

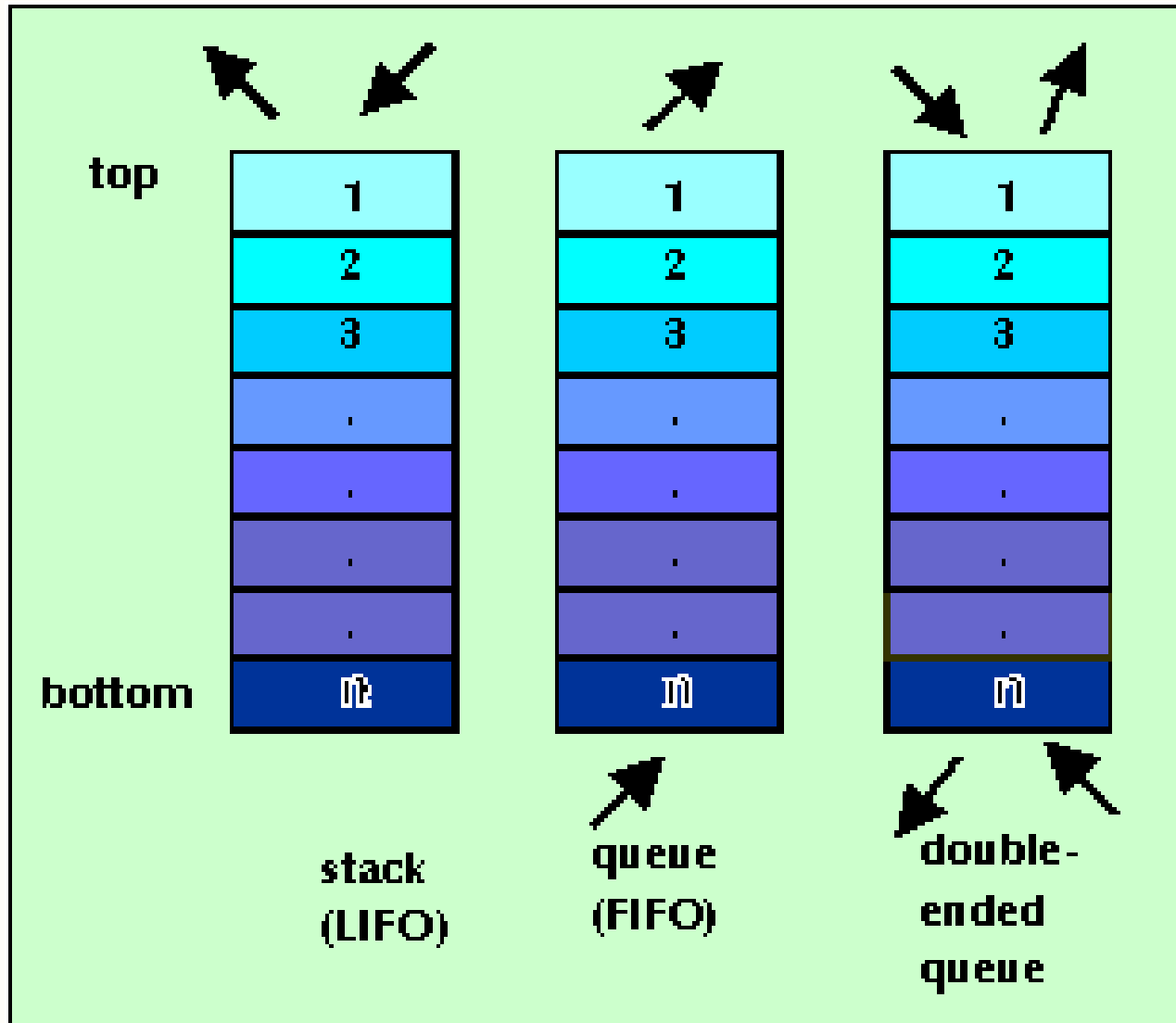


Διάλεξη 06: Συνδεδεμένες Λίστες & Εφαρμογές Στοιβών και Ουρών

Στην ενότητα αυτή θα μελετηθούν τα εξής επιμέρους θέματα:

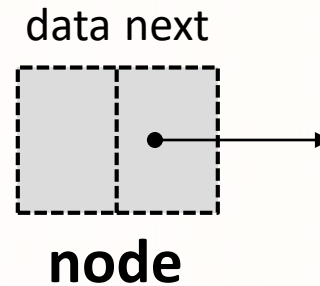
- Υλοποίηση ΑΤΔ με Συνδεδεμένες Λίστες
- Εφαρμογή Στοιβών 1: Αναδρομικές συναρτήσεις
- Εφαρμογή Στοιβών 2: Ισοζυγισμός Παρενθέσεων
- Εφαρμογή Στοιβών 3: Αντίστροφος Πολωνικός Συμβολ.
- Εφαρμογή Ουρών 1: Διερεύνηση κατά Πλάτος

