

Algoritmy – IAL



Ing. Ivana Burgetová, Ph.D.

Ing. Radek Hranický, Ph.D.

FIT VUT v Brně

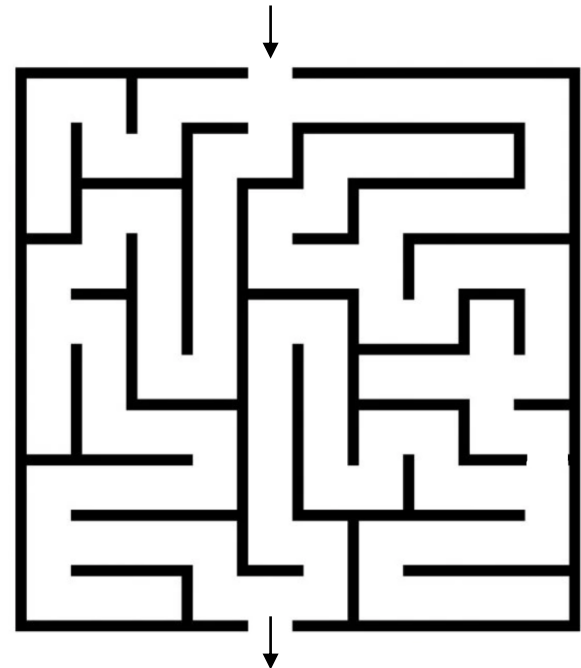
2025/26

Algoritmy

- Proč algoritmy?
- Co je na nich asi tak zajímavého?



1 problém – více různých řešení:
Jak vybrat to „**správné**“?



Obsah

- Organizační informace
- Algoritmy a datové struktury – úvod
- Složitost algoritmů – základní pojmy
- Klíčové pojmy – abstrakce, datové struktury, funkce, rekurze

Personální zajištění předmětu

□ Garant:

- [Ing. Ivana Burgetová, Ph.D.](#)
 - Organizace celého předmětu včetně zkoušek

□ Přednášející:

- [Ing. Ivana Burgetová, Ph.D.](#) – úterý 12:00 (E112 a okolí)
- [Ing. Radek Hranický, Ph.D.](#) – středa 18:00 (D105)

□ Asistenti - konzultace k domácím úlohám a k projektu:

- [Ing. Daniel Dolejška](#) – DÚ1 a náhradní projekt
- [Ing. Jan Zavřel](#) – DÚ2

Informace k předmětu

□ IS VUT

- Všechny termíny – půlsestrální a semestrální zkoušky
- Všechna zadání – domácí úkoly a náhradní projekty

□ Moodle VUT

- Studijní materiály – prezentace k přednáškám
- Soubory potřebné pro řešení domácích úloh
- Doplnující materiály ke studiu i k projektům
- Diskuzní fóra

Přednášky

- Úterý 12:00 – 14:50
- Středa 18:00 – 20:50
- **Přednášky jsou základním zdrojem informací – chodte na ně!**
- Záznamy: budou zveřejňovány
 - Prosíme, nechodte na přednášku, pokud nejste zcela FIT.
- Výjimky (**přednášky odpadnou**):
 - Státní svátek úterý **28. 10. 2025**, přednáška odpadne i ve středu **29. 10. 2025**

Bodové hodnocení

□ Rozdělení bodů:

- 2 domácí úlohy (po 10 bodech)
- projekt s obhajobou pro tým 3-4 studentů (15 bodů)
obhajující bude vybrán losem
- půlsestrální zkouška (14 bodů)
- semestrální zkouška (51 bodů, **minimum 24 bodů**)

- Pro udělení **zápočtu** je nutné získat **minimálně 20 bodů za semestr** (tj. 40,8 %)

Doplňující informace

□ Domácí úkoly:

- 6 bodů za základní testy + 4 body za pokročilé testy
- Správná funkčnost na serveru eva.fit.vutbr.cz

□ Projekt:

- Týmový (4 studenti), společný s předmětem IFJ
- Body za funkčnost, obhajobu a dokumentaci (5+5+5 bodů)
- Náhradní projekty – téma: grafové algoritmy (7+5+3 bodů)

□ Půlsemestrální zkouška:

- Přihlašuje se student v IS VUT. **Bez přihlášení nebude přiděleno místo ani připraveno zadání – nelze se zúčastnit!**

Termíny

- *Žádost o uznání DÚ:* 16. 9. – 3. 10. 2025
- Přihlašování na projekty: 24. 9. – 5. 10. 2025
- 1. domácí úloha: 22. 9. – 19. 10. 2025
- 2. domácí úloha: 20. 10. – 23. 11. 2025
- Přihlašování na půlsemestrální zkoušku:
1. 10. – 15. 10. 2025

- Půlsemestrální zkouška: 21. 10. a 22. 10. 2025
(v době přednášky, pokud to půjde)
- Odevzdání projektu: 3. 12. 2025
- Obhajoby projektů: 4. 12. – 12. 12. 2025

