Základy algoritmizace, návrh algoritmu

Algoritmus

- **Předpoklady automatického výpočtu:**
  - předem stanovit (rozmyslet) přesný postup
  - během opakovaného provádění postupu již nepřemýšlet a postupovat mechanicky

- **Přesná definice algoritmu**
  - vyžaduje sestavit matematický model počítače (výpočtu)
  - konečný automat, zásobníkový automat, Turingův stroj, …

- **„Naivní“ (nepřesná) definice algoritmu**

  - **Algoritmus je konečná posloupnost elementárních kroků (příkazů), organizovaná tak, že po provedení každého kroku je jednoznačně určeno, jaký krok bude následovat nebo zda výpočet končí. Jeho provedení umožní pro každá přípustná data mechanickým způsobem a po konečném počtu kroků získat příslušná výstupní data.**

Vlastnosti algoritmu

- Při návrhu je třeba dbát na to, aby výsledný **algoritmus** byl:
  - **mechanický** – v průběhu jeho provádění již není nutné se vracet k porozumění dané úloze, to je zapotřebí pouze při návrhu algoritmu, jeho provedení lze svěřit i stroji.
  - **deterministický** – po provedení každého kroku musí být jasné, co má následovat, tj. zda výpočet končí nebo jaký krok se provede jako další.
  - **hromadný** – algoritmus neslouží k řešení jediné osamocené úlohy, ale celé třídy úloh, které jsou si navzájem podobné.
  - **konečný** – musí být zajištěno, že pro přístupná data výpočet po provedení konečného počtu kroku vždy jednou skončí.
  - **správný** – požadavek, aby algoritmus vytvořil pro všechna přípustná data příslušná výstupní data, tj. data, která lze považovat za adekvátní řešení daného problému. Správnost znamená, že pokud vstupní data splňují zadanou podmínku na vstup, musí výstupní data spolu se vstupními splňovat výstupní podmínku.

$$(P(\text{Vstup}) = \text{True}) \Rightarrow (Q(Vstup, Výstup)) = \text{True})$$

Použití algoritmů - některé důležité otázky a odpovědi

- **Lze každý problém řešit algoritmem?**

NE

  - existují principiálně algoritmicky neřešitelné problémy
  - u jiných problémů algoritmus neznáme

- **Lze o daném postupu vždy rozhodnout, zda je algoritmem?**

  NE

  - V některých případech neumíme ověřit, zda je postup resultativní (konečný). Konečnost nelze ověřit algoritmicky

- **Pokud algoritmus známe, je již „vyhráno“?**

  NE

  - Algoritmus může vyžadovat tak velké množství kroků, že i na velmi rychlém počítači je nereálné jej provést v přijatelném čase (**časová složitost**)
  - Algoritmus může vyžadovat neúnosně mnoho paměťových míst (**prostorová složitost**)
  - Algoritmus může pracovat se zaokrouhlovacími chybami tak „nešťastně“, že pohltí platné cifry (**numerická nestabilita**)

Prostorová a časová složitost

- **Účinnost (efektivnost) programu** je měřená skutečným časem výpočtu programu a počtem bytů paměti potřebných k provedení programu
  - Závislá na konkrétním HW a SW prostředí
  - Analýzou lze určit složitost nezávisle na konkrétní implementaci (zjednodušený model procesoru). Jde však často o komplikovaný výpočet
  - Většinou nás pouze zajímá, **jak rychle roste složitost v závislosti na rozsahu řešeného problému** – **asymptotická složitost**.
    - Někdy je podstatné znát **složitost v nejhorším možném případě** – pesimistický odhad, například při řízení časově kritických procesů
    - Jindy je důležitější znát **průměrnou složitost** – u náročných výpočtů, které se často opakují.

