

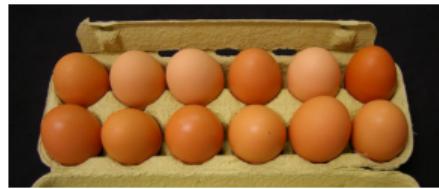
# Редици

Трифон Трифонов

Структури от данни и програмиране, спец. Компютърни науки, 2 поток, 2024/25 г.

3 октомври 2024 г.

Тази презентация е достъпна под лиценза Creative Commons Признание-Некомерсиално-Споделяне на споделеното 4.0 Международен © ⓘ ⓘ ⓘ ⓘ



Базирано на "IMGP0612 - dozen eggs" от RaeAllen,  
CC BY-NC SA-2.0

## АТД: Масив

Последователност от елементи от еднакъв вид, които могат да бъдат избирани по номер (индекс).

# АТД: Масив

Последователност от елементи от еднакъв вид, които могат да бъдат избирани по номер (индекс).

## Операции

- `create(n)` — създаване на масив със зададена големина
- `get(i)` — получаване на елемент с индекс  $i$
- `set(i,x)` — задаване на стойност  $x$  на елемента с индекс  $i$
- `size` — дължина на масива

# АТД: Масив

Последователност от елементи от еднакъв вид, които могат да бъдат избирани по номер (индекс).

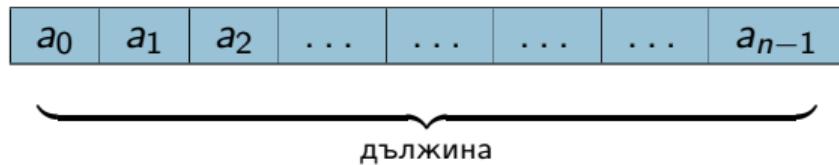
## Операции

- `create(n)` — създаване на масив със зададена големина
- `get(i)` — получаване на елемент с индекс  $i$
- `set(i,x)` — задаване на стойност  $x$  на елемента с индекс  $i$
- `size` — дължина на масива

## Свойства на операциите

- $a.set(i,x).get(i) = x$
- $a.set(i,x).get(j) = a.get(j)$ , ако  $i \neq j$
- $a.create(n).size = n$

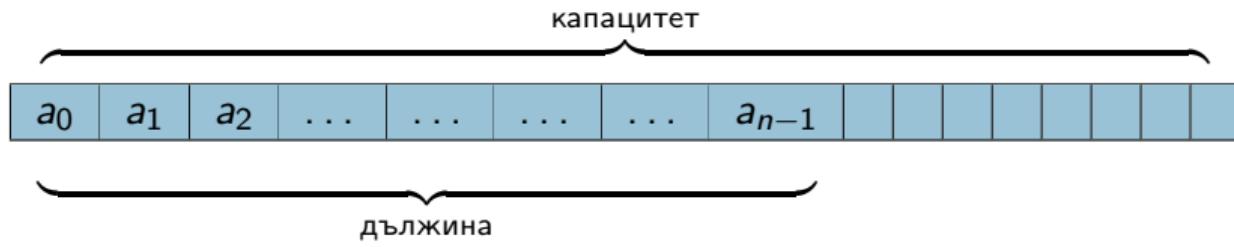
# Статично представяне



**Реализация:** масив във C++.

**Пример:** `int a[10];`

# Динамично представяне



Реализация: `std::vector`.

## std::vector<T>

Реализация на динамичен масив

- vector(n) — създава вектор с дължина n
  - size — дължина на вектора
  - capacity — капацитет на вектора
  - [i], at(i) — достъп до елемент на индекс i
  - front(), back() — достъп до първи и последен елемент
  - push\_back(x) — добавяне на елемента x в края
  - pop\_back() — изтриване на последния елемент
  - insert(...) — вмъкване на елементи на произволна позиция
  - erase(...) — изтриване на елементи на произволна позиция
  - ==, !=, <, >, <=, >= — лексикографско сравнение на два вектора
- int a[10], b[5];  
std::cout << ( a < b );

## std::vector<T>

Реализация на динамичен масив

- `vector(n)` — създава вектор с дължина  $n$
- `size` — дължина на вектора
- `capacity` — капацитет на вектора
- `[i], at(i)` — достъп до елемент на индекс  $i$
- `front(), back()` — достъп до първи и последен елемент
- `push_back(x)` — добавяне на елемента  $x$  в края
- `pop_back()` — изтриване на последния елемент
- `insert(...)` — вмъкване на елементи на произволна позиция
- `erase(...)` — изтриване на елементи на произволна позиция
- `==, !=, <, >, <=, >=` — лексикографско сравнение на два вектора

Специализация `vector<bool>`: реализирана чрез битови масиви

## std::string

Реализация на низ (динамична редица от символи)

- Всички методи на std::vector<char>
  - но не го наследява!
- Методите са съвместими с char\*
- replace(...) — подмяна на символи на произволна позиции
- +, +=, append(...) — конкатенация на низове
- <<, >> — операции за вход и изход
- c\_str() — конвертиране към стандартен C++ низ
- find(...), rfind(...) — търсене на първо/последно срещане
- find\_first\_of(...) — първо срещане на символ от друг низ
- substr(...) — извлечане на подниз
- compare(...) — сравнение с друг низ
- copy(...) — копиране на символи от C++ низ

# АТД: Наредена двойка

Двойка от елементи от потенциално различен тип, в която редът има значение.