Studijní zdroje

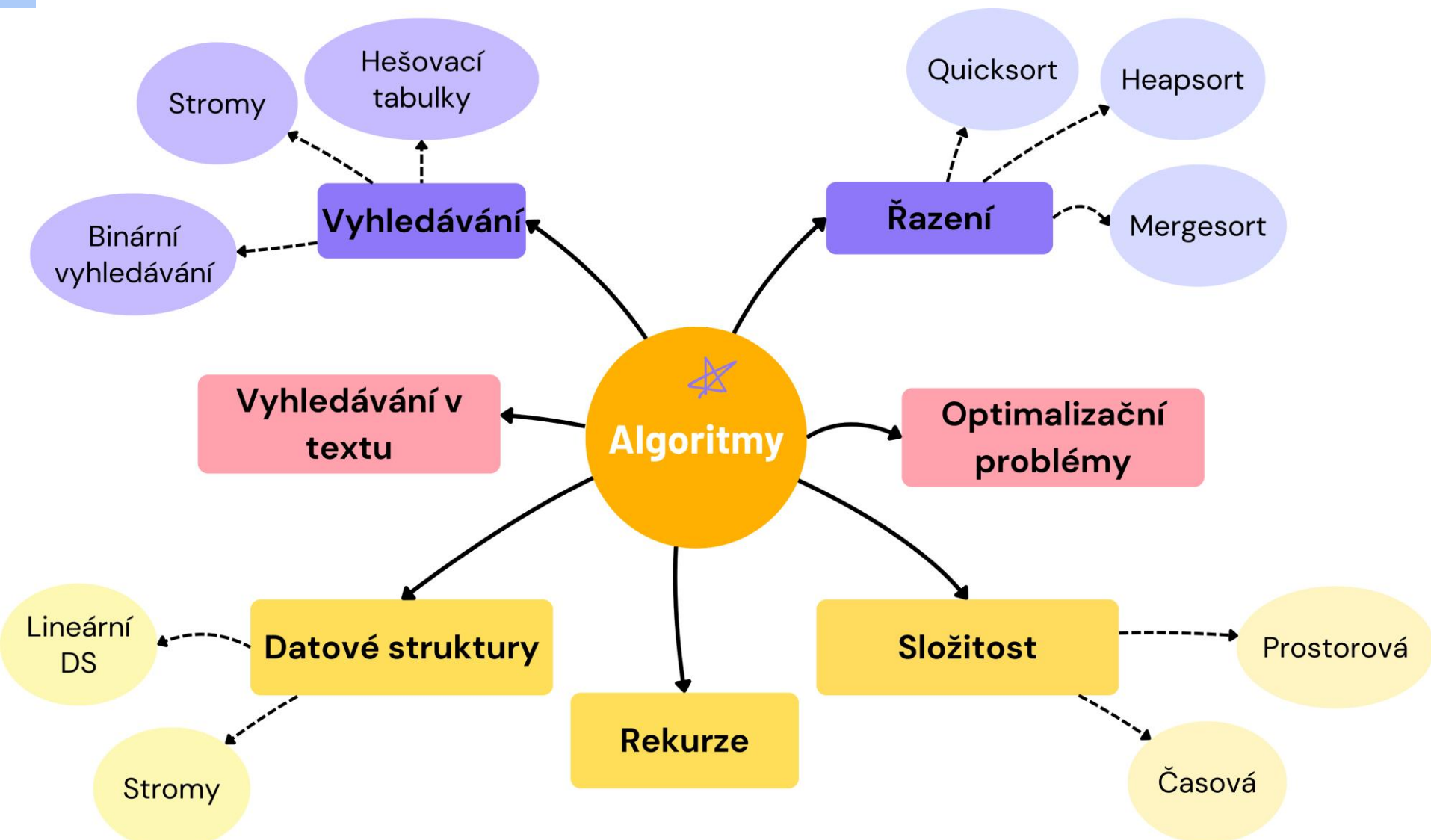
- ❑ **Prezentace k přednáškám** – budou postupně dostupné v Moodle VUT
- ❑ Mareš, M., Valla, T.: **Průvodce labyrintem algoritmů**, CZ.NIC, 2017
- ❑ Cormen, T.H., Leiserson, Ch.E., Rivest, R.L.: Introduction to Algorithms, Cambridge MIT Press, 2009
- ❑ Sedgewick, R.: Algoritmy v C, Addison Wesley 1998. Softpress 2003.

- ❑ Honzík, J.M.: Studijní opora IAL – již není aktualizovaná a neobsahuje témata nově zařazená do přednášek (např. grafové algoritmy, dynamické programování). Dostupná v Moodle VUT

Odhad časové náročnosti předmětu

- 1 kredit = 25-30 hodin práce
- 5 kreditům odpovídá 125-150 hod. studijní práce průměrného studenta, z toho:
 - Přednášky 39 hod
 - 2 domácí úlohy 26 hod
 - práce na projektu 35 hod
 - průběžné studium 20 hod
 - příprava na půlsem. a záv. zkoušku 30 hod

Náplň předmětu



Návaznosti předmětu

- **Schopnost algoritmizace** je klíčová při návrhu a implementaci SW projektů v dalších předmětech i v programátorské praxi.
Když se občas potkáme s našimi absolventy, často právě předmět Algoritmy zpětně hodnotí jako **nejpřínosnější** z celého studia.
- V dnešní době již máte řadu algoritmů a datových struktur dostupných v knihovnách. Pro **výběr nejvhodnějšího** algoritmu či datové struktury pro daný účel je však nutné umět jednotlivé alternativy zhodnotit. A pro **správné použití** datové struktury či algoritmu se porozumění principům určitě hodí.
- Probíraná témata se objevují u **státních závěrečných zkoušek**:
 - **27. Datové a řídicí struktury imperativních programovacích jazyků**
 - **28. Vyhledávání a řazení**
 - **30. Hodnocení složitosti algoritmů (paměťová a časová složitost, asymptotická časová složitost, určování časové složitosti)**
 - <https://www.fit.vut.cz/fit/info/rd/2020/rd39-201217.pdf>

Témata přednášek

1. Organizační informace. Úvod do **algoritmů**. **Asymptotická časová složitost**.
2. **Abstraktní datový typ** a jeho specifikace. Specifikace, implementace a použití **ADT seznam**.
3. Specifikace, implementace a použití ADT **zásobník**, **fronta**, vyhledávací tabulka. Implementace specifických typů polí.
4. **Binární strom**, algoritmy nad binárním stromem.
5. **Vyhledávání**, sekvenční, v poli, binární vyhledávání, binární vyhledávací stromy.
6. AVL strom, vyhledávání v tabulkách s rozptýlenými položkami, stromy s více klíči ve vrcholech.
7. **Řazení**, principy, bez přesunu, s vícenásobným klíčem. Metody řazení polí I.
8. Metody řazení polí II.
9. **Vyhledávání v textu**.
10. Techniky řešení problémů, rekurze, **dynamické programování**.
11. **Grafové algoritmy**, tvorba dokázaných programů.

