Algoritmi in podatkovne strukture 1

Visokošolski strokovni študij Računalništvo in informatika

Polja
(tabele)

Jurij Mihelič, UniLj, FRI

Polje

- Kaj je polje?
  - podatkovna struktura
  - zbirka: vsebuje elemente
  - zaporedno hrani elemente
  - dostop do elementa preko indeksa

Polje

- Zaporedno hranjenje elementov
  - elementi v pomnilniku zasedajo **zaporedne lokacije**
  - naključni dostop
    - dostop do poljubnega elementa je hitra operacija
  - vstavljanje in brisanje elementov
    - na koncu polja lahko hitro
    - na poljubno lokacijo počasno
  - izkoriščenost **predpomnilnika**
    - če hranimo *vrednosti*, potem dobra
    - če hranimo *reference*, potem slabša (manjša lokalnost)

Polje

- **Kapaciteta** polja
  - fizična velikost polja (java: `a.length`)
  - največje št. elementov v polju
- **Velikost** polja
  - logična velikost polja
  - dejansko št. elementov v polju
- **Izkoriščenost** polja
  - *velikost / kapaciteta*
  - učinkovito hranjenje podatkov v pomnilniku

Polje

- **Statično** polje
  - kapaciteta je fiksna, se ne spreminja
  - pozor: lahko je dinamično alocirano
    - Java: `new int[42]`
    - C: `malloc(42*sizeof(int))`

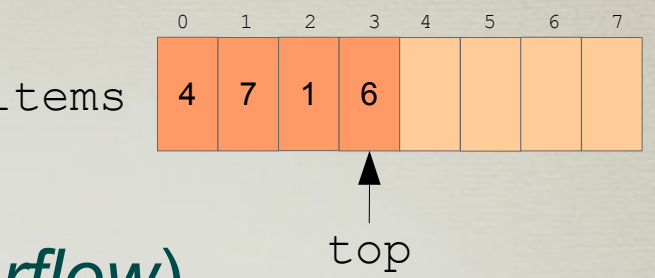
- **Dinamično** polje
  - kapaciteto polja je moč enostavno spreminjati
  - Java: `ArrayList`

Statično polje

- Polje
  - kot **sklad**
  - kot **vrsta** (in **dvrsta**)
  - kot **zaporedje**
  - kot **množica** in **vreča**

Statično polje

- Polje kot **sklad**
  - pozicija `top`
  - podliv / preliv (*underflow* / *overflow*)
  - preprečevanje postopanja (*loitering*)




```
fun push(x) is
    top++
    items[top] = x

fun pop() is
    x = items[top]
    items[top] = null   // postopanje
    top--
    return x

fun top() is
    return items[top]
```

Dodaj še preverjanje za napake

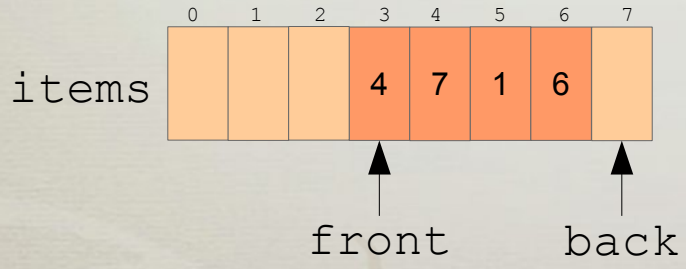
Statično polje

- Polje kot **vrsta**
  - analogija z avtomobilsko vrsto pred avtopralnico

  - analogija z številčenjem

Statično polje

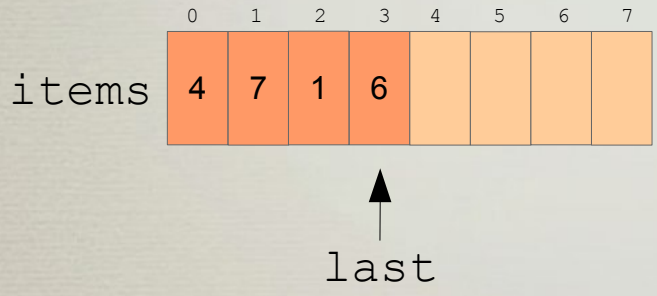
- Polje kot **vrsta** (in **vrsta z dvema koncema**)
  - poziciji `front` in `back`
  - podliv / preliv
  - detekcija prazne in polne vrste
  - preprečevanje postopanja




```
fun enqueue(x) is
    items[back] = x
    back = (back + 1) % length

fun dequeue() is
    x = items[front]
    items[front] = null // postop.
    front = (front + 1) % length
    return x

fun enqueueFront(x) is
    front = (front - 1) mod length
    items[front] = x

fun dequeueBack(x) is
    back = (back - 1) mod length
    x = items[back]
    items[back] = null
    return x
```