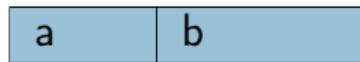
## Операции

- `create(a,b)` — създава двойка от елементите `a` и `b`
- `first` — първият елемент на двойката
- `second` — вторият елемент на двойката

## Свойства на операциите

- `create(a,b).first = a`
- `create(a,b).second = b`
- `create(p.first,p.second) = p`

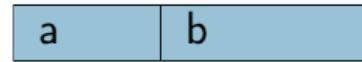
# Физическо представяне



Възможни реализации:

- `struct Pair { int first; char second; };`

# Физическо представяне



Възможни реализации:

- `struct Pair { int first; char second; };`
- `std::pair<T,U>`

# std::pair

## Реализация на наредена двойка

- `pair(x,y)` — създаване на двойка  $(x,y)$
- `first` — първи елемент
- `second` — втори елемент
- `==, !=, <, >, <=, >=` — лексикографско сравнение на две двойки

# АТД: Кортеж

Редица от фиксиран брой елементи от потенциално различен тип, в която редът има значение.

## Операции

- `create(...)` — създаване на кортеж по единични елементи
- `get(i)` — получаване на елемент с индекс/име  $i$
- `set(i,x)` — задаване на стойност  $x$  на елемента с индекс/име  $i$

## Свойства на операциите

- $\text{create}(a_1, \dots, a_i, \dots, a_n).get(i) = a_i$
- $t.set(i,x).get(i) = x$
- $t.set(i,x).get(j) = a.get(j)$ , ако  $i \neq j$

# std::tuple (C++11)

## Реализация на кортеж

- `tuple(...)` — създаване на кортеж с подадените елементи
- `tuple_cat(...)` — слепва произволен брой кортежи
- `get(i)` —  $i$ -ти елемент на кортежа
- `==, !=, <, >, <=, >=` — лексикографско сравнение на два кортежа

## Двоично търсене

 $\Theta(1)$ 

| right - left

Алгоритъм за двоично търсене в сортиран масив:

- 1 търсим елемент Y в сортиран масив в интервала [left; right]
- 2 първоначално left = 0, right = n - 1
- 3 намираме средата на масива mid = (left + right) / 2
- 4 сравняваме търсения елемент Y с ключа X на позиция mid
- 5 ако Y == X — успех
- 6 ако Y < X — търсим Y отляво, right = mid - 1 и към ③
- 7 ако Y > X — търсим X отдясно, left = mid + 1 и към ③

$$k = \lceil \log_2 n \rceil$$

$$\Theta(\log n)$$

$$\frac{n}{2^k} \leq 1$$

$$n \leq 2^k$$

$$k \geq \log_2 n$$

$$\begin{array}{ll} i=1 & n \\ i=2 & \frac{n}{2} \\ i=3 & \frac{n}{4} \\ \vdots & \end{array}$$

$$i=k \quad 1$$

# Двоично търсене

Алгоритъм за двоично търсене в сортиран масив:

- ① търсим елемент Y в сортиран масив в интервала [left; right]
- ② първоначално left = 0, right = n - 1
- ③ намираме средата на масива  $mid = (left + right) / 2$
- ④ сравняваме търсения елемент Y с ключа X на позиция mid
- ⑤ ако  $Y == X$  — успех
- ⑥ ако  $Y < X$  — търсим Y отляво,  $right = mid - 1$  и към ③
- ⑦ ако  $Y > X$  — търсим X отдясно,  $left = mid + 1$  и към ③

Времева сложност:

# Двоично търсене

Алгоритъм за двоично търсене в сортиран масив:

- ① търсим елемент Y в сортиран масив в интервала [left; right]
- ② първоначално left = 0, right = n - 1
- ③ намираме средата на масива mid = (left + right) / 2
- ④ сравняваме търсения елемент Y с ключа X на позиция mid
- ⑤ ако Y == X — успех
- ⑥ ако Y < X — търсим Y отляво, right = mid - 1 и към ③
- ⑦ ако Y > X — търсим X отдясно, left = mid + 1 и към ③

