

# C2142 Návrh algoritmů pro přírodovědce

## 3. Základní datové struktury

Tomáš Raček

Jaro 2017

# Datové struktury

---

**Datová struktura** popisuje uložení dat v paměti počítače.

Výběr datové struktury ovlivňuje:

- množství použité paměti (režie konkrétní struktury)
- složitost operací nad touto strukturou, např.:
  - přidání/odebrání prvku
  - zpřístupnění prvku
  - sekvenční průchod přes všechny prvky
  - nalezení minimálního/maximálního prvku
  - ...

Aktuálně známe:

- jednoduché proměnné (celá čísla, desetinná čísla, znaky,...)
- pole

# Pole I

---

**Pole** je soubor prvků stejného typu uložených v paměti za sebou.

Vlastnosti:

- zabírá souvislou oblast v paměti
- zpřístupnění prvku přes jméno pole a index (př.  $A[i + 1]$ )

## Indexace v poli

- výpočet adresy konkrétního prvku: adresa prvního prvku ( $A$ ) + velikost typu ( $D$ ) \* počet předchozích prvků
- 1D: adresa  $A[i] = A + D \cdot i$
- 2D: adresa  $A[i][j] = A + D \cdot (i \cdot n + j)$
- 3D: adresa  $A[i][j][k] = A + D \cdot (i \cdot n^2 + j \cdot n + k)$

# Pole II

---

## Výhody pole:

- jednoduchá implementace
- přímočaré použití
- zpřístupnění prvku v **konstantním** čase
- sekvenční přístup vede často v praxi k vysokému výkonu (díky využití vyrovnávací paměti)

## Nevýhody pole:

- problematická změna velikosti – Jak přidám/odeberu prvek?

# Statické vs. dynamické datové struktury

---

**Statické datové struktury** (např. pole) mají pevně danou velikost.

- neflexibilní přístup
- lze nadhodnotit velikost → plýtvání paměti ve většině případů

**Dynamické datové struktury** nabízí implicitní způsob změny velikosti.

- složitější na implementaci
- změna velikosti přináší jistou režii

# Dynamické pole I

---

**Nedostatky pole.** Pole neumožňuje libovolně přidávat/odebírat prvky, změna jeho velikosti je složitá → je potřeba znovu alokovat souvislý kus paměti a kopírovat prvky.

## Pozorování

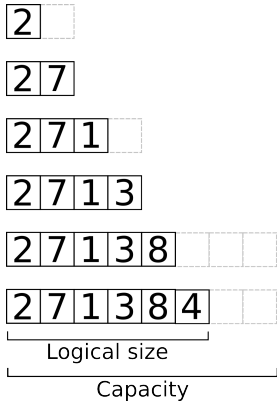
- přidávat prvky na začátek/doprostřed pole je obtížné
- přidání nebo odebrání prvku **na konci** pole je snazší → můžeme se vyhnout kopírování prvků
- pro přidání prvku na konec pole musí být v paměti místo

**Dynamické pole.** Rozšíření pole o možnost přidávat a odebírat prvky. Implementací jde o statické pole, které je logicky rozděleno na aktuálně zabranou a volnou oblast.

## Dynamické pole II

---

- prvky lze na konec přidávat až do vyčerpání kapacity
- poté je potřeba realokace celého pole, typicky např. na **dvojnásobnou** kapacitu
- zjevně má přidání prvku na začátek/doprostřed/na konec pole **lineární** složitost



Jak je tomu ale v praxi?

# Amortizovaná složitost

---

**Pozorování.** Při přidávání prvku na konec dynamického pole je většina těchto operací rychlá – mají **konstantní** složitost. Pokud je ale nutné pole realokovat, složitost je **lineární**.

- lineární složitost nastává ale velmi zřídka
- někdy může být užitečnější jiný aparát pro popis složitosti

**Amortizovaná složitost** neuvažuje operace izolovaně, ale v rámci větší skupiny.

- poskytuje reálnější odhad pro dlouhodobé chování datové struktury
- nedává horní odhad → některé operace mohou trvat „překvapivě“ dlouho



# Dynamické pole – amortizovaná složitost

---

**Příklad.** Uvažme dynamické pole o  $n$  prvcích, kapacitě  $2n$  a operaci přidání prvku na konec pole.

- prvních  $n$  operací přidání prvku na konec má složitost  $O(1)$
- následující operace je ale v  $O(n)$  → způsobí zvětšení pole na dvojnásobek
- dalších  $2n$  těchto operací je opět rychlých
- ...

**Pozorování.** Posloupnost  $n$  operací přidání prvku na konec pole má celkem složitost  $O(n)$ .

**Závěr.** Přidání prvku na konec dynamického pole má **konstantní** amortizovanou složitost. (Analogicky pro odebrání posledního prvku.)

## Dynamické pole IV – praxe

---

**Praxe.** Dynamické pole je implementováno například jako `list` v Pythonu nebo `std::vector` v C++.

**Měření.** Změřte dobu běhu programu, který postupně přidává celkem 100 000 prvků do pole (a) na začátek, (b) doprostřed, (c) na konec.

- (a) 8,7 s
- (b) 4,5 s
- (c) 0,1 s

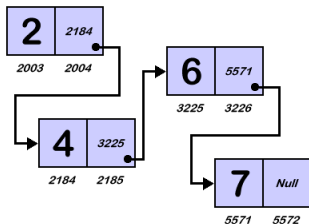
**Závěr.** Měření potvrzuje amortizovaný odhad složitosti. Dynamické pole je efektivní při přidávání/odebírání posledního prvku.