Ανασκόπηση

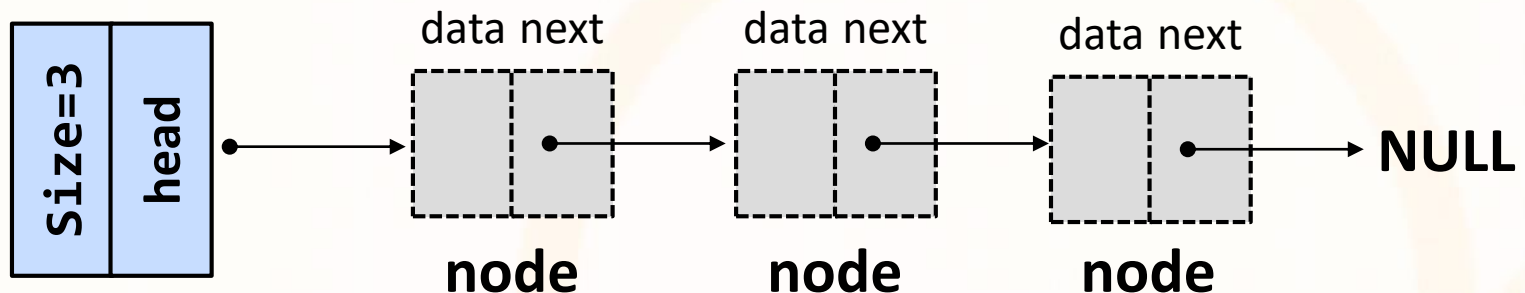


Συνδεδεμένες Λίστες

- Μία απλή συνδεδεμένη λίστα είναι μία συμπαγής δομή δεδομένων η οποία αποτελείται από μία ακολουθία κόμβων.
- Κάθε κόμβος αποτελείται από:
 - **Δεδομένα**
 - **Δείκτη/Αναφορά** προς τον επόμενο κόμβο της λίστας
- Επίσης μπορεί να χρησιμοποιηθεί μία βοηθητική δομή που να αποθηκεύει χρήσιμες πληροφορίες, π.χ., αναφορά στον πρώτο κόμβο, μέγεθος των στοιχείων, κτλ.

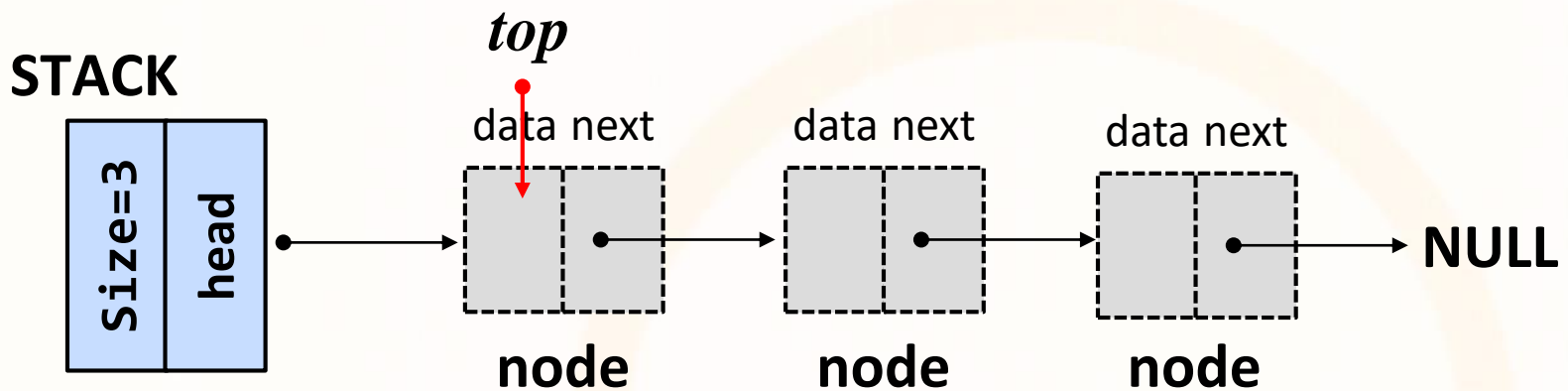


ADT



Στοιίβα ως Συνδεδεμένη Λίστα

- Μπορούμε να υλοποιήσουμε μία **Στοιίβα** με μία απλά συνδεδεμένη λίστα
- Κάθε κόμβος αποτελείται από ένα στοιχείο (στοιχεία της στοίβας) και από ένα δείκτη (προς τον επόμενο κόμβο της στοίβας). Η κορυφή της στοίβας είναι ο πρώτος κόμβος της λίστας,
- Χρησιμοποιούμε μια μεταβλητή για να φυλάγουμε στοιχεία σχετικά με τη στοίβα π.χ. **μέγεθος** (size) και δείκτη προς την **κορυφή της στοίβας (head)** που βρίσκεται και το στοιχείο **top**.



