

# **Структури от данни и въведение в алгоритмите**

**АМИР РУБИНЩЕЙН**

**Февруари 2012**

# **Структури от данни и въведение в алгоритмите**

**автор: АМИР РУБИНЩЕЙН  
докторант по Компютърни науки в Телавивския Университет,  
гр. Тел Авив, Израел, октомври, 2014 г.**

**преводач: ХРИСТОФОР НИКОЛОВ  
курсист в Софтуерната Академия на Телерик,  
гр. София, България, октомври, 2014 г.**

**създаване: февруари, 2012 г.      превод: октомври, 2014 г.**

**Сборник учебни материали по курса „Структури от данни М“ № 61145**

**Катедра по Софтуерно инженерство и информационни технологии  
Колеж „Орт Брауда“, Израел**

# Съдържание

<b>0. Увод</b>	стр. 4
<b>1. Въведение, сложност на алгоритмите</b>	стр. 7
<b>2. Основни структури от данни</b>	стр. 26
<b>3. Рекурсия и техниката „Разделяй, владей, присъединявай!“</b>	стр. 44
<b>4. Бързо сортиране (Quick Sort) и правилност на алгоритъм</b>	стр. 58
<b>5. Купчина и приоритетна опашка</b>	стр. 73
<b>6. Долна граница на сортирането и линейни сортирания</b>	стр. 91
<b>7. Задачата с избора</b>	стр. 105
<b>8. Бинарни дървета за търсене</b>	стр. 116
<b>9. AVL дървета</b>	стр. 133
<b>10. Разширяване на структури от данни и дървета от степени</b>	стр. 149
<b>11. Хаотични (хеш) таблици</b>	стр. 162
<b>12. Символни низове</b>	стр. 180

# Структури от данни и въведение в алгоритмите

## Data Structures And Introduction To Algorithms

### Увод

#### Извинявайте, тук „Структури от данни и въведение в алгоритмите“ ли е?

Този сборник от слайдове е предназначен за преподаване и изучаване на първи академичен курс по Структури от данни и алгоритми в специалностите по компютърни науки, софтуерно инженерство, инженерство на информационните системи и други подобни.

В различните университети и колежи се изучава успореден курс с примерните имена „Структури от данни“ и „Алгоритми“.

Въпреки това в повечето случаи тези курсове съдържат както глави от „Структури от данни“, така и от „Алгоритми“. На практика тези две теми са до толкова тясно свързани една с друга, че почти е невъзможно да се преподава едната без другата.

Следователно името, което ще фигурира в уводните страници на тези презентации, ще бъде:

### **„Структури от данни и въведение в алгоритмите“**

#### За какво ще се говори в курса?

Неформално:

**Структура от данни** – начин за организиране на данни с цел ефективно изпълнение на различни задачи с тяхна помощ

**Алгоритъм** – начин за решаване на (изчислителна) задача посредством обработка на данни

Пример от живота: дневник



Какви видове дневници ( $\rightarrow$  структури от данни) съществуват?  
Какви задачи ( $\rightarrow$  алгоритми) е лесно да се изпълнят чрез всеки вид?

---

## Какво ще учим в курса?

- Няколко **конкретни** примера за основни структури от данни и алгоритми
    - Структури от данни: масив, списък, купчина, дърво за търсене, хеш таблица, trie, ...
    - Алгоритми: търсения, сортирания (чрез сливане, бързо, чрез купчина, Радикс, ...), Select, ...
    - Сложни комбинации от гореспоменатите
    - Разбира се, съществуват и много повече такива, които няма да изучаваме (но трябва от някъде да започнем, а тези, които ще изучаваме са добра основа за продължение)
  - **Общи фактори, които трябва да се вземат предвид** при планирането на ефективни структури от данни и алгоритми в съответствие с изискванията на задачата
    - Налага се да дадем формално определение за понятието „ефективност“ (това е математическата част от курса)
    - Ще се запознаем с няколко общи подхода / гледни точки при планирането на структури от данни и алгоритми:  
Подходът „Разделяй, владей, присъединявай!“, долна граница на задачата, разширяване на структури от данни, динамично оптимиране, ...
- 

## Структура на презентациите

Всяка презентация е структурирана по следния начин:

- Основната част е представянето на новата тема под формата на лекция.
  - В края на всяка презентация се съдържат „**Въпроси за преговор**“. Това са сравнително лесни въпроси, чиято цел е да позволи на учещия да се увери, че е разбрал материала, и преговор върху материала на базисно ниво на разбиране.
  - След това следва препратка към препоръчителни упражнения от учебника, придружаващ презентациите (подробности относно учебника следват малко по-долу). Повечето упражнения от учебника се намират под определена под-точка в релевантната глава и така те позволяват упражняването на под-теми от съответната глава. Степента им на трудност е различна.
  - На края се съдържат няколко допълнителни упражнения с решения, по принцип със средна и висока степен на трудност.
- 

## Изисквания за предварителни знания

- Изучаването на поне един курс, включващ ползването на някакъв програмен език като C, C++, Java или друг език.

- Познаване на някои основни алгоритми като: двоично търсене, сортиране чрез сливане и няколко прости сортирания (чрез балончета, чрез вкарване, чрез избиране и други подобни).
  - Базисна способност за класифициране на алгоритми по класове времева сложност (логаритмични, линейни, квадратни, експоненциални алгоритми и т. н.)
  - Познаване на основни структури от данни като масив, свързан списък, опашка и стек.
- 

## Учебникът

Основният учебник, върху който ще се базират большинството (но не всички) презентации е:

Въведение в алгоритмите, глави 1-14, второ издание, 2008 г.  
(издателство на Университета Ха-Птуха, Израел)

Това е превод на първите 14 глави от книгата:

*Introduction to Algorithms*, 2<sup>nd</sup> ed. (MIT Press, 2000).  
Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein

Обърнете внимание: Книгата има и старо издание, но презентациите и номерата на упражненията съответстват на второто издание.

---

## Темите

Презентация №	Темите	
1	Увод и математически основи	Въведение, сложност на алгоритмите
2		Основни структури от данни
3		Рекурсия и рекурсивни алгоритми
4	Сортирането и изборът	Бързо сортиране (Quick Sort), доказване правилността на алгоритми
5		Сортиране чрез купчина и приоритетна опашка
6		Долна граница на сортирането и линейни сортирания
7		Проблемът с избора (намирането на i-тия елемент)
8		Бинарни дървета за търсене
9	Структури от данни за речник	AVL дървета
10		Разширяване на структури от данни и дървета от степени
11		Хаотични (хеш) таблици
12		Символни низове – trie и алгоритъмът LCS

## Тема 1

# Въведение, сложност на алгоритмите Introduction, Complexity of Algorithms

### В програмата:

Глави 1-3 от учебника

- Ще се запознаем с няколко основни понятия от областта на алгоритмите
  - Ще се срещнем отново с няколко познати алгоритма
  - Ще се научим да анализираме сложност (основно по време) на алгоритмите и за тази цел ще си служим с гръцките символи  $\Omega$ ,  $\omega$  (омега),  $\Theta$ ,  $\theta$  (тета).
- 

### Въведение и основни понятия

Изчислителна задача (computational problem)

Пасване на входящи данни (input) към изходящи данни (output).

Примери:	<u>Вход</u>	→	<u>Изход</u>
	цели числа		за всяко число абсолютната му стойност
	масиви от реални числа		масивът, когато е сортиран
	думи на английски език		за всяка дума група от значенията ѝ

### Алгоритъм (algorithm)

Редица изчисления, която за всеки валиден вход произвежда съответен изход.

Алгоритъмът на практика е решение на изчислителна задача.

Какви са позволените изчисления? Зависи от контекста.

Съществуват различни нива на резолюция:

- Побитови операции (bits)
  - Събиране, изваждане, умножение, деление, насочване към променлива (четене / писане), сравнение, ... **От наша гледна точка това ще бъдат „основните действия“.**
  - Сортиране на масив, ...
- 

### Въведение и основни понятия

#### Задачата за ябълковия пай

Вход: 2,5 чаши брашно, 5 ябълки, 3 яйца от свободно отглеждани кокошки, 0,5 чаша захар, ...

Изход: ябълков пай.

#### Алгоритъм, който решава задачата:

Твърде високо ниво на разбивка:

- Отвори чекмеджето!
- Извади пакета с брашно от чекмеджето!
- Отвори пакета с брашно!
- ...

Твърде ниско ниво на разбивка:

- Приготви от продуктите ябълков пай!
- 

## Въведение и основни понятия

### Осъществяване на алгоритъм (implementation)

Компютърна програма, точно представяне на алгоритъм на някакъв програмен език.

### Изпълнение на алгоритъм (execution)

Даване на заповед на **конкретни компютри** да изпълнят инструкциите, съдържащи се в осъществяването на алгоритъма.

### Кои алгоритми вече познавате? Какви задачи решават те?

---

## Търсене в сортиран масив

### Пример: Задачата с търсенето в сортиран масив

Определение на задачата:

Вход: сортиран масив A с n на брой елемента и някаква допълнителна стойност key.

Изход: индекс от A, чийто ключ е key, ако съществува такъв.

Тривиален алгоритъм, решаващ задачата – Линейно търсене:

```
Linear-Search1(A, n, key)
1. for i  $\leftarrow$  1 to n
2.   if A[i] = key
3.     return i
4.   return Nil
```

**Псевдо-кодът** (pseudo-code) е „език“ за описване на алгоритми, който позволява да се концентрираме върху структурата на алгоритъма, без да се замисляме за особеностите на всеки програмен език.

Псевдо-код не може да се изпълни от компютър, но трябва да е написан така, че лесно да се преведе на какъвто и да е програмен език (уточнение на стр. 16-17 от учебника).

Забележка: По същите причини първият индекс на масив в нашия курс обикновено ще бъде 1 вместо 0.

---

## Търсене в сортиран масив

### Подобрение 1:

Linear-Search2( $A, n, key$ )

1.  $i \leftarrow 1$
2. **while**  $i \leq n$  **and**  $A[i] \leq key$
3.     **if**  $A[i] = key$
4.         **return**  $i$
5.      $i \leftarrow i+1$
6. **return** Nil

#### Подобрение 2 – Бинарно (двоично) търсене:

Binary-Search( $A, n, key$ )

1.  $p \leftarrow 1, q \leftarrow n$
2. **while**  $p \leq q$                               ► sub-array to search not empty
3.      $mid \leftarrow \lfloor (p + q) / 2 \rfloor$
4.      $k \leftarrow A[mid]$
5.     **if**  $key = k$
6.         **return**  $mid$                               ► found!
7.     **if**  $key < k$
8.          $q \leftarrow mid - 1$
9.     **else**          $p \leftarrow mid + 1$
10. **return** Nil                                  ► not found

### Индекси за оценка ефективността на алгоритъм

Оценка на алгоритми и сравнение помежду им може да се извърши по различни критерии:

- До колко алгоритъмът е лесен за разбиране?
- До колко алгоритъмът е лесен за осъществяване?
- До колко е лесно да се докаже верността (правилността) на алгоритъма?
- Дължина на осъществяването
- Елегантност
- Сложност от гледна точка времето за изпълнение – количеството действия, които извършва алгоритъмът
- Сложност от гледна точка на паметта – нужното количество ресурси компютърна памет
- И още

Ние ще се фокусираме основно върху сложността от гледна точка времето за изпълнение и сложността от гледна точка на паметта, но винаги ще се стараем да подобряваме своя алгоритъм по всички критерии (например, винаги ще предпочетем по-„елегантния“ алгоритъм, ако при останалите критерии няма разлика ...).

### Анализ на времевата сложност

Когато анализираме сложността от гледна точка времето за изпълнение на един алгоритъм, ние на практика броим количеството основни действия, които той извършва, в зависимост от размера на неговите входни данни.

- Защо броим действия, а не време?

За да пренебрегнем характеристиките на конкретния компютър (например, скоростта на процесора, броя времеви цикли за основни действия)

Ще се съгласим, че всяко основно действие (събиране, умножение, сравнение, копиране и т. н.) изисква постоянен брой времеви цикли, и от тук нататък ще пренебрегнем тези константи (или ще вземем горна граница за всички).

- Защо в зависимост от размера на входните данни?

Понеже, най общо казано, колкото повече расте размера на входните данни, толкова повече се повишава и времето за изпълнение.

Например, сортиране на масив с дължина 100 елемента най-вероятно ще изисква повече действия от сортирането на масив с дължина 10 елемента.

Понятието размер на входните данни зависи от характера на входните данни на задачата. Например:

- масив: броя на клетките в масива
- цяло число: количеството битове, чрез което се представя числото
- матрица: броя на редовете и/или броя на колоните, броя на клетките
- граф (диаграма): броя на кръстовищата и/или броя на дъгите

## Обозначението $\Theta$ (тета)

Колко действия ще бъдат извършени от масив с размер  $n$ ?

Зависи къде (ако въобще) се намира търсеният елемент.

Linear-Search1( $A, n, key$ )

1. **for**  $i \leftarrow 1$  to  $n$
2.     **if**  $A[i] = key$
3.         **return**  $i$
4. **return** Nil

Ако  $key$  въобще го няма:

Ред	Брой пъти	Количество действия
1	$n + 1$	$c_1$
2	$n$	$c_2$
3	0	$c_3$
4	1	$c_4$

$$T(n) = c_1(n + 1) + c_2n + c_4 = (c_1 + c_2)n + (c_1 + c_4)$$

Получихме израз от типа  $an + b$ , това означава линейна функция по  $n$ .

Тоест, темпер на нарастване времето за изпълнение е линейно по  $n$ . Например, когато увеличим  $n$  по 2, времето за изпълнение ще нарасне приблизително два пъти (това е толкова по-точно, колкото по-голямо е  $n$ ).

Обозначаваме го с помощта на гръцката буква тета:  $T(n) = \Theta(n)$ .

- Ако  $key$  се намира в  $A[1]$ , общо ще се извършат постоянен брой действия (на зависи от  $n$ ). Обозначение  $\Theta(1)$ .

## Обозначението $\Theta$ (тета) – класове сложност

Използването на  $\Theta$  (тета) е много удобно, когато сравняваме поведението (изпълнението) на различни алгоритми – така могат да се сравняват темповете на нарастващо или порядъците на тяхното време за изпълнение.

Количество действия на различни алгоритми в зависимост от размера на входните данни:

	$T_1(n) = 140$	$T_2(n) = 50n + 30$	$T_3(n) = n^2$	$T_4(n) = n^2 + 2n$	$T_5(n) = 2^n$
$n = 10$	140	530	100	120	1024
$n = 20$	140	1030	400	440	1048576
$n = 100$	140	5030	10000	10200	$1.26 * 10^{30}$
complexity	$\Theta(1)$	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(2^n)$

\* Линк към стойността на Big Bang в Уикипедия: [http://en.wikipedia.org/wiki/Big\\_Bang](http://en.wikipedia.org/wiki/Big_Bang)

Изводи (за сега неформални):

- Можем да пренебрегнем константите
- Можем да пренебрегнем елементите от по-малък поръдък

## Зависимост на времето за изпълнение от входните данни при определена дължина

Както видяхме, понякога различни входни данни (вход) с еднакъв размер имат различно време за изпълнение.

Тоест, количеството действия понякога зависи от самите входни данни (а не само от размера им).

Ще обозначим с  $K(n)$  групата от всички входни данни с размер  $n$  на алгоритъма  $Alg$ .

За всеки вход  $I \in K(n)$ ,  $T(I)$  ще обозначава времето за изпълнение на алгоритъма, изпълняващ се върху този вход.

- Времето за изпълнение на алгоритъма  $Alg$  в най-лошия случай (worst-case) е:

$T_{worst}(n) = \max \{ T(I) \mid I \in K(n) \}$  (най-дългото време за изпълнение на  $Alg$  върху вход с дължина  $n$ )

- Времето за изпълнение на алгоритъма  $Alg$  в най-добрия случай (best-case) е:

$T_{best}(n) = \min \{ T(I) \mid I \in K(n) \}$  (най-краткото време за изпълнение на  $Alg$  върху вход с дължина  $n$ )

- Средното време за изпълнение на алгоритъма  $Alg$  (average):

$T_{average}(n) = (\sum_{I \in K(n)} T(I)) / |K(n)|$  (Средното време за изпълнение на алгоритъма  $Alg$  върху всички входове с дължина  $n$ , нарича се също очаквана стойност на времето за изпълнение)

Забележка: Тук предположихме, че входовете се срещат с равни вероятности.

По принцип повече ще се интересуваме от времето за изпълнение в най-лошия случай.

---

### Анализ на Бинарното търсене (Binary Search)

- Най-лошия случай: когато key въобще не се намира в A.

В края на първата итерация размерът на под-масива, в който търсим, намалява най-много до  $n/2$ .

В края на втората итерация: най-много до  $n/4$ .

В края на k-тата итерация: най-много до  $n/2^k$ .

Колко итерации? Ще търсим k в края на последната итерация:

$$\begin{aligned}\frac{n}{2^k} &< 1 \\ \Rightarrow n &< 2^k \\ \Rightarrow \log n &< k \\ \Rightarrow k &= \lfloor \log n \rfloor + 1\end{aligned}$$

(k е най-малкото цяло число, което реално е по-голямо от  $\log n$ )

При всяка итерация се извършват постоянен брой действия (ще отбележим с „c“), и затова:  $T(n) = c \cdot \lfloor \log n \rfloor + 1 = \Theta(\log n)$

- Най-добрия случай: ако key се намира по средата на A, в крайна сметка ще бъдат изпълнени постоянен брой действия, тоест  $\Theta(1)$ .

Забележка: Това е оптимален алгоритъм от гледна точка на времевата сложност за търсене в сортиран масив.

---

### Няколко полезни правила за анализ на сложност

Алгоритмите със структура от типа (константа  $k > 0$ ) са линейни:

$\Theta(n)$	<ol style="list-style-type: none"><li>1. <math>i \leftarrow 1</math></li><li>2. <b>while</b> <math>i &lt; n</math></li><li>3.        <math>i \leftarrow i + 1</math></li></ol>	<ol style="list-style-type: none"><li>1. <math>i \leftarrow 1</math></li><li>2. <b>while</b> <math>i &lt; n</math></li><li>3.        <math>i \leftarrow i + k</math></li></ol>	<ol style="list-style-type: none"><li>1. <math>i \leftarrow n</math></li><li>2. <b>while</b> <math>i &gt; 1</math></li><li>3.        <math>i \leftarrow i - k</math></li></ol>
-------------	--	--	--

Алгоритмите със структура от типа (константа  $k > 1$ ) са логаритмични:

$\Theta(\log n)$	<ol style="list-style-type: none"><li>1. <math>i \leftarrow 1</math></li><li>2. <b>while</b> <math>i &lt; n</math></li><li>3.        <math>i \leftarrow i * 2</math></li></ol>	<ol style="list-style-type: none"><li>1. <math>i \leftarrow 1</math></li><li>2. <b>while</b> <math>i &lt; n</math></li><li>3.        <math>i \leftarrow i * k</math></li></ol>	<ol style="list-style-type: none"><li>1. <math>i \leftarrow n</math></li><li>2. <b>while</b> <math>i &gt; 1</math></li><li>3.        <math>i \leftarrow i / k</math></li></ol>
------------------	--	--	--

А какво ще кажете за:

$\Theta(\log \log n)$

1.  $i \leftarrow 2$
2. **while**  $i < n$
3.      $i \leftarrow i * i$

$\Theta(\log \log n)$

1.  $i \leftarrow 2$
2. **while**  $i < n$
3.      $i \leftarrow i^k$

$\Theta(\log \log n)$

1.  $i \leftarrow n$
2. **while**  $i > 2$
3.      $i \leftarrow \sqrt[k]{i}$

Докажете:  $\Theta(\log \log n)$  !

## Сортиране чрез вкарване (Insertion Sort)

### Пример: задачата за сортирането

#### Определение на задачата:

Вход: списък (например масив) A от елементи ( $a_1, a_2, \dots, a_n$ ), върху които е зададен определен ред.

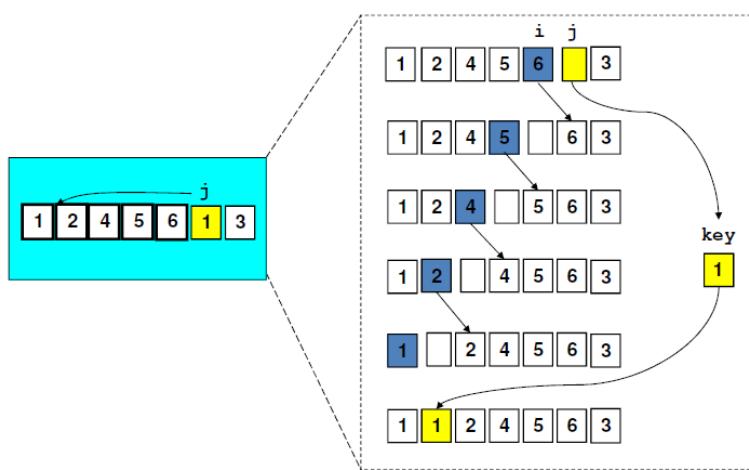
Изход: пермутация (permutation) на A: ( $a'_1, a'_2, \dots, a'_n$ ), така че са подредени от по-малко към по-голямо:  $a'_1 \leq a'_2 \leq \dots \leq a'_n$

#### Алгоритъм: Сортиране чрез вкарване (Insertion Sort)

```
Insertion-Sort(A, n)
1. for  $j \leftarrow 2$  to  $n$ 
2.     key  $\leftarrow A[j]$ 
3.     ► insert  $A[j]$  into the sorted sequence  $A[1..j-1]$ 
4.      $i \leftarrow j-1$ 
5.     while  $i > 0$  and  $A[i] > key$ 
6.          $A[i+1] \leftarrow A[i]$ 
7.          $i \leftarrow i - 1$ 
8.      $A[i+1] \leftarrow key$ 
```

## Сортиране чрез вкарване (Insertion Sort)

### Демонстрация на една външна итерация на Insertion Sort



## Анализ на Insertion Sort

```

Insertion-Sort( $A, n$ )
1.   for  $j \leftarrow 2$  to  $n$ 
2.      $key \leftarrow A[j]$ 
3.     ▶ insert  $A[j]$  into the sorted sequence  $A[1..j-1]$ 
4.      $i \leftarrow j-1$ 
5.     while  $i > 0$  and  $A[i] > key$ 
6.        $A[i+1] \leftarrow A[i]$ 
7.        $i \leftarrow i - 1$ 
8.      $A[i+1] \leftarrow key$ 

```

- Най-лошият случай е когато масивът е сортиран обратно (от по-голямо към по-малко). В този случай вътрешният цикъл се изпълнява  $j-1$  на брой итерации за всяка стойност на  $j$  между 2 и  $n$ . Външният цикъл се изпълнява  $n-1$  на брой итерации. Да предположим, че всяка вътрешна итерация изисква  $c_1$  действия и всяка външна итерация –  $c_2$  действия (без вътрешните итерации).
$$T(n) = c_1 \sum_{j=2}^n (j-1) + c_2(n-1) = c_1 \frac{n(n-1)}{2} + c_2(n-1) = \Theta(n^2)$$
- Най-добрият случай е когато масивът е сортиран (както обикновено, във възходящ ред). Този път вътрешният цикъл се изпълнява 0 итерации всеки път.
$$T(n) = c_2(n-1) = \Theta(n)$$

## Долни и горни бариери (граници)

Понякога искаме да зададем горна / долна граница на времето за изпълнение на даден алгоритъм.

- Тъй като е трудно да се изчисли порядъка ( $\Theta$ ) на времето за изпълнение, но е лесно да му се постави бариера отгоре / отдолу
- Тъй като ни интересува единствено горна / долна граница на времето за изпълнение (примери?)

Обозначение	Име на обозначението	Значение	Вид граница
$t(n) = O(g(n))$	главно О	$t$ не е асимптотно по-голямо от $g$	горна
$t(n) = \Omega(g(n))$	главна Омега	$t$ не е асимптотно по-малко от $g$	долна
$t(n) = o(g(n))$	малко о	$t$ е асимптотно по-малко от $g$	горна не плътна
$t(n) = \omega(g(n))$	малка омега	$t$ е асимптотно по-голямо от $g$	долна не плътна
$t(n) = \Theta(g(n))$	тета	същият асимптотен темп на нарастване	плътна

Трябва да обърнем внимание, че горни / долни граници могат да се изчисляват както за най-лошия случай, така и за най-добрания, а също и за средното време за изпълнение.

## Долни и горни бариери (граници)

Както казахме, понякога е трудно да се изчисли порядъка на времето за изпълнение и тогава се опитваме да изчислим по отделно горна и добра граница. Има (сложни) алгоритми, чиито горни и добра граници не са сходящи (не клонят към конкретно число), и тогава порядъкът на времето им за изпълнение не е известен.



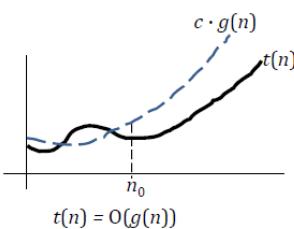
В нашия курс в повечето случаи:

- или ще можем да директно изчислим плътна граница ( $\Theta$ )
- или ще можем да изчислим горна граница и добра граница, които са сходящи (клонят към конкретни числа)

### Горна граница – главно O

Нека  $t(n)$  и  $g(n)$  са две положителни асимптотни функции.

**$t(n) = O(g(n))$ , ако съществуват положителни константи  $c$ ,  $n_0$ , така че за всяко  $n > n_0$ :  $t(n) \leq c \cdot g(n)$**



Например:

$$t(n) = 10n^2 + 30 = O(n^2)$$

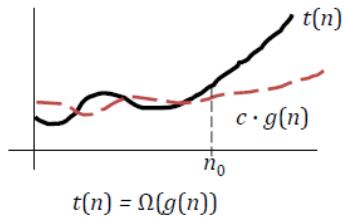
$$t(n) = 10n^2 + 30 = O(n^3)$$

c 20

### Добра граница – главна $\Omega$ (омега)

Нека  $t(n)$  и  $g(n)$  са две положителни асимптотни функции.

**$t(n) = \Omega(g(n))$ , ако съществуват положителни константи  $c$ ,  $n_0$ , така че за всяко  $n > n_0$ :  $t(n) \geq c \cdot g(n)$**



Например:

$$t(n) = 10n^2 + 30 = \Omega(n^2)$$

$$t(n) = 10n^2 + 30 = \Omega(n)$$

### Малко „о“ и малка „омега“

Нека  $t(n)$  и  $g(n)$  са две положителни асимптотни функции.

**$t(n) = o(g(n))$ , ако  $\lim_{n \rightarrow \infty} t(n)/g(n) = 0$**

Например:

$$t(n) = 10n^2 + 30 = o(n^3)$$

$$t(n) = 10n^2 + 30 \neq o(n^2)$$

**$t(n) = \omega(g(n))$ , ако  $\lim_{n \rightarrow \infty} t(n)/g(n) = \infty$**

Например:

$$t(n) = 10n^2 + 30 = \omega(n)$$

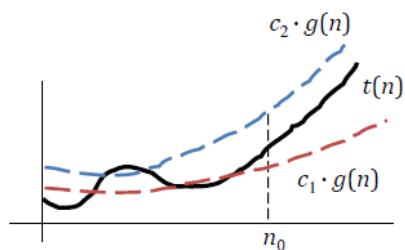
$$t(n) = 10n^2 + 30 \neq \omega(n^2)$$

### Плътна граница – $\Theta$

Сега (най-сетне) ще определим формално  $\Theta$  (тета):

Нека  $t(n)$  и  $g(n)$  са две положителни асимптотни функции.

**$t(n) = \Theta(g(n))$ , ако съществуват положителни константи  $c_1$ ,  $c_2$  и  $n_0$ , така че за всяко  $n > n_0$  да е изпълнено:  $c_1 g(n) \leq t(n) \leq c_2 g(n)$**



- $t(n) = \Theta(g(n))$ , само и единствено ако  $t(n) = O(g(n))$  и също  $t(n) = \Omega(g(n))$ .
- $t(n) = \Theta(g(n))$ , само и единствено ако  $t(n) = O(g(n))$  и също  $t(n) \neq o(g(n))$ .
- $t(n) = \Theta(g(n))$ , само и единствено ако  $t(n) = \Omega(g(n))$  и също  $t(n) \neq \omega(g(n))$ .

### Граници – полезни правила (1)

Докажете: ако  $f_1(n) = O(g_1(n))$  и  $f_2(n) = O(g_2(n))$ , то:

a)  $f_1(n) + f_2(n) = O(\max\{g_1(n), g_2(n)\})$

b)  $f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$

### Граници – полезни правила (2)

Твърдение: Нека, за някаква константа  $k$ , функциите  $f_1(n), \dots, f_k(n)$  са положителни асимптотни функции. Тогава:  $f_1(n) + f_2(n) + \dots + f_k(n) = \Theta(\max\{f_i(n)\})$

Например:  $74\log n + 2n^3 + 15n = \Theta(n^3)$

Доказателство: за достатъчно голямо  $n$ :

$$\max\{f_i(n)\} \leq f_1(n) + f_2(n) + \dots + f_k(n) \leq k(\max\{f_i(n)\})$$

Въпрос: Дали твърдението е вярно, и когато броят на събирамите не е задължително константа, а зависи от  $n$ ?

### Граници – полезни правила (3)

- $\log^k n = o(n^\varepsilon)$  за всяко  $\varepsilon > 0$  (доказателство посредством правилото на Лопитал)

Например:  $\log^2 n = o(n^{0.001})$

- $n^k = o(a^n)$  за  $a > 1$

- $a^n = o(b^n)$  за  $0 < a < b$

- Аритметична редица:

$$\sum_{i=1}^n i = \frac{1}{2}n(n+1) = \Theta(n^2)$$

- Геометрична редица:

$$(x \neq 1) \quad \sum_{i=0}^n x^i = \frac{x^{n+1} - 1}{x - 1} = \Theta(x^n)$$

- Сходяща геометрична редица:

$$(0 < x < 1) \quad \sum_{i=0}^{\infty} x^i = \frac{1}{1-x} = \Theta(1)$$

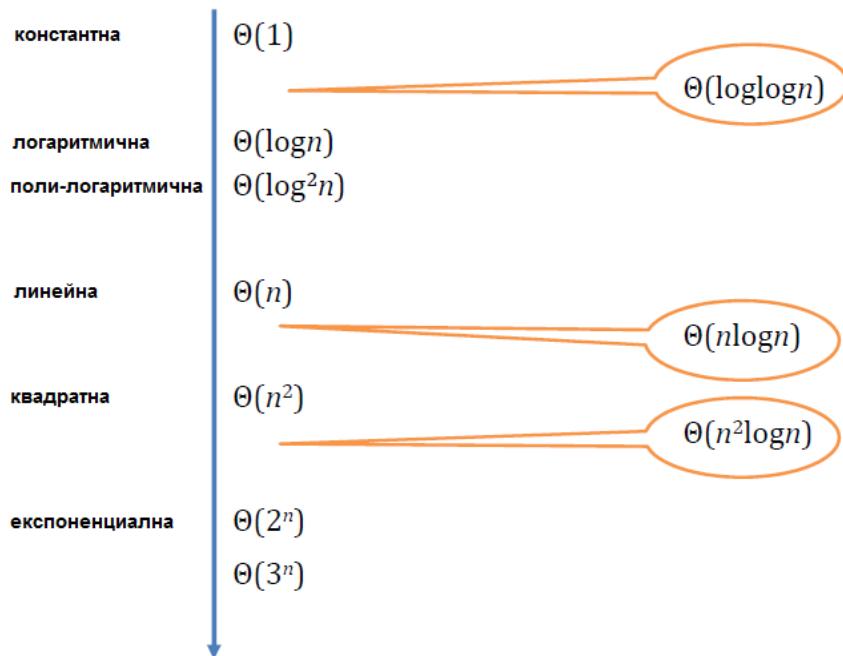
- Хармоничната редица:

$$H_n = \sum_{i=1}^n \frac{1}{i} = \Theta(\log n)$$

- Полином от степен  $d$ :

$$(a_d \neq 0) \quad \sum_{i=0}^d a_i n^i = \Theta(n^d)$$

## Йерархия на степените на сложност



## Йерархия на степените на сложност

Важно твърдение:  $\log(n!) = \Theta(n \log n)$

- $\log(n!) \leq \log(n^n) = n \log n \rightarrow \log(n!) = O(n \log n)$
- $\log(n!) = \log(n \cdot (n-1) \cdot \dots \cdot \lceil \frac{n}{2} \rceil \cdot \dots \cdot 2 \cdot 1) \geq$   
 $\log(n \cdot (n-1) \cdot \dots \cdot \lceil \frac{n}{2} \rceil) \geq \log(\lceil \frac{n}{2} \rceil \cdot \dots \cdot \lceil \frac{n}{2} \rceil) =$   
 $\log(\lceil \frac{n}{2} \rceil^{\lfloor \frac{n}{2} \rfloor + 1}) \geq \log(\frac{n}{2}^{\frac{n}{2}}) = \frac{n}{2} \log(\frac{n}{2}) =$   
 $\frac{n}{2} (\log n - 1) \geq \frac{n}{2} (\log n - \frac{1}{2} \log n) = \frac{1}{4} n \log n$   
 $\rightarrow \log(n!) = \Omega(n \log n)$

## Определяне времевата сложност на алгоритми

Ще анализираме сложността от гледна точка времето за изпълнение на следния алгоритъм като функция на  $n$ :

**Кой случай ще анализираме? Най-лошия? Най-добрия?**

**Този път няма разлика.**

**Във всеки скучай се извършват същия брой действия.**

```
Alg( $A, n$ )
1.    $s \leftarrow 0$ 
2.   for  $i \leftarrow 1$  to  $n$ 
3.        $j \leftarrow 1$ 
4.       while  $j \leq n$ 
5.            $j \leftarrow j + i$ 
6.            $s \leftarrow s + A[i]$ 
```

Има два вложени цикъла (един в друг).

Груб анализ: Външен цикъл, който изпълнява  $n$  на брой итерации.

Вътрешен цикъл, който изпълнява най-много  $n$  на брой итерации всеки път:

$$T(n) \leq c_1 + n(c_2 + n \cdot c_3) = O(n^2)$$

По- внимателен анализ:

Броят на вътрешните итерации зависи от стойността на  $i$ , а тя е равна на  $\lceil n/i \rceil$ .

$$T(n) = \Theta\left(\sum_{i=1}^n \lceil n/i \rceil\right) = \Theta(n \cdot (1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n})) = \Theta(n \cdot H_n) = \Theta(n \log n)$$

## Определяне времевата сложност на алгоритми

Ще анализираме сложността от гледна точка времето за изпълнение на следния алгоритъм, който приема сортиран масив  $A$ , размерът му е  $n$ , а ключът му за търсене е  $key$ :

```
Alg-Search( $A, n, key$ )
1.   for  $i \leftarrow 1$  to  $n$ 
2.        $found \leftarrow \text{Binary-Search}(A, i, key)$ 
3.       if  $found \neq \text{Nil}$ 
4.            $\text{print}(key \text{ is located at index } found)$ 
```

Най-лошият случай – нито едно търсене не успява

Груб анализ: Извиквания на бинарно търсене в под-масив с размер  $\leq n$ . Затова всяко търсене се извършва за време  $O(\log n)$ :

$$T(n) = n \cdot O(\log n) = O(n \log n)$$

По- внимателен анализ:

Сложността от гледна точка времето за изпълнение на търсенето в под-масив с дължина  $i$  е  $\Theta(\log i)$ :

$$T(n) = \sum_{i=1}^n \Theta(\log i) = \Theta(\log(n!)) = \Theta(n \log n)$$

Най-добрият случай – всички търсения успяват веднага

$n$  извиквания на бинарно търсене, което извършва  $\Theta(1)$  действия. Общо  $\Theta(n)$ .

Как изглежда резултатът (изходните данни), който дава най-добрият случай?

## Въпроси за преговор

- 1) Разбирате ли разликата между време за изпълнение в най-лошия / най-добрая / средния случай и горна / долнна / плътна граница?
- 2) Какво представлява сложността от гледна точка времето за изпълнение като зависимост от  $n$  за следните части програмен код? Обърнете внимание, че двете части извършват точно едно и също нещо.

1.  $n \leftarrow 100$   
2. **for**  $i \leftarrow 1$  to  $n$   
3. ...

1.  $n \leftarrow 100$   
2. **for**  $i \leftarrow 1$  to 100  
3. ...

- 3) Какво е значението на равенството  $n = o(m)$ ? Дайте пример за  $n$  и  $m$ , които го изпълняват!
- 4) Погледнете отново слайд 11 на име „Обозначението  $\Theta$  – класове сложност“! Да предположим, че разполагаме с компютър, изпълняващ един милиард основни действия в секунда. Колко време ще е нужно за изпълнението на алгоритмите, чито времена за изпълнение се описват чрез различните  $T_i$  за  $n = 10, 20, 100$ ?

## Отговори на въпросите за преговор

- 1) Например,  $O$  е горна граница, която може да се изчисли както за най-лошия случай, така и за най-добрая случай, а също и за средното време за изпълнение.
- 2) Сложността от гледна точка времето за изпълнение на лявата част код е  $\Theta(n)$ , а на дясната част е  $\Theta(1)$ . На практика задаваният въпрос тук е, какъв е темпът на нарастване на времето за изпълнение в зависимост от  $n$ . Тоест, ако, например, увеличим  $n$  по 2, колко пъти ще нарасне времето за изпълнение? В дясната част времето за изпълнение е линейно по  $n$  и ще нарасне приблизително 2 пъти, а в лявата част времето за изпълнение въобще няма да се измени, тоест то въобще не зависи от  $n$ .
- 3)  $n$  е асимптотно по-малко по порядък от  $m$ . Например,  $n = \log m$  или  $n = m^{0.99}$ , но не и  $n = 1/2m$  или  $n = m - 10$ .
- 4) Трябва да разделим всяко число на  $10^9$ . Отговор за  $n = 100$ :

	$T_1(n) = 140$	$T_2(n) = 50n + 30$	$T_3(n) = n^2$	$T_4(n) = n^2 + 2n$	$T_5(n) = 2^n$
$n = 100$	$1.4 \cdot 10^{-7}$ sec	$\sim 5 \cdot 10^{-6}$ sec	$10^{-4}$ sec	$\sim 10^{-4}$ sec	$1.26 \cdot 10^{21}$ sec Около 40 трилиона години. Големият взрив се е случил преди около 15 милиарда години.

## Редици и полезни правила

Аритметична редица

$$\sum_{i=1}^n i = \frac{1}{2}n(n+1) = \Theta(n^2)$$

Геометрична редица

$$(x \neq 1) \quad \sum_{i=0}^n x^i = x^{n+1} - \frac{1}{x-1} = \Theta(x^n)$$

Низходяща безкрайна геометрична редица

$$(0 < x < 1) \quad \sum_{i=0}^{\infty} x^i = \frac{1}{1-x} = \Theta(1)$$

Квадратна редица

$$\sum_{i=1}^n i^2 = \frac{n(n+1)(2n+1)}{6}$$

Кубична редица

$$\sum_{i=1}^n i^3 = \frac{n^2(n+1)^2}{4}$$

Линейност на сума

$$\sum_{i=1}^n (ca_i + b_i) = c \sum_{i=1}^n a_i + \sum_{i=1}^n b_i$$

Линейност на асимптотните обозначения

$$\sum_{i=1}^n \Theta(f(i)) = \Theta\left(\sum_{i=1}^n f(i)\right)$$

---

## Полезни закони за степенуването и логаритмите

$$a^{-k} = 1/a^k$$

$$(a^m)^n = a^{mn}$$

$$a^m a^n = a^{m+n}$$

$$\log_c(ab) = \log_c a + \log_c b$$

$$\log_c a^n = n \log_c a$$

$$\log_b a = \frac{\log_c a}{\log_c b}$$

$$\log_b(1/a) = -\log_b a$$

$$a^{\log_b c} = c^{\log_b a}$$

$$a = b^{\log_b a}$$

---

## Упражнения

---

### Препоръчителни упражнения от учебника

#### Глава 1

2.2-2

1.2-2

2.2-4

Задача 1-1

2.3-6

2.3-7

#### Глава 2

Задача 2-1

Задача 2-3	3.2-3
Задача 2-4	3.2-4
<u>Глава 3</u>	
3.1-1	Задача 3-1
3.1-2	Задача 3-2
3.1-3	Задача 3-3 (можете да пропуснете изразите съдържащи възвратната логаритмична функция $\log^*$ )
3.1-4	Задача 3-4

---

### Допълнителни упражнения

- 1) Даден е масив с размер A. Известно е, че първите  $n - \lceil \sqrt{n} \rceil$  елемента са сортирани. Предложете алгоритъм, който сортира A по линейна сложност от гледна точка на времето за изпълнение.
- 2) Даден е масив от реални числа с размер A и допълнително реално число z.
  - a) Предложете алгоритъм, който определя, дали съществуват два различни индекса i и j, така че  $A[i] + A[j] = z$ .
  - b) Какво трябва да се промени в алгоритъма, за да връща също и гореспоменатите i и j, ако те съществуват?
- 3) a) Анализирайте сложността от гледна точка времето за изпълнение на Alg1 в зависимост от n!
- 3) b) Анализирайте сложността от гледна точка времето за изпълнение на Alg2 в зависимост от n и m!
 

Използвайте предположението, че изчислението на израз от типа  $a^b$  изисква време  $\Theta(b)$ .

<b>Alg1 (n)</b>
1. $k \leftarrow n$
2. <b>while</b> $k > 0$
3. $k \leftarrow \lfloor k/5 \rfloor$
4. <b>for</b> $j \leftarrow 1$ to $\lfloor \log n \rfloor$
5. <b>for</b> $i \leftarrow 1$ to $\lfloor n/2 \rfloor$
6. $\text{print}(i)$

<b>Alg2 (n, m)</b>
1. <b>while</b> $n > 0$
2. $a \leftarrow m^n$
3. $i \leftarrow 1$
4. <b>while</b> $i \leq a$
5. $i \leftarrow i * 2$
6. $n \leftarrow n - 1$

### Допълнителни упражнения

- 4) Докажете или опровергайте всяко едно от следните твърдения по формален начин, като използвате определенията за асимптотни бариери (граници):

$$2^{2\log n} = \Theta(n \log n) \quad (\text{b}) \quad \text{a)} \quad \log(n \cdot \log n) = \Theta(\log n)$$

$$\binom{n}{3} = \Theta(n^3) \quad (\text{d}) \quad \text{c)} \quad 2^{2n+1} + 3^n = O(2^{2n})$$

- 5) Подредете следните функции по асимптотен порядък:

$$t_1(n) = 2^{\sqrt{n}} \quad t_2(n) = \log(n^n \cdot n!) \quad t_3(n) = (\log \log n)^{\log n}$$

6) Две точки от задача 3-4 от учебника:

Нека  $f(n)$  и  $g(n)$  са две положителни асимптотни функции. Докажете или опровергайте:

- c) Ако  $f(n) = O(g(n))$ , то  $\log(f(n)) = O(\log(g(n)))$ . Също така е дадено, че двете функции не са крайни (ограничени).
- d) Ако  $f(n) = O(g(n))$ , то  $2^{f(n)} = O(2^{g(n)})$ .

7) Покажете пример за две функции  $f(n)$  и  $g(n)$ , за които не е изпълнено  $f(n) = O(g(n))$ , а също не е изпълнено и  $f(n) = \Omega(g(n))$  (и от тук, че аналогията с алгебрата не винаги работи – както си спомняме, за всеки две реални числа  $x$  и  $y$  е изпълнено поне едно от двете: или  $x \leq y$  или  $x \geq y$ )!

---

Решение 1

Първо ще сортираме последните  $n - \lceil \sqrt{n} \rceil$  елемента на  $A$  с помощта на сортирането чрез балончета (Bubble Sort), например.

След това ще слеем под-масив  $A[1 .. n - \lceil \sqrt{n} \rceil]$  с под-масив  $A[n - \lceil \sqrt{n} \rceil + 1 .. n]$ .

Сложност от гледна точка времето за изпълнение:

- Етапът на сортирането ще изиска  $\Theta((\sqrt{n})^2) = \Theta(n)$
- Етапът на сливането ще изиска  $\Theta((n - \lceil \sqrt{n} \rceil) + \lceil \sqrt{n} \rceil)$
- Общо  $\Theta(n)$ .

Напомняне: Времевата сложност при сливането на два сортирани масива с размери  $n$  и  $m$  е  $\Theta(n + m)$ .

---

Решение 2

Директно решение:

Ще прегледаме всички възможни двойки (цикъл вложен в друг цикъл) и ще проверим сумите им. Съществуват  $\binom{n}{2}$  възможни двойки, и затова времевата сложност е  $\Theta(n^2)$ .

Ефективно решение:

На първия етап ще сортираме  $A$  с помощта на сортирането чрез сливане (Merge Sort). След това ще обходим елементите на  $A$  и за всеки елемент  $A[i]$  ще търсим  $z - A[i]$  чрез двоично търсене (Binary Search) в  $A$ . Ако търсенето успее в даден момент, ще върнем `true`, иначе ще върнем `false`.

Времева сложност: етапът на сортирането ще изиска  $\Theta(n \log n)$ . Освен това  $n$  пъти двоично търсене –  $\Theta(n \log n)$ . Общо:  $\Theta(n \log n)$ .

## Продължение на решение 2

По-ефективно решение (описан с псевдо-код):

Времева сложност:

Етапът на сортирането ще изисква  $\Theta(n \log n)$ . Редове 2-10 отнемат  $\Theta(n)$ .

Общо:  $\Theta(n \log n)$ .

```
Find-Sum ( $A, n, z$ )
1. Merge-Sort( $A, n$ )
2.  $l \leftarrow 1, r \leftarrow n$ 
3. while  $l < r$ 
4.     if  $A[l] + A[r] = z$ 
5.         return true
6.     if  $A[l] + A[r] < z$ 
7.          $l \leftarrow l + 1$ 
8.     else
9.          $r \leftarrow r - 1$ 
10.    return false
```

Забележки:

- Второто решение е по-ефективно от гледна точка на времето за изпълнение, но не по порядък, а по константи (етапът на сортирането е „тясното гърло“).
  - Обърнете внимание, че представихме едното решение с думи, а другото с псевдо-код!
  - Доказателството на правилността на първото решение е тривиално и следва от правилността на сортирането чрез сливане и на двоичното търсене. За да докажем правилността на второто решение, трябва да формулираме подходяща инвариантна цикъл.
- 

## Продължение на решение 2

b) В първото решение, ако сме намерили подходящи  $i$  и  $j$ , просто ще ги върнем.

Ще обърнем внимание, че при по-ефективните решения ние сортираме масива и затова, ако сме намерили подходящи  $i$  и  $j$ , това не е задължително да бъдат индексите на тези елементи в първоначалния масив.

Затова ще използваме помощен масив с размер  $n$ , в който за всеки елемент ще запазваме първоначалния му индекс. По време на сортирането, когато променяме местоположението на индексите, заедно с тях ще променяме местоположението и на тези индекси, и така в края на сортирането ще върнем първоначалните индекси на елементите (ако сме ги намерили).

---

## Решение 3

a)  $\Theta(n \log^2 n)$

б) за всяко  $k$  между 1 и  $n$  изпълняваме:

- еднократно изчисление на  $m^k$  за време  $\Theta(k)$
- $\log(m^k)$  итерации на вътрешния цикъл, като при всяка една имаме постоянен брой действия

Общо:

$$\sum_{k=1}^n (\Theta(k) + \Theta(\log(m^k))) = \sum_{k=1}^n (\Theta(k) + \Theta(k \log m)) = \sum_{k=1}^n \Theta(k \log m) = \Theta(n^2 \log m)$$

#### Решение 4

$$\log(n-\log n) \leq \log n \quad (c=1, n_0=1)$$

$$\log(n-\log n) \geq \log(n-\frac{1}{2}n) = \log(\frac{1}{2}n) = \log n - 1 \geq \log n - \frac{1}{2}\log n = \frac{1}{2}\log n \quad (c=\frac{1}{2}, n_0=4)$$

$n \geq 4$

$n \geq 4$

---

## Тема 2

# Основни структури от данни Basic Data Structures

### В програмата:

Глава 10 от учебника

- Ще опознаем и ще дадем определение за следните понятия:
    - Абстрактен тип данни (ADT = Abstract Data Type)
    - Структура от данни (Data Structure)
  - Ще повторим и ще задълбочим изучаването на темата за основните структури от данни
    - стек (пълнител), опашка, масив, свързан списък, дърво
- 

### Структури от данни – Data Structures

Абстрактен тип данни (ADT = Abstract Data Type):

Колекция от действия върху някакви данни:

- Потребителят, използваш ADT, познава действията и влиянието им върху данните.
- От него не се изисква да познава подробните около имплементацията.

Например:

#### **LIFO = Last In, First Out**

Пълнител (стек) (Stack) (под пълнител имайте предвид пълнител на оръжие)

Определя се от следните действия:

- Push(S, x) – вкарване на елемент в началото на стека (включително проверка за препълване – overflow)
- Pop(S) – изкарване на элемента, който стои в началото на стека (включително проверка за отрицателно препълване – underflow)
- StackEmpty(S) и StackFull(S) – проверка дали стекът е празен / пълен

Структура от данни (Data Structure): метод за съхранение и организация на данни, която позволява изпълнение на определени действия. Структурата от данни всъщност е имплементация на абстрактен тип данни (ADT).

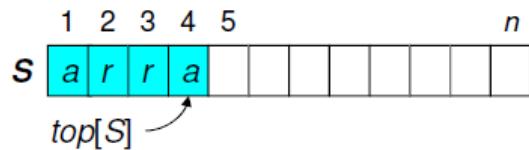
**Възможни структури от данни за имплементация на пълнител (стек):**

1. Масив
  2. Свързан списък
-

## Имплементация на стек (пълнител) посредством масив

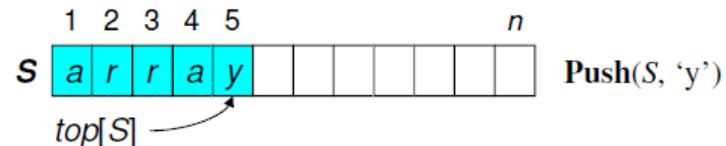
Push( $S, x$ )

1. **if** StackFull( $S$ )
2.     **error** "overflow"
3.  $top[S] \leftarrow top[S] + 1$
4.  $S[top[S]] \leftarrow x$



Pop( $S$ )

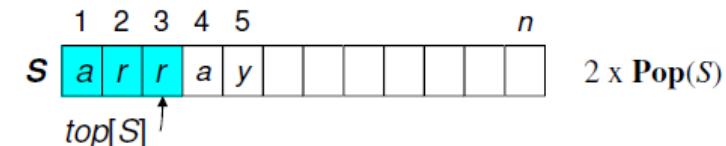
1. **if** StackEmpty( $S$ )
2.     **error** "underflow"
3.  $x \leftarrow S[top[S]]$
4.  $top[S] \leftarrow top[S] - 1$
5. **return**  $x$



Push( $S, 'y'$ )

StackFull( $S$ )

1. **return** ( $top[S] = length[S]$ )



2 x Pop( $S$ )

StackEmpty( $S$ )

1. **return** ( $top[S] = 0$ )

## Структури от данни – Data Structures

**FIFO = First In, First Out**

### Опашка (Queue)

Определя се от следните действия:

- Enqueue(Q, x) – вкарване на елемент в края на опашката (включително проверка за препълване – overflow)
- Dequeue (Q) – изкарване на елемента, който стои в началото на опашката (включително проверка за отрицателно препълване – underflow)
- QueueEmpty(Q) и QueueFull(Q) – проверка дали опашката е празна / пълна

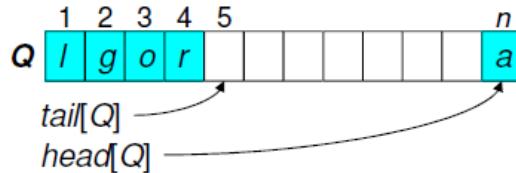
Възможни структури от данни за имплементация на опашка:

1. Масив
2. Свързан списък

## Имплементация на опашка посредством масив

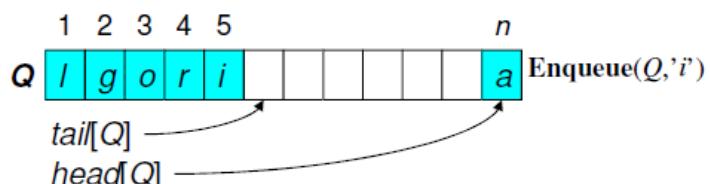
Enqueue( $Q, x$ )

1. **if** QueueFull( $Q$ )
2.     **error** "overflow"
3.  $Q[tail[Q]] \leftarrow x$
4.  $tail[Q] \leftarrow (tail[Q] \text{ mod } n) + 1$



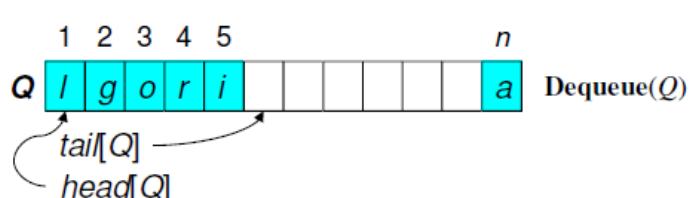
Dequeue( $Q$ )

1. **if** QueueEmpty( $Q$ )
2.     **error** "underflow"
3.  $x \leftarrow Q[head[Q]]$
4.  $head[Q] \leftarrow (head[Q] \text{ mod } n) + 1$
5. **return**  $x$



QueueEmpty( $Q$ )

1. **return** ( $head[Q] = tail[Q]$ )

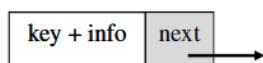


QueueFull( $Q$ )

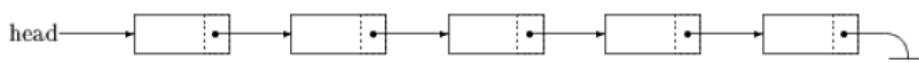
1. **return** (( $tail[Q] \text{ mod } n + 1 = head[Q]$ )

## Свързани списъци – Linked Lists

Свързаният списък е структура от данни, съставена от поредица от записи, като у всеки един запис има показалец (пойнтьр) към следващия запис.



