

# ДЪРВЕТА

ЛЕКЦИОНЕН КУРС “ПРОГРАМИРАНЕ НА JAVA”



# ИЗПОЛЗВАНЕ НА ДЪРВЕТАТА

- Бързо търсене и сортиране
- Създаване на структури при съставни данни:

Множествата данни притежават йерархична структура

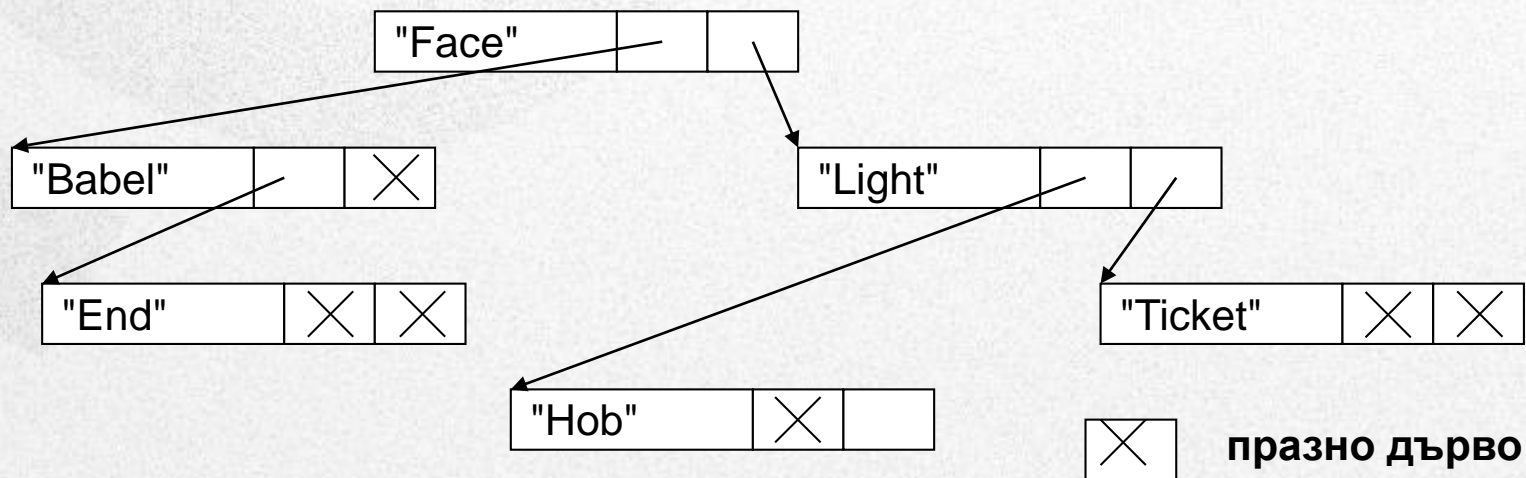
→ представяне като дърво

# ДЕФИНИЦИЯ: ДВОИЧНО ДЪРВО

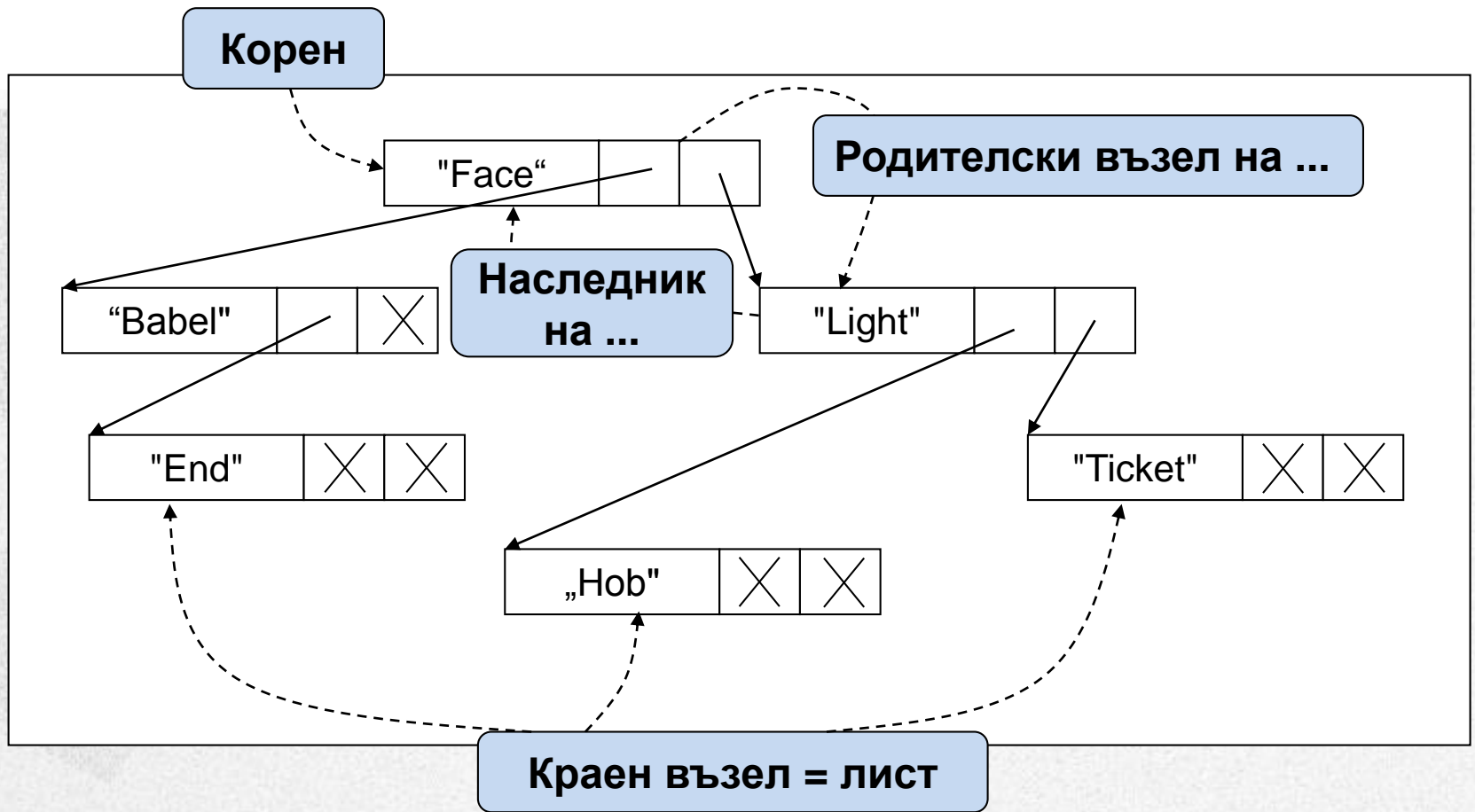
## Дефиниция двоично дърво:

- празно (без информация) или
- възел със съдържание (корен) и две двоични дървета (ляво и дясно поддърво)

## Пример:



# ОСНОВНИ ПОНЯТИЯ



- Всеки родител може да има повече наследници
- Всеки наследник има точно един родител

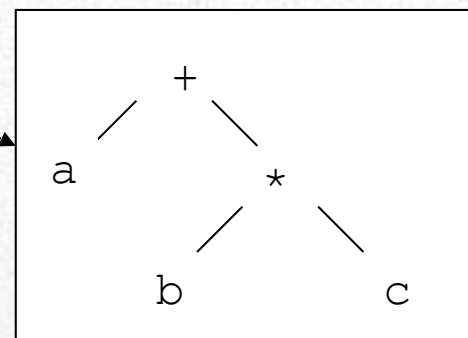
# СЪЗДАВАНЕ НА СТРУКТУРИ ПРИ СЪСТАВНИТЕ ДАННИ

**Пример:** обработвани изрази от компилатори

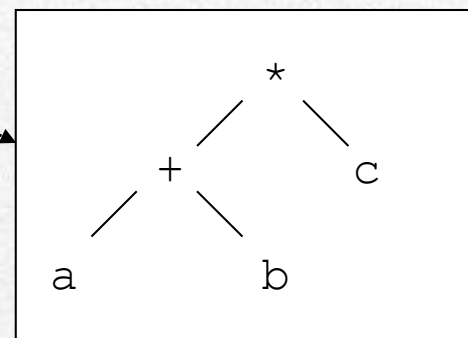


Дърветата представят приоритетни правила на операциите (синтактични дървета)

$a + b * c$



$(a + b) * c$





# ДЪРВЕТА КАТО АБСТРАКТНИ ТИПОВЕ ДАННИ

**Пример:** съдържание от тип 'String'

new1:		→ Tree
new2:	String	→ Tree
new3:	String x Tree x Tree	→ Tree
isEmpty:	Tree	→ boolean
left:	Tree	→ Tree
right:	Tree	→ Tree
value:	Tree	→ String