# Spojový seznam I

**Nedostatkem dynamického pole** je přidávání/odebírání prvku mimo jeho konec.

**Spojový seznam** představuje datovou strukturu s **konstantní** složitostí přidání/odebrání prvku.

- každý prvek obsahuje ukazatel na prvek následující – **next**
- na rozdíl od pole nevyžaduje souvislou oblast paměti

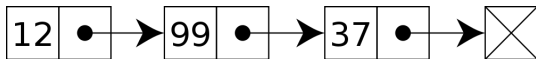


## Spojový seznam II

---

Přístup ke spojovému seznamu je realizován přes ukazatel na jeho začátek – **HEAD**.

Pro zvýšení efektivity přidávání prvků ( $\rightarrow O(1)$ ) na konec seznamu lze využít ukazatel na konec seznamu – **TAIL**.



**Pozorování.** Indexace (= zpřístupnění prvku) je v  $O(n)$ .

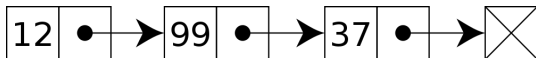
Průchod seznamem:

- sekvenční průchod v  $O(n)$  – stejně jako u pole
- v opačném pořadí ovšem  $O(n^2)$ !

## Spojový seznam III – příklad

---

### Příklad operace nad spojovým seznamem



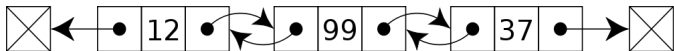
Přidání prvku na začátek seznamu:

1. alokace paměti pro nový prvek **N**
2. kopie vlastních dat do **N**
3. nastavení **N.next** na aktuální **HEAD**
4. nastavení ukazatele **HEAD** na nový prvek

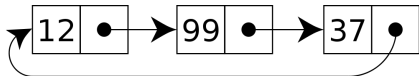
## Další typy spojových seznamů

**Oboustranně zřetěžený spojový seznam.** Je rozšířením spojového seznamu o ukazatel na předchozí prvek – **prev**.

- průchod seznamem v opačném pořadí v  $O(n)$
- zvyšuje paměťovou náročnost



**Cyklický spojový seznam** propojuje poslední prvek s prvním.



# Srovnání datových struktur

---

Složitosti operací nad základními datovými strukturami:

Operace	Pole	Dynamické pole	Spojový seznam
Indexace	$O(1)$	$O(1)$	$O(n)$
Vyhledávání	$O(n)$	$O(n)$	$O(n)$
Přidání prvku	-	$O(n) / O(1)^*$	$O(1)$
Odebrání prvku	-	$O(n) / O(1)^*$	$O(1)$
Paměťová režie	-	$O(n)$	$O(n)$

\* amortizovaná složitost pro poslední prvek

Závěr. Univerzálně nejlepší datová struktura neexistuje.

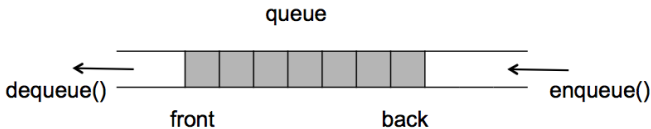
# Fronta I

---

**Fronta** je datová struktura typu **FIFO** (First In First Out).  
Implementuje tzv. **FCFS** (First Come First Served) přístup.

Operace:

- **enqueue** – přidá prvek na konec fronty
- **dequeue** – odebere prvek ze začátku fronty
- obě lze implementovat v  $O(1)$



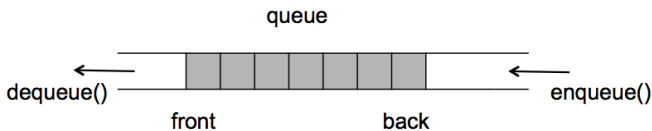


# Fronta II

---

## Implementace fronty

- (dynamické) pole
- spojový seznam



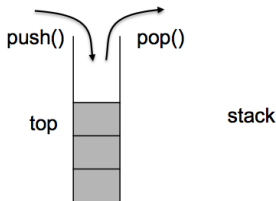
## Prioritní fronta.

- každému prvku je přiřazeno číslo – **priorita**
- priorita určuje, kam bude prvek do fronty zařazen
- několik možných implementací

# Zásobník

---

**Zásobník** je datová struktura typu **LIFO** (Last In First Out). Všechny ostatní vlastnosti jsou shodné s frontou.



Operace:

- **push** – přidá prvek na vrchol zásobníku
- **pop** – odebere prvek z vrcholu zásobníku
- obě lze implementovat v  $O(1)$

# Abstraktní datový typ

---

**Abstraktní datový typ (ADT)** je matematický model pro určité datové struktury definované svými operacemi a omezeními na nich.

- teoretický koncept zjednodušující analýzu chování
- někdy je součástí specifikace i složitost operací
- konkrétní implementaci lze zvolit

Příklady: seznam, fronta, zásobník, množina,...

## Ukázka pro zásobník

- zásobník  $S$ , prvek  $k$ , proměnná  $X$
- $S.\text{push}(k)$ ;  $X \leftarrow S.\text{pop}()$  je ekvivalentní  $X \leftarrow k$
- operace  $\text{push}()$  a  $\text{pop}()$  mají **konstantní** složitost