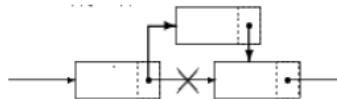
Както при масива, елементите са подредени в линеен ред, но с помощта на показалци (и не са задължително последователни в паметта).



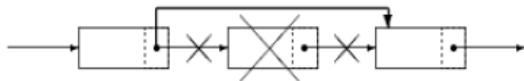
Последният запис сочи към Nil, а началото на списъка сочи към първия запис.

### Действия върху свързани списъци

- Търсене: посредством линейно обхождане на списъка, докато намерим елемент с търсения ключ, или докато стигнем до края на списъка.
  - време:  $\Theta(n)$
- Вкарване: при даден показалец към елемента, който е преди мястото за вкарване.
  - време:  $\Theta(1)$

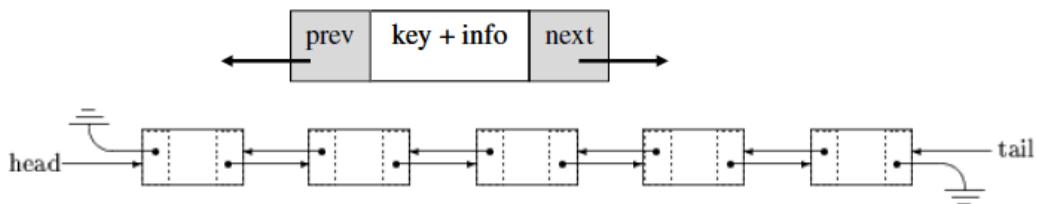


- Изваждане: при даден показалец към елемента, който е преди мястото за изтриване.
  - време:  $\Theta(1)$



## Версии на свързани списъци

Двупосочен свързан списък:



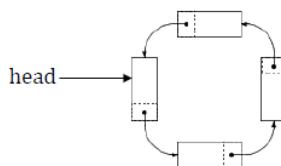
Преимущество: Може да се изведи елемент при даден показалец към него (а не към елемента пред него), а също и да се вкара елемент преди или след даден елемент.

Имплементация на търсене, вкарване и изтриване от двупосочен свързан списък можете да видите в учебника, стр. 172-173.

Имена на действията: List-Search(L, k), List-Insert(L, x), List-Delete(L, x)

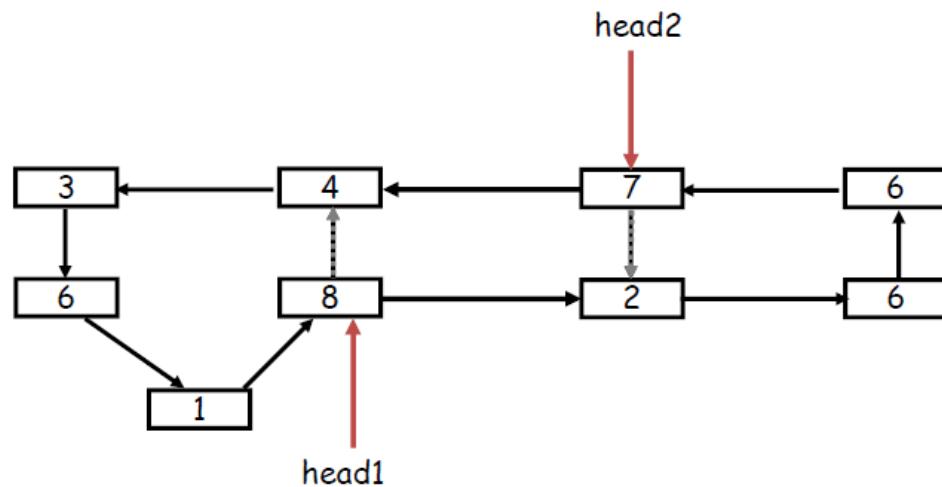
Сортиран свързан списък е свързан списък, в който редът на записите съвпада с реда на ключовете.

Кръгов свързан списък:



## Версии на свързани списъци

Изборът на вида на списъка, който ще използваме, зависи от ограниченията и целите. Например, обединението на списъци е лесно за изпълнение за постоянно време при кръгови списъци.



### Версии на свързани списъци

#### Упражнение

Както казахме, за изтриването на елемент от еднопосочен свързан списък, трябва да разполагаме с показалец към елемента пред него:



Предложете метод за изтриването на елемент от еднопосочен свързан списък при даден показалец към него!

Какви са недостатъците на метода?

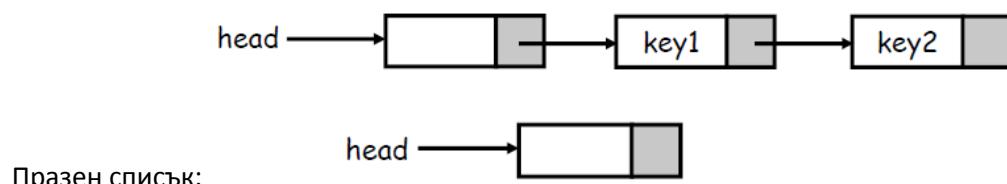
с 10

### Часови (страж) на свързан списък

Когато трием елемент от началото на списък или вкарваме елемент в началото на списък, трябва да актуализираме показалеца към началото му.

Това произвежда по-тромав код вследствие обработката на тези крайни случаи.

Може да се добави празен запис в началото на свързания списък, и тогава тези крайни случаи се обработват в рамките на обикновените случаи. Такъв запис се нарича **часови (страж) (sentinel)**.



Сега няма разлика между вкарване в началото на списъка и обикновено вкарване (началото (head) не се изменя):



### Свързани списъци спрямо масиви

Както казахме, свързаният списък е възможна структура от данни за ADT (абстрактните типове данни) стек (пълнител) и опашка.

- Стекът ще се имплементира, например, като еднопосочен свързан списък (елементи ще се вкарват и изкарват само от началото на списъка)
- Опашката ще се имплементира, например, като еднопосочен свързан списък с допълнителен показалец към края на списъка.

Кое е за предпочитане – масив или свързан списък?

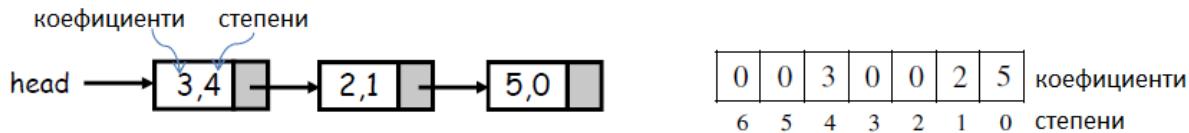
#### Предимства на свързания списък

- 1) Позволява заделяне на динамична памет в момент на нужда.
- 2) Не е нужно да се заделя място в непрекъснатата памет.
- 3) Изваждането на елемент от средата на свързан списък не оставя „дупка“. Затова обхождането на елементите е за линейно време по номера им.

#### Предимства на масива

- 1) Достъп до елемент по индекс за време  $\Theta(1)$ .

Пример: Да предположим, че искаме да намерим ефективен метод за представянето на полиноми най-много от шеста степен. Как ще бъде представен полиномът  $3x^4 + 2x + 5$ ? Кое е по-ресурсно-пестеливо?



### Масив и свързан списък като абстрактен тип данни (ADT)

На практика можем да се отнасяме към масива и към свързания списък не само като към структури от данни за имплементацията на ADT, но и като практически ADT. Например, за масив:

Масив (array)

Определя се от следните действия:

- Get(A, i) – връщане на елемента с индекс i
- Put(A, i, x) – складиране на елемента x на индекс i
- ArrayEmpty(A) и ArrayFull(A) – проверка дали масивът е празен / пълен

Структури от данни за масива:

- 1) Непрекъсната област в паметта, когато индекс i се изчислява от  $base + (i-1) \cdot \text{sizeof}(Type)$
- 2) Прекъсната област в паметта, тогава изчислението на индекса е по-сложно
- 3) Свързан списък (по принцип не е съвсем възможен) ...

И така понятията Абстрактен тип данни (ADT) и Структури от данни зависят от контекста.

.....

### Дву-/много-мерни масиви като абстрактен тип данни (ADT)

Двумерен масив	k-мерен масив
<ul style="list-style-type: none"><li>• Get(A, i, j)</li><li>• Put(A, i, j, x)</li><li>• ArrayEmpty(A) и ArrayFull(A)</li></ul>	<ul style="list-style-type: none"><li>• Get(A, i<sub>1</sub>, i<sub>2</sub>, i<sub>3</sub>, ..., i<sub>k</sub>)</li><li>• Put(A, i<sub>1</sub>, i<sub>2</sub>, i<sub>3</sub>, ..., i<sub>k</sub>, x)</li><li>• ArrayEmpty(A) и ArrayFull(A)</li></ul>

Стандартната структура от данни за дву-мерен масив от порядъка  $m * n$  е непрекъсната област в паметта, съставена от m едно-мерни масива с дължина n за всеки от тях.

- Разположението (i, j) се изчислява чрез  $base + ((i-1) \cdot n + (j-1)) \cdot \text{sizeof}(Type)$

#### Въпрос

При стандартна имплементация на матрица  $n * n$  времевата сложност на действието "транспониране" (разместване, разбръкване, мат. прехвърляне (в другата част на уравнението с обратен знак)) (transpose) е  $\Theta(n^2)$ . Предложете друга имплементация на матрица  $n * n$ , в която сложността на транспонирането е  $\Theta(1)$ , без да засягате сложността на действията Get и Put!

.....

### Дву-/много-мерни масиви

Ще дадем определение за следния абстрактен тип данни (ADT):

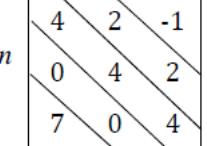
Диагонална матрица (Матрица на Тьоплиц) (Toeplitz matrix)

Квадратна матрица  $n * n$ , за която елементите на всичките й диагонали са еднакви.

Определя се от следните действия:

- Get(i, j) – връщане на елемента на местоположението (i, j).
- Put(i, j, x) – запазване на елемента x на местоположението (i, j). Трябва да се актуализира целия диагонал.

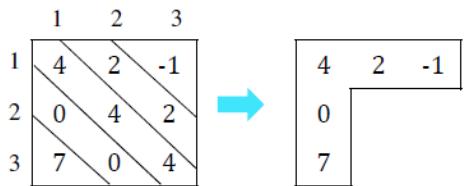
Стандартна имплементация ( $n$  едномерни масива):

$n$ 	<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="text-align: left;">сложност на мястото</th><th style="text-align: left;">Put(<math>i, j, x</math>)</th><th style="text-align: left;">Get(<math>i, j</math>)</th></tr> </thead> <tbody> <tr> <td style="text-align: left;"><math>\Theta(n^2)</math></td><td style="text-align: left;"><math>\Theta(n)</math></td><td style="text-align: left;"><math>\Theta(1)</math></td></tr> </tbody> </table>	сложност на мястото	Put( $i, j, x$ )	Get( $i, j$ )	$\Theta(n^2)$	$\Theta(n)$	$\Theta(1)$
сложност на мястото	Put( $i, j, x$ )	Get( $i, j$ )					
$\Theta(n^2)$	$\Theta(n)$	$\Theta(1)$					

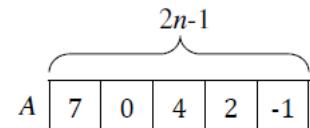
## Дву-/много-мерни масиви

По-ефективна структура от данни?

Ще запазим само по един „представител“ на всеки диагонал.



Тоест, ще запазим масив с дължина  $2n-1$ .



Ще обърнем внимание, че  $i-j$  е константа за всеки диагонал:  $j-i = \underbrace{-2, -1, 0, 1, 2}_{2n-1}$

Get( $i, j$ )

1. **return**  $A[j-i+n]$

Put( $i, j, x$ )

1.  $A[j-i+n] \leftarrow x$

сложност на местоположението	Put( $i, j, x$ )	Get( $i, j$ )
$\Theta(n)$	$\Theta(1)$	$\Theta(1)$

## Инициализация на масив за константно време

Инициализацията на масив  $A$  с дължина  $n$  от нули (или който и да е друг начален масив) изиска време  $\Theta(n)$ .

Това е особено проблемно, когато става дума за големи масиви, от чиито елементи само малка част биха били в употреба. Тоест, действията четене и писане биха били „евтини“, но инициализацията ще е „скъпа“.

Сега ще покажем метод за инициализация на масив за време  $\Theta(1)$  и в замяна ще използваме още памет.

### Грешно решение

Вместо да занулим A, ще поддържаме допълнителен масив INIT и в него ще запишем 1 само на местата, на които A съдържа „истинска“ стойност. По време на достъпването на елемент от A, който не съдържа истинска стойност, ще върнем 0.

<i>A</i>	?	8	?	?	3	?	?	?
<i>INIT</i>		1			1			

Проблемът:

Останалите места в INIT също биха могли да съдържат 1 и затова трябва да се инициализира!

---

### Инициализация на масив за константно време

#### Решение

Ще поддържаме два допълнителни масива и допълнителна променлива.

Начално положение – всички клетки са идентифицирани като „отпадък“.

	1	2	3	4	5	6	7	8
<i>A</i>	X	X	X	X	X	X	X	X
<i>from</i>	X	X	X	X	X	X	X	X
<i>to</i>	X	X	X	X	X	X	X	X

*↑*  
*top=1*

Например:  $A[5] \leftarrow 3$

*Get* ще върне 3 за всяко  $i = 5$

За всяко друго  $i$  ще върне 0

	1	2	3	4	5	6	7	8
<i>A</i>	X	X	X	X	3	X	X	X
<i>from</i>	X	X	X	X	1	X	X	X
<i>to</i>	5	X	X	X	X	X	X	X

*↑*  
*top=2*

#### **Is-Garbage( $i$ )**

1. if  $1 \leq from[i] < top$  and  $to[from[i]] = i$
2. return "not garbage"
3. else return "garbage"

#### **Get( $A, i$ )**

1. if Is-Garbage( $i$ )
2. return 0 (or other initial value)
3. else return  $A[i]$

#### **Put( $A, i, x$ )**

1.  $A[i] \leftarrow x$
2. if Is-Garbage( $i$ )
3.  $from[i] \leftarrow top$
4.  $to[top] \leftarrow i$
5.  $top \leftarrow top + 1$

## Инициализация на масив за константно време

Още един пример: $A[8] \leftarrow 12$	<b>Is-Garbage(<math>i</math>)</b>																																				
<b>Get</b> ще върне 12 за $i = 8$	1. <b>if</b> $1 \leq from[i] < top \text{ and } to[from[i]] = i$																																				
3 за $i = 5$	2. <b>return</b> "not garbage"																																				
За всяко друго $i$ ще върне 0.	3. <b>else</b> <b>return</b> "garbage"																																				
<table border="1"> <tr> <td>A</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td> </tr> <tr> <td></td><td>X</td><td>X</td><td>X</td><td>X</td><td>3</td><td>X</td><td>X</td><td>12</td> </tr> </table> <table border="1"> <tr> <td>from</td><td>X</td><td>X</td><td>X</td><td>X</td><td>1</td><td>X</td><td>X</td><td>2</td> </tr> <tr> <td>to</td><td>5</td><td>8</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td> </tr> </table> $top = 3$	A	1	2	3	4	5	6	7	8		X	X	X	X	3	X	X	12	from	X	X	X	X	1	X	X	2	to	5	8	X	X	X	X	X	X	<b>Get(<math>A, i</math>)</b> 1. <b>if</b> Is-Garbage( $i$ ) 2. <b>return</b> 0 (or other initial value) 3. <b>else return</b> $A[i]$
A	1	2	3	4	5	6	7	8																													
	X	X	X	X	3	X	X	12																													
from	X	X	X	X	1	X	X	2																													
to	5	8	X	X	X	X	X	X																													

<b>Put(<math>A, i, x</math>)</b>	1. $A[i] \leftarrow x$
	2. <b>if</b> Is-Garbage( $i$ )
	3. $from[i] \leftarrow top$
	4. $to[top] \leftarrow i$
	5. $top \leftarrow top + 1$

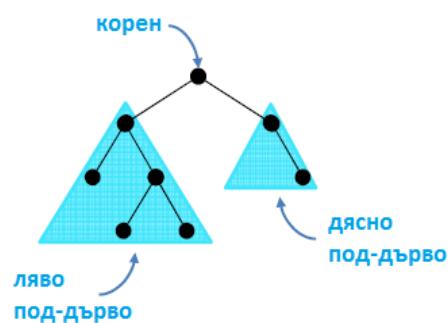
## Бинарни (двоични) дървета (Binary trees)

Бинарното дърво е структура от данни, съставена от записи, подредени в йерархичен ред (родител – дете), за разлика, например, от свързания списък, в който записите са подредени линейно (предишен – следващ).

Определение за бинарно дърво може да се даде рекурсивно:

### Бинарно дърво:

- или въобще не съдържа записи (празно бинарно дърво)
- или съдържа 3 чужди групи от записи:
  1. един запис, който се нарича корен (root)
  2. бинарно дърво, което се нарича ляво под-дърво
  3. бинарно дърво, което се нарича дясно под-дърво

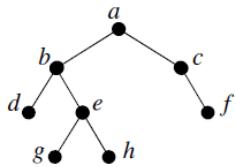


с 20

## Бинарни дървета (Binary trees)

### Понятия

- Запис в бинарно дърво се нарича кръстовище (възел) (връх) (node)
- „Връзката“ между два записа се нарича дъга (рамо) (edge)

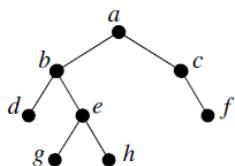


- За кръстовище v:
    - Левият наследник (left child) на v е корен на лявото му под-дърво
    - Десният наследник (right child) на v е корен на дясното му под-дърво
    - v се нарича родител (parent) на наследниците си, а те се наричат братя и сестри
    - Праотец (предшественик) (ancestor) на v е кръстовище, което се намира между него и корена
    - Потомък (descendant) на v е кръстовище, което се намира в под-дървото, на което v е корен
  - Кръстовище без наследници ще се нарича листо (leaf), иначе ще се нарича вътрешно кръстовище (internal node).
- 

## Бинарни дървета (Binary trees)

### Понятия

- Маршрут (пътека) (path) в дървото е поредица от кръстовища, между всеки две от които има дъга. Дължината на маршрута е броя на дъгите в него.



Например: Дължината на маршрута (a, b, e, h) е 3.

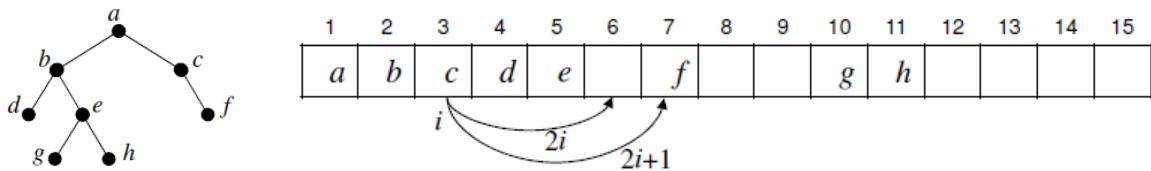
- Дълбочина (depth) на кръстовище е дължината на маршрута от корена на дървото до него.  
Например:  $\text{depth}(e) = 2$     $\text{depth}(a) = 0$
- Височина (height) на кръстовище е дължината на маршрута от него до най-дълбокото листо, което му е потомък.  
Например:  $\text{height}(b) = 2$     $\text{height}(a) = 3$

Височината на едно дърво е височината на корена му.

---

## Съхранение в паметта на бинарните дървета

Бинарно дърво може да се съхранява в масив, ниво след ниво, отляво надясно. За да се съхрани йерархичната структура, „отсъстващите“ наследници ще бъдат представени като празни клетки (което ще доведе до прахосване на памет):

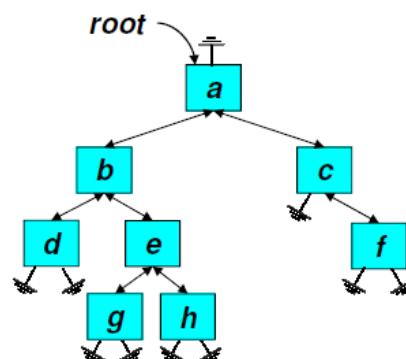
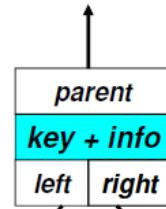


В повечето случаи е по-ефективно да се използват записи с показалци:

- \* У всеки запис има показалци към левия наследник и към десния наследник.
- \* А понякога и показалци към родителя.

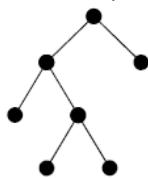
При корена на дървото `parent = Nil`.

При листата `left = right = Nil`.

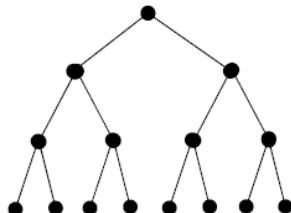


### Класификация на бинарните дървета

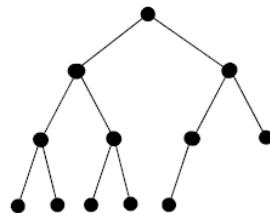
- У едно пълно дърво (full tree) всяко кръстовище има 2 или 0 наследника.



- Завършено дърво (complete tree) е пълно дърво, при което всички листа са с еднаква дълбочина.



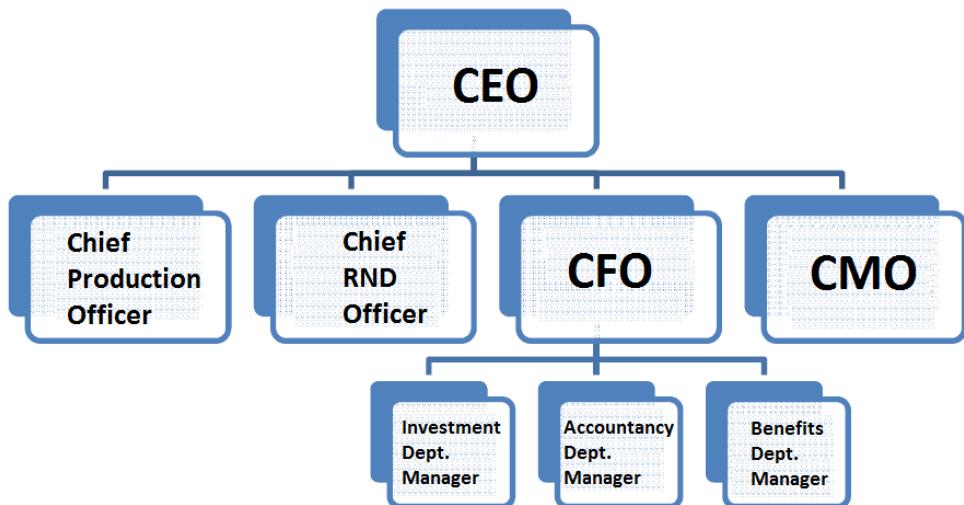
- Почти завършено дърво (almost complete tree) е завършено дърво, при което липсват последователни листа от дясната страна.



Съхранението в масив е по-подходящо за завършени или почти завършени дървета, тъй като тогава няма „дупки“.

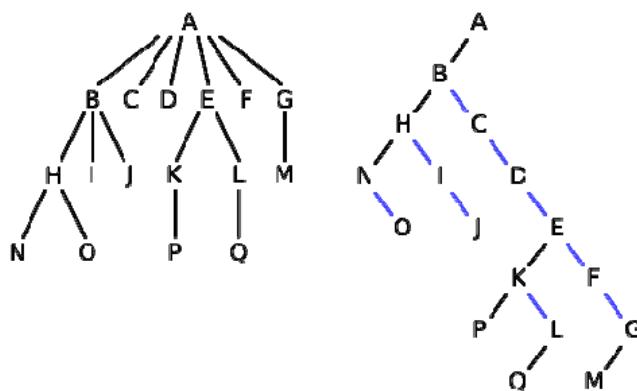
## Дървета от степен > 2

Понякога бихме искали да използваме дърво от степен по-голяма от 2.



## Дървета от степен > 2

Всяко дърво, чиято степен (максималният брой на наследниците на едно кръстовище) е равна на някакво естествено число k, може да се представи като бинарно дърво:



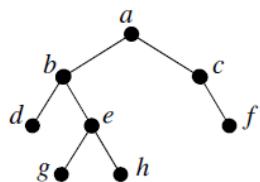
- На практика също и дървета от неограничена степен могат да се представят като бинарни дървета (подробности в учебника, стр. 180-181).

## Въпроси за преговор

- На езика С може да се създаде двумерен масив с размери 10\*10 посредством следния ред: int A[10][10];

Обяснете разликата между абстрактния тип данни (ADT) двумерен масив и структурата от данни, която се образува при изпълнението на гореспоменатия ред!

2. Обяснете защо не е възможно да се изпълни бинарно търсене върху сортиран свързан списък за време  $\Theta(\log n)$ . Кое свойство на масива не съществува за свързания списък?
3. Видяхме два метода за съхранение на бинарни дървета: масив и записи с показалци. Да предположим, че всеки запис (с допълнителна информация) заема 10 байта и всеки показалец заема 4 байта. Колко байта ще заеме всяка една от имплементациите за следното дърво:



---

### Отговори на въпросите за преговор

1. Абстрактният тип данни (ADT) двумерен масив позволява четене и съхранение на стойности по два индекса. По време на изпълнението на гореспоменатия ред се създава имплементация на такъв ADT посредством непрекъсната област в паметта, съдържаща 10 едномерни масива с дължина 10 за всеки един от тях.
2. При бинарно търсене в сортиран масив съществува предположението (хипотезата), че може да се достъпи всеки елемент от масива за време  $\Theta(1)$  посредством неговия индекс. При свързания списък това предположение не съществува. Например, за да достигнем средния елемент, трябва да преминем приблизително през половината от елементите.
3. При масив ще се нуждаем от 11 клетки, всяка от която ще заеме 10 байта. Общо 110 байта.  
При записи с показалци: имаме 8 записи, всеки един от които заема по 10 байта за ключа и допълнителна информация и още 4 байта за всеки един от 3-те показалеца. Общо 176 байта.  
Тоест, за това дърво за предпочитане е първият начин.  
Но, ако бяхме добавили наследник към кръстовище  $h$ , то имплементацията чрез масив щеше да стане по-прахосническа.

---

### Упражнения

---

### Препоръчителни упражнения от учебника

<u>Глава 10</u>	
10.1-2	10.2-2
10.1-5	10.2-3
10.1-6	10.2-6
10.1-7	10.2-7
c 30	Задача 1-10

### Допълнителни упражнения

1. Въпрос 10.1-6 от учебника.

Покажете как може да се имплементира опашка посредством 2 стека (пълнителя)!

Анализирайте времето за изпълнение на действията върху опашката!

2. Ще дадем определение за „минимален стек“ като абстрактен тип данни (ADT) поддържащ следните действия:

- Create – инициализация на структурата от данни, когато е празна.
- Insert(x) – вкарване на числото x в структурата.
- RemoveLast – изкарване на последно вкараното число и връщането му като резултат.
- Min – връщане на най-малкото число в структурата (без изваждането му).
- ChangeMin(k) – промяна на най-малкото число в структурата към k.

Позволено е да се предположи, че във всеки даден момент всички числа в структурата са различни едно от друго.

а) Предложете имплементация на „минимален стек“, когато времевата сложност, нужна за първите четири действия, е  $O(1)$ , а за действието ChangeMin –  $O(t)$ , като t е броят на елементите в структурата, които са вкарани след минималния елемент!

б) Взето е решение е да се добави следното действие:

- Add(d) – добавяне на d към всеки от елементите. Необходима времева сложност:  $O(1)$ .

Обяснете, как да се имплементира новото действие, и какви са необходимите промени при имплементацията на действието от точка (а), така че да няма промяна в сложностите им!

---

3. Дадени са m на брой групи  $F_1, F_2, \dots, F_m$ . Всяка група може да съдържа елементи от  $\{1, 2, \dots, n\}$ . Трябва да планирате структура от данни, която да поддържа следните действия (покажете структурата от данни и обясните как ще се извърши всяко едно от действията):

- Relate(i, a) – действие, което проверява, дали елементът a се намира в група  $F_i$ . Нужната времева сложност:  $O(1)$ .
  - Insert(i, a) – действие, което добавя елемента a към група  $F_i$ . Нужната времева сложност:  $O(1)$ .
  - Intersect(i, j, k) – действие, което извършва сечение (разрез) между групите  $F_i$  и  $F_j$  и вкарва резултата в  $F_k$  (тоест  $F_k = F_i \cap F_j$ ). Преди изпълнението на действието  $F_k$  е празна група. Нужната времева сложност:  $O(\min(|F_i|, |F_j|))$ .
  - Union(i, j, k) – действие, което извършва обединение на групите  $F_i$  и  $F_j$  и вкарва резултата в  $F_k$  (тоест  $F_k = F_i \cup F_j$ ). Преди изпълнението на действието  $F_k$  е празна група. Нужната времева сложност:  $O(\max(|F_i|, |F_j|))$ .
-

## Решение 1

Ще имплементираме опашката посредством два стека (пълнителя)  $S_1$  и  $S_2$  по следния начин:

Стек  $S_1$  ще представя края на опашката, а стек  $S_2$  ще представя началото на опашката. Тоест, нови елементи ще се вкарват в  $S_1$ , а най-старият елемент ще се изтегля от  $S_2$ .

По време на изкарването на елемент от опашката, ако  $S_2$  е празен, ще прехвърлим всички елементи един по един от  $S_1$  към  $S_2$  и тогава ще изтеглим елемента, който е в началото на  $S_2$ . Обърнете внимание, че след като направим това, няма нужда да се връщат всички елементи в  $S_1$ !

Следва псевдо-кодът за действията върху опашката, без проверки за препълване и отрицателно препълване overflow и underflow:

Dequeue( $Q$ )

1. if StackEmpty( $S_2$ )
2.     while not StackEmpty( $S_1$ )  
3.         Push( $S_2$ , Pop( $S_1$ ))
4. return Pop( $S_2$ )

Enqueue( $Q, x$ )

1. Push( $S_1, x$ )

Време за изпълнение:

$\Theta(n)$  в най-лошия случай

Време за изпълнение:

$\Theta(1)$

Как ще се имплементират QueueEmpty и QueueFull?

Как трябва да се променят Enqueue и Dequeue, за да се провери препълване и отрицателно препълване overflow и underflow?

.....

## Решение 2

Описание на структурата от данни:

Ще използваме двупосочен свързан списък.

За всеки запис в списъка, освен стойността на елемента, ще запазим допълнително поле, в което ще се съхранява минималната стойност сред стойностите на всички по-стари елементи (включително и самият елемент). Ще наречем това поле  $min$ .

Имплементация на действията:

Create – инициализация на празен свързан списък.

Insert( $x$ ) – добавяне на числото  $x$  в началото на списъка. Полето  $min$  на новия елемент ще бъде минимумът между  $x$  и полето  $min$  на предишното начало на списъка.

RemoveLast – изтриване началото на списъка и отпечатване на стойността му.

Min – връщане стойността на полето  $min$  принадлежащо на началото на списъка.

ChangeMin( $k$ ) – започваме от началото на списъка и се движим напред, докато стигнем до минималния елемент (как научаваме, че сме стигнали до него?). Променяме стойността на минималния елемент на ( $da бъде$ )  $k$  и актуализираме неговото поле  $min$  както при вкарване. Връщаме се назад до началото на списъка и по пътя актуализираме полето  $min$  на всеки елемент както при вкарване.

Всички действия се извършват за константно време, освен последното, което се изпълнява за  $O(t)$ .

### Решение 2(б)

Ще поддържаме променлива  $offset$ , която ще се инициализира със стойност 0. При всяко повикване на  $Add(d)$  ще добавяме към тази променлива  $d$ .

Промени в действията от точка (а):

$Insert(x)$  – вместо да вкарваме стойността  $x$ , ще вкарваме  $(x - offset)$ .

$RemoveLast$  – ще добавим към отпечатаната стойност  $offset$ .

$Min$  – ще добавим към връщаната стойност  $offset$ .

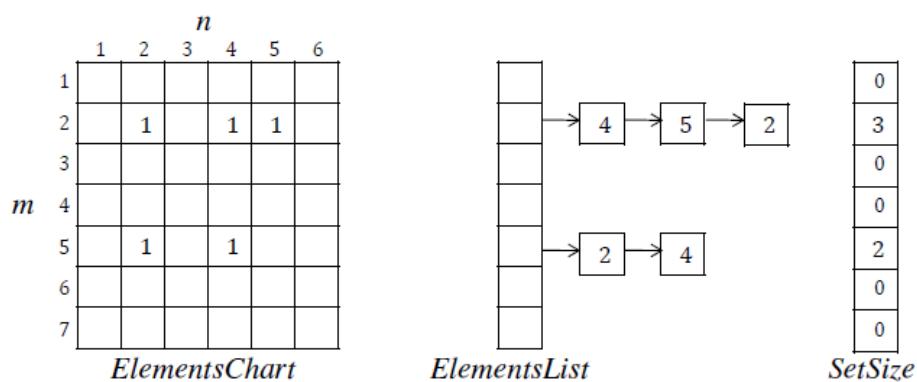
$ChangeMin(k)$  – вместо да променяме стойността на минималния елемент към  $k$ , ще го променим на  $(k - offset)$ .

Тоест, при четенето на стойност трябва да ѝ се добави  $offset$ , а при записването на стойност (добавяне / промяна) от нея трябва да се извади  $offset$ .

### Решение 3

#### Описание на структурата от данни:

- Булев масив с размери  $m * n$  на име  $ElementsChart$  като:  
 $ElementsChart[i, j] = True$  само и единствено, ако  $j \in F_i$ .
- Масив с размер  $m$  от свързани списъци на име  $ElementsList$  като:  
 $j$  се съдържа в списъка  $ElementsList[i]$  само и единствено, ако  $j \in F_i$ .
- Масив  $SetSize$  с размер  $m$ , в който  $SetSize[i] = |F_i|$ .



### Продължение на решение 3

Имплементация на действията:

```
INSERT( $i, a$ )
1   if not  $ElementsChart[i, a]$ 
2     then  $ElementsChart[i, a] \leftarrow \text{true}$ 
3     LIST - INSERT( $ElementsList[i], a$ )
4      $SetSize[i] \leftarrow SetSize[i] + 1$ 
```

```
RELATE( $i, a$ )
1   return  $ElementsChart[i, a]$ 
```

Извършваме  $\min\{|F_i|, |F_j|\}$  действия, всяко за време  $\Theta(1)$ . Общо:  $\Theta(\min\{|F_i|, |F_j|\})$ .

Intersect( $i, j, k$ )

Първо, ако  $SetSize[j] < SetSize[i]$ , ще извикаме  $Intersect(i, j, k)$ , така че във всеки скучай  $F_i$  да бъде най-малката.

Сега ще обходим  $ElementsList[i]$  и за всеки елемент  $x$  ( $x \in F_i$ ) ще проверим дали  $Relate(j, x) = \text{True}$ , и ако е така, ще извикаме  $Insert(k, x)$ .

Извършваме  $|F_i| + |F_j|$  действия, всяко от които за време  $\Theta(1)$ . Ще обърнем внимание, че  $|F_i| + |F_j| \leq 2\max\{|F_i|, |F_j|\}$ . И затова общо:  $\Theta(\max\{|F_i|, |F_j|\})$ .

Union( $i, j, k$ )

Ще обходим елементите на двата списъка  $F_i$  и  $F_j$ . За всеки елемент  $x$  ще извикаме  $Insert(k, x)$ .

## Тема 3

### Рекурсия и техниката „Разделяй, владей, присъединявай!“

### Recursion and the Divide-Conquer-Join Technique

#### В програмата:

Глава 4 от учебника

- Ще си припомним техниката „Разделяй, владей, присъединявай!“ за планирането на алгоритми.
  - Ще видим, как да изразяваме времето за изпълнение на рекурсивна функция посредством рекурентно отношение (recurrence relation).
  - Ще учим 4 метода за асимптотно решение (порядък) на рекурентно отношение:
    1. Методът на итерациите
    2. Методът дърво на рекурсията
    3. Главната теорема (Master Theorem)
    4. Методът на позиционирането  
(+ техниката за размяна на променливи)
- 

#### Техниката „Разделяй, владей, присъединявай!“

В тази лекция се ще срещнем с една от най-важните техники за планиране на алгоритми:  
Техниката „Разделяй, владей, присъединявай!“ (“Divide-Conquer-Join”).

С тази техника:

- Разделяме първоначалната задача на по-малки под-задачи → Разделяй
- Решаваме по отделно всяка под-задача → Владей
- Присъединяваме под-решенията към решението на първоначалната задача → Присъединявай

Например, при сортирането чрез сливане (Merge Sort):

- Разделяме масива на две половини
- Сортираме всяка половина по отделно
- Сливаме двете половини

Времето за изпълнение на рекурсивен алгоритъм може да се опише чрез рекурентно отношение (recurrence relation). Сега ще видим няколко метода за решаване на рекурентно отношение.

---

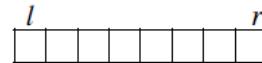
#### Метод 1 – Методът на итерациите

##### Пример 1

Ще анализираме времевата сложност на рекурсивно бинарно търсене в най-лошия случай.

Binary-Search-Rec ( $A, l, r, key$ )

1. **if**  $l > r$
2.     **return** NIL
3.  $mid \leftarrow \lfloor(l+r)/2\rfloor$
4. **if**  $key < A[mid]$
5.     **return** Binary-Search-Rec( $A, l, mid-1, key$ )
6. **else if**  $key > A[mid]$
7.     **return** Binary-Search-Rec( $A, mid+1, r, key$ )
8. **else return**  $mid$



Главното извикване (n е размерът на масива):

Binary-Search( $A, n, key$ )

1.     **return** Binary-Search-Rec( $A, 1, n, key$ )

$$t(n) = c + t(\lfloor n/2 \rfloor) = 2c + t(\lfloor n/4 \rfloor) = 3c + t(\lfloor n/8 \rfloor) = \dots = ic + t(\lfloor n/2^i \rfloor)$$

Кога  $n/2^i < 1$  ?

След  $\Theta(\log n)$  итерации (или по-точно:  $i = \lfloor \log n \rfloor + 1$  итерации).

$$= c (\lfloor \log n \rfloor + 1) + t(0) = \Theta(\log n)$$

## Метод 1 – Методът на итерациите

### Две технически подробности

- Размерът на една задача в повечето случаи е цяло число. Наред с това по принцип е възможно да пренебрегнем горните и долните гранични стойности, когато анализираме асимптотен порядък.  
Методът на позиционирането (която ще видим по-нататък) позволява да се докаже, че наистина става въпрос за пренебрежимо малка разлика.
- Ако не е отбелязано друго, предполагаме, че за малки стойности на n е изпълнено  $t(n) = \Theta(1)$ .

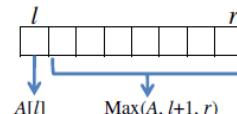
## Метод 1 – Методът на итерациите

### Пример 2

Дадена ни е рекурсивна функция, която изчислява максимума сред елементите на масива:

Max-Rec ( $A, l, r$ )

1. **if**  $l = r$
2.     **return**  $A[l]$
3.     **return** maximum{  $A[l]$  , Max-Rec( $A, l+1, r$ ) }



Главното извикване (n е размерът на масива):

Max-Array( $A, n$ )

1.     **return** Max-Rec( $A, 1, n$ )

Каква е времевата сложност на Max в зависимост от n?

$$t(n) = c + t(n-1) = 2c + t(n-2) = \dots = i \cdot c + t(n-i) = (n-1) \cdot c + t(1) = \Theta(n)$$

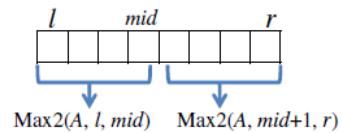

---

### Метод 1 – Методът на итерациите

#### Продължение на пример 2

Ето една друга рекурсивна версия за намирането на максимум:

```
Max-Rec-2( $A, l, r$ )
1. if  $l = r$ 
2.   return  $A[l]$ 
3.  $mid \leftarrow \lfloor (l+r)/2 \rfloor$ 
4.  $lmax \leftarrow \text{Max-Rec-2}(A, l, mid)$ 
5.  $rmax \leftarrow \text{Max-Rec-2}(A, mid+1, r)$ 
6. return maximum{  $lmax, rmax$  }
```



$$\begin{aligned} t(n) &= c + 2t(n/2) = c + 2c + 4t(n/4) = c + 2c + 4c + 8t(n/8) = \dots \\ &= c(1+2+\dots+2^{i-1}) + 2^i \cdot t(n/2^i) \\ &= c \underbrace{(1+2+4+8+\dots+2^{\log n - 1})}_{\text{инженерна редица}} + n \cdot t(1) = c \cdot (2^{\log n} - 1) + \Theta(n) \\ &= \Theta(n) \end{aligned}$$


---

### Метод 1 – Методът на итерациите

#### Пример 3

Сортиране чрез сливане (Merge Sort):

```
Merge-Sort( $A, l, r$ )
1. if  $l < r$ 
2.    $mid \leftarrow \lfloor (l+r)/2 \rfloor$ 
3.   Merge-Sort ( $A, l, mid$ )
4.   Merge-Sort ( $A, mid+1, r$ )
5.   Merge  $A[l..mid]$  with  $A[mid+1..r]$ 
```

$$\begin{aligned} t(n) &= 2t(n/2) + n = 2(2t(n/4) + n/2) + n = 4t(n/4) + 2n \\ &= 2^i \cdot t(n/2^i) + i \cdot n \\ &= n \cdot t(1) + n \log n = \Theta(n \log n) \end{aligned}$$


---

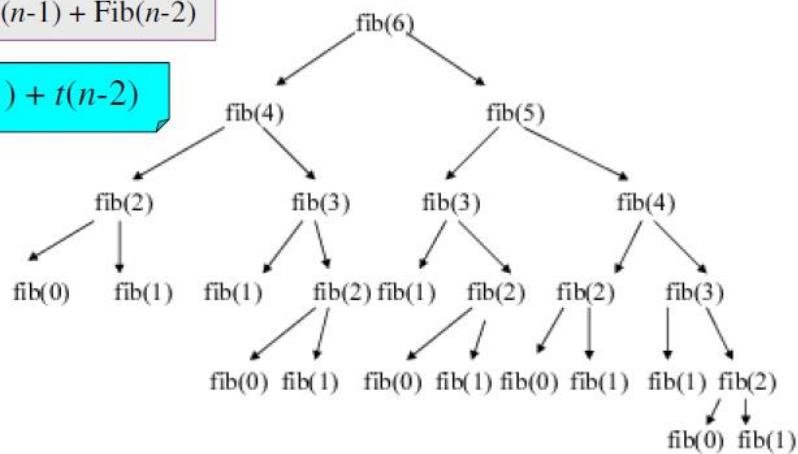
## Метод 1 – Методът на итерациите

### Пример 4

Изчисление на  $n$ -тия елемент от редицата на Фибоначи:

```
Fib(n)
1. if n = 0 or n = 1
2.   return n
3.   return Fib(n-1) + Fib(n-2)
```

$$t(n) = c + t(n-1) + t(n-2)$$



## Метод 1 – Методът на итерациите

### Продължение на пример 4

Ще намерим горна и добра граница (бариера).

$$t(n) = c + t(n-1) + t(n-2)$$

$$\begin{aligned}
 t(n) &\leq c + 2t(n-1) = c + 2c + 4t(n-2) = c + 2c + 4c + 8t(n-3) = \dots \\
 &= c(1+2+\dots+2^{i-1}) + 2^i \cdot t(n-i) \\
 &= c(1+2+\dots+2^{n-1}) + 2^n \cdot t(0) = c \cdot (2^n - 1) + 2^n = \mathcal{O}(2^n)
 \end{aligned}$$

$$\begin{aligned}
 t(n) &\geq c + 2t(n-2) = c + 2c + 4t(n-4) = c + 2c + 4c + 8t(n-6) = \dots \\
 &= c(1+2+\dots+2^{i-1}) + 2^i \cdot t(n-2i) = \Omega(\sqrt[4]{2^n})
 \end{aligned}$$

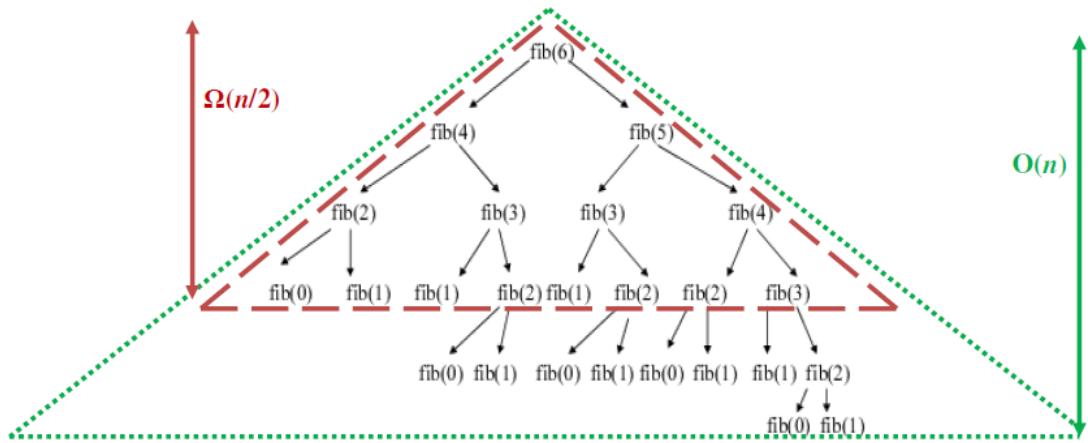
Чрез други методи, които се изучават в други курсове, може да се намери също и плътна граница:

$$t(n) = \Theta(\Phi^n) \quad \Phi = \frac{1+\sqrt{5}}{2}$$

## Метод 2 – Методът на рекурсивното дърво

Тип визуална версия на метода на итерациите. Разстиламе рекурсивното дърво на извикванията и обобщаваме количеството работа по всички кръстовища (възли).

Пример: Фиbonачи.



Количеството работа при всяко кръстовище (възел) е постоянно. Ще го отбележим с  $c$ .

$$\begin{aligned}
 &= c(1+2+\dots+2^{n-1}) = c(2^n-1) = O(2^n) \\
 \swarrow \nearrow &= c(1+2+\dots+2^{(n/2)-1}) = c(2^{n/2}-1) = \Omega(2^{n/2}) = \Omega(\sqrt{2^n})
 \end{aligned}$$

## Метод 2 – Методът на рекурсивното дърво

$$t(n) = 2t(n/2) + n^2$$

## Метод 2 – Методът на рекурсивното дърво

$$t(n) = t(n/3) + t(2n/3) + n$$

Обобщение:  $t(n) = t(\alpha n) + t(\beta n) + cn$ , когато  $\alpha + \beta = 1$ ,  $0 < \alpha, \beta < 1$  и  $c > 0$  е константа.

Въпрос: Какво се променя, когато  $\alpha + \beta < 1$ ?

## Метод 3 – Главната теорема (Master Theorem)

- $a \geq 1$  и  $b > 1$  са константи
- $f(n)$  е функция
- $T(n)$  се определя върху неотрицателни цели числа чрез рекурсията:  

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$$

Можем асимптотично да поставим граница на  $T(n)$  по следния начин:

1. Ако  $f(n) = O(n^{\log_b a - \varepsilon})$ , когато  $\varepsilon > 0$  е някаква константа,  
то  $T(n) = \Theta(n^{\log_b a})$

2. Ако  $f(n) = \Theta(n^{\log_b a})$ ,  
то  $T(n) = \Theta(n^{\log_b a} \cdot \log n)$

3. Ако  $f(n) = \Omega(n^{\log_b a + \varepsilon})$ , когато  $\varepsilon > 0$  е някаква константа,  
и също  $a \cdot f\left(\frac{n}{b}\right) \leq c \cdot f(n)$  за константа  $c < 1$  и стартирайки от достатъчно  
голямо  $n$ ,  
то  $T(n) = \Theta(f(n))$

Забележка: Значението на  $\frac{n}{b}$  може да бъде  $\left\lfloor \frac{n}{b} \right\rfloor$  или  $\left\lceil \frac{n}{b} \right\rceil$ .

---

### Метод 3 – Главната теорема (Master Theorem)

$$1. T(n) = 9T\left(\frac{n}{3}\right) + n$$

$$2. T(n) = T\left(\frac{n}{2}\right) + 1$$

---

### Метод 3 – Главната теорема (Master Theorem)

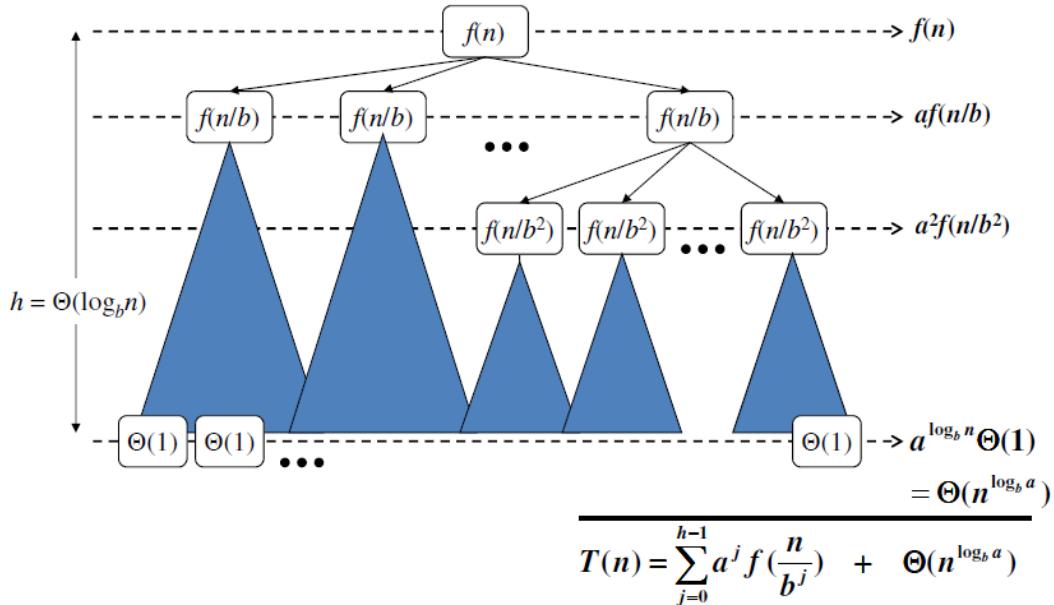
$$3. T(n) = 3T\left(\frac{n}{4}\right) + n \log n$$

$$4. T(n) = 2T\left(\frac{n}{2}\right) + n \log n$$

---

### Метод 3 – Главната теорема (Master Theorem)

Интуиция към теоремата (пълното доказателство на теоремата се намира в учебника):



### Метод 4 – Методът на позиционирането

При този метод ние отгатваме едно решение и го доказваме чрез индукция.

Недостатък: Методът не предоставя решение.

Значение: Позволява да се докаже решението на „трудни“ формули, за които е лесно да се отгатне решение.

Примери за „трудни“ формули, за които е лесно да се отгатне решение.

$$t(n) = t(\lceil n/2 \rceil) + 1 = \Theta(\log n)$$

$$t(n) = t(\lfloor n/2 \rfloor) + t(\lceil n/2 \rceil) + n = \Theta(n \log n)$$

$$t(n) = 2t(n/2 + 17) + 1 = \Theta(n)$$

## Метод 4 – Методът на позиционирането

$$t(n) = 2t(\lfloor n/2 \rfloor) + n$$

Ще демонстрираме метода върху следната формула:

Ще отгатнем горна граница:  $t(n) = O(n \log n)$

Ще докажем чрез индукция върху  $n$ : че съществува константа  $c$ , така че за всяко достатъчно голямо  $n$

$$t(n) \leq cn \log n$$

Предположение (хипотеза) на индукцията:  $t(\lfloor n/2 \rfloor) \leq c \lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor)$

$$\begin{aligned} t(n) &= 2t(\lfloor n/2 \rfloor) + n \leq 2(c \lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor)) + n \\ &\leq cn \log(n/2) + n \leq cn \log n - cn + n \leq cn \log n \end{aligned}$$

за  $c \geq 1$

Ами доказателството базата на индукцията?

По подобен начин се доказва долна граница (упражнение 4.1-2 от учебника).