Υλοποίηση Στοιβάς με Συνδεδεμένη Λίστα

- Με χρήση εσωτερικής κλάσης για τους κόμβους

```
public class Stack<E> implements IStack<E>
private class StackNode<E>{
    private E obj;
    private StackNode<E> next;
    StackNode(E obj, StackNode<E> head) {
        this.obj = obj;
        this.next = head;
    }
    public E getElement(){
        return this.obj;
    }
}
private StackNode<E> head;
private int size;

public Stack() {
    head = null;
    size=0;
}

public boolean isEmpty(){
    return this.size==0;
}
```

```
public void makeEmpty(){
    this.head = null;
    this.size=0;
}

public int size() {
    return this.size;
}

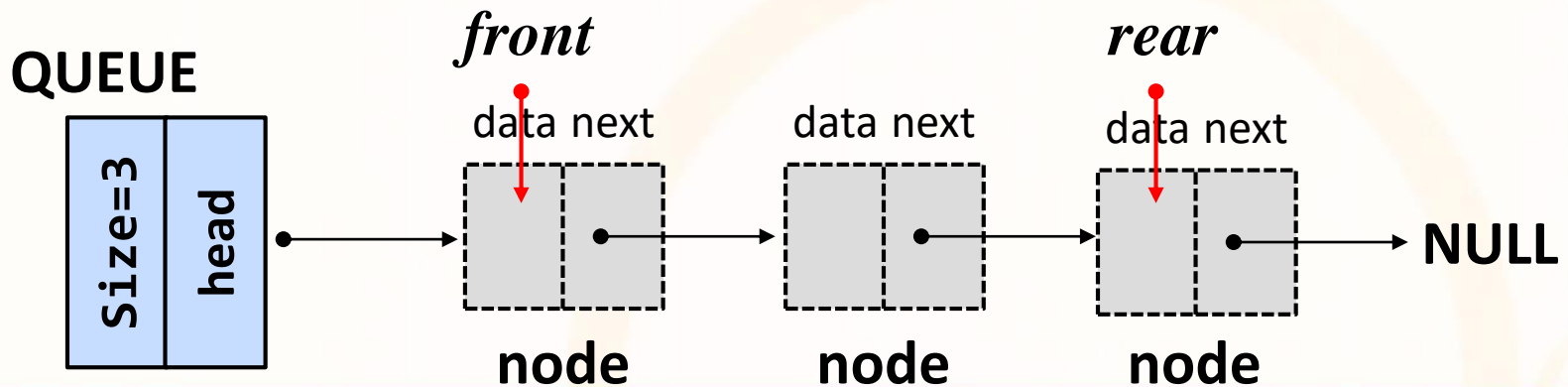
public E top() {
    return isEmpty()? null :
        head.getElement();
}

public void push(E obj) {
    StackNode<E> newNode = new
        StackNode<E>(obj, head);
    this.head = newNode;
    size+=1;
}

public void pop() {
    if (!isEmpty()) {
        this.head = this.head.next;
        size-=1;
    }
}
```

Ουρά ως Συνδεδεμένη Λίστα

- Μπορούμε να υλοποιήσουμε μία **Ουρά** με μία απλά συνδεδεμένη λίστα
- Κάθε κόμβος αποτελείται από ένα στοιχείο (στοιχεία της ουράς) και από ένα δείκτη (προς τον επόμενο κόμβο της ουράς).
 - Το πρώτο στοιχείο αποθηκεύεται στον πρώτο κόμβο
 - Κάθε νέο στοιχείο αποθηκεύεται στο πίσω μέρος
- Χρησιμοποιούμε μια μεταβλητή για να φυλάγουμε στοιχεία σχετικά με την ουρά π.χ., **μέγεθος** (size) και δείκτες προς την **κορυφή της ουράς (head)** που βρίσκεται και το στοιχείο **front** και προς το **τελευταίο στοιχείο της ουράς (rear)**.



Εφαρμογή Στοίβας 1: Αναδρομικές Διαδικασίες

- Οι στοίβες βρίσκουν μεγάλη χρήση στην πληροφορική για δημιουργία άλλων δομών και σε βασικό λογισμικό.
- Κλασικό παράδειγμα αφορά την **κλήση υποπρογραμμάτων** (function calls) και **αναδρομικών διαδικασιών**.
 - Σε κάθε κλήση οποιασδήποτε συνάρτησης ένα σύνολο από λέξεις (stack frame) φυλάσσεται σε μια στοίβα, από όπου μπορεί να ανασυρθεί.
 - Όταν μια συνάρτηση καλεί μια άλλη συνάρτηση οι **παράμετροι της συνάρτησης**, η **διεύθυνση επιστροφής** και οι **τοπικές μεταβλητές** της καλούσας συνάρτησης φυλάσσονται μέσα στη στοίβα του προγράμματος.
 - Έτσι, όταν η κληθείσα **συνάρτηση τερματίσει**, το περιβάλλον την καλούσας συνάρτησης **ανασύρεται** από τη στοίβα για να συνεχιστεί κανονικά η εκτέλεσή της.

Εφαρμογή Στοίβας 1: Αναδρομικές Διαδικασίες

- Όταν ένα πρόγραμμα φορτώνεται στην μνήμη του υπολογιστή, τότε η δεσμευμένη μνήμη οργανώνεται σε τρεις περιοχές (segments):
a) text (code) segment, b) stack segment, and c) heap segment.
- Το *text segment* περιέχει τον κώδικα υπό εκτέλεση, ενώ το *heap segment* επιτρέπει στον πρόγραμμα να δεσμεύσει μνήμη. Αυτή η μνήμη παραμένει μέχρι να αποδεσμευτεί από τον garbage collector ή να τερματιστεί το πρόγραμμα.
- Το *stack segment* από την άλλη χρησιμεύει για να φυλάγονται όλες οι πληροφορίες σχετικές με την εκτέλεση συναρτήσεων.

