Algoritmi in podatkovne strukture 1

Visokošolski strokovni študij Računalništvo in informatika

Podatkovni tipi in strukture

Jurij Mihelič, UniLj, FRI

Predstavitev podatkov

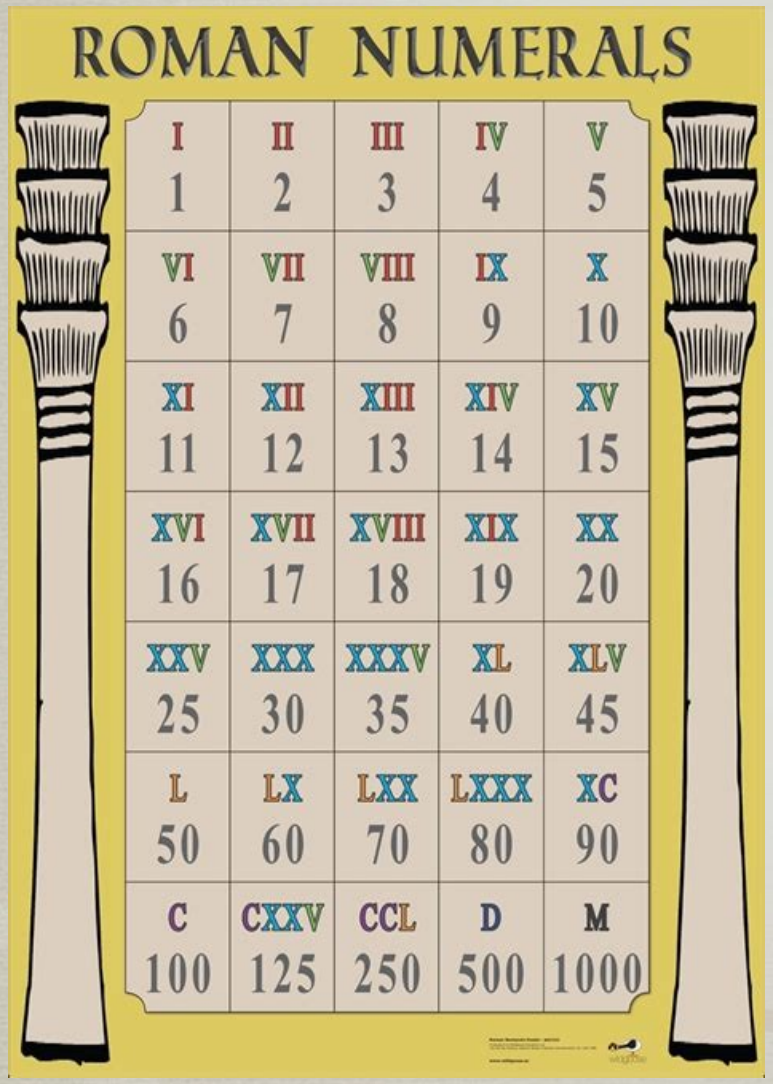
- Seštevanje celih števil

```
  1357
+ 2846
------
  1110
------
  4203
```

Seštevanje dveh celih števil
- števili napišemo eno pod drugo
- začnemo pri enicah in pomikamo proti levi
- seštevamo po dve števki in prenos

Predstavitev podatkov

- Seštevanje celih števil



Evropa v času,
ko je živel al-Khwārizmī,
uporablja rimske številke!

```
  MCDLI
+ DLXVII
---------
  MMXVIII
```

M=1000, D=500, C=100,
L=50, X=10, V=5, I=1

Podatkovni tipi

- Podatkovni tip
  - množica **vrednosti** in način **predstavitve** posameznih vrednosti
    - Java byte: vrednosti od -128 do 127 predstavljene z dvojiškim komplementom
  - množica **operacij** in njihove **implementacije**
    - Java byte: operacije +,-,*,/ in njihove implementacije
- Abstraktni podatkovni tip
  - model za podatke
  - brez predstavitve in brez implementacije