Statično polje

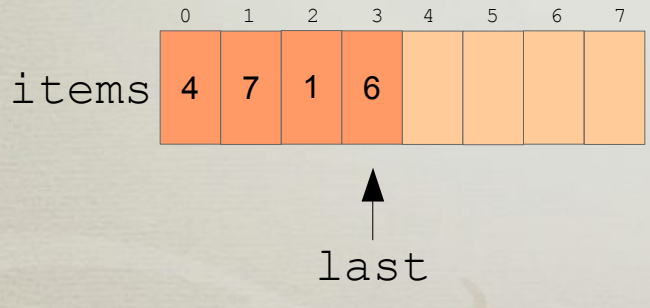
- Polje kot **zaporedje**
  - pozicija `last`
  - podliv / preliv
  - preprečevanje postopanja




```
fun get(i) is
    return items[i]

fun set(i, x) is
    items[i] = x

fun find(x) is
    for i = 0 to last do
        if items[i] == x then return i
    return -1

fun insert(i, x) is
    for j = last downto i do
        items[j + 1] = items[j]
    items[i] = x
    last++

fun delete(i) is
    for j = i to last-1 do
        items[j] = items[j + 1]
    items[last] = null // postopanje
    last--
```

Statično polje

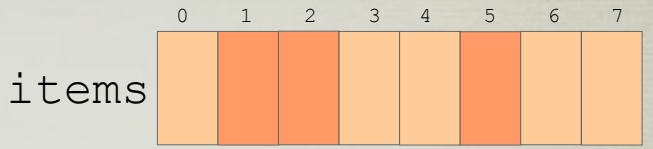
- Polje kot **vreča** in **množica** (1. način)
  - pozicija `last`
  - podliv / preliv
  - preprečevanje postopanja




```
// find(x), delete(i) kot zaporedje

fun add(x) is
    last++
    items[last] = x

fun remove(x) is
    i = find(x)
    if i >= 0 then delete(i)

// množica
fun addUnique(x) is
    if find(x) then return ERR
    last++
    items[last] = x
```

Statično polje

- Polje kot **množica** in **vreča** (2. način)
  - karakteristični (bitni) vektor
    - true/false za vsak element
  - omejitev
    - elementi množice/vreče so števila
    - števila so v omejenem intervalu




```
fun find(x) is
    return data[x]

fun add(x) is
    items[x] = true

fun remove(x) is
    items[x] = false
```

Dinamično polje

- Dinamična zbirka
  - kapaciteto polja (vrsta, dvrsta, množica, zaporedja,...) je moč enostavno spreminjati
    - v ozadju delovanja je statično polje
  - **dodajanje** elementa
    - `push(x)`, `enqueue(x)`, `add(x)`, `insert(i, x)`, ...
    - če *zmanjka* prostora, povečamo kapaciteto
  - **odstranjevanje** elementa
    - `pop()`, `dequeue()`, `remove(x)`, `delete(i)`, ...
    - če **velikost** zbirke postane majhna v primerjavi z njeno **kapaciteto**, zmanjšamo kapaciteto

Dinamično polje

- Sprememba kapacitete
  - kako?
    - ustvarimo novo polje
    - kopiramo elemente iz starega polja v novo
  - kdaj?
    - preverjanje velikosti polja ob klicu `push(x)` in `pop()`
  - koliko?
    - kako veliko je novo polje?
      - različne strategije
      - npr. 2 * trenutna velikost

```
fun resize() is
    c = 2 * size()
    a = newarray[c]
    for i = 0 to last do
        a[i] = items[i]
    items = a
```

Dinamično polje

- Zahtevnost
  - `resize()`
    - kopiranje elementov
    - torej $O(n)$ premikov elementov
  - `push(x)` in `pop()`
    - vsebujeta klic funkcije `resize()`
    - torej je njuna zahtevnost $O(n)$
    - čeprav brez upoštevanja `resize()` le $O(1)$

Amortizirana zahtevnost

- Zahtevnost zaporedja operacij
  - večina operacij stane malo
  - manjšina pa je zelo dragih
  - analiza najslabšega primera upošteva manjšino
- Amortizacija
  - porazdelitev velikih stroškov skozi daljše obdobje
  - **skupno zahtevnost** celotnega zaporedja operacij **porazdelimo na** (amortiziramo) **posamezno operacijo**
  - povprečna zahtevnost brez uporabe verjetnosti

Amortizirana zahtevnost

Primer

Zahtevnost zaporedja operacij `push()` in `pop()` in njuna amortizirana zahtevnost

Povzetek – polje

	operacija	statično polje	dinamično polje
sklad, vrsta, dvrsta	enqueue(x), push(x)	$O(1)$	$O(n)$
	dequeue(), pop()	$O(1)$	$O(n)$
	enqueueFront(x), push(x)	$O(1)$	$O(n)$
	dequeueBack(), pop()	$O(1)$	$O(n)$
zaporedje	get(i)	$O(1)$	$O(1)$
	set(i, x)	$O(1)$	$O(1)$
	find(x)	$O(n)$	$O(n)$
	insert(i, x)	$O(n)$	$O(n)$
	delete(i)	$O(n)$	$O(n)$
vreča, množica	remove(x)	$O(n)$	$O(n)$
	add(x) – vreča	$O(1)$	$O(n)$
	addUnique(x) – množica	$O(n)$	$O(n)$