Učební cíle a kompetence – specifické

- ❑ Seznámit se základními principy složitosti algoritmů
- ❑ Seznámit se se základními abstraktními datovými typy a strukturami, naučit se je implementovat a používat
- ❑ Seznámit se s vyhledávacími metodami, naučit se je implementovat a používat
- ❑ Seznámit se s řadicími metodami, naučit se je implementovat a používat
- ❑ Naučit se rekurzivní a nerekurzivní zápisy základních algoritmů
- ❑ Naučit se porozumět principům a analyzovat algoritmy vyhledávání a řazení.
- ❑ Seznámit se s metodami pro vyhledávání v textu
- ❑ Seznámit se se základy dynamického programování
- ❑ Seznámit se se základními grafovými algoritmy

Učební cíle a kompetence – generické

- ❑ Naučit se základům týmové práce při tvorbě malého projektu
- ❑ Naučit se základům tvorby dokumentace projektu
- ❑ Naučit se základům prezentace projektu a obhajobě dosažených výsledků
- ❑ Seznámit se se základy anglické i české terminologie v oblasti algoritmizace

Obsah

- Organizační informace
- Algoritmy a datové struktury – úvod
- Složitost algoritmů – základní pojmy
- Opakování – abstrakce, datové struktury, funkce, rekurze

Algoritmy a datové struktury

- Počítače používáme pro řešení nejrůznějších problémů:
 - Problém obchodního cestujícího
 - Problém nalezení nejkratší cesty
 - Hledání nejdelšího společného podřetězce
 - Vyhledávání a řazení
- Počítač potřebuje návod, jak daný problém vyřešit:
algoritmus

Algoritmy a datové struktury

□ Algoritmus:

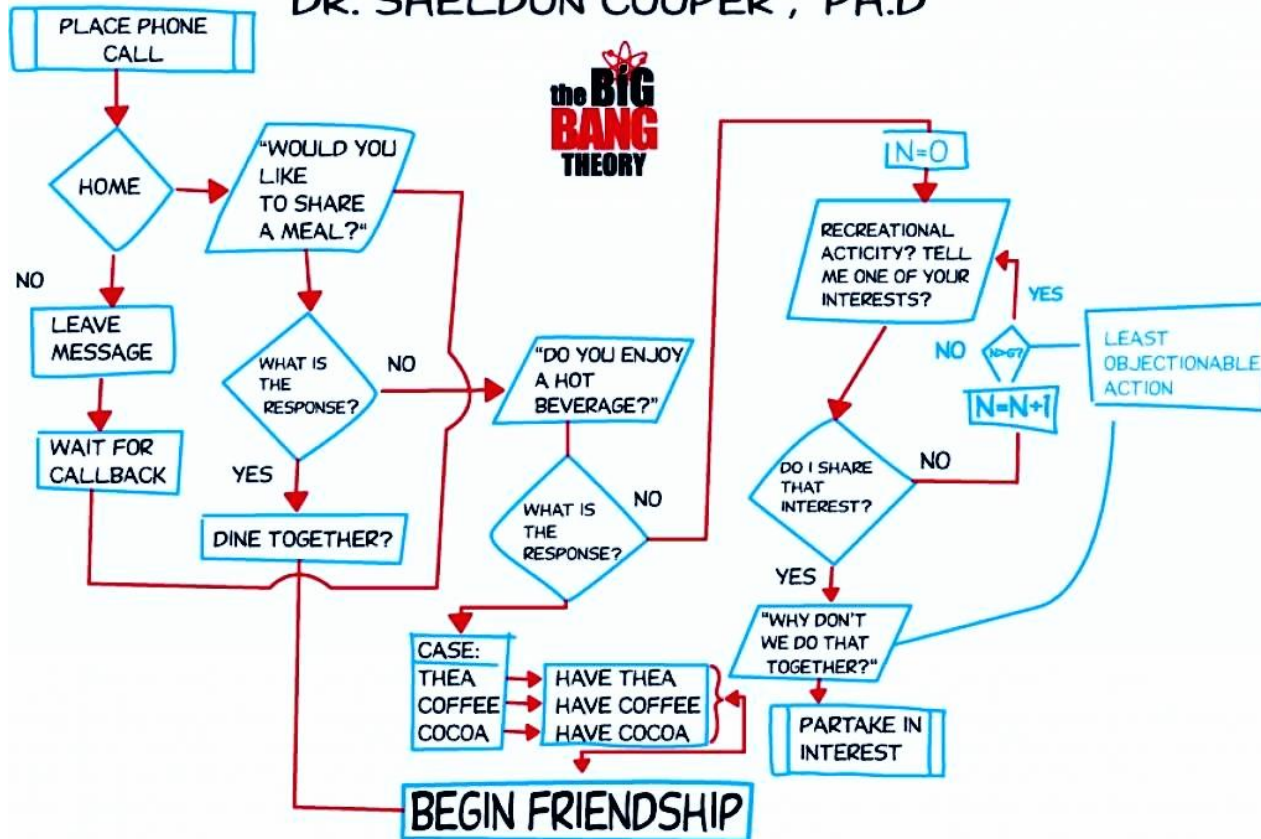
- **konečná**, uspořádaná množina úplně definovaných **pravidel** pro vyřešení nějakého **problému**
- **posloupnost** výpočetních kroků, které transformují **vstup** na **výstup**
- nástroj pro řešení dobře specifikovaného výpočetního **problému** (specifikovaného pomocí vztahů mezi vstupy a výstupy)
- **správnost** algoritmu – pro libovolný vstup skončí s korektním výstupem
- **vlastnosti**: konečnost, obecnost, determinovanost, resultativnost, elementárnost