Времева сложност:  $O(\log n)$  в средния и най-лошия случай

# Алгоритъм за бързо сортиране

- 1 Избираме елемент от масива ("ос")

# Алгоритъм за бързо сортиране

- ① Избираме елемент от масива ("ос")
- ② Разделяме масива на две части:

# Алгоритъм за бързо сортиране

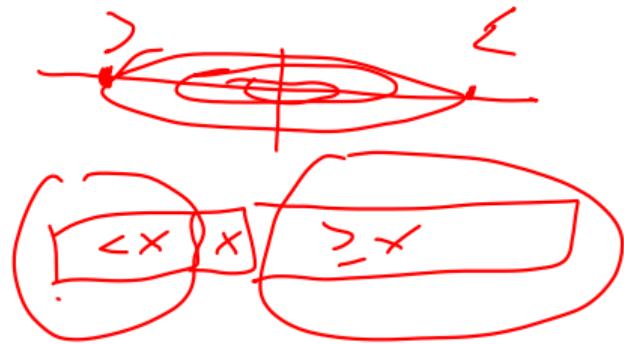
- ① Избираме елемент от масива ("ос")
- ② Разделяме масива на две части:
  - елементи по-малки от оста

# Алгоритъм за бързо сортиране

- ① Избираме елемент от масива ("ос")
- ② Разделяме масива на две части:
  - елементи по-малки от оста
  - елементи по-големи или равни на оста

# Алгоритъм за бързо сортиране

- ① Избираме елемент от масива ("ост")
- ② Разделяме масива на две части:
  - елементи по-малки от оста
  - елементи по-големи или равни на оста
- ③ поставяме оста между двете части на масива



# Алгоритъм за бързо сортиране

- ① Избираме елемент от масива ("ост")
- ② Разделяме масива на две части:
  - елементи по-малки от оста
  - елементи по-големи или равни на оста
- ③ поставяме оста между двете части на масива
- ④ рекурсивно сортираме поотделно двете части на масива

# Алгоритъм за бързо сортиране

- ① Избираме елемент от масива ("ост")
- ② Разделяме масива на две части:
  - елементи по-малки от оста
  - елементи по-големи или равни на оста
- ③ поставяме оста между двете части на масива
- ④ **рекурсивно** сортираме поотделно двете части на масива

Този подход за решение се нарича "разделяй и владей".

# Алгоритъм за бързо сортиране

- ① Избираме елемент от масива ("ост")
- ② Разделяме масива на две части:
  - елементи по-малки от оста
  - елементи по-големи или равни на оста
- ③ поставяме оста между двете части на масива
- ④ **рекурсивно** сортираме поотделно двете части на масива

Този подход за решение се нарича "разделяй и владей".

# Алгоритъм за бързо сортиране

- ① Избираме елемент от масива ("ост")
- ② Разделяме масива на две части:
  - елементи по-малки от оста
  - елементи по-големи или равни на оста
- ③ поставяме оста между двете части на масива
- ④ **рекурсивно** сортираме поотделно двете части на масива

Този подход за решение се нарича "разделяй и владей". Времева сложност:

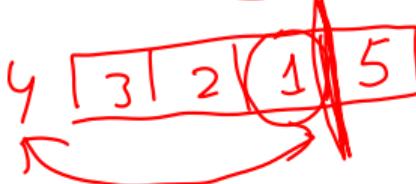
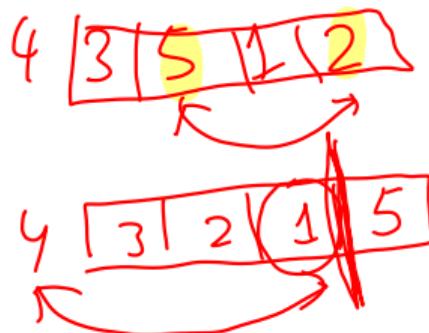
- $O(n \log n)$  в средния случай

# Алгоритъм за бързо сортиране

- ① Избираме елемент от масива ("ост")
- ② Разделяме масива на две части:
  - елементи по-малки от оста
  - елементи по-големи или равни на оста
- ③ поставяме оста между двете части на масива
- ④ **рекурсивно** сортираме поотделно двете части на масива

Този подход за решение се нарича "разделяй и владей". Времева сложност:

- $O(n \log n)$  в средния случай
- $O(n^2)$  в най-лошия случай



# Алгоритми за търсене и сортиране в STL

```
#include <algorithm>
```

- `find(begin, end, value)` – линейно търсене на елемент в контейнер
- `is_sorted(begin, end)` – проверка дали контейнер е сортиран
- `binary_search(begin, end, value)` – двоично търсене на елемент в сортиран контейнер
- `merge(begin1, end1, begin2, end2, output)` – сливане на два сортирани контейнера
- `sort(begin, end)` – сортиране на контейнер на място