary Search – Υλοποίηση με επανάληψη

```
int binaryLoopSearch(int p[], int searchkey, int low, int high)
{
    int middle;
    while ( low <= high )
    {
        middle = (low + high ) / 2;

        if (searchkey == p[middle])
            return middle;
        else if (searchkey < p[middle] )
            high = middle – 1;
        else
            low = middle + 1;
    }
    return -1;
}
```

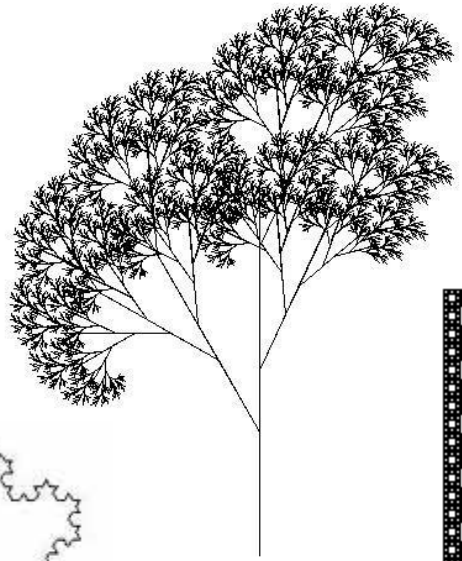
Binary Search – Υλοποίηση με αναδρομή

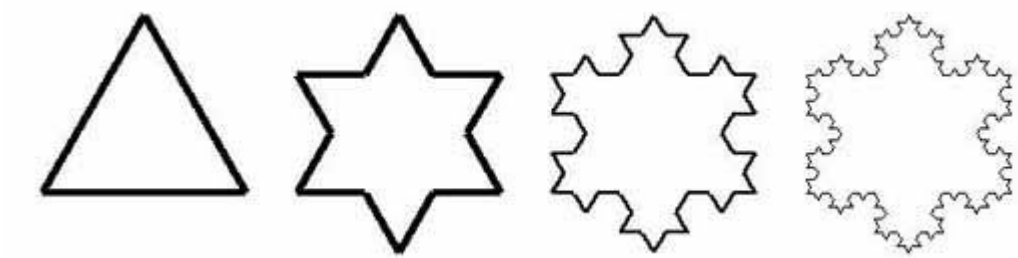
```
int binarySearch(int p[], int searchkey, int low, int high)
{
    int middle;

    middle = (low + high ) / 2;
     if (high < low)
            return -1;
   if (searchkey == p[middle])
            return middle;
    else if (searchkey < p[middle])
            return binarySearch(p, searchkey, low, middle-1);
    else
            return binarySearch(p, searchkey, middle+1,high);

    return -1;
}
```