- **Asymptotickou složitost** vyjadřujeme řádem růstu skutečné časové či prostorové složitosti. Tedy porovnáním s nějakou funkcí o jejímž růstu máme určitou představu.
  - **Říkáme, že časová složitost roste maximálně (minimálně) tak rychle, jako funkce *f(n)*,** jestliže existuje konstanta C tak, že pro dostatečně velké n je

    $$T(n) \leq f(n), \quad (T(n) \geq f(n)),$$

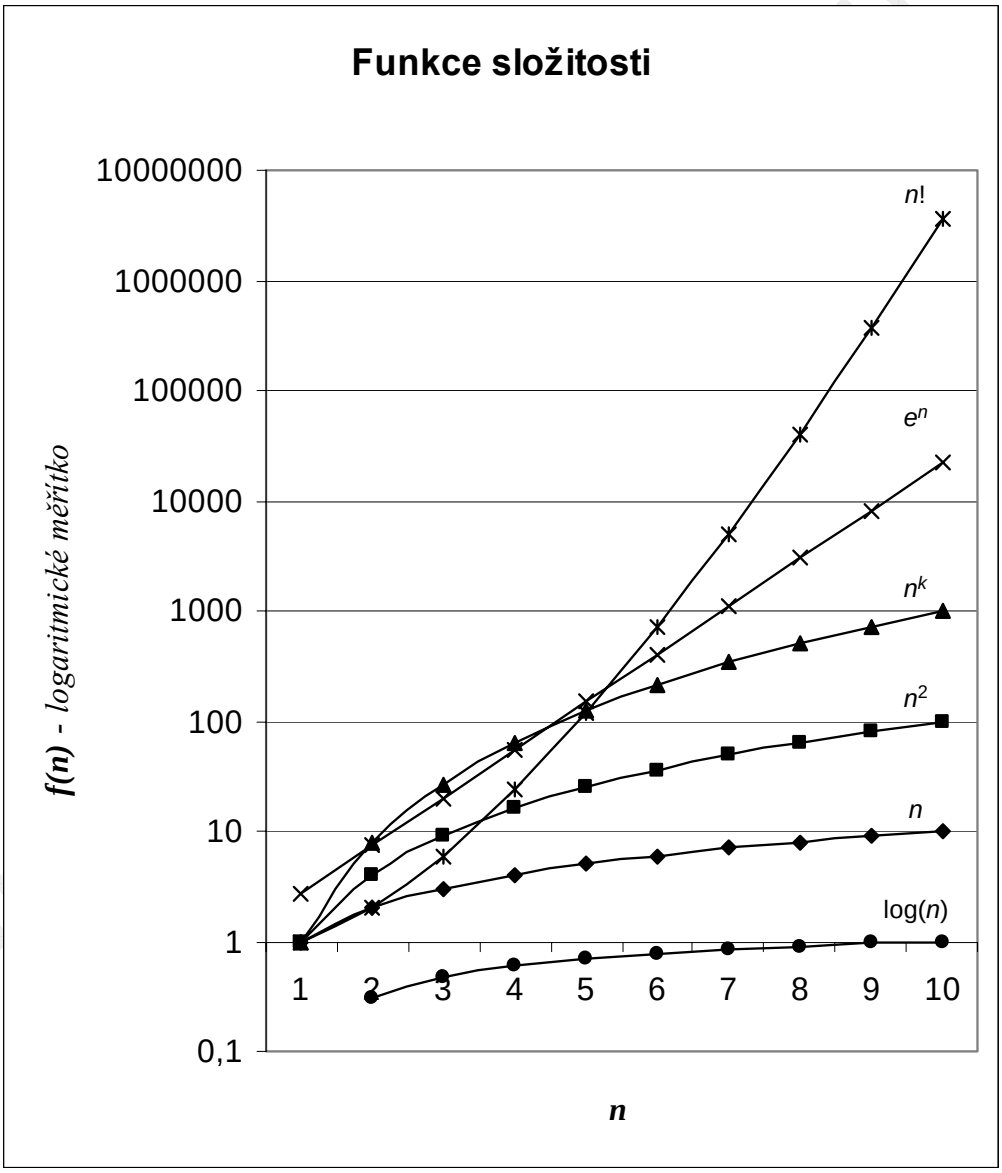
    - kde n je rozměr vstupních dat a T čas potřebný k realizaci výpočtu

  - **Nejčastěji** se pro porovnání užívají tyto **funkce**:
    - $f(n) \approx 1$ - jednotková složitost
    - $f(n) \approx \log n$ - logaritmická složitost
    - $f(n) \approx n \cdot \log n$ - linearitmická složitost
    - $f(n) \approx n$ - lineární složitost
    - $f(n) \approx n^2$ - kvadratická složitost
    - $f(n) \approx n^3$ - kubická složitost
    - $f(n) \approx n^k$ - obecně polynomická složitost
    - $f(n) \approx 2^n$ - exponenciální složitost
    - $f(n) \approx n!$, $f(n) \approx n^n$ - faktoriální složitost