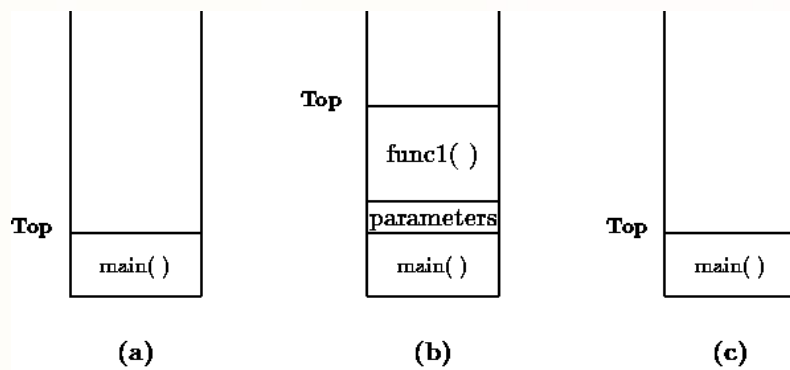
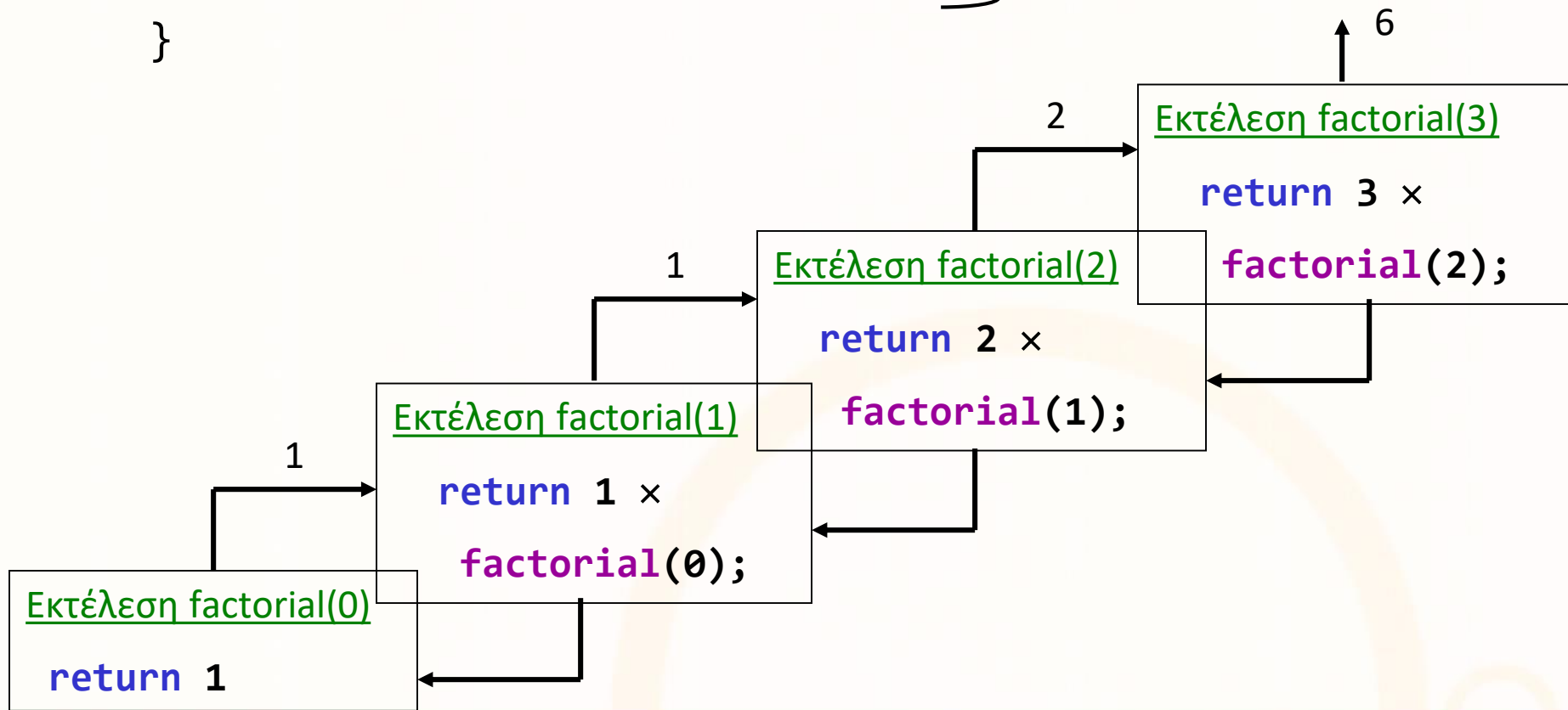


Figure 14.13: Organization of the Stack

Η εντολή ulimit -a στο unix δείχνει τους πόρους που είναι διαθέσιμους σε ένα πρόγραμμα: virtual memory, stack, cpu time, processes, κτλ.