## Размяна на променливи

Техника, която позволява понякога да се превърне формула, която изглежда „трудна“, в „лесна“.

$$t(n) = 2t(\sqrt{n}) + \log n$$

Ще обозначим:

$$m = \log n$$

$$t(2^m) = 2t(2^{m/2}) + m$$

Ще обозначим:

$$s(m) = t(2^m)$$

$$s(m) = 2s(m/2) + m \rightarrow \text{лесно е да се реши чрез всеки един от методите, които видяхме.}$$

$$s(m) = \Theta(m \log m)$$

$$t(n) = t(2^m) = s(m) = \Theta(m \log m) = \Theta(\log n \cdot \log \log n)$$

## Размяна на променливи

$$t(n) = 4\sqrt{n} t(\sqrt{n}) + n \cdot \log n \cdot \log \log n$$

Ще разделим на n:

$$\frac{t(n)}{n} = 4 \frac{t(\sqrt{n})}{\sqrt{n}} + \log n \cdot \log \log n$$

Ще обозначим:  $m = \log n$

$$\frac{t(2^m)}{2^m} = 4 \frac{t(2^{m/2})}{2^{m/2}} + m \cdot \log m$$

Ще обозначим:  $s(m) = \frac{t(2^m)}{2^m}$

лесно за  
решаване

$$s(m) = 4s(m/2) + m \cdot \log m = \Theta(m^2)$$

$$\frac{t(n)}{n} = s(m) = \Theta(m^2) = \Theta(\log^2 n)$$

$$t(n) = \Theta(n \log^2 n)$$

## Пример – умножение на матрици

- Търси се алгоритъм за умножение на матрици  $C = A * B$ 
  - $A, B, C$  са квадратни матрици с размери  $n * n$
  - Напомняне: Елементът  $C_{ij}$  на матрицата произведение  $C = A * B$  се изчислява чрез вътрешното умножение  $C_{ij} = \sum_{k=1}^n A_{ik} B_{kj}$
  - Прост алгоритъм:

```
Simple-Matrix-Multiply(A, B, C, n)
1. for i ← 1 to n
2.   for j ← 1 to n
3.     C[i, j] ← 0
4.     for k ← 1 to n
5.       C[i, j] ← C[i, j] + A[i, k] · B[k, j]
6. return C
```
  - Време за изпълнение  $\Theta(n^3)$ , памет  $\Theta(1)$

## Умножение на матрици - продължение

- А рекурсивен алгоритъм?
  - Идеята: Разделяй и владей (нека предположим, че  $n = 2^m$ )!

$$A \times B = \begin{array}{|c|c|} \hline A1 & A2 \\ \hline A3 & A4 \\ \hline \end{array} \times \begin{array}{|c|c|} \hline B1 & B2 \\ \hline B3 & B4 \\ \hline \end{array} = \begin{array}{|c|c|} \hline C1 & C2 \\ \hline C3 & C4 \\ \hline \end{array} = C$$

- Ще изчислим рекурсивно 8-те произведения на четвъртините и ще ги съберем

$$C1 = A1 \cdot B1 + A2 \cdot B3$$

$$C2 = A1 \cdot B2 + A2 \cdot B4$$

$$C3 = A3 \cdot B1 + A4 \cdot B3$$

$$C4 = A3 \cdot B2 + A4 \cdot B4$$

- Полученото рекурентно отношение (recurrence relation):  $T(n) = 8T(n/2) + 4(n/2)^2$
- Решение (случай 1 на главния метод):  $T(n) = \Theta(n^3)$
- Същото асимптотично време за изпълнение, но потреблението на паметта е  $\Theta(lgn)$
- Положението само се влошава!...

---

## Умножение на матрици - продължение

- Алгоритъмът на Шрасен (Strassen, 1969 г.)
  - Достатъчни са само 7 произведения на четвъртини!

$$P1 = A1 \cdot (B2 - B4)$$

$$P2 = (A1 + A2) \cdot B4$$

$$P3 = (A3 + A4) \cdot B1$$

$$P4 = A4 \cdot (B3 - B1)$$

$$P5 = (A1 + A4) \cdot (B1 + B4)$$

$$P6 = (A2 - A4) \cdot (B3 + B4)$$

$$P7 = (A1 - A3) \cdot (B1 + B2)$$

$$C1 = P4 + P5 + P6 - P2$$

$$C2 = P1 + P2$$

$$C3 = P3 + P4$$

$$C4 = P1 + P5 - P3 - P7$$

- Полученото рекурентно отношение (recurrence relation):  $T(n) = 7T(n/2) + 18(n/2)^2$
- Решение (случай 1 на главния метод):  $T(n) = \Theta(n^{\log_2 7}) = \Theta(n^{2.81})$
- Практически за  $n < 45$  простият алгоритъм е за предпочитане
- Най-добрият алгоритъм днес (от 1990 г, основно с теоретичен интерес):  $\Theta(n^{2.376})$

---

## Въпроси за преговор

1. При сортиране чрез сливане (Merge Sort) какво се прави при всеки един от етапите „Разделяй, владей, присъединявай“?
2. При метода на рекурсивното дърво защо искаме да намерим общата сума от стойностите на кръстовищата (възлите) в дървото?
3. Защо не може да се използва Главната теорема за решаване на рекурентно отношение (recurrence relation):  $T(n) = 2T(n/2) + n/\log n$  ?

4. Дадено е рекурентното отношение  $T(n) = 2T(n/2) + f(n)$ .

Дайте пример за функция  $f(n)$ , за която е изпълнен третият случай от Главната теорема, но не е изпълнено условието за регулярност на този случай!

Намек: Във функцията трябва да има разделение, например, между четно и нечетно  $n$ .

---

### Отговори на въпросите за преговор

1. „Разделяй!“: Разполовяване на масива на две. „Владей!“: Рекурсивно сортиране на всяка половина. „Присъединявай!“: Сливане на двете половини.

2. Всяко кръстовище (възел) представя време за изпълнение на някакъв етап от рекурсивния процес. Сумата на стойностите на възлите представя общото време за изпълнение на целия рекурсивен процес.

3. Защото не е изпълнено условието на нито един от случаите (дори на случай 1 – защо?).

4. Например:

$$f(n) = \begin{cases} n^2 & \text{if } n \text{ is even} \\ n^3 & \text{else} \end{cases}$$

Нека предположим в отрицание, че е изпълнено условието за регулярност. Тоест, съществува  $c < 1$ , така че за всяко достатъчно голямо  $n$  е изпълнено условието:  $a f(n/b) \leq c f(n)$ ,

но за всяко четно  $n$ , такова че  $n/2$  е нечетно, е изпълнено:

$$a f(n/b) = 2f(n/2) = 2(n/2)^3 = n^3/4 \leq cn^2$$

и това, разбира се, е невъзможно.

---

### Упражнения

#### Препоръчителни упражнения от учебника

Глава 4	4.2-2	4.3-2	Задача 4-2
4.1-1	4.2-3	4.3-3	Задача 4-3
4.1-2	4.2-4	4.3-4	Задача 4-4
4.1-6	4.2-5	4.3-5	
4.2-1	4.3-1	Задача 4-1	

---

#### Допълнителни упражнения

1. Намерете по метода на итерациите плътна асимптотна граница на рекурентното отношение (recurrence relation):  $T(n) = 3T(n/4) + n$

2. Дайте плътна асимптотна граница на следната рекурентното отношение:  
 $t(n) = n^3 + t(n/3) + t(n/4)$

3. Задача 4-2 от учебника

Масив A с размер n съдържа всички цели числа от 0 до n освен едно число.

а) Предложете алгоритъм, чието време за изпълнение е  $O(n)$  и, който намира липсващото число!

Сега ще предположим, че не можем да достъпим директно числата в A. Единственото позволено действие е достъп до някой бит от число в A, представено бинарно (това действие се извършва за постоянно време).

б) Повторете подточка (а) при гореспоменатото предположение!

4. Променете рекурсивния алгоритъм за бинарно търсене (Binary Search), така че вместо да разделяме масива на две (почти) равни части, той да го раздели на две части в съотношение 1:2, тоест да го раздели на една част с размер  $1/3$  и друга част с размер  $2/3$ . Напишете новия алгоритъм с псевдо код и анализирайте сложността му!

---

### Решение 1

$$\begin{aligned} T(n) &= 3T(n/4) + n \\ &= 3(3T(n/4^2) + n/4) + n = 3^2T(n/4^2) + n(1+3/4) \\ &= 3^2(3T(n/4^3)+n/4^2) + n(1+3/4) = 3^3T(n/4^3) + n(1+3/4+3^2/4^2) = \\ &\dots \\ &= 3^iT(n/4^i) + n(1+3/4+3^2/4^2+\dots+3^{i-1}/4^{i-1}) \end{aligned}$$

Рекурсията достига условието за спиране, когато  $n/4^i = 1$ , тоест  $i = \log_4 n$ .

$$\begin{aligned} T(n) &= 3^{\log_4 n} T(1) + n \cdot \sum_{k=0}^{\log_4 n - 1} \left(\frac{3}{4}\right)^k \\ &\leq 3^{\log_4 n} \cdot \Theta(1) + n \cdot \sum_{k=0}^{\infty} \left(\frac{3}{4}\right)^k = \Theta(n^{\log_4 3}) + 4n = o(n) + 4n \end{aligned}$$

И с това намерихме горна граница:  $T(n) = O(n)$ .

И освен това  $T(n) = 3T(n/4) + n > n$ , затова  $T(n) = \Omega(n)$ .

Общо получихме:  $T(n) = \Theta(n)$ .

с 30

---

### Решение 2

$$t(n) = n^3 + t(n/3) + t(n/4)$$

$$\text{Горна граница: } t(n) \leq n^3 + 2t(n/3)$$

$$\text{Долна граница: } t(n) \geq n^3 + 2t(n/4)$$

И в двета случая задачата може да се реши чрез Главната теорема (случай 3).

Двете граници са сходящи (клонят към конкретни числа) и получаваме  $t(n) = \Theta(n^3)$ .

### Решение 3

а) Ще сумираме числата в A и ще изчислим разликата от  $n(n + 1)/2$  до намерената сума. Тази разлика е липсващото число.

б) Нека обърнем внимание на следните свойства:

Ако се съдържат всички числа от 0 до n, то:

- Ако n е четно, броят числа, които завършват на бит 0 е с 1 по-голям от броя на числата, които завършват на 1.
- Ако n е нечетно, броят числа, които завършват на бит 0 е равен на броя на числата, които завършват на 1.

Тъй като липса едно число, ще можем да сравним броя числа, завършващи на 0, с броя числа, завършващи на 1, и в съответствие с това, дали n е четно или нечетно число, ще знаем какъв е последният бит на липсващото число.

След това ще продължим рекурсивно с групата, в която липса числото, и със следващия бит.

### Алгоритъмът:

- Ще инициализираме  $i \leftarrow 1$ .
- Ще обходим A и ще проверим най-малко значения за всяко число  $i$ -ти бит. Ще намерим  $i$ -тия бит на липсващото число в съответствие с гореспоменатите свойства.
- Ще продължим рекурсивно с групата, в която липса числото, и с  $i + 1$ .

### Продължение на решение 3

Подходящото рекурентно отношение (recurrence relation) е  $t(n) = n + t(n/2)$ .

Ще я решим по два начина:

#### По Метода на итерациите

$$\begin{aligned}t(n) &= n + t(n/2) = n + n/2 + t(n/4) = n + n/2 + n/4 + t(n/8) = \dots \\&= n(1 + \frac{1}{2} + \frac{1}{4} + \dots) + t(n/2^i) \\i = \log n &\leq 2n + t(1) = O(n)\end{aligned}$$

И с това намерихме горна граница:  $t(n) = O(n)$ .

Но също ни е ясно, че  $t(n) = \Omega(n)$ , тъй като  $t(n) \geq n$ .

Общо получихме:  $t(n) = \Theta(n)$ .

### Чрез Главната теорема

Изпълнено е  $n^{\log_2 1+\varepsilon} = n^\varepsilon$  и от тук става ясно, че говорим за третия случай:  $f(n) = n = \Omega(n^{\log_2 1+\varepsilon})$ .

Ще проверим също и условието за регулярност:  $af(n/b) = n/2 \leq cf(n) = cn$ .

И наистина съществува  $c < 1$ , както се изисква (например,  $c = 1/2$ ).

И от тук следва, че  $t(n) = \Theta(f(n)) = \Theta(n)$ .

.....

### Решение 4

```
BINARY-SEARCH1-2( $A, p, r, v$ )
1 if  $p > r$ 
2   then return NIL
3  $q \leftarrow p + \lfloor (r - p + 1)/3 \rfloor$ 
4 if  $v < A[q]$ 
5   then return BINARY-SEARCH1-2( $A, p, q - 1, v$ )
6 else if  $v > A[q]$ 
7   then return BINARY-SEARCH1-2( $A, q + 1, r, v$ )
8 else return  $q$ 
```

### Анализ на сложността

Най-лошият случай е, когато търсеният елемент не се намира в масива, а също така и когато при всеки етап от рекурсията се обръщаме към по-голямата част от масива (с размер приблизително две трети). В този случай:

$$t(n) = t(2n/3) + 1$$

Може да се реши чрез Главната теорема (случай 2) или по Метода на итерациите и получаваме:

$$t(n) = \Theta(\log n)$$

.....

## Тема 4

### Бързо сортиране Quick Sort

#### В програмата:

Глава 7 от учебника, с добавки:

- Ще учим за алгоритъма „Бързо сортиране“ (Quick Sort) и за алгоритъма за разделяне (Partition).
  - Ще покажем две версии на Бързото сортиране:
    1. Основна
    2. Случайна (по-нататък в курса ще разгледаме още една версия)
  - Ще се запознаем с понятието „случайни алгоритми“.
  - Ще учим, какво е стабилност (stability) на сортирането.
  - Ще учим, какво е доказателство за правилност (proof of correctness) на алгоритъм.
- 

#### Бързо сортиране – Quick Sort

- Развит от C. A. R. Hoare през 1962 г., когато е бил на 26 г.
- По подобие на Сортирането чрез сливане (Merge Sort) – рекурсивен алгоритъм, работи по метода „Разделяй, владей, присъединявай!“.
- За разлика от Сортирането чрез сливане (Merge Sort) –
  - сортира на място (in-place), тоест  $\Theta(1)$  елемента, складирани във всеки момент извън изходния масив.
  - върши цялата работа преди рекурсивните извиквания (етапът на разделянето (“Divide”)):  
разделя масива на две части (не задължително половинки!).
- Ще учим три версии:

Версия	Най-лош случай	Средно	Най-добър случай
Основна	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n \log n)$
Случайна	$\Theta(n^2)$		
Детерминантна (по-нататък в курса)	$\Theta(n \log n)$		

- Много бърз алгоритъм на практика
- 

#### Идеята

**Входните данни: масив A с размер n.**

### 1. „Разделяй!“

- Избери осев елемент (pivot)  $x$  от масива\*!
- Раздели  $A$  на две области:  $\leq x$  и  $> x$ , като  $x$  е между тях!



### 2. „Владей!“

- Сортирай рекурсивно  $A[1 \dots q-1]$
- Сортирай рекурсивно  $A[q+1 \dots n]$

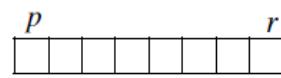
Версия	Оста (на всеки етап) се избира от под-масива, върху който работим:
Основна	като най-десния елемент
Случайна	като случаен елемент
Детерминантна	като медиана (median)

### 3. „Присъединявай!“

- Нищо, масивът е сортиран!

\* В други версии, различни от тези, които ще видим,  $x$  може да буде елемент, който въобще не е в  $A$ .

## Алгоритъмът



Входните данни: масив  $A$ , лява граница  $p$  и дясна граница  $r$ .

```
Quick-Sort ( $A, p, r$ )
1. if  $p < r$ 
2.    $q \leftarrow \text{Partition}(A, p, r)$ 
3.   Quick-Sort ( $A, p, q-1$ )
4.   Quick-Sort ( $A, q+1, r$ )
```

Първото извикване ( $n$  е размерът на масива):

```
Quick-Sort ( $A, 1, n$ )
```

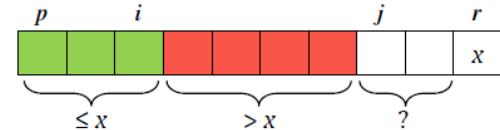
Какво е условието за спиране?

## Разделянето – Partition

<i>i</i>	<i>p, j</i>	<i>r</i>
	50 30 25 10 77 5 20 88 26	
<i>i</i>	<i>p j</i>	<i>r</i>
	50 30 25 10 77 5 20 88 26	
<i>i</i>	<i>p j</i>	<i>r</i>
	50 30 25 10 77 5 20 88 26	
<i>p, i</i>	<i>j</i>	<i>r</i>
	25 30 50 10 77 5 20 88 26	
<i>p i</i>	<i>j</i>	<i>r</i>
	25 10 50 30 77 5 20 88 26	
<i>p</i>	<i>i j</i>	<i>r</i>
	25 10 5 30 77 50 20 88 26	
<i>p</i>	<i>i j</i>	<i>r</i>
	25 10 5 20 77 50 30 88 26	
<i>p</i>	<i>i r, j</i>	
	25 10 5 20 77 50 30 88 26	
<i>p</i>	<i>i r, j</i>	
	25 10 5 20 26 50 30 88 77	

Partition( $A, p, r$ )

1.  $x \leftarrow A[r]$
2.  $i \leftarrow p - 1$
3. **for**  $j \leftarrow p$  **to**  $r - 1$
4.     **if**  $A[j] \leq x$
5.          $i \leftarrow i + 1$
6.         exchange  $A[i] \leftrightarrow A[j]$
7. exchange  $A[i + 1] \leftrightarrow A[r]$
8. **return**  $i + 1$



## Разделянето – Динамична демонстрация

Partition( $A, p, r$ )

1.  $x \leftarrow A[r]$
2.  $i \leftarrow p - 1$
3. **for**  $j \leftarrow p$  **to**  $r - 1$
4.     **if**  $A[j] \leq x$
5.          $i \leftarrow i + 1$
6.         exchange  $A[i] \leftrightarrow A[j]$
7. exchange  $A[i + 1] \leftrightarrow A[r]$
8. **return**  $i + 1$



При приключване на разделянето:

- $x$  се намира на финалното си място.
- вероятно една от областите е пътна.

## Доказване правилността на алгоритъм

За да се докаже, че един алгоритъм работи вярно (правилно) (тоест за всеки валиден вход произвежда желания изход), е нужно да докажем 2 неща:

1. **спиране** – за всеки валиден вход алгоритъмът спира след краен брой действия
2. **правилност на изхода** – за всеки валиден вход, когато алгоритъмът спира, изходът, който той произвежда, е очакваният изход.

Сега ще демонстрираме доказването правилността на алгоритъма Бързо сортиране (Quick Sort). Към края на този учебен материал има още един пример за доказване на правилността (на алгоритъма Сортиране чрез балончета (Bubble Sort)).

---

### Доказване правилността на Бързото сортиране (Quick Sort) – спирането

Твърдение 1:

Quick Sort спира (приключва след крайно време).

Доказателство:

- Partition (разделянето) със сигурност спира и действително извършва краен брой итерации, всяка от които е крайна, и още постоянен брой действия.
- Условието за спиране на Quick Sort е масив с размер 1. При всяко рекурсивно извикване размерът на задачата действително намалява (поне с 1), затова след краен брой етапи от рекурсията (всеки от които е краен) ще стигнем до условието за спиране.

Това, което трябва да се докаже (**Твърдение 1**), е доказано.

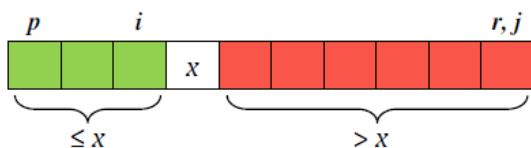
---

### Доказване правилността на Бързото сортиране (Quick Sort) – разделянето

Твърдение 2: При приключване на разделянето:

- $x$  се намира на финалното си място
- от лявата му страна са елементи, които не са по-големи от него
- от дясната му страна са елементи, които са по-големи от него

```
Partition( $A, p, r$ )
1.  $x \leftarrow A[r]$ 
2.  $i \leftarrow p - 1$ 
3. for  $j \leftarrow p$  to  $r - 1$ 
4.   if  $A[j] \leq x$ 
5.      $i \leftarrow i + 1$ 
6.     exchange  $A[i] \leftrightarrow A[j]$ 
7. exchange  $A[i + 1] \leftrightarrow A[r]$ 
8. return  $i + 1$ 
```

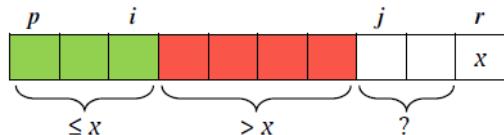


За доказването на Твърдение 2 ще се опитаме и ще докажем свойство, което е изпълнено в началото на всяка итерация на цикъла от 3-ти до 6-ти ред. Това свойство се нарича инвариант на цикъла (loop invariant).

#### Инвариантата на цикъла от 3-ти до 6-ти ред:

В началото на всяка итерация за всеки индекс k в масива:

- ако  $p \leq k \leq i$ , то  $A[k] \leq x$
- ако  $i + 1 \leq k \leq j - 1$ , то  $A[k] > x$
- $A[r] = x$



с 10

---

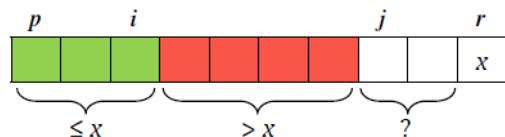
#### Доказване правилността на Бързото сортиране (Quick Sort) – разделянето

#### Инвариантата на цикъла от 3-ти до 6-ти ред:

В началото на всяка итерация за всеки индекс k в масива:

- ако  $p \leq k \leq i$ , то  $A[k] \leq x$
- ако  $i + 1 \leq k \leq j - 1$ , то  $A[k] > x$
- $A[r] = x$

```
Partition( $A, p, r$ )
1.  $x \leftarrow A[r]$ 
2.  $i \leftarrow p - 1$ 
3. for  $j \leftarrow p$  to  $r - 1$ 
4.   if  $A[j] \leq x$ 
5.      $i \leftarrow i + 1$ 
6.     exchange  $A[i] \leftrightarrow A[j]$ 
7. exchange  $A[i + 1] \leftrightarrow A[r]$ 
8. return  $i + 1$ 
```



#### Доказателство на инвариантата на цикъла (чрез индукция върху броя итерации):

**Основа:** В началото на първата итерация (а) и (б) са изпълнени с празно съдържание, а (в) е изпълнено заради ред 1.

**Стъпка:** Ще предположим, че инвариантата на цикъла е съществувала в началото на която и да е итерация m, и ще докажем, че тя съществува също и в началото на итерация m + 1.

Ако  $A[j] > x$ , то движим напред единствено j, затова свойство (б) продължава да е изпълнено, а свойствата (а) и (в) не са засегнати.

Ако  $A[j] \leq x$ , то заради размяната на 6-ти ред все пак  $A[i] \leq x$  (с новото i), затова свойство (а) продължава да е изпълнено и все пак  $A[j-1] > x$  (с новото j), затова свойство (б) продължава да е изпълнено, а свойство (в) не е засегнато.

**Край на цикъла:** В края на последната итерация  $j = r$ , тоест  $A[p \dots i] \leq x$ , а също и  $A[i+1 \dots r-1] > x$  и също  $A[r] = x$ .

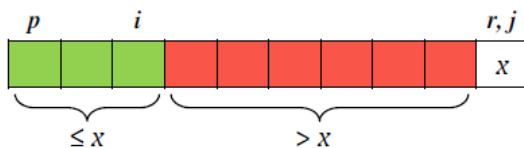
---

### Доказване правилността на Бързото сортиране (Quick Sort) – разделянето

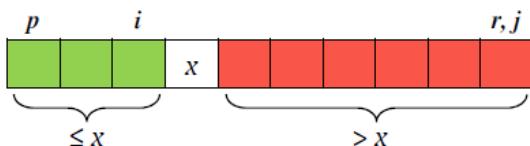
Продължение на доказателството на Твърдение 2

```
Partition( $A, p, r$ )
1.  $x \leftarrow A[r]$ 
2.  $i \leftarrow p - 1$ 
3. for  $j \leftarrow p$  to  $r - 1$ 
4.   if  $A[j] \leq x$ 
5.      $i \leftarrow i + 1$ 
6.     exchange  $A[i] \leftrightarrow A[j]$ 
7. exchange  $A[i + 1] \leftrightarrow A[r]$ 
8. return  $i + 1$ 
```

Както казахме в предишния слайд, от доказателството на инвариантата на цикъла следва, че състоянието при приключването на цикъла е такова:



От размяната на 7-ми ред следва правилността на Твърдение 2:



Това, което трябва да се докаже (Твърдение 2), е доказано.

---

### Доказване правилността на Бързото сортиране (Quick Sort)

Твърдение 3

Quick Sort( $A, 1, n$ ) сортира  $A$ .

```
Quick-Sort ( $A, p, r$ )
1. if  $p < r$ 
2.    $q \leftarrow \text{Partition}(A, p, r)$ 
3.   Quick-Sort ( $A, p, q-1$ )
4.   Quick-Sort ( $A, q+1, r$ )
```

### Доказателство (чрез индукция върху n):

**Основа:** За  $n = 1$  алгоритъмът е верен (зашо?).

**Стъпка:** Нека предположим, че Бързото сортиране (Quick Sort) сортира правилно масиви с размер  $k < n$ .

Според Твърдение 2 при приключването на 2-ри ред за всяко  $p \leq k_1 \leq q-1$  и за всяко  $q+1 \leq k_2 \leq r$  е изпълнено:  $A[k_1] \leq A[q] < A[k_2]$ .

Според предположението за индукцията на 3-ти и 4-ти ред се извършва правилно сортиране на два под-масива:  $A[p \dots q-1]$  и  $A[q+1 \dots r]$ .

От двете гореспоменати изречения следва, че при приключването на Бързото сортиране (Quick Sort) масивът  $A$  е сортиран.

Това, което трябва да се докаже (Твърдение 3), е доказано.

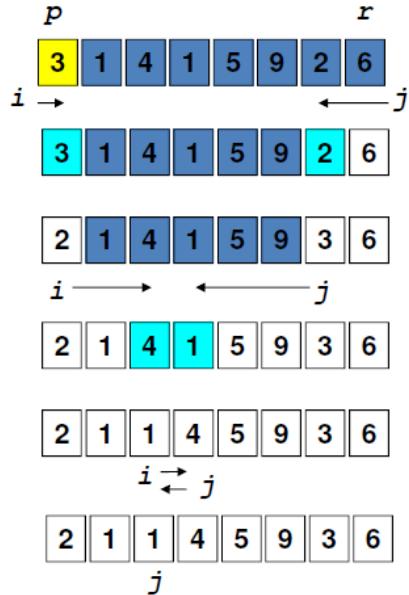
**С това приключи доказателството правилността на Бързото сортиране (Quick Sort), което трябваше да бъде демонстрирано.**

---

### Друга версия на разделянето

Първоначалното (оригиналното) разделяне, което е съставил Hoare:

```
Hoare-Partition ( $A, p, r$ )
1.    $x \leftarrow A[p]$ 
2.    $i \leftarrow p - 1$ 
3.    $j \leftarrow r + 1$ 
4.   while TRUE
5.       repeat  $j \leftarrow j - 1$ 
6.       until  $A[j] \leq x$ 
7.       repeat  $i \leftarrow i + 1$ 
8.       until  $A[i] \geq x$ 
9.       if  $i < j$ 
10.          exchange  $A[i] \leftrightarrow A[j]$ 
11.      else return  $j$ 
```



При приключването на разделянето:

- Дали  $x$  се намира задължително на финалното си място?
- Възможно ли е една от областите да е пътна?
- Каква промяна трябва да се извърши при Бързото сортиране (Quick Sort)?

Вижте задача 7-1 от учебника!

---

## Анализ на времето за изпълнение

Времевата сложност на разделянето (без значение в коя версия) е  $\Theta(n)$ .

- Най-лошият случай

Когато разделянето на всеки етап от рекурсията е небалансирано по най-крайния начин:

- В първата версия на разделянето: едната област е с размер 0, а другата е с размер  $n - 1$
- Във версията на Hoare?

Това се случва, когато осевият елемент (pivot) е минимумът или максимумът на под-масива, на всеки етап.

**Упражнение:** Демонстрирайте изпълнението на Бързото сортиране (Quick Sort) върху сортиран масив и върху обратно сортиран масив!

$$T(n) = T(n-1) + T(0) + \Theta(n) = \Theta(n^2)$$

- Най-добрият случай

Когато разделянето на всеки етап от рекурсията е балансирано: разделяне приблизително на две половини.

$$T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n)$$

Ако разделението е в съотношение  $\alpha : (1 - \alpha)$  за  $0 < \alpha < 1$ ?

.....

## Средно време за изпълнение

- Средно време за изпълнение

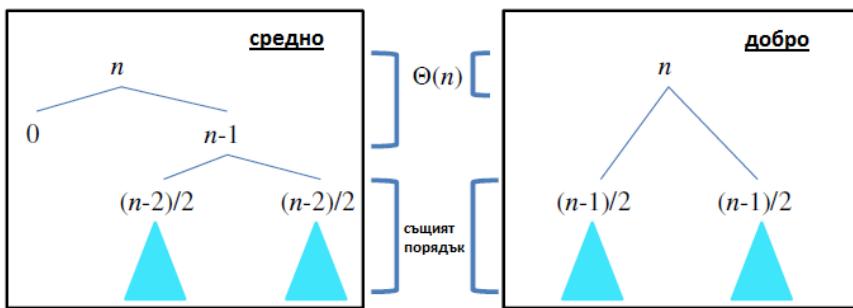
Твърдение: Средното време за изпълнение на Бързото сортиране (Quick Sort) е  $\Theta(n \log n)$ .

Какво е тълкуванието на средното време за изпълнение (или математическото очакване за времето за изпълнение)?

Средната стойност между времената за изпълнение на алгоритъм върху всичките му възможни входни данни с размер  $n$ .

**Няма да докажем гореспоменатото твърдение (има доказателство в учебника), но ще го обясним интуитивно:**

При което и да е изпълнение на Бързо сортиране (Quick Sort) имаме смесица от „лоши“ и „добри“ разделяния. За улеснение нека предположим, че имаме само най-лоши и най-добри разделяния и, че те се появяват редувайки се през едно (тоест лошо, добро, лошо, добро и т.н.). Тогава средното време за изпълнение е равно на времето за изпълнение в добрия случай:



## Случайни алгоритми

- Случайният алгоритъм е алгоритъм, който ползва генератор на случайни числа (random number generator).
- Преимущество на случайния алгоритъм пред детерминантния (не случайния) алгоритъм е имунитетът срещу „злия съперник“, който познава подробностите на алгоритъма и може да произведе резултат, който да представлява лош случай.
- На практика повечето среди за разработка предлагат генератор на псевдо-случайни числа – тоест детерминантен (не случаен) алгоритъм, който връща числа, които статистически изглеждат достатъчно случаини.
- Съществуват два вида случаини алгоритми:
  - **Лас Вегас:** алгоритъмът винаги връща верен резултат, но случайните избори влияят върху времето му за изпълнение и е вероятно, то да е голямо (случайното Бързо сортиране (Quick Sort) е такъв алгоритъм).
  - **Монте Карло:** времето за изпълнение не се влияе от случайните избори, но алгоритъмът е възможно, да върне грешен резултат (вероятността за това е ограничена – най-много 1/2).

## Случайно Бързо сортиране (Quick Sort)

Случайното Бързо сортиране има същите порядъци времена за изпълнение.

Тук можем да говорим също и за средно време за изпълнение, когато средната стойност е върху случаините избори, които извършва алгоритъмът (върху същите входни данни).

Random-Partition ( $A, p, r$ )

1.  $i \leftarrow \text{Random}(p, r)$
2.  $\text{exchange } A[r] \leftrightarrow A[i]$
3. **return** Partition ( $A, p, r$ )

Random-Quick-Sort ( $A, p, r$ )

1. **if**  $p < r$
2.  $q \leftarrow \text{Random-Partition}(A, p, r)$
3. Random-Quick-Sort ( $A, p, q-1$ )
4. Random-Quick-Sort ( $A, q+1, r$ )

Въпрос: Каква е вероятността да се случи най-лошият случай?

Шансът на първия етап да бъде избран минимумът/максимумът за ос (pivot) е  $2/n$ .

На втория етап:  $2/(n-1)$

На  $i$ -тия етап:  $2/(n-i+1)$

Общо шансът за това на всеки етап да бъде избран минимумът/максимумът е  $2^{n-1}/n!$ .

## Стабилност на сортирането

Още едно свойство, по което бихме могли да сравняваме алгоритми за сортиране, е стабилността (stability).

Ще казваме, че едно сортиране е стабилно (stable), ако спазва относителен ред на елементите с еднакви ключове.

3 1 3' 0 5 3'' → 0 1 3 3' 3'' 5

Бързото сортиране (Quick Sort) стабилно ли е?

А Сортирането чрез сливане (Merge Sort)?

А Сортирането чрез балончета (Bubble Sort)?

А Сортирането чрез вкарване (Insertion Sort)?

...

## Ограничаване дълбочината на стека (пълнителя) на рекурсията при Бързото сортиране (Quick Sort)

Както казахме, Бързото сортиране е сортиране на място (in place) – тоест  $\Theta(1)$  елемента се съхраняват във всеки момент извън подавания на входа масив.

Но това е рекурсивен алгоритъм – ресурсите памет, които той изисква, зависят също от стека (пълнителя) на рекурсията.

Каква е дълбочината на рекурсията при Бързото сортиране (Quick Sort):

- В най-лошия случай?
- В най-добрая случай?
- Средно?

Можем да намалим дълбочината на стека (пълнителя) на рекурсията посредством техника, която се нарича „заобикаляне на опашъчната рекурсия“.

- Опашъчната рекурсия е състояние, в което последното действие, което изпълнява един алгоритъм, е рекурсивно извикване.
- Това извикване може да се замени с итерация.

## Заобикаляне на опашъчна рекурсия (Tail Recursion)

### Задача 7-4 от учебника

Ще разгледаме следната версия на Бързото сортиране (Quick Sort), при която второто рекурсивно извикване върху дясната част е заменено с итерация на цикъл:

```
Quick-Sort'(A, p, r)
1.   while  $p < r$ 
2.     do ▶ partition and sort left sub-array
3.        $q \leftarrow \text{Partition}(A, p, r)$ 
4.       Quick-Sort'(A, p, q - 1)
5.        $p \leftarrow q + 1$ 
```

Забележка:

При съвременните компилатори осъществяването на заобикалянето на опашъчната рекурсия (tail recursion) се извършва автоматично.

- а) Обосновете се, защо тази версия работи правилно!
- б) Покажете случай, в който дълбината на стека (пълнителя) на рекурсията е  $\Theta(n)$ .
- в) Предложете промяна в Бързото сортиране' (Quick Sort'), така че дълбината на стека (пълнителя) на рекурсията да бъде ограничена в най-лошия случай до  $\Theta(\log n)$ , без да се засяга времевата сложност на алгоритъма.

## Заобикаляне на опашъчна рекурсия (Tail Recursion)

### Решение

- а) На практика Quick Sort и Quick Sort' вършат едно и също нещо:

И двата алгоритъма започват от същото разделяне, след което имаме рекурсивно извикване върху  $A[p \dots q-1]$ .

След това Quick Sort извършва рекурсивно извикване върху  $A[q+1 \dots r]$ .

Quick Sort' вместо това поставя  $p \leftarrow q + 1$  и след това извиква себе си с  $A[p \dots r]$ .

Така че и двете рутини обработват същите под-масиви и точно в същия ред.

- б) ако всеки път оста (pivot) е максимумът, то Quick Sort' ще извиква себе си всеки път с под-масив, чийто размер намалява с 1. При това положение ще имаме  $\Theta(n)$  на брой рекурсивни извиквания:

Quick-Sort'(A, 1, n)

Quick-Sort'(A, 1, n-1)

Quick-Sort'(A, 1, n-2)

...

Quick-Sort'(A, 1, 1)

Това се случва, когато масивът A е вече сортиран (във възходящ ред).

## Заобикаляне на опашъчна рекурсия (Tail Recursion)

### Решение

в) Идеята е да се извърши рекурсивното извикване на Quick Sort' при всяка итерация с по-малкия под-масив:

```
Quick-Sort''(A, p, r)
1.   while p < r
2.       ▶ partition and sort the small sub-array first
3.       q ← Partition (A, p, r)
4.       if q - p < r - p
5.           Quick-Sort''(A, p, q - 1)
6.           p ← q + 1
7.       else Quick-Sort''(A, q + 1, r)
8.           r ← q - 1
```

Размерът на под-масива при всяко извикване е най-много половината от настоящия размер, и затова броят извиквания (и от тук и дълбочината на стека (пълнителя)) е  $\Theta(\log n)$  в най-лошия случай.

Времето за изпълнение на сортирането не е засегнато, тъй като се извършват именно същите разделяния и се обработват именно същите под-масиви (може би сега единствено в различен ред).

.....

## Допълнителен пример за доказване на правилността – Сортиране чрез балончета (Bubble Sort)

Ще демонстрираме доказване на правилността върху Сортиране чрез балончета:

```
Bubble-Sort(A, n)
1.   for i ← 1 to n
2.       for j ← n downto i+1
3.           if A[j] < A[j-1]
4.               exchange A[j] ↔ A[j-1]
```

1. Спиране: Ясно е, че алгоритъмът спира, тъй като всеки цикъл извършва краен брой итерации и при всяка итерация краен брой действия.

2. Правилност на изходните данни (резултата): Ще формулираме свойства, които са изпълнени в началото на всеки един от циклите (инвариантите на циклите):

**Инвариантата на външния цикъл**: В началото на всяка итерация на for цикъла на редове 1-4 под-масивът  $A[1\dots i-1]$  съдържа най-малките  $i-1$  елемента в  $A$  в сортиран ред.

**Инвариантата на вътрешния цикъл**: В началото на всяка итерация на for цикъла на редове 2-4 елементът  $A[j]$  е минималният елемент в под-масива  $A[j\dots n]$ .

.....

## Правилност на Сортирането чрез балончета (Bubble Sort) – вътрешен цикъл

Bubble-Sort( $A, n$ )

1. **for**  $i \leftarrow 1$  **to**  $n$
2.     **for**  $j \leftarrow n$  **downto**  $i+1$
3.         **if**  $A[j] < A[j-1]$
4.             exchange  $A[j] \leftrightarrow A[j-1]$

**Инвариантата на вътрешния цикъл:** В началото на всяка итерация на for цикъла на редове 2-4 елементът  $A[j]$  е минималният елемент в под-масива  $A[j...n]$ .

**Инициализация:** В началото на първата итерация  $j = n$ , под-масивът  $A[j...n]$  съдържа един единствен елемент, затова съществува инвариантата на цикъла.

**Поддръжка:** Нека предположим, че инвариантата на вътрешния цикъл съществува в началото на която и да е итерация, тоест  $A[j]$  е минималният елемент в под-масива  $A[j...n]$ ; след сравнение с 3-ти ред, ако  $A[j] < A[j-1]$ , се извършва и размяната на 4-ти ред; в края на тази итерация (и в началото на следващата итерация)  $A[j-1]$  е минималният елемент в под-масива  $A[j-1...n]$ , тоест инвариантата на цикъла съществува също и в началото на следващата итерация.

**Приключване:** След извършването на последната итерация  $j = i$ ; елементът  $A[i]$  е минималният елемент в под-масива  $A[i...n]$ .

---

## Правилност на Сортирането чрез балончета (Bubble Sort) – външен цикъл

Bubble-Sort( $A, n$ )

1. **for**  $i \leftarrow 1$  **to**  $n$
2.     **for**  $j \leftarrow n$  **downto**  $i+1$
3.         **if**  $A[j] < A[j-1]$
4.             exchange  $A[j] \leftrightarrow A[j-1]$

**Инвариантата на външния цикъл:** В началото на всяка итерация на for цикъла на редове 1-4 под-масивът  $A[1...i-1]$  съдържа най-малките  $i-1$  елемента в  $A$  в сортиран ред.

**Инициализация:** Преди първата итерация  $i = 1$ , а под-масивът  $A[1...i-1]$  е празен, затова инвариантата на цикъла съществува празна.

**Поддръжка:** Нека предположим, че инвариантата на външния цикъл съществува в началото на  $i$ -тата итерация, тоест под-масивът  $A[1...i-1]$  съдържа най-малките  $i-1$  елемента в сортиран ред; както видяхме вътрешният цикъл прехвърля минималния елемент на под-масива  $A[i...n]$  в  $A[i]$ ; затова в края на  $i$ -тата итерация (в началото на итерацията  $i+1$ ) под-масивът  $A[1...i]$  съдържа най-малките  $i$  елемента в сортиран ред, тоест инвариантата на цикъла съществува също и преди  $(i+1)$ -тата итерация.

**Приключване:** След извършването на итерацията  $n-1$ , под-масивът  $A[1...n-1]$  съдържа най-малките  $n-1$  елемента в сортиран ред; но тогава  $A[n]$  е най-големият елемент, затова целият масив  $A[1...n]$  сега вече е сортиран.

**Това, което трябва да се докаже, е доказано.**

---

### Въпроси за преговор

1. За масив, чиито елементи са различни един от друг, коя стойност на  $q$  връща Partition, когато осевият елемент (pivot)  $A[r]$  е максимумът? А когато е минимумът?
  2. Демонстрирайте изпълнението на Бързото сортиране (Quick Sort) върху сортиран масив, за който всички елементи са различни един от друг! Каква е времевата сложност на изпълнението в този случай?
  3. Уверете се, че разбирате следното: при Бързото сортиране (Quick Sort) разделянето на масива на всеки етап е динамично, тоест: на всеки етап е възможно различно разделяне, за разлика от Сортирането чрез сливане (Merge Sort), при което разделянето е статично (винаги на две половини).
- 

### Отговори на въпросите за преговор

1. Когато осевият елемент е максимума, Partition ще върне стойността  $n$  (като  $n$  е размера на масива). Когато осевият елемент е минимума, Partition ще върне стойност 1.
  2. На всеки етап ще се извършва разделяне на лява област с размер  $n-1$  и празна дясна област. Сложността е  $\Theta(n^2)$ .
- 

### Упражнения

---

#### Препоръчителни упражнения от учебника

##### Глава 7

7.1-2	7.2-2	задача 7-1
7.1-3	7.2-3	задача 7-3
7.1-4	7.3-2	задача 7-4

с 30

---

#### Допълнителни упражнения

1. Нека променим ред 1 от алгоритъма Бързо сортиране (Quick Sort) на: **if**  $r-p \geq k$   
Тоест, не сортираме под-масиви с размер  $k$  или по-малки.

В такъв случай казваме, че изходните данни (резултатът) от Бързото сортиране (Quick Sort) е масив „почти сортиран с грешка с размер  $k$ “.

Анализирайте времевата сложност на този алгоритъм в най-лошия и в най-добрия случай в зависимост от  $n$  и  $k$ !

2. Демонстрирайте изпълнението на Бързото сортиране (Quick Sort) върху масива A[1...n], който е сортиран в обратен ред и чиито елементи са различни един от друг! Каква е сложността на времето за изпълнение в този случай?

3. Демонстрирайте изпълнението на Partition и изпълнението на Hoare-Partition върху масив, чиито ключове (всички) са равни един на друг!

Какви са времената за изпълнение на Бързото сортиране (Quick Sort) върху масив, чиито ключове (всички) са равни, при всяка една от версията на разделянето?

.....

### Решение 1

Най-лошия случай:  $\Theta(n^2 - k^2)$ .

Най-добраия случай:  $\Theta(n \log(n/k))$ .

### Решение 2

На първия етап осевият елемент (pivot) е минимумът: първият и последният елемент се разменят и продължаваме с под-масива A[2...n].

На втория етап осевият елемент (pivot) е максимумът: разделянето не променя нищо и продължаваме с под-масива A[2...n-1].

Третият етап е подобен на първия, четвъртия на втория и т. н.

Най-общо казано, на всеки етап се получава разделение, при което една от областите му е празна.

Времето за изпълнение е като в най-лошия случай –  $\Theta(n^2)$ .

.....

## Тема 5

# Приоритетни опашки и сортиране чрез купчина Priority Queues and Heap Sort

В програмата:

Глава 6 от учебника:

Ще се запознаем със структурата от данни купчина (heap) и ще разгледаме две нейни важни приложения:

- Приоритетна опашка (Priority Queue)
  - Алгоритъма Сортиране чрез купчина (Heap Sort)
- .....

### Приоритетна опашка (Priority Queue)

Приоритетната опашка е абстрактен тип данни (ADT), определен от следните действия:

- Create(PQ, L) – създаване на приоритетна опашка от даден списък с елементи L
- Insert(PQ, x) – вкарване на елемента, към който сочи x
- Max(PQ) – връщане на елемента с максималния ключ
- Del-Max(PQ) – изтриване на елемента с максималния ключ
- Change-Priority(PQ, x, k) – промяна на ключа на елемента, към който сочи x, на k

Това е разширение на опашка с тази разлика, че тук изваждането е според приоритетите, а не според времето на престой. На ключовете на елементите обикновено се дава значение на приоритет или предпочтение.

Приоритетните опашки имат множество разнообразни приложения в практиката, например:

- Симулация на събития по оста на времето (ключът: времето на събитието)
- Времево планиране на задачи (извършване на операции върху процесор, достъп до мрежа и т. н.)

Коя структури от данни осъществяват ефективно приоритетна опашка?

.....

### Приоритетна опашка (Priority Queue)

- Create(PQ, L) – създаване на приоритетна опашка от даден списък с елементи L
- Insert(PQ, x) – вкарване на елемента, към който сочи x
- Max(PQ) – връщане на елемента с максималния ключ
- Del-Max(PQ) – изтриване на елемента с максималния ключ
- Change-Priority(PQ, x, k) – промяна на ключа на елемента, към който сочи x, на k

### Осъществяване на приоритетна опашка чрез масив и чрез сортиран масив

- $m$  е размерът на началния списък  $L$
- $n$  е броят на елементите в даден момент

Времева сложност	Масив	Сортиран масив
Create	$\Theta(m)$	$\Theta(m \log m)$ **
Insert	$\Theta(1)$	$\Theta(n)$
Max	$\Theta(n)$ *	$\Theta(1)$
Del-Max	$\Theta(n)$	$\Theta(1)$
Change-Priority	$\Theta(1)$	$\Theta(n)$

\* – може да се подобри на  $\Theta(1)$ , за сметка на Change-Priority

\*\* – ако не е известно нищо за елементите на входните данни, не може да се сортира за по-малко от това време (ще бъде доказано по-нататък в курса)

### Въпрос (съдържа се във въпросите за преговор):

Каква е сложността при реализация чрез свързан списък и сортиран свързан списък?

## Приоритетна опашка (Priority Queue)

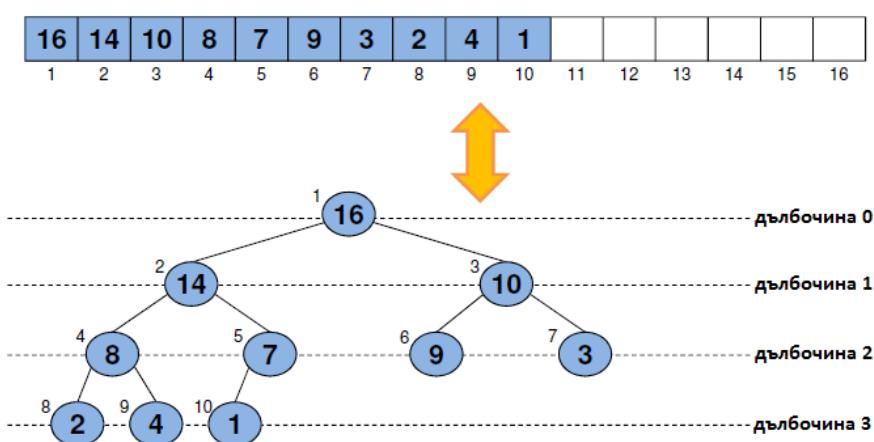
Сега ще покажем ефективна структура от данни за приоритетна опашка, която се нарича **купчина (heap)**.

Това ще е нещо като компромис между обикновен масив и сортиран масив.

Преимущество на реализациата чрез купчина (heap): **Нито едно действие, освен Create разбира се, няма да изиска линейно време!**

## Купчина (Heap)

Структурата от данни **купчина** е на практика **масив**, който представя почти пълно бинарно дърво.



Освен масива поддържаме променлива на име `heapSize`, която пази броя на елементите в купчината.

- В примера `heapSize = 10`.

При даден елемент с индекс  $i$  можем да изчислим индекса на децата и на родителя му:

**Parent( $i$ )**  
1. return  $\lfloor i/2 \rfloor$

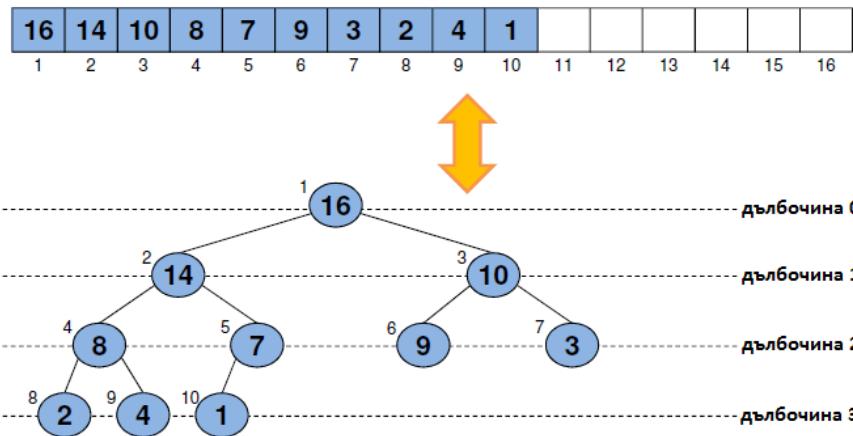
**Left( $i$ )**  
1. return  $2i$

**Right( $i$ )**  
1. return  $2i+1$

### Максимална купчина (Max Heap)

При **максималната купчина** е изпълнено:

Ако  $x$  е наследник на  $y$ , то  $\text{key}[x] \leq \text{key}[y]$ .



При **минималната купчина** е изпълнено: Ако  $x$  е наследник на  $y$ , то  $\text{key}[x] \geq \text{key}[y]$ .

От сега нататък, ако не е отбелоязано друго, ще говорим за максимални купчини.

### Свойства на купчината (heap)

Какъв е броят на елементите в купчина, чиято височина е  $h$ ?

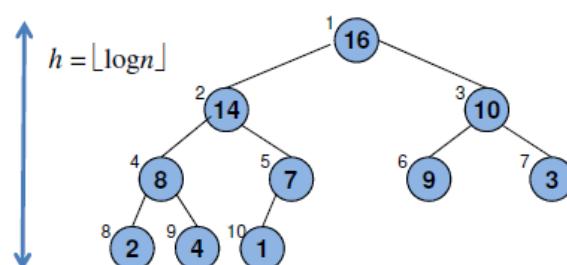
$$2^h \leq n \leq 2^{h+1} - 1$$

Каква е височината на купчина с размер  $n$  елемента?

От една страна:  $h \leq \log n$

От друга страна:  $h \geq \log(n+1) - 1 > \log n - 1$

И затова  $h = \lfloor \log n \rfloor = \Theta(\log n)$



При кои индекси се намират листата и колко листа има? Колко вътрешни кръстовища има?

Листата се намират на индекси  $\lfloor n/2 + 1 \rfloor$  до  $n$ .

Има  $\lceil n/2 \rceil$  листа и  $\lfloor n/2 \rfloor$  вътрешни кръстовища.

Упражнение: Докажете го чрез индукция върху  $n$ !

Кръстовище с индекс  $i$  е листо, само и единствено ако  $2i > n$ .

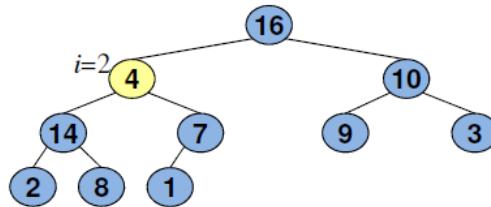
### Поправителни действия върху купчина

Сега ще се запознаем с две основни действия върху купчина.

Тяхната цел: Корекция на купчина, в която има някакво нарушение на свойствата на купчината. По-нататък ще ни служат при други действия.

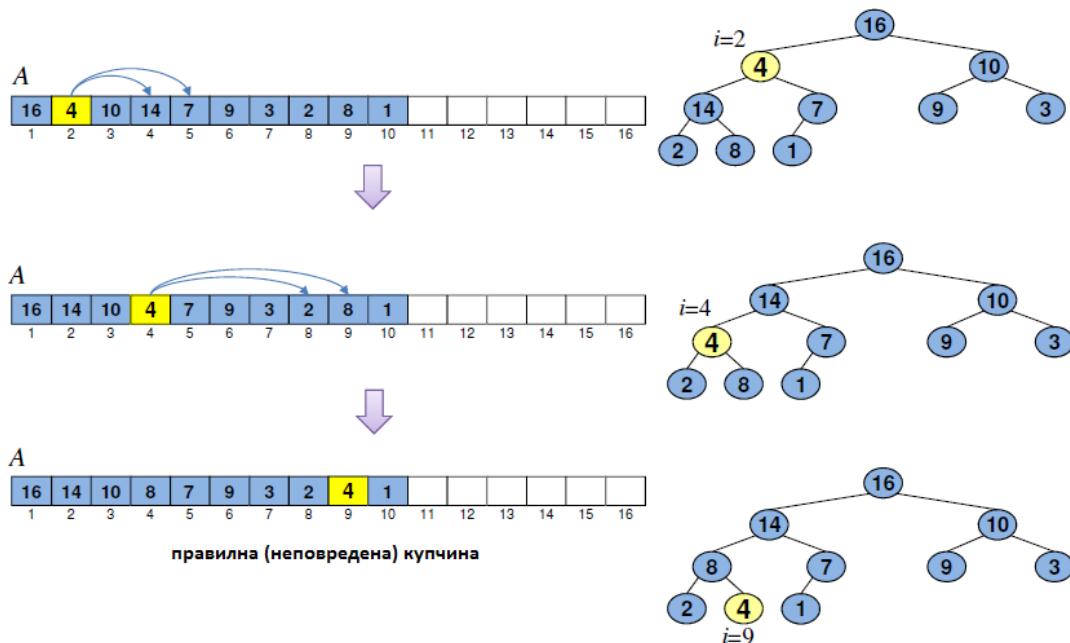
Действието Heapify Down:

- Получава масив  $A$  и индекс  $i$ .
- Предполага, че лявото и дясното под-дърво на  $i$  са правилни (неповредени) купчини.
- Възможно е,  $A[i]$  да е „нарушител“: по-малък от (поне един от) наследниците си
- Плъзва „нарушителя“ надолу чрез размяната му с максималния му наследник отново и отново.