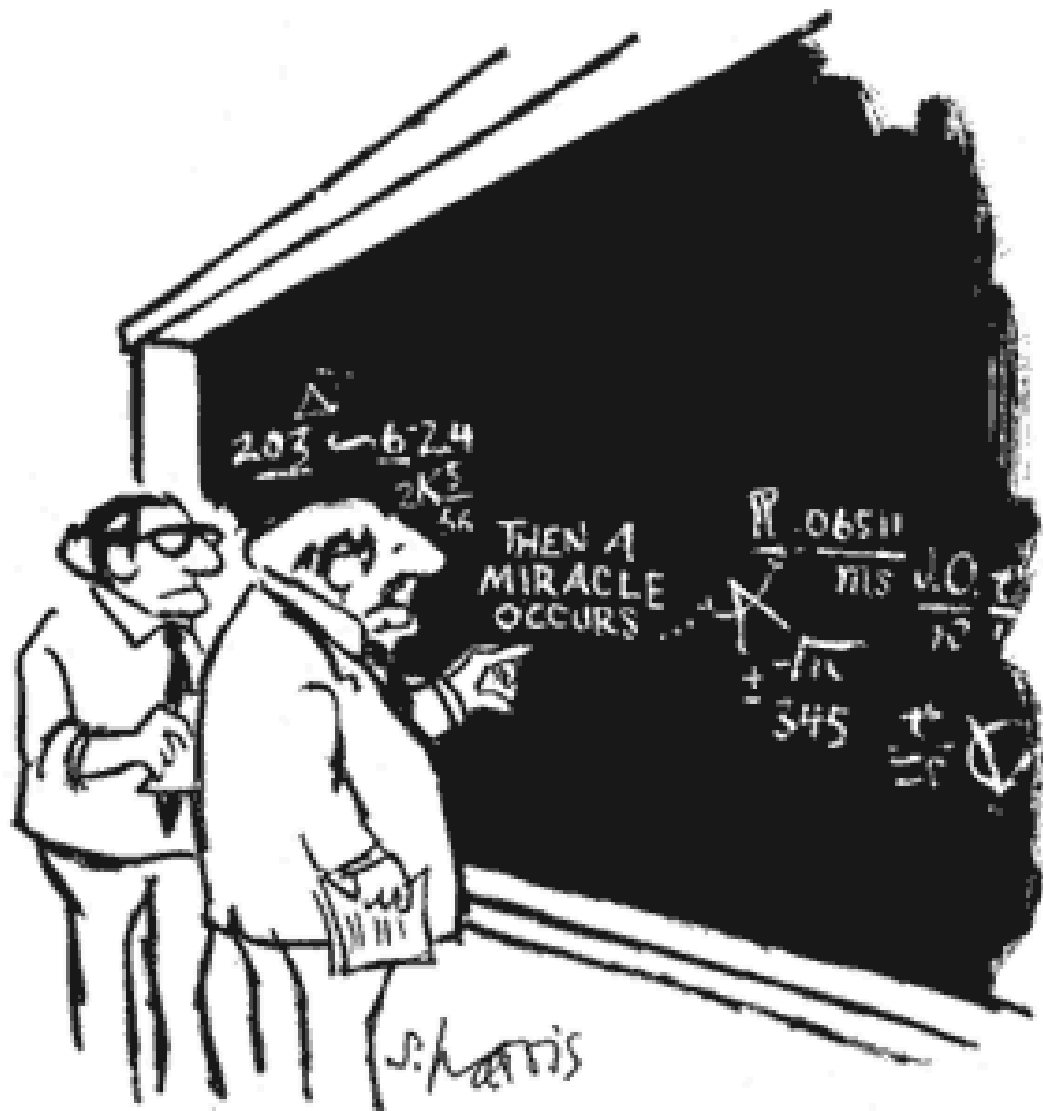
□ Popis algoritmu vs. implementace algoritmu

Algoritmy a datové struktury

THE FRIENDSHIP ALGORITHM

DR. SHELDON COOPER, PH.D





"I THINK YOU SHOULD BE MORE EXPLICIT HERE IN STEP TWO."

Algoritmy a datové struktury

- ❑ Programovací techniky:
 - Rozděl a panuj (divide and conquer)
 - Dynamické programování
 - Hledání s návratem (backtracking)
- ❑ Složitost algoritmů
- ❑ Možnosti paralelizace
- ❑ Datové struktury:
 - Způsob uložení a organizace dat za účelem umožnění přístupu k datům a jejich modifikaci
 - Usnadňují řešení problémů

Proč datové struktury?

- Datové struktury **statické x dynamické**
(pole x seznam, binární vyhledávací strom)
- Různé datové struktury mají své **výhody i nevýhody**:
 - Doba přístupu k jednotlivým položkám se liší
 - Zachování uspořádanosti prvků při vkládání nových dat je různě náročné
 - Rušení prvků nemusí být vždy jednoduché
- Pro **různé aplikace** jsou vhodné **různé datové struktury**

Proč algoritmy?

□ Úloha: dvojice prvků se zadaným součtem

- **Vstup:** uspořádaná posloupnost prvků a číslo s
- **Výstup:** dvojice prvků (ne nutně různých), jejichž součet je s
- **Příklad:**

□ Posloupnost prvků:

1	3	6	6	8	9	10
---	---	---	---	---	---	----

□ Číslo s : 12

□ Výstup: dvojice 3 a 9, 6 a 6

□ Řešení č. 1 – hrubou silou

- Sečteme všechny dvojice $x_i + x_j$
- Počet dvojic, které musíme sečíst: n^2

Proč algoritmy?

□ Řešení č. 2 – využití binárního vyhledávání:

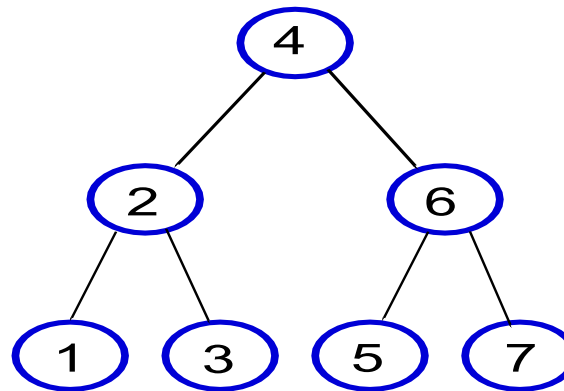
- **Myšlenka:** pokud zvolíme nějaké x_i , víme, že $x_j = s - x_i$ a můžeme x_j zkusit vyhledat
- Postupně zkoušíme všechna x_i a vyhledáváme k nim x_j
- Vyhledávat můžeme metodou binárního vyhledávání (půlení intervalu), nalezne prvek nejpozději po $\log_2 n$ krocích
- Celkově tedy provedeme **$n \cdot \log_2 n$** kroků

1	3	6	6	8	9	10
---	---	---	---	---	---	----

Proč algoritmy?

- Pozn: Kolikrát lze rozpůlit interval?
 - Kolikrát mohu dané číslo n dělit dvěma?
 - $y \approx \log_2 n$

1	2	3	4	5	6	7
---	---	---	---	---	---	---



Proč algoritmy?

□ Řešení č. 3 – metoda dvou jezdců

- Použijeme dva indexy – levý (i - začíná na 1. pozici a pohybuje se doprava) a pravý (j – pohybuje se doleva).
- Pravým indexem vyhledáváme dvojici k aktuálnímu prvku x_i .
- Vyhledávat budeme sekvenčně od konce pole, pokud je součet větší, pohybujeme se doleva, pokud je součet menší, posuneme se k dalšímu prvku (x_{i+1}).
- Pokud se posuneme k prvku x_{i+1} , pak prvek x_j se může nacházet na pozici, kde jsme skončili předchozí vyhledávání nebo vlevo od ní.
- Provedeme maximálně n kroků

1	3	6	6	8	9	10
---	---	---	---	---	---	----

Proč algoritmy?