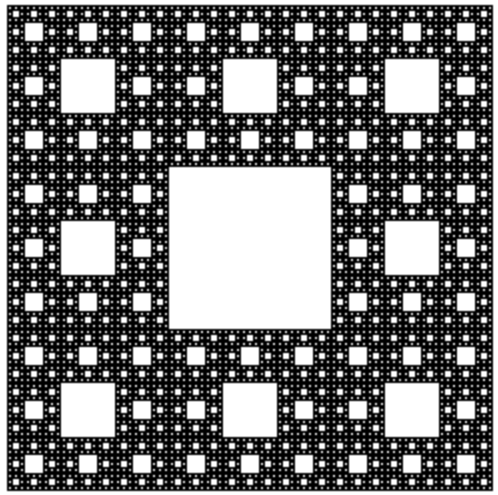
Άλλα παραδείγματα



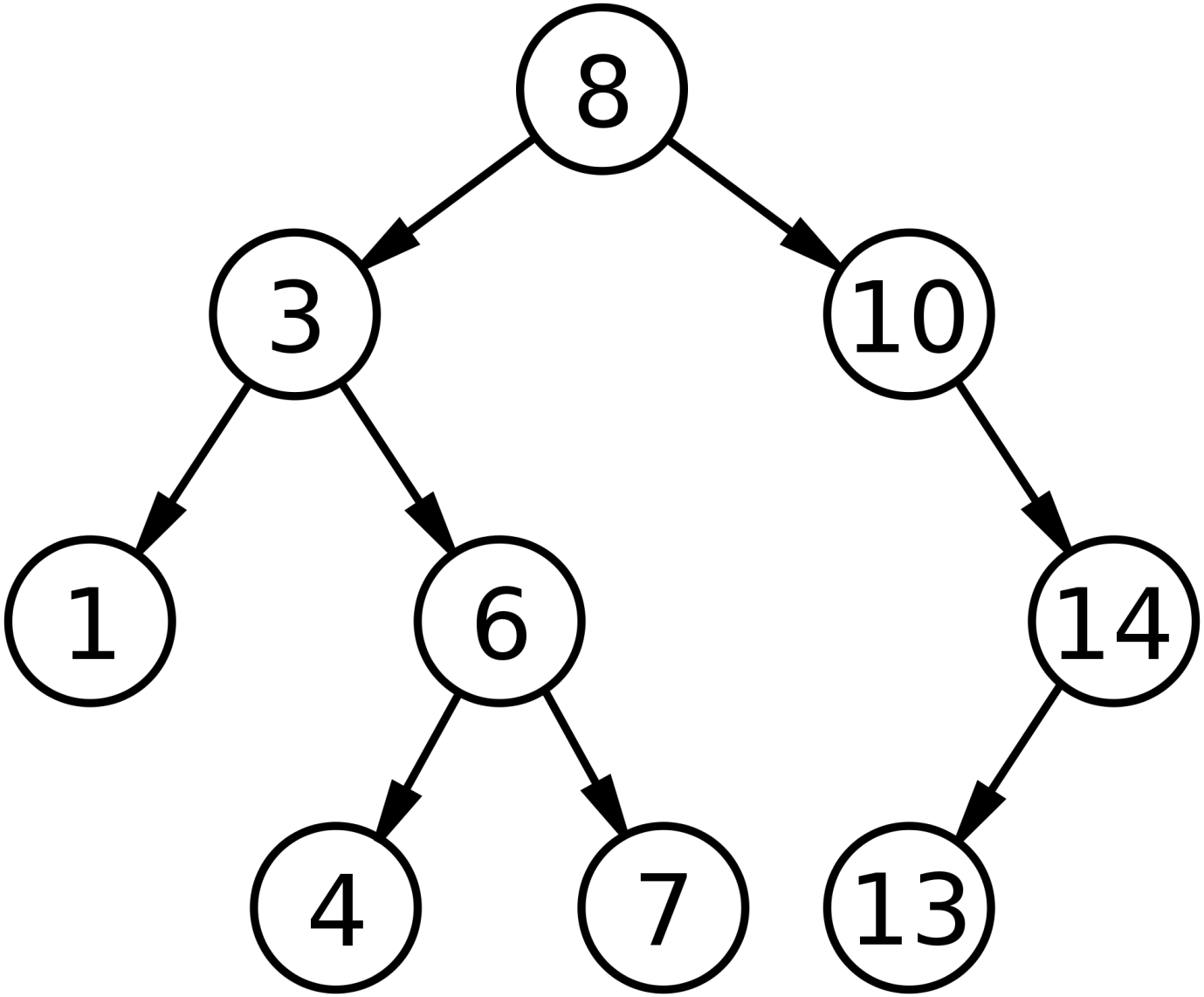
- Πως θα λύναμε αναδρομικά
  - Αναζήτηση
  - Ταξινόμηση
  - **Διαχείριση λιστών/δέντρων κτλ**
  - Βρες την έξοδο από τον λαβύρινθο
  - **Balance**
  - **Καρκινικές γραφές**
  - Φτιάξε ένα fractal δέντρο
  - Fractals







Binary trees



Binary trees

O(log N) < O(N)

Binary search trees: O(log N) search and insert

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

Print binary search tree

```
void print_inorder(tree_node *p)
{
   if (p != NULL)
   {
        print_inorder(p->left);
        printf("%d",p->data);
        print_inorder(p->right);
   }
}
```

Delete tree

```
void deleteword(struct wordtree **node) {
    struct wordtree *temp = NULL;
    if (node != NULL)  {
        if(*node != '\0') {
            if((*node)->right != NULL) {
                temp = *node; deleteword(&temp->right);
            }
            if((*node)->left != NULL) {
                temp = *node; deleteword(&temp->left);
            }
            if((*node)->word != NULL)
                free((*node)->word);
            if((*node)->firstline != NULL)
                deletelist((*node)->firstline);
            free(*node);
            *node = NULL;
        }
    }
}
```

C code for the sum_list program

```
int sum_list(struct list_node *l)
{
   if(l == NULL)
        return 0;
   return l->data + sum_list(l->next);
}
```

```
void free_list(struct list_node *l)
{
   struct list node *tmp;
   if(l == NULL)
        return 0;
   tmp = l->next;
   free(l);
   free_list(tmp);
}
```