## Конструктори:

new1: създава празно дърво

new2: създава дърво с корен, ляво и дясно поддърво празни

new3: създаване произволно дърво

**Реализация:** Tree.java

# БЪРЗО ТЪРСЕНЕ И СОРТИРАНЕ

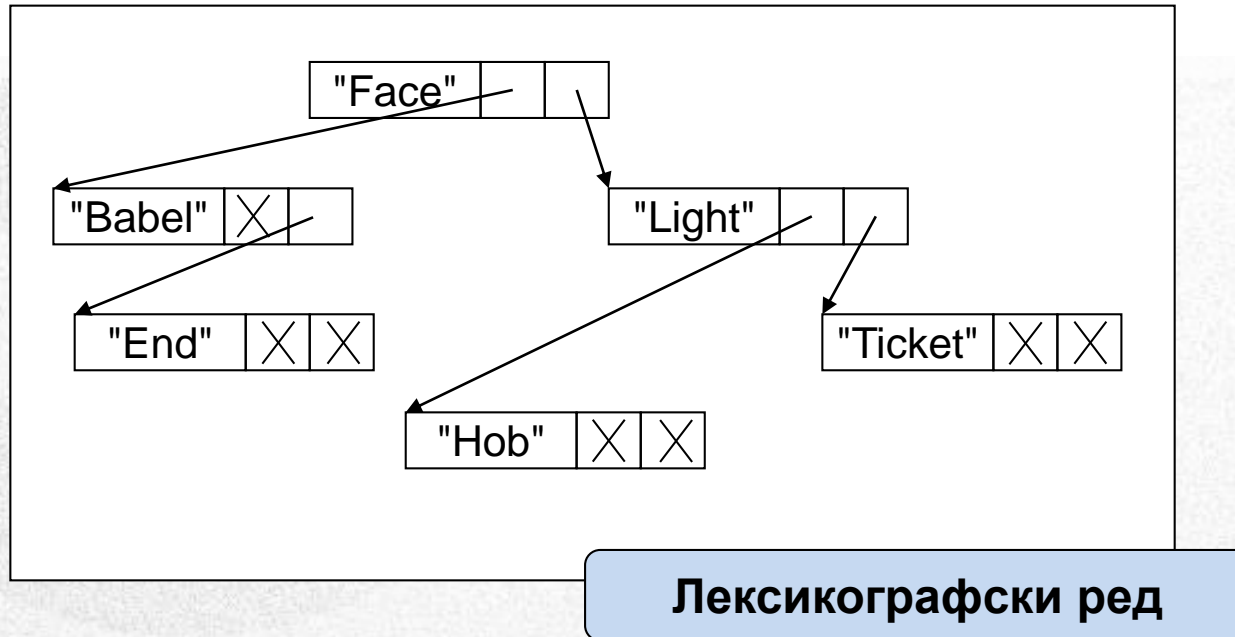
Основна задача в много програми:

Съхраняване и търсене на информация

	неограничена дължина	търсене	вмъкване	задраскване
Arrays	-	+	-	-
Свързани списъци	+	-	+	+
Дървета <sup>*)</sup>	+	+	+	(+)

<sup>\*)</sup> цената: на информационна единица → 2 указателя

# СОРТИРАНИ ДЪРВЕТА



Нарастващо сортирано двоично дърво:

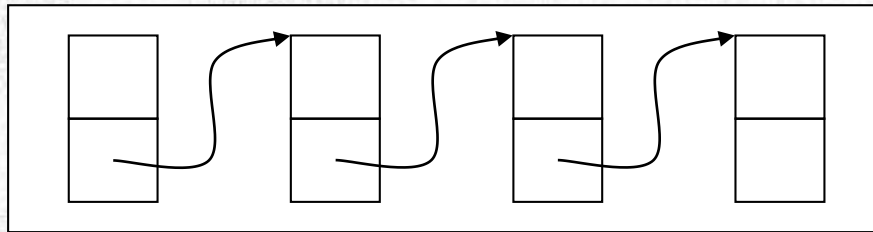
- Дървото е празно *или*
- Дървото се състои само от един възел *или*
- Съдържанието на възела е по-голямо от съдържанието на всички възли на лявото поддърво и по-малко от съдържанието на тези в дясното. Лявото и дясното поддървета са сортирани.



# ДЪРВЕТАТА СА РЕКУРСИВНИ СТРУКТУРИ ДАННИ:

## ОБРАБОТВАЩИТЕ АЛГОРИТМИ РЕКУРСИВНИ

- Итеративните алгоритми в общия случай неестествени
- Друга ситуация при списъците:



### Възприемане като:

- Рекурсивни структури:  
(списък: указател към клетка със съдържание + (остатъчен) списък)  
→ рекурсивен алгоритъм
- Итеративни структури:  
(списък: последователност от свързани клетки)  
→ итеративен алгоритъм

# ДЪЛЖИНА НА СПИСЪК: РЕКУРСИВЕН - ИТЕРАТИВЕН

```
public int length () {  
    if (rest == null)  
        return 1;  
    else  
        return 1 + rest.length();  
}
```

Указател към елемент на списъка:  
преминава отпред назад

```
public int length1 ()  
    int l = 1;  
    IntList actList = this;  
  
    while (actList.rest != null) {  
        actList = actList.rest;  
        l++;  
    }  
    return l;  
}
```

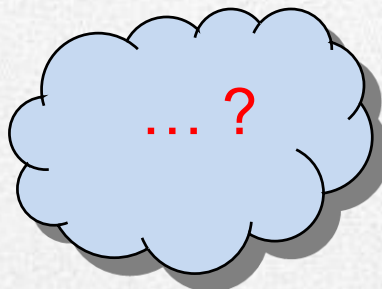
# ДЪЛЖИНА НА ДЪРВО: БРОЙ НА ВЪЗЛИТЕ

## Рекурсивно решение:

```
public int lengthTree () {  
    if (isEmpty())  
        return 0;  
    else  
        return 1  
            + left.lengthTree()  
            + right.lengthTree();  
}
```