- ❑ Jeden problém – mnoho různých řešení
- ❑ Každé řešení (algoritmus) má své výhody a nevýhody
- ❑ Pokud dokážeme zhodnotit **vlastnosti** jednotlivých řešení, můžeme vybrat to, které je v našem případě **nejvhodnější!**



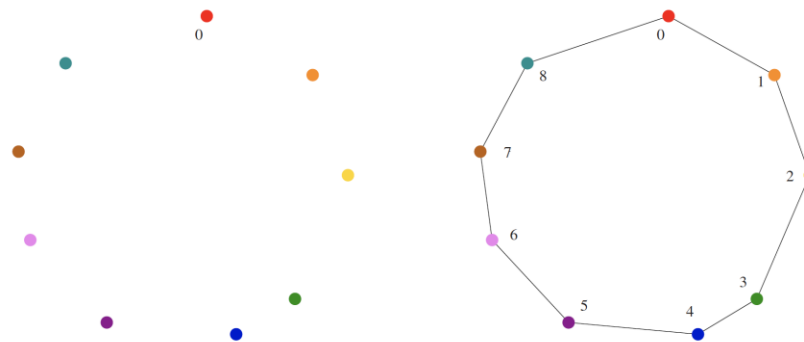
Algoritmus vs. heuristika

- Správnost algoritmu:
 - algoritmus musí vést ke správnému výsledku pro libovolný vstup
 - někdy je nalezení takového postupu problematické.
- Problém obchodního cestujícího (TSP - zjednodušeně):
 - Vstup: množina měst spojených silnicemi o daných délkách
 - Výstup: Nejkratší cesta procházející všemi městy a vracející se do výchozího města

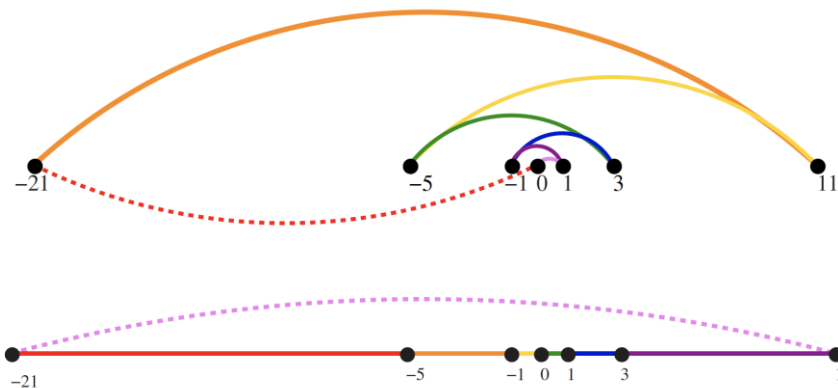
Algoritmus vs. Heuristika - TSP

□ Řešení č. 1: Použití nejbližšího souseda:

■ Vstup č. 1:



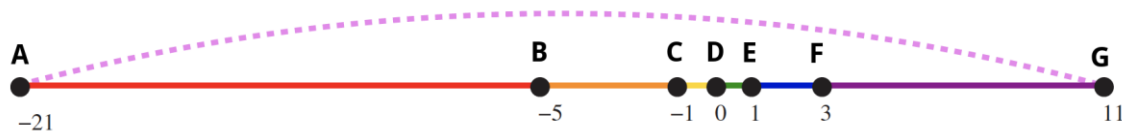
■ Vstup č. 2:



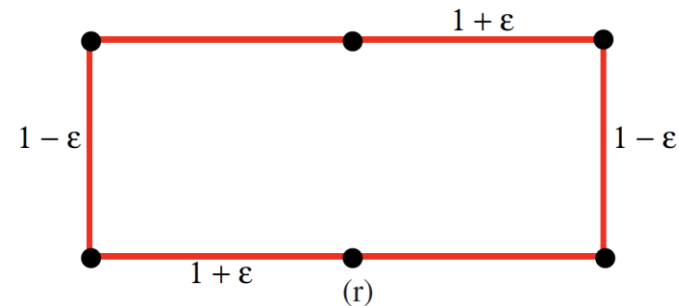
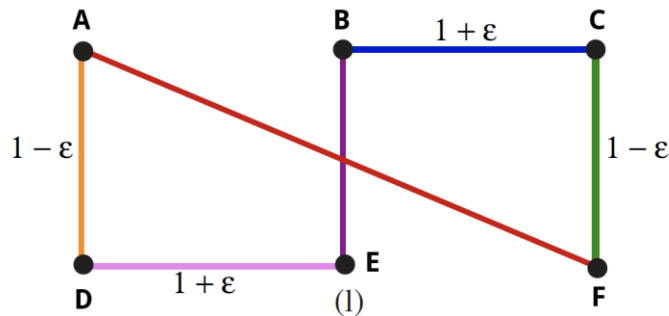
Algoritmus vs. Heuristika - TSP

□ Řešené č. 2: Spojování nejbližších párů:

- Udržujeme řetězce vrcholů a v každé iteraci spojíme 2 nejbližší řetězce.
- Vstup č. 1:



■ Vstup č. 2



Algoritmus vs. heuristika

- TSP – příklad problému pro jehož korektní vyřešení potřebujeme prověřit všechny možnosti – řešení hrubou silou – neúnosně zdlouhavé!
- **Heuristika:**
 - Postup, který **nedává vždy přesné řešení** problému.
 - Ve většině případů dává dostatečně přesné řešení v rozumném čase.
 - Nezaručuje nalezení přesného řešení.
 - Použijeme tehdy, pokud pro daný problém neexistuje přesný algoritmus, nebo jeho použití je neekonomické.

Jak popsat algoritmus?

- Přirozeným jazykem:
 - nejpřirozenější, ale nejméně přesný způsob.
- Pseudokódem:
 - zlatý střed
 - „*Programovací jazyk, který si nikdy nestěžuje na syntaktické chyby.*“
- Programovacím jazykem:
 - přesné, ale obtížné na zápis a pochopení.