Παράδειγμα: Παραγοντικός Αριθμός με Αναδρομή

```
int factorial ( int n ) {  
    if (n == 0) } Βήμα Τερματισμού  
        return 1;  
    else } Αναδρομικό Βήμα  
        return n x factorial(n-1);  
}
```



Αναδρομή και Διαχείριση Μνήμης

Στοίβα

factorial(2)

main

factorial(2)

return 2 * factorial(1)

factorial(2)

main

factorial(2)

return 2 * factorial(1)

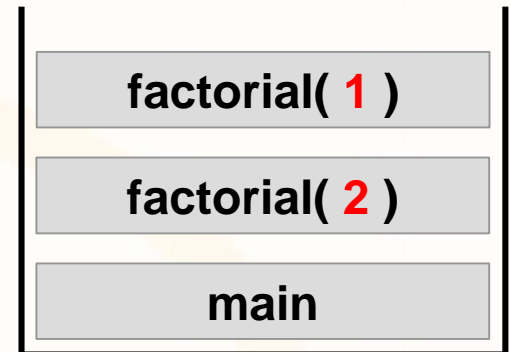
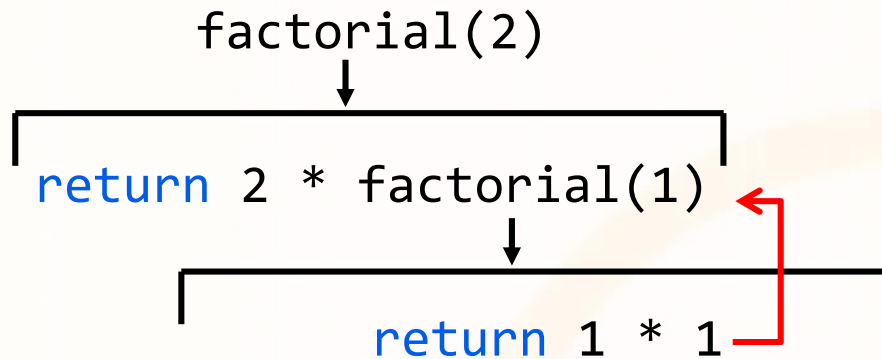
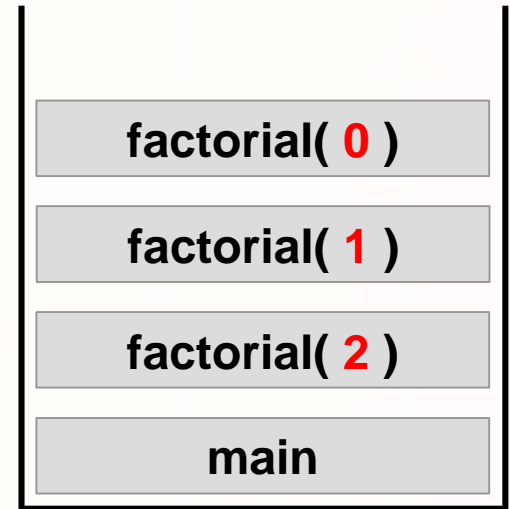
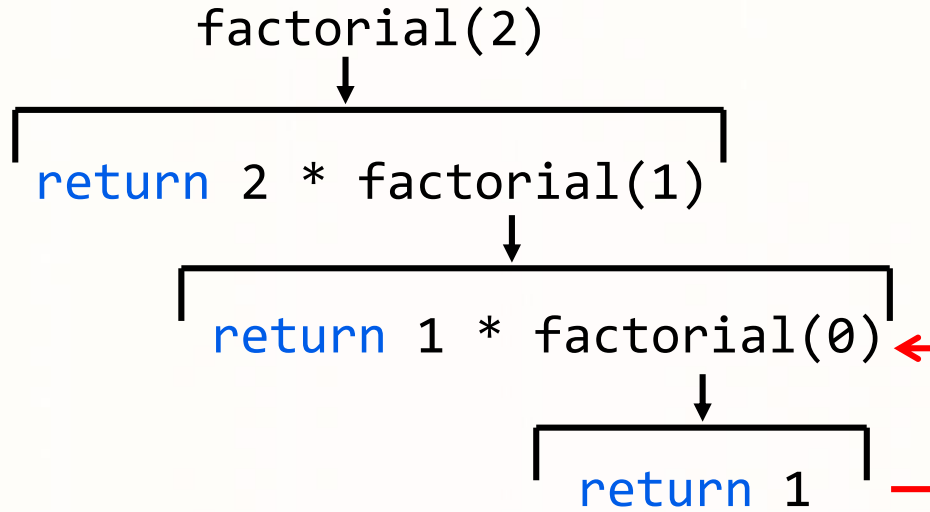
return 1 * factorial(0)

factorial(1)