### Поправителни действия върху купчина – Heapify Down

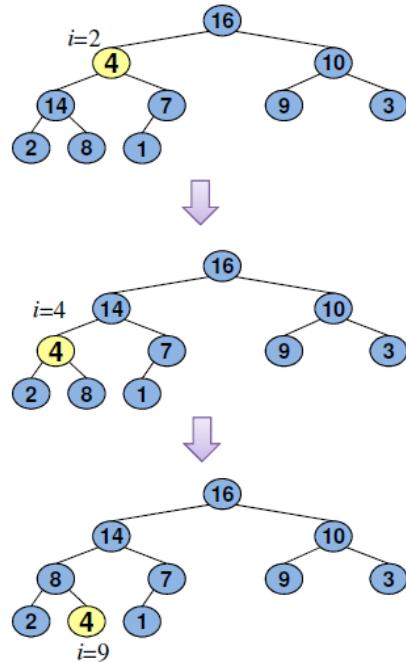
Пример: Heapify-Down( $A$ , 2)



### Поправителни действия върху купчини – Heapify Down

Heapify-Down( $A, i$ )

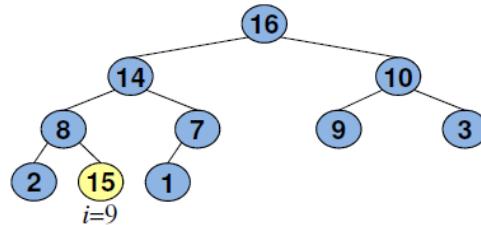
1.  $l \leftarrow \text{Left}(i), r \leftarrow \text{Right}(i), n \leftarrow \text{heapSize}[A]$
2.  $largest \leftarrow i$
3. **if**  $l \leq n$  **and**  $A[l] > A[i]$  **then**  $largest \leftarrow l$
4. **if**  $r \leq n$  **and**  $A[r] > A[largest]$  **then**  $largest \leftarrow r$
5. **if**  $largest \neq i$
6.     exchange  $A[i] \leftrightarrow A[largest]$
7.     Heapify-Down( $A, largest$ )



### Поправителни действия върху купчини – Heapify Up

Действието Heapify-Up:

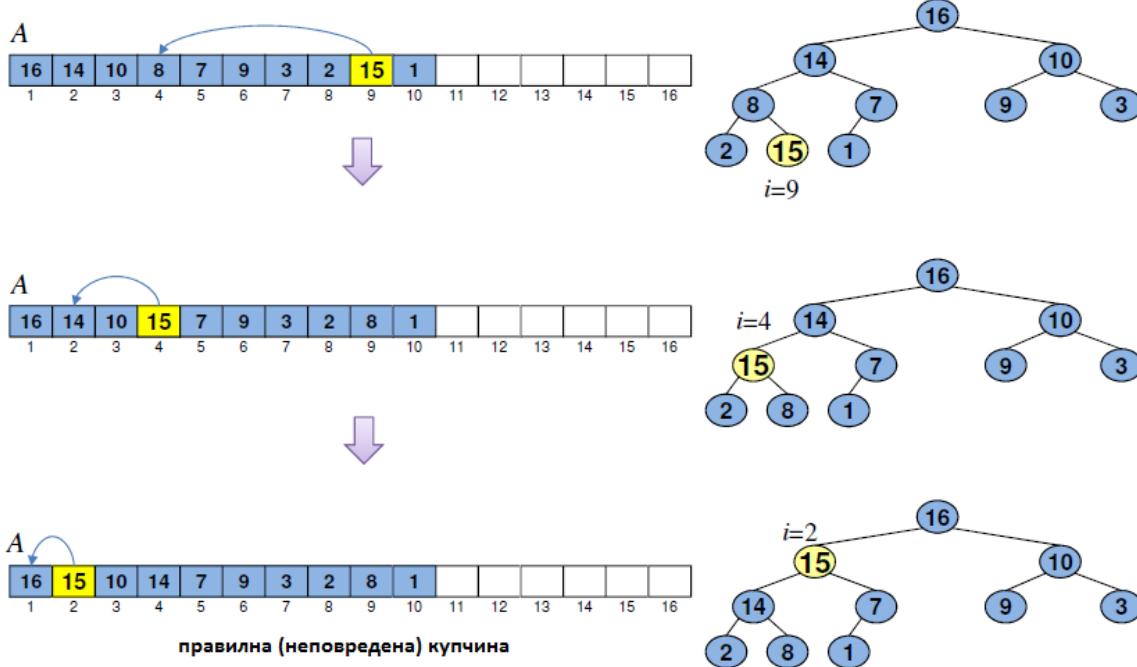
- Получава масив  $A$  и индекс  $i$ .
- Предполага, че  $A$  е правилна (неповредена) купчина, освен може би „нарушителя“  $A[i]$ , който е по-голям от родителя си.
- Плъзва „нарушителя“ надолу чрез размяната му с родителя му отново и отново.



Упражнение: Осъществете Heapify Up чрез псевдо код.

## Поправителни действия върху купчини – Heapify Up

Пример: Heapify-Up(A, 9)



## Поправителни действия върху купчини – анализ на сложността

### Времева сложност

В най-лошия случай от гледна точка броя на размените:

- Heapify-Down( $A, i$ ) се изпълнява за време  $\Theta(h(i))$ , като  $h(i)$  е височината на кръстовището  $i$
- Heapify-Up( $A, i$ ) се изпълнява за време  $\Theta(d(i))$ , като  $d(i)$  е дълбочината на кръстовището  $i$

В най-лошия случай от гледна точка на  $i$ :

Ако задействаме Heapify-Down върху корена или Heapify-Up върху листо, то времето за изпълнение ще бъде  $\Theta(\log n)$ .

### Сложност от гледна точка на допълнителната памет

Heapify-Down изисква  $\Theta(h(i))$  допълнителна памет за рекурсията.

Но можем да осъществим итеративна версия (без рекурсия) и тогава допълнителната памет ще е  $\Theta(1)$ .

## Действия върху купчини

До този момент разглеждахме Heapify-Down и Heapify-Up.

Сега ще се обърнем към осъществяването на действия свързани с приоритетните опашки и ще си помогнем с гореспоменатите две поправителни действия.

- Create(PQ, L) – създаване на приоритетна опашка от даден списък с елементи L
  - Insert(PQ, x) – вкарване на елемента, към който сочи x
  - Max(PQ) – връщане на елемента с максималния ключ
  - Del-Max(PQ) – изтриване на елемента с максималния ключ
  - Change-Priority(PQ, x, k) – промяна на кючка на елемента, към който сочи x, на k
- 

### Действия върху купчини – изграждане

Изграждане на купчина от даден списък с елементи

Входни данни: някакъв масив A с n на брой елемента.

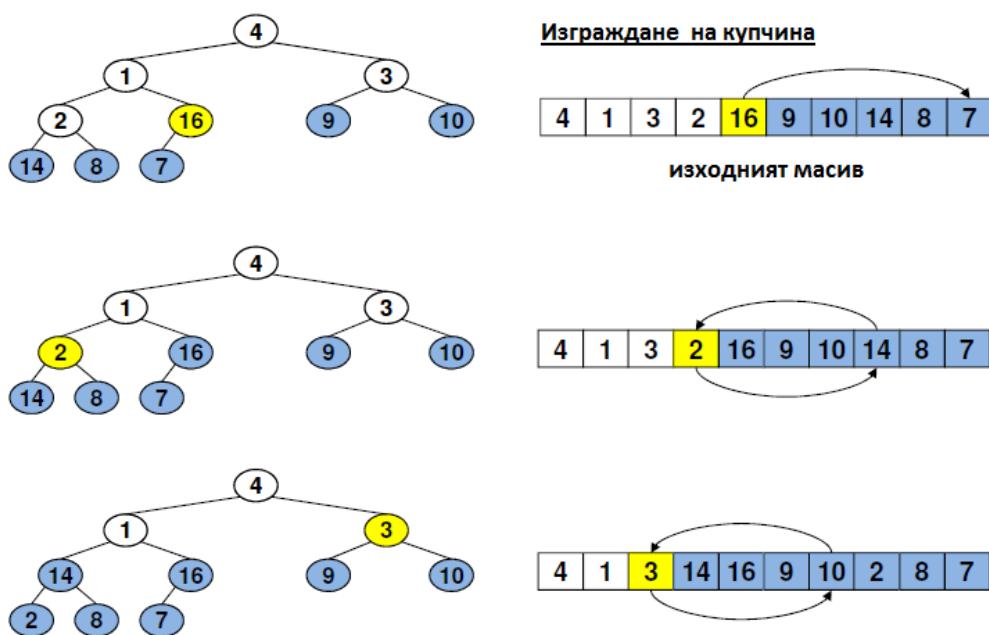
Изходни данни: същите елементи, подредени в A като валидна купчина.

Алгоритъмът: Обхождаме всички вътрешни кръстовища отдолу нагоре (bottom up) и задействаме върху тях Heapify-Down.

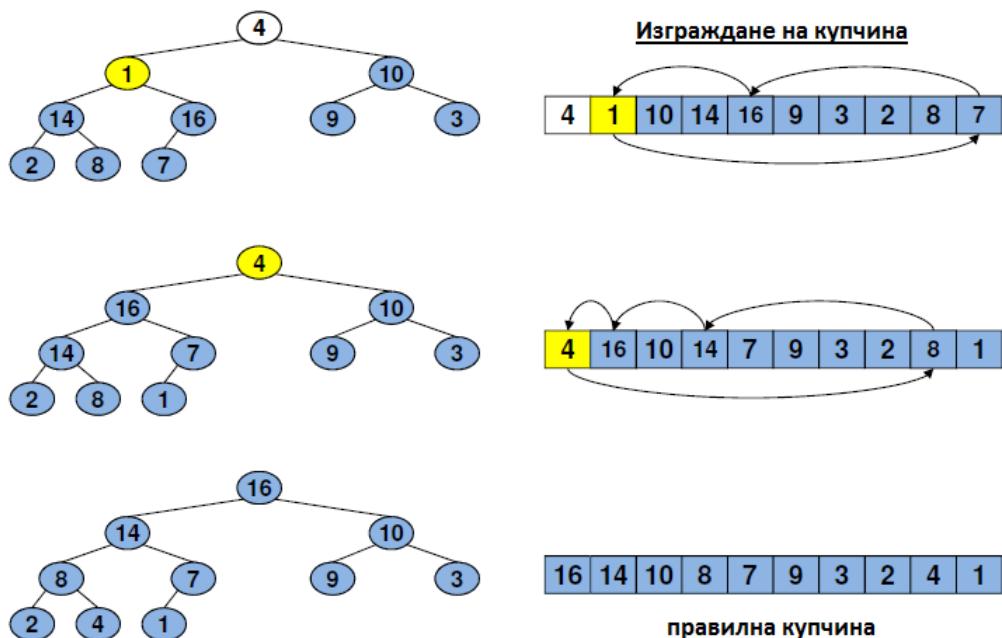
Build-Heap( $A, n$ )

1.  $heapSize[A] \leftarrow n$
2. **for**  $i \leftarrow \lfloor heapSize[A]/2 \rfloor$  **downto** 1
3.     Heapify-Down( $A, i$ )

### Действия върху купчини – изграждане



### Действия върху купчини – изграждане



### Действия върху купчини – изграждане

Build-Heap( $A, n$ )

1.  $heapSize[A] \leftarrow n$
2. **for**  $i \leftarrow \lfloor heapSize[A]/2 \rfloor$  **downto** 1
3.     Heapify-Down( $A, i$ )

- Защо обхождаме единствено вътрешните кръстовища?
- Защо отдолу нагоре (bottom up)?

### Анализ на времената за изпълнение – изграждане на купчина

#### Груб анализ

$\lfloor n/2 \rfloor$  извиквания на Heapify-Down всяко с  $O(\log n)$ . Общо  $O(n \log n)$ .

#### По- внимателен анализ

Сега ще докажем, че времето за изпълнение е  $\Theta(n)$ .

Интуиция: има много на брой „евтини“ извиквания и малко на брой „скъпи“ извиквания.

Ще преброим колко размени (exchanges) се извършват (за пълно време за изпълнение трябва да се умножи по константа).

- Има най-много  $n/2$  листа,  $n/4$  родители на листа,  $n/8$  прародители на листа и т. н. (пропуснахме най-високите гранични стойности)
- На едно кръстовище с височина  $u$  се извършват най-много  $u$  размени.
- Затова общият брой размени е най-много:

$$\begin{aligned} & \frac{n}{4} \cdot 1 + \frac{n}{8} \cdot 2 + \dots + \frac{n}{2^{i+1}} \cdot i + \dots + 1 \cdot \lfloor \log n \rfloor \\ &= (\frac{n}{4} + \frac{n}{8} + \frac{n}{16} + \dots + 1) \leq n/2 \\ &+ (\frac{n}{8} + \frac{n}{16} + \dots + 1) \leq n/4 \\ &+ (\frac{n}{16} + \dots + 1) \leq n/8 \\ &\dots \\ &+ 1 \leq 1 \end{aligned}$$

$$\sum_{i=1}^{\lfloor \log n \rfloor} \frac{i}{2^{i+1}} \leq 1$$

с 20



### Действия върху купчини – изтриване на максимум

#### Изтриване на максимум

Идеята:

- Ще прехвърлим към корена последното листо.
- Ще намалим размера на купчината `heapSize` с 1.
- Ще извикаме `Heapify-Down` върху корена, за да поправим възможно нарушение.

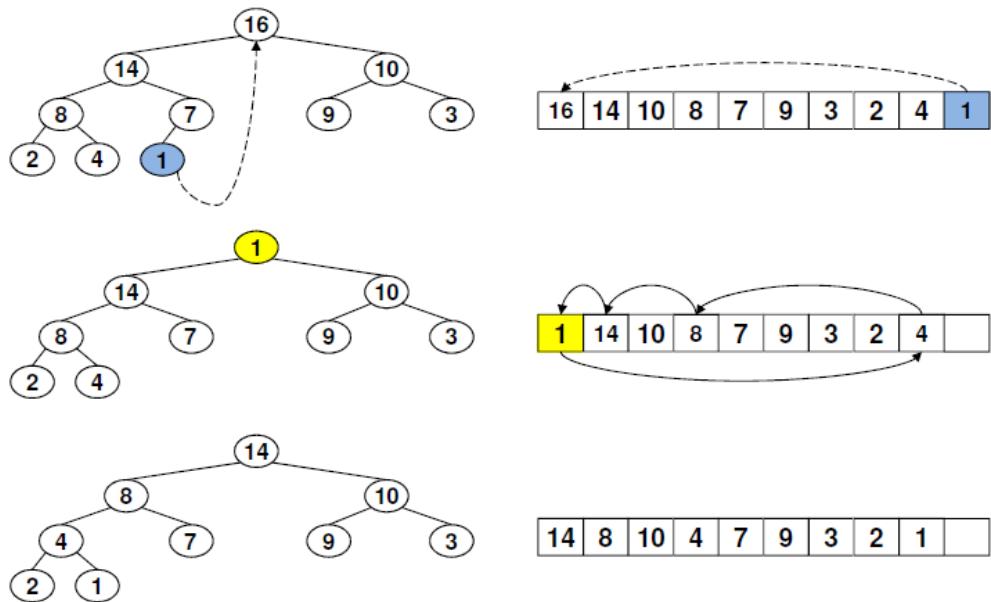
```
Heap-Extract-Max(A)
1. if heapSize[A] < 1
2.     error "heap underflow"
3. max ← A[1]
4. A[1] ← A[heapSize[A]]
5. heapSize[A] ← heapSize[A] - 1
6. Heapify-Down(A, 1)
7. return max
```

Сложността в най-лошия случай:  $\Theta(\log n)$



## Действия върху купчини – изтриване на максимум

### Изтриване на максимум



## Действия върху купчини – вкарване

### Вкарване

Идеята:

- Ще добавим новия елемент като следващо листо.
- Ще увеличим размера на купчината  $heapSize$  с 1.
- Ще извикаме **Heapify-Up** върху новото листо.

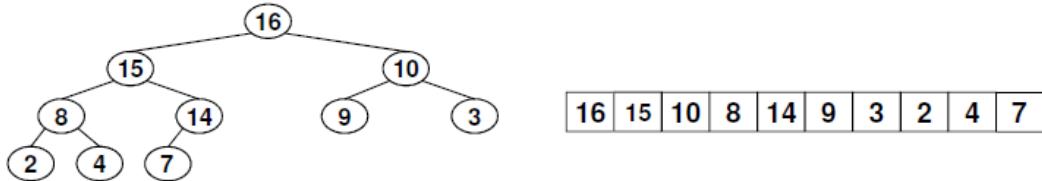
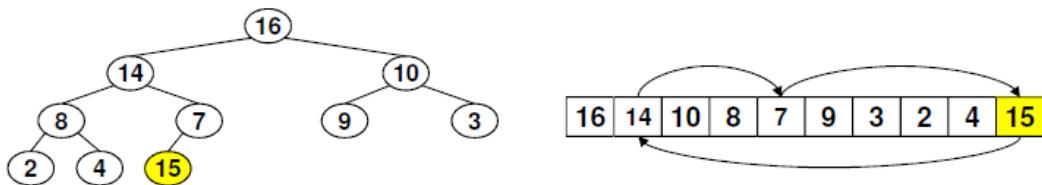
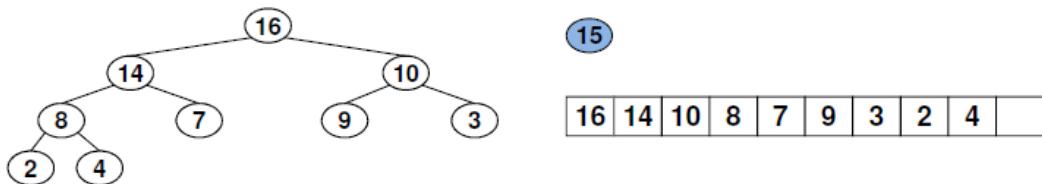
```
Heap-Insert( $A$ ,  $key$ )
1.  $heapSize[A] \leftarrow heapSize[A] + 1$ 
2.  $A[heapSize[A]] \leftarrow key$ 
3.  $\text{Heapify-Up}(A, heapSize[A])$ 
```

Сложността в най-лошия случай:  $\Theta(\log n)$

Забележка: Ако ни е даден показалец сочещ към елемента  $x$ , който искаме да вкарваме (който би могъл да има множество полета), то вкарваме показалеца в купчината по подобен начин, като извършим поправката според ключа на  $x$ .

## Действия върху купчини – вкаране

### Вкаране



## Действия върху купчини – вкаране

### Задача 6-1 от учебника

При даден масив  $A$  с размер  $n$ , от него можем да изградим купчина по начин, различен от този, който видяхме, чрез вкарането на елементите в купчината един след друг:

Build-Heap'(A, n)

1.  $heapSize[A] \leftarrow 1$
2. **for**  $i \leftarrow 2$  **to**  $n$
3.     Heap-Insert( $A$ ,  $A[i]$ )

(a) Демонстрирайте изграждане на купчина по този начин върху масива  $[1, 2, 3]$  и сравнете с получаващата се чрез Build-Heap купчина!

(б) Дали този подход е ефективен от гледна точка на времето за изпълнение?

## Действия върху купчини – промяна на ключ

### Промяна на ключ

Идеята:

- След промяната на ключа може да се получи нарушение нагоре или надолу.
- Затова ще поправим нагоре или надолу.

Heap-Change-Key( $A, i, key$ )

1.  $A[i] \leftarrow key$
2. Heapify-Down( $A, i$ )
3. Heapify-Up( $A, i$ )

Сложността в най-лошия случай:  $\Theta(\log n)$

- Дали е възможно, след промяната на ключа да има нарушение и нагоре и надолу?
- 

### Сортиране чрез купчина (Heap Sort)

Допълнително приложение на купчините е алгоритъмът за сортиране на купчина (Heap Sort).

Heap-Sort( $A, n$ )

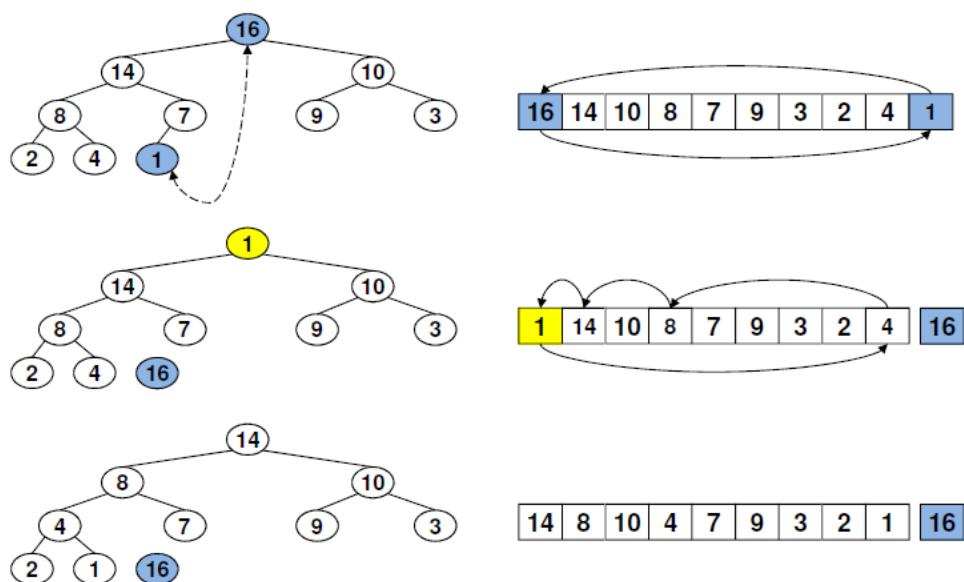
1. Build-Heap( $A$ )
2. **for**  $i \leftarrow n$  **downto** 2
3.     exchange  $A[1] \leftrightarrow A[i]$
4.      $heapSize[A] \leftarrow heapSize[A] - 1$
5.     Heapify-Down( $A, 1$ )

етапът на изграждането на купчината

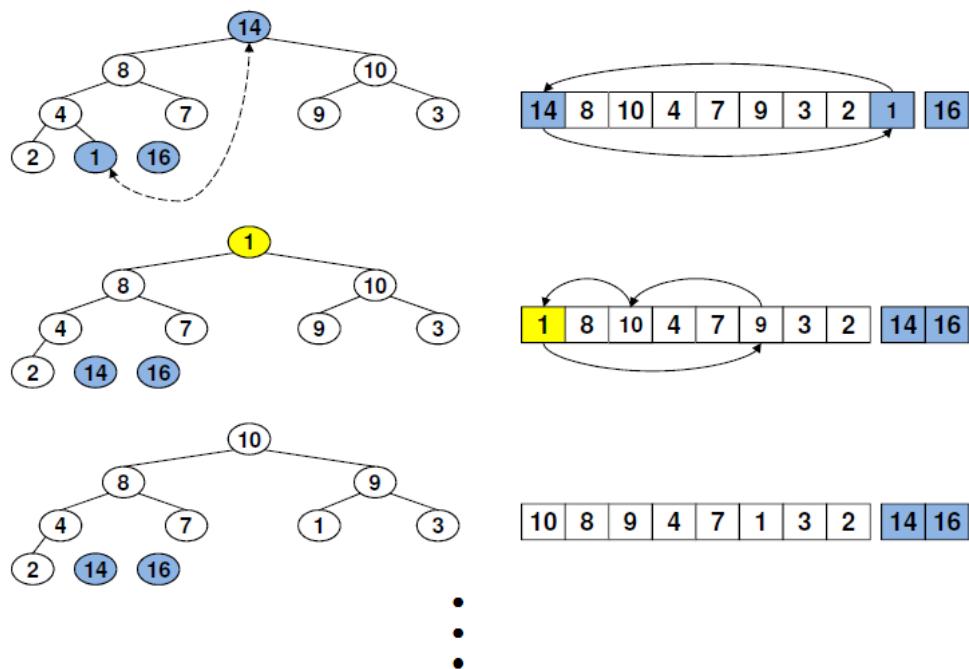
етапът на подреждането на елементите

### Сортиране чрез купчина (Heap Sort)

Демонстрация на етапа на подреждането на елементите (купчината вече е изградена):



### Сортиране чрез купчина (Heap Sort)



с 30

### Сортиране чрез купчина (Heap Sort)

Heap-Sort( $A, n$ )

1. Build-Heap( $A$ )
2. **for**  $i \leftarrow n$  **downto** 2
3.     exchange  $A[1] \leftrightarrow A[i]$
4.      $heapSize[A] \leftarrow heapSize[A] - 1$
5.     Heapify-Down( $A, 1$ )

$\Theta(n)$

$\Theta(\log k) \quad 1 \leq k \leq n-1$

Сложността в най-лошия случай:

$$\Theta(n) + \sum_{k=1}^{n-1} \Theta(\log k) = \Theta(n) + \Theta(\log n!) = \Theta(n) + \Theta(n \log n) = \Theta(n \log n)$$

Свойства на Купчинното сортиране (Heap Sort):

- сортира масив  $A$  с размер  $n$  за време  $\Theta(n \log n)$  в най-лошия случай.
- сортира на място (in place) и изиска  $\Theta(1)$  допълнителна памет (извън тази за масива от входните данни) (при условие, че Heapify Down се имплементира без рекурсия).

Дали това е стабилно сортиране?

### Въпроси за преговор

1. Приоритетна опашка може да се имплементира чрез свързан списък и чрез сортиран свързан списък. Каква е сложността на всяко едно от действията свързани с приоритетната опашка във всяка една от тези имплементации?
2. Максимална купчина ли е следният масив?  $< 23, 17, 14, 6, 13, 10, 1, 5, 7, 12 >$
3. Минимална купчина ли е един сортиран масив?
4. В една максимална купчина максималният елемент се намира на индекс 1, а вторият по големина елемент се намира на индекс 2 или 3. Дадена е максимална купчина с размер 15, с различни един от друг елементи.
  - (а) На кои индекси може да се намира 3-ят по големина елемент?
  - (б) На кои индекси може да се намира 4-ят по големина елемент?
5. На кои индекси може да се намира минимумът на максимална купчина с размер  $n$ ? Предположете, че всички елементи са различни един от друг!
6. Какво е времето за изпълнение на търсенето на даден елемент в купчина?
7. Покажете чрез пример, че купчинното сортиране не е стабилно сортиране!

### Отговори на въпросите за преговор

1. Действията могат да се имплементират със следните сложности (уверете се, че разбирате как):

	Свързан списък	Сортиран свързан списък
Create	$\Theta(n)$	$\Theta(n \log n)$
Insert	$\Theta(1)$	$\Theta(n)$
Max	$\Theta(n)$	$\Theta(1)$
Del-Max	$\Theta(n)$	$\Theta(1)$
Change-Priority	$\Theta(1)$	$\Theta(n)$

2. Не. Ключ 6 (шест) нарушива свойствата на купчината спрямо десния си наследник.
3. Да. Всяко кръстовище изпълнява свойството на минималната купчина.
4. (а) на индекси от 2 до 7 (дълбочина 1 или 2)  
(б) на индекси от 2 до 15 (дълбочина от 1 до 3)
5. Минимумът трябва да е листо, тоест на индекси от  $\lfloor n / 2 + 1 \rfloor$  до  $n$ .

6. Търсенето в купчина изисква в най-лошия случай линейно време. Това може да се докаже, например, с помощта на отговора на предишния въпрос: Ако търсим минимума, трябва да обходим  $\Theta(n)$  кръстовища.
  7. Например, купчинното сортиране върху  $<1, 1'>$  не запазва относителния ред между 1 и 1'.
- 

## Упражнения

---

### Препоръчителни упражнения от учебника

Глава 6	6.1-6	6.2-4	6.4-2	6.5-8
6.1-1	6.1-7	6.2-5	6.4-3	задача 6-1
6.1-2	6.2-1	6.3-1	6.5-1	
6.1-4	6.2-2	6.3-2	6.5-6	
6.1-5	6.2-3	6.4-1	6.5-7	

---

### Допълнителни упражнения

1. Какво е времето за изпълнение на Build Heap върху масив, сортиран в обратен ред?
  2. Имплементирайте действието  $Del(A, i)$ , което изтрива от максималната купчина A с размер n кръстовището с индекс i, с времева сложност  $\Theta(\log n)$ !
  3. Опишете структура от данни, позволяваща извършването на следните действия с нужната сложност:
    - $Init(S)$  – инициализация на структурата при дадена редица от елементи S с дължина m за време  $O(m)$ .
    - $Insert(x)$  – добавяне на x към структурата за време  $O(\log n)$  ( $n$  е настоящия брой елементи).
    - $Find-min$  – връщане на минимума за време  $O(1)$ .
    - $Find-max$  – връщане на максимума за време  $O(1)$ .
    - $Del-min$  – изваждане на минимума от структурата за време  $O(\log n)$ .
    - $Del-max$  – изваждане на максимума от структурата за време  $O(\log n)$ .
  4. В даден колеж преподават  $n$  на брой лектори. Данните на лекторите (име, адрес, възнаграждение) са запазени във файл. Искаме да отпечатаме имената на онези  $m$  на брой лектори, които получават най-високото възнаграждение.  
Дайте ефективно решение при следните ограничения:
    - Позволена сложност на допълнителната памет:  $O(m)$ . Дадено:  $m = o(n)$ .
    - Позволено е да се обходят данните от файла само веднъж.
-

### Решение 1

При всяка една от  $\lfloor n / 2 \rfloor$ -те итерации рутината Heapify Down ще приключи при първото извикване. Времето за изпълнение все още е  $\Theta(n)$ , но положителната константа във времето за изпълнение е по-малка.

---

### Решение 2

Ще преместим последното листо на индекс  $i$  и ще коригираме.

```
Del(A, i)
1.  $n \leftarrow \text{heapSize}[A]$ 
2.  $A[i] \leftarrow A[n]$ 
3.  $\text{heapSize}[A] \leftarrow \text{heapSize}[A] - 1$ 
4. Heapify-Down(A, i)
5. Heapify-Up(A, i)
```

Ще обърнем внимание, че е възможно нарушение в посока нагоре или надолу.

Можете ли да дадете пример за всеки един случай?

---

### Решение 3

Ще запазим елементите в две купчини едновременно: **минимална купчина** и **максимална купчина**. За всеки елемент ще запазим също и индекса на „близнака“ му от другата купчина.



На следващия слайд – осъществяването на действията

---

**Init(S):** Ще копираме елементите на S в два отделни масива. На този етап „близнаците“ се намират на един и същи индекс, затова за всеки елемент ще инициализираме индекса на „близнака“ му като индекс на самия него.

Ще изпълним Build Heap веднъж за минималната купчина и веднъж за максималната купчина. Всяко преместване на елемент ще причини актуализация на индекса, който е запазен при брат му „близнак“ (добавяне на константа към сложността).

Време:  $2 \cdot \Theta(m)$  за копирането и още  $2 \cdot \Theta(m)$  за изграждане на купчината. Общо  $\Theta(m)$ .

**Find-min / Find-max:** връщане на корена на съответната купчина. Време:  $\Theta(1)$ .

**Insert(x):** Ще вкараме x във всяка една от двете купчини. Отново при всяко преместване на елемент в една от купчините ще актуализираме съответно индекса, който е запазен при брат му „близнак“, в другата купчина.

Време: всяко вкарване:  $\Theta(\log n)$ , общо:  $\Theta(\log n)$ .

**Del-min:** Ще изтрием корена на минималната купчина и с помощта на индексите ще стигнем до „близнака“, след което ще изтрием и него от максималната купчина. Изтриването на корена ще се извърши както обикновено, а изтриването на „близнака“ ще се извърши, както видяхме в предишното упражнение. Имплементацията на Del-max е симетрична на Del-min.

Време: всяко изтриване:  $\Theta(\log n)$ , общо:  $\Theta(\log n)$ .

**Забележка:** Кръстовището близнак на корена на едната купчината е листо в другата купчина.

с 40

#### Решение 4

Ще използваме минимална купчина с размер m.

1. Ще копираме първите m елемента от файла в масив и ще инициализираме от него минимална купчина, като ключът ще бъде възнаграждението, а името и адресът ще бъдат придружаващи данни.
2. Сега ще обходим останалите лектори във файла започвайки от m+1, и за всеки лектор x: ако възнаграждението му е по-голямо от това на лектора в корена на купчината, ще извършим:
  - 2.1. Heap-Extract-Min
  - 2.2. Heap-Insert(x)
3. На края ще отпечатаме елементите на купчината.

### Сложност

Допълнителна памет: Използваме купчина с размер  $m$ , тоест  $\Theta(m)$ .

Време:

- Етапи 1 и 3 се изпълняват за  $\Theta(m)$ .
- На етап 2 се извършват  $n$ -т действия, като всяко едно от тях в най-лошия случай е съставено от две действия, за  $\Theta(\log m)$ .
- Общо:  $\Theta(m) + \Theta((n - m)\log m) = \Theta(n\log m)$ .

Въпрос: Какво щеше да се промени, ако искахме да отпечатаме имената сортирани по размера на възнаграждението?

.....

## Тема 6

### Долна граница на сортирането, линейни сортирания Lower bound to sorting, linear sorts

#### В програмата:

Глава 8 от учебника:

Ще отговорим на въпроса „Колко бързо може да се сортира?“

Ще учит, какво е долна граница за задача и ще намерим долна граница на задачата със сортирането.

Ще се запознаем с няколко линейни сортирания:

- Сортиране чрез преброяване (Counting Sort)
- Базисно сортиране (Radix Sort)
- Сортиране чрез кофа (Bucket Sort)

#### Колко бързо може да се сортира?

#### Напомняне относно определението за сортиране:

- Входни данни: поредица от елементи  $(a_1, a_2, \dots, a_n)$ , върху които е зададен някакъв ред.
- Изходни данни: перmutация  $(a'_1, a'_2, \dots, a'_n)$  от тези елементи, така че  $a'_1 \leq a'_2 \leq \dots \leq a'_n$ .

#### Алгоритъм за сортиране:

- Решение на задачата за сортирането, тоест: поредица от действия, която от всеки вход произвежда сортирания изход.  
Например: Insertion Sort, Bubble Sort, Merge Sort, Heap Sort, Quick Sort, ...
- Ще обърнем внимание, че нито едно от гореспоменатите не се изпълнява в най-лошия случай за  $O(n \log n)$ . Дали това е случай?



- с други думи: дали е известен алгоритъм, който сортира  $n$  елемента за време  $O(n \log n)$ ?
- и ако не – дали някой би могъл да измисли такъв алгоритъм един ден?
- и ако не – как да докажем, че нито един алгоритъм (известен или все още неоткрит) не може да направи това?

#### Долна граница на Сортирането чрез сравнения

##### Теорема:

Всеки алгоритъм за сортиране чрез сравнения върху поредица с дължина  $n$  извършва в най-лошия случай  $\Omega(n \log n)$  сравнения между елементите на входните данни.

### Алгоритъм за сортиране чрез сравнения (Comparison Sorting Algorithm):

Сортиране, при което решението за определянето на сортирания ред се базират единствено на сравненията между елементите на входните данни.

При дадени два елемента  $a$  и  $b$  можем да извършим всяка една от следните проверки:

$$a < b, \quad a \leq b, \quad a = b, \quad a \geq b, \quad a > b$$

Определянето на реда не се базира на други действия, като аритметични действия върху елементи (и дори въобще не предполагаме, че елементите са числа).

Всички сортирания, които разглеждахме до сега в курса, са сортирания чрез сравнения.

Извод: Сортиране чрез сравнения не може да се изпълни за  $O(n \log n)$ .

Затова Сортирането чрез кучина (Heap Sort) и Сортирането чрез сливане (Merge Sort) са оптимални сортирания от гледна точка на времевата сложност в най-лошия случай.

---

### Доказателство на долната граница – Модел на Дървото за взимане на решения

Сортиранията чрез сравнения могат да се описват по опростен начин чрез дървета за взимане на решения (decision trees).

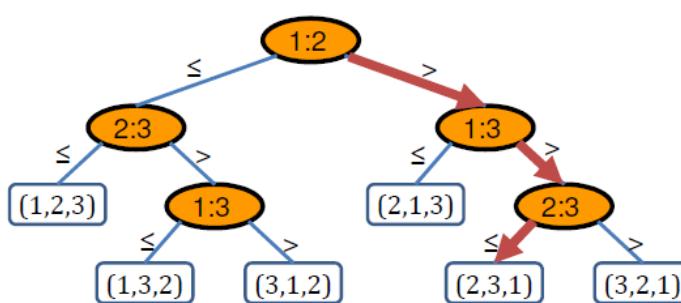
На всеки алгоритъм за сортиране чрез сравнения и на всеки размер на вход  $n$  съответства едно дърво за взимане на решения.

Пример: дърво за взимане на решения за Сортиране чрез вкарване (Insertion Sort) и  $n = 3$ .



Маршрут съответстващ на входните данни

$a_1$	$a_2$	$a_3$
9	3	7



- Изпълнението на един алгоритъм започва от корена и завършва при едно от листата.
- Всяко вътрешно кръстовище се отбелязва с  $i : j$  и представя сравнение, което алгоритъмът редактира между  $a_i$  и  $a_j$ .  
Ако  $a_i \leq a_j$ , продължението на изпълнението на алгоритъма се представя чрез завиване наляво в дървото, а ако е обратно, надясно.
- Всяко листо представя стойност на входните данни  $\langle \pi(1), \pi(2), \dots, \pi(n) \rangle$  и, когато достигнем листото, имаме изпълнено  $a_{\pi(1)} \leq a_{\pi(2)} \leq \dots \leq a_{\pi(n)}$

## Доказателство на долната граница – Модел на Дървото за взимане на решения

Броят на сравненията, които извършва един алгоритъм в най-лошия случай, се представя от най-дългия маршрут от корена до което и да е листо, тоест от височината на дървото.

Щом е така, ще намерим долната граница на височината на дървото (ще я отбележим с  $h$ ):

- Колко листа най-малко има в дървото за взимане на решения за сортиране на  $n$  елемента?  $\rightarrow$  поне  $n!$
- Колко листа най-много има в едно бинарно дърво с височина  $h$ ?  
 $\rightarrow$  най-много  $2^h$

И затова:  $n! \leq$  броят на листата  $\leq 2^h$

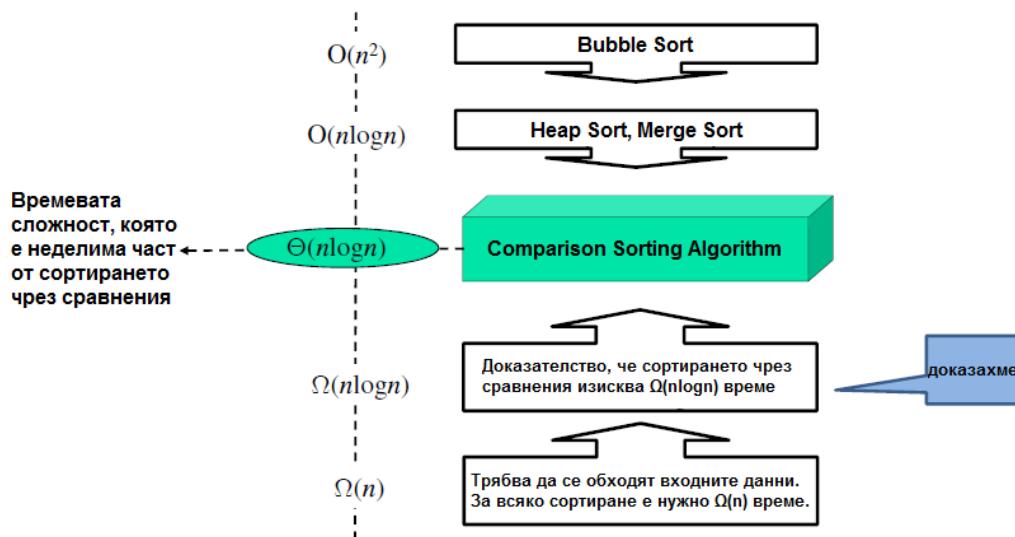
Тоест:  $h \geq \log_2(n!)$

И получаваме:  $h = \Omega(n \log n)$

С това доказваме, че всяко сортиране чрез сравнения върху входни данни с размер  $n$  извършва в най-лошия случай  $\Omega(n \log n)$  сравнения, и затова времето му за изпълнение е  $\Omega(n \log n)$ .

Забележка: Дървото за взимане на решения се отнася само и единствено до сравнителните действия между елементите на входните данни, които извършва алгоритъмът. Както изглежда, съществуват и други действия, като размяна на елементи, преместване напред на индекси чрез цикли и т. н.

## Горни и долнни граници на сортирането чрез сравнения

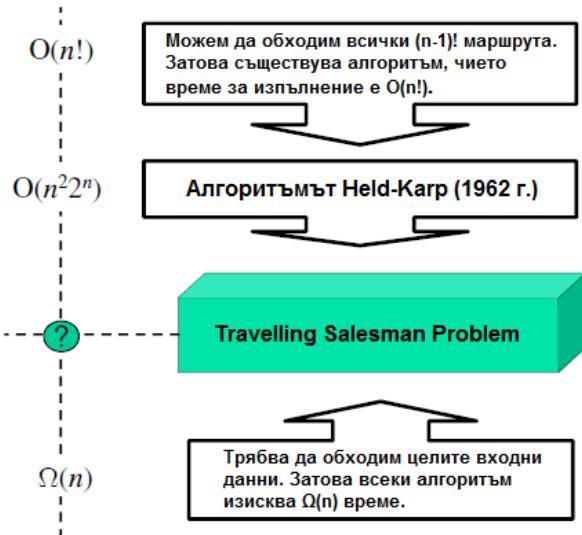


- Откриването на алгоритъм задоволява доказателството на горната граница на задачата (тя може да се реши чрез „не повече от ...“).
- За разлика от това при доказателството на долната граница на задачата (тя не може да се реши чрез „не повече от ...“) трябва да се вземат предвид всички алгоритми, които я решават (съществуващи и все още неоткрити!).

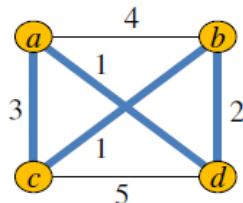
## Алгоритмични разлики

В много случаи има „алгоритмични разлики“ между най-ниската горна граница и най-високата добра граница, които са известни за дадена задача.

Пример: Задачата за пътуващия продавач (Travelling Salesman Problem).



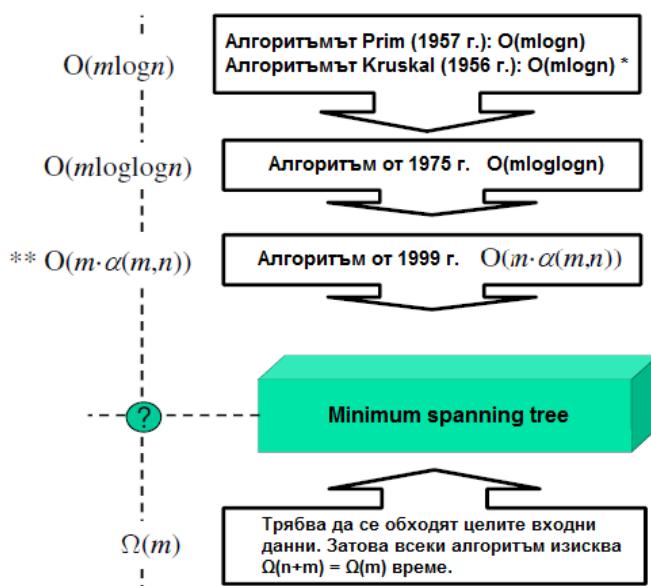
Дадени са  $n$  града и разстоянията между тях. Трябва да се изчисли най-краткия кръгов маршрут, който минава през всички градове.



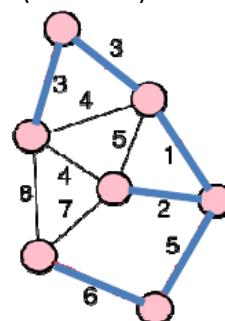
Открыт въпрос:  
Съществува ли алгоритъм, който се изпълнява за време  $O(c^n)$  за  $c < 2$ ?

## Алгоритмични разлики

Допълнителен пример: Задачата за минимално разпростиращото се дърво (Minimum spanning tree).



Дадени са  $n$  компютъра и между тях  $m$  кабела. Коя е най-кратката кабелна мрежа, която оставя всички свързани? ( $m \geq n - 1$ )



Открыт въпрос:  
Съществува ли алгоритъм, който се изпълнява за време  $O(m)$ ?

\* Определена версия на алгоритъма Крускал (Kruskal) се изпълнява за време  $O(n^2)$ , и ако  $m = \Theta(n^2)$ , той е по-ефективен.

\*\*  $\alpha(m, n)$  е обратна функция на Акерман, „почти константна“ функция.

## Линейни сортирания

Сега ще се запознаем с три сортирания, които при определени хипотези се изпълняват за линейно време в най-лошия случай:

- Сортиране чрез преброяване (Counting Sort)
- Базисно сортиране (Radix Sort)
- Сортиране чрез кофа (Bucket Sort)

Разбира се, тези сортирания не са сортирания чрез сравнения и извършват върху елементите на входните данни и допълнителни действия освен сравнения. Затова долната граница, която доказвахме, не важи за тях.

с 10

### Сортиране чрез преброяване (Counting Sort)

- Предполага, че входните данни са масив A с размер n, чийто ключове са цели числа от областта [1 ... k].
- Използва масив B с размер n, така сякаш той пише изходните данни, и помощен масив C с размер k.

A	1	2	3	4	5	6	7	8	9	10	11
	7	1	3	1	2	4	5	3	2	4	3

входен масив (n = 11)

C	1	2	3	4	5	6	7	8	9	10

помощен масив (k = 10)

1. **for**  $i \leftarrow 1$  **to**  $k$
2.     **do**  $C[i] \leftarrow 0$
3. **for**  $j \leftarrow 1$  **to**  $n$
4.     **do**  $C[A[j]] \leftarrow C[A[j]] + 1$
5. ►  $C[i]$  now contains the number of elements =  $i$ .
6. **for**  $i \leftarrow 2$  **to**  $k$
7.     **do**  $C[i] \leftarrow C[i] + C[i-1]$
8. ►  $C[i]$  now contains the number of elements  $\leq i$ .

C	1	2	3	4	5	6	7	8	9	10
	0	0	0	0	0	0	0	0	0	0

#### Етап 1 - зануляване

C	1	2	3	4	5	6	7	8	9	10
	0	0	0	0	0	0	0	0	0	0

#### Етап 2 - преброяване

C	1	2	3	4	5	6	7	8	9	10
	2	2	3	2	1	0	1	0	0	0

#### Етап 3 - натрупване

C	1	2	3	4	5	6	7	8	9	10
	2	4	7	9	10	10	11	11	11	11

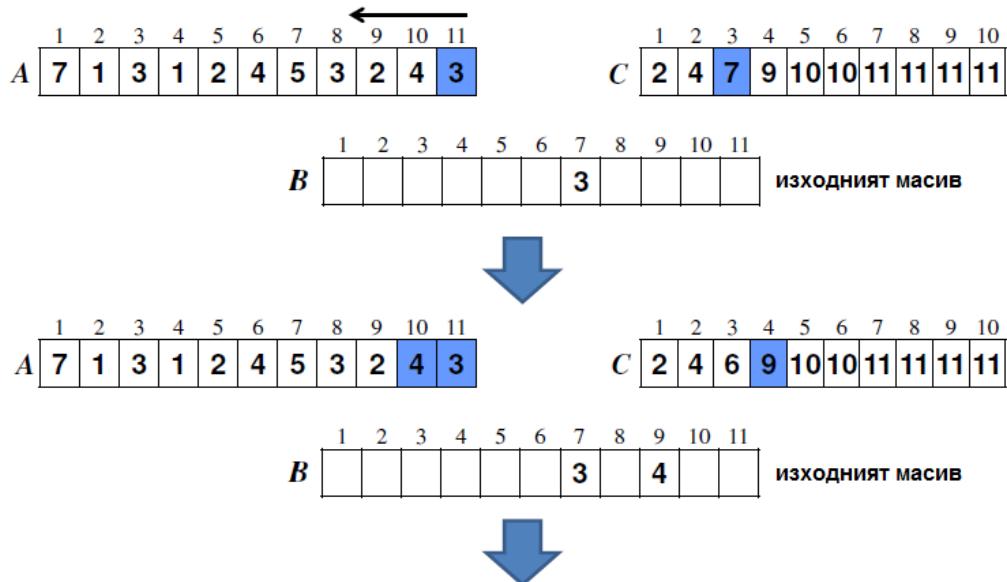
### Сортиране чрез преброяване (Counting Sort)

```

9. for  $j \leftarrow n$  downto 1
10.   do  $B[C[A[j]]] \leftarrow A[j]$ 
11.    $C[A[j]] \leftarrow C[A[j]] - 1$ 

```

Етап 4 - произвеждане на резултата



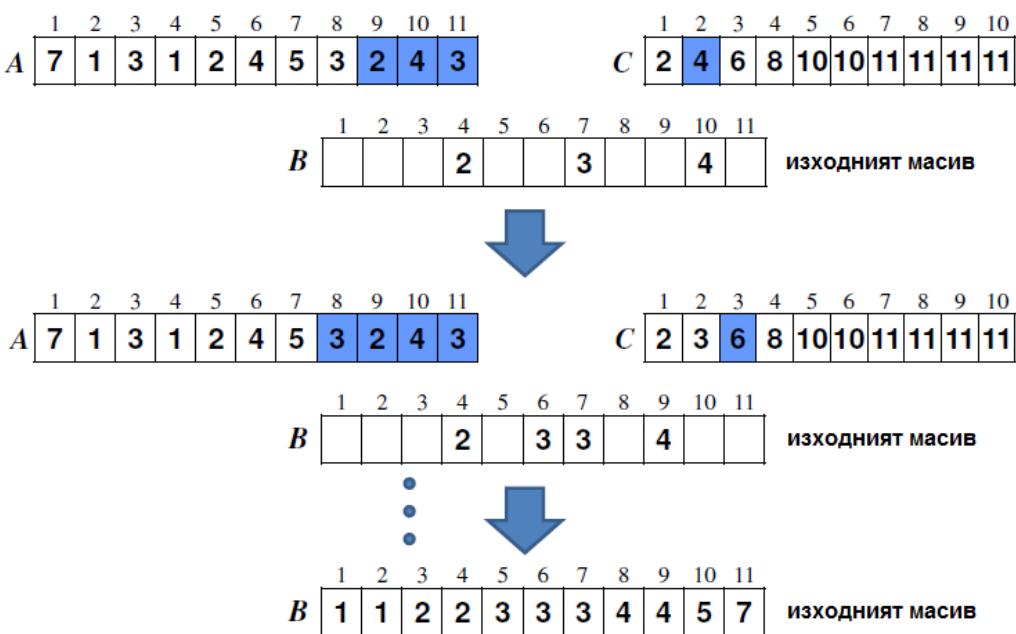
### Сортиране чрез преброяване (Counting Sort)

```

9. for  $j \leftarrow n$  downto 1
10.   do  $B[C[A[j]]] \leftarrow A[j]$ 
11.    $C[A[j]] \leftarrow C[A[j]] - 1$ 

```

Етап 4 - произвеждане на резултата



## Сортиране чрез пребояване (Counting Sort)

- Изпълнява се за време  $\Theta(n + k)$ .  
Когато  $k = O(n)$ , времето за изпълнение е  $\Theta(n)$ .
- Сложността на допълнителната памет е  $\Theta(n + k)$ .
- Въобще не извършва сравнения между елементите на входните данни!

```
Counting-Sort ( $A, B, k$ )  
1. for  $i \leftarrow 1$  to  $k$   
2.      $C[i] \leftarrow 0$   
3. for  $j \leftarrow 1$  to  $n$   
4.      $C[A[j]] \leftarrow C[A[j]] + 1$   
5. ►  $C[i]$  now contains  $| \{ \text{elements} = i \}|$   
6. for  $i \leftarrow 2$  to  $k$   
7.      $C[i] \leftarrow C[i] + C[i-1]$   
8. ►  $C[i]$  now contains  $| \{ \text{elements} \leq i \}|$   
9. for  $j \leftarrow n$  downto 1  
10.    $B[C[A[j]]] \leftarrow A[j]$   
11.    $C[A[j]] \leftarrow C[A[j]] - 1$ 
```

Въпрос: Защо обхождаме входните данни в обратен ред на етапа на произвеждането на резултата?

Въпрос: След втория етап (пребояването) вече е възможно да се произведе резултата (знаем колко пъти имаме всяко число). Защо не го правим?

.....

## Базисно сортиране (Radix Sort)

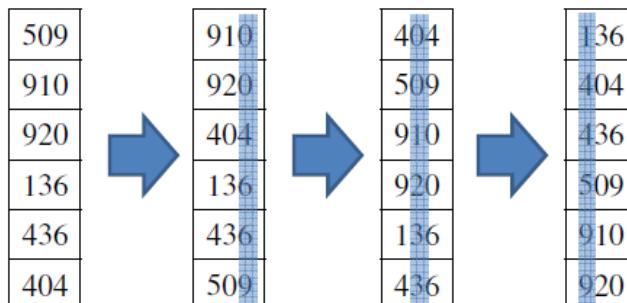
- Предполага, че входните данни са масив  $A$  с размер  $n$ , чиито ключове са елементи, дадени за представяне като цели числа от  $d$  на брой цифри с основа някакво  $b$ .

Например: Двоична основа (0-1), десетична основа (0-9), шестнайсетична основа (0-9, A-F), някаква азбучна основа (A-Z).