Tree.java

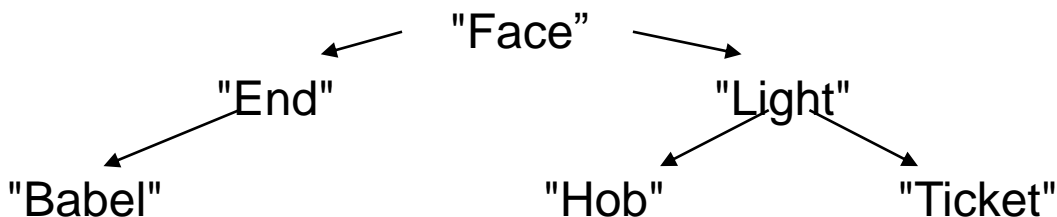
## Итеративно решение:



# СОРТИРАНИ ДЪРВЕТА: ПРИЕМАНЕ НА НОВ ЕЛЕМЕНТ

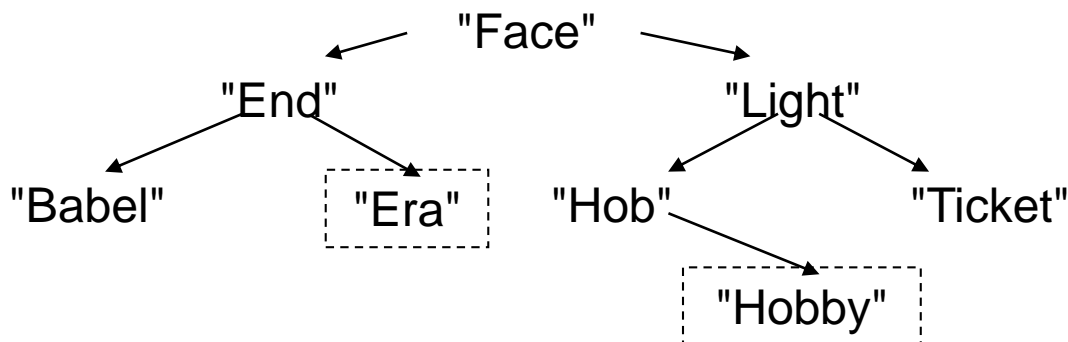
`insertSorted: String x Tree → Tree`

преди:



`insertSorted("Era", ); insertSorted("Hobby", );`

след:

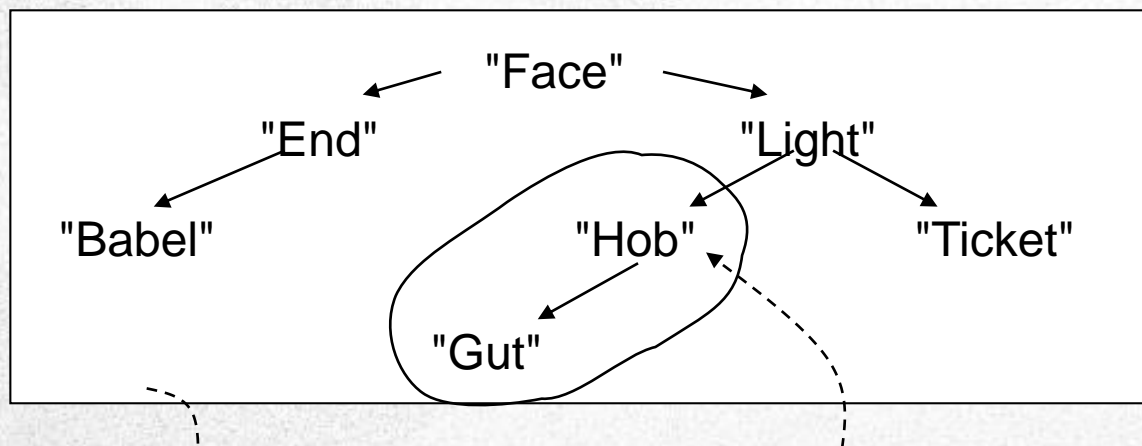



Принцип: - нов елемент долу  
- съществуващите записи непроменени




# СОРТИРАНИ ДЪРВЕТА: ТЪРСЕНЕ НА ЕЛЕМЕНТ

search: Tree x String  $\rightarrow$  Tree



search( , "Hob");

$\rightarrow$  поддърво, което има като съдържание на възела „Hob“

search( , "Stack");  $\rightarrow$  празно дърво

Метод на деление (двоично търсене)  
комплексност:  $O(\log n)$



# ДЪРВЕТА В JAVA: ПРЕДСТАВЯНЕ НА ДАННИ И КОНСТРУКТОРИ

```
public class Tree {  
    String cont;  
    Tree left, right;  
  
    public Tree () {  
        cont = null;  
        left = null;  
        right = null;  
    }  
    ...  
}
```