Podatkovni tipi

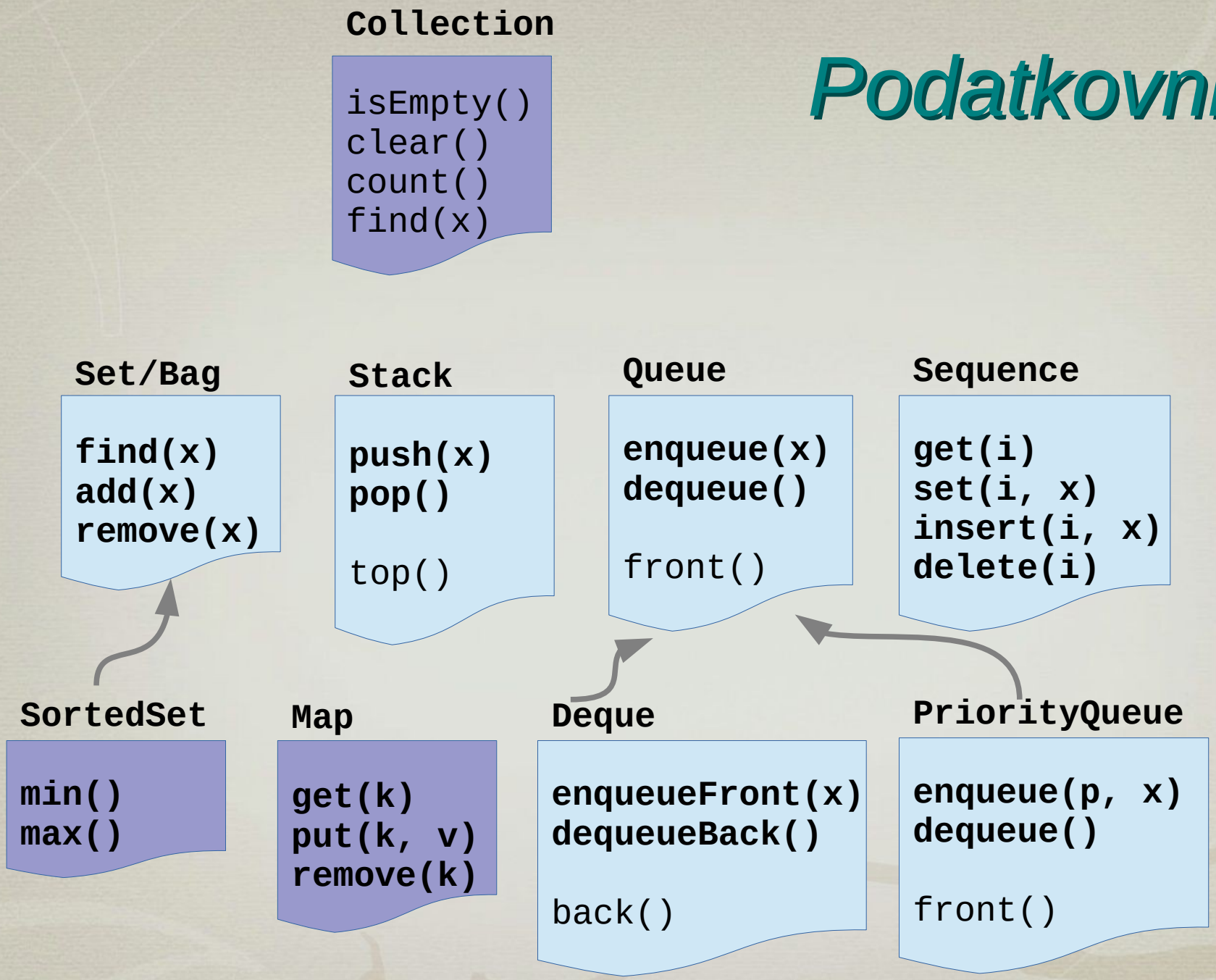
- Objektni opis
  - podatkovna struktura je prejemnik metode

```
Tip s
s.operacija(x)
```

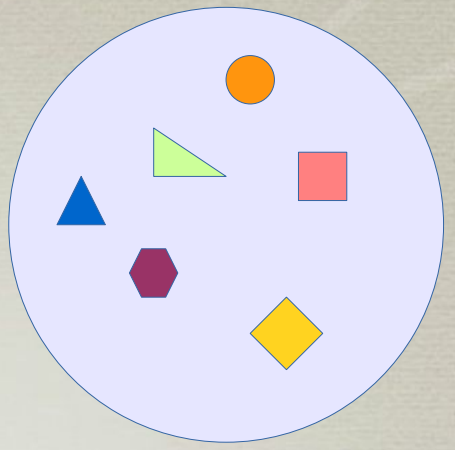
- Klasični opis
  - podatkovna struktura podana kot argument

```
Tip s
operacija(s, x)
```