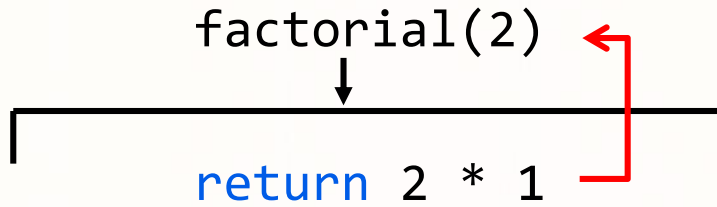
factorial(2)

main

Αναδρομή και Διαχείριση Μνήμης



Αναδρομή και Διαχείριση Μνήμης



Στοίβα

2



Εφαρμογή Στοιβάς 2: Ισοζυγισμός Παρενθέσεων

- Ο έλεγχος σύνταξης (π.χ. ενός προγράμματος) απαιτεί να ταιριάξουμε σύμβολα/λέξεις όπως:

`begin` με `end`

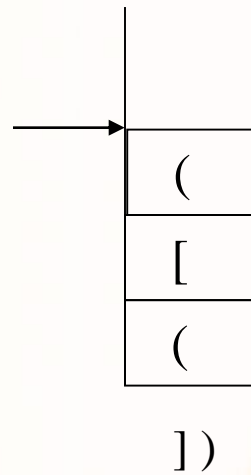
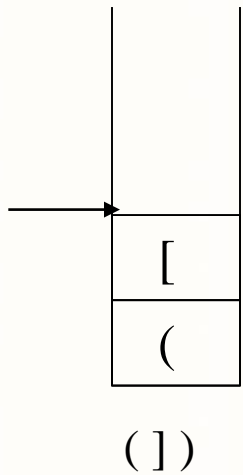
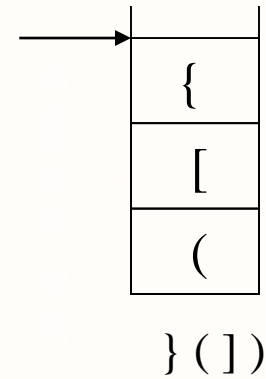
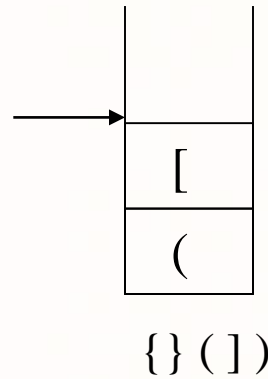
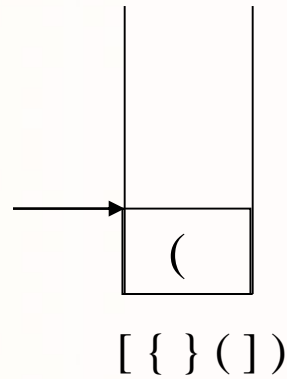
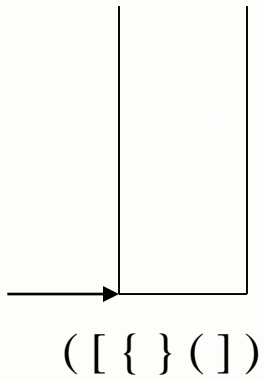
`else` με `if`

παρενθέσεις `{` με `}`

- Ας υποθέσουμε την ύπαρξη του συνόλου χαρακτήρων: `{, }, [,], (,)`.
- Πρόβλημα: να διαπιστώσετε αν μια συμβολοσειρά που περιέχει τους πιο πάνω χαρακτήρες είναι ισοζυγισμένη, δηλαδή όλες οι παρενθέσεις ταιριάζουν.
- π.χ. `{ [] }`
`([{ } { } []))`
`([] { () })`

Εφαρμογή Στοίβας 2: Ισοζυγισμός Παρενθέσεων

- Ανά πάσα στιγμή, η στοίβα περιέχει όλες τις `αριστερές` παρενθέσεις που δεν έχουν ακόμη `ταιριαστεί`.



Εφαρμογή Στοίβας 2: Ισοζυγισμός Παρενθέσεων

- Λύση βασισμένη σε στοίβες

```
public class IsozygismosParentheseon {  
    public static void main(String[] args) {  
        Stack<Character> stack = new Stack<Character>();  
        String test1 = "{ [ ] }"; String test2 = "( [ { } { } [ ] ) )";  
        String test3 = "( [ ] { ( ) } )";  
        for(Character c: test3.toCharArray()) {  
            if( !(c=='(' || c=='[' || c=='{'  
                || c=='}' || c==']' || c==')') ) continue;  
            if( c=='(' || c=='[' || c=='{' ) { stack.push(c); }  
            else {  
                if( stack.isEmpty() ) {System.out.println("Error"); return; }  
                Character check = stack.top();  
                stack.pop();  
                if( !match(check,c) ) {System.out.println("Error"); return; }  
            } }  
            if( stack.isEmpty() ) {System.out.println("Success"); }  
            else {System.out.println("Error"); }  
        }  
        public static boolean match(Character a, Character b){  
            return ( (a=='(' && b==')') || (a=='{' && b=='}') || (a=='[' && b==']') );  
        }  
    }  
}
```

Εφαρμογή Στοιβάς 3: Αντίστροφος Πολωνικός Συμβ.

- Ο αντίστροφος πολωνικός συμβολισμός (Jan Lukasiewicz, 1951) είναι μια μέθοδος αναπαράστασης αριθμητικών παραστάσεων που δεν κάνει χρήση παρενθέσεων.