Obsah

- Organizační informace
- Algoritmy a datové struktury – úvod
- Složitost algoritmů – základní pojmy
- Opakování – abstrakce, datové struktury, funkce, rekurze

Hodnocení algoritmů

- ❑ Potřebujeme zvolit **kritéria**, podle nichž budeme **hodnotit** jednotlivé algoritmy nezávisle na použitém programovacím jazyce nebo stroji.
- ❑ Nejčastější kritéria – **časové a paměťové nároky**
- ❑ Reálná doba běhu programu se může lišit pro různé stroje, různé sady vstupních dat atd.
- ❑ Využití tzv. časové a prostorové **složitosti algoritmů**:
 - způsob vyjádření vlastností algoritmu nezávisle na technických podrobnostech.
 - popis chování algoritmu pomocí jednoduchých matematických funkcí.

Časová složitost algoritmů

- ❑ Odvozena od počtu tzv. elementárních operací
- ❑ Elementární operace: sčítání, násobení, porovnání, skoky, atd.
- ❑ Časová složitost – řádový počet provedených operací v závislosti na velikosti vstupu (nejhorší případ)
- ❑ Velikost vstupu – příklady:
 - Počet prvků pole, které řadíme.
 - Větší z čísel, jejichž největšího společného dělitele počítáme.
- ❑ Určení elementárních operací – využití teoretických modelů strojů (např. Random Access Machine – RAM)

Časová složitost – příklad

- Algoritmy pro součty různých číselných řad
- Vstup algoritmů: číslo n

Algoritmus *Součet1*:

```
1. for i ← (1, n) :  
2.   for j ← (1, n) :  
3.     s ← s + j  
4. return s
```

Počet elementárních kroků: $c_1 \cdot n^2 + c_2$
Zjednodušeně: n^2

Algoritmus *Součet2*:

```
1. while n ≥ 1 :  
2.   s ← s + n  
3.   n ← n / 2  
4. return s
```

Počet elementárních kroků: $c_3 \cdot \log_2 n + c_2$
Zjednodušeně: $\log_2 n$

- Zásadní vliv má počet provedených součtů, příp. dělení.

Časová složitost

□ Určení časové složitosti:

1. Zhodnotíme, jak měřit velikost vstupu.
2. Určíme maximální možný počet elementárních kroků algoritmu provedených pro vstup o velikosti n .
3. Ve výsledné formuli ponecháme pouze nejrychleji rostoucí člen, ostatní zanedbáme.
4. Seškrtáme multiplikativní konstanty.

□ Průměrná časová složitost:

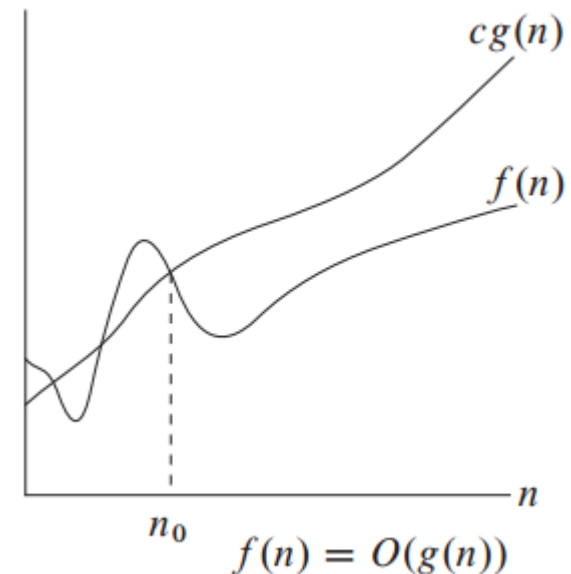
- Použijeme, pokud pro různé vstupy stejné velikosti vykoná algoritmus různý počet kroků.
- Aritmetický průměr časových nároků přes všechny vstupy dané velikosti.

Asymptotická časová složitost

- Nejčastěji používané hodnotící kritérium
- Chování algoritmu popíšeme porovnáním s jistou funkcí
- Pro n jdoucí k nekonečnu se chování algoritmu blíží této funkci
- Používají se tři různé složitosti:
 - O – Omikron (velké O , 'O, *big O*) – horní hranice chování
 - Ω – Omega – dolní hranice chování
 - Θ – Théta – třída chování

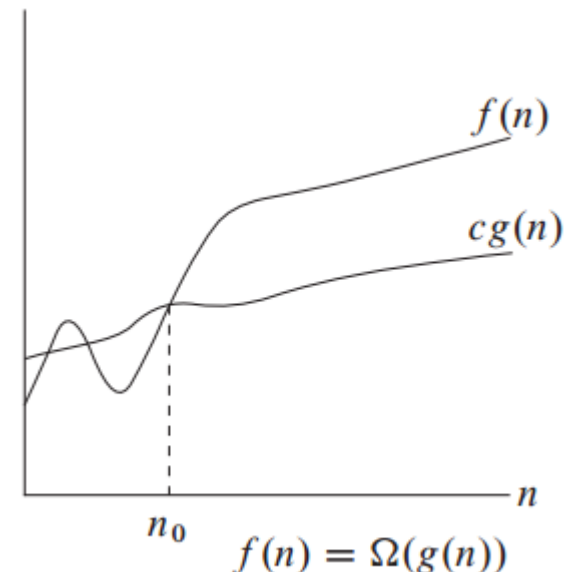
Složitost Omikron

- Složitost **Omikron** (nebo také Big O, 'O) vyjadřuje **horní hranici** časového chování algoritmu.
- **Omikron ($g(n)$)** označuje **množinu funkcí $f(n)$** , pro které platí:
 $\{f(n): \exists(c > 0, n_0 > 0) \text{ takové, že } \forall n \geq n_0 \text{ platí } [0 \leq f(n) \leq c \cdot g(n)]\}$
kde c a n_0 jsou určité vhodné kladné konstanty.
- Zápis $f(n) \in \text{Omikron}(g(n))$, nebo $O(g(n))$, označuje, že funkce $f(n)$ je **rostoucí maximálně tak rychle** jako funkce $g(n)$.
- Dostatečně velký násobek funkce $g(n)$ shora omezuje funkci $f(n)$ pro dostatečně velké n .