Podatkovni tipi



Množica

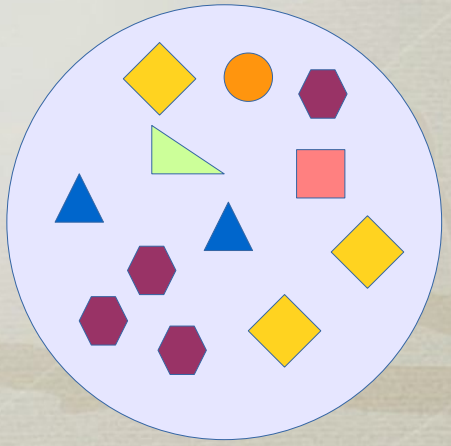


- Množica (*set*)
  - vsebuje enolične elemente
  - brez vrstnega reda

Set/Bag

```
find(x)
add(x)
remove(x)
```

- Vreča (*bag, multiset*)
  - kot množica
  - dovoljuje ponavljanje elementov



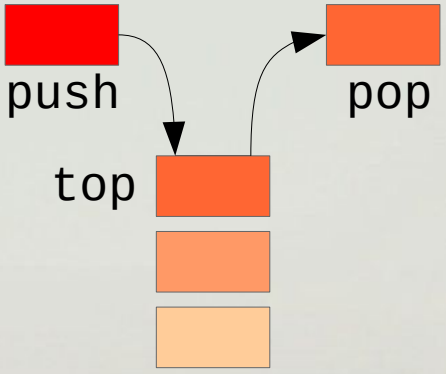
- Urejena množica/vreča
  - vsebuje še operaciji `min()` in `max()`

Sklad

- Sklad (*stack*)
  - LIFO – *last-in, first-out*
  - vrh sklada `top`
  - operaciji `push` in `pop`

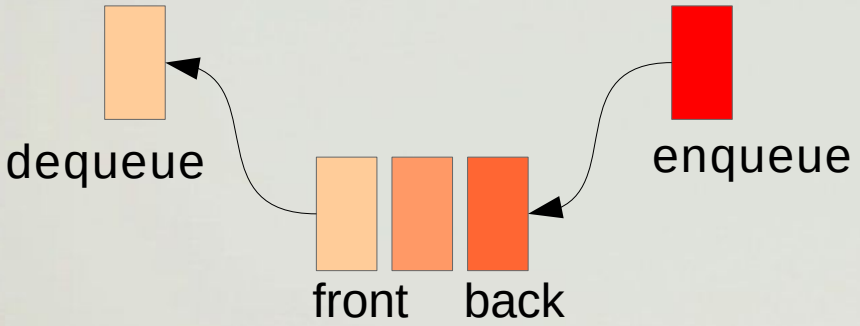
Stack

```
push(x)
pop()

isEmpty()
top()
```



Vrsta

- Vrsta (*queue*)
  - FIFO – *first-in, first-out*
  - sprednji in zadnji konec vrste
  - operaciji `enqueue` in `dequeue`



Queue

```
enqueue(x)
dequeue()

isEmpty()
front()
```

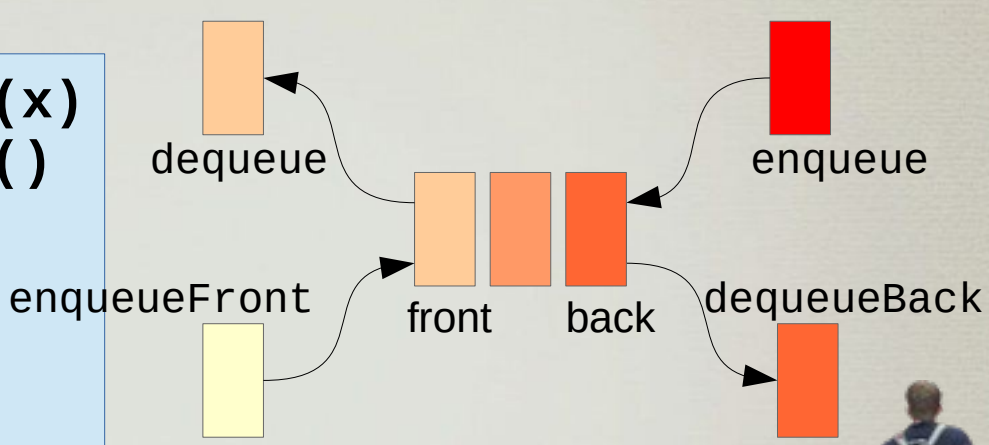
Vrsta z dvema koncema

dvrsta?

- Vrsta z dvema koncema (*deque* ali *dequeue*)
  - dodajanje in odvzemanje spredaj in zadaj

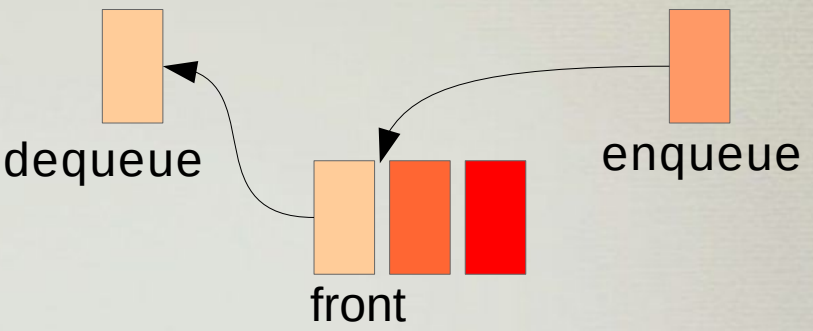
Deque

```
enqueue(x) = enqueueBack(x)
dequeue() = dequeueFront()
enqueueFront(x)
dequeueBack()

isEmpty()
front()
back()
```