### Алгоритъмът

Сортира на етапи:

- Започвайки от най-малко смислената цифра (LSD) до най-много смислената цифра (MSD).
- На всеки етап задейства стабилно сортиране (например Сортирането чрез пребояване).



## Базисно сортиране (Radix Sort)

### Въпроси:

- Защо се изисква на всеки етап стабилно сортиране?
- Защо да се сортира от LSD към MSD, а не обратно?

### Времева сложност

Съществуват  $d$  етапа.

Ако на всеки етап използваме Сортирането чрез пребояване, сложността на всеки етап ще бъде  $\Theta(n + b)$ .

Общо:  $\Theta(d(n + b))$ .

Когато  $d$  е константа и  $b = O(n)$ , Базисното сортиране (Radix Sort) се изпълнява за линейно време.

## Базисно сортиране (Radix Sort)

### Малко история

Базисното сортиране е било алгоритъма, използван в машините за сортиране на картони от края на 19-ти век до 70-те години на 20-ти век.

Входните данни на картоно-сортировачните машини са били картони за дупчене, които е измислил Херман Холерит (Herman Hollerith) през 1890 г. за американската Служба за пребояване на населението:

- Няколко колони, като във всяка от тях има няколко стойности за цифри и букви
- Всяка „клетка“ може да бъде продупчена или не

Картоно-сортировачната машина е проверявала всеки път по една колона и е разпределяла картоните по клетките по продупчената клетка.

Сортиране по няколко колони се прави по метода Базисно сортиране: повторно сортиране по колони, от по-малко значимата колона към по-значимата колона.

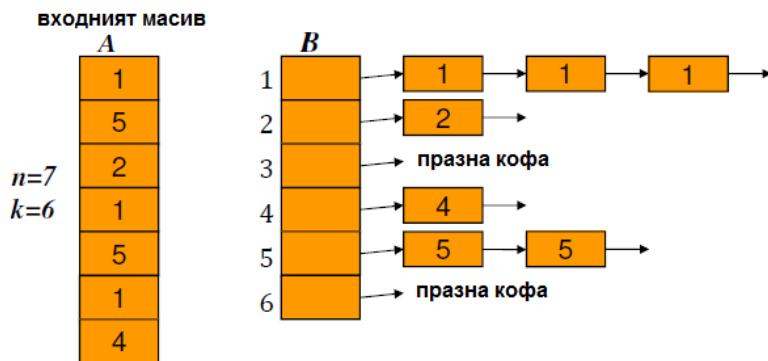
Холерит създава свой собствен бизнес, който през 1924 г. се превръща в IBM.

## Сортиране чрез кофа (Bucket Sort)

- Предполага, че входните данни са масив A с размер n, чиито ключове са цели числа между 1 и k.

### Идеята:

Етап 1: Ще разпръснем  $n$ -те ключа измежду  $k$  „кофи“ (всяка кофа е свързан списък).



Етап 2: Ще свържем кофите във верига.



- Как можем да се погрижим, сортирането да е стабилно?

Като вкарваме елементи в края на списъците (за това е нужен също и показалец към края на всеки списък).

---

### Сортиране чрез кофа (Bucket Sort)

Bucket-Sort( $A, n$ )

1. **for**  $i \leftarrow 1$  **to**  $n$
2.     insert  $A[i]$  into list  $B[A[i]]$
3. concatenate the lists  $B[1], B[2], \dots, B[n]$  together in order

### Времева сложност

- Разпръскването на елементите по кофите отнема общо  $\Theta(n)$  (вкарване в списък за  $\Theta(1)$ ).
- Връзването във верига на списъка отнема  $\Theta(n + k)$  без показалци към краишата на списъците или  $\Theta(k)$  с показалци към краишата на списъците.

Общо:  $\Theta(n + k)$ .

---

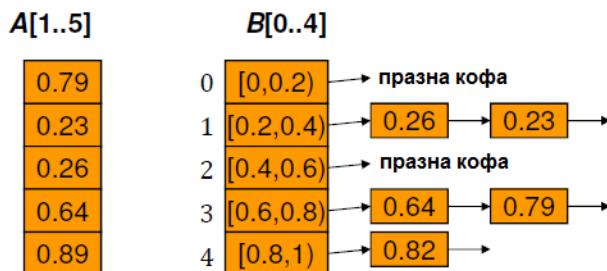
### Разширено сортиране чрез кофа (Bucket Sort)

- Предполага, че входните данни са масив  $A$  с размер  $n$ , съставен от реални числа в интервала  $[0, 1]$ , които притежават еднакво (хомогенно) статистическо разпределение в този интервал.

- Еднакво (хомогенно) разпределение: шансът едно число да се съдържа в дадена подобласт е пропорционално на дължината ѝ, например, шансът едни входни данни да се съдържат в подобластта  $[1/2, 3/4]$  е  $\frac{1}{4}$ .
- Ако числата се разпределят еднакво в областта  $[a, b]$  за които и да е  $a < b$ , можем да ги „нормализираме“. За  $x \in [a, b]$  ще използваме формулата  $(x - a)/(b - a)$ .

### Идеята

- Ще разпърснем  $n$ -те ключа измежду  $n$  „кофи“, като всяка една от тях покрива подобласт, чийто размер е  $1/n$  (всяка кофа е свързан списък).



- Ще сортираме всяка кофа и ще свържем сортираниите кофи във верига.

с 20

### Разширеното сортиране чрез кофа (Bucket Sort)

Bucket-Sort( $A, n$ )

1. **for**  $i \leftarrow 1$  **to**  $n$
2.     insert  $A[i]$  into list  $B[\lfloor nA[i] \rfloor]$
3. **for**  $i \leftarrow 0$  **to**  $n-1$
4.     sort list  $B[i]$  using (e.g.) insertion sort
5. concatenate the lists  $B[0], B[1], \dots, B[n-1]$  together in order

Понеже статистическото разпределение на ключовете е хомогенно, очакваме да намерим константен брой ключове във всяка кофа, и затова сортиранието на всяка кофа ще изисква  $\Theta(1)$  време.

Средната времева сложност на разширеното Сортиране чрез кофа е  $\Theta(n)$ .  
(доказателство в учебника, страници 146-147)

### Въпроси за преговор

1. Долната граница на сортиранието чрез сравнения, която доказвахме, се отнася до най-лошия случай. В най-добрания случай може, разбира се, да се сортира за време  $O(n \log n)$ . Покажете алгоритъм за сортиране чрез сравнения, който се изпълнява за линейно време в най-добрания случай!

2. При доказването на долната граница на сортирането чрез сравнения отбелаяхме, че в дървото за взимане на решения, което съответства на входни данни с размер  $n$ , има поне  $n!$  листа. Опитайте се да помислите, защо може да има повече от  $n!$  на брой листа в дървото за взимане на решения и, кога се случва това?
  3. Изразено чрез  $\Theta$  каква е височината на дървото за взимане на решения при Сортирането чрез балончета (Bubble Sort) за входни данни с размер  $n$ ? А при Сортирането чрез купчина (Heap Sort)?
  4. Каква е времевата сложност в най-лошия случай при разширеното Сортиране чрез кофа (Bucket Sort) (и кой е най-лошият случай)?
  5. Как се сортира за линейно време масив от букви на английски език?
  6. При Сортирането чрез преброяване (Counting Sort) след етап 2 (преброяването) вече може да се произведе резултата (знаем, колко пъти се съдържа всяко число). Защо не го правим? (Отговорът е в слайдовете.)
- 

### Отговори на въпросите за преговор

1. Алгоритъмът проверява, дали масивът е сортиран, и, ако е, просто го връща, иначе задейства Сортирането чрез купчина (Heap Sort) (например):

```
Linear-Best-Case-Sort( $A, n$ )
1. for  $i \leftarrow 1$  to  $n-1$ 
2.   if  $A[i] > A[i+1]$ 
3.     return Heap-Sort( $A, n$ )
```

Сортирането чрез вкарване (Insertion Sort) също се изпълнява за линейно време в най-добрия случай.

2. Ако има повече от  $n!$  листа, то съществува излишно сравнение, което се изпълнява от алгоритъма.
  3. Сортирането чрез балончета извършва в най-лошия случай  $\Theta(n^2)$  сравнения и това е височината на съответното дърво за взимане на решения. Сортирането чрез купчина извършва в най-лошия случай  $\Theta(n\log n)$  сравнения и това е височината на съответното дърво за взимане на решения.
  4. В най-лошия случай всички елементи влизат в същата кофа, тоест създава се една кофа с  $n$  елемента. Сортирането на тази кофа посредством Сортирането чрез вкарване (Insertion Sort) отнема  $\Theta(n^2)$  време (може, разбира се, да се подобри на  $\Theta(n\log n)$ , ако използваме Сортирането чрез купчина (Heap Sort)).
  5. Сортирането чрез преброяване (Counting Sort), например, ще работи за линейно време, като всяка буква се представя от число между 1 и 26.
-

## Упражнения

---

### Препоръчителни упражнения от учебника

<u>Глава 8</u>	8.2-3	8.3-4	задача 8-3
8.1-1	8.2-4	8.4-2	задача 8-4
8.1-4	8.3-2	задача 8-2	

---

### Допълнителни упражнения

1. а. Докажете, че всеки алгоритъм за сортиране чрез сравнения, който сортира масив с дължина 5, трябва задължително да извърши (в най-лошия случай) поне 7 сортирания.  
б. Колко сравнения извършват алгоритмите за сортиране чрез сравнения, с които се запознахме в този курс, върху масив с дължина 5 в най-лошия и в най-добрания случай?  
в. Предизвикателство: Напишете алгоритъм за сортиране чрез сравнения, който извършва 7 сравнения в най-лошия случай върху масив с дължина 5.
2. Един професор в една катедра в един колеж в северната част на страната твърди, че след дългогодишни изследвания е открил линеен алгоритъм, който приема купчина от  $n$  елемента и отпечатва елементите й сортирани.  
Бихте ли позволили на този професор да преподава Структури от данни? Ако да, покажете алгоритъм като този, който предлага професорът! Ако не, докажете, че твърдението му не може да бъде вярно!
3. Палиндром (Palindrome) с дължина  $n$  е поредица от символи  $a_1, a_2, \dots, a_n$ , за която е изпълнено  $a_i = a_j$  за всяко  $i = n - j + 1$ , за  $1 \leq i, j \leq n$ .

Дадена е поредица с дължина  $n$  от цели числа от 1 до  $3n$  и е известно, че всяко число се съдържа четен брой пъти. Възможно ли е да се превърне дадената поредица в палиндром с дължина  $n$ , който съдържа точно същите числа, за време за изпълнение  $O(n \log n)$ ? Покажете как или докажете, че не е възможно!

### Допълнителни упражнения

4. Дадени са  $n$  на брой цели числа от областта  $[0, n^2 - 1]$ . Предложете ефективен начин за сортирането им! Подсказка: Кое е максималното число между две цифри с основа  $n$ ?

#### 5. Задачата с параболния вектор

В задачата с параболния вектор е даден масив с размер  $n$  от различни цели числа (няма ограничение за размера им) и трябва да се подредят наново елементите на масива, така че да е изпълнено:

- за всяко  $1 < i \leq \lceil n / 2 \rceil$  да е изпълнено  $A[i] \geq A[i - 1]$
- за всяко  $\lceil n / 2 \rceil + 1 \leq i < n$  да е изпълнено  $A[i] \geq A[i + 1]$

Масив, който е подреден по гореспоменатия начин, се нарича параболен вектор.

Например, следният вектор е параболен вектор:  $< 3, 4, 12, 9, 8, 6 >$

- а. Дайте добра граница (изразена чрез  $\Omega$ ) за необходимата времева сложност за решаването на задачата с параболния вектор!
- б. Представете оптимален алгоритъм (както от гледна точка на времевата сложност, така и от гледна точка сложността на разположението) за решаването на задачата с параболния вектор!
- .....

### Решение 1

а. Докажахме, че височината на дървото за взимане на решения при сортирането чрез сравнения върху масив с размер  $n$  е поне  $\log_2(n!)$ .

Ще заместим  $n = 5$  и ще получим:  $h \geq \log_2(5!) \cong 6.9$

Височината е цяло число, затова височината на гореспоменатото дърво за взимане на решения е поне 7.

И тъй като височината на дървото за взимане на решения представя най-дългия маршрут от корена до листото, това също е и броят на сравненията в най-лошия случай, който алгоритъмът е извършил.

6.

сортиране	най-лош случай	най-добър случай
Сортиране чрез вкарване (Insertion Sort)	10	4
Сортиране чрез сливане (Merge Sort)	8	5
Бързо сортиране (Quick Sort)	10	6
Сортиране чрез купчина (Heap Sort)	12	9

.....

### Решение 2

Твърдението на професора не е вярно.

Нека предположим, че съществува алгоритъм, ще го наречем  $\text{Alg}$ , който получава купчина и отпечатва елементите й сортирани за линейно време според размера на купчината.

От това предположение следва, че всеки масив може да се сортира за линейно време: Ще построим от масива купчина за линейно време посредством Build-Heap, и тогава ще извикаме гореспоменатия алгоритъм  $\text{Alg}$  върху купчината, която построихме като входни данни.

Това, разбра се, е в противоречие с добра граница, която учихме относно сортирането.

### Решение 3

Сортирането чрез преброяване (Counting Sort) (или сортирането чрез кофа (Bucket Sort)) върху дадената редица ще се изпълни за време  $\Theta(n + k) = \Theta(n + 3n) = \Theta(n)$ .

След сортирането ще извършим едно обхождане и ще поставим елемент с четен индекс на следващото свободно място от началото и елемент с нечетен индекс на следващото свободно място от края.

.....

## Решение 4

Ще изprobваме няколко стратегии:

- 1) Сортиране посредством сравнения би отнело  $\Omega(n \log n)$ .
- 2) Сортирането чрез преброяване (Counting Sort) би отнело  $O(n + k) = O(n + n^2) = O(n^2)$ .
- 3) Ще прехвърлим всяко число към основа  $n$ . Така всяко число ще бъде най-много с 2 цифри.

- Сложност  $O(n)$  (при предположението, че деление и остатък се извършват за константно време)
  - Сега ще задействаме Базисното сортиране (Radix Sort) върху числата.
- Сложност  $O(d(n + b)) = O(2(n + n)) = O(n)$ .

Третото решение е предпочитаното (и е оптимално от гледна точка на порядъка – защо?).

с 30

## Решение 5

- a. Ще докажем, че долната граница на задачата е  $\Omega(n \log n)$ .

Нека предположим, че съществува алгоритъм, който превръща даден масив с размер  $n$  в параболен вектор и се изпълнява за време  $o(n \log n)$  (малко  $o$ ). Ще наречем този алгоритъм PV.

Сега ще покажем, че от първоначалното предположение следва, че съществува алгоритъм за сортиране, който се изпълнява за време  $o(n \log n)$ :

Алгоритъмът за сортиране (получава масив A с размер n):

1. Изпълнете PV върху масива A! // създава се параболен вектор
2. Обърнете реда на елементите във втората половина на A! // сега всяка половина е сортирана
3. Слейте двете половини на A! // сега A е сортиран

Този алгоритъм за сортиране се изпълнява за време  $o(n \log n) + \Theta(n) + \Theta(n) = o(n \log n)$ .

И това, разбира се, е противоречие спрямо долната граница на Сортирането чрез сравнения, която учихме. Затова първоначалното ни предположение е грешно.

б. Сортиране чрез купчина (Heap Sort) и след това обръщане реда на елементите в лявата половина на масива.

Времевата сложност е  $\Theta(n \log n)$ , а сложността на допълнителното месторазположение е  $\Theta(1)$ .

## Тема 7

# Задачата с избора Selection Problem

### В програмата:

Глава 9 от учебника:

- Ще дадем определение за задачата с избора.
- Ще се запознаем с ефективни решения на задачата с избора
  - Алгоритъмът Random Select
  - Алгоритъмът Select

### Задачата с избора

#### Определение на задачата

Входни данни: Група от  $n$  елемента и индекс  $1 \leq i \leq n$ .

Изходни данни:  $i$ -тият елемент е най-малък, тоест:

елементът\* на  $i$ -то място в сортирания ред.

\* елемент или елементът?

#### Решения на задачата (алгоритми, които я решават):

- Сортиране и връщане на елемента с индекс  $i$  за време  $\Omega(n \log n)$ .
- Съществува решение за линейно време за:  
 $i = 1$  (минимум)  
 $i = n$  (максимум)  
и на практика също и за  $i = k$  или  $i = n - k$ , като  $k$  е константа.

```
Minimum( $A$ ) ►  $A$  is array
1.  $min \leftarrow A[1]$ 
2. for  $i \leftarrow 2$  to  $\text{length}[A]$ 
3.   if  $A[i] < min$ 
4.      $min \leftarrow A[i]$ 
5. return  $A[min]$ 
```

Въпрос: Колко пъти средно се изпълнява ред 4?

Отговор: Шансът, елементът  $A[k]$  да бъде минимума на  $A[1 \dots k]$ , е  $1/k$ .

Сборът от математическите очаквания на броя на влизанията на 4-тия ред е  $1/2 + 1/3 + \dots + 1/n = H_n - 1 = \Theta(\log n)$ .

## Задачата с избора

- Съществува ли линейно решение за всяко  $i$ ?

Например, може ли да се намери медианата за линейно време?

Медианата (median) ще бъде определена като  $\left\lfloor \frac{n+1}{2} \right\rfloor$ -ия по размер елемент.

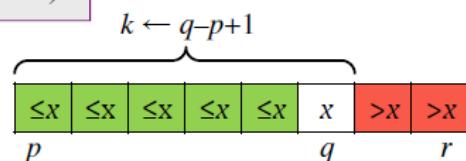
Сега ще представим два алгоритъма:

- ✓ Random Select, който решава задачата средно за линейно време, но не в най-лошия случай. Това е случаен рекурсивен алгоритъм.
  - ✓ Select, който решава задачата за линейно време в най-лошия случай (но е сложен ...). Това е детерминантен рекурсивен алгоритъм.
- 

## Random Select

Random-Select ( $A, p, r, i$ )

```
1. if  $p = r$ 
2.   return  $A[p]$ 
3.  $q \leftarrow$  Random-Partition ( $A, p, r$ )
4.  $k \leftarrow q-p+1$ 
5. if  $i = k$ 
6.   return  $A[q]$ 
7. else if  $i < k$ 
8.   return Random-Select ( $A, p, q-1, i$ )
9. else return Random-Select ( $A, q+1, r, i-k$ )
```



Алгоритъмът извършва разделение (със случайно избрана ос) на масива.

- Ако „имаме късмет“ и броят на елементите  $k$  в лявата част, включително оста, е точно  $i$ , то тогава приключваме;
  - Иначе, ако  $i < k$ , то продължаваме да търсим  $i$ -тия елемент от лявата страна;
  - Иначе, продължаваме да търсим  $(i - k)$ -тия елемент от дясната страна.
- 

## Random Select

### Демонстрация

Търсим 4-тия елемент.

Нека предположим, че случайно избраната ос (pivot) е 26.

50	30	20	25	10	15	18	77	5	20	88	26
1	2	3	4	5	6	7	8	9	10	11	12

Резултатът от делението:

Търсим 4-тия елемент отляво.

20	25	10	15	18	5	20	26	30	50	88	77
1	2	3	4	5	6	7	8	9	10	11	12

Нека предположим, че оста (pivot) е 15.

20	25	10	15	18	5	20
1	2	3	4	5	6	7

Резултатът от делението:

Търсим  $4 - 3 = 1$  първия елемент отдясно.

10	5	15	20	18	25	20
1	2	3	4	5	6	7

Нека предположим, че оста (pivot) е 18.

20	18	25	20
4	5	6	7

Резултатът от делението:

Спирате и връщате 18.

18	20	25	20
4	5	6	7

## Random Select

### Анализ на времената

Най-добър случай: едно деление и след него спирате.  $T(n) = \Theta(n)$

Най-лош случай: деление в съотношение  $0 : n - 1$  на всеки етап и продължаване на рекурсията с голямата част.  $T(n) = T(n - 1) + \Theta(n) = \Theta(n^2)$

Средно (доказателство в учебника):  $\Theta(n)$

В обобщение намерихме средно линейно решение на задачата с избора, но не в най-лошия случай.

## Частен случай – намиране на медиана

Въпрос: Как да намерим медианата на масив с размер  $n$  в средния случай с линейна времева сложност?

Отговор: Ще извикаме  $\text{RandomSelect}(A, 1, n, \left\lfloor \frac{n+1}{2} \right\rfloor)$ .

## Select

Идеята за подобрение е предложена през 1973 г. в статията:

“Time bounds for selection”, M. Blum, R. W. Floyd. V. Pratt, R. Rivest and R. Tarjan,  
*J. Comput. System Sci.* 7 (1973) 448-461.

Идеята: За да се подобри времето за изпълнение, трябва да се погрижим, деленията да бъдат относително балансирани.

- Не задължително в съотношение 1:1, но както ще видим, в най-крайния лош случай в съотношение приблизително 3:7.
- Ще покажем метод за избор на осевия елемент (pivot) по по-„умен“ начин, който ще го гарантира.
- Ще използваме малко по-различна версия за деление, която получава осев елемент и дели по него. Ще наречем тази версия „Деление с осев елемент“ – “Partition With Pivot”.

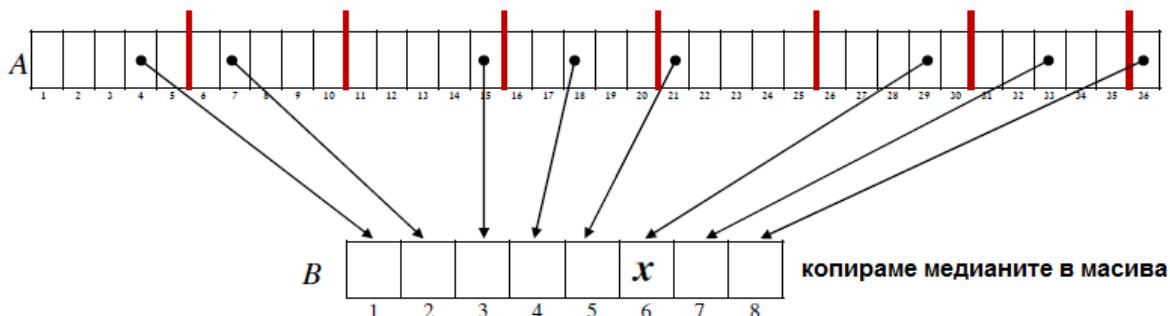
Идеята за избора на ос:

Методът на медианата на медианите.

## Select

Идеята

Разделяме на петорки и намираме медианата на всяка петорка (например чрез сортирането ѝ):



Авторите на този метод са доказали, че медианата на тези медиани (медианата на B) е добър осев елемент (pivot) за разделянето на A.

Познавате ли ефективен метод за намирането на медиана?

Рекурсивно извикване на Select! И в този пример трябва да извикаме Select(B, 1, 8, 4).

Медианата на медианите, която открихме (обозначена с x), е оста за разделянето на A.

Продължението е като при Random Select.  
с 10

## Select

Алгоритъмът

```
Select (A, p, r, i)
1. if  $p = r$ 
2.   return  $A[p]$ 
3.  $x \leftarrow \text{Choose-Pivot}(A, p, r)$ 
4.  $q \leftarrow \text{Partition-With-Pivot}(A, p, r, x)$ 
5.  $k \leftarrow q-p+1$ 
6. if  $i = k$ 
7.   return  $A[q]$ 
8. else if  $i < k$ 
9.   return Select (A, p, q-1, i)
10.  else return Select (A, q+1, r, i-k)
```

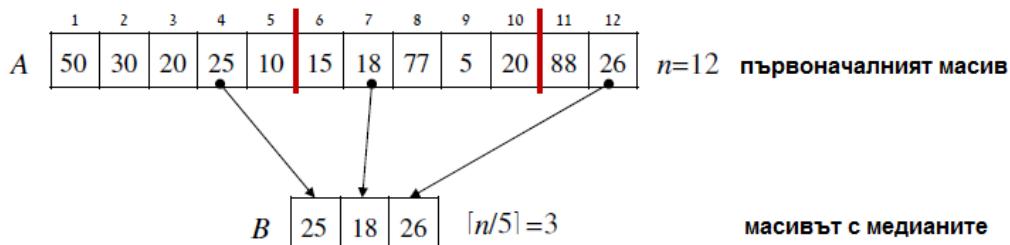
#### Choose-Pivot (A, p, r)

1. Ще намерим медианата на всяка петорка от последователни елементи (и освен това може би и остатъка на края), например, чрез сортиране на всяка група. Ще копираме тези медиани в помощен масив B (размерът му е  $\lceil n/5 \rceil$ ).
2. Ще намерим и ще върнем медианата на B (медианата на медианите). Ще го направим чрез задействането на Select върху B.

## Select

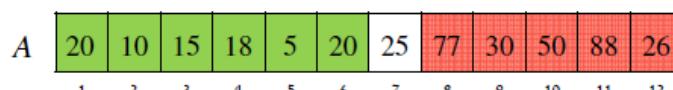
### Демонстрация

Търсим 4-ия елемент.



Намирането на медианата на медианите се извършва директно чрез сортиране (и без рекурсия), тъй като размерът на B  $\leq 5$ .

Деление с осев елемент (Partition With Pivot) на A около  $x = 25$ :



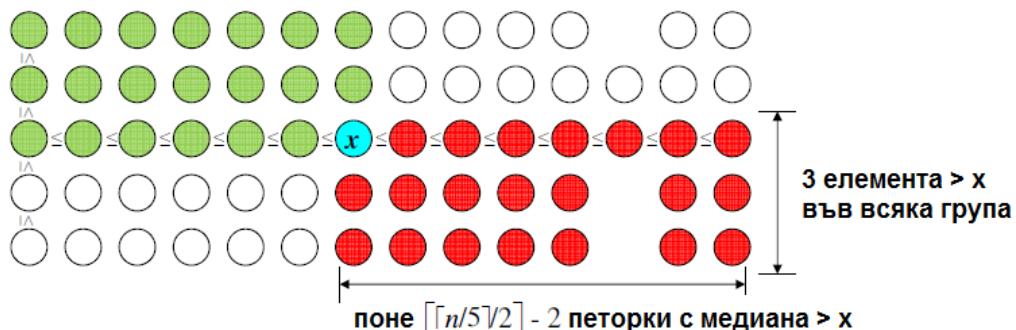
И продължаваме да търсим 4-ия елемент от лявата страна:  $\text{Select}(A, 1, 6, 4)$

## Select – Анализ на времената

### Анализ на времената в най-лошия случай

Нека предположим с цел улеснение, че всички елементи са различни един от друг.

С цел яснота ще подредим петорките на следната илюстрация според размера на медианата им отляво надясно (всяка петорка е сортирана отгоре надолу):



- Във всяка петорка, чиято медиана е по-голяма от оста (pivot) x, има точно 3 елемента > x.
- Колко такива петорки има? Тъй като x е медианата на медианите на петорките, половината от тях са > x. Ще извадим от това групата с по-малко от 5 елемента (ако има такава) и петорката включваща самия x.
- От тук следва, че в A има поне  $3(\lceil \frac{n}{5} \rceil / 2) - 2 \geq 3n/10 - 6$  елемента > x, тоест поне около 30%.

### Select – Анализ на времената

По подобен начин показваме, че в A има поне  $3n/10 - 6$  елемента > x.

С други думи дори и при най-лошото деление в по-малката част има поне  $3n/10 - 6$  елемента, а в по-голямата част има най-много  $7n/10 + 6$  елемента.



- Тоест „умният“ избор на x като медиана на медианите гарантира, че възможно най-лошото деление е в съотношение приблизително 3:7.
- „Цената“, която заплатихме – намирането на x изиска време: разделянето на масива на петорки, намирането на медианите и рекурсивното извикване.

### Select – Анализ на времената

Изчислението цената на намирането медианата на медианите (Choose Pivot):

- Намирането на медианата на всяка петорка изисква  $\Theta(1)$  време (защо?).  
А общо намирането на  $\lceil n / 5 \rceil$  медиани изисква  $\Theta(n)$  време.
- След това имаме рекурсивно извикване на Select върху масив с размер  $\lceil n / 5 \rceil$ .

Формулата, която описва общото време за изпълнение на Select в най-лошия случай е:

$$T(n) = T(\lceil n/5 \rceil) + T(7n/10 + 6) + \Theta(n)$$

намиране на медианите  
+  
деление

По метода на позиционирането може да се докаже, че  $T(n) = \Theta(n)$  (подробности в учебника, стр. 159). Това може да се види и чрез пренебрегването на константите и горните гранични стойности:

$$T(n) = T(n/5) + T(7n/10) + \Theta(n)$$

$\alpha$     +     $\beta$      $< 1$

Тоест, сега имаме линеен алгоритъм в най-лошия случай за решаването на задачата за избора.

---

### Select – Сложност по отношение на паметта

А каква е сложността по отношение на допълнителната памет на Select?

Изискванията за допълнителна памет на Select се изразяват в:

- Заделянето на масиви за медианите
- Дълбочината на стека (пълнителя) на рекурсията

Дълбочината на стека на рекурсията е логаритмична (защо?) и затова е асимптотно пренебрежима за разлика от заделянето на масивите за медианите, чийто размер е линеен спрямо размера на входните данни\*.

Рекурентното отношение описващо сложността по отношение на паметта на Select за входни данни с размер  $n$  е:

$$\begin{aligned} S(n) &= \max \{S(\lceil n/5 \rceil), S(7n/10 + 6)\} + \lceil n/5 \rceil \\ &= S(7n/10 + 6) + \lceil n/5 \rceil \\ \Rightarrow S(n) &= \Theta(n) \end{aligned}$$

\* Въпрос:

Как можем да се откажем от заделянето на масивите за медианите?

Каква е сложността по отношение на допълнителната памет с едно такова подобреие?

---

### Частен случай – намиране на медиана

Въпрос: Как да намерим медианата на масив с размер  $n$  при линейна времева сложност?

Отговор: Ще извикаме  $\text{Select}(A, 1, n, \left\lfloor \frac{n+1}{2} \right\rfloor)$ .

Забележка: Времевата сложност наистина е линейна, но скритите в нея константи са твърде големи. Затова от практична гледна точка понякога е препоръчително да се използва Random Select или дори да се сортира и да се върне средния елемент.

## Детерминантното Бързо сортиране (Quick Sort)

### Упражнение 9.3-3 от учебника

Как можем да подобрим Бързото сортиране, така че времето му за изпълнение върху масив с размер  $n$  да бъде  $\Theta(n \log n)$  в най-лошия случай?

Решение:

На всеки етап, на който трябва да изберем осев елемент (pivot) за деление, той ще се избере като медиана на релевантния под-масив посредством извикване на Select.

$$T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n)$$

Select + Partition

### Въпроси за преговор

1. Как можем да намерим минимума на един масив посредством извикване на Select? А максимума? Този начин ефективен ли е от теоретична и от практическа гледна точка?
2. За кои размери на масиви Choose Pivot не трябва повторно да извика Select? Какво прави в този случай Choose Pivot?
3. Опитайте се да проследите процеса на рекурсивните извиквания на Select върху масив A с размер 31!
4. Делението чрез осев елемент (Partition With Pivot) може да се осъществи чрез извикване на „обикновеното“ деление (Partition). Как?

### Отговори на въпросите за преговор

1. Минимум:  $\text{Select}(A, 1, n, 1)$ . Максимум:  $\text{Select}(A, 1, n, n)$ .  
От теоретична гледна точка времето за изпълнение и в двата случая е линейно по  $n$ . Но, както видяхме, можем да намерим минимум и максимум също и без използване на Select, чрез просто обхождане на масива. Това също е линейно решение, но с много по-малки константи. Затова от практическа гледна точка то е препоръчително.
2. За масив с достатъчно малък размер (например 5 или каквато и да е друга константа) можем просто да върнем медианата му (чрез сортиране).
3. Масивът B ще бъде с размер 7 и затова намирането на медианата му ще доведе до допълнително извикване на Select. B ще бъде разделен на петорки и ще се получи още един масив с медиани C с размер 2. Медианата  $x'$  на C ще бъде върната без допълнителни

извиквания на Select и ще послужи за разделянето на B. Сега търсенето на медианата на B ще продължи в лявата или в дясната част на разделението (или в най-добрия случай ще приключи веднага). Едва след намирането медианата x на B, тя ще бъде върната и ще послужи за разделянето на A, а след разделянето на A ще продължим да търсим i-тия елемент на A в лявата или в дясната част на разделението (или в най-добрия случай ще приключим веднага).

4. Разменяме x и елемента на последното място, след което просто извикваме Partition.

с 20

## Упражнения

### Препоръчителни упражнения от учебника

<u>Глава 9</u>	9.3-5	9.3-9
9.3-1	9.3-7	задача 9-1
9.3-3	9.3-8	задача 9-2, а - в

### Допълнителни упражнения

1. Ще определим най-често срещан елемент (елемент на мнозинството) в масив с размер n като елемент, съдържащ се  $n/2$  пъти. Предложете алгоритъм, който при даден масив с размер n намира най-често срещания елемент, ако съществува такъв, или в обратния случай съобщава, че не съществува най-често срещан елемент!

2. Даден е масив A с размер n с никакви елементи.

а. Покажете алгоритъм за отпечатване на най-малките  $\lfloor \sqrt{n} \rfloor$  на брой елемента в масива в сортиран ред за линейно време в най-лошия случай! Обосновете се, защо не може да се реши тази задача за линейно време  $O(n)$ !

б. Докажете, че не е възможно да се отпечатат най-малките  $\lfloor n / 10 \rfloor$  на брой елемента в масива в сортиран ред за време  $O(n \log n)$ !

3. Даден е масив, в който има m на брой различни един от друг никакви елементи.

Предложете структура от данни за изпълнение на следните действия, като се придържате към изискванията за времевата сложност:

- Init – инициализация на структурата от данни за  $O(m)$ .
- Insert(x) – добавяне на x към структурата за  $O(\log n)$ , n е броят на елементите в структурата към момента на извършване на действието.
- Find-Mid – отпечатване стойността на медианата за  $O(1)$ .
- Del-Mid – изваждане на медианата от структурата за  $O(\log n)$ .

Първо оишете, какво включва Вашата имплементация, и след това обяснете, как се извършва всяко действие, и защо отговаря на изискванията на сложността!

## Решение 1

### Първи опит

Някакво сортиране –  $\Omega(n \log n)$  и обхождане на масива, за да се провери, дали има  $n/2$  последователни еднакви елемента –  $\Theta(n)$ .

Общо:  $\Omega(n \log n)$  (плътната граница зависи от избора на сортиране).

### Втори опит

Ще обърнем внимание, че ако съществува най-често срещан елемент (елемент на мнозинството), то той е медианата.

Ще намерим посредством Select медианата, ще обходим масива и ще преброим, колко пъти се съдържа в него медианата. Ако се съдържа повече от  $n/2$  пъти, ще върнем него. Иначе ще съобщим, че няма най-често срещан елемент.

Сложност:  $\Theta(n)$

.....

## Решение 2

а. Ще задействаме Select за намирането на  $\lfloor \sqrt{n} \rfloor$ -ия елемент по размер:  $\text{Select}(A, 1, n, \lfloor \sqrt{n} \rfloor)$

Сега най-малките  $\lfloor \sqrt{n} \rfloor$  на брой елемента на A са разположени в лявата част на масива.

Ще ги сортираме посредством Сортирането чрез балончета (Bubble Sort) например.

Времевата сложност е:  $\Theta(n) + \Theta(\lfloor \sqrt{n} \rfloor^2) = \Theta(n)$

Дължни сме да обходим всички  $n$  елемента на масива, затова не е възможно да се реши задачата за време  $O(n)$ .

б. Нека предположим, че е възможно да се направи това. Тоест, съществува алгоритъм, нека го наречем Alg, който отпечатва една десета от елементите на даден масив, съдържащи най-малките елементи, в сортиран ред.

Тогава е възможно да се сортира всеки масив с размер  $n$  за време  $O(n \log n)$  (и това, разбира се е в противоречие с долната граница на сортирането):

- При наличието на входния масив ще му добавим отлясно  $9n$  пъти „безкрайност“ (практически първо можем да намерим максимума и да добавим  $9n$  елемента, които са равни на максимума плюс още един). Този етап се изпълнява за време  $\Theta(n)$ .
- Сега ще извикаме алгоритъма Alg върху „уголемения“ масив A и ще получим елементите на първоначалния масив сортирани.

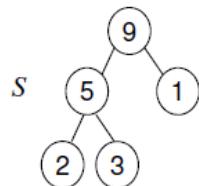
Обърнете внимание, че не е достатъчно да се покаже, че начинът от подточка (a) не работи тук.

.....

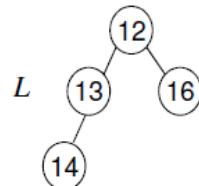
### Решение 3

#### Структурата от данни:

Минимална купчина L от най-големите  
 $\lfloor n/2 \rfloor$  на брой елемента



Максимална купчина S от най-малките  
 $\lceil n/2 \rceil$  на брой елемента



#### Имплементация на действията:

**Init** – Ще задействаме Select за намирането на медианата (включително разделянето на елементите около нея). Сега ще построим двете гореспоменати купчини от двете половини на масива с помощта на две извиквания на Build-Heap.

Време:  $\Theta(m) + 2\Theta(m/2) = \Theta(m)$

**Find-Mid** – Ще върнем стойността на корена на S (тоест  $S[1]$ ). Време:  $\Theta(1)$ .

**Insert(x)** – Ако  $x \leq S[1]$ , ще го вкарваме в S, иначе в L. Сега:

Ако  $|L| = |S| - 2$ , ще изтрием (Extract-Max) корена на S и ще го вкарваме (Heap-Insert) в L.

Иначе ако  $|L| = |S| + 1$ , ще изтрием корена на L и ще го вкарваме в S.

**Del-Mid** - ще изтрием корена на S. Ако  $|L| = |S| + 1$ , ще действаме, както описахме по-горе.

Време: двете последни действия в най-лошия случай:  $\Theta(3\log(n/2)) = \Theta(\log n)$ .

## Тема 8

# Дървета за Бинарно търсене Binary Search Trees

В програмата:

Глава 12 от учебника:

- Ще се запознаем с абстрактната структура от данни (ADT) „речник“.
  - Ще се запознаем със структурата от данни „дърво за бинарно търсене“ за имплементация на речник и още няколко полезни свойства и алгоритми свързани с дървета.
- 

### Речник (Dictionary)

Речникът е абстрактна структура от данни (ADT), определена от следните действия:

- Insert( $S, x$ ) – вкарване на елемента, към който сочи,  $x$  в структурата  $S$
- Delete( $S, x$ ) – изтриване на елемента, към който сочи,  $x$  от структурата  $S$
- Search( $S, k$ ) – търсене на елемента, чийто ключ е  $k$  и връщането му, ако съществува в  $S$

А понякога и допълнителни действия за търсене като:

- Minimum( $S$ ) – връщане на елемента с най-малък ключ
- Maximum( $S$ ) – връщане на елемента с най-голям ключ
- Successor( $S, x$ ) – връщане на елемента, следващ  $x$  в  $S$  (елемента с най-малкия ключ, който е по-голям от този на  $x$ )
- Predecessor( $S, x$ ) – връщане на елемента, предхождащ  $x$  в  $S$  (елемента с най-големия ключ, който е по-малък от този на  $x$ )

Последните 4 действия предполагат, че съществува някакъв ред при ключовете на елементите.

Речникът има много на брой разнообразни приложения (например?).

---

### Структури от данни за речника

- При имплементацията на речник за всяка една от структурите от данни, които учихме до момента, поне едно от действията ще се ще се извърши за линейно време по броя на елементите.
- Сега ще покажем възможна имплементация на речник посредством структурата от данни дърво за бинарно търсене.

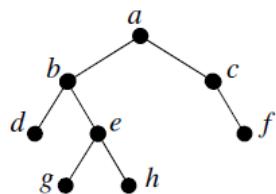
Средно всички действия ще се извършат със сложност  $\Theta(\log n)$ .

- В следващия урок ще покажем усъвършенстване на дърветата за бинарно търсене – AVL дървета, които гарантират сложност от  $\Theta(\log n)$  също и в най-лошия случай.
- 

## Дървета за Бинарно **търсене** (Бинарни дървета) – припомняне

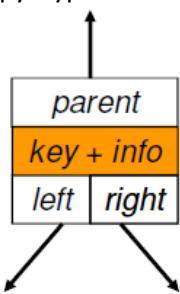
### Понятия – припомняне

- Кръстовище
- Дъга



- Корен
- Ляв наследник, десен наследник, родител, прародител, потомък
- Ляво под-дърво, дясно под-дърво
- Листо, вътрешно кръстовище
- Дълбочина на кръстовище
- Височина на кръстовище, височина на дърво
- Имплементация чрез масив, имплементация чрез записи с показалци:

Структура на запис (кръстовище):



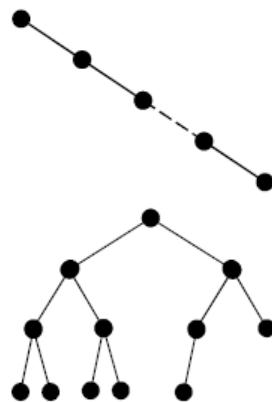
## Височина на Бинарно дърво

Ще отбележим броя на кръстовищата в бинарното дърво с  $n$ , а височината с  $h$ .

Каква е минималната височина и каква е максималната височина на едно бинарно дърво в зависимост от  $n$ ?

Максимална: верига или „зиг-заг“

$$h = n - 1 = \Theta(n)$$



Минимална: (почти) пълно дърво

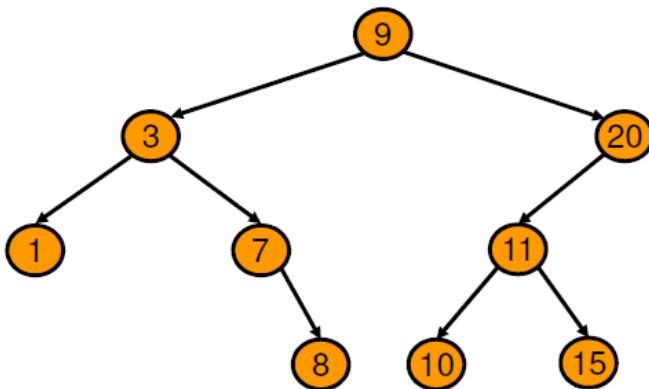
$$h = \lfloor \log n \rfloor = \Theta(\log n)$$

## Дърво за Бинарно търсене

Дървото за Бинарно търсене (Binary Search Tree) е бинарно дърво със следните свойства:

При дадено кръстовище с ключ k:

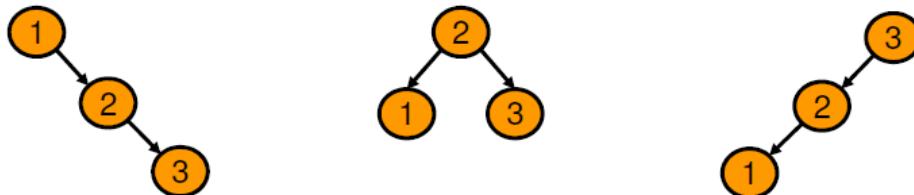
- Кръстовищата в дясното му под-дърво са с ключове  $\geq k$ .
- Кръстовищата в лявото му под-дърво са с ключове  $\leq k$ .



При дадена група от ключове съществува ли едно единствено дърво за бинарно търсене, съдържащо тези ключове?

## Дърво за Бинарно търсене

При дадена група от ключове съществува ли едно единствено дърво за бинарно търсене, съдържащо тези ключове?



Още?

## Колко различни бинарни дървета с n на брой кръстовища съществуват?

Ще отбележим отговора с  $b_n$ .

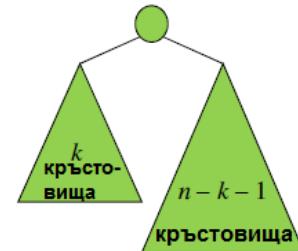
Рекурсивна формула за  $b_n$ :

За  $n = 0$ : има само едно дърво без кръстовища – празното дърво.

Затова  $b_0 = 1$ .

За  $n \geq 1$ : дърво, съставено от едно кръстовище при корена, k кръстовища в лявото под-дърво ( $k = 0, 1, \dots, n - 1$ ) и  $n - k - 1$  кръстовища в дясното под-дърво. Затова:

$$b_n = \sum_{k=0}^{n-1} b_k \cdot b_{n-k-1}$$



Решението на формулата (извън рамките на средствата, които учихме в този курс):

$$b_n = \frac{1}{n+1} \binom{2n}{n} = \frac{4^n}{\sqrt{\pi} n^{3/2}} (1 + O(1/n))$$

$b_n$  се наричат **Каталански числа** (Catalan Numbers).

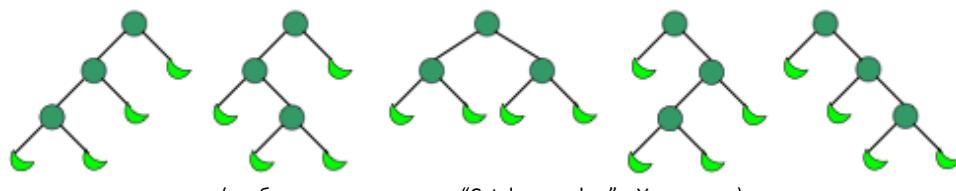
## Каталански числа (Catalan Numbers)

Каталанските числа представляват също и броя различни начини за поставяне на скоби върху  $n + 1$  различни фактора.

Например, когато  $n = 3$ , имаме 5 различни начина за поставяне на скоби върху 4 фактора:

$$((ab)c)d \quad (a(bc))d \quad (ab)(cd) \quad a((bc)d) \quad a(b(cd))$$

Съществува инжективна функция между гореспоменатата група от изрази и групата бинарни дървета с по  $n$  кръстовища:



с 10

## Търсене

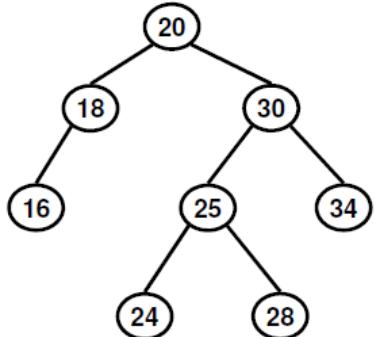
### Как да търсим ключ в бинарно дърво за търсене?

Tree-Search( $x, k$ )

1. **if**  $x = \text{nil}$  **or**  $k = \text{key}[x]$
2.     **return**  $x$
3. **if**  $k < \text{key}[x]$
4.     **return** Tree-Search( $\text{left}[x], k$ )
5. **else return** Tree-Search( $\text{right}[x], k$ )

При първото извикване  $x$  е коренът на дървото.

Какъв е маршрутът за търсене на 26 в следното дърво?



Какви са сложността по отношение на времето и по отношение на допълнителната памет?

Времева сложност:  $\Theta(h)$ .

Сложност по отношение на допълнителната памет:  $\Theta(h)$ .

---

## Търсене

Можем също така да напишем и итеративна версия:

Tree-Search( $x, k$ )

1. **while**  $x \neq \text{nil}$  **and**  $k \neq \text{key}[x]$
2.     **if**  $k < \text{key}[x]$
3.          $x \leftarrow \text{left}[x]$
4.     **else**  $x \leftarrow \text{right}[x]$
5. **return**  $x$

Времева сложност:  $\Theta(h)$ .

Сложност по отношение на допълнителната памет:  $\Theta(1)$ .

Въпрос: Каква е времевата сложност в най-добрния случай?

---

## Допълнителни заявки – минимум и максимум

Търсенето принадлежи към типа действия, които се наричат заявки.

При тези действия няма промяна в структурата от данни, а само изваждане на информация от нея.

#### **Допълнителни заявки:**

- Връщане на елемента с минимален ключ (връщането на максимума е симетрично):

Tree-Min( $x$ )

1. **while**  $left[x] \neq \text{nil}$
  2.      $x \leftarrow left[x]$
  3. **return**  $x$

Времева сложност:  $\Theta(h)$

А в най-добрия случай?

## Допълнителни заявки – следващ и предишен

- Връщане на следващия елемент след даден елемент x

### Случай 1:

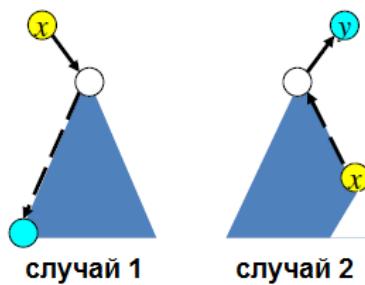
ХИМА ДЕСЕН НАСЛЕДНИК.

Следващият е минимумът на дясното под-дърво на  $x$ .

## Случай 2:

Х има лесен наследник.

Следващият (ако съществува) е първото кръстовище по нагорнището към корена, към който се качваме от ляв наследник.



случай 1	Tree-Successor( $x$ ) 1. if $right[x] \neq \text{nil}$ 2. return Tree-Min( $right[x]$ ) 3. $y \leftarrow parent[x]$ 4. while $y \neq \text{nil}$ and $x = right[y]$ 5. $x \leftarrow y$ 6. $y \leftarrow parent[y]$ 7. return $y$
случай 2	

Имплементацията за връщането на предишния елемент е симетрична.

Время сложности:  $\Theta(h)$ .

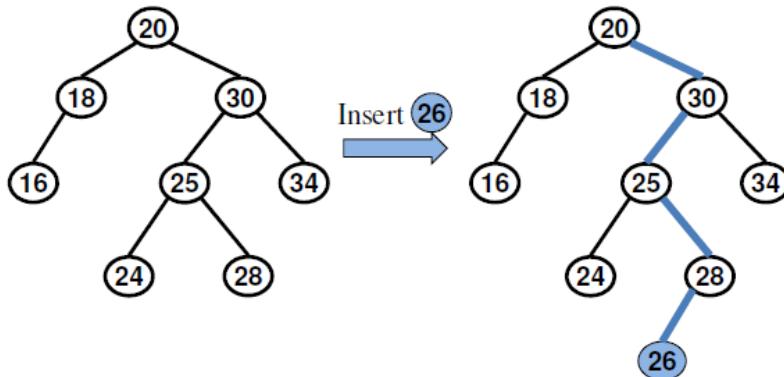
## Вкарване

Друг тип действия са динамичните действия, променящи структурата от данни (като вкарване, изтриване, промяна).

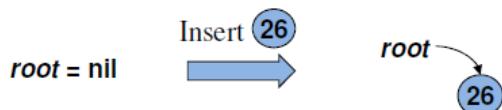
## Вкарване

Търсим съответното място за вкарване (по подобие на обикновеното търсене) и „окачваме“ новия елемент като листо.

## Демонстрация



Вкарване в празно дърво:



## Вкарване

### Алгоритъмът

Искаме да вкараме ново кръстовище  $z$  с ключ  $\text{key}[z]$  в дървото  $T$ . Предполагаме, че  $z$  съдържа полета, сочещи към левия и десния наследник и към родителя.

- Редове 1-7:  
Търсим родителя  $y$ , който е „готов да осинови“  $z$  (така че да се съхрани свойството на дървото за бинарно търсене).
- Редове 8-13:  
Поставяме  $z$  като наследник (ляв и десен) на  $y$ .
- Редове 9-10:  
Погрижваме се за крайния случай, в който  $z$  е вкаран в празно дърво.

Времева сложност:  $\Theta(h)$ .

### **Tree-Insert( $T, z$ )**

```

1.  $y \leftarrow \text{nil}$ 
2.  $x \leftarrow \text{root}[T]$ 
3. while  $x \neq \text{nil}$ 
4.    $y \leftarrow x$ 
5.   if  $\text{key}[z] < \text{key}[x]$ 
6.      $x \leftarrow \text{left}[x]$ 
7.   else  $x \leftarrow \text{right}[x]$ 
8.    $\text{parent}[z] \leftarrow y$ 
9.   if  $y = \text{nil}$ 
10.     $\text{root}[T] \leftarrow z$ 
11.    else if  $\text{key}[z] < \text{key}[y]$ 
12.       $\text{left}[y] \leftarrow z$ 
13.      else  $\text{right}[y] \leftarrow z$ 

```

## Изтриване

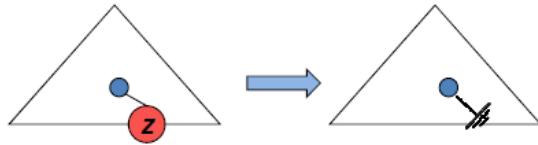
Искаме да изтрием от дървото T съществуващо кръстовище и ни е даден показалец z към това кръстовище.

- Откъде имаме показалец към това кръстовище?

Разделяме на случаи:

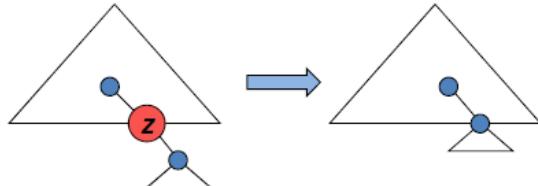
Случай 1: z е листо.

Поставяме nil на показалеца от родителя на z.



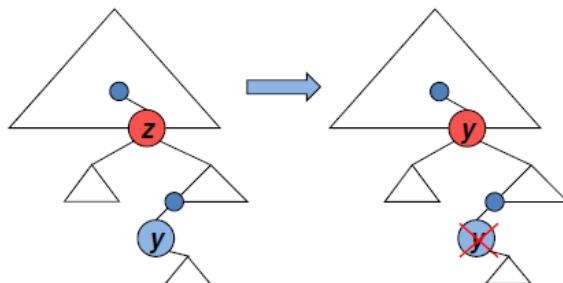
Случай 2: z има един единствен наследник.

Ще извършим „заобиколка“ от родителя на z към наследника му.



Случай 3: z има два наследника.

Нека y е следващият след z. Ще копираме съдържанието на y върху z и ще продължим с изтриването на y \*.