π.χ. αντί

$\alpha + \beta$

γράφουμε

$\alpha \beta +$

$(\alpha + \beta) * \gamma$

γράφουμε

$\alpha \beta + \gamma *$

$\alpha + (\beta * \gamma)$

γράφουμε

$\alpha \beta \gamma * +$

$\alpha * \beta + \gamma * \delta$

γράφουμε

$\alpha \beta * \gamma \delta * +$

$60 - ((5+3) * (2+4))$

γράφουμε

$60 5 3 + 2 4 + * -$

- Η συνήθης μορφή μιας παράστασης ονομάζεται **ενθεματική** (infix), γιατί οι τελεστές των πράξεων τίθενται *μεταξύ* των τελεστέων. Η πολωνική μορφή μιας παράστασης ονομάζεται **μεταθεματική** (postfix) γιατί οι τελεστές βρίσκονται *μετά* από τους τελεστέους.

Εφαρμογή Στοίβας 3: Αντίστροφος Πολωνικός Συμβ.

- Ζητείται η υλοποίηση ενός προγράμματος που να υπολογίζει αριθμητικές εκφράσεις αντίστροφου πολωνικού συμβολισμού. Για τον υπολογισμό τέτοιων εκφράσεων μπορεί να χρησιμοποιηθεί η έννοια της **στοίβας**.
- Η δομή του προγράμματος θα έχει τη μορφή της παρακάτω ανακύκλωσης:

```
MakeEmpty ();
```

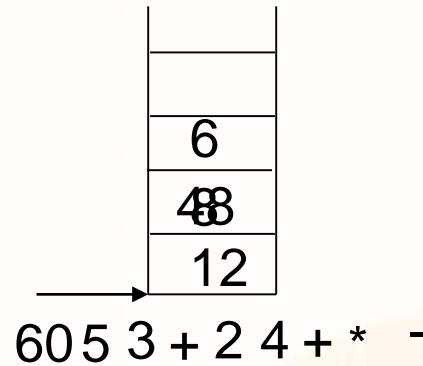
```
while (c = next character && c is not EOF) {  
    switch (c) {  
        integer:    Push (c) ;  
        *:          Push (Pop () * Pop ()) ;  
        +:          Push (Pop () + Pop ()) ;  
        ...:        ...  
        default:    error  
    }  
}  
print Top ();
```

υποθέτουμε
υλοποίηση της Pop
η οποία και
επιστρέφει και
αφαιρεί τον κόμβο
κορυφής στοίβας

Χρόνος Εκτέλεσης: **$O(n)$**

Εφαρμογή Στοιβάς 3: Αντίστροφος Πολωνικός Συμβ.

- Έστω η παράσταση $6053 + 24 + * -$.
(Ή σε ενθεματική μορφή: $60 - (5+3)*(2+4)$)



Εφαρμογή Στοίβας 3: Αντίστροφος Πολωνικός Συμβ.

- Υποθέτουμε την ύπαρξη
 - ακεραίων,
 - παρενθέσεων (,),
 - + , *
- Κανόνας: το * έχει μεγαλύτερη προτεραιότητα από το + .
- Στόχος: μετατροπή της ενθεματικής σε μεταθεματική μορφή. π.χ.
 $60 + ((5 + 3) * (2 + 4))$



60 5 3 + 2 4 + * +

- Χρησιμοποιώντας στοίβες η μετατροπή μπορεί να επιτευχθεί σε χρόνο $O(N)$.

Εφαρμογή Στοίβας 3: Αντίστροφος Πολωνικός Συμβ.

- Δεδομένο Εισόδου: λίστα L που περιέχει μια ενθεματική παράσταση

`MakeEmpty();`

για κάθε c στην L κάνε {

αν ο c είναι `integer`: τύπωσέ τον

`(` : γράψε `'('` στη στοίβα S

`)` : ανέτρεξε στη στοίβα αφαιρώντας και τυπώνοντας όλα τα στοιχεία μέχρι την πρώτη `'('` που θα συναντήσεις την οποία αφάιρεσε χωρίς να τυπώσεις.

`+` : αφάιρεσε και τύπωσε όλα τα `'+'` και `'*'` που βρίσκονται στη στοίβα μέχρι να συναντήσεις μια `'('` ή να αδειάσει η στοίβα και στη συνέχεια πρόσθεσε το `'+'` στη στοίβα

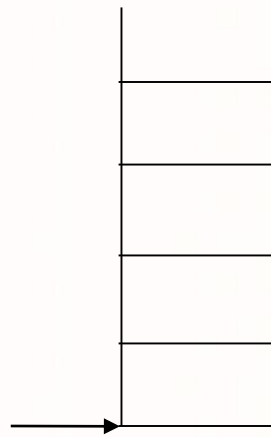
`*` : αφάιρεσε και τύπωσε όλα τα `'*'` που βρίσκονται στη στοίβα μέχρι να συναντήσεις μια `'('` ή ένα `'+'` ή να αδειάσει η στοίβα και στη συνέχεια πρόσθεσε το `'*'`

}

αφάιρεσε και τύπωσε όλα τα στοιχεία που παραμένουν στη στοίβα.