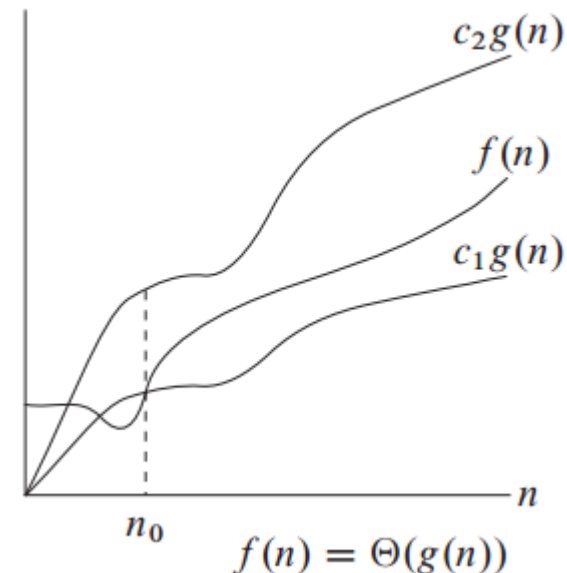
Složitost Omega

- Složitost **Omega** ($g(n)$) (nebo také Ω) vyjadřuje **dolní hranici** časového chování algoritmu.
- **Omega** ($g(n)$) označuje **množinu funkcí** $f(n)$, pro které platí:
 $\{f(n): \exists(c > 0, n_0 > 0) \text{ takové, že } \forall n \geq n_0 \text{ platí } [0 \leq c \cdot g(n) \leq f(n)]\}$
kde c a n_0 jsou určité vhodné kladné konstanty.
- Zápis $f(n) \in \Omega(g(n))$, nebo $\Omega(g(n))$, označuje, že funkce $f(n)$ je **rostoucí minimálně tak rychle** jako funkce $g(n)$.
- Funkce $g(n)$ je dolní hranicí množiny všech funkcí, určených zápisem $\Omega(g(n))$ nebo $\Omega(g(n))$.



Složitost Théta

- Složitost **Theta** ($g(n)$) (nebo $\Theta(g(n))$) vyjadřuje třídu chování algoritmu – ohraničuje funkci $f(n)$ z obou stran (časové chování shodné jako funkce $g(n)$.)
- **Theta** ($g(n)$) označuje množinu funkcí $f(n)$, pro které platí:
 $\{f(n): \exists(c_1 > 0, c_2 > 0, n_0 > 0) \text{ takové, že } \forall n \geq n_0 \text{ platí}$
 $[0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)]\}$, kde c_1 , c_2 a n_0
jsou určité vhodné kladné konstanty.
- Zápis $f(n) \in \text{Theta}(g(n))$, nebo $\Theta(g(n))$ označuje, že funkce $f(n)$ **roste tak rychle** jako funkce $g(n)$.
- Nejpresnější popis chování algoritmu – není možné ji vždy přesně stanovit



Vliv řádu algoritmu a kardinality úlohy

Počet prvků	$33n$	$46n \log n$	$13n^2$	$3.4n^3$	2^n
10	0,000 33 s	0,015 s	0,001 3 s	0,003 4 s	0,001 s
100	0,003 3 s	0,03 s	0,13 s	3,4 s	$4 \cdot 10^{14}$ stol.
1 000	0,033 s	0,45 s	13 s	94 hod	
10 000	0,33 s	6,1 s	22 min	39 dní	
100 000	3,3 s	1,3 min	1,5 dne	108 roků	
Maximální velikost n pro čas					
1 s	30 000	2 000	280	67	20
1 min	1 800 000	82 000	2 200	260	26

Vliv řádu a konstanty

	CRAY – 1 Fortran	TRS-80 Basic
Počet prvků	$3 n^3$	$19\,500\,000 n$
10	$3 \times 10^{-6} \text{ s}$	$200 \times 10^{-3} \text{ s}$
100	$3 \times 10^{-3} \text{ s}$	2 s
1 000	3 s	20 s
2 500	50 s	50 s
10 000	49 min	$3,2 \text{ min}$
1 000 000	95 let	$5,4 \text{ hod}$

Typické časové složitosti

- ❑ $\Theta(1)$ – označení algoritmů s konstantní časovou složitostí
- ❑ $\Theta(\log(n))$ – označení algoritmů s logaritmickou časovou složitostí:
 - Základ logaritmu není podstatný.
 - Např. rychlé vyhledávací algoritmy
- ❑ $\Theta(n)$ – označení algoritmů s lineární časovou složitostí
 - Např. běžné vyhledávací algoritmy
- ❑ $\Theta(n \cdot \log(n))$ – označení algoritmů nazvané **linearitmické**
 - Např. rychlé řadicí algoritmy
- ❑ $\Theta(n^2)$ – označení algoritmů s kvadratickou časovou složitostí
 - Algoritmy sestavené z dvojnásobného počítaného cyklu do n .
 - Např. jednoduché řadicí algoritmy (např. bubble-sort)

Typické časové složitosti

- $\Theta(n^3)$ – označení algoritmů s kubickou časovou složitostí
 - Algoritmy s touto složitostí jsou prakticky použitelné pouze pro málo rozsáhlé problémy.
 - Kdykoli se n zdvojnásobí, čas zpracování je osminásobný.

- $\Theta(k^n)$ – označení algoritmů s exponenciální časovou složitostí (k je přirozené číslo)
 - pro $k = 2$ – binomická časová složitost
 - Existuje několik prakticky použitelných algoritmů s touto složitostí.
 - Často se označují jako algoritmy pracující s „hrubou silou“ (angl. brute-force algorithms).
 - Kdykoli se n zdvojnásobí je čas řešení kvadrátem.