Porovnání růstu vybraných funkcí

$f(n)$ / n	1	2	10	20
log(*n*)	0	0,30	1	1,30
n	1	2	10	20
n^2	1	4	100	400
e^n	3	7	22 026	485 165 195
***n*!**	1	2	3 628 800	2 432 902 008 176 640 000

Průběh vybraných funkcí



- **Rozsah řešeného problému za 1 min. x 1 hod.**

$f(n)$	rozsah za 1 min.	rozsah za 1 hod.
n	n_1	$n_2 = 60 \cdot n_1$
$n \cdot \log n$	n_1	$n_2 \cong 60 \cdot n_1$
n^2	n_1	$n_2 \cong 8 \cdot n_1$
2^n	n_1	$n_2 \cong n_1 + 6$
$n!$	n_1	$n_2 = n_1 + \ln 60 / \ln (n_1 - 1)$

Prostorová (paměťová) složitost

- **Paměťová složitost** je dána paměťovými nároky na reprezentaci zpracovávaných dat – nikoliv délkou programu
  - obvykle pouze **asymptotický odhad** řádu růstu v závislosti na rozsahu řešeného problému

Časová složitost

- **Výpočet časové složitosti bývá náročná úloha**
  - zejména výpočet průměrné složitosti – je třeba odhadnout pravděpodobnost každé permutace vstupních dat

- **Obvykle se zjišťuje asymptotická složitost pro nejhorší možný případ**
  - neuvažuje se v časových jednotkách, ale v počtu nutných operací – **operační složitost**

- **Operační složitost programů v jazyce Pascal**
  - Operace **přiřazení**, **porovnání**, **aritmetické operace**, operace **vstupu/výstupu**, **přístupu k prvku** pole a složce záznamu mají **složitost 1**.

  - **Podmíněný příkaz** `if P then V1 else V2`
    Složitost je dána **součtem** složitosti podmínky **P** a **maximem** ze složitostí větví **V1** a **V2**.

  - **Cyklus (opakování)** `for I:=1 to n do B`
    Složitost je ***n* krát** složitost bloku **B**

    ```
    while P do B
    repeat B until P
    ```

    Složitost=(složitost **P** + složitost **B**)**.počet opakování**

Jak ověřit algoritmus?

- **Důkaz korektnosti:**
  - **Parciální (částečná) korektnost:**
    - Pokud vstupy splňují nějakou podmínku, musí vstupy spolu s výstupy splňovat nějakou podmínku.

      $(P(\text{Vstup}) = \text{True}) \Rightarrow (Q(Vstup, Výstup)) = \text{True})$.

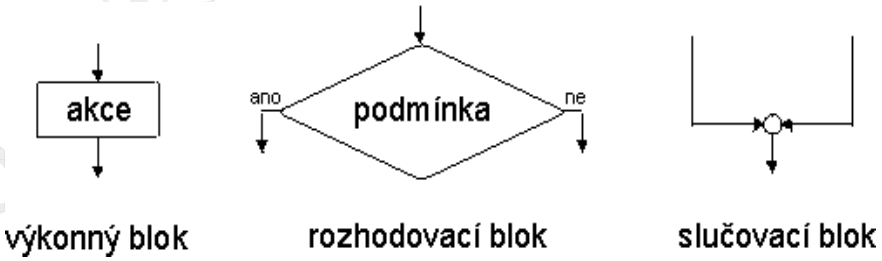
      Zachování této podmínky se sleduje krok po kroku během provádění algoritmu
  - **Resultativnost:**
    - Ověří se, že postup někdy skončí. Obecný návod k ověření neexistuje.

- **Testování na zkušebních datech**
  - **Ověří vždy pouze nesprávnost algoritmu** (pro vyvrácení správnosti stačí jediný protipříklad)
  - **Nikdy nelze testováním na zkušebních datech plně prověřit správnost.** Vždy lze jen snížit pravděpodobnost chyb
  - **Testovací data je třeba volit vhodně** (projít všechny cesty v algoritmu aspoň jednou verzí vstupních dat, brát hodnoty blízké mezi přípustnosti, ověřit reakci na nepřípustná data …)