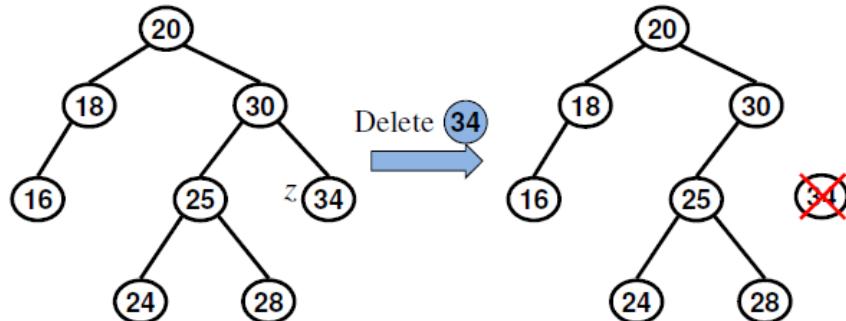


\* Изтриването на y принадлежи на случаи 1 или 2 (зашо?)

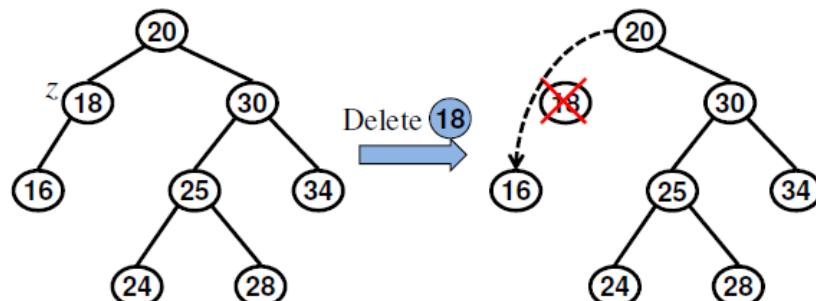
## Изтриване

Демонстрация

Случай 1



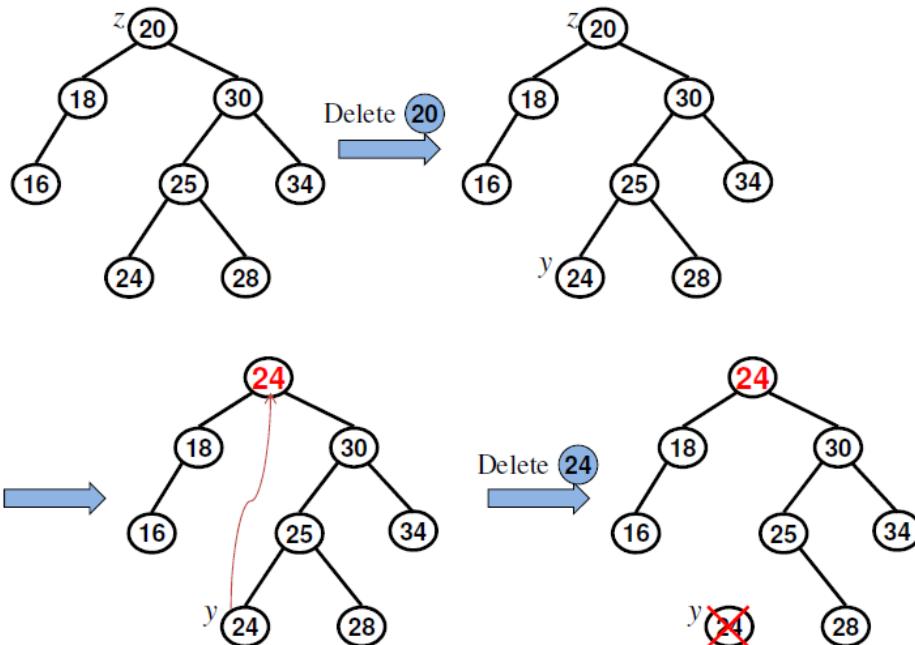
Случай 2



## Изтриване

### Демонстрация

Случай 3



## Изтриване

### Алгоритъмът

На редове 1-3 поставяме върху у кръстовището, което ще бъде „физически“ премахнато от дървото.

- Самото  $z$ , ако има най-много един наследник
- Следващият след  $z$ , ако има 2 наследника

Времева сложност:  $\Theta(h)$ .

**Tree-Delete( $T, z$ )**

```

1. if  $left[z] = \text{nil}$  or  $right[z] = \text{nil}$ 
2.   then  $y \leftarrow z$ 
3. else  $y \leftarrow \text{Tree-Successor}(z)$ 
4. if  $left[y] \neq \text{nil}$ 
5.   then  $x \leftarrow left[y]$ 
6. else  $x \leftarrow right[y]$ 
7. if  $x \neq \text{nil}$ 
8.   then  $p[x] \leftarrow p[y]$ 
9. if  $p[y] = \text{nil}$ 
10. then  $root[T] \leftarrow x$ 
11. else if  $y = left[p[y]]$ 
12.   then  $left[p[y]] \leftarrow x$ 
13.   else  $right[p[y]] \leftarrow x$ 
14. if  $y \neq z$ 
15.   then  $key[z] \leftarrow key[y]$ 
16.       ▶ copy  $y$ 's satellite data...
17. return  $y$  ▶ for recycling
    
```

## Проучвателно обхождане на дървета

Понякога искаме да обходим методично всички кръстовища на едно бинарно дърво с някаква цел. Такова действие се нарича проучвателно обхождане на дърво или сканиране на дървото.

Ще се запознаем с 3 метода за сканиране на бинарно дърво, които се различават един от друг по реда, в който се сканират кръстовищата.

### Начално проучвателно обхождане (pre-order)

- Посещаваме корена
- Рекурсивно проучвателно обхождаме лявото под-дърво
- Рекурсивно проучвателно обхождаме дясното под-дърво

Preorder-Tree-Walk( $x$ )

1. **if**  $x \neq \text{nil}$
2.     Visit( $x$ )
3.     Preorder-Tree-Walk( $\text{left}[x]$ )
4.     Preorder-Tree-Walk( $\text{right}[x]$ )

### Средно проучвателно обхождане (in-order)

Посещението на корена се извършва между проучвателното обхождане на лявото му под-дърво и проучвателното обхождане на дясното му под-дърво.

### Крайно проучвателно обхождане (post-order)

Посещението на корена се извършва след проучвателните обхождания на лявото и дясното му под-дърво.

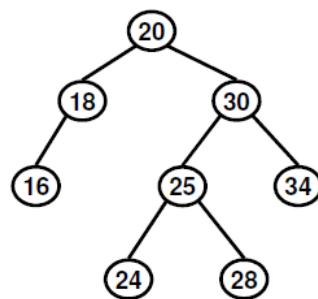
---

## Проучвателно обхождане на дървета

### Упражнение

Запишете реда на посещенията при всяко едно от проучвателните обхождания.

- Проучвателно обхождане pre-order:
- Проучвателно обхождане in-order:
- Проучвателно обхождане post-order:



## Проучвателно обхождане на дървета

### Твърдение:

Проучвателното обхождане in-order в едно дърво за бинарно търсене, в което Visit отпечатва ключа на кръстовището, дава в резултат сортирана редица (от ключовете на дървото).

### Доказателство:

Чрез индукция върху броя на кръстовищата в дървото.

Основа: За дърво с едно кръстовище твърдението със сигурност е вярно.

Стъпка: За дърво с повече от едно кръстовище първо ще бъдат отпечатани всички кръстовища в лявото под-дърво на корена, след това корена и едва след това всички кръстовища в дясното му под-дърво. Според предположението за индукция проучвателното обхождане на всяко едно от под-дърветата дава в резултат сортирана редица и според свойството на дървото за търсене цялата редица е сортирана.

.....

### Проучвателно обхождане на дървета

Времева сложност

Твърдение: Всяко едно от гореспоменатите три проучвателни обхождания се изпълнява върху дърво с  $n$  на брой кръстовища за време  $\Theta(n)$ .

Интуиция: Обхождаме всяко кръстовище 3 пъти: по пътя към левия му наследник, по пътя към десния му наследник и по пътя към родителя му.

Грешно доказателство: съответното рекурентно отношение е:  $T(n) = 2T(n/2) + 1 = \Theta(n)$

### **Каква е грешката?**

Доказателство: Ще обозначим броя на кръстовищата в лявото под-дърво на корена с  $k$ . Ще предположим, че посещението на едно кръстовище изисква  $d > 0$  действия.

$$T(0) = 1$$

$$T(n) = T(k) + T(n - k - 1) + d$$

Ще докажем по метода на позиционирането, че  $T(n) = (d + 1)n + 1$ :

Основа:  $T(0) = (d+1)\cdot 0 + 1 = 1$

Стъпка:  $T(n) = (d+1)k + 1 + (d+1)(n-k-1) + 1 + d = (d+1)n + 1$

.....

### Проучвателно обхождане на дървета – допълнителна памет

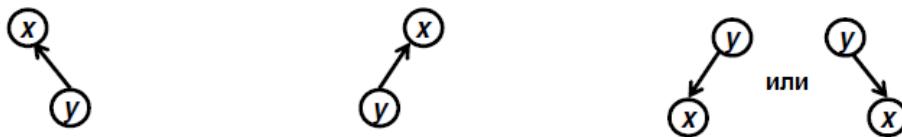
Сложност по отношение на допълнителната памет

Въпрос: Колко допълнителна памет изискват проучвателните обхождания в най-лошия случай?  
Какъв е най-лошия случай?

Отговор: За верига (или „зиг-заг“) дълбочината на рекурсията е  $\Theta(n)$ .

Подобрение: Проучвателните обхождания могат също да се имплементират и с  $O(1)$  допълнителна памет (разбира се, без рекурсия).

Идеята: Ще запазим във всеки момент две кръстовища: кръстовището, което посещаваме –  $x$ , и предишното кръстовище, което сме посетили –  $y$ . По такъв начин знаем, във всеки момент накъде трябва да продължим.



Продължение на проучвателното обхождане:

$x \leftarrow parent[x]$   
 $y \leftarrow x$

$x \leftarrow right[x]$   
 $y \leftarrow x$

$x \leftarrow left[x]$   
 $y \leftarrow x$

Разбира се, трябва да се погрижим за крайните случаи, както се изискава (например, когато  $x$  или  $y$  са равни на Nil).

### Проучвателно обхождане на дървета – допълнителна памет

#### Упражнение

а. Следният списък от ключове е получен чрез начално сканиране (pre-order) на дърво за бинарно търсене:

20 10 5 2 8 30 27 40

Възстановете първоначалното дърво!

б. Възможно ли е да се възстанови дърво за бинарно търсене при дадено post-order сканиране? А при in-order?

### Приложения на проучвателните обхождания – изчисление на височина

#### Как може да се изчисли височината на едно дърво за бинарно търсене?

(Височината на празно дърво е зададена като -1.)

```
height(x)
1. if x = nil
2.   return -1
3. else
4.   return 1 + max {height(left[x]), height(right[x])}
```

Сложност?

### Приложения на проучвателните обхождания – postfix изрази

Разбира се, всички проучвателни обхождания, които видяхме, могат да се изпълнят върху бинарни дървета, които не са дървета за търсене.

Приятен и полезен пример за това е изчислението на изрази със записа postfix.

Ще разгледаме следния израз и бинарното дърво, което представя структурата му:

$$(5 + 3) * ((20 / 10) + (8 - 6))$$

Проучвателно обхождане in-order на дървото (запис infix):

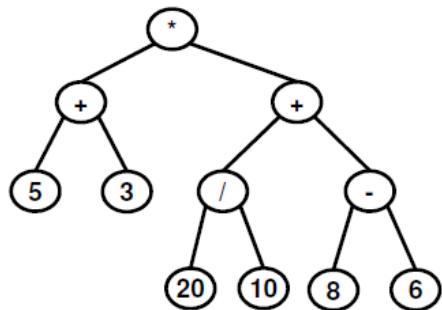
$$5 + 3 * 20 / 10 + 8 - 6$$

Извикването не е единствено – трябва да поставим скоби!

Запис postfix: 5 3 + 20 10 / 8 6 - + \*

Едно единствено извикване – не са нужни скоби.

(също така и за запис prefix)

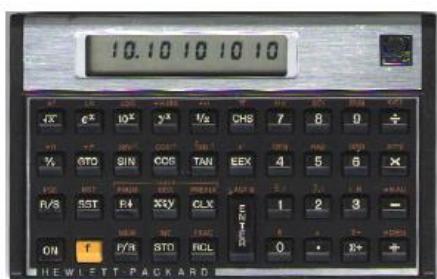


Изразите postfix се наричат Обратен полски запис (Reverse Polish Notation) на името на полския логик Lukasiewicz. При изразите postfix няма нужда от скоби: всеки postfix израз има най-много едно разлагане!

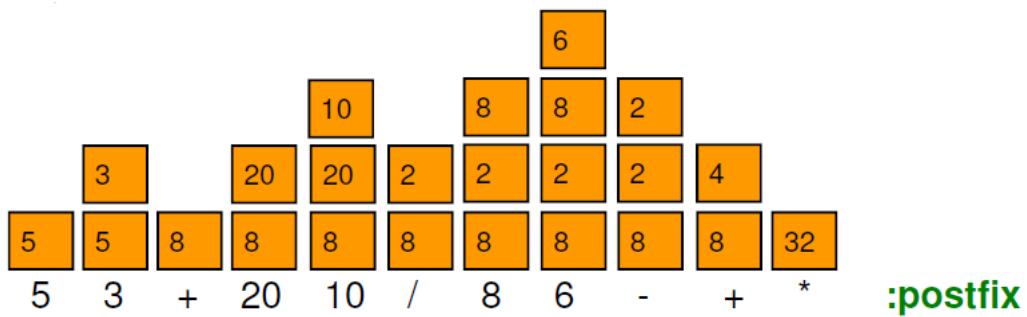
## Приложения на проучвателните обхождания – postfix изрази

### Алгоритъмът за изчисление на postfix изрази

1. Започнете с празен стек (пълнител)!
2. Обходете израза отляво надясно!
3. Ако следващият елемент е операнд – вкарайте го в стека!
4. Ако е действие – задействайте го върху двата елемента в началото на стека и вкарайте резултата в стека!



HP-10c programmable scientific, 1980's.  
Cost: \$80



### Средна височина

Видяхме по-рано, че максималната височина на едно бинарно дърво е  $n - 1$ , а минималната височина е  $\lfloor \log n \rfloor$ .

Въпрос: Искаме да вкараме в едно бинарно дърво ключовете 1, 2, 3.

- Колко различни последователности от вкарвания (ред на вкарване) са възможни?
- Кои от тях биха дали дърво с максимална височина?
- Кои от тях биха дали дърво с минимална височина?

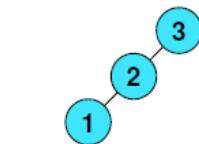
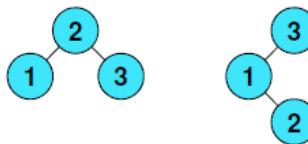
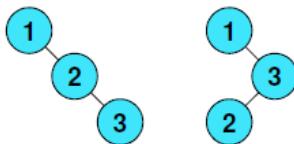
<1,2,3>

<1,3,2>

<2,1,3> and <2,3,1>

<3,1,2>

<3,2,1>



Средната височина е средно-аритметичната стойност на височините на дърветата, които се получават при различните последователности от вкарвания (ред на вкарване) (в примера:  $1^2/3$ ).

с 30

### Средна височина

Теорема:

Средната височина на едно дърво за бинарно търсене с  $n$  на брой кръстовища е  $\Theta(\log n)$ , като средната стойност се базира върху всички различни последователности от вкарвания (ред на вкарване) (само вкарвания, без изтривания или други действия).

Доказателство в учебника, страници 223-225.

Друга формулировка: Когато редът за вкарване на елементите е случаен, математическото очакване за височината е логаритмично по броя на кръстовищата.

## Въпроси за преговор

1. Посетете следния линк, предлагащ отлична симулация на дървета за бинарно търсене (и още няколко други типа дървета за търсене):

<http://people.ksp.sk/~kuko/bak/index.html>

Помогнете си със симулацията, за да разберете различните алгоритми, работещи върху дървета за бинарно търсене!

Допълнителни сайтове с демонстрации, които биха могли да помогнат:

- a. <http://nova.umuc.edu/~jarc/idsv/> (трябва да изберете Search Trees от менюто)
- b. <http://www.site.uottawa.ca/~stan/csi2514/applets/avl/BT.html> (трябва да отмените обозначението до AVL)

2. Нарисувайте всички възможни дървета, които се получават чрез вкарването на елементите 1, 2, 3, 4 в дърво за бинарно търсене!

3. Предложете рекурсивен алгоритъм за изчисление броя на кръстовищата в едно бинарно дърво! Каква е сложността му по отношение на времето и местоположението?

---

## Отговори на въпросите за преговор

3. Алгоритъмът:

```
Count(x)
1. if x = NIL
2.   return 0
3.   return 1 + Count(left[x]) + Count(right[x])
```

Сложността по отношение на времето е линейна по броя на кръстовищата.  
Сложността по отношение на паметта е линейна по височината на дървото.

---

## Упражнения

---

### Препоръчителни упражнения от учебника

Глава 12	12.2-1	12.2-7	12.3-4 *
12.1-1	12.2-2	12.2-8 *	12.3-5
12.1-2	12.2-3	12.2-9	12.3-6
12.1-3	12.2-4	12.3-1	Задача 12-1
12.1-4	12.2-5	12.3-2	Задача 12-2 (принадлежи към темата за символните низове)
12.1-5	12.2-6	12.3-3	

### Допълнителни упражнения

1. Дадено е бинарно дърво с  $n$  на брой кръстовища, като във всяко негово кръстовище има положително число.

Предложете рекурсивен алгоритъм за изчисление на най-тежкия маршрут от корена до листо – маршрут от корена до листо, за който сумата от стойностите на кръстовищата по дължината му е максимална! Предложете също и начин за отпечатването кръстовищата на този маршрут!

Какво би се променило, ако числата не бяха задължително положителни?

2. Един студент задействал проучвателни обхождания *in-order* и *pre-order* върху бинарното дърво, с което разполагал. След известно време той с учудване открил, че е загубил дървото, но все още разполага с резултатите от проучвателните обхождания.

Помогнете на студента да възстанови дървото! Каква е сложността на решението в най-лошия / най-добраия случай?

*in-order* : 2 6 4 7 1 3 8 5 9 10  
*pre-order* : 1 2 4 6 7 3 5 8 9 10

3. Можем да сортираме дадена група от  $n$  на брой числа по следния начин:

- отначало построяваме дърво за бинарно търсене, съдържащо тези числа, чрез вкарането на елементите един след друг в дървото.
- след това отпечатваме числата по получения ред чрез проучвателно обхождане *in-order* на дървото.

Какво е времето за изпълнение в най-лошия случай, в най-добраия случай и в средния случай на този алгоритъм за сортиране?

---

### Допълнителни упражнения

4. Докажете, че не съществува алгоритъм за изграждането на дърво за бинарно търсене от даден списък с  $n$  на брой елемента, чието време за изпълнение е  $O(n \log n)$ !

5. Друг вариант на упражнение 12.2-7 от учебника

Вътрешно сканиране (*in-order*) на дърво за бинарно търсене с  $n$  на брой кръстовища може да се осъществи чрез намирането на минимума и след това намирането на последователя на настоящото кръстовище  $n - 1$  пъти.

Следва изчисление на грешна плътна граница на времето за изпълнение в най-лошия случай:

- Височината на дървото е  $h = \Theta(n)$ .
- Всяко едно от действията Tree-Successor се изпълнява за време  $\Theta(h)$ .
- Затова общото време за изпълнение е  $(n - 1)\Theta(h) = \Theta(n^2)$ .

Обясните, къде е грешката, и, защо това изчисление пасва единствено на горна граница (подсказка: къде трябва да се смени  $\Theta$  с  $O$ ?)!

Обясните, защо на практика времето за изпълнение на този алгоритъм е  $\Theta(n)$ !

### Решение 1

```
Heavy-Weight( $x$ )
1. if  $x = \text{nil}$ 
2.   return 0
3.    $l \leftarrow \text{Heavy-Weight}(\text{left}[x])$ 
4.    $r \leftarrow \text{Heavy-Weight}(\text{right}[x])$ 
5.   return  $\text{key}[x] + \max\{l, r\}$ 
```

Дайте пример, в който решението не е вярно, ако са позволени и отрицателни числа по кръстовищата! Какво трябва да се промени в решението в такъв случай?

### Решение 4

Нека предположим, че съществува такъв алгоритъм.

Тогава можем да сортираме  $n$  на брой елемента за време  $O(n \log n)$  по следния начин:

- Ще изградим дървото с помощта на гореспоменатия алгоритъм за време  $O(n \log n)$ .
- Ще извършим проучвателно обхождане in-order по дървото за отпечатването на елементите му за време  $\Theta(n)$ .

Общо времевата сложност е  $O(n \log n)$  и това е в противоречие с долната граница на сортирането чрез сравнения, която доказахме.

### Решение 5

Намирането на последовател наистина се изпълнява в най-лошия случай за  $\Theta(h)$ , но по начина за сканиране, представен във въпроса, само в една част от случаите (пътите) се случва най-лошият случай, и много пъти намирането на последователя отнема по-кратко време (Дайте пример за състояние, в което намирането на последователя на едно кръстовище се изпълнява за константно време!). Тогава можем да поправим думите си и да кажем, че всяко едно действие относно последователя се изпълнява за време  $O(h)$ , и, че горната граница на общото време за изпълнение е  $O(n^2)$ .

Пътна граница:

Ще обърнем внимание, че редът за обхождане на дъгите и кръстовищата на дървото е напълно еднакъв с този от сканирането in-order. Тоест, обхождаме всяко кръстовище и всяка дъга еднакъв брой пъти и точно в същия ред. Известно е, че сканирането in-order на дърво се изпълнява за линейно време по броя на кръстовищата му.

## Тема 9

# AVL дървета AVL Trees

### В програмата:

Член 10.3 от книгата „Структури от данни“

Section 9.8 in “Data structures, algorithms and Software principles in C”

- Ще учим за AVL дървета – тип балансираны дървета за търсене.
- Ще се запознаем с действието „претъркуване“, чиято цел е да балансира дървото.

### Мотивация

### Напомняне

- Речникът е абстрактна структура от данни (ADT), определена от следните действия: Search, Delete, Insert, а понякога и от: Predecessor, Successor, Maximum, Minimum.
- Видяхме имплементация на речник посредством дърво за бинарно търсене. Сложността на следните действия е  $\Theta(h)$ , като  $h$  е височината на дървото.
  - $h = \Theta(\log n)$  – в най-добрия случай и средно
  - $h = \Theta(n)$  – в най-лошия случай
- Има няколко типа дървета за търсене, които се наричат балансираны:  $h = \Theta(\log n)$  – в най-лошия случай.
  - ❖ AVL дървета (в тази презентация)
  - ❖ Дървета „Две-Три“ (които са частен случай на дърветата B+)
  - ❖ Червено-черни дървета (глава 13 от учебника)
  - ❖ и още ...

### AVL дървета

AVL дърветата са измислени от двама учени, специалисти по компютърни науки: Adelson-Velsky, Landis.

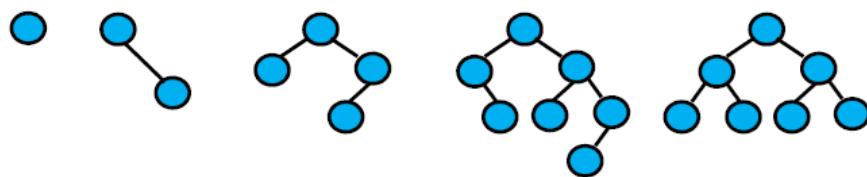
### Определение

AVL дървото е дърво за бинарно търсене, в което за всяко кръстовище  $v$  е изпълнено свойството:

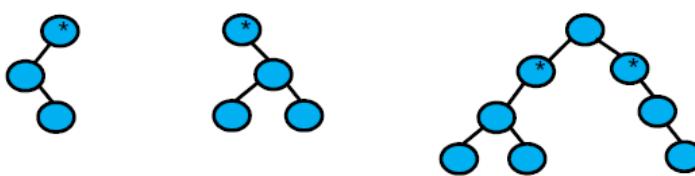
$$|h(left[v]) - h(right[v])| \leq 1$$

Напомняне: Височината на празно дърво се определя като -1.

Примери



Противоположни примери



\* кръстовища, в които балансът е нарушен

---

Граница на височината на AVL дърво

Твърдение

За едно AVL дърво с  $n$  на брой кръстовища и височина  $h$  е изпълнено  $h = \Theta(\log n)$ .

Доказателство

Долна граница: Какъв е максималният брой кръстовища в едно AVL дърво с височина  $h$ ?

$$n \leq 2^{h+1} - 1$$

$$h \geq \log(n+1) - 1$$

$$h = \Omega(\log n)$$

Горна граница: Какъв е минималният брой кръстовища в едно AVL дърво с височина  $h$ ?

Ще докажем, че  $n = \Omega(\Phi^h)$ , като  $\Phi$  е златното сечение:  $\Phi = \frac{1 + \sqrt{5}}{2}$

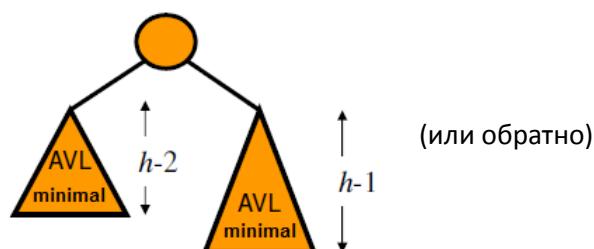
И от тук следва, че  $h = O(\log_{\Phi} n)$ .

---

Граница на височината на AVL дърво

Горна граница (продължение):

Как изглежда едно AVL дърво с височина  $h$  и с минимален брой кръстовища?



Дървета с такава структура се наричат **Дървета на Фибоначи**.

Припомняне: редицата на Фибоначи:  $f_1 = f_2 = 1$      $f_n = f_{n-1} + f_{n-2}$

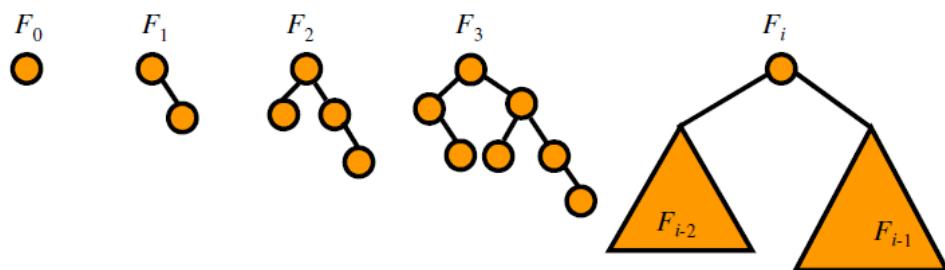
$$f_n = \frac{\Phi^n - \bar{\Phi}^n}{\sqrt{5}} \quad \text{като} \quad \Phi = \frac{1 + \sqrt{5}}{2} \approx 1.618 \quad \bar{\Phi} = 1 - \Phi = \frac{1 - \sqrt{5}}{2} \approx -0.618$$

---

### Граница на височината на AVL дърво

Горна граница (продължение):

Определение за дървото на Фибоначи посредством рекурсия:



Свойства: (упражнение: Докажете всяко едно от свойствата)

1. Височината на  $F_h$  е  $h$ .
  2.  $|F_h| = |F_{h-1}| + |F_{h-2}| + 1$  (като  $|F_i|$  е броят на кръстовищата в  $F_i$ ).
  3.  $F_h$  е AVL дървото с минимален брой кръстовища сред всички AVL дървета с височина  $h$ .
- 

### Граница на височината на AVL дърво

Горна граница (продължение):

Твърдение: Броят на кръстовищата в  $F_h$  е  $|F_h| = f_{h+3} - 1$  кръстовища, като  $f_i$  е  $i$ -тото число на Фибоначи.

Доказателство: чрез индукция върху  $h$ .

Основа: за  $h = 0$  и  $h = 1$  твърдението е изпълнено:



$$f_3 = 2 \quad f_4 = 3$$

Стъпка: Нека предположим, че твърдението е вярно за всяко  $h' < h$ .

$$|F_h| = |F_{h-1}| + |F_{h-2}| + 1 = (f_{h+2} - 1) + (f_{h+1} - 1) + 1 = f_{h+3} - 1$$

От какво следва всеки преход?

---

### Граница на височината на AVL дърво

Горна граница (продължение):

Ще обобщим: За AVL дърво с  $n$  на брой кръстовища и височина  $h$  е изпълнено:

(според свойство 3)  $n \geq |F_h|$

(според твърдението)  $n \geq |F_h| = f_{h+3} - 1 = \frac{\Phi^{h+3} - \bar{\Phi}^{h+3}}{\sqrt{5}} - 1 \geq \frac{\Phi^{h+3}}{\sqrt{5}} - 2$

тоест:  $\sqrt{5}(n + 2) \geq \Phi^{h+3}$

ще извадим  $\log_\Phi$ :  $h + 3 \leq \log_\Phi(\sqrt{5}(n + 2))$

$$h \leq \log_\Phi(n + 2) + \log_\Phi(\sqrt{5}) - 3$$

$$h = O(\log n)$$


---

### Времева сложност на действията върху AVL дърво

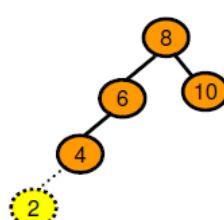
Извод: Всички заявки се изпълняват върху AVL дърво за логаритмично време по броя на кръстовищата му.

- AVL Search
- AVL Minimum, AVL Maximum
- AVL Successor, AVL Predecessor

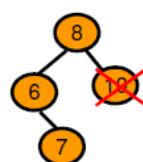
Ами вкарване и изкарване на елемент?

Също за логаритмично време, но тези действия биха могли да нарушият баланса на дървото.

Например: Добавяне на 2 към следното дърво:



Или изтриване на 10 от следното дърво:

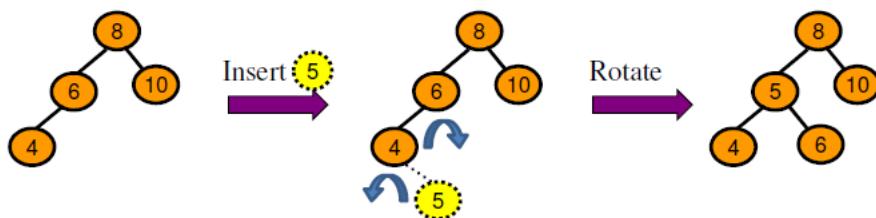
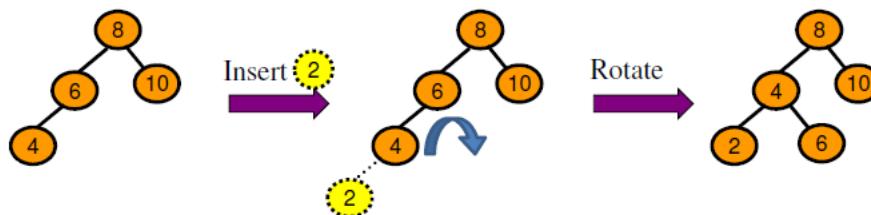


## Поправка след вкаране на елемент – претъркувания

Какво може да се направи, ако вкарането на елемент води до нарушаването на баланса?

Да се извърши претъркуване по дървото – промяна на няколко показалеца с цел да се балансира разликата във височините.

Ето няколко прости примера.



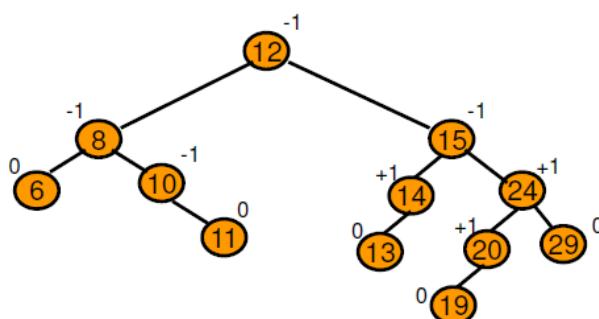
## Поправка след вкаране на елемент – претъркувания

Определение:

Балансиращият фактор (balance factor) на едно кръстовище е разликата между височината на лявото под-дърво и височината на дясното под-дърво на кръстовището.

$$BF[v] = h(left[v]) - h(right[v])$$

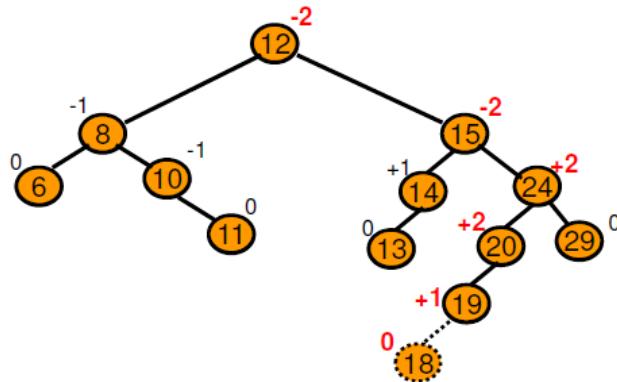
В едно нормално AVL дърво за всяко кръстовище  $v$  е изпълнено:  $|BF[v]| \leq 1$



### Поправка след вкаране на елемент – диагнози

Нека предположим, че сме вкарали 18 в дървото.

Въпрос:  
Кои стойности на BF са възможни сега?

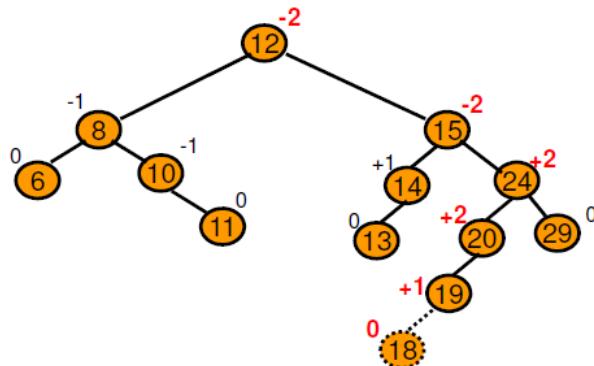


#### Диагноза №1

След вкаране балансиращият фактор не може да бъде по-голям от 2 по абсолютна стойност, тъй като се променя най-много с 1.

### Поправка след вкаране на елемент – диагнози

Въпрос: Кои са кръстовищата, при които е възможна промяна в балансиращия фактор?



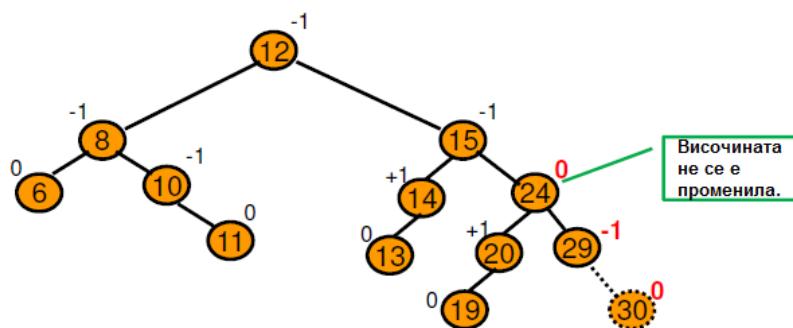
#### Диагноза №2

Единствените кръстовища, при които може би е нарушен балансиращият фактор са кръстовищата по дължината на маршрута на вкаране.

### Поправка след вкаране на елемент – диагнози

Въпрос: Как е възможно, да няма промяна във всички кръстовища по маршрута на вкаране?

Например, при вкаране на 30:



### Диагноза №3

Ако в гореспоменатия маршрут съществува кръстовище, чиято височина не се е променила вследствие вкарането на елемент, то балансиращите фактори в кръстовищата, които са над него по маршрута също не са се изменили.

### Поправка след вкаране на елемент – диагнози

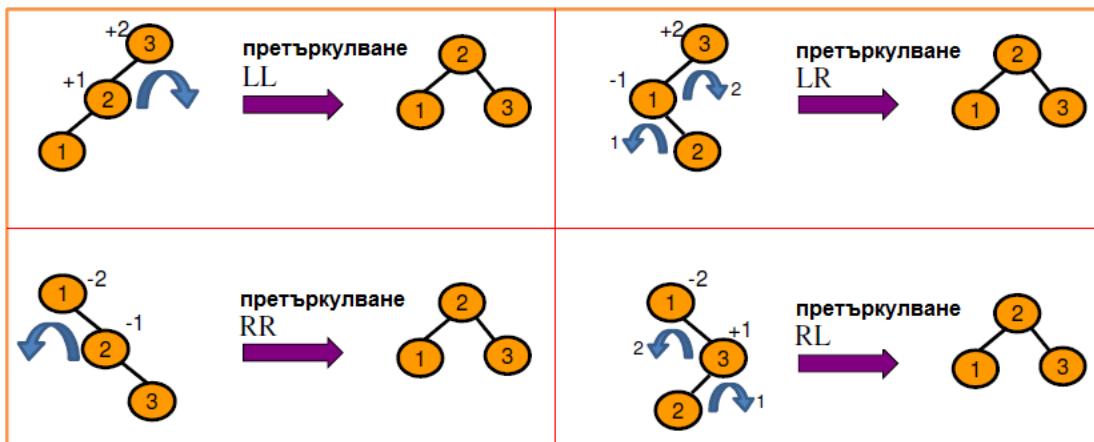
В светлината на трите диагнози следват първи линии при оформянето на алгоритъм за поправка след вкарането на елемент в AVL дърво:

- След добавянето на кръстовище в AVL дърво ще обходим кръстовищата му стапирайки от новото кръстовище нагоре в посока към корена.
- Ако балансиращият фактор в някое кръстовище не е в нормално състояние ( $\pm 2$ ) – ще извършим съответното претъркуване (както ще видим веднага).
- Ще приключим, когато стигнем до кръстовище, чиято височина е еднаква с височината му преди вкарането на елемента, или когато приключим с обработката на корена.

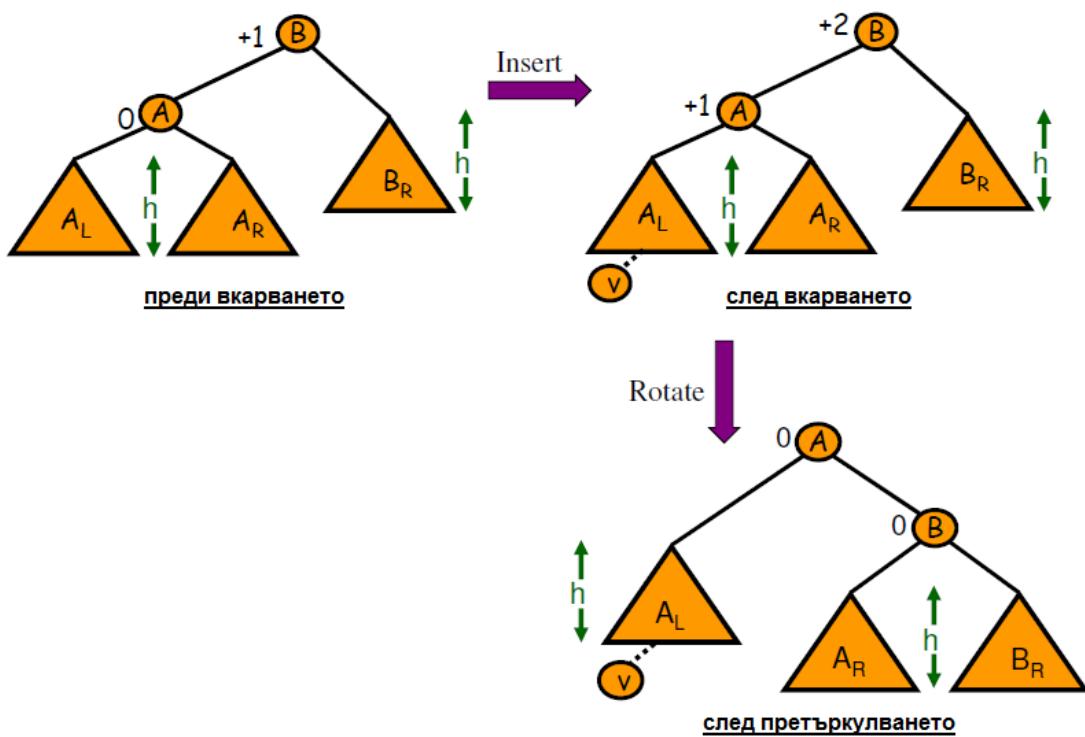
### Поправка след вкаране на елемент – претъркувания

Съществуват 4 вида претъркувания, съответстващи на 4-те възможни състояния на нарушен баланс\*:

BF[v]	BF на левия наследник	BF на десния наследник	Съответно претъркуване
+2	+1		LL
-2		-1	RR
+2	-1		LR
-2		+1	RL



### Претъркулване LL



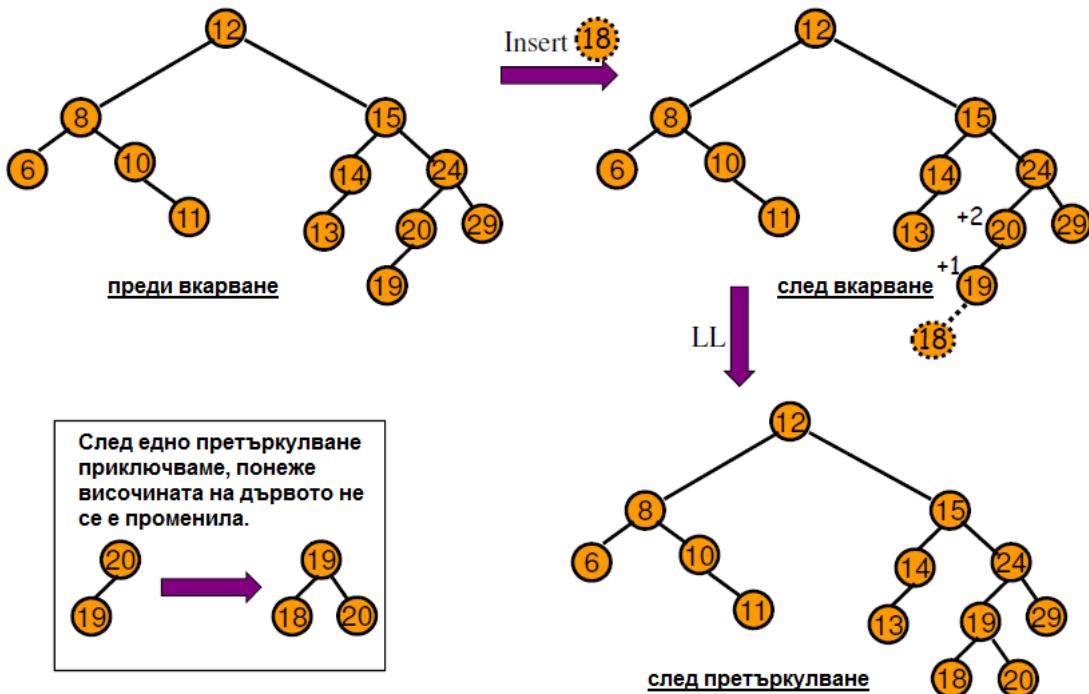
След претъркулването:

- Запазва се свойството на дървото за търсене.
- Нарушението на баланса е поправено в това под-дърво.

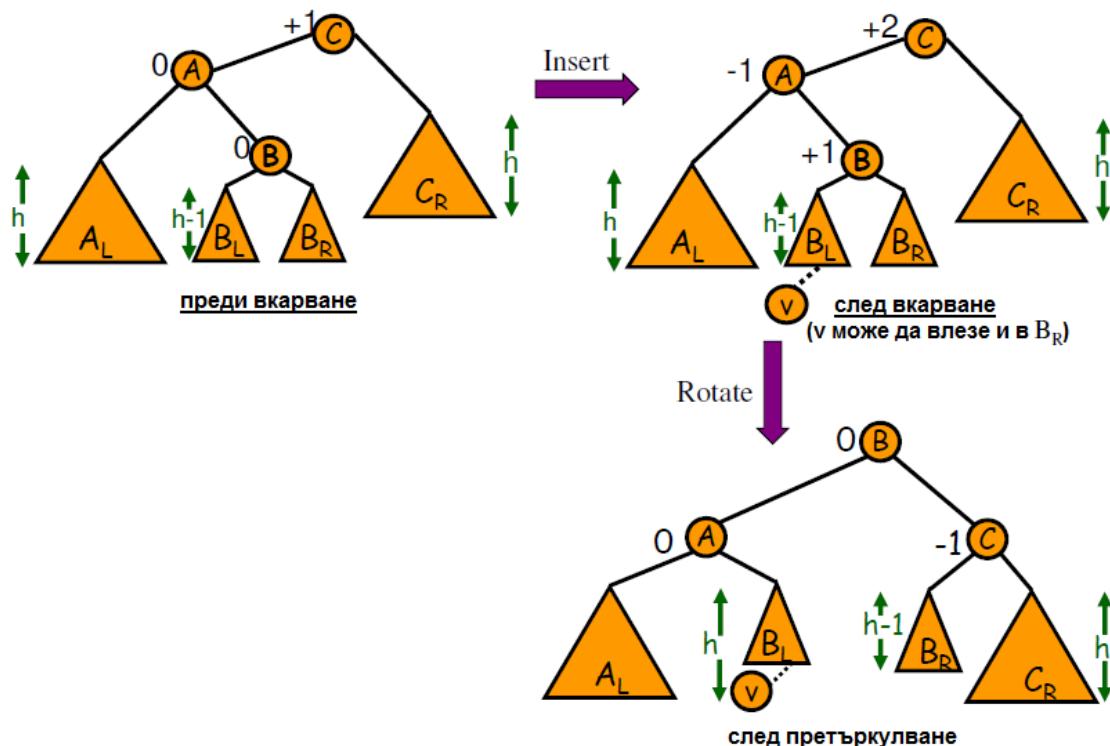
Претъркулването RR е симетрично.

### Претъркулване LL

#### Демонстрация



### Претъркуване LR



След претъркуването:

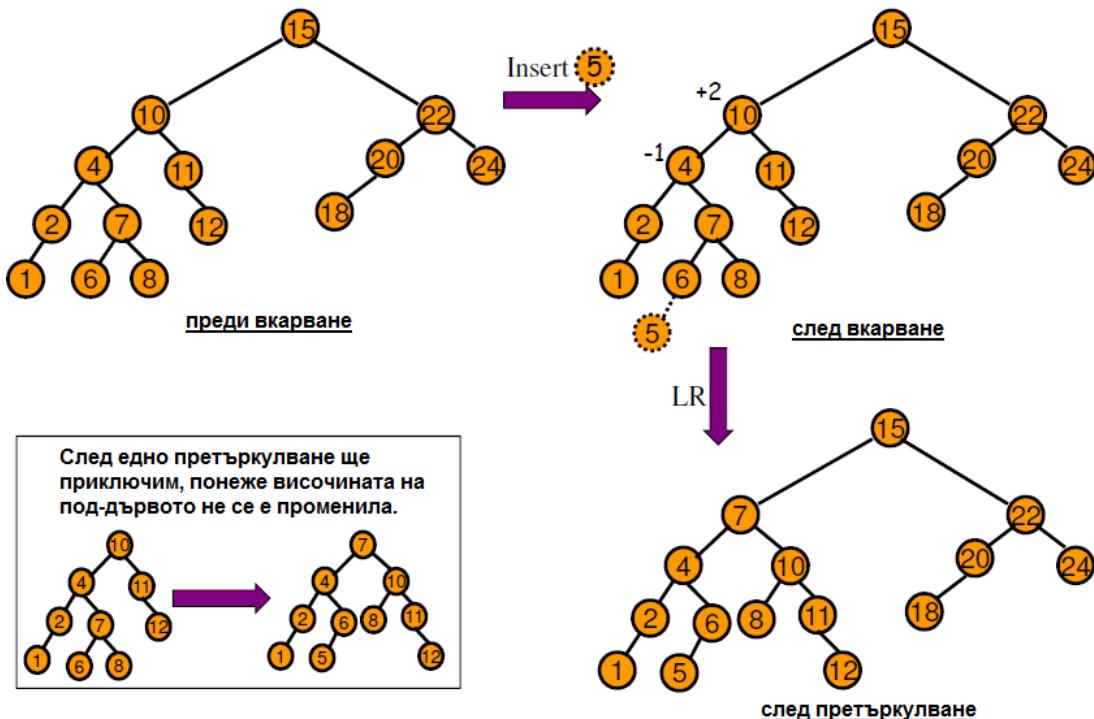
- Запазва се свойството на дървото за търсене.
- Нарушението на баланса е поправено в това под-дърво.

Претъркуването RL е симетрично.

с 20

## Претъркуване LR

### Демонстрация

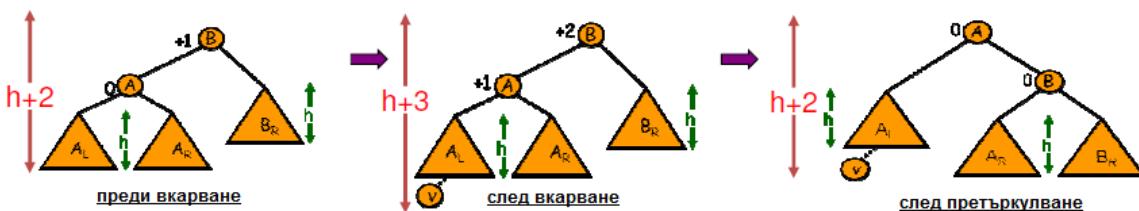


## Поправка след вкаране

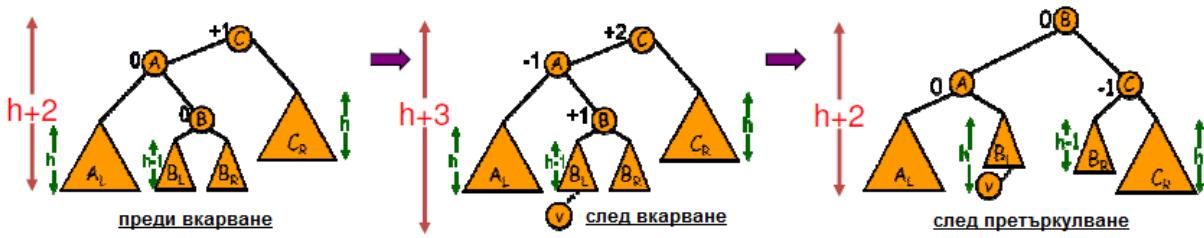
Твърдение: След вкаране на елемент е нужно най-много едно претъркуване за поправка на дървото.

Доказателство: Претъркуване вследствие вкаране на елемент винаги връща височината на под-дървото към първоначалната му височина преди вкарането.

### Претъркуване LL (RR е симетрично):



### Претъркуване LR (RL е симетрично):



### Алгоритъмът за вкаране на елемент

AVL-Insert( $T, z$ )

1. Вкарайте  $z$ , както обикновено (както при бинарно дърво за търсене)! Нека  $y$  е родителят на вкараното кръстовище.
2. Докато  $y \neq \text{Nil}$ , извършете:
  - 2.1. Изчислете  $\text{BF}[y]! *$
  - 2.2. Ако  $|\text{BF}[y]| < 2$  и височината на  $y$  не се е променила, приключете!
  - 2.3. Иначе, ако  $|\text{BF}[y]| < 2$ , и височината на  $y$  се е променила, преминете към следващата итерация с родителя на  $y$ !
  - 2.4. Иначе (тук  $|\text{BF}[y]| = 2$ ) извършете съответното претъркуване и приключете!

### Времева сложност

Обичайно вкаране на елемент в дърво за бинарно търсене

$\Theta(h)$

Намиране мястото на претъркуването (ако съществува)

$\Theta(h)$

Извършване на претъркуването (ако се извършва)

$\Theta(1)$

---

$\Theta(h) = \Theta(\log n)$

\* За това се изисква при всяко кръстовище да се запази височината му. Ще обърнем внимание на тази тема в следващата лекция.

### Поправка след изваждане на елемент

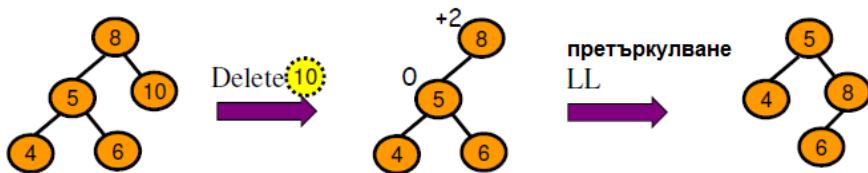
След изважддане на кръстовище също е възможно нарушение на балансиращите фактори.

Същите 4 типа претъркувания служат и тук за поправяне на нарушението.