Tree.java

## Вариант за реализация:

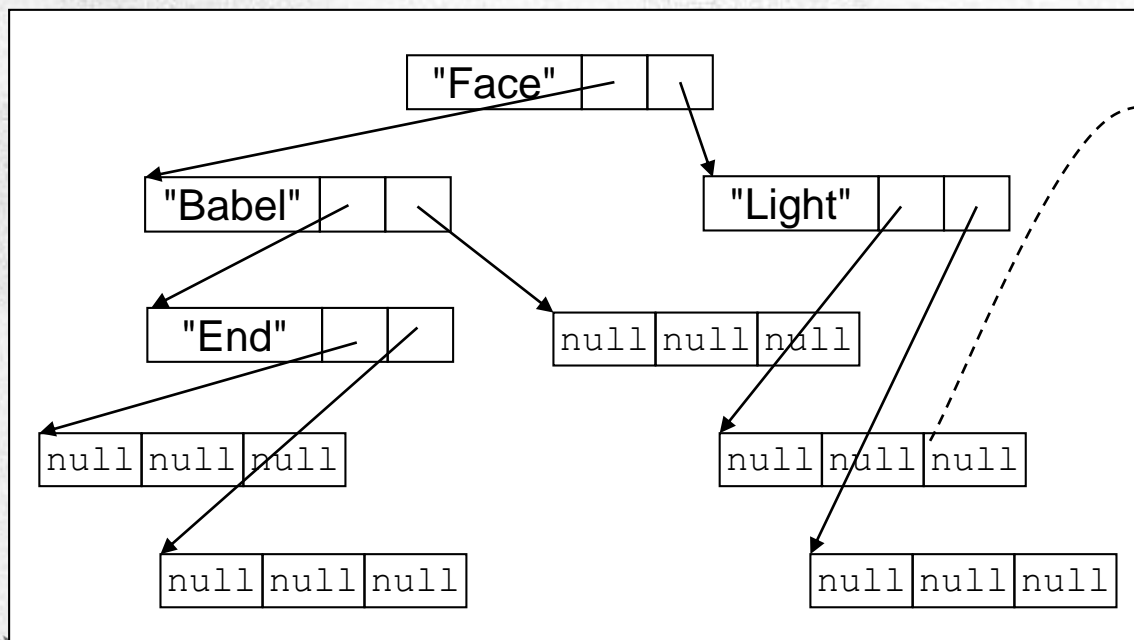
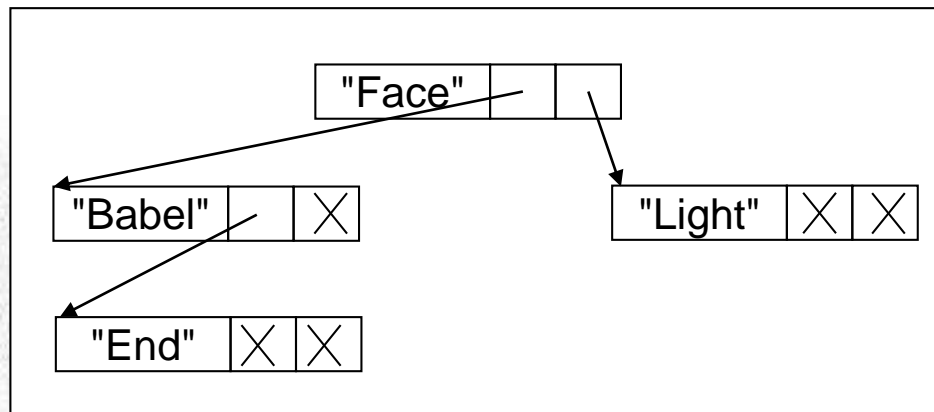
празно дърво като възел със съдържание =  
null

null	null	null
------	------	------

(не задължително

- но удобно
- и ... ИЗИСКВАЩО МНОГО ПАМЕТ)

# ПРАЗНИ ДЪРВЕТА: КАТО NULL-ОБЈЕКТ (X) ИЛИ СПЕЦИАЛНА КЛЕТКА ПАМЕТ



За всяко празно дърво  
Tree(x):

2 допълнителни полета за  
празни възли  
(в примера: 9 вместо 4  
възли)

# ПРИЕМАНЕ НОВ ЕЛЕМЕНТ: РЕАЛИЗАЦИЯ

```
public void insertSorted(String s) {
```

```
    if (isEmpty()) {  
        cont = s  
        left = new Tree();  
        right = new Tree();  
    }
```

ЛИСТ

```
    else if (s.compareTo(cont) == 0);
```

s вече се съдържа

```
    else if (s.compareTo(cont) < 0)  
        left.insertSorted(s);  
    else  
        right.insertSorted(s);
```

лексикографски  
по-малък  
→ въведи в ляво

```
}
```

Tree.java

isEmpty() → 

null	null	null
------	------	------

 → удобно разширение на дървото  
→ но: едно ново въвеждане изисква два нови записа

# ТЪРСЕНЕ НА ЕЛЕМЕНТ: РЕАЛИЗАЦИЯ

Tree.java

```
public Tree search (String s) {  
  
    if (isEmpty())  
        return null;  
  
    else if (s.compareTo(inhalt) == 0)  
        return this;  
  
    else if (s.compareTo(inhalt) < 0)  
        return links.search(s);  
    else  
        return rechts.search(s);  
}
```