Grafické znázornění algoritmu

VÝVOJOVÉ DIAGRAMY

- **Vývojové diagramy** se někdy též označují jako **bloková schémata**, **grafy řízení**
  - univerzální, avšak ne bezproblémový prostředek
  - znázorňují časový postup provádění kroků algoritmu pomocí orientovaného grafu
    - s **jediným počátečním uzlem** (začátek postupu)
    - s **jediným koncovým uzlem** (konec postupu).
    - jednotlivé uzly (bloky) jsou spojeny šipkami (orientované hrany grafu)



- **Bloky vývojového diagramu:**
  - **Výkonné bloky** - po daném kroku následuje vždy jednoznačně určený (následující) krok. Výsledkem kroku je obvykle změna obsahu paměti nebo akce na vnějším zařízení.
  - **Rozhodovací bloky** - po daném kroku mohou následovat dva různé bloky v závislosti na splnění či nesplnění určité podmínky. V těchto blocích se vyhodnotí splnění, či nesplnění podmínky.
  - **slučovací bloky** - slouží ke sloučení dvou šipek znázorňujících tok řízení do jediné. Neprovádí se v nich žádná akce ani rozhodování.

- **Typy uzlů ve vývojových diagramech**

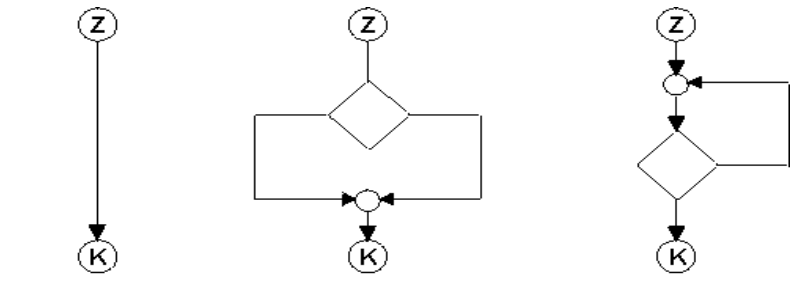
TYP UZLU	graf	počet	vstupy	výstupy
Počátek	Z	1	0	1
Konec	K	1	1	0
Výkonný		*n*	1	1
Rozhodovací (predikátový)		*m*	1	2
Slučovací		*m*	2	1

- **Pravidla pro vývojové diagramy**
  - Do **výkonného bloku** vede vždy jediná šipka a jediná z něj vychází
  - Do **rozhodovacího bloku** vede jediná šipka a vycházejí z něj dvě šipky
  - Do **slučovacího bloku** vedou dvě šipky a pouze jediná vychází
  - Existuje **počáteční blok**, z kterého vychází jedna šipka a žádná do něj nevede
  - Existuje **koncový blok**, do kterého vede jediná šipka a žádná z něj nevychází
  - Musí být splněna podmínka, že každý uzel leží na nějaké orientované cestě od počátečního uzlu do koncového uzlu. Každý uzel musí být dosažitelný.

- **Nejjednodušší jsou ty konstrukce, které žádný rozhodovací blok neobsahují a ty, které obsahují pouze jediný rozhodovací blok.**
  - Ukazuje se, že s těmito konstrukcemi **lze již vystačit**, pokud nebudeme algoritmus navrhovat „najednou“ v detailech, ale **postupným rozkladem** složitého úkolu na jednodušší, těch opět na jednodušší, … v řadě úrovní, až dospějeme k elementárním krokům algoritmu. Při těchto rozkladech **můžeme užívat pouze jednoduché základní konstrukce.**

Základní konstrukce

- **Sekvence**
  - obsahuje pouze výkonné bloky, seřazené za sebou
- **Selekce** (*výběr alternativ*)
  - obsahuje rozhodovací blok a dvě výkonné větve ukončené slučovacím bodem
- **Iterace** (*cyklus*)
  - nejprve slučovací blok, poté rozhodovací blok, z něj se jedna šipka vrací ke slučovacímu bloku a druhá pokračuje



- Každý uzel musí ležet aspoň na jedné cestě od počátečního uzlu k uzlu koncovému (musí být dosažitelný)

- Vývojové diagramy lze sestavit i velmi komplikované a mimořádně obtížně srozumitelné. **To je jejich hlavní nebezpečí a nevýhoda!!**

Strukturovaný návrh algoritmu

- **Důvody** co nejjednoduššího návrhu a realizace algoritmu:
  - **Méně chyb**
  - **Snadná srozumitelnost** pro ostatní a tím i snadné provádění změn (údržby z důvodu oprav chyb, změny prostředí, rozšiřování funkcí)
  - Obvykle i **rychlejší** a **méně nároků** na paměť