<b>BF[v]</b>	<b>BF на левия наследник</b>	<b>BF на десния наследник</b>	<b>Съответно претъркуване</b>
+2	+1 или 0		LL
-2		-1 или 0	RR
+2	-1		LR
-2		+1	RL

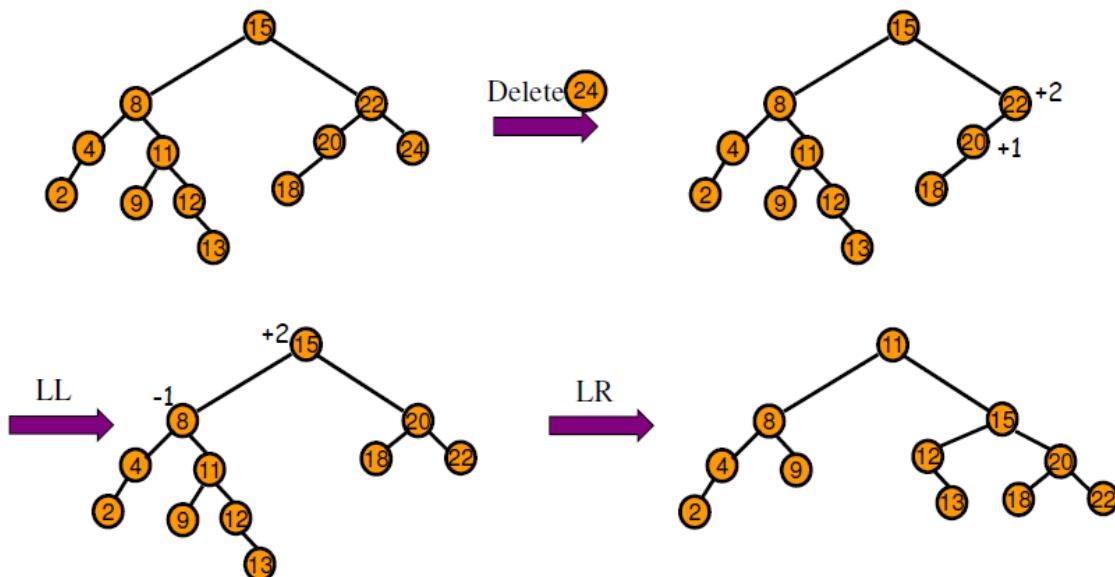
### Обърнете внимание на двете таблици!

При изважддане са възможни ситуации, които не са възможни при вкаране. Например:



### Поправка след изваждане на елемент

При изваждането на кръстовище е възможно повече от едно претъркулване (дори е възможно да се извърши по едно претъркулване на всяко ниво на дървото).



### Алгоритъмът за изваждане на елемент

#### AVL-Delete( $T, z$ )

1. Извадете  $z$ , както обикновено (както при бинарно дърво за търсене)! Нека  $y$  е родителят на кръстовището, което практически се изтрива.
2. Докато  $y \neq \text{Nil}$ , извършете:
  - 2.1. Изчислете  $\text{BF}[y]$ ! \*
  - 2.2. Ако  $|\text{BF}[y]| < 2$  и височината на  $y$  не се е променила, приключете!
  - 2.3. Иначе, ако  $|\text{BF}[y]| < 2$ , и височината на  $y$  се е променила, преминете към следващата итерация с родителя на  $y$ !
  - 2.4. Иначе (тук  $|\text{BF}[y]| = 2$ ) извършете съответното претъркулване и преминете към следващата итерация с родителя на  $y$ !

#### Времева сложност

Обичайно изваждане на елемент от дърво за бинарно търсене  
Най-много едно претъркулване на всяко ниво

$$\frac{\Theta(h)}{\Theta(h)} = \Theta(\log n)$$

\* За това се изисква при всяко кръстовище да се запази височината му. Ще обърнем внимание на тази тема в следващата лекция.

---

## Анимация

<http://people.ksp.sk/~kuko/bak/index.html>

---

## Въпроси за преговор

1. Посетете следния линк, който предлага отлична симулация на AVL дървета:  
<http://people.ksp.sk/~kuko/bak/index.html>

Постройте посредством симулацията дървото от слайд 12! Извършете няколко действия по вкарване и изтриване върху дървото – отначало сами върху хартия, а след това с помощта на симулацията за проверка на отговора ви!

2. Обяснете: Когато балансиращ фактор на кръстовище вследствие вкарване на елемент се променя от  $\pm 1$  на 0, тогава височината му не се променя. Можем ли да твърдим това и относно изваждане на елемент?

3. Както видяхме, вследствие изваждане на кръстовище от AVL дърво може да се получи ситуация (временна, преди поправката), в която имаме кръстовище с балансиращ фактор +2, а не ляв наследник с балансиращ фактор 0. Обяснете, защо такава ситуация не е възможна вследствие вкарване на кръстовище в AVL дърво!

4. Искаме да построим AVL дърво с целите числа от 1 до 7. Предложете такъв ред на вкарване, който да не изиска извършването на нито едно претъркуване! Предложете ред на вкарване, който да изиска максимален брой претъркувания!

5. Каква е формата на дървото, получено от вкарването на числата от 1 до 7 по реда им в AVL дърво?

---

## Отговори на въпросите за преговор

2. Ако балансирацият фактор на кръстовище  $v$  се променя вследствие вкарването от +1 на 0, то новото кръстовище е вкарано в дясното под-дърво, и сега височината му е равна на височината на лявото под-дърво. Затова височината на  $v$  не се променя. Твърдението е вярно и симетрично за другата ситуация.

При изваждане положението е обратното – именно промяна на балансирация фактор от 0 на  $\pm 1$  свидетелства за не-промяна на височината му.

3. Ако вследствие на вкарване в кръстовище  $v$  имаме балансиращ фактор +2, това свидетелства за това, че вкарването е извършено в лявото му под-дърво. Ако левият наследник на  $v$  има балансиращ фактор 0, то преди вкарването той е бил +1 или -1 (ако е бил 0, то той не се е променил вследствие на вкарването, и затова балансирацият фактор на  $v$  също не е трябвало да се променя). Според въпрос 2 височината на левия наследник не се е променила вследствие на вкарването, и затова балансирацият фактор на  $v$  също не е могъл да се промени.

4. Без претъркулвания, например: 4, 2, 6, 1, 3, 5, 7.

Максимален брой претъркулвания – сортиран ред или обратно сортиран ред (4 претъркулвания).

5. Пълно (завършено) дърво. Опитайте се да изпълните действията по вкарането сами, а след това проверете с помощта на една от анимациите!

.....

## Упражнения

.....

### Упражнения

1. Предложете оптимален от гледна точка на времевата сложност алгоритъм за построяването на AVL дърво от даден списък с  $n$  елемента. Докажете, че алгоритъмът ви е оптимален!

2. Покажете линеен алгоритъм за построяването на AVL дърво от сортиран масив A с  $n$  на брой елемента!

3. Дадени са две AVL дървета, всяко с  $n$  на брой кръстовища. Покажете, до колкото е възможно, ефективен алгоритъм за сливането на двете дървета в едно AVL дърво!

4. Дадени са някакви  $n$  на брой елемента. Всеки елемент има ключ и е известно, че има единствено  $\lfloor \log n \rfloor$  различни ключа. Предложете алгоритъм за сортиране на елементите според ключовете им, който се изпълнява за време  $\Theta(n \log \log n)$ !

.....

### Решение 1

1. Ще разгледаме първо директното решение: да вкараме елементите един след друг отначало в празно AVL дърво.

Времевата сложност:

$$\Theta\left(\sum_{i=1}^n \log i\right) = \Theta(\log n!) = \Theta(n \log n)$$

Съществува ли по-добро решение?

Нека предположим, че съществува решение, чиято сложност е по-добра –  $O(n \log n)$ .

Тогава можем също и да сортираме за време  $O(n \log n)$  чрез построяването на AVL дърво по гореспоменатия начин и след това проучвателно да го обходим *in-order* за линейно време, а това е в противоречие с долната граница на сортирането.

От тук следва, че гореспоменатият прост алгоритъм също е оптимален и от гледна точка на времевата сложност.

.....

### Решение 2 – първи начин

Ще заделим едно кръстовище за корена и в него ще складираме ключа в средното местоположение на масива (което е и медиана). След това рекурсивно ще построим две поддървета и ще ги свържем с корена.

```
Sorted-Array-2-AVL( $A, p, r$ ) ▶ first call  $p=1, r=n$ 
1. if  $p > r$  return nil
2. Create  $root$            ▶ a new node
3.  $mid \leftarrow \lfloor (p+r)/2 \rfloor$ 
4.  $key[root] \leftarrow A[mid]$ 
5.  $left[root] \leftarrow$  Sorted-Array-2-AVL ( $A, p, mid-1$ )
6.  $right[root] \leftarrow$  Sorted-Array-2-AVL ( $A, mid+1, r$ )
7. ▶ if AVL includes pointers to parents, update this info too
8. return  $root$ 
```

Съответното рекурентно отношение за времето за изпълнение е  $t(n) = 2t(n/2) + \Theta(1)$ , а асимптотното му решение е  $\Theta(n)$ .

---

### Решение 2 – втори начин

Ще построим (почти) завършено дърво, което, както е известно, е валидно AVL дърво (балансиращият фактор на всички кръстовища е 0 или 1).

Ще го направим така:

1. Ще изградим „скелет“ на (почти) завършено дърво, без стойности по кръстовищата, например, чрез рекурсия: ще заделим едно кръстовище за корена, след това рекурсивно ще построим лявото и дясното поддърво, ако трябва да съществуват (тоест, ако индексът на корена им не е по-голям от  $n$ ).

```
Build-Empty-AVL( $i, n$ )      ▶ first call  $i=1$ 
1. if  $i > n$  return nil
2. Create  $root$            ▶ a new node
3.  $left[root] \leftarrow$  Build-Empty-AVL( $2i, n$ )
4.  $right[root] \leftarrow$  Build-Empty-AVL( $2i+1, n$ )
5. ▶ if AVL includes pointers to parents, update this info too
6. return  $root$ 
```

2. Ще обходим масива чрез проучвателно обхождане in-order на дървото. Действието Visit ще бъде превърнато в записване на элемента от масива в кръстовището в дървото.  
Тук също времевата сложност е  $\Theta(n)$ .
- 

### Решение 3

Наивно решение: Обхождаме всички елементи на едно от дърветата и ги вкарваме един по един в другото дърво.

Сложност: Вкарваме  $n$  на брой пъти елемент в AVL дърво, чийто първоначален размер е  $n$ , а финалният му размер е  $2n$ .

Тоест, всяко вкарване на елемент се извършва за  $O(\log 2n)$  и за  $\Omega(\log n)$ , тоест за  $\Theta(\log n)$ .  
Общо  $\Theta(n \log n)$ .

По-ефективно решение:

- ✓ Ще извършим проучвателно обхождане in-order върху всяко едно от дърветата и ще запишем елементите му в масив.
- ✓ Ще слеем двата масива (обърнете внимание, че масивите са сортирани!).
- ✓ Ще построим от слетия масив, чийто размер е  $2n$ , AVL дърво по метода описан в предишния въпрос.

Сложност: Всички етапи се изпълняват за линейно време, тоест  $\Theta(n)$ .

-----

Решение 4

Структурата от данни:

Ще поддържаме AVL дърво със следните промени: Във всяко кръстовище в дървото ще има ключ и свързан списък на елементите притежаващи този ключ.

Алгоритъмът:

Ще обходим входните данни и ще вкараме елементите един след друг в дървото със следните промени:

1. За елемента  $x$  с ключ  $k$ , ако в дървото все още няма кръстовище с ключа  $k$ , ще добавим такова кръстовище и ще инициализираме показалец от него към празен свързан списък. Иначе няма нужда да добавяме кръстовище.
2. Ще вкараме  $x$  в началото на свързания списък на съответното кръстовище.

На края ще обходим проучвателно дървото in-order, като при всяко кръстовище ще обходим списъка му и ще извадим в изходните данни елементите, които се намират в него.

Времева сложност:

Както е известно, вкарването на елемент в AVL дърво се изпълнява за линейно време по височината му. В нашия случай дървото нараства, но крайният му размер е  $\lfloor \log n \rfloor$  кръстовища, и затова всяко вкарване ще се изпълни за време  $O(\log \log n)$ , а всички вкарвания – за  $O(n \log \log n)$ . Проучвателното обхождане на края, включително обхожданията на списъците, ще се изпълни за време  $\Theta(\log n + n) = \Theta(n)$ .  
Общо:  $O(n \log \log n)$ .

-----

## Тема 10

### Разширяване на структури от данни и дървета от степени

### Augmenting Data Structures and Rank Trees

#### В програмата:

- Ще учит как се разширяват съществуващи и познати структури от данни, за да паснат на решението на нови задачи.
  - Ще го демонстрираме чрез дървета от степени – разширение на AVL дървата.
- 

#### Мотивация

- В прости случаи абстрактните структури от данни (ADT) могат да се имплементират ефективно посредством позната структура от данни (кулчина, масив, AVL дърво, ...)
- Често се изисква интеграция на структури от данни
- Има случаи, в които ефективната имплементация е възможна чрез разширение на познати структури от данни.
- Например:

Нека предположим, че искаме да имплементираме речник –  $\text{Insert}(S, x)$ ,  $\text{Delete}(S, x)$ ,  $\text{Search}(S, k)$ , който поддържа също и следните действия:

- $\text{Select}(S, i)$  – връщане на най-малкия  $i$ -ти елемент в  $S$
- $\text{Rank}(S, x)$  – връщане на степенуването на  $x$  сред елементите на  $S$  (степенуването на един елемент е местоположението му в сортирания ред)

Например:

$$S = \{1, 5, 3, 6, 22, 10\}$$

$$\text{Rank}(S, 10) = 5$$

$$\text{Select}(S, 4) = 6$$

Забележка: С цел улеснение предполагаме, че елементите са различни един от друг.

---

## Познати решения

### AVL дърво

- Действията по речника ще се имплементират както обикновено за логаритмично време.
- Select( $S, i$ ) – ще извършим сканиране in-order по дървото и ще върнем елемента  $i$ , когото посещаваме.
- Rank( $S, x$ ) – по подобен начин чрез сканиране in-order. Ще преброим колко кръстовища сме посетили до достигането на  $x$ .  
Последните две действия се изпълняват в най-лошия случай, когато  $i = n$ , **за линейно време**.

### Сортиран масив

- Търсене за логаритмично време (бинарно търсене).
- Select( $S, i$ ) – ще достъпим елемента с индекс  $i$  за константно време.
- Rank( $S, x$ ) – ще изчислим индекса (= степенуването) на  $x$  за константно време (чрез просто изчисление на разликата в адресите).
- Но имаме вкаране и изтриване на елемент **за линейно време**.

### Купчина

- Търсене, Select, Rank **за линейно време** ...

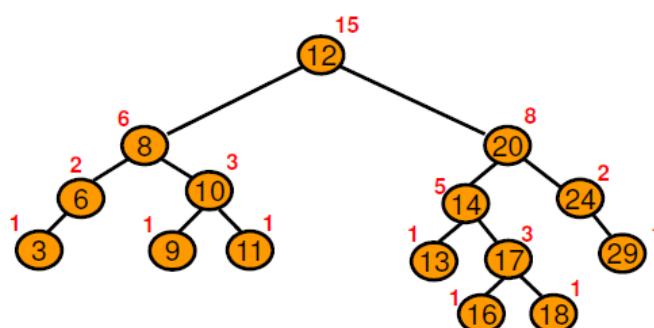
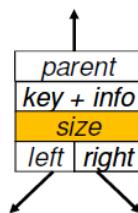
## По-ефективно решение – разширено AVL дърво

Сега ще покажем решение, базирано на разширението на AVL дърветата.  
Всички действия ще се изпълняват за логаритмично време.

Ще използваме AVL дърво, в което към всяко кръстовище ще добавим едно поле – **size**.

Това поле ще съдържа броя кръстовища в под-дървото на това кръстовище (включително и самото кръстовище).

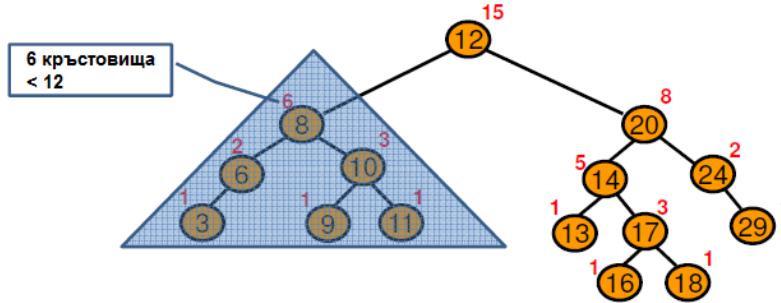
Такова дърво се нарича **дърво от степени** (rank tree).



Как тази допълнителна информация помага за имплементацията на Select и Rank?

## Tree Select

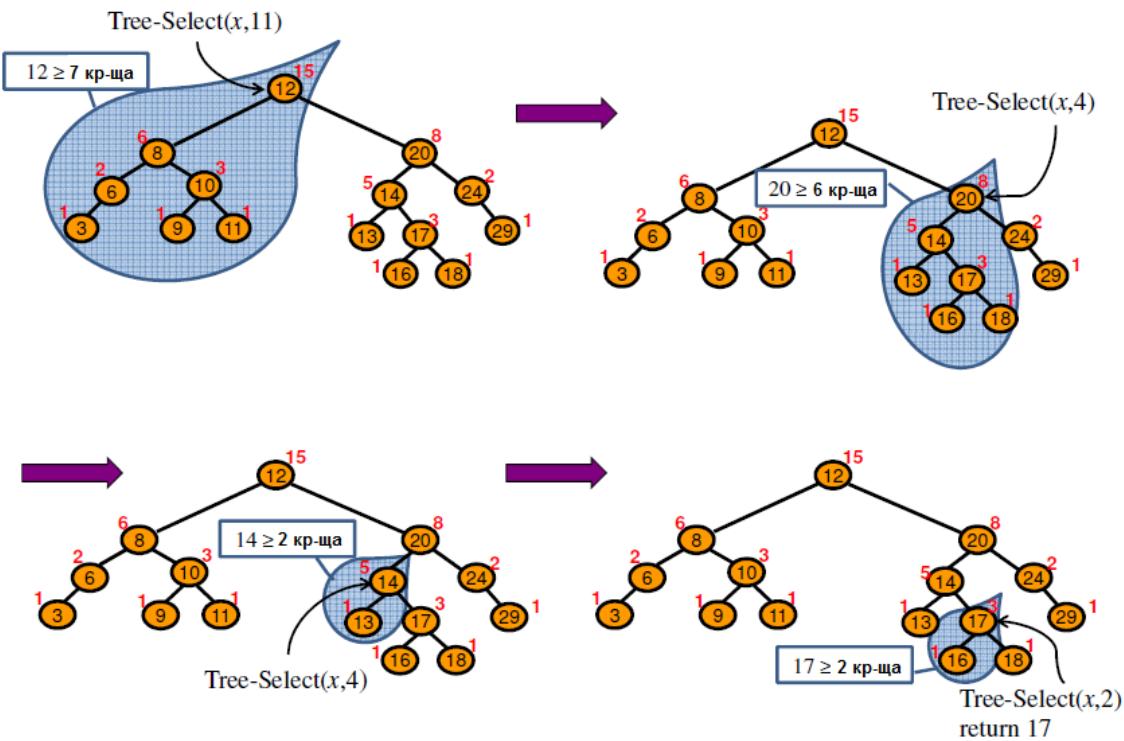
Ще обърнем внимание, че коренът на дървото е 7-ият най-малък елемент.



- Ако търсим най-малкия елемент  $i = 1$ , то това е коренът.
- Иначе, ако  $i < 7$ , ще търсим в лявото под-дърво на корена най-малкия  $i$ -ти елемент.
- Иначе (ако  $i > 7$ ), ще търсим в дясното под-дърво на корена най-малкия  $i-7$ -ми елемент.

## Tree Select

Ще демонстрираме Tree-Select( $s, 11$ ):



## Tree Select

Алгоритъмът:

```
Tree-Select( $x, i$ )
1.  $r \leftarrow \text{size}[\text{left}[x]] + 1$ 
2. if  $i = r$ 
3.   return  $x$ 
4. else if  $i < r$ 
5.   return Tree-Select( $\text{left}[x], i$ )
6.   else return Tree-Select( $\text{right}[x], i - r$ )
```

При първото извикване  $x$  е коренът.

Времева сложност:

На всяко ниво от дървото „прахосваме“ константно време. Затова времевата сложност е линейна по височината на дървото –  $\Theta(\log n)$ .

Сложност по отношение на допълнителната памет:

Стекът на рекурсията –  $\Theta(\log n)$ . Но е възможно да се имплементира итеративна версия с допълнителна памет  $\Theta(1)$ .

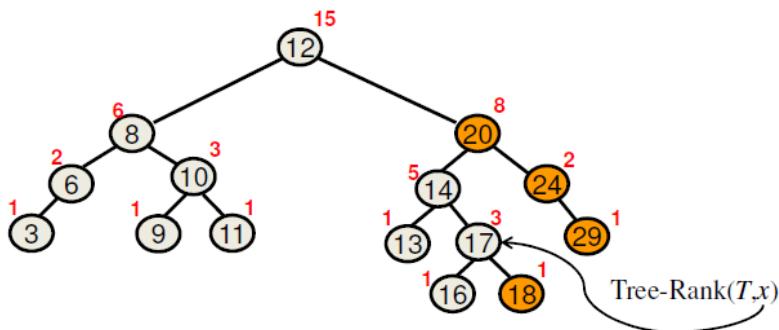
---

## Tree Rank

Как ще намерим степенуването на елемент при даден показалец към него?

Пример: Ще намерим степенуването на 17.

Белите кръстовища са тези, които са по-малки или равни на 17.



Идеята:

Първо ще преброим, колко кръстовища има в лявото под-дърво на 17 плюс 1 (за самото 17).

След това ще се покатерим от  $x$  до корена:

Всеки път, когато се качваме наляво от кръстовището, ще добавяме броя на кръстовищата в лявото под-дърво плюс 1.

## Tree Rank

### Алгоритъмът:

Т е коренът на дървото.  $x$  сочи към елемента, чието степенуване искаме да намерим.

```
Tree-Rank( $T, x$ )
1.  $r \leftarrow size[\text{left}[x]] + 1$ 
2.  $y \leftarrow x$ 
3. while  $y \neq \text{root}[T]$ 
4.   if  $y = \text{right}[\text{parent}[y]]$ 
5.      $r \leftarrow r + size[\text{left}[\text{parent}[y]]] + 1$ 
6.    $y \leftarrow \text{parent}[y]$ 
7. return  $r$ 
```

### Времева сложност:

На всяко ниво от дървото „прахосваме“ константно време. Затова времената сложност е линейна по височината на дървото –  $\Theta(\log n)$ .

Сложност по отношение на допълнителната памет:  $\Theta(1)$ .

с 10

---

## Tree Rank Key

### Въпрос:

Искаме да върнем степенуването на елемент с даден ключ от дърво от степени. Предложете решение на задачата!

### Отговор:

**Възможност А:** Ще намерим елемента, притежаващ дадения ключ и ще прехвърлим показалеца, сочещ към него, в Tree-Rank.

```
Tree-Rank-Key( $T, k$ )
1.  $x \leftarrow \text{AVL-Search}(T, k)$ 
2. return Tree-Rank ( $T, x$ )
```

Време:  $\Theta(\log n)$ .

Допълнителна памет: Ако използваме итеративна версия за търсене,  $\Theta(1)$ .

**Възможност Б:** Ще инициализираме  $r \leftarrow 0$ . Ще слезем от корена по маршрута за търсене на ключа и всеки път, когато слизаме надясно, ще добавяме към  $r$  броя кръстовища в лявото поддърво плюс 1. Ще го направим отново, когато стигнем до търсеното кръстовище.

Време:  $\Theta(\log n)$ .

Допълнителна памет:  $\Theta(1)$ .

---

## Поддържане на полето size по време на вкаране и изкаране

До сега видяхме, как добавянето на полето позволява да се осъществят действията Select и Rank за логаритмично време.

Сега трябва да покажем, че по време на вкаране или изкаране на елементи може да се актуализира това поле, без да се засяга сложността на действията по вкарането и изкарането!

---

## Поддържане на полето size по време на вкаране

Вкарането на елемент в AVL дърво е съставено от два етапа:

Етап 1 – слизане от корена надолу и вкаране на ново кръстовище.

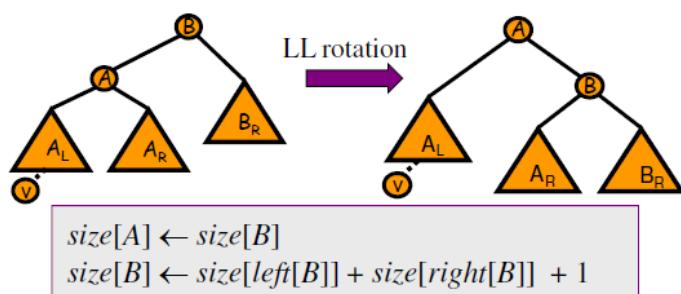
Етап 2 – качване от новото кръстовище в посока към корена с цел откриване на „AVL нарушители“ и може би извършване на едно претъркулване.

Как да актуализираме size на всеки един от тези етапи?

Етап 1 – Ще добавим 1 към полето size на всяко кръстовище, през което сме преминали (при новото кръстовище  $size[z] \leftarrow 1$ ).

Етап 2 – Актуализация на  $\Theta(1)$  кръстовища.

Например, при претъркулване LL:



Три всяко едно от останалите претъркувания актуализираме по подобен начин.

Актуализациите изискват добавяне на константи на всяко ниво в дървото и следователно не променят времевата сложност на вкарането.

---

## Поддържане на полето size по време на изваждане

Изваждането на елемент от AVL дърво е съставено от два етапа:

Етап 1 – обичайно изтриване от дървото за бинарно търсене.

Етап 2 – качване от новото кръстовище в посока към корена с цел откриване на „AVL нарушители“ и може би извършване на претъркувания.

Как да актуализираме size на всеки един от тези етапи?

Етап 1 – Нищо.

Етап 2 – По време на качването от кръстовището, което физически е било изтрито, ще извадим 1 от полето size на всяко кръстовище, през което сме минали. Ако са били извършени претъркувания в същото това време, ще актуализираме  $\Theta(1)$  кръстовища при всяко претъркуване (точно като при вкарването).

Актуализациите изискват добавяне на константи на всяко ниво в дървото и следователно не променят времевата сложност на изваждането.

---

## Разширяване на структура от данни

Ще обобщим това, което сме направили до сега:

Поискахме имплементация на абстрактна структура от данни (ADT), която не се поддържа за логаритмично време от нито една „проста“ структура от данни, която ни е позната, или от интеграция на такива.

За тази цел:

- |   |                        |
|---|------------------------|
| 1. Избрахме някаква <u>основа</u> на позната структура от данни   | AVL                    |
| 2. <u>Разширихме</u> я с някаква добавка                          | полето size            |
| 3. Уверихме се, че динамичните действия <u>не са засегнати</u>    | вкарване и изкарване   |
| 4. Показахме, как се имплементират <u>допълнителните действия</u> | Tree-Select, Tree-Rank |
- 

## Допълнителен проблем

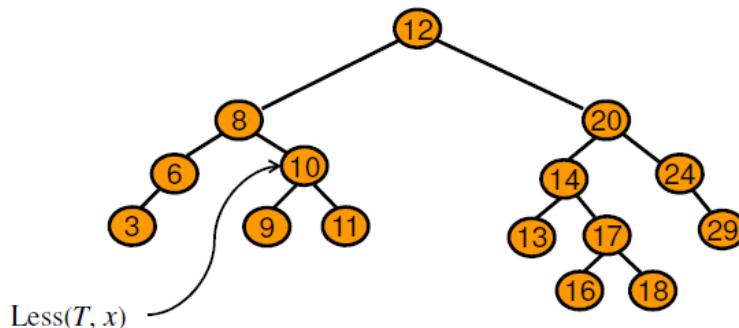
Нека предположим, че искаме да имплементираме речник без повторения, поддържащ също и следните действия:

- $\text{Less}(S, x)$  – връщане сумата на ключовете, които са по-малки/равни на ключа на елемента  $x$ .

Тук също си струва да използваме AVL дърво за основа.

Как да имплементираме Less?

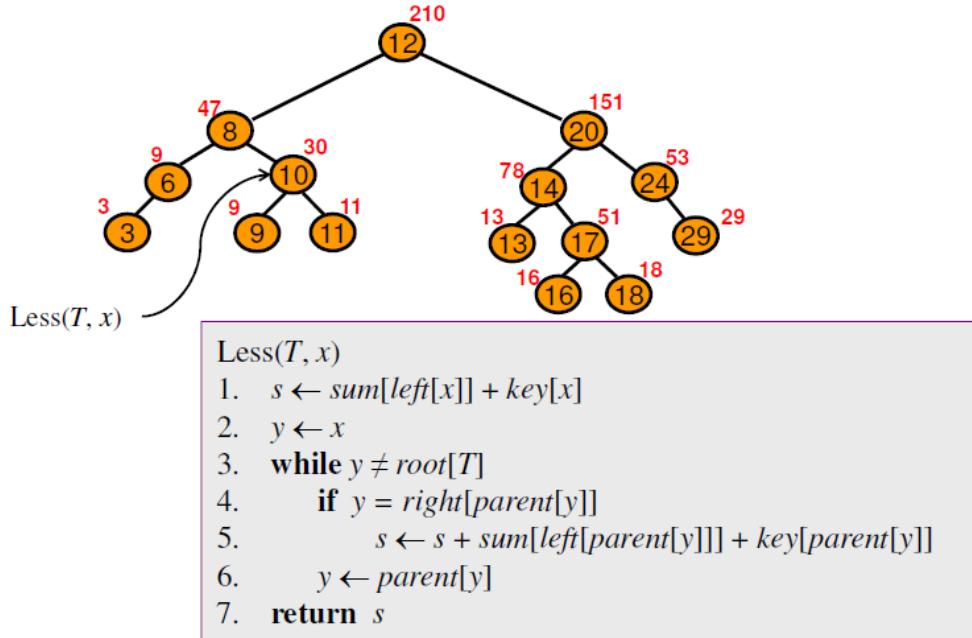
Например:



## Имплементация на Less

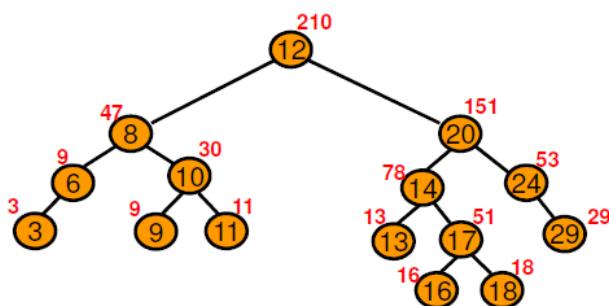
Идеята: AVL дърво, в което при всяко кръстовище има допълнително поле – sum, което съдържа сумата от ключовете в под-дървото на кръстовището (включително).

Имплементацията на Less прилика много на Rank, която видяхме.



## Поддръжка на sum

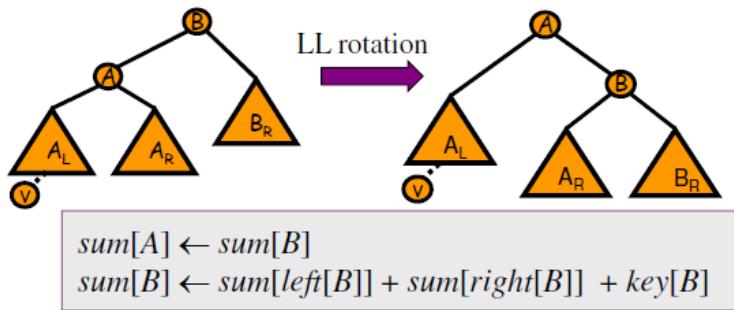
Сега трябва да покажем, че можем да поддържаме полето sum без да засягаме сложността на вкарването и изкарването.



Вкарване:

Етап 1 – по време на слизането ще добавим ключа на новия елемент към всяко кръстовище, през което сме преминали (при новото кръстовище  $sum[z] \leftarrow key[z]$ ).

Етап 2 – ще демонстрираме върху претъркуване LL:



### Поддръжка на sum

Изваждане:

Етап 1 – Нищо.

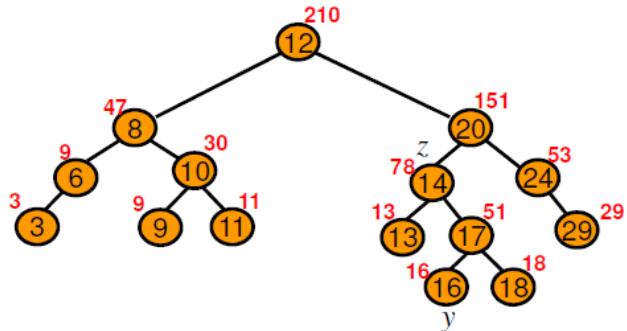
Етап 2 – **Ако** кръстовището, на което му е било изтрито z, е имало най-много един наследник:

- Ще се качим от изтритото кръстовище до корена и ще извадим key[z] от полето sum на всяко кръстовище

**Иначе:** Ще обозначим с у физически изтритото кръстовище (обърнете внимание, че ключът на z се е променил!).

- Ще извадим key[y] от всички кръстовища, които са между у и z (без да включваме у и z)
- От всички останали кръстовища, започвайки от z и стигайки до корена, ще извадим key[z].

За претъркуванията ще се погрижим, както при вкарването.



### Какво можем да поддържаме ефективно?

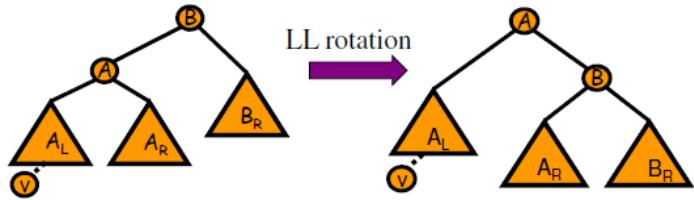
Въпрос:

Можем ли да поддържаме ефективно (без да засягаме времевата сложност на вкарването и изкарването) полета, съдържащи дълбочините на кръстовищата в AVL дървото?

Отговор:

Не. Има случаи (например, претъркувания, изтриване), вследствие на които трябва да се актуализират полетата за дълбочините на  $\Theta(n)$  кръстовища!

Например, дълбочините на кои кръстовища се променят при претъркуване LL?



Затова сложността на вкарването/изваждането пострадва.  
с 20

### Какво можем да поддържаме ефективно?

#### Теорема

Нека  $f$  е поле, разширяващо AVL дървото  $T$  с  $n$  на брой кръстовища.

Ако промяна в складираната информация в дадено кръстовище (включително и неговото поле  $f$ ) оказва влияние единствено върху верността на  $f$  на прародителите му, то тогава можем да поддържаме стойностите на  $f$  при вкарване и изкарване без да влияем върху асимптотното време за изпълнение  $\Theta(\log n)$  на тези действия.

#### Забележки:

- Теоремата отбелязва единствено достатъчно условие, но не и задължително условие.
- Теоремата не предоставя алгоритъм за актуализация на полетата, само отбелязва, кога може да се направи това.

### Обобщение

Видяхме как се разширява AVL дърво като пример за разширяване на структури от данни.

Това е методология, чрез която може да се разшири всяка структура от данни, която сме учили, като се изпълнят 4-те етапа, които видяхме.

За да знаем, коя структура от данни да изберем, и как да я разширим, понякога се изисква и не малко изобретателност ...

### Въпроси за преговор

1. Покажете, как трябва да се актуализират полетата  $size$  на дърво от степени вследствие на претъркуване LR!
2. Напишете нерекурсивна версия на Tree Select!
3. Обяснете, какви са необходимите промени за имплементацията на действията върху дърво от степени, ако при всяко кръстовище запазваме броя на кръстовищата в под-дървото на кръстовището, без да включваме самото кръстовище!

4. Обяснете, какви са необходимите промени за имплементацията на действието Less, когато допълнителното поле, което запазваме, съдържа сумата от кръстовищата в лявото под-дърво на кръстовището (вместо във всяко негово под-дърво)!

---

### Отговори на въпросите за преговор

2.

Iterative-Tree-Select( $T, i$ )

1.  $x \leftarrow \text{root}[T]$
2.  $r \leftarrow \text{size}[\text{left}[x]] + 1$
3. **while**  $i \neq r$
4.     **if**  $i < r$
5.          $x \leftarrow \text{left}[x]$
6.     **else**  $x \leftarrow \text{right}[x]$
7.          $i \leftarrow i - r$
8.      $r \leftarrow \text{size}[\text{left}[x]] + 1$
9. **return**  $x$

### Упражнения

#### Упражнения

1. При дадени елемент  $x$  в дърво от степени  $s$  на брой кръстовища и естествено число  $i$ , как можем да намерим  $i$ -тия следващ  $x$  за време  $O(\log n)$ ?

2. Как може да се имплементира речник, в който действията на следващия се изпълняват за време  $O(1)$ , а всички останали действия по речника за логаритмично време по броя на елементите в структурата?

3. Предложете имплементация на структура от данни, поддържаща следните действия ( $n$  е броят на елементите в даден момент):

- Insert( $x$ ) – добавяне на елемента  $x$  в структурата за време  $O(\log n)$
- Delete( $x$ ) – изтриване на елемента  $x$  от структурата за време  $O(\log n)$
- Find( $k$ ) – намиране на елемент с ключ  $k$  в структурата за време  $O(\log n)$
- Min() – намиране на елемента с минимален ключ в структурата за време  $O(1)$
- Between( $k_1, k_2$ ) – връщане на броя ключове между  $k_1$  и  $k_2$  (включително) за време  $O(\log n)$

### Решение 1

Tree-Successor-i( $T, x, i$ )

1.  $r \leftarrow \text{Tree-Rank}(T, x)$
2. if  $r+i > n$
3.     **return** Nil
4. **return** Tree-Select( $T, r+i$ )

Дали решението със задействането на Tree-Successor (започвайки от  $x$ )  $i-1$  пъти също е вярно?

---

### Решение 2

Ще използваме разширено AVL дърво, като при всяко кръстовище ще запазим допълнително и показалец succ към следващия му елемент (при максималното кръстовище този показалец ще съдържа Nil).

Действието относно следващия елемент може лесно да се изпълни посредством този показалец за време  $O(1)$ .

Поддръжка на показалеца succ при вкарване: ще добавим тези редове в края на обикновения алгоритъм за вкарване:

( $z$  е новото кръстовище)

```
succ[z]  $\leftarrow$  Tree-Successor( $T, z$ )
pre  $\leftarrow$  Tree-Predecessor( $T, z$ )
if pre  $\neq$  Nil
    succ[pre]  $\leftarrow z$ 
```

По време на претъркулванията няма нужда от извършването на никаква допълнителна актуализация (защо?).

Поддръжка на показалеца succ при изваждане: трябва да разделим на няколко случая.

Ако изтрием кръстовище с най-много един наследник – ще насочим succ на предходния елемент към следващия го.

Иначе? ...

---

### Решение 3

Ще използваме дърво от степени, както видяхме на лекцията (AVL дърво с допълнително поле size). Освен това ще запазим показалец към най-малкия елемент в дървото (който ще се актуализира по време на вкарване и изкарване, както ще обясним по-късно).

- Insert( $x$ ) – ще вкараме елемент в дърво от степени, както видяхме в слайдовете –  $O(\log n)$  и, ако ключът на вкарания елемент е по-малък от минимума, ще актуализираме показалеца сочещ към минимума –  $O(1)$ .

- $\text{Delete}(x)$  – ще извадим елемент от дърво от степени, както видяхме в слайдовете –  $O(\log n)$  и, ако има нужда, ще намерим новия минимум чрез извикване на AVL-Minimum –  $O(\log n)$ .
- $\text{Find}(k)$  – както обикновено при AVL дърво –  $O(\log n)$ .
- $\text{Min}()$  – ще върнем минималния елемент с помощта на показалеца за време  $O(1)$ .
- $\text{Between}(k_1, k_2)$  – ще използваме действието Tree-Rank, което връща степенуването на даден елемент.
  1. Не знаем, дали в структурата съществуват елементи с дадените ключове, затова първо ще го проверим, и, ако не съществуват, ще ги вкараем (ще ги наречем  $x_1$  и  $x_2$ ).
  2. Ще изчислим  $\text{Tree-Rank}(x_2) - \text{Tree-Rank}(x_1) + 1$ .
  3. Ще извадим от получената стойност броя на елементите (между 0 и 2), които сме вкарали, и това е търсеният резултат.
  4. На края, ако има нужда, ще изтрием елементите, които сме вкарали.

Всички гореспоменати действия се изпълняват за време  $O(\log n)$ .

.....

## Тема 11

# Хеш (хаотични) таблици Hash tables

### В програмата:

Глава 11 от учебника

- Ще се запознаем с още една имплементация на речника – хеш (хаотични) таблици \* (hash tables).
- Това е средно ефективна имплементация – всички действия ще се изпълняват за константно време.

\* понякога се наричат и **смесителни таблици**.

### Мотивация

Видяхме имплементация на речник посредством балансирано дърво за търсене, в което всички действия се изпълняват за време  $O(\log n)$ , като  $n$  е броят на елементите в структурата.

Възможно ли е да се имплементира речник с по-добра времева сложност?

Отговорът е да!

Може да се имплементира речник чрез масив (при положение, че е възможно да се картографира областта от ключовете в индекси на масива).

Всички действия ще се изпълняват за константно време.

Защо, ако въобще, използваме дървета за търсене?

Тъй като размерът на областта би могъл да бъде много голям.

Пример 1: Речник, в който ключът е номерът на личната карта, съставен от 9 десетични числа. Масивът ще съдържа  $10^9$  клетки, докато в страната имаме около от  $10^7$  жители (използване на около 1% от масива).

Пример 2: Английски речник. Броят стойности от оригинала в него е много малък спрямо броя стойности от превода от a до z с дължина до около 35.

От Уикипедия: Дължините на дългите думи в различните езици: [http://en.wikipedia.org/wiki/Longest\\_words](http://en.wikipedia.org/wiki/Longest_words)

## Мотивация

Сега ще дадем определение за **хеш таблици** (hash tables) и **хеш функции** (hash functions).

Това е един вид обобщение на идеята за масива.

Вместо да запазим по една клетка за всеки потенциален елемент, ще картографираме света на елементите в относително малка таблица и ще изчислим за всеки ключ индекс му в таблицата.

Както ще видим, можем да се погрижим, всички действия да се изпълняват средно за време  $\Theta(1)$ .

---

## Хеш таблици – определение

Даден е свят от елементи  $U$ .

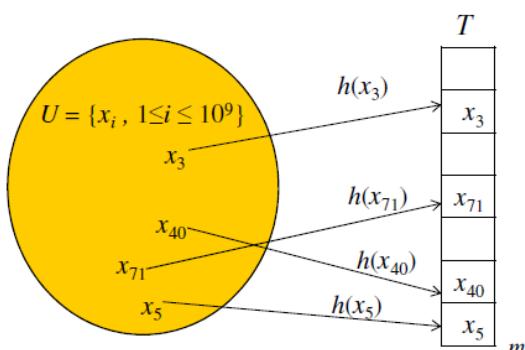
Изиска се имплементация на речник, в който във всеки даден момент да има  $O(n)$  елемента, като  $n = o(|U|)$ .

Ще заделим масив  $T$ , който ще се назава **хеш таблица**.

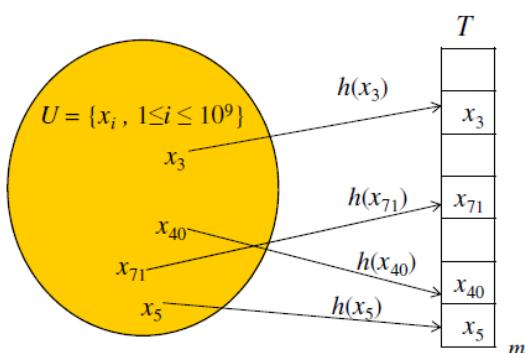
Размерът му ще бъде отбелаязан с  $m$ . Идеята е, че  $m$  може да бъде близко до  $n$ , но не и до  $|U|$ .

Ще дадем определение за **хеш функция**:  $h: U \rightarrow \{0, 1, \dots, m-1\}$

Елементът  $x \in U$  ще бъде картографиран в  $T[h(x)]$ .



## Проблеми



### Проблеми

- 1) Какво представлява една добра хеш функция?
- 2) Какво правим в случай на сблъсък?

Сблъсък: За  $x \neq y$  е изпълнено  $h(x) = h(y)$ .

---

### Какво следва?

#### 1. Какво представлява една добра хеш функция?

Ще дадем определение за това и ще покажем няколко метода за избор на добра хеш функция.

#### 2. Какво правим в случай на сблъсък?

Ще се запознаем с два подхода:

1. Метода на верижното хеширане
2. Метода на отвореното адресиране, с 3 възможности:
  - a. Линейна проверка
  - b. Квадратна проверка
  - c. Двойно хеширане (натрупване)

---

### Избор на добра хеш функция

Кои свойства бихме искали да изпълнява хеш функцията?

- 1) Времето за изчисление на функцията да е  $O(1)$ .
- 2) Функцията да „разпръсква добре“ ключовете в таблицата.

Формално: Функцията изпълнява **хипотезата за простото еднакво (униформено) хеширане**:

- Вероятността един ключ да се хешира (натрупа, позиционира) в дадена клетка е еднаква за всички клетки.
- Няма зависимост между стойностите от хеширането на различните ключове.

Съществуването на свойство 2 би ли предотвратило сблъсък?

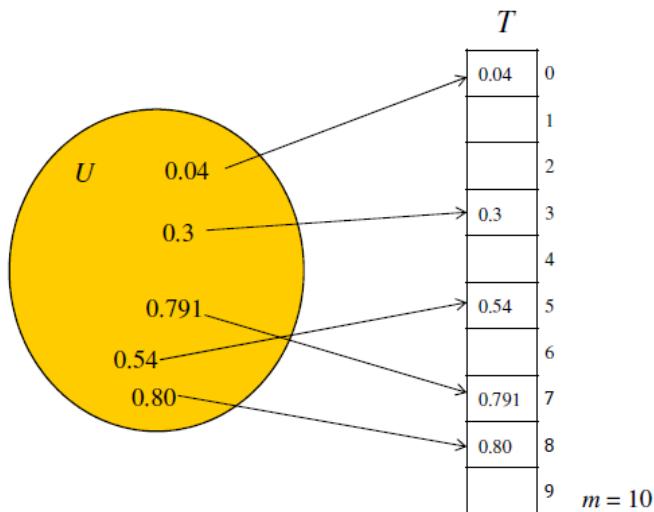
---

### Избор на добра хеш функция

Пример за добра хеш функция, като ключовете са реални числа с еднакво (униформено) разпределение, независимо в интервала  $[0, 1]$ .

$$h(k) = \lfloor km \rfloor$$

Например:



### Избор на добра хеш функция

Информацията за разпределението на ключовете помага при планирането на добра хеш функция. Но не винаги знаем, какво е разпределението на ключовете.

Все пак има методи за избор на хеш функция, които обикновено дават „добро“ разпръскване. Сега ще се запознаем с два такива метода:

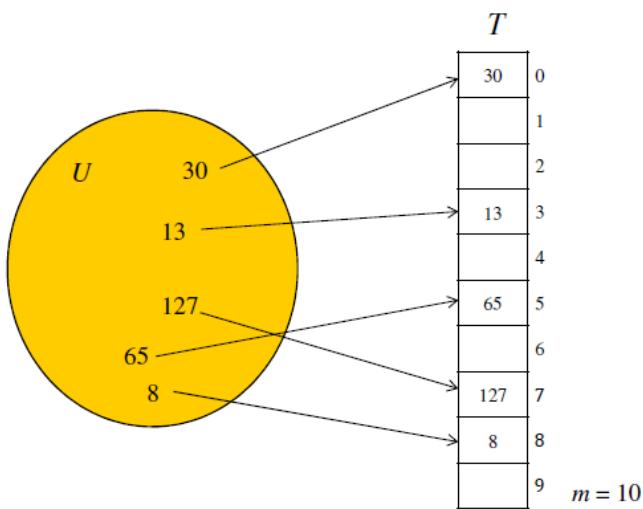
1. Метода на делението
2. Метода на умножението

Идеята е, да се извърши такова изчисление, което се очаква да не зависи от съществуващите шаблони в (статистическото) разпределение(то) на ключовете.

с 10

### Методът на делението

$$h(k) = k \bmod m$$

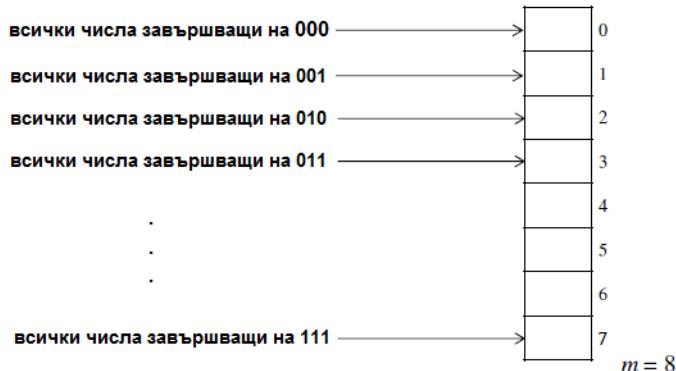


## Методът на делението

$$h(k) = k \bmod m$$

Какъв недостатък съществува при избора на  $m$ , което е целочислена степен на 2.

Подсказка: Помислете за бинарното представяне на ключовете.



Ако  $m = 2^p$ , само по-малко смислените  $p$  на брой бита се взимат предвид при изчислението стойността на хеширането.

Извод: По метода на делението е препоръчително, да не се избира  $m = 2^p$ , освен ако е известно предварително, че има еднаква вероятност за появата на всеки един от възможните шаблони на по-малко смислените  $p$  на брой бита.

.....

## Методът на делението

Изводът е верен не само за основа 2:

- Ако ключовете са числа с някаква основа  $b$ , изборът  $m = b^p$  създава хеширане, което се съобразява само с  $p = \log_b m$  на брой по-малко смислените цифри на ключа.
  - Препоръчително е стойността на хеширането да зависи от, колкото се може, повече информация от ключа.
- .....

## Методът на умножението

$$h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor, \text{ като } A \text{ е константа от интервала } 0 < A < 1.$$

Изборът на конкретната стойност на  $A$  зависи от характеристиките на ключовете.

Идеята е, стойностите на  $kA - \lfloor kA \rfloor$  да се разпределят еднакво в интервала  $[0, 1)$ .

По метода на умножението стойността на  $m$  по принцип не страда от ограниченията, които видяхме преди това при метода на делението.

### Упражнение

Дадена е хеш таблица с размер  $m = 10$  и хеш функция по метода на умножението с  $A = (\sqrt{5} - 1) / 2$ . Изчислете местоположенията, към които са картографирани ключовете: 61, 62, 63, 64, 65.

---

### Ключове, които не са числа

Трите метода за избор на хеш таблица, които видяхме, предполагаха, че ключовете са числа.

Какво се случва, когато това не е така?

Например, когато ключовете са символни низове?

Тогава ще „преведем“ ключовете към числа.

Например: ще използваме ASCII кодовете на буквите и така всяка буква ще бъде преведена към число с основа 128:

$a = 97, b = 98, c = 99, \dots$

“ab” =  $97 * 128 + 98 = 12514$

Има множество други много по-усъвършенствани методи.

Изборът на подходящия метод зависи до голяма степен от информацията относно характеристиките на ключовете.

---

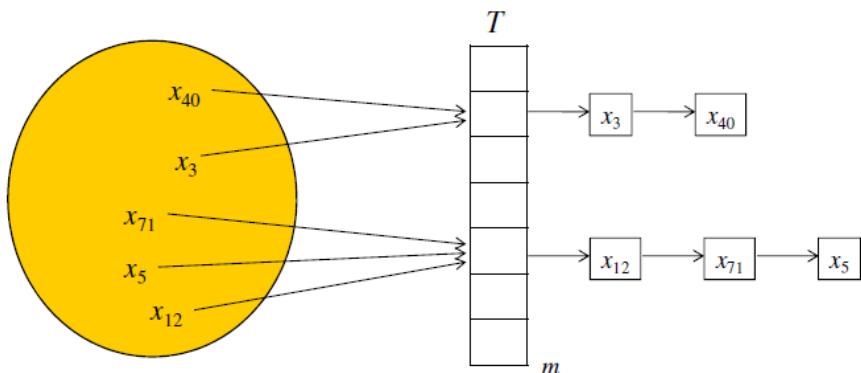
### Решения за сблъсъците

Сега ще представим два метода за решаване на проблема със сблъсъците:

- 1) Метода на верижното хеширане (Chain hashing)
  - 2) Метода на отвореното адресиране (Open addressing)
- 

### Решения за сблъсъците – Методът на верижното хеширане

Всяка клетка в  $T$  сочи към свързан списък. Ключ  $x$  ще се намира в свързания списък в клетка  $T[h(x)]$ .



Hash-Insert( $x$ ) – вкарай  $x$  в началото на списъка в клетка  $T[h(x)]$ !

Hash-Search( $k$ ) – търси ключа  $k$  в списъка в клетка  $T[h(x)]$ !

Hash>Delete( $x$ ) – изтрий  $x$  от свързания списък в клетка  $T[h(x)]$ !

.....

### Методът на верижното хеширане – сложност

#### Анализ на сложността

#### Най-лошия случай

Всички елементи са вкарани в един и същи списък.

Hash-Insert( $x$ ) –  $\Theta(1)$  \* при положение, че не се налага отначало да се проверява, дали елементът вече съществува.

Hash-Search( $k$ ) –  $\Theta(n)$

Hash>Delete( $x$ ) –  $\Theta(1)$  \* при положение, че списъците са двупосочни и е даден показалец към елемента за изтриване, иначе  $\Theta(n)$ .

#### Упражнение

Предложете промяна в метода на верижното хеширане, която да позволи вкарване, търсене и изтриване за време  $\Theta(\log n)$  в най-лошия случай!

.....

### Методът на верижното хеширане – сложност

#### Анализ на средната сложност под хипотезата за простото еднакво разпръскване

Ще зададем фактора на натоварването (load factor):  $\alpha = n / m$

Това на практика е средната дължина на списъка.

Сега ще докажем, че когато сблъсъците се решават по метода на верижното хеширане, търсенето се изпълнява средно за време  $\Theta(1 + \alpha)$ .

Следователно, ако изберем размера на таблицата  $m$ , така че да удовлетворява  $n = O(m)$ , то  $\alpha = O(m) / m = O(1)$  и всички действия ще се изпълняват средно за време  $\Theta(1)$ .

.....