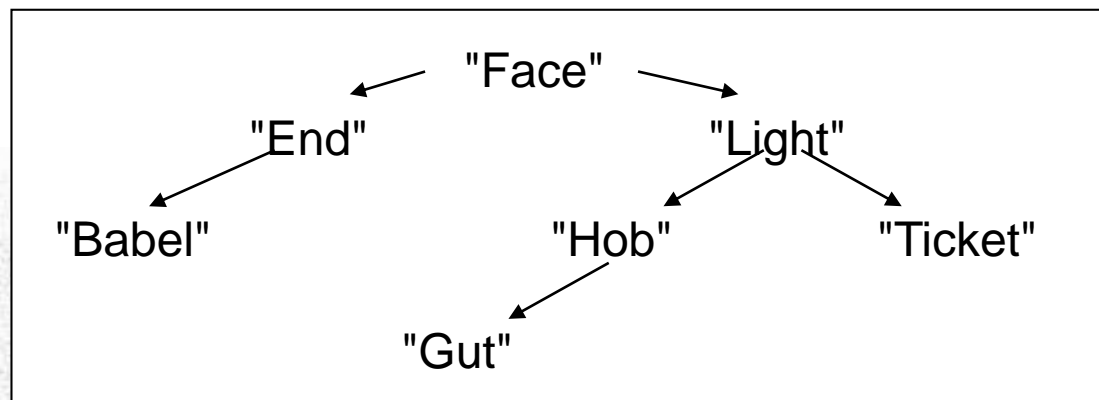
не намерен

намерен → резултат: поддърво

лексикографски по-малък → търси ляво

# ДЪРВЕТА: СТРАТЕГИИ ЗА ЛИНЕАРИЗИРАНЕ

2-размерно  
представяне:



линейно  
представяне:

**Последователност на извеждането:** кога коренът?

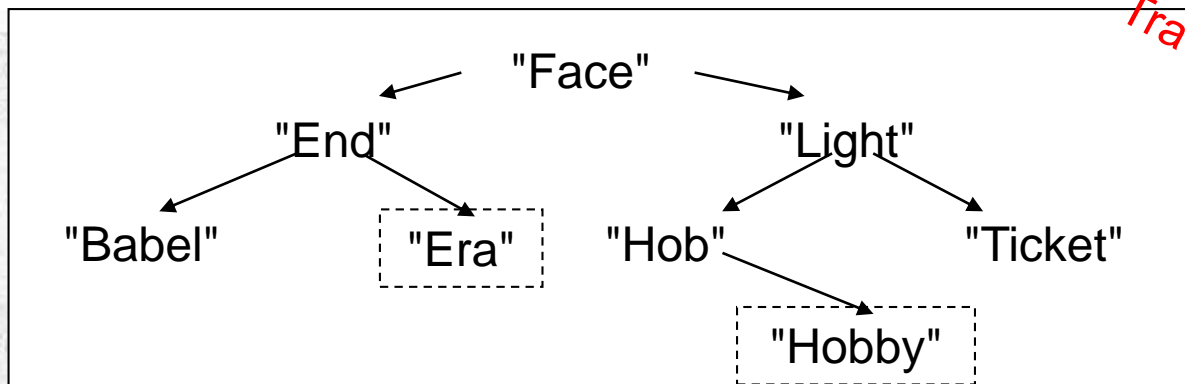
- inorder:
  1. ляво поддърво
  2. корен
  3. дясно поддърво
- preorder:
  1. корен
  2. ляво поддърво
  3. дясно поддърво
- postorder:
  1. ляво поддърво
  2. дясно поддърво
  3. корен

Каква стратегия  
сортирането?

Обработка  
синтактични  
структури



# СТРАТЕГИИ ЗА ЛИНЕАРИЗИРАНЕ: ПРИМЕР



Inorder:

Babel End Era Face Hob Hobby Licht Ticket

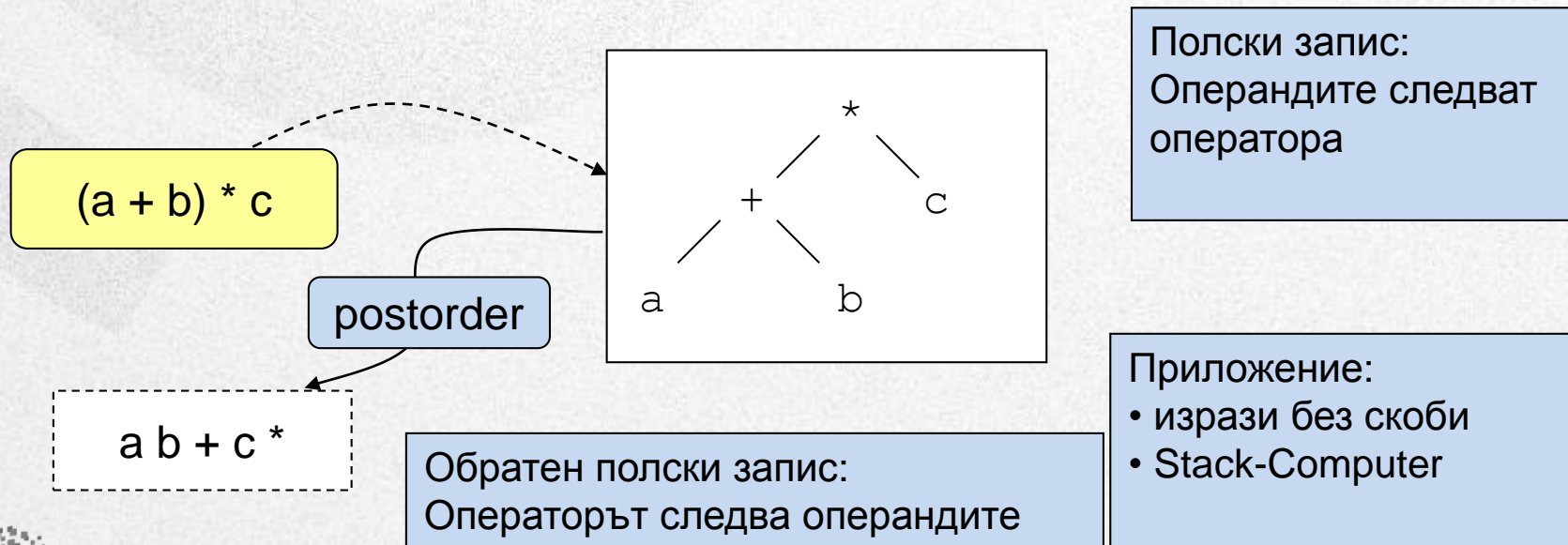
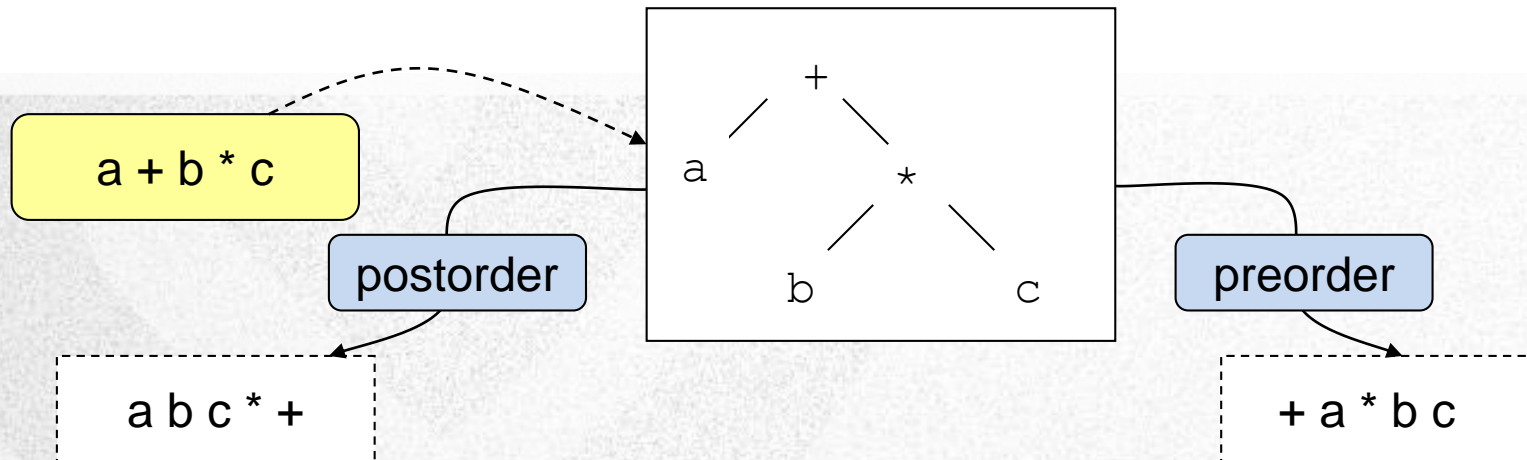
Preorder:

Face End Babel Erde Licht Hob Hobby Ticket

Postorder:

Babel Era End Hobby Hob Ticket Licht Face

# СМИСЪЛ НА 'POSTORDER' И 'PREORDER'



# СТРАТЕГИИ НА ИЗВЕЖДАНЕ: РЕАЛИЗАЦИЯ

```
public static void inorder(Tree b) {  
    if (!b.isEmpty()) {  
        inorder (b.left());  
        System.out.print(b.value() + " ");  
        inorder (b.right());  
    }  
}
```