- Preference **strukturovaného návrhu algoritmu – principy:**
  - **Hierarchický rozklad** složité úlohy na jednodušší (**postup shora dolů** – „**top-down**“) – v případě potřeby v několika – mnoha úrovních
  - Na všech úrovních rozkladu je možné **použít pouze základní struktury**:
    - **sekvence**
    - **selekce** (výběr)
    - **iterace** (opakování = cyklus)

Jiné způsoby rozkladu složité úlohy na jednodušší přípustné nejsou!!!

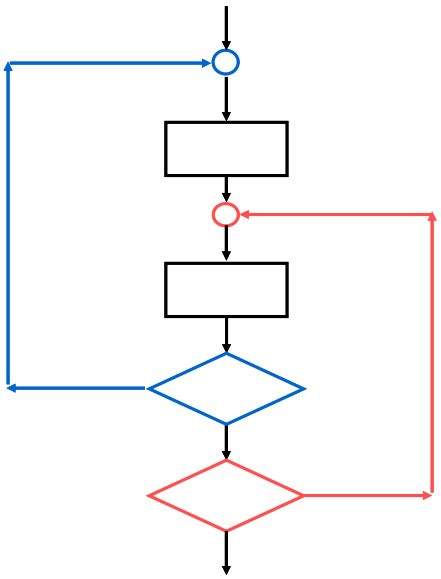
- **Lze každý problém, který umíme řešit nějakým algoritmem, řešit i strukturovaným algoritmem?**

ANO

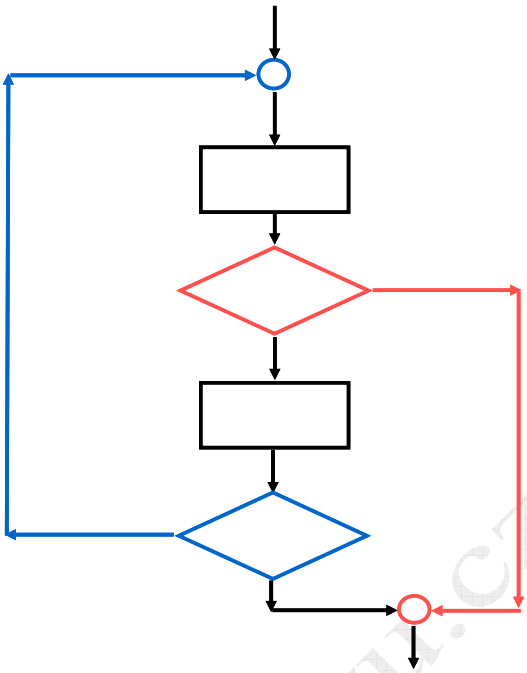
  - Dokonce se nic podstatného neztratí. Lze dokázat, že toho lze vždy dosáhnout přidáním jediné pomocné proměnné.

- **Strukturovaný algoritmus je přehlednější, jednodušší a obvykle i efektivnější.** Zkušený a odpovědný programátor navrhuje algoritmy a programy strukturovaně.