Prostorová složitost

- ❑ Měří paměťové nároky algoritmu
 - ❑ Kolik nejvíce elementárních paměťových buněk algoritmus použije.
 - ❑ Elementární paměťová buňka: proměnná typu integer, float, byte apod.
 - ❑ Vyjádříme ji jako funkci $f(n)$ v závislosti na velikosti vstupu n .
-

❑ Pomocná paměť:

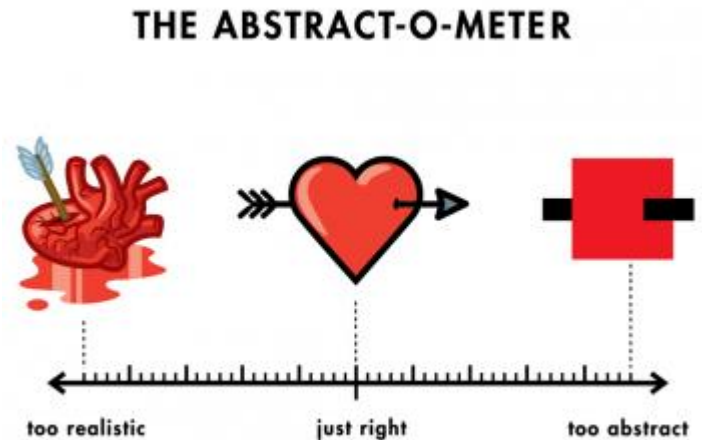
- extra paměť, kterou algoritmus využije, pokud nepočítáme velikost vstupu.
- umožní lépe rozlišit paměťové nároky algoritmů, jejichž prostorová složitost je stejná.

Obsah

- Organizační informace
- Algoritmy a datové struktury – úvod
- Složitost algoritmů – základní pojmy
- Opakování – abstrakce, datové struktury, funkce, rekurze

Abstrakce

- **Obecně:** myšlenkový proces zanedbávající odlišnosti a zvláštnosti a zjišťující obecné, podstatné vlastnosti a vztahy
- **V informatice:** odděluje účel entity od její implementace:
 - Zdůrazňujeme smysl entity
 - Potlačujeme způsob a detaily implementace (nepotřebujeme vědět, jak to funguje uvnitř)
 - Nástroj pro zvládnutí komplexnosti řešených problémů
 - Nástroje abstrakce: funkce, třídy, abstraktní datové typy, atd.
 - Často využíváme několik úrovní abstrakce



Datové typy a struktury

□ Datové typy:

- Jednoduché – bool, char, int, float, ...
- Strukturované – pole, **struktura**, unie, ...

□ Datové struktury – složeny z komponent jiného typu:

- Homogenní x heterogenní
- Statické x dynamické
- Komponentou strukturovaného typu může být jiný dříve definovaný strukturovaný typ – možnost tvorby hierarchických a rekurzivních struktur.
- **Průchod** – algoritmus, který postupně projde (a stejným způsobem zpracuje) všechny prvky homogenní datové struktury.

Statické proměnné (struktury)

- ❑ Dostávají jméno při deklaraci.
- ❑ Prostor jimi zaujímaný se vyhradí (alokuje) **v době překladu**.
- ❑ K obsahu těchto proměnných se dostáváme prostřednictvím jejich **jména**.
- ❑ Je-li struktura statická, **nemění** se za běhu programu ani počet ani uspořádání jejich komponent.
- ❑ Příkladem statické struktury je pole nebo záznam.

Dynamické proměnné (struktury)

- ❑ Vznikají i zanikají **v době běhu programu**.
- ❑ Nemohou mít jméno (identifikátor).
- ❑ K obsahu dynamických proměnných (struktur) se dostáváme **prostřednictvím ukazatele**.
- ❑ Počet i uspořádání komponent dynamických struktur se za běhu programu **mění**.
- ❑ Funkce `malloc()` a `free()`

Dynamické proměnné (struktury)

- ❑ K tvorbě dynamické struktury je vhodné (nutné) použít datového **typu struktura** (*záznam*). Její heterogennost umožňuje, aby dynamický prvek obsahoval vedle vlastní hodnoty také ukazatel(e).
- ❑ Příkladem dynamické struktury je seznam nebo strom.

Dynamické proměnné (struktury)

- Kdy je použijeme?
 - Pokud předem nevíme, kolik prvků daného typu budeme potřebovat uložit.
- Jak zajistit to, abychom měli ukazatel na každý vložený prvek?
 - Nemůžeme předem určit, kolik ukazatelů budeme potřebovat a „pojmenovat“ všechny ukazatele.
 - V každém nově vloženém prvku si zajistíme prostor pro uložení ukazatele na následníka (následníky).

Operace `malloc()`

- ❑ Vrací hodnotu „ukazující“ na paměťový prostor v paměťové oblasti vyhrazené pro tento účel (hromada, halda, angl. *heap*).
- ❑ Velikost prostoru je třeba určit parametrem této funkce, obvykle pomocí operátoru `sizeof()`.
- ❑ Vždy je potřeba otestovat návratovou hodnotu
- ❑ Vrácený ukazatel je vhodné přetypovat na ukazatel na datový typ, s nímž je ukazatel svázán.
- ❑ `NULL` je ukazatelová konstanta s hodnotou vyjadřující, že ukazatel neukazuje na žádnou proměnnou (strukturu). Tuto konstantu lze přiřadit ukazateli libovolného typu.

Operace `free(ptr)`

- ❑ Ruší použitelnost dynamické proměnné (struktury), na kterou ukazuje ukazatel `ptr`. Po této operaci je hodnota ukazatele **nedefinovaná**.
- ❑ Pokud se paměť, kterou zaujímá zrušená proměnná (struktura), vrátí do vyhrazené paměťové oblasti, říkáme, že DPP pracuje **s regenerací**. V jiném případě jde o mechanismus **bez regenerace** (v praxi sotva použitelné).
- ❑ S pamětí je potřeba dobře hospodařit:
 - Ke každé operaci `malloc()` patří i operace `free()`
 - Nesmíme ztratit ukazatel na dynamicky vytvořenou proměnnou!

Funkce

- ❑ Důležitý nástroj pro **zvyšování abstrakce**.
- ❑ Má formu uzavřeného podprogramu – v místě volání funkce se generuje skok do podprogramu, po provedení funkce se řízení vrací zpět za skok do podprogramu.
- ❑ Rozlišujeme deklaraci funkce, definici funkce a volání funkce
- ❑ Parametry funkce:
 - Předávání odkazem nebo hodnotou
 - Představují *interface*, kterým je funkce připojena k programu.
 - Pozor na vedlejší efekt.
- ❑ **Pozn:** procedura = funkce, která nevrací žádnou hodnotu.

Rekurze

- ❑ Metoda **definování** určitého **objektu pomocí sebe sama**
- ❑ Umožňuje definovat nekonečnou množinu objektů konečným popisem
- ❑ Rekurzivní struktura dat (lineární seznam)
- ❑ Rekurzivní struktura algoritmu
- ❑ Rekurzivní volání funkce – funkce je volána v těle sebe samé