Traverse.java

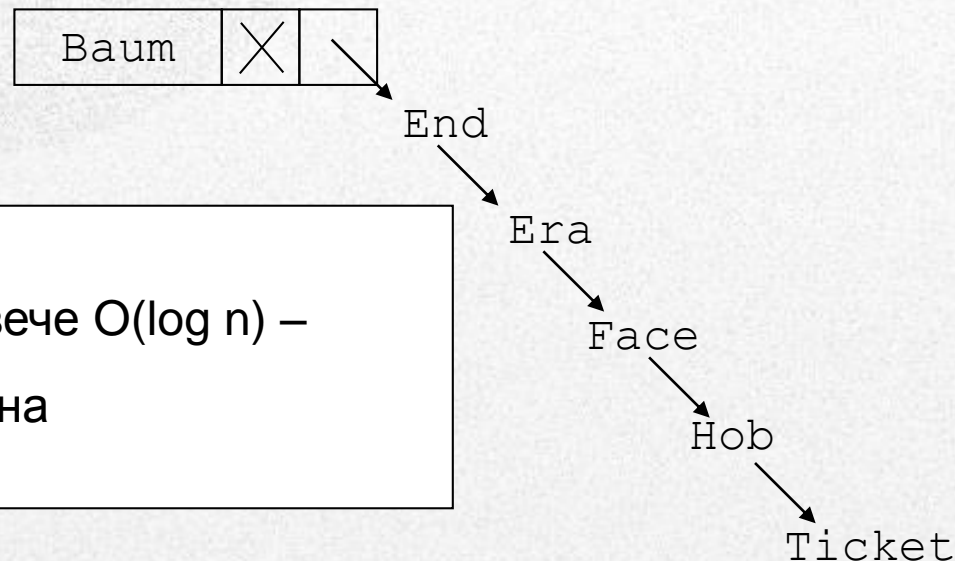
## Inorder:

1. Извеждаме ляво поддърво
2. Извеждаме корен
3. Извеждаме дясно поддърво

'static' подходящ?  
алтернатива?

# НЕБАЛАНСИРАНО ДЪРВО

```
t.insertSorted("Babel");  
t.insertSorted("End");  
t.insertSorted("Era");  
t.insertSorted("Face");  
t.insertSorted("Hob");  
t.insertSorted("Ticket");
```



**Търсене:** не повече  $O(\log n)$  –  
по-скоро линейна

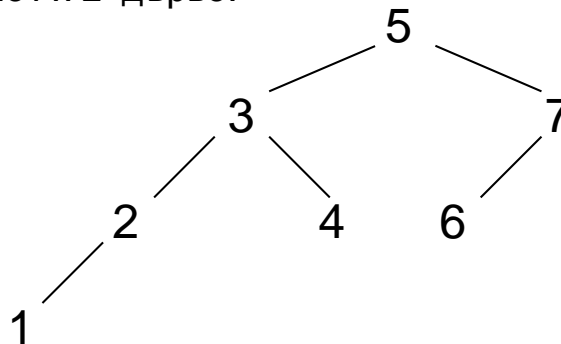


# AVL-ДЪРВЕТА: БАЛАНСИРАНИ ДЪРВЕТА

AVL-дърво:

За всеки възел, височините на двете  
поддървета (ляво и дясно) се  
различават най-много с 1

сортирано AVL-дърво:



За възел 5:

Височина(поддърво(3)) = 3

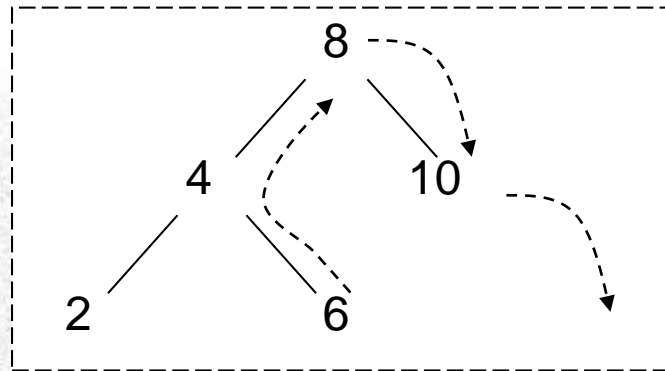
Височина(поддърво(7)) = 2

За възел 7:

...

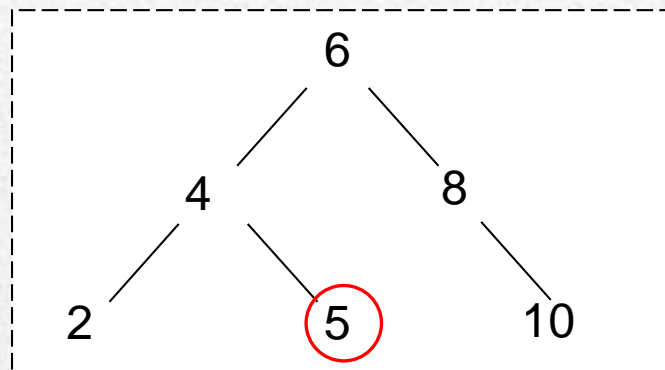


# ВЪВЕЖДАНЕ НА НОВ ВЪЗЕЛ В AVL-ДЪРВО



Въвеждане в дясната страна **безпроблемно**: 9 или 11

**Проблемно** : 1, 5, 7  
дървото трябва да бъде реорганизирано  
(нов корен и с това леви/десни поддървета)



БЛАГОДАРЯ ЗА ВНИМАНИЕТО!

КРАЙ “ДЪРВЕТА”