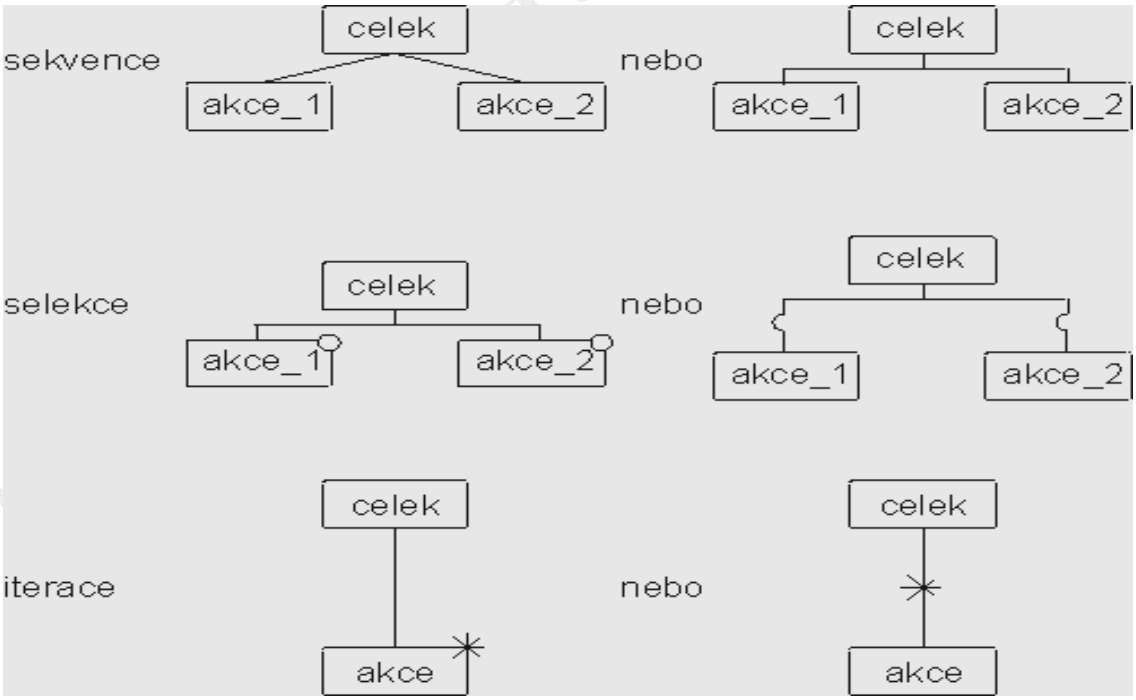
- **Příklad nestrukturovaného návrhu**



- **Jediná výjimka, jejíž povolení stojí za úvahu - předčasný výskok z cyklu**
  - Tak zvané „dobře navržené programy“ nebo BJ-programy tvoří širší třídu než strukturované algoritmy.



STRUKTUROGRAMY



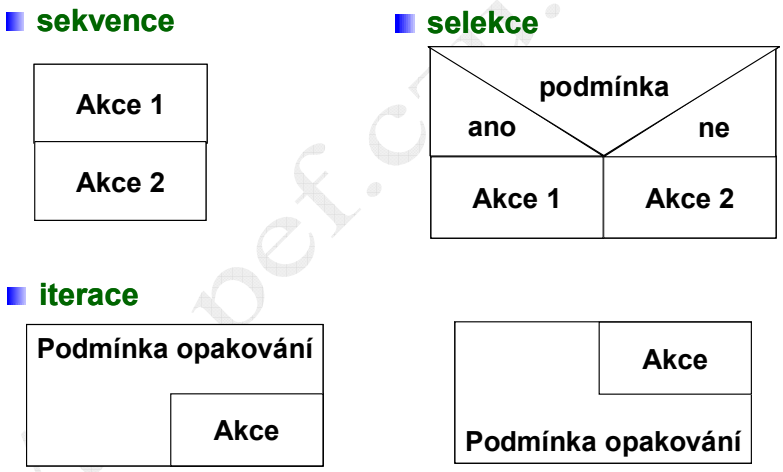
Výhody a nevýhody strukturogramů

- **Výhody:**
  - nelze sestavit nestrukturovaný algoritmus,
  - lze postupně zjemňovat návrh,
  - lze pojmenovávat celky,
  - lze se vždy soustředit na jednoduchou úvahu (jednu úroveň),
  - algoritmus lze odvozovat ze struktury dat

- **Nevýhody:**
  - způsob zakreslování není normalizován,
  - spojnice nezaznamenají časovou následnost, ale rozklad úlohy (to může být nenázorné a nezvyklé)

PLOŠNÉ STRUKTUROGRAMY

- Snaha zachovat výhody a minimalizovat nevýhody strukturogramů vede k **plošným strukturogramům**



- Plošné strukturogramy jsou velmi **názorný publikační nástroj**
  - **Časová následnost akcí** je v nich v porovnání se „stromovými“ strukturogram potlačena méně. Zpravidla stačí sledovat směr shora dolů.
  - Umožňují používat **pouze strukturované konstrukce**. Nebezpečí vzniku algoritmu, který není strukturovaný, nehrozí.

- **Částečnou nevýhodou** může být to, že
  - není kam zapsat **názvy celků**, což může vést k menší srozumitelnosti
  - při jejich konstrukci metodou „shora-dolů“ bývá obtížné správně o**dhadnout velikost ploch**, které si v obrázku vyhradíme

Ukázka plošného strukturogramu