Vrsta s prednostjo

- Vrsta s prednostjo (*priority queue*)
  - odvzemanje spredaj
  - dodajanje s prednostjo



PriorityQueue

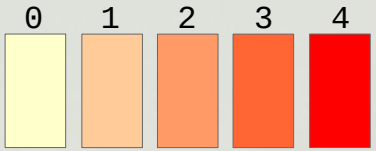
```
enqueue(p, x)
dequeue()

front()
```

Zaporedje

- Zaporedje (*sequence*)
  - naključni dostop: `get(i)`, `set(i, x)`
  - vstavljanje na pozicijo: `insert(i, x)`
  - brisanje na dani poziciji: `delete(i)`

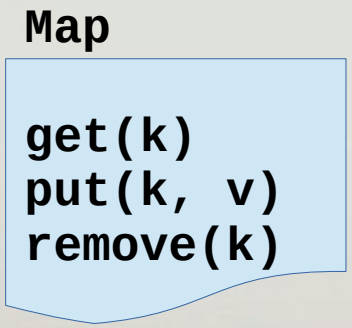
Sequence

```
get(i)
set(i, x)
insert(i, x)
delete(i)
```



Slovar

- Slovar (*dictionary, map*)
  - podoben množici
  - dostop do elementov preko ključa
  - dostop: `get(k)`, `put(k, v)`
  - odstranjevanje: `remove(k)`



Algebrski pogled

- Algebrske strukture
  - matematična definicija obnašanja tipa
  - strukture
    - brez (dvojiške) operacije
      - npr. množica, množica z unarno operacijo
    - z eno operacijo +
      - grupoid (magma), semigrupa, monoid, grupa, ...
    - z dvema operacijama + in *
      - polkolobar, kolobar, obseg, algebra
    - več operacij?

Algebrski pogled

- *Nat* =
  - types: Nat
  - operations: 0: → Nat
    succ: Nat → Nat
    add: Nat, Nat → Nat
  - equations: $n, m \in \text{Nat}$
    $\text{add}(n, 0) = n$
    $\text{add}(n, \text{succ}(m)) = \text{succ}(\text{add}(n, m))$

Algebrski pogled

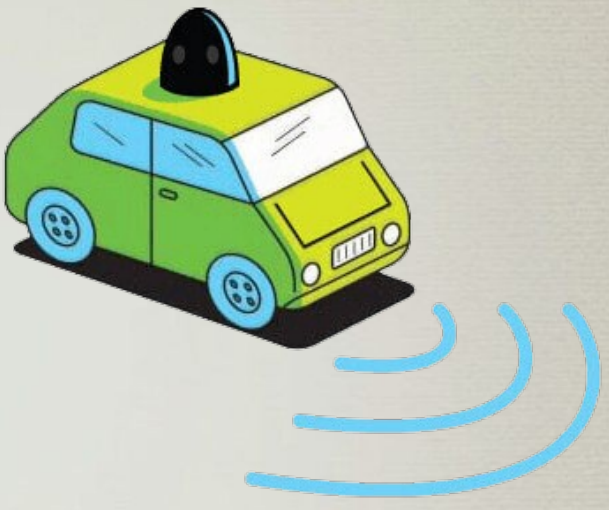
- *Int* =
  - types: Int
  - operations:
    0: → Int
    succ: Int → Int
    pred: Int → Int
    add: Int, Int → Int
  - equations:
    $n, m \in$ Int
    succ(pred(n)) = n
    pred(succ(n)) = n
    add(n, 0) = n
    add(n, succ(m)) = succ(add(n, m))
    add(n, pred(m)) = pred(add(n, m))

Algebrski pogled

- *Stack* =
  - types: Item = Int ∪ Nat
    Stack = Item* ∪ {error}
  - operations: empty: → Stack
    push: Item, Stack → Stack
    pop: Stack → Stack
    top: Stack → Item
  - equations: pop(push(*x*, *s*)) = *s*
    top(push(*x*, *s*)) = *x*
    pop(empty) = empty
    top(empty) = error

Algebrski pogled

- Zakaj se sploh truditi z *matematiko*?
  - ker omogoča izredno jasen in natančen pogled na podatkovne strukture
  - ker omogoča **dokaz pravilnosti** delovanja



- Kdaj resnično rabimo pravilnost?
  - preprečevanje katastrof
  - življenjsko kritične aplikacije
    - jedrske elektrarne, avtonomna vožnja
  - varnostno kritične aplikacije
    - bančne aplikacije