- Χρόνος Εκτέλεσης: **$O(n)$**

Εφαρμογή Στοίβας 3: Αντίστροφος Πολωνικός Συμβ.



$a + b * c * (d + e)$

Εφαρμογές Ουράς

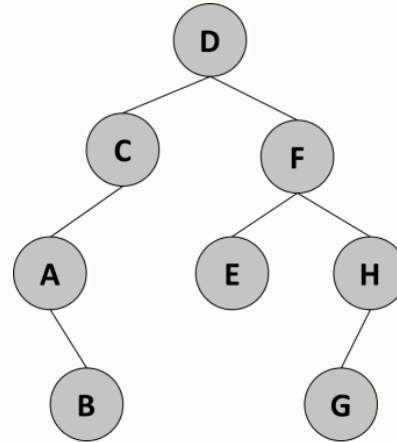
- Παραδείγματα Πραγματικού Κόσμου
 - Γραμμή Πελατών
 - Ασανσέρ
 - Μηχανές Λεφτών
 - Μηχανές Ποτών/Φαγητών
 - ...
- Παραδείγματα σε Υπολογιστές/Δίκτυα
 - Διαχείριση Περιορισμένων Πόρων, π.χ., bandwidth
 - Προσωμείωση Διακριτών Γεγονότων (Discrete event simulation)
 - Χρήση σε Αλγόριθμους, π.χ., BFS, Dijkstra, Huffman Coding, A*
 - ...

Εφαρμογή Ουράς 1: Διερεύνηση κατά Πλάτος (BFS)

- Στη θεωρία γράφων, η διερεύνηση κατά βάθος (breadth-first search (BFS)) επιτυγχάνει την αναζήτηση ανά επίπεδο

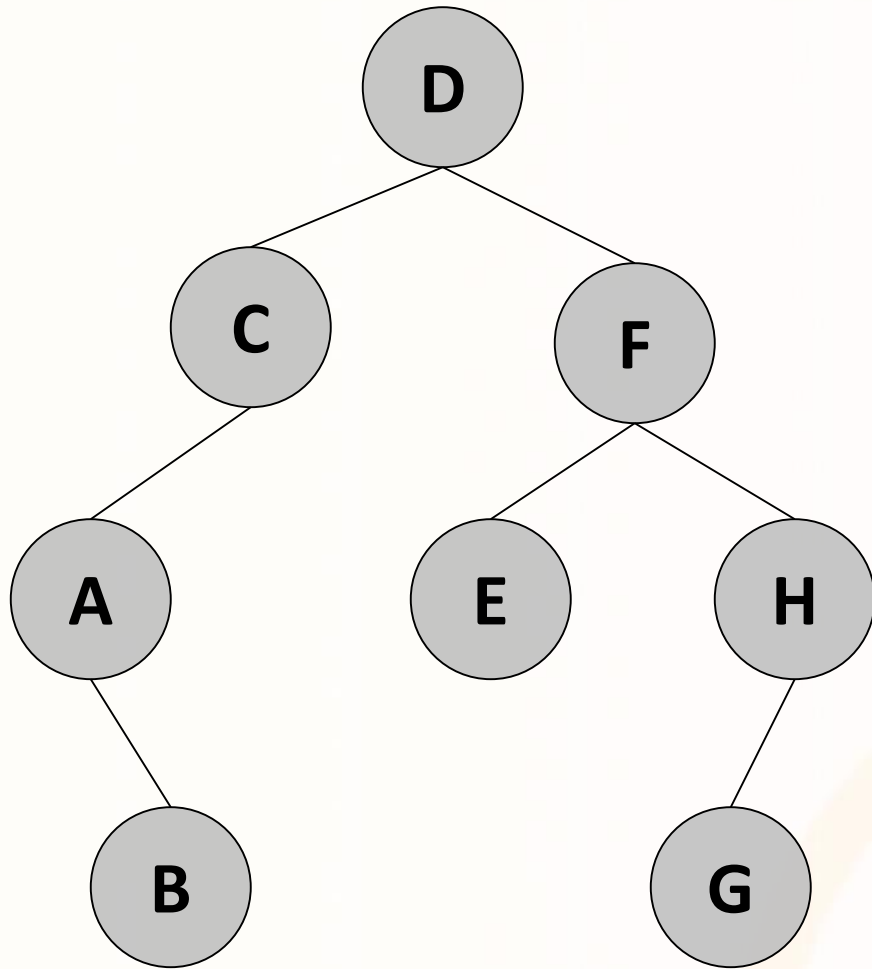
- Παράδειγμα

- Επίπεδο 1: D
- Επίπεδο 2: C, F
- Επίπεδο 3: A, E, H
- Επίπεδο 4: B, G



- Δεδομένου ενός κόμβου, μπορούμε κάθε φορά να έχουμε πρόσβαση στους απευθείας απόγονούς του
- Πως μπορεί να υλοποιηθεί;

Εφαρμογή Ουράς 1: Παράδειγμα Εκτέλεσης BFS



Ουρά Q

D

C,F

F, A

A, E, H

E, H, B

H, B

B, G

G

{}

Έξοδος Διαδικασίας

D

C

F

A

E

H

B

G