### Методът на верижното хеширане – сложност

#### Теорема

По метода на верижното хеширане и под хипотезата за простото еднакво разпръскване търсенето на ключ ще се изпълнява средно за време  $\Theta(1 + \alpha)$ .

## Доказателство

Ще обърнем внимание първо на проваленото търсене (ключът не е намерен).

При хипотезата за простото еднакво разпръскване всеки ключ пристига случајно в един от  $m$ -те на брой списъка.

- Търсенето на някакъв ключ  $k$  изисква обхождане на списъка  $T[h(k)]$  до края.
- Средната дължина на този списък при хипотезата за простото еднакво разпръскване е  $\alpha = n/m$ .
- Следователно трябва да се обходят средно  $\alpha + 1$  показалеца (включително и този на края на списъка).
- Останалите действия (изчислението на хеш функцията и достъпа до съответната клетка) ще се изпълнят за време  $\Theta(1)$  в най-лошия случай.
- Общо средно  $\Theta(1 + \alpha)$ .

с 20

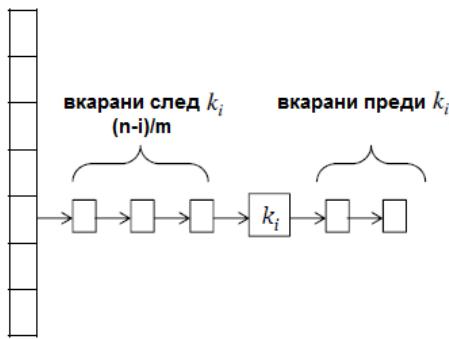


## Методът на верижното хеширане – сложност

### Доказателство (продължение)

Сега ще обърнем внимание на случая с успешното търсене на ключ.

Нека предположим, че редът за вкарване на елементите е бил  $k_1, \dots, k_n$ .



- След някакъв ключ  $k_i$  са добавени към структурата  $n-i$  на брой ключа.
- Затова средно дължината на списъка до  $k_i$  е  $(n-i)/m$ .
- От тук следва, че средното време за търсене на ключа  $k_i$  е  $1 + (n-i)/m$ .
- Ще изчислим средната стойност базирана върху всички ключове.

$$\begin{aligned} t &= \frac{1}{n} \sum_{i=1}^n \left( 1 + \frac{n-i}{m} \right) = 1 + \frac{1}{n \cdot m} \sum_{i=1}^n (n-i) = 1 + \frac{1}{n \cdot m} \sum_{i=0}^{n-1} i = 1 + \frac{1}{n \cdot m} \frac{(n-1)n}{2} \\ &= 1 + \frac{n-1}{2m} = 1 + \frac{\alpha}{2} - \frac{\alpha}{2n} = \Theta(1 + \alpha) \end{aligned}$$



## Методът на верижното хеширане – обобщение

Действие	Най-лош случай	Средно
Търсене	$\Theta(n)$	$\Theta(1 + \alpha)$
Вкаране	$\Theta(1) *$	$\Theta(1) *$
Изтриване	$\Theta(1) **$	$\Theta(1) **$

Ако  $n = O(m)$ , то всички действия ще се изпълняват средно за време  $\Theta(1)$ .

\* При положение, че не е нужно да се проверява, дали елементът вече съществува. Иначе времето е равно на времето за търсене.

\*\* При положение, че списъците са двупосочни и е даден показалец към елемента за изтриване.

---

## Решения за сблъсъците – отворено адресиране

- По метода на отвореното адресиране (Open Addressing) ако определена клетка е заета, търсим друга свободна клетка.  
Тоест, всяка клетка съдържа информация за най-много един ключ.  
Това изиска  $n \leq m$ , тоест  $\alpha \leq 1$ .
- Хеш функцията получава като параметър също и броя на предишните опити и връща следващото местоположение за проверка:  $h : U \times \{0, 1, \dots, m-1\} \rightarrow \{0, 1, \dots, m-1\}$   
Тоест, поредицата проверки за свободна клетка е  $h(k, 0), h(k, 1), \dots, h(k, m-1)$  и ще искаме тази поредица да бъде стойността (върнатия резултат) на  $\{0, 1, \dots, m-1\}$
- Сега ще покажем 3 типа проверки, които работят по метода на отвореното адресиране:

1. линейна проверка, 2. квадратна проверка, 3. двойно хеширане

---

## Линейна проверка

Хеш функцията:

$$h(k, i) = (h'(k) + i) \bmod m$$

като  $h'$  е „обикновена“ хеш функция  
(Метода на делението, Метода на умножението, други методи ...)

По този метод, ако определена клетка е заета, ще пробваме следващата клетка след нея.

Демонстрация:

Методът на делението:

$$h'(k) = k \bmod 17$$

Линейна проверка:

$$h(k, i) = (h'(k) + i) \bmod 17$$

Входните данни:

$$\text{Input: } 3, 5, 8, 13, 21, 55, 76, 131, 207, 338$$

0		
1		
2		
3	3	3
4	21	4
5	5	5
6	55	4
7	207	3
8	8	8
9	76	8
10		
11		
12	131	12
13	13	13
14		
15	338	15
16		

Предимство: Лесен за имплементация, създава се стойност (върнат резултат) на {0, 1, ..., m-1}.

Недостатък: Проблемът с **първичното натрупване**: Ако една заета клетка се предхожда от i заети клетки, шансът, тя да бъде следващата клетка, която ще се запълни, е  $(i+1)/m$ , а не  $1/m$ . Получават се дълги „последователности“ от заети клетки.

---

### Отворено адресиране – вкарване и търсене

#### Вкарване на елемент с ключ k

1.  $i \leftarrow 0$
2.  $j \leftarrow h(k, i)$
3. **Ако** клетката  $T[j]$  е свободна,
4.     Вкарваме в  $T[j]$  новия елемент и **връщаме j**
5. **Иначе:**  $i \leftarrow i + 1$
6.     **Ако**  $i < m$ , се връщаме на стъпка 2
7.     **Иначе връщаме „пълна таблица“**

#### Търсене на елемент с ключ k

1.  $i \leftarrow 0$
2.  $j \leftarrow h(k, i)$
3. **Ако** елементът в клетката  $T[j]$  е с ключ k, **връщаме j**
4. **Иначе ако**  $T[j]$  е свободна – връщаме „Ключът не бе намерен.“
5. **Иначе:**  $i \leftarrow i + 1$
6.     **Ако**  $i < m$ , се връщаме на стъпка 2
7.     **Иначе връщаме „Ключът не бе намерен.“**

### Отворено адресиране – изтриване

Какъв проблем се пробужда при изтриването?

- Ако просто изтрием един елемент от таблицата чрез поставяне на nil на мястото му. най-вероятно „ще прекъснем верижно търсене“.

Например:

Delete(21) и след това Search(207).

Търсенето ще стигне до първата празна клетка и ще се провали.

#### Решение на проблема:

Вместо да поставяме nil, ще поставим в клетката специална стойност **deleted**, чието значение е: „Клетката е свободна за вкарване и заета за търсене.“

#### Недостатъкът:

След множество изтривания таблицата би била претоварена с изтрити стойности (**deleted**) и времето за търсене ще бъде относително по-дълго за броя елементи, които се намират на практика в структурата.

0	
1	
2	
3	3
4	21
5	5
6	55
7	207
8	8
9	76
10	
11	
12	131
13	13
14	
15	338
16	

→ Затова, ако имаме множество изтривания, решението за сблъсъците по метода на верижното хеширане по принцип е за предпочитане.

## Отворено адресиране – продължение

Сега ще покажем два други метода за проверки, които се опитват да подобрят метода на линейната проверка:

- Квадратна проверка
- Двойно хеширане

Алгоритмите за вкарване, търсене и изтриване не са се променили (те са верни за общата парадигма на отвореното адресиране).

## Квадратна проверка

Хеш функцията:

$$h(k, i) = (h'(k) + ai + bi^2) \bmod m$$

като  $a$  и  $b \neq 0$  са константи

Демонстрация:

Методът на делението:

$$h'(k) = k \bmod 17$$

Квадратна проверка:

$$h(k, i) = (h'(k) + i + i^2) \bmod 17$$

Входните данни:

Input: 3, 5, 8, 13, 21, 55, 76, 131, 207, 338

Предимство: Не се получават „последователности“.

Недостатък: Проблемът с **второстепенното натрупване**: Все пак, ако за два ключа  $k_1 \neq k_2$  е изпълнено  $h'(k_1) = h'(k_2)$ , то и за двета ключа ще се извърши абсолютно същата поредица от проверки. Това е по-умерена форма на натрупване.

0	
1	
2	
3	3
4	21
5	5
6	55
7	
8	8
9	207
10	76
11	
12	131
13	13
14	
15	338
16	

## Двойно хеширане

Хеш функцията:

$$h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m$$

$h_1$  се нарича „функция на основата“  
 $h_2$  се нарича „функция на стъпката“

Демонстрация:

Методът на делението:

$$h_1(k) = k \bmod 17$$

Двойно хеширане:

$$h_2(k) = k \bmod 16 + 1$$

$$h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod 17$$

Входните данни:

Input: 3, 5, 8, 13, 21, 55, 76, 131, 207, 338

Предимство: Поредицата от проверки зависи не само от първичното местоположение  $h(k, 0) = h_1(k)$ , а също и от ключа.

0	76	8,13
1	207	3,16
2	3	3,4
3	21	4,6
4	5	5,6
5		
6		
7		
8	8	8,9
9		
10		
11		
12	55	4,8
13	13	13,14
14		
15	338	15,3
16	131	12,4

### Двойно хеширане – избор на стъпковата функция

$$h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m$$

В последния пример функцията на основата беше  $h_1(k) = k \bmod 17$   
 А функцията на стъпката беше  $h_2(k) = k \bmod 16 + 1$

Зашо избрахме така функцията на стъпката?

Отговор

a. Какъв е проблемът при следната функция на стъпката?  $h_2(k) = (k \bmod 17)$

b. Какъв е проблемът при следната функция на стъпката?  $h_2(k) = (k \bmod 17) + 1$

с 30

### Двойно хеширане – избор на размера на таблицата

$$h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m$$

Как си струва да изберем размера на таблицата  $m$ ?

Искаме поредицата от проверки да извърши обхождане на всички клетки в масива ( $h$  ще бъде „сюрективна (surjective)“ за всеки ключ). Тоест:

- За всяко  $k$   $h_2(k) \neq 0$
- За всяко  $k$  за  $h_2(k)$  и за  $m$  няма общи делители  $> 1$   
 Например, ако  $m = 20$ , не може  $h_2$  да получава (за нито един ключ  $k$ ) някоя от стойностите: 2, 4, 5, 6, 8, 10, 12, 14, 15, 16, 18, 20, ...

За да отговаряме на тези ограничения, едната от възможностите е да изберем за  $m$  просто число и да дадем определение:

$h_1(k) = k \bmod m$   
 $h_2(k) = (k \bmod (m - c)) + 1$   
за малко число  $c > 0$  (например,  $c = 1$  или  $c = 2$ ).

---

### Сравнение между методите за проверка

Ще се опитаме да разберем, защо двойното хеширане има по-добро изпълнение отколкото другите два метода за проверка.

#### Упражнение

а – Обяснете, защо при идеалното свободно адресиране имаме  $m!$  на брой различни възможни поредици от проверки!

Колко на брой различни възможни поредици от проверки имаме:

б – при линейната проверка?

в – при квадратната проверка?

г – при двойното хеширане?

#### Решение

а – тъй като всяка стойност (върнат резултат) на  $(0, 1, \dots, m-1)$  представлява поредица от проверки.

б+в – първоначалното местоположение определя поредицата от проверки. Затова са възможни единствено  $m$  различни поредици от проверки.

г - поредицата от проверки се определя както от първоначалното местоположение, така и от размера на стъпката.

За първоначалното местоположение имаме  $m$  на брой възможности.

За размера на стъпката имаме  $m-1$  на брой възможности (когато  $m$  е просто число, всяка стъпка с размер от 1 до  $m-1$  ще свърши работа).

Затова имаме  $\Theta(m^2)$  на брой различни възможни поредици от проверки.

Затова двойното хеширане представлява по-добро приближение до идеалното отворено адресиране отколкото линейната / квадратната проверка.

---

### Отворено адресиране с двойно хеширане – упражнение

#### Упражнение

Дадена е хеш таблица с размер 7, с решение за сблъсъците по метода на отвореното адресиране с двойно хеширане:

Базовата функция:  $h_1(k) = k \bmod 7$

Стъпковата функция:  $h_2(k) = (k \bmod 5) + 1$

Покажете резултата от изпълнението на следните действия:  
Insert(20), Insert(13), Insert(17), Delete(13), Search(17).

---

### Отворено адресиране с двойно хеширане – времева сложност

Теорема (доказателство в учебника, стр. 203-205)

При дадена хеш таблица с отворено адресиране, чийто коефициент на натоварване е  $\alpha = n / m < 1$ , и при (идеалното\*) предположение, че всяка от  $m!$  на брой поредици от проверки има същата вероятност:

Математическото очакване за броя на проверките по време на провалено търсене е ограничено от  $\frac{1}{1-\alpha}$ , а по време на успешно търсене то е ограничено от  $\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$ .

Извод

По условието на теоремата и за константна  $\alpha$  търсенето (провалено или успешно) се изпълнява средно за  $\Theta(1)$ .

\* Както казахме, методите за проверка, които видяхме, не изпълняват тази хипотеза, но двойното хеширане я приближава относително по-добре.

---

### Отворено адресиране – обобщение

Действие		Най-лошият случай	Средният случай
Търсене	Провалено	$\Theta(n)$	$\Theta\left(\frac{1}{1-\alpha}\right)$
	Успешно		$\Theta\left(\frac{1}{\alpha} \ln \frac{1}{1-\alpha}\right)$
Вкаране		$\Theta(n)$	както при проваленото търсене
Извриване		$\Theta(1)^*$	$\Theta(1)^*$

Средните времена за изпълнение при търсенето са при (идеалното) предположение, че всички поредици от проверки са равни по вероятност.

Ако  $\alpha$  е константа, всички действия ще се изпълняват средно за време  $\Theta(1)$ .

\* При положение, че е даден показалец или индекс към мястото за изтриване.

---

### Въпроси за преговор

- Сравнете двата метода за решаването на проблема със сблъсъците (Метода на верижното хеширане и Отвореното адресиране) от следните гледни точки:
  - желаната връзка между  $m$  и  $n$
  - времевата сложност при вкаране на елемент в най-лошия случай
  - начина за изтриване на елементи

2. Търсенето на елемент в хеш таблица, в която проблемът със сблъсъците се решава по метода на отвореното адресиране, се изпълнява в най-лошия случай за време  $\Theta(n)$ . Обяснете защо!  
Ще бъде ли правилно, да напишем  $\Theta(m)$  вместо  $\Theta(n)$ ?  
Гореспоменатото твърдение вярно ли е също така и след множество изтривания от таблицата?

3. Защо при отвореното адресиране с двойно хеширане си струва да изберем размера на таблицата  $m$  да бъде просто число?

Дадена е хеш таблица с размер  $m = 8$ , като проблемът със сблъсъците се решава по метода на отвореното адресиране с двойно хеширане. Базовата функция е  $h_1(k) = k \bmod m$ , а стъпковата функция е  $h_2(k) = k \bmod (m-1) + 1$ . Какво ще се случи, ако се опитаме да вкараме елемент с ключ 1, когато всички нечетни индекси вече са заети? Ще се намери ли свободна клетка за вкарване, ако таблицата не е пълна?

Какво щеше да се промени, ако размерът на таблицата беше  $m = 7$ , вместо 8.

### Отговори на въпросите за преговор

1.	Chain Hashing	Open Addressing
Връзката между $m$ и $n$	$n = O(m)$ (за ефективността на търсенето)	$n \leq m$
Времевата сложност при вкарването на елемент в най-лошия случай	$O(1)$ при положение, че не е нужно да се проверява, дали елементът вече съществува	$\Theta(n)$
Изтриването на елементи	обикновено изваждане от свързан списък	Обозначение на местоположението като „свободно за вкарване и заето за търсене“.

2. В най-лошия случай поредицата от търсения ще се натъкне отново и отново на заети клетки. Щом е така, дължината на поредицата от търсения в най-лошия случай ще бъде равна на броя на складирани елементи плюс още един.

Няма да е правилно, да напишем  $\Theta(m)$ , тъй като дължината на поредицата от търсения не е задължително от порядъка на  $m$  (възможно е  $n = o(m)$ ).

След множество изтривания много клетки в таблицата ще съдържат стойността `deleted`, а времената за изпълнение на действията ще зависят също и от броя на тези клетки, а не само от броя на елементите в таблицата.

3. На ключ 1 съответства следната поредица от проверки: 1, 3, 5, 7, 1, 3, 5, 7. Затова няма да се намери свободна клетка, дори и ако има такава (с четен индекс).

Ако  $m = 7$ , то поредицата от проверки ще бъде: 1, 3, 5, 0, 2, 4, 6. Ако съществува свободна клетка, тя ще бъде намерена.

### Упражнения

### Препоръчителни упражнения от учебника

Глава 11	11.2-3	11.4-1
11.2-2	11.3-1	11.4-2

---

### Допълнителни упражнения

1. Едно от ограниченията за ползването на Отвореното адресиране е нуждата да се знае предварително броя на елементите, които ще влязат в таблицата, за да се знае, какъв размер таблица да се задели.

*Динамичната хеш таблица* предлага следното решение:

Всеки път, когато искаме да вкарваме елемент и таблицата е пълна, ще създадем нова по-голяма таблица (ще променим съответно и хеш функцията) и ще вкарваме в нея всички „стари“ елементи един по един, а също така и новия елемент.

Посочете, какво е средното общо време за изпълнение за вкарването на  $2n$  на брой елемента в таблица с размер  $n$ , която тя е празна отначало:

- а) като размерът на новата таблица нараства с 1 всеки път.
- б) като размерът на новата таблица нараства 2 пъти всеки път.

Забележка: Предположете, че методът за проверка, който използваме, е идеален!

2. Даден е масив  $A$  от  $n$  на брой някакви елемента и число  $z$ .

Опишете алгоритъм, който се изпълнява с линейно математическо очакване за време, и, който връща двойка числа от  $A$ , чиято сума е  $z$ .

Ако няма такава двойка числа, алгоритъмът трябва да връща `nil`.

3. Задачата за уникалността на елемента (Element Uniqueness Problem):

Дадени са числата  $x_1, \dots, x_n$ . Предложете алгоритъм, който проверява, дали съществуват  $i \neq j$ , за които  $x_i = x_j$ .

с 40

---

### Решение 1

При Отвореното адресиране, ако методът за проверка, който използваме, е идеален (тоест, всички поредици от проверки имат еднаква вероятност), то вкарването на елемент в таблицата се изпълнява средно за  $\Theta(1)$ .

И при двете подточки първите  $n$  на брой вкарвания не изискват уголемяване на таблицата, и затова се изпълняват за общо време от  $n \cdot \Theta(1) = \Theta(n)$ .

### Подточка (а)

Всяко едно от следващите вкарвания изискват уголемяване на таблицата и копиране на всички елементи. Първия път ще се извършат  $n + 1$  вкарвания в новата таблица, втория път  $n + 2$  и т. н. (всяко вкарване средно за константно време).

Общо броят на вкарванията е  $\sum_{i=n+1}^{2n} i = \Theta(n^2)$  и затова средното общо време за изпълнение е  $\Theta(n^2)$ .

### Подточка (6)

Ще се извърши само едно уগолемяване в момента на вкарването на  $(n + 1)$ -тия по ред елемент в таблица с размер  $2n$ . Това уголемяване (включително вкарването на елементите) ще се изпълни за средно време от  $\Theta(n)$ . Всеки следващи  $n - 1$  на брой вкарвания ще се изпълнят средно за константно време. Общото средно време за изпълнение в такъв случай е  $\Theta(n)$ .

.....

### Решение 2

Ще вкараме всички числа в хеш таблица и след това ще обходим числата едно след друго, а за всяко число  $k$  ще търсим в таблицата  $z - k$ . Ако намерим желания резултат, връщаме двойката числа, които сме намерили. Иначе на края връщаме nil.

Ще изберем различните параметри на хеш таблицата:

- хеш функцията ще бъде по метода на умножението (методът на делението е подходящ само за цели числа).
- сблъсъците ще се решат, например, по метода на верижното хеширане (възможна е също и употребата на отвореното адресиране).
- при тези условия единственото ограничение върху размера на таблицата е  $n = O(m)$ , затова ще изберем, например,  $m = 0.5n$ .

Сложност: Имаме  $n$  на брой вкарвания и (най-много)  $n$  на брой търсения.

Всяко вкарване ще се изпълни за константно време в най-лошия случай, а търсенето за константно време в средния случай.

Затова в крайна сметка алгоритъмът ще се изпълнява средно за линейно време:

$$\begin{array}{ccc} \Theta(n) & + & \Theta(n) \\ \text{worst-case} & \text{average} & \text{average} \end{array} = \Theta(n)$$

.....

### Решение 3

#### Първо решение

Сортиране и след това допълнително обхождане и проверка, дали имаме два съседни еднакви елемента.

Времева сложност:  $\Theta(n \log n)$  (например, Merge Sort).

#### Второ решение

Ще вкараме числата в хеш таблица по метода на верижното хеширане (можем да използваме и отвореното адресиране).

Хеш функцията и размерът на таблицата  $m$  ще бъдат избрани по един от методите, които учихме.

Ако се получи сблъсък, ще проверим, дали елементът, който вкарваме е равен на един от елементите в списъка.

Времева сложност: Ще обърнем внимание на това, че тук вкарането е малко по-различно от обикновеното вкаране, и, че то на практика е равносилно на търсенето от гледна точка на времевата сложност.

Ще отбележим с  $\alpha_i$  фактора на натоварване в момента на вкарането на  $i$ -тия елемент. Това вкаране се изпълнява за средно време от  $\Theta(1 + \alpha_i) = \Theta(1 + i/m)$ . Общо, ако няма нито една двойка еднакви елемента:

$$\sum_{i=1}^n \Theta(1 + i/m) = \Theta(n) + \Theta(n^2/m) = \Theta(n + \alpha n)$$

Ако изберем  $m$ , така че  $n = O(m)$ , то  $\alpha$  е константа и средната сложност ще бъде линейна по  $n$ .

.....

## Тема 12

### Символни низове Strings

#### В програмата:

- Ще се запознаем с ефективна структура от данни за речник от символни низове – trie.
  - Ще се запознаем с алгоритъм за намиране на най-дългата обща под-редица. Това е алгоритъм, действащ по метода динамично оптимиране (dynamic programming). (библиография за тази тема: Въведение в алгоритмите, глава 16 от първото издание)
- 

#### Структурата от данни – trie

#### Мотивация

Определение: Символен низ (string) върху дадена група  $\Sigma$  е поредица от елементи от  $\Sigma$ .  $\Sigma$  ще наречем азбука, а елементите ѝ – букви.

Върху символните низове може да се определи съотношение на подредбата, което се нарича лексикографски ред (lexicographic order):

Ще казваме, че символният низ  $a = a_0a_1\dots a_p$  е лексикографски по-малък от символния низ  $b = b_1b_2\dots b_q$ , ако е изпълнено едно от двете условия:

1. Съществува цяло число  $j$ , така че  $a_i = b_i$  за  $i = 0, 1, \dots, j-1$  и **също така**  $a_j < b_j$ .  
или:
  2.  $p < q$  и  $a_i = b_i$  за  $i = 0, 1, \dots, p$ .
- 

#### Мотивация

Задачата – ефективно да реализираме речник (включително минимум, следващ елемент и т. н.) от символни низове.

Познати възможности:

- AVL дърво (на всяко кръстовище от което има символен низ или показалец към символен низ)
- Хеш таблица (на всеки елемент от която има символен низ или показалец към символен низ). Може, например, да се използва хеш таблица с верижно хеширане при положение, че разполагаме с добра хеш функция за символни низове.

Анализирайте ефективността на тези реализации!

**Запомнете!** Сравнението на два символни низа изисква линейно време по дължината на по-късия низ от двата.

- Предположете за улеснение, че всички символни низове са с еднаква дължина  $|e|$ !
  - Броят на символните низове ще бъде отбелаязан както обикновено с  $n$ .
- 

## Мотивация

### AVL дърво

Вкарване / търсене: При всяко кръстовище ще се извърши сравнение за време  $\Theta(l)$ , а броят на гореспоменатите сравнения ще бъде  $\Theta(\log n)$ . Затова всички действия ще се извършат за време  $\Theta(l \cdot \log n)$ .

Изтриване / минимум / следващ елемент:  $\Theta(\log n)$ .

### Hash table (хеш таблица)

Вкарване:  $\Theta(l)$  в най-лошия случай, при предположението, че изчислението на хеш стойността за символен низ изисква  $\Theta(l)$  време (иначе  $\Theta(1)$ ).

Изтриване: За константно време в най-лошия случай, ако списъците са дву-свързани (двупосочни) и е даден показалец.

Търсене:  $\Theta(l)$  средно,  $\Theta(l \cdot n)$  в най-лошия случай.

Минимум / следващ елемент:  $\Theta(l \cdot n)$  във всички случаи.

---

## Мотивация

Сега ще видим по-ефективна реализация, при която сложността на действията въобще не зависи от броя на символните низове в структурата, а само от дължината им.

---

## Структурата от данни trie

Първоизточникът на думата *trie* е думата *retrieval* (компютърна операция, която извиква съхранена във файл информация; (възпро)извеждане на данни).

Две предположения:

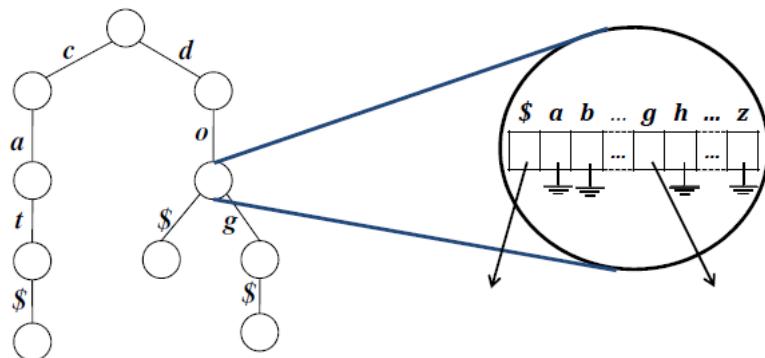
1. През цялата лекция ще предполагаме, че размерът на  $\Sigma$  е константа за целите на анализа на сложността.

2. Съществува символ  $\$$ , който не принадлежи на азбуката  $\Sigma$ . Този символ ще служи като лексикографски най-малък символ и ще представлява край на символния низ.

Trie е дърво, при което всяко кръстовище има най-много  $|\Sigma| + 1$  наследника за всяка една от буквите от азбуката и  $\$$ .

Възможна реализация: Всяко кръстовище е масив с размер  $|\Sigma| + 1$ .

Например:



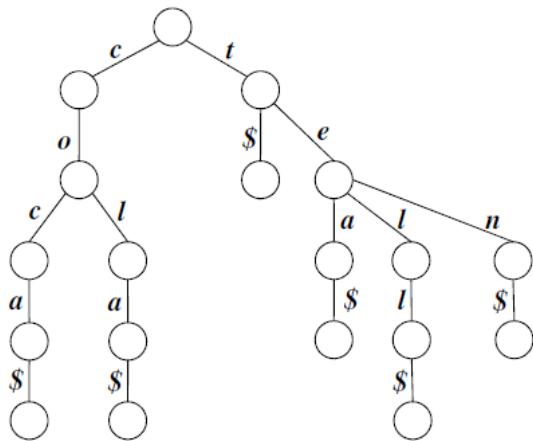
Trie съдържащо 3 символни низа: cat, do, dog.

.....

### Структурата от данни trie

Още един пример:

Кои символни низове се намират в следното trie?



Всеки символен низ е представен от маршрута от корена до листо (броят на листата е равен на броя на символните низове).

Сега ще опишем, как ще се изпълняват различните речникови действия върху trie.

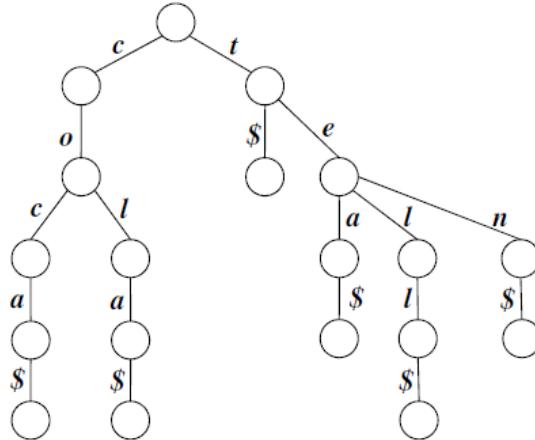
.....

### Действия върху trie

- Search(s) – следваме маршрута на s в trie. s може да се намери, само и единствено ако може да се стигне до листо ( $\$$ ).

- `Insert(s)` – следваме маршрута на  $s$ , докато се натъкнем на `Nil`, и тогава в останалата част от маршрута заделяме съответно нови кръстовища и показалци. На края добавяме `$`.
- `Delete(s)` – трябва да се изтрие частта от маршрута на  $s$ , която е след последната му разделителна точка\*.

\* Как да проверим, дали дадено кръстовище е разделителна точка? С каква сложност?



Гореспоменатите действия се изпълняват за време  $\Theta(|s|)$ .

с 10

---

### Действия върху trie

- `Minimum()` – следваме най-левия маршрут и „записваме буквите, през които сме преминали“.

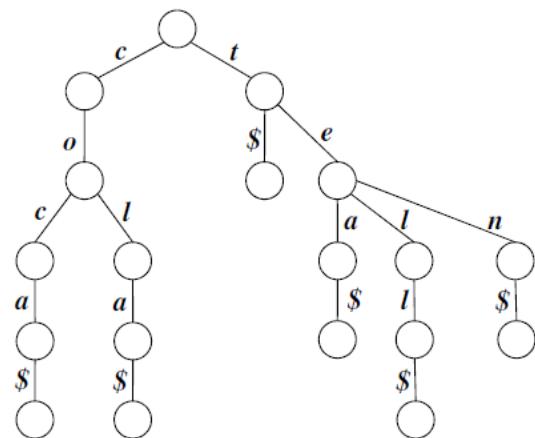
Сложност: При всяко кръстовище трябва да се сканира масива, за да се открие, кой е най-левият показалец, който не е `Nil`.

Това действие изисква време  $\Theta(|\Sigma|) = \Theta(1)$  при всяко кръстовище по маршрута.

Общо  $\Theta(|s_{\min}|)$ , като  $s_{\min}$  е **минималният символен низ**.

- `Successor(s)` – следваме маршрута на  $s$  до листото. От там се връщаме при последната разделителна точка, в която има по-десен показалец, който не е `Nil`, следваме този показалец и от там продължаваме както при `Minimum`.

Сложност:  $\Theta(|s| + |s'|)$ , като  $s'$  е следващият символен низ.



## Сложност по отношение на месторазположението

Дадено е trie, съдържащо  $n$  на брой символни низа, чиято обща дължина е  $m$ .

Въпрос: Каква е нужната сложност по отношение на паметта?

Отговор: Броят на кръстовищата умножен по количеството памет, която изиска всяко кръстовище.

- Всяко кръстовище съдържа масив с размер  $|\Sigma| + 1 = \Theta(1)$ .
- Броят на кръстовищата: имаме един корен, а също така и символен низ  $s_i$ , представен от  $|s_i| + 1$  на брой кръстовища. В най-лошия случай, ако въобще нямаме общи кръстовища, общият брой кръстовища е:

$$1 + \sum_{i=1}^n (|s_i| + 1) = 1 + m + n = \Theta(m)$$

В такъв случай необходимата сложност по отношение на паметта е линейна по общата дължина  $m$  и въобще не зависи от броя на символните низове.

## Сортиране на символни низове

Дадени са  $n$  на брой символни низа, чиято обща дължина е  $m$ .

Търси се ефективен алгоритъм, който сортира символните низове в лексикографски ред.

Възможно (неефективно) решение:

Да вкараме символните низове в AVL дърво (при всяко кръстовище да има символен низ или показалец към символен низ) и проучвателно да обходим дървото in-order.

Сложност:

С цел улеснение ще предположим, че всички символни низове са с еднаква дължина  $m/n$ .

Вкарването на  $n$  на брой символни низа в дървото изиска  $\Theta\left(\frac{m}{n} \sum_{i=1}^n \log i\right) = \Theta(m \log n)$

Обхождането включително отпечатванията изиска  $\Theta(n + m) = \Theta(m)$ .

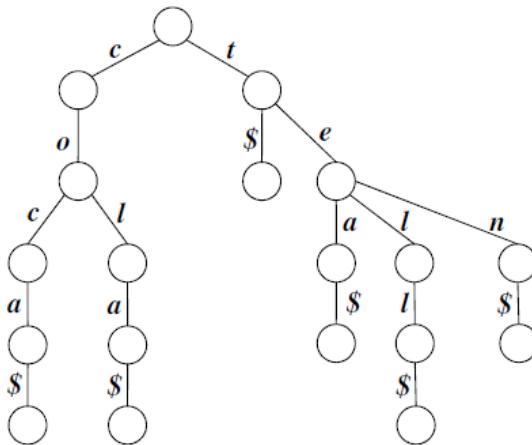
Общо:  $\Theta(m \log n)$ .

Сега ще покажем, как употребата на trie сваля сложността до  $\Theta(m)$ .

## Сортиране на символни низове с помощта на trie

### Упражнение

Дадено е някакво trie, в което има  $n$  на брой символни низа, чиято обща дължина е  $m$ . Опишете алгоритъм, който отпечатва символните низове, които са в trie-то, в лексикографски ред! Каква е сложността на решението ви?



### Сортиране на символни низове с помощта на trie

В продължение на предходното упражнение излагаме алгоритъм за сортиране на  $n$  на брой символни низа, чиято обща дължина е  $m$ .

1. Вкарваме символните низове един след друг в trie-то.
2. Извършваме проучвателно обхождане pre-order на trie-то по следния начин:
  - 2.1. Инициализираме празен свързан списък.
  - 2.2. При всяко слизане по дървото добавяме буквата, „през която сме минали“, в края на списъка
  - 2.3. При всяко качване по дървото изваждаме буквата, която се намира в края на списъка
  - 2.4. Всеки път, когато достигнем листо, отпечатваме съдържанието на списъка от началото до края

### Сложност

$$1. \Theta\left(\sum_{i=1}^n |s_i|\right) = \Theta(m)$$

2. Броят на кръстовищата в дървото, както казахме, в най-лошия случай е  $\Theta(m)$ , а проучвателното обхождане на дърво, както си спомняте, се извършва за линейно време по броя на кръстовищата (редове 2.2. и 2.3. оказват влияние единствено върху константите).

Ред 2.4. незадължително се изпълнява за константно време всеки път, но общото му време за изпълнение е  $\Theta(m)$ . (Зашо?)

Общо  $\Theta(m)$ .

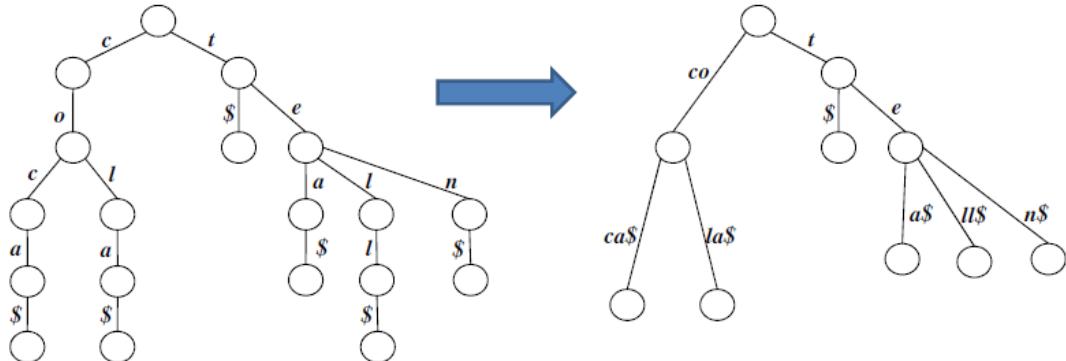
## Подобреие на сложността по отношение на месторазположението – компресия

Дадено е trie, съдържащо  $n$  на брой символни низа, чиято обща дължина е  $m$ .

Както си спомняме, необходимата сложност по отношение на времето е линейна по общата дължина  $m$  и въобще не зависи от броя на символните низове.

Възможно ли е да спестим памет?

Компресия на trie – отмяна на кръстовища с един единствен наследник, за да спестим кръстовища.



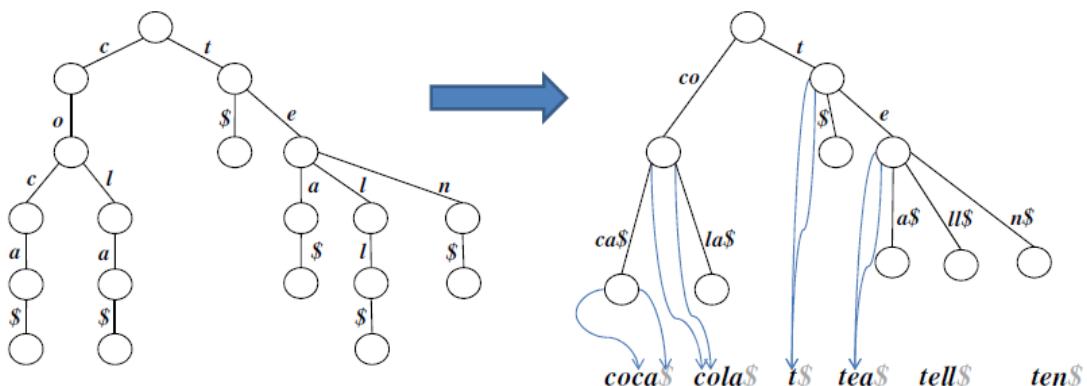
След компресията броят на кръстовищата намалява от  $\Theta(m)$  на  $\Theta(n)$ :

- Имаме  $n$  на брой листа (колкото е и броят на символните низове), най-много  $n/2$  родителя, най-много  $n/4$  прародителя и т. н.
- Затова броят на кръстовищата не е по-голям от  $n + n/2 + n/4 + \dots < 2n$ .

## Подобреие на сложността по отношение на месторазположението – компресия

Получи се проблем – не е достатъчно да държим масив във всяко кръстовище, тъй като кръстовищата представляват символни под-низове, а не букви.

Решение: При всяко кръстовище ще запазим два показалеца – към началото на символния под-низ, който представя кръстовището, и към края му.



Така при всяко кръстовище все пак се изиска  $\Theta(1)$  памет и, както казахме, броят на кръстовищата е спаднал до  $\Theta(n)$ .

Но сега трябва да запазим копия от символните низове, към които ще бъдат насочени показалците от кръстовищата – общо  $\Theta(m)$ .

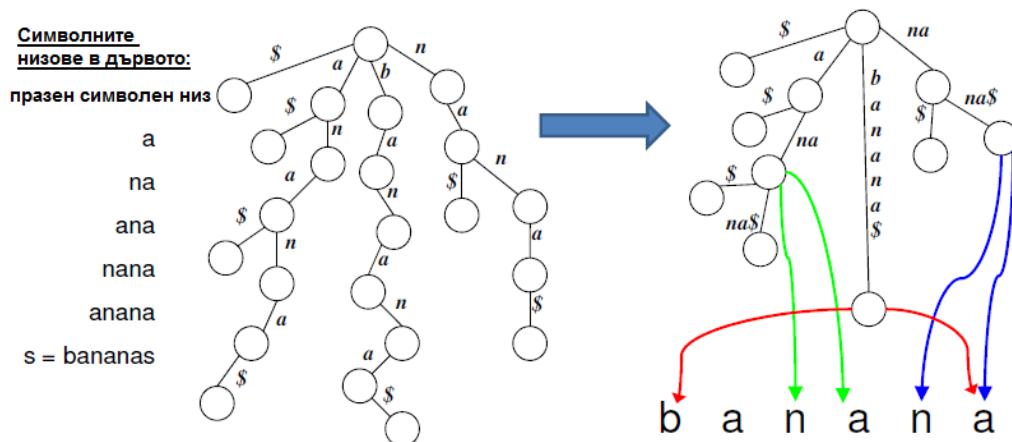
В такъв случай кога компресията на едно trie ще бъде ползотворна при пестенето на памет?

---

### Подобрение на сложността по отношение на месторазположението чрез дърво от окончания (suffix tree)

Гореспоменатото подобрение в сложността по отношение на паметта е по-релевантно в една важна и полезна версия на trie, която се нарича „дърво от окончания“ (suffix tree).

Това е trie, в което са вкарани всички окончания (наставки) от даден символен низ. Например, дърво от окончания за символния низ  $s = \text{"banana"}$ :



$\Theta(|s|)$  памет в компресираното trie, в най-лошия случай

$\Theta(|s|^2)$  памет в некомпресираното trie, в най-лошия случай

---

### Дърво от окончания (suffix tree)

Дърветата от окончания имат няколко важни употреби, измежду които са:

1. Намиране на под-низ от даден символен низ (търсачки, био-информатика).
2. Компресиране на информация (например, алгоритъмът Ziv-Lempel compression).
3. За два дадени символни низа намиране на най-дълъг общ (непрекъснат) под-низ.

За това можете да прочетете, например, в:

Algorithms on strings, Trees and sequences, Dan Gusfield

Chapters 5, 7.3, 7.4, 7.17

c 19

## Най-дълга обща под-редица Longest Common Subseries (LCS)

### Определение на задачата (проблема)

Входни данни: два символни низа: X с дължина m и Y с дължина n.

Изходни данни: дължината на най-дългата обща под-редица.

Забележка: Общата под-редица не е задължително да бъде непрекъсната (за разлика от „символния под-низ“).

Пример:       $X = basacaacb$        $Y = acabbc$   
Най-дългите общи под-редици: acab или acas

Наивно решение: Да се провери всяка възможна под-редица на X:  $\Theta(n2^m)$  (колко под-редици има редица с дължина m?)

### Обозначения

- Най-дългата обща под-редица на символните низове X и Y ще бъде отбелязана с  $LCS(X, Y)$ .
- Дължината на  $LCS(X, Y)$  ще бъде отбелязана с  $L(X, Y)$ .

За символния низ X:

- Ще отбележим с  $x_i$  i-тия символ на X.
- Ще отбележим с  $X[i..j]$  под-редицата от  $x_i$  до  $x_j$ .

## Най-дълга обща под-редица

### Три диагнози, които биха помогнали за утвърждаването на решение

За два символни низа: X с дължина m и Y с дължина n е изпълнено:

Диагноза 1: Ако една от редиците е празна, тогава  $L(X, Y) = 0$  и  $LCS(X, Y) = \emptyset$ .

Диагноза 2: Ако  $x_m = y_n$ , то

$$L(X, Y) = L(X[1..m-1], Y[1..n-1]) + 1$$

$$LCS(X, Y) = LCS(X[1..m-1], Y[1..n-1]) \circ x_m$$

Диагноза 3: Ако  $x_m \neq y_n$ , то

$$L(X, Y) = \max\{ L(X[1..m-1], Y), L(X, Y[1..n-1]) \}$$

$$LCS(X, Y) = LCS(X[1..m-1], Y) \text{ OR } LCS(X, Y[1..n-1])$$

## Просто рекурсивно решение

В светлината на тези диагнози предлагаме рекурсивен алгоритъм за изчисление на  $L(X, Y)$ :

```
L(X, Y)
1.  $m \leftarrow \text{length}[X]$ ,  $n \leftarrow \text{length}[Y]$ 
2. if  $n=0$  or  $m=0$ 
3.     return 0
4. if  $x_m = y_n$ 
5.     return  $L(X[1..m-1], Y[1..n-1]) + 1$ 
6. else
7.     return  $\max \{ L(X[1..m-1], Y) , L(X, Y[1..n-1]) \}$ 
```

### Сложност

Ще отбележим с  $s$  сбора от дълчините  $n + m$ .

В най-лошия случай влизаме на 7-мия ред всеки път:

$$T(s) = 2T(s - 1) + 1 = \Theta(2^s)$$

Експоненциална сложност по сбора от дълчините!

От какво следва това?

---

### Подобрено решение по метода на „динамичното оптимиране“ ("dynamic programming")

При рекурсивните решения на някои задачи, ако има припокриващи се под-задачи (тоест ако имат общи под-под-задачи), много пъти това води до експоненциална времева сложност. Това е така, тъй като алгоритъмът решава много пъти същите под-задачи.

(В този контекст спомнете си също така за рекурсивното решение за изчислението на  $n$ -тия елемент от редицата на Фибоначи!)

Динамичното оптимиране (dynamic programming) е метод, който позволява пестене на време с това, че всяка задача се решава само веднъж: съхраняваме в една таблица изчислените решения и ги изваждаме от таблицата в момента на нужда.

---

### Демонстрация на решението по метода на „динамичното оптимиране“ ("dynamic programming")

В следващата таблица  $C[i, j]$  ще включва  $L(X[1..i], Y[1..j])$ .

Инициализация: Първият ред и първата колона ще съдържат 0.

Приключване: При приключването на алгоритъма изходните данни ще бъдат записани в  $C[m, n]$ .

Тоест  $L(X, Y) = C[m, n]$ .

	$X =$	$a$	$c$	$a$	$b$	$c$
$X$	0	0	0	0	0	0
$a$	0					
$c$	0					
$a$	0					
$c$	0					
$b$	0					

### Стълки на алгоритъма:

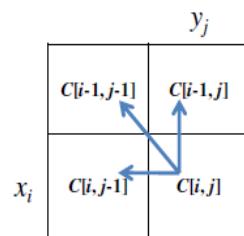
Ще изчислим стойностите в таблицата ред по ред. Можем да изчислим стойността на всяка клетка единствено от стойностите на 3 съседни клетки:

Ако  $x_i = y_j$ , то  $C[i, j] = C[i-1, j-1] + 1$ .

Иначе, ако  $x_i \neq y_j$ , то

ако  $C[i-1, j] \geq C[i, j-1]$ , то  $C[i, j] = C[i-1, j]$ .

Иначе  $C[i, j] = C[i, j-1]$ .



### Демонстрация на решението по метода на „динамичното оптимиране“ ("dynamic programming")

Ако  $x_i = y_j$ , то  $C[i, j] = C[i-1, j-1] + 1$ .

Иначе, ако  $x_i \neq y_j$ , то

ако  $C[i-1, j] \geq C[i, j-1]$ , то  $C[i, j] = C[i-1, j]$ .

Иначе  $C[i, j] = C[i, j-1]$ .

$X$	$Y =$	$a$	$c$	$a$	$b$	$c$
	0	0	0	0	0	0
$b$	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1
$a$	0	↖ 1	← 1	↖ 1	↑ 1	↑ 1
$c$	0	↑ 1	↖ 2	← 2	↖ 2	↖ 2
$a$	0	↖ 1	↑ 2	↖ 3	← 3	← 3
$c$	0	↑ 1	↖ 2	↑ 3	↑ 3	↖ 4
$b$	0	↑ 1	↑ 2	↑ 3	↖ 4	↑ 4

### Алгоритъмът

За удобство ще номерираме редовете и колоните започвайки от 0.

$L(X, Y)$

1.  $m \leftarrow \text{length}[X]$ ,  $n \leftarrow \text{length}[Y]$
2. **for**  $i \leftarrow 0$  to  $m$        $C[i, 0] \leftarrow 0$       //leftmost column
3. **for**  $j \leftarrow 0$  to  $n$        $C[0, j] \leftarrow 0$       //top row
4. **for**  $i \leftarrow 1$  to  $m$
5.     **for**  $j \leftarrow 1$  to  $n$
6.       **if**  $x_i = y_j$
7.            $C[i, j] \leftarrow C[i-1, j-1] + 1$
8.       **else if**  $C[i-1, j] \geq C[i, j-1]$
9.            $C[i, j] \leftarrow C[i-1, j]$
10.      **else**
11.         $C[i, j] \leftarrow C[i, j-1]$
11. **return**  $C[m, n]$

Сложност:  $\Theta(mn)$ .

Въпрос: Какво трябва да се добави, за да можем също да възстановим и самата LCS?

### Възстановяване на най-дългата обща под-редица

В допълнителна таблица B ще запазим „посоките“.

В края на алгоритъма:

- Ще инициализираме празен символен низ Z.
- Ще следваме маршрута започвайки от  $B[m, n]$ :  
Всеки път, когато  $B[i, j] = “↖”$ , ще добавяме  $x_i$  в началото на символния низ Z.

Обърнете внимание, че на края  $Z = \text{LCS}(X, Y)$ , но най-вероятно има още различни от Z най-дълги под-редици.

$\text{LCS}(X, Y) = “acac”$

		$Y = a \ c \ a \ b \ c$					
		$X$	$a$	$c$	$a$	$b$	$c$
$X$	$Y$	0	0	0	0	0	0
$b$	0	↑ 0	↑ 0	↑ 0	↖ 1	↖ 1	
<u>a</u>	0	↖ 1	← 1	↖ 1	↑ 1	↑ 1	
<u>c</u>	0	↑ 1	↖ 2	← 2	← 2	↖ 2	
<u>a</u>	0	↖ 1	↑ 2	↖ 3	↖ 3	↖ 3	
<u>c</u>	0	↑ 1	↖ 2	↑ 3	↑ 3	↖ 4	
$b$	0	↑ 1	↑ 2	↑ 3	↖ 4	↑ 4	

Сложност на възстановяването:  $\Theta(m + n)$ .

### Възстановяване на най-дългата обща под-редица

Изграждането на таблица B:

$L(X, Y)$

1.  $m \leftarrow \text{length}[X], n \leftarrow \text{length}[Y]$
2. **for**  $i \leftarrow 0$  to  $m$        $C[i, 0] \leftarrow 0$       //leftmost column
3. **for**  $j \leftarrow 0$  to  $n$        $C[0, j] \leftarrow 0$       //top row
4. **for**  $i \leftarrow 1$  to  $m$
5.     **for**  $j \leftarrow 1$  to  $n$
6.       **if**  $x_i = y_j$
7.            $C[i, j] \leftarrow C[i-1, j-1] + 1$
8.       **else if**     $C[i-1, j] \geq C[i, j-1]$
9.            $C[i, j] \leftarrow C[i-1, j]$
10.      **else**       $C[i, j] \leftarrow C[i, j-1]$
11. **return**  $C[m, n]$

$B[i, j] \leftarrow “↖”$
$B[i, j] \leftarrow “↑”$
$B[i, j] \leftarrow “←”$

На практика можехме да се откажем от таблица B. Как?

## Въпроси за преговор

1. Намерете най-дългата обща под-редица за символните низове:

X = BABBDAC

Y = BBADCA

чрез използването на алгоритъма LCS!

Намерете също и други най-дълги общи под-редици, различни от тази, която връща алгоритъмът!

с 30

---

## Отговори на въпросите за преговор

1. BBAC, BBDC, BBDA, BADC, BADA

---

## Упражнения

---

### Препоръчителни упражнения от учебника

Глава 12 от второто издание (2008 г.)

Задача 12-2

Глава 16 от първото издание (1998 г.)

16.3-1

16.3-2

16.3-5

---

### Допълнителни упражнения

1. Предложете алгоритъм, чието време за изпълнение е  $O(n^2)$ , и който намира най-дългата възходяща монотонна под-редица от редица с  $n$  на брой числа!

2. Търси се структура от данни за осъществяване на следните действия:

Init({ $s_1, s_2, \dots, s_n$ }) – инициализация на структурата при дадена група от  $n$  на брой символни низа.

Сложност –  $O(m)$ , като  $m$  е общата дължина на всичките символни низове.

Find-Permute( $s$ ) – проверка дали  $s$  е пермутация на един от символните низове  $s_1, \dots, s_n$ .

Сложност –  $O(|s|)$ .

Тоест, при предварително дадена известна група от символни низове, трябва да се отговори на следния въпрос:

Дали който и да е нов символен низ е пермутация на един от дадените символни низове?

3. Дърво от окончания (suffix tree) на даден символен низ  $s$  е trie, в което са вкарани всички окончания на  $s$ . (Окончание на символен низ е негов символен под-низ започващ от дадено място и стигащ до края му.)

При дадено дърво от окончания на символния низ  $s$  покажете, как може с помощта му да се открие, дали символният низ  $s'$  е (непrekъснат) под-низ на  $s$ !

---

### Решение 1

За поредицата  $X$  ще създадем нейно копие  $X'$  и ще го сортираме. Сега ще извикаме метода  $LCS(X, X')$ .

Резултатът е най-дългата обща под-редица между първоначалната редица и сортираното копие, а това също е и най-дългата монотонна под-редица на първоначалната редица.

Време за изпълнение:  $\Theta(n \log n)$  за сортирането плюс  $\Theta(n^2)$  за алгоритъма LCS. Общо  $\Theta(n^2)$ .

### Решение 2

Init:

- Ще сортираме всеки символен низ  $s_i$  посредством Counting Sort.  
Сложността на сортирането на един единствен символен низ  $s_i$  е:  
 $\Theta(n+k) = \Theta(|s_i| + |\Sigma|) = \Theta(|s_i|)$   
Общо –  $\Theta(m)$ .
- Ще вкарваме всички сортирани символни низове в trie-то един след друг.  
Сложността на вкарването на един единствен символен низ е  $\Theta(|s_i|)$ , общо  $\Theta(m)$ .

Find-Permute:

- Ще сортираме  $s$  по същия начин и ще извършим търсене по trie-то на сортирания  $s$ .  
Сложността на сортирането + търсенето е  $\Theta(|s|)$ .

### Решение 3

Символен под-низ на даден символен низ е началото (въведението) на което и да е негово окончание (финална секция).

Затова  $s'$  е символен под-низ на  $s$ , само и единствено ако съществува маршрут на  $s'$ , който започва от корена на дървото от окончания на  $s$ .